Analyzing how machines learn translation using a Word2Vec and RNN model

Statistical Machine Translations (SMT) are a form of machine translation that attempt to perform translations using a probability distribution, as opposed to other models that use rules or examples. This paper goes into detail about how two different forms of SMTs learn to translate between two languages. The two methods chosen were the Word2Vec model and recurrent neural networks. I will first go into detail about how the two models work and how we can use them for translations. I will then cover how I implemented the translation models, and finally I will show how the models behaved on text files that were in English and French with various parameters being tweaked and the strengths and weaknesses of both models.

Word2Vec

Word2Vec is a collection of deep learning models released by Google that attempts to create vector representations of words. Word2Vec takes a large corpus of text and generates a model from it using Skip-gram models. These methods attempt to generate a model that predicts the neighbors of a given word. The generated word vectors hold context and semantic information. This allows the model to create relationships using simple vector relationships.

 Consider a trained model that generates a vector for “King” (Vk), “Man” (Vm), and “Woman” (Vw). We can perform a vector operation and get the resulting vector for “Queen”.

This is a Word2Vec model trained on text from the NLTK corpus(“Paradise Lost”, “Alice In Wonderland”, “The Bible”, and “Pride And Prejudice”). Note how words related to the body have clustered together in the top right corner.

This shows that the Word2Vec is able to learn about the underlying relationships between the words.

The association is accomplished by the model going through the text and taking a set amount of words(“window” size) that appear around the current word as a vector which is fed into a neural network that attempts to learn the probabilities of which words are going to appear next given a certain word. The objective is to maximize the average log probability of correctly predicting the next word given a word for all words.

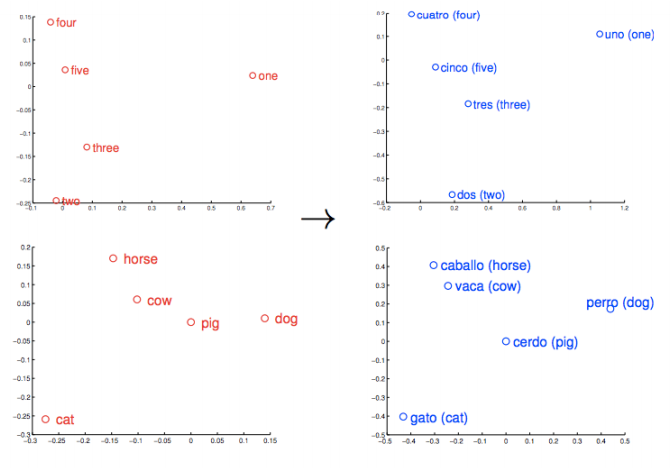
Example: “… I like eating pizza and cake. …”

The model would try to find the sum of the probabilities of the 3 words before and after “I” under the condition of “I”’s existence. This would then be repeated for “Like” and so on.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| I | Like | Eating | Pizza | And | cake |

For all words in the vocabulary(T), we calculate the probability of the next word() by summing the probabilities of the words –K to K distance from the word we are currently looking at. Performing this on the entire vocabulary gives us a model where words like “food” and “pizza” will be close to each other. We can also increase the window size parameter(K) to increase the associations that each word will have at the cost of increasing the learning time for the model.

Word2Vec translation

Words in different languages will obviously look different and appear in different parts of the sentence. But what will not change is the meaning of the word/phrase. “I like cats.” means the same thing in any language you use it. This property can be exploited in Word2Vec to translate words from one language to other languages. Consider the scatter plot of words above, if I used French novels to create the Word2Vec model instead of the English novels, I would still expect the same clustering behavior, the words wouldn’t be in the exact same location to each other, but their distances to other words and clustering behavior would still be preserved.

This example shows how similar words in different languages will be in the same region in a projection.

We can now perform the translation by creating a translation matrix that takes vectors from one space and converts them to the other space. We then find the vectors closest to the newly created vector in the translation space using the cosine distance, giving us a list of possible candidates of words that could be the same as the original word. This matrix can be found by solving the following optimization problem.

