Leipziger Institut für Informatik

Wintersemester 2017

# Seminararbeit

im Studiengang Informatik

der Universität Leipzig

Forschungsseminar Deep Learning

# Begriffsbildung, Konzepte und Überblick

Verfasser: Alexander Strätz

Matrikelnummer: 3671083

Betreuer: Ziad Sehili

Abgabetermin: 26.03.2018

# Inhaltsverzeichnis

# 1 Einleitung

Obwohl das Konzept der neuronalen Netze schon lange bekannt ist, sind diese erst seit wenigen Jahren in der Praxis angekommen. Dies liegt zum einen daran, dass früher nicht allzu viele getaggte Daten vorhanden waren. Diese sind für das Training der neuronalen Netze aber unerlässlich. Zum anderen ist viel Rechenpower nötig, damit die Lernphase in absehbarer Zeit abgeschlossen werden kann. Vor allem der effiziente Einsatz moderner Grafikkarten hat hier einen großen Leistungssprung herbeigeführt.

Deep Learning ist mittlerweile in vielen Bereichen des alltäglichen Lebens zu finden. Bei der Websuche wird es eingesetzt, um den Nutzern die Suchergebnisse anzuzeigen, die für diese am relevantesten sind. Im Bereich E-Commerce werden den Kunden auf ähnliche Weise Artikel empfohlen, die aufgrund ihrer vergangenen Einkäufe für diese interessant sein könnten. Auch für die Gesichtserkennung bei Kameras oder die Speech to Text-Funktion von Smartphones werden neuronale Netze genutzt. Auchs abseits des Informatiksektors wird Deep Learning immer beliebter. So werden zum Beispiel Bildaufnahmen von Tumorpatienten analysiert, um den Fortschritt der Krankheit beurteilen zu können. Und im Finanzbereich wird versucht mithilfe neuronaler Netze Aktienkurse vorhersagen zu können.

In dieser Arbeit wird zunächst das Konzept der neuronalen Netze am Beispiel der Filmempfehlung eingeführt. Im darauffolgenden Kapitel wird der Aufbau und die Funktionsweise genauer erläutert. Zuletzt wird noch auf praktische Probleme und Lösung beim Einsatz von Deep Learning eingegangen.

# 2 Einführung: Künstliche neuronale Netze

## 2.1 Beispiel: Filmbewertung

In diesem Kapitel wird das Konzept von neuronalen Netzen eingeführt. Zu diesem Zweck betrachten wird Abbildung 1. In diesem Szenario haben wird zwei Freunde Mary und John. Auf Grundlage ihrer Filmbewertungen möchten wir vorhersagen können, ob uns gewisse Filme gefallen würden oder nicht. Dabei reichen die Bewertungen von Mary und John von 1 (gar nicht gut) bis 5 (sehr gut). Zusammen mit unserer Meinung, ob uns der Film gefallen hat oder nicht, bildet dies unsere Trainingsdaten.

| Movie name | Mary's rating | John's rating | I like? |
|---|---|---|---|
| Lord of the Rings II | 1 | 5 | No |
| ... | ... | ... | ... |
| Star Wars I | 4.5 | 4 | Yes |
| Gravity | 3 | 3 | ? |

Abbildung 1: Tabelle Filmbewertungen [Le et al., 2015]

Unser Ziel ist es nun für den Film Gravity, den wir noch nicht gesehen haben, vorherzusagen, ob wir ihn anschauen sollen oder nicht. Diesen Szenario ist noch einmal in Abbildung 2 grafisch dargestellt. Auf den Achsen sind die Bewertungen unserer Freunde dargestellt. X bedeutet, dass

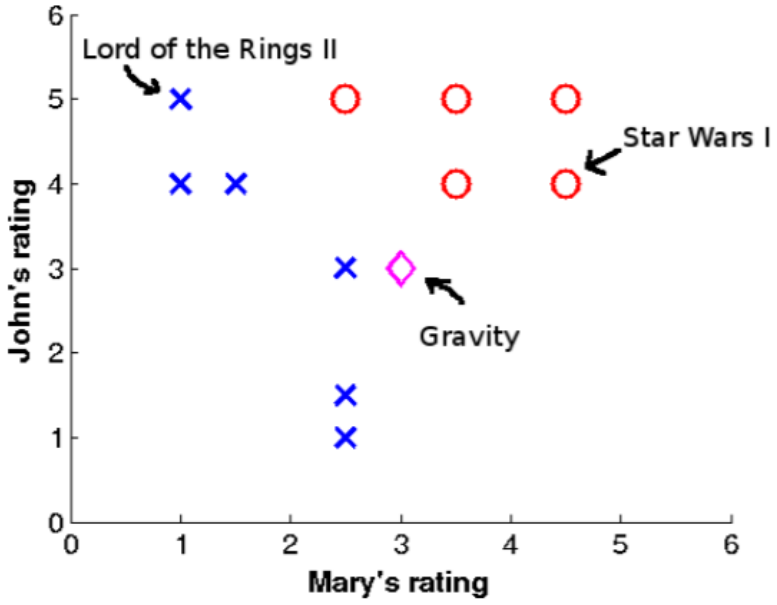


Abbildung 2: Visualisierung Filmbewertungen [Le et al., 2015]

uns der Film nicht gefallen hat und O, dass er uns gefallen hat.

Um nun auch zukünftige Filme vorhersagen zu können, müssen wir eine Entscheidungsfunktion aufstellen. Da die beiden Bereiche linear separierbar sind, können wir eine lineare Entscheidungsfunktion aufstellen.

$$h(x;\theta,b) = \Theta_1 x_1 + \Theta_2 x_2 + b = \Theta^T x + b \tag{1}$$

Die Entscheidungsfunktion ist dabei von den Bewertungen unserer Freunde $x_1$ und $x_2$ und den freien Parametern $\theta_1$, $\theta_2$ und $b$ abhängig.
Das Ergebnis der Entscheidungsfunktion kann irgendeine reelle Zahl sein. Da wir uns aber nur dafür interessieren, ob wir den Film anschauen sollen oder nicht, wollen wir den Wertebereich einschränken. Dies gelingt uns mit der Sigmoidfunktion (siehe Abbildung 3).

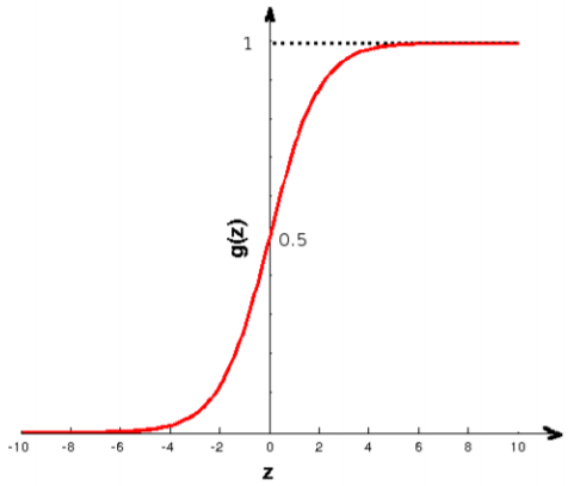


Abbildung 3: Sigmoid Funktion [Le et al., 2015]

Die Sigmoidfunktion ist folgendermaßen definiert: $\sigma(z) = \frac{1}{1+e^{-z}}$. Wie zu sehen ist, wird hiermit der Wertebereich auf zwischen 0 und 1 eingeschränkt. Dabei sagt uns eine 0, dass wir den Film nicht sehen sollen und eine 1, dass wir ihn sehen sollen.

