Karlsruhe Institute of Technology

Character Based Language Modeling and Applications in Speech Recognition

Master Thesis
of

Thomas Zenkel

at the Department of Informatics
Institute for Anthropomatics and Robotics

First Reviewer:	Prof. Dr. A. Waibel
Second Reviewer:	Prof. Dr. W. Tichy
Advisors:	Dr. Sebastian Stüker
	M.Sc. Matthias Sperber
	Dr. Kevin Kilgour

Time Period: 9th December 2016 – 8th June 2017

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, den 8. Juni 2017

Thomas Zenkel

Abstract

Character based approaches have recently attracted a lot of interest for language modeling tasks as they remove some of the drawbacks of word based models. They do not make use of a fixed size vocabulary and it is thus straight forward to deal with an arbitrary text stream without any changes in the design of the model. Furthermore, downstream application like speech recognition and machine translation can apply their information already at the character level.

The length of character sequences is in general a lot longer than the length of a word sequence, which is a challenge for the underlying character based model. Variants of recurrent neural networks like the Long Short Term Memory (LSTM) are commonly used to tackle variable length inputs like character sequences. However, LSTMs still have problems to backpropagate the error over long input sequences and thus do not provide the perfect solution to learn long term dependencies.

We analyze the source of these drawbacks and design an architecture for recurrent neural networks. We evaluate this architecture on a popular data set for character based language modeling consisting of Wikipedia articles. Our architecture was able to outperform stacked Gated Recurrent Units as well as results of stacked LSTM networks reported in the literature.

We additionally apply character based language models to a speech recognition system. The acoustic component of the system is implemented by the Connectionist Temporal Classification (CTC), as this offers the possibility to directly predict the probabilities of character sequences. We combine the information of the acoustic component and the language model in a search to find the most reasonable character sequence. Since no component of this setup uses a vocabulary, we are able to perform open vocabulary speech recognition.

This system is evaluated on the Switchboard Telephone Speech Corpus. During testing our model we are able to correctly recognize words which did not appear in the training data. This confirms that the system is not only able to learn the words included in the training text, but also generalizes to previously unseen words. We are additionally able to improve the state of the art on the Switchboard corpus for character based, open vocabulary CTC speech recognition system.

Zusammenfassung

Graphembasierte Ansätze haben neuerdings großes Interesse für Sprachmodellierungsaufgaben erfahren, da sie einige Nachteile von wortbasierten Modellen beheben. Da sie kein statisches Vokabular verwenden ist es für sie möglich mit beliebigen Textdaten umzugehen ohne das Design des zugrunde liegenden Modells anzupassen. Außerdem können Anwendungen, wie zum Beispiel Spracherkennungssysteme oder Systeme zum automatischen Übersetzen, die Informationen des graphembasierten Sprachmodell bereits früher einbinden und müssen nicht erst auf das Ende eines Wortes warten.

Die Länge einer Graphemsequenz ist im Allgemeinen deutlich länger als die Länge einer Wortsequenz. Dies stellt eine Herausforderung für das zugrunde liegende graphembasierte Sprachmodell dar. Variationen von rekurrenten neuronalen Netzen wie das Long Short Term Memory (LSTM) werden häufig für Eingabesequenzen mit einer variablen Länge verwendet. Trotzdem haben LSTMs Probleme das Fehlersignal über lange Eingabesequenzen zu propagieren und stellen daher nicht die perfekte Lösung dar um Abhängigkeiten über längere Zeitschritte zu erlernen.

Wir anaylisieren die Gründe für diese Nachteile und entwerfen eine Architektur für rekurrente neuronale Netze. Wir evaluieren diese Architektur auf einem bekannten Datensatz für graphembasierte Sprachmodelle, welcher aus Wikipedia Artikeln besteht. Auf diesen Datensatz übertraf unsere Architektur sowohl die Ergebnisse von gestapelten Gated Recurrent Units sowie gestapelten LSTM Netzen, die in der Literatur publiziert wurden.

Außerdem wenden wir graphembasierte Sprachmodelle in einem Spracherkennungssystem an. Die akkustische Komponente unseres Spracherkennungssystem basiert auf der Connectionist Temporal Classification (CTC) Funktion, da es dies ermöglicht, direkt die Wahrscheinlichkeit von Buchstabensequenzen vorherzusagen. Die Informationen der akkustischen Komponente und des Sprachmodells werden in einer Suche zusammengefügt um die beste Buchstabensequenz zu finden. Da keine Komponente dieses Systems ein Vokabular verwendet, ermöglichen wir damit ein Spracherkennungssystem, das alle möglichen Wörter erkennen kann.

Dieses System wurde auf dem Switchboard Telephone Speech Korpus evaluiert. Während des Testens unseres Modells war es uns möglich Wörter korrekt zu erkennen, die nicht in den Trainingsdaten vorkamen. Dies bestätigt die Vermutung, dass wir nicht nur die Wörter des Trainingstextes lernen, sondern auch auf noch nicht gesehene Wörter generalisieren können. Zudem war es uns möglich mit dem beschriebenen Sprachmodell die aktuell besten Ergebnisse auf dem Switchboard Korpus für graphembasierte CTC Spracherkennungssysteme zu übertreffen, die nicht durch ein Vokabular beschränkt werden.

Contents

1 Introduction **1**
1.1 Outline . . . 2

2 Background: Neural Networks **5**
2.1 Architectures . . . 5
2.1.1 Feedforward Neural Networks . . . 6
2.1.2 Recurrent Neural Networks . . . 7
2.2 Training . . . 9
2.2.1 Backpropagation . . . 10
2.2.2 Backpropagation Through Time . . . 12
2.2.3 Loss . . . 13
2.2.4 Optimizers . . . 14
2.2.5 Regularization . . . 15
2.3 Summary . . . 16

3 Background: Language Modeling **17**
3.1 Word Based Models . . . 17
3.1.1 Statistical Models . . . 18
3.1.2 Neural Models . . . 19
3.2 Character Based Models . . . 20
3.2.1 Statistical Models . . . 20
3.2.2 Neural Models . . . 21
3.3 Comparison . . . 22

4 Improving Character Based Language Models **25**
4.1 Related Work . . . 25
4.2 Architecture . . . 26
4.3 Experiments . . . 27
4.4 Conclusion . . . 29

5 Applications in Speech Recognition **31**
5.1 Introduction . . . 31
5.2 Background: Approaches for Speech Recognition . . . 32
5.2.1 Statistical HMM based ASR . . . 32
5.2.2 Connectionist Temporal Classification . . . 33
5.3 CTC Beam Search Algorithm . . . 37
5.3.1 CTC Beam Search Experiments . . . 38
5.3.2 CTC Beam Search Error Analysis . . . 39
5.4 Conclusion . . . 41

6 Conclusion **43**
6.1 Future Work 44

Bibliography **47**

List of Figures

2.1 Visualization of different activation functions 6

2.2 Visualization of a LSTM 8

2.3 Computation Graph of a simple neural network 12

2.4 Visualization of an unrolled recurrent neural network 13

4.1 Visualization of an unrolled Hierarchical Clockwork RNN 27

4.2 Pretraining of the first layer of the HCRNN 28

4.3 Pretraining of the first two layers of the HCRNN 28

5.1 Diagram of a HMM based speech recognition system 33

5.2 Diagram of a CTC based speech recognition system 34

5.3 Probability matrix of an utterance with the labeling “yeah” 35

5.4 Diagram of a CTC based speech recognition system with a language model 36

List of Tables

3.1 Test perplexities on the one billion word benchmark 22

3.2 Bits per character on the test set of Enwik8 22

4.1 Bits per Character on the Wikipedia Dataset Enwik8 for different architectures 28

5.1 Comparison of Word Error Rates for different decoding approaches . 39

5.2 Example output of a cherry picked utterance 39

5.3 Correctly recognized words that were not present in the training corpora 40

5.4 Insertion Rate, Substitution Rate and Deletion Rate for multiple decoding algorithm using character based AMs 41

1. Introduction

Language modeling tasks were almost entirely focused on word based models. The prevailing approach was to predict the next word based on a fixed context of N words. Smoothing techniques were used to generalize to unseen contexts or unseen words in a specific context. These techniques rely on information from a context of less words than the original model. However, the context only consisted of a small number of words due to the memory requirements to store the language model.

Feedforward neural networks were partly able to avoid using smoothing techniques, because they were able to learn relations between the meaning of different words. This enabled them to generalize and to predict probabilities for an unseen context. However, feedforward neural networks still rely on a fixed size context. Due to advances in recurrent neural networks (RNNs) architectures, word based models with a theoretically infinite context became competitive. By storing information of the previous words in its hidden state, RNNs are able to remember important information to predict the next words.

More recent architectures like the Long Short Term Memory (LSTM) also made character based language models a more reasonable option. Due to improvements to backpropagate the error signal between distant events LSTMs are able to learn long term dependencies more efficient. Because character sequences are considerable longer than word sequences, this lead to significant improvements in the performance of character based language models.

However, character based language models are still not adopted widely for downstream applications like automatic speech recognition. Traditional HMM based speech recognition applications almost completely rely on word based approaches. In these approaches the most probable word sequence is searched in a decoder based on the information of a phoneme based acoustic model (AM) and a word based language model (LM). To map a sequence of phonemes to a word a pronunciation lexicon is applied.

Therefore it made sense to rely on word based LMs. Because the LM was typically implemented as a count based N-gram model, a LM query only consisted of a memory read and was quite fast compared to the calculation needed to estimate the

probabilities of the AM. This lead to decoding approaches were LM lookaheads were performed as early as possible [SMFW01].

A more recent approach is the Connectionist Temporal Classification (CTC) [GFGS06]. This approach is able to map a variable length input sequence like speech features to a character sequence. The core assumption of the model is the conditional independence of its outputs. This makes it possible to efficiently calculate a probablity matrix of each utterance which can be used during decoding. Thus querying the AM becomes considerably cheap and only consists of a memory read. When decoding the CTC based acoustic component with a character based LM, the LM becomes the bottleneck. Especially neural approaches like RNN networks are computational very expensive.

While there has been a few publications on integrating character based, neural LMs within the CTC framework, these papers did not primary focused on using computational expensive and high performing neural LMs [ZYDS16, MXJN15]. While recent character based LM are almost entirely focused on LSTM based LMs, [ZYDS16, MXJN15] do not use or do not report large gains when using neural networks for their character based LMs.

The goal of this thesis is two fold. First of all we want to analyze the different character based LMs and compare them to word based approaches. By understanding the shortcomings of the recently used neural models we try to implement an improved neural network architecture for long input sequences like characters. The main goal of this thesis is to implement a speech recognition system which uses a character based language model. We want to keep the main advantages of a character based model, that is no need to have a fixed vocabulary of output words and the possibility of providing the LM information already at the character level.

1.1 Outline

In chapter 2 we describe the most relevant background on neural networks. We both describe feedforward neural networks as well as recurrent neural networks, which are able to process variable length input sequences. We also discuss the process of training a neural network, which involves the backpropagation of the error signal, updating the parameters of the model and regularizing the neural network.

We provide background for different language model approaches in chapter 3. We describe count based N-gram LMs used to predict words. Afterwards we focus on neural approaches that are able to provide a longer context. For character based models at first we focus on count based models which are heavily used in compression algorithms. We conclude this chapter with state of the art RNNs used for character based LMs.

Chapter 4 introduces a new neural network architecture designed for long input sequences. This architecture is evaluated on a character based language modeling dataset.

We include a character based LM in a CTC speech recognition system in chapter 5. We combine the information of the LM and the acoustic component in a straight forward beam search. This approach enables a purely character based model which can produce words without the restriction of a fixed vocabulary. We test this procedure

by including a state of the art neural LM, which is based on the LSTM network. This approach is evaluated for English on the Switchboard Telephone Speech Corpus.

In chapter 6 we conclude this thesis with a short summary and ideas for future work.

2. Background: Neural Networks

Neural networks are computational models which are used in many areas of machine learning, like speech recognition [WHHS+89], machine translation [BaCB14] and image recognition [RDSK+15]. In this chapter we will provide the theoretical background of neural networks by formalizing them mathematically.

Neural networks process their inputs by applying a series of mathematical functions. Some of these functions depend on a large number of parameters, which are called weights. The weights are learned during a training phase. During training one wants to find weights, which lead to a certain behavior of the neural network. We will discuss supervised training in this chapter, so we will define the desired behavior by a set of inputs and their desired outputs.

In the first part of this chapter we will talk about some well known architectures of neural networks. Neural networks can be divided in groups of different architectures based on the combination of mathematical functions they use. We will distinguish feedforward networks, which are used to process a fixed size input, and recurrent neural networks, which are used to process a series of inputs. In the second part of this chapter we will describe the process of training the network. This process tries to find a set of weights, that lead to the desired behavior of the neural network.

2.1 Architectures

In this section we will review some well-known architectures of neural networks. Before doing that, let us define some basic notations. First of all we will write vectors as lowercase characters, e.g. x, and matrices as uppercase characters, e.g. X. The weights of the neural networks will be denoted by weight matrices W, and bias vector b. The subscript of these matrices will serve as an identifier that let's us differentiate multiple matrices. When superscripts are used, they refer to the time step of the vector. So x^t denotes the value of the vector x at time step t. Functions applied to vectors, e.g. $\sigma(x)$, are applied element-wise. $[x, y]$ will denote the concatenation of two vectors. $x \odot y$ denotes the hadamard product of two vectors of the same dimension.

