Informatik
Hauptcampus

HOCHSCHULE TRIER

Übersetzung in einfache Sprache mit Hilfe von Transformern

Translation into simple language with the help of transformers

Bachvarov, Vladislav

Master-Abschlussarbeit

Betreuer: Prof. Dr. Hans Beise

Trier, 25.05.2022

Kurzfassung

Diese Arbeit befasst sich mit dem Thema Natural-Language-Processing (NLP), präziser gesagt Transformer. In den ersten Kapiteln der Arbeit wird eine Einführung in das Thema Vektorrepräsentation von Wörtern gegeben. Nachfolgend werden die Modelle Transformer und BERT erklärt und schließlich wird die Struktur als Programmcode dargestellt. Das letzte und wichtigste Kapitel verschafft einen Überblick über Neural-Machine-Translation und die Anwendung solcher Modelle für die Übersetzung. Anschließend werden diese Modelle untersucht und eine Schlussfolgerung anhand der Ergebnisse wird gezogen.

Inhaltsverzeichnis

1

Einleitung und Problemstellung

In dieser Arbeit wird tief in der Natural-Language-Processing Sphere eingestiegen. Das Ziel ist, die Möglichkeiten der Transformer und BERT zu untersuchen und daraus ein Modell zu entwickeln, das die Aufgabe „Übersetzung in einfache Sprache mit Hilfe von Transformern„ angeht. Die Idee ist, vorhandene Strukturen von Transformern und BERT vorzustellen und deren Anwendebereich zu erweitern. Es existieren bereits unterschiedliche Transformer Modelle, die für die Übersetzung angewendet werden, z.B. [Zhu20], [Ten22c] und [MTLM15], um einige Beispiele zu nennen.

Die Aufgabe, die mit dieser Ausarbeitung gestellt wird, ist ein Übersetzungsmodell zu implementieren, das komplexere Sätze in einfache umwandelt. Im Mittelpunkt der Arbeit stehen Transformer und BERT, weitere Transformer-ähnliche Modelle werden in dieser Ausarbeitung nicht betrachtet (z.B. RoBERTa, Electra). Zum Trainieren der Modelle wurde ein Hochleistungsrechner auf dem 'Elweritsch'-Cluster an der TU Kaiserslautern [uTM] verwendet.

Die Arbeit beginnt mit einer Einleitung in Wortvektorrepräsentation. Die Kapitel vier und fünf befassen sich mit der Struktur und Implementierung des klassischen Transformers beziehungsweise BERT. Im letzten Kapitel befinden sich die Analyse und das Ergebnis.

2

Word2Vec

In diesem Kapitel wird das Verfahren "Word2Vec"durch die Nutzung von neuronalem Netz vorgestellt. Word2Vec ist eine Darstellung von Wörtern mit Vektoren, was auch die Abkürzung erklärt. *Word* steht für *Wort*, *2* steht für *to*, also zu, und *Vec* steht für *Vector*. Diese Wortrepräsentation wird in der Natural Laguage Processing (NLP) benutzt und findet Anwendung in Bereichen wie Spamfilterung und Dokumentenanalyse. Jedoch erklärt diese Technik nur, wie die Wörter eines Textes dargestellt werden können. Das Verfahren, bei dem die möglichst passenden Vektoren in einem ausgewählten Text, auch Corpus genant, gelernt werden, heißt Word Embedding. Bei dieser Technik wird ein neuronales Netz eigesetzt. Die Vorgehensweise und die Idee wird folglich erklärt.

2.1 Word Embedding

Wie bereits in der Einleitung erwähnt wurde, ist Word Embedding ein Prozess, bei dem die Wörter eines Textes in Vektoren umgewandelt werden.

In der Literatur spricht man von zwei Arten von Wort2Vec Methoden - SKIP-gram und CBOW (Continous Bag of Words) [Ali]. Beide Verfahren verwenden ein neuronales Netz. Die beiden Methoden unterscheiden sich nach ihren Ein- und Ausgaben.

SKIP-gram Model

Bei dem SKIP-gram-Modell fließen die Targetwörter als Eingabe und das Modell versucht ein Kontextwort zu raten. Hier ist die Struktur eines Skip-gram Modells gegeben [Ali]:

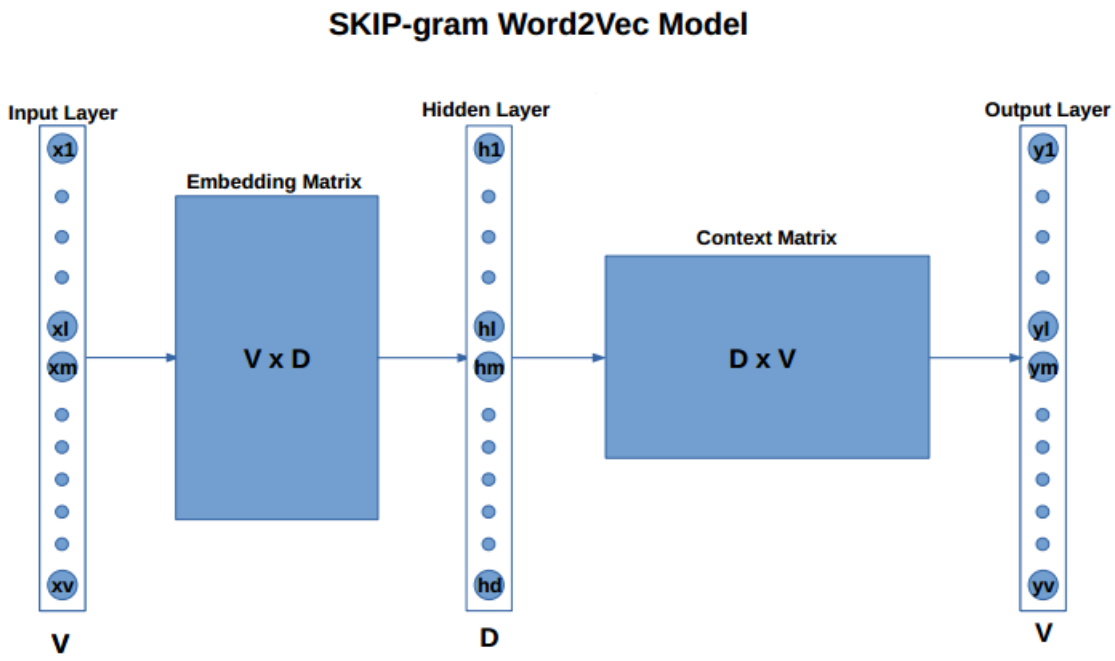


Abb. 2.1: Skip-gram Word2Vec Model

Der Abbildung 2.1 ist zu entnehmen, dass ein Skip-gram Modell aus einer versteckten Schicht, eine Ein- und eine Ausgabeschicht besteht. Die Eingabe sowie die Ausgabe ist ein Vektor, der aus V Komponenten besteht. V entspricht der Größe des Wörterbuchs (engl. Vocabulary). Die versteckte Schicht besteht aus D Variablen und stellt einen Vektor dar. D nimmt üblicherweise einen Wert zwischen 25 und 300. Diese Variablen können auch als Eigenschaften für die Wörter betrachtet werden. Je mehr Kriterien untersucht werden, desto besser können die Beziehungen zwischen den Wörtern dargestellt werden. Die Ausgabe ist wieder ein *V-dimensionaler* Vektor. Jedoch ist die Ausgabe kein One-Hot Vektor mehr, sondern ein Wahrscheinlichkeitsvektor, der die Wahrscheinlichkeiten für jedes Wort aus dem Corpus beinhaltet.

Die zwei Matrizen sind identisch, jedoch ist die Kontextmatrix die transponierte Embeddingsmatrix (Abb. 2.1). In der Matrix sind unsere Wortvektoren zu finden.

Die Multiplikation der Eingabe mit der Embedding-Matirx liefert den ausgeblendeten Schichtvektor. Eine weitere Multiplikation mit der Context-Matrix berechnet wiederum die Ausgabe.

CBOW Model

Umgekehrt fließen bei dem CBOW-Model die Kontextwörter als Eingabe, während das Targetwort geliefert wird. Die zwei Modelle besitzen die gleiche Anzahl an Schichten. Das CBOW-Modell ist ein umgedrehtes SKIP-gram-Modell, jedoch be-

steht die Eingabe aus w-Viele Vektoren statt nur einem, wo w das Kontextfenster darstellt. Als nächstes stellt Abb. 2.2 die beschriebene Struktur dar.

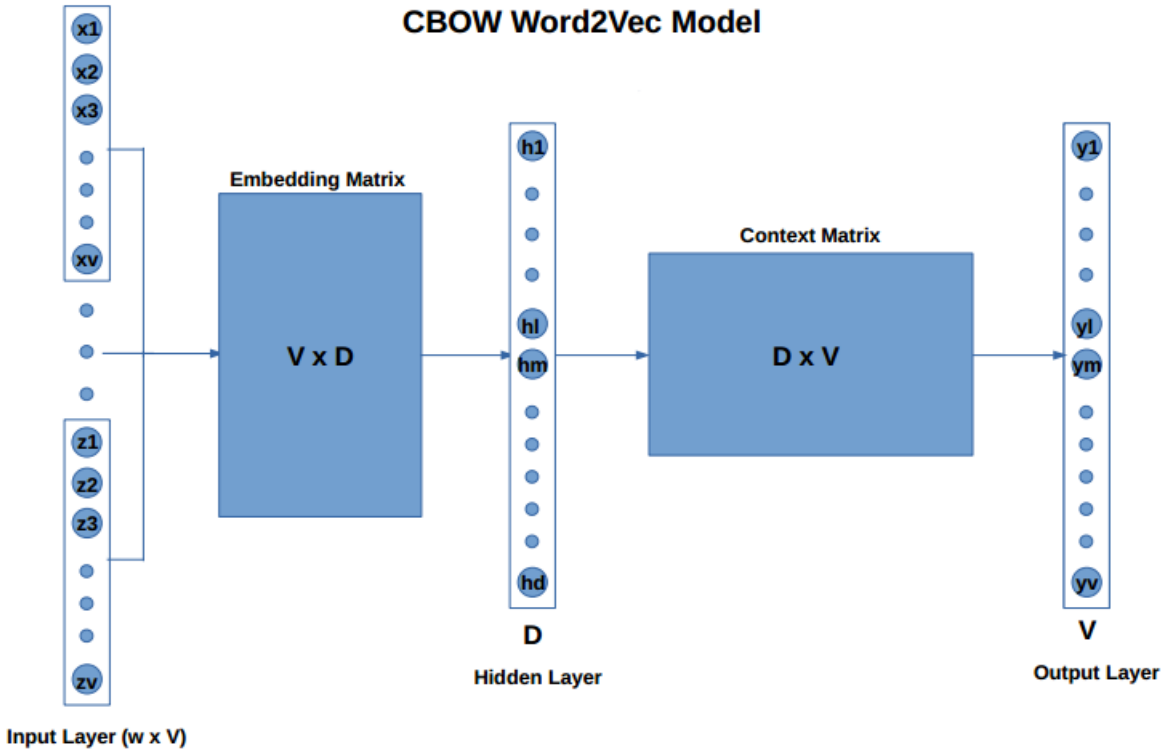


Abb. 2.2: CBOW Word2Vec Model

Die Struktur des CBOW-Modells besteht wieder aus einer Matrix für das Embedding der Target- und Kontextwörter. Die Größe der Matrizen hängt von den ausgewählten Hyperparametern und der Größe des Datensatzes ab. Die Anzahl der verwendeten Inputvektoren entspricht der Größe des gesetzten Fensters (Kontextfenster).

3

GloVe

Die Idee von **Glo**bal **Ve**ctors ist es, den Sinn hinter einem Wort in numerischer Vektorform darzustellen. GloVe ist eine weitere Möglichkeit neben word2vec(Kapitel 2) wie man Wörter oder sogar ganze Texte als Vektoren repräsentiert.

3.1 GloVe-Methode

Einige Variablen werden zuerst eingeleitet. Die Matrix der Word-Word-co-occurrence wird mit X notiert und eine beliebige Komponente 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 der folgenden Gleichung (3.1) definiert:

$$X_i = \sum_k X_{ik}.[\text{Uni}] \tag{3.1}$$

Schließlich definieren wir die Einheit P_{ij}, die beschreibt, wie hoch die Wahrscheinlichkeit ist, dass ein Wort j in dem Kontext vom Wort i vorkommt. Die Formel ist in der Gleichung (3.2) aufgeführt:

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

Ein kleines Beispiel wird angegeben, damit verstanden werden kann, wie bestimmte Aspekte aus dem gemeinsamen Auftreten von Wörtern gewonnen werden können. Es werden zwei Wörter i und j betrachtet, die in einem Corpus vorkommen. Das Beispiel stammt aus dem Artikel [Uni] und ist aus dem Themenbereich des thermodynamischen Zustands. Nehmen wir die Wörter $i = ice$ und $j = steam$. Die Beziehung der beiden Wörter kann so untersucht werden, indem die Relation mit anderen Probewörtern k berechnet wird. Für Wörter, die in direktem Bezug zu $i = ice$ stehen, erwarten wir, dass das Verhältnis $\frac{P_{ik}}{P_{jk}}$ groß ist, zum Beispiel wenn das Wort $k = solid$ gewählt wird. Analoge Beziehung kann bei Wörtern betrachtet werden, die näher an Bedeutung zu $j = steam$ stehen. In solchen Fällen werden wir einen kleineren Wert des Bruchs $\frac{P_{ik}}{P_{jk}}$ erhalten - zum Beispiel wählen wir das

Wort $k = gas$. Selbstverständlich ist der Wert des Bruchs gleich eins, wenn das Wort k zu beiden Wörtern *i, j* in Korrelation steht, oder wenn sich keine Korrelation ergibt. Als Beispiel können wir das Wort $k = sun$ wählen. Die Tabelle 3.1 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* [Uni]

Die Erwartungen werden durch die Tabelle 3.1 gerechtfertigt. Die Rate erlaubt uns, die Verhältnisse zwischen den einzelnen Wörtern besser zu verstehen. Mit Hilfe des Bruches wird unterschieden, wie die Wörter zueinander stehen, im Gegensatz zu der einfachen Wahrscheinlichkeit.

