Clemens Schulz
Matrikelnummer: 5334483
Bachelor Informatik
9. Semester
clemens.schulz@stud.uni-frankfurt.de

Bachelorarbeit

Automatische Disambiguation für das Englische

Clemens Schulz

Abgabedatum: 19.04.2018

Text Technology Lab
Goethe-Universität Frankfurt
Betreut durch
Prof. Dr. Alexander Mehler
und Tolga Uslu

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst habe. Ebenso bestätige ich, dass diese Arbeit nicht, auch nicht auszugsweise, für eine andere Prüfung oder Studienleistung verwendet wurde.

_____

Ort, Datum

_____

Unterschrift

Zusammenfassung

Begriffe sind häufig nicht eindeutig. Eine „Bank“ kann ein Finanzinstitut oder eine Sitzgelegenheit sein und die Stadt Frankfurt existiert mehr als einmal. Dennoch können sie in vielen Fällen problemlos von Menschen unterschieden werden. Computer sind noch nicht in der Lage, diese Leistung mit vergleichbarer Genauigkeit zu erfüllen.

Der in dieser Arbeit vorgestellte Ansatz baut auf dem für das Deutsche bereits gute Ergebnisse erzielenden fastSense auf und verwendet ein neuronales Netz, um Namen und Begriffe in englischen Texten mit Hilfe der Wikipedia zu disambiguieren. Dabei konnte eine Genauigkeit von bis zu 89,5% auf Testdaten erreicht werden.

Mit dem entwickelten Python-Modul kann das trainierte Modell in bestehende Anwendungen eingebunden werden. Die im Modul enthaltenen Programme ermöglichen es, neue Modelle zu trainieren und zu testen.

Inhaltsverzeichnis

1. Einleitung

1.1. Aufgabenstellung

Ziel dieser Arbeit ist es, ein Modell zu entwickeln, das Begriffe in englischen Texten disambiguieren kann. Trainings- und Testkorpora mit Beispielen für unterschiedliche Bedeutungen werden aus Artikeln der englischen Wikipedia generiert.

1.2. Motivation

Oberflächlich identische Begriffe sind nicht immer eindeutig, können aber trotzdem in vielen Fällen von Menschen problemlos unterschieden werden. Kaum einer denkt bei dem Satz „Sie saßen auf einer Bank im Park." an ein Finanzinstitut in einer Parkanlage, auf dessen Dach Personen sitzen. Für Computer ist diese Schlussfolgerung jedoch nicht offensichtlich.

Das Themengebiet der Disambiguation beschäftigt sich mit der Lösung dieses Problems. Dabei kann es in weitere Bereiche unterteilt werden, insbesondere in Named-Entity Disambiguation (NED) und Word-Sense Disambiguation (WSD), die beide ein sehr ähnliches Problem lösen.

NED befasst sich mit Namen von z.B. Personen, Orten, Filmen oder ähnlichen Dingen, die selten eindeutig sind. Es gibt z.B. weltweit über 30 Städte mit dem Namen „Berlin". WSD befasst sich mit der allgemeinen Bedeutung von beliebigen Worten, also z.B. ob mit „gehen" die Fortbewegung zu Fuß oder Verlassen eines Ortes gemeint ist.

Anwendungsfälle beider Bereiche sind unter anderem maschinelle Übersetzung, Semantik Analyse und Information Retrieval, sie können aber auch in anderen Gebieten nützlich sein. Es besteht eine enge Verbindung zu Part-of-Speech Tagging und Lemmatisierung. Systeme können abhängig von ihrem Design beim Finden dieser Informationen helfen oder sie für die Disambiguation nutzen.

Um Namen oder Begriffe disambiguieren zu können, müssen sie zunächst gefunden werden. Named-Entity Recognition, also erkennen von Namen, ist ein eigens Themengebiet mit vielen Problemen und steht nicht im Mittelpunkt dieser Arbeit. Es wird deshalb nur kurz behandelt.

Die in dieser Arbeit beschriebene Vorgehensweise für NED und WSD basiert auf einer Beschreibung von fastSense (Uslu, Mehler und Baumartz 2018). Es wird ein neuronales Netz verwendet, das mit TensorFlow umgesetzt wurde. Das implementierte *ned*-Modul bietet eine Python-Schnittstelle um beliebige Texte zu disambiguieren und Skripte zum Trainieren und Testen eigener Modelle.

2. Verschiedene Disambiguationsansätze

Machine Learning kann auf unterschiedliche Arten eingesetzt werden, um Disambiguationsprobleme zu lösen, aber auch Graph-Algorithmen können, wie später beschrieben, gute Ergebnisse erzielen. Die folgenden Abschnitte stellen verschiedene Ansätze kurz vor. Da unterschiedliche Testdaten verwendet wurden, wird auf einen direkten Vergleich aller Modelle verzichtet.

2.1. Support Vector Machines

Support Vector Machines (SVM) sind ein Machine Learning Ansatz, der Eingabevektoren auf einen hoch-dimensionalen Vektorraum abbildet und eine mit Hilfe eines Stützvektors (engl. support vector) definierte Hyperebene als Grenze für eine Klassifikation zweier Klassen verwendet (vgl. Cortes und Vapnik 1995, S. 273). Abbildung 2.1 veranschaulicht das Prinzip im zwei-dimensionalen Raum.

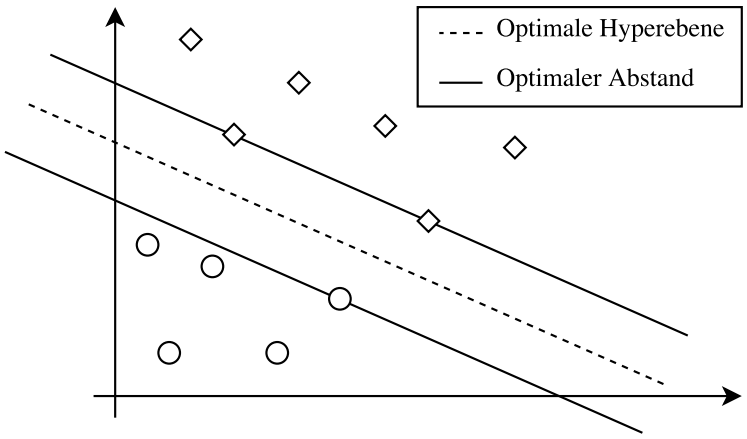

Abbildung 2.1.: Beispiel eines separierbaren Problems im zwei-dimensionalen Raum. (vgl. Cortes und Vapnik 1995, S.275, Figure 2)

Das auf diesem Ansatz aufbauende System *It Makes Sense* spannt einen Vektorraum aus unterschiedlichen Merkmalen auf und sucht für jede Wortart mehrere Stützvektoren, die jeweils eine Bedeutung von allen andern Bedeutungen trennen. Es erreicht eine Genauigkeit von 65.3% auf SensEval-2 und 72.6% auf SensEval-3 Testdaten (vgl. Zhong und Ng 2010, S. 81).

2.2. Neuronale Netze

Neuronale Netze (NN) sind aus einer beliebigen Anzahl Neuronen aufgebaut, die aus einer Aktivierungsfunktion bestehen, dessen Eingabewerte zuvor mit Gewichten und Bias ver-

rechnet werden. (vgl. *Stanford CS class CS231n Course Notes* o.D.) Bei Feed-Forward-Netzes gibt es mehrere Schichten (Hidden Layer) aus Neuronen, dessen Ausgabewerte als Eingabe für alle Neuronen der nächsten Schicht verwendet werden. Abbildung 2.2 zeigt die typische Darstellung solch eines Netzes. Besitzt ein Netz mehrere Hidden Layers, wird es als Deep Neural Network (DNN) bezeichnet.

Gewichte werden zufällig gewählt und schrittweise verbessert, indem eine Kostenfunktion minimiert wird. Dazu werden Gradienten abhängig von den zu optimierenden Variablen (Gewichte und Bias) gebildet, mit einer Lernrate multipliziert und von ihren vorherigen Werten abgezogen.

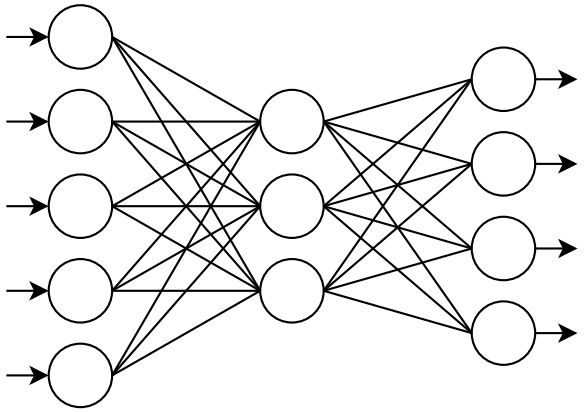

Abbildung 2.2.: Feed-Forward-Netz mit Eingabeschicht (5 Neuronen), Hidden Layer (3 Neuronen) und Ausgabeschicht (4 Neuronen). (vgl. *Wikipedia: Vereinfachte Darstellung eines NN* 2006)

NN sind sehr flexibel und können auf verschiedene Weise für WSD und NED eingesetzt werden. (Z. He et al. 2013) verwenden DNNs, um ein Ähnlichkeitsmaß zwischen Dokumenten und Bedeutungen zu finden. Dazu werden die ersten Schichten des Netzes als stacked denoising auto-encoder vortrainiert und um einen Hidden Layer erweitert, aus dem für Dokument und Bedeutung ein Skalarprodukt gebildet wird. Das Ergebnis entspricht dem Ähnlichkeitsmaß und wird vom NN maximiert. Auf TAC KBP 2010 Daten erreicht der Ansatz eine Gesamtgenauigkeit von bis zu 81,0% und auf AIDA Daten bis zu 85,6%.

DeepType ist ein ebenfalls auf NN basierender NED Ansatz, der in einem ersten Schritt Kategorien findet, über die sich auf die Bedeutung eines Begriffs schließen lässt. Anschließend wird ein NN trainiert, das diese Kategorien für mehrdeutige Begriffe und ihren Kontext vorhersagen kann. Der Ansatz erzielt einen F1 Wert von bis zu 91% auf TAC KBP 2010 Daten. (vgl. J. Raiman und O. Raiman 2018, S. 1,6)

2.3. Graphen

(Pershina, Y. He und Grishman 2015) setzen zur Lösung des Problems Graphen ein und verwenden einen Personalized PageRank, um die wahrscheinlichste Bedeutung für einen Begriff zu finden. Auf AIDA Daten konnte eine Gesamtgenauigkeit von bis zu 91,8% erreicht werden.

3. Modell

Das *ned*-Modul kann Bedeutungen für Begriffe in einem Textsegment finden. Ein Begriff muss dafür in einer Begriffsgruppe enthalten sein, der eine Menge an möglichen Bedeutungen zugewiesen wurde. Zur Bestimmung der wahrscheinlichsten Bedeutung innerhalb dieser Gruppe wird ein feedforward Netz mit einem Hidden Layer verwendet.

Der nächste Abschnitt bietet einen Überblick über den Disambiguationsalgorithmus und die darauffolgenden Abschnitte beschreiben neuronale Netz genauer.

3.1. Überblick

Algorithmus 1 wird verwendet, um mehrdeutige Begriffe in beliebigen Texten zu finden und zu disambiguieren.

Eingabe : Zu disambiguierender Text
Ausgabe : Position und Wikipedia-URL für alle mehrdeutigen Begriffe

```
1  Eingabetext in Absätze unterteilen;
2  für jeden Absatz tue
3      Absatz tokenisieren;
4      Mehrdeutige Begriffe finden;
5      wenn min. 1 mehrdeutiger Begriff gefunden dann
6          Tokens T umwandeln ; // z.B. n-Grams bilden
7          Menge M aller möglichen Bedeutungen für Begriffe im Absatz bilden;
8          Output Layer des NN für T und M berechnen;
9          für jeden mehrdeutiger Begriff tue
10             Ausgabeschicht auf für Begriff relevante Bedeutungen reduzieren;
11             Wikipedia URL für größtes Gewicht ermitteln;
12             URL und Position von Begriff zu Ergebnissen hinzufügen;
13         Ende
14     Ende
15 Ende
```

Algorithmus 1 : Algorithmus zum Finden und Disambiguieren mehrdeutiger Begriffe

Dieses Kapitel beschäftigt sich hauptsächlich mit dem Aufbau des zur Bedeutungsbestimmung verwendeten neuronalen Netzes. Kapitel 7 geht näher auf Implementierungsdetails ein, darunter auch auf Finden von mehrdeutigen Begriffen in Texten. Der Aufbau des Netzes ist in Abbildung 3.1 dargestellt und wird in Abschnitt 3.2 erklärt.

3.2. Aufbau des neuronalen Netzes

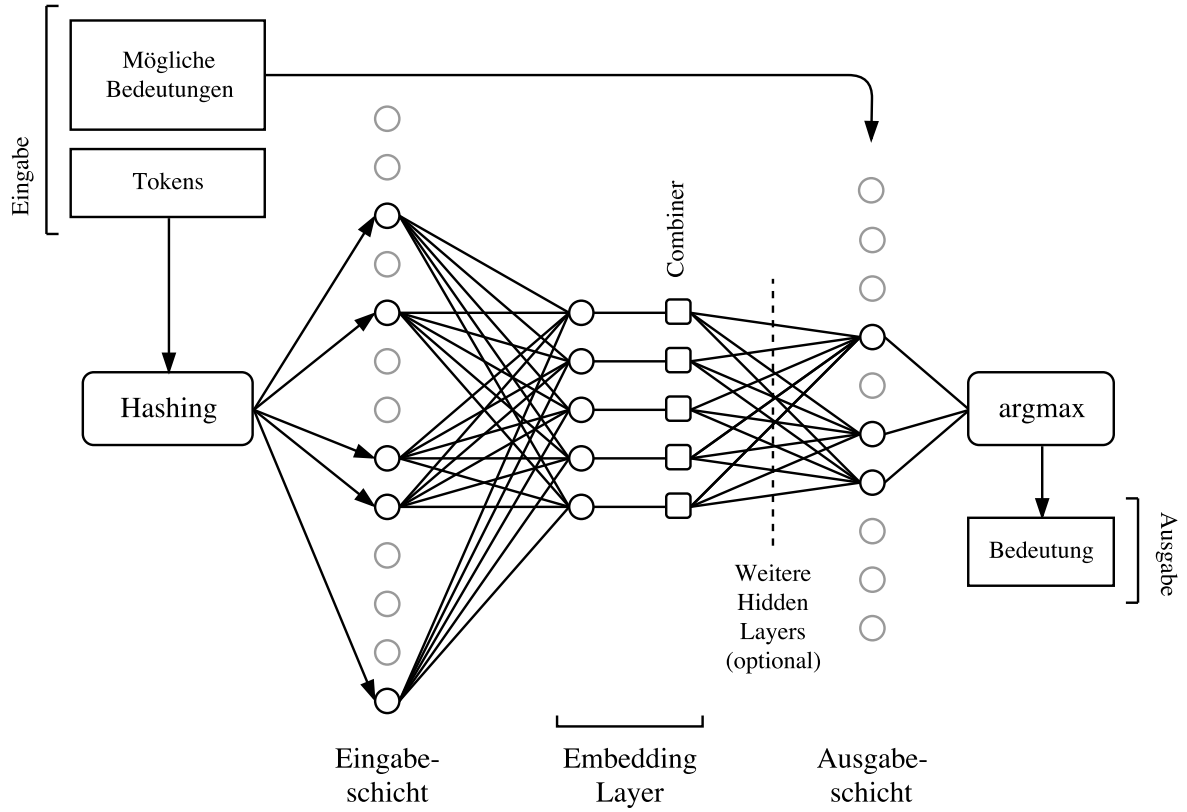

Abbildung 3.1.: Aufbau des NN für ein einzelnes Beispiel

Als Eingabe wird dem NN eine Liste aus Token übergeben, die entweder aus einem kompletten Absatz oder einzelnen Satz stammen. Den Token werden durch eine Hash-Funktion Zahlenwerte zugeordnet und über einen Embedding-Layer in niedrig-dimensionale Vektoren umgewandelt. Sämtliche Embedding-Vektoren eines Textsegments werden zu einem einzelnen Vektor kombiniert, in dem der Durchschnitt aus der Summe gebildet wird. Das Ergebnis wird mit den Gewichten der Ausgabeschicht multipliziert. Diese Architektur entspricht dem von fastText verwendeten NN (vgl. Joulin et al. 2016, S. 1-2).

Optional können Hidden Layers mit ReLU Aktivierungsfunktion zwischen Embedding-Layer und Ausgabeschicht eingefügt werden. Die Anzahl und Größen sind abhängig vom zugehörigen Hyperparameter. Um Overfitting zu vermeiden, kann Dropout (Srivastava et al. 2014) verwendet werden.

Ausgabeklassen entsprechen Bedeutungen, wobei eine Ausgabeklasse immer eindeutig einer Begriffsgruppe zuordenbar sein muss. Bedeutungen die konzeptuell identisch sind, aber in mehreren Begriffsgruppen vorkommen (z.B. „United States“ als Bedeutung für Gruppe „federal“ und Gruppe „USA“), bekommen jeweils eine eigene Ausgabeklasse zugewiesen. In den Trainingsdaten werden Beispiele einer Bedeutung für alle zugehörigen Ausgabeklassen wiederholt.

Um die Bedeutung eines Begriffs in einem Textabschnitt zu ermitteln, wird die Klasse mit dem größten Wert aus allen Ausgabeklassen, die sich in der Gruppe des zu disambiguierenden Begriffs befinden, ausgewählt (vgl. Uslu, Mehler und Baumartz 2018, S. 3).

