Technische Hochschule Brandenburg
Fachbereich Wirtschaft

Masterarbeit

Sentimentanalyse deutscher Twitter-Korpora mittels Deep Learning

Autor:	Enrico Bunde <enrico.bunde@th-brandenburg.de>
Matrikelnummer:	20137010
Erstgutachter:	Prof. Dr. Sven Buchholz
Zweitgutachter:	Dipl.-Inform. Ingo Boersch
Eingereicht am:	15.03.2019

Zusammenfassung

In dieser Arbeit sollen mittels zwei verschiedenen Deep Learning Modellen und Worteinbettungen eine Sentimentanalyse anhand deutscher Twitter-Korpora durchgeführt werden. Die untersuchte Problematik besteht zum einen in der Parameteroptimierung und zum anderen in der prototypischen Implementierung eines Ansatzes für erklärbare Künstliche Intelligenz sowie der Parameteroptimierung durch Rastersuche. Für die deutsche Sprache gibt es nur wenige Korpora und publizierte Ergebnisse für die Sentimentanalyse, daher soll diese Arbeit zur Überprüfung dienen, inwiefern diese Ergebnisse erreicht oder auch verbessert werden können und welche Parameter sich wie stark auf die Ergebnisse auswirken. Durch konzeptionelle und explorative Experimente sollten diese Thematiken untersucht werden und durch die genannten prototypischen Implementierung Wege aufgezeigt, wie Parameter und Deep Learning Modelle weiter optimiert werden können.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Die Klassifikation von Texten spielt heute eine große Rolle und betrifft bereits in der jetzigen Zeit viele Menschen. Ein Beispiel für Textklassifizierung stellt die automatische Erkennung von unerwünschten E-Mails dar, welche als Spam-Mails klassifiziert werden. Durch das Internet, die Digitalisierung und sozialen Medien wie Twitter wächst die Menge an Texten täglich weiter.

Diese Datenmengen können eingesetzt werden, um Deep Learning Modelle zu trainieren. Die Modelle können anschließend für die Klassifizierung von Texten Anwendung finden. Ein weiteres Einsatzgebiet liegt in der Sentimentanalyse. Dabei sollen Texte ausgewertet werden, um eine Stimmung, Meinung oder Gefühl zu erkennen.

Konkrete Anwendungsmöglichkeiten liegen zum Beispiel in der Sentimentanalyse von Kritiken, Bewertungen oder Texten in sozialen Medien. Durch ein trainiertes Deep Learning Modell kann ein Unternehmen somit beispielsweise automatisiert feststellen, in welchem Kontext deren Produkte oder Dienstleistungen auf Twitter vorkommen. Also werden diese häufiger in Tweets erwähnt, welche eine positive oder negative Stimmung transportieren. Diese Erkenntnis kann wiederum Einfluss auf eine Unternehmensstrategie nehmen.

1.1 Motivation

Für die Sentimentanalyse gibt es Datensätze, welche weit verbreitet und häufig eingesetzt werden. Oft handelt es sich dabei um englischsprachige Daten. Die Suche nach deutschsprachigen Datensätzen gestaltet sich schwierig, da es zwar einige größere Textdatensätze gibt, aber nur wenige für die Aufgabe der Sentimentanalyse.

Für eine solche Analyse wurden drei deutschsprachige Twitter-Korpora ausgewählt. Diese haben unterschiedliche Größen und verfügen über die drei Stimmungen *negativ*, *neutral* und *positiv*. Weiterhin gibt es zu diesen Datensätzen publizierte Ergebnisse, welche als Maßstab dienen.

1.2 Forschungsfrage und Gegenstand

Die Arbeit hat das Ziel, zwei verschiedene Deep Learning Modelle zu implementieren. Weiterhin sollen explorative Experimente mit unterschiedlichen Parametereinstellungen durchgeführt werden. Hinzu kommen zwei verschiedene Methoden für die Repräsentation von Wörtern in neuronalen Netzwerken. Mittels der zwei Modelle und eine Vielzahl von Experimenten sollen die bestmöglichen Parametereinstellungen für die Tests gefunden werden. Hieraus lassen sich einige Aufgaben und Forschungsgegenstände ableiten. Diese lauten wie folgt:

1. Implementierung der Modelle und Methoden für die Repräsentation von Wörtern.
2. Erstellung eines Konzepts für strukturierte Experimente.
3. Experimentelle Prüfung verschiedener Parametereinstellungen und deren Auswirkungen auf die Leistung.
4. Untersuchung und Visualisierung der Daten.
5. Vergleich der Experimente und Auswahl der Parameter für die Tests.
6. Prototypische Implementierung von LIME, einem Ansatz Deep Learning Modelle verstehen zu können.
7. Prototypische Implementierung der Rastersuche zur Optimierung von Hyperparametern.

1.3 Abgrenzung des Vorhabens

In dieser Arbeit wird Python als Programmiersprache sowie auf Python basierende Deep Learning Frameworks und Bibliotheken eingesetzt. Der Hauptfokus liegt auf der Implementierung der Deep Learning Modelle sowie den verschiedenen Experimenten. Dabei werden zwei Methoden für die Repräsentation von Wörtern eingesetzt sowie prototypisch, ein Ansatz Deep learning Modelle verstehen zu können und ein anderer um die Hyperparameter zu optimieren.

1.4 Aufbau der Arbeit

Die Arbeit wurde in sieben Kapitel gegliedert. In dem Kapitel 1 wird das Vorhaben beschrieben, Aufgaben und Forschungsgegenstände aufgestellt sowie die Arbeit und deren Ziel abgegrenzt.

In den Kapitel 2 und 3 werden die theoretischen Aspekte der Arbeit erläutert. Dabei wird in dem zweiten Kapitel zunächst auf die Theorie im Bereich Verarbeitung von natürlicher Sprache eingegangen und die Twitter-Korpora werden vorgestellt. In dem dritten Kapitel wird auf die Grundlagen im Deep Learning eingegangen, welche relevant für den praktischen Teil sind.

Während des Kapitel 4 soll das Vorgehen und die methodischen Ansätze erläutert werden. Weiterhin wird die Konzeption erstellt, welche die Basis der praktischen Experimente darstellt.

In dem nachfolgenden Kapitel 5 wird die Soft- und Hardware erläutert sowie die Kernelemente der Implementierung. Außerdem wird auf die praktische Umsetzung eingegangen. Wobei erläutert wird, wie die Daten vorverarbeitet wurden und die Implementierung mittels Python erfolgte.

Kapitel 6 dokumentiert alle Experimente, deren Parametereinstellungen und stellt alle Resultate in einer tabellarischen Form dar.

In dem letzten Kapitel 7 sollen die Ergebnisse zusammengefasst und ein Ausblick darüber gegeben werden, wie die Resultate in weiteren Arbeiten verbessert werden können.

2 Verarbeitung natürlicher Sprache, Deep Learning in der Sentimentanalyse und deutsche Twitter-Korpora

In diesem Kapitel werden relevante Themen aus den Bereichen der Verarbeitung von natürlicher Sprache (engl. Natural Language Processing - NLP) sowie Deep Learning in der Sentimentanalyse erläutert. Weiterhin werden die Korpora vorgestellt und es folgt eine Übersicht des resultierenden Datensatzes.

2.1 Verarbeitung natürlicher Sprache

Bei NLP steht die Verarbeitung von menschlicher Sprache mittels Computer im Vordergrund. Eine mögliche Anwendung besteht in der maschinellen Übersetzung (engl. machine translation), wobei ein Satz in einer menschlichen Sprachen eingelesen werden kann und derselbe Satz in einer anderen Sprache ausgegeben wird. Die meisten Anwendungen im Bereich NLP basieren auf Sprachmodellen (engl. language models), welche eine Wahrscheinlichkeitsverteilung über Wörter, Zeichen oder Bytes in einer natürlichen Sprache definieren (Goodfellow et al. 2016: S. 463).

Deep Learning hat sich in dem Bereich der Verarbeitung von natürlicher Sprache als effektiv erwiesen. Es gibt viele wissenschaftliche Arbeiten, welche sich mit Deep Learning im Bereich von Textklassifikation befassen. Es gibt Ansätze die sich mit der Klassifizierung von Texten auf Zeichen-Ebene befassen (Zhang et al. 2015). Eine anderes Anwendungsbeispiel liegt in der Generierung von Texten, mittels neuronalen Netzen (Sutskever et al. 2011). Auch die Sentimentanalyse mittels Twitter-Daten wurde bereits zahlreich untersucht. Zum Beispiel das Lernen von sentimentspezifischen Worteinbettungen (engl. Word Embeddings) für eine Twitter Sentimentklassifizierung (Tang et al. 2014). Weiterführende Beispiele für diesen Anwendungsbereich werden in Kapitel 2.5.1, 2.5.2 und 2.5.3 ausgeführt.

2.2 Mit Texten arbeiten

Texte stellen eine weitverbreitete Form von Sequenzdaten (engl. sequence data) dar. Was Sequenzdaten von anderen Datentypen abgrenzt, ist das diese in

einer bestimmten Reihenfolge vorliegen (Raschka & Vahid 2017, S. 738). Diese Sequenz kann bei Texten aus Buchstaben bzw. Zeichen oder aus Wörtern bestehen. Deep Learning Modelle können keine rohen Texte als Eingabe verarbeiten. Sie benötigen numerische Tensoren. Durch Vektorisierung können somit Texte in solche Tensoren transformiert werden. Dabei gibt es unterschiedliche Methoden, dies zu erreichen. Es ist möglich die Texte in Wörter oder Zeichen zu zerlegen und diese in Vektoren zur überführen. Die Ergebnisse dieser Zerlegung werden als Tokens (engl. tokens) bezeichnet (Chollet 2018: S. 181).

Ein einfaches Beispiel für die Transformation eines Tokens in einen Vektoren stellt die One-Hot-Kodierung (engl. one-hot encoding) von Wörtern oder Zeichen dar. Die Funktionsweise lässt sich wie folgt beschreiben. Jedem Wort wird ein eindeutiger ganzzahliger Index zugewiesen. Es folgt eine Umwandlung des Indexes i in einen binären Vektoren, mit der Größe N, die Größe des Vokabulars. Der Vektor wird dann nur Nullen enthalten, bis auf den i-ten Eintrag, eine 1. Ein kleines Beispiel lässt sich in Python wie folgt darstellen (Chollet 2018, S. 182). Dabei werden zunächst zwei Textdaten angegeben als *one_hots*. Danach folgt die Tokenisierung und anschließend die Vektorisierung der Texte innerhalb der Variablen *results*.

```
one_hots = ["onehot eins", "onehot zwei"]

...

for one_hot in one_hots:
  for word in one_hot.split():
    if word not in token_index:
      token_index[word] = len(token_index) + 1

...

results = np.zeros(shape=(len(one_hots),
                          max_length,
                          max(token_index.values()) + 1))
```

Mittels der obigen Implementierung wird dem Index 0, des resultierenden Vektors nichts zugewiesen und bleibt somit Null. Als Ergebnis werden die zwei folgenden Vektoren generiert. In den beiden Texten gibt es je zwei Worte

und drei verschiedene Worte insgesamt. Das erste Wort *onehot* ist identisch, so auch der erste Vektor, welcher dieses Wort repräsentiert. In dem ersten Text wird in der Zweiten Zeile das Wort *eins* dargestellt, wobei die 1 nun eine Spalte weiter liegt. In dem *onehot zwei* Vektor ist die 1 wieder um eine Spalte weiter gerückt und repräsentiert das Wort *zwei*. Somit wurde jedes Wort in einen eindeutigen Vektor umgewandelt und kann als Eingabe für ein Deep Learning Modell dienen.

$$onehot\ eins = \begin{bmatrix} [0 & 1 & 0 & 0] \\ [0 & 0 & 1 & 0] \end{bmatrix} \tag{1}$$

$$onehot\ zwei = \begin{bmatrix} [0 & 1 & 0 & 0] \\ [0 & 0 & 0 & 1] \end{bmatrix} \tag{2}$$

2.3 Worteinbettungen

Eine weit verbreitete und leistungsfähige Möglichkeit Wörter mit Vektoren zu verbinden, stellen die Worteinbettungen dar. Diese werden mit Fließkommazahlen abgebildet. Dabei gibt es zwei verschiedene Wege diese Worteinbettungen zu erhalten. Zum einen können diese während der eigentlichen Aufgabe, wie einer Sentimentanalyse erlernt werden. Dabei ähnelt der Lernprozess, dem Erlernen von Gewichtungen (engl. weights) von neuronalen Netzwerken. Ein anderer Weg Worteinbettungen einzusetzen besteht darin, vorberechnete oder vortrainierte Worteinbettungen (engl. pretrained word embeddings) einzusetzen (Chollet 2018, S. 184).

2.3.1 Word2Vec und vortrainierte Worteinbettungen

Ein Modell welches Worteinbettungen umsetzt wird Word2Vec genannt. Dabei stammt dieses Modell aus der Arbeit von (Mikolov et al. 2013), mit dem Titel *Efficient Estimation of Word Representations in Vector Space*. In dieser Arbeit wurden zwei Modell-Architekturen vorgestellt, welche Worteinbettungen von sehr großen Datensätzen erlernt haben. Anschließend wurde die Qualität der Worteinbettungen mittels Wortähnlichkeiten überprüft. Weiterhin wurden diese Ergebnisse mit verschiedenen Arten von neuronalen Netzwerken eingesetzt und es konnte eine große Steigerung der Genauigkeit erreicht werden.

Word2Vec kann durch zwei Modell-Architekturen realisiert werden. Ein Modell heißt Continuous Bag-of-Words (CBOW). Dabei versucht dieses Modell, aus einem gegebenen Kontext und Worten ein Zielwort vorherzusagen. Die zweite Architektur, mit der Word2Vec umgesetzt werden kann ist das Skip-Gram Modell. Ziel dieses Ansatzes ist es Wort-Vektoren dahingehend zu erlernen, dass diese gut darin werden, umliegende Wörter in einem Kontext, zu bestimmen (Meyer 2016: S. 2). Die Architekturen lassen sich nach (Mikolov et al. 2013) wie auf Abbildung 1 darstellen. Dabei ist gut zu erkennen, dass bei dem CBOW-Modell verschiedene Wörter als Input dienen können, um ein Wort als Ausgabe zu erhalten. Bei dem Skip-gram-Modell hingegen dient lediglich ein Wort als Input und dieses kann ein Wort in dem Kontext klassifizieren.

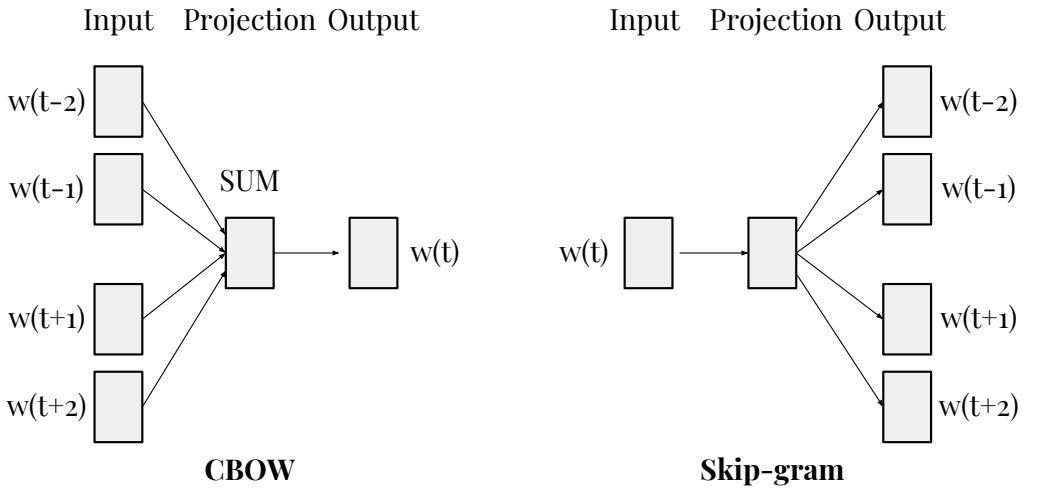


Abbildung 1: Architekturen von Word2Vec links ist das CBOW-Modell zu sehen und rechts das Skip-gram-Modell (in Anlehnung an Mikolov et al. 2013: S. 5)

Bei den hier verwendeten vortrainierten Worteinbettungen handelt es sich um eine Arbeit von (Müller 2015). Dabei wurden deutsche Worteinbettungen mittels Skip-gram trainiert. Die Merkmalsvektoren (engl. feature vectors) sind 300-dimensional und der Wortschatz besteht aus 608.130 Worten. Als Trainingsdaten haben zwei große deutsche Korpora gedient. Zum einen deut-

sche Wikipedia Artikel bis 2015 und zum anderen News Artikel bis 2013.

2.3.2 FastText und vortrainierte Worteinbettungen

Bei fastText handelt es sich um einen Ansatz und eine Bibliothek, welche durch das Labor von Facebooks AI Research (FAIR) entwickelt wurden. Dabei unterstützt fastText ebenfalls die zwei Architekturen CBOW und Skip-gram. Bei fastText handelt es sich nicht um einen neuen Ansatz Worteinbettungen zu trainieren. Es stellt eine Erweiterung des Word2Vec-Modells dar. Der Unterschied lässt sich wie folgt erläutern. Bei Word2Vec werden Worte als eine atomare Einheit gesehen und für jedes Wort wird ein Vektor generiert. Bei fastText hingegen, werden Worte in N-Grams unterteilt. Somit ließe sich beispielsweise das Wort *<apfel>* in folgende N-Grams aufsplitten: *<ap, apf, apfe, apfel, apfel>, pfe, pfel, pfel>, fel, fel>, el>*, wobei das kleinste N-Gram 3 und das größte 6 Zeichen groß sein kann. Diese Aufteilung in N-Grams kann über Wörter hinweg geschehen. Somit sollen selten vorkommende Wörter sinnvoller mit Vektoren verknüpft werden können, da Teile deren N-Grams mit anderen Worten zusammenhängen (Rajasekharan 2017).

Die vortrainierten Worteinbettungen wurden anhand von 50 Millionen deutschen Tweets trainiert und sind ebenfalls 300-dimensional. Der Wortschatz umfasst 533.796 Wörter.

2.3.3 Unterschiede der vortrainierten Worteinbettungen

Der größte Unterschied zwischen den Word2Vec und fastText Worteinbettungen besteht in den Trainingsdaten, welche für das Training eingesetzt wurden. Bei dem Word2Vec-Modell kamen Wikipedia und News-Artikel zum Einsatz. Für das fastText-Modell dienten Tweets als Korpus. Weiterhin beinhalten die Word2Vec-Worteinbettungen 608.130 Wörter, wohingegen die fastText-Worteinbettungen 533.796 Worte enthalten. Da sich die Trainingsdaten voneinander unterscheiden, sollen einige Experimente mit den Worteinbettungen durchgeführt werden.

Die hier eingesetzte Bibliothek für die Arbeit mit den vortrainierten Worteinbettungen heißt gensim (Řehůřek 2019). Damit wird es möglich eigene Worteinbettungen anhand von Daten zu erstellen und weiterzuverarbeiten. Die beiden vortrainierten Worteinbettungen lagen zunächst als Binärdatei

vor. Für die Implementierung war es aber nötig, diese in eine Textdatei umzuwandeln. Auf dem folgenden Codeausschnitt, wird diese Umwandlung dargestellt. Dabei werden die Variablen *model_ft* und *model_w2v* angelegt und mittels gensim werden die Binärdateien geladen. Durch gensims *save_word2vec_format()* werden die Binärdateien dann in eine Textdatei gespeichert.

```
model_ft = KeyedVectors.load_word2vec_format("
    embed_tweets_de_300D_fasttext", binary=False)
model_w2v = KeyedVectors.load_word2vec_format("german.model",
    binary=True)

model_ft.save_word2vec_format("fasttext_50mill_tweets.txt",
    binary=False)
model_w2v.save_word2vec_format("german_model_2.txt", binary=False
    )
```

Im Format einer Textdatei bestehen die Worteinbettungen aus einer ersten Spalte, in welcher die Worte stehen. Je Zeile gibt es ein Wort und für jedes Wort folgen 300 Fließkommazahlen, welche den Vektoren für das Wort darstellen.

Weiterhin ist es möglich ein Wort vorzugeben und ähnliche Wörter auszugeben. Mittels gensims *most_similar()* kann dies umgesetzt werden. Dabei zeigt Tabelle 1 die fünf ähnlichsten Wörter zu dem vorgegebenen Wort *gut*. Es ist bereits ein Unterschied zu erkennen. Da die Word2Vec-Worteinbettungen mittels Wikipedia und News-Artikeln trainiert wurden, ist die Wahrscheinlichkeit groß, dass die Wörter der deutschen Rechtschreibung entsprechen. Die Ergebnisse von fastText hingegen, zeigen, dass Worte wie *gutgut* oder *guuut* vorhanden sind. Dabei handelt es sich um Schreibweisen, die in Chats oder sozialen Medien oft verwendet werden. Zum Beispiel kann man das Wort *guuut* so interpretieren, das etwas mehr als nur gut ist und somit stellt es eine Hyperbel dar.

Word2Vec	**fastText**
hervorragend	schlecht
bestens	toll
schlecht	gutgut
prima	schoen
besser	guuut

Tabelle 1: In dieser Tabelle sind die fünf ähnlichsten Wörter zu dem Wort gut dargestellt. Auf der linken Seite sind die Worte für Word2Vec und auf der rechten Seite fastText. Dabei sind in der ersten Zeile die Worte, welche die höchste Ähnlichkeit aufweisen, aufgezählt. Die restlichen Worte sind in absteigender Folge sortiert.

Die Worteinbettungen lassen sich auch visualisieren. Auf der Abbildung 2 sind die fünf ähnlichsten Worte zu dem Wort gut dargestellt.

(a) Grafische Darstellung zu Word2Vec der fünf ähnlichsten Wörter zu dem Wort gut

(b) Grafische Darstellung zu fastText der fünf ähnlichsten Wörter zu dem Wort gut

Abbildung 2: Grafische Darstellung der fünf ähnlichsten Wörter zu dem Wort gut. Auf der linken Seite ist Word2Vec und auf der rechten fastText. Umgesetzt wurden die Visualisierung mit der Bibliothek t-SNE (Van der Maaten 2019). Bei Word2Vec ist zu sehen, dass die ähnlichen Worte im Winkel nach unten und links zu finden sind. Bei fastText befindet sich das Wort gut in der Mitte und die ähnliche Worte sind in alle Richtungen verteilt. Da es zwei unterschiedliche Modelle sind und verschiedene Trainingsdaten eingesetzt wurden, ist ein direkter Vergleich nicht möglich.

2.4 Deep Learning in der Sentimentanalyse

Deep Learning wird bereits in der Sentimentanalyse eingesetzt. Besonders in der englischen Sprache gibt es den Datensatz *Large Movie Review Dataset*, für die Sentimentanalyse eingestzt wurde. Dieser stammt von (Maas et al. 2011) für die Arbeit *Learning Word Vectors for Sentiment Analysis* und enthält nach eigenen Angaben 25000 Daten für das Training und Testen. Dabei sind verschiedene Sentimentalitäten in den Gruppen Melancholie (engl. melancholy), grässlich (engl. ghastly), glanzlos (engl. lackluster) und romantisch (engl. romantic) unterteilt.

Eine andere Arbeit in diesem Bereich stammt von (Aken et al. 2018) mit dem Titel *Challenges for Toxic Comment Classification: An In-Depth Error Analysis*. In dieser Arbeit geht es um die Klassifizierung von toxischen Kommentaren. Dabei werden verschiedene Deep Learning Modelle eingesetzt und verglichen. Als Datensatz wurden Kommentare der Wikipedia Diskussionsseiten gewählt, welche die Klassen sauber (engl. clean), toxisch (engl. toxic), obszön (engl. obscene), Beleidigungen (engl. insults), Identitätshass (engl. identity hate), schwerwiegend toxisch (severe toxic) und Bedrohung (engl. threat) beinhalten. Außerdem stand ein zweiter Twitter-Datensatz zur Verfügung, welcher die Klassen offensiv (engl. offensive), sauber und Hass (engl. hate) enthält.

Es gibt viele weitere Arbeiten, welche sich mit der Sentimentanalyse und naheliegenden Themen befassen. Dabei werden Korpora vorgestellt, Deep Learning Modelle verglichen oder neue Forschungsansätze aufgezeigt. Die Recherche zeigte allerdings, dass die Anzahl deutscher Korpora gering ist, beispielsweise im Vergleich mit englischsprachigen Datensätzen. In den folgenden Kapitel werden die verwendeten deutschen Twitter-Korpora vorgestellt und erläutert wie diese entstanden sind.

2.5 Deutsche Twitter-Korpora

Durch die Arbeit von (Cieliebak et al. 2017) wurden drei deutsche Twitter-Korpora gefunden. Dabei besitzen alle Datensätze dieselben drei Klassen *negativ*, *neutral* und *positiv*. Im folgenden werden die verschiedenen Korpora vorgestellt und die dazugehörigen Arbeiten zusammengefasst.

2.5.1 SB-10k: German Sentiment Corpus

Dieser Korpus enthält insgesamt 9.738 deutsche Tweets, welche in die drei genannten Klassen unterteilt sind. Entstanden ist der Datensatz in einer Zusammenarbeit zwischen der SpinningBytes AG und der Zurich University of Applied Sciences. Eingesetzt wurde der Korpus in der Arbeit von (Cieliebak et al. 2017) mit dem Titel *A Twitter Corpus and Benchmark Resources for German Sentiment Analysis*. Diese Arbeit verfolgte das Ziel, einen State of the Art Benchmark zur Verfügung zu stellen. Weiterhin wurden Worteinbettungen eingesetzt, die auf einen Datensatz von 300 Millionen Tweets trainiert worden sind.

Um die Daten zu erhalten, ist es notwendig, den Korpus von SpinningBytes (SpinningBytes 2019b) herunterzuladen. Dieser enthält zunächst die Tweet-IDs, Klassen und weitere Informationen. Weiterhin werden Scripte bereitgestellt, mit denen anhand der Tweet-ID der Tweet in den Korpus geschrieben wird. Die Tweets werden somit direkt über die Twitter-API angefragt. Dabei ist es zu folgendem Umstand gekommen. Mit der Zeit können Tweets editiert oder gelöscht werden. Wenn dies der Fall war, so wurde als Tweet ein *Not Available* eingefügt. Mittels der Pandas-Bibliothek kann die TSV-Datei anschließend bearbeitet werden. Dabei wurde die nicht verfügbaren Tweets sofort aussortiert. Die Tabelle 2 stellt die Verteilung der Klassen und Gesamtzahl der Tweets dar.

Sentiment	**Anzahl**
negativ	1.056
neutral	4.367
positiv	1.609
gesamt	**7.032**

Tabelle 2: In der Tabelle ist in der linken Spalte die Klasse abgebildet und in der rechten die Anzahl der Tweets. Es ist zu erkennen, dass die neutralen Tweets mit 4.367 mit Abstand die größte Menge darstellen. Die nächstkleinere Klasse ist die positive mit 1.609 Tweets und die kleinste stellt die negative mit 1.056 Tweets dar. Von den ursprünglichen 9.738 Tweets sind insgesamt 7.032 übrig geblieben.

2.5.2 DAI - Annotierter Twitter Sentiment Datensatz

Dieser Korpus enthält insgesamt 12.597 Tweets in 4 Sprachen (Englisch, Deutsch, Französisch, Portugiesisch). Dabei wurden die Tweets in die drei Klassen *negativ*, *neutral* und *positiv* unterteilt. Der Datensatz konnte vollständig erhalten werden und musste per E-Mail angefragt werden (DAI-Labor 2019). Der Korpus wurde in der Arbeit *Language-Independent Twitter Sentiment Analysis* von (Narr et al. 2012) verwendet. Der Fokus dieser Arbeit lag auf einer Analyse der Merkmale und Durchführbarkeit eines sprachunabhängigen, halbüberwachten Ansatzes zur Klassifizierung von Tweets anhand der Sentiments. In Tabelle 3 wird die Verteilung der Klassen und Gesamtzahl der Tweets dargestellt.

Sentiment	**Anzahl**
negativ	267
neutral	1.136
positiv	385
gesamt	**1.788**

Tabelle 3: Die tabellarische Darstellung der Tweets und deren Anzahl. In diesem Korpus ist eine ähnliche Klassenverteilung zu erkennen, wie in dem Korpus SB-10k: German Sentiment Corpus. Es sind 1.136 neutrale, 385 positive und 267 negative Tweets. Insgesamt umfasst dieser Datensatz 1.788 einzigartige Tweets. Diese wurden je von 3 verschiedenen Benutzern annotiert. Somit sind von den 12.597 Tweets 5.364 für die deutsche Sprache. Für das Projekt wurde je ein einzigartiger Tweet behalten.

2.5.3 German Twitter Sentiment

Dieser Korpus enthielt im Original 109.130 deutsche Tweets. Entstanden ist er für ein Forschungsartikel von (Mozetič et al. 2016) mit dem Titel *Multilingual Twitter Sentiment Classification: The Role of Human Annotators*. In der Arbeit wurde untersucht, wo die Grenzen von automatisierter Twitter Sentiment-Klassifizierung liegen. Dabei wurden Korpora in verschiedenen Sprachen angelegt. Dieser Datensatz war wie auch SB-10k: German Twitter Sentiment, zunächst als CSV-Datei herunterzuladen. Auch hier wurden anschließend anhand der Tweet-ID die Tweets in den Korpus geschrieben. Tabelle 4 bietet eine Übersicht der Klassenverteilung und Anzahl von Tweets.

Sentiment	Anzahl
negativ	15.381
neutral	47.427
positiv	21.348
gesamt	**84.156**

Tabelle 4: Die Klassenverteilung ist auch in dem dritten Korpus gleichbleibend. Die meisten Tweets sind neutral, 47.427. Danach folgen 21.348 positive und 15.381 negative Tweets. Von den ursprünglichen 109.130 Tweets sind 84.156 übrig. Die entfallenen Tweets waren *Not Available*.

2.5.4 Beschaffung, Zusammenfassung der Korpora und publizierte Ergebnisse als Maßstab

Für diese Arbeit wurden die drei Korpora zusammengefasst. Insgesamt sind es somit 92.976 Tweets. Bisher wurden nur die nicht vorhandenen Tweets ausgefiltert. Bei der tatsächlichen Vorverarbeitung der Tweets ist eine kleine Anzahl von Tweets herausgefallen. Dies wird in der Vorerarbeitung der Texte erläutert und war der Fall, wenn beispielsweise ein Tweet nur aus einer Internetadresse bestand. Die folgende Tabelle 5 fasst alle Korpora zusammen.

Sentiment	Anzahl
negativ	16.704
neutral	52.930
positiv	23.342
gesamt	**92.976**

Tabelle 5: Der entstandene Korpus enthält 52.930 neutrale, 23.342 positive und 16.704 negative Tweets.

In der Arbeit von (Cieliebak et al. 2017) werden verschiedene Ergebnisse für diese Korpora genannt. Unter (Cieliebak et al. 2017: S. 45, 46, 49) wird ein englisches System genannt, welches einen F1-Wert von 62,7% erreicht. Auch

wird von (Mozetič et al. 2016) ein F1-Wert von 53,6% dargestellt. Die Ergebnisse von (Cieliebak et al. 2017) haben Resultate zwischen 47,30% und 65,09%, wobei hier die F1-Werte nur von den positiven und negativen Klassen berechnet wurde, mittels $(F1_{pos} + F1_{neg})/2$. Der F1 Wert von allen drei Klassen war im Durchschnitt noch einmal 4,42% höher. Diese Ergebnisse können als Maßstab dienen, die hier erbrachten Resultate einordnen zu können.

