RNNs for Machine Translation

Nikita Traynin
University of Texas at Dallas
CS 4375.001 Intro. to Machine Learning
Richardson, TX, USA
Nxt170002@utdallas.edu

Abstract – I researched the application of RNN's (recurrent neural networks) to the task of machine translation. RNN's and various subtypes of the general RNN structure are by far the most popular neural net chosen for many NLP tasks, especially machine translation. In addition to the standard RNN node, there exist modern alternatives such as GRU's (gated recurrent units) and LSTM's (long-short term memory cells) which perform quite well. These cells can be aligned in a single direction or bidirectionally, and the manner in which they are applied further varies with each use-case. In this paper, I outline my research for several of the most popular models, and apply an RNN encoder/decoder to the translation task of translating the Bible from English to Spanish, and by extension, text written in King James Version bible-style English into text written in the NBV translation bible-style Spanish.

I. INTRODUCTION AND BACKGROUND WORK

The task I selected came from the predetermined list of available tasks. I was particularly interested in translation because I had no experience in RNNs or NLP, so I would learn a lot. Also, the ancient practicality of translation, combined with the modern innovativeness of neural networks, intrigued me. When I started researching RNNs, I quickly learned about the size of the field, and how RNN expertise is a broad umbrella, and it usually contains one if not several deeply connection layers in the process. My instinct was to start off with the most basic RNN structure to see how it goes.

Next, I needed to gather a large enough dataset. Not surprisingly, the Bible is the perfect such text. This is because it is already somewhat pre-processed. It is already labeled, in both languages, verse-by-verse, and book-by-book. Furthermore, the Bible is a very large set of consistently styled data. The very largest sets of data used for tasks like these have tens of millions of words, if not over 100 million (!!), but such data sets are reserved for complex models trained on custom GPUs. The Bible comes quite close at about 790,000 words for the KJV version and about 720,000 words for the NBV version [1].

My code is written in Python only, with three Python files and a README. The first Python file processes the data. It web scrapes both bibles, parses the XML, and creates the dataframes, word lists, and vocab sets for both bibles. The second Python file contains some helpful functions used in the algorithm, and the final file contains the feedforward and backpropagation algorithm.

II. THEORETICAL STUDY

In this section, I will go over three models, in increasing complexity. The first is the basic RNN concept. A recurrent neural network is simply a neural network in which a layer takes in new input as well as it's own previous output.

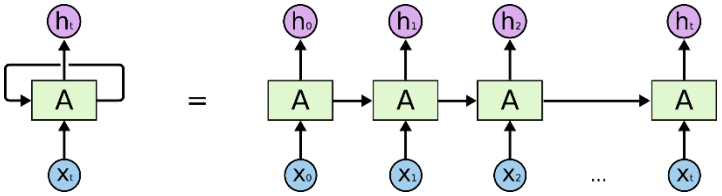

Figure 1: The compact graph on the left and unwrapped graph on the right.

In standard fully connected networks, a layer only has two terms to compute: its weight matrix times the input vector, plus the bias vector for that layer. Now, we have a third element – a second weight matrix times the output of the layer in the previous iteration [2].

The reason the structure is created in this manner is that it now understands sequential data. Analogously, CNN's perform better on visual data because they contain convolutional layers with filtesr, so there is a sense of "locality" – different areas of the same images contain different objects, and object boundaries and textures form certain shapes across the set of data. Every pixel has a different function in the image. Similarly, RNN's understand time series data [2]. In our use case, some words come before others in a sentence. The order of the words matters a great deal. Furthermore, there is the aspect of memory – words in the same sentence rely on other words in that sentence. Another interesting use case is predicting stock market behavior. This use case is even simpler, since the stock market is a Markov process – a stock's current price can only depend on its past prices, not on its future prices. This is why I consider this a "simpler" problem (take this with a grain of salt – in reality, translation is much more accurate than stock price predictions). The

translation of a certain word in a sentence depends on its context – both in future and past words (think of how Yoda talks). This is actually a pitfall of my model

