Karlsruhe Institute of Technology

Bachelor thesis

Cross-lingual, Language-independent Phoneme Alignment

Niklas Bühler

Handover date: 01.10.2021

Referees: Prof. Dr. Alexander Waibel
Prof. Dr.-Ing. Tamim Asfour
Advisors: Dr. Sebastian Stüker
M.Sc. Thai-Son Nguyen
M.Sc. Christian Huber

Interactive Systems Labs
Institute for Anthropomatics and Robotics
Department of Informatics
Karlsruhe Institute of Technology

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, 01.10.2021

Abstract

When documenting an unknown language – and especially its pronunciation – for the first time, linguists are oftentimes missing the necessary technology to do so efficiently. The work described in this thesis might facilitate future advancements in this regard by further developing techniques that make the computer-assisted examination of languages more efficient.

However, not only the process of manually examining an unseen language is very costly. Even in cases where automatic speech recognition technology is already utilized, the necessary process of data collection and preparation still remains costly. Cross-lingual approaches can alleviate this problem.

The goal of this thesis is to apply cross-lingual, multilingual techniques on the task of *phoneme alignment*, i.e. the task of temporally aligning a phonetic transcript to its corresponding audio recording.

Three different neural network architectures are trained on a multilingual data set and utilized as a source of emission probabilities in hybrid HMM/ANN systems. These HMM/ANN systems enable the computation of phoneme alignments via the Viterbi algorithm. By iterating this process, multilingual acoustic models are bootstrapped and the resulting systems are used to cross-lingually align data from a previously unseen target language. Finally, the results are scored and compared against each other.

Zusammenfassung

Linguisten fehlt oftmals die nötige Technologie, um unbekannte Sprachen – und vor allem deren Aussprache – effizient zu dokumentieren. Diese Arbeit kann zukünftige Vorhaben in diesem Bereich unterstützen, indem Methoden zur computergestützten Untersuchung von Sprachen weiterentwickelt werden.

Selbst wenn bereits automatische Spracherkennung im Einsatz ist, bleibt die hierfür notwendige Datenerfassung sehr aufwendig. Crosslinguale Ansätze können dieses Problem abschwächen.

Das Ziel dieser Arbeit ist es, cross- und multilinguale Methoden für das *Phonem-Alignment*, also das zeitliche Zuordnen von Phonemen und zugehörigen Sprachaufnahmen, anzuwenden.

Drei verschiedene Architekturen neuronaler Netze werden auf einem multilingualen Datensatz trainiert und als Quelle von Emissionswahrscheinlichkeiten in hybriden HMM/ANN Systemen eingesetzt. Diese HMM/ANN Systeme ermöglichen das Berechnen von Phonem-Alignments mittels des Viterbi-Algorithmus. Durch das Iterieren dieses Prozesses werden multilinguale akustische Modelle geschaffen und die resultierenden Systeme werden crosslingual verwendet, um Phonem-Alignments in einer bisher ungesehenen Sprache zu berechnen. Abschließend werden die Resultate der verschiedenen Systeme bewertet und miteinander verglichen.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Research Question . . . 1
- 1.3 Structure of this Thesis . . . 2

2 Background **3**
- 2.1 Hidden Markov Models . . . 3
  - 2.1.1 The Decoding Problem . . . 4
  - 2.1.2 The Viterbi Algorithm . . . 5
- 2.2 Artificial Neural Networks . . . 5
  - 2.2.1 Classification Problems . . . 6
  - 2.2.2 Feedforward Neural Networks . . . 7
  - 2.2.3 Neural Network Training . . . 13
  - 2.2.4 Time Delay Neural Networks . . . 17
  - 2.2.5 Recurrent Neural Networks . . . 17
  - 2.2.6 Long Short-Term Memory . . . 20

3 Related Work **25**

4 Main Contributions **27**
- 4.1 Hybrid HMM/ANN System . . . 27
- 4.2 Experimental Pipeline . . . 27
  - 4.2.1 Preparation and Preprocessing . . . 27
  - 4.2.2 Bootstrapping a Multilingual Acoustic Model . . . 29
  - 4.2.3 Evaluation . . . 30
- 4.3 Toolkits, Libraries and Data sets . . . 30
  - 4.3.1 Janus Speech Recognition Toolkit . . . 30
  - 4.3.2 PyTorch . . . 30
  - 4.3.3 Common Voice . . . 30
- 4.4 Experiments . . . 30
  - 4.4.1 Monolingual Feedforward Neural Network . . . 31
  - 4.4.2 Multilingual Feedforward Neural Network . . . 31
  - 4.4.3 Multilingual Time Delay Neural Network . . . 33
  - 4.4.4 Multilingual Stacked Bidirectional Long Short-Term Memory . . . 34

5 Evaluation **37**
- 5.1 Scoring Methods . . . 37
- 5.2 Results . . . 37
  - 5.2.1 Monolingual Feedforward Neural Network . . . 37
  - 5.2.2 Multilingual Feedforward Neural Network . . . 38
  - 5.2.3 Multilingual Time Delay Neural Network . . . 39
  - 5.2.4 Multilingual Stacked Bidirectional Long Short-Term Memory . . . 39

6 Conclusion **43**
6.1 Summary 43
6.2 Interpretation of Results 43
6.3 Further Research 44

1 Introduction

1.1 Motivation

There are roughly 7,000 languages spoken today. Because these languages are living, they are dynamically changing over time. And so is their number: Roughly 40% of these currently 7,000 languages are endangered[1] ([ESF21]). Furthermore, only a minor percentage of languages spoken today is sufficiently covered by technology or even by linguistic knowledge. Thus, many endangered languages – and with them a big part of their corresponding cultures – are completely and irreplaceably lost when their last speaker dies, as has been described in [Cry00].

There has been a recent surge in documentary linguistics ([Woo03]), which works towards documenting and thus preserving many of these endangered languages before they're lost for posterity. However, it is not to be assumed that the documentary linguistics community will be able to do so without the aid of automatic processing ([Add+16]).

The work at hand thus aims at improving the necessary technology that linguists are presently missing in order to efficiently document new languages. It does so by further improving on state-of-the-art methods for aligning phonemes to their speech recordings, while especially focusing on under-resourced languages.

Such technology will enable field linguists to better examine and document the pronunciation of a new language.

1.2 Research Question

From a purely technical perspective, the problem to be solved is known as force-alignment of a phonetic transcript to its respective audio recording, i.e. to generate a time-alignment of phonetic transcript and audio signal.

The standard method of producing such an alignment is combining a hidden Markov model (HMM) with a Gaussian mixture model (GMM) and running the Viterbi algorithm on this system ([RJ86]). However, inspired by the rise and ubiquitous success of deep learning, replacing the GMM with an artificial neural network (ANN) for obtaining emission probabilities has proven to be superior. This approach brings together the best from both worlds; the time-alignment capability of HMMs and the discrimination-based learning of ANNs (see also [FLW90]). There have been purely connectionist attempts as well, and although they showed promising results, they did not outperform HMM-based systems yet ([Han+14]).

However, artificial neural networks require large amounts of data for their training. This data is by definition not available when first documenting an as of yet undocumented language. Building a cross-lingual system – applying the system to languages that have not been used for training it – can alleviate this problem.

[1] A language is said to be endangered when its users begin to speak and teach a more dominant language to their children.

In the performed experiments, multiple approaches are compared against each other. First, an ordinary feedforward neural network is trained on a monolingual data set and applied cross-lingually. In further experiments, a multilingual data set is used to train the cross-lingual systems and the feedforward architecture is replaced by various more complex architectures.

1.3 Structure of this Thesis

The remainder of this thesis is structured as follows:

Chapter 2: In the Background chapter, the foundation of this thesis is laid by introducing various important theoretical concepts regarding hidden Markov models and artificial neural networks. This spans not only the general Viterbi algorithm, but also the different network architectures that replace the feedforward network in subsequent experiments.

Chapter 3: In the Related Work chapter, state-of-the-art techniques and results in the field of automatic speech recognition (ASR) are reviewed and presented, especially regarding phoneme recognition and alignment.

Chapter 4: The Main Contributions chapter starts with presenting the HMM/ANN system that is utilized in all experiments. The whole experimental pipeline is described, including the methods used for preprocessing the audio and preparing the data sets, the process of bootstrapping a multilingual acoustic model and the evaluation procedure. Utilized toolkits, libraries and data sets are presented as well. The descriptions of the performed experiments further specify the architectures of the neural networks and special methods used for training them in each experiment.

Chapter 5: In the Evaluation chapter, the results from the previous chapter are evaluated and compared against each other. The scoring functions used to compare these results are also defined.

Chapter 6: Finally, in the Conclusion chapter, the thesis is summarized and the evaluation results are interpreted and put into context. Possible directions for further research are proposed.

2 Background

This chapter serves as an introduction to the most important theoretical concepts this thesis builds upon, spanning hidden Markov models and artificial neural networks.

A Remark on Notation

In the following chapter, the requirement to address a vector's value in a specific time step will oftentimes arise. In cases where the variable is not a vector, x_t is sufficient notation, but in this special case, the notation x_i^t will be used to mean the value of the i-th element of x as of time step t.

2.1 Hidden Markov Models

Generally speaking, a hidden Markov model (HMM) is a pair of stochastic processes (X, Y), of which X is a Markov chain whose behavior is not directly observable. The behavior of Y is dependent on X and – on the contrary – directly observable. HMMs have established themselves as the standard approach used to model the temporal structure of speech in automatic speech recognition ([PJP15]).

Definition 2.1.1. *A* Markov chain *or* Markov process *is a stochastic model describing a sequence of possible events in which the probability of each event depends only on the previous event.*
Formally, a Markov chain is a sequence of random variables $X_1, X_2, X_3, \ldots$ *satisfying the* Markov property

$$P(X_{n+1} = x_{n+1} \mid X_1 = x_1, \ldots, X_n = x_n) = P(X_{n+1} = x_{n+1} \mid X_n = x_n).$$

A hidden Markov model traverses through several time steps. In each time step, it attains a distinct state and – based on the state attained – emits a symbol. Modeling the state transition process is undertaken by a Markov chain. Thus, the HMM changes its state based on a probability distribution that is only dependent on the state it's currently in. It's called hidden, because the exact state sequence is not directly observable from outside the model.

However, there's a second stochastic process that is dependent on this hidden process. This second process is observable; it emits symbols that are said to be output by the HMM. The probabilities defining which symbol is emitted at each time step are determined by the state the model is currently in.

In this thesis – as conventional in ASR and somewhat predetermined by finite precision computers – only discrete-time HMMs that likewise output discrete symbols are considered. Thus, let $T, N, K \in \mathbb{N}$, where T denotes the number of time steps the model traverses. In each time step, the HMM is in one of N possible states and emits one of K possible symbols.

Definition 2.1.2. *A* hidden Markov model *with N states and K observable symbols that traverses T time steps is completely defined as $\lambda = (Q, O, \pi, A, B)$ with*

(i) *A finite* set of states $Q := \{Q_1, Q_2, \ldots, Q_N\}$*, which are the possible values of the Markov chains random variables. The HMM traverses a sequence of attained states* $q = q_1 q_2 \ldots q_T$*, where* $q_t \in Q$ *for* $t \in \{1, \ldots, T\}$*.*

(ii) *A set* $O := \{O_1, O_2, \ldots, O_K\}$ *of* K observable symbols *that are emitted by the second stochastic process as* $o = o_1 o_2 \ldots o_T$ *with* $o_t \in O$ *for* $t \in \{1, \ldots, T\}$*.*

(iii) *The* initial probabilities*, defined by a probability distribution* π*, which gives the probability for starting in state* Q_i *as* $\pi_i := P(q_1 = Q_i)$ *for* $i \in \{1, \ldots, N\}$*.*

(iv) *A matrix* $A \in [0,1]^{N \times N}$ *of* state transition probabilities*, defining the conditional probabilities of transitioning between states as* $A = (a_{ij})$*, where* $a_{ij} = P(q_t = Q_j \mid q_{t-1} = Q_i)$ *for* $i, j \in \{1, \ldots, N\}$ *and* $t \in \{2, \ldots, T\}$*. Note that* a_{ij} *does not take into account the current time step* t*, so state transition probabilities are time-invariant.*

(v) *A matrix* $B \in [0,1]^{N \times K}$ *of* emission probabilities*, defining the conditional probabilities of emitting symbol* O_j *in state* Q_i *as* $B = (b_{ij})$*, with* $b_{ij} = P(o_t = O_j \mid q_t = Q_i)$ *for* $i \in \{1, \ldots, N\}, j \in \{1, \ldots, K\}$ *and* $t \in \{1, \ldots, T\}$*. Sometimes* $b_i(O_j)$ *is written instead of* b_{ij}*, as this allows for notation such as* $b_i(o_t)$*. Similar as with* A*, the emission probabilities given by* B *are time-invariant.*

HMMs can conveniently be depicted similar to the way finite-state automata oftentimes are. For example, the HMM in figure 2.1 consists of $N = |Q| = 2$ states and can output $K = |O| = 2$ different symbols. Its states are given as $Q = \{Q_1, Q_2\}$ and the set of symbols is $O = \{O_1, O_2\}$. The dotted arrows coming out of the *start* label depict the initial probabilities π_1 and π_2 for starting in state Q_1 or Q_2 each. The probabilities for transitioning between states are indicated as arrows $a_{1,1}, a_{1,2}, a_{2,1}$ and $a_{2,2}$. Those for emitting symbols are labeled $b_{1,1}, b_{1,2}, b_{2,1}$ and $b_{2,2}$.

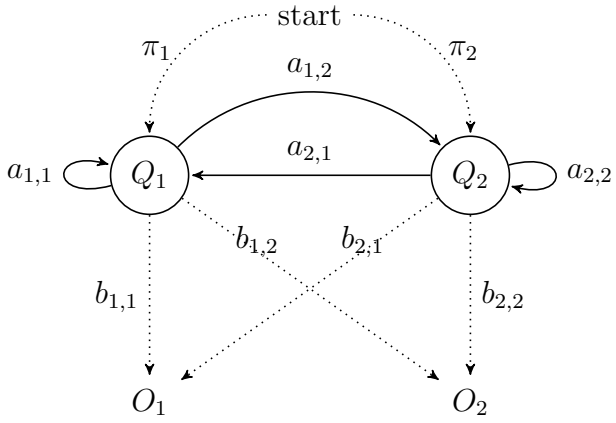


Figure 2.1: Example of an HMM with $N = K = 2$. The probabilities for starting in state Q_i are given by π_i. The variable a_{ij} determines the probability for transitioning from state Q_i to Q_j and b_{ij} gives the probability for outputting symbol O_j while in state Q_i.

Note that this visualization does not take into account the time dimension. The behavior of an HMM over time can be depicted in a *trellis* or *lattice diagram*.

2.1.1 The Decoding Problem

Given the limited information an HMM reveals towards observers, there are several problems that can be formulated concerning the acquisition of more information. One of these problems – the most relevant one regarding this thesis – is the *Decoding problem*.

Definition 2.1.3. *The* Decoding problem *for HMMs is defined as follows:*
Given an HMM λ and a possible observation sequence $o = o_1 o_2 \dots o_T$, what is

$$q^* := \underset{q \in Q^T}{\operatorname{argmax}} P(q, o \mid \lambda),$$

the most probable sequence of states the HMM might have attained while outputting o.

In this thesis, an HMM is used to model different phonemes as states. Their particular emission probabilities model the probabilities of the phoneme to correspond to a certain feature vector, i.e. a processed fraction of recorded speech. The solution to the decoding problem thus yields the most probable sequence of phonemes, which is equivalent to an alignment of audio recording and phonetic transcript. The exact process is further described in chapter 4.1.

