RNN/LSTM For Natural Language Generation of News Headlines (ON.ION)

Drew Bodmer and Santiago Pinzon

Association for the Advancement of Artificial Intelligence
pinzon.s@husky.neu.edu & bodmer.d@husky.neu.edu

Abstract

In this paper we set out to design and train a neural network capable of generating fake news headlines given a seed word or phrase that was comparable to a model built in Keras. We also examined the differences in language structure produced by two unique datasets. In order to accomplish this we created an LSTM network and trained it using headlines from the News Aggregator Data Set, as well as a dataset called OnionOrNot. We also implemented a variation on this network using an RNN we wrote ourselves. The Keras network was used to compare results between our two datasets and served as a base point to which we could compare our own network. In addition to comparing our own implementation to the Keras one, we also compared the performance of two types of models, one character-based and one word-based. Unfortunately our network never surpassed the prewritten network, but it stands as proof that what we wanted to achieve is very much possible, given enough time and resources.

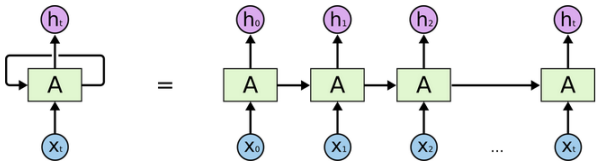
Introduction

This project was inspired by a conversation about The Onion, a satirical news publication that produces realistic sounding news articles. In the case of The Onion, these articles are written by human beings, so it would be cool to be able to put them out of business with an effective model. We set out to see if it would be possible to create a neural network capable of generating headlines that had no basis in reality. After doing some initial reading, and finding papers which had covered similar topics, such as this paper by Ashish Dandekar[1], we decided on using a Recurrent Neural Network (RNN) in order to accomplish this. We felt that because traditional neural networks generally do not work well on sequential data, we needed a more sophisticated model. We considered other options such as Markov Chains, but from reading literature it was clear that the best results would be obtained from RNNs, specifically Long Short Term Memory networks (LSTMs). Initially, we designed a network using Keras, which is included in Google's TensorFlow library. We felt that using a pre-existing framework would allow us to establish a baseline in terms of performance we could aim for. Once we had trained this network and examined some of its performance, we proceeded to attempt to write our own network. Due to limited resources, training our networks was a very slow process, sometimes requiring upwards of 2-3 hours to train just 10 epochs. The homebrew network was even slower, due to the lack of optimizations that were present in the library network. One change we made to speed up training was implementing mini-batch stochastic gradient descent in our backpropagation function, which would calculate the gradients for several examples before updating the weights for the model. We also used Truncated Backpropagation Through Time (TBTT) to reduce computations during the weight updates.

Background

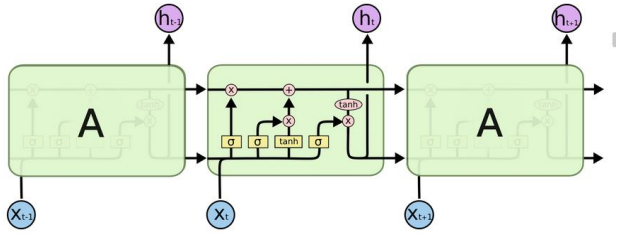
Unlike traditional neural networks, RNNs (Recurrent Neural Networks) have an internal memory which allows previous outputs to affect future ones. This is very useful in topics such as NLP (Natural Language Processing) where the order of the outputs matters just as much if not more than the outputs themselves. This type of data is called "sequential". An RNN has the following structure:

[1] https://link.springer.com/chapter/10.1007/978-3-319-64471-4_34#Sec3



An unrolled recurrent neural network.

The image on the left is the structure of a single node and the right is the same node unrolled over a training sequence. An input sequence **x** is fed into the RNN, and for each timestep x_t a new hidden state is calculated. The fact that prior values can affect future outputs allows RNNs to exhibit temporal behavior. However, as is evident from the right image, values more than a few iterations in the past do not affect the output as easily. This works well in scenarios that do not involve long sequences, but struggles at tasks like sentence generation. While RNNs are very useful for learning short term patterns and correlations, they struggle to learn the longer term dependencies that are an integral part of language. An RNN might be capable of choosing a word that plausibly comes next in a phrase, but it will struggle to always pick a word that fits given prior context. In vanilla RNN backpropagation, over many timesteps the gradients tend to vanish (if weights are <1) or explode (if weights are >1). The exploding gradient problem can generally be fixed with gradient clipping, which involves checking the gradients at each time step of backpropagation to make sure they are not too large. Vanishing gradients are harder to deal with, and require an LSTM. LSTMs include an added field known as the cell state, which is like a form of memory for the model. One example of an LSTM can be seen below:



The repeating module in an LSTM contains four interacting layers.

