UNIVERSITY OF CAMBRIDGE

Devan Kuleindiren
Robinson College
`dk503`

Computer Science Tripos - Part II Project

Language Modelling for Text Prediction

March 30, 2017

supervised by
Dr Marek Rei & Dr Ekaterina Shutova

Proforma

Name:	**Devan Kuleindiren**
College:	**Robinson College**
Project Title:	**Language Modelling for Text Prediction**
Examination:	**Computer Science Tripos – Part II, June 2017**
Word Count:	**?**
Project Originator:	Devan Kuleindiren & Dr Marek Rei
Supervisors:	Dr Marek Rei & Dr Ekaterina Shutova

Original Aims of the Project

The primary aim of the project was to implement and benchmark a variety of language models, comparing the quality of their predictions as well as the time and space that they consume. More specifically, I aimed to build an n-gram language model along with several smoothing techniques, and a variety of recurrent neural network-based language models. An additional aim was to investigate ways to improve the performance of existing language models on error-prone text.

Work Completed

All of the project aims set out in the proposal have been met, resulting in a series of language model implementations and a generic benchmarking framework for comparing their performance. I have also proposed and evaluated a novel extension to an existing language model which improves its performance on error-prone text. Additionally, as an extension, I implemented a mobile keyboard on iOS that uses my language model implementations as a library.

Special Difficulties

None.

Declaration

I, Devan Kuleindiren of Robinson College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

List of Figures

Chapter 1

Introduction

My project investigates the performance of various language models in the context of text prediction. I started by implementing a series of well-established models and comparing their performance, before assessing the tradeoffs that occur when you attempt to apply them in a practical context, such as in a mobile keyboard. Finally, I proposed a novel extension to an existing model which aims to improve its performance on error-prone text.

1.1 Language Models

Language models (LMs) produce a probability distribution over a sequence of words, which can be used to estimate the relative likelihood of words or phrases occurring in various contexts. This predictive power is useful in a variety of applications. For example, in speech recognition, if the speech recogniser has estimated two candidate word sequences from an acoustic signal; *'it's not easy to wreck a nice beach'* and *'it's not easy to recognise speech'*, then a language model can be used to determine that the second candidate is more probable than the first. Language models are also used in machine translation, handwriting recognition, part-of-speech tagging and information retrieval.

$$\begin{array}{rll}
\overbrace{\textit{Do you want to grab a}}^{w_1^k} & \overbrace{\underline{\textit{drink}}}^{w_{k+1}} & \overbrace{(0.327)}^{\mathbb{P}(w_{k+1}|w_1^k)} \\
 & \textit{coffee} & (0.211) \\
 & \textit{bite} & (0.190) \\
 & \textit{spot} & (0.084) \\
 & \vdots & \vdots
\end{array}$$

My project focuses on language modelling in the context of text prediction. That is, given a sequence of words $w_1w_2...w_k = w_1^k$, I want to estimate $\mathbb{P}(w_{k+1}|w_1^k)$. For instance, if a user has typed *'do you want to grab a '*, then a language model could be used to suggest probable next words such as *'coffee'*, *'drink'* or *'bite'*, and these predictions could further be narrowed down as the user continues typing.

1.2 Motivation

Benchmarking

Language models are central to a wide range of applications, but there are so many different ways of implementing them. Before using a language model in a particular context, it is important to understand how it performs in comparison to other methods available. In this project I focused in depth on the two most prominent types of language model: n-gram models and recurrent neural network-based models. n-gram models are typically coupled with smoothing techniques, which are explained in section 2.1. I investigated 5 different smoothing techniques and 3 different recurrent neural network architectures on a variety of datasets.

Error-prone Text

One problem with existing language models is that their next-word predictions tend to be less accurate when they are presented with error-prone text. This is not surprising, because they are only ever trained on sentences that do not contain any errors. Unfortunately, the assumption that humans will not make any mistakes when typing text is almost never valid. For this reason, I also investigated ways to narrow the gap in performance between language model predictions on error-prone text and language model predictions on error-free text.

1.3 Related Work

Chelba et al. [1] from Google explore the performance of a variety of language models on a huge, one billion word dataset. Their work presents the limits of language modelling, when vast quantities of data and computational resources are available. Chen and Goodman [2] compare the performance of a series of smoothing techniques for n-gram models, and later use their results to propose an extension to Kneser-Ney smoothing [3] which is implemented in this project.

In recent years, there have been joint efforts from Ng et al. at CoNLL to improve and compare the performance of grammatical error correction [4] [5]. Language modelling on error-prone text, however, has an important distinction: A test time, a language model cannot use words ahead of its current position to make predictions. In other words, if a language model is predicting the probability of the next word in the middle of an error-prone sentence, then it can only use the first half of the sentence for making corrections and predictions.

Chapter 2

Preparation

My preparation consisted of thoroughly understanding n-gram and RNN-based language models, as well as planning how to tie them all together in an efficient implementation.

2.1 n-gram Models

This section describes n-gram language models and the various smoothing techniques implemented in this project.

2.1.1 An Overview of n-gram Models

Language models are concerned with the task of computing $\mathbb{P}(w_1^N)$, the probability of a sequence of words $w_1 w_2 ... w_N = w_1^N$, where $w_i \in V$ and V is some predefined vocabulary[1]. By repeated application of the product rule, it follows that:

$$\mathbb{P}(w_1^N) = \prod_{i=1}^{N} \mathbb{P}(w_i | w_1^{i-1})$$

n-gram language models make the Markov assumption that w_i only depends on the previous $(n-1)$ words. That is, $\mathbb{P}(w_i|w_1^{i-1}) \approx \mathbb{P}(w_i|w_{i-n+1}^{i-1})$:

$$\mathbb{P}(w_1^N) \approx \prod_{i=1}^{N} \mathbb{P}(w_i | w_{i-n+1}^{i-1})$$

Using the maximum likelihood estimation, $\mathbb{P}(w_i|w_{i-n+1}^{i-1})$ can be estimated as follows:

$$\mathbb{P}(w_i|w_{i-n+1}^{i-1})_{MLE} = \frac{c(w_{i-n+1}^{i})}{\sum_w c(w_{i-n+1}^{i-1} w)}$$

where $c(W)$ denotes the number of times that the word sequence W was seen in the training set.

To a first approximation, the aforementioned n-gram language model provides reasonable results and is simple to compute. However, it does have one major issue: if, as an example, a 3-gram (trigram) model does not encounter the trigram *'the cat sat'* in the data

[1] The vocabulary is typically taken as all of the words that occur at least k times in the training set. k is typically around 2 or 3.

it is trained upon, then it will assign a probability of 0 to that word sequence. This is problematic, because *'the cat sat'* and many other plausible sequences of words might not occur in the training data. In fact, there are $|V|^n$ possible n-grams for a language model with vocabulary V, which is exponential in the value of n. This means that as the value of n is increased, the chances of encountering a given n-gram in the training data becomes exponentially less likely.