Während des Trainings werden dem Netz batchweise Paare aus Token und in Frage kommender Bedeutungen übergeben. Aus den möglichen Bedeutungen wird eine Menge gebildet, in der jede Bedeutung nur einmal vorkommt. Die Gewichts- und Biasmatrix der Ausgabeschicht wird für jeden Batch neu gebildet, indem Gewichte und Bias, die nicht in der Menge vorkommen, ignoriert werden. Dadurch entstehen vergleichsweise kleine Gewichts- und Biasmatrizen, mit denen sich schnell rechnen lässt.

Positionen der Ausgabeklassen - also der Bedeutungen - ändern sich in jedem Batch, weshalb die korrekten Bedeutungen aus den Trainingsdaten ebenfalls angepasst werden muss, indem der Index der erwarteten Klasse mit dem Index der equivalenten Klasse in der reduzierten Matrix ersetzt wird.

Trainiert wird das Netz, indem die Kostenfunktion $loss$ (Formel 3.1) mit mit dem Ergebnis x der Ausgabeschicht und den Indexen y der richtigen Bedeutungen eines Batches durch Gradient-Descent-Verfahren mit einer abfallenden Lernrate minimiert wird.

$$loss : \mathbb{R}^{N\times M}, \mathbb{R}^N \to \mathbb{R} \qquad loss(x, y) = \frac{1}{N}\sum_{i=1}^{N} -\log\left(\sigma_{y_i}\left(x_i\right)\right) \tag{3.1}$$

N entspricht der Anzahl der Beispiele in einem Batch. M ist die Anzahl aller in Frage kommender Ausgabeklassen in einem Batch. Die Softmax-Funktion σ_i für $1 < i \leq M$ ist definiert durch Formel 3.2.

$$\sigma : \mathbb{R}^M \to (0, 1] \qquad \sigma_i\left(x\right) = \frac{e^{x_i}}{\sum_{j=1}^{M} e^{x_j}} \tag{3.2}$$

3.2.1. Unterschiede zwischen Training, Test und Anwendung

Eingabedaten unterscheiden sich zwischen Training und realer Anwendung. Beim Training und bei Tests bezieht sich ein Beispiel immer auf genau einen zu disambiguierenden Begriff, wohingegen beim Disambiguieren eines gesamten Textes in jedem Absatz mehrere Begriffe vorkommen können und gemeinsam disambiguiert werden sollen. Daher werden drei unterschiedliche Netze für Training, Test und letztendlich Anwendung verwendet, die alle die selben Gewichte teilen.

Das im vorherigen Abschnitt beschriebene NN wird zum Training verwendet. Das Netz für Tests unterscheidet sich lediglich an der Ausgabeschicht: Durch den Aufbau des Netzes kann die am höchsten gewichtete Ausgabeklasse zu der Bedeutung einer unerwünschten Begriffsgruppe gehören. Daher wird die Differenz des höchsten und niedrigsten Wertes in der Ausgabematrix ermittelt und mit 1 addiert. Dieser Wert wird mit einer binären Maske aller in Frage kommender Bedeutungen multipliziert und zu der Ausgabematrix addiert. Die Operation lässt sich komplett auf einem Grafikprozessor (GPU) - also schnell - ausführen und stellt sicher, dass der höchste Wert nie zu einer ungewollten Ausgabeklasse gehört. Außerdem verschieben sich Indices nicht erneut für jedes einzelne Beispiel.

Beim Training ist dieser zusätzliche Schritt nicht notwendig, da die Kostenfunktion zwischen möglichen und unmöglichen Ergebnissen nicht unterscheidet. Aus Geschwindigkeitsgründen wird immer nur das Ergebnis der Zielklasse im Vergleich zu allen anderen Klassen betrachtet.

Wird das NN zur Disambiguation eines Textes verwendet, besitzen viele Eingaben identische Token und unterscheiden sich lediglich in der Auswahl der Bedeutungen. Daher wird für diesen Fall eine dritte Variante verwendet, die immer genau einen Absatz verarbeiten kann und die Ausgabeschicht für alle in Frage kommenden Bedeutungen aller Begriffe in den Eingabetoken berechnet. Dadurch werden viele Operationen zum Filtern von Bedeutungen überflüssig und können übersprungen werden. Allerdings werden sie teilweise in Python erneut umgesetzt, weshalb es in zukünftigen Version aus Performancegründen sinnvoll seien könnte, das NN um diese Aufgabe zu erweitern.

3.2.2. Vorverarbeitung der Eingabedaten

Text muss zunächst in Textsegmente unterteilt und tokenisiert werden, bevor er vom NN verarbeitet werden kann. Token können zusätzlich auf verschiedene Arten umgewandelt oder kombiniert werden. Folgende Parameter sind beeinflussbar:

- **Segmentgröße**: Um ein Wort oder eine Phrase zu disambiguieren, werden nur die Token aus einem ausgewählten Textsegment betrachtet. Die Größe dieses Segments kann im *ned*-Modul entweder einen Absatz oder einen einzelnen Satz umfassen. Andere Segmentbereiche, wie z.B. eine bestimmte Anzahl umliegender Wörter oder Sätze, wären ebenfalls denkbar, sind allerdings nicht implementiert, um die Parameterkomplexität möglichst gering zu halten.
- **Merkmale**: Token können optional umgewandelt oder gefiltert werden, um zu untersuchen, ob sich dadurch eine höhere Genauigkeit beim Klassifizieren erreichen lässt. Folgende Möglichkeiten stehen einzeln oder in Kombination zur Verfügung:
  - Lemma: Token werden durch Lemmata ersetzt.
  - Wortart (PoS): Die Wortart wird zum Token oder Lemma hinzugefügt. Das Tagset hängt vom verwendeten PoS-Tagger ab.
  - Ohne Satzzeichen: Alle Token für Satzzeichen werden entfernt.
  - Ohne Großbuchstaben: Alle Großbuchstaben werden in Kleinbuchstaben umgewandelt.
- **N-Gram**: In dem beschriebenen Netz hat die Reihenfolge von Token keine Auswirkung auf das Ergebnis, da ein Durchschnittsvektor berechnet wird. Um lokale Zusammenhänge von Wörtern nicht außer Acht zu lassen, können N-Grams gebildet werden. Sie bestehen aus allen N aufeinanderfolgende Token. Zusätzlich werden alle N-Grams für kleinere N gebildet.

  Der Satz „He's late.“ besteht aus 4 Token: „He“, „'s“, „late“ und „.“. Für einen N-Gram Wert von 3 würden folgende Token erzeugt werden: „He“, „'s“, „late“, „.“, „He_'s“, „'s_late“, „late_.“, „He_'s_late“ und „'s_late_.“

3.2.3. Hyperparameter

Neben den Parametern zur Vorverarbeitung von Token, kann der Aufbau und das Training des Netzes durch weitere Parameter beeinflusst werden:

- **Anzahl der Hash-Buckets**: Token müssen in Zahlenwerte umgewandelt werden, um vom NN verarbeitet werden zu können. Durch Hashing ist dies sehr effizient möglich. Die Anzahl der Hash-Buckets entspricht der Anzahl der unterschiedlichen Embedding-Vektoren, die für einzelne Token gebildet werden können. Als Hash-Funktion wurde *tf.string_to_hash_bucket_fast*[1] verwendet.
- **Embeddinggröße**: Die Anzahl der Dimensionen, die jeder Embedding-Vektor besitzt.
- $\sqrt{\mathbf{n}}$ **verwenden**: Im Embedding-Layer werden alle Token eines zu disambiguierenden Textsegments zu Vektoren $v_{1...n}$ umgewandelt und summiert. Um unterschiedlich lange Tokensequenzen auszugleichen, wird die Summe durch einen von der Anzahl der Token abhängigen Wert dividiert und dadurch Vektor v gebildet. Dazu stehen zwei Formeln zur Auswahl:

  Formel 3.3 berechnet den Durchschnitt aller Vektoren.

  $$v = \frac{1}{n} \sum_{i=1}^{n} v_i \tag{3.3}$$

  Formel 3.4, in Tabellen als $\sqrt{n}$ abgekürzt, ändert die Gewichtung der Gesamtsumme.

  $$v = \frac{1}{\sqrt{\mathbf{n}}} \sum_{i=1}^{n} v_i \tag{3.4}$$

  Beide Varianten werden von TensorFlow bereitgestellt (vgl. *Tensorflow API Dokumentation: tf.nn.embedding_lookup_sparse* o.D.).
- **Gradient Clipping**: Wenn aktiviert, werden Gradienten mit einer Länge größer als 1.0 auf genau diese Länge skaliert. (vgl. Pascanu, Mikolov und Bengio 2012, S. 6) Bei frühen Experimenten mit sehr hohen Lernraten und Batchgrößen traten Probleme mit zu großen Gewichten auf, die dazu führten, dass das Softmax-Ergebnis für die Kostenfunktion nicht mehr als 32-bit Float-Wert dargestellt werden konnte, da es unter 10^{-128} lag. Für die letztendlich untersuchten Parameter war Gradient Clipping nicht mehr erforderlich.
- **Lernrate**: Startwert η_0 für Lernrate. Die Lernrate bestimmt den Anteil der Gradienten, die bei jedem Update-Schritt von den Gewichten und Bias abgezogen wird.

[1] `https://www.tensorflow.org/api_docs/python/tf/string_to_hash_bucket_fast`

- **Abnahmerate und Abnahmeschritte**: Die Abnahmerate und Abnahmeschritte bestimmen, wie schnell die Lernrate abnimmt. Die Lernrate η_s ist für Schritt s durch Formel 3.5 gegeben.

$$\eta_s = \eta_0 \cdot Abnahmerate^{s/Abnahmeschritte} \tag{3.5}$$

- **Batchgröße**: Die Anzahl der Beispiele in einem Batch. Für jeden Update-Schritt werden die durchschnittlichen Gradienten aller Beispiele im Batch berechnet.
- **Epochen**: Die Anzahl der kompletten Durchläufe durch die Trainingsdaten.
- **Hidden Layers**: Kombinierte Embedding-Vektoren können durch eine beliebige Anzahl Hidden Layers umgewandelt werden. Ein Hidden Layer h_i mit zugehörigen Gewichten $W_i \in \mathbb{R}^{M \times N}$ und Bias $B_i \in \mathbb{R}^M$ entspricht Formel 3.6.

$$h_i : \mathbb{R}^N \to \mathbb{R}^M \qquad h_i(x) = max(0, W_i x + B_i) \tag{3.6}$$

  Der Wert des Hyperparameters wird als Liste der Ausgabevektorgröße M jedes Hidden Layers angegeben. *[]* wird verwendet, wenn kombinierte Embeddings direkt von der Ausgabeschicht verwendet werden sollen.

- **Dropout**: Anteil der zufällig ausgewählten Neuronen, die in einem reduzierten Netz für Dropout beibehalten werden sollen. Der Wert liegt zwischen 0.0 und 1.0, wobei 1.0 bedeutet, dass Dropout nicht verwendet wird.

3.3. Genauigkeitsmaß

Die Genauigkeit des Modells wird auf zwei Arten erhoben:

- **Micro**: Die Gesamtgenauigkeit wird bestimmt, also der Anteil der richtigen Vorhersagen aus allen Vorhersagen. Sie ist folgendermaßen definiert:

$$\frac{\text{\#Richtig bestimmte Bedeutungen}}{\text{\#Alle bestimmten Bedeutungen}} \tag{3.7}$$

- **Macro**: Die Genauigkeit für jede Bedeutung wird einzeln ermittelt und anschließend der Durchschnitt gebildet. Dadurch werden Bedeutungen mit wenigen Beispielen gleichstark gewertet wie Bedeutungen mit vielen Beispielen.

  Wenn n die Anzahl der Ausgabeklassen ist, dann ist die Macro-Genauigkeit folgendermaßen definiert:

$$\frac{1}{n} \sum_{i=1}^{n} \frac{\text{\#Bedeutung i richtig bestimmt}}{\text{\#Bedeutung i erwartet}} \tag{3.8}$$

3.4. Verworfene Ansätze

Neben dem in Abschnitt 3.2 beschriebenen Ansatz wurden mit weitere Variationen experimentiert, die aus verschiedenen Gründen verworfen wurden.

3.4.1. Eigene Matrixmultiplikation für jedes Beispiel

Anstelle einer gemeinsamen, reduzierten Ausgabeschicht für alle Beispiele in einem Batch, wurde das NN zuerst so implementiert, dass für jedes individuelle Beispiel eine Gewichtsmatrix für alle in Frage kommenden Ausgabeklassen erstellt wurde. Dieser Ansatz entspricht einer Kostenfunktion, die den Durchschnitt der Kosten von N Batches mit Größe 1 bildet, wobei N die eigentliche Batchgröße ist. Es nicht mehr nötig, ungewollte Bedeutungen in den Ausgabematrizen zu maskieren, da nur passende Bedeutungen vorhergesagt werden können. Außerdem haben Gradienten keinen Einfluss auf die Gewichte anderer Bedeutungen. Allerdings ist dieser Ansatz durch den erhöhten Rechenaufwand deutlich langsamer als der letztendlich gewählte Ansatz und wurde deshalb verworfen.

3.4.2. Überschneidende Bedeutungen

Bedeutungen, die in mehreren Begriffsgruppen vorkommen, werden auf unterschiedliche Ausgabeklassen aufgeteilt, so das jede Klasse eindeutig einer Gruppe zuordenbar ist. In einem verworfenen Ansatz wurde auf diese Beschränkung verzichtet und jede Bedeutung wurde immer genau einer Ausgabeklasse zugeordnet. Aufgrund schlechter Ergebnisse in anfänglichen Tests wurden Modelle dieser Form nicht weiter untersucht.

3.4.3. Gewichtete Kostenfunktion

Bedeutungen können zwischen 10 und 205.508 Trainingsbeispiele besitzen. Durch diese Ungleichverteilung werden häufige Bedeutungen bevorzugt ausgewählt. Mit der modifizierten Kostenfunktion $wloss$ (Formel 3.9) wurde versucht, dieses nicht zwangsläufig schlechte Verhalten zu vermeiden.

$$wloss : \mathbb{R}^{N\times M}, \mathbb{R}^N \to \mathbb{R} \qquad wloss(x, y) = \frac{1}{N}\sum_{i=1}^{N} \mathbf{w}_{\mathbf{y_i}} \cdot (-\log(\sigma_{y_i}(x_i))) \tag{3.9}$$

Kosten für ein Beispiel werden mit dem von der Anzahl der Beispiele abhängigen Gewicht w_y (Formel 3.10) multipliziert.

$$w_y = \frac{10}{\text{\#Beispiele in Trainingsdaten für Bedeutung } y} \tag{3.10}$$

Testergebnisse sind in Abschnitt 5.3.4 aufgelistet und werden in Abschnitt 6.2 diskutiert.

4. Trainingsdaten

Aus Verlinkungen in der englischsprachigen Wikipedia (Stand 20.12.2017) wurden 478.077 Artikel-URLs als Bedeutungen für 168.546 Begriffsgruppen ermittelt. Artikel-URLs verweisen entweder auf komplette Artikel oder einzelne Abschnitte, wie z.B. Abschnitt „Academic personnel“ aus dem Artikel „Academy (educational institution)“. Begriffsgruppen bestehen aus einem oder mehreren mehrdeutigen Begriffen, die die selbe Auswahl an Bedeutungen besitzen. (Abbildung 4.1)

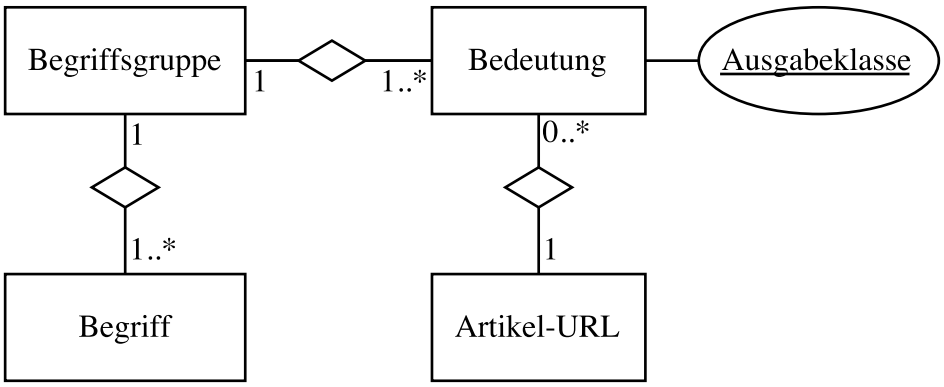

Abbildung 4.1.: ER-Diagramm mit Kardinalitäten

Trainings- und Testkorpora bestehen aus insgesamt 173.236.161 Beispielen. Die Datenerstellung wird in den nächsten Abschnitten erklärt. Weitere Details zur Implementation befinden sich in Abschnitt 7.

4.1. Finden von mehrdeutigen Begriffen

Wie bei fastSense, wird für die Liste aller mehrdeutigen Begriffe die Wikipedia-Kategorie „Disambiguation pages“ als Grundlage verwendet, auf dessen Seiten Artikel zu unterschiedlichen Bedeutungen eines Begriffs verlinkt sind. Seitentitel werden vereinheitlicht und einer neuen oder einer bereits bestehenden Begriffsgruppe mit übereinstimmenden Titel hinzugefügt. Etwa 1.500 Begriffsgruppen sind dadurch mehr als einer Disambiguations-Seite zuordenbar. Die Titelanpassung besteht aus drei Schritten:

- Vorangestellten Artikel („the“, „a“ und „an“) entfernen.
- Text in Klammern am Titelende entfernen. „f(x) (disambiguation)“ wird z.B. zu „f(x)“.
- Großbuchstaben in Kleinbuchstaben umwandeln.

