Hochschule Niederrhein
University of Applied Sciences

Elektrotechnik
und Informatik
Faculty of Electrical Engineering
and Computer Science

Textklassifikation mit neuronalen Netzen und klassischen Modellen

Michael Kass

Technischer Bericht Nr. 2019-01

IMPRESSUM

Technische Berichte des Fachbereichs Elektrotechnik und Informatik,
Hochschule Niederrhein

ISSN 2199-031X

HERAUSGEBER

Christoph Dalitz und Steffen Goebbels
Fachbereich Elektrotechnik und Informatik

ANSCHRIFT

Hochschule Niederrhein
Reinarzstr. 49
47805 Krefeld

http://www.hsnr.de/fb03/technische-berichte/

M. Kass: „Textklassifikation mit neuronalen Netzen und klassischen Modellen.“
Technischer Bericht Nr. 2019-01, Hochschule Niederrhein, Fachbereich Elektrotechnik und Informatik, 2019

Textklassifikation mit neuronalen Netzen und klassischen Modellen

Michael Kass
Another Monday Service GmbH Deutschland
Brüsseler Straße 89 – 93, 50672 Köln
michael.kass@gmx.de

Zusammenfassung

Dieser technische Bericht soll einen Einblick in aktuelle Methoden des überwachten Lernens zur automatischen Klassifikation von Textdokumenten geben. Dazu werden Classifier auf einer Teilmenge eines Datensatzes von 199 768 Notizen zu Störmeldungen eines deutschen Telekommunikationsunternehmens trainiert. Der Fokus liegt dabei auf aktuellen Architekturen neuronaler Netze, insbesondere Convolutional Neural Networks (CNN) und Recurrent Neural Networks (RNN), die Gegenstand aktueller Forschung sind. Diese Architekturen wurden unter Zuhilfenahme der Bibliotheken scikit-learn[1] und TensorFlow[2] implementiert, um mit einer dafür angefertigten Testumgebung in Hinblick auf ihre Erkennungsrate, ihre Geschwindigkeit und ihre Skalierbarkeit evaluiert zu werden. Unter den gleichen Kriterien werden zudem klassische Modelle wie Bayes-Classifier und logistische Regression für einen Vergleich herangezogen. Zusätzlich wird untersucht, welche Parameter dieser Modelle bezüglich dieser Kriterien optimiert werden können. Dabei kann gezeigt werden, dass logistische Regression gute Ergebnisse für jede Datensatzgröße liefert, bei sehr großen Datenmengen jedoch von neuronalen Netzen, insbesondere RNNs, übertroffen wird.

1 Einleitung

In vielen Anwendungsgebieten der Informationsverarbeitung ist es erforderlich, Freitexte, die keiner vorher festgelegten Form entsprechen, zu verarbeiten. Eine Teildisziplin ist als *automatische Textklassifikation* bekannt: Dabei wird jedem Text (bzw. *Dokument*) jeweils eine von n vordefinierten *Klassen* (oder *Kategorien*) durch ein als *Classifier* bezeichnetes Programm zugewiesen[3]. Die resultierende Klasse kann dann genutzt werden, um mit dem Dokument weiter zu verfahren – eine unstrukturierte, prinzipiell beliebig große Eingabe führt dabei zu einer eindeutigen Entscheidung; dies kann etwa zum *E-Mail-Routing*, also dem Einsortieren von eingehenden E-Mails in thematisch angemessene Postfächer, eingesetzt werden.

Ein sehr einfacher Classifier kann die Anwesenheit von Schlüsselwörtern prüfen und auf dieser Basis eine Entscheidung treffen. Komplexere Regeln können formuliert werden, indem nicht nur die Anwesenheit, sondern auch die Häufigkeit von Wörtern oder Wortkombinationen überprüft wird. Diesen Ansätzen ist gemein, dass sie mit dem Wissen eines mentschlichen Experten aufgestellt werden, weshalb sie als *regelbasiertee Klassifikation* bzw. *Knowledge Engineering* (KE) bezeichnet werden [1].

Um diesen Schritt, der für jede Problemstellung spezifisches Fachwissen benötigt, zu vermeiden, werden heutzutage Methoden eingesetzt, die Classifier ohne das manuelle Definieren von Regeln erstellen. Diese Ansätze bedienen sich statistischer Modelle und Methoden und sind ein Teilgebiet des *Machine Learning* (ML). Dazu wird zunächst die *Architektur* des Classifiers formuliert, und anschließend wird dieser mit einem möglichst großen Datensatz *trainiert*, um die Parameter so zu setzen, dass die Ausgabe des Classifiers möglichst gut den tatsächlichen Klassen der Eingangsdaten entspricht. Diese Trainingsmethoden werden als *überwachtes Lernen* (*supervised learning*) bezeichnet, da die tatsächlichen Klassen (die *ground truth*) bekannt sind und als Richtwert für das Training dienen. Ein durch ein solches Verfahren trainierter Classifier arbeitet zwar deterministisch, hat also eindeutige Regeln, die jedoch eher implizit die passende Klasse auswählen und nicht dem für Menschen intuitiv verständlichen „Wenn-Dann"-Format entsprechen. Daher werden diese Ansätze mitunter als *nicht regelbasiert* bezeichnet.

[1] http://scikit-learn.org/

[2] https://www.tensorflow.org/

[3] Die allgemeinere Form der Klassifikation, bei der jedes Dokument einen reellwertigen Zugehörigkeitsgrad zu jeder Klasse hat, wird hier im weiteren nicht behandelt, kann jedoch ebenfalls mit den hier vorgestellten Verfahren durchgeführt werden.

Dieser technische Bericht basiert auf meiner bis August 2018 an der Hochschule Niederrhein durchgeführten Masterarbeit, für die ich einige aktuelle Ansätze zur Testklassifikation, deren Ergebnisse dem Stand der Technik entsprechen, evaluiert habe. Diese Modelle werden hier vorgestellt und die Ergebnisse der Evaluation wiedergegeben. Im Unterschied zu bisherigen Veröffentlichungen, die meist eine Auswahl aus englischsprachigen Datensätzen ohne einen zugrundeliegenden realen Anwendungsfall verwenden, habe ich dazu einen Datensatz aus 199 768 Notizen zu Störmeldungen, die von Technikern eines deutschen Telekommunikationsunternehmens geschrieben wurden, herangezogen. Eines der Ziele war dabei auch, die so trainierten Modelle in einer Produktivumgebung einzusetzen. Die getesteten Modelle werden in *klassische Modelle* und die aktuell populären *neuronalen Netze* unterteilt, die in den Abschnitten 2.3 und 3 beschrieben werden, nachdem die gemeinsamen Grundlagen in Abschnitt 2 erläutert wurden. Abschnitt 4 geht dann auf die Architektur der getesteten neuronalen Netze ein, d. h., wie die unterschiedlichen Komponenten neuronaler Netze, die zuvor vorgestellt wurden, kombiniert werden, um einen vollständigen Classifier zu konstruieren. In Abschnitt 5 werden die mit diesen Modellen durchgeführten Tests beschrieben und deren Ergebnisse gedeutet. Schließlich werden die wichtigsten Ergebnisse im Fazit in Abschnitt 6 noch einmal zusammengefasst.

2 Grundlagen

In diesem Kapitel werden die Grundlagen der eingesetzten Modelle und Algorithmen kurz erklärt, bevor diese in den folgenden Kapiteln näher vorgestellt werden.

2.1 Repräsentation der Daten

Um für die Klassifikation nutzbare Features aus Texten gewinnen zu können (*Feature Extraction*), muss der Text in eine geeignete Darstellung überführt werden. Ein einfacher Ansatz besteht darin, das Auftreten jedes Wortes als ein Feature zu betrachten. Da der Featurevektor eine feste Dimensionalität hat, muss das Vokabular V im Voraus bekannt sein. Der Featurevektor $\vec{x}$ hat demzufolge $|V|$ Komponenten, wobei jede Komponente genau einem Wort und jedes Wort genau einer Komponente zugeordnet ist. Für den Wertebereich der Elemente x_i gibt es verschiedene Varianten, von denen drei häufig verwendete für diesen Bericht evaluiert wurden:

1) $x_i \in \{0; 1\}$
   Dies ist das *Boolesche Modell*. Ist ein Wort in einem Dokument vorhanden, so ist die zugehörige Komponente des Vektors 1, ansonsten 0.

2) $x_i \in \mathbb{N}_0$
   Dieses Modell bezieht die *absolute Worthäufigkeit* (*Term Frequency*, TF) eines Wortes mit ein. Jede Komponente gibt die Anzahl der Vorkommnisse eines Wortes im Dokument an.

3) $x_i \in \mathbb{R}_{>0}$
   Bei reellwertigen Features ist eine oft verwendete Möglichkeit die *TF-IDF-Gewichtung*: Die Komponente des Vektors ist größer, je häufiger das Wort im Dokument vorkommt, und niedriger, in je mehr Dokumenten das Wort auftritt (*Inverse Document Frequency*, IDF). Der Grundgedanke dieser Gewichtung (und ihrer Varianten) liegt darin, dass die Gewichte von Wörtern mit mutmaßlich geringer Trennschärfe verringert werden. Für diesen Bericht wird die folgende Berechnungsvorschrift eingesetzt:

$$\begin{aligned}\text{tf-idf}_{t,d} &= \text{tf}_{t,d} \cdot \text{idf}_t \\ &= \text{tf}_{t,d} \cdot \left(\log \left(\frac{|D|}{\text{df}_t} \right) + 1 \right)\end{aligned}$$

   Dabei ist $\text{tf}_{t,d}$ die Häufigkeit des Wortes t im Dokument d, df_t die Anzahl der Dokumente, in denen das Wort t vorkommt und $|D|$ die Gesamtanzahl der Dokumente.

Diese Modelle sagen über ein Dokument lediglich aus, *welche* Wörter enthalten sind nicht aber, in welcher Reihenfolge. Diese Modelle werden daher als *Bag of Words* (BoW) bezeichnet, da diese Darstellung einer *Multimenge* von Wörtern entspricht.

Diese Darstellung als Featurevektor entspricht wiederum dem *Vektorraummodell*, das im Information Retrieval Anwendung findet. Die Featurevektoren $\vec{x}_i$ sind also Teil des Vektorraums $X^{|V|}$, der wiederum von $|V|$ Basisvektoren (in diesem Zusammenhang auch als *One-Hot-Vektoren* bezeichnet) aufgespannt wird[4]. Diese Basisvektoren sind genau die einzelnen Terme des

[4] Je nach verwendetem Modell ist $X = \{0; 1\}$, $X = \mathbb{N}_0$ oder $\mathbb{R}_{>0}$, wobei nachfolgend vom allgemeinsten Fall $X = \mathbb{R}$ ausgegangen wird, der die Obermenge dieser Mengen darstellt.

Vokabulars V, und jeder Dokument-Featurevektor $\vec{x}(d)$ ist eine Linearkombination dieser Basisvektoren:

$$\vec{x}(d) = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_{|V|} \end{pmatrix} = x_1 \begin{pmatrix} 1 \\ 0 \\ \vdots \\ \vdots \\ 0 \end{pmatrix} + x_2 \begin{pmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} + \ldots + x_{|V|} \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}$$

Während es intuitiv erscheint, dass mit einem solchen Modell Texte klassifiziert werden können, deren Themen durch auftretende Fachbegriffe charakterisiert sind, so können Sachverhalte, in denen die Wortreihenfolge eine Rolle spielt, nicht adäquat dargestellt werden. Um dies zu ermöglichen, wird das Modell erweitert, indem die Wortvektoren der einzelnen Wörter aneinandergefügt werden, sodass sich eine $|V| \times |S|$-Satzmatrix S ergibt:

$$S = \left[\begin{pmatrix} s_{1,1} \\ s_{2,1} \\ \vdots \\ s_{|V|,1} \end{pmatrix}, \begin{pmatrix} s_{1,2} \\ s_{2,2} \\ \vdots \\ s_{|V|,2} \end{pmatrix}, \ldots, \begin{pmatrix} s_{1,|S|} \\ s_{2,|S|} \\ \vdots \\ s_{|V|,|S|} \end{pmatrix} \right]$$

$$= \begin{bmatrix} s_{1,1} & s_{1,2} & \ldots & s_{1,|S|} \\ s_{2,1} & s_{2,2} & \ldots & s_{2,|S|} \\ \vdots & \vdots & \ddots & \vdots \\ s_{|V|,1} & s_{|V|,2} & \ldots & s_{|V|,|S|} \end{bmatrix}$$

Da jeder Wortvektor ein Basisvektor ist, ist die Matrix dünnbesetzt. Zur effizienten Speicherung kann die Matrix daher auch als Zeilenvektor dargestellt werden:

$$\hat{S} = \begin{bmatrix} \hat{s}_1 & \hat{s}_2 & \ldots & \hat{s}_{|S|} \end{bmatrix} \tag{1}$$

Dabei bezeichnen die Einträge des Zeilenvektors je den $\hat{s}_i$-ten Basisvektor des Wortvektorraums. Diese Darstellung ist zur Matrixdarstellung äquivalent und vereinfacht die Interpretation als Featurevektor: Jeder Satz hat damit $|S|$ verschiedene Features, die je einen diskreten Wert aus $\{1, \ldots, |V|\}$ annehmen können. Jedes Feature kann damit als die Frage formuliert werden: „Welches Wort s_i steht an Position i?“

Daraus folgt, dass die Satzlänge $|S|$ konstant sein muss, denn diese bestimmt die Dimensionalität des Featurevektors (bzw. die Breite der Satzmatrix). Da Sätze in Klassifikationsproblemen in der Regel verschiedene Längen haben, müssen längere Sätze (unter Informationsverlust) gekürzt und kürzere durch einen speziellen Vektor (etwa den Nullvektor oder aber einen weiteren Basisvektor, wofür der Vektorraum um eine zusätzliche Dimension erweitert wird) aufgefüllt werden, der meist als *Padding*-Vektor oder -Token (kurz *PAD*) bezeichnet wird. Ebenfalls ist der Fall zu betrachten, dass in einem zu klassifizierenden Text ein Wort auftritt, das nicht bereits im Training bekannt war, was als *Out-Of-Vocabulary* (OOV) bezeichnet wird. Während dies im Vektorraummodell nicht gesondert behandelt werden muss (da dort die Features schlicht die Vorkommnisse bereits bekannter Wörter sind und ein unbekanntes Wort somit kein Feature ist), ist es im Satzmodell erforderlich, dass jeder Position des Satzes ein Featurewert zugewiesen werden kann. Dazu kann sich abermals mit einem weiteren Basisvektor beholfen werden: Tritt ein unbekanntes Wort in einem Satz auf, bekommt das entsprechende Feature diesen als *Unknown*-Token (kurz *UNK*) bezeichneten Vektor zugewiesen.

Die Darstellung als Matrix erhält zwar Positionsinformationen, doch ist die Wahl der Features nicht unbedingt optimal:

1) Jedes Feature kann $|V|$ verschiedene Werte annehmen, doch diese Werte drücken keine quantitativen Maße aus, die miteinander verglichen werden, d. h. die Werte können zwar als ganze Zahlen dargestellt werden, etwa $\hat{s}_1 = 5$, doch besteht keinerlei Beziehung zu $\hat{s}_1 = 6$ oder $\hat{s}_1 = 4$. In vielen Fällen trifft dies jedoch nicht zu; so stehen etwa die Wörter *Buche* und *Eiche* thematisch in einer Beziehung, doch deren Darstellungen im Vektorraummodell sind genau so weit voneinander entfernt wie die von *Giraffe* und *Mond*. Anders formuliert: Wenn es Korrelationen zwischen Featurewerten (also den Zeilen der Satzmatrix) gibt, so werden diese durch dieses Modell nicht ausgedrückt.

2) Ein Wort kann an verschiedenen Stellen des Satzes auftreten. Ein Satz mit $\hat{s}_7 = 5$ hat vielleicht eine ähnliche Bedeutung wie ein Satz mit $\hat{s}_8 = 5$, doch sind dies zwei verschiedene Features, die lediglich den gleichen Wert annehmen. Auch die Werte in den Spalten der Matrix können also korreliert sein, was durch dieses Modell ebenfalls noch nicht klar wird.

Punkt 1 liegt am zugrundeliegenden Vektorraummodell. Da alle Wortvektoren Basisvektoren sind, die einen Vektorraum aufspannen, liegen diese alle ausschließlich in der ihnen zugehörigen Dimension und sind stets orthogonal zueinander. Um diesem Problem entgegenzuwirken, können die Vektoren aus dem hochdimensionalen Raum $X^{|V|}$ in einen Raum mit weniger Dimensionen X^E projiziert werden. Diese Methoden werden als *Word Embedding* bezeichnet, und die resultierenden, niedrigdimensionalen Vektoren schlicht als *Embeddings* (*Einbettungen*).

Ziel dieser Einbettungen ist nicht nur, die Dimensionalität (und damit die Anzahl erforderlicher Variablen) zu reduzieren, sondern auch, ähnliche Darstellungen für semantisch ähnliche Wörter zu erhalten, sodass diese im Vektorraum nah beieinander liegen. Jede Komponente des Vektors kann damit als eine Eigenschaft des Wortes interpretiert werden, und der Vektor als Ganzes kodiert die Bedeutung des Wortes. Einbettungen können beispielsweise mittels Singulärwertzerlegung oder neueren Ansätzen wie *Word2Vec* [2] oder *fastText*[5] [3] erzeugt werden und dann als Eingabe für einen Classifier dienen. Ist dieser Classifier ein neuronales Netz, so ist es ebenfalls möglich, die One-Hot-Vektor-Darstellung (bzw. die indizierte Darstellung) als Eingabe zu verwenden und daraus im Verlauf des Trainings Einbettungen zu errechnen, die dann in tieferen Schichten des Netzes verwendet werden.

2.1.1 Vorverarbeitung und Tokenisierung

Es bleibt zu klären, wie die *Tokenisierung* – d. h. die Extraktion von Tokens (Wörter, Zeichen, Wort-n-Gramme[6] oder Zeichen-n-Gramme[7]) aus dem Dokument – erfolgt.

Eine sehr grundlegende Art, Tokens zu erhalten, besteht darin, den Text an allen Leerzeichen aufzuteilen. Da es neben dem normalen Leerzeichen auch weitere Zeichen gibt, die ähnliche Funktionen erfüllen – etwa Zeilenumbrüche, nicht umbrechbare Leerzeichen und Leerzeichen ohne Breite – werden diese unter dem Begriff *Whitespace* zusammengefasst und gleich behandelt[8]. Die Eingabe „Dies ist ein Beispielsatz.“ wird damit zur Tokenmenge $\{$"Dies", "ist", "ein", "Beispielsatz."$\}$.

Dabei wird ein neues Problem ersichtlich: Die Behandlung von Satzzeichen. Prinzipiell gibt es dabei verschiedene Möglichkeiten, wobei in den für diesen Bericht durchgeführten Tests Satzzeichen als eigene Tokens betrachtet werden[9].

Diese sehr einfache Variante der Tokenisierung produziert jedoch immer noch viele verschiedene Terme, die bei genauerer Betrachtung (nahezu) äquivalent sind. Aus diesem Grund können Vorverarbeitungsschritte angewendet werden, um mehrere Tokens auf die gleiche Darstellung abzubilden. Dazu gehören unter anderem das Entfernen oder Ersetzen seltener Zeichen (einschließlich Umlaute) und *Lowercasing* (sodass Wörter ungeachtet ihrer Groß- und Kleinschreibung als äquivalent betrachtet werden). Neben dem Zusammenfassen von Termen ist es ebenfalls möglich, zu seltene und/oder zu häufige Terme (*Stoppwörter*) zu entfernen, was – wie auch die TF-IDF-Gewichtung – auf der Annahme fußt, dass diese über keinen signifikanten Informationsgehalt verfügen. Diese und andere Techniken werden als *Feature Selection* bezeichnet.

2.2 Bewertung

Bei der Bewertung (und den Vergleich) eines Classifiers sind zwei grundlegende Begriffe zu unterscheiden: *Effektivität* und *Effizienz*, wobei letztere als sekundäres Kriterium zu betrachten ist.

Effektivität. Dieser Begriff bezeichnet die Güte eines Classifiers. Das wichtigste Maß bei Textklassifikationsproblemen ist der Anteil der korrekt klassifizierten Dokumenten an der Gesamtheit der Dokumente und wird als *Erkennungsrate*, oder – in Anlehnung an den in englischsprachigen Veröffentlichungen verwendeten Begriff *Accuracy* – als *Genauigkeit* bezeichnet. Das Gegenteil der Genauigkeit ist die *Fehlerrate* (Error Rate $= 1 -$ Accuracy). Die Genauigkeit bzw. die Fehlerrate muss immer vor dem Hintergrund der Klassenverteilung betrachtet werden. Gehören 95% der Dokumente der gleichen Klasse an, so kann ein Classifier, der nichts weiter tut, als sich stets für die häufigste

[5] `https://fasttext.cc/`

[6] Mit diesem Begriff werden in diesem Bericht Kombinationen aus n benachbarten Wörtern bezeichnet; einzelne Wörter sind ein Spezialfall der Wort-n-Gramme mit $n = 1$

[7] Wobei Zeichen wiederum ein Spezialfall der Zeichen-n-Gramme mit $n = 1$ sind

[8] Prinzipiell haben diese Zeichen jedoch unterschiedliche semantische Bedeutungen.

[9] Dies ist insbesondere im Satzmatrix-Modell sinnvoll; im Vektorraummodell hingegen ist davon auszugehen, dass Satzzeichen keine besonders große Trennschärfe haben, da diese ohne Positionsinformationen nahezu bedeutungslos sind

Klasse zu entscheiden, eine Genauigkeit von 95% erzielen. Die Fehlerrate (hier: 5%), die erzielt wird, wenn stets die häufigste Klasse ausgewählt wird, wird als *Null Error Rate* bezeichnet, da für diese Fehlerrate keinerlei Features betrachtet werden müssen und kein „sinnvoller" Classifier diese Fehlerrate überschreitet.

Genauere Informationen liefert die *Confusion Matrix* (*Wahrheitsmatrix*), die die Anzahl der Dokumente für jede Kombination aus vom Classifier erkannter Klasse und tatsächlicher Klasse darstellt. Tabelle 1 zeigt ein Beispiel für eine solche Matrix.

Die Zeile gibt die korrekte Klasse eines Dokuments an, während die Spalte die Klasse angibt, die der Classifier ausgewählt hat. Auf der Diagonalen stehen damit alle korrekt klassifizierten Dokumente, die *True Positives*. Die Accuracy ist der Anzahl der True Positives an der Gesamtheit der Dokumente, d. h. Accuracy $= \frac{7+6+7}{30} = \frac{2}{3} \approx 66{,}67\%$. Darüber hinaus gibt es klassenspezifische Statistiken, etwa den *Recall*, der für jede Klasse den Anteil der Dokumente angibt, die vom Classifier korrekt erkannt wurden (d. h. der Quotient aus Diagonaleintrag und Zeilensumme). So beträgt der Recall für die Klassen *Politik* und *Unterhaltung* 70%, während für die Klasse *Sport* nur 60% der Dokumente korrekt klassifiziert werden konnten. Das Gegenstück dazu ist die *Precision*, die den Anteil der korrekt klassifizierten Dokumente an allen Dokumenten, die dieser Klasse zugewiesen wurde, angibt (d. h. der Quotient aus Diagonaleintrag und Spaltensumme). So hat *Politik* eine Precision von 70%, während es bei *Sport* ca. 85,71% und bei *Unterhaltung* etwa 76,92% sind. Die Confusion Matrix liefert auch eine detailliertere Sicht auf den Classifier: So lässt sich etwa sagen, dass der Classifier *Politik* und *Unterhaltung* nicht so gut unterscheiden kann wie etwa *Politik* und *Sport*.

Die Effektivität des Classifiers ist aber immer vor dem Hintergrund der verwendeten Daten zu betrachten. Ist ein Classifier ausdrucksstark genug (d. h. ist seine *Kapazität* groß genug), so kann er Daten, die für das Training verwendet werden, immer korrekt klassifizieren. Dies sagt jedoch nichts darüber aus, wie gut neue Daten erkannt werden, was jedoch der praktische Anwendungsfall ist. Insbesondere besteht die Gefahr des *Overfittings*, d. h., dass der Classifier zu sehr an die Trainingsdaten angepasst wurde, sodass er neue Daten nicht erkennt[10]. Feature Selection kann genutzt werden, um überspezifische, seltene Features zu entfernen (für neuronale Netze existieren viele weitere Techniken, siehe Abschnitt 3.4.3).

Aus diesem Grund wird die Genauigkeit eines Classifier auf einer separaten, zu den Trainingsdaten disjunkten Menge – den *Testdaten* (*Test Set*) – evaluiert. Die Testdaten werden entweder vorher bestimmt, oder es wird vor dem Training ein Teil der Trainingsdaten entfernt und als Testdatensatz verwendet (die Testdaten werden dann auch als *Holdout Set* bezeichnet). Wichtig ist, dass das Test-Set niemals während des Trainings genutzt werden darf[11]. Für einige Methoden (siehe auch Abschnitt 3.4.3) wird auf die gleiche Weise eine weitere Menge benutzt, die als *Validation Set* (oder *Dev Set*) bezeichnet wird und dazu dient, den Classifier so anzupassen, dass Overfitting reduziert wird. Im Gegensatz zum Test-Set wird das Dev-Set während des Trainings verwendet (wobei die Daten nicht in der gleichen Weise

[10] Als Extrembeispiel sei ein Classifier genannt, der jedes Dokument im Volltext abspeichert und diesem Volltext jeweils die passende Klasse zuweist. Jedes Dokument aus dem Trainingsdatensatz kann dann mit einer Genauigkeit von 100% klassifiziert werden, aber auf unbekannten Daten wird bestenfalls die Null Error Rate erreicht; es besteht also extremes Overfitting.

[11] Selbst, wenn dieser Grundsatz eingehalten wird, besteht immer noch das Risiko des *impliziten* Overfittings: Werden mehrere Classifier trainiert, wählt ein Anwender den Classifier aus, der auf dem Test-Set das beste Ergebnis erzielt hat. Doch möglicherweise liegt eine höhere Genauigkeit nicht am Classifier, sondern an den Gegebenheiten des Test-Sets selbst. Der Classifier funktioniert dann besser, obwohl er nicht explizit auf dieser Menge trainiert wurde.

