Informatik
Hauptcampus

H O C H
S C H U L E
T R I E R

Natural Language Processing und Text übersetzung

Natural Language Processing and Text Translation

Bachvarov, Vladislav

Master-Abschlussarbeit

Betreuer: Prof. Dr. Hans Beise

Trier, Abgabedatum

Vorwort

Ein Vorwort ist nicht unbedingt ntig. Falls Sie ein Vorwort schreiben, so ist dies der Platz, um z.B. die Firma vorzustellen, in der diese Arbeit entstanden ist, oder einigen Leuten zu danken, die in irgendeiner Form positiv zur Entstehung dieser Arbeit beigetragen haben. Auf keinen Fall sollten Sie im Vorwort die Aufgabenstellung nr erlern oder vertieft auf technische Sachverhalte eingehen.

Kurzfassung

In der Kurzfassung soll in kurzer und pranter Weise der wesentliche Inhalt der Arbeit beschrieben werden. Dazu zen vor allem eine kurze Aufgabenbeschreibung, der Lsungsansatz sowie die wesentlichen Ergebnisse der Arbeit. Ein higer Fehler fr die Kurzfassung ist, dass lediglich die Aufgabenbeschreibung (d.h. das Problem) in Kurzform vorgelegt wird. Die Kurzfassung soll aber die gesamte Arbeit widerspiegeln. Deshalb sind vor allem die erzielten Ergebnisse darzustellen. Die Kurzfassung soll etwa eine halbe bis ganze DIN-A4-Seite umfassen.

Hinweis: Schreiben Sie die Kurzfassung am Ende der Arbeit, denn eventuell ist Ihnen beim Schreiben erst vollends klar geworden, was das Wesentliche der Arbeit ist bzw. welche Schwerpunkte Sie bei der Arbeit gesetzt haben. Andernfalls laufen Sie Gefahr, dass die Kurzfassung nicht zum Rest der Arbeit passt.

The same in english.

Inhaltsverzeichnis

1

Einleitung und Problemstellung

Begonnen werden soll mit einer Einleitung zum Thema, also Hintergrund und Ziel erlert werden.

Weiterhin wird das vorliegende Problem diskutiert: Was ist zu lsen, warum ist es wichtig, dass man dieses Problem lst und welche Lsungsanse gibt es bereits. Der Bezug auf vorhandene oder eben bisher fehlende Lsungen begrndet auch die Intention und Bedeutung dieser Arbeit. Dies knnen allgemeine Gesichtspunkte sein: Man liefert einen Beitrag fr ein generelles Problem oder man hat eine spezielle Systemumgebung oder ein spezielles Produkt (z.B. in einem Unternehmen), woraus sich dieses noch zu lsende Problem ergibt.

Im weiteren Verlauf wird die Problemstellung konkret dargestellt: Was ist spezifisch zu lsen? Welche Randbedingungen sind gegeben und was ist die Zielsetzung? Letztere soll das beschreiben, was man mit dieser Arbeit (mindestens) erreichen mchte.

2

Word2Vec

In dieser Kapitel wird das Verfahren "Word2Vec"durch die Nutzung von Neuronalem Netz vorgestellt. Word2Vec ist eine Darstellung von Wörtern mit Vecoren, was auch aus die Abkürzung klar wird - Word ist klar; 2 - to; und Vec - Vector und das ganze "word to vector". Dieses Model ist am meisten in der Natural Laguage Processing (NLP) verbreitet und wird in vielen Bereichen der Informatik genutzt, unter anderem in Spamfilterung und Dokumentenanalyse. Jedoch diese Technik besagt nur wie die Wörter eines Textes dargestellen werden können. Das Verfahren, bei dem die möglichst passenden Vektoren in einem ausgewählten Text, auch Corpus genant, gelernt werden, heißt Word Embeddings. Bei dieser Technik wird ein Neuronales Netzt eigesetzt. Die Vorgehensweise und die Idee wird folglich erklärt.

2.1 Word Embedding

Wie es schon in der Einleitung erwähnt wurde, Word Embedding ist der Prozess, bei dem die Wörter eines Textes in mathematischen Vektoren gewandelt werden. Zuerst muss der Corpus vorbeiretet werden. Ich stelle hier nur die Theorie und in einer späteren Kapitel (!? WICHTIG WELCHE GENAU!?) gehe ich tiefer in dem Programmcode.

!! DAS HIER GEHÖRT IN EINE ANDERE KAPITEL !!! !!! DIE KAPITEL FÜR TEXTVORBEREITUNG ODER SOWAS!!!

Als der Text vorbeitet ist, sodass es von Sonderzeichen und alle unnötigen Zeichen bereinigt wird. Wenn der Text vorbeireitet ist, werden die Wörter aus dem Corpus bestimmt und jeder erhält einen Index. Üblicherweise werden die Wörter nach ihrer Häufigkeit angeordnet. Das häufigste Wort erhält somit den Index 1. Als nächstes werden die Wörter im Korpus durch ihren Index ersetzt, um alle Trainingspaare fürs Lernen generiert zu werden. Dies erfolgt in dem es durch das Corpus iteriert und in einem bestimmten Fenster, oder in der Literatur auch als Window gezeichnet, alle Contextwörter und den Targetwort ausgelesen werden. Das Targetwort ist das Wort in der Mitte, während die Wörter um das Targetwort entsprechend die Kontextwörter.

!!! BIS HIER MUSS WEG !!!!

In der Literatur werden zwei Arten von Wort2Vec Modelle - SKIP-gram und CBOW (Continous Bag of Words). Beide Modelle verwenden ein Neuronales Netz mit einem oder zwei versteckten Schichten (siehe !!KAPITEL MIT DEM PROGRAMMCODE!!). Die beiden Methoden unterscheiden sich nach ihren Ein- und Ausgaben.

SKIP-gram Model

Bei dem SKIP-gram-Model fließen die Targetwörter als Eingabe und das Model versucht ein Kontextwort zu raten. Hier ist die Struktur eines Skip-gram Models:

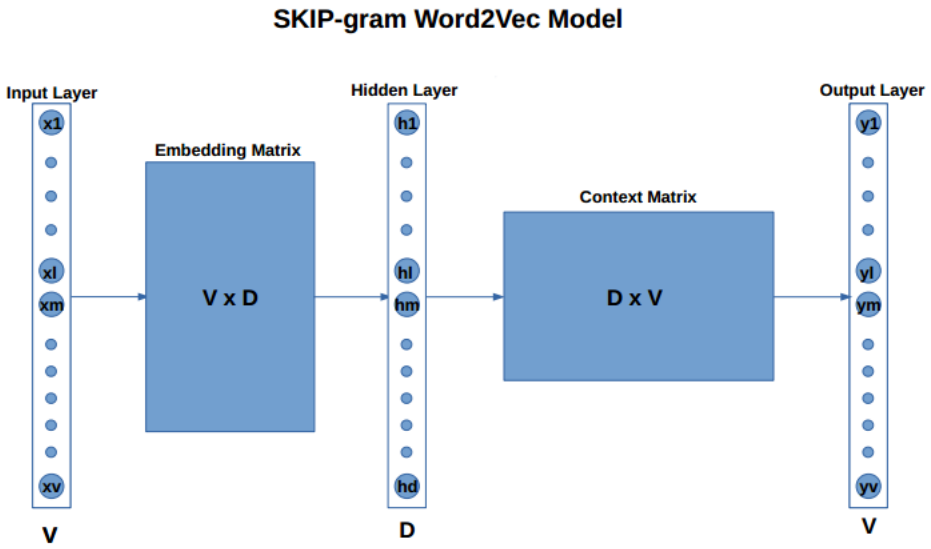

Abb. 2.1: Skip-gram Word2Vec Model

Aus der Abbildung 2.1 ist es zu entnehmen, dass ein Skip-gram Model aus einer hidden Schicht und zwei Eigabeschichten. Die Eingabe sowie die Ausgabe ist ein Vektor, der aus V Componente besteht. Das entspricht die größe des Wörterbuchs (engl. Vocabulary). Das versteckte Schicht besteht aus D Variablen und stellt einen Vektor dar. Die Dimensionalität dieses Vektors nimmt üblich einen Wert zwischen 25 und 300. Diese Variablen können auch als Eigenschaften für die Wörter betrachtet werden. Je mehr Kriterien es untersucht werden, desto besser die Beziehung zwischen Wörtern wiederspiegelt werden kann. Die Ausgabe ist wieder einen *V-dimensionalen* Vektor. Jedoch die Ausgabe ist kein One-Hot Vektor mehr, der das Kontextwort wiedergibt, sondern einen Wahrscheinlichkeitsvektor, dass der Wort mit der entsprechenden Index der richtige Contextwort ist.

Die zwei Matrizen sind identisch, jedoch die Kontextmatrize ist die transponierte Embeddingsmatrize. Diese Matrix beinhaltet unsere Wortvektoren.

Der Abbildung 2.1 nach besteht das neuronale Netz aus drei Schichten. Im Hiddenlayer steht ein Vektor, der abhängig von unsere Eingabe den Wortvektor repräsentiert. Die Ausgabe ist ein softmax

CBOW Model

Die Kontextwörter sind die Eingabe in dem CBOW-Model und das Model ratet der Targetwort. Die zwei Modelle besitzen die gleiche Anzahl an Schichten. Das CBOW-Model ist ein umgedrehtes SKIP-gram-Model, jedoch die Eingabe besteht aus w-Viele Vektoren statt nur eins. Als nächstes stellt die 2.2 Abbildung die beschriebene Struktur.