Durch Weglassen von Artikeln (Wortart) können Informationen verloren gehen, über die auf die Bedeutung eines Wortes geschlossen werden kann. Wird über „The River“ gesprochen, ist wahrscheinlich Literatur oder ein Film gemeint, wohingegen „River“ alleine sich ehr auf einen Fluss bezieht. Um diese Unterscheidung auszunutzen, müsste allerdings der vorangestellte Artikel zuverlässig als Teil eines Begriffs erkennbar sein und auch immer der vollständige Titel in Texten verwendet werden. Um mögliche Fehlerquellen und Komplexität einer strengeren Vorauswahl zu vermeiden, übernimmt das NN diese Aufgabe. Gruppentitel werden nur zum Finden von mehrdeutigen Begriffen verwendet und entscheiden, welche Bedeutungen möglich sind. Der Gruppentitel an sich wird nicht zur Disambiguierung verwendet, da er nicht Teil der Eingabe des neuronalen Netzes ist.

Eingeklammerte Zusätze am Ende von Titeln werden entfernt, da sie entweder eine Disambiguation kennzeichnen (z.B. „Apple (disambiguation)“) oder einen Begriff weiter disambiguieren (z.B. „Return (programming)“). Klammern, die ohne Leerzeichen dem Titel folgen (z.B. „O(n)“), werden nicht entfernt.

Kleinbuchstaben werden verwendet, da Groß- und Kleinschreibung bei der Suche von Gruppentiteln in Texten ignoriert wird. Probleme durch unterschiedliche Schreibweisen eines Begriffs (z.B. „DVD“ und „Dvd“) werden dadurch vermieden.

Für Begriffsgruppen können über Titel von Weiterleitungen auf Disambiguationsseiten mehr als 170.000 Synonyme gefunden werden. Die Seitentitel werden wie beschrieben umgewandelt und Begriffsgruppen als zusätzliche Begriffe zugeordnet. Dadurch wird z.B. nicht nur das Wort „alien“ als mehrdeutiger Begriff erkannt, sondern auch „aliens“, da die Seite „Aliens“ auf die Disambiguationsseite „Alien“ weiterleitet. Titel, die im Konflikt zu bestehenden Begriffen stehen, werden ignoriert.

4.2. Finden von Bedeutungen

Nachdem alle mehrdeutigen Begriffe feststehen, können ihnen Bedeutungen zugeordnet werden. Dazu werden die Links auf Disambiguationsseiten untersucht. Alle Links, deren Titel den vereinheitlichten Seitentitel enthalten, werden als Bedeutung ausgewählt. Links, die sich im teilweise vorhandenem Abschnitt „See also“ befinden, werden ignoriert, da sie meist irrelevant sind. FastSense geht bis zu diesem Punkt genauso vor.

Nicht immer sind Disambiguationsseiten vollständig und teilweise werden erwünschte Links ausgeschlossen, wie z.B. der Link zu „Extraterrestrial life“ auf der Seite „Alien“. Diese Fehler werden korrigiert, indem Titel von Links in der gesamten Wikipedia untersucht werden. Nachdem Weiterleitungen aufgelöst wurden, wird überprüft, ob min. 5 Links existieren, deren vereinheitlichter Titel mit einer Begriffsgruppe übereinstimmt und die alle das selbe Ziel besitzen. Ist dies der Fall, wird der verlinkte Artikel oder Abschnitt ebenfalls als Bedeutung zur Begriffsgruppe hinzugefügt.

Bedeutungen, für die weniger als 14 Beispiele gefunden werden können, werden verworfen. Jede Bedeutung besitzt so min. 10 Trainings- und zwei mal 2 Testbeispiele. Etwa 40% aller Begriffsgruppen haben durch das Kriterium nur noch eine Bedeutung. Diese Gruppen werden nicht entfernt, da sie möglicherweise positive Auswirkung auf das Training der Embedding-Vektoren haben könnten und nur für einen kleinen Anteil aller Beispiele verant-

wortlich sind (siehe Abschnitt 5.2).

Damit sich Gewichte und Bias in der reduzierten Ausgabeschicht des NN nicht überschneiden, wenn die selbe Bedeutung (d.h. Wikipedia-URL) in unterschiedlichen Begriffsgruppen vorkommt, wird jede Bedeutung immer als Paar aus Wikipedia-URL und Begriffsgruppe betrachtet, dem eine eigene Ausgabeklasse im NN zugewiesen wird.

Eine Begriffsgruppe besitzt im Schnitt 3,3 Bedeutungen. Bei 17 der 25 zahlenmäßig Bedeutungsreichsten Begriffe handelt es sich um Jahreszahlen. An der Spitze befindet sich „2010“ mit 468 Bedeutungen. Wenn Bedeutungen nur über Disambiguationsseiten gesucht werden, sinkt die durchschnittliche Anzahl an Bedeutungen auf 2,7 und die Gruppe „star“ besitzt mit 72 die höchste Anzahl an Bedeutungen.

4.3. Finden von Beispielen

Einem Absatz oder einzelnem Satz werden Bedeutungen über im Text enthaltene Links zugewiesen. Existieren Links zu Artikeln, die als Bedeutung markiert wurden, wird der Absatz als Beispiel für jeden dieser Artikel gekennzeichnet. Weiterleitungen werden aufgelöst, bevor URLs verglichen werden. Wie bei fastSense werden Absätze und Sätze aus Artikeln zu einer Bedeutung, unabhängig von enthaltenen Links, immer als Beispiel für sich selbst markiert.

Für jeden markierten Absatz lassen sich dadurch alle Bedeutungen über die Artikel-URLs zuordnen. Es können auch mehrere Bedeutungen für einen Link existieren, wenn die URL mehr als einer Begriffsgruppen zugewiesen wurde.

Beispiele werden für jeweils ein Paar aus Absatz und Bedeutung generiert und enthalten folgende Ein- und Ausgabewerte für das NN:

- **Token**: Eine Menge aus Token, die mit den in Abschnitt 3.2.2 beschrieben Parametern für einen Satz oder Absatz erzeugt wurden.
- **Mögliche Bedeutungen**: Alle Ausgabeklassen für Bedeutungen, die der zu disambiguierende Begriff annehmen kann.
- **Bedeutung**: Die Ausgabeklasse für die Bedeutung des untersuchten Begriffs. Die Bedeutung muss in den möglichen Bedeutungen vorkommen.

Nicht alle Teile der Wikipedia sind zur Disambiguation geeignet. Um die Datenqualität zu steigern, wird eine Vorauswahl getroffen. Zu den ignorierten Inhalten gehören:

- Tabellen
- Fast alle Templates (siehe Abschnitt 7.3.2)
- Absätze, die aus weniger als 50% Buchstaben bestehen
- Irrelevante Wikicode-Tags, wie unter anderem *<ref>* für Quellangaben, *<math>* für mathematische Formeln und *<syntaxhighlight>* für Programmcode
- Absätze, die Wikicode- oder HTML-Tags enthalten

- Absätze, die kürzer als als 15 Zeichen sind oder weniger als 4 Leerzeichen enthalten

Absätze, die nach Tokenisierung aus weniger als fünf Token bestanden, werden ebenfalls ignoriert. Insgesamt blieben 59.333.502 Absätze (oder 128.689.963 Sätze) übrig, von denen 31.842.587 in Beispielen verwendet wurden.

4.4. Aufteilung in Datensets

Beispiel werden auf drei Korpora aufgeteilt:

- **train**: Beispiele für Training
- **dev**: Beispiele für Tests zum Optimieren der Hyperparameter
- **test**: Beispiele für Tests nach Optimierung der Hyperparameter

Alle Beispiele einer Bedeutung werden gesucht, zufällig gemischt und die Anzahl der gewünschten Beispiele in einem Testkorpus mit *round(Anzahl * 0.15)* berechnet. Aus den gemischten Beispielen wird für *dev* und *test* die benötigte Anzahl vom Anfang der Liste entnommen. Die übrigen Beispiele werden *train* zugeteilt.

Zusätzlich kann für jeden Korpus eine zweite Variante generiert werden, die Sätze anstelle von Absätzen verwendet. Dazu wird vorher für jedes Beispiel der Index des Satzes gespeichert, in dem sich der Link zur Bedeutung befindet (sofern vorhanden). Die genaue Anzahl und Aufteilung der Beispiele ist in Tabelle 4.1 zu finden.

Tabelle 4.1.: Anzahl der Beispiel in Trainings- und Testkorpora

Art	Korpus	Beispiele	%
Absatz	train	121.275.847	70%
	dev	25.980.157	15%
	test	25.980.157	15%
Satz	train	138.408.867	70%
	dev	29.646.082	15%
	test	29.646.053	15%

5. Experimente

Es wurde die Auswirkung von Parametern auf die Genauigkeit des Modells untersucht. Kapitel 5 stellt die Vorgehensweise und Ergebnisse vor, die im darauf folgendem Kapitel 6 diskutiert werden.

5.1. Vorgehensweise

Aufgrund des großen Trainingskorpus und der damit verbundenen Trainingszeit von etwa 2,5 Stunden pro Epoche, konnte keine vollständige Parameterstudie durchgeführt werden. Stattdessen wurden einzelne Parameter nacheinander gezielt angepasst und zusammen mit den zuvor besten Parametern getestet.

Zunächst wurden Parameter ohne zusätzliche Hidden Layers optimiert. Dabei wurde überwiegend die Auswirkung von Batch- und Embeddinggröße untersucht. Anschließend konnte das Ergebnis durch Hinzufügen von nicht linearen Hidden Layers verbessert werden.

5.2. Baseline

Für die Testkorpora wurde die Genauigkeit folgender zwei Ansätze berechnet:

- **Most Frequent Sense (MFS)**: Für jeden Begriff wird immer die häufigste Bedeutung ausgewählt, also die Bedeutung mit den meisten Beispielen in der zugehörigen Begriffsgruppe. Dieses Modell dient als Baseline.
- **Mindestgenauigkeit (MIN)**: Für Begriffe mit mehr als einer Bedeutung wird immer eine falsche Auswahl getroffen.

Sowohl *dev*, als auch *test* besitzen die gleiche Genauigkeit für jeweils MFS und MIN, da sich die Anzahl der Beispiele für jede Bedeutung nicht unterscheidet.

Tabelle 5.1.: Genauigkeit von MFS und MIN

Model	Micro	Macro
Most Frequent Sense (MFS)	59,1%	40,5%
Mindestgenauigkeit (MIN)	5,7%	12,4%

Die Genauigkeit für MIN liegt nicht bei 0%, da Gruppen mit nur einer Bedeutung - wie bereits in Abschnitt 4.2 beschrieben - nicht aus den Trainingsdaten ausgeschlossen werden und deshalb nie falsch klassifiziert werden können.

5.3. Ergebnisse

Die Ergebnisse für die aus der Wikipedia generierten Korpora werden in den folgenden Abschnitten vorgestellt. Abschnitt 5.3.5 bezieht sich auf SemCor- und Senseval-Daten.

5.3.1. Englische Wikipedia

Für beide Testkorpora (*dev* und *test*) konnte eine Genauigkeit von bis zu **89,5%** (Micro), bzw. **77,9%** (Macro) erreicht werden. Für das beste Ergebnis wurden die Parameter aus Tabelle 5.2 beim Training verwendet.

Tabelle 5.2.: Optimierte Hyperparameter

Parameter	Wert
Segmentgröße	Absätze
Token	Token ohne Satzzeichen
N-Gram	2
Anzahl der Hash-Buckets	10.000.000
Embeddinggröße	25
$\sqrt{\mathbf{n}}$ verwenden	Ja
Gradient Clipping	Nein
Lernrate	1.0
Abnahmerate	0,98
Abnahmeschritte	100.000
Batchgröße	32
Epochen	8
Hidden Layers	[25]
Dropout	Nein

Das NN wurde solange trainiert, bis die Kosten für *dev* konvergierten. Tests wurden nach jeder Epoche durchgeführt. Abbildung 5.1 zeigt den Trainingsverlauf sowohl mit, als auch ohne Hidden Layers.

5.3.2. Parameteroptimierung

Tabelle C.2 enthält alle Testergebnisse für *dev*, die mit verschiedenen Parametern ohne Hidden Layers erreicht wurden. Aufgrund des Rechenaufwands wurden nur Ergebnisse nach einer Epoche untersucht und nicht die darauf folgende Steigung. Die Rangfolge der Parameterkombinationen könnte sich nach weiteren Epochen verändern, wie Abbildung 5.1 zeigt.

Aufgrund eines Fehlers unterscheiden sich die verwendeten Daten geringfügig von den zuvor beschriebenen Daten:

- In Beispielen, die aus mehr als einem Satz bestehen, wurde der erste Satz verdoppelt und der letzte Satz entfernt. Ein Absatz „A. B. C. D.“ wurde zu „A. A. B. C.“ umgewandelt. Absätze bestehend aus einem einzelnen Satz blieben unverändert.

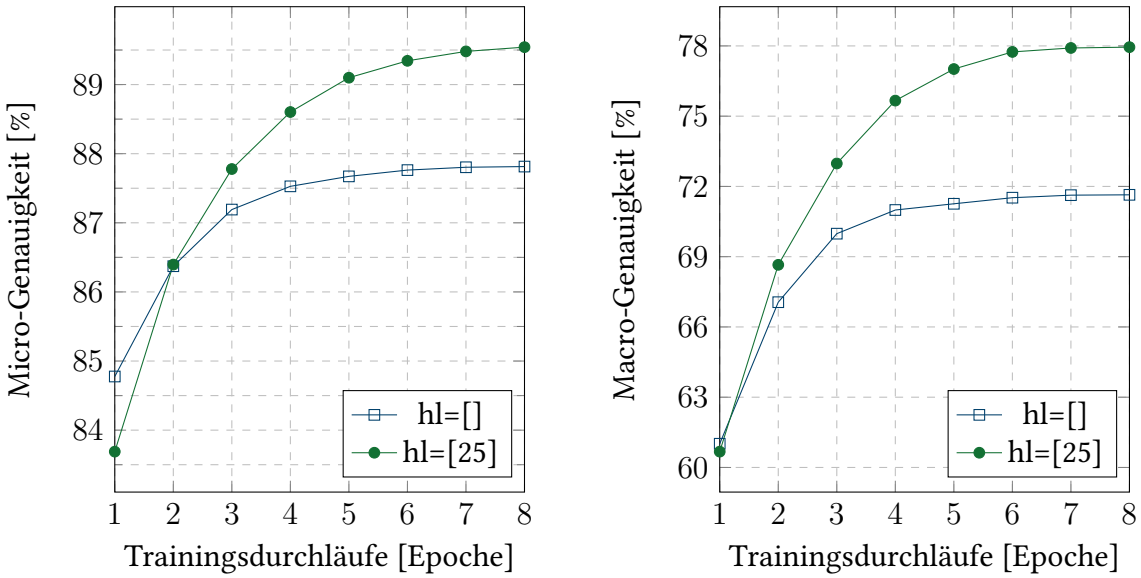

Abbildung 5.1.: Genauigkeit auf *dev*-Daten; hl = Hidden Layers

- Absätze durften aus min. 50% Buchstaben **und Leerzeichen** bestehen. Dadurch wurden Beispiele wie „ . . . . .“ (50% Leerzeichen, 50%, Satzzeichen) nicht herausgefiltert.

Dieses Problem beschränkt sich auf die Ergebnisse in Tabelle C.2. Tabelle C.1 enthält weitere Ergebnisse, die nicht betroffen sind. Für nicht angegebene Parameter wurden die Werte aus Tabelle 5.2 verwendet.

5.3.3. Vergleich von Merkmalen

Die Auswirkung von unterschiedlichen Merkmalen wurde in Tabelle 5.3 untersucht. Es wurden die optimierten Parameter ohne Hidden Layers verwendet.

Tabelle 5.3.: *dev*-Testergebnisse für unterschiedliche Merkmale

	1 Epoche		2 Epochen		3 Epochen	
Feature	Micro	Macro	Micro	Macro	Micro	Macro
Token (np)	**84,78%**	**61,01%**	86,37%	67,06%	**87,19%**	69,98%
Token	84,59%	60,86%	86,34%	67,49%	87,08%	70,10%
Token (np, lc)	84,56%	60,66%	**86,44%**	**67,69%**	87,07%	69,86%
Token (lc)	84,49%	60,89%	86,18%	67,36%	87,03%	**70,20%**
Token + PoS	84,45%	60,37%	86,20%	66,96%	87,02%	69,75%
Lemma	84,39%	60,54%	86,21%	67,36%	87,00%	70,16%

np = No punctuation/Keine Satzzeichen, **lc** = Lowercase/Nur Kleinbuchstaben

5.3.4. Gewichtete Kostenfunktion

Die gewichtete Kostenfunktion aus Abschnitt 3.4.3 führt zu einer deutlichen Verschlechterung der Gesamtgenauigkeit. Nach 5 Epochen können nur **54,0%** aller Bedeutungen aus den Testdaten richtig bestimmt werden, also 33,6% weniger als unter Verwendung der ungewichteten Kostenfunktion mit identischen Parametern. Die abnehmende Steigung der Genauigkeit während des Trainingsverlaufs lässt darauf schließen, dass auch nach weiteren Epochen keine deutlich höheren Werte zu erwarten sind. (Tabelle 5.4) Die durchschnittliche Genauigkeit pro Bedeutung verschlechtert sich vergleichsweise gering um 7,7% auf **64,1%** und liegt damit noch immer über der Baseline.

