Karlsruhe Institute of Technology

# Optimization of DNN Acoustic Models for Low Resource and Mobile Environments

Master's thesis
submitted by

**Alexander Tu**

at the Department of Informatics
Institute for Anthropomatics and Robotics

| | |
|---|---|
| Reviewers: | Prof. Dr. Alexander Waibel |
| | Prof. Dr. Tamim Asfour |
| Advisor: | Dr. Sebastian Stüker |

Process Period: November 14th, 2018 – May 13th, 2019

I hereby declare that this document has been composed by myself and describes my own work unless otherwise acknowledged in the text.

Karlsruhe, May 13th, 2019

# Abstract

Deep neural networks have shown to excel in many speech recognition tasks thanks to their deep and wide network structures as well as the large number of parameters. These big models have excellent performance, but are slow, computationally demanding and require strong hardware. This is an issue for systems with limited memory or computational power like mobile devices.

This thesis deals with the optimization of hybrid DNN/HMM acoustic models for speech recognition. The focus of this work is the exploration of different ways to reduce a model's footprint to make it more viable for devices with less computational power. We are following three different approaches. Applying vector quantization to training data and parameters of the neural network, training the model while simulating the usage of 4-bit/8-bit/16-bit floating point numbers instead of 32-bit, and reducing the overall model size by pruning the nodes of the network.

We can show that two of the explored methods achieve a notable reduction in model size while keeping the recognition performance almost on par with the baseline system. Quantizing the model parameters to lower precision floats show a small increase of the error rate by 0.2% to 10.1% with a reduction of the model size by a factor of up to 4. We were able to shrink the model by 37% while keeping the error rate at 10.2% by systematically removing nodes in the hidden layers. Smaller models are possible by removing more nodes but are accompanied by increasing error rates.

Combined approaches were explored and achieved a word error rate of 9.8% in the best case. If it weren't for the simulation it could be possible to reduce the model footprint by up to 85% achieving a size of 13mb for the neural network while running. Smaller models achieved a word error rate of 10.3% while running 20% faster and having a size 90% smaller than the baseline.

# Zusammenfassung

Neuronale Netze haben sich bei vielen Aufgaben der Spracherkennung durch ihre tiefen und breiten Netzstrukturen sowie die Vielzahl der Parameter bewährt. Diese großen Modelle überzeugen mit einer ausgezeichneten Leistung, sind jedoch langsam, rechenintensiv und erfordern starke Hardware. Dies führt dazu, dass diese Modelle nur bedingt geeignet sind für Anwendungen auf Endgeräten mit begrenztem Speicher oder Rechenleistung wie Handys oder Smartphones.

Diese Arbeit beschäftigt sich mit der Optimierung hybrider DNN/HMM Akustikmodelle für Spracherkennung. Der Schwerpunkt dieser Arbeit liegt auf der Erkundung verschiedener Möglichkeiten den Fußabdruck eines Modells zu verkleinern, um es für Geräte mit weniger Rechenleistung zugänglicher zu machen. Wir verfolgen drei verschiedene Ansätze. Vektor Quantisierung von Trainingsdaten und Parametern des neuronalen Netzwerks, Training neuronaler Netze bei dem wir die Verwendung von 8-bit bzw. 16-bit Fließkommazahlen anstelle von 32-bit für die Modellparameter simulieren, und Reduzierung der Gesamtmodellgröße durch Veränderung der Topologie des Netzwerks.

Wir können zeigen, dass zwei der angewendeten Methoden die Größe der trainierten Modelle deutlich verringern kann, bei einer geringen Verschlechterung der Erkennungsleistung. Versuche mit 16-bit und 8-bit Fließkommazahlen ergeben einen kaum bemerkbaren Anstieg der Fehlerrate um 0,1%, bei einer Verkleinerung des Modells um einen Faktor von bis zu 4. Durch das systematische Entfernen von Modellknoten waren wir in der Lage Platzeinsparungen von 37%, bei einer Fehlerrate von nur 10,2%, zu erzielen. Durch das Entfernen weitere Knoten sind kleinere Modelle möglich, bringen jedoch eine höhere Fehlerrate mit sich.

Kombinierte Lösungen wurden untersucht und erreichen im besten Szenario eine Wortfehlerrate von 9.8%. Würde wir statt der Simulation tatsächlich 8-bit floats verwenden wäre eine potentielle Reduzierung der Modellgröße um 85% auf 13mb möglich. Mit kleineren Modelle erreichen wir eine Wortfehlerrate von 10.3%, die 90% kleiner als das Ursprungssystem sind und 20% schneller sind.

# Table of Content

# List of Figures

# List of Tables

# 1. Introduction

## 1.1 Motivation

Smartphones and tablets play a big role in our everyday lives and have started to replace laptops and desktop computers as our main computing device. Despite the fact that on-screen keyboards are much harder to input text than their physical counterparts, mobile devices became the main tool to access the Internet, read the news, write messages or interact with people on social media . Besides typing most devices offer their users the possibility to use their voice to dictate texts. Speech is natural and one of the fastest modality with a high bandwidth of 1000-4000 characters per minute. Apple iOS devices are shipped with Siri and Google's Android phones offer the services of Google Assistant [SBBB+10]. Furthermore users are able to do simple tasks ranging from voice search to playing music and calling someone, to controlling lights or home entertainment systems with their voice. Speech recognition and other speech applications make their ways into our lives and modern technologies make them widely available for everyone.

A big problem of those applications is that the work is done on large servers with heavy computational hardware to process the language. Therefore to be able to use the applications to their full extent, a constant or stable Internet connection is required. Mobile connections are often slow or limited sometimes even non existent causing high latency or complete failure. The need of an Internet connection brings along another issue. With the growing importance people put on their privacy, many of them do not agree to have their data uploaded to third party servers.

Those problems would not exist if the speech recognition systems were small enough. Smaller systems offer offline capabilities by allowing them to run locally on the device. The performance would not depend on the state of the Internet connection therefore it would run more reliably as well as possibly having a lower latency. The upload of the spoken data to a server would not be necessary anymore also. Unfortunately smaller systems often perform worse than their full size counterparts. To create a usable offline system it needs to maintain a high accuracy while being small, not bearing too much load on the available resources or consume too much memory while running.

With the ultimate objective in mind to shrink down an automatic speech recognition system so that it could potentially run locally on an embedded device, the goal of this thesis is to explore different approaches to optimize a DNN acoustic model built with a feed forward neural network to reduce its footprint while maintaining a reasonable accuracy.

## 1.2 Overview

Chapter 2 provides all relevant background informations as well as theoretical explanations and definitions of neural networks and modern automatic speech recognition systems. Afterwards we give a short introduction to acoustic modelling using neural networks. Then we talk about the k-Means algorithm for vector quantization. The chapter is concluded with a short explanation of the floating point number format and the IEEE 754 standard. Chapter 3 discusses various related works and publications. Chapter 4 outlines the experimental setup introducing the frameworks used, the structure of the training and evaluation data as well as the overall training setup. In chapter 5, 6 and 7 we describe the performed experiments as well as the evaluation of the outcome. Finally in the last chapter we summarize our work, discuss our insights and give an outlook on future work.

# 2. Background

This chapter provides a quick overview of artificial neural networks. As there are many different types of networks and machine learning algorithms, we will only focus on the ones relevant for this work. After that it explains the basic concepts of modern automatic speech recognition as well as the usage of neural networks for acoustic modeling. Furthermore we introduce the k-Means Algorithm, an algorithm used for quantization of data. In the last section we talk about the floating point number format and how it is defined by the IEEE 754 standard.

## 2.1 Neural Networks

Artificial Neural Networks (ANN) are computational models inspired by the biological neural network of the human brain. The idea behind it was to create a model that is capable of learning and adapting if necessary. ANNs have shown great success in many fields of machine learning like machine translation [DZHL+14a], image recognition [KrSH12], facial recognition [LADSOS17] and speech recognition [DeHK13].

This section provides a quick overview of the basic feed forward neural network as well as some techniques used for training. There are many other variations of neural networks that are not relevant for this work and therefore not mentioned.

### 2.1.1 Perceptron

The first research on artificial neural networks date back to the first mathematical definition of a neuron by McCulloch and Pitts in 1943 [McPi43]. They showed that such networks were able to calculate nearly any logic function. Going from there, Frank Rosenblatt developed the first operational model of a neural network, the so-called *perceptron* [Rose58].

The perceptron is a linear binary classifier which maps a $(n+1)$-dimensional input vector $x$ to a binary output value $y$. It is the basic unit of computation in a neural network. The schematics of such a neuron can be seen in Figure 2.1. The neuron consists of four parameters:

- $\mathbf{w}$: Weights of all inputs.

- **Bias** $b$: Shifts the activation function to the left or right.
- $\mathbf{\Sigma}$: Calculates the weighted sum of the input vector $x$.
- **Activation function** $\phi$: Non-linear function which calculates the output value $y$ depending on the weighted sum of the input.

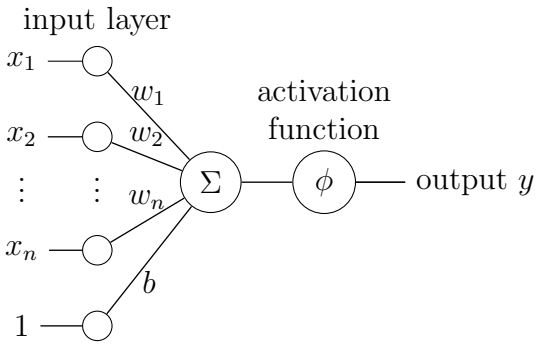


Figure 2.1: Rosenblatt Perceptron

The output of a neuron can be written as the following equation:

$$y = \phi(\sum_{i=0}^{n} w_i x_i + b) = \phi(\mathbf{w}^T \mathbf{x} + b) \tag{2.1}$$

The training of the perceptron is supervised. Given a training set $X = (x, t_x)$ where $t_x$ stands for the target output value of the input vector $\mathbf{x}$ the perceptron algorithm tries to find the optimal weights $\hat{w}$ by minimizing the Mean Squared Error

$$E_{\text{MSE}}(w) = \frac{1}{2} \sum_{x \in X} (t_x - y_x)^2 \tag{2.2}$$

In the first step the parameters are initialized usually with 0 or small random values. After specifying a learning rate $\eta \in (0, 1)$ and a threshold $\gamma$ the algorithm repeats the following steps for each $x \in X$ [FrSc99]:

1. Compute the current output $y_x$ of the network.
2. Update the weights $\mathbf{w} = w_i \leftarrow w_i + \Delta w_i$ with $\Delta w_i = -\eta \nabla E(w)$.
3. Terminate if $E(w)$ falls below threshold $\gamma$.

As long as the training set can be separated linearly the perceptron converges. The biggest weakness of a single perceptron is the inability to learn non-linear separable functions such as XOR. This weakness can be overcome by applying the same concept to larger networks with multiple layers of neurons called *multilayer perceptrons* (MLP) which open up the possibility of learning higher-order functions [RuNo09].

