Insights into Seq2seq Models for Text Correction

Kevin Xie (UID: 304402775)

Siyang Sun (UID: 504400191)

Contents

1 Introduction

Spellcheckers have been around since the dawn of the Information Age. This idea of an automated process to correct human spelling errors came naturally with the budding use of computers. The original spellcheckers simply accessed a list of accepted words, and checked if each word in the checked passage was a member of that list. The famous Microsoft Word spellchecker still used a predetermined dictionary, but allowed users to add their own custom words. However, as typing on computers, mobile phones, and other electronic devices became more ubiquitous, spellcheckers had to get smarter. They needed to keep up with each user's individual typing habits and frequency of word use, and take into account the layout of the keyboard. From here, we have autocorrection software. These require machine learning and a much smarter algorithm than simply checking a static list of allowed words. In this paper, we will be building our own spellchecker using recurrent neural networks and natural language processing. We decided to create a smarter traditional spellchecker to correct errors rather than an autocorrection software that can figure out what we meant to text to our friends. This means we will train our model with correctly spelled words and correctly arranged sentences, in order to check for proper English.

In this paper, we will first discuss our model and methods, and tackle the mathematics behind recurrent neural networks in the sequence-to-sequence model. Then, we will dive deeper into the algorithm we used to generate the model and build the spellchecker. After going over the results, we will discuss difficulties, improvements, and further investigation.

2 Seq2seq Model

To obtain our model, we used a grid search to find the optimal hyperparameters and overall architecture. Hyperparameters can include constraints, weights, or training rates. This entails an exhaustive search of a predetermined subset in the hyperparameter space to find the "best" set of hyperparameters.

We will be using recurrent neural networks for our spellchecker. Specifically, we will be using a sequence-to-sequence model to do this. A sequence-to-sequence model's two bidirectional RNNs consists of an encoder and a decoder, often used for translating between two languages. In this model, the encoder reads a sentence and converts it into a vector. The decoder then defines a probability of the translation based on that vector. It accomplishes this by expressing the joint probability as a product of conditional probabilities. The encoder and decoder are trained together in order to maximize the probability of an accurate translation, or in this case, spelling correction.

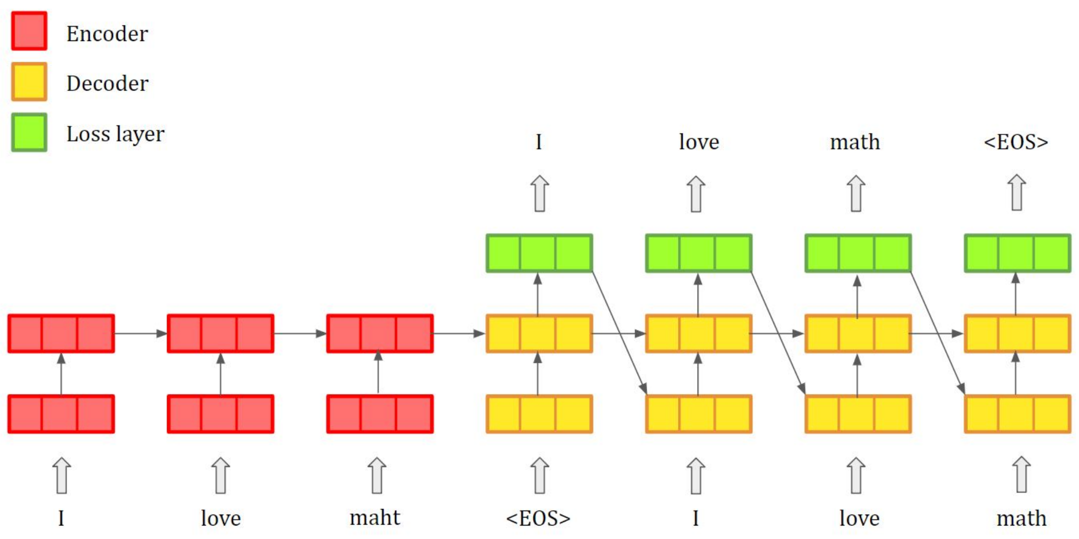


Figure 1: Basic diagram depicting seq2seq model

In detail, the encoder generates a high-dimensional feature vector c from the sequence of input vectors $\mathbf{x}$, which contains n vectors. This process was detailed by Bahdanau *et al.* at ICLR 2015 [1]. Let h_t be a hidden state at time t which depends on x_t and the previous state, h_{t-1}. The use of hidden states based on the time t transforms this space-problem into a time-problem. f is some nonlinear function such that $h_t = f(x_t, h_{t-1})$. Then c is generated by some nonlinear function q of all the h_ts.

