THESIS

END-TO-END SPEECH RECOGNITION USING CONNECTIONIST TEMPORAL CLASSIFICATION

Master Thesis

by

Marc Dangschat

Thesis submitted to the Department of Electrical Engineering and Computer Science, University of Applied Sciences Münster, in partial fulfillment of the requirements for the degree

Master of Science (M.Sc.)

Advisor: Prof. Dr.-Ing. Jürgen te Vrugt
Co-Advisor: Prof. Dr. rer. nat. Nikolaus Wulff

Münster, October 11, 2018
Abstract

Speech recognition on large vocabulary and noisy corpora is challenging for computers. Recent advances have enabled speech recognition systems to be trained end-to-end, instead of relying on complex recognition pipelines. A powerful way to train neural networks that reduces the complexity of the overall system. This thesis describes the development of such an end-to-end trained speech recognition system. It utilizes the connectionist temporal classification (CTC) cost function to evaluate the alignment between an audio signal and a predicted transcription. Multiple variants of this system with different feature representations, increasing network depth and changing recurrent neural network (RNN) cell types are evaluated. Results show that the use of convolutional input layers is advantageous, when compared to dense ones. They further suggest that the number of recurrent layers has a significant impact on the results. With the more complex long short-term memory (LSTM) outperforming the faster basic RNN cells, when it comes to learning from long input-sequences. The initial version of the speech recognition system achieves a word error rate (WER) of 27.3%, while the various upgrades decrease it down to 12.6%. All systems are trained end-to-end, therefore, no external language model is used.
# Contents

**Glossary**

**List of Abbreviations**

1. **Introduction**  
   1.1 Automatic Speech Recognition  
   1.2 Contributions  
   1.3 Thesis Structure  

2. **Theory**  
   2.1 Input and Output  
   2.1.1 Bucketing  
   2.2 Activation Functions  
   2.2.1 Softmax  
   2.3 Dense Layer  
   2.4 Convolutional Layer  
   2.5 Dropout  
   2.6 Recurrent Neural Networks  
   2.6.1 Recurrent Neural Network Cell  
   2.6.2 Long Short-Term Memory  
   2.6.3 Gated Recurrent Unit  
   2.6.4 Bidirectional-Recurrent Neural Network  
   2.7 Connectionist Temporal Classification  
   2.7.1 Alignments  
   2.7.2 Determining Valid Alignments  
   2.7.3 Intermediate Cost Function  
   2.7.4 Cost Function  
   2.7.5 Inference  
   2.7.6 Connectionist Temporal Classification Implementations  
   2.8 Summary  

3. **Related Work**  
   3.1 History of Speech Recognition Technologies  
   3.2 Connectionist Temporal Classification Models  
   3.3 Encoder-Decoder Based Models  
   3.4 Summary  

4. **Method**  
   4.1 Training Material  
   4.1.1 Corpora Selection  
   4.1.2 Preparations  
   4.1.3 Statistics  
   4.2 Features  
   4.2.1 Feature Types  
   4.2.2 Feature Normalization  
   4.3 Model Description  

iv  
v  
1  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40
4.4 Training Process ................................................. 43
4.5 Extended Model ................................................. 44
  4.5.1 Different Cell Types ....................................... 46
  4.5.2 Gradually Increasing Training Difficulty ............... 46
4.6 Summary .......................................................... 47

5 Evaluation .......................................................... 48
  5.1 Baseline System ................................................. 49
  5.2 Input Feature Types ............................................ 49
  5.3 Normalization Methods ....................................... 50
  5.4 Number of Units per Layer .................................. 51
  5.5 Comparison with Reference System ......................... 51
  5.6 Extended Model ................................................ 52
  5.7 Comparison of Convolutional and Dense Input Layers .... 52
  5.8 Number of Recurrent Layers ................................ 53
  5.9 Different Cell Types ......................................... 54
  5.10 Gradually Increasing Training Difficulty ............... 55
  5.11 General Observations ....................................... 56
  5.12 Summary ...................................................... 58

6 Conclusion .......................................................... 59
  6.1 Summary ........................................................ 59
  6.2 Limitations .................................................... 60
  6.3 Practical Implications ....................................... 60
  6.4 Future Work .................................................... 60

List of Figures .......................................................... 62

List of Tables .......................................................... 63

References ............................................................. 64

A Appendices .......................................................... 68
  A.1 Additional Corpora Information .............................. 68
  A.2 Confidence Interval Estimations ............................ 70
  A.3 Server Setup ................................................... 71
Glossary

beam search A greedy search algorithm. Similar to breadth-first search, but only considering a limited number of best states. 6, 15, 24–27, 31, 42, 43, 70

cost Measurement of how well a model has classified a given input. The cost function is also known as loss, error or objective function. 2, 3, 5, 15, 16, 19, 21–24, 26, 29, 42, 43, 47–50, 52, 53, 55, 56, 59

cuDNN The Nvidia compute unified device architecture (CUDA) deep neural network library (cuDNN) is a GPU-accelerated collection of highly tuned primitives for deep neural networks. 26, 42

dend-to-end An end-to-end trained system is a single deep neural network (DNN) that is trained as a whole. It does not have any subcomponents that are trained independently. It directly learns to convert the given input data (e.g. audio) into the desired output form (e.g. text), without intermediate states. i, 2–4, 27–33, 38, 51, 57, 59, 60

grapheme Graphemes are single letters or groups of letters, like “s” and “ee” in “speech”. 2, 29

infer The process of evaluating an input with a neural network in order to classify it, is called inference. 6, 9, 15, 28, 40

overfitting Overfitting occurs when a network has extracted more information from the training data than justified. In other words it has memorized the training data, instead of learning to classify it. 9, 38, 49

phoneme Phonemes are the individual sounds made by speaking, like /p/ in /speech/. 1, 16, 27–29
List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM</td>
<td>acoustic model 1, 27, 28, 30</td>
</tr>
<tr>
<td>ASR</td>
<td>automatic speech recognition 1–6, 14, 15, 19, 26–34, 36, 38–41, 47, 48, 51, 55–61, 68</td>
</tr>
<tr>
<td>CC</td>
<td>Creative Commons 47, 68</td>
</tr>
<tr>
<td>CNN</td>
<td>convolutional neural network 8, 9, 26, 45, 46, 50, 52, 53, 58</td>
</tr>
<tr>
<td>CTC</td>
<td>connectionist temporal classification i, 2–6, 15–17, 19, 21–32, 36, 39, 42, 43, 47–50, 52–56, 59, 60</td>
</tr>
<tr>
<td>DNN</td>
<td>deep neural network iv, 2, 4, 5, 11, 27, 28, 32, 40, 58, 59</td>
</tr>
<tr>
<td>ED</td>
<td>edit distance 43, 48–51, 53, 54, 57, 58, 70</td>
</tr>
<tr>
<td>GRU</td>
<td>gated recurrent unit 3, 10, 13, 14, 26, 29, 31, 46, 54, 55, 58–60</td>
</tr>
<tr>
<td>HMM</td>
<td>hidden markov model 27, 28, 32, 33</td>
</tr>
<tr>
<td>LAS</td>
<td>Listen, Attend and Spell 30, 31</td>
</tr>
<tr>
<td>LM</td>
<td>language model 1–3, 27, 29–32, 51, 52, 56–60</td>
</tr>
<tr>
<td>LR</td>
<td>learning rate 43, 53, 55, 56, 59</td>
</tr>
<tr>
<td>LSTM</td>
<td>long short-term memory i, 3, 10–13, 26, 29, 30, 46, 49, 51, 52, 54, 55, 58, 59, 71</td>
</tr>
<tr>
<td>LVCSR</td>
<td>large vocabulary continuous speech recognition 33, 60</td>
</tr>
<tr>
<td>MELSPEC</td>
<td>mel spectrogram 1, 3, 5, 29, 38, 39, 41, 47, 49–55, 58, 59</td>
</tr>
<tr>
<td>MFCC</td>
<td>mel frequency cepstral coefficient 3, 5, 29, 31, 38, 39, 47, 49–52, 58, 59</td>
</tr>
<tr>
<td>NN</td>
<td>neural network 2, 5, 7–10, 12, 13, 26, 27, 29, 31, 32, 40–42, 44–47, 53, 54, 59, 60</td>
</tr>
<tr>
<td>PM</td>
<td>pronunciation model 1, 2, 27, 30</td>
</tr>
<tr>
<td>ReLU</td>
<td>rectified linear unit 7, 11, 42, 44, 54</td>
</tr>
<tr>
<td>RNN</td>
<td>recurrent neural network i, 2, 3, 5, 6, 10–12, 14–20, 22, 23, 26, 27, 29–31, 33, 39, 41–47, 49–59, 71</td>
</tr>
<tr>
<td>STD</td>
<td>standard deviation 40, 50, 51, 70</td>
</tr>
<tr>
<td>WER</td>
<td>word error rate i, 29, 31, 43, 46, 48–60, 70</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Automatic Speech Recognition

Automatic speech recognition (ASR), also known as speech to text (STT) or just speech recognition, is the computer-driven process of converting a spoken utterance into its appropriate transcription. Transcribing unsegmented speech data requires the system to convert input streams of real-valued, noisy data into strings of discrete labels. The theoretical optimum for such a process is that it recognizes all intelligibly spoken words with perfect accuracy. Even with the ever decreasing error rates seen in modern ASR systems, they still have a way to go to achieve near perfect accuracy. As current systems are often trained for specific tasks, where they can achieve error rates that are comparable to those of humans [Sto+17; Pri17]. Suppose a system trained in a controlled environment, with a set of specific microphones, for the commands to a smartspeaker. Such a system would have trouble understanding a person driving a car, where the signal-to-noise ratio is much lower. Therefore, it fails the overall objective of being the ASR system that works for everyone, every time.

ASR systems can provide a natural way of human-machine interaction. To accomplish this, they have to achieve error rates that are low enough for the given task, over a broad spectrum of different inputs. For comparison, human control groups achieve word error rates between 5.1% and 5.8% on common ASR corpora [Sto+17; Pri17]. For most users to perceive voice inputs to a computer as natural, the transcription error rate has to be below that of humans [Ng16]. Given the complexity of human languages, this is no trivial task. Spoken utterances vary greatly based on the speaker’s emotion, speaking rate, dialect, gender and education. It is further affected by the recording quality and environment (e.g. background noise and reverberation). The usage of new words (e.g. proper names) or slang that is unknown to the system can be challenging. Another hindrance are homophones (e.g. bight, bite and byte) where different words are pronounced in the same way.

![Recognition Pipeline](image)

Figure 1: The recognition pipeline of a traditional ASR system. Features get extracted from an audio signal and sent to the acoustic model (AM). Which in turn generates prediction scores for sub-words. The pronunciation model (PM) converts sequences of sub-word predictions into actual words. Finally, the language model (LM) provides an additional score for the predicted words. All components are used in conjunction, to search for the most probable transcription for the given audio signal.

Traditional ASR systems are comprised of several parts that are trained independent of each other. These individual components are then connected to make up a *speech recognition pipeline*. A common setup is illustrated in figure 1, it contains an acoustic model (AM), a pronunciation model (PM) and a language model (LM). The AM transforms acoustic features (e.g. mel spectrogram (MELSPEC) windows) into a set of sub-words, for example into phonemes
or graphemes. Based on a sequence of sub-words, the PM predicts a sequence of words. The PM rules are generally hand-crafted, containing entries like the phoneme /kæt/ that maps to the word “cat”. Finally, the LM calculates a probability for each predicted sequence of words [Sto+17]. These LM probabilities are based on large text corpora, representing the targeted language and domain. Developing and training of all these independent components is increasing the complexity of the overall system. Because of that, there has been interest in developing systems that can train all the components jointly. Such end-to-end trained systems have shown solid results in recent years [Gra+14; Han+14; Amo+15; Bah+16].

To increase the areas of use for speech recognition, the systems have to work on large, natural vocabularies, with different speakers and a reasonable delay. One could argue that for most personal assistants this reasonable delay is close to real time. Deep neural networks (DNNs) and end-to-end training have the potential to be important milestones in the advancement of speech recognition systems.

Doctors and lawyers use ASR systems to dictate notes. Motorists use them as a hands-free control for their vehicles multimedia system. Even fighter pilots use them to control non vital parts of their flight-computer [Eur06]. Call centers employ speech recognition technologies to prioritize and connect callers to the appropriate department. With the rise of smartphones, wearables (e.g. smartwatches, smartglasses), home automation (e.g. smartspeakers), voice recognition can become the predominant input method for such devices. Furthermore, they are an integral part of natural language processing (NLP) systems [Den+18, p. 1]. With constantly improving classification rates, ASR systems can rival traditional human-computer interfaces and in the future perhaps become the dominant interface [Ng16].

This thesis describes an end-to-end trained ASR system, based on recurrent neural networks (RNNs) and the connectionist temporal classification (CTC) cost function. While traditional feed-forward neural networks (NNs) can classify sub-word units and even individual words, they do not work well for continuous speech. This is because they are not proficient at modeling temporal dependencies [Gra+06]. Given an audio-transcription tuple \((x, y)\) for training, it is generally unknown how the transcription \(y\) aligns to the audio signal \(x\). In fact, both the audio length \(T\) and the transcription length \(U\) can vary, with the ratio between them not being fixed; this poses a challenge in speech recognition. RNNs are specifically designed to work on sequence-data. They produce one prediction for every time-frame of input \((1 : 1)\). Thus, the predicted output has to be transformed into the actual label sequence \((T : U)\) during post-processing. For example the used training corpus has an average audio length of \(T = 569\) feature frames or 5.69 seconds (section 4.1). The RNN has to align these to the transcription which is on average only \(U = 80\) characters long. This requires a new form of cost function that is able to evaluate such alignments. CTC is one possible solution for this. It enables the training of an ASR system on pairs of speech and transcription, without the use of intermediate states or knowledge of the alignment between them.
1.2 Contributions

The primary contribution of this thesis is the development and description of an end-to-end trained speech recognition system. It is based on the Deep Speech (1) system described in [Han+14]. The ASR system uses a recurrent neural network (RNN) and the connectionist temporal classification (CTC) cost function to determine the model’s error during training. It further has the following properties:

- it is an end-to-end trained system;
- it is speaker independent;
- it works for a single speaker at a time, as opposed to conversational speech;
- it recognizes clean and noisy speech;
- it transcribes English utterances;
- it is targeted at inputs that are between 0.7 and 17 seconds long.

Different input features (i.e. mel spectrogram (MELSPEC) and mel frequency cepstral coefficient (MFCC)) and input normalization methods are evaluated. The system is trained on speech corpora that are free to use for educational purposes. No special linguistically annotated training data, like treebanks, is required. The following is explicitly not part of the system:

- the integration of an external language model (LM),
- transcription of multiple languages with one model,
- the handling of ambiguous pronunciations for the same word (e.g. “triple A” and “AAA”),
- the integration of an attention mechanism (encoder-decoder paradigm)
- and distributed training.

The above system is evaluated and compared to a reference ASR system. Its computational requirements are low enough, so that it can be trained on the currently available consumer hardware. Expanding on this baseline ASR system, modifications to improve it are implemented and evaluated. These modifications are based on the Deep Speech 2 system [Abk+15]. Individual upgrades are then compared with the baseline measurements. This added complexity increases the computational demands, therefore the expanded system can no longer be trained on current consumer hardware. With the improved system the following aspects are evaluated:

- convolutional input layers are compared to dense ones;
- the effects of an increased amount of RNN layers;
- an alternative normalization method;
- and the effects of more complex RNN cells, namely gated recurrent units (GRUs) and long short-term memory (LSTM) cells.
1.3 Thesis Structure

The remainder of this thesis is as follows. Section 2 describes the underlying algorithms used to construct DNNs that are used in ASR systems. Section 3 provides a brief summary of the historical speech recognition techniques, up until the point when end-to-end trained systems emerged. Following is an overview of related work from the two commonly used methods for end-to-end trained speech recognition systems; that are connectionist temporal classification (CTC) and attention based systems. In section 4 the design of the baseline ASR system and the corpora used for training are explained. This is followed by a description of the models training process. Section 4 closes with a listing of the enhancements made to the described baseline model. Section 5 then evaluates the previously described systems. It describes the used error metrics and presents the results of the performed experiments. Following each result is a discussion and interpretation, which further points out the limitations of the chosen approach. The thesis closes with a conclusion in section 6, highlighting the most important results and suggesting future work.
2 Theory

This section\(^1\) covers the building blocks of NNs that can be used to construct a state-of-the-art ASR system. Beginning with the inputs to the network, up onto the interpretation and decoding of the output. Providing insight into the underlying algorithms that make up the speech recognition models described later on. It further describes a cost metric that is used to train a RNN on unaligned tuples of input-sequences and labels, namely the CTC algorithm.

Since the following formulas use plenty of superscript and subscript, they are omitted when they are not required for the specific aspect of the section. Vectors \(v\) and matrices \(M\) are marked bold in this work. Unless stated otherwise, all matrices and vectors are assumed to be real valued. For example \(Mv\) represents the multiplication of matrix \(M\) with vector \(v\), it is further assumed that both have valid dimensions. The element-wise application of a function \(f\) to the elements of vector \(v\) is denoted with \(f(v)\). The term componentwise operation refers to operations on vectors with equal length \(N\). A componentwise operation between two vectors \(a\) and \(b\) is defined as:

\[
\begin{bmatrix}
a_1 \\
a_2 \\
\vdots \\
a_N
\end{bmatrix} \circ 
\begin{bmatrix}
b_1 \\
b_2 \\
\vdots \\
b_N
\end{bmatrix} = 
\begin{bmatrix}
a_1 \circ b_1 \\
a_2 \circ b_2 \\
\vdots \\
a_N \circ b_N
\end{bmatrix}
\]

where \(\circ \in \{+,-,\ast,/\}\). To distinguish the componentwise multiplication and division, the \(\otimes\) and \(\odot\) notations are used respectively.

If a sequence \(S = [A, B, C, \cdots]\) is given, then \(|S|\) represents the number of elements in it. \(S_i\) represents the \(i\)-th element of the sequence. Let \(1 \leq i < j \leq |S|\), then \(S_{i:j}\) is the subsequence of \(S\) from the \(i\)-th to the \(j\)-th element, therefore, \(S_{1:|S|}\) is an alternative form of \(S\).

The following figures generally use green boxes to indicate some form of NN. The inputs and outputs of these NNs are represented by blue and purple circles, respectively.

NNs comprised of the following components can be trained efficiently, using the backpropagation through time algorithm [Wer88]. Its objective is to reduce the cost or error of the network for a given input example. The audio-transcription tuple \((x, y)\) and the current NN parameters \(\theta\) are used to calculate the networks cost \(\mathcal{L}(x, y; \theta)\). Its derivative \(\nabla_\theta \mathcal{L}(x, y; \theta)\), with respect to the networks parameters, is then used to update the weights via gradient descent. A in detail description of the backpropagation algorithm can be found in [Wer88; Ska18, pp. 93–102; Yu+14, pp. 61–65; Hea15, pp. 113 – 130].

2.1 Input and Output

The network’s inputs are tuples from a dataset \(\mathcal{X}\). These tuples consist of an utterance \(x\) and a label \(y\): \(\mathcal{X} = \{ (x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \cdots \}\). Let \((x^{(i)}, y^{(i)}) \in \mathcal{X}\), then each utterance \(x^{(i)}\) is a time-series of length \(T^{(i)}\), where every discrete time-slice is a vector of audio features\(^2\): \(x_t^{(i)}, t \in \{1, 2, \cdots, T^{(i)}\}\). The time-slices \(x_t^{(i)}\) are vectors of fixed length \(L\), for example \(L = 26\) MFCC features per time-slice. The label \(y_t^{(i)}\) is a vector of length \(U^{(i)}\), where each element is a character of the target alphabet \(y_t^{(i)} \in \{a, b, c, \cdots, z, \_\}, t \in \{1, 2, \cdots, U^{(i)}\}\) (section 4.1.2).

\(^1\) The section is written in cooperation with Weweler, including the exchange of relevant literature. Weweler is developing a DNN based text-to-speech system (TTS) at the time [Wew18].

