



# **Clientseitiges Deep Learning durch Klassifizierung von deutschsprachigen Clickbaits**

Ugur Tigu

Master-Thesis

zur Erlangung des akademischen Grades Master of Science (M.Sc.)

Studiengang Wirtschaftsinformatik

Fakultät IV - Institut für Wissensbasierte Systeme und  
Wissensmanagement

Universität Siegen

14. Dezember 2020

Betreuer

Prof. Dr.-Ing. Madjid Fathi, Universität Siegen

Johannes Zenkert, Universität Siegen

**Tigu, Ugur:**

Clientseitiges Deep Learning durch Klassifizierung von deutschsprachigen Clickbaits / Ugur Tigu. –

Master-Thesis, Aachen: Universität Siegen, 2020. 25 Seiten.

**Tigu, Ugur:**

Client-side deep learning through classification of German clickbaits / Ugur Tigu. –

Master Thesis, Aachen: University of Siegen, 2020. 25 pages.

## **Erklärung**

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich bin damit einverstanden, dass meine Arbeit veröffentlicht wird, d. h. dass die Arbeit elektronisch gespeichert, in andere Formate konvertiert, auf den Servern der Universität Siegen öffentlich zugänglich gemacht und über das Internet verbreitet werden darf.

Aachen, 14. Dezember 2020

Ugur Tigu



# Abstract

## ***Clientseitiges Deep Learning durch Klassifizierung von deutschsprachigen Clickbaits***

Ein im Internet weit verbreitetes Phänomen sind *Clickbaits-Nachrichten* (auf deutsch „Klickköder“). Ziel dieser Arbeit ist die Entwicklung eines Deep Learning Verfahrens, welches deutsche Clickbait Nachrichten automatisch erkennen soll. Die Arbeit stellt einen Datensatz vor, welches aus zwei Klassen von Nachrichten Überschriften besteht und zum trainieren eines Deep Learning Ansatzes verwendet wird. Dieser Datensatz wird durch Web Scraping erstellt und gelabelt. Das Ergebnis dieser Arbeit ist ein Modell für die Textklassifizierung, entwickelt in TensorFlow.js. Dieses Modell wird vollständig clientseitig in den Browser eingebettet und benötigt somit keinen Server.

## ***Client-side deep learning through classification of German clickbaits***

A widespread phenomenon on the Internet are clickbaits. The aim of this thesis is the development of a deep learning model which should automatically recognize German clickbait titles. The thesis presents a data set, which consists of two classes of news headlines and is used to train a deep learning approach. This data set is created using web scraping and hand-labeled. The result of this work is a model for text classification, developed in TensorFlow.js. This model is completely embedded in the browser on the client side and therefore does not require a server.



# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
<b>2</b>	<b>Deep Learning</b>	<b>3</b>
2.1	Einleitung . . . . .	3
2.2	Das Perzeptron . . . . .	5
2.3	Bias . . . . .	6
2.4	Mehrschichtiges Perzeptron . . . . .	6
2.5	Sigmoid-Neuron . . . . .	8
2.6	Aktivierungsfunktionen . . . . .	8
2.6.1	Sigmoid . . . . .	9
2.6.2	Tanh . . . . .	10
2.6.3	ReLU . . . . .	10
2.7	Optimierungsalgorithmen . . . . .	11
2.7.1	Batch Gradientenabstiegsverfahren . . . . .	13
2.7.2	Stochastische Gradientenabstiegsverfahren . . . . .	13
2.8	Lernrate im Deep Learning . . . . .	14
2.9	Unteranpassung und Überanpassung . . . . .	14
2.10	Regularisierung . . . . .	15
2.10.1	Early Stopping . . . . .	16
2.10.2	Dropout . . . . .	16
2.10.3	L1 und L2 . . . . .	17
2.11	Verlustfunktion und Kreuzentropie . . . . .	18
2.11.1	Verlustfunktionen . . . . .	18
2.11.2	Kreuzentropie . . . . .	19
2.12	Convolutional Neural Network . . . . .	21
2.12.1	Architektur . . . . .	21
2.12.2	Convolutional Layer . . . . .	22
2.12.3	Pooling Layer . . . . .	24
2.12.4	Vollständig verbundene Ebenen . . . . .	25
	<b>Abkürzungsverzeichnis</b>	<b>vii</b>
	<b>Tabellenverzeichnis</b>	<b>ix</b>

<b>Abbildungsverzeichnis</b>	<b>xi</b>
<b>Quellcodeverzeichnis</b>	<b>xiii</b>
<b>Literatur</b>	<b>xv</b>



## **Kapitel 1**

### **Einleitung**



## Kapitel 2

# Deep Learning

### 2.1 Einleitung

Künstliche neuronale Netze stellen eine Klasse von Modellen des maschinellen Lernens dar, die vom Zentralnervensystem von Säugetieren inspiriert sind. Jedes Netz besteht aus mehreren miteinander verbundenen „Neuronen“, die in „Schichten“ organisiert sind. Neuronen in einer Schicht leiten Nachrichten an Neuronen in der nächsten Schicht weiter (sie „feuern“ im Jargon). Erste Studien wurden in den frühen 50er Jahren mit der Einführung des „Perzeptrons“ [1] begonnen, eines zweischichtigen Netzwerks, das für einfache Operationen verwendet wird, und in den späten 60er Jahren mit der Einführung des „Back-Propagation-Algorithmus“ (effizientes mehrschichtiges Netzwerktraining) (gemäß [2], [3]) weiter ausgebaut. Einige Studien argumentieren, dass diese Techniken Wurzeln haben, die weiter zurückreichen als normalerweise zitiert [4].

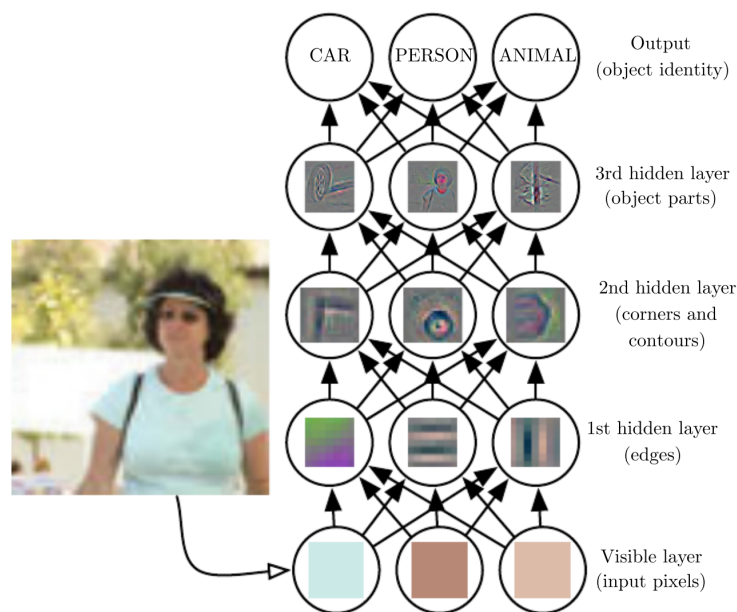
Neuronale Netze waren bis in die 80er Jahre ein Thema intensiver akademischer Studien. Zu diesem Zeitpunkt wurden andere, einfachere Ansätze relevanter. Ab Mitte der 2000er Jahre ist das Interesse jedoch wieder gestiegen, hauptsächlich aufgrund von drei Faktoren: einem von G. Hinton [3], [5] vorgeschlagenen bahnbrechenden Algorithmus für schnelles Lernen, die Einführung von GPUs um 2011 (für massive numerische Berechnungen) und die Verfügbarkeit großer Datenmengen.

Diese Verbesserungen eröffneten den Weg für modernes „Deep Learning“, eine Klasse neuronaler Netze, die durch eine erhebliche Anzahl von Neuronenschichten gekennzeichnet ist, die in der Lage sind, auf der Grundlage progressiver Abstraktionsebenen komplexe Modelle zu erlernen. Sie werden als „tief“ bezeichnet, als es

vor einigen Jahren damit begonnen wurde, 3-5 Schichten zu verwenden. Jetzt sind Netzwerke mit mehr als 200 Schichten vorstellbar.

Das Lernen durch progressive Abstraktion ähnelt Visionsmodellen, die sich über Millionen von Jahren im menschlichen Gehirn entwickelt haben. In der Tat ist das menschliche visuelle System in verschiedene Schichten unterteilt. Erstens sind unsere Augen mit einem Bereich des Gehirns verbunden, der als visueller Kortex (V1) bezeichnet wird und sich im unteren hinteren Teil unseres Gehirns befindet. Dieser Bereich ist vielen Säugetieren gemeinsam und hat die Aufgabe, grundlegende Eigenschaften wie kleine Änderungen der visuellen Ausrichtung, der räumlichen Frequenzen und der Farben zu unterscheiden.