## 2.2 Gradientenverfahren

Unser Ziel ist es, eine Entscheidungsfunktion zu haben, die für jedes Trainingsbeispiel möglichst genau korrekt eine 0 oder 1 vorhersagt. Mathematisch ausgedrückt, ist das Ziel also:

$h(x^{(1)};\theta,b) \approx y^{(1)}$
$h(x^{(2)};\theta,b) \approx y^{(2)}$
...

Dazu werden die Parameter $\Theta$ und $b$ zunächst zufällig initialisiert. Anschließend wird eine Kostenfunktion aufgestellt, die minimiert werden soll (siehe Gleichung 2):

$$\sum_{i=1}^{m}(h((x)^{(i)};\theta,b) - y^{(i)})^2. \tag{2}$$

Um die Kostenfunktion zu minimieren, wird das stochastische Gradientenabstiegsverfahren verwendet. Dabei werden die Parameter schrittweise in Richtung einer kleiner werdenden Kostenfunktion angeglichen.

$$\theta_1 = \theta_1 - \alpha\Delta\theta_1 \tag{3}$$
$$\theta_2 = \theta_2 - \alpha\Delta\theta_2 \tag{4}$$
$$b = b - \alpha\Delta b \tag{5}$$

$\alpha$ ist dabei eine nicht negative reelle Zahl und wird der Lernfaktor genannt. Dieser bestimmt wie schnell sich die Parameter ändern. Um sicher zu stellen, dass die Parameter so geändert werden, dass sich die Kostenfunktion ihrem Minimum nähert, müssen die partiellen Ableitungen der Kostenfunktion nach den betrachteten Parametern betrachtet werden (siehe Gleichung 6).

$$\Delta\theta_1 = \frac{\partial}{\partial\theta_1}((h(x)^{(i)};\theta,b) - y^i)^2 \tag{6}$$

In Abbildung 4 wird das Gradientenabstiegsverfahren dargestellt. Die Kostenfunktion C ist hier nur von den zwei Parametern $v_1$ und $v_2$ anhängig. Allerdings ist die Vorgehensweise für mehr als zwei Parameter analog. Die grüne Kugel gibt an, weche Werte die Parameter im Moment besitzen. Der grüne Pfeil zeigt, wie sich die Parameter in Richtung der minimalen Kostenfunktion verschieben. Die Länge des Pfeils wird dabei durch den Lernfaktor bestimmt.

Das Verfahren des Gradientenverfahrens lässt sich folgendermaßen zusammenfassen:

1. Initialisiere Parameter $\theta$ und $b$ zufällig
2. Wähle zufälliges Beispiel $x^{(i)}, y^{(i)}$
3. Berechne die partiellen Ableitungen für $\theta_1$, $\theta_2$ und $b$

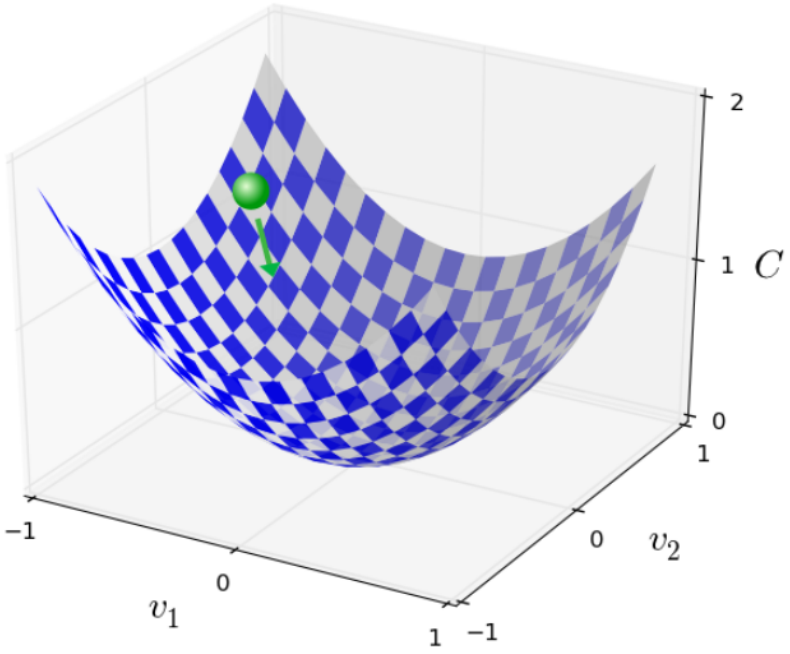


Abbildung 4: Gradientenverfahren [Nielsen, 2015]

4. Aktualisiere die Parameter, zurück zu Schritt 2

Das Gradientenverfahren wird gestoppt, wenn sich die Parameter nicht mehr ändern oder nach einer festgelegten Anzahl von Iterationen. Schließlich bestimmt die Entscheidungsfunktion, ob man den Film anschauen soll ($h > 0,5$) oder ob man ihn nicht anschauen soll ($h < 0,5$). Dies ist analog dazu, ob sich die Filme oberhalb oder unterhalb der Entscheidungsgeraden befinden (siehe Abbildung 5).

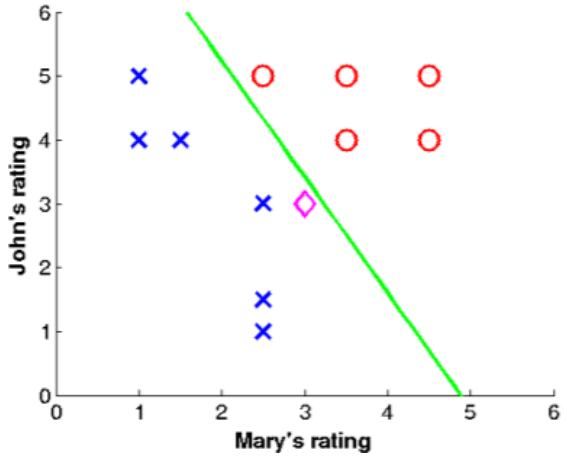


Abbildung 5: Lineare Entscheidungsfunktion [Le et al., 2015]

In Abbildung 6 ist das neuronale Netz zu unserem einfachen Szenario zu sehen. Aufgrund der linearen Separierbarkeit der Klassen ist nur eine Inputschicht für die Bewertungen von John und Mary und eine Outputschicht mit dem Ergebnis vonnöten. Sollte das Problem erwartungsgemäß nicht linear zu lösen sein, so müssen weitere Schichten zum neuronalen Netz hinzugefügt werden.