Ein direkter Vergleich der Ergebnisse ist allerdings nicht möglich. Die verschiedenen Resultate sind von Systemen für die *International Workshop on Semantic Evaluation (SemEval)* entstanden. Dabei handelt es sich um eine Serie von Aufgaben für semantische Systeme, die jährlich stattfinden (SemEval 2019). In (Cieliebak et al. 2017) wird ein System eingesetzt, welches aus einem 3-Phasen Training besteht. Dabei wurden zunächst die rohen Tweets genutzt um Worteinbettungen zu generieren. Anschließend werden die Worteinbettungen angepasst, indem Merkmale wie Smileys erlernt wurden. Danach folgte das eigentliche Training eines Deep Learning Modells mit annotierten Tweets. Die Worteinbettungen wurden außerdem mittels Skip-gram an einem 300 Millionen Tweets großen Korpus trainiert. Ein zweiter 40 Millionen Tweets großer Datensatz wurde für das Erlenen von Smileys genutzt. Daher steckt in diesen Systemen bereits mehr Leistungskraft als es mit den hier verwendeten Worteinbettungen möglich sein müsste.

3 Deep Learning

Deep Learning stellt ein Teilgebiet von Machine Learning dar. Daher ist es notwendig, gewisse Kenntnisse aus dem Bereich Machine Learning zu besitzen, um auch Deep Learning gut verstehen zu können. Um dies zu erreichen, sollen im folgenden einige Begriffe und Grundlagen dargestellt werden.

3.1 Was ist Deep Learning

Bei Deep Learning handelt es sich um ein Bereich innerhalb des Maschinellen Lernens (engl. machine learning), welches wiederum ein Bereich innerhalb der Künstlichen Intelligenz (engl. artificial intelligence) darstellt. In Abbildung 3 sind diese Beziehungen zu sehen.

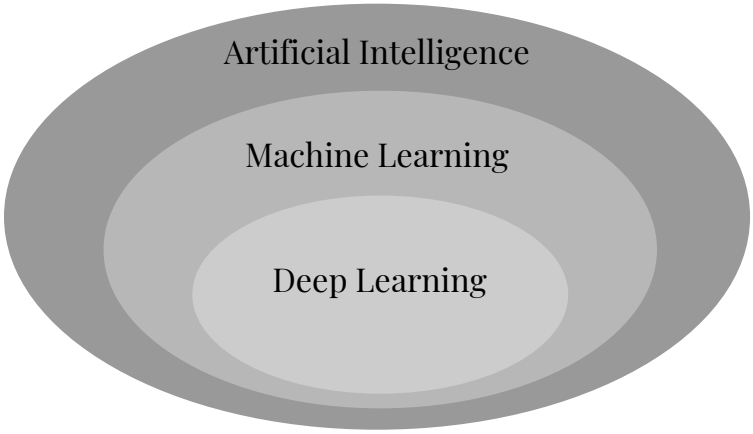


Abbildung 3: Die Beziehung zwischen Künstlicher Intelligenz, Maschinellem Lernen und Deep Learning (in Anlehnung an Gibson & Patterson 2017: S. 4)

Die Wurzeln von Deep Learning und neuronalen Netzwerken wurden durch die Biologie inspiriert. Dabei sind beispielsweise die in neuronalen Netzen eingesetzten Neuronen (engl. neurons), denen von biologischen Neuronen eines Gehirns nachempfunden. Ein häufig genanntes Kriterium, welches bereits seit den 1980er Jahren besteht, lautet, dass ein neuronales Netzwerk mehr als nur zwei Schichten haben muss, um als Deep learning angesehen zu werden. Allerdings haben sich in den Jahren die Architekturen und auch die Rechenleistung verändert. Einige Änderungen lauten wie folgt (Gibson & Patterson

2017: S. 3-6):

- Mehr Neuronen
- Komplexere Möglichkeiten Schichten und Neuronen zu verbinden
- Erhöhte Rechenleistung

Mehr Neuronen soll bedeuten, dass die Modelle komplexer geworden sind. In moderneren Modellen wie Convolutional Neural Networks (CNNs) oder Recurrent Neural Networks (RNNs), welche später erläutert werden, gibt es neben den dichten (engl. dense) Schichten noch andere Arten von Schichten. Durch diese neuen Architekturen gibt es auch einen Zuwachs an Parametern, welche dank der erhöhten Rechenleistung verarbeitet werden können.

3.2 Multilayer Feedforward Networks

Diese Art von Netzwerk hat verschiedene Namen. Sie werden als Multilayer Feedforward Networks, Deep Feedforward Networks, Feedforward Neural Networks oder auch Multi-Layer Perceptrons (MLPs) bezeichnet. Dabei handelt es sich um eine der fundamentalen Deep Learning Modelle, auf denen viele kommerzielle Anwendungen basieren. Der Begriff feedforward (dt. Vorwärtskopplung) stammt daher, dass die Informaionen in diesen Modellen nur vorwärts fließen (Gibson & Patterson 2017: S. 51-52). Die Abbildung 4 stellt ein solches Modell dar.

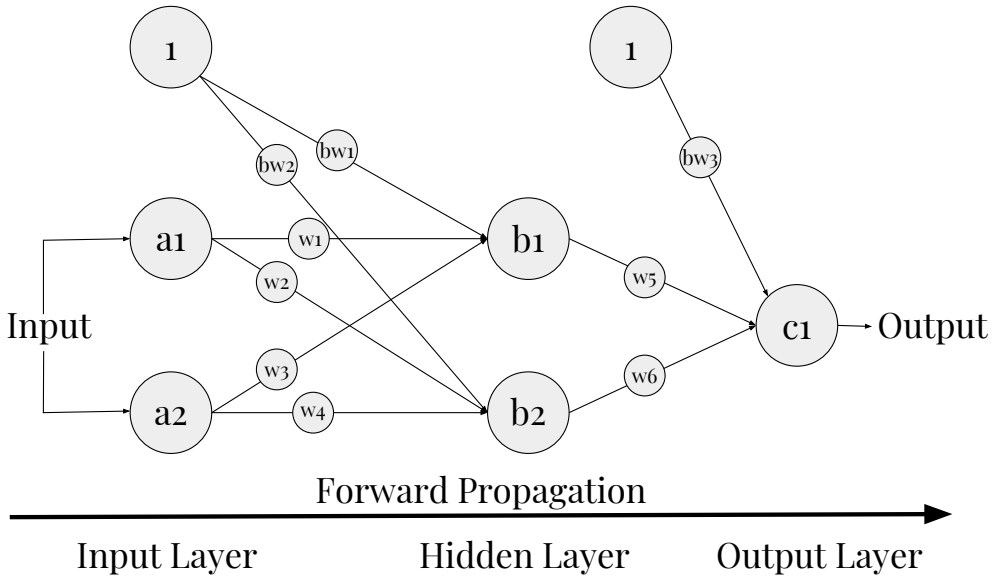


Abbildung 4: Eine Darstellung eines Multilayer Feedforward Networks und Forward Propagation (in Anlehnung an Taylor 2017). Dabei ist zu erkennen, dass es zwei Neuronen für die Eingabe gibt, eine versteckte Schicht mit ebenfalls zwei Neuronen und ein Neuron in der Ausgabeschicht. Zwischen den Schichten nehmen die Gewichtungen Einfluss auf die Berechnungen in den Neuronen. Informationen können nur vorwärts fließen.

In diesem Modell werden zwei Funktionen eingesetzt, welche in jedem Neuron Anwendung finden. Dabei handelt es sich zum einen um den Operator für die Summierung (engl. summation operator) und zum anderen eine Aktivierungsfunktion (engl. activation function). Der Operator für die Summierung wird in einem neuronalen Netzwerk alle Eingabe der Neuronen summieren und einen Input für das Netzwerk erstellen. Es folgt der Operator für die Summierung während der Vorwärtsausbreitung (engl. forward propagation). In dieser Formel steht das b für die Eingabe eines Bias-Neurons, das i stellt den Index der Summierung dar. Dabei wird mit 1 begonnen und endet bei n. Das X_i repräsentiert ein einzigartiges Neuron und W_i ein einzigartiges Gewicht welches sich an der Grenze des Neurons befindet (Taylor 2017).

$$input = b + \sum_{i=1}^{n} X_i W_i \tag{3}$$

Die Aktivierungsfunktion erhält die Ausgabe des Operators für die Summierung und transofrmiert diesen in den finalen Output eines Neurons. Ein Beispiel für eine Aktivierungsfunktion stellt die logistische Aktivierungsfunktion (Sigmoidfunktion) dar. In der folgenden Formel wird die dazugehörige Funktion dargestellt, wobei das x für die Eingabe eines bestimmten Neurons steht und e für die mathematische Konstante (Taylor 2017).

$$f(x) = \frac{1}{1 + e^{-x}} \tag{4}$$

3.3 Kapazität, Unteranpassung und Überanpassung

Eine der größten Herausforderungen für Machine Learning liegt darin, eine gute Leistung bei neuen und unbekannten Inputs bzw. Daten, zu erbringen. Die Eigenschaft, eine gute Leistung bei unbekannten Daten wird Generalisierung (engl. generalization) genannt. Durch das Training eines Machine Learning Modells mit einem Trainings-Datensatz, wird es möglich, den Trainingsfehler (engl. training error) zu messen. Ziel ist es, diesen zu reduzieren. Im Bereich Machine Learning kommt noch hinzu, dass auch der Generalisierungsfehler (engl. generalization error) und Testfehler (engl. test error) minimiert wird. Bei dem Generalisierungsfehler geht es um den erwarteten Wert von Fehler bezogen auf einen neuen und unbekannten Input (Goodfellow et al. 2016: S. 110).

Konzeptuell ist die Kapazität eines Deep Learning Modells sehr wichtig. Diese sollte der Komplexität der Aufgabe und den gegebenen Daten angemessen sein. Dabei kann eine zu kleine Kapazität dazu führen, dass komplexe Aufgabe nicht gut gelöst werden und tendieren somit zur Unteranpassung (engl. underfitting). Das Gegenteil tritt ein, wenn die Kapazität zu groß ist, dann neigt ein Modell zur Überanpassung (engl. overfitting). Somit lässt sich die Kapazität als eine Art Wert beschreiben, welcher für die Messung eines Modells zur Unter- oder Überanpassung eingesetzt werden kann (Rodriguez 2017).

Ein Modell, welches zur Unteranpassung neigt wird eine höhere Fehlerrate aufweisen und Daten nicht sehr gut klassifizieren. Das heißt, dass es zu vielen Fehlern und Fehlklassifikationen kommen kann. Bei der Überanpassung werden die vorhandenen Daten zu gut erlernt. So, dass für diese konkreten

Daten zwar geringe Fehler und gute Klassifizierungen resultieren. Auf unbekannten Daten wird diese Leistung allerdings nicht übertragbar sein. Somit weist das Modell außerdem eine schlechte Generalisierung auf und wird diese Leistung nicht bei unbekannten Daten erreichen. Zwischen diesen beiden Eigenschaften liegt die angemessene Kapazität des Modells (Gibson & Patterson 2017: S. 26-27). Die folgende Abbildung 5 stellt diese Verhalten noch einmal grafisch dar.

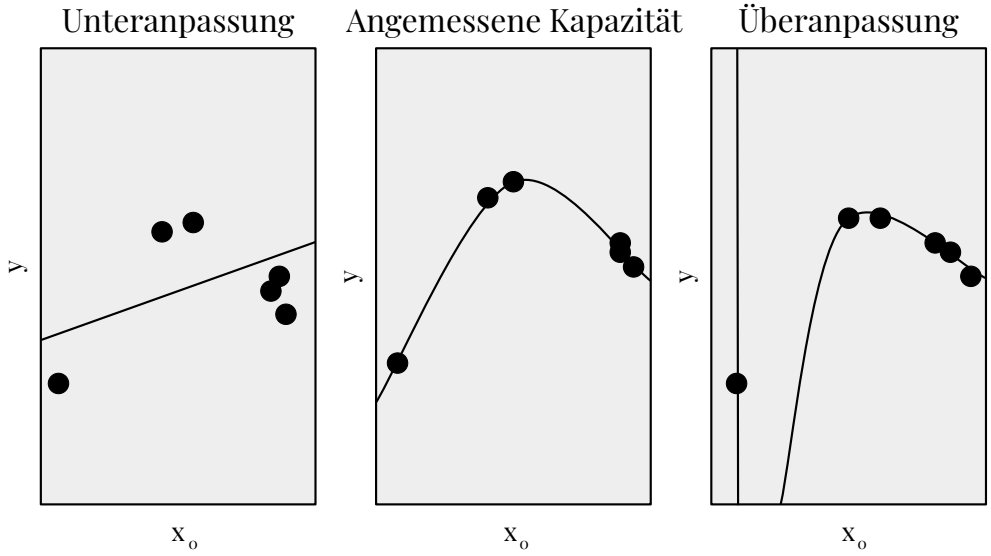


Abbildung 5: Grafische Darstellung von Unter- und Übernpassung sowie einer angemessenen Kapazität (in Anlehnung an Goodfellow et al. 2016: S. 113). Auf der linken Abbildung ist die Unteranpassung zu sehen. Eine Gerade, die zwischen den Datenpunkten verläuft. In der Mitte ist eine angemessene Kapazität zu sehen, wobei versucht wird effizient alle Datenpunkte zu treffen. Auf der rechten Seite versucht die Überanpassung jeden Datenpunkt mit zu hohem Aufwand, zu treffen.

3.4 Hyperparameter und Validierungsdaten

Ein Machine Learning Algorithmus hat verschiedene Einstellungen, welche Hyperparameter genannt werden. Durch dessen Nutzung kann das Verhalten von solchen Algorithmen kontrolliert und beeinflusst werden. Um diese Hyperparameter zu konfigurieren wird ein Datensatz für die Validierung ein-

gesetzt. Diese Daten werden von dem Algorithmus während dem Training nicht gesehen (Goodfellow et al. 2016: S. 121). Durch die Kreuzvalidierung (engl. cross-validation) soll geschätzt werden, wie gut ein Modell generalisiert. Dies wird mit einer Aufteilung der Trainingsdaten in N Splits aufgeteilt und in Trainings- und Validierungsdaten gruppiert. Nach (Gibson & Patterson 2017: S. 22) haben Forscher festgestellt, dass ein Split von $N = 10$ gut funktioniert, wobei allerdings keine feste Anzahl vorgeschrieben wird. Die folgende Abbildung 6 stellt die Aufteilung des gesamten Datensatzes in Trainings-, Validierungs- und Testdaten dar.

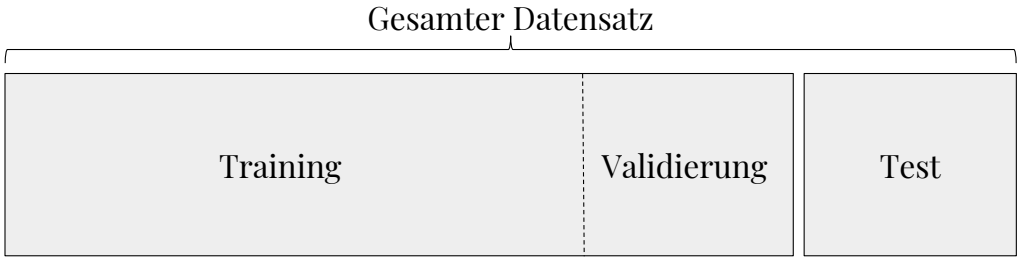


Abbildung 6: Abbildung zur Aufteilung des Datensatzes in Trainings-, Validierungs- und Testdaten. Dabei wurden 20% der gesamten Daten als Testdaten aus dem Datensatz herausgelöst und werden ausschließlich für die Tests eines trainierten und konfigurierten Modells eingesetzt. Die restlichen 80% des Datensatzes dienen für das Training, wobei 20% dieser Daten für die Validierung dienen.

Einige Hyperparameter sollen im folgenden erläutert werden. Dabei handelt es sich um jene Parameter die in der Implementierung eingesetzt wurden.

3.4.1 Anzahl und Größe der Schichten

Die Anzahl der Schichten wirkt sich auf die Architektur des Modells aus. Bei einem MLP kann man diesen Parameter beeinflussen, indem eine oder mehrere dichte Schichten hintereinander eingesetzt werden. Die Größe der Schichten leitet sich aus der Anzahl der Neuronen ab. In der Ein- und Ausgabeschicht sind diese Einstellungen leicht zu finden. Die Eingabeschischt benötigt beispielsweise die Anzahl von Neuronen, welche der Wortschatzgröße entspricht. Bei der Ausgabeschicht setzt sich die Anzahl der Neuronen aus der Anzahl der Klassen zusammen. Bei den übrigen Schichten wird es

schwerer die Anzahl von Neuronen zu bestimmen. Die effektivste Anzahl von Neuronen in den übrigen Schichten lässt sich beispielsweise durch Experimente herausfinden (Gibson & Patterson 2017: S. 100-101).

3.4.2 Aktivierungsfunktionen

Aktivierungsfunktionen werden eingesetzt um die Ausgabe von Neuronen einer Schicht vorwärts zu propagieren, zu der nächsten Schicht. Sie werden eingesetzt, um Nichtlinearität in ein neuronales Netzwerk zu implementieren (Gibson & Patterson 2017: S. 65-66). Ohne Aktivierungsfunktionen würden die dichten Schichten eines neuronalen Netzwerkes nur aus zwei lineare Operationen bestehen, einem Skalarprodukt (engl. dot product) und und einer Addition $output = dot(W, input) + b$. Wodurch folgendes Problem entstehen würde. Es können nur lineare Transformationen erlernt werden. Somit könnte der Hypothesenraum (engl. hypothesis space) nur lineare Transformationen umsetzen, was eine Begrenzung des Hypothesenraumes bedeutet. Auch durch die Stapelung von linearen Schichten würde den Hypothesenraum nicht vergrößern. Um diesen Raum auszuweiten, werden nicht-lineare Aktivierungsfunktionen eingesetzt (Chollet 2018: S. 72).

Ein gutes Beispiel hierfür stellt der Rectifier bzw. die gleichgerichtete lineare Einheit (engl. rectified linear unit - ReLU) dar. Nach (Goofellow et al. 2016: S. 173) handelt es sich hierbei um eine der Standardempfehlung für Aktivierungsfunktionen. Dabei wird ein Neuron aktiviert, wenn die Eingabe einen bestimmten Wert bzw. Menge übersteigt. So lange die Eingabe unter Null liegt, wird die Ausgabe ebenfalls Null sein. Übersteigt die Eingabe eine Schwelle (engl. threshold), ist eine lineare Beziehung zur abhängigen Variable $f(x) = max(0, x)$ zu erkennen. Dies lässt sich wie in Abbildung 7 darstellen (Gibson & Patterson 2017: S. 69).

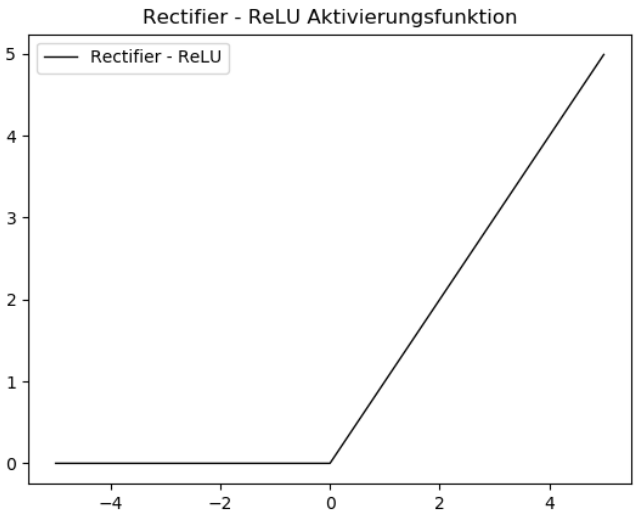
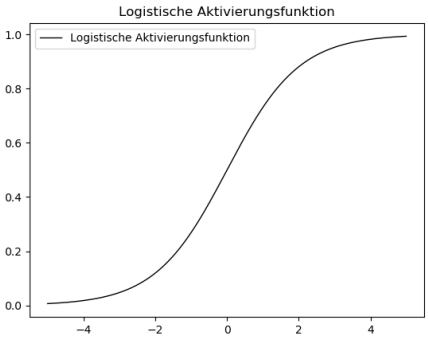


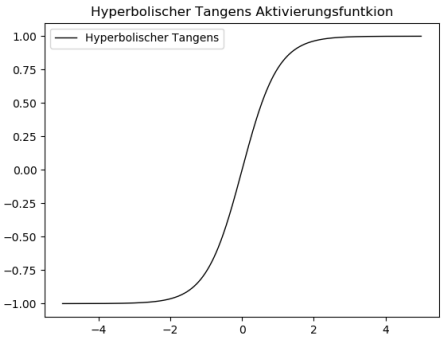
Abbildung 7: Visualisierung von ReLU. Es ist zu erkennen, dass anfänglich die Eingabewerte ≤ 0 bleiben und der Wert $= 0$ gesetzt. Sobald die Werte ≥ 0 werden, sind diese gleich der Eingabe.

Es gibt noch viele weitere Aktivierungsfunktionen. Zum Beispiel die bereits erwähnte logistische Aktivierungsfunktion oder der Hyperbolischer Tangens (engl. hyperbolic tangent function - tanh). Diese Funktion ist der logistischen ähnlich, außer, dass ihre Ausgaben zwischen -1 und 1 liegen (Taylor 2017). Die Formel für den Hyperbolischen Tangens lautet wie folgt, wobei x die Eingabe der Funktion darstellt. Also der Netzwerk-Input, welcher durch den Operator für die Summierung berechnet wird. Bei dem e handelt es sich um die mathematische Konstante. Anschließend werden diese beiden Aktivierungsfunktionen grafisch dargestellt.

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{5}$$



(a) Visualisierung von Sigmoid



(b) Visualisierung des Hyperbolischen Tangens

Abbildung 8: Auf der linken Seite ist die grafische Darstellung für eine logistische Aktivierungsfunktion zu erkennen. Dabei ist sieht man, wie Variablen in Werte zwischen 0 und 1 konvertiert werden. Auf der rechten Seite ist der Hyperbolische Tangens dargestellt. Es ist eine große Ähnlichkeit festzustellen. Der Unterschied liegt darin, dass hier die Ergebnisse zwischen -1 und 1 liegen.

3.4.3 Kostenfunktion und Kreuzentropie

Die Kostenfunktion (engl. cost function), auch Verlustfunktion (engl. loss function) oder Zielfunktion (engl. objective function) stellt den Wert dar, welcher während des Trainings minimiert werden soll. Er repräsentiert einen Messwert für den Erfolg des Netzwerkes bei der Aufgabenlösung. Für Klassifizierungen mit mehreren Klassen ist die Kategorische Kreuzentropie (engl. categorical crossentropy) die richtige Kostenfunktion (Chollet 2018: S. 60).

Um die Kreuzentropie besser beschreiben zu können, dient folgendes Beispiel. Das Ziel ist es ein Neuron mit mehreren Eingabe-Variablen zu trainieren. Diese können durch $x_1, x_2, \ldots$ repräsentiert werden sowie die dazugehörigen Gewichtungen $w_1, w_2, \ldots$ und dem Bias b. Dies lässt sich wie auf Abbildung 9 darstellen (Nielsen 2018).

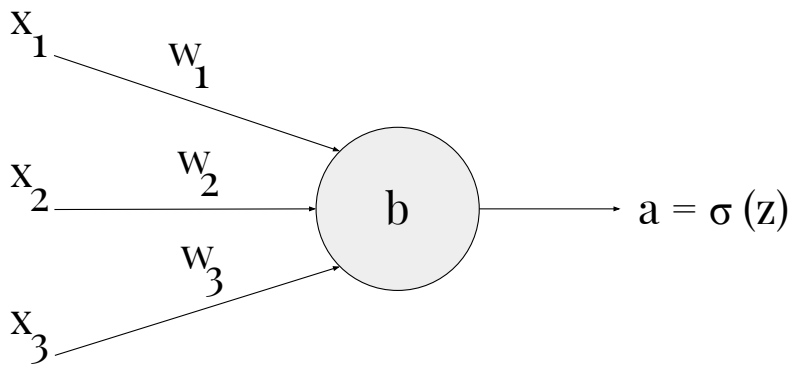


Abbildung 9: Beispiel für ein Neuron, welches durch mehrere Eingabe-Variablen trainiert wird. Bei x handelt es sich um die Eingaben, w stellt die dazugehörigen Gewichtungen dar und b den Bias. Die Ausgabe wird durch $a = \sigma(z)$ definiert (in Anlehnung an Nielsen 2018).

Die Ausgabe des Neurons lautet $a = \sigma(z)$, wobei $z = \sum_j w_j x_j + b$ die gewichtete Summe der Eingaben darstellt. Die Kostenfunktion der Kreuzentropie für dieses Neuron lässt sich mit folgender Formel beschreiben. Dabei steht n für die Gesamtzahl der Trainingsdaten, die Summe gilt über alle Eingaben für das Training sowie x und y die entsprechenden, gewünschten Ausgaben (Nielsen 2018).

$$C = -\frac{1}{n} \sum_x [y \ln a + (1 - y) \ln(1 - a)] \tag{6}$$

Nach (Nielsen 2018) sind zwei Eigenschaft der Grund, warum die Kreuzentropie als Kostenfunktion angesehen werden kann. Zum einen, weil diese nicht-negativ ist, da $C > 0$ ist. Der zweite Grund liegt darin, dass wenn die Ausgabe des Neurons nahe der gewünschten Ausgabe für alle Trainingseingaben x liegt, wird die Kreuzentropie nahe 0 sein. Dies ist der Fall, wenn die gewünschte Ausgabe y beispielsweise 0 oder 1 sind, wie bei einem Klassifizierungsproblem. Zusammenfassend lässt sich sagen, dass die Kreuzentropie

positiv ist und gegen Null tendiert, wenn ein Neuron besser geworden ist, in der Berechnung der gewünschten Ausgabe y, für alle Trainingseingaben x.

3.4.4 Optimierungsalgorithmen

Die meisten aller Deep Learning Modelle benötigen Optimierungsalgorithmen. Dabei ist die Aufgabe gemeint, eine Funktion $f(x)$ zu minimieren oder maximieren, wobei sich x ändern kann. Die meisten Optimierungsprobleme haben das Ziel $f(x)$ zu minimieren. Mittels dieser Funktion soll die Zielfunktion minimiert werden, auch Kostenfunktion genannt (Goodfellow et al. 2016: S. 82).

Ein wichtiger Bestandteil stellt der Gradientenabstieg (engl. gradient descent) dar. Dieser nimmt Einfluss af die Qualität der Vorhersagen eines neuronalen Netzwerkes. Indem er die aktuelle Steigung bezüglich der Gewichtungen erkennen kann. Dabei kann die Steigung gemessen werden und die Gewichtungen angepasst. Dies wird ermöglicht, wenn ein Derivat der Kostenfunktion den Gradienten produziert. Somit gibt der Gradient den nächsten Schritt vor, welchen der Optimierungsalgorithmus ausführen soll. In Abbildung 10 wird dies grafisch dargestellt (Gibson & Patterson 2017: S. 30).

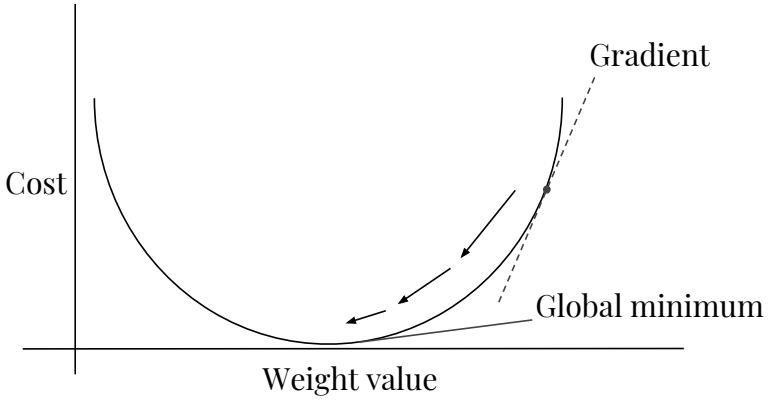


Abbildung 10: Auf der Y-Achse sind die Werte der Kostenfunktion und auf der X-Achse die, der Gewichtungen. Dabei ist der Verlauf in einer U-förmigen Kurve dargestellt. Am niedrigsten Punkt dieser Kurve liegt das globale Minimum, also der niedrigste Wert für die Kostenfunktion sowie die dazugehörigen Gewichtungen. Wird der Gradient gemessen bzw. berechnet, so werden idealerweise Schritte unternommen, welche das globale Minimum erreichen. (in Anlehnung an Gibson & Patterson 2017: S. 30).

Ein Optimierungsalgorithmus ist der stochastische Gradientenabstieg (engl. stochastic gradient descent - SGD). Hierbei handelt es sich um einen Optimierungsansatz, welcher die Gewichtungen eines neuronalen Netzwerkes anpasst. Ziel ist es, die beste Kombination aus Gewichtungen zu finden, welche den niedrigsten Wert für die Kostenfunktion erbringen (Gibson & Patterson 2017: S. 98). Dies kann mittels folgender Gleich gelöst werden: Gradient $(f)(W) = 0$ für W. Dabei handelt es sich um eine Polynomgleichung mit N Variablen, wobei N für die Anzahl der Koeffizienten des Netzwerks steht. SGD versucht die Parameter für jeden Datensatz $x^{(i)}$ und das Label $y^{(i)}$, zu updaten.

$$\theta = \theta - \alpha \nabla_\theta J(\theta; x^{(i)}; y^{(i)}) \tag{7}$$

Weiterhin gibt es noch den Mini-Batch Gradientenabstieg, wobei dieser auf Basis des Mini-Batches bestehend aus n Trainingsdaten, ein Update durchführt (Ruder 2019).

$$\theta = \theta - \alpha \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)}) \tag{8}$$

Dadurch kann beispielsweise die Varianz der Parameter-Updates reduziert werden, was zu einer stabileren Konvergenz führen soll. Dabei handelt es sich um die Parameterfindung für kleinstmögliche Ergebnisse der Kostenfunktion, dem so genannten loss (Ruder 2019).

