UNIVERSITY OF CAMBRIDGE

Devan Kuleindiren
Robinson College
dk503

Computer Science Tripos - Part II Project

Language Modelling for Text Prediction

April 28, 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:	12131
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

1 Introduction **5**
- 1.1 Language Models . . . 5
- 1.2 Motivation . . . 6
- 1.3 Related Work . . . 6

2 Preparation **7**
- 2.1 n-gram Models . . . 7
  - 2.1.1 An Overview of n-gram Models . . . 7
  - 2.1.2 Smoothing Techniques . . . 8
- 2.2 Recurrent Neural Network Models . . . 11
  - 2.2.1 An Overview of Neural Networks . . . 11
  - 2.2.2 Recurrent Neural Networks . . . 14
  - 2.2.3 Word Embeddings . . . 16
- 2.3 Software Engineering . . . 16
  - 2.3.1 Requirements . . . 16
  - 2.3.2 Tools and Technologies Used . . . 17
  - 2.3.3 Starting Point . . . 18

3 Implementation **19**
- 3.1 System Overview . . . 19
  - 3.1.1 The Language Model Hierarchy . . . 20
  - 3.1.2 Testing . . . 21
- 3.2 n-gram Models . . . 21
  - 3.2.1 Computing the Vocabulary . . . 22
  - 3.2.2 Counting n-grams Efficiently . . . 22
  - 3.2.3 Precomputing Smoothing Coefficients . . . 23
- 3.3 Recurrent Neural Network Models . . . 24
  - 3.3.1 TensorFlow . . . 24
  - 3.3.2 Network Structure . . . 26
  - 3.3.3 Long Short-Term Memory and Gated Recurrent Units . . . 26
  - 3.3.4 Word Embeddings . . . 29
  - 3.3.5 Putting Theory into Practice . . . 30
- 3.4 Mobile Keyboard . . . 32
  - 3.4.1 Updating Language Model Predictions On the Fly . . . 33
- 3.5 Extending Models to Tackle Error-Prone Text . . . 34
  - 3.5.1 The Approach . . . 34

4 Evaluation **36**
4.1 Evaluation Methodology 36
4.1.1 Metrics 36
4.1.2 Datasets 37
4.2 Results 38
4.2.1 Existing Models 38
4.2.2 On a Mobile Device 41
4.2.3 On Error-Prone Text 42

5 Conclusions **45**

Bibliography **46**

A Backpropagation Recurrence Relation **48**

B Perplexity **50**

C Project Proposal **51**

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 improves 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.

$\overbrace{\textit{do you want to grab a}}^{w_1^k}$	$\overbrace{\underline{\textit{drink}}}^{w_{k+1}}$	$\overbrace{(0.327)}^{\mathbb{P}(w_{k+1}\|w_1^k)}$
	coffee	(0.211)
	bite	(0.190)
	spot	(0.084)
	⋮	⋮

My project focuses on language modelling in the context of text prediction. That is, given a sequence of words $w_1 w_2 ... w_k = w_1^k$, 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'*.

1.2 Motivation

Benchmarking

Language models are central to a wide range of applications, but there are so many different ways of implementing them. In this project I tackled this vast array of choice by focusing in depth on the two most prominent types of language model: n-gram models and recurrent neural network-based models. I investigated n-gram models coupled with five different smoothing techniques and recurrent neural network (RNN) models of three different flavours 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. 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 techniques [4] [5]. These methods, however, are allowed to take the whole text as input, whereas language models cannot see beyond the next word that they are trying to predict. Therefore, language modelling on error-prone text presents a slightly different challenge.

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. By repeated application of the product rule, it follows that:

$$\mathbb{P}(w_1^N) = \prod_{i=1}^{N} \mathbb{P}(w_i | w_1^{i-1}) \tag{2.1}$$

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

$$\mathbb{P}(w_1^N) \approx \prod_{i=1}^{N} \mathbb{P}(w_i | w_{i-n+1}^{i-1}) \tag{2.2}$$

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

$$\mathbb{P}(w_i | w_{i-n+1}^{i-1})_{MLE} = \frac{c(w_{i-n+1}^{i})}{\sum_w c(w_{i-n+1}^{i-1} w)} \tag{2.3}$$

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

This model provides reasonable results and is simple to compute, but it does have one major issue: if, for example, a 3-gram model does not encounter the trigram *'the cat sat'* in its training data, 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 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.

A crude solution to this problem is to exponentially increase the size of the training set, but this requires 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, and the ones which I have implemented are described in the next section.

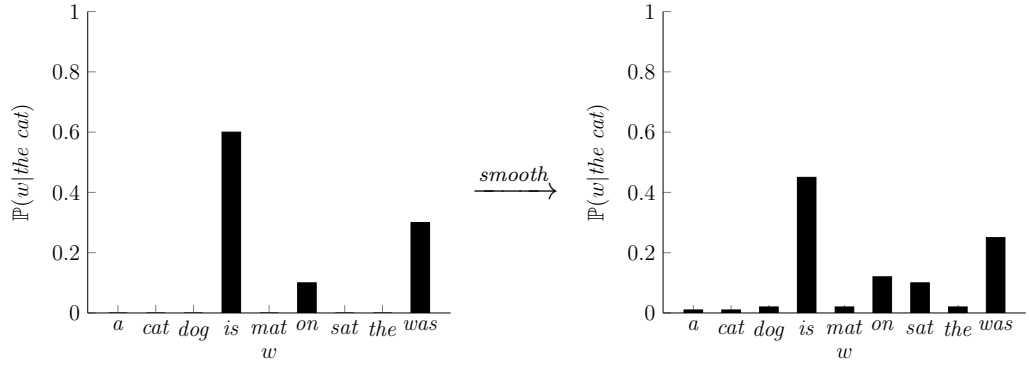


Figure 2.1: A toy example of smoothing, where the vocabulary is $V = \{a,\ cat,\ dog,\ is,\ mat,\ on,\ sat,\ the,\ was\}$.

2.1.2 Smoothing Techniques

Add-One Smoothing

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

$$\mathbb{P}(w_i|w_{i-n+1}^{i-1})_{\text{ADD-ONE}} = \frac{c(w_{i-n+1}^{i}) + 1}{\sum_w c(w_{i-n+1}^{i-1}w) + |V|} \tag{2.4}$$

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. For example, if both *'cat is'* and *'cat pizza'* are unseen in the training data of a bigram model, then $\mathbb{P}(is \mid cat)_{\text{ADD-ONE}} = \mathbb{P}(pizza \mid cat)_{\text{ADD-ONE}}$. A simple way to reduce this problem is to employ *backoff*, a technique whereby you recurse on the probability calculated by the $(n-1)$-gram model. In this case, it is likely that $\mathbb{P}(is)_{\text{ADD-ONE}} > \mathbb{P}(pizza)_{\text{ADD-ONE}}$, which could be used to deduce that *'cat is'* is more likely than *'cat pizza'*.

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:

$$\begin{aligned}\mathbb{P}(w_i|w_{i-n+1}^{i-1})_{\text{ABS}} = &\frac{max\{c(w_{i-n+1}^{i}) - D, 0\}}{\sum_w c(w_{i-n+1}^{i-1}w)} \\ &+ \frac{D}{\sum_w c(w_{i-n+1}^{i-1}w)} N_{1+}(w_{i-n+1}^{i-1}\bullet)\mathbb{P}(w_i|w_{i-n+2}^{i-1})_{\text{ABS}}\end{aligned} \tag{2.5}$$

where

$$N_{1+}(w_{i-n+1}^{i-1}\bullet) = |\{w \mid c(w_{i-n+1}^{i-1}w) \geq 1\}| \tag{2.6}$$

and the base case of recursion is given by $\mathbb{P}(w)_{\text{ABS}} = \mathbb{P}(w)_{\text{MLE}}$. $N_{1+}(w_{i-n+1}^{i-1}\bullet)$ 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] suggest setting D to the value:

$$D = \frac{n_1}{n_1 + 2n_2} \tag{2.7}$$

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 bigrams *'bottle cap'* and *'bottle Francisco'*. If neither have been seen in the training data, and *'Francisco'* occurs more frequently than *'cap'*, then the absolute discounting model would backoff onto the unigram distribution and assign a higher probability to the *'bottle Francisco'* bigram, despite the fact that *'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, $N_{1+}(\bullet w_i^j)$:

$$N_{1+}(\bullet w_i^j) = |\{w \mid c(ww_i^j) \geq 1\}| \tag{2.8}$$

Kneser-Ney smoothing[1] is defined as follows:

$$\mathbb{P}(w_i|w_{i-n+1}^{i-1})_{\text{KN}} = \frac{max\{\gamma(w_{i-n+1}^{i}) - D, 0\}}{\sum_w \gamma(w_{i-n+1}^{i-1}w)} \tag{2.9}$$

$$+ \frac{D}{\sum_w \gamma(w_{i-n+1}^{i-1}w)} N_{1+}(w_{i-n+1}^{i-1}\bullet)\mathbb{P}(w_i|w_{i-n+2}^{i-1})_{\text{KN}} \tag{2.10}$$

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

where

$$\gamma(w_{i-k+1}^{i}) = \begin{cases} c(w_{i-k+1}^{i}) & \text{for the outermost level of recursion (i.e. } k = n) \\ N_{1+}(\bullet w_{i-k+1}^{i}) & \text{otherwise} \end{cases} \tag{2.11}$$

and the unigram probability is given as:

$$\mathbb{P}(w_i)_{\text{KN}} = \frac{N_{1+}(\bullet w_i)}{\sum_w N_{1+}(\bullet w)} \tag{2.12}$$

Modified Kneser-Ney Smoothing

Chen and Goodman noticed that the ideal average discount value, D, 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]:

$$\begin{aligned} \mathbb{P}(w_i|w_{i-k+1}^{i-1})_{\text{MKN}} = & \frac{max\{\gamma(w_{i-k+1}^{i}) - D(c(w_{i-k+1}^{i}), 0\}}{\sum_w \gamma(w_{i-k+1}^{i-1} w)} \\ & + \lambda(w_{i-k+1}^{i-1}) \mathbb{P}(w_i|w_{i-k+2}^{i-1})_{\text{MKN}} \end{aligned} \tag{2.13}$$

where γ is defined in equation 2.11, and λ is defined as:

$$\lambda(w_{i-k+1}^{i-1}) = \frac{D_1 N_1(w_{i-k+1}^{i-1}\bullet) + D_2 N_2(w_{i-k+1}^{i-1}\bullet) + D_{3+} N_{3+}(w_{i-k+1}^{i-1}\bullet)}{\sum_w \gamma(w_{i-k+1}^{i-1} w)} \tag{2.14}$$

and

$$D(c) = \begin{cases} 0 & \text{if } c = 0 \\ D_1 & \text{if } c = 1 \\ D_2 & \text{if } c = 2 \\ D_{3+} & \text{if } c \geq 3 \end{cases} \tag{2.15}$$

where N_1, N_2 and N_{3+} are defined analogously to N_{1+} in equation 2.6. Chen and Goodman suggest:

$$D_1 = 1 - 2D\frac{n_2}{n_1} \qquad D_2 = 2 - 3D\frac{n_3}{n_2} \qquad D_{3+} = 3 - 4D\frac{n_4}{n_3} \tag{2.16}$$

where D is as defined in equation 2.7.

Katz Smoothing

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

$$r^* = (r+1)\frac{n_{r+1}}{n_r} \tag{2.17}$$

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

$$\mathbb{P}(w_i|w_{i-n+1}^{i-1})_{\text{GT}} = \frac{c^*(w_{i-n+1}^{i})}{\sum_{r=0}^{\infty} n_r r^*} \tag{2.18}$$

Katz smoothing [10] is then defined as:

$$\mathbb{P}(w_i|w_{i-n+1}^{i-1})_{\text{KATZ}} = \begin{cases} \mathbb{P}(w_i|w_{i-n+1}^{i-1})_{\text{GT}} & \text{if } c(w_{i-n+1}^{i}) > 0 \\ \alpha(w_{i-n+1}^{i-1})\mathbb{P}(w_i|w_{i-n+2}^{i-1})_{\text{KATZ}} & \text{otherwise} \end{cases} \tag{2.19}$$

where

$$\alpha(w_{i-n+1}^{i-1}) = \frac{1 - \sum_{\{w_i \mid c(w_{i-n+1}^{i})>0\}} \mathbb{P}(w_i|w_{i-n+1}^{i-1})_{\text{KATZ}}}{1 - \sum_{\{w_i \mid c(w_{i-n+1}^{i})>0\}} \mathbb{P}(w_i|w_{i-n+2}^{i-1})_{\text{KATZ}}} \tag{2.20}$$

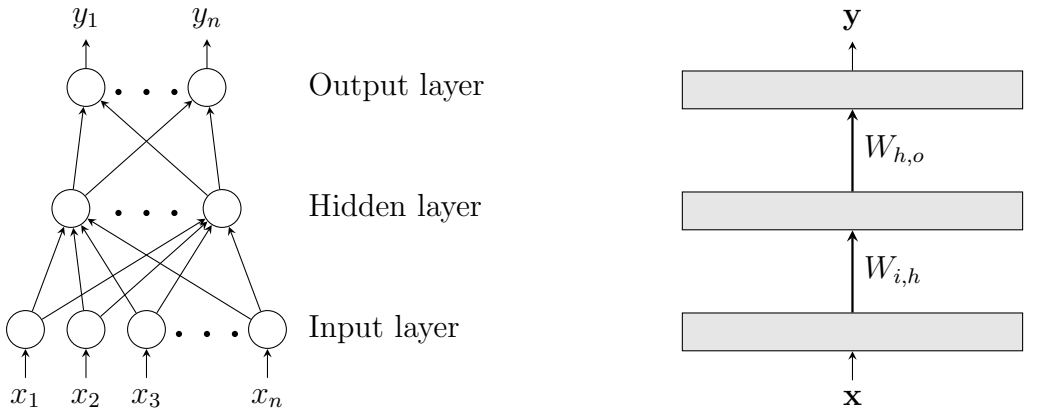
In practice, the infinite sum in equation 2.18 cannot be computed. To get around this issue, Katz takes n-grams with counts above some threshold k as reliable and only applies the Good-Turing estimate to those with a count less than or equal to k. Katz suggests $k = 5$. This modification requires a slightly different equation to 2.19 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.

