An Experiment with Sequence to Sequence Networks and Story Generation

Ryan Zembrodt
University of Kentucky
ryan.zembrodt@uky.edu

Introduction

The project[1] discussed here, supervised and guided by Dr. Harrison, had the end-goal of training a model from a novel, in particular *Harry Potter and the Sorcerer's Stone*, using a sequence to sequence network. This model would then be able to predict, or "generate" an output sentence given an input sentence, and therefore be able to generate stories. Several topics within machine learning and natural language processing were explored with this project, including recurrent neural networks, sequence to sequence networks, and word embeddings. A large portion of the project was dedicated to the study of various word embeddings and their performance and effectiveness when training a model on the corpus using a sequence to sequence network. The final result shows that out of all embeddings trained and tested, a pre-trained *GloVe* embedding, provided by the authors of its paper, gave the best model, but still with room for much improvement. This project ended up being the largest *Python* project I've worked with, increasing my knowledge in many related areas, both within machine learning as well as more general programming concepts.

Background

Of the various strategies used in this project, the most important for the project is the recurrent neural network (RNN). An RNN is a feed-forward neural network that allows information to persist due to loops within them. For some neural network with input x_i and output y_i, its last hidden state, h_{i+1}, will also be outputted and used as an input, along with x_{i+1}, in the next step's input for the neural network. If thought of unraveled, an RNN will be given a sequence of input $(x_1, \ldots, x_t)$, and compute the sequence of output $(y_1, \ldots, y_t)$. Each value in the input sequence will be given to a separate copy of the neural network, with each network connected as a chain by their last hidden states. (Olah 2015) This hidden state connection allows information to pass between steps in the neural network, and therefore persist across the entire input, allowing RNNs to map sequences to sequences if the size of input and output is known ahead of time. This sequence to sequence mapping is why RNNs have had success with various problems, such as language translation. (Sutskever, Vinyals, and Le 2014)

[1] github.com/zembrodt/story-generation

Sequence to sequence networks take the concept of RNNs a step further: having two RNNs work together to map an input sequence to a target sequence, with one taking the input sequence, the other computing the target sequence. (Sutskever, Vinyals, and Le 2014) These two RNNs are known as the encoder and decoder. The encoder RNN maps the input sequence into a single vector of a fixed size, therefore encoding the "meaning" or "concept" of the input. The decoder will then take the encoder output, and compute a target sequence. This concept removes the restriction that the output sequence size is known for every input sequence. A simple decoder uses only the *context vector*, or last output of the encoder, as its initial hidden state. An attention decoder, however, allows the decoder RNN to "focus" on a different part of the encoder's outputs for every step of the decoder's own outputs. A set of *attention weights* are calculated and multiplied by the encoder output vectors to create a weighted combination. The result contains information about specific parts of the input sequence, and will help the decoder compute the correct output. These weights are calculated with another feed-forward layer, using the decoder's input and hidden state as its input. (Robertson 2017)

Input for sequence to sequence networks is always in the form of vectors, but they are applied in many natural language problems. For the networks to solve these problems, vector representations of words must be developed. One such way to do this is creating one-hot vectors for words, with every index in the vector representing a different word from the vocab. This approach, however, is not ideal, as it leads to very large and sparse vectors, and will likely not provide a semantic relation from one word to another. Word embeddings provide such relations between words in a vocab, creating a vector space representation of the words. (Tensorflow 2018) There exist various methods to create such word embeddings, such as *GloVe*, that are supervised methods that count statistics within the given corpus, such as word occurrences, word-word co-occurrences, and others. With this information, the model can map the statistics down to smaller and denser vectors than the previously mentioned one-hot vectors. The resulting vector space also captures the semantic meaning of words within the corpus the model was trained on, where semantically similar words are mapped to vectors near each other. (Pennington, Socher, and Manning 2014)

Lastly, the perplexity of a probability model is a metric used to evaluate trained models. In a sense, it is used to determine how "perplexed" a model is with target output after given its associated input. A lower perplexity will indicate that the model performs well at predicting the target output. It is calculated with the equation

$$perplexity = b^{-\frac{1}{N}\sum_{i=1}^{N} log_b p(x_i)}$$

Where p is a proposed probability model, and $(x_1, \ldots, x_N)$ is the test sample.