Es wird geschätzt, dass V1 aus etwa 140 Millionen Neuronen besteht, zwischen denen zig Milliarden Verbindungen bestehen. V1 wird dann mit anderen Bereichen (V2, V3, V4, V5 und V6) verbunden, wobei die Bildverarbeitung zunehmend komplexer wird und komplexere Konzepte wie Formen, Gesichter, Tiere und vieles mehr erkannt werden. Es wird geschätzt, dass es 16 Milliarden menschliche kortikale Neuronen gibt und etwa 10-25% des menschlichen Kortexes dem Sehen gewidmet sind [6]. Deep Learning hat sich von dieser schichtbasierten Organisation des menschlichen visuellen Systems inspirieren lassen: Höhere künstliche Neuronenschichten lernen grundlegende Eigenschaften von Objekten, während tiefere Schichten komplexere Konzepte dieser Objekte lernen (siehe Abbildung 2.1).



**Abbildung 2.1:** Illustration eines Deep-Learning-Modells aus [7] und [8]: Ein Computer kann ohne weiteres das Bild in dieser Abbildung nicht erfassen, da es keine sensorischen Rohdaten verstehen kann. Das Bild in dieser Abbildung ist nur eine Sammlung von Pixelwerten. Die Funktionszuordnung von einem Satz von Pixeln zu einer Objektidentität ist sehr kompliziert. Deep Learning löst diese Schwierigkeit, indem das gewünschte komplizierte „Mapping“ in eine Reihe verschachtelter einfacher Mappings aufgeteilt wird, die jeweils durch eine andere Ebene des Modells beschrieben werden. Die Eingabe wird auf der „sichtbaren Ebene“ (visible layer) dargestellt. Diese Schicht wird so genannt, weil sie die Variablen enthält, die wir beobachten können. Dann extrahieren eine Reihe „verborgener Ebenen“ (hidden layer) zunehmend abstrakte Merkmale aus dem Bild. Diese Ebenen werden als „verborgen“ bezeichnet, da ihre Werte nicht in den Daten angegeben sind. Stattdessen muss das Modell bestimmen, welche Konzepte zur Erklärung der Beziehungen in den beobachteten Daten nützlich sind. Angesichts der Pixel kann die erste Schicht Kanten nur leicht identifizieren, indem die Helligkeit benachbarter Pixel vergleicht. Angesichts der Beschreibung der Kanten durch die erste verborgene Ebene kann die zweite verborgene Ebene nach Ecken und erweiterten Konturen suchen. Angesichts der Beschreibung des Bildes durch die zweite verborgene Ebene in Bezug auf Ecken und Konturen kann die dritte verborgene Ebene ganze Teile bestimmter Objekte erkennen, indem bestimmte Konturen und Ecken gefunden werden. Schließlich kann diese Beschreibung des Bildes in Bezug auf die darin enthaltenen Objektteile verwendet werden, um die im Bild vorhandenen Objekte zu erkennen.

## 2.2 Das Perzeptron

Das Perzeptron kann ins deutsche mit dem Begriff der „Wahrnehmung“ übersetzt werden. Das Perzeptron ist ein einfacher Algorithmus mit einem Eingabevektor  $x$  mit  $m$  Werten  $(x_1, \dots, x_m)$ . Es wird oft als „Eingabe-Features“ oder einfach als „Features“ bezeichnet und zurückgegeben wird entweder eine 1 „Ja“ oder eine 0 „Nein“ (siehe Formel 2.1).

In Formel 2.1 ist  $w$  ein Vektor welches das Gewicht darstellt, und  $wx$  das Punktprodukt aus  $\sum_{j=1}^m w_j x_j$ ,  $b$  ist der Bias. Aus  $wx + b$  ist die Grenzhyperebene definiert, die die Position gemäß den  $w$  und  $b$  zugewiesenen Werten ändert.

$$fx = \begin{cases} 1 & wx + b > 0 \\ 0 & \text{ansonsten} \end{cases} \quad (2.1)$$

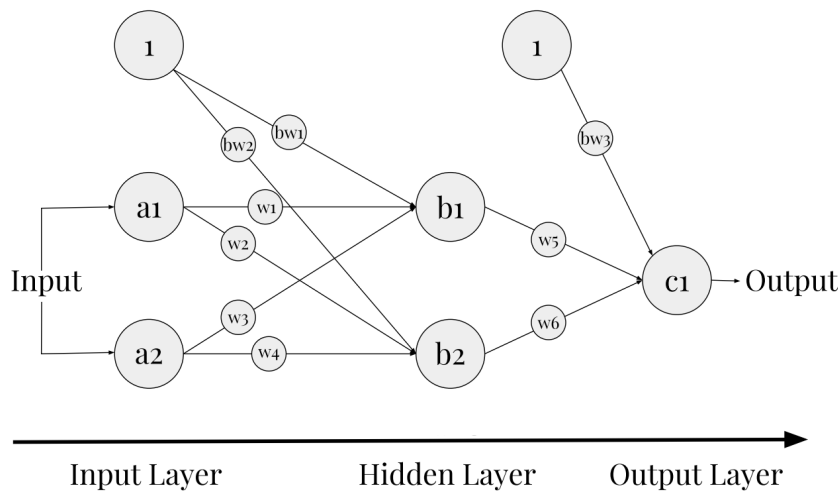
Mit anderen Worten, ist dies ein sehr einfacher, aber effektiver Algorithmus. Beispielsweise kann das Perzeptron bei drei Eingabemerkmalen (Rot, Grün und Blau) unterscheiden, ob die Farbe weiß ist oder nicht. Es soll beachtet werden, dass das Perzeptron keine „Vielleicht“-Antwort ausdrücken kann. Es kann mit „Ja“ (1) oder „Nein“ (0) antworten. Das Perzeptron-Modell kann also benutzt werden, indem durch Anpassung von  $w$  und  $b$ , das Modell „trainiert“ wird.

### 2.3 Bias

Der Bias kann als Maß dafür vorgestellt werden, wie einfach es ist, das Perzeptron dazu zu bringen, um eine 1 auszugeben. Um es biologischer auszudrücken, ist der Bias ein Maß dafür, wie einfach es ist, das Perzeptron zum Feuern zu bringen. Für ein Perzeptron mit einem wirklich großen Bias ist es für das Perzeptron extrem einfach, eine 1 auszugeben. Wenn der Bias jedoch sehr negativ ist, ist es für das Perzeptron schwierig, eine 1 auszugeben [9, S. 7].

### 2.4 Mehrschichtiges Perzeptron

In der Vergangenheit war „Perzeptron“ der Name eines Modells mit einer einzigen linearen Schicht. Wenn es mehrere Schichten hat, wurde es daher als mehrschichtiges Perzeptron (Multi-layer perceptron / MLP) bezeichnet. Die Eingabe- und Ausgabebene ist von außen sichtbar, während alle anderen Ebenen in der Mitte ausgeblendet sind - daher der Name ausgeblendete Ebenen (hidden layers). In diesem Zusammenhang ist eine einzelne Schicht einfach eine lineare Funktion, und der MLP wird daher erhalten, indem mehrere einzelne Schichten nacheinander gestapelt werden (siehe Abbildung 2.2).



**Abbildung 2.2:** Ein Beispiel für ein mehrschichtiges Perzeptron in Anlehnung an [10]: Jeder Knoten in der ersten verborgenen Schicht empfängt eine Eingabe und „feuert“ eine 0 oder 1 gemäß den Werten der zugehörigen linearen Funktion. Dann wird die Ausgabe der ersten verborgenen Schicht an die zweite Schicht übergeben, wo eine andere lineare Funktion angewendet wird, deren Ergebnisse an die endgültige Ausgabeschicht übergeben werden. Die letzte Schicht besteht nur aus einem einzelnen Neuron. Es ist interessant festzustellen, dass diese geschichtete Organisation vage der Organisation des menschlichen Sichtsystems ähnelt, wie zuvor besprochen.

Was sind die besten Entscheidungen für das Gewicht  $w$  und den Bias  $b$ ? Um diese Frage zu beantworten, wird nur ein einzelnes Neuron (ein einzelner Knoten) betrachtet.

Im Idealfall werden eine Reihe von Trainingsbeispielen bereitgestellt und der Computer muss das Gewicht  $w$  und den Bias  $b$  so einstellen, dass die in der Ausgabe erzeugten Fehler minimiert werden.