In den Abbildungen 7 und 8 ist zu sehen, wie eine weitere Schicht dabei hilft das Problem auf den linearen Fall zurückzuführen. Dabei löst die erste Entscheidungsfunktion den oberen Problembe-

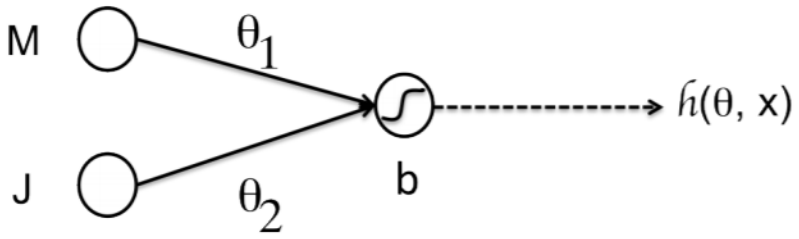


Abbildung 6: Neuronales Netz [Le et al., 2015]

reich und ignoriert die Kreise in der linken unteren Ecke und Entscheidungsfunktion 2 macht das selbe analog für den unteren Problembereich. Die Funktion in der Outputschicht fasst schließlich die beiden Resultate zusammen und gibt das Ergebnis aus.

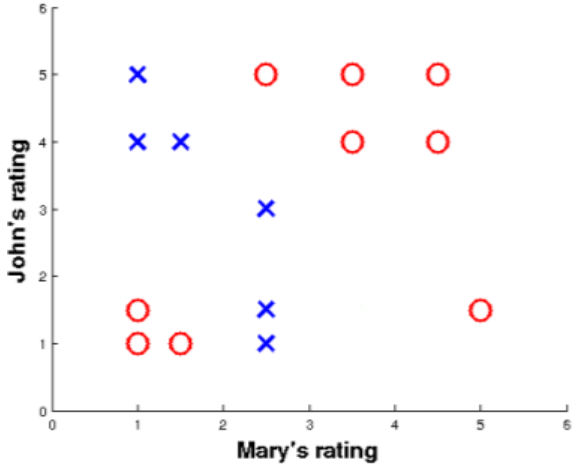


Abbildung 7: Nichtlinearer Fall:: Grafik [Le et al., 2015]

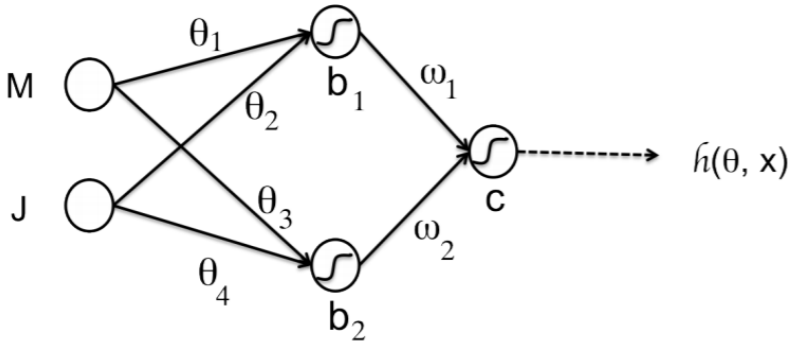


Abbildung 8: Nichtlinearer Fall: Schema [Le et al., 2015]

# 3 Aufbau künstlicher neuronaler Netze

In Abbildung 9 ist der Aufbau eines künstlichen Neurons zu sehen. Diese sind die Grundbausteine eines jeden neuronalen Netzes. Jedes Neuron erhält $x_1$ bis $x_n$ Inputwerte, die jeweils mit einem Gewicht $w$ multipliziert werden. Anschließend werden alle Ergebnisse zusammen mit einem Bias-Wert, dessen Gewicht immer eins ist, aufsummiert und in die Aktivierungsfunktion gegeben. Diese gibt schließlich ein Ergebnis aus und gibt es an das nächste Neuron weiter.

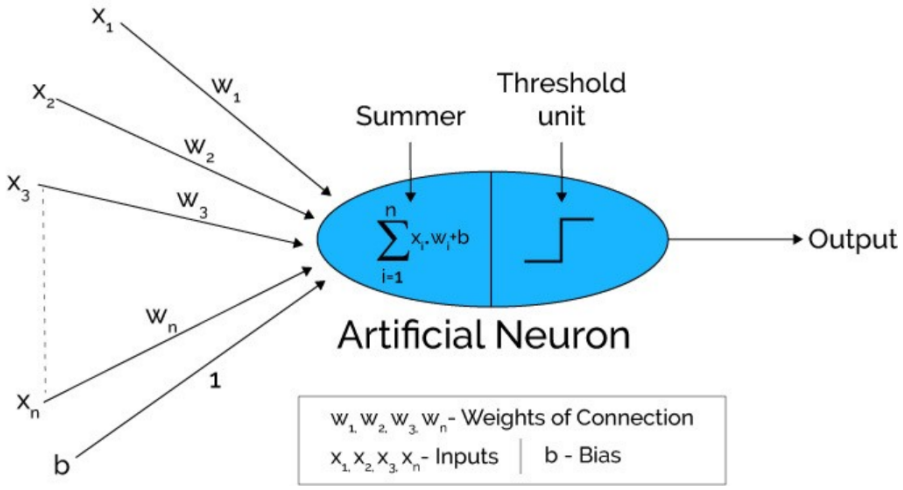


Abbildung 9: Aufbau des künstlichen Neurons [Xenostack, 2017]

Mit der Sigmoidfunktion wurde bereits im vorherigen Kapitel eine mögliche Aktivierungsfunktion vorgestellt. Eine weitere Aktivierungsfunktion, die sich immer größerer Beliebtheit erfreut, ist die Rectified Linear Unit (siehe Abbildung 10). Diese gibt für negative Eingabewerte eine Null zurück und für positive Eingabewerte die Identität.

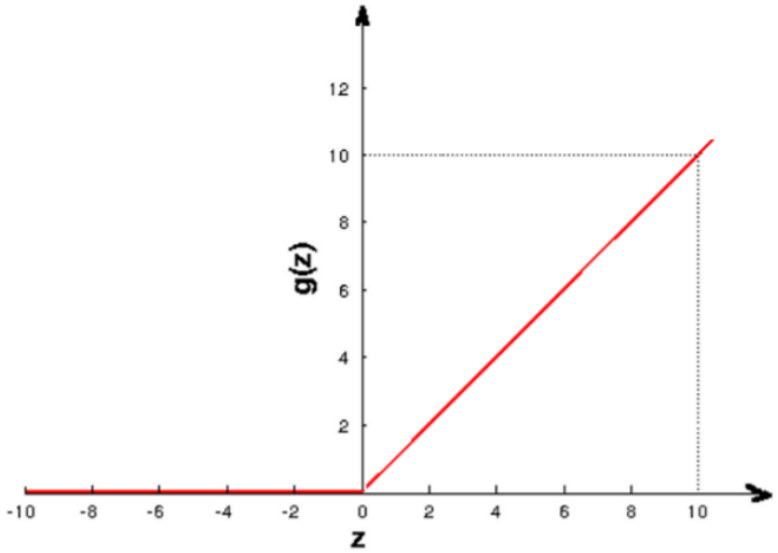


Abbildung 10: Rectified Linear Unit [Le et al., 2015]

Ein neuronales Netz besteht aus mehreren Schichten von künstlichen Neuronen. Die erste Schicht, die für die Eingabewerte verantwortlich ist, wird der Input Layer genannt. Die letzte Schicht, die der Output Layer genannt wird, gibt die Ergebnisse aus. Alle Schichten dazwischen werden Hidden Layers genannt (siehe Abbildung 11 ).

