

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

## **Masterarbeit**

vorgelegt am: 21. Januar 2021

Fakultät IV - Institut für Wissensbasierte Systeme und  
Wissensmanagement, Universität Siegen



Eingereicht von: Ugur Tigu  
Matrikelnummer: 1299779  
Studiengang: Wirtschaftsinformatik, Master of Science (M.Sc.)  
Erstprüfer: Prof. Dr.-Ing. Madjid Fathi  
Betreuer: Johannes Zenkert

# Inhaltsverzeichnis

<b>Abbildungsverzeichnis</b>	<b>IV</b>
<b>Tabellenverzeichnis</b>	<b>VI</b>
<b>Abkürzungsverzeichnis</b>	<b>VII</b>
<b>Formelverzeichnis</b>	<b>IX</b>
<b>Listings</b>	<b>X</b>
<b>1 Einleitung</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Forschungsfragen . . . . .	2
1.3 Aufbau der Arbeit . . . . .	2
<b>2 Deep Learning</b>	<b>4</b>
2.1 Einleitung . . . . .	4
2.2 Überwachtes Lernen . . . . .	4
2.3 Das Perzeptron . . . . .	5
2.4 Mehrschichtiges Perzeptron . . . . .	5
2.5 Sigmoid-Neuron . . . . .	6
2.6 Aktivierungsfunktionen . . . . .	7
2.6.1 Sigmoid . . . . .	7
2.6.2 Tanh . . . . .	8
2.6.3 ReLu . . . . .	9
2.6.4 softmax . . . . .	9
2.7 Optimierungsalgorithmen . . . . .	10
2.7.1 Gradientenabstiegsverfahren . . . . .	10
2.7.2 Batch Gradientenabstiegsverfahren . . . . .	11
2.7.3 Stochastische Gradientenabstiegsverfahren . . . . .	11

2.8	Backpropagation . . . . .	12
2.9	Lernrate im Deep Learning . . . . .	13
2.10	Unteranpassung und Überanpassung . . . . .	14
2.11	Regularisierung . . . . .	15
2.11.1	Early Stopping . . . . .	15
2.11.2	Dropout . . . . .	16
2.11.3	L1 und L2 . . . . .	17
2.12	Metriken im Deep Learning . . . . .	17
2.13	Verlustfunktion und Kreuzentropie . . . . .	18
2.13.1	Verlustfunktionen . . . . .	18
2.13.2	Kreuzentropie . . . . .	19
2.14	Convolutional Neural Network . . . . .	20
2.14.1	Architektur . . . . .	21
2.14.2	Convolutional Layer . . . . .	22
2.14.3	Padding . . . . .	23
2.14.4	Pooling . . . . .	23
2.14.5	Vollständig verbundene Ebenen . . . . .	24
<b>3</b>	<b>Die Natürliche Sprache</b>	<b>26</b>
3.1	Tokenisierung durch Reguläre Ausdrücke . . . . .	26
3.2	N-Gramm . . . . .	27
3.3	Part-of-Speech Tagging . . . . .	27
3.4	One-Hot-Encoding . . . . .	28
3.5	Worteinbettungen . . . . .	29
3.6	Word2Vec . . . . .	31
3.7	Word2Vec mit der Skip-Gram Architektur . . . . .	31
3.8	CNN in der Textverarbeitung . . . . .	34
<b>4</b>	<b>Clickbaits und die Klassifizierung von Clickbaits</b>	<b>37</b>
4.1	Was sind Clickbaits? . . . . .	37
4.2	Wie werden Clickbaits klassifiziert? . . . . .	40
<b>5</b>	<b>Konzeption und Methodik</b>	<b>43</b>
<b>6</b>	<b>Erstellung des Datensatzes</b>	<b>46</b>
6.1	Datenbeschaffung durch Web Scraping . . . . .	46
6.2	Rohdatenanalyse . . . . .	48

6.3	Labeln der Daten . . . . .	49
6.4	Analyse der Daten . . . . .	51
<b>7</b>	<b>Das Deep Learning Modell</b>	<b>53</b>
7.1	Analyse der Technologie . . . . .	53
7.2	Vorverarbeitung . . . . .	56
7.3	Das Erstellen eines Vokabulars . . . . .	57
7.4	Die Modell Architektur . . . . .	60
7.5	Das Training . . . . .	63
7.6	Evaluation . . . . .	64
7.7	Experimente . . . . .	66
7.8	Speichern und Konvertieren . . . . .	70
<b>8</b>	<b>Deep Learning Modelle im Web</b>	<b>71</b>
8.1	Serverseitiges Deep Learning . . . . .	71
8.2	Clientseitiges Deep Learning . . . . .	72
8.3	Tokenisierung und Padding . . . . .	73
8.4	Laden des Modells und Vokabulars . . . . .	75
8.5	Die Vorhersage des Modells . . . . .	76
8.6	Vergleich beider Ansätze . . . . .	77
<b>9</b>	<b>Fazit und Ausblick</b>	<b>79</b>
<b>A</b>	<b>Anhang</b>	<b>81</b>
	<b>Literaturverzeichnis</b>	<b>82</b>

# Abbildungsverzeichnis

2.1	Das mehrschichtige Perzeptron . . . . .	6
2.2	Darstellung der Sigmoid-Aktivierungsfunktion . . . . .	7
2.3	Darstellung der Tanh-Aktivierungsfunktion . . . . .	8
2.4	Darstellung der ReLu-Aktivierungsfunktion . . . . .	9
2.5	Der Gradientenabstiegsverfahren . . . . .	11
2.6	Die Vorwärts- und Rückwärtsschritte im Backpropagation . . . . .	13
2.7	Einfluss der Lernrate auf den Verlust . . . . .	14
2.8	Vergleich der Unteranpassung mit der Überanpassung . . . . .	15
2.9	Dropout . . . . .	16
2.10	Darstellung der Kreuzentropie am Beispiel eines Neurons . . . . .	20
2.11	Vergleich eines NN mit einem CNN . . . . .	22
2.12	Die Faltung in einem CNN . . . . .	23
2.13	Max-Pooling . . . . .	24
3.1	One-Hot-Encoding als Eingabematrix . . . . .	29
3.2	Darstellung der Worteinbettungen . . . . .	30
3.3	Worteinbettungen erzeugen Analogien zwischen Wörtern . . . . .	31
3.4	Vergleich zwischen CBOW und Skip-Gram Architektur . . . . .	32
3.5	CNN in der Textverarbeitung . . . . .	35
3.6	1-dimensionale CNN . . . . .	36
4.1	Beispiele von Clickbaits . . . . .	38
4.2	Beispiele von Nicht-Clickbaits . . . . .	38
4.3	Die Kernformen des Clickbaits . . . . .	39
4.4	Clustering von Überschriften mit t-SNE . . . . .	42
5.1	Darstellung der Konzeption . . . . .	43
6.1	Vergleich der Wortlängen der Schlagzeilen aus den Rohdaten . . . . .	49
6.2	Word Cloud Analyse für die Clickbaits Schlagzeilen . . . . .	52

6.3	Word Cloud Analyse für die Wikinews Schlagzeilen . . . . .	52
7.1	Die Darstellung eines Deep Learning Modells in TensorFlow . . . . .	54
7.2	Die Architektur von TensorFlow.js . . . . .	56
7.3	Vergleich der Trainingsdaten mit den Validierungsdaten . . . . .	65
7.4	Auswirkung der Einbettungsdimensionen . . . . .	67
7.5	Auswirkung der Lernrate . . . . .	68
7.6	Auswirkung der zusätzlichen Schichten . . . . .	69
8.1	Das Frontend . . . . .	77

# Tabellenverzeichnis

3.1	Beispiele aus dem Stuttgart-Tübingen-Tagset . . . . .	27
3.2	Vergleich des One-Hot-Encodings mit Worteinbettungen [13, 312] . . . . .	30
6.1	Vergleich der Anzahl und Herkunft Rohdaten nach dem Scrapingvorgang	46
6.2	Vergleich der Ergebnisse des POS-Taggings . . . . .	48
6.3	Beschreibung des gelabelten Datensatzes . . . . .	51
7.1	Beschreibung der Schichten des Modells . . . . .	63
7.2	Die Ergebnisse der Evaluation des Modells . . . . .	64
7.3	Vorhersagen des Modells für neue Schlagzeilen . . . . .	66
7.4	Die Ergebnisse des Modells mit 320 Einbettungsdimensionen . . . . .	67
7.5	Die Ergebnisse der Evaluation des Modells mit einer Lernrate von 0.0006	68
7.6	Das „komplexere“ Modell . . . . .	69
7.7	Die Ergebnisse des Modells mit zusätzlichen Schichten . . . . .	69

# Abkürzungsverzeichnis

**CNN** Convolutional neural network

**NLP** Natural Language Processing

**ReLU** Rectified Linear Units

**ML** Machine Learning

**GPU** Graphics Processing Unit

**LSTM** Long short-term memory

**CBOW** Continuous Bag of Words

**GRU** Gated Recurrent Units

**t-SNE** t-Distributed Stochastic Neighbor Embedding

**API** Application Programming Interface

**SQL** Structured Query Language

**HTTP** Hypertext Transfer Protocol

**UI** User Interface

**CSS** Cascading Style Sheets

**HTML** Hypertext Markup Language

**AJAX** Asynchronous JavaScript and XML

**JSON** JavaScript Object Notation

**POS** Part-of-speech-tagging

**RNN** Rekurrentes neuronales Netz

# Formelverzeichnis

2.1	Funktion des Perzeptron . . . . .	5
2.2	Sigmoidfunktion . . . . .	7
2.3	Ableitung der Sigmoidfunktion . . . . .	8
2.4	Die Tanh-Funktion . . . . .	8
2.5	Die Ableitung der Tanh-Funktion . . . . .	8
2.6	Die ReLu-Funktion . . . . .	9
2.7	Die Ableitung der ReLu-Funktion . . . . .	9
2.8	Die softmax-Funktion . . . . .	10
2.9	Der Gradientenabstiegsverfahren . . . . .	11
2.10	Der stochastische Gradientenabstiegsverfahren . . . . .	12
2.11	Backpropagation . . . . .	12
2.12	Regularisierung . . . . .	17
2.13	L1-Regularisierung . . . . .	17
2.14	Precision . . . . .	17
2.15	Recall . . . . .	18
2.16	F1-Score . . . . .	18
2.17	Kostenfunktion . . . . .	18
2.18	Partielle Kostenfunktion der Gewichte . . . . .	18
2.19	Partielle Kostenfunktion des Bias . . . . .	19
2.20	Kreuzentropie Kostenfunktion . . . . .	19
2.21	Herleitung der Kreuzentropie . . . . .	20
3.1	Wahrscheinlichkeiten bei Skip-Gram . . . . .	33
3.2	Multiplikationen der Skip-Gram . . . . .	33
3.3	Der Lernalgorithmus des Skip-Gram . . . . .	34

# Listings

6.1	Beispiel eines Scrapers . . . . .	47
6.2	Die Funktion labelt nach bestimmten Kriterien den Datensatz . . . . .	50
6.3	Funktionen die die durchschnittliche Wortlänge erkennen und nach Interpunktionszeichen filtern . . . . .	50
6.4	Tagger Funktion . . . . .	51
7.1	Beispiel eines Tensors in TensorFlow . . . . .	53
7.2	Funktion für das Aufteilen des Datensatzes . . . . .	57
7.3	Die Preprocessing-Funktion . . . . .	57
7.4	Die Tokeniser-Funktion . . . . .	58
7.5	Die Label-Funktion . . . . .	59
7.6	Die Dataset Erstellung . . . . .	60
7.7	Das Bilden des Models . . . . .	60
7.8	Das Training des Models . . . . .	63
7.9	Das Evaluieren des Models . . . . .	64
8.1	Beispiel eines Servers für die Vorhersage von Clickbaits . . . . .	71
8.2	Wichtigste Elemente der Benutzeroberfläche . . . . .	73
8.3	Die splitText Funktion . . . . .	73
8.4	Die tokenize Funktion . . . . .	74
8.5	Das useLoadModel Hook . . . . .	75
8.6	Auszug aus der predict Funktion . . . . .	77

# 1 Einleitung

## 1.1 Motivation

Das Aufkommen von Online-Nachrichtenagenturen und die Explosion der Anzahl der Benutzer, die Nachrichten mit diesem Medium konsumieren, haben dazu geführt, dass mehrere Webseiten miteinander konkurrieren, um die Aufmerksamkeit der Benutzer zu erregen. Dies hat dazu geführt, dass die Medien kreative Wege geschaffen haben, um Leser auf ihre Website zu locken. Eine der am häufigsten verwendeten Techniken ist die Verwendung von Clickbait-Überschriften. Diese Überschriften wurden speziell dafür entwickelt, um das Interesse des Lesers an dem zu wecken, was versprochen wird. Wenn auf den Artikel geklickt wird jedoch, liefert dieser Artikel normalerweise nicht den Inhalt, den der Leser ursprünglich gesucht hat.

Deep Learning gewinnt immer mehr Beliebtheit, neben der Wissenschaft, kann Deep Learning auch kommerziell für den Endbenutzer entwickelt werden. Durch *TensorFlow.js*, einer für JavaScript entwickelten Version von der beliebten Deep Learning Bibliothek TensorFlow, ist es möglich Deep Learning Anwendungen „sehr nah“ am Endbenutzer zu entwickeln und somit ohne einen Server Inferenz zu bilden. Diese ist eine relativ neue Art und Weise, Deep Learning Anwendungen zu schreiben. Sie wird in dieser Arbeit als „clientseitiges Deep Learning“ beschrieben. Clientseitiges Deep Learning macht das Deep Learning flexibler, denn es kann im Browser auf dem Server oder auf einem mobilen Endgerät laufen. Es kann schneller sein, da es in sehr kurzer Zeit eine Antwort geben kann, da die Kommunikation mit einem Server entfällt. Der Nutzer muss außerdem seine Daten nicht preisgeben.

Für deutsche Clickbaits gibt es keinen Datensatz. Um eine binäre Klassifikation durchzuführen wird ein Datensatz erstellt. Dieser Datensatz kann in seinem Rohdatenformat falsche Aussagen treffen. Da die Seiten woher die Daten stammen nicht reine Clickbaits anbieten, sondern auch Nachrichten die nicht in diese Kategorie fallen. Für das trennen dieser Daten ist ein händisches Labeln notwendig, was Zeit und Ressourcen in Anspruch nimmt. Dieses Problem kann durch „automatisches Labeln“ behoben werden. Dazu wird es einmal Nötig sein, eine Literaturanalyse zu machen, um zu sehen, wie Clickbaits im

Vorfeld automatisch erkannt und getrennt werden können.

## 1.2 Forschungsfragen

Die Arbeit soll folgende Fragen beantworten:

1. Wie kann ein Datensatz für deutsche Clickbaits erstellt werden?
2. Wie gut können aus den ermittelten Daten ein Modell trainiert werden?
3. Wie können Deep Learning Modelle in eine Web-Anwendung eingebettet werden?

Diese Fragen werden im Kapitel 9 beantwortet.

## 1.3 Aufbau der Arbeit

Die Arbeit beginnt im **Kapitel 2** mit einem theoretischen Einstieg über Deep Learning. Zunächst werden Grundlegende Themen behandelt, die dem Leser einen Einstieg in die praktischen Anwendungen, welche in den späteren Kapiteln erfolgen, näher erläutern sollen. Um den Scope der Arbeit nicht ins unendliche zu ziehen, werden die Algorithmen *Gradientenabstiegsverfahren* und *Backpropagation* nicht sehr tief behandelt. Es werden nur wichtige und relevante Themen hineingezogen. Da später Experimente mit dem Modell gemacht werden, sind die „Hyperparameter“ bei Deep Learning Modellen bedeutsam. Diese Parameter sind z.B. die *Lernrate* und das Konzept der *Über- und Unteranpassung*. Es gilt das Modell soweit es geht zu optimieren, mit bestimmten Maßnahmen wie *Regularisierung*. In diesem Kapitel soll auch gezeigt werden, wie Deep Learning Modelle lernen. Es stellt sich heraus, dass dafür bestimmte Konzepte wie *Aktivierungsfunktionen* oder *Verlustfunktionen* vorgestellt werden müssen. Es gibt immer mehrere Möglichkeiten ein Problem mittels Deep Learning zu lösen. Hier wird der Fokus auf ein Netzwerk, der *Convolutional neural network (CNN)*-Netzwerke vorgestellt. Das CNN-Netzwerk wird in den späteren Kapiteln angewendet, jedoch bedarf es an theoretischer Grundlage.

Im **Kapitel 3** geht es um die Verarbeitung der natürlichen Sprache. Es werden zunächst die standardisierten Verfahren wie *Reguläre Ausdrücke* vorgestellt. Diese können dafür verwendet werden, um ein Text in seine einzelnen Elemente *Tokens* umzuwandeln, da die Tokenisierung ein wichtiger und häufig angewandter Prozess ist. Neben dem Verfahren der Tokenisierung, gibt es andere Verfahren im *Natural Language Processing (NLP)*. Das *Tagging* der Wörter mit der zugehörigen Grammatik wird auch in diesem Kapitel behandelt. Eine Frage die beantwortet werden muss ist, wie Computer die natürliche

Sprache lernen. Hier wird zunächst dargestellt, dass ein Computer nur Zahlen versteht und eine Umwandlung von Wörtern in Zahlen gelingen muss. Sicherlich gibt es dafür mehrere Verfahren, hier wird die Methode der *Worteinbettungen* vorgestellt. Kapitel 3 wird damit abgeschlossen, indem eine Verbindung mit Kapitel 2 hergestellt wird. CNN werden häufig für Bilddateien verwendet. Hier wird gezeigt, wie CNNs auch bei der Textverarbeitung angewendet werden können.

Gemeinsam mit Kapitel 2 und Kapitel 3 wird der theoretische Teil der Arbeit abgeschlossen und im **Kapitel 4** geht es dann um die Literatur. Um Clickbaits zu klassifizieren ist eine Definition von Clickbaits nötig. Dieses ist eine Herausforderung, denn Clickbaits sind sehr subjektiv und können nur sehr schwer definiert werden. Es gibt jedoch einige „Muster“ die aus mehreren Arbeiten zu entnehmen sind, die sich mit diesem Thema beschäftigt haben. Es soll in diesem Kapitel auch erforscht werden, welche Ansätze es gibt, in Bezug auf *Klassifizierung von Clickbaits*. Hier wird gezeigt, wie verwandte Arbeiten das Problem gelöst. Diese Arbeiten werden in den nächsten Kapiteln als Grundlage für weiteres Vorgehen verwendet werden können.

Im **Kapitel 5** wird das Konzept der Arbeit gemeinsam mit der angewendeten Methodik vorgestellt. Dieser Teil der Arbeit soll den praktischen Teil der Arbeit begründen. Es gibt viele Möglichkeiten wie ein Problem gelöst werden kann, hier wird erläutert, was die Gründe waren. Es wird quasi eine „Machbarkeitsanalyse“ vollzogen. Dieses Kapitel soll die nachfolgenden **Kapiteln 6, Kapitel 7 und 8** einleiten. Im Kapitel 6 geht es zuerst darum, wie der Datensatz generiert wurde. Im Kapitel 7 wird das Deep Learning Modell entwickelt, aus den Daten die im Kapitel 6 erstellt wurden. Hier findet neben dem Training auch die Evaluation des Modells statt. Im Kapitel 8 befindet sich der Ansatz, das ganze Modell in eine Webanwendung einzubetten. Im **Kapitel 9** wird schließlich die Arbeit abgeschlossen, indem die Forschungsfragen beantwortet und ein Ausblick über Verbesserungsmöglichkeiten der Arbeit gegeben werden.

# 2 Deep Learning

## 2.1 Einleitung

Künstliche neuronale Netze stellen eine Klasse von Modellen des maschinellen Lernens dar, die vom Zentralnervensystem der Säugetiere 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. Erste Studien wurden in den frühen 50er Jahren mit der Einführung des „Perzeptrons“ [32] begonnen, eines zweischichtigen Netzwerks, welches für einfache Operationen verwendet wurde, und in den späten 60er Jahren mit der Einführung des Backpropagation-Algorithmus [41], [18] weiter ausgebaut wurde.

Deep Learning ermöglicht es einem Computermodell, welches aus mehreren Verarbeitungsebenen bestehen, Darstellungen von Daten mit mehreren Abstraktionsebenen zu erlernen. Diese Verfahren haben den Stand der Technik in Bezug auf Spracherkennung, visuelle Objekterkennung und viele andere Bereiche wie z.B. die Genomik dramatisch verbessert. Deep Learning entdeckt komplexe Strukturen in großen Datenmengen, indem es den Backpropagation-Algorithmus verwendet. Dieses ist ein Verfahren, welches der Maschine zeigt, wie sie ihre internen Parameter ändern soll. Diese Parameter werden mit jeder „Schicht“ neu berechnet, aus den Ergebnissen der vorherigen Schicht. Tiefe Faltungsnetzwerke (CNN) haben Durchbrüche bei der Verarbeitung von Bildern, Video, Sprache und Audio gemacht, während wiederkehrende Netze (Rekurrentes neuronales Netz (RNN), Long short-term memory (LSTM)) sequentielle Daten wie Text und Sprache beleuchtet haben [22].

## 2.2 Überwachtes Lernen

Die häufigste Form des maschinellen Lernens, ob tief oder nicht, ist das überwachte Lernen. Beim überwachten Lernen werden zunächst große Mengen an Daten gesammelt, die jeweils mit ihrer Kategorie gekennzeichnet (gelabelt) sind. Während des Trainings wird der Maschine ein Exemplar aus den Daten gezeigt und es wird eine Ausgabe in Form

eines Bewertungsvektors erzeugt, einer für jede Kategorie. Der Wunsch ist es, für jede Kategorie die höchste Punktzahl zu erreichen. Es wird eine Zielfunktion (Optimierungsfunktion) berechnet, die den Fehler (oder Abstand) zwischen den Ausgabewerten und den gewünschten Bewertungsmustern misst. Die Maschine ändert dann ihre internen einstellbaren Parameter um, um den Fehler zu minimieren. Diese einstellbaren Parameter werden *Gewichte* genannt. Gewichte sind reelle Zahlen die als „Knöpfe“ gesehen werden können, die die Eingabe-Ausgabe-Funktion der Maschine ist. In einem typischen Deep-Learning-System können Hunderte Millionen dieser einstellbaren Gewichte vorkommen [22]. Neben den Gewichten taucht der Begriff *Bias* auf, welches im Deep Learning ein Maß dafür angesehen wird, das Perzeptron zum „feuern zu bringen“ [27, 7].

## 2.3 Das Perzeptron

Das *Perzeptron* ist ein einfacher Algorithmus mit einem Eingabevektor  $x$  mit  $m$  Werten ( $x_2, \dots, x_m$ ). Es wird oft als „Eingabe-Features“ oder einfach als „Features“ bezeichnet. Es gibt entweder eine 1 „Ja“ oder eine 0 „Nein“ zurück (siehe Formel 2.1). Dabei 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.

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

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 dafür verwendet werden, indem durch Anpassung von  $w$  und  $b$ , das Modell trainiert wird.

## 2.4 Mehrschichtiges Perzeptron

Wenn ein Modell nicht nur eine einzige lineare Schicht hat, sondern wenn mehrere Schichten in Form von Perzeptronen zusammengebracht werden, handelt es sich dabei um ein mehrschichtiges Perzeptron. Die Eingabe- und Ausgabeebene ist von außen sichtbar, während alle anderen Ebenen in der Mitte ausgeblendet oder verborgen sind („hidden layers“). Es werden mehrere lineare Funktionen (einzelne Schichten) nacheinander

gestapelt und somit ein mehrschichtiges Perzeptron erzeugt.

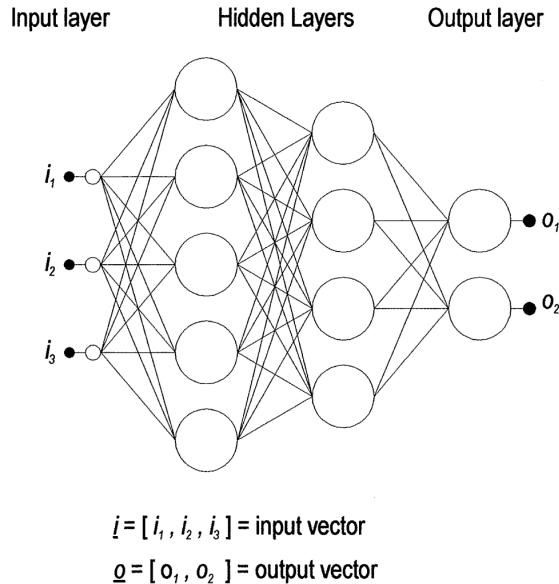


Abbildung 2.1: Das mehrschichtige Perzeptron mit zwei verborgenen Schichten, entnommen aus [17]

## 2.5 Sigmoid-Neuron

Neben dem Perzeptron gibt es das *Sigmoid* Neuron. Das Perzeptron gibt bekanntlich nur eine 0 oder eine 1 zurück. Es ist also eine Funktion nötig, die sich ohne „Diskontinuität“, schrittweise von 0 auf 1 ändert. Mathematisch bedeutet dies, dass eine stetige Funktion nötig ist, mit der die Ableitung berechnet werden kann. Dieses Problem kann überwunden werden, indem einen neuer Typ eines künstlichen Neurons eingeführt wird, der als *Sigmoid-Neuron* bezeichnet wird. Sigmoid Neuronen ähneln Perzeptronen, sind jedoch so modifiziert, dass kleine Änderungen ihres Gewichts und ihres Bias nur eine geringe Änderung ihrer Leistung bewirken. Dies ist der Grund dafür, dass Netzwerke lernen können. [27, 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“) nur aus zwei linearen Operationen bestehen - einem Punktprodukt und einer Addition:  $Ausgabe = Punkt(w, Eingabe) + b$ . Die Schicht könnte also nur lineare Transformationen, „affine Transformationen“ der Eingabedaten lernen. Um Zugang zu einem viel umfangreicheren Hypothesenraum zu erhalten, wird eine Nichtlinearitäts- oder Aktivierungsfunktion benötigt [15, S. 72].

