Jiaxin Wang

A Recursive Recurrent Neural Network Decoder for Grammatical Error Correction

Computer Science Tripos – Part II

Emmanuel College

May 12, 2022

Declaration

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

Signed *Jiaxin Wang*

Date May 12, 2022

Acknowledgements

I would like to thank my supervisors, Dr Zheng Yuan and Dr Christopher Bryant, for their help and guidance throughout this project.

My gratitude extends to my Director of Studies, Dr Thomas Sauerwald, for his valuable comments on my dissertation.

I would like to give my special thanks to Declan Shafi for his unconditional support and understanding.

Proforma

Candidate Number:	2432D
Project Title:	A Recursive Recurrent Neural Network Decoder for Grammatical Error Correction
Examination:	Computer Science Tripos – Part II, 2022
Word Count:	10213 words[1]
Code line count:	1186 [2]
Project Originator:	The Author
Supervisor:	Dr Zheng Yuan, Dr Christopher Bryant

Original Aims of the Project

This project aims to build a statistical machine translation (SMT) system for grammatical error correction using a recursive recurrent neural network (R^2NN) model as the decoder. It involves constructing a translation confidence based phrase pair embedding (TCBPPE) using a feedforward neural network (FNN) and a recurrent neural network (RNN), implementing the R^2NN model and training the R^2NN SMT system. A baseline SMT system (Moses) was built and trained with the same data. The R^2NN SMT was evaluated against baseline SMT on the grammatical error correction task using the $F_{0.5}$ metric.

Work Completed

All the core goals of the project have been achieved. Two SMT systems were built, namely Moses SMT and R^2NN SMT. For Moses SMT, a baseline phrase-based SMT system was constructed. For R^2NN SMT, the recursive recurrent neural network was implemented from scratch and the phrase pair embedding (TCBPPE) was obtained to be used with R^2NN. Both systems were trained and tested with the same data. Finally,

[1]This word count was computed by `\immediate\write18{texcount -1 -sum -merge -q diss.tex output.bbl > diss-words.sum} and \input{diss-words.sum} words`

[2]Code line count was computed by `find ./model ./helper_scripts -name "*.py" -print | xargs wc -l`

a comparison was carried out on the evaluation results of the two SMT systems for the task of grammatical error correction.

Special Difficulties

None.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Grammatical Error Correction (GEC) is the task of producing a grammatically correct sentence given a potentially erroneous text while preserving its meaning. One of the many motivations behind this task is that it plays a significant role in helping learners of a foreign language understand the language better. Being an English as a second language (ESL) learner myself, it is often difficult for me to spot the errors in my English writing since English grammar is quite different from that of my first language. Native English speakers may wish to use a GEC system to avoid mistakes in their writing, especially in a professional environment (e.g. work emails).

One common approach to GEC is to treat GEC as a translation problem and use a statistical machine translation (SMT) system to solve it. We train an SMT system that takes an erroneous source sentence as input and the grammatically correct sentence as the expected output. By searching for the best 'translation' from the source sentence, the aim is to find a grammatically correct sentence that preserves the meaning of the original sentence.

Liu et al. proposed a novel model for SMT using a **recursive recurrent neural network** (R^2NN) [12]. It has a tree structure similar to a recursive neural network, with recurrent input vectors to encode global information that cannot be generated by the child node representations. In the paper, they showed that their R^2NN SMT model achieved better performance than a state-of-the-art baseline system on a Chinese-to-English translation task. However, they did not publish their code, and no implementation of such an R^2NN SMT model is publicly available.

It is therefore of interest to implement such a neural network and apply it to the task of GEC. For one, publishing the code could benefit other researchers in this area who are interested in natural language processing (NLP), GEC, and neural networks. For another, if the R^2NN model is proven to be useful for GEC, it could help with developing better GEC systems and assist millions of English learners.

1.2 Problem Overview

An SMT system has four components: a language model (LM) which computes the probability of a given word sequence being valid, a translation model (TM) which constructs mappings of phrases between source and target corpora and calculates translation probabilities, a reordering model (RM) which handles phrase reordering, and a decoder which looks for the best translation candidate [19]. The R^2NN model proposed by Liu et al. was used in the end-to-end decoding process of SMT, i.e. as a decoder. The goal of this project is to implement such a decoder, integrate it into an SMT system, apply it to GEC and evaluate its performance against a baseline SMT system.

The GEC system should aim to correct all types of errors in non-native English text, including grammatical, lexical and orthographic errors, as defined in the BEA 2019 Shared Task [1]. Lexical errors are mistakes at the word level, such as wrong choice of words that do not fit in their context. Grammatical errors violate the grammar of a language, such as in subject-verb agreement. Orthographic errors include spelling errors but also casing errors (e.g. upper and lower case) and punctuation errors.

Moses [11] was used as the baseline SMT system to be evaluated against the R^2NN SMT. Moses is a toolkit commonly used for statistical machine translation. The toolkit contains a decoder (Moses decoder) and various tools for training and tuning. It also provides a user guide on how to use the decoder together with other models to build a SMT system. For this project, I built and trained a Moses SMT system. I also implemented and trained an R^2NN SMT system which uses the R^2NN model as the decoder. Finally, I compared the performance of both systems on the task of grammatical error correction.

1.3 Related Work

The paper by Liu et al. [12] discussed the structure of R^2NN and how it can be used as a decoder. They also proposed a way to initialise the phrase pair embedding called **translation confidence based phrase pair embedding** (TCBPPE) to generate the leaf nodes. This was used as the vector representations of phrase pairs. After training the system on data from the IWSLT 2009 dialog task [17], they tested the system on a Chinese-to-English translation task. Finally, they evaluated their system against a baseline decoder using the case insensitive IBM BLEU-4 method [16]. Details of R^2NN and TCBPPE are discussed in the following chapters.

SMT systems have been successfully used for the task of grammatical error correction. Yuan and Felice [21] applied Moses SMT with GIZA++ and IRSTLM to grammatical error correction in the CoNLL-2013 Shared Task [14]. Junczys-Dowmunt and Grundkiewicz [8] developed a GEC system based on an SMT system, namely phrase-based Moses [11], and they ranked third place in the CoNLL-2014 Shared Task [13]. They also discussed the improvement that can be achieved by parameter tuning towards the evaluation metric for GEC in phrase-based SMT systems [9]. Yuan et al. [20] discussed how

SMT can be applied to GEC and how a re-ranking model can improve the performance of SMT-based GEC system.

This project aims to implement the R^{2}NN together with TCBPPE in the paper [12] and build a SMT system. A phrase-based Moses SMT system [11] was built to be used as a baseline system. Both systems were applied to GEC and then evaluated by the ERRANT scorer [2, 5], the official evaluation tool of the BEA 2019 Shared Task [1].

Chapter 2

Preparation

2.1 Starting Point

The goal of this project is to implement the R^{2}NN decoder proposed by Liu et al. in their paper *A Recursive Recurrent Neural Network for Statistical Machine Translation* [12] and to compare its performance against a baseline Moses [11] decoder.

The Part IB Computer Science Tripos course Formal Models of Language[1] presented the foundation theory of Natural Language Processing, while Artificial Intelligence[2] gave an introduction to neural networks. Prior to this project, I did not have any coding experience with neural networks or any statistical machine translation systems. For this reason, I had to read academic papers to get familiar with the research in the relevant areas.

2.2 Neural Networks

2.2.1 Feedforward Neural Network

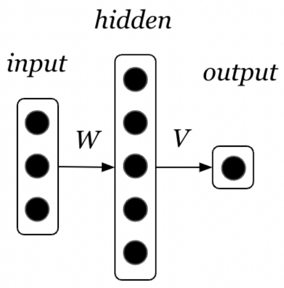


Figure 2.1: Example of a feedforward neural network

[1]https://www.cl.cam.ac.uk/teaching/2021/ForModLang/
[2]https://www.cl.cam.ac.uk/teaching/2021/ArtInt/

A feedforward neural network consists of three parts: an input layer, one or more hidden layers, and an output layer. In a feedforward neural network, data only flows in one direction (forward) from input to output. Figure 2.1 shows an example of a feedforward neural network with one hidden layer. If the input to the network is represented by x, the hidden layer is computed by

$$f(W(x))$$

and the output layer can be computed as

$$g(V(f(W(x))))$$

where W, V are the weights and f, g are the activation functions. It has a simple structure, but it can be powerful in certain tasks.

2.2.2 Recurrent Neural Network (RNN)

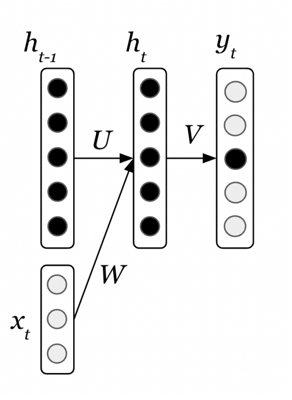


Figure 2.2: Example of a recurrent neural network (RNN)

Recurrent neural networks are usually used to deal with sequences. They allow access to the input data as well as past data to compute the next state. As shown in Figure 2.2, the hidden layer h_t is computed using both input vector x_t at time t and the hidden state h_{t-1} which contains the history information from time 0 to $t-1$. For each timestep t, the hidden state can be expressed as:

$$h_t = Wx_t + Uh_{t-1}$$

And we can compute the probability of the output vector y_t at time t by

$$p(y_t) = \frac{e^{y_t}}{\sum_i e^{y_i}}$$

2.2.3 Recursive Neural Network (RvNN)

A recursive neural network has a tree-like structure. Figure 2.3 illustrates an example of a basic RvNN architecture. The parent node representation is computed from its child nodes' representation as follows:

$$p_{n,n+1} = f(W[x_n; x_{n+1}])$$

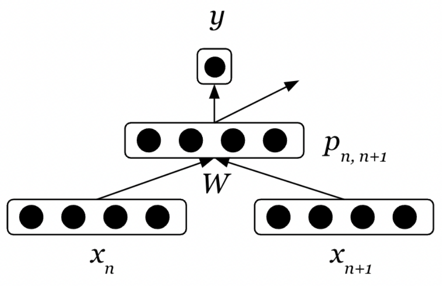


Figure 2.3: Example of a recursive neural network (RvNN)

where f is the activation function. The same weight matrix W is applied recursively over the input.

