

Einsatz von Sprachmodellen zu Stance Detection und Sentiment Analysis

Lina Suttrup
1120810

Betreuung:
Prof. Dr. Oswald

Institut für Verteilte Intelligente Systeme
Universität der Bundeswehr München
Fakultät für Elektrotechnik und Technische Informatik

Inhaltsverzeichnis

1	Einleitung	8
1.1	Natural Language Processing	8
1.2	Sentiment Analysis	9
1.3	Stance detection	9
1.4	Zielsetzung und Vorgehen	10
2	Sprachmodelle	11
2.1	Problemstellung	11
2.2	Sprachmodelle auf Basis von LSTMs	12
2.2.1	ELMo	13
2.2.2	ULMFit	14
2.3	Sprachmodelle auf Basis von Transformern	15
2.3.1	GPT	17
2.3.2	BERT	20
2.3.3	RoBERTa	21
2.3.4	XLNet	23

3	Einsatz von Sprachmodellen	25
3.1	Versuchsaufbau	25
3.1.1	Verwendete Bibliotheken	26
3.2	Die Daten	26
3.2.1	Apple Sentiment	27
3.2.2	US Airline Sentiment	29
3.2.3	T4SA	30
3.2.4	Fake News Challenge	30
3.2.5	SemEval	31
3.3	Sentiment Analysis	32
3.3.1	Ausgangsbasis	32
3.3.2	ELMo	33
3.3.3	ULMFit	35
3.3.4	RoBERTa	38
3.3.5	XLNet	41
3.3.6	GPT	43
3.4	Stance Detection	44
3.4.1	ULMFit	44
3.4.2	RoBERTa	45
3.4.3	XLNet	46
4	Auswertung	48
4.1	Ergebnisse	48
4.2	Ausblick	49

Bestätigung

Hiermit versichere ich, dass die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt wurden.

Ferner habe ich vom Merkblatt über die Verwendung von Abschlussarbeiten Kenntnis genommen und räume das einfache Nutzungsrecht an meiner Arbeit der Universität der Bundeswehr München ein.

Neubiberg, den 10.09.2020

Lina Suttrup

1. Einleitung

Was wäre, wenn Computer von ihren Erfahrungen lernen könnten, so wie Menschen es tun? Eine Frage, aus der viele Forschungsfelder entsprungen sind. **Künstliche Intelligenz** ist ein großes Themengebiet, aber ohne **Machine Learning** sind Computer nicht „schlauer“ als die Programme, die sie ausführen - und die von Menschen programmiert werden. Zwar kann ein Computer ein Problem mit vorgegebenen Lösungsweg meist weit schneller lösen als der Mensch selber, aber wie kommt er auf diesen Lösungsweg? **Machine Learning** und **Deep Learning** konzentrieren sich auf genau diese Problemstellung.

Seit der Entwicklung erster **Machine Learning** Algorithmen hat dieses Feld immense Fortschritte gemacht, besonders in den letzten Jahren, in denen auch **Deep Learning** durch die Entwicklung immer schnellerer Prozessoren sowie der Möglichkeiten der modernen GPUs zur Parallelisierung von Aufgaben immer weiter in den Vordergrund rücken konnte. Machine Learning und **Deep Learning** ist heute schon weit verbreitet und wird in den verschiedensten Feldern eingesetzt - von E-Mail Spam Klassifizierung bis hin zur Erkennung von Tumoren auf Röntgenbildern, es bleibt aber weiterhin ein großes Forschungsfeld.

1.1 Natural Language Processing

Ein wichtiger Teil der Künstlichen Intelligenz ist die **Computerlinguistik**, oder auch **Natural Language Processing (NLP)**. Menschliche Sprache ist für Maschinen nur schwer verständlich, da sie im Gegensatz zu Maschinensprachen keiner genauen Logik folgen: Sie ist im Grunde unvorhersehbar und führt selbst bei Menschen zu Missverständnissen. Trotzdem gibt es heutzutage viele Anwendungen von **NLP** im Alltag, und Beispiele wie die heutigen Sprachassistenten zeigen, wie weit dieses Feld in einer sehr kurzen Zeit voranschreiten konnte.

Aufgrund der Komplexität des Problems gibt es meherer Herangehensweisen, um natürliche

Sprache für Maschinen verständlich zu machen. Ihnen allen ist gemein, dass sie Sprache durch Zahlen repräsentieren.

Es gibt Ansätze, die Wörter als einzelne Einheiten nehmen, und diesen einfach eine Zahl zuordnen. Dies hat den Vorteil, dass es leicht umzusetzen ist und auch keine hohe Anzahl an Trainingsdaten braucht. Das Problem daran ist, dass so sehr viel Bedeutung verloren gehen kann - in natürlicher Sprache sind häufig die umgebenden Wörter genauso wichtig wie das betrachtete Wort. Kontext und Wortzusammenhänge gehen durch diesen Ansatz verloren.

Eine weitere bekannte Technik ist der Einsatz von **Wortvektoren**, die durch ihre hohe Dimensionalität viele Beziehungen zwischen Wörtern bewahren können. Diese Repräsentation ist vielversprechend, aber auch hier kann nicht immer der Kontext gewahrt werden, da einem Wort genau ein Vektor zugeordnet wird.

Eine Verbesserung, nicht nur in diesem Aspekt, bringen **Sprachmodelle**. Sie können vielfältig eingesetzt werden, haben einen großen Wortschatz in verschiedenen Sprachen und sind vortrainiert - sie müssen entsprechend nur der Aufgabe angepasst werden, was einer enormen Zeiterparnis entspricht. Durch Feinabstimmung der verschiedenen Parameter kann bei Sprachmodellen auch viel erreicht werden [17].

1.2 Sentiment Analysis

Sentiment Analysis ist die Einordnung von Text in Kategorien von Emotionen, wobei es verschiedene Skalen gibt. Die einfachste ist hierbei die Einordnung in „Neutral“, „Positiv“ und „Negativ“, aber es gibt auch Zuordnungen zu Werten von -1 bis 1 oder anderen Zahlenwerten. Diese Werte werden nicht immer mit einheitlicher Bedeutung genutzt.

1.3 Stance detection

Bei **Stance Detection** geht es darum, herauszufinden, wie die Haltung des Autors eines Textes zu einer bestimmten These oder einem allgemeinen Begriff ist. Beste Ergebnisse wurden durch Unterteilung der Aufgabe in zwei Schritte erzielt; Zunächst wird **Topic Modeling** genutzt, um herauszufinden, ob der Text überhaupt mit der Stance in Verbindung gebracht werden kann. Daraufhin wird, falls dies der Fall ist, auf die Stance bezogene **Sentiment Analysis** genutzt, um die Haltung des Autors zu ermitteln.

1.4 Zielsetzung und Vorgehen

Ziel dieser Arbeit ist es, zu erforschen wie gut sich verschiedene **Sprachmodelle** zum Einsatz in **Sentiment Analysis** und **Stance Detection** eignen. Dazu werden die verschiedenen Modelle vorgestellt und analysiert, welche für den Einsatz in Frage kommen, da die Modelle durch unterschiedlichen Aufbau oder auch verfügbare Datensätze nicht alle gleich gute Herangehensweisen an die Problemstellung darstellen. Für die Versuche werden mehrere **Corpora** verwendet, von denen die meisten öffentlich zugänglich sind. Die **Corpora** bestehen aus Tweets, die sich aufgrund der limitierten Zeichenlänge und der Popularität der Plattform *Twitter* gut für die Auswertung in beiden Aufgaben eignen oder aus Artikeln, die für **Stance detection** eine gute Grundlage bieten. Die Ergebnisse werden ausgewertet und miteinander verglichen.

2. Sprachmodelle

Bis zur Einführung von Sprachmodellen gab es für NLP in Machine Learning ein großes Problem: Textdaten, wie auch Sprache, sind inherent sequentiell. Im Gegensatz zu Bildern, bei denen alle Daten auf einmal verarbeitet werden können, konnte die Textverarbeitung die Beschleunigung durch den Einsatz von GPUs nicht nutzen, da viele Aufgaben nicht parallelisiert werden konnten [3].

Weiterhin gab es keine Möglichkeit, ein schon vortrainiertes Modell für andere Aufgaben weiter zu verwenden, da das Training für die verschiedenen Aufgaben und auch Domänen sehr spezifisch angegangen werden muss. Diese Faktoren führten zu einem deutlichen Mehraufwand von NLP z.B. gegenüber Bildverarbeitung.

Das Ziel von Sprachmodellen ist es, parallele Verarbeitung der Daten möglich zu machen sowie, im Gegensatz zu vortrainierten **Wortvektor Embeddings** wie **GLoVe** den Kontext der einzelnen Wörter im Satz zu wahren. In den Embeddings wird jedem Wort ein Vektor zugewiesen, sodass ähnliche Wörter nah beieinander liegen und die Beziehung zwischen Wörtern erhalten bleibt. Dadurch können interessante Rechnungen aufgestellt werden:

London - England + Spanien = Madrid

Mit ganzen Textsequenzen können aber auch diese Vektor Embeddings aufgrund der fehlenden Kontextinformationen nur bedingt gut umgehen.

2.1 Problemstellung

Wie schon erwähnt muss Sprache sequentiell verarbeitet werden. Dazu werden meist **Gated RNNs** und **Long Short Term Memory (LSTM)** - Netzwerke verwendet [3]. Durch einige

Techniken kann man mit diesen Netzen auch gute Laufzeiten erreichen, aber es besteht ein weiteres Problem: Der Kontext der Wörter geht schnell verloren, da diese Netze einem Wort meist auch nur eine Bedeutung zuordnen können. Der Umstand, dass diese Netze für jede Anwendung spezifisch trainiert werden müssen, macht ihren Einsatz sehr umständlich.

Ein ebenfalls sehr großes Problem in NLP ist die Tatsache, dass **Überwachtes Lernen** mit erheblichem manuellen Aufwand - dem Annotieren und Labeln der Daten - verbunden ist. Es gibt zwar sehr große Mengen an Daten, die durch das Internet verfügbar sind (z.B. alle *Wikipedia-Artikel*), diese eignen sich aber nur für unüberwachte Lernmethoden, da sie nicht gelabelt sind. Die Menge an gelabelten Daten hingegen ist verschwindend gering im Vergleich zu allen frei verfügbaren. Dies legt die Entwicklung eines Systems nahe, das mit **teil-** oder **fernüberwachtem Lernen** auskommt. Heutige Sprachmodelle versuchen hier anzusetzen, indem sie in einem ersten Schritt unüberwacht mit Korpusen unterschiedlicher Größe die Repräsentation einer Sprache lernen, um dann im zweiten Schritt an die zu erfüllende Aufgabe angepasst zu werden. Besonders interessant hierbei sind neueste Entwicklungen [11], durch die ein Modell sogar mit nur einem Beispiel der zu erfüllenden Aufgabe gute Ergebnisse erzielen kann. Dieses Konzept nennt sich **Transfer Learning**, durch Anwendung in der Bildverarbeitung konnten dadurch große Verbesserungen in Zeit und Genauigkeit erreicht werden [1], da ein vortrainiertes Modell nur noch mit Feinabstimmungen für die spätere Aufgabe vorbereitet werden muss.

Das grundsätzliche Ziel von Sprachmodellen ist die Ausgabe des Wortes, das am wahrscheinlichsten auf eine Eingabesequenz folgt. Direkt anwendbar ist dies z.B. auf Textgeneration und Übersetzungen, während für andere Aufgaben leichte Änderungen vorgenommen werden müssen um z.B. für **Sentiment Analysis** eine Zahl zu erhalten, die dem in der Eingabe erkannten Sentiment entspricht. Sieht man diese Zahl als das vorrauszusagende Wort an können diese Änderungen recht leicht umgesetzt werden.

2.2 Sprachmodelle auf Basis von LSTMs

LSTMs können aufgrund ihrer Architektur gut mit kurzen Sätzen umgehen, aber je länger die Wortsequenz, desto schlechter kann sie verarbeitet werden [9].