### 2.6.1 Sigmoid

Die Sigmoidfunktion wird mit der Formel 2.2 definiert und in der Abbildung 2.2 dargestellt, die Ableitung der Sigmoidfunktion wird in der Formel 2.6.1 definiert. 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 wird, dann wird  $e^z \rightarrow 0$  also  $\sigma(z) \rightarrow 1$ , während wenn  $z = wx + b$  sehr groß und negativ wird, wird  $e^{-z} \rightarrow 0$  also  $\sigma(z) \rightarrow 0$ . Mit anderen Worten, ein Neuron mit Sigmoidaktivierung hat ein ähnliches Verhalten wie das Perzepron, aber die Änderungen sind allmählich und Ausgabewerte wie 0,54321 oder 0,12345 sind vollkommen legitim. In diesem Sinne kann ein Sigmoid-Neuron auch mit „vielleicht“ antworten [8, 10].

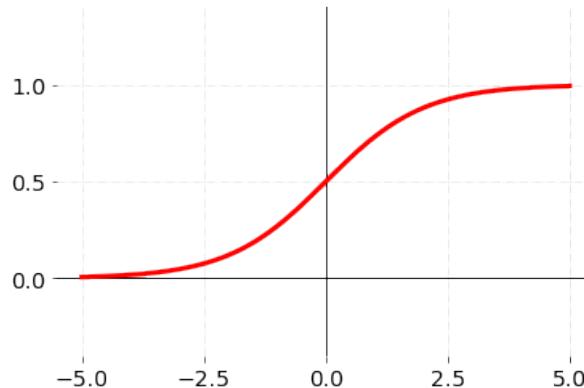


Abbildung 2.2: Darstellung der Sigmoid-Aktivierungsfunktion (eigene Darstellung)

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

## 2.6.2 Tanh

Die Tanh-Aktivierungsfunktion wird mit der Formel 2.4 definiert ihre Ableitung wird in der Formel 2.5 berechnet. 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 Werte enthalten kann (vergleiche Abbildung 2.3). Für die Ableitung der Tanh-Aktivierungsfunktion gilt,  $e^z = \frac{d}{dz} e^z$  und  $e^{-z} = \frac{d}{dz} e^{-z}$ .

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

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

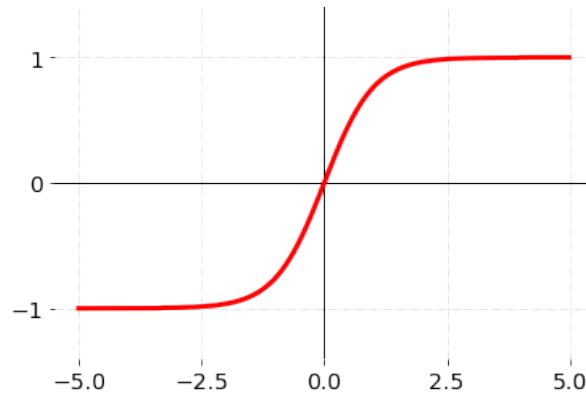


Abbildung 2.3: Darstellung der Tanh-Aktivierungsfunktion (eigene Darstellung)

### 2.6.3 ReLu

Die Rectified Linear Units (ReLU) ist eine relativ einfache Funktion, sie ist relativ neu und wird sehr häufig verwendet [8, 11]. Sie wird in Formel 2.6 definiert, die zugehörige Ableitungsfunktion wird in der Formel 2.7 berechnet. Wie in Abbildung 2.4 zu sehen, ist die Funktion für negative Werte Null und wächst für positive Werte linear.

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

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

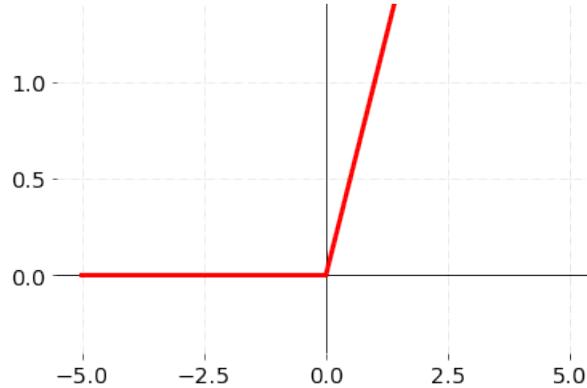


Abbildung 2.4: Darstellung der ReLu-Aktivierungsfunktion (eigene Darstellung)

### 2.6.4 softmax

Die grundlegende Schwierigkeit bei der Durchführung kontinuierlicher Mathematik auf einem digitalen Computer besteht darin, dass unendlich viele reelle Zahlen mit einer endlichen Anzahl von Bitmustern dargestellt werden müssen. Dabei entstehen Rundungsfehler die problematisch sein können, insbesondere wenn sie sich über viele Operationen hinweg zusammensetzen. Sie führen dazu, dass theoretisch funktionierende Algorithmen in der Praxis fehlschlagen. Ein Beispiel für eine Funktion, die gegen Rundungsfehler stabilisiert, ist die softmax-Funktion [19, 80-81].

$$\text{softmax}(x)_i = \frac{\exp(x_i)}{\sum_{j=1}^n \exp(x_j)} \quad (2.8)$$

## 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. Die meisten Optimierungsprobleme werden in Bezug auf Minimierung von  $f(x)$  formuliert. Die Maximierung kann über einen Minimierungsalgorithmus durch Minimieren von  $-f(x)$  erreicht werden. Die Funktion die minimiert oder maximiert werden soll, wird als *Objektive Funktion* bezeichnet. Es wird auch *Kostenfunktion* oder *Verlustfunktion* bezeichnet. Zum Beispiel ist die Funktion  $x^* = \arg \min f(x)$  eine solche Funktion.

### 2.7.1 Gradientenabstiegsverfahren

Der Gradientenabstiegsverfahren ist einer der beliebtesten Algorithmen zur Optimierung neuronaler Netze. Der Gradientenabstiegsverfahren ist ein Weg, um eine Zielfunktion  $J(\theta)$  zu minimieren, die durch die Parameter eines Modells  $\theta \in \mathbb{R}^d$  parametrisiert wird. Die Parameter werden in der entgegengesetzten Richtung des Gradienten der Zielfunktion  $\nabla \theta J(\theta)$  angepasst. Die Lernrate  $\eta$  bestimmt die Größe der Schritte, die unternommen werden, um ein lokales Minimum zu erreichen. Mit anderen Worten, wird der Richtung bergab gefolgt, die durch die Zielfunktion erzeugt wird, bis ein „Tal“ erreicht wird (siehe Abbildung 2.5) [33].

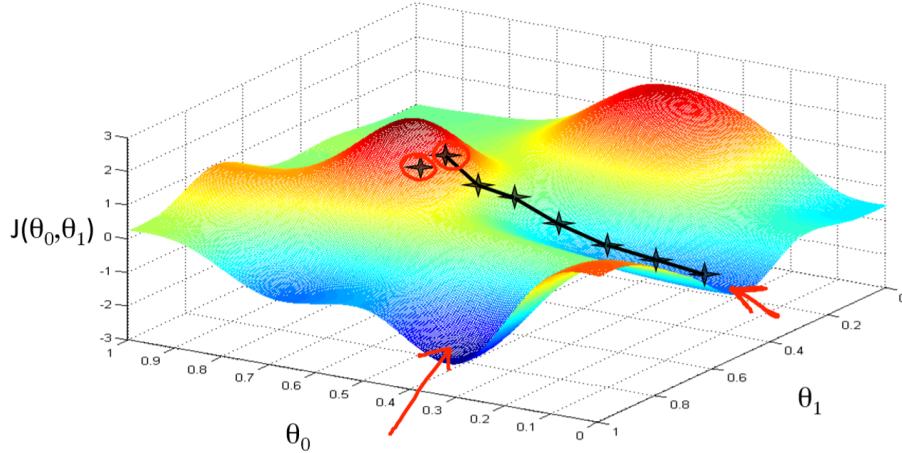


Abbildung 2.5: Darstellung des Gradientenabstiegsverfahrens, entnommen aus [3].

### 2.7.2 Batch Gradientenabstiegsverfahren

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

$$\theta = \theta - \eta \cdot \nabla_{\theta} J(\theta) \quad (2.9)$$

Da der gesamte Datensatz berechnet werden muss, um nur eine Aktualisierung durchzuführen, kann der Batch Gradientenabstiegsverfahren sehr langsam sein und ist für Datensätze, die nicht in den Speicher passen, nicht zu handhaben. Der Batch Gradientenabstiegsverfahren ermöglicht es auch nicht, das Modell mit neuen Beispielen im laufenden Betrieb zu aktualisieren [33].

### 2.7.3 Stochastische Gradientenabstiegsverfahren

Im Gegensatz dazu führt der stochastische Gradientenabstiegsverfahren eine Parameteraktualisierung für jedes Trainingsbeispiel durch. Der Batch Gradientenabstiegsverfahren 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 laufendem Lernen verwendet werden [33].

$$\theta = \theta - \eta \cdot \nabla_{\theta} J(\theta; x^{(i)}; y^{(i)}) \quad (2.10)$$

## 2.8 Backpropagation

Das *Backpropagation*-Verfahren bezieht sich auf die Gewichte eines mehrschichtigen Netzes und wendet dabei praktisch die Anwendung der Kettenregel der Differentialrechnung an. Es wird vom Gradienten in Bezug auf die Ausgabe (oder die Eingabe der Nachfolge) rückwärts gearbeitet. Die Kettenregel sagt dabei aus, wie zwei kleine Effekte (eine kleine Änderung von  $x$  auf  $y$  und der von  $y$  auf  $z$ ) zusammengesetzt sind. Eine kleine Änderung von  $\Delta y$  in  $x$  wird zuerst in eine kleine Änderung  $\Delta y$  in  $y$  umgewandelt, indem sie mit  $\frac{\partial y}{\partial x}$  multipliziert wird (die partielle Ableitung). In ähnlicher Weise erzeugt die Änderung  $\Delta y$  eine Änderung von  $\Delta z$  in  $z$  (siehe Formel 2.8) [22].

$$\begin{aligned}\Delta z &= \frac{\partial z}{\partial y} \Delta y \\ \Delta y &= \frac{\partial y}{\partial x} \Delta x \\ \Delta z &= \frac{\partial z}{\partial y} \frac{\partial y}{\partial x} \Delta x \\ \frac{\partial z}{\partial x} &= \frac{\partial z}{\partial y} \frac{\partial y}{\partial x}\end{aligned}\quad (2.11)$$

Die Gleichungen, die zur Berechnung des Vorwärtsdurchlaufs (siehe Abbildung 2.6) in einem neuronalen Netz mit zwei verborgenen Schichten und einer Ausgangsschicht verwendet werden, bilden jeweils ein Modul, durch das man Gradienten zurück propagieren kann. Auf jeder Ebene wird zuerst die Gesamteingabe  $z$  für jede Einheit berechnet. Dann wird eine nichtlineare Funktion auf  $z$  angewendet, um die Ausgabe der Einheiten zu erhalten. Eine nichtlineare Funktion kann z.B. eine ReLu- oder eine Sigmoid-Funktion oder eine tanh-Funktion sein. Im Rückwärtsdurchlauf (siehe Abbildung 2.6) wird in jeder verborgenen Schicht die Ableitung der Fehler in Bezug auf die Ausgabe jeder Einheit berechnet. Dies geschieht durch Vergleichen der Ausgaben mit der richtigen Antwort [22].

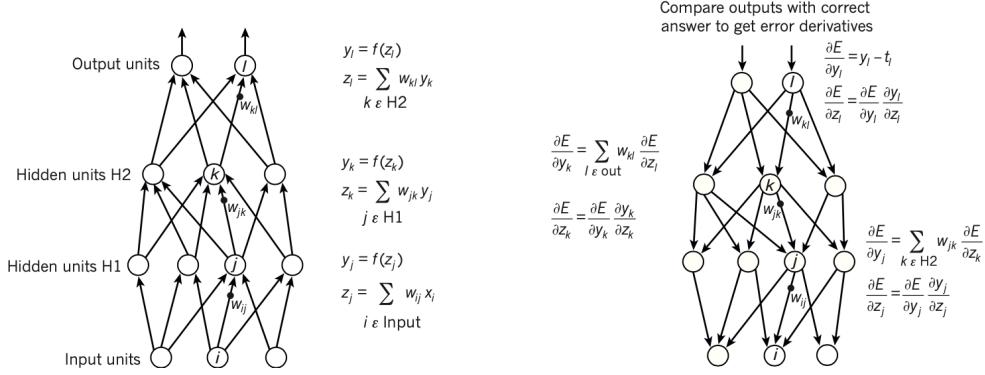


Abbildung 2.6: Darstellung vergleicht die Schritte des Backpropagation Verfahrens nach vorne und hinten (in Anlehnung an [22]).

Die Backpropagation-Gleichung wird wiederholt angewendet. Der Rückwärtsdurchlauf bezieht sich auf die Idee, die Differenz zwischen Vorhersage- und Istwerten zu verwenden, um die Hyperparameter der verwendeten Methode anzupassen. Für die Anwendung ist jedoch immer eine vorherige „Vorwärtspropagation“ erforderlich.

## 2.9 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 Funktionsraum der Verlustfunktion durchquert. Ein großer Koeffizient (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). Sehr kleine Lernraten können mehr Zeit zum Lernen in Anspruch nehmen und die Belastung eines bereits rechenintensiven Prozesses erhöhen [28, 77].

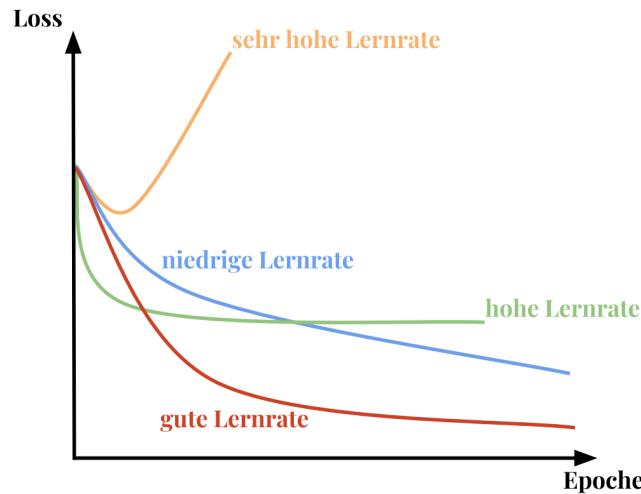


Abbildung 2.7: Vergleich unterschiedlicher Lernraten und deren Effekte auf den Verlust. Bei niedrigen Lernraten ist eine „lineare“ Verbesserungen zu sehen. Mit hohen Lernraten werden sie exponentieller. Höhere Lernraten verringern den Verlust schneller, bleiben jedoch bei schlechteren Verlustwerten hängen (grüne Linie). Dieses liegt daran, dass die Optimierung zu viel „Energie“ enthält und die Parameter „chaotisch herum springen“ und sich nicht an einem Ort in der „Optimierungslandschaft“ niederlassen können (in Anlehnung an [36]).

## 2.10 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 *Unteranpassung*, wie in Abbildung 2.8 dargestellt. Wenn die Linie zu gut zu den Daten passt, besteht das gegenteilige Problem, welches als *Überanpassung* („Overfitting“) bezeichnet wird [28, 27].

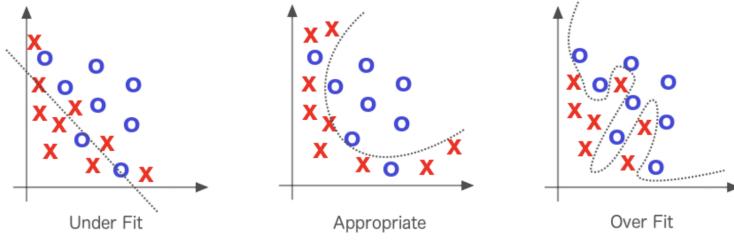


Abbildung 2.8: Die Abbildung vergleicht die Überanpassung mit der Unteranpassung. Die einzelnen Punkte passen sich dem trainierten Modell zu sehr an, der Verlust wird also so klein, dass das Modell nicht mehr zuverlässige Ergebnisse liefern kann. Dies ist darauf zurückzuführen, dass das Modell „zu viel“ 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 [28, 27]).

## 2.11 Regularisierung

Die Regularisierung lässt die Auswirkungen von außer Kontrolle geratenen Parametern minimieren, indem verschiedene Methoden oder Strategien verwendet werden, um die Parametergröße im Laufe der Zeit zu minimieren. Der Hauptzweck der Regularisierung besteht darin, die Überanpassung zu kontrollieren [28, 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. Die Strategien zur Vorbeugung dieser Probleme werden zusammen als Regularisierung bezeichnet. Tatsächlich war die Entwicklung effektiver Regularisierungsstrategien eine der wichtigsten Forschungsanstrengungen auf diesem Gebiet. Regularisierung kann schließlich definiert werden als „jede Änderung, die wir an einem Lernalgorithmus vornehmen, um dessen Generalisierungsfehler, aber nicht seinen Trainingsfehler zu reduzieren“ [19, 228].

### 2.11.1 Early Stopping

Wenn große Modelle trainiert werden, um eine bestimmte Aufgabe zu 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

Validierungsfehler (und damit einem besseren Testfehler) erhalten werden kann, indem zu dem Zeitpunkt mit dem niedrigsten Validierungsfehler zur Parametereinstellung zurückgekehrt wird. Jedes Mal, wenn sich der Fehler der Validierung 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 [19, 246].

### 2.11.2 Dropout

Eine weitere Strategie um Überanpassung zu vermeiden wird in [35] dargestellt. *Dropout* bietet eine rechnerisch kostengünstige, aber leistungsstarke Methode zur Regularisierung dar. Es ist das aufteilen des Netzwerkes in mehreren Teile. Es wird also ein Ensemble aus diesen kleinen Teilen („sub-networks“) gebildet [19, 258].

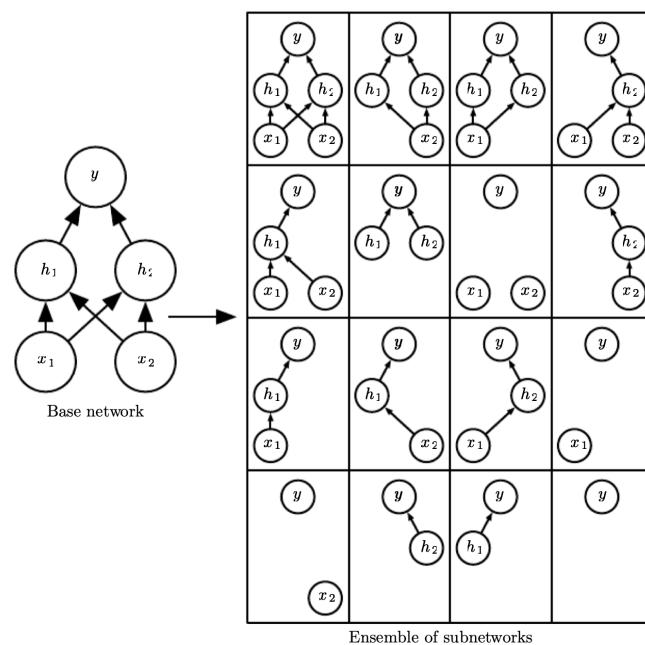


Abbildung 2.9: Dropout trainiert ein Ensemble. Ein Ensemble besteht aus allen Teilnetzwerken. Es wird durch das entfernen von Einheiten aufgebaut. Das Ensemble besteht aus 16 Teilmengen, aus den vier Einheiten des Basis-Netzwerks. Die 16 Subnetze werden durch das Löschen verschiedener Teilmengen von Einheiten aus dem ursprünglichen Netzwerk gebildet (entnommen aus [19, 260]).

### 2.11.3 L1 und L2

Eine weitere Strategie der Regularisierung ist die Modifikation der  $L1$  und  $L2$  Gewichte. Beim Deep Learning wird die Größe von Vektoren, mit einer Funktion, die als „Norm“ bezeichnet wird, [19, 39] gemessen. Formal ist diese Norm  $L^p$  definiert als:

$$\|x\| = \left( \sum_i |x_i|^p \right)^{\frac{1}{p}}. \quad (2.12)$$

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 als  $\|x\|$  bezeichnet, wobei der Index 2 weggelassen wird [19, 39].

Wenn zwischen Elementen zu unterscheiden ist, die genau Null sind und Elementen die klein, aber ungleich Null sind, wird die  $L^1$ -Norm angewendet. Die  $L^1$ -Norm kann vereinfacht werden als [19, 40]:

$$\|x\|_1 = \sum_i |x_i|. \quad (2.13)$$

## 2.12 Metriken im Deep Learning

Im Deep Learning werden oftmals Klassifizierungsprobleme gelöst. In diesem Abschnitt werden die wichtigsten Metriken für das Messen solcher Probleme definiert.

**Precision** Precision (Präzision) gibt die Häufigkeit an, mit der ein Modell bei der Vorhersage der positiven Klasse korrekt war. Das ist:

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives}. \quad (2.14)$$

**Recall** Ist eine Metrik, die folgende Frage beantwortet: Wie viele der möglichen positiven Bezeichnungen hat das Modell korrekt identifiziert? Es wird definiert durch:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}. \quad (2.15)$$

**F1-Score** Der F1-Score ist das harmonische Mittel der Precision und des Recalls. Es wird mit folgender Formel berechnet:

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}. \quad (2.16)$$

## 2.13 Verlustfunktion und Kreuzentropie

### 2.13.1 Verlustfunktionen

Innerhalb eines neuronalen Netzwerks wandelt eine *Verlustfunktion* oder auch *Kostenfunktion* genannt, alle möglichen Fehler in eine Zahl um, um den Gesamtfehler des Netzwerks darzustellen. Im Wesentlichen ist es ein Maß dafür, wie falsch ein Netzwerk liegt.

Das Neuron lernt dadurch, indem es Gewichte und 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 [27, 61]. Gegeben sei die quadratische Verlustfunktion:

$$C = \frac{(y - a)^2}{2}, \quad (2.17)$$

dabei ist  $a$  die Ausgabe des Neurons, wenn die Eingabe des Trainings  $x = 1$  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:

$$\frac{\partial C}{\partial w} = (a - y)\sigma'(z)x = a\sigma'(z) \quad (2.18)$$

$$\frac{\partial C}{\partial b} = (a - y)\sigma'(z) = a\sigma'(z). \quad (2.19)$$

Aus der Abbildung 2.2 ist die Kurve der Sigmoidfunktion zu sehen. Die Kurve wird sehr flach, wenn der Ausgang des Neurons nahe bei 1 liegt, und daher wird  $\sigma'(z)$  sehr klein. Die Gleichungen 2.18 und 2.19 sagen dann aus, dass  $\partial C/\partial w$  und  $\partial C/\partial b$  sehr klein werden. Dies ist der Grund warum das lernen langsamer wird.

