Machine Translation with Recurrent Neural Networks

Henry Maguire

October 20, 2017

Abstract

Here I build, train and analyse a Neural Machine Translator using Sequence-to-Sequence modelling with unidirectional encoder-decoder architecture. The aim is to see whether a single LSTM 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[1] 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[2] of translations, log-linear models [3] 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[4], 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[5] 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[6] 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[7] and will be outlined later on. The final layer then projects a probability distribution over

[1] Google's Neural Machine Translation System
[2] Statistical Machine Translation: Della and Della
[3] Excellent overview of machine translation by Neubig
[4] Google's Neural Machine Translation System
[5] Andrej Karpathy's blog
[6] Wu et al. 2016
[7] Cho et al. 2014

predicted words in the target language at each time step. Simple RNN cells suffer from a number of pathologies, such as vanishing gradients[8], 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, 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[9], 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 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

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[10].

[8] Vanishing gradients on Denny Britz's blog

[9] Google's word2vec API

[10] Callison-Burch et al. 2006

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:

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

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

One example of a benchmark model is to just translate word for word the source sentence using dictionary definitions, via some online translation software API[11]. 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

The datasets are subsets of the WMT'14 parallel corpora which consist of translated TED talks carried out by human interpreters. I used a subset of the data made up of around 1M French and English phrase pairs. The dataset is quite noisy and very open domain, ranging from politics and current affairs to conversational language. I apply no data-selection or processing to narrow down the size of the domain which makes the problem more difficult. In fact, in 7 the authors are able to restrict the domain so 93% of the dataset is covered by only 15,000 words - this is a far smaller vocabulary than my implementation would need to have the same coverage.

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

[11] `https://pypi.python.org/pypi/googletrans`

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[12]. Even more strangely, the semantic relations between embedded words can be preserved through binary vector operations, for example the sum of embeddings:

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

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 Long Short-Term Memory Units

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

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

[12] `http://colah.github.io/posts/2014-07-NLP-RNNs-Representations/`

[13] Socher et al. 2013

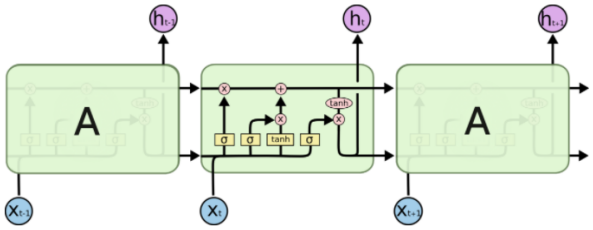

Figure 1: Schematic of an unrolled Long Short-Term Memory cell. σ represents a sigmoid gating function Figure from this blog post by Britz.

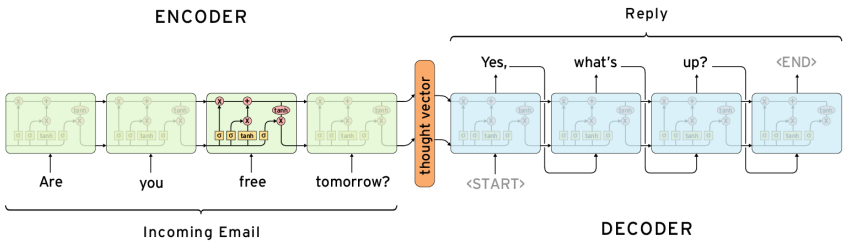

Figure 2: Schematic of encoder-decoder architecture. In this instance, the decoder is fed in previously generated tokens as input.

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 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.

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 2 shows a schematic of an encoder-decoder model applied to chat-bots, where the previously generated tokens are fed in. 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 implemented this random mixing into my final code.

3 Analysis and Data Exploration

I performed some analysis on the part of the WMT'14 dataset that I am going to use for my translation system in order to gain some intuition about how to preprocess it. I first looked at a few statistics in order to find a reasonable sequence length limit and a justification for doing so. I found that reducing the range of sequence lengths makes the algorithm orders of magnitude faster to train.

