Karlsruhe Institute of Technology

Carnegie Mellon University

Reverberation Robust Acoustic Modeling Using Time Delay Neural Networks

Master's Thesis of

Emanuel Jöbstl

at the Department of Informatics
Interactive Systems Lab

Reviewer:	Prof. Dr. Alexander Waibel
Second reviewer:	Prof. Dr.-Ing. Rainer Stiefelhagen
Advisor:	M.Sc. Markus Müller
Second advisor:	Dr. Sebastian Stüker

4. November 2017 – 3. May 2018

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

This thesis was written during an exchange at Carnegie Mellon University in Pittsburgh (Pennsylvania) and was kindly supported by a scholarship from DAAD.

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Pittsburgh, 1st of Mai, 2018

. . . . . .

(Emanuel Jöbstl)

Acknowledgments

First, I would like to thank Prof. Dr. Alexander Waibel for his insightful advice and support. The foundation of the InterACT and CLICS exchange programs was foresightful, and I am proud to be one of many students who have greatly benefited from these programs.

I would also like to thank my advisors, M.Sc. Markus Müller and Dr. Sebastian Stüker for their valuable input. I highly appreciate the critical feedback on the results obtained while working on this thesis.

Interesting and important input was received during discussions with the team members of Carnegie Mellon's InterACT lab. I am very thankful that Florian Metze, Ramon Sanabria, Shruti Palaskar and Susanne Burger shared their practical knowledge with me.

Last but not least, I would like to thank Martin Thoma for his critical questioning of my results and his excellent help with proofreading of this work. Furthermore, Martin's thorough public documentation of his own research work was of great practical help to me.

Abstract

This work investigates robust acoustic modeling for speech recognition systems based on hidden Markov models. The focus of this work is put on time delay neural networks. We first design a time delay neural network model for acoustic modeling and provide empirical results that justify our choice of design parameters. Then, we train the time delay neural network on augmented data, and compare its performance on reverberated data with conventional fully connected neural networks.

Zusammenfassung

In dieser Arbeit wird untersucht, wie eine robuste akustische Modellierung für Spracherkennersysteme, die auf Hidden Markov Models basieren, erreicht werden kann. Der Fokus der Arbeit liegt dabei auf Time Delay Neural Networks, die mit augmentierten Daten trainiert werden. Dazu entwerfen wir ein Time Delay Neural Network, wobei die Designentscheidungen empirisch untermauert werden. Danach führen wir Experimente mit verrauschten Daten durch und vergleichen unsere Ergebnisse mit Ergebnissen, die durch vollverbundene neuronale Netze erzielt wurden.

Contents

List of Figures

List of Tables

1. Introduction

Automatic speech recognition is an important way of human computer interaction. The fundamental problem automatic speech recognition attempts to solve is to transform natural spoken language to text, which can then easily be processed by computer systems. Especially with the advent of smart mobile devices, more and more use cases for automatic speech recognition are available, mainly in the form of smart assistants as like Google Now, Microsoft's Cortana and Apple's Siri [1] [2]. There are also many use cases which are not targeting end users, for example the automatic transcription of university lectures [3] and the simultaneous transcription of speeches in the European parliament [4].

Despite the recent success of automatic speech recognition, many systems still rely on microphones which are close to the speaker, or microphone arrays and beam forming. For many use cases, this is a serious drawback. Users might want to use their smart assistant without picking up their device every time. During lectures, it is hard to transcribe questions from the audience without handing a microphone to the person who asked the question. The work [5] gives a good overview about the problems that arise when distant microphones are used. The most prominent problem is reverberation. Reverberation happens, informally speaking, when a signal is interfered by weaker, delayed reflections of itself.

The goal of this work is to investigate how automatic speech recognition could be made more robust against reverberation, using an acoustic model based on time delay neural networks. Our approach for this is the following:

- Chapter 2 introduces basic signal processing and hidden Markov models. This chapter also focuses on the properties of reverberated audio from a signal processing perspective.
- Chapter 3 introduces important related work. We explain automatic speech recognition with a special focus on the structure of acoustic models and the hyperparameters encountered when decoding. We also give a brief introduction to neural networks which is followed by an introduction to time delay neural networks. We conclude the chapter with a summary of acoustic modelling using neural networks. A special focus is put on sequence training criteria and their differentiation.
- Chapter 4 outlines design decisions for our time delay neural network acoustic model and underpins them with experimental evidence. We also briefly investigate the behavior of discriminative training in practice.
- Chapter 5 describes the evaluation of our best time delay neural network acoustic model on reverberated data. We also compare the performance on reverberated data to a fully connected network which performed similarly on a clean dataset.

2. Preliminaries

This chapter summarizes concepts we built upon in this work. First, we give a brief introduction to signal processing and use this framework to explain the properties of reverberation in a more formal way. Then, we introduce hidden Markov models, which are an important concept for understanding most speech recognition systems.

2.1. Reverberation and Reverberated Audio

This section focuses on a formal definition of reverberation and provides some intuition why reverberation makes automatic speech recognition difficult. Reverberation should not be confused with echo: Reverberation overlays a signal with many reflections in a relatively short time context, while echo refers to few reflections that happen up to several seconds from each other. We also give a very brief introduction to signal processing and system theory, according to the book [6].

2.1.1. Continuous Signals and Systems

Sound can, as any other continuous *signal*, be described as a continuous function $y(t)$ where t indicates the time. In literature, the argument t is often dropped when not explicitly needed to make equations easier to read. Signals can be fed into *systems*, which in turn creates an output signal. In the context of this work, we only consider linear, time invariant systems.
Let S be a linear time invariant system, y_1, y_2 continuous signals, and c_1, c_2 constants. The following three properties hold if and only if a given system is an linear time invariant system:

$$S\{c_1 y_1(t) + c_2 y_2(t)\} = c_1 S\{y_1(t)\} + c_2 S\{y_2(t)\} \qquad \text{(linearity)}$$

The linear property allows us to treat application of a system to a signal as a linear transformation in the function space our signals are defined in.

$$y_1(t) = S\{y_2(t)\} \implies y_1(t - t0) = S\{y_2(t - t0)\} \qquad \text{(time invariance)}$$

The time invariance property guarantees that the behavior of a system never depends on the time. In other words, if the input signal to a system is shifted in time, the only difference to the output is a shift in time as well.

$$\left.\begin{aligned} y_1(t') &= y_2(t') \\ x_1(t') &= S\{y_1(t')\} \\ x_2(t') &= S\{y_2(t')\} \end{aligned}\right\} \implies x_1(t') = x_2(t') \qquad \text{(causality)}$$

The causality property guarantees that, if two signals are equal for all times t' before a chosen time t_0, the output signals of the system processing this signals will also be equal up to this point. Simply put, the output of a system up to time t_0 can not depend on any input that happens after t_0. It is worth to note that all real systems are always causal.

A linear time-invariant system can be characterized by its so called impulse response g, defined as the system output when presented with a so called dirac impulse δ.

$$g(t) = S\{\delta(t)\}$$

The dirac impulse δ is a function that is formally defined by the following equation.

$$g(t_2) = \int_{-\infty}^{\infty} y(t)\delta(t - t_2)\,\mathrm{d}t \tag{2.1}$$

It can be said that the dirac impulse is zero for all t not equal to zero. The integral over the dirac impulse defined to be one.

Given definition 2.1, as well as the linear property, we can show that the impulse response of a system is indeed sufficient to calculate the output signal for any given input signal.

$$\begin{aligned} x(t) &= S\{y(t)\} \\ &= S\left\{\int_{-\infty}^{\infty} y(\tau)\delta(\tau - t)\,\mathrm{d}\tau\right\} \\ &= \int_{-\infty}^{\infty} y(\tau)S\{\delta(\tau - t)\}\,\mathrm{d}\tau \\ &= \int_{-\infty}^{\infty} y(\tau)g(\tau - t)\,\mathrm{d}\tau \end{aligned}$$

Furthermore, we define the convolution operation, $*$, for two given signals as follows.

$$(y_1 * y_2)(t) = \int_{-\infty}^{\infty} y(\tau)y_2(\tau - t)\,\mathrm{d}\tau \qquad \text{(convolution)}$$

A convolution is thus an operation that combines two functions to create a new function. Given this definition, we can write the output signal x of a system S given an input signal y as convolution with the impulse response g of the system.

$$x(t) = S\{y(t)\} = (y * g)(t)$$

2.1.2. Properties of Reverberation

The acoustic properties of a room can be approximated as a linear time invariant system [5]. The properties of this system are dependent on the properties of the room itself, for example the shape, size and surface of the wall, as well as the location of the sound source and the location of the receiver. Especially regarding automatic speech recognition, [7]

gives results that show that the location of a speaker relative to the microphone can have a very large impact on recognition results.

The measured impulse response of a real reverberation can be seen in figure 2.1. This specific sample is taken from the Aachen Impulse Response database [8]. Such a sample can be created by creating a very brief sound impulse, for example a clap, in an otherwise silent room, and then recording the sound for a few seconds.

As described in [5], we can divide the impulse response of a reverberation into the direct transmitted sound itself, early reflections, and late reverberation. The intuition behind is that the original sound wave arrives at the receiver first. After that, reflections of the signal which were reflected once by the walls of the room arrive. These reflections are already dampened significantly. Then, reflections of reflections will be received, and so on, until the sound waves become so weak that the room is silent again.

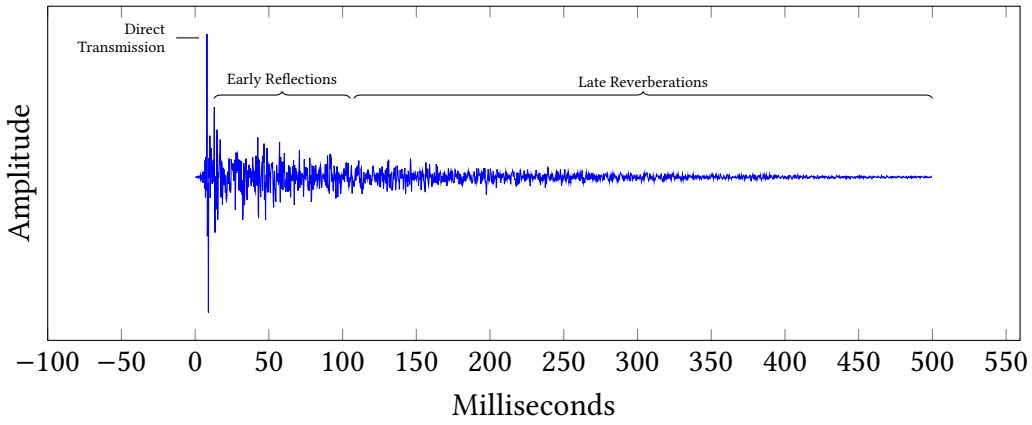

Figure 2.1.: Room impulse response of a lecture hall

It is important to note that late reverberations can be measurable for several hundred milliseconds.

To illustrate the impact of reverberation in a more formal way, we can use the decomposition into direct transmitted sound, early reflection, and late reverberation. We define a approximation of a room impulse response that subsequently overlays an audio signal with weaker copies of itself.

$$g_{rir}(t) = w_0\delta(t) + \sum_{1}^{n} w_n * \delta(t - t_n)$$

Here, w_n are weighting factors, which represent the dampening of our reflections. t_n are the delays until our reflection is received. We now consider the system S_{rir} associated with the impulse response g_{rir}, apply the audio signal y and observe the output x.

$$
\begin{aligned}
x(t) &= S_{rir}\{y(t)\} \\
&= (g_{rir} * y)(t) \\
&= \int_{-\infty}^{\infty} g(\tau) y(\tau - t)\,\mathrm{d}\tau \\
&= \int_{-\infty}^{\infty} \left[w_0 \delta(t) + \sum_n w_n * \delta(t - t_n) \right] y(\tau - t)\,\mathrm{d}\tau \\
&= \int_{-\infty}^{\infty} \delta(t) y(\tau - t)\,\mathrm{d}\tau + \sum_n \int_{-\infty}^{\infty} w_n \delta(t - t_n) y(\tau - t)\,\mathrm{d}\tau \\
&= w_0 y(t) + \sum_n w_n y(t - t_n)
\end{aligned}
$$

If the room impulse response is non-zero over at a certain time interval t_n, the audio signal produced at t will still influence the received signal x at $t + t_n$.

2.1.3. Impact of Reverberation on Digital Audio Samples

Before formally introducing automatic speech recognition during a later chapter, we want to show that the impact of reverberation can be significant for many applications that process sound or speech.

Before a signal can be processed on a computer, it has to be measured. This process is called *sampling*. Formally, we can describe sampling as the following operation, where t_A is called the sampling interval. We call $y_{digital}$ a discrete signal:

$$
y_{digital}(t) = y(t) * \sum_{n=0}^{\infty} \delta t - n t_A
$$

This yields a time series of infinite length which is hard to process. Thus, signals are usually cut into pieces, which are then independently processed from each other. This is called windowing. For the most simple form of windowing, we can set all signal values outside of a certain range to be zero. Formally, this can be defined by multiplying the signal with a rectangle function $\sigma_{rect,a}(t)$:

$$
\sigma_{rect,a}(t) = \begin{cases} 1 & \text{if} \quad -a < t < a \\ 0 & \text{otherwise} \end{cases}
$$

When working with signals, especially for classification tasks, methods built upon the *fourier transform* are used very often [9] [10] [**kitasr2018stueker**]. The fourier transform transforms a signal $y(t)$ from its time domain to the frequency domain $Y(f)$. The resulting function $Y(f)$ is called the *spectrum* and gives the distribution of energy over all frequencies for the original signal $y(t)$. The fourier transformation can be defined for continuous or discrete signals, as well as for discrete and windowed signals. In the case of discrete

windowed signal, this is called the *short time fourier transform*, which was first described in [11]. It can formally be defined as follows, with an arbitrary window function σ:

$$Y(n,\omega) = \sum_{m=-\infty}^{\infty} y(mt_A)\sigma((n-m)t_A)e^{-j\omega n} \tag{2.2}$$

The function $Y(n,\omega)$ gives the signal magnitude for a certain time window n and a frequency window ω. The time resolution depends reciprocally on the frequency resolution and vice versa. It is not possible to increase the frequency resolution while not decreasing the time resolution.

We can investigate the effects of a reverberated signal on the short time fourier transform by applying the same approach as in the previous chapter. The resulting short time fourier spectrum of the reverberated and sampled signal is given as follows:

$$Y(n,\omega) = \sum_{m=-\infty}^{\infty} w_0 y(mt_A)\sigma((n-m)t_A)e^{-j\omega n} + \sum_{m=-\infty}^{\infty}\sum_{n} w_n y(mt_A - t_n)\sigma((n-m)t_A)e^{-j\omega n} \tag{2.3}$$

Equation 2.3 shows that, for sufficiently large t_n and w_n, significant noise is added to neighboring time windows. To recapitulate from the last chapter, the t_n for a large room can be up to several tenths of seconds, while the window size for most applications is in the hundreds of milliseconds.

This mathematic observation shall serve as a motivation for this work. Reverberation, especially late reverberation can be a hard problem that significantly distorts measurements of signals. To cope with reverberation, applications can consider large time windows in the first place. Time delay neural networks, which are introduced in section 3.2 can naturally deal with a such wide windows in a stable way.

We conclude this explanation with a more visual example. Figure 2.2 shows the magnitude of the short term fourier transformation for a short speech segment, as well as the short term fourier transformation for the same speech segment after it has been reverberated using the impulse response shown in Figure 2.1. Such a magnitude representation is also called *spectrogram* in literature.

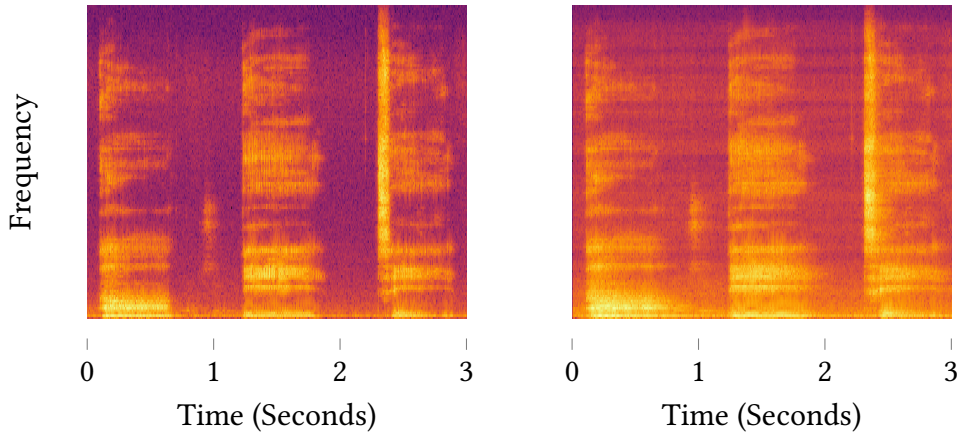

Figure 2.2.: Spectrogram of a clean and a reverberated audio sample. The left side shows a clean recording of a speaker saying *"A B C"*. The right side shows the recording after it was reverberated by the impulse response shown in Figure 2.1. It can clearly be seen that the reverberation caused the signal to become smudged along the time axis.

