Jiaxin Wang

A Recursive Recurrent Neural Network Decoder for Grammatical Error Correction

Computer Science Tripos – Part II

Emmanuel College

April 22, 2022

Proforma

Candidate Number:	-
Project Title:	A Recursive Recurrent Neural Network Decoder for Grammatical Error Correction
Examination:	Computer Science Tripos – Part II, 2022
Word Count:	5040 [1]
Code line count:	1127
Project Originator:	-
Supervisor:	Dr Zheng Yuan, Dr Christopher Bryant

Original Aims of the Project

-

Work Completed

-

Special Difficulties

-

[1] This word count was computed by `detex diss.tex | tr -cd '0-9A-Za-z \n' | wc -w`

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]

Contents

List of Figures

Acknowledgements

-

Todo list

Chapter 1

Introduction

This project concerns my implementation of a *recursive recurrent neural network* model (R^2NN) proposed by Liu et al. (2014) [8] This is to be integrated in a statistical machine translation (SMT) system to attempt the task of grammatical error correction.

More details

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 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)[8].

Technical motivations: why R2NN? Academic motivation: publishing code for R2NN as no code available

1.2 Problem Overview

The task of Grammatical Error Correction can be seen as a machine translation process, where the input is a text which may contain grammatical errors, and the output is an error-free text. This project uses a statistical machine translation (SMT) approach to solve GEC.

A recursive recurrent neural network (R^2NN) was proposed by Liu et al. (2014) [8] for SMT. The goal of this project is to implement the proposed model to be used for GEC. The model should aim to correct all types of errors, namely grammatical, lexical, and orthographical errors. Its performance will be evaluated against a baseline SMT system, Moses[7].

1.3 Related Work

-

Chapter 2

Preparation

2.1 Starting Point

This project is based on the idea presented in the paper *A Recursive Recurrent Neural Network for Statistical Machine Translation* [8]. An R^2NN model was proposed, but the implementation details are not given in the paper.

The Part IB Computer Science Tripos course Artificial Intelligence[1] 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. I have found it useful to follow the coding examples on PyTorch[2] website.

An example SMT system is available on Moses[3] website. It provides a detailed tutorial on how to install Moses, how to prepare corpus and how to train the SMT.

2.2 Theory

2.2.1 SMT system

A typical SMT system consists of four main components: The language model (LM), the translation model (TM), the reordering model and the decoder[10]. 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 the task of GEC, this would be the most probable correction to the original erroneous sentence.

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

2.2.2 Moses SMT

Moses baseline system uses KenLM as the language model. For the translation model, it uses GIZA++ to obtain a word alignment model. The model should be trained to produce a phrase table and associated scores. Reordering tables are also created during this process. Eventually, Moses decoder will find the best translation candidate given input based on the scores it calculated in previous stages.

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.

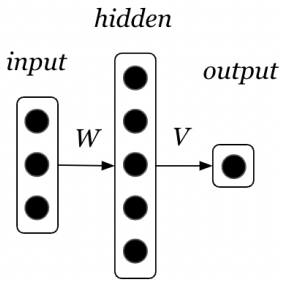

Figure 2.1: Example of a feedforward neural network

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, 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)

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.

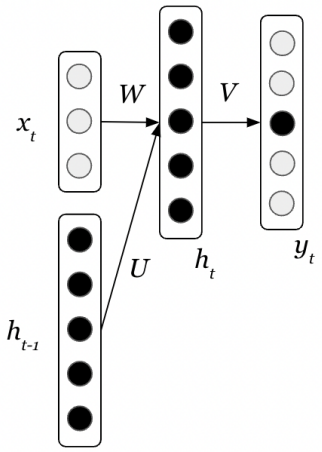

Figure 2.2: Example of a recurrent neural network (RNN)