Durch die sequentiellen Verarbeitung steigt die Verarbeitungszeit mit der Textlänge an, wodurch **LSTMs** generell langsamer sind als **RNNs**, da die Neuronen komplexer sind.

Trotz dieser Nachteile können mit dieser Architektur für einige Anwendungen geeignete Sprachmodelle erstellt werden.

2.2.1 ELMo

ELMo baut auf der Idee der Wortvektoren oder auch **Word Embeddings** auf. Die von **ELMo** erlernten Wort-Embeddings können in den eigenen, für Aufgaben spezifisch entwickelten Netzen eingebunden werden, um gegenüber sehr festgelegter Vektoren wie **GloVe** Ergebnisse zu erzielen, die zum Zeitpunkt der Entwicklung dieses Modells zu den besten gehörten [5].

Architektur

Die zweischichtige Architektur besteht aus jeweils einem **Feed-Forward** Netz, dessen Layer

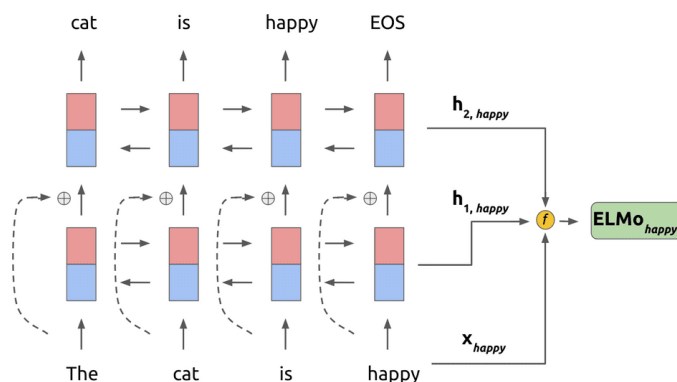


Abbildung 2.1: Aufbau des ELMo-Modells [6]

in Abbildung 2.1 in rot eingezeichnet sind und einem **Feed-Backward** Netz, das in blau dargestellt ist. Damit ist **ELMo** ein sogenanntes **bidirektionales Sprachmodell**. Die Schichten bestehen aus jeweils 4096 LSTM-Zellen und zwischen den zwei Schichten gibt es eine Verbindung zur Eingabeschicht.

Input und Output

Wie in der Abbildung 2.1 zu sehen, gibt es drei Repräsentationen für ein Wort, eine in jeder Schicht. Zusammengenommen ergeben sie das **ELMo Embedding**. Das erste Embedding der Wörter entsteht schon vor dem eigentlichen Sprachmodell, wie Abbildung 2.2 zeigt. Die Wörter werden in Character-Tokens aufgeteilt und in einem Convolutional Network mit 2048 Filtern und Max Pooling, gefolgt von einem zweischichtigen **Highway Netz** verarbeitet, bevor sie in das LSTM-Netz gegeben werden.

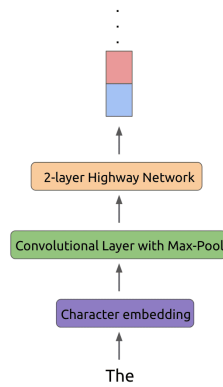


Abbildung 2.2: Vorverarbeitung der Wörter [6]

Trainingdaten

Das Modell wurde mit dem 1B Word Benchmark trainiert, das einen Korpus von 0,8 Milliarden Wörtern besteht und ist damit ein insgesamt sehr großes Modell.

2.2.2 ULMFit

Universal Language Model Fine-Tuning oder auch **ULMFit** ist ein Vorschlag eines von der Architektur recht einfach gehaltenen Sprachmodells, das Transfer Learning in NLP zugänglich machen wollte. Der Ansatz ist besonders darauf fokussiert, ein mit allgemeinen Daten trainiertes Modell auf eine spezifische Aufgabe mit einem eigenen Kontext (z.B. Medizin, IT etc.) fein abzustimmen [1]. Dadurch wird die Größe des gelabelten Datensatzes und die Anzahl der domänenspezifischen Dokumente, die für ein von Grund auf neu implementiertes Modell benötigt würden stark gemindert.

Architektur

Das Modell besteht aus drei LSTM-Schichten, die, um Overfitting auf den Daten zu vermeiden, mit einer sehr hohen Dropout-Rate versehen sind. Pro Schicht hat **ULMFit** 1150 versteckte Neuronen.

Bei der Arbeit mit **ULMFit** werden generell drei Schritte ausgeführt [1]:

1. Trainieren des Modells mit einem allgemeinen, großen Datensatz
2. Feinabstimmung des Sprachmodells für die NLP-Aufgabe

3. Feinabstimmung des Klassifikators für die Aufgabe

Schritt 1. muss bei Einsatz des Modells nicht ausgeführt werden, da diese Version des Modells verfügbar ist. Durch diesen Schritt lernt das Modell, in den verschiedenen Schichten die Merkmale der (in diesem Fall englischen) Sprache abzubilden. Bei Schritt 2. wird mit **discriminative Fine-Tuning** und einer abgeschrägten, dreieckigen Lernrate gearbeitet, um Merkmale der in der Aufgabe genutzten Sprache zu lernen. Diese Lernrate wird angewandt, da die verschiedenen Schichten unterschiedliche Merkmale lernen sollen und sich damit die letzten Schichten nicht mehr viel verändern. Schritt 3. nutzt **gradual unfreezing** sowie die beiden in Schritt 2. genutzten Techniken für die Feinabstimmung. Hierfür werden an das Sprachmodell zwei lineare Schichten mit ReLu Activation, dropout und Batchnormalisierung und eine Softmax-Schicht für die Ausgabe.

Input und Output

Die Input-Sequenzen werden vor der Verarbeitung in den LSTM-Schichten in 400 dimensionale **Word Embeddings** umgewandelt.

Wie bei Sprachmodellen üblich ist der Output von **ULMFit** eine Wahrscheinlichkeitsverteilung über das Vokabular, um das mit höchster Wahrscheinlichkeit als nächstes im Satz kommende Wort auszugeben.

Trainingsdaten

ULMFit wurde mit dem Datensatz **Wikitext-103**, der aus 103 Millionen Wörtern in Wikipedia Artikeln besteht. Die Feinabstimmung liefert schon mit 100 Datensätzen gute Ergebnisse [1].

2.3 Sprachmodelle auf Basis von Transformern

Transformer sind die Basis der heute meistgenutzten Sprachmodelle [7][9]. Sie wurden 2017 entwickelt, um das Problem der sequentiellen Verarbeitung in Maschinellem Übersetzen zu lösen: Als Eingabe kann ein **Transformer** eine ganze Textsequenz verarbeiten. Die wichtigste Komponente in diesem System ist **Attention**: Dadurch kann einem Wort sein Kontext innerhalb der betrachteten Sequenz zugeordnet werden [3].

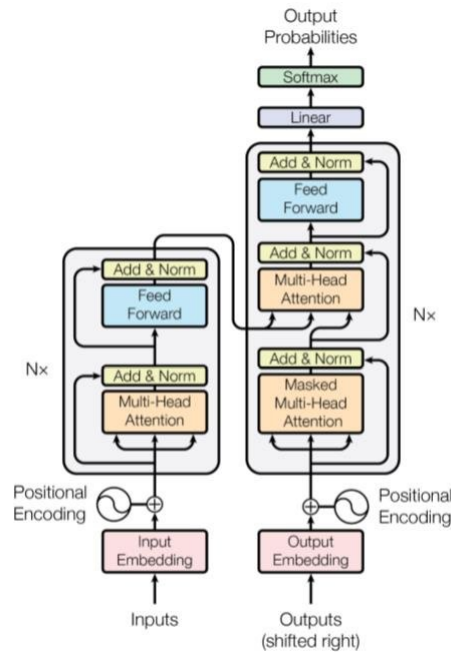


Abbildung 2.3: Aufbau eines Transformers [3]

Encoder

Wie in Abbildung 2.3 zu sehen, wird die Eingabe zunächst in Wortvektoren für jedes Eingabewort umgewandelt, die sich aus der Addition des Vektors im **Embedding Space** mit einem **Positionsvektor (Positional Encoding)**, der die Position des Wortes im Satz enthält, ergibt. Der Positionsvektor wird durch Sinus- und Kosinusfunktionen bestimmt und ist wichtig, da Sequenzen parallel verarbeitet werden und die Position dadurch verloren geht. Dieser Eingabevektor wird an einen **Encoder** weitergegeben, der eine **Attention-Matrix** für die Eingabesequenz erstellt. Sie besteht aus einem Vektor der Länge der Textsequenz für jedes Wort. Diese Matrix enthält Informationen darüber, wie relevant die einzelnen Wörter der Sequenz füreinander sind: In dem Satz

Der braune Hund

bezieht sich z.B. das Wort „Der“ sowie das Adjektiv „braune“ auf das Wort „Hund“. In dem Attention-Vektor wären entsprechend die Werte für „Hund“ in den Vektoren der beiden anderen Wörter höher. **Attention** ist der Schlüsselpunkt für die Funktion der Transformer: Dadurch erhalten die Wortvektoren ihre eigentliche Kontextinformation. Um die Werte zu optimieren

werden hier die Mittelwerte von acht Vektoren pro Wort ermittelt.

Alle bisher errechneten Vektoren werden danach aufaddiert, normalisiert und jeder Vektor wird in ein Feed-Forward-Netz gegeben. Hierbei kann parallelisiert werden, da die einzelnen Vektoren voneinander unabhängig sind. Die Ausgabe des **Encoders** sind die mit Kontext und Attention encodierten Wortvektoren.

Decoder

Der **Decoder** ist ähnlich aufgebaut. Er nimmt einen zweiten Input an - der eigentliche Output bei Supervised Learning - der, wie Abbildung 2.3 zeigt, zunächst auch in einen Eingabevektor mit Positionsinformation und Attention-Matrix umgewandelt wird. Im nächsten Schritt werden Input und Output gemeinsam in ein weiteres Netz zur Attention-Bestimmung gegeben, in dem die Attention Matrix der beiden Vektoren zueinander bestimmt wird: Die Frage die hierbei beantwortet wird ist, wie welche Wörter in den beiden Sequenzen zueinander stehen. Dies ist gut an dem Beispiel der Sprachübersetzung zu sehen, bei der der zu lernende Output der Input in einer anderen Sprache ist.

The brown dog

wäre entsprechend hier der Output (Decoder Input). Es wird in diesem Schritt analysiert, wie das Wort „The“ zu „Der“, sowie „braune“ zu „brown“ und „Hund“ zu „dog“ steht. Diese Vektoren werden wieder aufaddiert, normiert und in ein Feed-Forward-Netz gegeben, um noch einmal addiert und normiert zu werden.

Danach wird noch eine lineare und eine Softmax-Schicht angewandt, um am Ende in diesem Beispiel das Wort zu erhalten, dass mit der höchsten Wahrscheinlichkeit als nächstes in dem Output-Satz steht. In der Lernphase wird hierzu **Masking** genutzt, bei dem je ein Wort in dem zu lernenden Output-Satz „maskiert“, also als leerer Platzhalter markiert wird.

2.3.1 GPT

OpenAI hat mit **Generative Pretraining** für Sprachmodelle bis zum Zeitpunkt dieser Arbeit drei Hauptversionen von **GPT** veröffentlicht [9][10][11].

Architektur

Die Sprachmodelle nutzen den **Decoder**-Teil des Transformers, um ein **autoregressives** System zu erzeugen: Wie in vorhergegangenen Sprachmodellen wird das jeweils wahrscheinlichste nächste Wort ausgegeben und danach der entstehende Satz weiter als Input genutzt, wie in Abbildung 2.4 zu sehen ist.