Tabelle 5.4.: Ergebnisse für *dev*-Testset nach Training mit gewichteter Kostenfunktion

Features	NG	S	E	$\sqrt{n}$	LR	DR	HL	Batch	Epoche	Micro	Macro
Token	2		25	✓	1	0,98	[]	32	5	54,0%	64,1%
Token	2		25	✓	1	0,98	[]	32	4	53,8%	63,3%
Token	2		25	✓	1	0,98	[]	32	3	52,6%	61,6%
Token	2		25	✓	1	0,98	[]	32	2	49,8%	57,8%
Token	2		25	✓	1	0,98	[]	32	1	42,6%	48,8%
Token (np)	2		25	✓	1	0,98	[]	32	3	52,1%	61,2%
Token (np)	2		25	✓	1	0,98	[]	32	2	49,3%	57,3%
Token (np)	2		25	✓	1	0,98	[]	32	1	43,1%	48,2%

np = Keine Satzzeichen, **NG** = n-Gram, **S** = Sätze, **E** = Embeddinggröße, $\sqrt{\mathbf{n}}$ = Erklärt in Abschnitt 3.2.3, **LR** = Lernrate, **DR** = Abnahmerate, **HL** = Hidden Layers, **Batch** = Batchgröße

5.3.5. Senseval und SemEval

Weitere Modelle wurden mit Beispielen aus SemCor 3.0 trainiert, um zu zeigen, dass der Ansatz auch für WSD geeignet ist. Die von (Uslu, Mehler und Baumartz 2018) für fastSense konvertierten Daten wurden erneut in ein kompatibles Format umgewandelt. Getestet wurde auf Senseval 2 (English all-words) (SE2) und Senseval 3 (English all-words) (SE3) sowie SemEval-2007 Task 17 Subtask 1 (SE7), SemEval-2013 Task 12 (SE13) und SemEval- 2015 Task 13 (SE15) (vgl. Uslu, Mehler und Baumartz 2018, S. 5).

Die Gewichte und Bias wurden in jedem Modell zufällig initialisiert. Anders als bei fastSense, wurden die für Beispiel aus der englischen Wikipedia optimierten Parameter verwendet und keine Parameterstudie für SemCor durchgeführt. Da die Lernrate von 1,0 für die Daten ungeeignet hoch ist und teilweise Probleme beim Berechnen der Gradienten verursacht, wurde Gradient Clipping verwendet. Beispiele enthalten größtenteils keine Punkte, allerdings können vereinzelt Satzzeichen auftreten, die nicht entfernt wurden.

Modelle wurden bis zur Erfüllung einer Abbruchbedingung unterschiedlich lange trainiert, um Overfitting zu reduzieren. Nach jeder Epoche wurde der gleitende Durchschnitt der Kostenfunktion für die Testdaten berechnet. Das Training wurde beendet, sobald der

Wert aller Testsets vom negativen in den positiven Bereich überging. Aufgrund der Form der vorliegenden Daten wurden jeweils eigene Netze für Senseval und SemEval trainiert.

Im Vergleich zu fastSense konnte die Genauigkeit für Senseval auf bis zu 90,6% (Tabelle 5.6) gesteigert werden. Für SemEval konnten nur teilweise bessere Ergebnisse erzielt werden (Tabelle 5.7).

Obwohl mehrfach eine hohe Micro-Genauigkeit erreicht wurde, lag die durchschnittliche Macro-Genauigkeit bei allen Tests zwischen 0,7% und 1,7%, da ein hoher Anteil an Bedeutungen aus den Trainingsdaten in den Testdaten fehlt. Für Bedeutungen ohne Testdaten wurde eine Genauigkeit von 0% festgelegt. Die höchste erreichbare Macro-Genauigkeit für jeden Testkorpus ist in Tabelle 5.5 aufgelistet.

Tabelle 5.5.: Höchste erreichbare Macro-Genauigkeit

SE2	SE3	SE7	SE13	SE15
1,96%	1,75%	0,86%	1,60%	1,14%

Tabelle 5.6.: Testergebnisse für Senseval-Daten

	N-Gram	Epochen	SE2		SE3	
			Micro	Macro	Micro	Macro
Token	1	95	**90,6%**	**1,72%**	-	-
	2	79	88,9%	1,68%	-	-
Lemma	1	73	88,6%	1,67%	-	-
	2	88	88,2%	1,67%	-	-
Token + PoS	1	66	88,7%	1,67%	86,1%	1,53%
	2	87	88,0%	1,67%	87,2%	1,53%
Lemma + PoS	1	76	88,7%	1,69%	**88,9%**	**1,58%**
	2	100	88,5%	1,67%	87,3%	1,53%

Tabelle 5.7.: Testergebnisse für SemEval-Daten

	N-Gram	Epochen	SE7		SE13		SE15	
			Micro	Macro	Micro	Macro	Micro	Macro
Token	1	64	**86,2%**	**0,72%**	80,8%	1,24%	77,4%	**0,84%**
	2	76	84,1%	0,70%	81,2%	**1,24%**	**78,8%**	0,83%
Token + PoS	1	47	-	-	**84,2%**	0,96%	-	-
	2	52	-	-	80,5%	0,92%	-	-

6. Diskussion der Ergebnisse

Die folgenden Abschnitte gehen näher auf die Ergebnisse ein und diskutieren Beobachtungen, Probleme und Verbesserungsmöglichkeiten.

6.1. Intuition

Um die Auswirkung verschiedener Parameter besser zu verstehen, ist es hilfreich, für Embeddings einen niedrig-dimensionalen Vektorraum zu wählen und diesen zu visualisieren. Abbildung 6.1 bildet einen zwei-dimensionalen Vektorraum ab, in dem sich zufällig initialisierte Embedding-Vektoren - dargestellt durch blaue Punkte - für 60 Hash-Buckets befinden. Das neuronale Netz wird wiederholt mit einem Batch aus acht Beispielen für drei verschiedene Bedeutungen aus der selben Begriffsgruppe trainiert. Das Ergebnis des Embedding-Layers, also die Kombination aus allen Embedding-Vektoren für Token eines Beispiels, werden durch rote Kreuze markiert.

Die Batch-abhängige Ausgabeschicht legt über ihre Gewichte und Bias die Grenzen (rotblaue Linien) zwischen den Bedeutungen fest. Da immer die selben Beispiele verwendet werden, unterscheidet sich die Auswahl der Gewichte in der Ausgabeschicht nicht zwischen Trainingsschritten. Betrachtet man die Position der Vektoren im zeitlichen Verlauf, ist zu erkennen, wie sich die Embedding-Vektoren anordnen, um Cluster aus kombinierten Vektoren zwischen den Entscheidungsgrenzen zu bilden.

Zusätzliche Hidden Layers ermöglichen nicht lineare Entscheidungsgrenzen, mit denen Cluster leichter voneinander getrennt werden können.

6.2. Ungleichmäßige Verteilung von Beispielen

Bedeutungen in den Trainings- und Testdaten besitzen teilweise eine stark unterschiedliche Anzahl an Beispielen. In den Wikipedia-Daten können für die häufigsten Bedeutungen (Tabelle 6.1) bis zu 21.000 mal mehr Beispiele gefunden werden, als für Bedeutungen an der Mindestgrenze von 14 Beispielen. Zusätzlich werden Beispiele wiederholt, wenn Bedeutungen sich in mehreren Begriffsgruppen befinden. Diese ungleichmäßige Verteilung führt dazu, dass häufige Bedeutungen anfänglich bevorzugt ausgewählt werden. In Abbildung 6.1 ist dieses Verhalten nach Schritt 10 gut erkennbar.

Auch im trainierten Modell ist dieses Verhalten beobachtbar. Die Differenz der Genauigkeit zwischen der ersten und letzten Epoche in Abbildung 5.1 unterscheidet sich zwischen Micro und Macro-Genauigkeit stark. Die Gesamtgenauigkeit (Micro) steigt lediglich um etwas über 3%, wohingegen die durchschnittliche Genauigkeit pro Bedeutung (Macro) um fast 11% steigt. Wenige Bedeutungen mit sehr vielen Beispielen scheinen eine starke Auswirkung auf die Anfangsgenauigkeit zu haben. Zwischen anderen Bedeutungen wird erst im weiteren Trainingsverlauf genauer unterschieden.

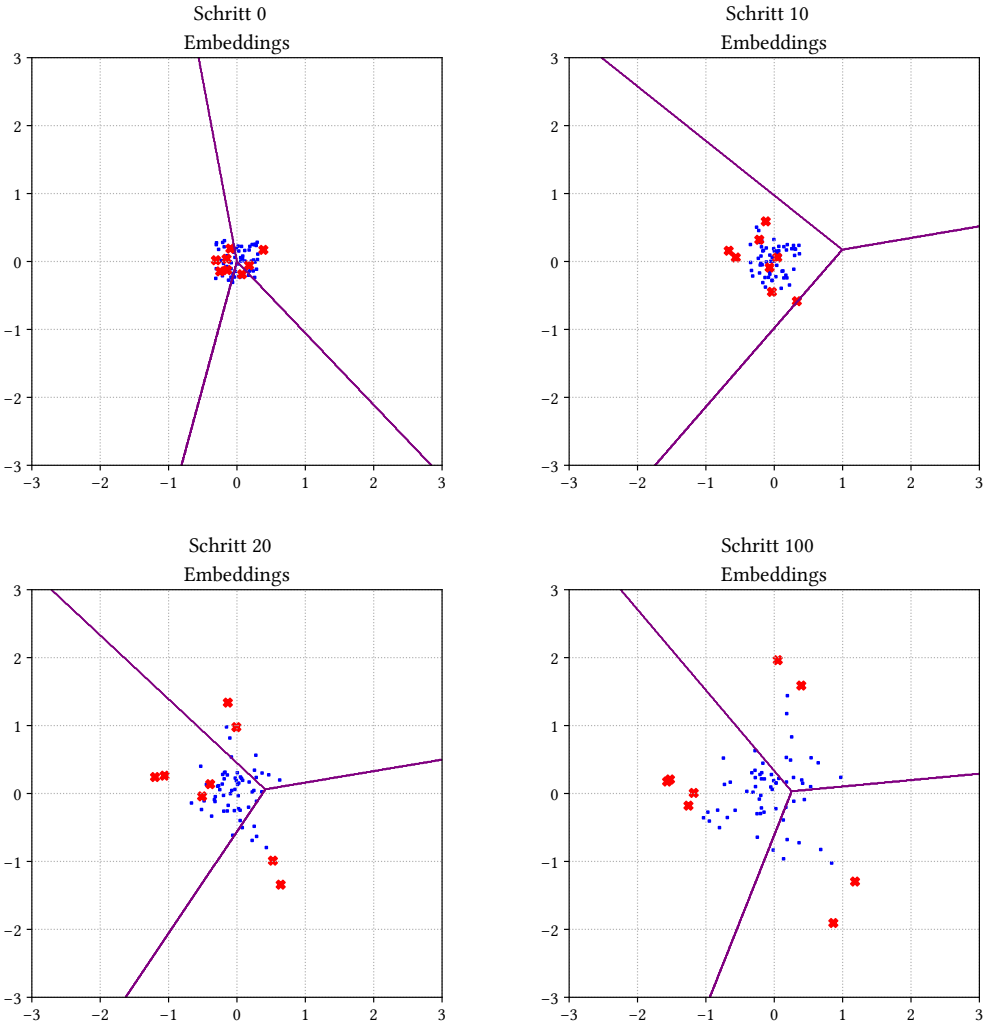

Abbildung 6.1.: Einfaches Modell zur Veranschaulichung. Embeddings (blau), Beispiele (rot) und Bedeutungsgrenzen (rotblau) in 2-D Vektorraum.

Mit der gewichteten Kostenfunktion aus Abschnitt 3.4.3 wurde der Trainingsverlauf beeinflusst, sodass diese anfängliche Verschiebung der Bedeutungsgrenzen kaum stattfindet. Obwohl die Genauigkeit für alle Beispiele stark bis unter die Baseline fällt, verschlechtert sich die Genauigkeit für einzelne Bedeutungen im Durchschnitt nur um weniger als 8%. Es ist möglich, dass optimierte Parametern oder eine andere Gewichtungsfunktion bessere Ergebnisse erzielen, aufgrund der deutlichen Verschlechterung wurde dieser Ansatz jedoch nicht weiter untersucht.

Sinnvoller erscheint es, unterschiedliche Netze mit jeweils einer Teilmenge an Begriffsgruppen zu trainieren. Abhängig von der Aufteilung, müssten Embedding-Vektoren weniger Bedeutungen unterschiedlicher Bereiche abbilden und werden so weniger stark von unpassenden Bedeutungen mit vielen Beispielen beeinflusst. Die tatsächliche Auswirkung müsste in einer weiterführenden Arbeit untersucht werden.

Tabelle 6.1.: Bedeutungen mit den meisten Beispielen.

Bedeutung	Gruppen	Beispiele
United States	19	293582
Association football	17	181512
World War II	9	157995
France	9	130567
Germany	10	108334
India	8	104834
New York City	11	101986
United Kingdom	15	96856
Canada	8	95190
Iran	6	91967

6.3. Sätze vs. Absätze

Absätze eignen sich besser zur Disambiguation als einzelne Sätze. Modelle, die mit Sätzen arbeiten, erreichten wiederholt eine schlechtere Genauigkeit als vergleichbare Modelle, die Absätze verwenden. Mit den besten getesteten Parametern erreichten Absätze im Vergleich zu Sätzen nach der ersten Epoche eine um 6,5% bessere Gesamtgenauigkeit, bzw. 4,9% bessere durchschnittliche Genauigkeit pro Bedeutung.

6.4. Batchgröße

Die Anzahl der Beispiele in einem Batch hat eine spürbare Auswirkung auf die Trainingszeit, da sie mit jeder Halbierung die Anzahl der Trainingsschritte verdoppelt. Abbildung 6.2 zeigt die durchschnittliche Trainingsdauer für verschiedene Batchgrößen.

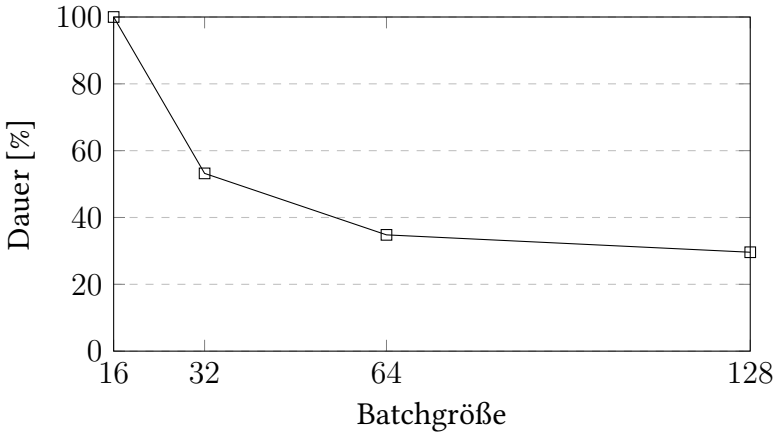

Abbildung 6.2.: Durchschnittliche Dauer einer Epoche

Tabelle 6.2 gruppiert die Ergebnisse aus Tabelle C.2 nach Batchgröße. Obwohl die Ergeb-

nisse von anderen Parametern abhängen und nicht gleichmäßig verteilt sind, lässt sich die Auswirkung der Batchgröße abschätzen. Unter Miteinbeziehung der Trainingszeit scheint ein sinnvoller Wert zwischen 16 und 32 Beispielen zu liegen.

Tabelle 6.2.: Durchschnitt der Genauigkeiten aller Tests aus Tabelle C.2 für Modelle, die Absätze verwenden. Gruppiert nach Batchgröße.

Batchgröße	#Modelle	Avg. Micro	Avg. Macro
4	1	81,6%	47,8%
16	2	**83,7%**	**56,5%**
32	6	80,5%	55,1%
64	21	78,1%	50,9%
128	1	75,7%	47,3%

Avg. Micro = Durchschnittliche Micro-Genauigkeit,
Avg. Macro = Durchschnittliche Macro-Genauigkeit

6.5. $\sqrt{n}$

Die alternative Formel zur Kombination von Embedding-Vektoren aus Abschnitt 3.2.3 erzielte in frühen Tests deutlich bessere Ergebnisse (über 10%) für die Macro-Genauigkeit und wurde deshalb für alle weiteren Tests verwendet.

Der Grund für die Verbesserung lässt sich im niedrig-dimensionalen Vektorraum erahnen. Abbildung 6.3 zeigt Embeddings von zwei identisch initialisierten Netzen nach 100 Trainingsschritten.

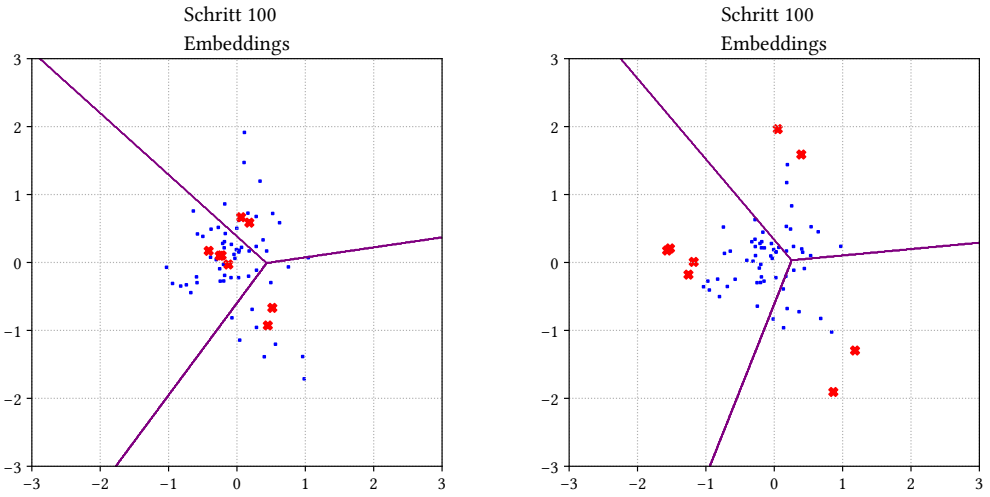

Abbildung 6.3.: Embeddings nach 100 Schritten. Links: Durchschnitt; Rechts: $\sqrt{n}$

Die linke Abbildung verwendet Formel 3.3, also den Durchschnitt aller Token in einem Beispiel, während die rechte Abbildung Formel 3.4 ($\sqrt{n}$) verwendet. Erwartungsgemäß liegen die kombinierten Vektoren (rot) in der linken Abbildung zwischen den teilweise sehr langen Embedding-Vektoren (blau). In der rechten Abbildung sind Embeddings weniger stark in eine Richtung ausgeprägt, dafür aber die Kombinationsvektoren, wodurch Entscheidungsgrenzen leichter gezogen werden können und einzelne Embeddings insgesamt weniger Auswirkung auf eine Entscheidung haben.

