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.

Supervisor: Prof. T. R. Niesler

November 2019

Acknowledgements

Declaration

I, the undersigned, hereby declare that the work contained in this report is my own original work unless otherwise stated.

Signature: ....
Werner Van der Merwe

Date: ....

Abstract

English

The English abstract.

Afrikaans

Die Afrikaanse uittreksel.

Contents

Declaration ii

Abstract iii

List of Figures vi

List of Tables vii

Nomenclature 1

1. Introduction 2

2. Literature Study 3

2.1. The Language Model . . . 3

2.1.1. The Need for the Statistical Language Model . . . 3

2.1.2. Mathematical Preliminaries . . . 4

2.1.3. N-Gram Language Model . . . 4

2.1.4. Estimating N-gram Probabilities . . . 5

2.2. Smoothing . . . 6

2.2.1. Good-Turing . . . 6

2.2.2. Backing-Off . . . 7

2.3. Perplexity . . . 9

2.4. Feedforward Neural Networks . . . 10

3. Creating a language model 12

3.1. Text corpus . . . 12

3.2. SRILM . . . 13

3.2.1. Background . . . 13

3.2.2. Implementation . . . 13

3.3. Baseline Language Model . . . 15

4. Applying a neural network for language model smoothing 17

4.1. Python & Pytorch . . . 17

4.2. Developing the neural network . . . 18

4.2.1. Features . . . 18

4.2.2. Structure . . . 18

4.2.3. Training 19
4.3. Implementation 20
4.4. Problem encountered 20
4.4.1. Initial results 20
4.4.2. Comparing various thresholds 22
4.5. Comparing final results 22

5. An alternative approach to the application of a neural network 24
5.1. Training set partitioning 24
5.2. Additional feature 25
5.3. Establishing a new baseline 25
5.4. Difference in implementation as opposed to Chapter 4 25
5.5. Results 26

6. Analysing performance 27
6.1. N-grams present in training set 27
6.2. NN characteristics 29
6.3. Effect on unseen trigrams 32

7. Conclusion 33

Bibliography 34

A. Project planning schedule 35

B. ECSA outcomes 36

C. ARPA File Format 37

D. Python Code Extracts 38

E. NN Training Results 41

List of Figures

2.1. A single node, or neuron, producing an output, given multiple inputs. Reproduced from [1]. 10

3.1. The split between training (blue), development (green) and test (red) sets. 12

4.1. NN structure. 19
4.2. Target output (yellow) with NN output (blue) after training. 19
4.3. P_{MLE} (red), P_{GT} (yellow) and P_{NN} (blue) for the 3,000,000 lowest probability trigrams in the training set. 21

6.1. A trigram $w_1w_2w_3$ and its breakdown into smaller constituent components. 27
6.2. A simulated trigram with MLE = 0.5. 30
6.3. A simulated trigram with MLE = 0.06. 30
6.4. The percentage decreased probability estimated by the NN, with bigram count kept constant. 31

D.1. Loading n-gram counts to dictionary. 38
D.2. Retrieving inputs from n-gram counts file. 38
D.3. Neural network class. 39
D.4. Neural network training. 39
D.5. Rewriting an ARPA file with NN calculated probability estimates. 40
D.6. Copy bigram probabilities from smoothed ARPA file to NN ARPA file. . . 40

E.1. Training fit compared between MSE and MAE for varying learning rates, with target output (yellow) and NN output (blue). 42

List of Tables

2.1. Increase of parameters for n-gram size . . . 5

3.1. Perplexity Output of each LM on development set . . . 15
3.2. Perplexity output of each LM on test set . . . 16

4.1. Perplexities achieved by the bigram LM with trigram NN probabilities for the dev set. . . . 20
4.2. Results for LMs with different thresholds. . . . 22
4.3. Results of each LM on test set. . . . 23

5.1. Result of Good-Turing smoothed LMs, incorporating different MLE values, on test set. . . . 26

6.1. Comparison of baseline probabilities, NN estimated probabilities and n-gram counts $C(\cdot)$, for a given trigram. . . . 28
6.2. Comparison of probabilities estimated for trigrams exceeding the threshold. 28
6.3. Comparison of probabilities estimated for trigrams with all NN inputs listed. 29
6.4. Starting input n-gram counts for two separate trigrams. . . . 30
6.5. Probabilities for trigrams taken from LM. . . . 32

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.