\(^2\) Used features are spectrograms (i.e. mel frequency cepstral coefficient (MFCC) and mel spectrogram (MELSPEC)), they are described in section 4.2.1.
With _ representing the space character. Note that the audio length $T^{(i)}$ it not necessarily equal to the label length $U^{(i)}$. [Han+14]

Given an input vector $x^{(i)}$, the network outputs a sequence of character probabilities $\hat{y}^{(i)}_t = P(a_t|x^{(i)})$ for each time-step $t \in \{1, \cdots, T^{(i)}\}$, with $a_t \in \{a, b, c, \cdots, z, _\}$ from the target alphabet. Through training the network is nudged to have the highest likelihood for a character sequence that matches the original label sequence $y^{(i)}$. To infer the path with the highest overall probability (i.e. the most likely transcription), a search algorithm like beam search can be used. Evaluation and decoding of such a sequence of character probabilities is explored in section 2.7, which describes the CTC algorithm. [Han+14]

### 2.1.1 Bucketing

Since the later described ASR system does not use online training where the weights are updated after every single input example; but instead uses mini-batches for training, where the gradients of a whole mini-batch of inputs are summed up and only then the networks weights are updated [Hea15, pp. 125 – 127]. For batch creation, the system has to account for varying input length $T^{(i)}$ in the used corpora (section 4.1). To accomplish this the inputs are grouped roughly based on their length, using a method called bucketing. Instead of creating a batch from examples of the whole dataset, where the length can vary extremely. Examples of the training dataset $(x^{(i)}, y^{(i)}) \in X$ with similar audio length $T^{(i)}$ are grouped into buckets. Each bucket can hold a number of examples to create batches. For each batch created from a bucket, every audio example $x^{(i)}$ and label $y^{(i)}$ is padded to the same length, as the longest example in the batch.

![Figure 2: The bucket boundaries (red) aligned on the length distribution of the training examples (green). This allows each of the 64 bucket to fill up at an equal rate. Some durations are common enough to receive a bucket containing only same length examples. Those are clustered around the 2 second mark. The training examples are plotted using a total of 75 bins.](image)

To evaluate an input, a RNN requires as many unrolls as the input is long (section 2.6). Bucketing is used to reduce the overall number of unrolls required. Ideally all examples within a batch have the same length so that the amount of padding is minimal. This can be problematic for example lengths that are uncommon (e.g. examples that are longer than 16 seconds in figure 2), because it reduces variance within these batches. Or even prevent assembly of complete batches, due to a lack of examples with the required length. To compensate for this the bucket boundaries are chosen based on the audio length distribution of the training dataset. Assuming that the examples are read in random order, this should on average allow for each bucket to fill up at the same rate. An example of this is shown in figure 2, it illustrates how the bucket boundaries are aligned to the training datasets length distribution.
2.2 Activation Functions

The function name $g$ is used to indicate an arbitrary activation function. The most basic activation function is called linear activation function, it simply returns the original value $g(x) = x$. This is commonly used in the output layer of NNs during regression tasks, where the network learns to predict a numeric value. [Hea15, pp. 12 sq.]

However, the purpose of most activation functions is to introduce non-linearity into the NN, allowing for more complex classification patterns. Figure 3 shows plots of some frequently used activation functions. They include the sigmoid ($\sigma$) function in equation (1), the hyperbolic tangent ($\tanh$), and the rectified linear unit (ReLU) function in equation (2). With the last one being inexpensive to compute due to its simple derivative. [Hea15, pp. 14 – 17]

\[
\sigma(x) = \frac{1}{1 + e^{-x}} \tag{1}
\]

\[
relu(x) = \max(0, x) \tag{2}
\]

2.2.1 Softmax

Like linear activation functions, the softmax activation function is usually used in the output layer of a NN. While linear activations can be used for regression, softmax is used in classification tasks where the input is assigned a discrete class. This can be done by the unit with the highest activation, claiming the input as a member of its class.

The softmax function takes the linear activations of the final layer as input. It then converts them into a probability distribution over the outputs; that is each unit’s activation $\phi_i$ is between 0 and 1 ($0 \leq \phi_i \leq 1$) and the sum over all softmax activated outputs is $\sum_i \phi_i = 1$. For the $i$-th unit of layer $h$ the softmax activation $\phi_i$ is given by:

\[
\phi_i(h) = \frac{e^{h_i}}{\sum_{j=1}^{J} e^{h_j}} \tag{3}
\]

where $J$ is the number of units (i.e. the number of classes) in the output layer. [Hea15]
2.3 Dense Layer

Dense layers, also known as fully connected layers are a basic building block for NNs. They connect each of their units (neurons) with every unit of its input layer, hence the name dense layer. The output of the $j$-th dense layer $h^{(j)}$ of a NN is given by:

$$h^{(j)} = g\left(W^{(j)}h^{(j-1)} + b^{(j)}\right)$$  \hspace{1cm} (4)

where $j \in \mathbb{N} \geq 1$ is the indicator for the $j$-th layer. By definition $h^{(0)}$ is equal to the network’s input $x$. $W^{(j)}$ is the $m \times n$ dimensional weight matrix of the $j$-th layer, where $m = \text{length}(h^{(j-1)})$ and $n = \text{length}(h^{(j)})$. Its dimensionality is omitted to improve readability. $b^{(j)}$ is the bias vector of the $j$-th layer. Finally, $g$ represents some activation function. [Goo+16, p. 189]

2.4 Convolutional Layer

Convolutional neural networks (CNNs) get their name from the mathematical operation with the same name. They were first introduced into machine learning for image-recognition tasks. The idea is to emulate the parts of the human visual cortex that use local receptive fields to identify objects. This is done by groups of neurons that only consider small, localized parts of an image for their decision, they are called kernel or filter. The kernel is then moved over the whole image, extracting features for every visited part of the input. NNs typically employ many of these kernels per convolutional layer. Each individual kernel learns a specific feature that is universally helpful for recognition. The number of kernels in a layer is commonly referred to as number of channels. In contrast to the fully connected dense layers, that use one weight per input and per unit; convolutional layers are sparsely connected, with one weight responsible for multiple input units. [Hea15, pp. 174 – 175]

This subsection covers the two-dimensional case as it is used later on, however, any dimensionality is possible. Only discrete convolutions are considered. The local receptive field in CNNs is realized via a learnable weight matrix $K$, the kernel. It is applied to parts of a two-dimensional input (e.g. an image or a spectrogram). A local receptive field that considers a $K_w \times K_h$ area of the input is called a kernel or filter of size $K_w \times K_h$. The second parameter for a convolutional layer is the kernels stride or hop. It governs how far the kernel is moved in any direction to evaluate the next window. For the sake of argument, assume that the first kernel is applied in the top left corner of the input. A stride of $(S_w, S_h)$ is moving the kernel in increments of $S_w$ to the right. Until it reaches the far right side, then it is moved back to the left and down by $S_h$ units. Figure 4 illustrates how a $2 \times 2$ kernel, with a stride of $(1, 1)$ is applied to the first position of a $4 \times 3$ input. Results for the subsequent convolution steps are indicated as well. [Goo+16, pp. 327 – 329]

Application of the convolution operation, denoted by $*$, to a two-dimensional input matrix $I_{I_h \times I_w}$ produces the output matrix $C_{C_h \times C_w}$. Its elements are:

$$C_{i,j} = (I * K) (i,j) = \sum_{m=1}^{K_w} \sum_{n=1}^{K_h} I (m + i, n + j) K (m,n)$$  \hspace{1cm} (5)

where $K_{K_h \times K_w}$ is the kernel’s weight matrix. The indices $1 \leq i \leq I_w; 1 \leq j \leq I_h$ are incremented based on the desired stride. Assuming that no input padding is required, the dimensionality of the resulting matrix $C_{C_h \times C_w}$ is:

$$C_w = (I_w - K_w) / S_w + 1$$
$$C_h = (I_h - K_h) / S_h + 1$$  \hspace{1cm} (6)
Figure 4: Example for a discrete convolution operation. A 2 × 2 kernel applied to a 4 × 3 input. With a stride of (1, 1), the resulting output has the dimensionality 3 × 2. Figure derived from [Goo+16, p. 330].

where \( a//b \) represents the integer division of \( a \) and \( b \). [Goo+16, pp. 327 – 329]

Optimized CNN implementations often use cloned weight matrices and execute a single large matrix multiplication instead of using the described iterative process. [Goo+16, p. 329]

### 2.5 Dropout

*Dropout* is a regularization method used to prevent a NN from overfitting the training data. During training, it is “hiding” a certain percentage of input units from a layer. This prevents the layer’s units from becoming overly dependent on certain inputs. The dropped units are chosen randomly at the beginning of each new training step. Dropped units do not receive a weight update for that step. Dropout can be used in fully connected layers. However, it does not work for convolutional layers, since each weight in a CNN is responsible for multiple inputs [Hea15, pp. 175 – 176]. [Hin+12]

The dropping of units effectively reduces the linear activation (i.e. the layer’s activation before the activation function is applied) of the dropout layer. This is of course is not desirable during inference, where every unit should contribute to the output. To counteract this increase in the summed up linear activation during inference, the layer’s linear output is scaled down by a factor proportional to the percentage of dropout. [Hin+12]
2.6 Recurrent Neural Networks

This subsection introduces recurrent neural networks (RNNs) in general and presents some variations. But first, it is worth mentioning that the term RNN is used ambiguous. For once, it is an umbrella term used to describe any NN that contains some form of recurrence; like the loop shown on the left side of figure 5. This loop allows the neural network \( A \) to pass on information it gathered at time-step \( t \) to the next step \( t+1 \). RNN is also used to describe a type of cell that is used in recurrent layers. The term cell is used to differentiate it from more primitive elements, like the units of a dense layer.

\[
A \cdot h_T = A \cdot h_1 \cdot x_1 \cdot A \cdot h_2 \cdot x_2 \cdot A \cdot h_3 \cdot x_3 \cdots
\]

Figure 5: A RNN contains a loop that allows access to information from previous time-steps. The loop can be realized by chaining multiple copies of a feed-forward network \( A \) together.

Unlike image classification which can be done on a single picture, speech is a sequential problem. Understanding a standalone word can sometimes be difficult, while the same word is almost always understood if it is used in a sentence, and therefore has context. This context can be provided by RNNs, because they are able to store and access information extracted at earlier time-steps.

Many variations of RNN cells are in use today. To name a few, beginning with the most basic to the more complex ones: RNN, GRU, and LSTM. While the increased complexity of the LSTM makes them generally better at handling complex and long sequences, this comes at a cost. As they need to compute and store multiple inner states for every time-step. The following subsections present only one version for each of the cells, while there are numerous variants. [Joz+15] is a study that compares several variations and presents their advantages and disadvantages. [Han+14]

The remaining figures in this subsection use the notations shown in figure 6, in addition to the already established notation (section 2). Where yellow boxes indicate a NN layer, with the used activation function stated within. Red ellipsis indicate a componentwise operation, e.g. addition or the application of some function. Arrows indicate vectors. With splitting and merging arrows representing the duplication of a vector and the concatenation of two vectors, respectively.

Figure 6: Notation for recurrent neural network (RNN) illustrations.
2.6.1 Recurrent Neural Network Cell

The recurrent neural network (RNN) cell is a relatively simple element that can utilize learned context from previous time-steps. Its output is calculated as follows:

\[
h_t = g(W x_t + U h_{t-1} + b)
\]  

(7)

where \(x_t\) is the input to the RNN layer \(h\) at time \(t\). \(W\) represents the cells input weight matrix. \(U\) is the cells hidden state weight matrix, it influences which parts of the previous hidden state (the context) \(h_{t-1}\) are relevant for the current state \(h_t\). The initial hidden state \(h_0\) can be initialized with either a constant value or random values. And lastly, the bias vector is denoted with \(b\). Common activation functions \(g\) for RNNs are the hyperbolic tangent or the ReLU function. Figure 7 shows a RNN cell using its hidden state \(h_{t-1}\) and the current input \(x_t\) in order to predict the output \(h_t\) at time \(t\). [Han+14]

![Diagram of an unrolled simple recurrent neural network (RNN) cell for the time-steps \(t-1\), \(t\) and \(t+1\).](image)

Figure 7: Display of an unrolled simple recurrent neural network (RNN) cell for the time-steps \(t - 1\), \(t\) and \(t + 1\).

The simple structure of the RNN cell make them comparatively fast to compute. They also do not require much memory. However, they suffer from both the exploding and vanishing gradient problem. During the backpropagation process a small gradient can still train the last layers of a DNN; while it then vanishes completely after a few layers preventing the earlier layers of the DNN from learning anything. This is a problem in deep networks and therefore RNNs. Because as they are unrolled over several time-steps, they become deep networks themselves. On the other hand, the exploding gradient problem describes the accumulation of errors during gradient calculation. Resulting in ever growing gradients, and therefore unreasonably large weight updates. [Joz+15]

To handle these problems most DNNs use more complex versions of the RNN, like the LSTM. Besides careful initialization of the RNN’s parameters and normalization of the inputs, there are not many options to reduce the likelihood of encountering vanishing gradients in basic RNNs. Some additional measurements that can prevent exploding gradients are, the use of decaying weights, gradient normalization and gradient clipping, to prevent them from becoming too large. [Joz+15]

2.6.2 Long Short-Term Memory

The long short-term memory (LSTM) [Hoc+97] is a version of the RNN introduced by Hochreiter et al. in 1997. The LSTM architecture enables the cell to decide which parts of its learned context to keep, while also allowing it to discard information that it deems as no longer necessary. This active context management improves the LSTM’s ability to learn long term context. It also
helps with the vanishing gradient problem. Consequently, the computational costs and required memory are increased. [Hoc+97]

Instead of passing on a single information vector $h_t$ to the next state, LSTMs use an additional cell-state vector $C_t$. The cell itself now contains four internal NN layers, instead of the single one used in the basic RNN. They are designed to manage the information the cell remembers, forgets and uses to predict its output $h_t$. [Hoc+97]

\[
\begin{align*}
A & \downarrow h_{t+1} \\
C_{t+1} & \downarrow \sigma \\
C_t & \downarrow \tanh \\
h_t & \downarrow A
\end{align*}
\]

Figure 8: Interactions of a long short-term memory (LSTM) cell over consecutive time-steps. The view into the center cell illustrates the internal connections and the gates that govern the information flow.

Figure 8 shows the interactions of a LSTM cell at consecutive time-steps. It uses the notation from figure 6. The primary element of a LSTM cell is the cell-state, the top vector from $C_{t-1}$ to $C_t$ in figure 8. It has only minimal interactions and allows context to be transported from one time-step to the next. The elements below the cell-state vector allow the LSTM to remove (forget) and add (learn) information from and to the cell-state. LSTMs use three gates to manage the cell-state and output. The gates are the sigmoid activated NNs. With the hyperbolic tangent activated NN providing the input gate with new hypotheses. The gates result vectors are componentwise multiplied with other vectors. Since the codomain of the sigmoid function is between 0 and 1, the gates regulate how much and which parts of the other vector are used. [Hoc+97]

For the remainder of this subsection $W_j$ and $U_j$ are input and hidden weight matrices of the respected gate $j$, furthermore $b_j$ represents the gate’s bias vector. The leftmost gate is the forget gate ($f_t$). It considers a new input $x_t$ and the previous hidden-state $h_{t-1}$. Based on this information it determines which parts of the cell-state $C_{t-1}$ to forget and which to keep. In the context of language processing, the cell-state could contain the character sequence of a word. When a new word starts this part of the context can be forgotten, so that the sequence for the new word can be stored. The forget gate vector $f_t$ is given by:

\[
f_t = \sigma (U_f h_{t-1} + W_f x_t + b_f)
\]

The second gate is the input gate. It consists of two NNs. A sigmoid layer ($i_t$) that determines, based on the hidden state $h_{t-1}$ and the current input $x_t$, which parts of a new candidate $\tilde{C}_t$ are added to the cell-state $C_t$. Candidates are produced by an hyperbolic tangent activated NN layer ($\tanh$). The new candidate is also derived from the hidden state $h_{t-1}$ and the current input
The input gate vector $i_t$ and the candidate $\tilde{C}_t$ are calculated as follows:

$$i_t = \sigma(U_i h_{t-1} + W_i x_t + b_i)$$

$$\tilde{C}_t = \tanh(U_{\tilde{C}} h_{t-1} + W_{\tilde{C}} x_t + b_{\tilde{C}})$$

With the first two gates the old cell-state $C_{t-1}$ can be updated. This is done by multiplying the old cell-state with the forget gate vector $f_t$, in order to forget the old context that is no longer required. And then adding the new and scaled candidates $i_t \otimes \tilde{C}_t$ to it:

$$C_t = f_t \otimes C_{t-1} + i_t \otimes \tilde{C}_t$$

The last step is the calculation of the new hidden-state $h_t$, which despite its name is also the cell’s output. It is controlled by the output gate ($o_t$). A sigmoid activated NN that determines which parts of the updated cell-state $C_t$ are relevant for the output, based on the current input $x_t$ and the old hidden-state $h_{t-1}$. To squash the values of the cell-state $C_t$ into the range between $-1$ and $+1$, the hyperbolic tangent is applied. Following this, the cell-state is regulated by componentwise multiplying it with the output gate vector $o_t$. The output of the LSTM cell $h_t$ is given by:

$$o_t = \sigma(U_o h_{t-1} + W_o x_t + b_t)$$

$$h_t = o_t \odot \tanh(C_t)$$

All calculations required for the LSTM cell are summarized in equation (8) [Hoc+97].

$$f_t = \sigma(U_f h_{t-1} + W_f x_t + b_f)$$

$$i_t = \sigma(U_i h_{t-1} + W_i x_t + b_i)$$

$$\tilde{C}_t = \tanh(U_{\tilde{C}} h_{t-1} + W_{\tilde{C}} x_t + b_{\tilde{C}})$$

$$C_t = f_t \otimes C_{t-1} + i_t \otimes \tilde{C}_t$$

$$o_t = \sigma(U_o h_{t-1} + W_o x_t + b_t)$$

$$h_t = o_t \odot \tanh(C_t)$$

2.6.3 Gated Recurrent Unit

Cho et al. introduced a simpler variation of the LSTM called gated recurrent unit (GRU) [Cho+14]. It reduces the computational complexity of the LSTM while still achieving comparable results [Amo+15]. The GRU merges the cell’s hidden-state $h_t$ and the cell-state $C_t$. It further has only two regulating gates, a reset gate and an update gate. The reset gate ($r_t$) regulates how much information from the previous hidden-state $h_{t-1}$ is used to predict new candidates. While the update gate ($z_t$) determines which parts and in what proportions, the cell’s previous hidden-state $h_{t-1}$ and the candidates for the current time-step $\tilde{h}_t$ make up of the output $h_t$. This mechanic gives the cell the ability to ignore inputs $x_t$ and base the output completely on its hidden-state $h_{t-1}$. Intermediate states and output $h_t$ of a GRU are computed as follows:

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z)$$

$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r)$$

$$\tilde{h}_t = \tanh(W_h x_t + U_h (r_t \otimes h_{t-1}) + b_h)$$

$$h_t = (1 - z_t) \otimes h_{t-1} + z_t \otimes \tilde{h}_t$$

where $\mathbf{1}$ is a vector containing only ones, $z_t$ and $r_t$ are the update and reset gate vectors, respectively. The selected candidates are in the $\tilde{h}_t$ vector. Finally, the cells output at time-step $t$ is $h_t$. Figure 9 illustrates a GRU with its reset and update gate. [Hea15, pp. 252 – 253; Amo+15]
Figure 9: Display of a gated recurrent unit (GRU) for consecutive time-steps. The view into the center cell illustrates the interactions of update gate and reset gate with the cells input $x_t$ and hidden-state $h_{t-1}$.

2.6.4 Bidirectional-Recurrent Neural Network

Bidirectional-recurrent neural networks are RNNs with the ability to access context from both the past and the future [Gra+06]. This is achieved by having two separate layers of RNN cells. A forward network layer $\overrightarrow{h_t}$ using some RNN cell. It receives the inputs in the usual order, that is from the beginning to the end $[x_1, \cdots, x_T]$. And a backward network layer $\overleftarrow{h_t}$, that receives the inputs in reverse order, that is $[x_T, x_{T-1}, \cdots, x_2, x_1]$. The outputs from a bidirectional-RNN are often concatenated, as it is the case in this thesis.