It is a bit difficult to see, but each node passes not one but two values to the next node. In the case of the image above, the upper line is the cell state while the bottom line is a combination of the previous output as well as the current input. There are many different variations on LSTMs and each has its own method of updating its cell state, but at its most basic, every time it receives new information, it decides how much of its cell state to "forget" and overwrites it with new information accordingly. In the case of the LSTM pictured above, how much of the cell state to forget/overwrite is determined by two separate sigmoid functions which scale the data to the range [0, 1]. The equation for an example sigmoid function can be seen below:

$$S(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}.$$

These sigmoid functions are called "gates". The cell state decides what the RNN remembers, where values close to 1 mean "remember everything" and values close to 0 mean "forget everything". The cell state remedies the vanishing gradient problem because it can remain relatively unchanged through an entire sequence. It acts as a "highway" for some information to pass without being forgotten. It is through this combination of new and old information that the network is able to formulate grammatically correct sentences.

We chose Google Colab as the platform on which to build our network. This was largely because neither of us felt we had a laptop powerful enough to train the network. Training a neural network can be very slow, even with a GPU, and neither of our laptops actually had one outside of the basic Intel Graphics card they came with. That is where Colab came into play. Much like Google Drive, it allows users to write and share code chunks with each other, a task which could already be done through Git. Unlike Git however, Colab has the added benefit that the code is run on Google's cloud servers, which have far more processing power than the average college student's laptop. For example, one laptop we had access to had 16 GB of RAM, whereas the free version of Colab has access to 25 GB by default. On top of this already fairly substantial upgrade, even the free version of Colab can run code on GPUs provided by Google. This is not as simple as just hitting run on a piece of code, as the script needs to be written in such a way that it can actually be run on a GPU, but it proved to be a simple matter of just switching libraries for matrix and vector math.

Related Work

One paper we drew inspiration from is written by Ashish Dandekar. In this paper, he set out to create a tool capable of generating realistic sounding headlines, as a way of training neural networks to detect fake news. He utilized a lot of techniques that we never got the chance to implement but would like to eventually. One such technique involved pre-processing the data by not only tokenizing sentences, but also encoding syntactic information into the training data. This allowed the network to not only learn the relation between various words, as well as grammatical structures. This meant that his network was capable of generating sentences that followed proper grammar. This is an area where our network still struggles, and leaves a lot of room for improvement. However, we did take a lot of inspiration from his methodology. For example, we chose to use the same data set in training our network. We used the same News Aggregator dataset, published by UC Irvine's Machine Learning department. This dataset contains

422,937 news headlines already sorted by topic. This allowed us to use only the Science and Technology sections in training our own network. Another article that was extremely useful in learning about the structure of RNNs and LSTMs was Andrej Karpathy's "The Unreasonable Effectiveness of Recurrent Neural Networks"[2]. Karpathy gives an extremely clear explanation of RNNs and provides examples of text generation using LSTMs. We also looked at an article by Denny Britz using a word-based language model instead of a character-based one, which was an interesting variation on conventional models.[3]

Project Description

For training, we had an important choice to make, whether to train our network on characters or words. Each option had its unique advantages and disadvantages. On the one hand, training it with characters would mean that our dictionary would only be 64 characters long, meaning that the computation for each timestep would be massively less. The natural downside to this choice is that the network would be creating sentences character by character, and as such the sequences are *much* longer. For example, in order to generate the following sentence "Reliable Amazon Isn't Anymore" the network would need to make a logical choice 29 times. On the contrary, if we trained our network on words instead, 29 choices would become four. However, it would come with the drawback that it is impossible given our time and resource constraints for the network to have a dictionary of every possible word in the English language. As a result, we would have to restrict the dictionary to a set of commonly used words. The limit we ultimately chose would be the 10,000 most commonly used words in the training data. Since the change in the model is small, we chose to implement both variations on our network. This way we would be able to compare their behavior and be able to make an informed decision moving forward.

Word-based Model:

In order to train our network we first had to pre-process the training data. We did this by first only selecting headlines from the News Aggregator relating to science and technology. We felt that in restricting our network to just one topic, we would achieve more distinct results because the vocabulary would be primarily related to this topic. We also wanted to see the grammatical differences between results from this dataset and a dataset from the Onion. Once we had selected our headlines, we then had to generate a dictionary for our network. We did this by tokenizing all the headlines into specific words, removing any punctuation. From there, we selected the 10,000 most common words, in order to prevent training words that aren't used very often. Next, we parsed through the entire data set and made two changes to the data. First we appended start and end tokens to each sentence, so that the network would learn the natural end to sentences. After that, we converted our training data into a series of numbers, where each number corresponded to that words index in the dictionary. For example the training set ["START_SENTENCE", "hello", "my", "name", "is", "END_SENTENCE"] might be converted to [0, 245, 953, 256, 1140, 1]. This would allow our network to learn the correlation between numbers rather than words. Converting between the two only required two dictionaries, each the inverse of the other.

