ASSIGNMENT COVER SHEET

THIS COVER SHEET SHOULD BE ATTACHED TO THE FRONT OF YOUR ASSIGNMENT WHEN IT IS SUBMITTED

STUDENT NAME: Kiarie Ndegwa

STUDENT ID NUMBER: u4742829

COURSE NAME: Individual Research project.

COURSE CODE: COMP8715

DUE DATE: 5pm 28th October 2016

Submission of this assignment constitutes a declaration that:

• No part of this work has been copied from any other person’s work except where due acknowledgement is made in the text; and

• No part of this work has been written by any other person except where such collaboration has been authorised by the course lecturer concerned, and

• No part of this work has been submitted for assessment in another course in this or another part of the University except where authorised by the lecturer(s) concerned.

Student Name: Kiarie Ndegwa
Date: 28th October 2016

Author: Kiarie Ndegwa\textsuperscript{1} and Supervisor: Dr. Lizhen Qu\textsuperscript{2}

\textsuperscript{1}Research School of Computer Science, A.N.U.
\textsuperscript{2}DATA61/CSIRO

*To whom correspondence should be emailed to: kiarie.ndegwa@csiro.au

October 28, 2016
Abstract

Key words: Recurrent Neural Networks, Attention Mechanisms, Gated recurrent networks, Long short term memory units, NUCLE Corpus, collocation errors, grammar error detection

This project aimed to evaluate a novel new neural architecture on the task of grammar error correction with particular focus on collocation errors. The project resulted in the implementation and optimization of a bi-directional LSTM with global soft attention. To the best of this project team’s knowledge, no other team in the world has of yet tried to use such an architecture for the task of grammar error correction. It was intended that this method be tested against two other similar implementations. The first of which is a state of the art network comprised of a bidirectional LSTM network, whose results were only published a couple of weeks before the formal commencement of this project and the second of which won the 2014 CONLL shared NLP task. Model perplexities were used as the metric with which different architectures were compared and optimized during the exploratory experimentation phase of this project. The fastest and lowest scoring perplexity was achieved by a bi-direction network with global soft attention, containing 2, 1000 dimensional hidden layers on the encoder and decoder side and optimized with SGD. This algorithm when trained using collocation only sentences from the NUCLE2014 corpus and achieved a perplexity of 5.5, which was lower than other benchmark networks, the best of which achieved a perplexity of 7.5 on the same data set. Despite this initial promising results, such a high perplexity yielded a $f_{0.5}$ of 0 at the task of error correction when using the official M2 $f_{0.5}$ CONLL scorer. Due to time constraints and technical difficulties; pre-training using the Brown Corpus, an imperative part of this experiment was not accomplished. It is theorized that this would have lead to a perplexity boost that would have precipitated into measurable $f_{0.5}$ scores, as pre-training only with the Brown Corpus models accomplished perplexities in the range of 1.013. This compounded with our limited data set could have resulted in a performance boost.
List of Figures

1.1 Figure 0: Thesis mind map. ................................................. 3

2.1 Figure 1: Biological neurons vs Artificial neurons. .................. 5

2.2 Figure 2: Commonly used activation functions. ...................... 6

2.3 Figure 3: Multi-layer perceptron. ...................................... 7

2.4 Figure 4: The ability of MLPs to continuously warp an input ...... 10

2.5 Figure 6: Equivalent RNNs, after being unfolded ................. 11

2.6 Figure 5: Different RNN configurations, each box represents a matrix, the arrow represent some matrix manipulation function, i.e. summation or the application of some activation function. The red boxes represent the input, whilst the green functions represent the hidden layers of the network and the green the output layers. .......................................................... 12

2.7 Figure 10: A Bi-directional RNN ................................. 13

2.8 Figure 8: Long short term memory. ................................... 14

2.9 Figure 9: Gated recurrent unit. ....................................... 15

2.10 Figure 11: A standard sequence to sequence RNN. ............... 17

2.11 Figure 12: Attention applied to a sequence to sequence network. .... 18

2.12 Figure 10: Global soft attention carried out on an image. .......... 19

2.13 Figure 10: Modeling global soft attention in sentences. .......... 20
2.14 Figure 10: Examples of hard attention networks. A) Picture A, describes, a basic glimpse sensor, $/rho(x_t, L_{t-1})$, $L_{t-1}$, are the coordinates within the images being analyzed. Each of these glimpses in turn generates a sub-images. B) A glimpse sensor can be attached to a multi-layer perceptron, whose hidden layers are represented by $(\theta^l, \theta^g)$ and $g_i$ is the output, C) This in turn can be used to construct a RNN architecture as shown below.

2.15 Figure 11: Hard vs soft attention.

2.16 Figure 10: Global attention

2.17 Figure 11: Local attention

2.18 Figure 17: A Feed forward Neural Language model.

2.19 Figure 18: CBOW and the Skip-gram model.

2.20 Figure 18: A simple recurrent neural network for a recurrent language model.

3.1 Figure 12: The diagram to the left demonstrates the exponentially exploding gradient, whilst the one the right demonstrates the vanishing gradient.

3.2 Figure 13: Gradient clipping exploding gradients.

3.3 Figure 12: Gradient descent in action. The ellipses in the center represent the loss function comprised of 2 interrelated model parameters $w_1$ and $w_2$, represented by the x and y axes. As is the case in a 1 layer MLP with two neurons. The algorithm iteratively steps from the higher loss function values represented by the outer ellipses to the innermost ellipse. The innermost ellipse representing the minima of the function and the final weights of the optimized function.

3.4 Figure 14: Preventing overfitting via the use of early stopping.

3.5 Figure 15: A standard MLP vs one with dropout.

4.1 Figure 16: Relationship between the main NLP components implemented during this project.
List of Tables

5.1 Table 1: Exploratory perplexity results of various architectures carried out on 300 word embeddings. ................................. 52

5.2 Table 2: Perplexity scores VS number of layers in encoder and decoder. 52

5.3 Table 5: $f_{0.5}$ scores of the state of the art vs this projects model without pre-training ................................................................. 53

5.4 Table 7: Examples of generated text by the bi-direction network with attention and lower perplexity score. Red text is the unaltered text segment, green text is corrected sentence according to the network .............. 54
Contents

List of Figures ii

List of Tables iv

1 Introduction. 1

1.1 Overview and motivation. 2

1.2 Project scope 2

1.2.1 Thesis mind map 2

2 Overview of neural network Architectures 4

2.1 Neural networks: Definitions and basics 4

2.2 What are artificial neural networks? 5

2.2.1 The artificial neuron 5

2.2.2 Multi-layer perceptrons. 7

2.3 Sequence modeling using recurrent neural networks 10

2.3.1 Bi-directional networks 13

2.3.2 Gated recurrent networks 15

2.3.3 Sequence to sequence recurrent networks. 16

2.4 Attention Mechanisms in neural networks 18
# Table of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNN</td>
<td>Deep Neural Network.</td>
</tr>
<tr>
<td>CNN</td>
<td>Convolution Neural Network.</td>
</tr>
<tr>
<td>NICTA</td>
<td>National ICT Australia Ltd.</td>
</tr>
<tr>
<td>SVM</td>
<td>Support Vector Machine.</td>
</tr>
<tr>
<td>SVM</td>
<td>Support Vector Regression.</td>
</tr>
<tr>
<td>ANN</td>
<td>Artificial Neural Network.</td>
</tr>
<tr>
<td>MLP</td>
<td>Multi-layer perceptron.</td>
</tr>
<tr>
<td>MCP</td>
<td>McCullochPitts.</td>
</tr>
<tr>
<td>CBOW</td>
<td>Continuous bag of words model</td>
</tr>
<tr>
<td>RNNLM</td>
<td>Recurrent neural network language model.</td>
</tr>
<tr>
<td>NNLM</td>
<td>Neural net language model.</td>
</tr>
<tr>
<td>GEC</td>
<td>Grammar error correction.</td>
</tr>
<tr>
<td>seq2seq</td>
<td>Sequence to sequence.</td>
</tr>
<tr>
<td>RNN</td>
<td>Recurrent neural network.</td>
</tr>
<tr>
<td>ESL</td>
<td>English as a second language.</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics processing unit.</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction.

There are millions of people around the world trying to learn the English language effectively. However, crossing over into the native speaker threshold continues to remain a complex challenge due to nuanced language idiosyncrasies; which often fall out of the demarcation of semantic or syntactic formalism deductible via the use sentence parsers such as Parsey Mcparsface [1]. More often than not, these lapses of “incorrect” English can be attributed to the misuse of collocations.

Collocation errors include multi-word expressions (MWEs), such as ”to kick the bucket” as opposed to ”kick the buckets” or ”telephone booth/box” as opposed to ”telephone cabinet”. Due to the diversity and frequency of such phrases there are currently no effective analytical approaches to ease the problem. It is estimated that there exist as many collocation relationships within the English language alone as there are words in its entire lexicon [24]. This project therefore attempts at ”taking the bull by the horns” by implementing a state of the art recurrent neural network, that will serve as an effective collocation error corrector that can be used for educational purposes amongst ESL students.

The key difference between the algorithm implemented and tested for the purposes of this project is that it represents a single monolithic deep network. That is, a network trained to fix multiword errors. This is an important distinction to note, as most if not all winning teams for previous CONLL NLP competitions[19] constructed a solution that was comprised of an ensemble of state of the art algorithms; comprised of, rule based analytical algorithms, external ranking using language models, alongside the use of sequential deep networks. Implementing a collocation corrector as a monolithic deep network has the added advantage of allowing for shorter training times and a simpler flexible design form.
1.1 Overview and motivation.

This project wished to construct a sequence to sequence deep network that could correct complex collocation errors given input sentences. In the process it was expected that this algorithm simplify current complex grammar error correction architectures - consisting of ensembles of algorithms; by implementing a single neural network that could effectively accomplish the same task.

Currently the record highest correction accuracy is held by the competition winners of the Cambridge university NLP team at the CONLL shared task challenge[19]. This team achieved a $f_{0.5}$ score of 15.65. This project therefore wishes to implement a system that can supersede this score.

The architecture that is implemented in this project is inspired by state of the art neural translation models [12]. The main advantage of the system implemented during this project is that it augments this translation system with a relatively new neural architecture called an attention mechanism. After a comprehensive literature survey it was discovered, that such a system had never been used for the task at hand and this would potentially result in better state of the art collocation correction results.

1.2 Project scope

This project has a twofold goal. Firstly to vigorously mathematically model the different neural network components relevant to this projects final optimized system. Thereby giving the reader a thorough understanding of how different components are directly related to each other. Secondly, to retrofit and augment a neural translation model in order to construct a grammar error corrector that can accurately and effectively correct collocation errors. The finished algorithm once fully trained will result in a correction efficacy comparable to state of the art algorithms. Constructing this algorithm will involve learning to program in the Torch/Lua scripting language, alongside learning how to use Python data mining packages to clean up the various data sets needed for the training process.

The resultant final model will then be evaluated against the state of the art ensemble systems using the $f_{0.5}$ scoring metric. Several model configuration therefore have to be extensively trained and tested before the finding and optimizing the best architecture and comparing its efficacy against the current state of the art.

1.2.1 Thesis mind map

A summary of this thesis’ mind map is summarized in the accompanying diagram below:
Figure 0: Thesis mind map.
Chapter 2

Overview of neural network Architectures

This section aims to exhaustively describe the atomic constituents of neural networks that this thesis builds upon. It’s divided into 4 main sections detailing the various components of this project’s neural network. The accompanying sections to this chapter are summarized below:

- A theoretical description of the most basic computational unit of a neural network—the artificial neuron. Further describing how collections of these units can be configured to form multilayer perceptrons, that can be used to solve linear regression problems and logistic classification,

- The means through which artificial neurons can be used to form recurrent neural networks with the ability to effectively model sequential data

- The means through which attention mechanism function and can boost sequence modeling

- How language models are used to boost language based sequential networks.