One way to get around this problem is to exponentially increase the size of the training set. This does, however, require significantly more memory and computation, and assumes that additional training data is available in the first place. An alternative solution is to adopt a technique called *smoothing*. The idea behind smoothing is to 'smooth' the probability distribution over the words in the vocabulary such that rare or unseen n-grams are given a non-zero probability. There are a variety of methods that achieve this. The ones which I have implemented are described in the next section.

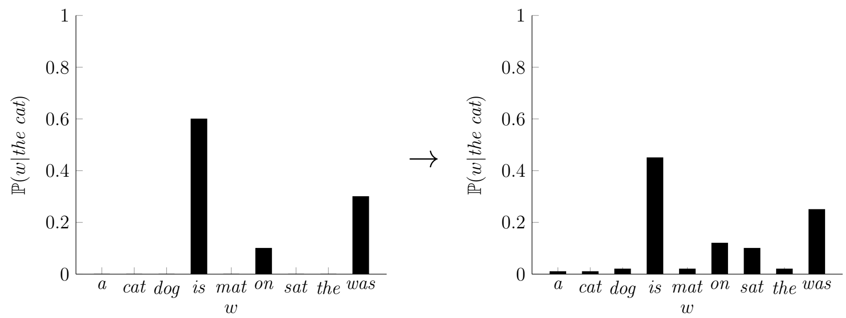


Figure 2.1: A toy example of smoothing. The probabilities of words that frequently follow *'the cat'* are distributed to other less frequently occurring words in the vocabulary. The vocabulary is $V = \{a, cat, dog, is, mat, on, sat, the, was\}$.

2.1.2 Smoothing Techniques

Add-One Smoothing

Add-one smoothing [6] simply involves adding 1 to each of the n-gram counts, and dividing by $|V|$ to ensure the probabilities sum to 1:

$$\mathbb{P}(w_i|w_{i-n+1}^{i-1})_{\text{ADD-ONE}} = \frac{c(w_{i-n+1}^{i}) + 1}{\sum_w c(w_{i-n+1}^{i-1}w) + |V|}$$

One issue with add-one smoothing is that it gives an equal amount of probability to all n-grams, regardless of how likely they actually are. As an example, if both *'the cat'* and *'pizza cat'* are unseen in the training data of a bigram model, then $\mathbb{P}(cat \mid the)_{\text{ADD-ONE}} = \mathbb{P}(cat \mid pizza)_{\text{ADD-ONE}}$, despite the fact that *'the'* is much more likely to precede *'cat'* than *'pizza'*. This problem can be reduced by employing *backoff*, a technique whereby you recurse on the probability calculated by the $(n-1)$-gram model. In this case, it is likely that $\mathbb{P}(the)_{\text{ADD-ONE}} > \mathbb{P}(pizza)_{\text{ADD-ONE}}$, which could be used to deduce that *'the cat'* is more likely than *'pizza cat'*.

Absolute Discounting

Absolute discounting employs backoff by interpolating higher and lower order n-gram models. It does this by subtracting a fixed discount $0 \leq D \leq 1$ from each non-zero count:

$$\begin{aligned}\mathbb{P}(w_i|w_{i-n+1}^{i-1})_{\text{ABS}} = & \frac{max\{c(w_{i-n+1}^{i}) - D, 0\}}{\sum_w c(w_{i-n+1}^{i-1}w)} \\ & + \frac{D}{\sum_w c(w_{i-n+1}^{i-1}w)} N_{1+}(w_{i-n+1}^{i-1}\bullet)\mathbb{P}(w_i|w_{i-n+2}^{i-1})_{\text{ABS}}\end{aligned}$$

where

$$N_{1+}(w_{i-n+1}^{i-1}\bullet) = |\{w \mid c(w_{i-n+1}^{i-1}w) \geq 1\}|$$

and the base case of recursion $\mathbb{P}(w)_{\text{ABS}}$ is given by the maximum likelihood unigram model. $N_{1+}(w_{i-n+1}^{i-1}\bullet)$ is the number of unique words that follow the sequence w_{i-n+1}^{i-1}, which is the number of n-grams that D is subtracted from. It is not difficult to show that the coefficient attached to the $\mathbb{P}(w_i|w_{i-n+2}^{i-1})_{\text{ABS}}$ term ensures that the probabilities sum to 1.

Ney, Essen and Kneser [7] suggested setting D to the value:

$$D = \frac{n_1}{n_1 + 2n_2} \tag{2.1}$$

where n_1 and n_2 are the total number of n-grams with 1 and 2 counts respectively.

Kneser-Ney Smoothing

Kneser and Ney proposed an extension to absolute discounting which takes into account the number of unique words that precede a given n-gram [8]. As a motivating example, consider the bigram *'bottle cap'*. If *'bottle cap'* has never been seen in the training data, then the absolute discounting model would backoff onto the unigram distribution for *'cap'*. Using the unigram distribution, *'Francisco'* might be given a higher probability than *'cap'* (assuming *'Francisco'* occurs more frequently than *'cap'*). This would result in the bigram *'bottle Francisco'* being given a higher probability than *'bottle cap'*. Clearly, this is undesirable, because *'Francisco'* only ever follows *'San'*.

From this example, it seems intuitive to assign more probability to those n-grams that follow a larger number of unique words. Kneser and Ney encapsulate this intuition by replacing some of the absolute counts $c(w_i^j)$ with the number of unique words that precede the word sequence w_i^j, $N_{1+}(\bullet w_i^j)$:

$$N_{1+}(\bullet w_i^j) = |\{w \mid c(w_i^j w) \geq 1\}|$$

Kneser-Ney smoothing[2] is defined as follows:

$$\begin{aligned}\mathbb{P}(w_i|w_{i-n+1}^{i-1})_{\text{KN}} = & \frac{max\{\gamma(w_{i-n+1}^{i}) - D, 0\}}{\sum_w \gamma(w_{i-n+1}^{i-1}w)} \\ & + \frac{D}{\sum_w \gamma(w_{i-n+1}^{i-1}w)} N_{1+}(w_{i-n+1}^{i-1}\bullet)\mathbb{P}(w_i|w_{i-n+2}^{i-1})_{\text{KN}}\end{aligned}$$

[2] This is actually the interpolated version of Kneser-Ney smoothing, which differs slightly in form to the equation presented in the original paper.

