Karlsruhe Institute of Technology

Phoneme classification and alignment through recognition on TIMIT

Bachelor's Thesis of

Linus Schilpp

at the Department of Informatics
Institute for Anthropomatics and Robotics
Interactive Systems Lab

Reviewer:	Prof. Dr. Alexander Waibel
Second reviewer:	Prof. Dr.-Ing. Tamim Asfour
Advisor:	M.Sc. Christian Huber
Second advisor:	M.Sc. Juan Hussain

01. July 2021 – 31. October 2021

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.
Karlsruhe, 31.10.2021

L. Schilpp
(Linus Schilpp)

Abstract

In this work we explore a hybrid between ANNs and DTW for phoneme alignment on the TIMIT dataset. The idea is to use the output probabilities of a neural phoneme recognition model together with a probability-based DTW in order to align phonemes.
For phoneme recognition we achieve 18.1% FER which is an 4.0% improvement over the state-of-the-art.
Our alignment results in a 86.3% phoneme boundary accuracy with a 20ms tolerance.
Furthermore phoneme classification based on recordings of single phonemes is being tried resulting in an accuracy of 66.68%.
Apart from that we introduce the CyclicPlateauScheduler, a new learning rate scheduler combining triangular cyclic learning rates with ReduceLROnPlateau.

Zusammenfassung

In dieser Arbeit experimentieren wir mit Machine-Learning-Modellen für phonetische Ausrichtung von Text und Audio, die aus neuronalen Netzen und Dynamic Time Warping (DTW) zusammengesetzt sind. Diese werden auf dem TIMIT dataset ausgewertet. Ausgabewahrscheinlichkeiten eines neuronalen Phonemerkenners werden an einen wahrscheinlichkeitsbasierten DTW Algorithmus übergeben, der die Phoneme ausrichtet.

Für Phonemerkennung erreichen wir eine Frame Error Rate (FER) von 18.1%, was eine Verbesserung um 4.0% gegenüber dem state-of-the-art ist. Unsere Ausrichtung resultiert in einer Phonemgrenzengenauigkeit von 86.3% mit 20ms Toleranz. Desweiteren erhalten wir eine Genauigkeit von 66.7% für Phonemklassifikation basierend auf Audioaufnahmen von einzelnen, getrennten Phonemen. Abgesehen davon führen wir den CyclicPlateauScheduler ein, der zyklische Lernraten mit der ReduceLROnPlateau Technik in einen Lernraten-Scheduler kombiniert.

Contents

List of Figures

List of Tables

1 Introduction

The motivation for the experiments carried out in this thesis, is to create a system, capable of highlighting the currently spoken text segment in a transcript for a playing audiofile.
Several partially conflicting goals were aimed for:

- **High accuracy:** The system should be able to highlight the correct segment
- **Short runtime:** The system should be lightweight and fast
- **Multi-language:** The system should be able to highlight transcripts of multiple languages, preferably also unseen ones to a certain degree

In order to reduce dependence on a certain language, we decided to create a phoneme-based alignment system, as this makes it less dependent on a single language than word based systems. Our general approach to this problem is to create a phoneme recognition system, which includes timing information and then use the output of this system in order to align the phonemes to the transcript. Initially, we try out phoneme classification based on recordings of single phonemes. Then we evaluate Transformer and Recurrent neural network (RNN) based models on recordings of a whole sentence. Building on that, we create a phoneme alignment system using probability-based Dynamic Time Warping (DTW), which is then refined using custom cost functions and a weighted loss, in order to increase its accuracy.

Common approaches to align phonemes are Hidden Markow Model (HMM) based approaches and neural networks and also hybrids between them. Solely HMM based models usually do not achieve as high accuracies as hybrids between neural networks and HMMs.

Our work is structured in the following way:

First we outline the basic neural network architectures and other techniques we build on in chapter 2. Next we take a look at the TIMIT dataset in chapter 3.
After that we define the tasks and how performance is measured in chapter 4.
Then we summarize related work in chapter 5. Subsequently we explain our models and training methods in chapter 6 and evaluate them in chapter 7.
Finally we conclude with closing remarks in chapter 8.

2 Fundamentals

2.1 Audio processing

2.1.1 Waveform and spectrograms

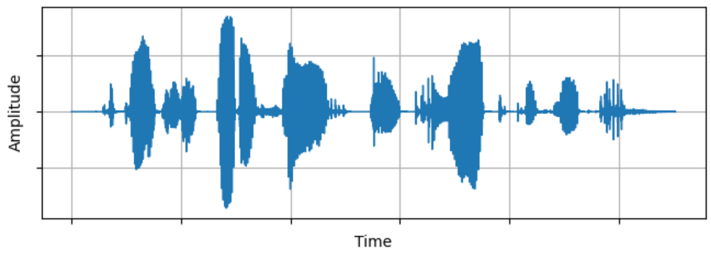


Figure 2.1: Waveform of the sentence "The gorgeous butterfly ate a lot of nectar."

While a raw audio waveform is a graphical representation of the shape of a wave [46], it is often advantageous to convert them into a spectrogram, which resembles the frequency mapping in the human ear [21]. A spectrogram is a visual representation of the loudness of a signal at different frequencies over time.[47] Humans perceive the difference of loudness between two pitches on a logarithmic scale compared to frequency. The mel-scale has been modeled based on this perception and is given by the formula:

$$m = 2595 \cdot log_{10}(1 + \frac{f}{700})$$

where m denotes the value in the mel scale and f is the input frequency.

An example melspectrogram is shown in 2.2. In this example the loudness is visualized by the lightness of the color where darker colors represent low loudness and lighter colors represent high loudness.

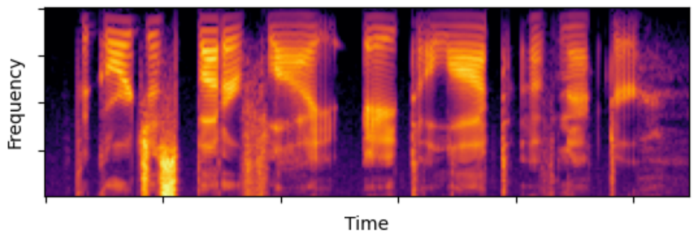


Figure 2.2: Spectrogram of the sentence "The gorgeous butterfly ate a lot of nectar."

2.2 What are Artificial Neural Networks (ANNs)?

Artificial Neural Networks are "computing systems inspired by biological neural networks" [48]. They consist of units called neurons, which are connected by edges.
A neuron receives signals from incoming edges, which are weighted by multiplying each signal with the weight value of each respective edge. The weighted signals are then summed up, with an additional bias term added and passed to an activation function. The activation function finally determines the output a neuron, which is propagated along all its outgoing edges.
In a more formal way, the output of a single neuron can be written as [30]:

$$\sigma(\boldsymbol{w} \cdot x + b)$$

where:
$\boldsymbol{w}$ denotes the weight matrix containing the weights of all incoming edges
x is a vector containing the incoming signal values
b is the bias vector
σ the activation function.

2.3 Architectures of ANNs

2.3.1 Feedforward neural network (FFNN)

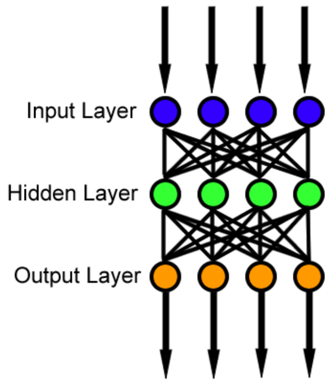


Figure 2.3: Typical architecture of a FFNN, Image by Paskari CC BY-SA 3.0 [31]

The general architecture of a feedforward neural network is depicted in Figure 2.3.
It consists of an input layer, one or several hidden layers and an output layer. Each layer is fully connected to its adjacent layers. Signals start propagating from the neurons in the input layer, passing through the hidden layers to the the output layer.

2.3.2 Time delayed neural network (TDNN)

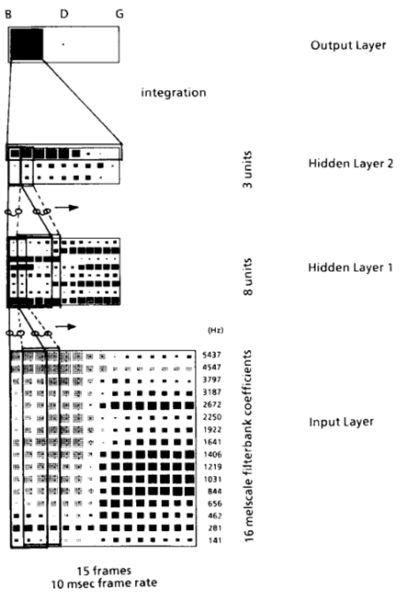


Figure 2.4: Original TDNN architecture, Image by [45] used with permission of the authors

Feature extraction in neural networks is often performed using Time delayed neural networks. They were originally introduced by Waibel et al. in [45] and operate by sliding filter matrices across the time axis of an input feature matrix. Inputs covered by a window are then fed into a TDNN unit, which calculates the output for each timestep by multiplying all values with the the corresponding weight value of the filter matrix and then applying an activation function on their weighted sum, as visualized in 2.5. The weight entries of the filter matrix are the same for each timestep, that is processed by the sliding window and are later updated using gradient descent.

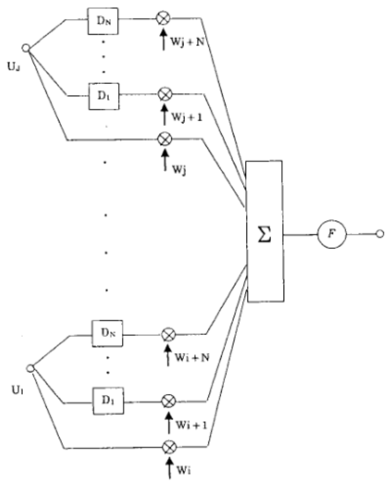


Figure 2.5: A TDNN unit, Image by [45] used with permission of the authors

By using the output matrix as input feature for the next layer, a hierarchical feature extraction can be achieved as shown in 2.4. An important advantage of TDNNs is that they are able to extract features, while being insensitive to shifts in time [45].

2.3.2.1 Convolutional neural network (CNN)

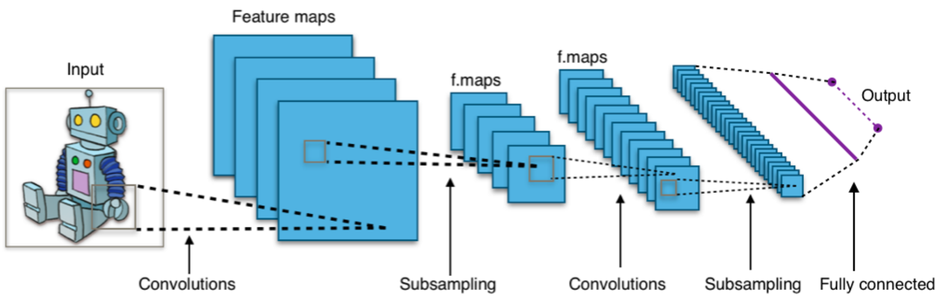


Figure 2.6: Typical architecture of a CNN, Image by Aphex34 CC BY-SA 4.0 [3]

Convolutional neural networks are a special type of TDNN where the delay is applied on the space instead of the time dimension. Their network architecture is inspired by biological processes, namely the the organization of the animal visual cortex.[49] The main component of CNNs are so-called convolutional layers, which produce their output based on a sliding window calculation. At each window location the resulting value is obtained by performing a component-wise multiplication of a filter tensor and the area covered by the window and then summing up the values and applying an activation function. This operation is called convolution. Every filter used, produces a new tensor called a feature map, as depicted in 2.7

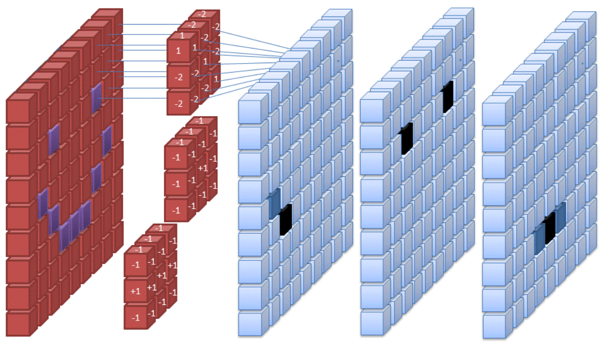


Figure 2.7: CNN filters and feature maps, Image by Cecbur CC BY-SA 4.0 [8]

In this example, the features maps (blue matrices) are being extracted from the input matrix (large red matrix) using three filters (small red matrices).
By combining convolutional layers with other types of layers such as pooling and fully connected layers, a bottom-up hierarchical feature extraction can be achieved, as shown in 2.6

2.3.3 Recurrent neural network (RNN)

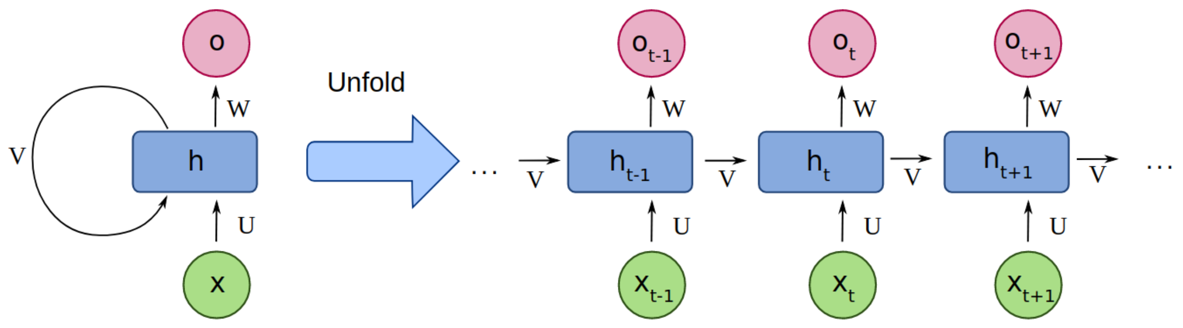


Figure 2.8: Basic RNN architecture, Image by Ixnay CC BY-SA 4.0 [23]

Recurrent neural networks are "a class of neural networks, that allow previous outputs to be used as inputs, while having hidden states"[2] An input sequence is processed in recurring steps, consuming one input at a time. In 2.8 we see the input x_{t-1} (green) being passed to hidden layers h_{t-1} (blue) whose result gives the output o_{t-1} (red). Additionally, a hidden state v is is fed back to the next step The same process is repeated for the rest of the input sequence. Several ways exist to calculate the next output o_t and the next hidden state h_t at each step. The standard recurrent cell calculates the next hidden state h_t and output y_t using the following formulas [51]:

$$h_t = \sigma(W_h h_{t-1} + W_x x_t + b)$$
$$y_t = h_t$$

where σ denotes the activation function. W_h and W_x are the weights and b is the bias.