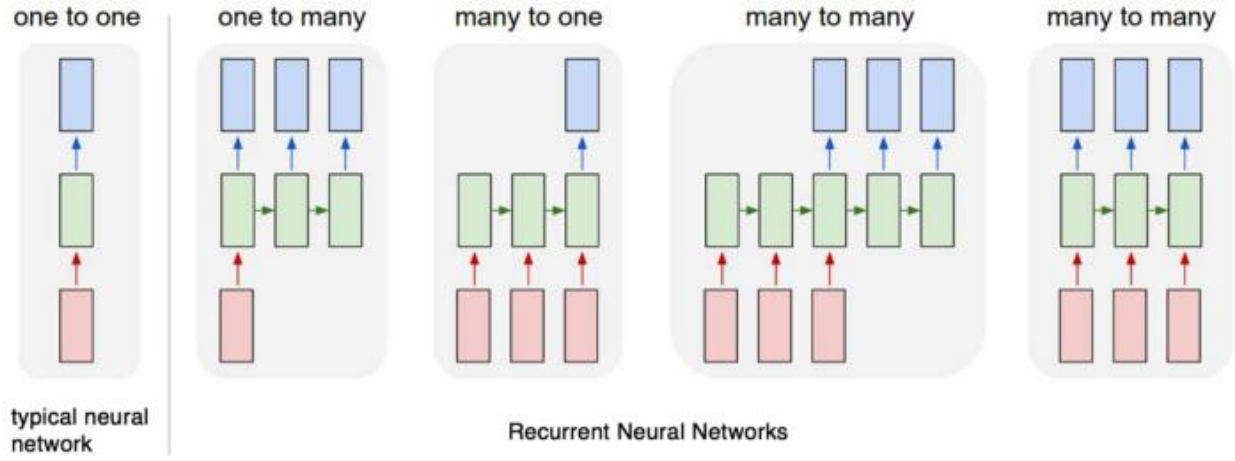

Figure 2

– it only relies on past words. One way to overcome this is to have a bidirectional RNN, but this is outside the scope of this project.

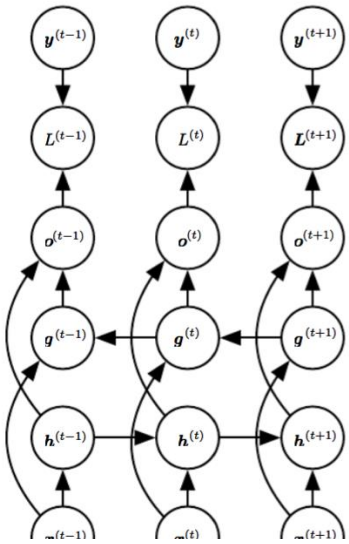

Figure 3: A bidirectional RNN

Now, I will discuss the encoder/decoder structure. An important nuance to consider is that the input and output of this model might be different lengths. That is, the input sentence might be a different number of words than the output sentence. And, every input/output pair has a different length statistic than others. So, we cannot "hard-code" these input sizes (Fig.2). Thus, we separate our task into processing the entire English verse (I've mentioned sentences, but in reality, our batches will be verses. For the most part, bible verses are a single sentence, but some may have two or three), and then decoding it with two separate, slightly modified RNNs. Here is a diagram of the structure, which is much more descriptive than any words and equations:

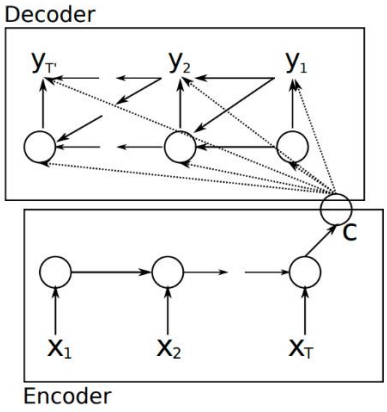

Figure 4

The encoder does not produce outputs, but just goes from one hidden layer to the next, until the final hidden layer executes and produces an activation value vector. This vector is considered the output of our encoder and is the "context" vector. This vector is then passed to every layer of the decoder. So, each

layer of the decoder has three inputs, rather than two. They are the output from the previous hidden layer, the context vector, and the previous hidden layer activation (and also an optional bias vector) [2][4].

Another consideration is – how do we represent our input? Our input is a verse, which is an ordered list of words. Each word is a string, but how do we operate mathematically on a string? We must convert this word to a numerical form, which is a daunting task on its own. This is called word embedding and is a universal task in all of NLP. The simplest and most effective way to store words, tried and true, is vectors. We first "one-hot-encode" the words – meaning each word gets a unique vector. This vector is all zeros except for a single index where there is a one. This a simple one-to-one representation of the words. There are several problems. This vector is sparse, and thus wastes space. It also does not contain any information about the similarity of the words, so we cannot make any comparisons. This will be problematic for model training. The solution is to "embed" this vector.

