2017

An alternative approach to training Sequence-to-Sequence model for Machine Translation

Vivek Sah
Colby College

An alternative approach to training Sequence-to-Sequence model for Machine Translation

By: **Vivek Sah**

Advisor: **Stephanie Taylor**

Honors Thesis

Computer Science Department
Colby College
Waterville, ME
May 9, 2017

Contents

1 Abstract

Machine translation is a widely researched topic in the field of Natural Language Processing and most recently, neural network models have been shown to be very effective at this task. The model, called sequence-to-sequence model, learns to map an input sequence in one language to a vector of fixed dimensionality and then map that vector to an output sequence in another language without any human intervention provided that there is enough training data. Focusing on English-French translation, in this paper, I present a way to simplify the learning process by replacing English input sentences by word-by-word translation of those sentences. I found that this approach improves the performance of a sequence-to-sequence model which is 3-layer deep and has a bidirectional LSTM encoder by more than 30% on the same dataset.

2 Introduction

Deep neural networks have been shown to very effective at wide variety of tasks including computer vision, speech recognition, image captioning and many others. The way they work is that provided enough training data, these models learn to approximate functions which map the input data to the output data. We will briefly discuss how neural networks work in general and then talk about recurrent neural networks to develop understanding of sequence to sequence models.

2.1 Neural Networks

The building block of a neural network is called a neuron. Each neuron takes in inputs and gives an output. Let's look at how a sigmoid neuron works. Let's say x_1, x_2, x_3 are inputs to this neuron.

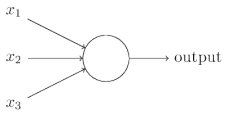


Figure 1: A simple neuron

Each input is associated with a weight value. To calculate the output for this neuron, we multiply the inputs by their wights and add them. We then add a bias term. Let's call this quantity z, that is, $z = \sum x_i w_i + b$. Next, we squash the z value using a sigmoid gate (called activation gate). Overall, we get:

$$output = \frac{1}{1 + e^{-z}}$$

.

A neural network consists of layers of such interconnected neurons. In general, it consist of an input layer, one or may hidden hidden layers and an output layer. A simplified diagram would look like:

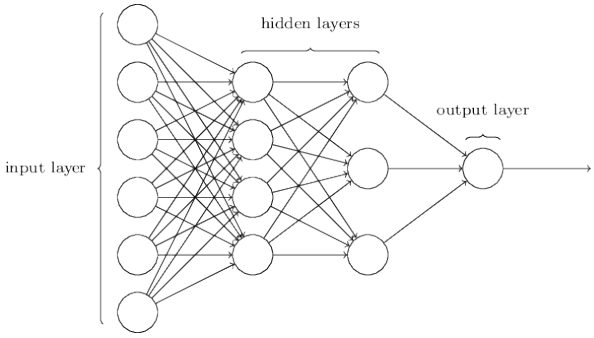


Figure 2: A neural network

All the nodes in the input layer take in an input, calculate output and pass it as input to the nodes in the hidden layer. The hidden layer nodes do the same and pass their output as input to other hidden layer nodes or to one of the output nodes. The ouput of output layer nodes is the output of the whole neural network. Each of the interconnections between the nodes corresponds to a weight variable.

To make the network learn, we calculate the loss between the predicted output and desired output using a loss function and minimize it using an optimization function such as gradient descent and backpropagation. The details of these algorithms are not discussed here. They basically find out how much each variable, such as weights, affects the loss and by how much they should be changed so as to drive down the loss.

Each training step involves calculating loss and updating weights and biases to decrease it. With enough data and training data, these models learn to map inputs to outputs.