Ein weiterer Algorithmus heißt Adagrad, wobei es sich um einen Gradientenbasierten Optimierungsalgorithmus handelt. Dieser Ansatz stammt von (Duchi et al. 2011). Er adaptiert die Lernrate beispielsweise indem er sie verkleinert, bei Parametern, welche häufig vorkommenden Features zugeordnet sind und kann sie erhöhen, bei welchen, die nicht so häufig vorkommen. In der Arbeit von (Dean et al. 2012) wurde festgestellt, dass Adagrad die Robustheit von SGD erhöht und sie setzten diesen Ansatz in großen neuronalen Netzen bei Google ein (Ruder 2019). Bisher wurden Updates für alle Parameter θ bei jedem Parameter θ_i umgesetzt, wobei die Lernrate α dieselbe geblieben ist. Dabei setzt Adagrad eine andere Lernrate für jeden Parameter θ_i, zum Zeitschritt t. Zunächst folgt die Darstellung von Adagrad, wie die Parameter ein Update erfahren und dann vektorisiert werden. Dabei soll g_t für einen Gradienten zum Zeitpunkt t stehen. Somit ist $g_{t,i}$ die partielle Ableitung für die Zielfunktion bezugnehmend auf den Parameter θ_i, am Zeitpunkt t.

$$g_{t,i} = \nabla_\theta J(\theta_{t,i}) \tag{9}$$

Das Update durch SGD für jeden Parameter θ_i zum Zeitpunkt t wird zu:

$$\theta_{t+1,i} = \theta_{t,i} - \alpha g_{t,i} \tag{10}$$

Mittels der Update-Regel von Adagrad wird die Lernrate α am Zeitschritt t, für jeden Parameter θ_i basierend auf den zuletzt berechneten Gradienten für θ_i geändert (Ruder 2019).

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\alpha}{\sqrt{G_{t,ii} + \epsilon}} g_{t,i} \tag{11}$$

Hier stellt $G_t \in \mathbb{R}^{d \times d}$ eine diagonale Matrix dar, bei der jedes diagonale Element i, i die Summe der Quadrate der Steigungen darstellen. Bezugnehmend auf θ_i am Zeitpunkt t, wobei ϵ dabei hilft, die Division durch Null zu vermeiden. Weiterhin beinhaltet G_t die Summe der Quadrate der vergangenen Gradienten von allen Parameter θ entlang der Diagonalen. Es folgt die Vektorisierung, indem ein Matrix-Vektor-Produkt $\odot$ zwischen G_t und g_t gebildet

wird (Ruder 2019):

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\alpha}{\sqrt{G_t + \epsilon}} \odot g_t \tag{12}$$

Ein Vorteil von Adagrad nach (Ruder 2019) besteht darin, dass die Lernrate nicht manuell angepasst werden muss. Oft wird ein Wert von $0,01$ ausgewählt.

Ein Weiterer Optimierungsalgorithmus ist RMSprop, welcher versucht die zu stark abnehmende Lernrate in Adagrad zu beheben. Hierbei handelt es sich um einen unveröffentlichten Ansatz, welcher auf eine Vorlesung von (Hinten) zurückgeht.

$$MeanSquare(w,t) = 0,9MeanSquare(w,t-1) + 0,1(\partial E/\partial w^{(t)})^2 \tag{13}$$

Der letzte Optimierungsalgorithmus wird Nesterov-accelerated Adaptive Moment Esitmation (Nadam) genannt. Dieser Algorithmus kombiniert Nesterov accelerated gradient (NAG) mit dem Optimierungsalgorithmen Adam. Nach (Ruder 2018)lässt sich das Update mittels Nadam wie folgt darstellen:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon}(\beta_1 \hat{m}_t + \frac{(1-\beta_1)g_t}{1-\beta_1^t}) \tag{14}$$

3.4.5 Lernrate und Momentum

Bei der Lernrate (engl. learning rate) handelt es sich um einen Parameter, welcher die Optimierung beeinflusst. Während dem Einsatz von SGD werden nach der Berechnung des Fehlergradienten die Gewichtungen des Modells aktualisiert, was durch die Backpropagation möglich ist. Die Größe des Updates während des Trainings wird als Lernrate bezeichnet. Somit stellt die Lernrate einen konfigurierbaren Hyperparameter dar, welcher meistens im Bereich zwischen $0,0$ und $0,1$ liegt. Dieser Parameter nimmt Einfluss darauf, in welcher Geschwindigkeit das Modell lernt (Brownlee 2019). In der folgenden Abbildung 11 ist dargestellt, wie die Lernrate den Loss beeinflussen kann.

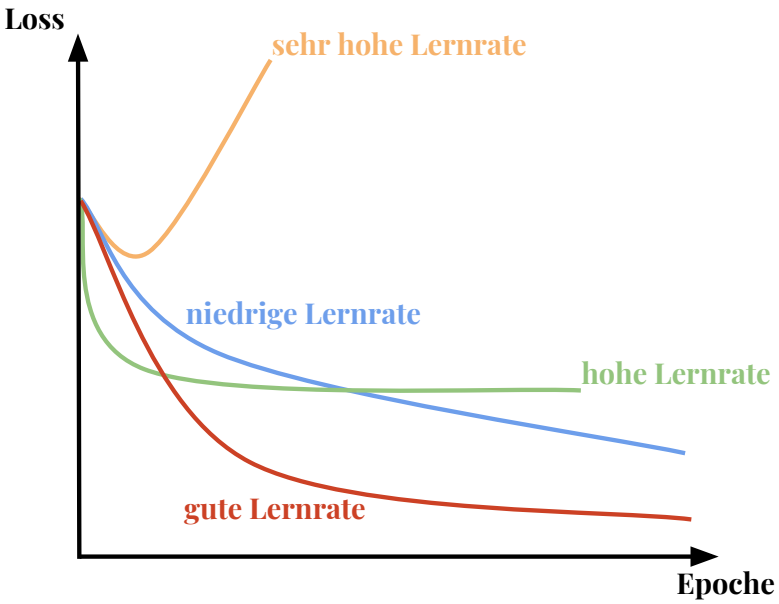
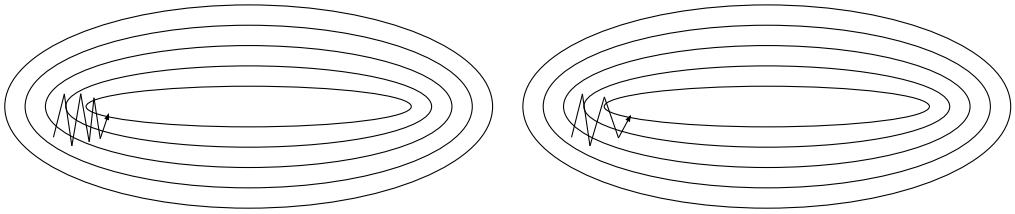


Abbildung 11: Auf dieser Abbildung ist der Einfluss der Lernrate auf den Loss dargestellt. Dabei wirkt sich eine sehr hohe Lernrate (orange) sehr negativ auf den Loss aus. Dieser sinkt nur kurz und steigt danach steil an. Mit einer hohen Lernrate (grün) ist zu sehen, dass der Loss sehr schnell sinkt, dann aber gleich bleibt. Mit einer niedrigen Lernrate (blau) sinkt der Loss langsamer aber kann einen noch niedrigeren Wert als zuvor erreichen. Eine gute Lernrate (rot) lässt den Loss nicht zu schnell, aber auch nicht zu langsam sinken und somit die beste Minimierung des Wertes für den Loss erreichen. (in Anlehnung an Karpathy 2019).

Momentum soll einem Algorithmus dabei helfen, aus Bereichen herauszukommen, in denen dieser sonst stecken bleiben würde. Somit sollen die Updates zu einem Minima führen. Das macht Momentum für die Lernrate so wichtig, wie die Lernrate für die Gewichtungen ist (Gibson & Patterson 2017: S. 79). Auf Abbildung 12 ist zu sehen, wie sich SGD ohne Momentum und mit Momentum verhält.



(a) SGD ohne Momentum (b) SGD mit Momentum

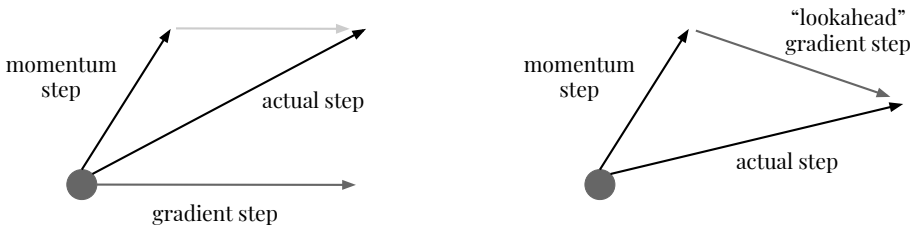
Abbildung 12: Es wird SGD ohne Momentum (links) und SGD mit Momentum (rechts) dargestellt. Dabei hat SGD ohne Momentum Schwierigkeiten Bereiche, die tiefer sind, welche um ein lokales Optima existieren, zu navigieren. Dabei bewegt sich SGD nur langsam zum Optimum. Mit Momentum soll SGD beschleunigt und diese Schwankungen oder Sprünge gedämpft werden. (in Anlehnung an Ruder 2019).

Ein Update mittels Momentum lässt sich durch die Folgende Formel darstellen. Wobei v die aktuelle Geschwindigkeit des Vektors darstellt, der die gleichen Dimensionen wie der Parametervektor θ hat. Die Lernrate stellt α dar und $\gamma \in (0, 1]$ wie viele Iterationen der vorherigen Gradienten, in ein aktuelles Update einbezogen werden.

$$v = \gamma v + \alpha \nabla_\theta J(\theta; x^{(i)}, y^{(i)}) \tag{15}$$

$$\theta = \theta - v \tag{16}$$

Das Nesterov Momentum stellt eine andere Version zum zuvor dargestellten Momentum dar. Nach (Karpathy 2019) ist die Popularität zuletzt gestiegen und in der Praxis scheint es etwas besser zu funktionieren als das Momentum. Nach (Goodfellow et al. 2016: S. 300) besteht der Unterschied in diesen Beiden Arten von Momentum darin, dass der Gradient an einer unterschiedlichen Stelle ausgewertet wird. Auf Abbildung 13 ist ein Update mit Momentum dargestellt und eines mit Nesterov Momentum.



(a) Momentum Update (b) Nesterov Momentum

Abbildung 13: Auf der linken Seite ist ein Update mit Momentum dargestellt und auf der rechten Seite eines mit Nesterov Momentum. Der Unterschied besteht darin, dass der Gradient nicht an der aktuellen Position (dem Kreis) ausgewertet wird. Das Momentum navigiert in die Richtung des Pfeils (momentum step). Daher wird mit Nesterov Momentum der Gradient an der llookaheadPPosition auswertet (in Anlehnung an Karpathy 2019).

3.5 Regularisierung im Deep Learning

Ein bereits erwähntes Problem im Deep Learning stellt der Umstand dar, dass ein Algorithmus nicht nur an Trainingsdaten, sondern auch mit unbekannten Daten gut umgehen können soll (Goodfellow et al. 2016: S. 228). Ein Ziel von Regularisierung (engl. regularization) ist somit das Bekämpfen von Überanpassung und eine bestmögliche Generalisierung des Modells (Gibson & Patterson 2018: S. 104). In den folgenden Kapitel sollen einige Strategien der Regularisierung erläutert werden.

3.5.1 Early Stopping

Bei dem Training von großen Modellen mit dem Problem von Überanpassung, kann häufig auffallen, dass der Trainings-Fehler stetig abnimmt und der Validierungs-Fehler wieder beginnt zu steigen. Somit wird es möglich, ein Modell zu erhalten, was einen besseren Validierungs-Fehler (vielleicht auch Test-Fehler) besitzt, wenn die Parameter von dem Zeitpunkt genommen werden, an dem der Validierungs-Fehler am niedrigsten war. Dies kann man umsetzen, indem das Modell ausgeführt wird, bis der Fehler der Validierung für eine gewisse Zeit nicht verbessert wurde. Dank der Effektivität und Einfachheit, zählt Early Stopping (dt. frühes Anhalten) zu einer weit-

verbreiteten Strategie von Regularisierung in Deep Learning (Goodfellow et al. 2016: S. 246).

3.5.2 Dropout

Bei Dropout (dt. aussteigen) handelt es sich um eine rechnerisch günstige und leistungsstarke Methode, um verschiedene Modelle regularisieren zu können. Eine Untersuchung zum Thema Dropout bietet (Srivastava et al. 2014), mit dem Titel *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*. Allgemein lässt sich Dropout als praktische Methode für das aufteilen von Ensembles sehr großer neuronaler Netzwerke verstehen. Dabei beinhaltet dieses aufteilen das trainieren mehrerer Modelle sowie der Evaluation von mehreren Modellen an jedem Testbeispiel. Dies wird als unpraktisch betrachtet, wenn jedes Modell ein großes neuronales Netzwerk darstellt. Da das Training und die Evaluation solcher Netzwerke lange Laufzeiten und viel Speicherplatz benötigen (Goodfellow et al. 2016: S. 258).

Dropout trainiert ein Ensemble, welches aus allen Subnetzwerken (engl. subnetworks) gebildet wird. Hierfür werden alle Nicht-Ausgabe Einheiten entfernt, von einem darunter liegenden Basis-Netzwerk. In vielen modernen neuronalen Netzwerken kann eine Einheit durch die Multiplikation der Ausgabe mit dem Wert Null, aus einem Netzwerk entfernt werden. Wobei dies sehr einfach ausgedrückt ist (Goodfellow et al. 2016: S. 258).

Wie zuvor beschrieben, wird mittels Dropout ein Ensemble trainiert, welches alle Sub-Netzwerke beinhaltet. Zur näheren Erläuterung zu Dropout dient die Abbildung 14. Ein solches Ensemble entsteht und kann trainiert werden, indem Nicht-Ausgabe Einheiten von einem zugrundeliegenden Basis-Netzwerk entfernt werden (Goodfellow et al. 2016: s. 2060).

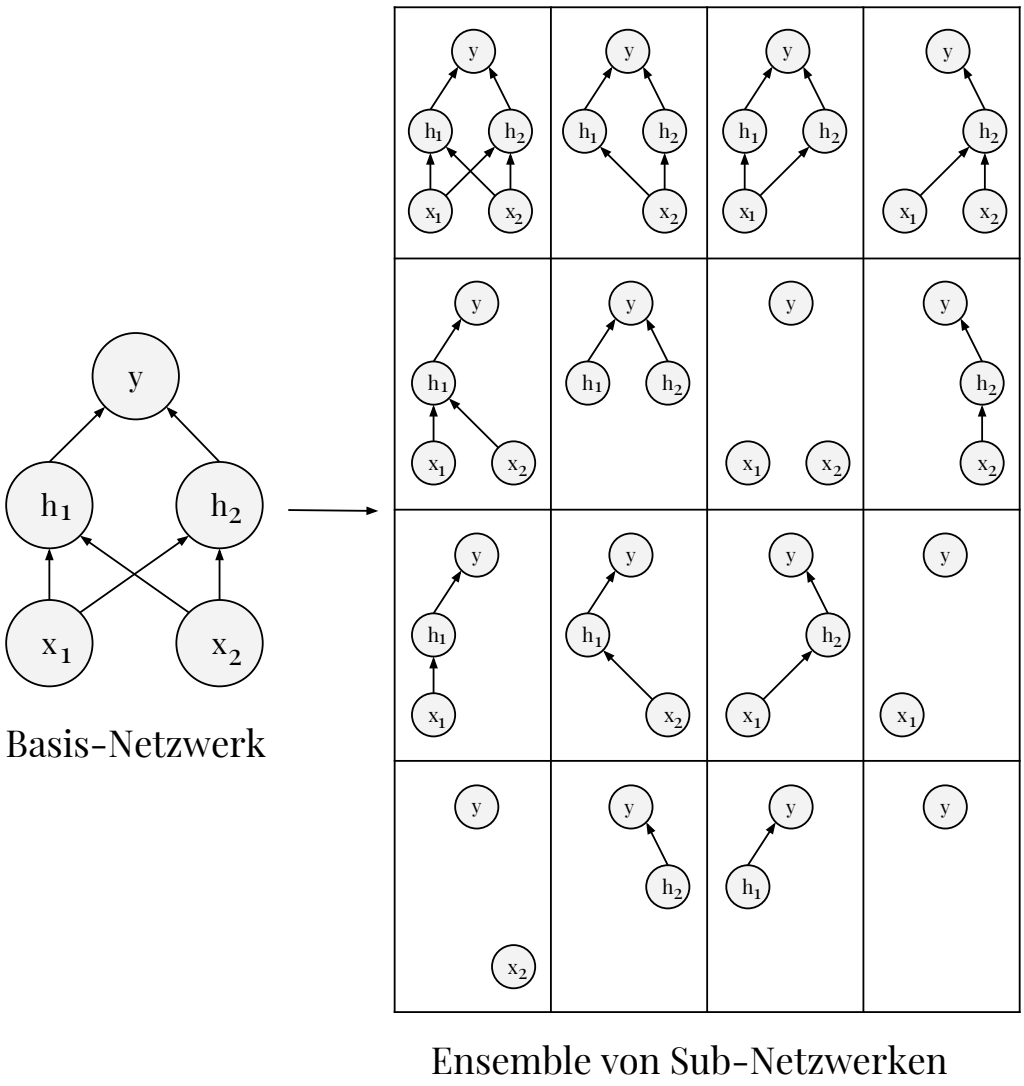


Abbildung 14: Auf der Abbildung sieht man auf der linken Seiten zunächst das Basis-Netzwerk. Dieses Basis-Netzwerk besteht aus zwei Eingabe- und versteckte Schichten, mit je zwei Neuronen und einem Ausgabe-Neuron. Daher sind für das gegebene Basis-Netzwerk sechzehn Sub-Netzwerke möglich. Auf der rechten Seite sind diese sechzehn Sub-Netzwerke dargestellt, wobei bei jedem Netzwerk andere Kombinationen von Schichten und Neuronen eingesetzt werden (in Anlehnung an Goodfellow et al. 2016: S. 260).

3.5.3 Regularisierung mittels L1 und L2

Weitere Möglichkeiten von Regularisierung stellen L1 und L2 (auch Gewichtsverfall - engl. weight decay genannt) dar. Dabei handelt es sich um Normen, mit denen die Größe von Vektoren gemessen werden kann. Dabei basiert L^2 auf L^p, wobei $p \in \mathbb{R}, p \geq 1$ (Goodfellow et al. 2016: S. 39).

$$\|x\|_p = \Big(\sum_i |x_i|^p \Big)^{\frac{1}{p}} \tag{17}$$

Dabei stellen diese Normen, Funktionen, dar, die Vektoren Nicht-Negativen Werten zuordnen. Außerdem verfolgen L^1 und L^2 das Ziel, die Parameter eines neuronalen Netzes nicht zu groß werden zulassen, in eine Richtung. Große Gewichtungen sollen verkleinert werden (Gibson & Patterson 2017: S. 104). Wenn es wichtig ist, zwischen Elementen zu unterscheiden, die genau Null und klein, aber nicht Null sind. Dann wird L^1 angewandt. Da es sich hierbei um eine Funktion handelt die in alle Richtungen gleich schnell wächst, dabei aber eine mathematische Einfachheit beibehält. Formell lässt sich diese Norm wie folgt beschreiben (Goodfellow et al. 2016: S. 40).

$$\|x\|_1 = \sum_i |x_i| \tag{18}$$

Die Anwendung von L^1 ist sinnvoll, wenn die Differenz zwischen Null und Nicht-Null Elementen sehr wichtig ist. Die L^2 Norm ist L^p, wobei $p = 2$ ist und wird auch euklidische Norm genannt. Dabei handelt es sich um die euklidische Entfernung vom Ursprung zum Punkt x (Goodfellow et al. 2016: S. 39 - 40). L^2 ist rechnerisch effizienter als L^1 (Gibson & Patterson 2017: S. 104).

3.6 Convolutional Neural Networks

Bei Convolutional Neural Networks (CNNs) handelt es sich um eine spezielle Form von neuronalen Netzwerken für das Verarbeiten von Daten, welche eine gitterartige Topologie aufweisen. Dabei können Datenbeispiele, Zeitreihendaten darstellen, welche ein 1D-Gitter sind oder Bilddaten, die als 2D-Gitter von Pixeln verstanden werden können. Der Name CNN lässt darauf schließen, dass durch das Netzwerk eine mathematische Operation, die Faltung (engl. convolution) durchgeführt wird. Dabei handelt es sich um eine spezielle Art von linearer Operation. Weiterhin lässt sich ein CNN dadurch beschreiben,

dass es diese Faltung einsetzt, wo andere Netzwerke Matrix Multiplikationen durchführen. Dies sollte mindestens in einer Schicht vorkommen (Goodfellow et al. 2016: S. 330).

3.6.1 Architektur

CNNs wandeln Eingabedaten einer Eingabeschicht über alle verbundenen Ebenen in eine Reihe von Klassenbewertungen (engl. class scores) um, welche durch die Klassifikationsschicht gegeben sind. Es gibt verschiedene Variationen von CNN, welche alle auf die Architektur in Abbildung 15 basieren. Dabei ist immer eine Eingabeschicht notwendig, worauf die Feature-Extraktionsebenen (engl. feature-extraction layers) sowie den Klassifikationsschichten (engl. classification layers).

Die Eingabeschicht kann beispielsweise Dreidimensionale Eingaben in Form von Bildern entgegennehmen. Dabei wird die Größe (engl. size) als Breite $\times$ Höhe und eine Tiefe (engl. depth), welche die Farbkanäle repräsentiert (z. B. drei für RGB-Farbkanäle). Anschließend kommen die Feature-Extraktionsschichten, welche durch sich wiederholende Schichten bestehen. Dazu gehören die Faltungsschichten (engl. convolutional layers), die Aktivierungsfunktion ReLU sowie den Pooling-Schichten (engl. pooling layers). Diese Schichten finden eine Anzahl von Features, welche zunehmend Features höherer Ordnung konstruieren. Danach folgt Klassifikationsschicht, in welcher eine oder mehrere vollständig verbundene Schichten existieren. Diese nimmt die Features der höheren Ordnung und produziert Klassenwahrscheinlichkeiten sowie Ergebnisse. Die Klassifikationsschichten sind mit allen Neuronen der vorherigen vollständig verbunden. Die Ausgabe der letzten Schicht ist meistens ein Zweidimensionaler Output, mit den Dimensionen $[b \times N]$, wobei b für die Anzahl der Datenbeispiele im Mini-Batch und N die Anzahl der Klassen darstellen (Gibson & Patterson 2017: S. 128-129).

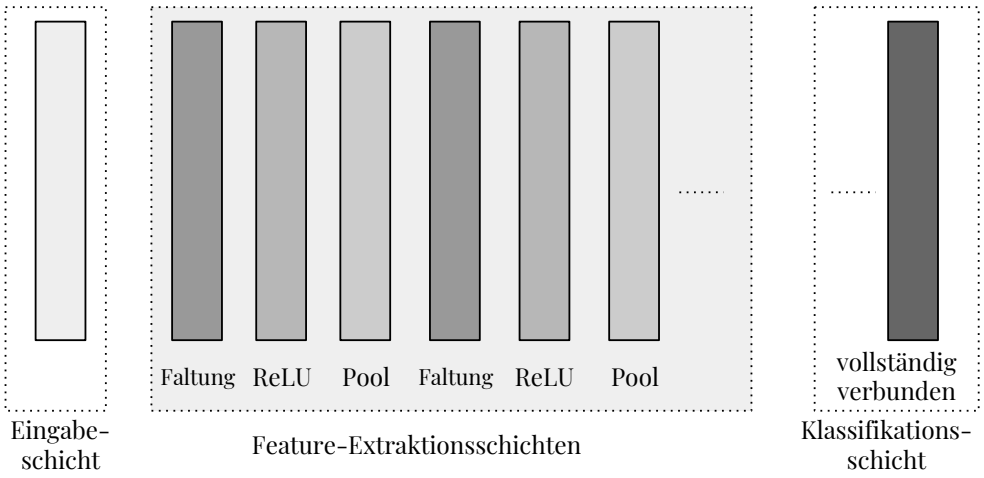


Abbildung 15: Hier ist die allgemeine Architektur eines CNNs dargestellt. Dabei sind die Schichten in drei Bereiche unterteilt. Zunächst wird die Eingabeschicht dargestellt. Danach folgen die Feature-Extraktionsschichten, welche aus den drei dargestellten Schichten bestehen und sich wiederholen können. Dabei ist hier zunächst eine Faltungsschicht, dann eine Schicht in der die Aktivierungsfunktion angewendet wird und anschließend eine Pooling-Schicht. Als letztes folgt die Klassifikationsschicht die außer vollständig verbundenen Schichten besteht sowie einer Ausgabeschicht, welche die Klassifizierungen ausgibt (in Anlehnung an Gibson & Patterson 2017: S. 128).

3.6.2 Eingabeschicht

Die Eingabeschicht wird eingesetzt, um die Eingabedaten zu laden und speichern. Im Falle eines Bilddatensatzes würden die Eingabedaten aus der Breite, Höhe und Anzahl von Kanälen bestehen. Hierbei wäre die Anzahl typischerweise drei, für die RGB-Werte der Pixel. Auf Abbildung 16 ist Eingabeschicht in 3D abgebildet (Gibson & Patterson 2017: S. 130).

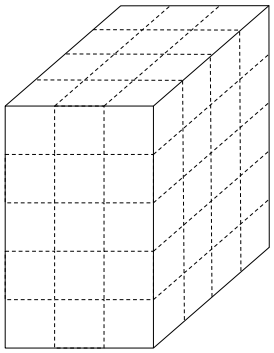


Abbildung 16: Dies ist eine grafische Darstellung der Eingabeschicht in einer 3-Dimensionalen Ansicht. Dabei handelt es sich um die zuvor erwähnten Eingabedaten für eine Vielzahl von Datenbeispielen (in Anlehnung an Gibson & Patterson 2017: S. 130).

3.6.3 Faltungsschicht

Faltungsschichten stellen eine der wichtigsten Kernelemente CNN Architekturen dar. Wie auf Abbildung 17 dargestellt, transformieren Faltungsschichten die Eingabedaten. Dies geschieht, indem ein Patch von lokal verbundenen Neuronen aus der vorherigen Schicht ausgewählt wird. Dann wird ein Skalarprodukt zwischen der Region von Neuronen aus der Eingabeschicht und den Gewichtungen der Verbindung zur Ausgabeschicht, gebildet. Die resultierende Ausgabe hat allgemein dieselbe räumliche Dimension oder kleinere. Es kann auch vorkommen, dass die Anzahl der Elemente in der dritten Dimension der Ausgabe ansteigt (Gibson & Patterson 2017: S. 130-131).

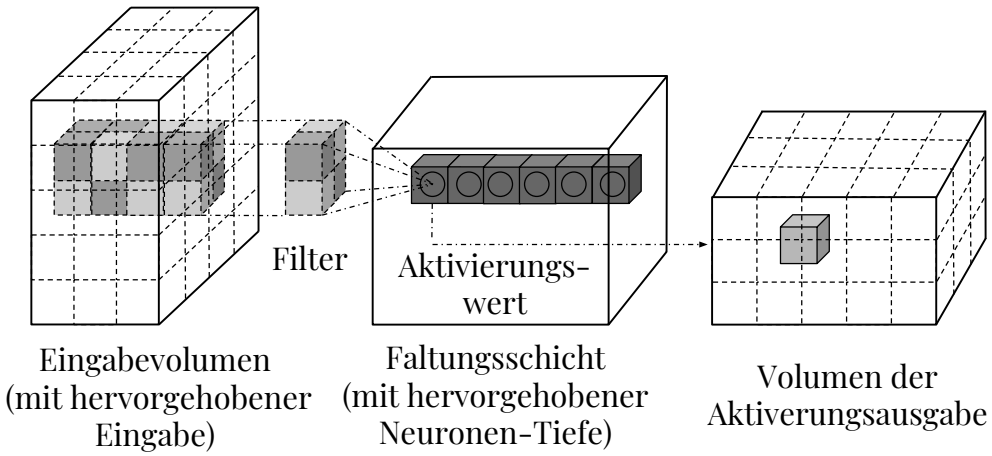


Abbildung 17: Hier wird die Falttungsschicht mit Eingabe- und Ausgabevolumen grafisch dargestellt. Dabei wird das Eingabevolumen durch eine Filterung in die Faltungsschicht überführt, wo der Aktivierungswert ermittelt wird und das Volumen der Aktivierungsausgabe resultiert (in Anlehnung an Gibson & Patterson 2017: S. 131).

Eines der Kernkonzepte der Faltungsschicht ist die Faltung. Dabei ist eine Faltung eine definierte mathematische Operation, welche die Regeln beschreibt, für das Verschmelzen zweier Sätze von Informationen. Dies spielt auch in der Physik und Mathematik eine große Rolle und definiert eine Brücke zwischen der Raum-Zeit Domäne und dem Gebiet der Frequenzen, durch den Einsatz von Fourier-Transformationen (Gibson & Patterson 2017: S. 131).

Die Faltungs-Operation wird auf der Abbildung 18 dargestellt. Sie wird auch Feature-Detektor (engl. feature detector) eines CNNs genannt. Der Input für eine Faltung können Daten oder eine Feature-Map-Ausgabe (engl. feature map output) von einer anderen Faltung. Man könnte dies als Filter interpretieren, in dem der Kernel die Eingabedaten nach bestimmten Informationen filtert (Gibson & Patterson 2017: S. 131).

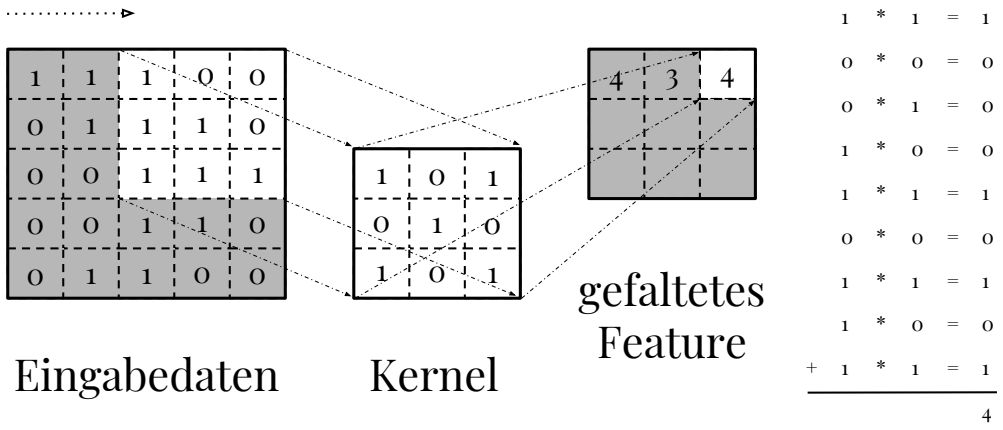


Abbildung 18: Die Abbildung zeigt, wie der Kernel über die Eingabedaten geschoben wird, um die Ausgabe, das gefaltete Feature (engl. convoluted feature), zu erstellen. Bei jedem Schritt wird der Kernel mit den Werten der Eingabedaten innerhalb seiner Grenzen multipliziert, wodurch ein einzelner Eintrag in der Feature-Map-Ausgabe (engl. output feature map) getätigt wird. Die Gewichtungen in einer Faltungsschicht werden auch als Filter oder Kernel bezeichnet. Der Filter wird mit dem Input gefaltet und es resultiert eine Feature-Map (engl. feature map) oder Aktivierungs-Map (engl. activation map). Faltungsschichten führen eine Transformation am Input Datenvolumen durch. Dabei wird die Aktivierungs-Map für jeden Filter an der Dimension der Tiefe gestapelt, um dadurch das 3D Output Volumen zu generieren. (in Anlehnung an Gibson & Patterson 2017: S. 131).