Character-based Model:

The only significant difference in the two models is the way the training data is created. For the character model, all the headlines are joined together to create a large corpus of text. From there, a dictionary of all characters is created, and trimmed to the most common. The text is then split into sequences of 40 characters. We also artificially augmented the data by creating overlapping sequences. The data was then processed the same way as in the word model, without the addition of sentence delimiters because the sequences used were not full sentences.

Once we had preprocessed our data into a series of numbers representing words, we created two sets of training vectors, the input and the output. As mentioned above, in the case of our word network, the input vector was the headline itself with a "START_SENTENCE" token appended to the beginning. The output vector would be the headline with a "END_SENTENCE" token appended to the end. This way, the relationship between the i_{th} value in each vector was the same as that of two sequential words in the headline. The network was set up this way for several reasons, firstly, it simplified the task of teaching the network the natural progression of words with just one vector, but additionally, including the start and end tokens had two uses. Firstly, we would then be able to give the network a seed of "START_SENTENCE" as a way of generating completely random headlines with no predetermined topic. But more importantly, the inclusion of the end token meant that the headline could end before the desired number of words, if the network felt that was necessary. For example if we requested a ten word headline seeded with "Apple", the network could output

[START_SENTENCE, Apple, releases, a, new, ipod, nano, END_SENTENCE]

While this headline is in fact only five words long, the sentence reached its logical conclusion and as a result there was no need to generate another five words. This logic was

[2] https://karpathy.github.io/2015/05/21/rnn-effectiveness/

[3] http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/

included in our sentence generating function as a way of ensuring that the network did not just keep tacking on words once the end of a thought had been reached.

There are several options for how to evaluate the behavior and performance of the network. For training our network we evaluated it based on the cross-entropy loss between the training data and our results. The cross-entropy loss determines the difference between an expected and predicted probability.[4]

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right)$$

$$H(p,q) = -\sum_x p(x)\,\log q(x)$$

The cross-entropy equation

Another option would have been to use the perplexity of the model. Perplexity is an extension of entropy, but in the case of natural language processing, it represents how confused the network is by a new word. A higher perplexity means that the network is closer to just choosing words at random, whereas a lower one means it has some understanding of the relationships between words.We ultimately chose to evaluate the training of the network based on loss, but it stands to reason that evaluating it on perplexity could have beneficial results.

When it came to evaluating the fake headlines our network created, we used a much more subjective process. We would run the network several times, generating headlines with random seeds and lengths, to see what it would come up with. We would then read through the results, saving any that seemed interesting enough, taking snippets of larger headlines if they made some amount of sense. Deciding whether a headline is realistic is a fairly subjective process, and as such we did not have much choice other than to evaluate them by hand.

Empirical Results

In the beginning, the headlines outputted by our network were little more than random words and phrases thrown together, often making little to no sense. Occasionally a fragment of an idea could be extracted from a larger nonsensical sentence. Our word-based network consistently outputted headlines such as:

Xbox 11

or

American valuable parrots

or even

Streaming later to android

These were fragments of thoughts, and nothing more, but they proved that the network was heading in the right direction. It was around this time that another issue with the network became apparent. Even with the sentence generation stopping when a "END_SENTENCE" token was reached, the network had a habit of getting stuck on a word. For example, the network would occasionally start a sentence fairly logically:

Leaked Were Unveils…

but then it would take an interesting turn, and repeat the same word over and over, ad infinitum

… Vmworld Vmworld Vmworld Vmworld Vmworld Vmworld Vmworld Vmworld Vmworld Vmworld

We finally realized that our models' word prediction function was incorrect, and was asking it to predict the next word from the same sequence over and over. We eventually got results looking like this:

morpheus galaxy to for video is apple not to premium

Which is not incredible, but is close to a readable sentence.

Our initial implementation of the character based network had very similar behavior, with the exclusion of the repeated word bug. Some of the headlines it would output in the beginning were as follows:

new years puying deuto crower vife

or

reportedly stall a softband decaunt nead

or even

smartwatch dicnersing gmail-totere with duvidentents

These headlines were much more varied than those generated by our initial word-based implementation, but they had the downside that typos could and would occur fairly often. It is entirely possible that these typos were due to lack of training time for the network, and that if we simply left it to run for days at a time they would decrease, but we never got the time nor the opportunity to test this hypothesis.

Our keras implementations were able to create legible output, even verging on sentences. We trained the same model on the News Aggregator and on the Onion datasets, and received markedly different outputs.

