Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

InIT Institut für angewandte
Informationstechnologie

Bachelorarbeit
FS16 Studiengang Informatik

Automatische Stimmerkennung mit Deep Learning (Neuronalen Netzen)

Autoren	Yanick Lukic Carlo Vogt
Hauptbetreuung	Oliver Dürr Thilo Stadelmann

Datum	9. Juni 2016

Zusammenfassung

Deep Learning, besonders in Form von Convolutional Neural Networks (CNNs), hat in den letzten Jahren zu wesentlichen Verbesserungen in der Bildanalyse und verwandten Bereichen geführt. Dieser Fortschritt wird der Verschiebung weg von handgefertigten Features und individuellen Systemen hin zum automatisierten Lernen von Features aus nahezu unverarbeiteten Daten zugeschrieben. Für Sprecherclustering ist es jedoch noch immer üblich, handgefertigte Verarbeitungsketten wie MFCC Features und GMM-basierte Modelle einzusetzen.

In dieser Arbeit verwenden wir einfache Spektrogramme als Input für zwei unterschiedliche CNN Ansätze und evaluieren das optimale Design dieser Netzwerke für Sprecherclustering. Für den ersten Ansatz untersuchen wir die Frage, wie ein Netzwerk, das auf Sprecheridentifikation trainiert wurde, für Sprecherclustering eingesetzt werden kann. Der zweite Ansatz ist spezifisch für das Clustering von mehrdimensionalen Daten. Hierzu werden paarweise Constraints als Labels verwendet. Dazu verwenden wir eine mit der KL-Divergenz zusammenhängende Loss-Funktion, um die Distanz zwischen gleichen Paaren zu minimieren und zwischen ungleichen Paaren zu maximieren. Beide Ansätze verwenden das Konzept von *Speaker Embeddings*: Aktivierungen von Hidden-Layers eines vortrainierten Netzwerks stellen die Featurevektoren für das Clustering.

Wir demonstrieren unsere Ansätze auf dem bekannten TIMIT Datensatz und erreichen eine Misclassification Rate von 0.05 für das Clustern von 40 unbekannten Sprechern. State of the Art Ansätze erreichen bei identischem experimentellen Aufbau eine Misclassification Rate von 0.0625. Somit verbessern wir den State of the Art um 20%.

Abstract

Deep learning, especially in the form of convolutional neural networks (CNNs), has triggered substantial improvements in computer vision and related fields in recent years. This progress is attributed to the shift from designing features and subsequent individual sub-systems towards learning features and recognition systems end to end from nearly unprocessed data. For speaker clustering, however, it is still common to use handcrafted processing chains such as MFCC features and GMM-based models.

In this thesis, we use simple spectrograms as input to two different CNN architectures and study the optimal design of those networks for speaker clustering. For the first architecture we elaborate on the question how to transfer a network, trained for speaker identification, to speaker clustering. The second architecture directly addresses the problem of clustering high-dimensional data by using pairwise constraints provided as labels. In order to do this, we use a loss function associated with KL divergence to minimize the distance of similar pairs and maximize the distance between dissimilar pairs. Both approaches make use of *speaker embeddings*, by using the activations of hidden layers of a pre-trained network as feature vectors for clustering.

We evaluate our approaches on the well-known TIMIT dataset and achieve a misclassification rate of 0.05 for clustering 40 unknown speakers. State of the art approaches achieve misclassification rates of 0.0625 with the same experimental setup. We therefore exceed the state of the art by 20%.

Vorwort

Vor einem Jahr traten wir an Thilo Stadelmann, bezüglich einer Projektarbeit, heran. Als er uns voller Begeisterung von der Idee, Sprechererkennung mit neuronalen Netzwerke zu betreiben, erzählte, wurden wir sogleich neugierig. Obwohl wir noch nicht genau wussten, was wir mit all diesen Begriffen und Ideen anfangen sollten, war unser Interesse geweckt. Nach intensiver Einarbeitung und ersten Resultaten nahm die Motivation und die Begeisterung immer mehr zu. Auf erste grössere Fortschritte folgten auch diverse Rückschläge, die unsere Arbeit immer wieder um Tage oder auch Wochen zurückwarfen. Kleine anfängliche Unsicherheiten unsererseits hatten zum Teil grosse Auswirkungen auf die Resultate und unsere Erkenntnisse. Thilo Stadelmann und Oliver Dürr, unsere Betreuer, waren zum Glück immer geduldiger mit uns, als wir mit uns selbst. Schlussendlich können wir auf eine gelungene Arbeit zurückblicken, die uns weitergebracht hat und auf die wir sehr stolz sind. An einer Fachhochschule ein wissenschaftliches Forschungsthema zu bearbeiten ist nicht immer möglich und wir sind sehr dankbar diese Möglichkeit bekommen zu haben.

Die Bachelorarbeit zu zweit zu bearbeiten hat uns als Team und individuell weitergebracht. Diverse Meinungsverschiedenheiten konnten wir stets positiv nutzen, Rückschläge als Team wegstecken und unsere Ideen weiter vorantreiben. Was uns sicher auch in Zukunft bei neuen Herausforderungen helfen wird. Wir werden sicher immer mit Freude und Stolz auf unser letztes Bachelorstudiumsjahr zurückblicken.

Wir möchten an dieser Stelle unseren Betreuern Thilo Stadelmann und Oliver Dürr für ihre Unterstützung und Geduld danken. Wir stiessen mit unseren Lösungsansätzen immer auf offene Ohren und hatten wir einmal keine, konnten wir auf diverse Inputs und Anregungen zählen. Den unkomplizierten und freundschaftlichen Umgang haben wir sehr geschätzt, vielen Dank dafür.

Yanick Lukic & Carlo Vogt

06. Juni 2016, Winterthur

Erklärung betreffend das selbständige Verfassen einer Bachelorarbeit an der School of Engineering

Mit der Abgabe dieser Bachelorarbeit versichert der/die Studierende, dass er/sie die Arbeit selbständig und ohne fremde Hilfe verfasst hat. (Bei Gruppenarbeiten gelten die Leistungen der übrigen Gruppenmitglieder nicht als fremde Hilfe.)

Der/die unterzeichnende Studierende erklärt, dass alle zitierten Quellen (auch Internetseiten) im Text oder Anhang korrekt nachgewiesen sind, d.h. dass die Bachelorarbeit keine Plagiate enthält, also keine Teile, die teilweise oder vollständig aus einem fremden Text oder einer fremden Arbeit unter Vorgabe der eigenen Urheberschaft bzw. ohne Quellenangabe übernommen worden sind.

Bei Verfehlungen aller Art treten die Paragraphen 39 und 40 (Unredlichkeit und Verfahren bei Unredlichkeit) der ZHAW Prüfungsordnung sowie die Bestimmungen der Disziplinarmassnahmen der Hochschulordnung in Kraft.

Ort, Datum:

……

Unterschriften:

……...

……

……

Das Original dieses Formulars ist bei der ZHAW-Version aller abgegebenen Bachelorarbeiten zu Beginn der Dokumentation nach dem Abstract bzw. dem Management Summary mit Original-Unterschriften und -Datum (keine Kopie) einzufügen.

Inhaltsverzeichnis

1. Einleitung **12**
- 1.1. Motivation . . . 12
- 1.2. Ausgangslage . . . 12
- 1.3. Zielsetzung . . . 13
- 1.4. Aufbau . . . 13

2. Grundlagen **14**
- 2.1. Maschinelles Lernen . . . 14
  - 2.1.1. Supervised Learning . . . 14
  - 2.1.2. Unsupervised Learning . . . 14
  - 2.1.3. Reinforcement Learning . . . 15
- 2.2. Neuronale Netzwerke . . . 15
  - 2.2.1. Perzeptron . . . 16
  - 2.2.2. Sigmoid Neuronen . . . 16
  - 2.2.3. Layer . . . 17
  - 2.2.4. Aufbau von neuronalen Netzwerken . . . 18
  - 2.2.5. Loss-Funktion (cost function oder objective function) . . . 19
- 2.3. Optimierungsmethoden . . . 19
  - 2.3.1. Backpropagation . . . 19
  - 2.3.2. Epochen . . . 20
  - 2.3.3. Softmax Normalisierung . . . 20
  - 2.3.4. Overfitting . . . 20
- 2.4. Optimierungsalgorithmen für das Gradientenverfahren . . . 20
  - 2.4.1. Momentum . . . 20
  - 2.4.2. Nesterov-Momentum . . . 21
  - 2.4.3. Adam . . . 21
  - 2.4.4. Adadelta . . . 22
- 2.5. Convolutional Neural Networks . . . 22
  - 2.5.1. Convolution . . . 22
  - 2.5.2. Aufbau CNN . . . 23
- 2.6. Recurrent Neural Networks . . . 24
- 2.7. Clusteringansätze . . . 25
  - 2.7.1. Siamesische Architektur . . . 25
  - 2.7.2. Paarweise KL-Divergenz . . . 25
  - 2.7.3. Sprecher-Embeddings . . . 26
- 2.8. Spektrogramm . . . 26

3. Vorgehen und Methoden **27**
- 3.1. Auswahl der Ansätze . . . 27
  - 3.1.1. Clustering mittels Identifikationsnetzwerk . . . 27
  - 3.1.2. Clustering Netzwerk mit paarweiser KL-Divergenz als Loss-Funktion . . . 28
- 3.2. Konzept . . . 28
  - 3.2.1. Grundarchitektur: Identifikationsnetzwerk . . . 28
  - 3.2.2. Grundarchitektur: KL-Divergenz Netzwerk . . . 29
  - 3.2.3. Form der Eingabedaten . . . 29
- 3.3. Datensätze . . . 30
  - 3.3.1. TIMIT Datensatz . . . 30
  - 3.3.2. MNIST Datensatz . . . 30

3.4. Experimenteller Aufbau . . . 31
3.5. Messgrössen . . . 31
3.5.1. Accuracy . . . 31
3.5.2. Misclassification Rate (MR) . . . 31
3.6. Verwendete Bibliotheken . . . 32
3.7. Hardware . . . 32

4. Implementation **33**
4.1. Allgemein . . . 33
4.1.1. Datenaufbereitung . . . 33
4.1.2. Training . . . 34
4.2. Paarweise KL-Divergenz . . . 35
4.2.1. Initialisierung des Netzwerks . . . 35
4.2.2. Berechnung der Loss-Funktion . . . 35
4.2.3. Testen der Accuracy . . . 37

5. Resultate Identifikationsnetz **39**
5.1. Architektur . . . 39
5.2. Experiment 1: Evaluation der Filter . . . 39
5.3. Experiment 2: Speaker Embedding . . . 41
5.4. Fazit . . . 43

6. Resultate KL-Divergenz **44**
6.1. Architektur . . . 44
6.2. Experiment 3: Clustering von MNIST . . . 44
6.3. Experiment 4: Clustering von Spektrogrammen . . . 45
6.4. Experiment 5: Statische Convolution . . . 47
6.5. Experiment 6: Mehr gleiche Sprecher . . . 48
6.6. Experiment 7: Optimierungsverfahren . . . 49
6.7. Experiment 8: Training mit 590 Sprechern . . . 52
6.8. Fazit . . . 53

7. Schlussfolgerungen **55**
7.1. Fazit . . . 55
7.2. Ausblick . . . 56
7.2.1. Clustering mit Deep Learning Verfahren . . . 56
7.2.2. Netzwerk mit paarweiser KL-Divergenz . . . 56

8. Verzeichnisse **58**
Literaturverzeichnis . . . 58
Abbildungsverzeichnis . . . 62
Tabellenverzeichnis . . . 63
Listingverzeichnis . . . 64

A. Anhang **I**
A.1. Projektmanagement . . . I
A.1.1. Offizielle Aufgabenstellung . . . I
A.2. Weiteres . . . II
A.2.1. Beschreibung der elektronischen Daten . . . II
A.2.2. Sprecherlisten . . . II

1. Einleitung

Sprechererkennung (speaker recognition) ist die Bezeichnung für das Erkennen von *wer* hat gesprochen, und nicht wie Spracherkennung (speech recognition) das Erkennen von *was* gesprochen wird. Sprechererkennung versucht ein Modell eines Sprechers zu erstellen, das möglichst gut von einem Computer verarbeitet werden kann.
Die Sprechererkennung wird in die drei Unteraufgaben Sprecherverifizierung, Sprecheridentifizierung und Sprecherclustering aufgeteilt. Die Aufgabenstellung der Sprecherverifizierung ist das Bestätigen ob ein Sample eines Sprechers zu einer gegebenen Vorlage passt. Das bedeutet zwischen Sample und Muster besteht eine $1{:}1$ Beziehung. Sprecheridentifizierung hat beliebig viele Vorlagen, ein Sample wird mit allen Vorlagen verglichen und einer zugewiesen ($1{:}N$ Beziehung). Bei dem Task Sprecherclustering ist nicht bekannt wie viele Sprecher und welche Sprecher vorkommen können. In diesem, am schwierigsten zu lösenden Fall, besteht eine $N{:}N$ Beziehung zwischen Samples und Muster.

1.1. Motivation

Sprechererkennung wird in vielen Bereichen eingesetzt. Beispiele sind biometrische Zutrittskontrollen oder automatisierte Protokollführung. Zusätzlich gibt es diverse weitere Anwendungsbeispiele. Computer sind jedoch noch bei weitem nicht so genau in den Tasks des Sprechererkennens und Sprecherclusterings wie das menschliche Gehör. Mit Convolutional Neural Networks wurden gute Ergebnisse in Bildverarbeitungstasks erzielt. Dies könnte ebenfalls für Audioanalysen angewendet werden. Durch das Verwenden von lernfähiger Convolution könnten Maschinen den Rückstand zum Menschen verringern. Mit besseren Resultaten könnte in der technischen Umsetzung von Sprechererkennung und -clustering vermehrt auf maschinelle Lernverfahren, insbesondere neuronale Netzwerke, gesetzt werden.

1.2. Ausgangslage

Die vorliegende Bachelorarbeit wurde unter Betreuung von Thilo Stadelman und Oliver Dürr am Institut für angewandte Informationstechnologie (InIT) an der ZHAW School of Engineering erarbeitet. Es ist eine Weiterführung unserer Projektarbeit Sprechererkennung mit Convolutional Neural Networks [1].

Stand der Technik

Traditionelle Verfahren im Bereich der Sprecheridentifikation verwenden Gaussian Mixture Models (GMMs) mit Mel Frequency Cepstrum Coefficient (MFCCs) als Featurevektoren [2]. Kürzlich wurde dieser Ansatz um den Einsatz von Joint Factor Analysis [3] und Intermediate Vectors (i-vectors) [4] erweitert. Damit können kompakte Sprecherrepräsentationen fixer Länge für gegebene Audiodaten von Sprechern erstellt werden. Obwohl dieses Verfahren der State of the Art ist, hat es einige Schwachstellen. Der Einsatz von MFCCs als Featurevektoren ignoriert die meisten sprachspezifischen Charakteristiken des Sprachsignals. Insbesondere werden keine Merkmale gesucht, die das Unterscheiden zwischen Sprechern ermöglichen würde.

Sprecherclustering verwendet meist dieselben Ansätze wie die Sprecheridentifikation [5]. Neuere Ansätze basieren darauf weitere Informationen hinzuzuziehen. Beispiele sind der zusätzliche Einsatz von mehreren Mikrofonen die einzelnen Sprechern zugeordnet werden. So können Probleme, wie überlappende Sprecher, gelöst werden [6]. Ein anderer Ansatz ist das verwenden von Videomaterial zum eigentlichen

Audiostream, um mittels Bildanalyse wie zum Beispiel Gesichtserkennung weitere Informationen bezüglich dem aktuellen Sprecher zu erhalten [7].
Diese Ansätze erreichen in diesen spezifischen Aufgaben sehr gute Ergebnisse. Die eigentliche Stimmerkennung wird jedoch nicht verbessert.

1.3. Zielsetzung

Ziel dieser Arbeit ist es mittels Convolutional Neural Networks ein Identifikationsnetzwerk für Sprechererkennung zu implementieren und unseren Ansatz aus der Projektarbeit [1] weiterzuentwickeln. Zusätzlich soll untersucht werden wie sich Identifikationsnetzwerke für Sprecherclustering eignen. In einem zweiten Teil soll ein Netzwerk trainiert werden, das das Sprecherclustering selbständig lernen kann. Diese Ansätze sollen experimentell mit den State of the Art Ansätzen verglichen werden. Die Arbeit wird mit den Softwarebibliotheken `Theano`, `Lasagne` und `nolearn.lasagne` umgesetzt. Im Anhang unter dem Punkt A.1.1 ist die offizielle Aufgabenstellung der Arbeit angefügt.

1.4. Aufbau

Im Kapitel 2 werden die Grundlagen für das Verständnis der Arbeit vermittelt. Es wird die Theorie von neuronalen Netzwerken beschrieben. Dabei werden Grundkenntnisse in der Informatik und Mathematik vorausgesetzt. In Kapitel 3 wird beschrieben, welche Ansätze gewählt wurden und nach welchem Konzept die Architekturen aufgebaut sind. Für das Implementationskapitel 4 wurde eine Auswahl von speziellen Komponenten unserer Arbeit getroffen. Der im Rahmen dieser Arbeit programmierte Quellcode ist auf der beigelegten CD enthalten. Die Resultate sind in die Kapitel 5 und 6 aufgeteilt. In Kapitel 5 Resultate Identifikationsnetz werden die Experimente beschrieben, die im Ausblick der Arbeit [1] aufgelistet sind. In Kapitel 6 Resultate KL-Divergenz sind die Resultate unseres zweiten Ansatzes beschrieben. Ein zusammenfassendes Fazit und mögliche zukünftige Schritte und Möglichkeiten werden im Kapitel 7 aufgelistet und erläutert.

2. Grundlagen

In diesem Kapitel wird ein Einstieg in neuronale Netzwerke und Methoden gegeben, die in dieser Arbeit verwendet wurden. Die wichtigsten Inhalte, die benötigt werden um die Arbeit zu verstehen, werden im folgenden Kapitel vermittelt.

2.1. Maschinelles Lernen

Der Grundgedanke von maschinellem Lernen ist ein System zu entwickeln, das sich selbst programmieren kann. Es ist der wissenschaftliche Bereich mit dem Ziel autonom lernende Computersysteme zu bauen. Das System soll selbstständig von Daten lernen und das gelernte Muster auf neuen Daten anwenden können. Maschinelles Lernen basiert auf der Kombination von Informatik und Statistik [8].
Im Folgenden werden die drei Teilgebiete des maschinellen Lernens kurz erklärt.

2.1.1. Supervised Learning

Für Supervised Learning wird dem Netzwerk gesagt welcher Output für einen gegebenen Input erwartet wird. Anhand des Erwartungswerts und des errechneten Werts des Netzwerks kann eine Fehlerberechnung durchgeführt werden. Ziel ist es diesen Fehler zu minimieren. Der Nachteil dieser Lernmethode ist, dass alle Trainingsdaten bereits einem Erwartungswert zugewiesen sein müssen [9]. In dieser Arbeit wird ausschliesslich mit Supervised Learning gearbeitet.

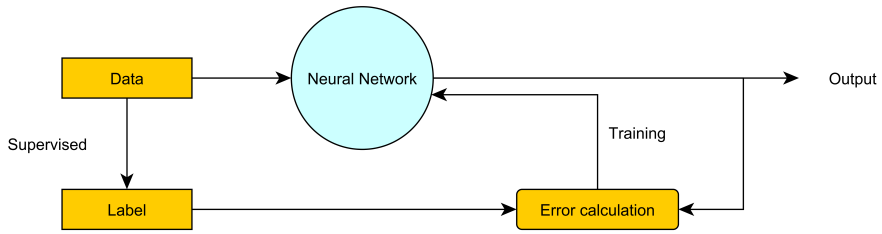

Abbildung 2.1.: Supervised Learning.

2.1.2. Unsupervised Learning

Der Unterschied zu Supervised Learning ist, dass kein Erwartungswert vorgegeben ist. Die Fehlerberechnung muss anhand einer Regel erstellt werden. Diese Regel kann ebenfalls maschinelles Lernen beinhalten und seine Parameter anpassen [9].

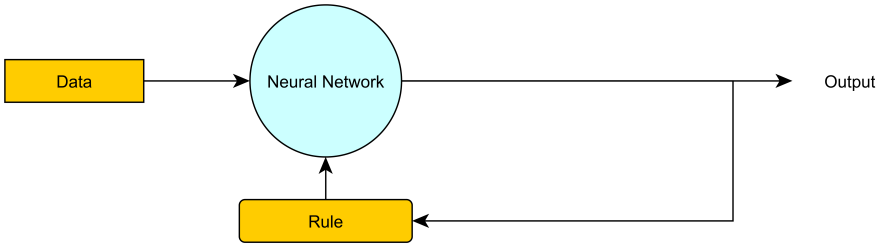

Abbildung 2.2.: Unsupervised Learning.

2.1.3. Reinforcement Learning

Bei Reinforcement Learning wird gelernt wie etwas zu lösen ist. Dem Netzwerk wird nicht ein optimaler Output in die Fehlerberechnung eingegeben, es wird lediglich gesagt ob der Output korrekt oder inkorrekt ist. Wie dieser Output auszusehen hat wird nicht definiert. Reinforcement Learning funktioniert nach dem "Trial und Error"Prinzip[10].

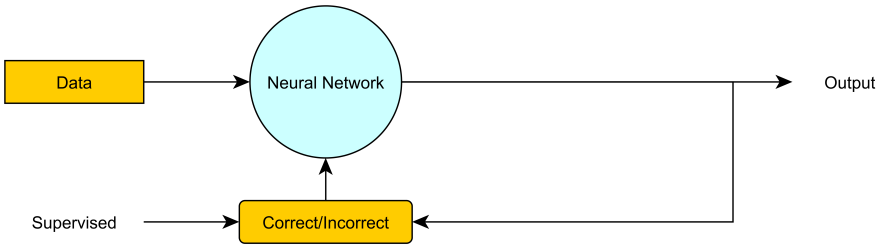

Abbildung 2.3.: Reinforcement Learning.

