Karlsruhe Institute of Technology

Deep Neural Network Language Models for Low Resource Languages

Bachelor Thesis of

Kuangyuan Jin

At the Department of Informatics
Institute for Anthropomatics

Reviewer:	Prof. Dr. Alexander Waibel
Second reviewer:	Dr. Sebastian Stüker
Advisor:	Dipl.-Inform. Kevin Kilgour

Duration: December 15, 2013 – April 14, 2014

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

Karlsruhe, April 14, 2014

.....

(**Kuangyuan Jin**)

Zusammenfassung

Diese Arbeit beschreibt, wie tiefe neuronale Netzarchitekturen zur Sprachmodellierung eingesetzt werden können. Während N-gram Sprachmodelle, die auf wenig Daten trainiert sind, eine deutlich schlechtere Leistung aufweisen, versuchen Sprachmodelle aus neuronalen Netzen dies zu verhindern, indem sie die Wahrscheinlichkeitsschätzung in einem kontinuierlichen Raum durchführen. Diese Sprachmodelle sind in der Lage, zwischen Wahrscheinlichkeiten von diskreten Wortsequenzen zu interpolieren, indem sie Wörter in eine geeignete kontinuierliche Repräsentation verwandeln, und können somit eine bessere Generalisierungsfähigkeit für ungesehene Ereignisse erzielen.

Die tiefen neuronalen Netze, die in dieser Arbeit benutzt werden, sind als mehrlagige Perzeptrons mit einer Projektionsschicht, einer oder mehreren versteckten Schichten und einer Ausgabeschicht implementiert. Optimierungen am standardmäßigen Lernalgorithmus werden vorgenommen um lange Trainingszeiten zu verkürzen, wobei eine enorme Beschleuningung mithilfe von verschiedenen Methoden ermöglicht werden kann. Neuronale Netze unterschiedlicher Größen werden für Sprachen trainiert, für die wenige Resourcen vorhanden sind, wie zum Beispiel Vietnamesisch, Tamil und Laotisch. Dabei werden unterschiedliche Trainingsparameter untersucht, um Sprachmodelle mit bestmöglicher Leistungsfähigkeit zu erhalten.

Die Sprachmodelle aus neuronalen Netzen werden ausführlich an den Testkorpora des IARPA Babel Programs ausgewertet und mit N-gram Backoff Sprachmodellen verglichen, die auf dem neuesten Stand der Technik sind und mit Kneser-Ney Smoothing trainiert werden. Außerdem werden neue Sprachmodelle, die beide Ansätze durch Interpolation kombinieren, erzeugt und an Aufgaben für Spracherkennung getestet. Die interpolierten Modelle sind in der Lage, konsistente Reduktion der Wortfehlerrate bis zu 0,6% absolut für Vietnamesisch und 0,8% für Laotisch zu erreichen.

Abstract

This work describes how deep neural network architectures can be used for language modeling. While n-gram language models experience a significantly worse performance when trained on sparse data, neural network language models try to prevent this by performing probability estimation in a continuous space. They are capable of interpolating between probabilities of discrete word sequences by converting those words into a continuous representation and can thus achieve a better generalization ability for unseen events. The deep neural networks used in this work are implemented as multilayer perceptrons with a projection layer, one or more hidden layers and an output layer. Optimizations to the standard learning algorithm are carried out to reduce long training time where an enormous speed-up can be obtained by using different methods. Neural networks of various sizes are trained for low resource languages, such as Vietnamese, Tamil and Lao, while experimenting with different training parameters to receive best possible performance. The neural network language models are thoroughly evaluated on test corpora from the IARPA Babel program and compared to state-of-the-art n-gram backoff language models trained with Kneser–Ney smoothing. Additionally, language models that combine both approaches by interpolating between two models are created and tested in speech recognition tasks. Those interpolated models are able to obtain consistent word error rate reductions up to 0.6% absolute for Vietnamese and 0.8% for Lao.

Contents

1. Introduction

1.1 Motivation

Language modeling has been employed in many natural language processing applications including speech recognition, machine translation, part-of-speech tagging, information retrieval and text generation. Those models can be used for tasks involving prediction of the next word in a speech or text sequence. In automatic speech recognition (ASR) systems, they form one of the three components of the decoder and are crucial to the performance of the whole system. The block diagram of a typical ASR system is shown in Figure 1.1. The other two parts of the decoder, acoustic model and dictionary, are used to find word sequences that match the features extracted from the signal. Since they only utilize acoustic information, they have the flaw of being unable to differentiate between homophonic word sequences. Hence, language models are required to choose between homophones and additionally, they can significantly reduce the search space for possible word combinations.

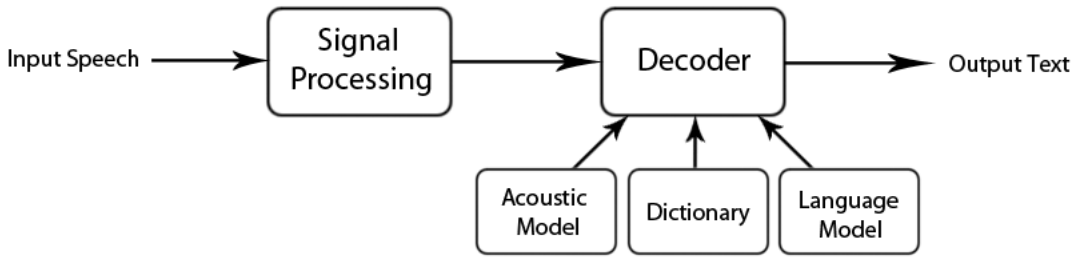

Figure 1.1: Block diagram of a standard ASR system.

Language models try to capture the properties of a natural language and provide syntactical and semantical knowledge on those languages. They are capable to distinguish grammatically well-formed word sequences from incorrect ones and meaningful phrases from nonsensical ones. There are many ways to model languages. Especially statistical methods have been highly researched and successfully employed

in various kinds of applications. N-gram language modeling is a commonly applied technique which uses a large text corpus to estimate the distribution of a language. With sufficient training data, n-gram models that show satisfactory performance can be easily created. But in case of data sparseness, particularly for low resource languages, n-gram models usually do not provide such good performance. Hence, other approaches are necessary for languages that are rarely spoken.

Another method of statistical language modeling utilizes neural networks to perform probability estimation in a continuous space. Neural networks are one of the most popular and promising fields of artificial intelligence research. They process information by simulating the human brain and are capable to solve a wide variety of tasks that can not be solved using ordinary algorithmic approaches. Their ability to learn by example makes them very flexible and powerful. Neural networks are used for various tasks such as pattern recognition, machine learning, clustering, feature detection and data compression. Recent competitions show the superiority of deep neural networks compared to alternative techniques especially in pattern recognition and machine learning [1]. Neural networks have also been researched and successfully applied in many other subject areas, for example in the medicine to recognize diseases from scans and for business purposes.

Deep neural network architectures have been efficiently used for language modeling as proven in works such as [BDVJ03], [ScGa02] and [Schw07]. In all those works, neural network language models that are able to provide a better generalization ability than n-gram language models in case of less data have been created. But most of the existing works deal with languages like English, German or French, where data sparseness is not such a big issue and with the huge amount of available data, n-gram models that show equally good performance have already been built. There has not been many works that deal with low resource languages, where n-gram modeling is less effective and new techniques like neural network models are needed. Hence in this work, we employ deep neural networks to perform continuous language modeling for low resource languages such as Vietnamese, Tamil and Lao. Networks of various sizes are thoroughly trained and evaluated to receive best performance. A comparison of neural network language models with state-of-the-art n-gram language models is carried out and those two approaches are combined to achieve even better performance. At last, the combined models are integrated into a speech recognition system to test their ability in practical tasks.

1.2 Overview

The first part of this work consists of an introduction to basics needed to understand different techniques of statistical language modeling. Chapter 2 provides background knowledge for n-gram language models by deriving the necessary theory using probabilistic methods. Then it describes three metrics for evaluating language models that are used in this work which are word error rate, cross-entropy and perplexity. To improve standard n-gram language models, techniques called smoothing have been developed. An illustration of three relevant smoothing techniques, interpolation, backoff and Kneser-Ney smoothing, can also be found in chapter 2. The next chapter gives an overview to neural networks and learning methods. A brief summary of related works where different types of neural networks are used for language

[1] http://www.kurzweilai.net/how-bio-inspired-deep-learning-keeps-winning-competitions

modeling and other tasks forms the second half of this chapter. Chapter 4 describes the architecture of the neural network used in this work. It provides a detailed description of each of the three layer types of the architecture, which are the projection, hidden and output layer, and the backpropagation algorithm for training the network. Chapter 5 explains how the architecture and additional functionalities can be efficiently implemented and introduces the tools Theano and SRI Language Modeling Toolkit that are used to accomplish these tasks. An overview to the Janus Recognition Toolkit used in our speech recognition system is also given here. Results of experiments on the neural network language models can be found in chapter 6. The networks are evaluated with respect to the time necessary for model training, performance on the test data and error reduction in speech recognition tasks. This work concludes by summarizing the most important results and giving possible goals for future works.

2. Background

Language modeling is used to improve applications for natural language processing by capturing the properties of a language. In automatic speech recognition systems it provides another independent information source next to the acoustic model to predict the next word in a speech sequence. Especially problems like homophones or large search space can be solved. Using a deterministic model, for example based on formal grammar, a given word sequence can only be accepted or rejected, which is inappropriate for spoken language. Therefore a statistical approach is commonly used where probabilities are assigned to word sequences to estimate the distribution of a natural language. A widely used method to make these probability estimations is called n-gram language modeling.

2.1 N-Gram Language Models

Given a word sequence $w_1, ..., w_k$, the probability $P(w_1, ..., w_k)$ can be decomposed according to rules of conditional probability as

$$P(w_1, ..., w_k) = P(w_1)P(w_2|w_1)P(w_3|w_1, w_2)...P(w_k|w_1, ..., w_{k-1}) \ .$$

Hence we need the conditional probability $P(w_k|h)$ where $h = w_1, ..., w_{k-1}$ is called the history or context of word w_k. To make estimations for the whole language, it is necessary to calculate $P(w_k|h)$ for all combinations of words from the vocabulary. The complexity grows exponentially with the vocabulary size and length of the history, so the precalculation of those probabilities for every history is in most cases an impossible task.

A practicable solution is to replace the entire history by a feasible equivalence class of the history such that $P(w_k|h) \approx P(w_k|[h])$. Under the Markov assumption, which says that the future behavior of a dynamical system only depends on its recent history, the full context can be shortened to the n-1 previous words. With the following definition of equivalence classes

$$[w_1, ..., w_{k-1}] = w_{k-n+1}^{k-1}$$

where w_{k-n+1}^{k-1} denotes the word sequence $w_{k-n+1}, ..., w_{k-1}$, the conditional probability can be approximated by

$$P(w_k|w_1, ..., w_{k-1}) \approx P(w_k|w_{k-n+1}^{k-1}) \ .$$

For the joint probability of the word sequence yields

$$P(w_1, ..., w_k) = \prod_{i=1}^{k} P(w_i|w_{i-n+1}^{i-1}) \ .$$

Generally, a language model based on n−1 preceding words is called an n-gram model. The words unigram, bigram and trigram denote n-gram models with n = 1, n = 2 and n = 3, respectively. In case of unigram, the number of previous words is zero and the conditional probability is approximated by $P(w_k)$ which means that the probability of each word is independent of its context. To calculate probabitilies for words at the beginning and end of a sentence, two distinguished tokens, <s> and </s>, symbolizing the start and end of a sentence, are padded to all sentences and treated as additional words in the vocabulary. Also the token <unk> is often appended to the vocabulary to replace out of vocabulary (OOV) words.

2.1.1 Estimation of N-Grams

The standard approach for n-gram models to estimate $P(w_k|w_{k-n+1}^{k-1})$ is the maximum likelihood estimation. It uses a large amount of training corpus and determines the frequency of word sequences by counting their occurences in the text. For example the likelihood of unigrams can be computed by

$$P_{ML}(w_k) = \frac{c(w_k)}{N}$$

where $c(w_k)$ denotes the count of the word in the text and N the total number of words in the training corpus. The conditional probability for a word given its history can be calculated with the count of the whole word sequence normalized by the number of times the history occurs as follows

$$P_{ML}(w_k|w_{k-n+1}^{k-1}) = \frac{c(w_{k-n+1}^{k})}{c(w_{k-n+1}^{k-1})} \ .$$

However, there is a severe problem in n-gram modeling directly derived from frequency counts. When confronted with word successions that have not been seen in the training corpus, the probability becomes zero. This happens due to data sparseness since the training corpus is always limited. But for an n-gram to receive a zero probability does not necessarily mean that it is an impossible occurence in reality. Those can merely be rare events with no representative in the training data. In speech recognition systems, a zero probability can lead to errors, as it disallows the word sequence regardless of how informative the acoustic signal is. Hence some sort of algorithm is necessary to assign nonzero probabilities to unseen words or n-grams. Smoothing is one of the methods used to address this problem. But before introducing smoothing techniques relevant to this work, some useful evaluation metrics for language modeling are presented in the following section.

2.2 Evaluating Language Models

When comparing different language models, an appropriate evaluation metric is needed. Of course, the easiest way is to run the systems using those language models and check which one produces fewer errors. But usually the performance of an entire system depends on other factors or the combination of its components as well. Evaluating language models by system test runs can also be expensive and time consuming. So it would be better to have a system independent measurement. In the following the three common measures word error rate, cross-entropy and perplexity are introduced which are also used for evaluation in this work.

