Universität der Bundeswehr München

Fakultät für Elektrotechnik und Technische Informatik

Projektarbeit im Studiengang Technische Informatik und Kommunikationstechnik

Analyse von Konzepten zur Erstellung Transformer-basierter Sprachmodelle am Beispiel von BERT und ALBERT

Autor: Nicolas Bockmühl

Prüfer: Prof. Dr. Norbert Oswald

Abgabe: 13.03.2021

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

In den letzten Jahren ist es im Bereich der Neuro-Linguistischen-Programmierung zu sehr großen Fortschritten gekommen. Bestimmte Architekturen wie beispielsweise die Transformer-Architektur haben dabei geholfen den Fortschritt so schnell voranzutreiben. Wie in [8] beschrieben benutzt die Transformer-Architektur einen *Attention*-Mechanismus, welche diese Architektur besonders auszeichnet. Durch diesen Mechanismus ist es möglich den Kontext von einzelnen Wörtern zu anderen Wörtern in einem Satz zu verstehen und darzustellen. Dadurch ist eine weitaus effizientere Verarbeitung von Sätzen möglich.
Aufbauend auf dieser im Jahr 2018 entwickelten Architektur und dem verwendeten *Attention*-Mechanismus wurden im Laufe der Zeit viele unterschiedliche Sprachmodelle entwickelt. Besonders bekannt sind unter anderem die Sprachmodelle der BERT-Familie. Das Auffallende an diesen Sprachmodellen ist, dass sie nicht direkt für eine bestimmten Aufgabe wie z.B. *Question-Answering* entwickelt wurden, sondern dass diese Sprachmodelle ein *Pre-training* durchlaufen, in welchem ihnen das Grundverständnis von Sprache beigebracht wird. Ziel hierbei ist es den Modellen beizubringen, wie sie die Sprache verstehen und weiterverarbeiten können. Nach dem *Pre-training* erfolgt dann ein *Fine-tuning*. Hierbei wird das Modell auf eine bestimmte Aufgabe spezialisiert. Der große Vorteil bei den Modellen mit *Pre-training* besteht darin, dass man ein Modell, welches dieses *Pre-training* durchlaufen hat, sehr leicht auf unterschiedliche Aufgaben spezialisieren kann. Den größten Trainingsaufwand muss man beim *Pre-training* betreiben, während das *Fine-tuning* nur noch einen sehr kleinen Aufwand darstellt [1].

1.2 Zielsetzung

Ziel dieser Arbeit ist es ein grundlegendes Verständnis darüber zu erlangen, wie die einzelnen Konzepte funktionieren, um ein Sprachmodell zu entwickeln. Besonderes Augenmerk wird hierbei zuerst auf die Transformer-Architektur, sowie den darin enthaltenen

Attention-Mechanismus gelegt. Aufbauend auf diesen Konzepten wird weiterführend das Sprachmodell BERT und dessen Funktionsweise analysiert. Des Weiteren und Hauptfokus liegt dann allerdings auf dem Sprachmodell ALBERT, welches eine Weiterentwicklung des Sprachmodells BERT ist [4]. Abschließend wird ein Vergleich der beiden Sprachmodelle BERT und ALBERT vorgenommen.

Kapitel 2

Grundkonzepte NLP

Im Bereich der Neuro-Linguistischen-Programmierung gibt es verschiedenste Grundkonzepte, welche hier zu Beginn genauer erläutert werden. Diese Konzepte bilden die Grundlage zur Verarbeitung von natürlicher Sprache durch einen Computer.
Das Hauptproblem eines Computers ist, dass er keine Sätze oder auch nur einzelne Wörter verstehen kann, weswegen mit folgenden Konzepten versucht wird die einzelnen Wörter eines Satzes in Zahlenfolgen umzuwandeln. Wenn die einzelnen Wörter dann in Folgen aus Zahlen umgewandelt wurden kann ein Computer mit diesen umgewandelten Wörtern weiterarbeiten.

2.1 *Tokenization*

Unter dem Begriff *Tokenization* versteht man den Prozess, dass ein Eingabetext in eine Folge von einzelnen elementaren Einheiten zerlegt wird [3]. Diese elementaren Einheiten werden als Token bezeichnet und durch eine Zahl und nicht mehr durch eine Zeichenfolge repräsentiert. Der Vorteil dieses Prozesses ist, dass der Computer mit diesen Token, welche durch eine Zahl repräsentiert werden, besser arbeiten kann als direkt mit den einzelnen Zeichenfolgen des Eingabetexts.
Betrachten wir das Beispiel in Abbildung 2.1. Hier ist zu erkennen wie der Satz durch den Prozess der *Tokenization* in die einzelnen Zeichenfolgen zerlegt wird. Diese einzelnen Folgen von Zeichen können durch ein Token, welchem eine eindeutige Zahl zugeordnet ist, repräsentiert werden. Kommt in einem Text ein Wort mehrmals vor kann es durch dasselbe Token repräsentiert werden. Auch bei weiteren späteren Eingaben kann ein Wort, welchem bereits ein Token zugewiesen wurde, das schon vorhandene Token erneut nutzen anstatt ein neues anzulegen.

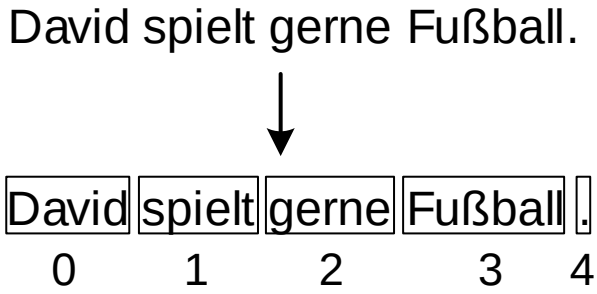


Abbildung 2.1: Beispiel Tokenization

In dem Beispiel in Abbildung 2.1 wurde ein sehr einfaches Konzept gewählt, wie man den Eingabetext zerlegt. Hier ist bei jedem Leerzeichen oder Satzzeichen eine Trennung eingefügt worden, was bewirkt, dass die Token jeweils genau ein Wort oder ein Satzzeichen abbilden.
Hier stellt sich allerdings die Frage, ob dies der richtige Ansatz ist oder ob es bessere Konzepte zur Satzzerlegung gibt[5]. Betrachtet wird hierzu beispielsweise das Wort „don't". Mit dem Ansatz nach jedem Leerzeichen oder Satzzeichen eine Trennung einzufügen würde aus dieser Zeichenfolge ein Token erstellt werden. Eine andere Möglichkeit wäre allerdings dieses Wort in die zwei Token „do" und „n't" zu zerlegen. Dieses Problem der korrekten Zerlegung wird noch größer, wenn man Namen, Adressen oder Wörter mit Bindestrichen etc. betrachtet.
Ein weiterer Ansatz besteht darin, ein Verb in seine Grundform und die angehängte Endung zu zerlegen [5]. Zum Beispiel ist es dadurch möglich das Wort „playing" in ein Token mit dem Inhalt „play" und einem zweiten Token mit der Endung „ing" zu zerlegen. Hierdurch kann man das erste Token auch auf andere Formen des Wortes „play" anwenden. Allerdings wird hier die Realisierung komplexer, da zuerst die Grundform eines Verbs und die angehängte Endung korrekt getrennt werden müssen.
An den vorausgehenden Überlegungen ist zu erkennen, dass es viele verschiedene Möglichkeiten gibt, wie man eine Zeichenfolge in einzelne Token zerlegen kann. Oft ist es hierbei zielführend, wenn vorher bekannt ist in welcher Sprache sich diese Zeichenfolgen befinden und welche weiteren Besonderheiten sie aufweisen. Außerdem sollte beachten werden, welches Konzept der verwendete *Tokenizer* verfolgt und wie die erstellten Token zur Weiterverarbeitung genutzt werden sollen.
In Kapitel 4 wird der *Tokenizer* den das Sprachmodell BERT verwendet genauer erläutert.