where

$$\gamma(w_{i-k+1}^i) = \begin{cases} c(w_{i-k+1}^i) & \text{for the outermost level of recursion (i.e. } k = n) \\ N_{1+}(\bullet w_{i-k+1}^i) & \text{otherwise} \end{cases} \tag{2.2}$$

and the unigram probability is given as:

$$\mathbb{P}(w_i)_{\text{KN}} = \frac{N_{1+}(\bullet w_i)}{\sum_w N_{1+}(\bullet w)}$$

Modified Kneser-Ney Smoothing

Chen and Goodman experimented with different discount values D in Kneser-Ney smoothing and noticed that the ideal average discount value for n-grams with one or two counts is substantially different from the ideal average discount for n-grams with higher counts. Upon this discovery, they introduced a modified version of Kneser-Ney smoothing [3]:

$$\mathbb{P}(w_i|w_{i-k+1}^{i-1})_{\text{MKN}} = \frac{max\{\gamma(w_{i-k+1}^i) - D(c(w_{i-k+1}^i), 0\}}{\sum_w \gamma(w_{i-k+1}^{i-1} w)} + \lambda(w_{i-k+1}^{i-1})\mathbb{P}(w_i|w_{i-k+2}^{i-1})_{\text{MKN}}$$

where γ is defined in equation (2.2) ***TODO:*** Keep this number up to date., and λ is defined as:

$$\lambda(w_{i-k+1}^{i-1}) = \frac{D_1 N_1(w_{i-k+1}^{i-1}\bullet) + D_2 N_2(w_{i-k+1}^{i-1}\bullet) + D_{3+} N_{3+}(w_{i-k+1}^{i-1}\bullet)}{\sum_w \gamma(w_{i-k+1}^{i-1} w)}$$

and

$$D(c) = \begin{cases} 0 & \text{if } c = 0 \\ D_1 & \text{if } c = 1 \\ D_2 & \text{if } c = 2 \\ D_{3+} & \text{if } c \geq 3 \end{cases}$$

Chen and Goodman suggest setting the discounts to be:

$$D_1 = 1 - 2D\frac{n_2}{n_1} \qquad D_2 = 2 - 3D\frac{n_3}{n_2} \qquad D_{3+} = 3 - 4D\frac{n_4}{n_3}$$

where D is as defined in equation (2.1). ***TODO:*** Keep this number up to date.

Katz Smoothing

Katz smoothing is a popular smoothing technique based on the Good-Turing estimate [9]. The Good-Turing estimate states an n-gram that occurs r times should be treated as occurring r^* times, where:

$$r^* = (r+1)\frac{n_{r+1}}{n_r}$$

where n_r is the number of n-grams that occur r times. Converting this count into a probability simply involves normalising as follows:

$$\mathbb{P}(w_i|w_{i-n+1}^{i-1})_{\text{GT}} = \frac{c^*(w_{i-n+1}^i)}{\sum_{r=0}^{\infty} n_r r^*} \tag{2.3}$$

Katz smoothing [10] is then defined as:

$$\mathbb{P}(w_i|w_{i-n+1}^{i-1})_{\text{KATZ}} = \begin{cases} \mathbb{P}(w_i|w_{i-n+1}^{i-1})_{\text{GT}} & \text{if } c(w_{i-n+1}^{i}) > 0 \\ \alpha(w_{i-n+1}^{i-1})\mathbb{P}(w_i|w_{i-n+2}^{i-1})_{\text{KATZ}} & \text{otherwise} \end{cases} \tag{2.4}$$

where

$$\alpha(w_{i-n+1}^{i-1}) = \frac{1 - \sum_{\{w_i \ | \ c(w_{i-n+1}^{i})>0\}} \mathbb{P}(w_i|w_{i-n+1}^{i-1})_{\text{KATZ}}}{1 - \sum_{\{w_i \ | \ c(w_{i-n+1}^{i})>0\}} \mathbb{P}(w_i|w_{i-n+2}^{i-1})_{\text{KATZ}}}$$

In practice, the infinite sum in equation (2.3) cannot be computed. To get around this issue, Katz takes n-gram counts above some threshold k as reliable and only applies the Good-Turing estimate to those with a count less than or equal to k. Katz suggests $k = 5$. This modification requires a slightly different equation to (2.4) and is presented in Katz's original paper.

2.2 Recurrent Neural Network Models

In this section I give a brief introduction to neural networks, recurrent neural networks (RNNs) and how RNNs can be used in the context of language modelling. A thorough description of the more complex RNN architectures, gated recurrent units (GRUs) and long short-term memory (LSTM), is given in chapter 3.

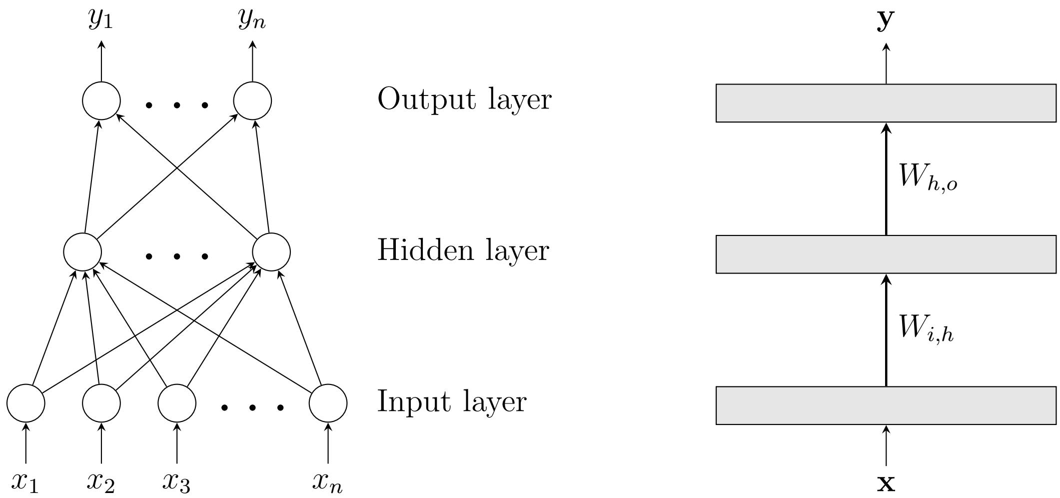
2.2.1 An Overview of Neural Networks

The human brain is a furiously complicated organ, packed with a network of approximately 86 billion neurons[3] that propagate electrochemical signals across connections called synapses. Artificial neural networks, or neural networks, were originally developed as a mathematical model of the brain [12], which despite being substantially oversimplified, now provides an effective tool for classification and regression in modern-day machine learning.