2.2.1 Word Error Rate

A standard metric for the performance of speech recognition or machine translation systems is the word error rate (WER). The WER is derived from the Levenshtein distance and is calculated by contrasting the recognized word sequence with the reference word sequence. The general difficulty of measuring performance lies in the fact that the two word sequences can have different lengths which makes a word by word comparison unsuitable. This problem is solved by first performing alignment search between output hypothesis and reference sequence. Therefor missing words are inserted to and needless words are deleted from the hypothesis to have two word sequences of the same length for comparison. Then the word error rate can be computed as

$$WER = \frac{S + D + I}{N}$$

where S is the number of substitutions, D the number of deletions, I the number of insertions and N the number of words in the reference word sequence.

2.2.2 Cross-entropy and Perplexity

In information theory, entropy is a measure of the average uncertainty in a random variable and describes how predictable the information is. Natural language can also be seen as an information source emitting words from the vocabulary. A probabilistic language model which is an estimation for the true distribution of the language can be evaluated using the cross-entropy H for two distributions over the same probability space according to the following formula

$$H(p, q) = -\sum_{x} p(x) \log_b q(x)$$

where p is a true distribution and q an estimation for p. Since the true distribution for a language is unknown, an estimation has to be made. For a given text composed of sentences $s_1, ..., s_n$ the cross-entropy of a language model can be computed by

$$H(P) = -\sum_{i=1}^{n} \frac{1}{N} \log_b P(s_i)$$

where P is the given language model and N the total number of words in the sample text. Calculation of the cross-entropy can be interpreted as evaluating how difficult it is for the language model to predict the next word in a word sequence. It is

generally assumed that a lower cross-entropy and thus an easier prediction task for the model corresponds with a language model that performs better in applications.

From the cross-entropy the more frequently used evaluation metric called perplexity is derived. Perplexity evaluates the branching factor of a language model and like cross-entropy, models with less perplexity are better than models with higher perplexity. The perplexity *ppl* of a language model P is defined as

$$ppl(P) = b^{H(P)} = b^{-\sum_{i=1}^{n} \frac{1}{N} \log_b P(s_i)} .$$

The use of log probabilities has computational advantage over using real probabilities since the product of probabilities can be replaced with the more efficient summation of the values. It also improves numerical stability since the logarithm of a number in the $[0, 1]$ interval can be represented more accurately than the number itself. The base b of the log probability is usually chosen to $b = 2$ or $b = 10$.

2.3 Smoothing Techniques

Smoothing describes methods for adjusting maximum likelihood estimation models to hopefully produce more accurate probabilities, particularly for unseen data. The main task is to assign nonzero probabilities to all word sequences and thus avoid zero counts for sequences that were not seen during training. This can be achieved by for example collecting some of the existing probability mass from well observed events, which is called discounting, and distributing it to unseen events. A simple smoothing technique illustrated in [Lids20] is called add-one smoothing and extends the maximum likelihood estimate by pretending that each n-gram occurs one time more than it actually did. However, this scheme has the flaw of assigning the same probability to every unseen n-gram although those may differ in their likelihoods, and shows poor performance as for example stated in [GaCh90] and [GaCh94]. Many advanced smoothing techniques have been developed over the years. Some of the smoothing methods do not only prevent zero probabilities, but they also attempt to improve the accuracy of the model as a whole. In the following sections we introduce some basic smoothing techniques and methods relevant to this work.

2.3.1 Linear Interpolation

Linear interpolation is a class of smoothing techniques that involve combining higher-order probabilities with information from lower-order models. When there is little data for directly estimating an n-gram probability, useful information can be provided by the corresponding (n−1)-gram probability estimation. A simple method for model interpolation is described by Jelinek and Mercer in [JeMe80]. For example a trigram maximum likelihood model can be interpolated with a bigram maximum likelihood model like in the following

$$P_{interp}(w_k|w_{k-2}, w_{k-1}) = \lambda P_{ML}(w_k|w_{k-2}, w_{k-1}) + (1 - \lambda) P_{ML}(w_k|w_{k-1})$$

where λ is a weighting factor to be set experimentally. Since the occurence of a bigram is more likely than that of a trigram, the lower-order model, and with it the interpolation result, may have a nonzero probability. Brown et al. presented in their work [BPSL+92] an elegant way of model interpolation where the nth-order

smoothed model is defined recursively as a linear interpolation between the nth-order maximum likelihood model and the (n−1)th-order smoothed model. The formula is as follows

$$P_{interp}(w_k|w_{k-n+1}^{k-1}) = \lambda(w_{k-n+1}^{k-1})P_{ML}(w_k|w_{k-n+1}^{k-1})+(1-\lambda(w_{k-n+1}^{k-1}))P_{interp}(w_k|w_{k-n+2}^{k-1})$$

where λ now depends on the word context and can be set to a fixed value. However, to obtain better performance, the interpolation weight is often trained in variations of interpolation smoothing such as held-out interpolation and deleted interpolation.

2.3.2 Backoff Smoothing

Although the interpolation approach may provide feasible probabilities to unseen n-grams, in case of nonzero counts, interpolated models still use information from the less accurate lower-order models. Backoff smoothing has been developed to address this problem. Like interpolation, it combines higher-order and lower-order models, but uses only the higher order models when it is possible, for example when there were enough counts in the training corpus. If the higher order models are not that informative, it backs off to lower-order models, so the model with the most reliable information is always used. The backoff algorithm can be defined recursively as

$$P_{backoff}(w_k|w_{k-n+1}^{k-1}) = \begin{cases} P^*(w_k|w_{k-n+1}^{k-1}) & \text{if } c(w_{k-n+1}^{k}) > 0 \\ \alpha(w_{k-n+1}^{k-1})P_{backoff}(w_k|w_{k-n+2}^{k-1}) & \text{otherwise} \end{cases}$$

where P^* is a discounted probability model and α is the backoff weight of the history which is necessary for $P_{backoff}$ to be normalized.

Backoff smoothing is central to several advanced smoothing techniques. Katz extends in [Katz87] the intuitions of backing off higher-order models with lower-order models by using Good-Turing discounting described in [Good53]. Katz smoothing has become a widely used smoothing technique in speech recognition. A similar approach by Church and Gale can be found in [ChGa91].

2.3.3 Kneser-Ney Smoothing

Kneser and Ney introduced in [KnNe95] a way to improve interpolation and backoff smoothing methods. The lower-order model for both smoothing methods is generally taken to be a smoothed maximum likelihood model which is sometimes unsuitable for estimating higher-order probabilities. Instead of a probability model that describes how likely the lower-order n-gram is, Kneser and Ney suggested to use the probability of how likely is the lower-order n-gram to appear as a continuation for different words. The following probability model is used by Kneser-Ney smoothing

$$P_{KN}(w_k|w_{k-n+2}^{k-1}) = \frac{|\left\{w_{k-n+1} : c(w_{k-n+1}, w_{k-n+2}^{k-1}, w_k) > 0\right\}|}{\sum_{w_k'} |\left\{w_{k-n+1} : c(w_{k-n+1}, w_{k-n+2}^{k-1}, w_k') > 0\right\}|}$$

which sets the lower-order probability proportional to the number of different words the (n−1)-gram completes. The backoff Kneser-Ney for example is then given by

$$P_{KN}'(w_k|w_{k-n+1}^{k-1}) = \begin{cases} P_{abs}(w_k|w_{k-n+1}^{k-1}) & \text{if } c(w_{k-n+1}^{k}) > 0 \\ \alpha(w_{k-n+1}^{k-1})P_{KN}(w_k|w_{k-n+2}^{k-1}) & \text{otherwise} \end{cases}$$

where P_{abs} denotes a discounted probability model using the absolute discouting method described in [NeEs91]. It is defined by subtracting a fixed discount d from each nonzero frequency count of the maximum likelihood estimate as follows

$$P_{abs}(w_k|w_{k-n+1}^{k-1}) = \frac{\max\left\{c(w_{k-n+1}^{k}) - d, 0\right\}}{\sum_{w_k'} c(w_{k-n+1}^{k-1}, w_k')} \ .$$

Kneser-Ney smoothing is another smoothing technique that is commonly used for speech recognition and machine translation. A variation of Kneser-Ney smoothing, which is refered to as modified Kneser–Ney smoothing, generally shows good performance.

2.3.4 Comparison of Smoothing Techniques

Chen and Goodman presented in their work [ChGo96] a comparison of several widely used algorithms for smoothing n-gram language models like the ones mentioned in previous sections, but except for Kneser-Ney smoothing. Additionally to the existing smoothing techniques they developed two novel methods that they called the average-count and one-count method. Chen and Goodman carried out experiments over many training set sizes on different training corpora using bigram and trigram language models. They investigated how these factors affect the relative performance of smoothing methods. Their smoothed language models were measured through the cross-entropy on the test data.

The experiments demonstrated that performance could vary widely with respect to training data size, training corpora and n-gram order with the most significant factor being the data size. Their training data size ranged from 100 up to 10 M sentences and experiments displayed a high correlation between corpus size and cross-entropy on the test data. The performance of their baseline implementation using Jelinek-Mercer smoothing is shown in Figure 2.1. While additive smoothing had always poor performance, Katz smoothing and held-out interpolation generally performed well with cross-entropies significantly less than the baseline method in almost all situations, except for cases with very little training data. The held-

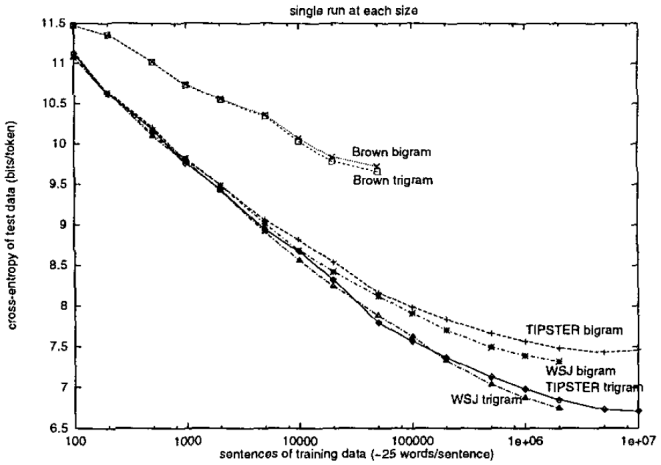

Figure 2.1: Cross-entropy of baseline bigram and trigram models trained on different corpora using up to 10 M sentences. Source: [ChGo96].

out interpolation algorithm sometimes performed better than Katz smoothing in sparse data situations, but was outperformed by Katz smoothing on trigram models produced from large training sets and on bigram models in general. Their novel methods average-count and one-count were superior to existing methods for trigram models and performed well on bigram models. For example, the relative performance of their smoothed trigram models is shown in Figure 2.2.

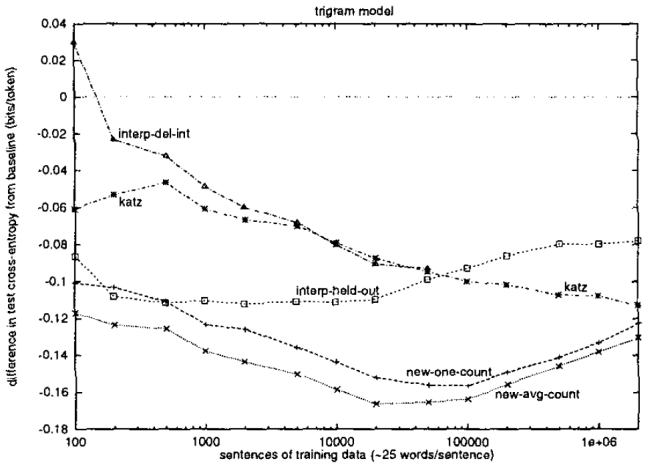

Figure 2.2: Relative performance of various trigram models on Wall Street Journal corpus with respect to baseline. Source: [ChGo96].

While smoothing is one technique to address sparse data issues, performance of smoothed language models still depends highly on the size of training data used. A different approach to fight the data sparseness problem is to perform the language model probability estimation in a continuous space by using neural network architectures. Neural networks are capable of learning from data samples through a training process, for example to learn a distributed representation for semantically related word sequences from word sequences in the training corpus. The following chapter firstly gives an overview to the backgrounds required for understanding neural networks before introducing this new approach of statistical language modeling.

3. Neural Networks

Neural networks are mathematical models for information processing that are inspired by biological nervous systems, in particular the brain. They are composed of simple processing elements, the neurons, and sets of adaptive weights which describe the link strength between neurons. Neurons are connected together to form a network and work in parallel to propagate information through the network to solve specific problems. Just like people, neural networks learn by example through a training process where adjustments to the connection weights between the neurons are carried out. In this way, identically constructed neural networks can be used to perform different tasks depending on the received training.

Neural networks are capable of approximating non-linear functions of their inputs and have been used to solve a wide variety of tasks, especially tasks where no problem solving algorithm exists. A well trained neural network has the ability to recognize trends in the given data and similarities among different input patterns, especially those that have been corrupted by noise. Due to their massive parallel nature, neural networks can be efficiently used for real time operations. They are also very fault tolerant such that when an element of the network fails they can still continue processing without a significant loss of performance. Neural networks have been successfully employed in applications for pattern recognition and machine learning. They also offer improved performance over conventional technologies in areas which include machine vision, signal filtering, virtual reality, data compression, data mining and many more.

3.1 Neural Network Basics

A neural network is represented as a group of nodes which are connected with each other. Usually, the nodes are arranged into a hierarchical structure consisting of disjoint layers but neural networks that cannot be easily grouped into layers do exist as well. A simple three layered neural network is shown in Figure 3.1. Each node represents an artificial neuron and the edges represent connections from the output of one neuron to the input of another with the arrows denoting the direction of information flow.