The extended context provided by bidirectional-RNNs has shown to be beneficial for sequential systems [Gra+06]. However, bidirectional-RNN layer can begin their computation only after the last input $x_T$ is available, while unidirectional-RNN can start immediately. This is something to consider when developing an interactive ASR system.
2.7 Connectionist Temporal Classification

This subsection covers the connectionist temporal classification (CTC) algorithm. After motivating the algorithm, the subsection is split up into multiple parts that develop the final CTC function. It starts with details on input-sequence to output-sequence alignment. This is then expanded on with a special graph structure that is used to find a set containing all valid alignments. The structure can then be utilized to design an intermediate cost function, which is intended to motivate the final algorithm. This intermediate function is then further upgraded into the final CTC cost function. It provides a differentiable error metric that is required by the backpropagation algorithm to train the RNN on unaligned sequences. The results of a RNN trained with the CTC cost function can be decoded with a modified version of the beam search algorithm. The subsection closes by listing a few relevant CTC implementations.

Unaligned (or unsegmented) data is common in tasks such as handwriting recognition, lipreading, action labeling in videos and speech recognition [Gra+06]. Commonly the input-sequence and the corresponding label sequence are known in these situations, but the exact alignment between them is not. Finding this alignment computationally poses a challenge in ASR [Han+14]. For a RNN to directly work on unaligned sequence data, a new cost function is required. The CTC algorithm [Gra+06] has been introduced by Graves et al. in 2006, with the intention to apply RNNs directly to sequence labeling. CTC provides a cost function for training and a decoding algorithm for inference. The cost function’s objective is to help train a RNN, so that it can predict the most likely output-sequence, for a given input-sequence. While the decoding algorithm infers the most likely transcription from the outputs of a CTC trained RNN. [Gra+06]

2.7.1 Alignments

In speech recognition the input (i.e. audio) is a signal over time and the desired output a sequence of characters, the transcription. A softmax activated RNN receives a sequence of input feature vectors $X = [x_1, \ldots, x_T]$ with length $|X| = T$ and produces a sequence of probability distributions over the target alphabet, with the same length. However, the desired transcription $Y$ that the RNN should learn, has the length $|Y| = U$. Given small enough steps during feature extraction, the number of input vectors is usually larger than the number of characters in the label $T > U$. This is also a prerequisite for the CTC algorithm, as it only works for labels that are shorter or of equal length than the input-sequence $U \leq T$. [Gra+14]

In order to train a RNN the cost function needs to evaluate how well the networks output matches the desired target. To determine this, it needs to know what the desired target at each time-step is. However, this information is not available with the free corpora used. Because they only contain unaligned audio-transcription tuples, and therefore no timing information. This prevents the straightforward application of traditional cost functions (e.g. mean squared error or cross-entropy) for training. To use them, the desired label needs to be aligned to the input sequence. [Han+14]

Therefore, a way to align an input-sequence of length $T$ to a character sequence of length $U$ needs to be found. It is possible to align both sequences by expanding the transcription. For example by duplication every character until it has the same length as the input-sequence. A more sophisticated heuristic can further take the average time it takes to utter each character into account and duplicate the characters based on that information. This is done in figure 10 where the sequence $[T, O, D, O]$ is expanded to the input length of 8, resulting in the sequence...
Which in turn collapses back to the desired sequence \([T, O, D, O]\), by removing all but one of each repeating character. However, the approach is prone to errors, as a single falsely classified input \(x_t\) could easily add a new character to the sequence. Even when the characters are duplicated based on their average phonetic length, this approach is not ideal, as it does not account for rate of speech or breaks in between words. [Gra+06; Han17]

Such label stretching approaches can even be detrimental to the training of a RNN. Consider a question answering system and the input-sequence \([\text{color}, \text{of}, \text{the}, \text{ocean}]\). Training the RNN that the output should be blue at all time-steps \([\text{blue}, \text{blue}, \text{blue}, \text{blue}]\) is not helpful, as the correct answer is greatly depended on the last element of the input-sequence.

Figure 10 illustrates a sequence of probability distributions over the exemplary target alphabet \(\{B, C, D, O, T, U\}\). The sequence is produced by a bidirectional-RNN with softmax activation. For each of the 8 input vectors \(x_t\) a probability distribution is produced. Within the figure, \(y_t^A\) represents the probability of character \(A\) at time-step \(t\), given the input-sequence \(X\).

To enable a RNN to output a character prediction at every time-step, even if the sequence of inputs is longer than the target sequence, CTC introduces a new symbol into the target alphabet. This new symbol is called the blank-label, it is denoted with \(\epsilon\). It allows the network to predict a regular character only if it is confident and output the blank-label otherwise. This leads to the prediction of a high probability for the correct character at localized points in time. Whereas a RNN trained with a traditional cost function and a stretched target sequence would produce distributed probabilities, without clear segmentation. Figure 11 shows how the label probabilities for an utterance are aligned to a sequence of phonemes; illustrating both the localized CTC predictions and the classical framewise approach. [Gra+06; Gra+14]

With the addition of the blank-label, the collapsing of sequences is now a two-step process:

1. Remove all but one repeating character.
2. Remove all blank-labels.
This also allows repeated characters in the transcription, which are not possible with the simple character duplication approach. Because the sequence $[H, E, L, L, O]$ would collapse to $[H, E, L, O]$, by removing the repeated characters. While a predicted sequence containing blank-labels in between $[\epsilon, H, \epsilon, \epsilon, E, \epsilon, L, \epsilon, L, \epsilon, O, \epsilon]$ results in $[\epsilon, H, \epsilon, E, \epsilon, L, \epsilon, O, \epsilon]$ after removal of duplicate characters and decodes to $[H, E, L, L, O]$ when the blank-labels are removed. [Gra+14]

Figure 11: Comparison of the predicted probabilities over time of both a CTC trained RNN and a traditional framewise trained RNN. CTC allowing the RNN to output the blank-label (thin dashed line) if it is not confident to which class an input belongs. Only requiring it to output a normal character if it is confident, which leads to strong localized activations. Whereas the framewise classification has to predict a character for every time-step, even if it is not confident. Figure by [Gra+06].

The CTC algorithm can be used without the blank-label, as it represents a special case. That is introduced to handle the complicated alignments from speech to transcription and to allow for repeated characters. A real label for every time-step is desirable in tasks such as action labeling in videos, therefore, the blank-label would not be used there. [Gra+14]

### 2.7.2 Determining Valid Alignments

The expansion of the target transcription via character duplication and the blank-label introduce a new problem. Since the expanded targets contain additional characters, multiple expansions can collapse to the same transcription. Assume an input-sequence with only four time-steps and the desired transcription $[C, A, T]$, then all the following seven expansions are valid alignments:

\[
[\epsilon, C, A, T] \\
[C, \epsilon, A, T] \\
[C, A, \epsilon, T] \\
[C, A, T, \epsilon] \\
[C, C, A, T] \\
[C, A, A, T] \\
[C, A, T, T]
\]

More general, without any repeating characters in the transcription, there are $\binom{T+U}{T-U}$ possible valid alignments for an audio sequence of length $T$ and a transcription of length $U$. [Han17]

---

\[3\] For $T = 100$ and $U = 10$, that is approximately $4.4 \times 10^{21}$. 
To derive all valid alignments some constraints on the decoding are necessary, since exhaustive search over all possible alignments\(^4\) is not an efficient option. Because the target sequence is known during training, the first constraint is to limit the considered characters to only characters in the transcription. Any sequence containing another character has a probability of zero to be valid. This is shown in figure 12, where only the characters that are contained in the target sequence \([T, O, D, O]\) are left. The dashed arrows indicate one of many alignments that can be decoded. This first constraint ensures that the decoded transcription contains only valid symbols, however, the order can still be wrong.

To find all valid alignments, a second constraint is needed. It expands the previous structure, as illustrated in figure 13. With the rows reordered and duplicated, so that all characters of the target sequence are represented from top to bottom, as indicated on the left side of figure 13. If a character occurs multiple times, then the corresponding row is duplicated multiple times; in this example the row for the character “O”. Therefore, the resulting table has as many rows as the target sequence has characters. The red arrows indicate all paths through the nodes, with each one being a valid alignment. A graph that covers all possible valid alignments is created, using the following rules:

- The graph must start at the top-left (i.e. source at \(y^T_1\) node).
- The graph must end at the bottom-right (i.e. sink at \(y^O_8\) node).
- The graph is monotonically descending.

This is shown in figure 13, where the solid arrows represent all valid alignments for the sequence \([T, O, D, O]\) to an input sequence of length \(T = 8\).

\(^4\) For \(T = 100\) and 28 characters in the target alphabet, there are \(2^{100} \approx 5.2 \times 10^{144}\) possible alignments.
This example does not use the blank-label to be more compact. If the blank-label is used, one is inserted before and after each regular character of the target sequence $Y$, with length $|Y| = U$. The resulting sequence is marked with $Z = [\epsilon, y_1, \epsilon, y_2, \epsilon, \cdots, \epsilon, y_U, \epsilon]$ and has the length $|Z| = 2U + 1$. This guarantees that repeated characters are possible. With the rules adjusted for the additional blank-label. A normal character can now be followed by the next normal character in the sequence or the blank-label, as illustrated in figure 14, whereas the blank-label can only be followed by the next character in the sequence. In addition to that, the path can now start at the top most blank-label or at the first regular character. Furthermore, it can end at either the bottom most blank-label or the last character. Figure 14 shows all valid alignments for the target sequence $Z = [\epsilon, T, \epsilon, O, \epsilon, D, \epsilon, O, \epsilon]$ to an input-sequence of length $|X| = 8$.

### 2.7.3 Intermediate Cost Function

At this point valid alignments are introduced, however, they are not unique. The ability to decode the sequence of probability distributions from a RNN into the desired transcription is still missing. More importantly, a training metric that teaches a RNN to produce such alignments is unknown. This subsection presents an intermediate version of the CTC cost function. Which introduces the main concept and is updated in the following section to the final CTC cost function. The blank-label is omitted to reduce the complexity of this example.

Given the table structure from figure 13 and a target sequence without the blank-label. A dynamic programming algorithm (e.g. the Viterbi algorithm [Vit06]) can be used to find the most probable path from source to sink. For each node, the Viterbi algorithm keeps track of the best incoming edge and the score of the best path from the source to that node. Since the blank-label is not used, each node has only two possible predecessors (figure 13). Let $Y$ be the target sequence (e.g. $Y = [T, O, D, O]$) and $Y_s$ be the $s$-th character in that sequence (i.e. $Y_1 = T$). Then the score for a path ending in the node of the $s$-th character, at time $t$ is defined as the score of the best path to that node, times the probability of the node $y_s^Y$ itself:

$$\alpha_t(s) = \begin{cases} 
\alpha_{t-1}(s) \ast y_s^Y & \text{if } \alpha_{t-1}(s) \geq \alpha_{t-1}(s - 1) \\
\alpha_{t-1}(s - 1) \ast y_s^Y & \text{otherwise}
\end{cases}$$  \hspace{1cm} (10)

The score of the source node is initialized with the probability given by the RNN $\alpha_1(1) = y_1^Y$. All other nodes in the first column are initialized to zero $\alpha_1(s) = 0$, with $s \geq 2$, as they are not valid starting nodes. Since the best path has to end in the bottom-right node (figure 13), it can be derived by looking up the best incoming edge to that node and follow that path back to the source node. [Han17]

The probability distributions are learned by a RNN in an iterative process. Assume that for the initial iteration they are chosen based on a reasonable heuristic, like the mean phoneme length or better a previously trained ASR system. Now that the best valid alignment is known, the network can be trained to improve the score of this alignment. This can be done using a traditional cost function, because the target character for each time-step can be taken from the best alignment. Let $b_t$ be the character from the best alignment at time-step $t$ and $B_t$ the probability distribution for time $t$, where the symbol from the best path always occurs $B_t(b_t) = 1$. Based on the cross-entropy function $xent$, the divergence of the character $b_t$, with regards to the
Figure 14: RNN predicted probabilities for each time-step. The graph contains all valid alignments for the target sequence $Z = [\epsilon, T, \epsilon, O, \epsilon, D, \epsilon, \epsilon]$, for an input length of 8. The probability of the $s$-th character of sequence $Z$ at time-step $t$ is represented by $z^s_t$. To improve the readability all connections leaving a normal character use rounded red arrows, while connections leaving a blank-label $\epsilon$ use straight blue bold arrows.
predicted probability distribution \( Y_t \) at time-step \( t \) is:

\[
\sum_{t=1}^{T} \text{ent}(Y_t, B_t) = - \sum_{t=1}^{T} \log \left( \frac{Y_t}{b_t} \right) \tag{11}
\]

The resulting cost gradient with respect to the \( t \)-th output probability distribution is a vector with all zeros, except at the component that corresponds to the target symbol in the estimated alignment. It can then be used in the backpropagation algorithm to reduce the cross-entropy costs. [Gra+06; Han17]

The overall process to find and refine alignments is therefore:

1. Randomly initialize alignments, then
2. train the model with the given alignments, then
3. decode with the new model to obtain better alignments, then go back to (2).

### 2.7.4 Cost Function

The intermediate approach strongly depends on the initial alignment chosen by the heuristic. If it is suboptimal, then the cost function is prone to local optima, as it considers only the single best path derived from it. Therefore, the CTC cost function should not commit to any specific alignment. To accomplish this, it has to consider every possible valid alignment, instead of just the best one. This is accomplished by summing over the probabilities of all valid alignments \( A_{X,Z} \) between the input-sequence \( X \) and the blank-label modified target transcription \( Z \). The set of all valid alignments is retrieved from a graph structure like the one in figure 14. With each alignment \( A \in A \) being of length \( |X| = T \), that is \( A = [a_1, a_2, \ldots, a_T] \). With the probability for an alignment \( A \) being the product of the probabilities for each character \( a_t \) of that alignment at time \( t \), given the input sequence \( X \). The CTC objective for an input-sequence \( X \) and its corresponding transcription \( Z \) is:

\[
P(Z|X) = \sum_{A \in A_{X,Z}} \prod_{t=1}^{T} P_t(a_t|X) \tag{12}
\]

[Han17]

However, even computing only the valid alignments can be expensive for long input-sequences (section 2.7.2). To reduce the computational demands Graves et al. introduce a recursive version that is proportional to the metric in equation (12) [Gra+06]. Since all valid alignments start either in the first blank-label or the first valid character, the scores for the first time-step \( \alpha_1(s) \) are initialized as follows:

\[
\begin{align*}
\alpha_1(1) &= z_1^1 \\
\alpha_1(2) &= z_1^2 \\
\alpha_1(s) &= 0, \quad 3 \leq s \leq |Z|
\end{align*}
\tag{13}
\]

Where \( z_t^s \) is the probability of the \( s \)-th character from sequence \( Z \) at time-step \( t \), as illustrated in figure 14. With the recurrent relation given by:

\[
\alpha_t(s) = \begin{cases} 
\bar{\alpha}_t(s) * z_t^s & \text{if } (Z_s = \epsilon) \text{ or } (Z_{s-2} = Z_s) \\
(\bar{\alpha}_t(s) + \alpha_{t-1}(s-2)) * z_t^s & \text{otherwise}
\end{cases}
\tag{14}
\]
with:

\[
\bar{\alpha}_t(s) = \alpha_{t-1}(s-1) + \alpha_{t-1}(s)
\]  

(15)

As illustrated in figure 14, there are two cases to consider for the score \( \alpha_t(s) \):

**Case 1:** If the current symbol is a blank-label \( Z_s = \epsilon \), it can only be reached from the same blank-label at the previous time-step (i.e. \( t - 1 \)) or from its preceding regular character \( Z_{s-1} \) at the previous time-step. Furthermore, to make sure that repeated characters (e.g. \( \epsilon, A, \epsilon, A, \epsilon \)) are kept in the collapsed sequence, the blank-labels between regular characters need to be preserved. This is the case when the current symbol \( Z_s \) is a regular character and the previous regular symbol \( Z_{s-2} \) is the same character \( (Z_{s-2} = Z_s) \). In other words, a jump from the first “A” in the example sequence above to the second one is not allowed, because the blank-label \( Z_{s-1} \) between them needs to be preserved. This ensures that the directly preceding symbol \( Z_{s-1} \) is not skipped in these cases. Because, either the path transitioned directly from it, or the path has already passed through it at an earlier time-step.

In this case the score for the two valid subsequences is:

\[
\alpha_{t-1}(s-1) + \alpha_{t-1}(s)
\]

Case 1 is illustrated in figure 15a \((Z_s = \epsilon)\) and 15b \((Z_{s-2} = Z_s)\). [Han17]

**Case 2:** This case occurs otherwise, for example when the previous symbol is the blank-label \( Z_{s-1} = \epsilon \) and there is no repeating character. In this case the node has three valid predecessors. This allows both a direct transition from the previous normal symbol \( Z_{s-2} \), while skipping the intermediate blank-label and also a transition over the intermediate blank-label.

The score for the three valid subsequences is:

\[
\alpha_{t-1}(s-2) + \alpha_{t-1}(s-1) + \alpha_{t-1}(s)
\]

Case 2 is illustrated in figure 15c. [Han17]

In the last time-step \( T \), all valid alignments end in either the last blank-label \( Z_{|Z|} \) or the last regular character \( Z_{|Z|-1} \) (figure 14). A score proportional to the desired probability from equation (12) can be calculated:

\[
P(Y|X) = \alpha_T(|Z|) + \alpha_T(|Z| - 1)
\]

(16)

The resulting score is the sum of the scores from all paths ending in the last blank-label \( \alpha_T(|Z|) \) and the scores of all paths ending in the last regular character \( \alpha_T(|Z| - 1) \). This score describes how well the probability distributions from the RNN match the target sequence \( Y \) for the given input-sequence \( X \). Since the recursion is prone to floating-point underflow [Gra+06]: instead of maximizing the score directly, the CTC costs minimize the negative log-likelihood:

\[
\mathcal{L}_{CTC}(X,Y) = -log \left( P(Y|X) \right)
\]

(17)

The resulting gradient for an input-sequence \( X \) and its transcription \( Y \) is a vector with all zeros, except at the positions corresponding to symbols that occur in the blank-label modified transcription \( Z \). [Gra+06; Han17]
Figure 15: Illustration of both cases for the recursive CTC algorithm. The red node at \((t, s)\) is being evaluated. The blue nodes connected with arrows are viable predecessors. In case 1, the preceding symbol (i.e. \(s - 1\)) cannot be skipped as they are either a regular character (a) or a blank-label required to encode recurring characters (b). In the other case (c), a valid path is allowed to skip the preceding symbol.