### 2.1.2 Feed Forward Neural Network

A Feed Forward Neural Network is the most basic form of a MLP. It consist of three types of layers: An input layer, one or more hidden layers, and an output layer. If the network consists of multiple hidden layers it is often referred to as a deep neural network. Each layer consists of multiple single neurons. The name of the network is derived from the fact that the information between the neurons only flow in one direction. There are no cyclic connections between the neurons [GoBC16]. An illustration of such a network can be seen in Figure 2.2. As seen in the figure the number of neurons does not have to be the same for each layer. For classification the size of the input layer is at least as large as the dimension of the input vector while the output size consists of as many neurons as there are classes in the problem.

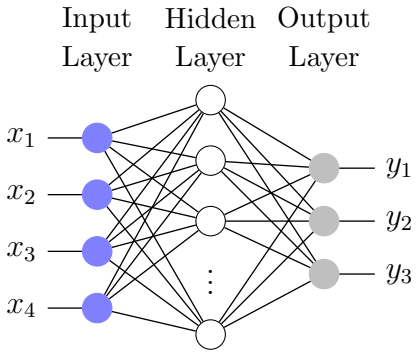


Figure 2.2: A feed forward neural network with an input layer of 4 neurons, a single hidden layer, and an output layer of 3 neurons

Each layer processes an input $x$ and calculates an output $y$. The output is then fed as the input for the next connected layer. A very important role in calculating each neuron's output is played by its activation function. They are very powerful as it was shown that non-linear activation functions used in a net with at least one hidden layer are capable of approximating any function [Horn91]. Some of the most commonly used functions are listed below and visualized in Figure 2.3.

The **sigmoid** function $\sigma(x)$ is a very common activation function. The output is between 0 and 1 which makes it possible to interpret the values as probabilities

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.3}$$

The **hyperbolic tangent** $tanh(x)$ maps the input to a number between -1 and 1.

$$\tanh(x) = 1 - \frac{2}{1 + e^{-2x}} \tag{2.4}$$

The **rectified linear unit** ReLU outputs all values bigger than 0. Negative values are set to 0. It has the strongest biological motivation and is the most successful and widely-used activation function [RaZL18].

$$\text{ReLU}(x) = max(0, x) \tag{2.5}$$

For classification problems with more than two possible outputs, the **softmax** function is used as the activation function for the last layer. It's a generalization of

the sigmoid function and normalizes the output values to a range between 0 and 1. All output values will sum up to 1 which allows them to be interpreted as class probabilities.

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{2.6}$$

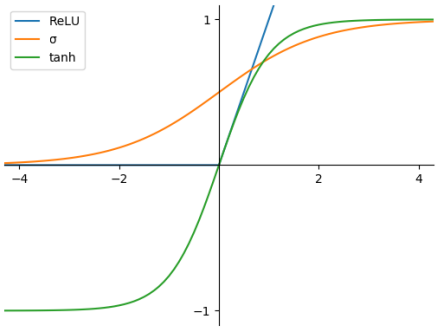


Figure 2.3: Graphical representation of the rectified linear unit, sigmoid and hyperbolic tangens activation functions

### 2.1.3 Training

Neural Networks are known for their ability to learn from data and improve their performance by adjusting their own parameters. Similar to the perceptron training, we try to minimize an error function also called a cost function. There are three main concepts for training [RuNo09]:

**Supervised training**
The train set is labeled and the optimal output is known during the training. It's easy to compare the actual output with the optimal output which makes the calculation of the cost function a simple task.

**Reinforcement learning**
The training data is not labeled. Instead of prior knowledge of the optimal output the environment is giving feedback on the outputs the system is giving. Good output is being rewarded, bad output discouraged. The net is trying to maximize the rewards received.

**Unsupervised training**
Neither knowledge about the training set nor the environment is known. The system is trying to find patterns and distributions that are not random noise. This training method is used popularly for clustering.

#### Backpropagation

The **backpropagation** algorithm is a supervised training algorithm and has shown great success in finding good sets of parameters for neural networks. The algorithm operates in two repeating steps.
First a randomly picked input sample $x$ is fed to the net and a prediction $y$ is

determined. After the forward pass the loss $l$ is calculated with the help of an error function and the desired target output $t$ .

$$l = E(y, t) \tag{2.7}$$

In most cases the mean squared error (see eq. 2.2) is sufficient but does not perform so well in the case of classification tasks. For such problems the cross entropy function is the loss function of choice

$$E_{\text{CE}} = -\sum_{i=1}^{k} t_i \log(y_i) + (1 - t_i) \log(1 - y_i) \tag{2.8}$$

In the second phase, we are going backwards through the net to calculate the error value for each single neuron to see its affect on the overall error. The weights are then updated to improve its prediction with respect to the loss by multiplying the gradients of the error functions with a learning rate and adding the result to the parameters. These two steps are repeated until the algorithm converges into a local or global minimum [Sche13].

**Stochastic Gradient Descent**

There are different ways to apply the backpropagation algorithm when training a neural network. The first option is applying the weight updates after each single training sample and is called **stochastic gradient descent**. While this may work, it can be slow to converge and noisy data might lead to inconsistent weight updates that don't reflect the overall data.
Adjusting the weights by calculating the errors on all training samples and updating the weights by the average error is called **batch gradient descent**. This approach is much more robust but suffers from very long calculation times on big datasets.

A hybrid of both approaches is called **mini batch gradient descent** and combines both of their benefits. The training set is divided into smaller batches, usually 64-1024 samples per batch. The idea behind it is that these batches have a similar distribution than the whole data set and are robust against noisy data points unlike single samples. The algorithm then iterates over each subset and calculates the weight update for each minibatch [RuNo09]. An iteration over all minibatches is called an epoch.

**Newbob Scheduling**

Newbob Scheduling is a scheduling algorithm to control the learning rate $\eta$ of neural networks during training. The learning rate plays an essential role in the training process so it is important to find the right value. Too high learning rates can cause good local minima to be skipped or the training not to converge. If the learning rate is too low the network takes longer to learn and it is more likely to end up in a bad local minimum.
The scheduler operates as follows:

The learning rate $\eta$ is kept constant as long as the reduction of the validation error between two epochs is bigger than a given threshold $\tau_1$ allowing for fast learning and skipping of local minima. For the subsequent epochs the learning rate is scaled by a factor $c$ until the change of the validation error between the current and last epoch falls below a second threshold $\tau_2$. This phase allows for fine-tuning the weights and guarantees convergence [VeBG10].

**Momentum**

Another technique for avoiding local minima is the usage of momentum. Instead of applying the gradient directly to our model we calculate a moving average of the gradient of the last batches and use that. There are different ways of achieving momentum for gradient descend. The one used in this work is equal to the implementation used in [Joeb18]. It introduces a velocity term $v$ which is a linear combination of the last velocity and the current gradient. The momentum is depicted as $\rho$ and acts as a scaling factor which regulates the amount of velocity that is applied to the gradient.

$$v_{i,t+1} = v_{i,t} * \rho + \eta * \frac{\partial E_t}{\partial \theta_{i,t}} \tag{2.9}$$

$$\theta_{i,t+1} = \theta_{i,t} - v_{i,t+1} \tag{2.10}$$

$\frac{\partial E_t}{\partial \theta_{i,t}}$ denotes the gradient of the loss function $E$ with respect to learnable parameters $\theta$ of the network.

## 2.2 Automatic Speech Recognition

Automatic Speech Recognition (ASR), also known as speech to text (STT) deals with the conversion of human speech signals into the corresponding word sequence in machine-processable form. High variability of the spoken language due to factors such as environmental noise, gender, moods or dialects of the speaker, word ambiguities and much more make the recognition task very complex.
Instead of functions that try to map from speech to text, signal processing techniques are used in combination with statistical models to find the most likely transcription. The basic architecture of an ASR system is shown in Figure 2.4.

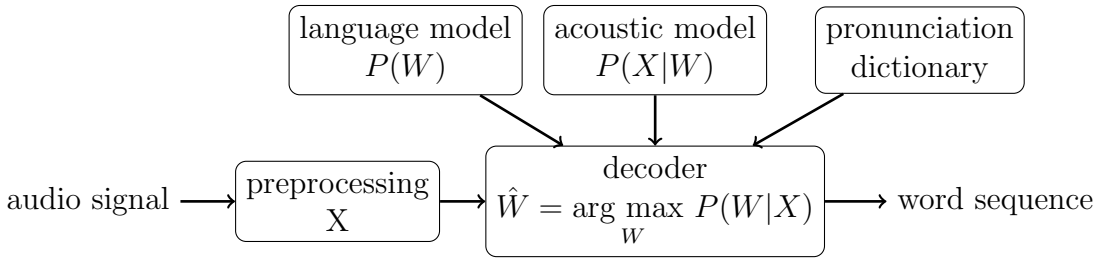


Figure 2.4: Diagram of a speech recognition system

First the audio signal is recorded using a microphone. The analogue signal is then sampled in time and amplitude resulting in a discrete digital signal. The preprocessing extracts prominent features and transforms the signal into a sequence of feature vectors $X = X_1X_2...X_n$.
Our goal is to search the space of possible sentences to find the best matched word sequence $\hat{W}$ with the highest probability given the features $X$. In other words we try to find the word sequence which maximizes $P(W|X)$. This can be formulated as

$$\hat{W} = \arg\max_W P(W|X) \tag{2.11}$$

Applying Bayes rule [HuAH01] the equation can be rewritten to

$$\hat{W} = \arg\max_{W} P(W|X) \tag{2.12}$$

$$= \arg\max_{W} \frac{P(X|W)P(W)}{P(X)} \tag{2.13}$$

$$= \arg\max_{W} P(X|W)P(W) \tag{2.14}$$

Equation 2.14 is also known as the fundamental formula of speech recognition. $P(X|W)$ is called the *acoustic model* and represents the probability of seeing the sequence of feature vector $X$ given a hypothesized sequence of words $W$. The acoustic model integrates knowledge of acoustics and phonetics. Since there are a large amount of words they are decomposed into smaller word units like phonemes.

$P(W)$ is the a-priori probability of a hypothesized word sequence $W$ and is called the *language model*. It is typically obtained with the help of big text corpora and provides knowledge of linguistic properties like grammar and language style.

$P(X)$ is the a-priori probability of observing a feature $X$. But since $X$ is fixed in the maximization of 2.14 $P(X)$ can be omitted from the equation.

The *pronunciation dictionary* contains information on which sequence of phonemes belongs to which word. The quality of the dictionary plays a very important role for the recognition.

As this work only deals with the building and modification of the acoustic model we do not go further into details about the language model, pronunciation dictionary or the decoding process. The next section will only cover some more details about the acoustic model.

### 2.2.1 Acoustic Model

As written above the acoustic model assigns probabilities to sequences of phonemes given a sequence of feature vectors received from the preprocessing. A phoneme describes a minimal unit of speech sound in a language that can be used to distinguish one word from another. There are around 40 phonemes used for a typical English spoken language systems. Their lengths vary between 30ms to 200ms. There are several other factors that need to be accounted for like speaker variations, pronunciation variations, environmental variations, and context-dependent phonetic coarticulation variations. To deal with such variances most systems use hidden Markov models for the statistical modelling of speech signals [HuAH01, p. 414].