Faltungsschichten besitzen Parameter für die verschiedenen Schichten sowie zusätzliche Hyperparameter. Der Gradientenabstieg findet Anwendung, um Parameter so zu trainieren, dass die Ergebnisse (engl. scores) von Klassen mit den Labels im Trainingsdatensatz übereinstimmen. Außerdem gibt es in den Faltungsschichten einige relevante Komponenten (Gibson & Patterson 2017: S. 132):

- Filter
- Aktivierungs-Map
- Hyperparameter der Faltungsschicht

Die Parameter einer Faltungsschicht sind für die Konfiguration der Filter von Schichten zuständig. Dabei kann die Filtergröße dem Eingabevolumen entsprechen, wobei diese normalerweise Eindimensional ist. Dies wird beispielsweise relevant, bei dem Einsatz von CNNs im Bereich NLP. Filter, wie zum Beispiel Faltungen werden über die Breite und Höhe des Eingabevolumens angewendet, dies wurde in Abbildung 18 dargestellt. Dabei werden Filter für jede Tiefe des Eingabevolumens angewendet. Die Ausgabe eines Filters kann berechnet werden, indem das Skalarprodukt des Filters und Eingaberegion erzeugt wird (Gibson & Patterson 2017: S. 132).

Wird ein Filter auf das Eingabevolumen angewendet, erhält man eine Ausgabe, welche als Aktivierungs-Map eines Filters bezeichnet wird. Der Filterzahl (engl. filter count) ist ein Hyperparameterwert für jede Faltungsschicht. Dieser Hyperparameter kontrolliert außerdem, wie viele Aktivierungs-Maps aus der Faltungsschicht erzeugt werden, um als Eingabe für die nächste Schicht zu dienen. Dabei wird dies als die dritte Dimension (für die Anzahl der Aktivierungs-maps) betrachtet, für die Ausgabe des Aktivierungsvolumen in dreidimensionaler Form. Dabei kann der Hyperparameter für die Filterzahl frei gewählt werden, wobei einige Werte bessere und andere schlechtere Ergebnisse liefern (Gibson & Patterson 2017: S. 133).

Bei den Aktivierungs-Maps spielen die Aktivierungen eine große Rolle. Dabei handelt es sich um ein numerisches Ergebnis, wenn Neuronen entschieden haben, Informationen durchzulassen. Genauer gesagt handelt es sich um eine Funktion des Inputs der Aktivierungsfunktion, die Gewichtungen der Verbindungen (für den Input sowie dem Typ der Aktivierungsfunktion). Der Filter wird über die räumlichen Dimensionen Breite $\times$ Höhe des Eingabevolumens angewendet, wobei Informationen durch das CNN vorwärts geleitet werden. Dabei wird eine zweidimensionale Ausgabe, welche als Aktivierungs-Map des Filters bezeichnet wird. Auf der Abbildung 19 wird dargestellt, wie diese Aktivierungs-Map sich auf das Konzept eines gefalteten Feautres bezieht (Gibson & Patterson 2017: S. 133-134).

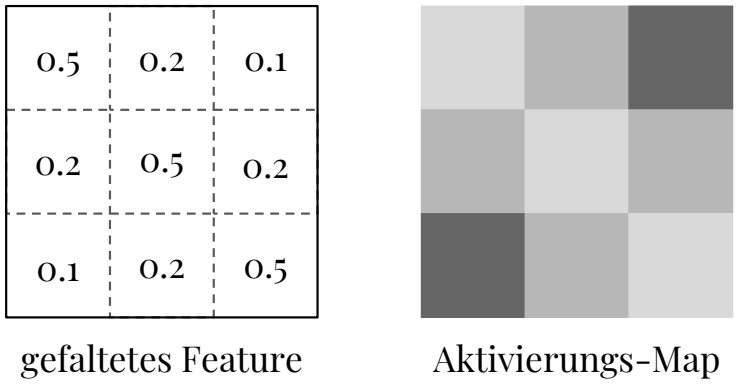


Abbildung 19: Auf der linken Seite ist ein gefaltetes Feature zu sehen und auf der rechten Seite die dazugehörige Aktivierungs-Map. Dabei handelt es sich um eine Illustration, wie Aktivierungs-Maps dargestellt werden können (in Anlehnung an Gibson & Patterson 2017: S. 134).

Um die Aktivierungs-Map zu berechnen, wird der Filter über das Eingabevolumen in der Tiefe geschoben. Dabei wird das Skalarprodukt zwischen den Einträgen im Filter und dem Eingabevolumen berechnet. Hier repräsentiert der Filter die Gewichtungen, welche mit dem bewegten Fenster bzw. der Teilmenge von den Eingabeaktivierungen multipliziert wird. Netzwerke lernen die Filter, welche aktiviert werden, wenn sie eine bestimmte Art von Feature in den Eingabedaten und einer räumlichen Position finden. Dann werden die dreidimensionalen Ausgabevolumen für die Faltungsschicht erstellt, indem die Aktivierungs-maps entlang der Tiefen-Dimension gestapelt werden. Dies ist in der Abbildung 20 zu sehen (Gibson & Patterson 2017: S. 134-135).

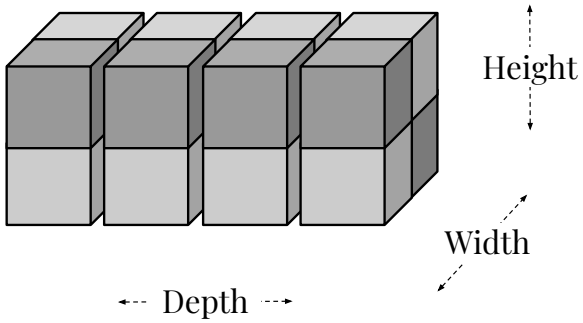


Abbildung 20: Auf dieser Grafik wird das Aktivierungsvolumen als Ausgabe der Faltungsschicht illustriert. Dabei enthält das Ausgabevolumen Einträge, welche als Ausgabe eines Neurons betrachtet wird, die nur ein kleines Fenster des Eingabevolumens betrachten (in Anlehnung an Gibson & Patterson 2017: S. 135).

Es gibt verschiedene Hyperparameter der Faltungsschicht. Dazu können folgende gehören (Gibson & Patterson 2017: S. 138):

- Größe des Filters/Kernels
- Tiefe der Ausgabe
- Stride
- Zero-Padding

Jeder Filter ist räumlich klein mit Bezug auf die Breite und Höhe der Filtergröße. Ein Beispiel könnte in der ersten Faltungsschicht liegen, welcher eine Filtergröße von $5 \times 5 \times 3$ besitzt. Dies würde bedeuten, dass der Filter 5 Pixel breit und auch 5 Pixel hoch ist sowie 3 Farbkanäle genutzt werden, wenn ein RGB-Farbkanal Anwendung findet (Gibson & Patterson 2017: S. 138).

Die Tiefe der Ausgabe des Volumens kann manuell gewählt werden. Hyperparameter der Tiefe kontrollieren die Anzahl der Neuronen in einer Faltungsschicht die mit derselben Region des Eingabevolumens verbunden sind

(Gibson & Patterson 2017: S. 139).

Mit der Konfiguration von Strides wird festgelegt, wie weit sich das Filterfenster pro Anwendung bewegen wird. Für jedes mal wenn die Filterfunktion auf die Eingabespalte angewendet wird, wird eine neue Tiefen-Spalte für das Ausgabevolumen erstellt. Wenn ein niedriger Wert für Strides gewählt wird, wie zum Beispiel 1, bei dem nur ein einzelner Schritt getätigt wird, dann werden mehr Tiefen-Spalten im Ausgabevolumen zugewiesen. Außerdem führt es auch zu stärker überlappende empfängliche Felder (engl. receptive fields) zwischen den Spalten, was wiederum zu einem größeren Volumen in der Ausgabe führt. Das Gegenteil tritt auf, wenn ein hoher Wert für Strides gewählt wird. Dann tritt eine geringere Überlappung auf und führt auch zu kleineren räumlichen Ausgabevolumen (Gibson & Patterson 2017: S. 139).

Ein weiterer Hyperparameter heißt Zero-Padding. Hiermit wird es möglich, die räumliche Größe des Ausgabevolumens steuern zu können. Die wird eingesetzt, wenn die räumliche Größe des Eingabevolumens im Ausgabevolumen beibehalten werden soll (Gibson & Patterson 2017: S. 139).

3.6.4 ReLU Aktivierungsfunktion als Schicht

In CNNs werden häufig ReLU Aktivierungsfunktionen als Schichten eingesetzt. Diese Schicht wendet eine Elementweise Aktivierungsfunktion über die Schwellenwerte der Eingabedaten an. Wenn zum Beispiel $max(0, x)$ auf Null, wir dieselbe Dimension ausgegeben, wie in der Eingabeschicht. Wird diese Funktion über dem Eingabevolumen ausgeführt, so werden beispielsweise die Pixelwerte geändert, die räumlichen Dimensionen der Eingabedaten für die Ausgabe werden nicht verändert (Gibson & Patterson 2017: S. 138).

3.6.5 Pooling-Schichten

Pooling-Schichten sind normalerweise zwischen aufeinander folgenden Faltungsschichten angeordnet. Dies liegt daran, dass Pooling-Schichten genutzt werden, um die räumliche Größe (Breite × Höhe) der Datenrepräsentation schrittweise zu reduzieren. Somit tragen Pooling-Schichten zur Kontrolle bezüglich der Überanpassung bei (Gibson & Patterson 2017: S. 140).

Die Pooling-Schicht verwendet eine $max()$ Operation, um die Eingabeda-

ten räumlich zu verändern. Dieser Vorgang wird Max-Pooling genannt. Bei einer Filtergröße von 2×2 würde die $max()$ Operation die größte der Vier Zahlen in der Filterregion nehmen. Dabei hat diese Operation keine Auswirkungen auf die Tiefen-Dimension (Gibson & Patterson 2017: S. 140).

Weiterhin setzen Pooling-Schichten Filter ein, um einen Downsampling-Vorgang für das Eingabevolumen durchzuführen. Diese Downsampling-Operationen werden entlang der räumlichen Dimension von Eingabedaten ausgeführt. Dies bedeutet, dass, wenn ein Eingabebild beispielsweise 32 Pixel breit und 32 Pixel hoch ist, das Ausgabebild kleiner wird (z.B. 16 Pixel breit und 16 Pixel hoch). Eine weitverbreitete Einstellung für Pooling-Schichten besteht darin, einen 2×2 Filter mit Strides von 2 anzuwenden. Hierdurch wird jede Tiefen-Spalte im Eingabevolumen um einen Faktor von Zwei auf die räumlichen Dimensionen (Breite und Höhe) heruntergerechnet wird. Dabei kann der Downsampling-Vorgang dazu führen, dass 75 Prozent der Aktivierungen gelöscht werden (Gibson & Patterson 2017: S. 140).

Pooling-Schichten besitzen keine Parameter für die Schicht, sondern zusätzliche Hyperparameter. Diese Schicht verfügt über keine Parameter, da eine feste Funktion des Eingabevolumens berechnet wird. Außerdem ist es nicht üblich, dass hier Zero-Padding eingesetzt wird (Gibson & Patterson 2017: S. 140).

3.6.6 Vollständig verbundene Schichten

Die vollständig verbundenen Schichten werden eingesetzt, um die Ergebnisse für die Klassen zu berechnen, welche die Ausgabe des Netzwerkes darstellen. Die Dimensionen des Ausgabevolumens lautet $[1 \times 1 \times N]$, wobei N die Anzahl der Klassen darstellt. Im Falle des Twitter-Korporas wäre N gleich 3, da es 3 Klassen in diesem Korpus gibt (negativ, neutral, positiv). Dabei sind alle Neuronen dieser Schicht mit allen Neuronen der letzten Schicht verbunden. Vollständig verbundene Schichten haben normale Parameter der Schicht sowie Hyperparameter. Weiterhin führt diese Schicht eine Transformation auf dem Eingabevolumen durch, welche von den Aktivierungen im Eingabevolumen und den Parametern abhängen (Gibson & Patterson 2017: S. 140-141).

3.6.7 CNN mit Worteinbettungen für Text

CNNs können auch für die Arbeit im Bereich NLP eingesetzt werden. Zwei Arbeiten zu diesem Thema stammen von (Kim, Yoon 2014) und (Zhang & Wallace 2016), wobei hier die Arbeit mit CNNs für die Klassifizierung von Sätzen im Fokus stand. Die hier eingesetzten Beispiele gehen außerdem auf die beiden Arbeiten und besonders (Zhang & Wallace 2016) zurück.

Ein Schwerpunkt in dem Einsatz von CNNs mit Worteinbettungen liegt darin, die Dimensionen der Matrizen zu verfolgen und verstehen. Um die Funktionsweise von Textklassifikation mit CNNs und Worteinbettungen veranschaulichen zu können, wird ein kurzer englischer Beispielsatz gewählt: *I like this movie very much!* mit einer Dimension von 5, für die Worteinbettungen (Kim, Joshua 2017; Zhang & Wallace 2016).

Dabei ist auf Abbildung 21 die CNN Architektur für die Satzklassifizierung mit dem vorher genannten Beispielsatz zu sehen. Diese ist zusätzlich mit Hashtags versehen, um die Erklärung zu strukturieren. Es werden drei Filterbereichsgrößen (engl.filter region sizes) mit 2, 3 und 4 gewählt, von denen jede 2 Filter hat. Filter wenden dann Faltungen auf die Satzmatrix aus und generieren Aktivierungs-Maps, mit einer variablen Länge. Es wird auf jede Aktivierungs-Map 1-Max-Pooling angewendet, sodass die größte Zahl jeder Aktivierungs-Map aufgezeichnet wird. So werden aus allen sechs Aktivierungs-Maps univariate Feature-Vektoren (engl. feature vectors) erzeugt. Die sechs Features sind verkettet und bilden einen Feature-Vektoren für die vorletzte Schicht. Der letzte Softmax-Funktion erhält später diesen Feature-Vektoren als Eingabe und nutzt diesen, um den Satz zu klassifizieren. Hier wird eine binäre Klassifizierung dargestellt, somit sind zwei mögliche Ausgangszustände möglich (Kim, Joshua 2017; Zhang & Wallace 2016).

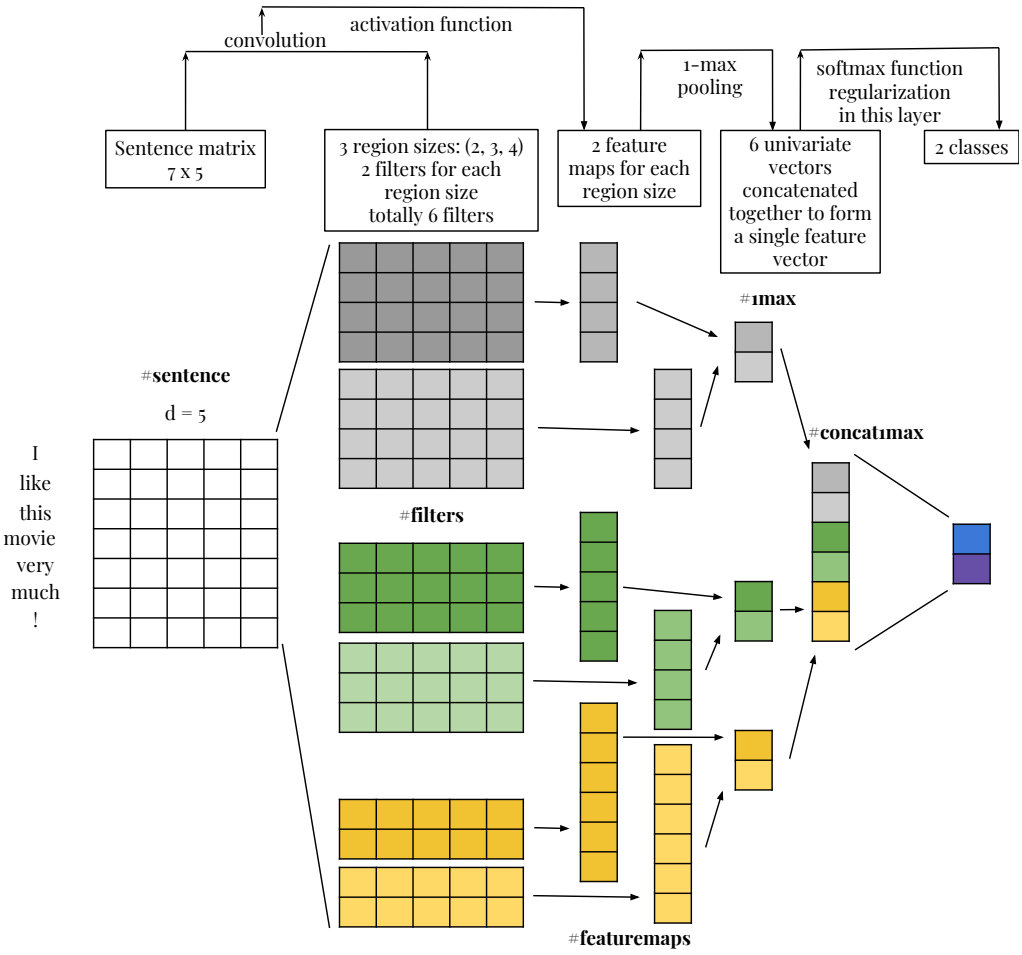


Abbildung 21: Hier ist die CNN Architektur für Satzklassifizierung dargestellt. Die Eingabe stellt der englische Satz *I like this movie very much!* dar und die Worteinbettungen haben Fünf Dimensionen. Eine genauere Erläuterung zu der Funktionsweise folgt in den nächsten Absätzen anhand der Hashtags (in Anlehnung an Zhang & Wallace 2016: S. 4).

Im folgenden werden alle Hashtags und die damit verbundenen Schritte erläutert. Zunächst der #sentence. Dabei ist zum einen der Beispielsatz: *I like this movie very much!*, relevant. Es sind somit sechs Worte und ein Ausrufezeichen gegeben, welches hier als Wort behandelt wird. Somit werden Sieben Worte in dem Beispielsatz gezählt. Außerdem wurde eine Dimension von 5, für die Wort-Vektoren gewählt. Dabei soll s die Länge der Sätze und d die

Dimension der Wort-Vektoren repräsentieren. Somit entsteht eine Matrix für den Satz in der Form $s \times d$ oder 7×5 (Kim, Joshua 2017).

Es folgen die #filters. Im Bereich der Computer Vision liegt eine Besonderheit von CNNs darin, das deren räumliche Orientierung 2D ist. Ähnlich wie Bilder haben auch Texte eine solche Orientierung. Allerdings besitzen Texte keine zweidimensionale, sondern eine eindimensionale Struktur, wobei die Reihenfolge der Wörter eine Bedeutung hat. Wie zuvor beschrieben, wurde ein 5-Dimensionaler Wort-Vektor ausgewählt, wodurch die Wörter in einen solchen 5-Dimensionalen Vektor ersetzt werden. Somit wird es notwendig, eine Dimension für den Filter festzulegen, welcher an die Größe des Wort-Vektors angepasst werden muss und die Regionsgröße (engl. region size), h, variiert. Weiterhin bezieht sich die Regionsgröße auf die Anzahl der Zeilen der Satzmatrix, die gefiltert werden würden. Dabei ist anzumerken, dass #filters der Abbildung 21 nur eine Illustration von Filtern sind (Kim, Joshua 2017).

Um die #featuremap zu erläutern, dient die Abbildung 22. Dabei wurden einige zahlen in die Satzmatrix und die Filtermatrix eingetragen.

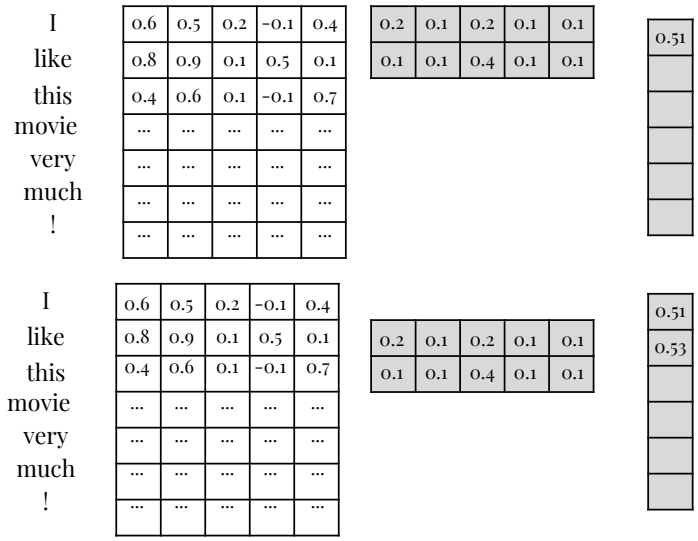


Abbildung 22: Hier ist die Satz- und Filtermatrix dargestellt. Diese Abbildung wird in dem folgenden Absatz näher erläutert (in Anlehnung an Kim, Joshua 2017).

Die Abbildung 22 soll die Wirkung des 2-Wort-Filters auf die Satzmatrix aufzeigen. Zunächst wird der 2-Wort-Filter, welcher durch die 2×5 Matrix, w repräsentiert wird, über die Wortvektoren von I und *like* geschoben. Danach wird ein Elementweises Produkt für alle 2×5 Elemente gebildet und summiert $(0,6 \times 0,2 + 0,5 \times 0,1 + \ldots + 0,1 \times 0,1 = 0,51)$. Somit wird der Wert $0,51$ als erstes Element für die Ausgabesequenz, o, für den Filter aufgezeichnet. Anschließend bewegt sich der Filter um ein Wort nach unten und überdeckt nun die Wort-Vektoren von *like* und *this* und führt dieselbe Operation aus, um den Wert $0,53$ zu erhalten. Hierdurch wird o die Form von $(s-h+1 \times 1)$ bzw. $7-2+1 \times 1$ besitzen. Um die Aktivierungs-Map von c zu erhalten, wird ein Bias, wie zum Beispiel ein Skalar in der Form 1×1 gewählt. Anschließend wird eine Aktivierungsfunktion wie eine ReLU angewandt, wodurch c resultiert, mit derselben Form wie o, $s-h+1 \times 1$ (Kim, Joshua 2017).

Bei #1max ist zu beachten, dass die Dimensionalität von c sowohl als s als auch von h abhängig ist. Somit variiert sie, durch Sätze unterschiedlicher Länge und Filter mit unterschiedlichen Regionsgrößen. Um dieses Problem zu lösen, wird eine 1-Max-Pooling Funktion genutzt, um die größten Werte

des Vektors c zu extrahieren (Kim, Joshua 2017).

Der letzte Abschnitt behandelt #concat1max. Nach dem 1-Max-Pooling liegt ein Vektor mit einer fixen Länge von 6 Elementen (= Anzahl der Filter = Anzahl der Filter pro Regionsgröße (2) $\times$ Anzahl der berücksichtigten Regionsgröße (3)). Der Vektor, mit fester Länge kann dann in eine Softmax-Schicht welche vollständig verbunden ist, eingegeben werden, um die Klassifizierung durchzuführen. Der Fehler aus einer Klassifizierung kann mittels Backpropagation durch die folgenden Paramenter das Lernen unterstützen (Kim, Joshua 2017).

- die w Matrix, welche o erzeugt
- Bias-Term der zu o hinzugefügt wird, um c zu erhalten
- Wort-Vektoren (optional)

3.7 Recurrent Neural Networks und Long Short Term Memory

Recurrent Neural Networks (RNNs) gehören zu den neuronalen Netzwerken, welche für die Verarbeitung von sequentiellen Daten verwendet werden. Im Gegensatz zu CNNs, welche auf die Verarbeitung von Gittern mit den Werten X (z.B. ein Bild) spezialisiert sind, verarbeiten RNNs eine Folge von Werten $x(1), \ldots, x(\tau)$. Somit können RNNs auf viel längere Sequenzen skaliert werden, als es für ein CNN möglich wäre. Dabei können die meisten RNNs Sequenzen mit einer variablen Länge verarbeiten (Goodfellow et al. 2016: S. 373).

Long Short-Term Memory (LSTM) gilt als eines der am häufigsten eingesetzten Varianten von RNNs. Diese Netzwerke wurden bereits 1997 eingeführt (Hochreiter & Schmidhuber 1997). Eine der wichtigsten Komponenten von LSTMs stellen die Speicherzelle (engl. memory cell) und die Tore (engl. gates) dar. Dazu zählen auch das Vergiss-Tor (engl. forget gate) und Eingabe-Tor (engl. input gate). Die Inhalte von Speicherzellen werden durch diese beidenTore moduliert. Wenn diese Tore geschlossen sind, wird der Inhalt der Speicherzelle unverändert bleiben zwischen einem Zeitschritt (engl. time

step) und dem nächsten. Die Struktur mit den Toren schafft es, dass Informationen über viele Zeitschritte hinweg, aufbewahrt werden können (Gibson & Patterson 2017: S. 150).

3.7.1 LSTM Architektur

Um die Verbindungen und den Aufbau von LSTMs, seinen Elementen und Schichten kann man mit einem Multilayer Feedforward Networks beginnen. Wie auf Abbildung 23.

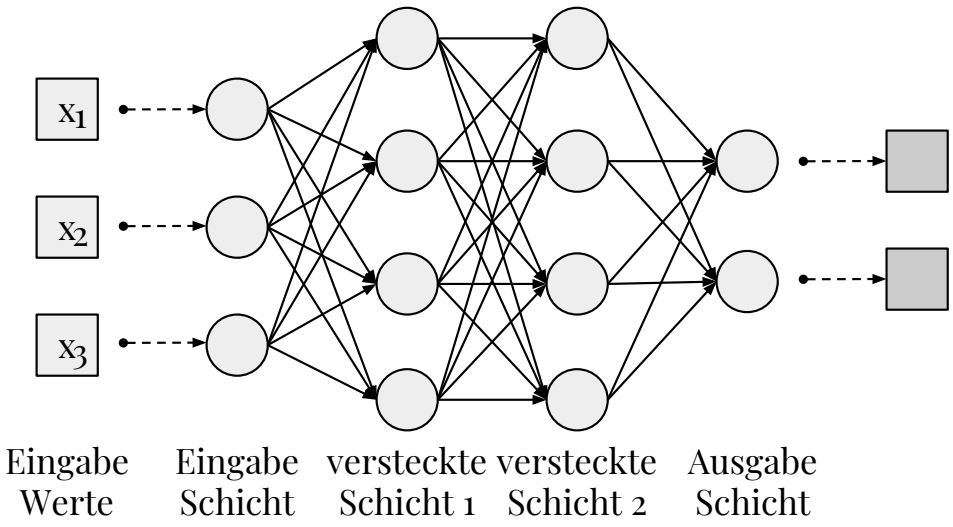


Abbildung 23: Hierbei handelt es sich um eine Architektur eines Multilayer Feedforward Networks. Dieses verfügt über Eingabewerte, eine Eingabeschicht, zwei versteckten Schichten und einer Ausgabeschicht. Anhand dieser Visualisierung soll die Erläuterung zu LSTM aufgebaut werden (in Anlehnung an Gibson & Patterson 2017: S. 151).

Wenn nun eine Schicht aus dem Netzwerk einzeln dargestellt wird und das als Neuronen, in einer abgeflachten Repräsentation, so würde dies wie auf Abbildung 24 aussehen (Gibson & Patterson 2017: S. 151).

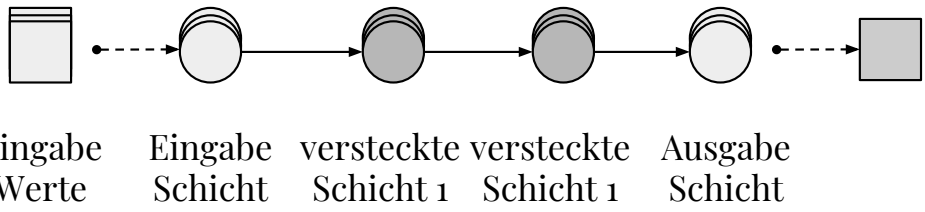


Abbildung 24: Auf dieser Illustration ist eine Schicht aus dem Multilayer Feedforward Network einzeln dargestellt. Dabei beinhalten dieser Ausschnitt ebenfalls die Eingabewerte, Eingabeschicht, zwei versteckte Schichten und eine Ausgabeschicht (in Anlehnung an Gibson & Patterson 2017: S. 152).

Mit RNNs wird ein Verbindungstyp eingesetzt, welcher die Ausgabe eines Neurons in einer versteckten Schicht, mit der Eingabe desselben Neurons verbindet. Daher wird es möglich, dass Eingaben aus einem vorherigen Zeitschritt, als eingehende Information eingehen. Dabei zeigt die Abbildung 25, durch die Veränderung der Abbildung 24, wie die wiederkehrenden (engl. recurrent) Verbindungen eines LSTMs visualisiert werden können (Gibson & Patterson 2017: S. 152).

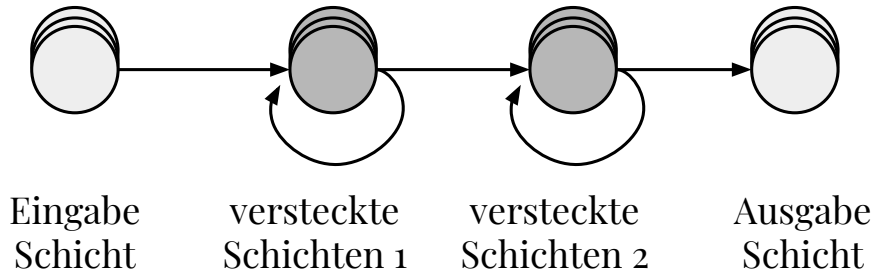


Abbildung 25: Auf dieser Abbildung ist die grafische Darstellung von wiederkehrenden Verbindungen in den versteckten Schichten dargestellt (in Anlehnung an Gibson & Patterson 2017: S. 152).

In der Abbildung 26 wird weiterhin visualisiert, wie das Entrollen des Netzwerks aus der Abbildung 25 zu verstehen ist. Auch wie die Informationen durch das Netzwerk, in vorwärts fließen und durch die Zeit.

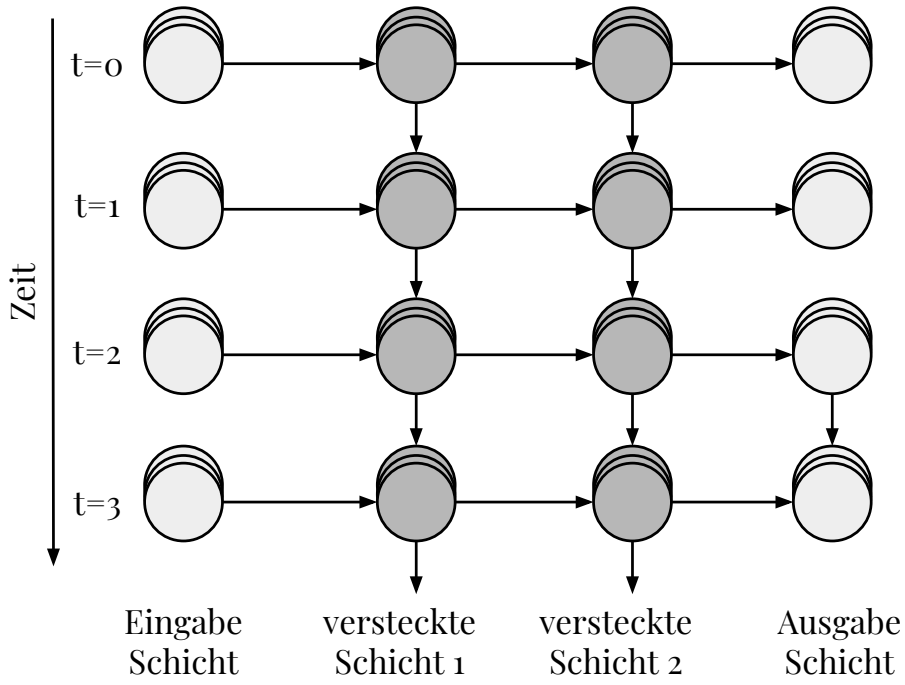


Abbildung 26: Auf dieser Abbildung ist ein RNN dargestellt, welches entlang der Zeitachse ausgerollt ist. Dabei handelt es sich um die verschiedenen Zeitschritte (in Anlehnung an Gibson & Patterson 2017: S. 153).