Acronyms and abbreviations

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

Chapter 1

Introduction

Chapter 2

Literature Study

2.1. The Language Model

2.1.1. The Need for the Statistical Language Model

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.

2.1.2. 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 models that predict some future unit without looking too far into the past [2].

2.1.3. N-Gram Language Model

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 next 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 being 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. Models tend mostly to incorporate bigrams and trigrams [4].

2.1.4. Estimating N-gram Probabilities

The simplest way to estimate the probability is with a Maximum Likelihood Estimation, known as MLE. To obtain this estimate, we observe the n-gram counts from a training corpus. These counts are thereafter normalized to give [2]:

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

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 have not necessarily seen the same n-grams. Even if we increase the size of the training set, the majority of words are still considered to be uncommon and n-grams containing these words are rare [4]. The validation set will most likely contain several unseen n-grams whose relative frequencies according to Equation (2.5) is considered to be 0.

The probability of occurrence for a sequence of words should never be 0, however since all 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 unseen n-grams [4]. The next section will discuss different kinds of smoothing and how they are implemented.

2.2. Smoothing

2.2.1. Good-Turing

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 probability estimate.

We begin with the Good-Turing probability estimate:

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

Where $N_{\sum}$ is the total number of n-grams and r^* is a re-estimated frequency 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 as counts and combining Equations (2.6) and (2.7) we obtain:

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

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 considered as the probability of occurrence of any unseen n-gram.

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 posisitive estimates before calculating the probability.

2.2.2. Backing-Off

Instead of redistributing the probability mass equally amongst 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 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, 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

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

Where $P_{MLE}(w)$ represents a MLE of that 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 $w_i w_j w_k$ for example. The bigram $w_i w_j$ and unigram w_k could both have high counts in our training set, however there could be a particular reason, possibly grammatical, for these words to not appear as a trigram. 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 practise [2, 4].

2.3. Perplexity

The quality of a language model (LM), is often quantified in terms of a measurement known as perplexity. By definition, “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. Perplexity is the reciprocal per-word geometric mean probability of the given sequence [3]. In layman's terms, perplexity is the number of words the LM considers might occur next. The lower the perplexity, the more accurate the LM assigns probabilities to n-grams.

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.

2.4. 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.

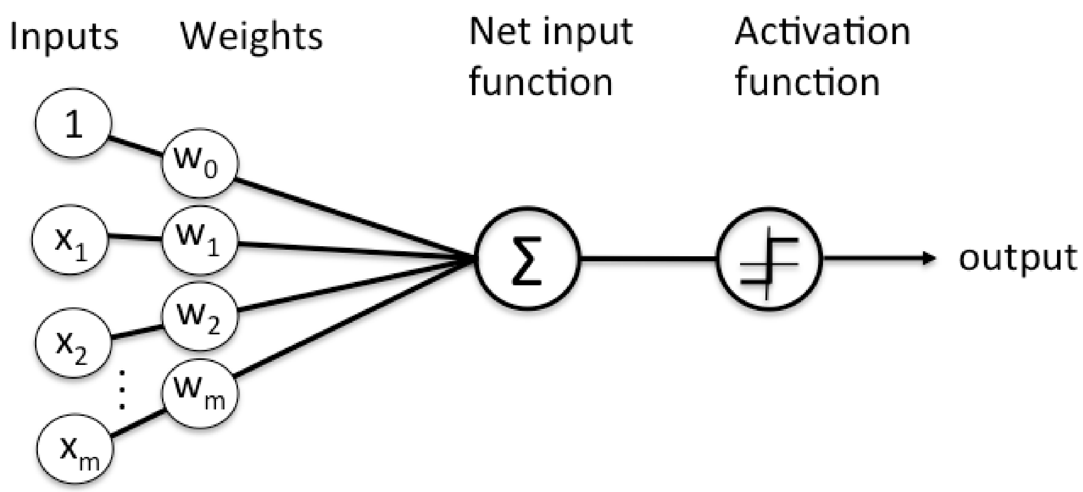


Figure 2.1: A single node, or neuron, producing an output, given multiple inputs. Reproduced from [1].

