Jiaxin Wang

A Recursive Recurrent Neural Network Decoder for Grammatical Error Correction

Computer Science Tripos – Part II

Emmanuel College

April 28, 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 [signature]

Date [date]

Proforma

Candidate Number:	-
Project Title:	A Recursive Recurrent Neural Network Decoder for Grammatical Error Correction
Examination:	Computer Science Tripos – Part II, 2022
Word Count:	8082 words[1]
Code line count:	1136 [2]
Project Originator:	-
Supervisor:	Dr Zheng Yuan, Dr Christopher Bryant

Original Aims of the Project

This project aims to build an SMT system for grammatical error correction using a recursive recurrent neural network (R^2NN) model as the decoder. It involves constructing 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) will be built and trained with the same data. The R^2NN SMT will be 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, a comparison was carried out on the evaluation results of the two SMT systems for the task of grammatical error correction.

[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`

Special Difficulties

None.

Acknowledgements

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 ESL (English as a second language) learner myself, it is often difficult for me to spot the errors in my English writing since the 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, SMT-based 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 called *recursive recurrent neural network* (R^2NN)[11]. It has a tree structure similar to 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 the R^2NN SMT achieved better performance than a state-of-the-art baseline system on a Chinese-to-English translation task. However, they did not publicised their code, and no implementation of such R^2NN can be found on the Internet.

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 NLP (natural language processing), 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[15]. 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 task 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 specified in the BEA 2019 Shared Task[1], namely grammatical, lexical and orthographic errors. Lexical errors are mistakes at word level, such as wrong choice of words that do not fit in their context. Grammatical errors violate the grammar of a certain language. Orthographic errors refer to incorrect spelling of words.

Moses[10] will be 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 will build and train a Moses SMT system. I will implement and train an R^2NN SMT system which uses the R^2NN model as the decoder. Finally, I will compare the performance of both systems on the task of grammatical error correction.

1.3 Related Work

The paper[11] by Liu et al. 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. After training the system on data from the IWSLT 2009 dialog task, 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. 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[17] applied Moses SMT with GIZA++ and IRSTLM to grammatical error correction in the CoNLL-2013 Shared Task[13]. Junczys-Dowmunt and Grundkiewicz[8] developed a GEC system based on an SMT system, namely phrase-based Moses[10], and they ranked on the third place in the CoNLL-2014 Shared Task[12]. Yuan et al.[16] 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^2NN together with TCBPPE in the paper[11] and build a SMT system. A phrase-based Moses SMT system[10] will be built to be used as a

baseline system. Both systems will be applied to GEC and will be evaluated by ERRANT scorer[2][5], the evaluation tool specified in 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 in the paper *A Recursive Recurrent Neural Network for Statistical Machine Translation*[11], so this is the key paper my project would base on. I have not used Moses[10] before, but a tutorial is available on the official website[1]. This tutorial covers how to install Moses, how to prepare training data and how to train it.

The Part IB Computer Science Tripos course Formal Models of Language[2] presents the foundation theory of Natural Language Processing, while Artificial Intelligence[3] gives an introduction to neural networks and explains how forwarding and backpropagation works. Prior to this project I did not have any coding experience with neural networks.

2.2 Theory

2.2.1 SMT system

The book *Speech and Language Processing*[9] 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

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

[1] https://www.statmt.org/moses/?n=Moses.Baseline
[2] https://www.cl.cam.ac.uk/teaching/2021/ForModLang/
[3] https://www.cl.cam.ac.uk/teaching/2021/ArtInt/

where $P(F)$ is ignored inside $argmax_E$ 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 we have 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.

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 we can compute the **translation probability** using the alignment and construct a phrase translation table. The table should contain all the phrase pairs and the associated probabilities. 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, we expand the hypothesis by finding possible phrase translation for words in the source sentence F. A cost is computed, and we would keep the n-lowest cost translation at each stage. We will keep expanding it until we have covered all words in F. Eventually, we output the translation that gives the lowest cost.

2.2.2 Moses SMT

Moses[10] is a statistical machine translation toolkit. A tutorial on how to build a baseline SMT system is available on Moses website[4].

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++[14] is the alignment toolkit used to construct a translation model in baseline Moses. After training, we will obtain a language model, a translation model and a re-ordering model. These models will be used by Moses decoder to find the best translation for a given input sequence of words.

2.2.3 Feedforward Neural Network

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.

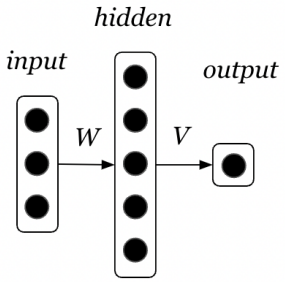

Figure 2.1: Example of a feedforward neural network

2.2.4 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,

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

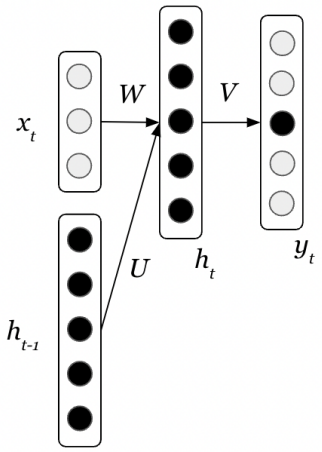

Figure 2.2: Example of a recurrent neural network (RNN)

the hidden layer h_t is computed using both input 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:

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

2.2.5 Recursive Neural Network (RvNN)

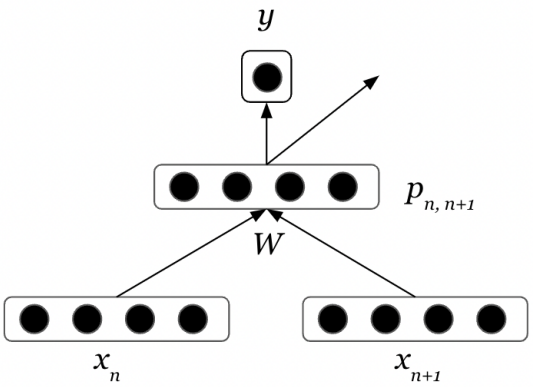

Figure 2.3: Example of a recursive neural network (RvNN)

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:

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

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

2.2.6 Recursive Recurrent Neural Network (R^2NN)

The R^2NN proposed by Liu et al.[11] 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 $s^{[m,n]}$ is the representation 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 distortion 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

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

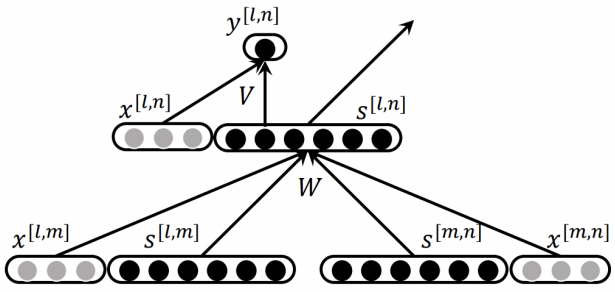

Figure 2.4: Recursive recurrent neural network (Liu et al., 2014, p.1494)

2.3 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^2NN 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

R²NN SMT for GEC

Following the R^{2}NN paper by Liu et al.[11],

- Phrase pair embeddings (PPE) 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 F0.5 scores

2.4 Choice of Tools

2.4.1 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 will be using `python 3.8` and PyCharm as my IDE.

2.4.2 Libraries

PyTorch
The `PyTorch`[5] library is one of the most popular machine learning frameworks. There are other similar libraries (such as `TensorFlow`) but I find `PyTorch` tutorials are easier to follow.

NumPy
My project is likely to involve statistical processing. I would be using the `NumPy`[6] library for this purpose.

pandas
`pandas`[7] is a powerful library for processing tabular data. This would be used to manipulate phrase tables in my project.

[5] https://pytorch.org/
[6] https://numpy.org/
[7] https://pandas.pydata.org/

2.4.3 Dataset

The dataset introduced in BEA 2019 Shared Task[1] will be used in this project. I chose the corpora (**FCE v2.1**) which is immediately downloadable from the website[8] to start with. The corpora have been standardised to be easily evaluated by ERRANT[2][5]. ERRANT is a toolkit used to annotate parallel data and compare hypothesis against reference to produce various evaluation metrics including F0.5 score. I may request other corpora as an extension of my project.

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

2.4.4 Version Control and Backup

Git will be used for version control. The entire project, including all the written code and my dissertation, will be pushed to GitHub regularly.

[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 describes the implementation of a Moses baseline SMT system and an R^{2}NN SMT system. Since the main purpose of this project is to compare the performance of R^{2}NN decoder against Moses decoder, the R^{2}NN system should use the same language model, translation model and reordering model as Moses. For the translation model, the R^{2}NN paper[11] proposed a *translation confidence based phrase pair embedding* (TCBPPE) to be used with the R^{2}NN decoder. The TCBPPE will base on a phrase table produced by Moses translation model. In the end, the R^{2}NN decoder will make use of language model scores, translation model scores, reordering model scores from Moses and TCBPPE to find the best translation candidate.