### 2.13.2 Kreuzentropie

Nach [27, 62] kann die Lernverlangsamung gelöst werden, indem die quadratische Verlustfunktion durch eine andere Verlustfunktion ersetzt wird. Diese Funktion wird als *Kreuzentropie* bezeichnet. Die Abbildung 2.10 zeigt die Kreuzentropie mit mehreren Eingabeveriablen und entsprechenden Gewichten und Bias. Die Ausgabe des Neurons ist  $a = \sigma(z)$ , wobei  $z = \sum_j w_j b_j + b$  ist, die gewichtete Summe des Inputs. Die Kreuzentropie Kostenfunktion für dieses Neuron wird definiert durch:

$$C = -\frac{1}{n} \sum_x [y \ln a + (1 - y) \ln(1 - a)] \quad (2.20)$$

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, bei der Berechnung der gewünschten Ausgabe  $y$  für alle Trainingseingaben  $x$ . Die Kostenfunktion hat jedoch den Vorteil, dass sie im Gegensatz zu der quadratischen Kostenfunktion das Problem der Verlangsamung des Lernens vermeidet. Um dies näher zu beschreiben, soll die partielle Ableitung der Kreuzentropie in Bezug auf die Gewichte berechnet werden [27, 63].

$$\begin{aligned}
\frac{\partial C}{\partial w_j} &= -\frac{1}{n} \sum_x \left( \frac{y}{\sigma(z)} - \frac{1-y}{1-\sigma(z)} \right) \frac{\partial \sigma}{\partial w_j} = \\
&\quad -\frac{1}{n} \sum_x \left( \frac{y}{\sigma(z)} - \frac{1-y}{1-\sigma(z)} \right) \sigma'(z) x_j \\
\frac{\partial C}{\partial w_j} &= \frac{1}{n} \sum_x \frac{\sigma'(z) x_j}{\sigma(z)(1-\sigma(z))} (\sigma(z) - y) \\
\frac{\partial C}{\partial w_j} &= \frac{1}{n} \sum_x x_j (\sigma(z) - y)
\end{aligned} \tag{2.21}$$

Die Geschwindigkeit, mit der das Gewicht  $w$  lernt, wird durch  $\sigma(z) - y$  gesteuert, also durch den Fehler in der Ausgabe. Je größer dieser Fehler wird, desto schneller lernt das Neuron. Dies ist ein gewünschtes Verhalten. Insbesondere wird die Lernverlangsamung vermieden, die durch den Term  $\sigma'(z)$  in der analogen Gleichung für die quadratische Kostenfunktion 2.18 verursacht wird. Wenn die Kreuzentropie verwendet wird, wird der Term  $\sigma'(z)$  aufgehoben und somit ist es egal, ob es klein ist. Diese Aufhebung ist das Besondere, welches durch die Kreuzentropie Kostenfunktion gewährleistet wird [27, 63-64].

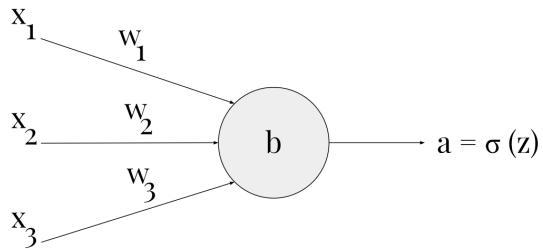


Abbildung 2.10: Das Neuron wird mit 3 Eingabewerten ( $x_1, x_2, x_3$ ) den dazugehörigen Gewichten ( $W_1, W_2, W_3$ ) trainiert. Der Bias ist durch  $b$  angegeben und die Ausgabe mit  $a = \sigma(z)$  (in Anlehnung an [27])

## 2.14 Convolutional Neural Network

CNNs, *Convolutional Neural Networks* oder ConvNets, dienen zur Verarbeitung von Daten in Form mehrerer Arrays, beispielsweise eines Farbbildes aus drei 2-dimensionalen Arrays mit Pixelintensitäten in den drei Farbkanälen. Viele Daten liegen in Form mehrerer

Arrays vor: 1D für Signale und Sequenzen, einschließlich Sprache, 2D für Bilder oder Audio und 3D für Video. Hinter ConvNets stehen vier Schlüsselideen, die die Eigenschaften natürlicher Signale nutzen: lokale Verbindungen, gemeinsame Gewichte, Pooling und die Verwendung vieler Schichten [22]. In den folgenden Abschnitten werden diese Konzepte näher erklärt.

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 weisen 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. 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 und räumlichen Daten verwenden kann. Ein wichtiges definierendes Merkmal von CNNs ist eine Operation, die als Faltung („convolution“) bezeichnet wird. [5, 315-316].

### 2.14.1 Architektur

In CNNs sind mehrere Schichten miteinander verbunden und jeder Schicht hat eine Gitterstruktur. Die Beziehungen zwischen den Schichten werden von einer Schicht zur nächsten vererbt, da jedes Merkmalswert aus 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 [5, 318].

Die Architektur eines typischen ConvNets ist in mehrere Phasen unterteilt. Die ersten Stufen bestehen aus zwei Arten von Schichten: Faltungsschichten und Poolschichten. Einheiten in einer Faltungsebene werden in sogenannten „Feature-Maps“ organisiert, in denen jede Einheit über eine Reihe von Gewichten, die als „Filterbank“ bezeichnet werden, mit lokalen Patches in den Feature-Maps der vorherigen Ebene verbunden ist. Das Ergebnis dieser lokal gewichteten Summe wird dann durch eine Nichtlinearität wie eine

ReLU geleitet. Alle Einheiten in einer Feature-Map verwenden dieselbe Filterbank[22].

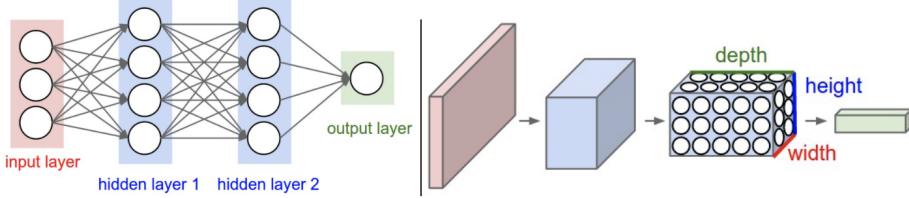


Abbildung 2.11: Links: Ein reguläres 3-Schicht-Neuronales Netz. Rechts: Ein ConvNet. Es ordnet seine Neuronen in drei Dimensionen (Breite, Höhe, Tiefe) an, wie in einer der Ebenen dargestellt. Jede Schicht eines ConvNet wandelt das 3D-Eingangsvolumen in ein 3D-Ausgangsvolumen von Neuronenaktivierungen 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 [37].

### 2.14.2 Convolutional Layer

Die Parameter der Convolutional Layer (Faltungsschicht) 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 Tiefe, weil Bilder die Tiefe 3 haben, also die Farbkanäle). Während des Vorwärtsdurchlaufs 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“ („feature map“) erstellt, die die Antworten dieses Filters an jeder räumlichen Position angeben. Intuitiv lernt die 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 Muster auf höheren Ebenen des Netzwerks. Es entstehen somit ein ganzer Satz von Filtern in jeder Faltungsschicht (z. B. 12 Filter), und jeder von ihnen erzeugt eine separate zweidimensionale Aktivierungskarte. Diese Karten werden entlang der Tiefendimension gestapelt und das Ausgabevolumen erzeugt [37].

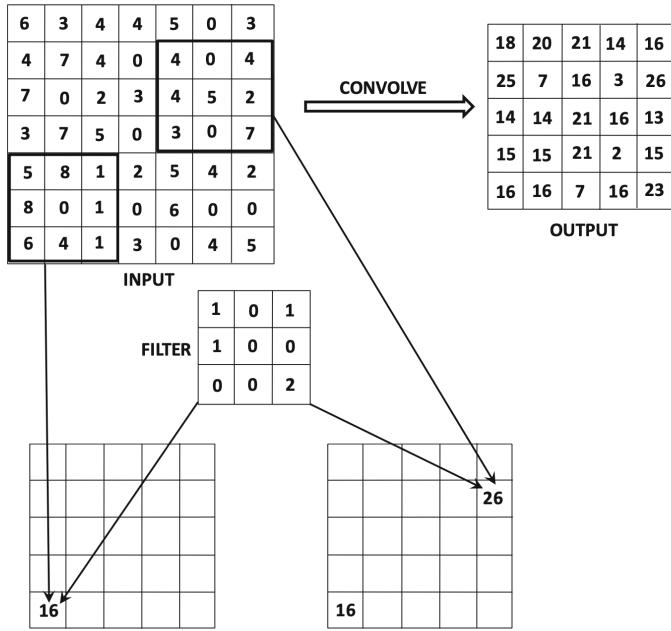


Abbildung 2.12: Die Faltungsoperation geschieht durch eine Punktprodukt Operation des Filters, was über alle räumlichen Positionen wiederholt wird aus [5, 321].

### 2.14.3 Padding

Die Faltungsoperation verringert die Größe der  $(q + 1)$ -ten Schicht im Vergleich zur Größe der  $q$ -ten Schicht. Diese Art der Größenreduzierung ist im Allgemeinen nicht wünschenswert, da sie dazu neigt, einige Informationen entlang der Ränder zu verlieren. Dieses Problem kann durch *Padding* („Auffüllen“) gelöst werden. Beim Padding werden neue Werte rund um die Ränder der Feature-Maps hinzugefügt. Der Wert jedes dieser aufgefüllten Feature-Werte wird auf 0 gesetzt. 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 der Punktprodukt Operation, nur über den Teil der Ebene ersetzt, in dem Werte definiert sind [5, 323].

### 2.14.4 Pooling

Es ist üblich, regelmäßig eine *Pooling Ebene* zwischen aufeinander folgenden Faltungsebenen einer ConvNet-Architektur einzufügen. Die Funktion dieser Ebenen 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 Abschnitt der Eingabe und ändert die Größe räumlich. Die gebräuchlichste Form ist eine Pooling-Ebene mit Filtern der Größe  $2 \times 2$ , die mit einem „Stride“ von 2 Samples pro Scheibe in der Eingabe, um 2 entlang der Breite und Höhe angewendet werden, wobei 75% der Aktivierungen verworfen werden [37].

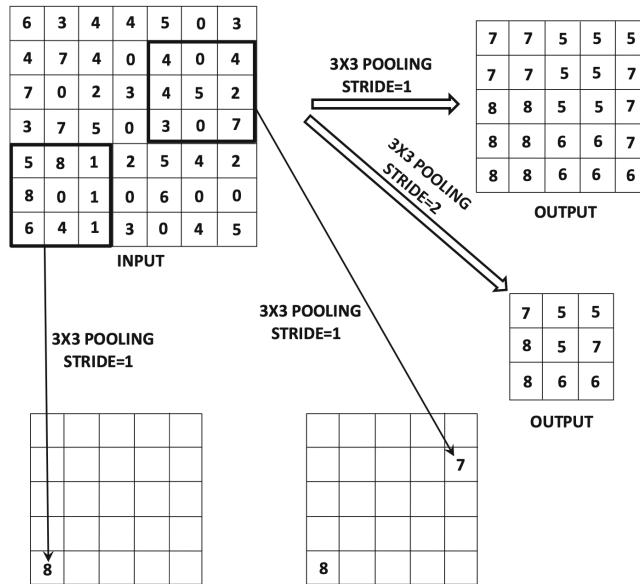


Abbildung 2.13: Ein Beispiel für ein Max-Pooling einer Aktivierungskarte der Größe  $7 \times 7$  mit Stride von 1 und 2. Ein Stride von 1 erzeugt eine  $5 \times 5$ -Aktivierungskarte mit stark wiederholten Elementen aufgrund der Maximierung in überlappenden Regionen. Ein Stride von 2 erzeugt eine 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

### 2.14.5 Vollständig verbundene Ebenen

Am Ende wird die sogenannte *vollständig verbundene Ebenen* des Netzwerkes verwendet, um Klassenwerte zu berechnen, die als Ausgabe des Netzwerks dienen sollen. In einem traditionellen vorwärts gerichteten Neuronalen Netzwerk wird jedes Eingangsneuron mit jedem Ausgangsneuron in der nächsten Schicht verbunden. Dieses wird auch als vollständig verbundene Schicht bezeichnet. Der einzige Unterschied zwischen der vollständig verbundene Ebene und Faltungsschichten besteht darin, dass die Neuronen in der Faltungsschicht nur mit einer lokalen Region der Eingabe verbunden ist. Die Neuronen in beiden Schichten berechnen jedoch immer noch Punktprodukte, sodass ihre funktionale

Form identisch ist. Zwischen beiden Schichten ist eine Konvertierung möglich [37].

# 3 Die Natürliche Sprache

Die *Verarbeitung natürlicher Sprache* NLP, ist ein theoretisch motivierter Bereich von Computertechniken, zum Analysieren und Darstellen natürlich vorkommender Texte auf einer oder mehreren Ebenen der Sprachanalyse, um eine menschenähnliche Sprachverarbeitung für eine Reihe von Aufgaben oder Anwendungen zu erreichen [23].

Der Umgang mit Textdaten ist problematisch, da heutige Computer, Skripte und Modelle für maschinelles Lernen, keinen Text im menschlichen Sinne lesen und verstehen können. Wörter können viele verschiedene Assoziationen aufrufen. Diese sprachlichen Assoziationen sind das Ergebnis recht komplexer Berechnungen beim Menschen. Machine Learning (ML)-Modelle haben dieses vorgefertigte Verständnis der Bedeutung nicht.

## 3.1 Tokenisierung durch Reguläre Ausdrücke

Die Segmentierung eines Textes in seine Einheiten ist die erste Voraussetzung für dessen Weiterverarbeitung. In der Informatik können einzelne Wörter bzw. „Tokens“ z.B. durch Leerzeichen voneinander abgetrennt werden. *Reguläre Ausdrücke* können die Tokenisierung verbessern. Reguläre Ausdrücke können dabei helfen die Sprache so zu trennen, wie angegeben. Sie ist eine Sprache. Diese Sprache wird in allen Computersprachen verwendet. Formal ist ein regulärer Ausdruck eine algebraische Notation zur Charakterisierung einer Reihe von Zeichenfolgen. Sie sind besonders nützlich für die Suche in Texten, wenn ein Muster gesucht wird und ein Korpus von Texten durchsucht werden muss. Eine Suchfunktion für reguläre Ausdrücke durchsucht den Korpus und gibt alle Texte zurück, die dem Muster entsprechen. Der Korpus kann ein einzelnes Dokument oder eine Sammlung sein [20, 3]. Reguläre Ausdrücke sind also bestimmte Regeln die Muster in einem Text erkennen lassen. Neben der Tokenisierung können sie auch für das extrahieren von Informationen aus Textsequenzen verwendet werden.

## 3.2 N-Gramm

Sprachmodelle sind Wortfolgen, zu denen Wahrscheinlichkeiten zugewiesen wurden. Das einfachste Sprachmodell ist das *N-Gramm* Sprachmodell. Es ist eine Folge von N Wörtern (ein 2-Gramm oder Bigramm ist eine Folge von zwei Wörtern usw.). N-Gramm wird meistens dafür verwendet, um das nächste Wort in aus einer Sequenz vorherzusagen [20, 31]. Angenommen ein Korpus besteht aus folgenden 4 Sätzen:

1. Es regnet in Berlin.
2. Es regnet in Köln und es ist 10 Grad.
3. Es regnet und hagelt in ganz Deutschland.
4. In Deutschland herrscht Regen.

Um die Wahrscheinlichkeit  $P(\text{regnet} / \text{in})$  herauszufinden, wird die Anzahl des Wortes „regnet“ im Korpus gezählt. Es wird gezählt, wie oft „regnet“ und „in“ zusammen vorkommen (2 Mal) und dieses wird dividiert durch 3, da „regnet“ insgesamt 3 Mal im Korpus vorkommt. Das Bigramm „regnet in“ hat also eine Wahrscheinlichkeit von 2/3.

## 3.3 Part-of-Speech Tagging

*Part-of-speech-tagging (POS)* ist der Prozess des Zuweisens eines Tags, zu jedem Wort aus einem Eingabetext. Die Eingabe ist eine Folge von tokenisierten Wörtern und einem „Tagset“, und die Ausgabe ist eine Folge von Tags, eines pro Token [20, 148]. Je nach Sprache gibt es verschiedene „Tagger“, für das Deutsche gibt es das Stuttgart-Tübingen-Tagset [2], einige Tags können aus der Tabelle 3.1 entnommen werden. Die Tagger wurden ursprünglich manuell erfasst und in heutiger Zeit durch das Maschinelle Lernen automatisiert.

Tabelle 3.1: Beispiele aus dem Stuttgart-Tübingen-Tagset

POS	Beschreibung	Beispiel
ADJA	attributives Adjektiv	das <i>große</i> Haus
ADJD	adverbiales oder prädikatives Adjektiv	er fährt <i>schnell</i> , er ist <i>schnell</i>
ADV	Adverb	<i>schon</i> , <i>bald</i> , <i>doch</i>
NE	Eigenarten	<i>Hans</i> , <i>Hamburg</i> , <i>HSV</i>
NN	normales Nomen	<i>Tisch</i> , <i>Herr</i> , <i>das Reisen</i>
VVFIN	finites Verb, voll	du <i>gehst</i> , wir <i>kommen</i> an

### 3.4 One-Hot-Encoding

Die verborgenen Schichten eines mehrschichtigen neuronalen Netzwerks lernen die Eingaben des Netzwerks so darzustellen, dass die Ausgaben leicht vorhergesagt werden können. Dies wird demonstriert, indem ein mehrschichtiges neuronales Netzwerk trainiert wird, um das nächste Wort in einer Sequenz aus einem lokalen Kontext vorheriger Wörter vorherzusagen [10]. Jedes Wort im Kontext wird dem Netzwerk als „Eins-aus-N-Vektor“ dargestellt, d.h. eine Komponente hat den Wert 1 und der Rest ist 0, dieses Verfahren wird auch als *One-Hot-Encoding* bezeichnet. In einem *Sprachmodell* lernen die anderen Schichten des Netzwerks, die Eingangsvektoren in einen Ausgangsvektor für das vorhergesagte nächste Wort umzuwandeln, um die Wahrscheinlichkeit vorherzusagen, dass ein Wort im Vokabular als nächstes erscheinen wird [22]. Das Netzwerk lernt „Wortvektoren“, die viele aktive Komponenten enthalten, von denen jede als separates Merkmal des Wortes interpretiert werden kann. Dieses Vorgehen wurde erstmals im Zusammenhang mit dem Lernen verteilter Darstellungen für Symbole demonstriert [34]. Die semantischen Merkmale waren in der Eingabe nicht explizit vorhanden. Das Lernverfahren wurde als eine gute Möglichkeit entdeckt, die strukturierten Beziehungen zwischen den Eingabe- und Ausgabesymbolen in mehrere „Mikroregeln“ zu zerlegen. Das Lernen von Wortvektoren hat sich auch als sehr gut erwiesen, wenn die Wortsequenzen aus einem großen Korpus stammen und die einzelnen Mikroregeln unzuverlässig sind [10].

Das One-Hot-Encoding ist also ein „spärlicher Vektor“ in dem ein Element auf 1 gesetzt wird und alle anderen Elemente auf 0 gesetzt werden. One-Hot-Encoding wird üblicherweise verwendet, um Zeichenfolgen mit einer endlichen Menge möglicher Werte darzustellen.

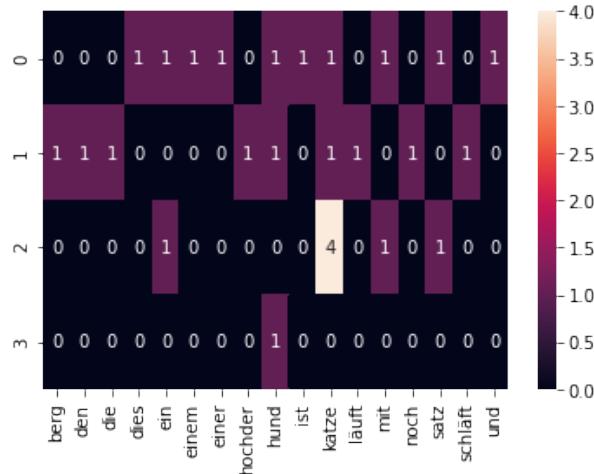


Abbildung 3.1: Die Grafik zeigt wie ein Beispielkorpus welches aus 5 Sätzen besteht, in einer Matrix dargestellt werden kann. Je nach Häufigkeit wird jedes Wort aus dem Wortschatz, entsprechend den Sätzen im Korpus abgebildet. Der Korpus besteht aus folgenden 5 Sätzen: „Dies ist ein Satz mit einer Katze und einem Hund.“, „Die Katze läuft den Berg hoch.“, „Der Hund schläft noch.“, „Ein Satz mit Katze Katze Katze Katze.“ und „Ein Hund.“ Eigene Darstellung.

### 3.5 Worteinbettungen

Die numerischen Werte sollten so viel wie möglich von der sprachlichen Bedeutung eines Wortes erfassen. Eine gut ausgewählte, informative Eingabedarstellung kann einen massiven Einfluss auf die Gesamtleistung des Modells haben. *Worteinbettungen* sind der vorherrschende Ansatz um dieses Problem zu lösen und so weit verbreitet, dass ihre Verwendung praktisch in jedem NLP-Projekt angenommen wird. Unabhängig davon, ob es ein Projekt in den Bereichen Textklassifikation, Sentimentanalyse oder maschinelle Übersetzung ist [22]. Vektoren zur Darstellung von Wörtern werden im Allgemeinen als Einbettungen bezeichnet, da das Wort in einen bestimmten Vektorraum eingebettet wird [20, 99].

Eine Einbettung eine Übersetzung eines hochdimensionalen Vektors in einen niedrigdimensionalen Raum. Beispielsweise kann ein Satz als „spärlicher Vektor“ mit Millionen Elementen (hochdimensional) dargestellt werden. Jede Zelle im Vektor repräsentiert ein separates Wort. Der Wert in einer Zelle gibt an, wie oft dieses Wort in einem Satz vorkommt. Da ein einzelner Satz wahrscheinlich nicht mehr als 50 Wörter hat, enthält fast jede Zelle eine 0. Die wenigen Zellen, die nicht 0 sind, enthalten eine kleine Zahl,

welches die Häufigkeit des Auftretens dieses Wortes darstellt. Als „dichter Vektor“ mit mehreren hundert Elementen (niedrigdimensional), wird ein gegebener Satz dargestellt, indem jedes Element ein Gleitkommawert zwischen 0 und 1 erhält. Dies ist die Einbettung. Die Werte können trainiert werden und sagen mehr aus als die „spärlichen Vektoren“.

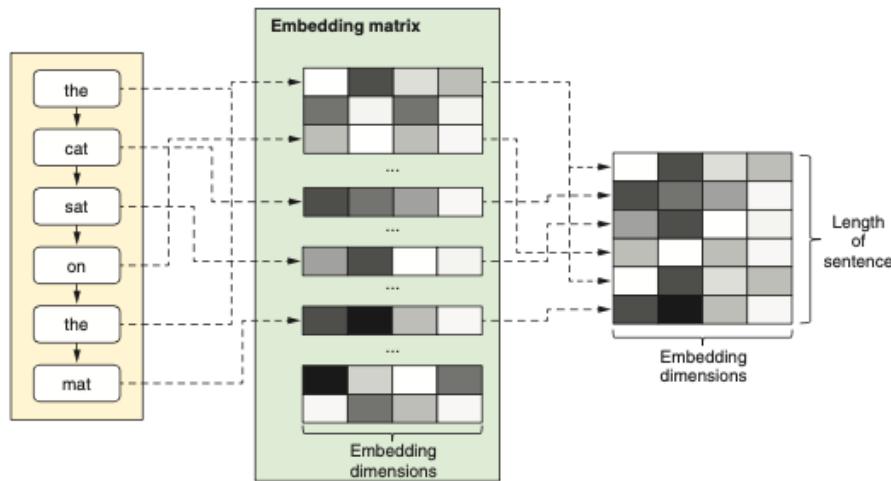


Abbildung 3.2: Jede Zeile der Einbettungsmatrix entspricht einem Wort im Vokabular, und jede Spalte ist eine Einbettungsdimension. Die Werte der Elemente der Einbettungsmatrix, die im Diagramm als Graustufen dargestellt sind, werden zufällig ausgewählt und trainiert. Entnommen aus [13, 311]

Tabelle 3.2: Vergleich des One-Hot-Encodings mit Worteinbettungen [13, 312]

	One-Hot-Encoding	Worteinbettungen
<b>Hartkodiert</b>	ja	nein, gelernt: Die Einbettungsmatrix ist ein trainierbarer Gewichtsparameter. Die Werte spiegeln die semantische Struktur des Wortschatzes nach dem Training wider.
<b>Dichte</b>	spärlich, die meisten Elemente sind Null, einige Eins.	dicht, Elemente nehmen ständig wechselnde Werte an
<b>Skalierbarkeit</b>	nicht skalierbar, Die Größe des Vektors ist proportional zur Größe des Vokabulars.	skalierbar. Die Einbettungsgröße muss nicht mit der Anzahl der Wörter im Vokabular zunehmen.

### 3.6 Word2Vec

*Word2Vec* [26] Worteinbettungen sind Vektordarstellungen von Wörtern, die normalerweise von einem Modell gelernt werden, wenn große Textmengen als Eingabe eingegeben werden (z. B. Texte aus Wikipedia, Wissenschaft, Nachrichten, Artikel usw.). Diese Darstellung von Wörtern erfasst die semantische Ähnlichkeit zwischen Wörtern unter anderen Eigenschaften. Word2Vec-Worteinbettungen werden so gelernt, dass der Abstand zwischen Vektoren für Wörter mit enger Bedeutung (z. B. „König“ und „Königin“) näher ist als der Abstand für Wörter mit völlig unterschiedlichen Bedeutungen (z. B. „König“ und „Katze“).

Bei der One-Hot-Encoding sind die Wörter „gut“ und „großartig“ genauso unterschiedlich wie „Tag“ und „Nacht“. Hier kommt die Idee, verteilte Darstellungen zu erzeugen. Intuitiv wird eine gewisse Abhängigkeit eines Wortes von den anderen Wörtern eingeführt. Die Wörter im Kontext dieses Wortes würden einen größeren Anteil dieser Abhängigkeit erhalten. In der One-Hot Darstellung dagegen, sind alle Wörter unabhängig voneinander.