Neural networks consist of a series of nodes which are joined by directed and weighted connections. Inputs are supplied to some subset of the nodes and then propagated along the weighted connections until they reach the designated output nodes. In the context of the brain, the nodes represent neurons, the weighted connections represent synapses and the flow of information represents electrochemical signals.

An important distinction to be made is whether the neural network is cyclic or not. Acyclic neural networks are called feed-forward neural networks (FNNs), whereas cyclic neural networks are denoted recurrent neural networks (RNNs) which are covered in section 2.2.2. There are a variety of FNNs, but the most prominent is the multilayer perceptron (MLP) [13], which I will outline below.

[3]According to a study by Azevedo et al. [11].



(a) An example multilayer perceptron. (b) A simplified notation of (a).

Figure 2.2: Two alternative notations for the same network: a multilayer perceptron with one hidden layer. In the simplified notation, the grey rectangle represents a vector of neurons, the thin arrow represents a vector and the thick arrow represents a matrix of weights.

The Multilayer Perceptron

The multilayer perceptron consists of layers of neurons. The first layer is the input layer, the last is the output layer and any layers in between are called *hidden layers*. If there is more than one hidden layer then it is called a *deep* neural network. The input neurons simply pass on the input values that they are given. The neurons in the subsequent layers of the network compute the weighted sum of their inputs, before passing that value through an *activation function* and outputting it:

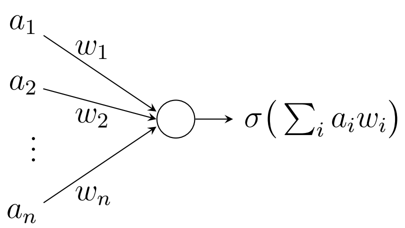
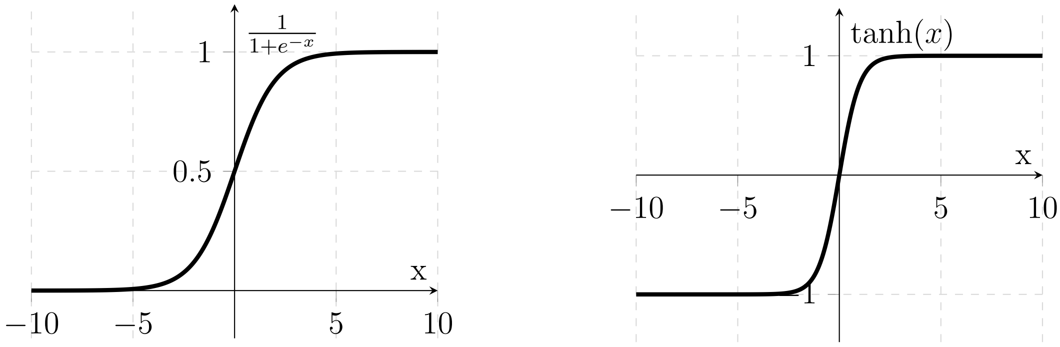


Figure 2.3: The computation carried out by each neuron. The inputs are denoted a_i, the weights on the input connections are w_i and the activation function is σ.

It is worth noting that activation functions should be chosen to be non-linear. Any combination of linear operators is linear, which means that any linear MLP with multiple hidden layers is equivalent to an MLP with a single hidden layer. Non-linear neural networks, on the other hand, are more powerful, and can gain considerable performance by adding successive hidden layers to re-represent the input data at higher levels of abstraction [14] [15]. In fact, is has been shown that a non-linear MLP with a single hidden layer containing a sufficient number of neurons can approximate any continuous function on a compact input domain to arbitrary precision [16].

Frequently used activation functions include the sigmoid and the hyperbolic tangent functions. These are both non-linear functions squashed within the ranges $(0, 1)$ and $(-1, 1)$ respectively. Their steep slope at the origin is supposed to mimic an axon, which fires its output after the input reaches a certain potential. More importantly, these functions are differentiable, which allows for the network to be trained using gradient descent. This is discussed below.



(a) The sigmoid function.

(b) The hyperbolic tangent function.

Figure 2.4: Two popular activation functions.

Backpropagation and Gradient Descent

FNNs compute a function, f, parameterised by the weights $\mathbf{w}$ of the network, mapping an input vector $\mathbf{x} = (x_1, ..., x_n)^T$ to an output vector $\mathbf{y} = (y_1, ..., y_m)^T$.

$$\mathbf{y} = f_{\mathbf{w}}(\mathbf{x})$$

The entire point of training a neural network is to get it to learn a particular mapping from input vectors to output vectors. What the inputs and outputs represent depends on the problem at hand. For example, in the context of classifying pictures of animals, the input vector might represent the pixel values of an image and the output vector might represent a probability distribution over the set of animals in the classification task.

In order to train a neural network, you supply it with a training set, which is just a list of input-target pairs:

$$\mathbf{s} = ((\mathbf{x}_1, \mathbf{t}_1), ..., (\mathbf{x}_N, \mathbf{t}_N))$$

Where $\mathbf{t}_i$ is the vector that the network should output given the example input $\mathbf{x}_i$. A differentiable loss function, $\mathcal{L}(\mathbf{y}, \mathbf{t})$, is also defined, which essentially says how badly the network output $\mathbf{y}$ matches the target output $\mathbf{t}$. Then, a measure of how much error the network produces over the whole training set can be defined as follows:

$$\mathcal{L}_{\text{TOTAL}} = \sum_{i=1}^{N} \mathcal{L}(f_{\mathbf{w}}(\mathbf{x}_i), \mathbf{t}_i)$$

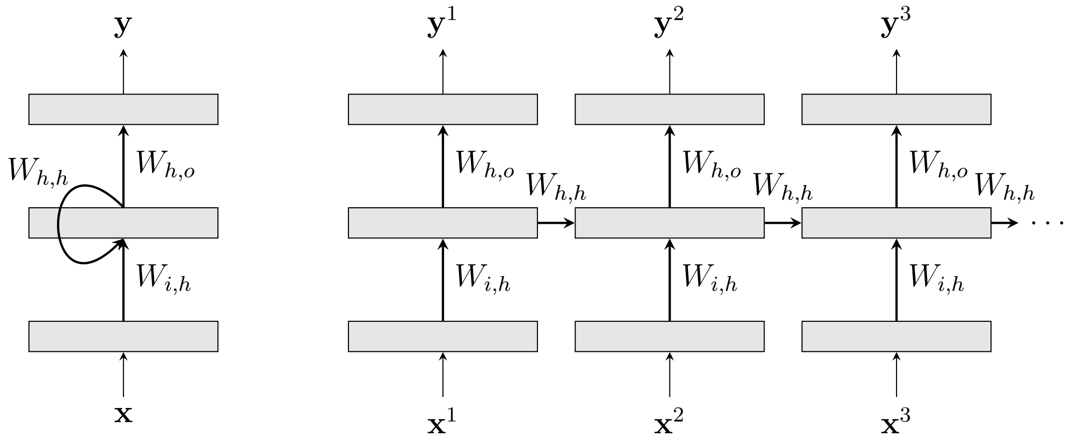
The end product of the *backpropagation* algorithm is the partial derivative:

$$\frac{\partial}{\partial w_{i,j}}(\mathcal{L}_{\text{TOTAL}})$$

for every weight $w_{i,j}$ in the neural network. *Gradient descent* is an optimisation technique that uses these derivatives to adjust the weights such that the loss is minimised over the training set.