2.1.1 Feedforward Neural Networks

First of all we will discuss feedforward networks. A feedforward network consist of one or multiple layers. Each layer processes an input $x \in \mathbb{R}^n$ and calculates an output $y \in \mathbb{R}^m$. A learned weight matrix $W \in \mathbb{R}^{m \times n}$ and a bias vector $b \in \mathbb{R}^m$ are used to transform the input. An activation function g, which is applied element wise, calculates the final output vector y:

$$y = g(W \cdot x + b) \tag{2.1}$$

This function implements one layer of the neural network. The output of one layer is used as the input of the following layer. Also notice that the input vector x and the output vector y of a layer can have different dimensions. Neural networks with a large number of layers are also sometimes called "deep" neural networks.

An important part of each layer is the activation function g. Some popular activation functions often used in intermediate layers of a neural network are depicted in figure 2.1. The sigmoid function $\sigma(x)$ maps the input to a value between 0 and 1. The hyperbolic tangens $\tanh(x)$ maps its input to a number between -1 and 1, while the rectified linear unit $ReLU$ outputs unbounded positive values. Another frequently used activation function is the softmax function s:

$$s(x_i) = \frac{e^i}{\sum_{j=1}^{k} e^j} \tag{2.2}$$

It maps a k dimensional vector $x \in \mathbb{R}^k$ to a probability distribution and is often used for classification task in the last layer of the neural network.

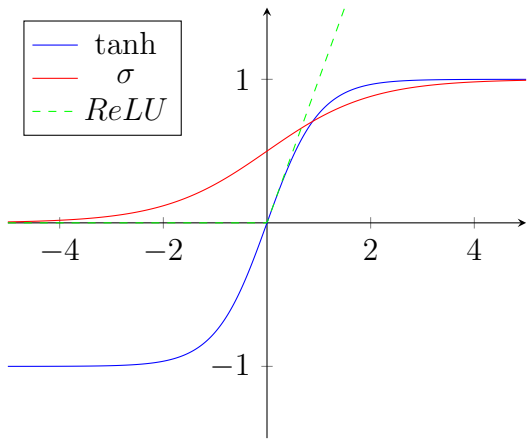

Figure 2.1: Visualization of different activation functions: hyperbolic tangens tanh, sigmoid σ and Rectified Linear Unit $ReLU$

An interesting property of feedforward neural networks is their ability to approximate mathematical functions. Given a nonconstant, bounded and monotonically-increasing continuous activation function g a feedforward network with a single hidden layer can approximate any function on compact subsets of $\mathbb{R}^n$ with a finite number of hidden units [HoSW89]. This property is also refered to as the universal approximation theorem.

While this property is theoretically interesting, in practice single layer neural networks are rarely used to produce competitive results. The main problem is to find the right set of weights during training to approximate a function or learn a task like recognizing speech. This task becomes significantly easier, if the number of weights gets smaller. This is an important motivation to introduce weight sharing.

Suppose we have different input vectors x^1 and x^2, which describe the same training example. Instead of learning two separate weight matrices W_1 and W_2, we can use a single weight matrix $W = W_1 = W_2$. This makes sense if the inputs vectors x^1 and x^2 are two different features, which are observed at slightly different moments in time. Another example would be that x^1 and x^2 are features which describe the pixels at different positions in the same picture.

Weight sharing is used in time delay neural networks (TDNNs) to classify phonemes for speech recognition [WHHS+89]. Because of sharing the weights at different time positions, TDNN units are able to recognize features independent of their time-shift. Another example for image recognition are convolutional neural networks (CNN), which apply the same weights at different positions within an image and are therefore shift invariant. CNNs were successfully applied for large scale image recognition benchmarks [KrSH12] and were recently also applied to sequence to sequence tasks like machine translation [GAGY+17].

2.1.2 Recurrent Neural Networks

Recurrent neural networks (RNNs) deal with a sequence of inputs $x^1, \ldots, x^T$. A RNN processes the sequence of inputs one input at a time. In contrast to a feedforward neural network it is able to store information of previous inputs by keeping a hidden state. We will denote the vector representing the hidden state as h. The hidden state is normaly initialized with a 0 vector. One of the first RNNs was the Elman network [Elma90]. A single layer of an Elman network can be implemented as follows:

$$h^t = \tanh(W_h \cdot [h^{t-1}, x^t] + b_h) \tag{2.3}$$

In the Elman network the hidden state h^t both stores the information of the previous inputs and serves as the output of the layer. The hidden state at time step t is calculated based on the previous hidden state h^{t-1} and the input vector at the current time step x^t. Note that the hidden state h gets multiplied with a weight matrix at every time step. We will discuss the consequences of this in section 2.2.2 when talking about training the neural networks.

A more sophisticated version of RNNs is the Long Short Term Memory (LSTM) [HoSc97]. An important difference is that in an LSTM the hidden state does not get

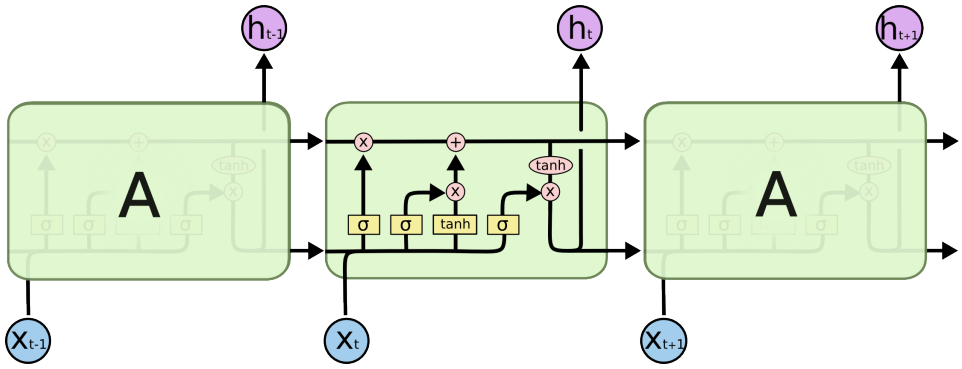

Figure 2.2: Visualization of a LSTM. Source: [Olah15]

multiplied by a matrix at every time step. But let us look at a layer of the LSTM network first:

$$
\begin{aligned}
f^t &= \sigma(W_t \cdot [h^{t-1}, x^t] + b_f) \\
i^t &= \sigma(W_i \cdot [h^{t-1}, x^t] + b_i) \\
o^t &= \sigma(W_o \cdot [h^{t-1}, x^t] + b_o) \\
z^t &= \tanh(W_z \cdot [h^{t-1}, x^t] + b_z) \\
c^t &= f^t \odot c^{t-1} + i^t \odot z^t \\
h^t &= o^t \odot \tanh(c^t)
\end{aligned}
\tag{2.4}
$$

In the LSTM network we differentiate between the context vector c^t, which saves the information of previous inputs and the output of the layer h^t. At every time step t we first update the context vector c^t. To do this we calculate two vectors f^t and i^t of numbers between 0 and 1. These vectors determine how much of the previous context vector c^{t-1} should be forgotten and how much should be added to each context unit. We also calculate a vector z^t which determines what should be added to the previous context vector c^{t-1}. Based on this information we update the context vector.

Additionally we calculate the vector o^t, which determines how much of the context vector should be outputted. The updated context vector c^t gets squashed with the tanh function and multiplied with the vector o^t. One motivation for squashing the context vector c^t is that, in theory, the context vector can grow linearly and is unbounded. The network can learn weights that lead to high values of the forget gate f^t and high values of the input gate i^t. That situation will make it possible that at each time step the values of the context vector can get bigger. A visualization of the LSTM layer is provided in figure 2.2.

A popular variation of the LSTM layer is the Gated Recurrent Unit (GRU) [CVMGB+14]. A GRU has significantly less parameters than the LSTM. This is achieved with a few modifications. First of all the forget gate and the input gate are coupled: $f^t = 1 - i^t$.

In addition the output and the context vector of the network are the same: $h^t = c^t$. These modifications lead to the following equations for a GRU layer:

$$\begin{aligned} i^t &= \sigma(W_i \cdot [h^{t-1}, x^t] + b_i) \\ r^t &= \sigma(W_o \cdot [h^{t-1}, x^t] + b_o) \\ z^t &= \tanh(W_z \cdot [r \odot h^{t-1}, x^t] + b_z) \\ h^t &= (1 - i^t) \odot h^{t-1} + i^t \odot z^t \end{aligned} \tag{2.5}$$

Also notice that we do not use an activation function to squash the hidden state h^t. As the forget gate f^t and the input gate i^t are coupled, the hidden state is bounded and can not get bigger than one. That is the main reason that no activation function is needed to squash the hidden state.

LSTM and GRU layers are widely used recurrent layers and implemented in many popular neural network frameworks [NDGM+17b, AABB+16]. While we presented the most popular variants of GRU and LSTM layers, the different frameworks use slightly different equations and implementations. However, in terms of performance on different benchmarks the different variations of LSTMs are quite similar.

[JoZS15] and [GSKS+16] tested different recurrent architectures. Starting with LSTM and GRU layers [JoZS15] randomly replaced, removed or added activation functions at appropriate parts of the computation graph of their current recurrent network. Different element wise operations (multiplication, addition and subtraction) were also considered. While these papers showed that LSTM and GRU layers are competitive to all other modifications found during the architecture search, a few other relevant conclusions are:

- Coupling the input and forget gate (f^t and i^t) simplifies the layers without hurting performance
- The forget gate seems to be a crucial component of the LSTM network
- Whenever the cell state (c^t) is unbounded, an output activation function is important to improve performance
- Hyperparameter interactions are quite small and the hyperparameters can be tuned independently to improve performance
- Initializing the forget bias of the LSTM to a large number is significant in all tested benchmarks

2.2 Training

We will now discuss how to find the parameters, specifically the values of the weight matrices W and bias vectors b of the neural model. The process of searching for appropriate parameters is called training the neural network or learning the parameters.

We only deal with supervised learning in this section. That means that during training for each input vector x we already know the desired output vector y^*. We call a input vector and its corresponding output vector training example. The set of training examples is also referred to as training data. During training we want to find weights that fit our training data.

We will first talk about an algorithm called backpropagation that is able to update our weights based on the training data in a desired direction. Afterwards we deal with the problem of applying backpropagation to recurrent neural networks. We will define different function that are able to calculate an error or loss based on the training data. Subsequently we deal with optimizers, which are designed to use backpropagation to minimize the error on the training set efficiently. The section concludes with regularization strategies that prevent that the neural network is able to learn the training data by heart.

2.2.1 Backpropagation

For supervised learning a core component to find a good set of weights is the backpropagation algorithm. Assume that we already have a set of values for all weight matrices and bias vectors of the neural network. Then we can calculate the output of the neural network given an input x. We will call the output y. Suppose that we have some kind of error or loss function that let's us calculate a loss given the actual output vector y and the desired output y^*:

$$l(y^*, y) \in \mathbb{R}_{\geq 0} \tag{2.6}$$

Based on the current parameters and a training example the backpropagation algorithm calculates a direction for each parameter of the neural network. If we update the value of the parameters slightly in this direction we can optimize the loss function. We will now show how the backpropagation algorithm calculates these directions.

To demonstrate the key principles of the backpropagation algorithm we will adapt an example of [Neub17]. Suppose we have this simple two layer network:

$$\begin{aligned} h' &= W_{xh} \cdot x + b_h \\ h &= \tanh(h') \\ y &= w_{hy} \cdot h + b_y \end{aligned} \tag{2.7}$$

Note that the first layer consists of a weight matrix W_{xh} and a bias vector b_h, while the second layer consists of a weight vector w_{hy} and a bias scalar b_y. So with this two layer network we will map an input vector x to a scalar y. For each training example x, y^*, we define a loss function in the following way:

$$l(y^*, y) = (y^* - y)^2 \tag{2.8}$$

A convenient representation of a neural network is a computation graph. The computation graph of this network is depicted in figure 2.3. The first graph shows the neural network, the second one additionally adds the loss function.

So suppose we have a training example x, y^*. We can now calculate the output y of the network. Then it is trivial to calculate the loss, in this case the squared error. This is also referred to as the forward calculation of the network, since we start at the beginning of the computation graph with an input x and iteratively apply different functions until we reach the end.