Der Durchschnitt ist im Gegensatz zur Summe aller Embeddings unabhängig von der Tokenanzahl eines Beispiels, kann allerdings auch nie länger als der längste enthaltene Vektor sein. Teilen der Summe durch $\sqrt{n}$ führt zu einem Mittelweg zwischen Durchschnitt und Summe, da Beispiele mit besonders vielen Token nicht automatisch einen deutlich längeren Vektor bilden als Beispiele mit wenigen Token, die Länge des Ergebnisvektors aber nicht komplett eingeschränkt wird.

6.6. Datenqualität

Die Datenqualität ist insbesondere bei Bedeutungen mit wenig Beispielen wichtig. Bei der Auswahl von Beispielen wurden deshalb mehrere Millionen Absätze ausgelassen. Da viele Bedeutungen mit nur zehn Beispielen trainiert werden, können bereits wenige irrelevante Beispiele das Ergebnis verändern. Des Weiteren besitzen diese Bedeutungen nur zwei Beispiele in den Testdaten, was dazu führt, dass ein einzelnes falsches Beispiel eine Verschlechterung von bis zu 50% verursachen kann. Auf die Macro-Genauigkeit kann dies eine große Auswirkung haben.

6.7. Beispiele mit mehreren Bedeutungen

Ein Absatz aus der Wikipedia kann mehreren Bedeutungen zugewiesen bekommen, wenn er mehr als einen Link enthält oder ein Link auf eine Bedeutung in mehreren Begriffsgruppen verweist. Da das NN kein Training mit mehreren Bedeutungen für ein Beispiel unterstützt, werden Token für jede Bedeutung wiederholt und als eignes Beispiel gespeichert. Zwar kann ein Absatz nie mit der selben Bedeutung in Trainings und Testdaten erscheinen, dennoch ist es möglich, dass der selbe Absatz mit unterschiedlicher Bedeutung vorkommt. Eine echte Trennung ist dadurch nicht gegeben.

Beim Vergleich der Genauigkeit auf *dev*- und *test*-Daten ist auffällig, dass bei jedem mit unterschiedlichen Parametern durchgeführtem Test die Genauigkeit nie mehr als 0,01% zwischen beiden Korpora abweicht. Dieser Hinweis auf eine mögliche Abhängigkeit der Daten untereinander macht eine Aussage über die Generalisierbarkeit des Modells nur eingeschränkt möglich. Es kann allerdings festgestellt werden, wie gut das Modell die Trainingsdaten abbildet. (Tabelle 6.3)

Eine Änderung des Aufteilungsalgorithmus aus Abschnitt 4.4 ist nicht trivial, da für jede Bedeutung sichergestellt werden muss, dass mindestens zehn Beispiele in den Trainings- und jeweils zwei Beispiele in den Testdaten vorkommen. Bedeutungen, die diese Bedingung nicht erfüllen können, müssen ausgelassen werden.

Tabelle 6.3.: Genauigkeit nach 8 Epochen mit optimierten Parametern

Datenset	Micro	Macro
train	91,80%	83,82%
dev	89,54%	77,95%
test	89,54%	77,94%

Da sämtliche Ergebnisse in dieser Arbeit auf voneinander abhängigen Daten basieren, wurde das Problem einer optimalen Aufteilung nicht weiter betrachtet und eine approximative Lösung für die bestehenden Daten gefunden, die es ermöglicht, die Auswirkung abzuschätzen. Algorithmus 2 teilt die vorhandenen Beispiele neu auf, sodass der selbe Absatz nie in zwei unterschiedlichen Datensets vorkommt. Anders als bei einer optimalen Auswahl von Bedeutungen, können Anforderungen an die Mindestanzahl von Beispielen nicht immer eingehalten werden, weshalb für 13 Bedeutungen weniger als zwei Testbeispiele und für 271 Bedeutungen weniger als zehn, aber min. sieben Trainingsbeispiele gefunden werden konnten. Der Anteil der Trainingsdaten sank auf ca. 60%.

Eingabe : B: Menge aller Beispiele
Ausgabe : Menge $train$, dev und $test$

```
1  train = ∅; dev = ∅; test = ∅;
2  S = Bedeutungen in B, aufsteigend sortierte nach Anzahl der Beispiele;
3  für Bedeutung s ∈ S tue
4      B_s = Alle Beispiele für Bedeutung s;
5      für Beispiel b ∈ shuffle(B_s) tue
6          wenn train ∪ dev ∪ test enthält Beispiel mit selben Absatz wie b dann
7              Beispiel b wird in selbes Datenset eingefügt;
8          sonst
9              wenn Zielanteil von B_s in dev oder test noch nicht erreicht dann
10                 Beispiel b in dev oder test (wenn Ziel für dev erreicht) einfügen;
11             sonst
12                 Beispiel b in train einfügen;
13             Ende
14         Ende
15     Ende
16 Ende
17 return train, dev und test;
```

Algorithmus 2 : Algorithmus zum Aufteilen von Beispielen auf Trainings- und Testsets ohne Überschneidungen von Absätzen.

Abbildung 6.4 vergleicht die alten Testergebnisse *dev & test* mit den neuen Ergebnissen auf die unabhängigen Testdaten *dev (neu)* und *test (neu)*. Die Micro-Genauigkeit ist geringfügig niedriger und variiert stärker. Die Macro-Genauigkeit steigt überraschenderweise um etwa

1%. Die Veränderungen sind nicht zwangsläufig auf die unabhängigen Beispiele zurückzuführen, da die Verteilung der Anzahl der Beispiele sich geändert hat und in beiden Testsets nicht mehr symmetrisch ist.

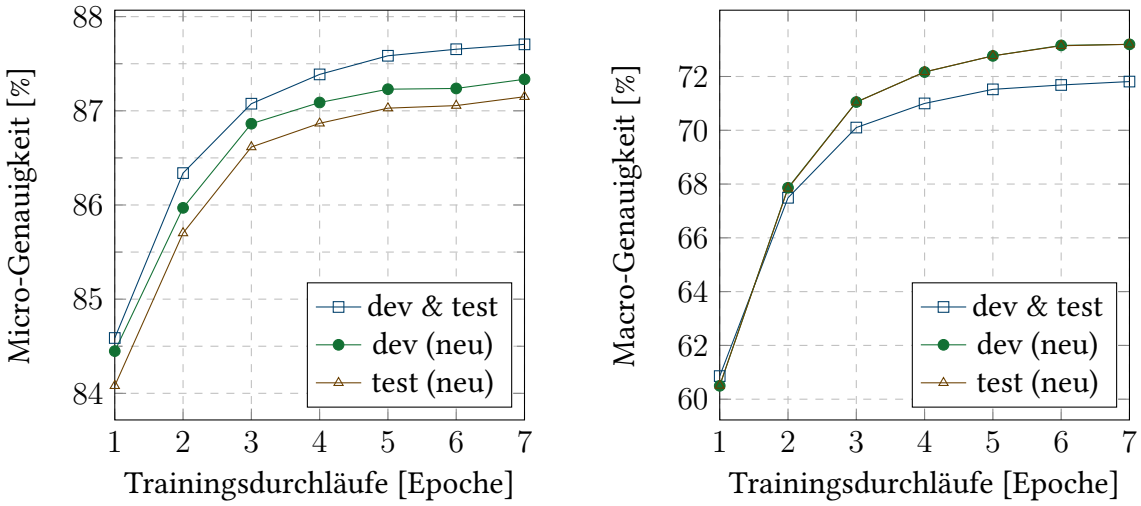

Abbildung 6.4.: Vergleich zwischen normalen (*dev & test*) und streng getrennten Testdaten.

Insgesamt wird deutlich, dass voneinander abhängige Daten nur einen nicht eindeutig positiven Einfluss auf Testergebnisse haben und eine schärfere Trennung nicht zwingend erforderlich ist.

6.8. Wenig Daten

Wie bereits mehrfach erwähnt, besitzen viele Bedeutungen nur wenige Beispiele. Abbildung 6.5 stellt die Anzahl der Bedeutungen und Beispiele in den Trainingsdaten pro Genauigkeit in den Testdaten grafisch dar.

Sofort auffällig ist, dass Bedeutungen mit sehr vielen Beispielen immer eine höhere Genauigkeit besitzen. Die max. Anzahl an Beispiele pro Bedeutung (Abbildung 6.5, unten) steigt nahezu exponentiell zur Genauigkeit. Bei der durchschnittlichen Anzahl an Beispielen (Abbildung 6.5, Mitte) ist dieser Trend nicht mehr einfach erkennbar. Hier besitzen Genauigkeiten, die mit wenigen Testdaten erreicht werden können (0, $^1/_2$, $^1/_3$, $^2/_3$, ...), unüberraschend durchschnittlich weniger Beispiele als die Genauigkeiten, die nur mit vielen Beispielen entstehen können (z.B. 99%, also min. 100 Trainingsbeispiele).

Insbesondere 100% und 0% Genauigkeit besitzen eine auffällig hohe Anzahl an Bedeutungen und eine niedrige durchschnittliche Anzahl an Beispielen. Der Abfall der Anzahl der Bedeutungen bei 100% im Vergleich zu 99% ist durch Begriffsgruppen mit nur einer Bedeutung zu erklären, die selbst mit wenigen Beispielen immer richtig vorhergesagt werden. Die hohe Anzahl an Bedeutungen mit wenigen Beispielen und 0% Genauigkeit macht deutlich, dass die Mindestanzahl an Beispielen für jede Bedeutung bei großen Datenmengen angehoben werden sollte, um bessere Ergebnisse zu erzielen.

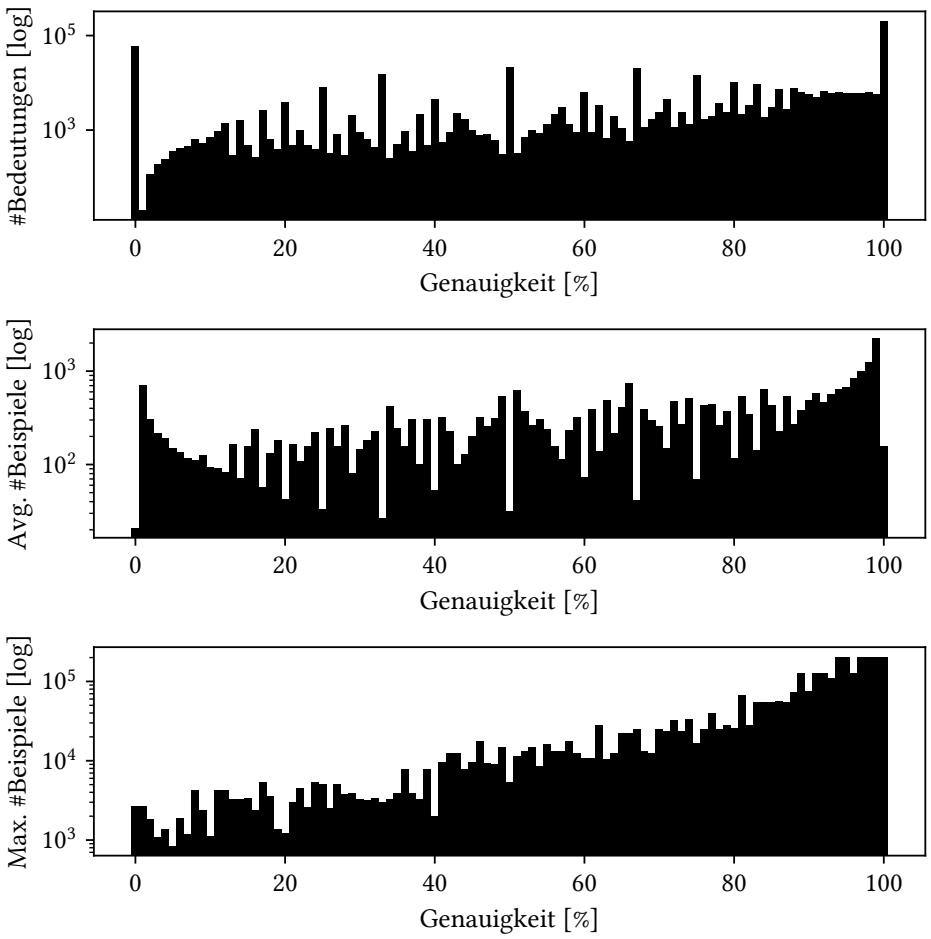

Abbildung 6.5.: Oben: Histogramm der Genauigkeit (gerundete Prozent) pro Bedeutung für bestes Modell; Mitte: Durchschnittliche Anzahl an Beispielen in *train* für Bedeutungen; Unten: Max. Anzahl an Beispielen; Alle Y-Achsen sind logarithmisch skaliert.

6.9. Zusammenfassung

Die Ergebnisse bestätigen, dass der Ansatz von fastSense auch für große englischsprachige Datenmengen geeignet ist und verbessert ihn durch angepasste Parameter und Einführung eines weiteren Hidden Layers. Bei den aufgezeigten Problemen und Vorschlägen besteht weiteres Potenzial zur Verbesserung. Des weiteren ist zusätzliche Optimierung der Performance erstrebenswert, um mehr Parameter vergleichen zu können.

7. Technische Umsetzung

Zur Umsetzung des beschriebenen Ansatzes wurde das Python-Modul *ned* erstellt. Es erfordert Python 3.5 und lässt sich über *pip*[1] einfach installiert. Es wurde unter Linux und macOS getestet und kann über die Kommandozeile oder direkt aus Python heraus verwendet werden.

7.1. Überblick

Das Modul besteht aus mehreren Kommandozeilen-Skripts, mit denen Trainings- und Testkorpora aus Wikipedia-Dumps erstellt werden können und sich Modelle trainieren und testen lassen. Zusätzlich können in beliebigen Python-Programmen die Klassen zur Disambiguation importiert werden. Zur Implementierung des NN wurde TensorFlow verwendet. Folgende Kommandozeilen Skripte sind im Modul enthalten:

- **ned-wiki-prepare**: Extrahiert Absätze und Links aus XML-Dumps der englischen Wikipedia, schreibt diese in eine Zwischendatei und ermittelt Begriffsgruppen und Bedeutungen nach dem in Abschnitt 4 beschrieben Verfahren. Die Informationen werden in einer Datenbank gespeichert. Zusätzlich wird gespeichert, welche Absätze Beispiele für welche Bedeutungen sind und in welchem Datenset sie sich befinden sollen.
- **ned-wiki-export**: Liest die mit *ned-wiki-prepare* erzeugten Daten und exportiert Beispiele in einem für das NN geeignete Format. Auf die Token der Beispiele werden wahlweise die in Abschnitt 3.2.2 erklärten Vorverarbeitungsschritte angewendet.
- **ned-train**: Trainiert, testet und speichert Modelle.
- **ned**: Interaktives Tool zum Experimentieren mit Modellen. Es kann Bedeutungen für eingegebene Absätze ermitteln.

Für Python-Programme stehen folgende Klassen zur Verfügung:

- **Disambiguator**: Klasse zum Finden und Disambiguieren von mehrdeutigen Begriffen in beliebigen Texten.
- **CoreNlpBridge**: Stellt Verbindung zwischen Standford CoreNLP und Disambiguator her. CoreNLP wird benötigt um Absätze zu tokenisieren.

[1] `https://pip.pypa.io/en/stable/`

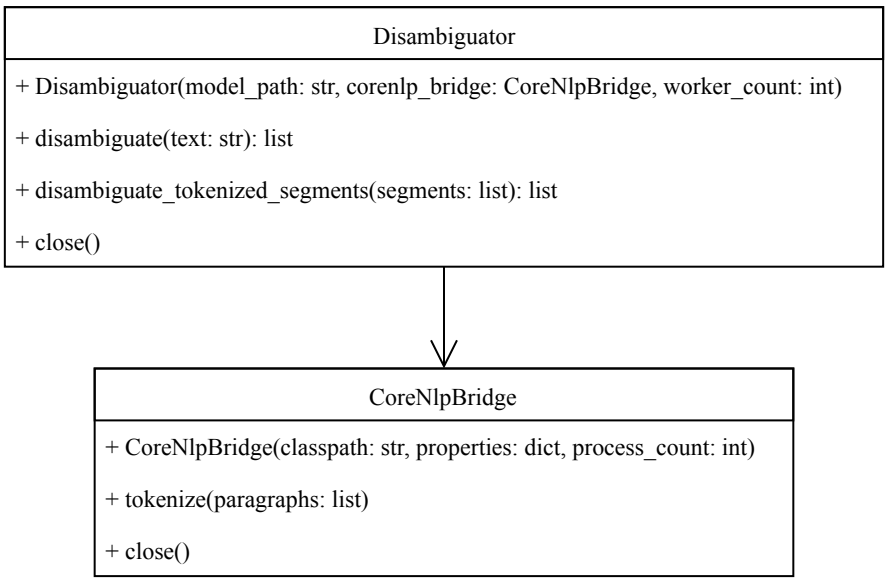

Abbildung 7.1.: Übersicht über öffentliche Klassen des *ned*-Moduls

7.2. Verwendete Software

Die Implementation des Moduls baut auf verschiedenen Softwarekomponenten auf, die in diesem Abschnitt vorgestellt werden.

7.2.1. TensorFlow

TensorFlow ist ein Framework, das verschiedene Machine Learning Algorithmen implementiert, die in C++ oder Python verwendet werden können. Das Open-Source Projekt wurde 2015 von Google veröffentlicht. (vgl. Martín Abadi et al. 2015)

Mit dem Framework wurde das in Abschnitt 3.2 beschriebene neuronale Netz implementiert, trainiert und getestet. Die vorhandene Implementation für die Messung der durchschnittlichen Genauigkeit pro Ausgabeklasse wurde neu implementiert, um eine größere Anzahl an Klassen zu unterstützen. Die Änderungen wurden dem Projekt als Pull-Request[2] zur Verfügung gestellt und in die offiziellen GitHub-Repository übernommen.