Das ursprüngliche **GPT**-Modell besteht aus 12 solcher Decoder-Blöcke, die hintereinander

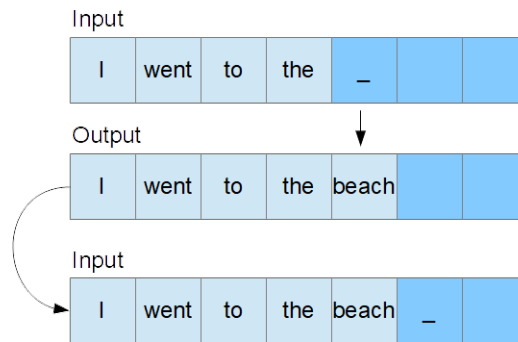


Abbildung 2.4: Konzept eines autoregressiven Modells

durchgegangen werden. Die folgenden Modelle bieten auch jeweils eine Version mit 12 Decodern, dies ist aber immer die kleinste Version. Die größte Version von **GPT-2** umschließt 48 Decoder-Schichten, während **GPT-3** bis 96 Schichten hat. Das momentan größte GPT-Modell hat somit mit Decoder-Parametern und eigenen Parametern für Positional Encoding und Embedding 175 Milliarden Parameter, ein großer Schritt nach den 1,5 Milliarden von **GPT-2**.

Es ist ein Meilenstein für teilüberwachtes und auch unüberwachtes Lernen: **GPT-3** kann Aufgaben teilweise im Ansatz lösen, ohne Beispiele gesehen zu haben oder auch für diese Aufgabe gelernt zu haben. Besser funktioniert aber die **Few-Shot**-Technik: Man zeigt dem Modell eine gewisse Anzahl (unterer zweistelliger Bereich oder weniger) an Beispielen der Aufgabe, die gelöst werden soll und fragt danach ab. Dies ist aufgrund der wenig vorhandenen gelabelten Datensätze ein großer Vorteil dieses Modells.

Input und Output

GPT arbeitet als Input mit sogenannten **Kontexten**: Dazu gehören hier die Beschreibung der zu erfüllenden Aufgabe, eventuelle Beispiele und der **Prompt**, der dazu auffordert, die Aufgabe mit dem Prompt als Input zu lösen. Ein Beispiel für einen Kontext ist [11]:

A "whatpu" is a small, furry animal native to Tanzania.

An example of a sentence that uses the word whatpu is:
We were traveling in Africa and we saw these very cute whatpus.
To do a "farduddle" means to jump up and down really fast.
An example of a sentence that uses the word farduddle is:

Diese Aufgaben und Sätze sind von Menschen geschrieben als Input an das **GPT**-Modell gegeben worden. Die in Anführungszeichen stehenden Wörter sind frei erfunden, um zu zeigen, wie das Sprachmodell mit neuen Wörtern umgeht. Der von **GPT** generierte Output ist:

One day when I was playing tag with my little sister, she got really excited and she started doing these crazy farduddles.

Viel Feinabstimmung braucht **GPT-3** damit nicht mehr, was einer enormen Zeitersparnis entspricht. Die Vorgängermodelle sowie auch **GPT-3** für bessere Ergebnisse sollten jedoch mit einem überwachten Trainingsschritt fein abgestimmt werden.

Trainingsdaten

Die genutzten Trainingsdaten sind ein wesentlicher Unterschied der Versionen: Das erste Modell wurde mit verschiedenen Datensets trainiert, die zusammen eine Größe von knapp über 15GB erreichen, darunter die 1B Word Benchmark und PG-19. Für **GPT-2** wurden ca. 40GB Daten genutzt, enthalten in dem Datensatz **WebText**, der für diese Aufgabe erstellt wurde. Die für **GPT-3** verwendeten Daten hingegen umfassen über 500GB alleine mit dem genutzten Teil des **CommonCrawl** Datensatzes, der Texte aus dem Internet von 2016 bis 2019 enthält und auf Duplikate überprüft wurde. Weiterhin wurden das eigene **WebText2** und die Datensätze **Books1**, **Books2** und das englischsprachige Wikipedia. Die Batchsize dieses Modells ist 0,5 Millionen. Das Modell übertrifft damit an Größe bei Weitem das, was bis zu seiner Veröffentlichung vorhanden war.

Wichtig zu beachten ist, dass diese Modelle aufgrund der Qualität der generierten Texte, die größtenteils nicht mehr von Menschen geschrieben zu unterscheiden sind, auch gewisse Risiken bergen [11]: Es ist deutlich einfacher, damit Fake News und Fehlinformationen zu generieren und zu verbreiten.

2.3.2 BERT

Wie auch **GPT** basiert **BERT** auf der Idee, einen Teil des **Transformers** zu gebrauchen, um ein Sprachmodell zu erstellen.

Architektur

Bei **BERT** wird im Unterschied zu **GPT** nicht der **Decoder**, sondern der **Encoder** genutzt: Es gibt zwei Versionen von **BERT**, die kleinere (**BERT Base**) besteht aus 12 Encoder-Schichten - genannt **Transformer-Blöcke**, während **BERT Large** 24 Blöcke (und damit 340 Millionen Parameter) umfasst [7]. Zur Zeit seiner Veröffentlichung stellte **BERT** für viele Aufgaben in NLP die state-of-the art Ergebnisse.

Schon am Namen dieses Sprachmodells kann man erkennen, dass es nach dem Modell von **ELMo** kommen soll: Die von **GPT** nicht genutzte, aber in **ELMo** sehr erfolgreich angewandte Bidirektionalität sollte wieder eingeführt werden. Dies hat den Hintergrund, dass in Sprache nicht nur die Wörter wichtig für den Kontext eines bestimmten Wortes sind, die im Satz davor stehen - vielmehr ist die ganze Sequenz wichtig, um den ganzen Kontext zu erfassen. Ein reines Feed-Forward-Netz ist dazu nicht in der Lage. Die **Transformer**-Architektur bietet aufgrund der **Self-Attention**-Schichten eine gewisse Möglichkeit für bidirektionalen Kontext. Um dies auszunutzen, wird **BERT** mit verschiedenen Datensätzen und zu einem gewissen Grad randomisiert vortrainiert. Zu diesem Vortraining gibt es zwei Komponenten, die in einem Schritt ausgeführt werden. **BERT** erhält dazu als Input zwei Sätze. Das **Masked Language Model** wird mit jedem der beiden Sätze trainiert, in denen zufällig Wörter maskiert werden, was effektiv einem Lückentext entspricht. **BERT** muss nun lernen, die in dem Satz fehlenden Wörter einzusetzen. Die ausgegebenen Wortvektoren der maskierten Wörter werden mit einer Output-Schicht mit dem Vokabular entsprechender Anzahl an Neuronen (hier 30000, das Vokabular von **WordPieces**) mit Softmax in eine Wahrscheinlichkeitsverteilung umgewandelt und mit einer one-hot codierten Version des tatsächlichen Wortes abgeglichen. Auch möglich ist, dass statt der Maskierung ein Wort durch ein nicht in den Satz passendes ersetzt wird und das Modell diesen Fehler korrigieren soll. Ebenso wird **Next Sentence Prediction** umgesetzt, bei der das Modell ausgibt, ob die beiden als Input gegebenen Sätze korrelieren. Der Output wird, wie bei **Transformern** üblich, parallel über eine gesamte Eingabesequenz ausgegeben.

Für die Feinabstimmung des Sprachmodells werden die Parameter der Output-Schicht durch einen aufgabenspezifischen Datensatz neu erlernt.

Input und Output

Als Input verarbeitet **BERT** Sequenzen, wobei hier spezielle Tokens genutzt werden. Das [MASK]-Token ersetzt maskierte Wörter während des Vortrainings, um zu signalisieren, welche Wörter untersucht werden müssen. Diese Mechanik kann aber auch Probleme erzeugen, da das Token nur während des Vortrainings vorkommt: Bei Feinabstimmung und Test ist es nicht vorhanden, es kann daher nicht mit Sicherheit gesagt werden, wie **BERT** lernt, damit umzugehen [13].

Zu Beginn einer Sequenz wird das Token [CLS] gesetzt, das als Platzhalter für den Klassifikator fungiert. Der Klassifikator wird nach Verarbeitung durch **BERT** mit einem aus einer einzigen Schicht bestehenden Feed-Forward-Netz mit Softmax bestimmt. Soll mehr als ein Label ausgegeben werden muss hier die Anzahl an Output-Neuronen angepasst werden. Für die anderen Wörter der Sequenz werden die erstellten Wortvektoren ausgegeben, bei **BERT Base** haben diese 768 Dimensionen.

Das Token [SEP] wird gebraucht, um zwei zusammenhängende Sequenzen zu trennen, z.B. bei Frage/Antwort-Paaren. Hierbei wird dann an der Position des [CLS]-Tokens kein Klassifikator mehr ausgegeben, sondern die Wahrscheinlichkeit, dass der Satz nach [SEP] auf den Satz vor dem Token folgt.

So kann **BERT** für die verschiedensten Aufgaben verwendet werden. Aber auch die reinen Word Embeddings können ausgelesen und in einem eigenen erstellten Netz für weitere Aufgaben verwendet werden. Da jeder Decoder-Block des Modells einen eigenen Vektor für jedes Wort ausgeben kann, muss man im Einsatz dabei jedoch darauf achten, welche dieser Schichten die am besten Kontextualisierung des Wortes darstellt. In Tests der Entwickler erreichte hierbei eine Verkettung der letzten vier Schichten die besten Ergebnisse [7].

Das Embedding der Wörter in der Vorverarbeitung folgt einer **Byte Pair Codierung** auf Buchstabenebene.

Trainingsdaten

Zum Vortraining von **BERT** wurden die englischsprachigen *Wikipedia* Artikel sowie BookCorpus mit insgesamt 13GB genutzt.

2.3.3 RoBERTa

Da **BERT** ein sehr vielversprechendes Modell war, wurde versucht, dieses zu optimieren, ohne die ursprüngliche Architektur zu ändern. Ein besonderer Fokus lag dabei auf den [MASK]-

Tokens, die in keinem anderen Sprachmodell verwendet werden [8].

Architektur

Die hier benutzte Architektur ist im Grunde die in **BERT** genutzte. Verschiedene Parameter wurden dabei geändert, wie die höchste verwendete Lernrate und den Epsilon-Parameter des Adam Optimizers.

Input und Output

Insgesamt wurde **RoBERTa** am deutlichsten über den Input beim Vortraining geändert. Trotz der besseren Ergebnisse bei **BERT** durch die parallel zum **Masked Language Model** ausgewerteten **Next Sentence Prediction** werden bei **RoBERTa** diese Ergebnisse nicht bestätigt. Bei der Implementation von **RoBERTa** wurde **NSP** entsprechend weggelassen.

Die Encodierung der Sequenzen in der Vorverarbeitung wurde ebenfalls geändert: Es wird eine **Byte Pair Codierung** auf Byte-Ebene (entgegen Buchstabenebene bei **BERT**) genutzt. Dadurch werden die Wörter unterteilt, das Vokabular umfasst 50000 solcher Wortteile. Durch diese Embeddings kommen bis zu 20 Millionen Parameter zu den **BERT**-Versionen hinzu.

Es werden ebenfalls längere Batches mit einer Größe von 8000 verwendet und die Trainingsdaten werden in längere Sequenzen aufgeteilt. Der Einsatz der [MASK]-Tokens wird hier auch ein wenig abgeändert: Statt die Tokens einmal für die Daten zu bestimmen - wenn auch zufällig - werden sie bei **RoBERTa** dynamisch bestimmt. Dies hat den Vorteil, dass das Modell, wenn es einen Satz zum zweiten Mal sieht, dieser nur mit sehr geringer Wahrscheinlichkeit dieselbe Verteilung der Tokens hat. Dadurch kann das Modell den Kontext insgesamt besser erfassen.