Um dies etwas konkreter zu machen, wird angenommen, dass es eine Reihe von Katzenbildern vorhanden sind und eine weitere separate Reihe von Bildern, die keine Katzen enthalten. Angenommen, jedes Neuron empfängt Eingaben vom Wert eines einzelnen Pixels in den Bildern. Während der Computer diese Bilder verarbeitet, möchten wir, dass unser Neuron seine Gewichte und seine Vorspannung so anpasst, dass immer weniger Bilder falsch erkannt werden. Dieser Ansatz scheint sehr intuitiv zu sein, erfordert jedoch eine kleine Änderung der Gewichte (oder des Bias), um nur eine kleine Änderung der Ausgänge zu bewirken. Wenn wir einen großen Leistungssprung haben, können wir nicht progressiv lernen. Es wird gewünscht, wie ein „Kind“ zu lernen, nach und nach. Das Perzeptron zeigt jedoch

dieses „Stück für Stück“-Verhalten nicht. Ein Perzeptron gibt entweder eine 0 oder eine 1 zurück und das ist ein großer Sprung, der beim Lernen nicht hilft.

### 2.5 Sigmoid-Neuron

Das Verhalten des Perzeptron ist sehr „uneben“, sodass ein „glatteres“ nötig ist. Wir brauchen eine Funktion, die sich ohne Diskontinuität schrittweise von 0 auf 1 ändert. Mathematisch bedeutet dies, dass wir eine stetige Funktion benötigen, mit der wir die Ableitung berechnen können.

Dieses Problem kann überwunden werden, indem einen neuer Typ eines künstlichen Neurons eingeführt wird, der als Sigmoid-Neuron. Sigmoidneuronen ähneln Perzeptronen, sind jedoch so modifiziert, dass kleine Änderungen ihres Gewichts und ihres Bias nur eine geringe Änderung ihrer Leistung bewirken. Dies ist die entscheidende Tatsache, die es einem Netzwerk von Sigmoidneuronen ermöglicht, zu lernen [9, S. 8].

Genau wie ein Perzeptron hat das Sigmoid-Neuron die Eingaben  $x_1, x_2, \dots$ , aber anstatt nur 0 oder 1 zu sein, können diese Eingänge auch beliebige Werte zwischen 0 und 1 annehmen. Also zum Beispiel 0,123 welches eine gültige Eingabe für ein Sigmoid-Neuron ist. Ebenso wie ein Perzeptron hat das Sigmoid-Neuron Gewichte für jede Eingabe,  $w_1, w_2, \dots$  und einen Bias,  $b$ . Die Ausgabe ist jedoch nicht 0 oder 1, stattdessen ist es  $\sigma, (wx + b)$ , wobei  $\sigma$  als Sigmoidfunktion bezeichnet wird und durch Formel 2.2 definiert ist.

$$\sigma(z) = \frac{1}{1 + e^{-z}} \quad (2.2)$$

### 2.6 Aktivierungsfunktionen

Ohne eine Aktivierungsfunktion (auch als Nichtlinearität bezeichnet) würde die dichte Schicht (dense layer) nuraus zwei linearen Operationen bestehen - einem Punktprodukt und einer Addition:  $Ausgabe = Punkt(w, Eingabe) + b$ . Die Schicht konnte also nur lineare Transformationen (affine Transformationen) der Eingabedaten lernen. Um Zugang zu einem viel umfangreicheren Hypothesenraum zu erhalten, wird eine Nichtlinearitäts- oder Aktivierungsfunktion benötigt [11, S. 72]. Es



gibt weitaus mehr Aktivierungsfunktionen, als die in diesem Abschnitt beschrieben. Es sollen hier nur die gängigsten 3 vorgestellt werden.

### 2.6.1 Sigmoid

Die Sigmoidfunktion wurde bereits mit der Formel 2.2 definiert und in der Abbildung 2.3 dargestellt, die Ableitung der Sigmoidfunktion ist in der Formel 2.3. Sie hat kleine Ausgangsänderungen im Bereich  $(0, 1)$ , wenn der Eingang im Bereich  $(-\infty, \infty)$  variiert. Mathematisch ist die Funktion stetig. Ein Neuron kann das Sigmoid zur Berechnung der nichtlinearen Funktion  $\sigma(z = wx + b)$  verwenden. Wenn  $z = wx + b$  sehr groß und positiv ist, dann wird  $e^z \rightarrow 0$  also  $\sigma(z) \rightarrow 1$ , während wenn  $z = wx + b$  sehr groß und negativ ist dann wird  $e^{-z} \rightarrow 0$  also  $\sigma(z) \rightarrow 0$ . Mit anderen Worten, ein Neuron mit Sigmoidaktivierung hat ein ähnliches Verhalten wie das Perzeptron, aber die Änderungen sind allmählich und Ausgabewerte wie 0,54321 oder 0,12345 sind vollkommen legitim. In diesem Sinne kann ein Sigmoid-Neuron „vielleicht“ antworten [12, S. 10].

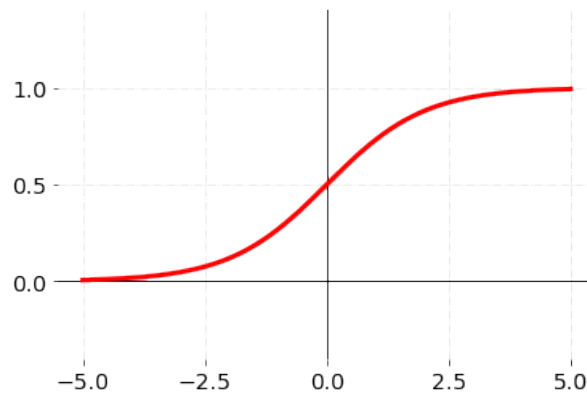


Abbildung 2.3: Darstellung der Sigmoid-Aktivierungsfunktion (eigene Darstellung)

$$\begin{aligned}
 \sigma'(z) &= \frac{d}{dz} \left( \frac{1}{1+e^{-z}} \right) = \frac{1}{(1+e^{-z})^2} \frac{d}{dz} = (e^{-z}) = \frac{e^{-z}}{(1+e^{-z})} \frac{1}{(1+e^{-z})} = \\
 &= \frac{e^{-z}+1-1}{(1+e^{-z})} \frac{1}{(1+e^{-z})} = \left( \frac{(1+e^{-z})}{(1+e^{-z})} - \frac{1}{(1+e^{-z})} \right) \frac{1}{(1+e^{-z})} = \\
 &= \left( 1 - \frac{1}{(1+e^{-z})} \right) \left( \frac{1}{(1+e^{-z})} \right) = (1 - \sigma(z))\sigma(z)
 \end{aligned} \tag{2.3}$$

### 2.6.2 Tanh

Die Tanh-Aktivierungsfunktion wird mit der Formel 2.4 definiert. Sie hat ihre Ausgangsänderungen im Bereich  $(-1, 1)$ . Sie hat eine Struktur, die der Sigmoid-Funktion sehr ähnlich ist. Der Vorteil gegenüber der Sigmoidfunktion besteht darin, dass ihre Ableitung steiler ist, was bedeutet, dass sie mehr Wert erhalten kann (vergleiche Abbildung 2.4). Für die Ableitung der Tanh-Aktivierungsfunktion gilt,  $e^z = \frac{d}{dz}e^z$  und  $e^{-z} = \frac{d}{dz}e^{-z}$ , somit ist die Ableitung dieser Funktion in der Formel 2.5 angegeben.

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \quad (2.4)$$

$$\begin{aligned} \frac{d}{dz}\tanh(x) &= \frac{(e^z + e^{-z})(e^z + e^{-z}) - (e^z - e^{-z})(e^z - e^{-z})}{(e^z + e^{-z})^2} = \\ &= 1 - \frac{(e^z - e^{-z})^2}{(e^z + e^{-z})^2} = 1 - \tanh^2(z) \end{aligned} \quad (2.5)$$

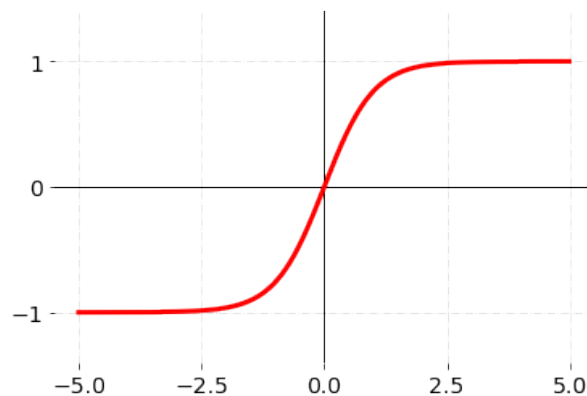


Abbildung 2.4: Darstellung der Tanh-Aktivierungsfunktion (eigene Darstellung)

