Bachelor thesis
Computer Science

Radboud University

Phonetic Classification in TensorFlow

Author:
Timo van Niedek
S4326164

First supervisor/assessor:
Prof. Dr. Tom Heskes
t.heskes@science.ru.nl

Second supervisor/assessor:
Prof. Dr. David van Leeuwen
david.vanleeuwen@gmail.com

June 29, 2016

Abstract

In this thesis, we present an implementation of two different neural network models for phonetic classification in TensorFlow. TensorFlow is an open-source machine learning library developed at Google. We used this library to create two different models: the Deep Feedforward Neural Network (DFNN) and the Deep Long Short-Term Memory (DLSTM) model. The advantage of using TensorFlow to create these models is that they are highly customizable – a feature that many speech recognition libraries lack. Therefore, the presented models are very suitable for further experimentation. We compare the performance of our models for a number of different settings. We report a frame error rate of 41.55% for the DFNN architecture, and 28.14% for the DLSTM architecture. The estimated phone error rates are 22.02% and 23.67% respectively.

Contents

Chapter 1

Introduction

Automatic Speech Recognition (ASR) is the process of translating spoken language into text using automated computer systems. This process has been a subject of research since the 1950s and has since seen many successful implementations such as voice search, dictation, robotics, and automatic translation.

Speech recognition is a challenging task (for more details, see Forsberg, 2003). Not all of the information that humans have is available to ASR systems. We comprehend speech using not only spoken words but also body language and common grammatical structures. Additionally, speech signals can vary in quality depending on the microphone used and changes in environmental noise. This phenomenon is called channel variability. Another problem is speaker variability: every speaker has a different voice, depending on anatomical qualities such as the shape of vocal tract, tongue and lips, gender and age. Furthermore, people tend to speak at different speeds depending on the topic, setting and mood. Regional and social dialects are other sources of speaker variability. Another challenge is posed by the co-articulation problem. Speech is continuous: there are no clear boundaries between sounds or words. Two sounds pronounced together sound different than when they are pronounced separately.

These problems make it difficult to find a common structure in every speech signal. We need to find such a structure in order to build a robust ASR system. Instead of trying to model the structure of speech manually, we can generate models and train them to find this structure and use it to predict what was spoken. One of the first successful speech models was developed at IBM in the 1980s, using Hidden Markov Models (HMMs) (Lee & Hon, 1989). These models have been used in conjunction with Gaussian Mixture Models (GMMs) in many ASR systems ever since. In the past decade, Deep Neural Networks (DNNs) gained popularity in ASR systems due to advances in learning algorithms and hardware. They are an alternative to GMMs and have been shown to outperform them (Hinton et al., 2012).

To create a speech recognition system, we first need to recognize the individual sounds before we can detect words and sentences. This task, called phonetic classification, is the process of determining for a small frame of speech which sound was spoken. DNNs are often used for phonetic classification because they can model the complex structures in speech data and thus recognize it.

There are several libraries specifically composed for ASR that implement DNNs, such as Kaldi (Povey et al., 2011) and RWTH ASR (Lööf et al., 2007; Rybach et

al., 2011). These packages contain predefined structures capable of performing ASR, where the flexibility in choosing different types of networks or low level parameters is limited. To gain new insights into creating models for speech recognition, we need to use libraries that allow many different models and tuning of parameters. One such library is TensorFlow, which was developed by researchers at Google and open sourced in november 2015 (Abadi et al., 2015). TensorFlow can be used to build deep learning systems for any task. Many types of networks can be created, thus allowing for more options for experimentation. In this thesis we show that it is possible to create a system for performing phonetic classification using TensorFlow. Specifically, our contributions are:

- To provide software to create and train a neural network for phonetic classification using TensorFlow. These networks are the DFNN and the DLSTM network, both of which may be several layers deep and have many hidden units per layer. The purpose of this software is to create a basis for building models for phonetic classification in TensorFlow.

- To test the performance of our networks on the TIMIT data set (Garofolo, Lamel, Fisher, Fiscus, & Pallett, 1993). TIMIT can be used as a benchmark data set for evaluating acoustic models. We provide a comparison of phone and frame error rates between different models.

Phonetic classification using TensorFlow has currently not been studied. This thesis therefore provides the first analysis of the performance of models created in TensorFlow on the TIMIT data set. However, classification of the TIMIT set has been a subject of research for several decades. In this thesis we implement modified versions of a randomly initialized DNN (Mohamed, Dahl, & Hinton, 2012) and an LSTM model (Graves, Mohamed, & Hinton, 2013; Sak, Senior, & Beaufays, 2014). These models have very different structures and were chosen because they show the flexibility and extensibility of TensorFlow.

In the next chapter, we further detail the task of phonetic classification. In chapter 3, we describe the data set TIMIT and how it was preprocessed. The architectures that were used in this thesis are explained in chapter 4 before showing how they were implemented in TensorFlow in chapter 5. Chapter 6 contains the description of our experiments and the results. We conclude this thesis in chapter 7.

Chapter 2

Phonetic Classification

The task of phonetic classification consists of determining for a segment of audio which *phones* were spoken. A phone is the smallest unit corresponding to a distinctive speech sound. They are independent of language, because they are distinguished purely based on sound. There is an important distinction between phones and *phonemes*. A phoneme is the smallest unit of sound which can be used to distinguish two words. They are dependent on a language, unlike phones. If two phones are realizations of the same phoneme, they are called *allophones*. It is not necessary for a speech recognition system to be able to distinguish two allophones, because they will never cause confusions between words.

Our goal was to create a system that can tell us which phones were spoken in a segment of audio. Neural networks are excellent at automatically performing this task. These neural networks start with no knowledge about the structure of audio. Therefore, at first they can do no better than randomly guessing the phones. By showing them examples of speech signals and the corresponding phones, they can be taught to find the structure in the signal, leading to more informed guesses. It has been shown previously that by showing a neural network enough examples, it can eventually recognize phones from new speech data.

The reason we want to build a system that can correctly recognize phones is that such a system lies at the core of many speech recognition systems. The output of a phonetic classifier can be used as an input for a word recognizer, for example. If we want to include a phonetic classifier in these systems – or whatever other applications may arise in the future – it needs to be accurate. Therefore, much effort has been put into improving the algorithms and neural network models for phonetic classification. Hinton et al. (2012) have provided a good, though already slightly outdated overview of the previous work in this field.

Chapter 3

Data

To train a neural network for recognizing phones, it needs to be shown many examples of audio segments and the corresponding phonetic annotations. For each example, it will predict the phonetic contents of the audio. The true phonetic annotations will then be compared to these guesses in order to determine how well the network performs. These examples form a training set, and the final evaluation of a system is based on a held-out test set. In this chapter, we describe TIMIT: the data set from which these examples were drawn. We also show how this data set was preprocessed to serve as input for neural networks.