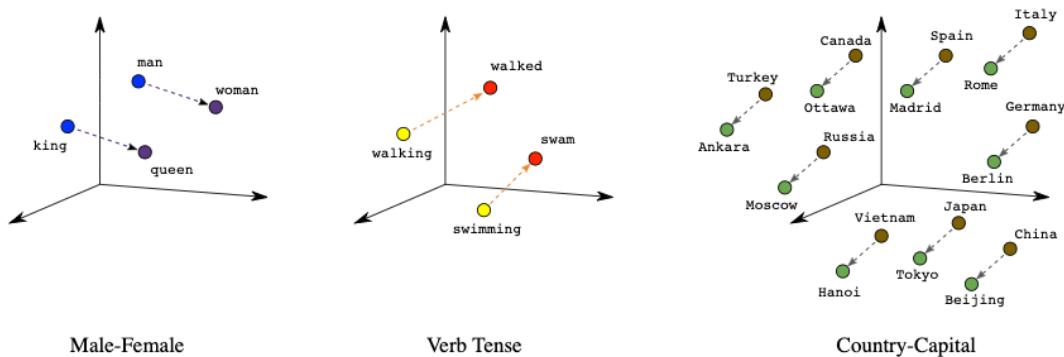


Abbildung 3.3: Durch Worteinbettungen können interessante Analogien zwischen einzelnen Wörtern gefunden werden. (Entnommen aus [1])

### 3.7 Word2Vec mit der Skip-Gram Architektur

Wörter können als spärliche, lange Vektoren mit vielen Dimensionen dargestellt werden. Eine alternative Methode ist die Darstellung eines Wortes mit der Verwendung von kurzen Vektoren, mit einer Länge von 50-1000 und einer großen dichte (die meisten Werte sind nicht Null). Es stellt sich heraus, dass dichte Vektoren in NLP-Aufgaben bessere Ergebnisse abgeben, als spärliche Vektoren. Wenn beispielsweise 100-dimensionale Worteinbettungen als Merkmal verwendet wird, kann ein Klassifikator nur 100 Gewichte lernen, um die Bedeutung des Wortes darzustellen. Wenn stattdessen einen 50.000-dimensionalen

Vektor eingeben wird, müsste ein Klassifikator Zehntausende von Gewichten für jede der spärlichen Dimensionen lernen. Dichte Vektoren können besser verallgemeinern und helfen eine Überanpassung zu vermeiden, da sie weniger Parameter als spärliche Vektoren mit expliziten Zählungen enthalten. Schließlich können dichte Vektoren die Synonyme besser erfassen als spärliche Vektoren. Zum Beispiel sind „Auto“ und „Automobil“ Synonyme und können beide im gleichen Zusammenhang verwendet werden. In einer typischen spärlichen Vektordarstellung sind beide Dimensionen unterschiedliche Dimensionen. Da die Beziehung zwischen diesen beiden Dimensionen nicht modelliert wird, können spärliche Vektoren möglicherweise die Ähnlichkeit zwischen Auto und Automobil als Nachbarn nicht erfassen [20, 110-111].

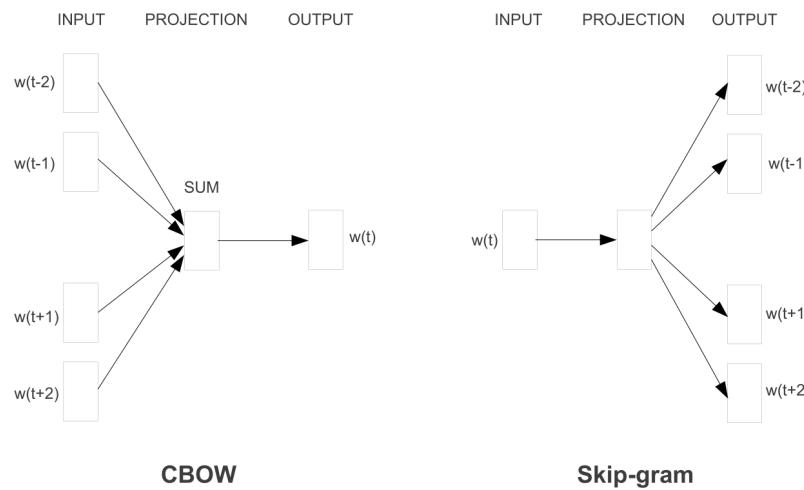


Abbildung 3.4: Die CBOW-Architektur sagt das aktuelle Wort basierend auf dem Kontext voraus, während das Skip-Gramm die umgebenden Wörter voraussagt, wenn das aktuelle Wort gegeben ist [25]).

Der Skip-Gram-Algorithmus und der Continuous Bag of Words (CBOW)-Algorithmus, sind aus dem Softwarepaket *Word2Vec* entstanden [26][25]. Die Intuition von Word2Vec ist, dass anstatt zu zählen, wie oft jedes Wort  $w$  in der Nähe von einem anderen Wort vorkommt, ein Klassifikator für eine binäre Vorhersageaufgabe trainiert wird. Die erlernten Gewichte werden dann als Worteinbettungen bezeichnet [20, 111]. Dabei wird ein Zielwort  $t$  mit Kandidaten aus dem Kontext  $c$  in ein *Tupel* gesetzt.  $P(+/t,c)$  sagt dann aus, wie wahrscheinlich es ist, dass ein Kontextwort  $c$  ein echter Kontext ist. Zum Beispiel sei gegeben der Satz „Wir essen Spaghetti zum Abendessen...“. Wenn der Kontext von  $\pm 2$  Wörter betrachtet wird, und  $t$  das Zielwort „essen“ ist, wird die Klassifikation für das

Tupel (*essen, spaghetti*) „true“ und für das Tupel (*essen, auto*) „false“ zurückgeben [20, 111].

Die Ähnlichkeit eines Wortes zu einem anderen Wort, kann durch den Skalarprodukt berechnet werden. Dies ist zunächst nur eine Zahl zwischen  $-\infty$  und  $+\infty$ . Um daraus eine Wahrscheinlichkeit zu berechnen, wird die *sigmoid*-Funktion  $\sigma(x)$  angewendet. Die Logistikfunktion gibt eine Zahl zwischen 0 und 1 zurück. Um die Wahrscheinlichkeit zu berechnen, muss gewährleistet werden, dass die Summe  $c$  ist das Kontextwort und  $c$  ist nicht das Kontextwort eine 1 ergeben [20, 112].

$$P(-|t, c) = 1 - P(+|t, c) = \frac{e^{-t \cdot c}}{1 + e^{-t \cdot c}} \quad (3.1)$$

Skip-Gramm macht die starke, aber sehr nützliche vereinfachende Annahme, dass alle Kontextwörter unabhängig sind, so dass ihre Wahrscheinlichkeiten multipliziert werden können [20, 112]:

$$P(+|t, c_{1:k}) = \prod_{i=1}^k \frac{1}{1 + e^{-t \cdot c_i}} \quad (3.2)$$

Word2Vec lernt Einbettungen, indem es mit einem anfänglichen Satz von Einbettungsvektoren beginnt und dann die Einbettung jedes Wortes  $w$  iterativ verschiebt, um mehr in der Nähe der Einbettungen von Wörtern zu kommen, die ähneln. Für das Training eines binären Klassifikators sind zunächst negative Beispiele nötig. Das Skip-Gram benötigt mehr negative als positive Beispiele für das Training. Das Verhältnis zwischen positiven und negativen Beispielen wird mit einem Parameter  $k$  festgelegt. Für jede Trainingseinheit  $t, c$  werden  $k$  negative Stichproben erstellt, die jeweils aus dem Ziel  $t$  und einem „noise word“ besteht. Ein „noise word“ ist ein zufälliges Wort aus dem Lexikon, das nicht das Zielwort  $t$  sein darf [20, 113].

Das Ziel des Lernalgorithmus besteht darin, mittels gegebenen positiven und negativen Beispielen diese Einbettungen so anzupassen, dass die Ähnlichkeit der Ziel- und Kontextwortpaare  $(t, c)$  aus den positiven Beispielen maximiert werden und die Ähnlichkeit der Paare  $(t, c)$  aus den negativen Beispielen minimiert wird. Formell lässt sich dieses mit folgender Formel ausdrücken:

$$\begin{aligned}
L(\theta) &= \sum_{(t,c) \in +} \log P(+|t, c) + \sum_{(t,c) \in -} \log P(-|t, c) = \\
&\quad \log \sigma(c \cdot t) + \sum_{i=1}^k \log \sigma(-n_i \cdot t) = \\
&\quad \log \frac{1}{1 + e^{-c \cdot t}} + \sum_{i=1}^k \log \frac{1}{1 + e^{n_i \cdot t}}.
\end{aligned} \tag{3.3}$$

Die stochastische Gradientenabstieg kann verwendet werden, um dieses Ziel zu erreichen, indem die Parameter (die Einbettungen für jedes Zielwort  $t$  und jedes Kontextwort oder „noise word“  $c$  im Vokabular) iterativ modifiziert werden [20, 114].

### 3.8 CNN in der Textverarbeitung

Anstelle von Bildpixeln sind die Eingaben für die meisten NLP-Aufgaben Sätze oder Dokumente, die als eine Matrix dargestellt werden können. Jede Zeile der Matrix entspricht einem Token, normalerweise einem Wort, aber es kann sich auch um ein Zeichen handeln. Das heißt, jede Zeile ist ein Vektor, welches aus Wörtern und Zeichen besteht. Typischerweise sind diese Vektoren Worteinbettungen (niedrigdimensionale Darstellungen) wie Word2Vec oder GloVe, aber sie können auch One-Hot-Vektoren sein, die das Wort in ein Vokabular indizieren [43].

In der Vision (z.B. Klassifikation von Bildern) „gleiten“ die Filter über lokale „patches“ eines Bildes, in NLP jedoch geleitet der Filter über die ganze Zeile der Matrix (Wörter). Daher entspricht die Breite der Filter normalerweise der Breite der Eingabematrix. Die Höhe kann variieren, aber „Schiebefenster“ mit jeweils mehr als 2-5 Wörtern sind typisch. Pixel, die nahe beieinander liegen, sind wahrscheinlich verwandt (Teil desselben Objekts), aber das Gleiche gilt nicht immer für Wörter. In vielen Sprachen können Teile von Phrasen durch mehrere andere Wörter getrennt werden. Der kompositorische Aspekt ist ebenfalls nicht offensichtlich. Es ist klar, dass Wörter in gewisser Weise zusammengesetzt sind, wie ein Adjektiv, das ein Substantiv modifiziert, aber wie genau dies funktioniert, was Darstellungen auf höherer Ebene tatsächlich „bedeuten“, ist nicht so offensichtlich wie im Fall von Computer Vision. Angesichts all dessen, scheinen CNNs nicht gut für NLP-Aufgaben geeignet zu sein. Wiederkehrende neuronale Netze (RNNs) sind intuitiver. Sie ähneln der Art und Weise, wie Menschen die Sprache verarbeiten (oder zumindest wie Menschen denken, dass sie die Sprache verarbeiten). Menschen lesen nacheinander

von links nach rechts. Es stellt sich aber heraus, dass CNNs, die auf NLP-Probleme angewendet werden, recht gut funktionieren. Ein großes Argument für CNNs ist, dass sie schnell sind. Faltungen sind ein zentraler Bestandteil der Computergrafik und werden auf der Hardwareebene auf Graphics Processing Unit (GPU)s implementiert. Mit einem großen Wortschatz kann das Berechnen schnell „teuer“ werden. Faltungsfilter lernen automatisch gute Darstellungen, ohne das gesamte Vokabular darstellen zu müssen [16].

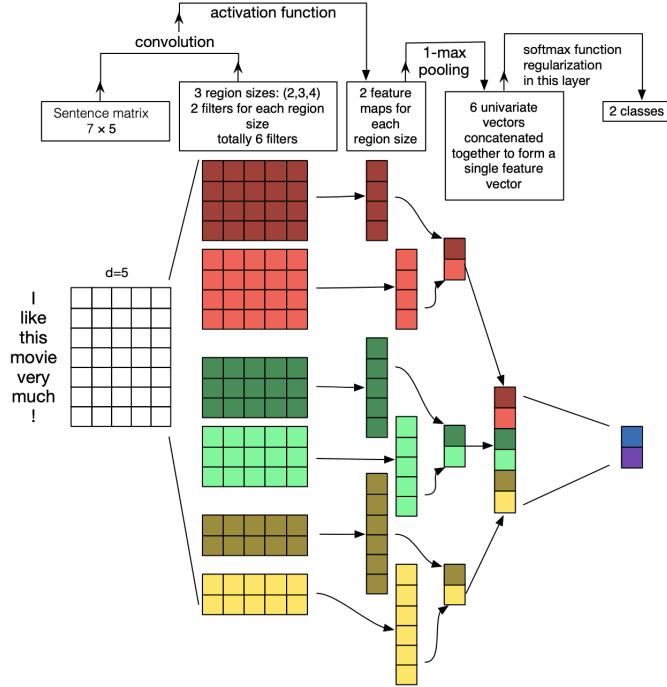


Abbildung 3.5: Illustration einer CNN-Architektur zur Satzklassifizierung. Es sind drei Filterbereichsgrößen vorhanden: 2, 3 und 4. Filter führen Faltungen in der Satzmatrix durch und generieren Feature-Maps (mit variabler Länge). Über jede Karte wird ein 1-Max-Pooling durchgeführt, d. H. die größte Anzahl von jeder Merkmalskarte wird aufgezeichnet. Somit wird aus allen sechs Karten ein univariater Merkmalsvektor erzeugt, und diese sechs Merkmale werden verkettet, um einen Merkmalsvektor für die vorletzte Schicht zu bilden. Die letzte softmax-Schicht empfängt dann diesen Merkmalsvektor als Eingabe und verwendet ihn zur Klassifizierung des Satzes an. Hier wird eine binäre Klassifikation angewendet und es gibt daher zwei mögliche Ausgangszustände [43].

Eindimensionale CNNs können für Sequenzen wie Sätze verwendet werden. Der 1D-CNN-Algorithmus beinhaltet das Gleiten eines Kernels, wobei die Gleitbewegung in

eine Richtung oder Dimension erfolgt. An jeder Gleitposition wird eine Schicht des Inputs extrahiert [13, 312-313]. Angenommen ein Satz besteht aus 9 Wörtern. Jedes Wort wird als Vektor dargestellt. Wenn ein Filter der Größe 2 vorhanden ist, gleitet dieser 8 Positionen nach unten, dieses ist die Größe des „Kernels“. Im nächsten Schritt geschieht eine Punktprodukt Operation zwischen der Eingabe und dem Kernel. Dieses ist ein linearer Prozess und wird bis zum Ende wiederholt.

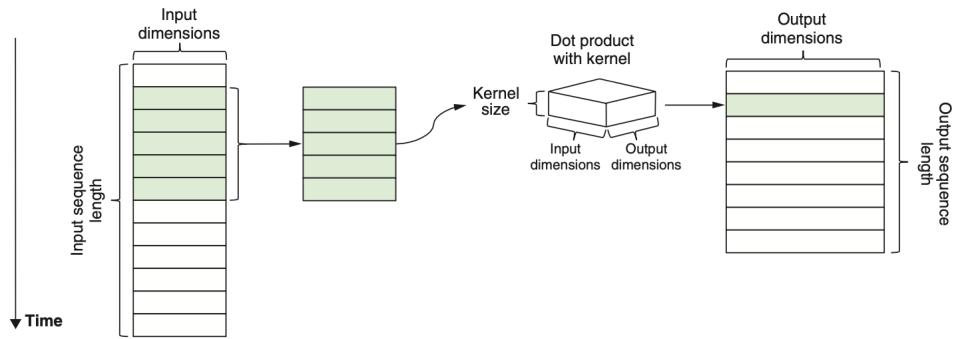


Abbildung 3.6: Darstellung der Faltungsoperation beim 1D-CNN. Entnommen aus [13, 313]

# 4 Clickbaits und die Klassifizierung von Clickbaits

Clickbaits besitzen semantische und syntaktische Eigenschaften, auf die besonders vorgegangen werden muss. Dies geschieht in Form von Analyse und Vorverarbeitung. In diesem Abschnitt werden Ansätze aus der Literatur verglichen, die diese semantischen und syntaktischen Besonderheiten bearbeiten. Die Ansätze aus der Literatur, haben die Oberaufgabe Clickbaits zu Klassifizieren. Damit entstehen Unteraufgaben wie das Sammeln von Daten, Einbetten der Wörter und schließlich entwickeln der Modelle. Die Studie der Literatur soll dabei eine Hilfe sein und bestimmte Ansätze sollen übernommen werden. Zunächst muss darüber erforscht werden, was Clickbaits überhaupt sein können. Nachdem eine Einordnung von Clickbaits stattgefunden hat, ist im nächsten Schritt zu erforschen, welche Ansätze es gibt, bezüglich der Klassifikation von Clickbaits. Diese zwei Fragen werden in diesem Kapitel durch Literatur beantwortet.

## 4.1 Was sind Clickbaits?

Was sich hinter einer Schlagzeile oder einem Titel verbirgt ist ohne weiteres nicht erkennbar. Die meisten Menschen nehmen heutzutage Ihre Nachrichten über soziale Medien auf. Auf sozialen Medien werden meistens nur die Schlagzeilen gezeigt und wenn der Nutzer einen entsprechenden Eintrag interessant oder lesenswert findet, klickt er auf diesen Link. Das Wort Clickbait stammt aus den beiden Wörtern *Click* („Klick“) und *bait* („Köder“). Es ist also eine „Falle“. Viele Medienunternehmen, teilweise auch große, nutzen diese Falle um mehr Klicks zu generieren. Das Problem ist dabei oftmals, dass die Nutzer durch die „Spannung“ oder „Frage“ in der Schlagzeile eine Erwartung haben, die nicht immer oder nur teilweise erfüllt werden kann. In der Studie von [24] wurden ca. 1,6 Mio. Facebook-Posts untersucht und es wurde festgestellt, dass 25 bis 60% der Fälle Clickbaits-Posts waren, je nach Medienunternehmen und Thema.

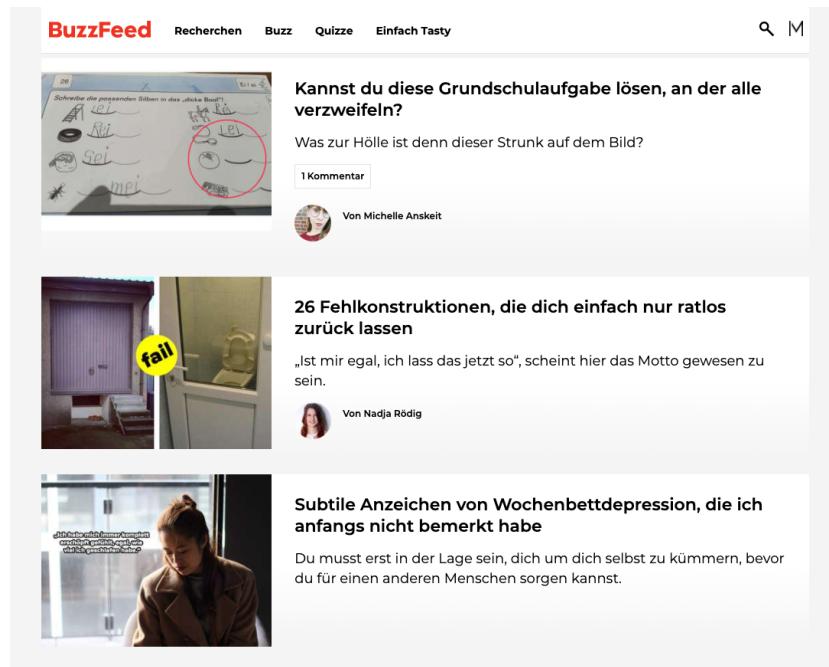


Abbildung 4.1: Beispiele von Clickbaits Nachrichten im Internet

Abbildung 4.2: Beispiele von Nicht-Clickbaits Nachrichten im Internet

Es gibt keine eindeutige Definition für Clickbaits. Clickbaits weisen unterschiedliche Formen. Nach [11] sind Clickbaits, eine Praxis um Klicks zu generieren, durch eine attraktive Überschrift, wessen Inhalt die Erwartungen nur teilweise oder gar nicht erfüllt. Clickbaits können also als eine Täuschung oder Trick der Medien betrachtet werden. Eine andere Definition [30] betrachtet das Thema Clickbaits eher als Werbung für Onlineinhalte und die damit verbundene Verbreitung oder „Viral gehen“ durch die sozialen Medien.



Abbildung 4.3: Die 4 Kernformen des Clickbaits in Anlehnung an [9, 71]

In [9, 70-71] werden Clickbaits in 4 Unterkategorien aufgeteilt. Der Autor bezieht sich dabei auf 4 Quellen und teilt Clickbaits als *Listicles* [40], *Fragen* [21], *Forward Referencing* [12] und *Thumbnails* [42] auf. Diese 4 Kategorien betrachtet der Autor als Kernkategorien. Aus [9, 71] werden Clickbaits außerdem in weitere Gestaltungsformen wie etwa, dass sie „Übertreibungen“ sein können also falsche Versprechen geben oder irreführend sind und unklar sein können. Es wird außerdem auf die unangemessene und vulgäre Sprache und der Formatierung, wie etwa den übertriebenen Gebrauch von Großschreibung oder Satzzeichen, aufmerksam gemacht. Der Autor [9, 75-76] macht deutlich, dass Clickbaits kein neu eingesetztes Stilmittel im Journalismus seien. Schließlich verwendet der Autor den Begriff „Neugier“ beim Leser. Leser wollen oftmals über Themen wie Tod, Gewalt, Sex und Prominente [38] unterhalten werden.

Fraglich ist also, ob diese Themen auch als Clickbait betrachtet werden können oder

wo die Grenze liegt zwischen Medien die Unterhalten und Clickbaits. Es sollte schließlich einem Leser klar sein, dass bestimmte Nachrichten den Zweck der reinen Unterhaltung haben. Die Frage ob diese Art der Nachrichten Clickbait ist oder Unterhaltung sollte also nicht nur am Titel selbst, sondern auch im Zusammenhang mit dem Inhalt gemessen werden. Leider hat wohl jeder Mensch eine andere Erwartung, sodass eine solche Messung für jeden Unterschiedlich sein kann.

## 4.2 Wie werden Clickbaits klassifiziert?

In diesem Abschnitt geht es darum, aus der Literatur die aktuelle Forschung im Zusammenhang mit der Klassifikation von Clickbaits zu ermitteln. Diese Analyse hat den Zweck bestimmte „Erfolgsmodelle“ zu erkennen und von diesen evtl. in den späteren Teilen der Arbeit Gebrauch machen zu können.

In der Studie [14] wurde eine Browser-Erweiterung erstellt, welches Clickbaits erkennen soll. Es wurden umfangreiche Daten sowohl für Clickbaits als auch für Nicht-Clickbait-Kategorien gesammelt. Für die Nicht-Clickbaits-Kategorien wurden 18.513 Wikinews Artikeln gesammelt. Der Vorteil dieser Artikel ist, dass diese von einer Community erstellt werden und jeder Nachrichtenartikel vor der Veröffentlichung von der Community geprüft wird. Es gibt Stilrichtlinien, die eingehalten werden müssen. Um Clickbaits zu finden, haben die Autoren manuell aus Seiten wie „Buzzfeed“ oder „Upworthy“ ca. 8000 Titel gecrawlt. Um falsche Negative zu vermeiden (d. h. die Artikel in diesen Bereichen, bei denen es sich nicht um Clickbaits handelt), wurden sechs Freiwillige rekrutiert um die Überschriften zu labeln. Schließlich wurden 7500 Titel zu jeder der beiden Kategorien zugefügt und es stand ein Datensatz der Größe 15.000 Artikel zur Verfügung. Um Daten zu erfassen, können auch Soziale Medien wie Twitter herangezogen werden, wie im Beispiel von [29].

Laut [14] sind die herkömmlichen Nicht-Clickbait Schlagzeilen kürzer als Clickbait Schlagzeilen. Traditionelle Schlagzeilen enthalten in der Regel meistens Wörter, die sich auf bestimmte Personen und Orte beziehen, während die Funktionswörter (Nomen, Verben und Adjektive) den Lesern zur Interpretation aus dem Kontext überlassen bleiben. Es wird hier als Beispiel gegeben „Visa-Deal oder kein Migranten-Deal, Türkei warnt EU“. Hier sind die meisten Wörter Inhaltswörter, die die wichtigsten Erkenntnisse aus der Geschichte zusammenfassen, und es gibt nur sehr wenige Verbindungsfunktionswörter zwischen den Inhaltswörtern. Auf der anderen Seite sind Clickbait Schlagzeilen länger. Die Sätze, enthalten sowohl Inhalts- als auch Funktionswörter. Ein Beispiel für solche Schlagzeilen ist „Ein 22-Jähriger, dessen Ehemann und Baby von einem betrunkenen

Fahrer getötet wurden, hat ein Facebook-Plädoyer veröffentlicht“. Obwohl die Anzahl der Wörter in Clickbait-Schlagzeilen höher ist, ist die durchschnittliche Wortlänge kürzer. Es werden häufig Wörter verwendet wie „Sie werden“, „Sie sind“. Im Durchschnitt haben die Wörter bei den Clickbaits längere Abhängigkeiten als Nicht-Clickbaits. Der Hauptgrund ist die Existenz komplexerer Phrasensätze im Vergleich zu Schlagzeilen ohne Clickbait. Es ist außerdem zu sehen, dass in Clickbait-Schlagzeilen Stoppwörter häufiger verwendet werden. Clickbait Überschriften verwenden häufig Determinantien wie „ihre“, „meine“, die auf bestimmte Personen oder Dinge im Artikel verweisen. Die Verwendung dieser Wörter trägt in erster Linie dazu bei, den Benutzer neugierig auf das Objekt zu machen, auf das verwiesen wird, und ihn zu überzeugen, den Artikel weiter zu verfolgen.

