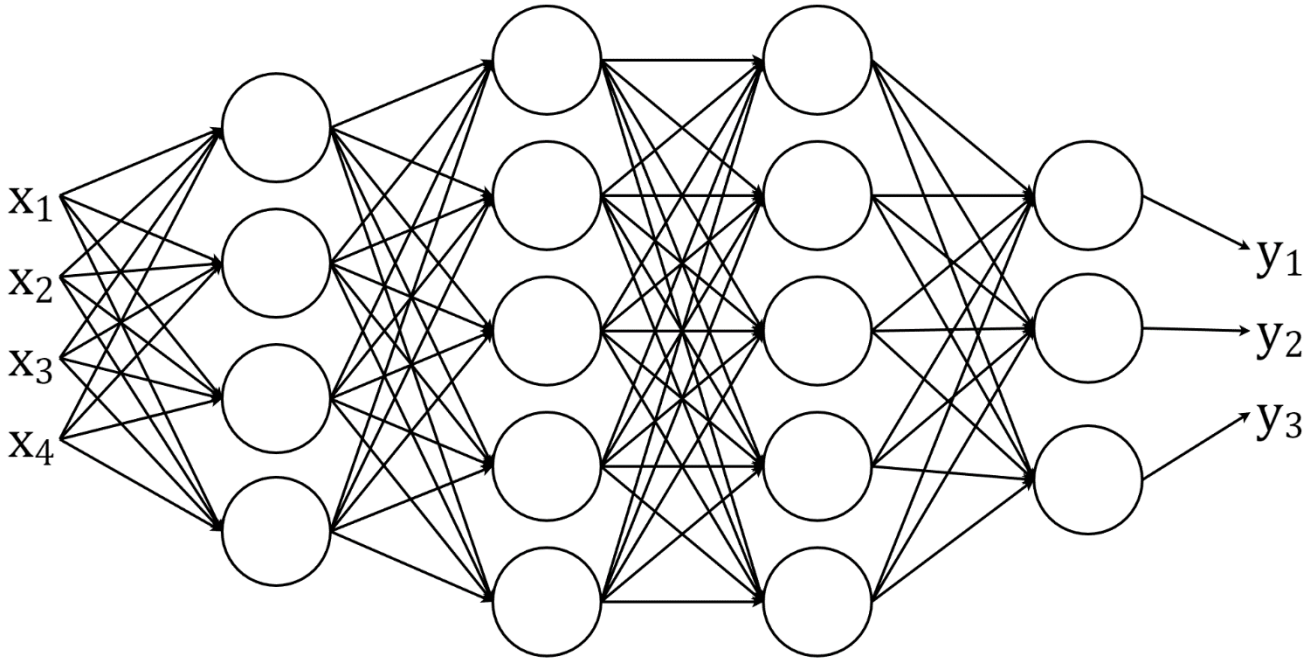
Wie funktioniert Deep Learning?

NEURONALE NETZE ZUR KLASSIFIKATION VON HANDGESCHRIEBEN ZIFFERN



Reto Merz, 4a
betreut durch Andreas Nüesch

Gymnasium Oberwil, September 2018

Inhaltsverzeichnis

1 Vorwort

1.1 Themenwahl

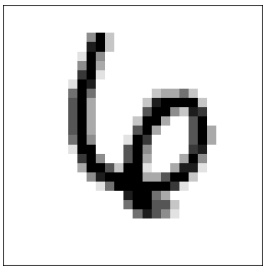
Künstliche Intelligenz fasziniert mich schon seit mehreren Jahren. Deep Learning, ein Teilgebiet der künstlichen Intelligenz, ist zurzeit der vielversprechendste Ansatz, ein Programm durch Erfahrung lernen zu lassen und spielt bereits im alltäglichen Leben eine Rolle. So wird Deep Learning bereits zur Spracherkennung, zum maschinellen Sehen, sowie zur maschinellen Übersetzung eingesetzt und könnte in nächster Zukunft selbstfahrende Autos oder autonome virtuelle Assistenten ermöglichen.

Neben der Aktualität dieser Form von künstlicher Intelligenz habe ich mich aufgrund meines Interesses an der Mathematik und den Algorithmen hinter Deep Learning für dieses Thema entschieden. Zudem begeistert mich, dass die Implementierung dieser Algorithmen relativ simpel ist und dennoch beeindruckende Resultate liefert.

1.2 Danksagung

Danken möchte ich meiner Mutter Dorothea Merz für das Lektorat der Arbeit und meiner Betreuungslehrperson, Andreas Nüesch, für die Freiheit, die mir beim Arbeiten gelassen wurde.

2 Einleitung



Das Ziel der Arbeit ist es, ein Programm zu schreiben, welches Bilder von handgeschriebenen Ziffern korrekt nach den darauf zu sehenden Ziffern einteilt. Das hört sich zwar einfach an (jeder erkennt auf dem Bild links eine Sechs), ist aber komplizierter, als man denken könnte. Was für einen Menschen trivial ist, kann teilweise sehr schwierig zu automatisieren sein. Das Einteilen von Bildern nach Ziffern lässt sich kaum auf klare Regeln herunterbrechen: Niemand schreibt die gleiche Ziffer jedes Mal exakt gleich, und jeder schreibt Ziffern ein wenig anders (z.B. die Sieben mit oder ohne Querstrich, oder die Neun mit einem mehr oder weniger geraden unteren Bogen). Es müsste jede noch so kleine Abweichung berücksichtigt werden.

Solche Probleme, welche einfach für Menschen, aber schwierig für Maschinen sind, lassen sich teilweise mit Machine Learning (zu Deutsch *maschinelles Lernen*) lösen. Machine Learning umfasst alles, was Programme befähigt, von Erfahrung zu lernen. In dieser Arbeit werden Modelle von neuronalen Netzen mithilfe vieler Beispiele darauf trainiert, Ziffern auf Bildern wie oben zu erkennen.

Die Arbeit ist in drei grobe Teile gegliedert:

- Im ersten Teil wird erklärt, wie und weshalb Deep Learning funktioniert (alles was Machine Learning und künstliche neuronale Netze betrifft, gehört zu Deep Learning). Es wird darauf eingegangen, was ein künstliches neuronales Netz ist und wie man es trainieren kann, Klassifikationsaufgaben zu lösen.
- Im zweiten Teil ist ein Programm dokumentiert, welches Deep Learning in Python 3 implementiert.
- Im dritten Teil wird das Programm verwendet, um die Effektivität verschiedener Funktionen zu testen, welche die Informationsverarbeitung innerhalb eines künstlichen neuronalen Netzes (Aktivierungsfunktionen) steuern und welche beim Bewerten von Netzen (Kostenfunktion) eine Rolle spielen.

3 Machine- und Deep Learning

In diesem Teil wird die Theorie hinter modellbasiertem maschinellen Lernen und Deep Learning dargelegt.

Begriffe sind fett markiert und im Begriffsverzeichnis auf S. 62 aufgelistet. Nummerierte Formeln befinden sich neben dem Text im Formelverzeichnis auf S. 64.

3.1 Modellbasiertes maschinelles Lernen

Im Zentrum des modellbasierten maschinellen Lernens stehen **Modelle**, welche **Vorhersagen** (engl.: *predictions*) zu einem gegebenen Input erstellen. Beim Trainieren (oder «Lernen») findet das Modell Zusammenhänge zwischen Beispielinputs und zugehörigen erwarteten Outputs. Nach erfolgreichem Training stimmen die Modellvorhersagen fast vollständig mit den erwarteten Outputs überein. Das Modell soll dabei auch auf Beispiele ausserhalb der Trainingsbeispiele **generalisieren** (d.h. dass die Modellvorhersagen ebenfalls bei zuvor ungesehenen Inputs möglichst genau sind).

3.1.1 Grundbegriffe

Daten

Ein **Trainingsdatensatz** (engl.: *training dataset*) enthält die Beispiele, mit denen ein Modell trainiert wird. **Attribute** sind die Typen von Daten, anhand derer eine Vorhersage getroffen werden soll. Die **Features** (engl. für Eigenschaft oder Merkmal) sind die Werte, die für die Attribute eingesetzt werden und die **Labels** (engl. für Etiketten oder Aufschriften) das korrekte Resultat, welches bei der Vorhersage vom einem optimalen Modell nach dem Training erwartet wird.

In dieser Arbeit verwende ich den MNIST-Datensatz. Er beinhaltet 70'000 Bilder von handgeschriebenen Ziffern mit zugehöriger Angabe der Ziffer, um die es sich auf einem Bild handelt. Die Bilder sind graustufig und haben eine Grösse von 28x28 Pixeln. Eine graustufige Farbe kann mit einer Zahl zwischen 0 und 1 beschrieben werden, wobei die 0 für einen komplett weissen Pixel und 1 für einen komplett schwarzen Pixel steht. Die Attribute sind die Helligkeitswerte der 784 Pixel. Jedes Beispiel enthält zu jedem Pixel einen konkreten Helligkeitswert, das sind die Features. Die Angabe der Ziffer, die auf einem Bild zu sehen ist, ist das Label.

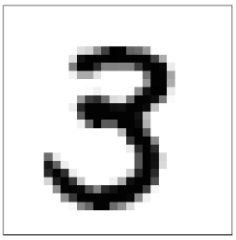


Abbildung 1: Bild der Ziffer 3 aus dem MNIST-Datensatz

Um zu bestimmen, wie gut ein Modell generalisiert, wird die Genauigkeit des Modells mit einem **Testdatensatz** ermittelt, der andere Beispiele als der Trainingsdatensatz beinhaltet. (Géron, Supervised learning, 2017)

Modellbegriff

Ein Modell ist im Grunde eine mathematische Funktion, welche die Features als Input eindeutig auf eine Vorhersage abgebildet.

Wenn ein Modell einen Wert vorhersagen soll, spricht man von einem **Regression**smodell. Die Geradengleichung ist z.B. ein Regressionsmodell, welches bei linearen Beziehungen zwischen Features und Labels sinnvoll ist:

$$y = \theta_1 x + \theta_0$$

x ist ein Feature, θ_0 und θ_1 sind die **Modellparameter** und y die Vorhersage. Je nach Modellparameter sind die Vorhersagen eines Modells zu einem Datensatz besser oder schlechter. Das Modell zu **trainieren** bedeutet, dessen Parameter von Startwerten ausgehend schrittweise so zu verändern, dass die Modellvorhersagen zum Trainingsdatensatz in jedem Schritt besser mit den Labels übereinstimmen.[1]

Modelle sind nicht auf einen Input und einen Output beschränkt. Bei höherdimensionalen Modellen werden die Inputs und Outputs in jeweils einem Vektor zusammengefasst (Features $x = (x_1 \quad x_2 \quad \cdots)$, Vorhersage $y = (y_1 \quad y_2 \quad \cdots)$). Gleiches gilt für das Label.

Parameter, die nicht erlernt werden und somit vor dem Training festgelegt werden müssen, nennt man **Hyperparameter**. Dazu gehören z.B. Parameter, die den Lernalgorithmus beeinflussen. (Géron, Model-based learning, 2017)

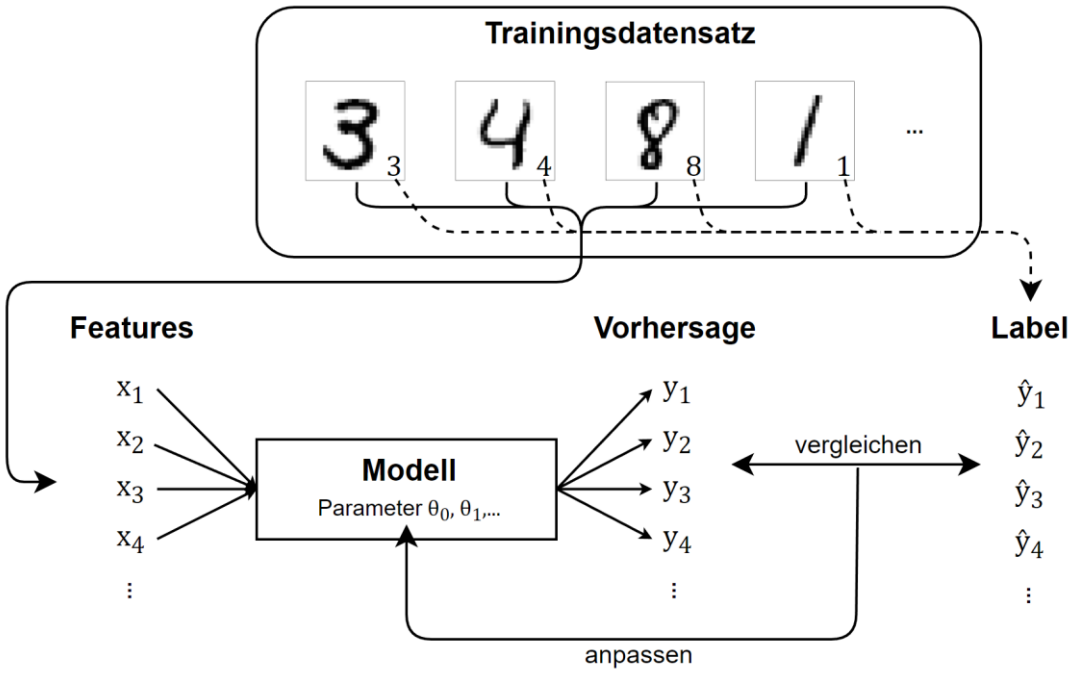


Abbildung 2: Schema des modellbasierten maschinellen Lernens

Im folgenden Beispiel soll die Note eines Schülers mit dem Attribut der Prüfungsvorbereitungszeit vorhergesagt werden. Es gibt keine Noten unter der 1 und keine über der 6, und höhere Prüfungsvorbereitungszeit sollte zu einer höheren vorhergesagten Note führen. Als Modell eignet sich folgende logistische Funktion, welche beide Anforderungen erfüllt:

$$y = \frac{5}{1 + e^{-(\theta_1 x + \theta_0)}} + 1$$

In der folgenden Abbildung ist der fiktive Trainingsdatensatz durch die blauen Punkte repräsentiert. Die x-Komponente der Punkte sind Features und die y-Komponenten die

[1] Bei diesem linearen Modell kann man die optimalen Werte für θ_0 und θ_1 exakt bestimmen, deshalb verwendet man bei diesem Modell üblicherweise keinen iterativen Trainingsprozess. Bei komplizierten Modellen, bei denen optimale Modellparameter nicht exakt gefunden werden können, ist ein solcher Prozess aber notwendig.

zugehörigen Labels. Die drei Kurven sind logistische Modelle (nach der obigen Gleichung) mit unterschiedlichen Parametern, die unterschiedlich gute Vorhersagen bezüglich des Trainingsdatensatzes liefern.

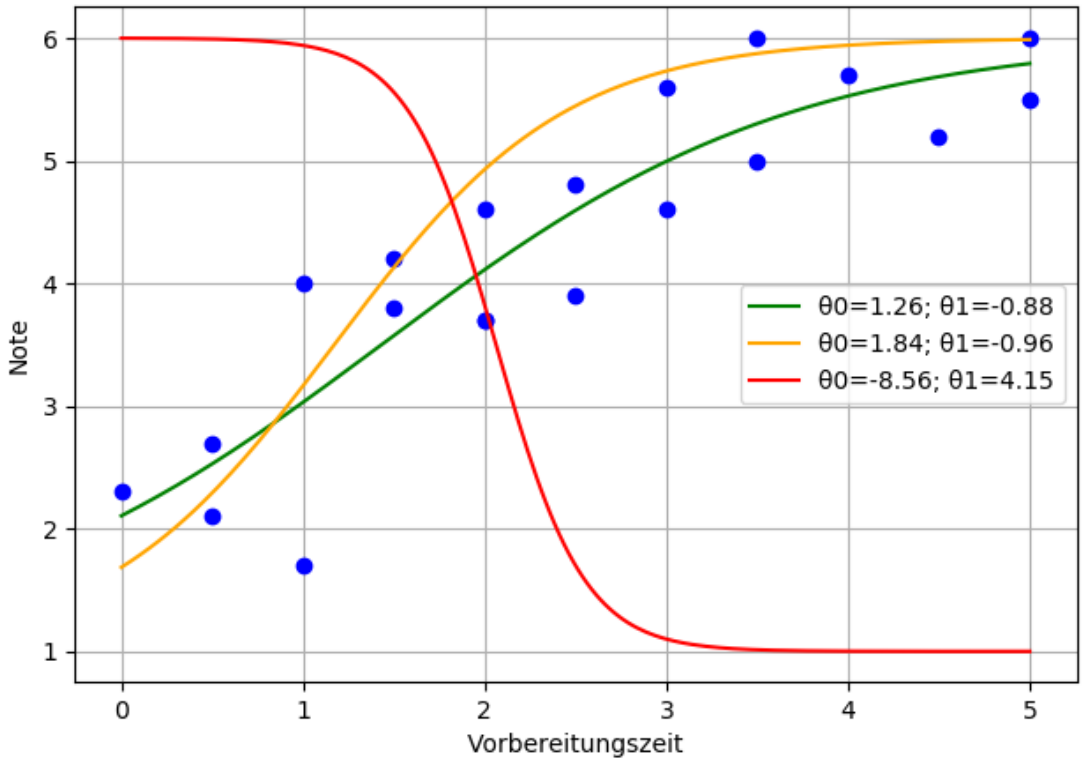


Abbildung 3: Logistische Modelle im Vergleich; je nach Modellparameter macht ein Modell bessere oder schlechtere Vorhersagen.

Klassifikation

Modelle können neben der Regression weitere Aufgaben lösen. In dieser Arbeit geht es hauptsächlich um **Klassifikation**, also das Einteilen von Features in bestimmte Kategorien, sogenannte **Klassen**. Das Einteilen von MNIST-Bildern nach den darauf sichtbaren Ziffern gehört zu diesem Aufgabentyp.

Klassifikation und Regression funktionieren relativ ähnlich. Ein Klassifikationsmodell gibt vorerst wie ein Regressionsmodell Zahlen aus. Im Gegensatz zu einem Regressionsmodell, in dem diese Zahlen als solche verstanden werden sollen, werden sie bei Klassifikationsmodellen interpretiert und einer Klasse zugewiesen. Beim Trainieren von Klassifikationsmodellen werden die Modellparameter nach den Outputzahlen angepasst, aus denen die Klasse bestimmt wird. Ein Klassifikationslabel besteht also auch aus Zahlen.

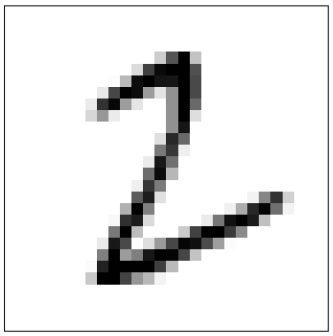
Möchte man Daten in mehr als zwei Klassen einteilen, so muss das Modell so viele Zahlen ausgeben, wie es Klassen gibt. Jeder der Klassen wird einer der Outputs des Modells zugeteilt. Nach dem Training soll das Modell an der Stelle, die zur korrekten Klasse gehört, eine 1 ausgeben. Bei allen anderen Outputs soll das Modell eine 0 ausgeben. Wenn man sich den Output des Modells als Vektor denkt, dann sollte der Outputvektor an einer Stelle eine 1 haben und an allen anderen eine 0. Die Labels bei Klassifikationen sind sogenannte **«One-Hot»-Vektoren**, da nur ein Element des Vektors «heiss», d.h. 1 ist.

Beim MNIST-Datensatz gibt es zehn Klassen, nämlich die Ziffern von 0 bis 9. Somit muss ein MNIST-Modell zehn Zahlen ausgeben. Sinnvollerweise wird dem ersten Output die Ziffer

0, dem zweiten die Ziffer 1 usw. zugeordnet. Der Labelvektor $\hat{y}$ des MNIST-Datensatz für eine 2 ist also beispielsweise wie folgt:

$$\hat{y}_{\text{Ziffer 2}} = (0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0)$$

Eine Vorhersage einer 2 wird aber nur in den seltensten Fällen exakt mit dem Label übereinstimmen. In der folgenden Abbildung sieht man das Bild einer handgeschriebenen Zwei mit einer möglichen Vorhersage:



$$y_{\text{Ziffer 2}} = \begin{pmatrix} 9.670 * 10^{-6} \\ 4.043 * 10^{-3} \\ 8.592 * 10^{-1} \\ 6.907 * 10^{-2} \\ 4.797 * 10^{-5} \\ 7.315 * 10^{-3} \\ 4.663 * 10^{-7} \\ 2.779 * 10^{-4} \\ 1.147 * 10^{-5} \\ 1.113 * 10^{-5} \end{pmatrix}^T$$

Abbildung 4: MNIST-Bild der Ziffer 2 mit Vorhersage

Die Komponenten der Vorhersage können als eine Art «Wahrscheinlichkeit»[2] interpretiert werden, mit der das Modell den Input einer entsprechenden Klasse zuweist. Je höher der Wert eines Outputs, desto wahrscheinlicher hält das Modell den Input für die entsprechende Klasse.

Im obigen Beispiel ist das Modell relativ sicher, dass es sich beim Input um eine 2 handelt, da der Wert an der Stelle der Ziffer 2 ($8.592 * 10^{-1}$) um einen Faktor von mehr als zehn grösser ist als alle anderen Outputwerte. Zusätzlich sieht man, dass das Modell das Bild wahrscheinlicher für eine 3 ($6.907 * 10^{-2}$), eine 1 ($4.043 * 10^{-3}$) oder eine 5 ($7.315 * 10^{-3}$) hält als z. B. eine 6 ($4.663 * 10^{-7}$) oder eine 0 ($9.670 * 10^{-6}$).

Um die Vorhersage eindeutig einer Klasse zuzuweisen, wählt man die Klasse, deren zugehöriger Output den höchsten Wert hat.

3.1.2 Kostenfunktionen

Die Vorhersagen eines Modells werden mit einem Fehlermass bewertet, das je kleiner ist, desto geringer die Abweichung zwischen Vorhersagen- und Labelvektor ausfällt. Funktionen, welche so einen Zweck erfüllen, nennt man **Kostenfunktionen** (engl.: *cost function*).

Kostenfunktionen können Vorhersagen auf mehreren Ebenen bewerten. Das Fehlermass c vergleicht einen der Outputs des Modells mit dem zugehörigen Element des Labelvektors. Die Kosten einer Vorhersage C werden als die Summe von den c aller Outputs definiert.

$$C(y, \hat{y}) = \sum_{n=1}^{k} c\,(y_n, \hat{y}_n) \tag{1}$$

[2] Die Summe der Komponenten einer Vorhersage muss nicht 1 sein, d.h. dass die «Wahrscheinlichkeitsverteilung» nicht normalisiert ist.

Analog sind die Kosten über einem Datensatz $\bar{C}$ mit p Einträgen das arithmetische Mittel der Kosten aller Vorhersagen, die aus den Features des Datensatzes berechnet werden können (hier soll y_i die Vorhersage zum i-ten Beispiel eines Datensatzes und $\hat{y}_i$ den entsprechenden Labelvektor bedeuten).

$$\bar{C} = \frac{1}{p}\sum_{i=1}^{p} C(y_i, \hat{y}_i) \tag{2}$$

Kostenfunktionen sollten sinnvollerweise folgende Eigenschaften besitzen:

- c ist minimal, wenn $y_n = \hat{y}_n$
- je grösser $|y - \hat{y}|$, desto grösser c
- C ist partiell differenzierbar nach jedem y_n (siehe 3.1.3 Gradient Descent)

Ein Beispiel einer Kostenfunktion ist der «Mean Squared Error» (kurz MSE, engl. für *mittlere quadratische Abweichung*). Wie der Name vermuten lässt, sind die Kosten mit dem MSE das arithmetische Mittel der Abweichungen von Vorhersage und Label im Quadrat.[3] Es ist einfach ersichtlich, dass der MSE alle soeben genannten Eigenschaften einer Kostenfunktion erfüllt.

$$\bar{C}_{MSE} = \frac{1}{2p}\sum_{i=1}^{p}\sum_{n=1}^{k}(y_{n,i} - \hat{y}_{n,i})^2, \qquad \text{mit } c(y_n, \hat{y}_n) = \frac{1}{2}(y_n - \hat{y}_n)^2 \tag{3}$$

Wenn y der Outputvektor und $\hat{y}$ der Labelvektor ist, dann lässt sich der MSE auf Ebene einer Vorhersage mit dem Skalarprodukt auch folgendermassen ausdrücken:

$$C = \frac{1}{2}(y - \hat{y}) * (y - \hat{y}) \tag{4}$$

(Nielsen, Learning with gradient descent, 2017)

3.1.3 Gradient-Descent zum Trainieren von Modellen

Perspektivenwechsel

Durch das Einführen von Kostenfunktionen wird das Trainieren eines Modells zu einem Minimierungsproblem. Das Modell hat die höchste Genauigkeit über dem Trainingsdatensatz, wenn die Kosten über dem Trainingsdatensatz am geringsten sind. Das Ziel eines Trainingsalgorithmus ist es, die Modellparameter zu finden, bei denen die Kosten über dem Trainingsdatensatz möglichst gering sind.

In diesem Kontext muss man die Kosten über einem Datensatz als abhängig von den Modellparametern betrachten, wobei die Features, Vorhersagen und Labels, welche in gewisser Weise zum Datensatz gehören, als konstant angesehen werden.

$$\bar{C}(\theta_0, \theta_1, \theta_2, \dots)$$

[3] Das «Mean» im MSE bezieht sich nur auf den Durchschnitt der Kosten C eines Datensatzes und nicht auf den Durchschnitt aller c.

Gradient-Descent

Ein Optimierungsverfahren zum Minimieren von Funktionen $f(x_1, x_2, \dots, x_k)$ des Typs $\mathbb{R}^k \to \mathbb{R}$ ist **Gradient-Descent** (engl. für *Gradientenabstieg*). Die Idee hinter Gradient-Descent ist es, einen (Ortsvektor zum) Startpunkt[4] $p_{t=0} = (p_{1,t=0} \quad p_{2,t=0} \quad \cdots \quad p_{k,t=0})$ im n-dimensionalen Definitionsraum von f zu wählen, dessen Komponenten als Input für f dienen. Die Komponenten des Startpunktes werden dann schrittweise so verändert, dass der Funktionswert $f(p)$ in jedem Schritt abnimmt. Das wiederholt man, bis man einen Punkt findet, dessen Funktionswert genügend klein ist.

Der Gradient-Descent-Algorithmus nähert sich iterativ einem Minimum an und eignet sich nicht, um Minima exakt zu bestimmen. Das resultierende Minimum muss nicht unbedingt in der Nähe des globalen Minimums liegen, da der Algorithmus zu einem beliebigen lokalen Minimum in der Nähe des Startpunktes konvergiert. Gradient-Descent eignet sich dennoch zum Optimieren von Modellen, wie man in der Praxis zeigen kann (siehe 5. Experiment). Es wird zurzeit noch nicht vollständig verstanden, weshalb das so ist, denn die Anwendbarkeit von Gradient-Descent hängt von der Form der $k+1$-dimensionalen Kostenfunktion ab, welche je nach Modell oder Trainingsdatensatz unterschiedlich ist.

Für Gradient-Descent benötigt man den Begriff des Gradienten. Der Gradient $\nabla f(p)$ ist ein Vektor im Definitionsraum von f, der von p aus in die Richtung der schnellsten Zunahme von f zeigt. Wenn man den Funktionswert von f verringern möchte, muss der nächste Punkt p_{t+1} von p_t aus in der Richtung von $-\nabla f(p_t)$ liegen. Die Grundgleichung von Gradient-Descent mit η als positive Proportionalitätskonstante ist deshalb wie folgt:

$$p_{t+1} = p_t - \eta * \nabla f(p_t) \tag{5}$$

Die Komponenten des Gradienten sind die partiellen Ableitungen $\frac{\partial f}{\partial x_n}$ von f nach allen Veränderlichen x_k von f an der Stelle p:

$$\nabla f(p) = \left(\frac{\partial f(p)}{\partial x_1} \quad \frac{\partial f(p)}{\partial x_2} \quad \dots \quad \frac{\partial f(p)}{\partial x_k}\right) \tag{6}$$

Die partielle Ableitung einer multidimensionalen Funktion in einem Punkt p ist die Tangentensteigung in der Achsenrichtung der Veränderlichen, nach der man ableitet, und ist wie folgt definiert:

$$\frac{\partial f(p)}{\partial x_n} = \lim_{h \to 0} \frac{f(p_1, \dots, p_n + h, \dots, p_k) - f(p)}{h} \tag{7}$$

Beim partiellen Ableiten einer Funktion muss man alle Veränderlichen, nach denen man nicht ableitet, als Konstanten behandeln. Ansonsten gelten für partielle Ableitungen dieselben Regeln wie bei gewöhnlichen Ableitungen.