2.2. Hidden Markov Models

A *Hidden Markov Model* (*HMM*) is a discriminative model. Understanding hidden Markov models throughly is fundamental for understanding automatic speech recognition systems that build upon hidden Markov models.

To explain hidden Markov models, we first introduce the concept of a *Markov chain.* A Markov chain is a sequence of random variables $X = x_1, x_2, \ldots .x_{t-1}, x_t \ldots, x_T$ and a finite number of states $s_1, \ldots, s_n$, where the probability of entering a certain state at time $t + 1$ only depends on the state at time t:

$$P(x_{t+1} = s_{j_{t+1}} | x_t = s_{j_t}, x_{t-1} = s_{j_{t-1}} \ldots, x_1 = s_{j_1}) = P(x_{t+1} = s_{j_{t+1}} | x_t = s_{j_t})$$

This assumption is also called the *Markov assumption.* If the probability of moving from state s_{j_t} to state $s_{j_{t+1}}$ is independent of the current time t, we call a Markov chain *homogeneous.*

We now extend our homogeneous Markov chain and assume that we can no longer observe the state s_{j_t} at a given time t directly, but a symbol v_k that was emitted. We

formally define:

$$
\begin{aligned}
S &= \{s_1, \ldots, s_n\} && \text{(states)} \\
V &= \{v_1, \ldots, v_m\} && \text{(symbols)} \\
A &= (a_{ij}) && \text{(state tansmission probability)} \\
a_{ij} &= p(x_{t+1} = s_j | x_t = s_i) \\
B(k) &= (b_j(k)) && \text{(emisson probability)} \\
b_j(k) &= p(v_k | x_t = s_j) \\
\pi &= (\pi_i) && \text{(initial state probability)} \\
\pi_i &= p(x_1 = s_i)
\end{aligned}
$$

The tuple $\lambda = (S, V, A, B, \pi)$ specifies a *hidden Markov model*. We furthermore introduce the graphical notation for hidden Markov models seen in figure 2.3. Unspecified transitions and observations have probability zero.

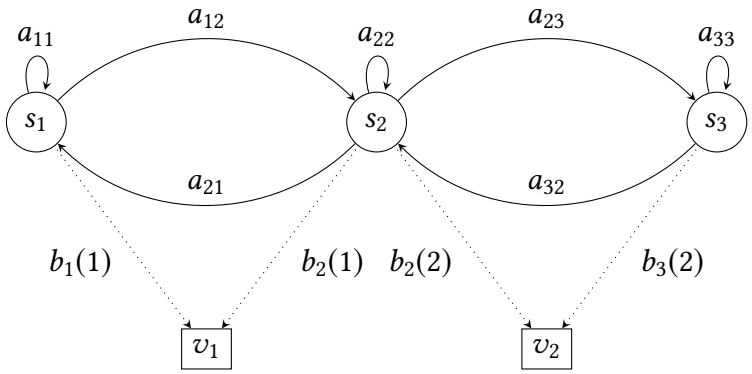

Figure 2.3.: Graphical example of a hidden Markov model with three states and two observations

2.2.1. Forward and Backward Algorithm

Given a sequence of observed emissions, a so called observation $O = \{o_1, \ldots, o_t\}$, a hidden Markov model can be used to evaluate the probability $p(O, \lambda)$ of the observation, given the parameters λ. For calculating this joint probability, the *forward algorithm* or the *backward algorithm* is used. For the forward algorithm, let $\alpha_T(j)$ be the probability of having observed O and being in state s_j at the end of the observation. We define recursively for any previous time t:

$$
\begin{aligned}
\alpha_1(j) &= \pi_j b_j(o_1) \\
\alpha_t(j) &= b_j(o_t) \sum_{i=1}^{N} a_{ij} \alpha_{t-1}(i) \\
p(O|\lambda) &= \sum_{j=1}^{N} \alpha_T(j)
\end{aligned}
$$

For the backward algorithm, we define a similar recursive algorithm, starting from the latest time observation. Here, $\beta_0(j)$ is the probability of observing O when starting from state s_j.

$$\begin{aligned} \beta_T(j) &= 1 \\ \beta_t(j) &= \sum_{i=1}^{N} a_{ij} b_j(o_{t+1}) \beta_{t+1}(i) \\ p(O|\lambda) &= \sum_{j=1}^{N} \beta_0(j) \end{aligned}$$

2.2.2. The Decoding Problem

Besides calculating the probability of an observation sequence, finding the most likely state sequence $X = \{X_1, X_2, \ldots, X_T\}$ given an observation O and a hidden Markov model λ is of interest for automatic speech recognition, as we will explain in section 3.3.5. This problem is called the *decoding problem*, which is solved by the *viterbi algorithm*. The viterbi algorithm is very similar to the forward algorithm. The main difference is that we search for the maximum probability in each step instead of calculating the sum:

$$\begin{aligned} \delta_(j) &= \pi_j b_j(o_1) \\ \mathcal{T}_t(j) &= \arg\max_k \left\{ a_{ij} \delta_{t-1}(k) \right\} \\ \delta_t(j) &= b_j(o_t) a_{ij} \delta_{t-1}(\mathcal{T}_t(j)) \end{aligned}$$

The path with the highest probability can be found by starting at the highest δ_T for the last time step, and then tracing the assigned maximum predecessors backwards using $\mathcal{T}$.

2.2.3. Learning Hidden Markov Model Parameters

Given a hidden Markov model topology, which is essentially only the number of states n and the number of emission symbols m, as well as a training set of observations O, we can use the so called *Baum-Welch* algorithm to optimize the initial state probabilities π, transmission probabilities A and emission probabilities B.

We first define two auxiliary variables, given an observation O and the parameters λ: $\gamma_t(i)$, the probability of being in state i at time t and $\xi_t(i, j)$, the probability of being in state i at time t and state j at time $t + 1$.

$$\begin{aligned}\gamma_t(i) &= P(x_t = i|O,\lambda)\\ &= \frac{P(x_t = i, O|\lambda)}{P(O|\lambda)}\\ &= \frac{\alpha_t(i)\beta_t(i)}{\sum_{j=1}^{N}\alpha_t(j)\beta_t(j)}\end{aligned} \qquad \begin{aligned}\xi_t(i,j) &= P(x_t = i, x_{t+1} = j|O,\lambda)\\ &= \frac{P(x_t = i, x_{t+1} = j, O|\lambda)}{P(O|\lambda)}\\ &= \frac{\alpha_t(i)a_{ij}\beta_{t+1}(j)b_j(o_{t+1})}{\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_t(i)a_{ij}\beta_{t+1}(j)b_j(o_{t+1})}\end{aligned}$$

Using this two auxiliary variables, we can find new parameters λ iteratively. Let π^*, a_{ij}^* and b_j^* be the new parameters after one iteration of the Baum-Welch algorithm.

$$\begin{aligned}b_j^*(k) &= \frac{\sum_{t=1,o_t=v_k}^{T}\gamma_t(j)}{\sum_{t=1}^{T}\gamma_t(j)}\\ \pi_i^* &= \gamma_1(i)\\ a_{ij}^* &= \frac{\sum_{t=1}^{T}\xi_t(i,j)}{\sum_{t=1}^{T}\gamma_t(j)}\end{aligned}$$

It should be noted that the Baum-Welch algorithm is a special case of the *expected maximization* (*EM*) algorithm applied to hidden Markov models. The expected maximization algorithm gives a maximum-likelihood estimate of parameters, even if the data set used for the estimation has incomplete or missing values. A proof can be found in [12].

3. Related Work

This chapter will give a brief introduction to neural networks and time delay neural networks, as well as an introduction to automatic speech recognition with hidden Markov models. We also describe acoustic modelling using neural networks. These are the necessary building blocks for the work presented in this thesis.

3.1. Neural Networks

In machine learning research, the goal of a *neural network* is to approximate arbitrary functions. The basic idea of neural networks, so called *perceptrons*, were first introduced by Rosenblatt in [13]. While the first neural networks were biologically motivated, neural networks can be interpreted as composition of functions. The relation between these functions forms a directed graph. In this work, we will only cover *feed forward neural networks*. They are called feed forward neural networks because there are no feedback connections: The relation between all functions in the neural network forms an acyclic directed graph. Feed forward networks are an important building block for many machine learning applications.
More formally, as described in [14], a feed forward neural network can be described as a model $y = f^*(x, \theta)$, approximating an existing function $y = f(x)$. In this example x is the input, y is the output and θ are the model parameters which are learned during training. f is the function to approximate and f^* is a composition of many functions with the parameters θ.
In practice, the functions composing a feed forward neural network are often simply chained, so that their relation graph simply forms a path. In this case, the functional components are called layers. A feed forward neural network of this form with n layers can be written as follows:

$$y = f_n^* \dots (f_2^*(f_1^*(x, \theta_1), \theta_2) \dots, \theta_n)$$

For this type of neural network, f_1^* is applied to the input, then f_2^* is applied to the output of f_1^* and so on, until the final layer is reached. The output of the final layer is the output of the neural network. Figure 3.1 shows a graph based representation of such a neural network. Each node represents a function, arrows represent data flows.

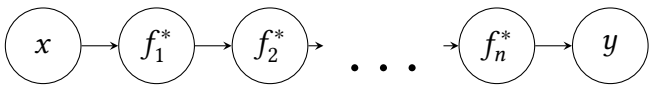

Figure 3.1.: Simple graphical interpretation of a feed forward neural network

Even with this simplification, it has been shown that such a feed forward neural network can approximate any function with any desired accuracy. This is called the *universal approximation theorem* [15]. The caveat is finding the correct function f_n^* to use in each layer and the correct parameters θ_n. Also, finding the function to approximate is non-trivial in the first place. We will discuss these problems in the next sections.

3.1.1. Training Neural Networks

As with most machine learning approaches, we train the neural network by using a *training dataset*, containing a lot of data points sampled from the distribution we seek to approximate. We usually do not want the neural network to excel only on the training data set, but rather to perform well on unseen data. This ability is called *generalization*. We can approximate the error on unseen data by testing the neural network on a *test dataset*. The test dataset must never be used for adjusting the neural network parameters, as this will make the test dataset worthless. The situation where a network performs well on the training dataset but worse on the test dataset is called *overfitting*. The network basically memorizes the distribution of the training dataset, but fails to generalize on the test dataset.
Given the definition in the past section, we can treat the problem of training a given neural network as finding appropriate parameters θ to minimize a certain *error function* or *loss function* $E = \mathcal{L}(y)$, where E denotes the error and y the network output. The error E is usually some metric that judges the network performance based on the training data set. The state of the art algorithm for optimizing the parameters of a neural network is called *stochastic gradient descend*, a special application of naive *gradient descend*.

3.1.1.1. Naive Gradient Descent

To apply gradient descend, we have to calculate the derivative of the loss function, given a certain input, with respect to a certain parameter θ_i. Since this derivative is multi-dimensional, it is called the *gradient*. Given a feed forward neural network, we can write the error as follows:

$$E = \mathcal{L}(f_n^* \dots (f_2^*(f_1^*(x, \theta_1), \theta_2) \dots, \theta_n))$$

Thus, the derivative of the error with respect to a certain weight can be calculated by applying the chain rule:

$$\frac{\partial E}{\partial \theta_i} = \frac{\partial \mathcal{L}}{\partial y_n} \frac{\partial f_n^*}{\partial y_{n-1}} \cdots \frac{\partial f_{i+2}^*}{\partial y_i} \frac{\partial f_i^*}{\partial \theta_i}$$

Where y_k is the result of f_k^*.
We know that the gradient $\frac{\delta E}{\delta \theta_i}$ will become zero in a local minimum, local maximum or saddle point of our error E. Since the gradient also gives the direction of the steepest slope, we can simply update our parameters iteratively, until we converge into a local minimum.

$$\theta_{i,t+1} = \theta_{i,t} - \epsilon * \frac{\delta E_t}{\delta \theta_{i,t}}$$

In this equation, t denotes the current time and the parameter ϵ is the so-called learning rate. ϵ has to be chosen by the designer of the neural network and influences not only the speed of convergence, but also whether the network converges at all. In literature, propagating the error gradient through a neural network is called *backpropagation*. Backpropagation in the context of neural networks was first described in [16].

3.1.1.2. Stochastic Gradient Descend

In practice, naive gradient descend does not work well, as described in [17]. The reason is that the function we seek to optimize when training a neural network is not convex and might have many local minima. Optimizing iteratively on a stationary error term makes this approach prone to falling into local minima.

The state of the art algorithm for training neural networks is called *stochastic gradient descent* (*SGD*), which was first described in a context of neural networks in [18]. Stochastic gradient descent introduces two fundamental changes and is. First, we no longer calculate our error term and corresponding gradient for the whole dataset, but for a randomly sampled subset of our data set, which is called a *mini batch*, where the size of the mini batches is a design parameter. Second, we decrease our learning rate while training progresses. According to [14], iterating on mini batches adds noise to our error term, which in turn offers a regularization effect which was shown to lead to better generalization.
The noise introduced by stochastic gradient descend originates from the fact that we batch our dataset. Even if all batches, if averaged, represent the same distribution as our whole training dataset, each batch has a slightly different distribution. The shape of the loss function, and thus the gradient, changes slightly with each mini batch. This is the main reason why stochastic gradient descend is less prone to get stuck in local minima then naive gradient descend. The noise added by this stochastic process does not go away, even when we reach a global minimum. According to [14], this is be the main motivation for decreasing the learning rate over time. Furthermore, this property implies that the mini batch size is also an important design choice for achieving good generalization, not only for achieving fast training.

A disadvantage of stochastic gradient descend is the slower convergence, since we need more steps. This is remedied by the fact that it is easier to calculate the gradient for a small mini batch instead of the whole dataset at once. In practice, we usually shuffle our dataset, split it into mini-batches, and then process each mini batch once. An iteration over all mini batches in the dataset is called an *epoch*.

3.1.1.3. Learning Rates and Learning Rate Scheduling

With stochastic gradient descend, we choose a separate learning rate ϵ_k for each epoch k. In the scope of this work, we only introduce *exponential decay*, a very simple scheduling algorithm for the learning rate, although numerous other schedulers exist.
Exponential starts with a learning rate k_0 for the initial batch and multiplies the learning

rate by a factor of $0 < p < 1$ after each epoch.

$$k_n = k_{n-1} * p$$

A variant of exponential decay keeps the learning rate constant for a number epochs, until the improvement of error measured on the test dataset falls below a certain threshold. This variant is called *newbob*. Newbob scheduling is widely used by the speech recognition community. It was first introduced in [19].

3.1.1.4. Momentum

Momentum is another technique that was shown to avoid local minima. When using momentum, we do not apply the gradient directly to our model, but rather use a moving average of the gradient of the last batches. There are multiple versions to achieve momentum when using stochastic gradient descend. This work uses the notation introduced in [20], which was throughly analyzed in the context of deep learning in [21]:

$$v_{i,t+1} = v_{i,t} * \rho + \epsilon * \frac{\delta E_t}{\delta \theta_{i,t}}$$
$$\theta_{i,t+1} = \theta_{i,t} - v_{i,t+1}$$

Here, v is called the *velocity*, it is a linear combination of the last velocity and the current gradient. ρ is the term defining the momentum. It controls how much of the velocity is added to the current gradient. All other variables are defined as in section 3.1.1.1.

3.1.2. Neural Network Architectures

In practice, neural network layers are often formed by combining an affine transformation with a non-linear function. Since affine transformations of data vectors can be interpreted as matrix-vector multiplications, we can write a layer function in the following way, where x_k is the input vector for kth layer, W_k is the matrix defining the affine transformation of the kth layer, and φ is the non-linear activation function of layer k.

$$f_k^*(x_k) = \varphi_k(W_k x_k)$$

The count of layers, as well as the size of each W_k are design decisions and depend on the task. We call the count of layers *depth* of a neural network, and count of rows in W_k the *width* of layer k. The contents of W_k, called the *weights* of layer k, are the trainable parameters θ for models of this form.