#### Hidden Markov Model (HMM)

A hidden Markov model is an extension to the Markov chain [HuAH01, p. 386] and introduces a non-deterministic process that generates output observation symbols in any given state. It iss called hidden because the state sequence is not directly observable [HuAH01, p. 378]. A HMM is defined as a 5-tuple $\lambda = (S, \pi, A, B, V)$ consisting of

- $\boldsymbol{S = \{q_1, q_2, ..., q_N\}}$ A set of states where $s_t$ is denoted as the state at time $t$

- $\pi = \{\pi_i\}$ An initial probability distribution. $\pi_i = P(s_0 = q_i)$ is the probability of state $i$ at time $t = 0$.
- $\boldsymbol{A = \{a_{ij}\}}$ A transition probability matrix. $a_{ij} = P(s_t = q_j | s_{t-1} = q_i)$ is the probability of going from state $i$ into state $j$.
- $\boldsymbol{B = \{b_i(k)\}}$ An output probability matrix. $b_i(k)$ is the probability of emitting symbol $o_k$ when entering state $q_i$.
- $\boldsymbol{V = \{o_1, o_2, ...o_m\}}$ A set of observable symbols.

Given the above definition three fundamental problems are encountered when working with HMMs. These problems can be solved efficiently with dynamic programming.

**Evaluation Problem**: Given a model $\lambda$ and a sequence of observations $O = o_1, o_2, ..., o_T$, how to determine $P(X|\lambda)$, the probability of the observation given a model. This problem is solved by the *forward backward algorithm*[HuAH01, p. 391].

**Decoding Problem**: Given a model $\lambda$ and a sequence of observations $O = o_1, o_2, ..., o_T$, how to determine the state sequence $S = s_1, s_2, ..., s_T$ that produced the observations with the highest probability. The *Viterbi algorithm* is used for this problem [HuAH01, p.385].

**Learning Problem**: How to adjust the model parameters $\{A, B, \lambda\}$ to maximize $P(X|\lambda)$. This can be solved with the help of the *Baum-Welch algorithm* [HuAH01, p. 387].

The most common way in speech recognition is to model each phone with three HMM states: Begin, middle and end. This helps with handling time invariances and improves recognition. An example can be seen in Figure 2.5

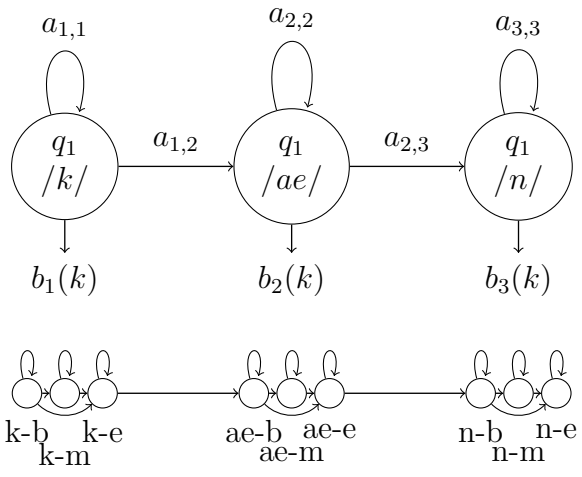


Figure 2.5: Top: Simple HMM example for the word „can". Bottom: Three states for each phone.

### GMM-HMM Model

For the last two decades a combination of HMM and Gaussian mixture models (GMM) have been used for acoustic modeling in state-of-the-art speech recognition systems. GMMs are weighted sums of different Gaussians or normal distributions and defined as follows:

$$p(x) = \sum_{i+1}^{N} w_i N(x|\mu_i, \Sigma_i) \tag{2.15}$$

where $N(x|\mu_i, \Sigma_i)$ denotes the Gaussian density function with mean vector $\mu$, covariance matrix $\Sigma$ and weights $w$ which satisfy $\sum_{i=1}^{N} w_i = 1$. As GMMs can approximate any continuous probability density function they can be used in place of the output probability matrix $B$ in a HMM [HuAH01, p. 392].

In a GMM-HMM acoustic model each state of the HMM consists of a GMM with a set of distributions each with their own weights, mean, and covariance matrix. The more distributions a GMM consists of, the better and more accurate the model performs. As there are much more parameters to estimate this approach requires a much larger set of training data for the ideal number of distributions.

### Hybrid DNN-HMM Model

In the last few years advances in machine learning and computer hardware have led to more efficient methods for training deep neural networks. Different research groups have shown that DNN based acoustic models can outperform GMM-HMM models and are now a central component in state-of-the-art speech recognition systems [SAMB+14].

The GMM can be replaced with a neural net which has been trained with the softmax function 2.6 to output the probabilities for a multi-class classification task. Each DNN output class corresponds to a single HMM state in the form of $P(\text{HMMState}|\text{AcousticInput})$. For the HMM to be able to compute a Viterbi alignment or run forward-backward algorithm requires the input in form of $P(\text{AcousticInput}|\text{HMMState})$. Luckily using Bayes' rules the latter can be calculated from first.

$$P(\text{AcousticInput}|\text{HMMState}) = \frac{P(\text{HMMState}|\text{AcousticInput}) \cdot P(\text{AcousticInput})}{P(\text{HMMState})}$$

$P(\text{AcousticInput})$ is the unknown probability of seeing a specific feature vector. As the probability is the same for all states it can be seen as an unknown scaling factor and be omitted [HDYD+12].

## 2.2.2 Error Metrics

Every machine learning system should be able to generalize which means it should also work well on unseen data. As the testing is always done with unknown data we need a way to measure the performance of the system. Common error metrics in speech recognition are the frame error rate and word error rate.

#### 2.2.2.1 Frame Error Rate (FER)

The frame error rate measures the error on frame level and is most often used in relation with acoustic models. The metric counts the number of misclassifications of frame labels by the model. Given a sequence of predictions $O = (o_1, ..., o_n)$ and a sequence of reference labels $T = (t_1, ..., t_n)$ the frame error rate can be written as follows:

$$\text{FER} = \frac{1}{n} \sum_{i=0}^{n} 1_{o_i \neq t_j} \tag{2.16}$$

$$1_{o_i \neq t_j} = \begin{cases} 1 \text{ if} & o_i \neq t_j \\ 0 \text{ otherwise} \end{cases} \tag{2.17}$$

This metric is used over the loss function to determine the point in time to terminate the training due to its similar behavior as the word error rate. It is preferred over the word error rate to measure the training performance because it does not require a speech recognition system to be computed.

#### 2.2.2.2 Word Error Rate (WER)

In speech recognition often the recognized word sequence (hypothesis) differs from the reference sequence in words and length. To measure the performance on word level it is common to calculate the edit distance between the reference and hypothesis by counting the minimum numbers of operations needed to transform the hypothesis into the reference. The operations are *insertion*, *deletion* and *substitution*. The word error rate is derived from the Levenshtein distance on word level between two word sequences $a$ and $b$ as defined in 2.18

$$\text{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if} \min(i,j) = 0 \\ \min \begin{cases} \text{lev}_{a,b}(i-1,j) + 1(\text{deletion}) \\ \text{lev}_{a,b}(i,j-1) + 1(\text{insertion}) \\ \text{lev}_{a,b}(i-1,j-1) + 1_{a_i \neq b_j}(\text{substitution}) \end{cases} & \text{otherwise} \end{cases} \tag{2.18}$$

$$1_{a_i \neq b_j} = \begin{cases} 0 \text{ if} a_i \neq b_j \\ 1 \text{ otherwise} \end{cases}$$

The word error rate is defined in 2.19 and its values lie between 0 and 1, sometimes it is also given in percent. In the event that the hypothesis is longer than the reference sequence the WER will be bigger than 1.

$$\text{WER} = \frac{\#\text{insertions} + \#\text{deletions} + \#\text{substitutions}}{\#\text{words in reference}} \tag{2.19}$$

## 2.3 K-Means

K-Means is an unsupervised clustering algorithm that aims to divide $n$ data points $\mathbf{X} \in \mathbb{R}^d$ into $k$ clusters $\mathbf{S}$ in a $d$-dimensional vector space. It was introduced more than 50 years ago [MacQ+67] but still enjoys great popularity to this day especially for vector quantization.

The idea is to start with $k$ centroids, one for each cluster $S_i$ with $i = \{1, .., k\}$, assign each data point to the closest centroid and update the centroids depending on the data points associated with the cluster. A set of centroids is also referred to as codebook.

### Lloyd Algorithm

The most popular variation of the k-Means algorithm is the Lloyd algorithm and was first proposed 1957 [Lloy82]. It works as follows:

1. **Initialisation:** Select $k$ random centroids $c_1, c_2, ..., c_k$.
2. **Assignment:** For each data point $x \in \mathbf{X}$ determine the nearest cluster by calculating the least squared Euclidean distance to each centroid.

   $S_i = \{x_p : ||x_p - c_i||^2 \leq ||x_p - m_j||^2 \forall j, 1 \leq j \leq k\}$
3. **Update:** Set the means of each cluster as their respective centroid.

   $c_i = \frac{1}{|S_i|} \sum_{x_j \in S_i} x_j$
4. Repeat step 2 und 3 until the centroids no longer change.

### K-Means++

The initialization of the centroids plays a big role as different locations cause different results. K-Means++ is a seeding technique that tries to tackle k-Means problem of finding poor clustering sometimes, resulting in better accuracy and faster convergence [ArVa07]. It achieves that by proposing a specific way of choosing the initial centroids.

1. Choose an initial centroid $c_1$ uniformly at random from X.
2. Choose the next center $c_i$, selecting $c_i = x' \in X$ with probability $\frac{D(x')^2}{\sum_{x \in X} D(x)^2}$.
   - D(x) denotes the shortest distance from a data point $x$ to the closest center that has already been chosen.
3. Repeat step 2 until k centroids have been chosen.
4. Continue with the regular k-Means algorithm.

Although the initial seeding takes up extra time the k-Means++ algorithm consistently outperforms the regular k-Means while also having faster completion times. A small example is illustrated in Figure 2.6.

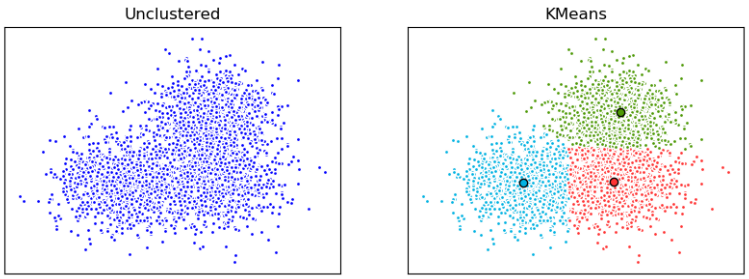


Figure 2.6: Visualization of k-Means. On the left the unclustered data, on the right the clustered data using k-Means and k-Means++

## 2.4 Floating Points

Floating point numbers are an efficient and by far the most popular way of representing real numbers in modern computers. Representing numbers as integers with a finite set of bytes has some significant disadvantages. Numbers with fractions like 1.234 can not be represented and very large numbers that don't fit into 32 bits like $2 \times 10^{30}$ can not be handled either [MBDJ+].
The floating point representation is defined by four integers:

- **radix** (or **base**) $\beta \geq 2$, it is always assumed to be even.
- **precision** $p \geq 2$.
- two **extremal exponents** $e_{\min}$ and $e_{\max}$ given that $e_{\min} < e_{\max}$.