2.2 Recurrent Neural Networks

The simplified neural network that we just discussed has one major limitation. It has fixed size input and is thus suited to tasks where the input sequence varies. This is especially true when we are dealing with languages. Let's look at the task of predicting next word in a sentence given previous words in the sentence. For this purpose, a fixed input layer neural network would not work. To address this limitation, we use a variant of neural network called **Recurrent Neural Network**. It consists of single cell, which takes in a input, calculate its call state, and then gives an output. With each new input, the RNN cell updates its state which can be thought of as updating memory. However, in practice, we roll out the network depending on the length of input sequence and visualize it as:

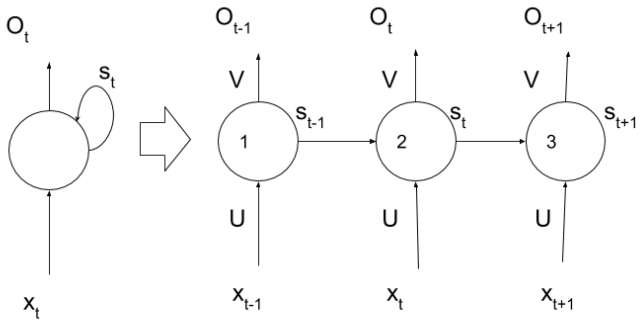


Figure 3: A recurrent Neural Network

With each input, x_t, the RNN cell updates its cell state s_t with the following rule:

$$s_t = \tanh(Ux_t + Ws_{t-1})$$

.

Here, s_{T-1} is the previous cell state and tanh squashes the value between -1 and 1.

To calculate the output, we use the following rule:

$$o_t = softmax(Vs_t)$$

At time step t, these networks can also be thought of as regular neural network except that the weights connecting input nodes (U) to the hidden nodes(RNN cell) are same across all inputs. Similarly, the weights connecting hidden nodes, W, are also same and so are the weights connecting each hidden node to the output. With this analogy, we can see that the regular optimization and backpropagation algorithms can be applied to train such networks.

However, this version of RNN does not do very well on longer sequences because of something called *vanishing gradient* problem which makes the network forget what it saw a certain time ago. The solution to this problem is to replace the vanilla RNN cell with something called LSTM (Long Short Term Memory) cell. The details of how it works and solves the problem of remembering long term dependencies can be found in this paper [3].

2.3 Machine Translation

Translating documents using machines is a very old problem. To do this, one could employ someone who knows both the source language and the target language. However, this is very slow in practice. So, scientists have been working for several years to come up with a automated machine translation system. Before looking at how they did it, let's think about how would one approach the problem of translation.

- **Word by word translation**: This is the simplest way to approach this problem. One can just replace each word in the source sentence with the corresponding word in the target language.

Figure 4: Example of word by word translation

- **Use grammar rules of the source and target language**: Translating word by word ignores context and might not work well. Instead, one can try to incorporate grammar rules to divide the sentences into chunks of words, translate and reorder if necessary. For example:

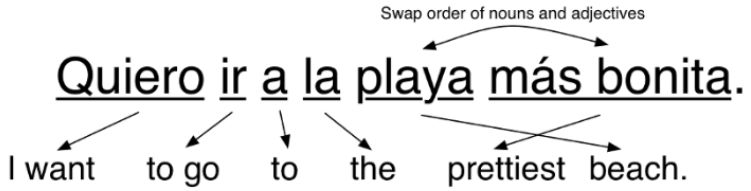


Figure 5: Example of using grammar rules

In fact, the second approach was the one which scientists building the early translation systems used. They brought in linguists who were expert in the interested language. They would program each and every language rule to build the translation system. However, this approach was only effective for official documents which were written in standard grammar. For normal communication, this method was not very effective.

- **Use statistics**: Since rule based system did not work for normal human communication, new models were developed which used statistics. They used *parallel corpora* to train the models. Parallel corpora contains the same text written in multiple languages.In fact, ancient Egyptian hieroglyphics were decoded using the Rosetta stone which had a parallel translation into Greek.Instead of generating one translation,the statistical model looks at multiple translation of the same group of words in the training data and scores them based on their probability of occurring in the training data.

  It then generates all the possible sentences. For example (in Figure 6), it could generate *I want to go to more pretty the beach.* It could also generate *I want to go to the prettiest beach.* It then scores each possible sentence based on how frequently the chunks in sentence occur and how normal they sound based on the training examples. The second

Figure 6: Example of using statistics

sentence definitely sounds more English and contains chunks which are more likely to occur in an English sentence.