Zu betrachten ist, dass die Wahrscheinlichkeit der Koinzidenz von drei Eingangsgrößen *i*, *j*, *k*, abhängt. Die Allgemeinform der Funktion ist in der Gleichung (3.3) angegeben:

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

Die Wortvektoren für untersuchte Wörter *i,j* sind durch $w \in \mathbb{R}$ gegeben und das betrachtete Wort k ist mit Vektor $\tilde{w} \in \mathbb{R}$ repräsentiert. In der Gleichung entsteht die rechte Seite aus dem Corpus (Vocabulary). F hängt in diesem Fall von den drei Vektoren $w_i, w_j, \tilde{w}_k$ ab. F wird im nächsten Schritt 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 einer Vektordifferenz erfolgen. Somit fällt der Fokus nur auf die Funktionen, die von der Differenz der zwei Vektoren abhängen. Die Änderung wird aus der Gleichung (3.4) ersichtlich.

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

Für die Gleichung ist es zu erwähnen, dass die Parameter von F Vektoren sind, während die rechte Seite ein Skalar ist. Während F als eine komplexere Funktion, die von einem neuronalen Netz parametrisiert werden kann, genommen werden kann, würde das die Linearstruktur der Formel verschleiern. Es wird das Skalarprodukt genommen, damit die Verschleierung vermieden wird. Die Gleichung (3.5) stellt die Änderung dar.

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

Laut [Uni] erfolgt in der word-word Koinzidenzmatrizen der Unterschied zwischen Wort und Kontextwort willkürlich. Die Formel für F muss erlauben, dass

zwei Rollen beliebig ausgetauscht werden können. 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 erforderlich. Zuerst muss sichergestellt werden, dass die Formel homomorphisch zwischen den Gruppen $(\mathbb{R}, +)$ und $(\mathbb{R}_{>0}, \times)$ ist:

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

was nach Gleichung (3.5) wie folgt aussieht:

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

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

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

Zunächst wird die Gleichung umgestellt, jedoch werden einige Konstanten 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 der Gleichung (3.9) gegeben:

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

Nach der Einführung in Wortvektoren steigen wir in den nächsten Kapiteln ins Hauptthema Transformer ein. Die Folgekapitel erläutern die Struktur und Implementierung von dem Transformer und **B**idirectional **E**ncoder **R**epresentation from **T**ransformer (oder kurz BERT). Schließlich wird ein Modell vorgestellt, das versucht die Hauptaufgabe dieser Ausarbeitung zu lösen.

4

Der Transformer

In der zweiten Hälfte des Jahres 2017 veröffentlichte ein Team von Wissenschaftlern ihr Papier "Attention Is All You Need"[Vas17], in welchem sie ein neues Modell vorstellten. Das Projekt zur Entwicklung wurde unter Google Research and Google Brain aufgeführt. Dieses Modell bekam den Namen "Transformer".

4.1 Struktur des Transformers

Der Transformer besteht aus zwei Haupteinheiten, deren Namen Encoder und Decoder sind [Vas17]. Die beiden Einheiten bestehen aus gleicher Anzahl Encoder-, bzw. Decoder-Layers. Der im Artikel vorgestellte Encoder verfügte über 6 Encoderschichten und 6 Decoderschichten. Die Abb. 4.1 beschreibt den Transformer.

In den linken Schichten fließt der Input, meistens mehrere Sätze, durch eine Attention- und eine FeedForward-Network(FFN)-Schicht. Rechts werden die Targeteingaben, die zugehörigen Sätze für den Input, von zwei Attention- und von einer FFN-Unterschichten bearbeitet. Der Input- und der Targetsatz werden eingebettet, bevor sie in den Encoder, bzw. Decoder, eingegeben werden. Nachdem der Einbettungsvektor steht, wird dieser durch eine positionelle Kodierung beeinflusst. Das erfolgt, indem der Positionsvektor mit dem Einbettungsvektor zusammenaddiert werden.

Die nächsten Unterkapitel betrachten die einzelnen Aufbauelemente des Transformers im Detail. Wichtig ist es, anzumerken, dass die positionelle Einbettung von keiner Einheit im Transformer berechnet wird, sondern die Ausgabe einer Funktion ist. Diese ist im nächsten Kapitel erläutert.

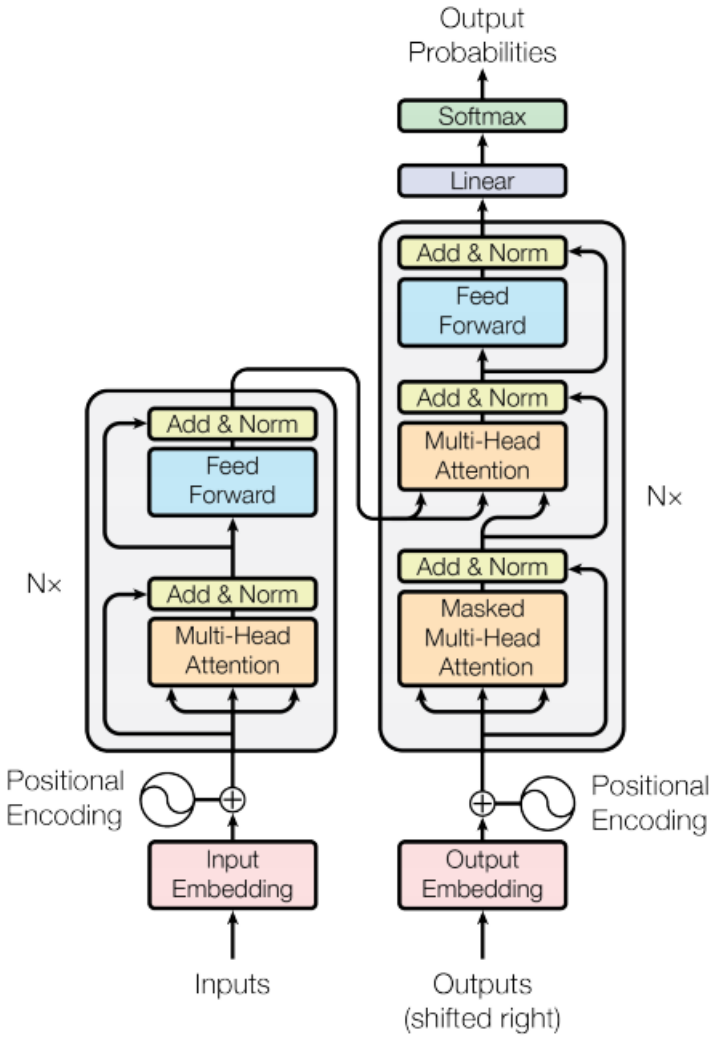


Abb. 4.1: Das Transformer-Modell [Vas17]

4.1.1 Positionelle Einbettung

Die positionelle Einbettung passiert nach der Umwandlung vom Text in Vektor. Wie die Bezeichnung verrät, wird Information über die Position des Wortes im Satz in den Wortvektor eingeschlossen. Diese zusätzliche Aktion ist eingesetzt, da der Transformer im Vergleich zu anderen Modellen oder Verfahren, kein Convolutional- oder Recurrent-Neural-Network beinhaltet.

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

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) [\text{Rot21}] \tag{4.1}$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)[\text{Rot21}] \tag{4.2}$$

Der *PE*-Vektor besteht aus d_{model}-Dimensionen, eine übliche Größe ist $d_model = 512$. Für jede Dimension wird der Wert des Eintrages entweder mit der Sinus- oder Kosinusfunktion berechnet. Die geraden Dimensionen werden mit der Sinusfunktion und die Ungeraden mit der Kosinusfunktion kodiert.

Als nächstes wird betrachtet, wie der PE-Vektor im Einbettungs-Vektor eingelagert wird. In der Literatur werden die Vektoren ganz einfach addiert. Diese Vorgehensweise könnte jedoch Probleme verursachen und wichtige Informationen könnten verloren gehen. Es existieren mehrere Möglichkeiten, wie man den Verlust von Informationen vermeidet. In der Quelle [Rot21] wird folgende Formel benutzt:

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

wo der Eingabevektor y_i durch eine Konstante $math.sqrt(d_{model})$ skaliert wird. Somit ist die Information aus der Einbettung verstärkt und der Einfluss des PE-Vektors vermindert. Die Variable y_i ist der eingebettete Vektor, während *pe* der Positionalsvektor und d_{model} die Kardinalität eines Vektors ist. Der Eingabevektor wird mit der Wurzel von d_{model} skaliert und erst dann wird der Positionalvektor addiert.

4.1.2 Encoder und Decoder

Encoder und Decoder sind die essenziellen Bestandteile des Transformers. Der Encoder erhält die Einbettung und führt sie durch ein Attention- und ein Feed Forward Network-Layer. Zwischen jeder Unterschicht besteht eine residierte Verbindung (siehe Abb. 4.1), sodass die Ausgaben von der letzten Unterschicht mit den Ausgaben von der Aktuellen addiert und weiterhin normiert werden. Der Decoder besitzt eine zusätzliche Unterschicht im Vergleich zum Encoder (Abb. 4.1). Im Transformer beihaltet der Decoder drei Schichten (siehe Abb. 4.1). Die letzten zwei sind die selben wie im Encoder. Die erste Teilschicht im Decoder ist eine Masked-Multi-Head-Attention-Schicht. Die Verbindung zwischen Encoder und Decoder erfolgt in der zweiten Unterschicht - nämlich in der MHA-Schicht. Da werden die Ausgaben vom Encoder und vom MMHA-Schicht zusammengeführt. Nach der Bearbeitung liefert das Model einen softmax-Vektor, der abhängig von der Aufgabenstellung die gesuchte Antwort sein sollte. In den folgenden Unterkapiteln werden die Unterschichten im Detail untersucht.

Multi-Head-Attention Layer

Der Multi-Head-Attention Layer kommt in beiden Einheiten vor. Dieser Schicht folgt eine Masked-Multi-Head-Attention im Decoder und die Einbettungsschicht im Encoder.

Die Eingabe in dem Multi-Head-Attention-Layer vom Encoder ist der Vektor, der die positionellen und eingebetteten Angaben vom Text erhählt. Der Unterschied in der Schicht in Encoder und Decoder entsteht durch ihre Eingaben. Die

Information ist für den Encoder komplett verfügbar, und im Decoder wird diese maskiert, und so lernt das Modell eine Prognose zu erstellen.

Ziel des MHA-Layer im Encoder ist es, die Beziehung zwischen einzelnen Wörtern zu bestimmen. Das wird erzielt, indem jedes Wort aus dem Satz mit allen anderen abgebildet wird. Das Problem dabei ist die Größe des Wortvektors, weil er eine Kardinalität von d_{model} besitzt. In der Quelle [Rot21] erhält d_{model} den Wert von 512. Die große Anzahl der Dimensionen würde große Laufzeit anfordern, wenn mehrere Ansichten untersucht werden. Das bedeutet wiederum, dass mehr Leistung erfordert wird, um weitere Ansichten zu bestimmen. Eine bessere Alternative ist es, die Dimensionen jedes Wortes, bzw. Satzes, in 8 Teile zu zerlegen, dabei besteht jeder Teil aus 64 Dimensionen. In [Rot21] ist jeder Teil als Head bezeichnet, daraus entsteht auch der Name **Multi-Head**-Attention-Layer.

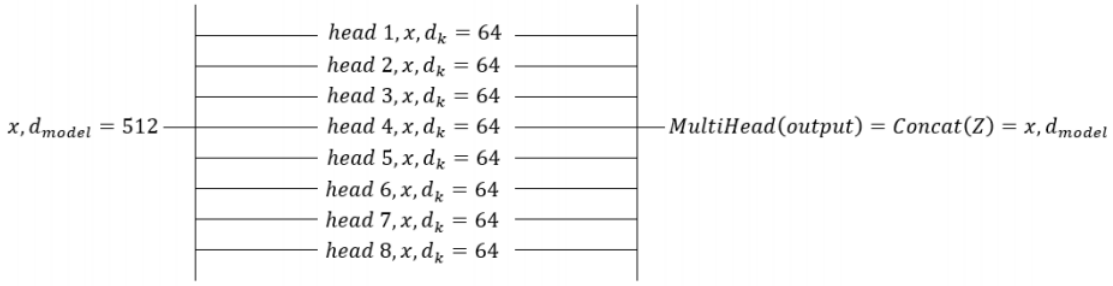


Abb. 4.2: Heads [Rot21]

Diese Heads laufen parallel. Der Vorteil dabei ist es, dass die Laufzeit verringert wird und 8 unterschiedliche Repräsentationen betrachtet werden. In der Abb. 4.2 ist es zu erkennen, 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:

$$Z = (z_0, z_1, z_2, z_3, z_4, z_5, z_6, z_7)[\text{Rot21}] \tag{4.4}$$

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 [Ten]:

- 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 der anderen Wortvektoren sucht, inklusive sich selbst bei Selbstattention.
- Einem Schlüsselvektor (auch als Key-Vektor bezeichnet K), der mit dem Ziel angepasst wird, einen Attention-Wert zu liefern.
- Einem Wertvektor (auch als Value-Vektor bezeichnet V), der darauft trainiert wird, einen weiteren Anttention-Wert bereitzustellen.