2.3.3.1 Bidirectional RNN (BiRNN)

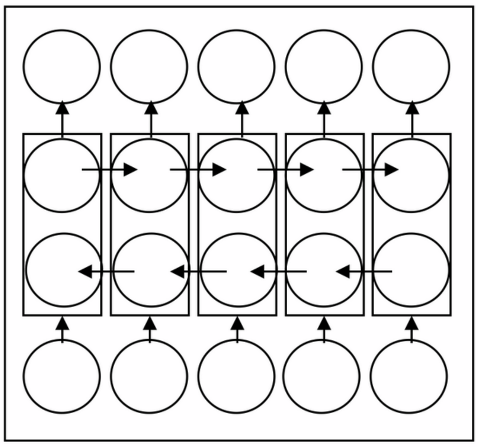


Figure 2.9: Architecture of bidirectional RNNs, Image by Incfk8 CC BY-SA 4.0 [22]

Unidirectional RNNs are able to integrate information of the input sequence up to the processed element. To be able to also integrate information of future elements of the sequence, bidirectional RNNs (BIRNN) were introduced in [35]. The architecture of a BiRNN is visualized in 2.9. As we can see, the input (bottom) is processed in order and reverse order (middle) and then both outputs for each step are passed to the output layer (top)

2.3.3.2 Multilayer RNN

By passing the output sequence of an RNN into another, the depth of a RNN can be increased. Multilayer RNNs work better on some tasks than shallower ones, even though the reason for this is not theoretically clear.[16]

2.3.3.3 Long short-term memory (LSTM)

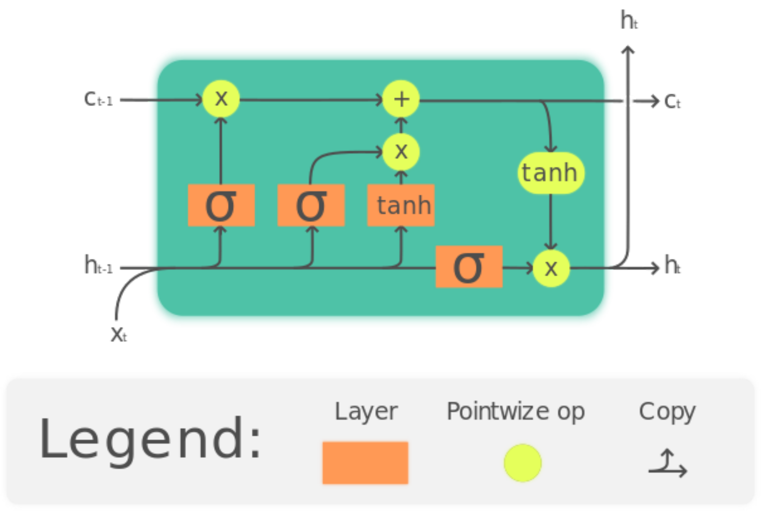


Figure 2.10: Architecture of a LSTM cell, Image by Guillaume Chevalier CC BY-SA 4.0 [9]

While RNNs are able to deal with varying sequence length, they are prone to the so-called vanishing gradient problem, which happens during backpropagation (see 2.4.2.2) This happens due to signals tending to vanish during gradient calculation [17], which effectively limits RNNs to shorter sequences. In order to overcome this limitation, a mechanism to store long-term information was introduced in [18]: The long short-term memory cell, which is capable of storing long-term information in its cell state.
Several variations of LSTM cells exist, such as LSTM cells with forget gate and peephole connection, which where proposed in [15]. The forget gate allows the cell to remove information from the cell state, while the peephole allows the cell to overcome a lack of information, caused by not having direct connections between the gates and cell state [51]. In figure 2.10 we can see a LSTM cell with forget gate and without peephole connection. This type of LSTM cell can

be described using the following formulas [51]:

$$\begin{aligned}
f_t &= \sigma\left(W_{fh}h_{t-1} + W_{fx}x_t + b_f\right)\\
i_t &= \sigma\left(W_{ih}h_{t-1} + W_{ix}x_t + b_i\right)\\
\tilde{c}_t &= \tanh\left(W_{\tilde{c}h}h_{t-1} + W_{\tilde{c}x}x_t + b_{\tilde{c}}\right),\\
c_t &= f_t \cdot c_{t-1} + i_t \cdot \tilde{c}_t\\
o_t &= \sigma\left(W_{oh}h_{t-1} + W_{ox}x_t + b_o\right)\\
h_t &= o_t \cdot \tanh\left(c_t\right)
\end{aligned}$$

f_t is the output of the layer of the forget gate (leftmost orange layer in figure 2.10). This output determines how much information of the cell state c_{t-1} will be preserved. For instance if $f_t = 0$ all information will be thrown away.

i_t and c_t are the outputs of the layers of the input gate (second and third orange layer in figure 2.10), which are being used for adding information to the cell state.
Finally the the result of of the layer of the output gate (rightmost orange layer in figure 2.10 is being used for deciding which information will be output [51].

2.3.3.4 Gated Recurrent Unit (GRU)

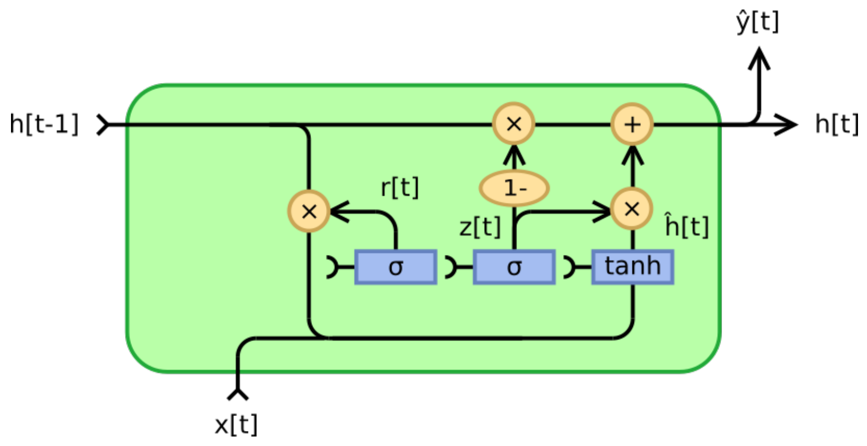


Figure 2.11: Architecture of a GRU unit, Image by Jeblad CC BY-SA 4.0 [24]

Since LSTMs are computationally rather complex, a simpler architecture was proposed in [10]: the Gated Recurrent Unit. GRUs combine the forget and input gates into a single update gate [51]. A GRU cell can be described using the following formulas [51] and is shown in figure 2.11.

$$\begin{aligned}
r_t &= \sigma\left(W_{rh}h_{t-1} + W_{rx}x_t + b_r\right),\\
z_t &= \sigma\left(W_{zh}h_{t-1} + W_{zx}x_t + b_z\right)\\
\hat{h}_t &= \tanh\left(W_{\hat{h}h}\left(r_t \cdot h_{t-1}\right) + W_{\hat{h}x}x_t + b_z\right)\\
h_t &= \left(1 - z_t\right) \cdot h_{t-1} + z_t \cdot \hat{h}_t.
\end{aligned}$$

2.3.4 Transformer

The transformer uses an encoder-decoder structure where the encoder maps an input sequence $(x_1, \ldots, x_n)$ to a sequence of continuous representations $z = (z_1, \ldots, z_n)$, which are then autogressively decoded to an output sequence $(y_1, \ldots y_m)$ [44].

2.3.4.1 Attention

Attention is a method that allows neural networks to focus more on certain parts of an input. It maps a query and a set of key-value pairs to an output, by summing up the values that are scaled by weights. The weights are calculated using a compatibility function, taking the query and key as input [44].

2.3.4.2 Multi-Head Attention

The transformer uses an attention mechanism, allowing it to focus on different representation subspaces and positions. Different representations of queries, keys and values are calculated using linear projections that are learned. The attention is then calculated in parallel on all projections, which are then concatenated and projected again [44]. Mathematically, this can be described as:

$$\begin{aligned} \text{MultiHead }(Q,K,V) &= \text{ Concat } (\text{ head }_1, \ldots, \text{ head }_\text{h}) W^O \\ \text{where head }_\text{i} &= \text{ Attention } \left(QW_i^Q, KW_i^K, VW_i^V\right) \\ \text{and Attention }(Q,K,V) &= \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \end{aligned}$$

$W_i^Q \in \mathbb{R}^{d_\text{modd} \times d_k}$, $W_i^K \in \mathbb{R}^{d_\text{model} \times d_k}$, $W_i^V \in \mathbb{R}^{d_\text{model} \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_\text{model}}$

2.3.4.3 Positional encoding

Since information about the order of a sequence would be otherwise be lost, it is necessary to add a positional encoding to the input and output embeddings, before passing them into the the input or output stack. For the transformer model, positional encoding is calculated as:

$$\begin{aligned} PE_{(pos,2i)} &= \sin\left(pos/10000^{2i/d_\text{model}}\right) \\ PE_{(pos,2i+1)} &= \cos\left(pos/10000^{2i/d_\text{model}}\right) \end{aligned}$$

where each position is mapped to an individual sinus function [44].

2.3.4.4 Encoder

The encoder is composed of $N > 1$ layers. Each layer has two sublayers. The first sublayer contains a Multi-Head-Attention mechanism and the second sublayer contains a fully-connected FFNN. The output of the Multi-Head-Attention or the output of the FFNN together with the original input to the sublayer (residual connection) produces the output of the sublayer, after applying a layer norm [44]. This can be described by the following formula:

$$\text{output} = \text{LayerNorm}(x + \text{Sublayer}(x))$$

2.3.4.5 Decoder

The decoder is composed of $M \geq 1$ layers. Its layer architecture is almost identical to an encoder layer, except that the Multi-Head Attention in the first sublayer is masked and there is another sublayer between the first and second sublayer. The third sublayer contains one more Multi-Head Attention, which uses the output of the encoder stack. Like in every other sublayer, its output is produced after applying the layer norm, as described at 2.3.4.4. Furthermore, masking in the first layers Multi-Head Attention mechanism is applied, to prevent that predictions at a certain position i to use outputs at positions after i [44].

2.4 Training ANNs

2.4.1 Loss function

In order to measure the error of a neural network and be able to reduce it, a so-called loss function is useful. The loss function should be differentiable, since the algorithm for reducing the error described at 2.4.2.2 relies on partial derivatives. Otherwise, it needs to be approximated by a differentiable function.

2.4.1.1 Cross-entropy loss

For classification tasks, the Cross-entropy loss function is often applied, which is given by

$$l(x, y) = -\frac{1}{N} \sum_{n=1}^{N} \sum_{c=1}^{C} \log \left(\frac{\exp(x_{n,c})}{\sum_{i=1}^{C} \exp(x_{n,i})} \right) y_{n,c}$$

where x denotes the input tensor and y the target tensor, both of shape (N, C)
(N = sequence length, C = number of classes) containing the output and target scores of the neural network [11].

2.4.2 Gradient Descent

2.4.2.1 Gradient descent

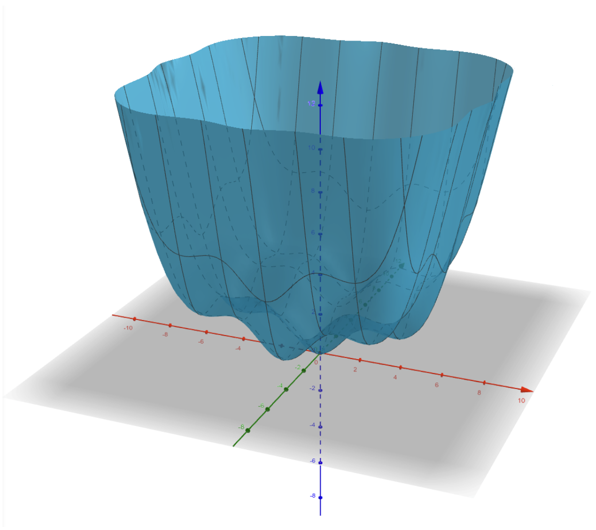


Figure 2.12: 3D plot of the function $C(x, y) = (0.4x + 0.5\sin x)^2 + (0.4y + 0.5\sin y)^2$. Created using GeoGebra [19]

In order to understand gradient descent, it is useful to imagine the loss function as a 2-dimensional surface, where the x and y coordinates are variables that can be changed. These variables are the weights and biases of the neural network. Learning using gradient descent means to tune these parameters to reduce the overall loss, which can be visually understood as a ball rolling downhill on a error surface [30] as visualized in 2.12. Since analytically finding the minimum of a function dependent on many parameters would be extremely difficult, it is more viable to gradually approach a minimum of the function. The gradient of the loss function is the direction of highest slope. Therefore, the negative of this gradient is direction of highest decrease. Consequently, adding a multiple of the negative of the gradient to the current parameter vector always reduces the loss, if it is not near a local minimum.
Near a local minimum, the updated loss may become larger again, if the learning rate is too high. This can be compared to a ball, that has a too high velocity, racing past the local minimum and uphill again. Additionally it is possible, that the loss converges to a suboptimal local minimum, when the learning rate is chosen too low, which is again very easy to imagine if compared to a ball. For reducing the impact of inappropriate learning rates, it is often useful to use adaptive learning rate scheduling, as described at chapter 2.4.3.