2.2 Wortvektoren

Nachdem die Eingabesequenz erfolgreich in unterschiedliche Token zerlegt wurde stellt sich die Frage, wie man die Bedeutung eines Tokens am besten darstellen kann. Der einfachste Weg ist, wie im Beispiel in Abbildung 2.1 dargestellt, dass die Token aufsteigend durchnummeriert werden. Dadurch ist zwar eine eindeutige Identifikation der Token über die zugeordneten Zahlen möglich, allerdings sagen die Zahlen so gut wie nichts über die

Bedeutung oder den Inhalt des Tokens aus.
Ein weitaus besserer Ansatz besteht darin ein Token durch einen Vektor zu repräsentieren [3]. Hierdurch ist die Möglichkeit gegeben den einzelnen Token eine Bedeutung zu verleihen. Durch diese Darstellung ist es möglich Token mit einer ähnlichen Bedeutung auch mit ähnlichen Vektoren zu versehen während Token die keine Gemeinsamkeiten haben sehr unterschiedliche Vektoren aufweisen.
Ein Beispiel ist Abbildung 2.2 zu entnehmen. Zur Veranschaulichung wurde die Vektorgröße hier auf zwei Dimensionen heruntergebrochen. Typische Werte für die Größe eines solchen Wortvektors sind 512 Dimensionen [8]. In der abgebildeten Grafik ist gut zu erkennen, dass die Wörter „laufen“ und „rennen“ räumlich nah aneinanderliegen, da sich ihre Bedeutung ähnelt und sie in der gleichen Kategorie einzuordnen sind. Vergleicht man diese Wörter jedoch mit dem Wort „Haus“ ist zu erkennen, dass dieses Wort räumlich weiter entfernt einzuordnen ist, da die Bedeutung des Wortes „Haus“ sich sehr stark von der Bedeutung der anderen Wörter unterscheidet.

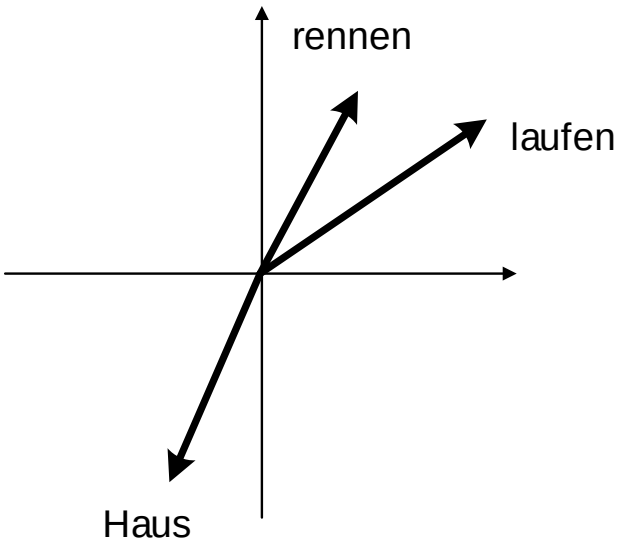


Abbildung 2.2: Wortvektoren Koordinatensystem

Durch die Darstellung der Wörter als Vektoren ergibt sich die Möglichkeit mit diesen Vektoren zu rechnen. Ein sehr bekanntes Beispiel aus [3] ist, dass der Vektor des Wortes „König“ ausgewählt wird und von diesem Vektor der Vektor des Wortes „Mann“ subtrahiert wird. Addiert man auf den resultierenden Vektor den Vektor des Wortes „Frau“ erhält man annähernd den Vektor des Wortes „Königin“. Dieses Berechnungsbeispiel verdeutlicht, dass in den Vektoren tatsächlich eine gewisse Bedeutung der einzelnen Wörter steckt. Allerdings lässt sich nicht nachvollziehen welche Zahlenwerte in dem Vektor welche Bedeutungsklassen repräsentieren. Es lässt sich nur über die Entfernung bestimmen, ob zwei Vektoren eine ähnliche Bedeutung haben oder ob sich die Bedeutung sehr stark unterscheidet [3].

Kapitel 3

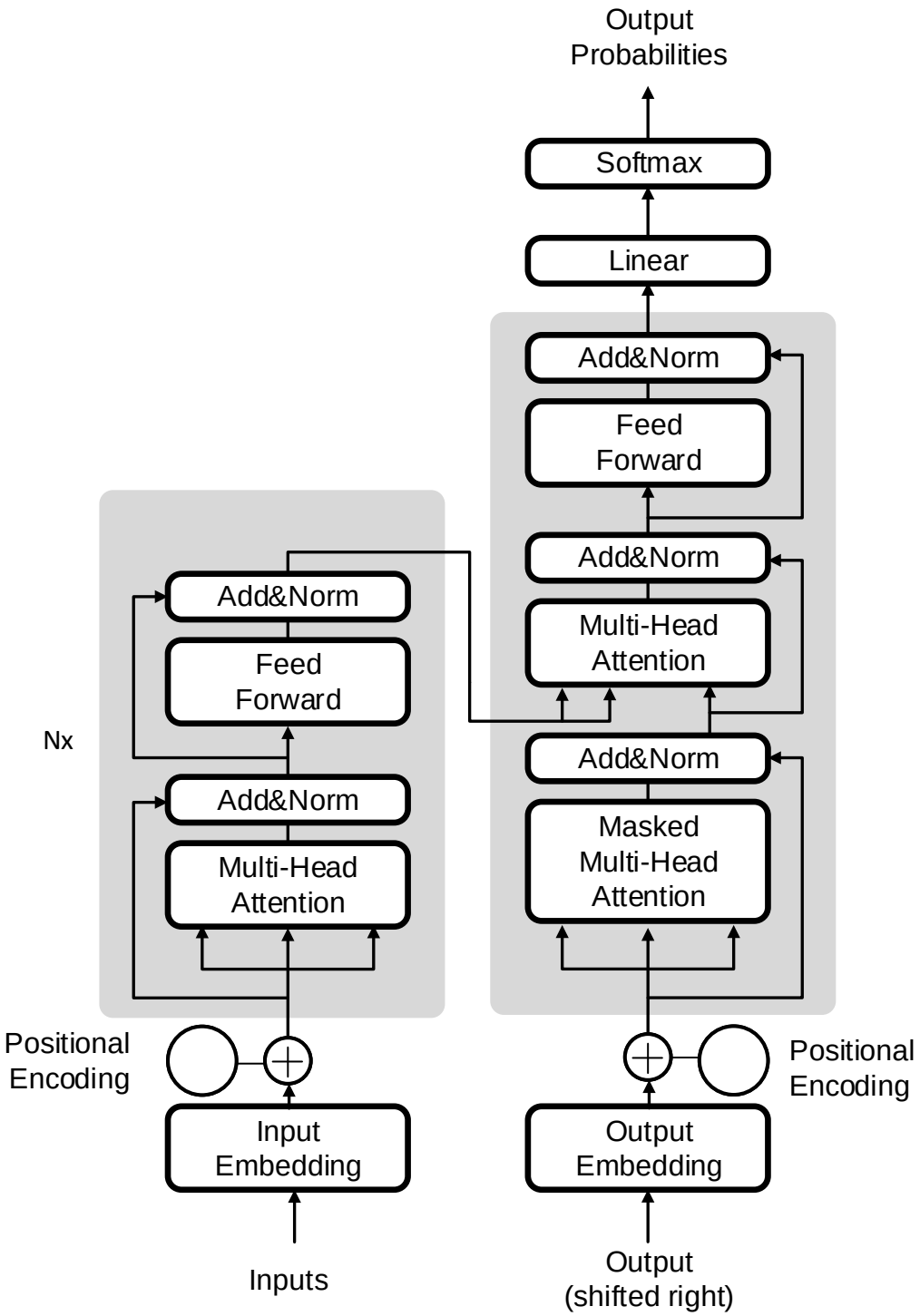
Transformers

Ein Transformer ist eine *Deep-Learning*-Architektur, welche eine Folge von Zeichen als Eingabesequenz bekommt und eine andere Folge von Zeichen als Ausgabesequenz zurückgibt [8]. Die Länge der einzelnen Sequenzen können von wenigen Wörtern bis hin zu ganzen Seiten voller Text variieren.
Oft werden Transformer-Architekturen für Übersetzungsvorgänge von einer Sprache in eine andere Sprache genutzt [8]. Besonders interessant ist jedoch der sogenannte *Encoder*-Teil der Transformer-Architektur, auf welchen wir später noch zu sprechen kommen, da dieser in den Sprachmodellen BERT [1] und ALBERT [4] verwendet wird.