2.1 Neural networks: Definitions and basics

This section describes the biological origin that inspired the artificial neuron and its mathematical definition. It is then followed by a description of how they can be configured to work together to form a multi-layer perceptron(MLP) which in turn can be used to model any function as per the universal approximation theorem. The mathematical means through which these MLPs can be used to separate data is also described in detail. Related theories proposed to explain why MLPs are so effective at categorization and data modeling are then briefly touched upon, with a brief description of the manifold hypothesis.
2.2 What are artificial neural networks?

2.2.1 The artificial neuron

The most basic unit of a neural network is the artificial neuron. It is a biological inspired computational unit. In many ways it can easily be thought of as the simulation of a simple single type of a biological neuron. Its inputs and outputs are analogous to the signals transmitted to a neuron’s dendrite pathways and the resultant electrical output emitted through its axon, should some electrical potential within the Soma be surpassed. A basic biological neuron is shown in the diagram below [27]

![Biological neuron diagram](image)

Figure 1: Biological neurons vs Artificial neurons.

The artificial neuron was first proposed and designed by Warren S. McCulloch, a neuroscientist, and Walter Pitts in 1943 [14]. Their mathematical formulation models the basic biological behavior of a neuron as described in figure [2.2].

This neuron takes in a vector of input \( X \); comprised of any type of numerical constituents - for most applications; normalized floating point numbers. This input vector \( X \) is then multiplied piece-wise by another vector \( W \). Where \( W \) is a matrix, each of whose members can be thought of as artificial dendrites.

The resultant summed input is then passed through a activation function\(^1\). This activation function results in a probability distribution that represents the probability of the artificial neuron firing given its current input. Should the activation function’s resultant output surpass a certain learned threshold, then the neuron will 'fire' and produce an output. This behavior can be summarized by the equation below [2.1].

\[
y_k = f \left( \sum_{j=0}^{m} w_{kj} x_j + b \right),
\]

\[ (2.1) \]

Where \( y_k \) is the artificial neuron’s output at index \( k \), and weight matrix \( w \), indexed by integers \( k \) and \( j \) and the input vector \( x \)’s members are indexed by \( j \). The weights in the

\[^1\text{This is also known as an transfer function and in many cases this is used interchangeably.}\]
matrix are therefore indexed as coming from matrix weight members indexed by number $k$ going to input vector members referenced by $j$. $f(\cdot)$ represents the activation function modeling the output along a probability curve. This activation function can take a myriad of different forms each of which squeeze output values from the summation highlighted above into a range with values mostly between -1 and 1.

The most well used of these activation functions are the Heaviside function (Step function), the hyperbolic tangent($\tanh$), the Logistic-Sigmoid($\phi$) and Rectified Linear unit($ReLu$). These functions mathematical definitions are summarized in the table below:

<table>
<thead>
<tr>
<th>Table of activation functions.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tanh function</strong></td>
</tr>
<tr>
<td><strong>Sigmoid</strong></td>
</tr>
<tr>
<td><strong>Step function</strong></td>
</tr>
<tr>
<td><strong>ReLu function</strong></td>
</tr>
</tbody>
</table>

For the sake of the accompanying section it is important to understand the shapes of the transfer functions highlighted above as they will be used in the accompanying section to describe the universal approximation theorem [4] and the innate ability of a multi-layer perceptron to find a dividing hyperplane given a multidimensional categorization problem. It should be noted that these activation functions are all differentiable and can therefore be calculated using backpropagation—the details of which are expanded on in the following sections.

![Figure 2: Commonly used activation functions.](image-url)
2.2.2 Multi-layer perceptrons.

An Multi-layer perceptron (MLP) can be thought of as an acyclic graph of interconnected artificial neurons. These are arranged so that they form consecutive layers, each of which is interconnected. It should be noted that there are no interconnections between the neurons that are constituents of a single layer. When two or more of these layers are stacked together they form a MLP. The layers sandwiched between the input and output of the MLP are known as hidden layers- as can be seen from the image, this is so; as they are not visible from the input and output layers. In the diagram above, layers 1 and 2 are the MLP's hidden layers.

The size of an MLP is dictated by the number of neurons within it. The interconnections between these layers are known as weights. As was described in the image, this weights represent the input signals from the axons of neurons from the previous layer to the current layer and are analogous to dendrites in a biological neuron.

This one directional processing of input information in the form of numerical vectors forms a process known as forward propagation. This forward pass through a MLP is in topological order, moving from the input to the output and can be described be best described by the equation below:

\[ a_j^l = f \left( \sum_{k=0}^{m} w_{jk}^l a_{k}^{l-1} + b_j^l \right), \]  

(2.2)

Where \(a_j\) is the \(j^{th}\) neuron in the \(l^{th}\) layer of the MLP, \(f\), the chosen activation function, \(w_{jk}^l\) the weight between the previous neuron \(j\) and the current neuron \(k\) and \(\sum_{k=0}^{m} w_{jk}^l a_{k}^{l-1}\),

---

2Deep networks are generally comprised of several layers of neurons, there is currently no clear definition of where the dividing line lies between MLPs and deep networks.

---

Figure 3: Multi-layer perceptron.
the summation of the previous activation functions of the previous layer plus the a single bias node $b_j$.

This equation can in turn be vectorized as shown below [2.3]:

$$a^l = f \left( \sum_{k=0}^{m} w^l a^{l-1} + b^l \right), \quad (2.3)$$

Where $a^l, a^{l-1}$ and $w^l$ represent the vectorized output of the previous layers. This function therefore represents the weighted input to the neurons in layer $l$.

It is important to realize the similarity of equation [2.2] to multiple linear regression modeling, such as that shown in [2.4], which also has a succinct vectorized form.

In its simplest form, the multiple linear regression model is trying to fit data to a linear function. For example, given a two dimensional linear space, with an independent $X$ axis and $Y$ dependent axis, through which a 2 dimensional data set is distributed non-stochastically. The regression model tries to fit a curve to this sets members as inclusively as possible. This is intended to minimizes some distance metric between its predicted model and individual points within the sample set. An example of a simple regression model is shown below [2.4].

$$y_i = \beta_1 x_{i1} + \cdots + \beta_p x_{ip} + \epsilon_i = x_i^T \beta + \epsilon_i, \quad i = 1, \ldots, n, \quad (2.4)$$

The model described in [2.4], describes a data set $\{y_i, x_{i1}, \ldots, x_{ip}\}_{i=1}^n$ of $n$ countable units. Whereby a linear regression model finds a curve that fits the dependent outputs $y_i$, using the inputs represented by $x_i$ and coefficients $\beta$. Whereas $\epsilon_i$ represents an unobserved random variable that adds noise to the models linear relationship. Vectorizing this equation in a similar manner as previously demonstrated; results in [2.6]:

$$y = X \beta + \epsilon, \quad (2.5)$$

Where $Y, X$ and are matrices containing the entire data set, whilst the models parameters are contained within the variable $\beta$.

The effectiveness of a regression model to fit a curve to the data set is governed by the cost function that dictates the distance between each point and the model’s curve. This accuracy metric therefore determines efficacy of the model. In many ways a single layered multi-layer perceptron can be seen as a linear regression algorithm that finds a best fitting curve.

However, when dealing with data distributed in dimensional spaces higher than 2 containing more nuanced differences, traditional multi-linear regression models fall dramatically short. When attempting to model non-linear data sets with multi-variate regressors, one would intuitively think that the coefficients in $\beta$ can simply be multiplied together in attempt to model non-linearities within the data set. This assumption is false. The multiplication of all coefficients by some other linear matrix of coefficients, rather than
encode the non-linear complexity of the data set within the model; instead form another a larger matrix albeit one that is still linear. This is best demonstrated below, where the coefficients are encoded in the matrices $W_n$, where $n$ is a monotonic integer value.

$$ y = W_n \ast W_2 \ast W_1 \ast X + \varepsilon \iff y = W \ast X + \varepsilon, \quad (2.6) $$

Conversely, due to the non-linear nature of the activation functions between each consecutive summed layer in a MLP, consecutive layers cannot simply be summarized by a linear matrix representing all constituent layers weights. Here lies the crux of MLPs advantage over other conventional multi-linear regressors. A rule of thumb is that the more complex and non-linear the data set then the more neurons and fully connected layers required to model effectively model it. This forms the basis of the universal approximation theorem—that states that a neural network with more than a single hidden layer can approximately model any continuous multivariate function on a compact subspace of dimensionality $\mathbb{R}^n$[4].

There is however no such thing as a free lunch and this very advantage has the heinous disadvantage of requiring significantly more computational resources to tune all the additional independent neurons and model complex nonlinearities. For this reason the true advantage of multi-layers could not be fully exploited until recent times due to the inhibitive computational cost associated with finding adequately tuning the weights of a multilayer perceptron with several layers. This pitfall therefore served as the algorithm’s Achilles tendon [23] up until the advent of cheaper faster parallel computing hardware in the form GPU cards and the invention of the backpropagation algorithm described in detail below.

There a number of theories that attempt to explain the effectiveness of neural nets at modeling linearly inseparable data such as that shown in the images [2.4a, 2.4b]. One of the most famous being the manifold theorem. This posits that most data is comprised of a low dimensional manifolds separable by a hyperplane in a higher dimension [8]. Thereby it can be seen that the role of a neural net is to perform a form of ambient isotropy - the means through which a manifold is continuously twisted and stretched to see if it can be untangled before finding a dividing line between its constituent manifolds[2].

The image [20] below tries to depict this process by revealing how two data sets are continuously warped and twisted (but never segmented or broken) by the chosen activation function until a plane or hyper-plane can be drawn between the data set (given a categorization problem). An interesting observation to notice is that, the neural net’s input and output can be seen as a bijection between two sets, i.e. the transformations between the output and input can be inverted by a continuous function in both directions [21]. The precise mathematical investigation of how and why artificial neurons and MLPs work is however beyond the scope of this thesis.
(a) A non-linear boundary represented by the red and blue curves.

(b) The two curves warped continuously by the logistic activation function, allowing for a linear separating plane to be drawn between them.

Figure 4: The ability of MLPs to continuously warp an input

As will be described in detail in the section pertaining to how these networks are efficiently trained, the biggest breakthrough in this field was the advent of the backpropagation algorithm [11]. This single-handedly revived a fledgling field and opened up a myriad of applications. So-called deep neural nets begun to fully take advantage of this innate modeling ability and are largely responsible for the novel applications that are becoming ubiquitous in every day life. Their applications range from computer vision tasks such as image classification, to online language translators to speech synthesis and even music generation.

2.3 Sequence modeling using recurrent neural networks

The primary challenge of modeling sequences is to find an effective means of correctly encoding varying lengths of sequences. Particularly when given sequences that contain long term interdependencies within their member tokens, such as is the case in a language’s sentences. Recurrent neural networks are algorithms meant to effectively model such sequences. These rather than model input as fixed sized vectors as is the case with
MLPs- can model sequences of varying lengths. They accomplish this by having their neurons within their equivalent hidden layers have intra-layer interconnections. As shown in the image below.

As can be seen in [2.5], the intra-connected weights between a hidden layers are represented by matrix \( W \), whereas inputs \( X \) to the neuron are mapped by a weight vector \( U \), and a resultant output from the neuron mapped to an output \( O \) by a matrix \( V \).

When rolled out this hidden layer intra-connections are made more visible. This RNN loop can be therefore be thought of as a deep network, the longer the sequence it can take as input; then the deeper the network and the more computational intensive the process of training [6]. A RNN at its most fundamental level can be thought of as a loop, whereby previous outputs rather than being simply discarded; are instead feed back into the next hidden layer of the RNN. Therefore unlike the MLP the RNN takes in two inputs at any given time.

The different configurations of a recurrent network are summarized in the schematic below:
As can be seen in [2.6], these different configurations find use in a multitude of different applications. This include end to end translation neural nets, given a delayed many to many relationship, to named entity recognition in many to one relationships, and sentence POS tagging forming a one to one relationship.