3.1.2.1. Activation Functions

Non-linear activation functions, or simply *activation functions*, can be roughly classified into two groups: Activation functions which are applied element-wise and thus do not change the dimension of the data, as well as activation functions which are applied on groups of elements of the input vector and change the dimension. The latter case is commonly fund with so called pooling functions. The choice of the activation function φ has significant impact on the performance of a neural network and has been subject to many bodies of research, as summarized in [22]. In this work, we will only discuss the *ReLU*, *p-norm* and *softmax* activation functions.

Softmax

$$\varphi_\tau^{softmax}(X)_i = \frac{e^{\frac{x_i}{\tau}}}{\sum_{x_j \in X} e^{\frac{x_j}{\tau}}}$$

The softmax activation is defined as an operation over the whole input, but does not reduce the dimension. It maps all output values to a space between 0 and 1, preserves the rank of each output and also guarantees that the sum over all outputs is exactly 1. Therefore it produces a valid probability distribution and is usually used for the last layer of a neural network for classification problems. The term τ is called the softmax temperature. It can be used to change the contrast of the distribution produced by the softmax activation. A higher temperature τ will lead to a distribution more smooth. A lower τ will lead to a more sharp distribution, that concentrates a higher probability at the maximum value.

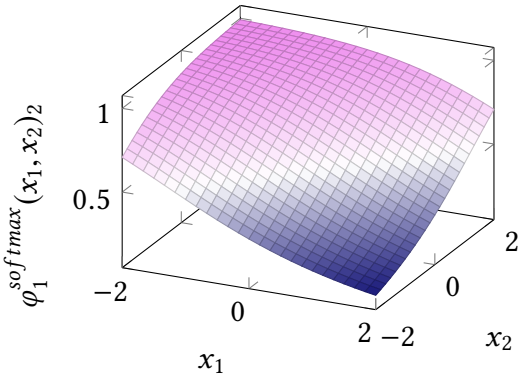

Figure 3.2.: Example of a vanilla softmax ($\tau = 1$) for an input vector containing two values

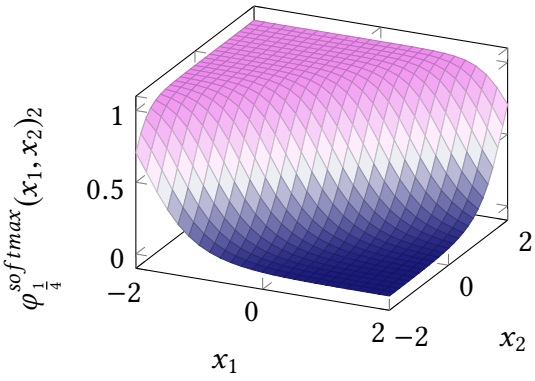

Figure 3.3.: Example of a softmax with adjusted temperature ($\tau = \frac{1}{4}$) for an input vector containing two values

Rectified Linear Unit (ReLU)

$$\varphi^{ReLU}(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$$

First introduced in [23], ReLU nonlinearities have proven successful in practice. The computation of ReLU nonlinearity is cheap, also the gradient never saturates for positive values of x. For negative values, the gradient is zero, which can be a disadvantage.

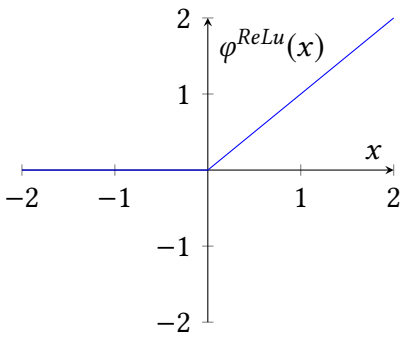

Figure 3.4.: The ReLU function.

P-norm Pooling

$$\varphi^{Lp}(X) = \sqrt[p]{\sum_{x \in X} x^p}$$

The p-norm nonlinearity was described by [24]. This nonlinearity operates on a set X of input elements and reduces them to one. The size of the set X, called group size, as well as the p are design parameters. For a fixed p of 2, the p-norm is also called L2 pooling.

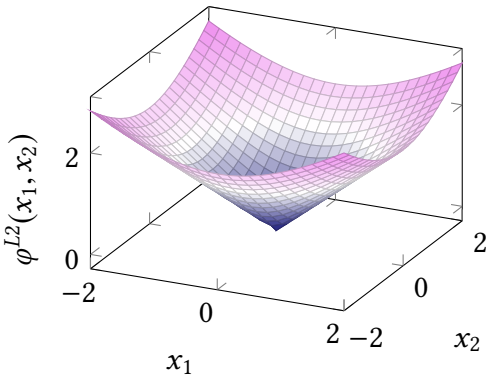

Figure 3.5.: Example of p-norm with $p = 2$ and an input group containing two values

Max Pooling

$$\varphi^{max}(X) = \max_{x \in X} x$$

Max pooling is a nonlinearity that, like p-norm, operates on a group of inputs X. The output of a max pooling nonlinearity is the largest element in X.

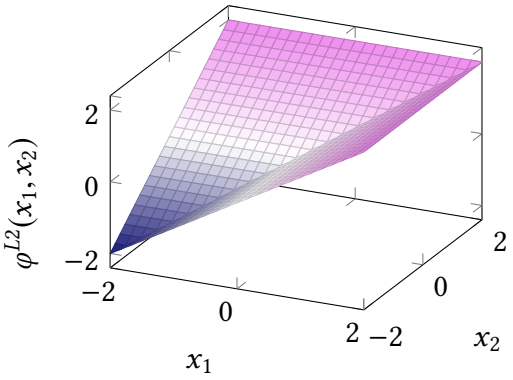

Figure 3.6.: Example of max pooling with an input group containing two values

3.1.2.2. Batch Normalization

Several different forms of batch normalization exist. In this work, we only consider the concept introduced in [25], which can be formulated as follows:

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}}$$

Here, $\mathrm{Var}[x]$ is the variance of our input vector x, $\mathrm{E}[x]$ is the expected value of x and ϵ is a small constant to avoid division by zero. The normalization is not applied on a single value of x, but rather on a whole batch when using stochastic gradient descend. The motivation behind batch normalization is to fix the distribution of activations within the network, where the mean is fixed to 0 and the variance is fixed to 1. This makes it easier to use nonlinearities which saturate for very large or very small activations.

3.1.3. The Cross Entropy Loss Function

There exist numerous loss functions for neural networks. Technically, any metric can be used as loss function, although loss functions do not necessarily have to be metrics. One of the most widely loss functions for classification tasks is the *cross entropy* (*CE*) function. For two probability vectors p and q, the cross entropy can be written as follows:

$$H(p, q) = -\sum_x p(x) \log q(x) \tag{3.1}$$

The cross entropy loss function minimizes the so called *Kullback-Leibler* (*KL*) divergence. The Kullback-Leibler divergence measures the distance of two probability distributions and was introduced in [26]. The Kullback-Leibler divergence is equal to zero if, and only if, the two given distributions are also equal. For two probability vectors p and q, it can be written in the following way:

$$D_{KL}(p, q) = -\sum_x p(x) \log \frac{p(x)}{q(x)}$$

To show the connection between the cross entropy and the Kullback-Leibler divergence, we first assume that q is the distribution we seek to optimize. Thus p can be thought of as an example from our training dataset, or in other words, the distribution we seek to match. Therefore, p, can be assumed to be fixed. We can write the Kullback-Leibler divergence as follows:

$$\begin{aligned} D_{KL}(p, q) &= -\sum_x p(x) \left[\log p(x) - \log q(x)\right] \\ &= -\sum_x p(x) \log p(x) - \sum_x p(x) \log q(x) \end{aligned}$$

Since we are not interested minimizing the distribution p, we can drop the first sum, and thus receive the cross entropy loss as in equation 3.1.

3.1.4. Cross Entropy Loss for Classification

When training neural networks, we often deal with classification problems. In this case, we seek to assign a single class to a given input sample: Our target probability p has exactly one element which is one, all other elements are zero. In this case, the cross entropy loss can be simplified to the so called *negative log likelihood loss*, where i is the index of the correct class in p.

$$D_{NLLL}(i, q) = -\log q(i) \tag{3.2}$$

It has to be noted that this loss functions all operate on probability vectors: All elements have to be between 0 and 1 and sum to unity. Therefore, a softmax activation function is usually applied before calculating the loss.

It can further be shown that training using a cross entropy loss function trains the network to predict *posteriori* probabilities [27]. Formally, given an network input x and a class c_i, the network will predict $p(c_i|x)$. Since posteriors are inherently biased towards more frequent classes, we might want to estimate the likelihood $p(x|c_i)$ instead, depending on our use case. With bayes' theorem, we can write the posterior depending on the likelihood.

$$p(c_i|x) = \frac{p(x|c_i)p(c_i)}{p(x)}$$

We now assume $p(x)$ to be equal for all x and solve the equation for the likelihood:

$$p(x|c_i) = \frac{p(c_i|x)}{p(c_i)}$$

The probability of observing a certain class $p(c_i)$ is called a *prior*. The prior can be estimated from the training data, or from the network output given a random sample of the training data set. Using the likelihood instead of the prior is especially important when the examples in the training set are highly unbalanced with regard of one or more classes.

3.2. Time Delay Neural Networks

Time delay neural networks were introduced by Waibel et al. in [28]. The purpose of time delay neural networks is to model long time dependencies in a robust way. The central idea behind this network type is to use several time frames of input as well as layers that impose temporal structure.

To achieve this, we not only consider the current frame $x_{0,t}$, but also the n previous frames $x_{0,t-1}, \ldots, x_{0,t-n}$ as input. These frames are delayed in time, hence the name. Then, the a weighted sum is applied to the input, where the weights are learned parameters. This is called a *TDNN unit* or *TDNN layer* with a filter kernel of size n. TDNN layers can be stacked. If this is the case, preceding layers have to be extended to produce an output

that contains multiple time frames. We call a network of stacked TDNN layers time delay neural network.
Formally, we can calculate the output of the k-th TDNN layer at time t for the feature vectors with size one:

$$x_{k,t} = \sum_{i=1}^{n} w_{k,i} x_{k-1,t-i}$$

Where $w_{k,i}$ is denotes the i-th weight of the weighted sum used by the k-th layer and $x_{k,t}$ denotes the output of the k-th layer at time t.
This equation can be rewritten using a discrete convolution operator. In this case w_k is called a convolution kernel.

$$x_k = w_k * x_{k-1}$$

For feature vectors containing more than one feature, we introduce the concept of *channels*. A channel can be considered a dimension in the feature. We can write this case for p input channels and q output channels, where $w_{k,j}$ is now a multi-dimensional kernel, producing the output channel j. Each TDNN layer will learn q such kernels, while each kernel is large enough to take all p input channels into account.

$$x_{k,j} = w_{k,j} * x_{k-1}$$

The concept of interpreting TDNN layers as convolutions is not new. The generalization to multi-dimensional convolutions is called *convolutional neural networks* and was shown to be incredibly successful [23]. Since TDNNs only apply the convolution over the time dimension, it can be useful to interpret a TDNN as a so called *finite impulse response* (*FIR*) filter. As described in [6], finite impulse response filters are inherently stable, as they the output is always a sum of finite elements. They also do not accumulate rounding errors. In [28], another interesting property is given: Since there are a lot less learned parameters than operations, a TDNN layer is forced to only focus on the most important features in the data, which leads to better generalization. This so-called *parameter sharing* is achieved by averaging gradients of all operations for the respective weight, independent of the time context t. According to [14], this is one of the main reasons of success for convolutional neural networks in general.

3.2.1. Pooling, Stride and Splicing

Many successful convolutional architectures, as [23], combine convolutional layers with pooling nonlinearities. This is done because pooling over the layer output enables the network to learn several different representations of the same concept, where the pooling will forward the output of the most dominant representation to the next layer [14].
Furthermore, it was shown that using larger steps in the convolution operation can lead to better results [29]. In the context of TDNNs, this means that we would not use adjacent frames for concatenating our input, but frames which are further apart. In literature, this is called *stride* or *strided convolution*, where the stride s gives the distance between concatenated frames. For a strided convolution, the input vector can be written as:

$$x_{0,t}, x_{0,t+s}, \ldots, x_{0,t+ns}$$

It is also possible to define a list of indexes $S = (s_1, ..., s_n)$ to concatenate, which is especially useful if the distance between concatenated frames should not be uniform. This operation is called *splicing* and was introduced by [30]. In this case, our input vector becomes:

$$x_{0,ts_1}, x_{0,t+s_2}, \ldots, x_{0,t+s_n}$$

3.2.2. A visual Example

TDNNs allow a straight-forward interpretation of their layer outputs as multi-dimensional signals, which makes visualization over time helpful. We want to conclude this section with the visualization of a tiny TDNN that was trained to distinguish 53 phones in an audio sample. We use 40 log-mel coefficients as features. Log-mel coefficients are introduced in section 3.3.1.1.

Figure 3.7 shows a tiny TDNN model in the time domain. Our input vector has 40 channels and a time context of 21. The first TDNN layer uses 40 filter kernels of size $5 * 40$ and a stride of 4. It and is followed by an L2 polling with group size 2, resulting in a layer output with 5 time frames and 20 channels. The second TDNN layer uses 40 filter kernels of size $5 * 20$ and a stride of 5. It is also followed by an L2 pooling with group size 2, resulting in an output spanning 1 time frame and 20 channels. In the end, we apply a linear layer that maps the 20 channels to 53 phone classes, including special classes for silence and noise.

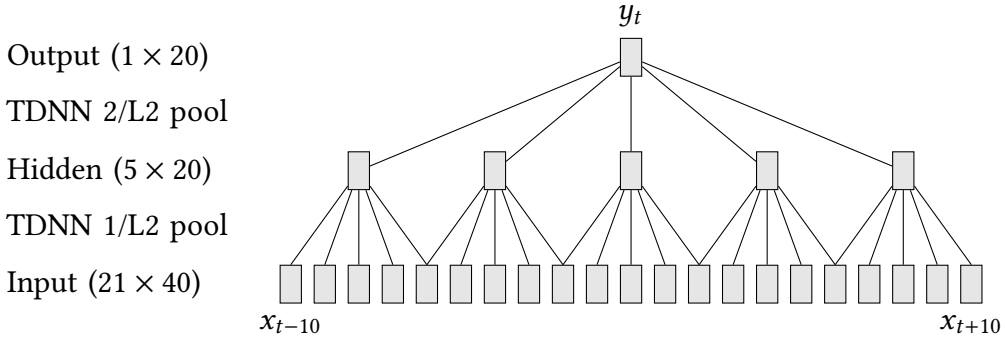

Figure 3.7.: Tiny TDNN model

We trained this tiny TDNN model on 14 hours of voice data using SGD and the newbob-learning rate scheduler. Figures 3.8, 3.9 and 3.10 show the learned filter kernels and weights after training. Warmer colors indicate higher values, the horizontal axis corresponds to time, while the vertical axis corresponds to channels. For the first TDNN layer, it can be seen that filters which are in the same pooling group learn similar parameters. For the other layers, the filters are harder to implement, as TDNN layers do not preserve the ordering of channels. They are still shown for completeness.