2.2.2 Recurrent Neural Networks

In the context of language modelling, the input is a sequence of words and the output is also a sequence of words. The problem with FNNs is that these sequences may have a varying number of words in them, yet FNNs have fixed input and output vector sizes. One way to get around this problem is to look at a finite window of words at any given time, and encode that window into a vector, but even for modest window sizes this can lead to a huge number of weights in the network, which becomes more difficult to train. A better solution is to use a recurrent neural network.



(a) An RNN.

(b) An unrolled RNN.

Figure 2.5: Two different representations of the same RNN.

A recurrent neural network can be constructed from an FNN by adding connections from each neuron in a hidden layer back to every neuron in that layer, including itself. This gives something like the network shown in figure 2.5(a). An easier way to visualise an RNN is by *unrolling* it as shown in figure 2.5(b). By unravelling the RNN as shown, it is easy to see that can effectively represent a neural network with an infinite sequence of vectors as inputs and an infinite sequence of vectors as outputs. The notation $\mathbf{x}^t$ and $\mathbf{y}^t$ is used to denote the input vector and output vector at time step t.

- How RNNs can (in principle) handle infinite sequences.
- BPTT
- Various architectures.

A diagram and brief description of each will suffice here:

Vanilla Recurrent Neural Networks

Gated Recurrent Unit

Long Short-Term Memory

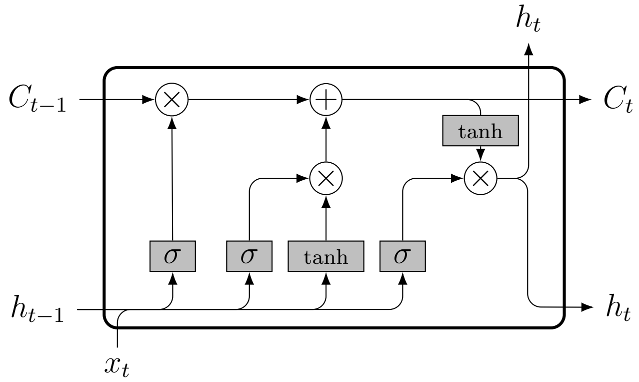


Figure 2.6: The structure of a basic LSTM cell.

2.2.3 Word Embeddings

- Vector space model
- Learnt during backpropagation

Other general points to make about language models:

- LMs have a vocabulary V.
- How to compute vocabulary (minimum frequency, etc)
- How to handle OOV words
- How to handle sentence boundaries

2.3 Software Engineering

2.3.1 Starting Point

- All language models written from scratch.

2.3.2 Requirements

- Project proposal goals.

2.3.3 Tools and Technologies Used

- Version Control
  - Git
  - GitHub
- Testing
  - Google Test for C++
  -
- Build Tools - Bazel
- Serialisation - Google Protocol Buffers
- Machine Learning - TensorFlow, NLTK, SLURM (open source job scheduler used by the HPCS)

2.3.4 TensorFlow

Perhaps this could go in the implementation section.

Chapter 3

Implementation

3.1 Development Strategy

3.1.1 Version Control and Build Tools

3.1.2 Testing Strategy

3.2 System Overview

3.2.1 Interface to Language Models

3.3 n-gram Models

3.3.1 Counting n-grams Efficiently

3.3.2 Precomputing Smoothing Coefficients

3.4 Recurrent Neural Network Models

3.4.1 Long Short-Term Memory

3.4.2 Gated Recurrent Units

3.4.3 Word Embeddings

Show some cool embedding plots with PCA.

3.4.4 Parameter Tuning

Dropout, embedding, learning rate decay, momentum?, gradient clipping

3.5 Extending Models to Tackle Error-Prone Text

3.5.1 Preprocessing the CLC Dataset

3.5.2 Error Correction on Word Context

3.6 Mobile Keyboard

3.6.1 Updating Language Model Predictions On the Fly

Chapter 4

Evaluation

In this chapter, I will first describe the benchmarking framework that I built to evaluate the language models, before proceeding onto the results. The results section is threefold: firstly I will present the performance of the existing language models that I implemented, secondly I will focus on the tradeoffs faced when employing those models on a mobile device, and finally I will display my findings in language modelling on error-prone text.
TODO: Include these too:

- Examples of generated sentences.
- Embedding visualisations - or perhaps put this into the implementation?
- Visualisations of the neuron activations for each word in a sentence.
- The number of parameters included in the LM in the PTB table (like Google did).
- The training time of each LM.

4.1 Evaluation Methodology

4.1.1 Metrics

In the context of text prediction, there are essentially two questions one might want to answer when evaluating a language model:

1. How accurately does the language model predict text?
2. How much resource, such as CPU or memory, does the language model consume?

In order to answer these questions, I implemented a generic benchmarking framework that can return a series of metrics that fall into one of the two aforementioned categories when given a language model. These metrics include perplexity, average-keys-saved, memory usage and average inference time. The first two are concerned with the accuracy of language models and the latter two relate to the resource usage. I also recorded how long it took to train each model.

Perplexity

Perplexity is the most widely-used metric for language models, and is therefore an essential one to include so that my results can be compared with those of other authors. Given a sequence of words $w_1^N = w_1 w_2 ... w_N$ as test data, the perplexity PP of a language model L is defined as:

$$\mathrm{PP}_L(w_1^N) = \sqrt[N]{\frac{1}{\mathbb{P}_L(w_1^N)}} = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{\mathbb{P}_L(w_i|w_1^{i-1})}} \tag{4.1}$$

where $\mathbb{P}_L(w_i|w_1^{i-1})$ is the probability computed by the language model L of the word w_i following the words w_1^{i-1}. The key point is that **lower values of perplexity indicate better prediction accuracy** for language models trained on a particular training set.