7.2.2. mwparserfromhell

mwparserfromhell (Kurtovic, Winegar, Riamse et al. o.D.) ist ein Python-Modul, das Wikitext parst und in einen abstrakten Syntaxbaum umwandelt. Wikitext ist die Auszeichnungssprache, in der sämtliche Artikel der Wikipedia vorliegen. Zum Formatieren von Texten verwendet es eine Mischung aus HTML und eigenen, erweiterbaren Anweisungen, die später zu HTML umgewandelt werden.

[2] `https://github.com/tensorflow/tensorflow/pull/15946`

7.2.3. Stanford CoreNLP und Pyjnius

Stanford CoreNLP ist ein Softwarepaket, das natürlichsprachige Texte analysieren kann. Es bieten eine Reihe an Annotatoren, die unter anderem Sätze, Token, Lemma und PoS-Tags finden können. Das Paket kann über einen Webserver oder direkt in Java verwendet werden. (vgl. Manning et al. 2014, S. 55-60)

Erste Entwicklungsversionen des *ned*-Moduls verwendeten die API des Webservers, um Absätze zu tokenisieren. Für jeden Absatz musst eine eigene Anfrage gesendet und die Antwort empfangen und geparst werden. Neben diesem unnötigen Overhead legt das Softwarepaket ein Limit von 10.000 Zeichen pro Anfrage für den Server fest. Wird das Paket direkt in Java verwendet, existiert diese Beschränkung nicht.

Um direkt aus dem Python-Modul mit Java zu interagieren, wurde das Paket *pyjnius* (Virbel, Pettier, Margulies et al. o.D.) verwendet. Durch diese Änderung können alle 59 Millionen Absätze in etwa einem Drittel der zuvor benötigten Zeit verarbeitet werden.

Javas interne Repräsentation von Strings, die eine modifizierte Variante von UTF-8 verwendet, führte zu einem Problem, da *pyjnius* fälschlicherweise diese interne Repräsentation in Python als Bytes in standardisiertem UTF-8 Format behandelt. Obwohl Texte in der modifizierte Kodierung häufig gültigem UTF-8 Text entsprechen und dekodiert werden können, sind String mit bestimmten Sonderzeichen nicht lesbar. Tritt der Fehler auf, entnimmt das *ned*-Modul die vorher nicht zugänglichen Bytes aus der Exception und wandelt sie mit dem Modul *ftfy* (Speer o.D.) selber in einen Python-String um.

Dass das neuronale Netz und CoreNLP auf dem gleichen Server laufen, muss nicht immer wünschenswert sein. Unterstützung für CoreNLP über die Server API ist nicht im Modul enthalten, kann allerdings leicht nachgerüstet werden, indem eine alternative Variante der Klasse *CoreNlpBridge* selber erstellt wird.

7.3. Trainingsdaten erstellen

Die Skripte *ned-wiki-prepare* und *ned-wiki-export* können Trainingsdaten erstellen. Das Vorgehen wurde bereits in Abschnitt 4 erklärt. Dieser Abschnitt geht näher auf einige Implementierungsdetails ein.

7.3.1. ned-wiki-prepare und ned-wiki-export

Trainingsdaten enthalten Beispiele, dessen Token durch die in Abschnitt 3.2.2 angegebenen Parameter beeinflusst werden. Um die Trainingsdauer über mehrere Epochen zu reduzieren, werden Änderungen angewendet, bevor die Trainingsdaten geschrieben werden. Änderungen der Parameter erfordern neue Trainingsdaten, weshalb dieser mehrfach wiederholbare Schritt von einem eigenen Skript übernommen wird.

Beispiele müssen einmalig von *ned-wiki-prepare* gefunden und auf Trainings- und Testsets verteilt werden. Das Skript tokenisiert gleichzeitig alle möglicherweise verwendeten Absätze. Anschließend kann *ned-wiki-export* beliebig oft Beispiele mit unterschiedlichen Parametern in das Trainingsformat umwandeln.

7.3.2. Parsen von Wikitext

Um den Wikitext sämtlicher Artikel aus dem XML-Dump der Wikipedia in ein brauchbares Format umzuwandeln, wurden eine Kombination aus Vorverarbeitungsschritten, dem Python-Modul *mwparserfromhell* und einem Auswertungsschritt des resultierenden abstrakten Syntaxbaums gewählt. Der Algorithmus 3 beschreibt den Vorgang grob. Die Vorverarbeitungsschritte parsen Teile des Wixitextes selber und passen diesen an, um Fehler des eigentlichen Parsers zu reduzieren.

Input : Wikitext
Output : Menge aus Abschnitten bestehend aus: Index, Überschrift, Menge aller Fragmentbezeichner (siehe Abschnitt 7.3.5), Index des Übergeordneten Abschnittes, Paragraphen bestehend sowohl aus Text als auch der Menge aller im Text vorkommenden Links.

1 Finde Templates die nicht in Tabellen oder Parametern anderer Templates verschachtelt wurden;
2 Übergebe bestimmte Templates an vordefinierte Funktionen und ersetze sie mit dem Rückgabewert;
3 Entferne alle übrig bleibenden Templates;
4 Finde und entferne alle Tabellen;
5 Finde Anfangs- und End-Tags von Listen und Listeneinträgen und ersetze sie durch zwei Zeilenumbrüche;
6 Finde und entferne alle Vorkommnisse von ″ und ‴ (Wikitext Styletags);
7 Teile Text in Abschnitte auf, indem jede Zeile mit einer Überschrift als Start eines neuen Abschnittes angesehen wird;
8 **für *jeden*** *Abschnitt* **tue**
9 | Parse Abschnitt mit *mwparserfromhell*;
10 | Ermittele Informationen über Abschnitt aus dem geparsten Text;
11 | Erzeuge Liste mit Absätzen und zugehörigen Links aus dem geparsten Text;
12 | Füge Informationen und Absätze zu den Ergebnissen hinzu;
13 **Ende**
14 **return** *Ergebnisse*;

Algorithmus 3 : Algorithmus zur Parsen von Wikitext

7.3.3. Probleme mit Templates in Wikitext

Wikitext kann durch Templates und Erweiterungen sehr komplex werden. Templates ermöglichen es häufig verwendete Texte und Elemente einmal zu definieren um sie anschließen in unterschiedliche Artikel einzubinden. Dabei können Parameter übergeben werden, die entweder eingesetzt oder von Erweiterungen ausgewertet werden, um das Template weiter zu beeinflussen. Über Erweiterungen lassen sich neue Tags und Funktionen nachrüsten, wie z.B. der Tag *<score>* für die Darstellung von Noten oder die Funktion *if*.

Aufgrund dieser Möglichkeiten ist es nicht einfach, alle Artikel richtig zu parsen, insbesondere da sich unbalancierte öffnende oder schließende Tags in Templates befinden können, die Templateersetzung zwingend erforderlich machen.

Nicht-Ersetzen von Templates bringt einen weiteren Nachteil mit sich: Templates wie *Asof*, das datumsabhängige Aussagen markiert und anstelle der Formulierung „As of *<DATE>*“ verwendet werden kann, und ähnliche Templates werden direkt in Texten eingebunden. Auslassen dieser Templates würde zu Textlücken führen. Auch wichtige Informationen außerhalb von Texten können verloren gehen, wie z.B. Anker für Links, die über das Template *Anchor* gesetzt werden.

Der originale Wikitext-Parser aus der MediaWiki Software, die von Wikipedia verwendet wird, kann mit den Problemen durch Templates und Erweiterungen problemlos umgehen. Da sich der Parser nicht direkt in Python verwenden lässt, der resultierende HTML-Code erneut geparst werden müsste und eine lokale Wikipedia Instanz erforderlich wäre, wurde er nicht verwendet.

Stattdessen wurden Python-Funktionen zum Umwandeln ausgewählte Templates geschrieben, um das Problem etwas zu reduzieren. Die meisten der 445 Templates werden durch einfache Tabellenstart oder -end Tags ersetzt, um Seiten fehlerfrei auswerten zu können. Die zuvor erwähnten Templates *Anchor* und *Asof* werden ebenfalls ersetzt.

Alle Templates, die nicht ersetzt werden, werden entfernt. Das hat den Vorteil, dass häufige Textbausteine, wie z.B. Hinweise auf Disambiguations-Seiten oder Info-Boxen, automatisch ignoriert werden und der Anteil der relevanten Absätze steigt.

7.3.4. Linktitel

Linktitel werden für die Auswahl von Bedeutungen verwendet. Der Titel kann von dem Titel des verlinkten Artikels abweichen. Verfasser haben dazu zwei Möglichkeiten. Sie können einen alternativen Titel eingeben oder einen Suffix an den Titel des Zielartikels anhängen. Suffix Anhängen erfordert keinen speziellen Wikicode, sondern wird über sprachabhängige Regeln automatisch durchgeführt. Dieser Vorgang wird als „Blending“ (*Wikipedia Help: Wikitext - Blend link* o.D.) bezeichnet und nicht von *mwparserfromhell* unterstützt. Wird z.B. der Artikel „Apple“ verlinkt und folgt dem Link ein „s“ ohne Leerzeichen, wird der Link als „Apples“ angezeigt, obwohl im Wikicode nur „Apple“ verlinkt wird. Beim Auswerten des abstrakten Syntaxbaums muss dies erkannt und korrigiert werden.

7.3.5. Weiterleitungen, Abschnitten und eindeutige Ziele

Weiterleitungen stellen ein Problem beim Finden von Bedeutungen dar, da Links zum selben Ziel auf unterschiedliche URLs verweisen können. Um Beispiele zu finden, die auf synonyme Bedeutungen verlinken, müssen alle Weiterleitungen aufgelöst werden, bevor das Linkziel untersucht wird. Dadurch werden z.B. alle Links zu dem Artikel „Alien life“ in Links zu dem Artikel „Extraterrestrial life“ umgewandelt, da es sich um eine Weiterleitung handelt. Der Schritt muss mehrfach wiederholt werden, da es Weiterleitungen zu Weiterleitungen geben kann.

Für Links und Weiterleitungen zu Abschnitten werden Fragmentbezeichner verwendet. Ein Fragmentbezeichner ist in einer URL der Teil hinter der Raute (#) und wird in der Wikipedia automatisch aus der Überschrift eines Abschnitts generiert. Anders als bei Artikeln, die über ihren normalisierten Titel eindeutig identifizierbar sind, können Abschnitte über unterschiedliche Fragmentbezeichner, z.B. durch benutzerdefinierte Anker, identifiziert werden. Anker werden unter anderem gesetzt, um alte Links nach Umbenennung eines Abschnitts nicht ins Leere führen zu lassen. Um Bedeutungen, die zu einen Abschnitt mit mehreren Fragmentbezeichnern gehören, korrekt zuzuordnen, werden jedem Abschnitt in einem Artikel eine Nummer zur Identifikation zugewiesen und alle zughörigen Fragmentbezeichner in einer Datenbank gespeichert.

Alle Links werden anschließend in Artikel ID und ggf. Abschnittsnummer umgewandelt und sind dadurch für jedes Ziel einzigartig. Existiert ein verlinkter Artikel oder Abschnitt nicht, wird der Linktitel als normaler Text behandelt. Wird auf einen Abschnitt mit einem Fragmentbezeichner verlinkt, der verbotenerweise für mehrere Abschnitte in Verwendung ist, wird der erste der in Frage kommenden Abschnitte gewählt.

7.4. Speicherformat für Trainingsdaten und trainierte Modelle

Um Trainings- und Testbeispiele schnell einlesen zu können, werden sie als TFRecords - TensorFlows Standardformat für Beispiele - serialisiert und komprimiert gespeichert. Die Daten werden auf mehrere Dateien mit jeweils bis zu 3 Millionen Beispielen aufgeteilt. Dadurch können Beispiele nicht nur innerhalb eines Buffers gemischt werden, sondern zusätzlich auch die Reihenfolge, in der die Dateien gelesen werden.

Wird das Modell eingesetzt, um einen unbekannten Text zu disambiguieren, muss dieser zuvor tokenisiert werden. Damit die Vorverarbeitungsschritte der Token (siehe Abschnitt 3.2.2) zu dem trainierten Modell passen, werden diese Informationen in einer JSON-Datei gespeichert. Informationen zu Begriffsgruppen und derer in Frage kommenden Bedeutungen sowie die zugehörigen Werte für Ein- und Ausgabeklassen werden in einer SQLite3 Datenbank hinterlegt.

Trainierte Modelle werden als *SavedModel* - einem von TensorFlow verwendeten Format zum serialisieren von Modellen - zusammen mit der JSON-Datei und Datenbank gespeichert.

7.5. Modelle trainieren

Modelle können über *ned-train* trainiert werden. Modell-Parameter werden in einer JSON-Datei übergeben.

Das NN wurde als Unterklasse von *tf.estimator.Estimator* implementiert und profitiert dadurch von vorhanden Funktionen zum Laden, Speichern, Trainieren und Testen. Trainingsfortschritt und Testergebnisse können über TensorBoard[3] - einer Komponente von TensorFlow - visualisiert werden, wie in Abbildung 7.2 dargestellt.

Mit TensorFlows Dataset API wurde eine Pipeline gebaut, die mehrer Dateien mit Trainings- oder Testdaten parallel liest und parst. Beim Training werden Beispiele zufällig gemischt.

[3] `https://github.com/tensorflow/tensorboard`

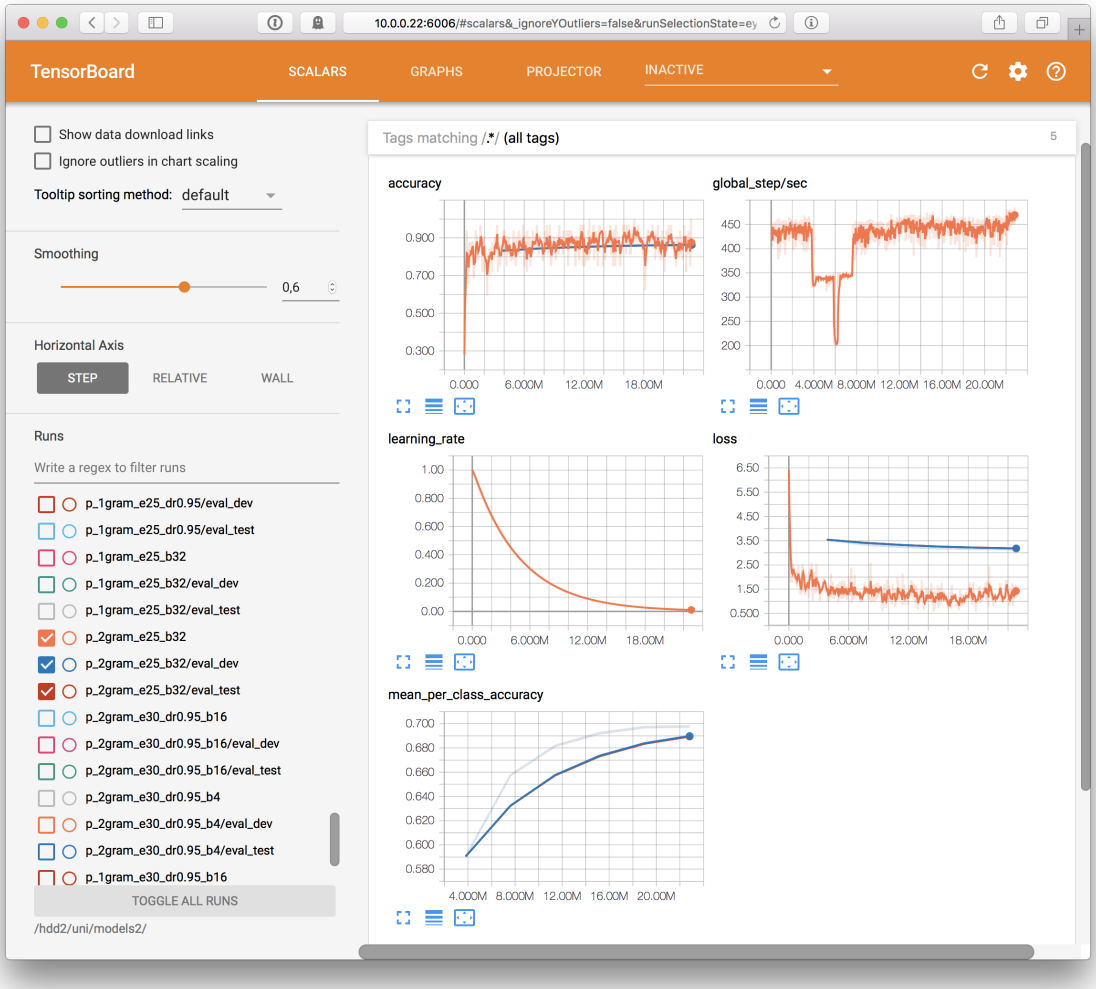

Abbildung 7.2.: Trainingsfortschritt in TensorBoard. Screenshot vom 12.03.2018

7.6. Modelle verwenden

Trainierte Modelle, die als *SavedModel* gespeichert werden, können sowohl in Python, als auch über das im Modul enthaltene *ned*-Skript verwendet werden. Ebenfalls kann das trainierte NN aufgrund das Standardformates einfach in anderen von TensorFlow unterstützte Sprachen eingebunden werden.

7.6.1. Python

Der Klasse *Disambiguator* muss der Pfad zu dem Ordner des *SavedModels* übergeben werden. Zusätzlich muss eine Instanz der CoreNlpBridge erstellt werden, die die Verbindung zwischen CoreNLP und Python herstellt. Der Disambiguator erstellt eine TensorFlow Session und lädt das trainierte Modell. Über die Methode *disambiguate()* können mehrdeutige

Begriffe gefunden und disambiguiert werden. Zum Beenden der Session muss *close()* aufgerufen werden oder das Objekt in Zusammenhang mit einem *with*-Block verwendet werden.