3.1 The TIMIT Corpus

TIMIT is a speech data set designed for developing speech recognition systems (Garofolo et al., 1993). It is recorded by Texas Instruments (TI) and transcribed at the Massachusetts Institute of Technology (MIT). The set contains English sentences spoken by 630 speakers in eight dialects of American English. Additionally, time-aligned word and phonetic transcriptions are provided. The phonetic transcriptions are used in this thesis as the target labels for classification. TIMIT is split in a suggested testing and training division. There are no speakers that occur in both the testing and training set, with the motivation that if a model trained on the training set performs well on new speakers, it generalizes well.

There are three collections of sentences: SA, SI and SX. The two SA sentences were read by all speakers in order to expose their dialects. As is consistent with other similar studies (Lee & Hon, 1989; Mohamed et al., 2012; Graves, Mohamed, & Hinton, 2013), these were not included in our experiments to reduce bias towards these sentences.

3.2 Data Preparation

3.2.1 Speech Data

The TIMIT set contains recorded wave audio files of many pronounced sentences, called *utterances*. These are not directly suitable as input for a neural network because they contain too much information, making it difficult for a speech recognition system to capture the important information. In essence, a speech recognition system would

be confused by the sheer amount of data, most of which is not relevant for recognizing phones. The raw audio data is a waveform that is constantly changing. We can simplify this representation by assuming that, for a small frame of audio, the signal does not change significantly. A frame length of 10 ms is used in this thesis.

To extract the most relevant features from this framed waveform, it is usually converted to a mel-frequency cepstrum. This cepstrum represents the features of a speech spectrum that are the most relevant in human perception of speech. They were introduced by Davis and Mermelstein (1980) and have been used in state-of-the-art ASR systems ever since. We will not go into detail on how a mel-frequency cepstrum is generated; for more information see the work of Davis and Mermelstein. A mel-frequency cepstrum consists of a number of real-valued Mel-Frequency Cepstral Coefficients (MFCCs). We have used 13 MFCCs as input features that are generated using Kaldi's 's5' recipe, which is a standard recipe for converting TIMIT's audio files to MFCCs.

MFCCs are subject to noisiness. To reduce noisiness, it is common practice to normalize the coefficients. Mean and variance MFCC normalization has been shown to improve accuracy when classifying TIMIT (Shirali-Shahreza & Shirali-Shahreza, 2010). The MFCC data is represented by matrices of shape $f \times 13$, where f is the number of frames in the sentence. We normalized the coefficients using the formula

$$\hat{c}_{i,j} = \frac{c_{i,j} - \overline{c_{*,j}}}{\sigma(c_{*,j})}. \tag{3.1}$$

Here, i indexes over the frames and j indexes over the coefficients. The mean of one coefficient computed over the utterance is denoted by $\overline{c_{*,j}}$ and $\sigma(c_{*,j})$ is its standard deviation.

3.2.2 Phonetic Labels

```
0 9640 sil
9640 11240 sh
11240 12783 iy
12783 14078 hh
14078 16157 ae
16157 16880 dcl
16880 17103 d
17103 17587 y
17587 18760 er
18760 19720 dcl
```

Figure 3.1: Phonetic annotation for the beginning of `dr1faks0sa1.phn` ("She had your d(ark...)").

The target labels for classifying these MFCCs are phones. For each utterance in TIMIT, a phonetic annotation is provided. As an example, the first ten phones in the phonetic annotation file `dr1faks0sa1.phn` are shown in Figure 3.1. The two numbers denote sample in the wave audio file where one phone starts and ends. The wave audio files have a sample rate of 16000 samples per second.

Lee and Hon (1989) give a number of modifications to the set of phones that have since become standard. Therefore, these modifications have been applied. They mention 64 different phonetic labels. To clarify, these 64 labels include the three closure symbols that Lee and Hon use for folding silences into: `cl`, `vcl` and `sil`. There were some complications that arose when we followed their research, in that their phone table could not be used as-is (Lee & Hon, 1989, Table. 2-2). In this table the `ax-h` phone is missing, which does in fact occur in TIMIT's phonetic annotations. This phone was folded into its allophone `ah`. Examples of these phones are the 'u' in 'suspect' and the 'u' in 'but' respectively. The `#h` phone which appears in the phone table does not appear in any annotations in TIMIT, therefore it was not included in our class labels. In line with Lee and Hon's research, the glottal stops annotated with `q` have been removed from the labels. The MFCCs for the frames that contained a glottal stop have been removed as well.

There are fifteen phones in the set of labels that have an allophone. These phones have been folded into their allophones (again, in line with Lee and Hon (1989)). This leaves 48 different phonetic labels that are used as class labels for training. For evaluation, we have used a set further reduced to 39 labels. These reductions are based on the within-group confusions given by Lee and Hon (1989, p. 4). See table A.1 for the exact foldings that were used in this thesis.

The phonetic labels were first converted to scalar values and then to one-hot vectors. These vectors consist of 48 elements where all elements are zero except for the index of the label, which equals one. For example, the one-hot representation of the phone `ah` is $[0, 0, 1, 0, 0, \ldots, 0]$.

3.2.3 Matching Data & Labels

As described in section 3.2.1, the input data is the MFCCs of the TIMIT speech data. These files are matched by file name with the phonetic annotations (the `.ph` files) in the TIMIT data set. The MFCCs are generated per frame, but the phonetic labels are annotated based on the samples in the audio files, which have a sample rate of 16000 samples per second. Therefore, we converted the annotations to a list with for each frame the annotated phone. Because the annotations do not always cover the silence at the end of utterances, there are many utterances with more frames of MFCCs than annotated frames. In these cases, we removed the MFCCs for which there was no annotation provided.

Chapter 4

Neural Networks

After the data and labels are prepared, they are used to train a neural network for phonetic classification. There are a wide variety of neural networks that have been successfully applied for phonetic recognition. In this chapter, we give a short explanation of artificial neural networks in general, before we describe the architectures that were used in our experiments. These architectures are the Deep Feedforward Neural Network (DFNN) and a deep Recurrent Neural Network (RNN) containing Long Short-Term Memory cells (DLSTM).

4.1 Artificial Neural Networks

Artificial neural networks are models that can approximate complex problems whose structure is often not known. The basic building block for creating them is a neuron. A neuron is a unit which takes a number of real-valued inputs, and outputs an activation value based on some function over the inputs. The simplest such function is the linear activation function, found in linear neurons.

$$y = b + \sum_i x_i w_i$$

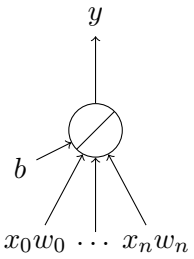

Figure 4.1: A simple linear neuron.