Um lexikalischer, semantische, orthografische und morphologische Merkmale zu erfassen, benutzen die Autoren aus [7] neben Worteinbettungen auch Zeicheneinbettungen. Um Informationen außerhalb einzelner oder fester Wortfenster zu erfassen, untersuchten die Autoren dabei verschiedene RNN-Architekturen wie LSTM, Gated Recurrent Units (GRU) und Standard-RNNs. Dieses sind Wiederkehrende neuronale Netzwerkmodelle, die für sequentielle Daten wie Sprache und Text gut modelliert werden können.

Die Arbeit von [6] schlägt ein Modell vor, welches CNNs benutzt. CNNs werden für verschiedene Deep-Learning-Aufgaben verwendet. Es wurde hier nur eine Faltungsschicht in das CNN-Modell eingebaut. Die erste Schicht wird für das Einbetten der Wörter in Vektoren verwendet. Dabei wurden 2 verschiedene Worteinbettungen in Bezug genommen. Ein vortrainiertes und eines welches von Grund auf lernen musste und sich während des Trainings weiterentwickelt. In der nächsten Schicht werden Filter in verschiedenen Größen verwendet um Faltungen über Wortvektoren zu erzeugen.

Die Autoren der Arbeit aus [31] Clustern zunächst die aus den Schlagzeilen erstellten Vektoren mit der sogenannten t-Distributed Stochastic Neighbor Embedding (t-SNE) Methode nach [39]. Dieser Algorithmus rekategorisiert die Schlagzeilen in mehrere Gruppen und reduziert die vielen Dimensionen von Clickbaits im Datensatz. Die Autoren beginnen erst im nächsten Schritt mit dem Training. Dabei entstehen 11 Kategorien von Clickbaits (unter Anderem Kategorien wie „Mehrdeutig“, „Übertreibung“ oder „Neckerei“).

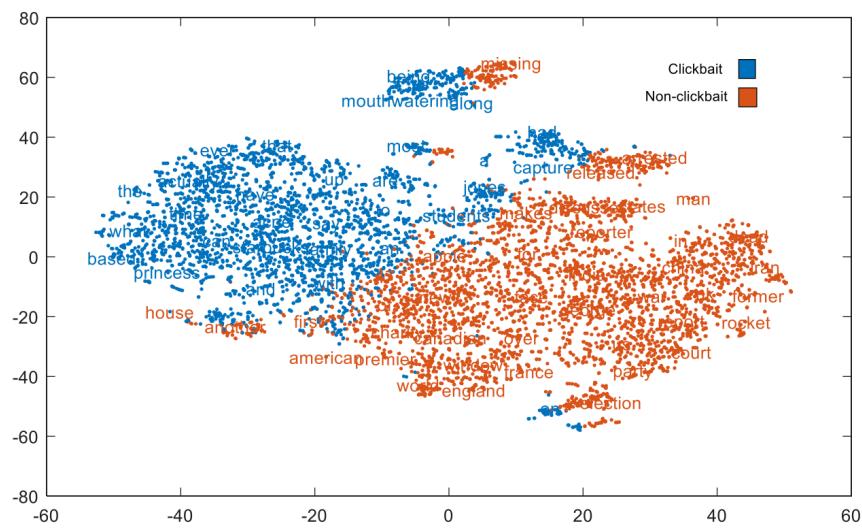


Abbildung 4.4: Die Autoren verwenden das t-SNE Algorithmus nach [39] um die Dimensionen zu Reduzieren und es entstehen mehrere Unterkategorien von Clickbaits. Entnommen aus [31].

## 5 Konzeption und Methodik

In diesem Kapitel geht es darum, den Konzept des praktischen Teils der Arbeit wiederzugeben. Der praktische Teil der Arbeit besteht aus 3 Bereichen. Zunächst wird ein Datensatz erstellt, dann wird anhand der Daten aus diesem Datensatz ein Modell trainiert und schließlich wird dieses Modell in eine Webanwendung umgewandelt (siehe Abbildung 5.1). In diesem Abschnitt soll klargestellt werden, wie ein clientseitiges Deep Learning für deutsche Clickbaits stattfinden kann. Es wird also untersucht, welche Methoden angewendet werden können und es werden die ersten Vorüberlegungen vorgestellt, welche in den weiteren Kapiteln durch praktische Anwendung explizit werden.

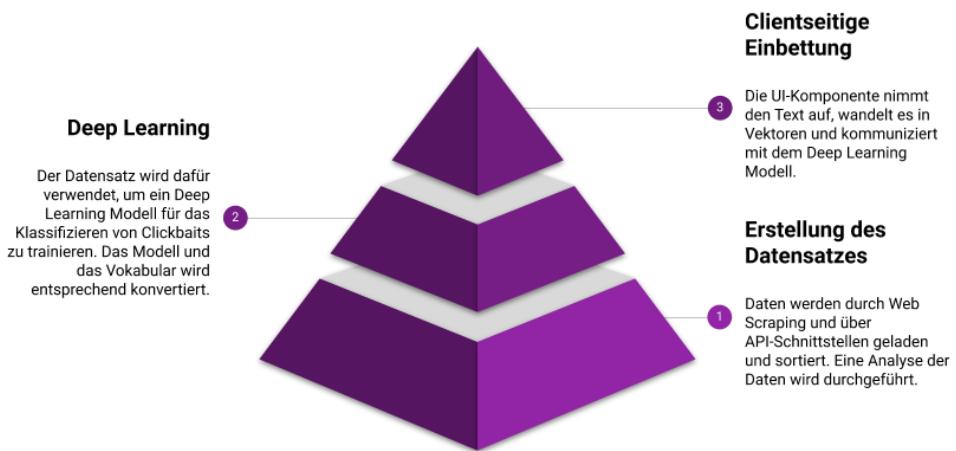


Abbildung 5.1: Darstellung der Konzeption

Deep Learning Modelle benötigen eine große Menge an Beispielen um zu lernen. Neben der Anzahl an Daten, muss auch ein Zusammenhang zwischen den Daten und dem Modell herrschen. In der Arbeit von von [14], haben die Autoren Daten aus dem Web geladen. Sie sind dabei so vorgegangen, dass sie bestimmte Seiten gecrawlt haben, die

viele Clickbaits beinhalten und als Kontrast zu diesen Nachrichten Wikinews herangezogen haben. Ähnlich kann also vorgegangen werden. Es müssen zunächst einige Seiten gefunden werden, die deutsche Clickbaits oder „Unterhaltungsnachrichten“ mit ähnlichen Charakter wie Clickbaits gefunden werden. Wikinews bietet auch eine deutsche Version. Wikipedia bietet eine offene Application Programming Interface (API), welches auch für die Nachrichten benutzt werden kann. Diese API kann dafür verwendet werden, um deutsche Nachrichtentitel zu extrahieren. Die Wikipedia-News-API bietet einen Kostenlosen Endpunkt an, welches verschiedene Nachrichtenportale wie z.B. Politik, Wirtschaft, Kultur, Sport und Wissenschaft anbietet. Die Datenaquise für Clickbaits verläuft dagegen anders. Der Datensatz von [14] beinhaltet jeweils 7.500 Nachrichten je Kategorie. Es ist zu berücksichtigen, dass diese Studie nicht mit Deep Learning Modellen arbeiten, sondern mit klassischen Machine Learning Algorithmen. Um die Datenextraktion zu automatisieren bietet sich die Software *Scrapy* an. Scrapy ist ein Open-Source Tool, welches das Extrahieren von großen Mengen an Daten erleichtert. Es ist in Python geschrieben und kann durch seine Middleware Funktion erweitert werden. Die Daten können so z.B. direkt in eine Structured Query Language (SQL)-Datenbank geschrieben werden.

Der mittlere Kern der Pyramide in Abbildung 5.1 ist das Deep Learning Modell, welches als Inferenzmaschine betrachtet werden. Es wird durch die Eingabe der gekennzeichneten Daten trainiert und in ein entsprechendes Format gebracht. Es gibt mehrere Anbieter für das Deep Learning. Die meisten von Ihnen sind Open Source. Die bekanntesten Bibliotheken für das Deep Learning sind *TensorFlow* (welches von Google unterstützt wird) und *PyTorch* (welches von Facebook unterstützt wird). Seit März 2018 bietet TensorFlow die Möglichkeit, Modelle in der Programmiersprache JavaScript zu trainieren und zu benutzen. Klassischerweise werden Deep Learning Modelle dem Endbenutzer durch das Web angeboten. Es wird zunächst ein Modell trainiert und dann auf einem Server geladen. Der Server bietet einen Hypertext Transfer Protocol (HTTP)-Endpunkt an, welches die Schnittstelle zwischen dem Nutzer und des Modells ist. Durch *TensorFlow.js* wird das Modell in den Speicher des Nutzers, also in seinen Browser geladen. Dann kann der Nutzer mit diesem Modell im Speicher wesentlich einfacher kommunizieren. Um dieses zu verwirklichen sind aber bestimmte Voraussetzungen nötig. Erstens muss dass Modell in irgendeiner Art und Weise in einer Form sein, welches eine solche Benutzung zulässt. Zweitens muss das Modell die Eingabe des Nutzers verstehen. Drittens muss das Modell in die Umgebung eingebaut werden. Das Modell kann auch komplett auf *TensorFlow.js* trainiert werden. *TensorFlow.js* läuft in JavaScript, kommuniziert aber mit der *TensorFlow*-API. Es bieten sich zwei Möglichkeiten des Trainings an. Entweder kann das Modell in JavaScript mit *TensorFlow.js* trainiert werden oder das Modell wird

herkömmlich mit TensorFlow trainiert und dann durch eine Konvertierung in ein Format gebracht, welches in TensorFlow.js laufen kann.

Das Ergebnis der Arbeit ist die Einbettung des Modells in eine Benuteroberfläche. Es sollte eine Benuteroberfläche entstehen, welches die Eingabe des Nutzers versteht und in einer Art und Weise dem Modell, welches in den Browser geladen werden muss, weitergibt. Das Modell muss mit der User Interface (UI) kommunizieren. Das Frontend wird in JavaScript geschrieben. Im modernen JavaScript können mit Frontend-Bibliotheken das Frontend flexibler und einfacher entwickeln und sind einfacher aufrechtzuhalten. Es wird die Bibliothek *React* verwendet, um den Zusammenhang zwischen Deep Learning und moderner Webentwicklung zu verwirklichen.

# 6 Erstellung des Datensatzes

## 6.1 Datenbeschaffung durch Web Scraping

Um die erste Forschungsfrage zu beantworten, muss ein Korpus erstellt werden, für deutsche Clickbaits. Für die Erstellung eines solchen Korpus sind zunächst bestimmte Seiten einzuzgrenzen, die Clickbait Nachrichten anbieten. Die bekannteste Seite ist „buzzfeed“<sup>1</sup>. Außerdem wurden auch Webseiten wie „web.de“ oder „tv-movie“ nach Clickbaits gescannt. Das Screening wurde am November 2020 durchgeführt. Um jedoch zu gewährleisten, dass die Nachrichten zeitlich weit auseinander liegen und somit eine größere Diversität haben, wurden Webseiten ausgewählt die Ihre Nachrichten in gewisser Weise Archivieren. Die gesammelten Daten reichen teilweise über ein Jahr zurück. Wie im Listing 6.1 in Zeile 12 zu sehen ist, besitzt die Seite im Beispiel eine Pagination-Funktion. Dieses Pagination reicht in eine weite Vergangenheit zurück.

Tabelle 6.1: Vergleich der Anzahl und Herkunft Rohdaten nach dem Scrapingvorgang

Quelle	Methodik	Elemente
de.wikinews.org	API-Zugang	10.612
web.de	Web Scraping	14.163
tvmovie.de	Web Scraping	9.428
buzzfeed.de	Web Scraping	798
promipool.de	Web Scraping	26.779
heftig.de	Web Scraping	605
frauenseite.net	Web Scraping	148
bravo.de	Web Scraping	7.476
Summe Clickbaits		59.407
Summe Nachrichten		10.612
Summe insgesamt		70.019

Das Verfahren wurde automatisch mittels eines programmierten Scrapers je Seite durchgeführt. Ein Scraper ist ein Programm, welches Informationen aus einer Webseite

---

<sup>1</sup>Beinhaltet außerdem Schlagzeilen aus den Unterseiten „tasty“ und „quiz“

automatisch extrahieren kann. Dieses gelingt, indem es nach Hypertext Markup Language (HTML)- oder Cascading Style Sheets (CSS)-Attributten oder sogar Asynchronous JavaScript and XML (AJAX)-Anfragen durchsucht oder diese imitiert. Im Listing 6.1 wird ein Web-Scraper vorgestellt. Scrapy führt in einer Schleife HTTP-Requests durch welche jeweils eine Antwort bekommen. Es können auch AJAX-Anfragen gesendet werden (z.B. ein POST-Request). Wenn die Anfrage erfolgreich ist, kann die daraus resultierende Antwort auf bestimmte Eigenschaften und Attribute durchsucht werden (im Beispiel werden CSS-Attribute durchsucht). Diese Attribute beinhalten meistens die gewünschte Information, welche dann wie in der `parse_url` und `parse_page` Methode gezogen und in eine Datenbank gespeichert wird.

```
1 import scrapy
2 from scrapy.http.request import Request
3 from datetime import datetime
4 from klickscraper.items import WebdeItem
5
6
7 class WebdeSpider(scrapy.Spider):
8     name = "webde"
9     scraped_at = datetime.now()
10
11     def start_requests(self):
12         for i in range(287):
13             url = f"https://web.de/magazine/unterhaltung/stars/p{i}"
14             yield Request(
15                 dont_filter=True,
16                 callback=self.parse_url,
17                 url=url)
18
19     def parse_url(self, response):
20         follow_ulrs = response.css(
21             ".teaser-article__full").css("a::attr(href)").getall()
22         for f in follow_ulrs:
23             yield Request(
24                 url=f,
25                 dont_filter=True,
26                 callback=self.parse_page)
27
28     def parse_page(self, response):
29         if len(response.css("p::text").getall()) > 10:
30             text = " ".join(response.css("p::text").getall())
31         else:
32             text = None
```

```

33     yield WebdeItem(title=response.css("title::text").get(),
34                     text=text,
35                     scraped_at=self.scraped_at)

```

Listing 6.1: Beispiel eines Scrapers

## 6.2 Rohdatenanalyse

Um aus den Rohdaten Schlüsse über die sprachlichen Eigenschaften zu ziehen wird im folgenden Abschnitt eine explorative Rohdatenanalyse durchgeführt. Um dies möglichst zu vereinfachen, wird der Rohdatensatz, welches in einer SQL-Datenbank gespeichert wurde in ein Pandas<sup>2</sup> Dataframe umgewandelt. Aus der Abbildung 6.1 ist zu sehen, dass die Wortlänge in den Schlagzeilen der Wikinews Schlagzeilen länger ist. Dieses Phänomen war nach der Literaturanalyse aus Kapitel 4 zu erwarten. Neben der Länge der Wörter aus den Schlagzeilen, ist auch die Analyse der Wortauswahl relevant. Aus der Literaturstudie können bestimmte Merkmale von Clickbaits festgestellt werden. Clickbaits enthalten meistens eine Frage oder Zahlen. Außerdem ist bei Clickbaits auch eine gewisse Wortauswahl festzustellen. Die Zuordnung der Wörter je Clickbait-Schlagzeile zu den Wortarten (Part-of-speech-Tagging oder POS-Tagging) wurde mittels der Python NLP-Bibliothek Spacy durchgeführt. Auch hier wurden Erkenntnisse aus Kapitel 4 berücksichtigt. Die Ergebnisse dieser Analyse sind aus der Tabelle 6.2 zu entnehmen.

Tabelle 6.2: Vergleich der Ergebnisse des POS-Taggings

<b>TAG</b>	<b>Auffälliges Wort (Häufigkeit)</b>
ADJA (attributives Adjektiv)	neu (1759), neue (1474)
ADJD (adverbiales Adjektiv)	krass (171), wirklich (349)
ADV (Adverb)	so (5142), endlich (357)
PDAT (Demonstrativpronomen)	dies (3566)
PROAV (Pronominaladverb)	darum (588), deshalb (251)
PWAT (Interrogativpronomen)	welch (154)
PWAV (Relativpronomen)	wie (488) warum (184)

---

<sup>2</sup>Pandas ist ein schnelles, flexibles und benutzerfreundliches Open-Source-Tool zur Analyse und Bearbeitung von Daten in Python.

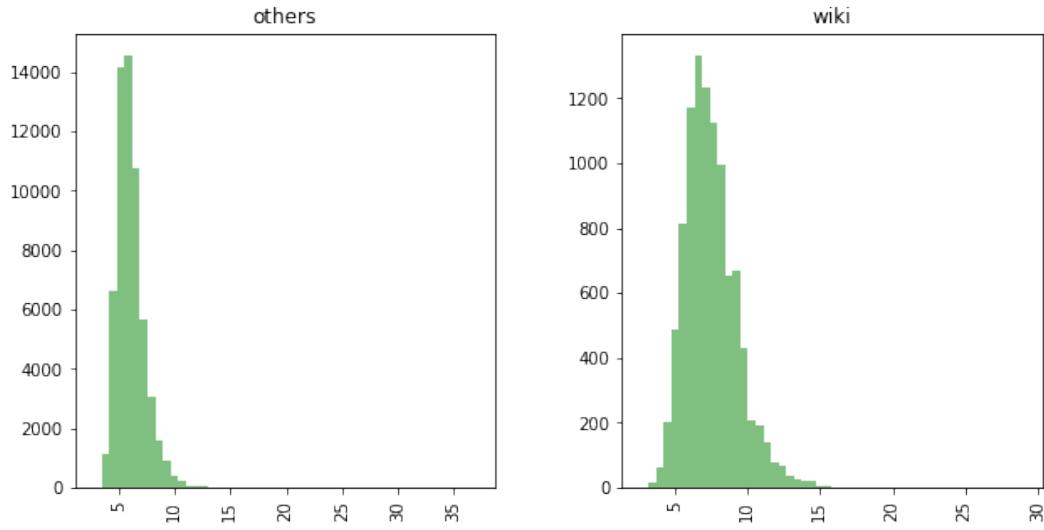


Abbildung 6.1: Die Rohdatenanalyse zeigt, dass das Histogramm rechts (Wikinews) eine größere Varianz hat, als das Histogramm links (Clickbaits).

### 6.3 Labeln der Daten

Aus der Tabelle 6.1 ist zu entnehmen, dass ca. 60.000 potenzielle Clickbaits vorhanden sind. Es wird angestrebt einen Datensatz in der Größenordnung von 10.000 - 20.000 Schlagzeilen zu generieren. Mit den 10.000 Wikinews Artikeln ist die erste Hälfte des Datensatzes praktisch vorhanden. An diesem Datensatz muss nichts mehr gelabelt werden, da angenommen wird, dass diese Nachrichten einen Standard haben, sodass diese als Nicht-Clickbaits gekennzeichnet werden. Fraglich ist jedoch, ob nur die Auswahl eines Mediums ausreichend ist. Um die restlichen 10.000 Schlagzeilen, also die Clickbaits zu ermitteln, bietet sich die Möglichkeit, die Daten per Hand zu kennzeichnen. Diese Aufgabe nimmt jedoch relativ viel Ziel in Anspruch, sodass für diese Aufgabe ein „regelbasiertes Labeln“ ausgeführt werden kann. Bestimmte Muster und Eigenschaften können durch Techniken der NLP programmatisch erkannt und somit das Labeln automatisiert werden. Fraglich ist jedoch, die Qualität dieses Ansatzes. Dieser Ansatz kann das menschliche Labeln natürlich nicht ersetzen und sollte wenn möglich erweitert und verbessert werden.

Es können folgende Muster und Eigenschaften erkannt werden:

1. Clickbaits sind meistens Fragen wie „Welcher Schuh passt dir am meisten?“
2. Clickbaits enthalten meistens Zahlen in Form eines Listings „Das sind die 10 schnellsten Autos“

3. Clickbaits haben eine niedrigere Wortlänge als normale Nachrichten
4. Clickbaits beinhalten einige für sie markanten Wörter wie „diese“ oder „so“

Diese Erkenntnisse können programmatisch umgewandelt werden und somit der Rohdatensatz um ein vielfaches reduziert werden. Somit braucht es zunächst kein händisches Labeln der Daten, da dieser Prozess völlig automatisch erfolgt. Der Nachteil dieses Verfahrens ist, dass die Daten falsche Positive haben können und evtl. nicht Präzise genug sind. Im folgenden werden die Ansätze die oben beschrieben wurden in Python-Code transformiert. Die Funktion aus Listing 6.2 kann z.B. dafür verwendet werden, um den Datensatz nach bestimmten Kriterien wie „Beinhaltet die Schlagzeile ein Fragezeichen oder eine Zahl“ zu filtern. Ebenfalls kann die Wortlänge durch die Funktion aus Listing 6.3 ermittelt werden. Die Anwendung der Funktion aus Listing 6.4 soll dazu die Titel zu filtern, die bestimmte Wörter enthalten. Diese Wörter wurden in der Analyse aus 6.2 erkannt und sind bekannt dafür, dass sie in Clickbaits oft vorkommen.

Mit der Funktion aus Listing 6.4 können bestimmte Tags ermittelt und der Titel nach dem Vorhandensein dieser Tags gefiltert werden.

```

1 def contains_word(word, row):
2     for r in row:
3         if r in word:
4             return 1
5
6
7 def label_data_with_arg(df, col_name, arg_):
8     return df[col_name].apply(
9         lambda x: contains_word(arg_, re.findall(r"[\w']+|[.,!?;]", x.lower()
10 )))
```

Listing 6.2: Die Funktion labelt nach bestimmten Kriterien den Datensatz

```

1 import re
2 import string
3
4
5 def remove_punc(text):
6     text = re.sub("\[.*?\]", "", text)
7     text = re.sub("https?:\/\/\S+|www\.\S+", "", text)
8     text = re.sub("<.*?>+", "", text)
9     text = re.sub("[%s]" % re.escape(string.punctuation), "", text)
10    text = re.sub("\n", "", text)
11    text = re.sub("\w*\d\w*", "", text)
12    return text
```

```

13
14 def get_avg_length(string):
15     words = remove_punc(string).split()
16     try:
17         count = int(sum(len(word) for word in words) / len(words))
18     except ZeroDivisionError:
19         count = 1
20     return count

```

Listing 6.3: Funktionen die die durchschnittliche Wortlänge erkennen und nach Interpunktionsfiltern

```

1
2 just_clickbait_all_df["has_keyword"] = label_data_with_arg(
3     just_clickbait_all_df, "title", ["diese", "besten", "krassen", "neue", "darum", "dinge", "mehr", "alle", "so", "quiz"])

```

Listing 6.4: Tagger Funktion

## 6.4 Analyse der Daten

Die Summe der Wikinews Nachrichten beträgt ca. 10.000. Mit der Zugabe weiterer 10.000 Clickbaits, die durch das Labeln entstehen, ergibt sich ein Datensatz mit 20.000 Beispielen. Die Tabelle 6.3 verschafft einen Überblick über alle Daten. Interessant ist dabei, dass bei den Clickbaits ca. 33% aller Clickbaits ein Fragezeichen oder eine Zahl beinhalten und ca. die Hälfte aller Daten im Datensatz ein bestimmtes Tag wie „diese“ oder „so“ enthalten. Bei den Wikinews Nachrichten liegen diese Anteile sehr weit unten. Die durchschnittliche Wortlänge beträgt bei den Clickbaits bei 5, während bei den Wikipedia Nachrichten dieses bei 7 liegt. Bei der Betrachtung der Wörter mit der Word-Cloud Analyse können bestimmte Themen erkannt werden. Auffällig bei den Clickbaits sind neben „Promis“ auch die Wörter „darum“, „quiz“, „sieht“ und „macht“. Bei den Wikipedia Nachrichten geht es mehr um Deutschland und der Welt.