The hidden state of a RNN or cell state, $S$, can be modeled mathematically as shown below:

$$S = f(U.x_t + W.S_{t-1}), \quad (2.7)$$

Where the function $f$ represents a activation function as previously described. And the current input $x$ and the previous cell state $S_{t-1}$. The output of the RNN $o_t$ can be modeled by the following equation:

$$o = f(V.S_t), \quad (2.8)$$

Where the activation function $f$ is applied piece-wise to the mapping matrix $v$.

Another important more subtle difference between this architecture and that of MLPs’ is that the mapping matrices, $U, V, W$ are all shared across all steps, i.e. they are performed across all steps of a single input. Whereas MLPs have different parameters for every different layer. These results in a training regime that is more complex and challenging to achieve as this introduces instabilities with the optimization algorithms used to train such networks. These numerical instabilities are the vanishing and exploding gradient problems described in further detail in section [3]. The primary problem of how information is stored within the fixed size cell state of a RNN therefore serves as its biggest weakness as imperative inra-sequence information gets lost by the network when analyzing long sequences.

Luckily there are means through which these memory loss problems are solved. This solutions take the form of various architectures detailed further in this chapter’s accompanying sections.
2.3.1 Bi-directional networks

As mentioned in the section above the key problem of modeling long sequences, is finding a means to find and store important relationships between components that are distantly temporally separated.

Bi-directional RNNs solve this issue by making sure that all the distantly separated interdependencies are exposed to the network concurrently. They can be visualized as being comprised of two stacked RNNs as described in the section above. The first RNN takes in the sequence moving forward in time, whilst the second receives the same input but reversed in time [9]. The hidden states of these two architectures cells states are then combined via concatenation and mapped into an output using a mapping matrix. Diagram [2.7] demonstrates the inner working of such a bidirectional architecture.

![Diagram 2.7: A Bi-directional RNN](image)

Figure 10: A Bi-directional RNN

The concatenated hidden states and weights of this RNN can be modeled according to the equations below:

\[
\begin{align*}
\vec{S} &= f(\vec{U}.x_t + \vec{V}.\vec{S}_{t-1} + \vec{b}) \\
\bar{S} &= f(\bar{U}.x_t + \bar{V}.\bar{S}_{t-1} + \bar{b})
\end{align*}
\]

\[O_t = f.U.[\vec{S}; \bar{S}] \]  

(2.9)  

(2.10)  

(2.11)

As can be seen in the above equations, the new cell state representing both RNNs is summarized by the new memory cell $[\vec{S}; \bar{S}]$, where the left and right arrows represent the direction of the sequence. Where, $f$ is a activation function, (usually the ReLu). Whereas the output vector is represented $O_t$ at time step, $t$ given the mapping matrices $W$ and $V$. 
Though effective at modeling interdependencies key disadvantage of this architecture is that the number of parameters required for modeling a given sequence double. Secondly interdependencies in long sequences still aren’t quite fully effectively modeled as they could be.

### 2.3.1.1 Long term short term memory

Currently, the most effective means of capturing long term inter-sequence dependencies, is by using gated recurrent recurrent units. One of the most well tested gated RNN is based on the long short term memory cell. These have the advantage of modeling inter-dependencies without requiring more than 2 or more stacked RNNs. Thereby requiring less computation resources than bi-directional RNNs. This is accomplished by efficiently encoding relevant information within their hidden layer.

These computational cells are comprised of 4 main components, namely: an input gate, an output gate, a memory cell and a forget cell. Each of this have their own sets of weights that are optimized during training. This computational units form the basis of the model implemented throughout this thesis. A schematic of a LSTM cell is demonstrated in the image below. [2.8].

![Figure 8: Long short term memory.](image)

The behavior of the above cell [2.8] can be modeled by the following equations.

\[
\begin{align*}
i &= \sigma(U^i.x_t + S_{t-1}.V^i + b^i) \\
f &= \sigma(U^f.x_f + S_{t-1}.V^f + b^f) \\
o &= \sigma(U^o.x_t + S_{t-1}.V^o + b^o) \\
g &= \tanh(U^g.x_t + S_{t-1}.V^g + b^g) \\
c_t &= c_{t-1} \circ f + g \circ i \\
s_t &= \tanh(c_t) \circ o
\end{align*}
\]

(2.12)

Where the respective input, forget and output gates are represented by, \(i, f, o\). Each of these gates in turn have their own set of weights that map the input to the hidden cell.
state, represented by the matrices $W$ and output matrices $V$ representing the mapping to the output states.

The gated candidate’s hidden state is a mapping from the current input $x_t$ and previous cell state $S_{t-1}$. This is similar to the basic RNN described in the previous section. However, rather than naively calculating its current state $c_t$ directly from the previous hidden state and current input; its first passes its previous hidden state through the output $o$ and the forget gate $f$ and, and its current input through the input gate $i$. This allows the network to chose relevant information from the previous state with relevant information from the current input. These chosen parameters are then used to calculate the cells current hidden state.

It is important to note that the $\sigma$ activation functions are chosen so as to represent the probability of gaining or losing information from the previous cell state or the current input. For this reason the values need to range from 0 to 1. 0 meaning the gate is completely closed and vice versa for a value of 1.

2.3.2 Gated recurrent networks

Another similar but newer gated computational unit that has been found to achieve results roughly similar to the LSTM highlighted above whilst employing fewer components is the gated recurrent unit (G.R.U.) [6]. This architecture is comprised of two gates, that are more or less similar in overall functionality to those in a LSTM cell.

A GRU artificial neuron is depicted below [2.12]. As can be seen it is comprised of a reset gate, $r$ and an update gate $z$.

![Figure 9: Gated recurrent unit.](image)

For the sake of consistency, the hidden and candidate hidden states are abbreviated as $h$ and $\tilde{h}$ are replaced with the abbreviations $s$ and $\tilde{s}$. The mathematical model of the
The above computational unit can therefore be described by the equations below:

\[
\begin{align*}
    z &= \sigma(U^z.x_t + S_{t-1}.V^z + b^z) \\
    r &= \sigma(U^r.x_t + S_{t-1}.V^r + b^r) \\
    \tilde{s} &= \text{tanh}(X^t.x_h + (S_{t-1} \circ r).V^h + b^h) \\
    s_t &= (1 - z) \circ h + z \circ s_{t-1}
\end{align*}
\]  

(2.13)

The above equations show that the input and forget gates of an LSTM have been combined to form a single update gate, \( z \). Whereas the reset gate \( r \) directly manipulate previous hidden states. Unlike the LSTM there are no internal cell states within a GRU. Additionally the current state of a GRU network is not passed through an output gate to the next states as is the case within a LSTM.

It should be noted that the performance of a GRU and LSTM has been in many ways found to perform equally well when modeling sequences. For this reason the model implemented throughout this thesis is based on a LSTM node, though further investigation into GRUs efficacy could be further explored as they are a fairly new architecture.

### 2.3.3 Sequence to sequence recurrent networks.

Sequence to sequence network are comprised of layered RNNs. They are currently the most complex class of RNNs as they model interdependencies of two different sequences concurrently through shared hidden layers. They are comprised of two components, namely a hidden layer that takes in the input sequence and an output layer that produces the output sequence. The sequence to sequence layer described below rather than serve as one to one RNN serves as a delayed many to many RNN. That is, the input sequence must entirely be consumed by the network before the network produces an output. They have therefore recently extensively found use in neural translation tasks.

The example used during this section is comprised of a single layer neural sequence to sequence model. The schematic of which is shown below [5]:
As can be seen from the schematic, the job of the encoder is to translate a variable length sequence into a fixed size vector, $h_i$, whose calculation specifics have been expanded on in the sections above. The sequence input in the above depiction takes in one hot encoded vectors, whereas the model implemented in this project uses CBOW word vectors. The resultant hidden layer after the entire input on the encoder side is consumed is the $h_T$ vector.

This fixed size vector is then passed through an activation layer forming $z_i$ on the decoder side. This activation layer on the encoder side forms the hidden layer of the decoder. Where:

\[
z_i = f(h_T, u_{t-1}, z_{t-1})
\]

(2.14)

The previous predicted word on the decoder side is $u_{t-1}$; $z_{t-1}$ is the previous hidden state on the decoder side and $h_T$ the final hidden state from the encoder side. Predicted words, $U$ from the decoder are calculated through a series of steps. The first step is to calculate a scoring for each word in the decoder side’s lexicon

\[
e(k) = w_k^T z_i + b_k,
\]

(2.15)

Where $w_k$ and $b_k$ are the target word vector and a network bias respectively. The calculated scores for each word are then converted into probabilities by passing them through a softmax normalizer as detailed below:

\[
p(w_i = k|w_1, w_2, ..., w_{i-1}, h_T) = \frac{exp(e(k))}{\sum_j exp(j)}
\]

(2.16)

The highest probability words are then passed onto the next state of the decoders hidden state, $z_t$. The above process is repeated until an end of sentence token is reached.
Recently, attention mechanism have began to get applied between the encoders and decoders hidden layers resulting in a new state vector \( \tilde{h}_t \). As shown in the diagram below [12]:

![Figure 12: Attention applied to a sequence to sequence network.](image)

Attention mechanisms are detailed further in [2.4]. This project sees such an attention mechanism tested and optimized, alongside a multi-attention mechanism.

### 2.4 Attention Mechanisms in neural networks

Another recent bio-inspired algorithm that has gained significant traction over the last couple of years is the attention mechanism. This attempts to imitate how attention is focused on certain important areas of with a data entry. An easy to grasp example of what attention tries to mimic, is shown in the image in figure 10 [2.12]:

---

\(^3\)Multi-attention is an attention mechanism between each hidden layer in a stacked sequence to sequence recurrent network
As can be seen from the above image, one’s eye doesn’t trivially focus on each and every pixel of a given photograph, rather it finds what it thinks is the most salient patch of pixels within an image representing an object. It therefore pays less attention to periphery items it deems unimportant. In this case the most important feature, a wind surfer, is at the focal point of this algorithm, followed by the ocean/beach.

This model had successful initial results within the field of computer vision, where sliding windows were conventionally used in image identification. These sliding windows form computational bottlenecks, as the number of convolutions that need to be carried out scale linearly with the number of image pixels [12].

Given its recent success in computer vision, this same attention models have only recently been taken up by the NLP community. An example of this is visually demonstrated in the diagrams below [2.17] [12]. Where it can be seen the attention focuses on particular parts of each sentence in the block of text. It is therefore not difficult to observe the potential benefits of training such a model alongside, gated recurrent units and bi-directional networks to vigorously effectively model sentence interdependencies.

Attention comes in 2 main forms, namely hard and soft attention. Each of these models can in turn can be tuned to have local or global attention. These different types of tuning describe the probability boundary of the distribution of attention [10], i.e. given a sentence whether to focus on a subset of the sentence, or the entire sentence at hand. The mathematical and algorithmic details of how these models are implemented is expanded on in further detail in the sections that follow below.

2.4.1 Stochastic hard attention

Hard attention refers to the fact that the focal area, usually a section of the picture or sequence being analyzed, has sharp boundaries. I.e. the algorithm cannot consider
information outside this hard boundary.

As this subsection of the data is not continuous and therefore non-differentiable, the standard backpropagation algorithm routinely used to train neural networks cannot be employed. Instead in many cases a reinforcement learner needs to be trained to find a suitable policy⁴.

Hard attention reinforcement agents make use an algorithm known as a glimpse sensor, so-called on account of how they take inspiration on how the human eye glances over visual data. These sensors, are for most applications daisy chained together to form a glimpse network [18] as demonstrated in [2.14].

---

⁴This is the equivalent of a loss/cost function when dealing with reinforcement learning

---

Figure 10: Examples of hard attention networks. A) Picture A, describes, a basic glimpse sensor, /\(\rho(x_t, L_{t-1})\), /\(l_{t-1}\), are the coordinates within the images being analyzed. Each of these glimpses in turn generates a sub-images. B) A glimpse sensor can be attached to a multi-layer perceptron, whose hidden layers are represented by (\(\theta^l_\theta^g\)) and \(g_t\) is the output. C) This in turn can be used to construct a RNN architecture as shown below.
The glimpse network shown above generates a feature vector \( g_t = f_g(x_t, l_{t-1}; \theta_g) \), where \( \theta_g = [\theta_0^g, \theta_1^g, \theta_2^g] \), where \( \theta_n^g \) for \( n \in \mathbb{Z} \) represent the hidden states vectors.