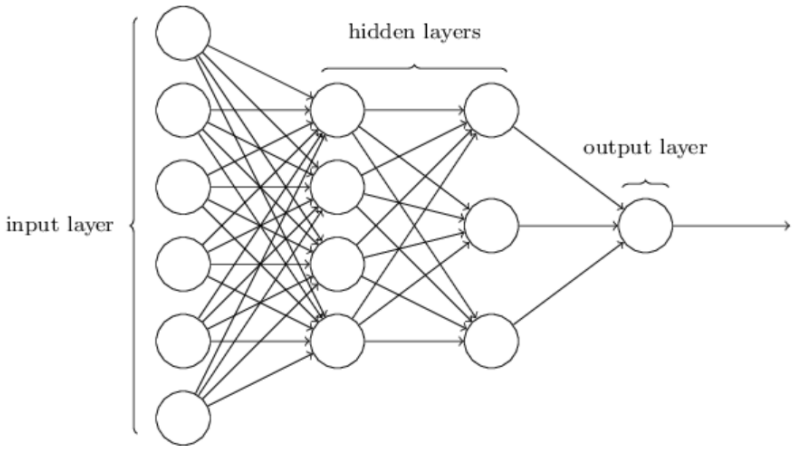


Abbildung 11: Schichten im KNN [Nielsen, 2015]

Während die Input- und die Outputschicht meist vom Problemfall vorgegeben sind, besitzt man in der Gestaltung des Hidden Layers große Freiheiten. In Abbildung 12 und 13 werden zwei generelle Aufbaumöglichkeiten vorgestellt.

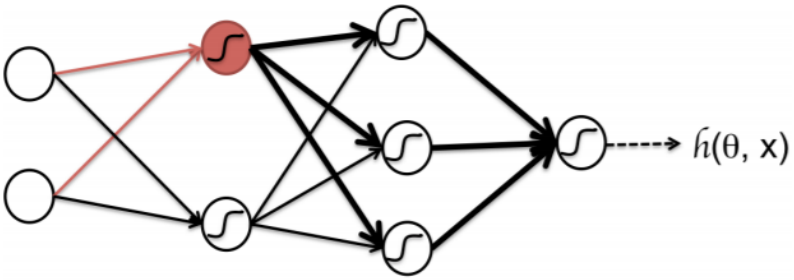


Abbildung 12: Tiefes Netz [Le et al., 2015]

Man kann entweder ein Netz gestalten, dass viele Schichten besitzt und dafür nur wenige Knoten pro Layer oder man entwirft ein flaches Netz mit wenigen großen Schichten. Allerdings hat sich gezeigt, dass ein Shallow Layer wenig Sinn macht und bei gleicher Neuronenzahl fast immer ein schlechteres Ergebnis als ein Deep Layer vorweist. Das liegt daran, dass bei einem tiefen Netz das Resultat einzelner Neuronen öfters wiederverwendet wird.

In den Abbildungen ist das am Beispiel des rot eingefärbten Neurons zu sehen. Im tiefen Netz wird das Resultat des betrachteten Neurons an drei weitere Neuronen geschickt, die alle den Wert in ihre Entscheidungsfunktion mit einfließen lassen. Im flachen Fall hingegen wird das Ergebnis direkt an die Outputschicht prozessiert und dieses hat somit keinen Einfluss auf den Rest des neuronalen Netzes.

Alle betrachteten Beispiele waren bis jetzt Feedforward-Netze, bei denen die Ergebnisse immer von der vorherigen Schicht zur nächsten weitergegeben werden. Außerdem waren sie zusätzlich vollständig verknüpft, das heißt ein Neuron gibt sein Ergebnis an alle Neuronen der nächsten Schicht weiter. Beides ist nicht zwingend erforderlich. Es haben sich bestimmte Designprinzipien für den Hidden Layer bewährt.

So gibt es zum Beispiel das Convolutional Neural Network (CNN), bei dem das Netz aus Convolutional- und Poolinglayers besteht. Oft wird diese Art von Netz bei der Verarbeitung von Bilddaten verwendet. Zuerst wird hier beispielsweise eine Kantenerkennung durchgeführt und im nächsten Schritt werden die Daten reduziert, indem mehrere Werte zu ihrem Maximum zusammengefasst werden.

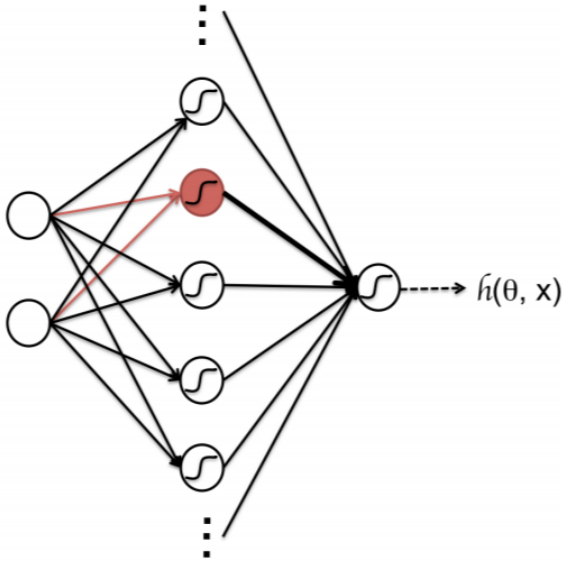


Abbildung 13: Flaches Netz [Le et al., 2015]

Ein Netz bei dem die Werte nicht nur an die nächste Schicht weitergegeben werden, ist das Recurrent Neural Network (RNN). Hier sind auch Verbindungen zu Neuronen derselben oder einer vorangegangen Schicht erlaubt. Dies entspricht einer Art von Gedächtnis für die Aktivierungsdaten und ist vor allem bei der Verarbeitung von Sequenzen von Vorteil. Ein Anwendungsfall für RNNs ist zum Beispiel die Spracherkennung.

## 3.1 Backpropagation Algorithmus

Neben dem Gradientenverfahren ist der Backpropagation Algorithmus das wichtigste Prinzip eines neuronalen Netzes. Dieser ist dafür zuständig die partiellen Ableitungen der Kostenfunktion nach den Parametern zu berechnen. Diese sind zwingend erforderlich, um zu wissen, in welche Richtung die Parameter im Rahmen des Gradientenverfahrens zu verschieben sind, um das Minimum der Kostenfunktion zu erreichen.

