Machine Translation with Recurrent Neural Networks

Henry Maguire

October 13, 2017

Abstract

Here I build, train and analyse a Neural Machine Translator using Sequence-to-Sequence modelling. I use unidirectional encoder-decoder architecture. The aim is to see whether a single RNN layer in both the encoder and decoder, with limited computational and time resources, can be made to outperform a dictionary based method on a French-to-English dataset.

1 Introduction

Over the years of development in machine learning, performance in many tasks has improved substantially. A clear example is speech recognition, where Google's speech recognition system is now "95% accurate" in the English language and has "improved 20%" since 2013 alone. In many areas the field of NLP still lags behind, with chatbots still being restricted to very closed domain conversations and remain very unconvincing when passing for humans, despite many recent technological improvements. However, the application of machine learning in language translation has been very successful, with the Google neural MT system [reducing errors][9] by 60% compared with phrase-based systems. Statistical and rule-based machine translation has been used since World War Two, which require a large amount of domain knowledge and human input, hard-coding of grammar into large binary trees, etc. Many developments have been made in using Bayesian inference, maximum likelihood[10] of translations, log-linear models [11] and other sophisticated SMT techniques - the accuracy of these systems has slowly improved over time with computational power. Other approaches have involved translating both source and target phrases into a common intermediate language, again using pre-developed knowledge of the structure of each language. Cutting edge solutions often use a combination of established grammatical models and deep neural networks[1], since deep neural networks are able to approximate arbitrary functions, which makes them useful in supervised learning applications. The backpropagation algorithm allows correlations to be learned from data by updating the weights of a neural network iteratively, based on how far from the ideal scenario an approximation/prediction was. Neural networks can also take advantage of the symmetry of a problem. Recurrent Neural Network[2] share parameters across "time steps" - so that each time we see the word "cat" we don't have to re-learn the weights in the network. This also allows sequential data to be handled, with a varying number of inputs mapping to a fixed number of outputs, so it generalises very well to natural language processing tasks. This means that all you need to develop a translation system is a load of parallel text in both the source and target language, as well as enough computing power. It turns out that both requirements have recently become a lot easier to satisfy, with free access to large parallel corpora and relatively cheap GPUs.

The Google MT system[3] deploys two Deep Recurrent Neural Networks which are connected end-to-end, one which generates fixed size vector representations for variable length input phrases in the source language (the encoder) and another which takes these vectors and generates variable length sequence representations of them in the target language (the decoder). This is known as Encoder-Decoder architecture[4] and will be outlined later on. The final layer then projects a probability distribution over predicted words in the target language at each time step. Simple RNN cells suffer from a number of pathologies, such as vanishing gradients[5], which make it difficult for them to keep track of long-term language dependencies. To get around this the Google NMT system uses Long Short-Term cells, which are gated units that can adaptively learn to forget and remember different weights depending on the context of a current word - this theoretically allows long-term dependencies to be learned. The effectiveness of the system increases greatly with the depth of the network so they also use 8 decoder and 8 encoder LTSM layers, which in turn expands the hardware requirements substantially. Another breakthrough of the Google system is in the use of attention mechanisms[6], whereby the entire output of the encoder layer is made visible to the (first) decoder layer throughout the computation. This allows the decoder to keep track of the entire context of the word at all timesteps, using important summary representations of the whole phrase. Another advanced technique is in the handling of rare words, where they split up individual words into sub units, for example ‘"going"‘ becomes ‘"go", "ing"‘ etc - this would require some kind of auxillary learning algorithm.

1.1 Problem Statement

The problem of machine translation requires decoding the *meaning* of phrases and then encoding this meaning into an entirely different representation/script. The meaning of phrases can be embedded in a distributed vector representation, much like the meaning of words can be with methods like [word2vec](), as a kind of conditional probability distribution. In theory, RNNs can learn the types of conditional probability distributions which are necessary for this task e.g.

Given the phrase "Mr Smith was walking" what is the probability of each word in the vocabulary appearing next?

Effectively, variable length sequences can be mapped to fixed length vectors, enabling sequences of words to be generated probabilistically either one word at a time or via beam search etc (such as in Andrej Karpathy's blog[2] which performs this task at the character level to generate almost compilable Linux code)! The probability distributions over target words increases with vocabilary size, so the vectors can become very sparse and the number of model parameters often increases exponentially - this is another huge problem in machine translation and is a reason why closed domain MT systems (say, political, military or legal documents) have long outperformed those used in open domains (arbitrary conversation).