Sequence length statistics	French	English
Min. length	1	1
Max. length	124	125
Mean	24.0	20.2
Median	21	18
Mode	14	12
1st Quart.	13	11
IQR	18	15
3rd Quart.	31	26

Firstly, the French corpus has 20% more characters, so is less space efficient at conveying the same information. From figure 3 it can be seen that although the maximum sequence lengths are very high in both the French and English corpora (124 and 125 words respectively), the mean and median are far lower, with a mean larger than median due to the skewed, long tailed distribution. In fact, 25% of the Fr and En data is made up of sequences smaller than 13 and 11 tokens long, respectively and 50% by the medians 21 and 18. The most common length of phrase for French and English is 14 and 12, respectively. Due to the skewedness of the data, the interquartile range overlaps with zero in each case so there is no obvious lower bound to define outliers by using $Median - 1.5IQR$. An upper bound of $Median + 1.5IQR$ would mean that any sequences longer than 58 and 49 could be deemed outliers, using the standard approach. However, these sequences are still very long, considering that most of the data is a third of the length of this upper bound. Processing these sequences will be very challenging as their presence will cause batches to be padded with a large number of `<PAD>` tokens. The single layered LSTM encoder/decoder will also find it very difficult to compress/extract all of the information in the very long sequences. Also the above definition of outlier would also assume Poisson distributed data and this is clearly more akin to something like a log-normal distribution.

Overall from figure 3 we can see that the English corpus has far more shorter sequences than French, meaning that it is more information efficient, using fewer words to convey similar meaning than French. In order to make the model easier to train I initially considered sequences up to the median length. This means that French (English) sequences longer than 21 (18) tokens long were discarded while maintaining half of the overall dataset. In a second trial I used even shorter sequences, only up to length 18 and 15 for French and English respectively. Each of these word length cutoffs independently leaves behind 41.4% of total sequences, but both together leaves 35% of the data. Therefore 354,834 sequence pairs remain to train and test on. Although this is a harsh approximation, I think it is justified since the

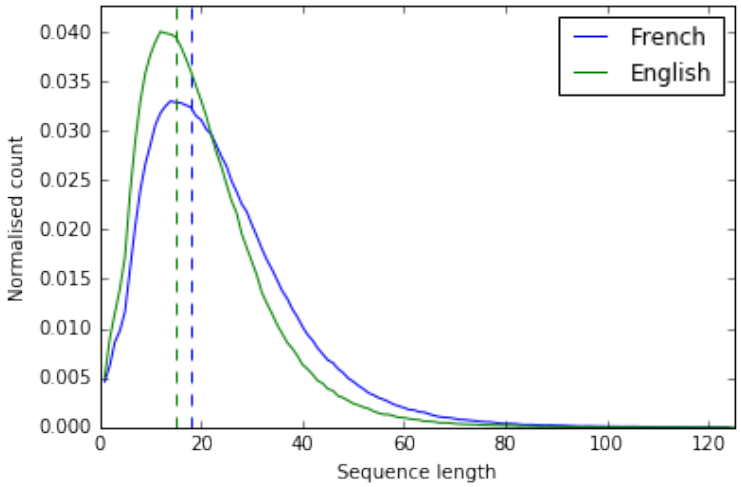

Figure 3: Normalised frequency of sequence lengths in French and English corpora. Vertical lines represent the chosen cutoff in frequency lengths which were 18 phrases for French and 15 phrases for English.