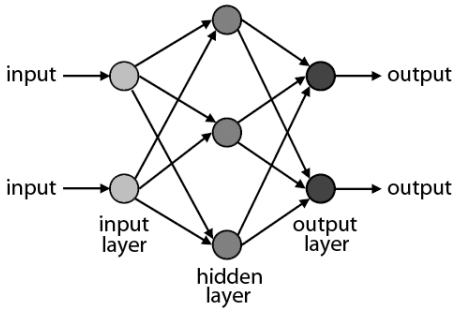

Figure 3.1: A simple neural network.

Typically, a neural network consists of an input layer for interactions with the environment to receive input, an output layer to present the processed data and hidden layers lying in between that do not have any interactions with the environment. Increasing the complexity of a neural network, and thus its computational capacity, requires the addition of more hidden layers, and more neurons per layer.

A neuron is a simple processing unit with many inputs and one output. The inputs are usually weighted with a value from the interval $[-1, 1]$, i. e. the effect that each input has on the output calculation depends on the weight of the particular input. Figure 3.2 displays exemplarily how input signals are processed inside a neuron.

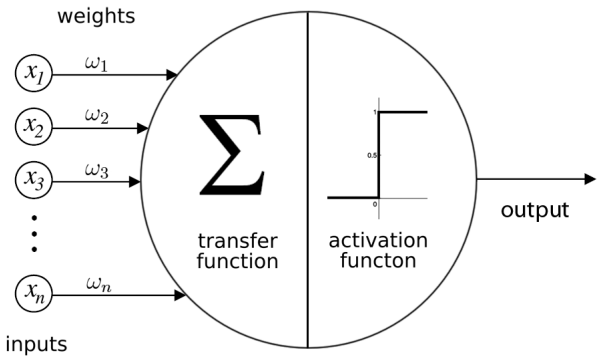

Figure 3.2: Processing pattern within a neuron.

Signals which are delegated through the network to the neuron are firstly passed to the transfer function Σ. It is usually defined as a summation of the weighted inputs as proposed by McCulloch and Pitts in [McPi43]. The computation of the weighted sum can also be interpreted as the inner product of the input vector and the weight vector as in the following

$$\Sigma(\mathbf{x}, \mathbf{w}) = \sum_{i=1}^{n} x_i \omega_i = \mathbf{x} \cdot \mathbf{w}$$

where $\mathbf{x} = (x_1, ..., x_n)$ and $\mathbf{w} = (\omega_1, ..., \omega_n)$. Such a McCulloch-Pitts neuron is also called a perceptron. The result of the transfer function is then passed to the activation function which sets the inner activation state of the neuron. There are a number of commonly used activation functions such as the step function and

the sigmoid function shown in Figure 3.3. Sometimes the output of the activation function is further processed by an output function, but commonly the result of the activation function is emitted as the final output of the neuron.

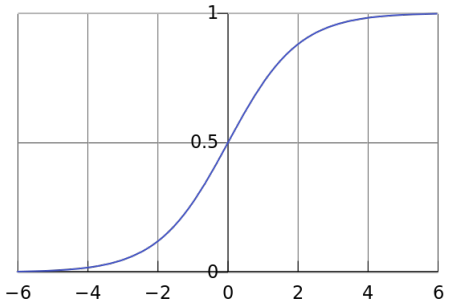

Figure 3.3: Graph of a sigmoid function. Source: [1]

3.1.1 Learning

Neural networks are not programmed. They typically learn during a training phase using a specific learning algorithm to update the weights or other parameters during each training iteration. Once the network has been trained, it enters a production phase where it produces results independently. There are numerous algorithms available for training neural network models. They commonly use a set of observations to find an optimal solution for a given task. Therefor a cost function is defined such that no other solution has a cost less than the optimal solution. Learning algorithms search through the solution space to find a set of parameters that has the smallest possible cost. The cost function is chosen depending on the desired task and the used learning paradigm.

A differentiation is mainly made between two learning paradigms, the supervised and unsupervised learning method, which reflect how the training data is presented to the neural network. In supervised learning, a set of example pairs consisting of input data and desired output is given and the aim is to find network parameters to match the examples. The cost function is then defined to evaluate the mismatch between the output of the network and the data. A widely used cost function is the mean-squared error function. From the combination of supervised learning and gradient descent optimization derives the well-known backpropagation algorithm which is also used in this work and will be described in section 4.5. In unsupervised learning, only the input data is given and there is no example output to correct the network. The cost function can then be chosen according to prior knowledge about the problem domain.

To train a neural network efficiently, one first needs to have a good training corpus which is representative and without redundancy. The examples must be selected carefully otherwise useful time is wasted for training or, in the worst case, the network might be functioning incorrectly. Neural networks can also be unpredictable since they need to find out how to solve problems by themselves. Training by minimizing the cost function does not always guarantee to find the global optimum as the cost function may have many local minima. It can sometimes lead to divergence

[1] http://en.wikipedia.org/wiki/Sigmoid_function

of the network if the computation gets too far away from a local minimum. There is also the possibility of overfitting the network on the training data that arises from overtraining the network. The model begins to memorize the training data and thus becomes incapable to generalize unseen events. To avoid this problem, the training corpus is usually divided into different datasets for cross-validation. Some form of regularization is also commonly used to adress the problem of overfitting.

3.2 Deep Neural Networks

A deep neural network is defined as a neural network with at least one hidden layer of units between the input and output layer. It has more modeling capability compared to shallow networks due to the extra layers and nonlinearity. There is a wide variety of deep architectures with different computational abilities. Simple deep networks are typically designed as feedforward networks where input signals only propagate forwards from the input layer to the output layer. The multilayer perceptron is a well known feedforward network that derives from the perceptron algorithm in [Rose57]. It is used in this work for language modeling and a detailed description can be found in chapter 4. Further examples of feedforward networks are the radial basis function networks introduced in [BrLo88] and the self-organizing maps from [Koho82]. Recent research has also successfully applied the deep learning architecture to recurrent neural networks which are, in contrast to feedforward networks, models with data flow in both directions. Those networks are very powerful and can get extremely complicated. Examples of recurrent networks can be found in [Hopf82] and [HiAS85] which describes the Hopfield networks and Boltzmann machines, respectively.

3.2.1 Continuous Space Language Models

The idea of using deep neural networks for language modeling is to address the data sparseness problem by performing language model probability estimation in a continuous space. A description of neural network language models for continuous speech recognition can be found in [Schw07]. Schwenk utilized multilayer perceptrons consisting of three layers which were a projection layer with 50 to 250 neurons, a hidden layer with 200 to 1500 neurons and an output layer with a size equal to the length of the word list. The architecture can be found in Figure 3.4. The neural network language models were evaluated on several large vocabulary speech recognition tasks for English, French and Spanish. Short-lists were used to limit probability calculation to frequent words instead of all words from the vocabulary and the models were interpolated with backoff language models for probabilities of words that were not in the short-list. In all the described experiments, the neural network language models were compared to 4-gram backoff models where modified Kneser–Ney smoothing was used.

For English conversational speech Schwenk employed neural networks with a short-list of 2 k words and text corpora from 7.2 M to 27.3 M words. A perplexity reduction of about 9% relative to the baseline model could be obtained independently of the training data size, for example from 62.4 to 57.0 using 7.2 M data. Consistent WER reductions of 0.4% to over 0.8% absolute could be achieved with the maximum improvement being from 23.04% to 22.19%. The speech recognizer for French broadcast news with a word list of 200 k and a short-list of 12 k words could

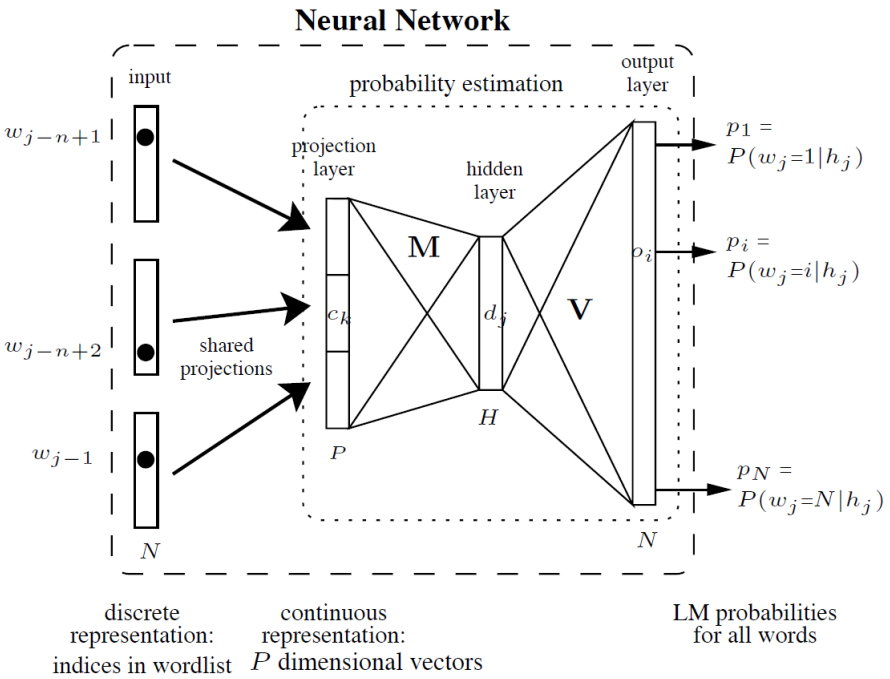

Figure 3.4: Neural network language model for continuous language modeling. Source: [ScGa04].

decrease the perplexity from 74.2 to 71.4 and the WER from 10.74% to 10.51%. Schwenk further carried out tests on English and Spanish parliament speeches with about 35 M words of training data per language. For the English system with a short-list of 4 k words, a relative perplexity reduction of 15% from 106.4 to 90.2 was obtained and the WER improved from 10.84% to 9.95%. The short-list of the Spanish system used 2 k words and the neural network language model achieved a relative improvement in perplexity of 10% and a WER reduction of 0.64% absolute.

3.2.2 Language Models with Structured Output Layers

Neural network language models with structured output layers (SOULs) were introduced in the paper [LOAG+11]. The SOUL neural networks extended the standard approach of Schwenk by a hierarchical structure of the output layer which was described in [MnHi08]. To structure the output vocabulary, words were clustered and represented by a binary tree, and probabilities of the paths in this binary tree given the history were estimated, rather than directly the word itself. With this novel technique, vocabularies of arbitrary size could be handled without the use of short-lists. Like the multilayer perceptron, training was performed using the backpropagation algorithm.

The SOUL language models were experimented on Mandarin Chinese speech-to-text tasks with a vocabulary size of 56 k words. For the baseline system, 4-gram language models were trained on 48 different corpora of overall 3.2 billion words, smoothed using Kneser-Ney discounting and linearly interpolated to receive a well tuned word-based language model. The neural network language models using short-lists and SOULs were both trained on various corpora of about 25 M words and for each test configuration 4 models of the same type were interpolated. Experimental results

have shown that the baseline perplexity of 211 could be reduced to 187 and 185 by an interpolation of the baseline 4-gram model with a 4-gram neural network model with a short-list of 8 k and 12 k words, respectively. Instead of WER, character error rate (CER) was used to evaluate recognition performance for Mandarin. A decrease of the CER from 9.8% to 9.5% and 9.4% could be achieved by the two models. Replacing the standard short-list with a full-vocabulary SOUL language model, the perplexity could be further improved to 180 and the CER to 9.3%. Increasing the order of the neural networks from 4 to 6 could achieve additional performance gain with the 6-gram SOUL language model scoring a perplexity of 162 which outperformed the 12 k short-list model by 10, and a CER of 9,1%.

3.2.3 Recurrent Neural Network Language Models

Beside feedforward neural networks, recurrent architectures have also been successfully applied for natural language modeling as described in [MKBC+10]. While feedforward approaches usually utilize a fixed length context that has to be specified before training, the recurrent models employed an unlimited size of context and were capable of learning long-term dependencies. The network architecture was a simple recurrent neural network which was also called Elman network in [Elma90]. As shown in Figure 3.5, the network consisted of an input layer, an output layer and two context layers with 30 to 500 hidden units each. To train the networks, the standard backpropagation algorithm and a variation called backpropagation through time were used.

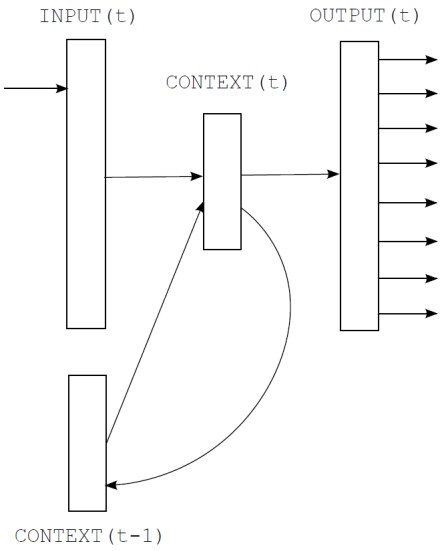

Figure 3.5: Simple reccurent neural network for language modeling. Source: [MKBC+10].

The recurrent network based language models were interpolated with 5-gram backoff language models smoothed with modified Kneser-Ney where the interpolation weight for the neural network model was set to 0.75. Evaluation was performed on several standard speech recognition tasks for English by comparing the combined models with the 5-gram backoff models. On Wall Street Journal corpora the perplexity could be reduced from 221 to 156 and the WER decreased by 1.8% absolute from 13.5%

to 11.7% using a training data size of 6.4 M words. With a smaller training corpus of 200 k words, the perplexity improved from 336 to 271 and the WER from 16.4% to 15.4%. These enormous improvements were due to the not well tuned baseline systems which were probably easy to outperform. Further experiments were carried out on state-of-the-art systems where the the baseline 4-gram backoff models were trained on 1.3 G words while the recurrent network model only used 5.4 M words. A significant performance gain on the WER could be achieved by the combined model which reduced the baseline WER of 24.1% to 23.3% by 0.8% absolute.