In Figure 4.1, x_i is the ith input and w_i its corresponding weight. The inputs for a neuron can come from some given input features such as speech data, or from the activation value of other neurons. The neurons receive an additional bias input b. One layer of a neural network consists of many of these neurons in parallel. A typical neural network consists of one input layer, a number of hidden layers, and an output layer. An artificial neural network is considered *deep* when it has more than one hidden layer.

4.1.1 Training a neural network

The weights and biases of each neuron can be trained in order to find values such that the whole network can capture patterns in the problem. The goal of training for classification is to find weights and biases such that for any input in the task domain, the outputs of the network are the probabilities for each class of the input belonging to that class. In phonetic classification, the input to the network is speech data, and the classes are the set of possible phones.

Training a neural network is done using the backpropagation algorithm (Rumelhart, Hinton, & Williams, 1988). In each training step, the network is presented with input features and the target or true output for those features. Using the difference between the output of the network and the true output, a loss function is calculated. The error derivatives are propagated backwards through the network. The weights and biases are then updated using these error derivatives, which approximate gradients of the weights. These updates are dependent on the learning rate, which controls the speed at which weights are learned.

We can also control when the weights are updated. We can update the weights after each training case, or after a number of training cases, called a *mini-batch*. The advantages of using mini-batch training are that we need to do less weight updates, and we can compute the gradients for different batches in parallel. By adapting the weights cumulatively, we save computational time. Usually, these mini-batches are picked randomly from the data set in order to have a balanced amount of examples for every class in the mini-batch.

4.1.2 Dropout Regularization

A common challenge when training neural networks is the problem of overfitting. The goal of training a neural network is to model the underlying relationship between the data and the labels. However, the examples that the neural network sees during training are only a small subset of the data that it would encounter in a real-world application setting. When we train a neural network to recognize speech, we can only feed it some utterances of which we hope that they are representative of all the possible utterances that it might encounter. Instead of modeling the relationship between the utterances and the phonetic contents, the neural network tries to fit the data that is seen as well as possible. This leads to overfitting: the neural network can accurately classify the utterances that it has seen during training, but does not generalize well.

One solution to overfitting is to average the output probabilities of multiple models. Since this is computationally expensive, an alternative method was introduced by Srivastava, Hinton, Krizhevsky, Sutskever, and Salakhutdinov (2014) – dropout. The core idea of dropout is to turn off random neurons and their connections during every training cycle. During testing, all neurons are kept on to create an average prediction over multiple thinned networks.

4.2 Deep Feedforward Neural Networks

A feedforward neural network is a type of artificial neural network with no cycles. All connections between neurons are from one layer to the next. The most basic

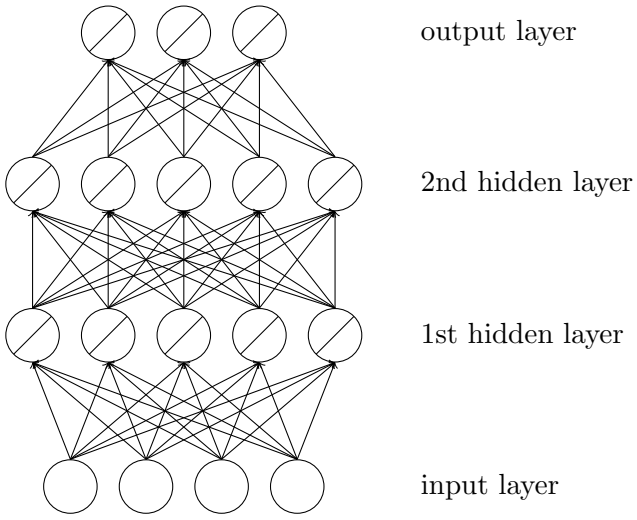

Figure 4.2: A deep feedforward neural network with two hidden layers and linear activations.

feedforward neural neural network consists of one layer of neurons. Deep feedforward neural networks are feedforward neural networks with a number of hidden layers. An example is given in Figure 4.2. They are better at modeling complex structures than simple feedforward neural networks. However, they are difficult to train due to their large number of parameters. Due to recent advances in training algorithms and computer hardware, deep feedforward neural networks with many large layers have become viable to use in many systems. We use deep feedforward networks in our experiments because they have been shown to perform well for phonetic classification: Mohamed et al. (2012) report a phone error rate of 20.7%.

4.3 Recurrent Neural Networks

Recurrent neural networks (RNNs) are neural networks that have cycles. These recurrent networks can be used to model sequential data, and are most effective when the output of one time step depends on the previous time steps. RNNs are a sensible choice for phonetic classification, because for each frame of speech data, the phonetic class probabilities depend on the previous frames. For example, if the previous frame contained silence, it is very likely that the frame at the current time step also contains silence. This dependency even extends beyond phone boundaries: the phonetic class probabilities of the current frame depend on the previous phone.

A recurrent layer typically sends its output activations not only to the next layer, but also to itself for the next time step. See Figure 4.3 for an example of an RNN. h_t^l denotes the hidden state for layer l at time t. The initial hidden state h_0^l of an RNN is a vector of zeros, and is updated at each time step. The input to the recurrent layer for time step t is x_t. In this example, the unrolled network can be thought of as multiple copies of one neural network that send information to the next copy. The weights of the connections are therefore identical in all the copies. Recurrent layers may be stacked on top of each other.

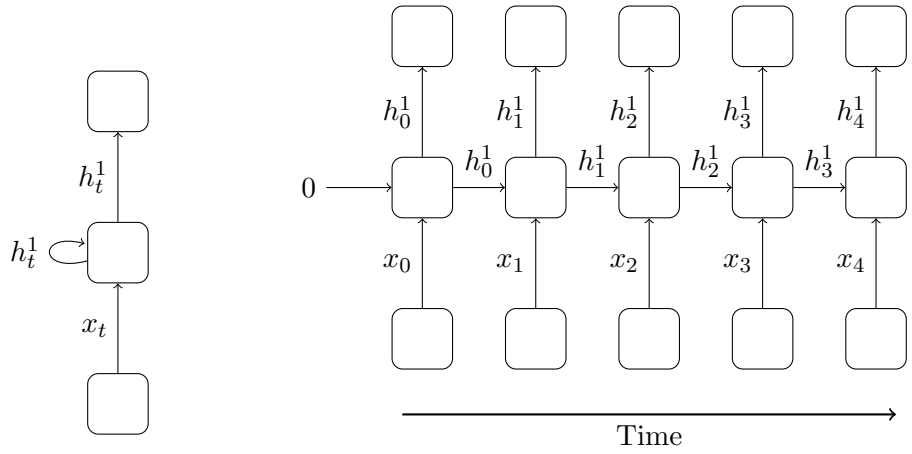

Figure 4.3: An RNN and its unrolled version for five time steps. The nodes each correspond to one layer with any number of neurons.