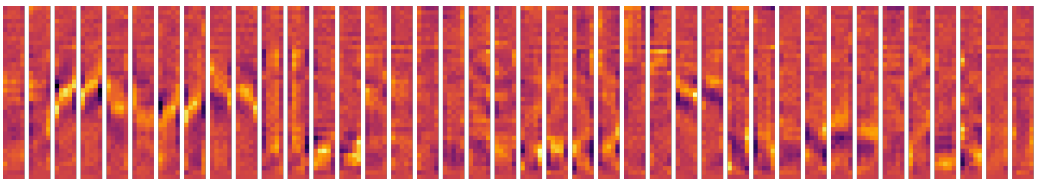

Figure 3.8.: 40 learned filters of TDNN layer 1

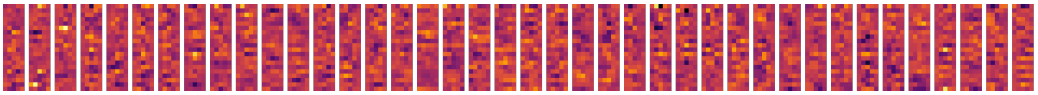

Figure 3.9.: 40 learned filters of TDNN layer 2

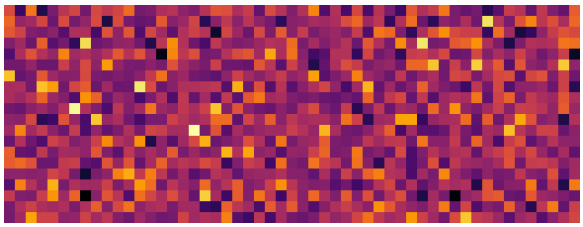

Figure 3.10.: Affine transformation learned by the final linear layer

We now visualize the outputs of different layers given an audio sample of a few seconds length. Figures 3.11 to 3.17 show the output of the TDNN and L2 pooling layers, as well as the output after the linear and the softmax layer. Such layer outputs are also called *activations*. The input features, 40 log-mel features per time frame, are shown in Figure 3.11 and originate from a female speaker saying *"And these are always periods, ladies and gentlemen, accompanied by turbulence."*.

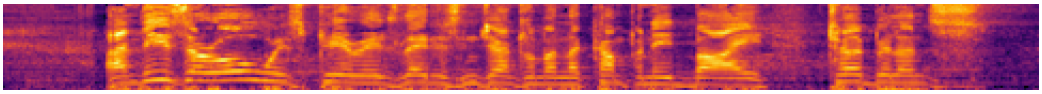

Figure 3.11.: Input features extracted from the audio sample

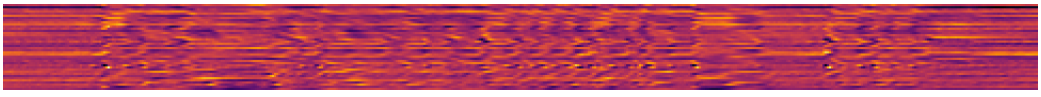

Figure 3.12.: Output of the first TDNN layer

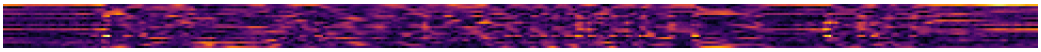

Figure 3.13.: Output of the first L2 pooling layer

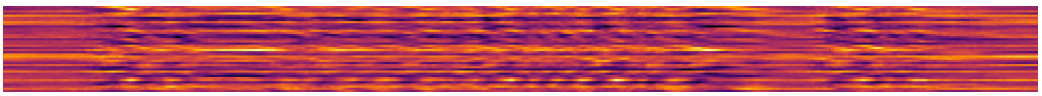

Figure 3.14.: Output of the second TDNN layer

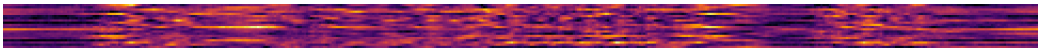

Figure 3.15.: Output of the second L2 pooling layer

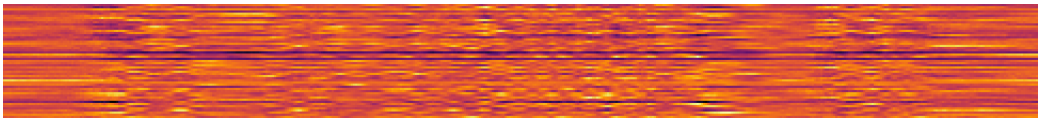

Figure 3.16.: Output of the linear layer

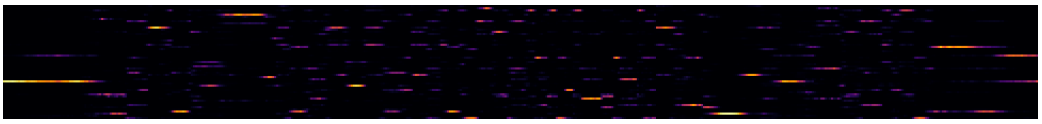

Figure 3.17.: Output of the softmax layer, the final network output

We observe the TDNN outputs in figures 3.12 and 3.14 and their respective L2 pooling outputs in figures 3.13 and 3.15. It can be seen that the TDNN filter kernels generated a signal with multiple channels, where the L2 pooling reduced the number of channels but preserved dominant features. The maximum activations in the TDNN layer output are still dominant after the L2 pooling. We can see that the linear layer output in figure 3.16 generated a prediction for each time frame by combining the features generated by the previous TDNN and L2 pooling layers. Here, the maximum activation corresponds to the most likely phone. The softmax layer in figure 3.17 essentially increases contrast, so the phone with the maximum probability can be easily seen. In this case, each row in the image corresponds to a single phone class, whereas each column corresponds to a time frame.

3.3. Automatic Speech Recognition

Automatic speech recognition (*ASR*) is the task of generating a text transcription from a given sample of spoken language using a computer system. In literature, this problem is also referred to as *speech to text* (*SST*).
Many approaches exist for solving this task. In this work, we will focus on automatic speech recognition using systems that are based on hidden Markov models. The contents of this section, except otherwise noted are based on the books [31], [32] and [1].

Automatic speech recognition systems usually follows an architecture that separates

preprocessing of the audio signal and *decoding*. The preprocessing transforms a brief time window of the audio signal into a feature vector. The decoding uses a statistical model assembled from an *acoustic model*, a *dictionary* and a *language model* to calculate the most likely text representation, given the feature vectors.

More formally, we can treat the decoding as a classification problem. Let a be a set of feature vectors, we seek to find the word sequence w^* that was most likely for a under our model. That can formally be written as follows:

$$w^* = \arg\max_w P(w|a)$$

We do not know $P(w|a)$, but can re-write it using Bayes' theorem:

$$w^* = \arg\max_w \frac{P(a|w)P(w)}{P(a)}$$

Since $p(a)$ is the same for all possible word sequences w, we can drop the denominator from this equation:

$$w^* = \arg\max_w P(a|w)P(w) \tag{3.3}$$

Equation 3.3 is called the *fundamental equation of automatic speech recognition.* While the fundamental equation seems straight forward, the difficult task is to create a reliable and computationally tractable model for approximating the probability of an acoustic feature given a word sequence, $P(a|w)$ and the probability of observing a word sequence $P(w)$. Finding $P(a|w)$ is the purpose of the acoustic model. The language model is responsible for finding $P(w)$.

3.3.1. Preprocessing

Preprocessing, also called the *frontend*, transforms some audio signal into a sequence of feature vectors. To do so, we sample an audio signal using a microphone, then the signal is windowed and the frequency spectrum is calculated for each window of the signal. This process is called short time fourier transform, as defined equation 2.2. Hence, the resulting spectral coefficients are also often called *fourier coefficients*. It should be noted that there are more sophisticated approaches to extract the frequency components of a windowed signal, notably the *continous wavelet transform*, as described in [33], which has better properties in terms of time and frequency resolution.

There exist numerous approaches to transform the spectrum of a signal to useful features. In this work, we only discuss so called *log-mel coefficients*.

3.3.1.1. Log-Mel Coefficients

Log-mel coefficients were introduced in [28] and [34]. This approach is physiologically motivated: Similar to human hearing, the mel-scale provides a better relative frequency resolution in lower frequencies [28].
As given in [35], the mel-scale is defined by the following relation to the regular frequency given in Hertz f_{Hz}:

$$MEL(f_{Hz}) = 2595 \log_{10}\left(1 + \frac{f_{Hz}}{700}\right)$$

Very often, the number of coefficients is reduced to get smaller feature vectors $(\sigma_1, ..., \sigma_n)$. This is done by summing all coefficients in certain windows σ_k on the mel-scale:

$$MEL_k = \int \sigma_k(f) * MEL(f)\delta f$$

Usually a triangle window is used. In literature, the weighted sum is sometimes referred to as *filter bank* [9].
The k-th log-mel coefficient is then calculated by applying a logarithm:

$$lMEL_k = \log(MEL_k)$$

A graphical example of log-mel coefficients can be seen in figure 3.11.

In literature, *Mel-Frequency-Cepstral-Coefficients* (*MFCCs*) are frequently mentioned. They are not to be confused with log-mel coefficients. MFCCs can be calculated by applying an inverse discrete cosine transform on the log-mel coefficients [10].

3.3.2. Acoustic Model

The acoustic model A is responsible of giving us the likelihood of observing a certain word sequence W, for a given sequence of features X.

$$P(X|W)$$

In the case relevant for this work, acoustic models make use of hidden Markov models combined with some discriminative classification approach. The discriminative classifier is responsible for predicting the observation probability for a certain symbol of the hidden Markov model. The most prominent classifier for this specific task is the gaussian mixture models, which combines several normal distributions to approximate a more complex distribution. The parameters of the acoustic model are usually learned using the Baum-Welch equations introduced in the section 2.2.

Before we explain the architecture of the hidden Markov model used for the acoustic model, we have to introduce the linguistic concepts of *phonemes*, *phones* and *allophones*. Phonemes are sound atoms in a spoken language, that are relevant for the meaning of a spoken word. In other words, if a phoneme in a spoken word changes, the meaning of this word would change. Allophones are different pronunciation version of the same

phoneme. *Phones* are simply distinct sounds that are found in a language, regardless of whether swapping a phone changes the meaning of the word or not.

In the context of this work, we only work with acoustic models that model so called context dependent sub-phones, which are then combined into words or word sequences using the dictionary. Sometimes, allophones are modeled as well, to capture variants. For simplicity we will also refer to allophones as phones as well.
Context dependent phones include predecessor and successor phones into their model. This happens by introducing extra hidden HMM states for n-tupels of phones. These context-dependent phones are then called *polyphones*. In the case of a context of two, three or four phones, they are called *diphones*, *triphones* or *quinphones*, respectively. This approach leads to a very high number of HMM states quickly, so context-dependent phones have to be chosen carefully.
For modeling so-called sub-phones, the model distinguishes between start, middle, and end part of an phone as three distinct hidden Markov model states, which represent the phone together. This has the advantage of introducing speed invariance into the model: It does no longer matter whether a certain phone was spoken very slowly or fast. Figure 3.18 shows a hidden Markov model for such an approach. It is important to note that for this approach, several transition probabilities in the HMM are set to zero to force a certain topology.

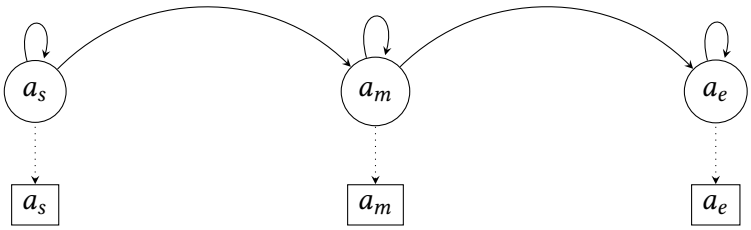

Figure 3.18.: Hidden Markov model for a phone model with three sub-states for start a_s, middle a_m and end a_e

A single HMM state in such a model is also referred to as *distribution* or *senone* for historical reasons [1].

Given a HMM model, we assume we can decompose any word sequence W into a state sequence $s_1, ..., s_n$ and every observation X into an observation sequence $o_1, ..., 0_n$. We can now write the distribution $P(X|W)$ in a state-based way:

$$P(X|W) = \prod_{s_j \in W} p(o_j|s, s_{j-1})$$

3.3.3. Dictionary

The purpose of the dictionary is to describe the pronunciation of words in terms of phones. A common way to create a dictionary is to generate it using a set of rules and a list of words, and then fine-tune it by hand. The dictionary also contains different pronunciation variants for each word.

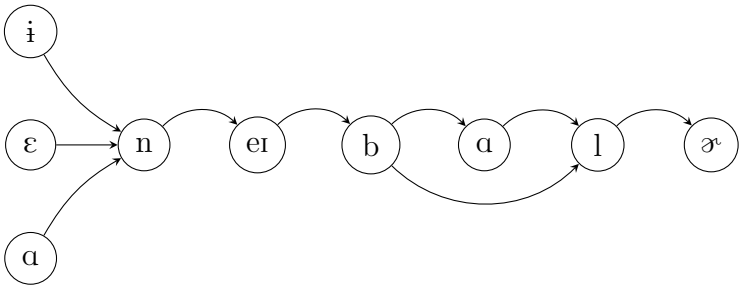

Figure 3.19.: Markov chain for pronunciation variants of the word *enabler*, where the state names correspond to their respective IPA phones

Figure 3.19 shows such a model for different variants of a single word. Emissions are not shown, as each state in this model refers the acoustic model associated with the corresponding phone. Transitions that are not modeled in the dictionary are zero, also the dictionary has no trainable parameters. This example does not take polyphones into account. A model including polyphones would lead a significantly larger model.

3.3.4. Language Model

In the fundamental formula of speech recognition, the language model gives the probability of a word sequence:

$$P(W)$$

For simplification, we can re-write the language model as probability of a word $w \in W$ following a certain sequence of words.

$$P(W) = \sum_{w_i \in W} p(w_i | w_{i-1}, ..., w_{i-n})$$

Since languages tend to have many words, calculating the probability for each possible word sequence would be intractable. Instead of this, so called n-gram language models can be used. n-gram language models count the occurrence of word tuples of length n in a large text corpora. The occurrences are then used to calculate the probability of a word w_i, given $n-1$ predecessors $w_{i-1}, ..., w_{i-n}$. In other words, n-gram language models estimate $p(w_i | w_{i-1}, ..., w_{i-n})$.

For a 2-gram model, the transition probabilities can be directly derived by counting of occurrences for each successor of each word. An example for such a model is shown in figure 3.20. For larger n, we have a state for each feasible combination of words. The number of states quickly grows very large in this case.

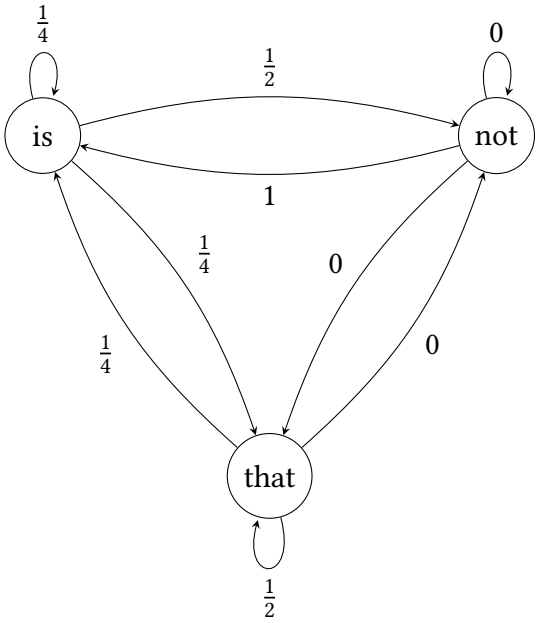

Figure 3.20.: Markov chain built from a 2-gram language model for the sentence *"That that is, is; that that is not, is not."*, omitting start and end literals

It should be noted that other approaches for language modeling exist, for example recurrent neural network based language models [36].

3.3.5. Decoding Process

The decoding step combines the acoustic model, dictionary, and language model to find the text for a given observed utterance. This section describes decoding in a very fundamental way. In real-world applications, many more details are considered. [32] and [1] give a very detailed description of different decoding approaches.
The *viterbi approximation* [32] states that the most likely word sequence can be approximated with the most likely state sequence. With this assumption, it is possible decompose any word sequence W into a number possible sequences of HMM states using the introduced HMM-based acoustic model and the dictionary. Let $Q_W = (s_1, ..., s_n)$ be such a HMM state sequence associated with a word sequence W. We can re-write the fundamental formula of speech recognition in the following way [37]:

$$W^* = \arg\max_W P(W) \max_{Q_W \in W} P(X|Q_W)$$

With this formulation of the search problem, finding the probability for a certain word sequence given the observation sequence X is reduced to finding the probability of observing a state sequence, given an observation sequence. Since our acoustic model is HMM-based, we can use the viterbi algorithm to find this probability. We can even expand the idea and include the language model into the viterbi search: Every time a word boundary is crossed, we can multiply the probability of our current path with the probability of the

given word transition. This approach can be formulated in the following way:

$$W^* = \arg\max_W \prod_{w_i \in W} P(w_i|w_{i-1}, ..., w_{i-n}) \max_{Q_w \in w_i} \prod_{s_j \in Q_w} p(o_j|s_j, s_{j-1}) \tag{3.4}$$

Here, we evaluate the language model for each word w_i in our word sequence W, given the predecessors $w_{i-1}, ..., w_n$ relevant for the language model. We multiply this probability with the sum of likelihoods of each state s_j in most likely variant Q of word w_j, given the previous state and the observation at o_j. With this formulation, the whole model can be expressed as a single HMM. More specifically, word transition probabilities are also modeled in the HMM for each possible word transition. While it is, in theory, possible to search for the most likely word sequence using the viterbi algorithm using this approach, models built this way fail for large vocabulary tasks as they become intractable. A more thorough explanation about how language models are efficiently integrated can be found in [31].