2.2. Neuronale Netzwerke

Die typischen Netzwerke sind zurzeit Feedforward neuronale Netzwerke. Der Name *Feedforward* beschreibt, dass diesen Netzwerken eine Funktion ohne Feedbackmöglichkeit zugrunde liegt. Der Input x ergibt bei gleichbleibender Funktion immer denselben Output. Sie werden meist für Klassifizierungsaufgaben eingesetzt. Dabei wird eine Approximationsfunktion f^* gesucht, die dem Input x eine Klasse y zuordnet[11].

$$y = f^*(x, \theta) \tag{2.1}$$

Die Parameter θ werden trainiert und so angepasst um eine möglichst gute Approximation der Zielfunktion zu erreichen. Für diese Trainings werden in den meisten Fällen eine grosse Anzahl von Samples benötigt, die für den gewünschten Task die korrekte Klasse y bereits zugewiesen haben. Dies ist eine Schwäche von neuronalen Netzwerken. Die Beschaffung von geeigneten Samples gestaltet sich oft als schwierig und zeitaufwendig.

2.2.1. Perzeptron

Ein neuronales Netzwerk ist eine Verknüpfung von mehreren Neuronen. Um die Funktionsweise eines Neurons zu erklären, beginnen wir erst mit der eines Perzeptrons. Ein Perzeptron ist ein einfacher Typ eines Neurons, es nimmt $1...N$ Inputs an und gibt eine binäre Ausgabe zurück.

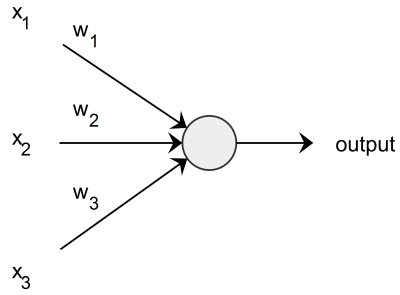

Abbildung 2.4.: Perzeptron mit drei Inputs.

Jedem Input x_i wird eine Gewichtung w_i zugewiesen. Der Input wird mit den Gewichtungen multipliziert und aufsummiert. Für die binäre Output-Entscheidung wird ein Schwellwert (Threshold) verwendet.

$$output = \sum_{n=0}^{N} x_n * w_n \tag{2.2}$$

Wir ein Bias b zur Gewichtung des Perzeptrons einbezogen, wird dieser zu den aufsummierten Produkten, von Inputs und Gewichten, addiert.

$$output = [\sum_{n=0}^{N} x_n * w_n] + b \tag{2.3}$$

Ein Bias in einem Perzeptron ist im Normalfall definiert als $b = -treshold$. So kann direkt auf negative oder positive Outputs geprüft werden [12].

2.2.2. Sigmoid Neuronen

Der Schwellwert von Perzeptronen kann bei kleinen Änderungen der Gewichte den Output schnell verändern. Dies kann einen negativen Einfluss auf den Lernprozess haben oder das Lernen komplett verunmöglichen. Aus diesem Grund werden in neuronalen Netzwerken Neuronen mit einer Aktivierungsfunktion eingesetzt. Ein Beispiel hierfür ist die Sigmoid Funktion. Mit dieser löst man das Problem der Perzeptronen. So können kleine Änderungen nicht mehr zu gegensätzlichen Ergebnissen führen. Im Unterschied zu Perzeptronen haben Sigmoid Neuronen keine binäre Entscheidung, sie können jeden Wert zwischen 0 und 1 annehmen. Die Sigmoid Funktion α ist wie folgt definiert:

$$\alpha(z) \equiv \frac{1}{1 + e^{-z}} \tag{2.4}$$

Mit der Anzahl Inputs N und den dazugehörigen Gewichten ergibt sich die folgende Formel. Als z wird die Funktion 2.3 angenommen.

$$output = \frac{1}{1 + e^{-[\sum_{n=0}^{N} x_n * w_n] + b}} \tag{2.5}$$

Bei einem grossen Wert für z approximiert der Output gegen 1, bei einem negativen z gegen 0.

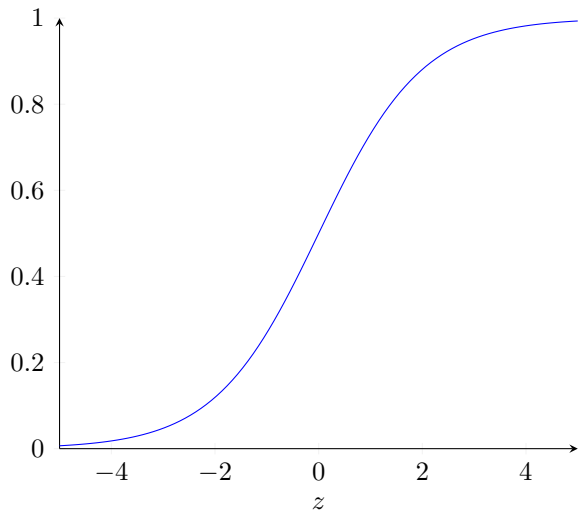

Abbildung 2.5.: Sigmoid Funktion.

Bei einem Neuron mit Aktivierungsfunktion verschiebt der Bias die gesamte Funktion auf der x-Achse. Bei einem neutralen Bias ist der Output-Mittelwert bei 0 auf der x-Achse.
Die Sigmoid Funktion ist die bekannteste Aktivierungsfunktion. In dieser Arbeit werden wir jedoch die Rectifier Funktion [13] verwenden. Der Aktivierungsfunktionsgraph wird in Abbildung 2.6 dargestellt. Mit der Rectifier Funktion wurden bei Audiodaten gute Ergebnisse erzielt [14].

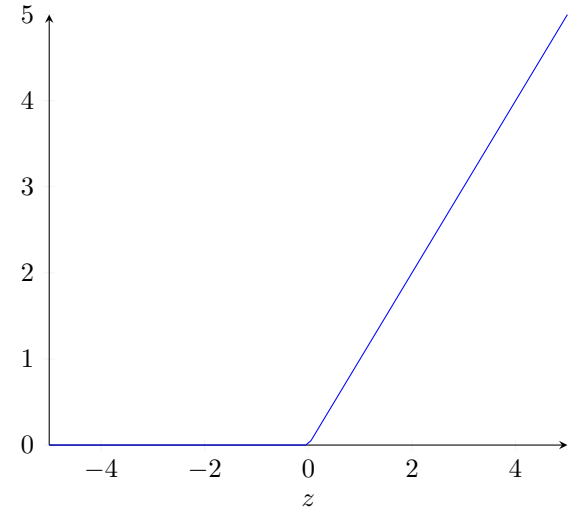

Abbildung 2.6.: Rectified Linear Unit (ReLU) Aktivierungsfunktion [14].

2.2.3. Layer

Als Layer wird eine Gruppe von Neuronen bezeichnet die sich im Netzwerk auf derselben Stufe befinden und untereinander keine Verbindung haben. Jeder Layer stellt eine approximierte Funktion f^* dar [11]. Ein Netzwerk mit drei Layern hat drei unterschiedliche Funktionen.

$$Network(x,\theta) = f^3(f^2(f^1(x,\theta),\theta),\theta) \tag{2.6}$$

Layer werden in Input-Layer, Output-Layer und Hidden- oder Dense-Layer unterschieden. Der Input-Layer beinhaltet die ersten Neuronen, bei denen die Daten eingegeben werden. Der Output-Layer ist

der letzte Layer und der Ort an dem die Resultate abgelesen werden können. Als Hidden Layer werden alle Layer zwischen Input- und Output-Layer bezeichnet.

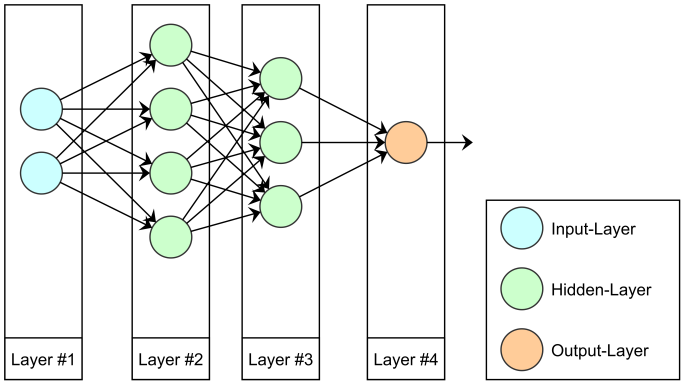

Abbildung 2.7.: Layer in einem Feedforward neuronalen Netzwerk.

2.2.4. Aufbau von neuronalen Netzwerken

Als einfaches Beispiel für ein neuronales Netzwerk kann ein Netzwerk mit zwei Layern betrachtet werden. Dieses Netzwerk bildet ein Input-Muster direkt auf ein Output-Muster ab. Eine Funktion, die mit einem solchen Zwei-Layer-Netzwerk dargestellt werden kann, ist die AND-Funktion [15].

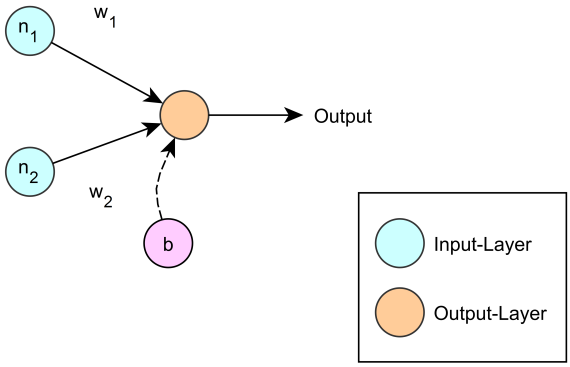

Abbildung 2.8.: Neuronales Netzwerk ohne Hidden-Layer.

Der Output des in Abbildung 2.8 dargestellten Netzwerks wird wie folgt berechnet:

$$output = n_1 * w_1 + n_2 * w_2 + b \tag{2.7}$$

Mit einem solchen minimalen Netzwerk 2.8 können viele Problemstellungen nicht abgebildet werden. Für komplexere Tasks werden zusätzliche Neuronen und Layer benötigt. In Abbildung 2.9 ist ein neuronales Netzwerk mit je einem Input- und Output-Layer sowie zwei Hidden-Layern dargestellt.

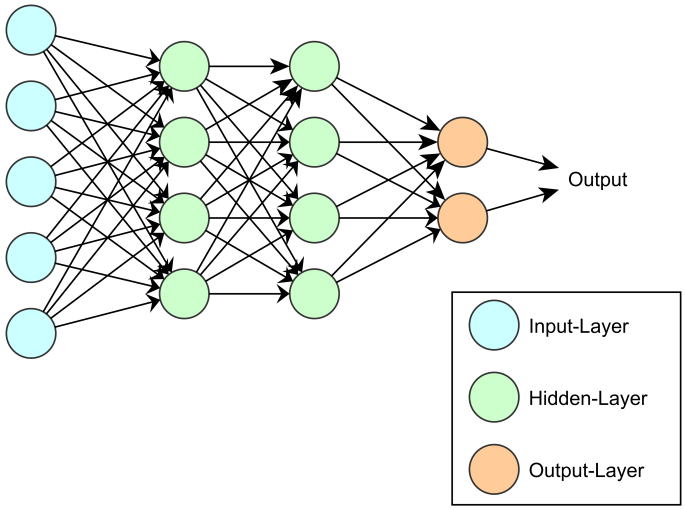

Abbildung 2.9.: Neuronales Netzwerk mit zwei Hidden-Layer.

Von Deep Learning kann gesprochen werden sobald ein Netzwerk Hidden-Layers beinhaltet.

2.2.5. Loss-Funktion (cost function oder objective function)

Für das berechnen des Fehlers, den ein neuronales Netzwerk während des Trainings macht, wird eine Loss-Funktion verwendet. Dabei wird Bezug auf die Netzwerkparameter (Bias und Gewichte) θ genommen, die in der aktuellen Trainingseinheit verwendet werden. Ein oft verwendetes Beispiel ist die quadratische Kostenfunktion. Diese ist im Folgenden dargestellt.

$$C(x,\theta) \equiv \frac{1}{2N} \sum_{i=0}^{N} |f(x_i,\theta) - y_i|^2. \tag{2.8}$$

Es werden die quadrierten Distanzen zwischen den Netzwerk-Outputwerten $f(x_i,\theta)$ und den Soll-Werten y_i berechnet und aufsummiert. Dabei ist N die Anzahl Samples in x. Das Ziel ist es nun mit Trainings- und Optimierungsmethoden diese Kostenfunktion zu minimieren [12].

2.3. Optimierungsmethoden

2.3.1. Backpropagation

Backpropagation ist eine weitverbreitete Methode um neuronale Netzwerke bei *Supervised Learning* Tasks zu trainieren. Das Ziel dabei ist, die Gewichte des Netzwerks so anzupassen, dass Inputs korrekten Outputs zugeordnet werden. Grob werden dabei folgende Schritte durchgeführt:

- **Forward Pass**: Ein Input wird durch das Netzwerk propagiert und generiert einen Output.
- **Fehler berechnen**: Basierend auf dem Ist- und Soll-Output wird ein Fehler berechnet.
- **Backward Pass**: Der Fehler wird rückwärts durch das Netzwerk propagiert und die Gewichte werden entsprechend ihrem Einfluss auf den Fehler angepasst.

Das Anpassen der Gewichte geschieht über das sogenannte Gradientenverfahren. Dabei handelt es sich um ein numerisches Verfahren um Optimierungsprobleme zu lösen. Von einem angenäherten Startwert schreitet man jeweils in die Richtung des negativen Gradienten, bis man keine Verbesserung mehr erziehlt.

Das Verfahren wird angewendet um die Loss-Funktion, dargestellt durch eine Summe von differenzierbaren Funktionen, zu minimieren. Das wird erreicht indem der Gradient über eine Anzahl Samples berechnet wird und die Parameter dann abhängig von ihren Gewichten angepasst werden, was wiederum den Prozess der Backpropagation abbildet. Wieviele Samples verwendet werden, wird durch das verwendete Unterverfahren des Gradientenverfahrens bestimmt. Diese sind im Folgenden aufgelistet.

- **Batch Gradient Descent**: alle Samples
- **Stochastic Gradient Descent**: 1 Sample
- **Mini-Batch Gradient Descent**: b Samples

2.3.2. Epochen

Nachdem alle Trainingsdaten einmal durch das Netzwerk durchgegeben und zurück propagiert wurden, wird von einer Epoche gesprochen. Epochen werden in Mini-Batches unterteilt. Alle Daten eines Mini-Batches werden durch das Netzwerk gegeben, jedoch wird nur einmal pro Mini-Batch der Fehler ins Netzwerk zurückgerechnet. Mini-Batches ermöglichen es, auch auf Systemen Netzwerke zu trainieren, die über kleinere dynamische Speicher oder weniger Rechenleistung verfügen [12].

2.3.3. Softmax Normalisierung

Mittels Softmax Normalisierung wird ein K-dimensionaler Vektor z in einen Vektor $\sigma(z)$ mit reellen Werten konvertiert. Der Definitionsbereich der Werte entspricht dabei $D_f = [0,1]$ und die Werte eines Vektors summieren auf 1. Somit wird für jede Klasse $1,...,K$ die Wahrscheinlichkeit des Zutreffens für den gegebenen Input dargestellt [9].

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \text{für} j = 1,...,K \tag{2.9}$$

2.3.4. Overfitting

Als Overfitting bezeichnet man, wenn ein statistisches Modell oder ein Machine-Learning Algorithmus nur noch auf seine Trainingsdaten passt. Das hat zur Folge, dass es für keine anderen Daten verwendet werden kann. Overfitting lässt sich daran erkennen, dass der Trainingsfehler ab- und der Validierungsfehler zunimmt. Dieser Effekt lässt sich während des Trainings beobachten. Auslöser kann ein zu komplizierter Ansatz sein oder wenn zu viele Variablen verwendet wurden. Overfitting lässt sich vermeiden, indem das Training vor dem Effekt abgebrochen wird. Das Verwenden von Dropout-Layern ist eine weitere Methode. Diese setzen zufällige Gewichte auf Null und erschweren so, dass sich das Netzwerk zu stark auf die Trainingsdaten anpasst [16].

2.4. Optimierungsalgorithmen für das Gradientenverfahren

2.4.1. Momentum

Ziel des Trainings ist es eine stochastische Funktion $f(\theta)$ zu minimieren. Dazu kann ein einfaches Gradientenverfahren wie folgt verwendet werden:

$$\begin{aligned} g_t &\leftarrow \Delta_{\theta_{t-1}} f(\theta_{t-1}) \\ \theta_t &\leftarrow \theta_{t-1} - \eta * g_t \end{aligned} \tag{2.10}$$

η ist dabei eine definierte Schrittweite (Lernrate). Dieser Ansatz stösst an Grenzen sobald er über mehrere Dimensionen verwendet wird. So kann es schnell passieren, dass dieser in einem lokalen Minimum hängen bleibt. In diesem Fall wird ein Momentumsvektor m benötigt und eine Zerfallskonstante ρ.

$$\begin{aligned} g_t &\leftarrow \Delta_{\theta_{t-1}} f(\theta_{t-1}) \\ m_t &\leftarrow \rho * m_{t-1} + g_t \\ \theta_t &\leftarrow \theta_{t-1} - \eta * g_t \end{aligned} \tag{2.11}$$

So wird erreicht, dass Anteile der vorhergehenden Gradienten hinzugerechnet werden. Zeigt ein Gradient also in die gleiche Richtung wie im vorigen Schritt, werden die Schritte in die Richtung des Minimums erhöht. So sind oft weniger Trainingsiterationen nötig um ein Minimum zu erreichen und zusätzlich kann das Hängenbleiben in lokalen Minimas verhindert werden.

2.4.2. Nesterov-Momentum

Das Nesterov-Momentum ist eine Optimierung des normalen Momentums. Neu werden nicht nur die Parameter der letzten Iteration verwendet, sondern zusätzlich subtrahiert man davon das Produkt des letzten Momentums m_{t-1} mit der Konstante ρ und der Schrittweite η.

$$\begin{aligned} g_t &\leftarrow \Delta_{\theta_{t-1}} f(\theta_{t-1} - \eta\rho m_{t-1}) \\ m_t &\leftarrow \rho * m_{t-1} + g_t \\ \theta_t &\leftarrow \theta_{t-1} - \eta * g_t \end{aligned} \tag{2.12}$$

Dieses Verfahren ergibt in vielen Fällen signifikant bessere Resultate als ein einfaches Momentum [17].

2.4.3. Adam

Adam ist ein Algorithmus für stochastische Funktionen erster Ordnung. Er kombiniert die Methoden AdaGrad [17] und RMSProp [17]. Das Ziel ist es die Vorteile beider Verfahren zu vereinen [18]. Adam gibt eine individuelle Lernrate für unterschiedliche Parameter zurück. Hierfür schätzt er die ersten zwei Momentums eines Gradienten [18]. g_t entspricht dem Gradienten zum Zeitpunk t und g_i^2 einer elementweisen Quadrierung eines Gradienten. Die Parameter β_1 und β_2 sind Hyperparameter, die im Intervall $[0, 1]$ gewählt werden. Die Funktion $E[m]_t$ ist die Annahme für das erste Momentum der Gleichung.

$$E[m]_t = \beta_1 * E[m]_{t-1} + (1 - \beta_1) * g_t \tag{2.13}$$

$E[v]_t$ ist die Berechnung für die zweite Momentumsannahme.

$$E[v]_t = (1 - \beta_2) \sum_{i=1}^{t} \beta_2^{t-i} * g_i^2 \tag{2.14}$$

Für beide Momentumsannahmen wird jeweils eine Korrektur durchgeführt, die wie folgt berechnet wird:

$$\widehat{m}_t = \frac{E[m]_t}{1 - \beta_1^t} \tag{2.15}$$

$$\widehat{v}_t = \frac{E[v]_t}{1 - \beta_2^t} \tag{2.16}$$

Das Update für die Parameter in θ werden mit $\Delta\theta_t$ dargestellt. Dabei ist η eine global definierte Lernrate.

$$\Delta\theta_t = \Delta\theta_{t-1} - \frac{\eta * \widehat{m}_t}{\sqrt{\widehat{v}_t} + \epsilon} \tag{2.17}$$

Für die Konstanten β_1, β_2 und ϵ werden die Werte 0.9, 0.999 respektive 10^{-8} empfohlen [18].

2.4.4. Adadelta

Adadelta ist eine Lernratenmethode für ein Gradientenverfahren [19]. Der Grundgedanke ist, dass der Nenner in der Updatefunktion lediglich auf der Summe von ω quadrierten Gradienten vergangener Iterationen basiert. Durch die Quadrierung ist jeder der aufsummierten Werte positiv, der Nenner wird also beständig grösser. Dies bewirkt, dass die Lernrate über jede Iteration kleiner wird. Dieses Aufsummieren von quadrierten Werten ist jedoch ineffizient und aus diesem Grund wird dieser Wert durch $E[g^2]_t$ zum Zeitpunkt t nur angenähert.

$$E[g^2]_t = \rho E[g^2]_{t-1} + (1-\rho) g_t^2 \tag{2.18}$$

Mit ρ als Zerfallskonstante, analog zur Momentum Methode in Abschnitt 2.4.1.
Die Updatefunktion ist wie folgt definiert, dabei ist ϵ eine Konstante um nummerische Stabilität sicherzustellen [20].

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g_t^2 + \epsilon]}} g_t \tag{2.19}$$

θ sind die zu aktualisierenden Parameter und η eine global definierte Lernrate.

2.5. Convolutional Neural Networks

Dieser gesamte Abschnitt bezieht sich auf das Kapitel 9 aus dem Buch "Deep Learning" [11].
In normalen neuronalen Netzwerken müssen die zu lernenden Features angegeben werden. Convolutional Neural Networks (im Folgenden CNN) ermöglichen das selbständige erlernen von Features für gegebene Inputdaten. Die Daten müssen in einem Gitterraster vorliegen, zum Beispiel in Form eines zweidimensionalen Arrays (z.B. ein Bild).