2.1.2 The Viterbi Algorithm

The Viterbi algorithm calculates the most probable state sequence $q^* = q_1^* \dots q_T^*$ attained by an HMM while emitting a given sequence of symbols $o = o_1 \dots o_T$. It thus solves the Decoding problem by determining $q^* = \operatorname{argmax}_{q \in Q^T} P(q, o \mid \lambda)$. In algorithm 1, the Viterbi algorithm is given in pseudo code. This depiction is based on [RJ86].

Here, $\delta_t(i)$ describes the probability to end up in state Q_i at time step t on a most probable path. Note that this is *not* the same as the probability to end up in state Q_i summed over all possible paths.

In order to calculate these probabilities, the values for $\delta_1(i)$ are initialized by multiplying the probability π_i to start in state Q_i at time step 1 with the probability $b_i(o_1)$ to output symbol o_1 in state Q_i, resulting in $\delta_1(i) = P(Q_i, o_1 \mid \lambda) = \pi_i b_i(o_1)$. The other values of δ_t can then be calculated recursively over all time steps by setting $\delta_t(j) = \max_{1 \le i \le N}[\delta_{t-1}(i) a_{ij}] b_i(o_t)$ for every possible next state j, as $d_{t-1}(i) a_{ij}$ describes the probability to end up in state Q_i at time step $t-1$ and then make the transition to Q_j. Thus, $d_t(j)$ holds the maximum probability to attain state j in time step t, again constrained to a most probable path. This calculation also yields the most probable state in the previous time step, given by $\Psi_t(j) = \operatorname{argmax}_{1 \le i \le N}[\delta_{t-1}(i) a_{ij}]$.

After recursively calculating all states maximum probabilities $\delta_t(i)$ for each time step and the respective most probable previous states, the results can be obtained as follows: The probability of the most probable state sequence is given by the probability to arrive at the most probable last state, $P^* = \max_{1 \le i \le N}[\delta_T(i)]$. The actual sequence of states can then be obtained by collecting the most probable previous states in reverse order: $q_t^* = \Psi_{t+1}(q_{t+1}^*)$.

The complexity of this implementation is $O(TN^2)$, determined by the main for-loop that loops over all time steps, contributing the factor T, and state transitions, contributing the factor N^2.

2.2 Artificial Neural Networks

Artificial neural networks (ANNs) are a method of computation that is roughly inspired by the way biological brains work ([Ros58], [MP43]). While the central concepts inherent to connectionist models have been around for about half a century now, they could only recently begin to develop their true potential, as computing power and available data massively increased. During the last decade, connectionist models have reached an extensive prevalence, mostly because of their unmatched classification abilities. They are ubiquitously used for a variety of recognition and prediction tasks.

Algorithm 1: The Viterbi Algorithm

Data: HMM $\lambda = (S, V, \pi, A, B)$, output sequence $o = o_1 \dots o_T$
Result: Probability P^* of most probable state sequence $q^* = q_1^* \dots q_T^*$

```
for 1 ≤ i ≤ N do
    δ_1(i) = π_i b_i(o_1)                    // initialize the probabilities for all states in t = 1
end
for 2 ≤ t ≤ T do                             // for all time steps
    for 1 ≤ j ≤ N do                         // for all next states
        δ_t(j) = max_{1≤i≤N}[δ_{t−1}(i) a_ij] b_i(o_t)    // calculate each states probability recursively
        Ψ_t(j) = argmax_{1≤i≤N}[δ_{t−1}(i) a_ij]          // remember the most probable previous state
    end
end
P* = max_{1≤i≤N}[δ_T(i)]                     // total probability of the most probable state sequence
q*_T = argmax_{1≤i≤N}[δ_T(i)]                // most probable state in the last time step
for T − 1 ≥ t ≥ 1 do
    q*_t = Ψ_{t+1}(q*_{t+1})                 // build the most probable state sequence
end
```

In this thesis, the classification capabilities of artificial neural networks are utilized for classifying phonemes. Traditionally, HMMs have used Gaussian mixture models (GMMs) for producing the emission probabilities of their continuous observations in what is called a hybrid HMM/GMM system. However, due to the excellent classification results of ANNs, these systems are nowadays usually replaced by hybrid HMM/ANN systems ([MDH11]). In such a system, it's the ANNs task to supply the HMM with posterior emission probabilities for all HMM states/phonemes, i.e. produce probabilities $P(F \mid Q)$, where F is a feature vector of recorded speech and Q corresponds to a (sub-)state[1] of the HMM.

This section is based on [RN02] and [Goo+16].

2.2.1 Classification Problems

Classification problems are a special case of supervised learning, itself being a subfield of machine learning.

Definition 2.2.1. *The task of* supervised learning *is defined as follows: Given a training set of N sample input-output pairs*

$$(x_1, y_1), \dots, (x_N, y_N),$$

where each pair is determined by an unknown function f as $y_i := f(x_i)$, find a function h which approximates the function f.

A supervised learning problem is called *classification problem* when the possible values of y are finitely many. In this case, the y values are also called *labels* of their corresponding x values. When there are infinitely many possible y-values, the task is called *regression*.

Oftentimes supervised learning tasks are also differentiated by their property of being separable in a linear way (or by a linear classifier), giving rise to the designation of tasks as either *linear*

[1] It's sometimes useful to use multiple states (substates) to model a single phoneme.

or *non-linear*. A popular example of a non-linear classification problem is learning the *XOR-function*, defined as

$$\oplus(x_1, x_2) = x_1 + x_2 \mod 2 = \begin{cases} 1 & \text{if } x_1 \neq x_2 \\ 0 & \text{if } x_1 = x_2 \end{cases}$$

for $x_1, x_2 \in \{0, 1\}$. The different outputs of the XOR-function can not be separated in a linear way, i.e. by a single linear separator, as is shown in figure 2.2, see also [LP91].

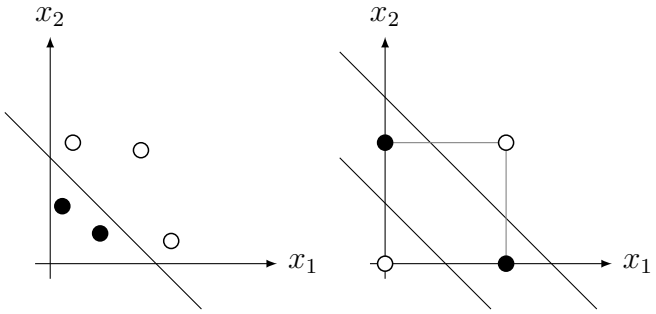


Figure 2.2: A depiction of an arbitrary linear classification problem on the left and the XOR classification problem on the right. The outputs of the XOR-function can not be separated using a single linear separator. White marks outline data points of class 0, black marks those of class 1.

In order to assess the performance of a supervised learning method, the whole data set is usually split up into disjoint *training* and *test sets* that are used for training and evaluation, respectively. The goal is not only to approximate the given pairs $(x_1, y_1), \ldots, (x_N, y_N)$ of the training set, but to achieve generalization. That is, to perform well (approximate f well) even on previously unseen pairs of the test set, that were not part of the training set. Not reaching generalization is a problem known as *overfitting* the data.

Error measures

In order to assess the performance of a supervised learning method, one can apply different error measures. For a binary classification task, data points are classified into the two classes *positive* and *negative*. When considering multiple classes instead, the class at hand would be described as positive and all other classes combined as negative. Both positive and negative predictions can be correct (*true*) or incorrect (*false*).

Table 2.1 shows four common error measures. *Accuracy* intuitively describes the proportion of predictions that were correct. *Precision* describes the proportion of positive predictions that were correct, while *recall* describes the proportion of correctly predicted positives. Finally, the F_1-Score is the harmonic mean of recall and precision, taking on a maximum value of 1 if both, precision and recall, are maximal, and a minimum value of 0 if one of them is 0.

2.2.2 Feedforward Neural Networks

Feedforward neural networks are the simplest type of neural networks. They do not contain any cycles in their connections.

Name	Definition	Calculation
Accuracy	$\frac{\#\text{ correct predictions}}{\#\text{ total predictions}}$	$\frac{\text{TP}+\text{TN}}{\text{TP}+\text{TN}+\text{FP}+\text{FN}}$
Precision	$\frac{\#\text{ correct positive predictions}}{\#\text{ total positive predictions}}$	$\frac{\text{TP}}{\text{TP}+\text{FP}}$
Recall	$\frac{\#\text{ correct positive predictions}}{\#\text{ total true positives}}$	$\frac{\text{TP}}{\text{TP}+\text{FN}}$
F_1-Score	$\frac{2}{\text{recall}^{-1}+\text{precision}^{-1}}$	$2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision}+\text{recall}}$

Table 2.1: Different error measures for classification tasks. The abbreviations are *TP* for *true positives*, *TN* for *true negatives*, *FP* for *false positives* and *FN* for *false negatives*.

The Perceptron

The *perceptron* ([Ros58]) is a simple classifier for binary linear classification problems and the simplest feedforward neural network there is. It's modeled as a simplified version of a biological neuron, adopting the idea that several incoming action potentials are added up in a weighted sum and – in case a certain threshold is exceeded – make the receiving neuron trigger its own action potential as well.

Formally, a perceptron can be described as a single computing unit, receiving N real-valued inputs $x_1, \ldots, x_N \in \mathbb{R}$ and outputting a value between 0 and 1. Inside the perceptron, these N input values are first multiplied by N weights $w_1, \ldots, w_N \in \mathbb{R}$. The weighted inputs are then summed up, constituting a weighted sum, to which a single bias $b \in \mathbb{R}$ is added as well. At last, an activation function is applied on the result of the previous steps. In the case of the standard perceptron, this is a Heaviside step function, which maps all positive values to 1 and all other values to 0, thus performing classification in classes 0 and 1. This whole calculation process is also illustrated graphically in figure 2.3.

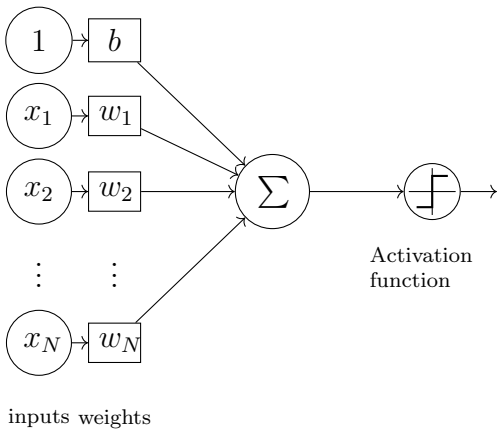


Figure 2.3: A graphical representation of the perceptron model. The N inputs are supplemented with a bias entry b. After totaling the weighted inputs and the bias, the result is fed into a Heaviside step function.

To simplify this calculation, the input scalars and weight scalars can be stacked into an input vector $x \in \mathbb{R}^N$ and a weight vector $w \in \mathbb{R}^N$ respectively. The weighted sum $\sum_{i=1}^{N} x_i w_i$ can thus be described as the dot product $x \cdot w$. The bias scalar b can also be prepended to the weight

vector w as zeroth value w_0. In this case, a 1-entry has to be prepended to the input vector x as zeroth value $x_0 = 1$ as well, yielding

$$x' = \begin{pmatrix} 1 \\ x \end{pmatrix} = \begin{pmatrix} 1 \\ x_1 \\ \vdots \\ x_N \end{pmatrix} \text{ and } w' = \begin{pmatrix} b \\ w \end{pmatrix} = \begin{pmatrix} b \\ w_1 \\ \vdots \\ w_N \end{pmatrix}.$$

The whole calculation can then be written in a single step, using the dot product, as

$$x \cdot w + b = \sum_{i=1}^{N} x_i w_i + b = \sum_{i=0}^{N} x'_i w'_i = x' \cdot w'.$$

In the remainder of this thesis, biases are assumed to be included in the usual weight matrices.

The perceptrons decision rule can now easily be defined as a threshold function that maps an input vector x to an output value $f(x)$ as

$$f(x) = \begin{cases} 1 & \text{if } x \cdot w + b = x' \cdot w' > 0, \\ 0 & \text{otherwise.} \end{cases}$$

This way, the perceptron can classify a given vector $x \in \mathbb{R}^N$ into one of two classes 0 and 1. It does so by implicitly spanning a linear decision boundary in the form of a hyperplane, specified by the equation $wx + b = 0$, defining two disjoint classes inside the $\mathbb{R}^N$. This hyperplane can be learned iteratively by adjusting the perceptrons bias and weights. The directions for these adjustments are given by evaluating the perceptron on sample data and correcting on each wrong classification.

Perceptron Training

As the perceptron is a supervised learning method, there is a training set $\mathbb{S} = \{(x_1, y_1), \ldots, (x_S, y_S)\}$ with $x_i \in \mathbb{R}^N, y_i \in \{0, 1\}$ available for training, which is used to adjust the perceptrons parameters accordingly.

The training algorithm is outlined in pseudo code as algorithm 2. Here, let $w', x'_s \in \mathbb{R}^{N+1}$ denote the *extended* weight and input vectors, as described above. Furthermore, let $x'_{s,i}$ denote the i-th component of vector x'_s.

Algorithm 2: The Perceptron Training Algorithm

`Data`: training set $\mathbb{S} = \{(x_1, y_1), \ldots, (x_S, y_S)\}$, learning rate η
`Result`: Weights w' after training

```
w'_i := 0^{N+1}                          // initialize the weights and bias as zero
for (x_s, y_s) ∈ S do                    // for all training data
    x'_s := x_s.prepend(1)
    ŷ_s := f(x'_s)                       // evaluate the perceptron for x'_s
    w' := w' + η(y_s − ŷ_s)x'_s          // adjust the weights
end
```

The perceptron starts with all weights and the bias initialized as 0 and iteratively updates them along evaluating all labeled training data. This learning process happens as follows:

For every pair of labeled data (x_s, y_s) in the training set $\mathbb{S}$, the perceptrons current prediction of the classification is calculated as $\hat{y}_s = f(x'_s)$. This prediction can be either correct or incorrect, and can be checked against the provided label y_s. The comparison is done implicitly while updating the weights in line 5. In the following, we examine both possible cases:

Case 1: Prediction correct If the prediction was correct, i.e. $\hat{y}_s = y_s$, then the error $e := y_s - \hat{y}_s$ is equal to zero, so the weights $w' := w' + \eta(y_s - \hat{y}_s)x'_s = w' + \eta 0 x'_s = w'$ are not updated.

Case 2: Prediction incorrect However, if the prediction was incorrect, i.e. $\hat{y}_s \neq y_s$, then the error $e := y_s - \hat{y}_s$ is either -1 or 1. Thus, the update is described by

$$w' := w' + \eta(y_s - \hat{y}_s)x'_s = \begin{cases} w' + \eta x'_s & \text{if } e = 1, \\ w' - \eta x'_s & \text{if } e = -1. \end{cases}$$

The adjustment is individual on a per-weight basis. In case the perceptrons prediction was wrong, each of the weights w'_i is corrected by $\eta x'_{s,i}$ into the corresponding direction.

The training loop, i.e. lines 2 through 6, can be repeated more than once, either a fixed amount of times or until the average error per iteration reaches some predefined threshold. Each iteration of this loop is called a *training epoch*. The *learning rate* $\eta \in [0, 1]$ specifies the volatility of the perceptrons learning process; the higher this rate, the stronger the perceptron updates its weights on new observations.

For linear classification problems, this training algorithm always converges ([Nov63]).