To better understand how NNs are capable of learning 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.

Chapter 3

Creating a language model

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.

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.

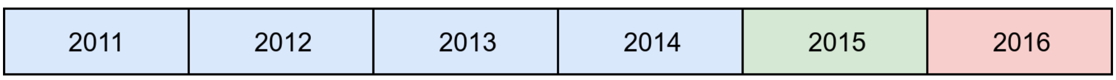


Figure 3.1: The split between training (blue), development (green) and test (red) sets.

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	152.128	199.633
Including singleton trigrams	36,648,350	150.533	197.427

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	117.544	175.55	35,573
Smoothed	104.939	154.264	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.

Chapter 4

Applying a neural network for 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 ensure the NN converges, these counts will have to be normalised to lie between 0 and 1. 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

Figure 4.1 shows the NN structure used. There are four input nodes, one for each input feature, two hidden layers, and one output node. 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.

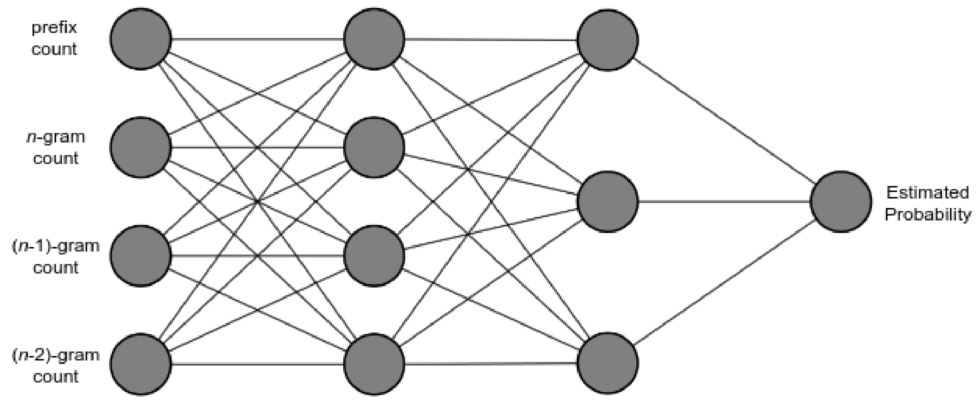
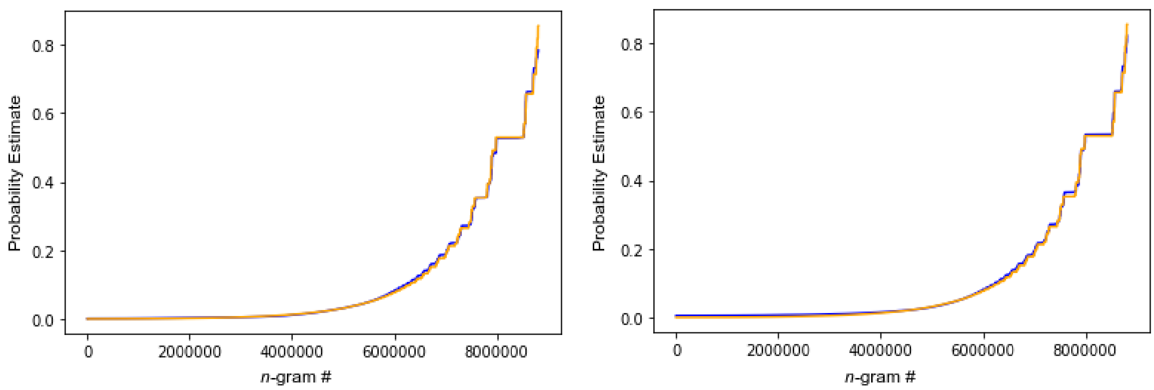


Figure 4.1: NN structure.

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 4.2 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 its probability estimate value.



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

Figure 4.2: 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.