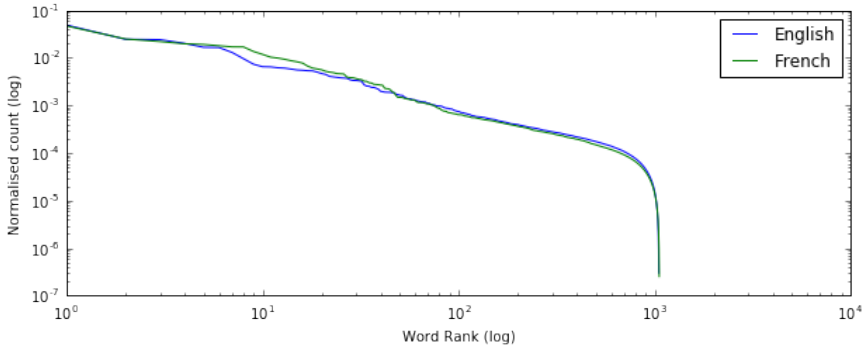

Figure 4: The Zipf distribution of word rank by word frequency in both English and French. The straight line on log-log axes indicates a power law scaling, which means that the frequency of a word is proportional to the inverse of its word rank. This allows the vocabularies to be truncated heavily.

aim of this experiment is to learn how to implement sequence-to-sequence modelling and not to create a cutting edge solution, so the decrease in rigour is worth the increase in trainability. Going from around 1,000,000 training examples to 340,000 should still be sufficient to train a toy model on.

Next I analysed word frequencies in order to choose a sensible limit to the size of the vocabulary for each language. In figure 4, we see normalised word frequency for words of each rank, that is the score that would be assigned to a word given the number of times it occurs in the text. For example, the word "the" occurs the most number of times in the English dataset and so is given rank of 1, etc. We see the famous Pareto distribution of words, the straight line indicating that there is a particular power law scaling. In this case the power law states that the rank 1 word is twice as common as the rank 2 word, the rank 2 word is three times as common as the rank 3 word and so on, which is shown nicely in the printed statistics above. The trailing off for uncommon words (large word rank) is due to the finite size of the dataset. The total vocabulary used in the English and French datasets are 140873 and 184012, with over half of the vocabulary words only occurring once in the entire corpus (known as hapax legomenon), which are nearly impossible to learn from. I chose to limit the vocabulary so that 94-95% of the data was covered, which is 25,000 French words and 20,000 English words. At around this level, 5 or 6 words in 100 will be replaced by an `<UNK>` token, this is likely to be sufficient to maintain enough information in source and target phrases for the model to learn from, while reducing the number of parameters in the model substantially.

Remarkably, it takes only around 14% of the vocabulary to cover this amount of text. In fact, with

just 207 French words and 350 English words you can cover 50% of the corpora. This also shows an interesting insight, although French has 31% more words in its vocabulary, far fewer words (41% fewer!) make up its *core* than English.

3.1 Algorithms

As outlined above and below, I have chosen to use Tensorflow to implement the sequence-to-sequence model. To make the development process as pedagogical as possible I have build the encoder-decoder architecture up without using the native Tensorflow `seq2seq` modules but instead used `tf.nn.dynamic_rnn` and `tf.nn.raw_rnn`. The full details are in the Jupyter notebook `machine-translator.ipynb`. I also used the Natural Language Toolkit Frequency distribution function to make the text analysis easier and a test/train splitting function from scikit-learn.

4 Methodology

4.1 Data Preprocessing

There are plenty of commas, full stops (periods) and other bits of punctuation in the datasets, which may affect the understanding of a sentence to a native speaker, as made famous in *Eats, Shoots and Leaves* [14], however for our intents and purposes all punctuation apart from apostrophes were removed from the texts. This is because these tokens will occur very often in the text and so the machine learning algorithms will weight their individual importance far higher than it actually is in conveying meaning. Some exceptions will be full stops indicating abbreviations, for example in F.B.I. Stripping punctuation decreased the corpora character length by around 4%, this is certainly non-negligible and will have helped a lot.

Thankfully, the dataset is already split up by whitespace into appropriate tokens. For example in French "j'" and "ai" are treated as separate tokens which make up "j'ai" to mean "I have" in English. Similarly, "didn" and "'t" are separated by whitespace. This makes tokenising the data as trivial as calling the `split()` method on each sequence string.