2.2.4 Recursive Recurrent Neural Network (R^{2}NN)

The R^{2}NN proposed by Liu et al. combines the features of RvNN and RNN. It has a tree-like structure similar to RvNN, with recurrent vectors added to integrate global information. As shown in Figure 2.4, $s^{[l,m]}$ and $s^{[m,n]}$ are the **representation vectors** of child nodes $[l, m]$ and $[m, n]$. The **recurrent input vectors**, $x^{[l,m]}$ and $x^{[m,n]}$ are added to the two child nodes respectively. They encode the global information, such as language model scores and translation model scores. A third recurrent input vector $x^{[l,n]}$ is added to the parent node $[l, n]$. The parent node representation is computed as

$$s_j^{[l,n]} = f(\sum_i \hat{x}_i^{[l,n]} w_{ji})$$

where $\hat{x}$ is the concatenation of vectors $[x^{[l,m]}; s^{[l,m]}; x^{[m,n]}; s^{[m,n]}]$, and f is the $HTanh$ function. The output, $y^{[l,n]}$, is computed as

$$y^{[l,n]} = \sum_j ([s^{[l,n]}; x^{[l,n]}])_j v_j$$

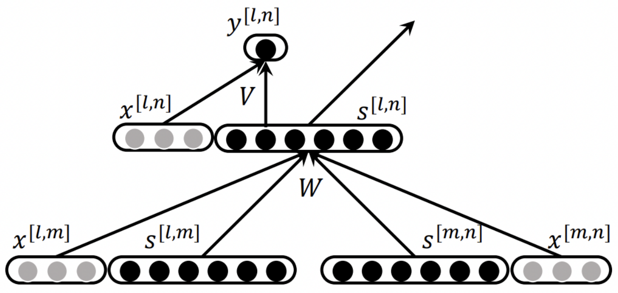


Figure 2.4: Recursive recurrent neural network from the paper [12]

2.3 Statistical Machine Translation

2.3.1 SMT system

The book *Speech and Language Processing* [10] gives a detailed introduction to statistical machine translation systems. A typical statistical machine translation system consists of four main components: a language model, a translation model, a reordering model and a decoder. If we are translating from a source language F to a target language E, we could model the process as a Bayesian noisy channel model and find the best translation $\hat{E}$ which gives the highest probability of $P(E|F)$. This is given by

$$\begin{aligned}\hat{E} &= \arg\max_E P(E|F) \\ &= \arg\max_E \frac{P(F|E)P(E)}{P(F)} \\ &= \arg\max_E P(F|E)P(E)\end{aligned}$$

where $P(F)$ is ignored inside $\arg\max_E$ because $P(F)$ is constant for a given source F.

Language Model

The language model computes $P(E)$, i.e. the probability of a given sequence of words being valid. Consider a sequence of words, $w_1, w_2, \ldots, w_n$, the probability of this sequence can be computed using **the chain rule**:

$$\begin{aligned}P(w_1, w_2, \ldots, w_n) &= P(w_1)P(w_2|w_1)\ldots P(w_n|w_1 \ldots w_{n-1}) \\ &= \prod_{i=1}^{n} P(w_i|w_1 \ldots w_{i-1}) \\ &= \prod_{i=1}^{n} P(w_i|w_1^{i-1})\end{aligned}$$

In SMT systems, this probability is calculated based on word counts in the training dataset. An **N-gram** model would look at $N-1$ words in the past:

$$P(w_n|w_1^{n-1}) \approx P(w_n|w_{n-N+1}^{n-1})$$

Therefore, we can approximate the probability of sequence $w_1, w_2, \ldots, w_n$ by

$$\begin{aligned}P(w_1, w_2, \ldots, w_n) &= \prod_{i=1}^{n} P(w_i|w_1^{i-1}) \\ &\approx \prod_{i=1}^{n} P(w_i|w_{i-N+1}^{i-1})\end{aligned}$$

where

$$P(w_i|w_{i-N+1}^{i-1}) = \frac{count(w_{i-N+1}^{i})}{count(w_{i-N+1}^{i-1})}$$

However, if a sequence did not appear in the training set, the above formula would give a probability of 0 for this sequence, even if the sequence is actually valid. To deal with this problem, a technique called **smoothing** is used to assign a non-zero probability to a sequence that is unseen in the training corpora. The details of smoothing fall outside the scope of this project. For more information, refer to the textbook [10].

Translation Model and Reordering Model

The translation model estimates $P(F|E)$, the probability that E generates F. Firstly, an alignment model is introduced to learn the mappings between words. Then the **translation probability** can be computed using the alignment and construct a phrase translation table. The table should contain all the phrase pairs and the associated probabilities. For example, the table will contain the probability of translating from the source phrase *I likes* to the target phrase *I like*. The reordering model handles phrase reordering. It measures the **distortion probability**, which is the probability of two consecutive phrases in the target language E being separated in source language F by a distance. Eventually, $P(F|E)$ is computed based on translation probabilities and distortion probabilities.

Decoder

The decoder searches for the $\hat{E}$ which maximizes $P(F|E)P(E)$, the product of the two probabilities computed above. This can be done using **beam search**. Starting with the null hypothesis (with no translations) as the initial state, the hypothesis can be expanded by finding possible phrase translations for words in the source sentence F. A cost is computed, and the n lowest cost translations are kept at each stage. The expansion is repeated until all the words in F have been covered. Eventually, the translation that gives the lowest cost would be the output.

2.3.2 Moses SMT

Moses [11] is a statistical machine translation toolkit. A tutorial on how to build a baseline SMT system is available on the Moses website.[3]

KenLM [7] is a library for building a language model and it is integrated into Moses. It provides language model queries that are both time-efficient and memory-efficient. GIZA++ [15] is the alignment toolkit used to construct a translation model in baseline Moses. After training, we obtain a language model, a translation model and a reordering model. These models are used by the Moses decoder to find the best translation for a given input sequence of words.

2.3.3 R^{2}NN SMT

The R^{2}NN SMT model consists of three parts: the translation confidence based phrase pair embedding (TCBPPE), global features, and an R^{2}NN decoder.

[3]https://www.statmt.org/moses/?n=Moses.Baseline

TCBPPE

A phrase pair embedding is a vector representation which encodes the meaning of a phrase pair. This was used as the **representation vector** in the R^{2}NN model.

Liu et al. [12] argued that the phrase pair embedding should capture the translation relationship between phrase pairs at phrase level. They presented a **translation confidence based phrase pair embedding** (TCBPPE) method and demonstrated that it gave a higher performance than simply taking the average of the embedding of words. Their phrase pair embedding is split into two parts: translation confidence with sparse features and translation confidence with a recurrent neural network. These two embeddings were obtained separately and then concatenated together to be used as the representation vector of each phrase pair.

Global features

The global features encode global information that cannot be generated by child representations. This was used as the **recurrent input vector** in the R^{2}NN model.

In the paper [12], three scores were selected to be the global features: language model scores, translation scores and reordering scores. These scores were concatenated together to be used as the recurrent vector of each phrase pair.

R^{2}NN decoder

A recursive recurrent neural network (discussed in Section 2.2.4) was used as a decoder to find the best translation candidate. For an input sentence, the decoder constructs a tree based on phrases in the sentence. An output score was computed based on TCBPPE, global features and the structure of the tree. The translation candidate that gives the highest score is considered to be the best translation candidate.

2.4 Software Engineering Techniques

2.4.1 Requirement Analysis

Based on the Project Structure section from my project proposal, the following requirements have been identified:

Data preprocessing

- Data should be prepared in a form that is accepted by Moses SMT and R^{2}NN SMT
- Preprocessing should be done carefully to avoid accidentally correcting some of the grammatical errors
  - E.g. capitalisation errors may go undetected if all sentences are lowercased during data preprocessing

Moses SMT for GEC

- A language model should be trained to model the probability of a given sentence being valid
- A translation model should be trained to construct a phrase translation table
- A reordering model should be trained to learn the reordering of phrases
- With the above three models and Moses decoder, a complete Moses SMT system should be built

R^{2}NN SMT for GEC

- Phrase pair embeddings should be learned by building a one-hidden-layer neural network and a recurrent neural network
- A recursive recurrent neural network (R^{2}NN) should be built and used as a decoder

Evaluation

- The performance of both SMT systems should be evaluated using $F_{0.5}$ scores

2.4.2 Development Methodology

I used the Agile methodology throughout the development of this project. Following the requirement analysis, the project can be divided into four modules: data preprocessing, Moses SMT, R^{2}NN SMT, and evaluation. Each module was built separately before being integrated together to be the final system. Flexibility was allowed within independent sub-modules. In addition, object oriented programming was used to develop the neural networks due to the modular structure of PyTorch.

2.5 Choice of Tools

2.5.1 Programming Language

Python is chosen to be the main programming language as it provides many libraries that are commonly used for natural language processing. For this project I used `python 3.8` and PyCharm as my IDE.

2.5.2 Libraries

PyTorch
The `PyTorch`[4] library is one of the most popular machine learning frameworks and it was used to implement the neural networks in this project.

[4]https://pytorch.org/

NumPy
My project is likely to involve statistical processing and for this purpose I used the `NumPy`[5] library.

pandas
`pandas`[6] is a powerful library for processing tabular data. It was used to manipulate phrase tables in my project.

Matplotlib
`Matplotlib`[7] is a visualisation library and it was used for plotting various graphs in this dissertation.

Care has been taken to address the licence requirements of the above libraries. Below is a summary of libraries used in this project.

Library	**Version**	**Licence**
Matplotlib	3.5.2	PSF licence
NumPy	1.22.1	BSD 3-Clause Licence
pandas	1.3.5	BSD 3-Clause Licence
PyTorch	1.10.1	BSD 3-Clause Licence

2.5.3 Dataset

I used the **FCE v2.1** corpus [18] in the BEA 2019 Shared Task as training, development and test data in this project. This is immediately downloadable from the website[8]. The corpora have been standardised to be easily evaluated by ERRANT [2, 5]. ERRANT is a toolkit used to annotate parallel data and compare a hypothesis against a reference to produce various evaluation metrics including the $F_{0.5}$ score. I may request other corpora as an extension of my project.

For language model training, I used the **One Billion Word** dataset [3]. This is a dataset used for language modeling and is available on GitHub[9].

The licences for the above datasets are listed below.

Dataset	**Licence**
FCE v2.1	Custom (non-commercial) licence
One Billion Word	Apache-2.0 licence

2.5.4 Version Control and Backup

Git was used for version control. The entire project, including all the written code and my dissertation, was pushed to GitHub regularly to protect against machine failures.

[5]https://numpy.org/
[6]https://pandas.pydata.org/
[7]https://matplotlib.org/
[8]https://www.cl.cam.ac.uk/research/nl/bea2019st/#data
[9]https://github.com/ciprian-chelba/1-billion-word-language-modeling-benchmark

Chapter 3

Implementation

This chapter concerns the implementation of the two major systems in this project. Section 3.1 describes how I set up the training data. In Section 3.2 and 3.3 I present the implementation of a baseline Moses SMT system and an R^2NN SMT system respectively. In Section 3.4 I give an overview of my code repository.

Since the main purpose of this project was to compare the performance of an R^2NN decoder against a Moses decoder, the R^2NN system made use of the language model and the translation model from the Moses system. Table 3.1 shows the dependencies between the Moses SMT and the R^2NN SMT.

SMT System	Number	Module	Dependency
Moses SMT	1	Language Model	None
	2	Translation Model	None
	3	Reordering Model	None
	4	Moses Decoder	1,2,3
R^2NN SMT	5	TCBPPE: Sparse Features	Phrase table from 2
	6	TCBPPE: RNN	Phrase table from 2
	7	Global Features	1,2
	8	R^2NN Decoder	5,6,7

Table 3.1: Dependencies between Moses SMT and R^2NN SMT

3.1 Setup

FCE Dataset

The FCE dataset contains 28327 sentences in the training set, 2187 sentences in the development set and 2695 sentences in the test set. All the files were provided in m2 format. However, Moses requires parallel data which is aligned at the sentence level. An example[1] taken from the m2 file looks like this:

```
S Her friend Pat had explained the whole story at her husband .
          A 8 9|||R:PREP|||to|||REQUIRED|||-NONE-|||0
```

The m2 format needs to be converted to sentence-aligned data to be used by Moses. A script[2] was provided on the BEA 2019 Shared Task website to generate the corrected text from an M2 file. Based on this script, I wrote some Python code to extract source sentences and corrected sentences into two separate files which were aligned at the sentence level.