We want to calculate now how each node or parameter in the computation graph affects the loss y. For this reason we want to calculate the partial derivatives of the loss l with respect to each parameter p: $\frac{dl}{dp}$

To do this we apply the chain rule multiple times. Suppose we have a variable z that depends on the variable y. Furthermore the variable y depends on the variable x. Then we the chain rule let's us calculate the derivative of z with respect to x as follows:

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx} \tag{2.9}$$

Applying the chain rule we can calculate each derivative :

$$\begin{aligned}
\frac{dl}{db_y} &= \frac{dl}{dy} \cdot \frac{dy}{db_y} \\
\frac{dl}{dw_{hy}} &= \frac{dl}{dy} \cdot \frac{dy}{dw_{hy}} \\
\frac{dl}{dh'} &= \frac{dl}{dy} \cdot \frac{dy}{dh} \cdot \frac{dh}{dh'} \\
\frac{dl}{db_h} &= \frac{dl}{dy} \cdot \frac{dy}{dh} \cdot \frac{d_h}{d'_h} \cdot \frac{dh'}{db_h} \\
\frac{dl}{dW_{xh}} &= \frac{dl}{dy} \cdot \frac{dy}{dh} \cdot \frac{d_h}{d'_h} \cdot \frac{dh'}{dW_{xh}}
\end{aligned} \tag{2.10}$$

Notice that at each layer we can reuse the results of the previous node. When applying the backpropagation algorithm we start at the end of the graph. By iterating through each node we can calculate the derivative of the loss with respect to each parameter. So we can efficiently reuse the results of the previous nodes.

Notice that we only deal with a single scalar as an output in this particular example. When dealing with a vector of outputs, we normally sum the loss of each scalar of the output vector to get the total loss. In common neural networks each weight can affect all output units. That's why we have to calculate the loss with respect to each output unit for each parameter. Thus the computational complexity for performing backpropagation grows linearly in the number of output units. However, there are methods which only calculate the derivative to a subset of the output units like Noise Contrastive Estimation [GuHy10] and importance sampling [BeSo03]. These models make sense for word based neural language models with a large output vocabulary.

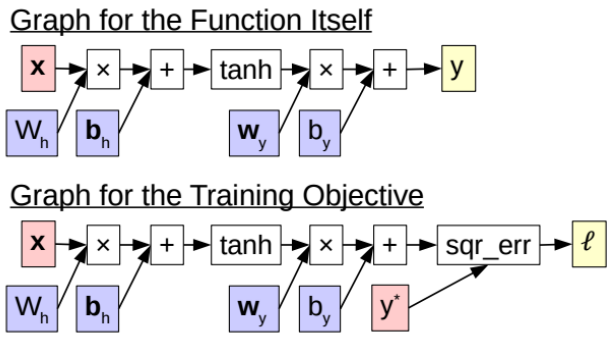

Figure 2.3: Computation Graph of a simple neural network. Source: [Neub17]

2.2.2 Backpropagation Through Time

We now deal with the problem of how to perform backpropagation for recurrent neural networks (RNNs). The key insight for this is that we can unroll the RNN in time. Unrolling a RNN layer is depicted in figure 2.4.

An unrolled RNN layer can be compared to a feedforward network consisting of many layers. However, a key difference of the unrolled RNN is that at each time step the same weight matrices are applied to transform the input and the hidden states. At times RNNs are also called "deep" in time.

We will now perform backpropagation and discuss the resulting gradients. For this purpose we will use the Elman RNN of the previous chapter, but split the weight matrix in a recurrent matrix R and the matrix W, which transforms the input. The equation of the Elman RNN is then written as follows:

$$h^t = \tanh(W \cdot x^t + R \cdot h^{t-1} + b) \tag{2.11}$$

To calculate the gradient we adapt the equations of [ZSKS16]. The total loss L will be summed over all time steps T. We calculate the derivative of the loss L with respect to the parameters θ as follows:

$$\frac{dL}{d\theta} = \sum_{1 \leq t_2 \leq T} \frac{dL^{t_2}}{d\theta} = \sum_{1 \leq t_2 \leq T} \sum_{1 \leq t_1 \leq t_2} \frac{\partial L^{t_2}}{\partial h^{t_2}} \frac{\partial h^{t_2}}{\partial h^{t_1}} \frac{\partial h^{t_1}}{\partial \theta} \tag{2.12}$$

We now can take a closer look at how to calculate the transport of the error between the time steps t_1 and t_2:

$$\frac{\partial h^{t_2}}{\partial h^{t_1}} := \prod_{t_1 < t \leq t_2} \frac{\partial h^t}{\partial h^{t-1}} = \prod_{t_1 < t \leq t_2} R^T diag[tanh'(R \cdot h^{t-1})] \tag{2.13}$$

The product in the previous equation can lead to vanishing or exploding gradients. First of all notice that $0 \leq \tanh' \leq 1$. This biases vanishing gradients since $\lim_{n \to \infty} x^n = 0$ for $0 \leq x < 1$. On the other hand the eigenvalues of R are crucial, too. High eigenvalues can lead to exploding gradients and tiny eigenvalues in turn can lead to vanishing gradients.

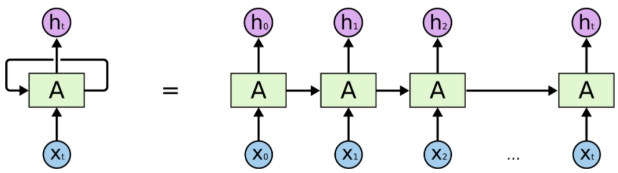

Figure 2.4: Visualization of an unrolled recurrent neural network. Source: [Olah15]

There are different techniques to mitigate this issue. An easy trick is to clip the gradient. Initializing the recurrent matrix R with the identity matrix and small random values on the off diagonal is another approach [LeJH15]. This leads to eigenvalues close to 1 at the start of training. [LeJH15] additionally used Rectified Linear Units as their activation function, which have constant gradients of 1 or 0.

The other common way is to have additive connections instead of recurrent matrices to change the state of the RNN layer. This is implemented in LSTM and GRU layers. For the following argument consider a simple additive connection of the hidden state h of the recurrent unit (for simplicity we neglect the input vector in the following example):

$$h^t = a \cdot h^{t-1} + R \cdot h^{t-1} \tag{2.14}$$

We use a scalar $a > 0$ and the recurrent weight matrix R in this equation. This can be rewritten as a single matrix multiplication by using the identity matrix I in the following way:

$$h^t = a \cdot h^{t-1} + R \cdot h^{t-1} = (a \cdot I + R) \cdot h^{t-1} = R' h^{t-1} \tag{2.15}$$

These additive connections favor a structure with positive and relatively high values on the diagonal of the recurrent weight matrix R'. Therefore additive connections can mitigate the issue of exploding or vanishing gradients.

In spite the success of RNNs like LSTMs and GRUs, it still seems to be difficult to learn long term dependencies using backpropagation and gradient descent. [BeSF94] makes the theoretical argument, that gradient descent becomes increasingly inefficient when the temporal span of the dependencies increases.

2.2.3 Loss

We will now define loss functions, which we try to optimize based on the training set. In the following we will denote an training example as $x \in \mathbb{R}^n, y^* \in \mathbb{R}^m$ and the output of our model given x will be denoted as $y \in \mathbb{R}^m$.

One common loss function is the mean squared error (MSE) which is defined as follows:

$$L_{MSE} = \sum_{i=1}^{m} (y_i - y_i^*)^2 \tag{2.16}$$

However, the MSE loss is not a good choice for classification problems, where the elements of the target vector y^* are either 0 or 1. For classification problems a reasonable loss function is the cross entropy (CE) loss:

$$L_{CE} = -\sum_{i=1}^{m} y_i^* log(y_k) + (1 - y_i^*) log(1 - y_k) \tag{2.17}$$

2.2.4 Optimizers

We will now discuss the process of updating the parameters of the neural network. Backpropagation allows us to calculate the gradients $\frac{dL}{d\theta}$ of the loss with respect to all parameters. We use the gradients to optimize our loss function. This subsection introduces some of the most important techniques to do so. We will refer to these techniques that optimize the loss function as optimizers.

Usually the parameters are optimized based on a fixed training set of multiple examples x and their labels y^*. One method is to pick a random example from the training set and update the parameters θ as follows:

$$\theta_i = \theta_{i-1} - \eta \frac{dL}{d\theta} \tag{2.18}$$

First of all notice that we use a learning rate $\eta > 0$, which let's us influence the size of the parameter update. To determine a good learning rate is a difficult problem and normally depends on your training data and can also depend on the progress of the training.

This method is called stochastic gradient descent (SGD), because a training example is stochastically picked and you follow the gradient to minimize the loss function. Picking an example at random is usually implemented by shuffling the training set at each iteration. However, updating the parameters after each example leads to a high fluctuation of the value of the loss function.

Mini batch gradient descent mitigates this problem by calculating the gradient for a batch of multiple examples. Afterwards the parameters get updated in the same way as for SGD. This leads to a parameter update which is more stable and additionally provides us with the possibility to parallelize the calculation of the gradients within a batch. As the parameters θ do not change within a batch, we can calculate the gradients of each example in parallel. This leads to significant speed improvements, especially on modern GPUs.

The problem on how to choose a proper learning rate η remains. In general a high learning rate leads to fast progress, but might not be able to find a local minimum. One common approach is to start with a relatively high learning rate that decreases during the training of the network.

However, there are also more sophisticated methods. One of them is to use SGD with momentum [Qian99]. When doing this we additionally use the information of the past gradients:

$$\begin{aligned} m_t &= \gamma \cdot m_{t-1} + \eta \frac{dL}{d\theta_{t-1}} \\ \theta_t &= \theta t - 1 - m_t \end{aligned} \tag{2.19}$$

The momentum term m keeps track of the past gradients. The weight of a past gradient is exponentially decaying determined by the scalar $\gamma \in [0, 1)$. This method leads to larger steps in parameter space, if the gradients at continuous time steps are pointed in the same direction.

Adaptive moment estimation, or Adam, [KiBa14] keeps both track of an exponentially decaying average of past gradients and of past squared gradients. In the following we use the hyperparameters $\beta_1, \beta_2, \epsilon \in [0, 1]$. b^t denotes b to the power of t. Adam then calculates the parameter update as follows:

$$\begin{aligned} g_t &= \frac{dL}{d\theta_{t-1}} \\ m_t &= \beta_1 \cdot m_{t-1} + (1 - \beta_1) g_t \\ v_t &= \beta_2 \cdot v_{t-1} + (1 - \beta_2) g_t \odot g_t \\ \hat{m}_t &= \frac{m_t}{1 - \beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1 - \beta_2^t} \\ \theta_t &= \theta_{t-1} - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \end{aligned} \tag{2.20}$$

m and v get initialized to the 0 vector. However, this biases m and v towards 0 at the beginning of training, especially if β_1 and β_2 are close to 1. $\hat{m}_t$ and $\hat{v}_t$ are calculated to counteract this issue.

A comprehensive summary of different optimizers is provided in [Rude16].

2.2.5 Regularization

Until now we only discussed minimizing a loss function based the training data. This can lead to parameters that overfit on the training data and do not perform good on similar, previously unseen data sets. However, our goal is to find parameters that generalize to unseen data.

In most of the cases this is solved by splitting our labeled data into multiple data sets: the training data, the validation data and the test data. The training data is used to calculate the gradients and update the parameters of our model. We keep track of the loss on the validation set without doing any parameter updates. If the loss on the validation set does not improve any more, we can stop training or lower the learning rate η.

It is also common to select hyperparamaters, like the network architecture or the choice of the optimizer, based on the loss measured on the validation set. After we determined all our hyperparameters and trained the neural network, the performance is tested on the test data.

Additional to this procedure there are a few other methods to prevent the neural network from overfitting on the training data. Popular ones include dropout [HSKS+12] and batch normalization [IoSz15].

2.3 Summary

This chapter introduced neural networks. Different architectures like feedforward and recurrent neural networks were defined. Afterwards we discussed the process of training the neural network. The whole process of calculating the gradients, defining a loss function, updating the parameters and trying to regularize the network was described. We additionally focused on why problems, like vanishing and exploding gradient, are an issue when training neural networks. In the remainder of this thesis this will be the foundation for successful results in language modeling applications.

3. Background: Language Modeling

In this chapter we will review different methods to create language models, which are an important component of automatic speech recognition systems. Language models discussed in this chapter assign a probability to a text segment. They thus are able to differentiate between probable and improbable transcriptions and are able to guide speech recognition systems towards creating linguistically sound outputs. A trivial example to emphasize the importance of a language model are homophones - words that are pronounced the same but differ in meaning and may also differ in their spelling. In contrast to the acoustic model the language model should for example be able to distinguish the words “to”, “too” and “two” based on their current context and assign reasonable probabilities to them.

This chapter emphasizes the procedures used in real world language models while also citing the most relevant theoretical work. We cover word based models at first and subsequently discuss characters based models. Each of these parts will describe statistically motivated approaches and models based on neural networks. The chapter concludes with a comparison of word and character based approaches.

3.1 Word Based Models

To discuss word based models we will first define the concept of a word a bit more formally. We start off with a set of basic units which we will call characters. In the following we make the assumptions, that the size of our character set will be quite small, say below 1000. This assumptions is reasonable for a number of languages like English and German. A word is then the concatenation of one or more characters. Due to the theoretically infinite number of possible words, most word based language models make use of a vocabulary - a set of words - which restricts the possible character sequences.

To create a language models usually large text corpora are used. For word based models it is evident that you have to preprocess the text to get a sequence of words. In many applications punctuation marks are removed, the text gets lowercased and

divided into sentences or utterances. Usually sentences are tokenized into a sequence of words based on the occurrence of whitespaces. If the vocabulary is not existent yet, the most frequent words of the text can be selected as the vocabulary. All words which are not in the vocabulary are then replaced by a symbol representing an unknown word. Additionally sentences are padded with symbols representing the start and end of the sentence.