Experiments

The goal of this project was to develop a sequence to sequence network to train on the corpus of a novel, a small corpus in comparison to other natural language datasets, and attempt to predict sentences given an input sentence as its input. This model was implemented using *PyTorch*, and modified from an existing[2] implementation of translating English sentences to French. Beyond the network, various word embeddings were implemented, both pre-trained and custom ones trained on the corpus used in this project. The use of various embeddings were important for two reasons: to reduce the loss value of the training model, as well as getting a head start on the accuracy of the initial vector space for word representation.

To train the model, a corpus of the first *Harry Potter* novel was used: *Harry Potter and the Sorcerer's Stone*. (Rowling 1998) The novel was parsed into a list of sentences, where the order of the sentences in the list is the order they appear within the novel. A sentence was determined as a series of words, potentially over multiple lines in the novel, and were split with various methods. Firstly, they are split on a sentence terminator: a period, exclamation point, or question mark. Sentences were also potentially split on ellipses, or multiple periods, in a naïve method of if the word following the ellipsis began with a capital letter. Lastly, dialogue, or a word or series of words surrounded by single or double quotes, were also split into their own sentences. Along with this sentence splitting, all text was lowercased and all punctuation was removed, except select punctuation such as exclamation points and question marks, kept for their potential semantic usefulness within the sentence. Contractions were also tokenized out, such as *"would've"* to *"would 've"* or *"Harry's"* to *"Harry 's"*. In this case single quotes were kept for their use with pre-trained word embeddings. Various other parsing methods were implemented that specifically relate to the given text, such as removal of chapter titles, "stuttering" dialogue, and others, in an attempt to create a smaller vocab and more accurate sentence representation of the novel. The accuracy of these parsing methods was never calculated, but the amount of sentences that may be incorrect or contain unwanted text, such as chapter titles or page numbers, is negligible with respect to the rest of the data that is correct. These inconsistencies should not have had much of an effect on the models trained with this data. With this list of parsed sentences, training and testing data was created by creating pairs of all the sentences, where each pair is a sentence with the sentence that follows it. For example, a group of sentences $(s_0, s_1, \ldots, s_{k-1}, s_k)$ would create the pairs $((s_0, s_1), (s_1, s_2), \ldots, (s_{k-2}, s_{k-1}), (s_{k-1}, s_k))$. These pairs allows the sequence to sequence network to train on an input sentence with the known target sentence. The pairs were shuffled randomly and split into 80% training data, and 20% testing data, with the train/test data split saved so variously trained models all trained on the same data for consistency. There was a total of 5,672 training pairs, and 1,148 testing pairs. Finally, a special *"end-of-sentence"* token was appended to each sentence to denote the end of a sentence, needed in the evaluation step.

Sequence to sequence network

The sequence to sequence network was created by creating an encoder RNN and an attention decoder RNN. Both RNNs are multi-layer gated recurrent units (GRU), implemented by *PyTorch*. GRU was chosen, as opposed to another RNN type, such as long short-term memory (LSTM), for its increased performance when training on smaller datasets.

Training

To train the model, an input sentence is given to the encoder, with every output and last hidden state stored. The decoder is then given a special *"start-of-sentence"* token as its first input and the previously stored last hidden state of the encoder as its first hidden state, with subsequent encoder outputs given to the decoder as inputs. Occasionally, for about half the data trained, *teacher forcing* is used. In this case, the real target output is used as input rather than the previous output of the decoder. The loss value for each decoder output is calculated as the negative log likelihood loss between the decoder output, and the target output within the target sequence. The model is trained for a specified amount of epochs, or a complete iteration through the training data, while calculating the loss value at each epoch. The core of this code was modified from the previously mentioned *PyTorch* project, converted to train for epochs rather than iterations, calculating loss per epoch, and displaying validation results. Validation values were calculated by creating a validation set, made up of 10% of the training pairs selected at random, and calculating the loss for each validation pair with the model at its current epoch after freezing learning.