		Erkannte Klasse			**Recall**
		Politik	**Sport**	**Unterhaltung**	
Tatsächliche Klasse	**Politik**	**7**	0	3	70,00%
	Sport	1	**6**	3	60,00%
	Unterhaltung	2	1	**7**	70,00%
Precision		70,00%	85,71%	76,92%	

Tabelle 1: Ein Beispiel für eine Confusion Matrix einer Klassifikation nach Themenbereichen mit 30 Dokumenten.

betrachtet werden, wie es beim Trainings-Set der Fall ist). Zusammenfassend gilt für die Mengen *Train*, *Dev* und *Test*:

$$(\text{Train} \cap \text{Dev}) \cup (\text{Dev} \cap \text{Test}) \cup (\text{Test} \cap \text{Train}) = \emptyset$$

Effizienz. Damit wird bezeichnet, wie sparsam ein Programm mit Ressourcen umgeht, was sich insbesondere auf die aufgewendete (Rechen-)Zeit bezieht. Da bei der Implementation der getesteten Methoden sehr viele Algorithmen aus unterschiedlichen Programmbibliotheken zusammenwirken, ist es schwierig, die exakte Laufzeit (etwa die *Laufzeitkomplexität* in O-Notation) zu bestimmen. Daher beschränkt sich die Bestimmung der Effizienz auf die Messung der Zeit, die für die Ausführung des Programms benötigt wird. Damit die gemessenen Werte Aussagekraft haben, ist darauf zu achten, dass alle Programme unter gleichen Bedingungen ausgeführt werden, d. h. mit gleicher Hardware, gleicher Software und gleichen Daten. Selbst dann unterliegen die Werte Schwankungen, etwa durch laufende Hintergrundprozesse, durch im Programm verwendete Zufallszahlen oder schlicht durch physikalische Effekte (etwa leicht schwankende Taktraten des Prozessors und des Speichers), weswegen diese Tests mehrfach durchgeführt werden müssen, um einen aussagekräftigeren Mittelwert bilden zu können.

Weiterhin wird der Speicherverbrauch betrachtet. Dieser fließt zwar nicht unmittelbar in die Bewertung eines Classifiers ein, kann aber einschränken, auf welcher Hardware dieser eingesetzt werden kann.

2.3 Klassische Modelle

Um die Ergebnisse neuronaler Netze besser deuten zu können, werden für diesen Bericht auch Modelle evaluiert, die bereits vor der jüngsten Renaissance der neuronalen Netze etabliert waren. Die Implementation dieser klassischen Modelle baut hierbei auf die Bibliothek *scikit-learn*[12] (sklearn) auf, die viele Funktionen zur Feature Extraction, Klassifikation und Evaluation bietet. Dabei wurden konkret die folgenden Methoden getestet:

Naive Bayes Classifier. Zu beachten ist, dass der Bayes-Classifier auf der Annahme fußt, dass die einzelnen Features stochastisch unabhängig sind, obwohl dies im Allgemeinen nicht der Fall ist[13]. Als Erweiterung bzw. Verallgemeinerung in Vektorraummodellen mit Termhäufigkeit (TF) oder TF-IDF kommt der *Multinomial Naive Bayes Classifier* (MNB) zum Einsatz, der die relative Häufigkeit jedes Terms innerhalb des gesamten Textes der Klasse (d. h. ohne Berücksichtigung der Dokumentgrenzen) betrachtet, anstatt nur die Wahrscheinlichkeit der Anwesenheit der einzelnen Terme zu modellieren.

Complement Naive Bayes. Dieser Classifier ist eine Abwandlung des MNB und wurde unter anderem für Textklassifikation mit ungleich verteilten Klassen entwickelt. Statt $P(x_i \mid c)$, also die Wahrscheinlichkeiten des Features x_i in der Klasse c zu berechnen, wird zu jeder Klasse c eine Komplementärklasse $\bar{c}$ gebildet, die alle $\vec{x} \notin c$ enthält. Die Klassenwahrscheinlichkeit $P(c)$ wird dann mit den Kehrwerten dieser Wahrscheinlichkeiten multipliziert, um die wahrscheinlichste Klasse zu erhalten [4].

Support Vector Machines. In ihrer grundlegendsten Form sind SVMs Classifier für Zwei-Klassen-Probleme, entscheiden also lediglich, ob ein Dokument d zu einer Klasse c gehört oder nicht. Es gibt verschiedene Möglichkeiten, SVMs für Mehrklassenprobleme zu erweitern. Beim sogenannten *One-Against-Rest* wird für jede der n Klassen ein SVM-Modell erstellt, das Dokumente dieser Klasse von den Dokumenten anderer Klassen unterscheiden kann. Da es dabei jedoch Überschneidungen geben kann, wird die Klasse ausgewählt, dessen Entscheidungsfunktion den höchsten Wert hat. Ein anderes Verfahren ist das *One-Against-One*-Verfahren: Dabei werden insgesamt $\frac{n^2-n}{2}$ SVM-Modelle konstruiert, wobei jedes Modell die Trainingsdaten von genau zwei Klassen erhält, sodass es schließlich für jede Kombination aus zwei Klassen ein Modell gibt, das diese Klassen unterscheidet. Soll nun ein neues Dokument klassifiziert werden, so wird dieses mit allen Modellen ausgewertet und schließlich die Klasse gewählt, für die am häufigsten entschieden wurde[14]. Dieses Verfahren ist effektiver als das One-Against-Rest-Verfahren [5].

[12] `http://scikit-learn.org/`

[13] Dies zeigt sich auch in der Namensgebung „*Naive* Bayes Classifier“

[14] Im Falle eines Gleichstands kann die Klasse mit dem niedrigsten Index ausgewählt werden.

Logistische Regression. Diese ist ein Regressionsmodell, dem die logistische Funktion zugrundeliegt. Im Gegensatz zu Modellen wie etwa der linearen Regression bedeutet dies, dass der Wertebereich der Regression auf das Intervall $(0; 1)$ beschränkt ist. Dient ein Featurewert als Eingabe, so kann das Ergebnis einer entsprechend parametrisierten logistischen Funktion als Klassenwahrscheinlichkeit in einem Zwei-Klassen-Problem interpretiert werden, wodurch die Regressionsfunktion als Classifier eingesetzt werden kann. Analog zu SVMs kann im Fall mit mehreren Klassen der One-Against-Rest-Ansatz verfolgt werden. Dabei wird dann jeweils die Klasse ausgewählt, deren Classifier die höchste Wahrscheinlichkeit errechnet hat.

3 Neuronale Netze

Die besten Ergebnisse in vielen Klassifikationsaufgaben erzielen heutzutage neuronale Netze (NN); auch in anderen Anwendungsgebieten sind neuronale Netze ein vieldiskutiertes Thema, an dem gegenwärtig sehr viel geforscht wird [6].

Oft gibt es verschiedene, in ihrer exakten Funktionsweise abweichende Implementationen einzelner Komponenten neuronaler Netze. Da die für diesen Bericht implementierten neuronalen Netze die Funktionen der Bibliothek *TensorFlow*[15] (TF) einsetzen, orientieren sich die nachfolgenden Beschreibungen im Zweifelsfall an der jeweiligen Implementation in TensorFlow.

3.1 Grundlagen

Im Mittelpunkt steht das künstliche Neuron, das eine Menge von n Eingaben erhält und daraus eine gewichtete Summe berechnet. Auf diese Summe wird ein *Bias* genannter Wert b addiert, und auf das Ergebnis wird schließlich eine nichtlineare *Aktivierungsfunktion* f angewendet, um die Ausgabe o zu erhalten. Mit den Eingaben $x_1, x_2, \ldots, x_n$ und den Gewichten $w_1, w_2, \ldots w_n$ ergibt sich damit die einfache Formel[16]:

$$o = f\left(b + \sum_{i=1}^{n} w_i \cdot x_i\right)$$

Werden die Eingabe als Spaltenvektor $\vec{x}$ und die Gewichte als Zeilenvektor $\vec{w}$ betrachtet, so ergibt sich die Formel:

$$\begin{aligned} o &= f\left(b + \vec{w} \cdot \vec{x}\right) \\ &= f\left(b + \begin{pmatrix} w_1 & w_2 & \ldots & w_n \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}\right) \end{aligned}$$

Da in den meisten Anwendungen ein einzelnes künstliches Neuron nicht ausreichend ist, werden mehrere künstliche Neuronen mit jeweils eigenen Gewichten auf die gleiche Eingabe angewendet. Bei m Gewichten ergibt sich damit eine $m \times n$-Gewichtsmatrix W und ein m-dimensionaler Bias-Vektor $\vec{b}$. Die Aktivierungsfunktion $\vec{f}$ wird dann elementweise auf den resultierenden m-dimensionalen Vektor angewendet, sodass sich die m-dimensionale Ausgabe $\vec{o}$ ergibt:

$$\vec{o} = \vec{f}\left(W \cdot \vec{x} + \vec{b}\right)$$

Dass die grundlegende Operation in neuronalen Netzen also eine Matrix-Vektor-Multiplikation ist, bedeutet, dass diese in heutiger Hardware sehr schnell berechnet werden können, da die Berechnungen parallelisierbar sind und Prozessoren über Instruktionen verfügen, die speziell auf Matrix-Vektor-Multiplikationen ausgelegt sind. Insbesondere Grafikprozessoren (GPUs) stechen hierbei hervor.

Eine einzelne Matrix-Vektor-Multiplikation mit Nichtlinearisierung ist jedoch meist nicht ausreichend. Aus diesem Grund können beliebig viele weitere derartige Operationen auf die Ausgaben angewendet werden. Die Kombination aus Matrix-Vektor-Multiplikation und Aktivierungsfunktion wird dann als eine Schicht bzw. als ein *Layer* bezeichnet – in Abgrenzung zu Schichten, die andere Operationen durchführen (und in den folgenden Abschnitten näher erläutert werden), genauer als *Fully Connected Layer* (FC-Layer). Auf diese Verkettung von Operationen geht die Bezeichnung „Netz“ zurück, und je mehr Schichten es hat, desto „tiefer“ ist es. In dieser Möglichkeit der tiefen Verschachtelung liegt zugleich auch die Stärke neuronaler Netze, denn durch eine Vergrößerung der Tiefe lassen sich oft bessere Ergebnisse erzielen als durch eine Vergrößerung der

[15] `https://www.tensorflow.org/`

[16] Alternativ kann der Bias-Wert auch als ein zusätzliches Gewicht w_0 dargestellt werden, der dann auf einen zusätzlichen Eingangswert $x_0 = 1$ angewendet wird.

„Breite“ – also die Anzahl der Gewichte einer Schicht – um die gleiche Variablenanzahl [7].

In der Praxis werden Breite und Tiefe eines Netzes experimentell bestimmt. Wichtig ist dabei lediglich, dass die Eingabegröße der ersten Schicht der Anzahl der Features und die Ausgabegröße der letzten Schicht dem gewünschten Ergebnis entspricht – bei einem Klassifikationsproblem etwa einem *Score*-Wert für jede Klasse, wobei die Klasse mit dem höchsten Wert ausgewählt wird. Die Schichten dazwischen – die *Hidden Layer* – können dabei in Funktionsweise, Größe und Anzahl beliebig gewählt werden.

3.1.1 Training

Die Gewichte neuronaler Netze werden nicht exakt bestimmt, sondern mit zumeist zufälligen Werten initialisiert und dann iterativ verbessert, sodass die optimale Lösung approximiert wird. Dies geschieht, indem eine *Kosten-* oder *Verlustfunktion* (*Loss*) $L(\hat{y}, y)$ aufgestellt wird, die ein Maß der Unähnlichkeit zwischen der Ausgabe des Netzes $\hat{y}$ und den tatsächlichen Klassen y darstellt. Diese Funktion soll dann minimiert werden, indem sie bezüglich der Gewichte der Ausgabeschicht abgeleitet wird. Dazu kommt der *Backpropagation*-Algorithmus zum Einsatz, der wiederholt die Kettenregel der Differentialrechnung anwendet, um die *Gradienten* für alle weiteren Schichten zu berechnen.

Um die Gewichte der einzelnen Schichten basierend auf den berechneten Gradienten anzupassen, kommt ein Optimierungsalgorithmus (*Optimizer*) zum Einsatz. Ein klassisches Beispiel ist *Stochastic gradient descent* (SGD), der die Gewichte anpasst, indem der Gradient mit einer *Lernrate* η multipliziert und von dem Gewicht subtrahiert wird. In den in diesem Bericht verwendeten neuronalen Netzen kommt hingegen der 2014 entwickelte Algorithmus *Adam* [8] zum Einsatz, der Näherungswerte der ersten und zweiten stochastischen Momente der Gradienten berechnet, um eine individuelle Lernrate für jedes Gewicht zu erhalten. Dieser Wert wird dann wiederum mit dem Faktor η skaliert.

Im sogenannten *Online-Training* wird jeweils ein einzelnes Dokument in das neuronale Netz eingegeben und dann unmittelbar die Gewichte angepasst, bevor das nächste Dokument eingegeben wird. Dies wird wiederholt, bis ein Abbruchkriterium (wie etwa eine bestimmte Anzahl an Trainingsschritten) erreicht ist. Wurden alle Dokumente genau einmal eingegeben, wird dies als eine *Epoche* bezeichnet. Eine andere Variante ist das *Batch-Training*, bei dem alle Dokumente auf einmal eingegeben werden. Die Gewichte werden dann basierend auf dem Mittelwert der Kostenfunktion angepasst. Einen Kompromiss stellt das *Minibatch-Training* dar, bei dem die Trainingsdaten in sogenannte *Minibatches* mit je k Dokumenten partitioniert werden. Diese Teilmengen werden dann nacheinander eingegeben, wobei nach jeder Teilmenge die Gewichte angepasst werden. Wurden alle Minibatches einer Epoche abgearbeitet, kann dieser Vorgang (mit einer neuen Partition) wiederholt werden.

3.1.2 Aktivierungsfunktionen

Für die Wahl der Aktivierungsfunktion bieten sich viele Möglichkeiten, wobei die beste Wahl letztlich durch Tests gefunden werden muss. Häufig verwendet wird die *Rectified Linear Unit* (ReLU), die 0 zurückgibt, sobald der Eingangswert negativ ist, d. h.:

$$\mathrm{ReLU}(x) = \max(x, 0)$$

Daraus ergibt sich jedoch das Problem, dass der Gradient 0 ist, wenn der Eingangswert negativ war. Dadurch kann das Training verlangsamt oder gar verhindert werden. Um diesem Problem entgegenzuwirken, kann als Variante die *Parameterized ReLU* eingesetzt werden, bei der der negative Teil nicht auf 0 gesetzt, sondern mit einem kleinen, trainierbaren Parameter a multipliziert wird:

$$\mathrm{PReLU}(x) = \max(x, 0) + a \cdot \min(x, 0)$$

Dabei kann entweder ein Parameter a für jeden Eintrag von x gelernt werden, oder es wird lediglich ein einziges a für den gesamten Vektor x verwendet, wodurch für jede Schicht des Netzes nur ein einzelner Parameter dazukommt [9].

Ein ähnliches Konzept verfolgt die sogenannte *Exponential Linear Unit* (ELU). Diese ist folgendermaßen definiert:

$$\mathrm{ELU}(x) = \begin{cases} x & \text{falls } x > 0 \\ e^x - 1 & \text{sonst} \end{cases}$$

Der positive Teil entspricht damit der ReLU. Analog zur PReLU gibt es ebenfalls eine parametrisierte Form,

die *Scaled ELU* (SELU). Diese verfügt über zwei trainierbare Parameter λ und a [10]:

$$\text{SELU}(x) = \lambda \cdot \begin{cases} x & \text{falls } x > 0 \\ a \cdot e^x - a & \text{sonst} \end{cases}$$

Darüber hinaus gibt es einige weitere Aktivierungsfunktionen, die häufig verwendet werden, etwa $\tanh$. Abb. 1 visualisiert die in diesem Bericht getesteten Funktionen.

3.2 Convolutional Neural Networks

Eine Eigenschaft, die allen bisher vorgestellten Modellen (einschließlich neuronaler Netze) gemein ist, ist die Annahme, dass die verwendeten Features voneinander unabhängig sind. In der Textklassifikation ist diese Annahme jedoch falsch, denn die Features sind sehr stark korreliert. Im Vektorraummodell betrifft dies etwa Synonyme und häufige Wort-n-Gramme, während ein Modell, das die Positionen der Wörter berücksichtigt, jedes Wort an jeder Stelle als eigenes Feature betrachtet – so würde etwa das Wort „Hallo“ an erster Stelle als unkorreliert zum Wort „Hallo“ an zweiter Stelle betrachtet werden. Dadurch kann ein solches Modell nicht von der Bedeutung eines Wortes an einer Position auf die Bedeutung des gleichen Wortes an einer anderen Position schließen und bräuchte geeignete Trainingsdaten für das Wort an jeder Position, würde also jedes Feature mehrfach lernen müssen.

Da dies praktisch nicht durchführbar ist, wurden Modelle entwickelt, die von dieser Annahme abrücken und stattdessen explizit davon ausgehen, dass Features korreliert sind. Dies erlaubt es im Bereich der Textklassifikation nicht nur, Wörter (bzw. n-Gramme) an verschiedenen Positionen zu erkennen, sondern auch, die Anzahl notwendiger Parameter zu reduzieren, indem die gleichen Parameter an verschiedenen Stellen der Eingabe ausgewertet werden. Diese Technik wird als *Parameter Sharing* bezeichnet und kann auf verschiedene Weisen umgesetzt werden, etwa von den sogenannten *Convolutional Neural Networks* (CNN).

Diese Netzwerke erhalten ihren Namen von der *diskreten Faltungsoperation*, der *Convolution*, die auf die Eingaben einer Schicht – des *Convolutional Layers* – angewendet wird. Ein anschauliches Beispiel ist der zweidimensionale Fall der diskreten Faltung, genauer gesagt der *diskreten Kreuzkorrelation*, die eine Funktion F auf jedem Punkt einer Eingabe I auswertet, um

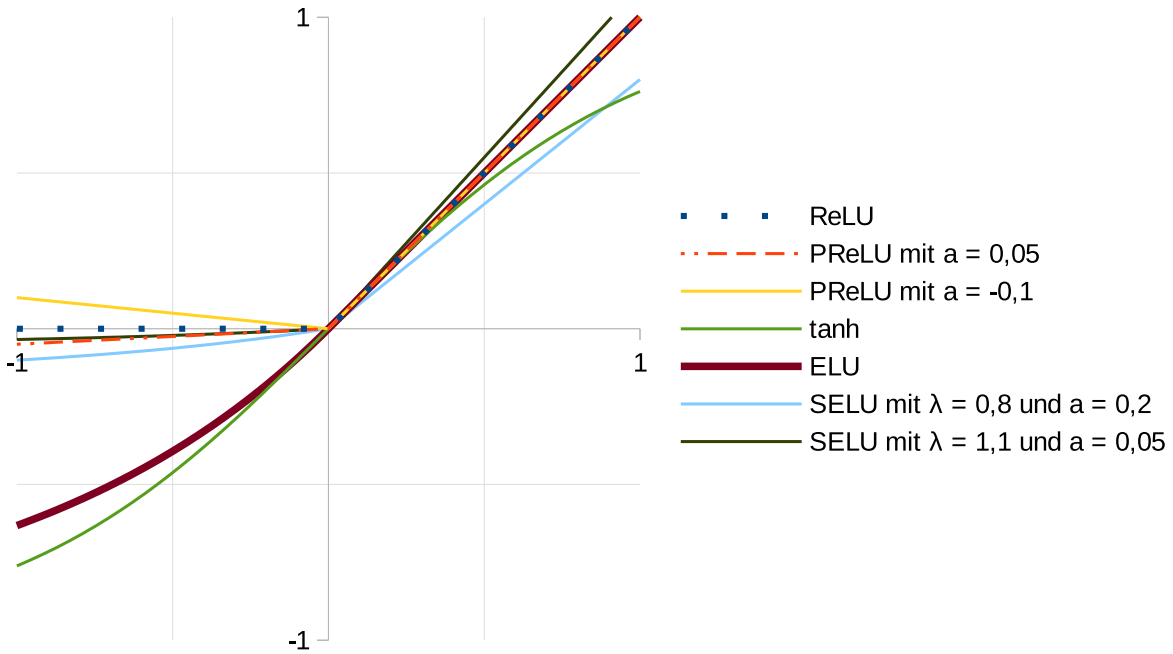


Abbildung 1: Übersicht über die Graphen einiger Aktivierungsfunktionen. Für $x > 0$ liefern ELU, ReLU und PReLU (ungeachtet des Parameters a) stets den gleichen Wert und überlagern sich daher im Graphen.

die Ausgabe O zu erhalten[17]:

$$\begin{aligned} O(i,j) &= (I \star F)(i,j) \\ &= \sum_m \sum_n I(i+m, j+n) \cdot F(m,n) \end{aligned}$$

Die Eingabe I kann als ein Bild und die Funktion F als ein Filter betrachtet werden[18], z. B. der Gauß-Filter, d. h. der zweidimensionale, diskrete und endliche Fall der Gewichtung mit der Normalverteilung. Für eine Filtergröße von 3×3 Pixeln ergibt sich:

$$F = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \\ \frac{1}{8} & \frac{1}{4} & \frac{1}{8} \\ \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \end{bmatrix}$$

Convolution bezeichnet dann nichts weiter als die Anwendung dieses Filters auf jeden einzelnen Bildpunkt. Im Fall des Gauß-Filters ist dies ein gewichteter Mittelwert einer 3×3-Umgebung jedes Punktes.

Ein Convolutional Layer beruht auf der Anwendung mehrerer Faltungen auf die Eingabedaten, wobei die Filtermasken mit zufälligen Werten initialisiert und dann trainiert werden. Die Parameter dieser Filtermasken treten damit an die Stelle der Gewichte der Eingaben – in der Bilderkennung bedeutet dies, dass nicht für jedes einzelne Eingabepixel ein Gewicht erlernt werden muss, sondern nur für die Pixel der Filtermaske. Dies erlaubt es, bestimmte Features an jeder Position eines Bildes zu erkennen, da nicht jedes Eingabeneuron (an jeder Position) für jedes Feature trainiert werden muss, sondern schlicht die Filtermaske an jeder Position ausgewertet wird. Da somit jeder Parameter mehrfach verwendet wird, sind CNNs eine Anwendung von Parameter Sharing. Dadurch wird die Anzahl der erforderlichen Parameter drastisch reduziert, was komplexere Objekterkenner ermöglicht und den Erfolg der CNNs in der Bilderkennung erklärt [11]. Analog zu gewöhnlichen künstlichen Neuronen wird auf die Filter ein Bias-Wert addiert und anschließend eine Aktivierungsfunktion ausgeführt. Die so erhaltenen Ausgaben – z. B. die gefilterten Bilder – werden als *Feature Maps* bezeichnet.

Wird die Filtermaske über das Bild „geschoben“, so kann diese nur ausgewertet werden, wenn es an jeder Position der Filtermaske einen korrespondierenden Punkt im Eingabebild gibt. Dadurch schrumpft das Ausgabebild um einen Pixel weniger als die Abmessungen der Filtermaske, d. h. die Anwendung eines 3×3-Filters auf ein 500×500-Eingabebild ergibt ein 498×498-Ausgabebild[19]. Diese Form der Convolution wird als „valid“ bezeichnet. Da dies insbesondere bei großen Filtermasken und bei einer größeren Anzahl verketteter Convolutional Layers zu sehr kleinen Feature Maps führen kann, besteht auch die Möglichkeit, das Eingabebild an den Rändern mit Nullen aufzufüllen, damit die Größe der Ausgabe der Größe der Eingabe entspricht. Dieser als *Padding* bezeichnete Vorgang führt zur sogenannten *„same“ Convolution*.

Textdaten hingegen sind im Gegensatz zu Bildern eindimensional, wodurch sich die Berechnung der Convolution vereinfacht; in der Praxis wird jedoch ein Satzmatrixmodell verwendet, wodurch sich wiederum zwei Dimensionen – Zeit und Embedding – ergeben. Dennoch wird in Bibliotheken wie TensorFlow diese Form der Convolution als eindimensional bezeichnet, da die Filtermaske nur entlang der zeitlichen Dimension bewegt wird und sich zu jedem Zeitpunkt über alle E Einträge der gesamten Embedding-Dimension erstreckt (was wiederum zur Folge hat, dass die Breite der Ausgabe in dieser Dimension auf 1 sinkt). Die Formel der Kreuzkorrelation vereinfacht sich damit zu:

$$O(i) = \sum_{m=1}^{m_{\max}} \sum_{n=1}^{E} I(i+m, n) \cdot F(m,n)$$

Durch Addition des Bias-Wertes b und Anwendung einer Aktivierungsfunktion f (z. B. ReLU) zur Nichtlinearisierung ergibt sich die Formel:

$$O(i) = f\left(b + \sum_{m=1}^{m_{\max}} \sum_{n=1}^{E} I(i+m, n) \cdot F(m,n)\right) \tag{2}$$

[17] Im Fall der tatsächlichen Faltung im eigentlichen Sinne wird F spiegelverkehrt ausgewertet; TensorFlow verwendet jedoch ebenfalls die Kreuzkorrelation, weshalb im Rahmen dieses Berichts mit dem Begriff „Faltung“ (bzw. „Convolution“) stets die Kreuzkorrelation bezeichnet wird.

[18] In der Tat stellen Convolutional Neural Networks den Stand der Technik bei Bildklassifikationsproblemen dar, sodass sich viele Veröffentlichungen auf Bilder beziehen.

[19] Ein weiteres anschauliches Beispiel wäre ein 3×3-Eingabebild, auf dem der 3×3-Filter nur an genau einer Stelle ausgewertet werden kann und dann ein 1×1-Ausgabebild ergibt

Ein neuronales Netz, das diese Form der Convolution verwendet, wird als *Time Delay Neural Network* (TDNN) bezeichnet und ist in Abb. 2 schematisch dargestellt. Da in der Verarbeitung von Textdaten nur diese *1D-Convolution* relevant ist, ergibt sich eine vereinfachte Schreibweise: Statt die Embedding-Dimension explizit anzugeben und etwa von einer „3×100-Convolution“ zu sprechen, wird dies in diesem Bericht zu „3-Convolution“ verkürzt, wobei sich die angegebene Filtergröße auf die Zeitachse bezieht und immer die volle Embedding-Dimension verwendet wird.

Wie von Goodfellow et al. [12] beobachtet wurde, ist es sinnvoll, nicht nur einen einzelnen, sondern mehrere Convolutional Layer in einem tiefen neuronalen Netzwerk zu verwenden. Bei der Bilderkennung konnten tiefere Strukturen Genauigkeiten erzielen, die nicht allein aus der höheren Anzahl an Parametern resultieren. Dazu muss die Ausgabe eines Convolutional Layers als Eingabe eines anderen Convolutional Layers dienen, wobei auf die geänderte Dimensionalität eingegangen werden muss. Die Reduktion der Embedding-Dimension auf eine Breite von 1 ist zwar unbedenklich, doch kommt eine weitere Dimension – nämlich die der Feature-Maps (bzw. Channels) – dazu.