Die Quelle [Rot21] bezeichnet die Attention als „Scaled Dot-Product Attention-von. Das ist eine Linearkombination der oben deklarierten Vektoren. Die folgende Formel aus [Rot21] stellt die Berechnung dar:

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

Diese Vektorrepräsentationen werden aus den Gewichtsmatrizen eingelesen. Die Gewichtsmatrizen sind am Anfang nicht bekannt und werden im Folge des Trainings bestimmt. Zu Beginn werden sie mit zufälligen Werten erstellt. Die Matrizen werden im Buch [Rot21] 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 das Wort x_n abzulesen ist, dann erfolgt das durch eine einfache Matrixmultiplikation, wobei x_n in diesem Fall den Indexwert vom Wort x_n repräsentiert:

$$Q_{x_n} = x_n * Q_w \tag{4.6a}$$

$$K_{x_n} = x_n * K_w \tag{4.6b}$$

$$V_{x_n} = x_n * V_w \tag{4.6c}$$

Wenn die Eingabe aus mehreren Wörtern besteht, wird eine Matrix mit den Dimensionen *Anzahl der Wörtern*$\times d_{model}$ (für d_{model} meistens 512 gewählt) erhalten.

Für jeden Eintrag in x wird eine Matrix mit ihren Attention-Vektoren, bzw. den Beziehungsvektoren zu jedem Eingabewort x_n (inklusive sich selbst) berechnet. Jeder Vektor in der Matrix hat 512 Einträge, und es gibt insgesamt m-viele Vektoren ($m - viele$ Wörter). Der 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 Attention-Schicht. Diese bekommt als Eingabe die konkatenierte Ausgabe von Matrix Z_{concat} und die unveränderten Eingabedaten der MHA-Attentionschicht x:

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

Mittels der Normierungsschicht werden dann folgende Berechnungen ausgeführt:

$$LayerNorm(v) = \gamma * \frac{v - \mu}{\delta} + \beta.\text{[Rot21]} \tag{4.8}$$

Die Variablen bedeuten Folgendes:

- μ ist der Durchschnitt von v mit Dimensionen d:

$$\mu = \frac{1}{d}\sum_{k=1}^{d} v_k.\text{[Rot21]} \tag{4.9}$$

- δ ist die Standardabweichung von v mit Dimensionen d:

$$\delta^2 = \frac{1}{d}\sum_{d}^{k=1}(v_k - \mu)^2.\text{[Rot21]} \tag{4.10}$$

- 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.

Gleichermaßen sieht die Funktionalität der MHA-Schicht im Decoder aus. Die Eingabe in der Schicht besteht 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 jedes Wort einzeln anzuwenden. Die Anwendung ist identisch, jedoch mit unterschiedlichen Parametern.
- Das Netzwerk besteht aus zwei Lineartransformationen und eine ReLU-Aktivierung 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 übereinstimmend. 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 gleichen Aufbau wie die MHA-Schicht. Diese Schicht unterscheidet sich von dieser im Encoder darin, dass die Eingabe "maskiert" wird. Das bedeutet, dass bestimmte Abschnitte ausgeblendet werden, sodass der Layer begrenzte Informationen erhält. Ziel der Maskierung ist es, das Modell 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 Modells werden 6 Klassen gebraucht. Diese sechs Klassen beschreiben die wichtigsten Komponenten des Transformers und das Modell selbst. Es gibt zwei Klassen für den Encoder und Decoder, zwei für die Layers im Encoder und Decoder, die Klasse für das Modell und eine Klasse für den Attentionlayer. Für einen besseren Überblick wird hier die Hauptklasse des Transformers vorgestellt:

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. Die Dense-Schicht repräsentiert ein komplett verbundenes Netzwerk mit *target_vocab_size*-vielen Neuronen. Die Ausgabe davon stellt die Prognose des Modells dar. Für jedes einzelne Wort im Satz wird der Decoded-Vektor in einen anderen Vektor mit Dimensionalität *sequence_length* × *target_vocab_size* umgewandelt. Für jede Stelle im Satz liefert die Dense-Schicht einen Vektor, der so groß ist wie die Anzahl der einzigartigen Wörter im Targetcorpus. Jeder Eintrag in diesem Vektor stellt die Wahrscheinlichkeiten dar, dass das Wort an der Stelle platziert werden soll. Somit ratet das Modell. Der Encoder und Decoder werden mit den bestimmten Hyperparametern initialisiert. Im beschriebenen Fall wird ein Transformer mit folgenden Hyperparametern verwendet:

Listing 4.2: Hyperparameter

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

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

Encoder

Die Encoder Klasse besteht aus mehreren Encoder-Schichten und aus der entsprechenden Embedding-Schicht. 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 Worteinbettung und die zweite für die positionelle Einbettung. In der dritte Zeile werden alle Encoderschichten erstellt. Am Ende wird eine Dropout-Schicht angehängt, die dabei Hilft, dass das Modell nicht übertrainiert (overfitted) wird.

Decoder

Die Decoder-Klasse wird ebenso in einer separaten Klasse ausgelagert. Die Programmzeilen 4.4 widerspiegeln 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)]
```

In der Zeile 3 aus dem Listing 4.4 werden die Decoder-Schichten erstellt. Ebenso werden in den Zeilen 1 und 2 die zwei Einbettungsoperationen für die Einbettungsschicht definiert. Die Embedding-Schicht unterscheidet sich von der Einbettungsschichten im Encoder, im Sinne von Variablen, und spielt die Rolle einer Look-up-Tabelle für das Targetcorpus.

5

BERT

Das folgende Kapitel beschäftigt sich mit der Struktur der **B**idirectional **E**ncoder **R**epresentations from **T**ransformers. Wie aus dem Namen zu schließen ist, basiert BERT auf dem Transformer (4).

5.1 Struktur von BERT

BERT basiert auf dem Transformer [JDT19]. Das BERT-Model nutzt die Hauptcharakteristik des Encoders, Beziehungen zwischen den Wörtern im Corpus zu erlernen. Im Vergleich zum Transformer, wo zwei Einheiten zusammenarbeiten, ist in BERT nur die Encoder-Einheit nötig, weil nur ein Sprachenmodel erstellt werden soll, wie beim Transformer werden die Eingaben vorverarbeitet. In dieser Hinsicht haben die beiden Modelle kleine Unterschiede. In BERT wird nicht nur Information über die Position der einzelnen Wörter eingelagert, sondern auch die Angaben über die Segmente der Eingabe (falls eine Eingabe aus mehreren Segmenten, bzw. Sätzen besteht) werden integriert. Außerdem werden diese im BERT-Model als Hyperparameter initialisiert. Das bedeutet, dass alle Positional- und Segmentvariablen erlernt werden müssen. Diese kleine Änderung erfordert ein unterschiedliches Vorgehen beim Lernen des Modells. Der Lernprozess von BERT 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 Hyperparameter des Einbettungslayers. In dieser Phase soll die Token-Tabelle (Tokenizer) bereits verfügbar sein. Die zweite Phase des Lernens versucht, die vorgegebene Task zu lösen. Für das beste und schnellste Ergebnis sollte für die zweite Phase derselbe Tokenizer verwendet werden. Jedoch ist diese Anforderung kein Muss, denn die Anwendung von zwei getrennten Tabellen würde zu längeren Lernzeiten führen. Durch die große Anzahl an vortrainierten Modellen im Netz lohnt es sich, ein Modell wiederzuverwenden und für die eigene Task anzupassen. In den folgenden Kapiteln werden beide Phasen aufgeführt. Es wird zuerst der Prozess der Vorerlernung vorgestellt und danach wird erklärt, wie ein vortrainiertes Modell für die gewünschte Task angepasst werden kann.

5.2 Implentierung

Die folgenden drei Kapitel stellen das BERT-Modell detaillierter vor und geben eine mögliche Implementierung des Modells mit der Bibliothek Tensorflow von Google. Die Implementierung basiert auf Publikationen [JDT19], [Ten22a] und [Ten22b]. Im Paper [JDT19] aus dem 2019 wird ein neues Vorgehen im Bereich des Natural Language Processing vorgestellt. Die Autoren stellen eine bessere Alternative zur Fineeinstellung der Transformermodelle vor. Im Artikel wird die unidirektionale Beschränkung des Transformers verbessert, indem ein Masked-Language-Model (MLM) als Vortrainierungsziel verwendet wird. Das verwendete Modell maskiert zufälligerweise Tokens aus der Eingabe und die Idee ist, diese Token-IDs aus dem Kontext herzuleiten. Dieses Ziel verbindet den linken und rechten Kontext vom Satz und der trainierte Transformer ist bidirektional. Die Autoren kombinieren dieses Ziel mit dem Next-Sentence-Prediction-Ziel (NSP), sodass sie zwei Tasks gleichzeitig verarbeiten. Das Ziel der zweiten Aufgabe ist es vorauszusagen, ob zwei gegebene Sätze im Text nacheinander vorkommen.

In der zweiten Phase wird die Task nach Bedarf ausgesucht. Im vorliegenden Fall ist das eine Übersetzung aus dem Englischen ins Portugiesische. Die Task, die später gelöst werden soll, muss nicht vom Pre-Training abhängen. Das bedeutet, dass eine Zieländerung des vortrainierten Modells immer möglich ist, sobald ausreichend 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 Modell. In der vorliegenden Ausarbeitung werden dieselben Mechanismen verwendet, die im Atrikel [JDT19] vorgestellt werden. Das heißt, dass zwei Tasks (NSP und MLM) gleichzeitig gelöst werden. Das Masked Language Model wurde im vorigen Abschnitt 5.2 eingeleitet, aber in diesem Kapitel wird der Prozess näher vorgestellt. Die Eingabedaten werden speziell für das Modell vorbereitet. Laut dem Artikel [JDT19] werden 15% der Wörter im Batch verborgen. Das bedeutet, dass höchstens 10 Wörter bei einer Größe des Batches von 64 und 5 Wörter bei einer Größe von 32 maskiert werden. Jedes Wort aus dem Batch hat eine gleich hohe Wahrscheinlichkeit von 10% entweder maskiert oder durch ein weiteres Wort ersetzt zu werden. In 80% der Fälle bleibt das Wort erhalten.

Beim Next-Sentence-Prediction ist die Aufgabe vorherzusagen, ob beide Sätze kontextuell verbunden sind. Hier werden in der Hälfte der Fälle zwei aufeinander folgende Sätze genommen. Die weiteren Paare sind zwei kontextuell unterschiedliche Sätze. In diesem Sinne gelten die eingebetteten Sätze als Input und haben 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 englischer Sprache. Außerdem besteht es aus mehr als 100 millionen Tokens. Die Trainingsdatei besteht aus mehr als 35 Tausend Zeilen Text. Der Datensatz ist im Literaturverzeichnis verlinkt und kann heruntergeladen werden. Als Erstes werden die Daten aus dem Corpus fürs Training vorbereitet.

Die Klasse Wiki2Corpus bereitet das ganze Corpus vor. Zuerst werden alle Wörter in Zahlen mit Hilfe eines Tokenizers verwandelt. Dafür wird aus der Bibliothek *d2l* [AS] der Tokenizer genutzt. Als nächstes wird die Vocabulary erstellt, damit Inferenzen des Modells später in Worte verwandelt werden. Der Wortschatz ergibt sich aus dem gesamten 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. Ein Programmcode, der 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 nötige 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 werden. Die neu erzeugten Datensätze müssen mittels Schlüsselwörter abgegrenzt werden. Die Schlüsselwörter ['CLS'],

['SEP'], ['MASK'] und ['PAD'] müssen an die entsprechenden 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 verwendet. Die Einsetzung der Schlüsselwörter kann sowohl vor der Erstellung des Samples als auch nachher erfolgen. Die neu erzeugten NSP-Satzpaare 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)
```

Die Vorbearbeitung ist nun abgeschlossen und die Trainingsdaten können verwendet werden. Der nächste Schritt ist die Erstellung des Modells.

BERT Class

Die vorliegende Arbeit basiert auf dem im [JDT19] vorgestellten Modell. Für die gestellte Aufgabe werden folgende Hyperparameter definiert:

Listing 5.5: Die Hyperparametern vom BERT

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

Diese Konfiguration entspricht dem $BERT_{BASE}$-Modell, das im Artikel [JDT19] vorgestellt wird. Es wird zusätzlich das $BERT_{LARGE}$-Modell definiert, das durch folgende Werte charakterisiert wird: $num\_layer = 24$, $d_{model} = 1024$, $num\_heads = 12$. Das Base-Modell besitzt 110 Mio. Parameter, während das große Modell dreimal so viele Parameter aufweist - insgesammt 340 Mio. Dadurch ist das Trainieren eines BERT-Models sehr anspruchsvoll, jedoch empfiehlt es sich, auf ein vortrainiertes Modell zurückzugreifen, da auf diese Weise der Arbeitsaufwand um die Hälfte reduziert werden kann.