4.4 Long Short-Term Memory Networks

The backpropagated errors that flow through time in an RNN suffer from exponential vanishing or blow-up (Bengio, Simard, & Frasconi, 1994). Therefore, it is difficult to train RNNs on signals with long-term dependencies, such as speech. The long short-term memory (LSTM) cell, developed by Hochreiter and Schmidhuber (1997), is a recurrent cell that overcomes this problem. They introduced a complex cell with a number of components that together act similar to a memory cell. Inside one cell, multiple layers called *gates* are used. There are three gates: the input gate, the forget gate and the output gate. These gates are layers that are trained to make decisions on which values to update, which information should be forgotten, and which information should be output to the next time step.

There are quite some variants on the LSTM cell, such as adding peepholes (Gers & Schmidhuber, 2000), coupled input and forget gates, and the gated recurrent unit (GRU) (Cho et al., 2014). Greff, Srivastava, Koutník, Steunebrink, and Schmidhuber (2015) give a comparison of nine different LSTM architectures, and conclude that they perform roughly the same. Therefore, we use the basic LSTM cell whose implementation in TensorFlow is based on the work of Zaremba et al. (2014). This research also shows how dropout can be applied on RNNs with LSTM cells, namely between all connections that are not recurrent. The LSTM cell used in their research can be found in Figure 4.4. For the very first LSTM layer, h_t^{l-1} is simply the output of the layer below, which might not be recurrent.

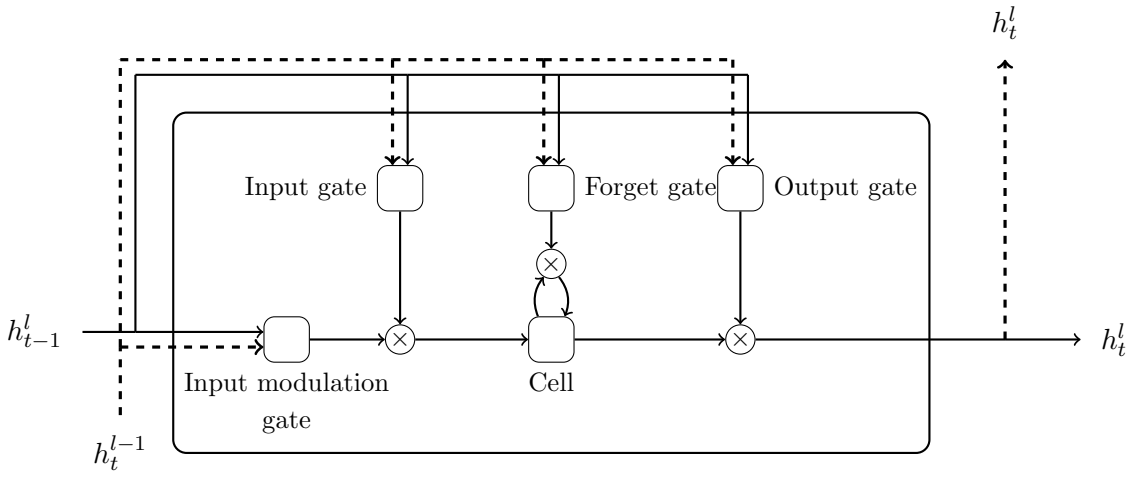

Figure 4.4: The LSTM cell. Diagram adapted from Zaremba et al., 2014. Dropout may be applied on the dashed connections.

Chapter 5

Implementation

We implemented the neural networks described in the previous chapter in TensorFlow. TensorFlow is a library that includes many operations for creating various artificial neural networks, training them and evaluating them (Abadi et al., 2015). These operations can be used to specify much of the functionality of the neural networks described in the previous chapter. However, creating an end-to-end neural network system for a specific task requires many design choices. TensorFlow provides the building blocks for creating neural networks and training them, but the actual implementation can take many different forms, and depends on the task that we want to accomplish. An overview of these building blocks and the implementation details of our architectures are described in this chapter.

5.1 Concepts of TensorFlow

Many details of how TensorFlow works can be found in the white paper by Abadi et al. (2015). We briefly highlight the most important concepts that TensorFlow uses.

A machine learning system can be represented in TensorFlow using the data flow graph. This is a directed graph where the nodes contain computations and the edges are the flow of *tensors* through this graph. Tensors are mathematical objects that can be described using an n-dimensional array. They are the primary data type in TensorFlow: they are used to store data which can be transformed by *operations*. These operations describe the actual functionality of the computation, and an instantiation of an operation corresponds to a node in the data flow graph. The input and output of operations are zero or more tensors.

To create a machine learning system in TensorFlow, it needs to be expressed as a data flow graph. This data flow graph can then be interacted with using a *session*. The most important function of a session is to run the data flow graph. When running a computation, a dictionary of inputs is fed to the graph. The graph then executes the operations, and outputs the result of the final operation.

Neural networks can be specified in TensorFlow as a data flow graph. A neural network layer is created by adding ops to the graph that multiply the weights with the inputs, add the biases and perform some activation function over the result. When we specify these operations, nothing has been computed yet. The computation starts when the output tensor is evaluated (or the last operator is run) for a given input.

The weights can be trained in a loop which evaluates the output of the neural network for a mini-batch, computes the loss for that batch and finally runs an optimization algorithm which tries to minimize that loss. The state of the tensors is not maintained across different evaluations. To maintain the state of the weights and biases during training, they are stored in *variables*.

5.2 Architectures

Data flow graphs representing neural networks can be created and trained using the operations included in TensorFlow. The architecture of the neural networks that we used are a Deep Feedforward Neural Network (DFNN) and a deep LSTM network (DLSTM). The data flow graphs for these architectures are created based on a number of parameters which are passed to our software in the command line. We provided default values for these parameters. These default values were found through experimentation or stem from previous research. The exact parameters for the networks are described in section 5.2.1 and 5.2.2.

Some of the implementation details and parameters were the same for both architectures. Both the DFNN and the DLSTM have multiple hidden layers. These hidden layers are fully connected to the next layer. Dropout can be applied on these connections.

To convert the outputs of the last layer to class probabilities, the softmax function is applied:

$$p_i = \frac{e^{x_i}}{\sum_j e^{x_j}}.$$

This function converts the output values such that they add up to one, where p_i is the class probability of class i, x_i is the output of the ith neuron in the output layer, and j iterates over the outputs in this layer.