### 2.6.3 ReLU

Vor kurzem wurde eine sehr einfache Funktion namens ReLU (REctified Linear Unit) sehr beliebt, da sie dazu beiträgt, einige bei Sigmoiden beobachtete Optimierungsprobleme zu lösen [12, S. 11]. Eine ReLU wird relativ einfach in und wird in der Formel 2.6 definiert, die zugehörige Ableitungsfunktion ist in der Formel 2.7. Wie in Abbildung 2.5 zu sehen, ist die Funktion für negative Werte Null und wächst für positive Werte linear. Die ReLU ist relativ einfach zu implementieren (im All-

gemeinen reichen drei Anweisungen aus), während das Sigmoid einige Größenordnungen mehr benötigt.

$$f(x) = \begin{cases} 0 & \text{wenn } x < 0 \\ x & \text{wenn } x \geq 0 \end{cases} \quad (2.6)$$

$$f'(x) = \begin{cases} 1, & \text{wenn } x > 0 \\ 0, & \text{sonst} \end{cases} \quad (2.7)$$

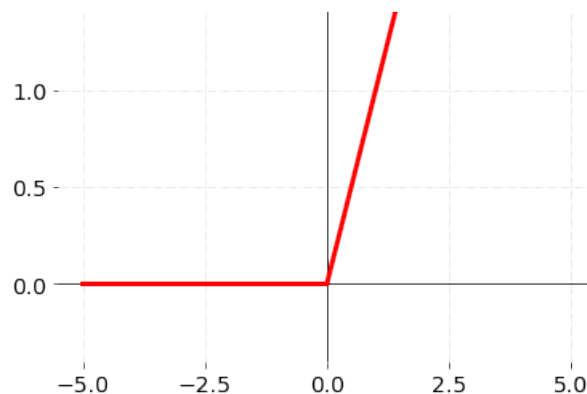
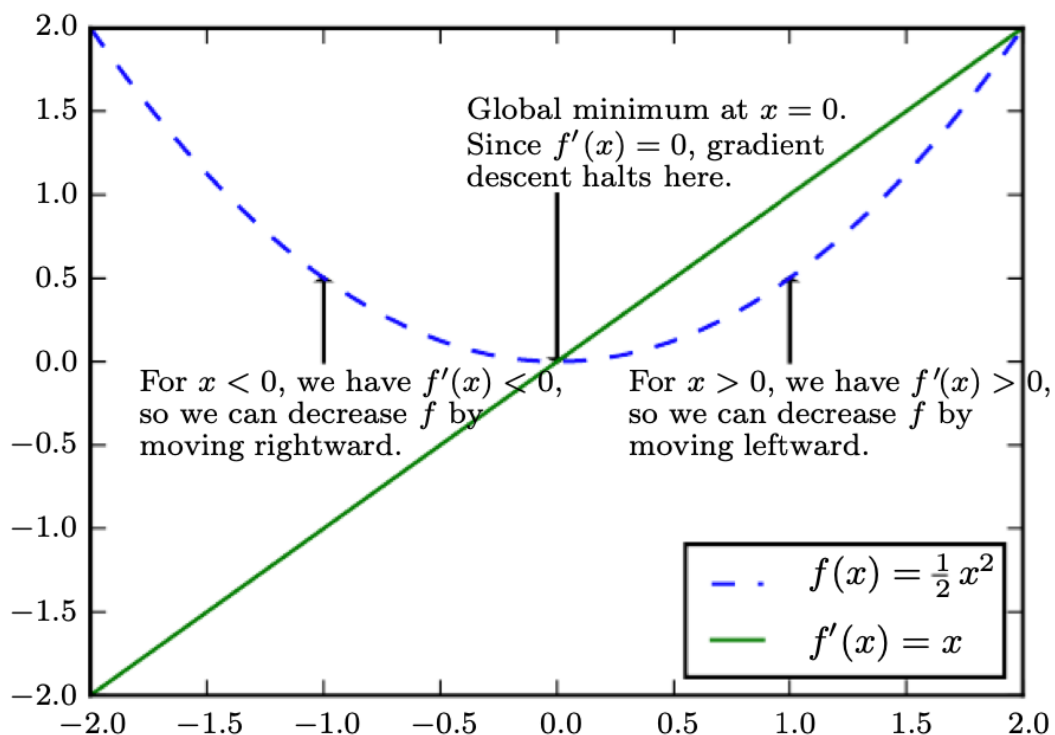


Abbildung 2.5: Darstellung der ReLu-Aktivierungsfunktion (eigene Darstellung)

## 2.7 Optimierungsalgorithmen

Die meisten Deep-Learning-Algorithmen beinhalten irgendeine Art von Optimierung. Optimierung bezieht sich auf die Aufgabe, eine Funktion  $f(x)$  durch Ändern von  $x$  entweder zu minimieren oder zu maximieren. Wir formulieren die meisten Optimierungsprobleme normalerweise in Bezug auf die Minimierung von  $f(x)$ . Die Maximierung kann über einen Minimierungsalgorithmus durch Minimieren von  $-f(x)$  erreicht werden. Die Funktion, die wir minimieren oder maximieren möchten, wird als „objektive Funktion“ oder „Kriterium“ bezeichnet. Wenn wir es minimieren, können wir es auch als Kostenfunktion, Verlustfunktion oder Fehlerfunktion bezeichnen. Die Funktion  $x^* = \arg \min f(x)$  ist eine solche Funktion.



**Abbildung 2.6:** Darstellung des Gradientenabstiegs, entnommen aus [7]. Der Algorithmus nimmt die Ableitung der Funktion bis hin zum Minimum.

Beim Gradientenabstieg können wir uns die Qualität der Vorhersagen unseres Netzwerks als Landschaft vorstellen. Die Hügel stellen Orte (Parameterwerte oder Gewichte) dar, die viele Vorhersagefehler verursachen. Täler repräsentieren Orte mit weniger Fehlern. Wir wählen einen Punkt in dieser Landschaft, an dem wir unser anfängliches Gewicht platzieren möchten. Wir können dann das Anfangsgewicht basierend auf dem Domänenwissen auswählen (wenn wir ein Netzwerk trainieren, um eine Blumenart zu klassifizieren, wissen wir, dass die Blütenblattlänge wichtig ist, die Farbe jedoch nicht). Wenn wir das Netzwerk die ganze Arbeit machen lassen, wählen wir die Anfangsgewichte möglicherweise zufällig aus. Der Zweck besteht darin, dieses Gewicht so schnell wie möglich bergab in Bereiche mit geringerem Fehler zu bewegen. Ein Optimierungsalgorithmus wie der Gradientenabstieg kann die tatsächliche Neigung der Hügel in Bezug auf jedes Gewicht erfassen. Das heißt, es weiß, in welche Richtung es geht. Der Gefälle-Abstieg misst die Steigung (die durch eine Gewichtsänderung verursachte Fehleränderung) und nimmt das Gewicht einen Schritt in Richtung Talboden [13, S. 34] (siehe Abbildung 2.6).

Es gibt drei Varianten des Gradientenabfalls, die sich darin unterscheiden, wie viele Daten wir zur Berechnung des Gradienten der Zielfunktion verwenden. Abhängig von der Datenmenge machen wir einen Kompromiss zwischen der Genauigkeit der Parameteraktualisierung und der Zeit, die für die Durchführung einer Aktualisierung benötigt wird [14]. In den nächsten beiden Abschnitten werden zwei Varianten vorgestellt.

### 2.7.1 Batch Gradientenabstiegsverfahren

Der Standard Gradientenabstiegsverfahren, auch Batch-Gradientenabstiegsverfahren genannt, berechnet den Gradienten der Kostenfunktion zu den Parametern  $\theta$  für den gesamten Trainingsdatensatz.

$$\theta = \theta - \eta \cdot \nabla_{\theta} J(\theta) \quad (2.8)$$

Da der gesamte Datensatz berechnet werden muss, um nur eine Aktualisierung durchzuführen, kann der Batch-Gradientenabstieg sehr langsam sein und ist für Datensätze, die nicht in den Speicher passen, nicht zu handhaben. Der Batch-Gradientenabstieg ermöglicht es auch nicht, das Modell mit neuen Beispielen im laufenden Betrieb zu aktualisieren [14].