Subsequently, 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 finalised our NN 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. Problem encountered

4.4.1. Initial results

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)	195.495	262.218	754,093
Good-Turing Smoothed (baseline)	152.128	199.633	0
Neural Network Smoothed	113.705	148.252	932,287

Testing our new LM on our development set results 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 4.3 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.

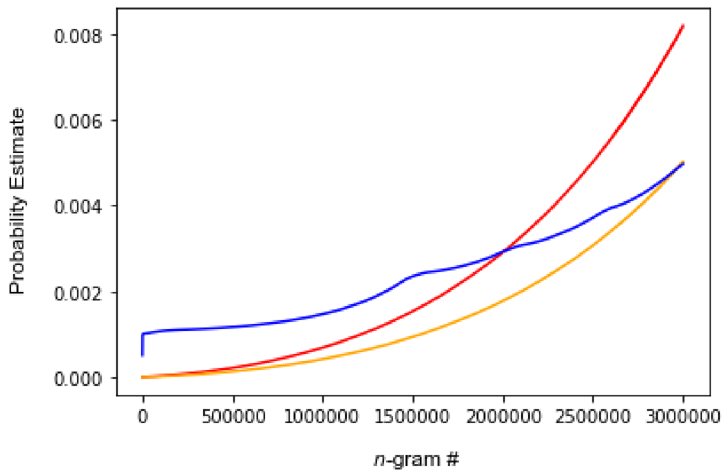


Figure 4.3: P_{MLE} (red), P_{GT} (yellow) and P_{NN} (blue) for the 3,000,000 lowest probability trigrams in the training set.

This poses a problem when SRILM re-normalises the ARPA file. 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 due to in 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. One option would be to redesign how the NN is implemented, for example by using multiple NNs, each responsible for its own unique output range between 0 and 1.

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.

4.4.2. Comparing various thresholds

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	155.693	204.568	118
$1.2 \times P_{MLE}(\cdot)$	$P_{MLE}(\cdot)$	18.53	153.438	201.445	6
$1.1 \times P_{MLE}(\cdot)$	$1.1 \times P_{MLE}(\cdot)$	19.48	154.28	202.611	13
$1.1 \times P_{MLE}(\cdot)$	$P_{MLE}(\cdot)$	19.48	153.388	201.377	3
$P_{MLE}(\cdot)$	$P_{MLE}(\cdot)$	20.56	153.373	201.355	0
$P_{MLE}(\cdot)$	$P_{GT}(\cdot)$	20.56	152.257	199.811	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)	117.544	175.55	35,573
Good-Turing (baseline)	104.939	154.264	0
Neural Network with Threshold	104.719	153.914	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.

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 only equal the performance of this smoothing approach and not improved 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 for inputs to our NN. The second half we use to find the ML probability estimate for that same given trigram and take it as target output for our NN.

These two inputs 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 he bigram $(w_j w_k)$ and unigram (w_i) both have high counts, but 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 in 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.

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.

The fundamental difference between the previous NN and this one is that the NN developed in Chapter 4 computes a discounted probability estimate, whereas the NN developed in this chapter, computes an ML probability estimate for a given trigram. Both NNs only calculate probabilities for trigrams occurring less than 8 times. The ML probability estimate produced by our 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 have the same probability. Therefore we have seven different discount factors, one for each number of occurring n-gram applicable 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)	131.463	196.398
Neural Network	131.538	196.519

In the following chapter, the results are analyzed in more depth, to better understand which aspects the NN is under-performing in, as well as the instances it out-performed the baseline.

Chapter 6

Analysing performance

Figure 6.1 visualises how a trigram is broken up into smaller n-grams. Each n-gram's count is used as input to the NN. We investigate in this chapter the relationship between these n-gram counts and how they affect certain characteristics of the NN. We draw example trigrams from our training set to illustrate this. SRILM calculates an MLE for a given trigram with only the prefix and trigram counts.