3.7.2 LSTM Einheiten und Speicherzelle

Eine LSTM Einheit (engl. LSTM unit) ist eine Variation eines klassischen Neurons. Dabei besitzt jede LSTM Einheit zwei Arten von Verbindungen. Eine Verbindung zum vorherigen Zeitschritt, also den Output dessen Einheit. Sowie eine andere, zu der vorherigen Schicht. Mittels der Speicherzelle wird es bei LSTM möglich, einen Zustand (engl. state) über die Zeit hinaus aufrechtzuerhalten. Somit stellt es ein elementares Konzept dar. Ein anderer Begriff für den Körper einer LSTM Einheit lautet LSTM Block (engl. LSTM block). Dieser ist auf der Abbildung 27 zu sehen. Dabei beinhaltet die LSTM Einheit verschiedene Komponenten (Gibson & Patterson 2017: S. 153-154):

- Drei Tore
  - Eingabe-Tor

  - Vergessen-Tor
  - Ausgabe-Tor
- Block Eingabe
- Speicherzelle
- Ausgabe der Aktivierungsfunktion
- Peephole-Verbindungen

Dabei sind drei verschiedene Tor-Einheiten relevant. Das Eingabe-Tor soll die Einheit vor unnötigen Eingabeereignissen schützen. Mit dem Vergessen-Tor sollen Einheiten einen vorherigen Inhalt des Speichers vergessen. Durch das Ausgabe-Tor sollen Inhalte der Speicherzelle zum Output der Einheit durchlassen oder auch nicht (Gibson & Patterson 2017: S. 154).

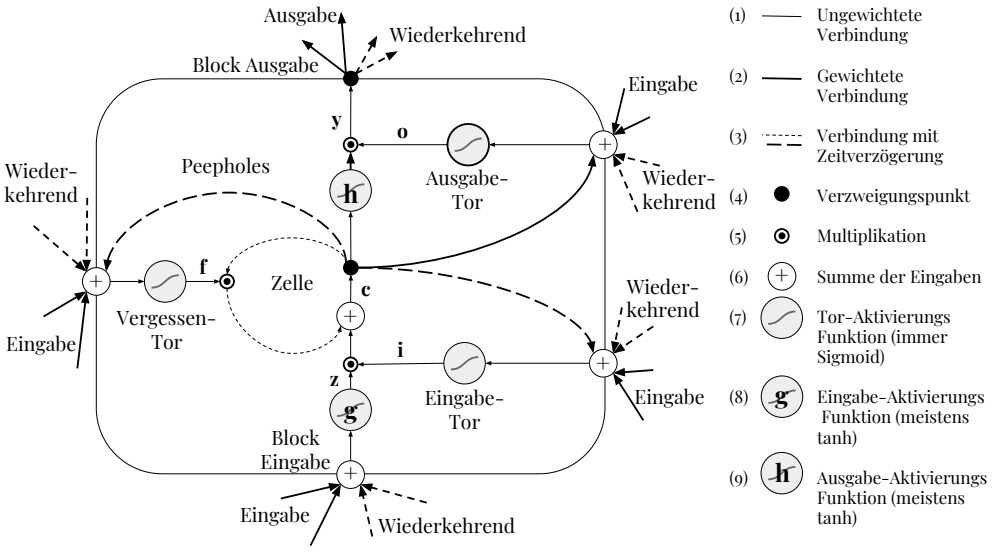


Abbildung 27: Diese Abbildung stellt einen LSTM Block dar, mit allen Verbindungen und Toren. Die Ausgabe eines LSTM Blocks ist wiederkehrend mit der Eingabe sowie allen anderen Toren verbunden. Dabei verfügen die Eingabe-, Vergessen- und Ausgabe-Tore in einer LSTM Einheit über eine Sigmoid Aktivierungsfunktion, mit Einschränkung auf $[0, 1]$. Bei der Ein- und Ausgabe einer LSTM Einheit nutzen normalerweise eine tanh Aktivierungsfunktion (in Anlehnung an Gibson & Patterson 2017: S. 154).

Mit einer Notation von (Greff et al.2015) lassen sich für die Vektoren bei einem Vorwärtspass (engl. forward pass) in einer *LSTM* Schicht wie folgt darstellen. Mit Formel 19 wird die Blockeingabe berechnet. Mittels 20 wird das Eingabe-Tor definiert. Das Vergessen-Tor enthält die Formel unter 21. Die Formeln für den Zellen-Zustand (engl. cell state) sind unter 22, des Ausgabe-Tors unter 23 und der Block-Ausgabe 24 (Gibson & Patterson 2017: S. 155).

$$z^t = g(W_z x^t + R_z y^{t-1} + b_z) \tag{19}$$

$$i^t = \sigma(W_i x^t + R_i y^{t-1} + p_i \odot c^{t-1} + b_i) \tag{20}$$

$$f^t = \sigma(W_f x^t + R_f y^{t-1} + p_f \odot c^{t-1} + b_f) \tag{21}$$

$$c^t = i^t \odot z^t + f^t \odot c^{t-1} \tag{22}$$

$$o^t = \sigma(W_o x^t + R_o y^{t-1} + p_o \odot c^t + b_o) \tag{23}$$

$$y^t = o^t \odot h(c^t) \tag{24}$$

3.8 Vorhersagen erklären mittels LIME - Local Interpretable Model-Agnostic Explanations

Trotz der Vielseitigen Erfolge im Bereich Machine Learning und Deep Learning stellen diese Modelle meistens Black-Boxen dar. Es ist eindeutig, was in ein solches Modell als Eingabe eingeht und auch das Ergebnis ist einfach zu erkennen und verstehen. Aber wie genau dieses Ergebnis entstanden ist und welche Parameter dafür gesorgt haben oder im Kontext von Textklassifikation, welche Eingabe-Worte haben dieses Ergebnis beeinflusst. Diese Fragestellungen bleiben häufig unbeantwortet. Zu diesem Thema haben (Ribeiro et al. 2016a) eine Arbeit mit dem Titel *Why Should I Trust You? Explaining the Predictinos of Any Classifier* einen Ansatz präsentiert.

In dieser Arbeit wird ein Ansatz für die Lösung dieses Problems vorgestellt, welcher Local Interpretable Model-Agnostic Explanations (LIME) genannt. Dabei handelt es sich um eine neuartige Technik, Vorhersagen von jedem

Klassifizierer erklären zu können und das auf eine interpretierbare und glaubhafte Art und Weise. Dies wird durch das lernen eines interpretierbaren Modells lokal an der Vorhersage umgesetzt (Ribeiro et al. 2016a: S. 1). In ihrer wird ein anschauliches Beispiel dafür gebracht, wann und warum diese Erklärungen zu Vorhersagen wichtig sein können. Hierzu dient die folgende Abbildung 28.

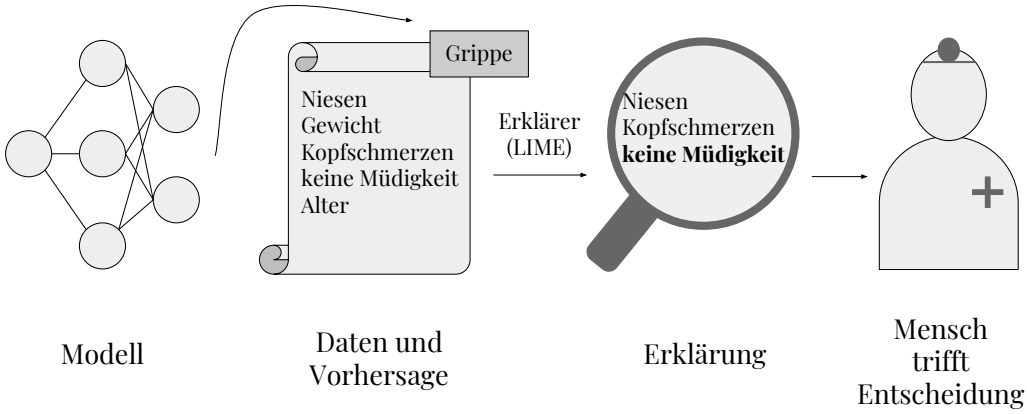


Abbildung 28: Hier wird illustriert wie eine individuelle Vorhersagen erklärt werden kann. Dabei geht um das Beispiel, dass ein Arzt durch ein Modell eine Vorhersage basierend auf den vorliegenden Daten bezüglich des Krankheitszustandes eines Patienten erhält. Dabei wird zunächst ein Modell eingesetzt welches anhand der Symptome eine Krankheit klassifiziert, eine Grippe in diesem Fall. Mittels des LIME-Erklärers werden die ausschlaggebenden Symptome dargestellt und als Erklärung aufbereitet. Der Arzt und Mensch hat nun die Möglichkeit, die Vorhersage des Modells mit seinem Kenntnisstand und Erfahrung zu überprüfen. Er kann diese Vorhersage bestätigen und der Vorhersage des Modells vertrauen oder diese widerlegen. (in Anlehnung an Ribeiro et al. 2016a: S. 2).

Dieser Ansatz wird prototypisch eingesetzt. Dabei liegt das Ziel darin, die Implementierung zu realisieren und eine der hier verwendeten Architekturen (LSTM oder CNN) mit diesem Ansatz umzusetzen. Hiermit wäre es möglich die Vorhersagen zu untersuchen, verstehen und erklären. Somit lässt sich die Sinnhaftigkeit von Klassifizierungen einstufen und es wird möglich, zu sehen, welche der Worte die Klassifizierungen beeinflusst haben und wie stark.

4 Methodik und Konzeption

Im folgenden sollen Methoden beschrieben werden, welche ausgewählt wurden und Anwendung fanden. Anschließend wird ein Konzept vorgestellt, dass die durchzuführenden Experimente klar strukturieren und definieren soll.

4.1 Angewandte Methoden

Eine der grundlegenden Methoden stellt die Entwicklung der Deep Learning Modelle dar. Für diese Aufgabe, wurden zwei konkrete Ansätze ausgewählt und implementiert. Dabei handelt es sich um ein LSTM und CNN. Weiterhin musste über die Art und Weise der Implementierung entschieden werden. Als Programmiersprache wurde Python gewählt. Außerdem wurden Keras mit TensorFlow als Backend ausgewählt. Diese Auswahl wird durch die breite Verbreitung und Dokumentation dieser Frameworks begründet.

Um die Ergebnisse einordnen zu können, wird eine Performanceanalyse durchgeführt. Zunächst kann die Performance der beiden Architekturen verglichen werden. Dabei ist anzumerken, dass jedes Modell je mit Word2Vec- und fastText Worteinbettungen implementiert wurden. So, dass verschiedene Vergleichsmöglichkeiten zur Verfügung stehen. Weiterhin kann durch unterschiedliche Experimente festgestellt werden, inwiefern die Leistung der Modelle steigerungsfähig ist. Als Maßstab für die Ergebnisse dienen einige publizierte und bereits vorgestellte Resultate aus anderen wissenschaftlichen Arbeiten.

Während der Entwicklung und Implementierung war außerdem die Einhaltung von Best Practices ein wichtiger Bestandteil. Der Quellcode sollte gut kommentiert und für dritte verständlich sein. Dies spielt auch eine Rolle für die Reproduzierbarkeit.

Zwei methodische Vorgehen lassen sich als Learning by Doing und Forschung durch Entwicklung beschreiben. Hiermit ist gemeint, dass einige Kenntnisse bezogen auf Deep Learning und im Umgang mit den Frameworks erst während der praktischen Arbeit gewonnen und später theoretisch vertieft wurden.

Weitere Methoden lagen in der Beschaffung von Daten mittels Python, der Auswahl von Architekturen und Einrichtung der Implementierungsumge-

bung. Auch die Recherche von Literaturen und Quellen war sehr wichtig. Hierfür wurden verschiedene Fachbücher, Internetseiten, Dokumentationen und Tutorials genutzt, um die Theorie zu durchdringen sowie die praktische Umsetzung zu erreichen.

4.2 Konzeption

Deep Learning wird bereits in der Sentimentanalyse eingesetzt. Allerdings gibt es in der deutschen Sprache zum aktuellen Zeitpunkt keine große Auswahl an Datensätzen für das Entwickeln von eigenen Deep Learning Modellen in diesem Bereich. Die Arbeit von (Cieliebak et al. 2017) zielt darauf ab, dieses Problem zu lösen, einen weiteren Datensatz zur Verfügung zu stellen und einen Maßstab für Sentimentanalyse von deutschen Tweets anzubieten.

Diese Ergebnisse stellen den Maßstab dar, allerdings ist ein direkter Vergleich schwer. In den anderen Arbeiten wurden komplexere Systeme eingesetzt. Hier ausschließlich die Implementierung von CNN und LSTM.

Eine wichtige Komponente der Modelle stellen die Worteinbettungen dar. Dabei wurden vortrainierte Word2Vec- und fastText-Worteinbettungen gefunden. Die Beiden vortrainierten Worteinbettungen wurden auf unterschiedliche Korpora trainiert und somit ist auch hier interessant, inwiefern dies die Leistung der unterschiedlichen Varianten unterscheidet.

Die Experimente werden durch das Testen von verschiedenen Einstellungen für die Parameter in Tabelle 6 geprägt. Die Modelle werden zunächst mit identischen Einstellungen für diese Parameter und der generellen Architektur ausgeführt, um einen Startwert für die Genauigkeit, den Loss und dem F1 Wert zu erhalten. Anschließend werden verschiedene Einstellungen der Parameter vorgenommen. Dabei sollen die Ergebnisse und Kombinationen der Parameter sowie deren Einstellung später tabellarisch zusammengefasst werden.

Art des Parameters	**Bezeichnung**
Erforderliche Parameter	Anzahl Neuronen in LSTM-/CNN-Schichten Anzahl LSTM-/CNN-Schichten Anzahl Neuronen versteckte Schichten Anzahl versteckte Schichten
Optionale Parameter	Lernrate Lernraten-Zeitplan Momentum Dropout Optimierungsalgorithmen Wortschatzgröße Tweet Länge L1 und L2 Regularisierung

Tabelle 6: Zu untersuchende Parameter für die Experimente. Diese wurden in erforderliche und optionale Parameter kategorisiert. Zum Beispiel ist es nötig, die Anzahl der Neuronen in Schichten anzugeben, aber nicht die Lernrate. Für diese Parameter werden verschiedene Werte und Alternativen ausgewählt und anhand des Trainings- und Validierungsdatensatzes untersucht. Aus den verschiedenen Experimenten werden dann unterschiedliche Konfigurationen ausgewählt, um diese anhand der Testdaten zu testen.

Neben diesen Experimenten wird prototypisch noch ein weiteres Vorgehen umgesetzt, welches für die Konfiguration von Deep Learning Modelle eingesetzt werden kann. Dabei handelt es sich um eine Optimierungstechnik von Hyperparametern. Mittels diesem Ansatz ist es beispielsweise möglich verschiedene Optimierungsalgorithmen anzugeben und das Netzwerk mit allen Algorithmen zu trainieren. Dies kann nacheinander oder Parallel geschehen. Anschließend können die Ergebnisse ausgegeben und verglichen werden.

Ein andere prototypische Implementierung stellt der LIME-Ansatz dar. Hiermit soll ein Deep Learning Modell umgesetzt werden und auf dem Twitter-Korpora trainiert. Mit diesem Training wird auch ein lokaler Erklärer von LIME trainiert und es wird möglich, durch das Modell getätigte Vorhersagen, zu untersuchen. Somit wird es möglich sein, nachvollziehen zu können, welche Worte, welchen Einfluss auf die Vorhersagen gehabt haben und wie sinnvoll diese waren.

5 Implementierung und Umsetzung

In diesem Kapitel werden die Soft- und Hardware vorgestellt, welche für die Experimente und Tests eingesetzt wurde. Weiterhin wird die praktische Umsetzung dargestellt und erläutert. Dabei soll die Vorverarbeitung des Korpus aufgezeigt werden sowie eine Visualisierungen zum Korpus. Anschließend wird die Python-Implementierung erläutert.

5.1 Soft- und Hardware

5.1.1 GPU

Durch die Nutzung einer Grafikkarte (engl. graphics processing unit - GPU) ist es möglich das Training von Deep Learning Modellen effizienter zu gestalten. Im Unterschied zu CPUs besitzen GPUs eine größere Anzahl an Rechenkernen (engl. arithmetic logic units - ALUs), welche die Matrixoperationen parallel durchführen. Bei der zentralen Verarbeitungseinheit (engl. central processing unit - CPU) sieht es anders aus. Diese haben im Vergleich zu GPUs weniger Rechenkerne, welche die Berechnungen sequentiell durchführen. Hier wurde eine Grafikkarte von NVIDIA genutzt, eine GeForce GTX 1080, mit 8 GB Speicher.

5.1.2 TensorFlow

Bei TensorFlow handelt es sich um eine skalierbare und plattformübergreifende Programmierschnittstelle, welche Open Source ist. Eingesetzt wird TensorFlow für die Implementierung und Ausführung von Algorithmen aus dem Bereich Machine Learning. Außerdem werden auch Wrapper für Deep Learning bereitgestellt. Entwickelt wurde TensorFlow durch Forscher und Ingenieure des Google Brain-Teams. Zwar wird die Hauptentwicklung dieses Projektes durch Googles Forscher und Ingenieure voran getrieben, jedoch beinhaltet es auch viele Beiträge der Open Cource Community. Zunächst wurde TensorFlow für den internen Gebrauch bei Google entwickelt. Allerdings folgte im November 2015 dann die Veröffentlichung (Raschka & Mirjalili 2017: S. 600).

TensorFlow ermöglicht die Ausführung von Deep Learning Modellen auf dem CPU oder der GPU. Um die Leistung der Modelle zu erhöhen, werden diese Modelle auf einer oder mehreren GPUs ausgeführt. Hier unterstützt TensorFlow GPUs, welche wiederum CUDA unterstützen. Aktuell kann TensorFlow

mittels verschiedenen Programmiersprachen implementiert werden und stellt Schnittstellen bereit. Hierzu zählt Python, welche die vollständigste Schnittstelle darstellt. Außerdem wird auch eine Schnittstelle für C++ angeboten und für Sprachen wie Java, Haskell, Node.js und Go, wobei diese nicht so stabil sind (Raschka & Mirjalili 2017: S. 600-601).

5.1.3 Keras

Keras stellt ein Deep Learning Framework für Python dar. Dabei werden verschiedene Möglichkeiten angeboten, unterschiedliche Deep Learning Modelle zur trainieren und auszuführen. Keras wurde ursprünglich für Forscher entwickelt. Der Nutzerkreis von Keras erstreckt sich von akademischen Forschern über Startups bis hin zu großen Unternehmen oder auch Studenten. Hierzu zählen Unternehmen wie Google, Netflix, Uber und CERN (Chollet 2018: S. 61).

Keras lässt sich als Bibliothek verstehen, welche übergeordnete Bausteine für die Entwicklung von Deep Learning Modellen anbietet. Weiterhin kümmert sich Keras nicht um Low-Level-Operationen wie Tensor-Manipulation. Für diese Aufgaben wird eine optimierte Tensor Bibliothek genutzt, welche als die Backend Engine von Keras dient. Es wird durch eine modulare Architektur möglich, noch weitere Backend Engines nutzen zu können wie zum Beispiel Theano oder CNTK. Auf der folgenden Abbildung 29 ist die Architektur dargestellt (Chollet 2018: S. 61).

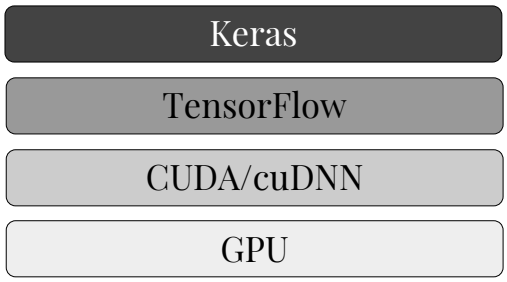


Abbildung 29: Auf dieser Abbildung ist der verwendete Soft- und Hardware Stack dargestellt. Dabei wird Keras eingesetzt, um ein Deep Learning Modell aufzubauen. TensorFlow dient als Backend Engine. Um den GPU nutzen zu können, wird darunter eine optimierte Bibliothek eingesetzt, welche die Operationen aufruft und durchführt. Dabei handelt es sich um die Nvidia CUDA Deep Neural Network library (cuDNN) (in Anlehnung an Chollet 2018: S. 62).

5.1.4 CUDA und cuDNN

Bei cuDNN handelt es sich um eine Bibliothek für neuronale Netzwerke, welche von Nvidia stammt. Sie ermöglicht die Rechenleistung von GPUs für Deep Learning zu nutzen. Weiterhin basiert diese Bibliothek auf der Compute Unified Device Architecture (CUDA). Diese wurde ebenfalls von Nvidia entwickelt und stellt eine Programmiertechnik darstellt, welche einzelne Programmteile durch die Grafikprozessoren bearbeiten lässt. Neben Keras und TensorFlow werden noch weitere Deep Learning Frameworks wie Caffee oder PyTorch unterstützt. Außerdem handelt es sich hierbei um eine freie Software, welche für akademische oder kommerzielle Zwecke einsetzbar ist (Litzel 2018).

5.2 Praktische Umsetzung

5.2.1 Vorverarbeitung der Tweets

Bevor die eigentliche Vorverarbeitung der Tweets starten konnte, wurden alle drei Korpora zu einen Korpus verschmolzen. Dies geschah durch den Einsatz von Pandas und dem Laden der Daten als Dataframe. Hier wurden bereits die ersten Tweets herausgefiltert. Es gab eine größere Anzahl von Tweets mit dem Inhalt *„Not Available“*. Diese wurden bereits bei dem zusammenfüh-

ren der Korpora entfernt. Die weitere Vorverarbeitung des gesamten Korpus wurde in Vier Schritte unterteilt. Zunächst wurden die URLs in den Tweets gefiltert und gelöscht. Folgend wird der dafür benötigte Python-Code dargestellt und danach die Erklärung.

```
with open("corpus.tsv", encoding="UTF-8") as f:
  s = f.read()

with open("corpus_1.tsv", "w", encoding="UTF-8") as g:
  s = re.sub('http[s]?://(?:[a-zA-Z]|[0-9]|[$-_@.&+]|[!*\(\),
    |(?:%[0-9a-fA-F][0-9a-fA-F]))+', '', s, flags=re.MULTILINE)
  g.write(s)
```

Zunächst wird der verschmolzene Korpus (*corpus.tsv*) geöffnet und gelesen. Anschließend wird eine neue Datei angelegt (*corpus_1.tsv*). Danach sollen alle URLs in dem gelesenen Korpus gefiltert und gelöscht werden. Es wurden alle URLs erfolgreich entfernt. Ein Problem ist bei der manuellen Durchsicht des Korpus aufgefallen. Zwar wurden alle URLs entfernt, aber in einigen, aber wenigen Tweets wurden Teile von URLs entdeckt. Dabei handelte es sich um die Buchstabenfolgen *http* und *~http*. Diese wurden mittels der Suchfunktion innerhalb der Programmierumgebung manuell entfernt.

In dem zweiten Schritt wurde folgendes Problem angegangen. Da vortrainierte Worteinbettungen eingesetzt wurden, können nur enthaltene Worte genutzt werden. Das bedeutet, dass Sonderzeichen oder Emoticons keine Berücksichtigung finden. Allerdings sind besonders im Internet Emoticons weit verbreitet, um ein bestimmtes Gefühl übermitteln zu können. Um diese Gefühle nicht zu verlieren, wurden verschiedene Versionen von Emoticons in Gruppen aufgeteilt, die anschließend in ein passendes Wort geändert wurden. In Tabelle 7 sind Beispiele für Emoticons und deren zugeordnetes Wort aufgeführt.

Emoticon	Wort
:D	lachen
:)	freude
:P	frech
;)	augenzwinkern
:O	erstaunt
:(	traurig
;(	weinen

Tabelle 7: Übersicht zu den Emoticons und Worten, durch denen diese ersetzt wurden. In dieser Übersicht sind die sieben verschiedenen Gruppen aufgezeigt. Dabei wurde je ein Beispiel für das Emoticon dargestellt und dem dazugehören Wort. Die manuelle Überprüfung hat gezeigt, dass die hier dargestellten Emoticons am häufigsten genutzt werden. Es gibt auch andere Möglichkeiten ein lachendes Emoticon zu erstellen, als das abgebildete.

Die Umsetzung erfolgte nach dem ähnlichen Prinzip wie im ersten Schritt und wird auch für die nächsten Schritte verfolgt. Hier folgt zunächst ein Codeausschnitt und danach folgt die dazu gehörige Erläuterung.

```
with open("corpus_1.tsv", encoding="UTF-8") as f:
  s = f.read()

with open("corpus_2.tsv", "w", encoding="UTF-8") as g:
  ...
  # 4. augenzwinkern (nach rechts lesbar)
  s = s.replace(";)", "augenzwinkern")
  s = s.replace(";]", "augenzwinkern")
  s = s.replace(";-)", "augenzwinkern")
  s = s.replace(";-]", "augenzwinkern")
  # 4. augenzwinkern (nach links lesbar)
  s = s.replace("(;", "augenzwinkern")
  s = s.replace("[;", " augenzwinkern")
  s = s.replace("(-;", "augenzwinkern")
  s = s.replace("[-;", "augenzwinkern")
  ...
  g.write(s)
```

Bei diesem Code handelt es sich nur um einen Ausschnitt. Es wird wie zuvor

der Korpus geöffnet und gelesen. Anschließend wird eine neue Datei erstellt. In dem eingelesenen Korpus werden die angegebenen Emoticons durch das Wort *augenzwinkern* ersetzt und in die neue Datei geschrieben. Dabei wurden gängige Versionen des Emoticons angegeben, welche von rechts oder auch links lesbar sein können. Die drei Punkte sollen darauf hinweisen, dass die anderen, in Tabelle 7 aufgezeigten Emoticons ebenfalls nach diesem Vorgehen gesucht und ersetzt wurden.

Während des folgenden Schrittes mussten Sonderzeichen entfernt werden und gleichzeitig auch die deutschen Umlaute wie *ä* durch eine andere Schreibweise wie *ae* ersetzt werden. Auch das *ß* wurde durch *ss* ausgetauscht. Die Begründung für dieses Vorgehen liegt in der Schreibweise in den Worteinbettungen. In diesen sind auch die Umlaute dementsprechend benutzt worden. Wenn diese nicht angepasst wären, würden Worte mit Umlaute nicht auffindbar sein. Hierfür wurde das identische Verfahren wie im Schritt zuvor angewandt. Dabei wurden die zu ersetzenden Sonderzeichen per Hand zusammengetragen. Allerdings war es nötig, manuell noch eine kleine Anzahl von Sonderzeichen zu entfernen.

Der letzte Schritt wurde in kleine Unterschritte aufgeteilt. Hier wurden zunächst die Sentimente durch Zahlen ersetzt. Tabelle 8 zeigt die Sentimente und numerische Werte auf.

Sentiment	**Numerischer Wert**
negativ	0
neutral	1
positiv	2

Tabelle 8: Diese Übersicht zeigt die drei vorhandenen Sentimente negativ, neutral und positiv sowie ihre die numerische Repräsentation. Dabei wurde das Sentiment negativ mit einer 0 ersetzt, neutral mit 1 und positiv mit einer 2.

Der nächste Teilschritt lag in der Aufteilung der Korpora. Bis zu diesem Zeitpunkt bestand der Korpus aus einer großen TSV-Datei, in dem Format *sentiment, tweet*. Bevor jedoch die Aufteilung stattfinden sollte, wurde der vorliegende Korpus mittels Scikit-Learn t-SNE untersucht. Anhand dieser Bibliothek ist es möglich den Korpus grafisch darzustellen. Zu diesem Zeitpunkt

waren noch mehrfach vorkommende Tweets enthalten. Die Gesamtzahl hat sich jedoch von 92.976 auf 91.538 Tweets reduziert. Dies liegt daran, dass einige Tweets nur aus einer URL bestanden haben und somit komplett aus dem Korpus entfernt wurden. Auf der Abbildung 30 ist der gesamte Korpus mittels t-SNE grafisch dargestellt.

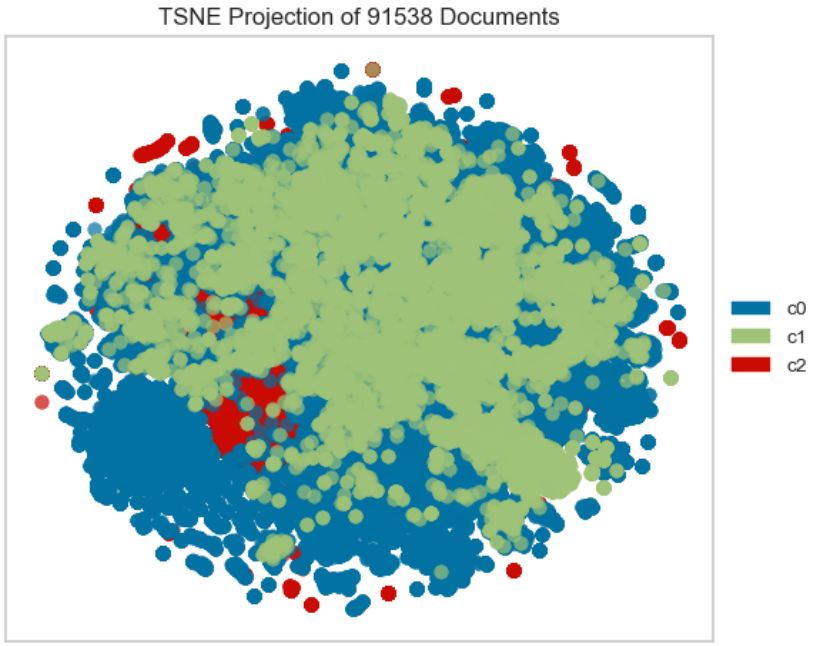


Abbildung 30: Auf dieser Abbildung sind alle Worte des gesamten Korpus als Punkt dargestellt. Dabei stellt das Kluster 0 (c0, blaugrün) die neutralen Tweets dar, Kluster 1 (c1, grün) die positiven und Kluster 2 (c2, rot) die negativen.

Es ist auch möglich die Klassenbalance darzustellen. Dies wird auf der folgenden Abbildung 31 illustriert.