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

Listing 5.6: BERT-Struktur bei Pre-Training

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

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

Listing 5.7: Definition des BERT-Encoder-Layers

```
# BERTEncoderLayer.py
# The three embedding layers
self.embedding = Embedding(input_vocab_size, self.d_model)
self.segment_encoding = Embedding(2, self.d_model)
self.pos_encoding = tf.Variable(
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:

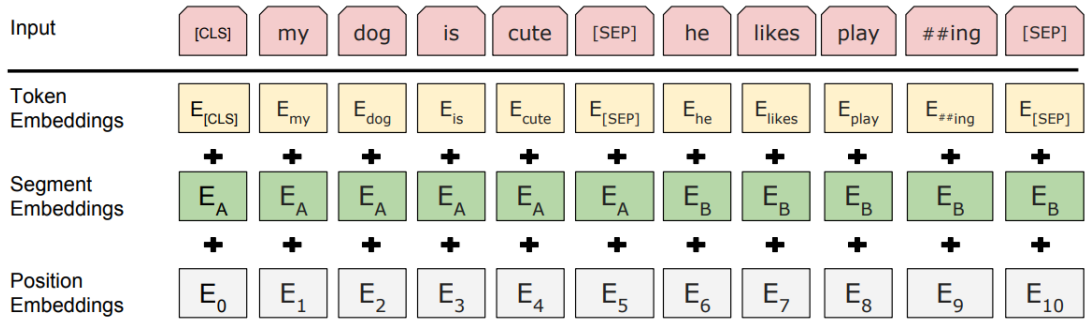


Abb. 5.1: BERT Eingabeschicht [JDT19]

Aus der Abbildung 5.1 kann man nicht nur die Funktion der Schichten lesen, sondern auch ihre Eingaben beobachten. In der Token-Embedding-Schicht werden die Tokens der Wörter gelesen und die Schicht liefert die zugehörige Vektor-Repräsentation. Die Segment-Embedding-Schicht erhält den Segmentenvektor, der bei der 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. Für den vorliegenden Zweck wird der Adaptive-Movement-Optimizer verwendet. Laut [Wie21] liefern Adam und RMSProp (**R**oot **M**ean **S**quare **Prop**agation) höhere Richtigkeit und niedrigere Fehlerrate als die anderen Optimizers, wie SGD(**S**tochastic **G**radient **D**escent) und AdaGrad(**Ada**ptiv **Gra**dients). Bei unserer ersten Task (Masked Language Processing) wird versucht, die maskierten

Wörter zu bestimmen. Das impliziert, dass jedes maskierte Wort mehreren Klassen zugehören kann, bzw. aus vielen Wörtern bestehen 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 aufweist. Während dessen ist *SparseCategoricalCrossentropy* besser geeignet, wenn die Ausgabe nur einen Integer-Wert beinhaltet bzw. daraus besteht. Dieser Integer stellt die zugehörige Klasse dar. Welche Fehlerfunktion verwendet wird, hängt nur von der Form der Ausgabe ab. Die vorliegende Arbeit beschäftigt sich überwiegend mit *CategoricalCrossEntropy*, da die Ausgaben ein Softmax-Vektor sind. Später wird bei der Anpassung 5.2.3 wird *SparseCategoricalCrossEntropy* verwendet.

5.2.3 Fine-Tuning von einem BERT-Modell

Das Modell für das Fine-Tuning wurde dem TensorflowHub [Hub21b] entnommen. Für diesen Zweck wird ein Modell mit den gleichen Hyperparametern, wie das im Unterkapitel 5.2.2 vorgestellte $BERT_{BASE}$ verwendet. Auf TensorflowHub stehen mehrere Modelle zur Wiederverwendung, hier unter anderem BERT-Multilangual-Cased-Modell, BERT-English-uncased, sowohl das Base-BERT-Modell als auch das Large-Modell. Dementsprechend kann das geeignete Modell für die Task ausgewählt werden. Im aktuellen Fall soll ein Translationsmodell aufgebaut werden. Für diesen Zweck wird das Multilingual-BERT-Modell [Hub21a] genutzt.

BERT-Modell

Das gelernte BERT-Modell kann wie folgt geladen werden:

Listing 5.8: Laden von dem BERT-Modell

```
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 Modell kann als eine Schicht geladen und somit in dem Modell angefügt werden. Für den Lernprozess braucht das Modell eine weitere Schicht, die die Prognose darstellen soll. Dafür bietet es sich an, eine einfache Dense-Schicht am Modell anzuhängen, die dann die Ausgabe des BERT-Modells mit einem Fully-Connected-Layer verbindet. Die Rolle der Dense-Schicht ist es, die Ausgabe in die richtige Shape zu verwandeln. Die Ausgabe vom BERT-Modell ist ein Vektor mit der Kardinalität *batch_size* $\times$ *sequence_length* $\times$ *d_model*. Die Dense-Schicht transformiert den Ausgabevektor in die Kardinalität *batch_size* $\times$ *sequence_length*

× *vocab_size*. Die *sequence_length* entspricht der Länge des Satzes und jeder Index beschreibt ein Wort im Satz. Die Variable *d_model* repräsentiert die Anzahl der Eigenschaften, nach denen Wörter klassifiziert werden. Die Dense-Schicht verbindet somit die Eingeschaften der einzelnen Wörter zu einem Neuron, wobei die Ausgabe des Neurons ein Wert ist, der für oder gegen ein bestimmtes Wort aus dem Wortschatz spricht. Ein weiterer Schritt ist notwendig, damit diese Werte in Wahrscheinlichkeiten umgewandelt werden können, was nämlich durch die Anwendung der Softmax-Funktion erfolgt. Das heißt, dass das Modell aufgefordert wird, eine Prognose zu erstellen, wie die Übersetzung des Eingabesatzes lauten wird. Hier ist das Modell gegeben:

Listing 5.9: Definition des BERT-Modells 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 einer Frage-Antwort-Task. Deswegen muss die Datensammlung im Voraus so vorbereitet werden, dass jeder Satz und seine Übersetzung gruppiert werden. Für diese Task wird der Datensatz */pt_to_en* von der Datensatzsammlung *ted_hrlr_translate* verwendet. Der Datensatz ist über die Bibliothek *tensorflow_datasets* zugänglich. Die Datensammlung enthält Sätze auf Englisch und Portugiesisch, also wird das Corpus für die Übersetzung aus dem Englischen ins Portugiesische genutzt. Die Datensammlung besteht jeweils aus 51785 Sätzen in Englisch und Portugiesisch, insgesamt also aus 103570 Sätzen. Die Sätze,

die die erlaubte Sequenzlänge überschreiten, werden verworfen. Die ausgewählte Sequenzgröße beträgt 128 Wörter. Da die höhere Satzlänge eine längere Lernzeit bedeutet, wurden in Bezug auf die Hardware, die zur Verfügung steht, eine Sequenzlänge von 128 Wörtern, einschließlich Schlüsselwörter '[CLS]' und '[SEP]', ausgewählt. 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-Modell 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 Adam verwendet, der über die üblichen Charakteristiken hinaus die Benutzung eines Schedulers für die Lernrate ermöglicht. Das bedeutet, dass die Rate im Laufe des Trainings angepasst werden kann. 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 steps
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')
12 # loss function
13 tf.keras.losses.SparseCategoricalCrossentropy(from_logits=
       ↪ True)
```

Der Optimizer wird durch die Bibliothek *official.nlp.optimization* bereitgestellt. Mit der Einstellung in dem Listing 5.11 wird der Optimizer konfiguriert, dass er für *num_warmup_steps*-viele Iterationsschritte die Lernrate aufsteigen darf. Nach dem Ablauf der Schritten wird die Lernquote wieder gesunken.

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

6

NMT-Basierte Modelle

In den zwei vorangegangenen Kapiteln wurde die Struktur von BERT 5.1 und Transformer 4.1 vorgestellt. Das Kapitel 6 beschäftigt sich mit Neural-Machine-Translation-Modellen (NMT), mit denen versucht wurde, Sätze in einfache Sprache zu übersetzen. Die Struktur der Modelle wird zuerst dargestellt und anschließend wird die Vorführung gegeben, wie diese Modelle in Python eingesetzt werden können. Zum Schluss werden die Erkenntnisse für jedes Modell zusammengefasst.

6.1 BERT Neural Machine Translation Model

In [Zhu20] wird ein Modell vorgestellt, das auf dem üblichen Transformer basiert und im wessen Kern BERT und Neural Machine Translation (NMT) Model zusammenarbeiten. Das NMT-Model ist der bekannte Transformer (siehe Kapitel 4). Im oben genannten Artikel wird das in dieser Arbeit verwendete Modell als BERT-fused Model beschrieben, jedoch wird es für die vorliegenden Zwecke BERT-Neural-Machine-Translation Modell, oder BNMT, genannt.

6.1.1 Struktur des BNMT-Model

Die Struktur des Modells ist durch die folgende Abb. 6.1 gegeben.

Die Autoren des Artikels [MTLM15] schlagen vor, dass die Eingabe vorerst in ein BERT-Modell fließt, dessen Ausgabe später in den Encoder und Decoder des Transformer-Models verläuft. Der Flussverlauf von Informationen wird im [Zhu20] in drei Schritte aufgeteilt.

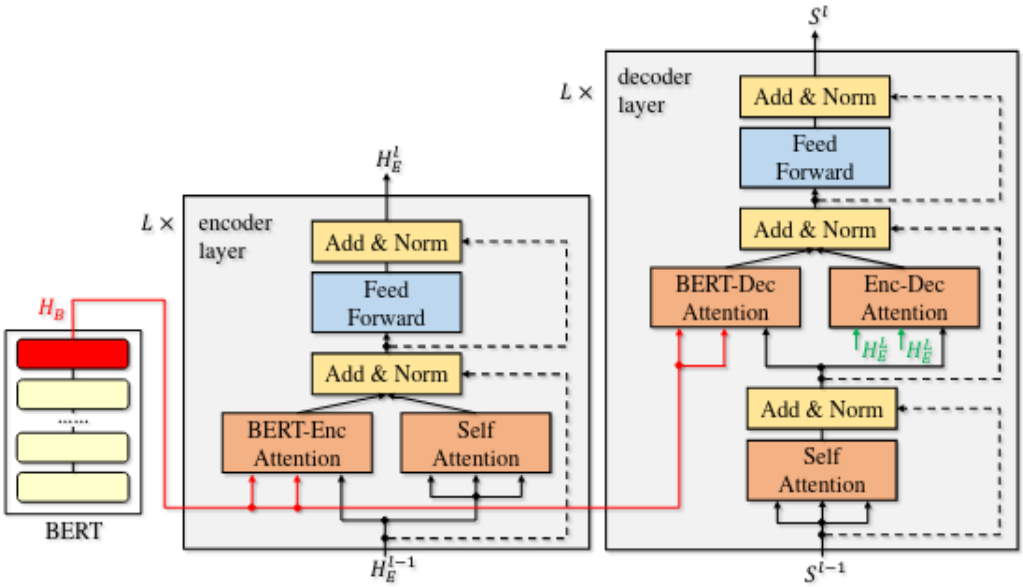


Abb. 6.1: BNMT-Model Struktur [Zhu20]

Vorerst werden einige Notationen eingeführt. Die Datensätze für die Eingabe- sowie Targetsätze werden durch X bzw. durch Y markiert. Für jeden beliebigen Satz $x \in X$ und $y \in Y$ bezeichnet l_x und l_y die Anzahl der Wörter in diesen Einträgen. Die einzelnen Elemente in x oder y werden entsprechend ihrer Position mit x_i bzw. y_i gekennzeichnet.

1. Schritt 1: Bei einer Eingabe $x \in X$, wo X die Menge der Eingaben ist, bettet BERT den x-Vektor als $H_B = BERT(x)$ ein, wobei H_B die Ausgabe des letzten Layers vom BERT ist, während $h_{B,i} \in H_B$ das i-te Element der Ausgabe darstellt.
2. Schritt 2: Die Variable H_E^l wird genutzt, um die Darstellung der versteckten l-ten Schicht zu beschreiben, wobei H_E^0 die Einbettung der Sequenz x ist. Das 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 wie folgt gegeben, dabei sind $attn_S$ und $attn_B$ die Attention-Modelle bzw. MHA-Schichten mit unterschiedlichen Parametern (die Definition ist in Abschnitt 4.1.2 zu finden):

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

   Weiterhin werden die berechneten Attentions $\tilde{h}_i^l$ nach ihrer Normierung dem Feed-Forward-Network eingeflossen und schließlich wird die Ausgabe standardisiert. Die Ausgabe aus der letzten Schicht ist durch H_E^L definiert.
3. Schritt 3: Die Eingabe im Decoder-Layer l wird durch $S_{<t}^l$ für den Zeitpunkt t, $S_{<t}^l = (s_1^l, , s_{t1}^l)$ bezeichnet. s_1^0 ist ein besonderes Zeichen, das den Anfang

der Sequenz bestimmt und s_t^0 ist die Voraussage des Modells zum bestimmten Zeitpunkt t. In der Schicht l erfolgt Folgendes:

$$s^l t = attn_S(s^{l1}t, S^{l1}_{<t+1}, S^{l1}_{<t+1}) \tag{6.2a}$$

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

Die $attn_S$, $attn_B$ und $attn_E$ stellen entsprechend Self-, BERT-Decoder- und Encoder-Decoder-Attention dar. Die Gleichung (6.2) führt eine Iteration über die Schichten und liefert das Ergebnis s_t^L. Die Ausgabe fließt durch eine Dense-Schicht und wird schließlich von einer Softmax-Funktion transformiert, was dazu führt, dass das t-te Wort $\hat{y}_t$ prognostiziert werden kann. Diese Schritte iterieren, bis ein Endzeichen erreicht wird, bzw. bis das Ende der Sequenz verarbeitet wird.

6.1.2 Implementierung

Als Nächstes wird ein Modell vorgestellt. Das Modell verwendet BERT für die Einbettung zu Beginn des Prozesses. Für die Implementierung werden die Bibliotheken *tensorflow* und *transformers* benutzt.

Das Modell wird in einige Elemente aufgeteilt. Manche Komponenten werden von der Tensorflow-Bibliothek bereitgestellt. Die Klasse des Modells ist gegeben:

Listing 6.1: BNMT_Model