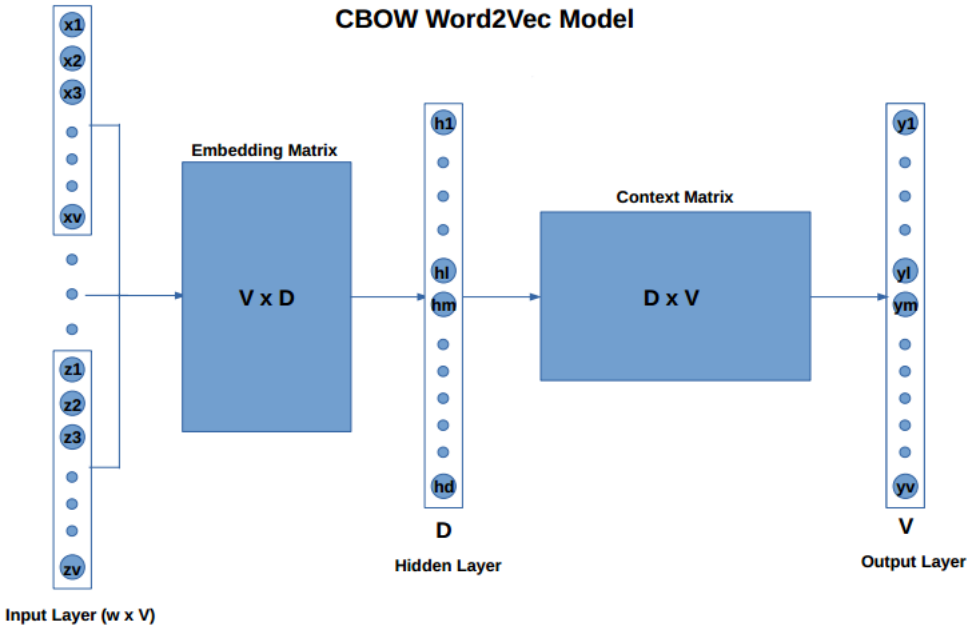

Abb. 2.2: CBOW Word2Vec Model

Die Struktur des CBOW-Models besteht wieder aus eine Matrix für das Embedding der Target- und Kontextwörter. Die größe der Matrizen hängt von der ausgewählten Hyperparameter und die Größe des Datensatzes. Die Anzahl der verwendeten Inputvektoren entspricht die größe des gesetzten Window (Kontextfenster).

2.1.1 Implementierung

3

Glove

3.1 The GloVe Method

Der Vorfall von Wörtern ist der Hauptquelle von Information für unsupervised Learning zum Lernen von Wortrepräsentation. Trotz der Zahlreiche Existenz von Algorithmen ist die Hauptfrage, wie man Sinngehalt aus den Statistiken hearuszieht und wie die resultierende Vectorrepräsentation von Wörter dieser Sinn spiegelt.

Wir führen einige Definitionen auf. Die Matrix der Word-Word-co-occurrence notieren wir mit X und ein Eintrag in der Matrix mit X_{ij}. Der Wert X_{ij} stellt dar, wie oft das Wort j in dem Kontext vom Wort i vorkommt. Weiterhin beschreibt die Einheit X_i die Anzahl des Vorkommens eines beliebigen Wortes in dem Kontext vom Wort i und ist in Gleichung 3.1 definiert:

$$X_i = \sum_k X_{ik}. \tag{3.1}$$

Schließlich definieren wir die Einheit P_{ij}, die die Wahrscheinlich bechreibt, dass ein Wort j in dem Kontext vom Wort i vorkommt. Die Formel ist in der Gleichung 3.2 aufgeführt:

$$P_{ij} = P(j|i) = \frac{X_{ij}}{X_i}. \tag{3.2}$$

Ein kleines Beispiel wird angegeben, damit es verstanden werden kann, wie bestimmte Aspekte aus der geimeinsammen Auftreten gewonnen werden können. Es werden zwei Wörter i und j betrachtet, die zur einer Menge gehören, für das Beispiel ist das der Thermodynamischenzustand. Nehmen wir die Wörter $i = ice$ ud $j = steam$. Die Beziehun der beiden Wörter kann so untersucht werden, indem ihre Relation mit anderen Probewörtern k berechnet wird. Für Wörter, die in direkter Bezug zu $i = ice$ stehen, erwarten wir, dass das Verhältnis $\frac{P_{ik}}{P_{jk}}$ groß ist. Zum Beispiel wird das Wort $k = solid$ betrachtet. Analog bei Wörter, die näher zu $j = steam$ sind, erhalten wir einen kleineren Wert des Bruchs $\frac{P_{ik}}{P_{jk}}$ - in diesem Fall nehmen wir das Wort $k = gas$. Selbstverständlich ist das Verhältnis des Bruchs bei Wörter die entweder zu beiden Wörtern i, j in Bezug stehen oder solchen zu keinem nah an eins. Die Tabelle stellt das Verhältnis zwischen den Wörtern dar.

Probability and Ration	k = solid	k = gas	k = water	k = fashion
P(k—ice)	1.9×10^{-4}	6.6×10^{-5}	3.0×10^{-3}	1.7×10^{-5}
P(k—steam)	2.2×10^{-5}	7.8×10^{-4}	2.2×10^{-3}	1.8×10^{-5}
P(k—ice)/P(k—steam)	8.9	8.5×10^{-2}	1.36	0.96

Tabelle 3.1: Tabelle von dem Vorkommen der beiden Wörter *ice* und *steam*

In der Tabelle 3.1 wird die Beziehung zwischen die Wörter *ice* und *steam* dargestellt. Die Erwartungen werden durch die Tabelle gerechtfertigt. Die Rate erlaubt uns besser die Verhältnisse zwischen die einzelnen Wörter zu verstehen. Mit Hilfer des Bruches werden besser unterschieden, wie die Wörter zueinander stehen, im Vergleich zu der einfachen Wahrscheinlichkeit.

Die oben genannten Argumente ergeben, dass der Anfang von Word-Vector-Learning mit der Rate des gemeinsamen Auftretens starten soll, anstatt die Wahrscheinlichkeiten selbst. Zu betrachten ist, dass die Kookurenzswahrscheinlichkeit hängt von drei Eingagnsgrößen ab - *i*, *j*, *k*. Die Algemeinform der Funktion ist in der Gleichung 3.3 angegeben:

$$F(w_i, w_j, \tilde{w}_k) = \frac{P_{ik}}{P_{jk}}, \tag{3.3}$$

mit Wortvektoren $w \in \mathbb{R}$ und andere Wortvektoren $\tilde{w} \in \mathbb{R}$. In der Gleichung ergibt sich die Rechte Seite aus dem Korpus. F hängt in diesem Fall von den drei Vektoren $w_i, w_j, \tilde{w}_k$ ab. F wird demnächst wegen der hohen Variation der Formel angepasst. Zuerst ist es gewünscht, die Information aus der Rate in Word-Vector-Raum darzustellen. Da Vektorräume ursprünglich linear sind, kann dies in einem Vektordiferenz erfolgen. Auf dieser wird der Fakus nur auf die Funktionen fallen, die von der Diferenz von den zwei Vektoren abhängen. Die Änderung ergibt sich in Gleichung 3.4.

$$F((w_i - w_j), \tilde{w}_k) = \frac{P_{ik}}{P_{jk}}. \tag{3.4}$$

Aus der Gleichung ist zu entnehmen, dass die Parameter von F Vectoren sind, während die Rechte einen Skalar ist. Währen F als eine komplexere Funktion, die von einem Neuronalen Netzt parametriesiert werden kann, genommen werden kann, würde das die Linearstruktur der Formel verschleiern. Es wird das Skalarprodukt genommen, damit das vermieden wird. Die Formel 3.5 stellt die Änderung dar.

$$F((w_i - w_j)^T \tilde{w}_k) = \frac{P_{ik}}{P_{jk}}. \tag{3.5}$$

In der word-word Kookuranzmatrizen erfolgt der Unterschied zwischen Wort und Kontextwort willkürlich. Die Formel für F muss es erlauben, die zwei Rollen beliebig zu tauschen. Das heißt also nicht nur $w \leftrightarrow \tilde{w}$ auszutauschen, sondern auch $X \leftrightarrow X^T$. Um diese Symmetrie zu verschaffen, sind zwei einfache Schritte erfordert. Zuerst muss vergewissert werden, dass die Formel homomorphisch zwischen die Gruppen $(\mathbb{R}, +)$ und $(\mathbb{R}_{>0}, \times)$ ist:

$$F((w_i - w_j)^T \tilde{w}_k) = \frac{F(w_i^T \tilde{w}_k)}{F(w_j^T \tilde{w}_k)}, \tag{3.6}$$

was nach Gleichung 3.5 wie folgt gelöst wird:

$$F(w_i^T \tilde{w}_k) = P_{ik} = \frac{X_{ik}}{X_i}. \tag{3.7}$$

Die Lösung von 3.6 ist $F = exp$, oder auch:

$$w_i^T \tilde{w}_k = \log(P_{ik}) = \log(X_{ik}) - \log(X_i). \tag{3.8}$$

Zunächst wird die Gleichung umgestellt, jedoch werden einige Konstanten, oder Bias, eingeführt. Es könnte der Wert von $\log(X_i)$ in einer Konstante b_i umgewandelt werden. Schließlich wird die Konstante b_k addiert, damit die Symmetrie erhalten wird. Die vereinfachte Formel ist in Gleichung 3.9 gegeben:

$$w_i^T \tilde{w}_k + b_i + \tilde{b}_k = \log(X_{ik}). \tag{3.9}$$

4

Der Transformer

In der zweiten Hälfte des Jahres 2017 ein Team von Wissenschaftlern veröffentlichte ihr Papier "Attention Is All You Need", in dessen sie ein neues Model vorstellten. Das Projekt zur Entwicklung wurde unter der Google Research and Google Brain aufgeführt. Dieses Model nahm die Name der "Transformer".