Two other important steps were removing rogue whitespace which caused the empty string '' to be one of the 20 most common words in the vocabulary. Another was removing capitalisation, which decreased noise in the dataset substantially by allowing counts of words to not be split into two by the presence of leading capital letters.

4.2 Implementation

Here I offer a brief *recipe book* for how I built the encoder-decoder model. Please see the iPython notebook for a detailed explanation. In the following discussion, I have drawn heavily from the Tensorflow source code Documentation strings and these tutorials[15].

Firstly, the source and target datasets were preprocessed as outlined above and dictionaries were compiled which have key-value pairs of words and word-IDs and vice-versa. The words were then replaced with their IDs or 2 if the word was not within the vocabulary. A batching system was then written to split up the data and pad shorter sequences with zeros. The data also needed to be formatted so that time-steps went down the columns of the input matrices.

I used pretrained word embeddings in order to speed up the training process, these were obtained from [16]. If there is no pretrained embedding was available, the embeddings were just initialised to a Gaussian random distribution with the same mean and variance as the overall embedding matrix.

The Natural Language Toolkit BLEU score implementation was used, which takes in multiple reference phrases and translation attempts and scores them as was outline above.

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

Firstly, tensorflow placeholders for the encoder inputs, encoder input lengths (the reason why is explained later), decoder targets and decoder inputs were made. The batches of input integer sequences were converted to batches of sequences of embedding vectors (equivalent to one hot encoding each word

[14] Eats, Shoots and Leaves
[15] Sequence to sequence tutorials by ematvey
[16] Pretrained word embeddings

as a vector and multiplying it by the embedding matrix). This means that the embedded inputs are tensors of dimension `(max_time_step, batch_size, embedding_hidden_dimension)`.

Next, the encoder was defined which requires specifying the number of hidden LSTM 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[17]. This latter option is more interesting and flexible so it is the approach that I have taken. 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 logits and calculate an argmax over the scores for the different words. i.e. maximizing the score of the predicted word. A softmax function was then applied to the logits/scores to get probabilities. The probability distributions are compared with the one-hot encoded predictions using cross-entropy. Now we have a tensor of dimension `(decoder max step number, decoder batch size)` since we will have calculated how well our prediction scored for each next-word prediction in the batch. A collective measure of batch loss was then calculated 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 combines elements of AdaGrad, where parameters which are updated less often get updated more strongly, and momentum where updates depend on the overall slope of previous updates. These features help to avoid local minima as well as performing efficiently[18].

Then the model was trained on a 15% of the available training data, which was around 75,000 samples, in order to test how long the model took to train given different sets of hyper-parameters. The encoder inputs were also reversed, as this has been shown to speed up training time substantially in language translation models. For example, rather than translating from $[a, b, c] \rightarrow [\alpha, \beta, \gamma]$ the source sequences are reversed: $[c, b, a] \rightarrow [\alpha, \beta, \gamma]$ so that a and α are in close proximity in time. This is a main theoretical contribution of this seminal paper[19].

4.3 Refinement

Even using the drastically reduced phrase lengths proved to be difficult to train properly. The maximum sequence lengths were decreased even further, to 18 and 15 for the preprocessed French and English raw

[17] This could be until an "$< EOS >$" token has been output by the model.
[18] AdaM paper: Kingma and Ba 2015
[19] Sutskever et al. 2014

data, leaving 354,834 sequence pairs for training and testing. Just making this change cut 500 seconds per epoch off the training time. Encoder and decoder hidden unit numbers were experimented with and it seemed that choosing 512 gave good performance while still being relatively quick to train.