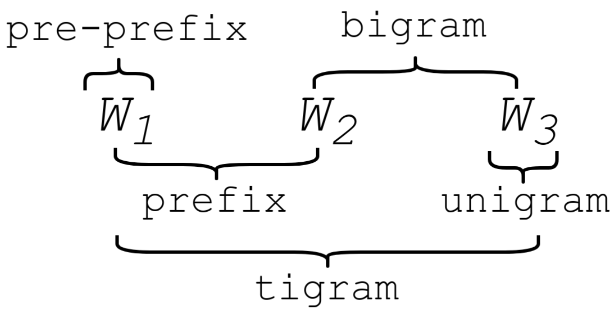


Figure 6.1: A trigram $w_1w_2w_3$ and its breakdown into smaller constituent components.

6.1. N-grams present in training set

The SRILM baseline (P_{MLE}) and NN calculated (P_{NN}) MLE values with the n-gram counts are listed in Table 6.1 for two example trigrams, both occurring exactly six times in the training set. These trigrams are thus subjected to smoothing and the NN probability is used to replace the existing MLE value. We selected these trigrams as the only two counts that differ are the unigram and bigram counts.

There are two aspects to the results in Table 6.1 that are of interest to us. Firstly the bigram count has a larger weight in the NN than the unigram count and results in the NN estimating a higher probability for the trigram "FOR TOP END", while the MLE SRILM calculates remains unchanged between the two trigrams.

Secondly, the NN estimates a smaller probability than the existing MLE for both trigrams. These two aspects are further assessed in the section that follows.

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
$C(w_3)$	95,973	25,932
P_{MLE}	0.0211	0.0211
P_{NN}	0.0153	0.0157

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. This threshold prevents excessive over-estimation of probabilities, however, it comes with a drawback. The freedom of the NN is effectively reduced, as it can no longer favour trigrams by only slightly exceeding the MLE. An example of this is given in Table 6.2.

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
$C(w_3)$	28,811	6,568
P_{MLE}	0.00384	0.00384
P_{NN}	0.00404	0.00409

Once again, we confirm that the NN calculates increased probabilities for trigrams encapsulating higher bigram counts. For the trigrams shown in Table 6.2, the NN probability will not be written to the ARPA file, as it exceeds the threshold. Instead, the MLE is used.

Next, we observe two trigrams that both have a substantially lower NN probability estimate, compared to the MLE. One being a grammatically incorrect trigram, which ideally should not be present in our training set, but in practise, it is seen four times during training.

Note the difference in bigram and prefix counts. We will be evaluating the significance of this on our output 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
$C(w_3)$	1,233,800	169,063
P_{MLE}	0.5	0.0652
P_{NN}	0.399	0.0532

It is unusual for a trigram to simultaneously have a high bigram count with a small prefix count and vice versa. This is indicative of the trigram occurring seldom for a particular reason. The second trigram does not have the same contrast in its bigram and prefix counts, however is discounted to the same extent. We therefore investigate these two sets of counts further in the section that follows.

6.2. NN characteristics

We now seek to first assess how the addition of the bigram, unigram and pre-prefix counts effect the probability the NN estimates. These counts are not utilised by SRILM when calculating the MLE for a given trigram. SRILM always produces the same MLE, 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. The bigram, unigram and pre-prefix counts are scaled, one at a time, until the output converges to deduce the impact each input has on the output probability. This is done for trigrams of two different MLE values as it influences the output tendencies significantly. These values are chosen close to the example trigrams in Table 6.3, aiming to provide context to those results. It was found that increasing the values for the additional counts, larger than those in Table 6.4, no longer had any considerable impact.

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

$P_{MLE}(w_3\|w_1, w_2)$	$C(w_1w_2)$	$C(w_1w_2w_3)$	$C(w_2w_3)$	$C(w_3)$	$C(w_1)$
0.5	8	4	8	1000	1000
0.06	50	3	50	1000	1000