3.3.5.1. Decoding Hyperparameter

The formulation of the viterbi algorithm given in section 2.2.2 is naive and visits all possible states. We can save resources by using a greedy breath-first search with a heuristic, that picks the next state to visit. Such a search algorithm is also called *A^* algorithm* in literature [38]. However, this approach is also not tractable in practice, as the model becomes very large for tasks with many vocabularies. Therefore, a so-called *beam search* is used. A beam search discards paths with a probability that fall below a certain threshold, which we call the master beam or *mb*.

We consider three more details. First, we want to avoid multiplications, because they are computationally expensive. We therefore maximize the negative log-likelihood instead of raw probabilities. Second, we add the scaling factor l_z to weight the acoustic model versus the language model, which was shown to be very useful in practice. Also we add another parameter l_p which can be used to penalize too short or too long word sequences. With this in mind, we can rewrite the fundamental formula in the following way:

$$\begin{aligned} W^* &= \arg\max_W \; -\log\left(P(X|W)P(W)^{lz}|W|^{lp}\right) \\ &= \arg\max_W \; -\log P(X|W) - l_z \log P(W) - l_p \log(|W|) \end{aligned}$$

The master beam *mb*, and the coefficients *lz* and *lp* are hyper-parameters that have to be tuned for optimal results.

3.3.6. Error Metrics

Every machine learning system needs to be tested on data it has not seen before. For this purpose, *error metrics* are used. Error metrics provide a way to objectively measure the performance of a machine learning system. In the field of automatic speech recognition,

three dominant error metrics are used: *frame error rate* (*FER*), *Word error rate* (*WER*) and *sentence error rate*.

3.3.6.1. Frame Error Rate

Frame error rate simply measures the classification error on frame level and is mostly used in connection with acoustic models. It can be calculated by counting how often the class of a frame was predicted incorrectly by the acoustic model. Given a sequence of classes predicted by our model $Y* = (y_1^*, ..., y_n^*)$ and a sequence of reference labels $Y = (y_1, ..., y_n)$, we can write the frame error rate as follows:

$$\mathrm{FER}(Y^*, Y) = \frac{1}{n} \sum_{i=0}^{n} 1|_{y_i^* = y_i}$$

Here, $1|_{y_i^* = y_i}$ is an indicator function that is one if y_i^* and y_i are equal and zero otherwise. The frame error rate is always between zero and one and is usually given as a percentage.

The interpretation of what a class represents depends on the acoustic model. It could, for example, be a phoneme, an allophone or a hidden Markov model state.

3.3.6.2. Word Error Rate

The word error rate is the most common error metric when testing complete speech recognition tools. Before we explain the word error rate, we introduce the so called *Levenshtein distance*. The Levenshtein distance of to sequences $A = (a_1, ..., a_n)$ and $B = (b_1, ..., b_n)$ gives the minimum number of insertions, deletions and substitutions which are necessary to transform A to B. The Levenshtein distance can recursively defined as:

$$\begin{aligned}
&\mathrm{LD}_{0,j}(A, B) = j \\
&\mathrm{LD}_{i,0}(A, B) = i \\
&\mathrm{LD}_{i,j}(A, B) = \begin{cases} \mathrm{LD}_{i-1,j-1}(A, B) + 0 & \text{if } a_i = b_i \\ \min \begin{cases} \mathrm{LD}_{i-1,j}(A, B) + 1 \\ \mathrm{LD}_{i,j-1}(A, B) + 1 \\ \mathrm{LD}_{i-1,j-1}(A, B) + 1 \end{cases} & \text{otherwise} \end{cases}
\end{aligned}$$

The Levenshtein distance of the whole sequence is:

$$\mathrm{LD}(A, B) = \mathrm{LD}_{|A|,|B|}(A, B)$$

The word error rate $\mathrm{WER}(W^*, W)$ of two word sequences $W*$ and W can now be defined using the Levenshtein distance:

$$\mathrm{WER}(W^*, W) = \frac{\mathrm{LD}(W^*, W)}{|W|}$$

Here, W is the *reference hypothesis*, W^* is the output for the system we seek to benchmark.

3.3.6.3. Sentence Error Rate

The *sentence error rate* counts errors on a sentence level. As soon as a single word in a sentence is different as in the reference, the sentence is rated as incorrect. The sentence error rate is usually calculated over a corpus V that contains many sentences $v_1, ..., v_n$. It can be formally written as:

$$\mathrm{SER}(V^*, V) = \frac{1}{n} \sum_{i=0}^{n} 1|_{v_i^* = v_i}$$

Here, V is the reference, $V*$ is the set of sentences predicted by our model. $1|_{v_i^* = v_i}$ is a indicator function that is one if and only if v_i^* equals v_i.
It should be noted that in ASR, a sentence is not essentially a sentence in the grammatical sense, but rather a certain segment of a larger corpus. Therefore, sentences are also referred to as *utterances*.

3.4. Acoustic Modeling using Neural Networks

As mentioned in section 3.3.2, acoustic models combine a discriminative classification algorithm with a hidden Markov model. Neural networks can be used as such a discriminative classification algorithm. A neural network can be used to directly predict the likelihood of a state s_i of the hidden Markov model $p(x|s_i)$ given a feature x. In this context, we call the states *labels* and the features *samples*. This approach makes our model essentially a Markov chain, since the states are now assumed to be directly observable. In [39], an excellent summary of the approach is given, although first experiments with neural network based acoustic modeling already happened significantly earlier [40]. Notably, recent advancements were with robust acoustic modeling using a TDNN by Peddinti et al. [41] [30]. Their experiments have shown that an acoustic model based on a TDNN was significantly better for reverberated speech than a acoustic model based on a fully connected network, when trained on reverberated data.

We will now introduce three loss functions that are used in the field of automatic speech recognition. Since neural network training is based on gradient descend, will also show how we can derive gradients from the given loss functions. Brief versions of this derivations can be found in [42], very detailed analysis of the loss functions in the context of ASR can be found in [43] and [44].

3.4.1. Maximum Likelihood Estimation

For maximum likelihood training, we use the viterbi algorithm to calculate the most likely state sequence on for a given reference sentence and corresponding observations. This is called a *forced alignment*. Then, each audio frame is labeled with its most likely state, according to the viterbi pass. We treat each state as a separate class and use this dataset for training a model using the negative log posterior as loss function. This optimizes the

frame error rate. Formally, this loss function be written for an utterance as:

$$\mathcal{L}_{\text{CE}} = -\sum_{t=0}^{n} \log p^*_{s_t}$$

Where n is the length of the utterance, s_t is the correct state at time t, given by the viterbi pass and o_t is the observed feature vector at time t. $p^*_{s_t}$ a shorthand for the predicted posterior probability for state s_t produced by our model, formally $p^*(s_t|o_t)$. For a single frame, this loss function is equal to the negative log likelihood loss from equation 3.2. Therefore, we also refer to this training variant as cross entropy loss based training.

We now assume that the probability $p^*_{s_t}$ is produced by the output of a neural network model which uses a softmax activation as the final layer. Let $y^*_{s_j}$ be the output of the layer before the softmax layer for state s_j. We can calculate the gradient for our loss function for a single frame with respect to $y^*_{s_j}$ as follows:

$$\begin{aligned}
\frac{\partial \mathcal{L}_{\text{ML}}}{\partial y^*_{s_j}} &= -\frac{\partial \log p^*_{s_t}}{\partial y^*_{s_j}} \\
&= -\frac{\partial \ \log \frac{\exp(y^*_{s_t})}{\exp\left(\sum_{i=0}^{n} y^*_{s_j}\right)}}{\partial y^*_{s_j}} \\
&= -\frac{\partial \ y^*_{s_t} - \sum_{i=0}^{n} y^*_{s_j}}{\partial y^*_j} \\
&= 1 - \delta_{tj}
\end{aligned}$$

Where δ_{ij} is the so called *kronecker delta*.

$$\delta_{ij} = \begin{cases} 1 & i = j \\ 0 & \text{otherwise} \end{cases}$$

After the neural network acoustic model was trained over the whole training set, labels can be re-written, and another neural network acoustic model can be trained with the new labels. This process can be iterated several times to improve results.

3.4.2. Maximum Mutual Information Estimation

Maximum mutual information (*MMI*) estimation was introduced for estimating hidden Markov model parameters in speech recognition systems in [45]. The training criterion maximizes the ability of the model to discriminate between the correct distribution and any other distribution. In other words, this training criterion minimizes the sentence error. Let $\mathcal{V}$ be the set of all utterances. In the context of speech recognition, we can give a loss function that maximizes mutual information between an observation sequence $O = (o_1, ..., o_n)$ and a word sequence $U \in \mathcal{V}$ as follows:

$$\mathcal{L}_{\text{MMI}} = -\log \frac{p(O|U)P(U)}{\sum_{V \in \mathcal{V}} p(O|V)P(V)}$$

In [42], a very similar formulation is given, which is maximized over all utterances, while our loss function is minimized for a single utterance U. This is more convenient when working with neural networks. The original formulation af a convenient optimization criterion for MMI estimation was given in [46].
Given the viterbi approximation from equation 3.4, we can assume that our word sequences can be separated to state sequences $S_U = (s_{U,1}, ..., s_{U,n})$ which we found using our speech recognition system with $P(U) = \prod_{t=0}^{n} p^*(s_{U,t})$. This approach was also chosen in [45], to simplify the error criterion. Furthermore, we replace the set $\mathcal{V}$ by the set $\mathcal{M}$, which contains the m best state sequences found during our forward-backward pass for the utterance U, a practical simplification which is given in [46]. The criterion becomes:

$$\mathcal{L}_{\text{MMI}} = -\log \frac{\prod_{t=0}^{n} p^*(o_t|s_{U,t}) p^*(s_{U,t})}{\sum_{V \in \mathcal{M}} \prod_{t=0}^{n} p^*(o_t|s_{V,t}) p^*(s_{V,t})}$$

With the theorem of bayes, we can expand:

$$p^*(o_t|s_{U,t}) = \frac{p^*(s_{U,t}|o_t)}{p^*(s_{U,t})}$$

With this, we can simplify and express $\mathcal{L}_{\text{MMI}}$ in terms of $p^*(s_{U,t}|o_t)$.

$$\mathcal{L}_{\text{MMI}} = -\log \frac{\prod_{t=0}^{n} p^*(s_{U,t}|o_t)}{\sum_{V \in \mathcal{M}} \prod_{t=0}^{n} p^*(s_{V,t}|o_t)}$$

This expression can be differentiated with respect to the posterior probability $p^*(s_{U,t}|o_t)$ to calculate a gradient for training a neural network model with backpropagation. Again, let $p^*_{s_{j,t}}$ be $p^*(s_{j,t}|o_t)$.

$$\begin{aligned}
\frac{\partial \mathcal{L}_{\text{MMI}}}{\partial p^*_{s_{j,t}}} &= \frac{\partial \log \sum_{V \in \mathcal{M}} \prod_{t=0}^{n} p^*_{s_{V,t}}}{\partial p^*_{s_{j,t}}} - \sum_{t=0}^{n} \frac{\partial \log p^*_{s_{U,t}}}{\partial p^*_{s_{j,t}}} \\
&= \frac{\sum_{V \in \mathcal{M}} \delta_{(s_{V,t})(s_{j,t})} \prod_{t=0}^{n} p^*_{s_{V,t}}}{\sum_{V \in \mathcal{M}} \prod_{t=0}^{n} p^*_{s_{V,t}}} \frac{1}{p^*_{s_{V,t}}} - \frac{\delta_{(s_{U,t})(s_{j,t})}}{p^*_{s_{j,t}}}
\end{aligned}$$

We can now rewrite the first fraction in terms of probabilities, more precisely the probability of visiting state s_j at time t while we observe O. The fraction is indeed this probability: We divide the sum of probability of state sequences which visit $s_{j,t}$ by the sum of the probability of all sequences. We can use this to simplify the first fraction.

$$\frac{\partial \mathcal{L}_{\text{MMI}}}{\partial p^*_{s_{j,t}}} = \frac{p(s_{j,t}|O,t)}{p^*_{s_{V,t}}} - \frac{\delta_{(s_{U,t})(s_{j,t})}}{p^*_{s_{j,t}}}$$

This formulation is familiar. It corresponds to the definition of $\gamma_t(j)$ from section 2.2.3, that is produced by the forward-backward algorithm when training hidden Markov model parameters. We conclude this derivation by a compact formulation of the gradient for the MMI loss function:

$$\frac{\partial \mathcal{L}_{\text{MMI}}}{\partial p^*_{s_{j,t}}} = \frac{\gamma_t(j) - \delta_{(s_{U,t})(s_{j,t})}}{p^*_{s_{j,t}}} \tag{3.5}$$

In literature, particularly [42], a variant, which has to be maximized, is used:

$$\frac{\partial \mathcal{L}_{\text{MMI}}}{\partial p^*_{s_{j,t}}} \approx \frac{\delta_{(s_{U,t})(s_{j,t})} - \gamma_t(j)}{\kappa} \tag{3.6}$$

Here, κ is an acoustic scaling factor, which corresponds to the l_z introduced in section 3.3.5.1.

3.4.3. Overall Risk Criterion Estimation

The family of *overall risk criterion estimation* (*ORCE*), or *minimum bayes risk* (*MBR*) objective functions for hidden Markov models was introduced in [47]. They are optimized to minimize the number of insertions, deletions and substitutions at either word, phone, or state level. The specific variants are called *minimum word error* (*MWE*) and *minimum phone error* (*MPE*) criterion for words and phones. For minimizing the error at state level, the criterion is called *state minimum bayes risk* (*sMBR*). Several pieces of literature suggest that the MPE and sMBR objective functions, if carefully tuned, perform better than frame based maximum likelihood estimation in experiments [48][43][44][30].

We can formally define the whole family as loss function:

$$\mathcal{L}_{\text{OCRE}} = \frac{\sum_{V \in \mathcal{V}} P(O|V)P(V)\lambda(V,U)}{\sum_{V' \in \mathcal{V}} P(O|V')P(V')}$$

Where $\mathcal{V}$ is a set containing all possible hypothesis for an observation O, and U is the reference hypothesis. $\lambda(V,U)$, called the raw accuracy, would ideally be the levenshtein distance of U and V, divided by the length of the correct hypothesis U. $\lambda(V,U)$ can either be measured on state level, phone level or on word level for sMBR, MPE and MWE, respectively. In practice, simpler metrics are often used, which do not rely on the computationally expensive calculation of the levensthein distance [48].