2.5.1. Convolution

Die mathematische Operation Convolution ist eine spezielle Variante einer linearen Operation auch Faltung genannt. In einem CNN wird diese Operation in mindestens einem Layer verwendet.

$$I = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1i} \\ a_{21} & \ddots & & \\ \vdots & & \ddots & \\ a_{j1} & & & a_{ji} \end{pmatrix}, g = \begin{pmatrix} b_{11} & \cdots & b_{1m} \\ \vdots & \ddots & \\ b_{n1} & & b_{nm} \end{pmatrix}$$

I ist die Input-Matrix und g der sogenannte Convolution-Filter. Die Funktion lautete nun wie folgt:

$$I'(i,j) = (I * g)(i,j) = \sum_{m=0} \sum_{n=0} I(i+m, j+n) g(m,n) \tag{2.20}$$

Auf jeder Position in der Matrix I wird der Filter g angewandt. Die Summe der Multiplikation wird an der Stelle (i,j) in der Ausgabe Matrix I' gespeichert. Stoppkriterium für die Summen ist jeweils die Breite beziehungsweise die Höhe des Convolution-Filters. Die ersten beiden Convolution-Operationen auf I mit dem Filter g wären somit:

$$I = \begin{pmatrix} 1 & 5 & 6 \\ 7 & 8 & 3 \\ 4 & 2 & 2 \end{pmatrix}, g = \begin{pmatrix} 1 \\ -1 \end{pmatrix} \tag{2.21}$$

$$1.Operation: \begin{pmatrix} 1 \cdot 1 & 5 & 6 \\ 7 \cdot (-1) & 8 & 3 \\ 4 & 2 & 2 \end{pmatrix} \longrightarrow 1 \cdot 1 + 7 \cdot (-1) = (-6) \longrightarrow \begin{pmatrix} (-6) & 5 & 6 \\ 7 & 8 & 3 \\ 4 & 2 & 2 \end{pmatrix}$$

$$2.Operation: \begin{pmatrix} 1 & 5 \cdot 1 & 6 \\ 7 & 8 \cdot (-1) & 3 \\ 4 & 2 & 2 \end{pmatrix} \longrightarrow 5 \cdot 1 + 8 \cdot (-1) = (-3) \longrightarrow \begin{pmatrix} (-6) & (-3) & 6 \\ 7 & 8 & 3 \\ 4 & 2 & 2 \end{pmatrix}$$

Dies wird für alle weiteren Matrizen-Position ausgeführt. Sollte der Filter eine nicht existente Position ausserhalb der Matrix erreichen, wird je nach Konfiguration für diesen Wert eine 0 angenommen oder diese Poisition in I' nicht ausgefüllt.

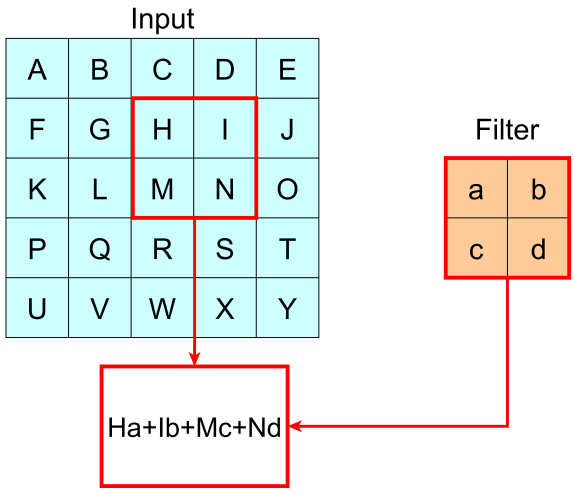

Abbildung 2.10.: Convolution mit einem $4x4$ Filter.

Wird der Convolution-Filter g aus Gleichung 2.21 auf ein Bild angewandt, werden die horizontalen Linien auf dem Bild erkannt und hervorgehobenen.

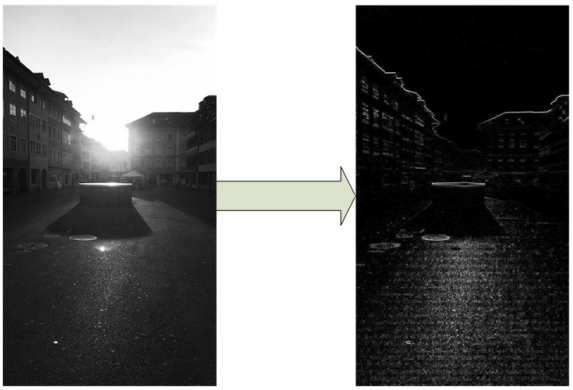

Abbildung 2.11.: Convolution an einem Beispielbild aus der Steinberggasse in Winterthur.

2.5.2. Aufbau CNN

Die Convolution wird als Input für den ersten Dense-Layer eines neuronalen Netzwerk verwendet. Die Backpropagation ändert neben den Parametern der Dense-Layer auch die Gewichte in den Convolution-Kernels. Um den Speicherplatz der neuronalen Netzwerke zu verringern werden nach Convolution-Layern Max-Pooling-Layer verwendet. Sie verringern die Daten indem sie jeweils in einem definierten Kernel nur den grössten Wert an den nächsten Layer weitergeben.

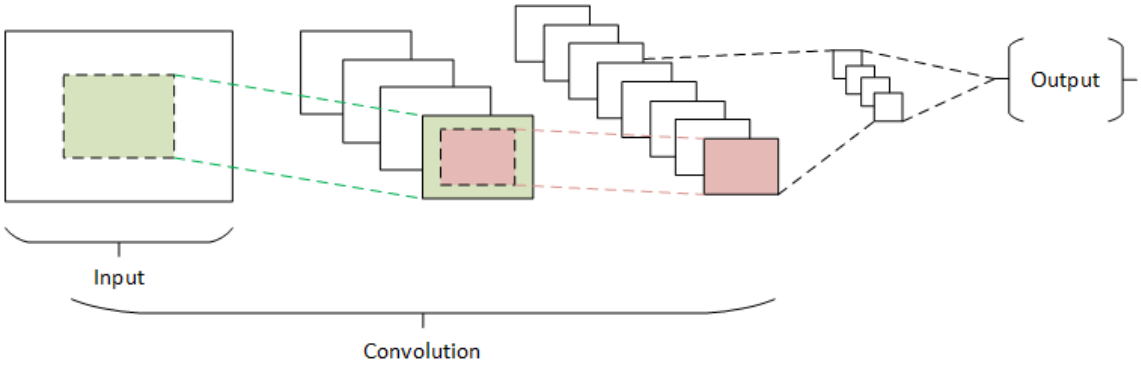

Abbildung 2.12.: Aufbau CNN.

In Abbildung 2.12 wird der Aufbau eines CNNs dargestellt. Es besteht aus einer Input-Matrix und zwei Convolution-Layern mit jeweils 4 beziehungsweise 8 Kernels. Der Output wird an 4 Neuronen eines Dense-Layers weitergegeben. Einfachheitshalber wurde auf Max-Pooling Layer verzichtet und nur ein Dense-Layer verwendet.

2.6. Recurrent Neural Networks

Recurrent Neural Networks (im Folgenden RNN) sind mathematisch gesehen dynamische Systeme. Dynamische Systeme können mit der zeitlichen Veränderung von physikalischen Systemen verglichen werden. Beispielsweise die Berechnung der Umlaufbahnen mehrerer Planeten unter Einbezug der Anziehungskräfte der jeweils anderen Planeten [21]. RNNs halten in ihrem internen Zustand eine nicht lineare Transformation der vorhergegangenen Eingaben, die sich mit jeder neuen Eingabe ändert. Somit ist jede Eingabe von den vergangenen Eingaben abhängig. Dies ist der Unterschied zu klassischen Feedforward Netzwerken die nach dem Training eine gleichbleibende Funktion als internen Zustand erhalten. Was zur Folge hat, dass die selbe Eingabe immer die gleiche Ausgabe ergibt [22].

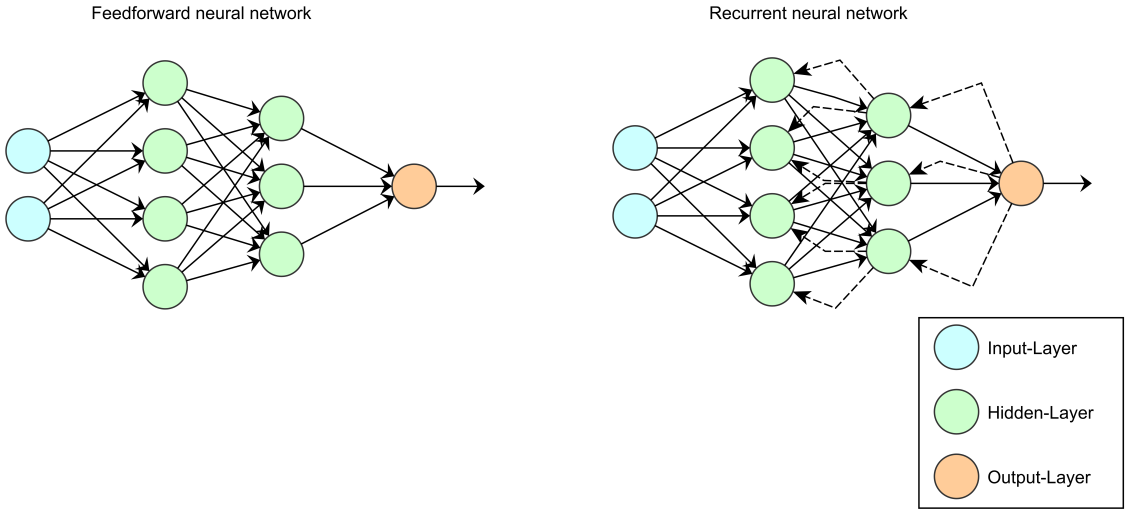

Abbildung 2.13.: Vergleich Feedforward neuronalen Netzwerk mit Recurrent neuronalen Netzwerk.

Neuronen können Verbindung zu allen Neuronen in anderen Layern besitzen, egal ob dies dem normalen Fluss von Input nach Output entspricht oder entgegengesetzt. Der Output kann jeweils dem nächsten Input angehängt werden. So ist das Trainieren von Sequenzen möglich. Ein Netzwerk hat ein Gedächtnis für die vorhergegangen Inputdaten [22].

2.7. Clusteringansätze

Im Folgenden beschreiben wir eine Auswahl an interessanten Clusteringansätze die mittels Deep Learning realisiert werden können.

2.7.1. Siamesische Architektur

Siamesiche neuronale Netzwerke sind eine Kombination von zwei neuronalen Netzwerken. Sie werden verwendet um zwei Patterns zu vergleichen. Der Output ist ein Unterschiedsmass der Inputdaten [23]. Inputdaten werden als Paar aufbereitet und jeweils in die zwei Netzwerke eingegeben. Die Outputvektoren des Netzwerks werden verglichen und ein Distanzmass (z.B. euklidische Distanz) berechnet. Diese Distanz wird in das Netzwerk zurückpropagiert. Hier gibt es unterschiedliche Vorgehensweisen. In den meisten Fällen wird der Fehler in einem Netzwerk zurück gerechnet und die errechneten Gewichte in das zweite kopiert.

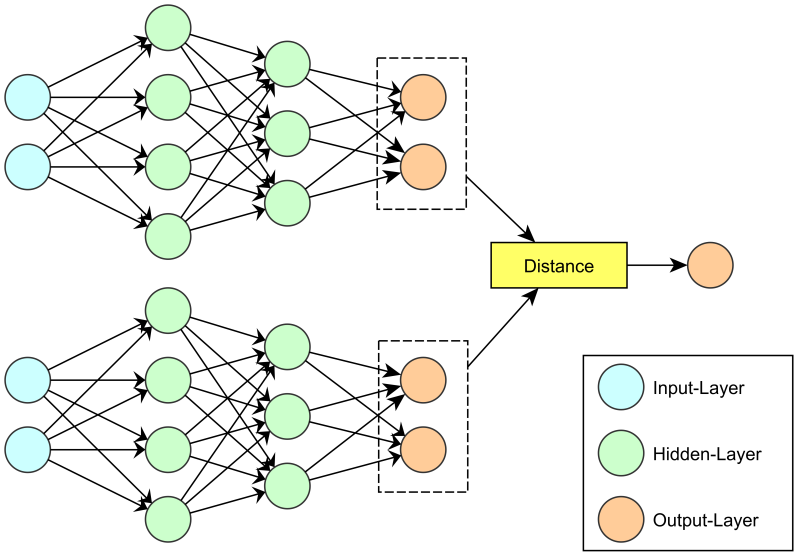

Abbildung 2.14.: Siamesische Architektur.

2.7.2. Paarweise KL-Divergenz

Diese Clusteringmethode wird in [24] vorgestellt. Das Verfahren ähnelt stark jenem einer siamesischen Architektur das die KL-Divergenz als Distanzmass einsetzt Sie kann aber auch bei dieser Architektur durch ein beliebiges Distanzmass ersetzt werden. Der grundlegende Unterschied ist, dass die Inputdaten nicht paarweise, sondern in Mini-Batches eingegeben werden. Die Loss-Funktion wird dann auf allen möglichen Paaren eines Mini-Batches berechnet und dann zurückpropagiert. Daraus entsteht der Performancevorteil, dass man weniger durch das Netzwerk propagieren muss um mit beliebig vielen Daten, in einem Schritt, die Parameteranpassungen durchzuführen.
Für die Berechnung des Losses seien die Verteilungen $\mathbf{P}$ und $\mathbf{Q}$ gegeben, die durch die Eingabe von x_p und x_q in das Netzwerk f berechnet wurden. Nimmt man für die Anzahl Outputnodes des Netzwerks k an, dann ist die totale Divergenz zwischen $\mathbf{P}$ und $\mathbf{Q}$ die Summe über k Nodes. Um dies als Loss-Funktion zu verwenden, wird definiert, dass wenn $\mathbf{P}$ und $\mathbf{Q}$ von einem ähnlichen Paar (x_p, x_q) kommen, der Loss die reine KL-Divergenz ist, andernfalls ist er der Hinge-Loss. Der Indikator I_s in Gleichung 2.24 ist gleich 1 wenn (x_p, x_q) ein ähnliches Paar ist und I_{ds} ist gleich 1 wenn es sich um ein unterschiedliches Paar handelt, anderfalls 0.

$$\mathbf{P} = f(x_p), \mathbf{Q} = f(x_q) \tag{2.22}$$

$$KL(\mathbf{P} \parallel \mathbf{Q}) = \sum_{i=0}^{k} P_i \log(\frac{P_i}{Q_i}) \tag{2.23}$$

$$loss(\mathbf{P} \parallel \mathbf{Q}) = I_s(x_p, x_q) \cdot KL(\mathbf{P} \parallel \mathbf{Q}) + I_{ds}(x_p, x_q) \cdot max(0, margin - KL(\mathbf{P} \parallel \mathbf{Q})) \tag{2.24}$$

Der Fehler für das Paar $\mathbf{P}$ und $\mathbf{Q}$ muss jedoch symmetrisch berechnet werden. Der totale Fehler L des Paars (x_p, x_q) ist also die Summe beider Richtungen:

$$L(\mathbf{P}, \mathbf{Q}) = loss(\mathbf{P} \parallel \mathbf{Q}) + loss(\mathbf{Q} \parallel \mathbf{P}) \tag{2.25}$$

2.7.3. Sprecher-Embeddings

Bei Sprecher-Embeddings werden die Aktivierungen eines Hidden-Layers als Sprecherrepräsentation verwendet[25]. Die Architektur dieser Netzwerke wird meist so aufgebaut, dass die Anzahl Neuronen pro Layer immer kleiner wird bis zum Embedding-Layer und danach wieder grösser. Diese Architektur gleicht der eines Autoencoders[26]. Dies ist jedoch nicht zwingend. So kann eine Repräsentation für einen Sprecher ausgelesen werden, die noch nicht auf den eigentlichen Identifikationstask optimiert wurde, sondern möglichst allgemeinen Features entspricht.

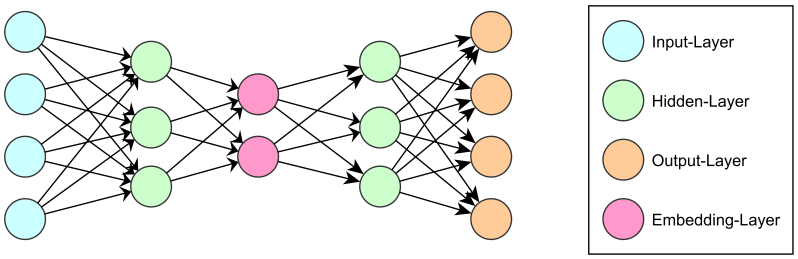

Abbildung 2.15.: Netzwerk mit einem Embedding-Layer.

2.8. Spektrogramm

Ein Spektrogramm ist eine dreidimensionale Darstellung von einem spektralen Inhalt eines Audiosignals. Dabei gibt die x-Achse den zeitlichen Kontext wieder und die y-Achse zeigt die Frequenz an. Farblich wird die Intensität jeder Frequenz zu einem gegebenen Zeitpunkt abgebildet, das Farbspektrum reicht dabei von blau (leise) bis rot (laut). Normalerweise wird die Zeit in Sekunden, die Frequenz in Hertz und die Intensität in Dezibel angegeben[27].

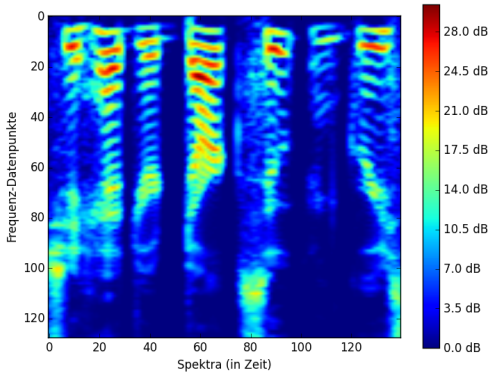

Abbildung 2.16.: Beispiel eines Spektrogramms.

3. Vorgehen und Methoden

Dieses Kapitel beschreibt unsere Grundüberlegungen zur vorliegenden Arbeit. Als erstes beschreiben wir unsere gewählten Ansätze sowie die damit verbundenen konzeptionellen Entscheidungen. Des Weiteren erklären wir die wichtigsten Messwerte sowie die verwendete externe Software.

3.1. Auswahl der Ansätze

In dieser Arbeit verwenden wir zwei unterschiedliche Ansätze. Für beide führen wir verschiedene Experimente durch. Die Ansätze vereinen dabei teilweise mehrere Clusteringansätze, die in Abschnitt 2.7 beschrieben werden.

3.1.1. Clustering mittels Identifikationsnetzwerk

In [1] lernte das Netzwerk die Sprecheridentifikation. Das Netzwerk wird aber unempfindlich gegenüber unbekannten Sprechern. Dem sollte entgegengewirkt werden indem für unbekannte Sprecher ein Sprecher-Embedding[25] als Repräsentation gewählt wird.

Den Experimenten aus [1] liegen einige Fehler und Schwachstellen zugrunde. Diese sind im folgenden kurz aufgeführt:

- Die Frequenz- und die Zeitrichtung des Convolutionfilters wurden vertauscht. Somit basieren alle Experimente auf einem Filter in Frequenzrichtung und nicht wie beschrieben in Zeitrichtung.
- Das Pooling und der Stride des Max-Pooling-Layers wurden stets in beide Richtungen durchgeführt, was nicht im Einklang mit dem gewählten eindimensionalen Filter steht.
- Die Spektrogramme wurden mit einer Sampling-Rate von $22'050$Hz generiert obwohl die TIMIT-Daten über eine Sampling-Rate von 16kHz verfügen. Dies resultiert in schlechter aufgelösten Spektrogrammen.

Für die Experimente in Abschnitt 5 haben wir diese Punkte wie folgt korrigiert:

- In Abschnitt 5.2 führen wir eine Evaluation der Filter durch. Dabei wird im Unterschied zu [1] auch das Pooling und der Stride miteinbezogen.
- Die Spektrogramme werden mit der korrekten Sampling-Rate von 16kHz generiert. Genaueres zur Generierung der Spektrogramme erklären wir in Abschnitt 3.2.3.

Neben einigen erneut durchgeführten Experimenten, ergänzen wir die Versuchsreihe um die Evaluierung des Sprecherclusterings basierend auf den Aktivierungen von Hidden-Layern die sich nach der Convolution befinden. Wir wenden hier das Prinzip des Sprecher-Embeddings an[25]. Hierführ evaluieren wir jeweils alle Layer und haben keinen spezifischen Embedding-Layer. Diese Experimente werden für Netzwerke durchgeführt, die sowohl auf die Identifikation von 100 als auch 590 Personen trainiert sind.

3.1.2. Clustering Netzwerk mit paarweiser KL-Divergenz als Loss-Funktion

Wir übernehmen das Trainingskonzept der paarweisen KL-Divergenz aus [24]. Dabei fliessen ebenfalls Aspekte eines siamesischen Netzwerks mit ein. Wir überprüfen auch bei diesem Ansatz die Möglichkeiten der Aktivierungen der Hidden-Layer beim Clustering und wenden Sprecher-Embedding an.
Um zu ermöglichen, dass wir auf unserer Codebasis aus [1] aufbauen können, implementieren wir das Konzept der paarweisen KL-Divergenz selbst. Als Referenz dient die Originalimplementation von [24], die mittels der Lua-Bibliothek `torch.nn` geschrieben ist.

3.2. Konzept

3.2.1. Grundarchitektur: Identifikationsnetzwerk

Unsere Architektur (siehe Abb. 3.1) basiert auf [28]: Das Netzwerk hat 32 respektive 64 Filter in seinen zwei Convolution-Layers (L1 und L3). Diese haben die Grösse $F \times T$ für die Frequenz (F) und Zeit (T). Wir evaluieren die optimale Filtergrösse in Abschnitt 5.2. Jeder Convolution (L1 und L3) folgt ein Max-Pooling-Layer (L2 und L4) mit Pooling-Size 4×4 und Stride 2×2 (während der Evaluierung testen wir auch eindimensionale Poolings und Strides). Das Netzwerk wird durch zwei Dense-Layers (L5 und L7) abgeschlossen mit jeweils $10 \cdot n_s$ respektive $5 \cdot n_s$ Neuronen, wobei n_s die Anzahl der zu identifizierenden Sprecher darstellt. Wir verwenden die Rectifier-Funktion als Aktivierungsfunktion für alle Neuronen und wenden die Softmax-Funktion auf den Output-Layer (L8) an. Dieser hat n_s Neuronen, wobei jedem Sprecher genau einer zugeordnet ist. Um Overfitting entgegenzuwirken fügen wir einen Dropout-Layer (L6) mit einer Dropout-Rate von 0.5 zwischen den Dense-Layern (L5 und L7) ein. Die Grundarchitektur bleibt stets dieselbe, wir erhöhen lediglich die Komplexität wenn die Anzahl der zu identifizierenden Sprecher erhöht wird.