W is the matrix we are searching for and Sk and Tk are pairs of words in two different languages. Once this matrix is learned, translation is done by performing a linear transformation taking a vector in the original language and multiplying it by the W matrix, to get a vector that is close to the representation of that word in the target language,

RNNs

Consider the sentence “I like cats.” again. If a model wants to translate the whole sentence, when it looks at the word cat, it needs to know about the previous words in the sentence as well. This requires a model that can remember information from the past. A Recurrent Neural Network is exactly what we need for this task. Unlike a convolutional neural network whose output is only dependent on the input and hidden layer configuration, an RNN is created to hold information about past events as well, which allows it to learn from sequences of vectors as opposed to just a single vector. A basic RNN has 3 weight matrices(Wh, Wx, Wy) and a hidden state vector(h). When the RNN gets an input vector(v) the following calculation is performed to update h.

We then calculate the output vector(o) using the hidden state and output weight matrix.

You can see from the previous computation that the value of h is constantly being updated based on the value of the incoming vector. This property allows future outputs from the RNN to be influenced by events in the past. Another helpful property of RNNs is their ability to translate one variable length sequence to another variable length sequence, this can prove very helpful in translation systems because some words might exist in one language and not in another.

RNN translation

Performing translations with RNNs is much more intricate and involved than the word2vec method. The RNN model that I decided to use is an “Encoder-Decoder” model which allows translation of variable length sequences to other variable length sequences. This is done by using 2 RNNs, an *encoder* and a *decoder* RNN. The encoder is fed a variable length sequence which it then converts into a fixed length vector. We also do not care about the output of the encoder, what we are actually looking for are the internal states of the encoder (which will be the fixed length vector). This fixed length vector is then fed into the decoder RNN, which results in a variable length vector that is the output that we want.

If we want to work with encoder-decoder models, we need to first modify our data to indicate START and STOP symbols to designate the start and stop points of a sequence. Once an RNN reads a START code, it will continually update its inner state H until it hits the STOP code.

The decoder takes the encoder state info, and along with its current internal state value, makes a prediction as to what it believes the next word will be. Once it has predicted the next word, it uses that word as input(along with the state info) to try to predict the next word.

This is a simple encoder-decoder model. Note how the output of the decoder is also fed back into the decoder to get the next word.

[START] I Like Cats. [STOP] [START] J’aime …

Encoder

Decoder

Encoder State Info

Let’s look at the encoder in more detail. The sentence input of the encoder is going to be a sequence of 1 hot vectors. The encoder will linearly project the incoming vector with a matrix E which will have as many columns as there are words in the vocabulary and the rows are up to the designer. The resulting projection is then used to update the internal state of the encoder H, which initially starts as an all-zero vector. The value of H after the STOP symbol is detected encodes and summarizes all the information about the sentence we have. We can then use H to send into the decoder to configure it to translate the sentence.

The decoder effectively reverses the work done by the encoder, but generates the output in a different language. We take the incoming H vector and calculate the internal state using a function like . Once we have calculated the internal state, we calculate a score for each word in our vocabulary.

The *bk* value is a bias that we don’t need to worry about. Wk is a target word that we want to score. The score of each word depends on how well it aligns (is parallel to) with the internal state, generating a higher value the more it aligns.

Once we have the scores for all the words, we can then compute the probability of each word being a likely translation candidate by using softmax.

We now have a vector of the probabilities of each word, all we need is to choose the highest probability and that is the best translation the RNN could generate.

Now all we need to do is train the system so it can actually translate from one language to another. We first need the proper data. Due to the fact that we are doing direct translations (unlike Word2Vec where we are trying to translate based on context), we will need a parallel corpus that has text phrases in a source and the exact same phrase in the target language. Once we have a parallel corpus, we convert each of the sentences to one-hot vectors and given any pair of sentences from the corpus, we can compute the log probability of sentence Y given sentence X, . is the set of parameters that defines the model. The log-likelihood of the entire corpus is

Where N is the size of the vocabulary. We can train the RNN to maximize this function using SGD and backpropagation.