3.1 Grundlegende Architektur

Wie in [8] beschrieben ist die Architektur eines Transformer in zwei Komponenten unterteilbar. Zum einen besteht die Architektur aus einem *Encoder*. Dieser nimmt die Eingabesequenz auf und verarbeitet diese weiter. Der Output des *Encoders* wird dann als Input in die zweite Komponente gegeben. Die zweite Komponente, genannt *Decoder*, verarbeitet den gegebenen Input und erzeugt damit den Output der gesamten Transformer-Architektur.
Wird der *Encoder* genauer unterteilt ist zu erkennen, dass diese Komponente aus sechs identischen kleineren *Encoder*-Komponenten besteht, welche einen großen *Encoder* bilden. Die einzelnen kleineren *Encoder* nehmen jeweils den Output der vorherigen Schicht als Input und geben selber den erzeugten Output an die nächste *Encoder*-Schicht weiter. Lediglich die erste *Encoder*-Schicht bekommt die tatsächliche Input-Sequenz als Input und lediglich die letzte *Encoder*-Schicht gibt ihren Output dann als Input an die *Decoder*-Komponente weiter.
Bei dem Decoder ist genau das gleiche Bild wie beim *Encoder* zu erkennen. Er besteht aus sechs einzelnen kleineren *Decodern* die auch jeweils ihren Input von der vorherigen *Decoder*-Schicht bekommen und ihren Output an die nächsthöhere *Decoder*-Schicht weitergeben.

Zusammenfassend ist zu sagen, dass die Architektur aus einem *Encoder* und einem *Decoder* besteht, wobei jede der beiden Komponenten aus sechs identischen Schichten bestehen. Der Input wird von der ersten Schicht des *Encoders* entgegengenommen, durchläuft dann jede Schicht, bis die Daten bei der letzten Schicht des *Decoders* als Output ausgegeben werden.
Werden die einzelnen *Encoder*-Schichten weiter aufgebrochen sind weitere Teilkomponenten zu erkennen aus denen eine *Encoder*-Schicht besteht. Zum einen aus einer *Attention*-Schicht und zum anderen aus einer *Feed-Forward*-Schicht. Auf jede der Schichten folgt zusätzlich eine *Add-and-Normalize*-Schicht [3]. Der Input in die *Encoder*-Schicht gelangt also zuerst in die *Attention*-Schicht. Das Konzept der *Attention* wird im Folgenden noch genauer erklärt. In der *Attention*-Schicht wird begutachtet wie stark die Wörter der Eingabesequenz im Bezug untereinanderstehen und welche Wörter der Eingabesequenz eine schwache oder starke Bindung haben [8] [2]. Dies geschieht mithilfe des *Attention*-Mechanismus. Der Output aus dieser Schicht passiert daraufhin die *Add-and-Normalize*-Schicht und wird als Input weiter an die *Feed-Forward*-Schicht gegeben. Nach dieser Schicht wird die zweite und letzte *Add-and-Normalize*-Schicht der gesamten *Encoder*-Schicht durchlaufen.
Wird der *Decoder* weiter in kleinere Teile aufgebrochen gibt sich wieder genau das gleiche Bild. Ein *Decoder* besteht ebenfalls aus einer *Attention*-Schicht und einer *Feed-Forward*-Schicht. Zusätzlich wird beim *Decoder* noch eine *Encoder-Decoder-Attention*-Schicht hinzugefügt. Auf jede der genannten Schichten folgt laut [8] wieder eine *Add-and-Normalize*-Schicht.



Quelle: [6]

Abbildung 3.1: Transformer-Architektur

3.2 *Attention*

Zu Beginn eines Durchlaufs werden die einzelnen Vektoren, welche die Wörter der Eingabesequenz repräsentieren, als eine Liste an die erste Schicht des *Encoders* übergeben. In dieser Schicht werden die Vektoren an die *Attention*-Schicht übergeben. In der *Attention*-Schicht läuft nun folgendes Prinzip ab. Diese Schicht betrachtet ein Wort in einem Satz und vollzieht nach auf welche anderen Wörter in dem Satz sich das aktuelle Wort bezieht. Infolgedessen wird der Vektor, welcher das Wort repräsentiert, angepasst [8]. Der angepasste Vektor bringt die Beziehung anderer Wörter in den Vektor des aktuellen Wortes ein, um ein besseres Verständnis des Eingabesatzes zu gewährleisten.

Betrachten wir den Beispielsatz in Abbildung 3.2. Als Eingabesequenz wurde hier der Satz „David spielt gerne Fußball, weil er Ballsportarten mag.“ aufgeführt. Nehmen wir als aktuelles Wort, welches betrachtet wird das Wort „er“ an. Für einen Menschen ist sehr einfach zu verstehen auf welche anderen Wörter sich das Wort „er“ bezieht. Für eine Maschine ist dies allerdings so gut wie unmöglich, weswegen hier die *Attention*-Schicht eingeführt wurde. Diese Schicht berechnet auf welche anderen Wörter sich das Wort „er“ bezieht und passt anschließen den Vektor, welcher das Wort „er“ darstellt an, um diese Beziehung zu anderen Wörtern darzustellen. In der Beispielsequenz wird also erkannt, dass sich das Wort „er“ eher auf das Wort „David“ und nicht auf das Wort „Fußball“ bezieht.

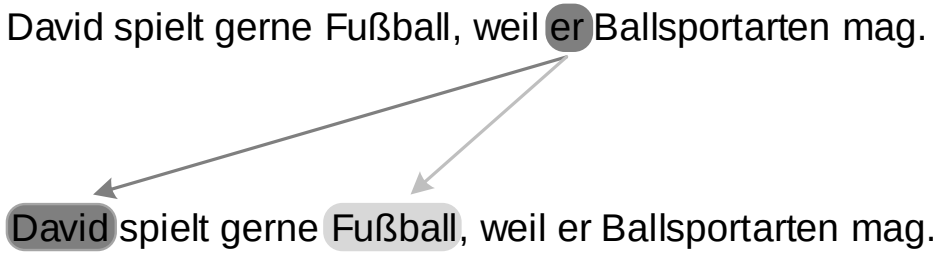


Abbildung 3.2: *Attention*-Beziehungen

Wie hier in der Abbildung zu erkennen probiert die Schicht letztendlich im Vektor des Wortes „er“ darzustellen, dass es mit dem Wort „David“ in Verbindung steht.
Um diese *Attention* zwischen Wörtern berechnen zu können, müssen aus jedem Wortvektor drei weitere Vektoren abgeleitet werden [2]. Ein *Query*-Vektor, ein *Key*-Vektor und ein *Value*-Vektor. Diese Vektoren werden erzeugt, indem man den Wortvektor mit drei trainierten Matrizen multipliziert, aus welchen sich dann die neuen Vektoren ergeben.
Der nächste Schritt der Berechnung besteht darin einen Score zu berechnen. Betrachten wir wie in dem Beispiel in Abbildung 3.2 das Wort „er“ und wollen dessen *Attention* berechnen. Dazu müssen wir das Wort „er“ gegen jedes andere Wort in diesem Satz prüfen. Die berechneten Scores mit den einzelnen Wörtern geben dann Aufschluss darüber wie viel Fokus man auf andere Teile dieses Satzes legen sollte. Diese Scores werden berechnet indem man das Skalarprodukt des *Query*-Vektors des ausgewählten Wortes (in diesem Fall „er“)

mit dem *Key*-Vektor des zu prüfenden Wortes berechnet.
Als nächstes wird der Score durch die Wurzel der Dimension des Vektors dividiert und anschließend an eine *Softmax()*-Funktion übergeben. Der berechnete *Softmax*-Wert drückt aus, wie viel jedes Wort das aktuelle Wort an dieser Position ausdrückt. Natürlich hat das Wort selbst an der eigenen Stelle den höchsten Wert, aber manchmal kann es nützlich sein sich auch um andere Stellen mit einem erhöhten Score zu kümmern, welche für das aktuelle Wort relevant sind.
Der vorletzte Schritt der Berechnung besteht darin das Skalarprodukt des Value-Vektors mit dem *Softmax*-Wert zu berechnen. Die Absicht hierbei ist, dass die Vektoren der Wörter auf die wir uns konzentrieren wollen intakt bleiben, während nicht relevante Wörter ausgelöscht werden, indem sich die Werte durch die Rechenoperation verkleinern.
Abschließend werden diese berechneten Vektoren aus dem vorletzten Schritt aufsummiert. Dieses Ergebnis ist der Output aus der *Attention*-Schicht an dieser Position für das aktuelle Wort. Diese Prozedur wird für jedes Wort der Eingabesequenz wiederholt.
Da die Berechnung mit einzelnen Vektoren zu aufwendig ist wird in der Implementierung direkt mit Matrizen gearbeitet, welche alle Vektoren einer Eingabesequenz auf einmal beinhalten. Außerdem können die oben genannten Schritte zur Errechnung der *Attention* in einer einzigen Formel verknüpft werden [8].