The Multilayer Perceptron / Feedforward Neural Network

A *multilayer perceptron* (MLP) or *feedforward neural network* consists of multiple layers of connected perceptrons, in this context also called *neurons* or *nodes*. The outputs of the first layer of perceptrons act as inputs to the next layer, which in turn supply the data for the following layer etcetera. In contrast to a single perceptron, the MLP is able to classify data that are not linearly separable, e.g. the XOR-function, provided a non-linear activation function is used.

If the perceptron is the connectionist analogy to a single biological neuron, then the multilayer perceptron is the equivalent to multiple linked neurons, passing along, reinforcing or inhibiting their action potentials.

Structure of the Feedforward Neural Network

An MLP consists of a minimum of three layers, an *input layer*, one or multiple *hidden layers* and an *output layer*. This structure is graphically displayed in figure 2.4. Each layer, except for the input layer, consists of one or multiple perceptrons. The input layer just feeds the inputs into the network by distributing each input value to every perceptron in the first hidden layer. In a fully connected MLP, all outputs of the previous layer are taken as inputs into every single perceptron of the next layer, spanning complete bipartite graphs in between layers.

The structural parameters of an MLP are *hyperparameters*, i.e. they can't be learned by the MLP itself and have to be chosen appropriately or otherwise determined. The amount of nodes in the input and output layer however are oftentimes defined by the problem and data at hand. If the (preprocessed) input data consists of vectors in $\mathbb{R}^N$, then N input nodes receive this

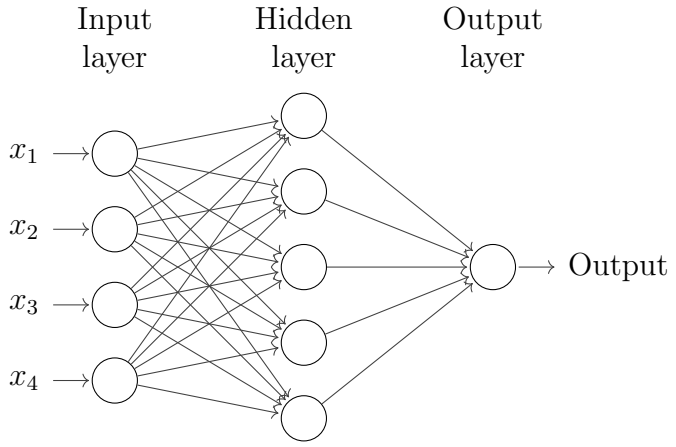


Figure 2.4: A graphical representation of an example multilayer perceptron. The $N = 4$ inputs are fed into the input layer and distributed to the hidden layer as input values to all of the five hidden nodes. All the outputs from the hidden layer are in turn distributed to the output layer as inputs to the single output node. Every node in this MLP consists of a whole perceptron, performing summation of its inputs, adding its bias and applying its activation function separately.

data in the input layer. The size of the output layer is generally determined by the desired output. For binary classification problems, the output layer would consist of a single neuron that either applies a sigmoid activation function[2] σ to produce a probability distribution over the two possible classes or a Heaviside step function to clearly output the presumed class, as a single perceptron does. A multi-class classifier would have one neuron in the output layer for each class and apply a softmax activation function $\sigma(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}}$ to model a probability distribution over all classes.

Forward pass

The process of passing information through the network in the default direction – starting at the input layer, traversing all hidden layers and ending in the output layer – is called a *forward pass* through the network. A single forward pass consumes one data point and outputs a prediction, based on the input data and the networks parameter configuration. It is performed to classify a real data point after the network has been trained, or during training, to produce predictions which are then used to adjust the networks parameters.

In a feedforward neural network, layers may differ in sizes, but in general, they all work the same way. An exception is the input layer, which just passes the input data into the network. As the networks layers are just linked batches of perceptrons, each hidden or output layer acts as follows:

Let $x := \begin{pmatrix} x_1 & \dots & x_I \end{pmatrix}^T$ denote the I input values for the layer at hand (i.e. the output of the previous layer) and $n_1, \dots, n_N$ the N neurons in the layer at hand. Every single of the N neurons in the layer takes all the data from the previous layer as input. Thus, every neuron defines weights for I inputs, resulting in NI different weights for the layer at hand. Let $w_{i,j}$ denote the weight for the connection between neuron n_i and input value x_j. The weight vector for neuron n_i can thus be defined as

$$w_i := \begin{pmatrix} w_{i,1} & w_{i,2} & \dots & w_{i,I} \end{pmatrix}$$

[2] See table 2.2.

and the weighted sum for neuron n_i can be computed as

$$s_i = \sum_{j=1}^{I} w_{i,j} x_j = w_i x.$$

Let W be the weight matrix for this layer, defined as

$$W := \begin{pmatrix} w_1 \\ \vdots \\ w_N \end{pmatrix} = \begin{pmatrix} w_{1,1} & \dots & w_{1,I} \\ \vdots & \ddots & \vdots \\ w_{N,1} & \dots & w_{N,I} \end{pmatrix}.$$

Computing the weighted sums for all neurons $n_1, \dots, n_N$ can now be condensed to a single matrix multiplication

$$s := Wx = \begin{pmatrix} w_{1,1} & \dots & w_{1,I} \\ \vdots & \ddots & \vdots \\ w_{N,1} & \dots & w_{N,I} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_I \end{pmatrix} = \begin{pmatrix} s_1 \\ \vdots \\ s_N \end{pmatrix}.$$

Usually, the activation functions inside a certain layer do not differ, so applying them componentwise can be written as

$$o := \sigma(s) = \begin{pmatrix} \sigma(s_1) \\ \vdots \\ \sigma(s_N) \end{pmatrix},$$

defining the output vector o of this layer.

Because of the separation of the network in L separate layers, the computation inside each layer can be seen as a function $f_l : \mathbb{R}^{N_{l-1}} \to \mathbb{R}^{N_l}, l \in \{1, \dots, L\}$, taking N_{l-1} inputs from the previous layer and outputting N_l values. Each layer function f_l is defined as

$$f_l(x) = \sigma_l(W^l x),$$

for $l \in \{1, \dots, L\}$, activation functions $\sigma_l : \mathbb{R}^{N_l} \to \mathbb{R}^{N_l}$ and weight matrices $W^l \in \mathbb{R}^{N_l \times N_{l-1}}$. The layer function f_1 for the input layer is given as the identity

$$f_1 \equiv \mathrm{id} = \mathrm{id}(I_{N_0} x),$$

by setting $\sigma_1 \equiv \mathrm{id}$ as the identity function and $W^1 = I_{N_0}$ as the identity matrix.

Thus, the whole networks forward pass can be described as

$$o := (f_L \circ \dots \circ f_1)(x) = f_L(\dots f_1(x)).$$

Non-linear Activation Functions

Non-linear activation functions or *Non-linearities* introduce non-linearity in between the linear computations of the MLP layers and thus play an essential part in the MLPs ability to perform non-linear classification. Without non-linear activation functions, the stacked layers of any MLP could be reduced to a simple two-layer model, essentially leaving over only a single or multiple detached perceptrons. This is the case because a single matrix multiplication is sufficient for calculating a linear combination of the input vectors.

As indicated before, not every neuron in the model has to use the same activation function. A few common non-linear activation functions are listed in table 2.2.

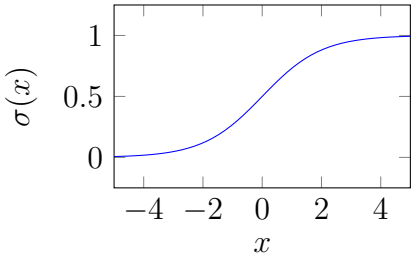
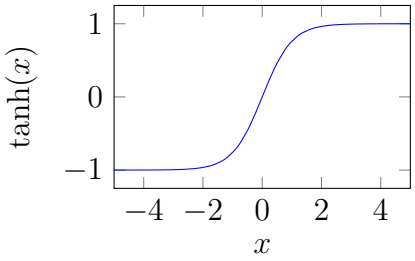
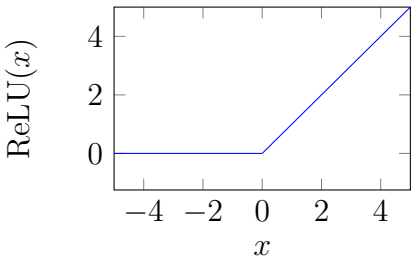
Name	Function	Derivative	Plot
Sigmoid	$\sigma(x) = \frac{1}{1+e^{-x}}$	$\sigma'(x) = \sigma(x)(1-\sigma(x))$	
tanh	$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$\tanh'(x) = 1 - \tanh(x)^2$	
ReLU	$\mathrm{ReLU}(x) = \max(0, x)$	$\mathrm{ReLU}'(x) = \begin{cases} 0 & x < 0, \\ 1 & x > 0. \end{cases}$	

Table 2.2: Non-linear activation functions.

Especially the output layer oftentimes uses a different activation function than the other layers in the network. A popular example is the *softmax function*, which normalizes the output of the network to a probability distribution over the possible output classes. It's defined as

$$\sigma(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}}.$$

2.2.3 Neural Network Training

The training technique for MLPs is similar to the one used for single perceptrons: Labeled training data is used to produce predictions from the MLP in a forward pass and the resulting error, defined by the deviation of this prediction from the actual label, is used to adjust the MLPs parameters.

However, adjusting the weights of all the neurons involved in the calculation is more complex than for a single perceptron. The error has to be propagated back through the whole network of layers in what is called a *backward pass* and each weight has to be updated only insofar as it contributed to the error. Propagating the error back through the network is performed according to the *backpropagation algorithm*. The method to determine how to adjust the weights is called *gradient descent*.

(Stochastic) Gradient Descent

Gradient descent is an iterative method for finding a local minimum of a differentiable multivariate function $F : \mathbb{R}^N \to \mathbb{R}$. The basic idea is that for a given point $x \in \mathbb{R}^N$, in whose

neighborhood F is defined and differentiable, the direction of fastest decrease is given by the negative gradient $-\nabla F(x)$. Iterating

$$x_0 := x$$
$$x_{i+1} := x_i - \epsilon \nabla F(x_i)$$

for sufficiently small $\epsilon \in \mathbb{R}_+$, results in the x_i moving against the gradient direction and thus towards a local minimum of F near x. The initial value x_0 strongly affects which local minimum the sequence converges to.

Stochastic gradient descent (SGD) is a stochastic approximation of gradient descent optimization, i.e. the actual gradient is approximated by not using the whole data set for its computation, but only a smaller, randomly selected subset thereof. This offers a trade-off between a lower convergence rate and faster iterations.

Now, let $\mathcal{L} : \mathbb{R}^N \times \mathbb{R}^N \to \mathbb{R}$ be a function that takes as input a ground truth y, e.g. the label of a data point in the training set, and an estimation $\hat{y}$, e.g. the prediction produced by an MLP. Based on these inputs, it outputs some quantification of the prediction error. Such a function is called a *loss function*. Because the loss function is minimized if and only if the MLPs performance is optimized, SGD can be performed on $\mathcal{L}$ to iteratively minimize the loss value and conversely improve the MLPs prediction performance. Note that when doing SGD on a loss function, one does not alter the input data, but the weights of the MLP; the input data is assumed to be constant. The process can be described as performing SGD on $\mathcal{L}(y, F(x, w))$ with respect to the variable w, where F is a function that evaluates the MLP defined on parameters w for the input x. How exactly each parameter in the MLP has to be altered in order to move against the gradient direction is calculated by the backpropagation algorithm.

Loss Functions

Two commonly used loss functions for neural networks are the *cross-entropy* and *mean squared error* (MSE) losses.

Cross-entropy or *log loss* is used to measure the performance of a neural network outputting probability values between 0 and 1, for example through a softmax output layer. The cross-entropy for N classes is defined as

$$\mathcal{L}_{\mathrm{H}}(y, \hat{y}) := -y \cdot \log \hat{y} = -\sum_{i=1}^{N} y_i \log \hat{y}_i,$$

where y is the label, usually a one-hot representation (having $y_i = 1$ if and only if class i is the correct one, else 0) of the correct classification and $\hat{y}$ is the network output.

The *mean squared error loss* is commonly used for regression tasks, e.g. for predicting continuous variables. It is simply defined as the mean squared error over all network outputs,

$$\mathcal{L}_{\mathrm{MSE}}(y, \hat{y}) := \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2,$$

where again y is the label and $\hat{y}$ is the networks output.

The Backpropagation Algorithm

The *backpropagation algorithm* is used to compute the gradient of a loss function with respect to the weights of an MLP. Stochastic gradient descent in combination with the backpropagation

algorithm is the de facto standard way of training artificial neural networks ([RHW85]). This section is adapted from [Nie15].

As already defined for the forward pass, let

$$\hat{y} := F(x) = (f_L \circ \cdots \circ f_1)(x)$$

denote the output of a feedforward neural network on L layers for input x, where f_l computes the output of layer l as

$$f_l(x) = \sigma_l(W^l x),$$

with activation function σ_l and weight matrix W^l.

The loss for each data point (x, y) in the training data is then defined as

$$\mathcal{L}(y, \hat{y}) = \mathcal{L}(y, F(x)).$$

In order to apply stochastic gradient descent with respect to the parameters specifying the evaluation of $F(x)$, this loss value's gradient for each parameter of the network has to be calculated individually. This is achieved by applying the chain rule to calculate the partial derivatives

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^l}.$$

However, for efficiency reasons, backpropagation avoids duplicate calculation by calculating the gradient of the weighted input for every layer, denoted as δ^l, starting from the last layer L. As each weight in W^l only affects the loss through its effect on the immediate next layer, δ^l contains all the data required for computing the gradient in layer l. Furthermore, the preceding layers $l-1, \ldots, 1$ can be computed recursively.

As before, let $s^l := W^l o^{l-1}$ be the weighted inputs in layer l and $o^l := \sigma_l(s^l)$ be the output of layer l.

Then the derivative of the loss in terms of the input is given as the total derivative

$$\begin{aligned}\frac{d\mathcal{L}}{dx} &= \frac{d\mathcal{L}}{do^L}\frac{do^L}{ds^L}\frac{ds^L}{do^{L-1}}\frac{do^{L-1}}{ds^{L-1}}\cdots\frac{do^1}{ds^1}\frac{ds^1}{dx} \\ &= \frac{d\mathcal{L}}{do^L}\sigma'_L W^L \sigma'_{L-1} W^{L-1} \ldots \sigma'_1 W^1,\end{aligned}$$

because for $l \in \{1, \ldots, L\}$ and $o^0 = x$ one has

$$\frac{do^l}{ds^l} = \frac{d\sigma_l(s^l)}{ds^l} = \sigma'_l \text{ and } \frac{ds^l}{do^{l-1}} = \frac{dW^l o^{l-1}}{do^{l-1}} = W^l.$$

The gradient $\nabla_x \mathcal{L}$ of the loss in terms of the input is then obtained by transposing the derivative of $\mathcal{L}$ in terms of x, as

$$\nabla_x \mathcal{L} = (\frac{d\mathcal{L}}{dx})^T = (W^1)^T \sigma'_1 (W^2)^T \sigma'_2 \ldots (W^L)^T \sigma'_L \nabla_{o^L} \mathcal{L}.$$

Backpropagation evaluates this expression from right to left, propagating the error from layer L all the way back through the network. This is done recursively, by starting with

$$\delta^L = \sigma'_L \nabla_{o^L} \mathcal{L}$$

and recursing

$$\begin{aligned}\delta^l &= \sigma_l'(W^{l+1})^T\delta^{l+1} \\ &= \sigma_l'(W^{l+1})^T \dots (W^L)^T\sigma_L'\nabla_{o^L}\mathcal{L}\end{aligned}$$

for $l \in \{L-1, \dots, 1\}$. The gradient for the weights in layer l is then determined as

$$\nabla_{W^l}\mathcal{L} = \delta^l(o^{l-1})^T.$$

This recursive computation of the gradients per layer is called the backpropagation algorithm.