In general a floating-point number $x$ is represented as

$$x = \pm d.dd\ldots d \cdot \beta^e \tag{2.20}$$
$$= \pm(d_0 + d_1\beta^{-1} + \cdots + d_{p-1}\beta^{-(p-1)}) \cdot \beta^e \tag{2.21}$$
$$= \pm m \cdot \beta^e \tag{2.22}$$

where

- $m$ is called the **mantissa** or **significand** and consists of $p$ digits.
- $e$ is called the **exponent** given that $e_{\min} < e \leq e_{\max}$.

[Gold91]
A graphical representation can be seen in Figure 2.7:

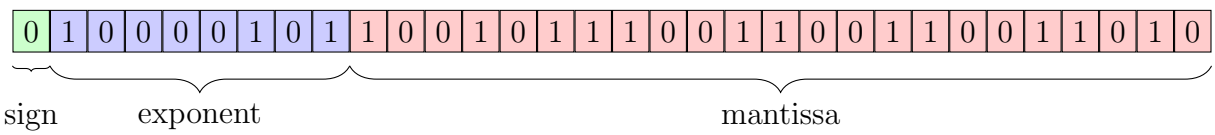


Figure 2.7: Binary representation of the number 101.8 following the IEEE 754 standard for 32-bit

### 2.4.1 IEEE Standard

In the current days almost all computers follow the IEEE 754 standard for their floating point representation. It was proposed in the year 1980 and established in the year 1985 introducing the 32-bit single precision float as well as the 64-bit double precision float. A new revision came in the year 2008 adding a definition for the 16-bit half precision float and 128-bit quad precision float [IEE08].
The IEEE Standard defines the integer parameters of a floating point number as follows:

- $b$, *base* (or *radix*) set to 2 or 10.
- $p$, numbers of digits in the significand/mantissa.
- $e_{\max}$, maximum exponent $e$.
- $e_{\min}$, minium exponent $e$, set to $1 - e_{\max}$.

The specific values are displayed in Table 2.1.

| | **sign** | **mantissa** | **exponent** | |
|---|---|---|---|---|
| | | **bits** | **bits** | $e_{\max}$ |
| **16-bit** | 1 | 10 | 5 | 15 |
| **32-bit** | 1 | 23 | 8 | 127 |
| **64-bit** | 1 | 52 | 11 | 1023 |
| **128-bit** | 1 | 112 | 15 | 16383 |

Table 2.1: Excerpt of the binary format parameters defined by the IEEE 754 standard

#### 2.4.1.1 Biased Exponent

In the IEEE Standard the exponent uses a biased representation which means the actual exponent value is offset by an exponent bias. The reason is that because using the signed two's complement representation would make comparison harder. Instead the exponent value is stored as an unsigned integer and when being interpreted it is converted into an exponent within a signed range by subtracting the bias. The purpose of the biases representation is that non-negative floating-point numbers can be treated as integers for comparison purposes which allow for easier arithmetic [Gold91].

The bias is defined as

$$b = 2^{k-1} - 1$$

where k denotes the length of the exponent bit vector. Lets take a 32-bit float as an example. The exponent is stored in 8-bits as an unsigned integer which results in a maximum value of 255. The bias is $2^{8-1} - 1 = 127$. As 0 and 255 serve a special purpose (see 2.4.1.3 and 2.4.1.4) the exponent is stored in a range of $\{1, 2, \ldots, 254\}$. Subtracting the bias leaves us with an exponent range of $[-126; 127]_{10}$.

#### 2.4.1.2 Normalized Numbers

Some representations of floating-point numbers are not unique. For example $0.01 \times 10^1$ and $1.00 \times 10^{-2}$ represent both the number 0.01. To achieve unique representations *normalized floating point numbers* are used. A number is normalized by bringing the mantissa into the range of $1 \leq m < \beta$ resulting in a leading nonzero digit. In our example $1.00 \times 10^{-1}$ is a normalized number.

Normal floating-point numbers can be noted as

$$(-1)^s \times 1.m \times 2^{e-b}$$

#### 2.4.1.3 Subnormal Numbers

As the leading binary digit is always nonzero there are no normal numbers with leading zeros. Instead leading zeros are moved to the exponent so that $0.01 \times 10^1$ is represented as $1.00 \times 10^{-1}$ instead.

The number 0 and small values around zero would lead to an exponent smaller than the possible minimum $e_{\min}$. Such numbers are called *denormalized* or *subnormal* numbers. By setting the leading significand digit to 0 and forcing the exponent to the minimum value subnormal numbers allow a representation closer to zero than the smallest normalized number. Because the sign can still be negative or positive, 0 can be presented as $+0$ or $-0$. As seen in Figure 2.8 without denormalized numbers, there would be a big gap between the number 0 and the smallest possible normal number.

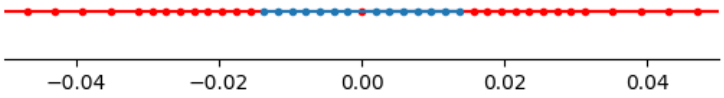


Figure 2.8: Subnormal numbers for 8-bit floats. Without subnormals (blue) there would be a gap between 0 and the next normal number (red)

The smallest normalized 32-bit number:

$$\pm 1.[00\ldots00]_2 \times 2^{-126} = \pm 2^{-126}$$

The smallest denormalized 32-bit number is much closer to zero:

$$\pm 0.[00\ldots01]_2 \times 2^{-126} = \pm 2^{-149}$$

#### 2.4.1.4 Special Cases

There are different special cases defined in the IEEE 754 standard. The first special case addresses the **signed zero**. Zero is represented by the exponent = 0 and an all zero mantissa. As the sign can still be 1 or 0, the number zero can take on two different values, $+0$ and $-0$. The standard defines $+0 = -0$ rather than $-0 < +0$

so that comparison like `if (x = 0)` don't cause undefined behavior.

The other special cases defined by the IEEE are covering numbers that don't fit into the range of defined numbers. For those cases numbers where the exponent consists of only ones (i.e $e = e_{\max} + 1$) are reserved [Gold91].

- If the mantissa is all zeroes the number represents $\infty$ or $-\infty$ depending on the sign bit. These values often result from an overflow and represent out-of-bound numbers.
- If the mantissa has some non-zero bits the number represents NaN, or "not a number". This number is used for error cases like $\sqrt{-1}$ or $\frac{0}{0}$

# 3. Related Work

First works on quantizing neural networks have been done by Gupta et al. [GAGN15] and Micikevicius et al. [MNAD+17]. The two works are quite similar as they both explore the training of neural networks using only half precision floating point numbers for network parameters such as weights, activations and gradients. Both works show that scalar quantization from 32-bit to 16-bit floating points can reduce the memory usage almost by a half without losing model accuracy.

The work by Wang et al. [WCBC+] is one of the first that shows the possibility to train a deep neural network with 8-bit floating point numbers. The lower precision numbers are used for numerical representation of data as well as computations during forward and backward pass. They introduce a new 8-bit format that allows for general matrix multiplication for deep learning without loss in model accuracy. They demonstrate that training with 8-bit floating point numbers gain a factor 2-4 speed up without compromising accuracy.

Vanhoucke et al. [VaSe] propose some techniques like batched lazy evaluation and optimized large matrix multiplication to reduce the computational costs for training and running deep neural networks on modern CPUs. The proposed techniques include the quantization of activations and intermediate layer weights into 8-bit chars. The input layer remained in 32-bit floating points to better accommodate the potentially larger dynamic range of inputs. They achieve a 2x speed up over their baseline setup and at the same time experience an absolute increase of only 0.1% in WER.

The work by Lei et al. [LSGS13] describe the development of a small-footprint, large vocabulary speech recognizer for mobile devices. They utilize a DNN-GMM acoustic model in combination with a compressed n-gram language model. In their setup they used a regular feed forward neural network trained using conventional backpropagation of gradients from a cross entropy error criterion with mini batch gradient descent. The minibatches were of size 200 frames with an exponentially decaying learning rate and a momentum of 0.9. To speed up the decoding process they first reduced the total number of parameters by node pruning and experimented with different numbers of nodes in input, output and hidden layers. The main steps of the DNN score computation were then speeded up with the help of numerous techniques. Additional to frame skipping [VaDH13] they utilize the batched lazy computation

as well as quantization of DNN described in Vanhoucke's work [VaSe]. Compared to the baseline GMM system their model achieved a relative WER improvement of 27.5% as well as a reduced memory consumption. The performance was still around 3% worse than their full-sized DNN setup. The overall model size could be reduced from 46mb to 17mb by 36%.

Based on their previous work by Lei et al. [LSGS13], McGraw et al. propose in [MPAA+16] a large vocabulary speech recognition system which is accurate, has low latency and a small memory and computational footprint by employing a quantized *Long Short-Term Memory* acoustic model. The system is used for dictation and voice commands and achieves 13.5% word error rate on the given tasks. The model is trained to optimize the connectionist temporal classification (CTC) criterion and predicts context independent phonemes. In comparison to the baseline cross entropy trained LSTM that predicts context dependent states it runs about 4 times faster. To further reduce the memory consumption the model parameters are quantized into a 8-bit integer based representation reducing the acoustic model's footprint to a forth of its original size. Through various additional techniques to optimize the language model and the decoding process they were able to build a system which is 7 times faster than real-time with a total system footprint of 20.3mb.

The work of Draper [Drap17] deals with language identification based on neural networks. He explores different network structures, different audio preprocessing and network post processing. One neural network structure he used was a feed forward net with node pruning. The geometry of the net was changed from 5 layers with 1000 neurons each to one of each layer having 200 less neurons than the previous one. Using that tree net structure turned out to be the best performing net with a relative improvement of 18% compared to the baseline setup.

Wang et el. propose in their work [WaLG15] to split each row vector of weight matrices of a neural network into sub-vectors and quantize them into a set of codewords using a split vector quantization algorithm. The optimal codebook is found using the LBG algorithm proposed in [LiBG80]. They compress the model even further by reshaping all weight matrices using singular value decomposition gaining a total reduction of the footprint by 75% to 80% without significant degradation of recognition performance. However, the work does not state if it is viable for real-time usage because it is unclear if this can be implemented efficiently while minimizing the runtime memory footprint.

# 4. Experimental Setup

This chapter provides an insight on the experimental setup used for this work introducing the necessary tools and frameworks as well as the data set used for training. All experiments deal with the neural network part of the acoustic model. The HMM part, the speech recognition itself, the language model as well as the dictionary uses the setup described in [NMSZ+].

## 4.1 Frameworks

This section provides a quick overview over the used tools for the experiments to train the different neural network models for the acoustic model and evaluate its performance as part of this thesis.

### 4.1.1 Janus Recognition Toolkit

The Janus Recognition Toolkit (JRTk) [WAWBC+94] is a general-purpose speech recognition toolkit developed at the Interactive Systems Lab (ISL) at Carnegie Mellon University and Karlsruhe Institute of Technology. It provides tools for the development of speech recognition systems for both research and application. The toolkit is written in C and offers a programmable shell to access Janus functionality through objects in the Tcl/Tk scripting language. JRTk offers methods for audio preprocessing, acoustic and language modeling, and the Ibis decoder. In this thesis JRTk was used to evaluate the performance of different DNN-HMM acoustic models where the DNN part was altered in several experiments to reduce size and computation time.