For the example above, the source sentence extracted was

Her friend Pat had explained the whole story at her husband .

and the corrected sentence was

Her friend Pat had explained the whole story to her husband .

One Billion Word Dataset

I downloaded the One Billion Word dataset for language model training. However, the files were too large (about 18.2 GB) to be handled by my machine. Since the dataset consists of fifty files, I decided to randomly choose 10 files from them and concatenated these files to be used in language model training.

3.2 Moses Baseline

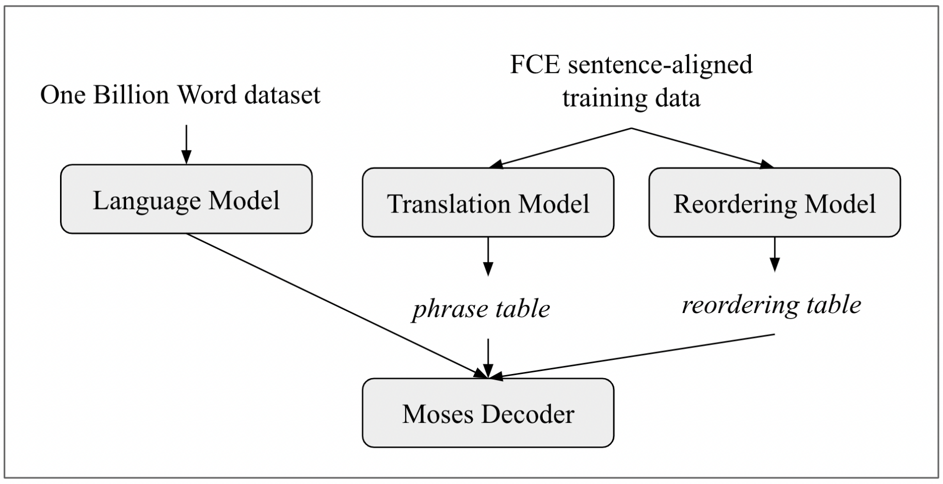


Figure 3.1: Overview of Moses SMT (training)

[1]This means that for source sentence S, the word at position 8 to 9 (*at*) should be corrected to the word *to*.

[2]https://www.cl.cam.ac.uk/research/nl/bea2019st/data/corr_from_m2.py

To build a baseline Moses SMT system, three models need to be trained: a language model, a translation model and a reordering model.

3.2.1 Language Model

The One Billion Word dataset was used for language model training. In the R^{2}NN paper, a 5-gram language model was used. Therefore, I used the KenLM toolkit which came with the Moses installation to train a 5-gram language model. As discussed in Section 2.3.1, this means that the language model would look at 4 words in the past. After training, the language model was able to compute a total score for any given sequence of words. The score represents the log probability of this sequence and it was used as the language model score (LM score) later on. The example below shows that our trained language model assigns a higher score to a valid English sentence compared to incorrect English sentences.

Sentence	**LM score**
I have an apple .	-9.682597
I have apple .	-11.434978
I has apple .	-13.841325

3.2.2 Translation Model and Reordering Model

The translation model was trained with FCE sentence-aligned data. There are three components in translation model training: word alignment, phrase extraction and scoring. Word alignment was obtained using GIZA++, a toolkit to train word alignment models. Phrases were then extracted from our training files and a **phrase table** was produced. A distance-based reordering model was also built, and a reordering table was created.

Table 3.2 shows part of the phrase table generated by Moses.[3] Each phrase pair from the training data has a unique entry in the phrase table. Four translation scores were computed for every phrase pair, namely inverse phrase translation probability `(target → source)`, inverse lexical weighting, direct phrase translation probability `(source → target)`, and direct lexical weighting. The phrase table also contains the alignment between the source phrase and the target phrase. The column named 'counts' contains three numbers. The first is the number of times the target phrase appeared in the training target file, `count(target)`. The second is the frequency of the source phrase in the training source file, `count(source)`. The third is the number of times that the source phrase is mapped to the target phrase, `count(source → target)`.

source	target	scores	alignment	counts
As we	As we	1 0.901657 0.846154 0.90511	0-0 1-1	11 13 11
As we	We	0.00115075 1.68879e-05 0.0769231 0.0153198	1-0	869 13 1
As we	as we	0.0212766 0.00117032 0.0769231 0.00482726	0-0 1-1	47 13 1

Table 3.2: Phrase table from Moses

[3]In the actual file, columns are separated by |||

After training, a configuration file named `moses.ini` was generated. By modifying the configuration file, we can change the language model, translation model and reordering model used by Moses decoder.

3.3 R^2NN SMT

There are three modules in the R^2NN SMT model: the translation confidence based phrase pair embedding (TCBPPE), global features, and an R^2NN decoder. Section 3.3.1 and Section 3.3.2 present implementation and training of two neural networks for sub-components of the TCBPPE, and Section 3.3.3 demonstrates how they are combined together to become the phrase pair embedding. Section 3.3.4 describes how I obtained global features, i.e. the recurrent input vector to R^2NN model. Finally, I discuss my implementation of an R^2NN decoder using the TCBPPE and global features in Section 3.3.5.

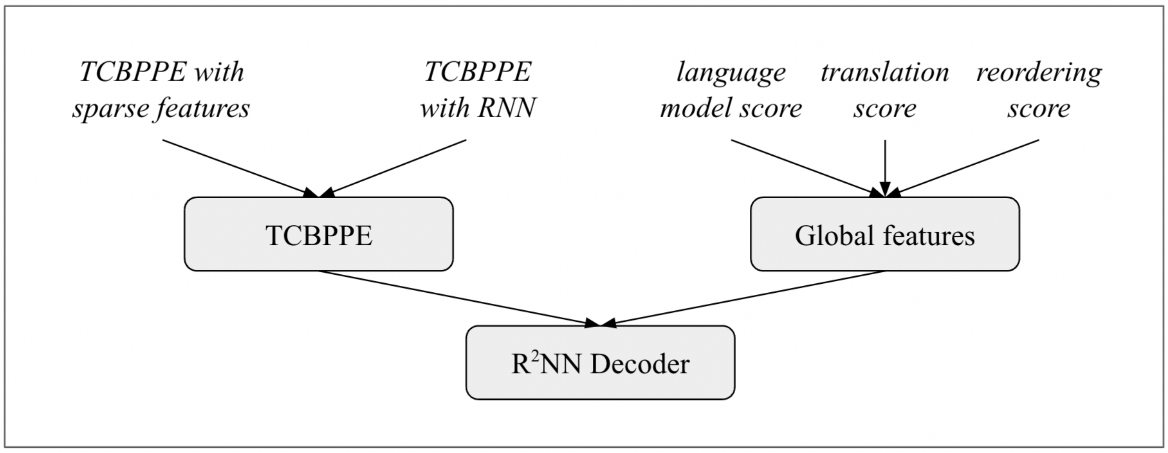


Figure 3.2: Overview of R^2NN SMT (training)

3.3.1 TCBPPE: Sparse Features

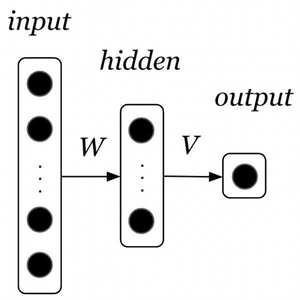


Figure 3.3: Structure of one-hidden-layer feedforward neural network

We use a one-hidden-layer feedforward neural network to obtain TCBPPE using sparse features as described in the paper [12]. The structure of the neural network is shown in

Figure 3.3, with the size of the hidden layer set to 20. To train the network, we need to encode training data as a sparse matrix. Each of the top 200,000 frequent phrase pairs in the Moses phrase table (see Section 3.2.2) is a feature, with an additional special feature added to represent all the infrequent phrase pairs. Therefore, for every sentence in the training data we obtain a binary vector of length 200,001. It only contains 1s in the positions where the phrase pair exists in the sentence.

By training the network, we obtain a hidden matrix W of size 200,001 by 20. W is used as the **phrase pair embedding matrix for sparse features** (`ppe_sparse`). The goal is to find the W that gives the best phrase pair embedding.

In the following sections, I will discuss the method used to obtain the top 200,000 phrase pairs since it is an essential component in encoding training data. I will describe how I encoded training data into a sparse matrix with the help of the top phrase table. Finally I will present the result of my network after training and how I obtained the first part of the TCBPPE with sparse features.

Obtain the top phrase table

Firstly, we need to obtain the 200,000 most frequent phrase pairs from the Moses phrase table in Section 3.2.2. Recall in Table 3.2, three counts were given in the Moses phrase table: the target phrase frequency, the source phrase frequency, and the phrase pair frequency. I used the `pandas` library to read the table and perform sorting on these counts. The phrase pairs were sorted in descending order based on their frequency of occurrence in the training data. If multiple phrase pairs have the same frequency, they were further sorted: first by target phrase frequency, and then by source phrase frequency, both in descending order. After sorting, a **top phrase table** was obtained by only keeping the top 200,000 rows of the table.

Encode training data

The next step was to encode our training data as a sparse matrix. Consider the following sentence pair:

Source sentence *What she would do ?*

Target sentence *What would she do ?*

The encoded vector (of length 200,001) for this sentence pair should look like this:

Encoded vector $[\underset{0}{0}, \ldots, \underset{39}{1}, \ldots]$

A 0 at position 0 means that the most frequent phrase pair does not exist in the sentence pair. A 1 at position 39 means that the 40th most frequent phrase pair exists in the sentence pair. If the sentence pair contains a phrase pair that is not in the top 200,000 most frequent pairs, it would contain a 1 at position 200,000.

One problem with this is that the resulting matrix is a sparse matrix of size `training_size` by 200,001. To save memory, instead of storing the whole matrix, the position indices of

1s in the vector were stored in a file named `phrase_pair_id` (i.e. the phrase pair ids used in each sentence pair). When loading the sentence pairs as training data, it was easy to retrieve the sparse matrix from `phrase_pair_id`.

Next, we need to get the expected output for each input sentence pair. Since the TCBPPE encodes translation model information, it is reasonable to use the translation scores from the Moses translation model. For each sentence pair S-T containing phrase pairs $p_1, p_2, ..., p_n$, the expected output score is computed by

$$score_{S\text{-}T} = \frac{\sum_n c_{p_n}}{n}$$

where c_{p_n} is the average translation score for phrase pair p_n.

Training the one-hidden-layer neural network

After encoding the training dataset, we can start training the neural network. Since the goal was to obtain the hidden matrix W that gives the best phrase pair embedding, we evaluated the performance of W after every epoch, i.e. one cycle of processing all the data in the training set, to decide when to stop training.