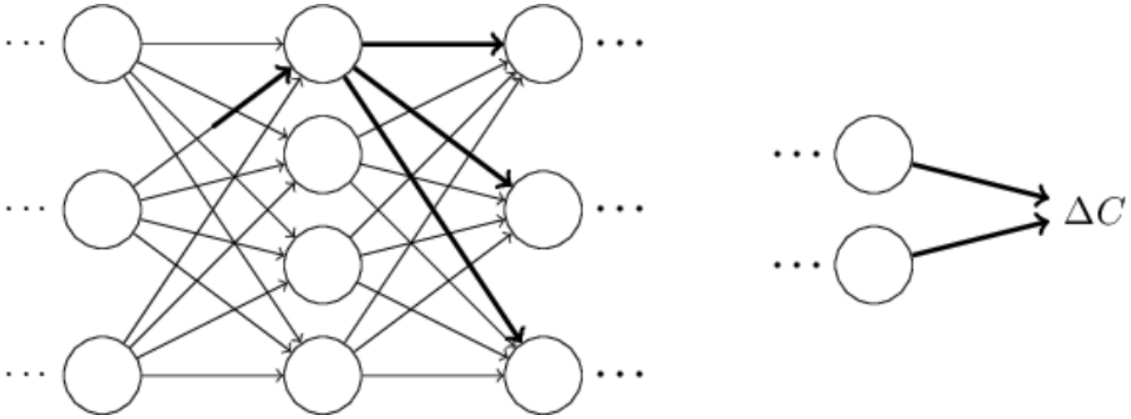


Abbildung 14: Backpropagation [Nielsen, 2015]

In Abbildung 14 wird angedeutet, wie kompliziert die Berechnung der Ableitung für die Kostenfunktion sein kann. Wird der Weitergabewert eines einzelnen Neurons abgeändert, wirkt sich dies auf alle Neuronen der nächsten Schicht und somit auf die Aktivierungsfunktion aller weiteren Neuronen und schließlich auf die Kostenfunktion aus.

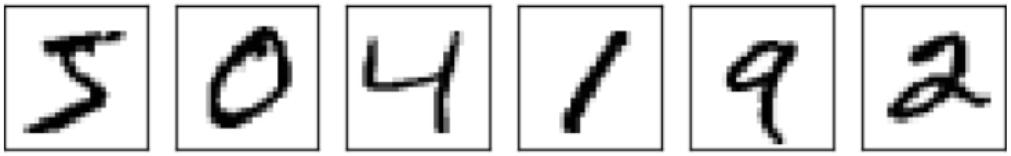

Abbildung 15: Handgeschriebene Zahlen [Nielsen, 2015]

Eine naive Vorgehensweise wäre für jeden Parameter die Ableitung näherungsweise zu berechnen, indem das Netz zweimal mit einem leicht abgeänderten Parameterwert durchlaufen wird. Allerdings ist dies bei einer Neuronenanzahl von mehr als 100.000, wie es in der Praxis oft vorkommt, viel zu rechenaufwendig.

Daher berechnet der Backpropagation-Algorithmus zunächst den Outputfehler. Das ist der Fehler, der in der Ausgabeschicht entsteht, wenn die Eingabe in die letzte Schicht leicht abgeändert wird. Ausgehend von diesem lässt sich dieser Fehler für jede weitere Schicht zurückrechnen, so dass man den Ausgabefehler für jedes Neuron erhält. Mit diesen lässt sich anschließend leicht die Ableitungen der Kostenfunktionen nach allen Gewichten und Biases berechnen. Im Folgenden ist der Backpropagation-Algorithmus mit den mathematischen Details zusammengefasst:

- **Input**: Setze die Aktivierung $a^1$ für den Input-Layer
- **Feedforward**: Berechne Input $z^l = w^l a^{l-1} + b^l$ und Aktivierung $a^l = \sigma(z^l)$
- **Output Fehler**: $\delta^l = \nabla_a C \odot \sigma'(z^L)$
- **Backpropagation des Fehlers**: $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$
- **Output**: Berechne $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$ und $\frac{\partial C}{\partial b^l_j} = \delta^l_j$

Zunächst wird der Input für die erste Schicht gesetzt und das neuronale Netz wird vorwärsts durchlaufen. Anschließend wird der Outputfehler ausgehend von der letzten Schicht in die umgekehrte Richtung zurückgerechnet. Somit können alle partiellen Ableitungen mit nur zwei Netzdurchläufen berechnet werden.

## 3.2 Beispiel: Zahlen erkennen

Ein bekanntes Beispiel bei der Anwendung von neuronalen Netzen ist das automatische Erkennen von Handschrift. Im Folgenden werden als Teilproblem davon handgeschriebene Zahlen betrachtet. Diese sollen nun vom KNN den korrekten Ziffern zugeordnet werden. Im betrachteten Fall handelt es sich um 64x64 Pixel Graustufenbilder. (siehe Abbildung 15).

Für den Inputlayer bietet sich somit eine Schicht aus $64 * 64 = 4096$ Neuronen an. Jedes Neuron entspricht einem Pixel und nimmt einen Wert zwischen 0 (weiß) und 1 (schwarz) an. Da es 10 verschiedene Klassen gibt, denen eine Ziffer zugeordnet werden kann, nutzen wir eine Schicht aus 10 Output-Neuronen. Das Bild wird entsprechend der Klasse zugeordnet, deren Output-Neuron den höchsten Wert zurückgibt. Für den Hidden Layer könnte man zum Beispiel eine einfache Schicht aus 15 Neuronen wählen. Der gesamte Aufbau des KNNs kann in Abbildung 16 betrachtet werden.
Obwohl das so konstruierte Netz sehr einfach gestaltet ist, kann es über 95% der Fälle richtig einordnen. Die Techniken, die erforderlich sind, um die Erfolgsrate zu steigern, werden im nächsten Kapitel vorgestellt.