```
1     # 1. Layer (BERT-Model)
2     self.bert_layer = transformers.TFBertModel.from_pretrained(
          ↪ model_name)
3     # 2. Layer Encoder Embedding layer
4     self.enc_embedding_layer = Embedding(vocab_size,
          ↪ hidden_layer_size, input_length=seq_len)
5     # 3. Layer Encoder
6     self.encoder_layers = [NMTEncoderLayer(hidden_layer_size,
          ↪ transformer_heads, 1e-3) for _ in range(
          ↪ transformer_heads)]
7     # 4. Decoder Embedding
8     self.decoder_emb = Embedding(vocab_size, hidden_layer_size,
          ↪ input_length=seq_len)
9     # 5. Layer (Decoder)
10    self.decoder_layers = [NMTDecoderLayer(hidden_layer_size,
          ↪ transformer_heads, 1e-3) for _ in range(
          ↪ transformer_heads)]
11    # 6. Layer (Translation layer)
12    self.dense_layer = tf.keras.layers.Dense(vocab_size)
13    self.softmax_layer = tf.keras.layers.Softmax()
```

Somit darf das Modell in sechs größere Komponenten zerlegt werden. Die erste Komponente erledigt die BERT-Einbettung. Zu diesem Zweck wird ein vortrainiertes BERT-Modell verwendet, das explizit an die deutsche Sprache angepasst ist. Das Modell ist von den Entwicklern von deepset.ai [dee19] bereitgestellt worden. Es ist ein standard BERT-Base Modell (Listing 5.5). Zeilen 4 bis 10 im Listing 6.1 beschreiben die Encoder- und Decoder-Modelle. Die letzten zwei Zeilen (12 und 13) beschreiben die Inferenz des Modells (Übersetzungsschicht). Die Ausgabe vom Decoder wird durch die Dense-Schicht erweitert, sodass sie die vorliegende Datensammlung repräsentiert. Als letztes wird eine Softmax-Funktion angewendet und als Ergebnis wird die Wahrscheinlichkeit für jedes Wort aus der Datensammlung, in der Sequenz vorzukommen, berechnet.

Die Encoder und Decoder im NMT-Modell sind modifiziert, um die Vorteile des vortrainierten BERT-Modells zu nutzen. Nach der Abb. 6.1 und den Formeln aus Abschnitt 6.1.1 werden die beiden Schichten wie folgt eingesetzt:

- Der Encoder:

Listing 6.2: BNMT_Encoder

```
1       self.self_mha = NMTMultiHeadAttention(d_model, num_heads)
2       self.bert_mha = NMTMultiHeadAttention(d_model, num_heads)
3
4       self.ffn = self.point_wise_feed_forward_network(d_model,
            ↪ dff)
5
6       self.normalization_Layer = LayerNormalization(epsilon=1e6)
7       self.normalization_Layer2 = LayerNormalization(epsilon=1e6
            ↪ )
8
9       self.drop_layer1 = Dropout(drop_rate)
10      self.drop_layer2 = Dropout(drop_rate)
```

In den Zeilen 1 und 2 werden die zwei Attention-Schichten initialisiert. Die erste und zweite Normierungsschicht werden in Zeilen 6 und 7 erstellt. Die letzte Komponente im Encoder wird in der Zeile 4 durch die Methode *point_wise_feed_forward_network* geliefert. Zwei zusätzliche Dropout-Layers werden instanziiert.

- Der Decoder:

Listing 6.3: BNMT_Decoder

```
1       self.self_mha = NMTMultiHeadAttention(d_model, num_heads)
2       self.bert_dec_mha = NMTMultiHeadAttention(d_model,
            ↪ num_heads)
3       self.enc_dec_mha = NMTMultiHeadAttention(d_model,
            ↪ num_heads)
4
```

```
5       self.ffn = self.point_wise_feed_forward_network(d_model,
            ↪ dff)
6
7       self.norm_layer = LayerNormalization(epsilon=1e-6)
8       self.norm_layer2 = LayerNormalization(epsilon=1e-6)
9       self.norm_layer3 = LayerNormalization(epsilon=1e-6)
10
11      self.drop_layer1 = Dropout(drop_rate)
12      self.drop_layer2 = Dropout(drop_rate)
13      self.drop_layer3 = Dropout(drop_rate)
```

Im Decoder werden drei Attention-Layers nach Abb. 6.4, die in Zeilen 1 bis 3 erstellt werden, angewendet. Jedem Multi-Head-Attention-Layer folgt eine Normierungsschicht. Die Schichten sind in den Reihen 7 bis initialisiert. Für die Erstellung des Feed-Forward-Network wird die gleiche Methode wie im Encoder verwendet. Im Decoder werden drei Dropout-Schichten eingesetzt, eine nach jedem Schritt.

Mit der Ausführung der oben genannten Programmzeilen ist die Implementierung des NMT-Modells abgeschlossen. Im weiteren Verlauf der Arbeit werden die Erkenntnisse aus den durchgeführten Testreihen vorgestellt.

6.1.3 Erkenntnisse

In der Testphase wurde das beschriebene Modell ausprobiert. Weiterhin werden unterschiedliche Lernraten ausprobiert, nämlich zwei Klassen - 1e-4, 1e-3.

Corpus

Die Anzahl der Daten im Trainingscorpus besteht aus 163 Paaren. Das Corpus wird in 2 Gruppen geteilt - eine Lerngruppe und eine Evaluierungsgruppe. Das Corpus stammt aus mehreren Quellen, zum einen aus dem politischen Programm der SPD (70 Paare), aus einem Antrag für Arbeitslosengeld (74 Paare) und den Ausgaben eines GPT-3 Testmodells (19 Paare). Die Letzteren bestehen aus nur 19 Paaren, da Open GPT-3 nur eine Probeversion bietet, die die Kosten von 18€ abdeckt, was nur für die 19 Paare ausreichte. Die Ausschnitte mussten manuell ausgewählt und gekennzeichnet werden. Ein zusätzliches Corpus musste erstellt werden, um die Anwendbarkeit der Modelle zu prüfen. Das Corpus stammt aus einem Wohngeldformular. Ein kleines Corpus wurde verwendet aufgrund der systematischen Charakteristiken von Transfered Learning, deren Vorteil ist, dass die Größe des Corpus für das Fine-Tuning eines Modells klein gehalten werden kann. In mehreren Quellen in der Literatur, wie z.B. [uPR, Ten] um nur einige zu nennen, wird aufgeführt, dass die Anwendung eines vortrainierten Modells die Notwendigkeit für große Datensätze senkt. Mit dem vorliegenden Corpus soll somit getestet werden, ob diese kleine Anzahl für das Trainieren von einem NMT-Modells ausreichend ist.

Analyse

Das Training mit dem Corpus wurde sowohl auf dem zur Verfügung stehenden Endgerät als auch auf einem HPC in Kaiserslautern ausgeführt. Nur die Ergebnisse aus dem HPC wurden in dem folgenden Kapitel analysiert, da sie gleich in beiden Fällen waren. Die Modelle trainierten auf dem 'Elweritsch'-Cluster ([uTM]) im TU Kaiserslautern.

Tests

Leider lieferten beide Modelle keine positiven Ergebnisse. Alle Variationen des Modells konvergierten so, dass immer eine bestimmte Klasse, bzw. ein Wort, ausgewählt wurde. Die Abschnitte in der Abb. 6.2 stammen aus der Trainingsausgabe des Modells, bei dem die Lernrate auf 1e-4 gesetzt wurde. Dieser Test lief für 30 Iterationen auf dem HPC und dauerte 25857 s (etwa 7 Stunden und 11 Min).

```
Postleitzahl (die Nummer vor der Stadt. Sie besteht aus fünf Zahlen).

[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
[UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK]
_____
Bank.

[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
[UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK]
_____
Die IBAN-Nummer finden Sie auch auf Ihren Kontoinformationen.

[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
[UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK]
```

Abb. 6.2: NMT-Model Ausgabe bei einer LR von 1e-4

In der Abb. 6.2 sind 6 Beispiele ersichtlich, die aus dem Testkorpus ausgewählt wurden. Die Gruppierung erfolgt in zwei Paaren. Die Listen in Abb. 6.2 repräsentieren die Prognose des Modells und die Sätze unterhalb der Zahlenlisten sind Target-Sätze. Die Prognose besteht aus der wiederholten Token-Id 2. Die Id repräsentiert den Token *[UNK]*, welcher dafür verwendet wird, unbekannte Wörter, bzw. solche die nicht in der Look-up-Tabelle vorkommen, zu ersetzen.

Im zweiten Test konnte das gleiche Ergebnis wie bei der Abb. 6.2 festgestellt werden. Die Prognosen unterscheiden sich nicht von dem Modell, das mit der Lernrate von 1e-4 gelehrt wurde. In Abb. 6.3 können die Prognosen eingesehen werden.

Die Prognose des Modells besteht ausschließlich aus den Token „[UNK]„.

Anhand dieser Betrachtungen kann entschlossen werden, dass das Modell fehlerhaft konvergiert.

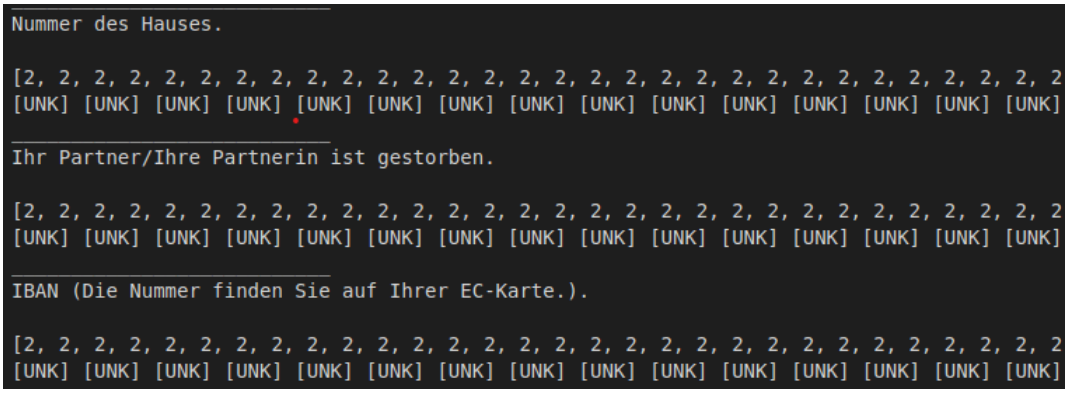


Abb. 6.3: NMT-Model Ausgabe bei einer LR von 1e-3

6.1.4 Schlussfolgerung

Der Grund für die Divergenz konnte nicht festgestellt werden. Die Eingaben wurden überprüft und zeigten keine Mängel.

Aus den Testergebnissen und der Eingabeuntersuchung kann hergeleitet werden, dass das Modell mit der vorliegenden Struktur ungeeignet für die anzustrebende Aufgabe ist. Im folgenden Kapitel wird ein gelungeneres Modell vorgestellt (besser im Sinne von Prognosequote und Konvergenz).

6.2 Seq2Seq Model

Das nächste Modell, das untersucht wurde, ist ein Sequenz-to-Sequenz-Model (seq2seq Model). Wie der Name andeutet, liefert das Modell eine entsprechende Antwortsequenz für jede beliebige Sequenz. Solche Modelle werden für Textgeneratoren verwendet, sind aber auch für Übersetzungsaufgaben geeignet. Die Idee ist, das Modell für die Aufgabe „Übersetzung in einfache Sprache„ anzupassen. Wie genau das erfolgt, wird in den nächsten Kapiteln vorgestellt.

6.2.1 Struktur

Laut [Ten22c] ist das Modell ein Neural Machine-Translation-Model und bezeichnet ein System, das die Wahrscheinlichkeit $p(y|x)$ für eine beliebige Sequenz x_1, ..., x_n berechnet, zu einer anderen y_1, ..., y_m zugeordnet zu werden. Das Modell besteht aus einem Encoder und einem Decoder. Der Encoder berechnet die Vektor-Repräsentation für jede Sequenz s und der Decoder erzeugt die Wörter schrittweise. Die Funktion, die vom Decoder berechnet wird, ist die bedingte Wahrscheinlichkeit:

$$logp(x|y) = \sum_{j=1}^{m} logp(y_j|y_{<j}, s)[\text{MTLM15}] \tag{6.3}$$

Eine solche Zerlegung kann im Decoder mit Hilfe von Recurrent-Neural-Networks (RNN) modeliert werden. In der Literatur verwendet man unterschiedliche RNNs,

was wiederum zu unterschiedlichen Ergebnissen führt. In diesem Kapitel wird das übliche Modell mit jeweils einem RNN für den Encoder und Decoder thematisiert. Ein zweites Modell wird gebaut, damit ein vortrainiertes BERT-Modell für den Encoder verwendet werden kann. So ein Prinzip nennt sich Transfered Learning [Ten]. Transfered Learning beschreibt den Prozess, in wessen Rahmen die erworbene Information aus unbeschrifteten Daten verwendet wird, um beschrifteten kleineren Datensatz zu bearbeiten. Das soll die Inferenz aus dem kleinen Datensatz verbessern, sodass der Lernprozess zu besseren Ergebnissen führt.

Die Struktur des Modells wird in der Abb. 6.4 ersichtlich:

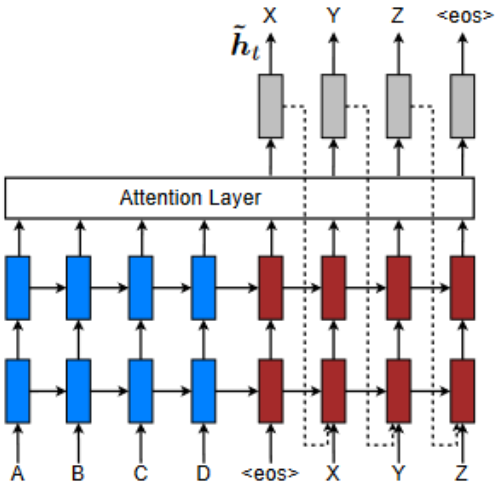


Abb. 6.4: NMT-Model Struktur [MTLM15]

Die blauen Rechtecke repräsentieren den Encoder und die Roten dementsprechend den Decoder. Der übliche Encoder bearbeitet die Wörter aus der Sequenz schrittweise, jedoch ergibt sich in unserem Fall, bedingt durch die verwendete Quelle [Ten22c], dass die Sequenz komplett betrachtet wird, woraus der Encodervektor entsteht.

Der Encoder für dieses Modell ist der BERT-base-german-cased-Model, der auf Huggingface [dee19] zu finden ist. Das nächste Unterkapitel setzt sich mit dem Decoder auseinander.

6.2.2 Decoder

Das Hauptziel des Decoders ist, die Prognose für den nächsten Token zu bestimmen. Die Operationen im Decoder können in 5 Schritte aufgeteilt werden [Ten22c]:

1. es wird der Targetvektor geliefert
2. ein RNN markiert die aktuell erzeugten Tokens
3. die Ausgabe vom RNN ist die Eingabe in der Attention-Schicht und es wird der Kontextvektor geliefert
4. der Attentionsvektor wird generiert

5. die neuen Prognosen für den Token werden dem Attentionvektor entnommen.

Der Decoder bringt Attention ein und beachtet somit besondere Abschnitte aus der Eingabesequenz. In der Attention-Schicht fließt eine Sequenz aus Vektoren und für jeden Sequenzvektor wird ein Attentionsvektor geliefert. Anschließend wird der Attentionsvektor in den Kontextvektor umgewandelt. Schließlich wird der Kontextvektor transformiert, sodass ein Logits-Vektor erhalten wird, dessen Größe gleich die Anzahl der Wörter im Corpus ist.

Der Attentionvektor wird mittels der folgenden Formel berechnet:

$$a_t(s) = \frac{\exp(score(h_t, \overline{h}_s))}{\sum_{s'=1}^{S} \exp(score(h_t, \overline{h}_{s'}))} [\text{MTLM15}] \tag{6.4}$$

Die Formel ist wie folgt zu lesen: t steht für den Decoder- und s für den Encoderindex. Die Variable a_{ts} ist die Attentionweights. h_t bezeichnet den Decoderzustand und h_s den aktuelle Zustand vom Encoder. Weiterhin ist die *score*-Funktion gegeben [MTLM15]:

$$score(h_t, \overline{h}_s) = \begin{cases} h_t^\top \overline{h}_s & dot \\ h_t^\top W_a \overline{h}_s & general \\ v_a^\top \tanh(W_a[h_t^\top; \overline{h}_s]) & concat \end{cases} \tag{6.5}$$

Hier sind W_a und v_a Variablen, die gelernt werden müssen. Die zweite und die dritte Varianten (*general* und *concat*) wurden zum Gegenstand der praktisch-experimentellen Testphase für die Berechnung des Kontextvektors. Im Kontextvektor befindet sich die Information über die Wichtigkeit der einzelnen Ausgangswörter für das Targetwort. Anhand des Kontextvektors wird die Prognose geliefert.

Als nächstes wird im Abschnitt 6.2.3 die Implementierung für das Modell erläutert.

6.2.3 Implementierung

Das Modell besteht aus zwei Bestandteilen, der Encoder und der Decoder. Für die Analyse wird mit drei unterschiedlichen Modellen gearbeitet. Zwei davon setzen ein vortrainiertes BERT-Modell als Encoder ein. Das dritte Modell besteht aus einer RNN-Schicht, ähnlich wie im Decoder.

6.2.4 Implementierung vom Encoder im NMT-Model ohne BERT

Der Encoder im NMT besteht aus zwei Schichten. Die Implementierung ist gegeben:

```
1     self.embedding = Embedding(self.vocab_size, enc_units)
2
3     self.gru = GRU(self.enc_units, return_sequences=True,
          ↪ return_state=True, recurrent_initializer='
          ↪ glorot_uniform')
```

Der Encoder hier besitzt keine positionelle Einbettung im Vergleich zum Encoder des Transformers (s. Listing 4.3) oder BERT (s. 5.2.2). Dieser Unterschied kann im Gegensatz zum Modell, das BERT für den Encoder einsetzt, in der Testphase eine Rolle spielen.

Implementierung vom Decoder

Der Decoder wird nach den 5 Schritten aus 6.2.2 implementiert und folgender Programmabschnitt stellt diese dar:

Listing 6.4: NMT-Decoder

```
1     # For Step 1. The embedding layer converts token IDs to
          ↪ vectors
2     self.embedding = Embedding(self.vocab_size, d_model)
3
4     # For Step 2. The RNN keeps track of what's been generated
          ↪ so far.
5     self.gru = GRU(d_model, return_state=True, return_sequences=
          ↪ True, recurrent_initializer='glorot_uniform')
6
7     # For Step 3. The RNN output will be the query for the
          ↪ attention layer
8     self.attention = BahdanauAttention(self.d_model)
9     # oder
10    # self. attention = LuongAttention(self.d_model)
11
12    # For Step 4. converting `ct` to `at`
13    self.Wc = Dense(self.d_model, activation=tf.math.tanh,
          ↪ use_bias=False)
14
15    # For Step 5. This fully connected layer produces the logits
          ↪  for each output token
16    self.fc = Dense(self.vocab_size)
```

Für den zweiten Schritt können auch andere RNNs benutzt werden, unter Anderem auch eine **LSTM**-Schicht (**L**ong **S**hort **T**erm **M**emory). Im Schritt drei wird in diesem Fall die Bahdanau-Attention verwendet, die zweite Option wäre die Luong's Multiplikative Attention, der Unterschied in der Funktionen ist in der Gleichung (6.5) zu sehen.

6.3 Erkenntnisse

In diesem Kapitel werden drei Modelle getestet - zwei BERT-basierte NMT-Modelle mit unterschiedlichen Score-Funktionen und ein NMT-Modell (Transformer-ähnliches Modell) ohne die Verwendung von BERT (mit der Bahndanau's additive

Attention). Die Modelle werden für 100 Iterationen trainiert und es werden vier unterschiedliche Lernraten geprüft - 0.001, 0.0001, 0.00002, 0.00001. Der verwendete Optimizer ist Adam (Adaptive Moment Estimation).

Für Trainingsdaten wird dasselbe Corpus (s. Abschnitt 6.1.3) genommen.

Ziel der Tests ist es das beste Modell zu bestimmen. Der Test hilft beim Erkunden der besten Lernrate und Bestimmung des Modells, das die Aufgabe „Übersetzung in einfache Sprache„ am besten löst. Als Kriterium dafür werden das BLEU-Score und ausgewählte Ein- und Ausgabe der Modelle angewendet.

Die drei Modelle besitzen auch unterschiedliche Anzahl an Variablen:

- Das Simple Language Model besitzt 78 598 704 Variablen
- Das BERT Simple Language Model mit Bahdanaus Attention besitzt 161 096 496
- Das BERT Simple Language Model mit Luongs Attention besitzt 159 914 544.

Die zwei BERT-Modelle sind doppelt so komplex wie das Simple-Language-Modell. Ob das ein Vorteil, oder Nachteil ist, zeigt sich in den Tests.

6.3.1 Training auf einem High Performance Computer

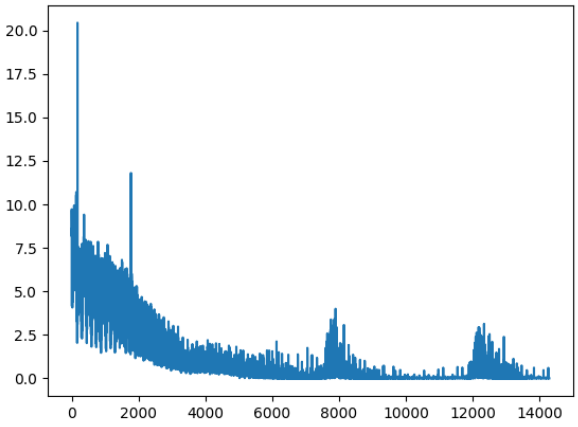
Die Experimente werden auf einem Hochleistungsrechner in Kaiserslautern durchgeführt. Die Modelle trainieren auf dem 'Elweritsch'-Cluster ([uTM]) im TU Kaiserslautern. Die Untersuchungen werden mit den Lernraten (0.001, 0.0001, 0.00002, 0.00001) durchgeführt. Für die letzte Lernrate werden nur Tests für die BERT-Modelle ausgeführt. Zum Trainieren wird das in Abschnitt 6.1.3 eingeführte Corpus verwendet. Für die Evaluierung wurden sowohl das Trainingscorpus als auch ein zusätzliches Corpus, das aus 10 Evaluierungspaaren besteht, benutzt. Alle Lernprozesse haben 100 Iterationen durchlaufen, außer den Tests mit der Lernrate 0.00001, die für 3 Phasen von jeweils 50 Iterationen durchlaufen haben.

Die Abbildungen, die als Nächstes dargestellt werden, zeigen in ihrer X-Achse die Anzahl der Iterationen, und in der Y-Achse werden die Werte für die Fehlerrate angegeben. Der Batch hat für alle Tests den Wert von 1. Das bedeutet, dass pro Iteration jeweils ein Trainingspaar bearbeitet wird.

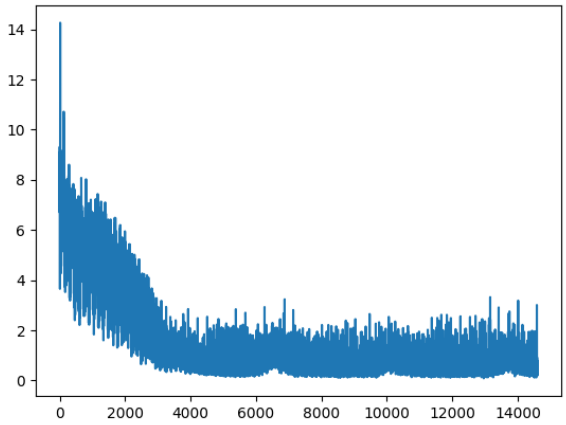
Testen mit Lernrate 0.001

Als erstes wird die Lernrate von **0.001** angewendet. Die Abb. 6.5a bis 6.5c zeigen den Lernprozess. Aus den Abbildungen kann ein besseres Verständnis dafür verschafft werden, wie geeignet eine niedrigere Fehlerrate für die gegebenen Größen der Modelle ist.

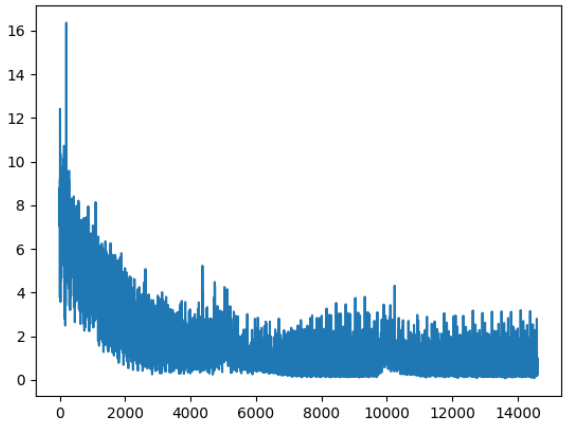
Laut Abb. 6.5a wird den Optimalzustand der Variablen schon bei dem Iterationsschritt 10000 erreicht. Innerhalb der ersten 3000 Iteraionen erfolgen die größten Verbesserungen des Modells und schätzungsweise ab dem Schnittpunkt von 10 000 können nur minimale Verbesserungen festgestellt werden. Danach steigen die Fehlerwerte der Modelle für einige Schritte sogar, aber in den letzen Epochen verbessert sich der Durchschnitt und es wird eine Fehlerquote von nahezu 0 erreicht.



(a) SLM-Model mit Lernrate 0.001



(b) BSLM-Model(Bahndanau)



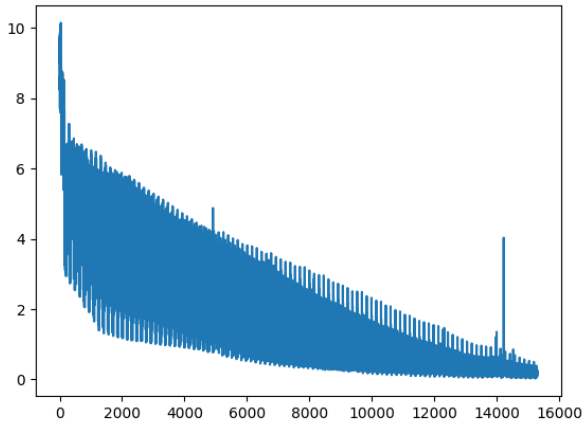
(c) BSLM-Model (Luong)

Abb. 6.5: Fehlerwerte bei den Tests mit der Lernrate 0.001

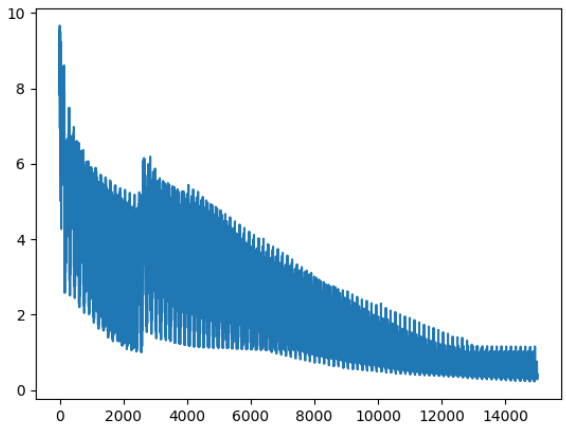
Die letzten Abb. 6.5b und 6.5c zeigen deutlich höhere Fehlerwerte als in der Abb. 6.5a. In Bezug auf die Fehlerwerte weist das Simple-Language-Modell die größte Verbesserung. Ob niedrige Fehlerwerte gleichbedeutend mit einem guten Modell sind, zeigt sich später bei der Untersuchung mit unbekannten Corpussätzen.

Testen mit Lernrate 0.0001

Die Abb. 6.6a, 6.6b und 6.7 zeigen eine stetige Verbesserung. Wenn die Fehlerwerte als Kriterium verwendet werden, könnte man sagen, dass das beste Modell wieder das SLM-Modell sei:



(a) SLM-Model



(b) BSLM (Bahdanau)

Abb. 6.6: Fehlerwerte bei den Tests mit der Lernrate 0.0001

Die Kurve der Fehlerfunktion für die Modelle BSLM (Bahdanau und Luong) sieht sehr ähnlich aus, jedoch könnte keinem der Modelle ein Vorteil zuerkannt

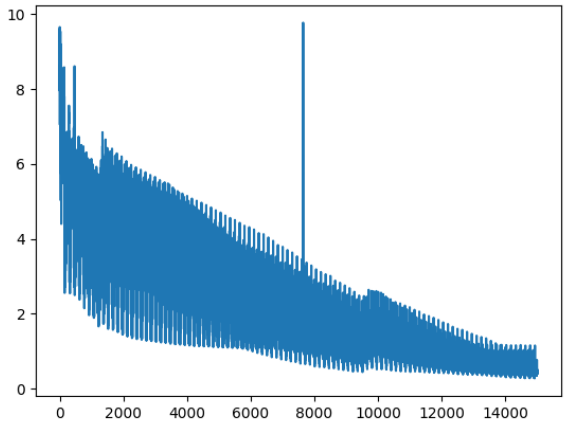


Abb. 6.7: Fehlerwerte bei den Tests mit der Lernrate 0.0001 (Fortsetzung - BSLM (Luong))

werden. Ab dem 13 000. Schritt ist ein Minimum erreicht, das durch die restlichen Iterationen kaum verbessert wird. Aus den Ausgaben der Tests wird jedoch in den späteren Iterationen trotzdem eine Verbesserung erzielt.

Testen mit 0.00002

Der letzte Test wird mit der Lernrate von **0.00002** durchgeführt. In den Abb. 6.8, 6.9a und 6.9b kann der Verlauf des Trainingsprozesses verfolgt werden.

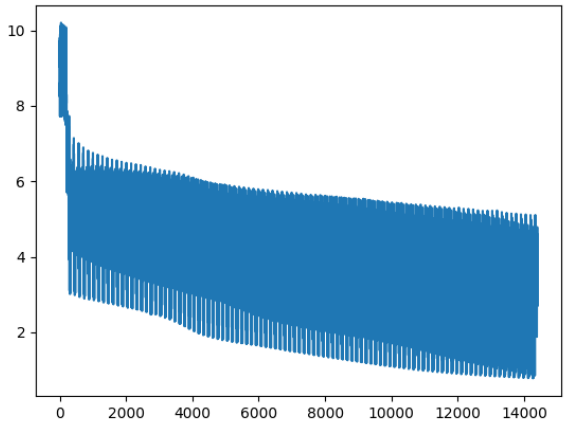
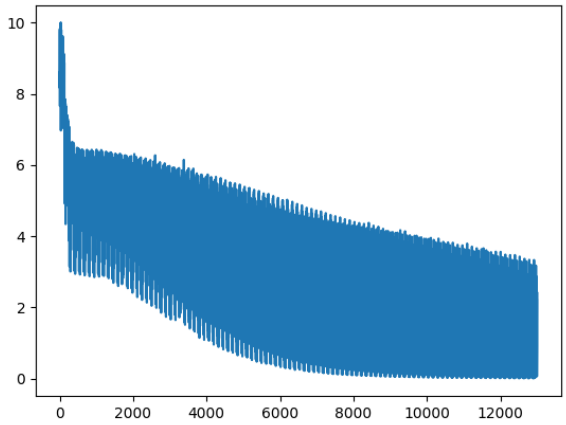
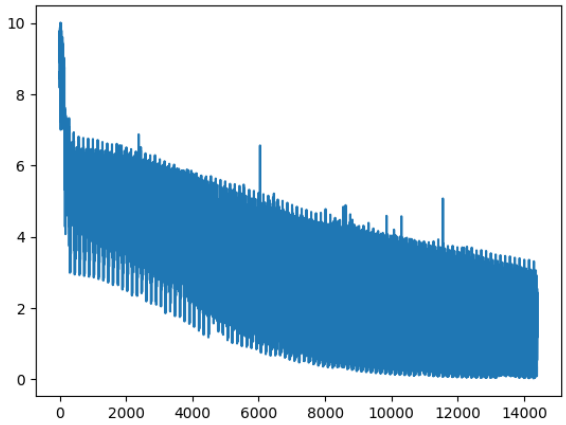


Abb. 6.8: Fehlerwerte bei den Tests mit der Lernrate 0.00002 (SLM-Model)



(a) BSLM (Bahdanau)



(b) BSLM (Luong)

Abb. 6.9: Fehlerwerte bei den Tests mit der Lernrate 0.00002 (Fortsetzung)

Die ersten Hundert Iterationen weisen die größte Optimierung der Fehlerfunktion auf. Außerdem kann ein klarer Vorteil der BERT-Modelle erkannt werden. Mit der Senkung der Lernrate steigen die Fehlerwerte, aber die BERT-Modele stellen ihre Fähigkeiten unter Beweis, trotz der größeren Anzahl an Variablen.

Testen mit 0.00001

Im Laufe der Untersuchung konnte aus den Ausgaben der BERT-Modelle festgestellt werden, dass die Senkung der Lernraten bei den dargestellten BERT-Modellen bessere Ergebnisse zu Folge hatte. Die ausgewählten Beispiele aus dem Listing 6.5 bestätigen diese Behauptung. Der Kandidat „Anrede„ wurde als Beispiel genommen, da er in allen Testdurchläufen eingebracht wurde.

Listing 6.5: Prognose für ausgewählte Eingaben der BERT-Modelle

