UNIVERSITEIT•STELLENBOSCH•UNIVERSITY
jou kennisvennoot • your knowledge partner

Neural networks for language model smoothing

Werner Van der Merwe
20076223

Report submitted in partial fulfilment of the requirements of the module Project (E) 448 for the degree Baccalaureus in Engineering in the Department of Electrical and Electronic Engineering at Stellenbosch University.

Study Leader: Prof. T. R. Niesler

November 2019

Acknowledgements

First and foremost, I would like to thank my parents for giving me the opportunity to complete my studies. To Prof. T.R. Niesler, thank you for the continued support throughout the busy semester. Lastly my close friends, without the light-hearted coffee breaks motivation would have been harder to come by.

Abstract

English

Current techniques used to smooth language models only utilise a fraction of the information available to them. This results in cases where n-grams, that can be distinguished further though the unused information, are given the same probability estimate in a language model. By developing a neural network capable of estimating a probability for a given n-gram from a larger set of inputs that the current techniques use, we show how n-grams are given probability estimates better suited to their context, without reducing the performance of the language model.

Afrikaans

Deesdae se maniere om taal modelle af te rond gebruik slegs n gedeelte van die informasie wat beskikbaar is. Dit veroorsaak dat n-gramme, waartussen daar verder onderskei kan word deur die onbenutte informasie, die selfde waarskynlikheid in die taal model besit. Deur gebruik te maak van ‘n neurale netwerk wat ‘n waarskynlikheid vir ‘n n-gram kaan bepaal met gebruik van meer toeveer data, wys ons hoe die n-gramme ‘n meer geskikte waarskynlikheid gegee word, sonder om die taal model se resultate te verswak.

Contents

List of Figures

List of Tables

Nomenclature

Variables and functions

$p(x)$	Probability density function with respect to variable x.
$P(A)$	Probability of event A occurring.
$N(A)$	Number of times event A occurred.
$E(x)$	Expectation of random variable x.
$C(w)$	Count of word w.

Acronyms and abbreviations

LM	Language Model
PP	Perplexity
NN	Neural Network
MLE	Maximum Likelihood Estimate
MSE	Mean Square Error
MAE	Mean Absolute Error
SGD	Stochastic Gradient Descent
SRILM	SRI Language Modeleling toolkit
GPU	Graphics Processing Unit

Chapter 1

Introduction

A language model is a statistical model used by machines to assess word sequences. It allows the machine to implicitly model the rules that make up a language, by observing and then probabilistically modelling the reoccurring patterns of words present in a training corpus. However, a language model can only model sequences it has already been exposed to in the training corpus. For a vocabulary size of 60 thousand words, there are 2.16×10^{14} possible sequences three words (trigrams) can produce. Therefore, the majority of trigrams are seen rarely during training and many remain unseen. A problem thus arises when the machine is tasked with assigning a probability to an unseen trigram. Rare trigrams are also subject to weak estimation, due to the small number of training examples.

To improve the robustness of a language model, specific steps are taken to allow it to effectively estimate probabilities for these unseen and rare events. *Smoothing* refers to various techniques that assign a non-zero probability to unseen events by adjusting the probability of seen events. This project investigates the use of a neural network to replace current mathematical approaches to smoothing a language model.

In this report, we start by analyzing the Good-Turing and Katz smoothing techniques and how they are implemented for an n-gram language model to improve its performance. This project aims to employ a neural network for smoothing and to achieve the same performance as the Good-Turing technique. The quality of a language model is expressed through its perplexity score on a test corpus.

Firstly, we establish a baseline language model for comparison with our proposed neural network smoothed language model. Thereafter we develop a neural network capable of predicting discounted probabilities for a trigram, trained on the Good-Turing estimates. We then expand on this neural network to directly calculate the estimates for a trigram through an alternative training approach. With the exception of the SRILM tools described in Section 3.2, all software and scripts used to achieve this were self-written.

Finally, the results are analyzed and discussed in some detail. As the topic of this work may be considered to be quite speculative, we do not elaborate on whether or not

the neural network is superior to existing smoothing techniques. Rather, we discuss how the neural network differs or conforms to existing smoothing techniques, the observed tendencies of the neural network, and the impact that the neural network has on the probability estimate of a given trigram.

Goals of this work

The work described in this report has the following two goals:

1. To determine whether a neural network can be used to smooth language model probability estimates as well as currently used methods.

2. To determine whether the neural network can address the over-estimation of probabilities of rare or unseen events that sometimes occurs using current methods.

Structure of the Report

The remainder of the report is structured as follows:

- Chapter 2 provides background to language modeling with current smoothing techniques, as well as how a neural network is capable of performing smoothing through regression.

- Chapter 3 describes how a language model is constructed and shows the performance of the baseline trigram model.

- Chapter 4 and 5 documents the development of both neural network approaches and presents the performance in terms of perplexity.

- Chapter 6 evaluates the results of the second neural network approach in greater detail, and discusses the tendencies the neural network exhibited in estimating the probability of a trigram.

Chapter 2

Literature Study

This chapter presents the background required to implement and test the techniques presented in the following chapters.

2.1. Statistical Language Models

Natural languages spoken by people are unlike programming languages, they are not designed, but rather emerge naturally and tend to change over time. Although they are based on a set of grammatical rules, spontaneous speech often deviates from it. Because of this, trying to program machines to interpret languages can be difficult.

Rather than trying to model formal grammars and structures of a language, we find more success in basing them on statistical models. This is far less complex and still allows machines to interpret certain ambiguities, occurring often under natural circumstances.

One such ambiguity can be explained by an example. If a machine's purpose was to translate a user's speech to text, it could encounter the following problem; having to determine which of two words, that share acoustic characteristics, were said by the user. Such as "what do you see?" vs "what do you sea?". For the machine to interpret this correctly, we make use of a statistical language model. These models succeed because words do not appear in random order i.e. their neighbours provide much-needed context to them.

A statistical language model, in short, simply assigns a probability to a sequence of words. By doing so, the machine is able to choose the correct sequence of words based on their probabilities. To be clear, the language model does not assign a probability based on acoustic data, but rather the possible word sequences themselves.

2.1.1. Mathematical Preliminaries

The probability of a sequence of words can be represented as follows:

$$P(\mathbf{w}(0, L-1)) \tag{2.1}$$

with $\mathbf{w}(0, L-1) = w(0), w(1), ..., w(L-1)$ representing a sequence of L words. This probability is used by the machine in the speech-to-text example to successfully distinguish between the ambiguous words and choose the correct sequence, much like a human uses the context of the given words.

The joint probability in Equation (2.1) can be decomposed, using the definition of conditional probabilities, into a product of conditional probabilities:

$$P(\mathbf{w}(0, L-1) = \prod_{i=0}^{L-1} P(w(i)|\mathbf{w}(0, i-1)) \tag{2.2}$$

Therefore the probability of observing the i^{th} word is dependent on knowing the preceding $i-1$ words. To reduce complexity, we approximate Equation (2.2) to only consider the preceding $n-1$ words:

$$\prod_{i=0}^{L-1} P(w(i)|\mathbf{w}(0, i-1)) \approx \prod_{i=0}^{L-1} P(w(i)|\mathbf{w}(i-n+1, i-1)) \tag{2.3}$$

We have now derived the basis for the n-gram language model using the **Markov** assumption. Markov models are described as probabilistic sequence models that predict some future unit without looking too far into the past [2].

2.1.2. N-Gram Language Models

In this report, we will only be focusing on one type of language model, the n-gram model. The n-gram model has several reasons for its success in becoming one of the preferred models. It considered as being computationally efficient and simple to implement [3].

The first step would be to determine the value of n. This is also referred to as the tuple size. Ideally, we would want a large tuple size, because words could still strongly influence others far down the sequence. However, the number of parameters that need to be estimated grows exponentially as tuple size increases. If we take a vocabulary size of 20,000 words, which is modest, Table 2.1 shows the number of parameters to be estimated for each tuple size [4].

Table 2.1: Increase of parameters for n-gram size

Tuple Size	Parameters
2 (bigram)	$20,000^2 = 4 \times 10^8$
3 (trigram)	$20,000^3 = 8 \times 10^{12}$
4 (four-gram)	$20,000^4 = 16 \times 10^{16}$
5 (five-gram)	$20,000^5 = 32 \times 10^{20}$

A five-gram model, although having more information available for potentially better estimation of the probabilities, is not computationally practical. Four-gram models are considered to be barely feasible. Practical n-gram models tend mostly to incorporate bigrams and trigrams [4].

2.1.3. Estimating N-gram Probabilities

The simplest way to estimate the probability is with a maximum likelihood estimation. To obtain this maximum likelihood estimate (MLE), we observe the n-gram counts from a training corpus. These counts are thereafter normalized to give [2]:

$$P_{rf}(w_n|w_1...w_{n-1}) = \frac{N(w_1...w_n)}{\sum_{\forall k} N(w_1...w_{n-1}, w_k)} \tag{2.4}$$

This equation can be further simplified to yield Equation (2.5) as the count of a given prefix is equal to the sum of all n-gram counts containing that same prefix.

$$P_{rf}(w_n|w_1...w_{n-1}) = \frac{N(w_1...w_n)}{N(w_1...w_{n-1})} \tag{2.5}$$

We now have a value that lies between 0 and 1, and will refer to this probability estimate as the relative frequency estimate.

A new problem arises because our training set and validation set do not necessarily contain all n-grams seen later on during testing. Even if we increase the size of the training set, the majority of words remain uncommon and n-grams containing these words are rare [4]. The validation set will most likely contain several n-grams not seen in the training set and whose relative frequencies according to Equation (2.5) are consequently zero.

The probability of occurrence for a sequence of words should never be zero. However, since many legitimate word sequences will never be seen in a practical training set, some form of regularization is needed to prevent overfitting of the training data. This is where

smoothing is applied. Smoothing decreases the probability of seen n-grams and assigns this newly acquired probability mass to the unseen n-grams [4]. The next two sections will discuss different kinds of smoothing and how they are implemented.

2.1.4. Good-Turing Smoothing

This smoothing technique was published by I. J. Good in 1953, who credits Alan Turing with the original idea [5]. It is based on the assumption that the frequency for each, in our case, n-gram follows a binomial distribution [2]. Specifically, n-grams occurring the same number of times are considered to have the same associated probability of occurance.