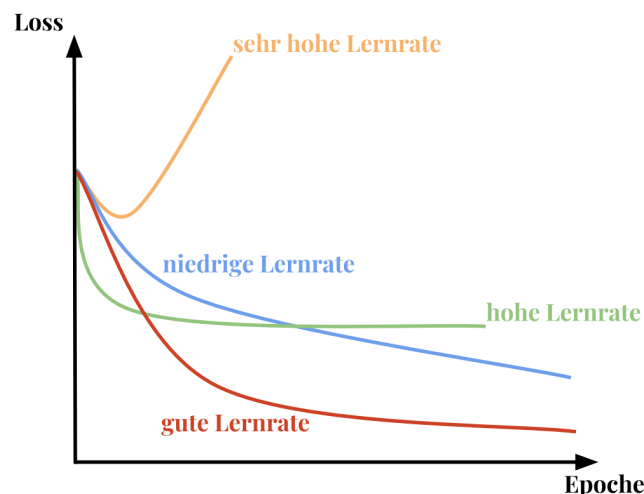
### 2.7.2 Stochastische Gradientenabstiegsverfahren

Im Gegensatz dazu führt der stochastische Gradientenabstiegsverfahren (SGD) eine Parameteraktualisierung für jedes Trainingsbeispiel durch. Der Batch-Gradientenabstieg führt redundante Berechnungen für große Datenmengen durch, da Gradienten für ähnliche Beispiele vor jeder Parameteraktualisierung neu berechnet werden. Der Stochastische Gradientenabstiegsverfahren beseitigt diese Redundanz, indem jeweils ein Update durchgeführt wird. Es ist daher in der Regel viel schneller und kann auch beim laufenden Lernen verwendet werden [14].

$$\theta = \theta - \eta \cdot \nabla_{\theta} J(\theta; x^{(i)}; y^{(i)}) \quad (2.9)$$

## 2.8 Lernrate im Deep Learning

Die Lernrate beeinflusst den Betrag, um den die Parameter während der Optimierung angepasst werden, um den Fehler des neuronalen Netzwerks zu minimieren. Es ist ein Koeffizient, der die Größe der Schritte (Aktualisierungen) skaliert, die ein neuronales Netzwerk auf seinen Parameter (Vektor  $x$ ) ausführt, wenn es den Verlustfunktionsraum durchquert. Ein großer Lernratenkoeffizient (z. B. 1) lässt die Parameter Sprünge machen, und kleine (z. B. 0,00001) lassen ihn langsam voranschreiten. Im Gegensatz dazu sollten kleine Lernraten letztendlich zu einem Fehlerminimum führen (es kann eher ein lokales als ein globales Minimum sein), aber sie können sehr lange dauern und die Belastung eines bereits rechenintensiven Prozesses erhöhen [13, S. 77].

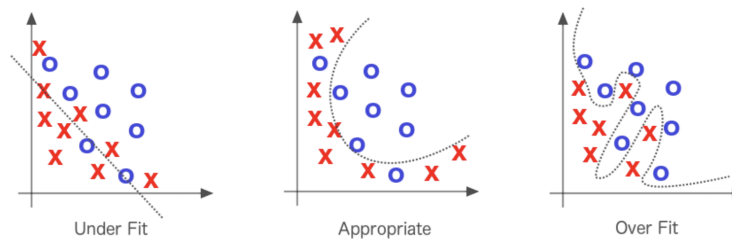


**Abbildung 2.7:** Vergleich unterschiedlicher Lernraten und deren Effekt auf den Verlust. Bei niedrigen Lernraten ist eine „lineare“ Verbesserungen zu sehen. Mit hohen Lernraten werden sie exponentieller. Höhere Lernraten verringern den Verlust schneller, bleiben jedoch bei schlechteren Verlustwerten hängen (grüne Linie). Dies liegt daran, dass die Optimierung zu viel „Energie“ enthält und die Parameter „chaotisch herumspringen“ und sich nicht an einem Ort in der Optimierungslandschaft niederlassen können (in Anlehnung an [15]).

## 2.9 Unteranpassung und Überanpassung

Optimierungsalgorithmen versuchen zunächst, das Problem der Unteranpassung „Underfitting“ zu lösen. Das heißt, eine Linie zu nehmen, die sich den Daten nicht gut annähert, und sie besser an die Daten heranzuführen. Eine gerade Linie, die über ein gekrümmtes Streudiagramm schneidet, wäre ein gutes Beispiel für eine Unter-

anpassung, wie in Abbildung 2.8 dargestellt. Wenn die Linie zu gut zu den Daten passt, haben wir das gegenteilige Problem, das als Überanpassung „Overfitting“ bezeichnet wird [13, S. 27].



**Abbildung 2.8:** Die Abbildung vergleicht die Überanpassung mit der Unteranpassung. Die einzelnen Punkte passen sich dem trainierten Modell zu sehr an, der Verlust wird also so klein, dass das Modell nicht mehr zuverlässige Ergebnisse liefern kann. Dies ist darauf zurückzuführen, dass das Modell „zu viel“ aus dem Trainingsdatensatz gelernt hat. Unteranpassung ist der Fall, wenn das Modell aus den Trainingsdaten „nicht genug gelernt“ hat, was zu einer geringen Verallgemeinerung und unzuverlässigen Vorhersagen führt (Grafik entnommen aus [13, S. 27]).

## 2.10 Regularisierung

Die Regularisierung hilft bei den Auswirkungen von außer Kontrolle geratenen Parametern, indem verschiedene Methoden verwendet werden, um die Parametergröße im Laufe der Zeit zu minimieren. Der Hauptzweck der Regularisierung besteht darin, die Überanpassung zu kontrollieren [13, S. 79].

Ein zentrales Problem beim maschinellen Lernen besteht darin, einen Algorithmus zu erstellen, der nicht nur bei den Trainingsdaten, sondern auch bei neuen Eingaben eine gute Leistung erbringt. Viele beim maschinellen Lernen verwendete Strategien sind explizit darauf ausgelegt, den Testfehler zu reduzieren, möglicherweise auf Kosten eines erhöhten Trainingsfehlers. Diese Strategien werden zusammen als Regularisierung bezeichnet. Tatsächlich war die Entwicklung effektiverer Regularisierungsstrategien eine der wichtigsten Forschungsanstrengungen auf diesem Gebiet, in den folgenden Abschnitten sollen einige Strategien dargestellt werden. Regularisierung kann schließlich definiert werden als „jede Änderung, die wir an einem Lernalgorithmus vornehmen, um dessen Generalisierungsfehler, aber nicht seinen Trainingsfehler zu reduzieren“ [7, S. 228].

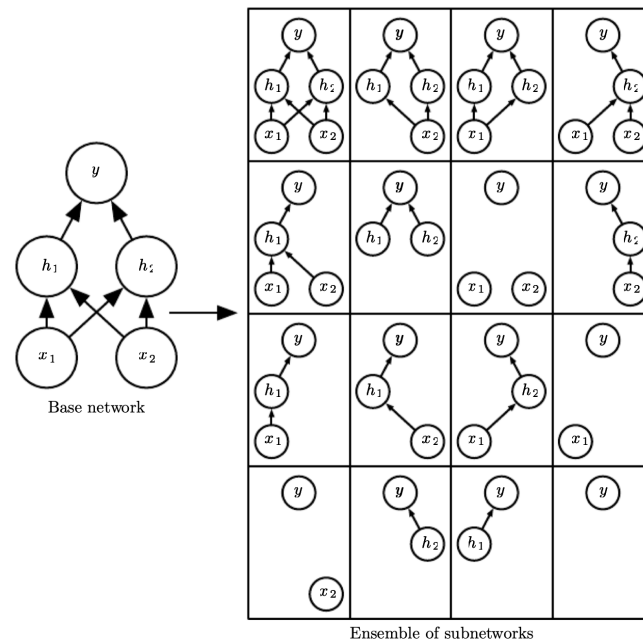
### 2.10.1 Early Stopping

Wenn große Modelle mit trainiert werden, um eine bestimmte Aufgabe lösen, wird häufig festgestellt, dass der Trainingsfehler mit der Zeit stetig abnimmt, der Fehler des Validierungssatzes jedoch wieder zunimmt. Dies bedeutet, dass ein Modell mit einem besseren Validierungssatzfehler (und damit einem besseren Testsatzfehler) erhalten werden kann, indem zu dem Zeitpunkt mit dem niedrigsten Validierungssatzfehler zur Parametereinstellung zurückgekehrt wird. Jedes Mal, wenn sich der Fehler im Validierungssatz verbessert, wird eine Kopie der Modellparameter gespeichert. Wenn der Trainingsalgorithmus beendet wird, wird diese Parameter anstelle der neuesten Parameter zurückgegeben. Diese Strategie wird als Early Stopping „frühes Stoppen“ bezeichnet. Es ist wahrscheinlich die am häufigsten verwendete Form der Regularisierung. Seine Popularität ist sowohl auf seine Wirksamkeit als auch auf seine Einfachheit zurückzuführen [7, S. 246].

### 2.10.2 Dropout

Eine weitere Strategie um Überanpassung zu vermeiden wird in [16] dargestellt. Dropout bietet eine rechnerisch kostengünstige, aber leistungsstarke Methode zur Regularisierung dar. Es ist das aufteilen von mehreren Ensembles großer Netzwerke. Es wird also ein Ensemble, welches aus mehreren Subnetzwerken (sub-networks) gebildet wird [7, S. 258].