### 4.1.2 PyTorch

There are many tools available for machine learning. Especially for the programming language Python many frameworks exist. Some of the more notable frameworks are TensorFlow by Google [ABCC+] and Theano developed by Montreal Institute for Learning Algorithms [BBBL+]. A more recent addition to the deep learning ecosystem is PyTorch which was used in this thesis. It is based on Torch and is mainly developed by Facebook. It provides a good GPU support as well as automatic differentiation and differs from the other frameworks by offering a define-by-run paradigm

instead of a define-compile-run paradigm where the user expresses a computational graph which is then processed and compiled to compute the gradient. Due to its simplicity and flexibility it enjoys great popularity [Ketk17]. PyTorch was used in combination with scikit-learn[1] to train a feed forward neural network and alter various parameters for different experiments. For this work we used Python2.7 with PyTorch Version 0.2.0_3.

## 4.2 Dataset

The training and test dataset for the neural network consists of the audio data used in the IWSLT 2016 Evaluation Campaign [CNSB+][NMSZ+]. The audio corpus contains 483 hours of data and is composed of the Quaero dataset from 2010-2012, Broadcast News [Graf] and TED-LIUM v2 [RoDE]. From the whole set 17 hours were randomly selected to be used as test set. The remaining 451 hours were used as the training set.

## 4.3 Training Setup

The training setup is closely related to the one described in [NMSZ+]. We are using the same speech recognition system, training data with the same labels as well as the decoding setup for evaluating our acoustic model. The system utilizes a 4-gram language model and the CMU Pronounciation Dictionary [CMU] which consists of 39 phones. The acoustic model uses quinphones with three states per phoneme and a left-to-right HMM topology. All possible quinphones are modeled by 8156 distributions which is why the HMM has 8156 different states.

The training itself uses a modified setup described and developed in [Joeb18] which is a custom framework built on top of PyTorch with the focus on training a time delay neural network acoustic model. Simple changes allow the system to be used with feed forward neural networks as well. The used training data in that work is identical to the one we use in this thesis.

The audio signals are recordings with a sample rate of 16kHz and are used as the input for preprocessing. Each resulting frame has a length of 32ms and the frameshift between successive frames is 10ms. Each frame of samples consist of 40 log-mel features and is stacked in a temporal context of 11 frames which means the current frame is observed in a context of 5 predecessor and 5 successor frames.

## 4.4 Experimental Approaches

This section provides an overview on the experimental setup. It introduces the baseline system which performance serves as ground truth for the experiments carried out. The performed experiments can be divided into three major groups, each of them following a different approach.

The first experiments follow a novel approach where we try to find out if it would be possible to replace the neural network with a lookup table created by vector quantizing the training data.

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

The second experiments work on the quantization of network parameters. Instead of quantizing whole vectors as in the first experiments, the single weights were quantized by simulating the conversion into different float types and their respective value range.

The last experiments focus on downsizing the neural network model by pruning the nodes of the layers and changing the network topology. The input and output layer remained untouched but the hidden layer were modified to take on a Christmas tree like shape.

### 4.4.1 Baseline Setup

In order to evaluate the affect of the optimization attempts a baseline setup is needed to compare the results with. The acoustic model we built is a hybrid DNN/HMM system. As mentioned before we are only dealing with the neural network part of the model.

The DNN is a fully connected feed forward neural network with a total of 8 layers. Since the training data uses a temporal context of 11 with 40 log-mel features each the input layer has a size of 440 neurons. The amount of neurons in the output layer is equal to the HMM states, 8156. There are 6 hidden layers each with 1600 neurons. A simple schematic of the network can be seen in Figure 4.1. The input layer and each hidden layer utilizes a ReLU activation to pass the input to the next layer. As the neural network was trained for a classification task the last layer uses the softmax activation function and cross entropy as cost function.

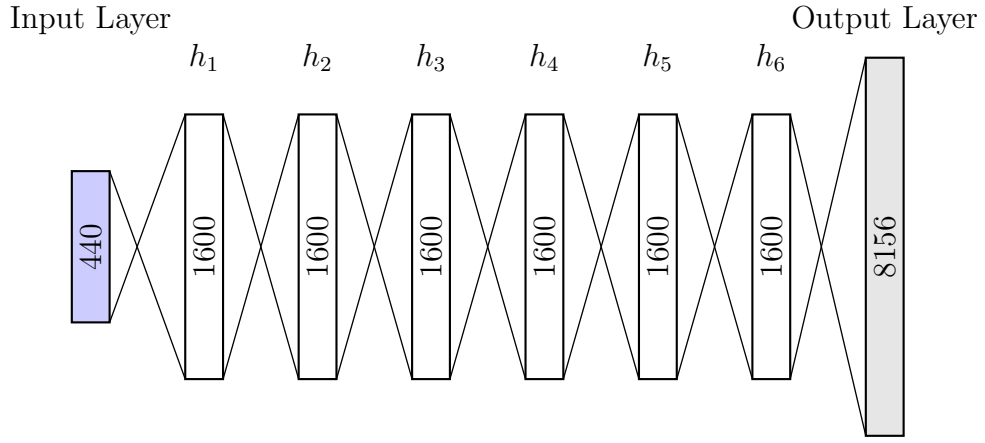


Figure 4.1: Baseline Feed Forward Network

Just like [NMSZ+] we use the same learning rate and scheduler for stochastic gradient descent training. The initial learning rate is set to 0.08 and we use a momentum of 5. We implement the Newbob algorithm as scheduler. If the validation error reduction falls below 0.5 the learning rate is halved after each epoch. The training will terminate as soon as the difference of validation error is smaller than the termination threshold which was set to 0.1. Training had to run for at least 4 epochs for the decaying to set in.

The baseline system achieved a frame error rate of **53.86%** on the training data and **59.98%** on the test data. The word error rate was **9.9%**

# 5. Vector Quantization with K-Means

The first experiments done were a completely novel approach where we clustered the training data to see if it might be possible to use the created lookup table instead of the neural network for classification. We assume that feature vectors with the same label have some correlation among each other and with the help of k-Means we might be able to generalize those. There haven't been done any research related to this approach yet and we were taking a high risk in failing. But if we manage to create a lookup table that is small enough and performs almost as good as the DNN it would reduce the necessary memory and hardware requirements by a huge margin as the lookup can be done efficiently on a CPU.

We decided to use k-Means as this is still one of the most used algorithms when it comes to vector quantization. Regular k-Means struggles with large datasets so instead we used a variation of the regular k-Means algorithm named Minibatch k-Means [Scul10]. As the name suggests the variation works with minibatches instead of the whole dataset at once. It was shown that Minibatch k-Means converges faster than regular k-Means although performing just slightly worse. Each minibatch is a subset of the input data, randomly selected from the dataset. Just like vanilla k-Means Minibatch k-Means operates in two steps. Assigning the samples to a centroid and updating them. The updates happen, in contrary to regular k-Means, on a per-sample basis. For each sample in the subset the centroid is updated by taking the moving average of the sample and all previous samples assigned to it. This results in an overall decreasing rate of change for a centroid over time.

For the evaluation we needed to create sets of centroids or codebooks of various sizes.

## 5.1 Global Clustering

We started out with $k = 15000$ and took the whole training set as input. The initial seeding points were set with k-Means$^{++}$ and the minibatch size was set to 2048. The average processing time for 100k data points took around 120 minutes. As the whole

data set consists of 160 million data points we had to reduce the size of training data and settled with 1 million randomly picked data points for the beginning. We created three more codebooks with the size $k = \{30.000, 60.000, 100.000\}$. For those sets we adjusted the amount of training data accordingly. All values are summarized in Table 5.1.

| **k** | **training data** | **average $t_{100k}$** |
|---|---|---|
| 15 000 | 1 000 000 | 120 min / 2 h |
| 30 000 | 5 000 000 | 318 min / 5.3 h |
| 60 000 | 10 000 000 | 704 min / 11.75 h |
| 100 000 | 10 000 000 | 997 min / 16.62 h |

Table 5.1: Summary of the created codebooks with the parameters $k$, the amount of data points used for training as well as the average time $t_{100k}$ it took to process 100k vectors

To evaluate the performance of the created centroids we took the baseline model and checked its performance when working with quantized data. We took the test data, found the closest codebook vector for each input and fed those to the system. If the lookup table provides appropriate codebook vectors the frame error rate should not vary much compared to the baseline. The resulting frame error rates can be seen in Table 5.2.

### 5.1.1 Results

| **Codebooks** | **Frame Error Rate** | **Size** |
|---|---|---|
| 15k | 93.29% | 26mb |
| 30k | 92.76% | 51mb |
| 60k | 93.53% | 102mb |
| 100k | 91.98% | 169mb |

Table 5.2: Frame error rate when using quantized input data for testing.

For all codebooks the frame error rate lies in the lower 90% which is an absolute increase of 30% compared to the baseline performance. Looking at the results it appears that the feature vectors provided by the codebooks do not represent feature vectors of the same label. We assume that happened because the calculated centroids don't have any information about labels or class affiliation due to the unsupervised nature of k-Means. Another factor was probably played by the low amount of training data we used to train the individual k-Means creating clusters which don't represent the whole feature set very well. Unfortunately using more data was too time consuming. We tried creating new codebooks using different minibatch sizes ranging from $2^5$ to $2^{12}$ but it did not yield any significant decrease in frame error rate.

## 5.2 Supervised Clustering

As the global clustering has not provided any useful results we decided to follow a „supervised" approach where the class labels were considered during the clustering. As perquisite the training data needed to be sorted first. The complete training set was divided by class label and all feature vectors of the same class were gathered

together in a separate file. To speed up the whole process the training data was split up into 10 parts and divided between several machines. The sorting was done on each machine using multiple CPUs and merged back together in a final step. We ended up with a total of 8018 different groups as some phonemes were not represented in the training set. For each group $g_i$ we calculated a k-Means codebook with $k_i$ centroids. We set a general upper limit $k_{max}$ for each $k_i$. Depending on $N_c$, the number of feature vectors of class c, $k_i$ was calculated with the formula 5.1

$$k_i = \begin{cases} k_{max} & \text{if } \frac{N_c}{3} > k_{max} \\ \frac{N_c}{3} & \text{otherwise} \end{cases} \tag{5.1}$$

The codebooks were again calculated with Minibatch k-Means and k-Means++. The single codebooks were then combined to form a big codebook containing all centroids. This way we were able to use the whole set of training data, create codebooks in a fraction of the time in comparison to the time needed to the global approach and we were sure that all available classes were represented. Table 5.3 shows the different codebooks created:

| | **cb5** | **cb10** | **cb20** | **cb30** | **cb40** | **cb50** |
|---|---|---|---|---|---|---|
| $k_{\mathbf{max}}$ | 5 | 10 | 20 | 30 | 40 | 50 |
| $k_{\mathbf{final}}$ | 40k | 80k | 160k | 240k | 320k | 400k |

Table 5.3: Summary of the supervised approach showing the general upper limit $k_{\max}$ as well as the number of centroids in the final codebook $k_{\text{final}}$

### 5.2.1 Results

We evaluated the performance of the supervised clusters following the same approach as with the globally clustered codebooks. As seen in Table 5.4 the frame error is still really high with values between 80% and 90%. The worst performing codebook performs slightly better than the best performing globally clustered one which is probably caused by fact that a lot more training data were used to create these codebooks. Checking the performance with test data we only achieve a frame error rate of 87.11%. Even when using training data for evaluation we only manage a FER of 75.75%. Though we trained the k-Means with knowledge of the labels the lookup does not return a codebook vector representing the input vector reliably.

| **Name** | $\mathbf{FER_{test}}$ | $\mathbf{FER_{test}}$ | **Size** |
|---|---|---|---|
| cb5 | 89.17% | 90.29% | 67mb |
| cb10 | 86.75% | 89.36% | 135mb |
| cb20 | 83.21% | 88.37% | 269mb |
| cb30 | 80.21% | 87.87% | 403mb |
| cb40 | 77.86% | 87.68% | 538mb |
| cb50 | 75.75% | 87.11% | 672mb |

Table 5.4: Frame error rates when validating the baseline system with quantized test data. The row second from the left shows the result of k-Means using training data for validation, the next row used test data for validation

Bigger codebooks probably yield better results but are accompanied with an exponential increase of lookup time to find the next closest codebook. We did not create

bigger codebooks as cb50 was already 672mb in size and the gain in accuracy would probably be minimal. There is still a lot of room for improvement but we decided not to perform any further experiments and moved on to the next task.