Statistical machine translation worked really well and was incorporated in Google Translate in 2000s.However, these models are hard to build and maintain. Moreover, for every pair, one has to build a whole new model.

This is where Deep Learning made a significant impact in this field. With Deep Learning models, one can treat them as a black box and just feed in input and output training sample and let it learn everything by itself without needing someone to tweak anything. The same model, if fed different language pairs, would get trained to translate between those pairs. We will next discuss this Deep Learning model. [2]

3 Sequence to Sequence models for English-French Translation

As we have already seen, when we are working with sequences (sentences) where input length could vary, recurrent neural networks seem to work best because of their flexibility. Hence, we will use RNNs to build a neural machine translation model. This model is called **Sequence to Sequence** (Seq2Seq)

model. This was a major breakthrough for machine translation. The model we are going to discuss was proposed by *Sutskever et al.*[8] who were inspired by earlier works by **Cho et al.**[1] and **Kalchbrenner and Blunsom** [4].

3.1 The Model

Seq2Seq model consists of an encoder and a decoder. The encoder takes in an input sequence and converts it to something called thought vector (or an immediate representation). Then the decoder takes the thought vector and converts it to a output sequence. We use a recurrent neural network for encoder and a another recurrent neural network for decoder. The complete setup looks like this:

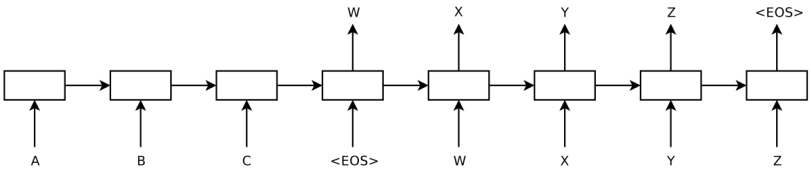


Figure 7: Sequence to Sequence model

While training, we might encounter long term dependency, so we will use LSTM which handles this issue as discussed earlier.

Assume we have input sequence $\{x_1, x_2....x_t\}$ and output sequence $\{y_1, y_2....y_{t'}\}$ where t may not be equal to t'. Using encoder-decoder system, we can calculate the conditional probability $P(y_1, ..., y_t|x_1, ..., x_t)$. As discussed, to calculate this, the encoder LSTM will first obtain a fixed size vector v for the input sequence $\{x_1, x_2....x_t\}$. We then use the decoder LSTM as a regular LSTM RNN Language Model. In a RNN Language Model, we input a word/token in the first node and it will output a probability distribution over all the target vocabulary. We take the most probable word/token and feed that as the next input and continue. In this way, we will get the required conditional probability that we were looking for.

$$P(y_1, ..., y_t|x_1, ..., x_t) = \prod_{t=1}^{T'} P(y_t|v, y_1, ..., y_{t-1})$$

3.2 Implementing Seq2Seq model

There are various available libraries which let us implement these models relatively easily. **Theano**, **PyTorch**, **Keras**,**TensorFlow** are some of the most popular ones. For the implementation, I chose TensorFlow mainly because it makes it easy to run the code on multiple GPUs, has a strong community and has a lot of tutorials, including one for machine translation.

TensorFlow has a built-in Seq2Seq function which takes in input sequence and target sequence and returns outputs and losses. However, I chose to implement it myself to better understand the model. I used other TensorFlow features to build a encoder and then a decoder. Moreover, TensorFlow has APIs for python which makes it easy to implement.

Below, I will describe my implementation of the model which I trained to translate from English to French.

3.2.1 Initialization

I first define the parameters used for the model such as RNN cell, size of hidden state of encoder, size of hidden state of decoder, number of layers for the encoder RNN and decoder RNN. I use LSTM cells for both the encoder and decoder. I also define the vocabulary size of source ($encoder\_vocab\_size$) training file and target training file ($decoder\_vocab\_size$).