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[2] that propagate electrochemical signals across connections called synapses. (Artificial) neural networks were originally developed as a mathematical model of the brain [12], which despite being substantially oversimplified, now provide an effective tool for classification and regression in modern-day machine learning.



(a) An example multilayer perceptron.

(b) A simplified notation for (a).

Figure 2.2: Two alternative notations for the same network.

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

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

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].

The Multilayer Perceptron

The multilayer perceptron consists of layers of neurons, where each layer is fully connected to the next one. The first and last layers are the input and output layers, and any layers in between are called *hidden layers*. The input neurons simply pass on the input values that they are given. Neurons in the subsequent layers, however, compute the following:

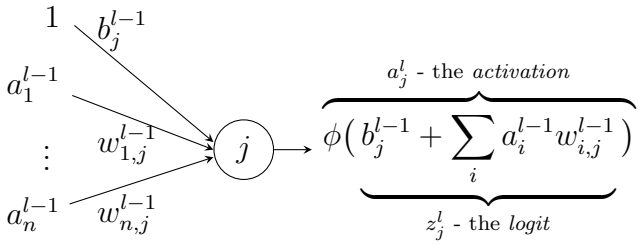
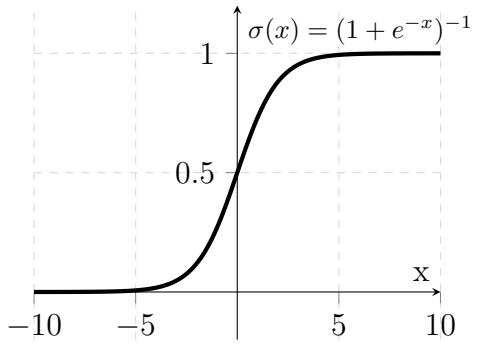


Figure 2.3: The computation carried out by each neuron. The outputs or *activations* of the neurons from the previous layer, $(l-1)$, are denoted a_i^{l-1}, the weights on the input connections are $w_{i,j}^{l-1}$, the *activation function* is ϕ and b_j^{l-1} is the bias variable.

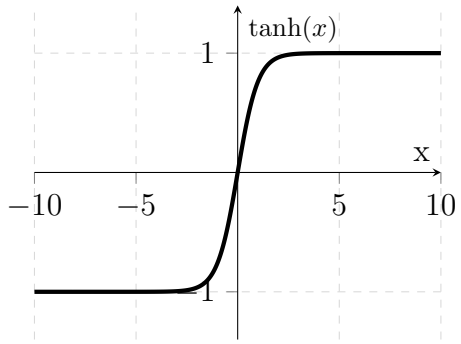
The constant-valued input 1 is known as a *bias* input, which is weighted by a bias variable b_j^{l-1}. Its purpose is to allow the neuron to shift the output of the activation function left and right by adjusting the value of b_j^{l-1}. Bias inputs are usually attached to every non-input neuron in the network.

Activation functions were originally created to mimic the firing activity of neurons. It is important that they are 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. Another important property of these functions is that they are differentiable, which allows for the network to be trained using *gradient descent*, which is discussed below.



(a) The sigmoid function, σ.



(b) The hyperbolic tangent function, tanh.

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 these vectors 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 $\mathbf{t}_k$ is the vector that the network should output given the example input $\mathbf{x}_k$. 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:

$$\mathcal{L}_{\text{TOTAL}} = \sum_{k=1}^{N} \mathcal{L}(f_{\mathbf{w}}(\mathbf{x}_k), \mathbf{t}_k) \tag{2.21}$$

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

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

for every weight $w_{i,j}^l$ in the neural network. It is a two stage algorithm that works as follows:

1. Calculate, $f_{\mathbf{w}}(\mathbf{x}_k)$ for each input example $\mathbf{x}_k$, and use those values to derive $\mathcal{L}_{\text{TOTAL}}$.
2. Given $\mathcal{L}_{\text{TOTAL}}$, calculate $\frac{\partial}{\partial w_{i,j}^l}(\mathcal{L}_{\text{TOTAL}})$ for each weight in each layer of the network by applying the chain rule backwards from the output layer to the input layer.

$\frac{\partial}{\partial w_{i,j}^l}(\mathcal{L}_{\text{TOTAL}})$ is calculated as follows:

$$\frac{\partial}{\partial w_{i,j}^l}(\mathcal{L}_{\text{TOTAL}}) = \sum_{k=1}^{N} \underbrace{\frac{\partial \mathcal{L}}{\partial f_{\mathbf{w}}(\mathbf{x}_k)} \frac{\partial f_{\mathbf{w}}(\mathbf{x}_k)}{\partial z_j^{l+1}}}_{\delta_j^{l+1}} \frac{\partial z_j^{l+1}}{\partial w_{i,j}^l} = \delta_j^{l+1} a_i^l \tag{2.22}$$

The delta term $\delta_j^{l+1} = \frac{\partial \mathcal{L}}{\partial z_j^{l+1}}$ varies in form for each layer. For the output layer, it has the following form:

$$\delta_j^L = \frac{\partial \mathcal{L}}{\partial f_{\mathbf{w}}(\mathbf{x}_k)} \phi'(z_j^L) \tag{2.23}$$

For the preceding layers, δ_j^l can be defined recursively in terms of the delta values for the neurons in subsequent layers in the network. This is defined below, and a detailed justification of it is given in appendix A:

$$\delta_i^l = \phi'(z_i^l) \sum_j \delta_j^{l+1} w_{i,j}^l \tag{2.24}$$

Gradient descent is an optimisation technique that uses the derivatives produced by the backpropagation algorithm to adjust the weights such that the loss is minimised over the training set. It does this by changing each weight value in the direction of the negative gradient of the loss, which can be visualised as moving downhill on the error surface plotted with respect to the weights in the network:

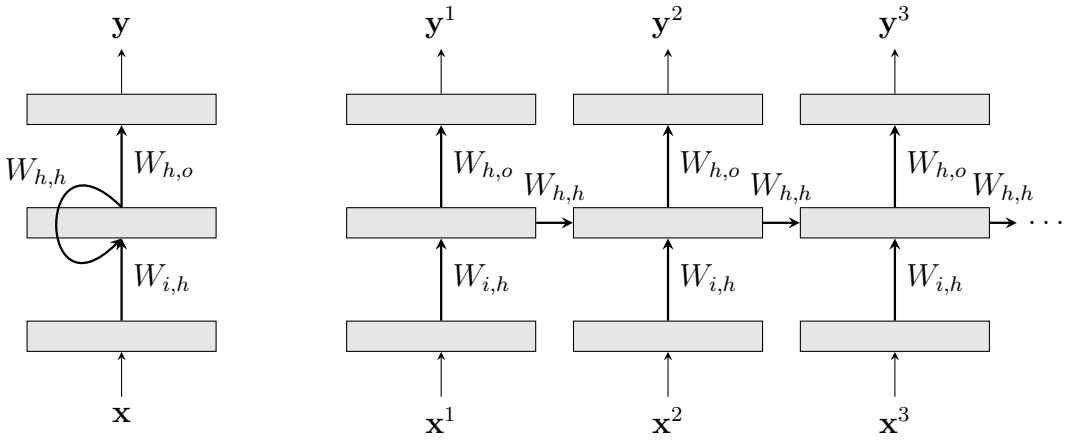
$$w_{i,j}^l \leftarrow w_{i,j}^l - \eta \frac{\partial}{\partial w_{i,j}^l}(\mathcal{L}_{\text{TOTAL}}) \tag{2.25}$$

$\eta > 0$ is known as the *learning rate*, which determines how large the steps are in the negative direction of the gradient. Ways for setting η, and other useful techniques when training neural networks in practice are described in chapter 3.

2.2.2 Recurrent Neural Networks

In the context of language modelling, the input is a sequence of words and the output is a sequence of probability distributions over the possible next words at each point. Ideally, the word with the highest probability at each step should be the next word in the input sequence. For example, given (*'the', 'cat', 'sat', ...*) as input, the network should produce something along the lines of ( ,  ,  , ...) as output. The problem with FNNs is that these sequences may have varying length, yet FNNs have fixed input and output vector sizes. Recurrent neural networks (RNNs) naturally lend themselves to tackling this issue.

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. A nice property about RNNs is that they can be unrolled to an arbitrary number of steps without the need for any additional parameters, because the same weight matrices are reused at each step. For this reason, RNNs are well suited for operating over sequences of



(a) An RNN.

(b) An *unrolled* RNN.

Figure 2.5: Two different representations of the same RNN.

input data, which is of course ideal in the context of language modelling. The problem of representing textual words as numerical input vectors for an RNN is addressed in section 2.2.3.

In most applications, the sequences of input vectors are typically ordered by time, so I use the notation $\mathbf{x}^t$ and $\mathbf{y}^t$ to denote the input and output vectors at time step t.

Backpropagation Through Time (BPTT)

Training an RNN is not so much different to training an FNN: the only difference is that gradients must also be calculated across time steps. As shown figure 2.5b, each weight is reused at every time step, so the weight derivative is now summed across the time steps:

$$\frac{\partial}{\partial w_{i,j}^l}\big(\mathcal{L}_{\text{TOTAL}}\big) = \sum_t \delta_j^{l+1,t} a_i^{l,t} \tag{2.26}$$

The delta term for the output layer is defined analogously to equation 2.23:

$$\delta_j^{L,t} = \frac{\partial \mathcal{L}}{\partial f_{\mathbf{w}}(\mathbf{x}_k^t)} \phi'(z_j^{L,t}) \tag{2.27}$$

The delta term for the preceding layers can be shown to have the form below. This is justified in more detail in appendix A:

$$\delta_i^{l,t} = \phi'(z_i^{l,t})\big(\sum_j \delta_j^{l+1,t} w_{i,j}^l + \sum_j \delta_j^{l,t+1} v_{i,j}^l\big) \tag{2.28}$$

where $v_{i,j}^l$ denotes the weight on the recurrent connection from neuron i in layer l back to neuron j in the same layer.

RNN Architectures

So far, the equations presented have assumed a *vanilla RNN*. That is, the j^{th} neuron, or *cell*, in layer l at time step t simply computes:

$$a_j^{l,t} = \phi\big(\sum_i a_i^{l-1,t} w_{i,j}^{l-1} + \sum_i a_i^{l,t-1} v_{i,j}^l + b_j^{l-1}\big) \tag{2.29}$$

This computation can be written for an entire layer of hidden neurons, $\mathbf{a}^{l,t} = (a_1^{l,t}, a_2^{l,t}, ..., a_H^{l,t})$, using matrix notation as follows:

$$\mathbf{a}^{l,t} = \phi\big(\mathbf{W}^{l-1}\mathbf{a}^{l-1,t} + \mathbf{V}^{l}\mathbf{a}^{l,t-1} + \mathbf{b}^{l-1}\big) \tag{2.30}$$

Various improvements to this architecture have been proposed, which capture longer-term information across the inputs. In this project, I have implemented the two most prominent alternatives, which are Long Short-Term Memory (LSTM) [17] and the Gated Recurrent Unit (GRU) [18]. These are outlined in figure 2.7 and discussed in much more detail in section 3.3.3.

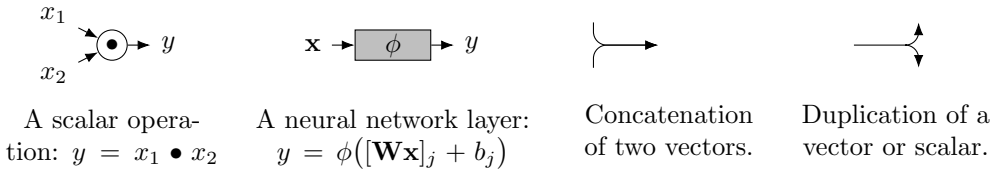
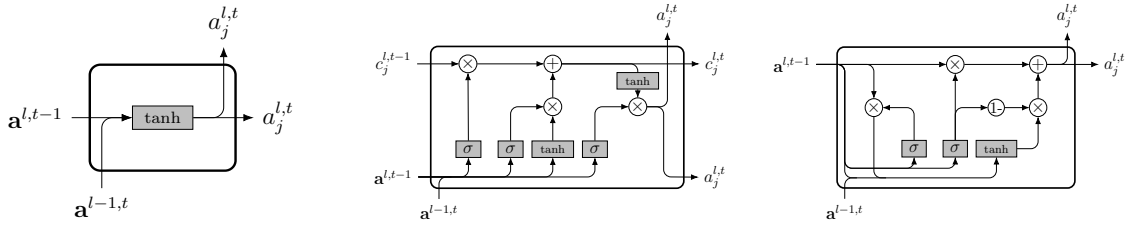


Figure 2.6: A notation for drawing RNN architectures.