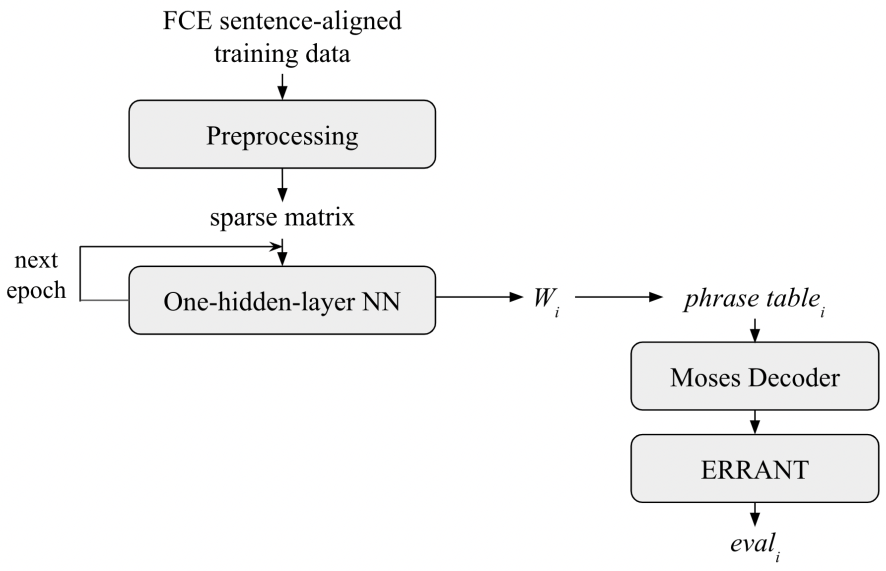


Figure 3.4: Training pipeline of the one-hidden-layer feedforward neural network

W was evaluated by updating the Moses phrase table. Each row in W was used as the phrase pair embedding (ppe) for the phrase pair it represents. Then the ppe was added to the phrase table as an additional feature. The updated phrase table was passed to the Moses decoder and ERRANT was used to evaluate the performance. Figure 3.4 demonstrates the training pipeline of the neural network.

The evaluation results $eval_i$ for W after epoch i are discussed in Section 4.3. Finally, the W that gave the best evaluation results was used as the first part of the TCBPPE (`ppe_sparse`).

3.3.2 TCBPPE: Recurrent Neural Network (RNN)

The other part of TCBPPE is a **translation confidence score** generated by a recurrent neural network (RNN). The structure of the RNN used by Liu et al. is shown in Figure 3.5, where e_i is the source word embedding, and f_{a_i} is the word embedding of the target word which is aligned to e_i. The translation confidence score from source s to target t is given by

$$T_{S2T}(s,t) = \sum_i \log p(e_i | e_{i-1}, f_{a_i}, h_i)$$

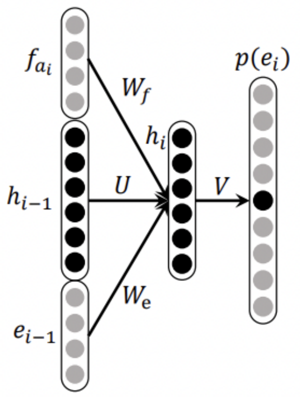


Figure 3.5: Recurrent neural network for translation confidence from the paper [12]

In the following sections I will describe the preprocessing of training data. I will illustrate my implementation of the RNN structure as shown in Figure 3.5 and how I trained the neural network to obtain the **translation confidence score** (`ppe_rnn`) of the TCBPPE.

Preprocess training data

Firstly, we need to decide what to use for word embedding. There are several popular pre-trained word embedding models available, such as GloVe[4] and fastText[5]. I used the fastText pre-trained models because fastText is based on character n-grams, whereas GloVe takes a word to be the smallest unit. For the task of Grammatical Error Correction, it is likely that out of vocabulary (OOV) words will occur in the source (such as misspelled words). fastText can easily generate word embeddings for those unseen words based on character n-grams.

Next, we construct a PyTorch dataset from the FCE training data. Consider a sentence pair with

$$s_1 s_2 \dots s_m$$

as a source sentence of m words, and

$$t_1 t_2 \dots t_n$$

[4]https://nlp.stanford.edu/projects/glove/
[5]https://fasttext.cc/docs/en/crawl-vectors.html

as a target sentence of n words. We use fastText to obtain a word embedding for each word in source and target. Let e_i be the word embedding for t_i in target, e_{i-1} be the word embedding for t_{i-1} i.e. the previous word in target, f_{a_i} be the word embedding for s_j which is the word in source that is aligned to t_i. The input to the RNN is e_{i-1} and f_{a_i}, and the output is the probability distribution of word e_i.

For special cases when either e_{i-1} does not exist (e.g. if e_i is the word embedding of the first word in the sentence) or when there is no f_{a_i} if e_i is not aligned to any word, a zero vector was used as the word embedding instead to avoid None Type error.

Training RNN

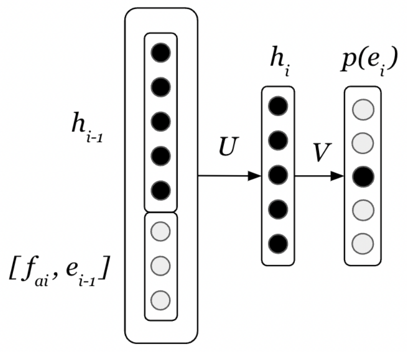


Figure 3.6: My implementation of an RNN according to Figure 3.5

Figure 3.6 shows the structure of my implementation of an RNN, following the paper [12]. The size of the word embedding was set to 50 to avoid memory issues, and the size of the hidden layer was set to 100. The input layer was a concatenation of f_{a_i}, e_{i-1} and the hidden layer h_{i-1}. `torch.nn.NLLLoss` was used to compute the negative log likelihood loss, and a log softmax layer was applied at the end to convert output to a categorical probability distribution.

The RNN was trained for 3 epochs and the model state was stored. Using the trained network, $T_{S2T}(s,t)$ for each phrase pair (s,t) in the top phrase table was computed by

$$T_{S2T}(s,t) = \sum_i \log p(e_i | e_{i-1}, f_{a_i}, h_i)$$

for e_i in source phrase and f_{a_i} in target phrase. This score was used as the other part of the TCBPPE for the phrase pair (s,t). After computing a score for all the phrase pairs in the top table, we obtained `ppe_rnn` of size 200,000 by 1.

3.3.3 TCBPPE

In Section 3.3.1, we obtained the first part of TCBPPE with sparse features, resulting in a matrix `ppe_sparse` of size 200,001 by 20. In Section 3.3.2, we obtained the second part of TCBPPE using a recurrent neural network, which gave a matrix `ppe_rnn` of size 200,000 by 1. To get the full phrase pair embedding, we need to concatenate `ppe_sparse` and `ppe_rnn` together to obtain a matrix of size 200,001 by 21, where the size of phrase pair embedding for each pair is of length 21. To achieve that, we need to append an element to the end of `ppe_rnn` to make it of size 200,001 by 1. I used value 0 for this purpose. The resulting matrix became the **representation vector** for all the phrase pairs, with the top 200,000 rows being the representation vector for the top 200,000 phrase pairs and the 200,001th row being the representation vector for all the infrequent phrase pairs.

3.3.4 Global Features

In the paper [12], three features were proposed to be used as global features, namely the language model score, translation model score and reordering model score. Language model scores can be obtained by querying the language model from Moses SMT (Section 3.2.1). Translation model scores are accessible from the Moses phrase table (Section 3.2.2). However, although a reordering table was created by Moses after training, it is unclear what each score means and Moses documentation did not give details of the reordering score components. In addition, Junczys-Dowmunt and Grundkiewicz disabled reordering in their experiment with phrase-based SMT [9] as word order errors are fairly rare in GEC. Hence, the reordering scores were omitted and only the translation model scores and the language model scores were used as global features in my implementation of R^2NN decoder.

For each phrase pair in the phrase table, we need to obtain a **language model score** and a **translation model score**. The language model score (LM score) was given by the trained 5-gram KenLM language model described in Section 3.2.1. The target phrase in each phrase pair was passed to the language model, and the resulting LM score was saved for this phrase pair. Four scores were calculated by Moses in the phrase table (Section 3.2.2). In order to make use of all four features, the translation model score was calculated by taking the average of the four scores for each phrase pair.

Finally, the language model score and the translation model score were concatenated to become a vector of length 2 and this was used as the **recurrent input vector** to the R^2NN model.

3.3.5 R^2NN Decoder

In the following sections, I will describe how I prepare the training data for R^2NN using forced decoding. I will present my implementation of an R^2NN by extending the implementation of a recursive neural network. Finally, I will discuss how I trained the recursive recurrent neural network.

Prepare training data

Following the R^2NN paper [12], **forced decoding** was used to generate positive samples. Forced decoding is a technique used to find out how the model derives a translation. We can perform forced decoding with Moses by constraining the output of the decoder to only be the reference sentences (i.e. the target sentences in FCE training dataset). This produced a file which contains the phrases used in the translation from source sentences to target sentences. From this file, we need to obtain

- the phrase pairs used in each sentence pair, and
- the scores associated with the translation.

Below is an example taken from the forced decoding output file.

```
TRANSLATION HYPOTHESIS DETAILS:
        SOURCE: [0..4] Dear Sir or Madam ,
  TRANSLATED AS: Dear Sir or Madam ,
  WORD ALIGNED: 0-0 1-1 2-2 3-3 4-4
SOURCE/TARGET SPANS:
  SOURCE: 0-1-2-3-4
  TARGET: 0-1-2-3-4
SCORES (UNWEIGHTED/WEIGHTED): core
   =(0.000,-5.000,1.000,-0.108,...,-31.078,0.000)
```

We extract the following information from this translation:

Phrase pairs used
Dear Sir or Madam , $\rightarrow$ *Dear Sir or Madam ,*

Scores
(0.000, -5.000, 1.000, -0.108, ..., -31.078, 0.000)

The phrase pairs used in this translation indicate how the source sentence was split into phrases. The list of source phrases is the input to the R^2NN decoder. The scores are the log probabilities computed from the features used in the decoder. We sum up the scores to be the **forced decoding score**, and this is the expected output score for this translation.

Building an R^2NN

Since no implementation of an R^2NN is publicly available, I developed my own version as follows.

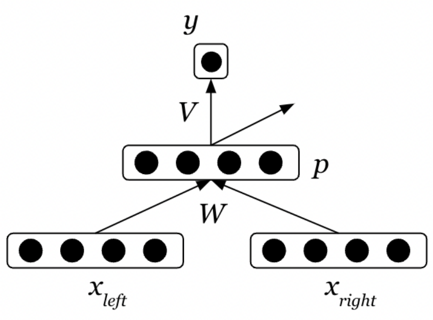


Figure 3.7: My implementation of a recursive neural network (RvNN)

As demonstrated in Figure 2.4, a recursive recurrent neural network (R^{2}NN) has a tree-like structure, so it was a good idea to start with building a recursive neural network (RvNN) which also has a tree structure. Inspired by a simple implementation of an RvNN [4], I constructed a network with the structure shown in Figure 3.7, where x_{left} and x_{right} represent the left child node and the right child node respectively. The parent node representation is computed as

$$p = W([x_{left}; x_{right}])$$

where $[a; b]$ denotes the concatenation of vectors a and b. The output score is computed as

$$y = V(p)$$

A separate class, `TreeNode`, was defined to make use of the above RvNN. A method named `greedy_tree` was used to construct a tree from a list of leaf nodes based on the output score from the RvNN. In principle, given a list of leaf node representations, a parent node (called **hypothesis**) was constructed for every pair of neighbouring nodes, and the parent node that gave the highest score computed by the RvNN was kept. The chosen neighbouring nodes were removed from the list, and their position in the list was replaced by their parent. This process was performed iteratively until only one tree node was left, which became the root node of the constructed tree. The final output score assigned to this tree was obtained by passing the tree to the RvNN. An algorithm to achieve this is given in Algorithm 1.

