

# **Bachelorarbeit**

im Studiengang Informatik

## **Training eines deutschen automatischen Spracherkennungsmodells basierend auf Jasper**

Vorgelegt von: German Bever  
bever.german@fh-swf.de  
MatNr. 10048672

Datum: 30. Mai 2021, Lüdenscheid

Erstprüfer: Prof. Dr. Christian Gawron  
Zweitprüfer: Prof. Dr. Heiner Giefers

# Eigenständigkeitserklärung

Hiermit erkläre ich, dass die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle sinngemäß und wörtlich übernommenen Textstellen aus fremden Quellen wurden kenntlich gemacht.

Lüdenscheid, 30.05.2021

Ort, Datum

Bever

Unterschrift des Verfassers

# Abbildungsverzeichnis

1.1	Schematische Darstellung einer traditionellen ASR-Grundstruktur . . .	2
2.1	Schematische Darstellung eines künstlich neuronalen Netzes . . . . .	6
2.2	Schematische Darstellung eines tiefen neuronalen Netzes . . . . .	7
2.3	Vergleich der Skalierbarkeit Deep Learning und älteren Algorithmen . .	7
2.4	Schematische Darstellung eines End-to-End Deep Learning Ansatzes . .	8
2.5	Softwarestack NVIDIA NeMo . . . . .	9
2.6	Schematische Darstellung eines konvolutionalen neuronalen Netzes . . .	10
2.7	Schematische Darstellung eines 1D konvolutionalen neuronalen Netzes .	11
2.8	ReLu-Funktionsgraph . . . . .	12
2.9	Schematische Darstellung eines neuronalen Netzes mit Dropout . . . . .	13
2.10	Schematische Darstellung eines CNN mit Dense Residual . . . . .	14
2.11	Schematische Darstellung der Modell-Architektur von Jasper . . . . .	16
2.12	CTC Beispiel anhand einer Audiodatei . . . . .	16
5.1	Schematische Darstellung der Verlustwerte mit 25 Epochen . . . . .	30
5.2	Schematische Darstellung der Verlustwerte mit 50 Epochen . . . . .	31
5.3	Schematische Darstellung der Verlustwerte mit 25 Epochen für ein vor- trainiertes Modell . . . . .	32

# Quellcodeverzeichnis

3.1	Codebeispiel zum Einbinden der Nemo-Modell Bibliotheken . . . . .	17
3.2	Codebeispiel der LightningModules . . . . .	18
3.3	Codebeispiel Lightning Trainer . . . . .	19
3.4	Codebeispiel zum Einbinden eines vortrainierten Jasper Modells . . . . .	20
4.1	Aufbau der Manifest-Datei für NeMo . . . . .	22
4.2	Codebeispiel für die Trainingsdatensätze und der Manifestdatei . . . . .	23
4.3	Codebeispiel zum Training eines Jasper Modells . . . . .	25
4.4	Codebeispiel zum Transfer-Lernen eines vortrainierten englischen Jasper Modells . . . . .	26
4.5	Codebeispiel zum Berechnen der Wortfehlerrate . . . . .	27

# Tabellenverzeichnis

3.1	Jasper10x5Dr LibriSpeech Wortfehlerrate . . . . .	21
5.1	Jasper10x5Dr Common Voice Wortfehlerrate . . . . .	33

# Abkürzungsverzeichnis

<b>ANN</b>	Artificial Neural Network
<b>API</b>	Application Programming Interface
<b>ASR</b>	Automatic Speech Recognition
<b>CNN</b>	Convolutional Neural Network
<b>CTC</b>	Connectionist Temporal Classification
<b>DNN</b>	Deep Neural Networks
<b>GPU</b>	Graphics Processing Unit
<b>JSON</b>	JavaScript Object Notation
<b>Jasper</b>	Just Another Speech Recognizer
<b>KI</b>	Künstliche Intelligenz
<b>KNN</b>	Künstlich euronales Netz
<b>LM</b>	Language Model
<b>NeMo</b>	Neural Modules
<b>ReLU</b>	Rectified Linear Unit
<b>TSV</b>	Tab Separated Values
<b>TTS</b>	Text-to-Speech
<b>NLP</b>	Natural Language Processing
<b>WER</b>	Word Error Rate

# Inhaltsverzeichnis

<b>Eigenständigkeitserklärung</b>	<b>i</b>
<b>Abbildungsverzeichnis</b>	<b>ii</b>
<b>Quellcodeverzeichnis</b>	<b>iii</b>
<b>Tabellenverzeichnis</b>	<b>iv</b>
<b>Abkürzungsverzeichnis</b>	<b>v</b>
<b>Inhaltsverzeichnis</b>	<b>vi</b>
<b>1 Einleitung</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Zielsetzung . . . . .	3
1.3 Aufbau der Arbeit . . . . .	3
<b>2 Stand der Technik</b>	<b>5</b>
2.1 Künstliche neuronale Netze . . . . .	5
2.2 Tiefe neuronale Netze . . . . .	6
2.3 Ende-zu-Ende-Lernen . . . . .	8
2.4 NVIDIA NeMo . . . . .	8
2.5 Konvolutionale neuronale Netze . . . . .	9
2.6 Jasper . . . . .	10
<b>3 Konzeption</b>	<b>17</b>
3.1 Technologien . . . . .	17
3.1.1 NVIDIA NeMo . . . . .	17
3.1.2 PyTorch und PyTorch Lightning . . . . .	18
3.1.3 Jasper . . . . .	20
3.2 Anforderungen . . . . .	20
3.2.1 Wortfehlerrate . . . . .	20
<b>4 Implementierung</b>	<b>22</b>
4.1 Trainingsdatensätze . . . . .	22
4.2 Konfigurationsdatei . . . . .	24
4.3 Modelltraining . . . . .	24
4.4 Transfer-Lernen . . . . .	26
4.5 Wortfehlerrate . . . . .	27

<b>5</b>	<b>Evaluation</b>	<b>29</b>
5.1	Ergebnisse . . . . .	29
5.1.1	Modelltraining . . . . .	29
5.1.2	Wortfehlerrate . . . . .	32
<b>6</b>	<b>Fazit und Ausblick</b>	<b>34</b>
	<b>Literaturverzeichnis</b>	<b>36</b>

# 1 Einleitung

## 1.1 Motivation

Die Automatische Spracherkennung (engl. Automatic Speech Recognition, ASR) erleichtert die Kommunikation zwischen Mensch-Mensch sowie Mensch-Maschine und ist seit über 60 Jahren ein aktives Forschungsgebiet [1, S. 1]. Ein Problem der ASR ist die automatische und genaue Transkription der menschlichen Sprache in Textform. Zur Transkription der Sprache wird ein Modell zur automatischen Spracherkennung trainiert, dass die unterschiedlichen menschlichen Aussprachen von Wörtern und Sätzen erlernen und wiedergeben kann [2, S. 1].

Die individuelle Aussprache, Redegeschwindigkeit, Sprachgewohnheiten, Betonung sowie der Dialekt erschweren die Spracherkennung des Modells [3, S. 25 f.]. Aufgrund dieser Tatsache ist die Verwendung von unterschiedlichen Audio- und den entsprechenden Textdaten zum Training des Modells wichtig. Die Weiterentwicklung der Prozessoren und Grafikprozessoren führte in den vergangenen Jahren zu einem Zuwachs der Rechenleistung und ermöglichen die Verarbeitung großer Datenmengen zum Trainieren der Modelle. Hierfür können kommerzielle oder frei verfügbare Softwaresysteme verwendet werden. Inwiefern die frei verfügbaren Modellansätze und die frei verfügbaren Trainingsdaten sich mit den aktuellen kommerziellen ASR-Produkten vergleichen, wird innerhalb dieser Arbeit untersucht.

Abbildung 1.1 zeigt eine schematische Darstellung einer Grundstruktur eines automatischen Spracherkennungssystems mit dem traditionellen Ansatz, in dem das eingehende Sprachsignal in Teilaufgaben oder Module zerlegt und bearbeitet wird. Die Bedeutung der folgenden Module ist:

- Akustisch-phonetische Analyse:
  - Das eingehende Sprachsignal wird mit Hilfe der akustisch-phonetischen Analyse in kleine sprachliche Einheiten unterteilt, für die die akustischen Charakteristika bekannt oder in einer Lernphase bestimmt worden sind

[2, S. 4 f.]. Mit Hilfe des gesprochenen Lautmusters oder auch Sonagramm können bestimmte bedeutungsunterscheidende Wörter, Phoneme genannt, festgestellt und somit zur Erkennung des Satzes beigetragen werden [2, S. 5 f.]. Das Ersetzen einer Phoneme, kann die Bedeutung des erkannten Satzes ändern, z. B. Beet - Bett oder rasten - rasen (kurzes und langes a).

- Wortidentifikation:
  - Bei der Wortidentifikation hat die Lexikonsuche die Aufgabe, das passende oder ähnliche Wort zu bestimmen und rückwirkend auf die Lautabgrenzung der akustisch-phonetische Analyse anzupassen [2, S. 146 f.].
- syntaktische Analyse
  - Bei der syntaktischen Analyse werden alle Wörter aus dem gesprochenen Satz, die zur selben Wortart gehören, in einer Gruppe zusammengefasst und auf Syntaxfehler überprüft [2, S. 167 f.].
- semantische Analyse
  - Bei der semantischen Analyse wird der Bedeutungsgehalt der Wörter im Satz analysiert und ggf. auf Syntaxfehler kontrolliert [2, S. 167 f.].

In jeder Verarbeitungsstufe können sich Rückführungen auf vorhergehende Teilaufgaben ergeben, um aus dem Sprachsignal die exakten Wörter und somit den geeigneten erkannten Satz zu erhalten.

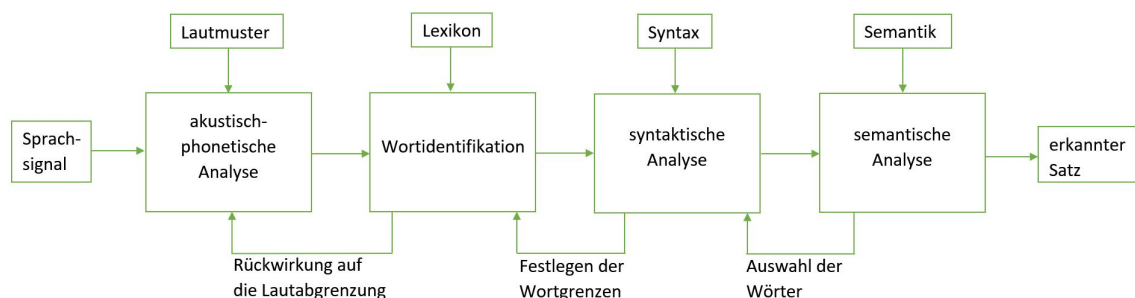


Abbildung 1.1: Schematische Darstellung einer traditionellen Grundstruktur für ein automatisches Spracherkennungssystem (Quelle: In Anlehnung an [2, S. 2 Bild 1.1])

Die anerkannteste Methode zur Bewertung der ASR-Modelle ist die Wortfehlerrate (engl. Word Error Rate, WER). Die WER dient zur Messung der Leistung von Spracherkennungs- oder maschinellen Übersetzungssystemen.

Ein akzeptabler WER-Wert für ein ASR-Modell liegt zwischen 10-20% und entspricht einer durchschnittlichen Spracherkennung. Im Vergleich ist der WER-Wert eines Menschen bei 4% [4, Kap. 2 f.] Je geringer der WER-Wert, desto besser ist das Spracherkennungssystem. Die WER berechnet sich als Quotient aus der Zahl der Wortfehler (Vertauschungen, Einfügungen, Auslassungen) und der Anzahl der gesprochenen Wörter [5]:

$$WER = \frac{S + D + I}{N} [5]$$

- N: Anzahl der Wörter die Gesprochen wurden
- S: Anzahl der vertauschten Wörter in der Erkennung
- D: Anzahl der fehlenden Wörter in der Erkennung
- I: Anzahl der eingefügten Wörter in der Erkennung

## 1.2 Zielsetzung

Unterschiedliche Unternehmen verwenden unterschiedliche Modelle zur automatischen Spracherkennung und stellen diese kommerziell zur Verfügung, wie z. B. Azure Speech Services, Watson Speech to Text als Clouddienste oder die Nuance Transcription Engine. Im Laufe dieser Bachelorarbeit wird versucht mit dem frei verfügbaren Modell Jasper [19] und dem frei verfügbaren Common Voice Corpus ein ASR-Modell zu trainieren, dass einen ähnlich guten WER wie kommerzielle ASR-Modelle erzielt. Basis der Überlegung ist es, mit einer Open-Source-Software einen möglichst geringen WER-Wert zu erreichen sowie die Differenz zwischen den kommerziellen und den freien Modellen zu berechnen. Ebenfalls wird innerhalb der Bachelorarbeit nicht der traditionelle Ansatz, sondern der moderne Ansatz zum Training des Spracherkennungsmodells verwendet.