While these preprocessing steps are reasonable for most applications, one should keep in mind that the preprocessing steps change the text corpus and the subsequent models estimate the probability distribution based on the altered corpus.

3.1.1 Statistical Models

Statistical language models represent the traditional variant of capturing linguistic information. Most of these models are count based. To predict the probability of a word given its previous words, the occurrences of the word in the specific context in the corpora are counted. The context is usually truncated to a specific number of words, since for a fixed corpus and an infinite context the counts are not high enough to confidently predict a probability. Also note that we only consider the context within a sentence.

So let's say we have a sentence $e = e_1, \ldots, e_n$ consisting of the sequence of n words. Let $e_i^k = e_i, \ldots e_k$ with $i \leq k$ denote a part of this sequence and $c(e_i^k)$ denote the count of this sequence in the corpus. Then we can estimate the probability of a word given its context of $N-1$ words as follows:

$$P(e_t|e_1^{t-1}) \approx P(e_t|e_{t-N+1}^{t-1}) \approx \frac{c(e_{t-N+1}^{t})}{c(e_{t-N+1}^{t-1})} \tag{3.1}$$

Now we can also approximate the probability of an entire sentence by multiplying the probabilities of each word given its context:

$$P(e) \approx \prod_{i=1}^{n} P(e_i|e_1^{t-1}) \approx \prod_{i=1}^{n} P(e_i|e_{t-N+1}^{t-1}) \tag{3.2}$$

This model is also called N-gram model, since only N-grams of N consecutive words are considered. Also note that we can motivate some of the preprocessing steps with these equations. When considering the first words of the sentence we assume that the sentence starts with a start of sentence symbol, so we can still consider N-Grams of N consecutive elements. Because of the padding the last word in each sentence will be an end of sentence symbol. This allows us to deal with variable length sentences.

Also note that if the probability of a single word within the sentence is zero, the probability of the whole sentence will also be zero. This is alleviated by the predetermined vocabulary and by replacing the remaining words with the unknown word symbol. Nevertheless it is still a crucial problem. Consider the sentences 'I study in Karlsruhe' and 'I study in Pittsburgh'. Even if all words are part of the vocabulary, it is possible that the N-gram 'study in Karlsruhe' is present in the training corpus, while 'study in Pittsburgh' was not part of the corpus. Thus we would assign the

entire second sentence a probability of zero, even if the sentence is linguistically sound.

To prevent this issue and avoid to assign zero probabilities, a number of methods to smooth the probability distribution of N-grams are possible. A simple smoothing method is interpolating multiple language models and including unigram and bigram models. More sophisticated options are Katz smoothing [Katz87] and Modified Kneser-Ney smoothing [NeEK94]. For a summary of smoothing methods we refer the reader to [ChGo96] and [Good01].

The properties of statistical N-gram models lead to large memory requirements. First of all when creating the language model the count or probability of every N-gram has to be stored. Since the number of possible N-grams grows exponentially with the vocabulary size and the maximal N-gram size N, for reasonable sized N-grams memory limitations already become an issue. Even with a careful implementation and storage of the language model, models requiring 1 TB of RAM are used for machine translation tasks [HPCK13]. However, to query the probability of a single word only a single read is required, so the query time is relatively fast.

Another advantage of these models is the ability to remember rare events. Consider the sentence "The Karlsruhe Institute of Technologie was founded in 1825". Given the context the model will assign the number "1825" a high probability, given that this specific sentence was present in the training corpus.

3.1.2 Neural Models

The first neural language models were also based on word N-grams [Schw07]. In these models each word of the current context is mapped to a vector. These vectors are normally learned with a large text corpus using tools like [MCCD13] or [JeMa]. The concatenation of these vectors forms the input to the neural language model. By applying multiple matrix multiplications and element wise functions with parameter matrices a probability distribution over a fixed sized vocabulary is calculated. These parameter matrices are 'learned' based on a text corpus. A formal description of this process is provided in chapter 2.

An advancement of N-gram models are recurrent neural network language models. At each time step only a single word forms the input to these recurrent models. But by maintaining an internal state the model is able to remember the previous words, so theoretically it is possible to consider an arbitrary long context to predict the next word. However, for most applications like machine translation and speech recognition only words of the current sentence are considered and the internal state resets after each sentence.

For neural models the memory requirements only increase linearly with the vocabulary size. For each word a new embedding vector has to be learned. Additionally you have to increase the size of the output layer. In contrast to count based models the computational requirements during training and during inference are high.

A neural model does not assume any similarity between two words a priori, even if the spelling of these words is similar. Since each word gets mapped to a vector, a neural model tries to map words that have a similar meaning to vectors which are close to each other. So in many cases words like "run" and "runs" will get mapped

to vectors which have a small distance to each other. However, these vectors have to be learned based on a training corpus and word based models do not consider the spelling of the words while learning the word vectors.

3.2 Character Based Models

In this section we consider character based language models. In contrast to word based models, we do not have any notion of words. Therefore we don't have to define a vocabulary, we only define an alphabet of valid characters. This leads to a few differences between the specific statistical and neural language models, which we will discuss in the following subsections.

3.2.1 Statistical Models

For word based language models we used N-gram models. For example in a 4-gram model we predict the next word based on the last three words. The length of an usual English word is on average about five characters. So if we want to consider three words in our context, a context of about 15 characters should be considered. However, a context of eight characters can already be to memory intensive. [MXJN15] claims that they used 21 GB to store a 7-gram character based language model.

While this is an important constraint, some count-based character based language models make use of the cache effect. The cache effects states that character sequences, that recently appeared in a text corpus have a higher probability of reoccuring [ClRo97]. This phenomenon is not exploited in the word based language models described in section 3.1. For word based models we estimated the parameters of the model based on a training corpus and they remained fixed during evaluation.

However, this is not necessary for language models. We can assume that the data is not stationary and update our probabilities during the evaluation. We will show this procedure as used in the approaches of [Maho02] and [Maho05].

In the following we will still determine the probabilities based on a $N-1$ character context. But instead of predicting the next character, we will estimate the probability of the next bit. Furthermore, the probabilities will be a function of the complete context of the next bit, that means all text that appeared before the next bit. However, more recent events will be weighted higher than less recent events.

So consider the trigram character context ca, 011 (bits can also be included in the context). Assume that the only occurrences of this context have been can, can, can, cat, cat. Thus the bit history of this context is 00011. We will weight the bit history with the inverse temporal model. Each event within the history is weighted with $1/t$, where t is the "age" of the bit. So the five bits (00011) will be weighted with 1/5, 1/4, 1/3, 1/2 and 1/1. Thus the probability of the next bit being 1 given the context will be:

$p(1|'ca011') = \frac{1/2+1/1}{1/5+1/4+1/3+1/2+1/1} \approx 0.657$

To save memory the inverse temporal model is approximated with the following algorithm, that keeps a counter n_0 and n_1. These counters keep track of the weighted count for each context as follows:

Initialize: $n_0, n_1 = 0$
if *bit x occurs* **then**
  increment n_x
  if $n_{1-x} > 2$ **then**
    set $n_{1-x} = \lfloor n_{1-x}/2 + 1 \rfloor$
  end
end

Algorithm 1: Algorithm to approximate the inverse temporal model

$\lfloor x \rfloor$ denotes rounding the number x. With these counters the probabilities are calculated as follows:

$$\begin{aligned} n_0 &= \frac{n_0}{n_0 + n_1} \\ n_1 &= \frac{n_1}{n_0 + n_1} \end{aligned} \tag{3.3}$$

Another difference to stationary models is that we can tackle the interpolation of different models more dynamically. We can interpolate the models by assigning them different weights. But instead of keeping the weights fixed, [Maho05] uses adaptive weights which change based on the performance of the models. This can be implemented by a simple, gradient based procedure. As in word based models it is common to train models of different N-gram sizes.

It is theoretically interesting not to assume a stationary text source, which was usually calculated based on a fixed training corpus. Using the previous seen text to change the probabilities of a model is also a memory efficient way to make use of a very large "context". Besides it is also very reasonable to assign higher weights to events that appeared more recently to predict the next event. It is assumed that the rate of learning for animals is also inversely proportional to the time of a signal (such as a dog hearing a bell) and a reward (the dog gets meat, causing it to learn to salivate when it hears the bell) [ScRe91].

However, when using language models in downstream tasks, you usually do not get the perfect information. So it is possible to create worse language model probabilities when updating the model. This can be the case when dealing with imperfect transcriptions of a speech recognition system. For word based models [ClRo97] already used adaptive language models with some success. However, I'm not aware of recent work on incorporating non stationary character based language models for downstream tasks like speech recognition.

3.2.2 Neural Models

Character-based neural network language models work similar to their word based counterparts. Instead of using feedforward networks, character based models mainly use recurrent neural networks. It is also common to use a large hidden state for the RNNs. In contrast to word based models, character based models additionally have

to keep track of the current position within a word. This information also has to be represented within the hidden state of the RNN.

For character based models the performance considerations change significantly. The language model has to be queried more often than in word based models. However, the main bottleneck of the neural network changes. For word based networks, the main performance issue is calculating the probability distribution over the vocabulary in the last layer of the neural network. Since the size of the output layer is determined by the size of the alphabet in character based models, this is not an issue anymore for languages like German and English.

3.3 Comparison

In this section we compare the performance of the different methods on benchmarks which were evaluated in recent publications.

For word based models a comparison of neural methods and statistical methods is performed in [JVSS+16]. The performance of the language models was evaluated on the one billion word benchmark [CMSG+13]. RNN based models significantly outperformed the best count based methods, which used Kneser Ney smoothing. However, by additionally considering the characters of each words in the input layer, some improvements over purely word based models were achieved. The individual words could be mapped to vectors by using a character convolutional neural network. Table 3.1 summarizes these results.

Method	Perplexity
Interpolated KN 5-Gram [CMSG+13]	67.6
2 Layer LSTM-8192-1024 [JVSS+16]	30.6
2 Layer LSTM-8192-1024 + CNN Inputs [JVSS+16]	30.0

Table 3.1: Test perplexities on the one billion word benchmark

For character based language models language models statistical motivated approaches perform similar to neural approaches. A popular benchmark to evaluate character based models is the english Hutter prize wikipedia dataset (Enwik8) [Hutt]. This dataset consists of 90M characters of training data and 5M characters each for the validation and test set each.

Method	Bits per Character
Stacked LSTM [Grav13]	1.67
decomp8 [Maho05]	1.28
Recurrent Highway Networks [ZSKS16]	1.27

Table 3.2: Bits per character on the test set of Enwik8

As shown in table 3.2, there has been considerable progress on the Enwik8 benchmark over simple stacked LSTMs. While a few other papers have improved over standard LSTM networks [ChAB16, HaDL16], [ZSKS16] was able to slightly beat the statistical approach.

A comparison between word and character based language models is not as straight forward. Word based approaches only assign probabilities to a fixed size vocabulary, while character based models assign probabilities to each possible character sequence.

However, word and character based language models can be applied to the same downstream task. For speech recognition this was done in [ZYDS16]. When decoding CTC based speech recognition systems word based language models performed better than character based ones. However, this system was only used with a N-gram based language model.

It remains to be seen if character or word based models will be more successful in the future. More modern RNN versions like LSTMs and similar architectures have led to improvements for character based models. Word based models recently make more use of the information provided by the spelling of a word. The use of subword units as used in [SeHB15] are a successful trade-off between purely word and character based approaches.

4. Improving Character Based Language Models

This chapter deals with character based language models. Especially, we will consider the advantages and disadvantages of recurrent neural networks (RNNs) when used for character language models. Our goal is to improve over the commonly used RNNs like Long Short Term Memories (LSTMs) or Gated Recurrent Units (GRUs).

One important difference between character and word based language models is the sequence length of the input. A typical sequence of characters is about five times longer than a sequence of words. As discussed in chapter 2 backpropagation through multiple time steps can lead to exploding or vanishing gradients. Another drawback is the performance. One has to do multiple calculations for each time step one after another, since the next calculation depends on the hidden state of the last time step. These problems are especially stressed for character sequences, since they are multiple times longer than word sequences.

We will try to tackle these problems and design a simple improvement over stacked RNNs. This chapter is structured as follows: We will first review recent approaches for character based systems and developments for different RNN models. Afterwards we will describe and discuss our architecture for a stacked RNN. The chapter concludes with experiments on a popular dataset for character based language modeling.

4.1 Related Work

LSTMs and GRUs are the most commonly used recurrent networks. By performing an architecture search [Zare15, GSKS+16] showed that simple variations of these recurrent networks do not lead to significant improvements. However, a few more sophisticated architectures were proposed in recent papers.

An interesting new approach are clockwork RNNs [KGGS14]. Clockwork RNNs partition the hidden state into several modules. These modules "tick" at different time scales. For example the first module could change its hidden state at every time

step, while the second module only changes its state at every fourth time step. This architecture was tested successfully for audio signal generation and spoken word classification.

Another interesting aspect is that RNNs are able to learn word embeddings by processing each character at a time. Afterwords the hidden state of the RNN can be used as a word embedding. For machine translation this method was used in two different variants. [LuMa16] produced word embeddings with character RNNs for rare words to achieve open vocabulary machine translation, while [JHOS+16] generated word embeddings for every source word with a character RNN.