The decoder predicts the next word y_t based on the words leading up to it and the context vector c. The joint probability of the entire segment of words y is determined by the product of the conditional probabilities.

The conditional probabilities are also mapped by some nonlinear function g. With s_t as a hidden state for the decoder at time t, we can rewrite each conditional probability as

$$p(y_i|y_1, \ldots, y_{i-1}, \mathbf{x}) = g(y_{i-1}, s_i, c_i).$$

We will then use these s and the previously calculated h to get our context vector. The nonlinear function we will use is a weighted sum, where the weights are similar to a softmax function.

We will use an alignment model a as a score of how good of a match the inputs at time j are to the outputs at time i within the encoder and decoder system. a is also trained with the rest of the system as a component neural network.

$$e_{ij} = a(s_{i-1}, h_j)$$

This is also known as an attention mechanism, which allows the encoder-decoder system to perform more efficiently in comparison to a pure memory mechanism. In this mechanism, the decoder can pay attention to different parts of the sentence at different times to get an output. This way, the encoder no longer has to convert every piece of information from the input sentence into a fixed-length vector [2].

These e_{ij} are used in a softmax function to squash the values into the range (0,1) and determine the weights. In this equation, n represents the length of the sequence of vectors $\mathbf{x}$. The following expression determines how important h_j is in determining the following state, s_i, from the previous state s_{i-1}.

$$a_{ij} = \frac{e^{e_{ij}}}{\sum_{k=1}^{n} e^{e_{ik}}}$$

Now, we have the information we need to determine the context vector c_i, which allows us to backpropagate through the cost function. Once we are finished, the decoder will be able to predict the next word.

$$c_i = \sum_{j=1}^{n} a_{ij} h_j$$

3 Algorithm

Our algorithm to solve the seq2seq model relies on Tensorflow for two functions: build_model() and train_model(). Before we interact with these functions we must first cultivate our data. We found 22 books in ASCII and cleaned them of any open source terms of use paragraphs which usually precede the actual book. Then we went line by line and removed any occurrence of "~<>{}[]*" which are garbage values and should not be in the book. We also forced two or more spaces to one as this sometimes occurs during book scanning. Finally we separately the books into sentences.

Before we fed these cleaned sentences into our model we made sure to remove lengths that were too long as this would make our training time increase severely. Looking at Table 1, we see that a majority of sentences fall below 111 characters long. We set this as our max length. For the min length we set it as 9 because most simple sentences seemed to hit at least that threshold. One could set the min length anywhere between 8-15 and achieve similar results. We then simulated errors in these sentences using a function called error_producer() at a 12% rate by iterating through each letter in the sentence and assigning a 3% chance that the letter will be swapped with it's neighbor, 3% chance the letter will be replaced by another letter, 3% a letter will be added between said letter and its neighbor, and a 3% chance the letter will be deleted.

3.1 build_model.py

Now we can build our model. One important thing to note, the seq2seqmodel() function has the encoding and decoding RNN hidden inside its implementation. The implementation itself is rather boilerplate and the exact code was an amalgam of templates found on the Tensorflow documentation website and a mixture of different open source codes online. We took the most inspiration from a repository developed by David Currie [3].

```
def build_model():
    # Create model input templates
    [array of input templates] = create_model_templates()
    # Construct decoding logits from seq2seqmodel;
    # logits map tensors to the softmax probability function
    # tensors are simply n-dimensional lists
    logits = seq2seqmodel([array of input templates],
                                  [max_length and other parameters])
    # Construct tensors for the decoding logits
```

mean	149.449326
std	158.554753
min	1.000000
25%	59.000000
50%	111.000000
75%	190.000000
max	8987.000000