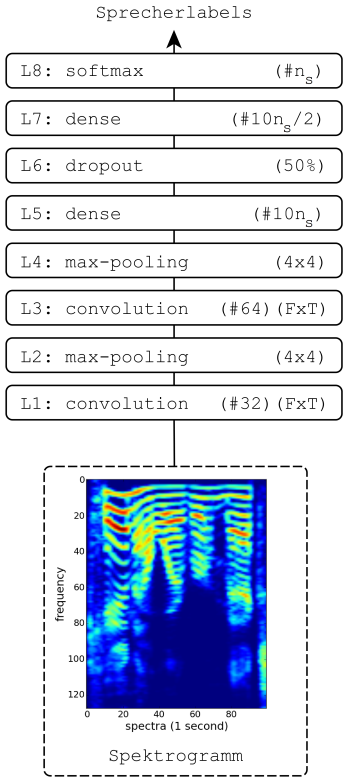

Abbildung 3.1.: Basisarchitektur für alle Identifikationsnetzwerke.

3.2.2. Grundarchitektur: KL-Divergenz Netzwerk

Die Grundarchitektur für das KL-Divergenz Netzwerk ist sehr ähnlich wie jene für das Identifizieren von Sprechern. Jedoch erweiterten wir das Netzwerk zwischen den Convolution-Layern (L1 und L4) und den Max-Pooling-Layern (L3 und L4) sowie zwischen den Dense-Layern (L7 und L10) um Batch-Normalization-Layers [29] (L2, L5 und L8). Dies ist inspiriert durch [24], woher auch die Idee für das KL-Divergenz Netzwerk stammt. Sie verwenden jedoch eine an *LeNet* [30] angelehnte Architektur, die sich aber nicht massgeblich von der unsrigen unterscheidet.

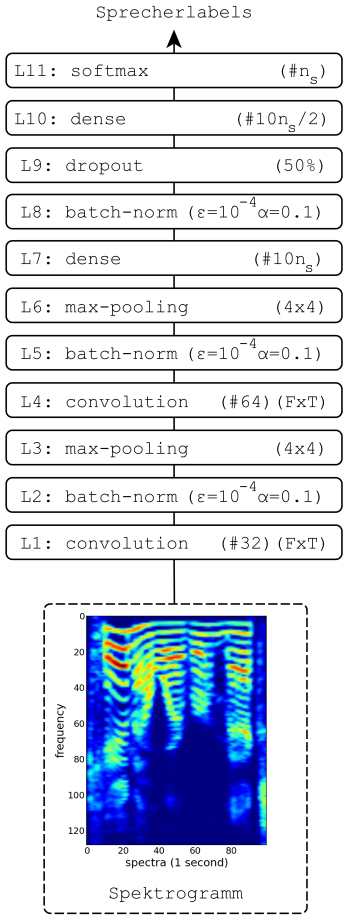

Abbildung 3.2.: Basisarchitektur für alle Netzwerke, die die paarweise KL-Divergenz als Loss-Funktion verwenden.

3.2.3. Form der Eingabedaten

Als Input für die neuronalen Netzwerke dienen Mel-Spektrogramme. Diese haben sich bereits in diversen Arbeiten bewährt [28] [1]. Durch die Anwendung der Mel-Skala werden die Audio-Features näher an die Funktionsweise des menschlichen Gehörs herangeführt. So können Menschen bei niedrigen Frequenzen kleine Änderungen in der Tonhöhe besser wahrnehmen, als bei hohen Frequenzen. Um die Frequenz in die Mel-Skala umzurechnen, wird die folgende Formel verwendet:

$$M(f) = 1125 \cdot ln(1 + \frac{f}{700}) \tag{3.1}$$

Für die Generierung der Spektrogramme wird auf die Python-Bibliothek `librosa` zurückgegriffen. Wir verwenden dabei die Sampling Rate der TIMIT Daten (16kHz), 1024 Samples als *FFT-Window* Länge

und 160 Samples als *hop length*. Daraus resultieren 100 Datenpunkte in Zeitrichtung für eine Sekunde Audiodaten und 128 Elemente in Frequenzrichtung. Anschliessend wenden wir eine elementweise Kompression an, $f(x) = \log(1 + C \cdot x)$ [28]. Durch das Erhöhen der Konstante C werden Werte mit schwacher Intensität im hochfrequenten Spektrum prominenter. Wir setzen $C = 10'000$ [31].

Damit die `lasagne`-Bibliothek Daten verarbeiten kann, müssen diese in eine definierte Datenstruktur gebracht werden. Dabei handelt es sich um eine 4D-Matrix mit folgender Struktur:

$$X[Sprechersatz, Kanal, Frequenz, Zeit] \tag{3.2}$$

Für jeden Satz wird ein Eintrag erstellt. Wir verwenden immer nur einen Kanal, die Frequenz hat 128 und die Zeit 800 Elemente. Die Zeitrichtung sollte dabei eigentlich dynamisch sein, jedoch wurde hier einfachheitshalber eine fixe Breite gewählt, da `Theano`, die Bilbiothek auf die `Lasagne` aufbaut, nicht ohne weiteres mit verschiedenen Dimensionen der Input-Matrix umgehen kann. Wir wählten deshalb eine Breite in die die längsten Sätze reinpassen. Daraus resultiert aber, dass bei kurzen Sätzen Bildabschnitte ohne Informationen gebildet werden. Diese werden beim Zusammenbauen der Mini-Batches weggeschnitten, was während dem Training geschieht. Der genaue Trainingsablauf ist im Kapitel Implementation unter dem Abschnitt 4.1.2 beschrieben.

Die Soll-Ergebnisse werden in einem 1D-Array gespeichert.

$$y[Sprecher] \tag{3.3}$$

Dabei wird einfachheitshalber auf die Sprecherlabels verzichtet und an deren Stelle ganzzahlige Nummern verwendet.

3.3. Datensätze

3.3.1. TIMIT Datensatz

Für die Daten wird auf den TIMIT Datensatz zurückgegriffen. Dieser umfasst 630 Sprecher, mit je 10 Sätzen in Studioqualität. Von diesen sind 192 weiblich und 438 männlich. Die Sprecher sind in die acht Hauptdialekte der USA unterteilt.
In allen Experimenten die mit Spektrogrammen arbeiten, verwenden wir den TIMIT Datensatz. Wir arbeiten jeweils mit den kompletten Daten. Dabei werden 40 Sprecher für das Clustering verwendet und die restlichen 590 oder Teilmengen daraus von 100 oder 10 Sprechern für das Training der Netzwerke. Aus den 40 Sprechern werden wiederum eine Teilmenge von 20 Sprechern extrahiert. Dieselben Daten verwendeten wir bereits in [1] und sie wurden ebenfalls von Thilo Stadelmann und Bernd Freisleben in [32] verwendet. In einige Experimenten verwenden wir auch Teilmengen von lediglich 10 oder 5 aus den 40 Sprechern. Wir nennen diese 40 Sprecher im Verlauf dieser Arbeit jeweils den *Reynolds Datensatz*. Die Sprecherlisten des Reynolds Datensatzes und der 100 Sprecher die für das Training verwendet werden, sind im Anhang A.2.2 aufgeführt.

3.3.2. MNIST Datensatz

Der MNIST Datensatz [33] ist eine Sammlung von handgeschriebenen Zahlen. Das Trainingsset umfasst $60'000$ und das Testset $10'000$ Beispiele. Die Zahlen sind in den Grössen normalisiert und in einem Bild fixer Grösse (28×28 Pixel) zentriert [30].
Wir verwenden den MNIST Datensatz in Experiment 6.2 um das Experiment aus [24] nachzubauen und somit unsere Implementation der paarweisen KL-Divergenz zu verifizieren.

3.4. Experimenteller Aufbau

Wir verwenden für das Clustering neben den gleichen Daten wie in [32] und [1] auch den gleichen experimentellen Aufbau. Das Clustering führen wir nicht auf Segmentebene durch, sondern fassen für jeden Sprecher jeweils die acht Sätze des Trainingsdatensatzes und die zwei Sätze des Testdatensatzes zu zwei Featurevektoren (Sprecherrepräsentationen) zusammen. Über alle Featurevektoren der 40 Sprecher oder der jeweiligen Teilmenge führen wir dann ein hierarchisches Clustering durch. Hierfür verwenden wir die Complete-Linkage in Verbindung mit der Cosinus-Metrik. Die Complete-Linkage bewährte sich bereits für andere Sprecherclustering Ansätze [34] [35]. In [1] ergaben unsere Tests, dass zusammen mit der Complete-Linkage, die Cosinus-Metrik die besten Ergebnisse liefert.
Ziel der Clusterings ist es, dass die zwei Featurevektoren jedes Sprechers zusammengefasst werden, bevor der Featurevektor eines Sprechers dem eines anderen zugewiesen wird.

3.5. Messgrössen

Neuronale Netzwerke können mittels verschiedenen Messwerten bewertet werden. Im Folgenden wird erklärt, welche im Verlauf dieser Arbeit verwendet werden und für welchen Bereich diese Anwendung finden.

3.5.1. Accuracy

Die Accuracy (Genauigkeit) eines Netzwerks wird meistens dann eingesetzt, wenn ein Netzwerk auf eine Klassifizierungsaufgabe trainiert wird. Die Accuracy ist wie folgt definiert [36]:

$$accuracy(t) = \frac{t}{N} \tag{3.4}$$

N ist dabei die totale Anzahl Objekte und *t* sind die richtig klassifizierten Objekte.
Wir verwenden die Accuracy um die Performance unserer Identifikationsnetzwerke zu testen. Sie findet auch Anwendung um zu verifizieren ob die Clusteringnetzwerke lernen.

3.5.2. Misclassification Rate (MR)

Als Messgrösse für die Qualität des Clusterings verwenden wir die Misclassification Rate (MR) [37]. Diese ist wie folgt definiert:

$$MR = \frac{1}{N} \sum_{j=1}^{N_s} e_j \tag{3.5}$$

Dabei beschreibt N die Anzahl Audio-Segmente, N_s die Anzahl Sprecher und e_j jene Segmente von Sprecher j, die nicht richtig zugewiesen sind. Die Misclassification Rate nimmt jeweils einen Wert zwischen 0 und 1 an. Ein kleiner Wert steht für eine kleine Wahrscheinlichkeit falsch zugewiesener Segmente.

3.6. Verwendete Bibliotheken

Für die Datenaufbereitung, die Implementation des CNNs und die Auswertungen sind zusätzlich zu den Standard-Bibliotheken von Python, noch weitere nötig. Diese werden im Folgenden kurz, mit der verwendeten Versionsnummer, aufgelistet.

Datenaufbereitung

- `NumPy 1.9.2`
- `SciPy 0.16.0`
- `librosa 0.4.1`

CNN

- `Theano 0.7.0`
- `Lasagne 0.2.dev1`
- `nolearn.lasagne 0.6a0.dev0`

Auswertung

- `munkres 1.0.7`

3.7. Hardware

Das Training der Netzwerke wird auf einem Datalab Server des InITs durchgeführt. Dieser verfügt über einen 2.10 GHz Intel(R) Xeon(R) Prozessor mit 6 Kernen, zwei GeForce GTX 780 Grafikkarten und 130 GB 1600 MHz DDR3 Arbeitsspeicher. Als Betriebssystem wird Ubuntu 14.04.3 verwendet.

4. Implementation

Das folgende Kapitel erklärt die Implementationen der wichtigsten Komponenten die notwendig sind um unsere Experimente durchzuführen.

4.1. Allgemein

4.1.1. Datenaufbereitung

Die Datenaufbereitung funktioniert für alle Experimente gleich. Sie ist in Abbildung 4.1 als Aktivitätsdiagramm dargestellt [1].

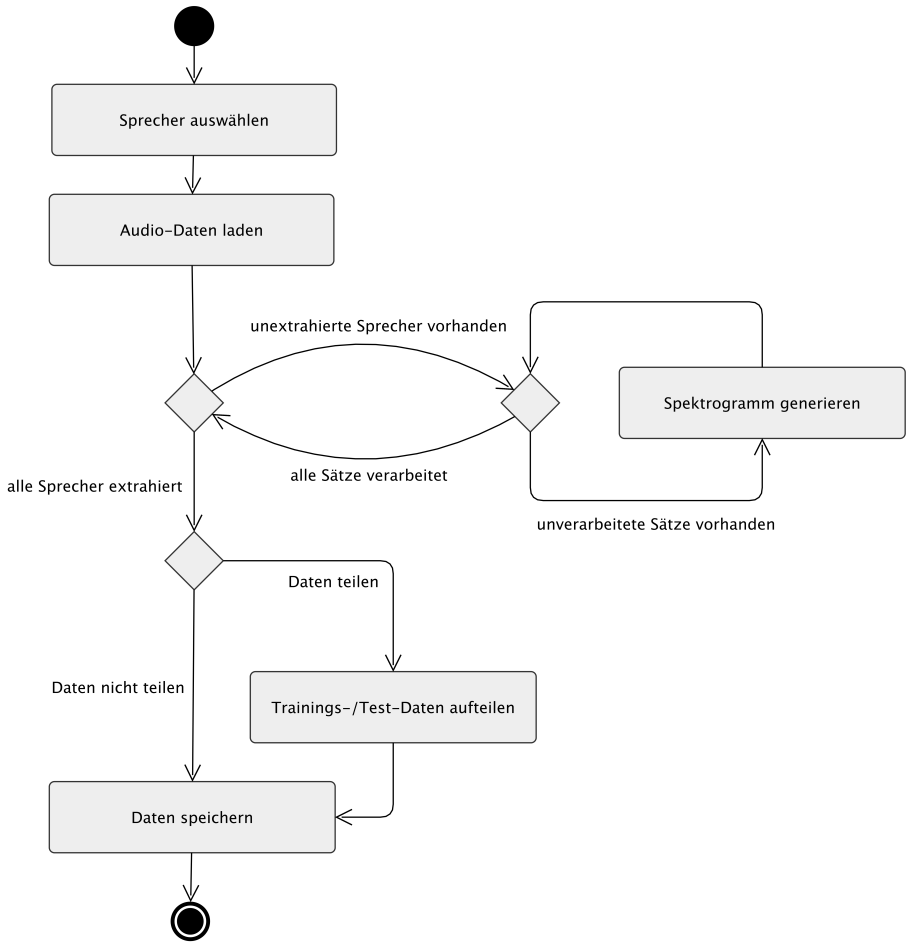

Abbildung 4.1.: Der Prozess der Datenaufbereitung.

- **Sprecher auswählen**: Die Auswahl geschieht über Textdateien, die Listen von Sprechern enthalten. Um zufällige Listen mit Sprechern zu generieren, kann *speaker_corpus_generator.py* verwendet werden.

- **Audio-Daten laden**: Die Funktion *DataExtractor.traverse_TIMIT_data* in *data_extractor.py* traversiert die TIMIT Daten und lädt die entsprechenden WAV-Dateien der Sprecher.
- **Spektrogramm generieren**: Für jede WAV-Datei wird ein Spektrogramm erstellt. Dies geschieht über die Funktion *mel_spectrogram* in *spectrogram_converter.py*.
- **Trainigs-/Test-Daten aufteilen**: Beim Extrahieren der Spektrogramme kann entschieden werden, ob die Daten in Trainings- und Testdaten aufgeteilt werden sollen. Dies geschieht dann durch die Klasse *SpeakerTrainSplit* in *speaker_train_splitter.py*
- **Daten speichern**: Die generierten Daten werden durch die Python-Bibliothek `cPickle` im pickle-Dateiformat gespeichert.

4.1.2. Training

Das Training unterscheidet sich für die beiden Ansätze lediglich in der Berechnung der Loss-Funktion. Im Folgenden gehen wir auf die grobe Struktur ein, die für beide identisch ist [1].

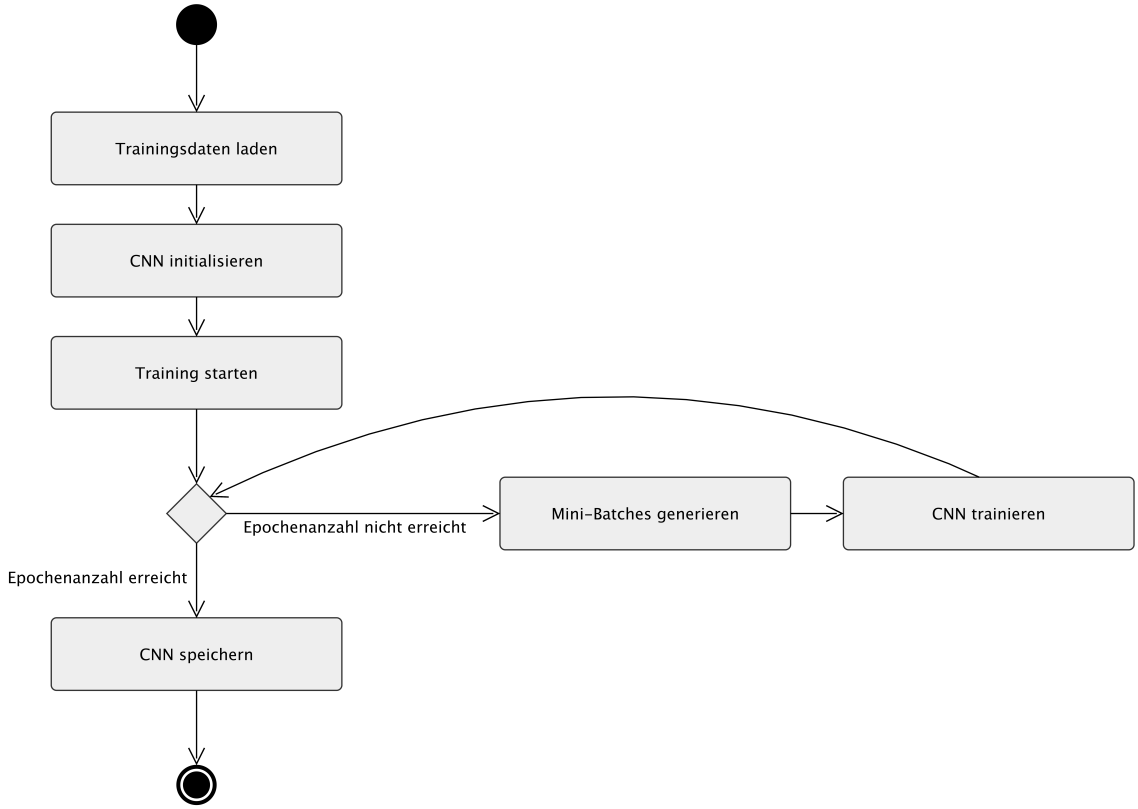

Abbildung 4.2.: Der grobe Trainingsablauf eines Netzwerks.

- **Trainingsdaten laden**: Die Trainingsdaten liegen im pickle-Dateiformat vor und werden mithilfe der Python-Bibliothek `cPickle` geladen. Dabei erhält man für jeden Satz ein Spektrogramm. Die Aufteilung in Sekundensegmente geschieht später.
- **CNN initialisieren**: Experimente mit einem Identifikationsnetzwerk initialisieren das CNN mithilfe der Klasse *NeuralNet* der Python-Bibliothek `nolearn.lasagne`. Die anderen Experimente verwenden direkt die Bibliotheken `Lasagne` und `Theano`.
- **Training starten**: Das Training wird gestartet indem der Trainingsloop angestossen wird.

- **Mini-Batches generieren**: Es werden jeweils Sekunden-Segmente aus dem Spektrogramm extrahiert. Um eine möglichst hohe Diversität in den Trainingsdaten zu erreichen, erfolgt die Auswahl der Sekunden-Segmente zufällig. Zuerst wird ein zufälliges Spektrogramm ausgewählt und anschliessend ein zufälliges Sekunden-Segment aus diesem Spektrogramm. Ein Mini-Batch beinhaltet bei den Identifikationsnetzwerken 128 Segmente und bei der KL-Divergenz Netzwerken 100.
- **CNN trainieren**: Die Netzwerke werden mittels Mini-Batch Gradient Descent trainiert. Alle Identifikationsnetzwerke wenden noch ein Nesterov Momentum als Optimierung an und verwenden als Loss-Funktion die *Kreuzentropie*. Für die Netzwerke mit paarweiser KL-Divergenz als Loss-Funktion testen wir in Abschnitt 6.6 verschiedene Optimierungsverfahren.
- **CNN speichern**: Die Parameter des trainierten CNNs werden mittels `cPickle` im pickle-Dateiformat gespeichert.

4.2. Paarweise KL-Divergenz

Für die Netzwerke die die KL-Divergenz als Loss-Funktion verwenden, entschieden wir uns dagegen `nolearn.lasagne` zu verwenden, da die Eingriffe in die Implementation zu tiefgreifend und somit zu aufwendig wären. Wir verwenden deshalb direkt die `Lasagne`-Bibliothek und stellenweise das darunterliegende `Theano`. Für diese Experimente ist deshalb einiges mehr an Implementationsarbeit nötig. Im Folgenden beschreiben wir die drei umfangreichsten Aspekte unserer Implementation, die Initialisierung, die Berechnung der Loss-Funktion und das Testen der Accuracy.

4.2.1. Initialisierung des Netzwerks

Durch den direkten Einsatz von `Lasagne` und `Theano` haben wir die Möglichkeit den gesamten symbolischen Graphen, der den Netzwerken zugrundeliegt, am Stück zu programmieren. Verwendet man `nolearn.lasagne` müssen die Komponenten, wie zum Beispiel die Kostenfunktion, einzeln überschrieben werden. Des Weiteren wollen wir sicherstellen, dass wir die volle Kontrolle über die einzelnen Aspekte unserer Experimente haben, was durch die eigene Implementation der Netzwerkinitalisierung und des Trainings- wie auch des Validierungsloops erreicht wird.