In TensorFlow wird dies gelöst, indem die Filtermasken dreidimensional sind, sich also über die Zeit-, Embedding- und Channel-Dimension erstrecken. Zwar lässt sich die Kreuzkorrelation für eine beliebige Anzahl an Dimensionen definieren, doch ist diese Verallgemeinerung im Fall des TDNN gar nicht nötig; da die Embedding-Dimension auf 1 reduziert wird, bedeutet dies für die Dimensionalität der Convolution:

$$O : \mathbb{R}^{|S| \times E \times 1} \to \mathbb{R}^{(|S| - m_{\max} + 1) \times 1 \times c}$$

Werden die letzten beiden Achsen dieser Ausgabe vertauscht, ergibt sich wieder eine Ausgabe der Form $\mathbb{R}^{x \times y \times 1}$, die problemlos mit der zuvor definierten Convolution-Operation verarbeitet werden kann. Zwar führt TensorFlow diese Vertauschung nicht durch, doch da der Filter der Convolution-Operation nach wie vor nur in Richtung der ersten Achse, d. h. der zeitlichen Dimension, bewegt wird, ist dies äquivalent zur Convolution mit vorheriger Vertauschung. In den in diesem Bericht behandelten Modellen ist immer jeweils eine dieser Dimensionen 1, sodass der allgemeine Fall der 3D-Convolution nicht behandelt werden muss.

3.2.1 Pooling

Werden mehrere Convolutional Layer in der Tiefe verknüpft, so wird von einer Schicht zur nächsten oft die *zeitliche Auflösung* reduziert, indem die Daten zwischen den Schichten herunterskaliert werden. Diese Vorgehensweise wurde in CNNs zur Bilderkennung etabliert und dient dort der Extraktion „höherstufiger“ Features. Ein einfaches Beispiel wäre etwa das Anwenden einer 3×3-Convolution auf das Eingabebild. Diese Filter sind sehr klein und können daher in der Regel kei-

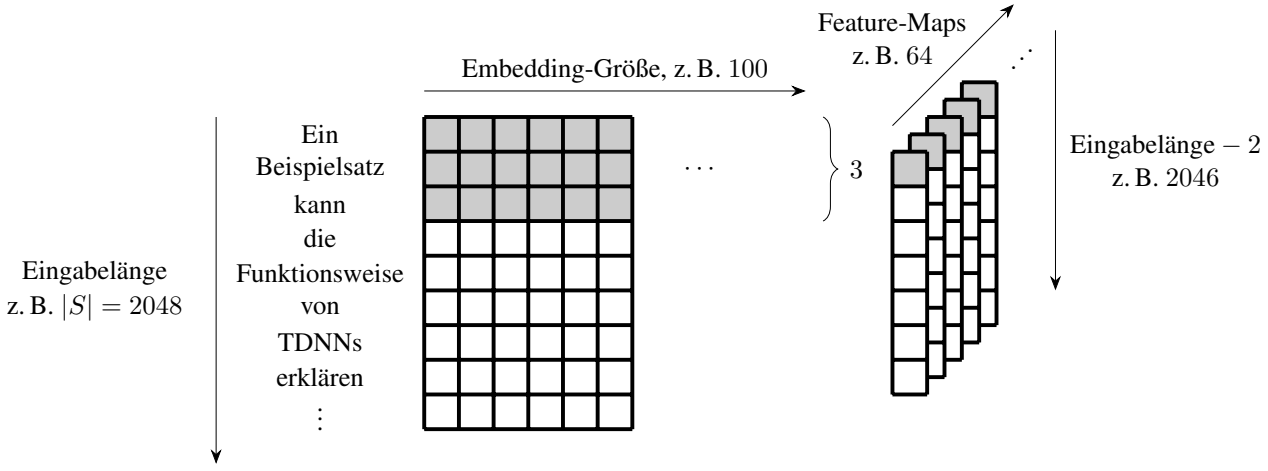


Abbildung 2: Funktionsweise eines TD-Layers ohne Padding. Die Größe der Filtermaske beträgt 3×100 ($m_{\max}$ und E aus Gleichung 2). Die Eingabe hat die Größe $2048 \times 100 \times 1$ (wobei die letzte Dimension die bisher vernachlässigte *Channel*-Dimension ist), die Ausgabe die Größe $2046 \times 1 \times 64$. Die *Breite* – d. h. die Embedding-Dimension – wird durch die Operation auf 1 reduziert, während die unterschiedlichen Feature Maps in der *Tiefe* – der Channel-Dimension – „gestapelt“ werden.

ne vollständigen Objekte erkennen. Stattdessen werden etwa Features wie Kanten, Ecken oder feinere Strukturen erkannt – sogenannte *Low-Level-Features*. Werden nun die resultierenden Feature Maps herunterskaliert (etwa um die Hälfte in jeder Dimension), so kann eine erneute 3×3-Convolution einen verhältnismäßig größeren Bereich abdecken und somit gröbere Strukturen erkennen. Dieser Vorgang kann wiederholt werden, bis Objekte einer gewünschten Größe erkannt werden können.

Diese Operation wird im Zusammenhang mit CNNs *Pooling* genannt. Die intuitive Art, ein Bild in Breite und Höhe auf jeweils die Hälfte herunterzuskalieren, besteht darin, jeweils den Mittelwert aus 2×2 Pixeln zu berechnen und diese zu einem Pixel zusammenzufassen. Diese Art des Poolings wird als *Average-Pooling* bezeichnet. In vielen Anwendungen erzielt das sogenannte *Max-Pooling* bessere Ergebnisse, bei dem nicht der Mittelwert, sondern das Maximum der beteiligten Pixel verwendet wird.

Im Gegensatz zu Skalierungsalgorithmen, die in der Bildbearbeitung verwendet werden, lassen sich Bilder mittels Pooling nur um einen ganzzahligen Faktor herunterskalieren, da – analog zur Convolution – ein „Filter“ mit fester Größe auf passende Bildbereiche angewendet wird. Wie oft dieser Filter angewendet wird, wird durch die Schrittweite, *Stride* genannt, festgelegt. Der 2×2-Filter aus dem Beispiel etwa würde mit einem Stride von 2×2 über das Bild bewegt werden, sodass jeweils ein 2×2-Bereich verarbeitet und dann der Filter um zwei Pixel verschoben wird. Wird die Filtermaske größer als der Stride gewählt, so gibt es überlappende Bereiche, sodass ein Pixel der Eingabe für mehrere Pixel der Ausgabe in Betracht gezogen wird. Für Pooling bieten sich die gleichen Padding-Optionen wie bei Convolution, woraus sich entscheidet, ob die Größe der Ausgabe auf- oder abgerundet wird[20].

So wie Convolution auf Textdaten verwendet werden kann, kann auch Pooling auf Textdaten verwendet werden. Da Pooling in der Regel auf eine Convolution folgt, ist die Eingabe ein Vektor (da die Embedding-Dimension auf 1 reduziert wurde). Jede Feature Map wird separat betrachtet, wodurch die Operation sehr einfach wird, wie in Abb. 3 dargestellt ist. Analog zur Bildverarbeitung, bei der erst Features in kleinen und dann in größeren Umgebungen betrachtet werden, kann Pooling bei Textdaten so verstanden werden, dass erst Zusammenhänge zwischen unmittelbar benachbarten Wörtern und anschließend zwischen weiter auseinanderliegenden Wörtern betrachtet werden. Als Extremfälle können die einzelnen Varianten der *Very Deep Convolutional Networks for Text Classification* (VDCNN) von Conneau et al. betrachtet werden, bei dem nicht Wörter, sondern einzelne Zeichen als Eingabe dienen und ausschließlich Filter der Größe 3 verwendet werden. Durch Pooling zwischen den einzelnen Schichten werden auf diese Weise immer größere Zusammenhänge erlernt, sodass die Erkennungsrate dieser Netzwerke mit der anderer aktueller Modelle vergleichbar ist [14].

Ein Sonderfall des Max-Poolings ist das Max-Pooling entlang der Zeitachse und damit über *alle* Eingaben einer Feature Map (da diese durch den Wegfall der Embedding-Dimension lediglich ein Vektor ist). Dadurch wird die zeitliche Auflösung auf 1 reduziert, sodass je Feature Map genau ein Wert – das *wichtigste* Feature – erhalten bleibt. Dieser als *Max-Over-Time-Pooling* bezeichnete Schritt liefert daher einen Vektor, der je Feature Map einen Eintrag hat, und wird in vielen Klassifikationsmodellen unmittelbar vor dem letzten Fully Connected Layer, der die Klassenwahrscheinlichkeiten berechnet, durchgeführt [15]. Da die Klassifikation letzten Endes nur anhand dieses Vektors erfolgt, muss dieser alle dafür notwendigen Informationen enthalten; analog zu Word Embeddings, die

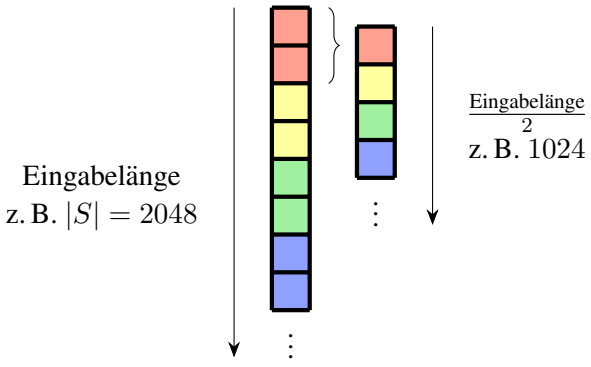


Abbildung 3: Pooling in TDNNs. Beim Max-Pooling wird aus jedem Paar die höchstwertige Eingabe ausgewählt, beim Average-Pooling hingegen der Mittelwert aus diesen gebildet. Sind mehrere Feature Maps vorhanden, so wird das Pooling für jede Feature Map separat durchgeführt, weswegen diese hier nicht aufgeführt sind.

[20] Aufgrund der vielen Gemeinsamkeiten zwischen Convolution und Pooling ist es auch möglich, Convolution mit Stride > 1 anstelle von Pooling zu verwenden, wie etwa im *All Convolutional Net* von Springenberg et al. [13].

die Bedeutung eines Wortes erfassen, kann dieser Vektor als Repräsentation der Bedeutung des Satzes – die *Satzeinbettung* – betrachtet werden. Da das Max-Over-Time-Pooling den Satz damit als einen Vektor *kodiert*, wird der Pooling-Layer in diesem Fall auch als *Encoder* bezeichnet.

Eine Verallgemeinerung des Max-Poolings stellt das *k-Max-Pooling* dar, bei dem nicht genau ein Feature, sondern die k größten Features pro Feature Map extrahiert werden. Zu beachten ist, dass die so extrahierten Features nicht nach ihrem Wert sortiert werden, sondern ihre relative Reihenfolge aus den Eingabedaten beibehalten, wie in Abb. 4 verdeutlicht wird [16].

3.3 Recurrent Neural Networks

Eine weitere Variante der neuronalen Netze sind die sogenannten *Recurrent Neural Networks* (RNN). Diese bieten – wie auch CNNs – die Möglichkeit, die gleichen Parameter mehrfach anzuwenden und somit die Anzahl der Parameter reduzieren, sind allerdings darauf spezialisiert, *sequentielle* Daten zu verarbeiten. In jedes Neuron bzw. jede *Einheit* (*Hidden Unit*) einer rekurrenten Schicht wird je ein Wert der Eingabeschicht – etwa ein Wort –, sowie zusätzlich eine Ausgabe (der *Zustand*) aller Neuronen, die das vorherige Element der Sequenz (das vorherige Wort) verarbeitet haben, eingegeben. Dies hat zur Folge, dass im Gegensatz zu CNNs nicht eine feste Nachbarschaft eines Eingabewertes, sondern der gesamte Verlauf der Sequenz bis zum Erreichen dieses Wertes betrachtet wird – ein rekurrentes neuronales Netz hat also ein „Gedächtnis“.

Die Ausgabe einer RNN-Schicht (auch *Zelle* genannt) zum Zeitpunkt t lässt sich folgendermaßen formulieren:

$$O(t) = f\left(W \cdot [x_t \oplus O(t-1)] + b\right) \tag{3}$$

Dabei stellt $x_t \oplus O(t-1)$ die Konkatenation des aktuellen Eingabewort-Embeddings x_t mit der Ausgabe zum vorherigen Zeitschritt $O(t-1)$ dar. W ist die Gewichtsmatrix, b der Bias-Vektor und f die Aktivierungsfunktion, z. B. $\tanh$ oder ReLU[21].

Das *Parameter Sharing* wird in RNNs dadurch realisiert, dass jede Einheit für diese Operationen die gleichen Parameter W und b verwendet. Analog zu CNNs, in denen eine Filtermaske über die Eingabe „geschoben“ wird, wird also auch bei RNNs eine Gewichtsmatrix an jeder Stelle der Eingabe ausgewertet, wobei jedoch stets das Ergebnis der Auswertung der vorherigen Stelle in die Berechnung einfließt. Dadurch enthält jede Ausgabe Informationen über alle bisherigen Wörter, was zur Folge hat, dass die letzte Ausgabe $O(|S|)$ Informationen über den gesamten Satz enthält und damit eine Satzeinbettung ist. Abb. 5 verdeutlicht diese Zusammenhänge.

[21] In einigen Veröffentlichungen [7] wird zwischen Zustand und Ausgabe einer rekurrenten Schicht unterschieden. Dabei ist der *Zustand* das Ergebnis der in Gleichung 3 als $O(t)$ bezeichneten Funktion und wird an das nächste Neuron der *gleichen* Schicht übergeben. Die *Ausgabe* wird dann als $P(t) = g(V \cdot O(t) + c)$ berechnet und an die *nächste* Schicht übergeben. Dies entspricht jedoch schlicht der Anwendung eines weiteren Fully-Connected-Layers auf den Zustand $O(t)$, sodass in diesem Bericht die Begriffe *Zustand* und *Ausgabe* im Kontext rekurrenter Einheiten synonym verwendet werden.

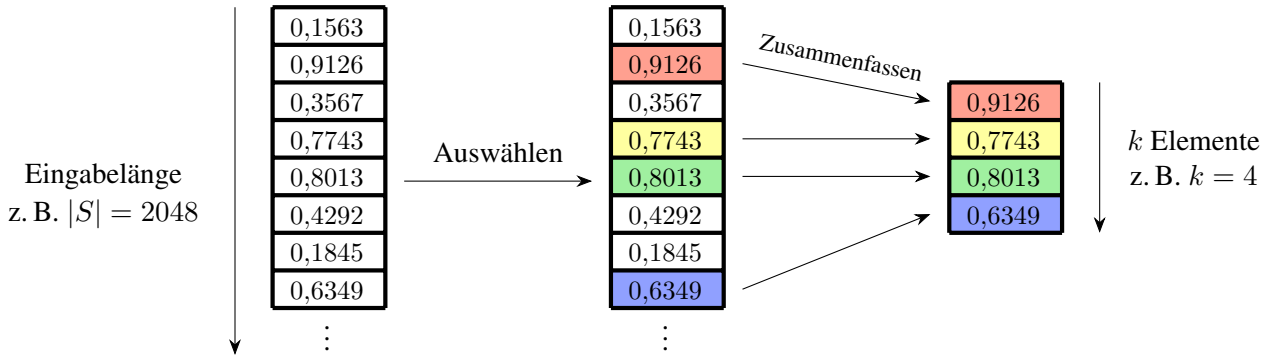


Abbildung 4: k-Max-Pooling in TDNNs. Da die Reihenfolge der Elemente erhalten bleibt, kann dies so verstanden werden, dass die zeitliche Auflösung nicht um einen bestimmten Skalierungsfaktor, sondern auf einen festen Wert k reduziert wird. Analog zum klassischen Max-Pooling bleiben dabei nur die wichtigsten Features erhalten, wobei beim k-Max-Pooling nicht je ein Feature in einem bestimmten Block, sondern k Features in der gesamten Eingabe ausgewählt werden.

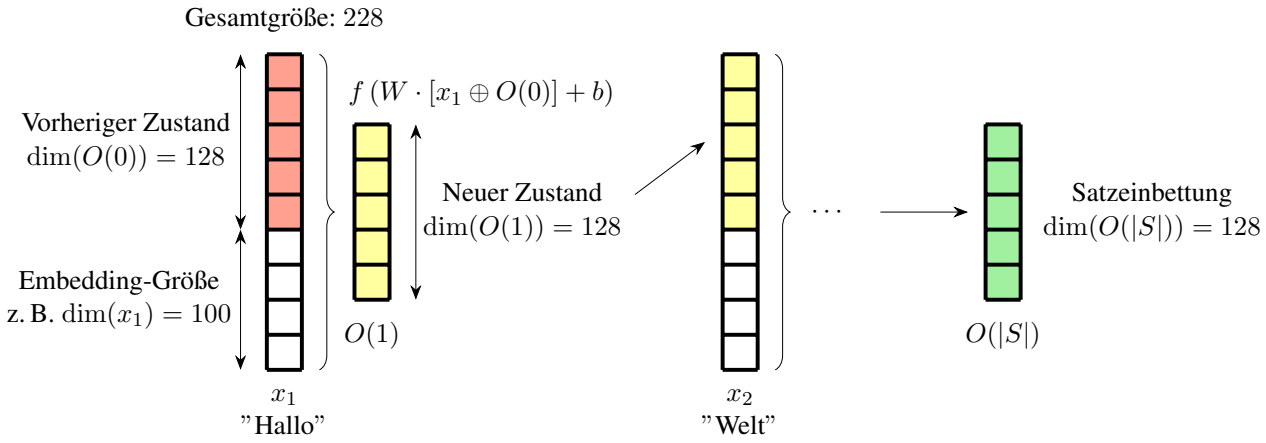


Abbildung 5: Funktionsweise eines RNN-Layers. Die Eingaben $x_1, x_2, \ldots, x_{|S|}$ werden jeweils mit der vorherigen Ausgabe $O(0), O(1), \ldots O(|S| - 1)$ verkettet und dann zur nächsten Ausgabe verrechnet. $O(0)$ ist ein Nullvektor (da es keine vorherige Ausgabe gibt, mit der x_1 verkettet werden kann) und die letzte Ausgabe $O(|S|)$ enthält die Satzeinbettung, die für die Klassifikation verwendet werden kann.

Zu beachten ist, dass nicht notwendigerweise die *letzte* Ausgabe benutzt werden muss. $|S|$ bezeichnet die *maximale* Satzlänge, und im Allgemeinen werden kürzere Sätze mit Nullvektoren aufgefüllt, um diese Länge zu erreichen. Da diese Nullvektoren keine weiteren Informationen enthalten, ist es meist nicht nötig, die Eingabe bis $x_{|S|}$ auszuwerten. Um die endgültige Satzeinbettung zu erhalten, genügt es daher, die Eingabe bis zum letzten Nicht-Nullvektor auszuwerten.

Das hier vorgestellte, grundlegende RNN-Modell hat die Schwäche, dass jede Ausgabe nur Informationen der vorherigen Wörter, also der Vergangenheit, nicht jedoch Informationen der nächsten Wörter, also der Zukunft, betrachtet. Doch oft sind die folgenden Wörter ebenfalls wichtig für die Bedeutung eines Wortes. Dieser Schwäche kann auf verschiedene Arten entgegengewirkt werden, von denen zwei bereits vorgestellt wurden:

1) Die Eingabe kann einen *Convolutional Layer* durchlaufen. Die Ausgaben dieser Schicht enthalten dann jeweils Informationen über eine feste Nachbarschaft der Eingaben, sodass der lokale Kontext erfasst wird. Diese können dann in ein RNN eingegeben werden, das wiederum den globalen Kontext ermittelt. Die daraus resultierende Satzeinbettung berücksichtigt die vollständige Vergangenheit sowie eine endlich große Zukunft (deren Größe von den Filtermasken der Convolution abhängt) der Eingabewörter. Dieses Modell ist als *CNN-RNN* oder *CRNN* bekannt und kann in der Spracherkennung eingesetzt werden, da dort der vollständige zukünftige Kontext nicht bekannt ist, aber eine kleine Menge an Eingaben gepuffert werden kann, um einen lokalen Kontext zu erhalten [17].

2) Es kann ein zusätzlicher *Recurrent Layer* benutzt werden, der die Eingabe von hinten nach vorne durchläuft. Beide RNN-Schichten sind voneinander unabhängig, arbeiten also auf der gleichen Eingabe und beziehen die Ergebnisse des jeweils anderen RNNs *nicht* mit ein; sie stehen also in der Hierarchie des neuronalen Netzes *nebeneinander*. Die Ausgabe $O_{\text{fw}}(t)$ des Vorwärts-RNNs und die Ausgabe des Rückwärts-RNNs $O_{\text{bw}}(t)$ werden dann wiederum verkettet, sodass sich zu jedem Zeitpunkt t der Vektor $O(t) = O_{\text{fw}}(t) \oplus O_{\text{bw}}(t)$ ergibt, der dann den globalen Kontext in beiden Richtungen enthält. Diese Variante wird daher treffenderweise als *bidirektionales RNN* bezeichnet. Um diese verketteten Zustände auswerten, kann wieder ein Encoder, wie etwa Max-Over-Time-Pooling, ein weiteres eindimensionales RNN oder der im nächsten Abschnitt erklärte Attention-Mechanismus eingesetzt werden.

3.3.1 Attention

Attention („Aufmerksamkeit“) ist eine Methode zur Berechnung einer Satzeinbettung, bei der ein gewichteter Mittelwert $\vec{v}$ von i Vektoren $\vec{h}_i$ – etwa die Zustände eines bidirektionalen RNN – gebildet wird. Es gibt

verschiedene Varianten, wobei die in diesem Bericht verwendete Variante derjenigen entspricht, die im *Hierarchical Attention Network* (HAN) von Yang et al. [18] Anwendung findet. Diese wird wie folgt berechnet:

$$\vec{u}_i = f\left(W \cdot \vec{h}_i + \vec{b}\right)$$

$$\alpha_i = \frac{\exp\left(\vec{u}_i^{\mathsf{T}} \cdot \vec{u}\right)}{\sum_i \exp\left(\vec{u}_i^{\mathsf{T}} \cdot \vec{u}\right)}$$

$$\vec{v} = \sum_i \alpha_i \cdot \vec{h}_i$$

W, $\vec{u}$ und $\vec{b}$ sind trainierbare Gewichte, deren Größe beliebig gewählt werden kann (im HAN ist $\dim(\vec{u}) = 100$), und f ist eine Aktivierungsfunktion (z. B. $\tanh$). Jeder Vektor $\vec{h}_i$ durchläuft damit einen Fully-Connected-Layer, um das Ergebnis $\vec{u}_i$ zu erhalten. Anschließend wird das Skalarprodukt aus $\vec{u}_i$ und dem *Kontextvektor* $\vec{u}$ berechnet. Je ähnlicher ein Vektor $\vec{u}_i$ also zum Kontextvektor $\vec{u}$ ist, desto stärker wird er gewichtet. Auf diese *unnormalisierten* Gewichte wird dann die *Softmax*-Funktion angewendet, um die normalisierten Gewichte α_i zu erhalten, mit denen die $\vec{h}_i$ multipliziert werden, um den gewichteten Mittelwert $\vec{v}$ zu erhalten.

Die Bezeichnung „Attention“ erklärt sich damit, dass der Mechanismus lernt, wichtige Zustände (z. B. Wörter eines Satzes) zu erkennen und seine „Aufmerksamkeit“ darauf zu richten, während weniger wichtige Zustände sehr geringe Beachtung finden. Im Gegensatz zum Max-Over-Time-Pooling, das für jede Dimension des Vektorraums jeweils den höchsten Wert extrahiert, werden die Vektoren dabei als Ganzes gewichtet.

3.3.2 LSTM und GRU

Da rekurrente Netze nur die aktuelle Eingabe und den vorherigen Zustand berücksichtigen und darauf stets die gleiche Parametermenge anwenden, wird der Einfluss von Zuständen geringer, je weiter diese in der Vergangenheit liegen. Dadurch „vergisst“ das RNN, und der eigentliche Vorteil, nämlich das Erfassen eines globalen Kontexts, geht dabei verloren. Dieses Problem kommt auch beim Training zum Tragen: Rekurrente Netze werden mit dem *Backpropagation-Through-Time*-Verfahren (BPTT) trainiert, bei dem eine rekurrente Schicht als ein sehr tiefes Netz (mit jedem Zeitschritt als eine einzelne Schicht) betrachtet und dann wie ein gewöhnliches neuronales Netz trainiert wird. Da dieses Netz jedoch mehrere hundert bis tausend Schichten hat, tritt dabei das *Vanishing-Gradient-* oder *Exploding-Gradient*-Problem auf: Durch das wiederholte Ableiten wird der Gradient immer kleiner bzw. immer größer, sodass die Schichten nahe der Eingabe (bzw. nahe des ersten Zeitschrittes) keine sinnvollen Werte mehr lernen können.

Um diesen Problemen entgegenzuwirken, wurden verschiedene Erweiterungen des RNN-Konzepts entwickelt, von denen zwei in diesem Bericht verwendet werden: *Long-Short-Term-Memory* (LSTM, „langes Kurzzeitgedächtnis“) und *Gated Recurrent Units* (GRU).

LSTM. Die LSTM-Architektur wurde erstmals 1997 von Hochreiter und Schmidhuber [19] vorgestellt und erweitert die RNN-Einheit um *Gates* („Gatter“), mit denen gesteuert wird, wie viele Informationen aus Eingabevektor und vorherigem Zustand in den aktuellen Zustand einfließen. Zusätzlich wird explizit zwischen Zustand und Ausgabe unterschieden, wobei auch hier wieder gesteuert wird, welche Informationen aus dem Zustand in die Ausgabe übernommen werden.

In TensorFlow (das sich im Wesentlichen an der Variante von Zaremba et al. [20] orientiert) wird die grundlegende RNN-Formel 3 dabei wie folgt erweitert:

$$\begin{aligned}
f_t &= W_f \cdot [x_t \oplus O(t-1)] + b_f \\
i_t &= W_i \cdot [x_t \oplus O(t-1)] + b_i \\
j_t &= W_j \cdot [x_t \oplus O(t-1)] + b_j \\
o_t &= W_o \cdot [x_t \oplus O(t-1)] + b_o \\
c_t &= c_{t-1} \circ \operatorname{sigm}(f_t + b_1) + f(j_t) \circ \operatorname{sigm}(i_t) \\
O(t) &= f(c_t) \circ \operatorname{sigm}(o_t)
\end{aligned}$$

$O(t)$ und $O(t-1)$ sind die Ausgaben des aktuellen bzw. vorherigen Zeitschrittes; x_t ist der aktuelle Eingabevektor und f die Aktivierungsfunktion. Die Werte i_t, j_t, f_t und o_t berechnen sich über jeweils eigene Gewichtsmatrizen und Bias-Werte aus der Konkatenation von x_t und $O(t-1)$; die Dimensionalität des resultierenden Vektors entspricht der Anzahl der Hidden Units.

f_t ist das *Forget Gate*. Dieser Wert bestimmt, welche Informationen des vorherigen Zustands c_{t-1} in den

neuen Zustand übergehen. In TensorFlow wird auf f_t zusätzlich ein – im Gegensatz zu b_i konstanter und nicht trainierbarer – Wert b_1 addiert, der *Forget Bias*; sofern nicht anders angegeben, ist $b_1 = 1$. Durch Anwendung der logistischen Funktion (sigm, da es sich um eine Sigmoidfunktion handelt) werden die Werte des Vektors komponentenweise auf das Intervall $(0; 1)$ normalisiert. Jede Komponente dieses Vektors wird dann mit der zugehörigen Komponente in c_{t-1} multipliziert (es ergibt sich das Hadamard-Produkt, $\circ$), um diese zu gewichten. Analog dazu wird das *Input Gate* i_t verwendet, um $f(j_t)$ – also die mittels eines Fully-Connected-Layers transformierten Eingaben – zu gewichten. Diese werden dann auf das Zwischenergebnis addiert, um den neuen Zustand c_t zu erhalten.