Trainingsdaten

Da auch die Entwickler von **BERT** schon angemerkt hatten, dass ein größerer Trainingsdatensatz förderlich für die Performanz des Modells sein würde, wurde dies hier umgesetzt. Zusätzlich zu den vorher verwendeten 13GB an BookCorpus und Wikitext werden die Datensätze **CC-News**, **OpenWebText** (eine frei verfügbare Version des in **GPT** genutzten Datensatzes) und **STORIES** eingesetzt, um insgesamt 160GB an Trainingsdaten zu erhalten, die das Modell dadurch insgesamt nicht so häufig sieht. Damit ist das Modell deutlich größer als das ursprüngliche **BERT**.

2.3.4 XLNet

XLNet basiert auf zwei Schlüsselideen: Der Einsatz einer **RNN**-ähnlichen Struktur, um noch mehr Kontext zu erfassen, als mit dem ursprünglichen **Transformer**-Modell möglich ist, und das Konzept von **BERT** so anzupassen, dass Maskierung im Training nicht mehr gebraucht wird, um eventuelle Fehler zu umgehen [13].

Transformer XL

In diesem Sprachmodell wird eine Abwandlung der **Transformer** genutzt. Da diese Sequenzen parallel verarbeiten, gibt es eine obere Grenze an Wörtern, die gleichzeitig verarbeitet werden können, die darin resultiert, dass auch die Anzahl an Wörtern, die als Kontext gelernt werden begrenzt ist. Diese Grenze ist aufgrund der heutigen Rechenmöglichkeiten recht hoch [13], aber sie existiert. Die Entwickler haben daher eine Technik erfunden, die die Funktionsweise eines **RNNs** nachbildet: Der Status des Netzes wird nach Verarbeitung einer Sequenz eingefroren und der folgenden Sequenz als Input mitgegeben. Dadurch können auch sehr lange Sequenzen „ganz“ verarbeitet werden. Dabei entsteht allerdings das Problem, das bei dem ursprünglichen **Transformer** durch **Positional Encoding** gelöst wurde: Das Modell kann einem Wort im Satz keine Position mehr zuordnen, da alles parallel verarbeitet wird. Dies wird im **Transformer XL** durch **relative positional embedding** umgangen. Durch dieses Embedding wird die relative Position eines Wortes zu einem anderen codiert, sodass die **Attention**-Werte für diese relative Relation korrekt bestimmt werden können.

Architektur

Der Aufbau des Modelles gleicht dem von **BERT**, mit 12 Schichten und den gleich gewählten Hyperparametern sowie einer Sequenzlänge von 512 hat das Modell eine sehr ähnliche Größe. Der Unterschied besteht hier nur in der Nutzung von **Transformer XL** statt der ursprünglichen Version. Diese lieferten bessere Ergebnisse, auch ohne Nutzung der im Folgenden erwähnten Permutation.

Input und Output

Um die Maskierung zu umgehen, die bei **BERT** im Vortraining sehr wichtig ist, wird bei **XLNet** die Idee der **Permutationsmodellierung** eingeführt. Der dadurch entstehende Unterschied wird in Abbildung 2.5 verdeutlicht. Während mit **BERT** alle Wortvektoren inklusive dem im Input maskierten parallel ausgegeben werden, findet hier bei **XLNet** eine Permutation statt: Die

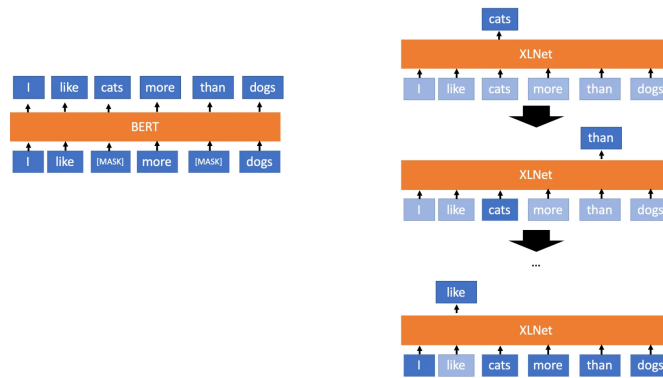


Abbildung 2.5: Unterschied der Textgenerierung von BERT zu XLNet: Permutation [13]

Vektoren werden in einer zufällig gewählten Reihenfolge ausgegeben. Dies gibt dem Modell auch wieder die Möglichkeit der Nutzung des Konzepts der Autoregressivität, das bei früheren Modellen erfolgreich eingesetzt wurde. Zu der autoregressiven Eigenschaft kommt durch die Permutation eine Bidirektionalität, da Wörter zu beiden Seiten des nächsten im Input sein können.

Trainingsdaten

Die bei **BERT** bewährten Trainingsdaten **BookCorpus** und der englischsprachige Teil von *Wikipedia* werden auch hier eingesetzt. Zusätzlich werden **Giga5**, **ClueWeb** und **Common Crawl** in einer gesäuberten Version benutzt, sodass insgesamt 126GB an Textdaten verwendet werden.

3. Einsatz von Sprachmodellen

Im Zuge dieser Arbeit wurden Versuche mit verschiedenen Sprachmodellen durchgeführt, die im Folgenden beschrieben werden.

3.1 Versuchsaufbau

Der Source Code für die Versuche ist in Python verfasst, da die Sprachmodelle alle eine übersichtliche Schnittstelle zur Verwendung in Python bieten. Die erstellten Netze sind durch die Bibliotheken *keras* [20] und *Tensorflow* [21] implementiert worden.

Der Code wurde zum Testen in **Jupyter Notebooks** auf Googles Plattform *Google Colab* [22] ausgeführt, da mit dieser Plattform eine Laufzeitumgebung mit Möglichkeit zur Nutzung von einer GPU sowie - falls nötig - einer TPU zur Verfügung steht. Die genaue Spezifikation der GPU kann dabei nicht ausgewählt werden, es sind *Nvidia K80s*, *T4s*, *P4s* und *P100s* verfügbar [23]. Die genutzten Datensätze werden über Links zu der Datenquelle (*kaggle.com*, *github.com*) oder lokal eingebunden. Für mehrere der Sprachmodelle waren dies nicht genug Ressourcen, deshalb wurden diese Experimente in einer docker-Umgebung auf einem GPU-Server ausgeführt, in der 16 Tesla V100-SXM3 zur Verfügung standen. Die Batch Größe wurde den jeweils verfügbaren Ressourcen angepasst.

Als Vergleichswerte zwischen den Sprachmodellen wurden die Genauigkeiten bei der Auswertung der jeweiligen Datensätze sowie die **F1-score** hergenommen. Der **F1-Wert** berechnet sich wie in Gleichung 3.1 beschrieben, wobei *tp* für *true positives*, *fp* für *false positives* und *tn* für *true negatives* stehen [16].

$$F1 = \frac{tp}{tp + \frac{1}{2}(fp + fn)} \quad (3.1)$$

Die Zeit zum Ausführen wurde gemessen, gibt aber aufgrund der unterschiedlichen genutzten GPUs keine hohe Aussagekraft sondern bloß einen ungefähren Richtwert bei Nutzung einer GPU.

3.1.1 Verwendete Bibliotheken

Für die verschiedenen Sprachmodelle ist die Nutzung mehrerer Bibliotheken für python nötig. Diese werden in dem jeweiligen Notebook eingebunden und teilweise zur Laufzeit installiert. Die wichtigsten dieser Bibliotheken sind im Folgenden aufgestellt:

- **Tensorflow**
- **Fastai**
- **Torch**
- **gpt**
- **simpletransformers** [29] für die Einbindung der transformerbasierten Sprachmodelle
- **sklearn** für Auswertung der Ergebnisse
- **nlTK**, **pandas**, **string** und andere Standardbibliotheken für Textverarbeitung

Die **SimpleTransformers** Bibliothek baut auf der Transformer-Bibliothek von **HuggingFace** auf und bietet einige Funktionen, um den Umgang mit Transformer-Modellen zu erleichtern. Die Effizienz dieser Bibliothek wurde in dieser Arbeit ebenfalls getestet.

3.2 Die Daten

Für **Sentiment Analysis** werden ausschließlich Daten genutzt, die von der Plattform *Twitter* genommen wurden. Diese Daten eignen sich sehr gut für diese Aufgabe, da die Länge der Textstücke auf 280 Zeichen begrenzt ist [19] und es eine große Nutzerbasis und damit eine große Vielfalt an formulierten Texten gibt. Hierbei werden die Tweets genommen, die nur Text und entsprechende Emojis enthalten und in Englisch verfasst sind. Die genutzten Datensätze sind

gelabelt, um damit **Supervised Learning** zu betreiben, da die Feinabstimmung der Sprachmodelle so erfolgen muss.

Die Daten werden alle vorverarbeitet, um die Texte einheitlich und gut verarbeitbar zu machen. Hierzu werden für die Tweets folgende Schritte durchgegangen:

- alle Wörter in Kleinbuchstaben umwandeln
- doppelte Buchstaben entfernen (aus „helloooo“ wird z.B. „hello“)
- Whitespaces an Anfang und Ende entfernen
- Emojis entfernen

Die Daten für den Teil **Stance detection** haben unterschiedliche Ursprünge. Datensätze bestehen aus Artikeln, deren Überschriften und den jeweiligen angesprochenen Stances, oder Tweets und der Haltung des Autors zu einem bestimmten, ebenfalls enthaltenen Thema. Auch diese Daten werden einer Datenreinigung unterzogen. Diese unterscheidet sich nur leicht von der für die **Sentiment Analysis** durchgeführte, da die Artikel im Allgemeinen keine sogenannten *handles* oder Emojis enthalten. Diese Datensätze sind gelabelt, um einen späteren Test möglich zu machen.

Wie bei sehr vielen Machine Learning Aufgaben hängen die Ergebnisse stark von der Qualität der genutzten Daten ab. Es wurde demnach versucht, alle Datensätze so zu reinigen, dass die Performanz optimal wurde. Auf etwaige zusätzliche Schritte wird in der jeweiligen Beschreibung eingegangen.

Die Datensätze werden in Test- und Trainingsdaten aufgeteilt, wobei der Anteil der Testdaten 33% des gesamten Datensatzes ist.

3.2.1 Apple Sentiment

Dieser Datensatz ist manuell gelabelt und wurde ursprünglich von *crowdfunder* [18] zur Verfügung gestellt. Die 1624 englischsprachigen Tweets wurden aufgrund von *hashtags* gefiltert, die die Firma *Apple* betreffen. Die hier verwendete Version [25] ist soweit gefiltert, dass nur Tweets und Labels enthalten sind. Alle nicht relevanten Tweets wurden ebenfalls entfernt. Die Labels dieses Datensatzes sind -1 (negativ), 0 (neutral) und 1 (positiv). Der Datensatz ist dabei unbalanciert, mit 42% an negativen, 49% neutralen und 9% positiven Tweets.

```

df_train.tweet = df_train.tweet.str.lower()
df_test.tweet = df_test.tweet.str.lower()

# Delete URLs
df_train.text = df_train.text.apply(lambda x: re.sub(r'http\S+', '', x))
df_test.text = df_test.text.apply(lambda x: re.sub(r'http\S+', '', x))

#Tokenize, better for emojis, double characters and handle stripping than pure text
tokenizer = TweetTokenizer(strip_handles=True, reduce_len=True)
df_train.tweet = df_train.tweet.apply(lambda x: tokenizer.tokenize(x))
df_test.tweet = df_test.tweet.apply(lambda x: tokenizer.tokenize(x))

# Detokenize for better processing
df_train.tweet = df_train.tweet.apply(lambda x: ' '.join(x))
df_test.tweet = df_test.tweet.apply(lambda x: ' '.join(x))

df_train.tweet = df_train.tweet.map(lambda x :
    x.translate(str.maketrans('', '', string.punctuation)))
df_test.tweet = df_test.tweet.map(lambda x :
    x.translate(str.maketrans('', '', string.punctuation)))

df_train.tweet = df_train.tweet.str.replace("[0-9]", "_")
df_test.tweet = df_test.tweet.str.replace("[0-9]", "_")

df_train.tweet = df_train.tweet.str.strip(string.whitespace)
df_test.tweet = df_test.tweet.str.strip(string.whitespace)

# Recreate index that was shuffeled when splitting test and train
df_train = df_train.reset_index(drop=True)
df_test = df_test.reset_index(drop=True)

```

Listing 3.1: Säuberung des Datensatzes

In diesem Datensatz sind keine Emojis enthalten, die entfernt werden müssten. Die Säuberung erfolgt wie in Listing 3.1 dargestellt und das Ergebnis ist in folgender Tabelle dargestellt:

tweet	sentiment
the secret of life from steve jobs in seconds some motivation from the late ceo of	0
it took just one month for the iphone plus to become the king of phablets is king jobspring technews	1
imessage isnt working thanks	-1

3.2.2 US Airline Sentiment

Wie auch der Datensatz aus Teil 3.2.1 wurde der **US Airline Sentiment** durch Anwendung von Filtern von *Twitter* bezogen [26]. Der Datensatz besteht aus 14640 in Englisch verfassten Tweets mit manuell gelabelten Meinungen der Nutzer zu verschiedenen Fluggesellschaften. Die Einteilung der Sentiment-Werte ist hier als Label statt numerisch gegeben: Es gibt die Labels *neutral*, *negative* und *positive*. Zur Verarbeitung durch die Sprachmodelle wurden diese Labels jeweils numerisch codiert. Auch dieser Datensatz ist unbalanciert: 63% sind negativ, 21% neutral und 16% positiv.