Translation also cannot happen on a word-by-word basis, due to idiomatic phrases and long term grammatical dependencies, for example:

Mr Smith was walking to the shop when a bird flew into **his** head.

Here the words “Mr” and “his” share the same gender, but this dependency is *hidden* for 11 words, this makes it very difficult for many types of machine translation systems. Simple RNNs, whose activations are always less than one due to the output range of the non-linearities applied (tanh, sigmoid, etc), lead to very small gradients over just a few time steps as the activations diminish further and further when multiplied by weights. This means that although not theoretically limited in their memory, simple RNNs cannot learn these dependencies in practice and so other architectures (which I will go into) or regularisation techniques (say imposing a threshold on the norm of the gradient) need to be used.

Another enormous problem is that even if we have a fixed vector representation of a phrase or word in our source language, this vector may live in a completely different vector space to the target phrase and so we have no way to measure similarity directly. This is where the magic of encoder-decoder architecture comes in, which I will discuss below.

1.2 The BLEU Metric

Evaluation Metrics

There are two points at which the accuracy of the model will be assessed. The first is during training when the decoder layer predicts the next word in the sequence for each timestep - for this I will use some standard loss such as cross-entropy (with softmax) to find the score (probability) vector. The second is during testing, to see the efficacy of the translation machine. For this I could use the BLEU metric to measure the accuracy of both the word-for-word translation and the Deep Learning approach. The BLEU metric is the most popular used in the field of machine translation since it is known to [correlate well with human judgment][7].

The BLEU scoring method uses N-gram frequency in the machine output and human translated phrase, whereby each sequence of words up to length ‘n‘ in the target and output translation are recorded. The BLEU score is based on the ratio of: the number of n-grams in the predicted phrase which appear in the target phrase, m and the total number of words in the predicted phrase, w_t. A score of $P = m/w_t = 1$ is a perfect score. This standalone metric would value translations which are dominated by just a single n-gram which is found in the target phrase. The value in the numerator is therefore augmented such that the number of times an n-gram in the predicted phrase can be counted is limited to the number of times it appears in the target phrase. For example, using a 2-gram BLEU score:

$$\text{Predicted phrase: “}\textbf{The dog the dog the dog}\text{”} \tag{1}$$

and

$$\text{Target phrase: “}\textbf{The dog}\text{ sat on the log”} \tag{2}$$

In the predicted phrase there are five 2-grams: **the dog** appears three times and dog the appears twice, of these only **the dog** appears in the translation and it appears once. Without the augmentation, the 2-gram score would be $P = 3/5$ but with the augmentation the score would be $P = 1/5$. Unfortunately, N is a hyperparameter, but fortunately research has shown that $n = 4$ seems to agree most strongly with ”monolingual human judgements”. Also, small values of n seem to measure how much information was retained in the translation whereas larger values are a measure of the readability/fluency. Given this latter definition, I expect that the benchmark model outlined above may score a reasonable unigram BLEU score but will probably not score very highly with respect to larger N-grams.

1.3 Benchmark Model

A Benchmark Model could be to just translate word for word the source sentence using dictionary definitions, via some online translation software API[8]. This may be cheating since we would have to reference some third party dictionary definitions, but it would provide a simple baseline with which we can compare. This method would no doubt miss all of the idiomatic phrases which may appear in the original corpus and is likely to get the ordering of adjectives and gendered articles confused. An example of a likely benchmark error is that the phrase “the black cat” in English becomes “le chat noir” - which uses the masculine definite pronoun (which doesn’t exist in English) and has the ordering of “chat” and “noir” reversed from the source phrase.

1.4 The Datasets

2 Neural Machine Translation

Now I will lay out some of the theoretical background necessary to understand how neural networks are used to develop language models and Encoder-Decoder Architecture

2.1 Word Embeddings

Firstly, how does a computer determine what a word *means*? If we one-hot encode our vocabulary, we are essentially defining a basis in a large vector space. We can find alternative, distributed representations - dense vectors of continuous values - of each word in our vocabulary of words by training a model to perform simple tasks. For example in Continuous-Bag-of-Words models (Skip-Gram models), such as those used in Word2Vec, a single-layer neural network is trained to predict each word in a text given a set of context words surrounding it (vice versa), a binary classification task. A by-product of training the model to carry out this process is that the columns of the hidden layer in the network are embedded vectors representing each word in the vocabulary, indexed by the row number at which its one-hot vector is “hot”. Strangely, these vectors can capture some syntactical and semantic properties of the vocabulary, with the various components determining things like the gender of a noun[14]. Even more strangely, the semantic relations between embedded words can be preserved through binary vector operations, for example the sum of embeddings:

$$\text{“Queen”-“King”+“Prince”=“Princess”}. \tag{3}$$

The expressiveness of these vectors will depend on the number of neurons in your hidden layer and the amount of data you train the model on.