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{(d_k)}})V \tag{3.1}$$

Q = *Query*-Matrix
K = *Key*-Matrix
V = *Value*-Matrix
d_k = Dimension

3.3 Multi-Head-Attention

In der Transformer-Architektur wird allerdings keine einfache *Attention* benutzt, sondern sogenannte *Multi-Head-Attention* [8]. Dadurch wird der Output der *Attention*-Schicht verbessert.
Durch das verwenden mehrerer *Attention*-Strukturen, erweitert sich die Fähigkeit dieser Architektur sich auf verschiedene Positionen zu konzentrieren. Es gibt also in jeder eigenen *Attention*-Struktur unterschiedliche Darstellungsräume mit unterschiedlich trainierten *Key*, *Value*, *Query*-Matrizen. Jede Matrix kann sozusagen einen anderen Darstellungsraum bieten. Durch dieses Konzept der *Multi-Head-Attention* ist es möglich sich nicht nur auf einen Teil des Kontexts zu konzentriert, sondern man probiert mit mehreren Matrizen so viele Kontexte und Abhängigkeiten wie möglich zu erkennen.
Die Ergebnisse der einzelnen *Attention*-Strukturen werden daraufhin zu einem Output zusammengeführt, mit der Intention, dass in diesem zusammengeführten Output mehr Kontext enthalten ist, als in einer normalen *Attention*-Schicht. Die Zusammenführung der

einzelnen Outputs erfolgt durch aneinanderreihen und anschließende Multiplikation mit einer Additions-Gewichts-Matrix.

3.4 Positionsbezogene Vektoren

Ebenfalls ein wichtiges Kriterium ist, an welcher Stelle in einer Eingabesequenz sich ein Wort befindet. Das ist der Grund dafür, weshalb zu Beginn, noch bevor die Eingabesequenz an die erste *Encoder*-Schicht übergeben wird, ein Positionsvektor auf den Wortvektor addiert wird [8].
Die Positionsvektoren folgen einem bestimmten Muster, welche das Modell leicht erlernen kann, um die absolute und relative Position der Wörter voneinander zu erkennen. Der Grund dieser Positionscodierung ist, dass vielsagende Distanzen von Wörtern in die Berechnungen mit einfließen können.

Kapitel 4

BERT

4.1 Einführung

BERT ist ein Sprachmodell, welches auf *Natural Language Processing* und neuronalen Netzen basiert [1]. Der Name ist eine Abkürzung und steht für „Bidirectional Encoder Representations from Transformers“. Wie sich aus diesen Namen heraus schon ableiten lässt benutzt BERT einen Teil der weiter oben erläuterten Transformer-Architektur. Jedoch nicht das ganze Modell wird verwendet, sondern lediglich der *Encoder*-Teil.
Bert benutzt im *Pre-training* sowohl *Masked Language Modelling* (MLM) als auch *Next Sentence Prediction* (NSP) [1]. Dies führt dazu, dass das Modell nicht einfach auswendig lernt, sondern versucht die Sprache und den Kontext zu verstehen. Da diese zwei Aufgaben nicht ausreichen um ein fertiges Sprachmodell zu trainieren kann das Modell nach Abschluss des *Pre-trainings* auf eine bestimmte beliebige Aufgabe mit geringem Aufwand spezialisiert werden. Der Prozess des Anpassens des Modells auf eine bestimmte Aufgabe wird *Fine-tuning* genannt [1]. Um das Modell auf eine bestimmte Aufgabe anzupassen wird lediglich eine weitere Output-Schicht benötigt.

4.2 Architektur

Das Sprachmodell BERT besteht aus mehreren übereinandergestapelten Transformers-*Encoder*-Schichten [1]. Genau wie bei der Transformers-Architektur bekommt BERT ebenfalls eine Zeichensequenz als Input übergeben. Diese Zeichensequenz wird durch die verschiedenen *Encoder*-Schichten des Modells nach oben durchgereicht. Jede *Encoder*-Schicht verwendet, wie oben erklärt, eine *Attention*-Schicht mit anschließender *Add-and-Normalize*-Schicht und schiebt diese Ergebnisse dann durch eine *Feed-Forward*-Schicht. Auf diese *Feed-Forward*-Schicht folgt eine weitere *Add-and-Normalize*-Schicht. Das Ergebnis wird anschließend an die nächste *Encoder*-Schicht weitergegeben. Die letzte *Encoder*-Schicht gibt dann für jede Position der

Eingabesequenz einen Output-Vektor aus. Aufbauend auf diesen Vektoren können weitere *Fine-tuning*-Aufgaben realisiert werden.

4.3 *Tokenizer*

Zu Beginn wird das Vokabular mithilfe eines *Tokenizers* in die Token zerlegt. BERT verwendet hierfür einen *WordPiece-Tokenizer*. Für das englischsprachige Modell sind 30000 Token verwendet worden [7].
Bei einem *WordPiece-Tokenizer* kann eine Zeichenfolge in eine von zwei Formen zerlegt werden [7]. Entweder erfolgt eine *One-to-one tokenization* oder eine *one-to-many tokenization*. Bei der *One-to-one tokenization* wird die Zeichenfolge einfach durch ein Token repräsentiert in der die ganze Zeichenfolge selbst enthalten ist. Einfacher ausgedrückt befindet sich das ganze Wort in einem Token. Bei der *One-to-many tokenization* wird eine Zeichenfolge oder ein Wort in mehrere Token zerlegt. Hierfür wird der *greedy-longest-match*-Ansatz gewählt. Wird eine Zeichenfolge mit der *One-to-many tokenization* zerlegt wird das erste Token *Head* und die restlichen Token *Tail* genannt. *Tail*-Token erkennt man daran, dass vor der eigentlichen Zeichenfolge die Zeichen „##“ angefügt werden.
Wird mithilfe dieser Methode beispielsweise das Wort „playing“ zerlegt erhält man als *Head* das Token „play“ und als *Tail* das Token „##ing“.

4.4 *Pre-training*

Zu Beginn muss das Sprachmodell mit den zwei im Folgenden erklärten *Pre-training*-Aufgaben trainiert werden, damit es ein grundlegendes Verständnis von der zu erlernenden Sprache bekommt. Der Kontext der einzelnen Wörter muss verstanden werden genauso wie die Beziehung einzelner Sätze untereinander. Im Nachhinein kann aufbauend darauf durch *Fine-tuning* dem Model eine spezifische Aufgabe beigebracht werden. Die folgenden Erläuterungen stammen aus den Ausführungen in [1].

4.4.1 *Masked Language Modeling*

Die erste Aufgabe die BERT beim *Pre-training* absolviert wird *Masked Language Modelling* (MLM) genannt. Hierbei werden 15% der Eingabesequenz durch ein sogenanntes [*MASK*]-Token ersetzt. Im Training versucht BERT die [*MASK*]-Token aus dem Kontext heraus zu erraten. Der Vorteil dieser Aufgabe besteht darin, dass sowohl der rechte als auch der linke Kontext um das [*MASK*]-Token betrachtet werden können.
Tatsächlich wird das [*MASK*]-Token nicht in 100% der Fälle an die entsprechenden Stellen eingefügt, sondern nur mit einer Wahrscheinlichkeit von 80%. In 10% der Fälle wird einfach ein zufälliges Token eingefügt und in weiteren 10% der Fälle das Original-Token beibehalten. Grund hierfür ist, dass das [*MASK*]-Token nur im *Pre-training* und nicht mehr

im *Fine-tuning* auftritt [1]. Durch diesen Schritt wird versucht die Nichtübereinstimmung zwischen *Pre-training* und *Fine-tuning* so gering wie möglich zu halten.