Da in den Texten noch Emojis vorhanden sind, werden diese durch den in Listing 3.2 gezeigten Code entfernt. Es wurden ebenfalls Versuche mit noch enthaltenen Emojis durchgeführt. Beispiele aus dem gesäuberten Datensatz sind in der folgenden Tabelle gegeben.

```
def remove_emoji(string):
    emoji_pattern = re.compile("[
        u"\U0001F600-\U0001F64F"    # emoticons
        u"\U0001F300-\U0001F5FF"    # symbols & pictographs
        u"\U0001F680-\U0001F6FF"    # transport & map symbols
        u"\U0001F1E0-\U0001F1FF"    # flags (iOS)
        u"\U00002702-\U000027B0"
        u"\U000024C2-\U0001F251"
        "]+", flags=re.UNICODE)
    return emoji_pattern.sub(r'', string)
```

Listing 3.2: Funktion zum Entfernen von Emojis

tweet	sentiment
youre my early frontrunner for best airline oscars	positive
what is going on with your bdl to dca flights yesterday and today why is every single one getting delayed	negative
do they have to depart from washington d c	neutral

3.2.3 T4SA

Der **T4SA** Datensatz [14] wurde von seinen Entwicklern zur **Cross-Media-Analyse** genutzt. Hierbei wurden zunächst die Texte der Tweets nach sentiment-Werten klassifiziert, um die ebenfalls in den Tweets enthaltenen Bilder zu labeln. Auch diese Tweets wurden gefiltert um sicherzustellen, dass alle Tweets Bilder enthalten, in Englisch verfasst und mindestens fünf Wörter lang sind. In dieser Arbeit stehen die enthaltenen Bilder nicht zur Betrachtung, aber der zu Sentiment-Werten gelabelte Datensatz lässt sich gut verwenden.

Hier genutzt werden 1179957 der im Datensatz enthaltenen Tweets. Diese Teilmenge besteht nur aus mit Sentiment-Werten versehenen Tweets. Dabei sind 15% negativ, 53% neutral und 31% positiv. Da die Daten auf mehrere Dateien verteilt sind, ist eine etwas aufwendigere Vorverarbeitung notwendig: Zusätzlich zu der in Listing 3.1 und 3.2 aufgeführten Testverarbeitung werden die beiden Dateien in einen Dataframe zusammengefasst, indem die Tweet-ID jeweils als Schlüsselwert genommen wird. Die Sentiment-Werte liegen in der Original-Datei als in drei Spalten (*POS*, *NEG* und *NEU*) aufgeteilte Wahrscheinlichkeitswerte vor. Der Spaltenname mit der höchsten eingetragenen Wahrscheinlichkeit wird im Folgenden als das Label der Zeile genommen und in den Modellen entsprechend numerisch codiert. Beispiele aus dem final genutzten Dataframe sind in der folgenden Tabelle aufgeführt.

tweet	sentiment
inbound marketing content ideas part	NEU
i think this song is about phil im excited	POS
yo this the ugliest filter eveeer son im weaaak	NEG

3.2.4 Fake News Challenge

Dieser Datensatz besteht aus 49972 Artikeln, verschiedenen Aussagen, die über den Text (oder einen anderen) getroffen wurden und der Beziehung zwischen den Aussagen und den Artikeln

[27]. Beziehungen werden in die Kategorien „unrelated“, „agree“, „discuss“ und „disagree“ eingeordnet. Diese Klassifizierung wurde für die Verarbeitung durch die Modelle numerisch codiert. Der Inhalt ist unbalanciert, 73% der Trainingsdaten sind mit „unrelated“, 18% mit „discuss“ und die restlichen 9% mit „disagree“ oder „agree“ gelabelt. Da es sich hier um Artikel handelt, müssen bei der Datenreinigung keine Emojis oder *Twitter-Handles* beachtet werden. Die in unterschiedlichen Dateien vorliegenden Artikel werden den Aussagen zugeordnet und danach in Kleinbuchstaben umgewandelt. Zeichensetzung sowie Whitespaces an Anfang und Ende des Artikels werden entfernt. Gekürzte Beispiele aus dem Datensatz sind im Folgenden gelistet.

Aussage	Artikel	Stance
mexico says students not among dead in mass grave	more graves have been discovered at the site where 43 students went missing	discuss
isis beheads american photojournalist james wright foley threatens more to come	afghanistan veteran sam arnold uploaded this spinechilling video of a us marine getting a direct headshot from a taliban sniper [...]	unrelated

3.2.5 SemEval

SemEval wurde ursprünglich für einen ausgeschriebenen Wettbewerb erstellt [28]. In diesem Datensatz sind 2814 Tweets enthalten, die gegen unterschiedliche **Targets** auf ihre Haltung überprüft wurden. Die Haltung ist klassifiziert durch die Kategorien „AGAINST“ (48%), „FAVOR“ (26%) und „NONE“ (26%). Auch hier wurden die Kategorien numerisch codiert. Die Säuberung läuft analog zu der Säuberung der Tweets in den vorangegangenen Datensätzen. Die Themen dieses Wettbewerbs sind *Hillary Clinton*, *Donald Trump*, *Atheismus*, *Feminismus*, *Klimawandel ist real* und *das Recht auf Abtreibung*.

Target	Tweet	Stance
hillary clinton	and handovertheserver she wiped clean 30k deleted emails explains dereliction of duty lies re benghazi etc tcot semst	AGAINST
hillary clinton	hillary is our best choice if we truly want to continue being a progressive nation ohio semst	FAVOR

3.3 Sentiment Analysis

Im Folgenden werden die Experimente zu **Sentiment Analysis** im Detail beschrieben. Hierbei wird zunächst das verwendete Modell beschrieben und dann auf die spezifischen Aspekte wie Codierung der Labels und Ergebnisse zu den verschiedenen Datensätze eingegangen.

3.3.1 Ausgangsbasis

Um einen Vergleich zu haben, wie performant und genau **Sentiment Analysis** mit Sprachmodellen ist, wurde zunächst ein Ansatz ausgewertet, der keinen Gebrauch von Deep Learning macht. Die Veränderung der Genauigkeit im Vergleich zu diesem Ansatz gibt einen Anhaltspunkt, wie viel Sprachmodelle in Sentiment Analysis leisten können.

Der jeweilige Datensatz wird mit pythons *pandas* Bibliothek eingelesen, gesäubert und im Anschluss mit der Bibliothek *TextBlob* ausgewertet. Diese Auswertung basiert auf manuell erstellten Dokumenten, in denen allen Wörtern Sentiment-Werte zugeordnet wurden. *TextBlob* werten diese Werte für einen gegebenen Text aus und gibt einen Wert zwischen -1 und 1 zurück (negativ zu positiv). Um mit den Labels der Datensätze vergleichbar zu sein, wurden diese float-Werte zur jeweils nächsten ganzen Zahl gerundet. Da diese Auswertung sequentiell erfolgt, wurde keine GPU genutzt. Die Auswertung erfolgt durch den in Listing 3.3 aufgezeigten Code. Um besser vergleichbar zu sein, wurde, wie in den anderen Modellen jeweils nur der Test-Dataframe ausgewertet.

```
data.insert(2, "blobpolarity",
            data.text.map(lambda x: int(round(TextBlob(x).sentiment.polarity)),
                          True))
```

Listing 3.3: Auswertung mit TextBlob

Apple Sentiment

Die Einstufungen müssen bei diesem Datensatz nicht umgerechnet werden. Aufgrund der begrenzten Anzahl an Daten in dem Datensatz dauert die Berechnung der Sentiment-Werte unter einer Minute. Der Genauigkeitswert liegt bei 53,0%.

US Airline Sentiment

In diesem Datensatz werden die Labels in eine numerische Form gebracht, sodass die Werte analog zum letzten Experiment bei -1, 0 und 1 liegen. Auch bei diesem Datensatz dauert die Auswertung unter einer Minute. Es wird eine Genauigkeit von 25,7% erreicht.

T4SA

Wie in dem **US Airline Sentiment** Datensatz werden die Labels numerisch codiert. Die Auswertung dauert 3 Minuten und ergibt eine Genauigkeit von 66,1%.

Auswertung

Die Genauigkeitswerte schwanken stark, da kein Lernvorgang stattfindet und alle Datensätze nach dem gleichen Prinzip ausgewertet werden, obwohl sie an sich unterschiedliche Sprachaspekte enthalten. Die häufigste Klassifizierung von TextBlob ist „Neutral“, wodurch die **F1-Score** hier jeweils am höchsten ist. Die Datensätze sind dabei nicht ganz ausgeglichen und in den Testsätzen **Apple Sentiment** und **T4SA** sind mehr neutral klassifizierte Tweets enthalten, was sich in der höheren Genauigkeit niederschlägt.

F1 score	Apple Sentiment	US Airline Sentiment	T4SA
-1	0,156	0,098	0,185
0	0,675	0,349	0,758
1	0,286	0,283	0,543

3.3.2 ELMo

Die verwendeten Datensätze werden zunächst gesäubert und in einen Dataframe eingelesen. Daraufhin wird das **ELMo-Modell** geladen und die Vektoren für die Eingabe bestimmt: Hierbei wird für eine Eingabesequenz ein einziger Vektor durch Mittelwertbildung bestimmt, um eine gute Weiterverarbeitung zu garantieren. Diese 1024-dimensionalen Vektoren werden abgespeichert, um immer auf sie zugreifen zu können.

Da **ELMo** kein eigenes Modell zur Sentiment Analysis bietet, werden hier zwei verschiedene Ansätze erprobt:

- Logistic Regression
- einschichtiges LSTM-Netz

Bei der **Logistischen Regression** wird eine Funktion gesucht, die die Beziehung zwischen den **ELMo**-Vektoren und den Labels des Datensatzes modelliert.

Das einschichtige LSTM-Netz wurde mithilfe von *keras* implementiert, der Input ist der 1024-dimensionale Wortvektor. Dieser wird in eine LSTM-Schicht mit 512 Neuronen gegeben, um danach in einer **fully connected** Schicht weiterverarbeitet und zu einem drei Labels umfassenden Output gemappt, wie Listing 3.4 zeigt. Die Anzahl der zu trainierenden Epochen wurde anhand des **Validation Loss** bestimmt.