This is a fun trick, but is it useful for Language modelling? It is apparent that if we could only work out a transformation which maps from the vector space of one vocabulary to that of another language then we may have generated some kind of dictionary between the two. In principle, this type of transformation can be made between any forms of data, such as mapping images to words for use in image classification[13]. However, learning a language is far more than just memorising a dictionary. A proper machine translation system will need to encode the meaning of entire phrases in one language - of which there are many more combinations that exist than words in a vocabulary - and then express this meaning in a different language, all the while navigating the complex rules and interdependencies between the various words.

2.2 Recurrent Neural Networks

As outlined in the Problem Statement, language translation involves keeping track of long-term dependencies in order to properly treat genders, tenses and other grammatical structures. Treating these relationships in time requires an efficient way to process variable-length sequences of data, which is difficult/impossible to do with standard Neural Networks. One way of dealing with this is by including a term in the calculation for the hidden state of each cell which refers to the hidden weights of the cell at the previous "time-step"

$$\mathbf{h}_t = \begin{cases} f(W_{xh}\mathbf{x}_t + W_{hh}\mathbf{h}_{t-1} + \mathbf{b}_t) & \text{if } t \geq 1, \\ 0 & \text{otherwise}, \end{cases} \tag{4}$$

where f can be any suitable non-linear activation function, but is almost always a tanh for standard RNNs. This means that information can be passed forwards in time, so that the temporal correlations of words can be learned by the network. Only the previous word is fed in, since information from further in the past should be contained within the hidden layer h_{t-1}.

During training, the recurrent network is unrolled in time and the errors are back-propagated like a normal feed-forward neural network. Each activation is acted on by a tanh function, which has very small gradients for large activations. As the derivative of the error is just the product of all the derivatives of various tanh functions at the nodes, this product will be minuscule. Thus the effect of the parameter updates is negligible, even with fancy Gradient Descent algorithms. This means that in practice, standard RNNs cannot remember correlations across many timesteps. In the next section, common approaches to circumventing this problem will be discussed.

2.3 LSTMs vs GRUs

In order to avoid the vanishing gradient problem, activations must have a gradient which is equal to one. The **Long Short-Term Memory cell** does this by using a memory cell $\mathbf{c}$, made up of four gates: the update gate $\mathbf{u}$, input gate $\mathbf{i}$, forget gate $\mathbf{f}$ and output gate $\mathbf{o}$. The update gate is rather like the hidden state of a standard RNN, taking $\mathbf{x}_t$ as input and giving

$$\mathbf{u}_t = \tanh(W_{xu}\mathbf{x}_t + W_{hu}\mathbf{h}_{t-1} + \mathbf{b}_u) \tag{5}$$

however the hidden state at a given timestep $\mathbf{h}_t$ is given by

$$\mathbf{h}_t = \tanh(\mathbf{c}_t) \odot \mathbf{o}_t. \tag{6}$$

This contains a component-wise multiplication of the memory cell output $\mathbf{c}_t$ passed through a non-linearity and the output gate $\mathbf{o}_t$, given by

$$\mathbf{o}_t = \sigma(W_{xo}\mathbf{x}_t + W_{ho}\mathbf{h}_{t-1} + \mathbf{b}_o). \tag{7}$$

$$\tag{8}$$

The output $\mathbf{o}_t$ takes in the current input and previous hidden state and combines them within a σ function, causing any large negative or positive values to go to zero or one, respectively. The state of the memory cell is given by

$$\mathbf{c}_t = \mathbf{i}_t \odot \mathbf{u}_t + \mathbf{f}_t \odot \mathbf{c}_{t-1}, \tag{9}$$

which conditionally takes into account the state of the previous memory cell, $\mathbf{c}_{t-1}$ and by being modulated by the forget gate

$$\mathbf{f}_t = \sigma(W_{xf}\mathbf{x}_t + W_{hf}\mathbf{h}_{t-1} + \mathbf{b}_f). \tag{10}$$