To derive a gradient for training a neural network, we again formulate the criterion on state level:

$$\mathcal{L}_{\text{OCRE}} = \frac{\sum_{V \in \mathcal{V}} \lambda(U,V) \prod_{t'=0}^{n} p^*_{s_{V,t'}}}{\sum_{V' \in \mathcal{V}} \prod_{t'=0}^{n} p^*_{s_{V',t}}}$$

Now, we differentiate, factor out the first fraction and cancel the last fraction:

$$\begin{aligned}\frac{\partial \mathcal{L}_{\text{OCRE}}}{\partial p^*_{s_{j,t}}} &= \frac{\sum_{V\in\mathcal{V}} \lambda(U,V) \prod_{t'=0}^{n} p^*_{s_{V,t'}}}{\sum_{V'\in\mathcal{V}} \prod_{t'=0}^{n} p^*_{s_{V',t}}} \frac{\sum_{V'\in\mathcal{V}} \delta_{(s_{V',t})(s_{j,t})} \frac{1}{p^*_{s_{j,t}}} \prod_{t'=0}^{n} p^*_{s_{V',t'}}}{\sum_{V'\in\mathcal{V}} \prod_{t'=0}^{n} p^*_{s_{V',t}}} \\ &- \frac{(\sum_{V\in\mathcal{V}} \lambda(U,V) \delta_{(s_{V,t})(s_{j,t})} \frac{1}{p^*_{s_{j,t}}} \prod_{t'=0}^{n} p^*_{s_{V,t'}}}{\sum_{V'\in\mathcal{V}} \prod_{t'=0}^{n} p^*_{s_{V',t}}} \frac{\sum_{V'\in\mathcal{V}} \prod_{t'=0}^{n} p^*_{s_{V',t}}}{\sum_{V'\in\mathcal{V}} \prod_{t'=0}^{n} p^*_{s_{V',t}}} \\ &= \frac{1}{p^*_{s_{j,t}}} \frac{\sum_{V\in\mathcal{V}} \lambda(U,V) \prod_{t'=0}^{n} p^*_{s_{V,t'}}}{\sum_{V'\in\mathcal{V}} \prod_{t'=0}^{n} p^*_{s_{V',t}}} \left[\frac{\sum_{V'\in\mathcal{V}} \delta_{(s_{V',t})(s_{j,t})} \prod_{t'=0}^{n} p^*_{s_{V',t'}}}{\sum_{V'\in\mathcal{V}} \prod_{t'=0}^{n} p^*_{s_{V',t}}} \right. \\ &\left. - \frac{\sum_{V\in\mathcal{V}} \lambda(U,V) \delta_{(s_{V,t})(s_{j,t})} \prod_{t'=0}^{n} p^*_{s_{V,t'}}}{\sum_{V\in\mathcal{V}} \lambda(U,V) \prod_{t'=0}^{n} p^*_{s_{V,t'}}} \right]\end{aligned}$$

We now use $\gamma_t(t)$ like before and introduce two new symbols to simplify the equation.

$$\frac{\partial \mathcal{L}_{\text{OCRE}}}{\partial p^*_{s_{j,t}}} = \frac{\gamma_t(j)}{p^*_{s_{j,t}}} \left[\overline{\lambda}(U,V) - \overline{\lambda}(U,V|s_{j,t})) \right]$$

$\overline{\lambda}(U,V)$ is the average raw accuracy for all sequences, weighted by the probability of each sequence. $\overline{\lambda}(U,V|s_{j,t})$ is the average raw accuracy for all sequences that pass through state j at time t, also weighted by the probability of each sequence.

Again, in [42], a variant of this gradient is defined:

$$\frac{\partial \mathcal{L}_{\text{OCRE}}}{\partial p^*_{s_{j,t}}} \approx \frac{\gamma_t(j)}{\kappa} \left[\overline{\lambda}(U,V) - \overline{\lambda}(U,V|s_{j,t})) \right]$$

Where κ is an acoustic scaling factor, like before.

4. Design of a TDNN acoustic model

Our approach for creating a robust TDNN acoustic model is to find a TDNN model that performs good on clean data first. Therefore, we had to make several design decisions. Most of the design decisions were justified by experiments, while some were taken from related work. This chapter summarizes the decisions made and the corresponding results. It should be noted that the experiments about robust acoustic modeling with TDNNs described in [30] and [41] provided the main motivation for this work, therefore we based some of our parameters on their result.

4.1. Training Data and System Setup

All experiments in this work only concern the neural network part of our acoustic model, which is a HMM/TDNN hybrid. The speech recognition system itself, as well as all data, the HMM part of the acoustic model, the dictionary and the language model, are based upon the system described in [49]. The system is built upon the Janus recognition toolkit [50]. We utilize a four-gram language model and the CMU Pronouncing Dictionary [51], which uses 39 phones. The acoustic model uses quinphones and has 8156 different distributions, which means that the HMM part of our acoustic model has 8156 different states.

The training and test dataset for the acoustic model consist of 468 hours of English speech from the TED-LIUM v2 [52], Broadcast News [53] and Quaero 2010-2012 datasets. From these 468 hours, 17 hours are randomly selected as test set, 451 hours are used for the training set.

The development dataset, used for tuning the hyperparameters, consists of the english IWSLT 2013 evaluation dataset for the ASR track [54]. This dataset consists of 3.9 hours of TED talks. The word error rates for all experiments in this chapter were measured on this development set.

Each frame of samples in the datasets consists of 40 log-mel features which were normalized over the whole utterance to have mean zero and variance two. Each frame covers 32 milliseconds. The frame shift between successive frames is 10 milliseconds. The sampling rate of the audio data was 16 kHz.

The neural network training was done using a custom framework, build on top of PyTorch [55]. Pytorch is a machine learning framework that supports GPU accelleration, parallelization along multiple systems and automatic differentiation. In the context of this work, several contributions were made to the PyTorch framework.

4.2. Neural Network Parameters

This section focuses on parameters that are related to the neural network design. For all models in this section, the word error rate was estimated by using l_p and l_z that were tuned for each model separately. The priors were estimated by counting labels over the whole training set.

4.2.1. Input Context

The time input context of all our TDNN models is $(-13, 9)$, which means the TDNN sees the current frame, thirteen frames in the past, and nine frames in the future. This parameter was taken from the smallest TDNN model described in [41].

4.2.2. Count and Width of Layers

The count of layers and width of each layer is one of the most important design parameters for neural networks. As in [41], all our models use the same amount of channels for each TDNN layer. Only the count of observed time frames changes with each layer.

Model:	Four Layer				Five Layer				
Layer	1	2	3	4	1	2	3	4	5
Kernel Size	5	2	2	2	5	5	3	2	2
Stride	3	2	2	2	2	1	1	1	1

Table 4.1.: Kernel size and stride parameters for the two different architectures

We decided to test a four layer model with exactly the same parameters as the smallest model in [41]. This model also included a splicing layer after layer one. The splicing configuration was $(0, 1, 2, 3, 3, 4, 5, 6)$, relative to the previous layer. For the second model we tested, we decided to use larger kernels and one more layer. For this purpose, we removed the splicing layer and increased the layer count to five. Table 4.1 gives the kernel size and stride over time for each layer in each architecture. For both models, we used an L2 pooling nonlinearity with group size of ten followed by a batch normalization layer after each TDNN layer.
Figure 4.1 shows the word error rate for the two different architectures, given the count of channels. It can be seen that the four-layer architecture performed better than the five-layer architecture. We found the optimal count of channels for the four-layer model to be 300. This contradicts [41], where 400 channels were used.

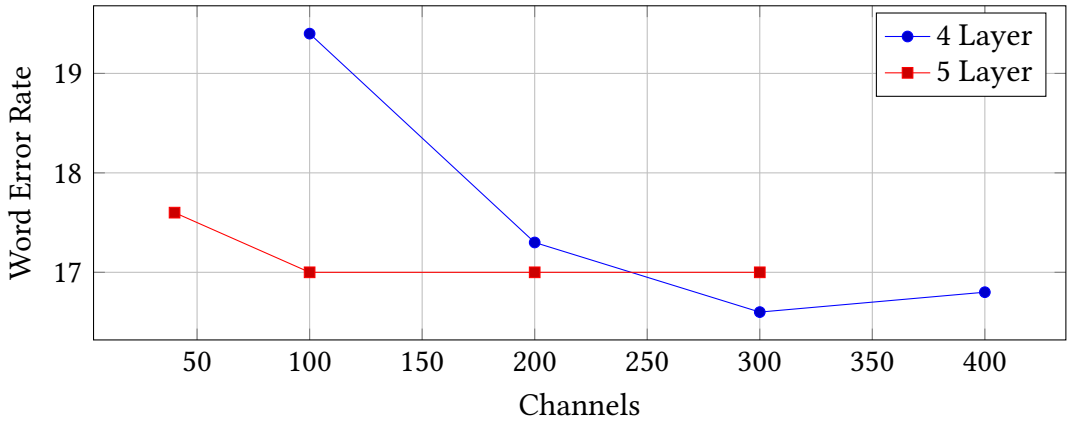

Figure 4.1.: Word error rate for different choices of layer and channel count

4.2.3. Nonlinearity

Following [24], we tested a L2 pooling nonlinearity with group size if ten after each TDNN layer. Using the L2 norm can be problematic, as the gradient is not defined when all inputs in the pooling group are zero. The authors of [24] propose to use a modified batch normalization layer after each L2 pooling layer, which solves the problem. We propose an alternate approach, which is setting the gradient to zero if all inputs become zero. Furthermore, we also tested max pooling as a possible nonlinearity. All experiments were conducted on the four-layer architecture described before. Figure 4.2 shows the results.

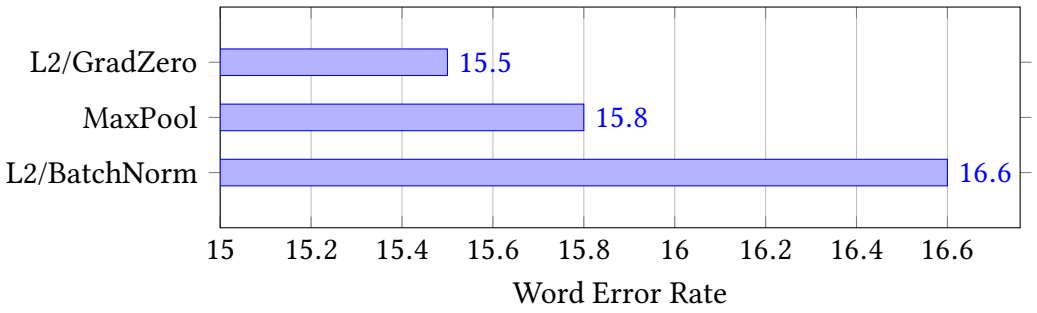

Figure 4.2.: Word error rate for different choices of nonlinearities

In our case, the usage of L2 pooling with a modified gradient outperformed the other nonlinearities. It was not possible to compare L2 pooling without any modifications, as our training became unstable.

4.3. Training Setup

This section is focused on the training setup for our acoustic model. Out setup is closely related to the setup described in [49]. We essentially use the same speech recognition system, the same labels, as well as the same samples for training our acoustic model.

4.3.1. Shuffling of Dataset

For our experiments, we benchmarked two different shuffling strategies with a six-layer fully connected network: Shuffling of the whole dataset once before training, and shuffling of the whole dataset before each epoch. The results can be seen in figure 4.3.

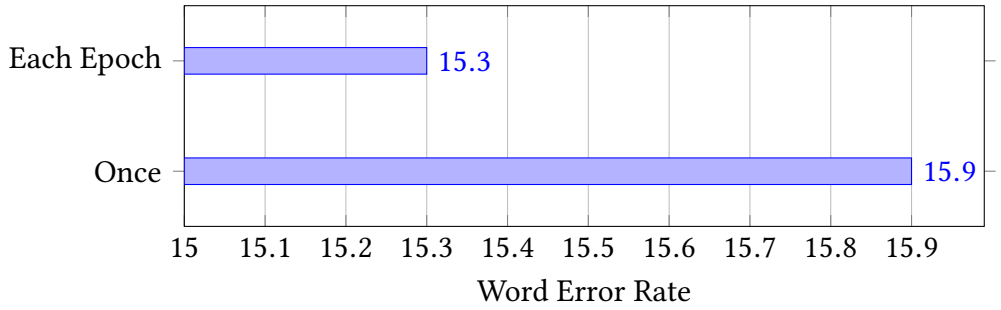

Figure 4.3.: Word error rate for different shuffling strategies

We can see that shuffling the training dataset before each epoch decreased the word error rate.

4.3.2. Learning Rate and Learning Rate Decay

As in [49], we utilize the newbob learning rate scheduler for stochastic gradient descend training. We used an initial learning rate of 0.08 and a momentum of 5. The SGD variant[1] used is the variant introduced in section 3.1.1.4. This variant was also used in [49].

Figure 4.4 shows the word error rate per epoch when using newbob. It can be seen that there were almost no improvements during epoch four, but the word error rate improved as soon as the decaying started after epoch four. Figure 4.5 shows the frame error rate per epoch for the same training run. It can be seen that exponential decay reduces the frame error rate significantly. The model used for this experiment was the four layer TDNN introduced in the previous section.

[1]The SGD variant is not equal to the default SGD variant implemented in PyTorch.

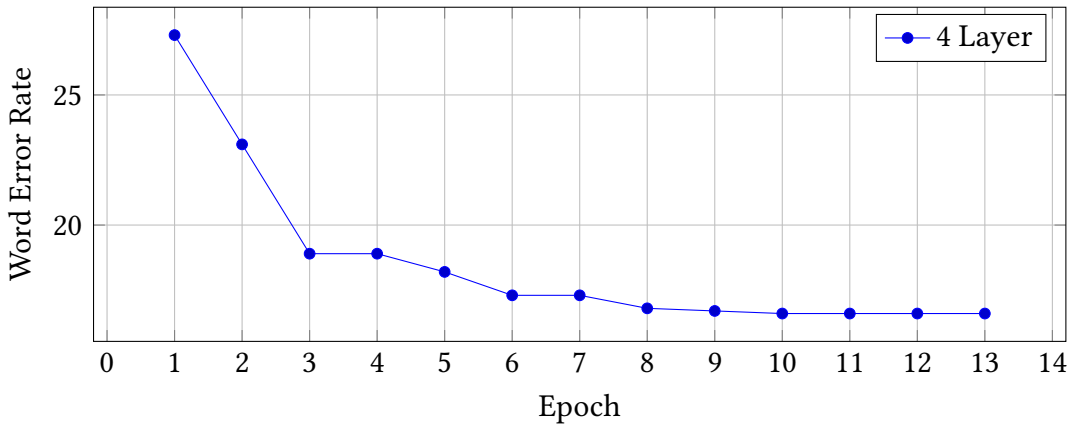

Figure 4.4.: Word error rate per epoch when using newbob training. The exponential decaying of the learning rate started after epoch four.

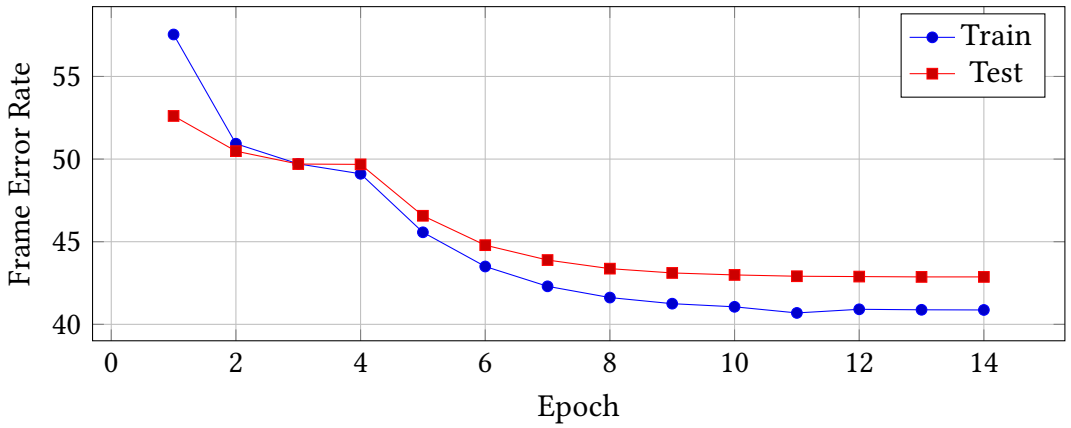

Figure 4.5.: Frame error rate per epoch when using newbob training. The exponential decaying of the learning rate started after epoch four.

4.3.3. MMIE Training

Since the experiments described in [30] show improvement when sMBR discriminative training is used, we attempted to use discriminative training as well. Our implementation used maximum mutual information estimation (MMIE), as described in section 3.4.2. We picked this training variant as a first step, since it is easier to implement than any variant of overall risk criterion estimation. Both approaches should show some improvement over cross entropy loss on frame level, according to several bodies of work [44] [42].

For MMIE training, we pre-trained a four layer TDNN acoustic model with frame-based cross entropy loss for a single epoch. Then, we started MMIE training on a per-utterance basis. For this purpose, we wrote a module that enabled interoperability between the Janus recognition toolkit and PyTorch. The training was done on multiple machines with a total

of 256 processors, the gradients were averaged before each SGD step.

While our experiments consistently showed high improvements in terms of frame error rate, the word error rate increased significantly: The model reached a WER of 17.6 using cross entropy loss, but only a WER 25.6 was reached after the MMIE training finished the first epoch. This effect was also described in more practically oriented literature [56] for MMIE without any further modifications.