We begin with the Good-Turing probability estimate:

$$P_{GT} = \frac{r^*}{N_\sum} \tag{2.6}$$

In Equation (2.6), P_{GT} is the estimated probability of an n-gram that is seen to occur r times during training, $N_\sum$ is the total number of n-grams and r^* is a discounted count formulated as follows [4]:

$$r^* = (r+1)\frac{E(C_{r+1})}{E(C_r)} \tag{2.7}$$

In Equation (2.7) C_r refers to the count of n-grams that occur exactly r times in the training data, and $E(\cdot)$ refers to the expectation. By approximating the expectations by the counts and combining Equations (2.6) and (2.7) we obtain:

$$P_{GT} = q_r = \frac{(r+1)\cdot C_{r+1}}{N_\sum \cdot C_r} \tag{2.8}$$

In Equation (2.8), q_r denotes the Good-Turing estimate for the probability of occurrence of an n-gram occurring r times in the training data. For unseen n-grams, $r = 0$, giving us:

$$C_0 \cdot q_0 = \frac{C_1}{N_\sum} \tag{2.9}$$

and where $C_0 \cdot q_0$ can be interpreted as the probability of occurrence of any n-gram not seen in the training set.

The Good-Turing technique can be applied to a language model, in order to provide nonzero probability estimates for unseen n-grams. However, this technique has two concerns. One is that our assumption in Equation (2.8) is only viable for values of $r < k$ (where k is a threshold, typically a value ranging from 5 to 10 [2,4]). The MLE estimate of high frequency words is accurate enough that they do need smoothing. The other concern is that it is quite possible for C_r to equal 0 for some value of r. This would result in

a probability estimate of zero, leading us back to our initial problem. To remedy this, one can smooth the values for C_r, replacing any zeroes with positive estimates before calculating the probability.

2.1.5. Backing-Off

Instead of redistributing the probability mass equally amongst all unseen n-grams, S. M. Katz suggests an alternative. His solution, introduced in 1987 [2], assigns a nonzero probability estimate to unseen n-grams, by *backing off* to the $(n-1)$-gram. This backoff process continues until a n-gram with a nonzero count is observed. By doing so, the model is able to provide a better probability estimate for an unseen n-gram.

In Katz's backoff model, a discounting factor reserves the probability mass for unseen n-grams. This discounting factor could be implemented in the form of Good-Turing discounting, but also by absolute discounting, linear discounting or other suitable estimators. The backoff procedure dictates the redistribution of this probability mass amongst unseen n-grams [2].

We will be combining Katz's backoff model with Good-Turing discounting. Our combined model is described as follows [6]:

$$P_{bo}(w_n|w_1...w_{n-1}) = \begin{cases} P^*(w_n|w_1...w_{n-1}) & \text{if} \quad N(w_1...w_{n-1}) > 0 \\ \alpha(w_1...w_{n-1}) \cdot P_{bo}(w_n|w_2...w_{n-1}) & \text{if} \quad N(w_1...w_{n-1}) = 0 \end{cases} \tag{2.10}$$

where P^* is the Good-Turing discounted probability estimate and $\alpha(w_1...w_{n-1})$ is the backoff weight introduced by Katz. The second case in Equation (2.10) shows how the procedure recursively backs off for unseen n-grams, thereby obtaining a nonzero probability estimate normalised by the backoff weight.

It is important to use discounted probability estimates to ensure that there is probability mass to distribute among unseen n-grams during backoff. To ensure that Equation (2.10) produces true probabilities, the backoff weight is chosen to satisfy the following requirement [2];

$$\sum_{\forall k} P_{bo}(w_k|w_1...w_{n-1}) = 1 \tag{2.11}$$

Equation (2.11) ensures that the probability estimate $P_{bo}(\cdot)$ is correctly normalised. In Equation (2.11), k represents the total number of words within a specified vocabulary.

To derive a the value for $\alpha(w_1...w_{n-1})$, we start by defining $\beta(w_1...w_{n-1})$ as the harvested probability mass obtained during discounting. β is calculated by subtracting from 1 the total discounted probability mass for all n-grams seen in our training set and sharing the same prefix [6]:

$$\beta(w_1...w_{n-1}) = 1 - \sum_{\forall k:r>0} P^*(w_k|w_1...w_{n-1}) \tag{2.12}$$

with r indicating the exact number of occurrence for the n-gram in the training set.

We normalise Equation (2.12) to ensure each $(n-1)$-gram only receives a fraction of the total mass. The normalising factor is calculated by summing the probability estimate of all $(n-1)$-grams sharing the prefix as the unseen n-gram [6]:

$$\gamma(w_1...w_{n-1}) = \sum_{\forall k:r=0} P^*(w_k|w_2...w_{n-1}) \tag{2.13}$$

Which is more conveniently expressed as:

$$\gamma(w_1...w_{n-1}) = 1 - \sum_{\forall k:r>0} P^*(w_k|w_2...w_{n-1}) \tag{2.14}$$

where r represents the exact number of occurrence for the original n-gram (not to be confused with the $(n-1)$-gram) in the training set. Combining Equations (2.12) and (2.13) we find:

$$\alpha(w_1...w_{n-1}) = \frac{1 - \sum_{\forall k:r>0} P^*(w_k|w_1...w_{n-1})}{1 - \sum_{\forall k:r>0} P^*(w_k|w_2...w_{n-1})} \tag{2.15}$$

In our case, we substitute values for n in Equations (2.10) and (2.15), yielding our bigram and trigram models:

$$P_{bo}(w_2|w_1) = \begin{cases} P(w_2|w_1) & \text{if} \quad N(w_1,w_2) > 0 \\ \alpha(w_1) \cdot P_{MLE}(w_2) & \text{if} \quad N(w_1,w_2) = 0 \end{cases} \tag{2.16}$$

with

$$\alpha(w_1) = \frac{1 - \sum_{\forall k:r>0} P^*(w_k|w_1)}{1 - \sum_{\forall k:r>0} P_{MLE}(w_k)} \tag{2.17}$$

and

$$P_{bo}(w_3|w_1,w_2) = \begin{cases} P^*(w_3|w_1,w_2) & \text{if } N(w_1,w_2,w_3) > 0 \\ \alpha(w_1,w_2) \cdot P_{bo}(w_3|w_2) & \text{else if } N(w_2,w_3) > 0 \\ \alpha(w_1,w_2) \cdot \alpha(w_2) \cdot P_{MLE}(w_3) & \text{otherwise.} \end{cases} \tag{2.18}$$

with

$$\alpha(w_1,w_2) = \frac{1 - \sum_{\forall k:r>0} P^*(w_k|w_1,w_2)}{1 - \sum_{\forall k:r>0} P^*(w_k|w_2)} \tag{2.19}$$

and where $P_{MLE}(w)$ represents the relative frequency probability estimate for the unigram i.e. $N(w)$ divided by the total count of unigrams in the corpus [6]. The last case in Equation (2.18) indicates a further backoff to the unigram if the bigram is also unseen.

Backoff models do exhibit a flaw under certain circumstances. Take the trigrams $w_iw_jw_k$ for example. The bigram w_iw_j and unigram w_k could both have high counts in our training set, while the trigram $w_iw_jw_k$ is rare or unseen. This could be due to data sparseness, but could also be for a particular reason, possibly grammatical. The backoff model, being very simple, is not capable of capturing this and will assign a probability estimate to the trigram that is most likely too high. Despite this, backoff models have proven to work well in practice [2, 4].

2.1.6. Perplexity

The quality of a language model (LM), is often quantified in terms of a figure known as perplexity. The word, "perplexity" indicates the inability to understand something. For an LM, perplexity shows the average degree of uncertainty the LM experiences in predicting each word in a a sequence.

Perplexity, referred to as PP, is defined by the following equation:

$$PP = [P(\mathbf{w}(0, L-1))]^{-\frac{1}{L}} \tag{2.20}$$

with $P(\mathbf{w}(0, L-1))$ representing the probability of a sequence of L words. Therefore, perplexity is the reciprocal per-word geometric mean probability of the given sequence [3]. Intuitively, perplexity represents the number of words the LM considers might occur next, on average. The lower the perplexity, the more the probabilities assigned by the LM agree with the actual word sequence.

Decomposing perplexity, using Equation (2.2), we find a more easily implemented formula:

$$\log(PP) = -\frac{1}{L}\sum_{i=0}^{L-1}\log[P(w(i)|\mathbf{w}(0, i-1))] \tag{2.21}$$

Substituting the LM probability estimates into the right-hand side of Equation (2.21), we obtain a perplexity value for the LM on the given sequence of words.

2.2. Feedforward Neural Networks

Neural networks, modelled on the human brain, consists of multiple layers of simple but highly interconnected units that are modeled using mathematical equations [1]. Each layer, comprising nodes that emulate biological neurons, is responsible for processing the inputs and finding the best way of combining them. In this report, we focus on using the neural network (NN) to perform regression. By this, we mean specifically taking in a set of input features and predicting an appropriate output.

In a feedforward NN, each node's output is passed on to the following layer and there are no recursive paths. Data only moves in one direction (it is "fed forward"). Feedforward NNs have proven to work well in regression related tasks [1]. This type of NN has the benefit of being effective yet simple to implement.

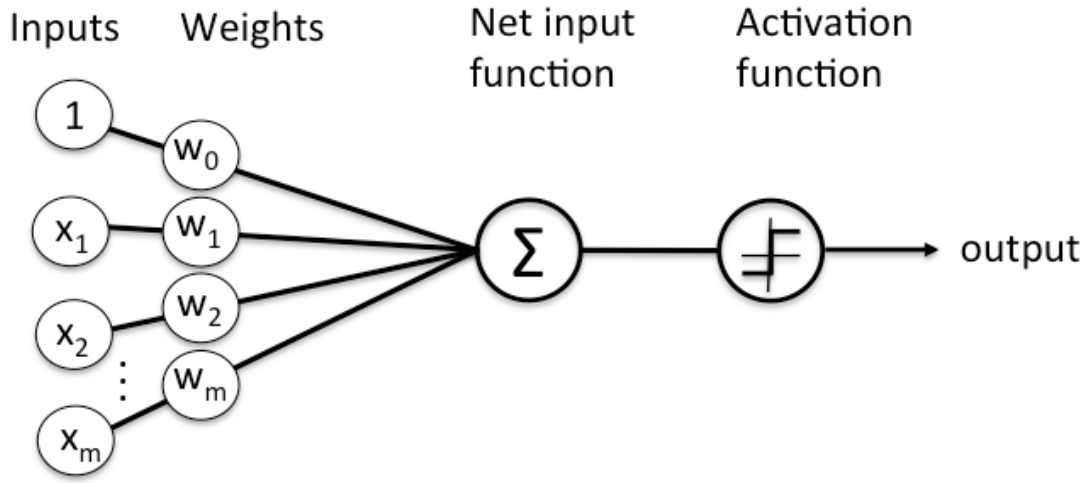


Figure 2.1: A single node, or neuron, producing an output, given multiple inputs. Reproduced from [1].

To better understand how NNs can learn a required task, we focus on a single neuron. Given a set of inputs and weights, the neuron produces an output dictated by its activation function, as shown in Figure 2.1. Each input is multiplied by a weight, w_n, which is

re-estimated after each training cycle. This weight indicates the significance assigned to the corresponding input with regards to the task the NN is trying to learn. The input-weight products are summed and passed through the node's activation function [1]. The activation function determines the extent to which this signal should progress further through the network, ultimately contributing to the output.