We used the cross-entropy loss function

$$C = -\sum_i d_i \ \log p_j,$$

where C is the cross-entropy loss and d_i is the target probability for the ith class given by the true label. The true labels are, as described in section 3.2.2, represented by one-hot vectors. These vectors represent the ideal class probability distribution. Therefore, the target probability for the true class is 1, and 0 for the other classes. The cross entropy loss is a standard way of measuring the quality of a network in phonetic classification when using the softmax output function (Hinton et al., 2012).

The objective for an optimization function is to minimize the loss. We used two different optimization functions: gradient descent and the more sophisticated Adam optimizer (Kingma & Ba, 2014). We used gradient descent because it is commonly used in phonetic classification (Mohamed et al., 2012; Graves, Mohamed, & Hinton, 2013; Sak et al., 2014). Adam optimization was chosen as an alternative because, according to Kingma and Ba (2014), it performs well on many optimization problems. A default learning rate of 0.02 was used for gradient descent (Mohamed et al., 2012), and 0.001 for Adam optimization (Kingma & Ba, 2014). We trained the networks with 15 epochs. After every epoch, the data set was shuffled.

5.2.1 Deep Feedforward Neural Network

The first architecture that we created is a Deep Feedforward Neural Network (DFNN), as was described in Section 4.2. The inputs for the DFNN are the MFCCs of one frame and the context of that frame – a number of frames before and after the frame. For the very start and end of an utterance, the context extends beyond the frames that are available. This is remedied by filling the missing context with the first or last frame for the start and the end of the utterance respectively.

The MFCCs for the context of the frame are used as additional input features. Early experiments showed that using context as additional features improved the accuracy of the network. The number of frames of the context was therefore kept constant for all further experiments; 11 frames of context (5 before, 5 after) were used because more frames do not yield a significantly better result according to the research of Mohamed et al. (2012).

Following their research, we set the number of neurons in each hidden layer to be equal. In this research, randomly initialized networks are compared with generatively pretrained networks. The weights of the pretrained networks are initialized by first training a multilayer Restricted Boltzmann Machine (RBM). This unsupervised learning phase is not necessary when we use Rectified Linear Units (ReLUs). These output a linear activation if the sum of the inputs is more than 0, and otherwise output 0. According to the experiences of Zeiler et al. (2013), ReLU activations combined with random weight initialization perform well enough to eliminate the need for generative pretraining. Therefore, to simplify the training of the DFNNs, we used random activations for the ReLU units instead of generative pretraining.

The weights are initialized by drawing random values from a truncated normal distribution with mean 0 and standard deviation 0.1. One run with a higher standard deviation for the weight initialization resulted in a much poorer performance. The biases are initially set to 0.1 instead of 0 to avoid dead neurons at the start of training.

If dropout regularization is applied to the network, the probability of keeping units during training is 0.5 for the hidden layers. This probability is close to optimal according to Srivastava et al. (2014). No dropout is applied to the inputs. During testing, the probability of keeping units is set to 1 to ensure that the entire network is used for evaluation.

The mini-batch size during training was 128, in line with Mohamed et al. (2012). We tried much smaller and much larger mini-batches, which did not lead to improved accuracy.

Thus, the parameters of the DFNN are

- the mini-batch size (default: 128),
- the context size (default: 11),
- whether to apply dropout regularization (default: yes),
- the number of training epochs (default: 15),
- the number of layers (default: 3),
- the number of units in the hidden layers (default: 1024),

- the optimization algorithm (default: Adam).

If the default values are used, a data flow graph describing a network with rectified linear activations is created with 143 input features (13 MFCCs times 11 frames of context), three fully connected layers with 1024 units on which dropout is applied, and a final layer with size 48 corresponding to the phonetic classes.

5.2.2 Deep Long Short-Term Memory Network

The DLSTM network was implemented as an alternative structure to the DFNN. The DLSTM network is a multilayer RNN with LSTM cells. For an explanation of RNNs and LSTM cells, we refer to Sections 4.3 and 4.4. The LSTM cells have two inputs: the hidden state of this layer at the previous time step (h_{t-1}^{l}) and the hidden state of the layer below at this time step (h_{t}^{l-1}). The hidden states are maintained by TensorFlow throughout the time steps by a single tensor of size $2 \times n_h \times l$ where n_h is the number of units in the gate layers and l is the number of LSTM layers. The initial hidden state h_0^l is specified as a parameter when creating an RNN. We initialize the hidden states with a tensor containing zeros.

The inputs to the DLSTM network are, as for any RNN, sequences. In many implementations of RNNs, these sequences can be of any length. Then, the neural network is trained by dynamically unrolling the RNN and backpropagating the gradients through time. Unfortunately, TensorFlow currently does not support dynamic unrolling of recurrent layers. RNNs can only be trained for a fixed sequence length, which is problematic because the utterances have many different lengths (between roughly 3 to 8 seconds).

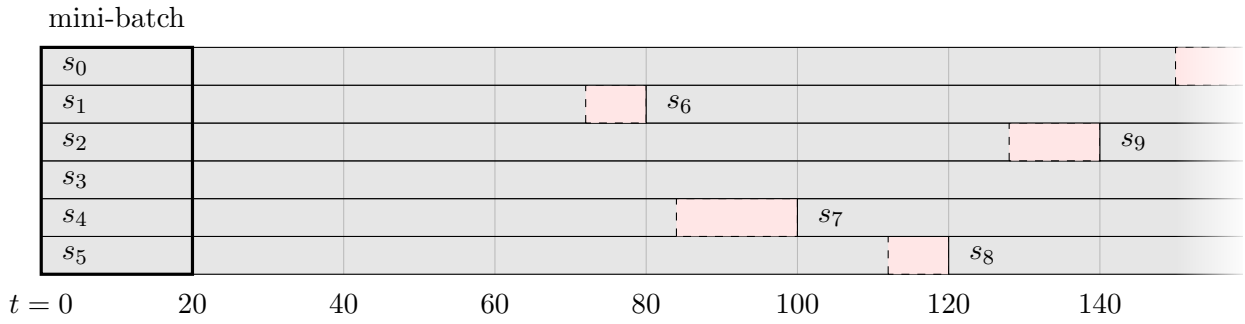

Figure 5.1: Method of mini-batching using padded sub-sequences.

The solution was to split the utterances into sub-sequences of the same length. This strategy was found by Sak et al. (2014), who use a sub-sequence length s of 20 frames. Each utterance is padded by repeating the last frame such that the number of frames is a multiple of the sub-sequence size. Mini-batches are created by iteratively retrieving sub-sequences for b utterances, where b is the mini-batch size. See Figure 5.1 for a schematic overview of this process. The padded frames are highlighted in red.

The utterances might not have the same amount of sub-sequences. Whenever the end of an utterance is reached, it is replaced in the next iteration by a new utterance.