Der Aufruf von *disambiguate()* gibt eine Liste mit Tuples zurück, die die Start- und Endposition des mehrdeutigen Begriffs im Text sowie die URL zum Wikipedia Artikel der erkannten Bedeutung enthalten.

Begriffe können überlappen. Anwender müssen nach Anwendungsfall entscheiden, welche Begriffe wirklich benötigt werden. Ein Beispiel für eine Überlappung ist „big river song“. Die Phrase kann in vier unterschiedliche mehrdeutige Begriffe unterteilt werden: „big river“, „river song“, „river“ und „song“.

7.6.2. ned-Skript

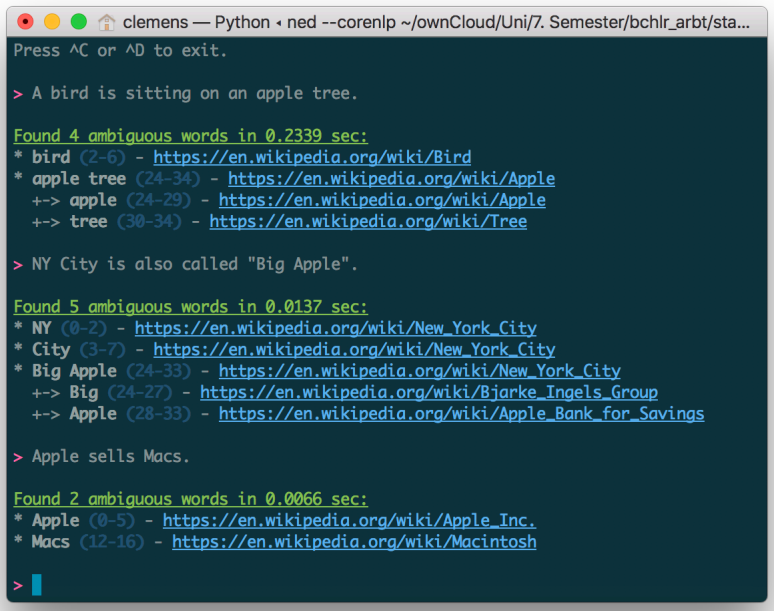

Abbildung 7.3.: Screenshot des *ned*-Skripts vom 12.03.2018

Das *ned*-Skript bietet die selbe Funktionalität wie die *Disambiguator*-Klasse, allerdings über ein Kommandozeileninterface. Abbildung 7.3 zeigt das Skript in Aktion. Es ist hauptsächlich zum Experimentieren mit Modellen gedacht.

7.6.3. Finden von mehrdeutigen Begriffen

Suchen und Finden von mehrdeutigen Begriffen ist ein eigenständiges Problem. Wörter in verschiedenen Formen müssen effizient in langen Texten auffindbar sein, wobei nicht jedes gefundene Wort wirklich ein Treffer sein muss.

Unter anderem gehört der Buchstabe „A“ zu den disambiguierbaren Begriffen und ist im Englischen ohne Betrachtung vom Kontext nicht von dem unbestimmten Artikel „a“ unterscheidbar. Um Fehler für diesen und ähnliche Begriffe zu vermeiden, müssen alle im Text gefundenen mehrdeutigen Begriffe Nomen sein.

Algorithmus 4 beschreibt den Suchvorgang, der jeweils einzeln für jeden Absatz durchgeführt wird. Der Algorithmus kann leicht parallelisiert werden, indem mehrere Absätze gleichzeitig durchsucht werden oder einAbsatz in gleich große Unterabschnitte unterteilt wird.

7.7. Performance

Um moderne Hardware effizient auszunutzen, können Absätze in langen Texten parallel disambiguiert werden. Da die CoreNLPBridge als eigenständige Komponente entworfen wurde, ist die Disambiguations-Pipeline (Abbildung 7.4) nicht optimal parallelisiert. Allerdings kann dadurch der Tokenizer leicht ausgetauscht werden.

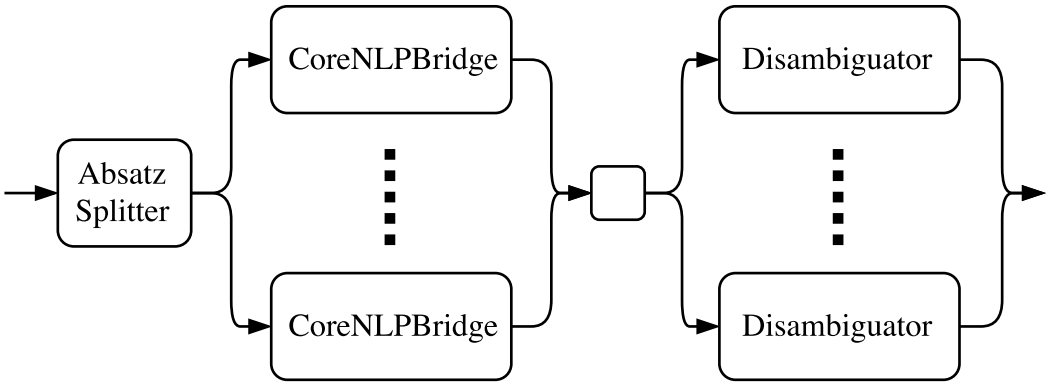

Abbildung 7.4.: Disambiguations-Pipeline

Die gesamte englische Wikipedia ohne Tabellen und Templates (siehe Abschnitt 4.3) kann in etwa 21,4 Stunden disambiguiert werden, wovon 16,8 Stunden alleine zum Tokenisieren benötigt werden. Das entspricht 45.824 Absätzen pro Minute oder 211.288 Absätze pro Minute auf bereits tokenisierte Daten. Der Test wurde auf einer Intel i7-6700 CPU (4 Kerne, 8 Threads) mit 8 parallelen Instanzen durchgeführt. Im Durchschnitt wurden 15GB Arbeitsspeicher belegt, wobei diese Zahl durch gemeinsam genutzten Speicher stark reduziert werden könnte. In Python ist dies nur mit Umwegen möglich. Mit 4 parallelen Instanzen reduziert sich der Speicherbedarf auf durchschnittlich 9GB und die Geschwindigkeit sinkt auf 33.571 Absätzen pro Minute.

Input :

- Eingabetext
- Sortiertes Array aller gesuchten mehrdeutigen Begriffe

Output : Menge aller mehrdeutigen Begriffe mit Positionsangabe

```
1  T = Tokenisierter Eingabetext;
2  m_i = Gesuchter mehrdeutiger Begriff an Array-Position i;
3  m_{-1} = Leerer String;
4  foreach Token t ∈ T do
5      Verwende Binäre Suche um Position i zu finden, sodass m_{i-1} < t ≤ m_i gilt;
6      while m_i beginnt mit t do
7          Markiere Begriff m_i;
8          i = i + 1;
9      end
10     j = 0;
11     while Markierte Begriffe existieren do
12         R = Rekonstruiere Text aus t und den nächsten j Token in T;
13         foreach Markierter Begriff x do
14             if R = x then
15                 if ∃ Token r ∈ R : r ist Nomen then
16                     Füge x und Position von R zu Ergebnissen hinzu;
17                 end
18                 Lösche Markierung für x;
19             else if Länge von x < Länge von R then
20                 Lösche Markierung für x;
21             end
22         end
23         j = j + 1;
24     end
25 end
26 return Ergebnisse;
```

Algorithmus 4 : Algorithmus zum Finden von mehrdeutigen Begriffen in Texten

7.8. Laufzeitanalyse

Zur Bestimmung der durchschnittlichen Laufzeit des Disambiguators mit einem Text aus p Absätzen und insgesamt t Tokens sowie durchschnittlich n mehrdeutigen Begriffen pro Absatz müssen folgende Schritte untersucht werden:

- Text in Absätze aufteilen: Der Text kann linear durchlaufen werden, um doppelte Zeilenumbrüche und ggf. Leerzeichen zwischen Textabschnitten zu finden.
- Tokenisieren und PoS-Tagging: Abhängig von CoreNLP. Es sind keine Informationen bezüglich der Laufzeit vorhanden.
- Finden von mehrdeutigen Begriffen und derer möglichen Bedeutungen: Durchschnittlich $O(t \cdot log(m))$, da binäre Suche verwendet wird. m ist die Anzahl der mehrdeutigen Begriffe und ist bei Verwendung des selben Modells konstant. Daher gilt $O(t)$.
- Mögliche Bedeutungen deduplizieren: $O(p \cdot n \cdot log(n))$. Die durchschnittliche Anzahl an möglichen Bedeutungen in einem Absatz hängt von der Anzahl der mehrdeutigen Begriffe ab. Obwohl es keine Überschneidungen von Bedeutungen unterschiedlicher Begriffe gibt und jeder mehrfach vorkommende Begriff nur einmal pro Absatz untersucht wird, kann dieser Schritt nicht ohne weitere Änderungen übersprungen werden, da Begriffsgruppen mehrere Begriffe mit den selben Bedeutungen enthalten können und sie erst an dieser Stelle zusammengefasst werden.
- Hashing, Embedding-Lookup und kombinieren von Tokens eines Absatzes: $O(t)$
- Matrix-Multiplikation mit Hidden Layer: $O(p)$, da die Matrizen eine konstante Größe besitzen und für jeden Absatz einmal berechnet werden.
- Matrix-Multiplikation mit Ausgabeschicht: $O(p \cdot n)$, da die Größe der Ausgabeschicht variabel ist und von der Anzahl der möglichen Bedeutungen abhängt.

Insgesamt ergibt sich bis zu diesem Punkt eine durchschnittliche Laufzeit (ohne CoreNLP) von:

$$O(t + p \cdot n \cdot log(n)) \tag{7.1}$$

Um die wahrscheinlichste Bedeutung für jeden Begriff zu finden, müssen die Ergebnisse der Ausgabeschicht in $O(n \cdot log(n))$ sortiert und anschließend in $O(n^2 \cdot log(n))$ mit den möglichen Bedeutungen jedes Begriffs verglichen werden. Die zugehörige URL kann in $O(log(a))$ gefunden werden, wobei a die Anzahl aller Ausgabeklassen ist. Da die Anzahl innerhalb eines Modells konstant ist, kann die Laufzeit als konstant angesehen werden. Daraus ergibt sich insgesamt eine Laufzeit von:

$$O(t + p \cdot n^2 \cdot log(n)) \tag{7.2}$$

8. Zusammenfassung

Es wurde gezeigt, dass der Ansatz von fastSense auch auf der englischsprachigen Wikipedia gute Ergebnisse erzielt. Durch eine effiziente Implementierung des NN konnten Parameter für eine größere Datenmenge optimiert werden. Mit Hilfe eines zusätzlichen Hidden Layers, wurde das Ergebnis weiter verbessert und eine Micro-Genauigkeit von nahezu 90% erreicht. Auch bei der Wiederholung von Tests mit SemCor-Daten konnten größtenteils Verbesserungen erzielt werden.

Das entwickelte Python-Modul ermöglicht es, neue Modelle zu trainieren oder bestehende zu verwenden. Standardformate zum Speichern von Modellen und zugehörigen Informationen erleichtern es, die Modelle in anderen von TensorFlow unterstützten Sprachen neu zu implementieren und in bestehende Textverarbeitungs-Pipelines einzubauen.

8.1. Offene Probleme

Die Trainingsdauer des NN ist trotz akzeptabler Geschwindigkeit verbesserungsfähig: Mit nur geringem Geschwindigkeitsverlust lassen sich mehrere Netze problemlos parallel in separaten Prozessen trainieren, was auf ein Bottleneck hinweist und weitere Optimierung erfordert.

Auch der Speicherbedarf lässt sich weiter reduzieren: Um die Tokenisierung zu parallelisieren, werden mehrere CoreNLP-Instanzen gestartet, die unnötig viel Arbeitsspeicher belegen. Durch Verwendung von Java-Threads anstelle von Python-Prozessen, kann eine einzelne CoreNLP-Pipeline parallel genutzt werden. Eine Umstrukturierung der *Disambiguator*- und *CoreNLPBridge*-Klassen wird dadurch allerdings erforderlich.

8.2. Anregungen für weiterführende Arbeiten

Folgende Ideen könnten in weiterführenden Arbeiten aufgegriffen werden:

- Die möglichen Bedeutungen eines Begriffs sind teilweise unvollständig, da Bedeutungen mit zu wenigen Beispielen ignoriert werden. Es könnte sinnvoll sein, diese Beispiele zum Training einer bedeutungslosen Ausgabeklasse für jede Begriffsgruppe zu verwenden.

- Manche Begriffe, wie z.B. der englische Artikel „a“, werden selten in ihrem offensichtlichen Kontext verlinkt. Sie werden später zwar als mehrdeutiger Begriff erkannt, können aber nicht ihrer tatsächlichen Bedeutung zugeordnet werden. Dieses Fehlverhalten wird nicht in den Testdaten reflektiert. Es könnte untersucht werden, ob Negativ-Beispiele, also Beispiele, die einen nicht verlinkten, mehrdeutigen Begriff enthalten, zu einer Verbesserung führen.

- Bei der Suche nach Beispielen wird nur das Ziel eines Links betrachtet und nicht der Linktitel. Würden für jede Begriffsgruppe nur Beispiele ausgewählt, die einen der zugehörigen Begriffe enthalten, verkleinert sich die Datenmenge, möglicherweise wären die Beispiele aber relevanter.
- Bedeutungen einer bestimmten Wortart zuzuordnen, könnte es ermöglichen, unpassende Bedeutungen besser ausschließen. So könnte für den Begriff „fly“ zwischen dem Nomen „Fly“ (Tier) und dem Verb mit Bedeutung „Flight“ unterschieden werden. Eine Einschränkung der disambiguierbaren Begriffe auf Nomen wäre dadurch nicht mehr erforderlich.
- Begriffsgruppen könnten auf mehrere NN aufgeteilt werden, z.B. nach Kategorien getrennt, um problemspezifischere Embeddings zu trainieren.

A. Programmdokumentation

Installation

```
cd src/
pip3 install .
```

Die Umgebungsvariable *JAVA_HOME* muss gesetzt sein, um CoreNLP zu verwenden.

Erstellen von Trainings- und Testkorpora aus Wikipedia-Daten

```
ned-wiki-prepare \
    --dump ./enwiki-20170920-pages-articles.xml.bz2 \
    --page_table ./enwiki-20170920-page.sql.gz \
    --categorylinks_table ./enwiki-20171220-categorylinks.sql.gz \
    --corenlp /path/to/folder/containing/corenlp/ \
    --db does_not_exist_yet.sqlite3 \
    --intermediate_output ./temp/

ned-wiki-export \
    --db does_not_exist_yet.sqlite3 \
    --intermediate ./temp/ \
    --output /path/to/output/folder/ \
    -f "name_fuer_daten,2,0,0,0,0,0" "anderer_name,2,0,0,0,0,1"
```

Argumente für *ned-wiki-prepare*:

`--dump`	Pfad zum BZ2-Archiv mit Wikipedia-Artikeln
`--page_table`	Pfad zur Gzip-Datei mit SQL-Dump der *page*-Tabelle
`--categorylinks_table`	Pfad zur Gzip-Datei mit SQL-Dump der *categorylinks*-Tabelle
`--corenlp`	Pfad zum Ordner, der Java-Klassen von CoreNLP enthält.
`--db`	Pfad zu einer noch nicht existierenden SQLite3-Datenbank, vorzugsweise auf einem schnellen Speichermedium. Benötigt etwa 18GB!
`--intermediate_output`	Ordner, in dem Zwischendaten abgelegt werden. Benötigt etwa 26GB freien Speicher.

Argumente für *ned-wiki-export*:

`--db`	Identisch zu zuvor verwendetem Pfad.
`--intermediate`	Identisch zu zuvor verwendetem Pfad.
`--output`	Ordner für Trainings- und Testdaten.
`-f`	Liste aller gewünschten Variationen der Vorverarbeitungsschritte. Folgende Informationen müssen durch Kommata getrennt werden: • Beliebiger Name für Daten • N-Gram-Größe • Token in Kleinbuchstaben umwandeln (0 = Nein, 1 = Ja) • Satzzeichen entfernen (0 = Nein, 1 = Ja) • PoS-Tags hinzufügen (0 = Nein, 1 = Ja) • Lemmata anstelle von Token verwenden (0 = Token, 1 = Lemmata) • Sätze anstelle von Absätzen verwenden (0 = Absätze, 1 = Sätze)

Datenbank und Zwischenspeicher können nach Erstellung der Trainingsdaten gelöscht werden. Informationen zu Argumenten können auch durch Anhängen von *-h* an den jeweiligen Befehl angezeigt werden.