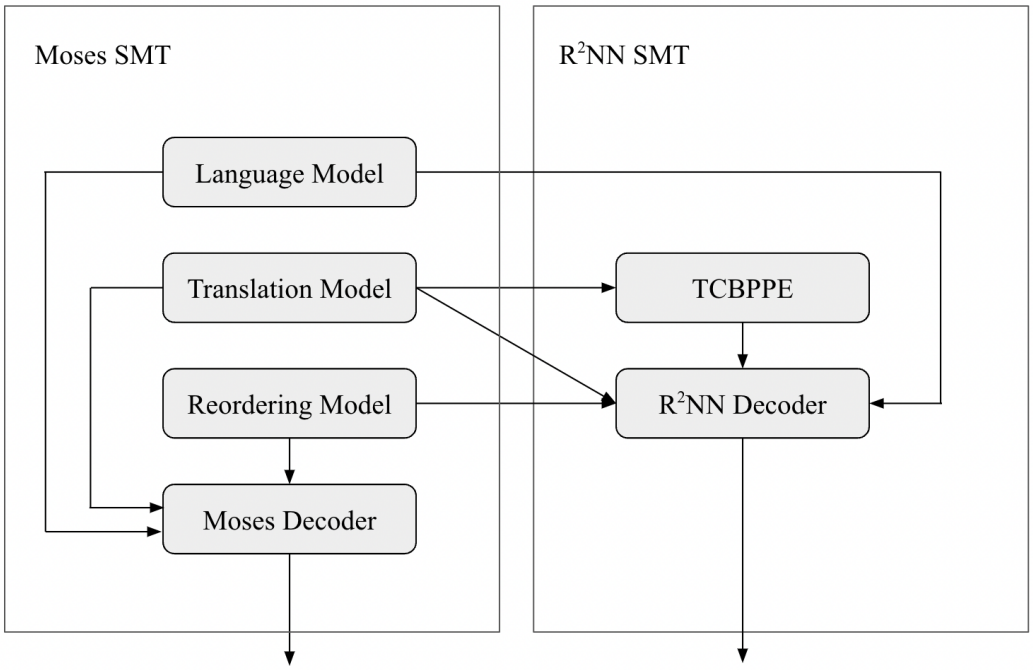

Figure 3.1: Overview of Moses SMT and R^{2}NN SMT

3.1 Setup

FCE Dataset

The FCE dataset was provided in m2 format. However, Moses requires parallel data which is aligned at the sentence level. An example 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`

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

The m2 format needs to be converted to sentence-aligned data to be used by Moses. Two files are generated from this, where the source file contains
Her friend Pat had explained the whole story at her husband .

and the target file contains
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 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

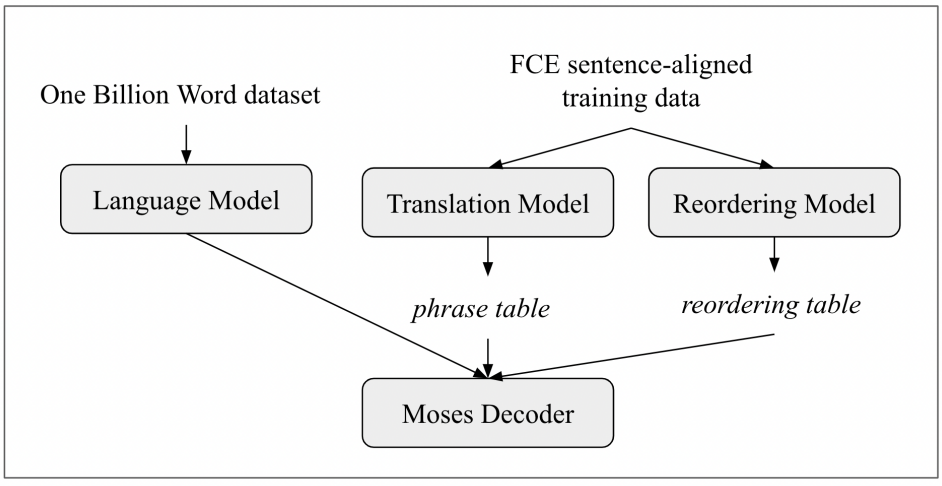

Figure 3.2: Overview of Moses SMT (training)

3.2.1 Language Model

The One Billion Word dataset is used for language model training. In the R²NN paper, a 5-gram language model is used. Here, I used KenLM which came with Moses installation to train a 5-gram language model. After training, the resulting language model is named `billion-training-monolingual-10.blm.en`. The following command is used to query the language model:

```
~/github/mosesdecoder/bin/query -n
    /path/to/billion-training-monolingual-10.blm.en < queries.txt
```

where `queries.txt` contains the sentences to be scored by the language model. For each sentence, the language model would return a total score (language model score), which is the log probability of this sentence. The example below shows that our trained language model assigns a higher score to a valid English sentence compared to incorrect English sentences.

I have an apple . LM score: −9.682597
I have apple . LM score: −11.434978
I has apple . LM score: −13.841325

3.2.2 Translation Model and Reordering Model

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

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 3.1: Phrase table from Moses

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

3.3 R²NN SMT

The R²NN SMT consists of three parts: translation confidence based phrase pair embedding (TCBPPE), global features, and an R²NN decoder.

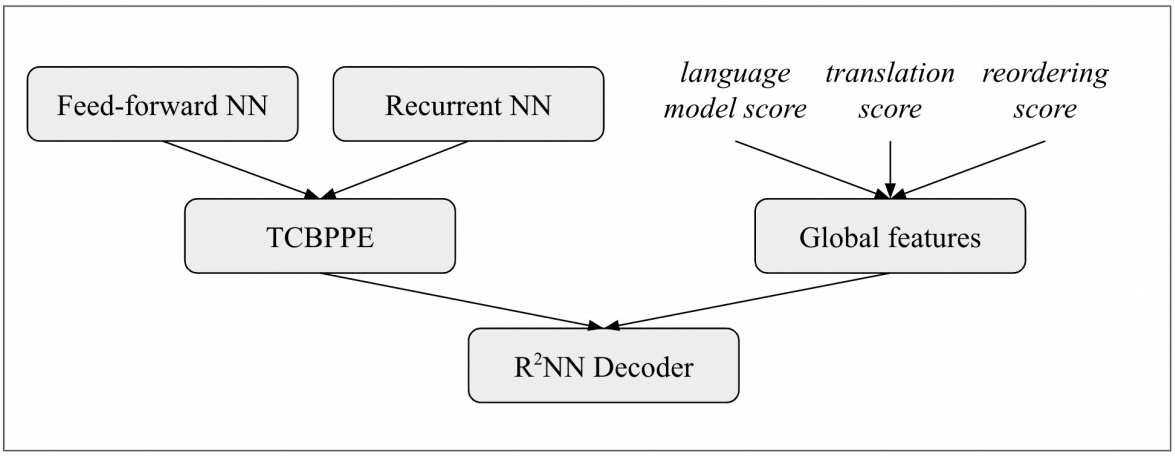

Figure 3.3: Overview of R^{2}NN SMT (training)

TCBPPE

A phrase pair embedding is a vector representation which encodes the meaning of a phrase pair. Recall that R^{2}NN model has a tree structure, where each node has a representation vector s and a recurrent vector x. The TCBPPE is to be used as the representation vector s and it is used to generate the leaf nodes of the derivation tree.