Evaluation

Once trained, the model is then evaluated. This step is similar to the above training step, but without target outputs. The decoder's output is fed back to itself for each step, creating an output sentence until it predicts the *"end-of-sentence"* token (or reaches the max length). Rather than take the top-predicted word for each step, the decoder in this method does a beam search for the top-*k* results. Here, the decoder finds the *k* output sentences with the best score from the given input sentence, returning the best non-empty sentences of the *k* sentences. To evaluate these output sentences, two approaches are used: perplexity and translation metrics.

The perplexity of the model is calculated for each test input by giving the model the input, and then forcing it to

[2] pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html

output the corresponding test output, and measuring its perplexity score of this sentence, rather than a sentence it was to predict.

The translation metrics are calculated by scoring the output predicted sentence against the real target sentence with metrics such as BLEU (Papineni et al. 2002) or METEOR (Lavie and Agarwal 2007).

Word embeddings

The model learns word embeddings naturally while it trains, but their initial values may help or affect how the model trains or how long the training takes. Three types of word embeddings were used within the project for various experiments. First, is a default random embedding, where the vector for each word is random. Next, a pre-trained *GloVe* 300-dimension embedding, trained on a 2014 Wikipedia dump and Gigaword 5, was used. The corpus for this embedding contained 6 billion total tokens, and a vocab of 400,000 words. Lastly, two custom embeddings were created from the entire *Harry Potter* corpus (all 7 novels) using *word2vec* (Mikolov et al. 2013), one with continuous bag of words, the other with stop-gram. Both embeddings were 300-dimensions and trained with a window size of 5 tokens. The theory of the two non-random embedding types was to provide a head start with the accuracy of the vector space for word representation. The custom embeddings, in comparison to the *GloVe* embedding, was developed to hopefully have the words' semantic relations be based off how they are used in *Harry Potter*, which may differ to their use in other texts, as well as include *Harry Potter*-specific words in the vector space. The main issue with these custom embeddings, however, is the size of both the corpus and the vocab. In comparison to the *GloVe* embedding's token count and vocab size, the *Harry Potter* corpus that was created has only 2,253,370 tokens. Of those tokens, only 1,220,874 remained after stopwords were filtered, which was the corpus that was used. Along with the token count, its vocab size had only 11,834 words. This lack of data, either token count or vocab size, most likely hindered this type of embedding.

Implementation

The project was implemented in *Python*, using various 3rd party modules, such as *PyTorch* for the sequence to sequence network and *NLTK* (Loper and Bird 2002) for natural language parsing and evaluation. As discussed before, the implementation of the encoder and decoder RNNs was modified from a similar *PyTorch* project. Perplexity was calculated using values given by the decoder, with the formula of $2^{-\frac{1}{N}\sum_{i=1}^{N} log_2 p}$, where p is the probability of each word given the previous word. The portion of the formula, $log_2 p$, was already calculated by *PyTorch*. The *BLEU* score was provided by a package within *NLTK*, while *METEOR* was a custom implementation[3] based on the algorithm described in its research paper. (Lavie and Agarwal 2007)

[3] github.com/zembrodt/pymeteor

Drawbacks

The project has several current drawbacks, which will be discussed more in-depth below. These drawbacks include incorrectly loading checkpoints, a minimum loss value of around 3, potentially due to training on a small corpus, and high perplexity values when given testing data.

Future work

Beyond correcting the drawbacks listed above, future work could include training and testing a working model on corpora of different types, such as news articles, song lyrics, etc. Training more custom embeddings could also prove beneficial, either training them for much longer than the experiments done for this project, or using *GloVe* to train custom word embeddings rather than *word2vec*.