As we are working with probabilistic values in language modelling, the sigmoid activation function is well suited for the neurons in our NN. This is because its output is bounded between 0 and 1. The sigmoid activation function is defined by:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.22}$$

Let us further elaborate on how the weights of the NN are re-estimated. Taking the output computed by the NN and comparing it with the target output, we calculate an error that is related to the difference between these values. A loss function dictates how this error is calculated. We will be implementing and comparing the performance of two different loss functions in Chapter 4. These two loss functions are mean square error (MSE) and mean absolute error (MAE). Both have been found to be effective when used for regression [7]. The formulas for MSE and MAE loss are respectively defined as:

$$Loss_{MSE}(x, y) = (x - y)^2 \tag{2.23}$$

and

$$Loss_{MAE}(x, y) = |x - y| \tag{2.24}$$

where x is the value computed by the NN and y is the target value.

Having calculated the error, we can infer the extent to which each weight needs re-estimation using an optimisation algorithm. The optimisation algorithm changes the value of each weight in a way that reduces the error. The optimisation algorithm we will use is Stochastic Gradient Descent (SGD). It is applied repeatedly for several iterations until converges, which occurs when the difference between the target- and actual output reaches local minima.

2.3. Chapter conclusion

This chapter has provided background material regarding current smoothing techniques as well as how a neural network is capable of performing smoothing through regression. In the next chapters, we will apply this to obtain baseline and neural-network-assisted language models.

Chapter 3

Creating a language model

In this chapter we describe the corpus used to construct both the baseline and neural-network-assisted language models. This is followed by a description of the SRILM toolks and how it is used to train a baseline language model for comparison.

3.1. Text corpus

The text corpus used in this report is provided by the University of Stellenbosch. It comprises of text collected from several major South African newspapers [8], separated and divided into year of publication. The data is stored in 17 preproccessed text files, spanning from 2000 to 2016. We partition this corpus by selecting the last six years worth of text, containing 156 million words.

The data has been preprocessed such that each sentence starts on a new line and is delimited by start of sentence (`<s>`) and end of sentence (`<\s>`) tags. For normalisation, all letters are presented in capital letters. Finally, whitespaces separate each token.

The six files, each containing the text accumulated for that year, are divided into three separate sets: training, development and test sets. Figure 3.1 shows this breakdown.

The training set consists of 125 million words, making up 80% of all data provided. The training set is used to estimate a LM as well as train a NN. Each individual file making up the training set, will have its n-gram counts extracted from it. Thereafter the counts will be combined into a single file, to be utilised in training the NN.

The development set is used to fine tune certain parameters in both the LM and NN before final testing. The outputs of several variations of LM and NN architectures may be compared in order to choose the LM or NN best suited for further consideration.

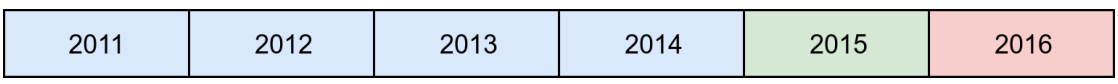


Figure 3.1: The split between training (blue), development (green) and test (red) sets.

Finally each developed LM is evaluated on the test set to compare performance and determine final results. Since the test set is used only in final evaluation, this assessment can be regarded as objective. Our conclusion is based on performance in this set.

3.2. SRILM

3.2.1. Background

The SRI Language Modeling Toolkit has been in development since 1995 by SRI International, a non-profit research organisation. SRILM consists of a collection of C++ libraries, executable programs and helper scripts. It is freely available for educational purposes [9].

SRILM enables the user to estimate and evaluate statistical language models, focusing on n-gram language modeling. SRILM is capable of loading in large text corpora efficiently and estimating the LM parameters. It can also calculate the probability of a given test set and express the result in terms of perplexity [9]. SRILM itself does not perform any text processing, and therefore assumes that the text has already been normalised and tokenised.

3.2.2. Implementation

To apply SRILM, it is first compiled into binary files for execution. Thereafter these binary files are executed and parsed the required parameters. This is all done from a Linux terminal.

Estimating a LM with SRILM is done in two steps. Firstly, a file containing the n-gram counts of a training set is determined. Secondly, these counts are used to estimate probabilities for each n-gram. These two steps can be combined. However, for large training sets, we separate them to prevent exceeding the available memory. The class executed through the command-line tool to accomplish this is *ngram-count*.

We start by feeding *ngram-count* four parameter options, a training set, tuple size, path at which to save the text file containing the n-gram counts, and finally the specified vocabulary. The vocabulary used consisted of the 60 thousand most frequently seen words appearing in the corpus from 2000 to 2014 i.e. in the training set. These counts will also be used to obtain input features for our NN models.

To obtain these counts, the following command is executed in the terminal:

```
ngram-count -text [training_set]
-order [tuple_size]
-write [output_file_path]
-vocab [vocab]
```

The n-gram counts for each file in the training set are saved and combined using *ngram-merge.* Having produced the final n-gram counts file, SRILM estimates a LM by executing:

```
ngram-count -read [ngram_counts] -lm [output_file_path]
```

The LM is stored in ARPA format at the specified path [9]. ARPA files store the total count of n-grams for each tuple size. Following that, each n-gram is listed with its conditional probability (in base 10 logarithmic form) followed by its backoff weight, if applicable. Note that $\log_{10} 0$ is represented as -99.

When estimating a LM, there are two additional parameters options we will be evaluating, namely:

```
-addsmooth [n]
-gt3min [n]
```

Unless otherwise specified, SRILM incorporates Good-Turing discounting together with Katz backoff to smooth the LM [9]. If a LM without any implemented smoothing technique is desired, *-addsmooth 0* should be specified. The *addsmooth* option, indicates that the LM should be smoothed by adding the count n to all n-gram counts. If $n = 0$, then SRILM effectively produces an unsmoothed LM.

By default, SRILM also discards n-grams, of tuple size 3 or higher, occurring only once in the training set. Essentially SRILM treats these n-grams as unseen. This is done for efficiency reasons. The singleton n-grams are known to have little benefit for the perplexity, while occupying a large proportion of computer memory, due to their large number.The argument, *-gt3min [n]* with $n = 0$, prevents this optimisation and includes all singleton n-grams in the LM. However, this could have negative ramifications. Section 3.3 elaborates on this and evaluates whether or not implementing this would be beneficial.

Our final use for SRILM is in evaluating the LM. SRILM does this by calculating perplexity, given a test set, when executing the the *ngram* class in the command-line tool. We parse the test set and LM:

```
ngram -ppl [test_set] -lm [LM] -debug [n]
```

SRILM calculates and outputs two different perplexity scores, referred to as *ppl* and *ppl1*. The former, indicating the perplexity score counting all tokens found in the test set and the latter, excluding end-of-sentence tags. Optionally, the level of detail printed is specified by the *-debug* parameter.

It is important to note that evaluating an unsmoothed LM will produce a perplexity score of infinity due to the zero probability assigned to any unseen n-gram and out-of-vocabulary present in the test set. When SRILM encounters a zero probability n-gram, it sets it aside to continue calculating the perplexity. The count of zero probability n-grams is thereafter displayed in the output. A reduction in perplexity, at the cost of more zero probabilities, is not an indication of improvement on the LM.

3.3. Baseline language model

To implement a NN smoothed LM, we first develop an unsmoothed LM as a baseline. Our objective is to improve on the perplexity scores achieved by this baseline. Together with the newly proposed NN smoothing, traditional smoothing will be used as a reference, to compare the extent to which the perplexity scores have improved. The traditional smoothing technique used is Good-Turing discounting together with Katz backoff.

Before developing theses models we first evaluate what impact the *-gt3min 1* parameter option has on a LM produced by SRILM. We construct two separate models using the training set. Both implement Katz backoff with Good-Turing smoothing. The difference is that the first model excludes singleton trigrams, whereas the second model includes singleton trigrams i.e. the second model was trained with the parameter option *-gt3min 1*. We evaluate both on our development set.

Table 3.1: Perplexity Output of each LM on development set

LM condition	# trigams	Perplexity (*ppl*)	Perplexity (*ppl1*)
Excluding singleton trigrams	10,359,369	152.13	199.63
Including singleton trigrams	36,648,350	150.53	197.43

As seen in Table 3.1, producing an optimised version does slightly decrease its perplexity, however in a small quantity. The drawback of producing an unoptimised LM is that it is large (roughly twice the size of an optimised LM). This makes memory management

another concern. Given the little improvement on perplexity, it is clear that working with an optimised LM is more beneficial.

We now estimate our baseline unsmoothed LM and reference, smoothed LM with default SRILM parameter options i.e. singleton trigrams omitted from the LM. We compute and compare their test set perplexity.

Table 3.2: Perplexity output of each LM on test set

LM condition	Perplexity (*ppl*)	Perplexity (*ppl1*)	Total Zero Probabilities
Unsmoothed	117.54	175.55	35,573
Smoothed	104.94	154.26	0

Table 3.2 shows a 12% to 13% improvement in perplexity when using a smoothed LM. As expected, the unsmoothed LM has a large number of zero probabilities.

3.4. Chapter conclusion

This chapter has described the development of a baseline trigram language model using the SRILM tools and shows the performance of the baseline language model on the test set. In the next chapter we turn our attention to the use of a neural network for language model smoothing.

Chapter 4

Applying a neural network to language model smoothing

In this chapter, we describe the process of designing a NN capable of predicting discounted probability estimates for trigrams occurring less than 8 times in a text corpus. For the remaining trigrams, smoothing is not necessary and we simply use their ML probability estimates.

The trigram probability estimates, computed by our NN, are stored in ARPA file format, and SRILM is used to recalculate the backoff weights. For bigram probability estimates, Good-Turing discounting is used.

Python libraries were written to achieve feature extraction, NN training, and the rewriting of ARPA files, allowing for modular use. See Appendix D for code extracts.

4.1. Python & Pytorch

Python is the chosen programming language in which our NN will be developed and implemented. It is an open-source, interpreted language, with libraries available for NN development. We execute our python scripts using the Jupyter Notebook python environment.

The Pytorch library enables the user to develop a NN from the ground up and to train it. It supports GPU accelerated computing, which reduces model training times significantly. Pytorch includes all necessary activation functions, loss functions and optimiser algorithms required for our experiments. Matrices used in Pytorch are referred to as tensors.

Python incorporates libraries capable of reading and writing text files. These libraries are used to write the probability estimates, computed by the NN, to a file in ARPA format.