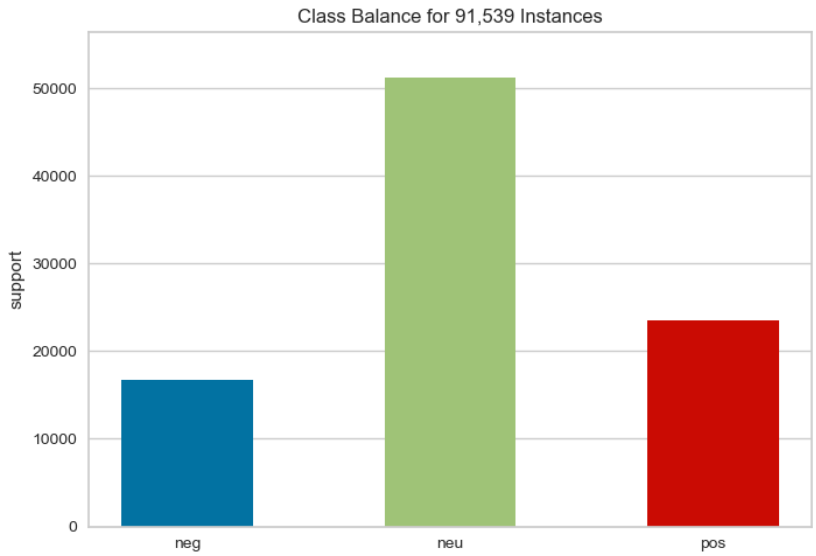


Abbildung 31: Auf dieser Grafik ist die Klassenbalance dargestellt. Dabei ist zunächst der Balken für die negative Klasse (neg) dargestellt, darauf folgen die neutrale (neu) Klasse und die positive (pos). Das die Klassenbalance unausgewogen ist, wurde bereits in der Zusammenführung der Korpora bemerkt. Grafisch ist deutlich zu sehen das mehr als die Hälfte der Daten zur Klasse neutral gehören. Für die anderen Klassen negativ und positiv sind weniger als die Hälfte der neutralen Daten vorhanden.

Mittels t-SNE lassen sich verschiedene Visualisierungen aus dem Korpus umsetzen. Die zuvor dargestellten Grafiken konnten Daten und Verteilungen des Korpus darstellen. Die nächste Abbildung 32 zeigt die Häufigkeitsverteilung (engl. frequency distribution) der Worte. Bevor diese Häufigkeitsverteilung entstehen konnte, wurden dieselben Stoppwörter verwendet, wie in der Keras-Implementierung.

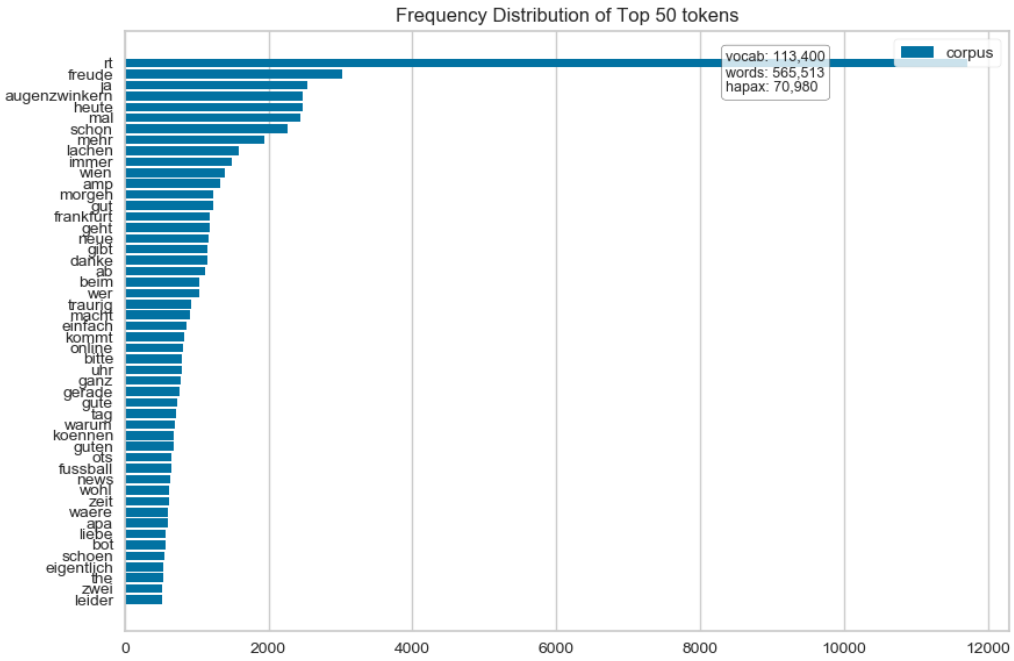


Abbildung 32: In diesem Diagramm wurden die 50 häufigsten Worte dargestellt. Dabei ist festzustellen, dass ein Wort *rt* mit Abstand am häufigsten vorkommt. Dabei handelt es sich um die Abkürzung von Retweet, dem Antworten auf Tweets. Das Wort *rt* kommt in den Worteinbettungen nicht vor und kann somit keine Gewichtung auf die Klassifizierung nehmen. Das zweit- und dritthäufigste Wort lautet *freude* und *augenzwinkern*. Dabei handelt es sich um zuvor festgelegte Worte, welche das dazugehörige Emoticon ersetzt haben. Daran ist auch zu erkennen, wie häufig diese Emoticons verwendet werden. Es sind noch weitere Worte zu finden, welche für die Umwandlung von Emoticons dienten. Dazu zählen in absteigender Reihenfolge, nach den bereits genannten, die Worte *lachen* und *traurig*. Wobei ein Anteil dieser Worte bereits vor der Umwandlung der Emoticons vorhanden gewesen sein könnte. Insgesamt sind wenige Wörter zu sehen, welche allein dargestellt auf ein Sentiment hinweisen könnten wie *gut*, *guten*, *schoen* oder *liebe*. Die übrigen Worte wie beispielsweise *ja*, *heute*, *neue*, *uhr* oder *fussball*, können in negativen, neutralen oder positiven Tweets vorkommen.

Eine weitere Art und Weise der Visualisierung stellt die grafische Darstellung der Lexikalischen Dispersion (engl. lexical dispersion) dar. Durch diesen Ansatz wird es möglich, zu veranschaulichen, an welchen Stellen und wie oft diese Worte in Texten vorkommen (Blackmoore 2015). Für die Abbildung 33 wurden Worte festgelegt und anschließend mittels t-SNE veranschaulicht.

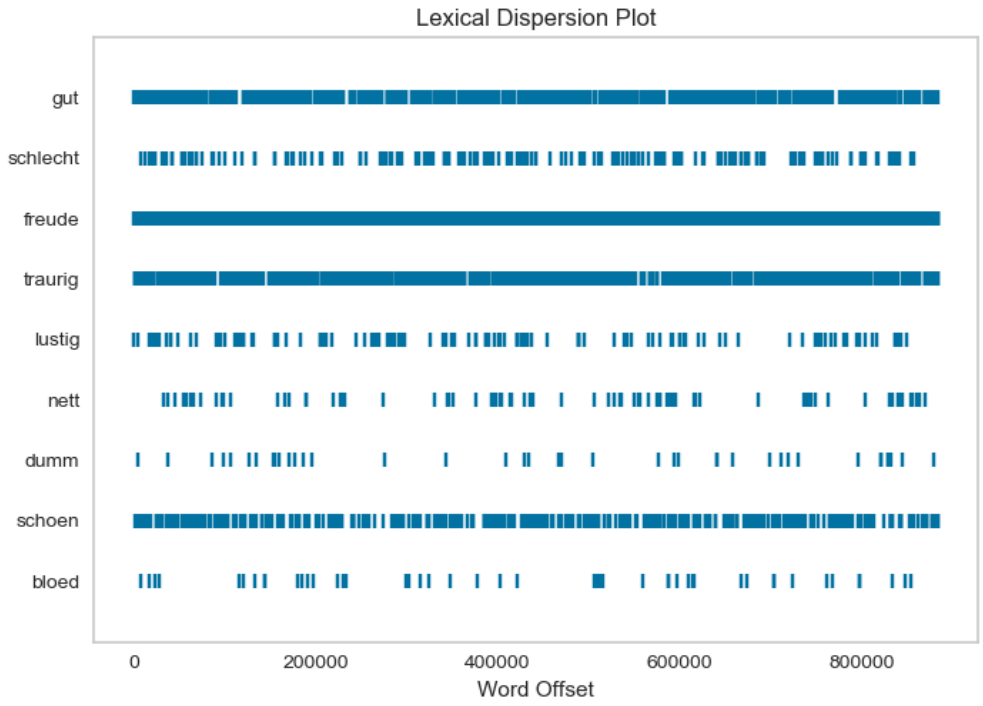


Abbildung 33: Hier ist die Lexikalische Dispersion dargestellt. Dabei wurde verschiedene Worte festgelegt, welche anschließend visualisiert wurden. Dabei wurden unterschiedliche Worte gewählt, die auf einen positiven oder negativen Tweet hindeuten könnten. Positive Worte sind dabei zum Beispiel *gut* oder *schoen*. Anhand der Visualisierung ist zu erkennen, dass die beiden Worte sehr häufig und in einer hohen Frequenz vorkommen. Mögliche negative Worte wie *dumm* oder *bloed* kommen weniger häufig und in einer geringeren Frequenz vor.

Nachdem der Korpus mittels t-SNE untersucht wurde, konnte die Vorverarbeitung der Tweets beendet werden. Die Tweets wurden je nach Sentiment

in eine extra Datei geschrieben. Der folgende Codeausschnitt zeigt, wie dies geschah. Wobei in diesem Schritt mehrmals vorkommende Tweets gelöscht wurden und der Korpora somit aus einzigartigen Tweets besteht.

```
df = pd.read_csv("corpus_6.tsv", sep=",", index_col=False,
    encoding="UTF-8")

negative = df.loc[df["sentiment"] == "0"].drop_duplicates()
...
negative_data = negative.to_csv("negative_data.tsv", sep=",",
    mode="a", index=False, encoding="UTF-8")
...
```

Hier wird dargestellt, wie zunächst der gesamte Korpus (*corpus_6.tsv*) als ein Pandas Dataframe geladen wird. Anschließend werden der Variablen *negative* alle Tweets mit dem Sentiment *0* zugewiesen, wobei mehrfach vorkommende Tweets gelöscht werden. Als nächstes galt es, diese Daten in eine neue Datei *negative_data.tsv* geschrieben. Die drei Punkte weisen drauf hin, dass diese Schritte auch für die anderen Sentimente durchgeführt wurden. In einem weiteren Schritt wurde das Dateiformat von einer TSV-Datei in eine TXT-Datei geändert. Somit lag der Korpus so vor, dass es je eine Datei für das Sentiment gab, also drei Dateien insgesamt. Diese enthielten nur noch die Tweets, für jeden Tweet eine extra Zeile. Nun folgt der letzte Verarbeitungsschritt. Zunächst soll kurz der Code dargestellt werden, für diese Verarbeitung und anschließend die Erläuterung.

```
with open("positive_data.txt", encoding="UTF-8") as f:
  start = 0
  op = ""
  cntr = 1
  for x in f.read().split("\n"):
    with open("pos/" + str(cntr) + "_tweet.txt", "w", encoding="
   UTF-8") as o:
      op = op + x
      o.write(op)
      o.close()
      op = ""
      cntr += 1
```

Mit diesem Code war es möglich die Datei für ein Sentiment einzulesen und anschließend für jede Zeile, also für jeden Tweet, eine extra Datei zu generieren, in dem dieser geschrieben wird. Hier wird die Datei *positive_ data.txt* geöffnet und gelesen. Für jede Zeile in dieser Datei wird der Ordner *pos*, für positiv, geöffnet und hineingeschrieben. Dabei wird jede Zeile (jeder Tweet) in eine Datei mit dem Namen *x_ tweet.txt* geschrieben, wobei x eine Variable ist, die für jede Datei hochgezählt wird.

Das Ergebnis dieser Verarbeitung stellen die drei Ordner *neg*, *neu* und *pos* dar, in denen ausschließlich die dazugehörigen einzigartigen Tweets zu finden sind. Wobei jeder Tweet in einer extra Text-Datei geschrieben steht.

5.2.2 Workflow der Python Implementierung

In diesem Teil wird aufgezeigt, wie die Implementierung funktioniert. Dies wird am Beispiel des CNN Modells getan, denn die Implementierungsansätze sind identisch. Abgesehen von den Architekturen. Es sollen die wichtigsten Bestandteile der Implementierung dargestellt werden.

Eine der wichtigsten Elemente stellte die Ausgabe des F1 Wertes dar. Denn auch die veröffentlichten Ergebnisse von (Cieliebak et al. 2017) beziehen sich auf diesen Wert. Ein großes Problem bestand darin, dass Keras seit der Version 2.0 die *precision*, den *recall* und *f1* entfernt haben. Hierzu wurden viele verschiedene und zum Teil sehr aufwändige Ansätze gefunden, diese Werte weiterhin zu berechnen. Allerdings gibt es einen einfachen Workaround, der dieses Problem behebt. Es war möglich die Funktionen für diese drei Werte aus einer älteren Keras Version direkt in den Quellcode einzufügen und aufzurufen.

Auf dem folgenden Codeausschnitt ist das Einlesen der Tweets und Labels zu sehen. Anschließend folgt die dazugehörige Erläuterung.

```
tweets_dir = "tweets"
train_dir = os.path.join(tweets_dir, "train")

labels = []
texts = []

for label_type in ["neg", "neu", "pos"]:
```

```
  dir_name = os.path.join(train_dir, label_type)
  for fname in os.listdir(dir_name):
    if fname[-4:] == ".txt":
      f = open(os.path.join(dir_name, fname), encoding="UTF-8")
      texts.append(f.read())
      f.close()
      if label_type == "neg":
        labels.append(0)
      elif label_type == "neu":
        labels.append(1)
      elif label_type == "pos":
        labels.append(2)
```

Zunächst wird das Verzeichnis der Tweets angegeben sowie das der Trainingsdaten. Als nächstes werden zwei Arrays angelegt. Einmal das Array *labels[]* für die Labels und das Array *texts[]* für die Tweets. Anschließend folgen einige Schleifen. Zum einen werden die Ordner *neg*, *neu* und *pos* durchlaufen und alle Tweets werden mit dem dazugehörigen Labels in die zuvor angelegten Arrays geschrieben.

Anschließend werden Stoppwörter angewendet und diese aus den Tweets entfernt. Weiterhin wurden Variablen für die maximale Länge der Tweets, der Wortschatzgröße und anderen Parametern definiert. Danach folgte die Tokenisierung, Sequenzierung, das Zero-Padding sowie dem Mischen der Daten. Danach wurden die Worteinbettungen eingelesen und die Vektoren für die Einbettungsschicht erstellt. Hierzu folgt ein Codeausschnitt und anschließend eine Erklärung.

```
embedding_index = {}
with open("fasttext_50mill_tweet.txt", encoding="UTF-8") as f:
  for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype="float32")
    embedding_index[word] = coefs
f.close()

embedding_dim = 300

embedding_matrix = np.zeros((max_words, embedding_dim))
```

```
for word, i in word_index.items():
  embedding_vector = embedding_index.get(word)
    if i < max_words:
      if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector
```

In dem ersten Teil dieses Schrittes wurde der *embedding_index* angelegt. Die Worteinbettungen werden geöffnet, in Zeilen gesplittet und dem *embedding_index* hinzugefügt. Die Dimension der Einbettungen beträgt 300 und wurde durch die 300-Dimensionalen vortrainierten Worteinbettungen vorgegeben. Der erstellten *embedding_matrix* wird die Eingabeschicht definieren und durch die Wortschatzgröße und der Einbettungs-Dimension beeinflusst. Anschließend wir das Keras-Modell erstellt, was in dem nächsten Codeausschnitt zu sehen ist. Die Erklärung erfolgt im Anschluss.

```
model = Sequential()
e = Embedding(max_words, embedding_dim,
                weights=[embedding_matrix],
                input_length=maxlen, trainable=False)
model.add(e)

model.add(Conv1D(filters ,
kernel_size ,
activation="relu",
strides=1))
model.add(MaxPooling1D())

model.add(Flatten())

model.add(Dense(hidden_dims, activation="relu"))
model.add(Dropout(0.3))

model.add(Dense(3, activation="softmax"))
```

Hier wird ein sequentielles Modell erstellt, welches es ermöglicht, verschiedene Schichten aneinanderzureihen. Die erste Schicht stellt die Einbettungsschicht *e* dar. In dieser wird die Wortschatzgröße, Dimensionalität und der Länge von Tweets angegeben. Außerdem wird hier die *embedding_matrix* geladen und *trainable=False* gesetzt, damit die vortrainierten Worteinbet-

tungen angepasst werden. Anschließend folgt eine Faltungsschicht mit 256 Einheiten (*filters*), einer Kernelgröße von 3 (*kernel_size*), der Aktivierungsfunktion ReLU und einem Stride-Wert von 1. Danach folgt eine Schicht zum ebnen der Gewichtungen, welche notwendig dafür ist, dass die dichte Schicht mit 256 Einheiten und einer ReLU-Aktivierungsfunktion weiter arbeiten kann. Die letzte Schicht verfügt über 3 Einheiten und einer Softmax-Aktivierungsfunktion, welche für die Mehrklassen-Klassifizierung eingesetzt wird, bei der die Datenbeispiele nur einer Klasse angehörig sind. Außerdem wird Dropout mit einem Wert von 0,3 verwendet.

Anschließend folgt die Kompilierung des Modells, wobei hier die Loss-Funktion, der Optimierungsalgorithmus und die Metriken. In dem folgenden Codeausschnitt wird dies umgesetzt und danach erläutert.

```
sgd = optimizers.SGD(lr=0.001, decay=1e-6,
                     momentum=0.9, nesterov=True)

model.compile(loss="categorical_crossentropy",
              optimizer=sgd,
              metrics=["accuracy", f1])
```

Hier wird der Optimierungsalgorithmus SGD eingesetzt, welcher mit den empfohlenen Konfigurationen der Keras Dokumentation eingestellt wurde. Dabei wird eine Lernrate, Lernratenzerfall sowie Momentum mit Nesterov. Das Modell wird mit der Loss-Funktion *categorical_crossentropy* ausgeführt und es werden die Metriken für Genauigkeit (*accuracy*) und den F1 Wert (*f1*) ausgegeben.

Der letzte Codeausschnitt stellt das Training des Modells dar.

```
callbacks = [EarlyStopping(monitor="val_f1",
                           patience=30, mode="max",
                           restore_best_weights=True)]

history = model.fit(x_train, y_train,
                    batch_size=64,
                    epochs=300,
                    validation_split=0.2,
```

```
callbacks=callbacks)
```

Hier wurde zunächst das Early-Stopping implementiert, wobei der F1 Wert überwacht wurde. Die Begründung liegt darin, dass während des Trainings beobachtet werden konnte, wie der Loss beginn zu steigen, die Genauigkeit und F1 Werte allerdings noch für einige Epochen anstiegen. Nach einem maximalen F1 Wert werden 30 Epochen durchlaufen, bevor das Training abbricht und auf bestmöglichen Gewichtungen zurückgesetzt werden. Außer der F1 Wert wird übertroffen. Mittels der *fit()* Methode wird das Modell trainiert. Hierfür werden die Tweets und die Labels, eine Batch-Größe, die Epochen und ein Validierungs-Split von 20% angegeben. Durch *callbacks* kann das zuvor definierte Early-Stopping integriert werden.

Der Code für die Testdaten wurde anhand desselben Schemata organisiert und umgesetzt. In der Implementierung wurde noch mehr Quellcode geschrieben. Zum Beispiel ist es mit Keras möglich, Modelle und deren Gewichtungen zu speichern und laden. Durch die *summary()* Methode lässt eine Zusammenfassung des Modells in der Konsole ausgeben. Mittels das Objekt *history* Objekt lassen sich die Genauigkeit, der Loss und F1 Wert ebenfalls in der Konsole ausgegeben oder als Diagramm darstellen.

6 Experimente und deren Resultate

In diesem Kapitel werden die zuvor definierten Experimente durchgeführt. Dabei wird mit einem ersten Experiment gestartet, um einen Startwert mit denselben Einstellungen für das LSTM und CNN zu erhalten. Anschließend werden nacheinander die Parameter mit unterschiedlichen Einstellungen getestet und die Ergebnisse in tabellarischer Form zusammengefasst.

6.1 Startkonfiguration der Modelle

In den Experimenten werden die unterschiedlichen Parameter anlehnend an der Dokumentation von Keras belassen. Dabei ging es zunächst darum, einen Startpunkt zu haben, um Ergebnisse zu erhalten, an denen die weiteren Resultate verglichen werden können. Die Tabelle 9 stellt eine Übersicht dar, für die Startparameter. Diese wurden in den nachstehenden Experimenten mit verschiedenen Konfigurationen untersucht.

Parameter	**CNN**	**LSTM**
Wortschatzgröße	10.000	10.000
Tweet Länge	100	100
Batchgröße	64	64
Lernrate	0,001	0,001
Optimierung	SGD	SGD
Dropout	0,3	0,3
LSTM Einheiten	-	256
CNN Filter	256	-
Dichte Schichten	1	1
Einheiten in dichten Schichten	256	256
Regularisierung mittels L1 und L2	Nein	Nein

Tabelle 9: Übersicht aller Parameter der CNN und LSTM-Modelle, welche in den folgenden Experimenten untersucht werden. Zunächst wurden identische Werte eingesetzt. Dabei sind die verschiedenen Parameter aufgeführt und die Werte aus der Implementierung für das CNN und LSTM übertragen worden.

6.2 Mit der Anzahl von Schichten experimentieren

Die Architektur des ersten Experiments bestand für das CNN aus einer Fahltungs- (kurz conv), Pooling- (kurz pool) und einer dichten (kurz dense) Schicht. Bei dem LSTM Modell wurde eine LSTM (kurz lstm) und eine dichte Schicht genutzt. In diesen Tests wurden alle anderen Parameter bei den zuvor dargestellten Einstellungen beibehalten.

Die Ergebnisse der Experimente werden in Tabelle 10 abgebildet. Die Resultate verändern sich nur in einem sehr kleinen Bereich. Die Anzahl der Schichten nahm keinen sehr großen Einfluss darauf, dass sich der F1 Wert verbessert. Allerdings stieg mit der Anzahl der Schichten auch die Zeit für das Training. Die abgebildeten Ergebnisse stammen aus der Validierung.

Embedding	**Modell**	**Schichten**	**Genauigkeit**	**F1 Wert**
Word2Vec	CNN	1 conv 1 pool 1 dense	63,88%	58,37%
		2 conv 2 pool 1 dense	60,07%	**58,90%**
		3 conv 2 pool 1 dense	60,30%	58,55%
		3 conv 3 pool 1 dense	59,51%	58,16%
	LSTM	1 lstm 1 dense	60,54%	57,63%
		2 lstm 1 dense	60,18%	57,61%
		2 lstm 2 dense	60,15%	**58,90%**
		3 lstm 1 dense	54,47%	54,47%
fastText	CNN	1 conv 1 pool 1 dense	61,29%	**58,59%**
		2 conv 2 pool 1 dense	57,75%	55,97%
		3 conv 2 pool 1 dense	60,37%	57,71%
		3 conv 3 pool 1 dense	56,08%	52,41%
	LSTM	1 lstm 1 dense	61,80%	60,43%
		2 lstm 1 dense	61,92%	60,32%
		2 lstm 2 dense	62,06%	**60,47%**
		3 lstm 1 dense	61,24%	60,10%

Tabelle 10: Tabellarische Zusammenfassung der Ergebnisse für die Experimente mit einer unterschiedlichen Anzahl von Schichten während der Validierung. Dabei wurde je für das CNN und das LSTM vier verschiedene Kombinationen von Schichten und deren Anzahl getestet. Die Ergebnisse unterscheiden sich nur mit einer sehr kleinen Steigerung oder einem Abstieg. Die besten Ergebnisse konnte LSTM mit fastText erreichen, wobei mit allen Kombinationen über 60% erreicht wurden. Danach folgen je das CNN und LSTM mit Word2Vec für Ergebnisse in Höhe von 58,90%. Das CNN mit fastText hat 58,59% erreicht.

Die besten Ergebnisse entscheiden sich nur in einem kleinen Bereich. Das LSTm mit fastText konnte dabei die besten Resultate erbringen. Da in allen Modell- und Worteinbettungs-Kombinationen gute Ergebnisse mit den einfachsten Architekturen erreicht werden konnten, soll diese für die nächsten Experimente als Architektur dienen. Das CNN wird aus einer Faltungs-, Pooling- und dichten Schicht bestehen und das LSTM aus einer LSTM- und einer dichten Schicht. Dies wird mit dem ersparten Rechenaufwand, bei nahezu gleicher Leistung begründet.

6.3 Experimente zur Wortschatzgröße

Bei dem nächsten Experiment sollte untersucht werden, welchen Einfluss die Wortschatzgröße nimmt. Dabei wurden verschiedene Größen getestet. Eine Hypothese bezüglich der Wortschatzgröße konnte man bereits vor den Tests aufstellen. Die Textdaten stammen aus Tweets. Die vortrainierten Worteinbettungen stehen ebenfalls bereit. Nun ist es möglich, dass über Twitter Worte verwendet werden, die es nicht in dem Word2Vec oder fastText Modell gibt. Auch Worte die einen Schreib- oder Tippfehler haben, können somit nicht in dem Modell gefunden werden. Daher könnte eine Hypothese wie folgt lauten. Ein großer Wortschatz kann die Leistung erhöhen, da so die Chancen erhöht werden, möglichst viele Worte in dem Modell zu finden und für das Training zu nutzen. In der Tabelle 11 sind alle Ergebnisse zu diesen Experimenten. Nur die in der Tabelle dargestellten Parameter wurden verändert, andere blieben unverändert.

Embedding	Modell	Wortschatzgröße	Genauigkeit	F1 Wert
Word2Vec	CNN	20000	58,53%	56,34%
		40000	60,14%	57,93%
		60000	59,80%	**57,93%**
		80000	60,13%	57,68%
	LSTM	20000	59,40%	57,05%
		40000	60,80%	58,68%
		60000	61,19%	**59,17%**
		80000	60,33%	58,44%
fastText	CNN	20000	60,50%	59,31%
		40000	60,83%	59,38%
		60000	60,95%	60,05%
		80000	61,30%	**61,29%**
	LSTM	20000	62,15%	60,84%
		40000	63,50%	61,78%
		60000	64,79%	**63,67%**
		80000	64,59%	63,02%

Tabelle 11: Tabellarische Zusammenfassung der Ergebnisse für die Experimente mit verschiedenen Wortschatzgrößen. Die zuvor aufgestellte Hypothese könnte als bestätigt angesehen werden. Bei einer Wortschatzgröße von 20.000 sind die schlechtesten Ergebnisse für den F1 Wert resultiert und das für alle Modelle und Einbettungen. Bei ansteigender Größe des Wortschatzes steigt auch der F1 Wert an. Ab einer Wortschatzgröße von 60.000 ist keine Verbesserung mehr zu erkennen, außer bei dem CNN mit fastText. Auch in diesem Experiment ist zu sehen, dass die fastText Worteinbettungen zu besseren Ergebnissen führen.

Für den Parameter der Wortschatzlänge wurden 60.000 ausgewählt. Dieser Wert brachte bei der Mehrheit der Experimente die besten Ergebnisse.

6.4 Mit der Länge von Tweets experimentieren

In einem weiteren Experiment wurde untersucht, welchen Einfluss die Länge der Tweets nehmen. Kein Tweet ist länger als 30 Wörter, somit wurden nur Tweet Längen untersucht, welche kürzer als 30 waren. Wenn ein Tweet weniger als die festgelegte Länge aufweist, dann werden die fehlenden Worte mit Nullen aufgefüllt. Wie zuvor erwähnt wurde mit einer Wortschatzgröße von

60.000 gearbeitet. Die Tabelle 12 stellt die Ergebnisse für die unterschiedlichen Längen dar.

Embedding	**Modell**	**Tweet Länge**	**Genauigkeit**	**F1 Wert**
Word2Vec	CNN	10	59,40%	56,26%
		20	60,56%	57,89%
		30	60,14%	**57,93%**
	LSTM	10	59,74%	57,68%
		20	59,68%	57,63%
		30	60,80%	**58,68%**
fastText	CNN	10	60,44%	59,61%
		20	60,91%	59,04%
		30	61,45%	**59,75%**
	LSTM	10	62,45%	61,41%
		20	64,08%	63,08%
		30	64,17%	**63,47%**

Tabelle 12: Tabellarische Zusammenfassung der Ergebnisse für die Experimenten mit verschiedenen Tweet Längen. Die besten Resultate für alle Modelle und Worteinbettungen konnte mit einer Tweet Länge von 30 erreicht werden. Auch bei der Tweet Länge konnte die fastText Worteinbettungen bessere Ergebnisse erreichen, besonders das LSTM.

Da alle Modelle und Worteinbettungen die beste Leistung bei einer Tweet Länge von 30 erreichten, soll diese Einstellung beibehalten werden.

6.5 Verschiedene Optimierungsverfahren und Lernraten

In den bisherigen Experimenten wurde SGD als Optimierung eingesetzt, mit einer Lernrate von 0,001. Keras bietet verschiedene Optimierungsverfahren an, welche auch durch einige Parameter wie der Lernrate anpassbar sind. Eine allgemeine Beobachtung für CNN und LSTM besteht darin, dass eine große Lernrate, wie 0,1 zu den schlechten Ergebnissen führt, da es nach wenigen Epochen in Überanpassung endet. Die Modelle lernten zu schnell, der Loss sank rasant und stieg ebenso rasch an und endete in einer Genauigkeit sowie F1 Werten unter 50%, während der Validierung. In Tabelle 13 sind die besten Ergebnisse für die getesteten Optimierungsverfahren mit der dazugehörigen Lernrate dargestellt.

Embedding	Modell	Opt.	Lernrate	Genauigkeit	F1 Wert
Word2Vec	CNN	SGD	0,001	60,56%	57,89%
		RMSprop	0,00001	60,63%	**58,50%**
		Adagrad	0,0001	60,41%	57,20%
		Nadam	0,00001	59,34%	57,09%
	LSTM	SGD	0,001	59,68%	57,63%
		RMSprop	0,00001	61,12%	58,09%
		Adagrad	0,0001	53,78%	53,58%
		Nadam	0,00001	61,33%	**58,44%**
fastText	CNN	SGD	0,001	61,54%	**59,75%**
		RMSprop	0,00001	60,25%	58,84%
		Adagrad	0,0001	60,14%	56,09%
		Nadam	0,00001	60,91%	59,26%
	LSTM	SGD	0,001	64,17%	**63,47%**
		RMSprop	0,00001	64,41%	63,21%
		Adagrad	0,0001	63,29%	60,01%
		Nadam	0,00001	63,71%	62,18%

Tabelle 13: Tabellarische Zusammenfassung der Ergebnisse für die Experimente mit verschiedenen Optimierungsalgorithmen und Lernraten. Für die Modelle mit Word2Vec Worteinbettungen schnitt RMSprop für das CNN und Nadam für das LSTM am besten ab. Bei den Implementierungen mit fastText wurden die besten Ergebnisse mit SGD erreicht. Insgesamt scheinen auch in diesem Experiment die Worteinbettungen mittels fastText eine bessere Leistung zu erbringen.

Die nächsten Experimente wurden weiterhin mit SGD durchgeführt, allerdings konnte auch RMSprop nahezu gleichwertige Ergebnisse erreichen. Daher sollen diese beiden Optimierungsalgorithmen für die Testdaten später eingesetzt werden.