Then, for each item (integer i) in the input sequence, I first convert it to a vector representation. To do this, the most straightforward way to do it make a one-hot vector, that is, a vector of length $encoder\_vocab\_size$ whose all coordinates are 0 except the i^{th} position which is 1. However, this results in very high dimensional vectors and matrices in our system. This makes the computation slow and resource-intensive. Instead, we will embed each input integer to a vector in a embedding space of lower dimension. We define a parameter $embedding\_size$ which is less than the vocabulary size. So, for input sequences, we define $encoder\_embeddings$ matrix which has $encoder\_vocab\_size$ rows and and $embedding\_size$ columns and randomly initialize it. For integer i in the input sequence, we take the i^{th} row of this matrix. We do the same for the decoder.

While training, this matrix gets updated to give us better representation of inputs in the embedding space.

3.2.2 Encoder

For the encoder, I used the TensorFlow built in RNN function which takes in embedded input sequence, the LSTM cell and returns the outputs and final state of the encoder RNN. At this point, we are not interested in the outputs of the encoder RNN, we just need the final state vector v of the encoder RNN. This vector v is the thought vector that we were talking about earlier.

3.2.3 Decoder

I used the TensorFlow built-in RNN function too but a slightly different version of it. In the The difference between encoder and decoder is that for the encoder RNN, we have all the inputs provided. But for the decoder, we have to generate the outputs from the thought vector v provided by the encoder RNN . The way we do that is as follows:

- Set the initial state of the decoder RNN to be the final state of the encoder RNN.
- Set the initial input of the decoder RNN to be EOS token.
- Get the initial output vector, find the token with the highest probability, get the and make it the next input to the RNN.
- Make a list of all the outputs from the decoder RNN and compare it to the expected outputs and find the loss. The loss function used here is cross entropy loss function.

After calculating the loss, I use the provided optimizers to minimize the loss. Note that this is a very general setup and depending on the data provided, it will work for any sequence to sequence mapping task. I will now describe how I prepared data to train this model for English-> French translation.

3.2.4 Loss function

To evaluate the performance, we need a loss function. For sequence to sequence models, **cross-entropy loss** function is widely used for evaluation.

The cross entropy function is defined as:

$$H_{y'}(y) := -\sum_i y'_i \log(y_i)$$

Then we calculate

$$\text{perplexity} = e^{loss}$$

3.2.5 Data preparation

I scraped some online sources to get about 2000 short and simple English sentences. Then I used Google Translate to get the corresponding French translation. About 1600 sentences were used for training and 400 used for testing. Since the sentences were simple, I assumed that the translation (from Google Translate) is accurate. Then I divided the data into four files -*train-source, train-target, test-source, test-target*. The model trains on *train-source, train-target* (English) files and we can test using *test-source, test-target* (French). Since we cannot feed in literal words into our model, we have to convert them to numbers. For English-French translation data, I did the following:

- Scan the training source and target file. For each new word, map it to an integer value and save the mapping. Reserve special values for *EOS* (for end of sentence), *UNK* (for words not in the mapping), and *PAD* (for empty space) tokens. To convert predicted tokens to words, also save a reverse mapping.

- Using the mapping, write a new tokenized file for all the four files *train-source, train-target*,*test-source, test-target*. In the tokenized version of these files, the words are replaced by the corresponding token.

  Since the mapping was constructed by just scanning the training files, there will be some words in the test file, which do not have a corresponding token value. These words will be replaced by a special *UNK* token.

3.3 Performance

For the training, we take a batch of tokenized sentences from the source and target data. Next we find the length of longest sentence and pad the rest

sentences in the batch to make it equal to that length. We do that same for target batch. Now, we can pass the batch of padded input and output tokenized sentences into our Seq2Seq model.

The Seq2Seq model predicts a sequence if output tokens. We then use the reverse mapping to convert them back to words.