Experiments

The experiments conducted for the above described model can be broken down into several sections as various concepts were implemented or tested: default implementation, beam search, and custom embeddings. Since these were implemented in various stages of the project, the code used to parse the natural language in the novel may not be consistent through all experiments, but the results from each experiment should not be greatly affected by this. Along with this, several evaluation metrics were not correctly implemented until later stages in the project, such as calculating perplexity or tracking loss values while training.

Default Implementation

The default implementation is close to the initial code used from the *PyTorch* sequence to sequence network. Beam search had not yet been implemented, only evaluating with the top-1 result, and random embeddings were used. Unfortunately, at this point in time, perplexity was not being calculated correctly, and would not be fixed until beam search was implemented. At this point, however, we have results from the translation studies.

Epochs	*BLEU* score	*METEOR* score
10	0.0	0.0157
40	0.0	0.0389

Table 1: Default implementation *BLEU* and *METEOR* scores

The initial results in *Table 1* had poor analysis scores, both at 0 or near 0. For both *BLEU* and *METEOR*, scores range from 0 to 1, with 1 meaning an exact match. However, the sentences they predicted were promising at the time.

Beam search

At this point, the evaluation of the trained model was updated to include beam searching of the top-*k* results, rather than the default top-1. The implementation of this was to check if the best scoring output sentence was perhaps not found by taking the top-1 output at each stage of the decoder,

but somewhere else. This model was once again trained for 40 epochs.

Epochs	*BLEU* score	*METEOR* score
40	0.0	0.0346

Table 2: Beam search *BLEU* and *METEOR* scores

This model showed no real improvement with the translation metric scores, as shown in *Table 2*, but the first perplexity study was also done during this stage.

Epochs	*Actual sentences*	*Random words*
25	85241.0183	306539.759

Table 3: Perplexity scores for *"Actual sentences"* and *"Random words"*

While the perplexity scores in *Table 3* were not very good, particularly due to the small amount of epochs the model was trained for, the *"actual sentences"* score being fairly smaller than the *"random words"* score was promising. Here, *"actual sentences"* refers to the score of given an input sentence, the perplexity of the model forcing it to evaluate to the real target sentence. *"Random words"*, on the other hand, were given an input sentence, the perplexity of the model forcing it to evaluate a random target sentence of the same length as the real target sentence, but where each word was replaced by a random word selected from the corpus. In future methods, explained below, this perplexity study will be split between training data and testing data, with other metrics used as well.

Custom Word Embeddings

The bulk of experiments were conducted with custom word embeddings. At this point in the project, models started to be trained for a larger amount of epochs, ranging from 100 to 500, and the under-fitting of the model to the data became apparent. The value of loss while training was also calculated and recorded for comparison and analysis of each model. It was also determined that translation metrics such as *BLEU* and *METEOR* may not be ideal to measure the effectiveness of the model, at least in its current state, and were no longer calculated.

Figure 1 shows an initial training of the model using the *GloVe* word embedding discussed previously, trained to 100 epochs. While its final loss value of around 6.72 is not an ideal value, this model seemed to still be learning, and the loss decreasing, when it finished training at 100 epochs. Nevertheless, it was decided to train two custom word embeddings with *word2vec* using the entire *Harry Potter* corpus, as discussed previously. *Figures 2* and *3* show results for both stop-gram (SG) and continuous bag of words (CBOW) implementations.

At this point, the SG implementation seemed the most promising, having finished training for 100 epochs with a loss value of 4.807. It was decided then to do a full study on all word embedding implementations for a full comparison between them, and for a larger amount of epochs.