Der neue Zustand dient dann wieder als Eingabe für die Aktivierungsfunktion f und wird komponentenweise mit den normalisierten Werten des *Output Gates* o_t multipliziert, um die neue Ausgabe $O(t)$ zu erhalten. Indem die passenden Gewichte für die Gatter erlernt werden, kann der LSTM-Mechanismus sowohl Zusammenhänge über große als auch über kleine Zeiträume betrachten.

GRU. Die Gated Recurrent Units (GRU) wurden 2014 von Cho et al. [21] vorgestellt und verfolgen ein ähnliches Konzept, sind jedoch einfacher aufgebaut. Die konkrete Implementation, die in TensorFlow verwendet wird, erfolgt dabei nach folgender Formel:

$$
\begin{aligned}
r_t &= \mathrm{sigm}\left(W_r \cdot [x_t \oplus O(t-1)] + b_r\right) \\
u_t &= \mathrm{sigm}\left(W_u \cdot [x_t \oplus O(t-1)] + b_u\right) \\
O(t) &= O(t-1) \circ u_t + (1 - u_t) \\
&\quad \circ f\left(W_c \cdot [x_t \oplus (O(t-1) \circ r_t)] + b_c\right)
\end{aligned}
$$

Der deutlichste Unterschied zum LSTM-Modell ist, dass nicht zwischen Zustand und Ausgabe unterschieden wird, sodass die Berechnung weniger Variablen erfordert. r_t ist das *Reset Gate*, u_t das *Update Gate*, die exakt wie die Gatter des LSTM-Modells berechnet werden, nämlich über ein Matrix-Vektor-Produkt mit logistischer Aktivierungsfunktion. u_t bestimmt dabei direkt, welche Informationen der vorherigen Ausgabe $O(t-1)$ übernommen werden. Das Komplement dieser Gewichtsverteilung wiederum wird verwendet, um die neuen Eingaben zu gewichten. Diese sind das übliche Matrix-Vektor-Produkt mit Aktivierungsfunktion f, wobei die Ausgabe des letzten Zeitschrittes $O(t-1)$ – nicht jedoch die aktuelle Eingabe x_t – mit r_t gewichtet und deswegen nur teilweise übernommen wird, was der GRU-Zelle erlaubt, den Zustand „zurückzusetzen“ und hauptsächlich die neue Eingabe zu betrachten.

3.4 Regularisation

Um Overfitting entgegenzuwirken, wurden verschiedene Verfahren entwickelt, die in ihrer Gesamtheit als *Regularisation* bezeichnet werden. Diese Methoden unterscheiden sich stark in ihrer Funktionsweise und können daher auch in der Regel kombiniert werden.

3.4.1 Dropout

Ein *Dropout*-Layer ist eine zusätzliche Schicht, die einen zufälligen Teil der Eingaben auf 0 setzt und dann ausgibt. Der Teil der Eingaben, der beibehalten wird, wird dabei beim *Training* über eine Wahrscheinlichkeit p_{keep} bestimmt, in diesem Bericht meist $p_{\text{keep}} = 0{,}5$. Die beibehaltenen Ausgaben werden dabei um den Faktor $\frac{1}{p_{\text{keep}}}$ skaliert, um den Erwartungswert der Summe beizubehalten. Zur Inferenzzeit, d. h. bei der Klassifikation, wird $p_{\text{keep}} = 1$ gesetzt.

Dropout kann so interpretiert werden, dass bei einem Netz mit n Parametern nicht nur ein Netz, sondern 2^n verschiedene Netze trainiert werden (also jede mögliche Kombination aus beibehaltenen und nicht beibehaltenen Werten). Zur Inferenzzeit wird dann ein Mittelwert dieser Netze betrachtet. Dadurch kann sich keines der Netze zu stark auf die Trainingsdaten spezialisieren, sodass der Fehler beim Testen reduziert wird [22].

Dropout kann an jeder Stelle eines Netzes eingesetzt werden, wobei bei der Anwendung auf die Eingabeschicht ein höherer Wert für p_{keep} empfohlen wird [23]. Das *Deep Averaging Network* (DAN) verwendet eine spezielle Form des Dropouts, bei dem nicht nur einzelne Werte, sondern Word Embeddings als Ganzes auf 0 gesetzt werden. Dabei wurden mit $p_{\text{keep}} = 0{,}7$ die besten Ergebnisse erzielt [24].

3.4.2 L1/L2-Loss

Eine weitere Methode zur Regularisation besteht darin, ein Netz beim Training zu „bestrafen“, wenn es zu

spezialisiert auf die Trainingsdaten ist. Dazu kann die Kostenfunktion erweitert werden, sodass ein Regularisationsterm darauf addiert wird. Dieser Term besteht aus der Betragsnorm (1-Norm) oder der euklidischen Norm (2-Norm) der Gewichte, die dann um einen einstellbaren Faktor λ skaliert werden. Eine Regularisation dieser Art hat zur Folge, dass weniger extreme Gewichte gelernt werden; insbesondere bei der euklidischen Norm werden Ausreißer sehr stark gewichtet. Diese Methode wird *L1-Loss* bzw. *L2-Loss* genannt.

Die modifizierte Kostenfunktion kann wie folgt beschrieben werden:

$$L_{\mathrm{L1}}(\hat{y}, y) = L(\hat{y}, y) + \lambda \cdot \sum_{W} \|W\|_1$$

Der L1-Loss kann auf jede beliebige Gewichtsmatrix W angewendet werden. In der Regel werden die Bias-Werte b jedoch *nicht* gewichtet.

Für den L2-Loss verfügt TensorFlow über eine etwas abgewandelte Version, die auf die rechenintensive Quadratwurzel verzichtet:

$$L_{\mathrm{L1}}(\hat{y}, y) = L(\hat{y}, y) + \frac{\lambda}{2} \cdot \sum_{W} \sum_{w \in W} w^2$$

Wichtig ist, den Parameter λ nicht zu hoch zu wählen, da sonst der Anteil der tatsächlichen Kostenfunktion im Verhältnis zur 1-/2-Norm verschwindend gering wird und das Training nahezu ausschließlich die Norm der Gewichte anstatt der tatsächlichen Kosten minimiert.

3.4.3 Early Stopping

Early Stopping bezeichnet Techniken, deren Grundprinzip darin besteht, das Training zu beenden, sobald Overfitting eintritt. Für diesen Bericht habe ich das sogenannte *Validation-based early stopping* angepasst, um das Training nicht nur dynamisch verkürzen, sondern auch ggf. verlängern zu können, wenn das Modell noch nicht vollständig trainiert ist. Dazu wird zunächst eine reguläre Trainingsdauer T_{max} festgelegt. Diese ist in der Regel ein kleiner Wert, etwa $4 \leq T_{\mathrm{max}} \leq 8$. Während des Trainings wird das Modell regelmäßig (z. B. nach jeweils 50 Minibatches) auf dem *Dev-Set* ausgewertet. Ist die dabei erzielte Genauigkeit höher als der bisher gemessene Maximalwert, so werden die Gewichte dieses Modells gespeichert. Anschließend wird die maximale Trainingsdauer angepasst:

$$T_{\mathrm{max}} = \max\left(\lceil T_{\mathrm{cur}} \cdot 1{,}5 \rceil; T_{\mathrm{max}}\right)$$

T_{cur} ist dabei die bei 0 startende aktuelle Trainingsepoche. Wird zu lange kein besseres Modell gefunden, so wird irgendwann die maximale Trainingsdauer erreicht und das Training beendet. Anschließend werden die Parameter des besten Modells geladen und auf dem *Test-Set* ausgewertet, um die Genauigkeit des Modells zu erhalten.

Das ständige Erhöhen der Trainingsdauer verhindert, dass das Training durch einen zu klein eingestellten Wert beendet wird, bevor Overfitting eintritt, d. h. so lange sich die Genauigkeit noch verbessert. Dies ist insbesondere wichtig, da unterschiedliche Architekturen teilweise sehr unterschiedliche Trainingsdauern haben und sich T_{max} vorher schwer abschätzen lässt, wenn noch keine vergleichbaren Architekturen getestet wurden. Dieses Verfahren verhindert also nicht nur Overfitting, sondern fügt auch eine gewisse Robustheit bezüglich der Trainingsdauer hinzu, sofern der initiale Wert nicht zu klein gewählt wird. Der Faktor $1{,}5$ wurde empirisch während der Experimentierphase (Abschnitt 5) bestimmt.

3.4.4 Batch Normalization

Batch Normalization ist ein Verfahren zur Normalisierung der Features eines Minibatches. Dabei wird während des Trainings jedes *Feature* – d. h. jede einzelne (skalare) Komponente jedes Featurevektors – skaliert, sodass der Mittelwert des Features innerhalb des Minibatches 0 und die Varianz 1 ist. Da zur Inferenzzeit hingegen in der Regel keine Minibatches vorliegen (sondern einzelne Dokumente klassifiziert werden), werden die Features anhand während des Trainings bestimmter gewichteter Mittelwerte der Feature-Mittelwerte und Varianzen skaliert. Dieses Verfahren kann das Training beschleunigen und zudem zur Regularisation beitragen [25].

Wird Batch Normalization mit Dropout kombiniert, so ergeben sich oft schlechtere Ergebnisse, als wenn jede dieser Techniken einzeln verwendet wird. Diesem Verhalten kann entgegengewirkt werden, indem alle Dropout-Schichten nach allen Batch-Normalization-Schichten im Netz stehen – der Datenfluss muss also

alle Batch-Normalization-Schichten durchlaufen haben, bevor Dropout angewendet werden darf [26].

4 Realisierung

Nachdem die theoretischen Grundlagen geklärt wurden, soll dieses Kapitel die konkret implementierten Klassifikationsmodelle erklären. Diese Modelle sollen nicht nur evaluiert, sondern später auch praktisch eingesetzt werden. Die Klassifikation wird dabei auf Hardware des Endanwenders durchgeführt, die unter Umständen nicht sehr leistungsstark ist. Da das Training hingegen einige Zeit in Anspruch nehmen kann (wie sich in Abschnitt 5.3 zeigen wird), ist es sinnvoll, die Modelle vorab auf einem Server zu trainieren und die trainierten Modelle dann auf der Hardware des Endanwenders einzusetzen.

Da sich diese Systeme aber in der eingesetzten Hard- und Software (was insbesondere das Betriebssystem betrifft) unterscheiden, habe ich ein Trainings- und Test-Skript in der plattformübergreifenden Skriptsprache *Python 3*[22] implementiert. Die klassischen Modelle bauen dabei auf scikit-learn auf, während für die NN-Modelle TensorFlow zum Einsatz kommt. Über die Python-Schnittstelle von TensorFlow kann ein neuronales Netz definiert werden. Die dafür verwendeten Funktionen führen nicht sofort eine Berechnung durch, sondern bauen einen Graphen auf, der einen Knoten mit der entsprechenden Operation enthält und über Kanten, die die Ein- und Ausgaben repräsentieren, mit anderen Knoten (Operationen) verbunden ist. Ist ein solcher Graph definiert, kann dieser mit Eingabedaten aufgerufen werden und gibt alle gewünschten Ausgaben zurück, wozu alle Operationen im Graphen, von denen diese Ausgaben abhängen, ausgeführt werden[23]. Dadurch kann die tatsächliche Ausführung aufwendiger Berechnungen der Implementierung von TensorFlow überlassen werden, die wiederum auf effiziente Methoden – d. h. Parallelisierung, Nutzung von CPU-Features wie SIMD[24]-Instruktionen oder aber Nutzung der GPU – zurückgreifen kann. Dies ist ein großer Vorteil, da Python als interpretierte Skriptsprache weit weniger effizient als systemnähere Sprachen ist und Parallelisierung nur auf Prozessebene – nicht jedoch auf Threadebene – möglich ist, wodurch viel Rechenleistung durch die Kommunikation verloren geht[25]. Lediglich Verwaltungsaufgaben, wie etwa das Vorbereiten der Daten und der Batches, müssen in Python implementiert werden. Für alle Modelle wird zudem die Bibliothek *NumPy*[26] verwendet, die unter anderem Datenstrukturen und Funktionen für mehrdimensionale Arrays zur Verfügung stellt, die wiederum sowohl von scikit-learn als auch von TensorFlow eingesetzt werden.

Die Testumgebung hat sowohl für klassische Modelle als auch für neuronale Netze einen ähnlichen Ablauf. Zunächst werden die *Trainingsparameter* geladen, die bestimmen, welcher Datensatz und welcher Classifier verwendet wird, und wie das verlaufen soll. Anschließend werden die *Architektur-Hyperparameter* geladen, die für jeden Classifier individuell einstellbar sind. Bei NN-Modellen sind dies zum Beispiel Anzahl und Größe der Schichten oder Aktivierungsfunktionen. Anschließend werden 90% des Datensatzes in das Train- und die restlichen Daten in das Test-Set partitioniert, wobei die relativen Klassenhäufigkeiten erhalten bleiben. Anschließend beginnt das Training, wobei klassische Modelle vollständig von sklearn trainiert werden, während das Training neuronaler Netze etwas komplexer ist, wie in Abschnitt 4.1 näher erläutert wird. Ist das Training beendet, wird das Modell auf dem Test-Set ausgewertet. Falls Kreuzvalidierung erfolgen soll, so wird anschließend ein neues Test-Set gewählt, das zum alten disjunkt ist. Die restlichen Daten sind Trainingsdaten, mit denen das Training erneut beginnt.

4.1 Neuronale Netze

Alle getesteten neuronalen Netze wurden mit gemeinsamen Mechanismen trainiert und evaluiert. Aus diesem Grund verwenden alle NN-Architekturen das gleiche

[22] `https://www.python.org/`

[23] In der Regel sind die Ausgaben die Klassifikationsergebnisse, die sich in der letzten Schicht des Graphen befinden, wodurch dieser vollständig abgearbeitet wird; es lassen sich jedoch auch beliebige Zwischenergebnisse ausgeben (z. B. die erlernten Embedding-Vektoren). Enthält der Graph Operationen, die für die Berechnung der angefragten Operationen nicht erforderlich sind, so werden diese auch nicht berechnet.

[24] *Single Instruction, Multiple Data* ist die Anwendung einer Instruktion auf einen Vektor. Bei dieser Instruktion kann es sich um *Fused Multiply-Add* (FMA) – also eine von einer Addition gefolgte Multiplikation – handeln, womit sich Matrix-Vektor-Produkte sehr schnell berechnen lassen.

[25] Dies ist auf die Referenzimplementation *CPython* bezogen. Andere Python-Implementation könnten abweichen.

[26] `http://www.numpy.org/`

Eingabeformat – nämlich das indizierte Satzmatrix-modell nach Gleichung 1 – und liefern die gleichen Ausgaben. Nachdem die Eingabedaten mit dem zuvor beschriebenen Trainings-Skript in Train- und Test-Set unterteilt wurden, wird ein stratifiziertes – d. h. die Klassenhäufigkeiten erhaltendes – Dev-Set erstellt, indem 10% der Dokumente aus dem Train-Set entfernt werden. Dieses Dev-Set wird benutzt, um die in Abschnitt 3.4.3 beschriebene Variante des Early Stopping umzusetzen. Das Train-Set wird zu Beginn jeder Epoche in eine zufällige Reihenfolge gebracht und dann in Minibatches mit je 64 Dokumenten unterteilt, um das neuronale Netz zu trainieren, wobei das *Training* im eigentlichen Sinne, d. h. das Anpassen der Gewichte, von TensorFlow übernommen wird.

4.1.1 Gemeinsame Komponenten

Obwohl die Architekturen sehr vielfältig sind, haben alle implementierten neuronalen Netze viele Komponenten gemein. So folgt auf die Eingabeschicht bei allen getesteten neuronalen Netzen ein Embedding-Layer, der die Eingaben in ihre entsprechenden Word Embeddings umwandelt. Dazu wird eine $|V| \times E$-Gewichtsmatrix W_{embed} mit zufällig initialisierten und trainierbaren Gewichten verwendet. Um daraus die Embedding-Vektoren zu erhalten, wird für den Wort-Index $\hat{s}$ die $\hat{s}$-te Zeile der Matrix W_{embed} geladen. Dies entspricht der Matrix-Vektor-Multiplikation $W_{\text{embed}}^{\mathsf{T}} \cdot s$, wobei s der zu $\hat{s}$ korrespondierende One-Hot-Vektor ist. Wie die nächste Schicht die so erhaltene Ausgabe S_{embed} verarbeitet, bleibt der jeweiligen Architektur überlassen.

Eine weitere Schicht, die allen Netzen gemein ist, ist die Ausgabeschicht. Diese verwendet einen Fully Connected Layer, um die (architekturabhängigen) Ausgaben der vorherigen Schicht in *Scores* für die einzelnen Klassen umzuwandeln. Dieser verfügt über die trainierbare $n \times m$-Gewichtsmatrix W_{scores} und den n-dimensionalen Bias-Vektor b_{scores}, wobei n die Anzahl der Klassen und $m = \dim(x)$ die Dimensionalität des Ausgabevektors x der vorherigen Schicht ist. Daraus wird zunächst der Vektor $\hat{y}_{\text{raw}} = W_{\text{scores}} \cdot x + b_{\text{scores}}$ berechnet, der die *unnormalisierten* Scores für alle Klassen enthält, d. h. die einzelnen Einträge des Vektors können beliebige Werte annehmen. Die Klasse mit dem höchsten Score wird dann als *Vorhersage* ausgegeben, und der Vergleich dieser Vorhersagen mit den tatsächlichen Klassen dient der Berechnung der *Genauigkeit* des Netzes.

Ebenfalls Teil der Ausgabeschicht ist die Berechnung der Kostenfunktion, wobei für alle getesteten Architekturen die Kreuzentropie verwendet wird. Für die Kreuzentropie muss die tatsächliche Wahrscheinlichkeitsverteilung eines Dokuments mit der vom neuronalen Netz berechneten Wahrscheinlichkeitsverteilung verglichen werden; der Wertebereich der Scores ist jedoch prinzipiell beliebig, wodurch diese in der Regel keine gültige Wahrscheinlichkeitsverteilung darstellen. Zu diesem Zweck werden die Scores mit der Softmax-Funktion normalisiert:

$$\hat{y}_{\text{softmax}} = \sigma\left(\hat{y}_{\text{raw}}\right)$$

Die normalisierten Scores $\hat{y}_{\text{softmax}}$ können dann mit der tatsächlichen Klasse y verglichen werden. Dies ist möglich, weil y ein One-Hot-Vektor ist, der die Klasse angibt, und damit ebenfalls eine gültige Wahrscheinlichkeitsverteilung ist.

Abb. 6 zeigt diesen grundlegenden Aufbau, auf den sich alle nachfolgenden Graphen – sofern nicht anders angegeben – beziehen[27].

4.1.2 Flaches und breites CNN

Eine sehr grundlegende Architektur ist das „flache und breite“ Convolutional Neural Network nach Yoon Kim [27]. Dieses verfügt über drei Convolutional Layer, die jedoch nicht in der Tiefe verschachtelt sind, sondern parallel *nebeneinander* stehen und auf den Word Embeddings der Eingabe arbeiten. Auf allen Ausgaben jedes Convolutional Layers wird dann Max-Over-Time-Pooling durchgeführt, um das jeweils wichtigste Feature für jeden Filter auszuwählen. Da in der Originalversion des Netzes jeweils 100 Filter pro Convolutional Layer verwendet werden, ergeben sich somit insgesamt 300 Werte. Diese werden dann konkateniert und passieren zunächst einen Dropout-Layer, bevor diese in den finalen Fully Connected Layer übergehen.

[27] Es ist zu beachten, dass die Graphen neuronaler Netze in vielen Veröffentlichungen mit der Eingabeschicht unten und der Ausgabeschicht oben dargestellt werden; in diesem Bericht werden jedoch alle Graphen mit der Eingabe oben und der Ausgabe unten dargestellt, da dies der natürlichen Leserichtung entspricht und der Begriff „*tiefes* neuronales Netz“ dann auch im Kontext des Graphen sinnvoll ist (das Hinzufügen weiterer Schichten erhöht die *Tiefe*, während bei umgekehrter Graphenrichtung die *Höhe* erhöht werden würde).

Abb. 7 zeigt den Graphen dieses Netzes. Die parallelen Operationen (Convolution und Pooling) werden separat dargestellt, könnten aber auch als eine gemeinsame Schicht betrachtet werden.

Die Anzahl und Größe der Filter wurde von Kim mithilfe einer Rastersuche ermittelt. Prinzipiell können diese Werte angepasst werden, um etwa noch mehr verschiedene Größen von Filtern zu verwenden; auch muss die Anzahl der Filter pro Größe nicht immer gleich sein.

4.1.3 Mehrschichtiges breites CNN

Eine mögliche Verallgemeinerung dieser breiten CNN-Architektur stellt die Verkettung mehrerer „breiter“ Blöcke (*Wide Blocks*) dar. Nach einem Block folgt ein (Max-)Pooling-Layer mit Stride 2, der also die zeitliche Auflösung halbiert (Subsampling). Darauf folgt ein weiterer Block. Dies ist analog zu CNNs zur Bilderkennung, bei denen die Auflösung mittels Pooling reduziert wird, um dann mit vergleichsweise kleinen Filtern größere Strukturen zu erkennen. Je tiefer das Netz wird (und je häufiger die Auflösung reduziert wird), desto größer sind dann auch die Strukturen, die im Text erfasst werden können. Abb. 8 zeigt eine Architektur mit zwei Blöcken, aber prinzipiell ist dies auf beliebige Tiefen (beschränkt durch die Länge der Eingabe) erweiterbar. Zu beachten ist, dass nach dem letzten Block (wie im grundlegenden Fall) stets Max-Over-Time-Pooling verwendet wird, um die endgültige Satzeinbettung zu erhalten.

4.1.4 VDCNN

Im Gegensatz zu den anderen hier vorgestellten Modellen verwendet das 2017 von Conneau et al. [14] vorgestellte *Very Deep Convolutional Network* (VDCNN[28]) nicht Wörter, sondern einzelne Zeichen als Eingabe. Das Kernelement des VDCNN stellt der *Convolutional Block* dar, der aus jeweils zwei Abfolgen von Convolution mit Größe 3, Batch Normalization und Aktivierungsfunktion (ReLU) besteht und in Abb. 9 dargestellt wird. Seiner Bezeichnung entsprechend verkettet das VDCNN eine – verglichen mit anderen Architekturen – hohe Zahl dieser Blöcke in der Tiefe. Die Autoren des VDCNN stellen Varianten mit 8, 16, 28 und 48 dieser Blöcke (und jeweils einem zusätzlichen Convo-

[28] Die zu dieser Abkürzung führende Schreibweise „Very Deep Convolutional *Neural* Networks“ findet sich im Text, jedoch nicht im Titel.

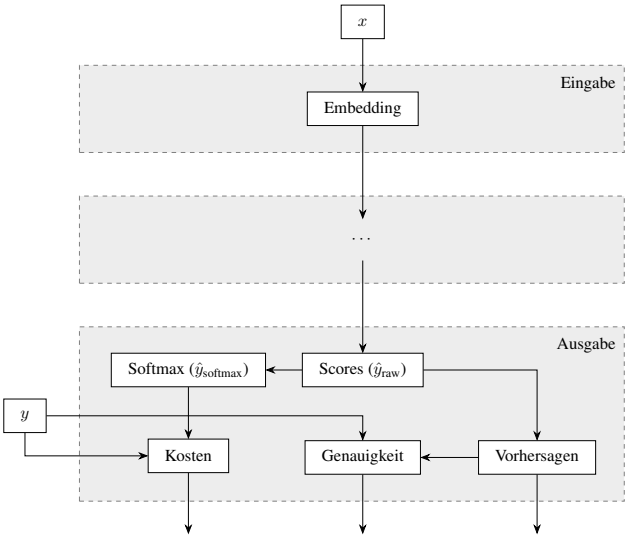


Abbildung 6: Gemeinsame Komponenten der getesteten neuronalen Netze. Die Eingabe x ist ein Batch von zu klassifizierenden Texten und durchläuft erst eine Embedding-Schicht, bevor die so umgewandelten Daten dann in die nächsten, architekturspezifischen Schichten übergehen. Deren Ausgaben wiederum gelangen in die Ausgabeschicht, in der die Scores $\hat{y}_{\text{raw}}$ der einzelnen Klassen berechnet werden. Erst in dieser Schicht werden die tatsächlichen Klassen y benötigt, um die Kosten und die Genauigkeit zu berechnen.

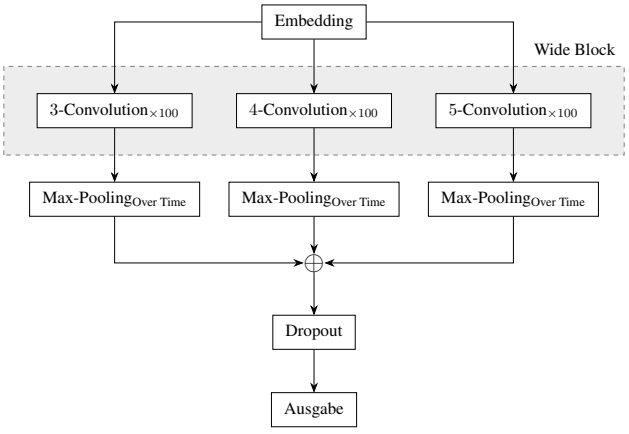


Abbildung 7: Das flache und breite CNN nach Yoon Kim. „i-Convolution$_{\times j}$“ bezeichnet eine 1D-Convolution-Operation mit einer Filtergröße von i und j Feature-Maps. Die Max-Pooling-Layer wählen für jede Filtermaske das größte Element aus (*Max-Over-Time-Pooling*). Anschließend werden die so erhaltenen Werte zu einem Vektor mit $3 \times 100 = 300$ Einträgen konkateniert, der dann zunächst in einen Dropout-Layer eingegeben wird, bevor er in die Ausgabeschicht gelangt.