The CTC algorithm is not a robust algorithm, as it still depends on the initial alignment \([Han17]\). However, it allows every valid alignment and does not fixate on the most likely one. The resulting gradient update only affects the weights that contribute to the symbols of the target sequence \(Z\). Therefore, the resulting weight update nudges the characters that occur into the correct direction. It does not affect the ones that are not part of the transcription. This results in very localized and strong activations for the correct symbol, which is illustrated in figure 16. The decision to only consider valid alignments is a trade-off to achieve reasonable computation times. \([Han17]\)

2.7.5 Inference

As seen in figure 10, a trained RNN produces probability distributions over the target alphabet. To infer the most likely transcription \(Y^*\) for a new input-sequence \(X\), these probability distributions have to be decoded. This is done in equation (18), where the variable \(Y\) is taken from a set containing all possible transcriptions. \([Han17]\)

\[
Y^* = \arg\max_Y P(Y|X) \tag{18}
\]

However, a RNN trained with the CTC cost function produces expanded alignments that contain the blank-label and not the already collapsed transcription. The most likely alignment \(A^*\) can be found by solving:

\[
A^* = \arg\max_A \prod_{t=1}^{T} P(a_{i}|X) \quad \text{with} \quad a_t \in A \tag{19}
\]

Where \(a_t\) is a character of a considered alignment \(A \in \mathcal{A}\) is from the set of all possible alignments. The desired transcription can be obtained by collapsing the alignment-sequence \(A^*\). \([Han17]\)

This approach works, but it does not consider the combined score of all the alignments that collapse to the same transcription. There are cases in which the single most probable alignment
Figure 16: On the left side are output activations for the same input-sequence, at different stages of training. With solid lines corresponding to normal symbols and the dashed line to the blank-label. On the right side are the corresponding costs, with values above the horizontal axis acting to increase the output activation and values below acting to decrease the activation. (a) shows the initial network with random weights. (b) shows an intermediate state at which the network begins to predict symbols and the error localizes around them. And (c) shows a trained model, with strong and correct predictions at localized points in time and nearly no error. [Gra+06]

Figure by [Gra+06]

has a higher score than all other alignments, however, the combined scores from alignments for another transcription is greater. For example, the score for \([A, A, A]\) is greater than the scores for \([B, B, B]\) and \([B, B, \epsilon]\). Since the last two alignments both collapse to the same transcription (i.e. \([B]\)), their combined score could be greater. In that case the output should be \([B]\) instead of \([A]\). [Han17]

This problem can be solved though a version of the beam search algorithm that is modified for CTC decoding. Figure 17 illustrates the modified beam search algorithm on the target alphabet \(\{A, B, \epsilon\}\). While the vanilla beam search produces a new set of transcription hypotheses at each time-step; by extending the hypotheses from the previous step, with all possible output characters and keeping only the most likely alignments. The modified version can handle multiple alignments that correspond to the same transcription. To accomplish this, each node stores the collapsed hypothesis (the blue boxes in figure 17) from the paths that reached it. This means that different alignment paths that collapse to the same transcription are merged together. This can be seen in figure 17 at (*) \(T = 2\), where the alignments \([\epsilon, A]\), \([A, A]\) and \([A, \epsilon]\) all collapse to \([A]\) at the next time-step. Also at (**) \(T = 3\), where the sequences \([B, A, \epsilon]\) and \([B, A, A]\) both collapse to \([B, A]\). [Han17]

In addition to the collapsed sequence for each node, the algorithm has to keep track of two scores. The combined score of all normal character nodes that reached it and the combined score of all blank-label nodes that reached it. This allows paths to split, if a node is reached from both a normal character node and a blank-label node, which allows repeating characters. This is illustrated in figure 17 at (+) \(T = 3\), where one of the paths considers a character that collapses (i.e. \(\epsilon\)) and the other a repeated character (i.e. \(A\)) that is then separated by a blank-label. A path splits if a node is entered by a character and a blank-label and left by a character that matches the nodes last character.
Consider the example case of the $[A]$-node at $(+)$ $T = 3$ in figure 17. It contains the collapsed transcription $[A]$ and has both a character node and a blank-label node entering it. The important exit path is the one that appends another “$A$”. This opens up two possible transcriptions to consider. If the entering node is a blank-label, the expanded alignment in that node already ends with $[\cdots, A, \epsilon]$. Therefore, appending another “$A$” results in the sequence $[\cdots, A, \epsilon, A]$ and collapses to $[A, A]$. The new score at the next time-step ($T = 4$) is then only affected by the blank-label score entering the node at $T = 3$. Else if the entering path is the character “$A$”, then the expanded sequence after adding an additional “$A$” is $[\cdots, A, A]$, which collapses to $[A]$. In this case the new score only considers the normal character score entering. Note that the score for any non-splitting node in figure 17 is the combined normal character and blank-label score. [Han17]

Figure 17: Example for the modified version of beam search algorithm that is designed to decode CTC alignments. The beam width is 3 and the target alphabet is $\{A, B, \epsilon\}$. $[\varnothing]$ represents an empty transcription. The blue boxes contain the hypothesized collapsed transcriptions. They additionally store the scores from blank-label nodes and normal character nodes reaching them. While the circles connected with solid lines represent possible extensions for that transcription. Purple circles represent likely extensions that are followed by a beam and empty circles are extensions that are unlikely and get pruned.

(*) and (**) highlight cases where multiple different paths collapse to the same transcription. (+) highlights a case where a node splits into two possible transcriptions, because a repeated character is possible.

Figure derived from [Han17].
2.7.6 Connectionist Temporal Classification Implementations

There are not many CTC implementations that can be used in a specific framework. As Hannun, one of the authors of Deep Speech [Han+14; Amo+15] states: this could be in part because even with a solid understanding of the algorithm, the implementation is difficult, due to several edge cases. He further advises that, because of the complexity of the algorithm, implementations should be written in a suitable language and require a high level of optimization. Both a cost function and a decoder implementation are required for training. [Han17]

To name a few notable implementations:

- Google TensorFlow has CPU implementations of both the CTC cost function and the CTC beam search algorithm\(^5\).

- Baidu Research has an open source CTC cost implementation named WarpCTC\(^6\). It was written for “Deep Speech 2: End-to-End Speech Recognition in English and Mandarin”. WarpCTC is written in C++ and compute unified device architecture (CUDA) and can be used on CPU or GPU. [Amo+15]

- Nvidia’s CTC cost implementation is included in the cuDNN library since version 7 [Han17].

2.8 Summary

This section introduced the NN building blocks that can be used in a state-of-the-art ASR system. Beginning with the representation of the input and output data. Presenting the classical unidirectional NN layers, namely dense and convolutional layers. In a dense layer each of its units is connected with every input unit, therefore, it has an individual weight for each connection. The convolutional layer instead uses a local receptive field to extract information. Therefore, a CNN requires less weight parameters than a dense NN. It learns what type of feature to extract during training and the information extracted by a CNN contains some spatial context.

The section further introduced recurrent layers and some different cells they use. RNNs have the ability to remember and access a context to classify input-sequences. They are therefore no longer relying solely on the current input for classification. The basic RNN cell and its more complex derivatives LSTM and GRU are explained. The later ones being better suited for long input-sequences, because the can actively manage the stored context, at the price of requiring more resources to compute.

Section 2.7 introduced the CTC algorithm. A cost function that can be used to train a RNN on unsegmented audio-transcription tuples. An introduced blank-label gives the RNN the freedom to output a regular character at localized points in time, only if it is certain, otherwise the blank-label is produced. The costs of such an input example can be calculated by using a recurrent algorithm to approximate the probability of all valid alignments between a transcription and an input-sequence. The recurrent algorithm uses a special graph structure that can be used to efficiently calculate the costs. To infer the desired transcription for a new input, from the outputs of a trained RNN, a modified beam search algorithm is used. Instead of only evaluating a given number of best paths, the modified version keeps track of alignments that collapse to the same transcription and merges them. Therefore, it considers the probability of all alignments that would collapse to the same transcription, instead of only the most likely alignment.

\(^5\) TensorFlow CTC: https://www.tensorflow.org/api_guides/python/nn
\(^6\) WarpCTC: https://github.com/baidu-research/warp-ctc
3 Related Work

This section presents the progress ASR systems have made over the decades; from the early beginnings, up until state-of-the-art DNNs emerged. Besides the historical approaches in section 3.1, there are DNNs that proofed to be successful in transcribing speech. These are the end-to-end trained DNNs. They have noticeably reduced the complexity of the overall ASR system. In this thesis, an end-to-end trained model refers to a single deep RNN architecture that receives audio features as input and transforms them into the matching transcription. It is trained as a whole, as opposed to its individual components (i.e. AM, PM and LM) being trained independent of each other. In other words, a single NN that replaces all the components within the traditional speech recognition pipeline. The end-to-end trainable models can broadly be categorized into CTC (section 3.2) and encoder-decoder (section 3.3) based systems. Since end-to-end trained systems do not rely on carefully designed pipelines and their components; they can be applied to a variety of speech, such as noisy environments and accented speech, by simply changing out the training material [Amo+15].

3.1 History of Speech Recognition Technologies

This section briefly summarizes the early history of speech recognition technologies, up onto the point when end-to-end trained systems began to outperform the classical approaches. [Den+18, pp. 2–17] provides a very in depth account of the milestones ASR systems have reached over the years.

Because of their potential, researchers were interested in ASR systems since many decades; they have undergone many iterations. With one of the earliest works being Audrey from Bell Laboratories in 1952. Audrey could classify the digits from 0 to 9 with a stated accuracy of 97%. However, it only worked for a unique speaker that it has been adjusted to. [Dav52]

IBM presented their Shoebox system at the Seattle World’s Fair in 1962. It could recognize and respond to sixteen spoken words, including the ten digits and perform simple arithmetic on them. The utterance: “one <pause> plus <pause> two <pause> total” should result in Shoebox printing out the number three. [Der+62]

In the early 1970s, the U.S. Department of Defense’s Defense Advanced Research Projects Agency (DARPA) initiated the Speech Understanding Research program. The Harpy Speech Understanding System, with the capability to understand 1,011 words was one of the results from this program. It was one of the first speech recognition systems that made use of the beam search algorithm. Indicating that even a comparably simple acoustic model (AM) combined with an efficient search algorithm can outperform the carefully handcrafted AMs of that time. [Wai+90]

The TIMIT Acoustic-Phonetic Continuous Speech Corpus is also a product of the DARPA project. Containing a total of 5.4 hours of speech, it was one of the first large speech corpora. It consists of utterances from 630 American English speakers of various dialects, each recording ten sentences. Every recording is manually segmented and annotated with exact timings for each individual phoneme and word. While early speech recognition systems of that decade were based on handcrafted rules, provided by linguists who relied on domain specific knowledge. The emergence of larger training corpora allowed machine learning algorithms to flourish. [Gar+93]

Beginning in the 1980s, researches started to introduce models based on empirically derived rules. The introduction of the hidden markov model (HMM), and even (shallow) NNs were responsible for this shift. The emerging new models are still comprised of several interconnected
submodules, within a traditional recognition pipeline (figure 1) [Yu+14, pp. 4 sq.]. However, the empirical approaches do not depend as heavily on handcrafted rules, instead they are able to exploit the growing amount of training data available. These models require assumptions about the underlying probability distributions of the training data [Gra+06]. Due to their ability to be trained from large corpora, they quickly became the de facto standard for speech recognition. The larger corpora also increased the amount of recognizable words to several thousand. [Den+18, pp. 4–7]

The above systems have one thing in common, they require discrete inputs. Speakers have to insert a pause between each word; making speech recognition applications cumbersome to use. To expand the fields of application, the systems have to work on continuous speech. Over the years, HMM based systems have improved regarding continuous speech and speaker independence [Sao+12]. [Den+18, pp. 4–7]

With the release of the pioneering “Learning Deep Architectures for AI” by Bengio in 2009, advocating for the use of deep architectures. The focus shifted away from HMMs and towards DNNs [Ben09]. Following this, DNNs gained popularity in several areas, like vision and language. Over the following years, they were adopted as a replacement for some components in the speech recognition pipeline. For example, they were successfully used to classify individual phonemes. [Den+18, pp. 6–11]

While these approaches have proved successful for many problems, they have several drawbacks: (1) they usually require a significant amount of task specific knowledge, e.g. to design the state models for HMMs, or choose the input features […]; (2) they require explicit (and often questionable) dependency assumptions to make inference tractable, e.g. the assumption that observations are independent for HMMs; (3) for standard HMMs, training is generative, even though sequence labeling is discriminative. [Gra+06]

In 2008, Google introduced voice recognition to its Voice Search app for the iPhone, able to recognize spoken search queries [Mar08]. Three years later, with the personal assistant Siri, Apple expanded on that. Bringing a mobile ASR system into the everyday life [Daw08]. In 2012, Google adopted DNNs as the core technology for their AM in Google Voice Search [Sak+15]. In cooperation with the University of Toronto, they were also among the first to introduce an end-to-end trained ASR system in 2014 [Gra+14]. It is based on the CTC algorithm, derived by Graves et al. in 2006 [Gra+06].

At that time, the available algorithms, corpora, and hardware allowed for various different end-to-end trained, speech recognition systems to be developed [Mia+15; Kim+17; Cha+16]. Resulting in DNNs to achieve better recognition rates, while also improving the system’s ability to abstract [Gra+14]. The recent advances enabled individuals to develop their own speech recognition systems; breaking the oligopoly of information technology companies and governments. The currently used methods for end-to-end DNN based ASR systems are presented in the two following subsections.
3.2 Connectionist Temporal Classification Models

Connectionist temporal classification (CTC) is one of the corner stones of the early end-to-end
trained ASR systems publicized in 2014 [Gra+14; Han+14]. As mentioned in section 2.7, CTC
enables RNNs to train on sequential data without requiring knowledge about the alignment
between audio and transcription.

Graves et al. published a CTC based model [Gra+14] that can be trained directly from audio-
transcription tuples. It does not require intermediate representation forms (e.g. phonemes or
graphemes). The model’s architecture is based on five bidirectional-LSTM RNN layers. With
the CTC cost function providing the error for the backpropagation algorithm. They modified
the CTC function to also includes a word error rate (WER) based term, to regulate the costs
in relation to the transcription length. This enabled them to optimize the NN, based on the
metric that is important for speech recognition (i.e. WER), instead of an intermediate metric
(e.g. CTC). They estimate the expected transcription cost via Monte-Carlo sampling. The model
is trained on the 81 hour long Wall Street Journal (WSJ1) corpus, using spectrograms (without
frequency rescaling) with 128 feature elements as input. They achieved a WER of 27.3%; where
a lower WER value is better. With the addition of an external lexicon, therefore, no longer being
a true end-to-end trained system, they report a WER of 21.9%. The addition of a trigram LM
further reduced the WER down to 8.2%. [Gra+14]

Hannun et al. developed the Deep Speech (1) speech recognition model [Han+14]. It consists of
three dense layers, followed by a single bidirectional-RNN layer, followed by two dense layers.
They use the standard CTC algorithm ([Gra+06]) as cost function. The simple design, with only
a single recurrent layer, allows for fast training runs. Their model is trained on a combination of
about 7,380 hours of read and conversational speech from both commercial and internal Baidu
corpora. Input features are logarithmically scaled filter banks8, with up to 160 data points. A
5-gram LM trained on 220 million phrases from CommonCrawl9 is used. They report a WER of
16.0%, measured on the entire Switchboard HUB5 test dataset. The result is from a committee
comprised of six of the described models. [Han+14]

With Deep Speech 2, Amodei et al. extended the Deep Speech (1) model [Amo+15]. They
focused on decreasing the overall training time by utilizing high performance computing (HPC)
techniques and implementing the algorithms optimized for GPGPU (section 2.7.6). This enabled
them to evaluate several changes to the Deep Speech (1) model. Frequency convolutions (1D
convolution), time-frequency convolutions (2D convolution), a varying number of recurrent layers,
different types of recurrent layer cells (i.e. RNN, GRU, LSTM) and batch normalization, to name
but a few. Deep Speech 2 is trained using a total of 11,940 hours of speech. With 2,380 hours
from commercial corpora, 8,600 hours of Baidu internal data and 960 hours from the open
LibriSpeech ASR corpus11. For their internal validation dataset, they report a WER of 14.9%
for their strongest non LM system and 9.5% for the same system using a 5-gram LM. Their best
performing system achieves a WER of 7.9% on the test dataset. [Amo+15]

---

8 Wall Street Journal: https://catalog.ldc.upenn.edu/LDC94S13B
9 It is not stated how these filter banks are constructed. But they do not use MELSPEC or MFCC.
10 CommonCrawl: http://commoncrawl.org
11 Switchboard: https://catalog.ldc.upenn.edu/LDC2002S23
12 LibriSpeech: [Pan+15]
3.3 Encoder-Decoder Based Models

“At the same time, a new direction of neural network research has emerged that deals with models that learn to focus their attention to specific parts of their input. [Bah+16]” Attention based models gained popularity as an alternative to CTC. These models use the encoder-decoder paradigm that is commonly combined with an attention mechanism [Cha+16; Bah+16]. It merges AM, PM and LM into a single neural network, where CTC often requires an external LM [Chi+17].

Encoder-decoder systems originate from the sequence-to-sequence (seq2seq) models used in neural machine translation. In ASR the encoder (usually a RNN) reads the whole input-sequence \( \{x_t, \text{where } t \in \{1, \cdots, T\} \} \) and its final layer produces a fixed length, high level representation \( h \) for it. The decoder then creates the entire output (e.g. the transcription) based on this representation. Figure 18 shows two examples where a seq2seq system can be used. [Bah+16]

![Encoder-Decoder System Diagram]

Figure 18: Two examples where encoder-decoder, sequence-to-sequence (seq2seq) systems can be used. It shows how the intermediate state \( h \) can be generated from various types of input data. This intermediate can then be decoded into different types of output data. The order and length of the output does not necessarily have to align to that of the input. As can be seen in the differently structured sentences in English and the German.

Attention mechanisms improve the encoder-decoder architecture by presenting the decoder only a small subset of the total encoding at a time. Ideally, this set contains only the relevant input frames, as well as the required context to decode a target character or word. Which parts to select and present is learned by the attention mechanism, as illustrated in figure 19. This enables the decoder to transcribe the subset and use the gained knowledge together with the next subset to transcribe it. Attention based networks operate close to the way humans would solve such a task. Instead of listening to the whole audio sequence, memorizing it and only after that, trying to understand it as a whole. The attention network continuously adds new bits of information, which also depend on the previously understood parts of the utterance. [Cha+16; Bah+16]

Chan et al. present Listen, Attend and Spell (LAS), an end-to-end neural speech recognizer. Where the listener acts as AM encoder, to convert an utterance \( x = [x_1, \cdots, x_T] \) into a high level representation \( h = [h_1, \cdots, h_U] \), with \( U \leq T \). The listener is a pyramid shaped bidirectional-LSTM. Having more units in the lower layers and fewer in the top ones, to reduce the amount of information handed to the speller. The speller is an attention-based character decoder. It converts \( h \) into a probability distribution over character sequences. The listener consists of two LSTM layers followed by dense layers. The model is trained on 2,000 hours of Google voice search utterances. The data was further augmented by adding noise from YouTube recordings.
Figure 19: Alignment between input feature encoding $h$ and the desired character sequence $y$ is learned by an attention mechanism. The counter above each plot is the iteration step counter of the NN. With the character hypothesis on the y-axis and time on the x-axis. Illustrated are example plots of learned attention alignments. From the very early stages of learning (left), to a clear attention alignment (right) several thousand iterations later.

Attention alignment data provided by Weweler.

and reverberation, using room simulations; increasing the amount of training data by a factor of 20. LAS achieved a WER of 14.1% on a Google voice search task without the use of a dictionary or external LM. 10.3% with the help of an external LM, rescoring the top 32 beam search paths. [Cha+16]

Bahdanau et al. describe another end-to-end trained ASR system that is based on the encoder-decoder paradigm. They improve training on long input-sequences by focusing the attention mechanism on parts of the input that are most promising to be relevant. The model uses four layers of bidirectional-GRUs for the decoder, with the top two layers reading only every second frame. An attention-based recurrent sequence generator (ARSG) is introduced as decoder that produces the output-sequence. The ARSG is a RNN with an additional attention mechanism subnetwork. The attention mechanism selects the temporal locations in the input on which the RNN should focus; while the RNN produces a prediction for the next character. The model is trained on the 81 hours WSJ1 corpus. Using 40 MFCC features combined with energy that is calculated over the MFCCs. These 41 features per window are further extended with their first and second order temporal derivatives, resulting in a total of 123 features per window. The system achieved a WER of 18.6% without a LM and 10.8% with the use of a trigram-LM. [Bah+16]

Kim et al. propose a joined CTC and attention approach for end-to-end speech recognition. In it the CTC algorithm helps the network to initially learn how to align the audio signal to the corresponding transcription and how to handle noise. This in turn enables the attention mechanism to produce better results later on, as opposed to a pure CTC model. The later one can struggle as training progresses, if no external LM is added. This joined model is trained on the 81 hour wall street journal corpus (WSJ1). It uses 40 MFCCs as input features. The model achieves a WER of 18.6%. [Kim+17]
3.4 Summary

This section presents the historical path ASR systems took. Beginning with handcrafted rules that allowed to distinguish between the ten digits. Over HMM based speech recognition pipelines. Later on, HMMs were able to extract large amounts of information from the ever growing amount of available data. From the introduction of NNs as replacement for individual parts of the traditional speech recognition pipeline. Up to the current state-of-the-art end-to-end trained DNNs, which are categorized into CTC and encoder-decoder based approaches. Related work based on these two approaches is presented. With Baidu’s Deep Speech projects being one of the successful end-to-end trained CTC systems.