Mathematically, gradient descent can be described by iteratively applying an update rule, until a certain stop criteria is met, e.g. a minimum factor, by which the loss should decrease at every iteration. The update rule is given by [30]:

$$v \longrightarrow v' = v - \eta \nabla C.$$

where:
v is the vector of parameters of the neural network
η is the learning rate
∇C is the gradient of the loss function C with respect to v, given by:

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \ldots, \frac{\partial C}{\partial v_m}\right)^T .$$

The gradient of the loss function can be calculated using the backprogation algorithm.

2.4.2.2 Backpropagation

As the name implies, it works by propagating the gradient of each layer in the network backwards to the previous layer, in order to calculate the gradient in the previous layer. For a neural network, the backpropagation algorithm can be described by following steps [30]:

1. **Input:** Set the corresponding activation a^1 for the input layer
2. **Feedforward:** For each $l = 2, 3, \ldots, L$ compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$
3. **Output error δ^L:** Compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$.
4. **Backpropagate the error:** For each $l = L-1, L-2, \ldots, 2$
   compute $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$
5. **Output:** The gradient of the cost function is given by $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$ and $\frac{\partial C}{\partial b^l_j} = \delta^l_j$

where:
a^l is the activation output of layer $l-1$
w^l is a matrix containing the weights of the connections between layer $l-1$ and layer l
b^l is a vector containing the bias of the neurons in layer l
σ denotes the activation function
L is the total number of layers in the neural network

2.4.2.3 Stochastic Gradient Descent

To reduce the variance of the change in gradient at each iteration of gradient descent, the gradient is often accumulated. A so-called minibatch is processed by the neural network and then the parameters are updated by averaging the gradients. While traditionally large batch sizes were aimed for in order to reduce computation, time recent work has shown that small batch sizes often lead to a better generalization performance [29].

2.4.3 Learning rate schedulers

2.4.3.1 Cyclic learning rates

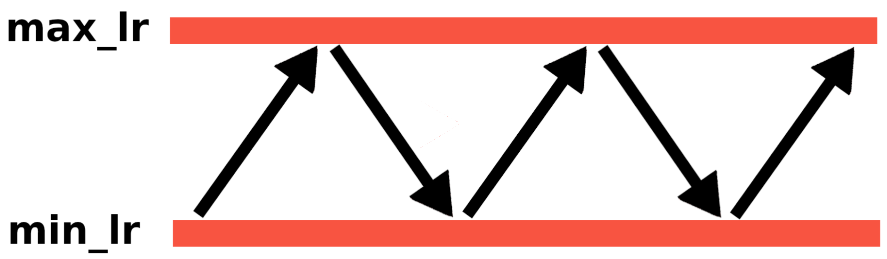


Figure 2.13: Illustration of a triangular cyclic learning rate

In 2017 Smith [38] proposed and showed the efficiency of cyclic learning rates for training neural networks, leading not only to better classification accuracy, but also often faster convergence, than without cyclic learning rates. Instead of only reducing the learning rate over time, it is cycled between a lower and an upper bound, as visualized in figure 2.18. According to Smith, the learning rate is the most important hyper-parameter to tune for training neural networks.

2.4.3.2 ReduceLROnPlateau

The idea of ReduceLROnPlateau is to reduce the learning rate, once a certain metric like validation loss has stopped improving [33]. For instance, if the validation loss at the end of the epoch is still larger than $best * (1 + threshold)$, the learning rate is reduced by a certain factor. Additionally, a patience of n epochs can be set, which only lowers the learning rate, if there was no improvement for n epochs and several other options such as cooldown are possible [33].

2.4.4 Dropout

Overfitting in a neural network means, that it learns the labels for the data samples by rote, instead of gaining the ability to generalize for other unseen samples. Dropout is an efficient and highly adopted method, that allows dealing with this, by randomly excluding a percentage of the units and their connections during training. This forces the network to make decisions for each single sample, based on different subsets of the weights and consequently makes it less likely that it memorizes whole samples and therefore reduces overfitting. During evaluation, the predictions of all these thinned networks are then averaged by using the whole network with smaller weights [40]. The dropout rate can range from 0 to 1, which corresponds to the percentage of random units to be dropped during training.

2.4.5 Regularization techniques

If a neural network has a lot of parameters available, it may use all of them, although it may be able to make the same predictions with much less parameters. This is because the amount of parameters is not accounted for.

2.4.5.1 L2 regularization

A simple method to optimize the network to use less parameters is L2 regularization, which adds the squared magnitude of the weights to the loss function, thereby making the neural network aim for using smaller and less weights. The loss function with l2 regularization can be expressed as [43]:

$$L_\lambda(\mathbf{w}) = L(\mathbf{w}) + \lambda\|\mathbf{w}\|_2^2$$

where:
$L(w)$ is the unregularized loss function
λ is the regularization factor

2.4.5.2 Weight decay

Another method to reduce the complexity of a neural network is weight decay, which exponentially decays the weights of a neural network at each step. It can be described by the formula [28]:

$$\boldsymbol{\theta}_{t+1} = (1-\lambda)\boldsymbol{\theta}_t - \alpha\nabla f_t\left(\boldsymbol{\theta}_t\right)$$

where:
λ is the rate of weight decay per step
$\nabla f_t\left(\boldsymbol{\theta}_t\right)$ is the t-th batch gradient
α is the learning rate

For the Stochastic Gradient Descent Optimizer, weight decay is equivalent to L2 regularization, but not for other optimizers such as Adam [28]. This often leads to confusion and the belief that these techniques are the same.

2.4.6 Checkpoints

During training, the network may perform better and then worse between subsequent epochs in terms of a metric like validation loss, which can be for instance due to the gradient bouncing around a local minimal. When the model is then finally evaluated, it may perform worse than it would have with the weights of a previous epoch, where the metric was better. Therefore, it is useful to make a checkpoint every time when the metric improved after an epoch.

2.4.7 Early Stopping

At some point, the the gradient is so close to a local minima, that the change after a gradient update becomes almost insignificant, with almost no effect on most metrics. In such cases, the training can be stopped early, to avoid wasting further computantial resources, when the network does no longer improve.

2.4.8 Data augmentation techniques

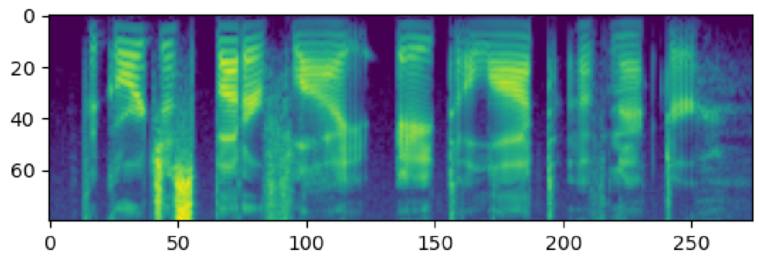


Figure 2.14: MelSpectrogram of the sentence "The gorgeous butterfly ate a lot of nectar."

Data augmentation can reduce overfitting and improve the accuracy of a neural network, by artifically creating a greater variety of data based on the existing samples.

2.4.8.1 Pitch Shift

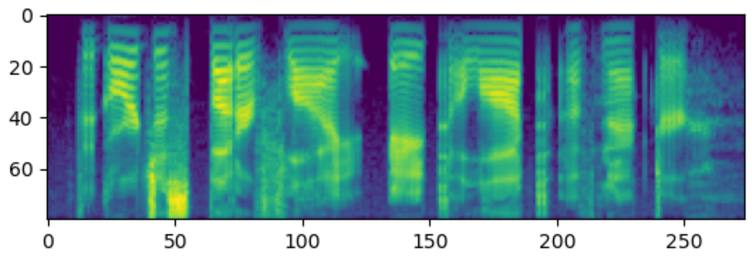


Figure 2.15: MelSpectrogram with Pitch Shift applied

By shifting the pitch of an audiofile by a random number of semitones, a larger amount of voice levels can be created.

2.4.8.2 Time Stretch

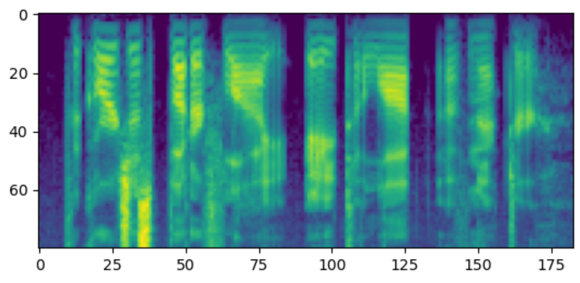


Figure 2.16: MelSpectrogram with Time Stretch applied

Stretching or shrinking the duration of an audiofile randomly can generate more diversity of faster and slower speakers.

2.4.8.3 Frequency Mask

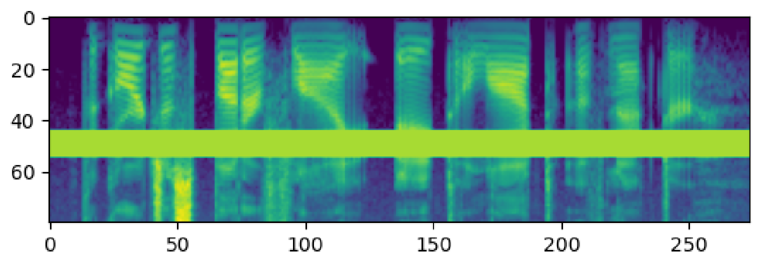


Figure 2.17: MelSpectrogram with Frequency Mask applied

Setting random frequencies in an audiofile spectrogram to the mean value of the spectrogram can make the model less dependent on single frequencies and more robust to noise.

2.4.8.4 Time mask

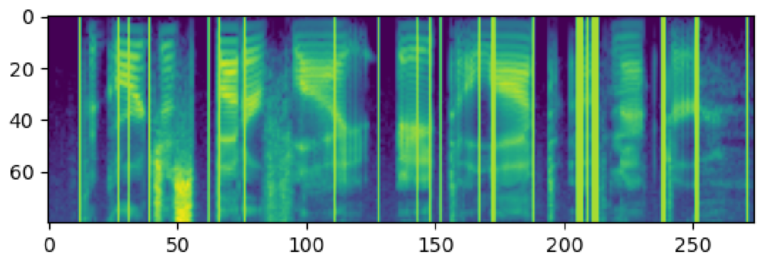


Figure 2.18: MelSpectrogram with Time Mask applied

Setting random time steps in a spectrogram to the mean value of the spectrogram can make the model less dependent on single points in time.

2.5 Dynamic Time Warping (DTW)

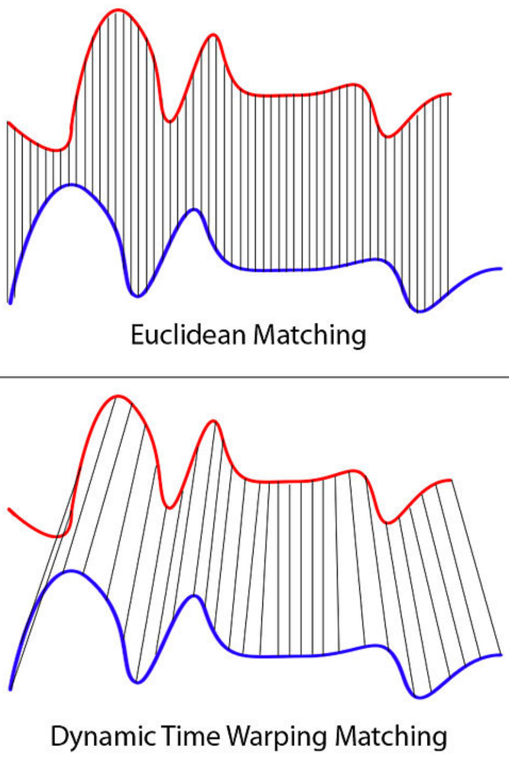


Figure 2.19: Visualization of DTW and Euclidian Matching, Image by XantaCross CC BY-SA 3.0 [50]

Dynamic Time Warping is an algorithm to find the optimal mapping between two similiar sequences X and Y of possibly different lengths. Compared to simpler methods like euclidian matching, which simply aligns the elements of both sequences based on their index, it can map several identical elements of one sequence to a single element of the other sequence. This is illustrated figure 2.19.
The algorithm works by computing the minimal cost for each possible alignment using dynamic programming and then backtracking along the resulting accumulated cost matrix, to obtain the best path to map the first sequence to the second.
The accumulated cost matrix can be calculated by the following strategy [36]:

1. First row: $D(1, j) = \sum_{k=1}^{j} c\left(x_1, y_k\right), j \in [1, M]$.
2. First column: $D(i, 1) = \sum_{k=1}^{i} c\left(x_k, y_1\right), i \in [1, N]$.
3. All other elements: $D(i, j) = \min\{D(i-1, j-1), D(i-1, j), D(i, j-1)\}+$ $c\left(x_i, yj\right), i \in [1, N], j \in [1, M])$

In pseudocode this can be expressed as [36]:

Algorithm 1: *AccumulatedCostMatrix*(X, Y, c)

```
$n \leftarrow |X|$;
$m \leftarrow |Y|$;
$dtw[] \leftarrow new[x \times m]$;
$dtw(0,0) \leftarrow 0$;
for $j = 1; j \leq m; j{+}{+}$ do                                  // first row
    $dtw(1, j) \leftarrow dtw(1, j-1) + c(1, j)$;
end
for $i = 1; i \leq n; i{+}{+}$ do                                  // first column
    $dtw(i, 1) \leftarrow dtw(i-1, 1) + c(i, 1)$;
end
for $i = 1; i \leq n; i{+}{+}$ do                                  // all other elements
    for $j = 1; j \leq m; j{+}{+}$ do
        $dtw(i, j) \leftarrow \min\{dtw(i-1, j); dtw(i, j-1); dtw(i-1, j-1)\} + c(i, j)$;
    end
end
return $dtw$;
```