4.2.2. Berechnung der Loss-Funktion

Um den Loss eines Minibatches zu berechnen müssen wir daraus alle möglichen Paare bilden. Somit sind für die Berechnung der Loss-Funktion zwei verschachtelte Loops nötig. Da wir jedoch symbolisch rechnen, kann dies nicht mit normalem Pythoncode implementiert werden. Die einzige Möglichkeit um mit `Theano` ein Loop zu implementieren, ist das Verwenden der *scan*-Funktion. Diese ist nicht besonders intuitiv, deshalb erklären wir im Folgenden deren Funktionsweise soweit, dass deren Einsatz in dieser Arbeit verstanden werden kann.
Die Funktion wird von `Theano` mehrheitlich für Berechnungen bezüglich RNNs verwendet. Deshalb hat die Funktion einige Parameter die für uns nicht relevant sind (z.B. *truncate_gradient*). Wir verwenden die folgenden Parameter und lassen die restlichen auf ihrem Standardwert:

- **fn**: Die Funktion die für jede Iteration aufgerufen wird. Dabei muss darauf geachtet werden, dass die Funktion über die nötigen Parameter verfügen muss. Wie sich diese zusammenbauen hängt von den restlichen Parametern ab, die der *scan*-Funktion übergeben werden.
- **sequences**: Eine Liste von Listen über die schrittweise iteriert werden soll. Einzelne Elemente dieser Listen werden als Parameter der Funktion *fn* übergeben. Die Anzahl Parameter entspricht dabei der Anzahl übergebener Listen in diesem Parameter, des Weiteren sind diese die ersten Parameter der Funktion *fn*.

- **non_sequences**: Eine Liste von Objekten die ebenfalls der Funktion *fn* übergeben werden sollen. Im Vergleich zu den Listen die mittels *sequences* übergeben werden, wird über diese Objekte nicht iteriert, sondern diese sind für alle Iterationsschritte gleich.
- **strict**: Wenn dieses Flag gesetzt wird, müssen alle Parameter der Funktion *fn* durch *sequences* oder *non_sequences* gesetzt werden. Andernfalls wird ein Fehler ausgegeben. Ist das Flag gesetzt resultiert das in einigen Optimierungen, was bei mehreren Iterationen einen positiven Einfluss auf die Performance haben kann.

Da zwei Loops nötig sind, sind auch zwei *scan*-Aufrufe nötig. In Listing 4.1 ist unsere Implementation dieser Loops aufgezeigt. Beide *scan*-Aufrufe iterieren über die gleichen Listen. Dabei sind *predictions* die berechneten Output-Vektoren und *targets* die Sollklassen (in unserem Fall der Sprecherindex). Diese werden mittels dem *sequence*-Parameter übergeben. Die mittels *non_sequences* als *Shared Variables* übergebenen Parameter unterscheiden sich jedoch. Lediglich der *margin*-Parameter ist bei beiden enthalten, dieser ist jedoch lediglich eine Konstante. In der Funktion *foreach* geben wir nochmals die gleichen Listen als *Shared Variables* mit, wie jene über die iteriert wird. Dadurch erhält die *step*-Funktion bei jedem Aufruf nicht nur den nächsten Output-Vektor (*prediction*), zusammen mit dessen Klassenzugehörigkeit (*target*), sondern auch die gesamten Listen von Output-Vektoren und Klassenzugehörigkeiten und die *margin*-Konstante. In der *step*-Funktion muss nun jedes mögliche Paar mit dem gegebenen Output-Vektor (*prediction*) gebildet werden. Hierzu wird erneut über alle Output-Vektoren und Klassenzugehörigkeiten iteriert und der aktuelle Output-Vektor und dessen Klassenzugehörigkeit werden zusammen mit der *margin*-Konstante als *Shared Variablen* an die eigentliche Funktion zur Berechnung des Losses übergeben.

```
1  def foreach(predictions, targets, margin):
2    result, updates = theano.scan(fn=step,
3                sequences=[predictions, targets],
4                non_sequences=[predictions, targets, margin],
5                strict=True)
6    return result
7
8
9  def step(prediction, target, predictions, targets, margin):
10   result, updates = theano.scan(fn=loss_with_kl_div,
11               sequences=[predictions, targets],
12               non_sequences=[prediction, target, margin],
13               strict=True)
14   return result
```

Listing 4.1: Unsere Implementation zweier verschachtelter Loops mittels *scan*-Funktion

Der Ablauf für die Berechnung des Losses für einen Mini-Batch ist in Abbildung 4.3 dargestellt und wird im Folgenden genauer beschrieben.

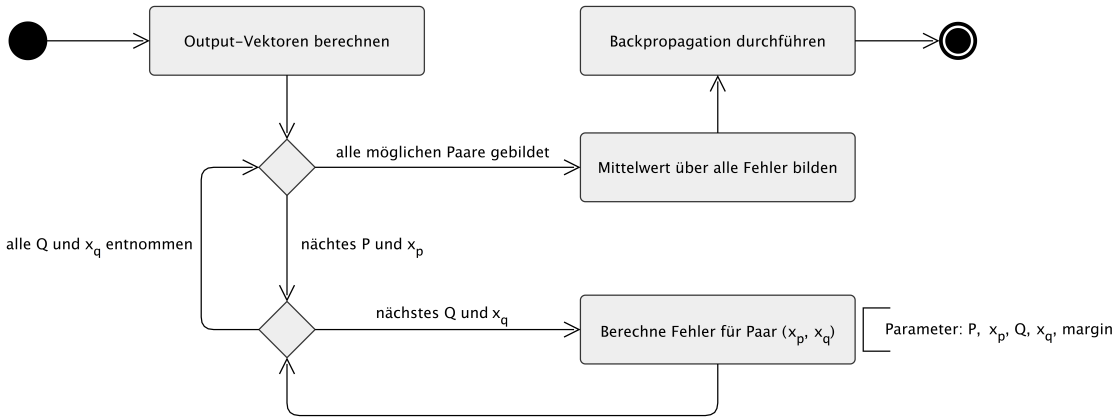

Abbildung 4.3.: Die implementierte Logik für die Berechnung des Losses eines Mini-Batches.

- **Output-Vektoren berechnen**: Berechnet für alle gegebenen Inputs X_b die entsprechenden Output-Vektoren und speichert diese in einem Array.
- **Berechne Loss für Paar** $(\mathbf{x_p}, \mathbf{x_q})$: Der Loss zwischen zwei berechneten Output-Vektoren P und Q wird anhand der paarweisen KL-Divergenz berechnet. Das Paar (x_p, x_q) wird dabei lediglich verwendet um zu bestimmen ob es sich um ein ähnliches oder ein unterschiedliches Paar handelt. Der Parameter *margin* ist eine Konstante und wird als *Shared Variable* an `Theano` übergeben. Die berechneten Losses werden in einem Array gespeichert.
- **Mittelwert über alle Losses bilden**: Es wird der Mittelwert über alle Losses im Array gebildet.
- **Backpropagation durchführen**: Die Netzwerkparameter werden auf Basis des mittleren Losses mittels Backpropagation angepasst.

4.2.3. Testen der Accuracy

Auch wenn wir das Netzwerk für das Clustering von unbekannten Sprechern verwenden wollen, wird es eigentlich als ein Identifikationsnetzwerk trainiert. So sind die einzelnen Sprecher lediglich nicht mehr einem festen Output-Neuron zugewiesen. In [24] lernte das Netzwerk die einzelnen Sprecher einem spezifischen Output-Neuron zuzuordnen, jedoch entschied das Netzwerk selber, welcher Sprecher welches Neuron erhält. Dies erschwert die Berechnung der Accuracy. Um dieses Problem zu lösen, verwenden wir die *Ungarische Methode* (*Hungarian Algorithm*), auch *Kuhn-Munkres-Algorithmus* genannt.

Ungarische Methode

Die *Ungarische Methode*[38] ist ein Algorithmus zum Lösen von gewichteten Zuordnungsproblemen auf bipartiten Graphen, das heisst Knoten lassen sich in zwei disjunkte Teilmengen A und B unterteilen, sodass zwischen den Knoten innerhalb Teilmengen keine Kanten verlaufen. Der Algorithmus versucht nun möglichst effizient die richtige Zuordnung der beiden Teilmengen aufeinander zu finden. So ist je nach Problemstellung das Ziel die Kosten der Zuordnungen zu maximieren oder zu minimieren. In unserem Fall wollen wir die Sprecherlabels den Output-Neuronen so zuweisen, dass über alle Sprecher hinweg eine möglichst hohe Accuracy erreicht wird.
Um das Zuweisungsproblem zu verdeutlichen, betrachte man die Tabelle 4.1. Sie zeigt eine mögliche Konfusionsmatrix, wenn man ein Netzwerk mit drei Output-Neuronen auf drei Sprecher trainieren würde und dabei für jeden Sprecher 10 Samples zur Verfügung hätte. Die Darstellung bedeutet zum Beispiel, dass bei zwei Samples von *Sprecher 1* das *Neuron 1* den höchsten Wert aufwies usw. Mittels der *Ungarischen Methode* kann nun diejenige Zuordnung gefunden werden, dass die Summe maximal wird. Die optimalen Werte sind in der Tabelle 4.1 markiert. Permutiert man nun die Spalten so, dass die Zuordnungen in der Diagonalen dargestellt sind, hat man die gleiche Konfusionsmatrix wie bei einem normalen Identifikationsnetzwerk und kann die Accuracy berechnen.

	Neuron 1	**Neuron 2**	**Neuron 3**
Sprecher 1	2	**5**	3
Sprecher 2	**4**	3	3
Sprecher 3	1	3	**6**

Tabelle 4.1.: Eine mögliche Zuordnung bei drei Sprechern mit je 10 Samples.

Für eine Kostenmaximierung durchläuft der Algorithmus folgende Schritte:

1. Nimm die grösste Zahl c_{max} einer Matrix C und befülle eine neue Matrix $\tilde{C}$ mit $c_{ij} - c_{max}$. Wobei c_{ij} die Elemente von C sind.
2. Bilde die Spaltenminima $\tilde{c}_{jmin}$ von $\tilde{C}$.
3. Rechne für jedes Element $\tilde{c}_{ij} - \tilde{c}_{jmin}$.
4. Bilde die Zeilenminima $\tilde{c}_{imin}$ von $\tilde{C}$.
5. Rechne für jedes Element $\tilde{c}_{ij} - \tilde{c}_{imin}$.
6. Finde eine Kombination von Nullen in $\tilde{C}$ sodass in jeder Zeile und Spalte genau eine Null ausgewählt ist.
7. Findet sich eine solche Kombination sind die gefundenen Stellen der maximale Pfad für die Matrix C.
8. Lässt sich keine Kombination von Nullen finden, dann finde die minimale Anzahl horizontaler und vertikaler Linien, die nötig sind um alle Nullen der Matrix $\tilde{C}$ zu streichen. Sind dabei mindestens n Striche nötig bei einer Matrix mit der Dimension $n \times n$ so ist die optimale Lösung bereits gefunden.
9. Finde das Minimum k_{min} der nicht gestrichenen Koeffizienten.
10. Rechne für alle nicht gestrichenen Koeffizienten $k_{ij} - k_{min}$ und für alle doppelt gestrichenen Koeffizienten $k_{ij} + k_{min}$. Einfach gestrichene Koeffizienten bleiben unverändert.
11. Gehe zu Punkt 6.

Will man eine Kostenminimierung durchführen wird Punkt 1 weggelassen. Das ist für uns jedoch nicht relevant.

Implementation

Wir berechnen die Accuracy jeweils auf den Validierungsdaten nach einer Trainingsepoche um den Lernfortschritt überwachen zu können. Um die *Ungarische Methode* anwenden zu können, benötigen wir jedoch eine Konfusionsmatrix. Diese wird jeweils vor dem Start der Validierung initialisiert und dann fortlaufend aktualisiert. Ist der Validierungsdatensatz vollständig durchgelaufen wird anhand der Konfusionsmatrix die Accuracy berechnet. Hierzu wird zuerst die Kostenmaximierung durch die *Ungarische Methode* durchgeführt und anschliessend gemäss der Formel 3.4 die Accuracy berechnet. Die *Ungarische Methode* wird mithilfe der Python-Bilbiothek `munkres` durchgeführt.

5. Resultate Identifikationsnetz

In den folgenden Experimenten haben wir die Schwachstellen aus [1] behoben. Konkret haben wir die Spektrogramme mit einer Sampling Rate von 16kHz neu generiert und die Filter neu evaluiert. Des Weiteren wenden wir die Idee von *Speaker Embeddings* [25] auf unsere Problemstellung an.

5.1. Architektur

Alle Experimente in diesem Kapitel verwenden die Grundarchitektur aus Abschnitt 3.2.1. Sie sind aber jeweils für die entsprechende Anzahl Sprecher konfiguriert (10, 100 oder 590).

5.2. Experiment 1: Evaluation der Filter

In [1] kamen wir zur Erkenntnis, dass ein Filter in Zeitrichtung besser ist als einer in Frequenzrichtung oder deren Kombination. Wie sich herausstellte, war diese Schlussfolgerung jedoch falsch. Die Filter wurden vertauscht, so dass wir eigentlich bessere Ergebnisse in der Frequenzrichtung erhielten. Des Weiteren wurde ein Max-Pooling und ein Stride in beide Dimensionen verwendet. Das kann die Aussagekraft des Filters schmälern, da dieser dann lediglich in eine Richtung angewendet wird, das Pooling aber hervorgehobene Features der Filterrichtung mit Features aus der anderen Richtung überschreiben kann.
In diesem Experiment wollen wir diese Fehler beheben und erneut die Filter evaluieren. Da wir nun neben den Filtern auch das Pooling und den Stride anpassen müssen, ergeben sich exponentiell mehr mögliche Convolution-Konfigurationen. Um schnell verschiedene Filter zu testen, verwenden wir einen minimalistischen Aufbau für dieses Experiment und evaluieren die besten Konfigurationen dann in einem aufwendigeren Aufbau.
Um schneller Ergebnisse zu erhalten, verwenden wir einen kleinen Datensatz von 10 Sprechern.

Aufbau

Um aussagekräftige Resultate zu erhalten, müssen für alle Konfigurationen die exakt gleichen Trainingsdaten verwendet werden. Um das zu erreichen generieren wir vorab die zu trainierenden Mini-Batches und verwenden dieselben für alle Netzwerke. Des Weiteren muss der *Random Seed* von `Lasagne` auf einen statischen Wert gesetzt werden um jegliche zufälligen Einflüsse zu eliminieren. Auf der CPU sind dann alle Trainings exakt reproduzierbar. Für die GPU reicht das aber noch nicht aus. Wir nehmen an, dass gewisse Optimierungen von `Nvidia Cuda` nicht deterministisch ablaufen. Daraus resultiert, dass auf der GPU nach jedem Trainingsdurchlauf kleine Unterschiede generiert werden, was im Endeffekt jedoch in grossen Unterschieden bezüglich den Accuracies desselben Netzwerkes endet.
Wir führten verschiedene Tests mit unterschiedlich konfigurierten Netzwerken durch. Dabei passten wir die Dimensionen der Filter der Convolution Layer, sowie das Pooling und den Stride der Max-Pooling-Layer an.

Auswertung

In der ersten Phase evaluieren wir eine sehr breite Variation von Konfigurationen. Dabei verwenden wir stets dieselben Trainingsdaten und entfernen die zufälligen Einflussfaktoren. Um ein schnelles Training zu ermöglichen trainieren wir in dieser ersten Phase jedoch auf der GPU, was eine perfekte Reproduzierbarkeit verunmöglicht. Deshalb führen wir die Experimente mehrmals durch und verwenden die jeweils höchste erreichte Accuracy für eine Konfiguration als Referenzwert. Eine Auswahl der erreichten Accuracies der jeweiligen Filter ist in Tabelle 5.1 abgebildet. Man sieht, dass die meisten Konfigurationen sehr nahe beieinander sind, somit kann man bei diesen noch keine Aussage treffen, welche Konfiguration die besten Ergebnisse liefert. Was aber auffällt ist, dass ein Filter in Zeitrichtung mit zweidimensionalem Pooling signifikant schlechtere Ergebnisse liefert, als alle anderen Konfigurationen. Des Weiteren ergibt ein Variieren der Filtergrösse keinen grossen Einfluss auf die Ergebnisse. Was ebenfalls auffällt, ist der Einfluss des Poolings und des Strides auf die Grösse des Netzwerks. So sind diese massiv kleiner und können schneller trainiert werden wenn ein 2D Pooling und ein 2D Stride verwendet wird.

Filter	**Pooling**	**Stride**	**Accuracy**
8×1	4×1	2×1	0.851
1×8	1×4	1×2	0.851
8×8	4×4	2×2	0.830
8×1	4×4	2×2	0.766
1×8	4×4	2×2	0.596

Tabelle 5.1.: Die Accuracies für die Konfigurationen im minimalistischen Experiment. Filter, Pooling und Stride sind mittels der Dimension Frequenz$\times$Zeit ($F \times T$) angegeben.

Im zweiten Teil trainieren wir eine Auswahl von Filtern die gut abgeschnitten haben für eine längere Zeit auf der CPU um eine exakte Reproduzierbarkeit zu gewährleisten. Doch auch dann ändert sich nichts im Vergleich zur obigen Feststellung. Alle Konfigurationen erreichen eine Accuracy von rund 90%.
Somit kommen wir zum Schluss, dass lediglich ein Filter in Zeitrichtung mit zweidimensionalem Pooling und Stride schlechtere Ergebnisse liefert. Wieso jedoch ein Filter in Frequenzrichtung bei zweidimensionalem Pooling und Stride funktioniert, muss geklärt werden. Dies sollte in zukünftigen Arbeiten weiterverfolgt werden. Auch sollte untersucht werden, ob das Netzwerk zeitspezifische Information beachtet, wenn der Filter nur in die Frequenzrichtung angewendet wird. Dies könnte zum Beispiel erreicht werden, indem man bei den Spektrogrammen die Spalten permutiert und dann testet ob das Netzwerk die Sprecher noch gleich gut erkennt. Falls ja, ignoriert es den zeitlichen Kontext komplett. Leider reicht der Umfang dieser Arbeit nicht um diese Experimente durchzuführen.
Wir entschieden uns einen 4×4 Filter mit 4×4 Pooling und 2×2 Stride zu verwenden. So erhalten wir kleinere Netzwerke, was ein einfacheres und schnelleres Training ermöglicht. Zugleich stellen wir sicher, dass die Filter Features der Frequenz- und der Zeitachse lernen können. Für alle weiteren Experimente in dieser Arbeit wird, falls nicht anders vermerkt, mit dieser Konfiguration gearbeitet.

5.3. Experiment 2: Speaker Embedding

In [1] untersuchten wir das Clustering anhand eines Netzwerks, das auf die Sprecheridentifikation trainiert war. Als Feature-Vektor für das Clustering diente dabei der 100-dimensionale Output-Vektor des Netzwerks. Im folgenden Experiment untersuchen wir ob es eine Verbesserung bezüglich des Clusterings gibt, wenn mit einem komplexeren Netzwerk gearbeitet wird und ob die Aktivierungen der Dense-Layer (L5 und L7) besser für das Clustering geeignet sind. Dort sollte das Netzwerk weniger sprecherabhängig sein [25].
Wir trainieren die Netzwerke zuerst für $10'000$ Mini-Batches und dann weitere $5'000$ Mini-Batches unter Einbezugnahme der Validierungsdaten. Der Grund hierfür ist, zuerst zu sehen wie sich die Accuracy auf dem Identifikationstask während des Trainings verhält um sicherzustellen, dass kein Overfitting stattfindet.

Auswertung

Wir führen das Experiment in zwei Phasen durch. In der ersten clustern wir nacheinander über die Aktivierungen aller Layer nach der Convolution (L5, L7 und L8). Wir verwenden dabei die unbekannten 40 Sprecher aus dem Reynolds Datensatz und deren Teilmenge von 20 Sprechern. Im zweiten Teil wollen wir feststellen, wie sich das Clustering für bekannte Sprecher verhält, wenn man die Aktivierungen der verschiedenen Layer verwendet.

Clustering mit unbekannten Sprechern

Als erstes wird das Clustering mit den 40 beziehungsweise 20 unbekannten Sprechern durchgeführt.

	20 Sprecher		**40 Sprecher**	
Layer	**MR 100**	**MR 590**	**MR 100**	**MR 590**
Dense 1	0.100	0.100	0.3	0.125
Dense 2	0.100	**0.100**	0.325	**0.050**
Softmax	0.450	0.250	0.700	0.450

Tabelle 5.2.: MR über alle Ebenen beim Clustering mit 20 und 40 Sprechern.

In Abbildung 5.1 sieht man klare Unterschiede zwischen den Clusterings auf Basis des Softmax-Layers (L8) und der darunterliegenden Hidden-Layer (L5 und L7). Letztere liefern signifikant bessere Ergebnisse. In Tabelle 5.2 sind die Ergebnisse des Netzwerks für 590 Sprecher jenen für 100 Sprecher gegenübergestellt.

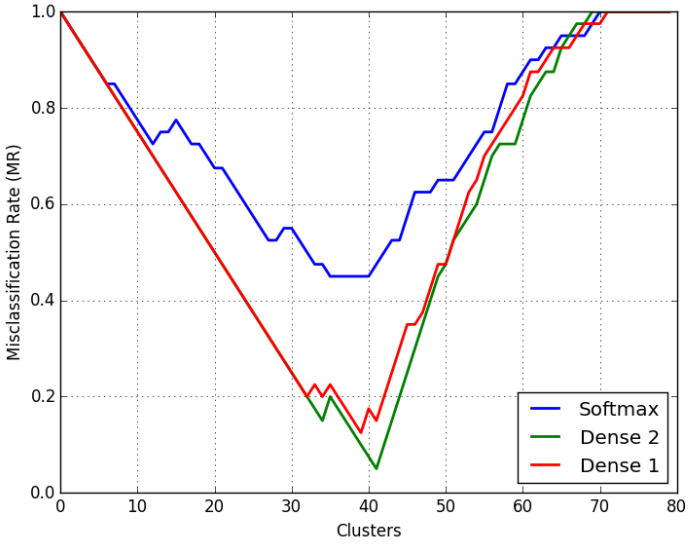

Abbildung 5.1.: MRs beim Clustering der Output-Vektoren aller Layer mit 40 unbekannten Sprechern.