Layer (type)	Output Shape	Param #
lstm_1 (LSTM)	(None, 512)	3147776
dense_1 (Dense)	(None, 3)	1539
activation_1 (Activation)	(None, 3)	0

Listing 3.4: Aufbau des LSTM-Netzes

Apple Sentiment

Die Bestimmung der Wortvektoren dauert ca. 15 Sekunden, die Auswertung erfolgt in unter einer Sekunde. Mit der **Logistischen Regression** wird eine Genauigkeit von 83,8% erreicht. Das LSTM-Netz wird in 20 Epochen trainiert und liefert eine Genauigkeit von 82,0%.

US Airline Sentiment

ELMo Wortvektoren sind nach 45 Sekunden berechnet, die Berechnung der **Logistischen Regression** dauert eine halbe Minute. Dadurch wird eine Genauigkeit von 81,0% erreicht. Aus dem wieder in 20 Epochen trainierten LSTM-Netz folgt eine Genauigkeit von 81,5%.

T4SA

Die Länge des Datensatzes wirkt sich in einer längeren Berechnungszeit aus: Die Wortvektoren sind nach 40 Minuten bestimmt. Auch die Berechnung der **Logistischen Regression** dauert mit 4 Minuten etwas länger und das in 20 Epochen trainierte LSTM-Modell braucht 30 Minuten. Die erreichten Genauigkeiten liegen bei 92,8% bzw. 95,4%.

Auswertung

Die erreichten Genauigkeiten wie auch die **F1**-Werte weisen darauf hin, dass bei der Arbeit mit **ELMo**-Vektoren der Einsatz von **Logistischer Regression** bei kleineren Datensätzen ausreichend ist, um ein Modell zu bestimmen, ein LSTM-Netz gibt kaum Verbesserungen. Da der **T4SA**-Datensatz groß ist, kann das LSTM-Netz länger trainiert werden und erreicht dementsprechend bessere Ergebnisse als die **Logistische Regression**. Die in diesem Experiment errechneten Genauigkeiten sind weit von state-of-the-art-Ergebnissen entfernt.

Man kann erkennen, dass die ersten beiden Datensätze zu negativen Sentiments unbalanciert sind, mit einem längeren Datensatz scheint dies aber ausgeglichen - hier ist das dritte Label häufig genug vorhanden. **Logistische Regression**

F1 score	Apple Sentiment	US Airline Sentiment	T4SA
-1	0,857	0,884	0,855
0	0,862	0,632	0,950
1	0,5	0,730	0,925

LSTM-Netz

F1 score	Apple Sentiment	US Airline Sentiment	T4SA
-1	0,843	0,887	0,904
0	0,845	0,642	0,969
1	0,419	0,728	0,953

3.3.3 ULMFit

Dieses Modell wurde nach dem Beispiel in [2] implementiert. Zunächst werden die Textdaten importiert und gereinigt. Dabei wird für alle Datensätze außer **Sentiment140** eine **one-hot**-

Codierung ausgeführt, da mit dem Modell Multi-Klassen Klassifizierung sonst nicht möglich ist. Dies resultiert in drei neuen Spalten in den Dataframes (negativ, neutral und positiv), die jeweils binär codiert sind.

```
learn = language_model_learner(data_lm , drop_mult=0.7 ,  
                                arch = AWD_LSTM, pretrained = True)
```

Listing 3.5: Laden des ULMFit Modells

Daraufhin wird das geladene **ULMFit** Sprachmodell mit den Tweet-Texten der Trainingsdaten initialisiert und verschiedene Lernraten mittels eines Lernratenfinders ausprobiert. Wie der Grafik 3.1 zu entnehmen ist, ist die gewählte Lernrate 0,01. Der nächste Schritt besteht aus

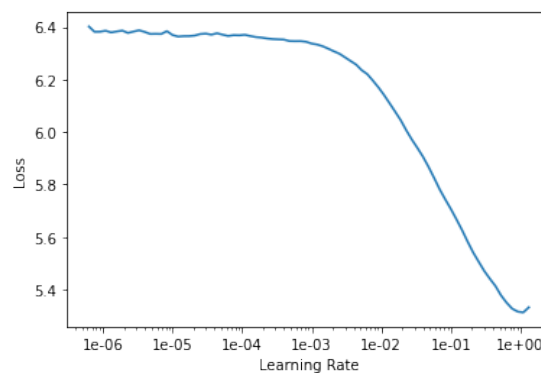


Abbildung 3.1: Lernratenfinder

dem **unfreezing** der Gewichte und dem Trainieren des Sprachmodells mit den Textdaten. Dazu wird die Anzahl der Epochen an den jeweiligen Datensatz angepasst. Es wird eine Anzahl gewählt, die der Größe des Datensatzes angemessen ist und nach dem **Validation Loss** entschieden, wann gestoppt wird: Die maximale Anzahl der Epochen ist erreicht, sobald das **Validation Loss** höher wird um **Overfitting** zu vermeiden. Die daraus errechneten Gewichte werden für die spätere Verwendung abgespeichert.

Danach wird der **Classifier** trainiert. Dieser erhält als Input die Textdaten und die drei Spalten für die Labels und wird nach den Metriken *Genauigkeit*, *precision* und *recall* trainiert. Mit **gradual unfreezing** werden zunächst nur die letzten zwei Schichten des Modells angepasst und im nächsten Schritt noch einmal alle Schichten trainiert. Dieser Vorgang kann dem Listing 3.6 entnommen werden.

```
learn.freeze_to(-2)  
learn.fit_one_cycle(2, slice(1e-2/(2.6**4), 1e-2), moms=(0.8,0.7), wd=0.1)
```

```
learn.unfreeze()  
learn.fit_one_cycle(2, slice(1e-3/(2.6**4), 1e-3), moms=(0.8,0.7), wd=0.1)
```

Listing 3.6: Gradual Unfreezing zum Trainieren des Classifiers

Apple Sentiment

Auch bei diesem Modell dauert das Trainieren des Modells in 20 Epochen und des Classifiers mit dem Datensatz unter einer Minute. Mit einer GPU kann hier schon ein gewisser Grad an Parallelisierung stattfinden. Erreichte Genauigkeiten liegen bei 82,4%.

US Airline Sentiment

Durch die **one-hot-Codierung** des Datensatzes fällt die anderweitige numerische Codierung der Labels weg. Die ermittelte Anzahl an Epochen, die das Sprachmodell trainiert wird, liegt bei zehn. Damit dauert das Training des Sprachmodells 50 Sekunden. Die Feinabstimmung des Classifiers dauert eine weitere Minute, es werden Genauigkeiten bis 88,5% erreicht.

T4SA

Die Vorverarbeitung verläuft analog zu der des **US Airline Sentiment** Datensatzes. Das Sprachmodell wird aufgrund der Menge der Daten 10 Epochen trainiert, die Trainingsdauer liegt damit bei ca 90 Minuten. Die Feinabstimmung des Classifiers dauert weitere 90 Minuten und die Genauigkeit liegt am Ende bei 97,5%. Dieser Datensatz würde von längerem Training noch einen großen Nutzen ziehen: Nach den vergangenen Epochen war noch ein starker sinkender Trend im **Validation Loss** zu erkennen. Dies kann mit der Größe des Datensatzes zusammenhängen, da kaum Möglichkeiten zum **Overfitting** bestehen.

Auswertung

Hier kann sehr gut beobachtet werden, wie sich die Größe des Datensatzes auf die Ergebnisse auswirkt: Die steigende Genauigkeit und **F1-Scores** zeigen, dass das Modell daran insgesamt

besser trainieren kann. Die Gefahr des **Overfitting** ist ebenfalls sehr gering und das Modell könnte insgesamt länger als hier angegeben trainiert werden.

F1 score	Apple Sentiment	US Airline Sentiment	T4SA
-1	0,743	0,896	0,957
0	0,783	0,670	0,986
1	0,25	0,756	0,981

3.3.4 RoBERTa

Für dieses Sprachmodell müssen die Eingabedaten leicht angepasst werden: Die Labels werden in die Zahlen 0, 1 und 2 codiert, damit von dem Modell verarbeitet werden können. Für die in Text gegebenen Label wird dies mit einem *dictionary* gelöst:

```
thisdict = {
    "negative": 0,
    "neutral": 1,
    "positive": 2
}
df_train.sentiment = df_train.sentiment.apply(lambda x: thisdict[x])
df_test.sentiment = df_test.sentiment.apply(lambda x: thisdict[x])
```

Listing 3.7: Umkodieren der sentiment-Label

Daraufhin wird das vortrainierte Modell von *pytorch* geladen und auf Multi-Label-Erkennung konfiguriert.

```
config = RobertaConfig.from_pretrained('roberta-base')
# Set number of output labels
config.num_labels = 3
```

Listing 3.8: Laden und Konfigurieren des RoBERTa-Modells

Die Eingabetexte werden danach in **RoBERTa**-Tokens und weiterhin **CUDA**-Vektoren umgewandelt, um von der Parallelisierung durch GPUs Gebrauch zu machen. Hierbei gibt es Probleme mit dem Zero-Padding: Da das Modell die speziellen [CLS] und [SEP]-Tokens braucht, kann dies nicht einfach durchgeführt werden, was dazu führt, dass die Batchgröße nicht angepasst werden kann.

Das Modell wird im nächsten Schritt über 10 Epochen mit einer Lernrate von 1^{-5} und dem **Adam-Optimizer** trainiert. Andere Lernraten wurden ausprobiert, gaben aber schlechtere Ergebnisse.

Eine alternative Version wurde mithilfe der **SimpleTransformers** Bibliothek implementiert. Diese hat den Vorteil, dass sie einfach zu nutzen und übersichtlich ist, wie aus Listing 3.9 hervorgeht.

```
from simpletransformers.classification import ClassificationModel

model = ClassificationModel('roberta', 'roberta-base', num_labels=3, args={
    'learning_rate': 3e-5,
    'num_train_epochs': 10,
    'reprocess_input_data': True,
    'overwrite_output_dir': True,
    'process_count': 10,
    'train_batch_size': 4,
    'eval_batch_size': 4,
    'max_seq_length': 512,
    'n_gpu': 4,
    'fp16': False
})

model.train_model(df_train)
```

Listing 3.9: RoBERTa mit SimpleTransformers

Apple Sentiment

Das Training über zehn Epochen mit diesem Datensatz dauert eine halbe Stunde. Weniger Epochen führen dazu, dass der Classifier nur zwischen zwei Klassen unterscheiden kann und die dritte außer Acht lässt, mit mehr Epochen steigt die Gefahr des **Overfitting**. Die erreichte Genauigkeit liegt bei 78,6%, da die dritte Klasse nur selten erkannt wird.

Mit dem **SimpleTransformer** Modell dauert das Training mit zehn Epochen 15 Minuten und ergibt eine Genauigkeit von 88,7%. Auch hier wird die dritte Klasse erst nach einigen Epochen Training erkannt. Weiteres Training führt zu **Overfitting**

US Airline Sentiment

Ohne **SimpleTransformers** wurde die Bearbeitung abgebrochen, da es so ineffizient ist, dass in 12 Stunden keine Epoche berechnet werden kann. Die Bearbeitungszeit mit der **SimpleTransformers** Implementation lag bei 15 Epochen und 90 Minuten mit 16 GPUs, der Genauigkeitswert liegt damit bei 85,0%.

T4SA

Da selbst unter Einsatz des GPU-Servers eine Epoche über neun Stunden zur Ausführung braucht, kann dieser Datensatz im Rahmen der Arbeit nicht mit der eigenen Implementierung untersucht werden. Mit **SimpleTransformers** dauert die Berechnung einer Epoche drei Stunden, liefert aber da schon eine Genauigkeit von 98,2%.

Auswertung

Das **RoBERTa**-Modell kann mit Multi-Klassen Umgebungen und unbalancierten Datensätzen nicht gut umgehen. Dies kann man besonders gut an den **F1**-Werten für das positive Label sehen: Um ein besseres Ergebnis zu erreichen muss mit vielen Daten lange trainiert werden. Die Implementierung an sich ist sehr ineffizient und kann daher mit großen Datensätzen nicht gut trainiert werden.