## 1.3 Aufbau der Arbeit

In diesem Abschnitt wird die weitere Struktur der Arbeit erklärt. Die Inhalte der nachfolgenden Kapitel sind wie folgt aufgeteilt:

### Stand der Technik

In Stand der Technik wird der moderne Deep Learning End-to-End Ansatz sowie die verwendeten Technologien für die vorliegende Arbeit veranschaulicht.

## **Konzeption**

Die Konzeption beinhaltet den hier verwendeten Systemaufbau bezüglich des Modells. In dem Kapitel wird zudem beschrieben, wie die jeweiligen Komponenten zusammenwirken und welche Anforderungen das Modell erfüllen soll.

## **Implementierung**

Innerhalb dieses Kapitels wird die Implementierung des Sprachmodells dokumentiert, die dazugehörigen Umwandlung der Testdatensätze sowie die Berechnung der Wortfehlerrate aufgezeigt.

## **Evaluation**

Dieser Abschnitt ist für die Evaluation vorgesehen. Es erfolgt eine Auswertung der Trainings- und Evaluierungsergebnisse.

## **Fazit und Ausblick**

Abschließend wird ein Fazit der Arbeit gezogen und ein Ausblick auf weitere Schritte und Verbesserungsmöglichkeiten gegeben.

## 2 Stand der Technik

In diesem Kapitel wird der aktuelle themenbezogene Stand aus Technik präsentiert. Es wird ein Überblick über künstliche sowie tiefe neuronale Netze gegeben. Wenn- gleich das Implementieren von neuen neuronalen Netzen kein Bestandteil dieser Arbeit ist, dient ein grundlegender Überblick über das Thema zum besseren Ver- ständnis der Bachelorarbeit. Daraufgehend wird ein Überblick über das Verwendete Toolkit NVIDIA NeMo und dem konvolutionalen neuronalen Akustikmodell Jasper für diese Bachelorarbeit gegeben.

### 2.1 Künstliche neuronale Netze

Künstliche neuronale Netze kurz: KNN (engl. Artificial Neural Network, ANN) ist durch das menschliche Gehirn inspiriert und wird für die Künstliche Intelligenz (KI), für das maschinelle Lernen und Deep Learning eingesetzt [6]. Im Gegensatz vom Entwickler vorprogrammierten Funktionen zum Abarbeiten von Informationen für den Computer wird diesem komplexes Denken angeeignet. Unstrukturierte große Mengen von Daten aus unserem Alltag wie z. B. Videos, Bilder oder Töne werden von einem Computer ausgewertet und in Mustern sowie Formen analysiert [7]. Die eigene angelernte Erfahrung unterstützt den Computer bei der selbstständigen Lö- sungsfindung unterschiedlicher Probleme.

Abbildung 2.1 zeigt eine schematische Darstellung eines künstlich neuronalen Net- zes. Das KNN besteht aus Neuronen in unterschiedlichen Schichten, die von außen oder von anderen Neuronen Informationen aufnehmen, verändern, weiterleiten und als Endergebnis ausgeben können [6]. Die Eingabeschicht nimmt bestimmte Infor- mationen auf, verarbeitet diese und leitet sie an die verborgene Schicht weiter. Die verborgene Schicht befindet sich zwischen der Eingabe- und Ausgabeschicht und bildet interne Informationsmuster ab. Die letzte Ausgabeschicht bewertet die Infor- mationen und gibt das Ergebnis an die Außenwelt weiter.

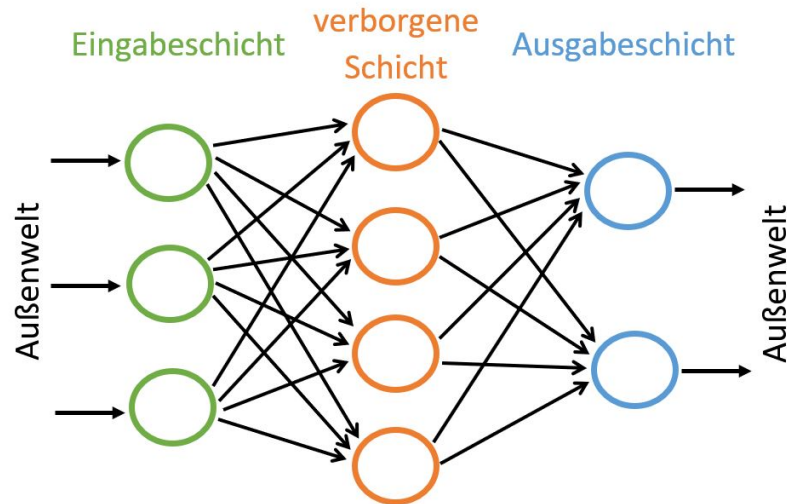


Abbildung 2.1: Schematische Darstellung eines künstlich neuronalen Netzes. Die Eingabeschicht (grün) erhält Informationen von der Außenwelt, dargestellt durch den unidirektionalen Pfeil. Diese Informationen werden zwischen den Neuronen (grün, orange und blau markiert) innerhalb der unidirektionalen Kanten verarbeitet und an die Ausgabeschicht für die Außenwelt ausgewertet und wiedergegeben (Quelle: In Anlehnung an [7])

Die Werte der Neuronen innerhalb des KNN werden ausgehend vom richtigen Ergebnis mit unzähligen Testdurchläufen optimiert, bis das tatsächliche Ergebnis dem gewünschten Ergebnis entspricht [7]. Die Anzahl und Gewichtung der unterschiedlichen Neuronen innerhalb der Schichten ist abhängig von den jeweiligen Parametern im speziellen Einsatzgebiet. Das Verwenden von sortierten und strukturierten Datensätzen sowie der Gebrauch von ausreichender Rechenleistung unterstützt das KNN beim Lernprozess, bis eine bestimmte Intelligenz erreicht wurde [6].

## 2.2 Tiefe neuronale Netze

Tiefe neuronale Netze (engl. Deep Neural Networks, DNN) ist eine spezifizierte Optimierungsmethode der künstlichen neuronalen Netze. Das DNN verwendet unterschiedliche Deep Learning Algorithmen mit Hilfe der KNN zum Erlernen eigener Prognosen und Entscheidungen für unterschiedliche Probleme [8]. Zur Lösung komplexer Probleme werden DNNs verwendet, die eine Vielzahl von verborgenen Schichten und Neuronen besitzen (siehe Abbildung 2.2). Die Mindestanzahl der verborgenen Schichten in einem DNN liegt bei zwei, kann aber je nach Menge der Eingabedaten deutlich erhöht werden.

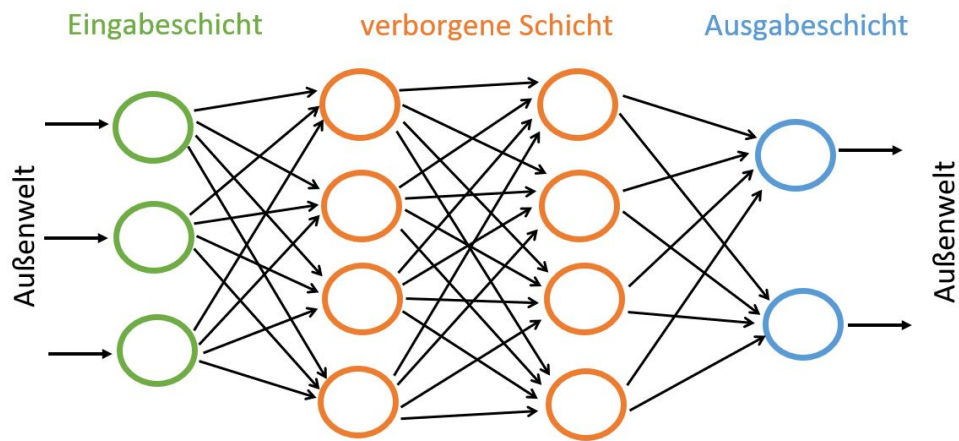


Abbildung 2.2: Schematische Darstellung eines tiefen neuronalen Netzes mit zwei verborgenen Schichten. Die Eingabeschicht (grün) erhält Informationen von der Außenwelt, dargestellt durch den unidirektionalen Pfeil. Diese Informationen werden zwischen den Neuronen (grün, orange und blau markiert) innerhalb der unidirektionalen Kanten verarbeitet. Das tiefe neuronale Netz enthält mehrere verborgene Schichten, in der die Informationen innerhalb der Neuronen verarbeitet und an die Ausgabeschicht für die Außenwelt ausgegeben werden (Quelle: In Anlehnung an [9, Kap. 1.3])

Die Anzahl der Neuronen und verborgenen Schichten beeinflussen den Lernprozess zur verbesserten Entscheidungsfähigkeit der Maschine, z. B. kann beim Einsatz einer Spracherkennung und einem gut trainierten Modell, der Wortschatz selbstständig mit neuen Wörtern oder Wortanwendungen erweitert werden [8]. Die Effektivität zum Erlernen von komplexen Mustern aus großen Datenmengen unterstützt die Skalierbarkeit der DNN siehe Abbildung 2.3.

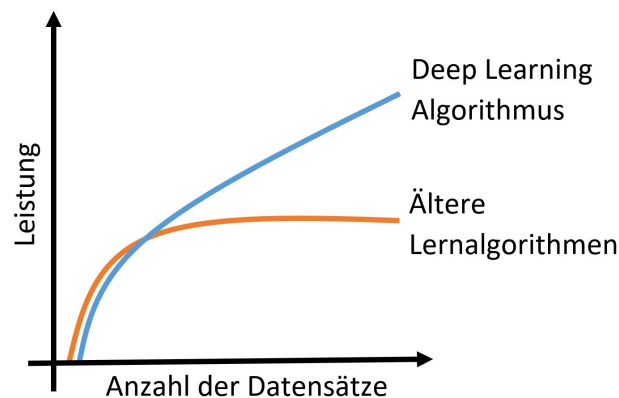


Abbildung 2.3: Vergleich der Skalierbarkeit zwischen Deep Learning und älteren Lernalgorithmen. Je mehr Datensätze der Deep Learning Algorithmus erhält, desto effektiver ist die Leistung im Vergleich zu älteren Lernalgorithmen (Quelle: In Anlehnung an [10, Folie 30])

## 2.3 Ende-zu-Ende-Lernen

Der moderne Ende-zu-Ende-Lernansatz (engl. End-to-End Deep Learning) in Abbildung 2.4 ersetzt den traditionellen Ansatz aus Abbildung 1.1.

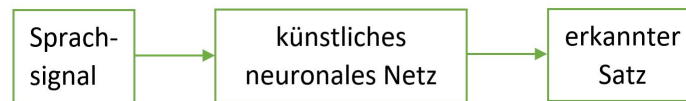


Abbildung 2.4: Schematische Darstellung eines End-to-End Deep Learning Ansatzes. Die Eingabe des Sprachsignals wird innerhalb des KNN Moduls direkt als erkannter Satz ausgegeben (Quelle: In Anlehnung an [11, Kap. 2.2.2])

Der moderne Ansatz verknüpft die unterschiedlichen Module aus dem traditionellen Ansatz, die selbstständig angelernt werden mussten, zu einem einzigen Modul, die sich an den KNN Vorteilen zum Lösen komplexer Probleme bedienen [12]. Ziel ist es, anhand der schematischen Darstellung aus Abbildung 2.4, nach der Eingabe des Sprachsignals den erkannten Satz direkt auszugeben und dabei die unterschiedlichen Module des traditionellen Ansatzes zu umgehen. Das Nutzen des End-to-End Deep Learning Ansatzes ermöglicht das Einsetzen von großen Testdaten, das Training intelligenter Modelle mit Hochleistungsrechnern sowie das Erforschen von neuronalen Netzarchitekturen [12].