Tabelle 6.3: Beschreibung des gelabelten Datensatzes

	<b>has_question</b>	<b>has_keyword</b>	<b>has_number</b>	<b>avg_word_length</b>	<b>label</b>
<b>count</b>	20000	20000	20000	20000	20000
<b>mean</b>	0.1782	0.3190	0.0854	6.1307	0.5000
<b>std</b>	0.3826	0.4661	0.2795	1.7874	0.5000
<b>min</b>	0	0	0	1.0000	0
<b>max</b>	1	1	1	27.0000	1



Abbildung 6.2: Die Darstellung zeigt das Aufkommen der Häufigsten Wörter aus den Titeln der Clickbaits Nachrichten



Abbildung 6.3: Die Darstellung zeigt das Aufkommen der Häufigsten Wörter aus den Titeln der Wikinews Nachrichten

# 7 Das Deep Learning Modell

## 7.1 Analyse der Technologie

*TensorFlow* ist eine Bibliothek, mit der Deep Learning in Python durchgeführt werden kann. Es wurde im Jahr 2015 von Google Open Source gemacht. Mit TensorFlow werden die Daten als „Tensoren“ dargestellt und fließen durch Schichten. Dieses ermöglicht die Inferenz und das Training in maschinellen Lernmodellen. Ein *Tensor* ist ein multidimensionales Array. In neuronalen Netzen und beim Deep Learning wird jedes Datenelement und jedes Berechnungsergebnis als Tensor dargestellt, z.B. kann ein Graustufenbild als 2-dimensionales Array dargestellt werden oder ein Farbbild als 3-dimensionales Array. Der Tensor kann unterschiedliche Datentypen haben (z. B. float32 oder int32). Neben dem Typen eines Tensors gibt es die zweite Eigenschaft, die Form eines Tensors. Die Form eines Tensors gibt die Größe des Tensors entlang aller seiner Abmessungen an. Beispiel: Ein 2-dimensionales Tensor hat die Form (128, 256). Ein Tensor kann unabhängig von den ursprünglichen Daten in ein sogenanntes Layer eingespeist werden, welches nur danach schaut, welcher Datentyp und Form der Tensor hat. Tensoren sind also eine Art Container die Daten organisieren und dafür verwendet werden können, dass diese parallel verarbeitet werden können [13, 27].

```
1 import tensorflow as tf
2
3
4 rank_2_tensor = tf.constant([[1, 2],
5                             [3, 4],
6                             [5, 6]], dtype=tf.float16)
```

Listing 7.1: Beispiel eines Tensors in TensorFlow

Um den zweiten Ausdruck in „TensorFlow“ zu verstehen, muss der Tensor als eine Art „Flüssigkeit“ vorgestellt werden, welches die Daten trägt. In TensorFlow fließt es durch ein Diagramm - eine Datenstruktur, die aus miteinander verbundenen mathematischen Operationen (Knoten genannt) bestehen. Wie Abbildung 7.1 zeigt, kann der Knoten aufeinanderfolgende Schichten in einem neuronalen Netzwerk sein. Jedes der Knoten nimmt Tensoren als Eingabe und erzeugt Tensoren als Ausgabe. Die „Flüssigkeit“ wird in

verschiedene Formen und Werte umgewandelt, wenn sie durch das TensorFlow-Diagramm „fließt“. Dies entspricht der Transformation, d.h. dem Kern dessen, was neuronale Netze tun. Mit TensorFlow können Ingenieure für maschinelles Lernen, alle Arten von neuronalen Netzen schreiben, von flachen bis zu sehr tiefen Netzen [13, 27-28].

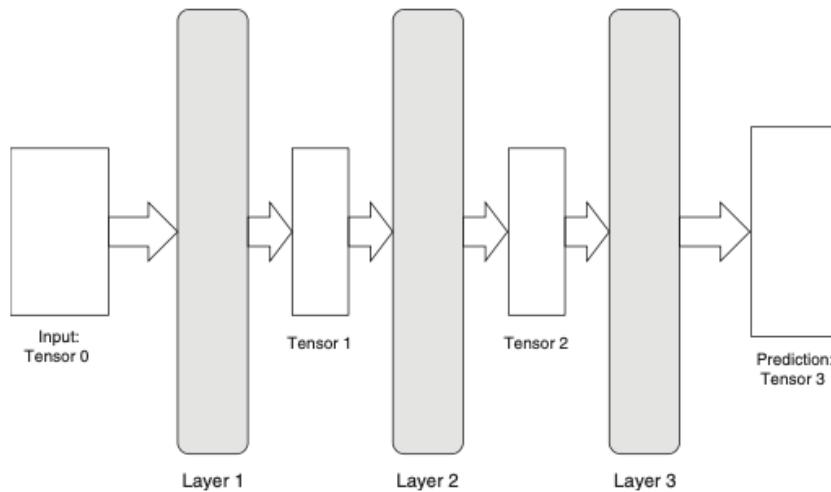


Abbildung 7.1: Die Darstellung eines Deep Learning Modells in TensorFlow - Die Tensoren fließen durch jede Schicht im Modell durch. Abbildung aus [13, 28].

Im Kern wurde TensorFlow sehr allgemein und flexibel konzipiert: Die Operationen können beliebige genau definierte mathematische Funktionen sein, nicht nur Schichten neuronaler Netze. Dies können beispielsweise mathematische Operationen auf niedriger Ebene sein, beispielsweise das Addieren und Multiplizieren von zwei Tensoren - die Art von Operationen, die innerhalb einer neuronalen Netzwerkschicht stattfinden. Dies gibt Deep-Learning-Ingenieuren und Forschern die Möglichkeit, beliebige und neuartige Operationen für Deep-Learning zu definieren. Für einen großen Teil der Deep-Learning-Praktiker ist die Manipulation solcher Operationen auf niedriger Ebene jedoch schwieriger als es sich lohnt. Dies führt zu aufgeblähtem und fehleranfälligerem Code und längeren Entwicklungszyklen. Die meisten Deep-Learning-Ingenieure verwenden eine Handvoll fester Schichttypen (z. B. Faltung, Pooling oder Dichte Schichten). In seltenen Fällen müssen neue Ebenentypen erstellt werden. Hier ist die „LEGO-Analogie“ angebracht. Bei LEGOs gibt es nur wenige Blocktypen. „LEGO-Entwickler“ müssen nicht darüber nachdenken, was nötig ist, um einen LEGO Block zu erstellen [13, 28].

In der Welt von TensorFlow ist das LEGO-Äquivalent die High-Level-API namens *Keras*. Keras bietet eine Reihe der am häufigsten verwendeten Arten von neuronalen Netzwerkschichten mit jeweils konfigurierbaren Parametern. Außerdem können Benutzer

die Schichten miteinander verbinden, um neuronale Netze zu bilden. Darüber hinaus enthält Keras auch APIs für:

- Festlegen, wie das neuronale Netzwerk trainiert werden soll (Verlustfunktionen, Metriken und Optimierer)
- Daten einspeisen, um das neuronale Netzwerk zu trainieren oder auszuwerten oder das Modell zur Inferenz zu verwenden
- Überwachung des laufenden Trainingsprozesses
- Modelle speichern und laden
- Plotten der Architektur und von Modellen [13, 29].

Es gab jedoch ein Problem: Es war nicht möglich, TensorFlow- oder Keras-Modelle in JavaScript oder direkt im Webbrowser auszuführen. Um Deep-Learning-Modelle im Browser bereitzustellen, musste dies über HTTP-Requests an einen Backend-Server getan werden. *TensorFlow.js* löst dieses Problem. Die JavaScript-API hat eine Keras-ähnliche High-Level-API welches auf dem Low-Level-Kern erstellt wurde, die es Benutzern erheblich erleichtert, Deep-Learning-Modelle in der JavaScript-Bibliothek zu definieren, zu trainieren und auszuführen. Um die Interoperabilität weiter zu verbessern, wurden Konverter erstellt, mit denen TensorFlow.js aus TensorFlow und Keras gespeicherte Modelle importieren und Modelle in diese exportieren kann [13, 29].

*Google Colab* ist ein kostenloser Cloud-Dienst auf Basis von Jupyter Notebooks, der GPU unterstützt. Dies ist nicht nur ein Tool zur Verbesserung der Codierungsfähigkeiten, sondern ermöglicht es auch absolut jedem, Deep-Learning-Anwendungen mit gängigen Bibliotheken wie PyTorch, TensorFlow, Keras und OpenCV zu entwickeln. Dabei wird die Hardware von Google über Cloud-Computer frei angeboten, sodass diese Kapazitäten zum Training von Deep Learning Modellen verwendet werden können.

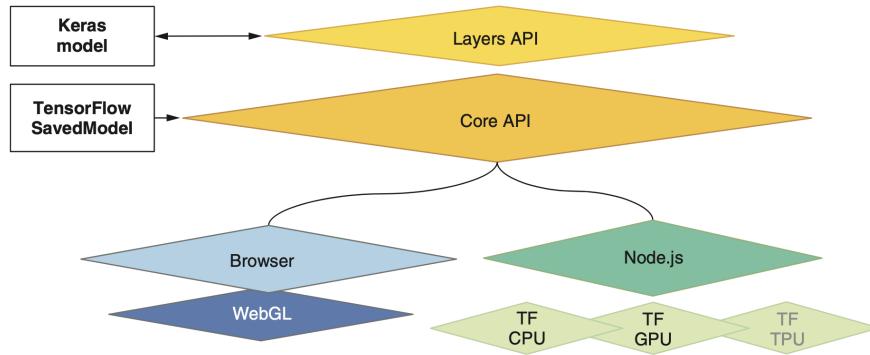


Abbildung 7.2: Die Architektur von TensorFlow.js - Abbildung zeigt die Verhältnisse zwischen der Python API von TensorFlow und Keras mit TensorFlow.js. Entnommen aus [13, 30]

## 7.2 Vorverarbeitung

Beim Vorverarbeitung (Preprocessing) der Daten sollte der Text zunächst in Einbettungen (Vektoren) umgewandelt werden, sodass ein Training erfolgen kann. Bei der Erstellung des Datensatzes wurden die Titel weitestgehend „gesäubert“. Bei einigen Schlagzeilen waren neben dem Titel auch Namen der Seite vorhanden, diese wurden entfernt. Jedoch sollte der Text, wenn noch nicht erfolgt, in Kleinbuchstaben umgewandelt werden. Die Leerzeichen nach und vor dem jeweiligen Satzzeichen sollten getrennt werden. Da bei Clickbaits Satzzeichen wie Fragezeichen oder Ausrufezeichen eine wichtige Bedeutung haben, werden diese nicht vollständig entfernt und in den Vokabular aufgenommen. Es werden also alle Satzzeichen außer Fragezeichen, Ausrufezeichen und Punkt entfernt. Stopwörter bieten normalerweise keine große Relevanz, sind hier jedoch auch bedeutsam, z.B. Präpositionen. Stopwörter werden nicht entfernt.

Beim Preprocessing wird der Datensatz in zwei Teile geschnitten, dem Trainingssatz und des Testsatz. Es ist üblich dieses Teilung in einer Ratio von 0.8/0.2 auszuführen. Das Ziel ist es vorherzusagen, ob eine bestimmte Schlagzeile Clickbait ist oder nicht. Das Modell muss also später vorhersagen mit neuen Texten treffen. Um den Erfolg des Modells messen zu können wird ein Teil des Datensatzes also getrennt von dem Trainingsdatensatz. Dieser kleinere Teil dient zur Überprüfung des Modells nach dem Training. Diese „neue“ Daten werden dann dem Modell nach dem Trainings gezeigt, um zu sehen wie gut es performt. Aus diesem Grund ist beim Preprocessing der Datensatz in Trainingssatz und Testsatz aufzuteilen. In der Funktion aus Listing 7.2 wird der Datensatz aus einer CSV-Datei gelesen und mittels Pandas in ein Dataframe umgewandelt. Es werden zufällig

80% der Titel „maskiert“ und entsprechend aufgeteilt und als Tupel zurückgegeben. Es entstehen somit ca. 16.000 Beispiele für das Training und 4.000 Beispiele für das Testen.

```

1 import pandas as pd
2 import numpy as np
3
4
5 def create_train_test(csv_file_path):
6     dataframe_ = pd.read_csv(csv_file_path).drop_duplicates()
7     msk = np.random.rand(len(dataframe_)) < 0.8
8     train = dataframe_[msk]
9     test = dataframe_[~msk]
10    train.reset_index(drop=True)
11    test.reset_index(drop=True)
12    return train, test;

```

Listing 7.2: Funktion für das Aufteilen des Datensatzes

Im Listing 7.3 werden die Buchstaben in Kleinbuchstaben umgewandelt. Mit dem regulären Ausdruck wird jedes Wort in Tokens umgewandelt. Um die Fragezeichen und Ausrufezeichen herum werden Leerzeichen gesetzt, um diese vom jeweiligen Wort zu trennen und auch als Token zu erhalten. Zahlen werden behalten.

```

1 import re
2
3
4 def text_cleaner(text):
5     newtext = re.sub(r"[^A-Za-z?!0-9üäöß]+", " ", text)
6     newtext = re.sub("([!?])", r" \1 ", newtext)
7     newtext = re.sub("\s{2,}", " ", newtext)
8     return newtext.lower()
9
10
11 train["Title"] = train["Title"].apply(lambda x: text_cleaner(x))
12 test["Title"] = test["Title"].apply(lambda x: text_cleaner(x))

```

Listing 7.3: Die Preprocessing-Funktion

### 7.3 Das Erstellen eines Vokabulars

Es ist wichtig ein Vokabular bekannter Wörter zu definieren, wenn One-Hot-Encoding oder ein Worteinbettungen verwendet werden. Je mehr Wörter vorhanden sind, desto größer ist die Darstellung von Dokumenten. Daher ist es wichtig, die Wörter nur auf diejenigen zu beschränken, von denen angenommen wird, dass sie vorhersagbar sind. Dies

ist im Voraus schwierig. Es ist oft wichtig, verschiedene Hypothesen zum Aufbau eines nützlichen Vokabulars zu testen. Aus dem vorherigen Abschnitt wurde die Interpunktionsentfernung entfernt. Zahlen wurden nicht entfernt und auch Stopwörter sind dem Datensatz erhalten geblieben. Es kann also aus allen Schlagzeilen eine Reihe aller bekannten Wörter ermittelt werden. Jedes Element im Vokabular wird als Zahl dargestellt. Es handelt sich also um eine Wörterbuch Zuordnung von Wörtern und deren Index. Somit kann das Wort auf eine einfache Weise aktualisiert und abgefragt werden.

Keras bietet eine Klasse namens `Tokenizer` (siehe Listing 7.4) welches diesem Zwecke dient. Diese Klasse ermöglicht die Vektorisierung von Textkorpora, indem jedes Token entweder in eine Folge von Ganzzahlen (jede Ganzzahl ist der Index eines Tokens in einem Wörterbuch) oder in einen Vektor umgewandelt wird. Die Methode `fit_on_texts` dieser Klasse aktualisiert den internen Wortschatz anhand einer Liste. Um den Text in eine Folge von ganzen Zahlen umzuwandeln ist das Ausführen der Methode `fit_on_texts` zwingend erforderlich. Dann können mit der Methode `texts_to_sequences`, die Wörter die der Tokeniser am häufigsten gesehen hat ermittelt werden. Der nächste Schritt ist das Padding. Mit der Methode `pad_sequences` kann eine Liste von Sequenzen mit einer bestimmten Länge in ein 2-dimensionales Numpy-Array umgewandelt werden. Sequenzen die länger als die angegebene maximale Länge haben, werden abgeschnitten, damit sie der gewünschten Länge entsprechen. Sequenzen die kürzer sind wiederum, werden mit einer Null aufgefüllt, bis sie dem maximalen Wert entsprechen. Das Vorfüllen oder Entfernen von Werten am Anfang der Sequenz ist die Standardeinstellung. Im aktuellen Datensatz haben die Titel eine maximale Tokenlänge von 21. Die maximale Länge wird auf 40 Tokens gesetzt. Es wird also nachher erlaubt, eine Abfrage mit maximal 40 Tokens durchzuführen. Es ist nicht zu erwarten, dass eine Schlagzeile mehr als 40 Tokens hat. Der Tokeniser gibt somit z.B. für den Titel „die 10 besten apps für mädchen“ einen Array der Länge 40 zurück, wobei die ersten 34 Werte eine Null enthalten und die restlichen 6 eine bestimmte Zahl, welche den Index des Wortes darstellt.

```
1 from keras.preprocessing.text import Tokenizer
2 from keras.preprocessing.sequence import pad_sequences
3
4
5 def tokenize_text(train_df, test_df, max_l):
6     tokenizer = Tokenizer(filters='#$%&()*+-/:;<=@[\\\]^_`{|}~\t\n')
7     tokenizer.fit_on_texts(train_df)
8     vocab_size = len(tokenizer.word_index) + 1
9     tokenized_text_train = pad_sequences(
10         tokenizer.texts_to_sequences(train_df), maxlen=max_l)
11     tokenized_text_test = pad_sequences(
```

```

12     tokenizer.texts_to_sequences(test_df), maxlen=max_l)
13     return {"tokenized_text_train": tokenized_text_train, "
14         tokenized_text_test": tokenized_text_test, "vocab_size": vocab_size, "
15         tokenizer": tokenizer}

```

Listing 7.4: Die Tokeniser-Funktion

Eine Worteinbettung ist eine Möglichkeit, Text darzustellen, bei der jedes Wort im Vokabular durch einen reellen Vektor in einem hochdimensionalen Raum ersetzt wird. Die Vektoren werden so gelernt, dass Wörter mit ähnlichen Bedeutungen eine ähnliche Darstellung im Vektorraum haben (nahe im Vektorraum). Dies ist eine aussagekräftigere Darstellung für Text als klassische Darstellungen, bei denen Beziehungen zwischen Wörtern oder Tokens ignoriert oder in Bigram- und Trigramm-Ansätzen erzwungen werden. Die Vektordarstellung für Wörter kann während des Trainings des neuronalen Netzwerks gelernt werden. Dieses kann in der Keras Deep Learning-Bibliothek mithilfe der Einbettungsebene ausgeführt werden. Alternativ können auch vortrainierte Einbettungen wie Word2Vec verwendet werden, die auf eine große Menge an Daten vortrainiert und für neue Aufgaben verwendet werden können. In dieser Arbeit wird die erste Alternative der Worteinbettung verwendet.

Die Keras-Einbettungsschicht erfordert Ganzzahl Eingaben, bei denen jede Ganzzahl einem einzelnen Token zugeordnet ist, das eine bestimmte reelle Vektordarstellung innerhalb der Einbettung repräsentiert. Diese Vektoren sind zu Beginn des Trainings zufällig, gewinnen jedoch während des Trainings für das Netzwerk an Bedeutung. Durch die Funktionen aus Listing 7.3 und Listing 7.4 wurde das Encoding bestimmt, also der Text in ein passendes Format gebracht. Schließlich werden die Klassenbezeichnungen für den Trainingsdatensatz und Testdatensatz definieren, die für das überwachte neuronale Netzwerkmodell erforderlich sind. Die Daten werden zuletzt in ein TensorFlow dataset umgewandelt, um sie leichter in Keras einzuspeisen. Somit entsteht ein Datensatz für das Training, mit einem Wortschatz von ca. 23.000 Vokabeln und einem erwarteten maximalen Eingang von 30 Tokens. Die Daten wurden in Trainingssatz und Testsatz aufgeteilt. Mit der Funktion aus Listing 7.5 werden die Labels *Clickbait 0* und *News 1* in ein Array umgewandelt. Im Listing 7.6 wird der Datensatz in ein TensorFlow Datensatz umgewandelt. Dadurch können die Daten direkt in ein Modell eingespeist werden. In diese Funktion gehen auch die labels *y* und *y\_test* hinein.

```

1 def create_labels(train_data, test_data):
2     encoded_labels = preprocessing.LabelEncoder()
3     y = encoded_labels.fit_transform(train_data["label"])
4     y = to_categorical(y)
5     y_test = encoded_labels.transform(test["label"])

```

```

6     y_test = to_categorical(y_test)
7     return y, y_test;
8
9 y, y_test = create_labels(train, test)

```

Listing 7.5: Die Label-Funktion

```

1 import tensorflow_datasets as tfds
2
3 train_dataset = tf.data.Dataset.from_tensor_slices((tokenized_text_train, y))
4 test_dataset = tf.data.Dataset.from_tensor_slices((tokenized_text_test,
5                                                 y_test))

```

Listing 7.6: Die Dataset Erstellung

## 7.4 Die Modell Architektur

In Listing 7.7 wird ein sequentielles Modell erstellt. Diesem Modell kann durch die Methode add jeweils eine Schicht angehängt werden, und Keras arbeitet dieses nacheinander ab. Die API von Keras wird verwendet um das Modell zusammen zu bauen. Die Definitionen der einzelnen Methoden mit der das Modell gebaut und trainiert wurde, sind aus der Dokumentation von Google [4] entnommen. Dort werden viele Begriffe, die im Zusammenhang mit dem bauen und trainieren eines Deep Learning oder Machine Learning Modells stehen, erklärt. Aus diesem Grund wird in den nächsten zwei Abschnitten auf diese Quelle verwiesen.

```

1 from tensorflow.keras.models import Sequential
2 from tensorflow.keras.layers import Conv1D, MaxPool1D, Dropout, Dense,
3                                         GlobalMaxPool1D, Embedding, Activation
4
5 def build_model(vocab_size, emb_dim, max_len, dropout_rate, learning_rate,
6                 n_labels):
7     loss = tf.keras.losses.BinaryCrossentropy()
8     metric = [tf.keras.metrics.BinaryAccuracy(name="accuracy")]
9     opt = tf.keras.optimizers.Adam(learning_rate=learning_rate)
10
11    model = Sequential([])
12    model.add(
13        Embedding(vocab_size, output_dim=emb_dim, input_length=max_len))
14    model.add(Dropout(dropout_rate))
15    model.add(Conv1D(filters=32, kernel_size=8,
16                     activation="relu", padding="same", strides=1))
17    model.add(GlobalMaxPool1D())

```

```

16     model.add(Dense(16, activation="relu"))
17     model.add(Dense(n_labels, activation="softmax"))
18     model.compile(loss=loss, metrics=metric, optimizer=opt)
19     model.summary()
20     return model
21
22 model = build_model(vocab_size=vocab_size, emb_dim=32, max_len=40,
                      dropout_rate=0.3, learning_rate=0.00006, n_labels=len(labels))

```

Listing 7.7: Das Bilden des Models

Zunächst müssen einige Parameter festgelegt werden. Diese sind `loss` also die Verlustfunktion, `metric` also nach welchen Metriken das Modell bemessen werden soll und `opt` der Optimizer wodurch das Modell lernt. Da es sich um eine binäre Klassifikation handelt, ist hier die `BinaryCrossentropy` als Verlustfunktion ausgewählt worden. Selbes gilt für die `BinaryAccuracy`. Als Optimizer wurde Adam ausgewählt. Die Lernrate ist ein Skalar, mit dem ein Modell über einen Gradientenabstieg trainiert wird. Während jeder Iteration multipliziert der Gradientenabstiegsalgorithmus die Lernrate mit dem Gradienten. Das resultierende Produkt wird als Gradientenschritt bezeichnet. Die Lernrate ist ein wichtiger Hyperparameter. Ein Optimizer ist eine spezifische Implementierung des Gradientenabstiegsalgoritimus. Die Klasse von TensorFlow für Optimierer ist `tf.train.Optimizer`. Die Auswahl der Hyperparameter erfolgt durch das experimentieren und testen des Verfassers dieser Arbeit (siehe Listing 7.7 Zeile 5-7).

Nachdem die Hyperparameter popularisiert wurden, wird dem Modell die erste Schicht zugeführt. Dieses ist die Einbettungsschicht. Die Worteinbettung sind eine Möglichkeit, ein Wort als Vektor darzustellen (ein 1-dimensionaler Tensor in TensorFlow). Durch Worteinbettungen können die Werte der Elemente des Vektors trainiert werden. Es muss also keine Regel angewendet werden, wie bei der One-Hot-Codierung und dessen Wort zu Index Zuordnung. Mit anderen Worten, wenn ein textorientiertes neuronales Netzwerk die Worteinbettung verwendet, werden die Einbettungsvektoren zu trainierbaren Gewichtungsparametern des Modells. Sie werden durch dieselbe Backpropagation-Regel wie alle anderen Gewichtungsparameter des Modells aktualisiert. Es gibt auch die Möglichkeit, eine vortrainierte Einbettungsschicht zu verwenden. Diese werden auf viel größere Mengen an Daten trainiert und können für vielseitige Zwecke verwendet werden, ohne dass von Grunde aus neu trainiert werden muss. In diese Fall trainiert das Modell die Einbettungen selbst, anstatt sich auf die vorab trainierten Einbettungen zu verlassen (siehe Listing 7.7 Zeile 10-11).

Um das Modell vor Überanpassung zu schützen gibt es bestimmte Strategien, die angewendet werden können. Dieses sind sogenannte Regularisierungstrategien. Es kann

als eine Art „Strafe“ betrachtet werden, für die Komplexität des Modells und das nicht Vorhandensein an Daten. Da bei wenig Daten und hoher Komplexität das Modell nicht wirklich lernt, sondern sich den gegebenen wenigen Daten „über anpasst“ können diese Strategien dagegen steuern (siehe Listing 7.7 Zeile 12). Wenn Neuronen Muster in Trainingsdaten vorhersagen, indem sie sich fast ausschließlich auf Ausgaben bestimmter anderer Neuronen stützen, anstatt sich auf das Verhalten des Netzwerks als Ganzes zu verlassen, entsteht eine Anpassung. Wenn die Muster, die eine Anpassung verursachen, nicht in den Validierungsdaten vorhanden sind, weil es zu wenig Daten vorhanden sind etwa, führt dies zu einer Überanpassung. Die Dropout-Regularisierung reduziert die Anpassung, da Dropout sicherstellt, dass sich Neuronen nicht nur auf bestimmte andere Neuronen verlassen können. Die Dropout-Regularisierung entfernt eine zufällige Auswahl einer festen Anzahl von Einheiten in einer Netzwerkschicht für einen einzelnen Gradientenschritt. Je mehr Einheiten ausfielen, desto stärker war die Regularisierung.