[4] Punkt und Vektor werden hier als Synonyme verwendet. Gemeint sind die Zahlen, die als Input zu f dienen.

Die zu minimierende Funktion nimmt oft nicht direkt einen Vektor entgegen, sondern die einzelnen Veränderlichen, die hier in einem Vektor zusammengefasst wurden. In diesem Fall wird der «Gradient»[5] komponentenweise angewandt:

$$p_{k,t+1} = p_{k,t} - \eta * \frac{\partial f(p_t)}{\partial x_k} \tag{8}$$

(Nielsen, Learning with gradient descent, 2017)

Lernrate η

Die Schrittgrösse η wird als **Lernrate** (engl.: *learning rate*) bezeichnet und ist ein Hyperparameter beim Training von Modellen. Je nach Funktion, die minimiert wird, funktioniert ein anderer Wert besser. Es gibt keine Regel, nach der man zuverlässig eine gute Lernrate findet; es muss also durch Ausprobieren ein Wert gefunden werden, der funktioniert:

- wenn η zu klein gewählt wird, sind die Schritte kleiner, als sie sein müssten, und Gradient Descent konvergiert nur langsam. Eine weitere Gefahr ist es, dadurch in einem lokalen Minimum stecken zu bleiben.
- wenn η hingegen zu gross gewählt wird, springt der Punkt am Minimum herum, ohne es zu erreichen.

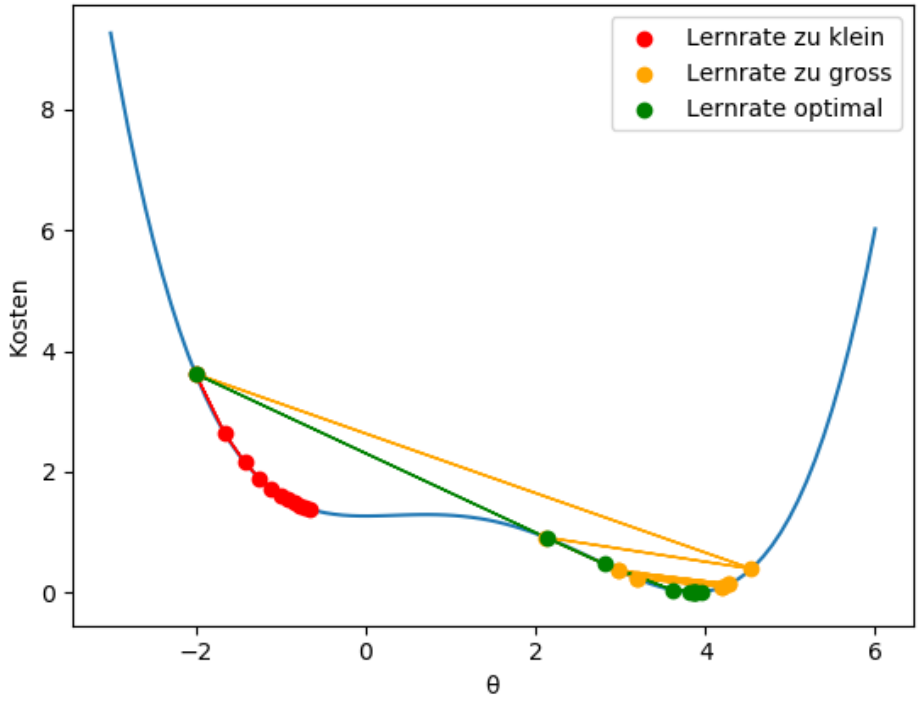


Abbildung 5: Auswirkung der Lernrate auf Gradient Descent anhand eines fiktiven Beispiels

Oft wird in der Praxis anfangs eine hohe Lernrate genutzt, um nicht in lokalen Minima stecken zu bleiben und um schnelle Konvergenz sicherzustellen. Später verringert man die

[5] Der Einfachheit halber wird die partielle Ableitung der Funktion nach einer der Veränderlichen oft Gradient genannt.

Lernrate, um in das Zentrum des Minimums zu fallen und nicht darum herumzuspringen. Die Verringerung kann manuell oder durch eine im Voraus bestimmten Regel erfolgen.

(Géron, Gradient Descent, 2017)

Stochastic Gradient-Descent

Beim maschinellen Lernen verwendet man Gradient-Descent, um die Kosten über dem gesamten Trainingsdatensatz zu minimieren, indem die Modellparameter schrittweise verändert werden. Die Trainingsdaten werden so oft durchgangen, bis das Modell zufriedenstellend genau ist. Einen Durchlauf der Trainingsdaten nennt man eine **Epoche** (engl.: *epoch*).

Bei grossen Trainingsdatensätzen und komplexen Modellen kann es sehr rechenaufwändig sein, den Gradienten der Kosten über dem Trainingsdatensatz exakt zu berechnen. Es stellt sich aber heraus, dass ein kleiner Teil der Trainingsdaten, ein sogenannter **Mini-Batch**[6], ausreicht, um den Gradienten des gesamten Datensatzes ausreichend gut zu approximieren. Der Vorteil liegt in den deutlich häufigeren Aktualisierungen der Parameter, was den Nachteil der verringerten Genauigkeit in der Regel übertrifft. Dieses Verfahren nennt man **Stochastic Gradient-Descent**[7] (zu Deutsch *zufälliger Gradientenabstieg*, kurz SGD), da es durch die verringerte Genauigkeit ein wenig Zufall in das Training einbringt.

Konkret wird $\frac{\partial C}{\partial \theta_k}$ jeder Vorhersage des Mini-Batches berechnet. Alle $\frac{\partial C}{\partial \theta_k}$ werden aufsummiert und skaliert, um für die Anzahl der Summanden zu kompensieren. Mit der Mini-Batchgrösse q gilt folgende Approximation:

$$\frac{\partial \bar{C}}{\partial \theta_n} \approx \frac{1}{q} \sum_{i=1}^{q} \frac{\partial C_i}{\partial \theta_n} \tag{9}$$

Mit SGD werden die Modellparameter θ_n basierend auf der Approximation und Formel 8 wie folgt aktualisiert:

$$\theta_{n,t+1} = \theta_{n,t} - \frac{\eta}{q} \sum_{i=1}^{q} \frac{\partial C_i}{\partial \theta_{n,t}} \tag{10}$$

Die Grösse der Mini-Batches ist erneut ein Hyperparameter mit folgendem Einfluss auf den Lernprozess:

- Wenn die Mini-Batches zu klein sind, werden die Modellparameter zwar oft aktualisiert, aber dafür ist die Approximation des Gradienten ungenau.
- Bei zu grossen Mini-Batches ist die Approximation genau, aber dafür werden die Modellparameter weniger oft aktualisiert und SGD ist weniger effektiv.

Der Trainingsdatensatz wird i.d.R. zufällig in die Mini-Batches unterteilt, so dass ein Mini-Batch Beispiele aus verschiedenen Klassen beinhaltet.

[6] Batch bezeichnet bei modellbasiertem maschinellen Lernen üblicherweise den ganzen Trainingsdatensatz, daher ist ein Teil der Trainingsdaten ein Mini-Batch.
[7] wird auch als Mini-Batch-Gradient-Descent bezeichnet, Stochastic Gradient-Descent kann auch für das Training mit der Mini-Batchgrösse 1 stehen

Zusammengefasst läuft der SGD-Algorithmus folgendermassen ab:

1. Parameter initialisieren
2. Für jede Epoche:
    - Trainingsdatensatz zufällig in Mini-Batches einteilen
    - Für jeden Mini-Batch:
        - Für jedes Beispiel im Mini-Batch:
            - Partielle Ableitungen der Kosten des Beispiels nach jedem Parameter berechnen und für jeden Parameter separat summieren
        - Modellparameter nach Gleichung 10 aktualisieren

(Nielsen, Learning with gradient descent, 2017)

3.2 Deep Learning und neuronale Netze

Deep Learning ist ein Teilgebiet des modellbasierten maschinellen Lernens, in dem die Modelle **künstliche neuronale Netze** (engl.: *artificial neural networks*) sind.

3.2.1 Das Perzeptron

Die erste und einfachste Form eines künstlichen neuronalen Netzes, das Perzeptron-Netz, wurde in den 50er-Jahren vom Psychologen Frank Rosenblatt entwickelt. Das Modell basiert auf dem Perzeptron[8], einem **künstlichen Neuron**, welches bereits in den 40er-Jahren eingeführt wurde. Das Modell orientiert sich an der Funktionsweise von biologischen Neuronen und versucht, die Vorgänge des Nervensystems auf abstrakte Art darzustellen. (Wikipedia - Perzeptron, 2018)

Eine Nervenzelle erhält Signale in Form von elektrischen Potenzialen über den synaptischen Spalt. Am synaptischen Spalt können die Signale entweder positiv (Spannung wird erhöht) oder negativ (Spannung wird gesenkt) verstärkt werden. Die Handhabung an einem synaptischen Spalt ändert sich selten. Alle ankommenden Signale wandern den Dendriten entlang in den Zellkörper, wo sie aufeinandertreffen. Dabei werden sie addiert, da es sich bei den Signalen um elektrische Potenziale handelt. Wenn die resultierende Spannung im Zellkörper über einem festen Schwellwert liegt, feuert das Neuron und ist aktiviert. Das bedeutet, dass ein Signal einer immer gleichen Spannung über das Axon an viele weitere Neuronen übermittelt wird. Falls der Schwellwert nicht überschritten wird, gibt das Neuron kein Signal weiter. (Campbell & Reece, 2011)

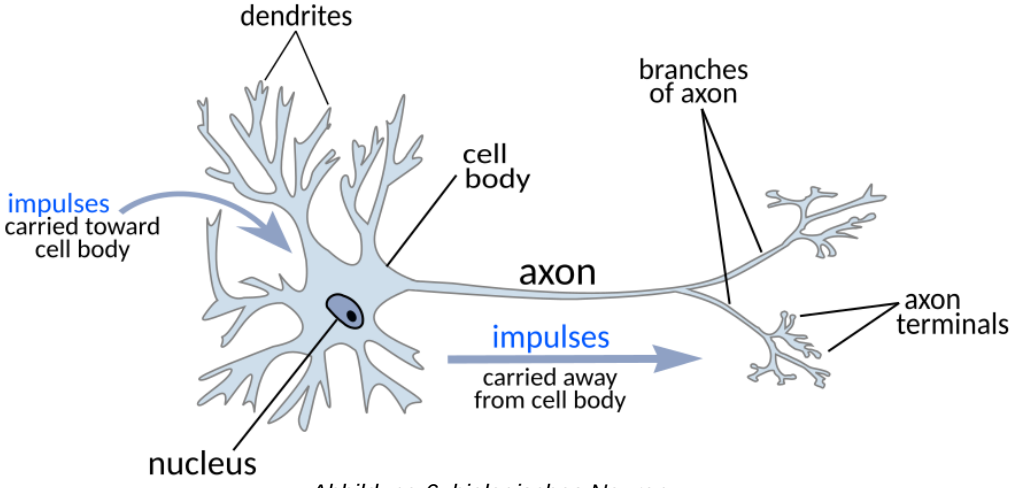


Abbildung 6: biologisches Neuron

Ein künstliches Neuron kommuniziert mit Zahlen anstatt elektrischen Potenzialen. Die Gewichtung, die bei biologischen Neuronen am synaptischen Spalt stattfindet, wird im Modell durch die Multiplikation der «Inputzahl» mit einem **Gewicht** ersetzt. Dieses Gewicht darf jede Zahl annehmen. Als nächstes werden die gewichteten Inputs aufsummiert. Die Formel für $\tilde{z}$ als Zahl im «Zellkörper», mit x_n als Inputzahl, w_n als Gewicht und k als Anzahl Inputs lautet also wie folgt:

$$\tilde{z} = w_1 x_1 \ + \ w_2 x_2 + \cdots + \ w_k x_k = \sum_{n=1}^{k} w_n x_n$$

[8] Perzeptron kann sich auch auf das Netz beziehen, in dieser Arbeit ist damit aber immer das künstliche Neuron gemeint.

Nun wird $\tilde{z}$ mit dem Schwellwert $\tilde{b}$ verglichen, um die **Aktivierung** a (engl.: *activation*) des Neurons zu erhalten. Der Schwellwert ist in allen biologischen Neuronen gleich, bei künstlichen Neuron darf $\tilde{b}$ aber von Neuron zu Neuron unterschiedlich sein. Die Aktivierung soll 1 sein, wenn das Neuron feuert, und das ist der Fall, wenn $\tilde{z} \geq \tilde{b}$. Ansonsten (wenn $\tilde{z} < \tilde{b}$) soll sie 0 sein.

$$a(\tilde{z}) = \begin{cases} 0, & \tilde{z} < \tilde{b} \\ 1, & \tilde{z} \geq \tilde{b} \end{cases}$$

Nun kann man diese Bedingung ein wenig umformen. Es ist üblich, den Schwellwert auf die andere Seite der Ungleichung zu nehmen. Dann erhält man:

$$a(\tilde{z}) = \begin{cases} 0, & \tilde{z} - \tilde{b} < 0 \\ 1, & \tilde{z} - \tilde{b} \geq 0 \end{cases}$$

Damit erhält man eine neue Grösse z, die **gewichtete Summe** (engl.: *weighted sum*) genannt wird. b $(= -\tilde{b})$ nennt man die **Neigung** (engl.: *bias*) des Neurons, da damit die Tendenz (unabhängig des Inputs) des Neurons zu feuern bestimmt ist (je grösser b, desto grösser z, umso häufiger $z > 0$ und $a = 1$).

$$z = \tilde{z} - \tilde{b} = \tilde{z} + b = \sum_{n=1}^{k} w_n x_n + b$$

Die Aktivierung ist also nur noch abhängig von z. Eine Funktion, die aus z a berechnet, nennt man eine **Aktivierungsfunktion**. Die Aktivierungsfunktion des Perzeptrons ist die Heavisidestufenfunktion:

$$a(z) = \begin{cases} 0, & z < 0 \\ 1, & z \geq 0 \end{cases}$$

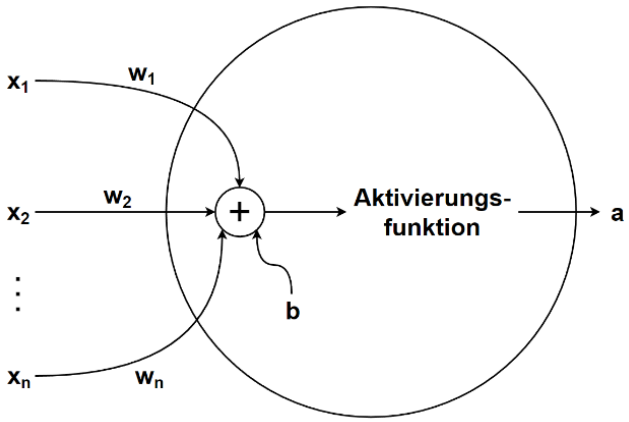


Abbildung 7: künstliches Neuron

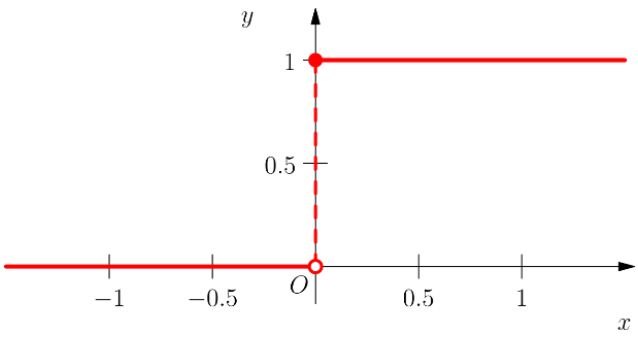


Abbildung 8: Heavisidestufenfunktion, Aktivierungsfunktion des Perzeptrons

Zusammengefasst bestimmt man die Aktivierung eines Perzeptrons in zwei Schritten:

$$z(x_1, \ldots, x_k) = \sum_{n=1}^{k} w_n x_n + b \tag{11}$$

$$a(z) = \begin{cases} 0, & z < 0 \\ 1, & z \geq 0 \end{cases} \tag{12}$$

(Nielsen, Perceptrons, 2017)

3.2.2 Künstliche neuronale Netze

Neuronen in künstlichen neuronalen Netzen sind in **Schichten** (engl.: *layers*) aufgereiht, so dass die Aktivierung jedes Neurons einer Schicht als Input für alle Neuronen der nächsten Schicht dient. Der Output des Netzes sind die Aktivierungen der letzten Schicht, daher auch **Outputschicht** genannt. Die Aktivierungen der Outputschicht sind die Vorhersage eines künstlichen neuronalen Netzes. Oft werden neuronale Netze mit einer **Inputschicht** aus Neuronen dargestellt. Diese «Neuronen» dienen nur dazu, den Neuronen der nächsten Schicht Inputs zu liefern, und sind deshalb eigentlich nur Platzhalter für die Features. Alle anderen Schichten neben Input- und Outputschicht nennt man **Zwischenschichten** (engl.: *hidden layers*). Ein Netz mit mehreren Zwischenschichten nennt man «deep» (engl. für tief), daher der Name «Deep Learning».

Neuronale Netze dieser Art nennt man Feedforward-Netze, da die Features Schicht für Schicht durch das Netz in eine Richtung «weitergefüttert» werden, ohne eine Berechnung in einer Vorschicht zu beeinflussen. Der **Feedfoward** bezeichnet also das Durchlaufen von Information durch ein solches Netz

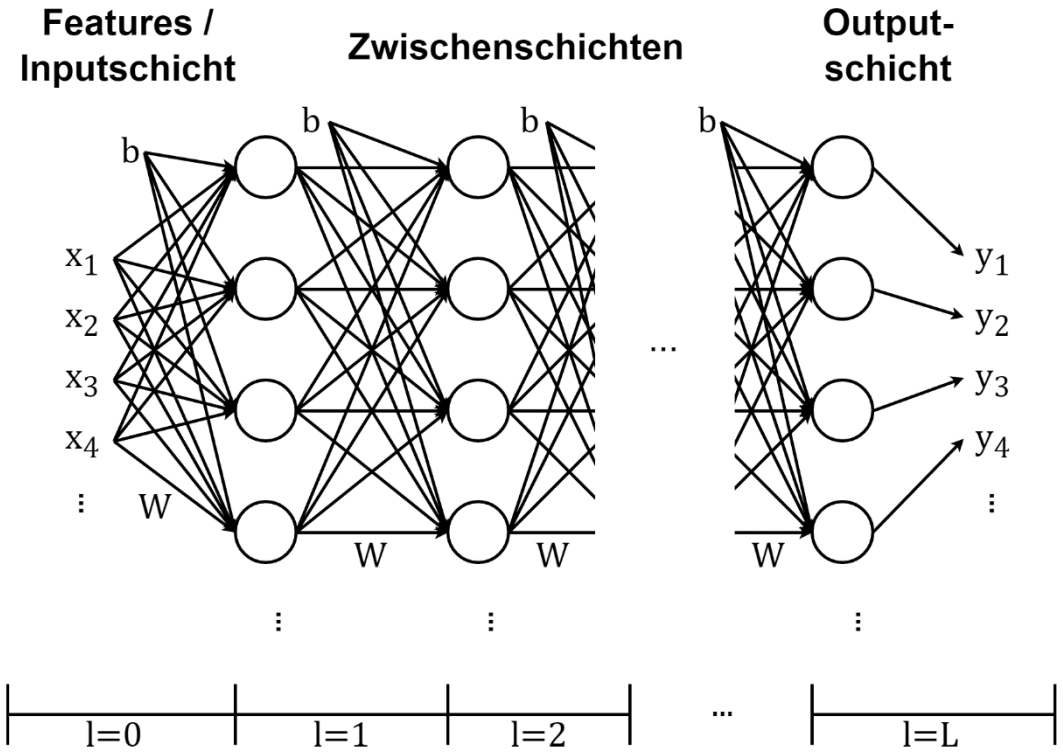


Abbildung 9: Schema eines künstlichen neuronalen Netzes; auf der Abbildung haben alle Neuronen einer Schicht zur Übersicht dieselbe Neigung, eigentlich kann jede Neigung verschieden sein.

Ein Neuron der ersten Zwischenschicht eines trainierten neuronalen Netzes sorgt für die Erkennung grober Muster in den Features und feuert bei einem MNIST-Modell z.B., wenn an einer Stelle im Inputbild eine hohe Konzentration an dunklen Pixeln vorhanden ist. Ein Neuron der zweiten Zwischenschicht feuert, wenn ein bestimmtes Muster in den Aktivierungen der ersten Zwischenschicht vorhanden ist. Das setzt sich bis zur Outputschicht fort, wobei jede Schicht für eine Abstraktion in der Informationsverarbeitung sorgt.

(Nielsen, The architecture of neural networks, 2017)

Die gewichteten Summen, Aktivierungen, Gewichte und Neigungen werden nach folgenden Definitionen gekennzeichnet:

- a_n^l ist Aktivierung des n-ten Neurons in der l-ten Schicht.[9] Sie wird aus der gewichteten Summe dieses Neurons z_n^l berechnet.
- Die Verbindungen zwischen Neuronen stehen in Abbildung 9 für die Multiplikation der Aktivierung mit dem zugehörigen Gewicht. $w_{n,m}^l$ ist das Gewicht, welches das n-te Neuron in der l-ten Schicht mit m-ten Neuron der Schicht $l+1$ verbindet.[10] Analog ist b_n^l die Neigung zum Bestimmen der gewichteten Summe des n-ten Neurons der Schicht $l+1$.
- Des Weiteren soll $|l|$ die Anzahl Neuronen im Layer l bedeuten (Es handelt sich nicht um den eigentlichen Betrag der Zahl der Schicht!).

3.2.3 Lineare Algebra in neuronalen Netzen

Bei grossen Netzen gibt es enorm viele Komponenten, pro Neuron eine Aktivierung, pro Neuron nach der Inputschicht eine gewichtete Summe und eine Neigung, und schlussendlich $n * m$ Gewichte zwischen allen Schichten mit n und m als Anzahl Neuronen der aufeinanderfolgenden Schichten. Bei den grössten Netzen, die in dieser Arbeit zum Einsatz kommen, sind das insgesamt 43'314 Zahlen, über die man den Überblick behalten müsste. Die Berechnung von gewichteten Summen und Aktivierungen lässt sich mit linearer Algebra deutlich übersichtlicher ausdrücken und ist zudem in Implementierungen schneller.

Die Aktivierungen einer Schicht werden zu den Komponenten eines Aktivierungs(zeilen[11])vektors.

$$a^l = \begin{pmatrix} a_1^l & \cdots & a_{|l|}^l \end{pmatrix}$$

Die Gewichte zwischen zwei Schichten sind die Komponenten einer Gewichtsmatrix.

$$W^l = \begin{pmatrix} w_{1,1}^l & w_{1,2}^l & \cdots & w_{1,|l+1|}^l \\ w_{2,1}^l & w_{2,2}^l & \cdots & w_{2,|l+1|}^l \\ \vdots & \vdots & \ddots & \\ w_{|l|,1}^l & w_{|l|,2}^l & & w_{|l|,|l+1|}^l \end{pmatrix}$$

Durch die Matrixmultiplikation von a^l und W^l erhält man einen Vektor, dessen Komponenten die gewichteten Summen $\tilde{z}$ ohne den Neigungsterm der $l+1$-ten Schicht sind:

$$a^l * W^l = \begin{pmatrix} \sum_{n=1}^{|l|} w_{n,1}^l a_n^l & \sum_{n=1}^{|l|} w_{n,2}^l a_n^l & \cdots & \sum_{n=1}^{|l|} w_{n,|l+1|}^l a_n^l \end{pmatrix}$$

[9] Der Exponent steht in dieser Arbeit bis auf wenige Ausnahmen für die Schicht, in der sich ein Element befindet.

[10] Die Schichtzahl eines Gewichts ist also die gleiche wie die der Aktivierung, mit der es bei der Berechnung der gewichteten Summe multipliziert wird, obwohl das Gewicht konzeptuell eher zum Neuron gehören sollte, dessen Aktivierung es beeinflusst. Gleiches gilt für die Neigungen. Der Grund für diese Zuweisung ist, dass die Gewichte der ersten Schicht im Programm den selben Index wie die Aktivierungen der ersten Schicht haben werden, da sie jeweils das erste Element einer Liste sein werden.

[11] Die Vektoren in dieser Arbeit sind Zeilenvektoren, da es einfacher ist, Zeilenvektoren in Programmen anzugeben ([1, 2, 3, …] statt [[1], [2], [3], …]).

Definiert man einen Neigungsvektor, dessen Komponenten die Neigungen einer Schicht sind, kann dieser dazu addiert werden, um einen Vektor der gewichteten Summen der nächsten Schicht zu erhalten.

$$b^l = \begin{pmatrix} b_1^l & b_2^l & \cdots & b_{|l+1|}^l \end{pmatrix}$$

$$z^{l+1} = a^l * W^l + b^l$$

$$= \begin{pmatrix} \sum_{n=1}^{|l|} w_{n,1}^l a_n^l + b_1^l & \sum_{n=1}^{|l|} w_{n,2}^l a_n^l + b_2^l & \cdots & \sum_{n=1}^{|l|} w_{n,|l+1|}^l a_n^l + b_{|l+1|}^l \end{pmatrix}$$

$$= \begin{pmatrix} z_1^{l+1} & z_2^{l+1} & \cdots & z_{|l+1|}^{l+1} \end{pmatrix}$$

Nun muss man nur noch damit die Aktivierung bestimmen. Dazu wendet man die Aktivierungsfunktion elementweise auf den z-Vektor an. Nach dieser Konvention haben alle Neuronen einer Schicht die gleiche Aktivierungsfunktion. Wenn z ein Vektor ist, ist $A(z)$[12] als die vektorisierte Aktivierungsfunktion zu verstehen.

$$a^{l+1} = A\left(z^{l+1}\right) = \begin{pmatrix} A\left(z_1^{l+1}\right) & A\left(z_2^{l+1}\right) & \cdots & A\left(z_{|l+1|}^{l+1}\right) \end{pmatrix}$$

Die Feedforward-Gleichungen in Vektorschreibweise sind zusammengefasst wie folgt:

$$z^{l+1}(a^l) = a^l * W^l + b^l \tag{13}$$

$$a^{l+1} = A(z^{l+1}) \tag{14}$$

(Nielsen, Implementing our network to classify digits, 2017)

3.2.4 Sigmoid-Neuronen

Damit ein neuronales Netz mit dem SGD-Algorithmus trainiert werden kann, muss eine kleine Änderung eines Parameters zu einer kleinen Änderung in den Kosten führen, so dass ein Schritt gewählt werden kann, der die Kosten verringert. Wenn man in einem Perzeptron-Netz ein Gewicht oder eine Neigung leicht ändert, ändert sich die nächste gewichtete Summe leicht. Da die Aktivierungsfunktion die Heavisidestufenfunktion ist, ändert sich die Aktivierung aber nicht, und somit auch nicht die Aktivierungen in den folgenden Schichten und die Kosten.