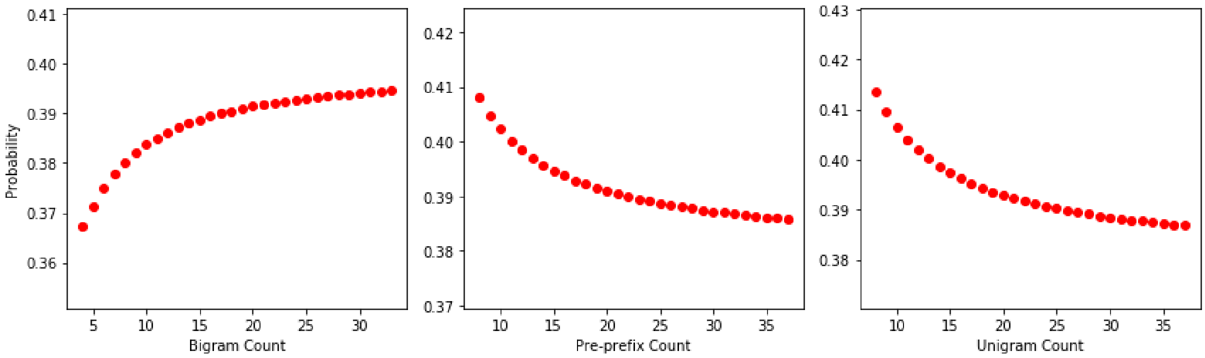


Figure 6.2: A simulated trigram with MLE = 0.5.

For a trigram with an MLE equal to 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. It is important to note that the majority of trigrams have large unigram and pre-prefix counts. This contributes to the NN probability estimate being smaller than the MLE SRILM calculates, for the first trigram in Table 6.3.

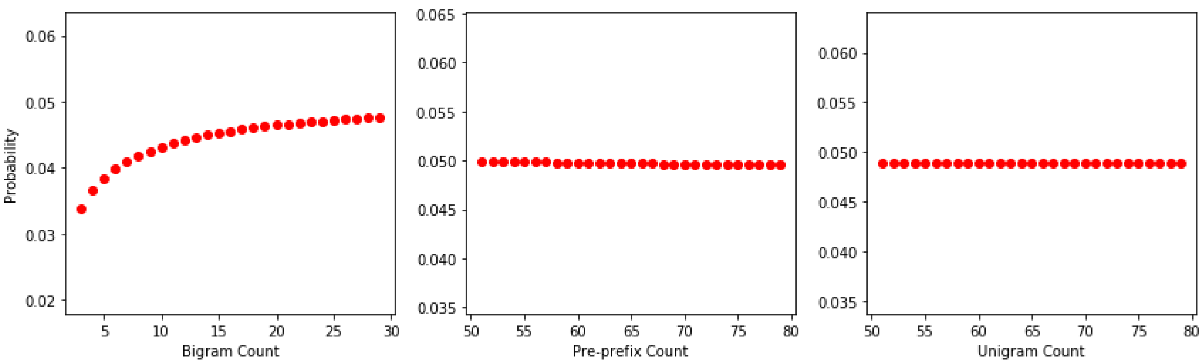


Figure 6.3: A simulated trigram with MLE = 0.06.

For the second simulated trigram, with MLE of 0.06, it is clear from Figure 6.3 that the bigram count is now the only input effecting the output probability. For the smallest possible bigram count, the NN reduced the output probability with 40% when compared to the output probability given larger bigram counts. We also conclude that generally the NN tends to estimate probabilities that are smaller than the MLE value, the extent to which dependant on the input counts.

For both simulated trigrams the output probability only increased with bigram count. However, beyond a certain value this became negligible. It is evident that the NN cannot appropriately reduce the probability of trigrams with a small prefix count and a large bigram count. These are part of the trigrams we want the NN to estimate a reduced probability for.

Focusing on the reverse situation we see positive results. If we, instead of increasing the bigram count, increase the prefix count, whilst keeping the bigram count equal to the smallest possible value, we see in Figure 6.4 that the NN reduces 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.

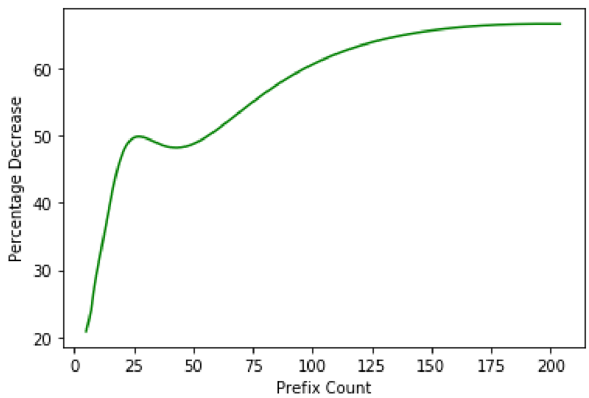


Figure 6.4: The percentage decreased probability estimated by the NN, with bigram count kept constant.