```
1    Candidate: Anrede
2    Target: Herr oder Frau.
3    # BSLM_HPC_General_lr1e-3
4    Prediction: ['Ich', 'mache', 'mir', 'einen', 'Kaffee']
5
6    # BSLM_HPC_General_lr1e-4
7    Prediction: ['Sie', 'sind', 'schwanger', '.']
8
9    # BSLM_HPC_General_lr2e-5
10   Prediction: ['Herr', 'oder', 'Frau', '.']
11
12   # BSLM_HPC_Bahdanau_lr1e-3
13   Prediction: ['Ich', 'mag', 'hier', 'zu', 'sein', '.', 'Die',
        ↪ 'Natur', 'ist', 'sehr', 'schn', 'und', 'die', 'Luft',
        ↪ 'ist', 'sehr', 'frisch', '.', 'Ich', 'fhle', 'keine',
        ↪ 'Probleme', 'hier', '.', 'Ich', 'fhle', 'mich', 'auch
        ↪ ', 'frei', '.']
14
15   # BSLM_HPC_Bahdanau_lr1e-4
16   Prediction: ['Wir', 'wollen', 'morgen', 'in', 'die', 'Stadt
        ↪ ', 'gehen', '.']
17
18   #BSLM_HPC_Bahdanau_lr2e-5
19   Prediction: ['Herr', 'oder', 'Frau', '.']
```

Als weitere Beobachtung konnte aus manchen Ausgaben Overfitting (s. Listing 6.6) nachgewiesen werden. Dies hatte zu Folge, dass zwei weitere Tests bestehend aus 3 Phasen als notwendig erachtet wurden. Die Entscheidung, das Lernen in drei Phasen aufzuteilen, kam nach der Evaluierung der anderen Lernraten. Die Modelle wiesen zwar ein gutes Minimum der Fehlerfunktion (s. Abb. 6.5a bis 6.5c) auf, jedoch zeigten sie bei unbekannten Sätzen sinnfreie Übersetzungen. Durch die Phasentrennung wird die Lösung dieses Problems angestrebt.

Listing 6.6: Beispiele für Overfitting

```
1    Candidate: Antragstellende Person
2    Target: Wer stellt den Antrag?
3
4    # SLM_HPC_v1_without_BERT_lr1e-3
5    Prediction: ['Nummer', 'von', 'der', 'Renten', '##
        ↪ versicherung', '.']
6
7    # SLM_HPC_v1_without_BERT_lr1e-4
8    Prediction: ['Sie', 'haben', 'eine', 'Frau', 'geheiratet',
        ↪ '.']
9
```