Das führt zu einer Bedingung, die neuronale Netze erfüllen müssen, wenn man sie mit SGD trainieren will: Die Aktivierungsfunktion darf nicht aus nur konstanten Teilen zusammengesetzt sein (bzw. die Ableitung der Aktivierungsfunktion darf nicht überall 0 sein). Wenn Gewichte und Neigungen zu einer gewichteten Summe führen, die im konstanten Bereich der Aktivierungsfunktion liegt, dann ist die partielle Ableitung der Kosten nach diesen Parametern 0 und somit werden sie nicht aktualisiert. Da die Heavisidestufenfunktion konstant ist, werden die Parameter eines Perzeptron-Netzes mit SGD nie aktualisiert und das Netz bleibt mit den Startparametern stehen.

[12] $A(z)$ wird in der Arbeit ebenfalls als nicht-vektorisierte Aktivierungsfunktion verwendet.

Weitere Eigenschaften von sinnvollen Aktivierungsfunktionen sind folgende:

- monoton steigend, eine Erhöhung der gewichteten Summe sollte nicht zu einer Abnahme der Aktivierung führen
- Auf ganz $\mathbb{R}$ definiert, z kann jede reelle Zahl annehmen, da die Gewichte und Neigungen i.d.R. nicht beschränkt sind

Das stetige Äquivalent der Heavisidestufenfunktion, das alle oben genannten Bedingungen erfüllt, ist die Sigmoidfunktion σ:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{15}$$

$$\sigma'(z) = \frac{e^{-z}}{(1 + e^{-z})^2} = \frac{e^{-z}}{1 + e^{-z}} * \frac{1}{1 + e^{-z}} = \big(1 - \sigma(z)\big) * \sigma(z) \tag{16}$$

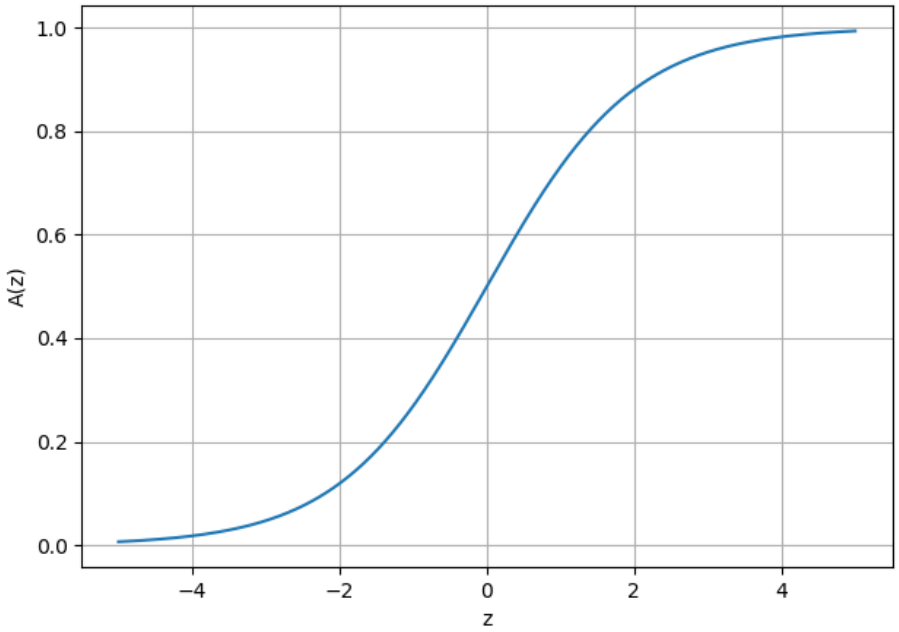


Abbildung 10: Graph der Sigmoidfunktion

Wichtige Eigenschaften der Sigmoid-Funktion sind:

- für alle z ist $\sigma(z) \in (0,1)$, wobei $\lim_{z \to \infty} \sigma(z) = 1$ und $\lim_{z \to -\infty} \sigma(z) = 0$
- $\sigma'(z)$ ist mit $\sigma'(0) = 0.25$ maximal und für alle z positiv

(Nielsen, Sigmoid neurons, 2017)

3.2.5 Überlegungen zur Modellkomplexität von neuronalen Netzen

Universal Approximation Theorem

Weshalb sind gerade neuronale Netze gute Modelle für Machine Learning? Eine Antwort darauf liefert das Universal Approximation Theorem (kurz UAT, engl. für *universelles Approximationstheorem*), nach dem ein neuronales Netz mit nur einer Zwischenschicht jede kontinuierliche Funktion approximieren kann.1990 wurde das Universal Approximation Theorem für neuronale Netze mit kontinuierlichen, beschränkten und nichtkonstanten Aktivierungsfunktionen von Kurt Hornik bewiesen. (Hornik, 1990)

In diesem Abschnitt beschreibe ich eine Beweisskizze des UAT für die Approximation von Funktionen von $\mathbb{R}$ nach $\mathbb{R}$ anhand eines Netzes mit einem Inputneuron, einer Zwischenschicht mit der Sigmoid-Aktivierungsfunktion und einem Outputneuron ohne Aktivierungsfunktion. Die Aktivierung des n-ten Zwischenneurons ist gegeben durch:

$$a_n = \sigma(w_n * x + b_n)$$

Der erste Schritt ist die Erkenntnis, dass eine solche Aktivierung eine Stufenfunktion mit beliebiger Stelle der Stufe bezüglich des Inputs x annähern kann. Für die Stufenstelle gilt $w_k * x + b_k = 0$, somit befindet sich die Stufe bei $x = \frac{-b_n}{w_n}$. Bei grossen positiven w_n ist jedes $x > \frac{-b_n}{w_n}$ nahe an 1 und jedes $x < \frac{-b_n}{w_n}$ nahe an 0.

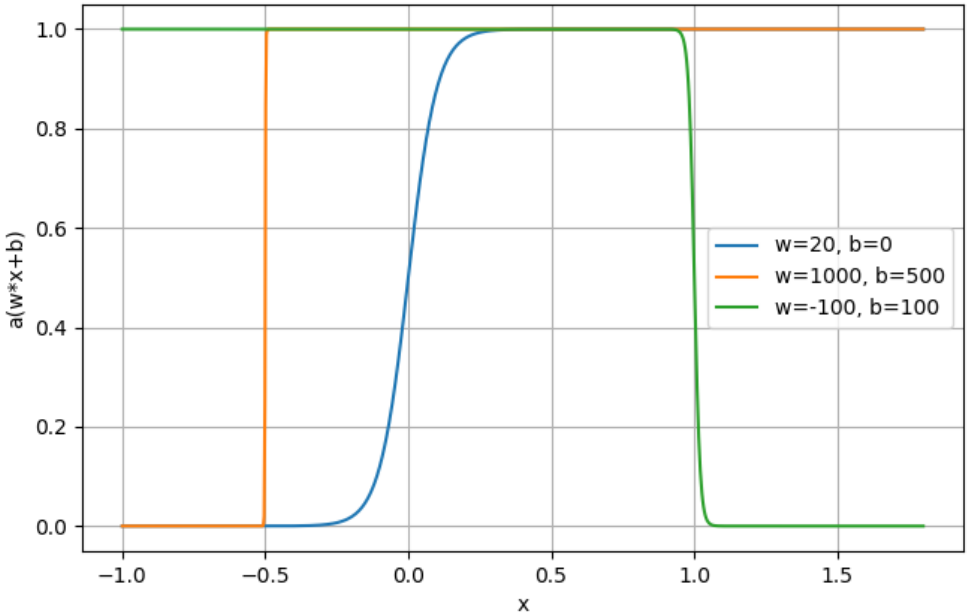


Abbildung 11: Sigmoidfunktion zur Approximation von Stufenfunktionen

Jede andere beidseitig beschränkte[13], monoton steigende und nichtkonstante Funktion kann ebenfalls eine Stufenfunktion annähern, somit gilt diese Beweisskizze für alle neuronalen Netze mit solchen Aktivierungsfunktionen.

Das Outputneuron gewichtet (v_n) und addiert die Aktivierungen der Zwischenschicht.

$$y = \sum_{n=1}^{k} v_n * a_n$$

Eine Teilsumme aus zwei gewichteten Aktivierungen kann in einem Bereich von x einen bestimmten Wert h annähern und sonst überall 0 sein. Dazu hat die Aktivierung des ersten Summanden ihren Schritt an der unteren Grenze des Bereichs und wird mit h gewichtet. Die Aktivierung des zweiten Summanden hat ihren Schritt an der oberen Grenze des Bereichs und wird mit $-h$ gewichtet.

[13] Die obere und untere Schranke dürfen nicht den gleichen Wert haben.

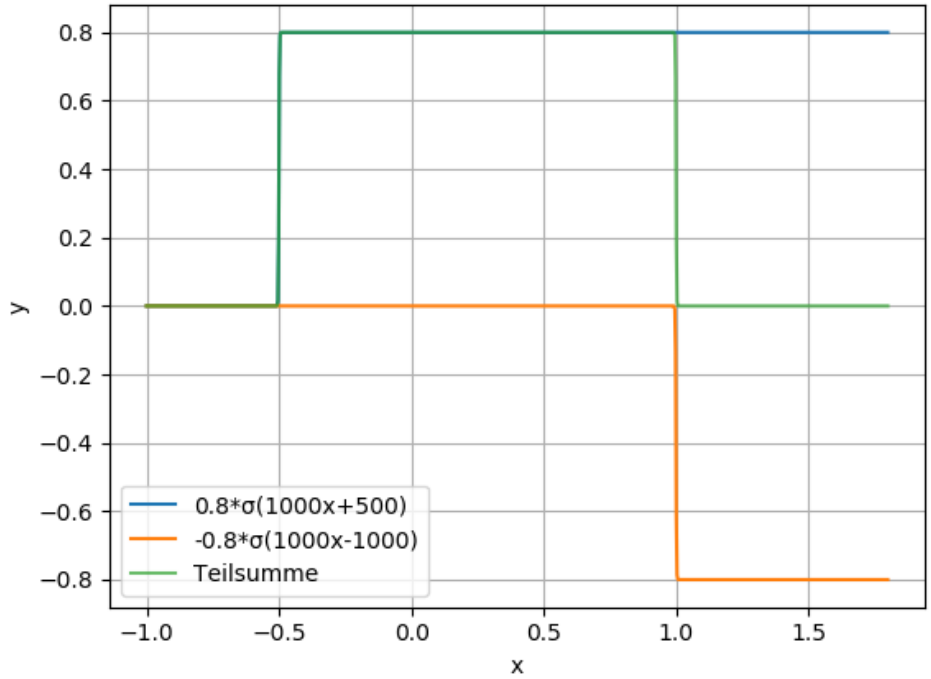


Abbildung 13: gewichtete Summe aus zwei Sigmoidaktivierungen nähert für x zwischen -0.5 und 1 den Wert h=0.8 an und ist sonst quasi 0

Mit beliebig vielen Zwischenneuronen lassen sich mit diesen Teilsummen beliebige reelle Funktionen zusammenbauen, indem jede Teilsumme den Funktionswert der anzunähernden Funktion in einem anderen Bereich von x übernimmt. Je mehr Zwischenneuronen zur Verfügung stehen, desto kleiner können die Bereiche werden (wenn die Funktion in einem konstanten Abschnitt approximiert wird) oder desto grösser ist der Abschnitt, in welchem die Funktion approximiert wird (wenn die Bereiche konstant gross sind), umso besser kann eine Approximation sein.

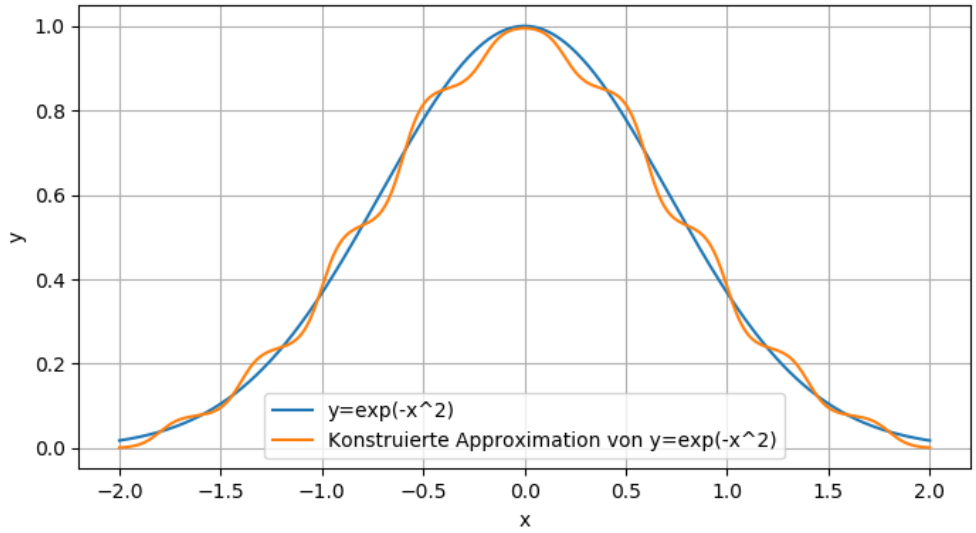


Abbildung 12: Approximation der Normalverteilung durch ein konstruiertes neuronales Netz mit 18 Sigmoidneuronen in der Zwischenschicht und somit 9 Bereichen einer Breite von 0.4. Da alle w_n mit 20 relativ klein sind, ist die Approximation nicht so rechteckig wie in Abbildung 12

Diese Beweisskizze lässt sich auf beliebige Funktionen von $\mathbb{R}^n$ nach $\mathbb{R}^m$ ausweiten, indem die Zwischenneuronen auf n-dimenionale Bereiche «reagieren» und indem die Gewichte der letzten Schicht in einem Bereich bei jedem der m Outputs einen anderen Wert erzeugen. (Nielsen, Universality with one input and one output, 2017)

Das UAT garantiert, dass es zu jeder sinnvollen Funktion ein Netzwerk gibt, welches die Funktion mit hoher Genauigkeit approximieren kann. Es macht aber keine Aussage darüber, wie gross das Netz mindestens sein muss, wie lange ein Netz braucht, um eine bestimmte Funktion zu lernen, und ob es überhaupt möglich ist, die Funktion mit einem bestimmten Trainingsalgorithmus zu erlernen (z.B. weil das Netz zu einem schlecht generalisierten Resultat kommt, auch wenn alle verfügbaren Trainingsdaten genutzt werden).

(Goodfellow, Bengio, & Courville, Universal Approximation Properties and Depth, 2016)

Neuronale Netze als Abbildungen des Featureraums

Eine weitere Überlegung zur Modellkomplexität von neuronalen Netzen befasst sich mit der geometrischen Intuition hinter den Operationen, die in ihnen durchgeführt werden.

Die Aktivierung jedes Outputneurons eines Netzes zur Klassifikation kann als «Entscheidung» interpretiert werden, mit Aktivierungen über 0.5 für die Entscheidung «gehört der Klasse an» und unter 0.5 für «gehört der Klasse nicht an». Alle Outputneuronen führen diese «Entscheidung» für ihre entsprechende Klasse durch. Die **Decision-Boundary** (engl. für *Entscheidungsgrenze*) ist die Grenze zwischen den beiden «Entscheidungsmöglichkeiten» eines Outputneurons im Inputraum eines Modells.

Die Decision-Boundary eines Neurons mit zwei Inputs ist eine Linie. Dies wird aus folgender Überlegung ersichtlich: Die Grenze der Aktivierung eines Sigmoidneurons ist bei $z = 0$. Dadurch muss an der Decision Boundary $z = x_1 w_1 + x_2 w_2 + b = 0$ gelten, welches die Gleichung einer Linie in der x_1/x_2-Ebene ist. Für die Decision-Boundary eines Neurons mit k Eingängen muss analog

$$\sum_{n=1}^{k} x_n w_n + b = 0$$

gelten. Dies ist die generelle Gleichung einer Hyperebene (in einer Dimension ein Punkt, in zwei Dimensionen eine Gerade und in drei Dimensionen eine Ebene) im k-dimensionalen Raum.

Dadurch kann eine Schicht aus Neuronen (ein Neuron pro Klasse) nur Datensätze korrekt klassifizieren, deren Klassen durch Hyperebenen trennbar sind. Damit ein neuronales Netz aus mehreren Schichten einen Datensatz erlernen kann, müssen die Aktivierungen der vorletzten Schicht nach den Klassen, aus denen sie erzeugt wurden, durch eine Hyperebene getrennt werden können.

Die Datenverarbeitung eines neuronalen Netzes beginnt in dessen Inputraum ($\mathbb{R}^{|l=0|}$), in dem jeder Punkt (oder Vektor; hier werden beide Begriffe synonym verwendet) ein möglicher Input für das Modell ist. Bei neuronalen Netzen wird ein (Ortsvektor zu einem) solcher(n) Punkt mit der Gewichtsmatrix multipliziert, was einer linearen Abbildung entspricht. Dadurch wird der Inputraum auf ein Koordinatensystem mit neuen Basisvektoren abgebildet, aber nicht verzerrt, d.h. es spielt keine Rolle, ob zwei Vektoren abgebildet und dann summiert oder zuerst summiert und dann abgebildet werden (je nach Gewichtsmatrix kann der Raum zusätzlich um Dimensionen erweitert oder eingeengt werden). Auf die Matrixmultiplikation folgt die Addition des Neigungsvektors; geometrisch bedeutet dies eine Verschiebung des Raumes. Diese Punkte, deren Komponenten die ersten gewichteten Summen sind, werden

darauf mit der Aktivierungsfunktion komponentenweise abgebildet. Die Sigmoid-Aktivierungsfunktion quetscht alle Komponenten der Punkte auf das Intervall $(0,1)$, somit wird der gesamte Raum auf einen Hyperwürfel (Verallgemeinerung eines Würfels; $a_n^l \in (0,1)^{|l|}$) abgebildet und verzerrt. Die Aktivierungen dieses Raums sind eine neue Repräsentation der Features. Durch jede weitere Schicht wird die Repräsentation durch die genannten Operationen in eine neue Repräsentation umgewandelt, bis zur letzten Repräsentation, den Outputaktivierungen.

Aus dieser Anschauung lässt sich ableiten, dass ein neuronales Netz ohne Aktivierungsfunktion (oder anders ausgedrückt mit der Einheits-Aktivierungsfunktion $A(z) = z$) nur linear separierbare Datensätze erlernen kann, denn dadurch sind neue Repräsentationen bloss nacheinander durchgeführte lineare Abbildungen und Verschiebungen des Featureraums (das Anwenden der Aktivierungsfunktion und das Verzerren des Raums fallen weg), welche sich durch eine einzige lineare Abbildung und Verschiebung zusammenfassen lassen. Dies entspräche dann einem neuronalen Netz mit nur einer Inputschicht, welche direkt mit der Outputschicht verbunden ist. Wie vorhin gezeigt, muss die Repräsentation der Features in der vorletzen Schicht linear separierbar sein, so dass ein Datensatz klassifiziert werden kann. Die vorletzte Schicht in einem Netz (ohne Aktivierungsfunktion) ist die Inputschicht, somit müsste der Datensatz im Featureraum bereits linear separierbar sein, um korrekt klassifiziert werden zu können.

(Olah, Neural Networks, Manifolds, and Topology, 2014)

3.2.6 Gewichte und Neigungen initialisieren

Um SGD durchführen zu können, muss man mit Startparametern beginnen. Schlechte Initialisierung der Startparameter kann es dem Trainingsalgorithmus im schlimmsten Fall unmöglich machen, ein zufriedenstellendes Resultat zu erreichen, oder die Konvergenz zu einem Minimum enorm verlangsamen.

Wenn alle Parameter anfangs auf den gleichen Wert gesetzt werden, dann verhalten sich die Neuronen bei gleichen Inputs identisch, sowohl beim Berechnen von Vorhersagen, wie auch beim Aktualisieren der Parameter durch SGD. Die Symmetrie der Neuronen verhindert, dass sie individuell auf Muster im Input reagieren, wodurch alle Neuronen dasselbe berechnen und somit eine Schicht des Netzes effektiv nur so viel tut wie ein einzelnes Neuron. Um die Symmetrie zu brechen, brauchen die Neuronen unterschiedliche Startparameter, was durch das Initialisieren mit zufällig gewählten Zahlen erreicht wird.

In dieser Arbeit verwende ich die Form der Initialisierung, welche im Buch «Neural Networks and Deep Learning» von Michael Nielsen empfohlen wird.[14] Sowohl Gewichte als auch Neigungen werden nach einer Normalverteilung mit Mittelwert 0 initialisiert. Die Varianz der Gewichte einer Schicht wird so skaliert, dass die Summe der Zufallsvariablen der Gewichte eine Varianz von 1 hat.[15] Die Neigungen werden mit Varianz 1 initialisiert, so dass sie die

[14] Andere Initialisierungen basieren ebenfalls auf Normalverteilungen, aber mit anderen Varianzen. Die Varianzen beziehen zudem die Anzahl Neuronen der nächsten Schicht mit ein und unterscheiden sich je nach Aktivierungsfunktion um einen Vorfaktor. (Géron, Xavier and He Initialization, 2017) Angesichts dessen ist diese Initialisierung kaum optimal, aber gut genug.

[15] Die Varianz der Summe von normalverteilten Zufallsvariablen ist die Summe der Varianzen der Summanden (siehe *https://en.wikipedia.org/wiki/Sum_of_normally_distributed_random_variables*).

gleiche Verteilung wie die Summe der Gewichte haben und deshalb am Anfang im Durchschnitt etwa den gleichen Einfluss auf die gewichtete Summe haben.

$$w_{t=0}^{l} \sim N\left(\mu = 0, \sigma^2 = \frac{1}{|l|}\right) \tag{17}$$

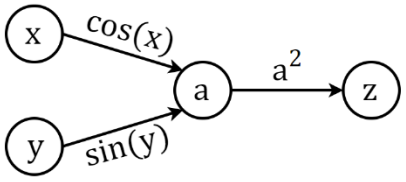
$$b_{t=0} \sim N(\mu = 0, \sigma^2 = 1) \tag{18}$$

(Nielsen, Weight initialization, 2017)

3.2.7 Backpropagation

Um SGD auf neuronale Netze anzuwenden, benötigt man Ausdrücke für alle $\frac{\partial C}{\partial w_{n,m}^{l}}$ und $\frac{\partial C}{\partial b_{m}^{l}}$, um nach Gleichung 10 die Gewichte und Neigungen mit dem Durchschnitt dieser $\frac{\partial C}{\partial w_{n,m}^{l}}$ und $\frac{\partial C}{\partial b_{m}^{l}}$ zu aktualisieren.

Für die Herleitung dieser Ausdrücke stellt man ein neuronales Netz als Computationgraph dar (vgl. nächste Abbildung). In einem Computationgraph sind die Knoten Variablen, während die Kanten diese Variablen (hier von links nach rechts) abbilden. Wenn mehrere Kanten auf einen Knoten treffen, werden die Eingänge addiert. Wenn mehrere Kanten von einem Knoten ausgehen, wird immer die Variable des Knotens weitergegeben. In Computationgraphs lassen sich Verkettungen von Funktionen übersichtlich darstellen, wodurch man einfach erkennen kann, wie Variablen voneinander abhängen und man partielle Ableitungen einfach mit der Kettenregel berechnen kann.



$$a(x,y) = \cos(x) + \sin(y)$$

$$z\big(a(x,y)\big) = a(x,y)^2 = (\cos(x) + \sin(y))^2$$

$$\frac{\partial z}{\partial x} = \frac{\partial a}{\partial x} * \frac{\partial z}{\partial a} = -\sin(x) * 2a$$

$$\frac{\partial z}{\partial y} = \frac{\partial a}{\partial y} * \frac{\partial z}{\partial a} = cos(x) * 2a$$

Abbildung 14: Beispiel eines Computationgraphs

(Olah, Calculus on Computational Graphs: Backpropagation, 2015)

Backpropagation (zu Deutsch *Fehlerrückführung*) ist das Verfahren, mit dem man die partiellen Ableitungen nach den Parametern von neuronalen Netzen für die Anwendung von SGD bestimmt. Die namensgebende Idee des Verfahrens ist es, in der letzten Schicht mit der Berechnung der partiellen Ableitungen anzufangen und dann Schicht für Schicht während dem **Backward Pass** (engl. für *verkehrter Durchgang*) entgegen der Richtung des Feedforwards zu rechnen.

Im ersten Schritt des Backpropagation-Algorithmus werden die partiellen Ableitungen von C nach den gewichteten Summen aller Schichten $\frac{\partial C}{\partial z_{n}^{l}}$ berechnet, woraus in einem zweiten Schritt die partiellen Ableitungen nach den Gewichten und Neigungen berechnet werden. $\frac{\partial C}{\partial z_{n}^{l}}$ nennt man den **Fehler** (engl.: *error*) e_{n}^{l} über dem n-ten Neuron der l-ten Schicht.

$$e_n^l = \frac{\partial C}{\partial z_n^l} \tag{19}$$

Analog zur Notation von Aktivierungen gilt in der Vektorschreibweise:

$$e^l = \begin{pmatrix} \frac{\partial C}{\partial z_1^l} & \frac{\partial C}{\partial z_2^l} & \cdots & \frac{\partial C}{\partial z_{|l|}^l} \end{pmatrix} \tag{20}$$

Der Fehler in der letzten Schicht lässt sich mit der Kettenregel finden. In der nebenstehenden Abbildung ist in einem Computationgraph dargestellt, wie eine gewichtete Summe der letzten Schicht auf die Kosten wirkt. Es gilt:

$$e_n^L = \frac{\partial C}{\partial z_n^l} = \frac{\partial a_n^L}{\partial z_n^L} * \frac{\partial C}{\partial a_n^L}$$

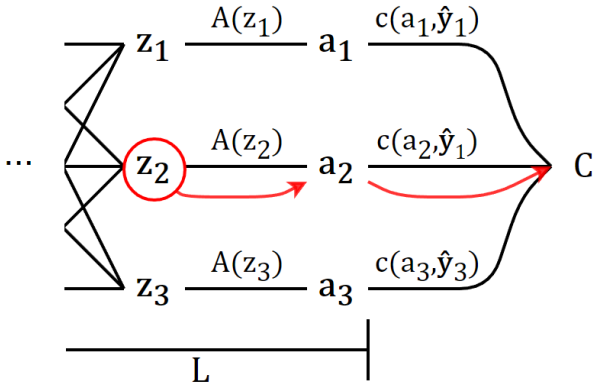


Abbildung 15: gewichtete Summe der letzten Schicht beeinflusst Kosten über Aktivierung

$\frac{\partial a_n^L}{\partial z_n^L}$ ist die Ableitung der Aktivierungsfunktion an der Stelle z_n^L, somit ist dieser Term $A'(z_n^L)$.
In der Komponentenschreibweise lässt sich der Fehler über der letzten Schicht schlussendlich so schreiben:

$$e_n^L = \frac{\partial C}{\partial a_n^L} * A'(z_n^L) \tag{21}$$

Um den Fehler in Vektorschreibweise auszudrücken, benötigt man eine Operation, welche zwei Vektoren elementweise multipliziert. Diese Operation ist bekannt als das Hadamard-Produkt. Als Operatorsymbol wird $\odot$ verwendet. (Wikipedia - Hadamard-Produkt, 2018)

$$a \odot b = (a_1 \quad a_2 \quad \cdots \quad a_n) \odot (b_1 \quad b_2 \quad \cdots \quad b_n) = (a_1 * b_1 \quad a_2 * b_2 \quad \cdots \quad a_n * b_n)$$

Mit $A'(z^l)$ als die vektorisierte Ableitung der Aktivierungsfunktion ergibt sich daraus der Fehler in der letzten Schicht:

$$e^L = \begin{pmatrix} \frac{\partial C}{\partial a_1^L} & \frac{\partial C}{\partial a_2^L} & \cdots & \frac{\partial C}{\partial a_{|L|}^L} \end{pmatrix} \odot A'(z^L) \tag{22}$$

Mit dem Fehler aus der letzten Schicht kann man den Fehler der vorherigen Schicht bestimmen. Die gewichtete Summe z_n^{l-1} beeinflusst die Aktivierung a_n^{l-1}, welche sich auf die Kosten auswirkt. Erneut gilt durch die Kettenregel:

$$e_n^{l-1} = \frac{\partial C}{\partial z_n^{l-1}} = \frac{\partial a_n^{l-1}}{\partial z_n^{l-1}} * \frac{\partial C}{\partial a_n^{l-1}}$$

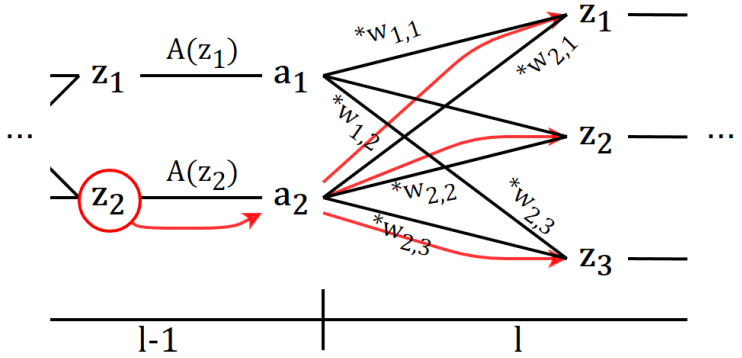


Abbildung 16: gewichtete Summe beeinflusst alle gewichteten Summen der nächsten Schicht über Aktivierung

Die Aktivierung a_n^{l-1} beeinflusst alle z_k^l der nächsten Schicht, daher ist $\frac{\partial C}{\partial a_n^{l-1}}$ die Summe aller $\frac{\partial z_k^l}{\partial a_n^{l-1}} * \frac{\partial C}{\partial z_k^l}$.

$$e_n^{l-1} = A'\left(z_n^{l-1}\right) * \sum_{m=1}^{|l|} \frac{\partial C}{\partial z_m^l} * \frac{\partial z_m^l}{\partial a_n^{l-1}}$$

Die Aktivierung a_n^{l-1} wird beim Bilden der m-ten gewichteten Summe mit $w_{n,m}^{l-1}$ multipliziert, somit ist $\frac{\partial z_m^l}{\partial a_n^{l-1}} = w_{n,m}^{l-1}$. Der Term $\frac{\partial C}{\partial z_m^l}$ ist per Definition e_m^l.

$$e_n^{l-1} = A'\left(z_n^{l-1}\right) * \sum_{m=1}^{|l|} w_{n,m}^{l-1} * e_m^l \tag{23}$$

Mit diesen Formeln kann nun jedes e_n^l berechnet werden, in der letzten Schicht mit Formel 21 und in allen vorherigen Schichten rekursiv mit Formel 23.