Adam Optimization

The *Adaptive Moment Estimation* (*Adam*) optimization method ([KB14]) can be used to train a network instead of the classical stochastic gradient descent described above.

While SGD keeps track of a single global learning rate η for all weight updates, which also doesn't change during the training process, the Adam algorithm maintains individual learning rates for each network parameter, that are separately adapted during the training process. These individual learning rates are computed from estimates of the first and second moments of the gradients. More specifically, Adam calculates an exponential moving average of the gradient and the squared gradient.

Adam optimization thus combines the methods introduced in the *Adaptive Gradient Algorithm* (*AdaGrad*, [DHS11]) and *Root Mean Square Propagation* (*RMSProp*, which wasn't published in a formal academic paper, but appeared in the lecture slides of a Coursera online class on neural networks by Geoffrey Hinton from the University of Toronto[3]).

Overfitting and Dropout

Especially for larger neural networks, *overfitting* is a common problem that appears when the amount of the networks parameters are disproportionally large to the complexity of the data ([Die95]). The goal of training a neural network is to have it perform well on data that was not encountered during its training. However, when the network has the capacity to specialize on the training data instead of generalizing from it, overfitting occurs. That is, the network learns to perform well on the training data, but poorly on previously unseen data. A typical symptom of this problem is a shrinking training error, while the validation error goes up.

A commonly used technique to alleviate overfitting is called dropout ([Hin+12]). Overfitting can be greatly reduced by randomly omitting a certain percentage, oftentimes half, of the networks neurons on each training sample. This method forces each neuron to learn a feature that is generally helpful in the whole network as opposed to co-adapting to other neurons. The effects of dropout on an example feedforward network are depicted visually in figure 2.5.

Minibatches and Batch Normalization

When using *minibatches*, the gradients for updating the weights during training are calculated across a smaller subset of the entire data set instead of the entire batch/epoch for gradient descent or a single training example for stochastic gradient descent.

Batch normalization or *batch norm* ([IS15]) is a method of normalizing the input to a networks' layers by recentering and rescaling it for each minibatch in order to stabilize the learning process and reduce the number of training epochs required.

[3] `http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`

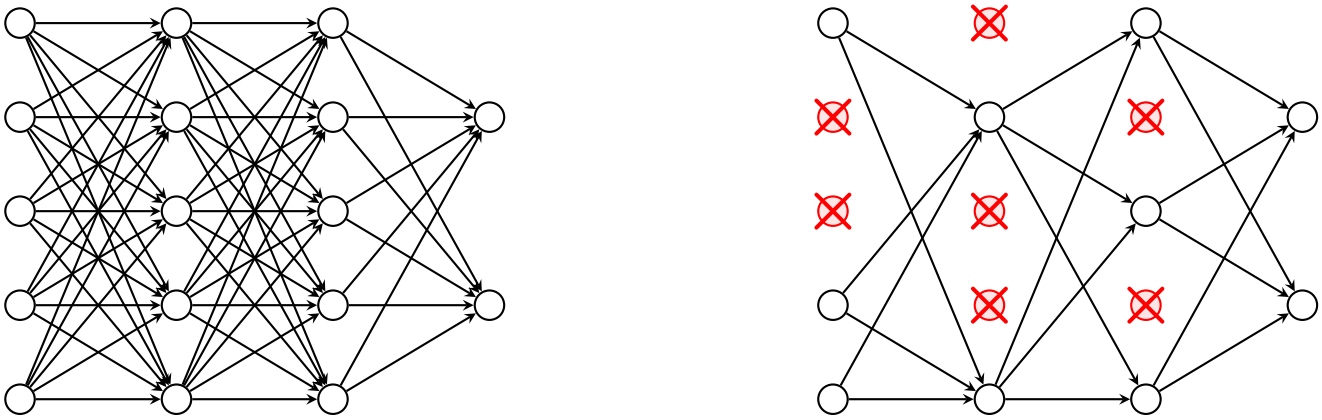


Figure 2.5: This figure shows a possible effect of dropout with $p = 0.5$ on a network with three layers. On the left, the network without dropout – as it would be used during evaluation – is shown, on the right, about half of the networks neurons have been dropped randomly – as might happen during training.

2.2.4 Time Delay Neural Networks

The occurrence of phonemes inside an utterance is not independent from the surrounding context, as speech is a temporal construct. Therefore, when classifying phonemes, it's oftentimes beneficial to not only consider a single audio feature, but a broader range. *Time Delay Neural Networks (TDNNs)* specialize in this ability. In fact, they were first developed for the exact task of phoneme classification ([Wai+89]).

In principle, a TDNN is a feedforward neural network. However, as opposed to receiving one feature vector of speech at a time, it accepts a whole sequence of feature vectors as its input. Furthermore, instead of using individual weights for every single connection, some weights are shared in what is called a *shift invariant connection*, much like a filter in a convolutional neural network ([LeC+99]). Overlapping two-dimensional fields of inputs are weighted and summed together before being passed through an activation function to constitute the data present in the next layer. This can be thought of as a two-dimensional filter moving across the field of input neurons. The weights of this moving filter are shared among all the shifted fields it is applied to.

Therefore, every neuron receives information of a whole range of feature vectors over a certain time frame. It can thus extract temporal information as well as the temporal context of a feature vector to be classified, which enables improved classification results ([PPK15]).

When performing backpropagation on a TDNN, the gradients are calculated for every time-shifted copy of the network and then averaged, so shared weights remain equal among all their instances.

An example of a TDNN is described in more detail in figure 2.6.

2.2.5 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are an example of non-feedforward networks. While feedforward neural networks are characterized by only passing information through their layers in a one-way fashion, RNNs additionally utilize connections that loop back into the same layer. Another difference is that they process sequential input over multiple time steps. In each step, a hidden node in an RNN passes along its value not only to the nodes in the next layer, but

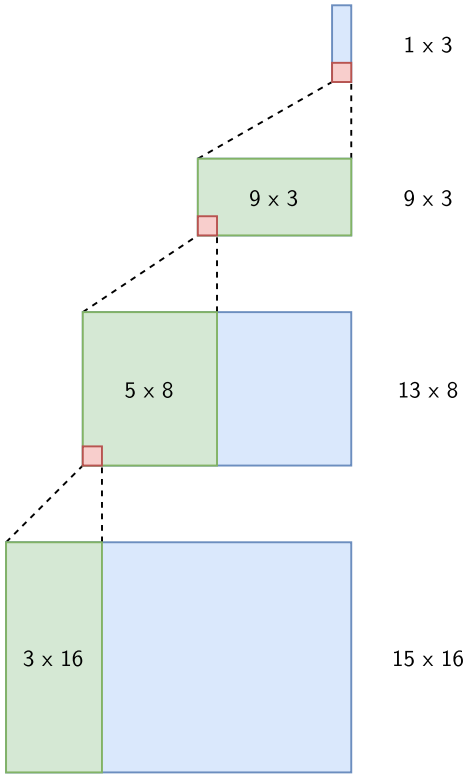


Figure 2.6: An example TDNN for classifying the phonemes */b, d, g/*, adapted from [SSW91]. The network consists of three feedforward layers. Its input layer takes a whole feature vector sequence as input, spanning 15 frames of 16 features each, thus there are 15×16 input neurons. Connections between each layer are shift invariant, i.e. in the first hidden layer, the same 3×16 filter is used for every of the 13 different regions of input it is applied to. The filter maps each region of 3×16 inputs to 8 output values in the first hidden layer. Therefore, there are $3 \times 16 \times 8$ different weights between the input and first hidden layer. The second hidden layer also employs a shift invariant filter for weighting its input values. This time, the filter size is 5×8 and the 13×8 input values are passed on to 9×3 neurons, using only $5 \times 8 \times 3$ different weights. Finally, the output layer applies another filter of size 9×3 to map each filter region on 3 neurons, resulting in another $9 \times 3 \times 3$ weights.

also to itself in the next time step. Conversely, it also takes into account its previous state when processing new input.

An example topology of an RNN is described in figure 2.7.

Similar to TDNNs, RNNs possess an increased capability for handling sequential or temporal data and are thus highly relevant for speech recognition tasks.

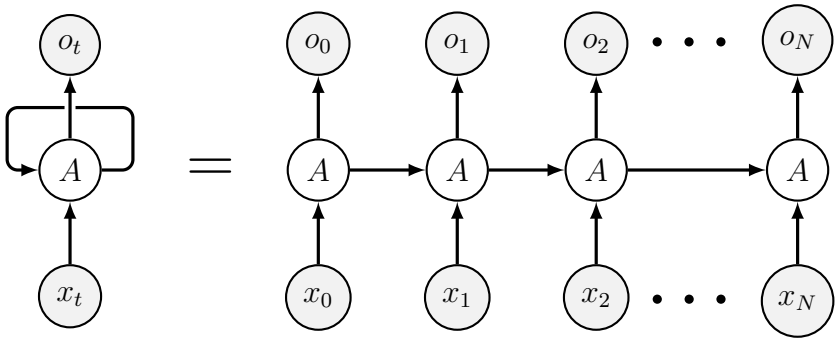


Figure 2.7: An example RNN topology. The inputs $x_t \in \{x_1, \ldots, x_N\}$ are consumed over multiple time steps. In each time step, the hidden layers forward their hidden representations to the next time step and – conversely – process the passed on representation from the previous time step. On the left side, the actual network topology is shown, on the right side it's presented in its unrolled state.

Elman and Jordan networks

Simple examples of RNNs are the *Elman* and *Jordan networks* ([Elm90]). They both consist of three feedforward layers; input, hidden and output.

In the Elman network, every node in the hidden layer has an additional connection to an individual context unit, which saves the hidden neurons output and forwards it to that same neuron in the next time step. Context units have their inbound weights set to constant 1. Their outbound weights are defined in a weight matrix W^u and are learned together with the other parameters of the network. The forward pass in an Elman network is therefore specified by the assignments

$$h_t = \sigma_h(W^h x_t + W^u h_{t-1})$$
$$o_t = \sigma_o(W^o h_t),$$

where h_t describes the output of the hidden layer at time step t and o_t describes the output of the output layer at time step t.

The Jordan network functions similarly to the Elman network, except for the context units taking their input from the output layer instead of the hidden layer. They still forward their saved state to the hidden layer in the next time step:

$$h_t = \sigma_h(W^h x_t + W^u o_{t-1})$$
$$o_t = \sigma_o(W^o h_t).$$

Backpropagation Through Time

The typical training procedure for feedforward networks can be adapted to work for RNNs. Backpropagation for RNNs is called *backpropagation through time (BPTT)*. In order to perform

backpropagation on a recurrent network, the network is first unrolled, as shown in figure 2.7. The unrolled network is now equivalent to a feedforward network with shared weights. Thus, the gradients can then be calculated accordingly, taking into account all the occurrences of a single weight and summing the weight updates.

RNNs are prone to a problem known as *vanishing gradients*. Gradient descent becomes increasingly inefficient for deeper network structures, as gradients are propagated by applying the chain rule, effectively multiplying – and thus further reducing – small numbers from the interval $(0, 1)$. As unrolled RNNs are essentially very deep feedforward neural networks with shared weights, the vanishing gradient problem becomes problematic when processing long input sequences.

This problem arises, because in backpropagation, $h_{t'} = \sigma_h(W^h x_t + W^u h_{t-1})$ is differentiated with respect to h_t for $t' > t$, which results in

$$\begin{aligned}\frac{\partial h_{t'}}{\partial h_t} &= \prod_{k=1}^{t'-t} W^u \sigma_h'(W^u h_{t'-k}) \\ &= \underbrace{(W^u)^{t'-t}}_{(1)} \underbrace{\prod_{k=1}^{t'-t} \sigma_h'(W^u h_{t'-k})}_{(2)} .\end{aligned}$$

If the spectral radius $\rho(W^u)$ of the weight matrix is less than 1, the first factor decays exponentially fast in $t' - t$. Furthermore, if the activation function's derivative is less than 1 for all values – as is the case for the sigmoid function –, the second factor tends to zero as well. The problem in the first factor persists even for alternative activation functions, like ReLU.

The vanishing gradient problem is shown to be severe both theoretically and experimentally for tasks involving long-term dependencies in [BSF94].

2.2.6 Long Short-Term Memory

Long short-term memory (LSTM) is a special recurrent network architecture, developed to mitigate the vanishing gradient problem of traditional RNNs. Like an RNN, an LSTM preserves its state for use during later time steps. However, in LSTMs this is done in a more fine-tuned way. In order to achieve this, an LSTM consists of gated neurons, called *LSTM cells*. These cells not only save their last state and supply it as part of the input to the next time step, but also modulate this saved state in more complex ways.

The LSTM cell

The structure of an LSTM cell is depicted in figure 2.8. Its central component is the internal state c_t that gets passed along through time. In each time step, old information can be forgotten and new information can be added to this internal state. This forgetting/updating process is handled by the cell's forget and update gates, whose behavior is learned during training. The updated state is then used to modulate the cells output in the output gate.

The cell's forget gate is responsible for forgetting selected parts of the current cell state and it's the first gate to be applied on the incoming previous cell state. It takes as input the last hidden state h_{t-1} and the current input vector x_t, weighting, concatenating and processing them in a sigmoid layer to produce the forget value f_t. By multiplying this forget value $f_t \in \sigma(\mathbb{R}^N) =$

$(0,1)^N$ pointwise with the incoming previous cell state c_{t-1}, the i-th entry in the cell state can either be kept intact ($(f_t)_i \approx 1$) or forgotten ($(f_t)_i \approx 0$):

$$f_t = \sigma_g(W^f x_t + U^f h_{t-1}).$$

Next, the new information to be stored in the state vector is determined and applied. This is done in the input/update gate. Another sigmoid layer on the weighted concatenation of h_{t-1} and x_t is used to determine which parts of the cell's state are to be updated. A tanh layer on the same input produces the candidate values $\tilde{c}_t$ to be inserted into the state. Their pointwise product constitutes the update value, which is then added to the cell state vector:

$$\begin{aligned} i_t &= \sigma_g(W^i x_t + U^i h_{t-1}) \\ \tilde{c}_t &= \tanh(W^c x_t + U^c h_{t-1}). \end{aligned}$$

This process of first selectively forgetting and then storing new data in the cell state constitutes the whole update process of the state vector:

$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t.$$

Finally, the cell produces an output vector as well. An unfiltered output vector o_t is first computed by feeding the weighted concatenation of h_{t-1} and x_t into a sigmoid layer. As the cell state should have an effect on the output as well, the state is fed into a pointwise tanh function and then multiplied with the unfiltered output to produce the hidden state vector h_t:

$$\begin{aligned} o_t &= \sigma_g(W^o x_t + U^o h_{t-1}) \\ h_t &= o_t \circ \tanh(c_t), \end{aligned}$$

The computations inside an LSTM cell are thus given by the equations

$$\begin{aligned} f_t &= \sigma_g(W^f x_t + U^f h_{t-1}) \\ i_t &= \sigma_g(W^i x_t + U^i h_{t-1}) \\ o_t &= \sigma_g(W^o x_t + U^o h_{t-1}) \\ \tilde{c}_t &= \tanh(W^c x_t + U^c h_{t-1}) \\ c_t &= f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \\ h_t &= o_t \circ \tanh(c_t), \end{aligned}$$

where $\circ$ is the Hadamard (element-wise) product, f_t, i_t and o_t denote the activation vectors of the forget gate, input/update gate and output gate, h_t is the hidden state vector of the cell, $\tilde{c}_t$ is the cells update candidate vector and c_t is the cells state vector; all at time t. The matrices W^x, U^x contain the weights for the respective gates $x \in \{f, i, o, c\}$. Besides the input vector x_t, a cell consumes the vectors h_{t-1} and c_{t-1} from the previous time step and produces a new pair of c_t and h_t vectors.