This reinforcement agent also contains an internal memory state, that encodes input information from the glimpse network. The glimpse network, encodes information from the data set into a hidden state, \( h_t \). Where \( h_t = f_h(h_{t-1}, g_t; \theta_a), f_h \)

The agents memory state is then used to decide how to deploy its sensor to location \( l_t \) within its retina like search space \( \rho(x_t, l_{t-1}) \) through which it gains environmental information though a limited bandwidth [18]. This glimpse network input from the agents current environment influences the agents chosen action \( a_t \). The overall purpose of a reinforcement learner is to chosen actions that maximize a reward signal, determined by the agents new observation from its previous state, \( x_{t+1} \) and its corresponding reward signal \( r_{t+1} \), which over time adds up to \( R = \sum_{t=1}^{T} r_t \).

This set up forms what is known as a partially observable Markov decision process (POMDP), whereby the true state of the environment is unobserved. The agent learns the policy \( \pi((l_t, a_t)|s_{1:t}; \theta) \). A policy partially based on the agent stochastically exploring its search space. For this reason, this model is called stochastic hard attention. How this reinforcement agent is trained is however beyond the scope of this thesis.

### 2.4.2 Deterministic soft attention

As mentioned in the prior section, modeling a stochastic system requires continuous sampling of its agents environment. Deterministic attention rather than do this, computes a soft attention weighted vector across the entire image/sequence. Forming a model that is both smooth and differentiable across the entire data set. This distribution forms a probability distribution that has higher values in what the network deems are salient features within the data.

Like the hard attention, the model has coordinates that determine the locus of its attention, abbreviated as \( l \). Learning the deterministic soft attention can therefore be seen as maximizing a marginal likelihood function as shown in 2.17.

\[
\theta(\{a_i\}, \{\alpha_i\}) = \sum_{i=1}^{L} \alpha_i a_i, \tag{2.17}
\]

Where \( a_i \) represents a multinoulli distribution\(^5\) and \( \alpha_i \) a weighted continuous context (moving focal point of the attention mechanism) applied to the system whereby \( i \), represents the coordinates of the attention given the input sequence. As equation [2.17] is differentiable standard back-propagation can applied maximize it.

The differences between the described stochastic hard and deterministic soft attention model outcomes can best be described visually[18] as shown in the images below:

\(^5\)A distribution comprised of a distribution of random variable between 0 and 1, similar a Bernoulli distribution but in a higher dimensional space
Figure 11: Hard vs soft attention.

From the diagram above, the left most image represents a seagull that is mid-flight. As can be seen at the top most row of diagrams representing soft attention, the attention is distributed throughout the entire image, despite being concentrated on important features. This hot spot can be seen to move over the image, as different parts of it are taken into consideration and categorized according to the task at hand, i.e. caption generation. Conversely the lower row, representing hard attention, has a uniform distribution but only in a small contained location within the image.

2.4.2.1 Global and Local attention

In a nutshell, global attention when dealing with sequential data, deals with it in all its entirety. Whereas local attention mechanisms only deal with a subsection of a sequence. Local attention is therefore less computationally draining than the global model whilst at the same time being differentiable thereby making it easier to implement and train than a hard attention model that aims to accomplish the same task.

Before delving into the intricacies of these variants of soft attention, we must first understand their shared functionality. For the purposes of this thesis, their usages is described in the context of a sequence to sequence RNN.

Therefore, given a sequence over time, $t$, they can be visualized as taking the hidden state of the topmost layer of a sequence to sequence LSTM, abbreviated at $h_t$. The goal of attention is therefore to to derive a context vector $c_t$ and produce an attention hidden layer $\tilde{h}_t$. This context vector $c_t$ has the ability to capture relevant information contained within the source side, which alongside the hidden state $h_t$ can be used to generate a predicted target word $c_t$.

This attention layer is comprised of a simple concatenation between the hidden state, $h_t$ and $c_t$ as demonstrated in the equation below:

$$\tilde{h}_t = \tanh(W_c[c_t; h_t]),$$  \hspace{1cm} (2.18)

This attention vector, $\tilde{h}_t$ is then used to produce a predictive distribution as shown in equation [2.19]

$$p(y_t | y < t, x) = \text{softmax}(W_s\tilde{h}_t),$$  \hspace{1cm} (2.19)
The means through which the context vector is calculated serve as the shedding point of commonality from these two approaches. Each architecture needs to calculate a variable length alignment vector, $a_t$, which serves as point of differentiation.

In a global attention mechanism, this alignment vector is based on the current target state, $\tilde{h}_t$ and the source states $\bar{h}_s$. Global attention works by taking into account all the encoding hidden states of a given encoder, when deriving the context vector, $c_t$, which is calculated as the weighted average according to $a_t$ given all the source states [12]. This is demonstrated in diagram [2.17].

![Global attention diagram](image)

The alignment vector, $a_t$, can be described by the formulation below:

$$a_t = \text{align}(h_t, \bar{h}_s) = \frac{\exp(score(h_t, \bar{h}_s))}{\sum_{s'} \exp(score(h_t, \bar{h}_{s'}))} \tag{2.20}$$

Where the scoring function in [2.20], is content-based, and comprised of 3 different alternatives described in the following equations [2.21] below:

$$score(h_t^T \bar{h}_s) = \begin{cases} 
h_t^T \bar{h}_s, & \text{dot} \\
h_t^T W_a \bar{h}_s, & \text{general} \\
W_a [h_t^T; \bar{h}_s], & \text{concat} 
\end{cases} \tag{2.21}$$

The purpose of which is to determine the centering of the context window $c_t$. The context vector $a_t$ is then aligned as is the case with global attention. In summary the computational path of the algorithm can be represented as, $h_t \rightarrow a_t \rightarrow c_t \rightarrow \tilde{h}_t$. As
can be observed from the above models global attention, has a major computational bottleneck. All hidden states in the encoder side of the sequence analyzer, must considered every time each individual target word is generated. Consequently longer sequences, such as paragraphs cannot be processed effectively.

Local attention bypasses this inefficiency by introducing another alignment vector named, \( p_t \) for the current aligned target word. This alignment factors in position in the algorithm as shown in the schematic [2.17] by is focusing on a small subset of the source positions per target word [12], following a similar paradigm to the hard attention network described in the sections above. A key distinction being that it remains continuous and therefore differentiable.

In other words the vector \( p_t \), restricts the alignment vector \( a_t \) to a fixed window size. This can be defined as \([p_t - D, p_t + D]\), where the window diameter size, \( D \) is determined via experiment/trial and error. This restricts the alignment vector to a size \( \in \mathbb{R}^{2D+1} \).

Local attention itself comes in two flavors, namely:

- **Monotonic alignment**: where the source and target sequences are fully aligned, i.e. \( p_t = p_t \).
- **Predictive alignment**: whereby the model predicts the aligned position of according to the equation,

\[
p_t = \text{Sigmoid}(v_p^T \tanh(W_p h_t))
\]  

(2.22)

Whereby \( W_p \) and \( v_p \) represent model parameters that are learnt to effectively predict positions. The efficacy of the focal point of the algorithm can be magnified by
modeling the position \( p_t \) as a Gaussian distribution as shown in the equation below:

\[
a_t(s) = \text{align}(h_t, \bar{h}_s) \exp\left(-\frac{(s - p_t)^2}{2\sigma^2}\right)
\]

(2.23)

Where \( \sigma = \frac{D}{2} \), where \( s \) is a real number between the window centered at \( p_t \).

### 2.5 Language models for Natural Language Processing

Another important component of the system implemented in this project is how a language’s sequences can be modeled via the use of a language model. A language model (L.M.) computes the probability of a sentence or a sequence of words. Such sentences can be represented by the probability distribution \( P(W) = P(W_1, W_2, W_3, ..., W_n) \), given words, \( W \) in some lexicon of size \( N \). This can further be extended to the related task of finding the upcoming probability of an upcoming word given a sequence, i.e. \( P(w_s|w_1, w_2, w_3, ..., w_n) \) [9].

There are two primary ways of modeling this probability, namely through statistical inference models and through neural models with continuous distributed vectors. The later of which acts as the baseline to this thesis’ training model [26].

#### 2.5.1 Statistical language models

Given that we are trying to model the joint probability of words within a sentence in correct ordering, conditional probabilities can be generalized using the chain rule. Whereby:

\[
P(X_1, X_2, X_3, ..., X_4) = \prod_i P(X_i|X_1X_2...X_{i-1})
\]

(2.24)

An example of the equation [2.24] above applied to a sentence, \( P(\text{"THE SKY IS SO CLEAR"}) = P(\text{THE}) \times P(\text{SKY}) \times P(IS) \times P(SO) \times P(\text{CLEAR}). \) The main issue involved in finding the probabilities of individual words with respect to a word as is the case in the example;

\[P(\text{THAT}|\text{THE SKY IS SO CLEAR THAT}) = \frac{\text{Count(\text{THE SKY IS SO CLEAR THAT})}}{\text{Count(\text{THE SKY IS SO CLEAR})}}\]

are that there are too many sentences to count given the above joint model and we’d need a impossibly large data set to generate a working LM.

Therefore a simplifying assumption is instead used to calculate the distribution. This assumption is known as the Markov assumption. This can be generalized by the equation below:

\[
P(W_1W_2, ..W_n) \approx \prod_i P(W_i|W_{i-k}, ...W_{i-1})
\]

(2.25)
An example of Markov assumption given the above examples, is $P(\text{THAT}—\text{THE SKY IS SO CLEAR}) \approx P(\text{THAT}—\text{CLEAR})$ or $P(\text{THAT}—\text{SO CLEAR})$.

So given that we want to find the condition based on the current word and the previous word, we would get a probability modeled based on $P(w_i|w_1w_2...w_{i-1}) \approx P(W_i|W_{i-1})$. This can further be generalized to take in bi-grams, trigrams, 4-grams and 5-grams\(^6\).

These statistical N-gram models however have the disadvantage of treating words like atomic units. That is there is no notion of similarity between different words as this are simply represented as indices in a large matrix of size, $V$, the size of the lexicon of the language being modeled. Secondly, statistical models require high quantities of data to model effective N-grams. Data set quantities that in many cases simply do not exist as of yet.

### 2.5.2 Neural language models.

Neural language models (NLM’s) have the advantage over statistical models, as they are able to take in the syntactic and semantic meaning of sentences and generating continuous dense vectors that can model words according to their similarity. However, it should be known that there are many other proposed means of accomplishing this task, such methods include Latent Semantic Analysis (LSA) and Latent Dirichlet Allocation (LDA); which become too computationally expensive when modeling languages on extensive data sets \([15, \ 3]\).

In summary NLM’s generate continuous vector representations by employing a shallow feed forward neural net or using by using a recurrent neural network. The details of which are expanded on below.

#### 2.5.2.1 Feed forward network language models.

These NLMs consist of 4 main components, namely, an input layer, a projection layer, a hidden layer and an output layer. The input layer is comprised of one hot encoded vectors of length $V$ i.e. the size of the languages vocabulary selected, similar to the traditional bag of words model.

The network below is taking words $w_{t-n+1}, w_{t-2}, w_{t-1}$. The projection layer consists of a tanh function, whilst hidden layer is comprised of a softmax function taking in the original one hot encoded vectors of the current word and the output from the $tanh$ projection layer \([3]\).