Formel 23 in Vektorschreibweise besteht aus dem Hadamard-Produkt eines Vektors $A'\left(z^{l-1}\right) = \begin{pmatrix} A'\left(z_1^{l-1}\right) & A'\left(z_2^{l-1}\right) & \cdots \end{pmatrix}$ und eines anderen, dessen Elemente die Summen aus Formel 23 sind.

$$e^{l-1} = A'\left(z^{l-1}\right) \odot \begin{pmatrix} \sum_{m=1}^{|l|} w_{1,m}^{l-1} * e_m^l & \cdots & \sum_{m=1}^{|l|} w_{|l-1|,m}^{l-1} * e_m^l \end{pmatrix}$$

Der zweite Vektor ist das Produkt der Matrixmultiplikation des Fehlers in der Schicht l und der transponierten Gewichtsmatrix der Schicht $l-1$.

$$\begin{pmatrix} \sum_{m=1}^{|l|} w_{1,m}^{l-1} * e_m^l & \cdots & \sum_{m=1}^{|l|} w_{|l-1|,m}^{l-1} * e_m^l \end{pmatrix} = \begin{pmatrix} e_1^l & \cdots & e_{|l|}^l \end{pmatrix} * \begin{pmatrix} w_{1,1}^{l-1} & w_{2,1}^{l-1} & \cdots & w_{|l-1|,1}^{l-1} \\ w_{1,2}^{l-1} & w_{2,2}^{l-1} & \cdots & w_{|l-1|,2}^{l-1} \\ \vdots & \vdots & \ddots & \\ w_{1,|l|}^{l-1} & w_{2,|l|}^{l-1} & & w_{|l-1|,|l|}^{l-1} \end{pmatrix}$$

$$= e^l * \begin{pmatrix} w_{1,1}^{l-1} & w_{1,2}^{l-1} & \cdots & w_{1,|l|}^{l-1} \\ w_{2,1}^{l-1} & w_{2,2}^{l-1} & \cdots & w_{2,|l|}^{l-1} \\ \vdots & \vdots & \ddots & \\ w_{|l-1|,1}^{l-1} & w_{|l-1|,2}^{l-1} & & w_{|l-1|,|l|}^{l-1} \end{pmatrix}^T = e^l * {W^{l-1}}^T$$

Zusammengefasst ist die Formel in der Vektorschreibweise für einen Fehler basierend auf dem Fehler in der nächsten Schicht folgende:

$$e^{l-1} = A'\left(z^{l-1}\right) \odot \left(e^l * {W^{l-1}}^T\right) \tag{24}$$

Nun braucht man Formeln, mit denen der Gradient aus den Fehlern bestimmt werden kann. Für die partielle Ableitung der Kosten nach einer Neigung gilt:

$$\frac{\partial C}{\partial b_n^l} = \frac{\partial C}{\partial z_n^{l+1}} * \frac{\partial z_n^{l+1}}{\partial b_n^l}$$

$\frac{\partial C}{\partial z_n^{l+1}}$ ist per Definition e_n^{l+1}, welches durch die vorherigen Formeln berechnet werden kann und $\frac{\partial z_n^{l+1}}{\partial b_n^l}$ ist 1, da die Summe z_n^{l+1} aus b_n^l und ansonsten zu b_n^l konstanten Termen besteht.

$$\frac{\partial C}{\partial b_n^l} = \frac{\partial C}{\partial z_n^{l+1}} * \frac{\partial z_n^{l+1}}{\partial b_n^l} = e_n^{l+1} * 1 = e_n^{l+1} \tag{25}$$

Es ergibt sich folgender Gradient der Kosten respektive der Neigungen einer Schicht:

$$\nabla \mathrm{C}_{b^l} = \begin{pmatrix} \frac{\partial C}{\partial b_1^l} & \cdots & \frac{\partial C}{\partial b_{|l|}^l} \end{pmatrix} = \begin{pmatrix} e_1^{l+1} & \cdots & e_{|l+1|}^{l+1} \end{pmatrix} = e^{l+1} \tag{26}$$

Die partiellen Ableitungen der Kosten nach einem Gewicht sind:

$$\frac{\partial C}{\partial w_{n,m}^l} = \frac{\partial C}{\partial z_m^{l+1}} * \frac{\partial z_m^{l+1}}{\partial w_{n,m}^l} = e_m^{l+1} * a_n^l \tag{27}$$

In Matrixform ist die Gleichung für den Gradienten der Kosten respektive einer Gewichtsmatrix:

$$\nabla \mathrm{C}_{W^l} = \begin{pmatrix} \frac{\partial C}{\partial w_{1,1}^l} & \frac{\partial C}{\partial w_{1,2}^l} & \cdots & \frac{\partial C}{\partial w_{1,|l|}^l} \\ \frac{\partial C}{\partial w_{2,1}^l} & \frac{\partial C}{\partial w_{2,2}^l} & \cdots & \frac{\partial C}{\partial w_{2,|l|}^l} \\ \vdots & \vdots & \ddots & \\ \frac{\partial C}{\partial w_{|l-1|,1}^l} & \frac{\partial C}{\partial w_{|l-1|,2}^l} & & \frac{\partial C}{\partial w_{|l-1|,|l|}^l} \end{pmatrix} = {a^l}^T * e^{l+1} \tag{28}$$

Der Backpropagation-Algorithmus sieht zusammengefasst wie folgt aus:

1. Fehler in der letzten Schicht mit Formel 22 berechnen
2. Für jede vorherige Schicht in umgekehrter Reihenfolge:
    - Fehler in der Schicht mit Formel 24 berechnen
3. Für jede Schicht:
    - Gradientvektor der Kosten respektive der Neigungen der Schicht mit Formel 26 berechnen
    - Gradientmatrix der Kosten respektive der Gewichte der Schicht mit Formel 28 berechnen

(Nielsen, Backpropagation, 2017)

3.2.8 Vanishing-Gradient-Problem

Die Kombination der Sigmoid-Aktivierungsfunktion mit der MSE-Kostenfunktion hat einen bedeuteten Nachteil gegenüber anderen Kombinationen von Aktivierungs- und Kostenfunktionen. Um den Nachteil aufzuzeigen, werden die im letzten Kapitel hergeleiteten Backpropagation-Gleichungen zur Berechnung des Gradienten etwas genauer untersucht.

Der Fehler der letzten Schicht eines neuronalen Netzes ist gegeben durch Formel 21:

$$e_n^L = \frac{\partial C}{\partial a_n^L} * A'(z_n^L)$$

Die partielle Ableitung der MSE-Kostenfunktion nach einer der Aktivierungen der letzten Schicht ist bestimmt durch:

$$\frac{\partial C}{\partial a_n^L} = \frac{\partial \left(\sum_{m=1}^{|L|} 0.5(a_m^L - \hat{y}_m)^2\right)}{\partial a_n^L}$$

$$= \frac{\partial 0.5(a_1^L - \hat{y}_1)^2}{\partial a_n^L} + \cdots + \frac{\partial 0.5(a_n^L - \hat{y}_n)^2}{\partial a_n^L} + \cdots + \frac{\partial 0.5\left(a_{|L|}^L - \hat{y}_{|L|}\right)^2}{\partial a_n^L}$$

$$= 0^{16} + \cdots + a_n^L - \hat{y}_n + \cdots + 0 = a_n^L - \hat{y}_n \qquad (29)$$

Dadurch ergibt sich für den Fehler in der letzten Schicht eines Sigmoid-Netzes, welches mit dem MSE trainiert wird, folgendes:

$$e_n^L = (a_n^L - \hat{y}_n) * \sigma'(z_n^L)$$

$\sigma'(z_n^L)$ wird bei fortgeschrittenem Training klein (quasi 0), da die Neuronen der letzten Schicht quasi gezwungen werden, bei jeder Vorhersage sehr grosse oder kleine z_n^L zu haben, um so gut wie möglich mit einem OneHot-Label übereinzustimmen. Bei saturierten Neuronen (also bei Neuronen, deren Aktivierung bei beinahe allen Beispielen des Trainingsdatensatzes nahe an 0 oder 1 ist) liegt z_n^L deshalb bei vielen Beispielen des Trainingsdatensatzes im flachen Bereich der Sigmoidfunktion, wodurch e_n^L immer klein ist.

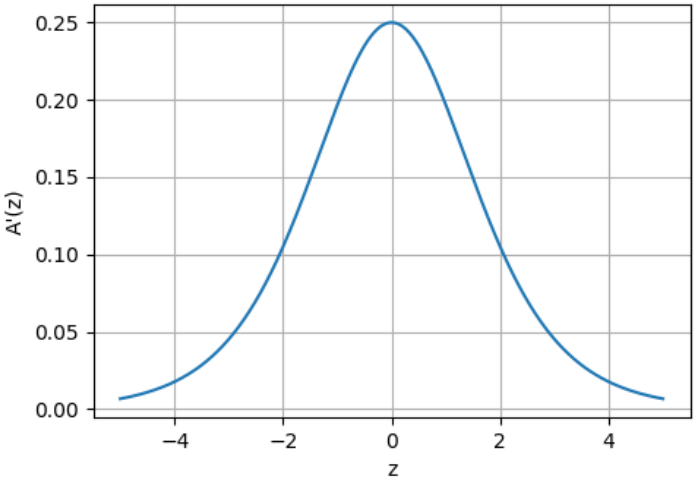


Abbildung 17: Ableitung der Sigmoidfunktion

Für jede vorherige Schicht gilt nach der Formel 23:

$$e_n^{l-1} = A'\left(z_n^{l-1}\right) * \sum_{m=1}^{|l|} w_{n,m}^{l-1} * e_m^l = \sigma'\left(z_n^{l-1}\right) * \sum_{m=1}^{|l|} w_{n,m}^{l-1} * e_m^l$$

[16] Nur einer der Summanden beinhaltet a_n^L, alle anderen sind Konstanten und fallen weg.

Wenn viele Neuronen der l-ten Schicht saturiert sind, werden deren Fehler e_m^l klein, wie es in der letzten Schicht bei fortgeschrittenem Training der Fall ist. Alle Fehler einer vorherigen Schicht müssen aufgrund der obigen Formel ebenfalls klein sein. Die Konsequenz einer Schicht aus saturierten Neuronen ist also, dass ihre Fehler und alle Fehler von vorangehenden Schichten ebenfalls klein sind. Falls die Fehler einer folgenden Schicht l nicht schon klein sind, kann der Fehler e_n^{l-1} durch die Multiplikation mit $\sigma'(z_n^{l-1})$ dennoch klein werden, falls z_n^{l-1} gross oder klein ist.

Problematisch an verschwindend kleinen Fehlern ist, dass die partiellen Ableitungen nach den Gewichten und Neigungen (siehe Formeln 25 und 27) dann ebenfalls klein sind. Dadurch werden die Gewichte und Neigungen pro Aktualisierung durch SGD nur minim geändert und das Netz lernt langsamer, als es müsste. Dieses Problem ist unter dem Namen **Vanishing-Gradient-Problem** (engl. für *verschwindender-Gradient-Problem*, kurz VGP) bekannt. (Wikipedia - Vanishing gradient problem, 2018)

Langsames Lernen ist nicht nur bei fortgeschrittenem Training ein Problem. Wenn die Varianz der Verteilung, aus der die Modellparameter einer Schicht zufällig initialisiert werden, unabhängig von der Anzahl der Inputs (Anzahl der Neuronen der vorherigen Schicht) ist, so können die gewichteten Summen in Schichten mit vielen Inputs anfangs enorm gross oder negativ werden. Dadurch ergeben sich erneut kleine $\sigma'(z_n^l)$. Eine gute Initialisierung muss deshalb nicht nur die Symmetrie des Netzes brechen, sondern auch verhindern, dass die Varianz der gewichteten Summen zu gross wird.

3.2.9 Kreuzentropie-Kostenfunktion

Eine Möglichkeit, das VGP anzugehen, ist die Wahl einer Kostenfunktion, die zusammen mit der Sigmoid-Aktivierungsfunktion den $\sigma'(z_n^L)$-Term aus dem Fehler der letzten Schicht eliminiert. Die Kostenfunktion müsste folgende Bedingung erfüllen:

$$e_n^L = \frac{\partial C}{\partial a_n^L} * \sigma'(z_n^L) = a_n^L - \hat{y}_n$$

Da die Aktivierungsfunktion die Sigmoidfunktion ist, kann man nach Gleichung 16 $\sigma'(z_n^L)$ mit $a_n^L * (1 - a_n^L)$ ersetzen. Der Einfachheit halber werden a_n^L und $\hat{y}_n$ für diese Herleitung a und respektive $\hat{y}$ genannt. Wenn man die Gleichung nach $\frac{\partial C}{\partial a}$ umstellt, erhält man:

$$\frac{\partial C}{\partial a} = \frac{a - \hat{y}}{a * (1 - a)}$$

Da a alleine nur die Kosten auf Ebene eines Outputs c beeinflusst und C als die Summe aller c definiert ist, und nur eines der c von einer der Aktivierungen abhängt, ist $\frac{\partial C}{\partial a}$ das Gleiche wie $\frac{dc}{da}$. Wenn man beide Seiten mit da multipliziert und integriert erhält man:

$$\int dc = \int \frac{a - \hat{y}}{a * (1 - a)} da$$

Die linke Seite der Gleichung ist c. Das Integral auf der rechten Seite kann durch eine Partialbruchzerlegung gelöst werden:

$$\frac{a-\hat{y}}{a*(1-a)} = \frac{A}{a} + \frac{B}{(1-a)}$$

$$a-\hat{y} = A*(1-a) + B*a = (-A+B)a + A$$

$$A = -\hat{y}, \qquad B = 1-\hat{y}$$

$$c = \int \frac{a-\hat{y}}{a*(1-a)}da = \int \left(\frac{-\hat{y}}{a} + \frac{1-\hat{y}}{(1-a)}\right)da = -\hat{y}\ln(a) - (1-\hat{y})\ln(1-a) + C$$

Bei Kostenfunktionen macht die Addition einer Konstante beim Training keinen Unterschied, da man das Minimum sucht, unabhängig davon, wie hoch es ist. Deshalb kann man die Integrationskonstante auf 0 setzen. Die Kosten auf der Ebene einer Vorhersage sind deshalb wie folgt:

$$C = -\sum_{n=1}^{k} \hat{y}_n \ln(a_n) + (1-\hat{y}_n)\ln(1-a_n) \tag{30}$$

Diese Kostenfunktion nennt man die Kreuzentropie (engl.: *cross entropy*), welche u.a. in der Informationstheorie eine Rolle spielt.

Die Kreuzentropie-Kostenfunktion macht nur in Kombination mit Aktivierungsfunktionen Sinn, die $A'(z) = \big(1-A(z)\big) * A(z)$ erfüllen, da dies die einzige Annahme über die Aktivierungsfunktion bei der Herleitung war.

Setzt man $\hat{y}_n = a_n$, so ergibt sich für die Kosten auf Ebene eines Outputs $-(\hat{y}_n \ln(\hat{y}_n) + (1-\hat{y}_n)\ln(1-\hat{y}_n))$, was für $\hat{y}_n = a_n \in (0{,}1)$ immer > 0 ist, aber der Grenzwert für $\hat{y}_n = a_n$ gegen 0 und 1 ist 0. Dadurch kann $-(\hat{y}_n \ln(a_n) + (1-\hat{y}_n)\ln(1-a_n))$ nur minimal sein, wenn $\hat{y}_n = a_n = 0$ oder $\hat{y}_n = a_n = 1$. Dies ist ein Problem, wenn $\hat{y}_n$ irgendeine Zahl sein kann (besonders, wenn sie nicht zwischen 0 und 1 liegt), da dann c nicht mehr je kleiner ist, desto kleiner $|a_n - \hat{y}_n|$. Bei Klassifikation mit One-Hot-Labels spielt das aber keine Rolle, da $\hat{y}_n$ sowieso nur 0 oder 1 sein kann.

Die partielle Ableitung der Kreuzentropie-Kostenfunktion nach einer der Aktivierungen wurde bereits in der Herleitung ersichtlich, denn es ist der Integrand, aus dem die Funktion hergeleitet wurde:

$$\frac{\partial C}{\partial a_n^L} = \frac{1-\hat{y}_n}{(1-a_n^L)} - \frac{\hat{y}_n}{a_n^L} \tag{31}$$

(Nielsen, The cross-entropy cost function, 2017)

Die Kreuzentropie-Kosten sind bei gleichen Vorhersagen und Labeln grösser (und in den Fällen $y = \hat{y} = 0$ und $y = \hat{y} = 1$ gleich) als die MSE-Kosten. Um zwei Netze zu vergleichen, welche mit unterschiedlichen Kostenfunktionen trainiert wurden, müssen die Kosten des einen Netzes auch mit der Kostenfunktion des anderen gemessen werden.

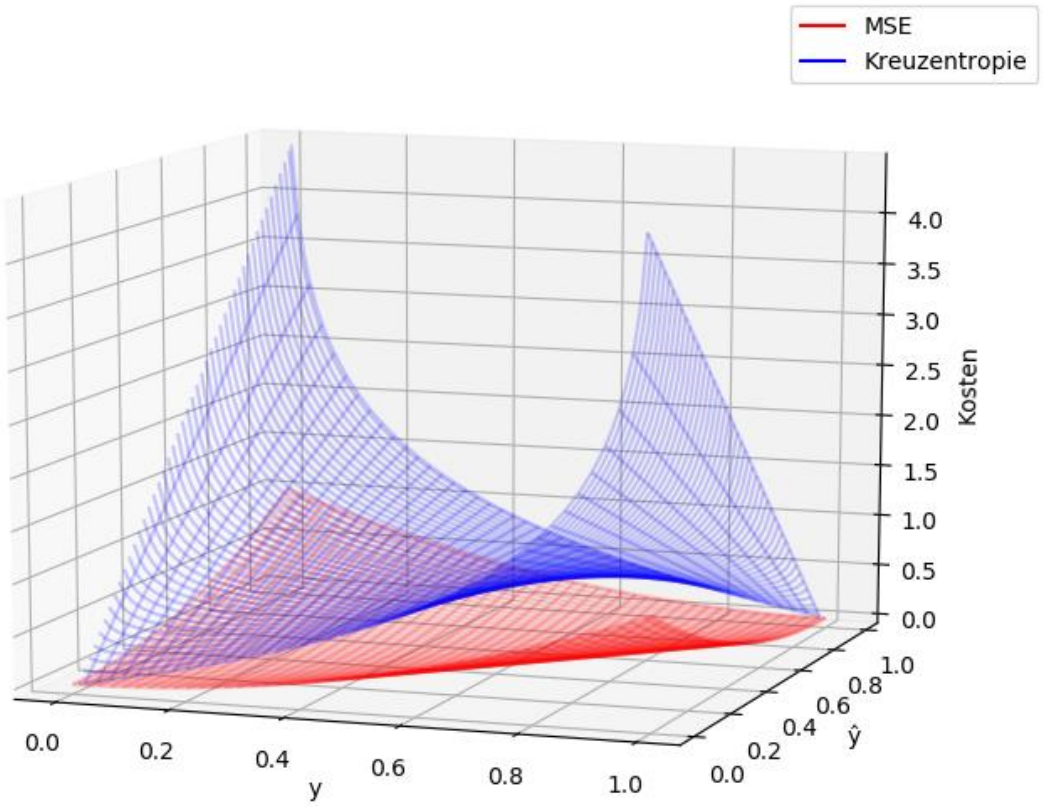


Abbildung 18: MSE und Kreuzentropie im Vergleich

Mit der Kreuzentropie-Kostenfunktion eliminiert man den $\sigma'(z_n^L)$-Term im Fehler der letzten Schicht, aber in den vorherigen Schichten ist er nach Formel 23 vorhanden. Dadurch kann der Gradient in vorderen Schichten dennoch klein werden, wodurch Netze mit vielen Schichten nur bedingt von der Kombination der Kreuzentropie-Kostenfunktion mit der Sigmoid-Aktivierungsfunktion profitieren.

3.2.10 ReLU

Eine andere Möglichkeit, das Vanishing-Gradient-Problem zu lösen, ist die Wahl einer Aktivierungsfunktion, deren Ableitung konstant ist. Damit würden alle $A'(z^l)$ während des Trainings sicher nicht kleiner werden, was in der Kombination mit dem MSE wie mit der Sigmoid-Aktivierungsfunktion und den Kreuzentropie-Kosten zu $e_n^L = (a_n^L - \hat{y}_n)$ führt. Das Problem daran ist, dass nur lineare Funktionen diese Eigenschaften besitzen, und wie man in 3.2.5 gesehen hat, kann ein neuronales Netz mit einer linearen Aktivierungsfunktion nur linear separierbare Datensätze lernen. Die Lösung des Problems ist eine teils lineare Funktion, wie zum Beispiel der positive Teil der Einheitsfunktion:

$$A(z) = \begin{cases} z, & z > 0 \\ 0, & z \leq 0 \end{cases} = \max(z, 0) = z^+ \tag{32}$$

Diese Funktion nennt man einen *Rectifier* (in Anlehnung an den Gleichrichter in der Elektrotechnik, der auf Englisch den gleichen Namen hat). Man hört aber viel häufiger von einer *Rectified Linear Unit* (kurz ReLU), was eigentlich der Begriff eines Neurons (auf engl. auch *unit* genannt) mit einem Rectifier als Aktivierungsfunktion ist, aber fälschlicherweise auch als Name der Aktivierungsfunktion verwendet wird.

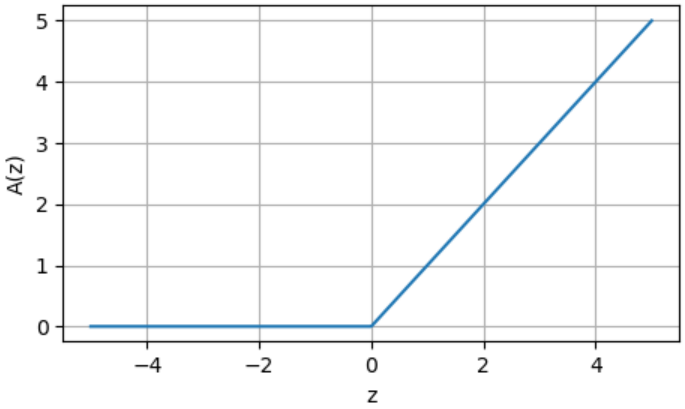


Abbildung 19: Rectifier-Aktivierungsfunktion

Die Rectifier-Aktivierungsfunktion ist nach rechts unbeschränkt und kann Funktionswerte grösser als 1 haben. Bei der Sigmoid-Aktivierungsfunktion ist das nicht so: Je grösser z_n^L, desto näher ist $a_n^L < 1$ an 1, umso geringer sind die Kosten bei $\hat{y}_n = 1$. Bei der Rectifier-Aktivierungsfunktion stimmt das nur für $z_n^L \leq 1$, denn je grösser $z_n^L > 1$, desto grösser $a_n^L > 1$, umso grösser die Abweichung zwischen a_n^L und $\hat{y}_n = 1$. Das macht es einem Netz «schwer», geringe Kosten zu erzielen, da z_n^L nicht irgendeine grosse Zahl sein darf, wenn $\hat{y}_n$ 1 ist, sondern so nahe wie möglich an 1 sein muss. Deshalb eignet sich die Rectifier-Aktivierungsfunktion nur bedingt als Aktivierungsfunktion der letzten Schicht.

Die Ableitung des Rectifiers ist in $z = 0$ eigentlich undefiniert. Da eine gewichtete Summe in einem neuronalen Netz aber theoretisch 0 sein kann, sollte man diese Stelle für die Anwendung ebenfalls definieren. Deshalb ergibt sich für die Ableitung der Rectifier-Funktion folgende Stufenfunktion:

$$A'(z) = \begin{cases} 1, & z > 0 \\ 0, & z \leq 0 \end{cases} \tag{33}$$

Leaky ReLU

Problematisch an ReLUs ist der konstante Teil der Aktivierungsfunktion bei negativen gewichteten Summen, durch den zwei Probleme auftreten können:

- Die Ableitung des Rectifiers ist im negativen Teil 0, wodurch der Gradient dennoch klein sein kann und die Parameter nur geringfügig geändert werden (abhängig davon, wie viele der Beispiele des Trainingsdatensatzes zu einem negativen Input führen).
- Haben alle Outputneuronen der letzten Schicht bei jedem Beispiel des Trainingsdatensatzes eine negative gewichtete Summe, so ist der Gradient jedes Parameters 0 und das gesamte Netz verbessert sich nicht mehr. Wenn einzelne Neuronen bei vielen Beispielen des Trainingsdatensatzes stark negative gewichtete Summen haben, wodurch die Gewichte, welche diese gewichtete Summe bilden, nicht mehr verändert werden, dann bleibt das Neuron stecken.

Die Probleme löst man, indem man den konstanten Teil der Rectifier-Aktivierungsfunktion durch eine lineare Funktion mit geringer Steigung $\alpha > 0$ ersetzt.[17] Die Ableitung im negativen

[17] α wird in dieser Arbeit als Hyperparameter vor dem Training festgelegt, könnte aber auch während dem Training mit SGD optimiert werden.

Bereich ist zwar immer noch klein, aber da sie nicht 0 ist, können sich steckengebliebene Neuronen befreien.

$$A(z) = \begin{cases} z, & z > 0 \\ \alpha z, & z \leq 0 \end{cases} \tag{34}$$

Ein Neuron mit dieser Aktivierungsfunktion ist eine *leaky* (engl. für undicht) *ReLU* (kurz: lReLU).