(a) A vanilla RNN cell. (b) An LSTM cell. (c) A GRU cell.

Figure 2.7: Various architectures of RNN neurons, using the notation from figure 2.6. Each diagram represents the computation of the j^{th} neuron or cell in layer l.

2.2.3 Word Embeddings

In order for an RNN to take a sequence of words as input, each word must first be mapped to a vector representation, known as a *word embedding*. Each word embedding represents a point in a high-dimensional space, and ideally, similar words will be assigned points which are closer together. In this dissertation, I get the RNN to learn good word embeddings through the application of backpropagation and gradient descent to the vector elements.

2.3 Software Engineering

This section details the project requirements and early design decisions that were made.

2.3.1 Requirements

The success requirements set out in the project proposal were:

R1	Language models (LMs) using the following techniques are implemented: • n-gram LMs with various smoothing techniques. • A vanilla RNN-based LM. • An LSTM-based LM.
R2	Comprehensible and reliable comparisons between the various LM implementations and their combinations are made regarding their accuracy, speed and resource consumption during both training and inference.
R3	A simple console application is developed to demonstrate the capability of the aforementioned language models in the context of next-word prediction.

The project proposal also suggested two possible extensions:

E1	Explore possible extensions to existing language models to improve their performance on error-prone text.
E2	Build a mobile keyboard that exhibits the functionality of the language models.

All requirements and extensions depend on **R1**, so I implemented that first. Afterwards, I built **R2** and **R3** in parallel before moving onto **E1** and **E2**.

2.3.2 Tools and Technologies Used

Below I describe and justify where necessary the tools and technologies that I used.

Version Control and Build Tools

I hosted my project in a repository on GitHub, used Git for version control, and used Bazel for running builds and tests on my project.

Machine Learning

I chose to use TensorFlow, an open source machine learning library, to assist in the implementation of my recurrent neural networks. There are two key reasons for this:

1. TensorFlow supports *automatic differentiation*. That is, each tensor operation has a partial derivative associated with it, so when you want to apply the second stage of backpropagation, TensorFlow can automatically apply the chain rule through any sequence of tensor operations. Without this, I would have to hard-code new gradient calculations every time I change the structure of the network, which is tedious and error-prone.

2. TensorFlow supports GPU-accelerated matrix operations.

Of course, there are many other machine learning libraries that offer comparable features. The reason I picked TensorFlow in particular was because I was already familiar with its codebase from using and contributing to it during an internship at Google in 2016.

In order to train large scale models on NVIDIA GPUs, I used the University's High Performance Computing service.

Languages

At the time of writing my code, TensorFlow had both a C++ and a Python API. However, the C++ API only had support for loading and running inference on models, not for training them.

One of the possible extensions I set out in my project proposal was to implement a mobile keyboard that makes use of my language model implementations. Android and iOS both provide little or no support for applications written in Python, but they do support code written in C++ via the JNI and Objective-C++ respectively.

For these reasons, I wrote code to train and export my RNN-based models in Python, and then wrote some C++ classes for loading and running inference on them. The benchmarking framework and my n-gram models were all written in C++. In order to export and import trained language models across different languages, I used Google's protocol buffers: a language neutral mechanism for serialising structured data.

Testing

I used Google Test for C++ unit tests and the `unittest` package for testing in Python.

2.3.3 Starting Point

My project codebase was written from scratch, with the assistance of the tools and libraries mentioned above. Apart from a basic knowledge of recurrent neural networks, I had to learn about language modelling, n-gram smoothing techniques, Long Short-Term Memory and Gated Recurrent Units through personal reading.

Experience	**Tools and Technologies**
Significant	C++, Python, Git, GitHub, TensorFlow
Some	Bazel, iOS, Protocol Buffers
None	Google Test, Objective-C++, SLURM

Figure 2.8: A summary of my prior experience.

Chapter 3

Implementation

This chapter details how I implemented a variety n-gram and RNN-based language models, how I built the mobile keyboard on iOS and how I extended an existing language model to improve its performance on error-prone text.

3.1 System Overview

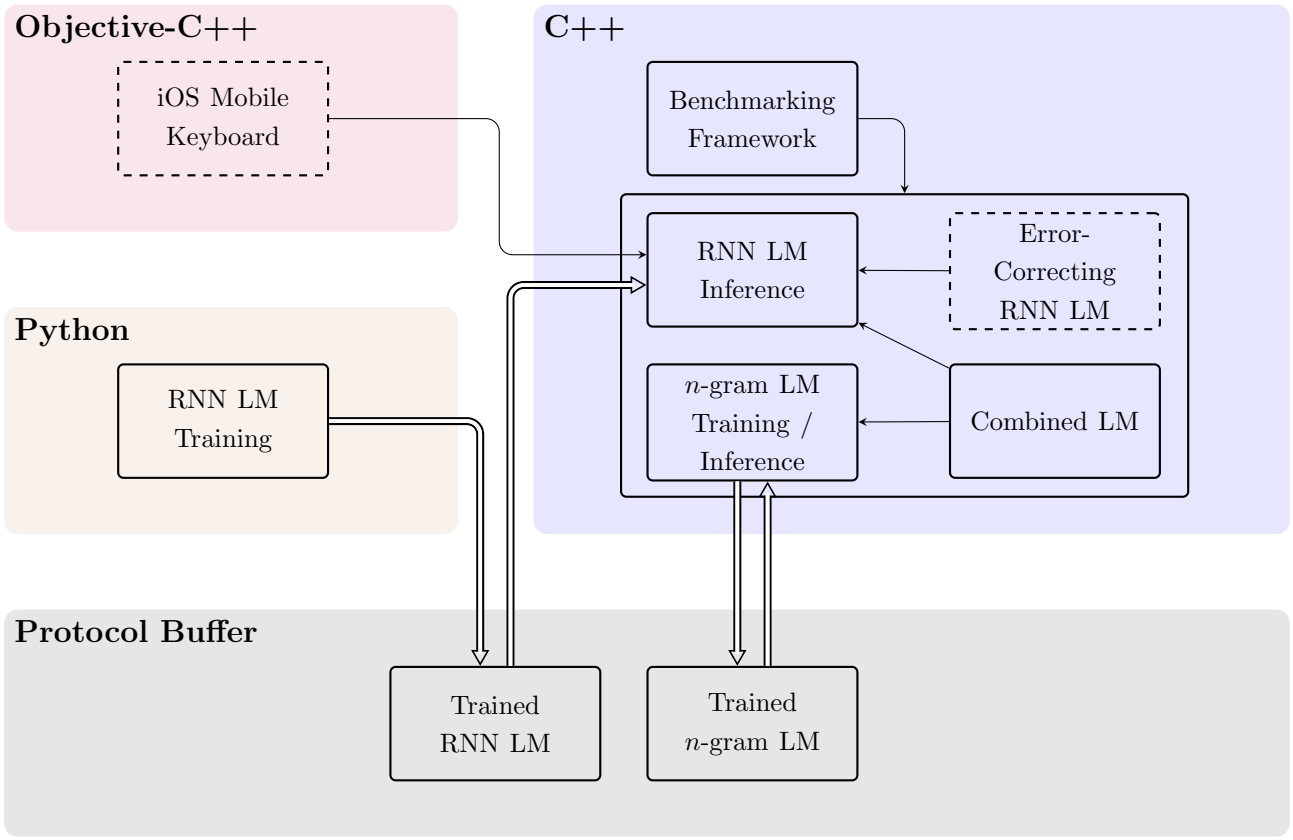


Figure 3.1: An overview of the project. Outlined rectangles denote groups of related files or code, of which the ones with a dashed outline represent project extensions. Thin arrows denote dependencies and thick arrows represent the movement of data.

As shown, all of the code for training and running inference on n-gram models was written in C++, and is described in section 3.2. The RNN-based language models were trained

in Python using TensorFlow, before being exported into a small group of protocol buffers that can be loaded into C++ for running inference, all of which is detailed in section 3.3. The 'Combined LM' component represents a language model which interpolates the probabilities across one or more n-gram and/or RNN-based language models. This part is not described in depth, because it is mostly trivial. I outline in section 3.4 how I built a mobile keyboard for iOS that utilises one of the RNN-based language models. Finally, the 'Error-Correcting RNN LM' component denotes an extension of my RNN-based language models which I made in order to improve their performance on error-prone text, which is described in section 3.5.

3.1.1 The Language Model Hierarchy

All of my language model implementations in C++ were joined by the same inheritance hierarchy, at the top of which was a single language model superclass, `LM`. This is the only class that the benchmarking framework depended on:

```
class LM {
protected:
    Vocab *vocab;
    ...
public:
    ...
    // Initialise 'preds' with the top k most probable next words the follow 'seq', in
    // order of decreasing probability.
    virtual void PredictTopK(std::list<std::string> seq,
                             std::list<std::pair<std::string, double>> &preds, int k);

    // Initialise 'pred' with the most probable next word that follows 'seq'.
    virtual void Predict(std::list<std::string> seq,
                         std::pair<std::string, double> &pred);

    // Return the probability of the word sequence, 'seq'.
    virtual double Prob (std::list<std::string> seq) = 0;

    // Given a sequence of words, 'seq', initialise 'probs' with the probability of each
    // word in the vocabulary following that sequence.
    virtual void ProbAllFollowing (std::list<std::string> seq,
                                   std::list<std::pair<std::string, double>> &probs) = 0;

    // The same as above, except the word-probabilities are stored in a character trie.
    virtual void ProbAllFollowing (std::list<std::string> seq, CharTrie *probs) = 0;
    ...
};
```

Every language model object stores a `Vocab` object, which contains the set of words in the language model vocabulary, and a mapping from those words to integer IDs. As a simple memory optimisation, the integer IDs are used in place of the string representations of such words in various language model data structures that are discussed later in this chapter.

`Predict()` and `PredictTopK()` both call `ProbAllFollowing()`. `Predict()` takes the maximum over the result and `PredictTopK()` uses a priority queue to extract the top k predictions. `ProbAllFollowing()` could be implemented by calling `Prob()` for each word in the vocabulary, but this would be inefficient in the case of an RNN-based language

model, which implicitly computes the probability distribution over every word in each forward pass of the network, hence why the methods are kept separate.

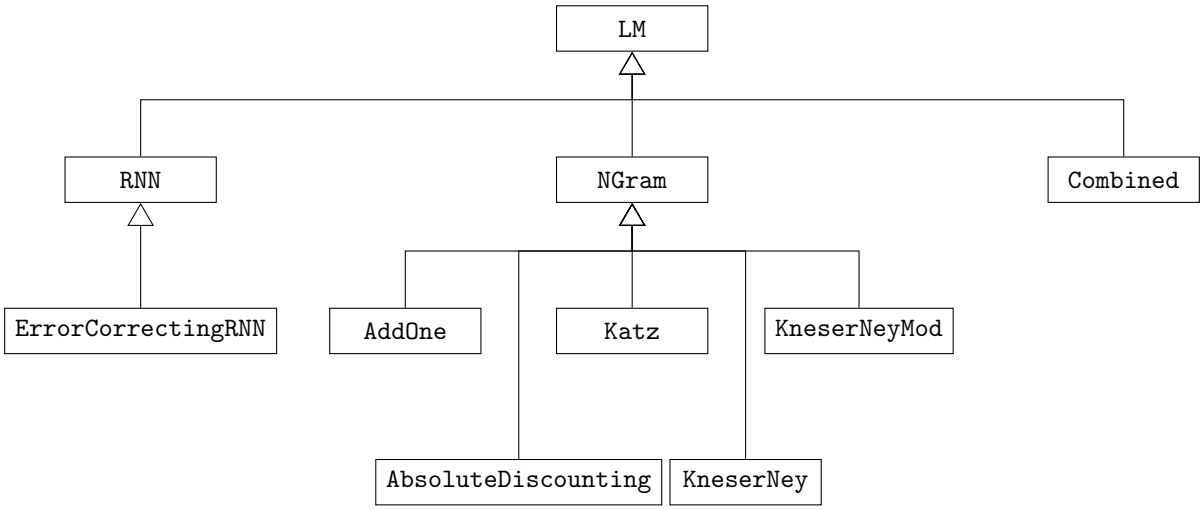


Figure 3.2: An outline of the C++ classes in the language model inheritance hierarchy. Methods and members are omitted for simplicity.

For the RNN-based language models, the differences between the vanilla RNN, GRU and LSTM implementations arise in the Python code where the models are constructed and trained. The C++ code for the RNN-based models just loads and runs inference on the model graph described in the exported protocol buffer. See 3.3.1 for more details.

3.1.2 Testing

To encourage correctness, every non-trivial C++ and Python class in the codebase has an accompanying test class which contains one or more unit tests. In particular, the language model classes are paired with tests that query their `Prob()` and `ProbAllFollowing()` methods and compare the results with values that I calculated by hand using the equations presented in chapter 2.

During the development of the project, all new code was tested before being committed to the codebase, and new test cases were immediately added upon discovery of any bugs.