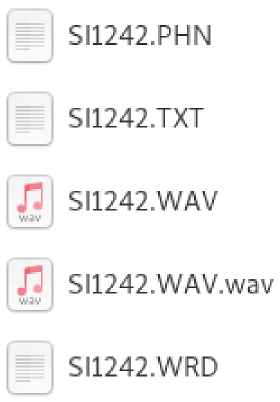
The alignment path is then obtained by starting at the right bottom of the matrix and iteratively choosing the immediate cell to the left and top, which has the smallest cost.
In pseudocode this can be expressed as [36]:

Algorithm 2: *OptimalWarpingPath*(dtw)

```
$path[] \leftarrow new\ array$;
$i \leftarrow num\_rows(dtw)$;
$j \leftarrow num\_columns(dtw)$;
while $(i > 1)$ & $(j > 1)$ do
    if $i == 1$ then
        $j \leftarrow j - 1$;
    else if $j == 1$ then
        $i \leftarrow i - 1$;
    else
        if $dtw(i-1, j) == \min\{dtw(i-1, j); dtw(i, j-1); dtw(i-1, j-1)\}$ then
            $i \leftarrow i - 1$;
        else if $dtw(i, j-1) == \min\{dtw(i-1, j); dtw(i, j-1); dtw(i-1, j-1)\}$ then
            $j \leftarrow j - 1$;
        else
            $i \leftarrow i - 1$;    $j \leftarrow j - 1$;
        end
    end
    path.add($(i, j)$);
end
return $path$;
```

3 The TIMIT dataset

The TIMIT corpus is a collection of 6300 audio recordings of 10 sentences by 630 speakers of eight major dialects of American English.
All recordings in total amount to about 5 hours of speech.
It was designed by a joint effort of the Massachusetts Institute of Technology (MIT), SRI International (SRI) and Texas Instruments, Inc. (TI).[14]
Each sentence has sentence-, word-, and phoneme-level labels containing the text and precise timings. The respective file types for a single utterance (here SI1242) are shown in 3.1



SI1242.PHN
SI1242.TXT
SI1242.WAV
SI1242.WAV.wav
SI1242.WRD

Figure 3.1: Different file types for a single utterance

The dataset is already divided into a training set containing 3696 sentences and a test set of 1344 sentences. Additionally a smaller core test set can extracted, which is a subset of the test sentences including 192 utterances.

4 Definitions

In order to avoid confusionsm as the terms "Phoneme classification" and "Phoneme recognition" are often used interchangeably in other works, we define them for our work in the following way:

4.1 Phoneme classification

Given:
Audiofile containing a single phoneme.
Goal:
Find the correct type of phoneme.
Quality measurements:

$$\text{accuracy} = \frac{\text{correct predicted phonemes}}{\text{total phonemes}}$$

4.2 Phoneme recognition

Given:
Audiofile containing a whole sentence.
Goal:
Find the correct type of phoneme at every location in time.
Quality measurements:

$$\text{frame error rate (FER)} = 1 - \frac{\text{correct predicted frames}}{\text{total frames}}$$

where frames are created by splitting the audio file into multiple frames with the same duration.

$$\text{phoneme error rate (PER)} = \frac{\text{edits}}{\text{total phonemes}}$$

where the number of edits is calculated using the levenshtein distance between the predicted sequence and the target sequence.

$$\text{F1 score} = \frac{\text{tp}}{\text{tp} + \frac{1}{2}(\text{ fp} + \text{ fn})}$$

where tp denote true positives, fp denote false positives and fn denote false negatives.

4.3 Phoneme alignment

Given:
Audiofile containing a whole sentence and the sequence of phonemes. spoken
Goal:
Find the correct start and end times for all phonemes in the audiofile.
Quality measurements:

$$\text{accuracy} = \left(\frac{\text{\# correct predicted boundaries}}{\text{\# total boundaries}} \right) \cdot 100\%$$

where predicted phoneme boundaries are correct, if they are less than a certain tolerance in milliseconds away from correct phoneme boundaries.

5 Related work

5.1 Phoneme classification

We did not find any other work that classify phonemes without context.

5.2 Phoneme recognition

In order to use a phoneme recognition model for aligning phonemes, it needs to be highly precise in the time dimension. Therefore, a low frame error rate, which includes time information is more desirable, than a low phoneme error rate in this context. For this reason, we only compare our work with other works, that include FER in their results. Models that are optimized for phoneme error rate achieve PER values as low as 8.3% [5].

Arel, Itamar, et al. (2011) [4] created a system using a semi-supervised Hierarchical Deep Recurrent Network (HDRN) and HMM. Before actually training the HDRN model for phoneme classification, it is pretrained in an unsupervised manner, to learn the general structure of speech signals, by passing spectrograms to it. Finally a HMM-based decoding is being used, in order to obtain the PER value. Similiar to a CNN, this approach achieves a hierarchical feature extraction. Mel Frequency Cepstral Coefficients (MFCCs) are being used as input features and the 61 phonemes are folded into 39, as proposed in [26]. Frames containing the silence phoneme were removed only from the beginning and end of the evaluated sequence. Their approach resulted in 24.16% FER and 23.60% PER.

Song, William and Cai (2015) [39] developed a system using a CNN combination with an RNN and Connectionist Temporal Classification (CTC) loss. The CNN network predicts phonemes framewise and its output probabilities are then used together with the CTC loss by a RNN for decoding the phoneme sequence. Logmel spectrograms are used as input features and the 61 phonemes are merged into 39, as proposed in [26]. Frames containing the silence phoneme were not removed from evaluation. They achieve 22.1% FER and 29.4% PER.

Shulby, Christopher Dane, et al. (2019) [37] created a system based on a HMM, CNN and Hierarchical Tree Support Vector Machine (HTSVM). The HMM labeler may serve for phoneme boundary detection, although its use is not made clear. The output of the HMM labeler is then used together with a spectrogram of the audiofile in a CNN feature extractor, which then outputs to the HTSVM. Finally the output of the HTSVM is used together with the output of the HMM labeler for PER smoothing, which then outputs phoneme labels. MFCCs are used as input features and the phonemes are folded, as proposed in [26].

Frames containing the silence phoneme were all removed from evaluation. Their approach results in 28% FER and 32% PER.

5.3 Phoneme alignment

Keshet et al. (2005) [25] developed a system based on discriminative learning. For input features they use MFCC+$\Delta + \Delta\Delta$, which includes the first and second derivatives of the cepstral coefficients. Based on kernel machines and large margin classifiers, their model learns an alignment function, mapping speech signal, phoneme representation and target alignment into an abstract vector space. Separating correct alignments from incorrect ones is the goal of the alignment function. Their approach resulted in a 92.1% agreement with a 20ms tolerance.

Hosom (2009) [20] created a system based on a hybrid between ANN and HMM. They used the Bark frequency scale, which is similiar to the mel-scale for their spectrograms and performed a custom phoneme folding: first mapping to 54 phonemes and then splitting some phonemes which results in a set of 61 phonemes. The ANN receives a context window of 5 frames to make a probability estimation for the phoneme at this location, which is then used by the HMM. Additionally they incorporated several other features, such as intensity discrimination and burst detection. They achieve a 93.36% agreement with a 20ms tolerance.

Stolcke et al. (2014) [41] developed a system combining ANN and HMM. They used several different types of input features such as MFCC and Perceptual Linear Prediction (PLP) and the phoneme folding proposed by [20]. Using HMMs to perform a forced alignment, they further improved the alignments, by adjusting them with ANN-based boundary-correction models. By receiving phonetic context and duration features as input, these boundary correction models predict a better location of the boundary. Their approach resulted in a 96.8% agreement with a 20ms tolerance, which is the best result we have found.

6 Methods

6.1 Technical setup

All program code of our experiments is written in the programming language Python and can be found on GitHub. The CNN model for phoneme classification is hosted at:
`https://github.com/sugeedarou/BA_ForcedAlignment_CNN`
The Transformer and RNN models can be found at:
`https://github.com/sugeedarou/BA_ForcedAlignment`
For phoneme classification, we use the TensorFlow machine learning library by Google. Phoneme recognition and alignment is performed using the PyTorch machine learning library by Facebook. We switched to PyTorch after our initial experiments, since we found it much easier to use.

6.2 Data preprocessing

6.2.1 Training, validation and test dataset

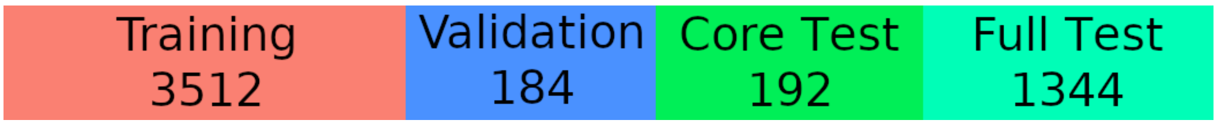


Figure 6.1: Division into training, validation and test sets

We use the given training and test split, as provided by the dataset creators (see figure 3) and test on the core subset for phoneme recognition and on the full test set for alignment as in related work. Additionally, we remove the dialect sentences (SA sentences), since they exist in both the training and test dataset, which is recommended by the authors of TIMIT [13]. Furthermore, 5% of the training set is being used for validation purposes.
The resulting division and respective utterance count for each of the training, validation and test sets is shown in 6.1
For our phoneme recognition and alignment models we use the whole sentence, whereas the phoneme classification model only uses the waveforms of phonemes, extracted from the utterances.

6.2.2 Conversion of audio waveforms to spectrograms

We use a frame length of 25ms and a stride of 10ms for the FFT window.

6.2.3 Phoneme folding

In the TIMIT dataset, 61 different phonemes are distinguished. While this is certainly useful for phonetic research, it is almost impossible to tell apart some phonemes.
For instance voiceless closures (e.g. of p, t, k, q) sound almost the same as the silence at the beginning and end of a recording.
Therefore, we use a two-step mapping of these similiar phonemes to a single phoneme, as proposed by [26].
First the following, phonemes are merged into a single one, as shown in table 6.1:

Phonemes	Folded phoneme
’ux’, ’uw’	’uw’
’axr’, ’er’	’er’
’ax-h’, ’ax’	’ax’
’em’, ’m’	’m’
’nx’, ’n’	’n’
’eng’, ’ng’	’ng’
’hv’, ’hh’	’hh’
’pcl’, ’tcl’, ’kcl’, ’qcl’	’cl’
’bcl’, ’dcl’, ’gcl’	’vcl’
’h#’, ’#h’, ’pau’	’sil’

Table 6.1: Phoneme folding step 1

Additionally, all glottal stops (symbol ’q’) are being removed. This leaves 48 phonemes, which are being used to train a phoneme recognition model.
When the phoneme recognition model outputs phonemes, they are further merged, as shown in table 6.2:

Phonemes	Folded phoneme
’cl’, ’vcl’, ’epi’, ’sil’	’sil’
’el’, ’l’	’l’
’en’, ’n’	’n’
’sh’, ’zh’	’zh’
’ao’, ’aa’	’aa’
’ih’, ’ix’	’ix’
’ah’, ’ax’	’ax’

Table 6.2: Phoneme folding step 2

Finally we have 39 phonemes, that are being used as a result of classification and recognition and also for alignment. In order to be consistent with the best reported result for FER by [39], we decided not to remove silence frames from evaluation. The resulting distribution of all phonemes in the training dataset without dialect sentences is shown in figure 6.2.3.

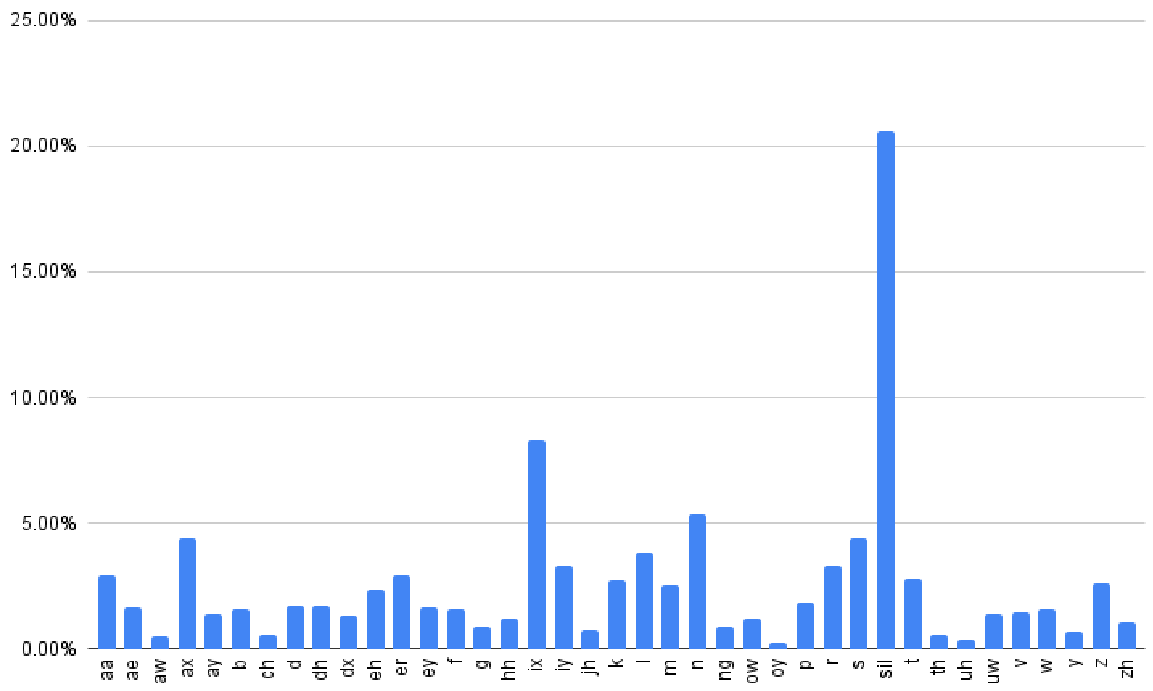


Figure 6.2: Distribution of phonemes in the folded training dataset of TIMIT

6.3 Models

6.3.1 CNN for phoneme classification

We train the CNN model InceptionV3[42] to classify the 48 phonemes and later fold them into 39 as described at 6.2. Even though we use transfer learning, the fine-tuning flag in TensorFlow is set, which unfreezes the top layers of the pretrained model and therefore allows it to use more trainable parameters. First we train the network based on logmel spectrograms of phonemes, extracted from the sentences. After that we train on decibel-scale spectrograms.