LSTM Training

Similarly to the training process of RNNs, LSTMs can be trained using backpropagation through time. While RNNs are prone to the vanishing and exploding gradients problem, LSTMs mitigate the vanishing gradients problem by their use of gated neurons. Also, the cell's state is updated through addition instead of multiplication, which further combats the problem.

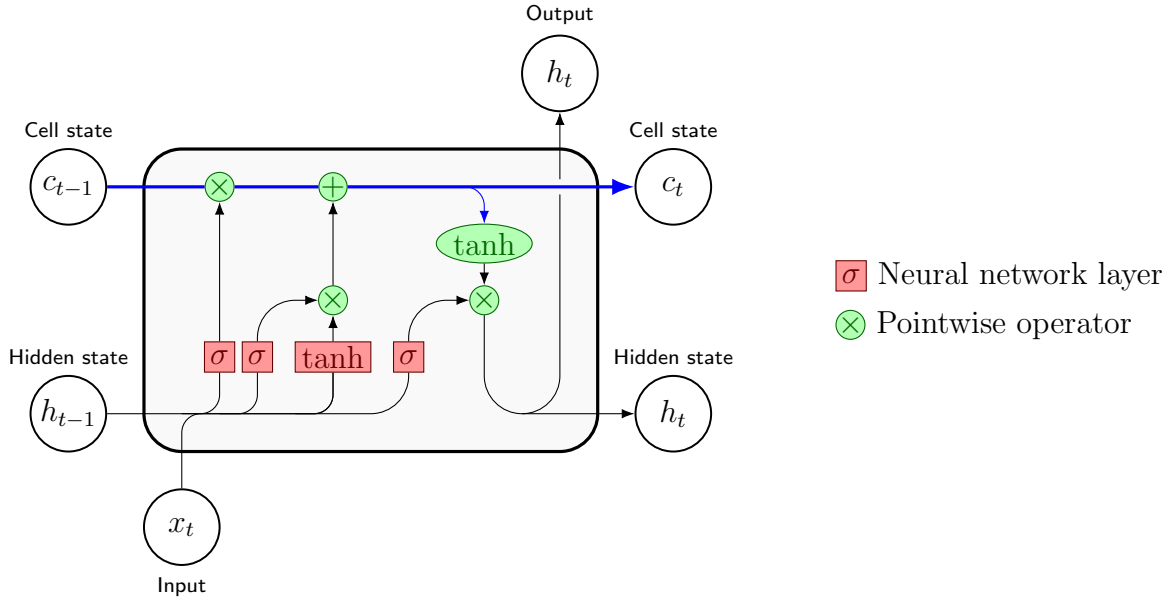


Figure 2.8: The inside structure of an LSTM cell. Joining arrows represent the concatenation of values, diverging arrows represent the copying of values. Each layer, marked red in the graphic, applies a weight matrix and the denoted activation function to its inputs. Every operator, marked green, only applies the denoted function to each of the input values element-wise. The blue arrow shows how the cell state runs through the whole construct, first being modified by the inputs and then itself influencing the produced output.

An LSTM cell saves long term dependencies in its state $c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t$. The gradient of state c_t with respect to its previous state c_{t-1} is given as

$$\begin{aligned}\frac{\partial c_t}{\partial c_{t-1}} &= \frac{\partial f_t \circ c_{t-1}}{\partial c_{t-1}} + \frac{\partial i_t \circ \tilde{c}_t}{\partial c_{t-1}} \\ &= f_t + \frac{\partial i_t \circ \tilde{c}_t}{\partial c_{t-1}}.\end{aligned}$$

Assuming the worst case scenario of $\frac{\partial i_t \circ \tilde{c}_t}{\partial c_{t-1}}$ vanishing, leaves the following gradient for cell state $c_{t'}$ with regard to state c_t, $t' > t$:

$$\frac{\partial c_{t'}}{\partial c_t} = \prod_{k=1}^{t'-t} f_k.$$

Now, the f_k's diminish the gradient, but only by the same amount as they determined how much information got passed from their respective states in the forward pass. Thus, the gradient vanishes only insofar as the influence of the previous states vanishes.

However, the exploding gradient problem still persists. This can be mitigated by clipping the gradient's magnitude at a threshold.

Bidirectional LSTM

LSTMs successfully exploit temporal dependencies in the data in one direction. Oftentimes it's helpful to rely on such dependencies in the input in the backward direction as well. For this purpose, *bidirectional LSTMs (BiLSTMs)* have been developed.

A BiLSTM consists of two LSTMs, which receive the same input sequence but pass their cell states in opposite directions. Their outputs are then combined and form the output of the BiLSTM network.

The structure of a BiLSTM network is outlined in figure 2.9.

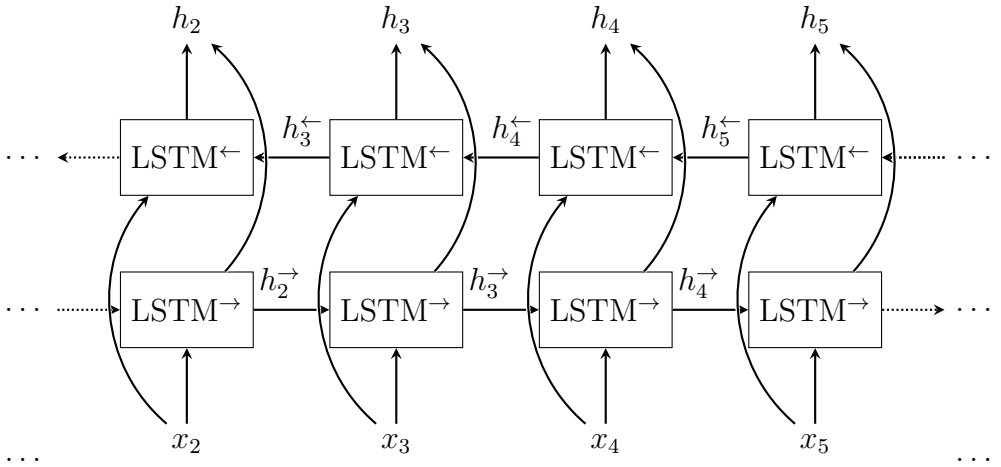


Figure 2.9: Architecture of a BiLSTM network. The network consists of two unidirectional LSTM networks, sharing the same input and producing a combined output by processing the context in the two directions of time independently.

Stacked (Bi-)LSTM

A stacked (Bi-)LSTM consists of multiple layers of (Bi-)LSTMs. Each LSTM cell in a subsequent layer takes as input the output of the previous LSTM cell, thus allowing a more complex processing of the sequence data. An example architecture of a stacked BiLSTM on two layers is presented in figure 2.10.

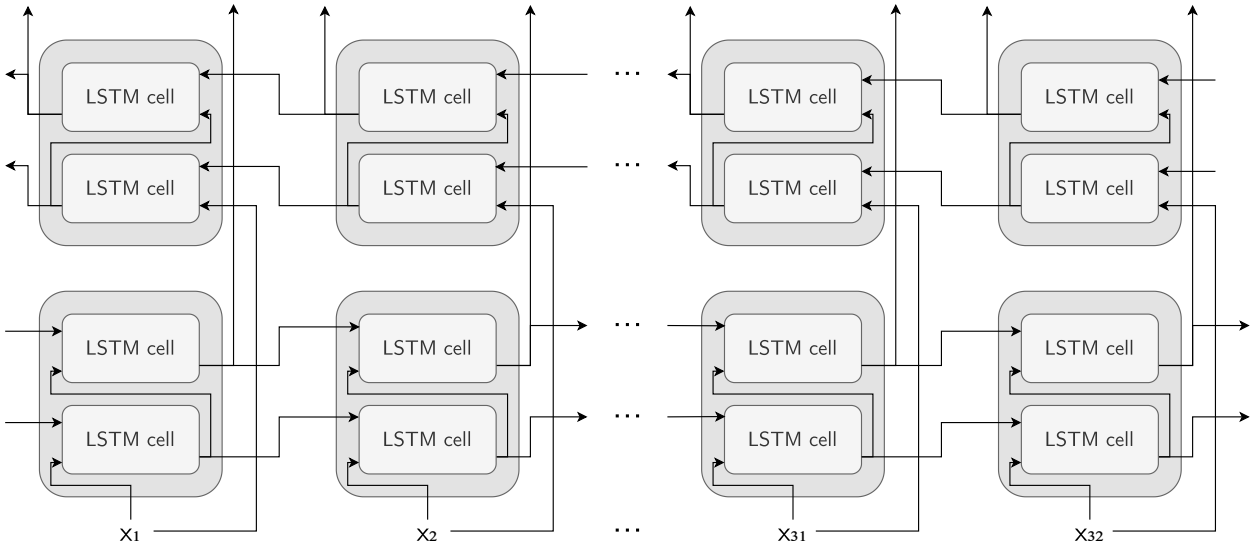


Figure 2.10: Architecture of a stacked BiLSTM network on two layers. The network is built similarly to a BiLSTM network, but two stacked LSTM cells substitute each LSTM cell.

3 Related Work

There are various topics that are related to the task and methodology this thesis is concerned with. Examples range from directly related topics like multilingual phoneme recognition and alignment to the application of cross-lingual methods in the field of ASR in general. Alternative approaches, like end-to-end systems, i.e. systems that are represented by a single model as opposed to a hybrid HMM/ANN pipeline, are presented as well. Furthermore, there are interesting methods that have been applied in the past to improve the performance of multilingual systems, like modulation techniques.

Phoneme Classification

For the task of framewise phoneme classification – the task fulfilled by the ANNs utilized in this thesis as well – [GS05] compares BiLSTMs to other ANN architectures. In the conducted experiments, BiLSTMs performed significantly better than unidirectional ones. LSTMs were also not only much faster to train than standard RNNs and feedforward networks, but also slightly more accurate. It was shown that BiLSTM offer a great way to exploit the time-dependencies in speech.

Multilingual Phoneme / Phone Classification

By supplementing the language-independent phone distributions that are normally used in multilingual acoustic modeling with language-dependent phoneme distributions, [Li+20] was able to improve performance by 2% phoneme error rate absolute. They were also able to improve the phone recognition accuracy by 17% for unseen languages.

(Cross-lingual) Phoneme Boundary Detection

A different approach for detecting the boundaries of phonemes in speech recordings was presented in [Fra+16]. Using a BiLSTM, they were able to outperform the alternatives using phoneme recognizers that were previously reported in the literature on the TIMIT data set. Their experiments also showed promising results regarding cross-lingual tasks.

Modulation Techniques for Multilingual Recognition Tasks

In [MSW18], language adaptation techniques, i.e. modulating the hidden layers of the utilized RNNs using Language Feature Vectors (LFVs), are introduced in order to decrease the error rates in multilingual phoneme / grapheme recognition tasks. These LFVs are obtained from a bottleneck layer in an additional network trained for language identification. Modulating the layers by LFVs, instead of only appending them, showed improved results.
This concept is further elaborated on and extended by Multiplicative Language Codes and Adaptive Neural Language Codes in the related PhD thesis [Mül18].

Cross-lingual Word-to-phoneme Alignment

In [Sta+12], cross-lingual word-to-phoneme alignment was used to derive a word segmentation. The applied methods outperformed state-of-the-art monolingual word segmentation approaches for an alignment of English words to Spanish phonemes.

End-to-end Approaches in ASR

Experiments have shown that it is not always necessary to rely on a hidden Markov model or even an explicit phonetic representation to achieve good results in the field of ASR in general:

Especially BiLSTMs have recently been used in end-to-end systems, e.g. in [GJ14], where only minimal preprocessing and no explicit phonetic representation or prior linguistic information were sufficient to achieve a word error rate (WER) of 27.3%. By supplementing a lexicon of allowed words, the WER could be reduced to 21.9% and using a trigram language model further reduced it to 8.2%.

[Han+14] also showed that BiLSTMs are suitable for performing end-to-end ASR but although they showed promising results, they did not outperform HMM-based systems. The experiments also showed that recurrent connections, especially bidirectional ones, e.g. inside (Bi-)LSTMs, are critical for good performance in the task of speech recognition.

4 Main Contributions

4.1 Hybrid HMM/ANN System

The task at hand is to align a given audio recording of speech with the corresponding phonemes, extracted from a given orthographic transcript. This orthographic transcript can be mapped to a phonetic transcript using a pronunciation dictionary, which maps words to sequences of phonemes that constitute the pronunciation of that given word. This phonetic transcript provides the topology for an HMM whose states correspond to phonemes and whose emitted symbols correspond to feature vectors of speech.

Different phones sound differently and so their typical feature vectors can in principle be distinguished. This distinction is a typical classification task and so it's appropriate to experiment with different connectionist models in order to solve it. Thus, various cross-lingual artificial neural networks will estimate posterior phoneme probabilities for given feature vectors. By calculating the prior probabilities for all phonemes, and applying Bayes theorem, one can obtain posterior probabilities for emitting a feature vector in a given phoneme state of the HMM:

$$P(F \mid Q) = \frac{P(Q \mid F)P(F)}{P(Q)},$$

where $P(F)$ is the probability to see feature vector F in the audio data, which is assumed to be a constant, $P(Q)$ is the prior probability of (sub-)phoneme/HMM-state Q, $P(Q \mid F)$ is the probability of feature vector F belonging to (sub-)phoneme Q, i.e. the classification result of the ANN, and $P(F \mid Q)$ is the probability of emitting feature vector F in HMM-state Q, i.e. the required probability for performing the Viterbi algorithm. These probabilities, together with the assumption of uniformly distributed state transition and initial probabilities, are sufficient for building the HMM.

Solving the Decoding problem for the described HMM and the series of feature vectors that were obtained from the audio recording gives the most probable state sequence responsible for emitting this sequence of feature vectors. This state sequence corresponds to a sequence of phonemes over time, which in turn yields an alignment of phonemes and audio recording over time.

This process is also presented visually in figure 4.1.

4.2 Experimental Pipeline

There are several steps involved in running the HMM/ANN pipeline for a single experiment.

4.2.1 Preparation and Preprocessing

Preparation of the Data

In order to perform the multilingual approach outlined above, data sets from various languages of the Common Voice project (see 4.3.3) have been united and preprocessed.