3.2 n-gram Models

Below I detail my implementation of the n-gram models and their associated smoothing techniques.

One of the first things I discovered when implementing n-gram language models is that there is a trade-off between how much computation is done at training time, and how much computation is done at inference time. For example, at one extreme you could compute only the number of times each n-gram occurs at training time, and leave the computation of the coefficients in the smoothing equations to be calculated on the fly. At the other extreme, you could compute and store the final value of the probability for every

possible n-gram that can be formed from the words in the language model vocabulary, and then return the relevant probability immediately at inference time. The problem with the first approach is that the language model would be too slow at inference time, and the problem with the second approach is that storing absolute probabilities for every possible n-gram would occupy far too much space to be practical. In my implementation, I try to strike a balance between these two extremes by precomputing and efficiently storing all of the relevant coefficients for the smoothing equations, and then combining them at inference time to compute the required n-gram probability.

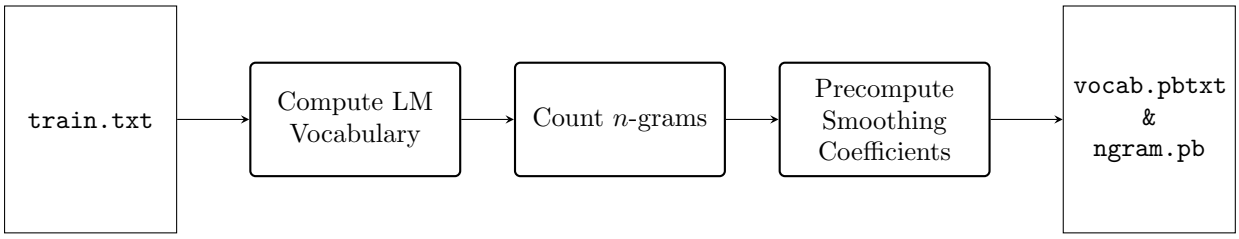


Figure 3.3: The n-gram training pipeline.

The three stages of my n-gram training pipeline are outlined in sections 3.2.1, 3.2.2 and 3.2.3 respectively.

3.2.1 Computing the Vocabulary

The first stage involves a single pass over `train.txt`, where words that occur at least k times are added to the vocabulary and mapped to an integer ID. k is called the *minimum frequency*. A special marker `<unk>` is included in the vocabulary to represent any out-of-vocabulary words. That is, when the model encounters a word it has not seen before, it treats it as `<unk>`. The marker `<s>` is also included and is used to denote the end of a sentence.

3.2.2 Counting n-grams Efficiently

For n-gram models without any smoothing, the only information that needs to be computed at this stage is $c(w_{i-n+1}^{i})$ and $\sum_w c(w_{i-n+1}^{i-1}w)$, which are the number of times each n-gram w_{i-n+1}^{i} occurred in `train.txt`, and the sum of all such counts for all words that follow the $(n-1)$-gram w_{i-n+1}^{i-1} respectively. Then, when the n-gram model is queried for the probability of w_{i-n+1}^{i}, it can simply return the first number divided by the second, as required in equation 2.3.

This situation becomes more complex once smoothing techniques are included. For instance, absolute discounting requires the knowledge of $N_{1+}(w_{i-n+1}^{i-1}\bullet)$, the number of unique words that follow w_{i-n+1}^{i-1}, for every such $(n-1)$-gram. Moreover, because it employs backoff, it actually requires this knowledge for every k-gram where $1 \leq k < n$. Kneser-Ney smoothing additionally requires $N_{1+}(\bullet w_{i-k+1}^{i})$, the number of unique words that precede w_{i-k+1}^{i}, and $\sum_w N_{1+}(\bullet w_{i-k+1}^{i}w)$, the number of unique pairs of words that follow and precede w_{i-k+1}^{i}, for every k-gram where $1 \leq k < n$.

In my implementation, I precompute and store all of these parameters in a temporary word-level trie, as shown in figure 3.4. This way, each parameter is only ever computed once and is accessible in $O(n)$ time. Given that I only consider n-grams for $n = 2$, 3, 4 or 5, the access time practically constant.

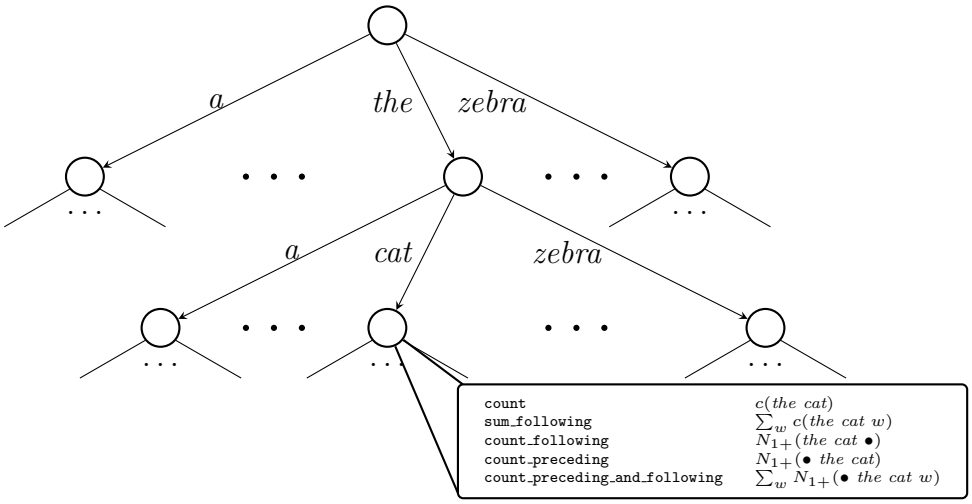


Figure 3.4: An example instance of the `CountTrie` class, which is the end result of the 'Count n-grams' stage in figure 3.3.

Populating a `CountTrie` instance works in two phases. In the first phase, all of the k-grams, where $1 \leq k \leq n$, from the training set are inserted into the trie. At every insertion, the relevant `count` and `count_preceding` values are updated. The second phase involves a full traversal of the trie in which the `sum_following`, `count_following` and `count_following_and_preceding` values are computed. Each of these values can be derived for a particular node by looking at the `count` or `count_preceding` values of its children.

It is worth noting that this data structure is temporary. That is, it is only used to enable fast computation of the smoothing coefficients in the next stage.

3.2.3 Precomputing Smoothing Coefficients

With a few exceptions, most smoothing techniques can be generalised to the following form:

$$\mathbb{P}(w_i|w_{i-n+1}^{i-1})_{\text{SMOOTH}} = \alpha(w_{i-n+1}^{i}) + \beta(w_{i-n+1}^{i-1})\mathbb{P}(w_i|w_{i-n+2}^{i-1})_{\text{SMOOTH}} \tag{3.1}$$

The probabilities of many different n-grams share some of the same α and β coefficients. For example, $\mathbb{P}(\textit{sat} \mid \textit{the cat})_{\text{SMOOTH}}$ and $\mathbb{P}(\textit{jumped} \mid \textit{the cat})_{\text{SMOOTH}}$ would both require $\beta(\textit{the cat})$. To avoid duplicated computation, the α and β values for every k-gram in the `CountTrie` instance are computed exactly once and stored in a new word-level trie, called `ProbTrie`. Each node in this trie stores two values, $\alpha(w_i^j)$ and $\beta(w_i^j)$, where w_i^j is the path of words that lead to the node. This way, the parameters can be fetched in $O(n)$ time at n-gram model query time, which again is practically constant since I only use $n \leq 5$. The `CountTrie` instance supplies the necessary parameters for deriving the smoothing coefficients used to populate the `ProbTrie` instance.

Once the `ProbTrie` instance has been populated, computing the probability of any n-gram can be done by traversing the data structure as shown in algorithm 1. Using this implementation, the only thing that each smoothing subclass has to override is the way in which the coefficients α and β are calculated when the `ProbTrie` instance is initially populated. This made it much easier to add and test new smoothing techniques.

Algorithm 1 Computing $\mathbb{P}(w_i|w_{i-n+1}^{i-1})_{\text{SMOOTH}}$

```
1:  procedure PROB(w_i, w_{i-n+1}^{i-1})
2:      if length(w_{i-n+1}^{i}) > 0 then
3:          node ← GETNODE(w_{i-n+1}^{i})
4:          α ← 0
5:          β ← 1
6:          if node then
7:              α ← node.α
8:          if length(w_{i-n+1}^{i-1}) > 0 then
9:              b_node ← GETNODE(w_{i-n+1}^{i-1})
10:             if b_node then
11:                 β ← b_node.β
12:         return α + (β · PROB(w_i, w_{i-n+2}^{i-1}))
13:     return 0
```

Katz smoothing is an exception in that it does not quite match the form presented in equation 3.1. To get around this issue, the Katz subclass simply overrides algorithm 1 to traverse the same data structure in a slightly different way.

Once the `ProbTrie` instance is populated, it is exported to a protocol buffer in binary format, `ngram.pb`. The vocabulary is exported to a protocol buffer in textual format, `vocab.pbtxt`.

3.3 Recurrent Neural Network Models

Below, I explain how I implemented my RNN-based language models. To do this, I first give a brief overview of TensorFlow in section 3.3.1. Then, in section 3.3.2, I explain the structure of the RNNs I use and how they apply to language modelling. In section 3.3.3 I give details of the GRU and LSTM architectures. Section 3.3.4 describes the word embeddings used in the models. Finally, section 3.3.5 outlines how I tuned the RNN hyperparameters and applied techniques to encourage convergence and avoid overfitting.

3.3.1 TensorFlow

TensorFlow [19] is an open-source machine learning library that I used to implement my RNN-based language models.

Computation in TensorFlow is defined in terms of a directed graph, where each node represents an operation and each directed edge represents the flow of a tensor, hence the name. TensorFlow programs consist of two key sections: the first is constructing the graph, and the second is executing one or more paths through the graph in what is called

a *session*. In order to execute a path through a graph, you specify one or more nodes to evaluate, and then TensorFlow automatically executes all operations that are required to evaluate the tensor(s) at the specified node(s).

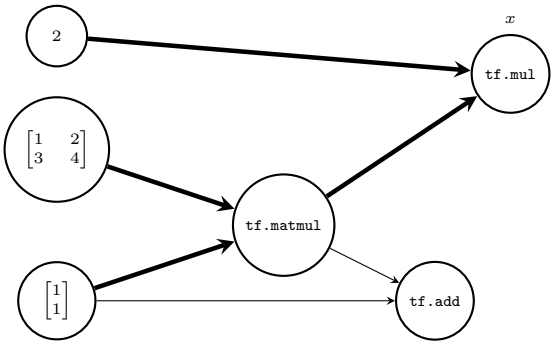


Figure 3.5: A toy TensorFlow computation graph. On evaluation of node x, only the operations on the highlighted paths are computed, and 20 (a rank-0 tensor) is returned.

Each operation takes zero or more tensors as input, and produces a tensor as output. There are some notable operations that take zero inputs: a *constant*, which outputs a tensor whose value cannot be changed; a *variable*, which is the same except the tensor that it outputs can be changed; and a *placeholder*, which is a promise to provide a tensor at execution time.

In my implementation, I described my RNN-based language models as a directed graph of tensor operations. For example: layers of neurons were represented as a vectors and the weights in-between each layer were represented as a matrix. In the case of the vanilla RNN, I computed a layer of activations using the matrix operations presented in equation 2.30. I used a *placeholder* to provide the input training data, and *variables* to store all of the trainable parameters of the network, namely the weights, bias variables and the word embeddings.

Once I constructed my RNN, I ran forward propagation through the network by evaluating the output nodes of the network in a *session*. TensorFlow has a useful method called `tf.gradients(ys, xs)`, which returns the partial derivatives of the sum of the tensor(s) in `ys` with respect to the variables in `xs`. This is known as *automatic differentiation*. By setting `xs` to the trainable parameters of the network and `ys` to the loss function, I avoided having to hard-code the backpropagation computation. Using these partial derivatives, I then applied an optimisation technique such as gradient descent to update the trainable parameters of the network.

After having trained the RNN in Python, I would *freeze* the computation graph and export it into a format that can be loaded into C++. This involved converting all of the variables in the graph to constants and saving the network, along with some metadata, into a protocol buffer. Using the C++ API, I could then load the network and run inference on it by evaluating the relevant tensors in a session, without having to know anything about the internal structure of the RNN.

3.3.2 Network Structure

The general structure of all of my RNN-based language models is described below.

Before the RNN is constructed, an initial pass is made over the training set to construct a vocabulary V. This is done in exactly the same way as for the n-gram models, as described in section 3.2.1, so I will not repeat the detail.

At the very start of the network is a word embedding lookup table, that maps words to vectors that represent points in a high dimensional space. In my implementation, I set the number of vector elements in each word embedding to be the same as the number of hidden neurons in each hidden layer of the RNN. This is a configurable parameter, H, which was typically set to 256.

After the embedding lookup is the input layer of the RNN into which the word embeddings are fed. There are then two hidden layers which are recurrently connected to themselves, followed by the output layer.