## 2.4 NVIDIA NeMo

Das Open-Source-Toolkit NVIDIA NeMo (**Ne** ural **Mo** dules) wird für die Entwicklung moderner KI-Modelle der Kommunikation verwendet [13]. Das Python-Toolkit unterstützt den Anwender bei der Erstellung, dem Training und der Feinabstimmung der Kommunikationsmodelle. Diese Modelle werden für die Anwendungen zur automatischen Spracherkennung (ASR), natürlichen Sprachverarbeitung (engl. Natural Language Processing, NLP) sowie Text-zu-Sprache (engl. Text-to-Speech, TTS) erstellt [13].

### NeMo Softwarestapel

NeMo besteht aus mehreren zusammengestellten Komponenten, neuronalen Netzen, Bibliotheken und Frameworks [14]. Abbildung 2.5 bildet das Grundmodell als Softwarestapel (engl. Softwarestack) ab. Der Anwender kann mit Hilfe von NeMo vortrainierte oder neue Modelle für den eigenen spezifischen Gebrauch verwenden

und diese innerhalb des PyTorch Frameworks, mit einer Vielzahl neuronaler Netze, integrieren und erweitern [14]. Das PyTorch Lightning Framework sowie NVIDIA-As Programmierungsmodell des Grafikprozessors (engl. Graphics Processing Unit, GPU) CUDA-X, ermöglichen einen schnellen Aufruf von Trainingsaktionen sowie dem Bereitstellen von zusätzlicher Rechenkapazität zur parallelen Skalierung und Feinabstimmung der Modelle [14].

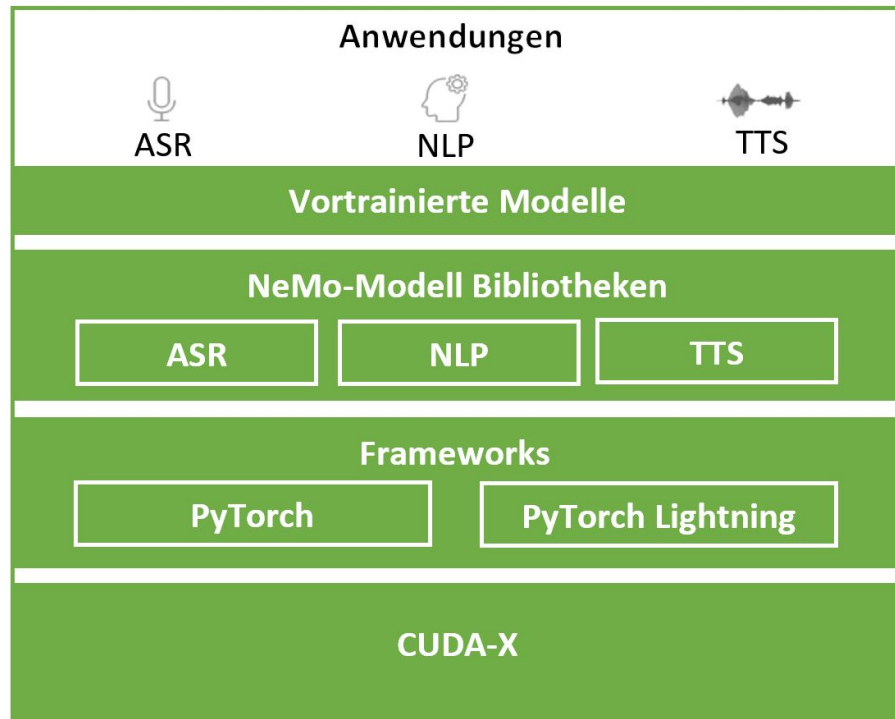


Abbildung 2.5: Darstellung unterschiedlicher Komponenten von NVIDIA NeMo. Für die Anwendung werden vortrainierte oder neue Modelle, mit den dazugehörigen Bibliotheken und Frameworks zum Training der unterschiedlichen Kommunikationsmodellen verwendet. Das Programmierungsmodell CUDA-X der GPU, ermöglicht eine effektive Skalierung sowie Feinabstimmung dieser Modelle (Quelle: In Anlehnung an [14])

## 2.5 Konvolutionale neuronale Netze

Ein konvolutionales neuronales Netz (engl. Convolutional Neural Networks, CNN) verwendet statt mehreren verborgenen Schichten wie beim DNN, gefilterte Schichten (engl. Convolution Layer) die in weitere Schichten, mit den sogenannten Pooling Layer, reduziert und am Ende des Prozesses wieder zusammengefügt und zum Ergebnis bewertet werden [16, Kap. 3].

Der Aufbau in Abbildung 2.6 ist eine schematische Darstellung eines CNN. Die Eingabe der Audiodatei wird als Spektrogramm in zweidimensionale Matrizen für die Convolution-Layer berechnet. Die Ansammlung dieser Matrizen werden Feature Maps genannt und werden mit der Pooling Schicht in weitere 2D Matrizen extrahiert, dieser Prozess kann mehrmals wiederholt werden. Zum Auswerten der Eingabe werden alle Informationen der Feature Maps, mit dem Flattening Prozess in eindimensionale Matrizen umgewandelt.

In der letzten Schicht der vollständig verbundenen Schicht (engl. Fully Connected Layer) dienen die eindimensionalen Matrizen als neuronales Netz zur Klassifizierung des Ergebnisses für die Ausgabeschicht. Am Ende wird eine Prognose des richtigen Ergebnisses in der Ausgabe ausgegeben [17].

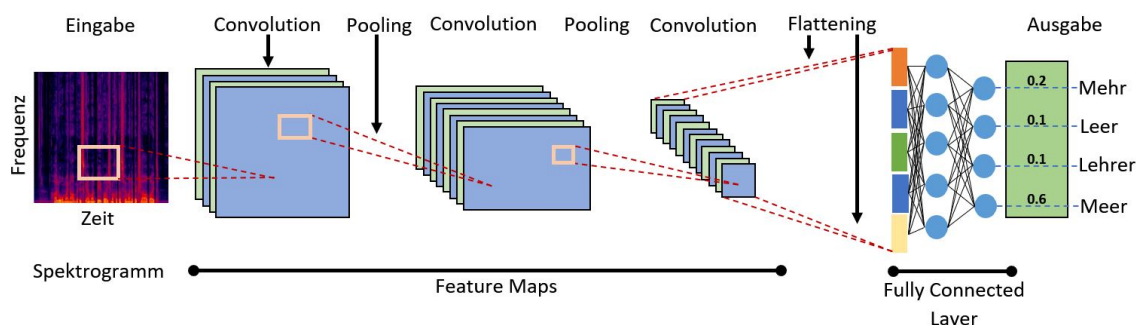


Abbildung 2.6: Schematische Darstellung eines konvolutionalen neuronalen Netzes für Audiodateien. Die Eingabe einer Audiodatei wird als Spektrogramm in mehreren Convolution-Layer in eine Feature Map gefiltert. Die Informationen aus der Feature Map werden mit dem Pooling Layer reduziert. Mit dem Flattening Prozess werden die Informationen aus den Feature Maps für den Fully Connected Layer lesbar gemacht. Die Ausgabe des Ergebnisses wird am Ende mit Hilfe der Gewichtung des neuronalen Netzes prognostiziert (Quelle: In Anlehnung an [17, Abb. 2] und [18])

## 2.6 Jasper

Das ASR-Modell von NVIDIA, Just Another Speech Recognizer (Jasper) [19], ist ein End-to-End konvolutionales neuronales Akustikmodell und verwendet eindimensionale Convolution-Layer, das Normalisierungsverfahren Batch-Normalization, die Aktivierungsfunktion Rectified Linear Unit (ReLU), die Regularisierungsmethode Dropout und die Schichtverbindung Dense Residual [19, Kap. 1].

## 1D Convolution-Layer

Jasper verwendet zum Training der Modelle eindimensionale statt zweidimensionale Convolution-Layer, um eine bessere Genauigkeit innerhalb der CNN zu erreichen [20, Kap. 5].

Die schematische Darstellung in Abbildung 2.7 zeigt das CNN mit dem 1D Convolution-Layer an. Der einzige Unterschied zur Abbildung 2.6 ist, dass das Spektrogramm vorerst in eine Eingabeschicht als Vektor oder eindimensionale Matrix gefiltert wird. Das Pooling und Flattening bis hin zur Prognose der Ausgabe sind die gleichen Prozesse.

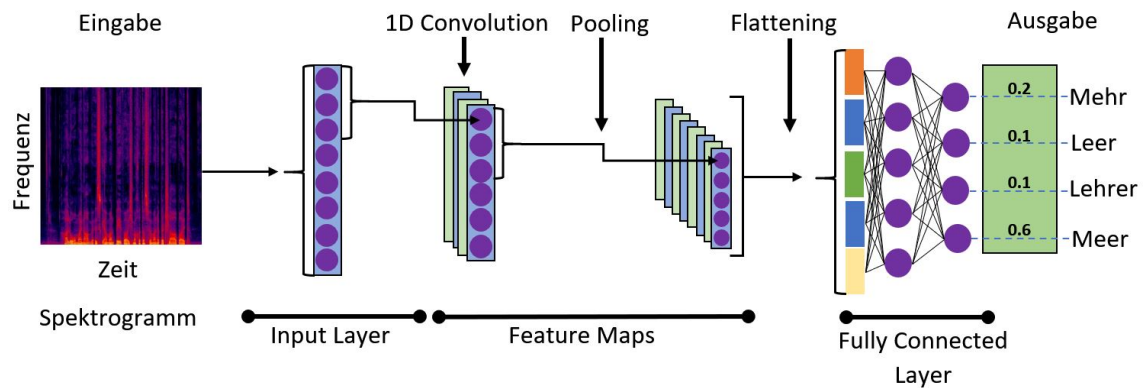


Abbildung 2.7: Schematische Darstellung eines 1D konvolutionalen neuronalen Netzes für Audiodateien. Die Eingabe einer Audiodatei wird als Spektrogramm in ein Input Layer als Vektor gefiltert, bevor es dann in mehreren Convolution-Layer in eine Feature Map extrahiert wird. Die Informationen aus der Feature Map werden mit dem Pooling Layer reduziert. Mit dem Flattening Prozess werden die Informationen aus den Feature Maps für den Fully Connected Layer zusammengefasst. Die Ausgabe des Ergebnisses wird am Ende mit Hilfe der Gewichtung des neuronalen Netzes prognostiziert (Quelle: In Anlehnung an [20, Abb. 2])

## Batch-Normalization

Die Batch-Normalization ist eine Normalisierung von Mittelwerten und Varianzen der Eingangsdaten basierend auf der Teilmenge der Batches während des Trainings [21]. Ein- und Ausgabedaten der Schichten erhalten mit der Standardnormalverteilung einen Mittelwert von null und die Varianz von eins, um das Training der Modelle zu beschleunigen [21, Kap. 3]. Bei einer Änderung der Ausgabe einer Schicht innerhalb eines neuronalen Netzes wird in den darauffolgenden Schichten eine neue

Verteilung der Gewichtung erlernt, dieses Problem ist als interne Kovariatenverschiebung bekannt und wird mit der Batch-Normalization reduziert [21, Kap. 2].

## ReLU

Beim Training eines neuronalen Netzes können die Werte der Neuronen unterschiedliche Zahlenwerte enthalten. Die Aktivierungsfunktion ReLu in Abbildung 2.8, sorgt für eine nichtlineare Umwandlung der Neuronenwerte. Mit ReLu werden alle positiven Eingaben nicht verändern, die negativen Eingaben werden auf null gesetzt [22, Kap. 2.4.2]. Die einfache Umrechnung der Neuronenwerte ermöglicht ReLu das Training eines neuronalen Netzes zu beschleunigen [22, Kap. 4].