Table 1: Data set statistics

```
    tf.summary(predictions(logits))
    # Construct the weights using the weighted cross-entropy error function
    tf.summary(cost(weights))
    # merge Tensorflow summary of tensors, and weights
    tf.summary.merge_all()
    # return graph created by nodes like [prediction, cost...]
    return [*graph]
```

3.2 train_model.py

To train our model we must feed it in batches. This is why we originally set our sentence length between 11 and 111 characters long. Seq2seq models are most efficient when they have a set length for text translation i.e. the batch size. The batch size for our final test run was 120 characters. Training a model is done in epochs which is an iteration through the entire data set. We set our max_epoch constant to 10 but it never reached past 3.

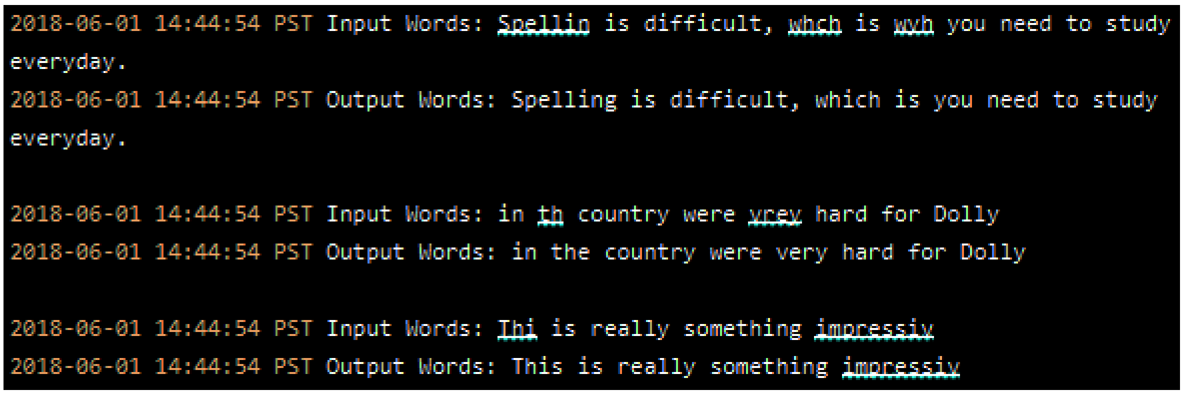
```
def train_model():
    stop_increment = 3
    min_increment = 0
    test_batch_amount = 100 or a high enough amount to check
                        that we have reached a local min
    for each epoch:
        for each batch:
            calculate the batch loss/error by running the batch
            using Tensorflow with nodes contained in the graph
            returned by build_model()
            if we have iterated through a test_batch_amount
               of batches:
                if we have reached a new minimum:
                    min_increment = 0
                else if there is no improvement:
                    min_increment += 1
                    if min_increment == stop_increment:
                        break
        if min_increment == stop_increment:
            save the model in disk as a checkpoint
            return
```

In summary, the algorithm works to minimize an error function, in this case the cross-entropy error function, by recurrently iterating through the batches and recalculating the weight vectors until they are optimized for the data set. The min is not guaranteed to be a global min since we stop after 3 iterations through the batches.

4 Results

We fed the model 22 books five times for a total of 110 books. Duplicating the data was not an issue as error_producer() is entirely random. We ran the setup on a Tesla K80 GPU with 4992 NVIDIA CUDA cores. The algorithm ran for about 80 minutes. To establish a passing grade for our model, the benchmark we produced was one spelling error correction on a majority of the data with no manipulations to correct words.