There are $|V|$ neurons in the output layer, each of which represent a word in the language model vocabulary. Rather than feeding the logits of the output neurons through an activation function, they are passed through the softmax function. That is, if the vector of logits of the output neurons is denoted $\mathbf{z}$, then the output of neuron j after the softmax function is defined as $\textsc{Softmax}(\mathbf{z})_j$, where:

$$\textsc{Softmax}(\mathbf{z})_j = \frac{e^{z_j}}{\sum_i e^{z_i}} \tag{3.2}$$

The outputs of the softmax function sum to 1 and are used to represent a probability distribution over all of the possible next words. That is, $\mathbb{P}(\text{next word} = j) = \textsc{Softmax}(\mathbf{z})_j$, where j is the ID of a particular word in the vocabulary. Each ID lies in the range $[0, |V|)$.

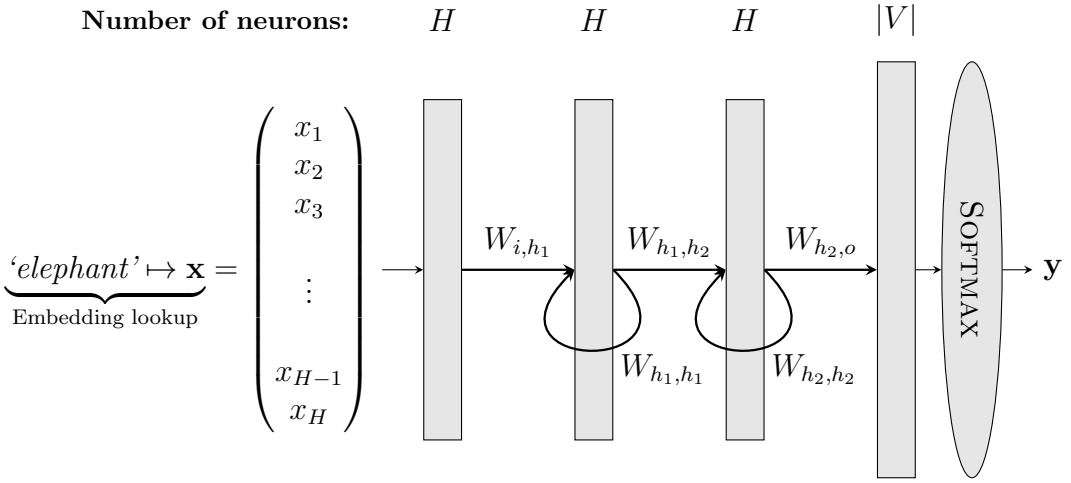


Figure 3.6: The structure of my RNN-based language models.

3.3.3 Long Short-Term Memory and Gated Recurrent Units

As mentioned in section 2.2.2, there are various architectures that can be used for the neurons of RNNs, three of which I implemented in this project. Under the vanilla RNN

architecture, a layer of neuron activations is computed as shown in equation 2.30, which is repeated below. The fact that the vanilla RNN keeps hold of the activations of the hidden layers from the previous time step means that it implicitly maintains some state:

$$\mathbf{a}^{l,t} = \phi\big(\mathbf{W}^{l-1}\mathbf{a}^{l-1,t} + \mathbf{V}^{l}\mathbf{a}^{l,t-1} + \mathbf{b}^{l-1}\big)$$

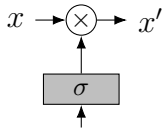
Unfortunately, it is difficult to train vanilla RNNs to capture long-term dependencies, because the influence of the inputs as they cycle through the recurrent connections in the network often either decay or blow up exponentially [20] [21], resulting in the weight gradients either vanishing or exploding disproportionately. These problems are known as the *vanishing gradient problem* and the *exploding gradient problem* respectively. Various RNN architectures have been constructed to improve upon these problems, two of which I implemented in my project and are outlined below.

Long Short-Term Memory

Hochreiter and Schmidhuber [17] originally proposed an architecture that directly tackles the vanishing gradient problem, called Long Short-Term Memory (LSTM). The state of each hidden layer l in their architecture at time step t consists of a tuple $(\mathbf{a}^{l,t}, \mathbf{c}^{l,t})$. $\mathbf{a}^{l,t}$, just like in the vanilla RNN cells, allows the LSTM cells to make decisions over a short period of time. The second state vector, $\mathbf{c}^{l,t}$, serves to retain longer-term information. At time step $t+1$, each element $c_j^{l,t}$ of $\mathbf{c}^{l,t}$ passes through a single LSTM cell in layer l, and can be modified by a series of three *gates*: the *forget gate*, the *input gate* and the *output gate*. These gates can remove, add and output information respectively from $c_j^{l,t}$.

A scalar operation: $y = x_1 \bullet x_2$ | A neural network layer: $y = \phi\big([\mathbf{W}\mathbf{x}]_j + b_j\big)$ | Concatenation of two vectors. | Duplication of a vector or scalar.

In what follows, I use the notation from figure 2.6, which is repeated above for convenience. Each LSTM gate has the following structure:



Each gate controls how much information from x is let through to x'. The sigmoid layer outputs a scalar value in the range $[0, 1]$, where 0 prevents any information from x passing through and 1 allows all of it to pass through.

The LSTM forget, input and output gates in cell j of layer l at time step t are denoted $f_j^{l,t}$, $i_j^{l,t}$ and $o_j^{l,t}$ respectively. If $[\mathbf{x}_1, \mathbf{x}_2]$ denotes the concatenation of vectors $\mathbf{x}_1$ and $\mathbf{x}_2$, and $[\mathbf{x}]_j$ denotes the j^{th} element of vector $\mathbf{x}$, then the gate values are computed as follows:

$$f_j^{l,t} = \sigma\big([\mathbf{W}_f^l[\mathbf{a}^{l,t-1}, \mathbf{a}^{l-1,t}]]_j + [\mathbf{b}_f]_j\big) \tag{3.3}$$

$$i_j^{l,t} = \sigma\big([\mathbf{W}_i^l[\mathbf{a}^{l,t-1}, \mathbf{a}^{l-1,t}]]_j + [\mathbf{b}_i]_j\big) \tag{3.4}$$

$$o_j^{l,t} = \sigma\big([\mathbf{W}_o^l[\mathbf{a}^{l,t-1}, \mathbf{a}^{l-1,t}]]_j + [\mathbf{b}_o]_j\big) \tag{3.5}$$

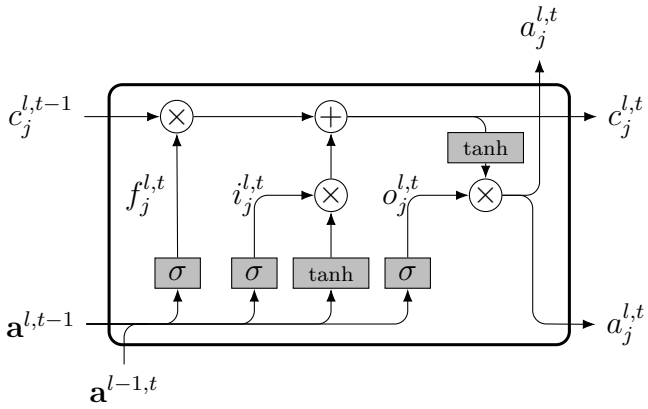


Figure 3.7: A summary of the structure of an LSTM cell. Specifically, this diagram represents the computation of the j^{th} cell in layer l at time step t.

Firstly, the forget gate determines how much of the long-term state value, $c_j^{l,t-1}$, should be 'forgotten'. Next, there is the opportunity for information derived from the short-term state, $\mathbf{a}^{l,t-1}$, and the previous layer activation, $\mathbf{a}^{l-1,t}$, to be added to the long-term state. The input gate decides how much of this information should be added. The application of the forget and input gates gives the new value of the long-term state value, $c_j^{l,t}$:

$$c_j^{l,t} = f_j^{l,t} \cdot c_j^{l,t-1} + i_j^{l,t} \cdot \tanh\left([\mathbf{W}_c^l[\mathbf{a}^{l,t-1}, \mathbf{a}^{l-1,t}]]_j + [\mathbf{b}_c]_j\right) \tag{3.6}$$

Finally, the output gate is used to control how much of the long-term state is used for the cell output, $a_j^{l,t}$:

$$a_j^{l,t} = o_j^{l,t} \cdot \tanh(c_j^{l,t}) \tag{3.7}$$

This structure is summarised in figure 3.7, and is the version of LSTM that I implemented in my project. There are many variants of LSTM, but I chose this one in particular because it represents a good baseline for the architecture. That is, most of the other versions are some form of extension of the one presented above, such as the notable use of *peephole connections* from Gers and Schmidhuber [22].

Gated Recurrent Unit

Whilst the LSTM architecture provides a significant improvement for retaining long-term information, it is computationally expensive to run. Cho et al. propose the Gated Recurrent Unit [18], which is a simplified version of LSTM that is cheaper to compute. The most significant changes are that it uses two gates rather than three, and it merges the long and short-term states into one.

The GRU consists of a *reset gate*, $r_j^{l,t}$, and an *update gate*, $u_j^{l,t}$:

$$r_j^{l,t} = \sigma\left([\mathbf{W}_r^l[\mathbf{a}^{l,t-1}, \mathbf{a}^{l-1,t}]]_j\right) \tag{3.8}$$

$$u_j^{l,t} = \sigma\left([\mathbf{W}_u^l[\mathbf{a}^{l,t-1}, \mathbf{a}^{l-1,t}]]_j\right) \tag{3.9}$$

The cell output, $a_j^{l,t}$, is then computed as follows:

$$a_j^{l,t} = u_j^{l,t} \cdot a_j^{l,t-1} + (1 - u_j^{l,t}) \cdot \tilde{a}_j^{l,t} \tag{3.10}$$

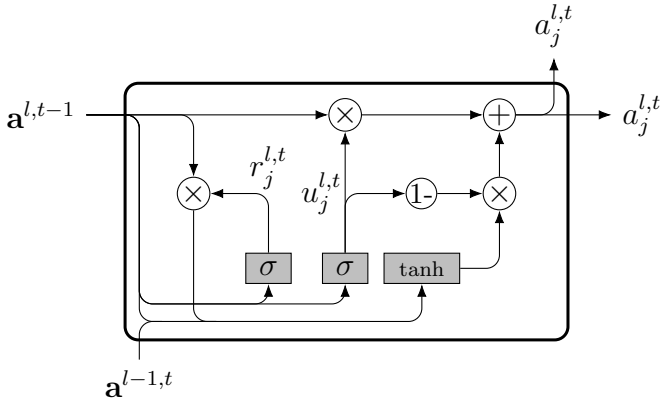


Figure 3.8: A summary of the structure of a GRU cell. Specifically, this diagram represents the computation of the j^{th} cell in layer l at time step t.

where

$$\tilde{a}_j^{l,t} = \tanh\left([\mathbf{W}_a^l[\mathbf{a}^{l,t-1} \odot \mathbf{r}^{l,t}, \mathbf{a}^{l-1,t}]]_j\right) \tag{3.11}$$

and $\odot$ represents element-wise multiplication.

The reset gate acts similarly to the LSTM forget gate, in that it allows the cell to drop information that it deems irrelevant. The update gate controls how much information from the previous state carries over to the new one. With the short-term and long-term states combined into one, it follows that cells either learn to capture short or long-term information. Those that capture short-term information tend to have more active reset gates, whereas those that capture long-term information tend to have more active update gates.

Abstraction Over the RNN Cells

The LSTM architecture maintains 2-tuples of state vectors, whereas the vanilla RNN and GRU architectures only maintain 1-tuples. Apart from this, the three RNN cells only differ in how they are connected internally. I exploited this in my code by abstracting the implementation of each RNN cell from the overall network structure that was presented in section 3.3.2, so that new cell architectures could easily be added or tested.

3.3.4 Word Embeddings

As mentioned in section 3.3.2, each input word is first mapped to a vector representation known as a *word embedding* which is learned as the network is trained. The aim is that similar words will be assigned points in the high-dimensional space which are closer together. This way, the RNN-based language model can more easily generalise to combinations of words that it has not seen before, by understanding which words can be substituted for one another.

In order to visualise such word embeddings, I used principle component analysis (PCA) to reduce the word embeddings to two-dimensional points that can be plotted on a scatter diagram. PCA is a dimensionality reduction technique that transforms the input data to

a new coordinate system such that the first coordinate represents some projection that achieves the greatest variance, the second coordinate the second greatest variance, and so on.