Algorithm 1 An algorithm which constructs a tree using a greedy method

```
$l \leftarrow list\ of\ leaf\ representation$
$n \leftarrow size(l)$
while $n \geq 2$ do
    $score_{max} \leftarrow -\infty$
    $hypothesis_{max} \leftarrow None$
    $index \leftarrow None$
    for $i \leftarrow 1, n$ do
        $hypothesis \leftarrow new\ TreeNode$
        $hypothesis.left \leftarrow l[i-1]$
        $hypothesis.right \leftarrow l[i]$
        $score \leftarrow RvNN(hypothesis)$
        if $score > score_{max}$ then
            $score_{max} \leftarrow score$
            $hypothesis_{max} \leftarrow hypothesis$
            $index \leftarrow i$
        end if
    end for
    $l[index-1 : index+1] \leftarrow [hypothesis_{max}]$
end while
```

My implementation of a recursive recurrent neural network (R^{2}NN) was based on the structure of the RvNN and `TreeNode`. There are two main differences between the RvNN shown in Figure 3.7 and the R^{2}NN discussed in Section 2.2.4:

- In the RvNN, we have exactly one vector for each node. In the R^{2}NN, each node has two vectors: a **representation vector** and a **recurrent input vector**. The representation vector is the translation confidence based phrase pair embedding (TCBPPE) which was discussed in sections 3.3.1, 3.3.2 and 3.3.3. The recurrent input vector encodes global information for the phrase pair and was addressed in Section 3.3.4.
- In order to be able to retrieve the translation from source to target, each node in the R^{2}NN should store the phrase pair string in addition to the numeric vectors. For example, if the left node phrase pair is

  $$the \rightarrow The$$

  and the right node phrase pair is

  $$apple \rightarrow apple$$

  then the parent node should have a translation pair

  $$the\ apple \rightarrow The\ apple$$

Therefore, Algorithm 1 needs to be modified by the following:

- The representation vector and the recurrent input vector are added to the neural network and they are used to compute a *score* which determines the best combination of child nodes;
- The phrase pair string is added to `TreeNode` as an attribute. When constructing a hypothesis (i.e. combining two child nodes), the phrase pair string of each child node should be joined in the hypothesis.

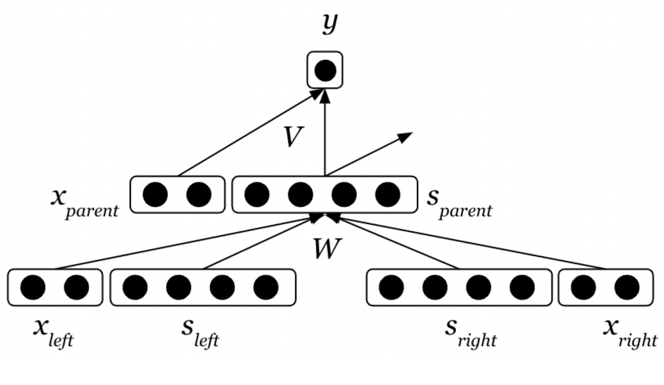


Figure 3.8: My implementation of a recursive recurrent neural network (R^{2}NN)

Figure 3.8 shows my implementation of an R^{2}NN. s_{left} and s_{right} is the TCBPPE for the left phrase pair and the right phrase pair respectively. x_{left} and x_{right} are the recurrent input vectors. The phrase pair embedding for the parent node, s_{parent}, is computed by

$$s_{parent} = f(W[x_{left}; s_{left}; x_{right}; s_{right}])$$

I used the same function f as Liu et al. in their paper [12], i.e. the $HTanh$ function:

$$HTanh(x) = \begin{cases} -1 & x < -1 \\ x & -1 \leq x \leq 1 \\ 1 & x > 1 \end{cases}$$

The output score y is computed as

$$y = V[x_{parent}; s_{parent}]$$

where the recurrent input vector for parent x_{parent} is obtained independently of the child nodes.

Training the R^{2}NN

In the R^{2}NN paper [12], Liu et al. used the following loss function:

$$Loss(W, V, s) = \max(0, 1 - y_{oracle} + y_t) \tag{3.1}$$

where s is the source sentence span, y_{oracle} is the forced decoding translation score, and y_t is the output score of an R^{2}NN translation hypothesis. The max function is used to guarantee a non-negative loss.

Recall that for each forced decoding translation we obtained two pieces of information: the **phrase pairs used** in the translation, and the **forced decoding score**. The forced decoding score is used as y_{oracle} which is the expected output score for this translation. The phrase pairs used in the translation form the initial leaf nodes, and the `greedy_tree` method is used to construct a tree from the leaf nodes. The output score given by the R^{2}NN model to this tree is y_t.

To better illustrate the idea, consider the following example taken from the training data.

Source sentence
It was so long time to wait in the theatre .

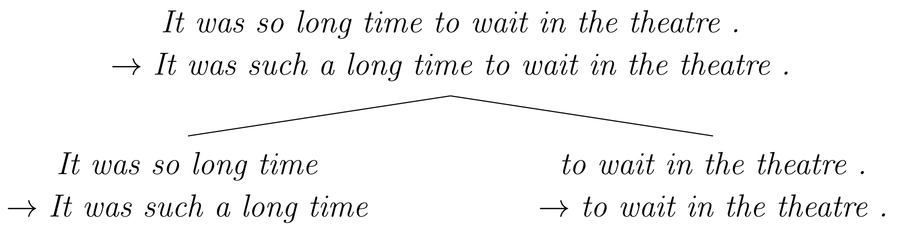
Target sentence
It was such a long time to wait in the theatre .

Phrase pairs used
It was so long time → *It was such a long time*
to wait in the theatre . → *to wait in the theatre .*

Forced decoding score
-77.797

The phrase pairs were used as the leaf nodes. For each of the two phrase pairs, the representation vector s and the recurrent input vector x were obtained. These vectors would be used by the R^{2}NN model to compute their parent representation vector, s_{parent}. This was used with the recurrent input vector of their parent x_{parent} to compute an output score y_t for this tree in the R^{2}NN model. The forced decoding score for this translation was used as y_{oracle}. Finally, the loss was computed using Equation 3.1 and the model state was updated. The R^{2}NN was trained for 1 epoch and the training took about five hours due to the complexity in tree construction.

Each node in the constructed tree has three attributes: the representation vector s, the recurrent input vector x, and the phrase pair string. The two vectors were used to compute the parent node representation, and the phrase pair string was used to retrieve the translation.


It was so long time to wait in the theatre .
→ It was such a long time to wait in the theatre .

It was so long time
→ It was such a long time

to wait in the theatre .
→ to wait in the theatre .

Looking at the above tree, the root node reveals what the source sentence is and what the translation hypothesis is. This feature was planned for testing (Section 4.4.2) to allow easy extraction of the translation hypothesis for any constructed tree.

3.4 Repository Overview

Figure 3.9 shows the repository structure of my project.

`errant/` contains a fork of the GitHub[6] repository `chrisjbryant/errant` [2, 5] which is used to annotate and score the hypothesis file (translated sentences) to the reference file (target sentences).

`fastText/` contains a fork of the GitHub[7] repository `facebookresearch/fastText` [6] which generates word embeddings in RNN training. The binarised model is saved to `part-ii-project/model/RNN/cc.en.50.bin`.

`giza-pp/` contains a fork of the GitHub[8] repository `moses-smt/giza-pp` [15] and it is used in Moses SMT to obtain alignment models.

`mosesdecoder/` contains a fork of the GitHub[9] repository `moses-smt/mosesdecoder` [11]. Moses decoder, as well as tools for building associated models can be found here.

`part-ii-project/` is where the main code of the project resides.

`corpus/` contains the downloaded FCE corpus as well as the preprocessed sentence-aligned FCE corpus.

[6]https://github.com/chrisjbryant/errant
[7]https://github.com/facebookresearch/fastText
[8]https://github.com/moses-smt/giza-pp
[9]https://github.com/moses-smt/mosesdecoder

`evaluation/` contains the intermediate files and output files from ERRANT.

`helper_scripts/` contains scripts to preprocess FCE corpus and scripts to perform some simple tasks such as plotting.

`lm/` contains the binary file of the language model from Moses.

`moses_exp/` is the working directory for Moses decoder and contains the intermediate files generated by Moses in Section 3.2.

`model/` includes three sub-directories, `sparse/`, `RNN/` and `R2NN/`, as well as related files (such as the top phrase table `pt_top`).

The following sections give further details of `part-ii-project/model/`.

3.4.1 `sparse/`

`top_phrase_table.py`
Process Moses phrase table to obtain the top 200,000 phrase pairs, `pt_top`.

`one_hot_encode.py`
Encode sentences in FCE training dataset as one-hot like tensors.

`calc_avg_feature.py`
Calculate the expected feature score for each sentence in FCE training dataset.

`one_hidden_layer_net.py`
Class definition for one-hidden-layer feedforward neural network.

`train_one_hidden_layer.py`
Code to load the dataset and perform training.

`update_phrase_table.py`
Write the new features obtained from the neural network to the phrase table.

`save_ppe_sparse.py`
Save the parameter W of the model to a file, `ppe_sparse.npy`, to be the phrase pair embedding with sparse features.

3.4.2 `RNN/`

`recurrent_nn.py`
Class definition for recurrent neural network (RNN).

`train_rnn.py`
Code to load the dataset and perform training.

`save_ppe_rnn.py`
Save the output of RNN to a file, `ppe_rnn.npy`, to be the phrase pair embedding with RNN.

3.4.3 `R2NN/`

`get_ppe.py`
Combine `ppe_sparse.npy` and `ppe_rnn.npy` to be the phrase pair embedding matrix, `ppe_matrix.npy`.

`obtain_pt_target.py`
Obtain all the target phrases from phrase table for querying Moses language model.

`save_top_lm.py`
Write the language model scores to the phrase table.

`R2NN.py`
Class definition for recursive recurrent neural network (R^2NN).

`train_r2nn.py`
Code to process Moses forced decoding output, create a PyTorch dataset, and train the R^2NN.

```
errant/
fastText/
giza-pp/
mosesdecoder/
part-ii-project/
├── corpus/
├── evaluation/
├── helper_scripts/
├── lm/
├── model/
│   ├── R2NN/
│   │   ├── R2NN.py
│   │   ├── get_ppe.py
│   │   ├── obtain_pt_target.py
│   │   ├── recursive_nn.py
│   │   ├── save_top_lm.py
│   │   ├── test_r2nn.py
│   │   ├── train_r2nn.py
│   │   └── r2nn_state/
│   ├── RNN/
│   │   ├── cc.en.50.bin
│   │   ├── recurrent_nn.py
│   │   ├── save_ppe_rnn.py
│   │   ├── train_rnn.py
│   │   └── rnn_state/
│   ├── sparse/
│   │   ├── calc_avg_feature.py
│   │   ├── one_hidden_layer_net.py
│   │   ├── one_hot_encode.py
│   │   ├── save_ppe_sparse.py
│   │   ├── top_phrase_table.py
│   │   ├── train_one_hidden_layer.py
│   │   ├── update_phrase_table.py
│   │   ├── one_hidden_layer/
│   │   └── phrase_tables/
│   └── ...
├── moses_exp/
│   ├── tuned_moses.ini
│   └── ...
└── project-reports/
```

Figure 3.9: Repository Overview

Chapter 4

Evaluation

4.1 Success Criteria

All of the success criteria specified in the project proposal have been met and are presented below.