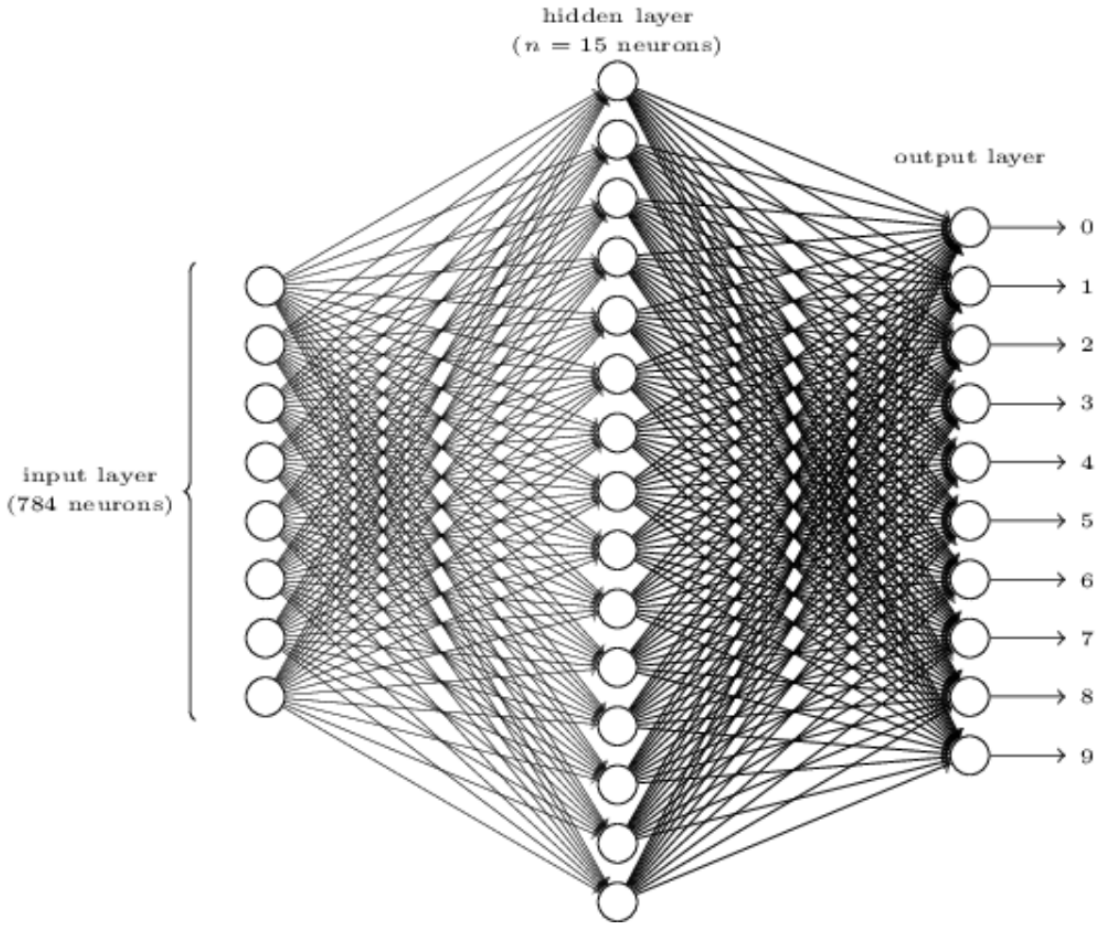


Abbildung 16: Aufbau des neuronalen Netzes [Nielsen, 2015]

# 4 Probleme/Lösungen

## 4.1 Overfitting

Ein Problem, das bei der Verwendung von KNNs adressiert werden muss, ist Overfitting. In Abbildung 17 ist zu sehen, wie die Gesamtkosten auf den Trainingsdaten mit fortschreitendem Training erwartungsgemäß immer weiter sinken.

Allerdings kann man in Abbildung 18 erkennen, dass die Klassifikationsgenauigkeit hingegen ab etwa Epoche 280 stagniert. Der Grund hierfür ist, dass sich das neuronale Netz immer weiter an die Testdaten anpasst. Damit verliert es aber die Fähigkeit auf anders aussehende Beispiele zu generalisieren.

Eine Lösung hierfür ist, die Klassifikationsgenauigkeit fortwährend zu überprüfen. Sollte sich diese ab einem gewissen Zeitpunkt nicht mehr verbessern, so wird der Algorithmus gestoppt. Natürlich unterliegt die Klassifikationsgenauigkeit gewissen Schwankungen, weswegen der Lernvorgang erst nach, zum Beispiel 10 aufeinanderfolgenden Epochen ohne Verbesserung, abgebrochen werden sollte.

Ein alternativer Lösungsansatz der Overfitting verhindert, ist die Regularisierung. Dazu wird die Kostenfunktion neu definiert. In Formel 7 ist die L2-Regularisierung zu sehen.

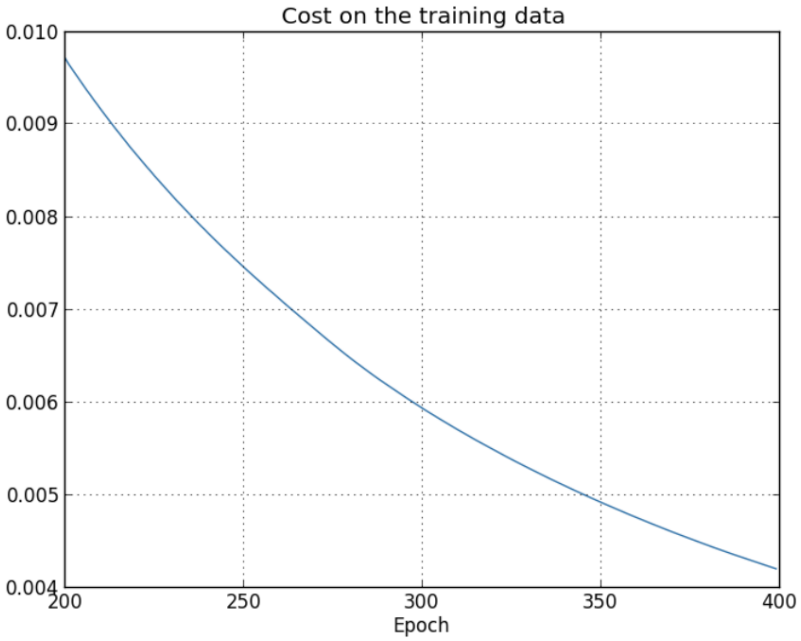


Abbildung 17: Gesamtkosten auf den Trainingsdaten [Nielsen, 2015]

$$C_{neu} = C_{alt} + \lambda \sum_{w} w^2 \tag{7}$$

Zur alten Kostenfunktion werden hier noch die quadratischen Gewichte addiert. Dies führt dazu, dass kleine Gewichte vorgezogen werden. Lambda ist dabei ein Faktor, der bestimmt, wie stark die Gewichte schrumpfen sollen. Kleinere Gewichte führen dazu, dass sich das Netz nicht so stark auf einzelne Knoten verlassen kann und mehr Knoten bei der Klassifizierung mit einbezogen werden müssen. Dies wirkt automatisch einem Overfitting entgegen.

$$C_{neu} = C_{alt} + \lambda \sum_{w} |w| \tag{8}$$

In Formel 8 ist die L1-Regularisierung zu sehen. Das Prinzip ist hier ähnlich. Statt der quadratischen Gewichte wird jedoch deren Betrag in der Kostenfunktion verwendet. Dadurch schrumpfen die Gewichte um einen konstanten Betrag. Die Folge ist, dass die meisten Gewichte gegen Null gehen und die übrigen wenigen Gewichte einen hohen Wert annehmen. Dies führt dazu, dass einfache Erklärungen bei der Klassifikation bevorzugt werden. Diese lassen sich in der Regel besser generalisieren als Lösungen, die ein komplexeres Muster an Knoten verwenden.

Eine weitere Technik zur Behandlung von Overfitting, ist der Dropout. Hier wird bei jedem Trainingsdurchgang zufällig die Hälfte der Neuronen im Hidden Layer gelöscht. Um den generellen Betrag der Gewichte nicht zu verfälschen, werden anschließend die Gewichte, die von den Neuronen im Hidden Layer ausgehen, halbiert. Dadurch, dass in jedem Trainingsdurchgang jedes beliebige Neuron gelöscht werden könnte, kann sich der Algorithmus nicht auf einzelne Knoten verlassen. So muss er auch alternative Lösungsmöglichkeiten entwickeln. Dem Overfitting wird so effektiv entgegen gewirkt.