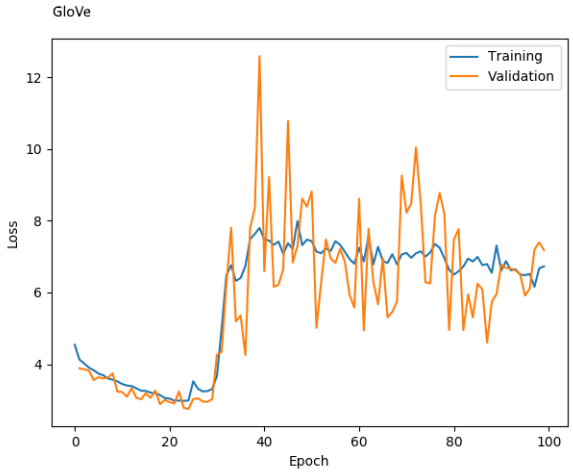

Figure 1: Initial training with *GloVe*

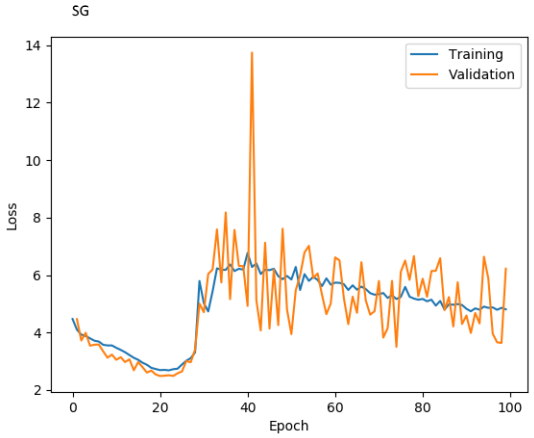

Figure 2: Initial training with *word2vec* SG

Figure 4 shows a comparison between random (called *none*), *GloVe*, *word2vec* with SG, and *word2vec* with CBOW embeddings. The result shows that out of all the models tested, only *GloVe* had any improvement, with a decreasing loss value after 500 epochs. However, it became apparent that the project developed was incorrectly saving and loading models, causing a large spike in the loss value when a checkpoint was loaded (see emphasis of *GloVe* value in *Figure 4*).

After conducting this comparison, however, it was determined that the custom *word2vec* embeddings should be trained on the *Harry Potter* corpus for longer. At this point, both were trained for 15 epochs over the corpus, taking a little over one minute. CBOW also seemed to be the better implementation to move forward with, rather than SG.

Figure 5 shows a comparison of two models trained for 500 epochs. One with the 15-epoch CBOW embedding, the other with a 300-epoch CBOW embedding. This result was once again corrupted by the loss spiking of loading a check-