This is then combined with the signal of the update gate, modulated by whether or not the input gate

$$\mathbf{i}_t = \sigma(W_{xi}\mathbf{x}_t + W_{hi}\mathbf{h}_{t-1} + \mathbf{b}_i), \tag{11}$$

is closed or open. This allows information to be conditionally remembered and forgotten over time. In summary, the update gate chooses how it wants to change the memory cell output, the input gate decides how much of this change is implemented, the forget gate controls how much of the previous memory cell state can be seen and the output gate determines how much of the current cell's state is seen in the next time step. Crucially, the derivative of the state of the memory cell $\frac{dc_t}{dc_{t-1}}$ is always equal to one, thus the vanishing-gradient problem is avoided when passing information between memory cells at different

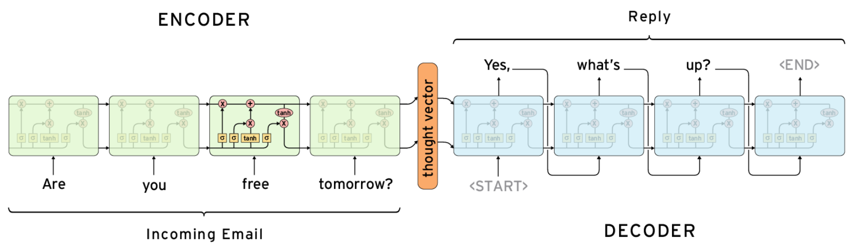

Figure 1: Schematic of encoder-decoder architecture. In this instance, the decoder is fed in previously generated tokens as input. Figure from [18]

time-steps. This important property of LSTMs vastly increases their recall range and capacity, however the extra gates add a lot more parameters to the model.

A similar, but slightly simpler form of RNN cell is called the **Gated Recurrent Unit**. In GRUs there is no forget gate **f**, so the entire previous state of the memory cell is visible to the

2.4 Encoder-Decoder Architecture

Currently, the most fundamental tool to relating two different sequences together is Encoder-Decoder architecture - this section will only discuss the case of language modelling but there are many other types of problem that it can be applied to. The idea is that a sequence of source text is fed into a recurrent neural network, word-by-word via their embeddings, so that the hidden state at the final time-step has encoded in it all the relevant information from the sequence. A second neural network then takes this encoded representation of the source phrase and attempts to generate a sequence of target data, word-by-word. The quality of each generated word is judged by whether or not it matches the corresponding word in the target sequence. Once a prediction has been made, the entire system of two RNNs is unrolled in time and the backpropagation algorithm is applied. In reality, rather than updating the weights one sample at a time, a batch of data will be fed in together and the loss will be defined as an average (for example) over the samples in the batch.

We have defined what the initial hidden state of the decoder is and what it should output, but what are the inputs? Since we want to find the probabilities of words given the sequence of words before it, at each time-step we feed in the previous word in the target sequence. During training time we can feed in the previous word of the target training data but at testing time we cannot, so the embedding of the word generated in the previous time-step is normally used. In fact, this same approach can be used at training time in order to increase the robustness of the model to its own errors, whereas using the ground truth target data decreases training time. Figure 1 shows a schematic of an encoder-decoder model applied to chat-bots, where the previously generated tokens are fed in. Apparently, during training it is best to feed in a random mixture of the off-set target data and the previously generated tokens, balancing robustness and length of training time. **I will try to implement this if possible**

3 Analysis

- How do we determine the vocabulary size?

3.1 Data Exploration

Although GRUs should be able to learn long-term dependencies in theory, in practice you would need a lot of computing power and a very powerful model to train your model. In my case it was necessary to choose a sequence length cutoff, so I decided to remove any sentence pairs where the target length was greater than 22. The fact that target lengths were chosen, as well as the length of 22, was fairly arbitrary although this roughly cuts the number of sentence pairs in half.

In order to optimise the model it will be necessary to use a subset of the training data as a development or validation set. The model will be fully trained on this subset of data, a few times, to find the most optimal setup and hyperparameters. For our English and French training datasets, the reduced dataset vocabulary sizes are 296,896 and 357,666 respectively. In order to make our model feasible to train we also need to limit the vocabulary size substantially, which could have proportionally different effects on training and development data. We need to make sure that the performance on training and testing during development to resemble that of the full training later on, so it will be good to study some statistical properties of the text.