The phrase pair embedding is split into two parts: translation confidence with sparse features and translation confidence with recurrent neural network. These two vectors will be obtained separately, and will be concatenated to be used as the representation vector s in the R^{2}NN.

Global features

The global features encode global information that cannot be generated by child representations. It includes language model scores, translation scores and reordering scores which are obtained from Moses. These scores are concatenated together to be used as recurrent vectors x to the R^{2}NN model.

R^{2}NN decoder

The recursive recurrent neural network (R^{2}NN) will be used as a decoder to find the best translation candidate. For an input sentence, the decoder would construct a tree based on phrases in the sentence. An output score will be computed based on TCBPPE, global features and the structure of the tree. The translation candidate that gives the highest score will be taken as the best translation candidate.

3.3.1 TCBPPE: Sparse Features

A one-hidden-layer feedforward neural network is used to get a translation confidence score using sparse features. The structure of the network is shown in Figure 3.4. Following the R^{2}NN paper[11], the size of the hidden layer is set to 20. The size of the input layer is

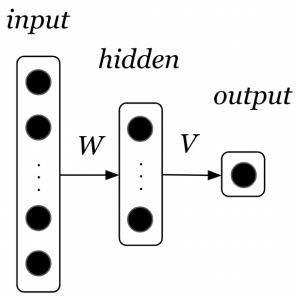

Figure 3.4: Structure of one-hidden-layer feedforward neural network

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

Top phrase table

A top phrase table `pt_top` will be used when encoding training data. It contains the top 200,000 most frequent phrase pairs that appeared in the training data. We can obtain `pt_top` from Moses phrase table using `pandas`. Recall that a column named "counts" is given in Moses phrase table (Table 3.1). It contains three numbers: the first is the number of times the target phrase appeared in training target file; the second is the number of times the source phrase appeared in training source file; the third is the number of times that the source phrase is mapped to the target phrase. To find the 200,000 most frequent phrase pairs, we can use `pandas` to perform a sort on "counts", and keep the top 200,000 rows. If `pt` is the phrase table loaded by `pandas`, then

```
pt[["count_co", "count_or", "count_or_co"]]
                          = pt["count"].str.split(expand=True)
pt_sort = pt.sort_values(["count_or_co", "count_co", "count_or"],
                          ascending=False)
most_freq_pt = pt_sort.head(200000)
```

would result in a table with 200,000 rows after sorting the counts column in descending order, with priority of

$$\text{phrase pair count} > \text{target count} > \text{source count}$$

Prepare training data

Next, we need to encode the training data (sentences) as one-hot like data to be used as input to the network. The phrase pairs in `pt_top` will each be a feature, and an additional feature is used for all the infrequent phrase pairs. For each sentence pair (source-target) in the training data, it needs to be encoded as a vector of length 200,001 consisting of 0s and 1s, where the position i in the vector contains a 1 if and only if the i^{th} most frequent phrase pair is found in this sentence pair. An example taken from the training file is shown below.

Source sentence *What she would do ?*

Target sentence *What would she do ?*

Encoded vector $[\underset{0}{0}, \ldots, \underset{39}{1}, \ldots, \underset{200000}{1}]$

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 40^{th} most frequent phrase pair exists in the sentence pair. A 1 at position 200,000 means that the sentence pair contains a phrase pair that is not in the top 200,000 most frequent pairs.

index	source	target	...
0	.	.	...
...	...	...	...
39	do	do	...

Table 3.2: Top 200,000 phrase pairs

One problem with this is that the resulting vector from sentence pairs are usually sparse vectors. To save memory, instead of storing the whole vector, the position indices of 1s in the vector are stored in a file named `phrase_pair_id` (i.e. the phrase pair ids used in each sentence pair are stored). When loading the sentence pairs as training data, it is easy to obtain the one-hot encoding of each sentence pair from `phrase_pair_id`.

The next step is to obtain the expected output for each input sentence pair. Since the TCBPPE encodes translation model information, it is reasonable to use the translation scores from 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 one-hidden-layer neural network

After obtaining the training dataset, we can train the neural network. The neural network will be trained for 10 epochs, where one epoch refers to one cycle of processing all the data in training set. After each epoch, the hidden matrix W is stored. The k^{th} row in W will be used as the phrase pair embedding (sparse) for the top k^{th} frequent phrase pair. That is, the embedding for the most frequent phrase pair is $W[0]$, and the embedding for phrase pairs that are not in the top 200,000 is represented by $W[200000]$.