4.1 Struktur des Transformers

Der Transformer besteht aus zwei Haupteinheiten, deren Namen Encoder und Decoder sind. Die beiden Einheiten bestehen aus gleicher Anzahl Encoder-, bzw. Decoder-, Layers. Der im Pappier vorgestellte Encoder verfügte über 6 Encoderschichten und 6 Decoderschichten. Die folgende Abbildung beschreibt den Transformer:

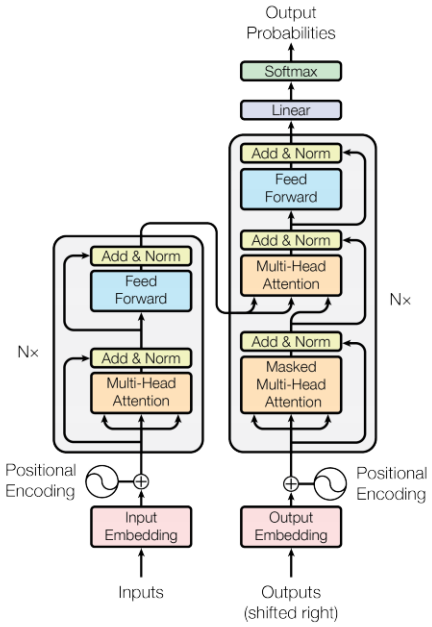

Abb. 4.1: Das Transformer-Modell

In den linken Schichten fließt der Input, meistens mehrere Sätze, durch eine Attention- und eine FeedForward-Network(FFN) Unterschicht. Rechts werden die Targeteingaben, die zugehörigen Sätze für den Input, von zwei Attention- und wieder von einer FFN-Unterschichten. Der Input- und der Targetsatz werden eingebettet, before sie in den Encoder, bzw. Decoder, eingegeben worden. Zuerst werden die einzelnen Wörter der Sätze durch ihre entsprechende Kodierung in Zahlen ersetzt. Demnächst wird eine Positionale Kodierung in der Eingabe eingebettet.

Die nächsten Unterkapitel betrachten die einzelnen Aufbauelementen des Transformers ins Details. Wichtig ist es Hier zu erwähnen, dass die Positionale Einbettung von keiner Einheit im Transformer durchgeführt ist, sondern eine zusätzliche Vorbearbeitung des Textes (Text Preprocessing). Jedoch erkläre ich die Mathematik, die dahinter steckt.

4.1.1 Positionale Einbettung

Die Positionale Einbettung passiert nach der Umwandlung von Text in Zahlen. Wie die Name erratet, wird Information über die Lage des Wortes im Satz in den Wortvektor eingebettet. Diese zusätliche Aktion ist notwig, da im vergleich zu anderen Modelle oder Verfahren, beihaltet der Transformer keine Convolutional- oder Recurenz-Netzwerke.

Die Formel nach dem der Vektor berechnet wird, ist gegeben:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) \tag{4.1}$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) \tag{4.2}$$

Der PE-Vektor besteht aus d_model-Dimensionen. Für jede Dimension wird der Wert des Eintrages entweder mit der Sinus oder der Cosinus-Funktion berechnet. Die geraden Dimensionen entsprechend mit der Sinusfunktion und die Ungeraden mit der Cosinusfunktion.

Als nächstes wird betrachtet, wie der PE-Vektor zum Einbettungs-Vektor addiert wird. In der Literatur werden die Vektoren ganz einfache addiert. Diese Vorgehensweise könnte jedoch Probleme verursachen und wichtige Informationen könnten verloren gehen. Es existieren mehrere Möglichkeiten, wie man das Verlust von Informationen zu vermeiden. Im Buch [Rot00] wird folgende Formel benutzt:

$$pc(i) = y_i * math.sqrt(d_{model}) + pe(i) \tag{4.3}$$

,wo der Eingabevektor durch eine Konstante skaliert wird. Die Variable y_i ist der eingebettete Vektor und pe ist der Positionalsvektor und d_{model} entspricht die Anzahl der Dimensionen des Vektoren benutzt im Model. Der Eingabevektor wird mit dem Wurzel vom Dimensionen skaliert und erst dann wird der Positionalvektor addiert.

4.1.2 Encoder und Decoder

Encoder und Decoder sind die essenziellen Bestandteile vom Transformer. Der Encoder erhält die schon veränderten Daten und führt sie durch ein Attention- und ein Feed Forward Network-Layer. Zwischen jeder Unterschicht besteht eine residierte Verbindung (siehe Abbildung 4.1), sodass die Ausgaben vom letzten Unterlayer mit den Ausgaben vom Aktuellen addiert und weiterhin normiert werden. Der Decoder besitzt eine Unterschicht mehr als der Encoder. In der Transformer beinhaltet der Decoder drei Schichten (siehe Abbildung 4.1). Die letzten zwei sind die selben wie im Encoder. Die erste Unterschicht im Decoder ist eine Masked-Multi-Head-Attention-Schicht. Die Verbindung zwischen Encoder und Decoder erfolgt in der zweiten Unterschicht - zwar in der MHA-Schicht. Da werden die Ausgaben vom Encoder und vom MMHA-Schicht zusammengeführt. Nach der Bearbeitung liefert das Model einen potenziellen Satz, der abhängig vom Aufgaben Stellung, die gesuchte Antwort sein sollte. In diesem Fall ist es die Übersetzung aus dem Englishen ins Spanische. In den folgenden Unterkapiteln werden die Unterschichten ins Details untersucht.

Multi-Head-Attention Layer

Der Multi-Head-Attention Layer kommt in den beiden Einheiten vor. Dieser Schicht folgt eine Normierungsschicht, die die Ausgaben aus dem MHA-Schicht und die Residial-Daten aus vorheriger Schicht addiert und normiert.

Die Eingabe in dem Multi-Head-Attention-Layer vom Encoder ist der Vektor, der die positionale und eingebettete Angaben vom Text erhählt. Im Decoder erhält der MHA-Layer die Eingaben von einer Masked-Multi-Head-Attention-Schicht, und somit ist die Information bis zum aktuell betrachteten Punkt. Der unteschied besteht darin, dass die Information für den Encoder komplett verfügbar ist, und im Decoder wird diese maskiert, und so lernt das Model zu raten. Darin besteht der Unterschied zwischen den MHA im Encoder und Decoder.

Ziel dem MHA-Layer im Encoder‘ ist es die Bezihung zwischen einzelnen Worten zu bestimmen. Das wird erzielt, indem jedes Wort aus dem Satz mit allen anderen abgebildet wird. Jedoch Jedes Wortvektor besteht aus d_{model} Dimensionen. In dem Buch [Rot00] entspricht die Anzahl an Dimensionen gleich 512. Die große Anzahl der Dimensionen würde große Laufzeit anfordern, wenn mehrere Ansichten untersucht werden wollen. Dies ist natürlich möglich mit den stärken Komputern von heute. Der Nachteil ist natürlich, dass das Model immer nur eine Ansicht über die Beziehungen der Wörter betrachtet und es natürlich noch mehr Leistung erfordern würde, um weitere Ansichten zu bestimmen. Eine bessere Alternative ist die Dimensionen jedes Wortes in 8 Teilen, jedem Teil (auch Head genannt im [Rot00]) entspricht 64 Dimensionen, aufzuteilen. Dann wird jeder 64-stückige Teil den 8 unterschiedlichen Heads (deswegen ist die Schicht Multi-Head-Attention-Layer genannt; siehe Abbildung 4.2) zum untersuchen gegeben.

Diese “Köpfe” laufen parallel. Der Vorteil dabei ist es, dass die Laufzeit verringert wird und es 8 unterschiedliche Repräsentationen betrachtet werden. In der

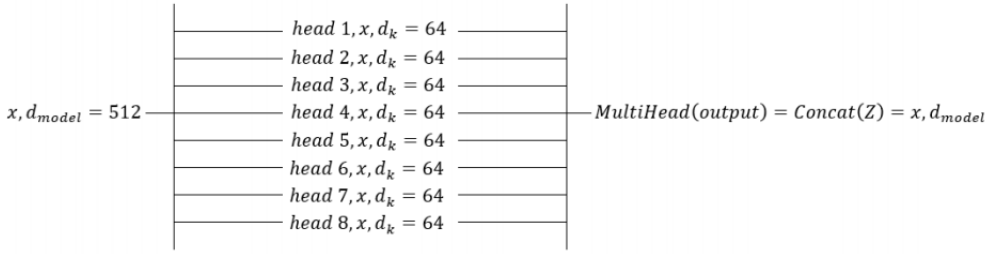

Abb. 4.2: Heads [Rot00]

Abbildung 4.2 erkennt man wie die MHA-Schicht aussieht. Nachdem die Daten von jedem Kopf vorhanden sind, werden die Ergebnisvektoren wieder konkateniert (siehe Abbildung 4.2). Die Ausgabe sieht, dann wie folgt aus:

$$Z = (z_0, z_1, z_2, z_3, z_4, z_5, z_6, z_7) \tag{4.4}$$

Die Matrix Z ist die aus den Ausgaben z_i aufgebaute Ergebnismatrix. Am Ende muss die Matrix Z zusätzlich konkateniert werden, um die ursprünglichen Dimensionen $x * d_{model}$ zu erhalten.

In jedem Kopf wird jedes Wort mit drei Vektoren repräsentiert:

- Einem Query-Vektor (Q), dessen Dimensionalität d_q gleich **64** ist. Der Vektor wird verwendet, oder trainiert, wenn der zugehörige Wortvektor für x_n die Key-Value-Paare gesucht sind, inklusive sich selbst.
- Einem Schlüsselvektor (auch als Key-Vektor bezeichnet K), der trainiert wird, um einen Attention-Wert zu ergeben.
- Einem Wertvektor (auch als Value-Vektor bezeichnet V), der trainiert wird, um einen weitere Anttention-Wert zu ergeben.