At the end of each mini-batch, the state is preserved for the utterances that contain more sub-sequences and the state is reset to zeros for the utterances that are replaced in the next mini-batch. We used a mini-batch size b of six sub-sequences with 20 frames to roughly match the batch size in the DFNN experiments: the total amount of frames in one batch is $6 \cdot 20 = 120$ for the DLSTM and 128 for the DFNN.

This results in a three-dimensional input tensor of shape $b \times s \times n_i$, where n_i is the number of input features. The input features are the 13 MFCCs for one frame without context. The first layer of our DLSTM network is a fully connected feedforward layer with n_h ReLU units. To prepare the input tensor for this layer, it is reshaped into a two-dimensional tensor of shape $bs \times n_i$. The time-dimension is removed because this layer is not recurrent. The output of the fully connected layer is a tensor of $bs \times n_h$. This tensor is converted into a list with s elements containing tensors of shape $b \times n_h$: a list of inputs to the LSTM layers for each time step.

Dropout is applied to each LSTM layer by dropping the non-recurrent connections with a probability of 0.8 during training (Zaremba et al., 2014). Again, the probability of keeping nodes during testing was set to 1 during testing. The LSTM layers were stacked to create multiple layers. The output of each layer becomes the input of the next: they are not fully connected. For simplicity, there is an equal number of hidden units in each LSTM layer.

The parameters of the DLSTM are

- the mini-batch size (default: 6),
- the sub-sequence length (default: 20),
- whether to apply dropout regularization (default: yes),
- the number of LSTM layers (default: 3),
- the number of units in the LSTM layers (default: 250),
- the optimization algorithm (default: Adam).

Chapter 6

Experiments

In this chapter, we present the experimental setup and results for the Deep Feed-forward Neural Network (DFNN) and Deep LSTM (DLSTM) architectures. We describe the evaluation method first because it is identical for both architectures.

6.1 Evaluation Method

In many speech recognition systems, a DNN is used to output probabilities for Hidden Markov Model states (Hinton et al., 2012). The HMMs are used to compute where one phone ends and the next phone starts. These alignments are used to convert the framewise classifications to phonetic classifications. The standard evaluation metric for phonetic classification is the phone error rate, which is the percentage of phones that were guessed incorrectly.

HMMs are currently not implemented in TensorFlow, which is why we did not include these in our research. This means that our systems are not capable of performing phone-level classification, but only frame-level classification. To compare our results with other research, we give an estimate of how well our networks would perform using the phone error rate. This estimate is found by using the true alignments as annotated in TIMIT's phonetic annotation files. For each phone, we compute the mean of the class probabilities of the frames between the phone's start and end time. Then, the class with the highest average probability is picked as the prediction for that phone. The phone error rate is computed between these predictions and the true phone labels.

This phone error rate is slightly lower than a system including an HMM would achieve. Essentially, our phone error rates are computed as if we would have a perfect HMM, which finds the alignments perfectly every time. We caution the reader not to compare our phone error rates with other research as conclusive results. Instead, these error rates are meant as an estimate of the performance of our networks, and as an comparison of performance between the experiments that are presented in this thesis. We include the frame error rate for each experiment as a fair metric for performance.

6.2 Deep Feedforward Neural Network

The parameters for the baseline experiment using DFNNs followed from the research of Mohamed et al. (2012). For the following experiments, we modified one parameter from this baseline per run, such that we could isolate which parameters led to improvement. If a modification led to a significant improvement, we kept it in the following tests.

6.2.1 Experiments

Baseline experiment

Frame error rate (Training set)	Frame error rate (Test set)	Phone error rate (Test set)
39.16%	47.68%	31.02%

Table 6.1: Performance of the DFNN using the baseline settings – gradient descent with learning rate 0.02, 3 layers with 1024 units and dropout regularization with 0.5 probability of keeping units.

Mohamed et al. (2012) concluded that a context size of 11, 17 or 27 frames was optimal. They found that a layer size of 1024 performed well enough – adding more units did not improve the accuracy significantly. We used three layers in the baseline experiment. Gradient descent with a learning rate of 0.02 was used as the optimization algorithm. In line with Mohamed et al., a mini-batch size of 128 was used. We deviated from their research by including dropout regularization and ReLU units with random weight initialization to replace the generative pretraining. See Table 6.1 for the results of this experiment.

Adam optimization

Learning rate	Frame error rate (Training set)	Frame error rate (Test set)	Phone error rate (Test set)
0.001	36.55%	45.58%	29.13%
0.0001	33.83%	43.51%	25.80%

Table 6.2: Performance of the DFNN using Adam optimization.

We modified the baseline experiment to use Adam optimization (Kingma & Ba, 2014) instead of gradient descent. The other parameters were left the same. The default learning rate for Adam optimization is 0.001. Using this learning rate, we achieved only slightly lower frame and phone error rates than the baseline experiment. In another run, we lowered the learning rate to 0.0001, which led to an even lower frame and phone error rate on the TIMIT core test set. Table 6.2 show the results of these experiments. For the following experiments, we used a learning rate of 0.0001.

Dropout

Probability of keeping units	Frame error rate (Training set)	Frame error rate (Test set)	Phone error rate (Test set)
1 (no dropout)	6.93%	47.30%	25.11%
0.8	23.99%	41.55%	22.02%
0.5 (see Table 6.2)	33.83%	43.51%	25.80%

Table 6.3: Performance of the DFNN with respect to the dropout probability using Adam optimization.

We modified the probability that a unit is kept in the network during dropout regularization in these experiments. The suggested probability is 0.5 (Srivastava et al., 2014). Intuitively, if this probability is raised, the amount of overfitting will increase, meaning that the frame error rates on the test and training set will be further apart. This is supported by the results in Table 6.3: the difference between frame error rates is 9.68% when the probability is 0.5, 17.56% when it is 0.8 and 40.37% when no dropout is applied.

It is however notable that when we applied dropout with probability 0.5, the frame and phone error rates on the test set are higher than when we used a probability of 0.8. Usually, when more regularization is applied, it is expected that the performance on unseen data should rise because the network should be able to generalize better. The reason why our tests did not reflect this, might lie in the fact that too much dropout causes the network to be too thin. It is possible that not enough units are retained in the test with 0.5 probability to properly fit the speech data. Because a probability of 0.8 led to the best results on the test set, we used this probability in further runs.

More layers

Number of layers	Frame error rate (Training set)	Frame error rate (Test set)	Phone error rate (Test set)
3 (see Table 6.3)	23.99%	41.55%	22.02%
5	27.34%	41.52%	22.70%

Table 6.4: Performance of the DFNN with respect to the number of layers. All layers had 1024 units.

We increased the number of layers in this experiment from three to five layers with 1024 units. We used a probability of keeping nodes during dropout of 0.8 and Adam optimization with learning rate 0.0001. These last two settings lead to a better performance in previous experiments.