We evaluate W by updating Moses phrase table: for each row (ppe) in W, we add the ppe as an additional feature to the phrase table. Then we pass the updated phrase table to Moses decoder and use ERRANT to evaluate the performance. Table 3.3 and Table 3.4 shows the evaluation result by ERRANT for training epoch 5 and training epoch 10 respectively. Among all the W from epoch 1 to 10, 10-epoch W gives the highest precision, recall and $F_{0.5}$ score. Hence, W will be used as the ppe matrix for sparse features (`ppe_sparse`).

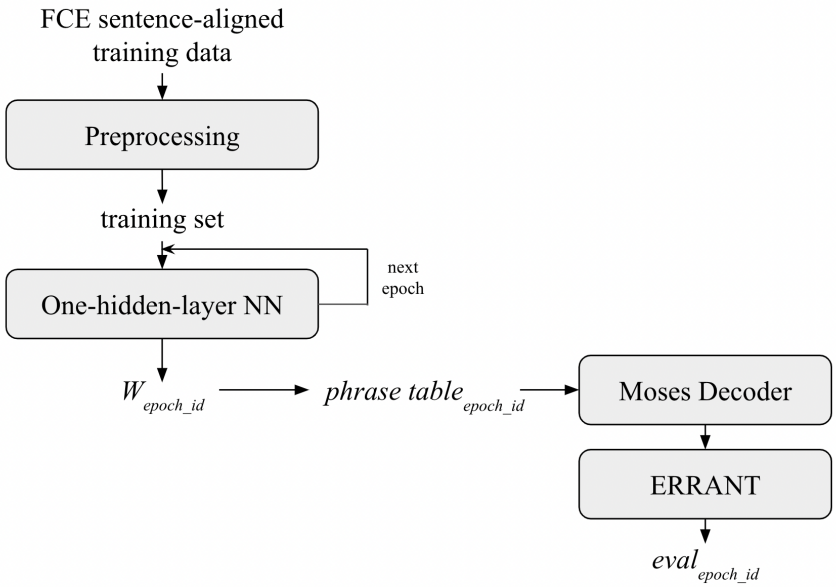

Figure 3.5: Training pipeline of one-hidden-layer feedforward neural network

TP	FP	FN	Prec	Rec	F0.5
543	2366	4006	0.1867	0.1194	0.1677

Table 3.3: ERRANT output after training for 5 epochs

TP	FP	FN	Prec	Rec	F0.5
545	2363	4004	0.1874	0.1198	0.1684

Table 3.4: ERRANT output after training for 10 epochs

3.3.2 TCBPPE: Recurrent Neural Network (RNN)

The other part of TCBPPE is generated by a recurrent neural network (RNN). The structure of the RNN used in shown in Figure 3.6, 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

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

Building RNN

Firstly, we need to decide what to use for word embeddings. There are several popular pre-trained word embeddings available, such as GloVe[1] and fastText[2]. I have chosen 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 OOV (out of vocabulary) words will occur in the source (such as misspelled words).

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

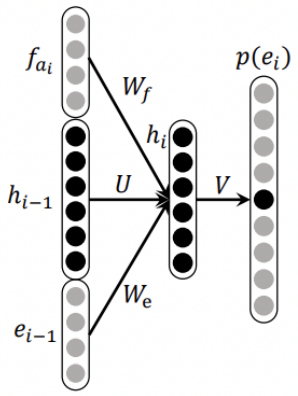

Figure 3.6: Recurrent neural network for translation confidence (Liu et al., 2014, p.1497)

fastText can easily generate word embeddings for those unseen words based on character n-grams.

Next step is to construct a PyTorch dataset from the FCE training data. For each S-T sentence pair, we obtain e_i and e_{i-1} from source S and f_{a_i} from target T. Then we define a class for RNN as follows:

```
class RecurrentNN(torch.nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RecurrentNN, self).__init__()
        self.U = torch.nn.Linear(input_size+hidden_size, hidden_size)
        self.V = torch.nn.Linear(hidden_size, output_size)
        self.softmax = torch.nn.LogSoftmax(dim=1)
```

where `input_size` should be size of word embedding multiplied by 2 (since it is e_{i-1} and f_{a_i} concatenated together). The size of word embedding is set to 50 to avoid memory issues, and the size of hidden layer is set to 100. The `torch.nn.LogSoftmax` layer would convert output to a categorical probability distribution, hence the `output_size` should be the size of vocabulary in our training data.

Training RNN

We train the RNN for 3 epochs and store the model state. Then, we load the model and run the model on the top phrase table `pt_top`. For each phrase pair from source phrase s to target phrase t, we compute

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

and save $T_{S2T}(s,t)$ to be the phrase pair embedding with RNN (`ppe_rnn`).

3.3.3 Global Features

Three features are proposed to be used as global features, namely language model score, translation model score and reordering model score. Language model scores can be ob-

tained by querying the trained 5-gram KenLM language model. Translation model scores are accessible from Moses phrase table. 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. Hence, the reordering scores are omitted and only translation model score and language model score will be used as global features in R^{2}NN decoder.

Translation model scores

The translation model scores are calculated by taking the average of the four scores given by each phrase pair in the phrase table. If `pt` is the phrase table loaded by `pandas`, we could calculate the translation model score for each phrase pair by

```
pt["confidence"] = pt["scores"].map(
                    lambda scores: round(
                    sum([float(i) for i in scores.split()])/4, 6))
```

The result will be stored in a file named `pt_top_id_confidence`.