3.2.4 Boltzmann Machines for Machine Translation

A novel approach of using language models based on Restricted Boltzmann Machines (RBMs) for statistical machine translation was presented in [NiWa12]. As it is common for RBM architectures, the neural networks consisted of two interconnected layers which were a visible input layer representing words of n-grams and a hidden layer with binary units. Instead of using error backpropagation for learning, the networks were trained with contrastive divergence which was introduced in [Hint02]. The RBM-based language models were trained on different kinds of corpora such as parliament and news speeches, and evaluated on German to English and English to French translation tasks with the Bilingual Evaluation Understudy (BLEU) algorithm.

Experiments were carried out to analyse the influence of the hidden layer size on translation quality where results have shown that best performance could be achieved with a hidden layer of 32 units. For German to English lecture translation tasks, a 4-gram RBM-based language model was trained and compare to an n-gram-based language model of the same order. The translation quality on the test data could be improved by 1.45 BLEU points from 23.02 to 24.47. A combination of an ngram-based in-domain language model and the RBM-based language model could further achieve a performance gain of 0.4 BLEU points. RBM-based language models were also tested on English to French lecture translation tasks. In this case, the baseline system already utilized several good n-gram based language models and the RBM language model could still achieve an improvement of 0.1 BLEU points from 31.90 to 32.03.

3.2.5 Deep Architechtures for Other Tasks

Deep neural networks have not only been successfully employed to model natural languages but also for other automatic speech recogniction (ASR) tasks such as acoustic modeling and signal preprocessing. An overview of the progress of deep neural networks on acoustic modeling can be found in [HDYM$^+$12]. Deep neural networks have been shown to outperform Gaussian mixture models on a variety of speech recognition benchmarks, sometimes by a large margin. For example the deep belief networks introduced in [Hint09] have achieved better phone recognition as shown in [MoDH12]. In [WHHS$^+$89], a Time-Delay Neural Network was presented for phoneme recognition which was able to discover temporal relationships between acoustic-phonetic features and outperformed systems based on Hidden Markov models (HMMs) at various recognition tasks. Deep neural networks were also applied for isolated word recognition as described in [WuCh93] and they achieved better

recognition rate than HMMs as well. Papers like [Liao13] explain, how deep neural networks can be used for speaker adaption to further improve speech recognition accuracy. As described in [HuZa10], neural networks can perform nonlinear transformations of acoustic signal features to reduce feature dimensionality which achieved considerably higher recognition accuracies compared to linear dimension reduction methods. In [MLOV$^+$12], a deep recurrent neural network was used to denoise input features and the model was competitive with existing feature denoising approaches.

Beside ASR, deep architectures have been applied to many other fields such as classification, computer vision, optimization and robotics, where they were shown to produce state-of-the-art results on various tasks. In [LLPN09], deep belief networks were applied to learn feature representations from unlabled audio data for classification where they presented very good performance. Deep networks can also be used to classify high-dimensional patterns with minimal preprocessing such as handwritten characters and high-resolution images where they were able to outperform various other techniques as stated in [LBBH98] and [KrSH12]. While traditional methods of computer vision cannot match human performance, neural networks have been able to achieve near-human performance on handwriting tasks. It was shown in [CiMS12] that they even outperformed humans by a factor of two on a traffic sign recognition benchmark. In [LRMD$^+$12], a multilayer network was trained to build a high-level feature detector which was able to detect faces on unlabeled images and achieved a 70% relative improvement over previous state-of-the-art methods.

4. Architecture

The architecture used in this work extends the neural network language model from [Schw07] by adding more complexity and computational power to the neural network. As proposed by Schwenk, the neural network is implemented as a feedforward network arranged into different layers. Each layer is fully connected to the next one which means that each node in one layer connects with a certain weight to every node in the following layer. The neural network employs nonlinear activation functions and uses the backpropagation algorithm for model training which makes it a standard multilayer perceptron. It consists of three or more layers which are an input layer, an output layer and one or more hidden layers. The input layer does not only provide interactions with the environment to receive input signals but is also responsible of projecting all the words in the history onto a continuous space. Hence it is commonly called the projection layer. Words are not directly passed to the network as inputs. Instead, indices that describe their position in the vocabulary are used. The role of the output layer is to emit the result of the network computation. The outputs of the network are the posterior probabilities of all words in the vocabulary or short-list indicated by their indices given the history. Layers between the projection and output layer are hidden layers that are necessary to achieve non-linear probability estimation. The architecture of the neural network language model is shown in Figure 4.1.

4.1 Projection Layer

The projection layer converts the inputs of the network, discrete word indices of the n−1 previous words $w_{k-n+1}^{k-1} = w_{k-n+1}, ..., w_{k-1}$ in the vocabulary of size N, into a reduced continuous space of dimension M with $M \ll N$. Instead of directly feeding in the indices, a 1-of-n coding scheme is used. Therefore each word in the history is represented by an N dimensional row vector of binary valued elements where the ith word of the vocabulary is coded by setting the ith element of the vector to 1 and all the other elements to 0.

Projection onto the continuous space is carried out by multiplying the coded index vector with the $N \times M$ dimensional projection weight matrix $\mathbf{P}$. This matrix multiplication derives from the transfer function of the neurons which is defined as the

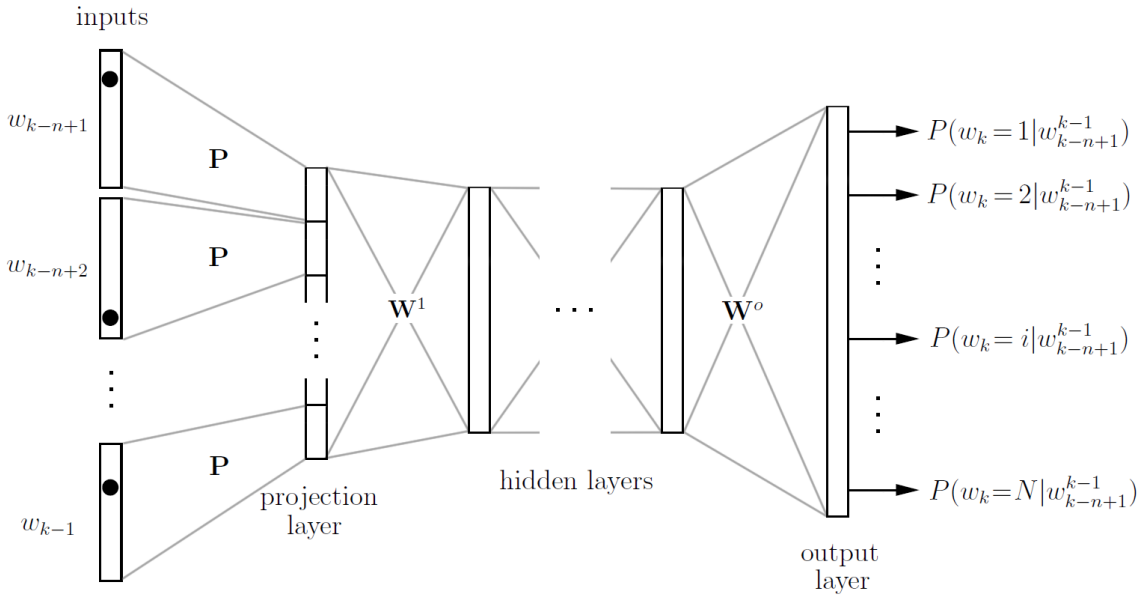

Figure 4.1: Architecture of the neural network language model.

standard weighted sum. Each neuron in this layer is connected with every entry of one input vector $\mathbf{x}$ and the connections are described by weights p_{jk} where j and k denote the link between the jth element of the input vector and the kth neuron. So the columns of the projection matrix correspond to the weights of a single neuron. For the weighted sum yields

$$\Sigma(\mathbf{x}, \mathbf{P}) = \sum_{j=1}^{N} x_j p_{jk} = \mathbf{x} \cdot \mathbf{P} \ .$$

Due to the 1-of-n coding the multiplication calculation can be substantially simplified. Given that the input vector only has one nonzero entry at position i, one only needs to copy the ith row of the projection matrix to receive the multiplication result. Since a projection is needed for each word of the context, the projection layer for an n-gram language model is built up of $(n-1) \cdot M$ neurons and the projection weight matrix is shared by all words from the context. Neurons in the projection layer differ from those of other layers by not using a nonlinear activation function. The output of such a neuron is simply the result vector of the projection, thus the final output of the projection layer is formed by concatenating the continuous projections of all the words in the context. This yields to an output vector of length $(n-1) \cdot M$ which is used as the input for the next layer in the network.

4.2 Hidden Layer

Hidden layers have a blackbox character since they do not have any interactions with the environment. They are used to achieve the ability to approximate non-linear functions that is necessary for probability estimation. The output of the projection layer is processed by the first hidden layer which propagates its own output to the next hidden layer and so on. The complexity of the neural network depends on the number of hidden layers and the size of each hidden layer. Optimal size for a hidden

layer depends on the size of the training corpus and is therefore a system parameter that has to be tuned during experiments.

Like the projection layer, hidden layers employ transfer functions in form of a weighted sum. The number of weights associated with each hidden layer neuron is defined by the dimensionality of the input vector from the previous layer. As activation function all hidden layer neurons employ the hyperbolic tangent function which has similar characteristics as the logistic sigmoid curve shown in Figure 3.3. The tanh function generates continuous values in $[-1, 1]$ and is defined as

$$\tanh(\mathbf{t}) = \frac{e^{\mathbf{t}} - e^{-\mathbf{t}}}{e^{\mathbf{t}} + e^{-\mathbf{t}}} \ .$$

For a vector-valued $\mathbf{t}$, the operations are computed element wise. The output of a hidden layer is calculated by passing the weighted sum and an additive bias to the activation function. With weights ω^i_{jk} of the ith hidden layer where j and k again denote the jth element of the input vector and kth neuron of the hidden layer, the output $\mathbf{h}^i$ of this layer is given by

$$\mathbf{h}^i = \tanh(\sum_j h^{i-1}_j \omega^i_{jk} + b^i_k) = \tanh(\mathbf{h}^{i-1} \cdot \mathbf{W}^i + \mathbf{b}^i)$$

where $\mathbf{W}^i$ and $\mathbf{b}^i$ is the weight matrix and bias vector of the ith hidden layer, respectively, and $\mathbf{h}^0$ denotes the output vector of the projection layer.

4.3 Output Layer

The output layer processes the output from the last hidden layer to compute the desired probabilities. It is created with a number of neurons equal to the size of the vocabulary or short-list and, like the other layers, a number of weights propotional to the dimensionality of the input vector. In contrast to standard language modeling where estimation of the posterior probability $P(w_k = i|w^{k-1}_{k-n+1})$ of a single word with index i is made, the neural network simultaneously predicts the language model probability of all words in the vocabulary list for the given n-gram context w^{k-1}_{k-n+1} that was fed forward through the network.

Similar to neurons of hidden layers, a bias is added to the weighted sum computation as follows

$$\mathbf{v} = \sum_j h^*_j \omega^o_{jk} + b^o_k = \mathbf{h}^* \cdot \mathbf{W}^o + \mathbf{b}^o$$

where $\mathbf{h}^*$ denotes the output of the last hidden layer and $\mathbf{W}^o$ and $\mathbf{b}^o$ the weight matrix and bias vector of the output layer. As activation function, the output layer utilizes a softmax normalization defined as

$$\mathbf{o} = \frac{e^{\mathbf{v}}}{\sum_{l=1}^{N} e^{v_l}}$$

where the exponential function of vector $\mathbf{v}$ and the division are computed element wise. The result vector $\mathbf{o}$ consists of one output of each output layer neuron and thus has a dimension of the vocabulary or short-list size. The softmax normalization function has an output space in the continuous interval $[0, 1]$ and can be seen as

a smoothed version of the winner-takes-all activation model described in [Brid90] where the unit with the largest input has output 1 while all other units have output 0. Using the softmax as an activation function allows a probabilistic interpretation of the neural network outputs since it is able to convert a raw value into a probability measure. Hence the output vector $\mathbf{o}$ contains the posterior probabilities computed by the network where value o_i of the ith output neuron corresponds directly to the conditional probability $P(w_k = i|w_{k-n+1}^{k-1})$ of the ith word in the vocabulary.

4.4 Cost Function

The cost function that has to be minimized during training consists of two components. For the first part we use the cross-entropy between the output and the target probability distribution defined as

$$H = -\sum_{i=1}^{N} t_i \log o_i$$

where t_i refers to the target probability which is simply set to 1.0 for the desired output of the neural network, e.g. the next word in the training sentence, and 0.0 for all the other ones. The second part is a regularization term that aims at preventing the neural network from overfitting the training data which is also called weight decay. It is calculated by summing up the squared weights of each layer of the network. The regularization term of layer l with weights ω_{jk} is computed by

$$R_l = \sum_{jk} \omega_{jk}^2 \ .$$

Let $R_1, ..., R_L$ denote regularization terms of the projection layer, the hidden layers and the output layer, the cost function in form of an error function is defined by the expression

$$E = H + \beta \sum_{l=1}^{L} R_l$$

where β is a normalization factor that is tuned experimentally to achieve optimal results during training.

4.5 Backpropagation

Training is performed with the standard backpropagation algorithm minimizing the error function described in the previous section. The backpropagation algorithm was firstly formulated by Werbos in [Werb74] and a detailed description can be found in [RuHW86]. Backpropagation is a supervised learning technique common for training multilayer neural networks. It employs gradient descent which is an optimization algorithm and requires the activation function used by the neurons to be differentiable. The backpropagation algorithm can be divided into two main steps which are propagation and weight update.