Die größte Ursache für die Entstehung von Overfitting ist die Tatsache, dass zu wenige Trainingsdaten zur Verfügung stehen. Dadurch, dass das KNN immer wieder auf den gleichen Daten trainiert, verliert es die Fähigkeit auf neue Daten zu generalisieren. Die offensichtliche Lösung ist, mehr Trainingsdaten zu beschaffen. Doch dies ist oft nur schwer möglich. In diesen Fällen

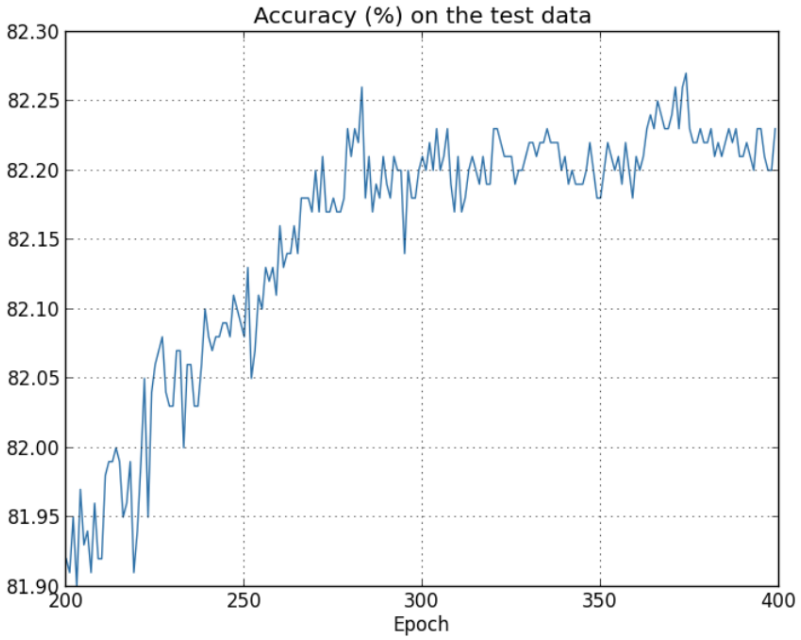


Abbildung 18: Genauigkeit der Klassifikation auf den Testdaten [Nielsen, 2015]

kann es von Vorteil sein, die originalen Trainingsdaten leicht zu modifizieren. So kann man zum Beispiel bei Bilddaten die Trainingsbilder durch Translation und Rotation leicht verändern. So kann man seinen Trainingsdatenpool leicht erweitern und das so entwickelte KNN kann besser generalisieren.

## 4.2 Initialisierung der Gewichte

In Kapitel 2 wurde gesagt, dass die Neuronengewichte zufällig initialisiert werden. Dies könnte man zum Beispiel mit einem Mittelwert von 0 und einer Standardabweichung von 1 verwirklichen. Betrachten wird in diesem Szenario ein einzelnes Neuron und gehen beispielsweise von 1000 eingehenden Neuronenverbindungen aus. Den Wert der Neuronen setzen wir zu 50% auf 0 und zu 50% auf 1.

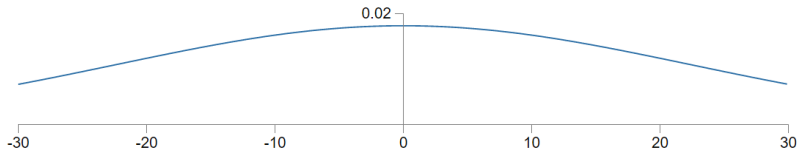


Abbildung 19: Eingangswert des Neurons [Nielsen, 2015]

In Abbildung 19 ist nun der erwartete Eingangswert für das betrachtete Neuron abgebildet. Wie man sieht, handelt es sich um eine sehr breite Wahrscheinlichkeitsdichtefunktion, bei der es äußerst wahrscheinlich ist, dass Werte sehr viel größer als 1 oder sehr viel kleiner als -1 angenommen werden. Warum dies wichtig ist, ist erkennbar, wenn man sich die Ableitung der Signmoid-Funktion anschaut (siehe Abbildung 20)

Den Maximalwert 0,25 nimmt die Ableitung bei $z = 0$ an. Zu den Rändern hin nimmt die Funktion sehr stark ab und geht schon ab Werten von $z = \pm 6$ gegen Null. In unserem Szenario war es jedoch sehr wahrscheinlich, dass der Betrag des Eingangswerts größer als sechs ist. Dies führt dazu, dass die Ableitung oft Werte nahe Null annimmt. Diese Ableitung geht beim Gradientenverfahren

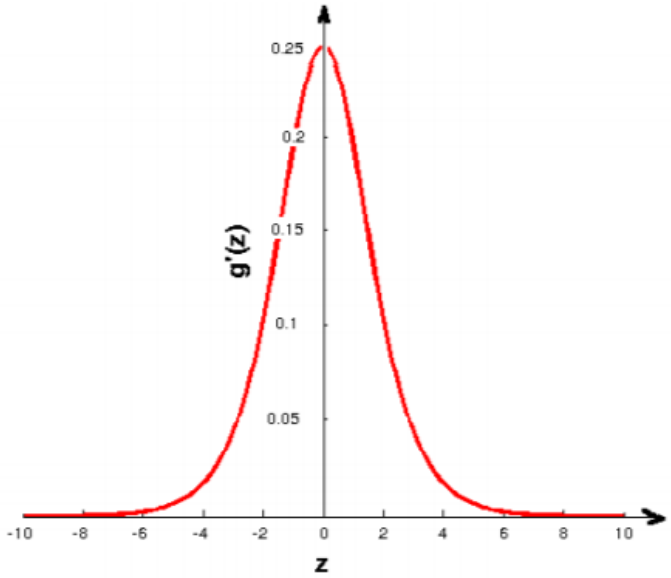


Abbildung 20: Ableitung der Sigmoid-Funktion [Nielsen, 2015]

jedoch direkt in den Lernfaktor ein. Demnach werden die Gewichte in jeder Epoche nur minimal geändert und das Lernen des KNNs findet nur äußerst langsam statt.

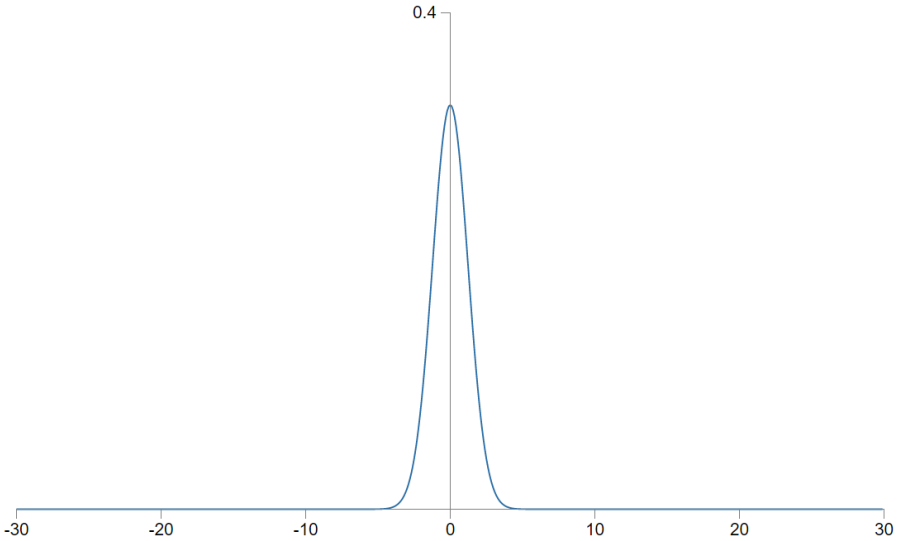


Abbildung 21: Eingangswert des Neurons [Nielsen, 2015]

Um dies zu verhindern, kann die Anzahl der eingehenden Neuronen bei der Initialisierung der Gewichte mit einbezogen werden. Ein guter Wert für die Initialisierung ist zum Beispiel ein Mittelwert von Null mit Standardabweichung $1/\sqrt{n_{in}}$, wobei $n_{in}$ die Anzahl der eingehenden Knoten ist. Dies führt zu einer sehr viel schmaleren Wahrscheinlichkeitsdichtefunktion wie in Abbildung 21 zu sehen ist. Dadurch, dass die Ableitung nun nicht mehr so oft Werte in der Nähe von Null annimmt, findet das Lernen der Parameter sehr viel schneller statt.