In closing the main alternative to CTC based systems is described, namely the encoder-decoder paradigm in conjunction with an attention mechanism. It is designed to be able to learn an alignment between input and output. In contrast to CTC systems, an attention based system can learn its own LM.
4 Method

This section characterizes the corpora used for training and the necessary preparations to make them usable. Followed by a description of the performed preprocessing steps. They include the selection of suitable features that can be extracted from audio data and the normalization process of said features. Following is the description of an ASR system that is composed of the components described in section 2. This system serves as a baseline and is used to compare later modification against. The baseline system is then improved on by changing multiple parts of its architecture. For example, exchanging the dense input layers with convolutional layers or replacing the basic RNN cells with more advanced versions. The section closes by detailing the training process, including the used hyperparameters.

4.1 Training Material

The creation and maintenance of the training dataset is one of the most important and time-consuming tasks in the development process of an ASR system. RNNs are able to learn from large amounts of data and even the traditional HMM based ASR systems are mostly data-driven [Gra+14; Moo03]. But how much and what type of data, is required to learn a language? Learning a language is not easy, even for humans. By collecting information from multiple studies, [Moo03] roughly estimates how many hours of speech the average human has been exposed to at different ages:

- **Age 2:** about 1,000 hours (∼6M words per year)
- **Age 10:** about 10,000 hours (∼14M words per year)
- **Age 50:** about 100,000 hours (∼14M words per year)
- **Age 80:** about 150,000 hours (∼14M words per year)

The way children learn to recognize speech is unsupervised learning, whereas this ASR system uses supervised learning.

In a more recent statement [Ng16], Ng suggests the following amounts of transcribed speech, to train an end-to-end speech recognition system. He estimates that for a proof of concept, one needs approximately 300+ hours of speech. Scientific papers require about 3,000+ hours of speech. Whereas commercial systems use from 10,000 hours, up to over 100,000 hours of speech for their training [Amo+15; Han+14; Cha+16]. [Ng16].

Ideally, the training data fits the targeted domain for the ASR system. The following points about the dataset should be taken into consideration: It covers all targeted accents within the language; consider a system trained only with people from Oxford being used by a Scotsman. Ideally, the data is representing of the target demographic. For a general large vocabulary continuous speech recognition (LVCSR) system, this includes male and female speakers, as well as children and elderly speakers. It contains recordings made with a wide range of different microphones, in diverse environments. Lastly the margin of error within the dataset itself should be low enough; this includes faulty labels, unreasonable acoustic distortions or volume levels. The last point is especially important to consider for free and for self-created datasets.
4.1.1 Corpora Selection

Training the ASR system on non-commercial datasets only, is one of the objectives of this thesis (section 1.2). While several free datasets exist, they do not necessarily fulfill all the criteria mentioned above; or the required information is not available. Licensing information for all used datasets can be found in appendix A.1. The following datasets are combined to build the training, validation and testing dataset. They contain mostly read or scripted speech, of varying quality.

The Mozilla Common Voice dataset contains recordings made by volunteers via an online recording tool\textsuperscript{12}. It contains about 350 hours of read speech. It is recorded with all kinds of different microphones, in environments ranging from quiet rooms to noisy shopping malls. Not every contributor to the English Common Voice corpus is a native speaker. The dataset contains voluntary information about the speaker’s gender, age and accent. Hence, this data is not available for most recordings in the dataset. Validation of the recorded utterances is crowdsourced as well. Everyone can listen to examples on the common voice website and vote if the audio matches the stated transcription. [Moz17a]

The LibriSpeech ASR Corpus\textsuperscript{13} contains excerpts from selected audiobook recordings of the LibriVox\textsuperscript{13} project. LibriVox provides a collection of about 12,000 recorded audiobooks from public domain books. The texts are read and recorded by volunteers. Recordings used for the LibriSpeech corpus have been segmented and aligned with text labels. There are about 460 hours of speech in the dataset. Some books used are dated, resulting in old and outdated words and names being used. Furthermore, some of the speakers try to enact Shakespeare. [Pan+15]

The Tatoeba (“for example” in Japanese) corpus contains many read text recordings. There are about 192 hours of speech, recorded by 6 unique speakers. The recording and speaker quality appears to be above average, with the transcriptions being accurate as well. Nevertheless, the corpus has it downsides: (1) The texts often consists of several permutations of the same sentence and (2) the majority of the recordings are from a single speaker. [Ho+06]

The TED-LIUM corpus contains recordings of TED-Talks\textsuperscript{14}. In this thesis TED-LIUM refers to the extended second version of the corpus. It contains about 207 hours of partially scripted speech. Each recording in the dataset represents a complete talk, including audience feedback (e.g. clapping, laughter) and filler words (e.g. “er”, “uh”, “um”). Consequently, the data needs to be transformed in order to be usable for training (section 4.1.2). Validation of random samples shows that the provided transcript with timestamps is not always accurate, especially shorter sequences with five or fewer words are prone to be erroneous. [Rou+14]

Most of the datasets come with predefined train, validation, and test subsets. If no subsets are defined, then the whole dataset is added to the training corpus. If a dataset contains “validated” and “other”\textsuperscript{15} subsets, only the validated ones are used. Table 1 shows the composition of the training dataset. The 4 hours of the LibriSpeech validation set are used as validation dataset.

\textsuperscript{12} Common Voice: https://voice.mozilla.org
\textsuperscript{13} LibriVox: https://librivox.org
\textsuperscript{14} TED: https://www.ted.com
\textsuperscript{15} Contains examples that could be flawed or have faulty labels.
Because a validation run is started after every training epoch and takes over an hour to complete, the validation set is kept small. Testing is only performed after the training finished, where performance is less of an issue. Therefore, the test dataset is composed of the LibriSpeech and Common Voice test sets. The combined test dataset contains 6 hours of speech. LibriSpeech being the base of the test dataset, while Common Voice adds examples with high variance. This allows testing of the model’s ability to generalize and handle noisy environments.

<table>
<thead>
<tr>
<th>Name</th>
<th>Male/Female</th>
<th>Mean Length</th>
<th>Examples</th>
<th>Used†</th>
<th>Total†</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common Voice</td>
<td>2.90</td>
<td>3.99s</td>
<td>52,590</td>
<td>58h</td>
<td>350h</td>
</tr>
<tr>
<td>LibriSpeech</td>
<td>1.08</td>
<td>12.59s</td>
<td>132,320</td>
<td>463h</td>
<td>464h</td>
</tr>
<tr>
<td>Tatoeba</td>
<td>—</td>
<td>2.32s</td>
<td>298,709</td>
<td>192h</td>
<td>192h</td>
</tr>
<tr>
<td>TED-LIUM</td>
<td>2.14</td>
<td>8.03s</td>
<td>83,539</td>
<td>186h</td>
<td>207h</td>
</tr>
<tr>
<td>Combined Dataset</td>
<td>1.91</td>
<td>5.69s</td>
<td>567,158</td>
<td>901h</td>
<td>–</td>
</tr>
</tbody>
</table>

* Hours used and hours total are rounded down to the next full hour.

Table 1: Distribution of the datasets used to create the combined training dataset. Male to female ratio for Tatoeba is not available; a strong male representation can be assumed. The mean length per example is stated in seconds. Examples is the number of individual audio files in that corpus. Used lists the number of hours that are used from that dataset. While total states the number of hours in the complete dataset.

### 4.1.2 Preparations

The following operations are performed before training starts. Every audio example in the combined dataset is:

1. converted to the WAV format,
2. sampled down to 16kHz, if necessary,
3. converted to mono,
4. stripped of files shorter than 0.7 seconds
5. and stripped of files longer than 17.0 seconds.

There are also two special cases. First, since Common Voice speaker accent information is not available for most of the examples; only examples from the U.S. of America, England, Australia and New Zealand are used. Random sampling suggests that many examples without accent information are not from native speakers. This is in line with Mozilla’s mission statement for the Common Voice dataset [Moz17a], but not with the objectives for this thesis. Secondly, because the TED-LIUM corpus appears to have a disproportional amount of inaccurate shorter labels, examples with a transcription of four or fewer words are omitted. Table 1 shows the markup of the final training dataset and how many hours of each dataset are included.

The Tatoeba corpus is not used in the early experiments. Therefore, some experiments did not use the full training corpora. Results for models that are trained without the Tatoeba dataset are marked with a reduced dataset tag.
All text labels (i.e. transcriptions) are converted to lowercase characters and filtered against a whitelist. Allowing only the characters a to z and the space character into the final label sequence. Hence, the target alphabet is \{a, b, c, \ldots, x, y, z, \_\}, with \(\epsilon\) being the CTC blank-label. Punctuation as well as the apostrophe are not included in the ASR system’s alphabet. Numbers and even the ten digits (i.e. 0 to 9) are not directly used as label, this is because of the following: Numbers are not common within the datasets, so there is not much training material for them. Numbers also have a multitude of special cases. For example, the number 1984 can be pronounced like “one nine eight four”, “nineteen eighty-four”, “nineteen-hundred eighty-four”, “one-thousand nine-hundred eighty-four”, and so on. There are however some examples containing numbers, those are already labeled in one of the previously mentioned ways. With all put together, a sentence like “The student’s bag won’t open.” is reduced to “the students bag wont open”.

4.1.3 Statistics

The distribution of examples with varying lengths in the created dataset is illustrated in figure 20. Emphasizing shorter, command like utterances with a length of 2 to 4 seconds. Distributions for the individual datasets that make up the combined corpus can be found in appendix A.1. The amount of words that the model has to learn from the 901 hours of training material is shown in table 2.

<table>
<thead>
<tr>
<th></th>
<th>Words</th>
<th>Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>8,912,133</td>
<td>46,122,731</td>
</tr>
<tr>
<td>Unique</td>
<td>81,090</td>
<td>27</td>
</tr>
<tr>
<td>Mean label length</td>
<td>15.6</td>
<td>80.8</td>
</tr>
<tr>
<td>Shortest label</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Longest label</td>
<td>84</td>
<td>422</td>
</tr>
</tbody>
</table>

Table 2: Training dataset transcription statistics.

The ten most common words are: “the” (461,863), “to” (270,959), “and” (246,560), “of” (220,573), “a” (198,632), “I” (171,289), “in” (135,662), “that” (130,372), “you” (127,414), and “Tom” (114,623). Note that the word Tom is in tenth place. This is because the Tatoeba corpus uses it as a placeholder for names in general, which of course is not ideal for training. On the other side of the spectrum, there are 27,437 (~1/3 of total) different words that occurred only once in the total of 81,090 unique words of the complete dataset. Furthermore, up to 37,473 words occurred only twice, 50,106 words occurred up to five times and 58,618 words occurred no more than ten times.

4.2 Features

As mentioned in section 3, Related Work; the types of features used in recent ASR systems, as well as the selected feature normalization method varies from system to system. This subsection presents the types of features used and why they are chosen.
Figure 20: Distribution of the length from the audio files of the combined (a) training dataset, (b) validation dataset and (c) test dataset. Ranging from the shortest examples at 0.7 seconds, to the longest being 17.0 seconds long.
4.2.1 Feature Types

The advancement of optimized input features for speech recognition has gone alongside with the development of ASR systems, as good features play an important part in ASR [Dav+80]. Today there are multiple types of features that have successfully been used for speech recognition [Abk+15]. Extracting useful features from the raw audio signal serves several purposes: (1) It increases the robustness against environmental distortions (e.g. reverberation, background noise); (2) it reduces the complexity of the input, which in turn improves the model’s ability to find useful features themselves; (3) and it also reduces the required memory to store the inputs. [Abk+15]

Mel frequency cepstral coefficients (MFCCs) are a commonly used feature and have been in use for over four decades [Dav+80]. With the rise of end-to-end systems, filter banks like the mel spectrogram (MELSPEC) are increasing in popularity. The mel-scale puts emphasize on lower frequencies, while getting coarser for higher frequencies. This is done in order to adjust to human hearing; because humans are able to recognizing small changes in pitch at low frequencies, but require larger changes the higher the frequency is. The shift away from the more refined MFCC features can in part be attributed to the end-to-end training philosophy. Where the model should learn for itself what type of features to extract. [Abk+15]

![Figure 21: Visualization of the process to convert continuous speech into a mel spectrogram (MELSPEC) and into mel frequency cepstral coefficients (MFCCs).](image)

The process to generate these features is illustrated in figure 21. With MELSPEC being a stepping stone on the way to MFCCs. The discrete cosine transformation that is applied to the filter banks compresses the output, leaving only a set number of coefficients. Example representations of both feature types can be seen in figure 22. Both feature types are selected, because of their popularity in ASR and their close relationship.

MELSPEC or MFCC feature calculations are performed at runtime and are not stored. This opens the possibility to artificially manipulate them (e.g. pitch shift, dynamic time warping, addition of artificial noise) in a later version of the system. Such manipulations are used to increase the size and variance of a dataset, which helps to prevent the model from overfitting [Chi+17]. Figure 22 shows an example MELSPEC and MFCC for the utterance “we did it”.

During training either MELSPEC or MFCC are used; they are marked with mel or mfcc in the result section, respectively. Each generated input feature vector $x_i^{(t)}$ contains 80 elements. The length of 80 is selected because it is in the range of vector lengths that comparable ASR systems use (section 3). Additionally, it fits well with the three time-frequency convolutions added in the extended model, which are described later in section 4.5.
Figure 22: Feature extraction example for a MELSPEC at the top and MFCCs at the bottom. The MFCCs include power. Encoded is the sentence: "we did it". Note that the frequency scales are logarithmic.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature type</td>
<td>MELSPEC or MFCC</td>
</tr>
<tr>
<td>Window length</td>
<td>25ms</td>
</tr>
<tr>
<td>Window step</td>
<td>10ms</td>
</tr>
<tr>
<td>Minimum frequency cutoff</td>
<td>64Hz</td>
</tr>
<tr>
<td>Maximum frequency cutoff</td>
<td>8,000Hz</td>
</tr>
</tbody>
</table>

Table 3: The parameters used for the feature extraction process.

Table 3 lists the parameters used to extract the MELSPEC and MFCC features. The minimum frequency is used to remove frequencies that most humans cannot produce [Wik12a]. Whereas the maximum frequency is dictated by the Nyquist frequency and the sampling rate of 16kHz. The overlapping window size is similar to the ones used in other ASR systems. However, the chosen step size of 10ms is shorter than in most systems. This is done to present the RNN with multiple sampling points of the same phoneme, as the CTC algorithm only needs to recognize one of those windows correctly and can produce the blank-label otherwise. Even though it also increases the amount of inputs, and therefore the number of required RNN unrolls.
4.2.2 Feature Normalization

Three kinds of input feature normalization are tested, in addition to no normalization as a control:

**Local normalization:** For each utterance \( x^{(i)} \in X \), that contains \( T^{(i)} \) feature vectors \( x_t^{(i)} \), with \( |x_t^{(i)}| = L \) elements each. The \( L \) element long mean vector \( x_{\text{mean}}^{(i)} \) and standard deviation (STD) vector \( x_{\text{std}}^{(i)} \) are calculated over each of the elements \( x_t^{(i)} \) from the sequence \( x^{(i)} \). Normalization is then applied to each time-slice:

\[
x_{t,\text{norm}}^{(i)} = \left( x_t^{(i)} - x_{\text{mean}}^{(i)} \right) \odot x_{\text{std}}^{(i)}
\]

Local normalization method is marked with: \texttt{local}.

**Local scalar normalization:** Local scalar normalization is again calculated on a per sample basis. Unlike \texttt{local} normalization it does not use mean and STD vectors, but scalars instead. Both \( \hat{x}_{\text{mean}}^{(i)} \) and \( \hat{x}_{\text{std}}^{(i)} \) are derived after flattening the sequence of feature vectors \( x^{(i)} \) to one dimension. Normalization is then applied in the same way:

\[
x_{t,\text{norm}}^{(i)} = \left( x_t^{(i)} - \hat{x}_{\text{mean}}^{(i)} \right) / \hat{x}_{\text{std}}^{(i)}
\]

where \( \hat{x}_{\text{mean}}^{(i)} \) is a vector in which every element is equal to the scalar \( \hat{x}_{\text{mean}}^{(i)} \). Local scalar normalization is marked with: \texttt{local\_scalar}.

**Global normalization:** Instead of calculating the mean and STD per utterance. For global normalization, they are calculated once over the complete training corpus \( X \). Each time-slice \( x_t^{(i)} \) is then normalized using the global mean \( x_{\text{mean}}^{\text{global}} \) and global STD \( x_{\text{std}}^{\text{global}} \) vectors:

\[
x_{t,\text{norm}}^{(i)} = \left( x_t^{(i)} - x_{\text{mean}}^{\text{global}} \right) \odot x_{\text{std}}^{\text{global}}
\]

Global normalization is marked with: \texttt{global}.

**No normalization:** The control, later marked with: \texttt{none}.

It is worth mentioning that Amodei et al. opted for a trainable normalization over each batch in Deep Speech 2 [Amo+15]. They report a relative performance increase of \( \sim 13\% \) for their system, compared to the system without batch normalization. Nonetheless, batch normalization is not tested in this ASR system as it causes problems for the deployed system. During training, batch normalization collects the empirical mean and the variance over each mini-batch and stores them. These values are then used during inference, however, this can have negative effects for a deployed system, where only a single utterance is evaluated instead of a complete batch. [Amo+15]

4.3 Model Description

The Deep Speech (1) architecture consists of many basic NN components and does not contain many non-standard parts [Han+14]. Therefore, its implementation and later on tweaking is straight forward. The described model can be trained on relatively modest (as of writing) consumer hardware; namely on an Nvidia GTX 1060, see “COSY” in appendix A.3 for additional information. And yet the trained DNN needs to be large enough to be able to utilize all the information that the large training corpora offers. This makes a model derived from the Deep Speech (1) architecture a promising candidate to start the development of an ASR system with.
Figure 23: Schematic of the baseline ASR system’s architecture. With the different dense layers (yellow) and the bidirectional-RNN layer (green). The overlapping spectrogram windows are illustrated by the connecting lines between the input layer and the mel spectrogram (MELSPEC). Illustrating the interactions of the RNN at various time-steps.
Graphic inspired by [Han+14].

This system serves as a baseline and is a point of reference to compare later improvements against.

The derived network architecture has a total of six layers. By convention, the input layer is assumed to be at the bottom of the network, while the output layer is at the top; figure 23 illustrates this hierarchy. Three dense layers with 2,048 units each, are at the bottom of the model. Out of those the first one receives the extracted and normalized input features $x^{(i)}$, that are taken from the padded audio-transcription-tuples from one of the buckets (section 2.1.1). Individual feature frames of an example are fed into the network time-step by time-step. Batches of eight examples with roughly equal length are taken from one of the bucketing queues. The training files are first shuffled and then distributed into a total of 64 buckets. The buckets are aligned to match the length distribution of the training dataset (figure 20a), so that every bucket fills up at the same rate. The 64 buckets are a compromise between variance within each batch and performance (i.e. reduction of padding). Figure 2 shows how the bucket boundaries are aligned to the training dataset. Since the complete training dataset does not fit into the RAM, each bucket is assigned a capacity and is constantly refilled with newly extracted features during runtime.