Im Buch [Rot00] wird das Attention als SScaled Dot-Product Attention"bezeichnet. Das ist eine Linearkombination der oben deklarierten Vektoren. Die folgende Formel ergibt seine Berechnung:

$$Attention(Q, K, V) = softmax\left(\frac{Q * K^T}{\sqrt{d_k}}\right) * V \tag{4.5}$$

Diese Vektorrepräsentationen werden aus den Gewichtsmatrizen eingelesen. Die Gewichtsmatrizen, sind am Anfang nicht bekannt und werden im Folge der Training bestimmt. Zu Beginn werden sie mit zufälligen Werten erstellt. Die Matrizen werden im Buch [Rot00] als Q_w, K_w und V_k beschriftet. Sie besitzen $d_k = \mathbf{64}$ Spalten und $d_model = \mathbf{512}$ Zeilen. Wenn zum Beispiel ein bestimmtes Query für den Wort x_n abzulesen ist, dann erfolgt das durch eine einfache Matrixmultiplikation:

$$Q_{x_n} = x_n * Q_w K_{x_n} = x_n * K_w V_{x_n} = x_n * V_w \tag{4.6}$$

Wobei x_n repräsentiert in diesem Fall den Indexwert vom Wort x_n. Wenn die Eingabe aus mehreren Worten besteht, wird eine Matrix mit den Dimensionen Anzahl der Worte*d_{model} (für d_{model} meistens 512 gewählt) erhalten.

Für jeden Eintrag in x wird eine Matrix mit seinen Attention-Vektoren berechnet, bzw. den Beziehungsvektoren zu jedem Eingabewort x_n (inklusive sich selbst). Jeder Vektor in der Matrix hat 512 Einträge, und es gibt insgesammt m-viele Vektoren ($m - viele$ Wörter). Den Attention-Vektor für jedes Wort wird erhalten, indem die Vektoren aus der Matrix summiert werden. Schließlich wird jedes Attention-Vektor von allen Heads zusammengeführt, und diese bauen die Attention-Matrix auf. Eine Normierungsschicht folgt der Attnetion-Schicht. Die erhält als Eingabe die konkatenierte Ausgabe Matrix $Z_{concant}$ und die unveränderten Eingabedaten der MHA-Attentionschicht x:

$$v = x + Z_{concat}. \tag{4.7}$$

Die Normierungsschicht führt folgende Berehcnungen dann aus:

$$LayerNorm(v) = \gamma * \frac{v - \mu}{\delta} + \beta. \tag{4.8}$$

Die Variablen bedeuten folgendes:

- μ ist der Durchschnitt von v mit Dimensionen d:

$$\mu = \frac{1}{d} \sum_{k=1}^{d} v_k. \tag{4.9}$$

- δ ist die Standardabweichung von v mit Dimensionen d:

$$delta^2 = \frac{1}{d} \sum_{d}^{k=1} (v_k - \mu)^2. \tag{4.10}$$

- γ ist ein Skalierungsparameter.
- β ist ein Bias-Vektor.

Die weiteren Normierungsschichten im Modell führen analoge Operationen und diese Erläuterung dient als Muster für die weiteren. Somit werden alle anderen Normierungsschichten nicht betrachtet.

Analog sieht die Funktionalität der MHA-Schicht im Decoder aus. Die Eingabe im der Schicht erfolgt aus dem Masked-Multi-Head-Attention-Layer und der Ausgabe des Encoders. Als nächstes wird der Feed-Forward-Network-Sublayer erläutert.

Feed Forward Network

Die Eingabe im Feed-Forward-Network ist die Ausgabe der Normierungsschicht (siehe Abbildung 4.1). Die Eingabe ist ein d_{model} Dimensionales Vektor. Die Struktur des FFN-Layers kann wie folgt beschrieben werden [Vas17]:

- Die Schichten sind sowohl im Encoder als auch im Decoder komplett verbunden.

- Die FFN ist für jeder Wort einzeln anzuwenden. Die Anwendung ist indentisch, jedoch mit unterschiedlichen Parametern.
- Der Netzwerk besteht aus zwei Lienearetransformationen und eine ReLU-Aktivation dazwischen:

$$FFN(x) = max(0, x * W_1 + b_1) * W_2 + b_2. \tag{4.11}$$

- Die Ein- und Ausgabe des FFN haben eine Dimensionalität von $d_model = 512$. Die innere Schicht besteht aus $d_{ff} = 2048$ Neuronen.

Sowohl im Encoder als auch im Decoder ist die Struktur und Funktionalität der FFN-Schicht gleich. Die Ausgaben der FFN-Schicht werden wieder normiert. Die Inputdaten von der nachfolgenden Normierungsschicht sind die Ausgabe vom FFN und dessen Eingabe (in beiden Einheiten gleich; siehe Abbildung 4.1).

Masked Multi-Head Attention Layer

Die letzte Schicht vom Decoder ist die Masked-Multi-Head-Attentnion-Schicht. Diese Schicht hat den selben Aufbau wie die MHA-Schicht. Diese Schicht unterscheidet sich von der im Encoder darin, dass die Eingabe "maskiert" wird. Das bedeutet, dass bestimmte Abschnitte maskiert werden, sodass der Layer begrenzte Informationen erhält. Ziel der Maskierung ist es dem Model zu zwingen, die unbekannten Stellen zu Raten. Deswegen betrachtet das Netz die Information nur bis zur aktuellen Position und die nachfolgenden Stellen müssen erraten werden.

4.2 Implementierung

Die Implementierung des Transformers ist sehr komplex. Der Programmcode kann außerdem auf der Tensorflow Seite [Web21] gefunden werden.

Die Klasse

Für den Aufbau des Models werden 6 Klassen Gebraucht. Diese sechs Klassen beschreiben die wichtigsten Komponente des Transformers und das Model selbst. Es gibt zwei Klassen für den Encoder und Decoder, zwei für die Layers im Encoder und Decoder, die Klasse für das Model und eine Klasse für den Attentionlayers. Zur besseren Überblick stelle ich die Hauptklasse des Transformers:

Listing 4.1: Definition des Transformers

```
1 self.encoder = Encoder(num_layers, d_model, num_heads, dff,
      ↪ input_vocab_size, pe_input, rate)
2 self.decoder = Decoder(num_layers, d_model, num_heads, dff,
      ↪ target_vocab_size, pe_target)
3 self.dense = Dense(target_vocab_size)
```

Die Codezeilen sprechen für sich selbst. Die Dense-Schicht repräsentiert einen komplett verbundenen Netzwerk mit *target_vocab_size*-viele Neuronen. Die Ausgabe davon entspricht die Schätzung des Models. Für jeden einzelnen Wort im Satz wird der Decoded-Vektor in einem anderen Vektor mit Dimensionalität *sequence_length* $\times$ *target_vocab_size* verwandelt. Für jede Stelle im Satz liefert die Denseschicht einen Vektor, der so Groß ist wie die Anzahl der einzigartige Wörter im Targetcorpus, mit den Wahrscheinlichkeiten, dass das Wort an der Stelle platziert werden soll. Somit ratet das Model. Der Encoder und Decoder werden mit den bestimmten Hyperparametern inizialisiert. In meinem Fall verwende ich einen Transformer mit folgenden Hyperparametern:

Listing 4.2: Hyperparameter

```
1 num_layers = 6
2 d_model = 512
3 dff = 2048
4 num_heads = 8
```

Diese Variablen haben die selben Werte wie der ursprüngliche Transformer [Vas17].

Encoder

Die Encoder Klasse besteht aus mehreren Encoderschichten und die entsprechende Embeddingschicht. Die Klasse sieht wie folgt aus:

Listing 4.3: Encoder

```
1 self.embedding = Embedding(input_vocab_size, d_model)
2 self.pos_encoding = self.positional_encoding(
      ↪ maximum_position_encoding, self.d_model)
3 self.enc_layers = [EncoderLayer(d_model, num_heads, dff, rate)
      ↪ for _ in range(self.num_layers)]
4 self.dropout = Dropout(rate)
```

Hier erkennt man die zwei Einbettungsschichten, eine für die Tokeneinbettung und die zweite für die Positionaleeinbettung. In der dritte Zeile werden alle Encoderschichten erstellt. Am Ende wird ein Dropoutschicht angehängt, der dabei Hilft, das Model nicht übertreniert (overfitting) zu werden.

Decoder

Die Decoder-Klasse wird ebenso in eine Separate Klasse ausgelagert. Die Programzeilen 4.4 spiegeln die vorgestellte Struktur des Decoders im oberen Kapitel 4.1:

Listing 4.4: Decoder Implementation

```
1 self.embedding = Embedding(target_vocab_size, d_model)
2 self.pos_encoding = self.positional_encoding(
      ↪ maximum_position_encoding, d_model)
```

```
3  self.dec_layers = [DecoderLayer(d_model, num_heads, dff, rate)
       ↪ for _ in range(num_layers)]
```

Entsprechend werden in Zeile 4 die Decoder-Schichten erstellt. Ebenso wird in Zeilen 1 und 2 die zwei Einbettungsoperationen für die Einbettungsschicht definiert. Diese Layers können sich von der Einbettungsschichten im Encoder unterscheiden.

5

BERT

In diesem Kapitel wird die Struktur der **B**idirectional **E**ncoder **R**epresentations from **T**ransformers. Wie aus der Name zu schließen ist, basier der BERT auf den Transformer (4).

5.1 Struktur des BERTs

Im Herzen des BERTs liegt der Transformer. Das BERT-Model nutzt die Hauptcharakteristik des Transformers, Beziehungen der Wörtern im Corpus zu erlernen. Im Vergleich zum Transformer, wo zwei Einheiten zusammenarbeiten, ist im BERT nur die Encoder-Einheit nötig, weil nur ein Sprachenmodel erstellt werden soll. Wie beim Transformer werden die Eingaben vorverarbeitet. In diesem Aspekt haben die beiden Modelle kleine Unterschiede. Während im Transformer die Einbettungparameter für die Eingaben vor dem Trainieren schon bekannt ist, werden diese im BERT-Model als Hyperparameter inizialisiert. Das bedeutet, dass alle Positional- und Segmenteinbettungsvariablen während des Trainingsprozesses erlernt werden. Die Tokeneinbettungstabelle wird auch bei dem BERT vor dem Lernprozess bakannt. Diese kleine Änderung erfordert ein unterschiedliches Vorgehen beim Lernen des Models. Der Lernprozess des BERTs wird aus diesem Grund in zwei Phasen aufgeteilt - Pre-training (erste Phase) und Fine-tuning (zweite Phase). Die erste Phase konzentriert sich auf die Erlernung der einzelnen Hyperparametern der Einbettungslayer. Die zweite Phase des Lernens versucht die vorgegeben Task zu lösen. Für den besten und schnellsten Ergebnis am Ende sollte die selbe Tokeneinbettungstabelle verwendet werden. Jedoch ist diese Anforderung keine Regel, denn die Anwendung von zwei getrennten Tabellen würde zu längeren Lernzeiten führen. Durch die große Anzahl an Vortrainierten Modellen im Netzt lohnt es sich ein Model wiederzuverwenden und für die eigene Task anzupassen. In den folgenden Kapitel werden beide Phasen dargestellt. Es wird zuerst der Prozess der Vorerlernung vorgestellt und danach wie ein vortrainierter Modell für die gewünschte Task angepasst wird.