The statistics of words in text is a very well studied field. One important and interesting finding is that the frequencies of words tend to follow a "Zipfian" distribution, whereby the frequency of a word is inversely proportional to it's frequency rank. This means that the most common word appears twice as often as the second most common, the second word appears three times as often as the third, etc. This is illustrated in the English (French) dataset, where the most common words are *the* (*de*) with a count of 60286 (1307231) and the second most common is *and* (*le*) with a count of 31213 (675469).The result of this is that a large proportion of a corpus of text can be accounted for with a relatively small vocabulary, for example 50% of the English and French datasets used are covered by just 146 and 81 words respectively! However, reading language with this many words missing would be difficult for even a fluent speaker, let alone someone/something trying to learn a language from scratch. Also, it makes accounting for uncommon words very difficult because their frequencies are so low. Another effect of the long-tailed distribution is that restricting the vocabulary size of both the development and training sets by the same amount is likely to have little effect on how much of the French and English datasets are covered. The difference is that the vocabulary in the training set misses out a larger proportion of the total vocabulary. The tables below shows a summary for the instance of choosing a vocabulary of 10000 and a development set of size 10000 from the 1013625 total training samples. We can see that over half of the vocabularies are Hapax Legomenon - words that only appear once in the corpus - which are near impossible to learn from. With all of this intuition, it should be reasonable to use the same vocabulary size for the development and training sets.

FRENCH	Corpora covered	Vocab covered	Proportion Hapax.
Dev.	90.0%	13.7%	52.4%
Full	89.7%	2.8%	52.4%

ENGLISH	Corpora covered	Vocab covered	Proportion Hapax.
English Dev	91.2	17.3%	51.4
English Full	90.9	3.4%	54.1

3.2 Algorithms

4 Methodology

4.1 Data Preprocessing

- Tokenized the Unicode data, splitting up text into sequences by line, then words and most punctuation. Some apostrophes (particularly in French), hyphens and full-stops are inherent parts of words and so were not treated as tokens in special cases. For instance *l'* in French
- Removed phrase pairs with target sequence longer than 20 tokens long
- Removed unnecessary whitespace and all capitalisation
- Removed full stops and commas preceded by spaces. Most other punctuation, apart from hyphens and question marks since these are fairly instrumental in many cases. QMs are actually a powerful indicator of the nature of a phrase.

4.2 Implementation

Before I built the neural nets, it was necessary to write some other code:

- Preprocess data and save training, development and testing datasets. As well as dictionaries from word to word-ID and vice-versa. Training data proportion of 5% used, dev. proportion 20%.
- build a batching system to split up the data and pad with zeros, choose whether time major (vertical) or not.
- Write the code for the benchmark dictionary based model, utilising the Python [GoogleTrans package](https://pypi.python.org/pypi/googletrans) as a dictionary.
- Load in polyglot word embeddings from https://sites.google.com/site/rmyeid/projects/polyglot#TOC-Download-the-Embeddings. If there is no pretrained embedding available, just initialise to Gaussian random distribution.

- Implement BLEU metric either by hand or using the NLTK implementation.

Any other helper/utility functions will be described when necessary. Next, we can begin building the model in tensorflow.

- Make tensorflow placeholders for the encoder inputs, decoder targets and decoder inputs
- Convert the input integer sequences to sequences of embedding vectors (equivalent to one hot encoding each word as a vector and multiplying it by the embedding matrix).
- Define the encoder, requires specifying the number of hidden LSTM/GRU units. Hidden units must be equal for encoder and decoder. The Tensorflow API means that different choices of RNN cell, such as LSTM or GRU are very easily chosen.

There are two main choices when handling sequences of varying lengths. We can use static RNNs, which just unroll every sequence to the same number of time steps (the maximum sequence length), and then pad the ends of shorter sequences with zeros. However, even when discarding longer length sequences, this is highly inefficient as the network will be processing a vast amount of padding tokens which have no meaning. Another option is to use dynamic RNNs, which unroll the network only to the length of the longest sequence in each batch, or to some other appropriate length. This is particularly important when generating new data or predictions, since it may not be know a priori what an appropriate length for decoder outputs[1]. This latter option is more interesting and flexible so it is the approach that I have taken. In the following discussion, I have drawn heavily from the Tensorflow source code Documentation strings and these tutorials https://github.com/ematvey/tensorflow-seq2seq-tutorials. In my code, I define the length of the decoder output tensors to be equal to the target length plus 5, to allow some room for error in the length of decoder predictions.