Language model scores

We obtain the language model scores for the target phrase by querying the language model:

```
~/github/mosesdecoder/bin/query
    -n /path/to/billion-training-monolingual-10.blm.en
    < pt_top_target
    > lm_total_scores
```

where `billion-training-monolingual-10.blm.en` is the binarised trained language model, `pt_top_target` is the file containing target phrases, one per line, and the output of this command is saved in `lm_total_scores`. We can write the language model score to the phrase table and name it `pt_top_lm`.

3.3.4 R^{2}NN Decoder

Recall that in Figure 2.4, each node in R^{2}NN consists of two parts: s for the representation of the phrase (TCBPPE), and x for the recurrent input vectors (global features). Let's first define two functions, `get_ppe` and `get_rec` which would return s and x respectively for any given source phrase and target phrase.

Obtain representation vector (TCBPPE)

We obtained two files, `ppe_sparse` and `ppe_rnn`, from section 3.3.1 and 3.3.2. We combine them to be the TCBPPE (`ppe_matrix`):

```
ppe_matrix = np.zeros((200001, 21))
ppe_matrix[:, 0:20] = ppe_sparse
```

```
ppe_matrix[0:200000, 20] = ppe_rnn
```

and the phrase pair embedding for the top i^{th} phrase pair is accessed by `ppe_matrix[i]` and is of length 21. `ppe_matrix[200000]` is used for any phrase pair that is not in the top 200,000 frequent ones. We can define `get_ppe` as

```
def get_ppe(source_phrase, target_phrase):
    mappings = pt_top_lm.loc[(pt_top_lm["source"] == source_phrase)
                             & (pt_top_lm["target"] == target_phrase)]
    ...
```

and return `ppe_matrix[200000]` if `mappings` is empty (i.e. no such phrase pair found in the top phrase table), otherwise return `ppe_matrix[mapping_id]`.

Obtain recurrent vector (Global features)

The recurrent vector consists of two scores: the translation model score and the language model score. From section 3.3.3, we created `pt_top_id_confidence` for translation model scores, and `pt_top_lm` for language model scores. With the help of these files, we can define `get_rec`. Similar to `get_ppe`, we first obain mappings from source phrase to target phrase. Then we look up the scores in `pt_top_id_confidence` and `pt_top_lm` using `mapping_id`.

```
def get_rec(source_phrase, target_phrase):
    mappings = pt_top_lm.loc[(pt_top_lm["source"] == source_phrase)
                             & (pt_top_lm["target"] == target_phrase)]
    ...
    t_score = pt_id_confidence.iloc[mapping_id]["confidence"]
    l_score = pt_top_lm.iloc[mapping_id]["lm"]
    return torch.tensor([t_score, l_score], dtype=torch.float32)
```

If `mappings` is empty, both `t_score` and `l_score` will default to 0.

Building recursive neural network (RvNN)

There has not been any published implementation of R^2NN. However, since R^2NN has a tree-like structure, it is sensible to start with building a recursive neural network. A good example is given in a blog post[4], where the recursive neural network is constructed from two parts: a class `TreeNode` which builds a greedy tree from the input, and a class `NeuralGrammar` which is the neural network that processes the tree. Following the post, I built a recursive neural network (RvNN) with similar structure:

```
class RecursiveNN(torch.nn.Module):
    def __init__(self, input_size):
        super(RecursiveNN, self).__init__()
        self.W = torch.nn.Linear(input_size * 2, input_size)
        self.V = torch.nn.Linear(input_size, 1)

    def forward(self, node):
```

```
        ...

class TreeNode:
    def __init__(self, representation=None):
        self.representation = representation
        self.left = None
        self.right = None

    def greedy_tree(self, x, model):
        ...
```

The `greedy_tree` function in class `TreeNode` takes argument `x` which is a list of representation of leaf nodes. It converts each leaf node representation to a `TreeNode` first, and then combine them in a greedy way. Based on the output of `model(combined_tree)`, it chooses the combination of the nodes that gives the highest score.

To use `RecursiveNN`, we construct a tree from representation of leaf nodes x and pass the tree to the neural network:

```
rvnn = RecursiveNN(4)
x = numpy.array([[8,8,8,8], [2,2,2,2], [3,3,3,3], [7,7,7,7]])
tree_node = TreeNode()
tree = tree_node.greedy_tree(x, rvnn)
parent_node, score = rvnn(tree)
```

Building R^2NN

To build R^2NN, we need to add recurrent input vector to the RvNN. Firstly, `ppe_size` and `rec_size` should be included in the initialisation signature. The size of W should be (`ppe_size` + `rec_size`) by `ppe_size`, since the phrase pair embedding for parent node is generated from child nodes, whereas the recurrent input vector for parent node is independent of child nodes. The size of V is (`ppe_size` + `rec_size`) by 1 because output score is of size 1 and is generated from concatenation of phrase pair embedding and recurrent vector of parent node.

```
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)
        self.V = torch.nn.Linear(self.input_size, 1)
```

The `forward` function should use the `get_rec` function defined in section 3.3.4 to obtain the recurrent input vector for parent node.

```
def forward(self, node):
    ...
    parent_ppe = torch.tanh(self.W(inp_vector))
```

```
parent_rec = get_rec(parent_source, parent_target)
...
```

At the same time, when `greedy_tree` function generate leaf tree nodes, it will make use of both `get_ppe` and `get_rec` to get tree node vector representation.