5.2 Implentierung

Die folgenden drei Kapitel stellen das BERT Model ins Details und eine Mögliche Implementierung des Models mit der Bibliothek Tensorflow vom Google. Die Implementierung basiert auf Publikationen (Hier Angeben Welche). Im Paper [JDT19] aus dem 2019 wird ein neues Vorgehen im Bereich des Natural Language Processing vorgestellt. Die Authoren stellen eine bessere Alternative zum Fineeinstellung der Transformermodelle. Im Artikel wird die unidirektionale Beschränkung des Transformer verbessert, indem ein Masked-Language-Model (MLM) als Vortrainierungsziel verwendet wird. Das verwendete Model maskiert zufälligerweise Tokens aus der Eingabe und die Idee ist, diese Token-ID aus dem Kontext herzuleiten. Dieses Ziel verbindet den linken und rechten Kontext vom Satz und das trainierte Transformer ist bidirektional. Die Autoren kombinieren dieses Ziel mit dem Next Sentence Prediktion-Ziel (NSP), sodass sie zwei Tasks gleichzeitig verarbeiten. Das zweite Ziel versucht zu raten, ob zwei gegebenen Sätze nacheinander im Text vorkommen.

In der zweiten Phase wird die Task nach bedarf ausgesucht. In meinem Fall ist das eine Übersetzungstask aus dem Englischen ins Portugiesische. Welche Task versucht wird, hängt nicht vom Pretraining. Das bedeutet, dass eine Zieländerung des vortrainierten Models immer möglich ist, sobald ausreichende Daten vorhanden sind.

Als nächstes wird die Anpassung der Einbettungs- und Encoderparametern im Code vorgestellt.

5.2.1 Pre-Training a BERT-Model

Der Prozess des Vortrainierens erfordert die meiste Zeit von den beiden Phasen. Der Grund dafür ist die große Anzahl an Variablen im Model. In meiner Ausarbeitung habe ich die selben Mechanismen, die im Atrikel [JDT19] vorgestellt werden, verwendet. Das heißt, dass es zwei Task gleichzeitig gelöst werden. Das Masked Language Model wurde im vorrigen Unterkapitel eingeleitet, aber in dieser Abschnitt stelle ich der Prozess näher vor. Die Eingabedaten werden speziell für das Model vorbereitet. Laut dem Artikel [JDT19] werden Wörter, die 15% vom Batchsize entsprechen, maskiert. Das bedeutet 10 Wörter bei einem Batchsize von 64 und 5 Wörter bei 32. Außerdem jedes Wort im Batch hat eine Wahrscheinlichkeit von 10% maskiert und 10% Chance durch ein weiteres Wort ersetzt zu werden. In 80% der Fälle wird das Wort behalten.

Beim Next-Sentence-Prediction ziel ist es zu raten, ob die beiden Sätze kontextuell verbunden sind. Hier wird in der hälfte der Fälle zwei benachbarten Sätze genommen und die weiteren Paare sind zwei kontextuell unterschiedliche Sätze. In diesem Sinne gelten die zwei Sätze tokenized als Input und einen Wert aus zwei Klassen als Label für die NSP-Task. Die Klassen können beliebig festgelegt werden, üblicherweise werden die Werte 0 (nicht benachbarte Sätze) und 1 (kontextuell benachbarten Sätze) verwendet.

5.2.2 Erstellen von Eingabe und Klasse

Für das Vortrainieren wird das *wikitext-2-v1* [Mer] Corpus verwendet. Das Corpus ist eine Sammlung von Wiki-Artikeln in der Englischen Sprache. Außerdem besteht es aus mehr als 100 millionen Tokens. Die Trainingsdatei besteht aus mehr als 35 Tausend Zeilen Text. Der Datensatz ist in der Literaturverzeichniss verlinkt und kann da heruntergeladen werden. Als Erstes werden die Daten aus dem Corpus vorbereitet fürs Training.

Die Klasse Wiki2Corpus bereitet das ganze Corpus vor. Zuerst werden alle Wörtern in Zahlen mit Hilfe eines Tokenizers verwandelt. Dafür wird aus der Bibliothek *d2l* [AS] den Tokenizer genutzt. Als nächstes wird die Vocabulary erstellt, damit Inferenzen des Models später in Worte verwandelt wird. Der Wortschatz ergibt sich aus den gesammt Text. Hier bietet d2l eine Vocabulary-Klasse, die jedem Wort aus dem Korpus einen Token zuweist. Die Klasse bietet die Möglichkeit auch seltene Wörter auszufiltern. Einen Programmcode die diese Operationen darstellt ist gegeben:

Listing 5.1: Nutzung der Dive into Deep Learning (d2l) Bibliothek

```
1 paragraphs = [d2l.tokenize(paragraph, token='word') for
    ↪ paragraph in paragraphs]
2
3 self.vocab = d2l.Vocab(sentences, reserved_tokens=['<pad>', '<
    ↪ cls>', '<sep>', '<mask>'])
```

Als nächstes werden die Samples für das NSP-Model vorbereitet. In der utils.py Datei werden diese und zusätzlich nötigen Methoden für die Vorbereitung erstellt. Diese sind gegeben:

Listing 5.2: Erstellen der Trainingsdaten für NSP

```
1  def get_nsp_data_from_paragraph(paragraph, paragraphs, max_len)
     ↪ :
2     nsp_data_from_paragraph = []
3     for i in range(len(paragraph) - 1):
4        # prepare sentence pairs and label
5        sentence_a, sentence_b, is_next = _get_next_sentence(
           ↪ paragraph[i], paragraph[i + 1], paragraphs)
6        if len(sentence_a) + len(sentence_b) + 3 > max_len:
7           continue
8        token, segment = _get_tokens_and_segments(sentence_a,
           ↪ sentence_b) # add keywords
9        nsp_data_from_paragraph.append((token, segment, is_next))
10    return nsp_data_from_paragraph
```

Nachdem die Samples für das NSP-Model vorbereitet wurden, müssen die Eingaben und Labels für das MLM erzeugt. Die neu erzeugten Datensätze müssen mittel Schlüsselwörter abgegrenzt werden. Die Schlüsselwörter ['CLS'], ['SEP'],

['MASK'] und ['PAD'] müssen an die entsprechende Positionen im Satz gefügt werden. Der ['CLS']-Token kennzeichnet den Beginn des Satzes. Der ['SEP']-Token wird am Ende des Satzes gestellt und so dient er auch zur Abgrenzung der Sätze, falls mehrere Sätze als Eingabe dem Model gegeben werden. Der Mask-Token ersetzt ein maskiertes Wort und der Padding-Token wird für die Erweiterung des Satzes bis zur maximalen Satzlänge. Die Einsetzung der Schlüsselwörter kann sowohl vor der Erstellung des Samples als auch nachher. Die neu erzeugeten NSP-Sätzepaare werden für das MLM maskiert. Die Erstellung der Input und Labelpaare erfolgt wieder durch mehreren Methoden in utils.py:

Listing 5.3: Erstellen von Trainingsdaten für MLM

```
1  def get_mlm_data_from_tokens(tokens, vocab):
2     candidate_pred_positions = []
3     for i, token in enumerate(tokens):
4        if token in ['<cls>', '<sep>']:
5           continue
6        candidate_pred_positions.append(i)
7     num_mlm_preds = max(1, round(len(tokens) * 0.15)) # number
         ↪ of masked tokens
8     # Mask the sentences
9     mlm_input_tokens, pred_positions_and_labels =
         ↪ _replace_mlm_tokens(tokens, candidate_pred_positions,
         ↪ num_mlm_preds, vocab)
10    pred_positions_and_labels = sorted(pred_positions_and_labels
         ↪ , key=lambda x: x[0])
11    pred_positions = [v[0] for v in pred_positions_and_labels]
12    mlm_pred_labels = [v[1] for v in pred_positions_and_labels]
13    # tokenize the sentences
14    return vocab[mlm_input_tokens], pred_positions, vocab[
         ↪ mlm_pred_labels]
```

Am Ende sieht die generierte Datensammlung wie folgt aus:

Listing 5.4: Eingabedaten

```
1     # input tokens  # segment vector # sequence length # masked
         ↪  token index # weights for the label
2     (self.all_token_ids, self.all_segments, self.valid_lens,
         ↪ self.all_pred_positions, self.all_mlm_weights,
3     # labels for the input # NSP labels for
4     self.all_mlm_labels,  self.nsp_labels) = pad_bert_inputs(
         ↪ examples, max_len, self.vocab)
```

Jetzt sind alle Trainingsdaten bereit. Der nächste Schritt ist die Erstellung des Models.

BERT Class

Hier verwende ich das vorgestellte Model im [JDT19]. Folgende Hyperparametern sind in meinem Fall definiert:

Listing 5.5: Die Hyperparametern vom BERT

```
1 cfg = {
2    'batch_size': 64,
3    'input_max_len': 64, # sequence length
4    'num_layers': 12, # number of attention layers
5    'd_model': 768, # number of neurons im model
6    'num_heads': 12, # number of heads in each attention layer
7    'depth_FF_Layers': 1024 # number of feed forward neurons
8 }
```

Diese Konfiguration entspricht dem $BERT_{BASE}$-Model, das im Artikel [JDT19] vorgestellt wird. Es wird zusätzlich das $BERT_{LARGE}$-Model definiert, der entsprechen $num\_layer = 24$, $d_{model} = 1024$, $num\_heads = 12$. Das Basemodel besitzt 110 Mio. Parametern, während das Großemodel dreimal so viel - insgesammt 340 Mio Parametern. Dadurch ist das Trainieren eines BERT-Models sehr anspruchsvoll, jedoch günstig sich ein vortrainiertes Model zu besorgen. Auf dieser Weise spart man sich die Hälfte der Zeit.