4.2 Architecture

For designing a new architecture we will adapt the idea of the clockwork RNN, that uses modules that work at different time scales. But instead of varying time scales for the units in the hidden state, the key idea is to let entire layers work at different speeds. Suppose we have an arbitrary RNN layer. We will denote a RNN layer that takes an input x and its previous hidden state h and produces an output y and a new hidden state h^* as $y, h^* = R(x, h)$.

To define the calculations at each layer l, first of all we need a global clock t. We additionally define different timescales for each layer. s_l denotes that layer l works at every ${s_l}^{\text{th}}$ time step. The scales of each layer stay the same or get bigger, so we will satisfy $s_{l-1} \leq s_l$ and we set $s_1 = 1$. Superscripts define the time step and subscripts the layer, so we can define the calculations of the architecture as:

$$y_l^t, h_l^t = \begin{cases} R(y_{l-1}^t, h_l^{t-1}), & \text{if } t \in \{s_l, 2 \cdot s_l, 3 \cdot s_l, \dots\} \\ y_l^{t-1}, h_l^{t-1}, & \text{otherwise} \end{cases} \tag{4.1}$$

Note that the hidden state and the output of a layer l stay the same, if the time step t is not a multiple of its time scale s_l. Otherwise the state and the output are updated based on the output of the previous layer y_{l-1} and the hidden state of the current layer h_l. We define y_0^t as the input of the neural network at time step t.

For each layer y_l and each time step t we calculate an output y_l^t. However, to predict a probability distribution it is not enough to only consider the output of the last layer L. This would lead to a probabilty distribution which is the same for s_L consecutive time steps, since the output of this layer does not change for s_L time steps. For this reason we merge the outputs of each layer and calculate the probability distribution with a softmax layer. This setup is depicted in figure 4.1.

We merge different layers by concatenating their outputs. Since we do not want that each output vector of the layers has the same influence at each timestep, we apply an additional RNN. The idea behind this is that the RNN can decide based on its hidden state how important each output of the different layers is. This can be implemented as follows:

$$\begin{aligned} c &= [y_1, \dots y_L] \\ y_m, h_m &= R(h_m, c) \\ p &= s(W \cdot y_m + b) \end{aligned} \tag{4.2}$$

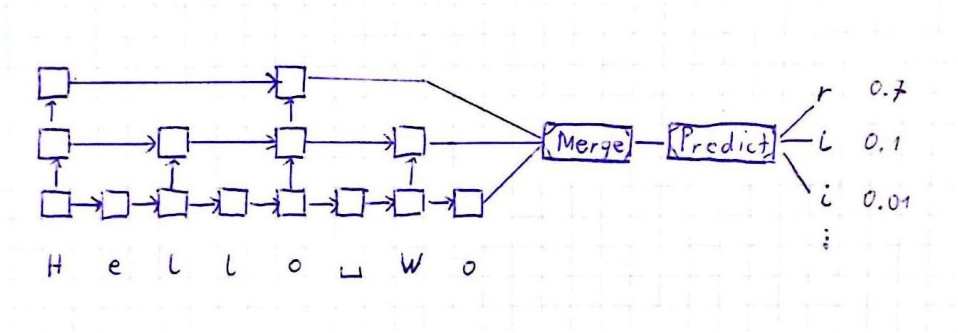

Figure 4.1: Visualization of an unrolled Hierarchical Clockwork RNN

$[y_1, \dots y_L]$ denotes the concatenation of the output of the different layers to form a context vector c. The merging RNN produces an output y_m. An additional weight matrix W and bias vector b is applied to the output y_m. The resulting vector is used by the softmax function s to calculate a probability distribution p for the current time step.

The design of this architectures has various advantages. First of all, we do not have to update the higher layers at each time step, which can save computational resources with a careful implementation. Another advantage is that we have a shorter path during backpropagation. If we backpropagate the error within the highest layer we have to perform a factor of s_L calculations less. This can help to deal with the vanishing gradient problem. Finally we can still use proven RNNs like LSTMs and GRUs as our RNN component for each layer.

In the following we will refer to the described architecture as hierarchical clockwork RNN (HCRNN).

4.3 Experiments

We use the Hutter Prize dataset Enwik8 both for training and evaluating the neural model. This data set consists of 100MB of English Wikipedia articles consisting of 205 unique symbols. We split the data in a training set (90MB), a validation set (5MB) and a test set (5MB). We train a HCRNN to predict the next character given its context, which is the classical language modeling task.

For our HCRNN architecture we use GRUs for the different layers as well as for the merging RNN. We use three layers each consisting of 512 units, while the merge GRU consists of 256 units. The timescales for the three layers are $s_1 = 1$, $s_2 = 4$ and $s_3 = 16$. The character inputs are mapped to a 32 dimensional embedding vector.

Training the HCRNN is performed iteratively by adding more layers at each training step. First of all only the first layer is trained (figure 4.3). Afterwards we add the second, slower layer and continue training (figure 4.3). Finally we train the full network as depicted in figure 4.1. We optimized the network with SGD with momentum using a learning rate of 0.1 and a momentum factor of 0.9. The implementation uses Keras with the Theano [ARAAA+16] backend.

With this setup we achieved an entropy of 1.60 Bits per character (BPC). We compare this architecture with a three layer GRU network also using 512 units per layer,

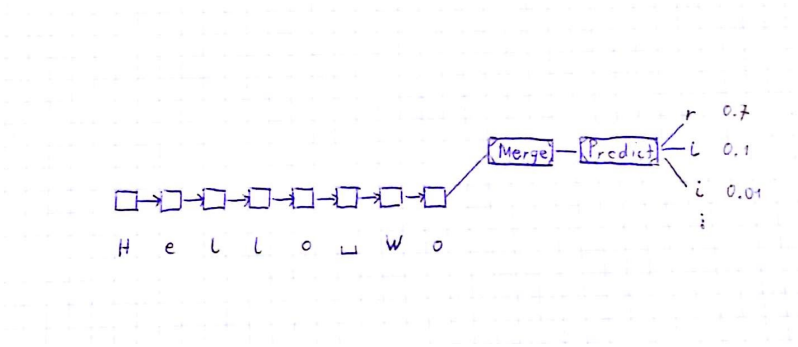

Figure 4.2: Pretraining of the first layer of the HCRNN

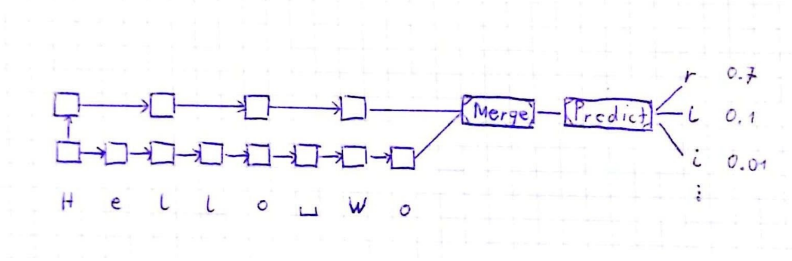

Figure 4.3: Pretraining of the first two layers of the HCRNN

Model	Parameter	Bits per Character
3-Layer Stacked GRU	4.12M	1.70
7-Layer Stacked LSTM [Grav13]	21.3M	1.67
3-Layer HCRNN	5.44M	1.60
7-Layer RHN [ZSKS16]	45M	1.27

Table 4.1: Bits per Character on the Wikipedia Dataset Enwik8 for different architectures

which performed slightly worse (1.70 BPC). All the hyperparameters during training were kept the same. Additionally we compare this setup to results reported in the literature. [Grav13] trains a large network consisting of seven LSTM layers with 700 units per layer (1.70 BPC). Although this network was significantly larger, our HCRNN architecture outperformed these results. The best numbers on this dataset were published very recently in [ZSKS16]. Recurrent Highway Networks (RHN) achieved the best results with 10 stacked layers and 1500 units per layer and achieved an entropy of 1.27 BPC. The results are summarized in table 4.1

The hierarchical clockwork RNN was able to outperform both a similar sized stacked GRU and a considerable larger stacked LSTM network. We speculate that this is due to the architectural improvements, which facilitate the propagation of the error over multiple time steps. However, the way bigger recurrent highway network clearly outperformed the HCRNN network. One reason for this is that the highway network is able to perform adaptive compuation. At each layer it can decide if it wants to transport the input or transform the input with a weight matrix. This is not possible in the HCRNN, the input of a layer always gets transformed. Apart from the parameter increase of the recurrent highway network this should be the main reason for their performance improvements.

4.4 Conclusion

We proposed a new architecture for recurrent neural networks, the hierarchical clockwork RNN (HCRNN). In the HCRNN the layers work at different time scales, which facilitates the propagation of the error over multiple time steps. We tested the architecture on a popular dataset for character based language modeling. We were able to obtain better results than a considerable larger stacked LSTM network.

5. Applications in Speech Recognition

This chapter describes the implementation of an open vocabulary speech recognition system. We implement a CTC acoustic component and decode it with the information of a character based language model. We compare the results to a system with the same acoustic component and a word based language model with a fixed vocabulary. These results are also published in [ZSMN+17] and some parts of this chapter overlap with the paper.

5.1 Introduction

Traditionally, Acoustic Models (AMs) of an Automatic Speech Recognition system followed a generative approach based on HMMs [RaJu86] where the emission probabilities of each state were modeled with a Gaussian Mixture Model. Since the AM works with phonemes as a target, during decoding the information of the AM had to be combined with a pronunciation lexicon, which maps sequences of phonemes to words, and a word based LM [SMFW01].

More recent work has been focused on solutions which come close to end-to-end systems. Connectionist Temporal Classification (CTC) acoustic models [GFGS06] can directly model the mapping between speech features and symbols without having to rely on an alignment between the audio sequence and the symbol sequence. However, the CTC objective function requires that its output symbols are conditional independent of each other. While this assumption is essential to learn a mapping between the speech features and the output sequence, it also entails to add linguistic information during decoding.

Other end-to-end approaches that are inspired by recent developments in machine learning system such as [BaCB14] are [BCSB+16, CJLV15]. By attending to different frames for each output symbol attention based speech recognition systems are able to map speech features to an output sequence.

This approach has no need to assume conditional independence between its output, and therefore is theoretically able to jointly learn acoustic and linguistic models implicitly.

In this chapter we compare the different approaches and discuss where a purely character based models can be integrated. We will choose the CTC framework for the implementation of a purely character based speech recognition system. While the CTC system is not providing a strictly end-to-end point of view, the separation of acoustic model and language model allows for domain independence and adaptation or re-use of speech recognition components.

Similar to [MXJN15] and [HwSu16] we combine the CTC acoustic component and the language model in a search to find the most reasonable transcription. We compare this procedure to a word based search as suggested in [MiGM15, SSRI+15].

Most importantly this combination allows us to create an open vocabulary speech recognition system.

5.2 Background: Approaches for Speech Recognition

In this section we will discuss the different approaches which can be used to implement a speech recognition system. Our goal will be to use one of these approaches to implement an open vocabulary speech recognition system. We will discuss the advantages and disadvantages for integrating a character based language model and discuss the need for a fixed vocabulary in each of these approaches

5.2.1 Statistical HMM based ASR

Statistical HMM based speech recognition systems have been used heavily in the last decades in toolkits like Kaldi [PGBB+11] and Janus [SMFW01]. These systems consist of several components as depicted in figure 5.1. The first step is to record and preprocess the speech. Afterwards statistical ASR system perform a search to look for the most probable word sequence W^*.

Within this search the information of several different models is combined. One component is the acoustic model $P(A|W)$, which calculates the propability of the audio feature A given a word sequence W. In almost all systems the acoustic model makes use of a mapping of words to phoneme sequences. This mapping is provided by a pronunciation lexicon. The last component is the language model $P(W)$, which is able to calculate a probability distribution for a word sequence $P(W)$.

Since these components cannot calculate the most probable word sequence W^* given the acoustic features A directly, we apply Bayes' theorem:

$$\begin{aligned} W^* &= \arg\max_W P(W|A) \\ &= \arg\max_W \frac{P(W) \cdot p(A|W)}{P(A)} \\ &= \arg\max_W P(W) \cdot p(A|W) \end{aligned} \tag{5.1}$$

Notice that we do not need to calculate $P(A)$, since this probability remains static for given acoustic features A and does not affect the search for the most probable word sequence.

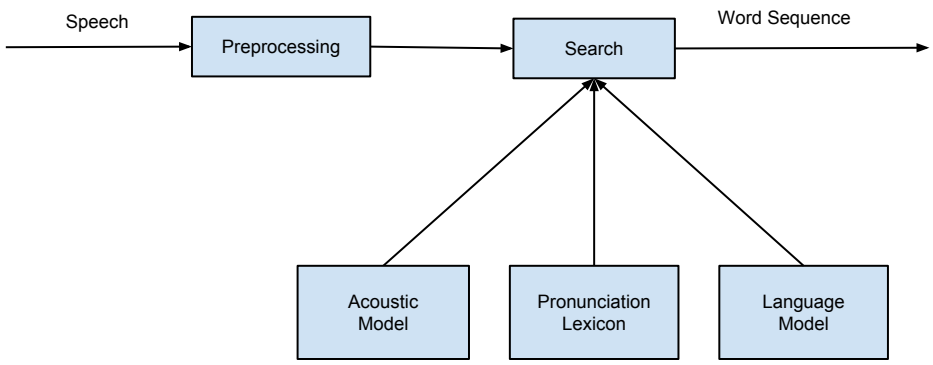

Figure 5.1: Diagram of a HMM based speech recognition system

There are multiple drawbacks if one wants to include a character language model into this setup. First of all the acoustic model is not character based. It depends heavily on the mapping of words to phonemes, therefore we cannot easily include the language model information at the character level.