From the results of these simulated trigrams, we conclude that the NN has been learned to appropriately under-estimate certain trigrams, as well as estimate separate probabilities to distinguishing between two trigrams that have the same MLE. The NN provided a more dynamic probability estimate, sensitive to changes in the input counts encapsulated by the trigram, which SRILM is not capable of.

6.3. Effect on unseen trigrams

Due to the increase in number of trigram probability estimates that are now smaller than previously calculated by SRILM, 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 never greater than to what it was before. This results in more probability mass that is redistributed during backoff.

The NN in its current implementation is not responsible for re-estimating probabilities of unseen n-grams nor 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 calculated using 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 re-estimated by the NN, decreasing it. The NN showed positive results in decreasing the probability difference between these 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 $P(w_3\|w_1, w_2)$	Trigam 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 certain 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.

Chapter 7

Conclusion

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," in *Proceeding of Interspeech, Denver, Colorado*, 2002.

Appendix A

Project planning schedule

Appendix B

ECSA outcomes

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.

```python
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.

```python
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
```

```python
## 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.

```python
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))
```

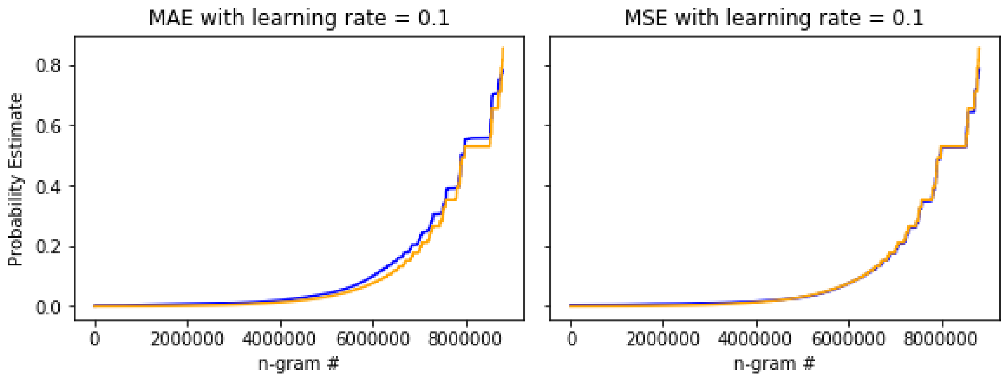
Figure D.5: Rewriting an ARPA file with NN calculated probability estimates.

```python
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()
```

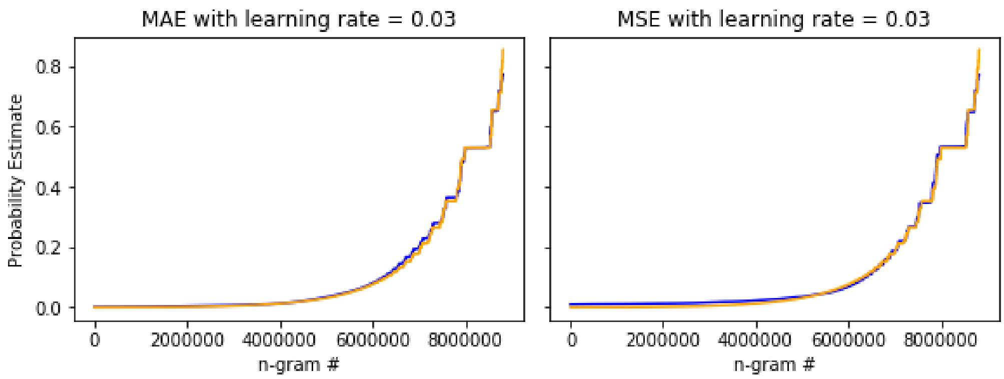
Figure D.6: Copy bigram probabilities from smoothed ARPA file to NN ARPA file.