Before training, the weights and biases of each layer are initialized randomly. The algorithm starts with a forward propagation of the sample input through the neural network in order to generate a network output. The output is then compared to

the desired output given by the training sample and an error is computed using the cost function. Subsequently, the error is propagated from the output layer back to the projection layer. Therefore the contribution of each weight and bias to the overall error is computed using the gradient of the error function with respect to the parameter. For ω, a weight or bias of any layer of the neural network, the gradient is given by the partial derivative of the error function as follows

$$\Delta\omega = \frac{\partial E}{\partial \omega} \ .$$

The last step is to update the weights and biases depending on their contribution to the error. The updated value for ω is computed by

$$\omega^{\text{new}} = \omega^{\text{old}} - \gamma\Delta\omega$$

where γ is a system parameter called the learning rate. It influences the speed and quality of learning where a higher learning rate corresponds with a faster training and a lower learning rate with a more accurate training. Learning rate is one of the parameters that have to be chosen experimentally to receive best training results. The used optimization procedure is called gradient descent since it finds a local minimum by taking steps proportional to the negative gradient of the function.

During training, the backpropagation algorithm is repeated for several epochs until the network converges to a local minimum. It has been shown, for example in [Bish95], that the outputs of a neural network trained in this manner converge to the desired posterior probabilities. Since the gradient is backpropagated through the projection layer, the neural network not only learns the posterior probabilities from the training examples but also the projection of the words onto the continuous space that is best for the probability estimation task.

5. Implementation

The multilayer perceptron is implemented using the architecture and learning algorithm described in the previous section with some optimizations being carried out in order to reduce long training time. Additionally, there is need to preprocess the training data since they are usually given as sentences of arbitrary length where the neural network only accepts word sequences of a fixed length as input. For example, to train a trigram neural network language model, the training data has to be provided as tuples consisting of a two words input and a one word output. Another important task is to adapt the neural network model into a simple form after the training phase. Given its nature of only storing weights instead of probabilities, requests have to be propagated through the network to compute the desired posterior probability. This can be very time consuming and difficult for system integration. Hence an access by simple table look-up like n-gram language models would be much more favorable. This chapter introduces some helpful tools and how they are used to implement the multilayer perceptron and other required functionalities. Furthermore, a brief overview to the Janus Recognition Toolkit is provided which is used to evaluate the adapted neural network language model on speech recognition tasks.

5.1 Training Optimization

With increasing complexity of the network architecture and a large training corpus, the time necessary for training the neural network grows as well. To reduce training time, several improvements to the standard implementation of the backpropagation algorithm have been presented in works such as [BDVJ03], [BeSé03] and [MoBe05]. A technique used in this work is to improve training speed by propagating several examples in form of a mini batch at once through the network. This was proposed in [BACD97]. The input and output vectors of each layer become matrices and hence the equations of each layer have to be changed from a vector-matrix multiplication to a matrix-matrix multiplication, for example for the projection layer described in section 4.1 yields

$$\Sigma(\mathbf{X}, \mathbf{P}) = \mathbf{X} \cdot \mathbf{P}$$

where matrix $\mathbf{X}$ consists of one input word vector per row. For a history of length two or more, the word vectors are ordered consecutively in the input matrix and

the multiplication result can be easily reshaped to perform the concatenation of projected vectors. Bias matrices used for the hidden layers and output layer are obtained by duplicating the bias vectors for each row of the matrix. Also a slight modification to the cost function of section 4.4 has to be made. The cross-entropy term for several examples is defined as the sum of the cross-entropy of each single example x, but to prevent the cost function from being depending on the batch size, the mean

$$H_{batch} = \frac{1}{B} \sum_{x} H$$

is used instead of the sum where B is the size of a mini batch. An evaluation of how different batch sizes affect the training time can be found in section 6.2.2.

5.2 Theano

Theano is a Python library that was developed to simplify the implementation of machine learning algorithms and to optimize their execution for performance speed-up. It is an open source project primarily developed by a machine learning group at the University of Montreal since 2008 [1]. In Python, array data types like vectors and matrices are commonly defined using the extension module NumPy which also provides many functions that are capable of operating on entire arrays at once. However, as stated in [Alte10] the composition of many such NumPy functions can be unnecessarily slow when each call is dominated by the cost of transferring memory rather than the cost of performing calculations. To reduce this overhead, optimizations are carried out by Theano before performing the calculations. Theano works on symbolically defined mathematical expressions while using similar convenient syntax as NumPy. It builds an internal graph using symbolic variables and operators to create a flexible represention of mathematical relations. The computational graph allows Theano to perform local graph transformations that can correct many unnecessary, slow or numerically unstable expression patterns.

Once an optimized graph is created by Theano, the same graph can be used to generate CPU as well as GPU implementations without requiring any changes to the actual program code. This is another huge advantage of using Theano. While common machine learning algorithms implemented with Theano are already superior to alternative implementations in terms of speed, an enormous speed-up can be further achieved by running Theano code on GPUs. The highly parallel structure of GPUs makes them more effective than general-purpose CPUs for algorithms where large blocks of data are processed. To run Theano code on GPUs, the only requirement is to have a CUDA enabled GPU where CUDA is a parallel computing platform created by NVIDIA. An experimental comparison of training speed on CPU and GPU can be found in section 6.2.1.

Like NumPy, Theano provides many functions for indexing and reshaping array data structures and all necessary mathematical operations such as the sum function and the logarithm. Furthermore, it predefines operations which are particular to neural networks and deep learning, for example the activation functions used for the multilayer perceptron, tanh and softmax. The calculation of the gradient which is crucial to our backpropagation algorithm is handled elegantly by Theano using

[1] https://github.com/Theano/Theano/

a symbolic differentiation function. Having the graph structure, computing the gradient is simply done by traversing the graph through all nodes from the outputs back to the inputs. For each node the gradient of its outputs with respect to its inputs can be calculated according to the operator used. Applying the chain rule of differentiation these gradients are composed in order to obtain the gradient of the output of the whole graph with respect to the inputs. Figure 5.1 displays a simple example of how differentiation can be easily computed with Theano.

```
1 import theano.tensor as T
2 from theano import function
3 x = T.dscalar()
4 y = x ** 2
5 dy = T.grad(y, x)
6 f = function([x], dy)
```

Figure 5.1: A Theano code example.

In Theano, variables are defined symbolically by assigning a type as shown in line 3 where `dscalar` denotes a 0-dimensional array of doubles. Line 4 defines y, another variable which represents the expression x^2 and also the next line defines a new variable dy for the expression $\partial y/\partial x$. In the last line, we create a function that takes x as input and gives dy as output. The function f can then be called to calculate the derivation result, for example calling $f(5)$ will directly yield 10. The subpackage `theano.tensor` contains all the important mathematical expressions required for machine learning algorithms and with the help of these provided functions, even complex mathematical algorithms like the multilayer perceptron algorithm can be developed into code that is easy to read and understand.

5.3 SRI Language Modeling Toolkit

SRI Language Modeling Toolkit (SRILM) is a toolkit for building and applying n-gram-based or related language models, and is used in a great variety of statistical modeling applications such as speech recognition, machine translation, statistical tagging and segmentation. It has been under development in the SRI Speech Technology and Research Laboratory since 1995 and is freely available for noncommercial purposes [2]. SRILM consists of a set of C++ class libraries implementing language models, executable programs built on top of these libraries to perform standard tasks such as training language models and testing them on data, and a collection of miscellaneous scripts for minor related tasks.

One of the main purposes of SRILM is to estimate language models from the given training data. This task is accomplished by the tool `ngram-count` which is capable of generating and manipulating n-gram counts of an arbitrary n-gram length. The resulting counts with the corresponding n-grams can be saved in a file or used for building an n-gram backoff language model. For the construction of more accurate backoff language models, one can choose from various implemented smoothing techniques like Good-Turing discounting and Kneser-Ney smoothing. In this work, `ngram-count` is used to create files constisting of n-grams of a fixed length out of the

[2] http://www.speech.sri.com/projects/srilm/

training data. The generated n-grams can then be easily splitted to fit the requirements of the neural network on the input samples. Before counting n-grams, the tokens <s> and </s>, necessary for indicating the beginning and end of a sentence, are automatically included to each sentence in the text corpus which is very convenient. There is also the option to assign a vocabulary file where out of vocabulary words are replaced with the unknown word token <unk>.

Since it is difficult to integrate our neural network language models into the speech recognition system for evaluation, the models are converted into the more practical ARPA format common to most language models trained with SRILM. The ARPA format is a look-up table consisting of log probability and n-gram pairs grouped into sections by n-gram length. Hence we need to generate n-grams, calculate their log probabilities using the neural network and write the pairs into a file like an ARPA backoff model. N-grams can be generated using the `lattice-tool` and lattice files created by the speech recognition system. Those lattice files contain all hypotheses made during a recognition task which results in a huge amount of n-gram combinations. With the `lattice-tool`, the n-grams can be extracted from the lattice files and written into a new file that can be directly processed by the neural network.

The `ngram` tool is another important SRILM tool that deals with tasks involving the evaluation of language models. It performs various operations including sentence scoring, perplexity computation and sentences generation. Here, perplexity is computed using a slightly different equation as introduced in section 2.2.2 and takes the number of OOVs and </s> tokens into account. The perplexity for sentences $s_1, ..., s_n$ is calculated by

$$ppl(P) = 10 \text{ ^ } (-\frac{1}{N - O + n} \sum_{i=1}^{n} \log_{10} P(s_i))$$

where N is the total number of words and O the number of OOVs. Aside from using the `ngram` tool to compute perplexities, it provides various types of model interpolation which is central to building the hybrid language models used for the speech recognition tasks. Hybrid language models result from a linear interpolation between the neural network language model in ARPA format and a backoff language model created using the `ngram-count` tool. The `ngram` tool performs interpolation of two or more language models with corresponding interpolation weights and automatically calculates the backoff weights of the new combined model. The new model is in ARPA format as well and can be directly used for speech recognition tasks with the Janus Recognition Toolkit.

5.4 Janus Recognition Toolkit

Janus Recognition Toolkit, also referred to as Janus, is a general-purpose speech recognition toolkit developed at the Interactive Systems Labs at Carnegie Mellon University and Karlsruhe Institute of Technology [3]. It is useful for both research and application development and is part of the JANUS speech-to-speech translation system. Researchers are able to build state-of-the-art speech recognizers using Janus and the toolkit allows them to develop, implement, and evaluate new methods.

[3] http://isl.anthropomatik.kit.edu/english/1406.php

Janus provides a flexible Tcl/Tk script based environment where Tcl is an open source scripting language and Tk a library for building graphical user interfaces. The Tcl/Tk programming environment is extended by object orientated addons which allow all components to be configured in a very flexible way without the need to modify source code.

Janus utilizes the concept of Hidden Markov Models for continuous speech recognition and offers many state-of-the-art techniques for acoustic preprocessing, acoustic model training, and speech decoding. Janus supports various frequent audio formats and performs the necessary tasks for acoustic preprocessing such as short-term fourier analysis and calculation of Mel-frequency scaled cepstral coefficients. Features for training acoustic models include Viterbi training, MMIE training and speaker adaptive training. For decoding tasks, it provides a flexible language model interface for n-gram language models and grammars, and functions for lattice generation, manipulation and rescoring.

6. Evaluation

This chapter presents a thorough evaluation of the neural network language models built using the multilayer perceptron. The network is trained on text corpora of three low resource languages which are Vietnamese, Tamil and Lao. Evaluation is carried out for different aspects such as training speed, performance on test corpus and achievements in speech recognition tasks. In terms of training speed, the influence of various system parameters on the time necessary for training the network are investigated and optimal values are searched for to ensure an acceptable training speed while producing a satisfactory training result. The performance of neural network language models of different sizes is evaluated on the test dataset and compared to a baseline language model of the same n-gram order. To further test the neural network models on practical tasks, they are integrated into a speech recognition system using Janus for each language. The following sections give a summary of the most important results that were achieved during the various experiments.

6.1 Training Data

The neural network language models for the languages Vietnamese, Tamil and Lao are trained using data released within the Babel program of the Intelligence Advanced Research Projects Activity (IARPA) [1]. It provides for each language a training corpus to build language models and a development corpus to test the generalization ability of the language models. To prevent overfitting issues during learning, a validation set has to be created out of the available corpora for the three languages. This can be done by splitting either the training data or the development data into two parts where only one of those is used to train or test the network while the other is held back for cross-validation purposes. During training, each time every example from the training set has been seen, the network tests its generalization ability on the validation set. Depending on the result, it decides whether to continue training or to stop before the network begins to memorize the examples.

For an extensive analysis of different network parameters, especially large layer sizes, a suitable amount of training data is required. Since Vietnamese is the only language with enough data out of the three, development tests on various parameters

[1] http://www.iarpa.gov/Programs/ia/Babel/solicitation_babel.html

are mostly carried out on the Vietnamese data. The IARPA training corpus for Vietnamese contains 944 k words and the development set has a length of 112 k words. With the training corpus consisting of 88 k lines of text, 10 k lines are randomly selected to form the validation set which results roughly in a training set of 830 k words and a validation set of 114 k words. The vocabulary has a length of 7 k words and no short-list is used for the network output layer.

While the Vietnamese corpus is already small, for Tamil which is a language spoken in South India and North-east Sri Lanka, even less data is available. With a vocabulary of over 16 k words and only 77 k words of training text, short-lists have to be used to achieve meaningful training results. Limiting the word list to words that at least occur twice in the training corpus, the size of the word list reduces to 5.5 k. But also larger word lists are considered which are constructed by appending the 5.5 k list randomly with words that only occur once in the training corpus. The development corpus consists of roughly 59 k words and is splitted in two halves to create the valiation set.