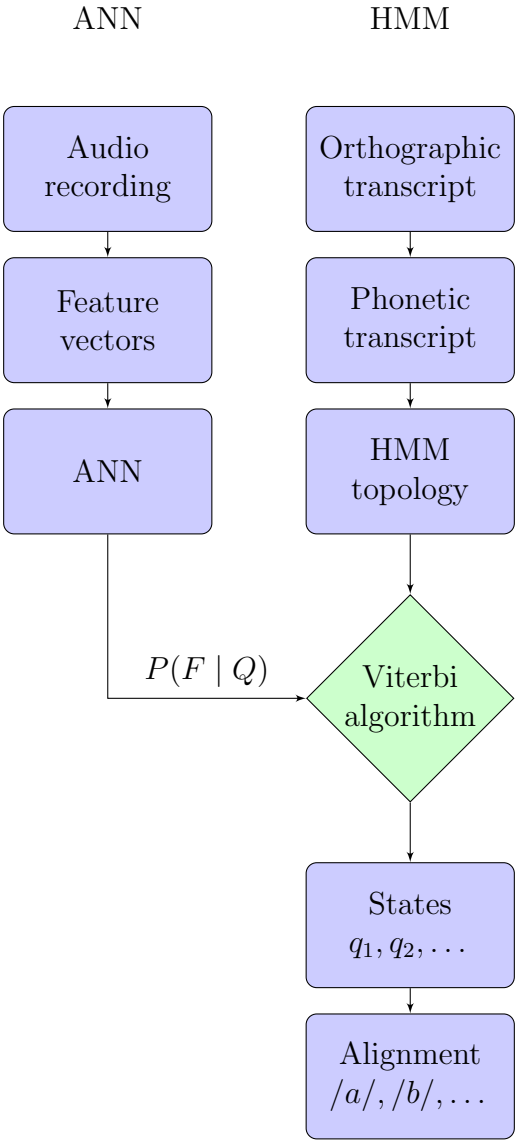


Figure 4.1: Overview of the utilized hybrid HMM/ANN system. The input is given as pairs of audio recordings and their corresponding orthographic transcript. The left path of the diagram shows the processing of the audio recording in a neural network pipeline in order to obtain the probability $P(F \mid Q)$ of feature F being emitted in HMM state Q. In the right path, the phonetic transcript is obtained by looking up the words of the given orthographic transcript in a pronunciation dictionary. Based on this phonetic transcript, the topology of an HMM is built and used to perform the Viterbi algorithm, using the provided posterior probability from the left path. The produced most probable sequence of states is then used to obtain an alignment.

For the multilingual data set, 32,000 utterances of five languages have been combined to form a data set of 160,000 utterances, or 207 hours of speech recordings.

The chosen *known* languages constituting this data set are German, Russian, French, Spanish, and Swedish.

English was chosen as *unknown* language, or target language of the cross-lingual approach. The English data set is used for evaluation and consists of 32,0000 utterances, or 50 hours in total.

Preprocessing of the Audio Signals

All the audio recordings are preprocessed in Janus as follows:

First, the audio signal is mapped from the time domain to the frequency domain by performing a fast Fourier transform ([Coc+67]) on 257 points at a sampling rate of 16kHz.

Secondly, a Mel scale filter bank matrix, an array of 40 bandpass filters for the Mel scale, is created. The Mel scale was invented to mimic the non-linear perception of sound in the human ear ([SVN37]). Mel filter banks give a higher resolution at low frequencies and a lower resolution at high frequencies, as the human ear is less discriminative at higher frequencies as well. This matrix is then applied to each sampled frame by matrix multiplication.

Afterwards, Log-Mel features are calculated by taking the logarithm as $\log(\text{Mel} + 1)$.

Finally, mean subtraction and normalization are performed to normalize twice the standard deviation to the value 1.

4.2.2 Bootstrapping a Multilingual Acoustic Model

A multilingual acoustic model, i.e. the output of the ANN part in the outlined system, can be bootstrapped by basing it on a monolingual acoustic model and iteratively improving it from there.

As a preparation for the bootstrapping process, the pronunciation dictionaries of all the languages that are to be part of the multilingual system have to be mapped to only use the phonemes present in the phoneme inventory of the language of the monolingual system.

In the first iteration, the monolingual acoustic model together with the phoneme mapping can be used to roughly align the multilingual data set. The thusly labeled data set can now be used to create a first multilingual acoustic model.

The previously obtained feature vectors and corresponding labels of the training data set are then used to train the respective neural network of the current experiment. After training the network for multiple epochs, the whole training data set is evaluated in order to produce a new acoustic model, i.e. the probabilities for seeing the given feature vectors, given a state of the HMM. As the neural network is a classifier for different HMM states (subphonemes), given a feature vector of speech, the obtained probabilities are actually just the posterior probabilities of the HMM states given a feature vector, but can be transformed into the required probabilities by separately calculating the prior probabilities for all states and applying Bayes theorem.

The HMMs work with scores instead of raw probabilities, so the posterior emission probabilities are transformed to scores as follows:

$$\text{score} = -8 \log(\text{prob}).$$

These produced scores for the different utterances in the training data set are written to disk as Janus-readable matrices and can thus be loaded for the next iteration, where the previously existing system for labeling is replaced with the new acoustic model.

This process is iterated, in order to achieve more accurate acoustic models in each subsequent pass.

4.2.3 Evaluation

Finally, the multilingual acoustic model is used in the neural network to evaluate – i.e. obtain scores for – the evaluation data set. These scores are then used in a final labeling process to obtain the phoneme alignment of the evaluation data set.

At last, the obtained alignments are compared to the ground truth by the scoring processes described in 5.1.

4.3 Toolkits, Libraries and Data sets

4.3.1 Janus Speech Recognition Toolkit

All tasks regarding the HMM utilized in this thesis and the preprocessing were carried out with the Janus Speech Recognition Toolkit (JRTk; [Fin+97]) developed at the Karlsruhe Institute of Technology and Carnegie Mellon University. The JRTk is implemented in the C programming language, but grants high flexibility through its object-oriented programming interface in the Tcl programming language [Ous+89].

The JRTk was used not only for performing the final Viterbi alignment, but for iteratively labeling the data set in the bootstrapping process and for preprocessing as well.

4.3.2 PyTorch

PyTorch ([Pas+19]) is an open source machine learning library based on the Torch library ([CBM02]). Its main features are the support of GPU acceleration for tensor computing and deep neural networks built on a type-based automatic differentiation system.

PyTorch was used for training and evaluating the neural networks utilized in this thesis.

4.3.3 Common Voice

All the used raw audio recordings and their respective orthographic transcript stem from the Common Voice project ([Ard+19]).

The Common Voice data set is a multilingual collection of transcribed speech, intended for speech technology research and development. The project heavily relies on crowdsourcing for data collection and validation.

In this thesis, data from the languages English, German, Russian, French, Spanish and Swedish is used.

4.4 Experiments

As the whole HMM/ANN pipeline as described above is consistent across all experiments, this section is limited to describe the differences between the neural networks only. The networks are trained and utilized for phoneme classification, thus providing the HMMs with emission probabilities.

Although their architectures differ heavily, there are some commonalities most of them share: All networks are trained on the task of phoneme classification by receiving preprocessed audio frames as input and the corresponding phoneme label as one-hot encoded output. As they all output a probability distribution via a softmax activation function in the last layer, cross entropy loss is applied for optimization.
A minibatch size of 1024 and a split of 90/10 into training and validation set have been used during the training across all experiments. The training was usually performed for eight epochs in both iterations and the pretrained network states from the first iteration were used as initial states for the second iteration, as that was observed to lead to better results. During the training of the neural networks, the data sets were shuffled.

Each networks validation accuracies during training are presented in a separate table. Table 5.1 compares all networks cross-lingual phoneme classification accuracies on data of the target language.

4.4.1 Monolingual Feedforward Neural Network

The naïve approach, or baseline experiment, consists of a monolingual feedforward neural network. This approach, albeit simple, is already cross-lingual, because a German system is used to align an English data set. In order to make the system align English data, all the phonemes in the English pronunciation dictionary have to be mapped to their German counterparts first. The English orthographic transcripts can then be mapped to German phonetic transcripts, which in turn can be aligned using the German system.

Architecture

The neural network used for German phoneme classification consists of an input layer of 600 neurons, which receive a context of 15 feature vectors of size 40 each, followed by five hidden layers of 2,000 neurons each, one bottleneck layer of 1,000 neurons and an output layer of 16,130 neurons. The neurons in the output layer represent the probability distribution over 16,130 subphonemes that is output by the network for each classification sample. Each layer, except the last one, uses the ReLU activation function. The output layer applies a softmax activation in order to create the aforementioned probability distribution.

The networks architecture is visualized in figure 4.2.

Training

The monolingual network was trained with SGD using a learning rate progression of $\eta = 0.08$ for four epochs, then halving it every subsequent epoch. Its final validation accuracy was 48.1% after 13 training epochs. All the learning rates and validation accuracies during training are listed in table 4.1.

4.4.2 Multilingual Feedforward Neural Network

Architecture

The multilingual feedforward network receives a context of eleven frames in total – five previous to the one to be classified and five subsequent ones. These $11 \times 40 = 440$ input values are concatenated and passed into the networks input layer.

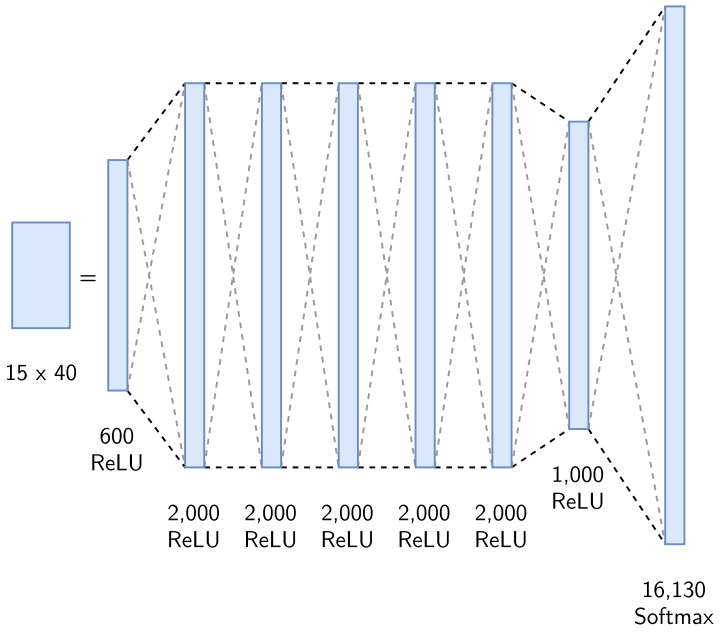


Figure 4.2: Monolingual feedforward neural network on eight layers with 600, five times 2,000, 1,000 and 16,130 neurons.
The applied activation functions are written below each layer. No dropout was applied during training.

Monolingual Feedforward Neural Network		
Epoch	Learning Rate η	Validation Accuracy
1	0.08	38.5%
2	0.08	41.4%
3	0.08	42.4%
4	0.08	42.7%
5	0.04	44.3%
6	0.02	45.5%
7	0.01	46.4%
8	0.005	47.0%
9	0.0025	47.5%
10	0.00125	47.7%
11	0.000625	47.9%
12	0.000313	48.1%
13	0.000156	48.1%

Table 4.1: Learning rates and validation accuracies of the monolingual feedforward neural network.

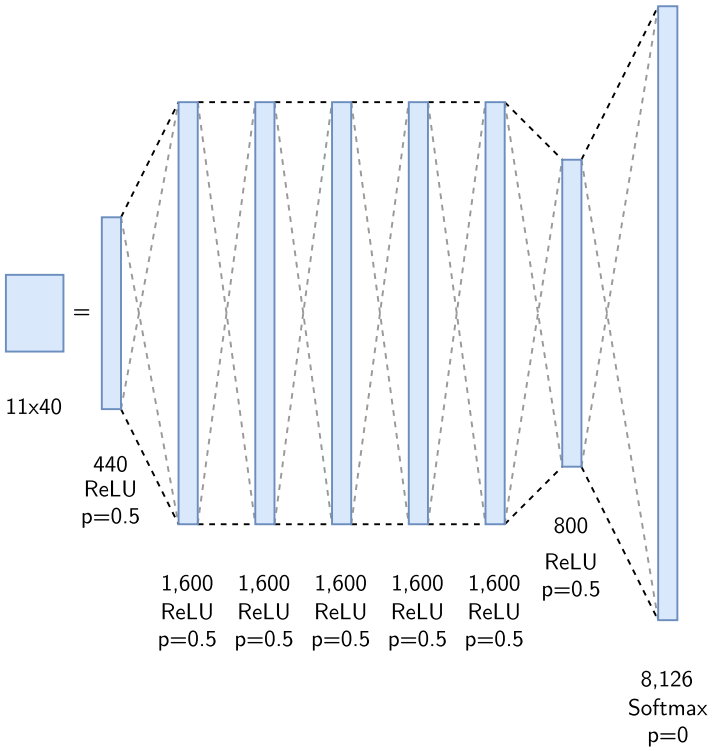


Figure 4.3: Multilingual feedforward neural network on eight layers with 440, five times 1,600, 800 and 8,126 neurons each.
The activation functions and dropout probabilities are written below each layer.

The network architecture is defined by eight layers of 440, five times 1,600, 800 and 8,126 neurons each. The input layer receives eleven feature vectors of length 40. Dropout with a probability of $p = 0.5$ is applied after each layer, except for the last one. Every layer uses a ReLU activation function, except for the last one, which uses a softmax activation in order to output the desired probability distribution across 8,126 subphonemes.

The whole architecture is visualized in figure 4.3.

Training

Instead of conventional SGD, the Adam optimizer with an initial learning rate of $\eta = 10^{-4}$ was utilized. Training was performed for 8 epochs in both iterations of the bootstrapping process, with a final validation accuracy of 51.1% in the first iteration and 48.4% in the second one. All validation accuracies are listed in table 4.2.

4.4.3 Multilingual Time Delay Neural Network

Architecture

The time delay neural network receives a context of 25 frames per classification. However, these $25 \times 40 = 1,000$ inputs are not stacked like they were for the feedforward networks, but convolved with a sliding 40×8 filter with a stride and dilation of 1. This defines the dimensions of the next layer as 80×18, where another filter of size 80×8 is applied. The filters of the next three time delay layers – with sizes $80 \times 5, 160 \times 5$, 160×3 and 800×1 – further shrink the dimensions to 800×1. These resulting 800 neurons are then fed into a final feedforward layer to output 8,126 values.

Multilingual Feed Forward Neural Network		
Epoch	Iteration 1	Iteration 2
1	45.8%	42.3%
2	49.6%	47.8%
3	50.6%	48.2%
4	51.1%	48.4%
5	51.1%	48.4%
6	51.2%	48.4%
7	51.2%	48.4%
8	51.1%	48.4%

Table 4.2: Validation accuracies of the multilingual feedforward neural network, across both iterations of the bootstrapping process.

Multilingual TDNN		
Epoch	Iteration 1	Iteration 2
1	46.7%	40.7%
2	52.1%	46.0%
3	52.7%	46.8%
4	53.1%	47.2%
5	53.3%	47.2%
6	53.3%	47.6%
7	53.4%	–
8	53.4%	–

Table 4.3: Validation accuracies of the multilingual time delay neural network, across both iterations of the bootstrapping process.

Dropout with a probability of $p = 0.5$ is applied after each layer, except for the last one. Every layer uses a ReLU activation function, except for the last one, which uses a softmax activation in order to output the desired probability distribution across 8,126 subphonemes.

Each of the time delay layers also applies batch normalization.

The whole architecture is visualized in figure 4.4.

Training

Again, the Adam optimizer was used during training, this time with an initial learning rate of $\eta = 10^{-3}$. Training was performed for 8 epochs in the first and 6 epochs in the second iteration of the bootstrapping process, with a final validation accuracy of 53.4% in the first iteration and 47.6% in the second one. All validation accuracies are listed in table 4.3.