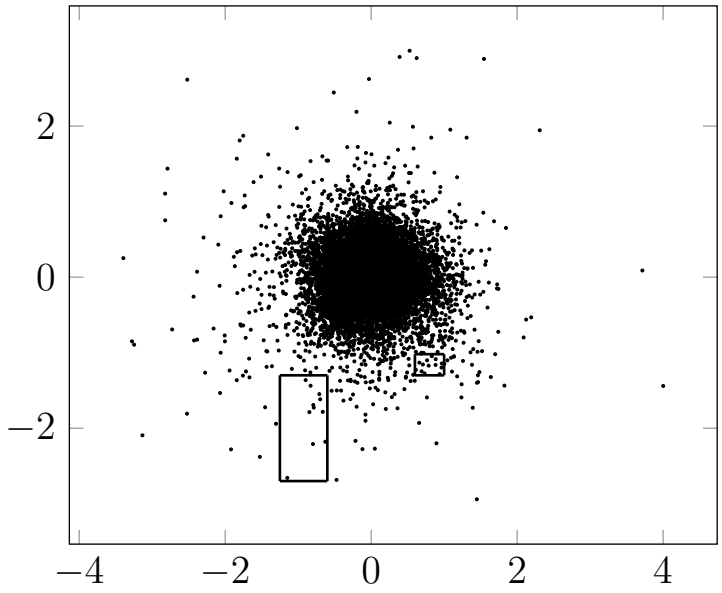
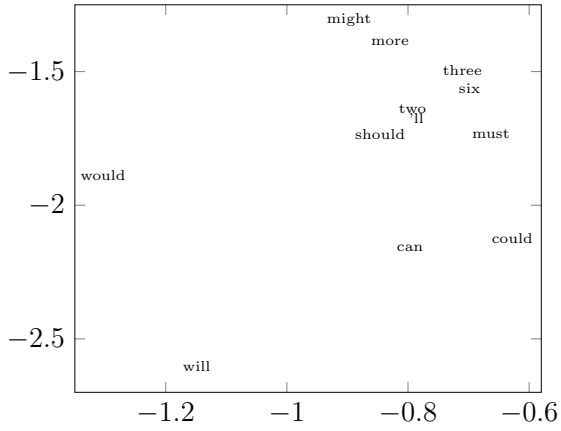
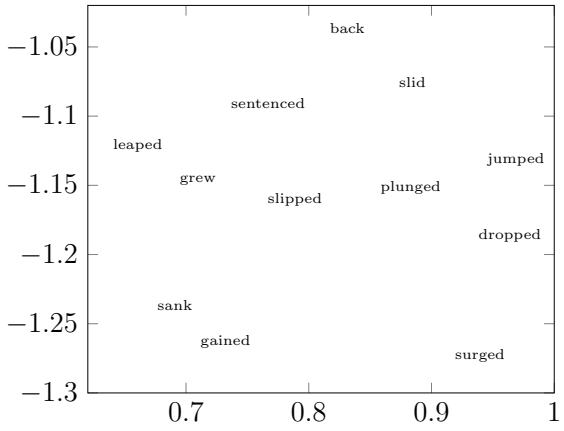


Figure 3.9: LSTM-based language model word embeddings projected onto two dimensions using PCA. The highlighted regions and are shown in figure 3.10.

The words that have been assigned a more distinguishable representation are given by the points which are further from the centre in figure 3.9.



(a) Words that express necessity, overlapping with a group of words that describe quantity.



(b) A group of words that characterise a dramatic change in value.

Figure 3.10: Two zoomed in regions of figure 3.9.

3.3.5 Putting Theory into Practice

In this section, I address a number of problems that are faced when training RNNs in practice and describe how I went about mitigating them.

Underfitting and Overfitting

In any supervised machine learning problem, there are one or more underlying patterns in the training data that need to be learned. *Underfitting* occurs when the model fails to capture these patterns, and is typically the case when the model is too simple for the training data, or it is not trained for long enough. This problem was easy to mitigate by adding more parameters to the model, such as the number of hidden neurons, and training it for a sufficient amount of time.

As well as underlying patterns in the training data, there also often exists some noise which you do not want the model to learn. *Overfitting* occurs when the model learns this noise, which makes it difficult for it to generalise to new data that it has not seen before. When each of my models were trained, a portion of the data called the test set was reserved for evaluating the performance of the model. The model was not allowed to see any of the data from the test set during training, so that its ability to generalise to unseen data was fairly assessed. In order to avoid overfitting my RNN-based language models, I also set aside a portion of data called the validation set. I used this to monitor, during training, the performance of my models on unseen data. As soon as this performance plateaued, I stopped the training process.

In my implementation, I evaluate the perplexity[1] of my language model on the validation set at the end of each training iteration. The learning rate starts at some initial value, η_0, and is then halved on every new iteration once the improvement of the perplexity on the validation set between two iterations drops below some threshold, Δ. After this point, when the improvement of the perplexity on the validation set drops below 0.1%, the training process is halted. I tried various values for η_0 and Δ, and chose the best ones for each RNN architecture: $\eta_0 \in \{1, 0.5, 0.1, 0.01, 0.001\}$, $\Delta \in \{10\%, 5\%, 1\%, 0.5\%\}$.

RNNs with a large number of parameters can be particularly susceptible to overfitting. Srivastava proposes a technique called *dropout* for reducing this problem [23], whereby some of the neurons and their connections are randomly dropped from the network during training. Zaremba et al. found that dropout works best in RNNs if it is only applied to the non-recurrent connections [24], which is the method I used in my RNN implementation.

Convergence

Whilst the LSTM and GRU architectures directly combat the aforementioned vanishing gradient problem, there is still the possibility that the gradients can explode, which can make the network unstable during training. In my implementation, I used *gradient clipping* to mitigate this issue, whereby the gradients are clipped once their norm exceeds a certain threshold.

The selection of the initial learning rate value, η_0, was also important for ensuring convergence in my RNNs. If the learning rate is too large, then the weight updates may overshoot the minimum which they are directed towards. On the other hand, learning rates that are too small result in very slow convergence or the network becoming stuck in local minima.

[1] Perplexity is the standard performance measure for language models, detailed in section 4.1.1.

3.4 Mobile Keyboard

As a project extension, I decided to apply one of my RNN-based language models in a practical context by building a mobile keyboard.

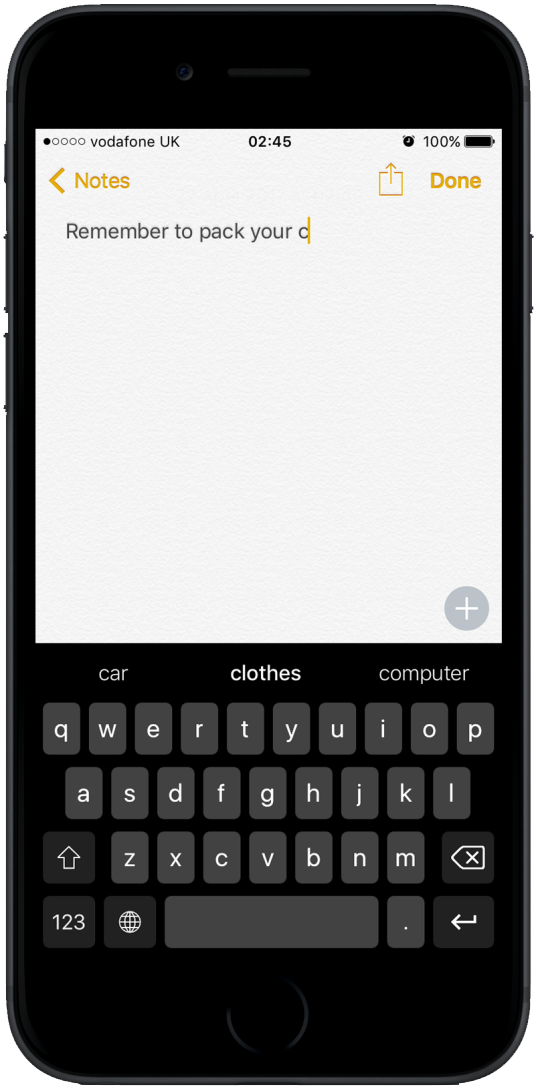


Figure 3.11: The mobile keyboard on iOS. The top three predictions are displayed at the top of the keyboard, and are updated with every character.

iOS applications are written in either Objective-C or Swift. An advantage of using Objective-C is that you can make use of Objective-C++ to link C++ code into your application, which is exactly what I needed to do. My C++ code, however, depended on TensorFlow, so in order to make my language models accessible within the mobile keyboard, I compiled both my RNN-language model implementation and a subset of TensorFlow's C++ API into a single static library by modifying one of the makefiles already present in the TensorFlow codebase.

To build a mobile keyboard in iOS, you have to implement what is called an 'App Extension'. App Extensions differ from normal applications in iOS in that they their functionality is system-wide. Apple insist that such ubiquitous components should be extremely

responsive and light-weight, which means that they have much stricter memory and CPU limitations than normal applications. Although they do not specify exactly what these limitations are, many people in the developer community estimate that the memory limit is approximately 30 MB.

In order to get an RNN-based language model to run under such memory and CPU limitations, I had to trade-off some of its prediction accuracy. In particular, I reduced the number of hidden layers, the number of hidden neurons and the size of the vocabulary to cut down both the CPU and memory pressure from the model at run time. The final model I used was a 1-layer vanilla RNN with 32 hidden neurons. I also removed the softmax layer, which is technically is not needed to rank the predictions, to further reduce the CPU overhead.

3.4.1 Updating Language Model Predictions On the Fly

A mobile keyboard should present the user with next-word predictions that are updated as they type. In my implementation, I model the state of the input text as a tuple (W, S, C), where W, S and C represent a sequence of words, spaces and non-space characters respectively. For example, the text in figure 3.11 would be represented as:

$$\underbrace{\texttt{Remember␣to␣pack␣your}}_{W}\underbrace{\texttt{␣}}_{S}\underbrace{\texttt{c}}_{C}$$

With every new character insertion or deletion, this state is implicitly updated. The interaction with the language model as a response to each insertion or deletion is illustrated in figure 3.12.

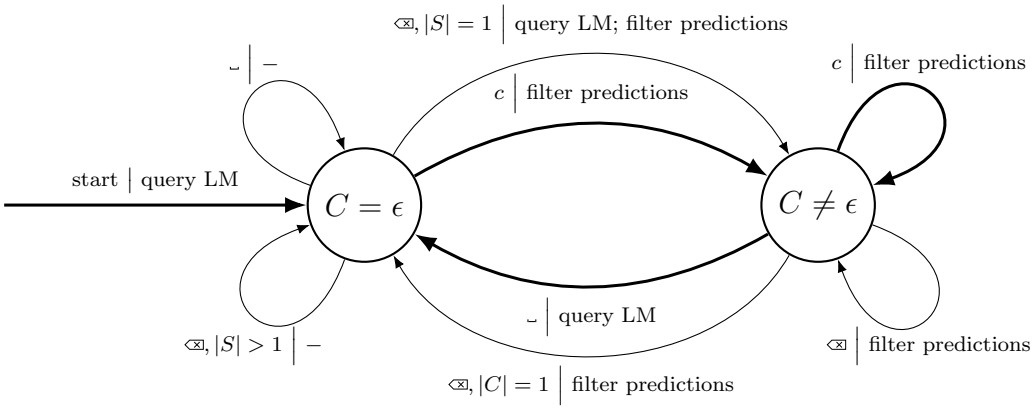


Figure 3.12: An FSM representing the language model interaction. The thick arrows denote the path taken when the user is inserting text without any deletions or unnecessary spaces.

c, ␣ and ⌫ represent a non-space character insertion, a space insertion and a backspace respectively. 'Query LM' represents the process of querying the language model with W. The language model vocabulary is stored in a character-level trie, and the probabilities for each word from the query result are inserted at the nodes of the trie. 'Filter predictions' represents the process of traversing this trie using C and taking the top three most probable words on the current branch. This approach finds the most probable next-word

predictions that begin with the characters that the user has already typed for the current word, whilst avoiding unnecessary language model queries.

As well as the basic input events illustrated in figure 3.12, my implementation also caters for: the shift key, caps lock, dragging the cursor to a new position, switching between letters and special characters, selecting displayed predictions and auto-capitalisation of new sentences.

3.5 Extending Models to Tackle Error-Prone Text

An issue with existing language models is that their predictions suffer when presented with error-prone text. For example, consider the two word sequences '*the weather is*' and '*the whether is*'. Whilst in the first case a language model might successfully predict words such as '*sunny*' or '*nice*', in the second case it will likely struggle to suggest reasonable word candidates. Evidence to support this claim is given in figure 4.8 in section 4.2.3.

In the context of text prediction, it is unfortunately very rare that a human user will not make any mistakes. It is therefore an important issue to address and in this section I propose an extension to a language model which aims to improve its performance on text containing errors.

3.5.1 The Approach

My approach is to attack the source of the problem directly by adding a layer of error correction between the input text and the language model. There are lots of different types of error that can occur in the English language, but my implementation only focuses on correcting spelling mistakes.

Error-correction can be broken up into two parts: detecting the errors and making suggestions to fix them. In the case of spelling mistakes, detection can be done with a single dictionary lookup. Candidate corrections for a spelling mistake should be similar to the original word and probable in the surrounding context. In my implementation, I use the Levenshtein distance as a measure of how similar two words are, which is the minimum number of single character insertions, deletions and/or substitutions that need to be made to get from one word to the other. I also use the language model itself to gauge how probable candidate corrections are in the surrounding context. Specifically, when searching for candidate replacements for a word w that is preceded by the words w_i^j, I feed w_i^j to the language model, iterate through the next-word predictions in order of decreasing probability and select the first word that has an edit distance within some threshold δ of the original spelling, as described in algorithm 2.

I implemented this approach as an extension to my RNN-based language models, hence the use of FeedForward() and *state* in algorithm 2.

Algorithm 2 Computing $\mathbb{P}(w_j|w_i^{j-1})_{\text{ERROR\_CORRECTING}}$

```
1: procedure Prob(w_j, w_i^{j-1})
2:     outputs ← ∅
3:     state = 0
4:     for w in w_i^{j-1} do
5:         w_correct ← w
6:         if w ∉ dictionary and outputs ≠ ∅ then
7:             SortByDecreasingProbability(outputs)
8:             for w′ in outputs do
9:                 d ← LevenshteinDistance(w, w′)
10:                if d ≤ δ then
11:                    w_correct ← w′
12:                    break
13:        outputs, state ← FeedForward(w_correct, state)
14:    return outputs[w_i]
```

Chapter 4

Evaluation

In this chapter, I 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 present the performance of the existing language models that I implemented, secondly I focus on the tradeoffs faced when employing those models on a mobile device, and finally I display my findings in language modelling on error-prone text.