Neural network language models are also trained for Lao which is spoken in the Southeast Asian country Laos. Since Lao is a rare language like Tamil, training has to be performed on extremely sparse data as well. The available training corpus consists of 92 k words and the development corpus is about the same size. Also in this case, the development corpus is divided for cross-validation instead of using the training data for this purpose. With a small vocabulary of 3.2 k words, models are trained with the full vocabulary.

6.2 Training Time

Training a neural network for language modeling can be very time consuming. For an effective and fast system development, training speed is thus a factor that has to be taken into consideration while building such models. This is especially important for training very large neural networks which would be impractical using standard implementations and badly chosen parameters. Hence experiments on different system parameters were carried out to find optimal values that can reduce training time without a significant loss of performance. The investigated system parameters included a comparison between CPU and GPU, the size of the mini batches and the learning rate of the backpropagation algorithm. The experiments in this section were all carried out with a trigram neural network model on the Vietnamese training dataset. The network contained one hidden layer with 200 neurons and the input vectors were projected onto a 100-dimensional continuous space which resulted in a projection layer of size 200 as well.

6.2.1 CPU and GPU

Implementing the multilayer perceptron in Theano allows it to run on both CPUs and GPUs. It is to expect that the GPU has a less processing time compared to the CPU due to its highly parallel structure and efficiency. The experiments were carried out on an Intel Pentium Xeon X5670 CPU at 2.93GHz and an NVIDIA Tesla M2075 GPU. A comparison of the processing time for various mathematical expressions can be found in Table 6.1 where functions such as exp, log and tanh were

Operation	CPU	GPU
exp($\mathbf{M}$)	4.263 s	0.247 s
log($\mathbf{M}$)	1.721 s	0.235 s
tanh($\mathbf{M}$)	2.881 s	0.242 s
sigmoid($\mathbf{M}$)	2.610 s	0.235 s
softmax($\mathbf{M}$)	4.779 s	0.243 s
$\mathbf{M} \cdot \mathbf{N}$	538.637 s	3.336 s

Table 6.1: Processing time of mathematical expressions on CPU and GPU.

calculated element wise. The matrices $\mathbf{M}$ and $\mathbf{N}$ both had a dimension of $D \times D$ with $D = 10000$ and were randomly generated from the interval $[0, 1]$.

As shown in the table above, a speed-up of at least factor 10 could be achieved in almost every case using the GPU. Some operations even performed 20 times faster on the GPU than on the CPU. Furthermore, the CPU seemed to be very inefficient regarding the dot product of two huge matrices where the GPU could beat it by a factor of 160. To evaluate the impact of these improvements on the training speed, experiments on the time required for training a multilayer perceptron were evaluated on both devices as well. Table 6.2 shows the results for different training lengths which are one single example, a batch of examples, a training epoch and a complete training period. In all cases, the learning rate was set to 0.1 and the size of the mini batches to 100.

Training length	CPU	GPU
example	0.012 s	0.014 s
batch	0.281 s	0.021 s
epoch	43 min 50 s	3 min 22 s
complete	20 h 42 min	1 h 32 min

Table 6.2: Processing time of different training lengths on CPU and GPU.

While the CPU was faster than the GPU when only one example was processed, it was much slower when batches of examples were propagated through the network. With the GPU, a speed-up factor of 13.5 for a complete training period could be achieved and due to this remarkable increase in speed, all following experiments were performed on the GPU. Also note that a batch of 100 examples required only a little bit more time than a single example on the GPU. This is further evaluated in the next section.

6.2.2 Batch Size

Training on mini batches instead of single examples has shown in previous experiments that additional speed-up can be obtained. To investigate the relation between the size of mini batches and the required training time, experiments on various batch sizes were carried out. The results can be found in Figure 6.1 where the learning rate was set to 0.1 like in the previous section.

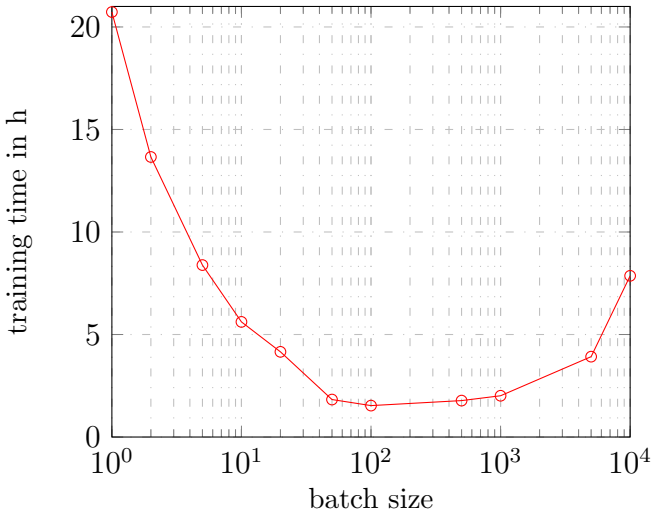

Figure 6.1: Training time using various batch sizes.

As expected, the long training time of over 20 hours using mini batches consisting of only one example could be significantly reduced by increasing the batch size. But it seemed that the batch size could not be arbitrarily augmented to decrease training time since a size of 1000 or more showed a slowdown instead of a speed-up compared to smaller batches. This is probably due to the highly complex matrix multiplications resulting from large batches which end up reducing the efficiency of this optimization approach. A mimimal training time of 1 hour and 32 minutes was obtained using a batch size of 100 which made an overall training time reduction by about a factor of 14. Hence for further experiments, 100 was used by default for the size of mini batches.

6.2.3 Learning Rate

Learning rate is a parameter that does not only affect the training speed but also the performance of the trained neural network. A higher learning rate corresponds with a shorter training since the algorithm descends faster towards the mimimum of the cost function. But it is possible that those wide steps made during gradient descent miss the minimum instead of finding it. A lower learning rate enables a more careful descent to find the minimum. However, this more accurate training can be extremely slow. In the experiments involing the learning rate, it was tried to find an optimal value for this parameter that leads to good performance while having an acceptable training time. Figure 6.2 shows the results on different values for the learning rate with respect to training time.

It can be seen that the training time decreased almost steadily by increasing the learning rate with training durations ranging from 5 hours and 20 minutes at a learning rate of 0.005 to a little bit more than 10 minutes using a learning rate of 5. The corresponding perplexities on the test corpus can be found in Figure 6.3.

Instead of a curve that goes up with increasing learning rates, it turned out to have a more parabolic form. While the ascent of the curve for large learning rates is unsurprising, training with smaller learning rates should have shown a much better performance. The experiments demonstrated that tiny values for the learning rate are also inappropriate. This is probably because that when the descending speed is

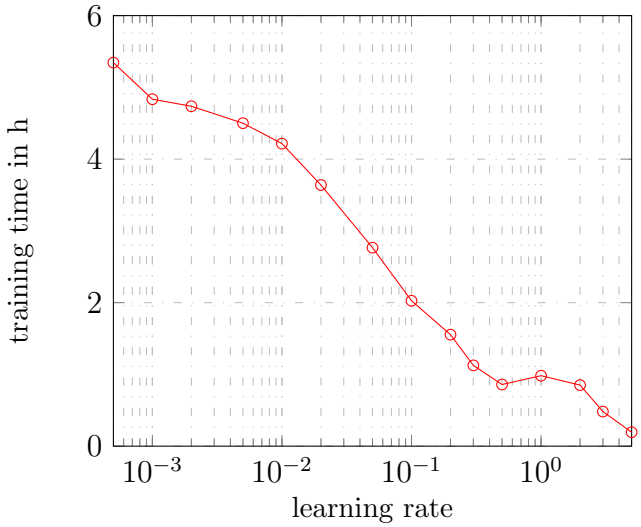

Figure 6.2: Training time with respect to different learning rates.

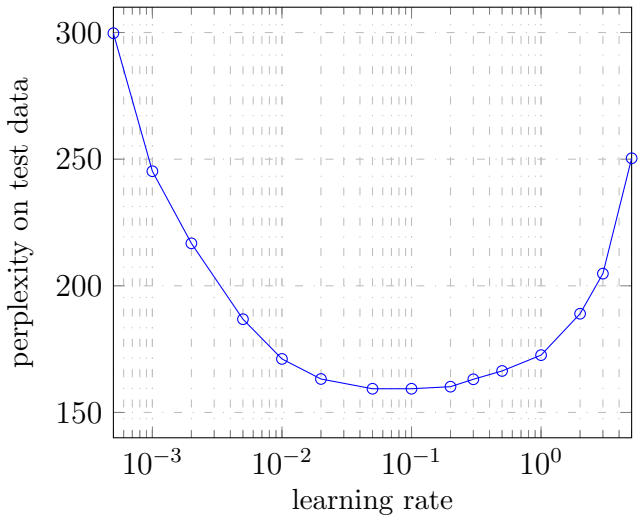

Figure 6.3: Perplexity on test data with respect to different learning rates.

too slow, the gradient of the cost function does not change very much between each backpropagation steps. It can happen that the network believes to have found the minimum and finishes training while the actual minimum is still far away.

Since the experimental results on the performance of various learning rates suggested that a minimum could be possibly found, it simplified the choice of an optimal parameter value. A close up view for learning rates between 0.01 and 1 is shown in Figure 6.4. Good performance could be achieved by choosing a value between 0.05 and 0.2 where a learning rate of 0.1 performed best in most of the cases.

6.3 Performance on Test Data

This section presents the performance of the neural network language models which was evaluated through the perplexity on the development dataset with respect to the necessary training time. The influence of the projection dimension was thoroughly investigated and optimal values were found for both bigram and trigram models.

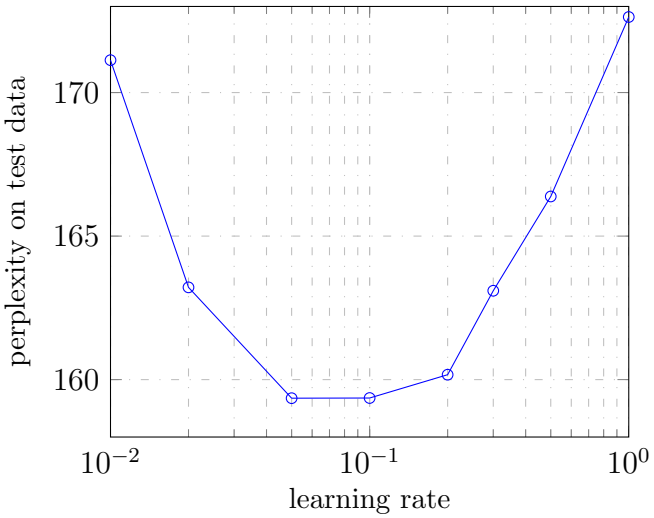

Figure 6.4: Perplexity on test data for learning rates between 0.01 and 1.

Since Vietnamese was the only language with enough data to train large networks, experiments on networks with different hidden layer sizes and numbers of hidden layers, especially large ones, were mainly carried out on the Vietnamese corpora. For Tamil and Lao where too little data was available, only smaller networks with one hidden layer were able to be effectively trained.

6.3.1 Dimension of Continuous Space

The task of the projection layer is to convert input word vectors into a continuous space representation. The impact of the dimension of this continuous space is investigated by training networks with different projection matrix sizes. Experiments were carried out on bigram and trigram networks with one hidden layer of size 200. The results using the Vietnamese training corpus can be found in Figure 6.5.

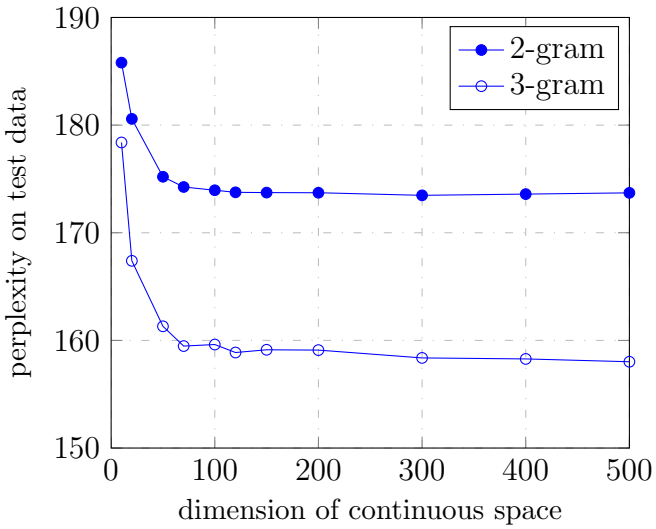

Figure 6.5: Perplexity of bigram and trigram neural network models with various projection sizes for Vietnamese.

To achieve good performance, the dimension of the continuous space should not be chosen to be very small. A significant improvement of about 10% relative in the perplexity could be achieved by increasing the dimension from 10 to 100 for both model orders. But the experiments displayed that a further increase would not provide such a big performance gain and even a dimension of 500 could not make any substantial perplexity reduction.

The required time for training each model is shown in Figure 6.6. Surprisingly, an increase in dimension did not necessarily lead to a longer training period. The training time for bigram networks even decreased until a projection layer size of 200 was reached. Trigram networks were shown to require significantly more training time with a 300 or more dimensional projection space which corresponds with a projection layer of 600 or more neurons. In consideration of both the performance and training time, a continuous space dimension of 500 for bigram models and 300 for trigram models was chosen for most of the networks for Vietnamese.