Since one-hot-encoding is one-to-one, the number of dimensions is the same as the number of possible words there are in the text, a.k.a. the size of the vocabulary. In our case, this means our one-hot vectors sit in 13,000-dimensional space, since there are about 13,000 unique vectors in the English bible (even more in the Spanish translation!). This is unnecessarily large for storing the basic meaning of words, so we should embed these vectors in a smaller feature space. We can manually set the embedding dimensions and play with the number to see what is right. A common number is 100 or 500, but we will start with 50 since we do not own a supercomputer [4]. There exist pre-trained word embedding models such as word2vec and GloVe, but we can instead use an embedding layer in our RNN that will train itself as the model iterates [3].

Lastly, there is an alternative structure that ameliorates some of the pitfalls of our RNN cell encoder/decoder. I present the LSTM – the long-short term memory cell. These cells have several additional components. In brief, they are the forget gate, the external input gate, and the output gate. The forget gate weights the context vector – in other words, it determines how much each part of the context is considered in the output. The next gate is the external input gate, which determines how much weight is assigned to the external input. Our external input is the previous hidden layer as well as the input vector x_t. Our weighted context and weighted external input is summed to create the new context, and is also passed to the output gate, which weights the result and outputs it out of the cell.

Clearly, LSTMs are by far the most complicated model. Feedforwarding and Back-propagating this model from scratch would take ages and training said model on a data set as big as the Bible would take eons. For this reason, I am sticking to the RNN encoder/decoder structure. This structure actually performs quite well, given a large enough size and properly tuned parameters [4].

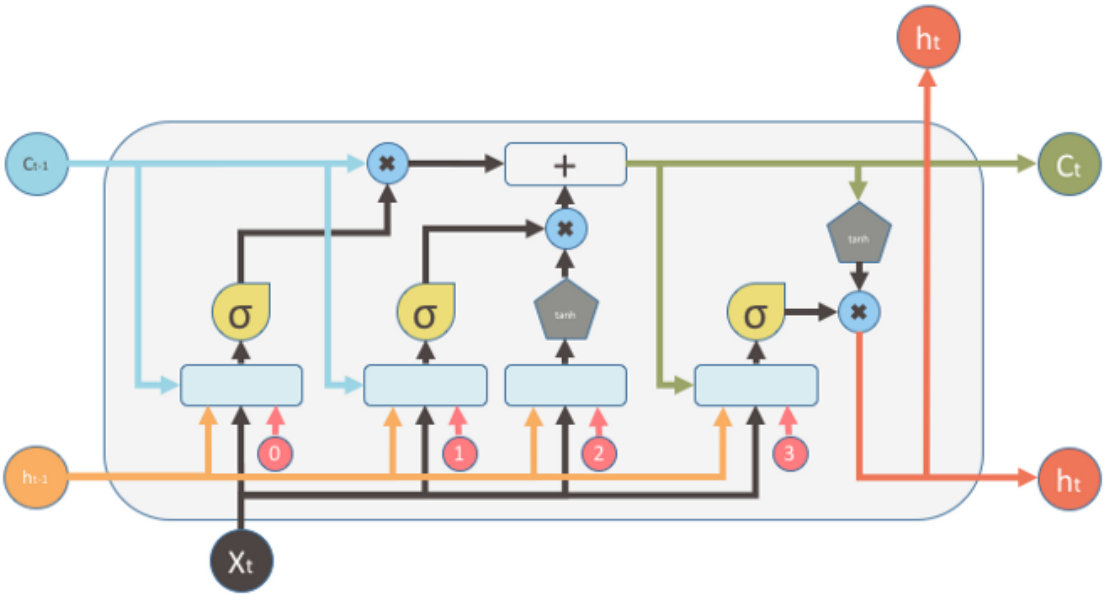

Figure 5: An LSTM cell

REFERENCES

[1] http://christos-c.com/bible/
[2] https://medium.com/@jianqiangma/all-about-recurrent-neural-networks-9e5ae2936f6e
[3] https://towardsdatascience.com/language-translation-with-rnns-d84d43b40571
[4] https://arxiv.org/pdf/1406.1078.pdf?source=post_page----------

Figure 6: A Vanilla RNN. Compact form on the left and unwrapped form is on the rig