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:

Q1. How accurately does the language model predict text?

Q2. How much resource, such as CPU or memory, does the language model consume?

I implemented a generic benchmarking framework that can return a series of metrics which are outlined below. The first two are concerned with **Q1** and the latter three relate to **Q2**.

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}. This somewhat arbitrary-looking formulation is more thoroughly justified with a bit of information theory in appendix B. The key point is that *lower values of perplexity indicate better prediction accuracy* for language models trained on a particular training set.

One issue with perplexity is that it is undefined if $\mathbb{P}_L(w_i|w_1^{i-1})$ is 0 at any point. 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

Average-keys-saved is a more user-oriented metric, which is defined as the number of characters that they can avoid typing as a result of the correct next words 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 ensures that the results are not biased by having longer words in the test data.

Memory Usage

This is measured as the amount of physical memory in megabytes occupied by the process running the language model.

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

Penn Tree Bank (PTB) Dataset

The Penn Tree Bank is a widely-adopted dataset for measuring the quality of language models, derived from text from the Wall Street Journal. The training set has 10,000 unique words and 887,521 words overall. 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 examination in 2000 and 2001 [25]. In this project I make use of a preprocessed version of this dataset, in which

error-prone and error-free versions of the exam scripts are aligned line by line in separate files. 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.

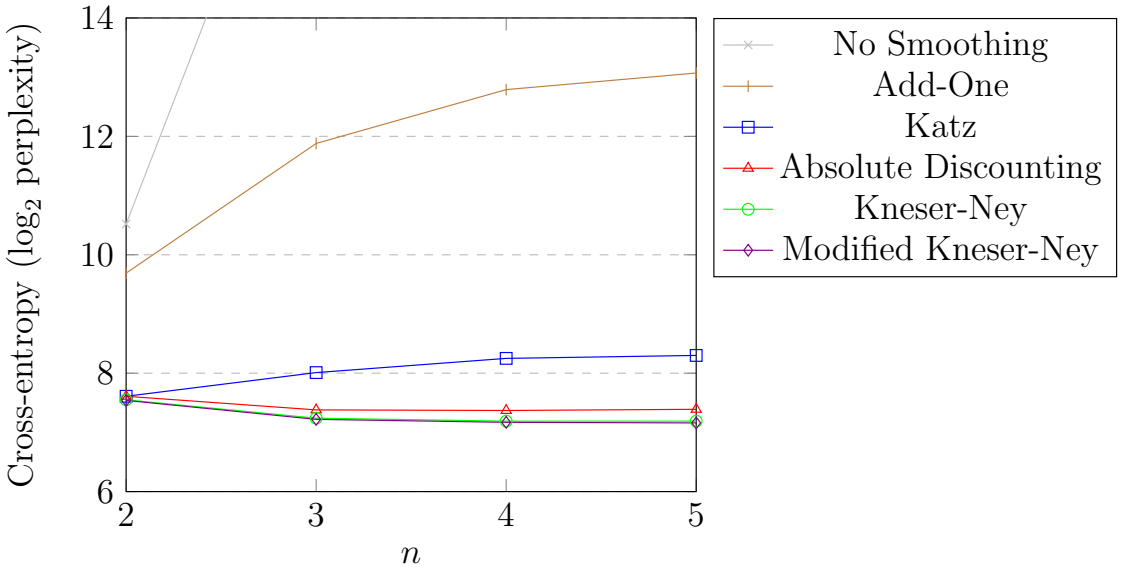


Figure 4.1: The cross-entropy of n-gram models trained on the PTB dataset.

Cross-entropy is the binary logarithm of perplexity, and 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, because one might expect that increasing n will always yield better results. For n-gram models with add-one or no smoothing, this is not the case, because they do not employ backoff. At higher values of n, n-grams are much more sparse, so without any backoff these 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, which is why its performance also drops with n.

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

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

money is the root of a new york city.
the meaning of life is a unit of the company's shares outstanding.
science and was up N N from $ N million or N cents a share.

(a) 5gram + modified Kneser-Ney smoothing

money is the root of the nation's largest public bank.
the meaning of life is the best way to get the way to the public.
science's most recent issue of the nation's largest economic indicators.

(b) 2-layer LSTM with 256 hidden neurons

Figure 4.2: Sentences generated from two different language models trained on the PTB dataset. The bold words were given to the model, and the non-bold words were generated until a full stop was predicted.

At a qualitative level it can be seen that the RNN-based language models are better at making predictions with long-term word dependencies than the n-gram smoothed models. This is demonstrated in figure 4.2, and confirmed by the readings in figure 4.3.

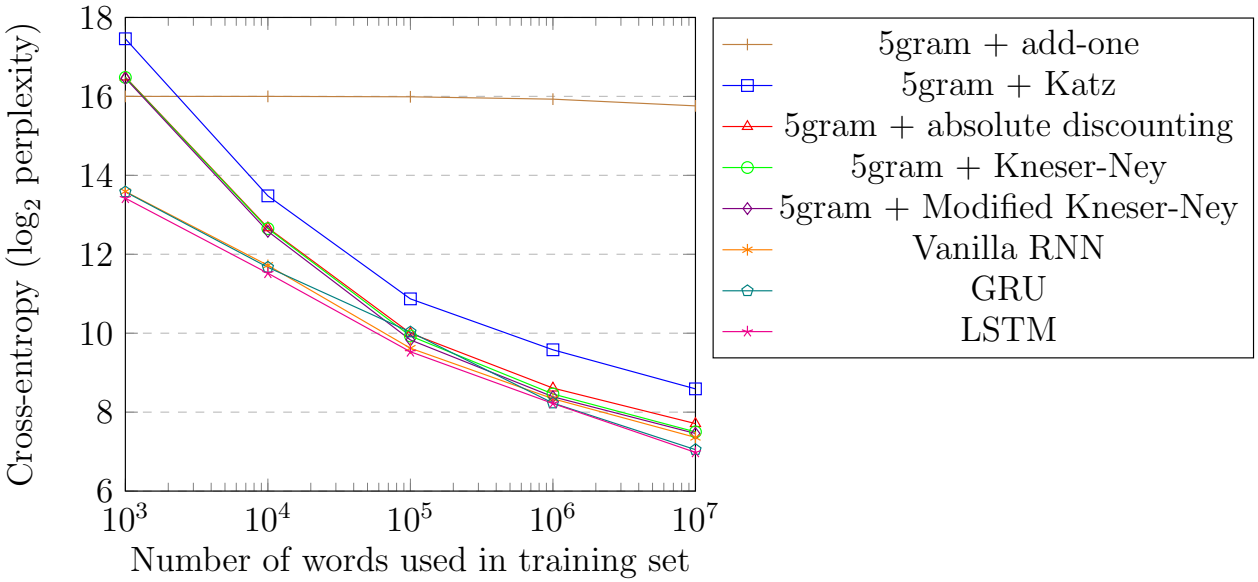


Figure 4.3: The cross-entropy of various language models with respect to the training set size, using the 1BW dataset.

As can be seen in figure 4.3, the RNN-based language models consistently outperform the n-gram based models in terms of prediction accuracy. This difference is most significant for small training sets, and starts to become more pronounced again once the training set grows large beyond 10^6 words.

Some datasets are more predictable than others, which means that metrics like perplexity and average-keys-saved depend on which dataset is used. In order to produce results that are comparable with the work of other authors, I have tested my language models on the Penn Tree Bank dataset, as shown in figure 4.4. The RNN-based models used 2 hidden layers each with 256 neurons. The perplexity was calculated over the whole of the test set, whereas average-keys-saved, a more expensive metric to compute, was taken over the first 1000 words of the test set.

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. † 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.*

Figure 4.4: A benchmark of various language models on the PTB dataset.

There is a lot of information that can be drawn from figure 4.4, so I will highlight only the most important and interesting points:

- The RNN-based language models outperform the n-gram models in terms of prediction accuracy. In my implementation, they run slightly faster at inference time and also consume less memory, although this is partly due to the fact that the n-gram models are entirely unpruned. The RNN-based language models, however, take significantly longer to train and obtain the correct hyperparameters for.

- Ignoring insignificant differences in memory and time, absolute discounting, Kneser-Ney smoothing and modified Kneser-Ney smoothing equal or outperform add-one and Katz smoothing in every metric.

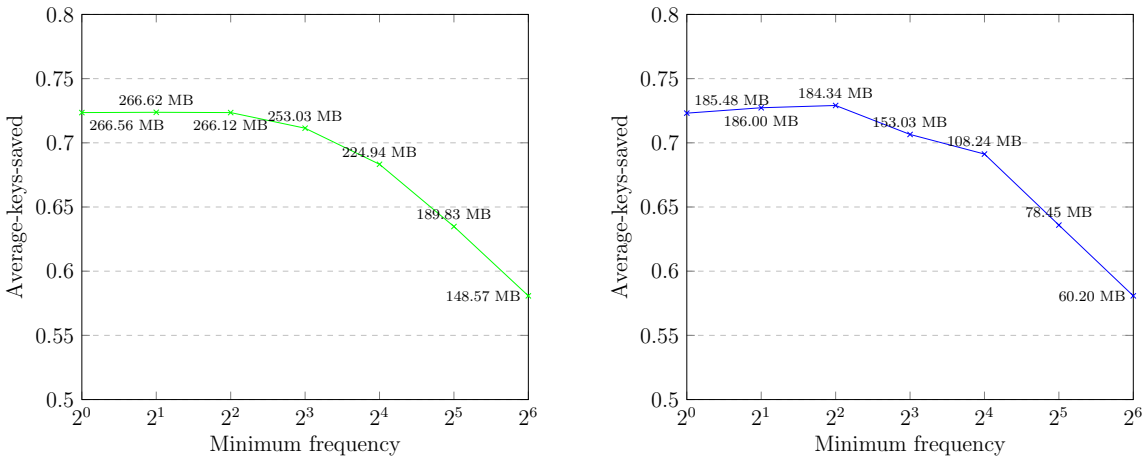
- LSTM yields marginally better prediction accuracy than the GRU architecture, but requires more memory. Apart from the memory overhead, GRUs significantly outperform the vanilla RNN architecture. The differences in training time for the RNN-based models should be taken lightly, because each model was trained for a different number of iterations, depending on how long it took to converge to a steady perplexity score on the validation set.

- Interestingly, a substantial improvement in perplexity and average-keys-saved can be obtained by simply averaging the probabilities produced by an n-gram and an RNN-based language model. This seems to imply that the two classes of language model complement one another, in the sense that RNN-based models make strong predictions when some of the n-gram predictions are weak, and vice versa.
- The combined language models can be improved even more by interpolating their probabilities rather than just averaging them. The result shown for the interpolation of 5-gram modified Kneser-Ney with LSTM used $\lambda = 0.38$, where the probability was calculated as λ(n-gram probability) + (1 - λ)(LSTM probability).

4.2.2 On a Mobile Device

The strict memory and CPU limitations I faced when implementing the mobile keyboard forced me to explore the tradeoffs between resource consumption and prediction performance.

Two obvious ways of decreasing the memory footprint of a language model are decreasing the vocabulary size and, for RNN-based models, decreasing the number of hidden neurons. What is less obvious, is how quickly the average-keys-saved drops with these two parameters.



(a) 3gram + modified Kneser-Ney smoothing (b) Vanilla RNN with 256 hidden neurons

Figure 4.5: The effect of the vocabulary size on average-keys-saved and memory usage for models trained on the PTB dataset.

The vocabulary size was altered by changing a parameter called the *minimum frequency*, which defines the minimum number of times a word must appear in the training set to be considered part of the language model vocabulary.

For both the n-gram and vanilla RNN models, the minimum frequency can be increased to approximately 2^4 before the average-keys-saved starts dropping rapidly. At this point, a saving of 77.24 MB is made in the vanilla RNN model, with a drop of only 0.03187 in average-keys-saved. The savings made on the n-gram model are less substantial.

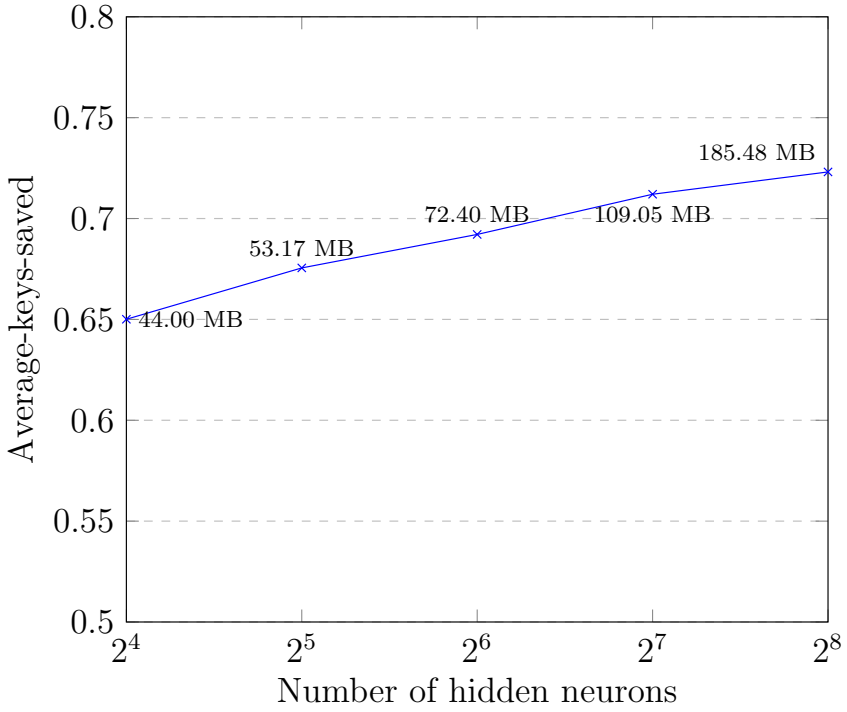


Figure 4.6: The effect of the number of hidden neurons on average-keys-saved and memory usage in a 2-layer vanilla RNN trained on the PTB dataset.