## 4.3 Problem des verschwindenden Gradienten

Bis jetzt wurden nur Beispiele mit einem Hidden Layer betrachtet. Deep Learning bezieht sich aber auf KNNs mit mehreren Hidden Layers. Und auch wenn Netze mit mehreren Hidden Layers mächtiger sind, sind bei deren Anwendung einige Dinge zu beachten. In Abbildung 22 ist die Lerngeschwindigkeit eines KNNs mit vier Hidden Layers abgebildet.

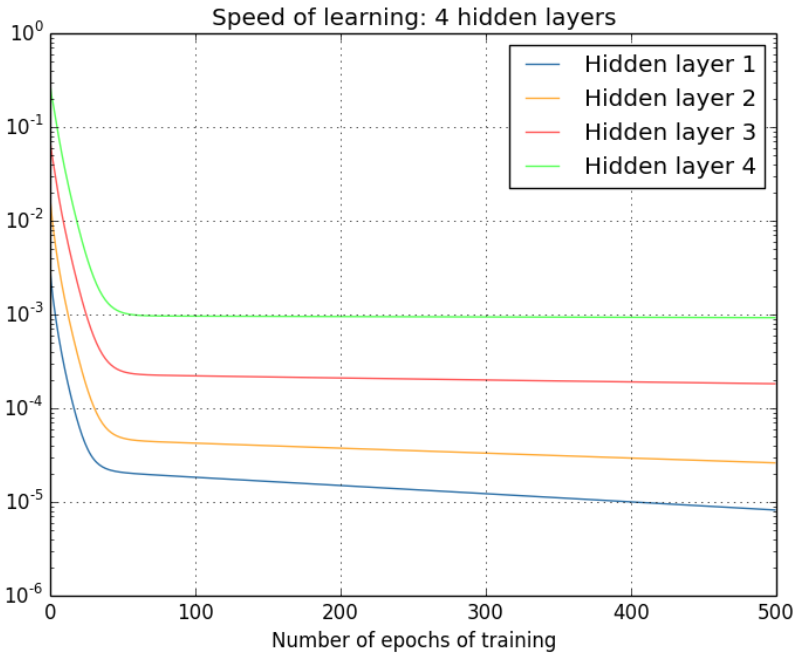


Abbildung 22: Geschwindigkeit des Lernens [Nielsen, 2015]

Es ist auffällig, dass das letzte Hidden Layer am schnellsten lernt und die Geschwindigkeit des Lernens von hinten nach vorne stetig abnimmt. So lernt die erste Schicht im Vergleich zur letzten je nach Epoche bis zu hundertmal langsamer. Dies ist natürlich ein großes Problem und führt dazu, dass sich die Gewichte der vorderen Schichten kaum ändern.

Der Grund hierfür ist die Ableitung der Kostenfunktion, die in den Lernterm als Faktor eingeht. Will man die Ableitung nach einem der vorderen Gewichte berechnen, setzt sich diese nach der Ableitungsregel aus den Faktoren der Ableitungen aller nachfolgenden Knoten zusammen (siehe Abbildung 23). Hier wird die Berechnung der Ableitung bei einem einfachen Netz gezeigt, bei dem jede Schicht aus nur einem Knoten besteht.

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1) \times w_2 \times \sigma'(z_2) \times w_3 \times \sigma'(z_3) \times w_4 \times \sigma'(z_4) \times \frac{\partial C}{\partial a_4}$$

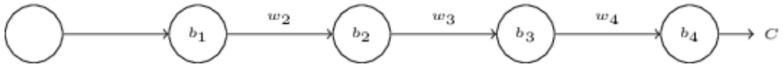


Abbildung 23: Ableitungsregel [Nielsen, 2015]

Dass dies ein Problem darstellt, wird offensichtlich, wenn man sich die Ableitung der Sigmoidfunktion in Erinnerung ruft (Abbildung 20). Sie nimmt Werte zwischen 0 und 0,25 an. Multipliziert man solche Werte mehrmals miteinander, wird der Term immer kleiner. Dadurch dass der Gradient in den früheren Schichten ein Produkt der Terme aus den späteren Schichten ist, entsteht eine instabile Situation. Dies führt in den meisten fällen zu einem verschwindenden Gradienten.

Dieses Problem wird entschärft, wenn man statt der Sigmoidfunktion die ReLU-Funktion verwendet (Abbildung 10). Die Ableitung ist hier für Werte größer Null immer Eins. Dadurch kann eine deutliche Verbesserung bei der Lerngeschwindigkeit in den vorderen Schichten erzielt werden.

# 5 Zusammenfassung

Neuronale Netze sind vor allem für Aufgaben gut geeignet, die vom Menschen intuitiv gelöst werden. Darunter fallen zum Beispiel die Spracherkennung, Gesichtserkennung und viele andere Bereiche. Vor allem die großen Mengen an getaggten Daten, die heutzutage zur Verfügung stehen und immer mächtiger werdende Grafikkarten fördern den Einsatz von künstlichen neuronalen Netzen immer weiter. Dabei setzen KNNs im Vergleich zu konventionellen Machine Learning Methoden auf eine Reihe von hierarchischer Schichten, um die Eingabedaten automatisch interpretieren zu können.

Den einzigen Vorwurf, den man KNNs machen kann, ist das sich in vielen Bereichen noch auf empirische Erkenntnisse verlassen wird. So gibt es noch keine klaren Richtlinien, wenn es um die Gestaltung des Hidden Layers, der Aktivierungsfunktion oder anderer Hyperparameter geht. Deswegen ist, auch wenn die Ergebnisse von Deep Learning schon jetzt vielversprechen sind, in Zukunft weitere Forschung nötig.

# Literatur

[Le et al., 2015] Le, Q. V. et al. (2015). A tutorial on deep learning part 1: Nonlinear classifiers and the backpropagation algorithm.

[LeCun et al., 2015] LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444.

[Nielsen, 2015] Nielsen, M. A. (2015). Neural networks and deep learning.

[Xenostack, 2017] Xenostack (2017). Overview of artificial neural networks and its applications. Online erhältlich unter https://www.xenonstack.com/blog/overview-of-artificial-neural-networks-and-its-applications abgerufen am 10. Januar 2018.