4.2. Developing the neural network

4.2.1. Features

We have chosen the following four n-gram counts as inputs for our NN:

- Prefix count
- N-gram count
- $(N-1)$-gram count
- $(N-2)$-gram count

The first two inputs are chosen as they are the counts used when calculating P_{MLE} of a trigram. The backoff n-gram counts are included in order to give the NN more detailed information of the trigram. The aim is for the NN to use these additional counts to better estimate the discounted probability by identifying patterns previously not utilised in Good-Turing discount.

These four inputs are determined for each trigram, occurring less than eight times, in the training set. This is done by loading all training set n-gram counts into a python dictionary and then iterating over each trigram, in each case retrieving from the python dictionary the required counts and saving them to a tensor. Once all counts have been retrieved for all applicable trigrams, we move on to training the NN. Appendix D shows the python script used to perform this.

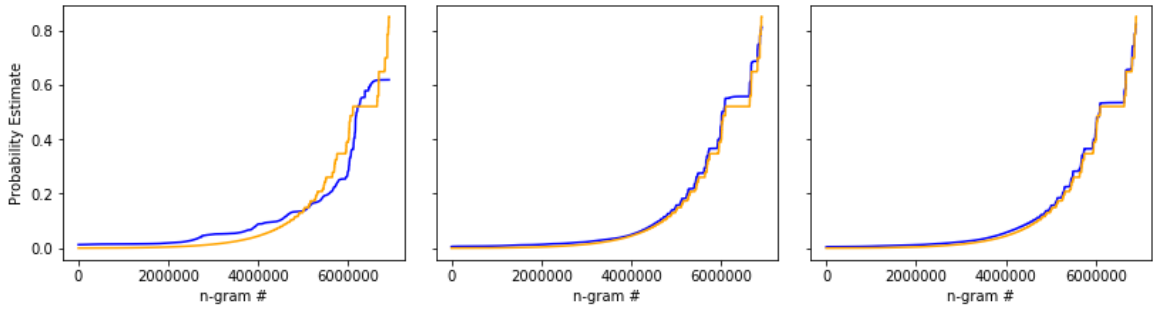
To aid the NN during training, the input counts are normalised to lie between 0 and 1. This allows the NN to find the optimal set of parameters faster and prevents mathematical artifacts associated with floating point number precision [10]. To do so we simply divide 1 by the counts. The resulting value can be considered to represent an n-gram frequency.

We use as target output, the probability estimate for the given trigram, present in the smoothed LM ARPA file. As we iterate over each trigram when obtaining input features, we save the corresponding output to a separate tensor.

4.2.2. Structure

The NN requires four input nodes, one for each input feature, and a single output node. To find the optimal number of hidden layers for the NN, three variations of the NN was developed and tested. Figure 4.1 shows the output produced by a NN with zero, one and two hidden layers separately. Without a hidden layer the NN is unable to

fit to the target output adequately. Adding one hidden layer with the same amount of nodes as the input layer, we see a large improvement in estimating the desired output probability. However, the NN struggles to accurately model higher value probability estimates. With two hidden layers, consisting of four followed by three nodes, the NN is capable of sufficiently estimating the output probabilities. A third hidden layer did not result in any significant improvement.



(a) No hidden layers. **(b)** One hidden layer. **(c)** Two hidden layers.

Figure 4.1: Target output (yellow) and NN output (blue) with varying the number of hidden layers within NN.

Figure 4.2 shows the final NN structure used. Each node incorporates the sigmoid activation function. The NN output, having a value between 0 and 1, represents the probability estimate for the given set of inputs.

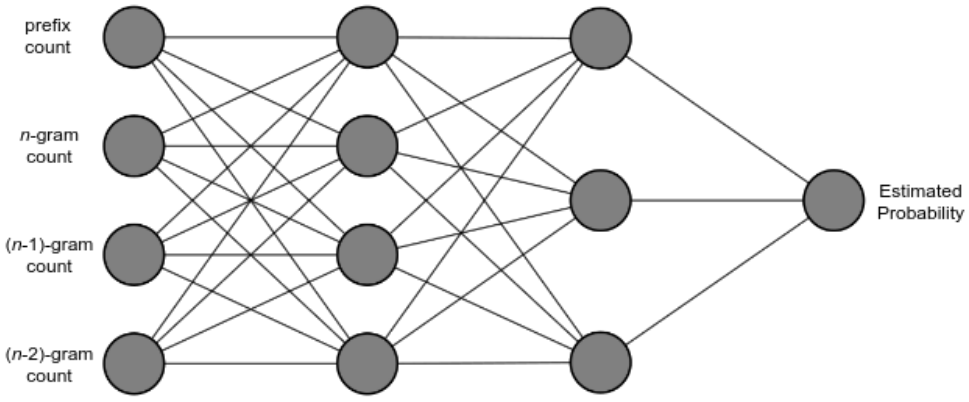
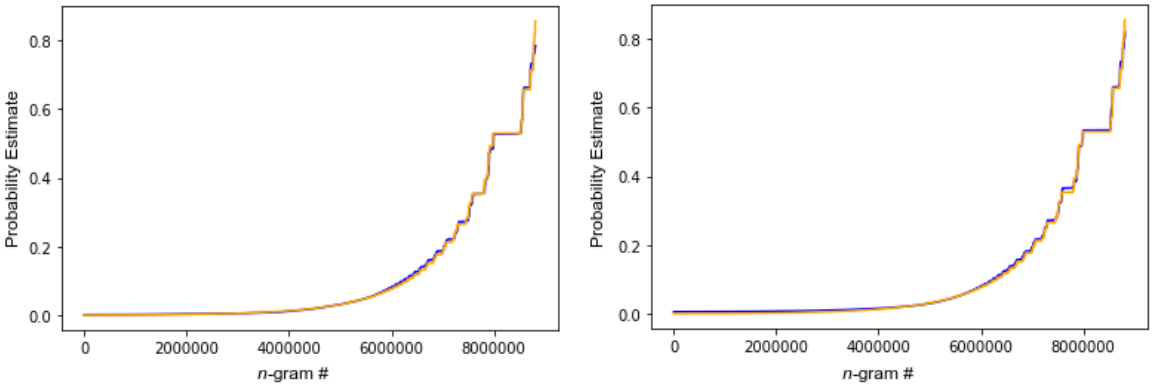


Figure 4.2: Final chosen NN structure for LM smoothing.

4.2.3. Training

Having accumulated our input and output tensors for training, we proceed with evaluating different loss functions and learning rates for our NN. The total number of trigrams in the training set applicable for smoothing is 8 813 319. Each trigram is used in training our final model.

We train a total of ten NNs, with learning rates: 0.001, 0.003, 0.01, 0.03 and 0.1; for each of the loss functions, MAE and MSE. Results are shown in full in Appendix E. Summarising the results, Figure 4.3 shows the learning rate resulting in the best fit on the training set, for both MSE and MAE loss functions. For representation purposes, the trigrams are sorted by their probability estimate value.



(a) MAE with a learning rate of 0.03. **(b)** MSE with a learning rate of 0.1.

Figure 4.3: Target output (yellow) with NN output (blue) after training.

The MSE loss function squares the difference in output and as a result it punishes large errors computed by the NN more severely than smaller errors. As seen during training, a large number of n-gram probability estimates result in small values. Therefore MSE struggles to fit smaller values to the target output. These smaller values will pose a significant problem when they accumulate.

Consequently, MAE outperforms MSE. MAE is therefore chosen as the loss function for our final NN, as it provides a better training fit, especially for smaller output probabilities.

4.3. Implementation

Having decided on the structure of our NN and trained its parameters, we proceed by using the NN to smooth the baseline unsmoothed LM. As we iterate over the ARPA file, we use the NN to compute a discounted probability estimate for each applicable trigram i.e. those occurring less than 8 times. Appendix D shows the python script used to achieve this.

To compute the discounted probability estimate for a given trigram, we first retrieve its associated n-gram counts from the python dictionary. For each applicable trigram, we reassemble this information into a tensor and pass the tensor to the NN. The NN computes

a probability estimate which we use to replace the existing ML probability estimate. Once all applicable trigrams have been discounted, we proceed to recalculate the backoff weights with SRILM. This ensures the probabilities all sum to 1.

As previously mentioned, we use Good-Turing discounted probability estimate to train the NN. These probabilities have already been computed using SRILM and are present in the smoothed ARPA LM file. The python script used to copy the bigram probability estimates over into the unsmoothed LM ARPA file is shown in Appendix D .

4.4. Problems encountered

Table 4.1: Perplexities achieved by the bigram LM with trigram NN probabilities for the dev set.

LM condition	Perplexity (*ppl*)	Perplexity (*ppl1*)	Total Zero Probabilities
Unsmoothed (baseline)	195.50	262.22	754,093
Good-Turing Smoothed (baseline)	152.13	199.63	0
Neural Network Smoothed	113.71	148.25	932,287

Testing our new LM on our development set resulted in very poor performance. Table 4.1 summarises the results. The reduction in perplexity is not indicative of an improvement, as the number of zero probabilities has increased.

Further investigation shows that, for trigrams with very small probabilities, the NN tends to predict probabilities that are too high. Figure 4.4 shows the probability estimates for the first three million trigrams. We see that, for the first two million trigrams, the NN probabilities exceed the ML probability estimates.

This poses a problem when SRILM re-normalises the backoff weights. A small percentage of the trigram probability estimates, instead of being discounted, are instead boosted. This causes probabilities to sum up to values much larger than one, making it impossible for SRILM to correctly recalculate backoff weights. This incorrect estimation is due to how the NN is currently set up. Optimising for learning rates and loss functions, we were able to reduce the extent, but not eliminate it.

The NN struggled to simultaneously model both high and low value probabilities accurately. One possible solution would be to redesign how the NN is implemented. For example by training multiple NNs, each assigned to a bracket within the probability range

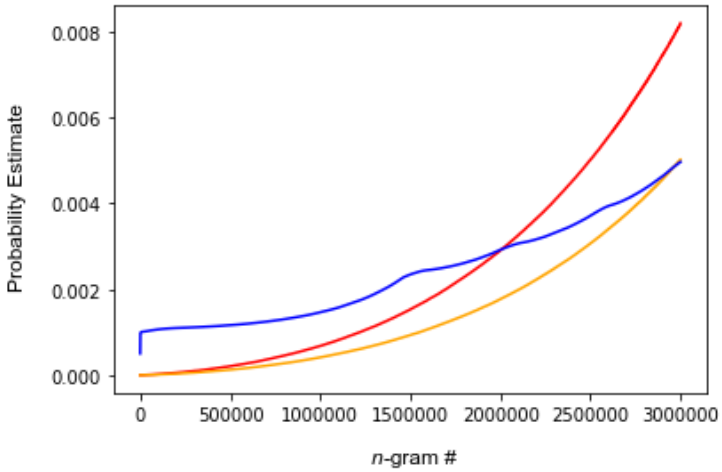


Figure 4.4: P_{MLE} (red), P_{GT} (yellow) and P_{NN} (blue) for the 3,000,000 lowest probability trigrams in the training set.