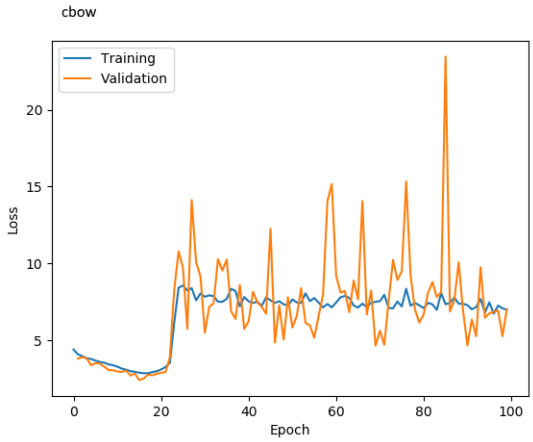

Figure 3: Initial training with *word2vec* CBOW

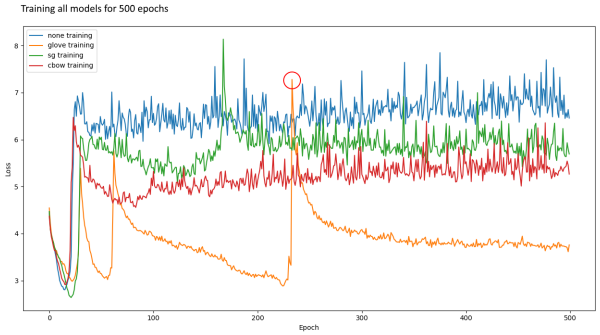

Figure 4: Comparing all embeddings

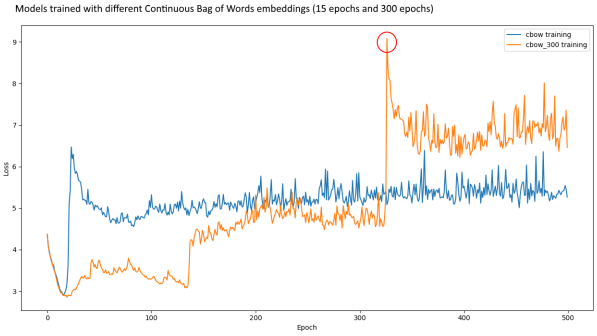

Figure 5: Comparing different *word2vec* CBOW embeddings

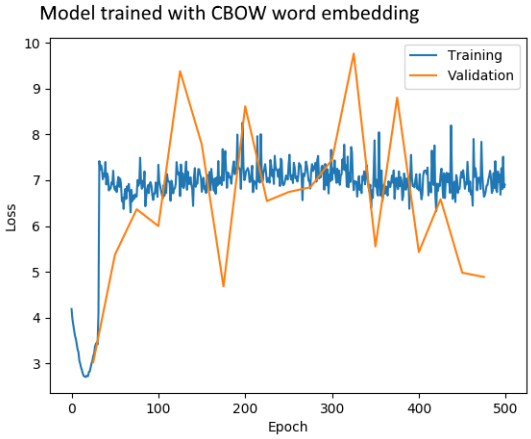

Figure 6: Initial training with *GloVe*

point (emphasized) for the 300-epoch CBOW embedding. However, before the checkpoint, the 300-epoch CBOW embedding was performing better, so it could be argued that training the word embedding for a larger amount of epochs positively affected the model's loss value while training.

Now it was determined to do a comparison of the custom CBOW embedding with the *GloVe* embedding, for 500 epochs, without any checkpoint loading, as this issue still hadn't been corrected.

Figures 6 and *7* compare these two embeddings with one another. Training with the CBOW embedding was not beneficial for the model's training process, and overall was not decreasing the loss value. *GloVe*, however, performed well. Without having to checkpoint, the loss value never spiked (after the initial spike), reached a minimum value of 2.983, and a final value of 3.083 at the 500th epoch. This was a much better epoch score than the 4.0 to 7.0 and greater values calculated earlier. Perplexity studies were once again conducted on this model, giving even more interesting results.

In *Table 4*, a further perplexity study was developed, for *"random sentences"*. For this study, the model was given the input sentence, and forced to evaluate a target sentence of the same length as the real target sentence, but randomly chosen from the data. The study was also split between training and testing data, where training data should outperform testing data, as it is data the model has seen, but with the goal of the testing data scoring close to the training data scores. Out of the three categories of studies, *"actual sentences"* should perform the best, as they are the real sentences to follow the input sentences. This will be followed by *"random sentences"*, as they are full real sentences, meaning they have a real sentence structure, but may have nothing to do with the input. Finally, *"random words"* should perform the worse, as they will likely not have a real sentence structure. *Table 4* shows that this is the case: training data usually outperforms testing data, *"actual sentences"* outperforms *"random sentences"*, which then outperformed *"random words"*. The 250th epoch model is taken from the minimum point in *Figure 7*, with an *"actual sentences"* value of 42.989, this perplexity value was actually reduced to 37.041 for the 500th epoch of the same model. In fact, all training data perplexity values between the two models was decreased. However, the testing perplexity values all increased. For the extra 250 epochs the