Training configuration:

- **batch size:** 32
- **optimizer:** Adam
- **dropout:** 0.5 (which achieves the strongest regularization effect [6])
- **regularizer:** L2 (regularization_factor = 10^{-4})

After training for 30 epochs, we load the model from the checkpoint which had the lowest validation loss and evaluate it on the core test set.

6.3.2 Models for phoneme recognition and alignment

In the same way we do for phoneme classification (see 6.3.1), all phoneme recognition models are trained on 48 phonemes, which are then folded into 39 phonemes for evaluation.
We calculate the FER, F1 score and alignment accuracy of each batch and then average the results of all batches.

6.3.2.1 Autoregressive Transformer

Initially we experimented using autoregressive transformers which was not successful. All attempts either resulted in the model assigning the same phoneme to all frames or similiar confusion matrices as in figure 7.1.2. For this reason, we then tried out an encoder-only model, as described next.

6.3.2.2 Encoder-only Transformer

Our second transformer model consists of 8 stacked transformer encoders, with 8 heads and a feed-forward layer as decoder. We also limited the transformer context to 128 frames, which corresponds to $128 * 10ms = 1.28s$, since this reduces its memory consumption and therefore allows using more encoder layers and heads. Furthmermore we use a feed-forward-layer before the transformer encoder layers to transform the 80 mel values of each frame into 256, which is the input size for the encoder. Additionally the filterbank values are divided by 4 with 2 added, which normalizes the values into the same range as character embeddings would be.

Training configuration:

- **batch size:** 16
- **optimizer:** AdamW (weight decay coefficient: 10^{-2})
- **lr scheduler:** CyclicPlateauScheduler (initial_lr=$1 \cdot 10^{-4}$, min_lr=10^{-10}, lr_patience=0, min_improvement_factor=0.95, lr_reduce_factor=0.5, lr_reduce_metric='val_loss')
- **positional dropout:** 0.1
- **transformer dropout:** 0.1
- **dropout:** 0.5
- **early stopping:** patience=10, metric='val_loss'

We chose a lower starting learning rate than for the BiRNN model and a rather strict $min\_improvement\_factor$ of 0.95, since the transformer model returned a NaN (not a number) loss, when the learning rate was too high.

6.3.2.3 BIRNNs

Several different types of RNNs were trained: BiRNN, BiLSTM and BiGRU.

Training configuration:

- **batch size:** 16
- **optimizer:** AdamW (weight decay coefficient: 10^{-2})
- **lr scheduler:** CyclicPlateauScheduler (initial_lr=$5 \cdot 10^{-4}$, min_lr=10^{-10}, lr_patience=0, min_improvement_factor=0.97, lr_reduce_factor=0.5, lr_reduce_metric='val_loss')
- **layer count:** 8
- **hidden states:** 512
- **layer dropout:** 0.2
- **dropout:** 0.5
- **early stopping:** patience=10, metric='val_loss'

It may be unclear, why we use an additional global learning rate scheduling, even though the optimizer Adam is adapting the learning rate for each parameter already. Loshchilov, Ilya, and Frank Hutter did analyse the Adam optimizer in [28] and came to the conclusion, that just being an adaptive gradient algorithm "does not rule out the possibility to substantially improve its performance by using a global learning rate multiplier".

6.4 The CyclicPlateau scheduler

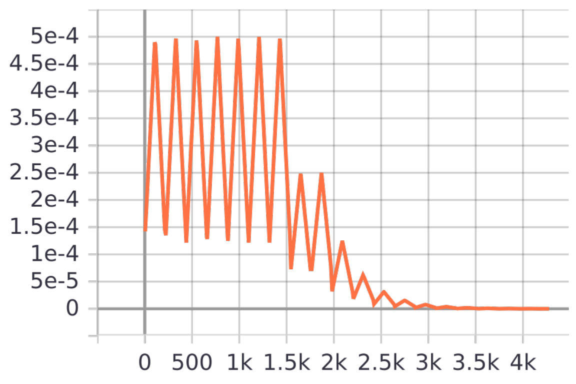


Figure 6.3: Example learning rate curve using the cyclic plateau scheduler and parameters of 6.3.2.3

Since default cyclic learning rate schedulers do not take into account if the validation loss is still improving, we decided to create a new learning rate scheduler.
The CyclicPlateauScheduler is a combination of an epoch-wise triangular cyclic learning rate with ReduceLROnPlateau in a single scheduler. Boundaries for the cycles are adjusted at the end of each epoch. The upper bound is the current learning rate and the lower bound is implicitly

defined as a fraction of the upper bound e.g. $\frac{1}{4} \cdot upper\_bound$. In pseudocode, the triangular policy for each step can be expressed as:

Algorithm 3: *training_step* $(step, steps\_per\_epoch, lr)$

```
$half\_steps \leftarrow steps\_per\_epoch \text{ 'div' } 2$;
$lower\_bound \leftarrow lower\_bound\_fraction \cdot lr$;
if $step < half\_steps$ then                      // interpolate upwards triangle line
    $c \leftarrow \frac{step}{half\_steps}$;
    $lr \leftarrow lr + c \cdot (lr - lower\_bound)$;
else                                              // interpolate downwards triangle line
    $c \leftarrow \frac{step - half\_steps}{half\_steps}$;
    $lr \leftarrow lr - c \cdot (lr - lower\_bound)$;
end
```

Boundary adjustment is done, based on the change of a certain validation metric e.g. *validation_loss* and given by:

Algorithm 4: *validation_epoch_end* $(metrics, lr\_reduce\_metric, best\_lr\_metric\_val, min\_improve\_factor, reduce\_metric\_too\_high\_count, lr\_patience, lr, min\_lr)$

```
$reduce\_metric\_val \leftarrow metrics[lr\_reduce\_metric]$;
if $reduce\_metric\_val > best\_lr\_metric\_val \cdot min\_improve\_factor$ then
    $reduce\_metric\_too\_high\_count \leftarrow reduce\_metric\_too\_high\_count + 1$;
end
if $reduce\_metric\_val < best\_lr\_metric\_val$ then          // metric improved
    $best\_lr\_metric\_val \leftarrow reduce\_metric\_val$;
end
if $reduce\_metric\_too\_high\_count > lr\_patience$ then      // not enough improvement
    $lr \leftarrow \max(lr \cdot lr\_reduce\_factor,\ min\_lr)$;
    $reduce\_metric\_too\_high\_count \leftarrow 0$;
end
```

The goal of this scheduler is to allow the model to slow down when approaching a good local minimum, while avoiding getting stuck in small valleys. The slowing down is achieved using the adjustment of learning rate boundaries and small valleys should be avoided due to the epoch-wise triangular learning rate, which encourages exploration.

6.5 Phoneme alignment through recognition and probability-based DTW

We take the sequence of output probabilities of a phoneme recognition model together with the target phoneme sequence for DTW input. Then we perform a standard DTW using a custom cost function, which takes a probability vector and a target phoneme index as input.

6.5.1 Cost functions

Let x be a vector of the probability sequence and y an element of the target phoneme sequence. Then the probability p of phoneme with index y, $(0 \leq y \leq 38)$ is given by $p = x[y]$. Consequently, cost function based on the complementary probability of the target phoneme index can be defined as:

$$cost_{probability}(x, y) = 1 - p$$

A better cost function can be defined, by assigning low probabilities an exponentially higher cost than high probabilities. In [7], who also apply probability-based DTW to a different problem, we found a cost function defined as:

$$cost_{exponential}(x, y) = \mathrm{e}^{-p}$$

Apart from that, a series of cost functions can be defined as:

$$cost_{root\_k}(x, y) = 1 - p^{1/k}, \ k \in \mathbb{N}$$

For $k = 1$ $cost_{root\_k}$ is identical to $cost_{probability}$.

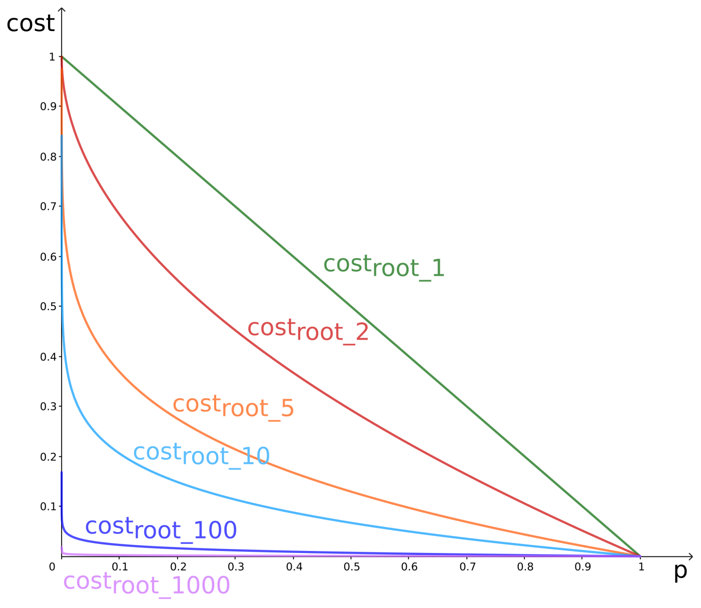


Figure 6.4: Plot of $cost_{root\_k}$ for different k

As we can see in figure 6.4, $cost_{root\_k}$ maps high probabilities to a similiar cost, while increasing the relative cost beween different low probabilities as k increases.

The +1 term of $cost_{probability}$ and $cost_{root\_k}$ can be omitted, since DTW is shift invariant, but we chose to include it as it makes semantically far more sense, since the cost error should be zero, if the probability for a certain phoneme is 100%.

6.5.2 Weighted cross-entropy loss

To optimize the phoneme recognition models for boundary detection, we use a weighted cross-entropy loss, which conditions the model to focus more on phoneme boundaries. If frame label at index $i - 1$ is different from frame label at index i, then we weight the loss around these indices for instance in the following way (50,100,100,50):

$$\begin{aligned} loss\_weights[i-2] &= 50 \\ loss\_weights[i-1] &= 100 \\ loss\_weights[i \quad\;] &= 100 \\ loss\_weights[i+1] &= 50 \end{aligned}$$

while all other weights are set to 1. Then we calculate the loss using the following formula as proposed by [27]:

$$l(x, y) = -\frac{1}{\sum_{k=1}^{N} w_k} \sum_{n=1}^{N} \sum_{c=1}^{C} w_n \log \left(\frac{\exp(x_{n,c})}{\sum_{i=1}^{C} \exp\left(x_{n,i}\right)} \right) y_{n,c}$$

As we can see, the weighted cross-entropy loss is very similiar to the default cross-entropy loss defined at 2.4.1.1. The averaging on the sequence length n has been replaced with an averaging over all loss weights and the loss for every element in the sequence is multiplied by its respective loss weight.
For phoneme recognition, it is important to classify all frames correctly and not only near phoneme transitions. Therefore we disable loss weighting for FER measurements.

7 Evaluation

In this section we evaluate the performance of different model architectures. We made a wrong mathematical assumption, that has a slight impact on the measured values. Instead of calculating the mean of all accuracies for an individual sentence, we calculated the error over a whole batch of 16 sentences. Nevertheless, the relative error of different models is still representative. Correctly measured values are given in our final recognition and alignment results.

7.1 Phoneme Classification with CNN

7.1.1 Logmel spectrogram images

On the test set the model had an accuracy of 12.77%.

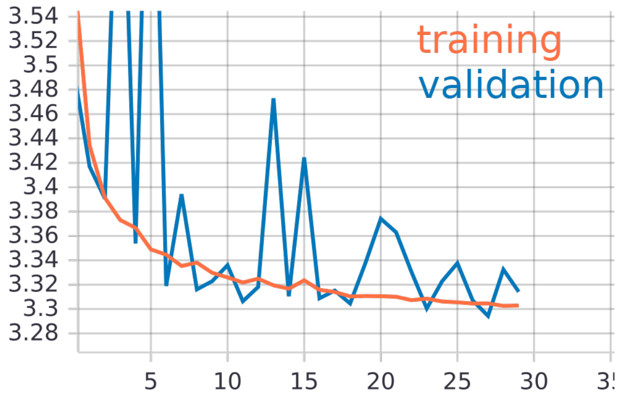


Figure 7.1: Training and validation loss of the CNN model using logmel spectrograms

As we can see in figure 7.1, the convergence of the validation loss is very unstable. While the training loss smoothly declines, the validation loss often varies greatly from epoch to epoch. Additionally, we can see that the training loss is lower than the validation loss on average, which is an indicator for overfitting.

7.1.2 Error analysis

Why did the CNN model fail to classify phonemes using logmel spectrograms? Evenly random guessing out of the 39 phonemes would result in a accuracy of $\frac{1}{39} \approx 2.56\%$. The model had an accuracy of 12.77% . Therefore it did not just guess randomly.

By looking at the confusion matrix at 7.1.2, we can see that the model takes a shortcut to decrease its error. Instead of actually learning to classify the phonemes, it seems to label most phonemes as either 'sil' or 'ix' and some also as 'b', 'd', 'dh', 'dx', 'eh', 'n' or 's', while all other phonemes are never predicted. Comparing this to the frequency distribution at 6.2.3, we can conclude that what the model actually learns, is to classify every input as one of the most frequent phonemes.