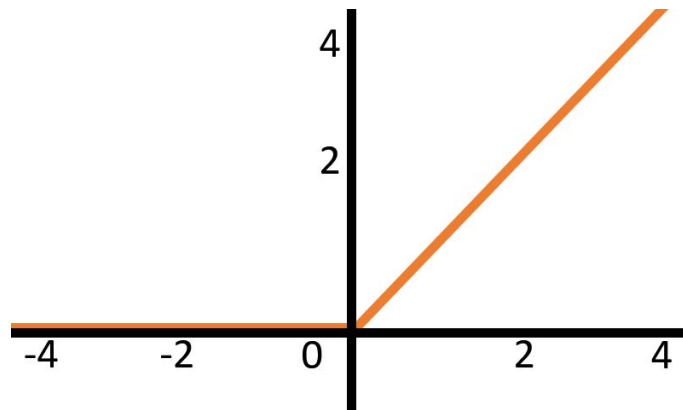


Abbildung 2.8: Aktivierungsfunktion ReLu im Funktionsgraph. Negative Eingaben werden mit ReLu auf null gesetzt. Bei einer positiven Eingabe wird der gleiche Wert beibehalten (Quelle: In Anlehnung an [22, Abb. 1])

## Dropout

Die Dropout-Methode wird verwendet, um eine Überanpassung (engl. Overfitting) innerhalb eines neuronalen Netzes zu vermeiden, d. h. dass das trainierte Modell keine spezifischen Features erlernt, um nicht zu komplex zu werden [23, Kap. 1]. Dropout wird während des Trainings bei jedem Trainingsschritt eingesetzt, bei der eine vorher angegebene Wahrscheinlichkeit verwendet wird, um bestimmte Neuronen nicht in Betracht zu ziehen. Beim Trainingsprozess mit den gleichen Datensätzen ohne Dropout, ist das Modell nur auf bestimmte Muster spezialisiert. Werden dem Modell neue Datensätze zum Lernen gegeben, können ggf. keine wichtigen Muster erkannt und dadurch falsche Ergebnisse geliefert werden. Mit Hilfe der Dropout-Methode werden beim Training gezielt Informationen nicht berücksichtigt und ermöglichen dem Modell unterschiedliche Muster zu erlernen [23, Kap. 7].

Abbildung 2.9 zeigt eine schematische Darstellung eines neuronalen Netzes. Auf der linken Seite werden die Informationen innerhalb aller Neuronen berücksichtigt. Das neuronale Netz auf der rechten Seite verwendet die Regulierungsmethode Dropout und zieht die durchgestrichenen Neuronen bei der Gewichtung nicht in Betracht.

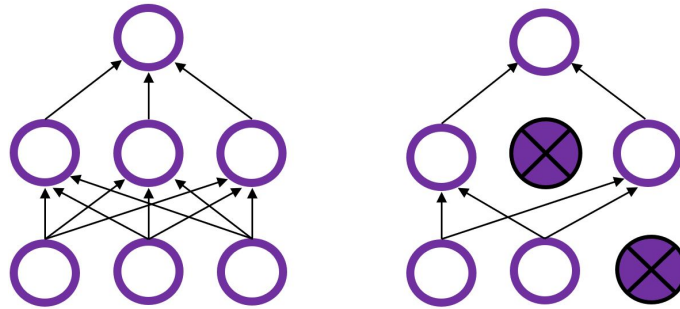


Abbildung 2.9: Schematische Darstellung eines neuronalen Netzes mit der Dropout-Methode. Auf der linken Seite ist ein neuronales Netz ohne Dropout. Das neuronale Netz mit der Regulierungsmethode Dropout ist rechts abgebildet. Durchgestrichene Neuronen werden für die Gewichtung nicht in Betracht gezogen (Quelle: In Anlehnung an [23, Abb. 1])

## Dense und Residual

Die Tiefe, Genauigkeit und die Effizienz des Trainings eines CNN ist mit kürzeren Verbindungen zwischen den jeweiligen Schichten kurz vor der Ein- und Ausgabe gewährleistet [24].

Die schematische Darstellung eines CNN in Abbildung 2.10 zeigt die Schichtverbindung mit Dense und Residual. Der Dense Block erstellt kurze Verbindungen innerhalb der Schichten, um die unterschiedlichen Features aus den vorherigen Schichten zu erlernen, ohne dabei die Schnelligkeit des Trainings zu beeinträchtigen [24, Abb. 2]. Eine Möglichkeit zum Erreichen einer besseren Tiefe für das CNN, ist die Verwendung von Residual Block, bei der eine oder mehrere Schichten übersprungen und am Ende der Ausgabe einer Schicht die jeweiligen Informationen hinzugefügt werden [24, Kap. 3].

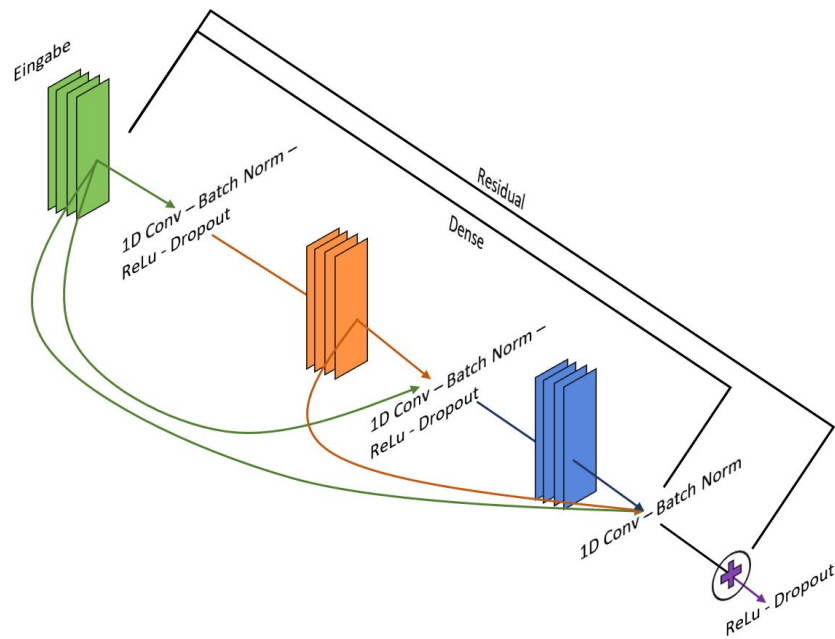


Abbildung 2.10: Schematische Darstellung eines CNN mit Dense Residual. Bei der Dense Methode erhält jede Schicht alle prognostizierten Feature-Maps als Eingabe der anderen Schichten. Die Residual Methode überspringt eine oder mehrere Schichten und fügt diese am Ende ein (Quelle: In Anlehnung an [24, Abb. 1] und [25, Abb. 1c])

## Modell-Architektur

In Abbildung 2.11 ist jeder der logischen Blöcke der Modell-Architektur von Jasper ein neuronales Modul. Die jeweiligen Methoden sind im Hinblick der Leistungsfähigkeit für das Training mit Hilfe der GPUs optimiert [19, Kap. 1]. Das Jasper BxR Modell hat B-Blöcke bestehend aus R wiederholenden Teilblöcken. Bei der Eingabe der Audiodatei als Spektrogramm wird für jeden Block die 1D-Convolution, die Batch-Normalization, ReLu sowie Dropout eingesetzt [19, Kap. 2 f.].

Der Prolog reduziert das Audiosignal für eine kürzere Verarbeitung und Effizienz des Modells [26]. Jede Eingabe innerhalb eines Blocks ist mit dem nächsten Block verbunden und wird innerhalb der Residual Methode durch ein 1D Convolution Filter projiziert, um unterschiedliche Informationen der Ein- und Ausgabeverbindung zu berechnen und diese dann durch die Batch-Normalization weiterzuleiten. Die Ausgabe nach der Batch-Normalization wird am Ende des Blocks hinzugefügt und das Ergebnis wird nach der Aktivierungsfunktion ReLu und nach dem Dropout an den nächsten Block weitergegeben. Dieser Prozess kann beliebig oft wiederholt werden.

Die Epilog-Blöcke CONV-BN-ReLu und der 1x1 CONV Block entsprechen den Fully Connected Layer. Der 1x1 CONV Block reduziert die Tiefe der jeweiligen Schicht und erhöht die Genauigkeit und die Geschwindigkeit des Modells, ohne dass eine fixierte Größe eingehalten werden muss.

Die Connectionist Temporal Classification (CTC) Schicht wird zum Training des neuronalen Netzes verwendet, dazu zählt das Berechnen des Verlusts, das Dekodieren der Ausgabe in ein Transkript sowie die Klassifizierung jeder einzelnen Audioausschnitte des Spektrogramms [28, Kap. 1]. Die Ausgabe des Modells ist eine Folge von Buchstaben, die der einzelnen Audioausschnitte entsprechen und dabei unabhängig der Anordnung zum richtigen Ergebnis führen kann. Das Vokabular besteht z. B. aus allen Alphabeten, Leerzeichen und dem Apostroph-Symbol, also insgesamt 29 Symbolen einschließlich des Leersymbols, das von der CTC-Verlustfunktion verwendet werden kann. In Abbildung 2.12 wird mit CTC aus dem Spektrogramm eine Folge von Buchstaben erkannt, die nach mehrmaligen Durchläufen zum gewünschten Ergebnis führt.

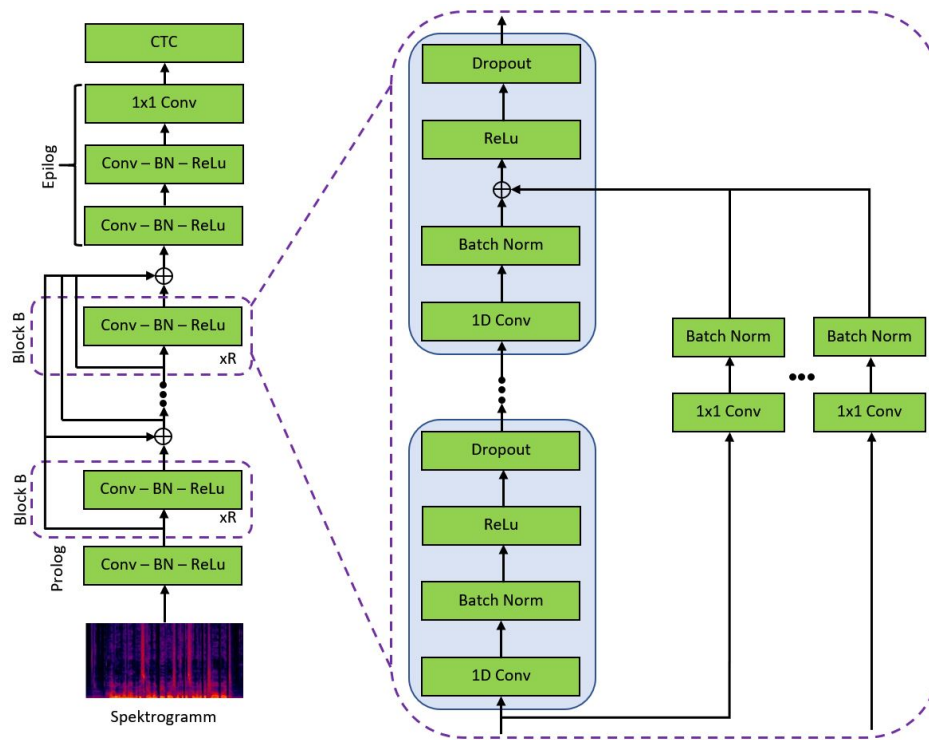


Abbildung 2.11: Schematische Darstellung der BxR Modell-Architektur von Jasper. Im Prolog wird die Audiodatei für eine effizientere Verarbeitung gekürzt. Jeder Block hat die folgenden Methoden: 1D-Convolution, die Batch-Normalization, ReLu sowie Dropout. Innerhalb des Modells verwenden die unterschiedlichen Blöcke die Dense und Residual Methoden. Diese Prozesse können beliebig oft wiederholt werden. Der Epilog Block entspricht dem Fully Connected Layer. Die Informationen des Epilog-Blocks werden an die CTC Schicht weitergegeben. Die letzte Schicht ordnet die Audio und Text Informationen zu (Quelle: In Anlehnung an [19, Abb. 1])

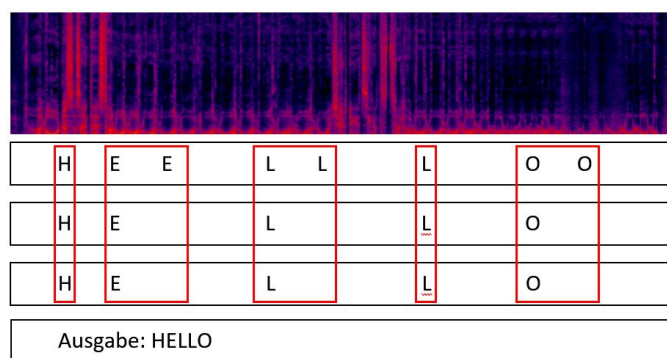


Abbildung 2.12: CTC Beispiel mit einer Audiodatei. Die Audioeingabe wird mit den vorgegeben Buchstaben prognostiziert und verzichtbare Buchstaben werden für die Ausgabe verworfen (Quelle: In Anlehnung an [29])

## 3 Konzeption

In diesem Kapitel werden die verwendeten Technologien betrachtet, die im Kapitel Stand der Technik erwähnt wurden. Die eingesetzten Softwarebibliotheken zur Erstellung, dem Training und der Evaluation des Sprachmodells werden genauer beschrieben. Daran angeknüpft wird ein Einblick der Jasper WER-Resultate bezüglich anderer Testdatensätze in Englisch gegeben. Abschließend werden die jeweiligen Anforderungen an dem zu entwickelnden Modell in der Bachelorarbeit für die deutsche Sprache aufgezeigt.