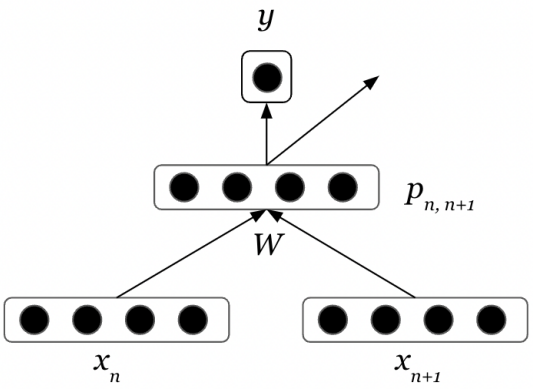

Figure 2.3: Example of a recursive neural network (RvNN)

2.2.6 Recursive Recurrent Neural Network (R^2NN)

The R^2NN proposed by Liu et al.[8] 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

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

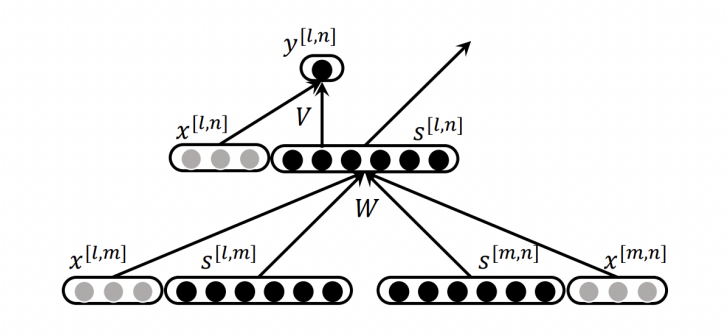

Figure 2.4: Recursive recurrent neural network (Liu et al., 2014, p.1494)

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^{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

Following the R^{2}NN paper by Liu et al.[8],

- 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`[4] 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`[5] library for this purpose.

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

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[7] 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[8].

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.

[4]https://pytorch.org/
[5]https://numpy.org/
[6]https://pandas.pydata.org/
[7]https://www.cl.cam.ac.uk/research/nl/bea2019st/#data
[8]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 a R^{2}NN SMT system. Since the main purpose of this project is to compare the performance of our 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[8] 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.