The model was trained for 6000 steps and at every 25 steps, its performance was checked by feeding in unseen sentences from the test data. Let's see how the model performed while changing various parameters. Note that the graphs show the perplexity of our Seq2Seq model against number of training steps. For a better visualization, instead of plotting raw perplexity values, we use moving average. This step decreases the zig-zag nature of the curve while preserving its nature. For the models, I used area under curve of perplexity against training time steps as a measure of performance. The lower the area under curve (AUC), the better the performance.

3.3.1 Effect of optimizers

TensorFlow provides various built-in optimizers such as *GradientDescentOptimizer, Adam Optimizer, Adagrad Optimizer*. Optimization in deep networks is a complex topic and will not be discussed here. They are discussed here [5] and [6] talk about these concepts in detail. We will however look at the performance of our model with different optimizers.

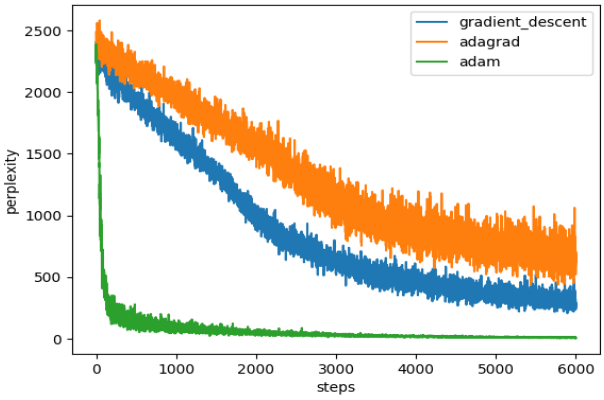


Figure 8: Effect of optimizers on learning

As we can see during the training Adam Optimizer converges faster. Hence, for the next experiments, Adam Optimizer is used.

3.3.2 Effect of Bidirectional Encoder

Let's say we want to translate the sequence $x_1, x_2, ...x_T$ to $y_1, y_2, ...y'_T$. If we have an incomplete sequence $x_1, x_2, ..x_t$ where $t < T$, its meaning might not be clear and might depend on future words in addition to past words. So, it would be nice if we could train our encoder RNN in both directions with respect to the input sequence. Such RNN is called **Bidirectional RNN** and was first proposed by Schuster and Paliwal in [7]

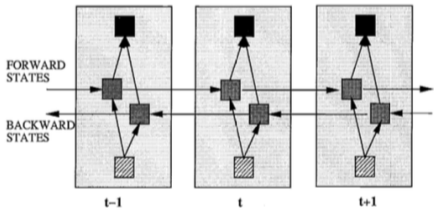
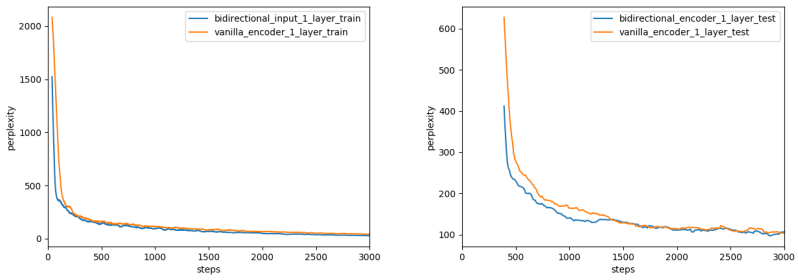


Figure 9: Bidirectional RNN

In this RNN, the neuron has two independent states, one for forward direction and one for backward direction. To get the thought vector v from the encoder, we concatenate both the RNN forward cell state and backward cell state.



(a) Training (b) Testing

Figure 10: Performance of Bidirectional Encoder

For training, the AUC decreased by 25.56 whereas for testing, the AUC went down by 18.04 %.

3.3.3 Making it Deep

Now we can see that the bidirectional encoder works better than vanilla encoder while building Seq2Seq model. Now, we want make the network deep and see how it performs. The deeper the model, the better the immediate representation (or the thought vector) will be.