## 5.3 Vector Quantization of Model Parameters

For the next steps we planned to work with the network parameters to see the performance of our model when the model parameters were clustered with vector quantization. First we needed training data to create the codebooks by running a single epoch with the baseline model and feeding it with our available training data. At the same time we wrote the output of the layers into a file. Unfortunately the file reached a size of several hundred gigabytes only after a few minibatches even when only taking the output of a single layer into account. At the time we did not have the capacities to store this huge amount of data. We tried various ways to store the file efficiently i.e *pickle*, *joblib* with the highest compression, *h5py* and writing to raw binary with *numpy* but none of the tools we tried worked to our satisfaction. We scrapped the idea for the time being and did not pursue it any further.

# 6. Simulation of Floating Point Numbers

After vector quantization we followed the approach of scalar quantization. Following the idea of [VaSe] and [LSGS13] we cast the network parameters in the hidden layers into different float presentations to see the impact on the recognition when using lower precision numbers instead of the 32-bit floats used by default. Converting the model parameters to 16-bit, 8-bit or even 4-bit could potentially reduce the size of the model by a factor $\times 2/4/8$. At the time this thesis was written PyTorch v0.2.0_3 only supported 64-bit, 32-bit and 16-bit floats natively. As there are no 8-bit and 4-bit floating point numbers we had to simulate those with 32-bit floating points. On contrary to the mentioned works we decided to quantize not only the model parameters but the input data as well. The different parameters used for 8-bit and 4-bit precision are displayed in Table 6.1.

| | **bits** | **s** | **e** | **m** | **e_max** | **e_min** |
|---|---|---|---|---|---|---|
| **8-bit** | 8 | 1 | 4 | 3 | 7 | -6 |
| **4-bit** | 4 | 1 | 2 | 1 | 1 | 0 |

Table 6.1: Overview of all parameters for 8-bit and 4-bit precision

## 6.1 16-Bit Precision

As PyTorch provides the ability to switch between the supported floating point types this experiment was very straight forward. Because we had to simulate 8-bit and 4-bit floating points we decided to simulate the 16-bit conversion as well for consistency reasons. During the forward pass we convert the output $y$ of each layer to half precision simply with `y = y.half()` right before applying the ReLU function. This is only possible because ReLU just passes through the input or sets it to 0. In case of other nonlinear activation functions the cast needs to happen after applying the nonlinearity. The output of the ReLU is then cast back to 32-bit float precision and passed to the next layer.

## 6.2 8-Bit Precision

For 8-bit precision we decided on 1 bit sign, 4 exponent bits and 3 mantissa bits which results in a bias $b = 7$, in short a 1.4.3.-7 number. With 8 bits there is a total of 255 possible values. As we tried to stay true to the IEEE 754 standard as close as possible the 16 values where the exponent consists of ones only are reserved for the special cases NaN, $\infty$ and $-\infty$. All possible values can be seen in Table 6.2 and have a maximum range of -240 to 240. If we swapped the number of bits for exponent and mantissa (1.3.4.-3) we would end up with a smaller range of -15 to 15 which is why we decided to go with the first variation.

| Exponent \ Mantissa | **0 eeee 000** | **0 eeee 001** | **0 eeee 010** | **0 eeee 011** | **0 eeee 100** | **0 eeee 101** | **0 eeee 110** | **0 eeee 111** |
|---|---|---|---|---|---|---|---|---|
| **0 0000 mmm** | 0.0 | 0.001953125 | 0.00390625 | 0.005859375 | 0.0078125 | 0.009765625 | 0.01171875 | 0.013671875 |
| **0 0001 mmm** | 0.015625 | 0.017578125 | 0.01953125 | 0.021484375 | 0.0234375 | 0.025390625 | 0.02734375 | 0.029296875 |
| **0 0010 mmm** | 0.03125 | 0.03515625 | 0.0390625 | 0.04296875 | 0.046875 | 0.05078125 | 0.0546875 | 0.05859375 |
| **0 0011 mmm** | 0.0625 | 0.0703125 | 0.078125 | 0.0859375 | 0.09375 | 0.1015625 | 0.109375 | 0.1171875 |
| **0 0100 mmm** | 0.125 | 0.140625 | 0.15625 | 0.171875 | 0.1875 | 0.203125 | 0.21875 | 0.234375 |
| **0 0101 mmm** | 0.25 | 0.28125 | 0.3125 | 0.34375 | 0.375 | 0.40625 | 0.4375 | 0.46875 |
| **0 0110 mmm** | 0.5 | 0.5625 | 0.625 | 0.6875 | 0.75 | 0.8125 | 0.875 | 0.9375 |
| **0 0111 mmm** | 1.0 | 1.125 | 1.25 | 1.375 | 1.5 | 1.625 | 1.75 | 1.875 |
| **0 1000 mmm** | 2.0 | 2.25 | 2.5 | 2.75 | 3.0 | 3.25 | 3.5 | 3.75 |
| **0 1001 mmm** | 4.0 | 4.5 | 5.0 | 5.5 | 6.0 | 6.5 | 7.0 | 7.5 |
| **0 1010 mmm** | 8.0 | 9.0 | 10.0 | 11.0 | 12.0 | 13.0 | 14.0 | 15.0 |
| **0 1011 mmm** | 16.0 | 18.0 | 20.0 | 22.0 | 24.0 | 26.0 | 28.0 | 30.0 |
| **0 1100 mmm** | 32.0 | 36.0 | 40.0 | 44.0 | 48.0 | 52.0 | 56.0 | 60.0 |
| **0 1101 mmm** | 64.0 | 72.0 | 80.0 | 88.0 | 96.0 | 104.0 | 112.0 | 120.0 |
| **0 1110 mmm** | 128.0 | 144.0 | 160.0 | 176.0 | 192.0 | 208.0 | 224.0 | 240.0 |
| **0 1111 mmm** | $\infty$ | NaN | NaN | NaN | NaN | NaN | NaN | NaN |

Table 6.2: 8-bit float representation for all possible positive numbers according to IEEE 754 standard. Negative values are not shown as the table is symmetrical

The simulation of 8-bit precision was done in two steps. First transforming the 32-bit float number $x$ into a 8-bit binary form by calculating the three necessary parameters. Finding the value of the sign bit was trivial. The exponent was calculated by bringing the given number $x$ into a value range of $1 \leq x < 2$ by multiple multiplications or divisions with 2. For the mantissa we had to treat all special cases. First the case for denormal numbers. If the original float was too small, it was flushed to 0. Afterwards the two special cases $\infty$ and NaN were handled and the mantissa set accordingly. In case of $\pm\infty$ the given float was set to the highest/lowest possible value, in our case $\pm$240.0. If non of the special cases were true the mantissa was calculated from the remainder. Finally with all parameters determined the 8-bit value could be obtained with the help of formula 2.4.1.2 from the previous chapter. The complete function that was used to convert a 32-bit float to an 8-bit float can be found in appendix A.

There were some numbers where the calculated mantissa would equal to $2^3 = 8$ causing an overflow in the mantissa as it is only 3-bits long. The mantissa ends up as $m = 0$ and together with the calculated exponent the conversion would result in a wrong value (i.e 31 becomes 16.0 instead of 30). For values $> 1$ we moved the number in steps of 0.5 to the next ten until the mantissa was correctly set. This issue also happened with some numbers between 0 and 1. For that case we reduced the precision of the decimal place by one (i.e 0.02358 to 0.0235). Both approaches appeared to be successful in avoiding the mentioned issue and managed to return a more reasonable conversion result.

The 8-bit simulation was done following the same scheme as the 16-bit conversion. Because the numbers were still in 32-bit precision there was no necessity to convert

the numbers back again. Unfortunately a forward pass with 8-bit simulation was 10 times slower than a regular forward pass (0.9s instead of 0.09s) causing a single epoch to take about 40 hours to finish instead of the regular 4-5 hours.

## 6.3 4-Bit Precision

For the 4-bit precision we used 1 sign bit, 2 exponent bits and 1 mantissa bit resulting in 16 possible values. Again, exponents consisting only of ones are reserved for $\infty$, $-\infty$ and NaN. All possible values are displayed in Table 6.3

| **Mantissa** / **Exponent** | **0 ee 0** | **0 ee 1** |
|---|---|---|
| **0 00 m** | 0 | 0.5 |
| **0 01 m** | 1.0 | 1.5 |
| **0 10 m** | 2.0 | 3.0 |
| **0 11 m** | $\infty$ | NaN |
| **1 00 m** | 0 | -0.5 |
| **1 01 m** | -1.0 | -1.5 |
| **1 10 m** | -2.0 | -3.0 |
| **1 11 m** | $-\infty$ | NaN |

Table 6.3: 4-bit float representation for all possible positive numbers inspired by IEEE 754 standard

As there are only a total of 12 actual values, the range is very limited and a simple conversion from 32-bit to 4-bit ends up with a lot of numbers as $+\infty$ or $-\infty$. To cope with the limited value range we have three different approaches. Each attempt used the same topology and scheduler as the baseline setup. All approaches were done with both the training data converted to 4-bit and keeping it in 32-bit floating point format.

### Clamping

In the first approach we decided to clamp the numbers which fell out of range of the biggest/smallest possible number in 4-bit precision. Every number bigger than 3 is set to 3, analogical for negative numbers. Input values that fell in between the given range were converted to 4-bit precision following the same procedure as 8-bit precision.

### Bucketing

Next we divided the range of input values into 11 buckets with values $\{-3, -2, -1.5, -1, -0.5, 0, 0.5, 1, 1.5, 2, 3\}$. During the training we determined the current minimum and maximum values of input data and model parameters of the current minibatch as well as the values from prior minibatches to divide the value range into 11 equidistant sections. We assigned each section to a bucket and for each number in a specific bucket range, the bucket value was returned instead.