---

\(^6\) A N-gram is simply the number of words whose conditional probability we are trying to model. So if we are finding the conditional probability of a single word this is known as 1-gram or unigram model and so on and so forth
The figure above is a multilayer perceptron where:

$$f(i, w_{t-1}, ..., w_{t-n+1}) = g(i, C(w_{t-1}), ..., C(w_{t-n+1}))$$  \hspace{1cm} (2.26)$$

Where $g$ is the neural network and $C(i)$ the $i-th$ word vector. This network is trained via a technique called hierarchical softmax that optimizes the computational bottleneck as highlighted in the image above.

- **Continuous bag of words model (CBOW):**
  This model is similar to the feed forward model described above. However it does not contain the softmax layer. This results in all words being projected onto the same hidden vector that is then passed through the piecewise tanh activation function. The resultant of which is averaged. This model is called a continuous bag of words model as it is in-discriminant of the order words are fed into it.
  It is trained using a log-linear classifier with 2, 4-gram context windows. The model is optimized to predict the middle word between these 2 4-grams. This model unlike the one hot encoded bag of words is continuous.

- **The Skip-gram model.** The continuous skip gram model unlike the CBOW model which works by predicting the word between two N-grams; works by predicting the surrounding words given a context window of size, $C$.

These two altered architecture both have the added advantage of easing the computational non-linear bottleneck of the standard feed forward neural language model whilst
performing similar in performance metrics such as perplexity and analogical reasoning tasks\cite{15, 17}. Thus allowing for language models to be trained on larger training sets such as the billion word corpus.

The two architectures are depicted in the diagram below:

![Diagram of CBOW and Skip-gram models](image)

**Figure 18: CBOW and the Skip-gram model.**

These word distributions allow for vector addition that is analogically sound. A couple of interesting examples are demonstrated below:

- king - man + woman = queen
- Iraq - Violence = Jordan
- Human - Animal = Ethics
- President - Power = Prime Minister
- Library - Books = Hall

### 2.5.2.2 Recurrent Neural language models

Recurrent neural language models were proposed to overcome the limitations of feed forward neural nets, i.e. the need to specify the size of its context window, whilst theoretically additionally encoding more complex data. Unlike the feed forward architecture,
the RNN does not have a projection layer, it is therefore only comprised of three main components, namely an input, hidden and output layer. The connections between its constituent hidden layers have a time-delay between them that helps form a type of network memory[16].

Figure 18: A simple recurrent neural network for a recurrent language model.

Like the feed forward neural languages covered in the sections above, they are optimized by using a differentiable log linear loss function paired with a numerical optimizing technique, usually the S.G.D. algorithm described further in the section [3].
Chapter 3

Training neural networks

This section aims to highlight the precise means through which MLPs, RNNs and LSTMs are trained through the forward and backward passes. The imperative backpropagation algorithm and its variant backpropagation through time algorithm are therefore vigorously mathematically described.

Numerical optimization techniques required to calculate these aforementioned algorithms, such as gradient descent, stochastic gradient descent and Adagrad are also fully detailed. This section then culminates with the description of hyper parameter tuning used to speed up training and avoid local minima alongside a section that describes regularization and dropout techniques, used to deter overfitting during training.

3.1 Modeling the loss function as a logistic classifier

Given a pattern matching problem, the role of the loss function is to yield a conditional probability \( P(C_k | x, w) \) which can be used to perform some task dealing with categorization or regression. Where \( C_K \) represents one of \( k \) classes and \( x \) an input pattern with parameters \( w \). The most probable class or prediction is then picked according to the chosen classifiers output which can be described by [3.1]:

\[
h(x) = \arg\max_x p(C_k | x, w)
\]

Where \( h(x) \) is a probabilistic classifier.

Given that we are trying to model our input \( x \), on target output \( z \) given our training set \( S \) comprised of a set of random variables, that are independent and identically distributed (i.i.d.), using parameters \( W \). We can model the total probability distribution of the data set, \( S \) over the model parameters \( w \) as shown below.

\[
p(S|w) = \prod_{(x,z) \in S} P(Z|x, w)
\]
We are therefore trying to find a single high dimensional parameter vector which maximizes the probability $P(w|S)$. The Bayes rule can be used to extract this probability function through inversion of the training set $S$ and the parameters of our model $w$ in [3.2] as described below:

$$p(w|S) = \frac{p(S|w)}{P(S)}$$  \hspace{1cm} (3.3)

A good approximation of such a parameter vector $W$ is known as the Maximum a priori approximation (M.A.P.) that maximizes $P(w|S)$ and thereby use this to make approximations. The M.A.P. equation can be derived from [3.3]. Whereby the denominator is set to 1 as the a priori probabilities $P(S)$ and $P(W)$ are independent of each other. Thus resulting in the equation below:

$$w_{MAP} = \arg\max_W p(S|w)p(w).$$  \hspace{1cm} (3.4)

Assuming that we have a uniform probability distribution of the models parameters $w$, over all entries in the training set $S$. The parameters in $p(w)$ can be ignored, we can therefore obtain the maximum likelihood (M.L.) distribution vector over the entire training set as shown below:

$$w_{ML} = \arg\max_W \prod_{(x,w) \in S} p(Z|x)$$  \hspace{1cm} (3.5)

This can in turn be used to minimize a negative log-likelihood of a given differentiable loss function $\mathbb{L}(S)$, which can be defined as the negative logarithm of the probability of an input within the training set $S$. Resulting in the following equation:

$$\mathbb{L}(S) = -\ln \prod_{(x,z) \in S} p(Z|x) = -\sum_{(x,z) \in S} p(z|x)$$  \hspace{1cm} (3.6)

It therefore suffices to derive the loss $\mathbb{L}(x, z)$ and $\frac{\partial \mathbb{L}(x,w)}{\partial w}$ of an example in the training set, $S$ to find the $w_{ML}$ vector, that represents the neural networks weights. Calculating this derivative, forms the crux of the backpropagation algorithm.

Equation [3.6] which represents the cross-entropy of a model parameterized by $W$ and the distribution of training examples in $S$, can therefore be extended to model sequences as is the case, when using RNNs. This is achieved by finding the normalized log probability of a single sequence that goes from $W_1...W_N$, modeled by equation [3.7].

$$H(W, S) = \lim_{n \to \infty} \frac{1}{n} \log_2 p(W_1...W_n)$$  \hspace{1cm} (3.7)

This cross entropy is used to calculate the efficacy of a sequence model such as a language model through the calculation of perplexity defined by the equation below:

$$\text{Perplexity}(W_1...w_N) = 2^{H(W,S)}$$  \hspace{1cm} (3.8)

The lower the perplexity the better the language model. Therefore finding the maximum likelihood vector $w$ is the equivalent of minimizing perplexity of a sequence model. Perplexity serves as the backbone to majority of the testing carried out through this project.
3.2 Backpropagation

3.2.1 Backpropagation applied to MLPs

A training epoch can be defined as being comprised of two computational motions. Namely the forward pass and the backward pass.

The forward pass of a MLP can described by:

\[ a_h = \sum_{i=1}^{I} w_{ih} x_i \]
\[ b_h = f_h(a_h) \]  \hspace{1cm} (3.9)

Where, \( a_k \) is the combination of all the previous layers output nodes indexed by \( i \) to the hidden layers nodes indexed by \( h \), and \( W \) the weight matrix containing all the model parameters \( w \) and \( x \) the input from the data set, and \( I \) the number of hidden layers within the MLP, and \( f \), the choice of the activation function picked as described in [2,2].

As was shown in equation [3.6], the weights of a well modeled neural network can be thought of as the maximum likelihood estimate of some models parameters \( w \). Thus given that this in an optimization problem, we are trying to find parameter values that minimize the difference of the source values, \( x \) and the target values \( z \) according to some differentiable loss function.

It is important to pick a differentiable loss function as this allows it to be calculated iteratively via the gradient descent algorithm.

Gradient descent works by iteratively adjusting the parameters weights, by finding the derivatives of the loss function and adjusting them toward the negative slope. Gradient descent algorithms are described in further detail in the accompanying section detailing hyperparameter tuning.

Thus, given the loss function \( L(S) \), and a training set with source target pairs \( (x, z) \in S \), we are trying to find, the derivative with respect to a multi-class discriminative model \( y \) parameterized by \( w \). Thus the functions model can be mathematically described by [3,10].

\[ L(x, z) = -\sum_{k=1}^{K} z_k \ln y_k \]  \hspace{1cm} (3.10)

Differentiating the above equation with respect to the layer \( a_k \) in order to propagate error back in the model, we can use the chain rule which can be generalized by [3,11]:

\[ \frac{\partial L(x, z)}{\partial a_k} = \frac{\partial L(x, z)}{\partial y_k} \cdot \frac{\partial y_k}{\partial a_k} \]  \hspace{1cm} (3.11)
Thus the gradient of [3.10] can be derived as shown below:

\[
\frac{\partial L(x, z)}{\partial y} = -\frac{z_k}{y_k}
\]  

(3.12)

Given that a multi-layer perceptron, is analogous to a softmax function applied recursively, each recursion representing a successive layer and its accompanying bias; applying the chain rule, results in the following:

\[
\frac{\partial L(x, z)}{\partial a_k} = \sum_{k'=1}^{K} \frac{\partial L(x, z)}{\partial y'_k} \cdot \frac{\partial y'_k}{\partial a_k}
\]  

(3.13)

The error can thus be propagated back to each individual layer within the network, by taking the partial derivative w.r.t. to each layer \(a_k\) as shown in the equation below:

\[
\frac{\partial L(x, z)}{\partial a_k} = y_k - z_k
\]  

(3.14)

Each individual weights’ partial derivative can in turn also be calculated using the chain rule. This results in the following function [3.15].

\[
\frac{\partial L(x, z)}{\partial w_{ij}} = \frac{\partial L(x, z)}{a_j} \cdot \frac{\partial a_j}{\partial w_{ij}}
\]  

(3.15)

### 3.2.2 Backpropagation applied to RNNs

The training regime of a RNN is exactly the same as that of a MLP and is therefore comprised of a forward and backward pass. However as described previously, at each time step, the network takes in two inputs. One input being the input \(x\) the other being \(b\) the hidden layer of the previous output.

Given a RNN, with \(I\) input units, \(H\) hidden units, and \(K\) output units. Whereby the input to the network is \(x_i^t\), indexed by \(i\) at and time step \(t\).

The summed network input is represented by \(a_h^t\) and its activation function can be represented by \(b_j^t\), where \(j\) is a unit within the RNN, at time step \(t\) of an input sequence \(T\).

The interrelation between these variables can be mathematically be described as shown below:

\[
a_h^t = \sum_{i=1}^{I} w_{ih}x_i^t + \sum_{h'=1}^{H} w_{h'h}b_{h'}^{t-1}
\]  

(3.16)

Whereby the activation function like in the MLP, is modeled on some activation function, \(f\) as detailed below:

\[
b_h^t = f_h(a_h^t)
\]  

(3.17)
This network input can further be expanded on to describe a RNN network with multiple layers represented by $K$ units. Whereby the network output and input units can be calculated at the simultaneously as in [3.18]:

$$a_k^t = \sum_{h=1}^{H} w_{hk} b_h^t$$  \hfill (3.18)

With regards to RNNs there are two widely used backpropagation algorithm variants, these are the real time recurrent learning (RTRL) and the backpropagation through time BPTT algorithms. Seeing as the project with this thesis is concerned is BPTT, only the BPTT algorithm is described in further detail below.

Seeing as conceptually we are trying to find parameters that minimize some loss function modeled around a sequence, we need to employ the chain rule iteratively through the network and factor in the time step as we step back through time. Additionally, we also need to factor in the influence of the previous layers activation function $b$.