This somewhat arbitrary-looking formulation can be better understood from a touch of information theory. In information theory, the cross-entropy $H(p, q)$ between a true probability distribution p and an estimate of that distribution q is defined as:[1]

$$H(p, q) = -\sum_x p(x) \log_2 q(x)$$

It can be shown that $H(p, q) = H(p) + D_{KL}(p||q)$ where $D_{KL}(p||q) \geq 0$ is the Kullback-Leibler distance between p and q. Generally speaking, the better an estimate q is of p, the lower $H(p, q)$ will be, with a lower bound of $H(p)$, the entropy of p.

The perplexity PP of a model q, with respect to the true distribution p it is attempting to estimate, is defined as:

$$\mathrm{PP} = 2^{H(p,q)} \tag{4.2}$$

Language models assign probability distributions over sequences of words, and so it seems reasonable to use perplexity as a motivation for a measure of their performance. In the context of language modelling, however, we do not know what the underlying distribution of p is, so it is approximated with Monte Carlo estimation by taking samples of p (i.e. sequences of words from the test data) as follows:

$$\mathrm{PP}_L(w_1^N) = 2^{-\frac{1}{N}\sum_i \log_2 \mathbb{P}_L(w_i|w_1^{i-1})}$$

With a little algebra, this can be rearranged to give equation (4.1).

One issue with perplexity is that it is undefined if $\mathbb{P}_L(w_i|w_1^{i-1})$ is 0 at any point. To get around this in my implementation, I replaced probability values of 0 with the small constant `1e-9`. Results that use this approximation are marked.

Average-Keys-Saved

It is typical for the top three next-word predictions to be displayed and updated as the user types in a mobile keyboard, as described in section ***TODO:*** XXX. Clearly, it is in the interest of the mobile keyboard developer to minimise the amount of typing a user

[1]Note that $H(p, q)$ is often also used to denote the joint entropy of p and q, which is a different concept.

has to do before the correct prediction is displayed. Average-keys-saved is based on this incentive, and is defined as the number of keys that the user would be saved from typing as a result of the correct next word appearing in the top three predictions, averaged over the number of characters in the test data.

As an example, if the user is typing `science` and the word `science` appears in the top three predictions after they have typed `sc`, then that would count as 5 characters being saved, averaging at $\frac{5}{7}$ keys saved per character. Averaging over the number of characters in the test data ensures that the results are not biased by the data containing particularly long words, which are easier to save characters on.

Memory Usage

This is measured as the amount of physical memory in megabytes occupied by the process in which the language model under test is instantiated.

Training Time

The amount of time it took to train the language model.

Average Inference Time

This is measured as the amount of time in milliseconds that the language model takes to assign a probability to all of the words in its vocabulary given a sequence of words, averaged over a large number of sequences.

4.1.2 Datasets

I used three datasets throughout the evaluation of my project:

Penn Tree Bank (PTB) Dataset

The Penn Tree Bank is a popular dataset for measuring the quality of language models, created from text from the Wall Street Journal. It has already been preprocessed such that numbers are replaced with `N`, rare words are replaced with `<unk>` and the text has been split up into one sentence per line. It has been split up into a training, validation and test set. The training set has 10,000 unique words and 887,521 words overall.

Given that the Penn Tree Bank is so widely adopted, I used it for all tests in which the size of the training data is fixed, and will refer to it as PTB.

One Billion Word (1BW) Benchmark

This is a much larger dataset produced by Google of approximately 1 billion words [1]. I used this dataset for tests in which the size of the training data is a variable under investigation, and will refer to it as 1BW.

Cambridge Learner Corpus (CLC)

The Cambridge Learner Corpus is a dataset of 1,244 exam scripts written by candidates sitting the Cambridge ESOL First Certificate in English (FCE) examination in 2000 and 2001 [17]. The original dataset contains the scripts annotated with corrections to all of the mistakes by the candidates. In this project I make use of a preprocessed version of the dataset, in which there is one file containing the error-free version of the exam scripts and there is another file containing the original exam scripts with their errors. These two files are aligned line by line. I used this dataset when exploring the performance of language models on error-prone text, and will refer to it as CLC.

4.2 Results

4.2.1 Existing Models

Smoothing techniques and the value of n in n-gram models

The first set of language models that I built were n-gram models, along with a series of smoothing techniques for improving their predictions on less frequent n-grams.

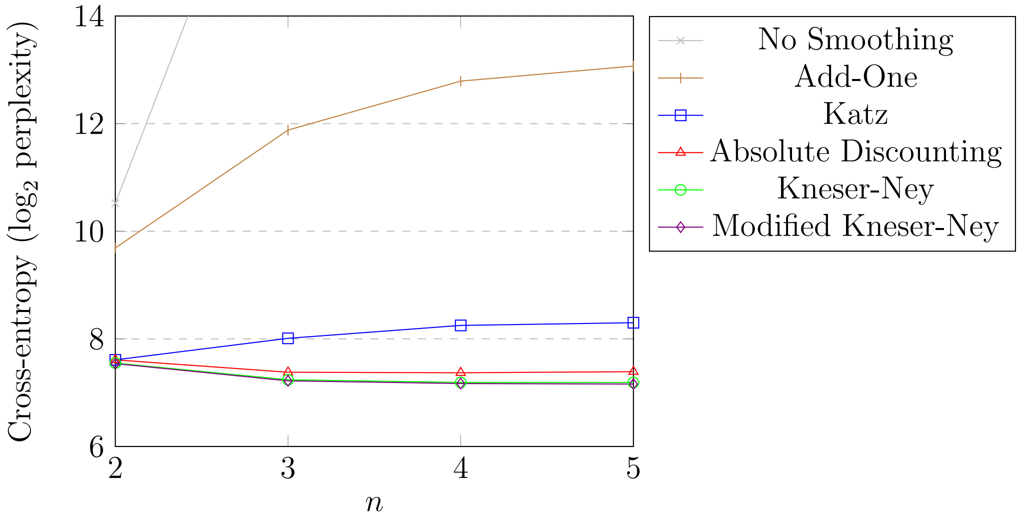


Figure 4.1: Cross-entropy of n-gram models trained on the PTB dataset.

Recall from equation (4.2) that cross-entropy is just the binary logarithm of perplexity, and that lower perplexity scores indicate better prediction accuracy. With this in mind, it is clear that modified Kneser-Ney smoothing offers the best prediction performance amongst the n-gram models.

The change in performance with the value of n is interesting. Intuitively, one might expect that increasing n will always yield better results, because this corresponds to increasing the number of words you use to make a prediction. However, for n-gram models with no smoothing, add-one smoothing or Katz smoothing, this is not the case. For n-gram models with add-one or no smoothing, this is because they do not employ backoff. At higher values of n, n-grams are much more sparse, so without any backoff higher n-gram models can only rely on sparse counts, resulting is lower probabilities being assigned to plausible sequences of words. Katz smoothing does employ backoff, and achieves much better performance, but it still distributes too much probability to rare n-grams.