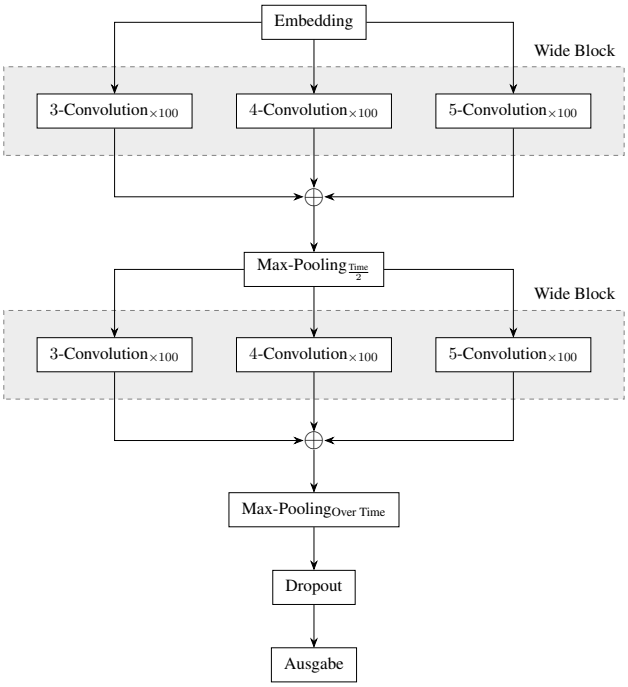


Abbildung 8: Die mehrschichtige Variante des flachen und breiten CNN. Zwischen jeweils zwei Blöcken steht ein Pooling-Layer, der die zeitliche Auflösung halbiert. Nach dem letzten Block folgt – wie auch in der ursprünglichen Variante – ein Max-Over-Time-Pooling-Layer.

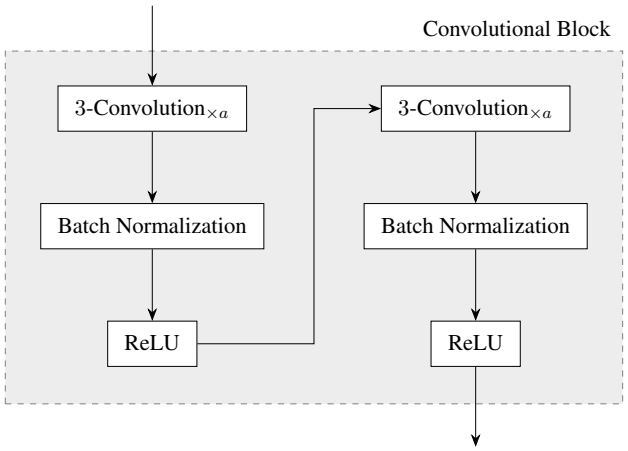


Abbildung 9: Darstellung eines *Convolutional Blocks* im VDCNN. Der Parameter a, der die Anzahl der Feature Maps bestimmt, beträgt entweder 64, 128, 256 oder 512.

lutional Layer vor dem ersten Block) vor, wobei die zweitgrößte Variante zum Zeitpunkt ihres Erscheinens neue Bestwerte auf einigen Datensätzen erzielte. Jede Convolution (einschließlich der initialen vor dem ersten Block) verwendet Padding, um die Dimensionalität der Eingabe beizubehalten.

Die ersten Schichten dieser tiefen Architektur bildet ein Embedding-Layer, der jedes Zeichen der Eingabe in einen 16-dimensionalen Vektor umwandelt und von einem Convolutional Layer mit Größe 3 und 64 Feature Maps gefolgt wird. Dessen Ausgaben wiederum werden in den ersten Block eingegeben, der ebenfalls 64 Feature Maps ausgibt. Danach folgt – je nach gewählter Tiefe – mindestens ein gleichartiger Block, bevor in einem Max-Pooling-Layer[29] die zeitliche Auflösung halbiert wird. Danach folgen wieder mehrere Blöcke, diesmal mit 128 Feature-Maps. Diese Folge von Blöcken und Pooling-Operationen setzt sich fort, wobei nach jedem Pooling die Anzahl der Feature Maps der folgenden Blöcke verdoppelt wird. Nach insgesamt drei Pooling-Operationen haben die letzten Blöcke je 512 Feature Maps.

Nach dem letzten Convolutional Block folgt k-Max-Pooling mit $k = 8$, sodass für jede Feature Map die 8 größten Werte (insgesamt 4096) extrahiert und in einen Fully-Connected-Layer mit ReLU-Aktivierung eingegeben werden. Dieser gibt 2048 Werte aus, die wiederum als Eingabe für einen weiteren Fully-Connected-Layer dienen, der ebenfalls 2048 Werte ausgibt. Diese Werte dienen dann der Score-Berechnung (siehe Abb. 6). Die Architektur der Variante mit 8 Blöcken ist in Abb. 10 dargestellt. Varianten mit mehr Blöcken folgen dem gleichen grundlegenden Aufbau, wobei lediglich die Anzahl der (gleichartigen) Blöcke zwischen den Pooling-Schichten variiert wird.

4.1.5 Einfaches RNN

Eine weitere, sehr grundlegende Architektur besteht aus lediglich einer einzelnen RNN-Schicht, wie in Abb. 11 zu sehen ist. Abgesehen von einer Dropout-Schicht zur Regularisation (und den üblichen Ein- und Ausgabeschichten) sind keine weiteren Schichten erforderlich, um einen funktionsfähigen Classifier zu erhalten. Anpassbare Hyperparameter sind der Typ der RNN-Zelle,

[29] Die Autoren haben auch andere Varianten des Poolings getestet, doch Max-Pooling erzielte die besten Ergebnisse und wird daher auch in diesem Bericht verwendet.

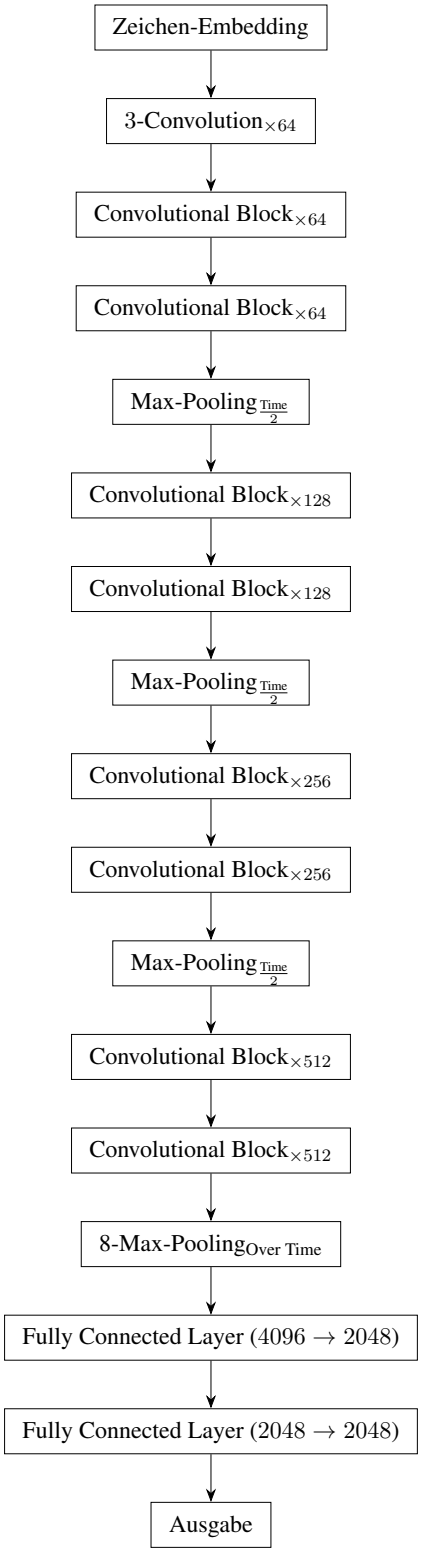


Abbildung 10: Die Grundlegende Architektur des VD-CNN mit 8 Blöcken. Bei Varianten mit mehr Blöcken werden zusätzliche Blöcke zwischen die Pooling-Layer eingefügt (z. B. jeweils 4 statt 2 wie in der Abbildung).

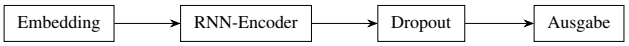


Abbildung 11: Ein einfaches RNN zur Textklassifikation. Das RNN liefert so viele Zustände, wie es Eingabewerte gibt; es wird jedoch nur der letzte Zustand an den Dropout-Layer übergeben, da nur dieser alle Informationen des Satzes enthält.

die Aktivierungsfunktion und die Größe des Zustandsvektors.

4.1.6 CNN-RNN

Wie in Abschnitt 3.3 beschrieben, kann einer rekurrenten Schicht ein Convolutional Layer vorangestellt werden, sodass jeder in das RNN eingehende Vektor nicht nur Informationen über ein einzelnes Wort, sondern über seine Nachbarschaft enthält. Dadurch erfasst das RNN den globalen Kontext in eine Richtung und den lokalen Kontext in beide Richtungen. Da ein Wide Block von sich aus bereits gute Ergebnisse erzielt, liegt es nahe, diesen anstelle eines einzelnen Convolutional Layers (mit nur einer Filtergröße) zu verwenden. Abb. 12 zeigt diesen Aufbau. Alle Hyperparameter des flachen CNN und des grundlegenden, unidirektionalen RNN können verwendet werden, um die Architektur anzupassen.

4.1.7 Bidirektionales RNN

Das bidirektionale Modell ist ebenfalls sehr simpel. Nach der Verkettung der Zustände des Vorwärts-RNN und des Rückwärts-RNN durchläuft die kombinierte Zustandsfolge einen *Encoder*, wie in Abb. 13 verdeutlicht wird. Mehrere *bidirektionale Blöcke* können in der Tiefe aufeinanderfolgen. Zwischen den einzelnen Blöcken findet kein Subsampling statt, da ohnehin immer der globale Kontext betrachtet wird – im Gegensatz zu CNN-Architekturen, bei denen ein lokaler Kontext betrachtet wird, der im Verhältnis zur Eingabe größer wird, je mehr diese herunterskaliert wird.

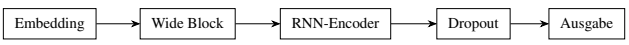


Abbildung 12: Ein einfaches CNN-RNN. Statt einer Sequenz von Worteinbettungen wird eine Sequenz von Feature Maps, die das dem RNN vorgeschaltete CNN ausgibt, zu einer Satzeinbettung kodiert.

4.1.8 Hierarchical Attention Network

Das 2016 von Yang et al. [18] vorgestellte *Hierarchical Attention Network* (HAN) wurde speziell für Datensätze mit mehreren Sätzen entwickelt und entsprach zu Beginn des Bearbeitungszeitraums der diesem Bericht zugrundeliegenden Masterarbeit dem Stand der Technik. Das HAN ist dem zuvor beschriebenen bidirektionalen RNN mit Encoder sehr ähnlich und verwendet die gleichen Komponenten. Diese werden jedoch in einer zweistufigen Hierarchie auf die Eingaben angewendet: In einem Vorverarbeitungsschritt wird der Eingabetext in n einzelne Sätze aufgeteilt. Jeder Satz für sich durchläuft ein bidirektionales RNN mit Encoder, wodurch sich insgesamt n Satzeinbettungen mit je m Komponenten ergeben (die Autoren verwenden $m = 100$, was jeweils 50 Hidden Units für das RNN in Vorwärts- und Rückwärtsrichtung entspricht), die jeweils den Inhalt des Satzes zusammenfassen. Diese n Vektoren werden dann als Sequenz konkateniert, sodass sich eine $n \times m$-Matrix ergibt, die wiederum ein weiteres bidirektionales RNN mit Encoder durchläuft.

Als Encoder kann außer dem namensgebenden Attention-Mechanismus auch eine weitere Methode verwendet werden, etwa ein unidirektionales RNN oder Convolution mit Pooling (eine Variante mit Max-Pooling ohne Convolution wurde von den Autoren getestet und als *HN-MAX* bezeichnet). Zu beachten ist, dass die RNNs und Encoder der untersten Hierarchieebene (d. h. der einzelnen Sätze) die gleichen Gewichte verwenden, sodass zum einen die Anzahl der Parameter reduziert wird (und unabhängig von der Anzahl der Sätze der ersten Hierarchieebene ist) und die Gewichte zum anderen positionsunabhängig sind, d. h. ein Satz ergibt stets die gleiche Repräsentation, ungeachtet, an welcher Stelle im Dokument er sich befindet.

Abb. 14 zeigt die in diesem Bericht verwendete Version dieses Netzes.

4.1.9 Deep Averaging Network

Das *Deep Averaging Network* [24] unterscheidet sich von den zuvor dargestellten Architekturen dadurch, dass die Wortreihenfolge nicht beachtet wird, sondern verworfen wird, indem der Mittelwert aller Wortvektoren gebildet wird (*Average-Over-Time-Pooling*). Zudem wird kein klassischer Dropout-Layer vor dem letzten Fully-Connected-Layer verwendet; stattdessen wird bereits direkt nach der Embedding-Schicht eine

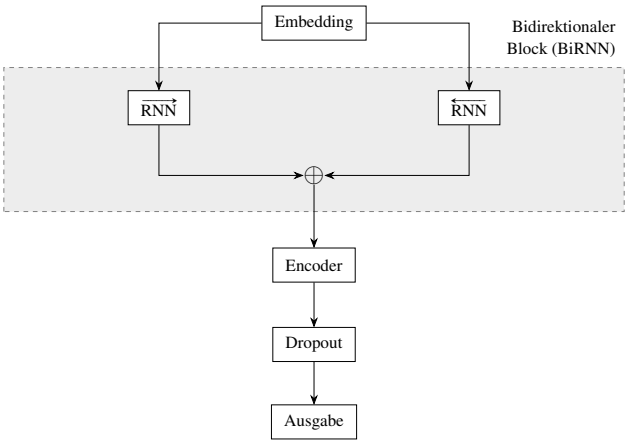


Abbildung 13: Ein einfaches, bidirektionales RNN. Die Eingabe wird einmal mit dem $\overrightarrow{\text{RNN}}$ vorwärts und einmal mit dem $\overleftarrow{\text{RNN}}$ rückwärts durchlaufen. Anstatt – wie beim unidirektionalen RNN – nur den letzten Zustand zu extrahieren, werden *alle* Zustände betrachtet, indem jeder Zustand $\overrightarrow{\text{RNN}}(t)$ jeweils mit dem zugehörigen Zustand $\overleftarrow{\text{RNN}}(t)$ konkateniert wird. Der *Encoder* ermittelt daraus die Satzeinbettung, wobei dies entweder ein unidirektionales RNN, ein Convolutional Layer mit Pooling oder ein Attention-Layer ist.

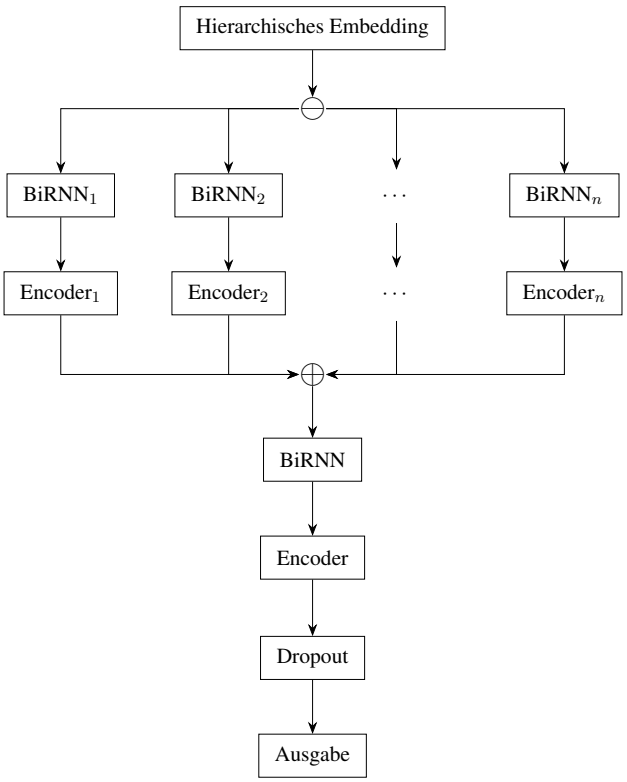


Abbildung 14: Hierarchical Attention Network. Die als „BiRNN“ bezeichneten Blöcke sind die in Abb. 13 verwendeten bidirektionalen Blöcke. „$\ominus$“ steht für das Aufteilen der Eingabe.

abgewandelte Form des Dropouts verwendet, die Wortvektoren als Ganzes verwirft (siehe auch 3.4.1). Die Klassifikation erfolgt dann über eine beliebig wählbare Verkettung von Fully-Connected-Layern. Abb. 15 verdeutlicht diese Struktur.

5 Durchgeführte Tests

Da die Evaluation nicht nur darin besteht, die Klassifikationsmodelle zu vergleichen, sondern letztlich das Ziel hat, ein möglichst gutes Modell zu finden, werden diese Modelle hinsichtlich der in Abschnitt 2.2 beschriebenen Kriterien evaluiert. Das primäre Kriterium ist die Genauigkeit; ein Modell wird ungeachtet aller anderen Leistungsparameter als besser betrachtet, wenn die Genauigkeit höher ist.

5.1 Datensatz

Der Datensatz, auf dem der größte Teil der Tests durchgeführt wird, besteht aus 199 768 Notizen zu Störmeldungen bei einem deutschen Telekommunikationsunternehmen, im Folgenden *Ticket-Datensatz* genannt. Jeder dieser Einträge ist einer von acht Klassen zugeordnert und enthält bis zu 20 Textvermerke, anhand derer die Klassifikation durchgeführt wird. Alle Vermerke stammen dabei von Technikern des Unternehmens, d. h., die ursprüngliche Störungsmeldung des Kunden ist darin nicht enthalten. Dies hat zur Folge, dass alle Einträge eine ähnliche Form und Sprache haben; schließlich sind diese von Technikern für andere Techniker geschrieben. Bei einer Störungsmeldung eines Kunden hingegen wäre eher zu erwarten, dass diese formlos ist und von Kunde zu Kunde stark variiert.

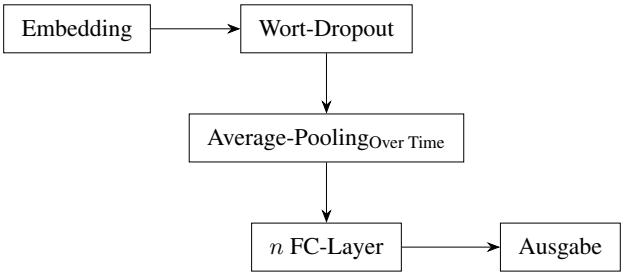


Abbildung 15: Das Deep Averaging Network. „n FC-Layer“ bezeichnet eine Verkettung von n Fully-Connected-Layern. Zu beachten ist, dass der Ausgabeblock ebenfalls einen Fully-Connected-Layer beinhaltet, sodass $n = 0$ ebenfalls möglichst ist. In dem Fall erfolgt jedoch keine Nichtlinearisierung, da der FC-Layer des Ausgabeblocks über keine Aktivierungsfunktion verfügt.

Diese Kommunikation innerhalb einer geschlossenen Gruppe bedeutet aber auch, dass die verwendete Sprache sehr spezifisch ist und Begriffe und Abkürzungen enthält, die von einem Laien nicht verstanden werden. Ebenso werden oft Messwerte, Telefonnummern und andere Zahlen verwendet, die nicht mehrfach vorkommen und damit bei der Tokenisierung kaum oder gar keinen Informationsgehalt bieten – insbesondere kommt es zu Out-Of-Vocabulary-Fehlern, da nicht jede in der Testmenge vorkommende Zahl bereits vorher bekannt sein kann. Des Weiteren enthalten die Texte eine Vielzahl an falschgeschriebenen Wörtern, was wiederum die gleichen Probleme verursacht.

Die Verteilung der Einträge des Datensatzes auf die einzelnen Klassen ist in Abb. 16 dargestellt. Die Klasse A macht mit 140 811 Einträgen über 70% der Daten aus, während die Häufigkeit der beiden seltensten Klassen G und H mit 64 und 31 Dokumenten verschwindend gering ausfällt. Diese sehr ungleichmäßige Verteilung hat einige Auswirkungen auf den Classifier und dessen Bewertung – so ist zu erwarten, dass die Einteilung in die seltensten Klassen ungenau ist, da für diese kaum Trainingsdaten vorhanden sind. Zudem ist eine Genauigkeit von 70%, die auf den ersten Blick ein guter Wert zu sein scheint, tatsächlich nur das absolute Minimum – denn diese Genauigkeit wird bereits von einem Classifier erreicht, der jedem Dokument stets die häufigste Klasse zuweist, ohne die eigentlichen Daten überhaupt zu untersuchen. Die *Null Error Rate* liegt demzufolge bei 29,51%.

Abb. 17 zeigt, wie viele Vermerke jedes Eintrags jeweils ausgefüllt sind. Mehr als die Hälfte der Einträge hat eine Länge von 4 oder weniger, wobei das Ma-

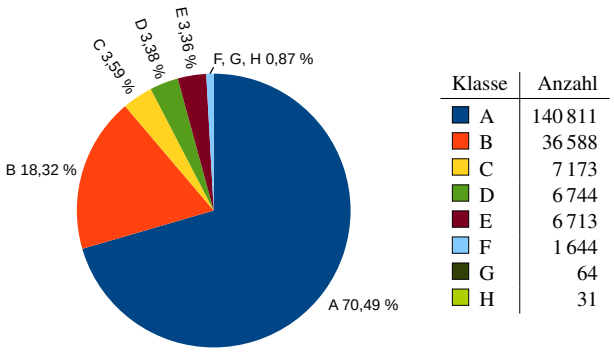
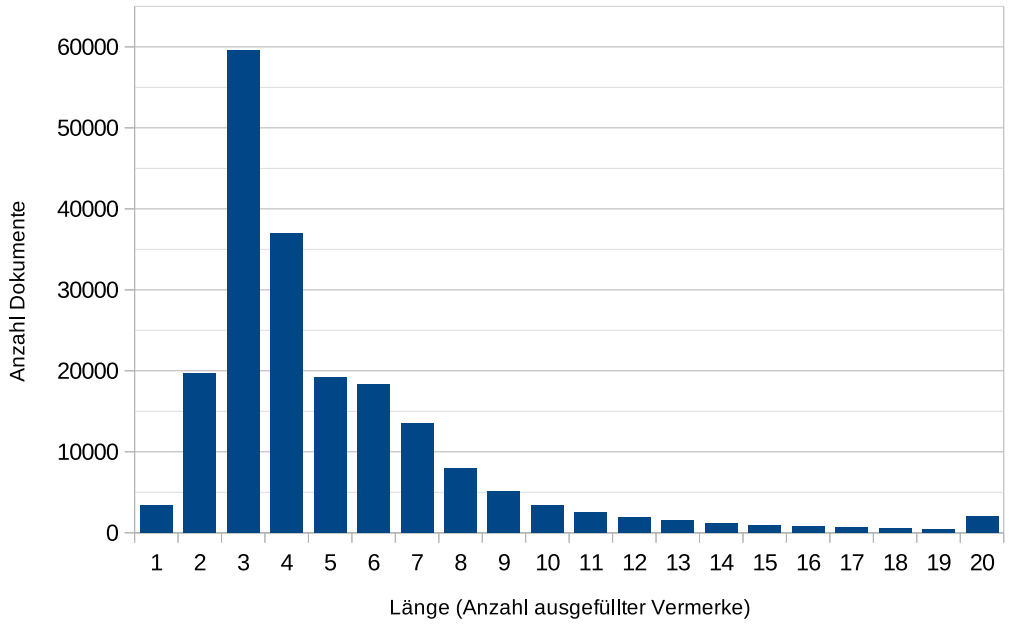


Abbildung 16: Verteilung der Klassen des Ticket-Datensatzes



Länge	Anzahl
1	3 429
2	19 662
3	59 603
4	36 961
5	19 152
6	18 332
7	13 482
8	7 994
9	5 078
10	3 429
11	2 516
12	1 967
13	1 546
14	1 184
15	973
16	822
17	624
18	534
19	444
20	2 036

Abbildung 17: Verteilung der Längen des Ticket-Datensatzes

ximum bei einer Länge von 3 erreicht wird. Darüber nimmt die Anzahl der Einträge mit wachsender Länge ab, wobei es einen Ausreißer bei einer Länge von 20 gibt. Da dies die maximal mögliche Länge ist, liegt die Vermutung nahe, dass sich hier viele Einträge ansammeln, die sonst eine noch größere Länge gehabt hätten und dann entweder gekürzt oder zusammengefasst wurden. Die durchschnittliche Länge beträgt 5,00, und es gibt keine leeren Einträge.

5.1.1 Vorverarbeitung

Für alle Klassifikationsmodelle wird die gesamte Eingabe zunächst in Kleinbuchstaben umgewandelt. Dies ist ein häufiger Vorverarbeitungsschritt und erscheint auf dem Ticket-Datensatz besonders sinnvoll, da Groß- und Kleinschreibung nicht in jedem Eintrag konsistent beachtet wird. Zusätzlich werden alle Umlaute substituiert, d. h. „ß“ wird zu „ss“, „ä“ wird zu „ae“, usw. Dies erfolgt vor dem Hintergrund, dass Umlaute im Ticket-Datensatz nur unregelmäßig verwendet und häufig durch ihre jeweilige Kombination aus zwei Buchstaben ersetzt werden. Zudem wird teilweise die alte deutsche Rechtschreibung verwendet, was sich insbesondere auf die Verwendung von „ß“ bzw. „ss“ auswirkt. Weitere Vorverarbeitungsschritte wie Stemming (das Entfernen von Suffixen, mit dem Ziel, das Wort auf seinen Wortstamm zu reduzieren) und Lemmatisierung (ähnlich dem Stemming, entfernt jedoch nicht nur Suffixe, sondern betrachtet das Wort als Ganzes und ersetzt es – wenn die gefundene Form bekannt ist – durch das *Lemma*, also die Grundform des Wortes im Wörterbuch) erfolgen nicht; nicht nur wäre dies angesichts der oft falschgeschriebenen oder datensatzspezifischen Wörter umständlich, zudem zeigen aktuelle Untersuchungen, dass diese Schritte bei neuronalen Netzen kontraproduktiv sind [28].

Die Tokenisierung wird stets an die Anforderungen des jeweiligen Modells angepasst, wobei folgende Formen unterschieden werden können:

- Bei neuronalen Netzen, die auf Zeichenebene arbeiten, wird bereits vorab ein Vokabular aus erlaubten Zeichen erstellt, und jedes Zeichen der Eingabe wird dann durch den entsprechenden Index innerhalb dieses Vokabulars ersetzt. Das Vokabular entspricht dabei dem, das in der VDCNN-Spezifikation verwendet wird, nämlich:

  ```
  abcdefghijklmnopqrstuvwxyz0123456
  789-,;.!?:'"/|#$%^&*~`+=<>()[]{}␣
  ```

  Dabei stellt „␣“ ein Leerzeichen dar; Zeichen außerhalb dieses Vokabulars werden durch ein UNK-Token ersetzt.