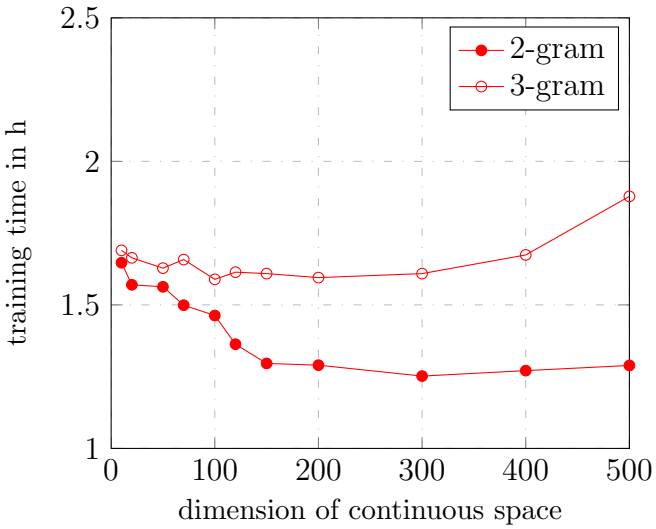

Figure 6.6: Training time of bigram and trigram neural network models with various projection sizes for Vietnamese.

6.3.2 Hidden Layer Size

The complexity of a neural network is given by its inner structure consisting of hidden neurons. To increase the computational capacity of a neural network, it requires more neurons in a hidden layer or additional hidden layers. Figure 6.7 displays how the performance of neural networks with one hidden layer could be improved by adding extra hidden neurons. As before, the neural networks were trained using the Vietnamese training set where the bigram and trigram model utilized a continuous dimension of 500 and 300, respectively.

By increasing the hidden layer size gradually from 100 to 3000 neurons, the perplexity of both models could almost be consistently reduced. While the perplexity of the bigram model decreased from 176.37 to 166.27 by about 6% relative, the trigram model received a performance gain of about 6.5% relative from 163.63 to 153.04. Since both curves show signs of convergence, further addition of neurons to the hidden layer probably would not be able to provide perplexity improvements that are

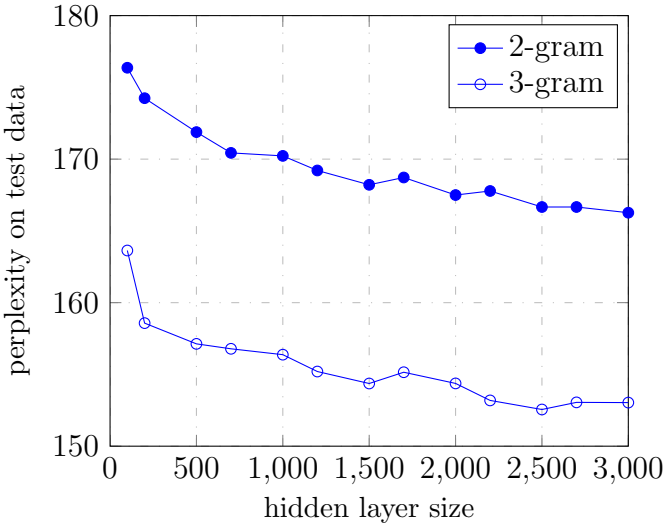

Figure 6.7: Perplexity of bigram and trigram neural networks with various hidden layer sizes for Vietnamese.

worth the costly training time. The required time to train three layered bigram and trigram models for hidden layer sizes up to 3000 is given in Figure 6.8. For sizes up to 1500, the required training time stayed roughly in the same magnitude for both models, but increased rapidly afterwards. Hence this makes training of very large hidden layers less efficient considering their gain in performance.

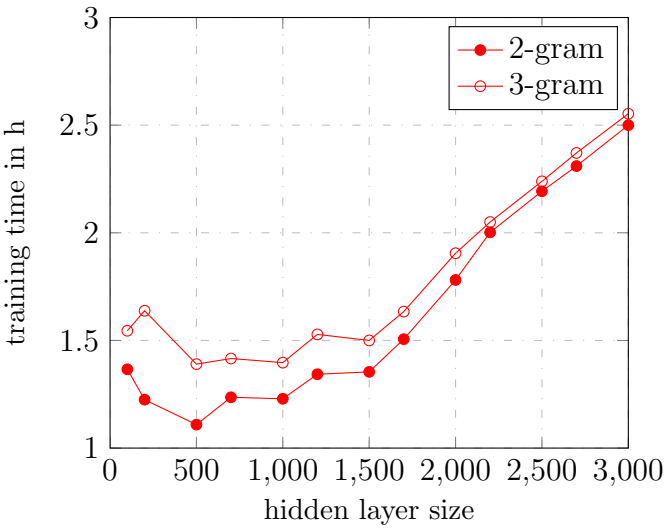

Figure 6.8: Training time of bigram and trigram neural networks with various hidden layer sizes for Vietnamese.

Adding a second hidden layer of the same size could significantly improve the performance of models with only one hidden layer. As shown in Figure 6.9, the perplexity decreased by about 10 for all bigram and trigram models with hidden layer sizes up to 2000. But also in this case, the perplexity converges with increasing number of hidden neurons so that a further substantial improvement by using two hidden layers with more than 2000 neurons is unlikely.

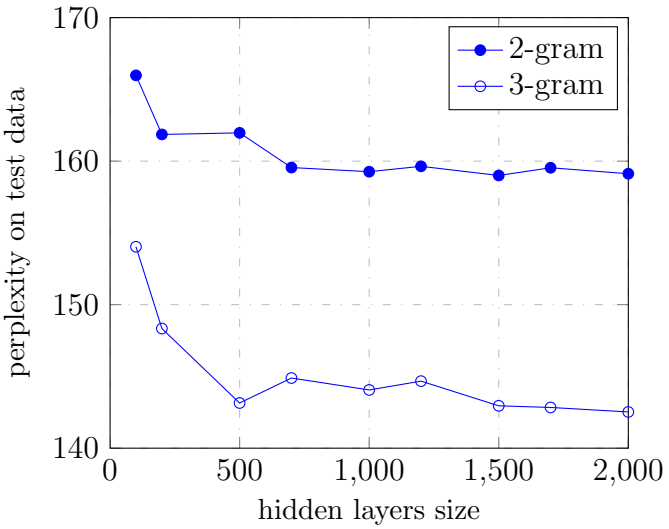

Figure 6.9: Perplexity of bigram and trigram neural networks with two hidden layers of the same size for Vietnamese.

The best performance using a second hidden layer was achieved by the models with two hidden layers of 2000 neurons which obtained perplexities of 159.12 for the bigram model and 142.52 for the trigram model. With additional complexity, the required training time increased as well. While a bigram or trigram model with one hidden layer could be trained in 2 h 30 min, it took the best bigram and trigram networks with two hidden layers 3 h 12 min and 3 h 30 min for training. An overview of training times for models with two hidden layer of various sizes can be taken from Figure 6.10.

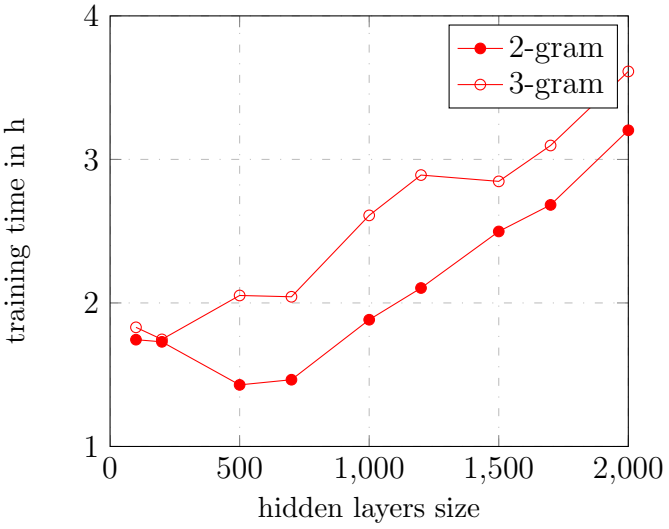

Figure 6.10: Training time of bigram and trigram neural networks with two hidden layers of the same size for Vietnamese.

Unfortunately, the significant improvements by adding a second hidden layer could not be continued with more extra hidden layers. Performance results of experiments with different numbers of hidden layers are displayed in Figure 6.11 where all hidden

layers consisted of 500 neurons. The according training times can be found in Figure 6.12. While adding a third hidden layer could only slightly improve perplexity, networks with four or more hidden layers even experienced a loss in performance compared to smaller networks. This happened probably because the lack of data caused the network to overfit when it tried to learn too many parameters from insufficient amount of examples. Hence, the network complexity including the number and size of hidden layers should not be arbitrarily increased but always adapted to the available training data for an optimal training result. Since three hidden layers with more than 500 neurons brought a decrease of performance as well, the overall best experimental performances on Vietnamese were 158.54 for bigram models and 140.95 for trigram models which were achieved by networks with three hidden layers of 500 neurons each.

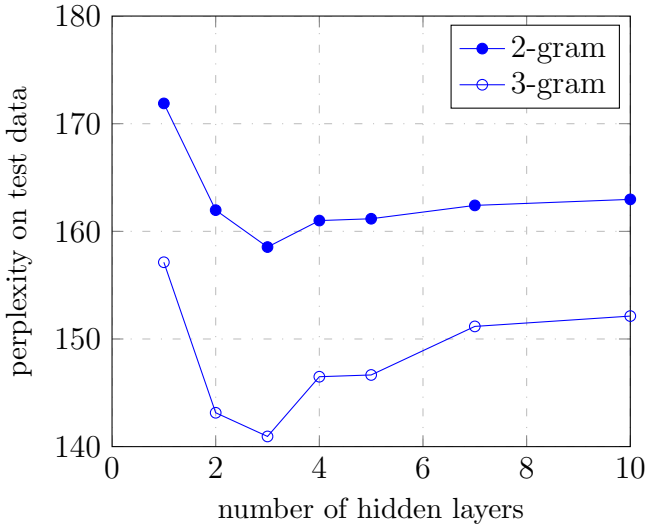

Figure 6.11: Perplexity of bigram and trigram neural networks with different numbers of hidden layers for Vietnamese.

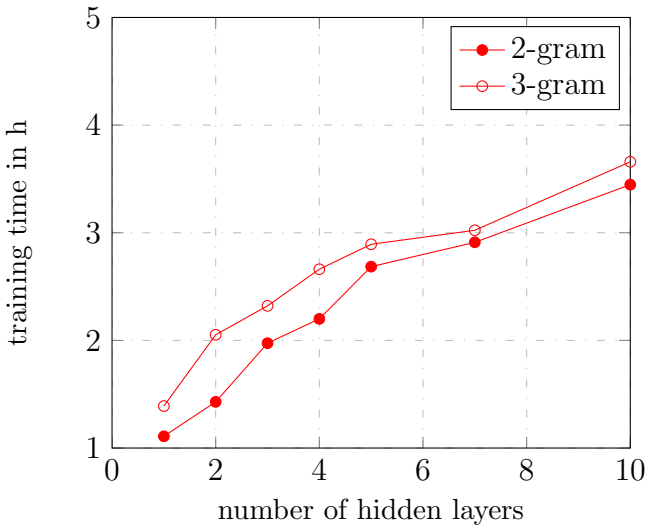

Figure 6.12: Training time of bigram and trigram neural networks with different numbers of hidden layers for Vietnamese.

Neural network language models for Lao were trained with extremely sparse data and experiments proved that the complexity of the network should be adapted to the size of the training corpora. During the experiments, satisfactory performances could not be obtained by networks with many or large hidden layers but smaller networks with one hidden layer were capable to outperform the bigger ones. Projection dimensions of 100 to 200 and hidden layer sizes of 100 to 300 were able to provide good results in most cases. Like Vietnamese, language models for Lao were trained on the full vocabulary. The best performances of bigram and trigram models along with the corresponding network parameters and training times can be found in Table 6.3.

Model	Projection size	Hidden size	Training time	Perplexity
2-gram	200	200	15 min	153.18
3-gram	100	200	14 min	148.09

Table 6.3: Performance of bigram and trigram models for Lao.

Since the lack of training data also applies to Tamil, similar network parameters were found to score less perplexities. For Tamil with a huge vocabulary of over 16 k words with respect to the available amount of data, no satisfying neural networks could be trained using the full vocabulary. Hence, short-lists were applied where list sizes of 5.5 k and 11 k words were experimented. The results of the best models for both short-list sizes can be found in Table 6.4.

Model	Projection size	Hidden size	Training time	Perplexity
2-gram 5.5k	200	200	12 min	458.04
3-gram 5.5k	100	200	13 min	453.11
2-gram 11k	500	200	25 min	578.85
3-gram 11k	200	200	20 min	554.62

Table 6.4: Performance of bigram and trigram models for Tamil.

6.4 Speech Recognition Tasks

The neural network language models for all three languages were further evaluated on speech recognition tasks and compared to 3-gram backoff language models smoothed with Kneser-Ney. For better system integration, the new models were converted to look-up probability tables by rescoring lattices generated by the speech recognition system. The converted models were then brought to ARPA format by backing off a trigram probability table with a bigram and unigram table where the unigram table was taken from the baseline model. Furthermore, hybrid models built by interpolating neural network models with baseline models were also evaluated.

The comparison results between baseline, neural network and hybrid models for Vietnamese are shown in Table 6.5 where the second column denotes that the best bigram and trigram neural networks with the according number of hidden layers were used for the neural network model in ARPA backoff format.

Model	# Hidden layers	Perplexity	WER
Baseline	-	160.43	50.1
Neural network	1	164.79	50.8
Hybrid	1	147.95	49.6
Neural network	2	159.83	50.2
Hybrid	2	144.04	49.5
Neural network	3	158.48	50.1
Hybrid	3	143.10	49.5

Table 6.5: Comparison of baseline, neural network and hybrid models on Vietnamese speech recognition tasks.