0 to 1, chosen so that its minimum and maximum output has a small enough difference to ensure all of the probabilities it estimates are accurate.

However, this would unnecessarily complicate and possibly not guarantee better results for smaller probabilities. For the majority of probability estimates, the NN probability is smaller than the MLE. Therefore, we turn our attention to removing the problematic predictions by imposing a threshold. The threshold value prevents the NN from making probability estimates that are smaller than the MLE, by using a threshold value in such cases.

For certain trigrams, it may be correct for the NN to favour them above others. We compare the performance achieved using various threshold values, with some greater than the ML probability estimates. We wish to eliminate the excessive overestimation of n-gram probabilities, whilst no preventing it completely.

The thresholding of probability estimates calculated by the NN is done immediately before the writing of the ARPA file. As we iterate over each trigram occurring less than 8 times, the NN calculates an associated probability estimate. This probability estimate is then compared to a threshold value. If it exceeds the threshold value, we replace it with either the threshold itself or an alternatively calculated probability. The python script used is shown in Appendix D

Table 4.2: Results for LMs with different thresholds.

Threshold Limit	Replacement Value	Times Threshold Exceeded (%)	Perplexity (*ppl*)	Perplexity (*ppl1*)	Total Zero Probabilities
$1.2 \times P_{MLE}(\cdot)$	$1.2 \times P_{MLE}(\cdot)$	18.53	155.69	204.57	118
$1.2 \times P_{MLE}(\cdot)$	$P_{MLE}(\cdot)$	18.53	153.44	201.45	6
$1.1 \times P_{MLE}(\cdot)$	$1.1 \times P_{MLE}(\cdot)$	19.48	154.28	202.61	13
$1.1 \times P_{MLE}(\cdot)$	$P_{MLE}(\cdot)$	19.48	153.39	201.38	3
$P_{MLE}(\cdot)$	$P_{MLE}(\cdot)$	20.56	153.37	201.36	0
$P_{MLE}(\cdot)$	$P_{GT}(\cdot)$	20.56	152.26	199.81	0

Table 4.2 shows the results of various implemented thresholds. Compared to the baseline LMs, it is clear that a threshold value, equal to the corresponding P_{MLE} value, replaced with P_{GT}, yields the best result. It was found that the NN incorrectly estimates about one in every five n-gram probabilities.

4.5. Comparing final results

We have now finalised our NN parameters and settled on the optimal threshold and threshold replacement value. We compare the NN smoothed LM performance against the baseline candidates performance on the test set.

Table 4.3: Results of each LM on test set.

LM condition	Perplexity (*ppl*)	Perplexity (*ppl1*)	Total Zero Probabilities
Unsmoothed (baseline)	117.54	175.55	35,573
Good-Turing (baseline)	104.94	154.26	0
Neural Network with Threshold	104.72	153.91	0

The NN smoothed LM was able to assign a probability to each sequence of words, without ever encountering a zero probability. Table 4.3 shows a slight improvement, however negligible, of perplexity when comparing the NN smoothed LM with the Good-Turing smoothed LM.

4.6. Chapter conclusion

In this chapter we have shown that a neural network can smooth a language model, by the addition of a threshold. The neural network smoothed language model delivering the same performance as Good-Turing smoothing. In the next chapter we apply a different training method to a neural network for smoothing.

Chapter 5

An alternative approach to the application of a neural network

Our findings in Chapter 4 show that it is indeed possible to replicate Good-Turing smoothing using a NN. The limitation, however, is that the results can at best only equal the performance of this smoothing approach and not improve on it. This is expected, as the target output the NN is tasked with imitating a Good-Turing smoothed LM.

In this chapter, we investigate an alternative implementation of NN smoothing. We focus on estimating the ML probability estimate of a trigram directly, as opposed to the discounted probability estimate,. Currently, SRILM calculates the MLE, using Equation 2.5, during the training of an LM. The goal of the newly proposed NN is to compute a improved ML probability estimate for trigrams, seen seldom in our training set. This value will then replace the MLE SRILM calculated, before being discounted according to the Good-Turing method.

5.1. Training set partitioning

To provide such ML probability estimates, we divide our training set into two halves. From one half we take the n-gram counts as inputs to our NN. The second half we use to find the ML probability estimate for the same trigram. This will be the training target for our NN.

The inputs and the targets are compiled as two separate python tensors. Once all possible values have been extracted from the training set, we reverse the roles of the two halves, now using the second half for inputs and the first half for outputs. This ensures the whole training set is utilised for training the NN.

5.2. Additional feature

As previously mentioned in Chapter 2, a trigram ($w_i w_j w_k$) may be unseen because it is rare, or because it should not occur. It may for example happen that the bigram ($w_j w_k$) and unigram (w_i) both have high counts, but that the trigram's count is found to be close to zero. The NN should be able to identify these cases and learn not to over-estimate the probabilities of such trigrams. Therefore, we add the unigram (w_i) count to our input features. The bigram count is already present among the NN inputs as the $(n-1)$-gram count. We refer to this unigram count as the pre-prefix count.

Our NN structure is changed to accommodate for the additional input feature. The input layer now contains five nodes, whereas it previously only consisted of four nodes. Another single node is added to both hidden layers. The resulting NN structure is thus 5-5-4-1. This structure was not further altered as the results it produced were satisfactory.

5.3. Establishing a new baseline

We develop a new baseline to provide a more suited comparison. We follow the same procedure as when computing the baseline in Chapter 3, however, we train the LM on only the second half of the training set.

As with all previously developed LMs, the baseline LM is produced using the MLE SRILM computes for each n-gram. This is the value we will be replacing the output of the NN with, for applicable trigrams. The perplexity is expected to be slightly higher than the previous baseline LM, as the set it is trained on, is now smaller.

5.4. Difference in implementation as opposed to Chapter 4

The self-written python libraries from Chapter 4 are reused, with slight alterations. The NN training script is expanded on to support an additional input feature during training and the script for extracting the applicable n-gram counts is modified to accommodate the new split in the training set.

Whereas the NN considered in Chapter 4 was presented with a discounted probability estimate as training target, in this case the NN is trained to estimate the ML probability estimate of a disjoint dataset (the other half). In this way it is hoped that the NN will learn to estimate the probability of rare events implicilty. Therefore, the ML probability

estimate produced by the NN must first be discounted before being written to the ARPA file. This is to reserve probability mass for redistribution amongst unseen n-grams.

Using SRILM, we determine the Good-Turing discount factors for the training set. These discount factors represent the amount of discounting required and are dependant on the counts of n-gram counts. Good-Turing discounting assumes that n-grams occurring the same number of times in the training set have the same probability. Therefore we have seven different discount factors, one for each number of occurring n-gram considered for smoothing. The discount factor, for n-grams occurring r times, is expressed as:

$$d_r = \frac{P_{GT}(\cdot)}{P_{MLE}(\cdot)} \tag{5.1}$$

From Equation 5.1, we see that multiplying the ML probability estimate with its respective discount factor, yields the Good-Turing discounted probability estimate. This is how we discount the ML probability estimates, computed by the NN, as they are written to the ARPA file.

Once again a threshold value is used to limit the excessive overestimation of n-gram probabilities. The threshold value, as well as the replacement value, is equal to P_{MLE}.

5.5. Results

During the writing of the ARPA file, the NN exceeded the threshold 21.7 % of the time whilst calculating the ML probability estimate. We see from Table 5.1 that the NN was not able to improve the perplexity achieved by the baseline model.

Table 5.1: Result of Good-Turing smoothed LMs, incorporating different MLE values, on test set.

MLE calculated with	Perplexity (*ppl*)	Perplexity (*ppl1*)
SRILM (baseline)	131.46	196.40
Neural Network	131.54	196.52

5.6. Chapter conclusion

The neural network developed in this chapter delivered the same performance as the baseline. It was unable to provide an improvement on the baseline model.

Chapter 6

Analysing performance

Figure 6.1 visualises how a trigram is broken up into smaller n-grams. The count of each of these n-grams is used as input to the NN. In this chapter we investigate the relationship between these n-gram counts and how they affect the probability estimated by the NN. We draw example trigrams from our training set to illustrate our findings. To provide a contrastive comparison, SRILM is used to calculate the MLE for a given trigram, which uses only the prefix and trigram counts.

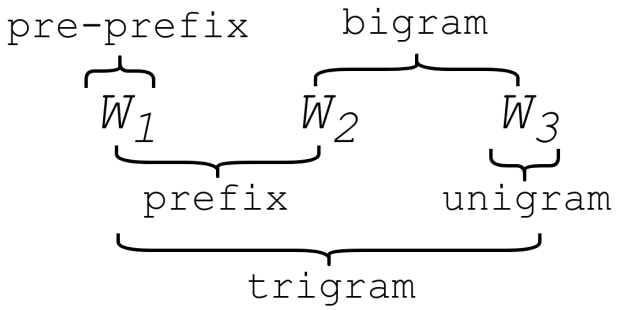


Figure 6.1: A trigram $w_1w_2w_3$ and its breakdown into smaller constituent components.

6.1. Probability estimates of seen n-grams

The SRILM baseline ML probability estimate (P_{MLE}) and the probability estimated by the NN (P_{NN}) are listed together with the n-gram counts in Table 6.1 for two example trigrams, both occurring exactly six times in the training set. These trigrams would thus be subjected to smoothing and the NN probability used to replace the MLE. We selected these trigrams for illustration as they differ only in their unigram and bigram counts.

Before comparing the NN probability estimates in Table 6.1, we calculate the expected occurrence for each trigram. Knowing the training set size, we have $P(\text{FOR}) \approx 0.0083$, assuming words occur independently. Since "TOP EIGHT" occurs 181 times and assuming it occurs independently of "FOR", we expect to see "FOR" immediately proceeding "TOP EIGHT", $C(\text{TOP EIGHT}) \times P(\text{FOR}) \approx 1.5$ times. Through the same calculation we

expect "FOR" to proceed "TOP END" approximately 8.9 times. However, both trigrams are seen exactly six times in the training set.

Table 6.1: Comparison of baseline probabilities, NN estimated probabilities and n-gram counts $C(\cdot)$, for a given trigram.

	FOR TOP EIGHT	FOR TOP END
$C(w_1w_2w_3)$	6	6
$C(w_1w_2)$	285	285
$C(w_1)$	518,703	518,703
$C(w_2w_3)$	181	1077
$C(w_2)$	31,686	31,686
$C(w_3)$	95,973	25,932
P_{MLE}	0.0211	0.0211
P_{NN}	0.0153	0.0157

Table 6.1 shows that the NN did indeed estimate an increased probability for the trigram "FOR TOP END", due to the bigram "TOP END" being much more frequent than "TOP EIGHT". Whilst the MLE SRILM calculates remains unchanged between the two trigrams. The NN estimated a smaller probability than the existing MLE for both trigrams. We assess the significance of this further in the section that follows.