### Scaling

Instead of dividing the range into buckets we scaled the input data and layer outputs during the training to the range $[-3, 3]$ and converted them into their respective 4-bit representation afterwards. The minimum and maximum values for scaling the values were again determined dynamically during training.

## 6.4 Results

| Precision | Frame Error Rate | Word Error Rate |
|---|---|---|
| **Baseline (32-bit)** | 59.98% | 9.9% |
| **16-bit** | 59.98% | 10.0% |
| **8bit** | 62.34% | 10.1% |
| **4-bit clamp** | 98.66% | n.e |
| **4-bit bucket** | 98.4% | n.e |
| **4-bit scale** | 96.63% | n.e |

Table 6.4: Frame error rate as well as word error rate when training a model with lower precision numbers. Due to the bad performance with 4-bit floating point numbers the word error rate could not be evaluated and marked as such (n.e)

Table 6.4 shows the results of our experiments when training a model with a lower precision than 32-bit floating point type format. By using 16-bit and 8-bit floating points we were able to create two systems that performed close to on par with the baseline. As expected casting the model parameters to half precision caused a barely noticeable decrease of 0.1% in accuracy even though we also quantized the input data to 16-bit. It appears that half precision can still cover the big range of the training data just fine. The results are similar to related work.

Even though casting all parameters and input data to 8-bit caused a slight increase in frame error rate, the WER only increased to 10.1%. The decrease in performance might fall back to the implementation on how the special cases were handled but most probably due to noise in the test data.

None of the 4-bit approaches showed great success. All trained models have a bad performance with an average FER of 98%. At first glance we suspected the casting of the input data as the cause of the drop in accuracy. But leaving the input data as 32-bit did not show any improvements. The main issue must lie within the conversion of the model parameters that causes the model not being able to learn. It appears that 4-bit precision is not sufficient to deal with the value range of the model parameters. Due to the bad performance the decoding script cancelled the execution and no word error rate could be measured. Therefore the error rate was not evaluated.

# 7. Node Pruning

As described in the work of [LSGS13] one possibility of reducing the size of a model and its footprint is the modification of the topology by pruning the nodes. The baseline setup has a total of 27M parameters and the trained model has a size of 102mb. By simply reducing the numbers of neurons in the layers the number of parameters is decreased resulting in faster training times and an overall smaller model. The input layer and the output layer remained untouched. The changes were only done on the intermediate layers.

| | $h_1$ | $h_2$ | $h_3$ | $h_4$ | $h_5$ | $h_6$ |
|---|---|---|---|---|---|---|
| **L100** | 1600 | 1500 | 1400 | 1300 | 1200 | 1100 |
| **L150** | 1600 | 1450 | 1300 | 1150 | 1000 | 850 |
| **L200** | 1600 | 1400 | 1200 | 1000 | 800 | 600 |
| **L250** | 1600 | 1350 | 1100 | 850 | 600 | 350 |
| **L300** | 1600 | 1300 | 1000 | 700 | 400 | 100 |
| **E1** | 1600 | 1590 | 1570 | 1520 | 1380 | 1010 |
| **E1.5** | 1600 | 1590 | 1560 | 1500 | 1330 | 860 |
| **E2** | 1600 | 1580 | 1540 | 1480 | 1280 | 720 |
| **E2.5** | 1600 | 1570 | 1530 | 1400 | 1060 | 130 |

Table 7.1: Numbers of neurons in the hidden layer for all linear and exponential Variations

Similar to the approach of Draper [Drap17] we decided to reduce the number of neurons in the successive hidden layers. We left the first layer to have a size of 1600 neurons in all variations. The numbers of neurons in the following hidden layers is always smaller than the numbers of neurons in the previous layer resulting in a Christmas tree like structure. The first variation reduced the hidden layer sizes in a linear fashion by decreasing the amount of neurons in each successive layer by a fixed amount. The numbers of neurons per layer can be calculated with following the equation:

$$f(x) = 1600 - N \cdot x$$

where $N$ is the fixed number which will be subtracted from the number of neurons and $x = \{0, 1, 2, 3, 4, 5\}$ represents the hidden layer. For our experiments we set

$N = \{100, 150, 200, 250, 300\}$ .
After that we reduced the nodes with exponential behavior. The neurons per layer were formulated as:

$$f(x) = 1600 - \frac{1600}{-x + 6} * \varphi$$

where $\varphi$ denotes a factor that is scaled with the value. The factor was set to $\varphi = \{1, 1.5, 2, 2.5\}$. The values are then rounded to the nearest ten. The values for all variations are displayed in Table 7.1. In total we trained 9 more models. 5 with a linear and 4 with an exponential behavior.

## 7.1 Results

It is common knowledge that bigger and deeper models have a higher accuracy and in return smaller models perform worse. The results in Table 7.2 show a clear indication that accuracy deteriorates with declining numbers of parameters but the impact is not as bad as assumed.

| | **Baseline** | **L100** | **L150** | **L200** | **L250** | **L300** | **E1** | **E1.5** | **E2** | **E2.5** |
|---|---|---|---|---|---|---|---|---|---|---|
| **WER** | 9.9% | 10.0% | 10.3% | 10.3% | 10.6% | 11.0% | 10.3% | 10.2% | 10.5% | 10.4% |
| **Parameters** | 27M | 19M | 15M | 12M | 9M | 6M | 20M | 17M | 13M | 10M |
| **Size** | 102mb | 73mb | 59mb | 46mb | 34mb | 23mb | 76mb | 64mb | 52mb | 40mb |
| **Size Proportion** | 100% | 70% | 56% | 44% | 33% | 22% | 75% | 63% | 45% | 37% |
| **Speed (GPU)** | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| **Speed (CPU)** | 1.0 | 1.39 | 1.74 | 2.2 | 2.83 | 4.1 | 1.34 | 1.57 | 1.87 | 2.35 |
| **Decoding Speedup** | 1.0 | 1.07 | 1.12 | 1.14 | 1.15 | 1.19 | 1.15 | 1.10 | 1.20 | 1.22 |

Table 7.2: Word error rate for the different pruned models, number of parameters in million, the resulting size, the size proportion to the baseline, how much faster the model runs compared to the baseline model on a GPU and CPU as well as how much faster the decoding process is when using the trained models

Pruning nodes and simply reducing the size of the hidden layers might cause some trouble for the neural net to model the training data. Removing too many nodes hinders the network to learn correctly. The average increase of WER for the linear variation is 0.54% with the worst case being *L300* with an absolute increase of 1.1% resulting in 11% word error rate but reducing the model size by 77% to only 23mb. The average increase for the exponential variation is 0.425% with the worst performing model being *E2* with an absolute increase of 0.6% resulting in a WER of 10.5% and a size reduction of 50.98% to 52mb.

The best performance with the least increase in error rate is the L-Variation *L100*. It manages a word error rate of 10% and a smaller size of 73mb which is 29% smaller than the baseline model. The E-Variation *E1.5* with a below average WER increase of 2.9% performs really well also. It achieves a word error rate of 0.3% while reducing the size of the model from 102mb to 64mb which is a reduction by 37%.

There is no noticeable speedup when using a GPU to validate the model and all models ran at approximately the same speed. There were noticeable differences on the CPU and, as expected, the smaller the model, the faster it runs. All tests were executed on the same CPU to maintain consistency. The speedup is linearly proportional to the number of parameters i.e *L300* is approximately a quarter the size of the baseline and runs 4 times faster. The decoding process could be sped up by a maximum of 22% when using the smallest exponential variation $E2.5$. It seems that the search still takes up most of the time during the decoding.

## 7.2 Combining Node Pruning and Floating Point Simulation

By combining both the node pruning and lower precision floating points we could create models which are a fraction of the baseline model's size and footprint. We did not combine the node pruning and 4-bit floating point simulation because the latter, as seen in the previous section, has proved unsuccessful. The resulting WER and size are displayed in Table 7.3. The speedup and number of parameters are not shown as there were no changes.

### 7.2.1 Results

| | **L100** | **L150** | **L200** | **L250** | **L300** | **E1** | **E1.5** | **E2** | **E2.5** |
|---|---|---|---|---|---|---|---|---|---|
| **$WER_{16\text{-bit}}$** | 10.1% | 10.1% | 10.0% | 10.4% | 11.0% | 10.2% | 10.2% | 10.2% | 10.6% |
| **Size** | 37mb | 29mb | 23mb | 17mb | 12mb | 38mb | 32mb | 26mb | 20mb |
| **$WER_{8\text{-bit}}$** | 9.9% | 9.8% | 10.3% | 10.3% | 10.9% | 10.0% | 10.2% | 10.3% | 10.3% |
| **Size** | 19mb | 15mb | 12mb | 9mb | 6mb | 19mb | 16mb | 13mb | 10mb |

Table 7.3: Resulting WER and size when combining node pruning with scalar quantization

The results for the 16-bit combination is comparable to the pruned model with parameters in 32-bit float precision which ought to be expected as the usage of 16-bit parameters didn't affect the performance much. To our surprise the best performing model manages to reduce the word error rate by 0.1% achieving a better performance than the baseline system. This was probably caused again by noise in the test data. Another possible explanation could be that the system generalizes better with the limited range of possible values so that it is less likely to overfit. In terms of overall performance E2.5 seems to be best as it hits the sweet spot of balance between size and performance. It manages a decoding speedup of 22% (see Table 7.2), reduces the size by 90% down to 10mb and performs only 0.4% worse than the baseline.

# 8. Conclusion

In this final chapter we summarize the results achieved in this work. After that we discuss some ideas for future work.

## 8.1 Summary

In this thesis we tried different approaches to optimize a DNN acoustic model. The focus of this work was put on the reduction of the footprint and memory usage to open up the possibilities for the system to run in an environment with limited hardware and memory. We modified the neural network part of the acoustic model starting with a feed forward neural network with 6 hidden layers of 1600 neurons each as a baseline. The baseline system manages a frame error rate of 59.98% and a word error rate of 9.9%

First experiments follow a novel approach where we try to create lookup tables in hope to replace the neural network of the acoustic model for classification. We create different codebooks with the help of the k-Means algorithm using the whole feature set to validate the performance of the baseline system when using quantized test data. Clustering the training data without considering the class label results in an average error rate of 92.89% and proves to be unsuccessful. Creating codebooks where the class affiliation of each feature vector is considered manages to achieve slightly better results. The best performing codebook achieves a frame error rate of 87.11% still leaving a lot of room for improvements.

We then move on to scalar quantization of the model parameters. We simulate the usage of 16-bit, 8-bit and 4-bit floats instead of 32-bit floating point numbers by converting the parameters to the next closest number in the given value range the lower precision floats offeres. The 16-bit and 8-bit model show almost no degradation in performance while reducing the footprint of the trained model by up to 75%. Due to it's limited value range the simulation with 4-bit floats prove to be unsuccessful increasing the frame error rate by an absolute value of 38%.

In the last approach we change the structure of the neural network by removing nodes from the hidden layers. We prune more and more nodes for each successive

hidden layer resulting in a Christmas tree like structure. With decreasing execution time and decreasing model size the word error rate peaks at 11% for the smallest model which is only a fifth of the original size and runs 20% times faster.