5 Results

The model was trained in batches of 100 for as many epochs of the training data as it took for the loss to stop decreasing substantially, this was around 35 epochs. Predictions were made on the test data every 5 epochs in order to observe how the training time affected the validity of the model. The feed previous probability was set to be $P_f = 0.7$ and the number of hidden units in the encoder and decoder was $N_h = 512$ each. Another model was trained with the same parameters but for $P_f = 0.5$. In figure 5 it can be seen that the feed previous probability does not affect the model quality significantly. The model also seems to have learned all that it can already by only 5 epochs. In the discussion below, $P_f = 0.7$ since this scores slightly higher and is likely to be more robust, and the number of epochs is 30.

After only around 6 epochs the model was making predictions on the training data which were more accurate than some of the data, for example:

Source: "scrutin du jour du lendemain le h 9"
actual: "there is a tie or difference between first 2 candidates is less than 10 votes"
predicted: "9 are no day in a between the day months".

Here the English translation is totally incorrect and is obviously just some noise in the dataset, however the NMT prediction is in the right ball-park, since the google translate answer is "9:00 am the day after polling day". Similarly, after 16 epochs the model made predictions on of the training data with a very similar meaning to the correct translation but expressed differently, for example:

Actual: some 15 of the park's native mammal species are considered rare threatened or endangered
predicted: several of percent the species's marine mammal species are rare threatened or endangered

which is quite impressive, given the short training time. Since the model was being fed 30% of ground truth target data as input during training, and had been learning these phrases for several epochs, these training outputs are not too indicative of success. Next section the results on the test set will be analysed.

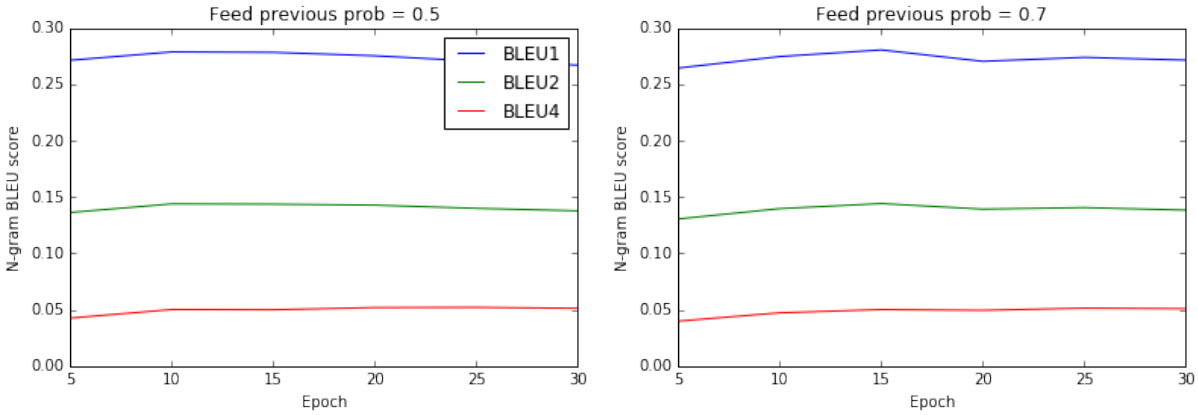

Figure 5: N-gram BLEU scores for feed previous probabilities of 0.5 (left) and 0.7 (right).

5.1 Model Evaluation and Validation

The corpus Unigram BLEU score, which is an average over all sequence unigram BLEU scores, for the Benchmark model was 0.347, whereas the NMT method achieves 0.277. This effectively means that the two methods manage to decode 35% and 28% of the total amount of information correctly, respectively. Similarly the BLEU2 and BLEU4 scores were 0.195 and 0.0876 respectively. In contrast, the NMT method achieves Bigram and 4-gram scores of 0.14169 and 0.0526, so one could conclude that the benchmark model performs better at this task.