**Abbildung 2.9:** Dropout trainiert ein Ensemble. Ein Ensemble besteht aus allen Teilnetzwerken. Es wird durch das entfernen von Einheiten aufgebaut. Das Ensemble besteht aus 16 Teilmengen, aus den vier Einheiten des Basis-Netzwerks. Die 16 Subnetze werden durch das Löschen verschiedener Teilmengen von Einheiten aus dem ursprünglichen Netzwerk gebildet (entnommen aus [7, S. 260]).

### 2.10.3 L1 und L2

Eine weitere Strategie der Regularisierung ist die Modifikation der L1 und L2 Gewichte. Manchmal müssen wir die Größe eines Vektors messen. Beim Deep Learning wird die Größe von Vektoren mit einer Funktion, die als „Norm“ bezeichnet wird [7, S. 39]. Formal ist die  $L^p$  gegeben als:

$$\|x\| = \left( \sum_i |x_i|^p \right)^{\frac{1}{p}}. \quad (2.10)$$

Normen, einschließlich der  $L^p$ -Norm, sind Funktionen, die Vektoren auf nicht negative Werte abbilden. Auf einer intuitiven Ebene misst die Norm eines Vektors  $x$  den Abstand vom Ursprung zum Punkt  $x$ . Die  $L^2$ -Norm mit  $p = 2$  ist als euklidische Norm bekannt. Es ist der euklidische Abstand vom Ursprung zum Punkt  $x$ . Die  $L^2$ -Norm wird beim maschinellen Lernen häufig verwendet, sie wird einfach als  $\|x\|$  bezeichnet, wobei der Index 2 weggelassen wird [7, S. 39].

Wenn zwischen Elementen zu unterscheiden werden soll, die genau Null sind, und Elementen, die klein, aber ungleich Null sind wird eine Funktion angewendet, die an allen Standorten mit der gleichen Geschwindigkeit wächst, aber die mathematische Einfachheit beibehält: die L1-Norm. Die  $L^1$ -Norm kann vereinfacht werden als:

$$\|x\|_1 = \sum_i |x_i|. \quad (2.11)$$

Die  $L^2$ -Norm wird üblicherweise beim maschinellen Lernen verwendet, wenn der Unterschied zwischen Null- und Nicht-Null-Elementen sehr wichtig ist [7, S. 40].

## 2.11 Verlustfunktion und Kreuzentropie

### 2.11.1 Verlustfunktionen

Innerhalb eines neuronalen Netzwerks wandelt eine Verlustfunktion alle möglichen Fehler, in eine Zahl um, die den Gesamtfehler des Netzwerks darstellt. Im Wesentlichen ist es ein Maß dafür, wie falsch ein Netzwerk ist. Auf einer technischeren Ebene werden ein Ereignis oder Werte einer oder mehrerer Variablen einer reellen Zahl zugeordnet. Diese reelle Zahl stellt den „Verlust“ oder die „Kosten“ dar, die mit dem Ereignis oder den Werten verbunden sind [10, S. 61–62].

Das Neuron durch lern dadurch, indem es den Gewicht und den Bias mit einer Rate ändert, die durch die partiellen Ableitungen der Kostenfunktion  $\partial C / \partial w$  und  $\partial C / \partial b$  bestimmt wird. Zu sagen, dass das „Lernen langsam ist“, ist also dasselbe wie zu sagen, dass diese partiellen Ableitungen klein sind. Die Herausforderung besteht darin zu verstehen, warum sie klein sind. Um dies zu verstehen, berechnen wir die partiellen Ableitungen [9, S. 61]. Gegeben sei die quadratische Verlustfunktion:

$$C = \frac{(y - a)^2}{2}, \quad (2.12)$$

dabei ist  $a$  die Ausgabe des Neurons, wenn die Trainingseingabe  $x = 1$  ist, und  $y = 0$  die entsprechende gewünschte Ausgabe. Um dies in Bezug auf Gewicht und

Bias expliziter zu schreiben, sei daran erinnert, dass  $a = \sigma(z)$  ist, wobei  $z = wx + b$  ist. Es ergeben sich durch die Anwendung der Kettenregel folgende Gleichungen:

$$\frac{\partial C}{\partial w} = (a - y)\sigma'(z)x = a\sigma'(z) \quad (2.13)$$

$$\frac{\partial C}{\partial b} = (a - y)\sigma'(z) = a\sigma'(z). \quad (2.14)$$

Aus der Abbildung 2.3 ist die Kurve der Sigmoidfunktion zu sehen, welches. Die Kurve wird sehr flach, wenn der Ausgang des Neurons nahe bei 1 liegt, und daher wird  $\sigma'(z)$  sehr klein. Die Gleichungen 2.13 und 2.14 sagen uns dann, dass  $\partial C/\partial w$  und  $\partial C/\partial b$  sehr klein werden. Dies ist der Grund warum das lernen langsamer wird.

### 2.11.2 Kreuzentropie

Nach [9, S. 62] kann die Lernverlangsamung gelöst werden, indem die quadratischen Verlustfunktion durch eine andere Verlustfunktion ersetzt wird. Diese Funktion wird als Kreuzentropie bezeichnet. Die Abbildung 2.10 zeigt die Kreuzentropie mit mehreren Eingabevariablen und entsprechenden Gewichten und dem Bias. Die Ausgabe des Neurons ist  $a = \sigma(z)$ , wobei  $z = \sum_j w_j b_j + b$  ist, die gewichtete Summe des Inputs. Wir definieren die Kreuzentropiekostenfunktion für dieses Neuron durch:

$$C = -\frac{1}{n} \sum_x [y \ln a + (1 - y) \ln(1 - a)] \quad (2.15)$$

wobei  $n$  die Gesamtzahl der Trainingselemente darstellt. Die Summe gibt die entsprechende gewünschte Ausgabe  $x$  und  $y$  über alle Trainingseingaben an. Zusammenfassend ist die Kreuzentropie positiv und tendiert gegen Null, wenn das Neuron „besser“ wird in der Berechnung der gewünschten Ausgabe  $y$  für alle Trainingseingaben  $x$ . Die entropieübergreifende Kostenfunktion hat jedoch den Vorteil, dass sie im Gegensatz zu den quadratischen Kosten das Problem der Verlangsamung des Lernens vermeidet. Um dies zu sehen, berechnen wir die partielle Ableitung der Kreuzentropiekosten in Bezug auf die Gewichte [9, S. 63].

$$\begin{aligned} \frac{\partial C}{\partial w_j} &= -\frac{1}{n} \sum_x \left( \frac{y}{\sigma(z)} - \frac{1-y}{1-\sigma(z)} \right) \frac{\partial \sigma}{\partial w_j} = \\ &= -\frac{1}{n} \sum_x \left( \frac{y}{\sigma(z)} - \frac{1-y}{1-\sigma(z)} \right) \sigma'(z) x_j \end{aligned} \quad (2.16)$$

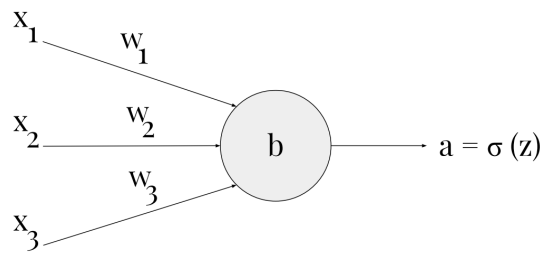
$$\frac{\partial C}{\partial w_j} = \frac{1}{n} \sum_x \frac{\sigma'(z) x_j}{\sigma(z)(1-\sigma(z))} (\sigma(z) - y) \quad (2.17)$$

$$\frac{\partial C}{\partial w_j} = \frac{1}{n} \sum_x x_j (\sigma(z) - y) \quad (2.18)$$

Die Gleichungen 2.11.2, 2.11.2 und 2.11.2 zeigen dass die Geschwindigkeit, mit der das Gewicht  $w$  lernt, durch  $\sigma(z) - y$  gesteuert wird, also durch den Fehler in der Ausgabe. Je größer der Fehler wird, desto schneller lernt das Neuron. Dies ist genau das ein gewünschtes Verhalten. Insbesondere wird die Lernverlangsamung vermieden, die durch den Term  $\sigma'(z)$  in der analogen Gleichung für die quadratischen Kostenfunktion 2.13 verursacht wird. Wenn wir die Kreuzentropie verwenden, wird der Term  $\sigma'(z)$  aufgehoben, und wir brauchen uns keine Sorgen mehr darüber zu machen, dass er klein ist. Diese Aufhebung ist das besondere, das durch die Kreuzentropie-Kostenfunktion gewährleistet wird [9, S. 63–64].