6.6 Regularisierung mittels L1 und L2

In diesen Experimenten wurde die Regularisierung mittels L1 und L2 untersucht. Dabei wurde zunächst L1 implementiert und anschließend L2. Während der Regularisierung mit L1 und L2 wurde die Dropout-Schicht entfernt. In der Tabelle 14 sind die Ergebnisse für die Experimente mit L1 dargestellt. Dabei wurden verschiedene Werte als Parameter angegeben.

Embedding	**Modell**	**Wert**	**Genauigkeit**	**F1 Wert**
Word2Vec	CNN	0,0001	60,27%	**58,29%**
		0,001	54,55%	53,28%
		0,01	54,93%	54,65%
		0,1	54,48%	54,78%
	LSTM	0,0001	59,81%	**57,81%**
		0,001	54,66%	57,45%
		0,01	55,73%	54,46%
		0,1	54,18%	53,82%
fastText	CNN	0,0001	61,19%	**59,70%**
		0,001	60,71%	57,09%
		0,01	54,35%	54,77%
		0,1	54,50%	54,78%
	LSTM	0,0001	54,52%	**54,75%**
		0,001	54,46%	54,75%
		0,01	54,42%	54,73%
		0,1	54,71%	53,28%

Tabelle 14: Tabellarische Zusammenfassung der Ergebnisse für die Experimente mit verschiedenen Werten für L1. In diesen Resultaten ist ein Trend zu sehen. Je kleiner der Wert für L1 ist, desto besser ist das Ergebnis. Sobald sich der Wert vergrößert verschlechtert sich das Ergebnis oder bleibt gleich. Hier ist das LSTM mit fastText das schlechteste, wobei das CNN mit fastText das beste Ergebnis liefert. Danach folgt das CNN und LSTM mit Word2Vec Worteinbettungen.

Ähnliche Einstellungen wurden für die Regularisierung mittels L2 eingesetzt. Diese Ergebnisse konnten die mit L1 übertreffen und sind in Tabelle 15 zusammengefasst. Ein ähnlicher Trend wie bei der Regularisierung mittels L1 ist nicht zu erkennen.

Embedding	Modell	Wert	Genauigkeit	F1 Wert
Word2Vec	CNN	0,0001	60,33%	58,61%
		0,001	61,00%	58,99%
		0,01	60,78%	59,15%
		0,1	61,38%	**59,83%**
	LSTM	0,0001	60,94%	**59,15%**
		0,001	61,14%	58,99%
		0,01	61,15%	58,69%
		0,1	60,56%	59,09%
fastText	CNN	0,0001	61,38%	59,63%
		0,001	60,45%	59,30%
		0,01	62,08%	60,83%
		0,1	62,20%	**61,14%**
	LSTM	0,0001	64,04%	62,81%
		0,001	64,21%	63,00%
		0,01	64,83%	**63,48%**
		0,1	64,61%	63,05%

Tabelle 15: Tabellarische Zusammenfassung der Ergebnisse für die Experimente mit verschiedenen Werten für L2. Es ist zu erkennen, dass das CNN mit Word2Vec und fastText mit einem großen Wert für L2 die besten Ergebnisse erreichten. Das LSTM mit Word2Vec hat die besten Ergebnisse mit einem kleinen L2 Wert erbracht und mit fastText mit dem zweitgrößten Wert. Wobei die Abstände zwischen den Ergebnissen der einzelnen Modelle nur sehr gering ausfallen.

6.7 Zusammenfassung weiterer Experimente und der Ergebnisse für die Validierung

Neben den zuvor beschriebenen Experimenten wurden noch weitere Parameter verändert und getestet. Die nicht dokumentierten Ergebnisse fielen schlechter aus oder im besten Fall, gleichbleibend. Dabei wurde beispielsweise für das CNN eine unterschiedliche Kernel-Größe getestet. Die Größen lauteten 1, 3 und 5. Wobei eine Kernel-Größe von 3 die mit Abstand besten Ergebnisse errungen hat. Für das LSTM wurde ein bidirektionaler Wrapper an den LSTM-Schichten getestet und führte zu keiner Leistungssteigerung.

Andere Experimente wurden mit verschiedenen Anzahlen von Einheiten in Schichten getätigt. Wobei eine Anzahl von 128 statt 256 Einheiten ähnliche Ergebnisse erreichte. Eine noch kleinere Anzahl verschlechterte die Resultate. Auch mit einer größeren Anzahl der Einheiten konnte nicht die Leistung, sondern nur der Rechenaufwand und die Trainingszeit erhöht werden.

Weitere Parametereinstellungen die ausgeführt wurden, bestanden in dem Einsatz von L1 und L2 gemeinsam, wodurch ebenfalls keine besseren Ergebnisse erreicht wurden. In einem Keras-Tutorial wurde außerdem eine Kombination von CNN und LSTM Architektur eingesetzt. Nach der Faltungs- und Pooling-Schicht wurde eine LSTM und dann eine dichte Schicht eingefügt. Auch hiermit konnten die zuvor dargestellten Resultate nicht überboten werden.

Für die Testdaten sollen verschiedene Parametereinstellungen und Kombinationen ausgewählt werden. Die Optimierungsalgorithmen SGD und RMSprop haben gute Ergebnisse für die Validierung eingebracht. Weiterhin soll die Regularisierung mittels Dropout, L1 und L2 eingesetzt werden. Diese Parameter werden unter der Verwendung von zwei unterschiedlichen Architekturen verwendet. Zum einen die einfache Architektur, bei der im CNN eine Faltungs-, Pooling-, und dichte Schicht ist sowie CNN mit zwei Faltungs-, Pooling- und eine dichte Schicht. Für das LSTM wird eine LSTM- und dichte Schicht sowie zwei LSTM- und dichte Schichten Anwendung finden.

6.8 Ergebnisse für den Testdatensatz

In der Tabelle 16 sind die verschiedenen Parameter für die Modelle aufgeführt, welche für die Testdaten eingesetzt wurden. Dabei sind verschiedene Parametereinstellungen dargestellt. Die Tests werden mit einer Architektur bestehend aus einer Faltungs-, Pooling- und dichten Schicht sowie zwei Faltungs-, Pooling- und einer dichten Schicht für das CNN umgesetzt. Bei dem LSTM wurden eine LSTM- und dichte Schicht sowie zwei LSTM- und dichte Schichten implementiert. Im folgenden sollen diese Architekturen mittels der Beschreibung kleine und große Architektur zusammengefasst werden. Weiterhin wurden Tests mit SGD und RMSprop durchgeführt. Für die Regularisierung wurde Dropout genutzt, was aus dem Modell entfernt wurde, wenn L1 oder L2 eingesetzt wurden.

Parameter	**Konfiguration**
Lernrate SGD	0,001
Lernrate RMSprop	0,00001
Einheiten LSTM Schicht	256
Einheiten CNN Schicht	256
Einheiten dichte Schicht	256
Anzahl Faltungs-/Pooling-Schicht im CNN	1/1 und 2/2
Anzahl LSTM-/dichte Schicht im LSTM	1/1 und 2/2
Dropout	0,3
L1	0,0001
L2	0,1

Tabelle 16: Hier ist eine tabellarische Zusammenfassung der Parameterkonfigurationen zu sehen, welche für die Testdaten eingesetzt wurden. Dabei handelt es sich um Parametereinstellungen die zuvor in den Experimenten an den Trainings- und Validierungsdaten Anwendung fanden. Es handelt sich um Optimierungsalgorithmen SGD und RMSprop mit einer dazugehörigen Lernrate, die Anzahl der Schichten im CNN und LSTM Modell, Anzahl der Faltungs-, Pooling-, LSTM- und dichten Schichten sowie Regularisierung mittels Dropout, L1 und L2.

Im folgenden werden die Ergebnisse für die Testdaten dargestellt. Zunächst werden die Ergebnisse in Tabelle 17 der kleinen CNN und LSTM Architektur mit SGD dargestellt.

Embedding	**Modell**	**Genauigkeit**	**F1 Wert**
Word2Vec	CNN	49,21%	45,10%
	LSTM	49,17%	45,70%
fastText	CNN	53,08%	**50,92%**
	LSTM	51,56%	49,65%

Tabelle 17: Hier sind die Ergebnisse für die kleine CNN und LSTM Architektur mit dem SGD Optimierungsalgorithmus dargestellt. Dabei fällt auf, das die Worteinbettungen mittels fastText bessere Ergebnisse erreichen und das CNN mit fastText die besten Ergebnisse von allen Einbettungen und Modellen erbrachte. Das beste Ergebnis lautet 50,92%.

In der nachstehenden Tabelle 18 wurden dieselben Architekturen wie zuvor

eingesetzt, allerdings mit dem Optimierungsalgorithmus RMSprop.

Embedding	**Modell**	**Genauigkeit**	**F1 Wert**
Word2Vec	CNN	49,87%	46,83%
	LSTM	50,08%	46,92%
fastText	CNN	51,36%	**49,07%**
	LSTM	50,66%	48,43%

Tabelle 18: In dieser Tabelle sind die Ergebnisse der kleinen Architekturen dargestellt, wobei der Optimierungsalgorithmus RMSprop eingesetzt wurde. Es ist festzustellen, dass die Worteinbettungen mittels Word2Vec einen kleinen Anstieg verzeichnen, der etwas über 1% beträgt. Die Modelle mit den fastText Worteinbettungen haben sich hingegen um etwas mehr als 1% verschlechtert. Das beste Ergebnisse konnte wieder mit CNN und fastText Worteinbettungen erreicht werden, 49,07%.

In Tabelle 19 sind Ergebnisse für die kleinen Architekturen und der Regularisierung mittels L1 dargestellt. Hier ist eine deutlicher Anstieg für den F1 Wert und der Genauigkeit zu erkennen.

Embedding	**Modell**	**Genauigkeit**	**F1 Wert**
Word2Vec	CNN	54,61%	54,60%
	LSTM	54,49%	54,60%
fastText	CNN	54,60%	**54,61%**
	LSTM	54,67%	54,46%

Tabelle 19: Hier sind die Ergebnisse für die Tests mit dem Optimierungsalgorithmus SGD und der Regularisierung mittels L1 zu sehen. Dabei ist ein deutlicher Anstieg der Genauigkeit und dem F1 Wert zu erkennen. Die Resultate liegen sehr nahe beieinander. Auch hier konnte CNN mit fastText Worteinbettungen, mit 54,61% die besten Resultate erreichen, wobei die Ergebnisse mit den Word2Vec Einbettungen nur 0,01% geringer ausfielen. Auch das LSTM mit fastText liegt nur dicht dahinter. Weiterhin scheint die Regularisierung mittels L1 zu einer besseren Generalisierung geführt zu haben, als der bloße Einsatz von Dropout.

Die tabellarische Übersicht 20 zeigt die Ergebnisse für die kleinen Architekturen mit dem Optimierungsalgorithmus RMSprop und der Regularsisierung

mittels L1. Es ist festzustellen, dass L1 in Kombination mit SGD zu besseren Ergebnissen führte.

Embedding	**Modell**	**Genauigkeit**	**F1 Wert**
Word2Vec	CNN	49,52%	44,99%
	LSTM	52,72%	49,71%
FastText	CNN	49,53%	44,99%
	LSTM	51,83%	**50,62%**

Tabelle 20: Die Ergebnisse mit L1 und RMSprop sind nicht mit denen zuvor erhaltenen Resultaten mittels L1 und SGD zu vergleichen. Es wurden schlechtere Werte für die Genauigkeit und F1 erhalten. Dabei scheinen die LSTM Modelle mit RMSprop eine bessere Leistung zu erbringen, als die CNNs. Auch in diesem Test wurde das beste Ergebnis mittels der fastText Worteinbettungen erhalten und lauten 50,62%.

In der nachstehenden Tabelle 21 werden die Ergebnisse für die Tests mit den kleinen Architekturen, SGD und Regularisierung mittels L2 zusammengefasst.

Embedding	**Modell**	**Genauigkeit**	**F1 Wert**
Word2Vec	CNN	50,09%	46,37%
	LSTM	49,89%	46,03%
fastText	CNN	51,69%	**50,58%**
	LSTM	51,56%	48,88%

Tabelle 21: Hier sind die Ergebnisse für die kleinen CNN und LSTM Architekturen mit SGD und L2 dargestellt. Dabei ist wieder zu erkennen, dass die fastText Worteinbettungen zu besseren Ergebnissen führten. Wobei das CNN mit 50,58% das beste Ergebnis erreichte.

Es folgen die Ergebnisse für die kleinen Architekturen mit RMSprop und L2 in Tabelle 22. Dabei ist zu erkennen, dass erneut die Modelle mit den fastText Worteinbettungen zu besseren Resultaten geführt haben.

Embedding	**Modell**	**Genauigkeit**	**F1 Wert**
Word2Vec	CNN	50,62%	46,50%
	LSTM	49,68%	46,68%
fastText	CNN	51,31%	49,64%
	LSTM	53,61%	**52,61%**

Tabelle 22: In dieser Übersicht sind die Ergebnisse für die kleinen Architekturen mit RMSprop und L2 zusammengefasst. Dabei wurden mittels fastText Einbettungen die besten Ergebnisse erreicht und das CNN mit 50,58% das beste Ergebnis von allen.

Diese Tests wurden auch mit den etwas größeren Architekturen durchgeführt. Dabei wurden für das CNN zwei Faltungs- und Pooling- sowie eine dichte Schicht verwendet. Für das LSTM wurden zwei LSTM- und dichte Schichten eingesetzt. Die Zusammenfassung der Ergebnisse für die große Architektur und SGD sind in Tabelle 23 dargestellt.

Embedding	**Modell**	**Genauigkeit**	**F1 Wert**
Word2Vec	CNN	54,60%	**52,32%**
	LSTM	50,70%	48,40%
fastText	CNN	51,78%	50,03%
	LSTM	51,55%	49,99%

Tabelle 23: Hier sind die Resultate für die größeren Architekturen mit SGD als Optimierungsalgorithmus. Die Ergebnisse sind denen der kleinen Architekturen überlegen. Das CNN konnte mit den beiden Worteinbettungen die besten Ergebnisse erreichen, wobei hier erstmals die Word2Vec Einbettungen zu den besten Ergebnissen führten, in Höhe von 52,32%. Danach folgt das CNN mit fastText 50,03%, wobei das LSTM mit fastText mit 49,99% sehr dicht an dem dem Ergebnis dran ist.

In der nachstehenden Tabelle 24 sind die Ergebnisse derselben Architekturen und RMSprop als Optimierungsalgorithmus aufgeführt. Dabei ist wieder zu erkennen, dass mit diesem Optimierungsalgorithmus schlechtere Ergebnisse als mit SGD resultieren.

Embedding	Modell	Genauigkeit	F1 Wert
Word2Vec	CNN	49,57%	47,52%
	LSTM	50,17%	47,94%
fastText	CNN	50,97%	**49,47%**
	LSTM	49,21%	49,21%

Tabelle 24: Diese Tabelle fasst die Ergebnisse für die größeren Architekturen mit dem Optimierungsalgorithmus RMSprop zusammen. Dabei erbringen die Modelle mit den fastText Worteinbettungen bessere Ergebnisse als die Word2Vec Einbettungen. Die besten Ergebnisse liefert das CNN mit 49,47%, dicht dahinter das LSTM mit 49,21%.

Die nachstehende Tabelle 25 stellt eine Übersicht der Ergebnisse für die Tests mit den größeren Architekturen, SGD und Regularisierung mittels L1 dar.

Embedding	Modell	Genauigkeit	F1 Wert
Word2Vec	CNN	54,17%	54,07%
	LSTM	54,60%	**54,60%**
fastText	CNN	54,60%	54,01%
	LSTM	54,61%	**54,60%**

Tabelle 25: In dieser Tabelle werden die Ergebnisse für die großen Architekturen in Kombination mit SGD und L1 dargestellt. Auch hier scheint L1 zu den besten Ergebnissen zu führen. Dabei liegen alle Ergebnisse über 54%. Das LSTM erreichte mit den beiden Worteinbettungen die Höchstwerte von 54,60%. Die CNN Architekturen liegen jedoch knapp dahinter.

Tabelle 26 stellt eine Übersicht der Ergebnisse für die größeren Architekturen, RMSprop und L1 dar. Dabei ist ein Unterschied zu den kleineren Architekturen zu erkennen. Bei diesen sind die Ergebnisse in der Kombination mit RMSprop insgesamt gesunken.

Embedding	Modell	Genauigkeit	F1 Wert
Word2Vec	CNN	50,03%	45,84%
	LSTM	54,60%	**54,60%**
fastText	CNN	52,41%	49,17%
	LSTM	54,61%	**54,60%**

Tabelle 26: Diese Übersicht stellt die Ergebnisse für die größeren CNN und LSTM Architekturen dar, welche mit RMSprop und L1 umgesetzt wurden. Dabei ist zu erkennen, dass nicht wie bei den kleinen Architekturen und der hier verwendeten Parameterkonfiguration alle Ergebnisse sinken. Hier konnten die LSTM Modelle mit beiden Worteinbettungen die besten Werte erreichen, 54,60%.

In der Tabelle 27 sind die Resultate für die größeren Architekturen mit SGD und L2 abgebildet.

Embedding	Modell	Genauigkeit	F1 Wert
Word2Vec	CNN	54,60%	**52,08%**
	LSTM	50,26%	47,38%
fastText	CNN	52,35%	50,69%
	LSTM	51,22%	49,43%

Tabelle 27: Hier werden Ergebnisse für die größeren Architekturen mit SGD und L2 zusammengefasst. Die CNN Modelle konnten mit beiden Worteinbettungen die besseren Ergebnisse erreichen, wobei das CNN Modell mit den Word2Vec Einbettungen mit 52,08% den Höchstwert darstellt. Die LSTM Modelle schneiden hier etwas schlechter ab, wobei die fastText Einbettungen noch einen etwas besseres Resultat lieferten als Word2Vec.

Die folgende Tabelle 28 bietet eine Übersicht der Resultate für die größeren Architekturen mit RMSprop und L2.

Embedding	**Modell**	**Genauigkeit**	**F1 Wert**
Word2Vec	CNN	50,88%	48,57%
	LSTM	51,42%	48,61%
fastText	CNN	51,73%	50,08%
	LSTM	53,66%	**52,54%**

Tabelle 28: Hier sind die Ergebnisse für die größeren Architekturen mit dem Einsatz von RMSprop und L2 zu sehen. Dabei erbringen die Modelle mit fastText bessere Ergebnisse und das LSTM mit 52,54% die besten Resultate.

Es wurden verschiedene Parametereinstellungen an dem Testdatensatz untersucht. Dabei kamen zwei Optimierungsalgorithmen SGD und RMSprop zum Einsatz, eine unterschiedliche Anzahl von Schichten der Faltungs-, Pooling-, LSTM- und dichten Schichten und Regularisierung mittels Dropout, L1 und L2.

Die besten Ergebnisse mit der kleineren Architektur wurden mittels L1 und SGD erreicht. Dabei lag der Höchstwert bei 54,61% für das CNN Modell mittels fastText Worteinbettungen. Das zweitbeste Resultat wurde mit dem LSTM Modell und RMSprop sowie L2 erreicht. Der Wert lag bei 52,61%. Mit den größeren Architekturen konnte je mit SGD und RMSprop und L1 Regularisierung ein Höchstwert von 54,60% erreicht werden. Dabei sind diese Ergebnisse mittels des LSTM mit Word2Vec und fastText erreicht worden. Das zweitbeste Ergebnis der größeren Architekturen konnten mit RMSprop und L2 durch das LSTM mit fastText gemessen werden, hier lag der Wert bei 52,54%.

Diese Ergebnisse sind durchaus verbesserbar. In dem Kapitel 2.5.4 wurden einige publizierte Resultate erörtert. Die F1-Werte hier sind durch alle Klassen berechnet, was sich zu den in (Cieliebak et al. 2017) unterscheidet, welche mittels $(F1_{pos} + F1_{neg})/2$ berechnet wurden. Weiterhin wurde dort berichtet, dass der F1-Wert mit allen drei Klassen 4,42% höher ausfiel. Daher könnte davon ausgegangen werden, dass ihr schlechtestes Ergebnis von 47,30% bei allen drei Klassen sich auf 51,72% erhöht ($47,30\% + 4,42\%$). Somit wären die hier erhaltenen Höchstwerte vergleichbar. Ein anderes Ergebnis stammte von (Mozetič et al. 2016) und lautet 53,60%, welches um 1% übertroffen werden konnte.

Die Leistung der Modelle lässt sich noch steigern. Dafür gibt es verschiedene Möglichkeiten und vorgehen. Ein Unterschied zur Arbeit von (Mozetič et al. 2016) lag zum Beispiel darin, dass verschiedene Emoticons in ein Wort umgewandelt wurden, welches dieses Emoticon repräsentieren sollte. In der genannten Arbeit wurden alle positiven Emoticons in das Wort positiv und negative Emoticons in das Wort negativ umgewandelt, was einen großen Einfluss auf deren Frequenz und Gewichtung nehmen würde. Ein weiterer Unterschied besteht zu der Arbeit von (Cieliebak et al. 2017) darin, dass dort ein komplexeres System eingesetzt wurde sowie eigene Worteinbettungen mittels 300.000 Millionen Tweets trainiert wurden. Die hier eingesetzten fastText Worteinbettungen von nur 50.000 Millionen Tweets. Weitere Möglichkeiten, wie die Leistung und Qualität der Modelle gesteigert werden kann, soll in den folgenden Kapiteln beschrieben werden. Dabei wird zunächst der LIME-Ansatz und im Anschluss die Rastersuche (engl. grid search) von Hyperparametern prototypisch implementiert und erläutert.

6.9 Prototypische Implementierung von LIME

Um die Ergebnisse besser verstehen und nachvollziehen zu können, wurde das LSTM Modell noch einmal mit der Integration von LIME implementiert. Die gleichnamige Bibliothek stammt von Marco Tulio Correia Ribeiro (Ribeiro 2019). Dabei handelt es sich um eine prototypische Umsetzung mit Scikit-learn und dem Twitter-Korpora. Allerdings ist anzumerken, dass keine vortrainierten Worteinbettungen eingesetzt wurden sowie keine Stoppwörter gefiltert. Ein interessanter Codeausschnitt folgt, dabei wird die Funktion zum Erklären der Vorhersagen eingesetzt.

```
explainer = LimeTextExplainer(class_names=class_names)

idx = 13674

exp = explainer.explain_instance(texts_test[idx],
      pipeline.predict_proba, num_features=10, top_labels=2)

exp.as_list()

out_as_html = exp.save_to_file("lstm_mc_idx13674.html")
```

In diesem Code wird zunächst die Variable *explainer* angelegt und dem *LimeTextExplainer* zugewiesen, welche die Klassen 0, 1 und 2 als Parameter entgegen nimmt, als *class_ names*. Der Variablen *idx* wird ein Tweet zugewiesen. Als nächstes wird eine Erklärung instanziiert, welche den zuvor festgelegten Tweet erklären soll. Dabei wird mit dem Parameter *num_features* die Anzahl der relevanten Worte festgelegt und mittels *top_ labels* die Anzahl, der darzustellenden Labels. Danach wird diese Erklärung als Liste geladen. Anschließend wird es möglich, diese Erklärung als HTML-Seite abzuspeichern. Die resultierende Seite und Erklärung ist auf der Abbildung 34 zu sehen.

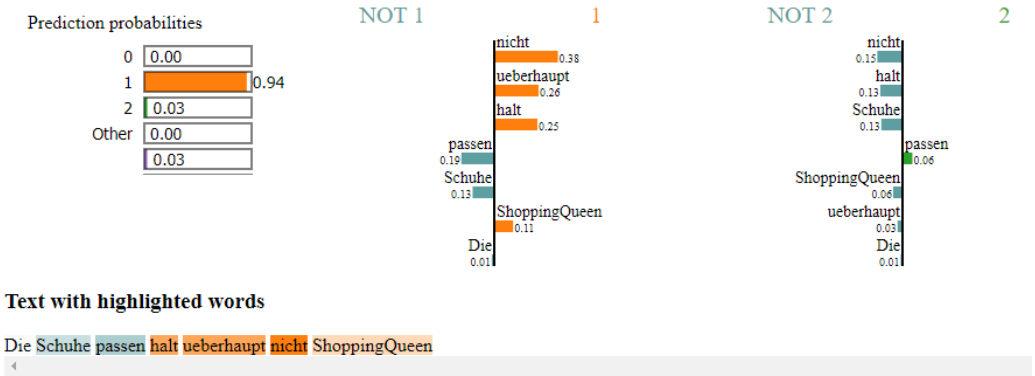


Abbildung 34: Ausgabe der Erklärung mittels LIME. Auf der Abbildung ist zunächst zu erkennen, dass auf der linken Seite die Wahrscheinlichkeiten für die Vorhersagen dargestellt werden. Daneben sind die beiden Klassen dargestellt, die am meisten in diesem Tweet repräsentiert werden. Hier ist beispielsweise zu erkennen, dass die Wörter nicht, ueberhaupt, halt oder auch ShoppingQueen dazu beitragen, dass die Klasse 1 vorhergesagt wurde. Es ist auch zu sehen, dass lediglich das Wort passen für Klasse 2 steht. Darunter wird der Tweet dargestellt. Dieser wird farblich markiert, je dunkler der Farbton ist, desto größer ist die Gewichtung dieses Wortes.

Mit dem nächsten Code wird es möglich, auch in der Konsole Informationen bezüglich der Erklärung auszugeben. Dabei gibt die erste Zeile, den Wert für den festgelegten Tweet aus. In der nächsten Zeile wird die Wahrscheinlichkeit ausgegeben, dass dieser Tweet positiv ist. Weiterhin werden dann die tatsächliche Klasse und die vorhergesagte aufgezeigt. In diesem Beispiel war die Wahrscheinlichkeit für eine positive Klasse 0,27%, die tatsächliche Klasse lautet 0 und die vorhergesagte 1. Es ist festzustellen, dass hier keine Worte

auf die eigentliche Klasse hindeuten.

```
print("idx: ", idx)
print("probability pos: ", pipeline.predict_proba([texts_test[idx
    ]])[0,2])
print("true class: ", y_test[idx])
print("original pred: ", pipeline.predict(y_test[idx]))
```

Mittels dieser Bibliothek und dem LIME-Ansatz selbst, wird es möglich Deep Learning Modelle erklären und dessen Entscheidungen nachvollziehen zu können. Dabei handelt es sich um ein sehr aktuelles Thema, die erklärbare Künstliche Intelligenz (engl. explainable artificial intelligence). Im Bereich NLP könnte dies beispielsweise wie folgt Anwendung finden. Wenn ein Deep Learning Modell trainiert wurde, können dank LIME Erklärungen ausgegeben werden. Diese helfen dem Menschen dabei, die Sinnhaftigkeit der Vorhersagen zu überprüfen. Dabei könnten Wörter entdeckt werden, welche für den Menschen als unwichtig erscheinen, aber große Gewichtung für das Deep Learning Modell besitzt.

6.10 Prototypische Implementierung mit Scikit-Learns Rastersuche

Die Optimierung von Hyperparametern in Deep Learning Modellen stellt eine sehr wichtige und komplexe Aufgabenstellung dar. Komplex, dar es sehr viele Parameter gibt, welche wiederum viele Einstellungen zulassen. Mittels einer Bibliothek von Scikit-Learn wird es möglich die Hyperparameter von Keras Modellen durch die Rastersuche zu optimieren. Dies wurde prototypisch mit dem Twitter-Korpora implementiert. Dabei wurden Experimente mit den Optimierungsalgorithmen, Lernrate und Momentum, Initialisierung der Gewichtungen, Aktivierungsfunktionen und der Anzahl von Neuronen in Schichten. Anhand der Anzahl von Einheiten in den Schichten soll die Implementierung erläutert werden. Dabei wird auf dem folgenden Codeausschnitt dargestellt, wie das Keras Modell aufgebaut wird. Anschließend folgt die Erläuterung.

```
def create_model(neurons=64):
  model = Sequential()
```

```
    e = Embedding(max_words, embedding_dim,
                  weights=[embedding_matrix],
                  input_length=maxlen, trainable=False)
    model.add(e)

    model.add(Conv1D(neurons,
                     kernel_size,
                     activation="relu",
                     strides=1))
    model.add(MaxPooling1D())

    model.add(Flatten())

    model.add(Dense(neurons, activation="relu"))
    model.add(Dropout(0.3))

    model.add(Dense(3, activation="softmax"))

    model.compile(loss="categorical_crossentropy",
                  optimizer=SGD(lr=0.01, momentum=0.6),
                  metrics=["accuracy", f1])

    return model
```

In diesem Code wurde das CNN Modell definiert. Dabei wird bereits der Parameter der Neuronen (*neurons*) angegeben und ein Wert von 64 zugewiesen. Dieser Parameter findet sich in der ersten Faltungs- und dichten Schicht wieder. In dem nächsten Codeausschnitt finden die weiteren statt. Dabei folgt im Anschluss die Erläuterung.

```
neurons = [64, 128, 256, 512]

param_grid = dict(neurons=neurons)

grid = GridSearchCV(estimator=model, param_grid=param_grid,
                    n_jobs=1)

grid_result = grid.fit(x_train, y_train)
```

Hier werden zunächst unter der Variablen *neurons* die unterschiedlichen Werte für die Anzahl der Neuronen angegeben. Diese werden anschließend un-

ter *param_grid* in ein Python Wörterbuch (engl. dictionary) transformiert. Durch *grid* wird die Bibliothek *GridSearchCV* für die Rastersuche vorbereitet, wobei einige Parameter angegeben werden. Dabei handelt es sich um das Deep Learning Model *model*, das Raster bestehend aus Parametern *param_grid* und mittels *n_jobs=1* wird festgelegt, dass das Training nicht parallel sondern nacheinander ausgeführt wird. Mittels *grid_result* und der *fit()* Methode wird das Modell trainiert. Mit einigen zusätzlichen Codezeilen lassen sich die Ergebnisse in der Konsole ausgeben. Die Ausgabe für diesen Durchlauf wird auf Abbildung 35 dargestellt.

```
best: 0.607798 using {'neurons': 128}
0.607315 (0.004493) with: {'neurons': 64}
0.607798 (0.005303) with: {'neurons': 128}
0.605354 (0.004255) with: {'neurons': 256}
0.604078 (0.003053) with: {'neurons': 512}
```

Abbildung 35: Auf dieser Abbildung ist die Ausgabe der Ergebnisse in der Konsole, zu sehen. Dabei wird in der ersten Zeile das beste Ergebnis für die Anzahl der Neuronen geschrieben, welche in diesem Test 128 darstellten. Außerdem ist darunter die mittlere und dann die Standardabweichung der Testergebnisse dargestellt sowie die dazugehörigen Parametereinstellungen.

Mittels der Rastersuche wird es möglich eine Vielzahl von Parametereinstellungen einzugeben, anzuwenden und die Ergebnisse ausgeben zu lassen. Wird dies parallel ausgeführt, kann eine große Anzahl von Parametern und Konfigurationen getestet werden. Dabei lässt sich diese Art der Parameteroptimierung sehr gut strukturieren und effizient sowie effektiv durchführen.