Preprocessing training data

Following the R^{2}NN paper[11], forced decoding will be used to generate positive samples. We can perform forced decoding with Moses by adding the following line to Moses configuration file, `moses.ini`:

```
ConstrainedDecoding path=/path/to/gold/standard/file
```

where the gold standard file should be target sentences in FCE training dataset. Then we can pass the updated configuration file to Moses decoder and do forced decoding on source sentences in FCE training set. The phrase pairs used in forced decoding for each sentence is stored in a file named `phrases_train`.

We define a function `process_phrases_file` to remove the sentences which did not give a forced decoding result in `phrases_train`. For each of the remaining sentences, we extract phrase pairs used and a total score given by Moses decoder. Then we can construct a PyTorch dataset where the input is a list of phrase pairs used in each sentence and the expected output is the total score of that sentence.

Training R^{2}NN

We recall the loss function defined in R^{2}NN paper[11]

$$Loss(W, V, s^{[l,n]}) = \max(0, 1 - y_{oracle}^{[l,n]} + y_t^{[l,n]})$$

where $y_{oracle}^{[l,n]}$ is the score from forced decoding (the expected score), and $y_t^{[l,n]}$ is the output score from R^{2}NN for a source span $s^{[l,n]}$. The max function is used to guarantee non-negative loss. Then we save the state of our trained model in `model_state_r2nn.pth`.

3.4 Repository Overview

Figure 3.7 shows the repository structure of my project.

`errant/` contains a fork of the GitHub[3] 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[4] 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`.

[3]https://github.com/chrisjbryant/errant
[4]https://github.com/facebookresearch/fastText

`giza-pp/` contains a fork of the GitHub[5] repository `moses-smt/giza-pp`[14] and it is used in Moses SMT to obtain alignment models.

`mosesdecoder/` contains a fork of the GitHub[6] repository `moses-smt/mosesdecoder`[10]. 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. It contains:

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

`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 concatenating files.

`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.

[5]https://github.com/moses-smt/giza-pp
[6]https://github.com/moses-smt/mosesdecoder

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 3.7: 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 R2NN model is implemented *(section 3.3.3 and 3.3.4)*
- ✓ An SMT-based GEC system making use of the R2NN decoder is built *(section 3.3.4)*
- ✓ A comparison between the performances of the two systems is performed *(chapter 4)*

4.2 Evaluation Metrics

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

The $F_{0.5}$ score is computed based on precision P and recall R. Let T_P be the true positive where system **correctly** made a change, F_P be the false positive where the

[1]https://en.wikipedia.org/wiki/BLEU

system **incorrectly** made a change, F_N be the false negative where the system **missed** a change that it should have made. We can calculate precision

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

and recall

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

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

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

4.3 Testing

We will use the held out FCE test set for testing. The m2 file of the FCE test set will be processed as described in section 3.1. We obtain two sentence-aligned files: one containing all the source sentences in the test set and one containing all the target sentences (gold reference). Moses decoder and R^{2}NN decoder will process the source file, and each system will produce a translation file which contains the translations of the source sentences. We can then use ERRANT to evaluate the translation file against the target file.

4.3.1 Moses Decoder

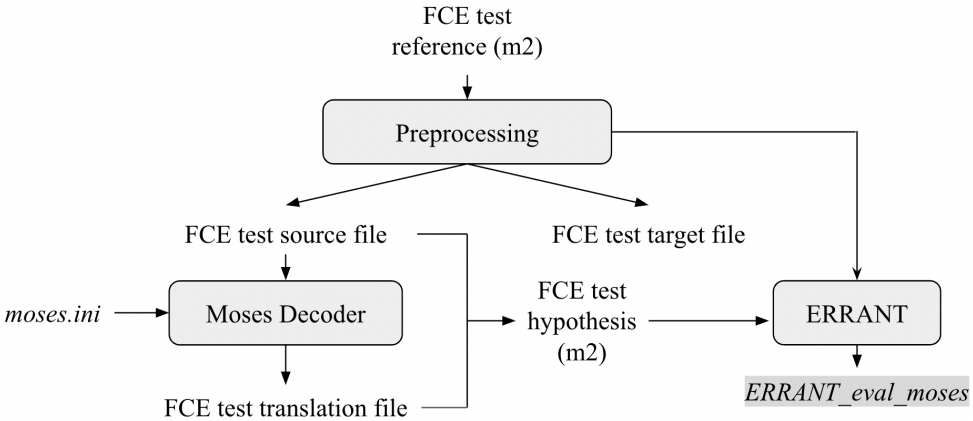

Figure 4.1: Testing pipeline of Moses SMT

Before testing Moses decoder, we should tune the parameters using a small amount of parallel data which is different from the training data. We will use the FCE development set for this purpose. The tuning process would optimize the weights for each feature as specified in `moses.ini`. Then we can use Moses decoder to translate the test set:

```
~/github/mosesdecoder/bin/moses
    -f /path/to/moses.ini
    < /path/to/fce/test/source
    > fce.test.moses.translated
```

where `fce.test.moses.translated` is the translation file produced by Moses decoder. We use ERRANT to annotate the source file with the translation file to obtain the hypothesis m2 file. After that, we can compute evaluation scores using ERRANT scorer and the result will be saved to `ERRANT_eval_moses`.