Erneut sollte man die Ableitung in $z = 0$ definieren. Dadurch ergibt sich folgende Ableitung des leaky Rectifiers:

$$A'(z) = \begin{cases} 1, & z > 0 \\ \alpha, & z \leq 0 \end{cases} \tag{35}$$

(Wikipedia - Rectifier (neural networks), 2018)

3.2.11 Softmax-Aktivierungsfunktion

Die Funktionsgleichung der Softmax-Aktivierungsfunktion ist folgende:

$$A\left(z_n^l\right) = \frac{e^{z_n^l}}{\sum_{m=1}^{|l|} e^{z_m^l}} \tag{36}$$

Die Softmax-Aktivierungsfunktion teilt ähnlich wie die Sigmoid-Aktivierungsfunktion den gewichteten Summen eine Zahl zwischen 0 und 1 zu. Die Aktivierung eines Neurons hängt aber neben der eigenen gewichteten Summe von allen anderen gewichteten Summen der selber Schicht ab. Die Softmax-Funktion ist eine normalisierte Exponentialfunktion, d.h. dass die Summe aller Aktivierungen einer Schicht 1 ergibt:

$$\sum_{n=1}^{|l|} A\left(z_n^l\right) = \sum_{n=1}^{|l|} \frac{e^{z_n^l}}{\sum_{m=1}^{|l|} e^{z_m^l}} = \frac{\sum_{n=1}^{|l|} e^{z_n^l}}{\sum_{m=1}^{|l|} e^{z_m^l}} = 1$$

Da die Softmax-Funktion zudem nur positive Funktionswerte hat, lassen sind die Softmax-Aktivierungen in der letzten Schicht als Werte einer Wahrscheinlichkeitsverteilung ansehen, welche die Wahrscheinlichkeit angibt, mit der die Features einer bestimmten Klasse angehören.

Durch das Exponenzieren werden Unterschiede zwischen den gewichteten Summen verstärkt, wodurch das grösste z_n^l nach dem Exponenzieren beinahe so gross wie die Summe im Nenner der Softmax-Funktion ist. Die resultierende Softmax-Aktivierung des Neurons ist daher etwas unter 1. Bei allen anderen Neuronen, deren exponenzierte gewichtete Summe deutlich kleiner als die Summe im Nenner ist, ergibt sich eine kleine positive Zahl als Aktivierung. Eine Eigenschaft der Softmax-Funktion ist es also, die grösste gewichtete Summe «auszusuchen». Aufgrund dieser Eigenschaft eignet sich die Softmax-Funktion allerdings nur als Aktivierungsfunktion der letzten Schicht. Durch das «Vergleichen» aller gewichteten Summen einer Schicht können Softmax-Aktivierungen näher an den Werten eines One-Hot-Labels liegen als Sigmoid-Aktivierungen.

Die Softmax-Funktion eignet sich zudem nur zur Klassifikation, wenn Features ausschliesslich einer Klasse angehören können, da nur eine der Aktivierungen der letzten Schicht nahe bei 1 sein kann.

Zum Bestimmen der Ableitung der Softmax-Funktion definiere ich S als die Summe aller $e^{z_m^l}$, die zu z_n^l konstant sind:

$$S = \sum_{m \neq n} e^{z_m^l}$$

Damit wird die Softmax-Funktion zu:

$$A\left(z_n^l\right) = \frac{e^{z_n^l}}{e^{z_n^l} + S}$$

Mit der Quotientenregel ergibt sich dann die Ableitung als:

$$A'(z_n) = \frac{e^{z_n}(e^{z_n} + S) - e^{z_n} * e^{z_n}}{(e^{z_n} + S)^2} = \frac{e^{z_n} * e^{z_n} + e^{z_n} * S - e^{z_n} * e^{z_n}}{(e^{z_n} + S)^2} = \frac{e^{z_n}}{e^{z_n} + S} * \frac{S}{e^{z_n} + S}$$

$$= A(z_n) * (1 - A(z_n)) \quad (37)$$

Da sich die Ableitung der Softmax-Funktion gleich wie die Sigmoid-Funktion als $A(z_n) * (1 - A(z_n))$ ausdrücken lässt, ergibt sich in Kombination mit der Kreuzentropie-Kostenfunktion ebenfalls $e_n^L = a_n^L - \hat{y}_n$. Die Softmax-Aktivierungsfunktion sollte also in Kombination mit der Kreuzentropie-Kostenfunktion ähnlich wenig vom VGP wie die Kombination aus Sigmoid-Aktivierungsfunktion und Kreuzentropie-Kostenfunktion betroffen sein.

(Wikipedia - Softmax function, 2018)

4 Programm

Das Produkt dieser Arbeit ist ein Programm, mit dem man beliebige neuronale Netze erstellen und trainieren kann. Dabei soll man die Grösse des Netzes, die Aktivierungsfunktion in jeder Schicht und die Kostenfunktion, sowie die Lernrate und die Mini-Batch-Grösse wählen können. Das Programm soll es ebenfalls ermöglichen, Daten des Lernprozesses zu erfassen und darzustellen.

Das Programm soll Deep Learning anschaulich darstellen und das Experimentieren mit neuronalen Netzen ermöglichen. Das Ziel davon ist es nicht unbedingt, die beste Performance zu bieten oder alle Features zu beinhalten, die bei professioneller Anwendung von Deep Learning nützlich sind.

Als Programmiersprache habe ich Python 3[18] gewählt, da ich mich genug in ihr für ein solches Projekt auskenne und der Code für Unerfahrene einfach lesbar ist. Zusätzlich verwende ich zwei Packages, welche nicht in der Standardinstallation von Python enthalten sind:

- NumPy implementiert ein Vektor/Matrixobjekt, auf das Matrixoperationen angewendet werden können, und weitere nützliche Funktionen, wie z.B. das Generieren von (Pseudo)Zufallszahlen.
- Matplotlib dient dem Erstellen von Plots und Grafiken.

Es wird davon ausgegangen, dass der Leser die Grundkonzepte des Programmierens wie Schlaufen und If/Else-Bedingungen versteht. Python ist eine relativ einfach lesbare Sprache und sollte für Personen mit Programmiererfahrung ohne spezifische Kenntnisse in Python verständlich sein. Die Syntax und alles, was für Python und die verwendeten Packages spezifisch ist, wird dennoch im Anhang auf S. 66 (siehe 8.3) anhand kurzer Beispiele erklärt.

Die Inspiration zum Schreiben eines solchen Programms kommt aus dem Online-Buch *Neural Networks and Deep Learning* (*http://neuralnetworksanddeeplearning.com*) von Michael Nielsen, in dem der Autor die Funktionsweise von Deep Learning in einem Programm erläutert. Das folgende Programm wurde von mir selbst entwickelt.

Der gesamte Code, sowie das Resultat des Experiments stehen bei *https://tinyurl.com/MADeepLearning* zum Download zur Verfügung.

[18] Python Version 3.6.4, NumPy Version 1.14.0, Matplotlib Version 2.1.2

4.1 MNIST in Python

Der komplette MNIST-Datensatz, welcher auf *http://yann.lecun.com/exdb/mnist/* verfügbar ist, beinhaltet 70’000 Bilder von handgeschriebenen Ziffern, als Helligkeitswerte der 28x28 Pixel, und die zugehörigen Labels. 60'000 der Bilder werden zum Trainieren verwendet, die restlichen 10’000 dienen als Testdatensatz.

In folgenden Teil werden Funktionen erklärt, welche für den Umgang mit dem MNIST-Datensatz nützlich sind.

4.1.1 Daten laden und formatieren

Um die MNIST-Daten in Python bearbeiten zu können, verwende ich den `loader.py` von python-mnist-0.5 (*https://pypi.python.org/pypi/python-mnist*).

Die folgende Funktion lädt die MNIST-Testdaten. Ihr einziges Argument ist der Pfad des Ordners, welcher die Dateien der MNIST-Website enthält.

```
def training_data(datapath):[19]
    try:
        from loader import MNIST[20]
    except ModuleNotFoundError:
        raise ModuleNotFoundError('loader.py von python-mnist-0.5 ' +
            '(https://pypi.python.org/pypi/python-mnist) nicht gefunden!')

    mnist_data = MNIST(path=datapath, return_type='numpy', gz=True)
    images, labels = mnist_data.load_training()
```

`images` ist ein NumPy-Array (siehe 8.3.6) mit allen Bildern, die wiederum NumPy-(Zeilen)Vektoren der Helligkeitswerte aller 28x28 Pixel sind.

Diese Helligkeitswerte in `images` sind als Zahlen zwischen 0 und 255 gegeben, und nicht wie gewollt zwischen 0 und 1. Um das zu beheben, wird das gesamte `images`-Array durch 255 geteilt.

```
    images = np.float64(images)[21] / 255
```

Bei `labels` handelt es sich um ein NumPy-Array mit den einzelnen Labels, aber diese sind als Ziffern 0 bis 9 angegeben, deshalb müssen sie ins «One-Hot»-Format umgewandelt werden. Die `oh_labels`-Liste (siehe 8.3.3) wird nach und nach mit den One-Hot-Vektoren gefüllt.

[19] siehe 8.3.8 Funktionen
[20] `loader.py` muss sich im selben Verzeichnis befinden, in dem das Programm ausgeführt wird, um importiert werden zu können.
[21] NumPy-Arrays haben Datentypen, `images` wird in diesem Schritt vom Typ `int32` zu `float64` umgewandelt, einerseits um Fliesskommazahlen im Array zu ermöglichen und andererseits um sehr grosse Zahlen zuzulassen.

```
    oh_labels = []
    for l in labels:[22]
        z = np.zeros(10)
        z[l] = 1
        oh_labels.append(z)
```

Zuletzt werden Features und Labels zu einer Liste kombiniert. Die resultierende Liste `data`, welche pro Eintrag eine Liste (wird später Punkt genannt) mit einem Bild und zugehörigem «One-Hot»-Label besitzt, wird zurückgegeben.

```
    data = list(zip(images, oh_labels))
    print('*Trainingsdaten geladen')
    return data
```

Das Laden der Testdaten funktioniert genau gleich, nur dass die Zeile

```
images, labels = mnist_data.load_training()
```

durch

```
images, labels = mnist_data.load_testing()
```

ersetzt werden muss.

Das Problem an dieser Funktion ist, dass es relativ viel Zeit benötigt, um die Daten zu formatieren, und dies bei jeder Verwendung der MNIST-Daten neu gemacht werden müsste. Deshalb speichere ich die formatierten MNIST-Daten als Pickle-Dateien, welche schnell geladen werden können. Pickle (engl. für *einlegen* oder *einmachen*) ist ein Standardmodul von Python, mit dem man Python-Objekte komprimiert speichern kann.

```
import pickle
```

Die folgende Funktion lädt die MNIST-Daten und speichert sie als Pickle-Datei:

```
def toPKL(path):
    traindata = training_data(path)
    testdata = testing_data(path)

    with open(path + '/mnist_train.pkl', 'wb') as file:
            pickle.dump(traindata, file)

    with open(path + '/mnist_test.pkl', 'wb') as file:
            pickle.dump(testdata, file)
```

4.1.2 MNIST-Bilder darstellen

Mit der folgenden Funktion können MNIST-Bilder dargestellt werden. Der Funktionsparameter `image` ist ein NumPy-Vektor der Helligkeitswerte aller 28x28 Pixel eines MNIST-Bildes.

[22] siehe 8.3.4 For-Schlaufen

```
def draw(image, title=None):
```

Zum Darstellen eines Bildes aus den Pixelwerten wird die `imshow`-Funktion verwendet. Deren Input muss eine Matrix sein, deshalb wird `images` in eine Matrix umgewandelt. Mit `cmap=plt.cm.binary` wird eingestellt, dass die Pixelwerte des Inputs zwischen 0 und 1 liegen.

```
    pixels = image.reshape((28,28))
    plt.imshow(pixels, cmap=plt.cm.binary)
```

Nun werden die Achsenbeschriftungen vom Plot entfernt und, falls vorhanden, der Titel gesetzt.

```
    plt.xticks([])
    plt.yticks([])

    if title is not None:
        plt.title(title)

    plt.show()
```

4.2 Deep Learning in Python

Ein neuronales Netz wird als Objekt repräsentiert, welches die Aktivierungsfunktionen, die Gewichte und die Neigungen eines Netzes speichert. Ein solches `Network`-Objekt besitzt Methoden, mit denen man Vorhersagen berechnen und das Netz trainieren kann.

Während des Trainings können folgende Daten erfasst werden:

- Kosten während des Trainings pro Epoche
- MSE über Trainingsdaten nach jeder Epoche
- MSE über Testdaten nach jeder Epoche
- Klassifikationsgenauigkeit über Testdaten nach jeder Epoche
- Durchschnitt der Beträge aller Elemente des Gewichtsgradienten pro Schicht nach jedem Mini-Batch

4.2.1 Aktivierungsfunktionen

Die Aktivierungsfunktionen repräsentiere ich als Klassen (siehe 8.3.9 Klassen) mit statischen Methoden, welche die Aktivierung mit `A(z)` und die Ableitung der Aktivierungsfunktion mit `dA_dz(z)` berechnen.

Die Repräsentation durch eine Klasse hat den Vorteil, dass sowohl die Aktivierungsfunktion als auch deren Ableitung in nur einem Objekt zusammengefasst ist. So eine Klasse kann einer Methode als Parameter dienen, wobei in der Methode wahlweise die Aktivierungsfunktion oder deren Ableitung berechnet werden kann.

Im Falle der Sigmoid-Funktion sieht diese Klasse (basierend auf Formeln 15 und 16) mit `np.exp(<x>)` als Exponentialfunktion so aus:

```
class Sigmoid:

    @staticmethod
    def A(z):
        return 1/(1+np.exp(-z))
```

```
    @staticmethod
    def dA_dz(z):
        a = Sigmoid.A(z)
        return a*(1-a)
```

Beide Methoden funktionieren, wenn `z` ein NumPy-Vektor ist. Dies liegt daran, dass alle Operationen (auch `np.exp(<x>)`) elementweise durchgeführt werden.

Bei der Implementierung der Rectifier-Aktivierungsfunktion wird $A(z) = \max(z, 0)$ (Formel 32) verwendet. `np.maximum(<z>, 0)` wendet die Maximumsfunktion elementweise auf einen `<z>`-Vektor an. Die stückweise definierte Ableitung (Formel 33) wird mit `np.where(<bedingung>, <falls_wahr>, <sonst>)` umgesetzt, welches elementweise die Bedingung prüft und dort, wo sie wahr ist, `<falls_wahr>` einsetzt oder anderenfalls `<sonst>`.

```
class ReLU:

    @staticmethod
    def A(z):
        return np.maximum(z, 0)

    @staticmethod
    def dA_dz(z):
        return np.where(z > 0, 1, 0)
```

Die Aktivierungsfunktion einer lReLU (Formel 34) und deren Ableitung (Formel 35) sind ebenfalls stückweise definiert, deshalb kommt `np.where()` erneut zum Einsatz. Es wird $\alpha = 0.01$ verwendet:

```
class lReLU:

    @staticmethod
    def A(z):
        return np.where(z > 0, z, z * 0.01)

    @staticmethod
    def dA_dz(z):
        return np.where(z > 0, 1, 0.01)
```

Bei der Softmax-Funktion (Formel 36) wird die Exponentialfunktion elementweise auf den `z`-Vektor angewandt. Der entstandene Vektor wird zum Schluss durch dessen Summe geteilt und zurückgegeben. Die Ableitung der Softmax-Funktion wird nach Formel 37 über die nicht-abgeleitete Funktion berechnet.

```
class Softmax:

    @staticmethod
    def A(z):
        exp_z = np.exp(z)
        return exp_z / exp_z.sum()

    @staticmethod
    def dA_dz(z):
        a = Softmax.A(z)
        return a*(1-a)
```

Diese Implementierung der Softmax-Funktion ist nicht gegen einen Overflow (Fehler, wenn ein Wert grösser als maximal für den verwendeten Datentyp möglich wird) geschützt, der bei grossen gewichteten Summen in `np.exp(z)` oder der Summe davon auftreten könnte. Dies wird beim Verwenden des Programms kein Problem, wenn der Datentyp `np.float64` verwendet wird, der Werte bis maximal $1.797 * 10^{308}$ speichern kann.

4.2.2 Netzwerkobjekt initialisieren

Nun zur Klasse eines neuronalen Netzes:

```
class Network:
```

Beim Initialisieren eines Netzwerkobjekts muss angegeben werden, wie viele Neuronen sich in wie vielen Schichten befinden (in `size`; Bsp.: [784, 50, 10] bedeutet 784 Inputneuronen, ein Hidden Layer mit 50 Neuronen und eine Ausgabeschicht mit 10 Neuronen) und die Aktivierungsfunktionen (Aktivierungsfunktionsklassen, wie vorher definiert) in einer Liste, welche zwischen jeder Schicht verwendet werden sollen. `rdm_state` ist ein Zustand des Zufallsgenerators.

```
    def __init__(self, size, actfuncs, rdm_state=None):
        self.size = size
        self.actfuncs = actfuncs
```

Da die `size`-Liste sowohl die Input- als auch die Outputschicht beinhaltet, muss sie ein Element mehr als die Liste mit Aktivierungsfunktionen besitzen. Sollte das nicht der Fall sein, wird ein `TypeError` ausgelöst und das Programm stoppt.

```
        if len(size)!=len(actfuncs)+1:
            raise TypeError('len(size)!=len(actfuncs)+1')
```

Die Anzahl Schichten eines Objekts wird für spätere Methoden nützlich sein, daher wird sie in `L` gespeichert.

```
        self.L = len(self.size)
```

Falls ein Zufallszustand gegeben ist, wird er gesetzt. Falls nicht, wird ein neuer Zufallszustand verwendet.

```
        if rdm_state is not None:
            np.random.set_state(rdm_state)
        else:
            np.random.seed()
```

Um wiederholbare Resultate zu liefern, wird der Zufallszustand als Variable des Objekts gespeichert.

```
        self.rdm_state = np.random.get_state()
```

Mit dem folgenden Code werden die Gewichte und Neigungen nach Formeln 17 und 18 initialisiert. `np.random.normal(<erwartungswert>, <standardabweichung>, <shape>)` erstellt ein Array, dessen Elemente normalverteilt sind.

```
        sigmas = [1 / np.sqrt(i) for i in self.size]
        self.weights = [np.random.normal(0, sigmas[i],
                                         (self.size[i], self.size[i+1]))
                        for i in range(self.L-1)]
        self.biases = [np.random.normal(0, 1, self.size[i+1])
                       for i in range(self.L-1)]
```

Alle Aktivierungen und gewichteten Summen werden ebenfalls gespeichert, wenn eine Vorhersage berechnet wird, da diese Zahlen zum Berechnen des Gradienten benötigt werden. Die Listen werden vorerst mit `None` gefüllt, weil deren Elemente später neu zugewiesen werden und deshalb schon existieren müssen.

```
        self.activations = [None for i in range(self.L)]
        self.w_sums = [None for i in range(self.L)]
```

Nun folgen Listen, welche Daten während des Trainings erfassen und dabei nach und nach gefüllt werden.

```
        self.epochcosts = []
        self.accuracys = []
        self.mse_test = []
        self.mse_train = []
        self.gradient_sizes = []
        self.epochs_trained = 0
```

4.2.3 Vorhersage berechnen

Die Methode zur Berechnung der Vorhersage nimmt die Features als Parameter entgegen.

```
    def predict(self, features):
```

Die Aktivierung in der Inputschicht `activations[0]` entspricht den Features. `weightedsums[0]` bleibt `None`, da die Inputschicht keine gewichteten Summen hat.

```
        self.activations[0] = features
```

Nun folgt der Feedforward in Vektorform nach den Gleichungen 6 und 7. Hier sollte man beachten, dass die Klasse der Aktivierungsfunktion aus der `actfuncs`-Liste aufgerufen wird und dann davon die statische `A(z)`-Methode ausgeführt wird.

```
        for l in range(self.L-1):
            self.weightedsums[l+1] = (np.matmul(self.activations[l],
                                                self.weights[l])
                                      + self.biases[l])
            self.activations[l+1] = self.actfuncs[l].A(self.weightedsums[l+1])
```

Zuletzt wird die Vorhersage zurückgegeben.

```
        return self.activations[-1]
```

[23] siehe 8.3.5 List-Comprehensions

4.2.4 Kostenfunktionen in Python

Kostenfunktionen habe ich aus demselben Grund wie Aktivierungsfunktionen als Klassen umgesetzt. Die Kostenfunktionsklassen haben zwei statische Methoden, `C(a, y)` berechnet die Kosten auf Ebene einer Vorhersage C und die andere berechnet den $\left(\frac{\partial C}{\partial a_1^L} \quad \frac{\partial C}{\partial a_2^L} \quad \cdots \quad \frac{\partial C}{\partial a_{|L|}^L}\right)$-Vektor.

Es folgt die Klasse der MSE-Kostenfunktion, wobei Formeln 4 und 29 verwendet werden. `np.dot(<v1>, <v2>)` berechnet das Skalarprodukt.

```
class MSE:

    @staticmethod
    def C(y, y_hat):
        return 0.5 * np.dot(y-y_hat, y-y_hat)

    @staticmethod
    def dC_da(y, y_hat):
        return y - y_hat
```

Die Kreuzentropie wird nach den Formeln 30 und 31 berechnet. `np.log(<arg>)` berechnet elementweise den natürlichen Logarithmus des Arguments.

```
class CrossEntropy:

    @staticmethod
    def C(y, y_hat):
        c = -(y_hat*np.log(y)+(1-y_hat)*np.log(1-y))
        return c.sum()

    @staticmethod
    def dC_da(y, y_hat):
        return (1-y_hat)/(1-y) - y_hat/y
```

4.2.5 Gradient eines Mini-Batches berechnen und anwenden

Bevor man den Gradienten eines Mini-Batches berechnen kann, muss man die Trainingsdaten in Mini-Batches aufteilen. Das übernimmt folgende Funktion:

```
    @staticmethod
    def create_mini_batches(dataset, mini_batchsize):
        np.random.shuffle(dataset)
        mini_batches = [dataset[i:i+mini_batchsize]
                        for i in range(0, len(dataset), mini_batchsize)]

        return mini_batches
```

Die zurückgegebene Liste enthält die Mini-Batches als Listen, deren Elemente Listen mit Features und Labels sind.

Die Aktivierungen sind mit der Sigmoid-Aktivierungsfunktion Zahlen zwischen 0 und 1, aber nie exakt 0 oder 1. Es kann aber passieren, dass ein Modell so genau ist und die Aktivierungen der letzten Schicht so nahe an 0 oder 1 kommen, dass selbst mit dem Datentyp `float64` nicht mehr zwischen einer Aktivierung sehr nahe an 0 oder 1 und den

Zahlen 0 oder 1 unterschieden werden kann. Als Resultat kann es im Programm vorkommen, dass die letzten Aktivierungen exakt als 0 oder 1 gespeichert werden. Das wird zu einem Problem, weil man beim Berechnen der Kreuzentropie-Kosten den Logarithmus von 0 berechnen muss, welcher nicht definiert ist (Gleich darauf müsste zum Berechnen der partiellen Ableitung der Kreuzentropie-Kosten durch 0 geteilt werden).

Aus diesem Grund habe ich eine Methode geschrieben, die die Werte 0 und 1 aus den Aktivierungen der letzten Schicht mit Zahlen sehr nahe an 0 oder 1 ersetzt:

```
def clean_activations(self):
    self.activations[-1][self.activations[-1]==0] = 0.000000000001
    self.activations[-1][self.activations[-1]==1] = 0.999999999999
```

Nun zur Berechnung des Gradienten und dessen Anwendung auf die Parameter nach Gradient Descent:

```
def update_params(self, mini_batch, costfunc, learningrate):
```

Zuerst werden einige Listen erstellt. `errors` wird später mit e^l jeden Layers gefüllt. `gradient_w` und `gradient_b` sind Listen, deren Elemente die Summe der Gradienten der Gewichte und Neigungen aus Formeln 26 und 28 in den entsprechenden Schichten sind.

```
    self.errors = [None for i in range(self.L)]
    self.gradient_w = [np.zeros(i.shape) for i in self.weights]
    self.gradient_b = [np.zeros(i.shape) for i in self.biases]
    self.mini_batchcost = 0
```

In der folgenden Schlaufe wird jeder Punkt des Mini-Batches durchgegangen. Zuerst wird die Vorhersage des Punktes und dessen Kosten berechnet.

```
    for point in mini_batch:
        self.predict(point[0])
        self.clean_activations()
        self.mini_batchcost += costfunc.C(self.activations[-1], point[1])
```

Dann wird mit Formel 22 der Fehler in der letzten Schicht bestimmt.

```
        self.errors[-1] = (costfunc.dC_da(self.activations[-1], point[1])
                           * self.actfuncs[-1].fp(self.w_sums[-1]))
```

Darauf folgt der Backward Pass mit der Berechnung des Errors in den restlichen Schichten. Verwendet wird Formel 24.

```
        for i in reversed[24](range(2, self.L)):
            last_error = self.errors[i]
            transposed_w = np.transpose(self.weights[i-1])
            self.errors[i-1] = (self.actfuncs[i-2].fp(self.weightedsums[i-1])
                                * np.matmul(last_error, transposed_w))
```

Nun wird der Gradient aus e^l nach Formeln 26 und 28 berechnet. Um die Matrixmultiplikation eines Zeilenvektors mit einem Spaltenvektor aus Formel 28 im Programm durchzuführen, ist es notwendig, beide Vektoren in eine Matrix mit einer Zeile/Spalte umzuformen.

[24] Dadurch wird die Reihenfolge der Schlaufe umgekehrt.

```
            for i in range(self.L-1):
                a_transposed = self.activations[i].reshape((self.size[i], 1))
                error = self.errors[i+1].reshape((1, self.size[i+1]))
                self.gradient_w[i] += np.matmul(a_transposed, error)
                self.gradient_b[i] += self.errors[i+1]
```

Nachdem der Gradient eines Mini-Batches berechnet wurde, wird er nach der SGD-Gleichung 10 auf die Gewichte und Neigungen angewendet.

```
        for i in range(self.L-1):
            self.weights[i] -= learningrate / len(batch) * self.gradient_w[i]
            self.biases[i] -= learningrate / len(batch) * self.gradient_b[i]
```

4.2.6 Klassifikationsgenauigkeit bestimmen

Die Netzwerkklasse benötigt ein paar Methoden, um die Klassifikationsgenauigkeit eines Netzes zu messen.