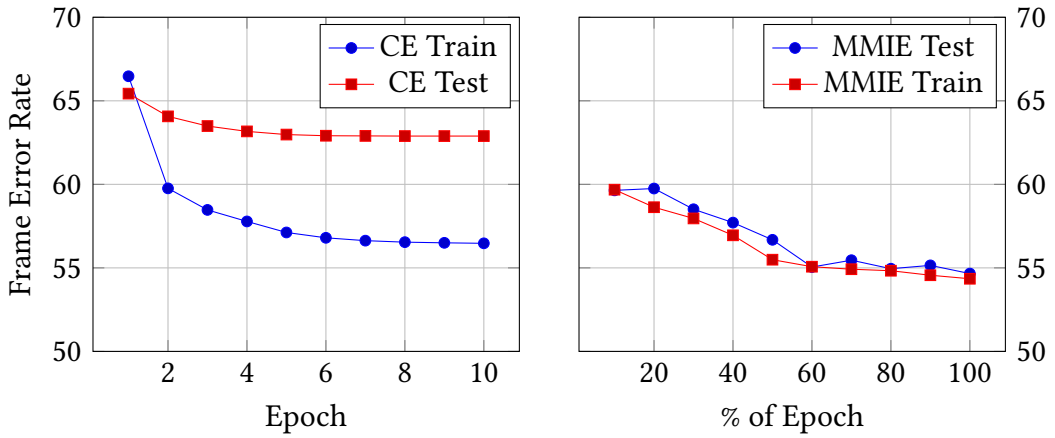

Figure 4.6.: Frame error rate per Epoch when using cross entropy loss, as well as frame error rate over a single epoch when using MMIE on a TDNN

Detailed results regarding the frame error rate are shown in figure 4.6. It can be seen that the MMIE training converged significantly faster than cross entropy training. Also, the error on the test data set is significantly closer to the error on the training data set.
These results were achieved using the four-layer TDNN model. Caution has to be taken when comparing the results: While the amount of data in both test data sets is the same, they are not equal as training and testing for MMIE happens on per-utterance basis, while the test and training sets for the cross entropy loss training were created on per-frame basis.

Although the results regarding frame error rate look interesting, we did not pursue this approach any further, due to the resulting high word-error rates and the numerous details one has to consider for working MMIE training on neural networks [56].

4.4. Decoding Parameters

Tuning decoding parameters is important for achieving a high accuracy on word level. The decoding parameters are not directly related to the acoustic model, but rather the decoding process. Different acoustic models might still require different decoding parameters for best performance.

4.4.1. Acoustic Model Scaling and Length Penalty

Our experiments have shown that the l_p and l_z parameters are depending on each other. Therefore we can not optimize them separately. We choose to perform a grid search over a reasonable parameter space.

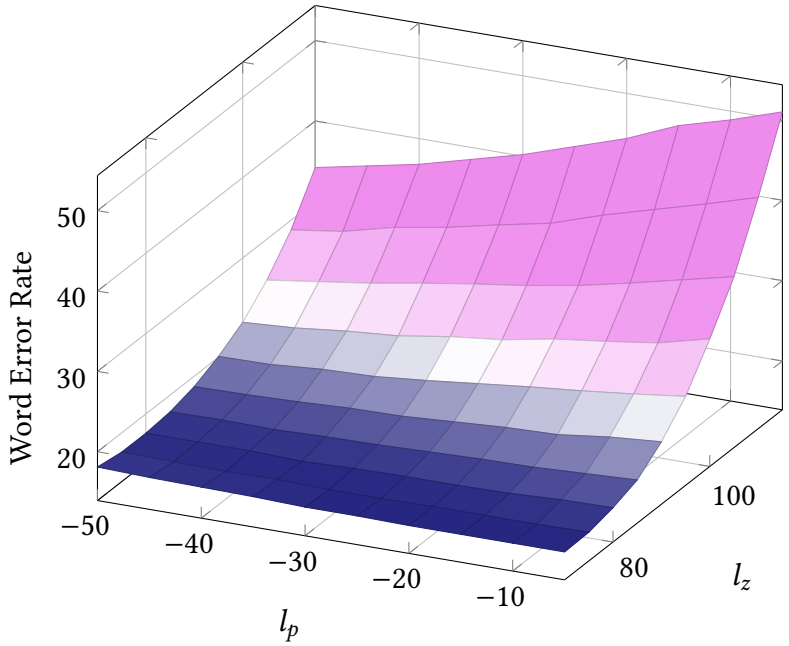

Figure 4.7.: Illustrative example of the word error rate for different l_p and l_z parameters for a four-layer TDNN

An example of the word error rate for different l_p and l_z for the four-layer TDNN model is illustrated in figure 4.7. Our experiments have shown that the optimal parameters are similar for each of the models we tested. It is still advisable to fine tune the parameters for each model. Taking the initial values for a small grid search from a similar model usually leads to good results.

4.4.2. Master Beam

We tested several architectures with different master beams. Master beams between four and six appeared to work best, but we did not find any pattern that correlates with the network architecture. This indicates that the optimal master beam depends on several factors, not just on the architecture of the neural network model itself.

4.4.3. Neural Network Priors

As described in section 3.1.4, it is important to scale the posteriors generated by the neural network with priors. We tested two different approaches of generating the priors: Generating them from the complete test dataset and also generating them from the output of a trained model. In the second case, we selected 15 hours of speech data randomly from our dataset, computed the neural network output on it, and counted the occurrence of each

label in the output.

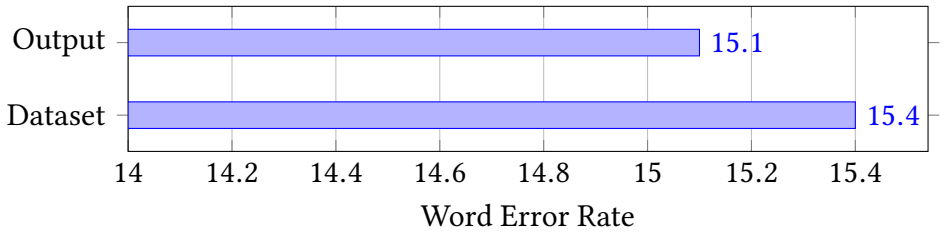

Figure 4.8.: Word error rate for priors estimated from the dataset and the model output

As illustrated in figure 4.8, calculating the priors from the output of the model decreased the word error rate significantly. These experiments were done on the four-layered TDNN model.

4.4.4. Softmax Smoothing

We also tested the impact of softmax smoothing on the neural network output. The motivation is that a beam search through a hidden Markov model does not work well when the acoustic model is overconfident regarding certain states. Figure 4.9 shows that softmax smoothing decreased the word error rate for our four-layer TDNN model, when carefully tuned.

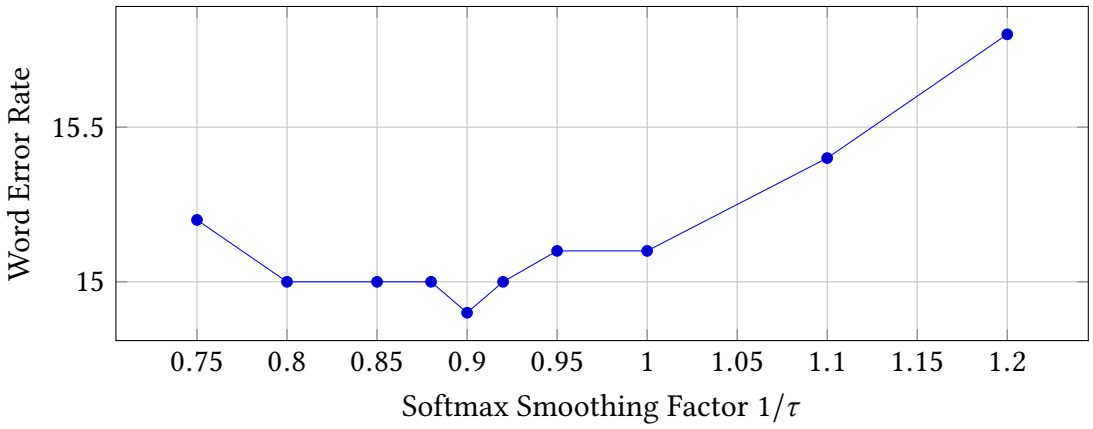

Figure 4.9.: Word error rate for different softmax adjustments $1/\tau$ for a four-layer TDNN model

5. Evaluation on Reverberated Data

This section contains the final evaluation. It describes how our model compares to a fully connected baseline model. The baseline model was tested with different input contexts. The purpose is to rule out that our model performed better only because of the size of the input context. The speech recognition system and preprocessing used for this evaluation is described in detail in chapter 4.

5.1. Neural Network Models

We compare the final TDNN model with a fully connected baseline model that was also used in [49]. This section describes these two models.

5.1.1. Fully Connected Baseline Model

We compare the results of our TDNN with a fully connected network that achieved comparable performance on the development set. The model consists of six linear layers with a width of 1600, followed by ReLU nonlinearities. The output layer is a linear layer followed by a softmax nonlinearity. We tested input contexts of $(-13, 9)$ as well as $(-5, 5)$.

5.1.2. TDNN Model

The TDNN model, which can be seen in figure 5.1, is based on the results documented in chapter 4. We tuned the hyperparameters so that the word error rate was minimal on the development set.

The model consists of four TDNN layers and one linear layer at the end, followed by a softmax nonlinearity. After each TDNN layer, a L2 pooling nonlinearity with a pool size of ten is used. As described in section 4.2.3, we set the gradient to zero if all inputs to the L2 pooling layer are zero. A splicing layer with the splicing configuration $(0, 1, 2, 3, 3, 4, 5, 6)$ is inserted after the first TDNN layer. The exact configuration of kernel sizes and strides can be found in table 4.1. The output of each TDNN layer has 3000 channels, the output of each pooling layer has 300 channels. The input context is $(-13, 9)$. The count of channels, the modified L2 pooling gradient and the omission of batch normalization was the main difference to the architecture described in [41] and [30].

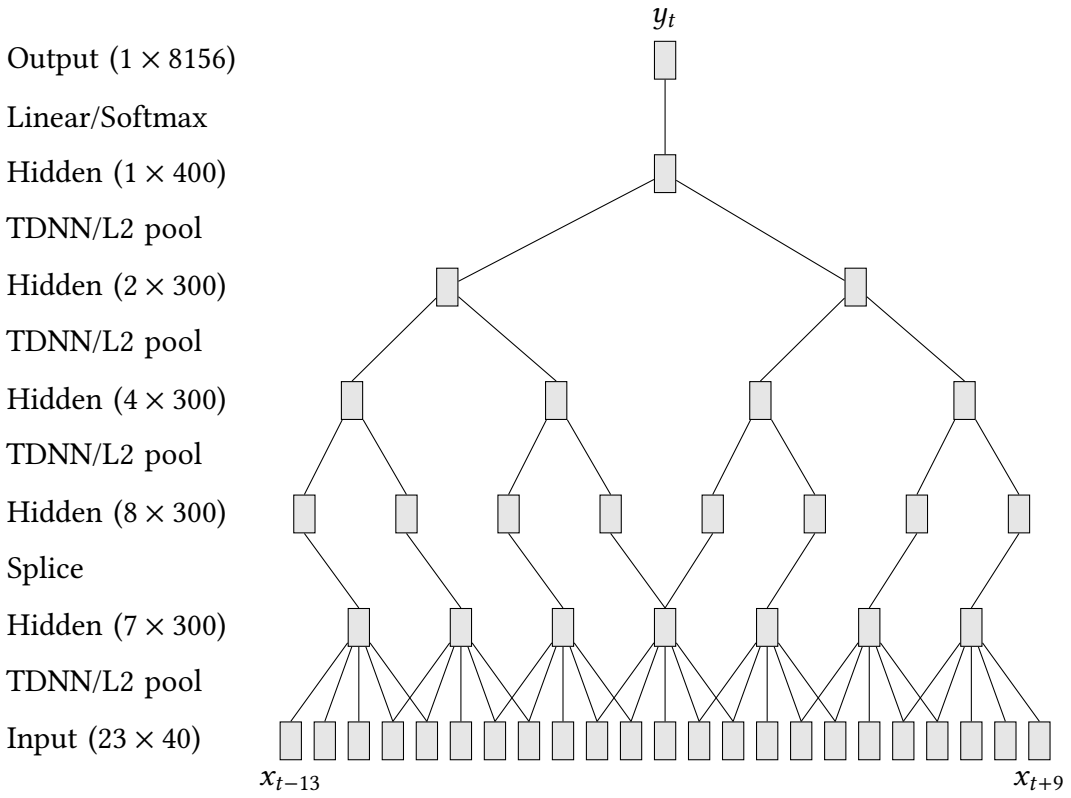

Figure 5.1.: Illustration of the final TDNN model in the time domain

5.2. Data Augmentation

For creating acoustic models that are robust against reverberation, we train them on a combination of clear and reverberated data. To create the reverberated data, we use a collection of recorded room impulse responses, following the theoretical insight given in section 2.1: For each audio sequence in our data set, we pick a random room impulse response and convolute the two signals to from a reverberated signal.

The room impulse responses we used are similar as in [7]. However we did only use the RWCP [57], OMNI [58] and ACE [59] datasets. The AIR dataset [8] was not used, since different recordings in the dataset vary significantly which made the dataset hard to normalize.

Evaluating the effects of different signal amplitudes was not a goal of this work, as we can assume that the audio frontend will always provide a reasonable gain. We therefore normalized the room impulse responses based on their signal energy before performing the convolution. For the ACE dataset, we found that the energy direct transmission path was very low compared to the early and late reverberations. In this case, we amplified the

direct transmission path to generate a usable result. Because of this normalization, each reverberated signal still had the same volume as the corresponding clean signal. Figure 2.2 shows an audio sample that was reverberated with this method.

5.3. Training Setup for Reverberated Data

We trained each model on the clean training set as described in chapter 4. We separately trained each model on a combination of the clean and reverberated training set, with a total length of 902 hours. The reverberated dataset was created by applying the previously described data augmentation to the complete clean dataset. We used the SGD variant from section 3.1.1.4 with newbob learning rate scheduling and momentum. The loss function was frame based cross entropy loss. The input features at each time frame were 40 log-mel coefficients as in [49]. We mean normalized our input features over the whole utterance with a resulting mean of zero and a resulting variance of two. This is different from the unnormalized 140 dimensional input vector used by [41] and [30] for their TDNN model.

Then, we tuned the l_p, l_z, master beam and softmax temperature parameters using the development set mentioned in chapter 4 for each model. The development set was not augmented. We calculate the priors by randomly sampling 15 hours form the clean data set and counting the labels produced by the model, given the randomly sampled data as input.

Since was impossible to fit the combined 902 hour dataset (54 GB) entirely into memory, and shuffled loading from disk was very slow with the large input context of $(-13, 9)$, we reduced the precision of the training dataset to 16 bits for all experiments in this section. We validated this approach by training a fully connected model on the clean dataset with the original precision and on the clean dataset with reduced precision. The difference in terms of frame error rate was only 0.02%[1], where the model trained on original precision was better. In terms of WER we did not observe any difference on the development set. On a reverberated validation set, the model trained on reduced precision achieved a slightly better word error rate. The reduced precision of 16 bit enabled us to hold the entire dataset in memory. With this approach, we were able to reach an average GPU utilization of 98%.

The TDNN model, which has 4.2 million parameters, trained for 14 epochs, where each epoch took 9.3 hours on the combined dataset on a single GTX 1080 Ti GPU. On the clean dataset, a single training epoch took 4.2 hours. The fully connected models with short and long input contexts had 13.2 million and 13.6 million parameters respectively. They trained for 12 and 11 epochs, where each epoch took 1.3 hours on the clean dataset and 2.3 hours on the combined dataset.

[1] 1202 frames out of 6013346 on the test dataset.

5.4. Validation Results

We validate the four-layer TDNN model with the input context $(-13, 9)$, as well as the fully connected (*FC*) model with the input contexts $(-13, 9)$ and $(-5, 5)$, trained on clean and augmented data respectively.
For the validation, we use the *tst2014* dataset, which was the validation dataset for the IWSLT 2014 conference [60]. Similar to our development dataset, the validation dataset also consists of TED talks. It has a total length of 2.1 hours. We test our models with a clean and a reverberated version of this dataset.