Implementation

I wanted to test the learning capabilities of both of these methods, so I think a simple way of testing this would be to split the test into testing a couple parameters of both models to see which would “converge” faster and make a translation. The corpus I used for the Word2Vec translations was a combination of the top downloaded novels on Project Gutenberg for the source text, and Alice In Wonderland for the target text. When testing the RNN, I had to find a parallel corpus so I used the downloadable study cards that Anki provides. I first tested the impact the size of the corpus had on both models, and kept using bigger sizes to see what was happening to the translation quality of the words. I also decided to test how the window size for the Word2Vec model affected translation quality. I then tested how long it would take, given a corpus with a fixed size, to finally be able to translate a word, this was done by having the model train for more epochs until translations were being made. I conducted these tests by creating a translation program using Word2Vec and another translation program using RNNs created from the Keras library. I then ran the tests above by changing the parameters of each translator and making it translate a couple words.

For the Word2Vec translator, the program takes two files in different languages, tokenizes them, and uses the Gensim library to run the Word2Vec model on the sentences. The program creates 2 Word2Vec models in both languages specified and then generates a translation matrix. I tried to create my own translation matrix but just couldn’t get the math right, so instead I used the translation matrix provided by Gensim. The program then creates a translation matrix to translate from the source language to the target language. Prior to running any translations, one has to pass in pairs of words to allow the matrix to make the proper associations in the Word2Vec model. I then begin running tests which modify the parameters of the model. The output of the whole program are python dictionaries which show a word in the source language and words that the model think are translations for the source word in the target language.

The RNN translator follows the encoder-decoder model to create the translator and I used the Keras library to implement the RNN. An encoder RNN was created that took a set of input sequences and output a state vector. A decoder RNN was also created that took the state vector and attempted to create a prediction. I trained the model on a parallel corpus whose parameters I modified so I could analyze how the translations were being affected. The model was trained by first taking all the source language vectors and feeding them into the encoder, this generated a state vector. A copy of the language vectors was made which was shifted to the right by one and fed into the decoder. Because of this shift, when the decoder reads a state vector, it tries to predict the next element in the sequence, and continues doing so until it read a STOP statement.

Testing

I began testing the Word2Vec implementation by decreasing the size of the generated word vector only for the french vector, I kept the size of the english vector constant so that I could keep adding words in the future if I needed to and not worry about the word not being found. I set the size of the french vector to 1000, 2000, 3000, and 4000. I expected that as the size of the french vector would increase, the associations the words would make would also increase and that is what happened. Some of the words did not translate correctly but the associations that those words make were getting closer to the meaning of those words. The word “Think” was never able to converge, I checked to see how frequent it appeared in the target text and found it rarely appeared, so the model didn’t really have a chance to learn it’s associations. I found that the more frequent the word appeared, the quicker it’s translation was obtained, this was apparent with the word pair (“She”, “Elle”) which was able to translate properly on the first corpus size. A less frequent word like the pair(“good”, “bon”) translated properly after the third test.

|  |  |  |
| --- | --- | --- |
| Vector Size | Words missed | Words correct |
| 1000 | 6 | 4 |
| 2000 | 5 | 5 |
| 3000 | 2 | 8 |
| 4000 | 2 | 8 |

I then checked to see how the window size modified the associations being made. I kept the size of the vectors the same but increased the window size. I changed the window size to 2,4,6, and 8 and found that it did not have that much of an impact on the quality of the translations. All the words were properly translated on the first test. I thought this meant that the window size is actually linked to the vector size, because a greater vector size would mean many more associations were encountered, so I ran a third set of tests that set the window size to be small and set the length of the french vector to be 1000 and 2000.

I then tested the RNN implementation of the translator. The first test I performed was variating the number of epochs that the model spends learning. I ran the training portion of the program through a loop that had the model train for 40 epochs and then try to translate a set of words from the corpus. I repeated this a set number of times and analyzed the generated translations to see if any translations were being made.