Trainieren und Testen von Modellen

```
ned-train \
    --data /path/to/output/folder/ \
    --models_dir /path/to/models/folder/ \
    --final_models_dir /path/to/final/models/folder/ \
    --jobs ./train_jobs.json
```

Argumente für *ned-train*:

`--data`	Identisch zu Ausgabeordner von *ned-wiki-export*.
`--models_dir`	Pfad zu Ordner, in dem Modelle während des Trainings gespeichert werden. Existiert bereits in Modelle mit gleichem Namen, wird es vor dem Training geladen.
`--final_models_dir`	Optional. Wenn angegeben, werden Modelle nach dem Training exportiert, sodass sie vom Disambiguator verwendet werden können. Für jedes exportierte Modelle wird ein eigener Unterordner erstellt. Der aktuelle Zeitstempel wird als Name verwendet. In dem Unterordner befindet sich die Datei *info.json*, die Informationen zu dem Modell, Parametern und Testergebnissen enthält.
`--jobs`	Pfad zu JSON-Datei mit Trainingsaufgaben.

train_jobs.json

```
[
    {
        "dataset_name": "name_fuer_daten",
        "model_name": "name_fuer_model",
        "params": {
            "hash_bucket_size": 10000000,
            "embedding_size": 25,
            "use_sqrtn_combiner": true,
            "clip_gradients": false,
            "learning_rate": 1.0,
            "decay_rate": 0.98,
            "decay_steps": 100000,
            "hidden_layer_sizes": [25],
            "dropout_keep_prob": 1.0
        },
        "target": {
            "metric_key": "loss",
            "flip_sign_of_metric": true,
            "end_if_slope_less_than": 0.0,
            "epochs_to_avg_over": 10,
            "test_after_epochs": 1
        }
        "batch_size": 32,
        "train_sets": ["train"],
        "test_sets": ["dev", "test"]
    }
]
```

Die JSON-Datei mit den Trainingsaufgaben muss ein Array enthalten, indem für jede Aufgabe ein eigener Dictionary enthalten ist. Folgende Schlüssel müssen in diesem enthalten sein:

`dataset_name`	Name der Daten. Entspricht dem Namen, der *ned-wiki-export* mit *-f* übergeben wurde.
`model_name`	Beliebiger Name für Modell. Existiert bereits ein Modell mit dem gleichen Namen, wird dieses weitertrainiert.
`params`	Dictionary mit Parametern
`target`	Dictionary mit Abbruch-Regel.
`epochs`	Kann verwendet werden, wenn *target* nicht existiert. Integer mit Anzahl der zu durchlaufenden Epochen, bevor Tests gestartet werden.
`batch_size`	Batch-Größe
`train_sets`	Name der Trainings-Daten, die durchlaufen werden sollen. Üblicherweise „train“.
`test_sets`	Name der Test-Daten, die durchlaufen werden sollen. Üblicherweise „dev“ und/oder „test“.

Im Dictionary für *target* müssen folgende Schlüssel enthalten sein:

`metric_key`	Zu betrachtender Wert. Mögliche Werte: *loss*, *accuracy* oder *mean_per_class_accuracy*
`flip_sign_of_metric`	*true*, um eine größer-als Bedingung (anstelle von kleiner-als) zu verwenden.
`end_if_slope_less_than`	Training wird beendet, wenn Steigung des betrachteten Wertes kleiner als der angegebene Wert ist.
`epochs_to_avg_over`	Glättet den Wert mit dem gleitenden Durchschnitt über die angegebene Zahl an vorherigen Werten.
`test_after_epochs`	Anzahl der Epochen, die durchlaufen werden sollen, bevor getestet wird. Üblicherweise 1.

Ist TensorBoard installiert, kann der Trainingsfortschritt beobachtet werde, indem der Pfad für *–models_dir* als *logdir* verwendet wird.

Ausprobieren von Modellen

```
ned \
    --corenlp /path/to/folder/containing/corenlp/ \
    --model /path/to/final/model/
```

Argumente für *ned*:

`--corenlp` Pfad zum Ordner, der Java-Klassen von CoreNLP enthält.

`--model` Pfad zu Ordner des gewünschten Modells. Das Modell muss exportiert worden sein, also ein Unterordner aus dem Ordner für *final_models_dir* sein. Der Ordner kann auch an einen anderen Ort kopiert werden.

Verwenden von Modellen in Python

Folgendes Beispiel zeigt die Verwendung der Disambiguator-Klasse:

```python
from ned import CoreNlpBridge, Disambiguator
import os.path

corenlp_path = "./corenlp/" # Pfad zu CoreNLP
model_path = "./mymodel/" # Pfad zu Modell
input_text = "..." # Text

classpath = os.path.join(corenlp_path, "*")

corenlp_bridge = CoreNlpBridge(classpath, properties=None, process_count=4)

with Disambiguator(model_path, corenlp_bridge, worker_count=1) as disambig:
    results = disambig.disambiguate(input_text)

    for start, end, url in results:
        ambiguous_word = input_text[start:end]
        print(ambiguous_word, "-", url)
```

Die Anzahl der parallelen Instanzen von *CoreNlpBridge* können über *process_count* festgelegt werden. Die Anzahl der parallelen Instanzen von *Disambiguator* können über *worker_count*

bestimmt werden. Bei sehr kurzen Texten macht ein Wert von 1 am meisten Sinn, da Parallelität nur auf Absatzebene erfolgt. Werden sehr viele kurze Texte (d.h. Anzahl der Absätze ist kleiner als *worker_count*) disambiguiert, kann es von Vorteil sein, die Texte aneinander zu hängen und in einem Durchgang zu verarbeiten.

Verwendung ohne CoreNLP

Es können auch bereits tokenisierte Texte disambiguiert werden:

```
from ned import CoreNlpBridge, Disambiguator
from ned.token import Token

model_path = "./mymodel/" # Pfad zu Modell

input_paragraphs = [
    [
        # Token für 1. Absatz
        Token(start=0, end=5, value="Hello", pos="UH", lemma=None,
            before="", after=" "),
        Token(start=6, end=11, value="World", pos="NNP", lemma=None,
            before=" ", after="")
    ],
    [
        ... # Token für 2. Absatz
    ],
    ...
]

with Disambiguator(model_path) as disambig:
    results = disambig.disambiguate_tokenized_segments(input_paragraphs)

    for start, end, url in results:
        print(start, "-", end, "=>", url)
```

Disambiguierbare Wörter oder Phrasen müssen min. ein Token mit *NN*, *NNS*, *NNP*, *NNPS* oder *FW* PoS-Tag enthalten um erkannt zu werden. *before* muss Leerzeichen (wenn vorhanden) zwischen dem aktuellen und vorangegangenen Token enthalten, damit Begriffe, die aus mehreren Token bestehen, erkannt werden können.

B. Eigene Trainingsdaten

Modelle können mit eigenen Daten trainiert und getestet werden, solange sie im richtigen Format vorliegen. Der Ordner für die Trainings- und Testdaten ist folgendermaßen aufgebaut:

1. **Begriffe und Bedeutungen**:

   Mehrdeutige Begriffe und deren Bedeutungen müssen in einer *sqlite3*-Datenbank mit dem Dateinamen *additional_data.sqlite3* auf oberster Ebene des Ordners der Trainingsdaten gespeichert werden. Das Schema der Datenbank wird in Abbildung B.1 beschrieben.

2. **Daten**:

   Auf Daten können verschiedene Vorverarbeitungsschritte angewendet werden, wie in Abschnitt 3.2.2 beschrieben. Für jede Variation wird ein eigener Unterordner mit beliebigem Namen angelegt, der folgende Informationen enthält:

   - **Token-Beschreibung**:

     Die Vorverarbeitungsschritte werden von einem *DataDescriptor* definiert und als JSON-Datei mit dem Namen *data_descriptor.json* gespeichert.

   - **Beispiele**:

     Beispiele werden in Trainings- und Testdaten aufgeteilt, die jeweils einen eigenen Unterordner besitzen (*train*, *dev* und *test*). Mit einem *ExampleWriter* können sie in das Zielformat umgewandelt werden.

Der Python-Code aus Abbildung B.2 zweigt beispielhaft, wie eigene Trainingsdaten erstellt werden können.

```sql
CREATE TABLE group_titles (
    id INTEGER, -- Lückenlose Nummerierung beginnend mit 0
    title TEXT -- Zu disambiguierendes Wort in Kleinbuchstaben
);

CREATE TABLE senses (
    id INTEGER PRIMARY KEY, -- Lückenlose Nummerierung beginnend mit 0
    url TEXT -- URL oder anderer Wert zum identifizieren einer Bedeutung
);

CREATE TABLE possible_senses (
    group_id INTEGER, -- id aus Tabelle group_titles
    sense_id INTEGER -- id aus Tabelle senses
);
```

Abbildung B.1.: SQL Schema für *additional_data.sqlite3*

```python
import os
from ned.data import DataDescriptor, ExampleWriter

output_path = ...  # Ordner der additional_data.sqlite3 enthält

data_path = os.path.join(output_path, "data name")
os.makedirs(data_path)

dd = DataDescriptor(
    n_gram_size=2,
    caseless=False,
    ignore_punctuation=False,
    add_pos_tags=False,
    uses_lemma=False,
    uses_sentences=False
)

dd_path = os.path.join(data_path, "data_descriptor.json")
dd.save(dd_path)

for name in ["train", "dev", "test"]:
    path = os.path.join(data_path, name)
    os.makedirs(path)

    example_writer = ExampleWriter(
        path=path,
        file_prefix=name,
        data_descriptor=dd
    )

    tokens = ["Tokens", "of", "example", "."]
    possible_senses = sorted([0, 1, 5])
    sense = 1

    example_writer.write(
        tokens=tokens,
        possible_senses=possible_senses,
        sense=sense
    )

    example_writer.close()
```

Abbildung B.2.: Beispielcode zum erstellen von Trainingsdaten

C. Weitere Ergebnisse

Folgende Tabellen enthalten weitere Ergebnisse. Alle Parameter, die nicht angegeben wurden, entsprechen den optimierten Parametern aus Tabelle 5.2.

Tabelle C.1.: Testergebnisse für *dev*. Absteigend sortiert nach Micro-Genauigkeit.

Features	NG	E	DKP	HL	Epoche	Micro	Macro
Token	2	25	1,00	[25]	1	86,1 %	68,2 %
Token (np)	2	25	1,00	[]	1	84,8 %	61,0 %
Token	2	25	1,00	[]	1	84,6 %	60,9 %
Token (np, lc)	2	25	1,00	[]	1	84,6 %	60,7 %
Token	3	25	1,00	[]	1	84,5 %	59,5 %
Token (lc)	2	25	1,00	[]	1	84,5 %	60,9 %
Token + PoS	2	25	1,00	[]	1	84,4 %	60,4 %
Lemma	2	25	1,00	[]	1	84,4 %	60,5 %
Token	3	25	1,00	[25]	1	84,2 %	60,9 %
Token (np)	2	25	1,00	[25]	1	83,7 %	60,7 %
Token	1	25	1,00	[]	1	83,5 %	61,7 %
Token + PoS	2	25	1,00	[25]	1	83,3 %	60,2 %
Token (lc)	2	25	1,00	[25]	1	83,3 %	60,4 %
Lemma	2	25	1,00	[25]	1	83,1 %	60,1 %
Token	2	25	1,00	[25]	1	83,1 %	59,6 %
Token	2	20	1,00	[20]	1	82,5 %	58,3 %
Token	1	25	1,00	[25]	1	80,8 %	58,3 %
Token	2	25	1,00	[15]	1	80,7 %	55,4 %
Token	2	25	0,66	[20]	1	80,2 %	53,3 %
Token	2	10	1,00	[20]	1	79,9 %	55,8 %
Token	1	25	0,66	[25]	1	78,0 %	52,0 %

lc = Kleinbuchstaben, **np** = Keine Satzzeichen, **PoS** = Part of Speech, **NG** = n-Gram, **E** = Embeddinggröße, **DKP** = Dropout Keep Probability, **HL** = Hidden Layers

Tabelle C.2.: Testergebnisse für *dev* mit fehlerhaften Daten. Absteigend sortiert nach Micro-Genauigkeit.

Features	NG	S	E	$\sqrt{n}$	Clip	LR	DR	Batch	Epoche	Micro	Macro
Token	2		30	✓		1	0,95	16	1	**84,0%**	55,9%
Token (np)	2		25	✓		1	0,98	32	1	83,4%	59,2%
Token	1		30	✓		1	0,95	16	1	83,3%	57,2%
Token	2		25	✓		1	0,98	32	1	83,2%	59,0%
Token (lc)	2		25	✓		1	0,98	32	1	83,1%	**59,3%**
Token	2		30	✓		1	0,98	64	1	82,4%	55,8%
Token	2		25	✓		1	0,98	64	1	82,1%	55,6%
Token	1		30	✓		1	0,98	64	1	82,0%	58,1%
Token	1		25	✓		1	0,98	32	1	81,9%	59,6%
Token	2		30	✓		1	0,95	4	1	81,6%	47,8%
Token	1		25	✓		1	0,95	64	1	81,4%	55,6%
Token	1		25	✓		1	0,98	64	1	81,2%	56,7%
Token	2		20	✓		1	0,98	64	1	81,0%	53,8%
Token	1		25	✓		1	0,99	64	1	80,8%	57,8%
Token	1		20	✓		1	0,98	64	1	80,4%	55,5%
Token	2		15	✓		1	0,98	64	1	79,7%	51,9%
Token	1		15	✓		1	0,98	64	1	79,0%	53,4%
Token	2	✓	25	✓		1	0,98	32	1	78,1%	56,0%
Token	2		10	✓		1	0,98	64	1	77,5%	48,9%
Token	2		10	✓	✓	1	0,98	64	1	77,4%	48,8%
Token	1		10	✓		1	0,98	32	1	77,0%	51,4%
Token	1		10	✓		1	0,98	64	1	76,6%	49,9%
Token + PoS	1		10	✓		1	0,98	64	1	76,5%	49,6%
Token	1		10	✓	✓	1	0,98	64	1	76,4%	49,7%
Token	1		10	✓		1	0,98	64	1	76,4%	49,6%
Lemma	1		10	✓		1	0,98	64	1	76,4%	49,9%
Lemma + PoS	1		10	✓		1	0,98	64	1	76,2%	49,1%
Token	1		10	✓		1	0,98	128	1	75,7%	47,3%
Token	1		10	✓		0,1	0,99	32	1	74,2%	42,0%
Token	2	✓	10	✓		1	0,98	32	1	73,2%	48,8%
Token	1		10		✓	1	0,98	64	1	73,0%	39,8%
Token	1	✓	10	✓		1	0,98	32	1	72,4%	49,7%
Lemma + PoS	1	✓	10	✓		1	0,98	32	1	72,4%	49,3%
Token + PoS	1	✓	10	✓		1	0,98	32	1	72,3%	49,1%
Lemma	1	✓	10	✓		1	0,98	32	1	72,1%	49,6%
Token	1		5	✓		1	0,98	64	1	71,6%	43,6%
Token	2		10		✓	1	0,98	64	1	71,3%	36,2%

lc = Kleinbuchstaben, **np** = Keine Satzzeichen, **PoS** = Part of Speech, **NG** = n-Gram, **S** = Sätze verwenden, **E** = Embeddinggröße, $\sqrt{\mathbf{n}}$ = Erklärt in Abschnitt 3.2.3, **Clip** = Gradient Clipping, **LR** = Lernrate, **DR** = Abnahmerate, **Batch** = Batchgröße

Abkürzungsverzeichnis

NED Named-Entity Disambiguation

WSD Word-Sense Disambiguation

NN Neuronales Netz

DNN Deep Neural Network

PoS Part-of-Speech

Literatur

Cortes, Corinna und Vladimir Vapnik (1995). „Support-vector networks“. In: *Machine learning* 20.3, S. 273–297.

He, Zhengyan et al. (2013). „Learning entity representation for entity disambiguation“. In: *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Bd. 2, S. 30–34.

Joulin, Armand, Edouard Grave, Piotr Bojanowski und Tomas Mikolov (2016). „Bag of Tricks for Efficient Text Classification“. In: *arXiv preprint arXiv:1607.01759*.

Kurtovic, Ben, David Winegar, Riamse et al. (o.D.). *mwparserfromhell: A Python parser for MediaWiki wikicode*. Zugriff: 28.2.2018. URL: https://github.com/earwig/mwparserfromhell.

Manning, Christopher D. et al. (2014). „The Stanford CoreNLP Natural Language Processing Toolkit“. In: *Association for Computational Linguistics (ACL) System Demonstrations*, S. 55–60. URL: http://www.aclweb.org/anthology/P/P14/P14-5010.

Martín Abadi et al. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. URL: https://www.tensorflow.org/.

Pascanu, Razvan, Tomas Mikolov und Yoshua Bengio (2012). „Understanding the exploding gradient problem“. In: *CoRR* abs/1211.5063. arXiv: 1211.5063. URL: http://arxiv.org/abs/1211.5063.

Pershina, Maria, Yifan He und Ralph Grishman (2015). „Personalized page rank for named entity disambiguation“. In: *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, S. 238–243.

Raiman, Jonathan und Olivier Raiman (2018). „DeepType: Multilingual Entity Linking by Neural Type System Evolution“. In: *arXiv preprint arXiv:1802.01021*.

Speer, Rob (o.D.). *ftfy: fixes text for you*. Zugriff: 28.2.2018. URL: https://github.com/LuminosoInsight/python-ftfy.

Srivastava, Nitish, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever und Ruslan Salakhutdinov (2014). „Dropout: A Simple Way to Prevent Neural Networks from Overfitting“. In: *Journal of Machine Learning Research* 15, S. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

Stanford CS class CS231n Course Notes (o.D.). Zugriff: 25.2.2018. URL: http://cs231n.github.io/neural-networks-1/.

Tensorflow API Dokumentation: tf.nn.embedding_lookup_sparse (o.D.). https://www.tensorflow.org/api_docs/python/tf/nn/embedding_lookup_sparse. Zugriff: 28.2.2018.

Uslu, Tolga, Alexander Mehler und Daniel Baumartz (2018). *fastSense: An Efficient Word Sense Disambiguation Classifier*.

Virbel, Mathieu, Gabriel Pettier, Benson Margulies et al. (o.D.). *pyjnius: Access Java classes from Python*. Zugriff: 28.2.2018. URL: https://github.com/kivy/pyjnius.

Wikipedia Help: Wikitext - Blend link (o.D.). Zugriff: 28.2.2018. URL: https://en.wikipedia.org/wiki/Help:Wikitext#Blend_link.

Wikipedia: Vereinfachte Darstellung eines NN (2006). Zugriff: 2.3.2018. URL: https://de.wikipedia.org/wiki/Datei:Neural_network.svg.

Zhong, Zhi und Hwee Tou Ng (2010). „It Makes Sense: A Wide-coverage Word Sense Disambiguation System for Free Text". In: *Proceedings of the ACL 2010 System Demonstrations*. ACLDemos '10. Uppsala, Sweden: Association for Computational Linguistics, S. 78–83. URL: http://dl.acm.org/citation.cfm?id=1858933.1858947.