Combining node pruning and scalar quantization we manage to create a model which performed better than the baseline system and achieves a word error rate of 9.8% and gaines a speed up of 12%. We are able to reduce the size of the model by 82.11% resulting in a model with a size of 15mb instead of 102mb. The smallest created system achieves a speed up of 22% while being 10 times smaller than the baseline system. The word error rate increases by only 0.4%.

In conclusion except for vector quantization the experiments prove to be successful as we manage to create models that perform with an accuracy close to the baseline system's performance while reducing its footprint drastically and therefore providing beneficial results. Looking at the results from an economic aspect the learnings from this work are not limited to the usage of mobile environments only. The gain in speed and savings of resources even for the worst performing model with a WER of 11% is a huge improvement over a system which would run 20% times slower and would take up way more resources. In a real-live scenario this could potentially save a lot of costs for servers as hardware and memory are considerably expensive.

## 8.2 Future Work

There are several possibilities to optimize and improve the performance of the introduced models. The approaches presented in this thesis only focus on the reduction of footprint and memory usage but not actively reducing the decoding process.

First of all putting back the ultimate goal in mind it would be necessary to perform experiments on actual embedded/mobile devices as all experiments were done on machines with slower CPUs.

The results show that the search still takes up most of the time of the decoding process. Therefore optimizing the language model brings one closer to the original goal.

Replacing the simulation of fixed point numbers and implementing 8-bit precision including the needed arithmetic like general matrix multiplication would erase the time consuming conversion as well as provide an actual speed up of processing time.

Following the approach of quantizing the model parameters with vector quantization should be continued. Exploring different ways to apply vector quantization and using different techniques like [WaLG15] or learning vector quantization are possible alternatives.

Recent state-of-the-art speech recognition systems utilize recurrent neural networks (RNN) instead of DNN and achieve excellent performance on large scale data [SaSB]. As McGraw et al. have shown in their work [MPAA+16] taking our learnings and working with RNN might improve our results even more.

Another possible way to speed up the model execution would be to train a model with *noise contrastive estimation* (NCE). The idea behind NCE is to train an unnormalized feed forward neural to encourage output scores which behave like probabilities and sum up to 1. This way the costly Softmax activation function would

be obsolete. Several works have successfully applied NCE in their language model [DZHL+14b][HuSR17]. Training an acoustic model with NCE would not only decrease training time but also accelerate the decoding process.

# A. Appendix

**Algorithm 1** Convert a given 32-bit floating point number to a possible 8-bit representation

```
1: procedure FloatToMinifloat(x)
2:     e ← 0; s ← 0; m ← 0
3:     bias ← 7
4:     if x = 0 then
5:         return 0
6:     end if
7:     if x < 0 then
8:         s ← 1; x ← −1 · x
9:     end if
10:    r ← x
11:    while r < 2 do
12:        r ← x/2; e ← e − 1
13:    end while
14:    while r ≥ 2 do
15:        r ← 2 · x; e ← e + 1
16:    end while
17:    if e ≤ −7 then
18:        if x ≥ 1/512 then                          ▷ denormals
19:            e ← 0; m ← int(512 · x + 1/2)
20:            return (−1)^s · m/512
21:        else
22:            return 0
23:        end if
24:    else
25:        if e > (bias + 1) then
26:            return (−1)^s · 240.0                  ▷ ±∞ set to min/max value
27:        else
28:            m ← int((r − 1) · 8 + 1/2)
29:            return (−1)^s · (1 + m/8) · 2^(e−bias)
30:        end if
31:    end if
32: end procedure
```

# Bibliography


[ABCC+] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu und X. Zheng. TensorFlow: A system for large-scale machine learning. S. 21.

[ArVa07] D. Arthur und S. Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2007, S. 1027–1035.

[BBBL+] J. Bergstra, F. Bastien, O. Breuleux, P. Lamblin, R. Pascanu, O. Delalleau, G. Desjardins, D. Warde-Farley, I. Goodfellow, A. Bergeron und Y. Bengio. Theano: Deep Learning on GPUs with Python. S. 4.

[CMU] The CMU Pronouncing Dictionary.

[CNSB+] M. Cettolo, J. Niehues, S. Stuker, L. Bentivogli, R. Cattoni und M. Federico. The IWSLT 2016 Evaluation Campaign. S. 14.

[DeHK13] L. Deng, G. Hinton und B. Kingsbury. New types of deep neural network learning for speech recognition and related applications: An overview. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE, 2013, S. 8599–8603.

[Drap17] D. H. Draper. Online Neural Network-based Language Identification, 2017.

[DZHL+14a] J. Devlin, R. Zbib, Z. Huang, T. Lamar, R. Schwartz und J. Makhoul. Fast and robust neural network joint models for statistical machine translation. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Band 1, 2014, S. 1370–1380.

[DZHL+14b] J. Devlin, R. Zbib, Z. Huang, T. Lamar, R. Schwartz und J. Makhoul. Fast and Robust Neural Network Joint Models for Statistical Machine Translation. Association for Computational Linguistics, 2014, S. 1370–1380.

[FrSc99] Y. Freund und R. E. Schapire. Large margin classification using the perceptron algorithm. *Machine learning* 37(3), 1999, S. 277–296.

[GAGN15] S. Gupta, A. Agrawal, K. Gopalakrishnan und P. Narayanan. Deep Learning with Limited Numerical Precision. *arXiv:1502.02551 [cs, stat]*, 2015.

[GoBC16] I. Goodfellow, Y. Bengio und A. Courville. *Deep Learning*. MIT Press. http://www.deeplearningbook.org, 2016.

[Gold91] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys* 23(1), 1991, S. 5–48.

[Graf] D. Graff. THE 1996 BROADCAST NEWS SPEECH AND LANGUAGE-MODEL CORPUS. S. 4.

[HDYD+12] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath und andere. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal processing magazine* 29(6), 2012, S. 82–97.

[Horn91] K. Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks* Band 4, 1991, S. 251–257.

[HuAH01] X. Huang, A. Acero und H.-W. Hon. Spoken Language Processing: A Guide to Theory, Algorithm and System Development. 2001.

[HuSR17] Y. Huang, A. Sethy und B. Ramabhadran. Fast Neural Network Language Model Lookups at N-Gram Speeds. 2017, S. 274–278.

[IEE08] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, August 2008, S. 1–70.

[Joeb18] E. Joebstl. Reverberation Robust Acoustic Modeling Using Time Delay Neural Networks, 2018.

[Ketk17] N. Ketkar. Introduction to PyTorch. In N. Ketkar (Hrsg.), *Deep Learning with Python: A Hands-on Introduction*, S. 195–208. Apress, Berkeley, CA, 2017.

[KrSH12] A. Krizhevsky, I. Sutskever und G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 2012, S. 1097–1105.

[LADSOS17] A. T. Lopes, E. de Aguiar, A. F. De Souza und T. Oliveira-Santos. Facial expression recognition with convolutional neural networks: coping with few data and the training sample order. *Pattern Recognition* Band 61, 2017, S. 610–628.

[LiBG80] Y. Linde, A. Buzo und R. Gray. An Algorithm for Vector Quantizer Design. Januar 1980, S. 84–95.

[Lloy82] S. Lloyd. Least squares quantization in PCM. *IEEE transactions on information theory* 28(2), 1982, S. 129–137.

[LSGS13] X. Lei, A. Senior, A. Gruenstein und J. Sorensen. Accurate and compact large vocabulary speech recognition on mobile devices. In *in Proc. Interspeech*, 2013.

[MacQ+67] J. MacQueen und andere. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, Band 1. Oakland, CA, USA, 1967, S. 281–297.

[MBDJ+] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefevre, G. Melquiond, N. Revol, D. Stehle und S. Tones. Handbook of Floating-Point Arithmetic. S. 10.

[McPi43] W. S. McCulloch und W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics* 5(4), 1943, S. 115–133.

[MNAD+17] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh und H. Wu. Mixed Precision Training. *arXiv:1710.03740 [cs, stat]*, 2017.

[MPAA+16] I. McGraw, R. Prabhavalkar, R. Alvarez, M. G. Arenas, K. Rao, D. Rybach, O. Alsharif, H. Sak, A. Gruenstein, F. Beaufays und C. Parada. Personalized Speech recognition on mobile devices. *arXiv:1603.03185 [cs]*, 2016.

[NMSZ+] T.-S. Nguyen, M. Muller, M. Sperber, T. Zenkel, K. Kilgour, S. Stuker und A. Waibel. The 2016 KIT IWSLT Speech-to-Text Systems for English and German. S. 6.

[RaZL18] P. Ramachandran, B. Zoph und Q. V. Le. Searching for activation functions. 2018.

[RoDE] A. Rousseau, P. Deléglise und Y. Estève. Enhancing the TED-LIUM Corpus with Selected Data for Language Modeling and More TED Talks. S. 5.

[Rose58] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review* 65(6), 1958, S. 386.

[RuNo09] S. Russell und P. Norvig. Artificial intelligence: a modern approach. 3rd. *Essex, UK: Parentice Hall*, 2009, S. 1152.

[SAMB+14] M. Shahin, B. Ahmed, J. Mckechnie, K. Ballard und R. Gutierrez-Osuna. A Comparison of GMM-HMM and DNN-HMM Based Pronunciation Verification Techniques for Use in the Assessment of Childhood Apraxia of Speech. 09 2014.

[SaSB] H. Sak, A. Senior und F. Beaufays. Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling. S. 5.

[SBBB+10] J. Schalkwyk, D. Beeferman, F. Beaufays, B. Byrne, C. Chelba, M. Cohen, M. Kamvar und B. Strope. "Your Word is my Command": google search by voice: A case study. In *Advances in speech recognition*, S. 61–90. Springer, 2010.

[Sche13] A. Scherer. *Neuronale Netze: Grundlagen und Anwendungen*. Springer-Verlag. 2013.

[Scul10] D. Sculley. Web-scale k-means clustering. In *Proceedings of the 19th international conference on World wide web - WWW '10*, 2010, S. 1177.

[VaDH13] V. Vanhoucke, M. Devin und G. Heigold. Multiframe deep neural networks for acoustic modeling. Mai 2013.

[VaSe] V. Vanhoucke und A. Senior. Improving the speed of neural networks on CPUs.

[VeBG10] K. Veselý, L. Burget und F. Grézl. Parallel Training of Neural Networks for Speech Recognition. In P. Sojka, A. Horák, I. Kopeček und K. Pala (Hrsg.), *Text, Speech and Dialogue*, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg, S. 439–446. Citation Key: 10.1007/978-3-642-15760-8_56.

[WaLG15] Y. Wang, J. Li und Y. Gong. Small-footprint high-performance deep neural network-based speech recognition using split-VQ. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, April 2015, S. 4984–4988.

[WAWBC+94] M. Woszczyna, N. Aoki-Waibel, F. D. Buo, N. Coccaro, K. Horiguchi, T. Kemp, A. Lavie, A. McNair, T. Polzin, I. Rogina und andere. JANUS 93: Towards spontaneous speech translation. In *Acoustics, Speech, and Signal Processing, 1994. ICASSP-94., 1994 IEEE International Conference on*, Band 1. IEEE, 1994, S. I–345.

[WCBC+] N. Wang, J. Choi, D. Brand, C.-Y. Chen und K. Gopalakrishnan. Training Deep Neural Networks with 8-bit Floating Point Numbers. S. 10.