Mohamed et al. (2012) point out that adding more layers leads to a better performance. In their research, the phone error rate increased by roughly 0.2% when increasing the number of layers from three to five. The results in Table 6.4 did not

show the same behavior. The frame error rate decreases very slightly, while the phone error rate increases. According to Mohamed et al., generative pretraining is necessary to benefit from having many hidden layers. Replacing the generative pretraining by ReLU units and random weight initialization could be the reason why adding more layers does not decrease the phone error rate.

Larger layers

Number of units per layer	Frame error rate (Training set)	Frame error rate (Test set)	Phone error rate (Test set)
1024 (see Table 6.3)	23.99%	41.55%	22.02%
2048	21.86%	41.57%	21.79%

Table 6.5: Performance of the DFNN with respect to the layer size.

We increased the number of units per layer from 1024 to 2048 for all three layers. This led to a very small improvement for the phone error rate, and an insignificant improvement for the frame error rate. The results from Table 6.5 are in line with the conclusion of Mohamed et al. (2012): adding more than 1024 units has little effect on the performance.

6.3 Deep LSTM Network

In the experiments of Graves, Mohamed, and Hinton (2013), a three layer LSTM network with 250 units per cell resulted in the lowest error rate. Their setup is different from ours in a number of ways. The LSTM network is bi-directional, meaning that it is not only trained forwards through time, but also backwards. Creating a bi-directional LSTM network is beyond the scope of this paper: we only use a forwards LSTM layers. Furthermore, Graves, Mohamed, and Hinton initialized their best performing network with the weights of another network. Instead, we use TensorFlow's default initialization, which stems from the research of Zaremba et al. (2014).

We used a mini-batch size of 6 sub-sequences of 20 frames, as described in section 5.2.2. We applied dropout regularization in every experiment. The non-recurrent hidden layer has 1024 ReLU units.

6.3.1 Experiments

Baseline experiment

In the baseline experiment, we used three LSTM layers with 250 units per cell. We used Adam optimization with learning rate 0.001 as an optimization algorithm because the DFNN experiments had shown that it performed better than gradient descent. Dropout regularization was applied using a probability of 0.8 to keep nodes for every training step. See Table 6.6 for the results.

Frame error rate (Training set)	Frame error rate (Test set)	Phone error rate (Test set)
25.32%	29.43%	25.36%

Table 6.6: Performance of the DLSTM using the baseline settings – Adam optimization with learning rate 0.001, three LSTM layers with 250 units and dropout regularization with 0.5 probability of keeping units.

Gradient descent

Frame error rate (Training set)	Frame error rate (Test set)	Phone error rate (Test set)
26.06%	28.14%	23.67%

Table 6.7: Performance of the DLSTM using gradient descent.

We replaced the Adam optimization algorithm with gradient descent. In this experiment, we used a learning rate of 0.02, which is the same as in the DFNN experiments. The results are shown in Table 6.7.

More units

Frame error rate (Training set)	Frame error rate (Test set)	Phone error rate (Test set)
22.52%	29.76%	24.89%

Table 6.8: Performance of the DLSTM with twice as much units per layer and Adam optimization.

In this experiment, we used 500 units for each of the three LSTM layers. As can be seen in Table 6.8, the frame error rate on the test set and the phone error rate were very close to the baseline experiment. Therefore, adding more hidden units does not significantly improve the accuracy of the DLSTM model.

6.4 Results

We briefly summarize the results from the experiments using the DFNN and DLSTM architectures. Training the DFNN with Adam optimization yielded a better performance than training with gradient descent. In contrast, gradient descent performed better for the DLSTM networks. Increasing the dropout probability from the suggested value of 0.5 (Srivastava et al., 2014) to 0.8 led to improved performance for the DFNN. Adding more units per layer did not decrease the error rates significantly for both architectures. Adding more layers hurt performance of the DFNN slightly.

The best performance for the DFNN model was found using three layers with 1024 units, Adam optimization and a dropout probability of 0.8. The frame error rate for this run is 41.55% and the phone error rate is 22.02%. One run with 2048 units in the hidden layers achieved a phone error rate of 21.79%, which is slightly lower. However, since the phone error rate is an estimate, we consider the network with the lowest frame error rate to have the best overall performance. A state-of-the-art phone error rate of 20.7% is given by Mohamed et al. (2012).

The DLSTM model that performed best had three LSTM layers of 250 units and was trained using gradient descent. The frame error rate is 28.14% and the estimated phone error rate is 23.67%. The frame error rate is roughly equal to the reported 28.2% for the bi-directional deep LSTM network by Graves, Jaitly, and Mohamed (2013). The phone error rate is 6% higher than the error rate of 17.7% found by a similar bi-directional LSTM network interfaced with an HMM (Graves, Mohamed, & Hinton, 2013).

Chapter 7

Conclusions and Future Work

In this thesis we have shown that it is possible to create systems for phonetic classification in TensorFlow. We created two different systems: a deep feedforward neural network and a recurrent neural network with LSTM cells. The best framewise performance on the TIMIT core test set was achieved by a deep LSTM network with three LSTM layers with 250 hidden units, trained using gradient descent. This network had a frame error rate of 28.14% and an estimated phone error rate of 23.67%. We therefore believe that the DLSTM network is the most promising of the two networks. However, using the trained DFNN network for phonetic classification costs less computation time than the DLSTM network. The main issue is that the unrolled DLSTM networks take sub-sequences of 20 frames as input. Therefore, the sequences first need to be split in sub-sequences before they can be classified. The input for the DFNN (11 frames of MFCCs) can be prepared faster than the input for the DLSTM.

The DLSTM experiments took roughly seven days to complete using a single Intel CPU E7-4870 v2 processor. We trained the networks for 15 epochs. Due to a lack of computational power and time, we could not include many experiments. Therefore, it is possible that different settings could improve accuracy. The main purpose of this thesis is however to provide a basis for further customization and experimentation, wherein our preliminary experiments suggest a number of settings that lead to a good performance.

Even though we implemented many features for training DFNN networks or DLSTM networks, there are indeed still many more extensions possible. The most obvious next step is to interface our networks with an HMM to create an actual end-to-end phonetic classification system. Another possible extension is adding a backwards layer in the DLSTM, making it a bi-directional LSTM network (Graves, Mohamed, & Hinton, 2013). Due to the scope of this thesis and computational limitations, many combinations of parameters could not be tested. This was especially problematic when training the DLSTM networks: these experiments often took several days to complete.