Analog zur Gleichung, und ergibt die Berechnung für den Bias folgende Gleichung:

$$\frac{\partial C}{\partial b} = \frac{1}{n} \sum_x (\sigma(z) - y). \quad (2.19)$$



**Abbildung 2.10:** Das Neuron wird mit 3 Eingabewerten  $(x_1, x_2, x_3)$  den dazugehörigen Gewichten  $(W_1, W_2, W_3)$  trainiert. Der Bias ist durch  $b$  angegeben und die Ausgabe mit  $a = \sigma(z)$  (in Anlehnung an [9])

## 2.12 Convolutional Neural Network

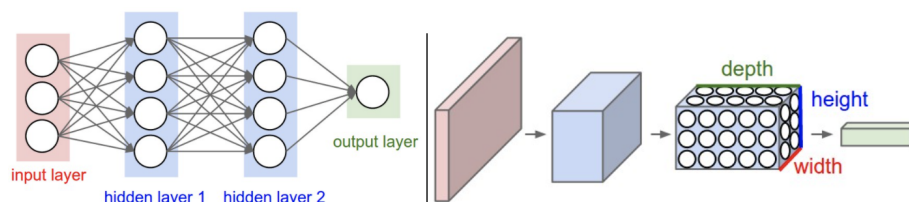
Convolutional Neural Networks (CNNs) arbeiten mit gitterstrukturierten Eingaben, die in lokalen Regionen des Netzes starke räumliche Abhängigkeiten aufweisen. Das offensichtlichste Beispiel für gitterstrukturierte Daten ist ein zweidimensionales Bild. Diese Art von Daten weist auch räumliche Abhängigkeiten auf, da benachbarte räumliche Orte in einem Bild häufig ähnliche Farbwerte der einzelnen Pixel aufweisen. Eine zusätzliche Dimension erfasst die verschiedenen Farben, wodurch ein dreidimensionales Eingabevolumen entsteht. Daher weisen die Merkmale in einem CNN Abhängigkeiten untereinander auf, die auf räumlichen Abständen beruhen. Andere Formen von sequentiellen Daten wie Text, Zeitreihen und Sequenzen können ebenfalls als Sonderfälle von Daten mit Gitterstruktur mit verschiedenen Arten von Beziehungen zwischen benachbarten Elementen betrachtet werden. Die überwiegende Mehrheit der Anwendungen von CNNs konzentriert sich auf Bilddaten, obwohl man diese Netze auch für alle Arten von zeitlichen, räumlichen und raumzeitlichen Daten verwenden kann. Ein wichtiges definierendes Merkmal von CNNs ist eine Operation, die als Faltung (convolution) bezeichnet wird. [17, S. 315–316].

### 2.12.1 Architektur

In CNNs sind mehrere Schichten miteinander verbunden und jeder Schicht hat Gitterstruktur. Die Beziehungen zwischen den Schichten werden von einer Schicht zur nächsten vererbt, da jeder Merkmalswert auf einem kleinen lokalen Bereich aus der

vorherigen Schicht basiert. Es ist wichtig, diese räumlichen Beziehungen zwischen den Gitterzellen aufrechtzuerhalten, da die Faltungsoperation und die Transformation zur nächsten Schicht von diesen Beziehungen abhängen. Jede Schicht im Faltungsnetzwerk ist eine dreidimensionale Gitterstruktur mit einer Höhe, Breite und Tiefe. Die Tiefe einer Schicht in einem Faltungsnetzwerk ist nicht die Tiefe des Netzwerks selbst. Das Wort „Tiefe“ bezieht sich auf die Anzahl der Kanäle in jeder Ebene, z. B. die Anzahl der Primärfarbkanäle (z. B. Blau, Grün und Rot) im Eingabebild [17, S. 318].

Ein ConvNet besteht aus Ebenen. Ein einfaches ConvNet ist eine Folge von Schichten, und jede Schicht eines ConvNet wandelt eine Schicht durch eine differenzierbare Funktion in ein anderes um. Es wird dabei drei Arten von Schichten geben um ein ConvNet Architekturen zu bauen: Faltungs-Schicht , Pooling-Schicht und die vollständig verbundene Schicht [18].

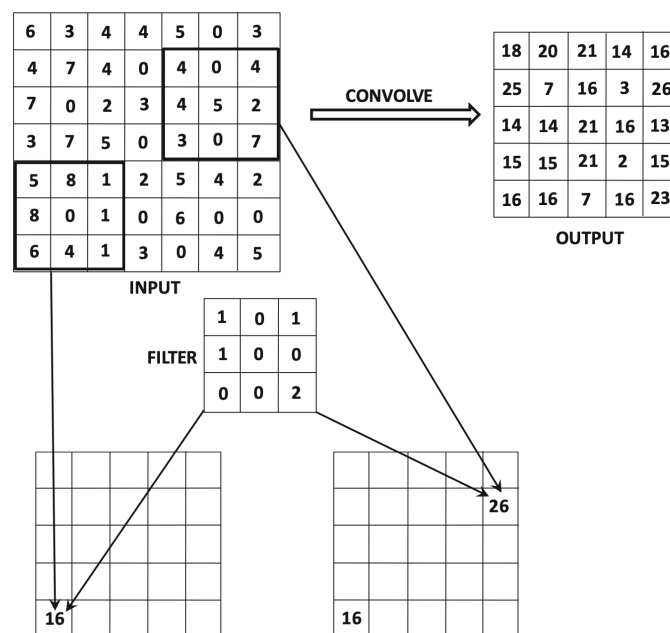


**Abbildung 2.11:** Links: Ein reguläres 3-Schicht-Neuronales Netz. Rechts: Ein ConvNet ordnet seine Neuronen in drei Dimensionen (Breite, Höhe, Tiefe) an, wie in einer der Ebenen dargestellt. Jede Schicht eines ConvNet wandelt das 3D-Eingangsvolumen in ein 3D-Ausgangsvolumen von Neuronenaktivierungen um. In diesem Beispiel enthält die rote Eingabeebene das Bild, sodass seine Breite und Höhe den Abmessungen des Bildes entsprechen. Die „Tiefe“ sind die 3 Farbkanäle (rot, grün, blau) aus [18].

### 2.12.2 Convolutional Layer

Die Parameter der CONV-Schicht bestehen aus einer Reihe von lernbaren Filtern. Jedes dieser Filter ist räumlich klein (Breite und Höhe), erstreckt sich jedoch über die gesamte Tiefe des Eingangsvolumens. Beispielsweise könnte ein typischer Filter auf einer ersten Schicht eines ConvNet die Größe  $5 \times 5 \times 3$  haben (d. H. 5 Pixel Breite und Höhe und 3, weil Bilder die Tiefe 3 haben, die Farbkanäle). Während des Vorwärtsthroughs wird jedes der Filter über die Breite und Höhe des Eingangsvolumens gefaltet und es werden die Punktprodukte zwischen den Einträgen des Filters und dem Eingang an einer beliebigen Position berechnet. Wenn der Filter über die Breite und Höhe des Eingangsvolumens gefaltet wird, wird eine zweidimensionale Aktivierungskarte erstellt, die die Antworten dieses Filters an jeder

räumlichen Position angibt. Intuitiv lernt das Netzwerk Filter, die aktiviert werden, wenn sie eine Art visuelles Merkmal sehen, z. B. eine Kante mit einer bestimmten Ausrichtung oder einen Farbfleck auf der ersten Ebene oder schließlich radähnliche Muster auf höheren Ebenen des Netzwerks. Es entstehen somit ein ganzer Satz von Filtern in jeder CONV-Schicht (z. B. 12 Filter), und jeder von ihnen erzeugt eine separate zweidimensionale Aktivierungskarte. Diese Karten werden entlang der Tiefendimension gestapelt und das Ausgabevolumen erzeugt [18].



**Abbildung 2.12:** Die Faltungsoperation geschieht durch eine Punktprodukt Operation über des Filters, was über alle räumlichen Positionen wiederholt wird „Tiefe“ sind die 3 Farbkanäle (rot, grün, blau) aus [17, S. 321].