The first test I ran trained 40 epochs 10 times, the second test ran 40 epochs 100 times. The first test was not able to translate any words because it ran for too short a time. The second test showed improvements as words like “fire” and “I” were being translated properly. I was still unable to get an actual phrase to translate completely. I then decided to increase the number of samples I was testing on from 100 to 1000 and found more words were being translated, like “Wait”, “hop”, “I”, “Be calm” and many others. I also began seeing phrases being translated, but many were still incorrect.

Results

Once the tests were done, I found that Word2Vec was much more effective at generating translations than the RNN. The Word2Vec model was much quicker at learning the text allowing me to quickly generate and test translations. Even when I tested the model using the smallest vector size for Word2Vec, I was still able to generate a couple translations for words. The size of the French corpus used was also smaller than the RNN’s French corpus. I believe the reason the Word2Vec model was much more effective was because I used lots of text that had context and meaning. This allowed the model to create stronger connections between word vectors making translations easier.

I also found that other RNN models were trained on massive parallel corpora, comprising millions of lines (several GB), and trained for days on GPU clusters. The corpus I used only had 145k lines, causing the phrase translation to be mediocre. However, I would still choose the Word2Vec model because I was able to create a fairly competent translator with it relatively easily in a few days. Creating the RNN took much longer and required me to consult lots of online resources to find examples that would help me get started. One place the Word2Vec model fails at is translating phrases. We are only able to translate single words which is a severe limitation for a translation program.

Conclusions

I didn’t know anything about NLP and RNNs when I started on this project, it was really interesting finding resources on Word2Vec and RNNs. I initially started out reading papers on Word2Vec and playing with it in DL4J, but I soon found DL4J to be too slow when I wanted to make a small change, so I switched over to Python. Once I was comfortable with Word2Vec, I started reading up on RNNs and began messing around with the tutorials Keras had on RNNs. Learning RNNs was much more complex, even with the simplicity of Keras, I was still getting lots of errors on mismatched dimensions. I eventually found tutorials that helped me create a translator. Once I had two simple translators, I began testing and seeing which model was more effective at learning. At the end I decided on Word2Vec because of how rapidly I was able to learn and use it. But if I had to use a translation system professionally, I would probably spend more time tuning an RNN because of its ability to generate phrase based translations. At the end, I am glad I was able to learn so much about NLP and machine learning. I don’t think I would have been able to learn about these topics if I didn’t have a reason to complete this project.

# Bibliography

Andrew Lamb, M. X. (n.d.). Convolutional Encoders for Neural Machine Translation. *CoRR*.

Britz, D. (2015, September 17). *Recurrent Neural Networks Tutorial, Part 1 – Introduction to RNNs.* Retrieved from WildML: http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/

Cho, K. (2015, May 25). *Introduction to Neural Machine Translation with GPUs.* Retrieved from Nvidia.

Greenstein, E., & Penner, D. (n.d.). Japanese-to-English Machine Translation Using Recurrent Neural Networks.

Karpathy, A. (2015, May 15). *The Unreasonable Effectiveness of Recurrent Neural Networks.* Retrieved from Andrej Karpathy Blog.

Kyunghyun Cho, B. v. (2014). Learning Phrase Representations using RNN Encoder–Decoder for statistical machine translation. *CoRR*.

Lie, S., Yang, N., Li, M., & Zhou, M. (2014). A Recursive Recurrent Neural Network for Statistical Machine Translation.

Mikolov, T., Le, Q. V., & Sutskever, I. (2013). Exploiting Similarities among Languages for Machine Translation. *CoRR*.

Neubig, G. (2017). Neural Machine Translation and Sequence-to-sequence Models. *CoRR*.

Vargas, E. (2017, march 13). *Recurrent Neural Networks for Language Translation*. Retrieved from Medium.

Xiaohong, J. (2017, september 26). *Translation Matrix Tutorial*. Retrieved from github: https://github.com/RaRe-Technologies/gensim/blob/develop/docs/notebooks/translation\_matrix.ipynb

Xiaohong, J. (2017, September 13). *Translation Matrix: how to connect “embeddings” in different languages?* Retrieved from Rare Technologies.