- Bei Netzen, die auf Wortebene arbeiten, werden nur Zeichen als Teil eines Tokens betrachten, wenn diese entweder alphanumerisch oder eines der folgenden Satzzeichen sind:

  `,;!?()`

  Alle anderen Zeichen werden durch Leerzeichen ersetzt, sodass Wörter gegebenenfalls aufgetrennt werden. Im nächsten Schritt werden die noch vorhandenen Satzzeichen ebenfalls mit Leerzeichen voneinander getrennt. Bei der Tokenisierung wird damit jedes Satzzeichen als eigener Term betrachtet. Liegt die Häufigkeit eines Terms in den Trainingsdaten unterhalb einer Schwelle $\mathrm{tf}_{\mathrm{min}}$, wird dieser bei der Tokenisierung durch ein UNK-Token ersetzt. In den meisten Tests dieses Berichts ist $\mathrm{tf}_{\mathrm{min}} = 2$.

Für fast alle Klassifikationsmodelle werden alle Vermerke konkateniert und damit als ein einziges, zusammenhängendes Dokument betrachtet, wobei zwischen den Vermerken jeweils ein Semikolon eingefügt wird, sodass die Grenze von einem neuronalen Netz erkannt werden kann; lediglich bei hierarchischen Modellen wird jeder Vermerk als ein einzelner Satz betrachtet, sodass das Dokument aus bis zu 20 Sätzen besteht.

5.2 Vorgehensweise

Insgesamt habe ich drei verschiedene Testphasen durchgeführt:

1) Am Anfang steht die *Experimentierphase*. Diese nimmt bei weitem den größten Teil der für diesen Bericht aufgewandten Zeit ein und dient dazu, mögliche Kandidaten für die späteren Phasen zu finden. Die Experimentierphase findet parallel bzw. abwechselnd mit der Implementation statt; ich habe jede Architektur zunächst implementiert, getestet und dann entweder angepasst (und erneut experimentell evaluiert), verworfen oder in die nächste Phase übernommen. Aufgrund der Vielzahl von Tests (und des damit verbundenen Rechenaufwands) habe ich die Architekturen in der Experimentierphase nicht kreuzvalidiert, sondern lediglich einfache Tests durchgeführt, um die Tendenz zu erkennen – denn oft gibt es bereits nach einem einfachen Test große Unterschiede zwischen zwei Modellen, die nicht durch die Varianz zu erklären sind. In so einem Fall kann frühzeitig entschieden werden, dass eine bestimmte Architektur nicht ohne Anpassungen weiterverfolgt werden soll. Feinere Unterschiede zwischen zwei Modellen können dann in einer der nächsten Phasen evaluiert werden. Alle Ergebnisse dieser ersten Phase sind im Anhang in Tabelle 8 zu finden. Diese dienen hauptsächlich Dokumentationszwecken, da die Aussagekraft unter der von kreuzvalidierten Ergebnissen liegt. Es kann jedoch die Varianz der kreuzvalidierten Ergebnisse ähnlicher Architekturen herangezogen werden, um diese Ergebnisse zu interpretieren. Deutlich zu sehen sind jedoch bereits die langen Trainingszeiten, die je nach Architektur und Hardware zwischen einigen Stunden und mehreren Tagen liegen.

2) Es folgt die *Hyperparametersuche*. Dazu habe ich geeignete Kandidaten der ersten Phase ausgewählt, um diese hinsichtlich der Genauigkeit und ggf. der Trainingszeit zu optimieren. Da verschiedene Architekturen dabei miteinander verglichen werden, muss in diesem Schritt bereits Kreuzvalidierung stattfinden, um Schwankungen der Genauigkeit (und Trainingszeit) zu reduzieren. Dabei wird davon ausgegangen, dass alle Parameter voneinander unabhängig sind, da eine erschöpfende Suche mit allen Parameterkombinationen *und* Kreuzvalidierung nicht realistisch möglich wäre[30]. Da die nötige Rechenzeit jedoch immer noch sehr hoch ist, habe ich nicht jeden einzelnen Eintrag des Datensatzes einmal in der Testmenge verwendet, sondern die Daten gemäß zehnfacher stratifizierter Kreuzvalidierung aufgeteilt und nur fünf der so erzeugten (disjunkten) Testmengen verwendet[31]. Aufgrund der sehr großen Anzahl an Trainingsdaten und der durch Stratifikation verringerten Varianz ist dabei zu erwarten, dass die Abweichung von den Ergebnissen tatsächlicher zehnfacher stratifizierter Kreuzvalidierung eher gering ausfällt. Ich habe für einige Parameter jeweils eine kleine Auswahl an Werten getestet und dann die Parameterkonfiguration, die dabei die beste durchschnitt-

[30] Auch, wenn diese Annahme in den meisten Fällen unzutreffend sein dürfte, lassen sich damit dennoch akzeptable Ergebnisse erzielen [29].

[31] Eine weitere Option wäre fünffache Kreuzvalidierung; in den Quellen der hier verwendeten Modelle wird aber zumeist ein Zehntel der Daten als Testmenge verwendet, während die Verwendung eines Fünftels eher unüblich ist. Durch die stark unterschiedlichen Größenverhältnisse gäbe es möglicherweise andere Ergebnisse, die dann mit anderen Tests (auch insbesondere im Rahmen dieses Berichts) nicht mehr vergleichbar wären.

liche Genauigkeit erzielt hat, für die nächste Phase in Betracht gezogen.

3) Die finale Phase ist die *Evaluation*, in der ich die besten Architekturen noch einmal mit Kreuzvalidierung getestet und dabei auch kleinere Teilmengen der Trainingsdaten verwendet habe, um festzustellen, wie skalierbar die jeweilige Architektur ist.

5.2.1 Testumgebung

Als Testumgebung habe ich das gleiche Python-Skript verwendet, mit dem die Modelle auch trainiert wurden. Dazu sammelt das Skript während des Trainings jedes Modells einige Daten zu Effektivität und Effizienz, nämlich:

- Die aktuelle Genauigkeit und Kosten auf dem Trainings-Set bei jedem Trainingsschritt
- Die aktuelle Genauigkeit und Kosten auf dem Dev-Set bei jedem Schritt, bei dem dieses ausgewertet wird (alle 50 Schritte bei allen Tests)
- Die bisherige Dauer des Trainings und die Inferenzzeit auf dem Dev-Set, sofern bei dessen Auswertung ein neuer Höchstwert für die Genauigkeit erreicht wurde
- Die Genauigkeit auf dem Test-Set mit dem Modell, das auf dem Dev-Set die höchste Genauigkeit erzielt hat[32]
- Die Confusion Matrix auf dem Test-Set, sobald das Training beendet wird

Diese Ergebnisse können in der Experimentierphase direkt ausgewertet werden. Für die beiden anderen Phasen kann das Skript angewiesen werden, k viele (von 10) Train- und Test-Sets abzuarbeiten. Die so gewonnenen Werte – Genauigkeit, Trainingszeit und Inferenzzeit – werden dann für jedes der k trainierten Modelle ausgegeben und am Ende zusammengefasst, wobei das Minimum, das Maximum, der Mittelwert und die Standardabweichung der jeweiligen Werte ausgegeben wird. Zusätzlich wird eine Confusion-Matrix aufgestellt, die die Ergebnisse aller Tests vereinigt.

[32] Dies wird mindestens einmal am Ende des Tests evaluiert, kann aber, falls erwünscht, auch jedes Mal ausgewertet werden, wenn ein neuer Höchstwert auf dem Dev-Set erreicht wurde und eine einstellbare Mindestanzahl and Trainingsschritten abgearbeitet wurde (damit nicht zu Anfang des Tests, bei dem sehr oft neue Höchstwerte erreicht werden, ständig das Test-Set evaluiert wird)

Das Skript kann außerdem eine Hyperparametersuche durchführen, indem die zu testenden Werte für jeden Hyperparameter angegeben werden. Der Parameter k zur Kreuzvalidierung kann ebenfalls verwendet werden, sodass für den ersten zu testenden Hyperparameterwert k viele Modelle trainiert werden. Wurden alle Werte für einen bestimmten Hyperparameter getestet, so wird derjenige ausgewählt, der im Mittel die höchste Genauigkeit erzielt hat, und der nächste Hyperparameter getestet (wobei alle bisher getesteten Hyperparameter auf ihren jeweils besten Wert gesetzt werden). Zusätzlich zu den bisher erwähnten Ausgaben wird am Ende die Hyperparameterkonfiguration ausgegeben, die die beste Genauigkeit erzielt.

Als Hardware stehen hauptsächlich zwei Testsysteme zur Verfügung:

1) Ein Server mit 24 CPU-Kernen und 48 Threads (2× *Intel Xeon E5-2680 v3* mit je 12 Kernen mit je 2 Threads bei 2,5 bis 3,3 GHz) und 32 GiB Arbeitsspeicher
2) Ein Rechner mit 4 CPU-Kernen und 8 Threads (*Intel Core i7-7700K* mit 4 Kernen und je 2 Threads bei 4,2 bis 4,5 GHz), 16 GiB Arbeitsspeicher und einer Grafikkarte (*NVIDIA GeForce GTX 1080*) mit 8 GiB Grafikspeicher

Aus Zeitgründen musste ich die Tests auf beide Systeme verteilen, was zur Folge hat, dass die gemessenen Trainings- und Inferenzzeiten nur mit Ergebnissen auf dem gleichen System verglichen werden können. Indem einzelne, ausgewählte Tests auf beiden Systemen durchgeführt werden, können diese jedoch zumindest abgeschätzt werden. Zudem bietet dies die Möglichkeit, die Leistung beider Systeme zu vergleichen – insbesondere, weil sich mit einer Server-CPU und einer Grafikkarte zwei sehr unterschiedliche Ansätze gegenüberstehen.

5.3 Ergebnisse und Deutung

Es folgen Zusammenfassungen und Deutungen der in den einzelnen Phasen erzielten Ergebnisse.

5.3.1 Experimente

Die Experimentierphase liefert bereits viele Erkenntnisse über die Leistungsparameter der getesteten Modelle, wenn auch noch viel Raum für Spekulationen

bleibt. Insgesamt wurden vier klassische Modelle getestet: Multinomial Naive Bayes (MNB), Complement Naive Bayes (CNB), Support Vector Machine (SVM) und Logistische Regression. Da diese sehr schnell zu trainieren sind, wurden diese direkt in die nächste Phase übernommen. Lediglich die Support Vector Machines stellen eine Ausnahme dar, da diese sehr hohe Trainingszeiten haben und es nicht möglich war, ein SVM-Modell auf dem vollständigen Ticket-Datensatz zu trainieren. Experimente mit kleineren Teilmengen legten nahe, dass die Genauigkeit mit der der logistischen Regression vergleichbar ist (die wiederum über der Genauigkeit der beiden Bayes-Varianten liegt). Daher wurden SVMs nicht weiterverfolgt.

Da neuronale Netze aus vielen Komponenten bestehen, deren Parameter angepasst werden können, konzentrierten sich die Experimente in erster Linie auf die Architektur. Die Ergebnisse in Tabelle 8 zeigen, dass reine Convolutional Neural Networks auf dem Ticket-Datensatz im Allgemeinen nicht die Genauigkeit erzielen, die bereits von einfachen rekurrenten Netzen erreicht wird. Innerhalb der reinen CNNs ist die Varianz – zumindest auf den ersten Blick – recht gering, und der Unterschied zwischen dem besten und dem schlechtesten Ergebnis beträgt weniger als 2%. Dies legt die Vermutung nahe, dass CNNs bzw. TDNNs im Bereich der Textklassifikation sehr robust sind und es einen sehr großen Spielraum bei den Architekturen gibt, die sich im schlechtesten Fall kaum negativ auf die Genauigkeit auswirken.

Dennoch: Auch wenn die Unterschiede zwischen den Architekturen klein sind, sind sie immer noch vorhanden. Das breite und flache CNN von Kim erzielt auf dem Ticket-Datensatz eine Genauigkeit von etwa 91,5%. Trotz des recht einfachen Aufbaus gestaltete es sich während der Experimentierphase schwierig, diesen Wert zu überbieten. Es zeigte sich bei diesen Tests jedoch deutlich, dass CNNs sehr stark von der Nutzung einer Grafikkarte profitieren – so wurde eine Variante dieses Netzes sowohl auf einer CPU als auch auf einer GPU trainiert, was bei letzterer nur ein Achtel der Zeit beanspruchte.

Die Experimente mit dem Very Deep Convolutional Network (VDCNN) gestalteten sich ebenfalls schwierig. Die Autoren des VDCNN arbeiten mit Dokumenten mit je 1 024 Zeichen; die Dokumente im Ticket-Datensatz haben jedoch teilweise deutlich über 10 000 Zeichen. Da sich ein Convolutional Neural Network am schnellsten auf einer GPU trainieren lässt, wurde für dieses Experiment die Grafikkarte benutzt. Dabei zeigte sich, dass selbst für die einfachste Variante des VDCNN mit 9 Blöcken nicht die gesamte Eingabe verwendet werden kann, da der Speicher der NVIDIA GeForce GTX 1080 nicht ausreicht. Daher musste die Eingabe auf 6 144 Zeichen beschränkt werden. Dieser Wert entspricht etwa der Hälfte des längsten Eintrags und wurde gewählt, da er ein Vielfaches einer möglichst hohen Zweierpotenz ($3 \cdot 2^{11}$) ist, wodurch die zeitliche Auflösung durch Subsampling sehr oft halbiert werden kann, ohne Padding einsetzen zu müssen. Da die Ergebnisse nicht zufriedenstellend waren, wurden die tieferen Varianten des VDCNN getestet – denn diese erzielten laut den Autoren bessere Ergebnisse.

Dabei ergab sich aber wiederum das gleiche Problem: Da durch die größere Tiefe mehr Variablen verwendet werden, wird auch mehr Speicher belegt. Daher musste die Eingabe auf 3 072 Zeichen gekürzt werden. Dadurch konnte die Genauigkeit aber nicht gesteigert werden – möglicherweise, weil zu viele Informationen verlorengehen, wenn die Eingabe so stark gekürzt wird. Um alle Informationen betrachten zu können und die Eingabe trotzdem zu kürzen, wurde an den ersten Convolutional Layer ein Max-Pooling-Layer angehängt, der die Eingabe von 12 288 Zeichen auf 3 072 Zeichen herunterskaliert. Zusätzlich wurde die Größe dieser ersten Convolution von 3 auf 5 erhöht, damit der durch Max-Pooling erhaltene Wert auf jeden Fall Informationen aus allen Zeichen innerhalb des Pooling-Fensters enthält und sich die vom Max-Pooling betrachteten Bereiche überlappen (jeder vom Pooling betrachtete Wert enthält dann Informationen von mindestens einem Zeichen, das auch in einem der benachbarten Pooling-Fenster betrachtet wird). Dies brachte jedoch keine sichtbare Verbesserung der Genauigkeit.

Schließlich wurde auch die Variante mit 29 Blöcken getestet. Auch hier musste die Eingabe gekürzt werden, diesmal auf 1 536 Zeichen. Aufgrund der deutlich höheren Anzahl an Variablen, die zu trainieren waren, verlief das Training deutlich langsamer. Während die verschiedenen Tests der Variante mit 17 Blöcken stets weniger als 40 Stunden brauchten, vergingen bis zur Konvergenz der Variante mit 29 Blöcken 228,24 Stunden. Bis mit einiger Gewissheit gesagt werden konnte, dass das Netz fertig trainiert war, vergingen 315,84 Stunden – also fast zwei Wochen. Dabei wurde der Trainingsdatensatz fast 219-mal durchlaufen, und

das endgültige Ergebnis war mit 90,7% nicht zufriedenstellend.

Die extrem lange Trainingsdauer ließ keine weiteren Tests mit dem VDCNN zu, weswegen dieses nicht weiterverfolgt wurde[33]. Dennoch muss angemerkt werden, dass die endgültige Genauigkeit von 90,7% nur unter Betrachtung der ersten 1 536 Zeichen jedes Dokuments erzielt wurde. Für Datensätze mit kürzeren Dokumenten ist das VDCNN daher möglicherweise immer noch eine Option.

Erfolgreicher hingegen verlief die Vergrößerung der Tiefe des breiten CNN. Varianten mit zwei bis drei Schichten erzielten höhere Ergebnisse auf den Testdaten als die Basisversion. Weniger klar ist, wie sich noch größere Tiefen auf das Ergebnis auswirken. Der Unterschied zwischen zwei und drei Schichten ist marginal und leicht durch die Varianz ähnlicher Tests zu erklären. Ein Test mit acht Schichten ergab eine Genauigkeit von 90,92%, ist also messbar schlechter als weniger tiefe Varianten. Varianten mit noch mehr Schichten wurden nicht getestet – bei einer Eingabegröße von 2 048 sind theoretisch bis zu 11 Subsampling-Layer denkbar, doch es ist anzunehmen, dass die Genauigkeit dabei weiter abnimmt. Möglich ist jedoch, dass sich im Bereich von drei bis sieben Schichten bessere Werte ergeben; die Tiefe des Netzes ist daher ein Kandidat für die Hyperparametersuche.

Ebenfalls wenig beweiskräftig sind die Tests der Filtergrößen. In der Originalversion werden jeweils 100 Filter der Größen 3, 4 und 5 verwendet. In der Experimentierphase wurden zusätzlich Filter der Größen 6 und 7 getestet (wobei die Anzahl der Filter reduziert wurde, um die Anzahl der Variablen nicht zu stark zu erhöhen); die Ergebnisse unterschieden sich jedoch nicht stark, sodass Anzahl und Größen der Filter weitere Kandidaten für die Hyperparametersuche sind.

Wenig erfolgreich verlief das Ersetzen des Max-Over-Time-Poolings durch einen Attention-Layer. Dabei konnte nur eine Genauigkeit von 91,25% erzielt werden; der Attention-Mechanismus hat jedoch anpassbare Parameter, sodass dies ein weiterer Kandidat für die Hyperparametersuche ist.

Bei rekurrenten Netzen zeigte sich während der Experimentierphase, dass einer der wichtigsten Hyperparameter der Typ der verwendeten RNN-Zelle ist. Wird das einfache, in Abschnitt 4.1.5 beschriebene unidirektionale RNN mit einer Basis-RNN-Zelle implementiert, so ergibt sich eine kaum zufriedenstellende Genauigkeit von gerade einmal 83,63%. Ein Test, bei dem die Anzahl der Hidden Units von 128 auf 512 erhöht wurde, resultierte in einer Genauigkeit von 70,69%, was in etwa der Null Error Rate entspricht; das Netz hat also nahezu immer für die häufigste Klasse entschieden. Es ist naheliegend, dass dies auf das Vanishing-Gradient-Problem zurückzuführen ist: Da die Eingabelänge 2 048 beträgt, ergeben sich für die Backpropagation Through Time ebenso viele Schichten. Das Netz ist damit so tief, dass die Backpropagation den Anfang des Netzwerks nicht mehr erreicht. Die Klassifikation erfolgt dann hauptsächlich anhand der späteren Schichten, die aber nicht die nötigen Informationen beinhalten (und da viele Eingaben deutlich kürzer sind, finden sich hier auch oft lediglich Padding-Vektoren).

Diese Vermutung wird auch dadurch untermauert, dass die Verwendung von LSTM- oder GRU-Zellen – die speziell zur Minderung genau dieses Problems entwickelt wurde – deutlich höhere Genauigkeitswerte liefert. Die Genauigkeit einer LSTM-Zelle liegt dabei mit 91,34% im Rahmen dessen, was von CNNs erzielt wird. Die gleiche Architektur mit einer GRU-Zelle hingegen liefert mit einer Genauigkeit von 91,84% einen Wert, der alle zuvor beobachteten Werte übersteigt. Dies deckt sich mit den Ergebnissen von Arkhipenko et al. [30]; alle weiteren Experimente wurden daher mit GRUs durchgeführt.

Experimente mit L2-Regularisation, abnehmender Lernrate und mehr Fully-Connected-Layern brachten keine Verbesserung der Genauigkeit. Erfolgversprechender war hingegen die Verwendung eines dem RNN vorangestellten Wide Block, wodurch eine Genauigkeit von 91,98% erzielt werden konnte. Diese erhöhte Genauigkeit kann dadurch erklärt werden, dass jeder vom RNN betrachtete Vektor Informationen über seine Umgebung enthält (wie in Abschnitt 3.3 beschrieben); der sehr geringe Unterschied und die Diskrepanz auf dem Dev-Set (92,22% Genauigkeit bei RNN, 92,16% bei CNN-RNN) legen jedoch nahe, dass dies eher durch zufällige Schwankungen zu erklären ist.

Eine einfache Implementation eines bidirektionalen RNN – bei dem die finalen Zustände beider RNNs konkateniert und dann in den Fully-Connected-Layer

[33] Dies erklärt auch, warum nicht auf einer CPU mit der gesamten Eingabe getestet wurde – dies wäre zwar aufgrund des großen Arbeitsspeichers problemlos möglich, doch die Trainingsdauer wäre ein Vielfaches der Trainingsdauer auf einer GPU.

eingegeben wurden – brachte keine höhere Genauigkeit als die zuvor getesteten Varianten. Anders verhielt sich dies jedoch, wenn eine Variante des in 4.1.7 vorgestellten bidirektionalen RNN verwendet wurde – wenn also jeder Zustand des Vorwärts-RNN mit dem jeweils passenden Zustand des Rückwärts-RNN konkateniert und die so entstandene Sequenz in einen Encoder eingegeben wurde.

Mit einem Wide Block als Encoder konnte eine Genauigkeit von 92,05% erreicht werden (die Verwendung dreier Fully-Connected-Layer nach dem Encoder brachte eine sehr ähnliche Genauigkeit von 92,1%). Vergleichbare Ergebnisse brachte die Verwendung eines unidirektionalen RNN als Encoder. Dabei wurden auch verschiedene Aktivierungsfunktionen getestet, wobei $\tanh$ mit dem Spitzenwert von 92,44% vor den Varianten mit PReLU (92,02%) und ELU (91,99%) liegt. Während die Genauigkeit letzterer Varianten nahezu gleich ist, gibt die Genauigkeit von $\tanh$ Grund zu der Annahme, dass dies eine bessere Wahl ist. Dennoch ist auch die Suche nach einer Aktivierungsfunktion ein Kandidat für die nächste Phase. Ein Attention-Layer als Encoder ergab in einem Experiment zwar nur eine Genauigkeit von 91,77% auf dem Test-Set, erzielte jedoch mit einem Wert von 92,53% den absoluten Höchstwert auf dem Dev-Set und sollte daher trotz des unscheinbaren Ergebnisses für die nächste Phase berücksichtigt werden.

Eine Vergrößerung der Tiefe durch Verwendung zweier aufeinanderfolgender bidirektionaler RNNs brachte keine sichtbare Verbesserung der Genauigkeit, erhöhte jedoch erwartungsgemäß die Trainingsdauer deutlich. Ebenfalls erfolglos war der Versuch, den letzten Fully-Connected-Layer bei dieser tieferen Variante wegfallen zu lassen und die vom Encoder (einem unidirektionalen RNN mit 8 Hidden Units) erhaltene Satzeinbettung direkt als Scores zu verwenden. Dabei ergab sich eine Genauigkeit von 91,5%; auffälliger als die Genauigkeit ist hingegen die Trainingszeit, die mit 158,7 Stunden (fast eine Woche) mehr als das Vierfache der Trainingsdauer des zuvor getesteten Modells betrug. Die Motivation hinter diesem ungewöhnlichen Test lag in der Frage, ob ein zum All-Convolutional-Network analoges All-Recurrent-Network ebenfalls gute Ergebnisse erzielen könne; diese Frage konnte durch dieses Experiment erfolgreich, wenn auch nicht zufriedenstellend, beantwortet werden.

Als nächstes wurden Modelle entwickelt und getestet, die zunächst einzelne Sätze verarbeiten, für diese jeweils eine Satzeinbettung berechnen und anschließend die so erhaltene Sequenz der Satzeinbettungen in einem weiteren Schritt verarbeiten, um eine Dokumenteinbettung zu erhalten, anhand der dann die Klassifikation erfolgen kann. Dieses Vorgehen entspricht dem des in Abschnitt 4.1.8 vorgestellten *Hierarchical Attention Network* von Yang et al. [18]. Von den drei[34] beschriebenen Encoder-Varianten wurden dabei Attention und CNN mit Pooling getestet; ein unidirektionales RNN als Encoder wurde nicht getestet, da die zu erwartende Rechenzeit zu hoch wäre und Tests mit anderen Architekturen nahelegten, dass die Leistung vergleichbar mit der eines Attention-Encoders wäre. Als RNN-Zelle wurden jeweils – wie im ursprünglichen Paper – GRUs eingesetzt. Als Sätze wurden jeweils die einzelnen Vermerke des Ticket-Datensatzes herangezogen[35], sodass sich insgesamt 20 GRNNs in der untersten Hierarchieebene ergaben, die jeweils die gleichen Gewichte nutzen. Mit 128 Hidden Units und Wide Blocks (wieder mit gemeinsamen Gewichten in der untersten Hierarchieebene) mit Max-Pooling als Encoder konnte so eine Genauigkeit von 91,95% auf dem Ticket-Datensatz erzielt werden. Dies entspricht in etwa dem *HN-MAX* von Yang et al. mit einem zusätzlichen Convolutional Layer. Eine Implementation des HAN, die sich so nah wie möglich an die Spezifikation im Paper hält, konnte Genauigkeiten von 91,97% und 92,41% erzielen, benötigte jedoch sehr hohe Trainingszeiten und konnte daher für die Hyperparametersuche nicht berücksichtigt werden.

Beide Varianten des hierarchischen Netzes erzielten Genauigkeitswerte, die in etwa denen der bisher getesteten rekurrenten Netze entsprechen. Möglicherweise ist der Aufbau des Ticket-Datensatzes für ein solches Modell ungeeignet: Die Stärke des HAN liegt darin, dass es den wichtigsten Satz ermittelt und diesem das größte Gewicht zuweist; da jedoch oft nur ein kleiner Teil der Vermerke ausgefüllt ist und der erste Vermerk oft viele Informationen enthält, liegt die Vermutung nahe, dass dem ersten Vermerk ein sehr hohes und allen anderen Vermerken sehr niedrige (niedriger, je weiter hinten diese stehen) Gewichte zugewiesen werden.

[34] Möglich wäre auch, für Satzeinbettung und Dokumenteinbettung jeweils verschiedene Encoder-Mechanismen zu verwenden, doch dies wurde nicht weiter betrachtet.