`correct_classification(<prediction>, <label>)` gibt `True` zurück, wenn die Vorhersage stimmt, ansonsten `False`. Die Vorhersage ist korrekt, wenn der höchste Wert des Vorhersagevektors und des Labelvektors den gleichen Index im Array haben. Den Index des grössten Werts eines Arrays findet man mit der `np.argmax(<array>)`-Funktion.

```
    @staticmethod
    def correct_classification(prediction, label):
        if np.argmax(prediction) == np.argmax(label):
            return True
        else:
            return False
```

Diese statische Methode wird in der nächsten Methode verwendet, um die Klassifikationsgenauigkeit über einem Datensatz zu ermitteln. Zu jedem Datenpunkt im Datensatz wird eine Vorhersage erstellt, welche mit dem zugehörigen Label verglichen wird. Dabei wird die Anzahl richtig klassifizierter Beispiele erfasst.

```
    def classification_accuracy(self, dataset):
        n_correct = 0
        for d in dataset:
            p = self.predict(d[0])
            if Network.correct_classification(p, d[1]):
                n_correct += 1
```

Aus der Anzahl der richtig klassifizierten Datenpunkte wird in einem letzten Schritt die Klassifikationsgenauigkeit berechnet und zurückgegeben.

```
        accuracy = n_correct / len(dataset)
        return accuracy
```

Mit einer weiteren Methode sollen die Kosten über einem Datensatz ermittelt werden können. Es wird über den Datensatz iteriert und zu jedem Datenpunkt eine Vorhersage berechnet, welche zur Berechnung der Kosten dieser Vorhersage verwendet wird. Im Programm sind die Kosten das arithmetische Mittel aller c, deshalb wird `cost` nicht nur durch die Datensatzlänge, sondern auch durch die Anzahl Outputneuronen geteilt.

$$\bar{C}_{im\ Programm} = \frac{1}{|L| * p} \sum_{i=1}^{p} \sum_{n=1}^{|L|} c_{n,i} = \frac{1}{|L| * p} \sum_{i=1}^{p} C_i \tag{38}$$

Die resultierenden Kosten lassen sich einfacher interpretieren, denn sie sind per Definition die durchschnittlichen Kosten auf Ebene eines Outputs. Diese Definition der Kosten wird im Programm überall verwendet, wo Kosten zur Leistungsbestimmung eines Netzes festgehalten werden.

```
    def cost_over_dataset(self, dataset, costfunc):
        cost = 0
        for d in dataset:
            p = self.predict(d[0])
            cost += costfunc.C(p, d[1])
        cost = cost / len(dataset) / self.size[-1]
        return cost
```

4.2.7 Netz trainieren

Die `train`-Methode eines neuronalen Netzes nimmt die Parameter `dataset`, einen Trainingsdatensatz wie in 4.1.1 anhand MNIST gezeigt, die Anzahl Epochen mit `n_epochs`, die Mini-Batchgrösse `mini_batchsize`, eine Kostenfunktion `costfunc` wie in 4.2.4 gezeigt und die Lernrate `learningrate` entgegen. Zu den optionalen Parametern der Methode gehören neben dem (falls vorhandenen) Testdatensatz `test` die Optionen, die Grösse des Gradienten und den MSE über den Trainingsdaten nach jeder Epoche zu erfassen (`record`), die Kosten nach jeder Epoche in die Konsole auszugeben (`print_cost`) und schliesslich Plots der Kosten (`show_cost`) und Genauigkeit über dem Testdatensatz (`show_acc`) nach dem Training anzuzeigen.

```
    def train(self, dataset, n_epochs, mini_batchsize, costfunc, learningrate,
              test=None, record=False, print_cost=True,
              show_cost=False, show_acc=False):
```

Zuerst werden nützliche Informationen in die Konsole ausgegeben und die Startzeit des Trainings erfasst.

```
        print('''
---Training gestartet---
{} Epochen, Datensatzgrösse: {}, Mini-Batch-Grösse: {}, Lernrate: {}
'''.format[25](n_epochs, len(dataset), mini_batchsize, learningrate))

        start_time = time.time()
```

Nun beginnt der eigentliche Trainingsprozess. Zu Beginn jeder Epoche wird die Variable `epochcost` auf 0 gesetzt und der Trainingsdatensatz in Mini-Batches unterteilt.

```
        for epoch in range(n_epochs):

            epochcost = 0
            mini_batches = Network.create_mini_batches(dataset, mini_batchsize)
```

[25] siehe 8.3.7 String Formatting

Nun wird der Gradient jedes Mini-Batches berechnet und angewandt. Zu jedem Mini-Batch werden die Mini-Batch-Kosten zu den Kosten der Epoche addiert. Falls `record` auf `True` gesetzt wurde, wird ebenfalls der durchschnittliche Betrag der Elemente des Gradienten erfasst.

```
for mini_batch in mini_batches:
    self.update_params(mini_batch, costfunc, learningrate)
    epochcost += self.mini_batchcost
    if record:
        self.gradient_sizes.append([np.mean(np.absolute(i))
                                    for i in self.gradient_w])
```

Nach der Definition der Kosten aus 4.2.6 (Formel 38) wird die Summe der Batchkosten durch die Datensatzlänge und durch die Anzahl Outputneuronen geteilt. Diese Kosten werden zur Liste der Epochenkosten hinzugefügt und die Zählervariable der Anzahl trainierter Epochen um eins erhöht. Falls so festgelegt, wird der Trainingsfortschritt und die Kosten der Epoche ausgegeben.

```
epochcost = epochcost / len(dataset) / self.size[-1]
self.epochcosts.append(epochcost)
self.epochs_trained += 1

if print_cost:
    print('Epoche {} von {} fertig. Kosten: {:.6f[26]}'.format(
           epoch+1, n_epochs, epochcost))
```

Falls ein Testdatensatz gegeben ist, wird der MSE und die Klassifikationsgenauigkeit über dem Testdatensatz gemessen und in den entsprechenden Listen gespeichert.

```
if test is not None:
    mse = self.cost_over_dataset(test, F.MSE[27])
    self.mse_test.append(mse)

    accuracy = 100 * self.classification_accuracy(test)
    self.accuracys.append(accuracy)
    if show_test:
        print('Genauigkeit: {:.3f} %'.format(accuracy))
```

Falls `record` auf `True` gesetzt wurde, werden ebenfalls die MSE-Kosten über dem Trainingsdatensatz erfasst.

```
if record:
    mse = self.cost_over_dataset(dataset, F.MSE)
    self.mse_train.append(mse)
```

Nachdem das Training beendet wurde, wird das in die Konsole ausgegeben, zusätzlich mit den finalen Kosten, falls sie nicht davor schon ausgegeben wurden, und der Zeit, die das Training benötigte.

[26] rundet auf 6 Nachkommastellen
[27] F bezieht sich auf die Datei, in der die Aktivierungs- und Kostenfunktionen gespeichert wurden.

```
        if not print_cost:
            print('letzte Kosten: {:.6f}'.format(self.epochcosts[-1]))
        print('Zeit: {:.2f} s'.format(time.time() - start_time))
        print('---Training abgeschlossen---')
        print('')
```

Schlussendlich werden die Kosten und die Genauigkeit, falls so eingestellt, in einem Plot dargestellt. Die `simple_plot(<daten_y>, <titel>, <x_beschriftung>, <y_beschriftung>)`-Funktion erstellt einen Plot (siehe 8.3.10) von den `daten_y` mit jeweils einem Abstand von 1 in die x-Richtung.

```
        if show_cost:
            simple_plot(self.epochcosts, 'Kosten während dem Training',
                        'Epoche', 'Kosten')

        if show_acc:
            simple_plot(self.accuracys, 'Entwicklung der Genauigkeit',
                        'Epoche', 'Genauigkeit [%]')
```

4.3 Ergebnisse Sigmoid mit MSE

Nun kann man das Programm verwenden, um ein MNIST-Modell zu trainieren. Als erstes werden die geschriebenen Programme importiert und die MNIST-Daten geladen:

```
import NeuralNetwork as N[28]
import Functions as F

SAVEPATH = 'C://Users//...'
DATAPATH = 'C://Users//...'
traindata = N.load(DATAPATH + 'mnist_traindata.pkl')
testdata = N.load(DATAPATH + 'mnist_testdata.pkl')
```

Als Beispiel verwende ich ein neuronales Netz mit einer Zwischenschicht aus 50 Neuronen. Die Aktivierungsfunktion ist in allen Schichten die Sigmoidfunktion.

```
conf = [784, 50, 10]
actfuncs = [F.Sigmoid, F.Sigmoid]

mnistnet = N.Network(conf, actfuncs)
```

Nun wird das Netz über 20 Epochen in Mini-Batches mit jeweils 10 Beispielen mit der MSE-Kostenfunktion trainiert. Die Lernrate beträgt 3.

```
costfunc = F.MSE
lr = 3
n_epochs = 20
batchsize = 10

mnistnet.train(traindata, n_epochs, batchsize, costfunc, lr,
               test=testdata, print_cost=True, show_cost=True, show_acc=True)
```

[28] Die Netz-Klasse und Funktionen zum Speichern/Laden von `.pkl`-Dateien befinden sich in `NeuralNetwork.py` und die Aktivierungs- und Kostenfunktionen in `Functions.py`.

Das resultierende Netz erreicht eine Klassifikationsgenauigkeit von mehr als 96.5 %. Die Kosten betragen weniger als 0.0015. Die Diagramme, welche vom Programm erstellt wurden, sind auf der nächsten Seite dargestellt.

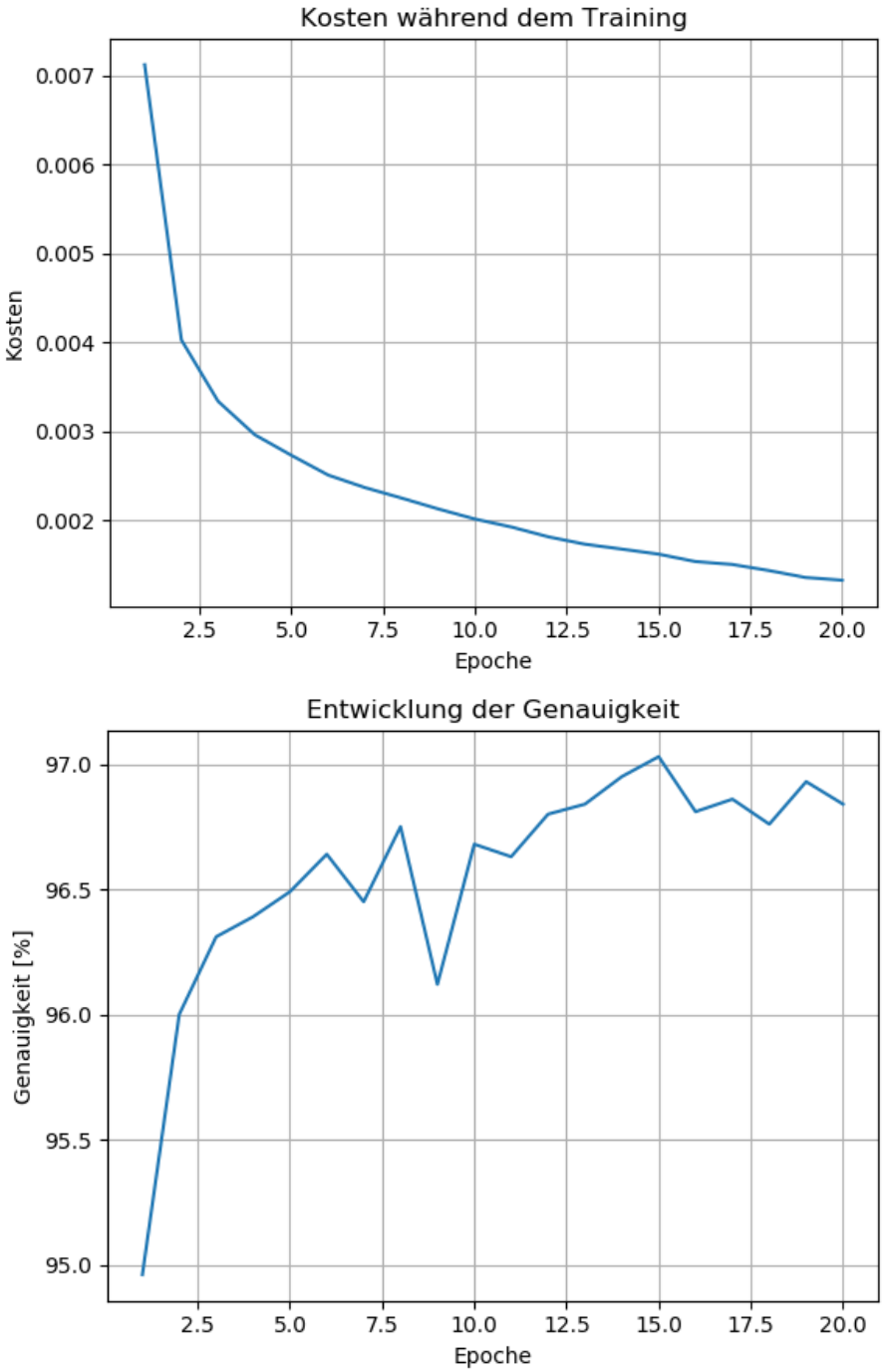


Abbildung 20: Ergebnis eines Sigmoidnetzes nach 20 Epochen Training

5 Experiment

5.1 Zu Untersuchen

Im Experiment werden neuronale Netze mit den verschiedenen Aktivierungs- und Kostenfunktionen aus dem Theorieteil verglichen. Die Netze werden dabei besonders auf den Trainings-MSE und die Klassifikationsgenauigkeit getestet. Folgende Aspekte werden untersucht:

- Ein Netz mit zwei Zwischenschichten kann theoretisch Funktionen besser approximieren (siehe 3.2.5) als eines mit nur einer Zwischenschicht, da die zusätzliche Schicht dem Netz eine weitere Repräsentationsstufe bietet.
- Das Vanishing-Gradient-Problem (siehe 3.2.8) führt in Sigmoidnetzen, welche mit der MSE-Kostenfunktion trainiert werden, zu sehr kleinen Gradienten bei fortgeschrittenem Training, wodurch sich das Netz kaum verbessert. Die Ursache des VGPs sind die $A'(z_n^L)$-Terme in den Fehlern der letzten Schicht, welche bei saturierten Outputneuronen quasi 0 sind, wodurch die partiellen Ableitungen nach allen Parametern ebenfalls quasi 0 sind.
  In der Arbeit wurden zwei mögliche Lösungen präsentiert:
  - Die Kreuzentropie-Kostenfunktion eliminiert in Kombination mit der Sigmoid-Aktivierungsfunktion den $\sigma'(z_n^L)$-Term aus dem Fehler der letzten Schicht (siehe 3.2.9).
  - Die Rectifier-Aktivierungsfunktion besitzt eine teilweise konstante Ableitung, wodurch $A'(z_n^L)$ während des Trainings nicht überproportional klein wird (siehe 3.2.10).
- Mit der Rectifier-Aktivierungsfunktion (siehe 3.2.10) kann ein Netz bei einem zu grossen Aktualisierungsschritt der Parameter steckenbleiben, indem die gewichteten Summen der letzten Schicht nach dieser Aktualisierung bei beinahe allen Beispielen des Trainingsdatensatzes negativ sind, wodurch jeder Fehler der letzten Schicht 0 ist. Wenn die Lernrate nicht zu gross ist, sollte dies aber nicht zu einem Problem werden. lReLUs sollen gleich gute Resultate wie ReLUs liefern, wobei sie zudem verhindern, dass Netze stecken bleiben.
  Sowohl ReLUs als auch lReLUs sind nur bedingt als Aktivierungsfunktion der letzten Schicht geeignet, da sie nicht beschränkt sind und deshalb deutlich grössere Aktivierungen als 1 hervorbringen können.
- Die Softmax-Aktivierungsfunktion (siehe 3.2.11) in der letzten Schicht sorgt für geringere Kosten gegenüber der Sigmoid-Aktivierungsfunktion, indem sie die grösste gewichtete Summe von den kleineren gewichteten Summen «trennt». Die Klassifikationsgenauigkeit eines Netzes mit der Softmax-Aktivierungsfunktion in der letzten Schicht und ansonsten Sigmoid-Schichten sollte aber nicht höher als bei einem reinen Sigmoid-Netz werden.
  Die Softmax-Funktion eignet sich nur als Aktivierungsfunktion in der letzten Schicht und wird auch nur so getestet.

Um diese Aspekte zu untersuchen, werden neuronale Netze mit folgenden Konfigurationen trainiert (beim Training werden alle Daten erfasst, welche in einem `Network`-Objekt gespeichert werden können, vollständige Liste bei 4.2):

Tabelle 1: Konfigurationen, welche getestet werden

Netzgrösse	Aktivierungsfunktionen	Kostenfunktion
784, 50, 10	Sigmoid in allen Schichten	MSE
784, 50, 50, 10	Sigmoid in allen Schichten	MSE
784, 50, 50, 10	Sigmoid in allen Schichten	Kreuzentropie
784, 50, 50, 10	ReLU in allen Schichten	MSE
784, 50, 50, 10	lReLU in allen Schichten	MSE
784, 50, 50, 10	Softmax in der letzten Schicht, sonst Sigmoid	Kreuzentropie
784, 50, 50, 10	Softmax in der letzten Schicht, sonst lReLU	Kreuzentropie
784, 50, 50, 10	Sigmoid in der letzten Schicht, sonst lReLU	Kreuzentropie

Da die Netze zufällig initialisiert werden und basierend auf den Startparametern unterschiedliche Resultate liefern, werden zu jeder Konfiguration drei Netze trainiert. Jedes Netz wird über 50 Epochen mit der Mini-Batch-Grösse 10 trainiert. Diese Mini-Batch-Grösse ist nicht speziell optimiert, aber dieser Wert hat beim Ausprobieren funktioniert und sollte, falls suboptimal, die Konvergenzgeschwindigkeit aller Konfigurationen gleich stark beeinträchtigen. Die Netze werden alle auf den MNIST-Datensatz trainiert.

Alle trainierten Modelle und Resultate sind wie der Code bei *https://tinyurl.com/MADeepLearning* verfügbar.

5.2 Experiment-Software

Um das Experimentieren zu vereinfachen und zu automatisieren, habe ich Funktionen erstellt (mit dem restlichen Code verfügbar), auf die hier kurz eingegangen wird.

- Die `experiment`-Funktion trainiert drei neuronale Netz mit einer angegebenen Konfiguration und speichert sie.
- `auswerten()` extrahiert Daten aus gespeicherten Netzen einer Konfiguration und speichert sie in csv-Dateien, welche u.a. in Excel zur weiteren Bearbeitung geöffnet werden können.
- Die `lr_search`-Funktion hilft beim Finden von geeigneten Lernraten. Sie nimmt als Argument eine Liste mit Lernraten entgegen und trainiert zu jeder ein Netz über 5 Epochen. Die Kosten während des Trainings der letzten Epochen werden zusammen mit den zugehörigen Lernraten in einem Diagramm dargestellt. Ein solches ist für ein Sigmoidnetz mit einer Zwischenschicht von 50 Neuronen und dem MSE als Kostenfunkton angewandt auf MNIST rechts zu sehen. Aus dem Diagramm kann man nun eine geeignete Lernrate ablesen, die darauf optimiert ist, die Kosten über dem Trainingsdatensatz nach 5 Epochen zu minimieren.

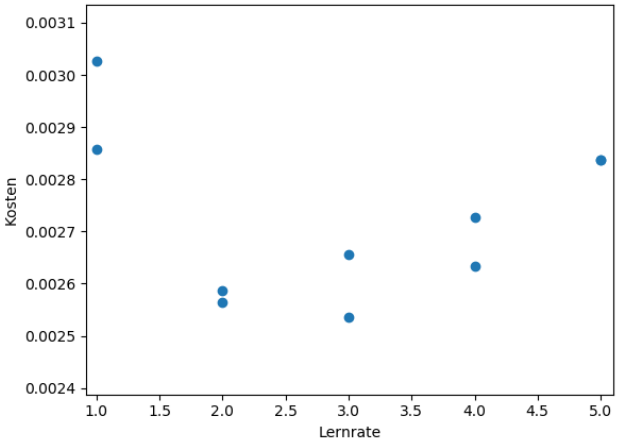


Abbildung 21: Diagramm der Kosten eines Sigmoidnetzes nach 5 Epochen Training bei verschiedenen Lernraten

5.3 1. Versuch

Die Lernraten der ersten Versuchsreihe wurden mithilfe von Diagrammen der `lr_search()`-Funktion ausgesucht und sind somit darauf optimiert, die Kosten nach 5 Epochen zu minimieren. Daraus ergaben sich folgende Lernraten:

Tabelle 2: Lernraten des ersten Versuchs

Netzgrösse	**Aktivierungsfunktionen**	**Kostenfunktion**	**Lernrate**
784, 50, 10	Sigmoid in allen Schichten	MSE	3
784, 50, 50, 10	Sigmoid in allen Schichten	MSE	3
784, 50, 50, 10	Sigmoid in allen Schichten	Kreuzentropie	0.2
784, 50, 50, 10	ReLU in allen Schichten	MSE	0.08
784, 50, 50, 10	lReLU in allen Schichten	MSE	0.08
784, 50, 50, 10	Softmax in der letzten Schicht, sonst Sigmoid	Kreuzentropie	0.2
784, 50, 50, 10	Softmax in der letzten Schicht, sonst lReLU	Kreuzentropie	0.05
784, 50, 50, 10	Sigmoid in der letzten Schicht, sonst lReLU	Kreuzentropie	0.07

Resultate

Die Konfiguration ReLU / MSE ist in zwei der drei Versuche schon nach wenigen Epochen steckengeblieben (d.h. alle Aktivierungen der letzten Schicht waren bei so gut wie jedem Beispiel des Trainingsdatensatzes 0, bzw. alle gewichteten Summen unter 0, wodurch der Gradient aller Parameter quasi 0 war und das Netz nicht weiter angepasst wurde).

Die Trainings-MSE-Kosten aller anderen Konfigurationen sind in folgender Abbildung dargestellt:

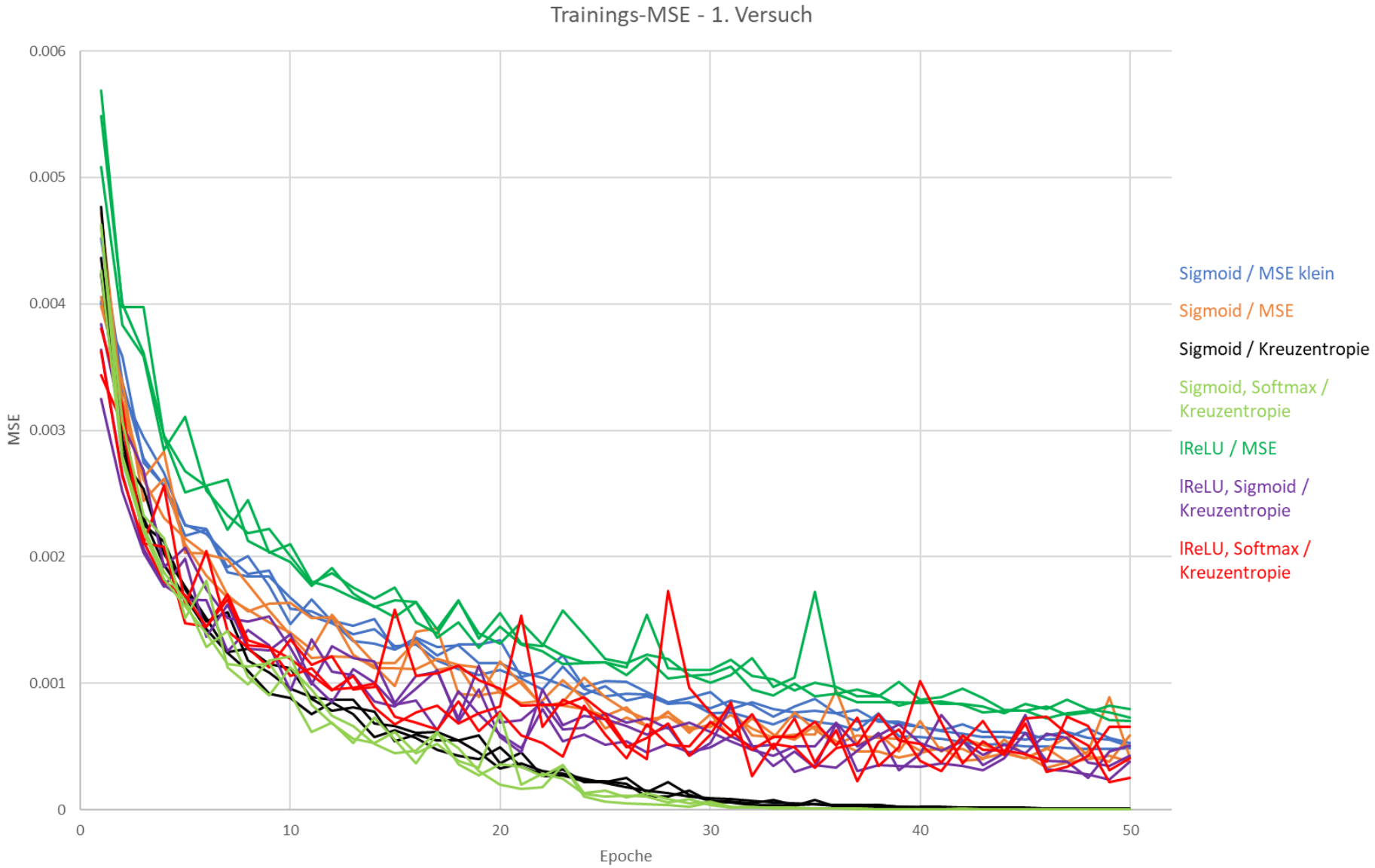


Abbildung 22: Experiment: Trainings-MSE des ersten Versuchs

Die Konfigurationen Sigmoid / Kreuzentropie und Sigmoid, Softmax / Kreuzentropie lernen den Trainingsdatensatz deutlich besser als alle anderen Konfigurationen, welche nicht deutlich unter einen MSE von 0.0002 kommen und zudem starke Sprünge in den Kosten während des Trainings aufweisen (besonders die lReLU-Netze). Dies ist ein Zeichen für eine zu hoch gewählte Lernrate (vgl. Abbildung 5 aus 3.1.3). Es ist zu erwarten, dass eine Lernrate, welche die Kosten nach 5 Epochen minimiert, tendenziell zu hoch für ein Training über 50 Epochen ist, dennoch hätte ich nicht gedacht, dass dieser Effekt so deutlich zu sehen ist. Die zu hohen Lernraten lassen die Netze nicht ihr volles Potenzial ausschöpfen, was den Vergleich der Konfigurationen mit diesen Resultaten verunmöglicht. Aus diesem Grund habe ich mich dazu entschlossen, das Experiment für alle Konfigurationen bis auf Sigmoid / Kreuzentropie und Sigmoid, Softmax / Kreuzentropie mit geringeren Lernraten zu wiederholen.

5.4 2. Versuch

Die Lernraten für den zweiten Versuch wurden basierend auf den Ergebnissen des ersten Versuchs etwas kleiner gewählt. Für die Konfiguration mit den ReLUs musste ich die Lernrate deutlich verringern, um zu sicherzustellen, dass die Netze nicht steckenbleiben.

Tabelle 3: Lernraten des zweiten Versuchs

Netzgrösse	Aktivierungsfunktionen	Kostenfunktion	Lernrate (alt)	Lernrate
784, 50, 10	Sigmoid in allen Schichten	MSE	3	2
784, 50, 50, 10	Sigmoid in allen Schichten	MSE	3	2
784, 50, 50, 10	ReLU in allen Schichten	MSE	0.08	0.01
784, 50, 50, 10	lReLU in allen Schichten	MSE	0.08	0.05
784, 50, 50, 10	Softmax in der letzten Schicht, sonst lReLU	Kreuzentropie	0.05	0.03
784, 50, 50, 10	Sigmoid in der letzten Schicht, sonst lReLU	Kreuzentropie	0.07	0.03

Resultate

Trainings-MSE

Die ReLU-Netze sind mit der deutlich verringerten Lernrate zwar nicht mehr steckengeblieben, aber wurden über die 50 Epochen kaum besser.

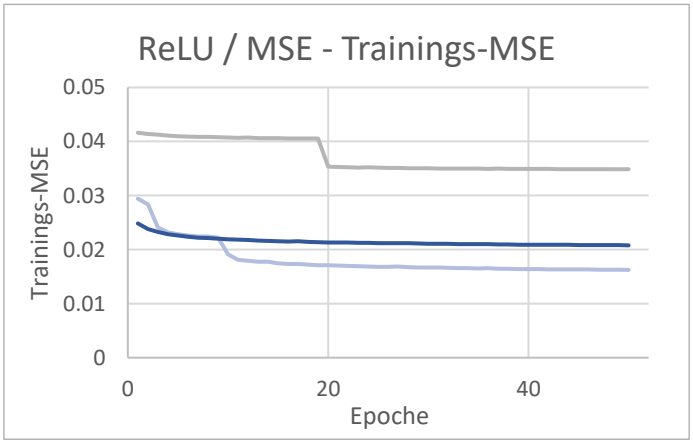


Abbildung 23: Trainings-MSE der Konfiguration ReLU / MSE

Bei den anderen Konfigurationen ergab sich mit den neuen Lernraten ein deutlich weicherer Verlauf ohne grosse Sprünge, dennoch waren die Kosten der letzten Epoche bei Sigmoid / MSE (784, 50, 50, 10) und bei lReLU / MSE beim ersten Versuch geringer. Die Trainings-MSE aller Konfigurationen mit der besseren Lernrate sind in der folgenden Abbildung dargestellt. Nur diese Resultate werden anschliessend berücksichtigt.