Das Model wird in einer eigenen Klasse definiert. Die Bestandteile des BERT-Models werden als Programmcode gegeben:

Listing 5.6: BERT-Struktur bei Pre-Training

```
1 self.encoding = BERTEncoderLayer(num_layers, d_model, num_heads
    ↪ , dff, input_vocab_length, maximum_positional_encoding,
    ↪ rate=rate, layer_norm_eps=layer_norm_eps)
2 self.nsp_layer = Dense(2)
3 self.mlm_layer = MLMLayer(input_vocab_length, d_model)
4 self.softmax = Softmax()
```

Das Model besteht aus vier Schichten, zwei davon werden in seperaten Klassen ausgelagert. Die zwei Klassen sehen wie folgt aus:

Listing 5.7: Definition des BERT-Encoder-Layers

```
1 # BERTEncoderLayer.py
2 # The three embedding layers
3 self.embedding = Embedding(input_vocab_size, self.d_model)
4 self.segment_encoding = Embedding(2, self.d_model)
5 self.pos_encoding = tf.Variable(
6 initial_value=tf.random_normal_initializer(mean=1., stddev=
    ↪ initializer_range)(shape=[1, maximum_position_encoding,
    ↪ d_model]),
7 trainable=True)
```

```
 8  self.dropout = Dropout(rate)
 9  # The Transformer Encoder Layer: 12 heads and 12 Layers
10  self.encoder_layer = [EncoderLayer(self.d_model, num_heads, dff
        ↪ , rate, layer_norm_eps) for _ in range(self.num_layers)]
11  # MLMLayer.py
12  # A Sequential Fully Connected Model
13  self.mlp = Sequential()
14  self.mlp.add(Dense(num_hiddens, activation='relu'))
15  self.mlp.add(LayerNormalization())
16  self.mlp.add(Dense(vocab_size))
```

Die Einbettungsschicht besteht aus drei erlernbaren Unterschichten. Die Funktionalität der Schicht ist in der Abbildung 5.1 dargestellt:

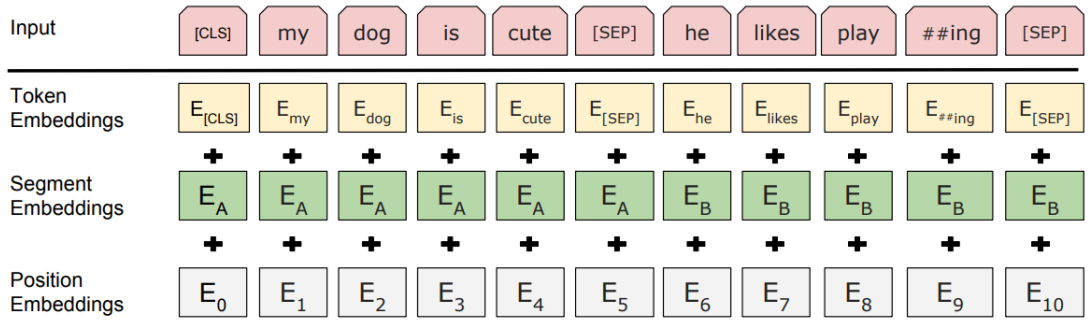

Abb. 5.1: BERT Eingabeschicht [JDT19]

Aus der Abbildung 5.1 kann man nicht nur die Funktion der Schichten lesen, sondern auch ihre Eingaben. In der Token-Embedding-Schicht werden die Tokens der Wörtern gelesen und die Schicht liefert der zugehörige Vektorrepräsentation. Die Segment-Embedding-Schicht erhält den Segmentenvektor, der beim Vorverarbeitung der Eingabepaare erstellt wird. Der Positional-Embedding-Layer codiert die Wortindexen in Vektoren. Die Summe der Ergebnisvektoren aus den einzelnen Schichten liefert den Eingebetteten Vektor, der die Eingabe im Encoder ist.

Optimizer und Loss-Funktion

Der Nächste Schritt ist die Definition der Optimizer und die Fehlerfunktion. Hier verwende ich den Adaptive-Movement-Optimizer. Laut [Wie21] liefern Adam und RMSProp (**R**oot **M**ean **S**quare **Prop**agation) bessere Richtigkeit und bessere Fehlerrate als die anderen Optimizers, wie SGD(**S**tochastic **G**radient **D**escent) and AdaGrad(**Ada**ptiv **Gra**dients). Bei unser ersten Task (Masked Language Processing) wird versucht die maskierten Wörter bestimmt zu werden. Das impliziert, dass jedes maskierte Wort einen aus mehreren Klassen gehören kann, bzw. einen aus vielen Wörtern sein kann. Aus diesem Grund können nur zwei Fehlerfunktionen

verwendet werden, die *CategoricalCrossEntropy* und *SparseCategoricalCrossEntropy* sind.

CategoricalCrossentropy ist geeignet, wenn die Ausgabe die partielle Zugehörigkeit zu den Klassen ist. *SparseCategoricalCrossentropy* ist besser geeignet, wenn die Ausgabe keinen Vektor aber einen Integerwert ist. Dieser Integer stellt die zugehörige Klasse dar. Welche Fehlerfunktion verwendet wird, hängt nur von der Form der Ausgabe. In unserem Fall habe ich mich für den *CategoricalCrossEntropy* entschieden, da Die Ausgaben einen Softmax-Vektor ist. Später bei der Anpassung 5.2.3 wird die *SparseCategoricalCrossEntropy* verwendet.

5.2.3 Fine-Tuning a BERT-Model

Das Model für den Fine-Tuning wurde vom TensorflowHub [Hub21b] genommen. Es wird ein Model mit den selben Hyperparametern wie das Model aus dem letzten Unterkapitel 5.2.2 verwendet. Auf TensorflowHub sind mehrere Modelle zur wiederverwendung, hier unteranderem BERT multilangual Cased Model, BERT English uncased, sowohl das Base-BERT-Model als auch das Large-Model. Dementsprächend kann das geeignete Model für die Task ausgewählt. In meinem Fall will einen Translationmodel aufbauen, für diesen Zweck habe ich das Multilingual BERT-Model [Hub21a] genutzt.

BERT-Model

Das gelernte BERT-Model kann wie folgt geladen werden:

Listing 5.8: Laden von dem BERT-Model

```
1    # directly from the website
2    pre_trained_model = 'https://tfhub.dev/tensorflow/
        ↪ bert_multi_cased_L-12_H-768_A-12/4'
3    # or from file system
4    pre_train_model_path = 'abs/path/to/Model/dir'
5    encoder_input = hub.KerasLayer(pre_trained_model, trainable=
        ↪ True, name="BERT_Encoder")
```

Das Model kann als eine Schicht geladen und somit in dem Model angefügt werden. Für den Lernprozess braucht das Model eine weitere Schicht, die das Raten darstellen soll. Dafür bietet sich eine einfache Dense-Schicht am Model anzuhängen, die dann die Ausgabe des BERT-Models mit einem Fully Conected Layer verbidet. Die Rolle der Dense-Schicht ist die Ausgabe in der richtige Shape zu verwandeln. Die Ausgabe vom BERT-Model ist einen Vektor mit der Kardinalität $batch\_size \times sequence\_length \times d\_model$. Die Dense-Schicht verwandelt den Ausgabevektor in der Kardinalität $batch\_size \times sequence\_length \times vocab\_size$. Die Sequenzgröße entspricht die Länge des Satzes, oder jeder Index beschreibt ein Wort im Satz. Die Variable *d_model* repräsentiert die Anzahl der Eigenschaften, nach dennen Wörter klassifiziert werden. Die Dense-Schicht verbundet somit die Eingeschaften der einzelnen Worten zu einem Neuron und die Ausgabe des Neurons ist

einen Wert, der für oder gegen einen bestimmten Wort aus dem Wortschatz. Ein weiterer Schritt ist gebraucht, damit diese Werte in Wahrscheinlichkeiten umgewandelt werden und das erfolgt nämlich durch Anwendung der Softmax-Funktion. Das heißt, dass es dem Model gesagt wird, dass es raten soll, wie die Übersetzung des Eingabesatzes lautet. Hier ist das Model gegeben:

Listing 5.9: Definition des BERT-Models zur Anpassung

```
# input Layer with shape=(None, seq_length)
# these are the expected inputs in the BERT model
inputs = dict(
    input_word_ids=tf.keras.layers.Input(shape=(
        ↪ max_sentence_size,), dtype=tf.int32),
    input_mask=tf.keras.layers.Input(shape=(max_sentence_size,),
        ↪  dtype=tf.int32),
    input_type_ids=tf.keras.layers.Input(shape=(
        ↪ max_sentence_size,), dtype=tf.int32)
)
# BERT-Model
encoder_input = hub.KerasLayer(pre_trained_model, trainable=
    ↪ True, name="BERT_Encoder")
outputs = encoder_input(inputs)
net = outputs['sequence_output']
# dropout layer to help prevent overfitting
net = tf.keras.layers.Dropout(0.1)(net)
# dense layer for guessing
output = tf.keras.layers.Dense(vocab_size, activation=None)(net
    ↪ )
# This is our Model
model = tf.keras.Model(inputs, output)
```