In [32] wurden auf denselben 20 und 40 Sprechern MRs von 0.0 und 0.0625 erreicht. Die dort verwendete Baseline, die einen MFCC-GMM Ansatz verwendet, erreichte für 40 Sprecher eine MR von 0.125. Das zeigt, dass sich unser Ansatz auf Augenhöhe mit dem State of the Art befindet.

Clustering mit bekannten Sprechern

In dieser Phase werden 40 zufällige Sprecher aus den 590 dem Netzwerk bekannten Sprechern ausgewählt. Mit diesen führen wir das Experiment erneut durch.

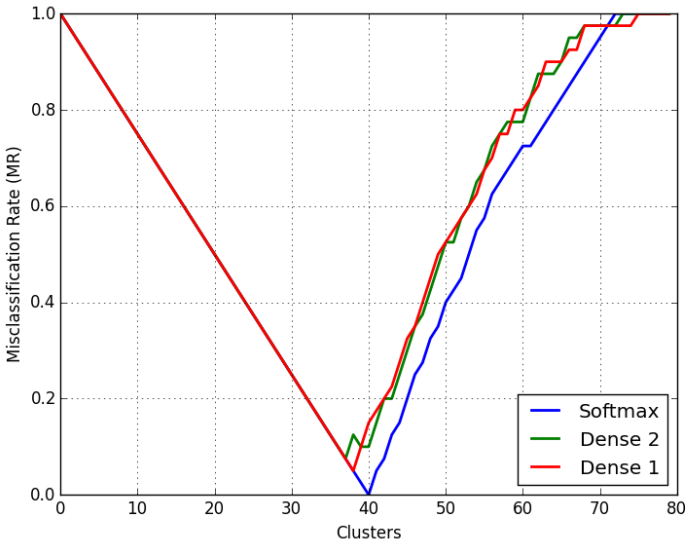

Abbildung 5.2.: MRs beim Clustering der Output-Vektoren aller Layer mit bekannten Sprechern.

Das Ergebnis auf dem Softmax-Layer (L7) fällt mit den bekannten Sprechern signifikant besser aus, so erreichten wir dort eine MR von 0.025. Interessant zu beobachten ist, dass auf den Hidden-Layern

(L5 und L7) keine massive Verbesserung stattgefunden hat. Damit wurde auf beiden lediglich eine MR von 0.1 erreicht. Das bestätigt unsere These, dass die Hidden-Layers (L5 und L7) noch stark sprecherunabhängig sind.

5.4. Fazit

Auf den Hidden-Layern (L5 und L7) konnten wir für unbekannte Sprecher eine klare Verbesserung zum Softmax-Layer (L7) feststellen. Wir begründen das damit, dass direkt nach der Convolution noch nicht oder nur wenig bezüglich den eigentlichen Sprechern gelernt wurde. Das Clustering wird nach den extrahierten Features gemacht. Diese Begründung wird bestätigt, clustert man die bekannten Sprecher. So sind diese durch den Softmax-Layer (L7) besser zu unterscheiden und liefern in den unteren Layern keine signifikant besseren Ergebnisse als auf unbekannten Sprechern.

6. Resultate KL-Divergenz

In diesem Kapitel beschreiben wir die Resultate unserer Experimente bezüglich den Netzwerken, die mit der paarweisen KL-Divergenz trainiert werden. Wir starten mit der Evaluation unserer Implementation mittels des Testaufbaus aus [24] und führen dann mehrere Experimente auf der Problemstellung des Sprecherclusterings durch. Die Experimente sind chronologisch angeordnet und zeigen so unseren Findungsprozess auf.

6.1. Architektur

Falls nicht explizit erwähnt, verwenden die Experimente in diesem Kapitel die Grundarchitektur aus Abschnitt 3.2.2. Sie wird jeweils auf die entsprechende Anzahl Sprecher angepasst.

6.2. Experiment 3: Clustering von MNIST

In [24] wird eine Architektur vorgeschlagen, die nicht die Klassen selbst lernt, sondern anhand von Distanzen zwischen den Klassen funktioniert. Die Idee ist, dass ähnliche Klassen einen möglichst ähnlichen Vektor als Output ergeben und unterschiedliche Klassen einen möglichst unterschiedlichen. Die Experimente wurden dabei auf den MNIST [33] und CIFAR-10 [39] Datensätzen durchgeführt. Für unsere Experimente bauen wir den Ansatz mit `Lasagne` nach. Dieses Experiment dient dazu die Implementation zu überprüfen.

Datensatz

Das Experiment verwendet den MNIST-Datensatz. Statt die Anzahl Sprecher wurde für n_s die Anzahl Zahlen (0-9) verwendet. Des Weiteren wurde der erwartete Input auf die Bildgrösse der MNIST-Beispiele angepasst (28×28 anstatt 128×100 Pixel).

Auswertung

Wir trainieren das Netzwerk für 10 Epochen mit einer statischen Lernrate von 0.1. Dabei verwenden wir $50'000$ Samples als Trainings- und $10'000$ als Validierungsdaten. Auf den separaten $10'000$ Samples Testdaten erreichen wir nach abgeschlossenem Training eine Accuracy von 99.27%. In [24] wird auf den selben Testdaten eine Accuracy von 99.6% erreicht, jedoch wird nicht beschrieben wie lange das Netzwerk trainiert wurde und die Validierungsdaten werden ebenfalls für das Training verwendet. Unser Ergebnis ist somit ausreichend um bestätigen zu können, dass unsere Implementation korrekt funktioniert.

6.3. Experiment 4: Clustering von Spektrogrammen

In diesem Experiment wenden wir unsere funktionierende Implementation des KL-Divergenz Netzwerks auf die eigentliche Problemstellung des Sprecherclusterings an.
Hierfür verwenden wir zwei verschiedene Netzwerke, das erste ist für 10 Sprecher und das zweite für 100 Sprecher ausgelegt.

Auswertung

Das Netzwerk für 10 Sprecher wird mit unserer Standardkonfiguration trainiert und erreichte auf den Testdaten der 10 Sprecher eine Accuracy von 93.1% für den Identifikationstask. Anhand der Losses in Abbildung 6.1 sieht man auch, dass keinerlei Overfitting auftritt.

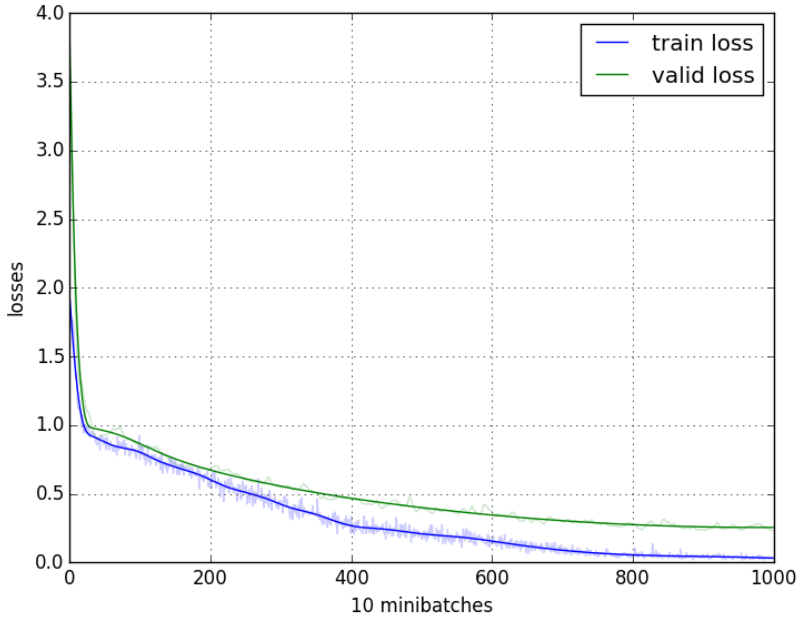

Abbildung 6.1.: Die Losses während des Trainings für $10'000$ Mini-Batches für das Netzwerk, das auf 10 Sprecher trainiert wird. Die Kurven sind per Spline-Interpolation geglättet.

Wir testen dieses kleine Netzwerk auch auf dem Clusteringtask. Hierfür verwenden wir 5 Sprecher aus dem Reynolds Datensatz, die dem Netzwerk unbekannt sind. Dabei wird beim Clustering auf dem Softmax-Layer (L11) eine MR von 0.0 erreicht. Das ist ein sehr gutes Ergebnis, da wir somit sagen können, dass das Netzwerk auch unbekannte Sprecher auseinanderhalten kann. Betrachtet man die t-SNE Plots[40] in Abbildung 6.2 sieht man, dass die Sprecher auch beim Clustering auf Segmentebene mehrheitlich auseinandergehalten werden können. Dabei scheint der erste Dense-Layer (L7) ein schöneres Clustering zu ermöglichen als der Softmax-Layer obwohl bei beiden auf unserem Clusteringtask eine MR von 0.0 erreicht wird. Auch wenn man 10 unbekannte Sprecher clustert, wird immernoch eine MR von 0.3 erreicht. Hier zeigt sich dann aber, dass der erste Dense-Layer tatsächlich bessere Ergebnisse liefert, so erreicht dieser eine MR von 0.1. Der zweite Dense-Layer (L10) liefert auf beiden Datensätzen schlechtere Ergebnisse als die anderen beiden Layer (L7 und L11).

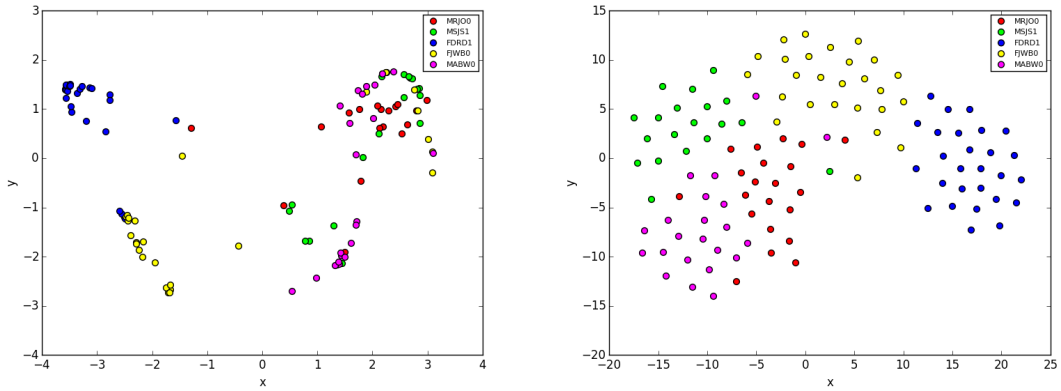

Abbildung 6.2.: t-SNE Plots auf den Output-Vektoren für den Softmax-Layer L11 (links) und den ersten Dense-Layer L7 (rechts). Verschiedene Farben stehen für verschiedene Sprecher und die Legende beinhaltet die Sprecherlabels. Weibliche Sprecher beginnen mit einem „F“ und männliche mit einem „M“.

Das Training mit 100 Sprechern ist nicht mehr ganz so einfach. Wir erreichen mit den ersten Tests lediglich eine Accuracy von 30% auf dem Identifikationstask. Dabei verwenden wir die bewährte Lernrate von 0.001[1] und trainieren das Netzwerk mit Mini-Batch Gradient Descent zusammen mit einem Nesterov Momentum. Da das Netzwerk zu Beginn zu lernen scheint und auch nach mehreren Trainingsdurchgängen bei der gleichen Accuracy hängen bleibt, nehmen wir an, dass es sich dabei um ein lokales Minimum handelt. Um dem entgegenzuwirken testen wir ebenfalls die Lernraten 0.01 und 0.1. Die erreichten Accuracies sind in Abbildung 6.3 aufgezeigt. Man sieht klar, dass auch die Lernrate von 0.01 Probleme hat über eine akzeptable Accuracy zu kommen, wobei das Netzwerk das mit einer Lernrate von 0.1 trainiert wird, auf den Testdaten eine Accuracy von rund 79.1% erreicht. In Abbildung 6.4 sieht man anhand der Losses für das Training dieses Netzwerks, dass keinerlei Overfitting auftritt. Es ist jedoch festzustellen, dass der Validierungsloss teilweise stark ausschlägt. Das ist aber nicht weiter bedenklich, da dieser, über alles betrachtet, beständig sinkt.

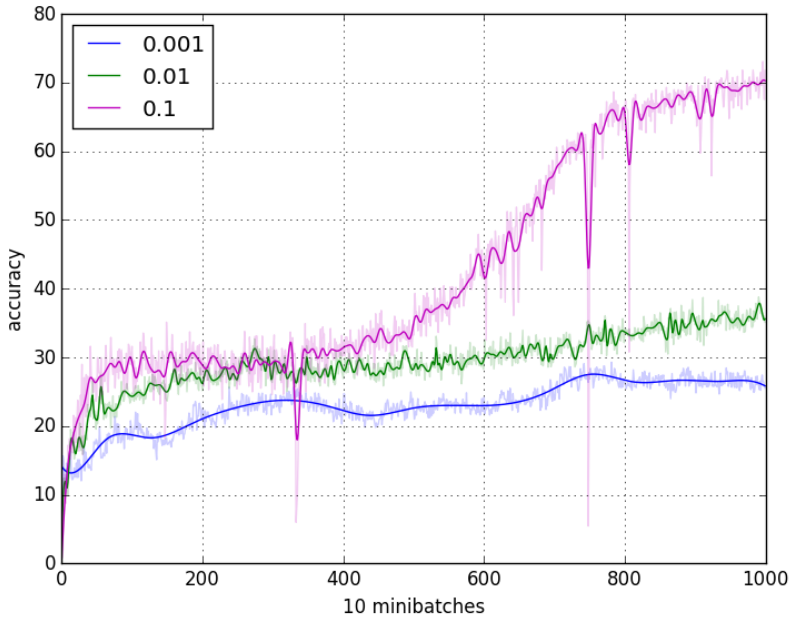

Abbildung 6.3.: Die Validierungs-Accuracies während des Trainings für verschiedene Lernraten.

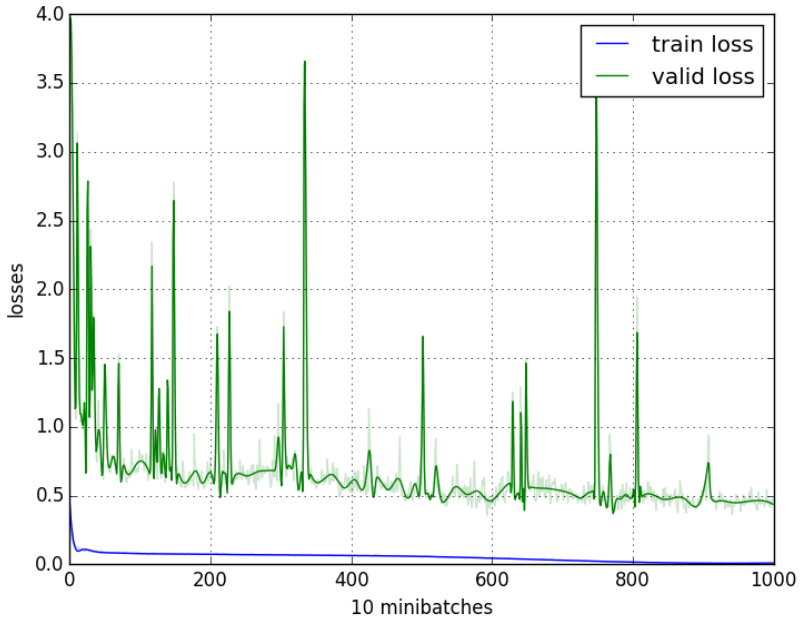

Abbildung 6.4.: Die Losses während des Trainings für $10'000$ Mini-Batches für das Netzwerk das auf 100 Sprecher trainiert wird. Dabei wird Mini-Batch Gradient Descent mit Nesterov Momentum, zusammen wir einer Lernrate von 0.1, verwendet.

Die erreichten Ergebnisse auf dem Identifikationstask sind mit jener der Identifikationsnetzwerke vergleichbar, bedingen jedoch eine extrem lange Trainingszeit. Dies kommt zum Teil von der symbolischen Implementation in `Theano` mittels der `scan`-Funktion. Diese kann nur beschränkt auf der GPU ausgeführt werden. Andernfalls könnte es gemäss [24] gut sein, dass das Training schneller ablaufen würde als bei den Identifikationsnetzwerken. Jedoch müsste dies mit einer anderen Bibliothek für neuronale Netze überprüft werden, da eine effiziente Implementation mittels `Theano` unmöglich scheint.

Auf dem Clusteringtask scheinen die Ergebnisse hinter jenen des Identifikationsnetzwerks abzufallen, auch wenn auf den mittleren Layern ebenfalls bessere MRs erreicht werden. Was auffällt ist, dass auf dem Softmax-Layer (L11) eine bessere MR erreicht wird, als mit den Identifikationsnetzwerken. In den folgenden Experimenten testen wir einige Heuristiken um auch auf dem Clusteringtask bessere Ergebnisse zu erzielen.

6.4. Experiment 5: Statische Convolution

Das verwendete Netzwerk lernt nur sehr langsam. Es ist auch nur schwer festzustellen ob es nach $10'000$ Mini-Batches überhaupt noch Fortschritte macht. Die Ergebnisse auf dem Clusteringtask sind nicht zufriedenstellend. Des Weiteren gehen wir für dieses Experiment davon aus, dass der Softmax-Layer (L11) eine Verbesserung gegenüber den Hidden-Layern auf dem Clusteringtask erreichen kann, wenn das Netzwerk gut trainiert ist. Dieser These wiedersprechen die Ergebnisse aus dem vorigen Experiment, wo auf dem ersten Dense-Layer (L7) leicht bessere Ergebnisse als auf dem Softmax-Layer erreicht wurden.
Um ein schnelleres Training zu ermöglichen, verwenden wir für dieses Experiment die vortrainierte Convolution eines Identifikationsnetzwerks. Diese ersetzt die Convolution des KL-Divergenz Netzwerks und führt während des Trainings keine Parameterupdates durch, wird also nicht mittrainiert.

Architektur

Die Grundarchitektur aus Abschnitt 3.2.2 wird für alle Layer nach dem letzten Max-Pooling-Layer (L6) übernommen. Lediglich die Convolution (L1 bis L6) wird durch ein vortrainiertes Netzwerk aus Abschnitt 3.2.1 ersetzt.
Die beiden Netzwerkteile werden jeweils mit unterschiedlichen 100 Sprechern trainiert die nicht im Reynolds Datensatz vorkommen. Der Grund hierfür ist, dass wir erreichen wollen, dass das Netzwerk auf Basis von Featurevektoren von beliebigen Sprechern clustern kann. Dies könnte verfälscht werden, wenn die Convolution die Sprecher bereits kennt.

Auswertung

Die Trainingszeit ist signifikant kürzer, da die Convolution nicht mehr trainiert werden muss. Mit integrierter Convolution dauert das Training von 10 Mini-Batches rund 85 Sekunden, mit statischer Convolution wird die Trainingszeit auf 50 Sekunden reduziert. Das entspricht einer Beschleunigung von rund 30%. Jedoch bleibt der erwartete Erfolg aus. Die Accuracy auf dem Identifikationstask erreicht lediglich 50% und auf dem Clusteringtask von 40 Sprechern wird auf dem Softmax-Layer (L11) eine MR von 0.425 erreicht, was circa jener vom Identifikationsnetzwerk entspricht (0.450). Wir führen das darauf zurück, dass das Netzwerk mit paarweiser KL-Divergenz im Endeffekt auch nur eine Klassifizierung lernt, aber in der Convolution explizit Features extrahiert die für die Unterscheidung zwischen zwei Sprechern notwendig sind. Ersetzen wir jetzt die Convolution mit jener eines trainierten Identifikationsnetzwerks werden die Hidden-Layers (L7 und L10) auf Basis dieser Features trainiert. Was heisst, dass diese im Endeffekt sehr ähnliche Parameter haben müssen wie beim Identifikationsnetzwerk um eine gute Sprecheridentifikation zu erreichen. Das resultierende Netzwerk ist also praktisch identisch mit dem Identifikationsnetzwerk. Wir schliessen daraus, dass das mittrainieren der Convolution essenziell ist und verwerfen diese Heuristik.

6.5. Experiment 6: Mehr gleiche Sprecher

Die Verwendung einer statischen Convolution bringt nicht die gewünschten Verbesserungen. Deshalb beschliessen wir grundlegende Eingriffe in den Aufbau der Mini-Batches zu machen. Aktuell werden diese aus komplett zufälligen Segmenten von zufälligen Sprechern zusammengesetzt. Da wir für jedes mögliche Segmentpaar dann die Loss-Funktion berechnen, resultiert das in signifikant mehr ungleichen Paaren als gleichen Paaren. Siamesische Netzwerke werden zum Beispiel immer abwechselnd mit einem gleichen und dann einem ungleichen Paar trainiert. Jedoch unterscheidet sich unser Ansatz explizit durch das Bilden von allen Paaren eines Mini-Batches von siamesischen Netzwerken. Wir erhöhen deshalb für dieses Experiment die Anzahl gleicher Sprecher pro Mini-Batch, das Verhältnis von gleichen zu ungleichen Paaren wird so reduziert.
Wir verwenden eine Mini-Batchgrösse von 100 und bilden dann alle möglichen Paare (auch jene mit sich selbst). Das ergibt $100 \times 100 = 10'000$ Paare pro Mini-Batch. Gemittelte Tests über $10'000$ Iterationen ergaben, dass durchschnittlich 198 gleiche und 9802 ungleiche Paare gebildet werden. Von den gleichen Paaren muss man aber noch die identischen Paare abziehen was 98 gleiche Paare ergibt. Somit haben wir ein Verhältnis von rund $1:100$ von gleichen zu unterschiedlichen Paaren. Für dieses Experiment erhöhen wir das Verhältnis auf $1:9$. Wir nehmen an, dass das Netzwerk so nicht mehr zu extrem in eine Richtung trainiert wird und schneller lernen sollte.

Auswertung

Auch diese Heuristik ergibt deutlich schlechtere Ergebnisse. So erreicht das trainierte Netzwerk lediglich eine Accuracy von 55% auf dem Identifikationstask und eine MR von 0.90 auf dem Softmax-Layer (L11) für 40 Sprecher. Das ist für beide Problemstellungen eine klare Verschlechterung. Wir gehen davon aus, dass die ungleichen Paare somit wichtiger für das Training sind, als die gleichen.