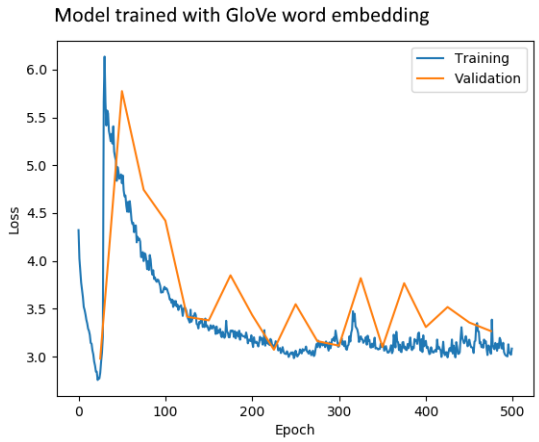

Figure 7: Initial training with *GloVe*

Epochs		Training data	Testing data
250	*Actual sentences*	42.9893	736176.1818
	Random words	146455354.9	147674868.5
	Random sentences	295889.4025	299692.5997
500	*Actual sentences*	37.0408	4872462.827
	Random words	1270328937	6201033178
	Random sentences	27962670.27	2313469.597

Table 4: Perplexity scores for model in *Figure 7*

model trained for, it seemed to continue learning, but fitted too exactly to the training data, and became more "perplexed" by data it had not seen before. Finally, the actual perplexity themselves were extremely high for the testing data, further showing that perhaps the model hasn't learned enough to handle unseen data. One cause of this could be due to the lack for training data, with just over 5,000 sentence pairs.

Conclusion

The project discussed had the end-goal of generating stories by training a sequence to sequence network on the first *Harry Potter* novel as its corpus. A large portion of the project, however, became a study of various word embeddings, and how they affect the training of the model. Both topics were something I was unfamiliar with at the start of the project, as well as the various tools used, such as *PyTorch*, *word2vec*, *NLTK*, and others. This was also the largest *Python* project I've undertaken, with roughly 2,800 lines of code, including comments and print statements. Due to this, I believe this project has also tremendously helped my skills in various related areas: *Python* programming, working with *Git*, writing in *markdown*, and reading and understanding documentation. While the end result and current status of the project isn't necessarily complete or a working story generator, I believe it to be a good start, and has immensely increased my knowledge on its related topics of RNNs, sequence to sequence networks, word embeddings, and how to leverage these tools in a language such as *Python*. I hope in the future to be able to further explore this project, or work in similar areas to take advantage of what I have learned.

References

Lavie, A., and Agarwal, A. 2007. Meteor: An automatic metric for mt evaluation with high levels of correlation with human judgments. In *Proceedings of the Second Workshop on Statistical Machine Translation*, StatMT '07, 228–231. Stroudsburg, PA, USA: Association for Computational Linguistics.

Loper, E., and Bird, S. 2002. Nltk: The natural language toolkit. In *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics - Volume 1*, ETMTNLP '02, 63–70. Stroudsburg, PA, USA: Association for Computational Linguistics.

Mikolov, T.; Sutskever, I.; Chen, K.; Corrado, G. S.; and Dean, J. 2013. Distributed representations of words and phrases and their compositionality. In Burges, C. J. C.; Bottou, L.; Welling, M.; Ghahramani, Z.; and Weinberger, K. Q., eds., *Advances in Neural Information Processing Systems 26*. Curran Associates, Inc. 3111–3119.

Olah, C. 2015. Understanding lstm networks. *Github blog*. http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

Papineni, K.; Roukos, S.; Ward, T.; and Zhu, W.-J. 2002. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, 311–318. Stroudsburg, PA, USA: Association for Computational Linguistics.

Pennington, J.; Socher, R.; and Manning, C. D. 2014. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, 1532–1543.

Robertson, S. 2017. Translation with a sequence to sequence network and attention. *PyTorch*. https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html.

Rowling, J. K. 1998. *Harry Potter And the Sorcerer's Stone*. Arthur A. Levine Books.

Sutskever, I.; Vinyals, O.; and Le, Q. V. 2014. Sequence to sequence learning with neural networks. In Ghahramani, Z.; Welling, M.; Cortes, C.; Lawrence, N. D.; and Weinberger, K. Q., eds., *Advances in Neural Information Processing Systems 27*. Curran Associates, Inc. 3104–3112.

Tensorflow. 2018. Vector representations of words. *Tensorflow*. https://tensorflow.org/tutorials/representation/word2vec.