- ✓ Data is pre-processed for the use of training the models *(Section 3.1)*
- ✓ A baseline SMT-based GEC system is built *(Section 3.2)*
- ✓ A one-hidden-layer neural network is built to learn the translation confidence score *(Section 3.3.1)*
- ✓ A recurrent neural network is built to learn the translation confidence score *(Section 3.3.2)*
- ✓ An R^{2}NN model is implemented *(Section 3.3.4 and 3.3.5)*
- ✓ An SMT-based GEC system making use of the R^{2}NN decoder is built *(Section 3.3.5)*
- ✓ A comparison between the performances of the two systems is performed *(Chapter 4)*

4.2 Evaluation Metrics

The ERRANT scorer [2, 5], which is the official metric of the BEA 2019 Shared Task [1], was used to evaluate the performance of the Moses SMT-based GEC system and the R^{2}NN SMT-based GEC system. Compared to other evaluation scripts commonly used in machine translation (such as BLEU [16]), ERRANT is more informative for GEC because it computes the $F_{0.5}$ score based on span-based correction.

The $F_{0.5}$ score is computed based on the precision P and the recall R. Let T_P denote a true positive where a system **correctly** made a change, F_P denote a false positive where a system **incorrectly** made a change, and F_N denote a false negative where a system **missed** a change that it should have made. Precision and recall can hence be calculated as follows.

$$P = \frac{T_P}{T_P + F_P}$$

$$R = \frac{T_P}{T_P + F_N}$$

$F_{0.5}$ is the weighted harmonic mean of P and R and is computed as

$$F_{0.5} = (1 + 0.5^2) \times \frac{P \times R}{0.5^2 \times P + R}$$

This weighs precision more than recall, since a change in P impacts less on the denominator than the same change applied to R. As a result, P will give a greater perturbation in the $F_{0.5}$ score since the numerator is evenly impacted by both P and R.

4.3 Evaluating the ppe_sparse Matrix

In Section 3.3.1, I described how I trained a one-hidden-layer neural network and obtained the hidden matrix W after each epoch. To decide what to use for the ppe_sparse matrix, we evaluate the performance of the Moses decoder using ERRANT after encoding W as an additional feature to the phrase table.

Figure 4.1, Figure 4.2 and Figure 4.3 show how the accuracy, recall and $F_{0.5}$ score change as the number of epochs increases. Training for 10 epochs gave the highest accuracy and recall as well as $F_{0.5}$ score. For this reason, W after epoch 10 was used as the first part of the TCBPPE (ppe_sparse).

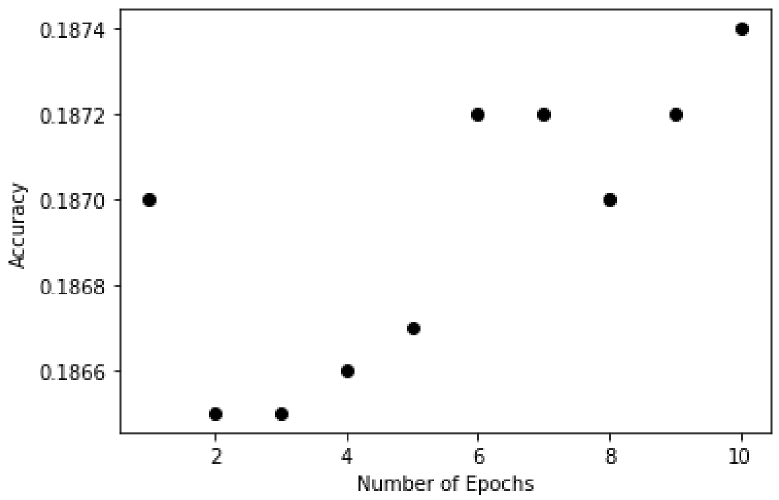


Figure 4.1: Plot of accuracy produced by ERRANT against the number of epochs W was trained for

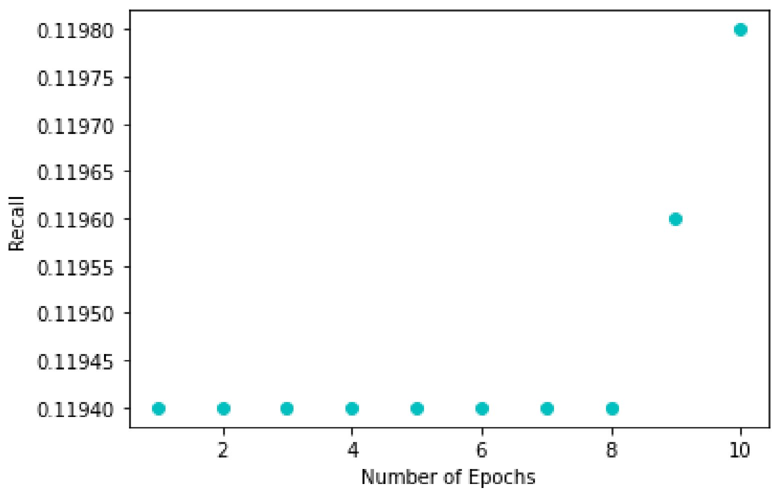


Figure 4.2: Plot of recall produced by ERRANT against the number of epochs W was trained for

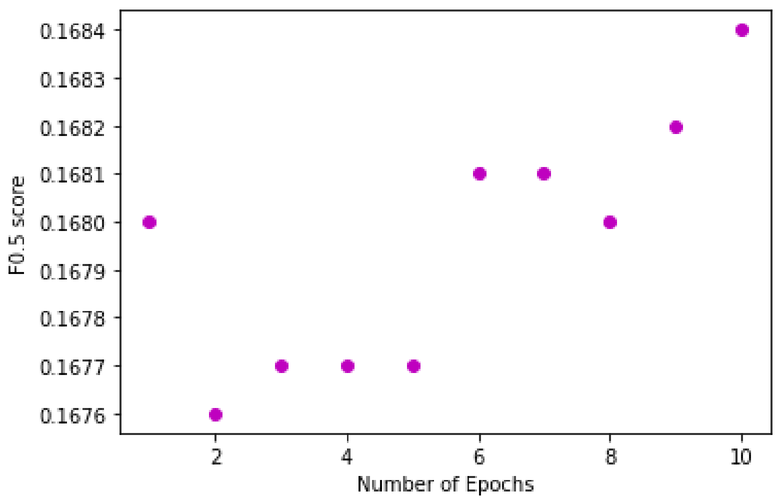


Figure 4.3: Plot of $F_{0.5}$ score produced by ERRANT against the number of epochs W was trained for

4.4 Testing

The held out FCE test set was used for testing. The m2 file of the FCE test set was processed as described in Section 3.1. We obtained two sentence-aligned files: one containing all the source sentences in the test set and one containing all the target sentences (gold reference). After the Moses decoder and the R^{2}NN decoder processed the source file, each system produced a translation file containing the translations of the source sentences. We can then use ERRANT to evaluate the translation file against the gold reference file. ERRANT first aligns the source text with the system output to produce a hypothesis m2 file. This file can then be compared with the reference m2 file, and we can count the matching edits in terms of T_P, F_P and F_N to compute the $F_{0.5}$ score.

4.4.1 Moses Decoder

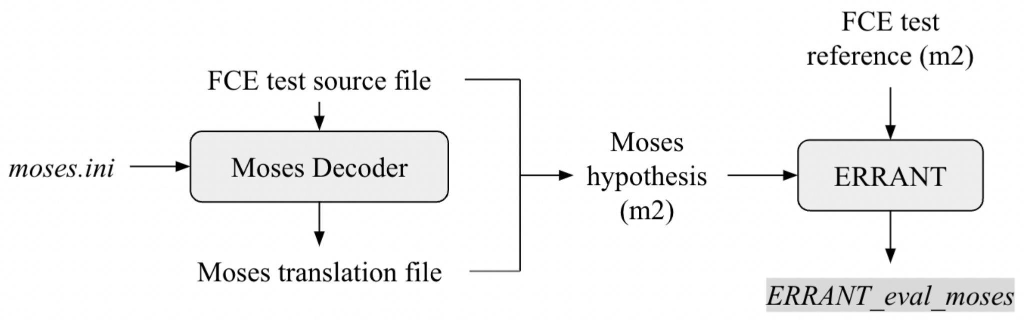


Figure 4.4: Testing pipeline of the Moses SMT

Before testing the Moses decoder, we should tune the parameters using a small amount of parallel data which is different from the training data. The FCE development set was used for this purpose. The tuning process optimized the weights for each feature as specified in `moses.ini`. Then we used the Moses decoder to translate the test set and produce a hypothesis file which contains the translation given by Moses. The evaluation results were stored in a file named `ERRANT_eval_moses`.

4.4.2 R^{2}NN Decoder

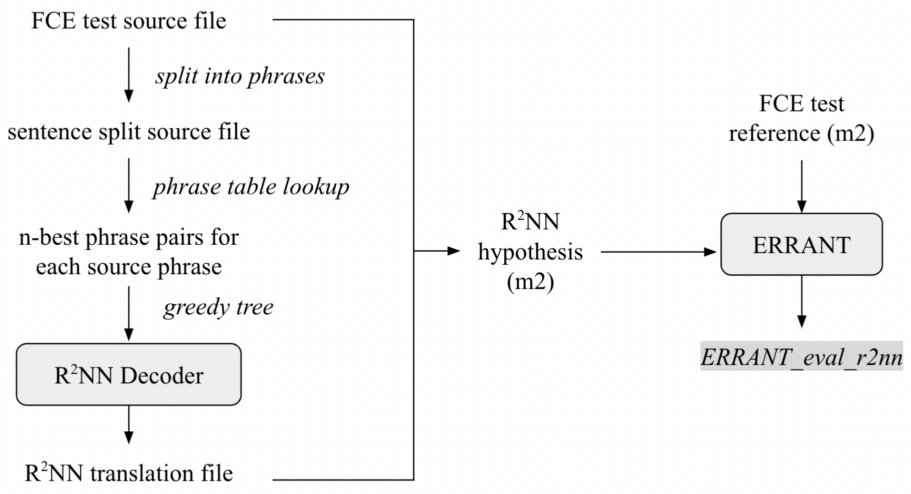


Figure 4.5: Testing pipeline of the R^{2}NN SMT

To use the R^{2}NN decoder, firstly we need to split the sentence into phrases. I proposed and implemented a search method: starting from the beginning word of a sentence, we look for the longest possible phrase that appears in our top phrase table. If a phrase is found, we keep the phrase and move on to the remaining part of the sentence; if no such phrases are found, we make that word a phrase, and move on to the next word. Intuitively, this would lead to using longer phrases which retain more contextual information than shorter phrases. An example is given below.

Sentence
I hope that he will recover soon and that he will make it to our conference .

Sentence split
I hope that / he will / recover / soon and / that he / will make / it to / our / conference / .

The next step was to obtain phrase pairs based on the sentence split. For each source phrase in the split, a search was performed on the phrase table to obtain a list of possible phrase pairs. The R^{2}NN model was used to compute a score for each phrase pair and only the n-best phrase pairs that gave high scores were kept. Then we recursively repeat the process and a tree was built based on the selected phrase pairs. The final translation hypothesis for this sentence was obtained by accessing the phrase pair string attribute of the root node, where the source string would be identical to the source sentence and the target string was the translation hypothesis.

List of possible phrase pairs
I hope that $\rightarrow$ *I hope that*
I hope that $\rightarrow$ *I hope*
...