Applying the chain rule to equation [3.18], therefore results in the following partial derivative:

$$\delta_t^h = f'(a_t^h) \left( \sum_{k=1}^{T} \delta_k^t w_{hk} + \sum_{h'=1}^{H} \delta_{h'}^{t+1} w_{hh'} \right)$$  \hfill (3.19)

Where $\delta_t^h$:

$$\delta_t^h \text{ def } \frac{\partial L}{\partial a_t^j}$$  \hfill (3.20)

After several mathematical derivations the derivatives with respect to the network weights can be derived and result in the final equation as shown below:

$$\frac{\partial L}{\partial w_{ij}} = \sum_{t=1}^{T} \frac{\partial L}{\partial a_t^j} \frac{\partial a_t^j}{\partial w_{ij}} = \sum_{t=1}^{T} \delta_t^j b_t^i$$  \hfill (3.21)

Unlike standard backpropagation through a MLP, BBPT works by propagating the error all the way back to the beginning of an arbitrary length sequence, by performing updates to highly correlated weights all at once using numerical optimization techniques such as stochastic gradient descent (described in further detail below).

These results in highly correlated updates to shared weights for each sequence, therefore making these numerical optimization technique unstable when training such network. This instability manifests itself in calculated gradients exponentially growing in size during training- resulting in exploding gradients; or weight update falling to values close to zero very rapidly- resulting in a process known as vanishing gradients. These gradient update mishaps are shown in the diagram below, along arbitrary time steps, $t$ [28]:

34
There are a couple of means to deter this gradient instability, namely the use of gradient clipping when dealing with exploding gradients and the use of gated units, as described in the section detailing modeling recurrent units [2.3], which are designed to primarily deter the vanishing gradient problem.

Gradient clipping can be demonstrated in the depiction below [3.4], whereby the gradient is normalized. Should this norm be too big then the gradient is clipped. This clipping is defined by the equation (3.22):

$$\Delta w \leftarrow \Delta w \frac{\Delta_{max}}{\max(|\Delta w|, \Delta_{max})}$$  \hspace{1cm} (3.22)

These gradient clipping is demonstrated in the diagram below [28].

As can be seen from the above demonstration, this is a simple yet effective fix.
other hand vanishing gradients are more challenging to fully solve. The algorithmic means through which this carried out is via the use of gated recurrent units, such as the gated recurrent unit, (G.R.U) and long short term memory (LSTM).

As this project primarily deals with LSTMs only the algorithmic details of this approach are covered in the accompanying section below.

### 3.2.3 Backpropagation applied to LSTMs

A LSTM is trained much like a RNN, whereby the forward pass given an input sequence of length $T$ and an input $X$, is fed into the network starting at $t = 1$ and incrementing up until $T$. Whereas in BPTT, is calculated from $T$ and recursively calculating unit derivatives and update unit weights up until $t=1$.

It is however comprised of 3 additional critical components that help it combat the vanishing gradient problem; these are listed below:

- an input gate ($i$)
- an output gate ($o$)
- a forget gate ($f$)
- a cell state

Each of these gates are in turn are comprised of associated matrices of weights that map different components of the network. Inter-cell connections are known as peephole gates whereas the rest are known as cell mappings. These weight matrices are learnt during the training process to optimize information coming in and out of the cell state.

The forward pass through an LSTM’s three gates can thus be described mathematically as shown the accompanying series of equations[3.24]. It should be noted that their order of calculation is imperative. That is during the forward pass it is important to carry out the calculation strictly in the following order:

$$
\text{Input gate} \rightarrow \text{Forget gate} \rightarrow \text{cell state} \rightarrow \text{Output gates} \rightarrow \text{Cell outputs} \ (3.23)
$$

The derivations for these equations are taken from the work by ???. Like the section above describing general RNNs, $a$ represents the accumulated input and $b$ the result of passing $a$ through an activation function abbreviated as $b$ and $s$ the cell state.

The RNN cell has peephole weights going to the input, output and forget gates, abbreviated as $w_{i}^c$, $w_{o}^c$ and $w_{f}^c$. $f$ is the activation function of the gates, $g$ and $h$ represent the cell input and output activation functions respectively. Whereas the cell state at time step $t$ is $s_t^c$. 

36
Whereas, $I$, $K$ and $H$, are the number of inputs, number of outputs and the number of cells within the hidden layer respectively.

\[
\begin{align*}
\text{Input gates} \\
\quad a^t_i &= \sum_{i=1}^{I} w_i x^t_i + \sum_{h=1}^{H} W_{hi} b^{t-1}_h + \sum_{c=1}^{C} W_{Ci} s^{t-1}_c \\
\quad b^t_i &= f(a^t_i) \\
\text{Forget gate} \\
\quad a^t_\phi &= \sum_{i=1}^{I} w_i \phi x^t_i + \sum_{h=1}^{H} W_{hi} \phi b^{t-1}_h + \sum_{c=1}^{C} W_{Ci} \phi s^{t-1}_c \\
\quad b^t_\phi &= f(a^t_\phi) \\
\text{Cell state} \\
\quad a^t_c &= \sum_{i=1}^{I} w_i c x^t_i + \sum_{h=1}^{H} W_{hc} b^{t-1}_h \\
\quad s^t_c &= b^t_\phi s^{t-1}_c + b^t_\phi g(a^t_c) \\
\text{Output gates} \\
\quad a^t_\omega &= \sum_{i=1}^{I} w_i \omega x^t_i + \sum_{h=1}^{H} W_{h\omega} b^{t-1}_h + \sum_{c=1}^{C} W_{C\omega} s^{t}_c \\
\quad b^t_\omega &= f(a^t_\omega) \\
\text{Cell outputs} \\
\quad b^t_\omega &= f(a^t_\omega)
\end{align*}
\]

The computational order of the backward pass can be summarized as:

\[
\text{Cell outputs} \rightarrow \text{Output gate} \rightarrow \text{Forget gate} \rightarrow \text{Input gate}
\]

(3.25)

The backpropagation algorithm can therefore be mathematically defined by the equations below:

\[
\begin{align*}
\epsilon^t_c & \overset{\text{def}}{=} \frac{\partial L}{\partial b^t_c} \\
\epsilon^t_\phi & \overset{\text{def}}{=} \frac{\partial L}{\partial s^t_c}
\end{align*}
\]

(3.26)

37
The next section describes the numerical optimization methods used to calculate the derivatives needed to calculate the above equations given a sequential data set.

### 3.3 Gradient descent, stochastic gradient descent and hyperparameter tuning

This section is divided into three highly related sections each of which need to be taken into consideration when carrying out effective numerical optimization.

#### 3.3.1 Gradient descent.

The gradients generated through backpropagation need to be followed in the direction of their loss function’s negative slope. The simplest means of doing this is via a technique known as steepest gradient descent [26]. An intuitive description of the algorithm can be shown below: Mathematically this finding this minima is described in the equation below:

\[
\Delta w^n = \alpha \frac{\partial L}{\partial w^n}
\]  

Where the $\Delta w^n$ is the $n^{th}$ weight update, and the learning rate $\alpha \in [0, 1]$. This update is continuously applied to the weights within the network until some stopping criteria is met. For the purposes of this thesis, our stopping criteria is comprised of a hard limit set by a hand picked number of epochs.

The biggest pitfall of the gradient descent method, is that it gets stuck in local minima. One means of partially solving this is by applying momentum to the update function as
Figure 12: Gradient descent in action. The ellipses in the center represent the loss function comprised of 2 interrelated model parameters $w_1$ and $w_2$, represented by the x and y axes. As is the case in a 1 layer MLP with two neurons. The algorithm iteratively steps from the higher loss function values represented by the outer ellipses to the innermost ellipse. The innermost ellipse representing the minima of the function and the final weights of the optimized function.

shown below:

$$\Delta w^n = m\Delta w^{n-1} - \alpha \frac{\partial L}{\partial w^n}$$  

(3.29)

Where $m \in [0, 1]$ can be defined as the momentum.

When all the above gradients are calculated with respect to a loss function defined over the entire training set, it is known as batch learning. Should individual updates be performed according to a single example, the process is known as online training.

The calculation of gradients with respect to a data set is quite slow particularly when given large data sets with millions if not billions of training pairs. A rule of thumb is that given a linear operation - such as calculating loss from the forward pass that processes $n$ floating operations on a training data set; then finding its gradients will take $3 \times$ more time.

For this reason standard batch descent using steepest gradient descent is not a feasible option when generating a state of the art deep network.

### 3.3.2 Stochastic gradient descent.

The most widely used descent algorithm used to factor the aforementioned highlighted shortcomings uses an approximate loss function, and is a method called the Stochastic gradient descent algorithm (S.G.D.). This works by taking a random sample of training set forming a subset or batch of examples and calculating the update $\Delta w$ based on this
sample. For this reason it is imperative to randomize the order in which training examples are fed into the network.

Equation [3.30] can therefore be modified to factor this in, forming a batch gradient descent. Whereby a subset of the training examples are picked by a random function $Q$.

$$
\Delta w^n = m\Delta w^{n-1} - \alpha Q\left(\frac{\partial L}{\partial w^n}\right) \quad (3.30)
$$

However, though S.G.D. drastically improves the speed of the gradient descent algorithm, there remains the issue of optimizing its constituent hyper parameters.

### 3.3.3 Hyperparameter tuning

Another important thing to factor in when deciding choosing an optimizer is picking an adequate step size, i.e. the size of the weight updates propagated through the network, abbreviated as $\alpha$ in the equations above.

A wrongly picked $\alpha$ parameter results in optimizer oscillating back and forth around the loss function’s minima or getting stuck in a local minima resulting in an inefficient algorithm. This manifests itself as a loss function oscillating from a higher value to a lower value repeatedly without settling down to a lower value or a loss function that could be further lowered.

To stop this, the S.G.D. optimizer is further altered to adjust this $\alpha$ step size intuitively during training. This results in an algorithm called the Adaptive gradient algorithm (Adagrad). This is the optimizer picked to train the neural network implemented and detailed in this thesis.

This also has the added advantage of being able to work with data sets comprised of sparse vectors, such as is the case in natural language processing applications and sparse vectors in image recognition tasks. The algorithm still has the step size defined by $\alpha$ but this is augmented with a term $G$ instead.

The $G$ term is described by the outer product as shown below:

$$
G = \sum_{t=1}^{T} g_t g_t^T \quad (3.31)
$$

Where $g_t$ can be defined by:

$$
g_t = Q\left(\frac{\partial L}{\partial w^n}\right) \quad (3.32)
$$
And $Q$ is the stochastically picked batch from the training data. The equation describing the final parameter updates $\Delta w$ can then be calculated as shown below:

$$\Delta w^n = m \Delta w^{n-1} - \frac{\alpha}{\sqrt{G}} g_t \quad (3.33)$$

It should be noted that picking a suitable optimization technique depends on your data set. Usually data sets that are comprised of sparse tensors, are better of being trained with Adagrad.

### 3.3.3.1 Regularization and Dropout

Another issue that needs to be addressed during training is picking an adequate set of $w$ parameters that stops the network from learning the wrong data features and not forming a generalizeable data model.

There are two main ways training can go wrong. Namely, through the process of under-fitting and over-fitting. These two processes are measured by splitting training data in two different ways and is known as the method of cross validation [29]. The details of these techniques are listed below:

- The holdout validation method: Whereby data is segmented into three. The first segment is comprised of the training set, the second the validation set and the third the test set. The test set is used to determn

- The K-folds validation method: Whereby only a training set are employed. The data set is split into $k$ segments, each of which is comprised of a training and test set. This has the advantage of using all entries within a given data set as data points get to be in the test set $K$ times, and the training set $k-1$ times.

Underfitting occurs when given a data set, the network generates a large error when classifying objects post training. Networks demonstrating this are known to be biased. Methods used to prevent a model from having a high bias include:

- Making sure that the network has enough hidden units to represent the required mappings.
- Training the said network for a sufficiently long time, so as to enable it to learn the correct mappings

Conversely, overfitting is the process whereby a model too closely models its training data and all its outlier peculiarities. Performing well by all error measures when exposed to training data, but performing drastically poorly when exposed to new data within the test/validation set. This is also known as high variance model. There are a number of solutions used to stop overfitting, however only the following three techniques are described in detail.
• Early termination: This is the process of observing the performance of neural network accuracy against a validation data set. Should the error rate stop receding and instead begin to increase, as the model is being trained then training is ceased -since this is the most observable warning sign of overfitting. This is intuitively described in the graph below [28]:

Figure 14: Preventing overfitting via the use of early stopping

• L2 regularization: This works by adding artificial constraints to the network and consequently implicitly reducing the number of free parameters. The particulars of this process is described in the equation below:

$$L' = L + \beta \frac{1}{2} ||W||^2_2$$  \hspace{1cm} (3.34)

Where $\frac{1}{2} ||W||^2_2$ is the $L_2$ norm which is equivalent to $\frac{1}{2}(w_1^2 + w_2^2 + ... + w_n^2)$, which penalizes large weights within the network, and $\beta$ is another hand picked hyperparameter.