Die Conv1D-Ebene (siehe Listing 7.7 Zeile 13) wird verwendet um die Faltung durchzuführen. Eine CNN ist ein neuronales Netzwerk, in dem mindestens eine Schicht die Faltungsschicht ist. Ein typisches neuronales Faltungsnetzwerk besteht aus einer Kombination der von Faltungsschichten, Poolingschichten, und vollständig verbundene Schichten. Die Faltungsschicht hier ist eine eindimensionale Faltungsschicht. Bestimmte Muster aus dem Text werden hier erkannt (z.B. ob nach einem negativen Verb ein bestimmtes Wort auftaucht. In dem Beispiel gibt es 32 Filter. Beim maschinellen Lernen werden Faltungsfilter normalerweise mit Zufallszahlen gesetzt, und dann trainiert das Netzwerk die idealen Werte. Die *kernel\_size* ist eine Zahl, die die Höhe des Faltungsfensters angibt. Zusätzlich zur eindimensionalen Faltungsschicht wird eine ebenfalls eindimensionale Poolingsschicht (siehe Listing 7.7 Zeile 15) zugeführt.

Die Dense-Schichten (siehe Listing 7.7 Zeile 16-17) sind verborgene Schichten, in der jeder Knoten mit jedem Knoten in der nachfolgenden verborgenen Schicht verbunden ist. Eine vollständig verbundene Schicht wird auch als dichte Schicht „dense layer“ bezeichnet. Das erste der beiden Schichten im Modell hat 16 Einheiten und das zweite hat genau so viele Einheiten wie es Labels gibt (dieses Modell hat genau 2, da es 2 Klassen gibt). Die letzte Schicht ist somit die Ausgabeschicht und teilt die Vorhersage mit.

Mit der `compile`-Methode lässt sich das Modell bauen und die `summary`-Methode gibt eine Übersicht über das Modell (siehe Tabelle 7.1).

Tabelle 7.1: Beschreibung der Schichten des Modells

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 40, 32)	740384
dropout (Dropout)	(None, 40, 32)	0
conv1d (Conv1D)	(None, 40, 32)	8224
global_max_pooling1d (GlobalMaxPool1D)	(None, 32)	0
dense (Dense)	(None, 16)	528
dense (Dense)	(None, 2)	34

## 7.5 Das Training

Mit der Funktion `train_model` aus Listing 7.9 lässt sich das Modell trainieren. Zunächst muss als Parameter angegeben werden, wie viele Trainingsepochen das Modell trainiert werden soll. Ein vollständiger Trainingsdurchlauf über den gesamten Datensatz, sodass jedes Beispiel einmal gesehen wurde wird als Epoche genannt. Die `batch_size` sind die Anzahl der Beispiele in einer Epoche. Vor dem Training bietet sich die Möglichkeit, die Daten zu mischen, dieses erfolgt mit dem Parameter `shuffle`. Analysen über das Training kann mit TensorBoard durchgeführt werden. TensorBoard bietet die Visualisierung und Werkzeuge, die für Experimente mit maschinellem Lernen erforderlich sind. Metriken wie Verlust und Genauigkeit können mit TensorBoard für jede Epoche dargestellt werden. Damit TensorBoard im späteren Verlauf verwendet werden kann, wird eine Callback-Funktion eingeführt, welches den Verlauf des Trainings in Logdateien speichert. Mit der `fit`-Methode erfolgt das Training (siehe Listing 7.9).

```

1 import os
2 import datetime
3
4 def train_model(num_epochs, batch_size, train_ds, test_ds, model, shuffle):
5     ds_train_encoded = train_ds.shuffle(shuffle).batch(batch_size)
6     ds_test_encoded = test_ds.batch(batch_size)
7     logdir = os.path.join(
8         "logs", datetime.datetime.now().strftime("%Y%m%d-%H%M%S"))
9     tensorboard_callback = tf.keras.callbacks.TensorBoard(
10         logdir, histogram_freq=1)
11     model.fit(ds_train_encoded, epochs=num_epochs,
12               validation_data=ds_test_encoded, callbacks=[
13                 tensorboard_callback])

```

```
14 train_model(num_epochs=7, batch_size=32, train_ds=train_dataset, test_ds=
    test_dataset, model=model, shuffle=1000)
```

Listing 7.8: Das Training des Models

## 7.6 Evaluation

Das Kreuzvalidierungsverfahren ist ein Mechanismus zum Testen, dafür um zu sehen, wie gut sich ein Modell auf neue Daten verallgemeinern lässt. Das Modell wird mit neuen Daten, welche dem Trainingsdatensatz zurückgehalten wurden, getestet. Mit der Bibliothek „sklearn“ können mit der Methode `classification_report` ein Testbericht mit den wichtigsten Klassifizierungsmetriken erstellt werden. Dieser Report gibt Auskunft über bestimmte Metriken, mit der das Performance des Modells auf den vorhandenen Daten gemessen werden kann.

```
1 import numpy as np
2 from keras.utils import to_categorical
3 from sklearn.metrics import classification_report
4
5 y_pred = to_categorical(np.argmax(
6     model.predict(tokenized_text_test), axis=1))
7
8 print(classification_report(y_test, y_pred, target_names=labels.values(),
    digits=4))
```

Listing 7.9: Das Evaluieren des Models

Tabelle 7.2: Die Ergebnisse der Evaluation des Modells

	precision	recall	f1-score	support
<b>Clickbaits</b>	0.9775	0.9589	0.9681	1993
<b>News</b>	0.9590	0.9776	0.9682	1961

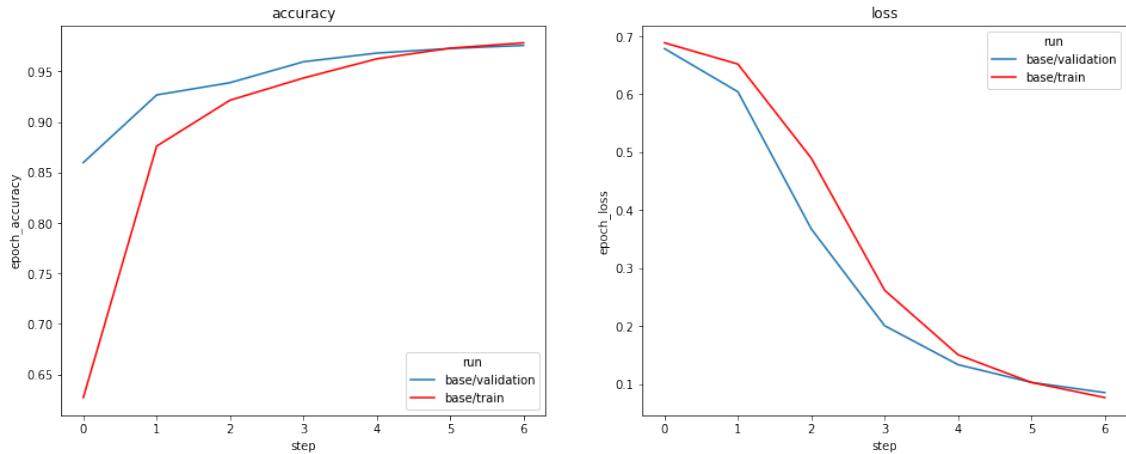


Abbildung 7.3: Blau: Validierungsdaten. Rot: Trainingsdaten. Links: Die Genauigkeit nimmt mit jeder Epoche zu und beide Kurven nähern sich an bis sie sich in der 7. Epoche treffen. Rechts: Die Verlustfunktionen beider Daten werden dargestellt. Die Graphen treffen sich erst in der 5. Epoche. Hier liegt kein optimaler „Fit“ vor, da beide Graphen meistens auseinander liegen.

Um zu das Modell auch praktisch zu testen und zu sehen wie es in der Realität performt sollen dem Modell neue Anfragen gesendet werden. Nachdem das Modell diese Anfragen beantwortet, kann gesehen werden wo es gut abschneidet und wo nicht und potenzielle Schwachstellen festgelegt werden. Natürlich wird das Modell in Wirklichkeit nicht so gut performen wie in der Tabelle 7.2. Die Realität bietet einen viel größeren „Datensatz“ und viel komplexere Anfragen. Es ist unmöglich, ein Modell zu entwickeln, welches auf alle Gegebenheiten trainiert wird. Zum Abschluss dieses Abschnittes sollen einige „Schwachstellen“ dieses Modells mittels praktischer Anwendung dargestellt werden. Es ist zu erwarten, dass das Modell „offensichtliche Clickbaits“ gut erkennt, jedoch ist zu beantworten, wie es sich „mehrdeutigen Fällen“ verhält.

In der Tabelle 7.3 sind die Ergebnisse des Experiments zu sehen. Es lassen sich bestimmte Muster erkennen. Offensichtliche Clickbaits und Nachrichten (die ersten 8 Schlagzeilen) werden erkannt. In diesen 8 Schlagzeilen haben Clickbaits die Eigenschaft, eine Zahl zu beinhalten und relativ kurze Wortlängen zu haben. Außerdem ist deutlich zu sehen, dass bei den Nicht-Clickbaits Nachrichten, es sich um Internationale Themen handelt und lange Wörter wie „Kanzlerkandidat“ oder „Arzneimittelwerbung“ vorhanden sind.

Das Modell kann aber auch falsch Alarm schlagen. Wie im Beispiel „Corona im News-Ticker: RKI meldet über 2 Mio. Infektionen - droht Mega-Lockdown?“, zwar ist hier ein

Tabelle 7.3: Vorhersagen des Modells für neue Schlagzeilen

Schlagzeile	Vorhersage
Die 10 lustigsten Bilder von Katzen!	Clickbait
Diese Rezepte solltest du nicht verpassen!	Clickbait
Aus diesem Grund sollten Sie ihr Smartphone während des Schlafens ausschalten	Clickbait
13 Dinge, die Sie garantiert noch nicht über ihr Smartphone wussten	Clickbait
Sondersitzung des Kabinetts - Bayern ruft erneut Katastrophenfall aus	News
Russland verschärft Ton im Pipeline-Streit	News
Nordkorea präsentiert offenbar modernisierte Rakettentypen	News
„Wandelnde Arzneimittelwerbung“ als Kanzlerkandidat?	News
Corona im News-Ticker: RKI meldet über 2 Mio. Infektionen - droht Mega-Lockdown?	Clickbait
Die 7 besten Tageslinsen 2021 im Vergleich	Clickbait
Er hatte alles – nur ein Erbe fehlte ihm: Zum Tod von Modegenie Pierre Cardin	Clickbait
„Pass mal auf“: Lanz fragt nach Tempolimit 130, Merz knallt ihm einen vor den Latz	News
Trump ist ein Politverbrecher wie Putin oder Erdogan	Clickbait
Wie Kinder die Corona-Pandemie beeinflussen	Clickbait
Was steht in den Impfstoff-Verträgen?	Clickbait

Wort „Mega“ welches ein oft in Clickbaits vorkommendes Wort ist und auch eine Zahl „2“ enthalten, ebenfalls wird eine Frage gestellt. Das Modell kann aber nicht erkennen, bzw. weiß nicht, dass diese Mittel (das Stellen einer Frage, verwenden von Zahlen usw.) auch in konventionellen Nachrichten verwendet werden. Hier wird deutlich, dass alleine die Klassifizierung des Titels der Nachricht, auch zu falschen Ergebnissen führen kann. Es sollte ebenfalls der Inhalt der Nachricht überprüft und aus dem Zusammenhang beider Ergebnisse eine Vorhersage gemacht werden.

## 7.7 Experimente

Das Modell aus der Tabelle 7.1 erzeugt keinen perfekten Fit. Durch Experimente soll versucht werden dieses Problem zu beheben. Lässt sich durch die Änderung einiger Hyperparameter, dieses Problem beheben? In diesem Abschnitt werden einige Experimente mit den Hyperparametern gemacht und das Modell zum Ursprünglichen Modell aus Tabelle 7.1 verglichen. Wie im Abschnitt 2.10 beschrieben ist die Über- bzw. Unteranpassung zu vermeiden. Zwar gibt es Strategien wie im Abschnitt 2.11 definiert, aber auch die Anpassung der Hyperparameter wie die Lernrate können Auswirkungen auf die Über- oder Unteranpassung haben. Die Experimente sollen untersuchen, ob und wie sich das Modell ändert. Das Modell aus Tabelle 7.1 hat 32 Einbettungsdimensionen. Die Lernrate beträgt 0.00006 und sie hat eine Dropout-Rate von 0.35. Ziel ist es zu sehen, ob das

Modell besser performt oder nicht und dabei nicht eine Überanpassung entsteht.

**Die Veränderung der Einbettungsdimension** Die Einbettungsdimension ist der Parameter, der über die Dimensionalität der Einbettung entscheidet. Je größer diese Dimension ist, desto mehr Parameter sollten entstehen, da das Modell diese Einbettungen lernen muss. Somit würde vom Volumen her ein größeres Modell entstehen. Dieser Parameter wird von 32 auf 320 gesetzt. Alle anderen Parameter werden nicht verändert. An der Performance ändert sich wenig (siehe Tabelle 7.4) jedoch entsteht eine Überanpassung ab der 3. Epoche des Trainings (siehe Abbildung 7.4).

Tabelle 7.4: Die Ergebnisse des Modells mit 320 Einbettungsdimensionen

	precision	recall	f1-score	support
<b>Clickbaits</b>	0.9761	0.9867	0.9814	1948
<b>News</b>	0.9869	0.9766	0.9817	2009

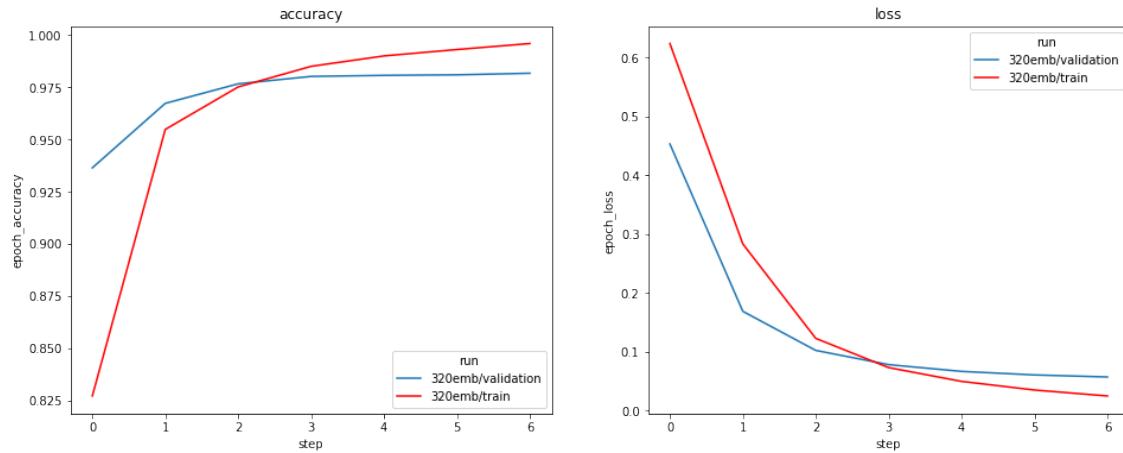


Abbildung 7.4: Ab der 3. Epoche findet eine Überanpassung statt.

**Die Veränderung der Der Lernrate von 0.00006 auf 0.0006** Im Abschnitt 2.9 wurde die Bedeutung der Lernrate im Deep Learning vorgestellt. Durch die verzehnfachung der Lernrate (von 0.00006 auf 0.0006) lernt das Modell mit einer höheren Rate und es ist in der Abbildung 7.5 deutlich zu sehen, dass eine Überanpassung herrscht. Das Modell hat bereits nach der ersten Epoche das meiste gelernt.

Tabelle 7.5: Die Ergebnisse der Evaluation des Modells mit einer Lernrate von 0.0006

	<b>precision</b>	<b>recall</b>	<b>f1-score</b>	<b>support</b>
<b>Clickbaits</b>	0.9778	0.9933	0.9855	1948
<b>News</b>	0.9934	0.9781	0.9857	2009

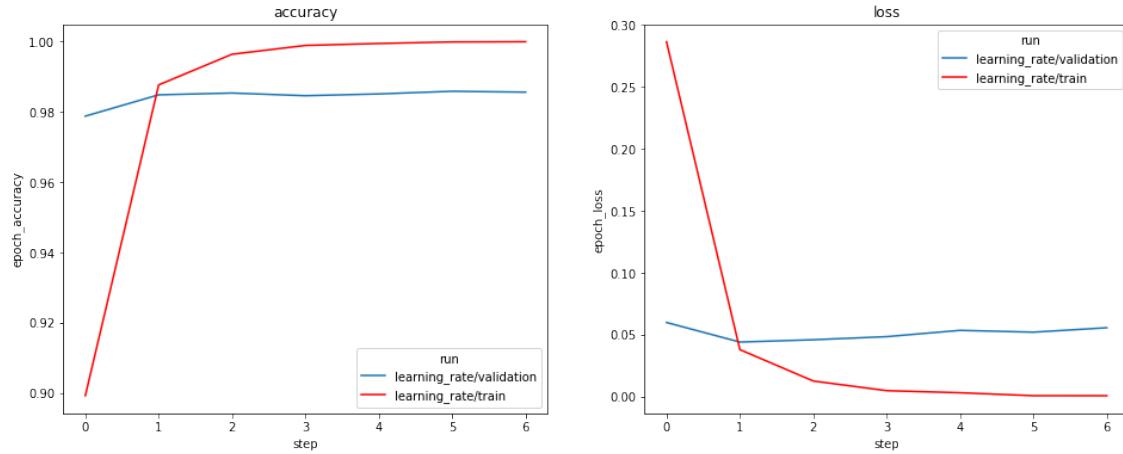


Abbildung 7.5: Es ist zu sehen, wie das Modell in der ersten Epoche das meiste gelernt hat und sich danach nur noch dem Modell anpasst.

**Das Hinzufügen von zusätzlichen Schichten** Dem Modell aus Tabelle 7.1 werden zusätzliche Conv1D-Schichten hinzugefügt. Zusätzlich werden dem Modell 256 statt 32 Einbettungsdimensionen hinzugefügt. Die Dropout-Rate wird auf 0.1 runter gesetzt, da dieses Modell mehr Schichten dieser Art besitzt. Die Lernrate bleibt unverändert. Die Schichten werden nach der Beschreibung aus der Tabelle 7.6 angepasst. Auch hier findet ein „schnelles Lernen“ statt, wie in Abbildung 7.6 dargestellt. Es findet auch hier eine Überanpassung statt. Das Modell hat zusätzliche Faltungsschichten (siehe Tabelle 7.6) und auch mehr Parameter zum trainieren. Es wurden außerdem 2 zusätzliche Dropout-Schichten hinzugefügt.

Tabelle 7.6: Das „komplexere“ Modell

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 40, 256)	6000640
dropout (Dropout)	(None, 40, 256)	0
conv1d (Conv1D)	(None, 40, 50)	38450
max_pooling (MaxPooling1D)	(None, 20, 50)	0
dropout (Dropout)	(None, 20, 50)	0
conv1d (Conv1D)	(None, 20, 100)	15100
max_pooling (MaxPooling1D)	(None, 10, 100)	0
dropout (Dropout)	(None, 10, 100)	0
conv1d (Conv1D)	(None, 10, 200)	60200
global_max_pooling1d (GlobalMaxPool1D)	(None, 200)	0
dropout (Dropout)	(None, 200)	0
dense (Dense)	(None, 100)	20100
dense (Dense)	(None, 2)	202

Tabelle 7.7: Die Ergebnisse des Modells mit zusätzlichen Schichten

	precision	recall	f1-score	support
<b>Clickbaits</b>	0.9722	0.9887	0.9804	1948
<b>News</b>	0.9889	0.9726	0.9807	2009

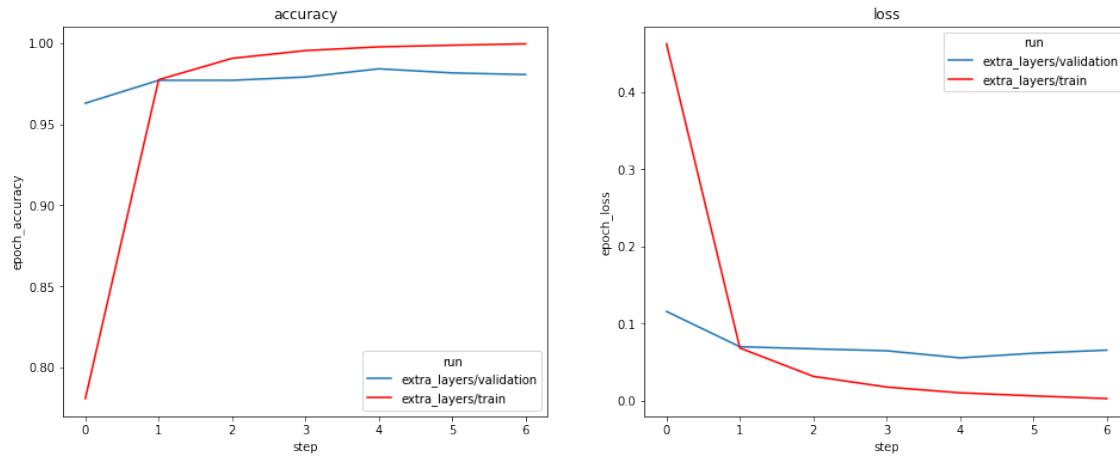


Abbildung 7.6: Das Modell „lernt schnell“ und passt sich früh an die Daten an.

## 7.8 Speichern und Konvertieren

Das Keras Modell wird als h5-Datei gespeichert. Diese Datei kann mit dem TensorFlow.js Konverter, welches Keras Modelle in für TensorFlow.js geeignete Modelle umwandelt, konvertiert werden. Das Ergebnis dieser Umwandlung ist, eine JavaScript Object Notation (JSON)-Datei `model.json`, welches das Modell beschreibt. Dazu kommen Binäre Dateien, welches die Gewichte des Modells sind. Zusammen können Sie für Inferenz im Web mit TensorFlow.js benutzt werden. Neben dem Modell werden alle Tokens mit ihrem Index als JSON gespeichert `vocab.json` und `tokenizer.json`. Die `vocab.json` und `tokenizer.json` sind Tabellen, die jedem Token aus dem Wortschatz einen entsprechenden Index geben. Diese beide Dateien sind notwendig um mit dem Modell zu kommunizieren. Da dieses Modells keine String aufnehmen kann, wird durch diese Dateien „synthetisch“ ein Vektor aus den Strings erstellt und diese werden erst dem Modell eingeführt.

# 8 Deep Learning Modelle im Web

Das Modell muss in einer Art und Weise für den Nutzer (Client) zugänglich gemacht werden. Das Modell welches in dieser Arbeit entwickelt wurde, kann nur Zahlen verstehen. Es muss also in erster Linie ein Encoder die Strings in Zahlen umwandeln. Je nach Plattform findet erst dann die Berechnung der Vorhersage statt. Im ersten Teil dieses Kapitels wird eine klassische „serverseitige“ Implementierung vorgenommen. Diese dient zu Demonstrationszweck und benötigt keine UI. Es wird lediglich ein HTTP-Endpunkt programmiert, welches eine JSON-Anfrage in ein Modell weitergibt und dann die entsprechende Antwort wieder als Antwort zurück gibt. Im Anschluss darauf wird die in dieser Arbeit vorgeschlagene Methode des clientseitigen Deep Learnings vorgestellt. Beide Methoden haben Ihre Vor- und Nachteile, diese werden anschließend analysiert.