Appendix E

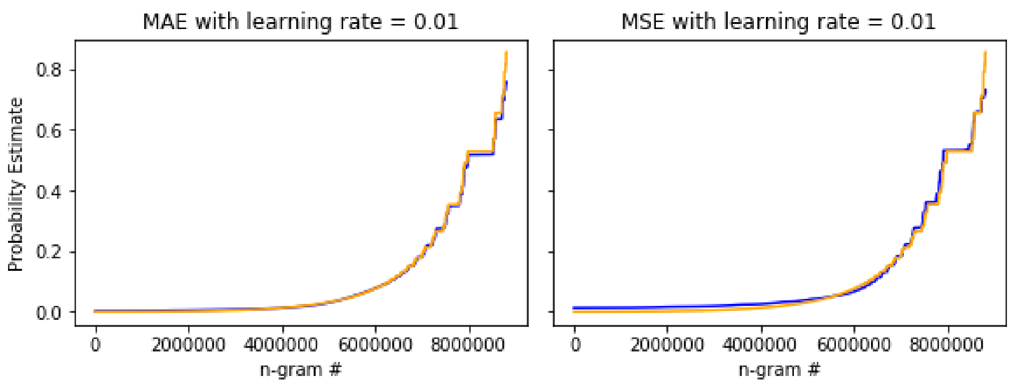
NN Training Results



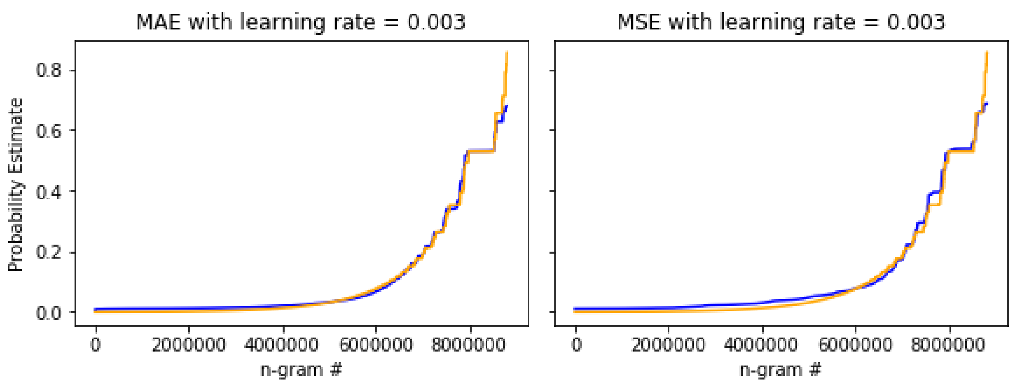
(a)



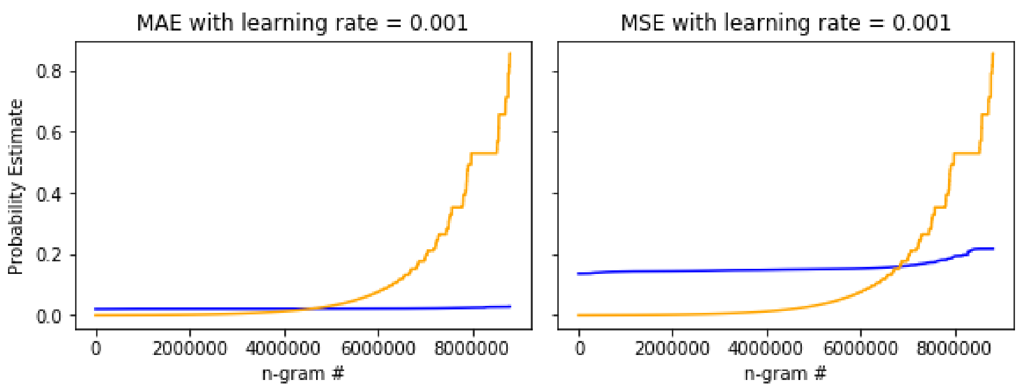
(b)



(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).