### 3.1 Technologien

Innerhalb dieser Bachelorarbeit wird das Toolkit NVIDIA NeMo auf Basis von PyTorch und PyTorch Lightning sowie dem Jasper Modell zur automatischen Spracherkennung für die deutsche Sprache verwendet.

#### 3.1.1 NVIDIA NeMo

NVIDIA NeMo umfasst das Erstellen, Trainieren und die Feinabstimmung mehrerer Modelle zur Spracherkennung. Das Toolkit ermöglicht den Entwickler das Entwickeln und Training für effektive Sprach- und Sprechmodelle, die mit mehrmaligen Durchläufen der Testdatensätze eine hohe Genauigkeit erzielen. Mit Hilfe des modularen, einfachen und flexiblen Aufbaus von NeMo, können mit wenig Code mehrere Modelle realisiert werden.

Quellcode 3.1 zeigt das Einbinden von NeMo und der benötigten NeMo-Modell Bibliotheken, z. B. für ASR, NLP und TTS [30].

```
1 # Import NeMo
2 import nemo
3 # Import ASR
4 import nemo.collections.asr as nemo_asr
5 # Import NLP
6 import nemo.collections.nlp as nemo_nlp
```

```

7 # Import TTS
8 import nemo.collections.tts as nemo_tts

```

Quellcode 3.1: Codebeispiel zum Einbinden der Nemo-Modell Bibliotheken [30, Kap. Quick Start]

### 3.1.2 PyTorch und PyTorch Lightning

PyTorch ist eine Bibliothek für Python Programme zum Implementieren von Deep Learning Projekten. Es ermöglicht den Entwicklern durch die Einfachheit, die Prozesse schnell zu erlernen, benutzen, erweitern und zu debuggen [31, Kap. 1.2]. Zum Training der unterschiedlichen komplexen Modelle wird PyTorch mit Hilfe der beschleunigten Berechnungen der GPUs sowie der numerischen Optimierung von generischen mathematischen Ausdrücken, die für das Deep Learning essenziell sind, zur Beschleunigung der Lernprozesse verwendet [31, Kap. 1.3].

PyTorch Lightning ist eine Schnittstelle zu PyTorch, die das Skalieren der Modelle auf mehreren GPUs ermöglicht und den Boilerplate-Code überarbeitet [32]. Mit Lightning kann der PyTorch Code in unterschiedliche Module organisiert werden, sodass dieser speziell bei komplexen Modellen leichter zu gestalten ist und somit die Produktivität beim Deep Learning erhöht.

In NeMo wird die Lightning Programmierschnittstelle (engl. Application Programming Interface, API) LightningModule sowie der Trainer verwendet. Das LightningModule ist ein neuronales Netzmodul und organisiert den PyTorch Code in fünf Abschnitte [33] (siehe entsprechende Zeile im Quellcode 3.2):

- Neuronale Berechnung der Informationen (init - Zeile: 4 - 6)
- Trainingsschleife der Datensätze (forward und training step - Zeile: 8 - 15)
- Validierungsschleife (validation step - Zeile: 17 - 21)
- Testschleife (test step - Zeile: 23 - 27)
- Optimierung (configure optimizers - Zeile: 29 - 30)

```

1 import pytorch_lightning as pl
2 class LitModel(pl.LightningModule):
3

```

```

4  def __init__(self):
5      super().__init__()
6      self.l1 = nn.Linear(28 * 28, 10)
7
8  def forward(self, x):
9      return torch.relu(self.l1(x.view(x.size(0), -1)))
10
11 def training_step(self, batch, batch_idx):
12     x, y = batch
13     y_hat = self(x)
14     loss = F.cross_entropy(y_hat, y)
15     return loss
16
17 def validation_step(self, batch, batch_idx):
18     x, y = batch
19     y_hat = self.model(x)
20     loss = F.cross_entropy(y_hat, y)
21     self.log('val_loss', loss)
22
23 def test_step(self, batch, batch_idx):
24     x, y = batch
25     y_hat = self.model(x)
26     loss = F.cross_entropy(y_hat, y)
27     self.log('test_loss', loss)
28
29 def configure_optimizers(self):
30     return torch.optim.Adam(self.parameters(), lr=0.02)

```

Quellcode 3.2: Codebeispiel der LightningModules [33]

Sobald die LightningModules organisiert wurden, wird der Lightning Trainer verwendet. Der Trainer unterstützt automatisch das Training, Validation sowie Tests des neuronalen Netzes und kontrolliert bei jedem Schleifendurchlauf die Gewichtung der Neuronen und kann je nach Situation das Overfitting mittels einem Stopp verhindern [34]. Quellcode 3.3 zeigt ein grundlegendes Beispiel eines Lightning Trainers.

```

1 model = MyLightningModule()
2 trainer = Trainer()
3 trainer.fit(model, train_dataloader, val_dataloader)

```

Quellcode 3.3: Codebeispiel Lightning Trainer [34]

### 3.1.3 Jasper

NVIDIA NeMo stellt das Jasper10x5Dr Modell, 10 Blöcke mit je 5 wiederholenden Unterblöcken zur Verfügung. Quellcode 3.4 zeigt die Methode zum Verwenden des Jasper Modells für die englische Sprache an. Das Model wurde auf Basis sechs unterschiedlichen englischen Datensätzen mit einer Gesamtzeit von über 3000 Stunden trainiert: LibriSpeech, Mozilla Common Voice, Wall Street Journal, Fisher, Switchboard und NSC Singapore English.

```
1 #Vortrainiertes Jasper Model
2 model = nemo_asr.models.EncDecCTCModel.
3     from_pretrained(model_name="Jasper10x5Dr-En")
```

Quellcode 3.4: Codebeispiel zum Einbinden eines vortrainierten Jasper Modells [30, Kap. NeMo Models]

Innerhalb des Modells können unterschiedliche Parameter in einer Konfigurationsdatei zum Training angepasst werden, z. B. das Einfügen weiterer Buchstaben, das Auswählen anderer Aktivierungsfunktionen, das Unterteilen der Datensätze in Batches, um diese in einer Epoche durchlaufen lassen zu können. Das vortrainierte Modell im Quellcode 3.4 erreicht mit einer Batch-Size von 64 und einer Epoche von 100 eine Wortfehlerrate von 3.37%.

## 3.2 Anforderungen

Im Rahmen dieser Bachelorarbeit wird ein nicht-vortrainiertes und ein vortrainiertes englische Jasper Modell für die deutsche Sprache angelernt. Das Ziel ist es, mit den deutschen Mozilla Common Voice Datensätzen das Jasper Modell mit unterschiedlichen Epochen zu testen, um eine geeignete Wortfehlerrate zu erreichen. Die Auswertung der jeweiligen Ergebnisse erfolgt im Kapitel Evaluation.

### 3.2.1 Wortfehlerrate

Als Maßstab der Wortfehlerrate werden die aktuellen Werte aus dem Jasper10x5Dr Modell betrachtet. Aus dem Paper *Jasper: An End-to-End Convolutional Neural Acoustic Model* wurden die Jasper Modelle mit unterschiedlichen Blöcken sowie Methoden miteinander getestet [19]. Das kleinere Jasper10x4 Modell liefert mit der Batch-Norm und ReLu bei 50 Durchläufen die besten Ergebnisse [19, Tab. 2]. Mit einem 10x3 Jasper Modell und 400 Durchläufen sind die unterschiedlichen Aktivierungsfunktionen beinahe identisch [19, Tab. 4].

Tabelle 3.1 zeigt die Wortfehlerrate mit den LibriSpeech Datensätzen. Das Jasper10x5Dr Modell wurde von den Autoren mit unterschiedlichen Sprachmodellen (engl. Language Model, LM) getestet. Die Sprachmodelle erfassen eine Abhängigkeit zwischen den Wörtern innerhalb des Lernprozesses. Ein Jasper10x5Dr Modell, dass kein LM verwendet, ist bei einer Wortfehlerrate unter 4%.

Das Verwenden der Sprachmodelle verbessert die Testresultate je nach Datensatz um 1-4%. Dev-Clean und Dev-Other sind Datensätze, die ausschließlich beim Trainingsprozess verwendet werden. Clean sind z. B. saubere, einfache sowie langsam gesprochene Aufnahmen. Other sind Datensätze, die teilweise mit schwer verstehenden Akzent oder Hintergrundgeräuschen aufgenommen wurden. Test-Clean und Test-Other sind Datensätze, die dem Modell nicht bekannt sind und zur endgültigen Überprüfung der Wortfehlerrate verwendet werden.

Modell	LM	dev-clean	dev-other	test-clean	test-other
Jasper10x5Dr	-	3.64	11.89	3.86	11.95
Jasper10x5Dr	6-gram	2.89	9.53	3.34	9.62
Jasper10x5Dr	Transformer-XL	2.68	8.62	<b><u>2.95</u></b>	8.79

Tabelle 3.1: Jasper10x5Dr LibriSpeech Wortfehlerrate [19, Tab. 5]

## 4 Implementierung

Im folgenden Kapitel wird auf die Umwandlung der Trainingsdatensätze für NeMo sowie die Implementierung zum Training der Jasper Modelle eingegangen. Im Anschluss wird das Skript für die Ermittlung der Wortfehlerrate vorgestellt.

### 4.1 Trainingsdatensätze

Zur Verwendung der deutschen Mozilla Common Voice Datensätze, werden die vorgegebenen .mp3 Sounddateien in das für NeMo erwartete .wav Format mit Hilfe der Python Bibliothek SoX umgewandelt [35]. Der Aufbau innerhalb einer Manifest-Datei im Quellcode 4.1 zeigt ein Trainingsbeispiel mit dem Pfad zur .wav Sounddatei, der jeweiligen Dauer in Sekunden sowie dem Text als Transkript.

```
1 {"audio_filepath": "<pfad_zu>/common_voice_de_21346099.wav",  
2 "duration": 4.704,  
3 "text": "aber wir verstehen ja die haushaltslage"}
```

Quellcode 4.1: Aufbau der Manifest-Datei für NeMo

Zur Erstellung der Manifest-Datei werden die im Common Voice mitgegebenen Tab Separated Values (TSV) Dateien für das Training, Validieren und Testen mit einem Python-Skript umgewandelt. Die Informationen der TSV Datei beinhalten den Pfad zur .mp3 Sounddatei, den dazugehörigen gesprochenen Satz sowie weitere Informationen, die für den Trainingsprozess nicht erforderlich sind, wie z. B. das Geschlecht der Person.