Although the perplexity and WER could not be significantly decreased using the neural network language models alone, the interpolated models were able to achieve great improvements compared to the backoff baseline model. The best model with one hidden layer was interpolated with a weight of 0.4 and reduced the perplexity by nearly 8% relative and the WER by 0.5% absolute. The networks with two and three hidden layers were both interpolated with a weight of 0.6 and improved the perplexity by about 10% relative and the WER by 0.6% absolute.

As shown in Table 6.6, substantial improvements could also be achieved for Lao despite the litte training data that was available and the much smaller networks with a single hidden layer that could be trained for this reason. While the baseline perplexity could be reduced by 9% relative with the neural network model alone, the WER of the new model was the same as the baseline WER. But using the interpolated model with a interpolation weight of 0.6 for the neural network model, the perplexity could be further decreased by 5% relative and an overall WER reduction of 0.8% absolute was obtained.

Model	Perplexity	WER
Baseline	139.75	63.7
Neural network	127.38	63.7
Hybrid	121.14	62.9

Table 6.6: Comparison of baseline, neural network and hybrid models on Lao speech recognition tasks.

Finally, comparison results for Tamil with respect to the perplexity on the development data can be found in Table 6.7 where 5.5 k and 11 k denote the size of the short-lists used by the output layers. It can be seen that the baseline perplexity could be significantly reduced in all cases by improvements of about 23% to 25% relative. Due to the late point of time that the speech recognition system for Tamil was available for evaluation, only an initial test with one hybrid model could be conducted within the time frame of this work. Since the system was still in development and the acoustic components were not yet the best ones, the baseline n-gram model resulted in a WER of 91.1%. Using the same system, the baseline WER

could be outperformed by 0.3% absolute with the hybrid language model using a 5.5 k wordlist and an interpolation weight of 0.5. It can be expected from further experiments, that different interpolation weights might lead to another gain in performance and that the models using a 11 k wordlist might probably outperform the 5.5 k models. Also higher improvements can be possibly achieved when integrated to a better developed system with more accurate acoustic models where the neural network language models can be utilized more effectively.

Model	Perplexity
Baseline	575.58
Neural network 5.5k	441.13
Hybrid 5.5k	436.28
Neural network 11k	437.77
Hybrid 11k	432.84

Table 6.7: Comparison of baseline, neural network and hybrid models on Tamil with respect to the perplexity on test data.

7. Conclusion and Further Work

This work described how language modeling can be performed in a continuous space. Neural networks in form of multilayer perceptrons were implemented and trained on the languages Vietnamese, Tamil and Lao where data sparseness is a severe problem for alternative methods like n-gram language modeling. The training of the networks was optimized using Theano implementations for GPU where a significant reduction of training time could be achieved. Experiments on different network parameters such as the batch size and the learning rate were carried out to investigate their influence on the training time and optimal values could be found to obtain most possible time reduction. These optimizations were important since all further experiments with neural networks of different sizes, especially large ones, could only be carried out with an acceptable training time.

The performance of various neural networks was thoroughly evaluated where different projection size, hidden layer size and number of hidden layers were used. Experiments have shown that the network complexity should be adapted to the size of the available training corpus. For Vietnamese, networks with three medium-sized hidden layers proved to perform best and for Tamil and Lao which had extremely little data, optimal results were obtained by smaller networks with a single hidden layer. In a comparison with state-of-the-art n-gram-based language models, consistent perplexity and WER reductions could be reported for all three languages with significant WER reductions up to 0.6% absolute for Vietnamese and 0.8% for Lao.

In this work, we proved how a multilayer perceptron can be effectively employed for language modeling, even for languages with extremely sparse data. To improve our multilayer perceptron for large vocabulary languages, the structured output layer described in section 3.2.2 could be used instead of short-lists which provides a possibility for future works on this area. Although the training time was significantly reduced by running the network on the GPU and utilizing mini batches, there are still many other potential optimization techniques such as the grouping and resampling of training data that need to be investigated in case of huge text corpora. The neural network language models in this work were evaluated on speech recognition tasks and it would be interesting to see the performance of those neural network language models integrated into other applications like machine translation as well.

Furthermore, there is the possibility that the speech recognition systems can be improved even more by not only applying neural networks to language modeling, but also to acoustic modeling.

Beside the multilayer perceptron, there are many other neural network architectures such as the recurrent networks described in the sections 3.2.3 and 3.2.4 which sound promising and can be tested on low resource languages as well. Finally, since the languages Vietnamese, Tamil and Lao are not the only low resource languages provided by the IARPA Babel program, experiments on various other languages with sparse copora such as Cantonese and Haitian Creole can be carried out in future works.

Bibliography

[Alte10] F. Alted. Why modern CPUs are starving and what can be done about it. *Computing in Science and Engineering*, **12**(2), 2010, pp. 68–71.

[BACD97] J. Bilmes, K. Asanovic, C. Chin and J. Demmel. Using phipac to speed error back-propagation learning. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, 1997, pp. 4153–4156.

[BBBL+10] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley and Y. Bengio. Theano: A CPU and GPU Math Compiler in Python. In *Proceedings of the 9th Python for Scientific Computing Conference*, June 2010.

[BDVJ03] Y. Bengio, R. Ducharme, P. Vincent and C. Jauvin. A Neural Probabilistic Language Model. *Journal of Machine Learning Research*, **3**(2), 2003, pp. 1137–1155.

[BeSé03] Y. Bengio and J.-S. Sénécal. Quick training of probabilistic neural nets by importance sampling. In *Proceedings of AISTATS Conference*, 2003.

[Bish95] C. Bishop. *Neural Networks for Pattern Recognition*. Clarendon Press, Oxford. 1995.

[BPSL+92] P. F. Brown, V. J. D. Pietra, P. V. de Souza, J. C. Lai and R. L. Mercer. Class-based n-gram models of natural language. *Computational Linguistics*, **18**, 1992, pp. 467–479.

[Brid90] J. S. Bridle. Probabilistic interpretation of feedforward classification network outputs, with relationship to statistical pattern recognition. In *Neurocomputing: Algorithms, Architectures and Applications*. Springer-Verlag, 1990, pp. 227–236.

[BrLo88] D. S. Broomhead and D. Lowe. Multivariable functional interpolation and adaptive networks. *Complex Systems*, **2**, 1988, pp. 321–355.

[ChGa91] K. W. Church and W. A. Gale. A comparison of the enhanced Good–Turing and deleted estimation methods for estimating probabilities of English bigrams. *Computer Speech and Language*, **5**, 1991, pp. 19–54.

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

[CiMS12] D. Ciresan, U. Meier and J. Schmidhuber. Multi-column deep neural networks for image classification. In *Proceedings of the 25th IEEE Conference on Computer Vision and Pattern Recognition*, 2012, pp. 3642–3649.

[Crus06] H. Cruse. *Neural Networks as Cybernetic Systems*. Brains, Minds and Media. 2nd and revised, 2006.

[Elma90] J. L. Elman. Finding Structure in Time. *Cognitive Science*, **14**(2), 1990, pp. 179–211.

[GaCh90] W. A. Gale and K. W. Church. Estimation procedures for language context: poor estimates are worse than none. In *COMPSTAT, Proceedings in Computational Statistics, Ninth Symposium*, Dubrovnik, Yugoslavia, September 1990, pp. 69–74.

[GaCh94] W. A. Gale and K. W. Church. What's wrong with adding one? In *Corpus-Based Research into Language*, Rodolpi, Amsterdam, 1994.

[Good53] I. J. Good. The population frequencies of species and the estimation of population parameters. *Biometrika*, **40**, 1953, pp. 237–264.

[HDYM+12] G. Hinton, L. Deng, D. Yu, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, G. Dahl and B. Kingsbury. Deep Neural Networks for Acoustic Modeling in Speech Recognition. *IEEE Signal Processing Magazine*, **29**(6), November 2012, pp. 82–97.

[HiAS85] G. E. Hinton, D. H. Ackley and T. J. Sejnowski. A Learning Algorithm for Boltzmann Machines. *Cognitive Science*, **9**(1), 1985, pp. 147–169.

[Hint02] G. E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Computing*, **14**(8), August 2002, pp. 1771–1800.

[Hint09] G. Hinton. Deep belief networks. *Scholarpedia*, **4**(5), 2009, pp. 5947.

[Hopf82] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. In *Proceedings of the National Academy of Sciences of the USA*, **79**, April 1982, pp. 2554–2558.

[HuZa10] H. Hu and S. A. Zahorian. Dimensionality Reduction Methods for HMM Phonetic Recognition. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, Dallas, TX, March 2010, pp. 4854–4857.

[JeMe80] F. Jelinek and R. L. Mercer. Interpolated estimation of Markov source parameters from sparse data. In *Proceedings of the Workshop on Pattern Recognition in Practice*, North-Holland, Amsterdam, The Netherlands, 1980, pp. 381–397.

[Katz87] S. M. 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**, 1987, pp. 400–401.

[KnNe95] R. Kneser and H. Ney. Improved backing-off for m-gram language modeling. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, **1**, Detroit, MI, May 1995, pp. 181–184.

[Koho82] T. Kohonen. Self-Organized Formation of Topologically Correct Feature Maps. *Biological Cybernetics*, **43**(1), 1982, pp. 59–69.

[KrSH12] A. Krizhevsky, I. Sutskever and G. E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*, pp. 1097–1105. Curran Associates, Inc., 2012.

[LBBH98] Y. LeCun, L. Bottou, Y. Bengio and P. Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, **86**, November 1998, pp. 2278–2324.

[Liao13] H. Liao. Speaker adaptation of context dependent deep neural networks. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, Vancouver, BC, May 2013, pp. 7947–7951.

[Lids20] G. J. Lidstone. Note on the general case of the Bayes–Laplace formula for inductive or a posteriori probabilities. *Transactions of the Faculty of Actuaries*, **8**, 1920, pp. 182–192.

[LIS13] LISA lab, University of Montreal. *Theano Documentation*, Release 0.6, December 2013.

[LLPN09] H. Lee, Y. Largman, P. Pham and A. Ng. Unsupervised Feature Learning for Audio Classification using Convolutional Deep Belief Networks. In *Advances in Neural Information Processing Systems 22*, 2009, pp. 1096–1104.

[LOAG+11] H. S. Le, I. Oparin, A. Allauzen, J. Gauvain and F. Yvon. Structured output layer neural network language model. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, Prague, May 2011, pp. 5524–5527.

[LRMD+12] Q. Le, M. Ranzato, R. Monga, M. Devin, K. Chen, G. Corrado, J. Dean and A. Ng. Building high-level features using large scale unsupervised learning. In *International Conference in Machine Learning*, 2012.

[McPi43] W. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, **5**, 1943, pp. 115–133.

[MKBC+10] T. Mikolov, M. Karafit, L. Burget, J. Cernocky and S. Khudanpur. Recurrent neural network based language model. In *Proceedings of the 11th Annual Conference of the International Speech Communication Association*, **9**, 2010.

[MLOV+12] A. Maas, Q. Le, T. O'Neil, O. Vinyals, P. Nguyen and A. Ng. Recurrent Neural Networks for Noise Reduction in Robust ASR. In *Proceedings of Interspeech*, Portland, OR, September 2012.

[MnHi08] A. Mnih and G. E. Hinton. A scalable hierarchical distributed language model. *Neural Information Processing Systems*, **21**, 2008, pp. 1081–1088.

[MoBe05] F. Morin and Y. Bengio. Hierarchical probabilistic neural network language model. In *Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics*, 2005.

[MoDH12] A. Mohamed, G. Dahl and G. Hinton. Acoustic Modeling Using Deep Belief Networks. *IEEE Transactions on Audio, Speech, and Language Processing*, **20**(1), January 2012, pp. 14–22.

[NeEs91] H. Ney and U. Essen. On smoothing techniques for bigram-based natural language modelling. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, **2**, 1991, pp. 825–829.

[NiWa12] J. Niehues and A. Waibel. Continuous Space Language Models using Restricted Boltzmann Machines. In *Proceedings of the International Workshop for Spoken Language Translation*, Hong Kong, December 2012.

[Rose57] F. Rosenblatt. The Perceptron–a perceiving and recognizing automaton. Report 85-460-1, Cornell Aeronautical Laboratory, 1957.

[RuHW86] D. E. Rumelhart, G. E. Hinton and R. J. Williams. Learning representations by back-propagating errors. *Nature*, **323**(6088), October 1986, pp. 533–536.

[ScGa02] H. Schwenk and J. Gauvain. Connectionist language modeling for large vocabulary continuous speech recognition. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, **1**, 2002, pp. 765–768.

[ScGa04] H. Schwenk and J. Gauvain. Neural network language models for conversational speech recognition. In *Proceedings of International Conference on Speech and Language Processing*, 2004, pp. 1215–1218.

[Schw07] H. Schwenk. Continuous space language models. *Computer Speech and Language*, **21**, 2007, pp. 492–518.

[Stol02] A. Stolcke. SRILM - An Extensible Language Modeling Toolkit. In *Proceedings of International Conference on Speech and Language Processing*, Denver, Colorado, September 2002, pp. 901–904.

[Werb74] P. J. Werbos. Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences. PhD thesis, Harvard University, 1974.

[WHHS+89] A. Waibel, T. Hanazawa, G. Hinton, K. Shikano and K. J. Lang. Phoneme recognition using time-delay neural networks. *IEEE Transactions on Acoustics, Speech and Signal Processing*, **37**(3), March 1989, pp. 328–339.

[WuCh93] J. Wu and C. Chan. Isolated Word Recognition by Neural Network Models with Cross-Correlation Coefficients for Speech Dynamics. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **15**(11), November 1993, pp. 1174–1185.