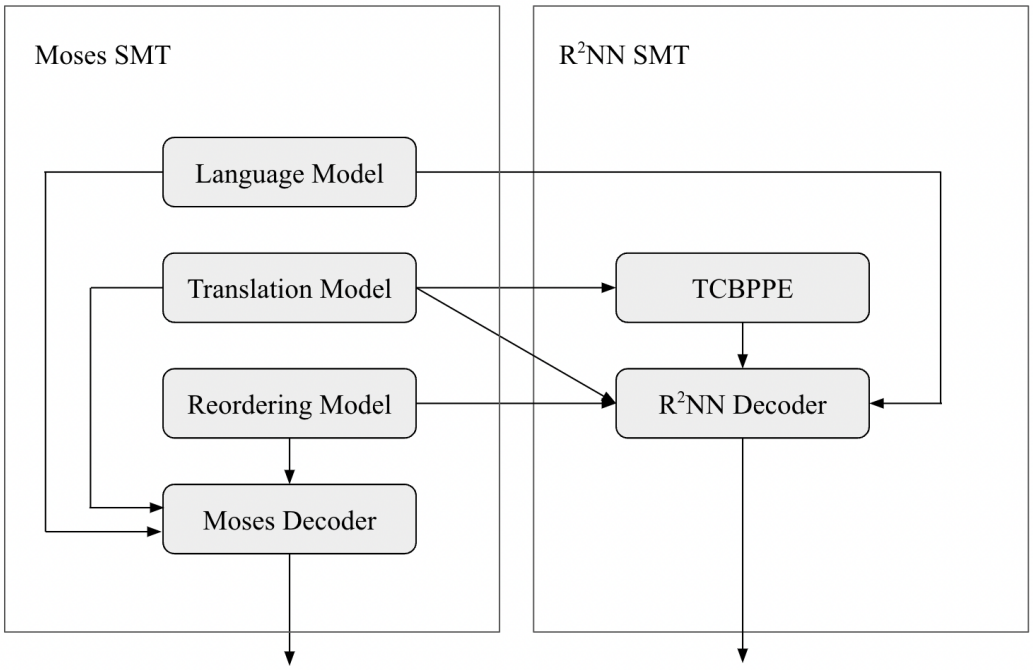

Figure 3.1: Overview of Moses SMT and R^{2}NN SMT

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

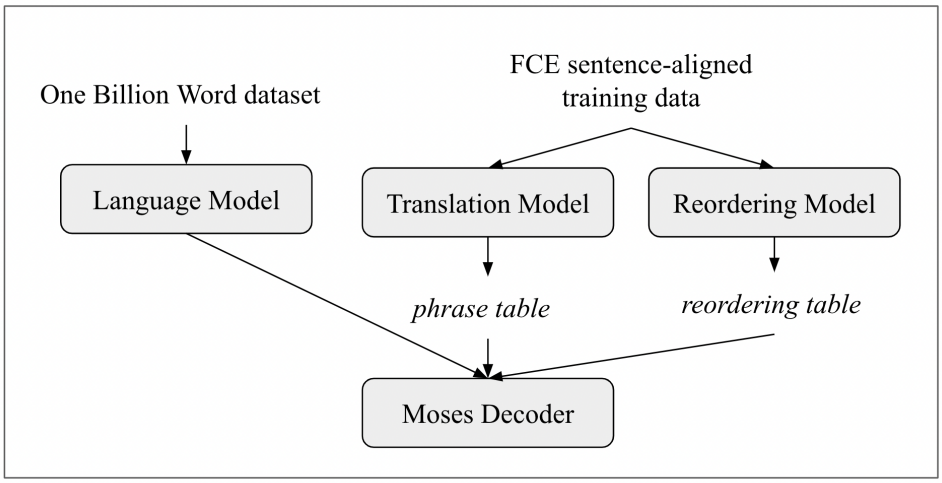

Figure 3.2: Overview of Moses SMT (training)

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 a R²NN decoder.

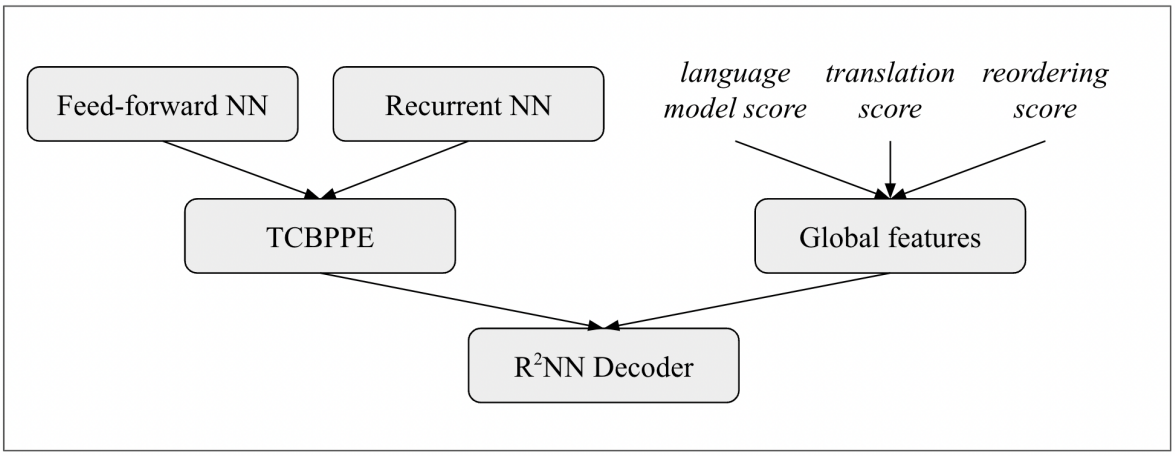

Figure 3.3: Overview of R^{2}NN SMT (training)