Datensatz

Die Übersetzungstask ähnelt sehr eine Frage-Antwort-Task. Deswegen müssen unsere Datensammlung so vorbereiten, dass wir Satz und Übersetzung gruppieren. Für diese Task habe ich den Datensatz *ted_hrlr_translate_pt_en_converter* verwendet. Der Datensatz ist auf dem Google Storage zu finden. In der Datensammlung sind Sätze auf Englisch und Portugiesisch, also wir führen eine Übersetzung vom Englischen ins Portugiesische. Die Datensammlung besteht aus 51785 Sätze jeweils in Englisch und Portugiesisch, insgesamt 103570 Sätze. Die Sätze die über die erlaubte Sequenzlänge sind, werden verworfen. Die Ausgewählte Satzgröße beträgt 128. Da die Höhere Satzlänge eine längeren Lernzeit bedeutet, habe ich in Bezug zu der Hardware, die mir zur Verfügung steht, eine Sequenzlänge von 128 Wörter, inklusive '[CLS]' and '[SEP]' Schlüsselwörter. Das Laden der Datensammlung erfolgt über den folgenden Programmcode:

Listing 5.10: Laden der Trainingsdaten

```
1    model_name = 'ted_hrlr_translate_pt_en_converter'
2    tf.keras.utils.get_file(f"{model_name}.zip", f"https://
        ↪ storage.googleapis.com/download.tensorflow.org/models
        ↪ /{model_name}.zip", cache_dir='.', cache_subdir='',
        ↪ extract=True)
3    tokenizer = tf.saved_model.load(model_name)
```

Nach der Ausführung des Codes wird die Datensammlung im aktuellen Ordner heruntergeladen und entpackt. Mit Hilfe der dritten Zeile wird der Datensatz zur Nutzung geladen. Der Nächste Schritt ist die Vorbereitung der Sätze für den Lernprozess. Hier Müssen die Eingaben in einem Dictionary gespeichert werden, da das BERT-Model so definiert wird. Wie die Eingabe aussieht, ist in der dritten Zeile aus dem Listing 5.9 zu erkennen.

Optimizer und Fehlerfunktion

Als Nächstes werden die Optimizer und die Fehlerfunktion definiert. Für den Optimizer wird zwar einen Adam-Optimizer, der aber einen Scheduler für die Lernrate besitzt. Das bedeutet, dass die Rate im Laufe des Trainings angepasst wird. Die Konfiguration für den Optimizer ist durch den Programmcode gegeben:

Listing 5.11: BERT Optimizer

```
1    from official.nlp import optimization
2    # learning rate
3    init_lr = 5e-5
4    # number steps pro epoch
5    steps_per_epoch = len(train_input_array)
6    # number of warmup stels
7    num_warmup_steps = int(0.1 * steps_per_epoch)
8    # the optimizer
9    optimizer = optimization.create_optimizer(init_lr=init_lr,
10   num_train_steps=steps_per_epoch,
11   optimizer_type='adamw')
```

Die genutzte Fehlerfunktion ist SparceCategoricalCrossEntropy. Der Grund dafür ist die Struktur des Labels und die Ausgabe des Models (siehe Erklärung im Unterkapitel 5.2.2). Die Fehlerfunktion hat keine besonderen Konfiguration, außer einen Parameter *from_logits* der auf True gesetzt wird. Dadurch wird der Fehlerfunktion gesagt, dass keine Softmax im Voraus angewendet wird. Dementsprechend wird eine Sotfmax-Funktion zuerst ausgeführt, bevor die Fehlerrate errechnet wird.

6

Neural Machine Translation

Die Neural Machine Translation Model (NMT) ist eins der Komplexeren Modele, die in seinem Kern ein BERT-Model und ein Transformer. Die Kombination der beiden Modelen bildet einen Supertransformer, sodass BERT und Transformer den NMT-Model ausbauen. Das BERT-Model wird für die Einbettung der Eingabe, welche Ausgabe in den Encoder und Decoder des Transformermodels später einfließt. Der Encoder muss somit keine eigene Einbettung für seine Eingabe berechnet, doch der Decoder muss seine eigene Einbettung berechnen. Für eine detailierte Erklärung der beiden Modellen kann als Referenz die früheren Kapitel benutzt werden (Kapitel 4 und 5.1). Es wird jedoch der Fluß der Informationen hier gegeben:

Schritt-1: Bei einer Eingabe $x \in X$, BERT bettet den x-Vektor als $H_B = BERT(x)$ ein. Wo H_B ist die Ausgabe des letzten Layers vom BERT. $h_{B,i} \in H_B$ stellt den i-th Wort in x.

Schritt-2: Sei H_E^l gegeben, was die Darstellung der versteckten l-ten Schicht beschreibt und H_E^0 ist die Einbettung der Sequenz x. Der i-te Element aus H_E^l wird als h_i^l für $0 < i < [l_x]$ definiert (begrenzt durch die Anzahl der Neuronen in jeder versteckten Schicht l). Die Berechnung von h_i^l ist gegeben:

$$\tilde{h}_i^l = \frac{1}{2}(attn_S(h_i^{l-1}, H_E^{l-1}, H^{l-1}) + attn_B(h_i^{l-1}, H_B, H_B)), \text{ für } 0 < i < [l_x] \quad (6.1)$$

Wo $attn_S$ und $attn_B$ sind Attention-Modele, bzw. MHA-Schichten, mit unterschiedliche Parametern (die Definition ist in Kapitel 4.1.2 zu finden).

Weiterhin werden die berechneten Attentions nach ihrer Normierung dem Feed-Forward-Network gegeben und schließlich wird die Ausgabe normiert. Die Ausgabe aus der letzten Schicht ist durch H_E^L definiert.

Schritt-3: die Eingabe im Decoder-Layer l wird durch $S_{<t}^l$ für den Zeitpunkt t, $S_{<t}^l = (s_1^l, , s_{t1}^l)$. s_1^0 ist ein besonderer Zeichen, der den Anfang der Sequenz bestimmt und s_t^0 ist die Voraussage des Models zur bestimmten Zeitpunkt t. Im Schicht l erfolgt folgendes:

$$s^l t = attn_S(s^{l1}t, S_{<t+1}^{l1}, S_{<t+1}^{l1}) \quad (6.2a)$$

$$\tilde{s}_t^l = 1/2(attn_B(s_t^l, H_B, H_B) + attn_E(s_t^l, H_E^L, H_E^L)), \; s_t^l = FFN(s_t^l) \quad (6.2b)$$

Die $attn_S$, $attn_B$ und $attn_E$ stellen entsprechend Self-, BERT-Decoder- und Encoder-Decoder-Attention dar. Eqn. (6.2) iteriert über die Schichten und liefert als Ergebnis s_t^L. Die Ausgabe fließt durch eine Dense-Schicht und schließlich mit einer Softmax-Funtion transformiert, sodass das t-te Wort $\hat{y}_t$ erraten wird. Die Entschlusselung verläuft bis ein Endzeichen erreicht wird, bzw. bis die Ende der Sequenz verarbeitet wird.

7

Vektordarstellung

7.1 Einleitung

Meistens bei der Repräsentation und Analyse einer nummerischen Datensammlung werden die Daten nach bestimmten Kriterien klassifiziert, meistens nach mehr als zwei oder drei, sodass eine Graphische Darstellung relative schwierig zu bilden ist. In der Datenanalyse existieren entsprechende Methoden zur Darstellung von Daten mit mehreren Komponenten. Eins dieser Methode ist Principal Component Analysis (abgekürzt PCA), oder Prizipiele Komponentenanalyse. Diese Methode ist ideal in der NLP zu verwenden, da die Wörter in mehrdimensionalen Vektoren repräsentiert werden, üblicherweise solche mit mehr als drei Komponente. Die Methode verringert die Anzahl der Komponenten auf eine kleinere Zahl, zwei oder drei Dimensionen für eine Darstellung der Wörter im 2D- oder 3D-Raum entsprechend, jedoch so viele Informationen wie möglich über die einzelnen Wörter zu behalten.

7.2 Prozess

Ekläre wie die Berechnung erfolgt und, dass es eine Matrix verändert. Erkläre über DataFrame

1. Schritt: Standardization

Im ersten Schritt des PCAs handelt es sich um Standardisierung der Komponenten, sodass jeder gleichmäßig zu der Analyse beibringt. Dieser Schritt ist wichtig, da PCA sehr sinsibel bezüglich Variation der Werte ist. Variablen mit großen Werten dominieren solche mit niedrigen und so ist das Endergebnis beieinflusst/ voreingenommen. Die Standardiesierung erfolgt in Formel:

$$z_{ij} = \frac{value - mean}{standard\ deviation}, \tag{7.1}$$

Wo Z_{ij} der normierte Wert mit Zeile i und Spalte j aus der Matrix ist. Der Meanwert ist der Durchschnitt in einem Vektor und der Standard Deviation bezieht sich auf dem selben Vektor. Nach der Normierung haben alle Werte der Matrix den selben Maßstab.

2. Schritt: Berechnung der Kovarianzmatrix

Nachdem die Normierte Matrix berechnet wurde, wird die Kovarianzmatrix erstellt. Ziel der Kovarainzmatrix ist es die Beziehung zwischen die Variablen zu bestimmen, bzw. wie sie wachsen. Die Abhängigkeit wird von dem Vorzeichen der Covarianzwert bestimmt - bei positivem Wert wachsen die Variablen proporzional und bei negativem - antiproporzional. Die Formel für die Kovarianz ist gegeben:

$$Cov(X,Y) = \sum \frac{E((X-\mu)(Y-\nu))}{(n-1)} \tag{7.2}$$

Die Variable n entspricht die Anzahl der Componenten in X und in Y. Die Zwei konstanten μ und ν sind die Durchschnittswerte der beiden Variablen X und Y. Mit E ist der Erwartungswert des Produktes gegeben.

Die Kovarianzmatrix ist eine $p \times p$ symmetrische Matrix mit Einträgen als die Corianzwert für alle möglichen Paare, gebildet aus allen Variablen, in diesem Fall normierten Wortvektoren. Zur Darstellung betrachten wir einen Datensatz mit 3 variablen x, y und z. Die Kovarianzmatrix sieht wie folgt aus:

$$\begin{pmatrix} Cov(x,x) & Cov(x,y) & Cov(x,Z) \\ Cov(y,x) & Cov(y,y) & Cov(y,Z) \\ Cov(z,x) & Cov(z,y) & Cov(z,Z) \end{pmatrix} \tag{7.3}$$

In der Diagonale der Matrix stehen die Werte für Covarianz der Variablen mit sich selbst. Dieser Wert entspricht der Varianz der Variable. Da die Covarianz Kommutative ist, sind die obere und untere Dreiecksmatrix symmetrisch in bezug auf die Diagonale, beziehungsweise gleich.