4.3.2 R^{2}NN Decoder

To use the R^{2}NN decoder, firstly we need to split the sentence into phrases. Liu et al. did not specify the method they used for phrase splitting in the paper[11], so I came up with a greedy 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. An example is given below.

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

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

The next step is to obtain phrase pairs based on the sentence span. For each phrase in the span, we have a list of possible phrase pairs. We use our R^{2}NN model to compute a score for each phrase pair and only keep the n-best phrase pairs. Then we recursively repeat the process and build a tree based on the phrase pairs. Every tree node has two attributes: `node.source` which gives the source phrase of this node and `node.target` which gives the target phrase. The translation for a sentence can be obtained by accessing the root node's `target` attribute of the tree. The resulting translation file will be saved to `ERRANT_eval_r2nn`.

Phrase pairs
(I hope that → I hope that), (I hope that → I hope), . . .

However, when it comes to testing, I discover that the search space is too big for my laptop to process. To process a sentence with k phrases while keeping 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, I decided not to keep the n-best phrase pairs but to keep the phrase pair that gives the highest score.

4.4 Results

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

We observe that the $F_{0.5}$ score of R^{2}NN is much lower than Moses baseline system. Even though R^{2}NN 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 low. We also observe that R^{2}NN gives more true positive but at the same time much

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
581	8342	3968	0.0651	0.1277	0.0722

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

more false positive (FP) than Moses baseline. It suggests that R^2NN tend to make changes to the source sentence even when it should not have.

A closer look at the translation file produced by R^2NN reveals that R^2NN system does not give translation for unseen phrases or phrases that are not in the top phrase table. Consider the following example:

Source sentence
Let 's congratulate Santos Dummont !

Gold reference
Let 's congratulate Santos Dummont !

Moses hypothesis
Let 's congratulate Santos Dummont !

R^2NN hypothesis
Let 's congratulate !

Since the phrase *Santos Dummont* did not appear in the top phrase table, no translation is given for this phrase in R^2NN hypothesis. This would contribute to the false positive of R^2NN system. We could mitigate this problem by setting the default phrase pair as *(SOURCE → SOURCE)* rather than *(SOURCE →)*.

Another example of R^2NN made edits to the source sentence where Moses did not:

Source sentence
I hope you will enjoy your stay in my country .

Gold reference
I hope you will enjoy your stay in my country .

Moses hypothesis
I hope you will enjoy your stay in my country .

R^2NN hypothesis
*I hope you will enjoy your **live** in my country .*

The phrases used in R^2NN for the source sentence are

I hope you will /enjoy your /stay in /my country .

And the phrase pairs selected are

(I hope you will → I hope you will) / (enjoy your → enjoy your) / (stay in → live in) / (my country . → my country .)

The phrase pair *(stay in → live in)* is selected because it has the highest score of all the pairs with *stay in* being the source phrase. Other phrase pairs, including *(stay in → stay in)* in gold reference, were not considered by R^2NN for upper tree combination. A possible solution to this is to keep n-best phrase pairs at the cost of expanding the search space.

4.5 Extension

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

1. Before a sentence is passed to the R^2NN, it has to be split into phrases. The way I split the sentence is described in section 4.3.2, but other algorithms to split the sentence may be used to obtain different initial phrases and hence different initial tree leaves.
2. Recall in section 4.3.2, I only kept the best phrase pair for upper tree combination to avoid having a huge search space. It is more sensible to keep 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^2NN SMT system is small. We could use other training data available on BEA 2019[1] in addition to the FCE dataset.
4. Inevitably, unseen word will exist no matter how big our training set is, especially for the task of GEC where the input data could contain misspelled words. We have seen that R^2NN does not give translation for unseen words (as discussed in section 4.4). A slightly better way to deal with it is to keep the original word/phrase, but this may ignore spelling errors. Alternatively we could expand the size of phrase tables at the cost of searching time.

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 Moses baseline SMT, and an R^2NN SMT. For Moses SMT, I started with building a working SMT system and trained it with FCE data. For R^2NN SMT, I constructed phrase pair embedding (TCBPPE) using two neural networks, a feedforward neural network (FNN) and a recurrent neural network (RNN). I implemented the R^2NN model from scratch. Then I built the R^2NN SMT by integrating TCBPPE and global features such as language model scores into the R^2NN decoder. Finally, a comparison was performed on the performance of Moses SMT-based GEC and R^2NN SMT-based GEC. R^2NN achieved better recall but due to its low precision, overall it gives 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 PyTorch library would certainly be helpful in my future career.

One important lesson I took from this project is that there will always be unforeseen difficulty. 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 do not 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^2NN) because it was far more challenging than the theory in the paper[11] looked like.

5.3 Future Work

As discussed in section 4.5, one of the possible extension is to use a different method to split the sentence into phrases. The initial phrases are important because they form the initial leaf nodes of the R^2NN tree. It is also possible to integrate the phrase splitting stage into the R^2NN structure instead of being a separate pre-processing stage.

Another extension would be to increase the number of best translation candidates kept for upper tree combination. However, care must be taken because the GEC system could be quite inefficient due to the large search space.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

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

[16] 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.

[17] 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

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.