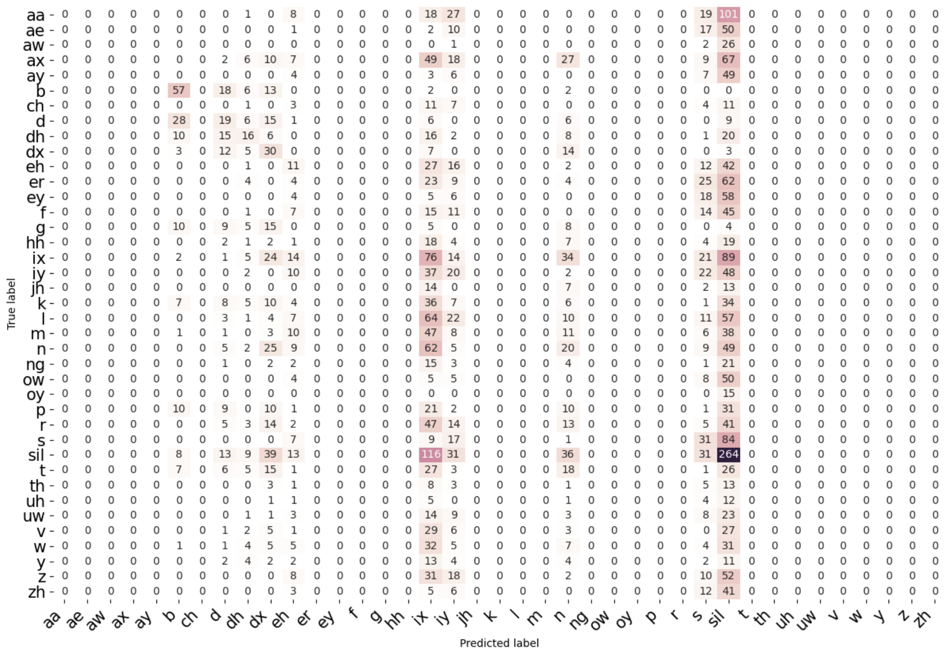


Figure 7.2: Confusion matrix of the CNN model using logmel spectrograms

7.1.2.1 Not enough training data?

First one might assume, that there is not enough training data with only 3512 utterances. However, since every utterance contains several phonemes, this amounts to 82.760 images of the 48 phonemes and therefore 1724 images per class on average. While this is not much, it is supposed to be sufficient for training a CNN like InceptionV3 with transfer learning.

7.1.2.2 Overfitting?

There seems to be a small tendency towards overfitting as discussed at 7.1.1. We have taken several strategies against this beforehand, using regularization techniques as described at 6.3.1. Nevertheless, the difference between training and validation loss is probably not large enough to explain the shortcut behaviour of the network.

7.1.2.3 Bad representation of data?

Since the same model performed reasonably good using decibel-scale spectrograms as discussed in the next section, it is very likely that we either chose the wrong parameters in our logmel extraction code or the resulting spectrograms are not well suited for the CNN architecture. Nevertheless it is an interesting case to see, what a model does to increase its accuracy, when it cannot learn from the provided data.

7.1.3 Decibel-scale spectrogram images

On the test set the model had an accuracy of 66.68%.

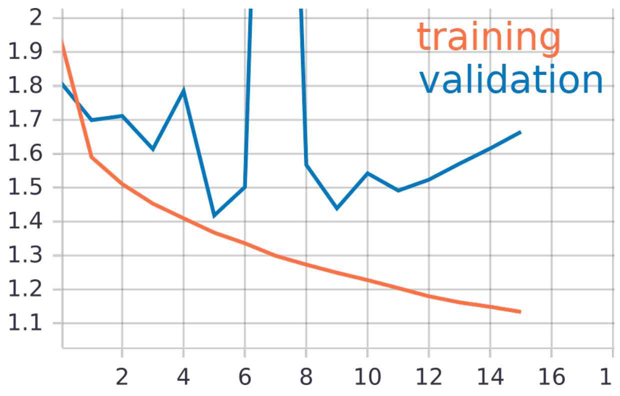


Figure 7.3: Training and validation loss of the CNN model using db-scale spectrograms

We stopped the training early after 15 epochs since the model overfitted.
As we can see in figure 7.3, the model starts to overfit from epoch 6, since the training and validation loss start to diverge. There is also a highest point in validation loss at epoch 7, where the model probably overfitted the most, since the training loss curves continue smoothly at this point.

7.1.3.1 Error analysis

From the confusion matrix in figure 7.4 we can reason that the model is able to classify the phonemes quite well, since major confusions such as ’m’ and ’n’ or or ’s’ and ’z’ are also hard to distinguish for humans.

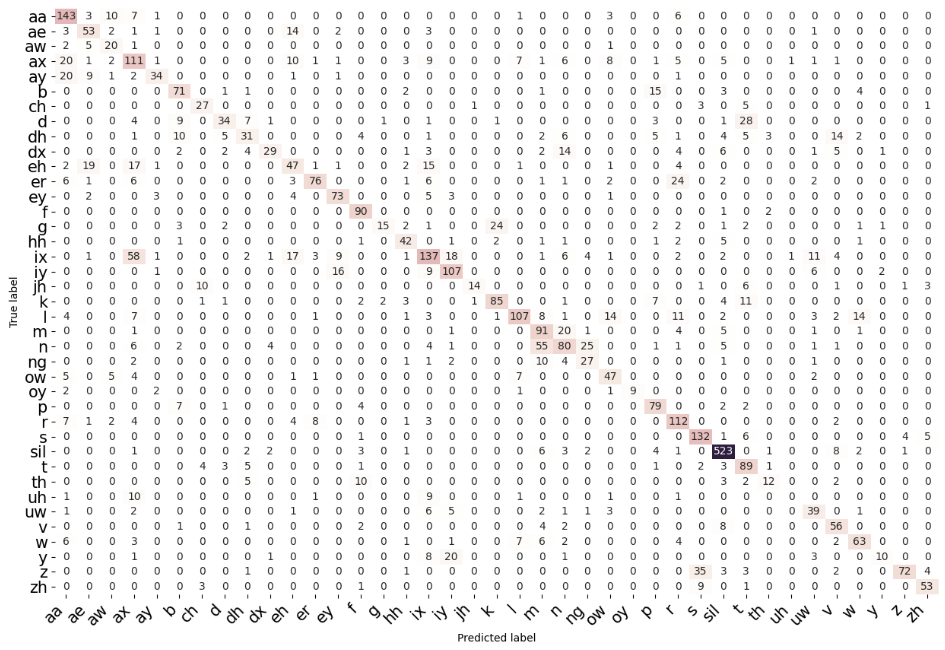


Figure 7.4: Confusion matrix of the CNN model using db-scale spectrograms

7.2 Phoneme Recognition

7.2.1 Encoder Transformer

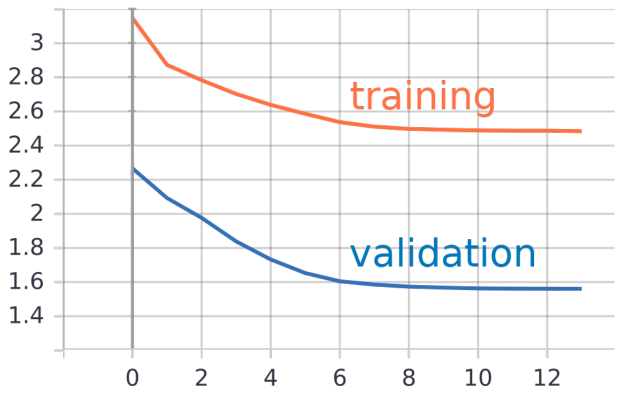


Figure 7.5: Training and validation loss of the Encoder Transformer model

On the core test set the model had an FER of 36.3%. The PER was also very high with 87.8%. For the F1 score we obtained 63.7%, which is a rather mediocre value. As we can see in figure 7.5, the convergence of the loss curves is very fast, which is likely due to the *min_improvement_factor* of 0.95, which leads to faster decrease of the learning rate and therefore a faster flattening of the curve.

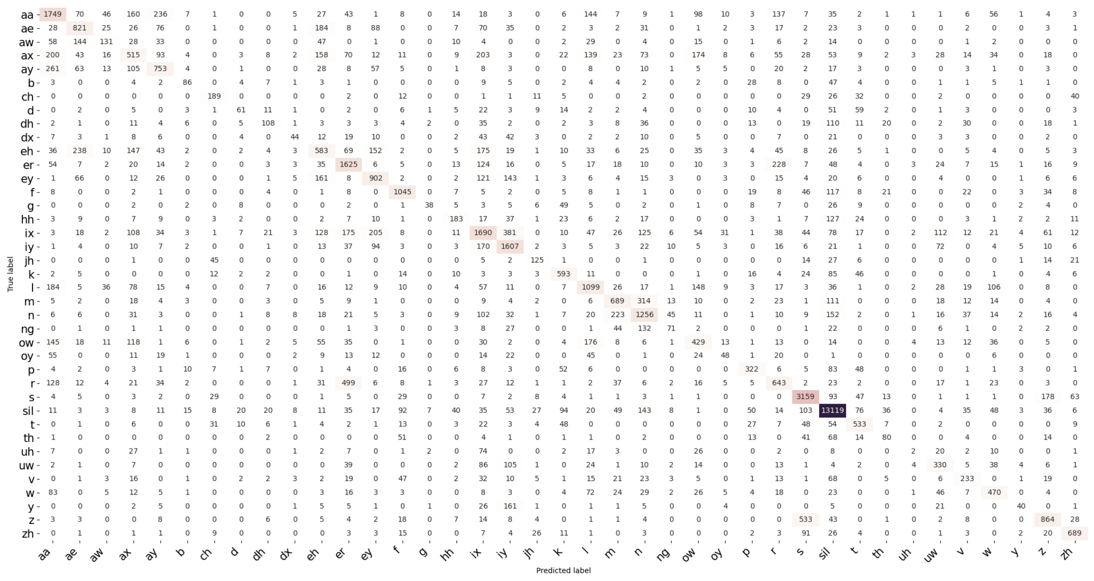


Figure 7.6: Confusion matrix of the Encoder Transformer model

7.2.1.1 Error analysis

From the confusion matrix in figure 7.2.1, we can reason that the most common mistakes of the model are again phonemes, that are also hard to distinguish for humans, such as ’s’ and ’z’. Another significant feature is the quadratic box of high classification counts in the left top corner, which contains confusions between different ’a’ sounds. These are hard to differentiate for humans too. Furthermore, a vertical line of high classifications counts goes through the phoneme ’sil’, which is the silence phoneme. It means that the model predicts silence, even though the correct label would be a different phoneme. The reason for this are probably mainly glottal stops, since they are short frames of silence inside sentence which makes it rather difficult to localize them exactly.

7.2.2 BiRNNs

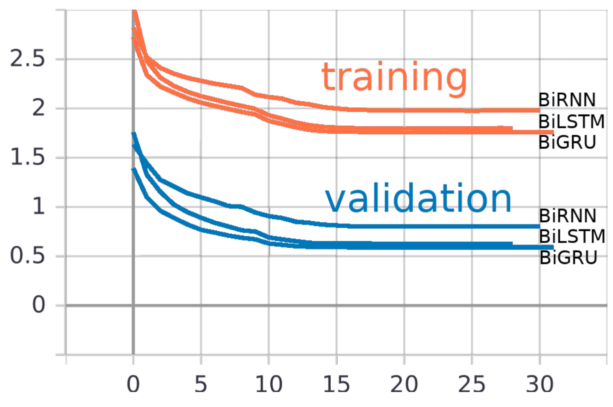


Figure 7.7: Training and validation loss of the RNN, LSTM and GRU model

The BiRNN achieved a FER value of 20.9%, which is state-of-the-art. Its PER and F1 values were 28.3% and 79.9% respectively. For the BiLSTM, we obtained a FER value of 18.4%, which is much better than the BiRNN. Its PER and F1 values were 21.9% and 81.6%. The best result was achieved by the BiGRU with 18.2% FER, 21.0% PER and a F1 score of 81.8%.
As we can see in 7.7, all RNN models have a very similiar convergence, with BiRNN performing slightly worse and BiLSTM and BiGRU being almost on a par. The lower performance of the BiRNN model is likely due to it missing a mechanism, to capture long term dependencies in the input sequence. In 7.7we can also see, that the training loss is much higher than the validation loss, which is due to the high amount of regularization by dropout and weight decay. Overall the loss curves converge in a very stable way, without major outliers.

7.2.3 Error analysis

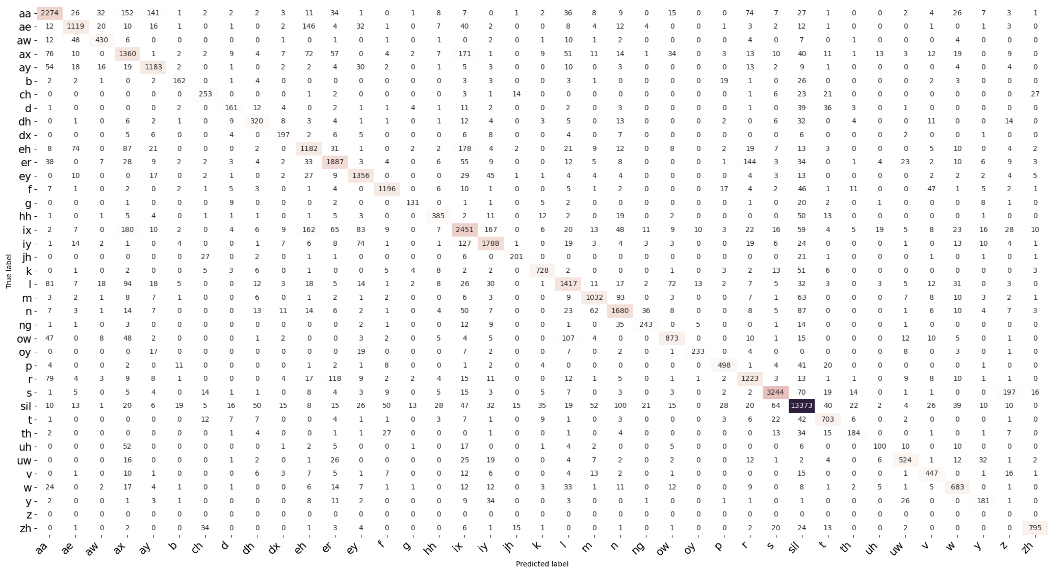


Figure 7.8: Confusion matrix of the GRU model for phoneme recognition

The confusion matrix in figure 7.8 has the same significant features as the EncoderTransformer model at 7.2.1.