The NN’s dense layer activations are given by:

$$h_t^{(j)} = \tilde{g} \left( W^{(j)} h_t^{(j-1)} + b^{(j)} \right)$$

(23)
where \( j \in \{1, 2, 3, 5\} \) is one of the dense layers. By convention \( \mathbf{h}_t^{(0)} \) is assumed to be the networks input \( \mathbf{x}_t^{(i)} \). The activation function \( \bar{g} \) is a modified version of the default ReLU from equation (2). It is capped at a maximum activation value of 20, to reduce the chance that the network suffers from the exploding gradient problem, and is defined as:

\[
\bar{g}(x) = \min(20, \max(0, x))
\]  

(24)

Since RNN layers are a major contributor to computation times the following concessions are made. The dense layers are followed by only a single bidirectional-recurrent layer, using basic RNN cells. A bidirectional-RNN is used because they achieve better results for sequential systems that use CTC, when compared to unidirectional-RNNs [Gra+06]. To further reduce the training times, every second feature window of the input sequence \( \mathbf{x}^{(i)} \) is skipped. Presenting the NN only the time-slices \( \mathbf{x}_{t'}^{(i)} \), where \( t' \in \{1, 3, 5, \cdots\} \). The output of the forward and backward RNNs for the 4-th hidden layer \( \mathbf{h}_t^{(4)} \) of the \( t \)-th time-step is computed as follows: [Han+14]

\[
\mathbf{\tilde{h}}_t^{(4)} = g \left( \mathbf{W}^{(4)} \mathbf{h}_t^{(3)} + \mathbf{U}^{(4)} \mathbf{\tilde{h}}_{t-1}^{(4)} + \mathbf{b}^{(4)} \right)
\]

\[
\mathbf{\hat{h}}_t^{(4)} = g \left( \mathbf{W}^{(4)} \mathbf{h}_t^{(3)} + \mathbf{U}^{(4)} \mathbf{\hat{h}}_{t+1}^{(4)} + \mathbf{b}^{(4)} \right)
\]  

(25)

The RNN cells use the standard ReLU activation function from equation (2), because the cuDNN version of a bidirectional-RNN does not support clipping. Both forward and backward RNN contain 2,048 basic RNN cells, therefore the whole bidirectional-RNN layer has a total of \( 2 \times 2,048 = 4,096 \) units.

One of the drawbacks of bidirectional-RNNs is, that they require the complete input-sequence \( \mathbf{x}^{(i)} \) of length \( T^{(i)} \) to begin calculations. This is because the backward RNN \( \mathbf{\hat{h}}_t^{(4)} \) needs to be computed sequentially from end \( t = T^{(i)} \) to beginning \( t = 1 \), while the forward RNN \( \mathbf{\tilde{h}}_t^{(4)} \) can be computed from start \( t = 1 \) to end \( t = T^{(i)} \) [Han+14]. The final output for the bidirectional-RNN layer for time-step \( t \) is the concatenated output of the forward \( \mathbf{\tilde{h}}_t^{(4)} \) and backward \( \mathbf{\hat{h}}_t^{(4)} \) RNNs:

\[
\mathbf{h}_t^{(4)} = \mathbf{\tilde{h}}_t^{(4)} \odot \mathbf{\hat{h}}_t^{(4)}
\]  

(26)

where \( \odot \) represents the concatenation operation.

The fifth layer is a dense layer that again is defined by equation (23). It is intended to interpret the outputs of the RNN and to support the network’s output layer to classify the input. The output layer is a linear activated (i.e. no activation function) dense layer. It contains a unit for each element of the target alphabet \( \mathcal{K} = \{a, b, c, \cdots, z, _, \epsilon\} \). The output is normalized using the softmax function from equation (27), resulting in a probability distribution over each character \( k \in \mathcal{K} \) of the target alphabet for every time-step \( t \).

\[
h_{t,k}^{(6)} \equiv \mathbb{P}(c_t = k | \mathbf{x}) = \frac{e^{\mathbf{W}_k^{(6)} \mathbf{h}_t^{(5)} + b_k^{(6)}}}{\sum_j e^{\mathbf{W}_j^{(6)} \mathbf{h}_t^{(5)} + b_j^{(6)}}}
\]  

(27)

where \( c_t \in \mathcal{K} \) and \( h_{t,k}^{(6)} \) is the activation of the output unit at time-step \( t \) for character \( k \). This is the network’s prediction for that character. Please note that in order to apply the softmax activation function, the output vector \( \mathbf{h}_t^{(6)} \) has to be considered componentwise. Therefore, \( \mathbf{W}_k^{(6)} \) and \( b_k^{(6)} \) represent only the part(s) of the weight matrix and bias vector that are relevant for the character \( k \).

During training the softmax activated output is fed into the CTC function to calculate the costs. Additionally, the output is fed into a CTC beam search decoder using 1,024 beams to
deduct the most likely sequence of characters. Then the WER and edit distance (ED) for the most probable character sequence is calculated. Finally, the weights are updated based on the CTC costs via backpropagation through time [Wer88].

Figure 23 illustrates the described network architecture and table 4 lists the dimensions of each layer. This architecture is referred to as 3d1r2d in the results section, with 3d indicating three dense layers, followed by 1r a single bidirectional-RNN layer and followed by 2d two dense layers, of which the last one is the output layer.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input method</td>
<td>Bucketing (64 buckets)</td>
</tr>
<tr>
<td>Input order</td>
<td>Shuffled</td>
</tr>
<tr>
<td>Number of units per dense layer</td>
<td>2,048</td>
</tr>
<tr>
<td>Number of units per RNN layer</td>
<td>2 × 2,048</td>
</tr>
<tr>
<td>Number of units in output layer</td>
<td>28</td>
</tr>
<tr>
<td>RNN cell type</td>
<td>Basic RNN</td>
</tr>
<tr>
<td>Beam search beam width</td>
<td>1,024</td>
</tr>
</tbody>
</table>

Table 4: The baseline model composition and dimensions.

4.4 Training Process

To begin the training process of a new model, all weight matrices \( W \) are initialized using a uniform Xavier weight initializer [Glo+10]. It samples values from a uniform distribution from the interval \([-\text{limit}, +\text{limit}]\), where

\[
\text{limit} = \sqrt{\frac{6}{\text{in} + \text{out}}}
\]

and \( \text{in} \) and \( \text{out} \) are the number of input and output units for the layer, respectively. The bias vectors \( b \) are initialized with all zeros.

The training examples are shuffled at the beginning of each epoch. Batches of size 8 are taken from the buckets and fed into the network. The relatively small batch size is the result of restrictions from the computation servers that are detailed in appendix A.3. A larger batch size should in theory reduce the amount of batches that induce an extreme weight change, which in turn is causing the model to end up in a non-recoverable state. This is especially problematic the more layers the network has, because of the exploding or vanishing gradient problem. A larger batch size would also reduce the training times, since the GPU can compute multiple examples at once.

The initial learning rate (LR) is set to \( 10^{-4} \), because tests show that CTC gets unstable if the LR is too large. To carefully approach an optima, the LR is decayed every 50,000 steps (\(~2/3\) of an epoch) by multiplying it with a factor of \( 3/4 \). The LR is capped at a minimum of \( 10^{-7} \) to reduce floating-point inaccuracies.

A momentum optimizer [Hea15, pp. 259 – 263] with a momentum of 0.9 is used as LR-policy. Momentum helps the model to escape local optima, that are difficult for stochastic gradient descent (SGD), because it could oscillate around the optima. Momentum also helps to keep the gradient in the general direction of the previous iteration, as well as dampening oscillation. To do this, it accumulates momentum from the previous iteration’s gradient and applies it to the current iteration’s gradient. [Hea15, pp. 259 – 263].
Following the backpropagation weight update the next batch is fed. The epoch is completed if no bucket contains a complete batch, discarding all examples that could not be combined into a complete batch.

The servers used for training are listed in appendix A.3. Table 5 summarizes the hyperparameters used for training.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight initializer</td>
<td>Xavier initializer</td>
</tr>
<tr>
<td>Bias initializer</td>
<td>Constant 0</td>
</tr>
<tr>
<td>Batch size</td>
<td>8</td>
</tr>
<tr>
<td>Minimum example length</td>
<td>0.7 seconds</td>
</tr>
<tr>
<td>Maximum example length</td>
<td>17.0 seconds</td>
</tr>
<tr>
<td>Learning rate</td>
<td>$10^{-4}$</td>
</tr>
<tr>
<td>Learning rate decay factor</td>
<td>$3/4$</td>
</tr>
<tr>
<td>Learning rate decay every</td>
<td>50,000 steps</td>
</tr>
<tr>
<td>Minimum learning rate</td>
<td>$10^{-7}$</td>
</tr>
<tr>
<td>Learning rate policy</td>
<td>Momentum optimizer</td>
</tr>
<tr>
<td>Momentum</td>
<td>0.9</td>
</tr>
<tr>
<td>ReLU cutoff</td>
<td>20.0</td>
</tr>
<tr>
<td>Dropout dense layers</td>
<td>10%</td>
</tr>
<tr>
<td>Dropout RNN layers</td>
<td>0%</td>
</tr>
</tbody>
</table>

Table 5: The default hyperparameters used for most test runs, unless stated otherwise for a specific result. These parameters also apply to the extended model from section 4.5, however, most of the test runs for that use an increased batch size of 16.

4.5 Extended Model

The following modifications are made to the baseline model described in section 4.3, most of them are inspired by the *Deep Speech 2* model [Amo+15]. The overall NN architecture is changed in the following ways:

First, the three dense input layers are replaced with three 2D convolutional layers, applying time-frequency convolutions to the input. 2D convolutions are somewhat uncommon for speech data, as most papers researched for this thesis use 1D time convolutions, if they used convolutions. The typical companion of the convolutional layer, the pooling layer, is not used for this model. Kernels and strides are listed in table 6. They are chosen so that the temporal convolutions reduce the input length by half; making the previously used skipping of every second input frame (section 4.3) obsolete. Temporal convolutions are commonly used to enable the model to compensate for variable input lengths and rate of speaking [Amo+15]. While kernels and strides for the frequency convolutions are chosen so that they reduce the input vector’s size from 80 elements down to 10. Figure 24 illustrates the reduction in size and the increasing number of output channels. The use of 2D convolutions allows the network to learn its own feature representations, as well as reducing the size of the input fed into the following RNN layer.

And secondly, the amount of RNN layers used is increased from a single layer, to up to five.
Figure 24: Time-frequency convolutions for an input with $T$ windows and 80 features per window. Only the Conv1 layer has a stride greater than 1, therefore reducing the number of time-steps it outputs. Both the first and second convolutional layer learn 32 kernels, and therefore output 32 channels. The last convolutional layer has 96 channels.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Channels</th>
<th>Kernel</th>
<th>Stride</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>32</td>
<td>$11 \times 41$ (2, 2)</td>
<td></td>
</tr>
<tr>
<td>Second</td>
<td>32</td>
<td>$11 \times 21$ (1, 2)</td>
<td></td>
</tr>
<tr>
<td>Third</td>
<td>96</td>
<td>$11 \times 21$ (1, 2)</td>
<td></td>
</tr>
</tbody>
</table>

Table 6: Parameters for the three CNN layers. All stated dimensions first state the time, followed by the frequency.

Additional RNN layers enable the NN to learn more information from the large training dataset, where the single layer could not store the same amount of information [Amo+15]. Another reason to increase the number of RNN layers are the theoretical results discussed in “Learning Deep Architectures for AI” by Bengio [Ben09]. They suggest that in order for a NN to learn the kind of complicated functions that can represent the high-level abstractions required for speech recognition, deep architectures are required [Ben09]. This change increases the required amount of VRAM and the training times noticeably. The deeper architectures also have a greater chance to run into either the exploding or vanishing gradient problem during training. Amodei et al. tested a similar network with up to eleven bidirectional-RNN layers [Amo+15]. However, this model was limited to a maximum of five layers RNN layers, due to the increased training times and the available VRAM.

Figure 25 is a side-by-side comparison of the architectural changes made to the Deep Speech (1) model. To keep the extended model with CNN layers (table 6) comparable to the baseline model, the remaining parameters (table 4) are kept the same. Convolutional layers are referred to with 1c in the results section, e.g. a NN with one convolutional layer, two RNN layers and three dense layers is labeled $1c2r3d$, with the last dense layer being the softmax activated output layer.
Figure 25: Side by side comparison of the network architectures. (a) The baseline model with 3 dense layers, 1 bidirectional-RNN (BDRNN) layer and 2 dense layers, with the top dense layer being the softmax activated output layer. (b) And the extended model comprised of 3 CNN layers, up to 5 bidirectional-RNN layers and followed by 2 dense layers.

4.5.1 Different Cell Types

The default RNN cell type used is the simple RNN cell. The decision to use them is primarily based on its low computational demands. While the simple RNN cells are by far the fastest to train, a GRU can still be trained faster than a LSTM [Amo+15]. A study of several LSTM variations suggests that, when properly initialized, GRUs can achieve comparable results to LSTMs [Joz+15]. Both include mechanisms that should enable them to better handle the long training sequences and prevent exploding or vanishing gradients.

To determine the model architecture that has the highest potential to achieve the best WER, initial experiments use the simple RNN cell. Following this, the best candidates are trained again, this time using the more potent GRU and LSTM.

4.5.2 Gradually Increasing Training Difficulty

A common way to train a NN is to feed the examples in random order. Feeding data in a predetermined order could, over time, allow the network to remember the sequence instead of learning useful features. Shuffling data also splits up sequences, where several permutations of the same sentence are directly following each other within the dataset; this is especially true for the Tatoeba corpus (section 4.1). It is also beneficial when using mini-batches, since stochastic gradient descent depends on the batches being representative of the overall training data. On the other hand, “[...] humans and animals learn much better when the examples are not randomly presented but organized in a meaningful order which illustrates gradually more concepts, and gradually more complex ones.” [Ben+09].

SortaGrad [Amo+15] assumes that the complexity of a training example is proportional to the length of its audio sequence. Another metric could be the number of characters or words, in its transcription. To begin a new training with easy examples and then gradually increase the
difficulty, the training data is sorted by length, from shortest to longest. Since the CTC cost function tends to produce larger gradients for longer sequences; starting with short sequences results in smaller weight adjustments in the beginning, minimizing the risk of exploding gradients. [Amo+15]

The extended model uses audio length based SortaGrad for the first training epoch, while the baseline model does not. After that, it switches back to bucketing with randomly shuffled inputs.

4.6 Summary

This section introduced the datasets used for training, development and the final testing of a trained model. The training dataset is built from the Creative Commons (CC) licensed corpora: Common Voice, LibriSpeech ASR, Tatoeba and TED-LIUM. With the final training set containing over 901 hours of labeled speech. The section further presented the commonly used audio features MELSPEC and the more refined MFCCs. Since the volume and recording environments of the merged dataset varies, the extracted features require normalization. Three normalization methods for that purpose, which work on a per example basis, are presented.

Following this is an ASR system that can be trained on consumer hardware. This system consists of three dense input layers that are used for internal preprocessing. Followed by a single bidirectional-RNN layer and two dense layers. With the final dense layer being softmax activated, so that it provides a probability distribution over the target alphabet, for each time-step. This probability distribution is then used by the CTC algorithm to determine the networks cost for the given input, which in turn is used by the backpropagation through time algorithm to adjust the networks weights.

The section closes by introducing a more sophisticated ASR system that expands the baseline system in multiple ways. Among these changes is a deeper architecture, with an increased number of RNN layers, allowing the NN to extract more information from the training material. To further improve the internal feature extraction, the three dense input layers are replaced by convolutional layers; performing 2D convolutions in both time and frequency space.
5 Evaluation

This section presents and discusses the results of the performed test runs. The sections 5.1 to 5.5 handle the baseline system, while sections 5.6 to 5.10 handle the extensions made to it. Finally, section 5.11 discusses some overarching points. Discussion and interpretation of each result follows in a separate subsection directly afterwards. Result tables in this section highlight the best result in bold font. But first the employed error metrics are defined.

Since the CTC cost is a vector for every element of the target alphabet (section 2.7), the following CTC values are the mean value of such a vector. Besides the CTC cost, the following two metrics are used to evaluate a model. A common error metric in ASR is the word error rate (WER). It is an extension to the Levenshtein distance and is a type of edit distance (ED). It works on a word level, instead of character level that the Levenshtein uses. WER provides a metric how equal two sentences are. A WER of 0 means both sentences are the same. The WER for a reference sentence $y$ and a predicted transcription $\hat{y}$ is calculated as:

$$L_{\text{WER}}(\hat{y}, y) = \frac{S + D + I}{N}$$

where $S$ is the number of words substituted, $D$ is the number of words deleted, and $I$ the number of words inserted into $\hat{y}$ to transform it into $y$. $N$ being the number of words in the reference sentence $y$. [Wik12b]

A WER of 0.1 relates to one misspelled word for every ten words. If the average sentence contains ten words, the error rate per sentence could reach 1, if the WER is 0.1. And even a single mistaken word can alter the semantic meaning of a sentence. It should also be mentioned that WER does not provide any information on which part of the prediction is wrong. While the WER usually is a value between 0 and 1, a WER greater than 1 is possible. This is the case if the prediction $\hat{y}$ is longer than the target $y$ sequence, and therefore requires additional deletions.

The aforementioned Levenshtein distance or character edit distance (ED) is used as a secondary error metric. It is calculated on a per character level. This allows for minor spelling errors to only affect a small portion of the error, as it is likely that some word-level errors are caused by a single character. For consistency with other research, the ED and WER are interpreted as a percentage value.

The results in sections 5.1 to 5.10 are based on a single training run with the described model, therefore, the margin of error for individual results is unknown. To put the following results into perspective, 20 training runs with a reduced baseline system are performed. All use varying random seeds and a reduced dataset that contains only the 300,000 shortest examples. The reduced baseline system and individual results for the 20 runs are listed in appendix A.2. Each of this training runs is stopped after exactly 15 epochs. Their performance is measured on the complete test dataset. The mean ED (character based) and mean WER are 29.4% and 71.4%, respectively. Given a confidence level of 95%, the estimated population mean of the ED is between 29.1% and 29.6% (±0.3%) and between 71.0% and 71.8% (±0.4%) for the WER. Please keep in mind that this model is less complex than the ones used in sections 5.1 to 5.10, so it is likely that the margin of error is larger for those.
5.1 Baseline System

Sections 5.1 to 5.5 list results for the baseline system described in section 4.3. It is sectioned into a comparison of different input feature types and normalization methods. Training runs for the baseline system are executed on the COSY servers, described in appendix A.3. Section 5.5 compares the baseline system with another system that is also based on the Deep Speech (1) paper [Han+14].

All models are trained for a maximum of 20 epochs, but are stopped early when the CTC costs are beginning to increase again (early stopping). However, tests on the reduced dataset do not use early stopping and can suffer from overfitting.

With regard to the full dataset, including the Tatoeba set, the following is worth mentioning. A iteration of the baseline system (3d1r2d) with three dense layers, one bidirectional-RNN layer that is followed by two dense layers is trained on 709 hours of the reduced dataset (without Tatoeba). It uses 80 locally normalized MFCC features and achieves a WER of 25.5%. With the same model, trained on the full 901 hours from the complete dataset, achieving a WER of 30.4%. Both results are measured on the same test dataset. This suggests that the low number of speakers (i.e. 6) in the Tatoeba corpus is not improving the model’s ability to abstract.

5.2 Input Feature Types

Results and discussion for the input features mel spectrogram (MELSPEC) and mel frequency cepstral coefficient (MFCC) that are presented in section 4.2.1.

5.2.1 Results

In the first two rows of table 7 are results for the same network, once trained with MELSPEC input features and once with MFCC input features. They are both tested on the standard baseline model (3d1r2d) using the complete training dataset. No input normalization is used (i.e. none).

The last two rows in table 7 contain results for the extended model, in order to provide some reference. It uses convolutional input layers and 5 bidirectional-LSTM layers (3c5r2d). Both of them used local normalization, instead of none. They are also trained on the complete training dataset.