F1 score	Apple Sentiment	US Airline Sentiment	T4SA
-1	0,801	-	-
0	0,826	-	-
1	0,3	-	-

Auch bei **SimpleTransformers** kann man die Unbalanciertheit der Daten gut erkennen, mit langen Datensätzen ist dies ausgeglichen.

F1 score	Apple Sentiment	US Airline Sentiment	T4SA
-1	0,908	0,913	0,964
0	0,896	0,688	0,987
1	0,714	0,796	0,981

3.3.5 XLNet

Wie auch bei dem **RoBERTa**-Modell werden hier die Labels so codiert, dass sie mit Indizes vergleichbar sind. Die für die Verarbeitung durch **XLNet** nötigen Tokens [SEP] und [CLS] werden an jeden Tweet angehängt und der Text wird mit einem `XLNetTokenizer` zu Tokens umgewandelt. Dies ergibt folgende Struktur für die Tweets:

```
[ '_we', '_need', '_more', '_products', '_like', '_companies', '_like',  
'_need', '_to', '_embrace', '_the', '_customization', '_of', '_quality',  
'_themes', '[', 's', 'ep', ']', '_[', 'cl', 's', ']' ]
```

Listing 3.10: Tweet nach Tokenization

Diese Tokens werden mit dem in Listing 3.11 aufgeführten Code in ids umgewandelt und daraufhin auf eine einheitliche Länge gebracht.

```
ids_train = [tokenizer.convert_tokens_to_ids(x) for x in tokenized_text_train]  
input_ids_train2 = pad_sequences(ids_train, maxlen=MAX_LEN, dtype="long",  
                                truncating="post", padding="post")
```

Listing 3.11: Umwandlung in Ids

Im nächsten Schritt werden die entstandenen IDs in Torch Tensoren gewandelt, um sie mit einem `DataLoader` verarbeiten zu können.

Das vortrainierte **XLNet** wird geladen und auf die Anzahl der Labels (drei) konfiguriert. Mit dem **Adam-Optimizer** und einer Lernrate von 1^{-5} wird das Modell mit dem jeweiligen Datensatz trainiert. Andere Lernraten geben schlechtere Ergebnisse. Die Anzahl der zu lernenden Epochen wurde dem jeweiligen Datensatz angepasst.

Da es sich um ein transformerbasiertes Modell handelt, wurde eine Alternative zum oben beschriebenen Vorgehen mittel **SimpleTransformers** implementiert. Diese unterscheidet sich nur geringfügig von dem in Listing 3.9 aufgeführten Code, hier wird statt des **RoBERTa**-Modells **XLNet** geladen und trainiert.

Apple Sentiment

Die Berechnung und Auswertung mit 40 Epochen Training dauert 20 Minuten und liefert eine Genauigkeit von 77,9%. Bei wenigen Epochen wird auch hier das positive Label nicht mit

einbezogen. Mehr Epochen führen bei dem kleinen Datensatz zu **Overfitting**. Auf dem GPU-Server dauert das Training in fünf Epochen bei dem **SimpleTransformers**-Modell neun Minuten und erreicht eine bessere Genauigkeit als Training mit zehn Epochen. Der maximale Wert liegt hier bei 86,1%.

US Airline Sentiment

Das Training dauert bei ebenfalls 40 Epochen ca. 40 Minuten, die dabei erreichte Genauigkeit beträgt 66,2%. Der beste Genauigkeitswert des **SimpleTransformers** Modells wurde mit fünf Epochen Training in 35 Minuten erreicht (ggü. 15 Epochen). Er liegt bei 84,3%.

T4SA

Da selbst unter Einsatz des GPU-Servers eine Epoche über neun Stunden zur Ausführung braucht, kann dieser Datensatz im Rahmen der Arbeit nicht mit der eigenen Implementierung untersucht werden. Die Implementation mit **SimpleTransformers** dauert unter Nutzung der 16 GPUs fünf Stunden für eine Epoche, liefert aber hier schon eine Genauigkeit von 98,0%. Die Auswertung dauerte sechs Stunden.

Auswertung

Auch hier ist zu beobachten, dass die Multi-Klassen Klassifizierung erst nach mindestens 10 Epochen funktioniert. Wieder empfiehlt es sich, bei den größeren Datensätzen länger zu lernen, bei den kleinen kann aufgrund von **Overfitting** nicht viel geändert werden. Dies ist für beide Implementierungen der Fall, wie auch schon bei **RoBERTa**. Wieder ist die **SimpleTransformer**-Implementierung effizienter und liefert bessere Ergebnisse.

F1 score	Apple Sentiment	US Airline Sentiment	T4SA
-1	0,763	0,809	-
0	0,836	0,283	-
1	0,447	0,320	-

SimpleTransformers

F1 score	Apple Sentiment	US Airline Sentiment	T4SA
-1	0,877	0,906	0,961
0	0,879	0,668	0,986
1	0,632	0,794	0,980

3.3.6 GPT

Das von *OpenAI* direkt bereitgestellte **GPT**-Modell wurde ebenfalls getestet, ergab aber auch nach intensivem Training keine brauchbaren Ergebnisse. Die dazu aufgestellte These ist, dass die Daten im Gegensatz zu dem Trainingsdatensatz beim Vortraining zu wenig und die Texte insgesamt zu kurz sind. Die jeweilige Ausgabe ergab teilweise eine sinnvolle Fortsetzung des voranstehenden Tweets, die tatsächlichen Label wurden demnach nicht gelernt, wie in Listing 3.12 zu sehen ist. Es ist möglich, dass mit sehr langem Training hier Ergebnisse erzielt werden können, dies ist im Rahmen dieser Arbeit aber nicht durchführbar.

```

Model prompt >>> // x bonus airmiles right now airmilesshops ||
===== SAMPLE 1 =====
noon
=====
Model prompt >>> // since when did stop replacing iphone units due to defect
just because i have some scratches on my phone doesnt mean i dropped it ||
===== SAMPLE 1 =====
nooooooooooooooooooooo
=====
Model prompt >>> // i had to get my logic board completely replaced if its
under warranty theyll do it for free make sure to back up b ||
===== SAMPLE 1 =====

=====
Model prompt >>> // thanks for pushing me into yosemite it has been the worst
so slow sand clunky cantgoback misssnowleopard whathappened ||
===== SAMPLE 1 =====
|| cm
=====

```

Listing 3.12: Ergebnisse des GPT-Modells

3.4 Stance Detection

In den folgenden Unterkapiteln wird auf die Experimente zu **Stance Detection** eingegangen.

3.4.1 ULMFit

Entsprechend dem Modell, dass für **Sentiment Analysis** genutzt wurde, wurde hier mit den Textdaten ein Sprachmodell antrainiert und anschließend der Classifier mit den Labels trainiert. Die Eingangsdaten wurden zur Verarbeitung der Klassen one-hot-codiert. Das vortrainierte fastai-Sprachmodell konnte ohne Probleme mit den zwei Textinputs umgehen.

FNC

Nach insgesamt 210 Minuten Training über 20 Epochen für das Sprachmodell und zehn für den Classifier liegt die Genauigkeit bei 94,5%.

SemEval

Aufgrund der kleinen Anzahl an Daten kann dieses Modell nicht gut trainiert werden. Das Training des Sprachmodells wie auch des Classifiers dauert unter einer Minute, nach acht Epochen sieht man durch steigendes **Validation Loss** (Abbildung 3.2) hier schon einen Ansatz von **Overfitting**. Die erreichte Genauigkeit liegt bei 40,1%.

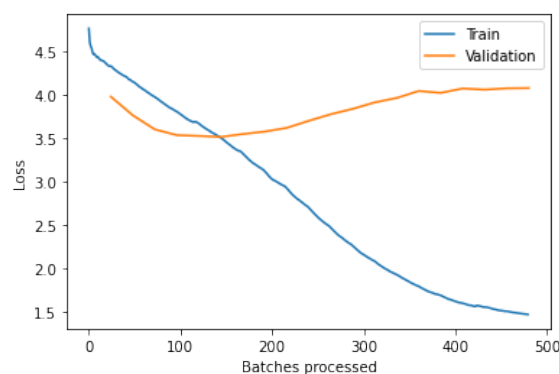


Abbildung 3.2: Steigendes Validation Loss

Auswertung

Die bei dem kleineren Datensatz (**SemEval**) erreichte Genauigkeit liegt unter der, die erreicht werden kann, wenn nur in eine Kategorie eingeordnet wird. Dies weist darauf hin, dass der Datensatz zu klein ist, um ihn in diesem Modell zu verwenden.

Da der zweite Datensatz mit hoher Wahrscheinlichkeit nicht richtig ausgewertet werden kann, kann darüber keine Aussage getroffen werden. Getestet wurde, ob zwischen den Trainings- und Testdatensätzen Dopplungen vorliegen. Dies war nicht der Fall, aber da verschiedene Stances mehrmals in beiden Datensätzen vorkommen, ist zu vermuten, dass das Ergebnis verfälscht ist. Es liegt aber weit über den durch reines „unrelated“ Kategorisieren erreichbaren 73%. Da kein eigener Testdatensatz vorhanden ist, kann dies jedoch nicht überprüft werden. Der Genauigkeitswert steigt jedoch immer weiter mit fortlaufendem Training, daher wird angenommen, dass **Overfitting** stattfindet, dass durch die Testdaten nicht aufgedeckt werden kann.

F1 score	FNC	SemEval
0	0,987	0,680
1	0,809	0,462
2	0,939	0,483
3	0,594	-

3.4.2 RoBERTa

Die Daten müssen für das mit **SimpleTransformers** implementierte Training in einem Dataframe vorliegen, dass die Spalten „text_a“, „text_b“ und „labels“ hat. Entsprechend wurden die Daten umformatiert. Die Anzahl der Labels wurde entsprechend dem Datensatz angepasst. Mit einer Lernrate von 3^{-5} wurde das Modell für eine dem Datensatz angepassten Anzahl an Epochen trainiert.

FNC

Das Trainieren in zehn Epochen mit 16 GPUs dauert 180 Minuten, die dabei erreichte Genauigkeit ist 99,2%. Wie auch schon bei **ULMFit** angemerkt wurde, ist dieses Ergebnis mit hoher Wahrscheinlichkeit jedoch verfälscht.

SemEval

Dieses Experiment wurde auf dem GPU-Server unter Einsatz von 16 GPUs ausgeführt. fünf Epochen brauchen dabei 15, zehn Epochen 19 und 15 Epochen 28 Minuten. Die erreichte Genauigkeit ist mit 62,6% bei zehn Epochen maximal.

Auswertung

Die erreichte Genauigkeit ist weit von einem Idealwert entfernt, gegenüber dem **ULMFit** Modell ist sie aber schon genügend hoch um zu sagen, dass dieses Modell besser lernt. Es ist wahrscheinlich, dass nicht genug Beispiele in dem Datensatz existieren, er ist zu klein.

F1 score	FNC	SemEval
0	0,999	0,678
1	0,965	0,560
2	0,983	0,586
3	0,932	-

3.4.3 XLNet

Beide Datensätze werden in die von **SimpleTransformers** benötigte Form gebracht und ausgewertet. Die Konfiguration des **XLNet**-Modells ist analog zu der in **Sentiment Analysis** vorgestellten Variante, die Anzahl der Labels und Epochen wird entsprechend angepasst.

Es wurde eine weitere Version analog zu der in **Sentiment Analysis** besprochenen implementiert, in der die Textspalten codiert und in Tokens umgewandelt wurden. Hierzu wurden die zu analysierenden Texte nach dem [SEP]-Token an die Aussage bzw. das Target angehängen, das weitere Vorgehen ist identisch.