For the model trained on News Aggregator, we achieved results that looked like this:

t recall - analyst blog gm faces probe in comcast to wearable google developer promines app app science to pro to internation to and to says

For Onion, we achieved results like this:

n really terrible job that pays shit area man is button his increde to expells 14 things customers him video park friendtursing mince jeweruss the ecterned watching

[4] https://www.superdatascience.com/blogs/convolutional-neural-networks-cnn-softmax-crossentropy

doncan horrand space bad bas dilier monniences stloting to fit out attennei
all 77 million in user donations decking the her machine the darreve basts who have when out of bathroom sweat now out and alfucker it track the while have to start in the onion of and this man have bookea on the town to all the onion life

An important note is the significant difference in tone between the results. The text generated by the News Aggregator reads like a NYTimes headline, while the Onion article reads almost like a someone who is complaining online. It also has distinguishable swearing, which is definitely indicative of the Onion.

Conclusions

Our results were both exciting and disappointing. While our Keras models performed fairly well, able to generate somewhat readable text, our home-built models did fairly badly and were much more work-intensive than expected. We would occasionally see full or partial thoughts pop up in our results, but never the full grammatically correct sentences that Ashish achieved. In several of the similar examples we found when researching, there was a common trend of training the network for hundreds of epochs, whereas we could only manage to train it for about ten at a time. This likely contributed a fair bit to the differences in performance between our network and those published in papers. It is possible that with enough training time and examples, our homebrew network could approach the Keras one. Unfortunately, considering that TensorFlow was written by professional researchers, I doubt our network would ever surpass it.

Santiago's Perspective -

I had never had much experience working with artificial intelligence prior to going into this class. I knew a small amount about the basis upon which they operated, and had always been interested in it, but I had never actually sat down and tried to code a network from scratch. This class as a whole was my first practical experience working with neural networks and artificial intelligence as a whole. I truly learned a lot from this class and from working with Drew. He had a fairly strong foundation in machine learning and algorithms coming into this class, which meant I always had help readily available should I need it. I learned a lot about neural networks from working on this project, primarily about the math that drives them. It is very easy to just dismiss the math behind a network when discussing them, and just treat it as a black box. Data goes in, data comes out, but as for what happened in between who knows. That being said, just because I had a decent understanding of how neural networks worked, it did not mean that actually implementing one would be easy. There is a lot I would have liked to have done differently with this project. First and foremost, I would have liked to start the project much sooner. I feel that with more time we would have been able to get better results. Unfortunately with the particular circumstances surrounding this semester, that was easier said than done. The transition to online classes meant that I often did not have enough time to devote to a class or project as I would have liked. Additionally, the fact that Drew and I could not meet in person did not help much either. Thankfully Colab meant that we could work on the same file without needing to use Github, but as with most forms of cooperating on code, only one person could really work on it at a time. This meant that even if both partners were free to collaborate, only one could physically work on the code at a time, slowing things down. Additionally, I think that we chose a topic that proved to be very interesting, but also very challenging given our time constraints. Had I known that this semester was going to progress the way it did, I might have chosen a slightly less complicated topic. I intend on continuing to improve this project. I think it would be interesting to apply the LSTM network to other problems more complicated than just headlines and potentially one day be able to generate entire articles as well.

Drew's Perspective - Looking back on this project, I would have liked to have set out clearer goals for the project at the beginning. We ended up going in two different directions, creating our own model and comparing the results of models trained on different data. I think it would have been much more interesting to dive deeper into the Keras model results, possibly by using more datasets and even attempting to classify the results back into each dataset. One aspect of machine learning that has frustrated me is the ever-present "black box" where all the actual machine learning processes-backpropagation, gradient descent, learning rate optimization-are hidden. This frustration motivated the aspect of the project in which we attempted to build our own RNN. We wanted to have a deep understanding of what the RNN was doing before using it. The issue with this was that actually implementing an RNN is difficult, and implementing an LSTM was extremely difficult especially given the limited resources (most examples are using Keras, Tensorflow, PyTorch, etc.). The struggle with trying to reinvent the wheel took a lot of time away from working on more interesting aspects of the project. That being said, I think that this project helped me learn a ton about recurrent neural networks and machine learning in general, and I have a much better grasp on the topics relevant to the project. It goes without saying, but the switch to online classes was a large issue as well as it hindered collaboration and slowed down progress a lot. At times it felt like we were working on two different projects because it was so difficult to update each other about every little change we made that day. If I had to start this project over from scratch, I would have focused much more on utilizing the Keras implementations to do something really cool, rather than getting lost in the weeds with backpropagation through time and LSTM

implementations. Moving forward, I would like to apply what I've learned about RNNs to other machine learning problems, particularly image captioning and video processing.