4.4.2 *Next Sentence Prediction*

Viele wichtige *Fine-tuning*-Aufgaben wie beispielsweise *Natural Language Inference* oder *Question Answering* können später mit einem BERT-Sprachmodell realisiert werden. Bei der *Natural Language Inference*-Aufabe versucht das Modell zu beantworten, ob eine gegebene Aussage wahr oder falsch ist. Bei der *Question Answering*-Aufgabe bekommt das Modell eine Frage und eine Textpassage gegeben und versucht aus der Textpassage die Antwort zu extrahieren. Diese Aufgaben basieren darauf, dass das Modell den Zusammenhang zwischen Sätzen versteht, was allerdings nicht direkt durch *Masked Language Modeling* trainiert wird. Aus diesem Grund wird eine weitere *Pre-training* Aufgabe hinzugenommen. Mithilfe der *Next Sentence Prediction* (NSP) erlangt das Modell Verständnis zwischen der Beziehung von unterschiedlichen Sätzen. Diese *Pre-training* Aufgabe ist sehr einfach zu realisieren und kann aus jedem normalen Textkorpus einfach erstellt werden.
Bei der Aufgabe, welche in [1] beschrieben ist, muss das Modell vorhersagen, ob zwei Sätze, welche als Input übergeben wurden, tatsächlich die aufeinanderfolgenden Sätze sind oder ob die Reihenfolge der Sätze nicht korrekt ist. Dementsprechend gibt das Modell entweder wahr oder falsch zurück. In 50% der Fälle ist ein Satz B, welcher auf einen Satz A folgt tatsächlich der darauffolgende Satz. In 50% der Fälle ist B ein zufälliger Satz und nicht der Satz der auf A folgt. Das Modell sagt dann vorher, ob der Satz B auf den Satz A folgt oder nicht.

4.5 *Fine-tuning*

Nachdem das *Pre-training* abgeschlossen wurde ist das Grundgerüst des Modells fertig. Die nächste Aufgabe besteht darin das Modell auf eine gewünschte spezielle Aufgabe anzupassen. Diesen Prozess der Anpassung des Sprachmodells auf eine spezielle Aufgabe bezeichnet man als *Fine-tuning*. Durch den *Attention*-Mechanismus des Transformers ist es möglich sowohl Aufgaben zu erlernen, welche sich auf das Verständnis von einfachem klaren Text konzentrieren oder auch Aufgaben in welchen mit Textpaaren gearbeitet wird.
Um mit dem Sprachmodell *Fine-tuning* zu betreiben werden die einzelnen Gewichte mit den Werten aus dem abgeschlossen *Pre-training* initialisiert [1]. Des Weiteren wird jetzt eine bestimmte Aufgabe wie beispielsweise Satzklassifikation mit einem passenden gelabelten Datensatz dazu trainiert. Bei der Satzklassifikation muss das Modell eine Eingabesequenz einer bestimmten Klasse zuordnen. Gelabelt bedeutet hier, dass zu jeder Textphrase ebenfalls ein Wert gegeben ist wie der Satz korrekt zu klassifizieren wäre. Das ist hier essentiell, da das Modell nach seiner Vorhersage vergleichen muss, ob es richtig oder falsch liegt. Je nachdem ob die Vorhersage wahr oder falsch ist, werden die Parameter des Modells angepasst. Nach einer vergleichsweise geringen Trainingszeit im Vergleich zum *Pre-training* ist das Modell auf diese eine spezielle Aufgabe trainiert und kann verwendet werden.
Möchte man das Modell auf eine andere Aufgabe trainieren kann man einfach ein neues

Modell erstellen und dieses mit den Gewichten aus dem *Pre-training* initialisieren und den beschriebenen Prozess neu starten. Somit ist es möglich mit einem Modell, welches das *Pre-training* abgeschlossen hat, viele verschiedene spezielle Aufgaben zu realisieren.

4.6 BERT Parameter

Es gibt zwei verschiedene Ausführung von BERT. Zum einen $\text{BERT}_{\text{BASE}}$ und zum anderen $\text{BERT}_{\text{LARGE}}$ [1]. Es ist möglich bei den Parametern unterschiedlichste Einstellungen vorzunehmen, wenn man ein BERT-Modell erstellen will. Im Folgenden werden die Parameter aufgeführt, welche in [1] verwendet wurden.
Der erste Parameter ist die Anzahl der verwendeten Schichten und wird als **L** bezeichnet. $\text{BERT}_{\text{BASE}}$ verwendet 12 aufeinander aufbauende Schichten, während $\text{BERT}_{\text{LARGE}}$ 24 solcher Schichten verwendet.
Der zweite Parameter ist die Anzahl der Neuronen in den *Hidden*-Schichten, welche mit **H** abgekürzt wird. $\text{BERT}_{\text{BASE}}$ verwendet eine *Hidden*-Größe von 768, während $\text{BERT}_{\text{LARGE}}$ eine *Hidden*-Größe von 1024 verwendet.
Der letzte Parameter ist die Anzahl der *Attention-Heads* bezeichnet als **A**. $\text{BERT}_{\text{BASE}}$ verwendet 12 *Attention-Heads* und $\text{BERT}_{\text{LARGE}}$ verwendet 16 *Attention-Heads*.
Insgesamt kommt $\text{BERT}_{\text{BASE}}$ damit auf eine Parameteranzahl von 110 Millionen Parametern und $\text{BERT}_{\text{LARGE}}$ auf eine Anzahl von 340 Millionen Parametern.

	$\text{BERT}_{\text{BASE}}$	$\text{BERT}_{\text{LARGE}}$
Schichten **L**	12	24
Hidden-Größe **H**	768	1024
Attention-Heads **A**	12	16
Gesamzanzahl Parameter	110 Millionen	340 Millionen

Tabelle 4.1: Parameter BERT

Die Parameterwerte für $\text{BERT}_{\text{BASE}}$ wurden genauso gewählt, um es mit dem Sprachmodell GPT von OpenAI vergleichbar zu machen [1].

4.6.1 Einfluss der Parameteranzahl

Abschließend zu BERT soll betrachtet werden, ob die Größe des Modells einen Einfluss auf dessen Performance hat. Dazu werden verschiedene Sprachmodelle mit unterschiedlichen Parametergrößen erstellt. Die Trainingsprozedur auf allen Sprachmodellen bleibt jedoch gleich, um eine Vergleichbarkeit der verschiedenen Modelle zu schaffen.
Zum Vergleich der Modelle wurden Aufgaben der *General Language Understanding Evaluation benchmark* (GLUE) herangezogen. Hierbei werden die Modelle nach dem *Fine-tuning* mit einem Score bewertet. Je höher der Wert ist, desto besseres ist das Sprachmodell in der Abarbeitung dieser bestimmten Aufgabe des GLUE. Zur Beurteilung

der Modelle wurden vier unterschiedliche Aufgaben aus dem GLUE ausgewählt und die Sprachmodelle gegen diese Aufgaben geprüft. Die genauen Beschreibungen der verwendeten GLUE-Aufgaben sind in 8.2 zu finden. Das Ergebnis der Auswertung ist in Tabelle 4.2 zu sehen.

Schichten	*Hidden*-Größe	*Attention-Heads*	MNLI-m	MRPC	SST-2
3	768	12	77.9	79.8	88.4
6	768	3	80.6	82.2	90.7
6	768	12	81.9	84.8	91.3
12	768	12	84.4	86.7	92.9
12	1024	16	85.7	86.9	93.3
24	1024	16	86.6	87.8	93.7

Quelle: [1]

Tabelle 4.2: Vergleich Parameteranzahl BERT

In Tabelle 4.2 ist zu erkennen, dass die Bewertungen der einzelnen Aufgaben immer besser ausfallen, je mehr Parameter im Modell benutzt werden. Das bedeutet also, dass man immer bessere Ergebnisse erhält, indem man die Parameteranzahl des Sprachmodells erhöht.