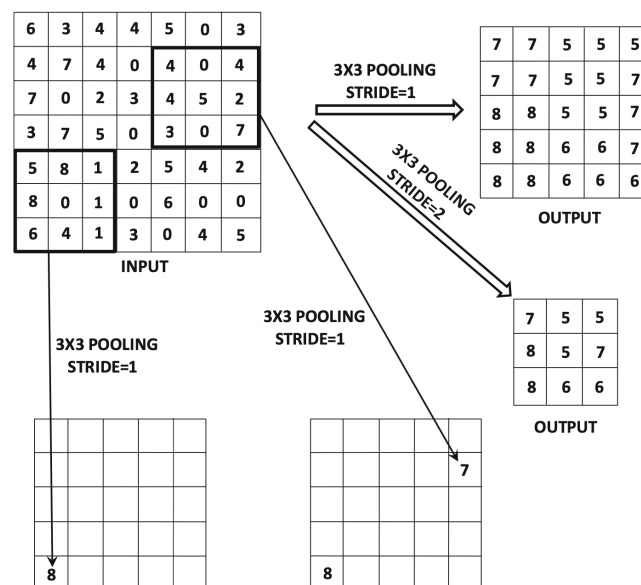
### Padding

Es ist zu beobachten, dass die Faltungsoperation die Größe der  $(q + 1)$ -ten Schicht im Vergleich zur Größe der  $q$ -ten Schicht verringert. Diese Art der Größenreduzierung ist im Allgemeinen nicht wünschenswert, da sie dazu neigt, einige Informationen entlang der Bildränder (oder der Feature-Map im Fall von ausgeblendeten Ebenen) zu verlieren. Dieses Problem kann durch Auffüllen (padding) gelöst werden. Beim Auffüllen werden „Pixel“ rund um die Ränder der Feature-Map hinzugefügt. Der Wert jedes dieser aufgefüllten Feature-Werte wird auf 0 gesetzt, unabhängig davon, ob die Eingabe oder die ausgeblendeten Ebenen aufgefüllt werden. Diese

Bereiche tragen nicht zum endgültigen Punktprodukt bei, da ihre Werte auf 0 gesetzt sind. Ein Teil des Filters aus den Rändern der Schicht wird „herausragt“ und dann durch das Durchführen des Punktprodukts nur über den Teil der Ebene, in dem die Werte definiert sind ersetzt [17, S. 323].

### 2.12.3 Pooling Layer

Es ist üblich, regelmäßig eine Pooling-Ebene zwischen aufeinanderfolgenden Conv-Ebenen in eine ConvNet-Architektur einzufügen. Seine Funktion besteht darin, die räumliche Größe der Darstellung schrittweise zu verringern, um die Anzahl der Parameter und die Berechnung im Netzwerk zu verringern und damit auch die Überanpassung zu steuern. Die Pooling-Ebene arbeitet unabhängig mit jedem Tiefenabschnitt der Eingabe und ändert die Größe räumlich mithilfe der MAX-Operation. Die gebräuchlichste Form ist eine Pooling-Ebene mit Filtern der Größe  $2 \times 2$ , die mit einem Schritt (Stride) von 2 Downsamples pro Tiefenscheibe in der Eingabe um 2 entlang der Breite und Höhe angewendet werden, wobei 75% der Aktivierungen verworfen werden. Jede MAX-Operation würde in diesem Fall maximal 4 Zahlen annehmen. Der „Volumen“ bleibt unverändert [18].



**Abbildung 2.13:** Ein Beispiel für ein Max-Pooling einer Aktivierungskarte der Größe  $7 \times 7$  mit Stride von 1 und 2. Ein Stride von 1 erzeugt eine  $5 \times 5$ -Aktivierungskarte mit stark wiederholten Elementen aufgrund der Maximierung in überlappenden Regionen. Ein Stride von 2 erzeugt eine  $3 \times 3$ -Aktivierungskarte mit weniger Überlappung [17, S. 326].



#### 2.12.4 Vollständig verbundene Ebenen

Diese Ebene wird am Ende des Netzwerkes verwendet, um Klassenwerte zu berechnen, die als Ausgabe des Netzwerks dienen sollen. In einem traditionellen vorwärtsgerichteten neuronalen Netzwerk wird jedes Eingangsneuron mit jedem Ausgangsneuron in der nächsten Schicht verbunden. Dies wird auch als vollständig verbundene Schicht bezeichnet. Der einzige Unterschied zwischen FC- und CONV-Schichten besteht darin, dass die Neuronen in der CONV-Schicht nur mit einer lokalen Region in der Eingabe verbunden sind. Die Neuronen in beiden Schichten berechnen jedoch immer noch Punktprodukte, sodass ihre funktionale Form identisch ist. Daher stellt sich heraus, dass es möglich ist, zwischen FC- und CONV-Schichten zu konvertieren [18].



# **Abkürzungsverzeichnis**



# **Tabellenverzeichnis**



# Abbildungsverzeichnis

2.1	Illustration eines Deep-Learning-Modells . . . . .	5
2.2	Das mehrschichtige Perzeptron . . . . .	7
2.3	Darstellung der Sigmoid-Aktivierungsfunktion . . . . .	9
2.4	Darstellung der Tanh-Aktivierungsfunktion . . . . .	10
2.5	Darstellung der ReLu-Aktivierungsfunktion . . . . .	11
2.6	Der Gradientenabstieg . . . . .	12
2.7	Einfluss der Lernrate auf den Verlust . . . . .	14
2.8	Vergleich der Unteranpassung mit der Überanpassung . . . . .	15
2.9	Basis-Netzwerks und Ensemble im Vergleich . . . . .	17
2.10	Darstellung der Kreuzentropie am Beispiel eines Neurons . . . . .	21
2.11	Vergleich eines NN mit einem CNN . . . . .	22
2.12	Die Faltung in einem CNN . . . . .	23
2.13	Max-Pooling . . . . .	24





## Listings



# Literatur

- [1] F Rosenblatt, „THE PERCEPTRON: A PROBABILISTIC MODEL FOR INFORMATION STORAGE AND ORGANIZATION IN THE BRAIN 1“, Techn. Ber. 6, S. 19–27.
- [2] Paul J. Werbos, „Backpropagation Through Time: What It Does and How to Do It“, *Proceedings of the IEEE*, Jg. 78, Nr. 10, S. 1550–1560, 1990. DOI: 10.1109/5.58337.
- [3] Geoffrey E Hinton und Simon Osindero, „A Fast Learning Algorithm for Deep Belief Nets Yee-Whye Teh“, Techn. Ber.
- [4] Jürgen Schmidhuber, „Deep Learning in Neural Networks: An Overview“, Techn. Ber., 2014. arXiv: 1404.7828v4. Adresse: <http://www.idsia.ch/%CB%9Cjuergen/DeepLearning8Oct2014.tex>CompleteBIBTEXfile.
- [5] David E. Rumelhart, Geoffrey E. Hinton und Ronald J. Williams, „Learning representations by back-propagating errors“, *Nature*, Jg. 323, Nr. 6088, 1986. DOI: 10.1038/323533a0.
- [6] Suzana Herculano-Houzel, *The human brain in numbers: A linearly scaled-up primate brain*, 2009. DOI: 10.3389/neuro.09.031.2009.
- [7] Aaron Courville Ian Goodfellow, Yoshua Bengio, *Deep Learning*. 2016, S. 1–10.
- [8] Faiz Bukhari Mohd Suah, „Preparation and characterization of a novel Co(II) optode based on polymer inclusion membrane“, *Analytical Chemistry Research*, Jg. 12, S. 40–46, 2017. DOI: 10.1016/j.ancr.2017.02.001.
- [9] MA Nielsen, „Neural networks and deep learning“, 2015.
- [10] Michael Taylor, *The Math of Neural Networks*. 2017.
- [11] Francois Chollet, *Deep learning with Python*. 2017. DOI: 10.23919/ICIF.2018.8455530.

- [12] Antonio Guili; Amita Kapoor; Sujit Pal, *Deep Learning with TensorFlow 2 and Keras: Regression, ConvNets, GANs, RNNs, NLP, and More with TensorFlow 2 and the Keras API*. 2019.
- [13] Josh Patterson und Adam Gibson, *Deep Learning a Practitioner's Approach*, 7553. 2019, Bd. 29, S. 1–73.
- [14] Sebastian Ruder, „An overview of gradient descent optimization algorithms“, Sep. 2016. arXiv: 1609.04747. Adresse: <http://arxiv.org/abs/1609.04747>.
- [15] Stanford University Course cs231n, „CS231n Convolutional Neural Networks for Visual Recognition“, *Stanford University Course cs231n*, S. 30, 2018. Adresse: <https://cs231n.github.io/neural-networks-3/%20http://cs231n.github.io/convolutional-networks/%7B%5C%%7D0Ahttp://cs231n.github.io/neural-networks-3/>.
- [16] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky und Ruslan Salakhutdinov, „Dropout: A Simple Way to Prevent Neural Networks from Overfitting“, *Techn. Ber.*, 2014, S. 1929–1958.
- [17] Charu C. Aggarwal, *Neural Networks and Deep Learning*. Springer International Publishing, 2018. DOI: 10.1007/978-3-319-94463-0.
- [18] Stanford University Course cs231n, „CS231n Convolutional Neural Networks for Visual Recognition“, *Stanford University Course cs231n*, S. 30, 2018. Adresse: <https://cs231n.github.io/convolutional-networks/>.