3. Berechnung der Eigenvektors und Eigenwerte

Der nächste Schritt erfordert die Berechnung der Eigenvektors und Eigenwerte der Kovarianzmatrix. Auf diesem Weg bestimmen wir die gesuchten prinzipiellen Komponenten. Diese Komponenten werden als lineare Kombination oder Mischung der Ursprungsvariablen erstellt. Die neuen Variablen sind unabhängig von einander. Der Prozess versucht die meiste Informationen aus allen Variablen in die ersten prinzipiellen Komponenten zu beladen. Das erlaubt es die Dimensionen zu verrigern, ohne große Mengen an Information zu verlieren. Es ist jedoch wichtig zu erwähnen, dass die prinzipiellen Komponenten nicht interpretierbar sind, da sie aus der Linearkombination der alten Variablen berechnet werden.

Geometrisch angesehen die prinzipiellen Komponenten sind Richtungen die einen maximalen Varianzwert darstellen. Das sind Geraden, die die meisten Punkte in einem n-Dimensionalen Raum beschreiben. Die Beziehung zwischen Varianz und Information ist es, dass je größer die Varianz bezüglich einer gegebenen Linie, desto mehr Punkten, bzw. Variable, entlang dieser Linie verteilt sind, umso mehr Information von dieser Linie getragen wird.

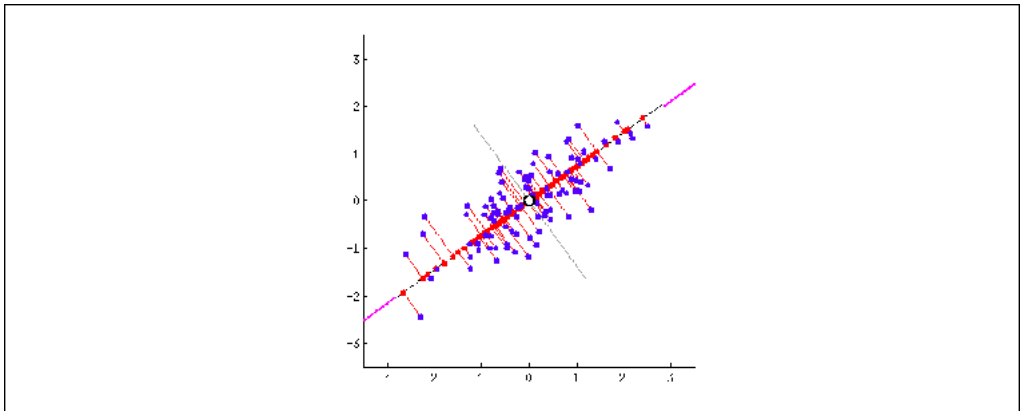

Abb. 7.1: Prinzipiele Linie

In der Abbildung 7.1 ist die Linie mit der größte Variation und so ist die die erste Prinzipielle Komponente (PK), da sie mit sich die meiste Information trägt. Falls die Variablen dann mit Hilfe dieser PK transformiert werden, wird die meiste Information übertragen. Nachdem die erste Komponente gewählt wird, wird die zweite auf den selben Prinzip gewählt, jedoch wird eine andere Linie gesucht, die dann unabhängig von der erste, meisten eine die Orthogonal zu der Erste liegt. Diese zweite besitzt entsprechend die zweitgrößte Varianz zu den Variablen aus der Datensammlung. Dieser Prozes wiederholt sich bis alle prinzipiellen Komponenten bestimmt sind, oder p oft - genau so oft wie wir Variablen in unsere Kovarianz-matrix haben.

Nun zurück zu den Eigenwerten und -vektoren. Zu Jedem Eigenwert gehört ein Vektor und umgekehrt. Sie Kommen immer in Paare und ihre Anzahl entspricht die dimensionalität der Matrix, wie schon oben erwähnt wurden. Ihre Beziehung in mathematischer Form kann wie folgt dargestellt werden:

$$Av = \lambda v, \tag{7.4}$$

wo A ist die Matrix, v der Eigenvektor und λ der zugehörige Eigenwert. Es existiert für eine quadratische Matrix einen Vektor v und einen Faktor λ, sodass bei der Multiplikation der Matrix mit dem Vektor, erhalten wir das gleiche Ergebnis, wie wenn wir den Vektor mit dem Faktor multiplizieren. Es kann die Formel in 7.4 umgeformt werden, um nun die Eigenvektoren und Eigenwerte zu berechnen:

$$(A - \lambda E).v = 0. \tag{7.5}$$

In der Gleichung enspricht E gleich der Einheitsmatrix. Der Eigenvektor ist Lösung der Gleichungssystem. Wir setzte für λ den entsprechenden Wert ein und lösen nach v. Jedoch muss der Eigenwert zuerst berechnet werden. Der wird aus der folgenden Formel errechnet:

$$det(A - E) = 0 \tag{7.6}$$

Wo wir nach den Nullstellen der Determinante suchen. Diese Nullstellen sind die Eingenwerte der Matrix A.

Die Eigenvektoren bestimmen eigentlich die Richtung der Axen mit der meisten Information, oder auch Prinzipielle Komponenten genannt. Die Eingewerte sind die Koeffizienten der Komponente. Je höher der Wert, desto mehr Information wird durch ihre Richtung repräsentiert. Falls die Eigenvektoren nach ihren Eigenwerte absteigend geordnet sind, werden die Ordnung der Prinzipiellen Komponenten bestimmt, sodass die erste Komponente entsprechend diese ist, die den größten Eigenwert besitzt. Als alle Vektoren angeordnet sind, wird einen neuen Vektor aus den zusammengesetzt. Jeder Eigenvektor ist eine Spalte im neuen Vektor. Als nächstes muss die Entscheidung getroffen werden, wie viele prinzipiellen Komponente erhalten werden. Diese Frage hat keine richtige Antwort, in meinem Fall brauche ich nur zwei Komponenten, um die Daten in einem zweidimensionallen Raum darzustellen.

4. Transformation der Daten

Im letzten Schritt wird der Prozess abgeschlossen, indem die Ursprungsdaten nach den gefundenen Richtungen (*Prinzipiellen Komponenten*), transformiert werden. Die Folgende Formel beschreibt die Operation:

$$T_L = V_L^T * Z^T, \tag{7.7}$$

wo T_L die transformierten, V enthält die L Prinzipiellen Komponenten und $emphZ$ beinhaltet den standardisierten Datensatz. Die erhaltene Matrix T hat die gewünschte L Anzahl an Komponenten.

QUELLEN: https://builtin.com/data-science/step-step-explanation-principal-component-analysis

https://royalsocietypublishing.org/doi/10.1098/rsta.2015.0202

https://en.wikipedia.org/wiki/Principal$_c omponent_a nalysis$

https://en.wikipedia.org/wiki/Sample$_m ean_a nd_c ovariance$

https://www.kaggle.com/jeffd23/visualizing-word-vectors-with-t-sne

https://towardsdatascience.com/visualizing-word-embedding-with-pca-and-t-sne-961a692509f5

https://towardsdatascience.com/visualization-of-word-embedding-vectors-using-gensim-and-pca-8f592a5d3354

https://studyflix.de/mathematik/eigenwert-1635

8

Zusammenfassung und Ausblick

In diesem Kapitel soll die Arbeit noch einmal kurz zusammengefasst werden. Insbesondere sollen die wesentlichen Ergebnisse Ihrer Arbeit herausgehoben werden. Erfahrungen, die z.B. Benutzer mit der Mensch-Maschine-Schnittstelle gemacht haben oder Ergebnisse von Leistungsmessungen sollen an dieser Stelle prntiert werden. Sie knnen in diesem Kapitel auch die Ergebnisse oder das Arbeitsumfeld Ihrer Arbeit kritisch bewerten. Wnschenswerte Erweiterungen sollen als Hinweise auf weiterfhrende Arbeiten erwt werden.

Literaturverzeichnis

AS. Amazon und Google Scientists: *Dive into Deep Learning.*

Hub21a. Hub, TensorFlow: *BERT multi cased Model: L=12, H=768, A=12*, 2021.

Hub21b. Hub, TensorFlow: *Tensorflow Hub Modelle*, 2021.

JDT19. Jacob Devlin, Ming-Wei Chang, Kenton Lee und Kristina Toutanova: *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.* Google Publication, 2019.

Mer. Merity, Stephen: *The WikiText Long Term Dependency Language Modeling Dataset.*

Rot00. Rothman, Denis: *Transformer for Natural Language Processing.* TODO:, 1000.

Vas17. Vaswanni, Ashish: *Attention Is All You Need.* Google Publication, 2017.

Web21. Website, TensorFlow: *Transformer model for language understanding*, 2021.

Wie21. Wierenga, Rick: *An Empirical Comparison of Optimizers for Machine Learning Models*, 2021.

A

Glossar

DisASTer	DisASTer (Distributed Algorithms Simulation Terrain), A platform for the Implementation of Distributed Algorithms
DSM	Distributed Shared Memory
AC	Linearisierbarkeit (atomic consistency)
SC	Sequentielle Konsistenz (sequential consistency)
WC	Schwache Konsistenz (weak consistency)
RC	Freigabekonsistenz (release consistency)

B

Erklng der Kandidatin / des Kandidaten

☐ ☐

☐ Die Arbeit habe ich selbststig verfasst und keine anderen als die angegebenen Quellen- und Hilfsmittel verwendet.

☐ Die Arbeit wurde als Gruppenarbeit angefertigt. Meine eigene Leistung ist ...

Diesen Teil habe ich selbststig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Datum

Unterschrift der Kandidatin / des Kandidaten