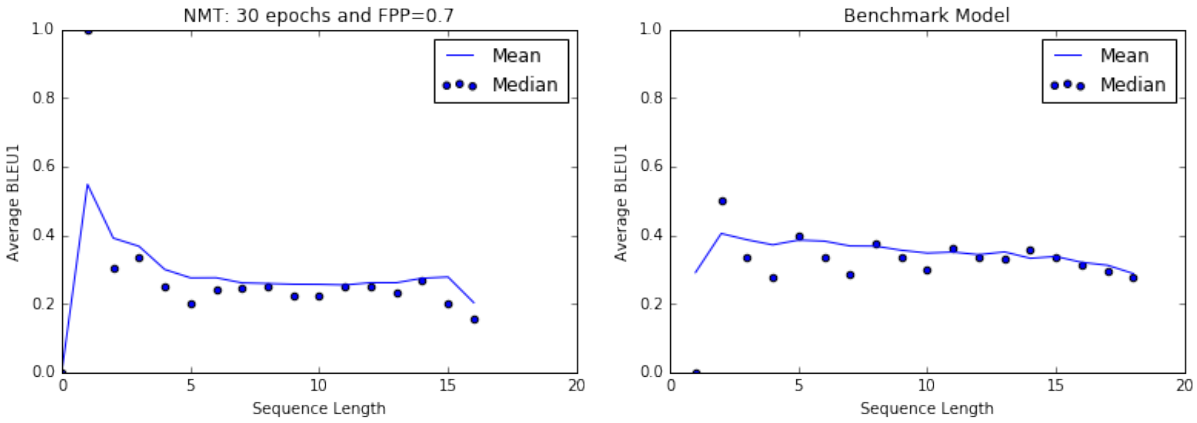

Figure 6: Average BLEU1 score for the NMT (left) and benchmark model (right) for sequences of each length.

Some examples of qualitatively well-matching translations (compared to the English reference phrases):

Target : 'we apologize for any inconvenience this has caused'.
Translation: 'we apologize for any inconvenience this may cause'.
Benchmark: 'we we apologize of the unpleasantness what this change a could you cause'

Target : 'guilty pleas reduce the length of trials'.
Translation: 'guilty pleas reduce the length trials trials'.
Benchmark: 'the pleas from guilt shorten the duration of the trial'.

Target : 'the stratospheric ozone layer prevents most ultraviolet radiation from reaching the earth'.
Translation: 'the ozone ozone layer prevents ultraviolet ultraviolet radiation reaching reaching the earth
Benchmark: "the layer after ozone stratospheric stop l' essential of influence ultraviolet after reach the terre".

In the Benchmark model the ordering of words is confused and some words have failed to be recognised by the dictionary, such as "l'" and 'terre". The NMT often gives something with a similar syntactical structure but says something quite unrelated e.g. :

Target : (sic) 'alternatively hormones may act on structures other then the acl'.
Translation: 'moreover `<UNK>` can be aggressively on in structures structures structures'.

Here we can also see here that the predictions have a tendency to repeat words at the end of sequences. It is also worth noting that the 4-gram BLEU score was not found to be useful at all, for example the NMT output:

Target: "it accurately portrays the agency's plans and priorities"
Prediction: "it accurately portrays the organisation's plans and priorities"

which is a veritable success, scored 0.59 BLEU4, whereeas:

Actual: "we all very much miss his exemplary spirit and skills"
Prediction: "the and and skill skill skill to all of all stories"

scored 0.65 BLEU4, even though it is essentially a failure.

In figure 6 the unigram BLEU scores are averaged over sequences of the same length. It can be seen that both the NMT and benchmark model perform fairly consistently over phrases of different lengths, with the NMT performing slightly worse for longer phrases and extremely well for single word phrases. The performance of LSTM cells is likely to decline for even longer sequences, as outlined in Cho et al7.

The NMT produces very similar phrase length statistics to the target corpus, with the same mean, median and 1st quartile of 9.4, 10.0 and 8.0 words, respectively. This is compared to the BM which produced slightly longer predictions, with mean and median of 11 and Q1 of 8.0. This is expected since some dictionary equivalents of singular French words will be multiple English words long, without

incurring any penalty, whereas losses due to predictions being too long are minimised when training the NMT.