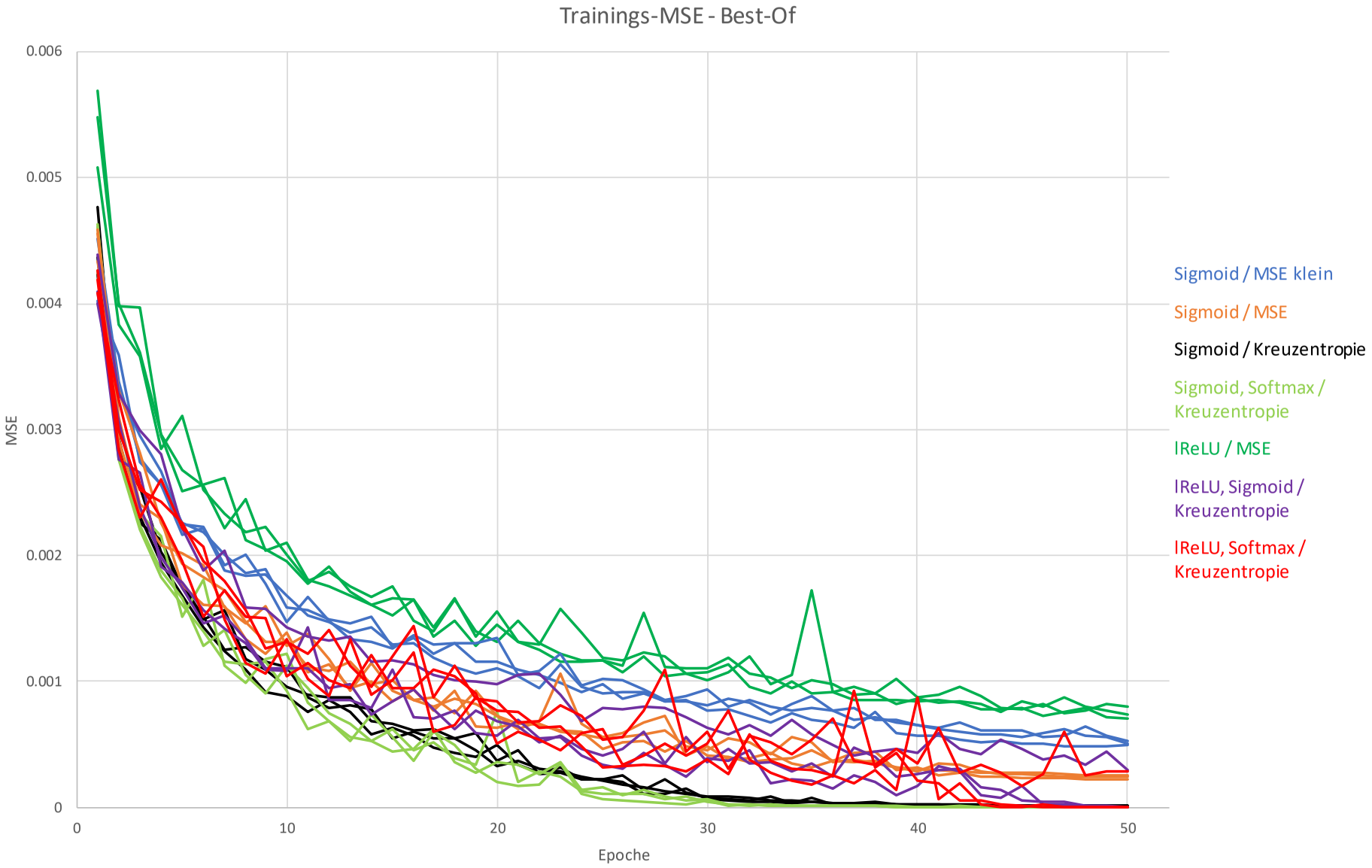


Abbildung 24: Experiment: Trainings-MSE aller Konfiguration en mit der besseren Lernrate

Aus der Abbildung ergeben sich folgende Beobachtungen:

- Die Konfiguration Sigmoid / MSE mit zwei Zwischenschichten ist deutlich besser als mit nur einer.
- Die Konfiguration Sigmoid / Kreuzentropie, sowie Sigmoid, Softmax / Kreuzentropie sind aber wie erwartet nochmals deutlich besser und konvergieren zu sehr geringen Kosten, wobei Softmax / Kreuzentropie geringfügig besser ist.
- Die Konfiguration mit nur lReLUs schneidet am schlechtesten ab, aber nicht weit hinter Sigmoid / MSE mit nur einer Zwischenschicht. Die ReLU-Netze mit der Sigmoid- und Softmax-Funktion in der letzten Schicht (und der Kreuzentropie-Kostenfunktion) gelangen auf das Niveau von Sigmoid / Kreuzentropie oder Sigmoid, Softmax / Kreuzentropie, aber nur bei zwei der drei trainierten Netze.
- Die Kapazität der neuronalen Netze wird nach 50 Epochen Training scheinbar relativ gut ausgenutzt, denn die Kurven der Trainings-MSE verlaufen im Bereich zwischen 40 und 50 Epochen sehr flach, d.h. dass sich die Netze nach 50 Epochen nicht mehr gross verbessern würden.

Klassifikationsgenauigkeit

Die Klassifikationsgenauigkeiten unterschieden sich bei fortgeschrittenem Training zwischen den Konfigurationen nur maximal um 1.5 % und weisen im Verhältnis zu diesen 1.5% von Epoche zu Epoche enorme Schwankungen (je nach Konfiguration von Epoche zu Epoche von über 0.5 %) auf und sind, dargestellt in einem Diagramm mit der Genauigkeit zu jeder Epoche, sehr unübersichtlich. Man würde sowieso das Netz mit der höchsten Klassifikationsgenauigkeit verwenden, deshalb habe ich zu jeder Konfiguration die maximale Klassifikationsgenauigkeit herausgesucht:

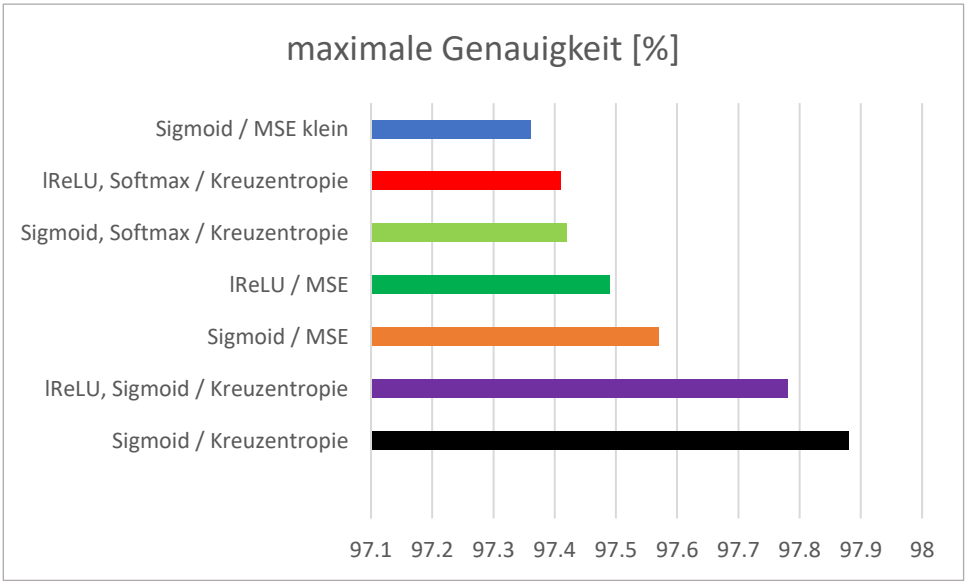


Abbildung 25: Experiment: maximale Klassifikationsgenauigkeit

Die beste Klassifikationsgenauigkeit von 97.88 % wurde von einem Sigmoid-Netz mit der Kreuzentropie-Kostenfunktion (nach nur 18 Epochen) erreicht. Trotz der deutlichen Unterschiede in der Trainings-MSE erreichen die Konfigurationen relativ ähnliche Klassifikationsgenauigkeiten – der Unterschied der schlechtesten zur besten Konfiguration beträgt nur 0.52 %.

Bei diesen Klassifikationsgenauigkeiten handelt es sich um Ausreisser; keine dieser Klassifikationsgenauigkeiten wurde nach 50 Epochen erfasst, sondern alle inmitten des Trainings. Dadurch sind diese Werte nur zu einem geringen Grad repräsentativ für den Erfolg dieser Konfigurationen, besonders im Hinblick auf das Anwenden dieser Konfigurationen auf andere Klassifikationsaufgaben. Dass es sich bei den Werten um Ausreisser handelt, sieht man an auch der Rangordnung, die hier völlig anders als bei der Trainings-MSE ist. Der Trainings-MSE korreliert negativ mit der Klassifikationsgenauigkeit, aber scheinbar ist die Korrelation nicht genug hoch, so dass die Unterschiede zwischen den Konfigurationen im Trainings-MSE in gleicher Weise bei der Klassifikationsgenauigkeit auftreten.

Grösse des Gradienten

Die Grösse des Gewichtsgradienten (der Durchschnittswert der Einträge aller Gewichtsgradientenmatrizen einer Epoche) in der ersten und letzten Schicht in Abhängigkeit der Trainings-MSE ist für alle Konfigurationen auf der nächsten Seite dargestellt.

Aus der Abbildung ergeben sich folgende Beobachtungen:

- Die Gradienten in der ersten Schicht sind generell deutlich kleiner als in der letzten Schicht.
- Zudem sieht man sofort, dass die Konfigurationen mit der Sigmoid-Aktivierungsfunktion und der MSE-Kostenfunktion zu deutlich kleineren Gradienten als bei allen anderen Konfigurationen führen. Zwischen den beiden gibt es sowohl in der ersten als auch in der letzten Schicht kaum Unterschiede.
- Die Konfigurationen Sigmoid / Kreuzentropie und Sigmoid, Softmax / Kreuzentropie haben jeweils in der ersten und letzten Schicht ähnlich grosse Gradienten, welche grösser sind als bei den Konfigurationen mit der Sigmoid-Aktivierungsfunktion und der MSE-Kostenfunktion.
- Die Konfiguration lReLU / MSE hat in der letzten Schicht etwa gleich grosse Gradienten wie die Konfigurationen Sigmoid / Kreuzentropie und Sigmoid, Softmax / Kreuzentropie. In der ersten Schicht sind die Gradienten der Konfiguration lReLU / MSE im Vergleich aber kleiner.
- Die lReLU-Netze mit der Sigmoid- oder Softmax-Aktivierungsfunktion in der letzten Schicht erzeugen mit Abstand die grössten Gradienten, sowohl in der ersten als auch in der letzten Schicht.

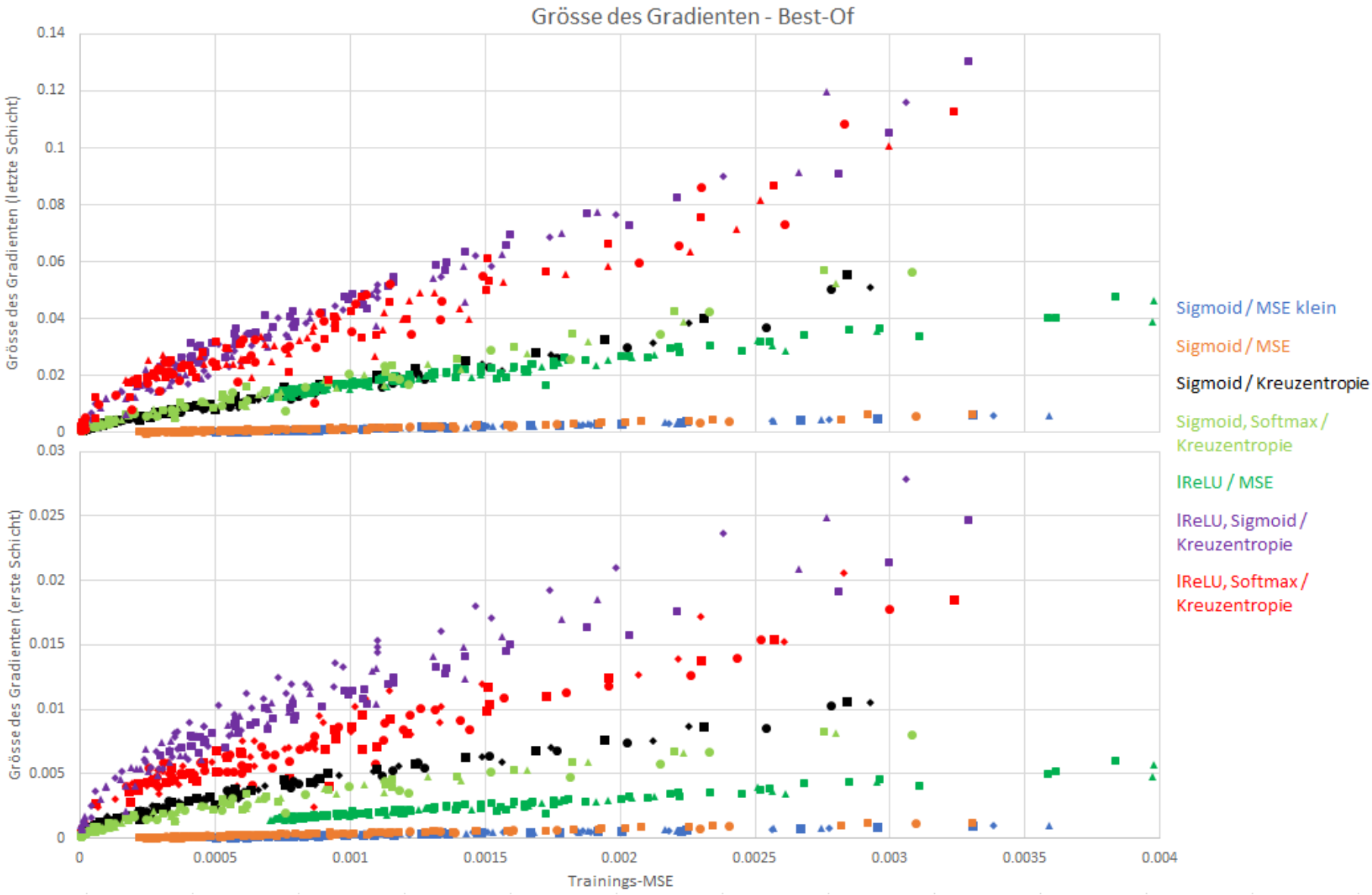


Abbildung 26: Experiment: Grösse des Gewichtsgradienten

5.5 Diskussion

Die Resultate entsprechen bis auf wenige Ausnahmen relativ genau dem, was erwartet wurde. Diese wenigen Ausnahmen sind folgende:

- Ich hätte aber nicht erwartet, dass das Steckenbleiben von ReLU-Netzen tatsächlich ein so grosses Problem ist. Nach diesem Resultat war umso weniger zu erwarten, dass dieses Problem mit den eigentlich sehr ähnlichen lReLUs nicht auftritt und diese mit der Sigmoid- oder Softmax-Aktivierungsfunktion in der letzten Schicht zu den besseren Konfigurationen dieses Experiments gehören.
- Etwas speziell sind die Konfigurationen mit lReLUs und der Sigmoid- und Softmax-Funktion in der letzten Schicht, bei jeweils zwei der drei trainierten Netze einen deutlich besseren Trainings-MSE als das dritte Netz erreichten. Dies könnte das Resultat einer noch immer zu hohen Lernrate, denn beide Konfigurationen weisen im Abbildung 24 relativ grosse Sprünge auf. Deren Trainings-MSE könnte mit einer nochmals geringeren Lernrate möglicherweise verbessert werden.
  Zudem haben diese Konfigurationen deutlich grössere Gradienten als andere Konfigurationen mit gleichen Aktivierungsfunktionen in der letzten Schicht und mit gleichen Kostenfunktionen (Sigmoid / Kreuzentropie; Sigmoid, Softmax / Kreuzentropie), obwohl die Gleichungen für e_n^L dann exakt gleich sind. Die einzige Erklärung ist, dass die e_n^L zur Berechnung des Gewichtsgradienten mit a_m^{l-1} multipliziert werden (Formel 27), die wegen der lReLUs aber (im Gegensatz zu Sigmoidneuronen) grösser als 1 sein können. Demnach könnte der ungebundene Teil der Rectifier-Aktivierungsfunktion (zusätzlich zur konstanten Ableitung) die Konvergenzzeit von SGD verringern.
- Bei der Konfiguration lReLU / MSE hätte ich einen deutlich kleineren Gradienten in der letzten Schicht als bei den Kombinationen Sigmoid / Kreuzentropie und Sigmoid, Softmax / Kreuzentropie erwartet, da $A'(z_n^L)$ bei neun Zehntel der Outputneuronen 0 sein sollte ($A(z_n^L)$ sollte bei neun der zehn Outputneuronen 0 sein) und bei den anderen Konfigurationen bei allen Outputneuronen $e_n^L = a_n^L - \hat{y}_n$ gilt. Dies könnte man erneut damit begründen, dass a_m^{l-1} anders als bei der Sigmoid- oder Softmax-Funktion grösser als 1 sein kann und deshalb den kleinen Fehler kompensiert (Formel 27).
  Der Unterschied zur Grösse des Gradienten in der ersten Schicht wäre damit ebenfalls begründbar, denn dort sind die Aktivierungen die Features, welche sicher zwischen 0 und 1 liegen.
- Zudem hätte ich erwartet, dass die Unterschiede in der Klassifikationsgenauigkeit grösser sein würden, nachdem ich die Unterschiede in der Trainings-MSE gesehen hatte.

Im Theorieteil der Arbeit wurden zudem weitere Punkte erwähnt, welche genauer in einem Experiment hätten untersucht werden können. Dazu gehört:

- Netze mit deutlich mehr Zwischenschichten
- das Verringern der Lernrate während des Trainings
- die Optimierung der Mini-Batch-Grösse auf die Konvergenzzeit
- verschiedene α-Werte bei lReLUs

6 Zusammenfassung und Ausblick

Neuronale Netze lassen sich gut auf das Erkennen von handgeschriebenen Ziffern anwenden. Im Experiment wurde eine maximale Klassifikationsgenauigkeit von 97.88 % ausserhalb des Trainingsdatensatzes erreicht. Je nach Aktivierungs- und Kostenfunktion ergeben sich Unterschiede, welche aber schlussendlich weniger als 1 % Klassifikationsgenauigkeit ausmachen.

6.1 Was es in Deep Leaning noch gibt

In dieser Arbeit wurde nur ein kleiner Teil dessen behandelt, was Deep Learning ausmacht. Des Weiteren gibt es:

- andere Aktivierungs- (z.B. der hyperbolische Tangens) und Kostenfunktionen (z.B. die euklidische Distanz zwischen Vorhersagen- und Labelvektor), wobei die wichtigsten und beliebtesten thematisiert wurden.
- andere künstliche Neuronen und künstliche neuronale Netze, welche z.B. von der biologischen visuellen Wahrnehmung inspiriert wurden und sich besonders zum Verarbeiten von Bild- oder Audiodaten eignen. (Wikipedia - Convolutional Neural Network, 2018) Andere Netze erlauben Verbindungen von Neuronen zu Neuronen derselben oder vorigen Schichten und eignen sich damit zum Verarbeiten von sequenziellen Daten wie z.B. Sprache. (Wikipedia - Rekurrentes neuronales Netz, 2018)
- Trainingsalgorithmen, welche frühere Aktualisierungsschritte bei der Berechnung neuer Schritte miteinbeziehen, um schnellere Konvergenz zu ermöglichen. (Goodfellow, Bengio, & Courville, Momentum, 2016)
- Regularisierungsmethoden, welche dafür sorgen, dass Netze besser generalisieren. L1- und L2-Regularisierung zwingen das Netz, die Gewichte während dem Training nicht zu gross werden zu lassen, weil grosse Gewichte auf eine Erlernung sehr spezifischer Merkmale des Trainingsdatensatzes hindeuten können, welche ausserhalb des Trainingsdatensatzes keine Bedeutung haben. (Nielsen, Regularization, 2017) Bei Dropout werden in jedem Trainingsschritt eine fixe Anzahl der Neuronen deaktiviert, wodurch das Netz gezwungen wird, die Klassen über mehrere Neuronen erkennen zu können (Nielsen, Other techniques for regularization, 2017)

Zudem gibt es natürlich viele andere Anwendungsmöglichkeiten als das Einteilen von handgeschriebenen Ziffern, von der Diagnose von Krankheiten aus Bildern von vermutlich kranken Zellen bis hin zur Wettervorhersage.

6.2 Was nicht im Programm enthalten ist

Das Programm ist nicht konzipiert, um die bestmögliche Performance zu liefern. Bei grossen Deep Learning-Projekten wird aber alle Performance benötigt, da das Trainieren von Netzen auch mit guter Hardware Tage bis Monate dauern kann. Bessere Performance wird bei professionellen Deep Learning-Implementierungen auf folgende Arten erreicht:

- In Python sind alle Variablen Objekte, wodurch sie zwar keinen festen Datentyp benötigen, die Programme aber langsamer als in anderen Programmiersprachen laufen. Bessere Performance kann durch eine Implementierung in einer anderen

Sprache erreicht werden, oder indem rechenaufwendige Python-Funktionen wie z.B. die zur Berechnung des Gradienten in einer schnellen Programmiersprache implementiert werden. So sind z.B. viele NumPy-Funktionen im schnellen C implementiert.

- Die Berechnungen im Programm dieser Arbeit werden auf der CPU ausgeführt. Matrix-Operationen, welche den Grossteil aller Berechnungen bei neuronalen Netzen ausmachen, können aber deutlich schneller auf Grafikprozessoren durchgeführt werden; und deshalb unterstützen die beliebtesten Deep Learning-Programmbibliotheken Grafikprozessoren oder andere Hardware, die auf die Matrizenoperationen spezialisiert sind.
- Bei neuronalen Netzen lassen sich alle Vorhersagen zu einem Mini-Batch in nur einem Feedforward berechnen, indem die Aktivierungsvektoren mit Matrizen ersetzt werden, deren Zeilen die Aktivierungsvektoren der einzelnen Beispiele des Mini-Batches sind (es lassen sich im Wesentlichen dieselben Gleichungen verwenden). Dies kann wegen gut skalierender Matrixmultiplikationen schneller sein, als wenn die Beispiele des Mini-Batches einzeln durchgegangen werden.

Deep Learning Bibliotheken wie «Tensorflow» lassen den Nutzer einen Computationgraph aufbauen. Dadurch können verschiedenste Modelle neben neuronalen Netzen umgesetzt werden, deren Gradient direkt durch den Computationgraph ausgerechnet wird.

Mein Programm ist in einer weiteren Hinsicht sehr ineffizient, denn es werden beim Berechnen des Fehlers der letzten Schicht die Ableitung der Aktivierungsfunktion und die partielle Ableitung der Kostenfunktion separat ausgerechnet, obwohl sich bei einer Kombination der Sigmoid- oder Softmax-Aktivierungsfunktion mit der Kreuzentropie-Kostenfunktion die Ableitung der Aktivierungsfunktion wegkürzt. Dadurch wird die Ableitung der Aktivierungsfunktion berechnet, aber dies wird effektiv nie benötigt. Die effizientere Alternative zu meiner Umsetzung wäre es gewesen, der `train`-Methode eine Funktion zur direkten Berechnung des Fehlers der letzten Schicht zu übergeben, wodurch die Aktivierungs- und Kostenfunktion aber nicht mehr unabhängig voneinander angegeben wären und somit ein Teil der Modularität des Programms verloren ginge.

Zudem ist es bei langen Trainingszeiten nützlich, während des Trainings über den Stand der Kosten informiert zu werden, um zu entscheiden, ob es sich überhaupt lohnt, ein Netz weiter zu trainieren. Die Kosten können zwar nach jeder Epoche in die Konsole ausgegeben werden, aber erst nach fertiggestelltem Training in einer Grafik dargestellt werden. Mit Matplotlib lassen sich Plots erstellen, die in Echtzeit aktualisiert werden, jedoch ist es mir nicht gelungen, so etwas zu implementieren (Der Live-Plot interferiert währenddem er auf neue Daten wartet mit den restlichen Berechnungen des Programms, welche vollständig zum Erliegen kommen.).

7 Schlusswort

Beim Erarbeiten dieser Maturaarbeit habe ich viel Spannendes gelernt, was sowohl Mathematik als auch Informatik betrifft und im Studium nützlich sein könnte. Insbesondere finde ich es erstaunlich, dass die teilweise heuristische Herangehensweise von Machine- und Deep Learning so gut funktioniert. Am meisten fasziniert hat mich aber, dass künstliche neuronale Netze basierend auf deren biologischen Plausibilität als Modelle eingesetzt wurden, und wie sich später zeigte völlig zurecht, denn sie können jede Funktion approximieren und sind deshalb in gewisser Weise die besten Modelle.

Meine Herangehensweise war teilweise etwas chaotisch – so habe ich mich sofort nach Beginn des Projekts ans Programmieren gesetzt und mir nach und nach neues Wissen angeeignet, wenn ich auf Probleme gestossen bin. So manches von meinem Code hat es nicht in die Arbeit geschafft, da sie auch so schon recht umfangreich geworden ist. Leider konnte ich mich aus demselben Grund nur mit den Grundlagen des Deep Learning auseinandersetzen. Deep Learning ist ein spannendes und sehr umfangreiches Thema und wird mich vermutlich zukünftig noch weiter beschäftigen.

8 Anhang

8.1 Begriffsverzeichnis

Aktivierung a (engl.: *activation*): der Ausgabewert eines künstlichen Neurons

Aktivierungsfunktion $A(z)$ (engl.: *activation function*): berechnet die Aktivierung eines Neurons aus der gewichteten Summe

Attribut: Sorte von Daten, anhand derer eine Vorhersage getroffen

Backpropagation (zu Deutsch *Fehlerrückführung*): Verfahren zum Berechnen des Gradienten bei künstlichen neuronalen Netzen

Backward Pass (engl. für *verkehrter Durchgang*): Rückwärtsdurchlauf eines künstlichen neuronalen Netzes zum Berechnen der Fehler

Decision Boundary (engl. für *Entscheidungsgrenze*): Grenze zwischen den beiden «Entscheidungsmöglichkeiten» eines Outputneurons im Inputraum eines Modells

Epoche (engl.: *epoch*): ein Durchlauf des Trainingsdatensatzes

Features x (engl. für Eigenschaft oder Merkmal): Werte zu den Attributen

Feedforward: das «Weiterfüttern» von Daten innerhalb eines künstlichen neuronalen Netzes

Fehler (engl.: *error*) e: partielle Ableitung der Kosten nach einer gewichteten Summe einer Schicht

generalisieren: Die Vorhersagen des Modells sind auch bei Beispielen ausserhalb des Trainingsdatensatzes gut.

Gewicht w (engl.: *weighted sum*): gewichtete die Inputs eines künstlichen Neurons

gewichtete Summe z: Summe der gewichteten Inputs eines künstlichen Neurons und dessen Neigung

Gradient Descent (engl. für *Gradientenabstieg*): Verfahren zum Minimieren von Funktionen des Typs $\mathbb{R}^n \to \mathbb{R}$

Hyperparameter: Parameter, die nicht «erlernt» werden

Inputschicht: gibt die Features an die erste wirkliche Schicht weiter

Klasse: Kategorie, in die eingeteilt werden soll

Klassifikation: die Aufgabe, Daten in Kategorien einteilen

Kosten(funktion) C (engl.: *cost function*): (berechnet) Mass für die Abweichungen von Vorhersage und Label

künstliches Neuron: Modell einer biologischen Nervenzelle

künstliches neuronales Netz (engl.: *artificial neural network*): Zusammenschluss vieler künstlichen Neuronen

Label $\hat{y}$ (engl. für *Etikette* oder *Aufschrift*): zu einem bestimmten Input erwarteter Output eines Modells

Lernrate η (engl.: *learning rate*): bestimmt die Schrittgrösse in Gradient Descent und hat Einfluss auf dessen Konvergenzgeschwindigkeit

Mini-Batch: kleiner Teil des Trainingsdatensatzes

Modell: mathematische Funktion, welche Vorhersagen berechnet

Modellparameter θ_n: bestimmt das «Verhalten» des Modells

Neigung b (engl.: *bias*): Tendenz eines künstlichen Neurons, eine hohe Aktivierung zu haben

One-Hot-Vektor: Klassifikations-Label, eine Komponente ist 1, alle anderen sind 0.