Mit dem Skript aus Quellcode 4.2 wird eine TSV Datei ausgelesen, die jeweiligen .mp3 Sounddateien werden in ein 16000 Kilohertz .wav Format umgewandelt, die Dauer der Sounddatei ermittelt und anschließend der Text normalisiert. Die daraus resultierenden Ergebnisse werden im JavaScript Object Notation (JSON) Format gespeichert.

```

1 #Erstellt aus einer vom User ausgewählten .tsv Datei die
2 #dazugehörige Manifest-Datei
3 def tsv_to_manifest(tsv_files, manifest_file, prefix):
4     manifests = []
5     #Liest alle Zeilen der mitgegeben .tsv Datei aus
6     for tsv_file in tsv_files:
7         dt = pd.read_csv(tsv_file, sep='\t', encoding='utf8')
8         for index, row in dt.iterrows():
9             try:
10                 entry = {}
11                 #Erstellt den Ordner für die .wav Dateien
12                 os.system("mkdir -p wavs/{0}".format(prefix))
13                 mp3_file = "clips/" + row['path']
14                 wav_file = "wavs/{0}/".format(prefix) + row['path'] +
15                     ".wav"
16
17                 #Verwendung der SoX Bibliothek zur Umwandlung der
18                 #.mp3 Datei in eine .wav Datei mit 16kHz
19                 subprocess.check_output("sox {0} -c 1 -r 16000
20                     {1}".format(mp3_file, wav_file), shell=True)
21
22                 #Dauer der Sounddatei wird ermittelt
23                 duration = subprocess.check_output(
24                     "soxi -D {0}".format(wav_file), shell=True)
25
26                 #Der Pfad der .wav Datei, die Dauer und der Text
27                 #werden in ein Array gespeichert
28                 entry['audio_filepath'] = wav_file
29                 entry['duration'] = float(duration)
30
31                 #Die normalize_str Funktion vergleicht und ersetzt
32                 #alle Zeichen die nicht im deutschen Alphabet sind
33                 entry['text'] = normalize_str(row['sentence'])
34                 manifests.append(entry)
35
36             except:
37                 print("SOMETHING WENT WRONG - IGNORING ENTRY")
38
39 #Ausgabe der Manifesteinträge in eine .json Datei
40 with codecs.open(manifest_file, 'w',

```

```
40 encoding='utf-8') as fout:
41     for m in manifests:
42         fout.write(json.dumps(m, ensure_ascii=False) + '\n')
```

Quellcode 4.2: Codebeispiel zur Umwandlung der Trainingsdatensätze sowie die Erstellung der Manifestdatei [36]

## 4.2 Konfigurationsdatei

Die Konfigurationsdatei beschreibt die jeweilige Architektur des Jasper Modells, die im menschenlesbaren YAML Format angezeigt wird. Innerhalb dieser Datei können je nach Sprache das Alphabet angepasst, die Batch-Size zum Training geändert sowie weitere Optionen vorgenommen werden, die die Leistung des Modells beeinflussen. Weiterhin kann jeder der Jasper Blöcke sowie Unterblöcke in der Konfigurationsdatei optimiert werden. Hierzu zählt das Modifizieren der Dropoutrate, die Verwendung von Residual sowie das Festlegen der Anzahl der Wiederholungen für bestimmte Blöcke.

## 4.3 Modelltraining

Im Verlauf dieser Bachelorarbeit wird ein Jasper Modell ohne Vorkenntnisse und ein vortrainiertes englisches Jasper Modell mit dem deutschen Trainingsdatensatz von Common Voice trainiert. Die Ergebnisse der Wortfehlerrate und Verlustwerte beim Training werden im Kapitel Evaluation dargestellt. Das Skript in Quellcode 4.3 beinhaltet mit wenig Zeilen Code, das Nötigste zum Training des Jasper Modells.

Bevor das Training mit NeMo gestartet werden kann, werden in Zeile 2 - 3 die entsprechenden Bibliotheken geladen. In den Zeilen 6 - 10 werden die jeweiligen Parameter aus der Konfigurationsdatei geladen. Daraufhin wird die PyTorch-Lightning Bibliothek importiert und der Trainer mit der gewünschten GPU Anzahl und den Epochen erstellt. Innerhalb der Zeilen 17 - 21 werden die Pfade zu den Trainingsdatensätzen für die Manifestdatei festgelegt. In Zeile 25 wird das entsprechende Jasper-Modell mit den Parametern der Konfigurationsdatei und dem Trainer instanziiert. Nach der Instanziierung kann das Modelltraining in Zeile 30 gestartet und nach dem Training in Zeile 33 gespeichert werden.

```

1 #Import der benötigten NeMo Bibliotheken
2 import nemo
3 import nemo.collections.asr as nemo_asr
4
5 #Lesen und Laden der Konfigurationsdatei
6 from ruamel_yaml import YAML
7 config_path = '<path_to_config>/jasper_10x5dr_ger.yaml'
8 yaml = YAML(typ='safe')
9 with open(config_path) as f:
10 params = yaml.load(f)
11
12 #ASR Trainer mit der jeweiligen Epochenanzahl erstellen
13 import pytorch_lightning as pl
14 asr_trainer = pl.Trainer(gpus=[1], max_epochs=25)
15
16 #Die Manifestdatei zu den Trainingsdatensätzen festlegen
17 from omegaconf import DictConfig
18 train_manifest = 'train.json'
19 validation_manifest = 'validation.json'
20 params['model']['train_ds']['manifest_filepath'] =
    train_manifest
21 params['model']['validation_ds']['manifest_filepath'] =
    validation_manifest
22
23 #Das Jasper Modell mit den jeweiligen Parametern
24 #instanciieren
25 jasper_asr_model = nemo_asr.models.EncDecCTCModel(
26     cfg=DictConfig(params['model']),
27     trainer=asr_trainer)
28
29 #Training starten
30 asr_trainer.fit(jasper_asr_model)
31
32 #Nach dem Training das Jasper Modell speichern
33 asr_trainer.save_to('<path_to_save_model>')

```

Quellcode 4.3: Codebeispiel zum Training eines Jasper Modells [37]

## 4.4 Transfer-Lernen

Das Transfer-Lernen wird verwendet, wenn die Anzahl der Datensätze für eine Sprache gering ist oder der Speicher für eine GPU nicht ausreicht [38]. Mit der Methode wird dem vortrainiertem Modell ein neues Alphabet mit einer anderen Sprache zum Training mitgegeben [39].

Das Skript aus Quellcode 4.3 wird für das Transfer-Lernen leicht modifiziert. Anstelle der Instanziierung eines neuen Jasper Modells wird in Zeile 2 - 3 in Quellcode 4.4 das englische Jasper10x5Dr Modell [40] geladen. Die Zeilen 6 - 10 ändern das Alphabet des geladenen englischen Modells auf Deutsch. In Zeile 14 - 17 werden dem Modell die Pfade aus der Konfigurationsdatei für die deutschen Trainings- und Validierungsdatensatz mitgegeben. In Zeile 21 und 22 wird das Training gestartet und am Ende des Trainings das Modell gespeichert.

```

1 #Das vortrainierte englische Modell laden
2 jasper_asr_model = nemo.asr.models.EncDecCTCModel
3   .from_pretrained(model=name"stt_en_jasper10x5dr")
4
5 #Das Alphabet des Jasper Modells ändern
6 jasper_asr_model.change_vocabulary(
7   new_vocabulary=[
8     " ", "a", "b", "c", "d", "e", "f", "g", "h", "i", "j",
9     "k", "l", "m", "n", "o", "p", "q", "r", "s", "t", "u",
10    "v", "w", "x", "y", "z", "ä", "ö", "ü", "ß", " '"]
11
12 #Dem Modell den deutschen Trainings- und
13 #Validierungsdatensatz mitgeben
14 jasper_asr_model.setup_training_data(
15   train_data_config=params['model']['train_ds'])
16 jasper_asr_model.setup_validation_data(
17   val_data_config=params['model']['validation_ds'])
18
19 #Training starten und nach dem Training das
20 #Jasper Modell speichern
21 asr_trainer.fit(jasper_asr_model)
22 asr_trainer.save_to('<path_to_save_model>')
```

Quellcode 4.4: Codebeispiel zum Transfer-Lernen eines vortrainierten englischen Jasper Modells mit dem deutschen Alphabet und den deutschen Datensätzen [37]

## 4.5 Wortfehlerrate

Zur Erkennung der Wortfehlerrate der Modelle werden die Testdatensätze aus Mozilla Common Voice mit dem Skript aus Quellcode 4.5 verwendet, um saubere Testergebnisse zu erhalten.

Mit dem Dataloader in Zeile 18 - 24 erhält man das Audiosignal, die Audiosignallänge, das Transkript und die Transkripttoken aus allen Test Batches. Das Hilfsobjekt zum Berechnen der Wortfehlerrate in Zeile 29 - 32, sammelt alle Einfügungen, Vertauschungen und Auslassungen der Test Batches in den Zähler und Nenner Array von Zeile 12 und 13. In Zeile 36 wird die Anzahl der Zähler und Nenner summiert, anschließend dividiert und die Wortfehlerrate ausgegeben.

```

1 #Test-Manifestdatei mit der Batch-Size 32 ausgewählt
2 model_config['model']['validation_ds']['batch_size'] = 32
3
4 #Konfiguration des Dataloaders für das Modell auf einer GPU
5 asr_model.setup_test_data(
6 test_data_config=model_config['model']['validation_ds'])
7 asr_model.cuda()
8
9 #Wortfehlererkennung zwischen Annahme und Prognose der
10 #Wörter. Zähler und Nenner für die WER aus den jeweiligen
11 #Test Batches werden im Array gesammelt
12 wer_nums = []
13 wer_denoms = []
14
15 #Loop über alle Test Batches. Wir erhalten das Audiosignal,
16 #die Audiosignallänge, das Transkript und die
17 #Transkripttoken
18 for test_batch in asr_model.test_dataloader():
19     test_batch = [x.cuda() for x in test_batch]
20     targets = test_batch[2]
21     targets_lengths = test_batch[3]
22     log_probs, encoded_len, greedy_predictions = asr_model(
23         input_signal=test_batch[0],
24         input_signal_length=test_batch[1]
25     )
26

```

```
27 #Das Modell hat ein Hilfsobjekt zum Berechnen
28 #der Wortfehlerrate
29 asr_model._wer.update(greedy_predictions, targets,
    targets_lengths)
30 _, wer_num, wer_denom = asr_model._wer.compute()
31 wer_nums.append(wer_num.detach().cpu().numpy())
32 wer_denoms.append(wer_denom.detach().cpu().numpy())
33
34 #Die angesammelten Zähler und Nenner summieren,
35 #am Ende dividieren und ausgeben
36 print(f"WER = {sum(wer_nums)/sum(wer_denoms)}")
```

Quellcode 4.5: Codebeispiel zum Berechnen der Wortfehlerrate [37]

# 5 Evaluation

In diesem Kapitel werden die Ergebnisse während des Modelltrainings und die entsprechende Wortfehlerrate aufgezeigt.

## 5.1 Ergebnisse

Das Training der Jasper10x5Dr Modelle erfolgte auf der GPU NVIDIA Tesla V100S mit 32 GB RAM, einer Batch Size von 32 sowie einer Epochenanzahl von 25 und 50.

### 5.1.1 Modelltraining

Mit dem Skript aus Quellcode 4.3 und Quellcode 4.4 wurden drei Jasper Modelle mit den Trainings- und Validierungsdatensatz von Mozilla Common Voice trainiert. Der Trainingsdatensatz ist der eigentliche Datensatz, der zum Training der Modelle verwendet wird. Der unterschiedliche Validierungsdatensatz setzt kurz vor Ende der jeweiligen Epoche im Training ein, um dem Modell eine voreingenommene Auswertung der Trainingsdaten zu ermöglichen. Diese Datenstichprobe, die Hyperparameteroptimierung [41] genannt wird, erlaubt es bestimmte Parameter innerhalb des Modells anzupassen und dadurch das Training des Modells zu verbessern.

Der Trainingsdatensatz hat 246.520 Audiodateien die insgesamt 393 Stunden entsprechen. Der Validierungsdatensatz hat 15.587 Audiodateien die insgesamt 25 Stunden entsprechen. Bei der ausgewählten Batch Size von 32 wird mit Hilfe von NeMo pro Epoche ein Datensatz von 8192 für den Trainingsdatensatz und 488 für den Validierungsdatensatz verwendet.

Während des Trainings wird jeweils die Epoche, der Fortschrittsbalken der jeweiligen Datensätze, die entsprechende Dauer der Iteration und der Verlust (engl. Loss) ausgegeben. Ziel ist es, einen möglichst geringen Verlustwert zum Ende des Trainings zu erhalten. Je geringer der Verlustwert, desto besser ist der Wert der Wortfehlerrate sowie die Prognose der Spracherkennung sowie der Buchstaben.

## Neues Modell mit 25 Epochen

In Abbildung 5.1 werden die Verlustwerte für ein neues Jasper10x5 Modell ohne Vorkenntnisse der deutschen Sprache in einem Liniendiagramm dargestellt. Die Y-Achse bildet die Verlustwerte des Modells von 700 bis 0 ab. Auf der X-Achse ist die Anzahl der Datensätze beim Training von 0 bis 200.000 dargestellt. Das Modell wurde mit 25 Epochen trainiert, mit je 8196 Datensätzen pro Epoche.