4.4.4 Multilingual Stacked Bidirectional Long Short-Term Memory

Architecture

The multilingual stacked BiLSTM receives a total context of 81 frames per classification – 40 previous and 40 subsequent ones to the frame to be classified. These $81 \times 40 = 3,240$ inputs are neither stacked like for the feedforward networks, nor convolved with a sliding filter like in

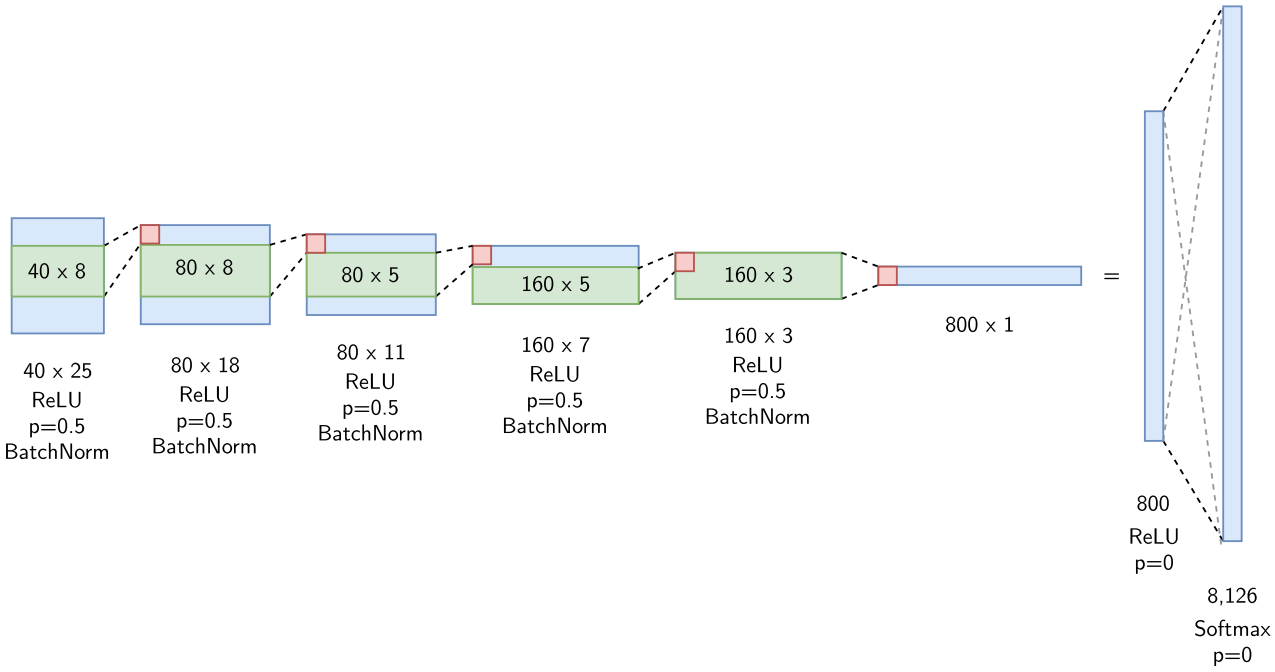


Figure 4.4: Multilingual time delay neural network on six layers, consisting of five convolutional time delay layers with dimensions 80×18, 80×11, 160×7, 160×3 and 800×1, followed by a feedforward output layer of size 8,126. The filter sizes of the first five layers are 40×8, 80×8, 80×5, 160×5 and 160×3.
The activation functions, dropout probabilities and applications of batch normalization are written below each layer.

the time delay neural networks. Instead, they are fed into the stacked BiLSTM as a sequence of inputs over time in both time dimensions.

The stacked BiLSTM uses hidden representations of size 20 and comprises two layers of BiLSTMs. The output of the stacked BiLSTM, hence of dimension $2 \times 81 \times 20 = 3,240$, is then concatenated and passed through a ReLU activation function and into a new layer of size 1,600, again with a ReLU activation and dropout with probability $p = 0.5$. The final layer is a softmax layer of size 8,126 again.

The whole architecture is visualized in figure 4.5.

Training

The Adam optimizer with an initial learning rate of $\eta = 10^{-4}$ was utilized during the training processes of the stacked BiLSTM. Training was performed for 8 epochs in each iteration of the bootstrapping process, with a final validation accuracy of 53.0% in the first iteration and 47.8% in the second one. All validation accuracies are listed in table 4.4.

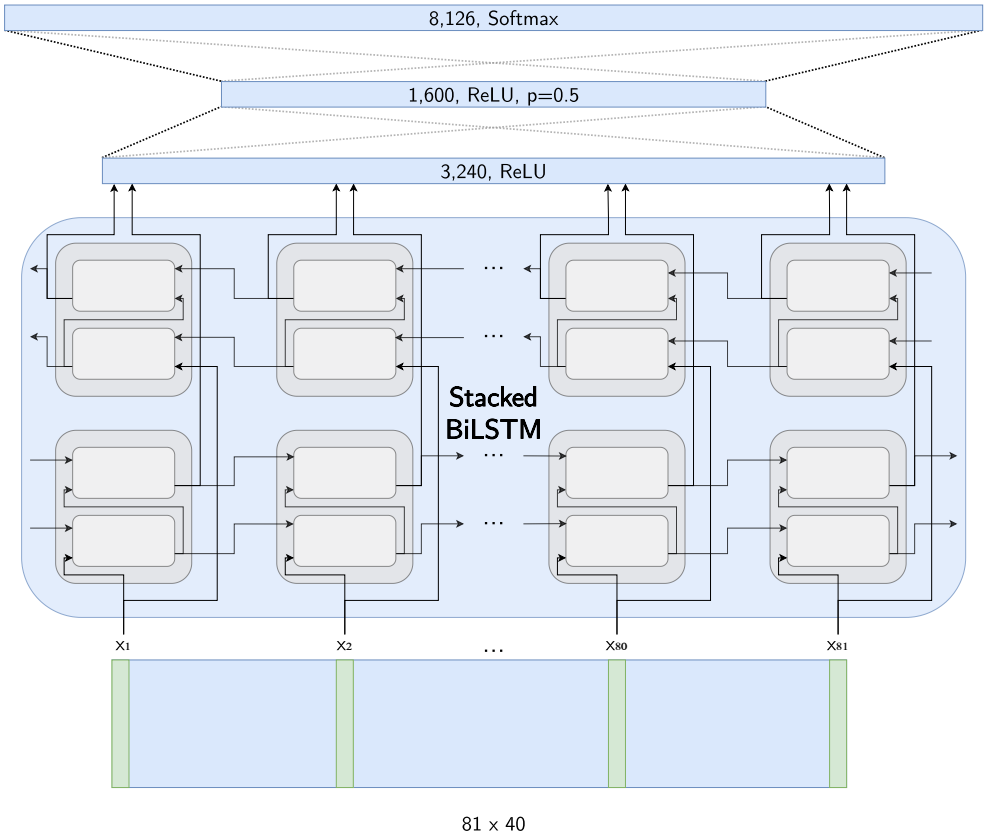


Figure 4.5: Multilingual stacked bidirectional LSTM on two internal layers and hidden representations of size 20.
The input is fed into the stacked BiLSTM first, then processed in two feedforward layers of sizes 1,600 and 8,126. The activation functions and dropout probabilities (if applied) are denoted inside the layers.

Multilingual Stacked BiLSTM		
Epoch	Iteration 1	Iteration 2
1	49.7%	44.6%
2	51.1%	46.0%
3	51.8%	46.6%
4	52.2%	46.9%
5	52.5%	47.3%
6	52.7%	47.5%
7	52.9%	47.5%
8	53.0%	47.8%

Table 4.4: Validation accuracies of the multilingual stacked BiLSTM neural network, across both iterations of the bootstrapping process.

5 Evaluation

5.1 Scoring Methods

In order to evaluate different alignment methods, a scoring method is necessary. There are several possible methods which can be applied.

Mean Squared Error (MSE) Score One of the most popular scoring methods in general, the *Mean Squared Error (MSE) Score* is calculated by taking the average of the squares of the errors of a given alignment. Here, the errors are defined as the deviations of predicted phoneme boundaries from a given ground truth alignment. More formally, letting Y_i be the point in time of transitioning from phoneme $i-1$ to phoneme i in the ground truth alignment and $\hat{Y}_i$ be the predicted point in time, the MSE Score is given as

$$\frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2.$$

Box Score The *Box Score* is inspired by the scoring method used in [Fra+16]. This scoring method counts the errors in the predicted phoneme boundaries, normalized by the total amount of phonemes in an alignment. Here, an error is a binary indicator of whether a given boundary was predicted correctly or not. However, a small error tolerance of 20 milliseconds in both directions is granted in order to exclude near misses from the errors. The score is then given as the amount of correctly (within the tolerance) guessed boundaries, divided by the total amount of phonemes in the alignment.

Overlap Score Another way of scoring a predicted alignment against a given ground truth is to calculate the phoneme overlap between the two. The resulting *Overlap Score* is given as the total time of matching phonemes divided by the total temporal length of the alignment.

5.2 Results

All experiments have been evaluated once after each performed iteration of the bootstrapping process. The outlined scoring methods were applied on the alignments and the resulting scores are presented in this section.
The different scoring results are compared more compactly in tables 5.2, 5.3 and 5.4. Table 5.1 also compares all networks' cross-lingual phoneme classification accuracies for reference.

5.2.1 Monolingual Feedforward Neural Network

During evaluation, the monolingual system achieved a total MSE score of $\bar{s}^{\text{MSE}} = 0.1161$, with a standard deviation of $\sigma^{\text{MSE}} = 6.3992$. The median of all MSE scores was $\tilde{s}^{\text{MSE}} = 0.0028$ and the trimmed mean (onesided and by 10%, to exclude outliers) was $\bar{s}^{\text{MSE}}_{0.1} = 0.0042$.

When scored with the box scoring method, the monolingual system achieved a mean box score of $\bar{s}^{\text{box}} = 0.4303$, with a standard deviation of $\sigma^{\text{box}} = 0.1335$. The median of all box scores was $\tilde{s}^{\text{box}} = 0.44$ and the trimmed mean was $\bar{s}^{\text{box}}_{0.1} = 0.4588$.

Finally, with the overlap scoring method, the system achieved a mean overlap score of $\bar{s}^{\text{overlap}} = 0.6708$, with a standard deviation of $\sigma^{\text{overlap}} = 0.1172$. The median of all overlap scores was $\tilde{s}^{\text{overlap}} = 0.6938$ and the trimmed mean was $\bar{s}^{\text{overlap}}_{0.1} = 0.6998$.

Summary

This experiment serves as a baseline for the other experiments. It scored a total MSE of 0.1161, while correctly predicting about 41.03% of phoneme boundaries (within the granted tolerance of 20 ms) and achieving a phoneme overlap of about 69.86%.

5.2.2 Multilingual Feedforward Neural Network

First Iteration

In the first iteration of the bootstrapping process, the multilingual system utilizing a feedforward neural network achieved a total MSE score of $\bar{s}^{\text{MSE}} = 0.1073$, with a standard deviation of $\sigma^{\text{MSE}} = 4.5812$. This is the best total MSE score across all experiments. The median of all MSE scores was $\tilde{s}^{\text{MSE}} = 0.0058$ and the trimmed mean was $\bar{s}^{\text{MSE}}_{0.1} = 0.0069$.

With the box scoring method, it achieved a mean score of $\bar{s}^{\text{box}} = 0.0709$, with a standard deviation of $\sigma^{\text{box}} = 0.0498$. The median of all box scores was $\tilde{s}^{\text{box}} = 0.0652$ and the trimmed mean was $\bar{s}^{\text{box}}_{0.1} = 0.0787$.

Finally, with the overlap scoring method, the system achieved a mean score of $\bar{s}^{\text{overlap}} = 0.4161$, with a standard deviation of $\sigma^{\text{overlap}} = 0.0978$. The median of all overlap scores was $\tilde{s}^{\text{overlap}} = 0.4174$ and the trimmed mean was $\bar{s}^{\text{overlap}}_{0.1} = 0.4360$.

Second Iteration

In its second iteration, this system achieved a slightly worse total MSE score of $\bar{s}^{\text{MSE}} = 0.1489$, with a standard deviation of $\sigma^{\text{MSE}} = 4.5837$. The median of all MSE scores was $\tilde{s}^{\text{MSE}} = 0.0134$ and the trimmed mean was $\bar{s}^{\text{MSE}}_{0.1} = 0.0183$.

The box scores were also worse than in the first iteration, with a mean box score of $\bar{s}^{\text{box}} = 0.0229$, with a standard deviation of $\sigma^{\text{box}} = 0.0326$. The median of all box scores was $\tilde{s}^{\text{box}} = 0.0122$ and the trimmed mean was $\bar{s}^{\text{box}}_{0.1} = 0.0254$.

A similar statement can be made about the overlap scoring method, where the system achieved a mean overlap score of $\bar{s}^{\text{overlap}} = 0.1832$, with a standard deviation of $\sigma^{\text{overlap}} = 0.0878$. The median of all overlap scores was $\tilde{s}^{\text{overlap}} = 0.1750$ and the trimmed mean was $\bar{s}^{\text{overlap}}_{0.1} = 0.1981$.

Summary

It can be seen that the results heavily depend on the applied scoring method.
Regarding the MSE scoring, the system encapsulating the multilingual feedforward network performed similar to the monolingual system – with a slightly better overall MSE score – although having fewer parameters, training data and epochs.
However, when considering the overlap scoring, it performed worse and with the box scoring it performed drastically worse.

These effects were even stronger in the second iteration of the bootstrapping process. This is probably linked to the dropped validation accuracies during second iteration training of the network.

5.2.3 Multilingual Time Delay Neural Network

First Iteration

In the first iteration, the system utilizing the multilingual time delay neural network achieved a total MSE score of $\bar{s}^{\text{MSE}} = 0.1452$, with a standard deviation of $\sigma^{\text{MSE}} = 5.1452$. The median of all MSE scores was $\tilde{s}^{\text{MSE}} = 0.0180$ and the trimmed mean was $\bar{s}^{\text{MSE}}_{0.1} = 0.0196$.

When scored with the box scoring method, the time delay system achieved a mean box score of $\bar{s}^{\text{box}} = 0.0146$, with a standard deviation of $\sigma^{\text{box}} = 0.0253$. The median of all box scores was $\tilde{s}^{\text{box}} = 0.0$ and the trimmed mean was $\bar{s}^{\text{box}}_{0.1} = 0.0163$.

Finally, with the overlap scoring method, the system achieved a mean overlap score of $\bar{s}^{\text{overlap}} = 0.1133$, with a standard deviation of $\sigma^{\text{overlap}} = 0.0721$. The median of all overlap scores was $\tilde{s}^{\text{overlap}} = 0.1024$ and the trimmed mean was $\bar{s}^{\text{overlap}}_{0.1} = 0.1242$.

Second Iteration

In the second iteration, this system achieved a total MSE score of $\bar{s}^{\text{MSE}} = 0.1616$, with a standard deviation of $\sigma^{\text{MSE}} = 3.3311$. The median of all MSE scores was $\tilde{s}^{\text{MSE}} = 0.0579$ and the trimmed mean was $\bar{s}^{\text{MSE}}_{0.1} = 0.0594$.

With the box scoring method, the time delay system achieved a mean box score of $\bar{s}^{\text{box}} = 0.0044$, with a standard deviation of $\sigma^{\text{box}} = 0.0154$. The median of all box scores was $\tilde{s}^{\text{box}} = 0.0$ and the trimmed mean was $\bar{s}^{\text{box}}_{0.1} = 0.0049$.

Finally, with the overlap scoring method, the system achieved a mean overlap score of $\bar{s}^{\text{overlap}} = 0.0181$, with a standard deviation of $\sigma^{\text{overlap}} = 0.0312$. The median of all overlap scores was $\tilde{s}^{\text{overlap}} = 0.0035$ and the trimmed mean was $\bar{s}^{\text{overlap}}_{0.1} = 0.0201$.

Summary

As with the feedforward system, the results of the system encapsulating the multilingual time delay neural network heavily depend on the scoring method.
With MSE scoring, the TDNN system performed slightly worse than the multilingual feedforward system, although its network had a higher validation accuracy during training. It also performed worse than the multilingual feedforward system in the other scoring methods.
Like with the feedforward system, these effects were amplified in the second iteration of the bootstrapping process.