A comparison of RNN-based models with n-gram models

As described in the implementation chapter, I also implemented three RNN-based language models which differ in the RNN cell architecture: vanilla RNN, Gated Recurrent Unit and Long Short-Term Memory.

- Plot perplexity (or cross-entropy) and average-keys-saved as a function of the amount of training data used.

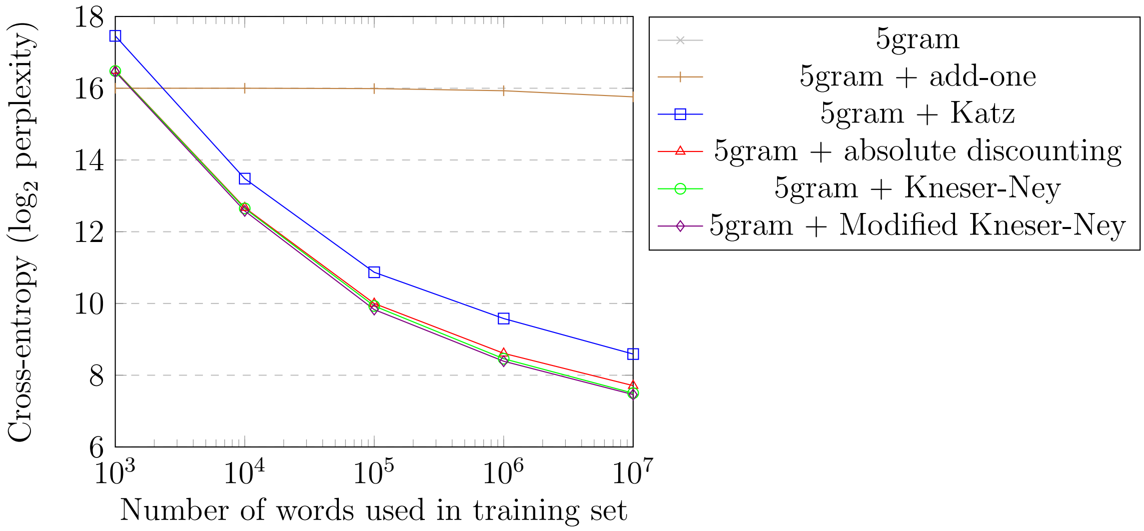


Figure 4.2: Cross-entropy of various language models with respect to the training set size.

Points to discuss:

- Average-keys-saved and guessing entropy taken over first 1000 words, whereas perplexity over whole test set.
- Note n-gram models are unpruned (hence large size).
- Katz beaten by absolute-discounting-derived methods on all fronts.
- Modified Kneser-Ney only offers marginal gain in perplexity.
- Reduction in performance of n-gram, add-one and Katz with N, due to sparsity of n-grams at higher lengths.
- Combination via average.
- Average inference times should be taken with a pinch of salt, because you have to consider CPU scheduling etc.
- Training times vary for LSTM and GRU in unexpected way, because LSTM happened to converge faster.

Language Model	**Perplexity**	**Average-Keys-Saved**	**Memory Usage (MB)**	**Training Time (min,secs)**†	**Average Inference Time (ms)**
3-gram	$4.54 \times 10^{5*}$	0.35014	266.91	**11s**	62
3-gram + add-one	3764.96	0.53063	266.94	**11s**	41
3-gram + Katz	256.95	0.68482	266.71	14s	88
3-gram + absolute disc.	166.03	0.72178	266.78	13s	63
3-gram + KN	150.73	0.72466	266.88	14s	54
3-gram + modified KN	149.54	0.72355	266.97	14s	54
5-gram	$1.96 \times 10^{8*}$	0.07167	737.36	26s	130
5-gram + add-one	8610.45	0.33886	737.30	26s	63
5-gram + Katz	314.49	0.67154	737.43	41s	156
5-gram + absolute disc.	167.38	0.72333	737.43	40s	126
5-gram + KN	146.35	0.72598	737.37	44s	114
5-gram + modified KN	142.68	0.72554	737.53	50s	116
Vanilla RNN	131.03	0.72776	**253.67**	15m 10s	39
Gated Recurrent Units	114.52	0.73993	271.39	28m 35s	**37**
Long Short-Term Memory	112.47	0.73617	287.13	18m 59s	38
LSTM, 5-gram + MKN (av)	96.07	0.75719	929.39	19m 49s	189
LSTM, 5-gram + MKN (int)	**94.70**	**0.75830**	927.20	19m 49s	190

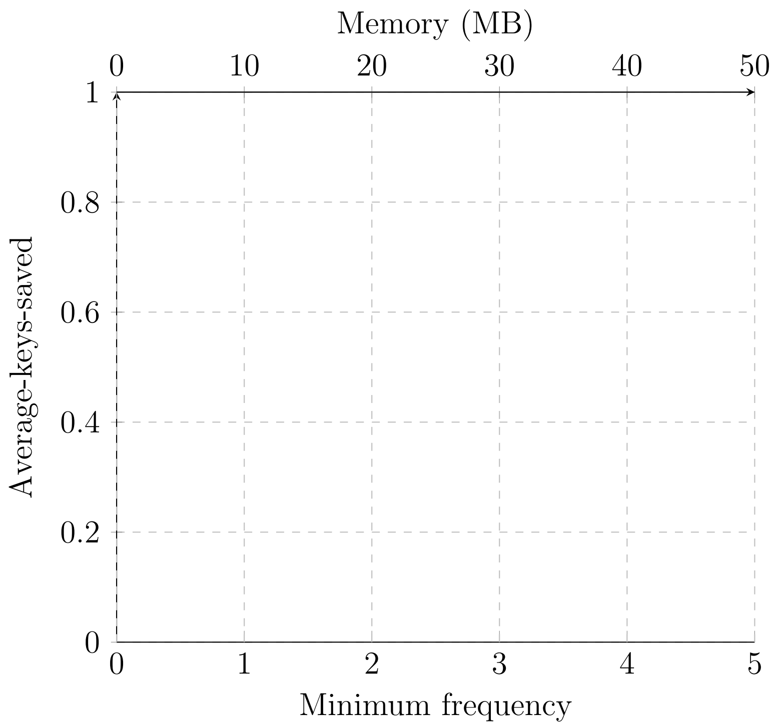
* *These perplexity scores use the approximation mentioned in section 4.1.1 where 0-valued probabilities are replaced with a small constant to avoid division by 0.* † *The n-gram models were trained on my laptop, whereas the neural models were trained on a GPU cluster on the High Performance Computing Service. Nevertheless, the n-gram models were still much faster to train.*

Figure 4.3: A benchmark of various language models on the PTB dataset.