7.2.3.1 Recognition vs. alignment accuracy

One might wonder why the alignment accuracy for 20ms with 86.0% is not much better than the phoneme recognition accuracy with 100% − FER = 100% − 18.2% = 81.8%. The reason for this is that they are using completely different metrics. While we measure the accuracy for for phoneme recognition based on the frame error rate, the accuracy for alignment is measured based on the number of phoneme transitions, that are within the tolerance, e.g. 20ms as explained in chapter 4.

7.2.3.2 10ms performance drop

Another result that stands out, is the massive drop in alignment accuracy between 10ms and 20ms tolerance, as seen in 7.7. This does not happen to such a great degree in [25]. In comparison to other works, they do not only use MFCCs for input features, but also their first and second derivatives. Possibly these derivatives contain additional useful information, especially with respect to phoneme transitions. Another likely reason is the fundamentally different architecture of [25], since the performance drop also happens for the other HMM+ANN based model by [20], which is much more comparable to ours than [25].

7.3 Further evaluation of the best model (GRU)

In this section, we perform further experiments on the best model (GRU). The training configuration was described at enumeration 6.3.2.3. Apart from that, the early stopping patience is lowered to 3 epochs, since this has little impact on the model performance and greatly reduces training time.

7.3.1 Performance analysis

7.3.1.1 Impact of layer count and hidden sizes

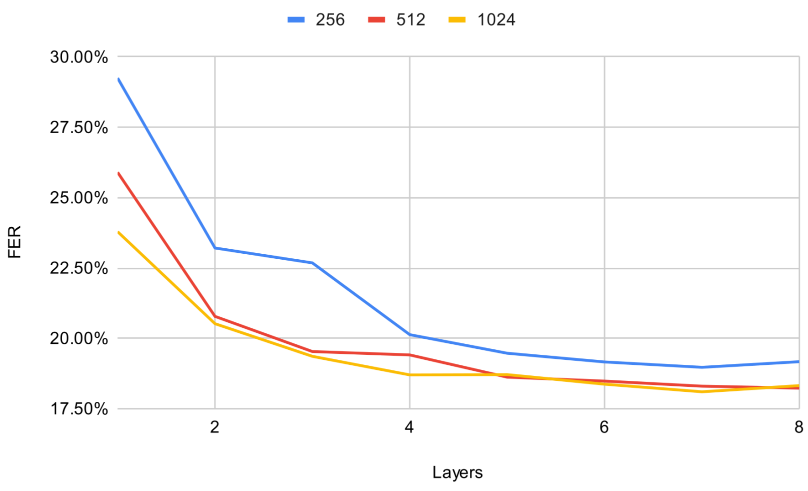


Figure 7.9: FER of the best model (GRU) for different amounts of layers and hidden sizes

From figure 7.9 we can conclude, that the amount of layers has the most impact on the FER value of the model. A larger hidden size of 512 compared to 256 also has a great influence on FER, but there is little difference between a hidden size of 512 and 1024. Nevertheless, we were able to gain a slight improvement over our previous FER value, with 18.1% FER for 7 layers and 1024 hidden states. PER and F1 score were 21.2% and 81.8% respectively.

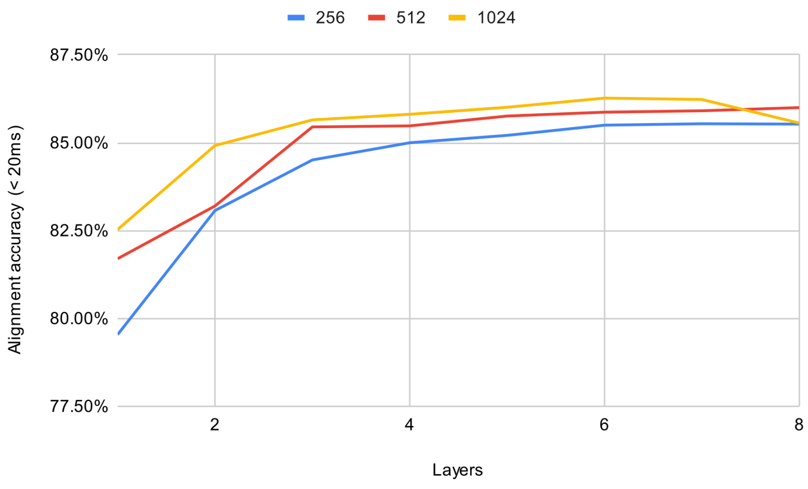


Figure 7.10: Alignment accuracy (< 20ms) of the best model (GRU) for different amounts of layers and hidden sizes

In figure 7.10, we can see that the alignment accuracy also highly depends on the amount of layers. While the model with 1024 hidden states clearly outperforms the other models for 3 layers and more, it declines again for 8 layers. A reason for this could be, that it has now too many neurons available, so it stores some individual patterns of the data, which negatively affects generalization.
Apart from tha,t the model with 6 layers and 1024 hidden states achieved an alignment accuracy of 86.3%, which is 0.3% better than previous results. The accuracies for other tolerances were as follows: 10ms: 49.6%, 30ms: 93.1%, 40ms: 95.8%.

7.3.1.2 Alignment accuracy for higher time tolerances

Since highlighting the spoken text segment for a playing audiofile does not be that precise in time, it is also useful to evaluate the performance of the model for larger tolerances.

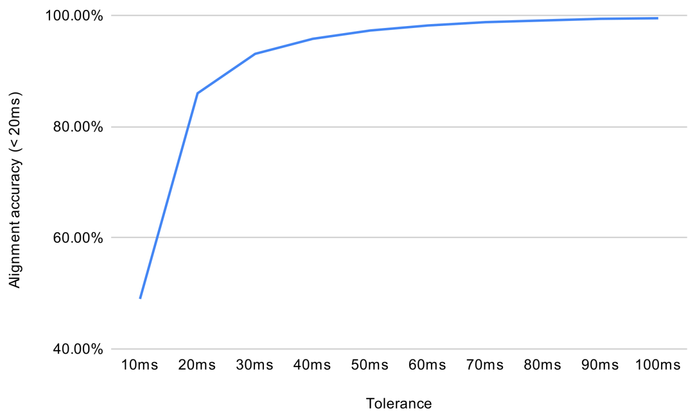


Figure 7.11: Alignment accuracy of the best model (GRU) for higher tolerances

In figure 7.11, we can see that the alignment accuracy converges to almost 100% for 100ms tolerance.

7.3.1.3 Impact of loss weights

Loss weights	**< 10ms**	**< 20ms**	**< 30ms**	**< 40ms**
4,4	46.7%	84.4%	92.3%	95.5%
20,20	48.4%	84.7%	92.0%	95.2%
50,50	48.5%	84.6%	91.9%	95.0%
100,100	49.3%	85.7%	91.9%	95.0%
2,4,4,2	46.6%	84.8%	92.6%	95.7%
10,20,20,10	48.1%	85.4%	92.9%	95.6%
20,50,50,20	48.7%	85.6%	92.8%	95.6%
50,100,100,50	**48.7%**	**86.0%**	**93.1%**	**95.8%**
1,2,4,4,2,1	46.7%	84.7%	92.5%	95.5%
5,10,20,20,10,5	48.2%	85.6%	92.9%	95.7%
10,20,50,50,20,10	48.4%	85.6%	92.9%	95.7%
20,50,100,100,50,20	**48.7%**	**86.0%**	**93.1%**	95.7%

Table 7.1: Impact of different loss weights on the alignment accuracy of the best model (GRU)

From table 7.1, we can conclude, that the loss weights (50,100,100,50) achieve the best alignment accuracy overall. Higher values lead to a better alignment accuracy.

7.3.1.4 Impact of cost functions for DTW

Cost function	**< 10ms**	**< 20ms**	**< 30ms**	**< 40ms**
$cost_{exponential}$	48.4%	85.6%	92.7%	95.4%
$cost_{probability}$	48.4%	85.6%	92.7%	95.4%
$cost_{root\_2}$	48.6%	85.8%	92.9%	95.6%
$cost_{root\_5}$	**48.7%**	85.9%	93.0%	95.7%
$cost_{root\_10}$	**48.7%**	**86.0%**	**93.1%**	**95.8%**
$cost_{root\_100}$	**48.7%**	**86.0%**	**93.1%**	**95.8%**
$cost_{root\_1000}$	**48.7%**	**86.0%**	**93.1%**	**95.8%**

Table 7.2: Impact of different DTW cost functions on the alignment accuracy of the best model (GRU)

In table 7.2, we can see that $cost_{exponential}$ and $cost_{probability}$ both lead to the same alignment accuracy, while $cost_{root\_k}$ gives better results as k increases. This is likely due to the small curvature of $cost_{exponential}$ between $p = 0$ and $p = 1$, which makes its shape very similiar to $cost_{probability}$. Further improvements in accuracy stagnate from $k = 10$.

7.3.1.5 Impact of data augmentation

We experimented with typical data augmentation techniques, which are described at 2.4.8. The following parameters were chosen:

- **Pitch shift:** -4 semitones to 4 semitones (randomly)
- **Time stretch:** 80% to 120% speed (randomly)
- **Frequency mask:** 20 mel bins (randomly)
- **Time mask:** every frame with probability 10%

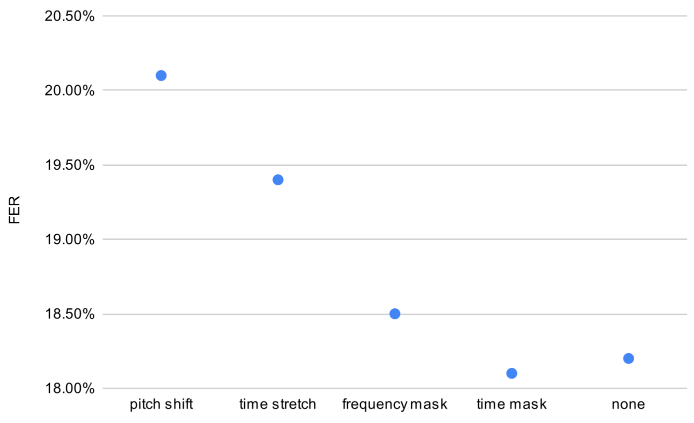


Figure 7.12: FER of the best model (GRU) for different types of data augmentation

In figure 7.13 we see that most data augmentation techniques increase the frame error, but time masking decreases it slightly.

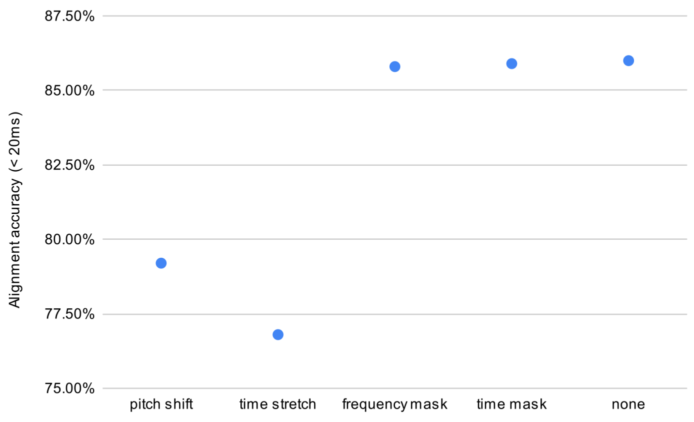


Figure 7.13: Alignment accuracy of the best model (GRU) for different types of data augmentation

From figure 7.13 we can conclude that our data augmentation has no positive effect on alignment. All values for alignment accuracy were lower than without data augmentation.

7.3.1.6 Runtime measurement

In this subsection, we measure the average inference time of a single sentences by the model. The GRU model, with different layer counts and hidden states was evaluated on a AMD Ryzen 3700x CPU and a NVIDIA RTX 2070 Super GPU. Results are shown in table 7.3.

Layers	Hidden states	Time CPU (ms)	Time GPU (ms)
3	256	64.3	42.0
5	512	291.9	55.7
8	1024	1898	192.9

Table 7.3: Impact of different DTW cost functions on the alignment accuracy of the best model (GRU)

7.4 Summary of phoneme recognition results

Method	FER	PER	F1
EncoderTransformer	36.3%	87.8%	63.7%
BiRNN	20.9%	28.3%	79.1%
BiLSTM	18.4%	21.9%	81.6%
BiGRU	**18.1%**	**21.2%**	**81.8%**

Table 7.4: Results of phoneme recognition on the core test set, using different architectures

From table 7.4, we can conclude that the optimized BiGRU model performed the best in all metrics, while being followed by the BiLSTM model.

7.4.1 Comparison with related works

Ref	Method	FER	PER	F1
[37]	CNN+HTSVM	28%	32%	63%
[4]	HDRN+HMM	24.16%	23.60%	-
[39]	CNN+RNN+CTC	22.1%	29.4%	-
Ours	BiGRU	**18.1%**	**21.2%**	**81.8%**

Table 7.5: Results of phoneme recognition on the core test set, compared with related work

As we can see in table 7.5, the BiGRU model outperforms the compared related work in all metrics.