Only 64% of words chosen by the benchmark model were actually in the target corpus. In effect, this could be lead to failures on the part of the BLEU scoring method I have chosen, since the target test corpus is made up of only singular reference sequences, rather than different paraphrases of the same phrase.

6 Conclusion

In summary, a single-layer unidirectional Neural Machine Translation system was built with encoder-decoder architecture. The system was trained on source (target) sequences up to 18 (15) tokens long on a relatively noisy and open-domain dataset, taking only a few hours on a single GPU. Tokens predicted by the system at each time-step were fed in as inputs for the next time-step as a statistical mixture with the ground truth data in order to improve robustness and decrease training time.

Overall the NMT system has an impressive ability to maintain information/vernacular, although is not consistently reliable at conveying the same meaning and generally not convincing as a human interpreter. On average it does not outperform the benchmark model on our chosen metrics, but often produces more fluent results than it.

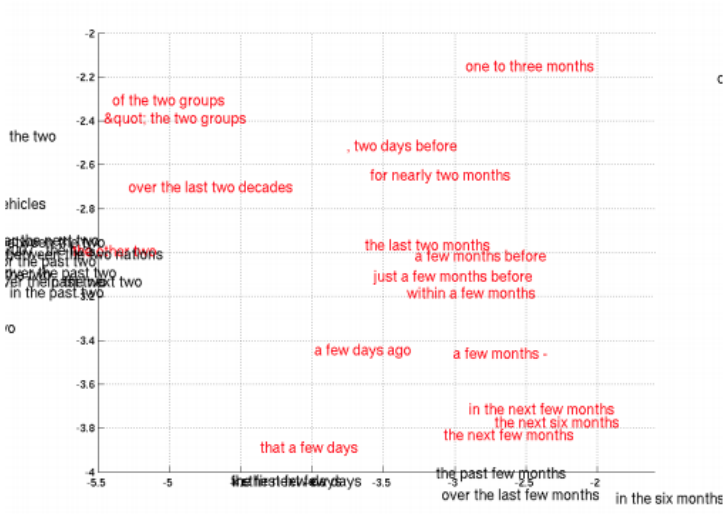

Figure 7: Hidden layers visualised using t-SNE representing phrase embeddings or 'thought vectors'. Image from 7.

The key advantage of the NMT solution is the fact that it finds a distributed representation for sequences within the final hidden state on the encoder layer, which enables the decoder the learn the meaning of sequences holistically. Figure 7 shows an image from 7 where the authors project the various column vectors of the final encoder hidden states onto a 2D space using a dimensionality reduction algorithm called t-SNE. It is clear from this figure that there is some notion of *distance* between different phrases - this particular neighbourhood of the phrase space deals with different concepts of time.

The dictionary method may have beaten the NMT method in this study, however it is unclear how you could push the dictionary method further, whereas this sequence-to-sequence model was incredibly primitive and can be improved enormously in a modular fashion. This makes it excellent for more difficult translation tasks, such as that between Japanese and English, where the encoding of meaning in words and symbols is even more abstract. This abstract meaning can be encoded within the hidden state of an LSTM without having to resort to a granular, dictionary method.

6.1 Improvement

It seems that the model is not expressive enough to make a convincing translation system. I think that having two shallower layers would work better than one deeper one while still being able to fit on a

single GPU. It would be interesting to analyse the qualitative effect that this has on the predictions and the increase in ability to handle longer sequences. I would also like to experiment with attention mechanisms, as these have been shown to increase performance drastically without increasing the number of parameters significantly. Bidirectionality of the encoder and decoder layers would also be likely to improve the performance of the model significantly, without a huge increase in computational cost. Writing as much of this by hand would be of great interest to me in the future.

On a more technical note, I could not work out how to save an entire sequence-to-sequence model effectively in Tensoflow. This made replicating results very difficult and I had to just save predictions on the test set during runtime. This was a major hindrance to development and would certainly be something I would change in the future.