FNC

Die eigene Implementation ist ineffizient und versucht, mehr Speicher auf einer Grafikkarte zu allozieren, als vorhanden ist, daher konnte dieser Versuch nicht durchgeführt werden. Bei Nutzung von **SimpleTransformers** dauert das Training von 10 Epochen vier Stunden, der Genauigkeitswert liegt bei 99,4%.

SemEval

Das nicht mit **SimpleTransformers** implementierte Modell wurde in zehn Minuten für 20 Epochen trainiert, kann aber nicht zwischen den eigentlichen Labels unterscheiden. Die Genauigkeit von 51,8% ist davon abzuleiten, dass die Daten stark unbalanciert sind.

Das andere Modell erreichte nach 15 Epochen Training (32 Minuten mit 16 GPUs) eine Genauigkeit von 57,4%.

Auswertung

Das Modell ist in der Performanz schlechter als **RoBERTa**. Wieder kann man an den Werten auch die Qualität und Länge der jeweiligen Datensätze erkennen.

F1 score	FNC	SemEval
0	0,999	0,649
1	0,976	0,525
2	0,986	0,487
3	0,966	-

Ohne SimpleTransformers

F1 score	FNC	SemEval
0	-	0,682
1	-	0
2	-	0
3	-	-

4. Auswertung

Im Folgenden werden die Ergebnisse aus den Experimenten des letzten Kapitels besprochen und mögliche Verbesserungen angesprochen.

4.1 Ergebnisse

Bei **Sentiment Analysis** werden durch die Sprachmodelle schnell bessere Ergebnisse erreicht als ohne. Besonders die nicht-transformerbasierten Modelle sind zeitlich bei der Ausführung sehr gut, die Genauigkeitswerte sind mit über 80% durchaus wettbewerbsfähig (s. Abbildung 4.1). Besonders **ULMFit** scheint gut geeignet für diese Aufgabe, da auch die Verarbeitungszeit hier, nicht wie bei den Transformer-Modellen, dem Datensatz angemessen ist. Ist ein großer Datensatz vorhanden, kann auch entsprechend gut trainiert werden. Die transformerbasierten Modelle scheinen deutlich resistenter gegen **Overfitting** auch bei kleinen Datensätzen zu sein, brauchen aber langes Training, um wirklich hohe Genauigkeiten zu erreichen. Das Problem scheint hierbei hauptsächlich bei der Multi-Klassen Klassifizierung zu liegen, da diese Modelle länger brauchen, um über zwei Klassen hinaus zu klassifizieren. Die deutlich längere Ausführungszeit stammt aus der höheren Komplexität der Transformer-Modelle.

Zur **Stance detection** scheinen LSTM-Modelle wie **ULMFit** weniger gut geeignet, hier werden die besten Ergebnisse durch Einsatz des **RoBERTa**-Modells mit **SimpleTransformers** erzielt, wie aus Abbildung 4.2 hervorgeht. Aus den Graphen kann gefolgert werden, dass die mittels **SimpleTransformers** abgestimmten Modelle sehr effizient sind. Die Bibliothek ist offensichtlich gut anzuwenden und bringt sehr gute Ergebnisse.

Wie schon erwähnt hat die Qualität der und die Quantität in den Datensätzen eine hohe Auswirkung auf das Ergebnis, dennoch kann gesagt werden, dass, wenn richtig konfiguriert, die ursprüngliche Transformer-Architektur der der XL-Transformer überlegen ist.

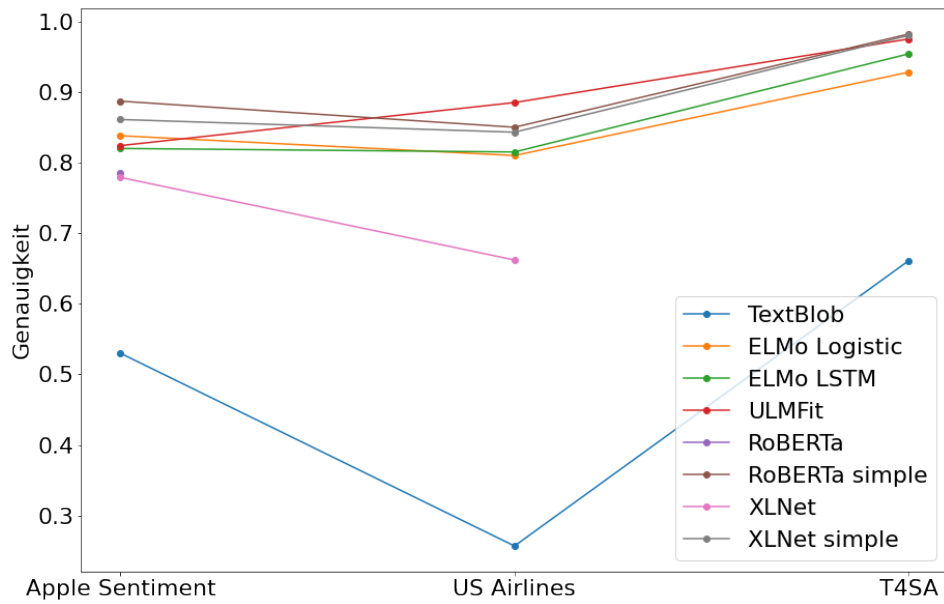


Abbildung 4.1: Genauigkeiten bei Sentiment Analysis

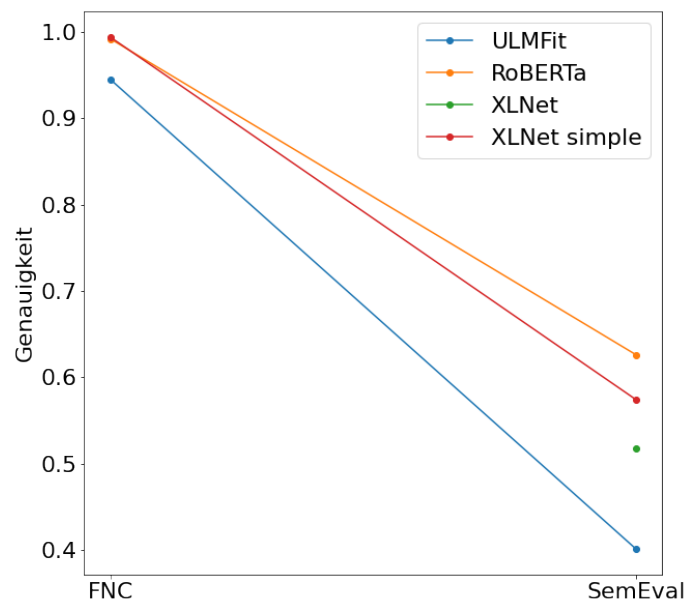


Abbildung 4.2: Genauigkeiten bei Stance Detection

4.2 Ausblick

In nachfolgenden Arbeiten kann die Trainingszeit und die Anzahl der getesteten Datensätze erhöht werden, da die meisten Modelle von mehr Trainingszeit profitieren können. Weiterhin

können auch bei der Textverarbeitung angebracht werden: Hierbei kann **Stemming** angewendet werden. Dadurch werden die Wörter in ihre Grundform umgewandelt mit vorausgehenden oder nachfolgenden Silben als separate Tokens. Aus dem Wort „playing“ werden so z.B. die zwei Tokens <play> und <ing>. Diese Art der **Tokenization** ist für den Einsatz mit Sprachmodellen am besten geeignet, da auch in diesen nicht alle Vokabeln einer Sprache enthalten sein können - das wäre schlicht ein zu großes Vokabular. Durch **Stemming** wird gewährleistet, dass die meisten Wörter, sowie die jeweilige Präfix und Suffix je einem Vektor zugeordnet werden können (sollten keine Rechtschreibfehler enthalten sein).

Da **NLP** ein sehr weit verbreitetes Thema ist, werden auch in den nächsten Jahren immer weitere Sprachmodelle veröffentlicht werden, die in den unterschiedlichen Aufgabenstellungen Stärken oder auch Schwächen haben werden und entsprechend getestet werden sollten.

Abbildungsverzeichnis

2.1	Aufbau des ELMo-Modells [6]	13
2.2	Vorverarbeitung der Wörter [6]	14
2.3	Aufbau eines Transformers [3]	16
2.4	Konzept eines autoregressiven Modells	18
2.5	Unterschied der Textgenerierung von BERT zu XLNet: Permutation [13]	24
3.1	Lernratenfinder	36
3.2	Steigendes Validation Loss	44
4.1	Genauigkeiten bei Sentiment Analysis	49
4.2	Genauigkeiten bei Stance Detection	49

Literaturverzeichnis

- [1] Howard, J., Ruder, S., Universal Language Model Fine-tuning for Text Classification. 2018, arXiv:1801.06146
- [2] Marev, K., Georgiev, K., Automated aviation occurrences categorization. 2019, <https://ieeexplore.ieee.org/document/8870055/>
- [3] Vaswani, A. et al., Attention Is All You Need. 2017, arXiv:1706.03762v5
- [4] Dai, Z., Yang, Z., Yang, Y., Carbonell, J., Le, Q., Salakhutdinov, R., Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context. 2019, arXiv:1901.02860v3
- [5] Peters, M., Neumann, M., Iyyer, M., Gardner, M., Clark, C. , Lee, K., Zettlemoyer, L., Deep contextualized word representations. 2018, arXiv:1802.05365v2
- [6] Mihail, E., Deep Contextualized Word Representations with ELMo. 2018, <https://www.mihaileric.com/posts/deep-contextualized-word-representations-elmo/>
- [7] Devlin, J., Chang, M., Lee K. , Toutanova, K., BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. 2019, arXiv:1810.04805v2
- [8] Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., Stoyanov, V., RoBERTa: A Robustly Optimized BERT Pretraining Approach. 2019, arXiv:1907.11692
- [9] Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I., Improving language understanding by generative pre-training. 2018, <https://openai.com/blog/language-unsupervised/>
- [10] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., Language Models are Unsupervised Multitask Learners. 2018, <https://openai.com/blog/better-language-models/>
- [11] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., Amodei, D., Language Models are Few-Shot Learners. 2020, arXiv:2005.14165

- [12] Yang, Z., Dai, Z., Yang, Y., Carbonell, J., Salakhutdinov, R., Le, Q., XLNet: Generalized Autoregressive Pretraining for Language Understanding. 2019, arXiv:1906.08237v2
- [13] Kurita, K., XLNet Explained. 2019, <https://mlexplained.com/2019/06/30/paper-dissected-xl-net-generalized-autoregressive-pretraining-for-language-understanding-explained>
- [14] Vadicamo, L., Carrara, F., Cimino, A., Cresci, S., Dell’Orletta, F., Falchi, F., Tesconi, M., Cross-Media Learning for Image Sentiment Analysis in the Wild. 2017, <http://www.t4sa.it/>
- [15] Go, A., Bhayani, R., Huang, L., Twitter Sentiment Classification using Distant Supervision. 2009
- [16] Wood, T., What is the F-score?. <https://deeptai.org/machine-learning-glossary-and-terms/f-score>
- [17] Ruder, S., NLP’s ImageNet moment has arrived. 2018, <https://ruder.io/nlp-imagenet/>
- [18] Crowdfunder, Apple Twitter Sentiment. 2016, <https://data.world/crowdfunder/apple-twitter-sentiment>
- [19] So twitterst du. <https://help.twitter.com/de/using-twitter/how-to-tweet>
- [20] <https://keras.io/>
- [21] <https://www.tensorflow.org/>
- [22] <https://colab.research.google.com/notebooks/intro.ipynb>
- [23] <https://research.google.com/colaboratory/faq.html>
- [24] <http://help.sentiment140.com/for-students>
- [25] <https://www.kaggle.com/seriousran/appletwittersentimenttexts>
- [26] <https://www.kaggle.com/crowdfunder/twitter-airline-sentiment>
- [27] <http://www.fakenewschallenge.org/>
- [28] <http://alt.qcri.org/semeval2016/task6/>
- [29] <https://simpletransformers.ai>