The model passed without a problem and even reached two corrections in 66% of our test data. Here is a select sample from our test:



```
2018-06-01 14:44:54 PST Input Words: Spellin is difficult, whch is wyh you need to study
everyday.
2018-06-01 14:44:54 PST Output Words: Spelling is difficult, which is you need to study
everyday.

2018-06-01 14:44:54 PST Input Words: in th country were vrey hard for Dolly
2018-06-01 14:44:54 PST Output Words: in the country were very hard for Dolly

2018-06-01 14:44:54 PST Input Words: Thi is really something impressiv
2018-06-01 14:44:54 PST Output Words: This is really something impressiv
```

Figure 2: Selected sample from testing

We are unsure of the lower bound for a two correction passing grade. However, we discovered that the lower bound for three corrections is far above 80 minutes as we could achieve 3 corrections in only a few sentences among tens of thousands of test sentences.

5 Discussion

A salient purpose of our model was to demonstrate that one could achieve objectively beneficial results with limited training times. That being said, extensive work was done and can be done to improve the model further. Successful text correction with neural networks has only been around for about 5 years, and it's changing day by day. The prevailing approaches to NLP as a whole are already being usurped.

5.1 Difficulties

Initially we did not know how long to train the method and it gave really poor results. We began with a training time of about 15 minutes. The results were lackluster at best.

A simple sentence like “I drakn wine.” would turn into “Wine wine wine.” Increasing the time to 30 minutes and corrections would begin to occur, but only at the expense of the correct words: “waht is going on?” would become “what is going going?”. After hours of testing, we gathered that the minimally sufficient training time for a passing grade was around 50 minutes.

5.2 Improvements

There are a litany of improvements we could make by optimizing model inputs. However, by simply increasing the training time, our results would improve the most. One particular example is a text corrector developed by Alex Paino which was trained for 2.5 hours and was able to correct they’re, their, and there in a sentence [5]. Although Paino’s model was optimized for grammar, its results easily surpass the capabilities of our model for text translation. However, a interesting and facile optimization to our model would be to combine it with Paino, perhaps forming an all purpose English corrector.

5.3 Better Models

Facebook’s artificial intelligence research (FAIR) team has uncovered a revolutionary way to approach text translation in machine learning models. The basis of this approach is applying a convolution neural network (CNN) instead of a recurrent neural network (RNN) to encode a given data set. The primary benefit of CNN architecture is that it allows a GPU to simultaneously compute all elements of a weight vector rather than sequentially compute them like a RNN does. What this amounts to is a ninefold speed increase compared to the speed of the traditional RNN. Thus, despite the fact that we tested our model on a high grade GPU, our RNN approach for text correction cannot fully utilize the capabilities of the hardware.

Despite the speed advantage, researchers have not been able to mimic the performance of a RNN for text translation until FAIR team’s breakthrough. The FAIRs teams text translation approach has a two attributes that allow it to perform well: a multi-hop attention mechanism in the decoding layer and gating [4]. Gating is an already implemented mechanism in RNNs that allows hidden units which store neural network information to pass on specific information to the next hidden unit. For example, gating helps us determine there, their, and they’re in a sentence because the previous words represented by the hidden units will clue us in. On the other hand, multi-hop attention mechanism is a boosted version of the attention mechanism we have discussed, which allows the decoder to look at multiple components of the sentence multiple times. In theory, this allows the decoder CNN to better capture complex relationships between sentence components.

References

[1] Bahdanau, Dzmitry, et al. "Neural Machine Translation by Jointly Learning to Align and Translate." *arXiv:*1409.0473[cs.CL].

[2] Britz, Denny. "Attention and Memory in Deep Learning and NLP." *WildML*, http://www.wildml.com/2016/01/attention-and-memory-in-deep-learning-and-nlp/.

[3] Currie, David, Spell-Checker, 2017, *Github repository* https://github.com/Currie32/Spell-Checker.

[4] Gehring, Jonas, et al. "A Novel Approach to Neural Machine Translation." *Facebook Code*, Facebook, 9 May 2017, code.facebook.com/posts/1978007565818999/a-novel-approach-to-neural-machine-translation/?utm_campaign=Artificial%2BIntelligence%2Band%2BDeep%2BLearning%2BWeekly&utm_medium=email&utm_source=Artificial_Intelligence_and_Deep_Learning_Weekly_13.

[5] Team, FloydHub. "Deep Text Corrector." *FloydHub Documentation*, docs.floydhub.com/examples/deep_corrector/.