Outputschicht: letzte Schicht eines künstlichen neuronalen Netzes

Regression: die Aufgabe, Werte vorherzusagen

Schicht: Die Neuronen einer Schicht haben alle Aktivierungen der vorherigen Schicht als Input und ihr Output wird an

Stochastic Gradient-Descent (zu Deutsch *zufälliger Gradientenabstieg*, kurz SGD): Gradient Descent, wobei der Gradient mit einem Mini-Batch approximiert wird

Testdatensatz: enthält andere Beispiele als der Trainingsdatensatz

trainieren: Modellparameter optimieren, um bessere Vorhersagen beim Trainingsdatensatz zu erreichen

Trainingsdatensatz: enthält Beispiele zum Trainieren eines Modells

Vanishing-Gradient-Problem (engl. für *verschwindender-Gradient-Problem*, kurz VGP): Problem, bei dem die Gradienten während dem Training verschwindend klein werden und ein Modell sich deshalb kaum mehr verbessert

Vorhersage y: wird vom Modell zu einem Input erstellt

Zwischenschicht: nicht Input- oder Outputschicht

8.2 Formelverzeichnis

(1)	Definition der Kostenfunktion	$C(y,\hat{y}) = \sum_{n=1}^{k} c\,(y_n, \hat{y}_n)$
(2)	Kosten über einem Datensatz	$\bar{C} = \frac{1}{p}\sum_{i=1}^{p} C(y_i, \hat{y}_i)$
(3)	MSE-Kostenfunktion	$\bar{C}_{MSE} = \frac{1}{2p}\sum_{i=1}^{p}\sum_{n=1}^{k}(y_{n,i} - \hat{y}_{n,i})^2\,, \qquad \text{mit } c(y_n, \hat{y}_n) = \frac{1}{2}(y_n - \hat{y}_n)^2$
(4)	MSE mit Skalarprodukt	$C = \frac{1}{2}(y - \hat{y}) * (y - \hat{y})$
(5)	Gradient Descent-Gleichung	$p_{t+1} = p_t - \eta * \nabla f(p_t)$
(6)	Definition Gradient	$\nabla f(p) = \left(\frac{\partial f(p)}{\partial x_1} \quad \frac{\partial f(p)}{\partial x_2} \quad \dots \quad \frac{\partial f(p)}{\partial x_n}\right)$
(7)	Definition partielle Ableitung	$\frac{\partial f(p)}{\partial x_k} = \lim_{h\to 0}\frac{f(p_1, \dots, p_k + h, \dots, p_n) - f(p)}{h}$
(8)	Gradient Descent (Komponentenweise)	$p_{k,t+1} = p_{k,t} - \eta * \frac{\partial f(p_t)}{\partial x_k}$
(9)	Approximation des Gradienten durch Mini-Batch	$\frac{\partial \bar{C}}{\partial \theta_n} \approx \frac{1}{q}\sum_{i=1}^{q}\frac{\partial C_i}{\partial \theta_n}$
(10)	SGD-Gleichung	$\theta_{n,t+1} = \theta_{n,t} - \frac{\eta}{q}\sum_{i=1}^{q}\frac{\partial C_i}{\partial \theta_{n,t}}$
(11)	gewichtete Summe	$z(x_1, \dots, x_k) = \sum_{n=1}^{k} w_n x_n + b$
(12)	Heavisidestufenfunktion	$a(z) = \begin{cases} 0, & z < 0 \\ 1, & z \geq 0 \end{cases}$
(13)	gewichtete Summe (Vektor)	$z^{l+1}(a^l) = a^l * W^l + b^l$
(14)	Aktivierung (Vektor)	$a^{l+1} = A(z^{l+1})$
(15)	Sigmoidfunktion	$\sigma(z) = \frac{1}{1 + e^{-z}}$
(16)	Ableitung der Sigmoidfunktion	$\sigma'(z) = \big(1 - \sigma(z)\big) * \sigma(z)$
(17)	Initialisierung der Gewichte	$w^l_{t=0} \sim N\left(\mu = 0, \sigma^2 = \frac{1}{\lvert l\rvert}\right)$
(18)	Initialisierung der Neigungen	$b_{t=0} \sim N(\mu = 0, \sigma^2 = 1)$
(19)	Definition Fehler	$e^l_n = \frac{\partial C}{\partial z^l_n}$

(20)	Definition Fehler (Vektor)	$e^l = \left(\frac{\partial C}{\partial z_1^l} \quad \frac{\partial C}{\partial z_2^l} \quad \cdots \quad \frac{\partial C}{\partial z_{\|l\|}^l}\right)$
(21)	Fehler der letzten Schicht	$e_n^L = \frac{\partial C}{\partial a_n^L} * A'(z_n^L)$
(22)	Fehler der letzten Schicht (Vektor)	$e^L = \left(\frac{\partial C}{\partial a_1^L} \quad \frac{\partial C}{\partial a_2^L} \quad \cdots \quad \frac{\partial C}{\partial a_{\|L\|}^L}\right) \odot A'(z^L)$
(23)	Fehler in vorheriger Schicht	$e_n^{l-1} = A'\left(z_n^{l-1}\right) * \sum_{m=1}^{\|l\|} w_{n,m}^{l-1} * e_m^l$
(24)	Fehler in vorheriger Schicht (Vektor)	$e^{l-1} = A'\left(z^{l-1}\right) \odot (e^l * {W^{l-1}}^T)$
(25)	partielle Ableitung nach Neigung	$\frac{\partial C}{\partial b_n^l} = e_n^{l+1}$
(26)	partielle Ableitung nach Neigung (Vektor)	$\nabla \mathrm{C}_{b^l} = e^{l+1}$
(27)	partielle Ableitung nach Gewicht	$\frac{\partial C}{\partial w_{n,m}^l} = e_m^{l+1} * a_n^l$
(28)	partielle Ableitung nach Neigung (Vektor)	$\nabla \mathrm{C}_{W^l} = {a^l}^T * e^{l+1}$
(29)	partielle Ableitung der MSE-Kostenfunktion	$\frac{\partial C}{\partial a_n^L} = a_n^L - \hat{y}_n$
(30)	Kreuzentropie-Kostenfunktion	$C = -\sum_{n=1}^{k} \hat{y}_n \ln(a_n) + (1 - \hat{y}_n) \ln(1 - a_n)$
(31)	partielle Ableitung der Kreuzentropie-Kostenfunktion	$\frac{\partial C}{\partial a_n^L} = \frac{1 - \hat{y}_n}{(1 - a_n^L)} - \frac{\hat{y}_n}{a_n^L}$
(32)	Rectifier-Aktivierungsfunktion	$A(z) = \begin{cases} z, & z > 0 \\ 0, & z \leq 0 \end{cases} = \max(z, 0) = z^+$
(33)	Ableitung der Rectifier-Aktivierungsfunktion	$A'(z) = \begin{cases} 1, & z > 0 \\ 0, & z \leq 0 \end{cases}$
(34)	Leaky ReLU	$A(z) = \begin{cases} z, & z > 0 \\ \alpha z, & z \leq 0 \end{cases}, \alpha > 0$
(35)	Ableitung Leaky ReLU	$A'(z) = \begin{cases} 1, & z > 0 \\ \alpha, & z \leq 0 \end{cases}$
(36)	Softmax-Aktivierungsfunktion	$A\left(z_n^l\right) = \frac{e^{z_n^l}}{\sum_{m=1}^{\|l\|} e^{z_m^l}}$
(37)	Ableitung Softmax	$A'(z_n) = A(z_n) * (1 - A(z_n))$
(38)	Kosten im Programm	$\bar{C}_{im\ Programm} = \frac{1}{\|L\| * p} \sum_{i=1}^{p} \sum_{n=1}^{\|L\|} c_{n,i} = \frac{1}{\|L\| * p} \sum_{i=1}^{p} C_i$

8.3 Python-Grundlagen

8.3.1 Installation

Der Download von Python ist bei *https://www.python.org/downloads/* verfügbar. Empfohlen wird aber die Installation über die Anaconda Distribution (*https://www.anaconda.com/download/*), die neben Python NumPy, Matplotlib und andere weitverbreitete Packages mitinstalliert. Installationsanweisungen für NumPy und Matplotlib sind bei *https://www.scipy.org/install.html* verfügbar.

8.3.2 Grundlegendes zur Syntax

Variablen in Python sind nicht an einen Datentyp gebunden. Im folgenden Beispiel wird der Variablen `x` die Ganzzahl 5 zugewiesen, danach der String `'abc'`.

```
>>> x = 5
>>> x = 'abc'
```

Untergeordneter Code muss in Python eingerückt werden, da es keine Zeichen gibt, welche das «Level» des Codes angeben. Folgendes Codebeispiel zeigt das anhand einer If/Else-Bedingung.

```
y = 3
if y > 0:
    print('y ist positiv')
elif y < 0:
    print('y ist negativ')
else:
    print('y ist 0')
```

8.3.3 Listen

Listen sind ein zentrales Element in Python. Sie sind wie Arrays in anderen Programmiersprachen mit zusätzlicher Funktionalität. In einer Liste können verschiedene Typen gespeichert werden und Listen haben eine dynamische Länge. Die Syntax zum Erstellen einer Liste ist wie folgt:

```
>>> liste = [1, 2, 3, 'vier']
```

Elemente einer Liste werden mit ihrem entsprechenden Index nach der Liste in eckigen Klammern aufgerufen (das erste Element hat Index 0).

```
>>> liste[1]
2
```

In Python können negative Indizes verwendet werden. Der Index -1 steht für das letzte Element, -2 für das vorletzte usw.

```
>>> liste[-1]
'vier'
```

Neue Elemente können mit `<liste>.append(<neues_element>)` hinzugefügt werden:

```
>>> liste.append([3.14, 2.71])
[1, 2, 3, 'vier', [3.14, 2.71]]
```

Mit der `len(<liste>)`-Funktion kann die Anzahl Elemente einer Liste abgerufen werden.

```
>>> len(liste)
5
```

8.3.4 For-Schlaufen

In Python wird bei einer for-Schlaufe nicht eine Laufvariabel inkrementiert, stattdessen iteriert man über iterierbare Objekte. Listen gehören u.a. dazu. Bei jedem Durchlaufen einer for-Schlaufe wird der Laufvariablen das nächste «Element» des iterierbaren Objekts zugewiesen.

```
>>> for i in ['hello', 'world']:
...     print(i)

hello
world
```

Um eine klassische for-Schlaufe zu realisieren, verwendet man die `range(<n>)`-Funktion. Sie erstellt einen Generator[29], welcher die natürlichen Zahlen (inklusive 0) bis (und nicht inklusive) `n` ausgibt. Über diese Generatorobjekte kann man ebenfalls iterieren:

```
>>> for i in range(3):
...     print(i)

0
1
2
```

Die `range`-Funktion kann um zwei Parameter erweitert werden. `range(<start>, <stopp>, <schritt>)` ist ein Generator, welcher Zahlen von `<start>` bis (nicht inklusive) `<stopp>` mit der Schrittweite `<schritt>` zurückgibt.

```
>>> for i in range(1, 5, 2):
...     print(i)

1
3
```

8.3.5 List-Comprehensions

Mit List-Comprehensions können Listen in einer Zeile erzeugt werden, die man ansonsten mit einer for-Schlaufe gefüllt hätte.

In beiden Beispielen wird die selbe Liste erstellt.

```
>>> liste = [][30]
>>> for i in range(5):
...     liste.append(i*i)

>>> liste = [i*i for i in range(5)]
```

[29] Beim Iterieren in einer for-Schlaufe verhält sich ein Generatorobjekt gleich wie eine Liste. Es erzeugt und speichert jeweils nur das nächste «Element». Das hat den Vorteil, dass nicht die ganze Liste im Arbeitsspeicher gehalten werden muss.
[30] leere Liste

```
[0, 1, 4, 9, 16]
```

8.3.6 NumPy-Arrays

Da es sich bei NumPy um ein zusätzliches Package handelt, muss dieses in das Programm importiert werden, um auf dessen Inhalt zugreifen zu können. Nach Konvention kürzt man `numpy` mit `np` ab.

```
import numpy as np
```

NumPy-Arrays repräsentieren Vektoren und Matrizen:

```
>>> v = np.array([0, 1, 2])
```
Vektor $v = (0 \quad 1 \quad 2)$

```
>>> M = np.array([[0, 1], [3, -2], [-1, 0]])
```
Matrix $M = \begin{pmatrix} 0 & 1 \\ 3 & -2 \\ -1 & 0 \end{pmatrix}$

Der Zugriff auf einzelne Elemente erfolgt gleich wie bei Listen:

```
>>> M[1][0]
3
```

Die Operatoren `+`, `-`, `*` und `/` führen angewendet auf NumPy-Arrays immer zu einer elementweisen Verknüpfung.

```
>>> np.array([0, 1, 2]) * np.array([3, 5, 8])
np.array([0,  5, 16])
```

Zum Durchführen einer Matrixmultiplikation verwendet man `np.matmul(<matrix1>, <matrix2>)`.

```
>>> np.matmul(v, M)
np.array([ 1, -2])
```

NumPy-Arrays haben das Attribut `shape`. Bei Matrizen ist die Anzahl Zeilen und Spalten darin festgehalten, für Vektoren die Anzahl Elemente.

```
>>> M.shape
(3, 2)
```

```
>>> v.shape
(3,)
```

Mit dem `shape` Attribut können neue Arrays mit einer bestimmten Form erstellt werden. `np.zeros(<shape>)` nimmt ein Shape entgegen und kreiert ein Array der in `shape` spezifizierten Form, welches mit Nullen gefüllt ist.

```
>>> np.zeros(M.shape)
array([[0., 0.],
       [0., 0.],
       [0., 0.]])
```

8.3.7 String Formatting

Um Werte aus Variablen in Strings einzusetzen, verwendet man in Python String Formatting. Dazu setzt man ein Paar von geschweiften Klammern überall dort in einem String, wo etwas einfügen werden soll. Die zusätzlichen Informationen werden in den String eingefügt[31], indem man die `<string>.format(<var1>, <var2>, ...)` Funktion mit den Variablen, die eingesetzt werden sollen, auf den String anwendet.

```
>>> typ = 'formatierter'
>>> 'Dies ist ein {} String'.format(typ)

'Dies ist ein formatierter String'
```

8.3.8 Funktionen

Die Syntax für Funktionsdefinitionen ist in Python ähnlich wie in anderen Programmiersprachen:

```
>>> def sign(y):
...     if y > 0:
...         return 1
...     elif y < 0:
...         return -1
...     else:
...         return 0

>>> sign(3)
1

>>> sign(-2)
-1
```

In Python ist es möglich, Funktionsparameter mit einem Standardwert zu versehen. Dadurch muss der Parameter nur übergeben werden, wenn vom Standardwert abgewichen werden soll. Im folgenden Beispiel ist der Standartwert von `text` der String `'world'`.

```
>>> def hello(text='world'):
...     print('hello {}'.format(text))

>>> hello()
hello world

>>> hello(text='universe')
hello universe
```

8.3.9 Klassen

Eine Klasse dient in objektorientierten Programmiersprachen als Vorlage zum Erstellen von Objekten. Ein Objekt besitzt Eigenschaften (mit dem Objekt assoziierte Variablen) und Methoden (Funktionen, die basierend auf den Eigenschaften eines Objekts operieren). Als Beispiel dient eine Neuron-Klasse, deren Instanzen Sigmoidneuronen mit Gewichten und

[31] Der ursprüngliche String wird eigentlich nicht verändert, sondern es wird ein neuer String erschaffen.

einer Neigung sind, die darauf basierend ihre Aktivierung zum einem gegebenen Input berechnen können.

`class Neuron:`

`self` bezieht sich in einer Klassendefinition auf eine Instanz. Eigenschaften von Objekten werden mit `self.<variable>` und Methoden mit `self.<methode>()` aufgerufen. `self` ist auch immer das erste Argument von Methoden. Die `__init__`-Methode wird aufgerufen, wenn ein neues `Neuron`-Objekt erstellt wird. Im Beispiel wird dem Objekt ein Gewichtsvektor und eine Neigung zugewiesen:

```
    def __init__(self, weights, bias):
        self.weights = weights
        self.bias = bias
```

Statische Methoden beziehen sich nicht auf eine Instanz der Klasse und sind daher nichts weiter als Funktionen, welche von der Klasse aus aufgerufen werden können. Sie werden mit einem `@staticmethod` auf der Zeile vor der Definition markiert. Die Berechnung von Sigmoid-Funktionswerten ist in der Beispielsklasse als statische Methode implementiert:

```
    @staticmethod
    def sigmoid(z):
        return 1/(1 + np.exp(-z))
```

Nun die Methode zur Berechnung der Aktivierung:

```
    def activation(self, x):
        if len(self.weights) != len(x):
            raise Exception('Input must have length {}'.format(len(self.weights)))

        z = np.dot(self.weights, x) + self.bias
        return Neuron.sigmoid(z)
```

Eine Instanz kann nun folgendermassen erstellt werden:

`n = Neuron(np.array([-1, 1]), -2)`

`n.activation(np.array([1, 3]))` gibt dann beispielsweise `0.5` zurück.

8.3.10 Plots mit Matplotlib

Matplotlib muss wie NumPy ebenfalls importiert werden, aus Konvention als `plt`.

`import matplotlib.pyplot as plt`

Die `scatter(<x>, <y>)`-Funktion von Matplotlib nimmt die Listen `x` und `y` , welche die x- und y-Koordinaten von Punkten enthalten. Die `scatter`-Funktion erstellt ein Scatterplot aus den Punkten $(x[n], y[n])$ für jedes n $(0 \leq n < \text{len}(x) = \text{len}(y))$. Um die Graphik anzuzeigen, muss `plt.show()` ausgeführt werden.

```
x = [0, 1, 2, 3]
y = [2, 1, 2, -1]
```

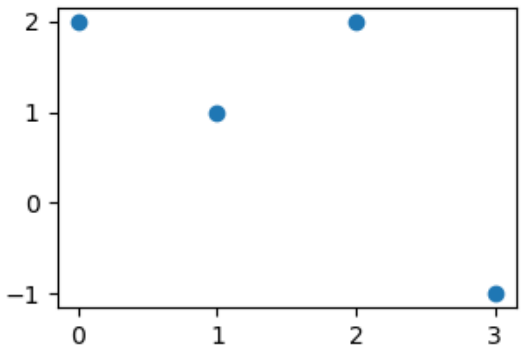


Abbildung 27: Beispiels-Scatterplot mit Matplotlib

```
plt.scatter(x, y)
plt.show()
```

Die `plot(<x>, <y>)`-Funktion von Matplotlib funktioniert ähnlich, anstatt die einzelnen Punkte zu zeichnen, werden die Punkte der x- und y-Listen durch eine gerade Linie verbunden. Wenn man genug Koordinaten nutzt, lassen sich so kontinuierliche Funktionen wie z.B. eine Normalparabel darstellen. Im folgenden Beispiel wird die Liste mit den x-Koordinaten durch die `np.linspace(<min>, <max>, num=<n>)`-Funktion generiert, die eine Liste mit `<n>` Zahlen von `<min>` bis `<max>` erstellt. Des Weiteren wird der Plot mit einem Titel, Achsenbeschriftungen und Gitternetzlinien ergänzt.

```
x = np.linspace(-3, 3, num=100)
y = [i*i for i in x]

plt.plot(x, y)
plt.title('Parabel (y=x^2)')
plt.xlabel('x-Achse')
plt.ylabel('y-Achse')
plt.grid(True, linestyle='-')
plt.show()
```

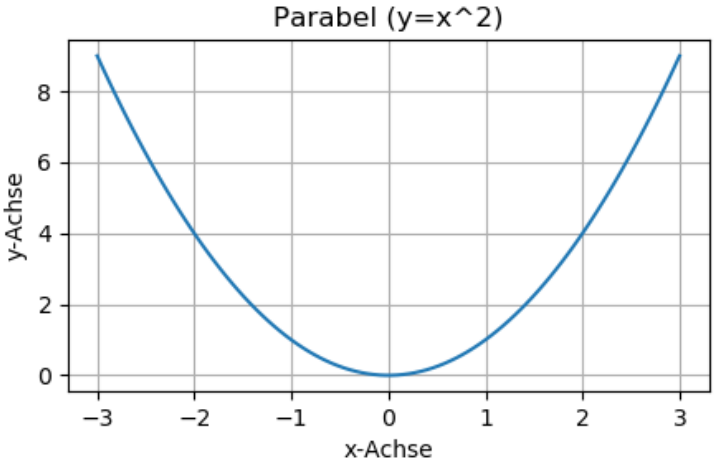


Abbildung 28: Beispielsplot mit Matplotlib

9 Literaturverzeichnis

Campbell, N. A., & Reece, J. B. (2011). Campbell Biologie. In N. A. Campbell, & J. B. Reece, *Campbell Biologie* (S. 609-618). Pearson.

Géron, A. (2017). Gradient Descent. In A. Géron, *Hands-On Machine Learning with Scikit-Learn & Tensorflow* (S. 113-114).

Géron, A. (2017). Model-based learning. In A. Géron, *Hands-On Machine Learning with Scikit-Learn & Tensorflow* (S. 18-20).

Géron, A. (2017). Supervised learning. In A. Géron, *Hands-On Machine Learning with Scikit-Learn & Tensorflow* (S. 8-9).

Géron, A. (2017). Xavier and He Initialization. In A. Géron, *Hands-On Machine Learning with Scikit-Learn & Tensorflow* (S. 279-280).

Goodfellow, I., Bengio, Y., & Courville, A. (2016). Momentum. In I. Goodfellow, Y. Bengio, & A. Courville, *Deep Learning Book* (S. Kapitel 8.3.2). Abgerufen am 12. August 2018 von http://www.deeplearningbook.org/contents/optimization.html

Goodfellow, I., Bengio, Y., & Courville, A. (2016). Universal Approximation Properties and Depth. In I. Goodfellow, Y. Bengio, & A. Courville, *Deep Learning* (S. Kapitel 6.4.1). Abgerufen am 12. August 2018 von http://www.deeplearningbook.org/contents/mlp.html

Hornik, K. (1990). *Approximation Capabilities of Muitilayer.* Abgerufen am 11. August 2018 von http://zmjones.com/static/statistical-learning/hornik-nn-1991.pdf

Nielsen, M. (2017). Backpropagation. In M. Nielsen, *Neural Networks and Deep Learning* (S. Chapter 2; The four fundamental equations behind backpropagation). Abgerufen am 6. Juli 2018 von http://neuralnetworksanddeeplearning.com/chap2.html#the_four_fundamental_equations_behind_backpropagation

Nielsen, M. (2017). Implementing our network to classify digits. In M. Nielsen, *Neural Networks and Deep Learning* (S. Chapter 1; Implementing our network to classify digits). Abgerufen am 22. September 2018 von http://neuralnetworksanddeeplearning.com/chap1.html#implementing_our_network_to_classify_digits

Nielsen, M. (2017). Learning with gradient descent. In M. Nielsen, *Neural Networks and Deep Learning* (S. Chapter 1; Learning with gradient descent). Abgerufen am 22. September 2018 von http://neuralnetworksanddeeplearning.com/chap1.html#learning_with_gradient_descent

Nielsen, M. (2017). Other techniques for regularization. In M. Nielsen, *Neural Networks and Deep Learning* (S. Chapter 3; Other techniques for regularization). Abgerufen am 22. September 2018 von http://neuralnetworksanddeeplearning.com/chap3.html#other_techniques_for_regularization

Nielsen, M. (2017). Perceptrons. In M. Nielsen, *Neural Networks and Deep Learning* (S. Chapter 1; Perceptrons). Abgerufen am 6. Juli 2018 von http://neuralnetworksanddeeplearning.com/chap1.html#perceptrons

Nielsen, M. (2017). Regularization. In M. Nielsen, *Neural Networks and Deep Learning* (S. Chapter 3; Regularization). Abgerufen am 22. September 2018 von http://neuralnetworksanddeeplearning.com/chap3.html#regularization

Nielsen, M. (2017). Sigmoid neurons. In M. Nielsen, *Neural Networks and Deep Learning* (S. Chapter 1; Sigmoid neurons). Abgerufen am 22. September 2018 von http://neuralnetworksanddeeplearning.com/chap1.html#sigmoid_neurons

Nielsen, M. (2017). The architecture of neural networks. In M. Nielsen, *Neural Networks and Deep Learning* (S. Chapter 1; The architecture of neural networks). Abgerufen am 7. August 2018 von http://neuralnetworksanddeeplearning.com/chap1.html#the_architecture_of_neural_networks)

Nielsen, M. (2017). The cross-entropy cost function. In M. Nielsen, *Neural Networks and Deep Learning* (S. Chapter 3; What does the cross-entropy mean? Where does it come from?). Abgerufen am 6. Juli 2018 von http://neuralnetworksanddeeplearning.com/chap3.html#what_does_the_cross-entropy_mean_where_does_it_come_from

Nielsen, M. (2017). Universality with one input and one output. In M. Nielsen, *Neural Networks and Deep Learning* (S. Chapter 4; Universality with one input and one output). Abgerufen am 11. August 2018 von http://neuralnetworksanddeeplearning.com/chap4.html#universality_with_one_input_and_one_output

Nielsen, M. (2017). Weight initialization. In M. Nielsen, *Neural Networks and Deep Learning* (S. Chapter 3; Weight initialization). Abgerufen am 6. Juli 2018 von http://neuralnetworksanddeeplearning.com/chap3.html#weight_initialization

Olah, C. (2014). *Neural Networks, Manifolds, and Topology.* Abgerufen am 22. September 2018 von Neural Networks, Manifolds, and Topology: https://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

Olah, C. (2015). *Calculus on Computational Graphs: Backpropagation.* Abgerufen am 22. September 2018 von Calculus on Computational Graphs: Backpropagation: https://colah.github.io/posts/2015-08-Backprop/

Wikipedia - Convolutional Neural Network. (2018). Abgerufen am 22. September 2018 von https://de.wikipedia.org/wiki/Convolutional_Neural_Network

Wikipedia - Hadamard-Produkt. (2018). Abgerufen am 8. August 2018 von https://de.wikipedia.org/wiki/Hadamard-Produkt

Wikipedia - Perzeptron. (2018). Abgerufen am 6. Juli 2018 von https://de.wikipedia.org/wiki/Perzeptron

Wikipedia - Rectifier (neural networks). (2018). Abgerufen am 22. September 2018 von https://en.wikipedia.org/wiki/Rectifier_(neural_networks)

Wikipedia - Rekurrentes neuronales Netz. (2018). Abgerufen am 22. September 2018 von https://de.wikipedia.org/wiki/Rekurrentes_neuronales_Netz

Wikipedia - Softmax function. (2018). Abgerufen am 22. September 2018 von https://en.wikipedia.org/wiki/Softmax_function

Wikipedia - Vanishing gradient problem. (2018). Abgerufen am 22. September 2018 von https://en.wikipedia.org/wiki/Vanishing_gradient_problem

10 Abbildungsverzeichnis

Beinahe alle Abbildungen wurden selbst erstellt, die Plots mit Matplotlib (Code verfügbar bei *https://tinyurl.com/MADeepLearning*) und die schematischen Darstellungen mit *https://www.draw.io/*. Die einzigen Ausnahmen sind:

- Abbildung 6: biologisches Neuron von *https://www.wpclipart.com/medical/anatomy/nervous_system/neuron/neuron.png.html*
- Abbildung 8: Heavisidestufenfunktion von *https://de.wikipedia.org/wiki/Heaviside-Funktion*

11 Tabellenverzeichnis

12 Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich diese Arbeit selbständig durchgeführt und keine anderen als die angegebenen Quellen, Hilfsmittel und Hilfspersonen beigezogen habe. Alle Textstellen in der Arbeit, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet.

_____

Reto Merz