• Dropout regularization: This is one of the most effective techniques used to stop overfitting. It works by stochastically setting some weight parameters to zero between network layers. This randomness is modeled along Bernoulli distribution\(^1\)[29]. Rather then feed this dropped out weights directly into the following activation unit, their expectation value is taken and the accompanying non-zero weights are scaled up.

It can be viewed as a technique in which noise is added to a neural networks constituent hidden units. Thus, stopping the network from relying to heavily on particular weights within the network. This is intuitively demonstrated in the image below:

\(^1\)given a random variable, the probability of getting the value 1 is $p$ and the probability of getting the value 0 is $1 - p.$
The standard MLP in the above image is described by $z^{(l+1)}_i = \sum_{i=1}^{3} y^l + b^{l+1}$ and the network output, $y^{(l+1)}_i = f(z^{(l+1)}_i)$ Where $l$ is the layer number and $i$ the indices of the previous activation units in layer $l$. The augmented MLP containing dropout has an extra term applied the weights $r^l_i$ which represents the dropout factor modeled on a Bernoulli distribution.

However when dealing with RNNs, it is important to note that dropout is only carried out between the input and output mappings and never between consecutive hidden units. The model that this thesis exclusively dealt with used dropout for the purpose of network regularization.
Chapter 4

Modeling Grammar Error Correction.

This brief section describes the final set of the configuration tested and trained throughout this project. It firstly details the collocation problem that this project aims to solve, followed by a section describing the evaluation metrics that are used to compare this project with the current state of the art. The final section of this chapter is comprised of a schematic detailing the important aspects of the final implemented algorithm. The final architecture described in this section is based on the sections described in the previously in this report. This section therefore rather than rehash fully the description of various components employed, summarizes the final overarching architecture of the system implemented during this project.

4.1 Background and problem definition

The reason this project wished to only solve the issue of collocation\(1\) errors is due to the complexity and prevalence of multi-word expressions within the English language. It has been estimated that there are nearly as many multi-word expression in the modern English language as there are words in its lexicon. Traditional rule based approaches therefore fall short of modeling such sentences, as this approach often leads to rule explosion [24] a process whereby there are as many rules as there are examples in the data set. Examples of some common multi-word expressions include the following:

- congressman at large,
- kick the bucket,
- trip the light,
- spill the beans

\(^{1}\) errors caused by word misalignment within multi-word expressions
As can be seen from the above examples, these expressions cannot be treated using formal rules of grammar between its constituent parts. Therefore an expression like "They kick the buckets" has no relation to they "They kicked the bucket."

4.2 Correction as a machine translation model.

As can be seen from the above examples collocation errors are varied and complex. Rule based analytic approaches therefore do not have sway with regards to detection and correction. To fully consider the rich context within which such multi-word expressions appear it is therefore important to model the entire sequence before detection and correction. For this reason it was decided to treat the collocation correction task in a similar manner to a machine translation manner. That is the input language is an incorrect language whereas the output language is a corrected language. The sequence to sequence LSTM described in 2.3.3. is therefore the workhorse of this project. Translation models have the unique advantage of being able to get rid of or augment input sequences and thereby allow for complex and simple corrections to be made onto variable length sequence.

4.3 Bi-directional LSTMs and global soft attention.

To fully capture the contextual information given a sequence, bi-directional LSTMs are used in the encoder side. This is meant to further boost the efficacy of capturing long term interdependencies within the sequence. The inner mechanics of bi-direction networks was described at large in the previous section. But in a nutshell it is comprised of two concatenated RNNs both of which receive the same sequence but in different temporal directions.

Another architecture involve to further the robustness of the above architecture is the global soft attention mechanism. The attention mechanism has the advantage of allowing the hidden layer to have a longer memory by taking into consideration past sequence tokens. It is hoped that such a mechanism boosts the networks ability to fully capture lone term dependencies within long sequences and capture innate patterns within multi-word expressions.

These two architectures represent the best of recurrent networks that allow for complex interrelations between sequences. It is important to note, that the use of an attention mechanism in the task of grammar corrections is to the best of this project’s knowledge the first time that such an architecture has been employed to the task of grammar correction. The global soft attention architecture is implemented as it is the simplest of the attention mechanisms to be implemented given the short time period of this project.
4.4 Implemented final design

To summarize, the final algorithm implemented during this project is comprised of 3 main components. The language model from which word embeddings are extracted as described in chapter 2.5, the language model used is extracted from the Glove data set comprised of a lexicon of 400,000 words. The training sets used to model the problem and the sequence to sequence recurrent network comprised of a bi-directional encoder side with global soft attention. The general algorithm is depicted in the diagram below:

![Diagram showing the relationship between the main NLP components implemented during this project.](image)

**Figure 16**: Relationship between the main NLP components implemented during this project

The system works by the following the underlying discrete steps described below:

- First, it scans the training corpus to create a lexicon of all its words. This lexicon is stored in a .dict file comprised of an index of all the lexicon's word members.

- This index is then associated with the .txt file of the pre-trained Glove language model. That is a new smaller subset of the language model is generated only using constituent words. The order of this new language model is ordered according to the .dict file indices. This language model is then stored in a .hdf5 file.

- Given an input sequence, it first converts this input into a series of .dict word indices, each of which is used to access the correct word embedding vector contained within the subset .hdf5 file.

- This sequence of vectors is then feed into the RNN as an input,
• The output of the RNN, is then re-indexed by the LM into a series of .dict indices which are then used to decode the output back into a readable sequence.

The model is implemented in the Torch Lua artificial intelligence Lua package, preprocessing is accomplished via the use of various Python 2.7 packages.
Chapter 5

Experimental methodology

This section details the experiments carried out on various RNN architectures applied to the problem of grammar error detection, with particular focus on collocation errors. The main architectures tested are listed below:

- Multi-layered sequence to sequence LSTM RNNs.
- Bi-directional sequence to sequence LSTM RNNs.
- Global soft attention applied to the LSTM encoder output.
- Multi-attention applied to a multi-layered LSTM.

The main measures used to analyze the performance of the algorithm are perplexity as described in [3.8] and $f_{0.5}$ scores using the M2 scoring algorithm\footnote{This is the standard CONLL official benchmarking algorithm}[19], used for benchmarking against other teams algorithms within the NLP field. A couple of data sets were employed during experimentation, namely the NUCLE – 2014 and Brown corpora. These were used to train and test the network against the current state of the art.

This chapter is split into 4 main sections as listed below:

- A section detailing the data sets employed throughout this project,
- A section detailing the closely related state of the art work, forming the baseline for this project,
- The final experimental set up including network exploratory results,
- Experimental results, including a comparison between the system implemented throughout this project vs the current state of the art,
5.1 Baselines

The most recent state of the art grammar error corrector was constructed by the team at Cambridge University\[22\]. This architecture has the primary advantage over other grammar error detection architectures in that it is entirely constructed from a single neural network to solve all grammatical error corrections. This team achieved their results by assembling a bi-directional LSTM neural network with 2 hidden layers on the encoder and decoder side and using the ADAM algorithm for dynamic optimization and using batch sizes of 64 sentences. Their resultant algorithm yielded the best $f_{0.5}$ measure to date when evaluated against the NUCLE2014 corpus achieving a $f_{0.5}$ score of 34.4 and 44 respectively against to separate corrected annotations. The results achieved specifically with regards to collocation errors was however omitted from the specifics of their published paper. It is important to note the challenging nature of producing gold standard annotations-essential for an effective neural net; as even the provided gold standard two annotation NUCLE training texts used throughout this project resulted in different $f_{0.5}$ scores of 45.5 and 55.7 respectively.

Another team from Cambridge who took home 2\textsuperscript{nd} place in the CONLL-2014 shared task [7], serves as another case study whose results this project wished to compete against. This team employed all possible algorithms to complete the task at hand. Their ensemble method was comprised of a rule based algorithm based on the universities\textsuperscript{2} prior work with ESL correction, paired with a language model re-ranker based on Microsofts LM and a LSTM network that they built. Their results with regards to collocation errors set the current state of the art results that this project wished to initially emulate and then surpass. Their combined system achieved a collocation correction $f_{0.5}$ accuracy of 15.65 and an overall grammar correction score of 37.33. This project therefore wished to implement a system with a $f_{0.5}$ higher than 15.75.

5.2 Experimental setup and implementation details

The experiments are concurrently carried out throughout this project are run on single nodes within the BRAGG super cluster. Each node is configured to run 2 Kepler Tesla K20 GPUs, and 8 cores with 32GB of RAM.

The neural networks employed throughout this project were implemented in Torch, building atop the model described by \[12\] and partially implemented by the Harvard NLP group. A large part of this project therefore consisted of retrofitting a state of the art Neural translation model to the Grammar error detection problem at hand and preparing data for training.

\textsuperscript{2}Cambridge Universities NLP Computer science department
All pre-processing of the data, was collected and cleaned using Python 2.7. Code snippets are included in the appendix. Extracting lexicon items from the CBOW LM, range from 20 min to 2hrs, for 300 word embeddings to 800 word embeddings.

Overfitting was averted by ceasing the training process as soon as it was detected, i.e. when the perplexity with respect to the validation set began to rise after a certain number of training epochs.

### 5.2.1 Evaluation metric

An important metric used to benchmark this experiment against other teams work is the trained model’s perplexity and $f_{0.5}/f_1$ scores. The $f_1$ score also known as the harmonic mean and it takes into consideration both the recall and precision to find the statistical discriminative accuracy of the tested algorithm. The $f_1$ and $f_{0.5}$ functions are described in greater detail in the equations below:

\[
F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \quad (5.1)
\]

Whereas the $f_{0.5}$ score while similar, weights recall lower than precision. Therefore a higher precision gives a better score than it would given the $f_1$ measure. The $f_{0.5}$ can be described by a generalization of the harmonic mean described below, where $\beta = 0.5$:

\[
F_\beta = \frac{(1 + \beta^2) \cdot \text{precision} \cdot \text{recall}}{\beta^2 \cdot \text{precision} + \text{recall}} \quad (5.2)
\]

### 5.3 Data set description

The data sets used throughout this experiments are comprised of the following data sets:

- **Brown Corpus**: This is comprised of a wide set of text drawn from 15 genres totaling to 1,014,312 words/tokens. This is used for pre-training the neural network for the purpose of grammar correction [13].

- **NUCLE2014**: This is a collection of 1414 essays written on a variety of topics by English Second Language students from the National University of Singapore. This collection is also comprised of the annotated corrections of these essays made by professional English linguists [19]. This project extract error sentences only with collocation results. The resultant data set only has 4190 training sets containing 5019 collocation errors.

---

3 A high recall means that an algorithm returns relevant results, i.e. true positives in a search space containing both true and false negatives

4 Precision measures the quality of the returned results, i.e. true positives out of a search space containing both true and false positives.
Theses corpora are not initially parallel, they are therefore processed from their initial formats into source and target .txt files that can be consumed by the algorithm. It cannot be understated how imperative and time intensive preparing data sets is. Pre-processing data comprised a significant chunk of this projects time line. The rule of thumb in this industry that 99% of time in machine learning tasks are used in data preparation, was duly confirmed.