In the NN smoothed LM, estimated probabilities are constrained, using a threshold, to not exceed the MLE. This is critical in the current implementation to maintain probabilities that sum to one and to prevent zero probabilities during perplexity calculations, as previously shown in Table 4.2. While this threshold prevents excessive over-estimation of probabilities, it has a drawback. The freedom of the NN is effectively reduced, as it can no longer favour trigrams by exceeding the MLE. Two trigrams with NN probability estimates exceeding the threshold are listed in Table 6.2.

During training, we observed that the probabilities the NN over-estimated were often for trigrams with a very small MLE. For example, we would expect the NN to over-estimate the probability of both trigrams in Table 6.2, due to their small MLE. This was found to be the case and subsequently the NN estimated probability was not written to the ARPA file. Instead, the MLE was used. The NN estimated a larger probability for "JOBS FOR CASH" than "JOBS FOR MONEY", but unfortunately the imposition of the threshold prohibits us from utilising this.

Table 6.2: Comparison of probabilities estimated for trigrams exceeding the threshold.

	JOBS FOR MONEY	JOBS FOR CASH
$C(w_1w_2w_3)$	2	2
$C(w_1w_2)$	521	521
$C(w_1)$	8,975	8,975
$C(w_2w_3)$	183	771
$C(w_2)$	518,703	518,703
$C(w_3)$	28,811	6,568
P_{MLE}	0.00384	0.00384
P_{NN}	0.00404	0.00409

Next, we observe two trigrams that both have substantially lower NN than MLE probability estimates. One is a grammatically incorrect trigram ("OF LOT OF"), which ideally should not be present in our training set at all, but in practice, it is seen four times. Note that, the bigram count $C(w_2w_3)$ for "OF LOT OF" is much higher than its prefix count $C(w_1w_2)$, while the prefix and bigram counts for the second trigram "<s> RECORDS WERE" only differ by ten. We will be evaluating the significance of this on the estimated probability. Table 6.3 lists both trigrams with their respective input counts.

Table 6.3: Comparison of probabilities estimated for trigrams with all NN inputs listed.

	OF LOT OF	<s> RECORDS WERE
$C(w_1w_2w_3)$	4	3
$C(w_1w_2)$	8	46
$C(w_1)$	1,233,800	3,041,784
$C(w_2w_3)$	10,899	56
$C(w_2)$	17,432	2,554
$C(w_3)$	1,233,800	169,063
P_{MLE}	0.5	0.0652
P_{NN}	0.399	0.0532

It is unusual for a trigram to have a high bigram count but also a small prefix count and vice versa. When this happens, it may indicate that the trigram occurs seldom, not due to a lack of data, but because it should not occur, for example because it is ungrammatical. The second trigram does not have the same contrast in its bigram and prefix counts. However both trigrams are discounted to some extent. For the trigram "OF LOT OF" it is

desired of the NN to estimate a probability, discounted to a greater extent than what the trigram "<s> RECORDS WERE" is , but this is not the case. Therefore we investigate these two sets of counts further in the section that follows.

6.2. Investigation of the behaviour of the NN estimates

We now seek to first assess how the addition of the unigram, pre-prefix and bigram counts affect the probability estimated by the NN. These counts are not utilised by SRILM when calculating the MLE for a given trigram. The MLE is always the same, regardless of the value of these additional counts. Thereafter we focus on the contrast between bigram and prefix counts.

The NN is given the inputs shown in Table 6.4, adapted from the two trigrams in Table 6.3. Then, the unigram, pre-prefix and bigram counts are varied individually. This is done in order to investigate the effect the differing bigram and prefix counts have on the output probability. It was found that increasing the values of the counts, to beyond those shown in Table 6.4, had little further impact.

Table 6.4: Artificial n-gram counts used for input for two separate trigrams.

	OF LOT OF	<s> RECORDS WERE
$C(w_1w_2w_3)$	4	3
$C(w_1w_2)$	8	50
$C(w_1)$	1,000	1,000
$C(w_2w_3)$	8	50
$C(w_3)$	1,000	1,000
P_{MLE}	0.5	0.06

For the trigram "OF LOT OF" ($P_{MLE} = 0.5$), we see from Figure 6.2 that an increase in either the pre-prefix or unigram counts, reduces the output probability of the NN. An increase in the pre-prefix count $C(w_1)$ without an accompanying increase in the trigram or bigram counts may indicate stronger evidence that, although w_1 is more frequent, its combination with w_2 and w_3 is not, and hence its probability is reduced. The same argument can be made for an increase in the unigram count $C(w_3)$.

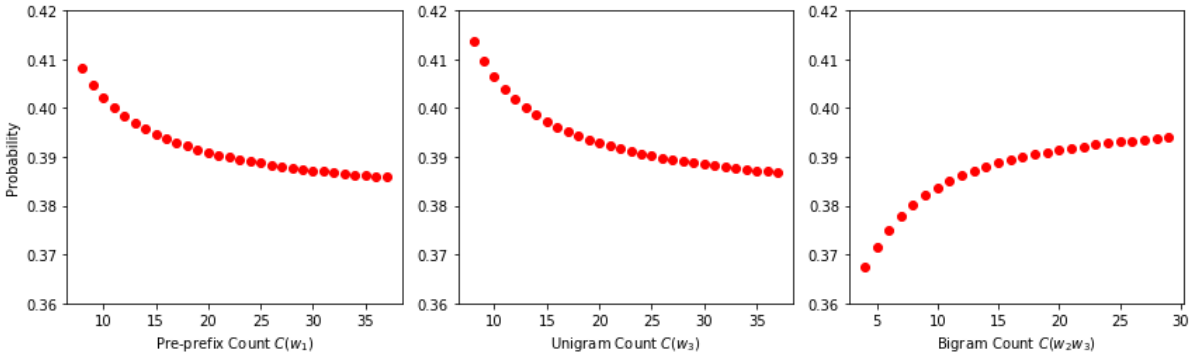


Figure 6.2: The effect on the NN estimated probability for the trigram "OF LOT OF" ($P_{MLE} = 0.5$) when artificially varying individual NN inputs.

For the trigram "<s> RECORDS WERE" ($P_{MLE} = 0.06$), we see from Figure 6.3 that a change in the pre-prefix and unigram counts has almost no effect. This suggests that if the combination of w_2 and w_3 is seen often enough, in this case 50 times, the probability for the trigram should remain as estimated, regardless of $C(w_1)$ and $C(w_3)$.

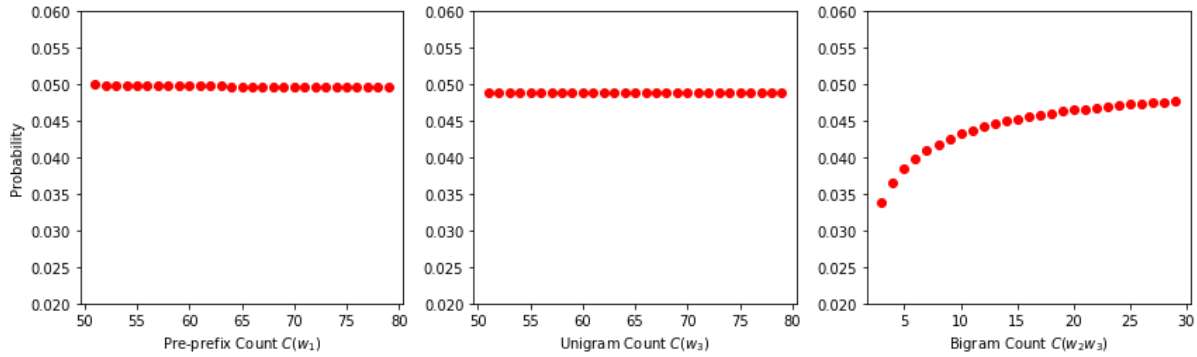


Figure 6.3: The effect on the NN estimated probability for the trigram "<s>RECORDS WERE" ($P_{MLE} = 0.06$) when artificially varying individual NN inputs.

For both simulated trigrams the output probability only increased with bigram count. However, it would be expected that if $C(w_2w_3)$ increases exclusively, the trigram probability would decrease. We backoff to the (n-1)-gram w_2w_3, to show using Equation (2.5) that increasing $C(w_2w_3)$, while $C(w_2)$ is kept constant, increases the (n-1)-gram probability. We hypothesise that the NN is influenced by this increase in backoff n-gram probability, causing the trigram probability to increase as the bigram count increases. This is exhibited by the trigrams in Table 6.1.

If we, instead of increasing the bigram count, increase the prefix count, whilst keeping the bigram count constant, we see in Figure 6.4 that the results are as expected. When w_3 is very rarely seen following w_2 it is likely to results in a lower MLE and should further decrease with a rise in prefix count $C(w_1w_2)$. The NN reduced the probability estimate significantly when compared to the MLE SRILM calculated. Increasing the bigram and

prefix counts simultaneously, did not produce a constant decrease, instead the amount of decrease in probability fluctuated, further supporting the argument above.

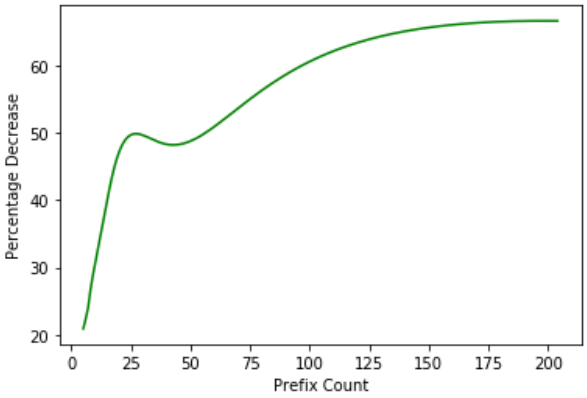


Figure 6.4: The percentage decreased probability estimated by the NN for a trigram when artificially varying the prefix count while maintaining a constant bigram count of eight.

In general the NN does estimate probabilities that are less than the MLE, making it difficult to distinguish which trigrams are under-estimated for a particular reason, showed with the trigrams in Table 6.3. However, the NN learned to further reduce the probability estimates for trigrams which encapsulate largely differing n-gram counts, as shown in Figure 6.4.

6.3. Probability estimates of unseen n-grams

Due to the increase in the number of trigram probability estimates that have been reduced from the MLE, the amount of discounted probability mass is effectively increased. The threshold also contributes to this, as each probability that the NN estimates for a given trigram is constrained to never exceed what it was before. This results in more redistributed probability mass during backoff.

The NN in its current implementation is not responsible for re-estimating the probabilities of unseen n-grams or those occurring more than seven times. However, due to the increase in probability mass redistributed, the probabilities for unseen n-grams are indirectly affected by the NN. The probabilities shown in Table 6.5 are obtained from the ARPA file for each LM.