6.6. Experiment 7: Optimierungsverfahren

Das Verwenden einer statischen Convolution und die Erhöhung der gleichen Sprecher pro Mini-Batch ergeben nicht die gewünschte Verbesserung. Wir nehmen an, dass das langsame Lernen mit der Lernrate zusammenhängen könnte. Dass das Netzwerk in lokale Minimas fällt und dort nicht mehr rauskommt. Bislang verwendeten wir immer SGD mit Nesterov Momentum. Das könnte zur Folge haben, dass wenn die Lernrate zu gross ist, diese nicht mehr konvergieren kann und wenn sie zu klein ist in lokale Minimas fällt und dort nur noch langsam rauskommt. Um dem entgegenzuwirken evaluieren wir Trainings mit dem *Adam*- und *Adadelta*-Algorithmus[18] [19]. Diese Verfahren passen die Lernrate dynamisch an. Zudem evaluieren wir ein Training mit einem reinen SGD (ohne Momentum) um festzustellen, ob bereits das Nesterov Momentum einen positiven Einfluss auf das Training hat.

Auswertung

Wir trainieren das Netzwerk viermal mit unterschiedlichen Verfahren. Die erreichten Accuracies während des Trainings auf dem Identifikationstask sind in Abbildung 6.5 abgebildet.

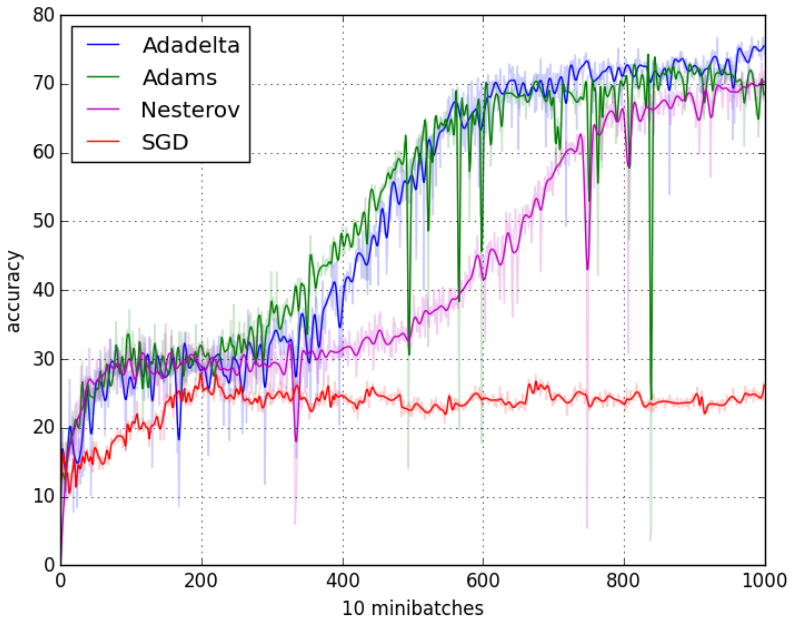

Abbildung 6.5.: Accuracies auf dem Identifikationtask von den Netzwerken die mit verschiedenen Trainingsverfahren trainiert werden.

Es ist ersichtlich, dass das Netzwerk, das mit einem reinen SGD trainiert wird, klar schlechtere Accuracies erreicht. Die anderen drei Verfahren erreichen ähnliche Accuracies, wobei jenes Netzwerk mit dem textitAdam-Verfahren gegen Ende einzubrechen scheint. Es fällt auch auf, dass das textitAdam-Verfahren zwar zu Beginn sehr schnell lernt, dann aber oft Einbrüche hat. Das *Adadelta*-Verfahren lernt zu Beginn zwar langsamer, aber durchwegs konstant ohne grosse Einbrüche. Als nächstes evaluieren wir die Performance der trainierten Netzwerke auf dem Clusteringtask.

In Tabelle 6.1 sind die erreichten MRs für den Clusteringtask beschrieben. Dabei werden, wie bereits beim Identifikationsnetzwerk, auch die Hidden-Layer (L7 und L10) ausgewertet. Es wird ersichtlich, dass auch bei dieser Architektur deutlich bessere Ergebnisse erzielt werden, wenn das Clustering auf den Aktivierungen der Hidden-Layer (L7 und L10) stattfindet. Das widerlegt unsere anfängliche These, dass die besten Ergebnisse auf dem Softmax-Layer (L11) zu beobachten sind. Man sieht auch, dass

die Performance auf dem Identifikationstask zu korrelieren scheint. So schneidet auch hier das mit *Adam* trainierte Netzwerk mit einer MR von 0.075 am besten ab. Dieser Wert ist vergleichbar mit den Ergebnissen aus [32], wo mit den gleichen 40 Sprechern eine MR von 0.0625 erreicht wurde.

	SGD	**Nesterov**	**Adadelta**	**Adam**
Dense 1	0.800	0.600	0.100	0.075
Dense 2	0.775	0.375	0.200	0.175
Softmax	0.800	0.600	0.500	0.550

Tabelle 6.1.: MR für alle Trainingsverfahren über alle Ebenen beim Clustering mit 40 Sprechern.

Die besten Ergebnisse bezüglich Accuracy und MR werden durch die Verfahren *Adadelta* und *Adam* erreicht. Sie sind vergleichbar mit den Ergebnissen mit dem Identifikationsnetzwerk und den Ergebnissen aus [32]. Um sicherzustellen, dass das volle Potential ausgeschöpft wird, trainieren wir diese beiden Netzwerke noch um $5'000$ Mini-Batches weiter und involvieren dabei die Validierungsdaten. Betrachtet man die Accuracies in Abbildung 6.1 sieht man, dass das Adam Verfahren zwar schneller lernt, aber nicht so konstant wie das *Adadelta*-Verfahren. Wir nehmen deshalb an, dass *Adadelta* über lange Sicht bessere Ergebnisse erzielen wird und deshalb mehr von diesen zusätzlichen Mini-Batches profitieren wird. Das wird auch deutlich, wenn man die erreichten MRs in Tabelle 6.2 betrachtet, die nach diesem zusätzlichen Training resultieren. So hat sich das Netzwerk mit dem *Adadelta*-Verfahren auf allen Layern nochmals deutlich gesteigert. Das Netzwerk mit dem *Adam*-Verfahren erreicht die exakt gleichen Ergebnisse wie vor dem zusätzlichen Training.

	Adadelta	**Adam**
Dense 1	0.075	0.075
Dense 2	0.075	0.175
Softmax	0.375	0.550

Tabelle 6.2.: MR für die zusätzlich trainierten Netzwerke mit dem Adam- und Adadelta-Verfahren, über alle Ebenen beim Clustering mit 40 Sprechern.

Da das Netzwerk mit dem textitAdadelta-Verfahren noch zu lernen scheint, trainieren wir dieses nochmals für $15'000$ Mini-Batches. Die erreichten MRs über das gesamte Training sind jeweils in Schritten von $5'000$ Mini-Batches in Abbildung 6.6 abgebildet. Man sieht, dass auf dem ersten Dense-Layer (L7) nach $20'000$ Mini-Batches keine Verbesserung mehr stattfindet. Jedoch wird dann eine sehr gute MR von 0.05 auf den 40 unbekannten Sprechern des Reynolds Datensatzes erreicht.

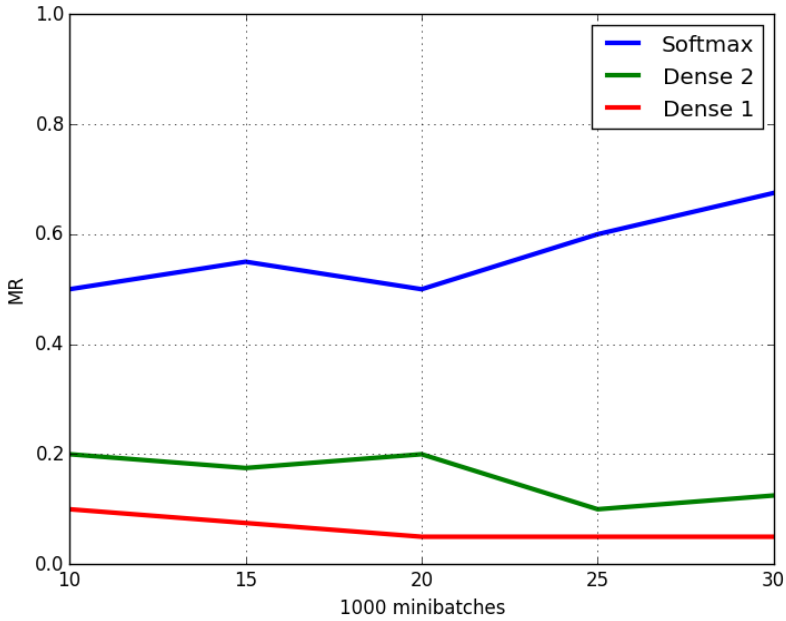

Abbildung 6.6.: MRs für verschiedene Trainingsstufen des Netzwerks das mittels Adadelta-Verfahren trainiert wird.

In Abbildung 6.7 ist das hierarchische Clustering der 40 Sprecher mittels Dendrogramm abgebildet. Man sieht dabei, dass lediglich ein Featurevektor Paar nicht korrekt zugewiesen wird. Jedoch wird der optimale Schnittpunkt (tiefste MR) vorher angesetzt, somit werden zwei Featurevektor Paare nicht gefunden.

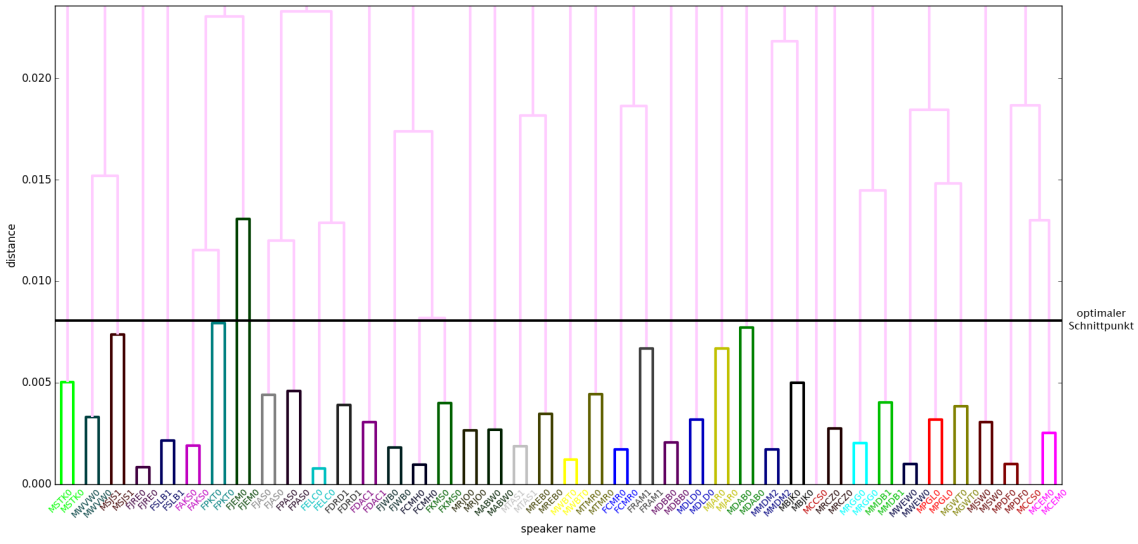

Abbildung 6.7.: Dendrogramm das das hierarchische Clustering für das für $25'000$ Minibatches trainierte *Adadelta*-Netzwerk darstellt. Korrekt zusammengefasste Featurevektor Paare sind entsprechend der Sprecherlabels eingefärbt. Labels die mit einem „F“ beginnen bezeichnen einen weiblichen Sprecher, jene mit einem „M“ einen männlichen.

Das *Adadelta*-Verfahren ergibt in diesem Experiment die besten Ergebnisse. Es ist aber gut möglich, dass auch mit dem *Adam*-Verfahren noch bessere Ergebnisse erreicht würden, wenn noch mehr Hyperparameter-Tuning durchgeführt würde.
Mit dem besten Netzwerk erreichen wir auf den 20 und 40 unbekannten Sprechern MRs von 0.0 respektive 0.05. Dieses Ergebnis unterbietet jene aus [32] wo MRs von 0.0 und 0.0625 erreicht wurden.

6.7. Experiment 8: Training mit 590 Sprechern

Unser Netzwerk mit paarweiser KL-Divergenz lernt Distanzen zwischen Sprechern. Somit sollte es theoretisch möglich sein ein Netzwerk mit mehr Sprechern zu trainieren als es Ouput-Neuronen hat. Jedoch stellten wir in den vorigen Experimenten fest, dass das Netzwerk automatisch Sprecher einem bestimmten Neuron zuzuweisen beginnt. Dies stimmt auch mit den Erkenntnissen aus [24] überein, wo das gleiche Verhalten beobachtet wurde. In diesem Experiment wollten wir herausfinden, wie sich das Netzwerk verhält, wenn es mit mehr Sprechern trainiert wird, als es Ouput-Neuronen hat.
Das Experiment wurde parallel mit der Evaluation der Optimierungsverfahren durchgeführt. Zu diesem Zeitpunkt erreichte das *Adam*-Verfahren noch die besten Ergebnisse, weshalb hier ebenfalls dieses verwendet wird. Zukünftige Arbeiten sollten dieses Experiment erneut mit dem *Adadelta*-Verfahren durchführen.Wir trainieren das Netzwerk mit allen 590 Sprechern, die nicht im Reynolds Datensatz enthalten sind. Da das Testen der Validierungsdaten auf dem Identifikationstask auf einem Netzwerk mit 100 Output-Neuronen nicht möglich und die Berechnung des Losses zu performanceintensiv ist, verzichten wir darauf und verwenden die Validierungsdaten ebenfalls für das Training.

Auswertung

Nach $10'000$ Mini-Batches erreicht das Netzwerk auf dem Clusteringtask für die 40 unbekannten Sprecher auf dem ersten Dense-Layer (L7) eine MR von 0.450 auf dem zweiten (L10) 0.750 und auf dem Softmax-Layer (L11) 0.825. Das sind signifikant schlechtere Resultate als in den vorherigen Experimenten. Betrachtet man den Verlauf des Losses auf den Trainingsdaten in Abbildung 6.8 sieht man, dass dieser in den ersten 1000 Mini-Batches zwar schnell sinkt, dann jedoch nur noch langsam gegen 0 konvergiert. Ein Lernfortschritt ist bis zum Schluss zu beobachten. Wir nehmen an, dass wenn man das Netzwerk noch länger trainieren würde, der Loss noch weiter sinken und somit die Clusteringperformance verbessern würde.

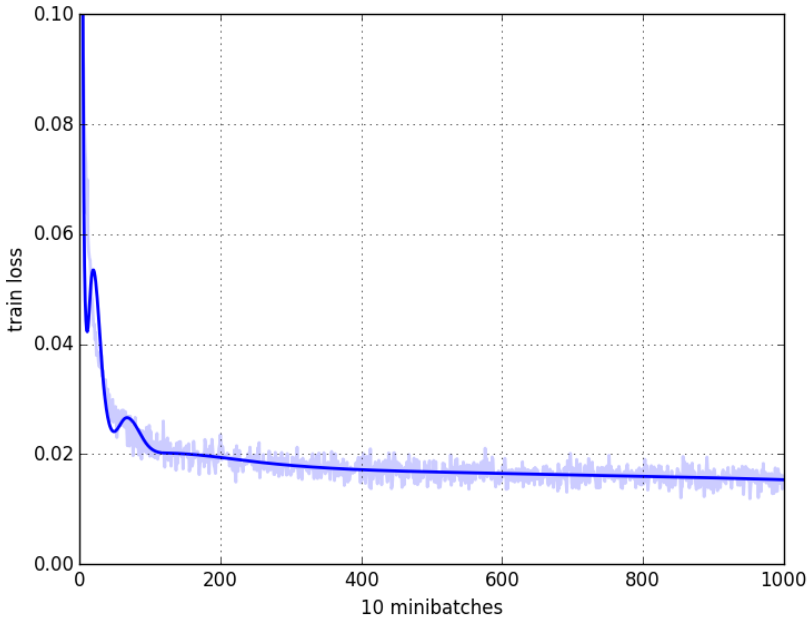

Abbildung 6.8.: Der Loss über $10'000$ Mini-Batches für das 100er Netzwerk, das mittels 590 Sprechern trainiert wird.

Weitere Experimente in diesem Bereich sind im Rahmen dieser Arbeit nicht mehr möglich und sollten in Zukunft weiterverfolgt werden. Dabei wäre das Experimentieren in einem kleinere Rahmen zu empfehlen (z.B. ein 10er Netzwerk das mittels 20 Sprechern trainiert wird).

6.8. Fazit

Mit unseren Experimenten verifizierten wir unsere Implementation der Konzepte aus [24]. So erreichen wir auf dem MNIST Datensatz mit der Referenzimplementation vergleichbare Ergebnisse.

Wir konnten zeigen, dass Netzwerke die mittels paarweiser KL-Divergenz trainiert werden, auch komplexere Problemstellungen als den MNIST Datensatz lösen können, konkret in unserem Fall Sprecheridentifikation und Sprecherclustering. Das Training gestaltet sich jedoch schwieriger. Ohne zusätzliche Optimierungsverfahren wie das Adam- oder Adadelta-Verfahren erreichen wir keine vergleichbaren Ergebnisse. Des Weiteren führte die Erhöhung der Anzahl gleicher Sprecher in einem Mini-Batch zu schlechteren Lernergebnissen. Wir begründen dies damit, dass das Zusammenziehen von gleichen Paaren weniger wichtig ist, als das Auseinanderhalten von unterschiedlichen.

Auf der Sprecheridentifikation erreichen wir vergleichbare Accuracies wie mit klassischen Identifikationsnetzwerken. Auf dem Clusteringtask erreichen wir mit lediglich 100 Sprechern Ergebnisse die mittels Identifikationsnetzwerken 590 Sprecher benötigen.

Wie bereits bei den Identifikationsnetzwerken werden bessere Clusteringergebnisse erzielt, wenn auf den mittleren Layern (L7 und L10) geclustert wird. Wir erreichen für 20 und 40 unbekannte Sprecher MRs von 0.0 respektive 0.05. Wir unterbieten damit die Ergebnisse aus [32] wo die gleichen Sprecher verwendet und MRs von 0.0 und 0.0625 erreicht wurden.

Für Netzwerke mit 100 Output-Neuronen benötigen das Training von $10'000$ Mini-Batches circa 24 Stunden. In [32] sind für das Clusteringexperiment circa 10 Stunden nötig. Jedoch muss dieses Training für andere 40 Sprecher erneut durchgeführt werden. Unser Modell hat den Vorteil, dass es für beliebige Sprecher eingesetzt werden kann.
Um die Featurevektoren für unser Clusteringexperiment von 40 Sprechern zu generieren, benötigt unser Netzwerk lediglich 15 Sekunden (das Kompilieren vom symbolischen Graphen mittels `Theano` nicht eingerechnet, da dies nur einmalig nötig ist).

7. Schlussfolgerungen

Dieses Kapitel beinhaltet ein zusammenfassendes Fazit über die erreichten Ergebnisse und beschreibt einige Ideen für zukünftige Arbeiten in diesem Gebiet.

7.1. Fazit

Das Ziel dieser Arbeit war es die Ansätze aus [1] weiterzuentwickeln und den neuen Ansatz der paarweisen KL-Divergenz[24] zu implementieren und zu evaluieren.

Wir konnten dabei erfolgreich unsere Fehler aus [1] beheben und erweiterten den damaligen Ansatz um das Konzept der Sprecher-Embeddings[25]. Dabei erreichten wir mit einem für 590 Sprecher trainierten Identifikationsnetzwerk MRs von 0.1 und 0.05 für 20 respektive 40 unbekannte Sprecher. Der State of the Art erreicht auf den gleichen unbekannten Sprechern MRs von 0.0 respektive 0.0625.

Im Rahmen dieser Arbeit implementierten wir erfolgreich den Trainingsansatz der paarweisen KL-Divergenz. Wir stiessen dabei jedoch an die Grenzen der verwendeten Python-Bibliotheken `Lasagne` und `Theano`. So war es nicht möglich, diese Trainings performant auf der GPU auszuführen, da keine ausreichende Unterstützung für die Implementierung von Loops besteht.
Die mittels paarweiser KL-Divergenz trainierten Netzwerke erreichten MRs von 0.0 und 0.05 für 20 respektive 40 unbekannte Sprecher und unterbieten alle bisherigen Ansätze. Ein grosser Vorteil gegenüber dem Ansatz mit einem Identifikationsnetzwerk ist, dass für diese Ergebnisse lediglich 100 Sprecher benötigt werden. Somit sind 83% weniger Trainingsdaten nötig um vergleichbare oder bessere Ergebnisse zu erreichen.
Der Ansatz scheint aber noch immer Verbesserungspotential aufzuweisen. So erreichten wir mit einem anders konfigurierten Netzwerk einmalig auf den 40 unbekannten Sprechern eine MR von 0.025 (Filtergrösse: 16×1, Pooling: 4×4, Stride: 2×2). Bei weiteren Versuchen mit gleicher Konfiguration erreichten wird diesen Wert nicht mehr.

Unsere Modelle haben neben den gleich guten oder sogar besseren Ergebnisse als in [32] den Vorteil, dass sie nach einmaligem Training für beliebige Sprecher eingesetzt werden können. Der Ansatz aus [32] muss für jedes Sprecherset neu trainiert werden.

Die Arbeit konnte alle Ziele erreichen, jedoch blieben auch einige Fragen offen. Auf diese werden wir im nächsten Abschnitt 7.2 eingehen.

7.2. Ausblick

In diesem Abschnitt wird aufgezeigt, wie an diese Arbeit angeknüpft werden kann.

7.2.1. Clustering mit Deep Learning Verfahren

Im Folgenden beschreiben wir grundsätzliche Überlegungen bezüglich dem Sprecher Clustering mittels Deep Learning Verfahren.