Kapitel 5

ALBERT

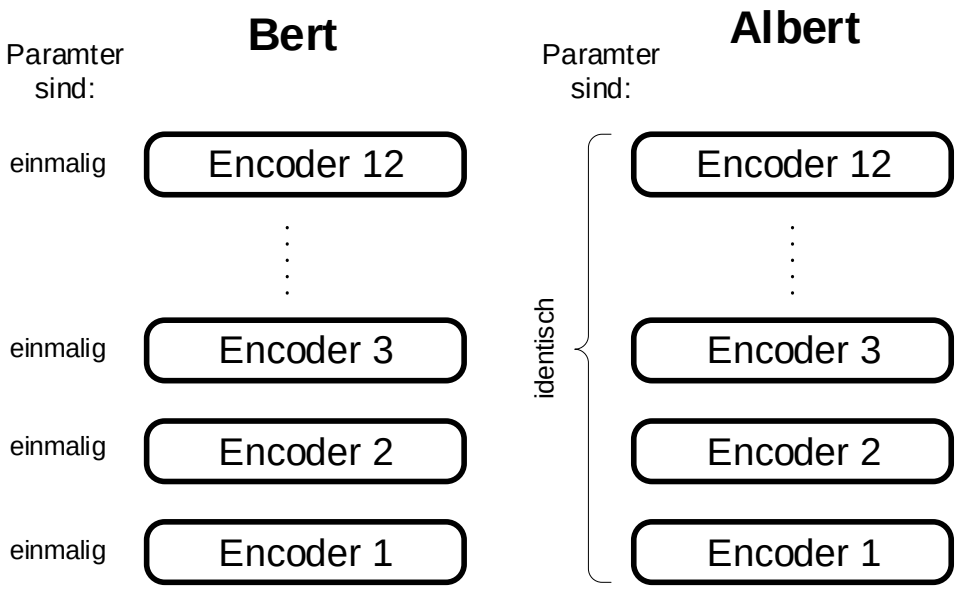
ALBERT ist ebenfalls wie BERT ein Sprachmodell. Genauer gesagt ist ALBERT eine Weiterentwicklung von BERT. Die Abkürzung ALBERT steht für „A Lite Bidirectional Encoder Representations from Transformers“ [4]. Ziel der Entwicklung von ALBERT ist es die sehr große Anzahl an Parametern im Sprachmodell zu verringern, um so den benötigten Speicherplatz und die benötigte Trainingszeit zu verkleinern.

5.1 Parameter-Reduktions-Techniken

Das Modell verwendet zwei unterschiedliche Techniken, um die sehr hohe Anzahl an Parametern, wie sie im Modell BERT vorhanden sind, zu verringern. Die im Folgenden dargestellten Änderungen wurden in [4] beschrieben.

5.1.1 *Cross-Layer Parameter Sharing*

Die erste der Reduktionstechniken wird „Cross-Layer Parameter Sharing“ genannt. BERT besteht aus 12 Transformer-*Encoder*-Blöcken. Jeder dieser Blöcke hat seine eigenen Parameter, welche alle für sich alleine trainiert werden. Bei ALBERT wurde eine entsprechende Reduktion vorgenommen, indem die Architektur soweit verändert wurde, dass nur noch die Parameter des ersten Transformer-*Encoder*-Blocks trainiert werden. Die Parameter dieser ersten Schicht werden dann für die übrigen elf Schichten übernommen, sodass diese Schichten keine eigenen Parameter trainieren müssen. Diese Technik führt zu einer erheblichen Parameter-Reduktion.



Quelle: [6]

Abbildung 5.1: Encoder-Schichten BERT und ALBERT

5.1.2 *Factorized Embedding Parameterization*

Im Sprachmodell BERT ist die *WordPiece*-Vektor-Größe E direkt verbunden mit der *Hidden*-Schicht-Größe H, was bedeutet, dass E=H ist.
Die kontextunabhängigen Repräsentationen von Wörtern werden in den *WordPiece*-Vektoren dargestellt, während die kontextabhängigen Repräsentationen von Wörtern in den *Hidden*-Schicht-Vektoren dargestellt werden. Da BERT den meisten Nutzen aus den kontextabhängigen Wörtern zieht, erscheint es sinnvoll die Größe von H im Vergleich zu E zu erhöhen (H > E). Da die Größen E und H allerdings direkt miteinander verknüpft sind erhöht sich E automatisch auch immer, wenn man H erhöhen will. Aus diesem Grund erscheint es sinnvoll die Größen H und E voneinander loszulösen, was zu einer großen Parameterreduktion führen kann.
Im Normalfall bei BERT wird ein Vokabular der Größe V verwendet. Die *WordPiece*-Vektor-Matrix hat folglich die Größe V x E. Wird H vergrößern, erhöht sich auch E und damit die gesamte *WordPiece*-Vektor-Matrix.
Bei ALBERT gibt es jetzt die Neuerung, dass hier eine Zerlegung der Vektor-Parameter durchgeführt wird, wodurch die Anzahl der Parameter drastisch sinkt. Anstatt die *WordPiece*-Vektoren direkt abhängig zu machen von der *Hidden*-Schicht-Größe H, werden diese Vektoren zuerst in einen Raum geringerer Größe E abgebildet. Dieser Raum E wird dann in den *Hidden*-Raum H abgebildet. Durch dieses Verfahren verringert sich die Ordnung von O(V x H) zu O(V x E + E x H). Dieses Reduktionsverfahren ist sehr gewinnbringend, wenn H >> E ist.

5.2 *Self-Supervised-Loss-Funktion*

Eine weitere Änderung die bei ALBERT vorgenommen wurde ist, dass die Trainingsaufgabe „Next Sentence Prediction“ durch eine neue ersetzt wurde. Die neue Aufgabe wird „Sentence Order Prediction“ genannt. Ursprünglich wurde *Next Sentence Prediction* verwendet, um *Fine-tune*-Aufgaben, welche ein Verständnis der Beziehung zwischen verschiedenen Sätzen benötigen, besser umsetzen zu können. Der Grund das *Next Sentence Prediction* durch *Sentence Order Prediction* ersetzt wurde ist, dass die Arbeit von Yang u. a. [9] ergeben hat, dass *Next Sentence Prediction* unzuverlässig ist. Der Hauptgrund dafür ist der Schwierigkeitsgrad dieser Trainingsaufgabe, da sowohl *Topic-Prediction* als auch *Coherence-Prediction* in einer Trainingsaufgabe realisiert wurden [4].
Unter *Topic-Prediction* ist zu verstehen, dass ein Modell vorhersagen soll, ob zwei Sätze zu einer Kategorie gehören. Im Gegensatz dazu soll das Modell bei der *Coherence-Prediction* vorhersagen, ob die Beziehung zwischen zwei Sätzen sinnvoll und korrekt erscheint.
Aus diesem Grund wurde *Sentence Order Prediction* eingeführt. Hier wird die *Topic-Prediction* vermieden und es wird sich nur auf den Zusammenhang zwischen mehreren Sätzen mittels *Coherence-Prediction* konzentriert.
Die Aufgabe bei *Sentence Order Prediction* besteht darin, die korrekte Reihenfolge eines Satzpaares vorherzusagen. Bei dieser Aufgabe werden zwei Sätze A und B an das Modell übergeben. Das Modell soll „Wahr“ zurückgeben, wenn sich die Sätze in der korrekten Reihenfolge befinden. „Falsch“ soll das Modell dann zurückgeben, falls die Reihenfolge der Sätze vertauscht wurde. Das Besondere dieser Trainingsaufgabe ist, dass immer die gleichen Satzpaare verwendet werden. Es werden also nicht zwei zufällige Sätze verwendet, sondern tatsächlich Sätze die aufeinanderfolgen. Bei Trainingsdaten, welche „Falsch“ zurückliefern sollen, wird die Reihenfolge der Sätze einfach getauscht. Dadurch erreicht man, dass sich die Trainingsaufgabe nur noch mit *Coherence-Prediction* und nicht mehr mit *Topic-Prediction* beschäftigt. Diese Aufgabe führt letztendlich dazu, dass das Modell genaue detailgetreue Unterscheidungen zwischen Satzbeziehungen vornehmen kann.

Kapitel 6

Vergleich von BERT und ALBERT

6.1 Vergleich der Parameter

Durch die Anwendung der oben genannten Reduktionstechniken hat das Sprachmodell $\text{ALBERT}_{\text{LARGE}}$ bis zu 18 mal weniger Parameter als $\text{BERT}_{\text{LARGE}}$. Wie in Tabelle 6.1 zu sehen beträgt die Größe der verwenden Schichten bei $\text{ALBERT}_{\text{LARGE}}$ wie auch $\text{BERT}_{\text{LARGE}}$ 24. Die *Hidden*-Größe beträgt bei beiden Modellen 1024. Durch die *Factorized Embedding Parameterization*-Technik ist es bei ALBERT jetzt möglich die *Embedding*-Größe anders zu wählen als die *Hidden*-Größe. Aus diesem Grund besitzt $\text{ALBERT}_{\text{LARGE}}$ nur eine *Embedding*-Größe von 128 während $\text{BERT}_{\text{LARGE}}$ dieselbe Größe wie die *Hidden*-Größe von 1024 annehmen muss. Daraus resultiert, dass $\text{ALBERT}_{\text{LARGE}}$ eine Gesamtparameteranzahl von 18 Millionen besitzt während $\text{BERT}_{\text{LARGE}}$ 334 Millionen Parameter beinhaltet.