From figure 4.6, it can be seen that much greater reductions in memory usage can be achieved for a given loss in average-keys-saved by decreasing the number of hidden neurons. Dropping from 256 to 32 hidden neurons in each layer gives a memory saving of 132.31 MB, with an average-keys-saved loss of 0.04758. It is intriguing that a 2-layer vanilla RNN with only 32 hidden neurons in each layer can achieve a better average-keys-saved score than the 5-gram model with Katz smoothing from figure 4.4.

Of course, there are several other techniques that can be used to optimise the memory and CPU overhead of language models for mobile platforms. These include, but are not limited to: reducing the number of hidden layers; using 16-bit floats for model parameters; removing the softmax layer from the RNN, since it is not strictly needed in ranking predictions; and memory-mapping the model to reduce memory pressure when it is initially being loaded.

4.2.3 On Error-Prone Text

In this section I present my findings for the performance of language models on error-prone text. The goal is to produce a language model whose predictions are not deterred by errors in the input text, as illustrated in figure 4.7.

Normally, language models are evaluated by feeding them a sequence of input words (the *inputs*) and seeing how well they predict that same sequence of input words shifted forward in time by one word (the *targets*). When evaluating my error-correcting language models, I did exactly the same thing, except that the inputs were allowed to contain errors in them and the targets were not.

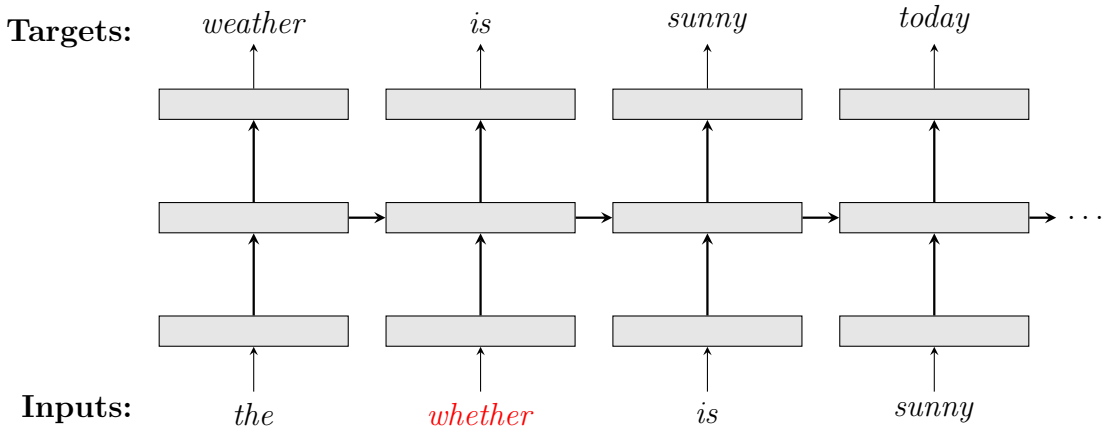


Figure 4.7: The ideal error-correcting language model.

I used the CLC dataset, which consisted of 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. In order to focus the evaluation on the error-correcting ability of my language models, I removed any pairs of lines from the test set files that were identical. I also removed any pairs of lines that contained corrections that involve the insertion or deletion of words, because my model was not designed to handle these types of errors, as explained in section 3.5.

The ideal error-correcting language model would predict words as if the input were not error-prone, therefore, to obtain an *upper bound* in performance, I evaluated the LSTM-based language model using `test.correct.txt` for the input words and `test.correct.txt` for the target words. In order to establish a *lower bound*, I evaluated the performance of the same model using `test.incorrect.txt` for the input words and `test.correct.txt` for the target words. This model did not contain any error-correcting capabilities.

Language model	**Perplexity**	**Average-keys-saved**
Upper bound	77.83	0.65483
Error-correcting LSTM; $\delta = 1$	87.99	0.64453
Error-correcting LSTM; $\delta = 2$	88.47	0.64376
Error-correcting LSTM; $\delta = 3$	88.57	0.64273
Lower bound	89.70	0.63912

Figure 4.8: The performance of my error-correcting language model with respect to the upper and lower bound, using the CLC dataset.

It can be seen from figure 4.8 that my error-correcting model improves the performance of language model predictions on text containing errors that do not require the insertion

or deletion of words. The best result arises from an edit distance threshold of 1, which seems attributed to the fact that a larger threshold allows for more false corrections, as shown in figure 4.9b.

Input	i am more than happy to give you the **nessessary** information.
Target	i am more than happy to give you the **necessary** information.
Prediction	i am more than happy to give you the **necessary** information.

(a) An accurate correction.

Input	the bus will pick you up right at your hotel **entery**.
Target	the bus will pick you up right at your hotel **entrance**.
Prediction	the bus will pick you up right at your hotel **every**.

(b) A false correction.

Input	**appart of** that, there is no recommendation as to what to wear.
Target	**apart from** that, there is no recommendation as to what to wear.
Prediction	**apart of** that, there is no recommendation as to what to wear.

(c) A missed correction.

Figure 4.9: Example corrections made by my error-correcting RNN with $\delta = 2$.

There is still a gap to be closed, which is demonstrated in figure 4.9c by the fact that my model does not correct non-spelling mistakes.

Chapter 5

Conclusions

The project was a success: I implemented a variety of n-gram and RNN-based language models, and benchmarked them in terms of both their prediction performance and their applicability to running on a mobile platform, as set out in my project proposal. Additionally, I built a mobile keyboard for iOS that uses my language models as a library, and proposed a novel extension to an existing language model that improves its performance on error-prone text.

My implementation of existing language models saw a level of performance that was comparable with measures made by existing authors. For example, my 5-gram modified Kneser-Ney smoothing model, vanilla RNN model and LSTM model all matched the results from Mikolov on the Penn Tree Bank dataset [26] [27]. This allowed me to make comprehensive and reliable comparisons between each of my models, and assess the trade-offs between accuracy and resource consumption. My work also saw such models complement one another, in that the interpolation of their probability distributions surpassed the performance of all of my stand-alone models.

I explicitly demonstrated that language model performance is deterred by error-prone text. Whilst I made some modifications to an existing model that saw an increase in performance on such text, there is still room for improvement. If I were given more time to work on the project, I would investigate ways to cater for non-spelling corrections. From the CoNLL shared tasks of 2013 [4] and 2014 [5], it is clear that there is a plethora of methods that already exist for grammatical error-correction. It would be interesting to see which of these could be applied to improve the performance of language models on error-prone text.

Throughout this dissertation, I have learnt a great deal about language modelling, but what have my language models learnt from me? Here is a one line summary of my work, in the words of an LSTM-based language model, trained on the significantly small training set that is the text of this LaTeX file:

'the number of words that the language model is a mobile of the network, and the input is the input set.'

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] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

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

[19] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA*, 2016.

[20] S. Hochreiter, *Untersuchungen zu dynamischen neuronalen Netzen*. PhD thesis, diploma thesis, institut für informatik, lehrstuhl prof. brauer, technische universität münchen, 1991.

[21] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE transactions on neural networks*, vol. 5, no. 2, pp. 157–166, 1994.

[22] F. A. Gers and J. Schmidhuber, "Recurrent nets that time and count," in *Neural Networks, 2000. IJCNN 2000, Proceedings of the IEEE-INNS-ENNS International Joint Conference on*, vol. 3, pp. 189–194, IEEE, 2000.

[23] N. Srivastava, *Improving neural networks with dropout*. PhD thesis, University of Toronto, 2013.

[24] W. Zaremba, I. Sutskever, and O. Vinyals, "Recurrent neural network regularization," *arXiv preprint arXiv:1409.2329*, 2014.

[25] 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.

[26] T. Mikolov, "Statistical language models based on neural networks," *Presentation at Google, Mountain View, 2nd April*, 2012.

[27] T. Mikolov, A. Joulin, S. Chopra, M. Mathieu, and M. Ranzato, "Learning longer memory in recurrent neural networks," *arXiv preprint arXiv:1412.7753*, 2014.

Appendix A

Backpropagation Recurrence Relation

To see how the recurrence relation for δ_i^l is derived, consider the neuron i in the first hidden layer of the neural network shown in figure A.1. Without loss of generality, suppose this layer is layer l.

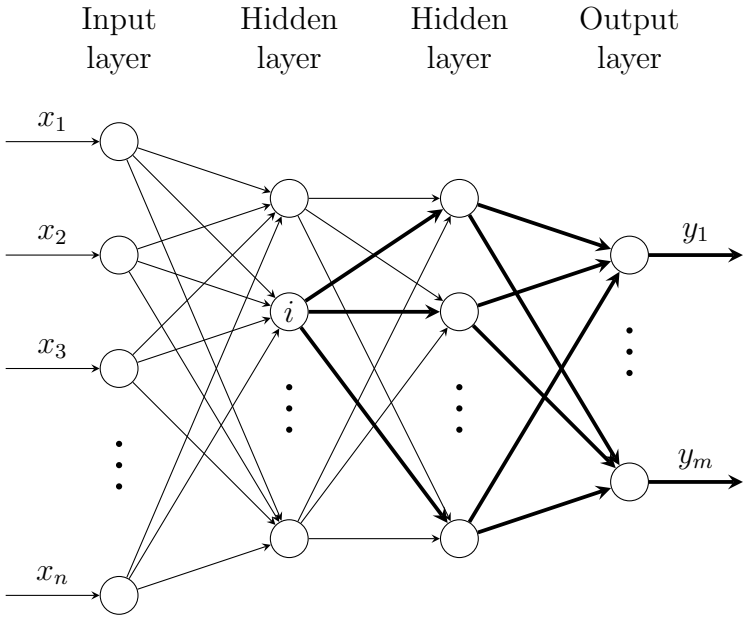


Figure A.1: An illustration of the connections along which the partial derivatives are chained for backpropagating to a particular node i.

From the diagram, it is clear that the derivative $\frac{\partial \mathcal{L}}{\partial z_i^l}$ depends on the derivatives of all of the neurons in layer $l+1$, and so from the chain rule it follows that:

$$\frac{\partial \mathcal{L}}{\partial z_i^l} = \sum_j \frac{\partial \mathcal{L}}{\partial z_j^{l+1}} \frac{\partial z_j^{l+1}}{\partial z_i^l}$$

Plugging in the definition of δ_i^l, and simplifying gives:

$$\delta_i^l = \sum_j \delta_j^{l+1} \frac{\partial z_j^{l+1}}{\partial z_i^l} = \sum_j \delta_j^{l+1} \frac{\partial z_j^{l+1}}{\partial y_i^l} \frac{\partial y_i^l}{\partial z_i^l} = \phi'(z_i^l) \sum_j \delta_j^{l+1} w_{i,j}^l$$

as required.

These ideas can be extended to a recurrent neural network, whereby the derivative $\frac{\partial \mathcal{L}}{\partial z_i^l}$ depends not only on the derivatives of the neurons in layer $l+1$, but also on the neurons in layer l at time step $t+1$:

$$\frac{\partial \mathcal{L}}{\partial z_i^{l,t}} = \sum_j \frac{\partial \mathcal{L}}{\partial z_j^{l+1,t}} \frac{\partial z_j^{l+1,t}}{\partial z_i^{l,t}} + \sum_j \frac{\partial \mathcal{L}}{\partial z_j^{l,t+1}} \frac{\partial z_j^{l,t+1}}{\partial z_i^{l,t}}$$

Again, plugging in the definition of $\delta_i^{l,t}$ and simplifying gives:

$$\delta_i^{l,t} = \sum_j \delta_j^{l+1,t} \frac{\partial z_j^{l+1,t}}{\partial z_i^{l,t}} + \sum_j \delta_j^{l,t+1} \frac{\partial z_j^{l,t+1}}{\partial z_i^{l,t}} = \phi'(z_i^{l,t}) \big(\sum_j \delta_j^{l+1,t} w_{i,j}^l + \sum_j \delta_j^{l,t+1} v_{i,j}^l \big)$$

as required.

Appendix B

Perplexity

The formula for perplexity, presented in equation 4.1 and repeated below for convenience, can be better understood from a touch of information theory.

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

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:

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

Language models assign probability distributions over sequences of words, and so it seems reasonable to use perplexity as a measure of how accurately they model the real underlying distribution. Unfortunately, in language modelling, the underlying distribution, p, is not known, 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.

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

Appendix C

Project Proposal