<table>
<thead>
<tr>
<th>Model</th>
<th>Feature</th>
<th>CTC</th>
<th>ED</th>
<th>WER</th>
<th>Epochs</th>
</tr>
</thead>
<tbody>
<tr>
<td>3d1r2d</td>
<td>mel</td>
<td>28.9</td>
<td>12.1%</td>
<td>33.5%</td>
<td>14</td>
</tr>
<tr>
<td>3d1r2d</td>
<td>mfcc</td>
<td>23.9</td>
<td>9.6%</td>
<td>27.3%</td>
<td>8</td>
</tr>
<tr>
<td>3c5r2d</td>
<td>mel</td>
<td>13.3</td>
<td>4.4%</td>
<td>12.9%</td>
<td>5</td>
</tr>
<tr>
<td>3c5r2d</td>
<td>mfcc</td>
<td>12.1</td>
<td>4.6%</td>
<td>12.7%</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 7: Results for the input feature types MELSPEC and MFCC. 3d1r2d is the baseline model and 3c5r2d is the extended model with LSTM cells. The best result within a test group is highlighted with **bold** font.
5.2.2 Discussion

All three cost metrics show that the more refined MFCC features are advantageous for the baseline model. It can be speculated that the dense layers lack the ability to extract suitable features from the two-dimensional MELSPEC, and therefore benefit from the more refined MFCC features. However, this would require additional tests to prove. The deeper, convolutional models achieve nearly identical results with both MELSPEC and MFCC. Suggesting that CNNs are able to extract useful features from both time and frequency domain. While the deeper RNN structure should allow for more granular levels of abstraction.

5.3 Normalization Methods

Results and discussion for the three feature normalization techniques presented in section 4.2.2.

5.3.1 Results

Table 8 contains test results for the three different input feature normalization methods discussed. The control that does not perform any input normalization is marked with none. All experiments use the baseline model (3d1r2d) with MELSPEC input features. The first four results are for models trained on the reduced dataset (without Tatoeba). These experiments are repeated on the full dataset, their results are in the bottom four rows of table 8.

<table>
<thead>
<tr>
<th>Method</th>
<th>Dataset</th>
<th>CTC</th>
<th>ED</th>
<th>WER</th>
<th>Epochs</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>reduced</td>
<td>29.6</td>
<td>11.2%</td>
<td>31.7%</td>
<td>20</td>
</tr>
<tr>
<td>local</td>
<td>reduced</td>
<td>29.0</td>
<td>10.7%</td>
<td>30.9%</td>
<td>20</td>
</tr>
<tr>
<td>global</td>
<td>reduced</td>
<td>30.6</td>
<td>11.3%</td>
<td>31.9%</td>
<td>20</td>
</tr>
<tr>
<td>local_scalar</td>
<td>reduced</td>
<td>31.9</td>
<td>11.4%</td>
<td>32.1%</td>
<td>20</td>
</tr>
<tr>
<td>none</td>
<td>full</td>
<td>32.0</td>
<td>13.5%</td>
<td>37.6%</td>
<td>16</td>
</tr>
<tr>
<td>local</td>
<td>full</td>
<td>29.5</td>
<td>12.5%</td>
<td>35.4%</td>
<td>15</td>
</tr>
<tr>
<td>global</td>
<td>full</td>
<td>31.4</td>
<td>13.1%</td>
<td>37.1%</td>
<td>14</td>
</tr>
<tr>
<td>local_scalar</td>
<td>full</td>
<td>31.7</td>
<td>13.2%</td>
<td>37.3%</td>
<td>15</td>
</tr>
</tbody>
</table>

Table 8: Results for different normalization methods. The models are trained on both the full and the reduced dataset, which does not include the Tatoeba examples.

5.3.2 Discussion

All three error metrics indicate that local normalization works best for the baseline model. Whereas the other three methods appear to achieve comparable, but slightly worse results. The local_scalar normalization manipulates the features for all frequencies with a constant factor. It is reasonable to assume that the first layers of the control, that uses no normalization (none), is able to learn this factor and adjust its weights accordingly. This could produce results that are comparable to those of the local_scalar normalization.

Nonetheless, the results for global normalization are unexpected, as this normalization method is common in machine learning. It is possible that the calculation of global mean and STD could suffer from floating-point inaccuracies, because of the huge amount of feature vectors.
5 EVALUATION

contained in the complete training dataset. Alternatively, multiple calculations of the STD, based on only 10,000 random samples, produces values that differ by as much as 40%. Random sample based calculations are therefore not further considered. The problem with global normalization could also indicate that a batch normalization method, as described in [Amo+15], could improve the results. Additional preprocessing, like volume normalization, could also help the system.

5.4 Number of Units per Layer

Results and discussion for an increase in the amount of neurons per layer.

5.4.1 Results

All models use 2,048 units per layer. To evaluate how an increase in units affects the WER, a baseline model (3d1r2d) with 3,000 units per layer is trained. The larger model averages to complete 0.73 batches per second, while the default model with 2,048 units per layer averages 1.44 batches per second. Both of them are trained on the same COSY server (appendix A.3). Each model is trained for 20 epochs on the reduced dataset. Their inputs are MELSPECs with local normalization. The larger model achieves a WER of 29.0% and a mean ED of 10.2%. Compared to 30.9% WER and 10.7% mean ED for the control, which has the default 2,048 units per layer. This is a relative percentage difference in WER of ~6.1 and ~4.7 in ED.

5.4.2 Discussion

The increase in units per layer is showing an improvement in both WER and ED. However, the additional 952 units per layer almost double the required computation time. Therefore, this improvement appears to be only useful for a production system and not worth the extra time for a proof of concept.

5.5 Comparison with Reference System

Results and discussion for the comparison of the best performing baseline model (section 5.1), with the Deep Speech (1) [Han+14] based implementation from Mozilla. The ASR system named Project DeepSpeech [Moz17b] uses 26 MFCCs as inputs, including one entry for the power. In addition to the 26 MFCC features \( x_t \) for the current time-step, they add four context frames from the past and four from the future to it. This stacked sequence \( [x_{t-4}, \cdots, x_{t-1}, x_t, x_{t+1}, \cdots, x_{t+4}] \) results in an input vector with \( 9 \times 26 = 234 \) elements.

Their network layout consists of three dense layers, one RNN layer and two dense layers, which is the Deep Speech (1) default architecture [Han+14]. They use 2,048 units per layer, as well. However, they opted to use a bidirectional-LSTM instead of the simple RNN cells used in [Han+14]. The model is trained on a combined dataset, consisting of LibiSpeech ASR, TEDLIUM and the commercial Fisher Corpus. This should make the test results comparable, to a certain degree. In addition to the end-to-end trained model, Mozilla has also integrated an optional external LM.

---

16 There are approximately \( 3.2 \times 10^8 \) feature vectors in the training dataset.
17 Relative percentage difference: https://en.wikipedia.org/wiki/Relative_change_and_difference
18 Fisher Corpus: https://catalog.ldc.upenn.edu/LDC2004T19
5.5.1 Results

The following results are for the pre-trained model of Project DeepSpeech, in version 0.1.1 [Moz17b]. Without the LM, it achieves a WER of 21.2\% on the same test corpora that is described in section 4.1. The WER drops down to 10.4\% with the external LM enabled.

5.5.2 Discussion

The change in the WER caused by the addition of an external LM is very noticeable. This is also highlighting one of the problems of the CTC cost function: its inability to enable the RNN to learn a strong internal LM. Some possible reasons for this are expanded further in section 5.11. The higher WER of 27.3\% for the baseline system could be due to differences in training material, tuned hyperparameters or the more powerful LSTM cells used. It is also possible that the context frames that are not used for this system, could contribute to the difference.

5.6 Extended Model

Sections 5.6 to 5.10 present results for the extended model described in section 4.5. In the extended model, the three dense input layers that the baseline system uses, are replaced by 2D convolutional layers. It also uses an increased amount of bidirectional-RNN layers. Experiments for the extended model use the SortaGrad method for the first epoch of training (section 4.5.2).

For the experiments with the extended models, two servers with Nvidia V100 GPUs are available. They enable the training of larger models and allow an increased batch size of 16, up from 8. The increased batch size reduces training times. More details on the servers can be found in appendix A.3.

The extensions are compared to the best performing baseline system. This is a 3d1r2d model, using local normalized MFCC features (table 7). Relative percentage differences stated in the following results and discussions are therefore in comparison to the baseline system that achieves a WER of 27.3\%.

5.7 Comparison of Convolutional and Dense Input Layers

Results and discussion for the comparison of dense input layers and convolutional input layers, as described in section 4.5.

5.7.1 Results

The network 3d1r2d uses three dense layers at the bottom, while the 3c1r2d network uses three convolutional layers instead. Parameters for the CNN layers are listed in table 6, all other parameters for both models are kept the same. Both models are trained on the full dataset. MELSPECs are used as input features, because they are less refined than MFCCs. This should encourage the networks to create a refined feature representation themselves. All three normalization methods and the none control are tested. Table 9 contains the results.

5.7.2 Discussion

Dense input layers only work well with local normalized inputs, as already discussed in section 5.3. This is no longer the case with convolutional layers. Across the board, they are performing better than the dense layers. They are also able to utilize both local and local_scalar
<table>
<thead>
<tr>
<th>Network</th>
<th>Normalization</th>
<th>CTC</th>
<th>ED</th>
<th>WER</th>
<th>Epochs</th>
</tr>
</thead>
<tbody>
<tr>
<td>dense</td>
<td>none</td>
<td>32.0</td>
<td>13.5%</td>
<td>37.6%</td>
<td>16</td>
</tr>
<tr>
<td>dense</td>
<td>local</td>
<td>29.5</td>
<td>12.5%</td>
<td>35.4%</td>
<td>15</td>
</tr>
<tr>
<td>dense</td>
<td>global</td>
<td>31.4</td>
<td>13.1%</td>
<td>37.1%</td>
<td>14</td>
</tr>
<tr>
<td>dense</td>
<td>local_scalar</td>
<td>31.7</td>
<td>13.2%</td>
<td>37.3%</td>
<td>15</td>
</tr>
<tr>
<td>convolutional</td>
<td>none</td>
<td>27.0</td>
<td>10.6%</td>
<td>29.2%</td>
<td>9</td>
</tr>
<tr>
<td>convolutional</td>
<td>local</td>
<td>22.7</td>
<td>9.1%</td>
<td>25.6%</td>
<td>8</td>
</tr>
<tr>
<td>convolutional</td>
<td>global</td>
<td>24.1</td>
<td>9.4%</td>
<td>26.7%</td>
<td>9</td>
</tr>
<tr>
<td>convolutional</td>
<td>local_scalar</td>
<td>23.6</td>
<td>9.0%</td>
<td>25.6%</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 9: Results for the comparison between dense and convolutional layers, for different normalization methods.

normalization, showing comparable results for them. Even the problematic global normalization is only slightly behind. However, the CNN system without any normalization (none) performed worse than any of the normalization methods.

The relative percentage difference in WER between the best performing dense normalization (local) and no normalization (none) is 6.2. However, for the CNN the relative percentage difference between local / local_scalar and none is 14.1, which is considerably larger.

A possible cause for this are the weights that a CNN shares between multiple elements of the input vector. They apply their local receptive field to the complete frequency spectrum of the input, using the same weights. In contrast to that, a dense layer has individual weights for each element of the input, therefore it can adjust them based on the frequency of their input element. This could make CNNs more dependent on normalized inputs.

5.8 Number of Recurrent Layers

Results and discussion for an increasing number of RNN layers, while keeping the number of units per layer constant. Section 4.5 described the expected increase in the model’s ability to abstract and memorize data, by increasing its depth.

5.8.1 Results

Table 10 lists the results on the test dataset for five models. The only changing parameter is the number of bidirectional-RNN layers. The number of units in each bidirectional-RNN layer is kept constant at $2 \times 2,048 = 4,096$. All models are trained on the full dataset, with a batch size of 16 and use local normalized MELSPEC inputs. Training is stopped as soon as the CTC cost starts to increase (i.e. early stopping).

5.8.2 Discussion

The results show a decrease in all three error metrics for each bidirectional-RNN layer added. This goes in tandem with an increase in the epochs needed for the NN to converge. Apart from the run with four recurrent layers (4r), which is stopped early after only 11 epochs. With the improvement on this run not being in line with the results for the models with three (3r) and five (5r) layers. Therefore, the 3c4r2d result could be considered an outlier. Furthermore, since the epochs until conversion for 3r and 5r are close together, it is feasible that the LR and its
rate of decay are not ideal and should be tuned to better suit the network’s depth. This is a relative percentage improvement of 42.1 in WER from the 3c5r2d system, compared to the baseline system.

5.9 Different Cell Types

Results and discussion for the effects of different RNN cell types, as described in section 4.5.1.

5.9.1 Results

For each cell type (i.e. RNN, GRU and LSTM) two deep networks are trained, one with four bidirectional-RNN layers 4r and one with five 5r. The results for the basic RNN cell use the standard ReLU activation function from equation (2). Trying to use the hyperbolic tangent instead, caused the NNs to diverge after a few epochs, leaving them in a non-recoverable state.

Table 11 lists the results for different types of RNN cell. The GRU model with five layers (5r) could not be trained due time constraints. All models are trained on the full dataset and use local normalized MELSPEC features. For reference: the 3c4r2d model using basic RNN cells completed $\sim 1.43$ batches per second and the LSTM model $\sim 0.45$ batches per second, both on the V100(a) server.\footnote{The GRU model completed $\sim 1.30$ batches per second on the V100(b) server. However, these values cannot be compared directly, due to the throttling required for the V100(a) server (appendix A.3).}

<table>
<thead>
<tr>
<th>Network</th>
<th>Cell</th>
<th>CTC</th>
<th>ED</th>
<th>WER</th>
<th>Epochs</th>
</tr>
</thead>
<tbody>
<tr>
<td>3c1r2d</td>
<td>RNN</td>
<td>17.4</td>
<td>6.8%</td>
<td>19.4%</td>
<td>10</td>
</tr>
<tr>
<td>3c5r2d</td>
<td>RNN</td>
<td>14.0</td>
<td>5.7%</td>
<td>15.8%</td>
<td>16</td>
</tr>
<tr>
<td>3c4r2d</td>
<td>GRU</td>
<td>16.8</td>
<td>6.7%</td>
<td>19.1%</td>
<td>10</td>
</tr>
<tr>
<td>3c5r2d</td>
<td>GRU</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>3c4r2d</td>
<td>LSTM</td>
<td>11.9</td>
<td>4.5%</td>
<td>12.6%</td>
<td>5</td>
</tr>
<tr>
<td>3c5r2d</td>
<td>LSTM</td>
<td>13.3</td>
<td>4.4%</td>
<td>12.9%</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 11: Results for the RNN cell types: RNN, GRU and LSTM. Each cell type is tested with four and five stacked bidirectional layers, except for the five layer GRU model.
5.9.2 Discussion

The four layers of bidirectional-GRU did not greatly improve, compared to the results for the four layers of basic RNN cells. The GRU result could be due to suboptimal cell initialization. As the [Joz+15] study states, that GRUs can perform on the same level as LSTM cells, if they are initialized correctly. Amodei et al. support this thesis, too. They have managed to achieve their best results using GRUs [Amo+15].

Even though the LSTM network requires more computation time during training, it appears to be able to converge faster than the basic RNN. Allowing the LSTM to be stopped several epochs before the basic RNN converges. The small drop in WER going from four to five LSTM layers could be within the margin of error. However, it could also be the result of insufficient normalization, as [Amo+15] have shown that training with up to 11 RNN layers is possible, while still achieving better results. Therefore, further normalization (e.g. batch-normalization) could improve the results for deeper networks. Another factor could be suboptimal hyperparameters, as they have not been altered over the course of the experiments, to keep results comparable.

The 3c5r2d LSTM provides a relative percentage improvement of 18.4 over the basic RNN version; and 52.7 when compared to the baseline system.

5.10 Gradually Increasing Training Difficulty

Results and discussion for the effect of gradually increasing training difficulty to bootstrap an ASR system, as described in section 4.5.2.

5.10.1 Results

To test the effects SortaGrad has on the training of the extended system, two models with three convolutional, four bidirectional-RNN and two dense layers (3c4r2d) are trained. Inputs are MELSPEC with local normalization. One is using input examples sorted by audio length for the first epoch, as described in section 4.5.2; this model is referenced as SortaGrad model. The second model is trained using randomly shuffled buckets for every epoch, it serves as the control.

The SortaGrad model, that receives sorted examples during the first epoch, successfully converged the first time. Training of the control model required six restarts, until it converged. The initial LR for the control is reduced to $10^{-5}$, down from $10^{-4}$ in order to train it.

Figure 26 shows the performance of both models, as measured on the validation dataset after each completed training epoch. The control converged after 16 epochs and the SortaGrad model after 11. Both trained models are evaluated on the test dataset, with the control achieving a WER of 20.3% and the SortaGrad model a WER of 19.4%.

5.10.2 Discussion

The baseline model with only a single recurrent layer can be trained using shuffled buckets for every epoch, including the first. However, a few of the training runs drifted into a non-recoverable state, where no valid CTC path could be decoded. With the deeper models generally appearing to be more prone to unstable states. Based on the limited data available, it is likely that the control (without SortaGrad) has a greater tendency to get unstable in the first few epochs. This can be caused by the calculated CTC costs, because they are not normalized by length. Therefore, a long training example fed to the network early on, can prompt a large weight

---

20 It is not on record how many training runs had this problem.
5.11 General Observations

This subsection points out and discusses general observations that are not necessarily measurable and are based on subjective observations. The presented observations are intended to provide a deeper understanding of some challenges an ASR system faces.

5.11.1 Pronunciation and Spelling

Speech recognition systems make phonetic mistakes, especially if they do not use a LM. Consider for example the “th” sounds of the voiceless fricative /θ/ and the voiced fricative /ð/ both have the same transcription, but are pronounced differently (e.g. “thing”, “thunder” vs. “the”, “this”). Another challenge are homophones, which are different words that are pronounced the same, e.g. “two” and “to”. The problems caused by homophones cannot even be corrected by a dictionary and would require a sophisticated LM [Gra+14].

Table 12 contains two example transcriptions\(^ {21}\), predicted by the 3c5r2d basic RNN model

---

\(^ {21}\) The examples are from the training dataset. Both of them are handpicked from a randomly chosen batch of four. They are presented to provide an example for the following arguments.
during the fifth training epoch. The examples let it appear feasible that the use of an external LM could reduce the WER. This thesis is further supported through the character ED, which is lower than the WER for all experiments. However, the use of an external LM would disqualify this system as a true end-to-end trained system. The RNN should also be capable of learning a LM itself, if it has enough training data available [Amo+15].

**Prediction (i):**
they stand unmoved in their solitary grapes well as she are
that they re seeing of all the world without any effort to show themselves off and that no one will attempt to drive them from that position

**Ground truth (i):**
they stand unmoved in their solitary greatness well assured that they are seen of all the world without any effort to show themselves off and that no one will attempt to drive them from that position

**Prediction (ii):**
once more the singer plays in the lady's dance but on by one they fall asleep thou the drowsy music and then the singer steps into the ring and un locks the tower and kisses the upperors daughter

**Ground truth (ii):**
once more the singer plays and the ladies dance but one by one they fall asleep to the drowsy music and then the singer steps into the ring and unlocks the tower and kisses the emperors daughter

<table>
<thead>
<tr>
<th>Table 12: Two transcriptions predicted by the ASR system and their ground truths. Underlined sections of the prediction highlight differences to the ground truth.</th>
</tr>
</thead>
</table>

Some English words are pronounced very different from how are actually spelled. Requiring the ASR system to learn that the same characters in one word are pronounced differently in another word. Languages where the written form is more closely related to the actual pronunciation, like Spanish and to a lesser extent German, are domains where a system without LM is likely to perform better. [Rao+15]

In example (i), the model predicted “they re” instead of “they are”. Searching the merged training corpus for the term they’re, that is encoded as “theyre” returns 3,170 results and “they are” 2,447 results. This case shows a general problem of the merged corpus, because the individual corpora all use different standards for their labels. Since the TED-LIUM dataset contains live recordings, utterances like “I a a I got [...]” can occur and have to be recognized correctly. However, fillers like “uh” and “em” have to be ignored, as they are not labeled within the dataset.

To classify every utterance correctly the model needs to learn multiple ways to say the same thing (e.g. “AAA” and “triple A”). One of the most challenging cases are numbers. Here are just a few ways someone could say 1984: “one thousand nine hundred and eighty-four”, “nineteen eighty-four”, “nineteen hundred and eighty-four” or “one nine eight four”. And this list does not even contain slang. Furthermore, numbers rarely occur in the training data, making them a difficult task for the system. However, when an utterance contains numbers then all datasets transcribe it correctly. That is, in the way it is pronounced in the recording.
5.11.2 Continuous Speech

Removing examples that are longer than 15 seconds, in contrast to the current 17 seconds, would improve the training times and memory usage. This is because unrolling a RNN is a costly operation. Even though, the network should then have more trouble transcribing longer sequences. However, the maximum training sequence length is not reduced further, because the combined corpus is still slightly smaller than 1,000 hours. For the system to transcribe utterances of arbitrary length another modification is needed. For example some form of stitching that allows the system to transcribe partial sequences and then combine the partial transcriptions.