Another inconvenience is that it is difficult to produce a reasonable word sequence without any information from the language model. Thus it is harder to compare the improvements of adding linguistic information directly.

Additionally, we need an entry in the pronunciation lexicon for every word. This means that it is not straight forward to recognize new or unknown words. This, however, is the main advantage of a character based language model in contrast to a word based one.

5.2.2 Connectionist Temporal Classification

Another method to recognize speech is based on the Connectionist Temporal Classification (CTC). A CTC acoustic component is directly able to predict the probability of a character or phoneme sequence given the acoustic features $P(C|A)$. We will focus on the prediction of a character sequence, since this will allow us to conveniently include the character language model.

A diagram of a CTC based speech recognition system is shown in figure 5.2. It still preprocesses the speech signal. But in contrast to HMM based models it can directly output a character sequence. It does not have the need to include any linguistic knowledge.

We will now discuss in detail how the CTC model is able to directly predict a character sequence. We will also see how the assumptions made during training a CTC model affect its effectiveness to implicitly learn linguistic information, which will be important for the latter part of this chapter.

The CTC component of our system is composed by multiple RNN layers followed by a softmax layer. RNN layers, which are composed by bidirectional LSTM units [HoSc97],

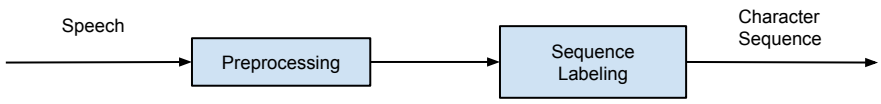

Figure 5.2: Diagram of a CTC based speech recognition system

provide the ability to learn complex, long term dependencies. A sequence of multiple speech features forms the input of our model. For each input the AM outputs a probability distribution over its target alphabet. The whole model is jointly trained under the CTC loss function [GFGS06].

More formally, let us define a sequence of n-dimensional acoustic features $\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$ of length T as the input of our model and L as the set of labels of our alphabet. These labels can be either characters or phonemes. We augment L with a special blank symbol $\emptyset$ and define $L' = L \cup \emptyset$.

Let $\boldsymbol{z} = (z_1, .., z_U) \in L^U$ be an output sequence of length $U \leq T$, which can be seen as the transcription of an input sequence. To define the CTC loss function we additionally need a many to one mapping $\mathcal{B}$ that maps a path $\boldsymbol{p} = (p_1, \ldots p_T) \in L'^T$ of the CTC model to an output sequence $\boldsymbol{z}$. This mapping is also referred as the squash function, as it removes all blank symbols of the path and squashes multiple repeated characters into a single one (e.g. $\mathcal{B}(AA\emptyset AAABB) = AAB$). Note that we do not squash characters that are separated by the blank symbol as this still allows us to create repeated characters in the transcription. Let us define the probability of a path as

$$P(\boldsymbol{p}|\boldsymbol{X}) = \prod_{t=1}^{T} y_k^t \tag{5.2}$$

where y_k^t is the probability of observing the label k at time t. To calculate the probability of an output sequence $\boldsymbol{z}$ we sum over all possible paths:

$$P(\boldsymbol{z}|\boldsymbol{X}) = \sum_{\boldsymbol{p} \in B^{-1}(\boldsymbol{z})} P(\boldsymbol{p}|\boldsymbol{X}) \tag{5.3}$$

To perform the sum over all path we will use a technique inspired by the traditional dynamic programming method used in HMMs, the forward-backward algorithm [RaJu86]. We additionally force the appearance of blank symbols in our paths by augmenting the sequence of output labels during training with a blank symbol between each of the labels of $\boldsymbol{z}$ as well as at the beginning and the end of the sequence.

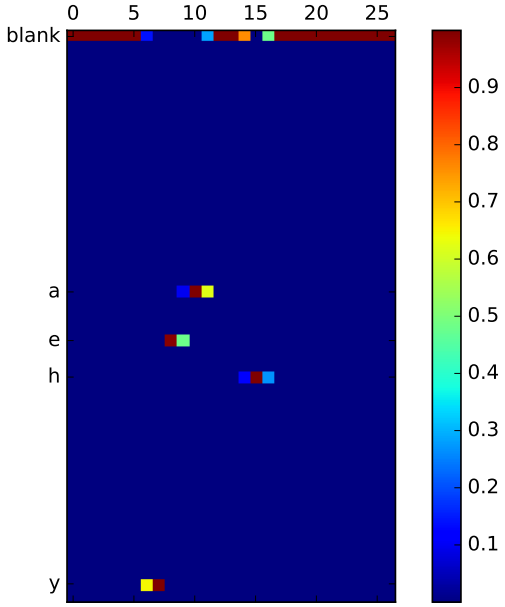

Figure 5.3: Probability matrix of an utterance with the labeling "yeah"

We train our acoustic model by optimizing the logarithm of this function. Note that this procedure assumes conditional independence between the output labels. So the CTC model does not know what its previous output were. This makes it more difficult for the model to learn linguistic information implicitly.

Given a sequence of speech features $\boldsymbol{X} = (\boldsymbol{x_1}, \ldots, \boldsymbol{x_T})$, we can now calculate the probability distribution over the augmented label set L' for each frame. In the remainder of the paper let $P^t_{AM}(k|\boldsymbol{X})$ denote the probability to encounter label $k \in L'$ at time step t given the speech features $\boldsymbol{X}$.

Figure 5.3 visualizes the probability matrix $P^t_{AM}(k|\boldsymbol{X})$ for an example utterance. The horizontal axis denotes the time steps, while the vertical axis denotes the different characters. Note that most of the probabilities are assigned to the blank label. However, at some time steps there are peaks at specific characters.

When looking at this matrix it is straight forward to produce a character sequence. By choosing the most probably sequence at each time step, the resulting path will get squashed to the character sequence "yeah". We will refer to this decoding strategy as greedy search. More formally greedy search is looking for path $p \in L'^T$ as follows:

$$\arg\max_{\boldsymbol{p}} \prod_{t=1}^{T} P^t_{AM}(p_t|\boldsymbol{X}) \tag{5.4}$$

The mapping of the path to a transcription z is straight forward and works by applying the squash function: $z = \mathcal{B}(p)$.

However, the CTC model is not always able to produce such a clean probability matrix. This motivates us to add a language model. For doing this we use a similar setup to statistical speech recognition systems. We perform a search and combine

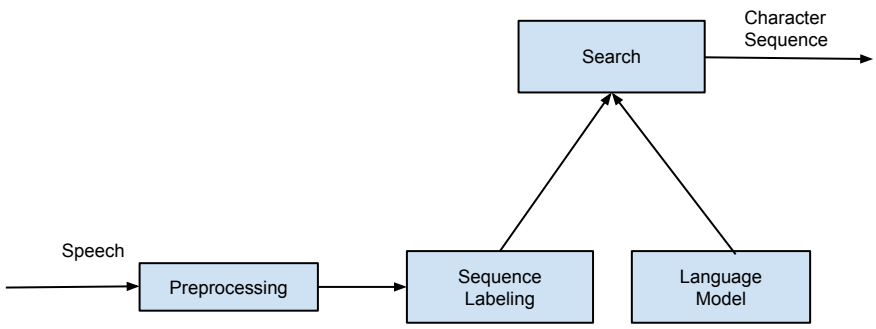

Figure 5.4: Diagram of a CTC based speech recognition system with a language model

the acoustic component and the language model to look for the most reasonable character sequence. However, we do not need a pronunciation lexicon anymore. This setup is depicted in figure 5.4.

The common way is to add a word based language model. For this we use a technique called Weighted Finite State Transducer (WFST). The WFST uses a N-gram model to add linguistic information. But for this we first of all preprocess the CTC probability sequence with the prior probability of each unit of the augmented label set L'.

$$p(\boldsymbol{X}|k) \propto P(k|\boldsymbol{X})/P(k) \tag{5.5}$$

This does not have a proper theoretical motivation since the result is only proportional to a probability distribution. However, by dividing through the prior probability units which are more likely to appear at a particular position than their average will get a high value.

The search graph of the WFST is composed of three individual components:

- A token WFST maps a sequence of units in L' to a single unit in L by applying the squash function $\mathcal{B}$
- A lexicon WFST maps sequences of units in L to words
- A grammar WFST encodes the permissible word sequences and can be created given a word based N-gram language model

The search graph is used to find the most probable word sequence. Note that the lexicon of the WFST allows us to deal with character as well as phoneme based acoustic models.

However, it is also possible to use a character based language models during the search. In contrast to statistical HMM based speech recognition systems, we now have the ability to directly add the linguistic information at the character level. There already has been prior work on this [HwSu16]. We will use this work to implement a purely character based system.

5.3 CTC Beam Search Algorithm

In contrast to the WFST based approach we can directly apply the probabilities at the character level with this procedure. For now assume that the alphabet of the character based LM is equal to L. We want to find a transcription which has a high probability based on the acoustic as well as the language model. Since we have to sum over all possible paths p for a transcription z and want to add the LM information as early as possible, our goal is to solve the following equation:

$$\arg\max_{\boldsymbol{z}} \sum_{\mathcal{B}^{-1}(z)=p} \prod_{t=1}^{T} y_{p_t}^t \cdot P'_{LM}(p_t|\mathcal{B}(\boldsymbol{p}_{1:t-1})) \tag{5.6}$$

Note that we cannot estimate a useful probability for the blank label $\emptyset$ with the language model, so we set $P'_{LM}(\emptyset|p) = 1 \forall p \in \mathcal{P}(L')$. To not favor a sequence of blank symbols, we apply an insertion bonus $b \in \mathbb{R}$ for every $p_t \neq \emptyset$. This yields the following equation:

$$P'_{LM}(k|\boldsymbol{p}) = \begin{cases} P_{LM}(k|\boldsymbol{p}) \cdot b, & \text{if } k \neq \emptyset \\ 1, & \text{if } k = \emptyset \end{cases} \tag{5.7}$$

where $P_{LM}(k|\boldsymbol{p})$ is provided by the character LM. As it is infeasible to calculate an exact solution to equation 6, we apply a beam search similar to [HwSu16][1].

However, in some cases it is favorable to give the LM a higher weight. We achieve this with a additional hyperparameter, the language model weight $w_{LM} \in \mathbb{R}$. We include this hyperparameter in the language model as follows:

$$P'_{LM}(k|\boldsymbol{p}) = \begin{cases} P_{LM}(k|\boldsymbol{p})^{w_{LM}} \cdot b^{w_{LM}}, & \text{if } k \neq \emptyset \\ 1, & \text{if } k = \emptyset \end{cases} \tag{5.8}$$

For $w_{LM} = 1$ this produces the original equation. If $0 < w_{LM} < 1$, the language model will be less important. For the case $w_{LM} > 1$ the language model gets a higher importance.

Also notice that we apply the language model weight to the bias, too. The motivation for this is, that we still want to have language model probabilities close to 1 for sound transcriptions, as we do not want to favor blank labels. We recognize that applying the language model weight to the bias is an approximation to achieve this.

For AMs which do not use spaces nor have another notion of word boundaries, it is possible to add this information based only on the character LM. This can be achieved by adding a copy of each transcription appended by the space symbol at each time step. This works surprisingly well, since spaces at inappropriate position will get a low LM probability. To the best of our knowledge this is a novel approach and can easily be extended to a larger number of characters. For example, an useful application for this would be to add punctuation marks during the beam search.

While this approach is only able to deal with character based CTC models, it can create arbitrary, open vocabulary transcriptions.

[1] Code is included within EESEN: https://github.com/srvk/eesen

5.3.1 CTC Beam Search Experiments

We want to apply these algorithms to a speech recognition task. For this we choose the Switchboard Telephone Speech Corpus. In this subsection we will describe the training process as well as the implementation details of the different components. Afterwards we will describe the word based language model and its integration in the WFST framework, which will serve as the comparison to the character based approach. The character based beam search concludes this subsection.

We use the Switchboard data set (LDC97S62) to train the AM. This data set consists of 2,400 two-sided telephone conversations with a duration of about 300 hours. It is composed of over 500 speakers with different US accents talking about 50 randomly picked topics. We pick 4000 utterances as our validation set for hyper parameter tuning. Our target labels are either character or phonemes.

We also augment the training set to get a more generalized model using two techniques. First, by reducing the frame rate, applying a sub sampling and finally adding an offset we augment the number of training samples. Second, we augment our training set by a factor of 10 applying slight changes to the speed, pitch and tempo of the audio files. The model consist of five bidirectional LSTM layers with 320 units in each direction. It is trained using EESEN [MGNK+16].