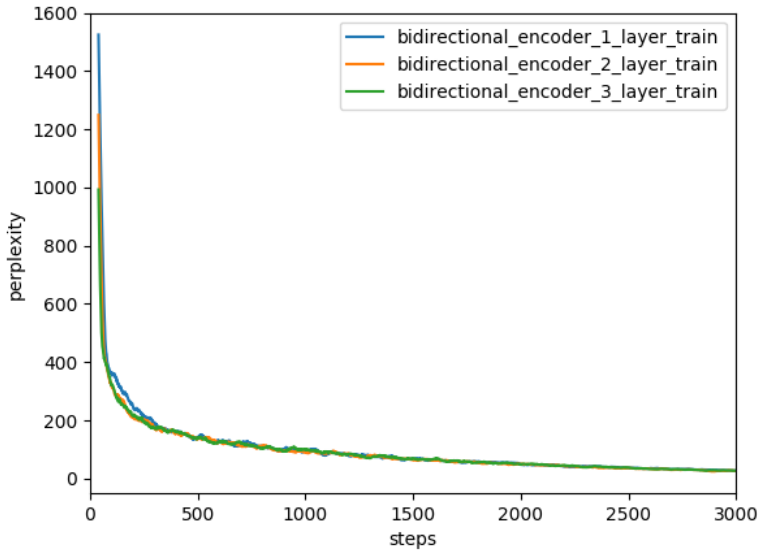


Figure 11: Training results for Deep Model

Compared to 1 layer, having 2 layers of encoder and decoder brought down the area under curve by 7.33 % for training and 10.37% for testing. On the other hand, having 3 layers brought down the area under curve by 9.36 % for training and 12.36 % for testing.

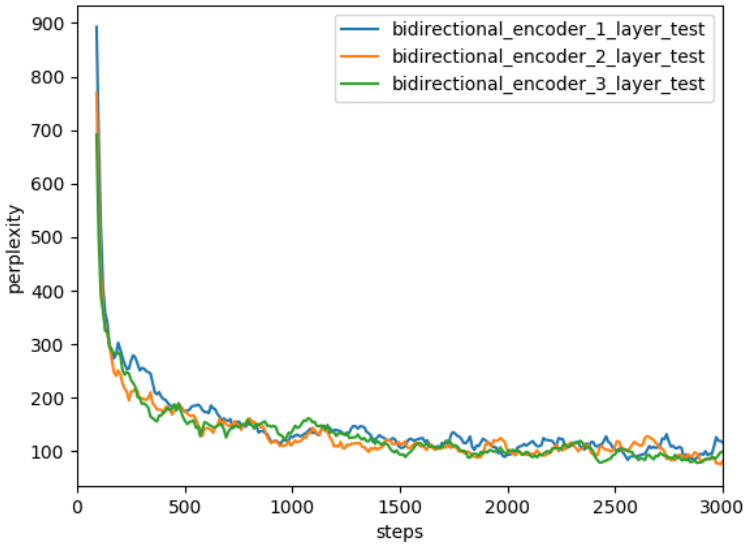


Figure 12: Testing results for Deep Model

4 An alternative approach

Translation from English to French is a tough challenge for anyone, especially when they are not familiar with both the languages. Our model is certainly not aware of English or French. So, I thought about whether I could make this task easier. One of the most straightforward way to translate is to replace each word in the English language with the corresponding French word using a dictionary. This approach does not work always as we saw earlier. But it does get us somewhere. For someone who knows some French, he/she might understand it or even better correct it. In other words, the problem of translating from English to French can be reduced to a problem of correcting a scrambled French sentence to a correct French sentence. So, the question arises: What if we train our Seq2Seq model to tackle the latter problem instead of the former on? The hypothesis is that it would be easier to train the model to solve the problem of correcting an incorrect French sentence to a correct Sentence would be easier than translating an English sentence to a French sentence.

4.1 Performance

The underlying model is left unchanged to check the effectiveness of the new model except for one aspect. The embedding matrix for source and target is same for the new setup. This is because the source and target language is same. So, they share the same vocabulary.