As mentioned in the Training Material section (4.1), approximately 292 hours of the Common Voice dataset are discarded for training, because most of them did not contain the optional information about the speaker’s accent. Even though accented speech is not one of the objectives for this thesis, it is possible that the limitations on the Common Voice corpus are too restrictive. It is possible that a larger training dataset with more variety could improve the overall results.

5.12 Summary

This section presented the results of various performed experiments on the baseline model and later on the extended model. It shows that the more refined MFCC input features have an advantage over their intermediate state (i.e. MELSPECS) if fed into a dense model (27.3% vs. 33.5% WER, table 7). It further shows that for a deeper network, with convolutional input layers, this advantage is barely noticeable (12.7% vs. 12.9% WER). From the three input normalization methods, only the local normalization shows an improvement for the baseline system, while local scalar and global normalization perform roughly on the same level as no normalization at all (table 8).

In addition to that, the baseline system is compared against the ASR system from Mozilla. The reference system, named Project DeepSpeech, is based on the Deep Speech (1) paper [Han+14], as well. It achieves a WER of 21.2% on the same test dataset, without the use of its external LM. The WER decreases to 10.4% with the LM enabled. Even though both systems are tested on the same test dataset, their training material was slightly different. For comparison, the best baseline system configuration achieved a WER of 27.3%.

The second part of the section evaluates results for the extended model. Test runs to measure the benefits convolutional input layers have over dense ones, show an overall decrease in WER independent of the normalization method used (table 9). While local (25.6% WER) and local scalar (25.6% WER) normalization now perform on the same level, global normalization (26.7% WER) is still slightly worse. The lack of normalized features (none), results in a noticeable increase in WER to 29.2% for the CNN.

An increase in RNN layers from one (25.6% WER), to up to five layers (15.8% WER, table 10) is evaluated. It shows, that the WER is decreasing noticeably with an increase in RNN layers. Lastly the effects of more sophisticated RNN cells, like the LSTM and GRU are evaluated. Showing that the computational more demanding LSTM outperform (12.6% WER) the best simple RNN (15.8% WER). GRU reveal some problems that are most likely caused by poor cell initialization, achieving only a WER of 19.1% using four layers (table 11).

The best preforming model is a 3c4r2d DNN, with bidirectional-LSTM cells. It uses local normalized MELSPECS input features and SortaGrad. The system achieves a mean ED of 4.5% and a WER of 12.6% on the test dataset.
6 Conclusion

Speech recognition on large vocabulary and noisy corpora is challenging for computers. Recent advances in end-to-end training have opened up new powerful ways for DNNs to transcribe speech. They no longer rely on handcrafted rules and reduce the complexity of the overall ASR system.

6.1 Summary

The main contribution of this thesis is the development and evaluation of an end-to-end trained ASR system that is based on the CTC cost function. To put the developed system into perspective, a brief history of speech recognition, as well as related end-to-end systems and research, is presented. To construct an ASR system, the required theory for the components that make up such a system are presented in section 2. Based on these components the construction of an ASR system, derived from Deep Speech (1) [Han+14], is described in section 4. A key component of the system is the CTC algorithm that allows the RNN to train on unaligned sequential data.

This ASR system is used to evaluate the effects of different input features on the WER, such as MELSPEC and MFCC. Tests show that the more refined MFCCs are better suited for dense input layers, while deeper RNNs with convolutional input layers are able to extract almost the same amount of information from the less refined MELSPECs. In addition to the different feature types, three variations of input normalization are evaluated. Out of those, the best performing normalization is calculated on a per component basis of the input time-series. However, examination of multiple training runs indicates that it is likely that a normalization on a per batch basis could outperform the local normalization. Bootstrapping the NN training by gradually increasing the example difficulty during the first epoch results in a generally more stable training. Which in turn allows for an increased LR, and therefore faster convergence of the model.

The designed baseline system achieves a WER of 27.3%, while the reference system Project DeepSpeech from Mozilla achieves a WER of 21.2% (10.4% with LM) on the same test corpus. For reference, the original Deep Speech system reports a WER of 16.0% on a different corpus, while also using a LM [Han+14]. This baseline system is then expanded on to further reduce the achieved WER. To increase the system’s ability to extract the best features from the input data, the first dense layers in the baseline system are replaced with convolutional layers. They are typically used in image recognition tasks. Tests show a noticeable decrease in WER, when compared to dense input layers.

To increase the amount of information the system can extract from the training data, the number of RNN layers is increased step by step, from one to up to five. Showing a decrease in WER for every bidirectional-RNN layer added, from 25.6% for a single layer down to 15.8% for five layers. However, using LSTM cells instead, no noticeable change in WER between four and five layers could be observed.

Reduction of the training times for each system tested is a key factor, therefore the simple RNN cells are used for most experiments. Nonetheless, there are more complex and powerful versions available, like the GRU and LSTM. These are evaluated against the most successful simple RNN version. The results show that the GRUs require optimal initialization, otherwise they do not perform well in this system. LSTM cells are able to achieve better results, when compared to the simple RNN version. While requiring significant more resources (i.e. time, memory), they solve most of the problems the RNN has (i.e. exploding and vanishing gradients).
and are able to better learn from longer sequences.

Baidu’s Deep Speech 2 model achieves a WER of 14.9% without the use of an external LM [Amo+15]. The best result, that the extended model achieves is a WER of 12.6%, however, both results are measured on different test datasets.

6.2 Limitations

The described system does have some limitations. Since it is a true end-to-end trained system, no external LM is used. Yet, as shown in the results for Mozilla’s Project DeepSpeech (section 5.5) and Baidu’s Deep Speech [Amo+15], the use of a LM can noticeably reduce the WER.

The compiled training dataset is free to use, which is great for everyone with the desire to train their own ASR system. But it is taken from several contexts (e.g. audio books and public speaking), preventing the system from focusing one a specific type of speech. With the combined training corpus containing 901 hours of speech being only of modest size, when compared to other systems.

The subpar performance of the GRU highlights the importance of a suitable NN parametrization. Because the results from different experiments should be comparable, the used parameters are not updated to better suit each experiment. This of course being less than ideal.

6.3 Practical Implications

Since the end-to-end design reduces the complexity of the overall system, a powerful ASR system can be constructed and trained by a single person. While the ability, provided by the CTC algorithm, to train on unaligned audio-transcription sequences, allows the integration of open speech corpora into the training material. These open corpora enable the construction of a large dataset, which is required to train a large vocabulary continuous speech recognition (LVCSR) system. The audio examples contain a large amount of variety, with different speakers and accents. This is necessary for future systems that intend to work for every speaker, all the time.

6.4 Future Work

While the enhanced system already achieves reasonable WERs, that allow understanding of most transcriptions; there are some aspects in which the system can be improved further. The three most promising areas are:

1. Since the CTC algorithm has problems developing a strong internal LM, while the attention mechanism is able to extract the necessary information to do so. A joined attention and CTC approach could be explored. This could combine the strength of CTC to learn the input to transcription alignments and the attention mechanisms ability to refine the alignments and to learn an internal LM.

2. For the system to be used as a production system the amount of training data has to increase. This can be done by incorporating additional corpora. In that case it appears reasonable to further normalize the different training examples. Both in audio quality and volume, as well as through unification of the used labels. Another option to increase the amount of training data, which other ASR systems already employ, is the artificial modification of training examples. For example by adding random noise or reverberation; or by shifting the pitch and playback speed.
3. This last point is not something that improves the model’s quality directly, but becomes more important with increasing system complexity. That is distributed training. While the described model is developed for a single GPU system, multi-GPU servers are getting more and more common. As the amount of experiments necessary to determine optimal hyperparameters, network layout, type of input, etc. increases, the required computation time also increases. While optimization and tuning of algorithms and their implementation sure is an option, it is often less expensive to just run the experiment on a new server with multiple GPUs instead.

Acknowledgments

The author especially wants to thank Markus Gilbert, Kathrin Ungru and Jürgen te Vrugt for their continued support in getting access to servers to run the larger experiments on. This thesis would not have been possible without their help. Yves-Noel Weweler for many helpful and inspiring discussions about the implementation and possibilities to improve the ASR system. He further wants to acknowledge his advisors Jürgen te Vrugt and Nikolaus Wulff for their guidance and support while writing this thesis.
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Traditional speech recognition system pipeline</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Bucketing boundaries</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>Common activation functions</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>Two-dimensional convolution</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>Unrolled recurrent neural network</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>Notation for recurrent neural network illustrations</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>Recurrent neural network cell</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>Long short-term memory</td>
<td>12</td>
</tr>
<tr>
<td>9</td>
<td>Gated recurrent unit</td>
<td>14</td>
</tr>
<tr>
<td>10</td>
<td>Probability distribution for each time-step</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>Connectionist temporal classification probability distribution</td>
<td>17</td>
</tr>
<tr>
<td>12</td>
<td>Probabilities for each time-step on a limited alphabet</td>
<td>18</td>
</tr>
<tr>
<td>13</td>
<td>Ordered probabilities for each time-step</td>
<td>18</td>
</tr>
<tr>
<td>14</td>
<td>All valid alignments including blank-labels</td>
<td>20</td>
</tr>
<tr>
<td>15</td>
<td>Connectionist temporal classification recursive algorithm</td>
<td>23</td>
</tr>
<tr>
<td>16</td>
<td>Connectionist temporal classification training progress</td>
<td>24</td>
</tr>
<tr>
<td>17</td>
<td>Modified beam search algorithm</td>
<td>25</td>
</tr>
<tr>
<td>18</td>
<td>Encoder-decoder</td>
<td>30</td>
</tr>
<tr>
<td>19</td>
<td>Example alignments for attention mechanism</td>
<td>31</td>
</tr>
<tr>
<td>20</td>
<td>Distribution of the audio examples length</td>
<td>37</td>
</tr>
<tr>
<td>21</td>
<td>Conversion process of speech, to mel spectrogram, to MFCC</td>
<td>38</td>
</tr>
<tr>
<td>22</td>
<td>Example plots of mel spectrogram and MFCC</td>
<td>39</td>
</tr>
<tr>
<td>23</td>
<td>Schematic of the architecture from the baseline model</td>
<td>41</td>
</tr>
<tr>
<td>24</td>
<td>Time-frequency convolutions</td>
<td>45</td>
</tr>
<tr>
<td>25</td>
<td>Baseline and extended model network architectures</td>
<td>46</td>
</tr>
<tr>
<td>26</td>
<td>Training progress</td>
<td>56</td>
</tr>
<tr>
<td>27</td>
<td>Individual dataset length distributions</td>
<td>69</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Training dataset composition</td>
<td>35</td>
</tr>
<tr>
<td>2</td>
<td>Training dataset transcription statistics</td>
<td>36</td>
</tr>
<tr>
<td>3</td>
<td>Feature extraction parameters</td>
<td>39</td>
</tr>
<tr>
<td>4</td>
<td>Baseline model composition and dimension</td>
<td>43</td>
</tr>
<tr>
<td>5</td>
<td>Hyperparameters</td>
<td>44</td>
</tr>
<tr>
<td>6</td>
<td>Convolutional layer parameters</td>
<td>45</td>
</tr>
<tr>
<td>7</td>
<td>Feature type results</td>
<td>49</td>
</tr>
<tr>
<td>8</td>
<td>Normalization results</td>
<td>50</td>
</tr>
<tr>
<td>9</td>
<td>Comparison between dense and convolutional layers</td>
<td>53</td>
</tr>
<tr>
<td>10</td>
<td>Results for increasing number of recurrent layers</td>
<td>54</td>
</tr>
<tr>
<td>11</td>
<td>Results for different recurrent neural network cell types</td>
<td>54</td>
</tr>
<tr>
<td>12</td>
<td>Example transcriptions</td>
<td>57</td>
</tr>
<tr>
<td>13</td>
<td>Dataset licenses</td>
<td>68</td>
</tr>
<tr>
<td>14</td>
<td>Confidence interval test run results</td>
<td>70</td>
</tr>
<tr>
<td>15</td>
<td>Results to determine the confidence interval</td>
<td>70</td>
</tr>
</tbody>
</table>
References


[Ng16] Andrew Ng. *As speech-recognition accuracy goes from 95% to 99%*. Dec. 15, 2016. URL: https://twitter.com/andrewyng/status/809579698883727360.


Appendices

A.1 Additional Corpora Information

All corpora used for the ASR system are licensed under a version of the Creative Commons (CC) license. Table 13 lists the individual datasets and their specific CC license.

<table>
<thead>
<tr>
<th>Name</th>
<th>License</th>
<th>Citation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common Voice</td>
<td>Public Domain Dedication</td>
<td>[Moz17a]</td>
</tr>
<tr>
<td>LibriSpeech ASR Corpus</td>
<td>Attribution</td>
<td>[Pan+15]</td>
</tr>
<tr>
<td>Tatoeba</td>
<td>Attribution</td>
<td>[Ho+06]</td>
</tr>
<tr>
<td>TED-LIUM v2</td>
<td>Attribution-NoCommercial-NoDerivs</td>
<td>[Rou+14]</td>
</tr>
</tbody>
</table>

Table 13: The used speech corpora with their associated Creative Commons (CC) license type.

Figure 27 shows the example length distributions of the four datasets that are used in the combined training dataset from figure 20a. The majority of the examples in the Common Voice (figure 27a) and Tatoeba (figure 27c) datasets have a length of about 3 seconds. While the LibriSpeech ASR Corpus (figure 27b) has an emphasis on long sequences, that often contain several sentences. Finally, the TED-LIUM (figure 27d) provides a broad spectrum of examples. They are distributed over the whole 0.7 to 17 seconds interval, on which the ASR system is trained for.
Figure 27: Audio example length distributions of the individual datasets that make up the combined train dataset.
A.2 Confidence Interval Estimations

This subsection contains background information for the calculated confidence intervals from section 5. Table 15 lists the results for the 20 individual test runs. They are performed with a reduced baseline system. Other than the regular baseline system, it has only 1,024 units per layer and a reduced beam search beam width of 512, while still using the same general architecture 3d1r2d. It is assumed that the samples are normally distributed. The equation for the standard deviation (STD) used is:

$$\sigma_\Delta (x_1, \cdots, x_N) = \sqrt{\frac{1}{N-\Delta} \sum_{i=1}^{N} (x_i - \bar{x})^2}$$  \hspace{1cm} (30)

where $\bar{x}$ is the samples mean and $N$ the number of samples. Confidence intervals for both the uncorrected ($\Delta = 0$) and the corrected ($\Delta = 1$) STD are stated in table 14. With only a single decimal place, both versions round to the same values. The confidence interval is calculated as follows:

$$\left[ \bar{x} - z^* \frac{\sigma_\Delta}{\sqrt{N}}, \bar{x} + z^* \frac{\sigma_\Delta}{\sqrt{N}} \right]$$  \hspace{1cm} (31)

with a critical value $z^* = 1.96$ for a two-sided confidence interval of 95%.

<table>
<thead>
<tr>
<th>Metric</th>
<th>$\Delta$</th>
<th>$\bar{x}$</th>
<th>$\sigma_\Delta$</th>
<th>Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>WER</td>
<td>0</td>
<td>71.4%</td>
<td>0.90%</td>
<td>[71.00%, 71.79%]</td>
</tr>
<tr>
<td>WER</td>
<td>1</td>
<td>71.4%</td>
<td>0.92%</td>
<td>[70.99%, 71.79%]</td>
</tr>
<tr>
<td>ED</td>
<td>0</td>
<td>29.4%</td>
<td>0.57%</td>
<td>[29.13%, 29.62%]</td>
</tr>
<tr>
<td>ED</td>
<td>1</td>
<td>29.4%</td>
<td>0.58%</td>
<td>[29.12%, 29.63%]</td>
</tr>
</tbody>
</table>

Table 14: Confidence interval’s bounds, mean $\bar{x}$ and standard deviation (STD) $\sigma_\Delta$ for varying degrees of freedom $\Delta \in \{0, 1\}$.

<table>
<thead>
<tr>
<th>Run</th>
<th>ED</th>
<th>WER</th>
<th>Run</th>
<th>ED</th>
<th>WER</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>29.0%</td>
<td>70.9%</td>
<td>11</td>
<td>29.4%</td>
<td>70.5%</td>
</tr>
<tr>
<td>2</td>
<td>30.1%</td>
<td>72.7%</td>
<td>12</td>
<td>29.0%</td>
<td>69.9%</td>
</tr>
<tr>
<td>3</td>
<td>30.0%</td>
<td>72.4%</td>
<td>13</td>
<td>29.2%</td>
<td>71.0%</td>
</tr>
<tr>
<td>4</td>
<td>29.7%</td>
<td>72.4%</td>
<td>14</td>
<td>29.4%</td>
<td>71.8%</td>
</tr>
<tr>
<td>5</td>
<td>29.0%</td>
<td>70.3%</td>
<td>15</td>
<td>29.5%</td>
<td>70.8%</td>
</tr>
<tr>
<td>6</td>
<td>30.1%</td>
<td>72.1%</td>
<td>16</td>
<td>29.9%</td>
<td>71.4%</td>
</tr>
<tr>
<td>7</td>
<td>29.5%</td>
<td>71.5%</td>
<td>17</td>
<td>29.6%</td>
<td>72.2%</td>
</tr>
<tr>
<td>8</td>
<td>27.5%</td>
<td>73.3%</td>
<td>18</td>
<td>29.6%</td>
<td>71.2%</td>
</tr>
<tr>
<td>9</td>
<td>29.1%</td>
<td>70.8%</td>
<td>19</td>
<td>29.0%</td>
<td>70.6%</td>
</tr>
<tr>
<td>10</td>
<td>29.1%</td>
<td>70.3%</td>
<td>20</td>
<td>29.8%</td>
<td>71.8%</td>
</tr>
</tbody>
</table>

Table 15: Word error rate (WER) and character based mean edit distance (ED) for the 20 test runs on a simplified model. The confidence intervals stated in section 5 are based on these results.
A.3 Server Setup

The following servers have been used for training:

- **COSY**
  - Intel Core i7-6700 at 3.40GHz
  - Nvidia GeForce GTX 1060, 6GB VRAM
  - 32GB RAM

- **V100(a)**
  - Intel Xeon E5-2660 v3 at 2.60GHz
  - Nvidia Tesla V100, 32GB VRAM
  - 128GB RAM

- **V100(b)**
  - Intel Xeon Gold 6126 CPU at 2.60GHz
  - Nvidia Tesla V100, 16GB VRAM
  - 128GB RAM

All test runs for the baseline model from section 4.3 are performed on the four available COSY servers. However, they are not able to run the improved versions from section 4.5 within a reasonable time, due to the additional RNN layers. The increased computational power that the V100(a) and V100(b) servers provide enable the larger networks to be trained and reduce the required training time.

It is worth mentioning that the V100(a) server has stability issues under high GPU loads. Causing the server to trigger its failsafe mode and requiring a manual restart afterwards. Resulting in several hours of lost computation time. The problems can be mitigated to some extent by reducing the batch size and overall network size. And furthermore reducing the Nvidia driver’s power limit to 130W down from 250W and enabling the driver’s persistence mode. This reduces the GPU’s temperature but also decreased the performance noticeably. Training a model with 5 bidirectional-LSTM layers at 150W completed 0.43 batches per second. Reducing the power further, down to 130W reduced this to 0.36 batches per second. Measurements with the same 5 layer bidirectional-LSTM and the GPU at full power could not be made.
Selbständigkeitserklärung

Ich versichere, die von mir vorgelegte Arbeit selbständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht.

Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen und ist noch nicht veröffentlicht worden. Ich bin mir bewusst, dass eine unwahre Erklärung rechtliche Folgen haben wird.

Ort, Datum

Unterschrift