As expected, a trigram occurring more than seven times show no change in probability. For both unseen n-grams, the probability has increased. The trigram "RESULTS SHOW A" had its probability revised downward by the NN. On the other hand, the NN increased the probability estimated for the unseen n-grams, thereby reducing the difference between

seen and unseen n-grams.

Table 6.5: Probabilities for trigrams taken from LM.

$w_1w_2w_3$	Baseline LM $P(w_3\|w_1, w_2)$	NN smoothed LM $P(w_3\|w_1, w_2)$	Trigram Count
THE RESULTS SHOW	1.09e-03	1.09e-02	23
RESULTS SHOW A	4.24e-02	3.30e-03	4
RESULTS SHOW AN	1.42e-03	1.53e-03	0
RESULTS SHOW HOW	3.84e-03	4.15e-04	0

The increase in redistributed probability mass resulted in the backoff weights increasing in value in the NN smoothed LM. This explains why the NN smoothed LM estimates larger probabilities for certain n-grams during backoff. This improves the ability of the LM to interpret unseen trigrams, which are likely occur, as shown in Table 6.5.

Chapter 7

Conclusion

Both goals set out at the start of the project were achieved. Two neural networks were developed to estimate, separately, a discounted probability as well as a maximum likelihood estimate for a given trigram. Each neural network was successfully implemented to smooth an unsmoothed language model. Both neural network smoothed language models matched the perplexity achieved by the baseline language model on the test set. In addition, the second neural network was shown to reduce the over-estimation of very rare n-grams in the dataset.

This was possible through thresholding the probability, estimated by the neural network, before writing it to the language model. With the threshold enforced, both neural networks affected roughly 80% of the probabilities by smoothing. The remaining probability estimates remained the Good-Turing estimates.

Further analysis showed how the second neural network is able to better distinguish between several n-grams that would all be given the same maximum likelihood probability. The neural network also properly penalised n-grams that showed through their constituent counts that they should be less likely than maximum likelihood estimation currently models them. This allowed an increased amount of probability mass to be reintroduced through backoff, improving the difference in probability between seen and unseen trigrams.

Recommendations

The solutions proposed in this report for the implementation of a neural network in language model smoothing has a shortcoming that can be improved on. If we consider values exceeding the threshold as an incorrect estimation, the neural network might be considered to have an accuracy of only 80%. This was caused by the enforcement of the threshold on the probability estimates of the NN. The threshold itself also introduced a problematic situation.

In the experiments carried out in this project, it was not possible to train the neural network to estimate both low and high probability values without problematic over-

estimation. A suggested solution is to divide the probability range of 0 to 1 into multiple brackets and to train a neural network on each individually. This should improve the accuracy of the probability estimated for any given trigram.

The threshold, while proving to be a simple solution to the over-estimation, compromised the freedom of the neural network to estimate probabilities. If the neural network were allowed to exceed the maximum likelihood estimate of a trigram, all of the probabilities in the language model would have to be re-normalised retroactively. It is recommended that the implementation of the threshold be re-evaluated by empirically investigating the effect of such retrospective re-normalisation. If this were successful, it would also eliminate the need to divide the probability into bands, as mentioned above

Our final recommendation is a fundamental change to the implementation of the neural network for smoothing. In this report, the neural network was only trained on seen trigrams. The neural network was never tasked with estimating probabilities for unseen trigrams, as Katz backoff was responsible for this. It is recommended to further investigate how one can implement the neural network to directly estimate probabilities for unseen n-grams.

Bibliography

[1] C. Nicholson, "A Beginner's Guide to Neural Networks and Deep Learning," 2019. [Online]. Available: https://skymind.ai/wiki/neural-network. [Accessed: 24- Aug- 2019].

[2] D. Jurafsky and J. Martin, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall, 2009.

[3] T. R. Niesler, "Category-based statistical language models," Ph.D. dissertation, University of Cambridge Cambridge, UK, 1997.

[4] C. Manning, C. Manning, and H. Schütze, *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

[5] I. J. Good, "The Population Frequencies of Species and the Stimation of Population Parameters," *Biometrika*, vol. 40, no. 3-4, pp. 237–264, 1953.

[6] L. Mak, "Next Word Prediction using Katz Backoff Model," 2019. [Online]. Available: https://rpubs.com/leomak/TextPrediction_KBO_Katz_Good-Turing. [Accessed: 6- Aug- 2019].

[7] P. Jha, "A brief overview of loss functions in pytorch," 2019. [Online]. Available: https://medium.com/udacity-pytorch-challengers/a-brief-overview-of-loss-functions-in-pytorch-c0ddb78068f7. [Accessed: 25- Aug- 2019].

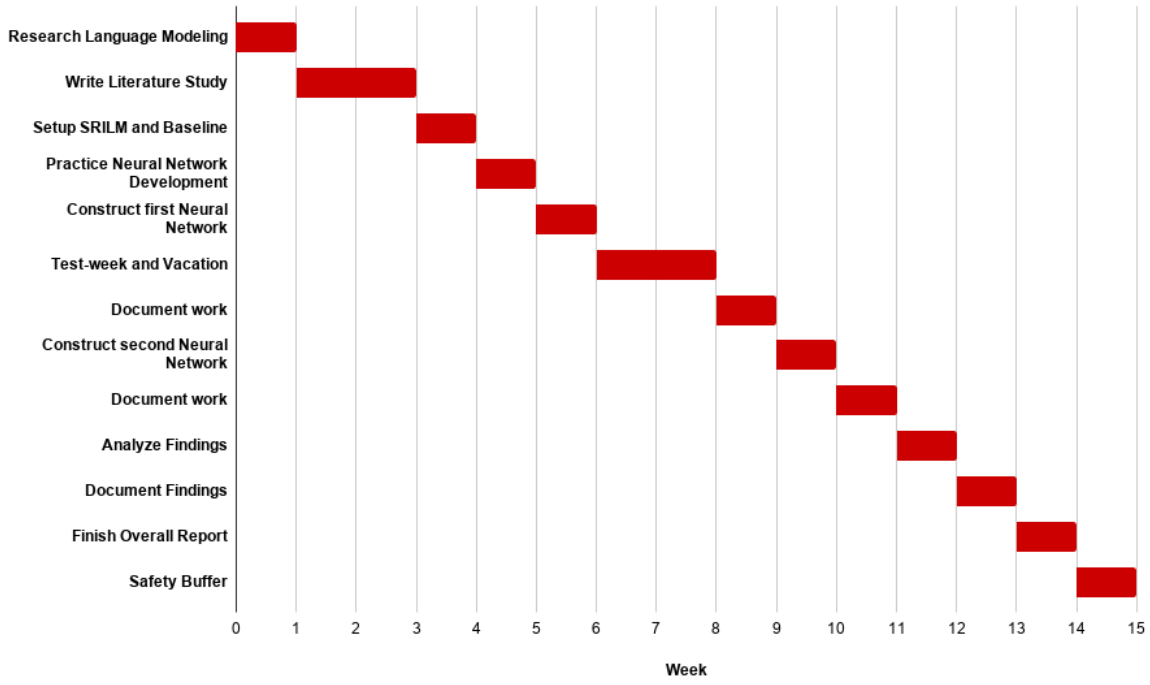
[8] H. Kamper, F. Wet, T. Hain, and T. Niesler, "Capitalising on North American speech resources for the development of a South African English large vocabulary speech recognition system," *Computer Speech and Language*, vol. 28, pp. 1255–1268, 2014.

[9] A. Stolcke, "SRILM - an extensible language modeling toolkit," Proceedings of *Interspeech*, Denver, Colorado, 2002.

[10] J. Jordan, "Normalizing your data (specifically, input and batch normalization)," 2019. [Online]. Available: https://www.jeremyjordan.me/batch-normalization/. [Accessed: 10- Sep- 2019].

Appendix A

Project planning schedule



Appendix B

Outcomes compliance

Table B.1: How each of the required ECSA outcomes were achieved during execution of the project.

ELO #	Title	Evidence	Reference
1	Problem solving	Formulating python code capable of rewriting ARPA files.	Section 4.3, page 21
2	Application of scientific and engineering knowledge	Setting up a neural network to estimate probabilities.	Chapters 4 and 5, pages 18 and 26
3	Engineering design	Developing an optimal neural network structure and finding the best training fit.	Chapters 4 and 5, pages 18 and 26
4	Investigations, experiments and data analysis	Investigating initial poor performance of the neural network and finding improvement through threshold. As well as the analysis of the neural network results.	Section 4.4, page 22. Chapter 6, page 29
5	Engineering methods, skills and tools, including information technology	Entirety of self written code used in implementation of both neural networks.	Chapters 4 and 5, pages 18 and 26
6	Professional and technical communication	Composing research into literature study.	Chapter 2
8	Individual work	Creation of self written python libraries for problem solving.	Chapters 4 and 5, pages 18 and 26
9	Independent learning ability	Taking research done before hand regarding language modeling and applying it to the analysis of the neural network's performance.	Chapter 6, page 29

Appendix C

ARPA File Format

```
\data\
ngram 1=n1
ngram 2=n2
...
ngram N=nN

\1-grams:
p     w          [bow]
...

\2-grams:
p     w1 w2      [bow]
...

\N-grams:
p     w1 ... wN
...

\end\
```

Appendix D

Python Code Extracts

```
file = '../../rsc/train_counts.txt'
first_read = open(file ,'r')

num_lines = sum(1 for line in open(file,'r'))

ngram_dict = {}
for x in tqdm(first_read, total=num_lines):
    line = x.split('\t')
    r = int(line[-1])
    ngram_dict[line[0]] = r
```

Figure D.1: Loading n-gram counts to dictionary.

```
file = '../../rsc/train_counts.txt'
first_read = open(file ,'r')

num_lines = sum(1 for line in open(file,'r'))
count = 0

for x in tqdm(first_read, total=num_lines):
    line = x.split('\t')
    r = int(line[-1])
    ngram = line[0].split(' ')
    tuple_size = len(ngram)

    if tuple_size == 3 and r < 8 and r!= 1: #applicble n-grams
        #prefix count
        inputs[0][count] = ngram_dict[ngram[0] + ' ' + ngram[1]]
        #trigram count
        inputs[1][count] = ngram_dict[line[0]]
        #backoff bigram count
        inputs[2][count] = ngram_dict[ngram[1] + ' ' + ngram[2]]
        #unigram count
        inputs[3][count] = ngram_dict[ngram[2]]

        count += 1
```

Figure D.2: Retrieving inputs from n-gram counts file.

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(4, 4)
        self.fc2 = nn.Linear(4, 3)
        self.fc3 = nn.Linear(3, 1)


    def forward(self, x):
        x = torch.sigmoid(self.fc1(x))
        x = torch.sigmoid(self.fc2(x))
        x = torch.sigmoid(self.fc3(x))
        return x
```

Figure D.3: Neural network class.