[35] Denkbar wäre auch wiederum die Aufteilung der Vermerke in Sätze, was jedoch angesichts der Beschaffenheit dieser Vermerke nur schwer automatisch möglich wäre.

Aus diesem Grund wurden in einem Experiment die durchschnittlichen Gewichte, die bei der Evaluation des Test-Sets berechnet wurden, ermittelt. Diese Gewichte werden in Tabelle 2 dargestellt und bestätigen diese Vermutung – der erste Vermerk dominiert den gewichteten Mittelwert mit 93,48%. Die Gewichte der zweiten und dritten Vermerke sind immerhin noch im einstelligen Prozentbereich, während alle anderen Gewichte verschwindend gering sind. In diesem Fall ist der Attention-Mechanismus überflüssig, da allen Sätzen immer nahezu die gleichen Gewichte zugewiesen werden, was wiederum erklärt, warum die Genauigkeit gegenüber einem „flachen“ bidirektionalen RNN nicht ansteigt.

Das Deep Averaging Network (DAN) wurde nicht im Detail getestet; die Experimente beschränkten sich auf die Variante ohne zusätzliche Fully-Connected-Layer. Dabei zeigte sich, dass diese mit Genauigkeiten von 89,75% und 90,18% weit unter den Leistungen der anderen Netzwerke liegen. Die Trainingsdauer war mit knapp 27 Stunden auf einer CPU ebenfalls vergleichsweise lang, und eine GPU brachte mit 19,67 Stunden eine deutliche, aber bei weitem nicht so drastische Verbesserung, wie es bei CNNs der Fall ist. Bemerkenswert ist, dass sehr oft über den gesamten Trainingsdatensatz iteriert werden musste, bis das beste Ergebnis erreicht wurde; dies war nämlich erst nach über 93 bzw. über 114 Epochen der Fall (wobei die Zeit pro Epoche recht klein ausfiel, was sich an der gesamten Trainingsdauer zeigt). Wird das Average-Pooling durch Attention ersetzt, so war das Training bereits zwar nach knapp 6 Epochen und weniger als 2 Stunden beendet, die Genauigkeit war mit 88,53% jedoch noch geringer.

5.3.2 Hyperparametersuche

Die Genauigkeit klassischer Modelle hängt stark von der Wahl der Features ab, da diese jedes Feature berücksichtigen und schlecht gewählte Features damit die Genauigkeit senken. Diese Zusammenhänge sind in Tabelle 3 dargestellt. Es zeigt sich, dass das binäre Modell – also die einfache Betrachtung, ob ein Term vorhanden ist oder nicht – bei allen getesteten klassischen Modellen die höchste Genauigkeit erzielt[36]. Komplizierte Feature senken bei allen Modellen die Genauigkeit, wobei Complement Naive Bayes und insbesondere logistische Regression weniger betroffen sind. Bei allen getesteten Features ist die logistische Regression den Varianten des Bayes-Classifiers weit überlegen.

Für die neuronalen Netze wurden basierend auf den Ergebnissen der Experimentierphase zwei Modelle für die Hyperparametersuche ausgesucht: Das bidirektionale RNN mit Attention-Encoder, da dieses die besten Ergebnisse erzielte, und das breite CNN, da dieses auf Grafikkarten schnell zu trainieren ist und damit eine Vielzahl von Tests ermöglicht, deren Ergebnisse möglicherweise wieder auf andere Architekturen übertragbar sind. Im Gegensatz zu klassischen Modellen liegt der Fokus dabei nicht auf den Features, da ein neuronales Netz selbstständig erlernen kann, welche Features zu beachten sind, und eine sorgfältige Wahl der Features damit nicht so wichtig ist.

Zunächst wurde untersucht, ob sich durch das Hinzufügen weiterer Convolutional Layers (mit Max-Pooling-Layern zum Subsampling zwischen den Convolutional-Layers) eine Steigerung der Genauigkeit ergibt. Da acht solcher Schichten in der Experimentierphase keine guten Ergebnisse erzielten, bildet dies eine Obergrenze. Getestet wurden daher zwei, drei vier und sechs Schichten. Die Filtergrößen bewegten sich im Intervall $[3; 7]$, wobei die Anzahl der Filter 50, 38, 30, 25, und 22 betrug, sodass mehr kleine und weniger große Filter trainiert wurden. Abb. 18 zeigt, dass es kaum einen Unterschied zwischen der Genauigkeit bei drei und vier Schichten gibt, während diese jedoch beide etwas schlechtere Ergebnisse als die Architektur mit zwei Schichten liefern. Wird die Tiefe noch weiter erhöht, verschlechtert sich die Genauigkeit, was sich durch den großen Einbruch bei sechs Schichten zeigt und die Vermutung der Experimentierphase bestätigt. Dies deckt sich außerdem mit den Ergebnis-

[36] Aus diesem Grund kann auf die Variante des *Multinomial* Naive Bayes verzichtet werden.

Vermerk	1	2	3	4	5	6 bis 20
Gewicht	93,48%	2,93%	2,42%	0,74%	0,26%	0,17%

Tabelle 2: Die durchschnittlichen, durch den Attention-Layer berechneten Gewichte der einzelnen Vermerke im Hierarchical Attention Network. Die Gewichte nehmen mit den späteren Vermerken monoton ab. Das durchschnittliche Gewicht des 20. Vermerks beträgt gerade einmal 0,0016%.

		Features			
		Binär	TF	TF-IDF	TF-IDF mit n-Grammen
Modell	MNB	85,61%	84,43%	81,10%	76,09%
	CNB	85,20%	84,35%	85,13%	82,48%
	Logistische Regression	**90,32%**	90,17%	89,67%	90,19%

Tabelle 3: Die Genauigkeit klassischer Modelle bei Verwendung binärer Wort-Features, Wort-Features mit Termhäufigkeit (TF), Wort-Features mit TF-IDF-Gewichtung und Wort-Features mit TF-IDF-Gewichtung mit TF-IDF-gewichteten Zeichen-$[2; 6]$-Grammen

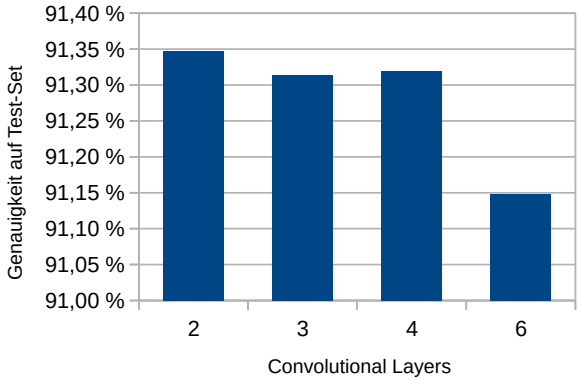


Abbildung 18: Einfluss der Tiefe, d. h. der Anzahl der Convolutional Layers (mit Max-Pooling zum Subsampling), auf die Genauigkeit des breiten CNN.

sen von Le et al. [31], die nur bei zeichenbasierten CNNs einen deutlichen Genauigkeitszuwachs durch Erhöhung der Tiefe beobachtet haben.

Motiviert durch den Erfolg des Attention-Layers bei bidirektionalen RNNs beschäftigte sich der nächste Teil der Hyperparametersuche mit der Frage, ob der Attention-Layer für das breite CNN angepasst werden kann, sodass die Leistungen des Max-Over-Time-Poolings erreicht oder übertroffen werden. Dazu wurde zunächst die Aktivierungsfunktion getestet, wobei dabei nicht nur die Aktivierungsfunktion des Attention-Layers, sondern auch die der Convolutional-Layer getestet wurden. Die Dimensionalität des Kontextvektors wurde stets auf $\dim(\vec{u}) = 100$ festgelegt. Wie Tabelle 4 zeigt, funktioniert $\tanh$ im Attention-Layer deutlich besser als PReLU, während der Unterschied zwischen ReLU und PReLU im Convolutional Layer weniger deutlich ausfällt. Die Kombination aus PReLU als Attention-Aktivierungsfunktion und ReLU als Convolutional-Aktivierungsfunktion wurde nicht getestet, da gemäß der Annahme über die Unabhängigkeit der Hyperparameter keine höhere Genauigkeit zu erwarten wäre und aus zeitlichen Gründen nicht jede Kombination getestet werden kann.

Der nächste zu optimierende Parameter ist die Anzahl der verwendeten Filter. Dazu wurde die Größe der Filter wurde auf 3, 4 und 5 festgelegt, und es wurde einmal mit 50 und einmal mit 100 Feature-Maps je Filter getestet, wobei letzteres der Konfiguration entspricht, mit der die ursprüngliche Architektur erstmals vorgestellt wurde. Mit dieser Konfiguration konnte eine Genauigkeit von 91,33% erzielt werden, während eine Reduktion der Feature-Maps je Filter auf 50 eine Genauigkeit von 91,36% ergab und dabei auch die in den vorherigen Tests verwendete Konfiguration übertraf. Dies ist ein Indiz dafür, dass die verwendete Filteranzahl schlicht zu hoch war.

Da einige Architekturen, wie etwa das VDCNN, die Anzahl der Feature-Maps nach jedem Pooling-Layer erhöhen, wurde als nächstes getestet, ob dies einen Einfluss auf die Genauigkeit des breiten CNN hat. Dazu wurde ein Hyperparameter, die *Zuwachsrate*, eingeführt, mit dem die Anzahl der Feature-Maps nach jedem Pooling-Layer multipliziert wird. Die in Abb. 19 dargestellten Ergebnisse sind uneindeutig; bei einem Faktor von 1,5 (d. h. mit 75 Feature-Maps in der zweiten Schicht) blieb die Genauigkeit etwa gleich, während diese mit einem Faktor von 2 leicht anstieg, mit einem Faktor von 3 jedoch wieder geringfügig abfiel. Das beste Ergebnis gab ein Faktor von 0,5, d. h. mit einer Halbierung der Feature-Maps auf 25 je Filtergröße in der zweiten Schicht. Aufgrund der geringeren Parameterzahl ist damit auch die Inferenzzeit ein wenig kürzer. Die mittlere Trainingszeit war hierbei jedoch am längsten, gefolgt von den Werten 2 und 3, die in dieser Rangfolge die gleichen Positionen einnehmen wie beim Vergleich der Genauigkeit.

Vor dem Hintergrund der verringerten Parameterzahl

		Conv.-Aktivierungsfunktion	
		ReLU	PReLU
Att.-Aktivierungsfunktion	tanh	**91,35%**	91,33%
	PReLU		91,22%

Tabelle 4: Vergleich von Aktivierungsfunktionen im breiten CNN mit Attention. Der beste Wert ist hervorgehoben. Es wurden nicht alle Kombinationen getestet, da gemäß der Unabhängigkeitsannahme jeder Parameter auf seinen bisher besten Wert festgelegt wurde und anschließend verschiedene Werte des nächsten Parameters getestet wurden.

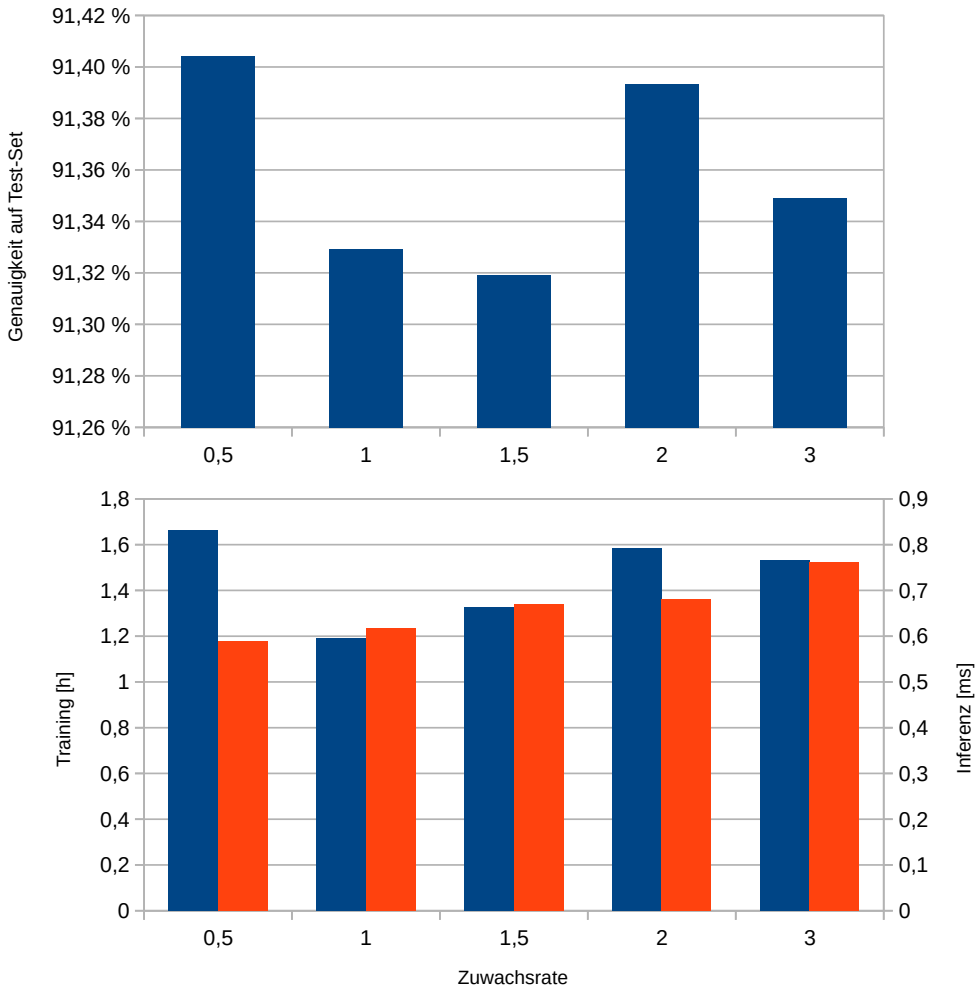


Abbildung 19: Einfluss der Zuwachsrate, d. h. des Faktors, mit dem sich die Anzahl der Feature-Maps mit jeder folgenden Schicht erhöht (bzw. verringert), auf die Genauigkeit sowie ■ Trainings- und ■ Inferenzzeit des breiten CNN mit Attention.

– statt 165 Feature Maps in beiden Schichten ergaben sich jetzt 150 in der ersten und 75 in der zweiten – wurden erneut einige Aktivierungsfunktionen getestet, insbesondere, da PReLU nun weniger Parameter lernen muss und sich möglicherweise besser eignet. Tabelle 5 zeigt die Ergebnisse dieses Tests und bestätigt diese Vermutung zum Teil: Mit PReLU als Aktivierungsfunktion der Convolutional Layer konnte eine Genauigkeit von 91,40% erzielt werden, was gleichauf mit dem Wert der regulären ReLU ist. Die Scaled Exponential Linear Unit (SELU) stellte sich hingegen als ungeeignet heraus, und für den Attention-Layer ist $\tanh$ nach wie vor die beste Aktivierungsfunktion.

Da sich der Attention-Layer nicht mit einer anderen Aktivierungsfunktion verbessern ließ, wurden zum einen verschiedene Werte für die Dimension des Kontextvektors $\vec{u}$ und zum anderen k-Max-Pooling mit verschiedenen Werten für k getestet. Wie sich in Abb. 20 zeigt, liefert ein kleinerer Kontextvektor bessere Ergebnisse, doch liegt die Genauigkeit stets unter der des Max-Over-Time-Poolings. Ähnlich verhält es sich mit k-Max-Pooling: Während es einige Schwankungen gibt, lässt sich dennoch sagen, dass ein größeres k zu einer geringfügigen Verringerung der Genauigkeit führt und stets sichtbar unter der Genauigkeit von $k = 1$ liegt.

Aufgrund der im Vergleich zu CNNs deutlich längeren Trainingsdauer konnten RNNs nicht so ausgiebig getestet werden; zudem wurde die Anzahl der Modelle, die für jede Parameterkonfiguration getestet (und deren Ergebnisse dann gemittelt) werden, von fünf auf drei reduziert. Es ist daher konsequent, dass als erster (und aus Zeitgründen einziger) Parameter die Anzahl der Hidden Units, d. h. die Dimensionalität des Zustandsvektors des bidirektionalen RNNs mit Attention, untersucht wurde, denn schließlich bestimmt diese maßgeblich die Anzahl der Variablen und damit auch die Trainings- und Inferenzzeit. Abb. 21 zeigt die Genauigkeit sowie die Trainings- und Inferenzzeit für 32, 50, 64, 100 und 128 Hidden Units. Es zeigt sich, dass mit 32 Hidden Units eine deutlich geringere Genauigkeit erreicht wird als mit 50. Mit noch größeren Zustandsvektoren nimmt die Genauigkeit geringfügig ab, wobei sich bei so wenigen Tests nicht sagen lässt, ob dies an zufälligen Schwankungen liegt, oder ob 50 tatsächlich das Maximum darstellt. Bei Trainings- und Inferenzzeit fällt das Ergebnis eindeutiger aus: Wie zu erwarten war, nehmen beide Zeiten mit größeren Zustandsvektoren zu. In Anbetracht dieser Tatsache ist 50 die beste Wahl für dieses Modell, da noch mehr Hidden Units die Trainings- und Inferenzzeit, nicht jedoch die Genauigkeit erhöhen.

5.3.3 Kreuzvalidierung und Evaluation

Abschließend wurden Naive Bayes, Complement Naive Bayes, logistische Regression, das zweischichtige breite CNN und das bidirektionale RNN mit Attention auf verschieden großen Teilmengen des Ticket-Datensatzes evaluiert, um die Skalierbarkeit zu testen. Für die klassischen Modelle werden lediglich binäre Wort-Features verwendet; eine Feature Selection fand nicht statt. Für neuronale Netze wurden jeweils die besten in der vorherigen Phase ermittelten Hyperparameter verwendet[37]. Für alle Modelle wurde jeweils eine zufällige, stratifizierte Teilmenge aus dem Ticket-Datensatz ausgewählt und das Modell nach dem üblichen Verfahren evaluiert, d. h., diese Teilmenge wurde noch einmal in Train-, Dev- und Test-Set unterteilt und das Modell darauf trainiert bzw. getestet. Dies wurde bei jedem Modell fünfmal getan und das Ergebnis gemittelt.

Abb. 22 zeigt die bei diesen Tests ermittelte Genauigkeit. Zwischen Naive Bayes und Complement Naive Bayes besteht kein großer Unterschied. Während sich beide Classifier bei kleinen Datenmengen abwech-

[37] Insbesondere betrifft dies die Verwendung von 1-Max-Pooling im CNN-Modell, da die Tests mit einem Attention-Layer unzufriedenstellend waren.

		Conv.-Aktivierungsfunktion		
		ReLU	PReLU	SELU
Att.-Aktivierungsfunktion	tanh	**91,40%**	**91,40%**	91,14%
	PReLU		91,29%	
	SELU		91,23%	

Tabelle 5: Vergleich von Aktivierungsfunktionen im breiten CNN mit Attention und einer Zuwachsrate von 0,5. Der beste Wert ist hervorgehoben.

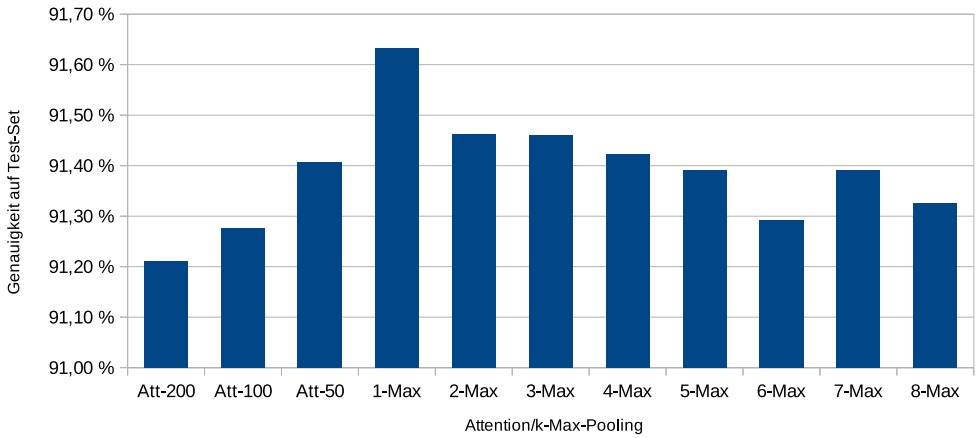


Abbildung 20: Vergleich von k-Max-Pooling und Attention auf dem breiten CNN. „Att-n“ bezeichnet jeweils einen Attention-Layer mit Kontextvektordimension $\dim(\vec{u}) = n$. „1-Max“ bezeichnet das reguläre Max-Over-Time-Pooling.

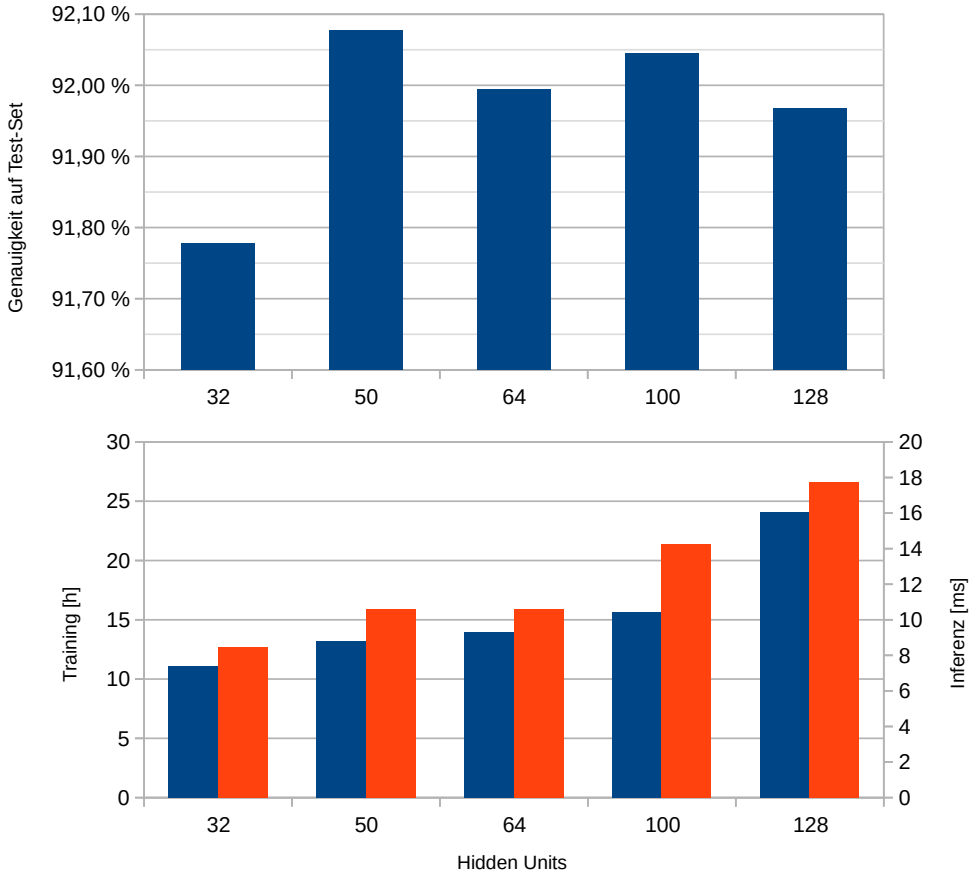


Abbildung 21: Einfluss der Hidden Units, d. h. der Dimensionalität des Zustandsvektors, auf die Genauigkeit sowie Trainings- und Inferenzzeit eines bidirektionalen RNNs mit Attention.

seln, liegt der reguläre Naive Bayes bei größeren Datensätzen mit etwa 0,5% vor dem Complement Naive Bayes. Es ist zu beobachten, dass beim regulären Naive Bayes ab etwa 45 000 Dokumenten (entsprechend einem Viertel des Ticket-Datensatzes) eine gewisse Stagnation eintritt, während die Genauigkeit des Complement Naive Bayes in diesem Bereich weiter ansteigt. Es ist daher möglich, dass dieser den regulären Naive Bayes bei noch größeren Datensätzen übertrifft.

Logistische Regression erzielt bei allen Datensatzgrößen eine höhere Genauigkeit als die Naive-Bayes-Modelle, die zudem stärker ansteigt. Auf dem gesamten Datensatz übersteigt die Genauigkeit die des Naive-Bayes-Classifiers um nahezu 4,7%. Wird die Fehlerrate betrachtet, so beträgt diese mit 9,68% etwa zwei Drittel der Fehlerrate des Naive-Bayes-Classifiers (14,34%).

Das CNN-Modell hat anfangs (bei 450 Trainingsdokumenten, entsprechend $\frac{1}{400}$ des Ticket-Datensatzes) eine Genauigkeit von gerade einmal 73,43% und liegt damit weit unter allen anderen Modellen. Die Genauigkeit skaliert jedoch sehr stark mit der Anzahl der Dokumente, sodass dieses Modell bei 4 500 Dokumenten bereits einzig von der logistischen Regression übertroffen wird. Werden die Genauigkeiten interpoliert, so dürften Naive Bayes und CNN bei 1 800 Dokumenten (entsprechend $\frac{1}{100}$ des Ticket-Datensatzes) gleichauf liegen. Erst ab 45 000 Dokumenten (interpoliert: 20 000) erzielt das CNN bessere Ergebnisse als logistische Regression und steigt danach noch stärker an, wodurch zu erwarten ist, dass dieser Unterschied auf größeren Datensätzen noch deutlicher ist.

Das RNN-Modell bietet selbst auf kleinen Datensätzen bereits eine deutlich höhere Genauigkeit als das CNN und die Bayes-Varianten. Wie auch das CNN-Modell benötigt es etwa 20 000 Trainingsdokumente, um die Genauigkeit der logistischen Regression zu übertreffen. Dies ist auch der Punkt, bei dem die Genauigkeit des CNN-Modells übertroffen wird, nachdem sie bei mittelgroßen Datensätzen leicht darunter lag.

Interessant ist das Verhalten der Modelle zwischen etwa 4 500 und 9 000 Trainingsdokumenten: Die Genauigkeit der logistischen Regression und des Complement Naive Bayes nehmen ab, während die Genauigkeit des regulären Naive Bayes um etwa den gleichen Wert ansteigt. Die Genauigkeit von CNN und RNN steigt zwar nach wie vor an, jedoch deutlich schwächer als an jeder anderen Stelle.