However, when it came to testing, I discovered that the search space was too big for my laptop to process. To process a sentence with k phrases while keeping the n best phrase pairs, the number of possible different trees is at least n^k. In order to finish testing within a reasonable amount of time and to reduce the complexity, I decided to keep the phrase pair that gives the highest score, i.e. $n = 1$. The evaluation results of the translation file were saved to `ERRANT_eval_r2nn`.

4.5 Results and Discussion

Table 4.1 and table 4.2 show the scores reported by ERRANT for the baseline Moses system and the R^{2}NN system.

TP	FP	FN	Prec	Rec	F0.5
479	632	4070	0.4311	0.1053	0.2663

Table 4.1: ERRANT scorer output for Moses on FCE test set

TP	FP	FN	Prec	Rec	F0.5
605	7088	3944	0.0786	0.133	0.0856

Table 4.2: ERRANT scorer output for R^{2}NN on FCE test set

We observe that the $F_{0.5}$ score of R^{2}NN is lower than the baseline Moses system. Even though the R^{2}NN system gives a higher recall (lower proportion of missed edits), its precision is lower than that of Moses. Since $F_{0.5}$ weighs precision twice as much as recall, the resulting figure is lower. We also observe that the R^{2}NN gives more true positives but at the same time more false positives (FP) than the baseline Moses. It suggests that the R^{2}NN tends to make changes to the source sentence even when it should not have.

Consider the source sentence

Hope you like it .

The gold reference, as well as the Moses hypothesis, did not make changes to this sentence. On the other hand, the R^{2}NN hypothesis was

I hope you like it .

This contributed to the false negative of the R^{2}NN model. Even though the edit did not change the correctness of the source sentence, it was unnecessary. This edit was made because the source sentence was split into the following phrases

Hope you / like it .

and the best phrase pairs that give the highest score for the R^{2}NN model are

Hope you $\rightarrow$ *I hope you*
like it . $\rightarrow$ *like it .*

There are two ways to mitigate this problem.

- Recall in Section 4.4.2, we only kept the best phrase pair that gives highest score for memory issues. (*Hope you* $\rightarrow$ *I hope you*) was selected because it gave the highest score. Other phrase pairs, including (*Hope you* $\rightarrow$ *Hope you*), were not considered by R^{2}NN for upper tree combination. If we extend it to n-best phrase pairs at the cost of expanding the search space, we would take the correct phrase pair into consideration.
- Another possibility is to use a different method to split the source sentence into phrases. For example, if the sentence was split into one phrase (with the phrase being the whole sentence), the correct phrase pair (*Hope you like it .* $\rightarrow$ *Hope you like it .*) would be obtained.

The FCE test set also contains sentences with more than one error. Consider the following example:

Source sentence
The best way is definetely , by air .
Gold reference
The best way is definitely by air .
Moses hypothesis
The best way is definitely , by air .
R^{2}NN hypothesis
The best way is definitely by air .

The source sentence contains multiple orthographic errors: a spelling mistake in *definetely*, and a punctuation mistake in the use of the comma. Moses corrected the misspelled word but failed to see the punctuation mistake, whereas the R^{2}NN model captured and corrected both errors successfully.

There are some interesting translation results produced by the R^{2}NN SMT. In the following example, two commas are put next to each other in the source sentence and the quotation mark does not come in pairs.

The most important building of Biasco is the , , Casa Cavalier Pellanda " .

Both the gold reference and the Moses hypothesis are identical to the source sentence. However, the output of the R^{2}NN model was

The most important building of Biasco is " Casa Cavalier Pellanda "

As we can see, the R^{2}NN model removed the two commas and added an opening quotation mark to match the closing quotation mark. The R^{2}NN model also removed the word *the* and the ending full stop. In my opinion, the correct sentence should be

The most important building of Biasco is the " Casa Cavalier Pellanda " .

which is different to the gold reference in the dataset. This reflects the complexity of GEC: human annotators do not always agree. It is also interesting that the R^{2}NN captured the punctuation errors of commas and quotation marks, but at the same time introduced more errors by removing the full stop and the article *the*.

Overall, the R^{2}NN model has its potential: we saw that it successfully spotted some of the errors that were missed by the Moses SMT and even the human annotator. However, it tends to make changes to the original sentences even when the changes are not necessary, resulting in more false positives than Moses. This has two implications: a lower precision as precision measures the proportion of **correct** edits out of all edits (large denominator), and a higher recall as recall measures the proportion of **missed** edits. For GEC, high recall may be more useful for teachers and learners since it will flag up their mistakes more often. The $F_{0.5}$ score of the R^{2}NN model was lower than the Moses SMT, but future researchers can refer to the implementation of such an R^{2}NN model and more work can be done to improve it. Since these two systems have different strengths and weaknesses, further development in the GEC field may combine elements of both methods.

Chapter 5

Conclusion

5.1 Summary of Work Completed

This project fulfilled all of the success criteria as demonstrated in Section 4.1. In this project, I successfully implemented two SMT-based GEC systems: a phrase-based baseline Moses SMT, and an R^{2}NN SMT. For Moses SMT, I started with building a working SMT system and trained it with FCE data. For R^{2}NN SMT, I constructed phrase pair embedding (TCBPPE) using two neural networks, a feedforward neural network and a recurrent neural network. I implemented the R^{2}NN model from scratch. Then I built the R^{2}NN SMT by integrating the TCBPPE and global features into the R^{2}NN decoder. Finally, a comparison was performed on the performance of Moses SMT-based GEC and R^{2}NN SMT-based GEC. R^{2}NN achieved better recall but due to its low precision, overall it gives a lower $F_{0.5}$ score than Moses.

5.2 Personal Reflections

Throughout this project, I gained more insight into neural networks and natural language processing, particularly in the area of grammatical error correction. I learnt how to build a neural network from scratch using PyTorch, and my experience with the PyTorch library will certainly be helpful in my future career.

One important lesson I took from this project is that there will always be unforeseen difficulties. For example, there were some system compatibility problems when I was installing Moses. There were times when my system would complain about running low on memory and stopped execution before it could produce any meaningful data after running for hours. There is no way I could predict all that, but the best I could do is to allow extra time for each task and not to leave it until the last minute. Another fact that I did not realise is that it is challenging to build a neural network from scratch, especially when the neural network has a novel structure. If I was going to start this project again, I would have put more time into constructing the recursive recurrent neural network (R^{2}NN) because it was far more challenging than the theory in the paper [12] looked like.

5.3 Future Work

Due to the time constraint of this project, I did not refine the R^{2}NN SMT further but there are several possible ways to improve its performance.

1. Inevitably, unseen words will exist no matter how big our training set is, especially for the task of GEC where the input data could contain misspelled words. In my implementation of the R^{2}NN, I kept the original word/phrase for unseen data. This may ignore errors like spelling mistakes. Alternatively we could expand the size of phrase tables but at the cost of memory and searching time.

2. Recall in Section 4.4.2, I only kept the best phrase pair for upper tree combination to avoid having a huge search space. It is more sensible to keep the n-best phrase pairs so that our model can consider the possibility of a tree which does not use all of the *best* phrase pairs and yet produce a higher score.

3. The dataset used to train our R^{2}NN SMT system is small. We could use other training data available on BEA 2019 [1] in addition to the FCE dataset.

4. Before a sentence is passed to the R^{2}NN, it has to be split into phrases. The way I split the sentence is described in Section 4.4.2, but other algorithms to split the sentence may be used to obtain different initial phrases and hence different initial tree leaves.

Bibliography

[1] Christopher Bryant, Mariano Felice, Øistein E. Andersen, and Ted Briscoe. The BEA-2019 shared task on grammatical error correction. In *Proceedings of the Fourteenth Workshop on Innovative Use of NLP for Building Educational Applications*, pages 52–75, Florence, Italy, August 2019. Association for Computational Linguistics.

[2] Christopher Bryant, Mariano Felice, and Ted Briscoe. Automatic annotation and evaluation of error types for grammatical error correction. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 793–805, Vancouver, Canada, July 2017. Association for Computational Linguistics.

[3] Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, and Tony Robinson. One billion word benchmark for measuring progress in statistical language modeling, 2013.

[4] dkohl. Daniel@world: Recursive neural networks. `https://daniel-at-world.blogspot.com/2019/09/recursive-neural-networks.html`. Accessed: 2022-03-02.

[5] Mariano Felice, Christopher Bryant, and Ted Briscoe. Automatic extraction of learner errors in ESL sentences using linguistically enhanced alignments. In *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers*, pages 825–835, Osaka, Japan, December 2016. The COLING 2016 Organizing Committee.

[6] Edouard Grave, Piotr Bojanowski, Prakhar Gupta, Armand Joulin, and Tomas Mikolov. Learning word vectors for 157 languages. In *Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018)*, 2018.

[7] Kenneth Heafield. KenLM: Faster and smaller language model queries. In *Proceedings of the Sixth Workshop on Statistical Machine Translation*, pages 187–197, Edinburgh, Scotland, July 2011. Association for Computational Linguistics.

[8] Marcin Junczys-Dowmunt and Roman Grundkiewicz. The AMU system in the CoNLL-2014 shared task: Grammatical error correction by data-intensive and feature-rich statistical machine translation. In *Proceedings of the Eighteenth Con-*

ference on Computational Natural Language Learning: Shared Task, pages 25–33, Baltimore, Maryland, June 2014. Association for Computational Linguistics.

[9] Marcin Junczys-Dowmunt and Roman Grundkiewicz. Phrase-based machine translation is state-of-the-art for automatic grammatical error correction. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1546–1556, Austin, Texas, November 2016. Association for Computational Linguistics.

[10] Daniel Jurafsky and James H. Martin. *Speech and Language Processing (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2009.

[11] Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondřej Bojar, Alexandra Constantin, and Evan Herbst. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th Annual Meeting of the ACL on Interactive Poster and Demonstration Sessions*, ACL '07, page 177–180, USA, 2007. Association for Computational Linguistics.

[12] Shujie Liu, Nan Yang, Mu Li, and Ming Zhou. A recursive recurrent neural network for statistical machine translation. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1491–1500, 2014.

[13] Hwee Tou Ng, Siew Mei Wu, Ted Briscoe, Christian Hadiwinoto, Raymond Hendy Susanto, and Christopher Bryant. The CoNLL-2014 shared task on grammatical error correction. In *Proceedings of the Eighteenth Conference on Computational Natural Language Learning: Shared Task*, pages 1–14, Baltimore, Maryland, June 2014. Association for Computational Linguistics.

[14] Hwee Tou Ng, Siew Mei Wu, Yuanbin Wu, Christian Hadiwinoto, and Joel Tetreault. The CoNLL-2013 shared task on grammatical error correction. In *Proceedings of the Seventeenth Conference on Computational Natural Language Learning: Shared Task*, pages 1–12, Sofia, Bulgaria, August 2013. Association for Computational Linguistics.

[15] Franz Josef Och and Hermann Ney. A systematic comparison of various statistical alignment models. *Computational Linguistics*, 29(1):19–51, 2003.

[16] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, page 311–318, USA, 2002. Association for Computational Linguistics.

[17] Michael Paul. Overview of the IWSLT 2009 evaluation campaign. In *Proceedings of the 6th International Workshop on Spoken Language Translation: Evaluation Campaign*, pages 1–18, Tokyo, Japan, December 1-2 2009.