7 Fazit und Ausblick

In dieser Arbeit konnte erfolgreich eine Sentimentanalyse mittels deutscher Twitter-Korpora und Deep Learning durchgeführt werden. Dabei wurden vortrainierte Worteinbettungen eingesetzt. Es folgten zahlreiche explorative Experimente sowie die Visualisierung des Korporas. Weiterhin konnten zwei prototypische Implementierungen umgesetzt werden. Zum einen LIME, um Deep Learning Modelle und Vorhersagen erklären zu können sowie die Rastersuche zur Optimierung von Hyperparametern. Die Ergebnisse sind zwar verbesserbar, aber durchaus annehmbar.

Um die publizierten Ergebnisse zu übertreffen wären noch einige Verbesserungen notwendig. Hierfür wurden bereits einige Beispiele genannt. Die Emoticons können anders transformiert werden und Worteinbettungen können auf größeren Datensätzen trainiert werden. Die Modelle können anhand von LIME verbessert oder die Optimierung mittels der Rastersuche durchgeführt werden. Außerdem können andere Deep Learning Modelle und Architekturen für die Experimente ausgewählt werden.

Weiterhin können die in der Einleitung abgeleiteten Aufgaben und Forschungsgegenstände beantwortet werden:

1. Implementierung der Modelle und Methoden für die Repräsentation von Wörtern.

Die Implementierung der Modelle und Worteinbettungen wurde erfolgreich durchgeführt.

2. Erstellung eines Konzepts für strukturierte Experimente.

Für die explorativen Experimente wurde zuvor ein Konzept erstellt. Hier wurde festgelegt, welche Parameter und Einstellungen untersucht werden sollten.

3. Experimentelle Prüfung verschiedener Parametereinstellungen und deren Auswirkungen auf die Leistung.

Nachdem ein Konzept erstellt wurde, konnte die Prüfung der verschiedenen Parametereinstellungen folgen. Dabei wurden alle Konfigurationen aus-

geführt und die Ergebnisse verglichen.

4. Untersuchung und Visualisierung der Daten.

Die Daten wurden auf unterschiedliche Art und Weise untersucht. Während der Vorverarbeitung wurden Emoticons analysiert und ersetzt. Die Worteinbettungen ließen sich visualisieren und dank der t-SNE Bibliothek war es möglich den Twitter-Korpus mittels verschiedener Vorgehen grafisch aufzubereiten.

5. Vergleich der Experimente und Auswahl der Parameter für die Tests.

Die Resultate der Experimente wurden anhand der F1-Werte verglichen. Dieser wurde als wichtiger Wert eingestuft, da mittels des F1-Wertes die Vergleichbarkeit ermöglicht wurde. Dabei wurden verschiedene Kombinationen der Parameter aus den explorativen Experimenten für den Testdatensatz ausgewählt.

6. Prototypische Implementierung von LIME, einem Ansatz Deep Learning Modelle verstehen zu können.

Es wurde erfolgreich eine prototypische Implementierung mit LIME umgesetzt. Dabei wurde der Twitter-Korpus verwendet.

7. Prototypische Implementierung der Rastersuche zur Optimierung von Hyperparametern.

Auch die prototypische Implementierung der Rastersuche konnte umgesetzt werden. Dabei war es möglich anhand des Twitter-Korpus verschiedene Parameter mit der Rastersuchen auszuführen und zu vergleichen.

Es ist festzustellen, dass trotz einer Verbesserbarkeit der Resultate annehmbare Ergebnisse mit wesentlich kleineren Modellen und Worteinbettungen erreicht werden konnten. Dies lässt darauf schließen, dass die publizierten Ergebnisse durchaus übertroffen werden können. Einige Möglichkeiten wie LIME oder die Rastersuche wurden hier prototypisch eingesetzt und vorgestellt.

Literaturverzeichnis

Aken, van Betty; Risch, Julian; Krestel, Ralf; Löser, Alexander (2018): Challenges for Toxic Comment Classification: An In-Depth Error Analysis, Beuth University of Applied Sciences, Hasso Plattner Institute, University of Potsdam, online abrufbar hier: http://aclweb.org/anthology/W18-5105, letzter Aufruf am 05.03.2019.

Blackmore, Isabelle (2015): Lexical Dispersion Variations for Science Articels, veröffentlicht am 13.01.2015, online abrufbar hier:
https://qm2awesome.wordpress.com/category/lexical-dispersion-plots/, letzter Aufruf am 12.03.2019.

Brownlee, Jason (2016): How to Grid Search Hyperparameters for Deep Learning Models in Pythin With Keras, online abrufbar hier:
https://machinelearningmastery.com/grid-search-hyperparameters
-deep-learning-models-python-keras/, letzter Aufruf am 13.03.2019.

Brownlee, Jason (2019): How to Configure the Learning Rate hyperparameter When Training Deep learning Neural Networks, online abrufbar hier:
https://machinelearningmastery.com/learning-rate-for-deep-learning-neural-networks/, letzter Aufruf am 10.03.2019.

Chollet, Francois (2018): Deep Learning with Python, Manning Pubclications Co., Shelter Island, NY.

Cieliebak, Mark; Deriu, Jan; Egger, Dominic; Uzdilli, Fatih (2017): A Twitter Corpus Benchmark Resources for German Sentiment, International Workshop on Natural Language Processing for Social Media, online abrufbar hier:
https://pdfs.semanticscholar.org/
a050/90ea0393284e83e961f199ea6cd03d13354f.pdf,
letzter Aufruf am 30.11.2018.

CLARIN.SI repository (2019): Twitter sentiment for 1 European languages - German Twitter Sentiment, online abrufbar hier: https://www.clarin.si/
repository/xmlui/handle/11356/1054, letzter Aufruf am 05.03.2019.

DAI-Labor (2019): Verfügbare Datensätze - Annotierter Twitter Sentiment

Datensatz, DAI-Labor - Distributed Artificial Intelligence Laboratory, online abrufbar hier:
http://www.dai-labor.de/kompetenzzentren/irml/datensaetze/, letzter Aufruf am 05.03.2019.

Dean, Jeffrey; Corrado, Greg S.; Monga, Rajat; Chen, Kai; Devin, Matthieu; Le, Quoc V.; Mao, Mark Z.; Ranzato, Marc Aurelio; Senior, Andrew; Tucker, Paul; Yang, Ke; Ng, Andrew Y. (2012): Large Scale Distributed Deep Networks, online abrufbar hier: http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf, letzter Aufruf am 09.03.2019.

Duchi, John; Hazan, Elad; Singer, Yoram (2011): Adaptive Subgradient Methods for Online Learning and Stochastic Optimization, online abrufbar hier: http://jmlr.org/papers/v12/duchi11a.html, letzter Aufruf am 09.03.2019.

Frochte, Jörg (2018): Maschinelles Lernen - Grundlagen und Algorithmen in Python, Carl Hanser Verlag München.

Gibson, Adam & Patterson, Josh (2017): Deep Learning - A Practitioner´s Approach, First Edition, O´Reilly Media, Inc.

Goodfellow, Ian; Begio, Yoshua; Courville, Aaron (2016): Deep Learning, MIT Press.

Greff, Klaus; Srivastava, Rupesh K.; Koutnik, Jan; Steunebrink, Bas R.; Schmidhuber, Jürgen (2017): LSTM: A Search Space Odyssey, Transactions On Neural Networks And Learning Systems, online abrufbar hier: https://arxiv.org/pdf/1503.04069.pdf, letzter Aufruf am 06.01.2019.

Hinton, Geoffrey (2019): Neural Networks for Machine Learning, Lecture 6a, Overview of mini-batch gradient descent, Slide 29, online abrufbar hier: http://www.cs.toronto.edu/ tijmen/csc321/slides/lecture_slides_lec6.pdf, letzter Aufruf am 09.03.2019.

Hochreiter, Sepp & Schmidhuber Jürgen (1997): Long Short-Term Memory, Technische Universität München, online abrufbar hier:
https://www.bioinf.jku.at/publications/older/2604.pdf,
letzter Aufruf am 05.01.2019.

Karpathy (2019): CS231n Convolutional Neural Networks for Visual Recognition, Stanford, online abrufbar hier: http://cs231n.github.io/neural-networks-3/, letzter Aufruf am 10.03.2019.

Kim, Joshua (2017): Understanding how Convolutional Neural Network (CNN) perform text classification with word embeddings, veröffentlicht am 2. Dezember 2017, online abrufbar hier: http://www.joshuakim.io/understanding-how-convolutional-neural-network-cnn-perform-text-classification-with-word-embeddings/, letzter Aufruf am 03.01.2019.

Kim, Yoon (2014): Convolutional Neural Networks for Sentence Classification, New York University, online abrufbar hier:
https://arxiv.org/pdf/1408.5882.pdf, letzter Aufruf am 04.01.2019.

Litzel, Nico (2018): Was ist cuDNN, veröffentlicht am 30.11.2018, online abrufbar hier: https://www.bigdata-insider.de/was-ist-cudnn-a-780686/, letzter Aufruf am 12.03.2019.

Maas, Andre L.; Daly, Raymond E.; Pham, Peter T.; Huang, Dan; Ng, Andrew Y.; Potts, Christopher (2011): Learning Word Vectors for Sentiment Analysis, Association for Computational Linguistics, online abrufbar hier: http://ai.stanford.edu/ amaas/data/sentiment/,
letzter Aufruf am 05.02.2019.

Mikolov, Tomas; Chen, Kai; Corrado, Greg; Dean, Jeffrey (2013): Efficient Estimation of Word Representations in Vector Space, online abrufbar hier: https://arxiv.org/pdf/1301.3781.pdf, letzter Aufruf am 04.03.2019.

Mitchell, Tom M. (1997): Machine Learning, McGraw-Hill Science, Engineering, Math.

Mozetič, Igor; Grčar, Miha; Smailović, Jasmina (2016): Multilingual Twitter Sentiment Classification: The Role of Human Annotators, Department of Knowledge Technologies, PLoS ONE 11(5), online abrufbar hier: https://journals.plos.org/plosone/article/file?id=10.1371/
journal.pone.0155036&type=printable, letzter Aufruf am 05.03.2019.

Müller, Andreas (2015): GermanWordEmbeddings - A toolkit to obtain and preprocess German corpora, train models and evaluate them with generated testsets, online abrufbar hier: https://devmount.github.io/GermanWordEmbeddings/, letzter Aufruf am 04.03.2019.

Narr, Sascha; Hülfenhaus, Michael, Albayrak, Sahin (2012): Language - Independent Twitter Sentiment Analysis, in Knowledge Discovery and Machine learning (KDML), LWA, online abrufbar hier: http://www.dai-labor.de/fileadmin/files/publications/narr-twittersentiment-KDML-LWA-2012.pdf, letzter Aufruf am 05.03.2019.

Nielsen, Michael (2019): Improving the way neural networks learn, Chapter 3, Neural Networks and Deep Learning, online abrufbar hier: http://neuralnetworksanddeeplearning.com/chap3.html, letzter Aufruf am 07.03.2019.

Rajasekharan, Ajit (2017): Answer - What is the main difference between word2vec and fastText?, online abrufbar hier: https://www.quora.com/What-is-the-main-difference-between-word2vec-and-fastText, letzter Aufruf am 04.03.2019.

Raschka, Sebastian & Mirjalili, Vahid (2017): Python Machine Learning: Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow, 2nd Edition, Packt Publishing,

Řehůřek, Radim (2019): gensim: Topic modelling for humans, Python Bibliothek, online abrufbar hier: https://radimrehurek.com/gensim/, letzter Aufruf am 05.03.2019.

Ribeiro, Marco Tulio Correia (2019): Lime - Github, online abrufbar hier: https://github.com/marcotcr/lime/, letzter Aufruf am 10.02.2019.

Ribeiro, Marco Tulio; Singh, Sameer; Guestrin, Carlos (2016a): „Why Should I Trust You?„Explaining the Predictions of Any Classifier, online abrufbar hier: https://arxiv.org/pdf/1602.04938.pdf, letzter Aufruf am 11.03.2019.

Ribeiro, Marco Tulio; Singh, Sameer; Guestrin, Carlos (2016b): Local Interpretable Model-Agnostic Explanations (LIME): An Introduction, online

abrufbar hier: https://www.oreilly.com/learning/introduction-to-local
-interpretable-model-agnostic-explanations-lime,
letzter Aufruf am 11.03.2019.

Rodriguez, Jesus (2017): A Different Way to Think About Overfitting and Underfitting in Machine Learning Part 1: Capacity, Medium, online abrufbar hier: https://medium.com/@jrodthoughts/a-different-way-to-think-about
-overfitting-and-underfitting-in-machine-learning-part-i-capacity
-738aa1bd5498, letzter Aufruf am 06.03.2019.

Ruder, Sebastian (2018): An overview of gradient descent optimization algorithms, online abrufbar hier: http://ruder.io/optimizing-gradient-descent/, letzter Aufruf am 09.03.2019.

SemEval (2019): International Workshop on Semantic Evaluation - SemEval-2019 - Sponsored by SIGLEX and Microsoft, online abrufbar hier:
http://alt.qcri.org/semeval2019/index.php?id=tasks,
letzter Aufruf am 05.03.2019.

Spinningbytes AG (2019a): Word Embeddings - Word Embeddings from Twitter, online abrufbar hier: https://www.spinningbytes.com/resources/
wordembeddings/, letzter Aufruf am 04.03.2019.

Spinningbytes AG (2019b): SB-10k: German Sentiment Corpus, online abrufbar hier: https://www.spinningbytes.com/resources/germansentiment/, letzter Aufruf am 05.03.2019.

Srivastava, Nitish; Hinton, Geoffrey; Krizhevsky, Alex; Sutskever, Ilya; Salakhutdinov, Ruslan (2014): Droptout: A Simple Way to Prevent Neural Networks from Overfitting, online abrufbar hier: http://jmlr.org/papers/
volume15/srivastava14a.old/srivastava14a.pdf, letzter Aufruf am 27.12.2018.

Sutskever, Ilya; Martens, James, Hinto, Geoffrey (2011): Generating Text with Recurrent neural Networks, University of Toronto, Toronto, online abrufbar hier: https://www.cs.utoronto.ca/ ilya/pubs/2011/LANG-RNN.pdf, letzter Aufruf am 03.03.2019.

Tang, Duyu; Wei, Furu; Yang, Nan; Zhou, Ming; Liu, Ting; Qin, Bing (2014):

Learning Sentiment-Specific Word Embedding for Twitter Sentiment Classification, Research Center for Social Computing and Information Retrieval Harbin Institute of Technology, China, Microsoft Research Beijing, China, online aburfbar hier: http://www.aclweb.org/anthology/P14-1146, letzter Aufruf am 03.03.2019.

Taylor, Michael (2017): Neural Networks Math - A Visual Introduction For Beginners, Blue Windmill Media, Printed in Poland by Amazon Fulfillment.

Van der Maaten, Laurens (2019): t-SNE, Github, online abrufbar hier: https://lvdmaaten.github.io/tsne/, letzter Aufruf am 05.03.2019.

Zhang, Xiang; Zhao, Junbo; LeCun, Yann (2015): Character-level Convolutional Networks for Text Classification, Courant Institute of Mathematical Sciences, New York University, NY, online abrufbar hier: http://papers.nips.cc/paper/5782-character-level-convolutional-networks-for-text-classification.pdf, letzter Aufruf am 03.03.2019.

Zhang, Ye & Wallace, Byron C. (2016): A Sensitivity Analysis of (and Practitioners´ Guide to) Convolutional Neural Networks for Sentence Classification, University of Texas at Austin, online abrufbar hier: https://arxiv.org/pdf/1510.03820.pdf, letzter Aufruf am 04.01.2019.

Anhang A - Implementierung LSTM Modell

```
import numpy as np
from keras.preprocessing.text import Tokenizer
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import CuDNNLSTM
from keras.layers import Embedding
from keras.layers import Flatten
from keras import regularizers
from keras.layers import Activation
from keras.preprocessing.sequence import pad_sequences
import matplotlib.pyplot as plt
from keras.utils.np_utils import to_categorical
from keras import backend as K
from nltk.corpus import stopwords
from keras import optimizers
from keras.callbacks import EarlyStopping

import os

def precision(y_true, y_pred):
"""Precision metric.
Only computes a batch-wise average of precision. Computes the
    precision, a
metric for multi-label classification of how many selected items
    are
relevant.
"""
true_positives = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
predicted_positives = K.sum(K.round(K.clip(y_pred, 0, 1)))
precision = true_positives / (predicted_positives + K.epsilon())
return precision

def recall(y_true, y_pred):
"""Recall metric.
Only computes a batch-wise average of recall. Computes the recall
    , a metric
for multi-label classification of how many relevant items are
    selected.
"""
true_positives = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
possible_positives = K.sum(K.round(K.clip(y_true, 0, 1)))
recall = true_positives / (possible_positives + K.epsilon())
return recall
```

```
def f1(y_true, y_pred):
"""Computes the F1 Score
Only computes a batch-wise average of recall. Computes the recall
    , a metric
for multi-label classification of how many relevant items are
    selected.
"""
p = precision(y_true, y_pred)
r = recall(y_true, y_pred)
return (2 * p * r) / (p + r + K.epsilon())

# prozedur für training
tweets_dir = "tweets"
train_dir = os.path.join(tweets_dir, "train")

labels = []
texts = []

for label_type in ["neg", "neu", "pos"]:
dir_name = os.path.join(train_dir, label_type)
for fname in os.listdir(dir_name):
if fname[-4:] == ".txt":
f = open(os.path.join(dir_name, fname), encoding="UTF-8")
texts.append(f.read())
f.close()
if label_type == "neg":
labels.append(0)
elif label_type == "neu":
labels.append(1)
elif label_type == "pos":
labels.append(2)

stopwords = set(stopwords.words("german"))
filtered_texts = []

for w in texts:
if w not in stopwords:
filtered_texts.append(w)

maxlen = 30
max_words = 60000

# training
tokenizer = Tokenizer(num_words=max_words)
```

```
tokenizer.fit_on_texts(filtered_texts)
sequences = tokenizer.texts_to_sequences(filtered_texts)

word_index = tokenizer.word_index
print("Found %s unique tokens: " % len(word_index))

# training
data = pad_sequences(sequences, maxlen=maxlen)

# training
labels = np.asarray(labels)
print("shape of data tensor", data.shape)
print("shape of label tensor", labels.shape)

# training
indices = np.arange(data.shape[0])
np.random.shuffle(indices)

# training
x_train = data[indices]
y_train = labels[indices]

print("training x_train, y_train")
print(x_train)
print(y_train)

# word2vec
embedding_index = {}
with open("german_model.txt", encoding="UTF-8") as f:
for line in f:
values = line.split()
word = values[0]
coefs = np.asarray(values[1:], dtype="float32")
embedding_index[word] = coefs
f.close()
print("found %s word vectors in word2vec: " % len(embedding_index
    ))

embedding_dim = 300

embedding_matrix = np.zeros((max_words, embedding_dim))
for word, i in word_index.items():
embedding_vector = embedding_index.get(word)
if i < max_words:
if embedding_vector is not None:
```

```
embedding_matrix[i] = embedding_vector

model = Sequential()
e = Embedding(max_words, embedding_dim,
weights=[embedding_matrix],
input_length=maxlen, trainable=False)
model.add(e)

model.add(CuDNNLSTM(256, return_sequences=True))
model.add(Activation("relu"))

model.add(CuDNNLSTM(256, return_sequences=True))
model.add(Activation("relu"))

model.add(Dense(256, activation="relu", kernel_regularizer=
    regularizers.l2(0.0001)))
#model.add(Dropout(0.3))
model.add(Dense(256, activation="relu", kernel_regularizer=
    regularizers.l2(0.0001)))

model.add(Flatten())

model.add(Dense(3, activation="softmax"))

#plot_model(model, to_file="lstm.png", show_shapes=True,
    show_layer_names=True)

y_train = to_categorical(y_train, num_classes=3)

print(y_train)

sgd = optimizers.SGD(lr=0.001, decay=1e-6, momentum=0.9, nesterov
    =True)

rmsprop = optimizers.RMSprop(lr=0.0001)
nadam = optimizers.Nadam(lr=0.00001)
adagrad = optimizers.Adagrad(lr=0.001)

model.compile(loss="categorical_crossentropy",
optimizer=sgd,
#optimizer=sgd,
metrics=["accuracy", f1])

callbacks = [EarlyStopping(monitor="val_f1", patience=30, mode="
    max", restore_best_weights=True)]
```

```
history = model.fit(x_train, y_train,
batch_size=64,
epochs=300,
validation_split=0.2,
callbacks=callbacks)

# score, acc = model.evaluate(x_train, y_train,
#                             batch_size=64)

model.summary()

print("acc: ", history.history["acc"])
print("val_acc: ", history.history["val_acc"])
print("loss: ", history.history["loss"])
print("val_loss: ", history.history["val_loss"])
print("f1: ", history.history["f1"])
print("val_f1: ", history.history["val_f1"])

plt.plot(history.history["acc"])
plt.plot(history.history["val_acc"])
plt.title("model accuracy")
plt.ylabel("accuracy")
plt.xlabel("epoch")
plt.legend(["train", "val"], loc="upper left")
plt.show()

plt.plot(history.history["loss"])
plt.plot(history.history["val_loss"])
plt.title("model vs validation loss")
plt.ylabel("loss")
plt.xlabel("epoch")
plt.legend(["train", "val"], loc="upper left")
plt.show()

plt.plot(history.history["f1"])
plt.plot(history.history["val_f1"])
plt.title("model vs validation f1")
plt.ylabel("f1")
plt.xlabel("epoch")
plt.legend(["train", "val"], loc="upper left")
plt.show()

# print("score: ", score)
# print("acc: ", acc)
```

```
# prozedur für test
test_dir = os.path.join(tweets_dir, "test")
test_labels = []
test_texts = []

for test_label_type in ["neg", "neu", "pos"]:
dir_name = os.path.join(test_dir, test_label_type)
for vname in os.listdir(dir_name):
if vname[-4:] == ".txt":
g = open(os.path.join(dir_name, vname), encoding="UTF-8")
test_texts.append(g.read())
g.close()
if test_label_type == "neg":
test_labels.append(0)
elif test_label_type == "neu":
test_labels.append(1)
elif test_label_type == "pos":
test_labels.append(2)

filtered_test_texts = []

for u in test_texts:
if u not in stopwords:
filtered_test_texts.append(u)

#test
tokenizer.fit_on_texts(filtered_test_texts)
test_sequences = tokenizer.texts_to_sequences(filtered_test_texts
    )

# test
test_data = pad_sequences(test_sequences, maxlen=maxlen)

# test
test_labels = np.asarray(test_labels)

# test
test_indices = np.arange(test_data.shape[0])
np.random.shuffle(test_indices)

# test
x_test = test_data[test_indices]
y_test = test_labels[test_indices]
```

```
print("test x_test, y_test")
print(x_test)
print(y_test)

y_test = to_categorical(y_test, num_classes=3)
print(y_test)

model.predict(x_test, batch_size=64)

print(model.metrics_names)

loss, acc, f1 = model.evaluate(x_test, y_test)

print("test_loss: ", loss)
print("test_acc: ", acc)
print("test_f1: ", f1)
```

Anhang B - Prototypische Implementierung LIME

```
from sklearn.model_selection import train_test_split
import pandas as pd
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from sklearn.pipeline import TransformerMixin
from sklearn.base import BaseEstimator
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Embedding, Bidirectional, LSTM,
    CuDNNLSTM, Dropout, Flatten
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.pipeline import make_pipeline
from lime.lime_text import LimeTextExplainer
import sklearn

df = pd.read_csv("corpus_6.tsv", sep=",")

texts_train, texts_test, y_train, y_test = train_test_split(df["
    tweet"].values.astype(str), df['sentiment'].values,
    train_size=0.8, random_state=42)

class_names = ["neg", "neu", "pos"]

print("texts_train: ", texts_train)
print("texts_test: ", texts_test)
print("y_train: ", y_train)
print("y_test: ", y_test)

vocab_size = 60000
maxlen = 30

class TextsToSequences(Tokenizer, BaseEstimator, TransformerMixin
    ):

def __init__(self, **kwargs):
super().__init__(**kwargs)

def fit(self, texts, texts_test, y=None):
self.fit_on_texts(texts)
return self
```

```
def transform(self, texts, y=None):
return np.array(self.texts_to_sequences(texts))


sequencer = TextsToSequences(num_words=vocab_size)

class Padder(BaseEstimator, TransformerMixin):

def __init__(self, maxlen=30):
self.maxlen = maxlen
self.max_index = None

def fit(self, X, y=None):
self.max_index = pad_sequences(X, maxlen=self.maxlen).max()
return self

def transform(self, X, y=None):
X = pad_sequences(X, maxlen=self.maxlen)
X[X > self.max_index] = 0
return X


padder = Padder(maxlen)

batch_size = 128
max_features = vocab_size + 1

def create_model(max_features):
model = Sequential()
model.add(Embedding(max_features, 128))
model.add(Bidirectional(CuDNNLSTM(128)))
model.add(Dropout(0.3))
model.add(Dense(128))
model.add(Dense(6, activation='softmax'))
model.compile('adam', 'categorical_crossentropy', metrics=['
    accuracy'])
return model

sklearn_lstm = KerasClassifier(build_fn=create_model, epochs=5,
    batch_size=batch_size,
max_features=max_features, verbose=1)

pipeline = make_pipeline(sequencer, padder, sklearn_lstm)

print(y_train.shape)
```

```
pipeline.fit(texts_train, y_train)

print('Computing predictions on test set...')
y_preds = pipeline.predict(texts_test)

print("f1 score: ", sklearn.metrics.f1_score(y_test, y_preds,
    average="weighted"))

explainer = LimeTextExplainer(class_names=class_names)

idx = 13674

exp = explainer.explain_instance(texts_test[idx], pipeline.
    predict_proba, num_features=10, top_labels=2)

exp.as_list()

out_as_html = exp.save_to_file("lstm_mc_idx13674.html")

text_sample = texts_test[idx]

print("idx: ", idx)
print("probability pos: ", pipeline.predict_proba([texts_test[idx
    ]])[0,2])
print("true class: ", y_test[idx])
print("original pred: ", pipeline.predict(y_test[idx]))

print('Sample {}: last 10 words (only part used by the model)'.
    format(idx))
print(" ".join(text_sample.split()[-10:]))
```

Anhang C - Prototypische Implementierung Rastersuche

```
import numpy as np
from sklearn.model_selection import GridSearchCV
from keras.wrappers.scikit_learn import KerasClassifier
from keras import backend as K
from keras.preprocessing.text import Tokenizer
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Embedding
from keras.layers import Dropout
from keras.layers import Flatten
from keras.preprocessing.sequence import pad_sequences
from keras.utils import to_categorical
from keras.optimizers import SGD
from keras.layers import Conv1D, GlobalMaxPooling1D, MaxPooling1D
    , GaussianNoise
from nltk.corpus import stopwords
import os
from keras.callbacks import EarlyStopping, ModelCheckpoint

def precision(y_true, y_pred):
"""Precision metric.
Only computes a batch-wise average of precision. Computes the
    precision, a
metric for multi-label classification of how many selected items
    are
relevant.
"""
true_positives = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
predicted_positives = K.sum(K.round(K.clip(y_pred, 0, 1)))
precision = true_positives / (predicted_positives + K.epsilon())
return precision

def recall(y_true, y_pred):
"""Recall metric.
Only computes a batch-wise average of recall. Computes the recall
    , a metric
for multi-label classification of how many relevant items are
    selected.
"""
true_positives = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
possible_positives = K.sum(K.round(K.clip(y_true, 0, 1)))
```

```
recall = true_positives / (possible_positives + K.epsilon())
return recall

def f1(y_true, y_pred):
"""Computes the F1 Score
Only computes a batch-wise average of recall. Computes the recall
    , a metric
for multi-label classification of how many relevant items are
    selected.
"""
p = precision(y_true, y_pred)
r = recall(y_true, y_pred)
return (2 * p * r) / (p + r + K.epsilon())


tweets_dir = "tweets"
train_dir = os.path.join(tweets_dir, "train")

labels = []
texts = []

for label_type in ["neg", "neu", "pos"]:
dir_name = os.path.join(train_dir, label_type)
for fname in os.listdir(dir_name):
if fname[-4:] == ".txt":
f = open(os.path.join(dir_name, fname), encoding="UTF-8")
texts.append(f.read())
f.close()
if label_type == "neg":
labels.append(0)
elif label_type == "neu":
labels.append(1)
elif label_type == "pos":
labels.append(2)

stopwords = set(stopwords.words("german"))
filtered_texts = []

for w in texts:
if w not in stopwords:
filtered_texts.append(w)

maxlen = 30
max_words = 60000
kernel_size = 3
```

```
hidden_dims = 256
filters = 256

tokenizer = Tokenizer(num_words=max_words)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)

word_index = tokenizer.word_index
print("Found %s unique tokens: " % len(word_index))

data = pad_sequences(sequences, maxlen=maxlen)

labels = np.asarray(labels)
print("shape of data tensor", data.shape)
print("shape of label tensor", labels.shape)

indices = np.arange(data.shape[0])
np.random.shuffle(indices)

x_train = data[indices]
y_train = labels[indices]

embedding_index = {}
with open("german_model.txt", encoding="UTF-8") as f:
for line in f:
values = line.split()
word = values[0]
coefs = np.asarray(values[1:], dtype="float32")
embedding_index[word] = coefs
f.close()
print("found %s word vectors in word2vec: " % len(embedding_index
    ))

print("Found %s vectors" % len(embedding_index))

embedding_dim = 300

embedding_matrix = np.zeros((max_words, embedding_dim))
for word, i in word_index.items():
embedding_vector = embedding_index.get(word)
if i < max_words:
if embedding_vector is not None:
embedding_matrix[i] = embedding_vector

def create_model(neurons=64):
```

```
model = Sequential()
e = Embedding(max_words, embedding_dim, weights=[embedding_matrix
    ], input_length=maxlen, trainable=False)
model.add(e)

model.add(Conv1D(neurons,
kernel_size,
activation="relu",
strides=1))
model.add(MaxPooling1D())

model.add(Flatten())

model.add(Dense(neurons, activation="relu"))
model.add(Dropout(0.3))

model.add(Dense(3, activation="softmax"))

model.compile(loss="categorical_crossentropy",
optimizer=SGD(lr=0.01, momentum=0.6),
metrics=["accuracy", f1])

return model


y_train = to_categorical(y_train, num_classes=3)

callbacks = [EarlyStopping(monitor="val_f1", patience=30, mode="
    max", restore_best_weights=True)]

model = KerasClassifier(build_fn=create_model, epochs=10,
    batch_size=64)

neurons = [64, 128, 256, 512]

param_grid = dict(neurons=neurons)

grid = GridSearchCV(estimator=model, param_grid=param_grid,
    n_jobs=1)

grid_result = grid.fit(x_train, y_train)

print("best: %f using %s" %(grid_result.best_score_, grid_result.
    best_params_))
means = grid_result.cv_results_["mean_test_score"]
```

```
stds = grid_result.cv_results_["std_test_score"]
params = grid_result.cv_results_["params"]

for mean, stdev, param in zip(means, stds, params):
print("%f (%f) with: %r" %(mean, stdev, param))
```

Ehrenwörtliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit mit dem Titel *Sentimentanalyse deutscher Twitter-Korpora mittels Deep Learning* selbstständig verfasst und keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt wurde.

Brandenburg,	15.03.2019,	
Ort,	Datum,	Unterschrift