5.4 Experimental results

The neural networks tested throughout this project are trained with the following configurations:

- SGD: Learning rate set to 1 for exploratory results with decay rate of 0.5\textsuperscript{5},
- Adagrad: This is set to a constant of 0.1 during optimization
- Batch size: This is set to a constant batch size 64 when pretraining, and 20 when testing on the NUCLE Collocation data,
- Dropout Regularization: This is set to a constant of 0.3,
- Loss function: Negative log likeihood of mean squared error,
- All weights are initialized using a zero mean Gaussian curve with a variance of 0.1.

5.4.1 Exploratory results

These set of experiments were carried out in an attempt to find the most suitable architecture configuration to the task at hand. Glove 100 Vector embeddings were used throughout this process, and represented a corpus of 400,000 words\textsuperscript{25}. The initial tests were carried out on the Brown Corpus data set. SGD was used for optimization purposes with its learning rate set to 1.

It was found through experimentation carried out by [12], that reversing the source sentence alongside the original sentence results in better perplexities for some sequential tasks. For this reason it is tested within the LSTM architecture implemented during this project. The source Brown corpus had 50,004 identical source and target pairs.

The holdout method was used for cross validation, with a total of 40,003 sentences used for training, and 9000 used for validation 1000 for testing. The following table contains the initial exploratory experimental results, using a LSTM with 2 layers, and a hidden layer with 500 dimensions. The word embeddings used throughout this phase of the experiment were carried out by 100 dimension CBOW word vectors.

\textsuperscript{5}This is set to decay after the 9th Epoch, the SGD hyperparameter decrease by 0.5 percent after every epoch to makes sure that the optimizer converges.
<table>
<thead>
<tr>
<th>Architecture</th>
<th>Training times</th>
<th>Perplexity in 1st Epoch</th>
<th>Perplexity before overfitting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq2seq LSTM</td>
<td>≈7 hrs</td>
<td>epoch 1: 447</td>
<td>epoch 12: 23.2</td>
</tr>
<tr>
<td>Reverse sequence LSTM</td>
<td>≈6 hrs</td>
<td>epoch 1: 447</td>
<td>epoch 11: 30.95</td>
</tr>
<tr>
<td>Bi-dir with LSTM</td>
<td>≈9 hrs</td>
<td>epoch 1: 640.5</td>
<td>epoch 11: 27.5</td>
</tr>
<tr>
<td>Attention with LSTM</td>
<td>≈6 hrs</td>
<td>epoch 1: 428.44</td>
<td>epoch 15: 10.92</td>
</tr>
<tr>
<td>Attention + bi-dir LSTM</td>
<td>≈9 hrs</td>
<td>epoch 1: 447</td>
<td>epoch 11: 6.87</td>
</tr>
</tbody>
</table>

Table 1: Exploratory perplexity results of various architectures carried out on 300 word embeddings.

The above architecture were comprised of the following number of weights:

- Seq2seq: 13,446,688 weights
- Reverse Sequence: 13,446,688 weights
- Global attention with LSTM: 16,650,688 weights
- Bi-direction with attention LSTM: 13,696,688 weights
- Attention + bi-direction LSTM: 16,908,688 weights

As can be seen from the above table the architecture that yielded the best results, was the combination of a bi-directional network with global soft attention, which trained 16,908,688 weights. The number of weight in these models also explain the extending training times.

The next step in experiment was optimizing the number of hidden layers on the encoder and decoder side of the sequence to sequence architecture.

5.4.1.1 Stacked layers vs perplexity

The following steps were carried out in an attempt to find out the optimal number of layers within the encoder and decoder side that would result in the lowest perplexities. As in the previous section, 100d word embeddings were used, and the hidden layers were comprised of 500d vectors. The experimental results are summarized in the table below:

<table>
<thead>
<tr>
<th>No of layers</th>
<th>Training times</th>
<th>Perplexity in 1st Epoch</th>
<th>Perplexity before overfitting</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>≈9 hrs</td>
<td>epoch 1: 340.5</td>
<td>epoch 11: 14.4</td>
</tr>
<tr>
<td>4</td>
<td>≈12 hrs</td>
<td>epoch 1: 628.44</td>
<td>epoch 11: 10.6</td>
</tr>
<tr>
<td>6</td>
<td>≈14.5 hrs</td>
<td>epoch 1: 847</td>
<td>epoch 11: 8.1</td>
</tr>
<tr>
<td>8</td>
<td>≈17 hrs</td>
<td>epoch 1: 947</td>
<td>epoch 11: 7.9</td>
</tr>
</tbody>
</table>

Table 2: Perplexity scores VS number of layers in encoder and decoder.
As can be seen from the above table, the deeper the network, the longer the training times, but the lower the perplexity scores. These results also demonstrated that given the size of the data set being used during this project, there isn’t a need to use a deeper network as this results in longer epochs and no real tangible improvement.

5.4.2 Final configuration vs state of the art G.E.C.

The final configurations chosen after the initial set of experiments above are listed below:

- Bi-directional LSTM network, with 2 hidden layers, 200d hidden layer, training batch size set to 64, dropout of 0.2, optimization using ADAM\(^{11}\) This served as an emulation of the Cambridge’s team recent state of the art Grammar error correction algorithm. This had a combined number of roughly 17 \(\times\) \(10^6\) weights.

- Bi-directional LSTM network, with 2 hidden layers, 1000d hidden layer, training batch size of 64, dropout of 0.3, using simple/standard SGD optimization. That served as this projects grammar error correction algorithm. This final architecture consisted of a total of roughly 53 \(\times\) \(10^6\) weights.

It was intended that these two models be trained using two different data configurations. The first configuration comprised entirely of the above architectures; trained only against collocation errors in the NUCLE corpus. The results of which are detailed below:

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Final Perplexities</th>
<th>(f_{0.5}) score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bi-directional network</td>
<td>7.4</td>
<td>0.0</td>
</tr>
<tr>
<td>Bi-directional network with Soft global attention</td>
<td>5.5</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Table 5: \(f_{0.5}\) scores of the state of the art vs this projects model without pre-training

The above poor scores were not surprising given that the state of the art algorithms described in the previous sections used extensive augmented data sets, that consisted of larger sets of training examples. Due to the limited time that this project had to implement the above algorithms it was not possible to clean and pre-process all the data sets. Although some of the examples below may seem like corrections, they are not collocation corrections. This examples however show the power of deep networks when it comes to modeling languages, even being able to learn some language rules given such a limited data set.

\(^{11}\) An adaptive learning algorithm not described within this thesis.
Therefore, it would have never *met* the KLM.

personal one, *irregardless* of the genetic testing result of another family member.

Contrarily, the main source of energy which ...

To certain extent, ...

... a wild debate on the banning of the use of surveillance ...

... most of our electricity was generated from coal, fuel, or oil refineries.

... as compared to other *corresponding* energy source.

... more research should be *carried out* on the nuclear power plant.

... no matter what the sequence of innovation-development process ...

**Table 7: Examples of generated text by the bi-direction network with attention and lower perplexity score.** Red text is the unaltered text segment, green text is corrected sentence according to the network.

The second configuration was meant to be pre-trained on the Brown Corpus before being trained on the NUCLE corpus cleaned up specifically to contain only collocation errors.

This pre-training would have been an essential part of the experiment as it would have helped the architecture deduce the probability distribution of the language as much as
possible, particularly given the small size of the NUCLE collocation error corpus.

However, despite the potential accuracy gain that this set-up could have achieved; due to time constraints, some critical technical issues could not be resolved within this project’s short time frame. This part of the experiment was therefore unfortunately incomplete.
Chapter 6

Conclusion and future work

6.1 Conclusion

As can be seen RNNs and all their flavors are by no means trivial to understand or simple to implement. It was the intention of this project to therefore vigorously understand the inner workings of neural networks and how to effectively train them. Other grasped imperative concepts also included learning how to model languages using statistical methods and neural models. With particular focus being paid to the computationally efficient CBOW method employed throughout this projects duration.

The architecture optimized throughout this project resulted in negligible $f_{0.5}$ scores on the NUCLE2014 data set with particular focus on the challenging task of collocation error correction. With a bit of further optimization and a larger training data set focused on collocation errors, this algorithm could be further optimized using pre-training on the Brown Corpus, which could have resulted in higher $f_{0.5}$ scores.

6.2 Future work

As previously mentioned the best architecture found for the task of grammar correction was found to be a bidirectional neural network with global softmax between its encoder and decoder. An architecture to the best of my knowledge no other team in the world has used to solve the grammar error correction task. The results gained throughout this project could therefore have been improved had a larger training corpora been used, such as the FCE data set and most importantly should pre-training have been accomplished. Such data sets could fully make use of 800d word embeddings and further optimize the system.

Potential future alterations of the neural algorithm could seen the algorithm augmented a language model re-ranker on the decoder side of the network to help the beam search pick suitable candidates more robustly during the decoding phase of the algorithm, thereby
speeding up the training times and improving the $f_{0.5}$ scores of the resultant system.
Bibliography


Appendices
1 Study contract
INDEPENDENT STUDY CONTRACT

Note: Enrolment is subject to approval by the projects co-ordinator

SECTION A (Students and Supervisors)

UniID: u4742829

SURNAME: Ndegwa  FIRST NAMES: Kiarie

PROJECT SUPERVISOR (may be external): Lizhen Qu - NICTA/DATA61

COURSE SUPERVISOR (a RSCS academic): ____________________________________________

COURSE CODE, TITLE AND UNIT: Individual Project__COMP8715__12 units__________________

☒ S2  YEAR: _____2016___________

PROJECT TITLE:

Deep check: Deep learning based Grammar Checker

LEARNING OBJECTIVES:

Deep learning, Convolutional Neural Networks, Recurrent Neural Networks, Natural Language processing, Entity resolution.

PROJECT DESCRIPTION:

Millions of people in the world are learning English. They should be able to benefit directly from an automated grammar checker. Thus, this project aims to apply deep learning methods to automatically correct grammar errors, with a particular focus on collocation errors. The key idea is to formulate it as a machine translation problem, which translates an error-prone sentence into a correct one. The implemented models will be evaluated on benchmark datasets, which contain short English texts written by non-native speakers of English.

It is intended that the student partake in the design, implementation and testing of a novel sequence to sequence recurrent network that can be used to accomplish this error correction and benchmark their results against current state of the art networks.

Research School of Computer Science
The learning objectives are summarized as follows:

a) Have an overarching deep understanding of recurrent neural nets and deep learning in the field of Natural language processing.

b) Understand the core algorithms and models of deep learning for grammar error correction.

c) Be able to read current research papers about grammar error correction.

d) Learn and be familiarized with current state of the art Deep learning frameworks such as Torch.

e) Implement a student designed algorithm to solve NLP problems.

f) Conduct extensive experiments to evaluate the implemented program with the state of the art.

ASSESSMENT (as per course’s project rules web page, with the differences noted below):

<table>
<thead>
<tr>
<th>Assessed project components:</th>
<th>% of mark</th>
<th>Due date</th>
<th>Evaluated by:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Report: name style: <em><strong><strong>Research report</strong></strong></em>_____</td>
<td><em>60</em> (60%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(e.g. research report, software description...)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Artefact: name kind: __ Working Algorithm__________</td>
<td><em>30</em> (30%)</td>
<td></td>
<td>Lizhen Qu</td>
</tr>
<tr>
<td>(e.g. software, user interface, robot...)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Presentation:</td>
<td><em>10</em> (10%)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MEETING DATES (IF KNOWN):
Weekly or bi-weekly upon availability.

STUDENT DECLARATION: I agree to fulfil the above defined contract:

……..Kiarie Ndegwa…………………………………..10h July 2016……..
Signature                                        Date

SECTION B (Supervisor):
I am willing to supervise and support this project. I have checked the student’s academic record and believe this student can complete the project.

……..Lizhen Qu…………………………………….10 July 2016……..
<table>
<thead>
<tr>
<th>REQUIRED DEPARTMENT RESOURCES:</th>
</tr>
</thead>
</table>

**SECTION C (Course coordinator approval)**

<table>
<thead>
<tr>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
</table>

**SECTION D (Projects coordinator approval)**

<table>
<thead>
<tr>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
</table>