## 8.1 Serverseitiges Deep Learning

Um das Deep Learning auf der Seite des Servers auszuführen wird ein Server geschrieben. Auf diesem Server läuft das Modell und wartet auf die Anfragen, die auf den Endpunkt /predict zukommen. Ein solcher Server kann mittels der Bibliothek Flask in Python programmiert werden. Der Server hat nur eine Route, diese Route wartet auf einen JSON-POST-Request. Nachdem der Server diese JSON-Information aufgenommen hat, kann er den Text in das Modell weitergeben. Da das Modell nur Zahlen versteht, ist das Encoden der Texte erforderlich. Die Methoden die hier verwendet werden sind sehr ähnlich zu den Methoden, die beim Trainieren verwendet wurden. Aus diesem Grund wird nicht näher auf diese eingegangen (vgl. Listing 8.1 mit Abschnitt 7.3).

```
1 from flask import Flask, request
2 from keras.preprocessing.text import tokenizer_from_json
3 from keras.preprocessing.sequence import pad_sequences
4 from keras.models import load_model
5 from flask import request
6 import numpy as np
7 import json
8
9
```

```

10 app = Flask(__name__)
11 max_len = 40
12 labels = np.array(["Clickbait", "News"])
13 model = load_model("model/model.h5")
14 with open("model/tokenizer.json") as f:
15     tokenizer = tokenizer_from_json(json.load(f))
16
17
18 def encode(text):
19     return pad_sequences(tokenizer.texts_to_sequences(text), maxlen=40)
20
21
22 @app.route("/predict", methods=["POST"])
23 def postJsonHandler():
24     if request.method == "POST":
25         return json.dumps({"prediction": labels[np.argmax(model.predict(
26             encode([request.get_json()["text"]])))]})
27
28 if __name__ == "__main__":
29     app.run()

```

Listing 8.1: Beispiel eines Servers für die Vorhersage von Clickbaits

## 8.2 Clientseitiges Deep Learning

Dieser Abschnitt beschäftigt sich mit dem clientseitigen Deep Learning. Um TensorFlow.js praktisch anzuwenden wurde in React<sup>1</sup> ein Frontend entwickelt, welches vom Modell Gebrauch macht. Für eine Inferenz ist ein Server nicht mehr nötig. Alleine der Zugriff auf die model.json Datei und die jeweiligen Gewichte reichen für diese Variante aus. Die Dateien werden auf einem Server von AWS<sup>2</sup> gehostet und können in den Browser geladen werden. Das Frontend wird außerdem die gesamte TensorFlow.js Bibliothek in den Browser laden. Da die gesamte Inferenz im Browser stattfindet, muss auf die Größe des Modells geachtet werden. Das in dieser Arbeit entwickelte Modell ist mit seiner gesamten Größe von ca. 3 MB relativ klein. Die Ordnerstruktur aus Listing 8.2 ist ein Ausschnitt der Frontend Ordnerstruktur und soll in den nächsten Abschnitten beschrieben werden. Somit kann ein Verständnis über die Zusammenhänge zwischen

---

<sup>1</sup>React ist ein Frontend Framework für JavaScript. Es wurde von Facebook entwickelt und ist Open Source.

<sup>2</sup>Amazon Web Services (AWS) ist eine Cloud-Service-Plattform, die Rechenleistung, Datenbankspeicher, Bereitstellung von Inhalten und andere Funktionen bietet.

Deep Learning und Webanwendung erläutert werden.

```

1 |____src
2 | |____constants
3 | | |____constants.js
4 | |____hooks
5 | | |____index.js
6 | | |____useLoadVocab.js
7 | | |____useLoadModel.js
8 | |____helpers
9 | | |____splitText.js
10 | | |____index.js
11 | | |____tokenize.js
12 | |____index.js
13 | |____App.js

```

Listing 8.2: Wichtigste Elemente der Benutzeroberfläche

### 8.3 Tokenisierung und Padding

Unter dem Verzeichnis „helpers“ befinden sich die Skripte zur Erstellung der Tokens. Der von dem User eingegebene Text muss in seine Tokens zerlegt werden und außerdem muss die Interpunktionszeichen gefiltert werden (außer Ausrufezeichen und Fragezeichen). Ein Wort darf außerdem nicht weniger als ein Zeichen enthalten und außerdem müssen die Wörter in Kleinbuchstaben umgewandelt werden, da die vocab.json nur solche enthält. Im Listing 8.3 wird die Funktion dargestellt, welche den Satz, den der Nutzer eingibt in Token umwandelt. Hier wird ein vorprogrammierter Tokeniser<sup>3</sup> verwendet, welches die Tokens auch taggen kann (Zahl, Wort usw.). Die Tokens werden entsprechend gefiltert und als Array zurückgegeben. Das Ergebnis dieser Prozedur muss der Prozedur aus der Python Umgebung gleichen und möglichst ähnliche Ergebnisse zurück geben, sonst wird das Modell in der Praxis etwas anderes zugeteilt bekommen als vorgesehen.

```

1 const tokenizer = require('wink-tokenizer');
2 const myTokenizer = tokenizer();
3
4 const splitText = (sentence) => {
5   const tokens = myTokenizer.tokenize(sentence);
6   const splitted = [];
7   tokens.forEach((token) => {
8     if (token.tag === 'number') {
9       splitted.push(token.value);

```

---

<sup>3</sup><https://www.npmjs.com/package/wink-tokenizer>

```

10     } else if (token.tag === 'word') {
11         if (token.value.length >= 1) {
12             splitted.push(token.value.toLowerCase());
13         }
14     } else if (token.tag === 'punctuation') {
15         if (token.value === '?' || token.value === '!') {
16             splitted.push(token.value);
17         }
18     }
19 });
20
21 return { splitted: splitted, splittedLen: splitted.length };
22 };
23
24 export default splitText;

```

Listing 8.3: Die splitText Funktion

Im nächsten Schritt müssen die Tokens in Zahlen umgewandelt werden. In der Python Umgebung wurde dies mit der Methode `pad_sequences` aus der Keras-Bibliothek ausgeführt. In JavaScript gibt es keinen Ersatz für diese Methode, sodass es selbst programmiert wird. Die `tokenize`-Funktion nimmt den Text den der Client eingibt und führt diese in die `splitText`-Funktion ein. Für alle Tokens wird ein Abgleich mit der `vocab.json` erstellt und wenn ein Treffer gefunden wurde, der Index in eine Array eingeführt. Die maximale Länge wird später aus der Datei `constants.js` entnommen und beträgt in diesem Beispiel 40. Damit entsteht ein Array der Länge 40, mit  $n$  Treffern und  $40-n$  Nullen. Der Parameter `vocabLoading` ist ein Boolean, welches aussagt, ob die `vocab.json` aktuell geladen wird oder nicht. Diese Datei (genauso wie die `model.json` und die Gewichte) werden in asynchron geladen. Dieses Laden kann in JavaScript den gesamten Prozess blockieren. Um dieses zu vermeiden werden sogenannte „Promises“ verwendet. Das Promise-Objekt repräsentiert in JavaScript den eventuellen Abschluss (oder Fehler) einer asynchronen Operation und den daraus resultierenden Wert.

```

1 import splitText from './splitText';
2
3 const tokenize = (text, vocabLoading, vocab, maxLen) => {
4     const { splitted } = splitText(text);
5     const tokens = [];
6
7     if (!vocabLoading) {
8         splitted.forEach((element) => {
9             if (vocab[element] !== undefined) {
10                 tokens.push(vocab[element]);
11             }
12         });
13     }
14
15     return tokens;
16 };

```

```

11     }
12   });
13
14   while (tokens.length < maxLen) {
15     tokens.push(0);
16   }
17 }
18 return tokens.slice(0, maxLen);
19 };
20
21 export default tokenize;

```

Listing 8.4: Die tokenize Funktion

## 8.4 Laden des Modells und Vokabulars

Die Dateien für das Modell und die jeweiligen Gewichte wie ebenfalls die Datei für das Vokabular wurden in AWS gehostet. Um diese Dateien in den Browser zu laden wurden „Hooks“ (ähnlich wie Funktionen) in React geschrieben. Der Grund dafür ist, dass dadurch eine Abstraktion dieser Prozesse stattfindet und diese Prozesse von dem eigentlichen Prozess der Erstellung einer Benutzeroberfläche getrennt werden. Damit lassen sie sich im Hauptprozess einfach aufrufen, und der Konsument dieser Funktion muss nicht mehr wissen, wie diese Daten geladen werden können. Damit sind diese Prozesse wiederverwendbar und leichter zu kontrollieren. Im wesentlichen wird aber in der Funktion 8.5 das Modell geladen (ähnlich wird auch das Vokabular geladen). Das Modell wird in eine Variable `model` gespeichert. Wenn das Modell noch lädt, wird dieses ebenfalls in eine Variable `modelLoading` gespeichert. Diese beiden Variablen werden der gesamten UI mitgeteilt. Somit können alle Komponenten diesen Status wissen und darauf reagieren. Durch das Einbinden von `TensorFlow.js` in die UI, hat der Client vollen Zugriff `TensorFlow`. Mit der Methode `loadLayersModel` kann ein Modell geladen werden, welches aus mehreren Ebenen besteht, einschließlich seiner Gewichte. Die Methode `ready` gibt ein Promise zurück, welches aufgelöst wird, wenn das aktuell ausgewählte Backend initialisiert wurde. Es findet also eine „asynchrone Initialisierung“ statt. Die Ausgabe der `useLoadModel` Hook ist das Modell einschließlich aller anderen Variablen im Zusammenhang mit dem Laden des Modells.

```

1 import { useState, useEffect } from 'react';
2 import * as tf from '@tensorflow/tfjs';
3
4 const useLoadModel = (url) => {

```

```

5  const [model, setModel] = useState();
6  const [modelLoading, setModelLoading] = useState(true);
7  const [ModelError, setModelError] = useState('');
8
9  const loadModel = async (url) => {
10    try {
11      const model = await tf.loadLayersModel(url);
12      setModel(model);
13      setModelLoading(false);
14    } catch (error) {
15      setModelError(error);
16      setModelLoading(true);
17    }
18  };
19
20  useEffect(() => {
21    tf.ready().then(() => {
22      loadModel(url);
23    });
24  }, [url]);
25
26  return {
27    model,
28    setModel,
29    modelLoading,
30    setModelLoading,
31    ModelError,
32    setModelError,
33  };
34};
35
36 export default useLoadModel;

```

Listing 8.5: Das useLoadModel Hook

## 8.5 Die Vorhersage des Modells

Die predict-Funktion wird in der Datei App.js aufgerufen und ist somit die Funktion, welches aufgerufen wird, wenn ein Ereignis stattfindet (der Nutzer auf den Button klickt). Die API von TensorFlow.js bietet eine Methode Namens tidy welches die bereitgestellte Funktion ausführt und nach der Ausführung alle von der Funktion zugewiesenen Zwischen-tensoren mit Ausnahme der Funktion vom Speicher entleert. Mit dieser Methode können

Speicherleaks vermieden werden. Ein Array mit den Tokens wird in ein 2-dimensionales Tensor umgewandelt und geht durch das Modell. Mit dispose wird der Tensor dann am Ende aus dem Speicher entsorgt. In Abbildung 8.1 befindet sich die UI.

```

1 const predict = async () => {
2     const predictedClass = await tf.tidy(() => {
3         const tokenisation = tokenize(inputText, vocabLoading, vocab, maxLen);
4         if (tokenisation.length > 0) {
5             const input = tf.tensor2d(tokenisation, [1, maxLen]);
6             if (!modelLoading) {
7                 const predictions = model.predict(input);
8                 return predictions.as1D().argMax();
9             }
10        }
11    });
12 };

```

Listing 8.6: Auszug aus der predict Funktion

## Clickbait 🇩🇪 Vorhersage



Abbildung 8.1: Die Benutzeroberfläche, welches im Hintergrund mit dem Deep Learning Modell kommuniziert.

## 8.6 Vergleich beider Ansätze

In diesem Abschnitt sollen die Ansätze „serverseitiges Deep Learning“ und „clientseitiges Deep Learning“ verglichen werden. Beide Ansätze haben Ihre Vor- und Nachteile. Die Vorteile für das clientseitige Deep Learning sind, dass es geringe Serverkosten hat, die Inferenzlatenz geringer ist und eine Datenprivatsphäre gewährleistet ist.

*Serverkosten* spielen beim Entwerfen und Skalieren von Webdiensten eine wichtige Rolle. Häufig müssen GPU Server bereitgestellt [13, 19]. Beim clientseitigen Deep Learning muss ein kleines Modell (in diesem Beispiel ca. 3 MB groß) bereitgestellt werden und daraus kann die Inferenz stattfinden.

Ein weiterer Punkt ist die *Inferenzlatenz*. Für bestimmte Arten von Anwendungen ist die Latenz Anforderung so hoch, dass die Deep-Learning-Modelle auf der Clientseite ausgeführt werden sollten. Alle Anwendungen, die Audio-, Bild- und Videodaten in Echtzeit enthalten, fallen in diese Kategorie. Wenn Daten erst auf ein Server geladen werden müssen um dann eine Antwort zu erhalten, steigt somit auch die Latenzzeit. Die clientseitige Inferenz behebt diese potenziellen Latenz- und Konnektivitätsprobleme, indem die Daten und die Berechnung auf dem Gerät gespeichert werden [13, 20]. Bei NLP Anwendungen ist dieses kein großer Argument, da Textdaten meistens kleiner sind, ist die Inferenzlatenz auch geringer.

*Datenschutz* ist ein weiterer Vorteil des clientseitigen Deep Learnings. Das Thema Datenschutz wird heute immer wichtiger. Für bestimmte Arten von Anwendungen ist Datenschutz eine absolute Voraussetzung. Anwendungen in Bezug auf Gesundheits- und medizinische Daten sind ein prominentes Beispiel. In vielen Ländern erlauben die Datenschutzbestimmungen für Gesundheitsinformationen nicht, dass z.B. Bilder auf einen zentralen Server übertragen werden. Ein weiteres Szenario könnten juristische Dokumente sein, die nicht auf ein drittes Server geladen werden sollten [13, 20].

Für *größere Modelle* eignet sich das clientseitige Deep Learning nicht, da es für den Nutzer nicht praktisch ist, ein Modell mit mehreren GB Größe, in den Browser zu laden. Ein weiterer Vorteil für das serverseitige Deep Learning ist, dass *Python* als Programmiersprache wesentlich reifer ist, als *JavaScript*, wenn es um Deep Learning geht.

## 9 Fazit und Ausblick

In diesem Kapitel sollen sowohl die Forschungsfragen aus dem Kapitel 1 beantwortet werden, als auch ein möglicher Ausblick, diese Arbeit zu ergänzen oder zu erweitern festgelegt werden.

**Wie kann ein Datensatz für deutsche Clickbaits erstellt werden?** Im Abschnitt 6.4 wurden die Ergebnisse eines Datensatzes vorgestellt, welches ohne den Einsatz eines Menschen gelabelt wurden. Das Labeln wurde vollständig durch ein Programm durchgeführt (siehe Abschnitt 6.3). Die Daten wurden im Vorfeld speziell für ein bestimmtes Problem gesammelt wurden, also die Clickbaits aus Seiten, wo es Möglicherweise viele Clickbaits gibt und Wikinews, als Gegenbeispiel dafür. Der Vorteil des „automatischen Labelns“ liegt darin, dass es keinen menschlichen Eingriff gibt und somit weniger Zeitaufwand entsteht. Der Nachteil ist, dass die Qualität sinkt, da alleine die Konditionen, die durch Programme festgehalten werden, die bestimmt sind. Hier könnte mit mehr Aufwand ein Datensatz mit einer besseren Qualität geben. Wie im Abschnitt 7.6 zu sehen ist, führt dieser Ansatz nicht immer zu einer richtigen Vorhersage. Das Modell denkt, bei einer Frage oft an die Klasse Clickbaits. Dieses ist rückzuschließen auf die Daten die aus Wikinews gesammelt wurden. Wikinews ist keine Plattform, welches mit Werbeeinnahmen lebt sondern mit Spenden, sodass es seine Schlagzeilen nicht mit Stilmitteln gestaltet, welches unbedingt die Aufmerksamkeit der Nutzer erhalten soll. Das Modell sieht entsprechend die Eingaben mit den Augen von „Wikinews“ oder „Nicht-Wikinews“. Hier sollten neben Wikinews andere Quellen herangezogen werden, die die Klasse der Nicht-Clickbaits „harmonischer“ machen, da anzunehmen ist, dass die Welt der Nachrichten nicht nur aus Wikinews und den anderen Quellen besteht.

**Wie gut können aus den ermittelten Daten ein Modell trainiert werden, um Clickbaits zu erkennen?** Wenn es um Sprache und die Analyse der Sprache geht, gibt es sehr viele Szenarien, die sehr schwierig in einem kleinen Modell, wie das Modell welches in dieser Arbeit erstellt wurde, abgebildet werden können. Dabei spielt auch die Rolle des Datensatzes neben dem Eigentlichen Modell eine wichtige Rolle. Diese Arbeit hat gezeigt,

dass ohne menschliches Labeln der Daten, völlig programatisch ein Datensatz erstellt werden. Es ist zunächst ein „Domainwissen“ notwendig, welches aus der Literaturanalyse und aus der sprachlichen Analyse durch NLP erlangt werden kann. Dieses Wissen kann dafür eingesetzt werden, um bestimmte Teile eines Rohdatensatzes verwenden zu können. Das menschliche Labeln der Daten wird aber dadurch nicht komplett ersetzt werden, da gesehen wurde, dass trotzdem Mängel entstehen können bei dieser Methode. Beim Training wurde z.B. deutlich, dass das Modell mit diesem Datensatz sich an die Daten überanpasst. Es ist erforschen, wie es sich mit einem anderen Datensatz, welches besser gelabelt wird, verhält. Die Aufgabe war aber, ein binäres Klassifikationsmodell zu erstellen, welches kurze Texte (Schlagzeilen) als Eingabe erhielt. Diese Aufgabe könnte erweitert werden. Als Eingabe könnte z.B. zusätzlich eine Videosequenz eingeführt werden, da viele Clickbaits auch auf Videoportalen vorkommen.

### **Wie können Deep Learning Modelle in eine Web-Anwendung eingebettet werden?**

Mit der steigenden Digitalisierung werden immer mehr Deep Learning Ansätze den breiten Massen zugänglich. Der größte Teil der Menschen benutzt dabei den Browser als Zugang zu diesen Modellen und Lösungen. Es kann ein Server eingerichtet werden, welches dem Clienten das Modell anbietet. Durch clientseitiges Deep Learning entstehen neue Möglichkeiten. Wie einfach ist es also statt der herkömmlichen Art, eine Deep Learning Web-Anwendung ohne einen Server zu erstellen? Diese Frage ist mit der Implementierung des Modells für das Web beantwortet wurden. Das Modell wurde mit Python entwickelt und in ein Webformat umgewandelt. Es darf nicht vergessen werden, dass beide Versionen den selben Input bekommen müssen, damit diese vergleichbar funktionieren. Die Tokens müssen also auf die selbe Art und Weise bearbeitet und umgewandelt werden. In Zukunft könnte eine mobile Version des Modells erstellt werden, da JavaScript auch auf mobilen Endgeräten arbeiten kann.

## **A Anhang**

# Literaturverzeichnis

- [1] Embeddings: Translating to a Lower-Dimensional Space. <https://developers.google.com/machine-learning/crash-course/embeddings/translating-to-a-lower-dimensional-space>. Accessed: 2020-12-15.
- [2] German Tagsets | Institut für Maschinelle Sprachverarbeitung | Universität Stuttgart. <https://www.ims.uni-stuttgart.de/forschung/ressourcen/lexika/germantagsets>. Accessed: 2021-01-08.
- [3] Gradient Descent: All You Need to Know | Hacker Noon. <https://hackernoon.com/gradient-descent-aynk-7cbe95a778da>. Accessed: 2020-12-24.
- [4] Machine Learning Glossary | Google Developers. [https://developers.google.com/machine-learning/glossary#learning\\_rate](https://developers.google.com/machine-learning/glossary#learning_rate). Accessed: 2021-01-15.
- [5] C. C. Aggarwal. *Neural Networks and Deep Learning*. Springer International Publishing, 2018.
- [6] A. Agrawal. Clickbait detection using deep learning. *Proceedings on 2016 2nd International Conference on Next Generation Computing Technologies, NGCT 2016*, (October):268–272, 2017.
- [7] A. Anand, T. Chakraborty, and N. Park. We used Neural Networks to Detect Clickbaits: You won't believe what happened Next! Technical report, 2019.
- [8] 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.
- [9] M. Appel. *Die Psychologie des Postfaktischen: Über Fake News, „Lügenpresse“, Clickbait & Co.* Springer Berlin Heidelberg, 2019.

- [10] Y. Bengio, R. Ducharme, P. Vincent, C. Jauvin, J. U. Ca, J. Kandola, T. Hofmann, T. Poggio, and J. Shawe-Taylor. A Neural Probabilistic Language Model. Technical report, 2003.
- [11] P. Biyani, K. Tsoutsouliklis, and J. Blackmer. Detecting Clickbaits in News Streams Using Article Informality. *Thirtieth AAAI Conference on Artificial Intelligence*, pages 94–100, 2016.
- [12] J. N. Blom and K. R. Hansen. Click bait: Forward-reference as lure in online news headlines. *Journal of Pragmatics*, 76:87–100, jan 2015.
- [13] S. Cai, S. Bileschi, and E. Nielsen. *Deep Learning with JavaScript: Neural networks in TensorFlow.js*. Manning Publications, 2020.
- [14] A. Chakraborty, B. Paranjape, S. Kakarla, and N. Ganguly. Stop Clickbait: Detecting and Preventing Clickbaits in Online News Media. Technical report.
- [15] F. Chollet. *Deep Learning with Python*. Manning Publications Company, 2017.
- [16] R. Francois Chaudhury and R. S. Mundra. Understanding Convolutional Neural Networks for NLP – WildML.
- [17] M. W. Gardner and S. R. Dorling. Artificial neural networks (the multilayer perceptron) - a review of applications in the atmospheric sciences. *Atmospheric Environment*, 32(14-15):2627–2636, 1998.
- [18] G. E. Hinton and S. Osindero. A Fast Learning Algorithm for Deep Belief Nets Yee-Whye Teh. Technical report.
- [19] A. C. Ian Goodfellow, Yoshua Bengio. *Deep Learning*. 2016.
- [20] D. Jurafsky and J. H. Martin. Speech and Language Processing An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition Third Edition draft. Technical report.
- [21] L. Lai and A. Farbrot. Social Influence What makes you click? The effect of question headlines on readership in computer-mediated communication. *Social Influence*, 9, 2014.
- [22] Y. Lecun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

- [23] E. D. Liddy. SURFACE SURFACE Center for Natural Language Processing School of Information Studies (iSchool) 2001 Natural Language Processing Natural Language Processing Natural Language Processing 1. Technical report.
- [24] M. Main, U. Rony, N. Hassan, and M. Yousuf. Diving Deep into Clickbaits: Who Use Them to What Extents in Which Topics with What Effects?
- [25] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient Estimation of Word Representations in Vector Space. Technical report.
- [26] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Distributed Representations of Words and Phrases and their Compositionality. Technical report, 2013.
- [27] M. A. Nielsen. *Neural networks and deep learning*, volume 2018. Determination press San Francisco, CA, 2015.
- [28] J. Patterson and A. Gibson. *Deep Learning a Practitioner'S Approach*, volume 29. 2019.
- [29] M. Potthast, T. Gollub, M. Hagen, and B. Stein. The Clickbait Challenge 2017: Towards a Regression Model for Clickbait Strength. Technical report.
- [30] M. Potthast, T. Gollub, K. Komlossy, S. Schuster, M. Wiegmann, E. Patricia, G. Fernandez, M. Hagen, and B. Stein. Crowdsourcing a Large Corpus of Clickbait on Twitter. Technical report.
- [31] A. Pujahari and D. Singh Sisodia. Clickbait Detection using Multiple Categorization Techniques. Technical report.
- [32] F. Rosenblatt. THE PERCEPTRON: A PROBABILISTIC MODEL FOR INFORMATION STORAGE AND ORGANIZATION IN THE BRAIN 1. Technical Report 6.
- [33] S. Ruder. An overview of gradient descent optimization algorithms. sep 2016.
- [34] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088), 1986.
- [35] N. Srivastava, G. Hinton, A. Krizhevsky, and R. Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. Technical report, 2014.
- [36] Stanford University Course cs231n. CS231n Convolutional Neural Networks for Visual Recognition. *Stanford University Course cs231n*, page 30, 2018.

- [37] Stanford University Course cs231n. CS231n Convolutional Neural Networks for Visual Recognition. *Stanford University Course cs231n*, page 30, 2018.
- [38] O. Tenenboim and A. A. Cohen. What prompts users to click and comment: A longitudinal study of online news. *Journalism*, 16(2):198–217, 2015.
- [39] L. Van Der Maaten and G. Hinton. Visualizing Data using t-SNE. Technical report, 2008.
- [40] B. Vijgen. THE LISTICLE: AN EXPLORING RESEARCH ON AN INTERESTING SHAREABLE NEW MEDIA PHENOMENON. *Studia Universitatis Babes-Bolyai - Ephemerides*, 59(1):103–122, 2014.
- [41] P. J. Werbos. Backpropagation Through Time: What It Does and How to Do It. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.
- [42] S. Zannettou, S. Chatzis, K. Papadamou, and M. Sirivianos. The good, the bad and the bait: Detecting and characterizing clickbait on youtube. *Proceedings - 2018 IEEE Symposium on Security and Privacy Workshops, SPW 2018*, pages 63–69, 2018.
- [43] Y. Zhang and B. C. Wallace. A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification. Technical report.

# **Erklärung**

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, insbesondere keine anderen als die angegebenen Informationen aus dem Internet. Diejenigen Paragraphen der für mich gültigen Prüfungsordnung, welche etwaige Betrugsversuche betreffen, habe ich zur Kenntnis genommen. Der Speicherung meiner Projekt-Arbeit zum Zweck der Plagiatsprüfung stimme ich zu. Ich versichere, dass die elektronische Version mit der gedruckten Version inhaltlich übereinstimmt.

Bielefeld, den 21. Januar 2021

(Ort) (Datum)

.....  
(Unterschrift)