Model	Param.	Schichten	*Hidden*	*Embedding*	Param.-*Sharing*
$\text{BERT}_{\text{BASE}}$	108M	12	768	768	Falsch
$\text{BERT}_{\text{LARGE}}$	334M	24	1024	1024	Falsch
$\text{ALBERT}_{\text{BASE}}$	12M	12	768	128	Wahr
$\text{ALBERT}_{\text{LARGE}}$	18M	24	1024	128	Wahr
$\text{ALBERT}_{\text{xLARGE}}$	60M	24	2048	128	Wahr
$\text{ALBERT}_{\text{xxLARGE}}$	235M	12	4096	128	Wahr

Quelle: [4]

Tabelle 6.1: Vergleich Parameteranzahl BERT und ALBERT

Anhand Tabelle 6.1 ist zu erkennen, dass die Reduktionstechniken zu einer erheblichen Einsparung der Parameter geführt haben. Dadurch ist es möglich noch größere Modelle als $\text{ALBERT}_{\text{LARGE}}$ zu erstellen und die Parameteranzahl immer noch unter der Anzahl von $\text{BERT}_{\text{LARGE}}$ zu halten.

6.2 Allgemeiner Vergleich

Um die verschiedenen Modelle zu vergleichen wurden wieder die GLUE-Aufgaben herangezogen. Zusätzlich wurden auch noch Aufgaben des „Stanfort Question Answering Dataset“ (SQuAD) und Aufgaben der „ReAding Comprehension from Examinations“ (RACE) verwendet. Die genauen Beschreibungen der verwendeten Aufgaben sind in 8.2 zu finden.
Die Verbesserung der Parametereffizienz stellt den größten Vorteil von ALBERT da. Vergleicht man $\text{ALBERT}_{\text{xxLARGE}}$ mit $\text{BERT}_{\text{LARGE}}$ ist in Tabelle 8.1 zu erkennen, dass $\text{ALBERT}_{\text{xxLARGE}}$ circa 70% weniger Parameter verwendet und in jeder getesteten Aufgabe einen besseren Wert erzielt als $\text{BERT}_{\text{LARGE}}$.
Ein weiterer Punkt ist der Datendurchsatz während des Trainings, bezeichnet als Geschwindigkeit. $\text{ALBERT}_{\text{LARGE}}$ ist 1.7x schneller als $\text{BERT}_{\text{LARGE}}$. $\text{ALBERT}_{\text{xxLARGE}}$ braucht jedoch 3x so lange aufgrund der größeren Struktur.

6.3 Auswertung der *Factorized Embedding Parameterization*

In Tabelle 8.2 ist zu erkennen welchen Effekt es hat, wenn man die *Embedding*-Größe E verändert. Bei dem ALBERT-Modell, welches ohne Parameter-*Sharing* arbeitet ist zu erkennen, dass eine Erhöhung von E auch immer eine kleine Verbesserung des Gesamtdurchschnitts zur Folge hat. Bei dem ALBERT-Modell, mit Parameter-*Sharing* ist zu erkennen, dass eine gewählte Größe von 128 für E den besten Wert erzielt.

6.4 Auswertung der *Cross-Layer Parameter Sharing*

Im Folgenden wird betrachtet wie sich die Durchschnittswerte verändern, wenn man unterschiedliche Parameter-*Sharing*-Mechanismen verwendet. In Tabelle 8.3 ist zu erkennen, dass der Durchschnittswert am niedrigsten ist, wenn alle Parameter geteilt werden. Des Weiteren ist zu erkennen, dass der größte Werteverlust dadurch entsteht, dass die *Feed-Forward*-Schicht-Parameter geteilt werden. Ebenfalls ist zu erkennen, dass die Anzahl der Gesamtparameter weiter sinkt je mehr Parameter geteilt werden.

6.5 Auswertung der *Sentence Order Prediction*

In Tabelle 8.4 ist zu erkennen, dass das Modell welches mit NSP trainiert wurde die SOP-Aufgabe nur mit 52% lösen kann. Dies führt zu der Schlussfolgerung das durch NSP tatsächlich nur *Topic-Prediction* trainiert wurde. Im Vergleich dazu, durchläuft das Modell, welches mit SOP trainiert wurde die NSP-Aufgabe ziemlich gut mit 78.9%. Weiterführend ist zu erkennen, dass der Durchschnittswert bei dem Modell, welches mit SOP trainiert wurde am höchsten ist.

6.6 Auswertung des Zeitaufwandes

Normalerweise werden Sprachmodelle immer bis zu einem gewissen Datendurchsatz trainiert. Im Folgenden wurde allerdings eine andere Strategie gewählt. Die Modelle wurden für eine bestimmte Zeit (hier circa. 33 Stunden) trainiert. In Tabelle 8.5 ist zu erkennen, dass das ALBERT-Modell nach 32 Stunden schon einen besseren Durchschnittswert erzielt als das BERT-Modell, welches sogar 2 Stunden länger trainiert worden ist. Allerdings durchläuft ALBERT nur 125000 Trainingsschritte, während BERT 400000 Trainingsschritte durchläuft.

Kapitel 7

Fazit

Als erstes wurden in dieser Arbeit die Grundlagen erläutert, welche essenziell wichtig sind für die Erstellung eines Sprachmodells. Direkt zu Beginn in Kapitel 2.1 ist zu erkennen, dass es viele verschiedene Ansätze gibt ein Sprachmodell zu erstellen, was schon mit der Wahl eines passenden *Tokenizers* beginnt. Im Weiteren wurde in Kapitel 3 der sogenannte *Attention*-Mechanismus der Transformers-Architektur erläutert. Dieser Mechanismus hat zu vielen Verbesserungen im Bereich der Verarbeitung von natürlicher Sprache geführt [8].
Als nächstes wurde im Kapitel 4 das Sprachmodell BERT betrachtet, welches die *Encoder*-Struktur aus der Transformers-Architektur verwendet. Neuartig bei dieser Art von Modellen ist, dass es nicht direkt auf eine bestimmte Aufgabe trainiert wird, stattdessen durchläuft das Modell erst ein *Pre-training* und kann danach durch ein *Fine-tuning* auf viele unterschiedliche Aufgaben angepasst werden. Problem des Sprachmodells BERT ist das es aus bis zu 334 Millionen unterschiedlichen Parametern besteht. Aus diesem Grund wurde eine Weiterentwicklung von BERT genannt ALBERT entwickelt. Ziel dieser Entwicklung war es die sehr große Anzahl an Parametern zu verringern, um so die Trainingszeit und den Speicherplatz effizienter zu nutzen.
In Kapitel 5 wurden die zwei Mechanismen erläutert die dazu führen, dass die Parameteranzahl des Modells ALBERT erheblich niedriger ist als die Parameteranzahl seines Vorgängers BERT. Außerdem wurde bei dem Modell ALBERT eine neue *Self-Supervised-Loss*-Funktion eingeführt.
Im letzten Kapitel 6 wurden die beiden Sprachmodelle ALBERT und BERT mithilfe unterschiedlichster Aufgaben verglichen. Zu Beginn wurde hier die Gesamtparameteranzahl aufgeführt. Bei dem Modell ALBERT kommt es hier zu Einsparungen bis um die 70%. Des Weiteren wurde ausgewertet, welchen Einfluss die Parameter-Reduktions-Techniken auf die Durchschnittswerte des Sprachmodells haben. Die *Factorized Embedding Parameterization* führt dazu, dass das ALBERT-Modell bei welchem alle Parameter geteilt werden bei einer *Embedding*-Größe von 128 den besten Durchschnittswert erzielt. Aus diesem Grund wurde die *Embedding*-Größe von 128 als Standard gesetzt [4]. Das *Cross-Layer Parameter Sharing* führt dazu, dass die Durchschnittswerte sinken. Allerdings reduziert sich die Gesamtparameteranzahl durch diese Technik erheblich. Ebenfalls die neue

Self-Supervised-Loss-Funktion führt zu besseren Durchschnittswerten.
Als letztes wird in diesem Kapitel vergleichen, was passiert wenn die unterschiedlichen Sprachmodelle für eine gleichlange Zeit trainieren werden. Im Ergebnis dazu ist zu sehen, dass $\text{ALBERT}_{\text{xxLARGE}}$ einen bessern Durchschnittswert aufweist als $\text{BERT}_{\text{LARGE}}$, obwohl das BERT-Modell 275000 mehr Trainingsschritte durchführt.
Der nächste Schritt, um das Modell $\text{ALBERT}_{\text{xxLARGE}}$ weiterzuentwickeln ist die Trainingsgeschwindigkeit zu erhöhen. Im Moment kann $\text{ALBERT}_{\text{xxLARGE}}$ nur 125000 Trainingsschritte abarbeiten, während $\text{BERT}_{\text{LARGE}}$ 400000 Trainingsschritte schafft. Lässt sich der Durchsatz der Trainingsschritte erhöhen, würde man bei gleicher Rechenzeit ein viel besseres Sprachmodell mit einem noch besseren Durchschnittswert erhalten.