TCBPPE

A phrase pair embedding is a vector representation which encodes the meaning of a phrase pair. Recall that our 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[8], the size of the hidden layer is set to 20. The size of the input layer is

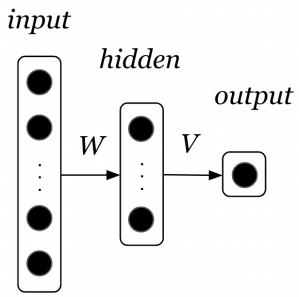

Figure 3.4: Structure of one-hidden-layer feedforward neural network

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:

[Source sentence] `What she would do ?`

[Target sentence] `What would she do ?`

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

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`).

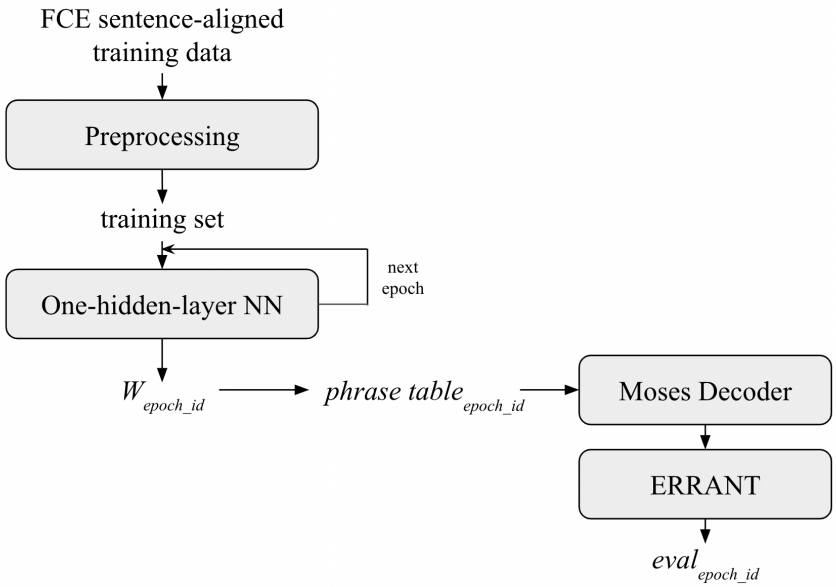

Figure 3.5: Training pipeline of one-hidden-layer feedforward neural network

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

Table 3.3: ERRANT output (5 epochs)

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

Table 3.4: ERRANT output (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

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

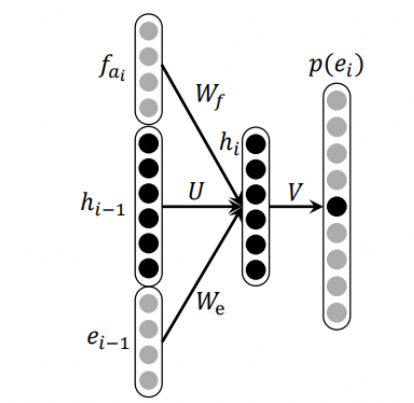

Figure 3.6: Recurrent neural network for translation confidence (Liu et al., 2014, p.1497)

that OOV (out of vocabulary) 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 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 obtained 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 R²NN 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 R²NN Decoder

Recall that in Figure 2.4, each node in R²NN 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[8], 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[8]

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

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).

[3]https://github.com/chrisjbryant/errant

`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`.

`giza-pp/` contains a fork of the GitHub[5] repository `moses-smt/giza-pp`[9] and it is used in Moses SMT to obtain alignment models.

`mosesdecoder/` contains a fork of the GitHub[6] repository `moses-smt/mosesdecoder`[7]. 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.

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

`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 3.7: Repository Overview

Chapter 4

Evaluation

Chapter 5

Conclusion

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] 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.

[8] 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.

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

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

Appendix A

Project Proposal