Der Anfangswert beim Training liegt bei 697 und hat am Ende des Trainings einen Verlustwert von 11.7. Das Liniendiagramm weist beim Trainingsbeginn einen erhöhten Verlustwert auf, der sich relativ schnell nach der ersten Epoche auf 68 reguliert. Beim fortlaufenden Training fällt der Verlustwert kontinuierlich ab. Teilweise sind Schwankungen der Werte zu erkennen, die sich nicht stark auf das Training auswirken lassen. Grund dafür könnte das Verwenden von neuen Datensätzen während des Trainings sein.

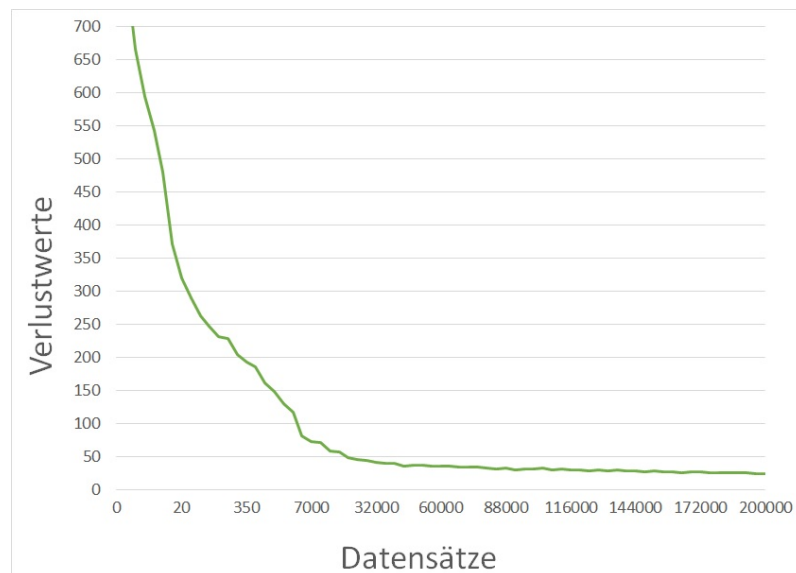


Abbildung 5.1: Schematische Darstellung der Verlustwerte für ein neues Jasper10x5Dr Modell ohne Vorkenntnisse der deutschen Sprache. Die Verlustwerte des Trainings sind auf der Y-Achse abgebildet. Die Anzahl der Trainingsdatensätze auf der X-Achse. Insgesamt wurde mit 25 Epochen trainiert, mit je 8196 Datensätzen pro Epoche

## Neues Modell mit 50 Epochen

In Abbildung 5.2 werden die Verlustwerte ebenfalls für ein neues Jasper10x5 Modell ohne Vorkenntnisse der deutschen Sprache mit 50 Epochen in einem Liniendiagramm dargestellt. Die Y-Achse bildet die Verlustwerte des Modells von 800 bis 0 ab. Auf der X-Achse ist die Anzahl der Datensätze beim Training von 0 bis 400.000 dargestellt. Der Anfangswert beim Training liegt bei 815 und hat am Ende des Trainings einen Verlustwert von 9.84.

Wie beim vorherigen Liniendiagramm, ist beim Trainingsbeginn ein erhöhter Verlustwert zu erwarten, der sich relativ schnell nach der ersten Epoche auf 71 reguliert. Beim fortlaufenden Training fällt der Verlustwert kontinuierlich ab. Innerhalb der Epochen treten die Schwankungen der Werte zwar häufiger auf, jedoch wird das Training nicht stark beeinflusst.

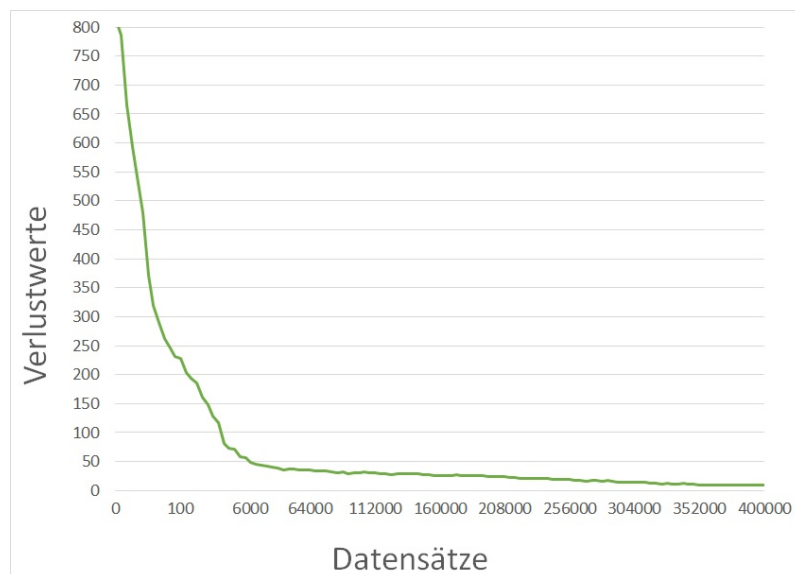


Abbildung 5.2: Schematische Darstellung der Verlustwerte für ein neues Jasper10x5Dr Modell ohne Vorkenntnisse der deutschen Sprache. Die Verlustwerte des Trainings sind auf der Y-Achse abgebildet. Die Anzahl der Trainingsdatensätze auf der X-Achse. Insgesamt wurde mit 50 Epochen trainiert, mit je 8196 Datensätzen pro Epoche

## Vortrainiertes Modell mit 25 Epochen

Das vortrainierte englische Jasper10x5 Modell wurde mit dem gleichen deutschen Trainingsdatensatz trainiert. Die Ergebnisse der Verlustwerte werden in Abbildung 5.3 in einem Liniendiagramm dargestellt. Das Modell hat vor dem Training eine Aktualisierung des Alphabets von Englisch auf Deutsch erhalten.

Die Y-Achse bildet die Verlustwerte des Modells von 700 bis 0 ab. Auf der X-Achse ist die Anzahl der Datensätze beim Training von 0 bis 200.000 dargestellt. Der Anfangswert beim Training liegt bei 702 und hat am Ende des Trainings einen Verlustwert von 32.7. Nach der ersten Epoche hat sich der Verlustwert auf 67.2 reguliert. Beim fortlaufenden Training fällt der Verlustwert zwar kontinuierlich ab, jedoch ändert sich dieser ab der 13. Epoche nicht mehr stark.

Im Gegensatz zu den vorherigen Modellen, indem ein Abstieg von Epoche zu Epoche erkennen war, ist bei diesem vortrainiertem Modell ein konstanter Verlustwert von 33 zu erfassen. Dieser ist gegebenenfalls durch Overfitting begründet und könnte sich bei einer höheren Epochenanzahl deutlicher im Diagramm darstellen lassen.

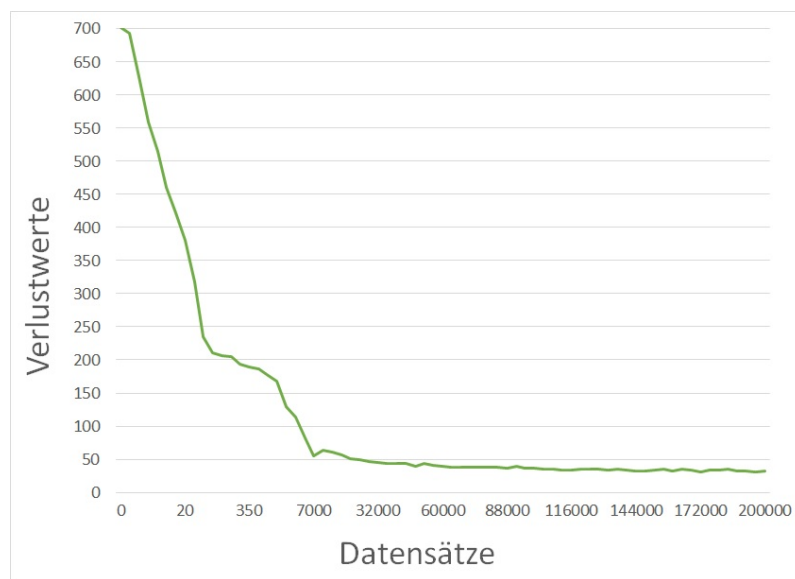


Abbildung 5.3: Schematische Darstellung der Verlustwerte für ein vortrainiertes englisches Jasper10x5Dr Modell mit deutschen Datensätzen. Die Verlustwerte des Trainings sind auf der Y-Achse abgebildet. Die Anzahl der Trainingsdatensätze auf der X-Achse. Insgesamt wurde mit 25 Epochen trainiert, mit je 8196 Datensätzen pro Epoche

### 5.1.2 Wortfehlerrate

Mit dem Skript in Quellcode 4.5 werden die jeweiligen Modelle auf die Wortfehlerrate mit dem Test- und Trainingsdatensätzen von Mozilla Common Voice geprüft.

In der Tabelle 5.1 werden die Ergebnisse der Jasper10x5Dr Modelle mit den Vorkenntnissen, Epochen und den beiden Wortfehlerraten in Prozent für den Test- und Trainingsdatensatz angezeigt.

Das erste Modell ohne Vorkenntnisse und einer Epoche von 25 wurde mit einer Wortfehlerrate von 31.65% mit dem Trainingsdatensatz und für den Testdatensatz von 43.19% geprüft.

Eine Verbesserung der Wortfehlerrate wurde beim zweiten Modell ohne Vorkenntnisse bei einer Epoche von 50 beobachtet. Die Wortfehlerrate beim Trainingsdatensatz liegt bei 28.80% und beim Testdatensatz bei 41.45%, dies entspricht eine Steigerung von 3.15% und 1.74%.

Das letzte vortrainierte Modell mit der englischen Sprache erzielte eine Wortfehlerrate von 52.20% und 62.50% bei einer Epoche von 25 für denselben Trainings- und Testdatensatz.

Modell	Vorkenntnisse	Epochen	train-clean WER %	test-clean WER %
Jasper10x5Dr	-	25	31.65	43.19
Jasper10x5Dr	-	50	28.80	41.45
Jasper10x5Dr	Englisch	25	52.20	62.50

Tabelle 5.1: Jasper10x5 Common Voice Resultate der Wortfehlerrate

## 6 Fazit und Ausblick

Im Rahmen dieser Arbeit wurden unterschiedliche ASR-Modelle mit Hilfe des Open-Source-Toolkits NVIDIA NeMo und den frei verfügbaren deutschen Sprachdatensätzen von Mozilla Common Voice erstellt, trainiert und evaluiert. Es wurden wichtige Grundlagen für den modernen Ansatz der Spracherkennung vorgestellt und aufgezeigt, wie komplex das maschinelle Lernen ist, speziell für die künstlichen neuronalen Netze. Das Anpassen der Prozesse und Methoden innerhalb der Jasper Konfigurationsdatei, erlaubt es dem Benutzer unterschiedliche Feinabstimmungen für das Training durchzuführen.

Ziel dieser Bachelorarbeit war der Versuch, eine möglichst geringe Wortfehlerrate zu erzielen und diese mit den kommerziellen ASR-Modellen zu vergleichen. Zwar wurde kein optimales Ergebnis erbracht, dennoch dienen die Informationen aus der Evaluation als Orientierung für das Training zukünftiger Modelle. Anhand der Modell-Architektur von Jasper in Abbildung 2.11 und den Versuchen vom Paper [19], ist zu erkennen, dass das Untersuchen eines idealen Modells ein zeitaufwendiger Prozess ist.

Angesichts der eingeschränkten Zeit der Bachelorarbeit ist das Erreichen einer guten Wortfehlerrate nicht unmöglich, jedoch sollten einige Faktoren vor dem Training optimiert werden. Das Experimentieren der Epochenanzahl und der Batch-Size wäre für die zukünftigen Modelle ein erster Anhaltspunkt. Die Steigerung der Epochen von 25 auf 50 für das Modell ohne Vorkenntnisse, erreichte bereits eine Verbesserung der Wortfehlerrate. Der Trainingszeitraum umfasste für 50 Epochen, auf einer Grafikkarte und einer Batch-Size von 32, um die zehn Tage.

Das vorher erwähnte Paper trainiert die Modelle auf 512 GPUs mit einer Epochenanzahl von 400 und für weitere Feinabstimmungen zusätzlich 100 Epochen mit einer Batch-Size von 64 [19, Kap. 6.2]. Das Resultat der globalen Batch-Size entspricht etwa  $512 \times 64 = 32.000$ , im Vergleich zu der Batch-Size innerhalb der Arbeit von  $1 \times 32 = 32$ .