[18] Helen Yannakoudakis, Ted Briscoe, and Ben Medlock. A new dataset and method for automatically grading ESOL texts. In *Proceedings of the 49th Annual Meeting*

of the Association for Computational Linguistics: Human Language Technologies, pages 180–189, Portland, Oregon, USA, June 2011. Association for Computational Linguistics.

[19] Zheng Yuan. *Grammatical error correction in non-native English.* PhD thesis, University of Cambridge, Cambridge, United Kingdom, 3 2017.

[20] Zheng Yuan, Ted Briscoe, and Mariano Felice. Candidate re-ranking for SMT-based grammatical error correction. In *Proceedings of the 11th Workshop on Innovative Use of NLP for Building Educational Applications*, pages 256–266, San Diego, CA, June 2016. Association for Computational Linguistics.

[21] Zheng Yuan and Mariano Felice. Constrained grammatical error correction using statistical machine translation. In *Proceedings of the Seventeenth Conference on Computational Natural Language Learning: Shared Task*, pages 52–61, Sofia, Bulgaria, August 2013. Association for Computational Linguistics.

Appendix A

Example Code

A.1 TreeNode

```
class TreeNode:
    def __init__(self, vector=None, source="", target=""):
        self.vector = vector
        self.source = source
        self.target = target
        self.left = None
        self.right = None

    def greedy_tree(self, sentence_span_tuple, model):
        sentence_span = [[tup[0] for tup in pair] for pair in
            sentence_span_tuple]
        num_pairs = len(sentence_span)
        leafs = []
        for pair_id in range(num_pairs):
            source_phrase = sentence_span[pair_id][0]
            target_phrase = sentence_span[pair_id][1]
            pair_rec = get_rec(source_phrase, target_phrase)
            pair_ppe = get_ppe(source_phrase, target_phrase)
            pair_vector = torch.concat((pair_rec, pair_ppe), 0)

            leaf_node = TreeNode(vector=pair_vector, source=source_phrase,
                target=target_phrase)
            leafs.append(leaf_node)

        while len(leafs) >= 2:
            max_score = float('-inf')
            max_hypothesis = None
            max_idx = None
            for i in range(1, len(leafs)):
                hypothesis = TreeNode()
                hypothesis.left = leafs[i - 1]
                hypothesis.right = leafs[i]
                vector, score = model(hypothesis)
                hypothesis.vector = vector
                hypothesis.source = hypothesis.left.source + " " +
                    hypothesis.right.source
                hypothesis.target = hypothesis.left.target + " " +
                    hypothesis.right.target
                if score > max_score:
                    max_score = score
                    max_hypothesis = hypothesis
                    max_idx = i
            leafs[max_idx - 1:max_idx + 1] = [max_hypothesis]
        return leafs[0]
```

A.2 R^{2}NN

```
class R2NN(torch.nn.Module):
    def __init__(self, ppe_size, rec_size):
        super(R2NN, self).__init__()
        self.input_size = ppe_size + rec_size
        self.W = torch.nn.Linear(self.input_size * 2, ppe_size, dtype=
            torch.float32)
        self.V = torch.nn.Linear(self.input_size, 1, dtype=torch.float32)

    def forward(self, node):
        left_node_vector = torch.zeros([self.input_size], dtype=torch.
            float32)
        left_node_source = ""
        left_node_target = ""
        right_node_vector = torch.zeros([self.input_size], dtype=torch.
            float32)
        right_node_source = ""
        right_node_target = ""

        if node.left is None and node.right is None:
            left_node_vector = node.vector
            left_node_source = node.source
            left_node_target = node.target
        if node.left is not None:
            left_node_vector = node.left.vector
            left_node_source = node.left.source
            left_node_target = node.left.target
        if node.right is not None:
            right_node_vector = node.right.vector
            right_node_source = node.right.source
            right_node_target = node.right.target

        inp_vector = torch.cat([left_node_vector, right_node_vector], 0)
        parent_ppe = torch.tanh(self.W(inp_vector))

        parent_source = left_node_source + " " + right_node_source
        parent_target = left_node_target + " " + right_node_target
        parent_rec = get_rec(parent_source, parent_target)

        parent = torch.cat([parent_rec, parent_ppe], 0)
        output = self.V(parent)
        return parent, output
```

Appendix B

Project Proposal

PROJECT PROPOSAL

Introduction

Statistical machine translation (SMT) is an approach to machine translation based on statistical models. Grammatical error correction (GEC), which is the task of producing grammatically correct text given potentially erroneous text while preserving its meaning, can be seen as a translation process from an erroneous source to a correct target. As such, it is possible to build an SMT system and apply it to the task of GEC.

A typical SMT system consists of four main components: The language model (LM), the translation model (TM), the reordering model and the decoder (Yuan, 2017). The LM computes the probability of a given sequence being valid. The TM builds a translation table which contains mappings of words/phrases between source and target corpora. The reordering model learns about phrase reordering of translation. The decoder finds a translation candidate who is most likely to be the translation of the source sentence. In terms of GEC, this would be the most probable correction to the original erroneous sentence.

A *recursive recurrent neural network* (R^2NN) was proposed by Liu et al. (2014) for SMT. It has features of both recursive neural networks and recurrent neural networks. This R^2NN network can be used to model the decoding process in SMT. A three-step training method was also given in the paper to train the R^2NN network.

This project aims to implement the proposed R^2NN decoder and build an SMT-based GEC system with it. The GEC system should aim to correct all types of errors, including grammatical, lexical, and orthographical errors. Considering that the original paper used data from IWSLT 2009 dialog task which could be out-of-date, this project will instead make use of the BEA-2019 shared task dataset, which is publicly available at (https://www.cl.cam.ac.uk/research/nl/bea2019st/), for training and testing purposes. Its performance will be evaluated against a baseline SMT system, namely Moses (Koehn et al., 2007). Moses is an open-source toolkit for SMT, and it has been used for the task of GEC (Yuan and Felice, 2013). When building the Moses baseline system, a language model is to be built and several figures such as the distortion score can be calculated. Since the proposed R^2NN decoder is to be used with a 5-gram language model and the training process requires distortion scores, it can make use of the calculated scores and the built language model when building Moses. After both SMT systems (i.e. Moses and R^2NN) are built, it is natural to evaluate the performance of the R^2NN decoder against Moses decoder.

Starting Point

The Part IB Computer Science Tripos course Artificial Intelligence [1] gives an introduction to neural networks and explains how forwarding and backpropagation works. The paper on R^2NN (Liu, et al., 2014) demonstrates the idea of combining recursive neural networks and recurrent neural networks, but implementation details or example codes are not provided. At the time of writing this proposal, I have no experience with using recursive neural networks or recurrent neural networks, which means I will need to study relevant materials for this project.

An example SMT system is available on Moses website[2]. By following the tutorial and build a Moses SMT system I should gain familiarity with the various models used in SMT.

The datasets I plan to use in this project are available on the BEA 2019 Shared Task website[3] for non-commercial purposes. The corpora provided are standardised, making it easy to perform pre-processing of data.

Project Structure

The project mainly consists of the following components:

1. Data preparation
2. Implementation of an SMT-based GEC system using Moses
    a. Language model training
    b. Translation and reordering model training
    c. Testing
3. Implementation of an SMT-based GEC system using R^2NN model
    a. Phrase pair embedding implementation
        i. Implementation of a one-hidden-layer neural network
        ii. Implementation of a recurrent neural network
        iii. Translation confidence score training
    b. R^2NN decoder implementation
        i. R^2NN model implementation
        ii. R^2NN model training
    c. Testing
4. Evaluation
    a. Comparison of performances of Moses and the R^2NN model

Success Criteria

CORE TASKS

[1] https://www.cl.cam.ac.uk/teaching/2021/ArtInt/
[2] http://www.statmt.org/moses/index.php?n=Main.HomePage
[3] https://www.cl.cam.ac.uk/research/nl/bea2019st/

This project will be successful if the following have been achieved.

1. Data is pre-processed for the use of training the models.
2. A baseline SMT-based GEC system is built.
3. A one-hidden-layer neural network is built to learn the translation confidence score.
4. A recurrent neural network is built to learn the translation confidence score.
5. An R^2NN model is implemented.
6. An SMT-based GEC system making use of the R^2NN decoder is built.
7. A comparison between the performances of the two systems is performed.

Possible Extensions

1. Train both GEC systems on an alternative GEC dataset and evaluate their performances
2. Experiment with different language models for both GEC systems
3. Experiment with alternative phrase pair embeddings for the GEC system using R^2NN

Timetable

9th Oct – 22nd Oct 2021

Background research on: statistical machine translation (SMT), Moses SMT, recursive neural networks, recurrent neural networks. Get familiar with relevant libraries.

Friday 15th Oct 2021 (5pm)

Final Proposal Deadline

23rd Oct – 5th Nov 2021

Build a Moses SMT system. Prepare the data for training the SMT.

6th Nov – 19th Nov 2021

Begin to implement phrase pair embedding. Build a one-hidden-layer neural network to learn translation confidence.

20th Nov – 3rd Dec 2021

Continue implementing phrase pair embedding. Build a recurrent neural network to learn translation confidence.

4th Dec – 21st Jan 2022 [Christmas Break]

Build an R^2NN decoder. Start with building a recursive neural network, then add recurrent input vectors to it. Train the parameters of the R^2NN decoder.

22nd Jan – 4th Feb 2022

Evaluate the performance of Moses SMT and the R^{2}NN-based SMT. Write the progress report.

Friday 4th Feb 2022 (12 noon)

Progress Report Deadline

5th Feb – 18th Feb 2022

Start with possible extensions if the core project is finished. Begin writing the dissertation.

19th Feb – 4th Mar 2022

Continue working on extensions if the core project is finished and writing the dissertation.

5th Mar – 18th Mar 2022

Continue writing the dissertation.

19th Mar – 22nd Apr 2022 [Easter Break]

Continue writing the dissertation. Send in the draft for review.

23rd Apr – 6th May 2022

Submit the dissertation.

Friday 13th May 2022 (12 noon)

Dissertation Deadline

Resources Declaration

For this project I will be using my personal laptop (1.9GHz quad-core Intel Core i7 processor, 8GB RAM, Windows 10). I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure. I will back up the entire project on Google Drive and use git for revision control. I might use a cloud GPU (available in Google Colab) to accelerate the speed of training the neural networks if needed.

The dataset from BEA-2019 shared task, which will be used in this project, can be found on (https://www.cl.cam.ac.uk/research/nl/bea2019st/). The training time for small models using only the public data from BEA-2019 shared task is expected to be a few hours, and for bigger models (if I plan to use additional data) it could be a few days. I should keep this in mind and leave enough time for training when implementing my project.

References

Liu, S., Yang, N., Li, M. & Zhou, M., 2014. *A Recursive Recurrent Neural Network for Statistical Machine Translation*.

Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondřej Bojar, Alexandra Constantin, Evan Herbst, 2007. *Moses: Open Source Toolkit for Statistical Machine Translation.* Association for Computational Linguistics.

Yuan, Z., 2017. *Grammatical error correction in non-native English*.

Yuan, Z., Felice, M., 2013. *Constrained grammatical error correction using Statistical Machine Translation,* Association for Computational Linguistics.

Christopher Bryant, Mariano Felice, Øistein E. Andersen and Ted Briscoe. 2019. *The BEA-2019 Shared Task on Grammatical Error Correction.* In Proceedings of the 14th Workshop on Innovative Use of NLP for Building Educational Applications (BEA-2019), pp. 52–75, Florence, Italy, August. Association for Computational Linguistics.