7.5 Summary of phoneme alignment results

Model	**< 10ms**	**< 20ms**	**< 30ms**	**< 40ms**
EncoderTransformer	34.2%	72.1%	83.3%	88.7%
BiRNN	47.4%	84.2%	91.4%	94.5%
BiLSTM	47.8%	85.1%	92.3%	95.1%
BiGRU	**49.6%**	**86.3%**	**93.1%**	**95.8%**

Table 7.6: Results of phoneme alignment on the full test set, using different architectures

In table 7.6, our final results for phoneme alignment are shown.

7.5.1 Comparison with related work

Ref	**Method**	**< 10ms**	**< 20ms**	**< 30ms**	**< 40ms**
[25]	Discr. learning	**80.0%**	92.3%	96.4%	**98.2%**
[20]	HMM+ANN	48.3%	93.4%	**96.8%**	**98.2%**
[41]	HMM+ANN	-	**96.8%**	-	-
Ours	BiGRU+DTW	49.6%	86.3%	93.1%	95.8%

Table 7.7: Results of phoneme alignment on the full test set, compared with related work

As we can see in table 7.7, the BiGRU model does not perform as good as related work for small tolerances, but is closer to them for higher tolerances.
For 10ms, the alignment accuracy is slightly better than [20], but far below [25].
For 20ms, the accuracy is 6.0% below [25] and 10.5% below [41].
For 30ms, and 40ms the alignment gets significantly better, with only 3.7% and 2.4% below the best related work.

8 Conclusion

We have observed the importance of contextual information for phoneme recognition as our RNN models performed much better than the context-free phoneme classification model. Compared to more advanced architectures these primitive RNN based architectures did achieve a lower frame error for phoneme recognition on TIMIT all resulting in state-of-the-art FER values between 18.1%-20.9%. This is very likely because of the higher complexity of the advanced models making them more susceptible to overfitting on such a small dataset. While our alignment resulting in 86.3% accuracy with 20ms tolerance did not achieve state-of-the-art it is still comparable to what a good HMM model would achieve, whose performance usually lies between 80%-89% [41]. Another advantage of the simple architecture is the high flexibility which allows scaling the layers and hidden size of the RNNs to optimize for either accuracy or runtime. Furthermore the probability-based DTW alignment and most improvements described at 6.5 can be used in conjunction with any phoneme recognition model that outputs frame-wise probabilies.

8.1 Future work

8.1.1 Improvements on the models presented in this work

- Replacing the GRU model with Light Gated Recurrent Units (LiGRU), as proposed in [32] may improve recognition and alignment performance, while reducing the computational cost
- Adding convolutional layers or a lightweight CNN before the RNN may improve accuracy, since they are very good for feature extraction
- Combining the phoneme recognition model with a n-gram language model on phoneme level may reduce confusions between similiar sounding phonemes, such as 's' and 'z', as it could give more context, that is not captured by the hidden state.
- Mapping to the phoneme set proposed by [20] may improve alignment accuracy, since it may be better suited for alignment
- Replacing DTW with the approximation algorithm FastDTW [34] would further reduce inference time, as it only has linear time and space complexity, while DTW has quadratic complexities. Although the impact on alignment accuracy would have to be evaluated.
- Adding boundary correction models as proposed in [41] may further improve the alignment performance.

- Using Soft-DTW, as proposed in [12] as a differentiable loss function may also improve the alignment performance, as it would directly optimize for alignment.

8.1.2 Multi-language alignment

To make the model applicable to multi-language alignment based on a single model we created a phoneme based approach which could be modified in future work to achieve this goal. In practice it would be useful to adjust the phoneme set so it can deal better with various languages. This could be achieved by mapping the phonemes to the International Phonetic Alphabet (IPA). A mapping between TIMIT and several phonetic alphabets can be found at [1]. Reducing the phonemes to a minimum amount that are common in most languages may be another approach. Furthermore a dictionary that converts a transcript to a phoneme sequence would be needed which could for instance be created by adjusting an existing phonetic dictionary to the given phoneme set or using incremental autolabeling.

Bibliography

[1] [Online; accessed 21-September-2021]. URL: https://www.isip.piconepress.com/projects/switchboard/doc/education/phone_comparisons/.

[2] Afshine Amidi and Shervine. *Recurrent neural networks cheatsheet star.* URL: https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks.

[3] Aphex34. *File:Typical cnn.png — Wikimedia Commons.* https://commons.wikimedia.org/wiki/File:Typical_cnn.png - License: https://creativecommons.org/licenses/by-sa/4.0/legalcode. [Online; accessed 06-August-2021]. 2015.

[4] Itamar Arel et al. "Acoustic spatiotemporal modeling using deep machine learning for robust phoneme recognition". In: *Afeka-AVIOS Speech Processing Conference.* 2011.

[5] Alexei Baevski et al. "wav2vec 2.0: A framework for self-supervised learning of speech representations". In: *arXiv preprint arXiv:2006.11477* (2020).

[6] Pierre Baldi and Peter J Sadowski. "Understanding dropout". In: *Advances in neural information processing systems* 26 (2013), pp. 2814–2822.

[7] Miguel Bautista et al. "Probability-Based Dynamic Time Warping for Gesture Recognition on RGB-D Data". In: vol. 7854. Nov. 2012. DOI: 10.1007/978-3-642-40303-3_14.

[8] Cecbur. *File:filters in a Convolutional Neural Network.gif — Wikimedia Commons.* https://commons.wikimedia.org/wiki/File:3_filters_in_a_Convolutional_Neural_Network.gif - License: https://creativecommons.org/licenses/by-sa/4.0/legalcode. [Online; accessed 07-August-2021], converted to PNG. 2019.

[9] Guillaume Chevalier. *File:The LSTM Cell.svg — Wikimedia Commons.* https://commons.wikimedia.org/wiki/File:The_LSTM_Cell.svg - License: https://creativecommons.org/licenses/by-sa/4.0/legalcode. [Online; accessed 08-August-2021], converted to PNG. 2018.

[10] Kyunghyun Cho et al. "Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: *arXiv preprint arXiv:1406.1078* (2014).

[11] *CrossEntropyLoss - PyTorch Documentation.* [Online; accessed 22-August-2021]. URL: https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html.

[12] Marco Cuturi and Mathieu Blondel. "Soft-DTW: a Differentiable Loss Function for Time-Series". In: (2018). arXiv: 1703.01541 [stat.ML].

[13] J. Garofolo et al. "TIMIT Acoustic-phonetic Continuous Speech Corpus". In: *Linguistic Data Consortium* (Nov. 1992). [Online; accessed 2021-08-07].

[14] John S. Garofolo et al. *TIMIT Acoustic-Phonetic continuous Speech corpus.* 1993. URL: https://catalog.ldc.upenn.edu/LDC93S1.

[15] Felix A Gers and Jürgen Schmidhuber. "Recurrent nets that time and count". In: *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium.* Vol. 3. IEEE. 2000, pp. 189–194.

[16] Yoav Goldberg. "A primer on neural network models for natural language processing". In: *Journal of Artificial Intelligence Research* 57 (2016), pp. 345–420.

[17] Sepp Hochreiter. "The vanishing gradient problem during learning recurrent neural nets and problem solutions". In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6.02 (1998), pp. 107–116.

[18] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-term Memory". In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: `10.1162/neco.1997.9.8.1735`.

[19] M. Hohenwarter et al. *GeoGebra.* `http://www.geogebra.org`. Dec. 2013.

[20] John-Paul Hosom. "Speaker-independent phoneme alignment using transition-dependent states". In: *Speech Communication* 51.4 (2009), pp. 352–368.

[21] *Human ear - Transmission of sound within the inner ear.* [Online; accessed 10-August-2021]. URL: `https://www.britannica.com/science/ear`.

[22] Incfk8. *File:Bidirectional recurrent neural network.png — Wikimedia Commons.* `https://commons.wikimedia.org/wiki/File:Bidirectional_recurrent_neural_network.png` - License: `https://creativecommons.org/licenses/by-sa/4.0/legalcode`. [Online; accessed 08-August-2021], converted to PNG. 2020.

[23] Ixnay. *File:Recurrent neural network unfold.svg — Wikimedia Commons.* `https://commons.wikimedia.org/wiki/File:Recurrent_neural_network_unfold.svg` - License: `https://creativecommons.org/licenses/by-sa/4.0/legalcode`. [Online; accessed 06-August-2021]. 2017.

[24] Jeblad. *File:Gated Recurrent Unit, type 3.svg — Wikimedia Commons.* `https://commons.wikimedia.org/wiki/File:Gated_Recurrent_Unit,_type_3.svg` - License: `https://creativecommons.org/licenses/by-sa/4.0/legalcode`. [Online; accessed 08-August-2021]. 2018.

[25] Joseph Keshet et al. "Phoneme alignment based on discriminative learning." In: Jan. 2005, pp. 2961–2964.

[26] K.-F. Lee and H.-W. Hon. "Speaker-independent phone recognition using hidden Markov models". In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 37.11 (1989), pp. 1641–1648. DOI: `10.1109/29.46546`.

[27] LeviViana. *How to weight the loss? - PyTorch Forums.* [Online; accessed 26-August-2021]. Jan. 2020. URL: `https://discuss.pytorch.org/t/how-to-weight-the-loss/66372/4`.

[28] Ilya Loshchilov and Frank Hutter. "Decoupled weight decay regularization". In: *arXiv preprint arXiv:1711.05101* (2017).

[29] Dominic Masters and Carlo Luschi. "Revisiting Small Batch Training for Deep Neural Networks". In: (2018). arXiv: `1804.07612 [cs.LG]`.

[30] Michael A Nielsen. *Neural networks and deep learning*. Vol. 25. Determination press San Francisco, CA, 2015.

[31] Paskari. *File:Feed forward neural net.gif — Wikipedia, The Free Encyclopedia*. `http://en.wikipedia.org/w/index.php?title=File%3AFeed%20forward%20neural%20net.gif&oldid=468419692` - License: `https://creativecommons.org/licenses/by-sa/3.0/legalcode`. [Online; accessed 05-August-2021], converted to PNG; modified: missing connecting lines were added. 2021.

[32] Mirco Ravanelli et al. "Light Gated Recurrent Units for Speech Recognition". In: *IEEE Transactions on Emerging Topics in Computing* 2 (Mar. 2018). DOI: `10.1109/TETCI.2017.2762739`.

[33] *ReduceLROnPlateau - PyTorch Documentation.* [Online; accessed 29-August-2021]. URL: `https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.ReduceLROnPlateau.html`.

[34] Stan Salvador and Philip Chan. "Toward accurate dynamic time warping in linear time and space". In: *Intelligent Data Analysis* 11.5 (2007), pp. 561–580.

[35] Mike Schuster and Kuldip K Paliwal. "Bidirectional recurrent neural networks". In: *IEEE transactions on Signal Processing* 45.11 (1997), pp. 2673–2681.

[36] Pavel Senin. "Dynamic time warping algorithm review". In: *Information and Computer Science Department University of Hawaii at Manoa Honolulu, USA* 855.1-23 (2008), p. 40.

[37] Christopher Dane Shulby et al. "Robust Phoneme Recognition with Little Data". In: *8th Symposium on Languages, Applications and Technologies (SLATE 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2019.

[38] Leslie N Smith. "Cyclical learning rates for training neural networks". In: *2017 IEEE winter conference on applications of computer vision (WACV)*. IEEE. 2017, pp. 464–472.

[39] William Song and Jim Cai. "End-to-end deep neural network for automatic speech recognition". In: *Standford CS224D Reports* (2015).

[40] Nitish Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting". In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.

[41] Andreas Stolcke et al. "Highly accurate phonetic segmentation using boundary correction models and system fusion". In: *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2014, pp. 5552–5556.

[42] Christian Szegedy et al. "Rethinking the Inception Architecture for Computer Vision". In: *CoRR* abs/1512.00567 (2015). arXiv: `1512.00567`. URL: `http://arxiv.org/abs/1512.00567`.

[43] Twan Van Laarhoven. "L2 regularization versus batch and weight normalization". In: *arXiv preprint arXiv:1706.05350* (2017).

[44] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems*. 2017, pp. 5998–6008.

[45] Alex Waibel et al. "Phoneme recognition using time-delay neural networks". In: *IEEE transactions on acoustics, speech, and signal processing* 37.3 (1989), pp. 328–339.

[46] *Waveform.* [Online; accessed 10-August-2021]. URL: https://www.merriam-webster.com/dictionary/waveform.

[47] *What is A SPECTROGRAM?* [Online; accessed 10-August-2021]. URL: https://pnsn.org/spectrograms/what-is-a-spectrogram.

[48] Wikipedia. *Artificial neural network — Wikipedia, The Free Encyclopedia.* http://en.wikipedia.org/w/index.php?title=Artificial%20neural%20network&oldid=1034200791. [Online; accessed 03-August-2021]. 2021.

[49] Wikipedia. *Convolutional neural network — Wikipedia, The Free Encyclopedia.* http://en.wikipedia.org/w/index.php?title=Convolutional%20neural%20network&oldid=1037144532. [Online; accessed 06-August-2021]. 2021.

[50] XantaCross. *File:File:Euclidean vs DTW.jpgg — Wikimedia Commons.* https://commons.wikimedia.org/wiki/File:Euclidean_vs_DTW.jpg - License: https://creativecommons.org/licenses/by-sa/3.0/legalcode. [Online; accessed 12-October-2021]. 2011.

[51] Yong Yu et al. "A Review of Recurrent Neural Networks: LSTM Cells and Network Architectures". In: *Neural Computation* 31.7 (July 2019), pp. 1235–1270. ISSN: 0899-7667. DOI: 10.1162/neco_a_01199. eprint: https://direct.mit.edu/neco/article-pdf/31/7/1235/1053200/neco_a_01199.pdf. URL: https://doi.org/10.1162/neco%5C_a%5C_01199.