Die Genauigkeit sowie die Trainings- und Inferenzzeit auf dem vollständigen Datensatz sind in Tabelle 6 zu sehen. Tabelle 7 zeigt zudem die Confusion Matrix für das bidirektionale RNN mit Attention, wobei anzumerken ist, dass diese für alle anderen Modelle ähnlich aussieht. Es zeigt sich, dass der Classifier größere Schwierigkeiten bekommt, Dokumente richtig zu klassifizieren, je seltener die Klasse ist. Dabei wird zumeist zugunsten einer häufigeren Klasse entschieden, anstatt die kleinen Klassen untereinander zu verwechseln, was daran zu erkennen ist, dass die Precision dieser Klassen recht hoch, der Recall jedoch gering ist und sich dieses Verhalten bei den häufigeren Klassen umkehrt. Für die beiden seltensten Klassen standen so wenige Daten zur Verfügung, dass keines der Dokumente korrekt klassifiziert werden konnte; dieses Verhalten zeigte sich bei allen getesteten Classifiern.

6 Fazit und Ausblick

Abschließend sollen die Ergebnisse dieses Berichts noch einmal kurz zusammengefasst und ein Ausblick auf mögliche Weiterentwicklungen gegeben werden.

6.1 Zusammenfassung

Zu Beginn dieses Berichts wurde ein Überblick über die Grundlagen der Textklassifikation mit neuronalen Netzen und klassischen Modellen gegeben. Der Schwerpunkt lag dabei auf Recurrent Neural Networks (RNNs) und Convolutional Neural Networks (CNN), da diese zum Zeitpunkt des Entstehens dieses Berichts die erfolgreichsten Modelle in diesem Anwendungsgebiet waren.

Anschließend wurden einige Modelle genauer vorgestellt, die von mir für diesen Bericht implementiert und mit einer ebenfalls für diesen Bericht angefertigten Testumgebung getestet wurden. Zu diesen Tests gehörte zunächst eine Experimentierphase, um einen groben Überblick zu erlangen, welche Modelle geeignet sein könnten. Anschließend wurden die dabei als am besten befundenen Modelle näher untersucht, um einige Hyperparameter zu optimieren. Die optimierten Modelle wurden anschließend auf verschieden großen Teilmengen des verwendeten Datensatzes evaluiert, um ein Verständnis für die Skalierbarkeit der Modelle zu erhalten. Als wichtigstes Kriterium gilt dabei die Genauigkeit, d. h. der Anteil der korrekt klassifizierten

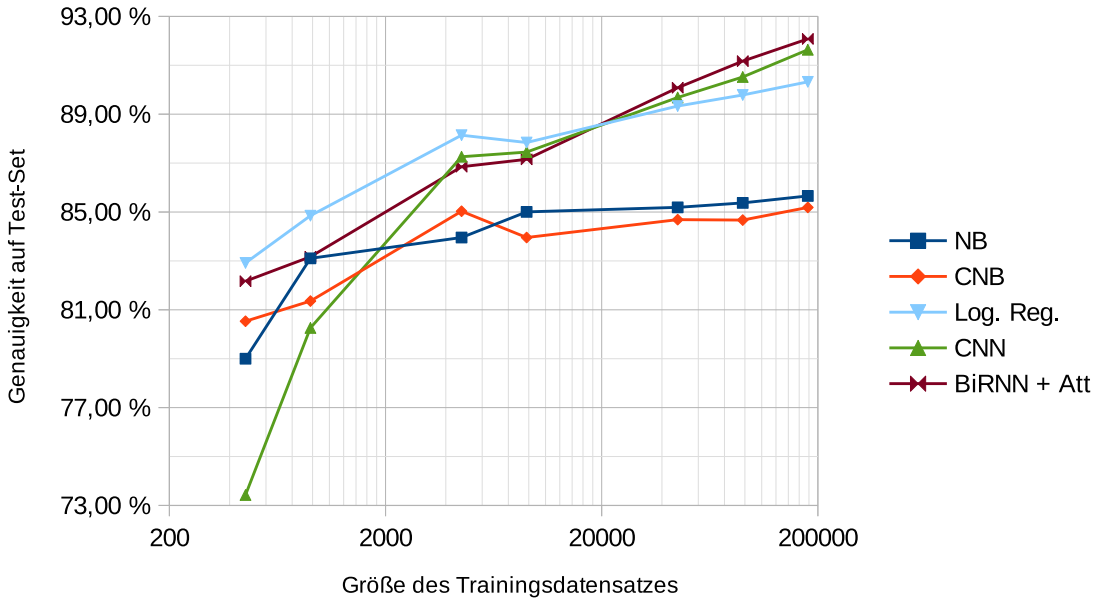


Abbildung 22: Einfluss der Größe des Trainingsdatensatzes auf die Genauigkeit von Naive Bayes (NB), Complement Naive Bayes (CNB), logistischer Regression, zweischichtigem CNN und bidirektionalem RNN mit Attention. Die logarithmisch skalierte x-Achse gibt die Anzahl der Dokumente in den Trainingsdaten an, wobei dies bei den NN-Modellen die Vereinigung aus Train-Set und Dev-Set ist.

Modell	Acc	t_{train} [min]	t_{inferenz} [ms]
Bidirektionales RNN mit Attention	**92,08%**	790,67	10,63
Zweischichtiges breites CNN	91,63%	106,52	0,54
Logistische Regression	90,32%	2,82	**0,10**
Naive Bayes	85,66%	**0,31**	**0,10**
Complement Naive Bayes	85,19%	0,32	**0,10**

Tabelle 6: Ergebnisse der Kreuzvalidierung. Die Zeiten für das CNN wurden auf einer Grafikkarte erzielt und sind daher nicht mit den anderen Zeiten vergleichbar.

		Erkannte Klasse								**Recall**
		A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	
Tatsächliche Klasse	**A**	**41310**	354	262	103	176	41	0	0	97,78%
	B	363	**10086**	24	312	177	15	0	0	91,88%
	C	500	41	**1568**	11	34	0	0	0	72,79%
	D	244	685	9	**990**	71	26	0	0	48,89%
	E	439	387	53	78	**1053**	6	0	0	52,23%
	F	163	76	3	40	23	**190**	0	0	38,38%
	G	9	6	2	1	3	0	**0**	0	0,00%
	H	4	4	1	0	3	0	0	**0**	0,00%
Precision		96,00%	86,66%	81,58%	64,50%	68,38%	68,35%	–	–	

Tabelle 7: Confusion Matrix des Bidirektionalen RNN mit Attention. Die Einträge sind die Summen aus insgesamt drei Testläufen.

Dokumente, doch auch die Geschwindigkeit des Trainings und der Inferenz wurden berücksichtigt.

Dabei stellte sich heraus, dass die Genauigkeit rekurrenter neuronaler Netze auf diesem Datensatz unübertroffen ist. Ein unidirektionales RNN erzielt bereits eine bessere Genauigkeit als alle anderen getesteten Modelle, und diese kann durch Erweiterungen dieses Modells noch erhöht werden; am erfolgreichsten war dabei ein bidirektionales RNN mit einem Attention-Layer. Der Vorteil von Convolutional Neural Networks hingegen liegt in sehr kurzen Trainings- und Inferenzzeiten auf Grafikkarten, während die Genauigkeit etwas hinter der rekurrenter neuronaler Netze liegt.

Damit neuronale Netze jedoch derartige Leistungen erbringen können, müssen erst einmal genug Trainingsdaten vorhanden sein. Ist dies hingegen nicht der Fall (bei dem in diesem Bericht verwendeten Datensatz waren dies etwa 20 000), sollte stattdessen logistische Regression eingesetzt werden. Ein solches Modell lässt sich selbst auf deutlich größeren Datensätzen in nur wenigen Minuten trainieren, während für neuronale Netze mindestens einige Stunden veranschlagt werden müssen. Varianten des Bayes-Classifiers lassen sich noch schneller trainieren, was jedoch angesichts der deutlich geringeren Genauigkeit unbedeutend ist.

6.2 Ausblick

Trotz aller Tests, die für diesen Bericht durchgeführt wurden, ist dies nicht mehr als eine Momentaufnahme. Der Stand der Technik im Bereich der Textklassifikation mit neuronalen Netzen ändert sich sehr schnell – dies zeigt sich etwa mit der Veröffentlichung des *Hierarchical Convolutional Attention Network* [32] (HCAN), das die Leistungen des bisherigen Stands der Technik – des Hierarchical Attention Network – übertrifft. Da dieses jedoch im Juli 2018 und damit spät innerhalb des Bearbeitungszeitraums der diesem Bericht zugrundeliegenden Masterarbeit veröffentlicht wurde, konnte es jedoch nicht mehr berücksichtigt werden. Dieses Modell verspricht jedoch nicht nur höhere Genauigkeit, sondern auch höhere Geschwindigkeit als das HAN und sollte daher Gegenstand zukünftiger Entwicklungen sein.

Nicht berücksichtigt werden konnten Ensembles, d. h. Classifier, die wiederum auf eine Menge von Classifiern zurückgreifen und dabei ein Klassifikationsergebnis auswählen (etwa durch Mehrheitsentscheid). Werden mehrere Classifier verwendet, die sich in ihrer Funktionsweise stark unterscheiden, so können dabei die Vorteile der Classifier genutzt und Ergebnisse erzielt werden, die die der einzelnen Classifier übersteigen [1].

Eine weitere Art von Classifiern, die nicht untersucht wurden, sind Entscheidungsbäume. Diese können automatisch aufgestellt werden und könnten zum Beispiel in Verbindung mit einem Ensemble eingesetzt werden.

Die Vorfilterung bietet ebenfalls noch viel Raum für Untersuchungen. So wurde bisher nur Lowercasing und das Ersetzen von Umlauten eingesetzt, da dies in Tests anderer Autoren die besten Ergebnisse erzielte. Diese Tests basierten jedoch auf Dokumenten in englischer Sprache, sodass es bei deutschsprachigen Dokumenten – insbesondere mit vielen Abkürzungen, Codes und Zahlen – Unterschiede geben könnte. Möglich wären etwa *Compound Splitting* (das Aufteilen von Komposita in ihre Bestandteile, was vor allem für die deutsche Sprache relevant ist [37]) oder Stemming. Für letzteres wäre eine „weiche“ Variante denkbar: Unterschiedliche Wörter mit dem gleichen Wortstamm könnten als unterschiedliche Terme angesehen, aber mit den gleichen Word Embeddings initialisiert werden. Ein neuronales Netz würde diese damit zunächst als gleich betrachten, könnte jedoch Unterschiede lernen, wenn diese das Klassifikationsergebnis verbessern. Ein ähnliches Verfahren könnte für Zahlen angewendet werden, um mehrere Zahlen auf die gleiche Repräsentation abzubilden und die Unterschiede anschließend zu lernen.

Möglicherweise wäre es auch sinnvoll, die Word Embeddings nicht fest einem Term zuzuordnen, sondern diese implizit zuzuweisen: Dafür könnte das HAN (oder HCAN) für Zeichen adaptiert werden. In der untersten Hierarchieebene würde ein bidirektionales RNN mit Attention alle Zeichen eines Tokens verarbeiten, um daraus einen niedrigdimensionalen Vektor zu erhalten. Dadurch wäre jedem Wort immer noch ein Vektor zugeordnet, doch es gäbe keine Out-Of-Vocabulary-Fehler mehr; die Vektordarstellung seltener Terme wäre dann ähnlich zu der Darstellung eines ähnlichen Terms, was insbesondere falschgeschriebenen Wörtern und Zahlen, die selten mehrfach vorkommen, zugute kommen würde.

Generell gibt es gerade im Bereich der Word Embeddings viele Möglichkeiten, die noch untersucht werden können. So können die Word Embeddings bereits vorher trainiert werden, anstatt diese mit zufälligen

Werten zu initialisieren. Dies könnte sich gerade auf kleinere Datensätze positiv auswirken, da für diese unter Umständen nicht genug Trainingsmaterial im Datensatz selbst vorhanden ist.

Darüber hinaus gibt es noch viele weitere Hyperparameter, die getestet werden. Ein Beispiel dafür ist eine exponentiell abnehmende Lernrate, die zwar in einigen Experimenten getestet, dann aber nicht weiterverfolgt wurde, da diese individuell an das jeweilige Modell angepasst werden muss.

In praktischen Anwendungsfällen kann es sinnvoll sein, Dokumente nicht zu klassifizieren, wenn sich der Classifier nicht eindeutig auf eine Klasse festlegen kann – zum Beispiel, wenn die höchste Klassenwahrscheinlichkeit unter einer bestimmten Schwelle liegt. In diesem Fall müssen die Dokumente manuell klassifiziert werden. Dabei werden sowohl Fehlerrate als auch Genauigkeit reduziert, da auch korrekt klassifizierte Dokumente unterhalb dieser Schwelle liegen können. Dennoch kann dies sinnvoll sein, da Fehler generell weniger kosten, je früher sie erkannt werden – es könnte also besser sein, derartige Dokumente frühzeitig manuell zu klassifizieren, anstatt diese erst falsch zu klassifizieren und danach manuell zu klassifizieren. Dies kann den zusätzlichen Aufwand bei der manuellen Klassifikation von Dokumenten, die der Classifier richtig klassifiziert hätte, aufwiegen [12][33].

Danksagung

Dieser Bericht wurde durch die Another Monday Service GmbH Deutschland ermöglicht, die den Datensatz und Rechenkapazitäten bereitgestellt hat.

Anhang

Ergebnisliste der Experimentierphase (nicht kreuzvalidiert)

Architektur und Parameter	Acc	t_{train} [h]	t_{inferenz} [ms]
BiRNN-RNN mit GRU-Zelle mit 128 Hidden Units und $\tanh$-Aktivierung, auf GPU trainiert	92,44%	20,81	20,8
Hierarchical Attention Network, auf GPU trainiert	92,41%	127,41	134,55
BiRNN-CNN mit GRU-Zelle mit 128 Hidden Units, 50, 38, 30, 25 und 22 Filtern der Größen $[3; 7]$, Max-Over-Time-Pooling und 3 Fully-Connected-Layern	92,10%	46,52	
BiRNN-CNN mit GRU-Zelle mit 128 Hidden Units, 50, 38, 30, 25 und 22 Filtern der Größen $[3; 7]$ und Max-Over-Time-Pooling	92,05%	32,21	
BiRNN-RNN mit GRU-Zelle mit 128 Hidden Units und PReLU-Aktivierung	92,02%	64,14	28,3
BiRNN-RNN mit GRU-Zelle mit 128 Hidden Units und ELU-Aktivierung	91,99%	34,64	30,2
CNN-RNN mit 50, 38, 30, 25 und 22 Filtern der Größen $[3; 7]$ und GRU-Zelle mit 128 Hidden Units	91,98%	19,04	
Hierarchical Attention Network	91,97%	18,93	22,3
Hierarchisches bidirektionales RNN-CNN mit GRU-Zelle mit 128 Hidden Units, 50, 38, 30, 25 und 22 Filtern der Größen $[3; 7]$ und Max-Over-Time-Pooling	91,95%	46,08	53,2
BiRNN-RNN mit zwei bidirektionalen Schichten, GRU-Zelle mit 128 Hidden Units und $\tanh$-Aktivierung, auf GPU trainiert	91,91%	36,45	32,8

Architektur und Parameter	Acc	t_{train} [h]	t_{inferenz} [ms]
Unidirektionales RNN mit GRU-Zelle und 128 Hidden Units	91,84%	10,77	
BiRNN-Att mit GRU-Zelle mit 128 Hidden Units[a]	91,77%	*37,20*	*44,8*
Unidirektionales RNN mit GRU-Zelle und 128 Hidden Units und L2-Regularisation mit $\lambda = 0{,}00005$	91,76%	13,90	
Unidirektionales RNN mit GRU-Zelle und 256 Hidden Units und L2-Regularisation mit $\lambda = 0{,}00005$	91,72%	19,14	
Bidirektionales RNN mit GRU-Zelle und 128; Klassifikation mit finalem Zustand beider Richtungen	91,69%	14,39	
Breites CNN mit drei Convolutional-Layern	91,65%	9,50	
CNN-RNN mit 50, 38, 30, 25 und 22 Filtern der Größen $[3; 7]$, GRU-Zelle mit 128 Hidden Units und exponentiell abnehmender Lernrate	91,62%	21,79	
Breites CNN mit zwei Convolutional-Layern und 50, 38, 30, 25 und 22 Filtern der Größen $[3; 7]$	91,52%	22,31	
BiRNN-RNN mit zwei bidirektionalen Schichten, GRU-Zelle mit 128 Hidden Units und $\tanh$-Aktivierung ohne Fully-Connected-Layer (d. h. der unidirektionale RNN-Layer hat nur 8 Hidden Units), auf GPU trainiert	91,50%	158,70	36,1
Breites CNN mit 100, 75, 60, 50 und 43 Filtern der Größen $[3; 7]$, auf CPU trainiert	91,47%	11,80	
Breites CNN mit 50, 38, 30, 25 und 22 Filtern der Größen $[3; 7]$	91,46%	14,30	
Breites CNN mit 50, 38, 30, 25 und 22 Filtern der Größen $[3; 7]$, L2-Regularisation mit $\lambda = 0{,}0001$ und exponentiell abnehmender Lernrate	91,45%	12,39	
Breites CNN mit 50, 38, 30, 25 und 22 Filtern der Größen $[3; 7]$	91,43%		
Unidirektionales RNN mit GRU-Zelle und 128 Hidden Units und 3 Fully-Connected-Layern	91,38%	22,53	
Breites CNN mit 100, 75, 60, 50 und 43 Filtern der Größen $[3; 7]$, auf GPU trainiert	91,35%	1,46	
Unidirektionales RNN mit LSTM-Zelle und 128 Hidden Units	91,34%	18,56	
Breites CNN mit 50, 38, 30, 25 und 22 Filtern der Größen $[3; 7]$ und L2-Regularisation mit $\lambda = 0{,}0001$	91,30%	8,34	
Breites CNN mit zwei Convolutional-Layern, 50, 38, 30, 25 und 22 Filtern der Größen $[3; 7]$ und Attention-Layer (mit $\dim(\vec{u}) = 100$) statt Max-Over-Time-Pooling, auf GPU trainiert	91,25%	1,72	0,8
Breites CNN mit zwei Convolutional-Layern, 50, 38, 30, 25 und 22 Filtern der Größen $[3; 7]$ und 4-Max-Pooling	91,20%	44,82	

[a] Auf AMD A4-5300 mit $2 \times 3{,}4$ GHz und 8 GiB Arbeitsspeicher. Auf dem Dev-Set wurde mit einer Genauigkeit von 92,53% der höchste bisher gemessene Wert erzielt

Architektur und Parameter	Acc	t_{train} [h]	t_{inferenz} [ms]
Breites CNN mit 50, 38, 30, 25 und 22 Filtern der Größen $[3; 7]$ und 4-Max-Pooling, auf GPU trainiert	91,06%	1,62	
VDCNN mit 9 Blöcken und 6144 Zeichen, auf GPU trainiert	90,93%[b]	25,54	
Breites CNN mit acht Convolutional-Layern, auf GPU trainiert	90,92%	2,57	0,9
VDCNN mit 17 Blöcken und 3072 Zeichen, auf GPU trainiert	90,91%	*36,55*[c]	
VDCNN mit 29 Blöcken und 1536 Zeichen, auf GPU trainiert	90,70%	228,24	
VDCNN mit 17 Blöcken, 12288 Zeichen und initialer Convolution + Max-Pooling auf 3072 Zeichen, auf GPU trainiert	90,46%	20,20	
Deep Averaging Network (ohne weitere FC-Layer), auf GPU trainiert	90,18%	19,67	0,4
VDCNN mit 17 Blöcken, 12288 Zeichen und initialer Convolution + Max-Pooling auf 3072 Zeichen, auf GPU trainiert	89,89%	35,52	
VDCNN mit 17 Blöcken und 3072 Zeichen, auf GPU trainiert	89,77%[b]	23,36	
Deep Averaging Network (ohne weitere FC-Layer)	89,75%	26,95	0,6
Deep Averaging Network (ohne weitere FC-Layer) mit Attention statt Average-Pooling, auf GPU trainiert	88,53%	1,49	0,5
Unidirektionales RNN mit Basis-Zelle und 128 Hidden Units	*83,63%*[d]		
Unidirektionales RNN mit Basis-Zelle und 512 Hidden Units	*70,69%*[e]	15,69	

Tabelle 8: Eine Tabelle mit den Ergebnissen aller während der Experimentierphase getesteten Modelle. Da dies nur einfache Tests (ohne Kreuzvalidierung) sind, unterliegen die Ergebnisse Schwankungen und sind daher nicht unbedingt repräsentativ. Die Ergebnisse der ausführlicheren Tests finden sich in den Abschnitten 5.3.2 und 5.3.3.

Literatur

[1] F. Sebastiani, "Machine learning in automated text categorization," *CoRR*, vol. cs.IR/0110053, 2001.

[2] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *ICLR (Workshop)*, 2013.

[3] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov, "Bag of tricks for efficient text classification," *arXiv preprint arXiv:1607.01759*, 2016.

[4] J. T. Jason D. M. Rennie, Lawrence Shih and D. R. Karger, "Tackling the poor assumptions of naive bayes text classifiers," in *Proceedings of the 20th international conference on machine learning*, pp. 616–623, 2003.

[5] C.-W. Hsu and C.-J. Lin, "A comparison of methods for multiclass support vector machines," *IEEE transactions on Neural Networks*, vol. 13, pp. 415–425, 3 2002.

[6] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi, "A survey of deep neural network architectures and their applications," *Neurocomputing*, vol. 234, pp. 11–26, 2017.

[b] Möglicherweise nicht fertig trainiert, da das beste Ergebnis kurz vor Ende des Trainings erzielt wurde
[c] Geschätzter Wert, da der Zugriff auf die Festplatte verlorenging, bevor das Ergebnis ausgegeben werden konnte
[d] Genauigkeit auf Dev-Set, nicht auf Test-Set
[e] Test vorzeitig abgebrochen

[7] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[8] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014.

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," *CoRR*, vol. abs/1502.01852, 2015.

[10] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter, "Self-normalizing neural networks," *CoRR*, vol. abs/1706.02515, 2017.

[11] W. Rawat and Z. Wang, "Deep convolutional neural networks for image classification: A comprehensive review," *Neural Computation*, vol. 29, no. 9, pp. 2352–2449, 2017.

[12] I. J. Goodfellow, Y. Bulatov, J. Ibarz, S. Arnoud, and V. D. Shet, "Multi-digit number recognition from street view imagery using deep convolutional neural networks," *CoRR*, vol. abs/1312.6082, 2013.

[13] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. A. Riedmiller, "Striving for simplicity: The all convolutional net," *CoRR*, vol. abs/1412.6806, 2014.

[14] A. Conneau, H. Schwenk, Y. LeCun, and L. Barrault, "Very deep convolutional networks for text classification," in *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1*, pp. 1107 – 1116, Association for Computational Linguistics, 2017.

[15] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. P. Kuksa, "Natural language processing (almost) from scratch," *Journal of Machine Learning Research 12*, pp. 2493 – 2537, 2011.

[16] N. Kalchbrenner, E. Grefenstette, and P. Blunsom, "A convolutional neural network for modelling sentences," *CoRR*, vol. abs/1404.2188, 2014.

[17] R. Fakoor, X. He, I. Tashev, and S. Zarar, "Constrained convolutional-recurrent networks to improve speech quality with low impact on recognition accuracy," *CoRR*, vol. abs/1802.05874, 2018.

[18] Z. Yang, D. Yang, C. Dyer, X. He, A. Smola, and E. Hovy, "Hierarchical attention networks for document classification," in *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 1480–1489, 2016.

[19] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[20] W. Zaremba, I. Sutskever, and O. Vinyals, "Recurrent neural network regularization," *CoRR*, vol. abs/1409.2329, 2014.

[21] K. Cho, B. van Merrienboer, Ç. Gülçehre, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," *CoRR*, vol. abs/1406.1078, 2014.

[22] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *CoRR*, vol. abs/1207.0580, 2012.

[23] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

[24] M. Iyyer, V. Manjunatha, J. Boyd-Graber, and I. Daum, Hal, "Deep unordered composition rivals syntactic methods for text classification," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, vol. 1, pp. 1681–1691, 2015.

[25] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015.

[26] X. Li, S. Chen, X. Hu, and J. Yang, "Understanding the disharmony between dropout and

batch normalization by variance shift," *CoRR*, vol. abs/1801.05134, 2018.

[27] Y. Kim, "Convolutional neural networks for sentence classification," *CoRR*, vol. abs/1408.5882, 2014.

[28] J. Camacho-Collados and M. T. Pilehvar, "On the role of text preprocessing in neural network architectures: An evaluation study on text categorization and sentiment analysis," *CoRR*, vol. abs/1707.01780, 2017.

[29] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, pp. 281–305, 2 2012.

[30] K. Arkhipenko, I. Kozlov, J. Trofimovich, K. Skorniakov, A. Gomzin, and D. Turdakov, "Comparison of neural network architectures for sentiment analysis of russian tweets," in *Computational Linguistics and Intellectual Technologies: Proceedings of the International Conference "Dialogue 2016"*, no. 15, pp. 50 – 58, 6 2016.

[31] H. T. Le, C. Cerisara, and A. Denis, "Do convolutional networks need to be deep for text classification?," *CoRR*, vol. abs/1707.04108, 2017.

[32] S. Gao, A. Ramanathan, and G. Tourassi, "Hierarchical convolutional attention networks for text classification," in *Proceedings of the 3rd Workshop on Representation Learning for NLP*, pp. 1 – 13, Association for Computational Linguistics, 7 2018.

[33] C. Chow, "On optimum recognition error and reject tradeoff," *IEEE Transactions on Information Theory*, vol. 16, pp. 41–46, 1 1970.

[34] A. Láinez Rodrigo and L. de Oliveira, "Sequential convolutional architectures for multi-sentence text classification," project report, Stanford University, 2015. CS224N - Final Project Report.

[35] P. Liu, X. Qiu, and X. Huang, "Recurrent neural network for text classification with multi-task learning," in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, pp. 2873 – 2879, 2016.

[36] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *arXiv preprint arXiv:1607.04606*, 2016.

[37] C. D. Manning, C. D. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[38] G. Huang, Z. Liu, and K. Q. Weinberger, "Densely connected convolutional networks," *CoRR*, vol. abs/1608.06993, 2016.

[39] C. Felden, H. Bock, A. Gräning, L. Molotowa, J. Saat, R. Schäfer, B. Schneider, J. Steinborn, J. Voecks, and C. Woerle, "Evaluation von Algorithmen zur Textklassifikation," Freiberger Arbeitspapiere 10, Technische Universität Bergakademie Freiberg, Fakultät für Wirtschaftswissenschaften, 5 2006.

[40] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *ICML Workshop on Deep Learning for Audio, Speech and Language Processing*, 2013.

[41] W. B. Cavnar and J. M. Trenkle, "N-gram-based text categorization," in *Proceedings of SDAIR-94, 3rd Annual Symposium on Document Analysis and Information Retrieval*, pp. 161–175, 1994.

[42] J. M. Rao and D. H. Reiley, "The economics of spam," *Journal of Economic Perspectives*, vol. 26, pp. 87–110, 9 2012.