Our WFST implementation is composed by three individual components. These components are implemented using Kaldi's [PGBB+11] FST tools. We determine the weights of the lexicon WFST by using a lexicon that maps each word to a sequence of CTC labels. The grammar WFST is modeled by using the probabilities of a trigram and 4-gram language model smoothed with Kneser-Ney [ChGo96] discounting. We create the language model based on Fisher transcripts and the transcripts of the acoustic training data using SRILM [Stol+02].

We train the Character LM with Fisher transcripts (LDC2004T19, LDC2005T19) and the transcripts of the acoustic training data (LDC97S62). Validation is done on the transcription of the acoustic validation data. These transcriptions are cleaned by removing punctuation marks and duplicate utterances. This results in a training text of about 23 million words and 112 million characters. The alphabet of the character LM consists of 28 characters (-abcdefghijklmnopqrstuvwxyz'), a start and end of sentence symbol, a space symbol and a symbol representing unknown characters. We cut all sentences to a maximum length of 128 characters. We use a embedding size of 64 for the characters, a single layer LSTM with 2048 Units and a softmax layer implemented with DyNet [NDGM+17a] as our neural model. Training is performed with the whole data using Adam [KiBa14] by randomly picking a batch until convergence on the validation data. Adam is used with the standard settings of DyNet. That means a learning rate of 0.001, $\beta_1 = 0.9$ and $\beta_2 = 0.999$. We retrain the resulting model on the Switchboard training data using Stochastic Gradient Descent with a low learning rate of 0.01, which is inspired by [WSCL+16]. To provide a comparison of the resulting language model, we also state the performance of the language model. The training procedure results in an average entropy of 1.34 bits per character (BPC) on the train set, 1.37 BPC on the validation set and 1.46 BPC on the evaluation set (LDC2002S09).

5.3.2 CTC Beam Search Error Analysis

We use the 2000 HUB5 "Eval2000" (LDC2002S09) dataset for evaluation. It consist of a "Switchboard" subset, which is similar to the training data, and the "Callhome" subset. These subsets allow to analyze the robustness of the individual approaches to some extend.

Search Method	Ac. Model	E2	CH	SW
Greedy	Character	37.2%	44.0%	30.4%
Char Beam	Character	25.1%	31.6%	18.6%
WFST	Character	23.6%	30.2%	17.0%
WFST	Phoneme	19.6%	25.5%	13.6%
Char Beam [MXJN15]	Character	30.8%	40.2%	21.4%
Char Beam [ZYDS16]	Character		32.1%	19.8%
WFST [ZYDS16]	Character		26.3%	15.1%
WFST Rescore [ZYDS16]	Character		25.3%	14.0%

Table 5.1: Comparison of Word Error Rates for different decoding approaches on the Eval2000 (E2), Call Home (CH) and Switchboard (SW) (sub-)sets.

For the Greedy Search, we use an alphabet consisting of upper and lowercase characters. As in [ZYDS16], an upper case character denotes the start of a new word. This procedure proved to be more successful than using an explicit space character. For all other character based AMs we only use lowercase characters without a space unit. Table 5.1 shows the results, and compares our findings (top part) against related results from the literature (bottom part).

As you can see the purely character based approach is performing significantly better than the greedy decoding, which does not explicitly add any linguistic information. Word based language models still perform slightly better than the character based approach, but notice that they still have the need for a vocabulary and are not able to generate other words while transcribing the speech signal.

In table 5.2 we give an example of the improvements of adding the linguistic information at the character level. You can see the reference of a test utterance and an incorrectly spelled version of it produced by the acoustic component using greedy search. Also note that the CTC model does not care about spaces in this case. The character based model is able to add enough information to completely correct the spelling mistakes. Also notice that spaces are included at the correct positions.

Method	Text
Reference	he is a police officer
Greedy	he'sapolifefolvisere
Character Beam	he's a police officer

Table 5.2: Example output of a cherry picked utterance

If we use a phoneme based CTC model, we cannot easily apply the character based language model anymore. However, to give a comparison we apply the WFST based word language model to decode the phoneme based CTC model. This leads to a 3.4% word error rate reduction. We think that phonemes are still a better abstraction for the acoustic components than characters. Characters can be pronounced

Word	Reference	Hypothesis
disproven	there was they have hit it is funny that they went with the i guess the reagonomics called for trickle-down theory i think that is pretty pretty pretty much **disproven** at this point	there there's they it it it's funny that they went with uh i guess the reganomics called for a trickle down there i think that's pretty pretty much a **disproven** at this point
ducting	either some way to improve the **ducting** or to put a booster fan going to the upstairs	either some way to improve the **ducting** or uh put a a booster fan going to the upstairs
fringing	you know it it has some hanging panes around it with metal fringing and the metal **fringing** was like had bent away from the from the glass so glued it back together	uh you know i i have some in in glassing pains around it with uh medal **fringing** in the middle ringing it was like i had uh been away from the from the glass so i got it back together
spick	and i would want to do that too you know make sure they do not have everything **spick** and span just because it is visiting time or whatever	and i would want to do that to you know make sure they don't have everything **spick** in span is because it's a visiting time or whatever
peppier	yeah we saw that and and they were very highly thought of and what surprised me is that the car was **peppier** than	yeah we saw that and they were very highly uh thought of and what surprised me is that the car was **peppier** than

Table 5.3: Correctly recognized words that were not present in the training corpora

differently in different context or in different words, which can be modeled with an pronunciation lexicon. However, to produce a good lexicon a significant amount of effort is required.

Another interesting comparison is how many errors the different system make by producing words, which did not appear in the training text. These mistakes are often similar to spelling mistakes. Using the character LM during the beam search significantly reduces incorrectly recognized words, which did not appear in the training text (199), by a factor of 30 compared to a simple greedy search (6274). This amounts to a rate of 0.5%, which compares favorably to the out of vocabulary rate of the WFST based approach 0.9%.

These remaining errors are for the most part very similar to valid English words, and could be considered spelling mistakes ("penately") or newly created words ("discussly"). Only on rare occasion does the Character LM not add a space between words ("andboxes"). Most notably, the Open Vocabulary approach was able to generate correct words, which did not appear in the training corpora. These words include: "boger", "disproven", "ducting", "fringing", "spick" and "peppier". In table 5.3 we show the appearances of these words in the generated transcriptions of our system as well as in the references of the test data.

Table 5.4 shows the insertion, deletion, and substitution rates. We consistently used an insertion bonus of 2.5 in our experiments with the beam search. The application of an insertion bonus every time when reducing the probability by the character based LM yields balanced insertion and deletion errors. Also note that for the models a language model weight of 1 produces the best results. Additionally the logarithm of the insertion bonus corresponds well with the entropy of the character language model on the validation set ($log_2(2.5) = 1.3$, validation entropy: 1.37 BPC). Overall, the error patterns of all three systems seem remarkably similar, even though the WFST system has been tuned more aggressively than the other two systems, and thus exhibits unbalanced insertions and deletions.

Method	I	S	D
Greedy	2.4%	26.2%	8.6%
Character Beam	3.6%	16.5%	5.0%
WFST	8.8%	13.0%	1.9%

Table 5.4: Insertion Rate (I), Substitution Rate (S) and Deletion Rate (D) for multiple decoding algorithm using character based AMs evaluated on the Eval2000 dataset.

We also compared our results in table 5.1 to prior work. Our character based acoustic component is competitive to the recently published results in [ZYDS16], which represent state of the art results. We are within 2% WER of the reported number for word based WFST. For open vocabulary, character based speech recognition we report an improvement of over 1% WER compared to previous results [ZYDS16, MXJN15].

5.4 Conclusion

In this section, we compared different decoding approaches for CTC acoustic models, which are trained on the same, open source platform. A "traditional" context independent WFST approach performs best, but the open vocabulary character RNN approach performs only about 10% relative worse, and produces a surprisingly small number of "OOV" errors.

We believe that these results show that there is currently a multitude of different algorithms that can be used to perform speech recognition in a neural setting, and there may not be a "one size fits all" approach for the foreseeable future. While WFST is well understood and fast to execute, the character RNN approach might perform well for morphologically complex languages.

We want to stress that purely character based approaches are able to recognize arbitrary words and do not have the need for a vocabulary. Nonetheless, the character based models are able to correctly recognize unseen words. We suspect that especially in morphologically rich language this will be of major importance and could allow performance improvements compared to more traditional word based approaches.

6. Conclusion

In this thesis we have discussed character based language models. We reviewed the learning process in recurrent neural networks and deduced some of their shortcomings when dealing with long sequences. In contrast to relatively short word sequences, for character sequences problems like exploding or vanishing gradients pose an important problem.

Based on the popular Gated Recurrent Unit, we proposed a neural network architectures that mitigates these problems. The neural network has fast and slow ticking layers and thus creates shorter paths within the computation graph of the unrolled network. By doing less matrix multiplications during backpropagation in slow ticking layers we argue that we mitigate vanishing gradients between distant events. We empirically evaluate our proposed architecture as a character based language model. We verified that this model performs better on this task than stacked Gated Recurrent Units.

Furthermore, we applied character based language models in speech recognition. We integrated these models in the Connectionist Temporal Classification (CTC) framework. We integrated the information of a character based acoustic component and the language model within a straight forward beam search.

This approach leads to a purely character based approach to speech recognition. We do not use any additional phonetic information about the language, which oftentimes has to be produced by experts. Additionally by having a purely character based system we do not have the need of a fixed size vocabulary. We are able to recognize any combination of characters, which enables an open vocabulary speech recognition system.

We evaluated this approach on the Switchboard Telephone Speech Corpus. We verified that we are able to recognize previously unseen words, which did not appear in the training data. This provides a strong indication that the acoustic component and the language model are able to generalize to unseen words of a language. Furthermore, we achieved the best results reported in the literature on the Switchboard corpus for character based, open vocabulary speech recognition using Connectionist Temporal Classification.

We empirically compared this approach with a phoneme based acoustic model and a word based language model on the Switchboard test set by measuring the word error rate (WER). Our purely character based approach (18.7% WER) performs slightly worse than the same system with a word based language model (17.0% WER). A phoneme based acoustic component still achieves considerable better results than both character based models (13.6%). However, note that the phoneme based model needs a pronunciation lexicon, while the word based language model needs a fixed vocabulary.

6.1 Future Work

We will discuss for which applications the proposed character based approach is useful, which improvements are reasonable and how the character based CTC framework can be valuable in combination with more recent attention based speech recognition approaches.

Subword Units

There is still a large performance difference between character based and phoneme based CTC models. We argue that purely character based models are not able to learn the pronunciation variants of different characters perfectly. The same character can be pronounced differently in different contexts or words. It is possible to predict larger units than characters in the CTC framework, for example subword units like character N-grams or byte pair units. This can be applied to both the acoustic component and the language model. We argue that this can help the CTC model to learn different pronunciation variants better.

Unsupervised Learning

We recognized a large gap between transcribing speech with the acoustic component only (30.4% WER) and the acoustic component combined with a language model (18.6% WER). Additionally, with a very large amount of training data the CTC acoustic component is able to learn a reasonable language model implicitly [SoLS16]. We argue that the information of the language model can help the acoustic component to improve. For example we can transcribe unseen speech data with our system while including the language model. This data can be used as additional, noisy training data for the acoustic model. Especially in situations, when we do not have a large amount of high quality training data for the acoustic component, this could lead to improvements.

Dialect and Accents

It is a labor-intensive task to create a pronunciation lexicon for different dialects or accents of a language. Since the character based CTC framework is able to learn this information implicitly, it can be applied directly to different variations of a language. A straight forward approach to implement this is to first train a CTC network in a language and retrain it with less training data on a specific dialect. This approach would still use all the training data available for a language and would use the data of a dialect as the in-domain data.

Combination with Attention Based Models

Attention based speech recognition system are another recent approach to tackle speech recognition tasks. However, they are a more powerful model and are thus prone to overfit the training data [CJLV15, BCSB+16]. CTC models provide a more restricted model, since they are based on the Markov assumption. It is possible to use the advantages of these two models during training. The CTC function can be used to bias the decoder of the attention based system to learn a useful decoding of the speech features. Early work on this has already been done in [KiHW16]. Especially in situation with small amounts of training data this combination can help the attention based system to generalize and avoid to overfit the training data.

Bibliography

[AABB+16] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin und andere. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[ARAAA+16] R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky und andere. Theano: A Python framework for fast computation of mathematical expressions. *arXiv preprint arXiv:1605.02688*, 2016.

[BaCB14] D. Bahdanau, K. Cho und Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[BCSB+16] D. Bahdanau, J. Chorowski, D. Serdyuk, P. Brakel und Y. Bengio. End-to-end attention-based large vocabulary speech recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2016 IEEE International Conference on*. IEEE, 2016, S. 4945–4949.

[BeSF94] Y. Bengio, P. Simard und P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks* 5(2), 1994, S. 157–166.

[BeSo03] Y. Bengio, J.-S. Senécal und andere. Quick Training of Probabilistic Neural Nets by Importance Sampling. In *AISTATS*, 2003.