```
net = Net().cuda()
optimizer = optim.SGD(net.parameters(), lr=0.01, momentum=0.9)

criterion = torch.nn.L1Loss()  #mean absolute error
#criterion = torch.nn.MSELoss()  #mean square error
```

```
## run the model for 3 epochs
for epoch in range(0,3):
    for x in tqdm(range(len(inputs)),position=0, leave=True):
        optimizer.zero_grad() #reset gradients

        ## 1. forward propagation
        net_out = net((inputs[x:x+1,:]))

        ## 2. loss calculation
        loss = criterion(net_out, outputs[x].reshape(1,1))
        #print(target[x])

        ## 3. backward propagation
        loss.backward()

        ## 4. weight optimization
        optimizer.step()
```

Figure D.4: Neural network training.

```
file = '../../rsc/unsmoothedLM.arpa'
first_read = open(file ,'r')
new_file = open("../../rsc/output_LM.arpa","w+")
num_lines = sum(1 for line in open(file,'r'))
current_ngram_len = 0
TH_exceeded = 0
count = 0
nn_input = torch.zeros(1, 4, dtype = torch.float, device = device)

for x in tqdm(first_read, total=num_lines, position=0, leave=True):
    if x == '\\end\\\n':
        current_ngram_len = -1
        new_file.write(x)
    elif x == '\n':
        new_file.write(x)
    elif current_ngram_len < 3:
        new_file.write(x)
    elif current_ngram_len == 3: #trigrams
        line = x.split('\t')
        r = ngram_dict[line[1][:-1]] #get ngram occurence

        if r > 1 and r < 8: #only smooth applicable trigrams
            prob = 10**float(line[0])  #retrieve ngram prob from ARPA
            ngram = line[1].split(' ') #retrieve ngram
            ngram[2] = ngram[2][:-1]
            count += 1

            ######setup nn input#######
            nn_input[0][0] = ngram_dict[ngram[0] + ' ' + ngram[1]]  #prefix count
            nn_input[0][1] = ngram_dict[line[1][:-1]]               #trigram count
            nn_input[0][2] = ngram_dict[ngram[1] + ' ' + ngram[2]]  #backoff bigram count
            nn_input[0][3] = ngram_dict[ngram[2]]                   #unigram count
            nn_input = 1/nn_input #normalise

            MLE = nn_input[0][0]/nn_input[0][1]  #get Threshold
            smoothed_prob  = net(nn_input)       #get NN value

            if smoothed_prob > (MLE): #evaluate threshold
                TH_exceeded += 1
                new_file.write(x) #write GT value
            else:
                logbase = math.log(smoothed_prob, 10)
                new_file.write('{:.7f}\t{}\n'.format(logbase, line[1][:-1]))
        else:
            new_file.write(x)

    if x == '\\1-grams:\n':
        current_ngram_len = 1
    if x == '\\2-grams:\n':
        current_ngram_len = 2
    if x == '\\3-grams:\n':
        current_ngram_len = 3

new_file.close()
print('MLE estimates exceeded: {:.2f}%'.format((TH_exceeded/count)*100))
```

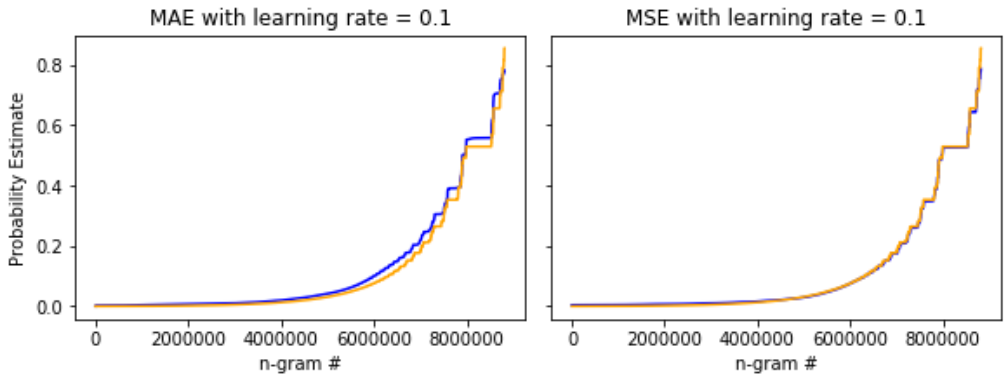
Figure D.5: Rewriting an ARPA file with NN calculated probability estimates.

```
file1, file2 = '../../rsc/nnLM.arpa', '../../rsc/smoothedLM.arpa'
first_read, second_read = open(file1 ,'r'), open(file2 , 'r')
new_file = open('../../rsc/output_LM.arpa',"w+")
num_lines = sum(1 for line in open(file1,'r'))

for x in tqdm(range(0,num_lines), position=0, leave=True):
    line1, line2 = first_read.readline().split('\t') , second_read.readline().split('\t')

    if line1[0][0] != '\\' and line1[0] !='\n' and line1[0][0] != 'n':
        ngram = line1[1].split(' ')
        r = len(ngram)

        if r == 2: #read until bigrams are reached in ARPA file
            for y in line2: #write smoothed bigram value
                new_file.write(y)
                if y[-1:] != '\n':
                    new_file.write('\t')
        else:
            for y in line1: #write values for unigrams and trigrams
                new_file.write(y)
                if y[-1:] != '\n':
                    new_file.write('\t')
    else:
        for y in line1:
            new_file.write(y)

new_file.close()
```

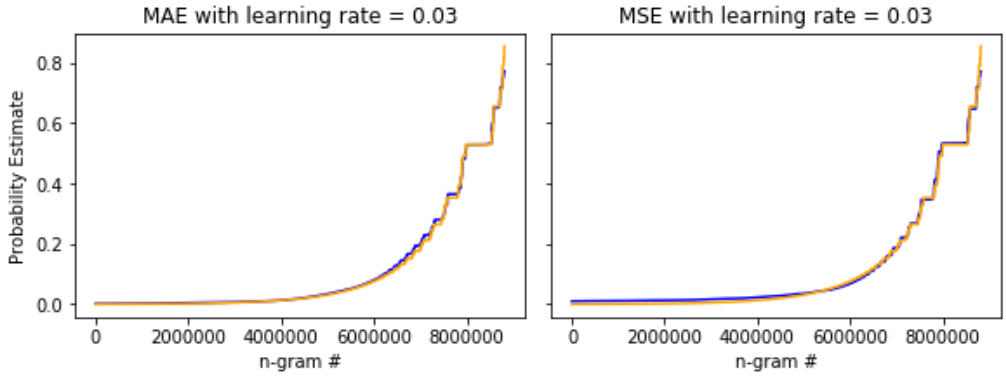
Figure D.6: Copy bigram probabilities from smoothed ARPA file to NN ARPA file.

Appendix E

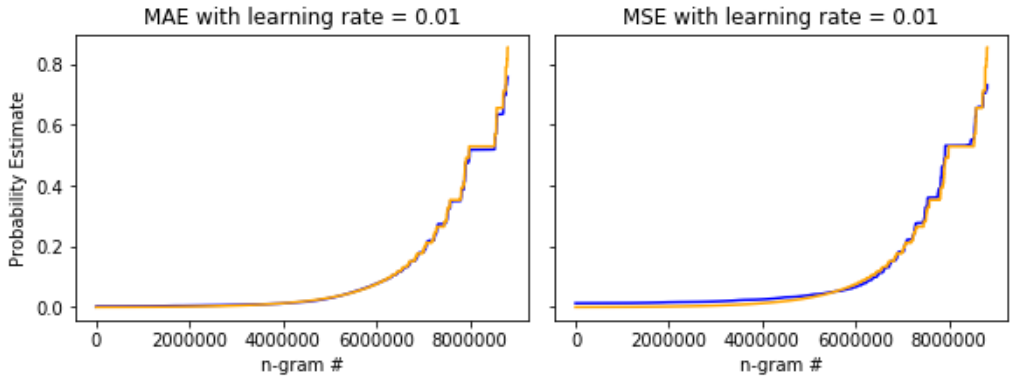
NN Training Results



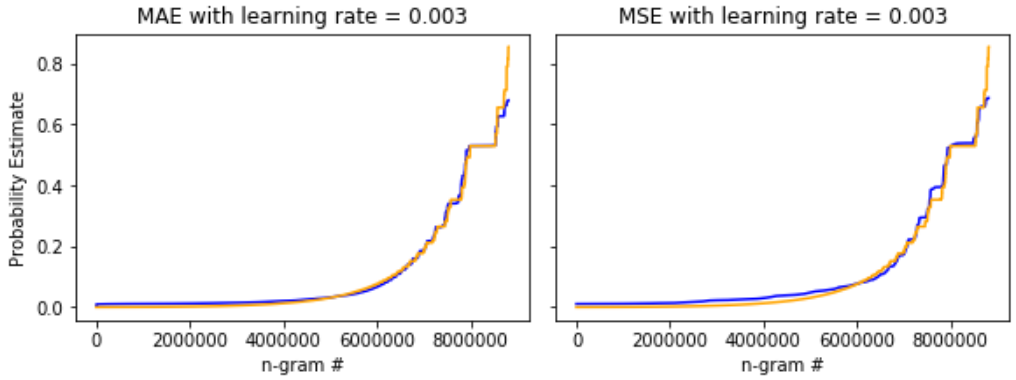
(a)



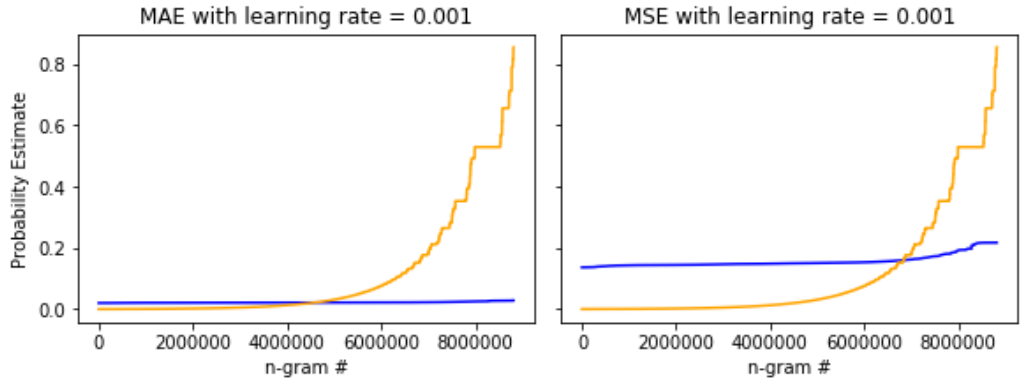
(b)



(c)



(d)



(e)

Figure E.1: Training fit compared between MSE and MAE for varying learning rates, with target output (yellow) and NN output (blue).