```
10    # BLM_HPC_v1_lr1e-3_Bahdanau_100EP
11    Prediction: ['Ich', 'mag', 'hier', 'zu', 'sein', '.', 'Die',
          ↪ 'Natur', 'ist', 'sehr', 'schn', 'und', 'die', 'Luft',
          ↪ 'ist', 'sehr', 'frisch', '.', 'Ich', 'fhle', 'keine',
          ↪ 'Probleme', 'hier', '.', 'Ich', 'fhle', 'mich', 'auch
          ↪ ', 'frei', '.']
12
13    Candidate: Stellung in Beruf?
14    Target: Was machen Sie?
15    # SLM_HPC_v1_without_BERT_lr1e-4
16    Prediction: ['Sie', 'haben', 'eine', 'Frau', 'geheiratet',
          ↪ '.']
17
18    # BLM_HPC_v1_lr1e-3_Bahdanau_100EP
19    Prediction: ['Ich', 'mag', 'hier', 'zu', 'sein', '.', 'Die',
          ↪ 'Natur', 'ist', 'sehr', 'schn', 'und', 'die', 'Luft',
          ↪ 'ist', 'sehr', 'frisch', '.', 'Ich', 'fhle', 'keine',
          ↪ 'Probleme', 'hier', '.', 'Ich', 'fhle', 'mich', 'auch
          ↪ ', 'frei', '.']
```

In den beiden Tests laufen die Bahdanau- und Luong-Modele für 3 Phasen á 50 Iterationen und in Abb. 6.10 und 6.11 ist die Fehlerfunktion für die letzten 50 Iterationen aufgezeichnet.

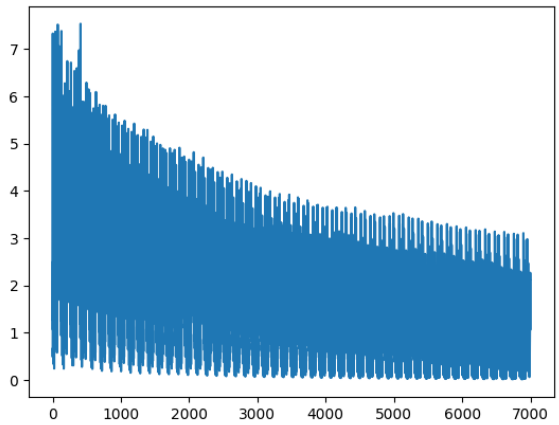


Abb. 6.10: Fehlerwerte bei den Tests mit der Lernrate 0.00001 (Bahdanau-Model)

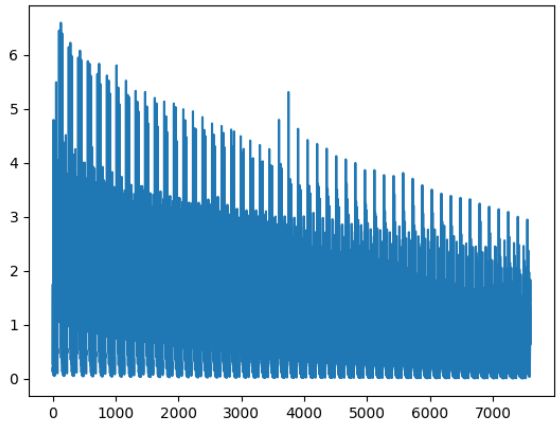


Abb. 6.11: Fehlerwerte bei den Tests mit der Lernrate 0.00001 (Luong-Model)

Aus der Abb. 6.10 und 6.11 geht hervor, dass das Bahdanau-Model kleinere Fehlerwerte und geringere Schwankung zeigt. Die Lernkurven sprechen für die Lernfähigkeit des Bahdanau-Modells. Die Schwankung der Kurve erklärt sich durch die Zufälligkeit bei der Auswahl der Trainingspaare. Das hat sich als eine gute Strategie erwiesen, was entsprechende Ergebnisse bei der Untersuchung mit BLEU erbracht hat.

6.3.2 BLEU-Score

BLEU ist kurz für **Bi**-**L**ingual **E**valuation **U**nderstudy und ist ein Berechnungsmodell zur schnellen und automatischen Evaluierung der Übersetzungen eines Modells. Für diesen Zweck werden Methoden aus der Bibliothek *nltk.translate* angewendet.

Die Berechnung des BLEU-Score erfolgt, indem der gewichtete geometrische Durchschnitt (Gleichung (6.6)) aus den N-Gramm-Werten in allen Rangordnungen von 1 bis n errechnet wird [Bro].

$$\left(\prod_{i=1}^{n} x_i^{w_i}\right)^{\frac{1}{\sum_{i=1}^{n} w_i}} \text{ [Wik]} \tag{6.6}$$

In der Gleichung (6.6) bezeichnet $x_i \in \mathbb{R}$ ein N-Gramm-Wert der Rangordnung i. Der N-Gramm x_i ergibt sich aus der Abdeckung von einer fragmentierten Target-Einheit und einem fragmentierten Kandidaten. Die Zahl n stellt die Anzahl der betrachteten Fragmente dar. Üblicherweise wird für n der Wert 4 eingesetzt. Der Wertebereich von x_i liegt zwischen 0 und 1, sodass x_i bei voller Übereinstimmung den Wert 1 und bei fehlender den Wert 0 erhält. Die Kandidaten dürfen auch partiell die Target-Einheit abdecken. In diesem Fall erhält x_i einen rationalen Wert. Die Variable w_i repräsentiert das Gewicht der Ordnung i. Üblicherweise

wird der Wert w gleich $\frac{1}{n}$ in allen Rangordnungen ausgewählt, um sicherzustellen, dass alle n-Gram-Werte gleichermaßen zum gesamt Score beitragen.

Als nächstes wird der BLEU-Score der Modelle untersucht. In der Abb. 6.12 sind die BLEU-Werte in absteigender Reihe gegeben:

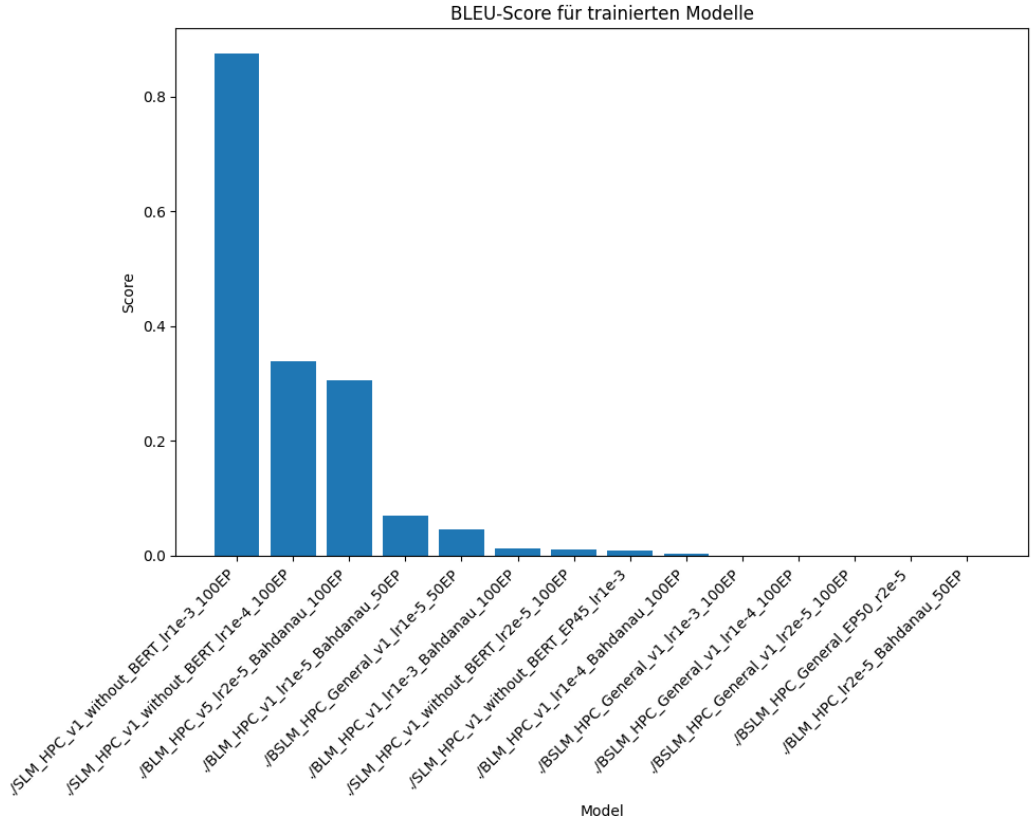


Abb. 6.12: BLEU-Scores

Interessanterweise weisen die SLM-Modele ohne BERT den besten Wert auf. Die am wenigsten zufriedenstellend bei der Übersetzung sind die BSLM-Modelle, bei welchen die *general*-Funktion (siehe Gleichung (6.5)) verwendet wird. Hierzu muss man anmerken, dass der BLEU-Score auf den ganzen Traningsdaten basiert.

Eine weitere Erkenntnis aus den Lernprozessen und Abb. 6.12 ist, dass die BERT-Modelle sich besser mit kleineren Lernraten entwickeln, während das simple Modell besser auf die höheren Lernraten reagiert. Die Beispiele im Listings 6.5 und 6.7 bestätigen diese These. Die Prognosen werden aus dem BLEU-Test mit dem Trainingssatzes ausgeschrieben. Die Ausgabe werden für die Eingabe „Anrede„ ausgesucht.

Listing 6.7: Prognosen von Simple-Language-Model

```
1    Candidate: Anrede
2    Target: Herr oder Frau?
3    # SLM_HPC_ohne_BERT_lr2e-5
```

```
4     Prediction: ['In', '##name', '.']
5
6     # SLM_HPC_ohne_BERT_lr1e-4
7     Prediction: ['Herr', 'oder', 'Frau', '.']
8
9     # SLM_HPC_ohne_BERT_lr1e-3
10    Prediction: ['Herr', 'oder', 'Frau', '.']
```

Zur Bestimmung der eigentlichen Anwendbarkeit der Modelle wird ein unbekannter Datensatz erstellt und in einem weiteren BLEU-Test verwendet. Im nächsten Kapitel wird ein Graph mit den Ergebnissen aufgeführt.

Untersuchung mit Evaluierungsdaten

Zu Bekräftigung der gewonnenen Erkenntnisse wird eine zusätzliche Untersuchung mit Sätzen, die nicht in dem Trainingscorpus vorkommen, durchgeführt. Das Ziel ist es, die Anwendbarkeit der Modelle zu prüfen und festzustellen, ob sie mit einem kleinen Corpus (Abschnitt 6.1.3) gelehrt werden können, sowie ob sie schlussendlich für einen Produktionseinsatz geeignet sind. Die Abb. 6.13 zeigt die Scores der Modelle:

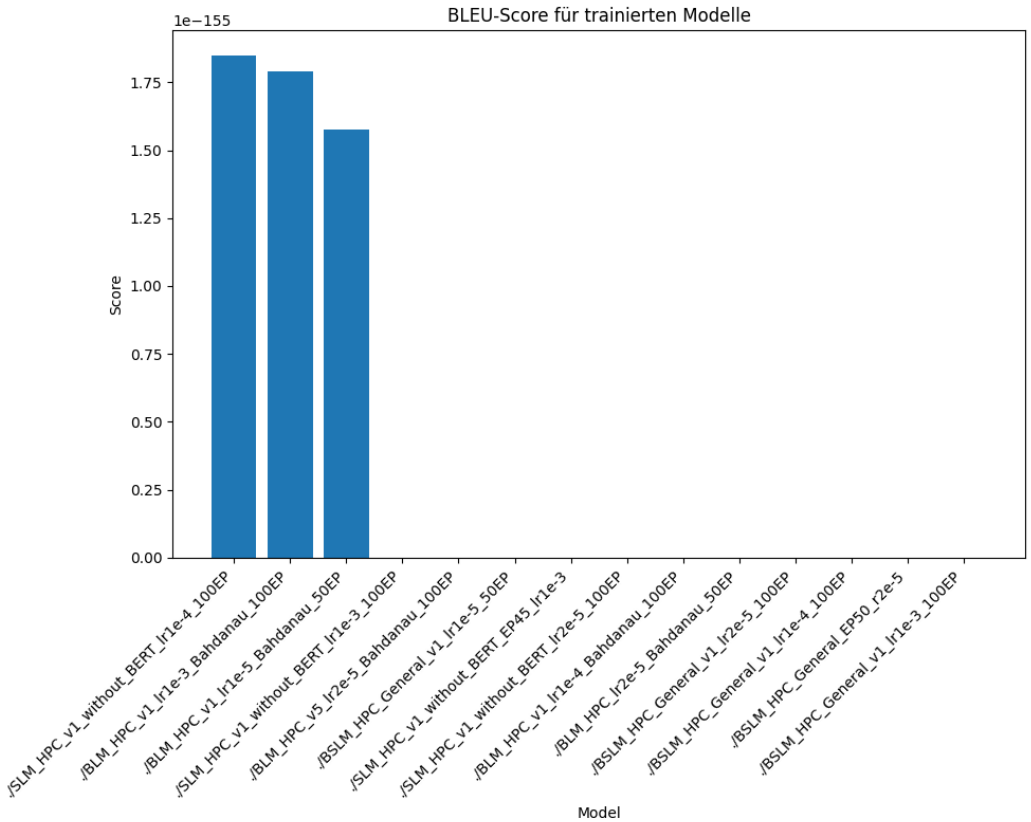


Abb. 6.13: BLEU-Scores für unbekannte Sätze

Bei diesem Test enttäuschen alle Modelle. Bemerkenswert ist, dass die besten Modelle laut Abb. 6.13 das SLM mit Lernrate 0.0001 und zwei BERT-Modelle mit Bahdanau Attention mit Lernrate 0.001 und 0.00001. Auffallend ist, dass sich in den unterschiedlichen Testläufen jeweils andere Modelle als die Besten erwiesen haben.

6.3.3 Schlussfolgerung

Aus den Abb. 6.12 und 6.13 zeigte sich, dass sich das Simple-Language-Model zwar als bestes Modell qualifizieren konnte, aber es scheiterte bei dem unbekannten Corpus, gleichermaßen wie alle anderen Modelle. Somit konnte keins der Modelle den Test mit dem unbekannten Corpus bestehen. Die BLEU-Tests rechtfertigten die niedrigen Erwartungen, weil die Lernprozesse zeigten, dass die Modelle bei dem Evaluierungsabschnitt fehlerhafte Prognosen geliefert haben. Angesichts des verwendeten kleinen Corpus ist es nicht möglich, ein funktionstüchtiges Modell zu trainieren. Die Hypothese war, dass der BERT-Encoder die kleine Größe des Corpus kompensieren würde, leider geht aus den Tests das Gegenteil hervor.

Ein größeres Corpus sollte für die vorliegende Aufgabe (Übersetzung in einfache Sprache) verwendet werden, um die Ergebnisse zu verbessern, oder idealerweise zum Erfolg zu führen. Es ist nicht möglich ein Modell zum Zweck der Übersetzung in einfache Sprache mit einem nicht ausreichenden, lediglich aus 168 Datensätzen bestehenden Corpus zu trainieren.

7

Zusammenfassung und Ausblick

In dieser Ausarbeitung wurde versucht, die Aufgabe „Übersetzung in einfache Sprache mit Hilfe von Transformern„ zu lösen. Es wurden zwei Modellstrukturen untersucht, für die jeweils 2 bzw. 3 Modelle entwickelt worden sind. Die Untersuchung basierte auf dem Transfered Learning Prinzip, was erlaubte, einen kleineren Datensatz zu nutzen. Leider wurde das Corpus zu klein erstellt, sodass die Modelle die Daten übertrainiert haben. Die Erkenntnis daraus ist, dass ein Corpus aus 168 Trainingspaaren nicht ausreichend ist, um ein Modell zu erstellen, das für die Übersetzung in einfache Sprache anwendbar ist.

Falls aber bei den erneuten Versuchen ein Datensatz im Umfang von mindestens 500 Paaren zugrunde läge, könnte man, ausgehend von den gewonnenen Erfahrungen und Ergebnissen durchaus mit einem erfolgreicheren Abschluss der Testreihen rechnen.

Literaturverzeichnis

Ali. Ali, Zafar: *A simple Word2vec tutorial.*

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

Bro. Brownlee, Jason: *A Gentle Introduction to Calculating the BLEU Score for Text in Python.*

dee19. deepset: *German BERT(aka "bert-base-german-cased")*, 2019.

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.*

MTLM15. Minh-Thang Luong, Hieu Pham und Christopher D. Manning: *Effective Approaches to Attention-based Neural Machine Translation.* Stanford Publication, 2015.

Rot21. Rothman, Denis: *Transformer for Natural Language Processing.* Packt Publishing Ltd, 2021.

Ten. Tensorflow: *Transfer learning and fine-tuning.*

Ten22a. Tensorflow: *Classify text with BERT*, 2022.

Ten22b. Tensorflow: *Fine-tuning a BERT model*, 2022.

Ten22c. Tensorflow.com: *Neural Machine Translation with Attention*, 2022.

Uni. University, Stanfor: *GloVe: Global Vectors for Word Representation.*

uPR. Pratik Ratadiya, Adiya Malte und: *Evolution of Transfer Learning in Natural Language Processing.*

uTM. TU Mainz, TU-Kaiserslauter und: *Elweritsch: Hochleistungsrechner.*

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.

Wik. Wikipedia: *Weighted geometric mean.*

Zhu20. Zhu, Jinhua: *Incorporating BERT into Neural Machine Translation.* Published for ICLR 2020, 2020.

A

Erklärung der Kandidatin / des Kandidaten

☐ ☐

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

Datum	Unterschrift der Kandidatin / des Kandidaten