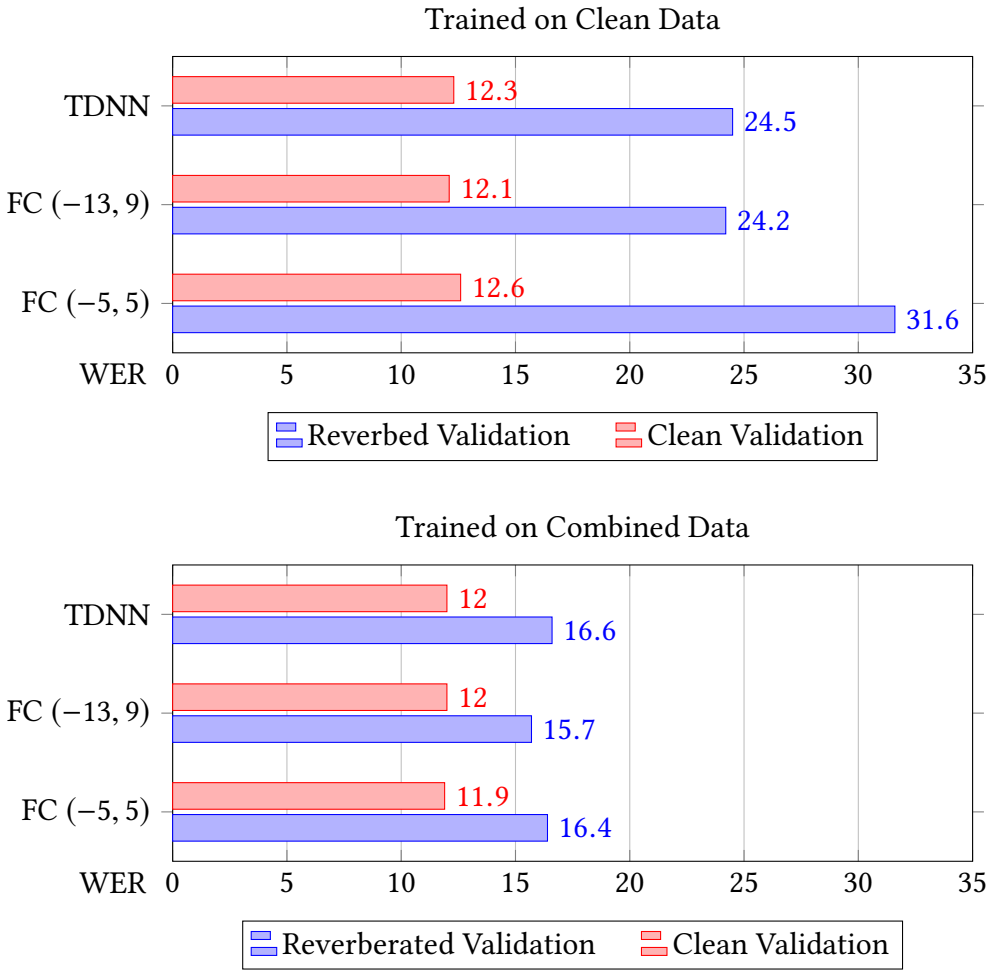

Figure 5.2.: Word error rate on the clean and reverberated validation dataset for models trained on clean and combined training data, respectively

The results of the validation can be seen in figure 5.2. For models trained on clean data, the word error rate on the reverberated validation set is high. It can be seen that models with larger input context performed better on unseen reverberated data. On the clean validation set, all models performed similar.

For models trained on reverberated data, the performance on the reverberated valida-

tion set is similar for all three models. The same holds for clean validation data. The fully connected model with a large input context outperformed the other models on the reverberated validation set by a small margin. All models performed slightly better on clean validation data than their counterpart that was trained on clean data only, where the difference was most significant for the fully connected model with short input context.

From this observations we conclude that data augmentation can be sufficient for improving acoustic model performance for reverberated audio. A larger input context can improve the robustness of the acoustic model in some cases.

6. Conclusion

During our evaluation in chapter 5 we found that our TDNN model did not yield a significant improvement over a fully connected model when trained on reverberated data. We also found that a fully connected model with the same input context was capable of slightly outperforming our TDNN model on the reverberated validation set. This results are difficult to generalize to TDNNs as a whole for the following reasons:

- In chapter 4, we tuned our TDNN on clean data, with the assumption that a TDNN model that performs well on clean data also performs well on reverberated data.
- The room impulse response normalization given in 5 might have been too aggressive, thus negating the advantage of TDNNs. In general, it is hard to measure the comprehensibleness of reverberated audio objectively.
- In literature [30] [41], sMBR training criteria are used for TDNN acoustic model training. It might be worth to investigate this training procedure more closely when working with reverberated data.

While the question whether TDNNs or fully connected networks are better for reverberation robust acoustic modeling is still unanswered, we provide several insights that can improve speech recognition systems:

- The modified L2 pooling nonlinearity introduced in section 4.2.3 performed better than L2 pooling combined with normalization.
- Augmented data can be used to boost the performance of acoustic models, even when only clean audio is of concern, as shown in chapter 5.
- The calculation of priors from the network output, as shown in section 4.4.3, given a randomly sampled subset of the training data, was shown to outperform priors which were calculated from the training data set.
- We also provided a mathematically sound differentiation of discriminative training criteria for neural networks in section 3.4, which is not available in literature in this form.

Overall, we were able to show that acoustic models can be made robust against reverberation, if they are trained using reverberated data as well, especially when the input context is large enough.

Bibliography

[1] Dong Yu and Li Deng. *AUTOMATIC SPEECH RECOGNITION*. Springer, 2016.

[2] Gustavo López, Luis Quesada, and Luis A Guerrero. "Alexa vs. Siri vs. Cortana vs. Google Assistant: a comparison of speech-based natural user interfaces". In: *International Conference on Applied Human Factors and Ergonomics*. Springer. 2017, pp. 241–250.

[3] Markus Müller et al. "Lecture Translator-Speech translation framework for simultaneous lecture translation". In: *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Demonstrations*. 2016, pp. 82–86.

[4] Sebastian Stüker et al. "The ISL 2007 english speech transcription system for european parliament speeches". In: *Eighth Annual Conference of the International Speech Communication Association*. 2007.

[5] Takuya Yoshioka et al. "Making machines understand us in reverberant rooms: Robustness against reverberation for automatic speech recognition". In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 114–126.

[6] Fernando Puente León and Holger Jäkel. *Signale und Systeme*. Walter de Gruyter GmbH & Co KG, 2015.

[7] Marvin Ritter et al. "Training Deep Neural Networks for Reverberation Robust Speech Recognition". In: *Speech Communication; 12. ITG Symposium; Proceedings of*. VDE. 2016, pp. 1–5.

[8] Marco Jeub, Magnus Schafer, and Peter Vary. "A binaural room impulse response database for the evaluation of dereverberation algorithms". In: *Digital Signal Processing, 2009 16th International Conference on*. IEEE. 2009, pp. 1–5.

[9] Puming Zhan and Alex Waibel. *Vocal tract length normalization for large vocabulary continuous speech recognition*. Tech. rep. Carnegie Mellon University, School of Computer Science, 1997.

[10] Steven B Davis and Paul Mermelstein. "Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences". In: *Readings in speech recognition*. Elsevier, 1990, pp. 65–74.

[11] Dennis Gabor. "Theory of communication. Part 1: The analysis of information". In: *Journal of the Institution of Electrical Engineers-Part III: Radio and Communication Engineering* 93.26 (1946), pp. 429–441.

[12] Jeff A Bilmes et al. "A gentle tutorial of the EM algorithm and its application to parameter estimation for Gaussian mixture and hidden Markov models". In: *International Computer Science Institute* 4.510 (1998), p. 126.

[13] Frank Rosenblatt. “The perceptron: a probabilistic model for information storage and organization in the brain.” In: *Psychological review* 65.6 (1958), p. 386.

[14] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[15] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. “Multilayer feedforward networks are universal approximators”. In: *Neural networks* 2.5 (1989), pp. 359–366.

[16] PJ Werbos. “Beyond regression: New tools for prediction and analysis in the behavioral sciences. Ph. D. thesis, Harvard University, Cambridge, MA, 1974.” In: (1974).

[17] D Randall Wilson and Tony R Martinez. “The general inefficiency of batch training for gradient descent learning”. In: *Neural Networks* 16.10 (2003), pp. 1429–1451.

[18] Léon Bottou. “Stochastic gradient learning in neural networks”. In: *Proceedings of Neuro-Nımes* 91.8 (1991), p. 0.

[19] Dan Ellis. *ICSI Speech FAQ: 6.3 How are neural nets trained?* 2000. URL: `http://www1.icsi.berkeley.edu/Speech/faq/nn-train.html` (visited on 04/10/2018).

[20] Boris T Polyak. “Some methods of speeding up the convergence of iteration methods”. In: *USSR Computational Mathematics and Mathematical Physics* 4.5 (1964), pp. 1–17.

[21] Ilya Sutskever et al. “On the importance of initialization and momentum in deep learning”. In: *International conference on machine learning*. 2013, pp. 1139–1147.

[22] Martin Thoma. “Analysis and Optimization of Convolutional Neural Network Architectures”. In: *arXiv preprint arXiv:1707.09725* (2017).

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[24] Xiaohui Zhang et al. “Improving deep neural network acoustic models using generalized maxout networks”. In: *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*. IEEE. 2014, pp. 215–219.

[25] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *CoRR* abs/1502.03167 (2015). arXiv: `1502.03167`. URL: `http://arxiv.org/abs/1502.03167`.

[26] Solomon Kullback and Richard A Leibler. “On information and sufficiency”. In: *The annals of mathematical statistics* 22.1 (1951), pp. 79–86.

[27] Michael D Richard and Richard P Lippmann. “Neural network classifiers estimate Bayesian a posteriori probabilities”. In: *Neural computation* 3.4 (1991), pp. 461–483.

[28] Alexander Waibel et al. “Phoneme recognition using time-delay neural networks”. In: *Readings in speech recognition*. Elsevier, 1990, pp. 393–404.

[29] Jost Tobias Springenberg et al. “Striving for simplicity: The all convolutional net”. In: *arXiv preprint arXiv:1412.6806* (2014).

[30] Vijayaditya Peddinti et al. “Jhu aspire system: Robust lvcsr with tdnns, ivector adaptation and rnn-lms”. In: *Automatic Speech Recognition and Understanding (ASRU), 2015 IEEE Workshop on*. IEEE. 2015, pp. 539–546.

[31] Ernst Günter Schukat-Talamazzini. *Automatische Spracherkennung*. Vieweg, Wiesbaden, 1995.

[32] Xuedong Huang et al. *Spoken language processing: A guide to theory, algorithm, and system development*. Vol. 95. Prentice hall PTR Upper Saddle River, 2001.

[33] Stéphane Mallat. *A wavelet tour of signal processing*. Academic press, 1999.

[34] Alex Waibel and B Yegnanarayana. “Comparative study of nonlinear time warping techniques in isolated word speech recognition systems”. In: *IEEE transactions on acoustics, speech, and signal processing* 31.6 (1983), pp. 1582–1586.

[35] William J Poser. *Speech Communication: Human and Machine*. 1990.

[36] Tomáš Mikolov et al. “Extensions of recurrent neural network language model”. In: *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*. IEEE. 2011, pp. 5528–5531.

[37] Kris Demuynck, Dirk Van Compernolle, and Patrick Wambacq. “Doing away with the Viterbi approximation”. In: *Acoustics, Speech, and Signal Processing (ICASSP), 2002 IEEE International Conference on*. Vol. 1. IEEE. 2002, pp. I–717.

[38] Peter E Hart, Nils J Nilsson, and Bertram Raphael. “A formal basis for the heuristic determination of minimum cost paths”. In: *IEEE transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107.

[39] Geoffrey Hinton et al. “Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups”. In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 82–97.

[40] Yoshua Bengio. “A connectionist approach to speech recognition”. In: *Advances in Pattern Recognition Systems Using Neural Network Technologies*. World Scientific, 1993, pp. 3–23.

[41] Vijayaditya Peddinti et al. “Reverberation robust acoustic modeling using i-vectors with time delay neural networks”. In: *Sixteenth Annual Conference of the International Speech Communication Association*. 2015.

[42] Arnab Ghoshal, Daniel Povey, et al. “Sequence-discriminative training of deep neural networks”. In: *Proceedings of INTERSPEECH*. 2013.

[43] Matthew Gibson. “Minimum Bayes risk acoustic model estimation and adaptation”. PhD thesis. University of Sheffield, Department of Computer Science, 2008.

[44] Daniel Povey. “Discriminative training for large vocabulary speech recognition”. PhD thesis. University of Cambridge, 2005.

[45] Lalit Bahl et al. “Maximum mutual information estimation of hidden Markov model parameters for speech recognition”. In: *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP’86*. Vol. 11. IEEE. 1986, pp. 49–52.

[46] Ralf Schluter and Wolfgang Macherey. "Comparison of discriminative training criteria". In: *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*. Vol. 1. IEEE. 1998, pp. 493–496.

[47] Janez Kaiser, Bogomir Horvat, and Zdravko Kacic. "A novel loss function for the overall risk criterion based discriminative training of HMM models". In: *Sixth International Conference on Spoken Language Processing*. 2000.

[48] Daniel Povey and Philip C Woodland. "Minimum phone error and I-smoothing for improved discriminative training". In: *Acoustics, Speech, and Signal Processing (ICASSP), 2002 IEEE International Conference on*. Vol. 1. IEEE. 2002, pp. I–105.

[49] Thai-Son Nguyen et al. "The 2016 KIT IWSLT speech-to-text systems for English and German". In: *Proceedings of the ninth International Workshop on Spoken Language Translation (IWSLT), Seattle, WA*. 2016.

[50] Michael Finke et al. "The Karlsruhe-verbmobil speech recognition engine". In: *Acoustics, Speech, and Signal Processing, 1997. ICASSP-97., 1997 IEEE International Conference on*. Vol. 1. IEEE. 1997, pp. 83–86.

[51] Kevin Lenzo. *The CMU Pronouncing Dictionary*. 2017. URL: `http://www.speech.cs.cmu.edu/cgi-bin/cmudict` (visited on 04/22/2018).

[52] Anthony Rousseau, Paul Deléglise, and Yannick Estève. "Enhancing the TED-LIUM Corpus with Selected Data for Language Modeling and More TED Talks." In: *LREC*. 2014, pp. 3935–3939.

[53] David Graff et al. "The 1996 broadcast news speech and language-model corpus". In: *Proceedings of the DARPA Workshop on Spoken Language technology*. 1997, pp. 11–14.

[54] Mauro Cettolo et al. "Report on the 10th IWSLT evaluation campaign". In: *Proceedings of the International Workshop on Spoken Language Translation, Heidelberg, Germany*. 2013.

[55] Adam Paszke et al. "Automatic differentiation in PyTorch". In: (2017).

[56] Hang Su et al. "Error back propagation for sequence training of context-dependent deep networks for conversational speech transcription". In: *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE. 2013, pp. 6664–6668.

[57] Satoshi Nakamura et al. "Acoustical Sound Database in Real Environments for Sound Scene Understanding and Hands-Free Speech Recognition." In: *LREC*. 2000.

[58] Rebecca Stewart and Mark Sandler. "Database of omnidirectional and B-format room impulse responses". In: *Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on*. IEEE. 2010, pp. 165–168.

[59] James Eaton et al. "The ACE challenge—Corpus description and performance evaluation". In: *Applications of Signal Processing to Audio and Acoustics (WASPAA), 2015 IEEE Workshop on*. IEEE. 2015, pp. 1–5.

[60] Mauro Cettolo et al. "Report on the 11th IWSLT evaluation campaign, IWSLT 2014". In: *Proceedings of the International Workshop on Spoken Language Translation, Hanoi, Vietnam*. 2014.

A. Appendix

A.1. Optimal Decoder Parameters

The following table contains a summary of all decoder parameters we found to be optimal for the experiments in our evaluation. This might be useful for reproducing our results or for further experiments. The table also shows the achieved word error rate on the clean and reverberated development data set.

Model Name	Training Data	l_p	l_z	mb	$1/\tau$	clean WER	rvb WER
TDNN $(-13, 9)$	clean	-15	95	6	0.9	14.9	27.6
FC $(-13, 9)$	clean	-10	90	6	0.8	15.1	28.4
FC $(-5, 5)$	clean	-10	90	6	0.8	15.3	35.5
TDNN $(-13, 9)$	clean + rvb	-5	95	6	0.85	14.8	20.4
FC $(-13, 9)$	clean + rvb	-10	90	6	0.78	14.7	19.8
FC $(-5, 5)$	clean + rvb	-10	95	6	1.0	14.6	20.3

Table A.1.: Optimal decoder parameters and word error rate on clean and reverberated development data set

It should be noted that the parameters shown here are implementation specific to the Janus recognition toolkit [50].