5.2.4 Multilingual Stacked Bidirectional Long Short-Term Memory

First Iteration

The system encapsulating the most complexity, i.e. the one utilizing a stacked BiLSTM network, achieved a total MSE score of $\bar{s}^{\text{MSE}} = 0.3180$ in the first iteration, with a standard deviation of $\sigma^{\text{MSE}} = 8.6341$. The median of all MSE scores was $\tilde{s}^{\text{MSE}} = 0.1639$ and the trimmed mean was $\bar{s}^{\text{MSE}}_{0.1} = 0.1628$.

When scored with the box scoring method, the BiLSTM system achieved a mean box score of $\bar{s}^{\text{box}} = 0.0011$, with a standard deviation of $\sigma^{\text{box}} = 0.0088$. The median of all box scores was $\tilde{s}^{\text{box}} = 0.0$ and the trimmed mean was $\bar{s}^{\text{box}}_{0.1} = 0.0012$.

Finally, with the overlap scoring method, the system achieved a mean overlap score of $\bar{s}^{\text{overlap}} = 0.0021$, with a standard deviation of $\sigma^{\text{overlap}} = 0.0112$. The median of all overlap scores was $\tilde{s}^{\text{overlap}} = 0.0$ and the trimmed mean was $\bar{s}^{\text{overlap}}_{0.1} = 0.0023$.

Second Iteration

In its second iteration, the BiLSTM system achieved a total MSE score of $\bar{s}^{\text{MSE}} = 0.7250$, with a standard deviation of $\sigma^{\text{MSE}} = 4.1788$. The median of all MSE scores was $\tilde{s}^{\text{MSE}} = 0.6292$ and the trimmed mean was $\bar{s}^{\text{MSE}}_{0.1} = 0.6084$.

With the box scoring method, the BiLSTM system achieved a mean box score of $\bar{s}^{\text{box}} = 0.0009$, with a standard deviation of $\sigma^{\text{box}} = 0.0082$. The median of all box scores was $\tilde{s}^{\text{box}} = 0.0$ and the trimmed mean was $\bar{s}^{\text{box}}_{0.1} = 0.0010$.

Finally, with the overlap scoring method, the system achieved a mean overlap score of $\bar{s}^{\text{overlap}} = 0.0013$, with a standard deviation of $\sigma^{\text{overlap}} = 0.0112$. The median of all overlap scores was $\tilde{s}^{\text{overlap}} = 0.0$ and the trimmed mean was $\bar{s}^{\text{overlap}}_{0.1} = 0.0014$.

Summary

Across all scoring methods, the BiLSTM system performed the worst out of all tested architectures. This is despite it having the highest cross-lingual phoneme classification accuracy on the target data across all multilingual networks[1].
The second iteration system also had a decreased performance.

[1] See table 5.1.

Experiment	Iteration 1	Iteration 2
Multilingual FFNN	39.5%	**36.8%**
Multilingual TDNN	39.6%	33.5%
Multilingual Stacked BiLSTM	**41.1%**	30.2%

Table 5.1: Comparison of the cross-lingual phoneme classification accuracies on the data set of the target languages (English).

Experiment	$\bar{s}^{\text{MSE}}$	σ^{MSE}	$\tilde{s}^{\text{MSE}}$	$\bar{s}^{\text{MSE}}_{0.1}$
Monolingual FFNN (1)	0.1161	6.3992	**0.0028**	**0.0042**
Multilingual FFNN (1)	**0.1073**	4.5812	0.0058	0.0069
Multilingual TDNN (1)	0.1452	5.1452	0.0180	0.0196
Multilingual Stacked BiLSTM (1)	0.3180	8.6341	0.1639	0.1628
Multilingual FFNN (2)	0.1489	4.5837	0.0134	0.0183
Multilingual TDNN (2)	0.1616	3.3311	0.0579	0.0594
Multilingual Stacked BiLSTM (2)	0.7250	4.1788	0.6292	0.6084

Table 5.2: Comparison of the MSE scoring results between all experiments in the first and second iteration: total MSE score $\bar{s}^{\text{MSE}}$, as well as its standard deviation σ^{MSE}, median $\tilde{s}^{\text{MSE}}$ and trimmed mean (10%) $\bar{s}^{\text{MSE}}_{0.1}$.

Experiment	$\bar{s}^{\text{box}}$	σ^{box}	$\tilde{s}^{\text{box}}$	$\bar{s}^{\text{box}}_{0.1}$
Monolingual FFNN (1)	**0.4303**	0.1334	**0.44**	**0.4588**
Multilingual FFNN (1)	0.0709	0.0498	0.0652	0.0787
Multilingual TDNN (1)	0.0146	0.0253	0.0	0.0163
Multilingual Stacked BiLSTM (1)	0.0011	0.0088	0.0	0.0012
Multilingual FFNN (2)	0.0229	0.0326	0.0122	0.0254
Multilingual TDNN (2)	0.0044	0.0154	0.0	0.0049
Multilingual Stacked BiLSTM (2)	0.0009	0.0082	0.0	0.0010

Table 5.3: Comparison of the box scoring results between all experiments in the first and second iteration: mean box score $\bar{s}^{\text{box}}$, as well as its standard deviation σ^{box}, median $\tilde{s}^{\text{box}}$ and trimmed mean (10%) $\bar{s}^{\text{box}}_{0.1}$.

Experiment	$\bar{s}^{\text{overlap}}$	σ^{overlap}	$\tilde{s}^{\text{overlap}}$	$\bar{s}^{\text{overlap}}_{0.1}$
Monolingual FFNN (1)	**0.6708**	0.1172	**0.6938**	**0.6998**
Multilingual FFNN (1)	0.4161	0.0978	0.4174	0.4360
Multilingual TDNN (1)	0.1133	0.0721	0.1024	0.1242
Multilingual Stacked BiLSTM (1)	0.0021	0.0112	0.0	0.0023
Multilingual FFNN (2)	0.1832	0.0878	0.1750	0.1981
Multilingual TDNN (2)	0.0181	0.0312	0.0035	0.0201
Multilingual Stacked BiLSTM (2)	0.0013	0.0112	0.0	0.0014

Table 5.4: Comparison of the overlap scoring results between all experiments in the first and second iteration: mean overlap score $\bar{s}^{\text{overlap}}$, as well as its standard deviation σ^{overlap}, median $\tilde{s}^{\text{overlap}}$ and trimmed mean (10%) $\bar{s}^{\text{overlap}}_{0.1}$.

6 Conclusion

This chapter forms the conclusion to the thesis.
First, the whole work is briefly summarized, then the obtained results are put into context and interpreted, and finally, possible directions for further research are proposed.

6.1 Summary

The goal of this thesis was to apply cross-lingual, multilingual methods on the task of phoneme alignment. For this goal, three different neural network architectures were built, trained and utilized in a hybrid HMM/ANN system to align multilingual data. This process was iterated to bootstrap a multilingual acoustic model and the resulting system was used to cross-lingually align data from a previously unseen target language. Finally, the results were scored and compared against each other.
The first feedforward neural network was trained monolingually on German data, while the others – another feedforward neural network, a time delay neural network and a stacked bidirectional long short-term memory network – were trained on a multilingual data set, consisting of Spanish, French, Russian, Swedish and German data. All networks were put to use in a cross-lingual context, as they were evaluated on English data.

6.2 Interpretation of Results

In general, the multilingual systems did not outperform the monolingual system, although all multilingual networks had higher phoneme classification accuracies, at least in the first iteration of the bootstrapping process[1].
A reason for the missing transfer of these improved results into the alignment results could be an imprecise cross-lingual application of the systems, i.e. an imprecise mapping of phonemes between the five training languages and the target language.

The systems utilizing more complex network architectures had a decreased alignment performance compared to the ones using simpler architectures. This is the case despite them having increased phoneme classification accuracies, not only on the training data set, but also on the evaluation data set in the target language[2].

The performance of all systems decreased in the second iteration of the bootstrapping process. This holds for considering the phoneme classification accuracy of their neural networks, cross-

[1] When comparing the validation/phoneme recognition accuracies of the monolingual and multilingual systems, one has to take into account that different validation data sets are used for this comparison. However, it can be assumed that the multilingual data set is more diverse than the monolingual one, as it contains German data plus data from four other languages. Still, every multilingual network performed better on the isolated task of phoneme classification than the monolingual one.
[2] See table 5.1.

lingual phoneme classification accuracy of their networks or alignment results of the whole systems.
Again, an imprecise mapping of phonemes could be the reason for this phenomenon, as this could lead to inaccurate labeling results in the first iteration, which in turn would decrease the quality of the training data in the second iteration, making it more complex to learn. The consequences would then be a decreased neural network validation accuracy and worse labeling results.

6.3 Further Research

Further research could address the possible problems stated in the above section, with the aim of alleviating them and obtaining better results.
This would include a more careful mapping of phonemes between languages in the bootstrapping process by employing more profound linguistic knowledge or even using data-driven approaches. Another attempt could be to choose the training languages more carefully, i.e. by comparing lexical similarities or other linguistic distances to the target language.

An alternative approach could be to improve the systems capability to handle multilingual data, for example by introducing modulation techniques[3].

[3]See the results presented in the Related Work chapter.

Bibliography

[SVN37] Stanley Smith Stevens, John Volkmann, and Edwin Broomell Newman. "A scale for the measurement of the psychological magnitude pitch". In: *The journal of the acoustical society of america* 8.3 (1937), pp. 185–190.

[MP43] Warren S McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133.

[Ros58] Frank Rosenblatt. "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain". In: *Psychological Review* 65.6 (1958).

[Nov63] Albert B Novikoff. *On convergence proofs for perceptrons*. Tech. rep. Stanford Research Inst, 1963.

[Coc+67] William T Cochran et al. "What is the fast Fourier transform?" In: *Proceedings of the IEEE* 55.10 (1967), pp. 1664–1674.

[RHW85] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. *Learning internal representations by error propagation*. Tech. rep. University of California, San Diego, 1985.

[RJ86] Lawrence Rabiner and Biinghwang Juang. "An introduction to hidden Markov models". In: *IEEE ASSP Magazine* 3.1 (1986), pp. 4–16.

[Ous+89] John K Ousterhout et al. *Tcl: An embeddable command language*. 1989.

[Wai+89] 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.

[Elm90] Jeffrey L Elman. "Finding structure in time". In: *Cognitive science* 14.2 (1990), pp. 179–211.

[FLW90] Michael Franzini, K-F Lee, and Alex Waibel. "Connectionist Viterbi training: a new hybrid method for continuous speech recognition". In: *International Conference on Acoustics, Speech, and Signal Processing*. IEEE. 1990, pp. 425–428.

[LP91] PJG Lisboa and SJ Perantonis. "Complete solution of the local minima in the XOR problem". In: *Network: Computation in Neural Systems* 2.1 (1991), pp. 119–124.

[SSW91] Masahide Sugiyama, Hidehumi Sawai, and Alexander H Waibel. "Review of tdnn (time delay neural network) architectures for speech recognition". In: *1991., IEEE International Sympoisum on Circuits and Systems*. IEEE. 1991, pp. 582–585.

[BSF94] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE transactions on neural networks* 5.2 (1994), pp. 157–166.

[Die95] Tom Dietterich. "Overfitting and undercomputing in machine learning". In: *ACM computing surveys (CSUR)* 27.3 (1995), pp. 326–327.

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

[LeC+99] Yann LeCun et al. "Object recognition with gradient-based learning". In: *Shape, contour and grouping in computer vision*. Springer, 1999, pp. 319–345.

[Cry00] David Crystal. *Language death*. Ernst Klett Sprachen, 2000.

[CBM02] Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. *Torch: A modular machine learning software library*. Tech. rep. Idiap, 2002.

[RN02] Stuart Russell and Peter Norvig. "Artificial intelligence: a modern approach". In: (2002).

[Woo03] Anthony C. Woodbury. "Defining documentary linguistics". In: *Language documentation and description* (2003), pp. 35–51.

[GS05] Alex Graves and Jürgen Schmidhuber. "Framewise phoneme classification with bidirectional LSTM and other neural network architectures". In: *Neural networks* 18.5-6 (2005), pp. 602–610.

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization." In: *Journal of machine learning research* 12.7 (2011).

[MDH11] Abdel-rahman Mohamed, George E Dahl, and Geoffrey Hinton. "Acoustic modeling using deep belief networks". In: *IEEE transactions on audio, speech, and language processing* 20.1 (2011), pp. 14–22.

[Hin+12] Geoffrey E Hinton et al. "Improving neural networks by preventing co-adaptation of feature detectors". In: *arXiv preprint arXiv:1207.0580* (2012).

[Sta+12] Felix Stahlberg et al. "Word segmentation through cross-lingual word-to-phoneme alignment". In: *2012 IEEE Spoken Language Technology Workshop (SLT)*. IEEE. 2012, pp. 85–90.

[GJ14] Alex Graves and Navdeep Jaitly. "Towards end-to-end speech recognition with recurrent neural networks". In: *International conference on machine learning*. PMLR. 2014, pp. 1764–1772.

[Han+14] Awni Y Hannun et al. "First-pass large vocabulary continuous speech recognition using bi-directional recurrent DNNs". In: *arXiv preprint arXiv:1408.2873* (2014).

[KB14] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[IS15] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *International conference on machine learning*. PMLR. 2015, pp. 448–456.

[Nie15] Michael A Nielsen. *Neural networks and deep learning*. Vol. 25. Determination press San Francisco, CA, 2015.

[PJP15] Jayashree Padmanabhan and Melvin Jose Johnson Premkumar. "Machine learning in automatic speech recognition: A survey". In: *IETE Technical Review* 32.4 (2015), pp. 240–251.

[PPK15] Vijayaditya Peddinti, Daniel Povey, and Sanjeev Khudanpur. "A time delay neural network architecture for efficient modeling of long temporal contexts". In: *Sixteenth Annual Conference of the International Speech Communication Association*. 2015.

[Add+16] Gilles Adda et al. “Innovative technologies for under-resourced language documentation: The BULB Project”. In: *Workshop CCURL 2016-Collaboration and Computing for Under-Resourced Languages-LREC*. 2016.

[Fra+16] Jörg Franke et al. “Phoneme boundary detection using deep bidirectional lstms”. In: *Speech Communication; 12. ITG Symposium*. VDE. 2016, pp. 1–5.

[Goo+16] Ian Goodfellow et al. *Deep learning*. Vol. 1. 2. MIT press Cambridge, 2016.

[Mül18] Markus Müller. “Multilingual Modulation by Neural Language Codes”. PhD thesis. Karlsruhe Institute of Technology, Germany, 2018.

[MSW18] Markus Müller, Sebastian Stüker, and Alex Waibel. “Multilingual adaptation of RNN based ASR systems”. In: *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2018, pp. 5219–5223.

[Ard+19] Rosana Ardila et al. “Common voice: A massively-multilingual speech corpus”. In: *arXiv preprint arXiv:1912.06670* (2019).

[Pas+19] Adam Paszke et al. “Pytorch: An imperative style, high-performance deep learning library”. In: *Advances in neural information processing systems* 32 (2019), pp. 8026–8037.

[Li+20] Xinjian Li et al. “Universal phone recognition with a multilingual allophone system”. In: *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2020, pp. 8249–8253.

[ESF21] David M. Eberhard, Gary F. Simons, and Charles D. Fenning. *Ethnologue: Languages of the World. Twenty-fourth edition*. 2021. URL: `https://www.ethnologue.com` (visited on 05/29/2021).