Kapitel 8

Anhang

8.1 Tabellen

	Modell-Typ	Parameter	SQuAD1.1	SQuAD2.0	MNLI	SST-2	RACE	Durchschnitt	Geschwindigkeit
BERT	base	108M	90.4/83.2	80.4/77.6	84.5	92.8	68.2	82.3	4.7x
	large	334M	92.2/85.5	5.0/82.2	86.6	93.0	73.9	85.2	1.0
ALBERT	base	12M	89.3/82.3	80.0/77.1	81.6	90.3	64.0	80.1	5.6x
	large	18M	90.6/83.9	82.3/79.4	83.5	91.7	68.5	82.4	1.7x
	xlarge	60M	92.5/86.1	86.1/83.1	86.4	92.4	74.8	85.5	0.6x
	xxlarge	235M	94.1/88.3	88.1/85.1	88.0	95.2	82.3	88.7	0.3x

Quelle: [4]

Tabelle 8.1: Allgemeiner Vergleich BERT und ALBERT

	Parameter-*Sharing*	E	Parameter	SQuAD1.1	SQuAD2.0	MNLI	SST-2	RACE	Durchschnitt
$\text{ALBERT}_{\text{BASE}}$	falsch	64	87M	89.9/82.9	80.1/77.8	82.9	91.5	66.7	81.3
	falsch	128	89M	89.9/82.8	80.3/77.3	83.7	91.5	67.9	81.7
	falsch	256	93M	90.2/83.2	80.3/77.4	84.1	91.9	67.3	81.8
	falsch	768	108M	90.4/83.2	80.4/77.6	84.5	92.8	68.2	82.3
	wahr	64	10M	88.7/81.4	77.5/74.8	80.0	89.4	63.5	79.0
	wahr	128	12M	89.3/82.3	80.0/77.1	81.6	90.3	64.0	80.1
	wahr	256	16M	88.8/81.5	79.1/76.3	81.5	90.3	63.4	79.6
	wahr	768	31M	88.6/81.5	79.2/76.6	82.0	90.6	63.3	79.8

Quelle: [4]

Tabelle 8.2: Effekt der *Embedding*-Größe

ALBERT	Parameter-*Sharing*	Parameter	SQuAD1.1	SQuAD2.0	MNLI	SST-2	RACE	Durchschnitt
E=768	alle	31M	88.6/81.5	79.2/76.6	82.0	90.6	63.3	79.8
	Attention	83M	89.9/82.7	80.0/77.2	84.0	91.4	67.7	81.6
	Feed-Forward	57M	89.2/82.1	78.2/75.4	81.5	90.8	62.6	79.5
	keine	108M	90.4/83.2	80.4/77.6	84.5	92.8	68.2	82.3
E=128	alle	12M	89.9/82.3	80.0/77.1	82.0	90.3	64.0	80.1
	Attention	64M	89.9/82.8	80.7/77.9	83.4	91.9	67.6	81.7
	Feed-Forward	38M	88.9/81.6	78.6/75.6	82.3	91.7	64.4	80.2
	keine	89M	89.9/82.8	80.3/77.3	83.2	91.5	67.9	81.6

Quelle: [4]

Tabelle 8.3: Effekt der *Cross-Layer Parameter Sharing*-Strategie

SP tasks	MLM	NSP	SOP	SQuAD1.1	SQuAD2.0	MNLI	SST-2	RACE	Durchschnitt
None	54.9	52.4	53.3	88.6/81.5	78.1/75.3	81.5	89.9	61.7	79.0
NSP	54.5	90.5	52.0	88.4/81.5	77.2/74.6	81.6	91.1	62.3	79.2
SOP	54.0	78.9	86.5	89.3/82.3	80.0/77.1	82.0	90.3	64.0	80.1

Quelle: [4]

Tabelle 8.4: Effekt der *Sentence Order Prediction*-Funktion

Modell	Schritte	Zeit	SQuAD1.1	SQuAD2.0	MNLI	SST-2	RACE	Durchschnitt
BERT-large	400k	34h	93.5/87.4	86.9/84.3	87.8	94.6	77.3	87.2
ALBERT-xxlarge	125k	32h	94.0/88.1	88.3/85.3	87.8	95.4	82.5	88.7

Quelle: [4]

Tabelle 8.5: Effekt der Trainingszeit

8.2 *Downstream*-Aufgaben

MNLI „Multi-Genre Natural Language Inference“ ist eine Klassifizierungsaufgabe. Dem Modell werden bei dieser Aufgabe zwei Sätze übergeben. Das Modell soll im Folgenden vorhersagen, ob der Satz B, welcher auf den Satz A folgt, eine Folge auf den Satz A ist oder ob Satz B ein Widerspruch auf Satz A ist oder ob Satz B sich neutral zu Satz A verhält [1].

MRPC „Microsoft Research Paraphrase Corpus“ ist eine Aufgabe bei der das Modell zwei Sätze, welche aus Online-Nachrichten extrahiert wurden, auf semantische Gleichheit überprüft [1].

SST-2 „Stanfort Sentiment Treebank“ ist eine Klassifizierungsaufgabe mit einem Satz. Die Sätze wurden aus Filmkritiken extrahiert. Die Aufgabe besteht darin, den Satz aufgrund seiner Semantik, einer Kategorie zuzuordnen [1].

SQuAD „Stanfort Question Answering Dataset“ ist ein Frage-Antwort Datensatz. Die Aufgabe besteht darin aus einem Paragraphen die konkrete Antwort anzugeben. Es wurden die Version 1.1 verwendet, welche aus 100000 Fragen besteht und die Version 2.0, welche zusätzlich 50000 unbeantwortbare Fragen beinhaltet.[4].

RACE „ReAding Comprehension from Examinations“ ist ein Datensatz, welcher mithilfe von Multiple-Choice-Fragen das Leseverständnis testen. Jede Instanz besteht aus vier Antwortmöglichkeiten. Als Input wird dem Modell die Passage, die Frage und die vier Antwortmöglichkeiten übergeben.[4].

Abbildungsverzeichnis

Literatur

[1] Jacob Devlin, Ming-Wei Chang, Kenton Lee und Kristina Toutanova. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL].

[2] Andrea Galassi, Marco Lippi und Paolo Torroni. "Attention in Natural Language Processing". In: *IEEE Transactions on Neural Networks and Learning Systems* (2020), S. 1–18. ISSN: 2162-2388. DOI: 10.1109/tnnls.2020.3019893. URL: http://dx.doi.org/10.1109/TNNLS.2020.3019893.

[3] Daniel Jurafsky und James H. Martin. *Speech and Language Processing*. 2020, S. 1–615.

[4] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma und Radu Soricut. *ALBERT: A Lite BERT for Self-supervised Learning of Language Representations*. 2020. arXiv: 1909.11942 [cs.CL].

[5] Christopher D. Manning, Prabhakar Raghavan und Hinrich Schütze. *An Introduction to Information Retrieval*. 2009, S. 1–544.

[6] Chris McCormick. *Should you switch from BERT to ALBERT?* Youtube. 25. Feb. 2021. URL: https://www.youtube.com/watch?v=vsGN8WqwvKg (besucht am 01. 03. 2021).

[7] Jannis Vamas. *BERT for NER*. 19. Juni 2019. URL: https://vamvas.ch/bert-for-ner (besucht am 01. 03. 2021).

[8] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser und Illia Polosukhin. *Attention Is All You Need*. 2017. arXiv: 1706.03762 [cs.CL].

[9] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov und Quoc V. Le. *XLNet: Generalized Autoregressive Pretraining for Language Understanding*. 2020. arXiv: 1906.08237 [cs.CL].