I ran the new set up with 3 layers, bidirectional encoder and compared with the regular 3-layer bidirectional-encoder English-French translation system. The results are given below:

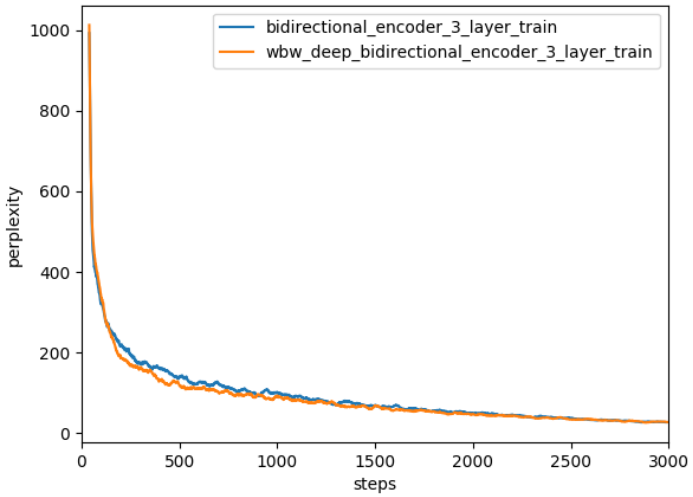


Figure 13: Training results for alternative approach

During training, both models performed similarly. The word-by-word model slightly outperformed and decreased the area under curve by 4.41 % . However, as predicted, the word-by-word model greatly outperformed the regular model during testing and decreased the area under the curve by 34.01%. This is a remarkable result and shows that the word-by-word model is very promising.

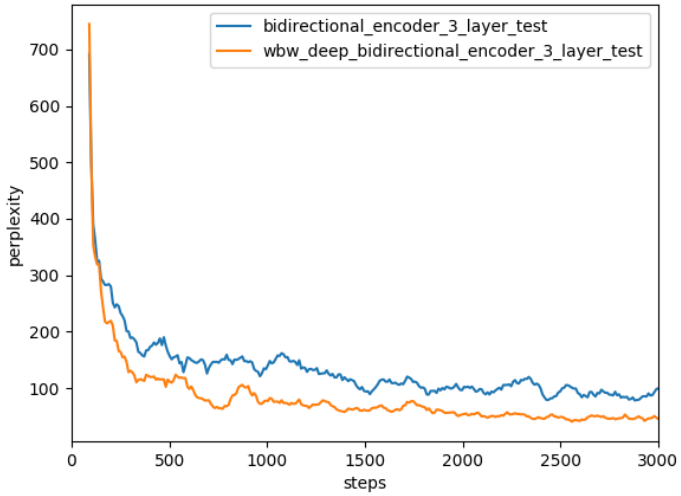


Figure 14: Testing results for alternative approach

5 Conclusion

In this paper, I explored sequence to sequence model and implemented it using TensorFlow. I showed how using Adam Optimizer, using bidirectional RNN for encoder and making the network deep improves the performance of the Seq2Seq model over vanilla Seq2Seq model. I then provided an alternative approach to machine translation by first word-by-word translation and then training Seq2Seq model to convert word-by-word translation into proper translation. The results were really promising and have inspired me to further investigate this idea. I used short and simple sentences and a rather limited data for this project. The big question now is whether this new approach would work on more complicated data.

6 Acknowledgments

I would really like to thank Professor Stephanie Taylor, my advisor, for her continued guidance and encouragement over the past year. I would also like to thank Randall Downer for helping me use the GPU clusters so that I could train my models and experiment with them. Without their support,

this thesis would not have been possible. Furthermore, I would like to thank Colby Faculty members and my family and friends who have supported me over the past four years at Colby.

7 References

[1] K. Cho, B. V. Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoderdecoder for statistical machine translation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014.

[2] A. Geitgey. Machine learning is fun part 5: Language translation with deep learning and the magic of sequences, Aug 2016.

[3] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, Nov. 1997.

[4] N. Kalchbrenner and P. Blunsom. Recurrent continuous translation models. In *EMNLP*, volume 3, page 413, 2013.

[5] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[6] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[7] M. Schuster and K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):26732681, 1997.

[8] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215, 2014.