One problem is that the dynamic_rnn function in Tensorflow requires that the inputs for all timesteps are passed to the RNN at once on the first time-step and so does not allow previously generated tokens to be fed back into the decoder layer. This means that I had to use `raw_rnn` and define my own dynamic unrolling, with the added functionality of feeding the previously generated token in at each time-step. I also enabled random mixing of target tokens and previously generated tokens as suggested as the optimal method in the literature. Using `raw_rnn` gives you great flexibility over the API of your RNN but it was quite hard to get it to work. Essentially, you define a map from `(time, previous_cell_output, previous_cell_state)` to `(elements_finished, input, cell_state, output)`, where `time` is the given time-step, `elements_finished` is a Boolean indicating whether the all of the sequences have been terminated with an `<EOS>` token. Initially, the decoder will have `cell_output` of `None` and `cell_state` equal to the final hidden layer of the encoder. The user can define the probability that previously generated tokens are fed in, which must be set to 1 at test time. If a generated token is to be fed into the decoder, the raw RNN feeds the `previous_cell_output` through a linear layer and *greedily* finds the most likely word with *argmax*.

- Once we have the output of the decoder for a batch, we apply a projection layer, as was done inside the `raw_rnn`. The projection layer is a linear tranformation which converts the dimension of the decoder output to be the same as the target vocabulary size, enabling scores to be calculated. Due to batching, the output will be a tensor with dimension `(decoder max step number, decoder batch size, target vocab size)`, so this needs to be flattened to apply a linear layer.
- Take the projection layer and calculate an argmax over the scores for the different words. i.e. maximizing the score of the predicted word. Also then apply a softmax function to the logits/scores to get probabilities.
- Calculate the cross-entropy between the target and predicted word. The target words were one-hot encoded to easily compare them to predicted word probability distributions.
- At this point we have a tensor of dimension `(decoder max step number, decoder batch size)` since we will have calculated how well our prediction has done for each next-word prediction in the batch. We calculated a collective measure of batch loss by taking the mean of all of the cross-entropies.
- Backpropagation was performed over the two unrolled RNNs, which was done by using the simple Tensorflow API by defining an Adam optimiser and then calling the minimise method. Adam was chosen because it uses momentum to avoid local minima as well as having adaptive learning rates, while remaining highly efficient[20].

[1]This could be until an " $< EOS >$ " token has been output by the model.

Then the model was trained on a fifth of the available training data, which is 80,000 samples, in order to observe the quality of different hyper-parameter regimes.

4.3 Refinement

I began using only 6000 words in the vocabulary, since this covered about 86% of the corpora I thought it would be sufficient, however I found that many of the target phrases were difficult to understand even for a native speaker of English. A 4% improvement does not sound like much, but in reality this a decrease in likelihood of the $< \mathrm{UNK} >$ token of 28.6%. In this section I will look at the performance of the model on the development set in order to illustrate and investigate the impact of changing the number of hidden units. I will also look at training time differences between the case of feeding in encoder inputs either forward or backwards. The optimal architecture will then be trained fully on the entire dataset.

5 Results

The model was trained for 100 epochs of the training data, with.

5.1 Model Evaluation and Validation

6 Conclusion

Even if the NMT system doesn't beat the benchmark across the board, it can be used to show some interesting things.

- Although the model may not work well, does the distribution of UNK tokens matches the original text?
-

6.1 Improvement

In order to improve the performance of my model, I would use a larger vocabulary, going from 10,000 words covering around 90% of the corpus to 20,000 words covering 94-95%. This would be a further 50% decrease in the raw likelihood of encountering an $< \mathrm{UNK} >$ token. The model also seems to be particularly sensitive to the presence of unknown words, especially in long sequences. I think that the reduced number of $< \mathrm{UNK} >$ tokens would be worth the increase in computational cost.

Although I originally set out to see what a single layer in the encoder and decoder could do, I think that having two shallower layers would work better than one deeper one. This larger model would most likely be able to fit on a single GPU and they would be able to capture much more of the long-term dependencies.