4.2.2 On a Mobile Device

Incorporating an RNN-based language model implementation into a mobile keyboard, as described in section ***TODO:*** XXX, presented a different series of challenges. The strict memory and CPU limitations of App Extensions in iOS forced me to explore the tradeoffs between resource consumption and prediction performance.

As shown in figure ***TODO:*** XXX, the vanilla RNN architecture presents the smallest memory overhead, and so this is what I use in the following experiments. I investigated the effect of both vocabulary size and the number of hidden neurons on the memory usage and the average-keys-saved:



Here, I want to focus on the tradeoff between accuracy and resource consumption. Specifically, I could look at the following:

- The effect of increasing the minimum frequency for a word to be considered in the vocabulary. (I.e. the effect of changing the vocabulary size).
- The effect of changing the number of hidden layer neurons.
- The effect of using RNN vs GRU vs LSTM.
- (Perhaps also the effect of pruning on n-gram models).
- ALSO: Perhaps compare using float 32 vs float 16 in terms of memory vs accuracy.

The trade-off between the vocabulary size and resource usage

The trade-off between the number of hidden neurons and resource usage

The trade-off between the number of hidden neurons and resource usage

Other optimisations (not graphed):

- Remove softmax layer.
- Store weights as float 16 rather than float 32.

4.2.3 On Error-Prone Text

I split the CLC dataset up into six files:

`train.correct.txt`	`train.incorrect.txt`
`valid.correct.txt`	`valid.incorrect.txt`
`test.correct.txt`	`test.incorrect.txt`

The training, validation and test sets all consisted of one file containing uncorrected text and another file containing the corresponding corrected text. I trained an 2-layer LSTM-based language model with 256 hidden neurons on `train.correct.txt`, and used `valid.correct.txt` to guide the learning rate decay. I filtered the test set files such that they only contained pairs of lines that were not identical and that did not contain any insertion or deletion corrections.

Before exploring ways to improve the performance on error-prone text, I first established an approximate upper and lower bound. To obtain the upper bound (i.e. the best possible perplexity), I evaluated the LSTM using `test.correct.txt` for the input words and `test.correct.txt` for the target words. and 'In order to establish a rough upper and lower bound on the performance of the LSTM on error-prone text, I first evaluated Things I aim to evaluate here are:

- The hypothetical upper and lower bounds on accuracy (i.e. the LM results on correct input and on incorrect input respectively).
- The effect of using the vocabulary vs different sized dictionaries for determining if a word should be replaced or not.
- The effect of edit distance on performance.
- An intuitive explanation behind the gap remaining between the current performance and the upper bound. Perhaps some suggestions for future work.

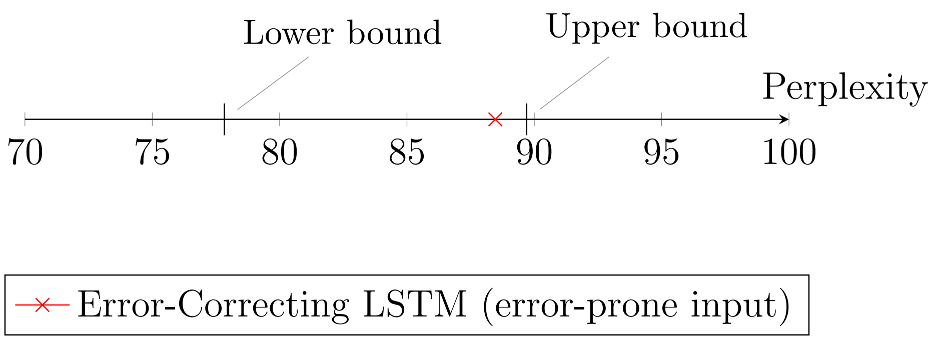


Figure 4.4: Perplexity on Cambridge Learner Corpus

Chapter 5

Conclusions

Conclusions:

-

Future work/possible extensions:

- Using Neural Machine Translation to correct the context, making use of attention.
-

Bibliography

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

[2] S. F. Chen and J. Goodman, “An empirical study of smoothing techniques for language modeling,” in *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, pp. 310–318, Association for Computational Linguistics, 1996.

[3] S. F. Chen and J. Goodman, “An empirical study of smoothing techniques for language modeling,” *Comput. Speech Lang.*, vol. 13, pp. 359–394, Oct. 1999.

[4] H. T. Ng, S. M. Wu, Y. Wu, C. Hadiwinoto, and J. Tetreault, “The conll-2013 shared task on grammatical error correction,” 2013.

[5] H. T. Ng, S. M. Wu, T. Briscoe, C. Hadiwinoto, R. H. Susanto, and C. Bryant, “The conll-2014 shared task on grammatical error correction,” in *CoNLL Shared Task*, pp. 1–14, 2014.

[6] W. E. Johnson, “Probability: The deductive and inductive problems,” *Mind*, vol. 41, no. 164, pp. 409–423, 1932.

[7] H. Ney, U. Essen, and R. Kneser, “On structuring probabilistic dependences in stochastic language modelling,” *Computer Speech & Language*, vol. 8, no. 1, pp. 1–38, 1994.

[8] R. Kneser and H. Ney, “Improved backing-off for m-gram language modeling,” in *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, vol. 1, pp. 181–184, IEEE, 1995.

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

[10] S. Katz, “Estimation of probabilities from sparse data for the language model component of a speech recognizer,” *IEEE transactions on acoustics, speech, and signal processing*, vol. 35, no. 3, pp. 400–401, 1987.

[11] F. A. Azevedo, L. R. Carvalho, L. T. Grinberg, J. M. Farfel, R. E. Ferretti, R. E. Leite, R. Lent, S. Herculano-Houzel, *et al.*, “Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain,” *Journal of Comparative Neurology*, vol. 513, no. 5, pp. 532–541, 2009.

[12] W. S. McCulloch and W. Pitts, “A logical calculus of the ideas immanent in nervous activity,” *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[13] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," tech. rep., DTIC Document, 1985.

[14] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

[15] Y. Bengio, Y. LeCun, *et al.*, "Scaling learning algorithms towards ai," *Large-scale kernel machines*, vol. 34, no. 5, pp. 1–41, 2007.

[16] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.

[17] H. Yannakoudakis, T. Briscoe, and B. Medlock, "A new dataset and method for automatically grading esol texts," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1*, pp. 180–189, Association for Computational Linguistics, 2011.

Appendix A

Project Proposal