Another topic of further research is to create a Convolutional Neural Network (CNN) for phonetic classification. CNNs are included in TensorFlow, and have been shown to be applicable for phonetic classification (Abdel-Hamid, Mohamed, Jiang, & Penn, 2012). This recent development is an interesting one, since these networks are actually mostly used for tasks related to computer vision. Implementing

these networks in TensorFlow for phonetic recognition should offer a new basis for experimentation – one that has not been explored as much as DFNNs and RNNs yet.

The source code for this project can be found at `https://github.com/TimovNiedek/timit_tf`.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., ... others (2015). Tensorflow: Large-scale machine learning on heterogeneous systems. *Software available from tensorflow. org*.

Abdel-Hamid, O., Mohamed, A.-r., Jiang, H., & Penn, G. (2012). Applying convolutional neural networks concepts to hybrid nn-hmm model for speech recognition. In *Acoustics, speech and signal processing (icassp), 2012 ieee international conference on* (pp. 4277–4280).

Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *Neural Networks, IEEE Transactions on*, *5*(2), 157–166.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

Davis, S. B., & Mermelstein, P. (1980). Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, *28*(4), 357–366.

Forsberg, M. (2003). Why is speech recognition difficult. *Chalmers University of Technology*.

Garofolo, J. S., Lamel, L. F., Fisher, W. M., Fiscus, J. G., & Pallett, D. S. (1993, February). DARPA TIMIT acoustic-phonetic continous speech corpus CD-ROM. NIST speech disc 1-1.1. *NASA STI/Recon Technical Report N*, *93*.

Gers, F. A., & Schmidhuber, J. (2000). Recurrent nets that time and count. In *Neural networks, 2000. ijcnn 2000, proceedings of the ieee-inns-enns international joint conference on* (Vol. 3, pp. 189–194).

Graves, A., Jaitly, N., & Mohamed, A.-r. (2013). Hybrid speech recognition with deep bidirectional lstm. In *Automatic speech recognition and understanding (asru), 2013 ieee workshop on* (pp. 273–278).

Graves, A., Mohamed, A.-r., & Hinton, G. (2013). Speech recognition with deep recurrent neural networks. In *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on* (pp. 6645–6649).

Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., & Schmidhuber, J. (2015). Lstm: A search space odyssey. *arXiv preprint arXiv:1503.04069*.

Hinton, G., Deng, L., Yu, D., Dahl, G. E., Mohamed, A.-r., Jaitly, N., ... others (2012). Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *Signal Processing Magazine, IEEE*, *29*(6), 82–97.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural*

computation, *9*(8), 1735–1780.

Kingma, D., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Lee, K.-F., & Hon, H.-W. (1989). Speaker-independent phone recognition using hidden markov models. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, *37*(11), 1641–1648.

Lööf, J., Gollan, C., Hahn, S., Heigold, G., Hoffmeister, B., Plahl, C., ... Ney, H. (2007). The rwth 2007 tc-star evaluation system for european english and spanish. In *Interspeech* (pp. 2145–2148).

Mohamed, A.-r., Dahl, G. E., & Hinton, G. (2012). Acoustic modeling using deep belief networks. *Audio, Speech, and Language Processing, IEEE Transactions on*, *20*(1), 14–22.

Povey, D., Ghoshal, A., Boulianne, G., Burget, L., Glembek, O., Goel, N., ... others (2011). The kaldi speech recognition toolkit. In *Ieee 2011 workshop on automatic speech recognition and understanding.*

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1988). Learning representations by back-propagating errors. *Cognitive modeling*, *5*(3), 1.

Rybach, D., Hahn, S., Lehnen, P., Nolden, D., Sundermeyer, M., Tüske, Z., ... Ney, H. (2011). Rasr-the rwth aachen university open source speech recognition toolkit. In *Proc. ieee automatic speech recognition and understanding workshop.*

Sak, H., Senior, A., & Beaufays, F. (2014). Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition. *arXiv preprint arXiv:1402.1128*.

Shirali-Shahreza, M. H., & Shirali-Shahreza, S. (2010). Effect of mfcc normalization on vector quantization based speaker identification. In *Signal processing and information technology (isspit), 2010 ieee international symposium on* (pp. 250–253).

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, *15*(1), 1929–1958.

Zaremba, W., Sutskever, I., & Vinyals, O. (2014). Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*.

Zeiler, M. D., Ranzato, M., Monga, R., Mao, M., Yang, K., Le, Q. V., ... others (2013). On rectified linear units for speech processing. In *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on* (pp. 3517–3521).

Appendix A

List of phones

Phone	Example	Training	Evaluation	Phone	Example	Training	Evalutaion
aa	b **o** tt	aa	aa	m	**m** o **m**	m	m
ae	b **a** t	ae	ae	n	**n** oo **n**	n	n
ah	b **u** t	ah	ah	ng	si **ng**	ng	ng
ao	b **ou** ght	ao	aa	nx	wi **nn** er	n	n
aw	b **ou** t	aw	aw	ow	b **oa** t	ow	ow
ax	**a** bout	ax	ah	oy	b **oy**	oy	oy
ax-h	s **u** spect	ah	ah	p	**p** ea	p	p
axr	butt **er**	er	er	r	**r** ed	r	r
ay	b **i** te	ay	ay	s	**s** ea	s	s
b	**b** ee	b	b	sh	**sh** e	sh	sh
ch	**ch** oke	ch	ch	t	**t** ea	t	t
d	**d** ay	d	d	th	**th** in	th	th
dh	**th** en	dh	dh	uh	b **oo** k	uh	uh
dx	mu **dd** y, dir **t** y	dx	dx	uw	b **oo** t	uw	uw
eh	b **e** t	eh	eh	ux	t **oo** t	ux	uw
el	bott **le**	el	l	v	**v** an	v	v
em	bott **om**	m	m	w	**w** ay	w	w
en	butt **on**	en	n	y	**y** acht	y	y
eng	wash **ing** ton	ng	ng	z	**z** one	z	z
er	b **ir** d	er	er	zh	a **z** ure	zh	sh
ey	b **ai** t	ey	ey	bcl	(b-closure)	vcl	sil
f	**f** in	f	f	dcl	(d-closure)	vcl	sil
g	**g** ay	g	g	gcl	(g-closure)	vcl	sil
hh	**h** ay	hh	hh	kcl	(k-closure)	cl	sil
hv	a **h** ead	hh	hh	pcl	(p-closure)	cl	sil
ih	b **i** t	ih	ih	qcl	(q-closure)	cl	sil
ix	deb **i** t	ix	ih	tcl	(t-closure)	cl	sil
iy	b **ee** t	iy	iy	q	(glottal stop)		
jh	**j** oke	jh	jh	epi	(epenthetic silence)	epi	sil
k	**k** ey	k	k	pau	(pause)	sil	sil
l	**l** ay	l	l	h#	(begin/end marker)	sil	sil

Table A.1: List of used phones. Underlined phones are folded.