Eine weitere Möglichkeit ist das Implementieren eines kleineren Jasper Modells und diesen mit denselben Datensätzen zu trainieren. Interessant wäre hier der Vergleich, ob das Modell durch einen kleineren oder größeren Trainingsdatensatz besser trainiert werden kann.

Tabelle 2 aus [19] hat jeweils das Jasper5x3 Modell und Jasper10x4 Modell mit 50 Epochen getestet. Für beide Modelle wurde die Batch-Norm und die Aktivierungsfunktion ReLu verwendet. Die Wortfehlerrate bei dem kleineren Modell für den Dev-Clean Datensatz war bei 8.82% und für das größere Modell bei 6.15%.

# Literaturverzeichnis

- [1] Jinyu Li, Li Deng, Reinhold Haeb-Umbach und Yifan Gong: *Robust Automatic Speech Recognition: A Bridge to Practical Applications*, 1. Auflage, Elsevier (Stand: Oktober 2015). <https://www.elsevier.com/books/robust-automatic-speech-recognition/li/978-0-12-802398-3>. [21.02.2021].
- [2] Alfons Gottwald: *Automatische Spracherkennung: Methoden der Klassifikation und Merkmalsextraktion*, 2. Auflage, Oldenbourg Wissenschaftsverlag (Stand: Oktober 1994). <https://www.degruyter.com/document/doi/10.1515/9783486785586/html>. [21.02.2021].
- [3] Beat Pfister und Tobias Kaufmann: *Sprachverarbeitung: Grundlagen und Methoden der Sprachsynthese und Spracherkennung*, 2. Auflage, Springer Vieweg (Stand: 2017). <https://www.springer.com/de/book/9783662528372>. [21.02.2021]
- [4] Andreas Stolcke Jasha Droppo: *Comparing Human and Machine Errors in Conversational Speech Transcription*, (Stand: August 2017). <https://www.microsoft.com/en-us/research/wp-content/uploads/2017/06/paper-revised2.pdf>. [21.02.2021]
- [5] Morris Gevirtz: *WORD ERROR RATE: What is Word Error Rate (WER)?*, (Stand: Dezember 2018). <https://deepgram.com/blog/what-is-word-error-rate/>. [21.02.2021]
- [6] Stefan Luber und Nico Litzel: *Definition: Was ist ein Neuronales Netz?*, (Stand: Februar 2018). <https://www.bigdata-insider.de/was-ist-ein-neuronales-netz-a-686185/>. [13.03.2021]
- [7] Jann Raveling: *Was ist ein neuronales Netz?: Begriffe rund um die KI, Maschinelles Lernen und neuronale Netze erklärt*, (Stand: April 2020). <https://www.wfb-bremen.de/de/page/stories/digitalisierung-industrie40/was-ist-ein-neuronales-netz>. [13.03.2021]

- [8] Stefan Luber und Nico Litzel: *Definition: Was ist Deep Learning?*, (Stand: April 2017). <https://www.bigdata-insider.de/was-ist-deep-learning-a-603129/>. [13.03.2021]
- [9] Michael Nielsen: *Neural Networks and Deep Learning*, (Stand: Dezember 2018). <https://static.latexstudio.net/article/2018/0912/neuralnetworksanddeeplearning.pdf>. [16.03.2021]
- [10] Andrew Ng: *What data scientists should know about deep learning*, (Stand: November 2015). <https://www.slideshare.net/ExtractConf>. [16.03.2021]
- [11] Dong Wang, Xiaodong Wang, und Shaohe Lv: *An Overview of End-to-End Automatic Speech Recognition*, (Stand: Dezember 2015). <https://www.mdpi.com/2073-8994/11/8/1018>. [16.03.2021]
- [12] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Erich Elsen, Jesse Engel, Linxi Fan, Christopher Fougner, Tony Han, Awni Hannun, Billy Jun, Patrick LeGresley, Libby Lin, Sharan Narang, Andrew Ng, Sherjil Ozair, Ryan Prenger, Jonathan Raiman, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Yi Wang, Zhiqian Wang, Chong Wang, Bo Xiao, Dani Yogatama, Jun Zhan und Zhenyao Zhu: *Deep Speech 2: End-to-End Speech Recognition in English and Mandarin*, (Stand: Dezember 2015). <https://arxiv.org/abs/1512.02595>. [16.03.2021]
- [13] Jocelyn Huang, Yang Zhang, Boris Ginsburg und Poonam Chitale: *Develop Smaller Speech Recognition Models with NVIDIA's NeMo Framework*, (Stand: Dezember 2019). <https://developer.nvidia.com/blog/develop-smaller-speech-recognition-models-with-nvidias-nemo-framework/>. [22.03.2021]
- [14] Raghav Mani, Kris Kersten, Sirisha Rella und Jocelyn Huang: *Speeding Up Development of Speech and Language Models with NVIDIA NeMo*, (Stand: Oktober 2020). <https://developer.nvidia.com/blog/announcing-nemo-fast-development-of-speech-and-language-models/>. [22.03.2021]
- [15] NVIDIA Developer: *Develop Smaller Speech Recognition Models with NVIDIA's NeMo Framework*, (Stand: Dezember 2019). <https://developer.nvidia.com/conversational-ai>. [22.03.2021]
- [16] Ossama Abdel-Hamid, Abdel-rahman Mohamed, Hui Jiang, Li Deng, Gerald Penn und Dong Yu: *Convolutional Neural Networks for Speech Recogniti-*

- on, IEEE, (Stand: Oktober 2014). <https://ieeexplore.ieee.org/document/6857341>. [25.03.2021]
- [17] Michele Valenti, Stefano Squartini, Aleksandr Diment, Giambattista Parascandolo und Tuomas Virtanen: *A Convolutional Neural Network Approach for Acoustic Scene Classification*, IEEE, (Stand: Juli 2017). <https://ieeexplore.ieee.org/abstract/document/7966035>. [25.03.2021]
- [18] Swapna K E: *Convolutional Neural Network / Deep Learning*, (Stand: August 2020). <https://developersbreach.com/tag/batch-normalization/>. [25.03.2021]
- [19] Jason Li, Vitaly Lavrukhin, Boris Ginsburg, Ryan Leary, Oleksii Kuchaiev, Jonathan M. Cohen, Huyen Nguyen und Ravi Teja Gadde: *Jasper: An End-to-End Convolutional Neural Acoustic Model*, (Stand: August 2019). <https://arxiv.org/abs/1904.03288>. [29.03.2021]
- [20] Sajjad Abdoli, Patrick Cardinal und Alessandro Lameiras Koerich: *End-to-End Environmental Sound Classification using a 1D Convolutional Neural Network*, (Stand: April 2019). [https://www.researchgate.net/publication/333914934\\_End-to-End\\_Environmental\\_Sound\\_Classification\\_using\\_a\\_1D\\_Convolutional\\_Neural\\_Network](https://www.researchgate.net/publication/333914934_End-to-End_Environmental_Sound_Classification_using_a_1D_Convolutional_Neural_Network). [29.03.2021]
- [21] Sergey Ioffe und Christian Szegedy: *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, (Stand: März 2015). <https://arxiv.org/abs/1502.03167>. [01.04.2021]
- [22] Abien Fred Agarap: *Deep Learning using Rectified Linear Units (ReLU)*, (Stand: Februar 2019). <https://arxiv.org/abs/1803.08375>. [01.04.2021]
- [23] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever und Ruslan Salakhutdinov: *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*, (Stand: Juni 2014). <https://jmlr.org/papers/v15/srivastava14a.html>. [01.04.2021]
- [24] Gao Huang, Zhuang Liu, Laurens van der Maaten und Kilian Q. Weinberger: *Densely Connected Convolutional Networks*, (Stand: Januar 2018). <https://arxiv.org/abs/1608.06993>. [01.04.2021]
- [25] Zhao Zhang, Zemin Tang, Yang Wang, Zheng Zhang, Choujun Zhan, Zhengjun Zha und Meng Wang: *Dense Residual Network: Enhancing Global Dense Feature*

- Flow for Character Recognition*, (Stand: Februar 2021). <https://arxiv.org/abs/2001.09021>. [01.04.2021]
- [26] NVIDIA: *Jasper For PyTorch*, (Stand: February 2021). <https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/SpeechRecognition/Jasper>. [01.04.2021]
- [27] NVIDIA: *Multidataset-Jasper10x5DR*, (Stand: September 2020). [https://ngc.nvidia.com/catalog/models/nvidia:multidataset\\_jasper10x5dr](https://ngc.nvidia.com/catalog/models/nvidia:multidataset_jasper10x5dr). [01.04.2021]
- [28] Alex Graves, Santiago Fernández, Faustino Gomez und Jürgen Schmidhuber: *Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks*, (Stand: Januar 2006). [https://www.researchgate.net/publication/221346365\\_Connectionist\\_temporal\\_classification\\_Labelling\\_unsegmented\\_sequence\\_data\\_with\\_recurrent\\_neural\\_networks](https://www.researchgate.net/publication/221346365_Connectionist_temporal_classification_Labelling_unsegmented_sequence_data_with_recurrent_neural_networks). [02.04.2021]
- [29] Awni Hannun: *Sequence Modeling With CTC*, (Stand: November 2017). <https://distill.pub/2017/ctc/>. [02.04.2021]
- [30] NVIDIA: *NVIDIA NeMo Documentation*, (Stand: September 2019). <https://docs.nvidia.com/deeplearning/nemo/user-guide/docs/en/stable/index.html>. [08.04.2021]
- [31] Eli Stevens, Luca Antiga und Thomas Viehmann: *Deep Learning with PyTorch*, Manning Publications, (Stand: Juli 2020). <https://www.manning.com/books/deep-learning-with-pytorch>. [08.04.2021]
- [32] Falcon, WA and .al: *PyTorch Lightning*, (Stand: Mai 2019). <https://github.com/PyTorchLightning/pytorch-lightning>. [08.04.2021]
- [33] Falcon, WA and .al: *PyTorch Lightning Docs: LightningModule*, (Stand: Mai 2019). [https://pytorch-lightning.readthedocs.io/en/stable/common/lightning\\_module.html](https://pytorch-lightning.readthedocs.io/en/stable/common/lightning_module.html). [08.04.2021]
- [34] Falcon, WA and .al: *PyTorch Lightning Docs: Trainer*, (Stand: Mai 2019). <https://pytorch-lightning.readthedocs.io/en/stable/common/trainer.html>. [08.04.2021]
- [35] Nvidia: *NVIDIA NeMo Documentation: Datasets*, (Stand: September 2019). <https://docs.nvidia.com/deeplearning/nemo/user-guide/docs/en/main/asr/datasets.html#preparing-custom-asr-data>. [20.04.2021]

- [36] Nvidia: *Recipe For CommonVoice*, (Stand: April 2020). <https://github.com/NVIDIA/NeMo/issues/568>. [22.04.2021]
- [37] Nvidia: *Nemo Tutorial*, (Stand: März 2021). [https://github.com/NVIDIA/NeMo/blob/r1.0.0rc1/tutorials/asr/01\\_ASR\\_with\\_NeMo.ipynb](https://github.com/NVIDIA/NeMo/blob/r1.0.0rc1/tutorials/asr/01_ASR_with_NeMo.ipynb). [22.04.2021]
- [38] Julius Kunze, Louis Kirsch, Ilia Kurenkov, Andreas Krug, Jens Johannismeier und Sebastian Stober: *Transfer Learning for Speech Recognition on a Budget*, (Stand: Juni 2017). <https://arxiv.org/abs/1706.00290>. [22.04.2021]
- [39] Jocelyn Huang, Oleksii Kuchaiev, Patrick O'Neill, Vitaly Lavrukhin, Jason Li, Adriana Flores, Georg Kucsco und Boris Ginsburg: *Cross-Language Transfer Learning, Continuous Learning, and Domain Adaptation for End-to-End Automatic Speech Recognition*, (Stand: Mai 2020). <https://arxiv.org/abs/2005.04290>. [22.04.2021]
- [40] NGC NVIDIA: *STT En Jasper10x5dr Model*, (Stand: März 2021). [https://ngc.nvidia.com/catalog/models/nvidia:nemo:stt\\_en\\_jasper10x5dr](https://ngc.nvidia.com/catalog/models/nvidia:nemo:stt_en_jasper10x5dr). [22.04.2021]
- [41] Marc Claesen und Bart De Moor: *Hyperparameter Search in Machine Learning*, (Stand: April 2015). <https://arxiv.org/abs/1502.02127>. [24.04.2021]