[ChAB16] J. Chung, S. Ahn und Y. Bengio. Hierarchical multiscale recurrent neural networks. *arXiv preprint arXiv:1609.01704*, 2016.

[ChGo96] S. F. Chen und J. Goodman. An empirical study of smoothing techniques for language modeling. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*. Association for Computational Linguistics, 1996, S. 310–318.

[CJLV15] W. Chan, N. Jaitly, Q. V. Le und O. Vinyals. Listen, attend and spell. *arXiv preprint arXiv:1508.01211*, 2015.

[ClRo97] P. R. Clarkson und A. J. Robinson. Language model adaptation using mixtures and an exponentially decaying cache. In *Acoustics, Speech, and Signal Processing, 1997. ICASSP-97., 1997 IEEE International Conference on*, Band 2. IEEE, 1997, S. 799–802.

[CMSG+13] C. Chelba, T. Mikolov, M. Schuster, Q. Ge, T. Brants, P. Koehn und T. Robinson. One billion word benchmark for measuring progress in statistical language modeling. *arXiv preprint arXiv:1312.3005*, 2013.

[CVMGB+14] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk und Y. Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[DACB+16] L. Duong, A. Anastasopoulos, D. Chiang, S. Bird und T. Cohn. An attentional model for speech translation without transcription. In *Proceedings of NAACL-HLT*, 2016, S. 949–959.

[Elma90] J. L. Elman. Finding structure in time. *Cognitive science* 14(2), 1990, S. 179–211.

[FGCSD+16] J. N. Foerster, J. Gilmer, J. Chorowski, J. Sohl-Dickstein und D. Sussillo. Intelligible language modeling with input switched affine networks. *arXiv preprint arXiv:1611.09434*, 2016.

[GAGY+17] J. Gehring, M. Auli, D. Grangier, D. Yarats und Y. N. Dauphin. Convolutional Sequence to Sequence Learning. *arXiv preprint arXiv:1705.03122*, 2017.

[GFGS06] A. Graves, S. Fernández, F. Gomez und J. Schmidhuber. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd international conference on Machine learning*. ACM, 2006, S. 369–376.

[Good01] J. T. Goodman. A bit of progress in language modeling. *Computer Speech & Language* 15(4), 2001, S. 403–434.

[Grav13] A. Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

[GrJa14] A. Graves und N. Jaitly. Towards End-To-End Speech Recognition with Recurrent Neural Networks. In *ICML*, Band 14, 2014, S. 1764–1772.

[GrJU16] E. Grave, A. Joulin und N. Usunier. Improving Neural Language Models with a Continuous Cache. *arXiv preprint arXiv:1612.04426*, 2016.

[GSKS+16] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink und J. Schmidhuber. LSTM: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 2016.

[GuHy10] M. Gutmann und A. Hyvärinen. Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In *AISTATS*, Band 1, 2010, S. 6.

[HaDL16] D. Ha, A. Dai und Q. V. Le. HyperNetworks. *arXiv preprint arXiv:1609.09106*, 2016.

[HCCC+14] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates und andere. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567*, 2014.

[HMJN14] A. Y. Hannun, A. L. Maas, D. Jurafsky und A. Y. Ng. First-pass large vocabulary continuous speech recognition using bi-directional recurrent DNNs. *arXiv preprint arXiv:1408.2873*, 2014.

[HoSc97] S. Hochreiter und J. Schmidhuber. Long short-term memory. *Neural computation* 9(8), 1997, S. 1735–1780.

[HoSW89] K. Hornik, M. Stinchcombe und H. White. Multilayer feedforward networks are universal approximators. *Neural networks* 2(5), 1989, S. 359–366.

[HPCK13] K. Heafield, I. Pouzyrevsky, J. H. Clark und P. Koehn. Scalable Modified Kneser-Ney Language Model Estimation. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*, Sofia, Bulgaria, August 2013. S. 690–696.

[HSKS+12] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever und R. R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[Hutt] M. Hutter. The human knowledge compression contest. 2012. *URL http://prize. hutter1. net.*

[HwSu16] K. Hwang und W. Sung. Character-level incremental speech recognition with recurrent neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2016 IEEE International Conference on.* IEEE, 2016, S. 5335–5339.

[IoSz15] S. Ioffe und C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[JeMa] R. JeffreyPennington und C. Manning. GloVe: Global Vectors for Word Representation.

[JHOS+16] A. R. Johansen, J. M. Hansen, E. K. Obeid, C. K. Sønderby und O. Winther. Neural Machine Translation with Characters and Hierarchical Encoding. *arXiv preprint arXiv:1610.06550*, 2016.

[JoZS15] R. Jozefowicz, W. Zaremba und I. Sutskever. An empirical exploration of recurrent network architectures. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, 2015, S. 2342–2350.

[JVSS+16] R. Jozefowicz, O. Vinyals, M. Schuster, N. Shazeer und Y. Wu. Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410*, 2016.

[Katz87] S. Katz. Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE transactions on acoustics, speech, and signal processing* 35(3), 1987, S. 400–401.

[KGGS14] J. Koutnik, K. Greff, F. Gomez und J. Schmidhuber. A Clockwork RNN. In *Proceedings of The 31st International Conference on Machine Learning*, 2014, S. 1863–1871.

[KiBa14] D. Kingma und J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[KiHW16] S. Kim, T. Hori und S. Watanabe. Joint ctc-attention based end-to-end speech recognition using multi-task learning. *arXiv preprint arXiv:1609.06773*, 2016.

[KLMR16] B. Krause, L. Lu, I. Murray und S. Renals. Multiplicative LSTM for sequence modelling. *arXiv preprint arXiv:1609.07959*, 2016.

[KoHa16] J. Koushik und H. Hayashi. Improving Stochastic Gradient Descent with Feedback. *arXiv preprint arXiv:1611.01505*, 2016.

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

[LeCH16] J. Lee, K. Cho und T. Hofmann. Fully Character-Level Neural Machine Translation without Explicit Segmentation. *arXiv preprint arXiv:1610.03017*, 2016.

[LeJH15] Q. V. Le, N. Jaitly und G. E. Hinton. A simple way to initialize recurrent networks of rectified linear units. *arXiv preprint arXiv:1504.00941*, 2015.

[LGZZ+16] A. M. Lamb, A. G. A. P. GOYAL, Y. Zhang, S. Zhang, A. C. Courville und Y. Bengio. Professor forcing: A new algorithm for training recurrent networks. In *Advances In Neural Information Processing Systems*, 2016, S. 4601–4609.

[LuMa16] M.-T. Luong und C. D. Manning. Achieving open vocabulary neural machine translation with hybrid word-character models. *arXiv preprint arXiv:1604.00788*, 2016.

[LuZR16] L. Lu, X. Zhang und S. Renais. On training the recurrent neural network encoder-decoder for large vocabulary end-to-end speech recognition. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, March 2016, S. 5060–5064.

[Maho02] M. V. Mahoney. The PAQ1 data compression program. *Draft, Jan* Band 20, 2002.

[Maho05] M. V. Mahoney. Adaptive weighing of context models for lossless data compression. Technischer Bericht, 2005.

[MCCD13] T. Mikolov, K. Chen, G. Corrado und J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[MDMM+16] C. Mendis, J. Droppo, S. Maleki, M. Musuvathi, T. Mytkowicz und G. Zweig. Parallelizing WFST speech decoders. In *Acoustics, Speech and Signal Processing (ICASSP), 2016 IEEE International Conference on*. IEEE, 2016, S. 5325–5329.

[MGNK+16] Y. Miao, M. Gowayyed, X. Na, T. Ko, F. Metze und A. Waibel. An empirical exploration of CTC acoustic models. In *Acoustics, Speech and Signal Processing (ICASSP), 2016 IEEE International Conference on*. IEEE, 2016, S. 2623–2627.

[MiGM15] Y. Miao, M. Gowayyed und F. Metze. EESEN: End-to-end speech recognition using deep RNN models and WFST-based decoding. In *Automatic Speech Recognition and Understanding (ASRU), 2015 IEEE Workshop on*. IEEE, 2015, S. 167–174.

[MnTe12] A. Mnih und Y. W. Teh. A fast and simple algorithm for training neural probabilistic language models. *arXiv preprint arXiv:1206.6426*, 2012.

[MXJN15] A. L. Maas, Z. Xie, D. Jurafsky und A. Y. Ng. Lexicon-Free Conversational Speech Recognition with Neural Networks. In *HLT-NAACL*, 2015, S. 345–354.

[NDGM+17a] G. Neubig, C. Dyer, Y. Goldberg, A. Matthews, W. Ammar, A. Anastasopoulos, M. Ballesteros, D. Chiang, D. Clothiaux, T. Cohn, K. Duh, M. Faruqui, C. Gan, D. Garrette, Y. Ji, L. Kong, A. Kuncoro, G. Kumar, C. Malaviya, P. Michel, Y. Oda, M. Richardson, N. Saphra, S. Swayamdipta und P. Yin. DyNet: The Dynamic Neural Network Toolkit. *arXiv preprint arXiv:1701.03980*, 2017.

[NDGM+17b] G. Neubig, C. Dyer, Y. Goldberg, A. Matthews, W. Ammar, A. Anastasopoulos, M. Ballesteros, D. Chiang, D. Clothiaux, T. Cohn und andere. DyNet: The Dynamic Neural Network Toolkit. *arXiv preprint arXiv:1701.03980*, 2017.

[NeEK94] H. Ney, U. Essen und R. Kneser. On structuring probabilistic dependences in stochastic language modelling. *Computer Speech & Language* 8(1), 1994, S. 1–38.

[Neub17] G. Neubig. Neural Machine Translation and Sequence-to-sequence Models: A Tutorial. *arXiv preprint arXiv:1703.01619*, 2017.

[Olah15] C. Olah. Understanding LSTM Networks. 2015.

[PGBB+11] D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motlicek, Y. Qian, P. Schwarz und andere. The Kaldi speech recognition toolkit. In *IEEE 2011 workshop on automatic speech recognition and understanding*, Nr. EPFL-CONF-192584. IEEE Signal Processing Society, 2011.

[Qian99] N. Qian. On the momentum term in gradient descent learning algorithms. *Neural networks* 12(1), 1999, S. 145–151.

[RaJu86] L. Rabiner und B. Juang. An introduction to hidden Markov models. *ieee assp magazine* 3(1), 1986, S. 4–16.

[RDSK+15] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein und andere. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision* 115(3), 2015, S. 211–252.

[Rude16] S. Ruder. An overview of gradient descent optimization algorithms. *CoRR* Band abs/1609.04747, 2016.

[Schw07] H. Schwenk. Continuous space language models. *Computer Speech & Language* 21(3), 2007, S. 492–518.

[ScRe91] B. Schwartz und D. Reisberg. *Learning and memory.* WW Norton & Co. 1991.

[SeHB15] R. Sennrich, B. Haddow und A. Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

[SMFW01] H. Soltau, F. Metze, C. Fugen und A. Waibel. A one-pass decoder based on polymorphic linguistic context assignment. In *Automatic Speech Recognition and Understanding, 2001. ASRU'01. IEEE Workshop on.* IEEE, 2001, S. 214–217.

[SoLS16] H. Soltau, H. Liao und H. Sak. Neural Speech Recognizer: Acoustic-to-Word LSTM Model for Large Vocabulary Speech Recognition. *arXiv preprint arXiv:1610.09975*, 2016.

[SoOc15] R. Soricut und F. J. Och. Unsupervised Morphology Induction Using Word Embeddings. In *HLT-NAACL*, 2015, S. 1627–1637.

[SSRI+15] H. Sak, A. Senior, K. Rao, O. Irsoy, A. Graves, F. Beaufays und J. Schalkwyk. Learning acoustic frame labeling for speech recognition with recurrent neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on.* IEEE, 2015, S. 4280–4284.

[Stol+02] A. Stolcke und andere. SRILM-an extensible language modeling toolkit. In *Interspeech*, Band 2002, 2002, S. 2002.

[Suts13] I. Sutskever. *Training recurrent neural networks.* Dissertation, University of Toronto, 2013.

[WHHS+89] A. Waibel, T. Hanazawa, G. Hinton, K. Shikano und K. J. Lang. Phoneme recognition using time-delay neural networks. *IEEE transactions on acoustics, speech, and signal processing* 37(3), 1989, S. 328–339.

[WSCL+16] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey und andere. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *arXiv preprint arXiv:1609.08144*, 2016.

[Zare15] W. Zaremba. An empirical exploration of recurrent network architectures. 2015.

[Zeil12] M. D. Zeiler. ADADELTA: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[ZSKS16] J. G. Zilly, R. K. Srivastava, J. Koutník und J. Schmidhuber. Recurrent highway networks. *arXiv preprint arXiv:1607.03474*, 2016.

[ZSMN+17] T. Zenkel, R. Sanabria, F. Metze, J. Niehues, M. Sperber, S. Stüker und A. Waibel. Comparison of Decoding Strategies for CTC Acoustic Models. In *Proceedings of the 17th Annual Conference of the International Speech Communication Association, Interspeech 2017*. International Speech Communication Association, 2017.

[ZYDS16] G. Zweig, C. Yu, J. Droppo und A. Stolcke. Advances in all-neural speech recognition. *arXiv preprint arXiv:1609.05935*, 2016.