Weitere Filterexperimente

Es sind weitere Experimente zu den Filtern nötig, um festzustellen, weshalb die meisten Filtergrössen, Pooling und Stride Kombinationen vergleichbare Ergebnisse liefern, während ein Zeitrichtungsfilter mit Pooling und Stride in beide Richtungen schlecht ist. Des Weiteren sollte evaluiert werden ob ein Filter in Frequenzrichtung Zeitzusammenhänge ignoriert. Das kann zum Beispiel erreicht werden indem man die Spalten des Spektrogramms zufällig permutiert. Kann das Netzwerk dann noch immer Sprecher zuweisen, ist der Zeitkontext für diesen Filter irrelevant.

Einsatz eines klaren Embedding-Layers

Unsere eingesetzte Architektur aus Abschnitt 3.2.2 ist eine Kombination aus unserem Identifikationsnetzwerk aus Abschnitt 3.2.1 und jenem aus [24]. Die besten Ergebnisse für das Clustering erreichten wir bei beiden Ansätzen (Identifikationsnetzwerk und paarweise KL-Divergenz) auf den Hidden-Layern mittels *Speaker Embeddings* [25]. Wir verwendeten dabei jedoch keinen expliziten Embedding-Layer, dieser wäre kleiner als die beiden umschliessenden Layer. Man könnte hier die Architektur soweit anpassen, dass ein Embedding-Layer hinzugefügt wird und die Clusteringperformance auf diesem evaluieren.

7.2.2. Netzwerk mit paarweiser KL-Divergenz

Dieser Abschnitt beschreibt weiterführende Ideen betreffend dem Ansatz der paarweisen KL-Divergenz.

Untersuchung des *margin*-Parameters

Die Loss-Funktion enthält neben der KL-Divergenz die Konstante *margin*, die stark dazu beiträgt wie weit ungleiche Paare voneinander entfernt sein sollten. Wir setzen den *margin*-Parameter stets gleich 2. Dieser Wert wurde in [24] empirisch für den MNIST Datensatz gefunden. Für das Einsatzgebiet von Spektrogrammen könnten in weiteren Experimenten andere Werte für den *margin*-Parameter evaluiert werden.

Einsatz anderer Loss-Funktionen

Zukünftige Arbeiten könnten andere Loss-Funktionen verwenden. So wird in [41] der sogenannte Triplet-Loss vorgestellt. Die Funktionsweise ist dabei sehr ähnlich zu jenem der paarweisen KL-Divergenz. Das Ziel ist verschiedene Inputs in einen euklidischen Raum abzubilden, wobei Inputs der gleichen Klasse nahe beieinander und unterschiedliche Klassen möglichst weit auseinander liegen sollen. Es werden dabei immer drei Inputs zusammen verwendet um den Loss zu berechnen, sogenannte *Triplets*. Ein Input dient dabei als *Anker*, einer als *Positiv* (gleiche Klasse wie Anker) und einer als *Negativ* (andere Klasse als Anker). Mit geschickter Auswahl dieser Triplets kann eine schnelle Konvergenz erreicht werden. Dies wurde in [41] für Gesichtsverifikation, -identifikation und -clustering getestet. Eine Adaption dieses Konzepts auf unseren Ansatz könnte interessant sein.

Weitere Untersuchungen bezüglich der Mini-Batch-Zusammensetzung

Wir führten ein Experiment durch, bei dem wir den Anteil an gleichen Sprechern in einem Mini-Batch erhöhten. In [41] wird zwar eine andere Loss-Funktion verwendet, ansonsten ist der Ansatz aber sehr ähnlich zu dem unsrigen. Dort wird darauf geachtet, welche Inputs zusammen einen positiven Lerneffekt auf das Netzwerk haben können, bevor dieses trainiert wird. So müsste ein Constraint definiert werden, der zwischen zwei Segmenten erfüllt werden muss, damit der Loss auf diesen berechnet und das Netzwerk trainiert wird. Sollte das möglich sein, könnte eine schnellere Konvergenz und somit ein schnelleres Training erreicht werden, was in der aktuellen Implementation ein Grundproblem darstellt.

Training mit mehr Sprechern als Output-Neuronen

In Abschnitt 6.7 testeten wir das Verhalten des Netzwerks, wenn dieses mit mehr Sprechern trainiert wird, als es Ouput-Vektoren hat. Wir verwendeten dabei das *Adam*-Verfahren, das sich als relativ instabil herausstellte. Zukünftige Arbeiten sollten dieses Experiment mit dem *Adadelta*-Verfahren durchführen. Des Weiteren könnte das Experiment in einem kleineren Rahmen durchgeführt werden, um einfacher Tests durchführen zu können. So wären Tests mit einem Netzwerk mit 10 Ouput-Neuronen und einem Trainingsset von 20 Sprechern, einfacher durchzuführen und die Ergebnisse besser zu interpretieren.

Verwendung von komplexeren Netzwerken

Wir trainierten lediglich Netzwerke mit maximal 100 Output-Neuronen. Mit den Identifikationsnetzwerken erreichten wir die besten Ergebnisse mit einem 590er Netzwerk. Wir gehen davon aus, dass auch mit den KL-Divergenz Netzwerken eine Verbesserung festzustellen wäre, wenn grössere Netzwerke verwendet würden. Jedoch würde die Implementation in `Lasagne` und `Theano` rund 3 Wochen benötigen um ein 590er Netzwerk für $10'000$ Mini-Batches zu trainieren. Das war im Rahmen dieser Arbeit nicht möglich.
Zukünftige Arbeiten sollten untersuchen ob auch bei den Netzwerken mit paarweiser KL-Divergenz eine Verbesserung festzustellen ist, wenn ein Netzwerk mit höherer Komplexität verwendet wird. Hierzu ist aber die Verwendung einer anderen Bibliothek zu empfehlen.

8. Verzeichnisse

Literaturverzeichnis

[1] Y. L. und Carlo Vogt, "Sprechererkennung mit Convolutional Neural Networks." 2015.

[2] D. A. Reynolds, T. F. Quatieri, and R. B. Dunn, "Speaker verification using adapted gaussian mixture models," *Digital signal processing*, vol. 10, no. 1, pp. 19–41, 2000.

[3] N. Dehak, P. Kenny, R. Dehak, P. Dumouchel, and P. Ouellet, "Front-end factor analysis for speaker verification," *Audio, Speech, and Language Processing, IEEE Transactions on*, vol. 19, no. 4, pp. 788–798, 2011.

[4] H.-S. Lee, Y. Tsao, H.-M. Wang, and S.-K. Jeng, "Clustering-based i-vector formulation for speaker recognition.," in *INTERSPEECH*, pp. 1101–1105, 2014.

[5] J. Ajmera and C. Wooters, "A robust speaker clustering algorithm," in *Automatic Speech Recognition and Understanding, 2003. ASRU'03. 2003 IEEE Workshop on*, pp. 411–416, IEEE, 2003.

[6] G. Friedland, A. Janin, D. Imseng, X. A. Miro, L. Gottlieb, M. Huijbregts, M. T. Knox, and O. Vinyals, "The icsi rt-09 speaker diarization system," *Audio, Speech, and Language Processing, IEEE Transactions on*, vol. 20, no. 2, pp. 371–381, 2012.

[7] I. Kapsouras, A. Tefas, N. Nikolaidis, G. Peeters, L. Benaroya, and I. Pitas, "Multimodal speaker clustering in full length movies," *Multimedia Tools and Applications*, pp. 1–20, 2016.

[8] T. M. Mitchell, *The discipline of machine learning*, vol. 9. Carnegie Mellon University, School of Computer Science, Machine Learning Department, 2006.

[9] K. L. Priddy and P. E. Keller, *Artificial neural networks: an introduction*, vol. 68. SPIE Press, 2005.

[10] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 1998.

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville, "Deep learning." Book in preparation for MIT Press, 2016.

[12] M. A. Nielsen, *Neural Networks and Deep Learning*. Determination Press, 2015.

[13] B. Xu, N. Wang, T. Chen, and M. Li, "Empirical evaluation of rectified activations in convolutional network," *arXiv preprint arXiv:1505.00853*, 2015.

[14] M. D. Zeiler, M. Ranzato, R. Monga, M. Mao, K. Yang, Q. V. Le, P. Nguyen, A. Senior, V. Vanhoucke, J. Dean, *et al.*, "On rectified linear units for speech processing," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pp. 3517–3521, IEEE, 2013.

[15] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," tech. rep., DTIC Document, 1985.

[16] D. M. Hawkins, "The problem of overfitting," *Journal of chemical information and computer sciences*, vol. 44, no. 1, pp. 1–12, 2004.

[17] T. Dozat, "Incorporating nesterov momentum into adam,"

[18] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[19] M. D. Zeiler, "Adadelta: an adaptive learning rate method," *arXiv preprint arXiv:1212.5701*, 2012.

[20] S. Becker and Y. Le Cun, "Improving the convergence of back-propagation learning with second order methods," in *Proceedings of the 1988 connectionist models summer school*, pp. 29–37, San Matteo, CA: Morgan Kaufmann, 1988.

[21] G. Teschl, *Ordinary differential equations and dynamical systems*, vol. 140. American Mathematical Soc., 2012.

[22] M. LukošEvičIus and H. Jaeger, "Reservoir computing approaches to recurrent neural network training," *Computer Science Review*, vol. 3, no. 3, pp. 127–149, 2009.

[23] J. Bromley, J. W. Bentz, L. Bottou, I. Guyon, Y. LeCun, C. Moore, E. Säckinger, and R. Shah, "Signature verification using a "siamese" time delay neural network," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 7, no. 04, pp. 669–688, 1993.

[24] Y.-C. Hsu and Z. Kira, "Neural network-based clustering using pairwise constraints," *arXiv preprint arXiv:1511.06321*, 2015.

[25] M. Rouvier, P.-M. Bousquet, and B. Favre, "Speaker diarization through speaker embeddings," in *Signal Processing Conference (EUSIPCO), 2015 23rd European*, pp. 2082–2086, IEEE, 2015.

[26] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 5786, pp. 504–507, 2006.

[27] H. Beigi, *Fundamentals of speaker recognition*. Springer Science & Business Media, 2011.

[28] S. Dieleman and B. Schrauwen, "End-to-end learning for music audio," in *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pp. 6964–6968, IEEE, 2014.

[29] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[30] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[31] M. Muller, D. P. Ellis, A. Klapuri, and G. Richard, "Signal processing for music analysis," *Selected Topics in Signal Processing, IEEE Journal of*, vol. 5, no. 6, pp. 1088–1110, 2011.

[32] T. Stadelmann and B. Freisleben, "Unfolding speaker clustering potential: a biomimetic approach," in *Proceedings of the 17th ACM international conference on Multimedia*, pp. 185–194, ACM, 2009.

[33] Y. LeCun, C. Cortes, and C. J. Burges, "The mnist database of handwritten digits," 1998.

[34] A. Solomonoff, A. Mielke, M. Schmidt, and H. Gish, "Clustering speakers by their voices," in *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, vol. 2, pp. 757–760, IEEE, 1998.

[35] H. Ghaemmaghami, D. Dean, R. Vogt, and S. Sridharan, "Speaker attribution of multiple telephone conversations using a complete-linkage clustering approach," in *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, pp. 4185–4188, IEEE, 2012.

[36] C. E. Metz, "Basic principles of roc analysis," in *Seminars in nuclear medicine*, vol. 8, pp. 283–298, Elsevier, 1978.

[37] M. Kotti, V. Moschou, and C. Kotropoulos, "Speaker segmentation and clustering," *Signal processing*, vol. 88, no. 5, pp. 1091–1124, 2008.

[38] J. Munkres, "Algorithms for the assignment and transportation problems," *Journal of the Society for Industrial and Applied Mathematics*, vol. 5, no. 1, pp. 32–38, 1957.

[39] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," 2009.

[40] L. Van der Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of Machine Learning Research*, vol. 9, no. 2579-2605, p. 85, 2008.

[41] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 815–823, 2015.

Abbildungsverzeichnis

2.1. Supervised Learning. . . . 14
2.2. Unsupervised Learning. . . . 15
2.3. Reinforcement Learning. . . . 15
2.4. Perzeptron mit drei Inputs. . . . 16
2.5. Sigmoid Funktion. . . . 17
2.6. Rectified Linear Unit (ReLU) Aktivierungsfunktion [14]. . . . 17
2.7. Layer in einem Feedforward neuronalen Netzwerk. . . . 18
2.8. Neuronales Netzwerk ohne Hidden-Layer. . . . 18
2.9. Neuronales Netzwerk mit zwei Hidden-Layer. . . . 19
2.10. Convolution mit einem $4x4$ Filter. . . . 23
2.11. Convolution an einem Beispielbild aus der Steinberggasse in Winterthur. . . . 23
2.12. Aufbau CNN. . . . 24
2.13. Vergleich Feedforward neuronalen Netzwerk mit Recurrent neuronalen Netzwerk. . . . 24
2.14. Siamesische Architektur. . . . 25
2.15. Netzwerk mit einem Embedding-Layer. . . . 26
2.16. Beispiel eines Spektrogramms. . . . 26

3.1. Basisarchitektur für alle Identifikationsnetzwerke. . . . 28
3.2. Basisarchitektur für alle Netzwerke, die die paarweise KL-Divergenz als Loss-Funktion verwenden. . . . 29

4.1. Der Prozess der Datenaufbereitung. . . . 33
4.2. Der grobe Trainingsablauf eines Netzwerks. . . . 34
4.3. Die implementierte Logik für die Berechnung des Losses eines Mini-Batches. . . . 36

5.1. MRs beim Clustering der Output-Vektoren aller Layer mit 40 unbekannten Sprechern. 42
5.2. MRs beim Clustering der Output-Vektoren aller Layer mit bekannten Sprechern. . . . 42

6.1. Die Losses während des Trainings für $10'000$ Mini-Batches für das Netzwerk, das auf 10 Sprecher trainiert wird. Die Kurven sind per Spline-Interpolation geglättet. . . . 45
6.2. t-SNE Plots auf den Output-Vektoren für den Softmax-Layer L11 (links) und den ersten Dense-Layer L7 (rechts). Verschiedene Farben stehen für verschiedene Sprecher und die Legende beinhaltet die Sprecherlabels. Weibliche Sprecher beginnen mit einem „F“und männliche mit einem „M“. . . . 46
6.3. Die Validierungs-Accuracies während des Trainings für verschiedene Lernraten. . . . 46
6.4. Die Losses während des Trainings für $10'000$ Mini-Batches für das Netzwerk das auf 100 Sprecher trainiert wird. Dabei wird Mini-Batch Gradient Descent mit Nesterov Momentum, zusammen wir einer Lernrate von 0.1, verwendet. . . . 47
6.5. Accuracies auf dem Identifikationtask von den Netzwerken die mit verschiedenen Trainingsverfahren trainiert werden. . . . 49
6.6. MRs für verschiedene Trainingsstufen des Netzwerks das mittels Adadelta-Verfahren trainiert wird. . . . 51
6.7. Dendrogramm das das hierarchische Clustering für das für $25'000$ Minibatches trainierte *Adadelta*-Netzwerk darstellt. Korrekt zusammengefasste Featurevektor Paare sind entsprechend der Sprecherlabels eingefärbt. Labels die mit einem „F“beginnen bezeichnen einen weiblichen Sprecher, jene mit einem „M“einen männlichen. . . . 51
6.8. Der Loss über $10'000$ Mini-Batches für das 100er Netzwerk, das mittels 590 Sprechern trainiert wird. . . . 53

Tabellenverzeichnis

4.1. Eine mögliche Zuordnung bei drei Sprechern mit je 10 Samples. 37

5.1. Die Accuracies für die Konfigurationen im minimalistischen Experiment. Filter, Pooling und Stride sind mittels der Dimension Frequenz×Zeit ($F \times T$) angegeben. 40
5.2. MR über alle Ebenen beim Clustering mit 20 und 40 Sprechern. 41

6.1. MR für alle Trainingsverfahren über alle Ebenen beim Clustering mit 40 Sprechern. . . 50
6.2. MR für die zusätzlich trainierten Netzwerke mit dem Adam- und Adadelta-Verfahren, über alle Ebenen beim Clustering mit 40 Sprechern. 50

Listings

4.1. Unsere Implementation zweier verschachtelter Loops mittels *scan*-Funktion 36

A. Anhang

A.1. Projektmanagement

A.1.1. Offizielle Aufgabenstellung

Zürcher Hochschule
für Angewandte Wissenschaften

School of Engineering

Themengebiet: Automatische Stimmerkennung mit Deep Learning (Neuronalen Netzen)
BA16_stdm_1

BetreuerInnen:	Thilo Stadelmann, stdm Oliver Dürr, dueo
Fachgebiete:	Datenanalyse (DA) Digitale Signalverarbeitung (DSV) Software (SOW)
Studiengang:	IT / WI
Zuordnung:	Institut für angewandte Informationstechnologie (InIT)
Interne Partner:	Institut für Datenanalyse und Prozessdesign (IDP)
Gruppengrösse:	2

Kurzbeschreibung:
Automatische Stimmerkennung ist eine wichtige Basistechnologie in verschiedenen kommerziellen Bereichen (z.B. biometrische Authentifizierung, Gesprächsanalyse für Teledienste, Medienmonitoring). Trotzdem liefern automatische Verfahren deutlich schlechtere Ergebnisse als menschliches Hören. Forschung der Betreuer hat kürzlich einen Weg gezeigt, diese Lücke zu schliessen:

Mittels Verfahren des maschinellen Lernens, wie sie in der Analyse von Bildern zum Einsatz kommen, lässt sich der zeitliche Verlauf von Sprache (der "Klang") extrahieren und in einem für die Stimme charakteristischen Modell abspeichern. Insbesondere Deep Convolutional oder Recurrent Neural Networks (CNNs / RNNs) bieten sich hier an, die in den letzten Jahren zu einem Milliardengeschäft und heissesten Eisen in der Machine Learning Forschung geworden sind.

In einer vorangegangenen Projektarbeit aller Beteiligter wurde bereits eine experimentelle Umgebung für Stimmerkennung mit CNNs implementiert, in der bekannte Datensätze geladen, analysiert und Ergebnisse ausgewertet werden können. Ziel dieser Bachelorarbeit ist es nun, die implementierte "Speaker Identification" in Richtung "Speaker Clustering" weiterzuentwickeln.

Inhalt:

- Kennenlernen des Hintergrunds von Speaker Clustering in Abgrenzung zu Speaker Identification
- Einarbeiten in die relevante Literatur
- Implementieren eines geeigneten Ansatzes (Auswahl zusammen mit Betreuern, z.B. CNN oder RNN)
- Geeignetes Wählen von Einstellungen anhand der Leistung auf gegebenen Testdaten
- Evaluieren anhand systematischer Experimente und Vergleich mit existierendem Baseline Ansatz
- Präsentation der Ergebnisse (Bericht, Demo)

Voraussetzungen:
Die Betreuer haben langjährige Erfahrung mit automatischer Sprechererkennung und stehen mit Rat, Tat und viel Einsatz zu Seite. Aus den Vorgängeraufgabenstellungen sind bereits mehrere hervorragende studentische Arbeiten hervorgegangen. Bei Interesse ist eine Fortsetzung im Masterstudium möglich.

Vorkenntnisse in Sprechererkennung, Maschine Learning oder Deep Neural Networks etc. sind nicht Voraussetzung. Das nötige Anwenderwissen wird im Rahmen dieser PA vermittelt und erarbeitet. Wir erwarten lediglich Freude am Experimentieren und Programmieren.

Die Arbeit ist vereinbart mit:
Yanick Lukic (lukicyan)
Carlo Vogt (vogtcar1)

Weiterführende Informationen:
https://dublin.zhaw.ch/~stdm/?page_id=77

Donnerstag 2. Juni 2016 10:49

A.2. Weiteres

A.2.1. Beschreibung der elektronischen Daten

Dieser Arbeit liegen einige elektronische Daten bei. Im Folgenden wird deren Struktur kurz erklärt.

- **0_Dokumentation**: Enthält die Dokumentation im PDF-Format.
- **1_Code**: Dieses Verzeichnis beinhaltet den gesamten Code der verwendet wurde. Aus speichertechnischen Gründen mussten wir auf das Beilegen von aufbereiteten Trainingsdaten verzichten.
- **2_Weiteres**: Enthält die Dokumentation der vorangegangenen Projektarbeit.

A.2.2. Sprecherlisten

Trainingsdaten 100 Sprecher

FCJF0	FLMA0	MDPK0	MDBP0
FDAW0	FLMC0	MEDR0	MDEM0
FDML0	FMJB0	MGRL0	MDLB0
FECD0	FMKF0	MJEB1	MDLC2
FETB0	FMMH0	MJWT0	MDMT0
FJSP0	FPJF0	MKLS0	MDPS0
FKFB0	FRLL0	MKLW0	MDSS0
FMEM0	FSCN0	MMGG0	MDWD0
FSAH0	FSKL0	MMRP0	MEFG0
FSJK1	FSRH0	MPGH0	MHRM0
FSMA0	FTMG0	MPGR0	MJAE0
FTBR0	FALK0	MPSW0	MJBG0
FVFB0	FCKE0	MRAI0	MJDE0
FVMH0	FCMG0	MRCG0	MJEB0
FAEM0	FDFB0	MRDD0	MJHI0
FAJW0	FDJH0	MRSO0	MJMA0
FCAJ0	FEME0	MRWS0	MJMD0
FCMM0	FGCS0	MTJS0	MJPM0
FCYL0	FGRW0	MTPF0	MJRP0
FDAS1	FJLG0	MTRR0	MKAH0
FDNC0	FJLR0	MWAD0	MKAJ0
FDXW0	FLAC0	MWAR0	MKDT0
FEAC0	FLJD0	MARC0	
FHLM0	FLTM0	MBJV0	
FJKL0	MCPM0	MCEW0	
FKAA0	MDAC0	MCTM0	

Speaker Clustering - 40 Sprecher

MPGL0	FDAC1	FJEM0	FJWB0
MSTK0	FPKT0	MWEW0	FDRD1
FCMR0	FJAS0	MREB0	MJSW0
MWBT0	MCCS0	FJRE0	FKMS0
MCEM0	MMDB1	MWVW0	FSLB1
MRGG0	MDLD0	FRAM1	MTMR0
MBJK0	MJAR0	MRCZ0	MDBB0
MPDF0	FAKS0	MABW0	
MDAB0	FELC0	FCMH0	
MMDM2	MTAS1	MRJO0	
MGWT0	MSJS1	FPAS0	