TWEET GENERATION USING GATED RECURRENT UNITS

Alex Kosla

ABSTRACT

Natural Language Processing is an exciting new subset of Machine Learning that allows machines to "understand" and emulate human language. One of the most popular ways of consuming human language in the present day is via social media, in which users can create recognizable personalities for themselves using only characters. Recurrent Neural Networks (RNNs) are well-suited for Natural Language Processing and I have created my own Recurrent Neural Network for the purpose of imitating not only human language, but specifically the vocabulary and speech patterns of a specific Twitter user. In particular, I experimented with the usage of Gated Recurrent Units (GRUs) and their efficacy with Natural Language Processing, specifically at an integer-encoded character-level. Additionally, I explored a solution to the issue of Recurrent Neural Networks requiring user input in order to operate, and devised a novel approach to determining the accuracy of text generation, comparing the output text to a corpus of words in the dataset. Using this accuracy metric, I was able to achieve a 94.2% accuracy and emulate many key aspects of the Twitter user's writing.

1. INTRODUCTION

There exist a great deal of media generators on the internet, some powered by Deep Learning, some not. Two existing examples include Garkov and Bots of New York. In the words of its creator, Josh Millard, 'Garkov is an application of the Markov model to transcripts of old Garfield strips, plus some extra code to make it all look like a genuine comic strip.' [1] Bots of New York is much more complex, and believable. Though documentation of its inner-workings are difficult to find, it seems to be a parody of the very popular social media content creator, Humans of New York, which offers readers brief vignettes of the lives of New Yorkers, via a brief monologue from a New Yorker and a picture of them [2]. I speculate that Bots of New York uses a corpus of words from Humans of New York to generate text using Deep Learning, and pairs it with an image of a human face also generated by Deep Learning [3]. The posts can do a very good job at imitating Humans of New York, but sometimes they can humorously go awry, with awfully disfigured faces and very strange, near-schizophrenic monologues.

Such projects raise a great deal of awareness for the potential of Machine Learning and inspire readers such as myself to get involved and introduce themselves to it. I especially think Artificial Intelligence has a bright future in the fledgling field of esports, the new billion dollar industry focused on competitive video games [4], whose influence I have seen firsthand while working at a esports-focused Data Science startup which used Machine Learning to assist coaches, organizations, players, analysts, commentators, and journalists in a deeper understanding of their game and what things can influence wins and losses.

For this purpose, I have chosen to imitate a social media user in the field of esports, specifically Duncan "Thorin" Shields. An esports journalist and analyst who prides himself on his knowledge of esports history, he tweets almost exclusively about esports, and he tweets very often. At the time of writing, he has over 80,000 tweets. He is someone that I personally follow on Twitter and I have grown very accustomed to his style of tweeting, which uses many words and phrases not found in the vocabulary of normal twitter users, largely due to heavy use of esports jargon. The niche nature of his Twitter allowed me to more easily analyze if my methods were producing generic English sentences, or sentences from a very specific person. Additionally, as I plan to publicly publish my results, some esports fans will hopefully find themselves inspired into learning more about Machine Learning. The more people involved with Machine Learning in esports, the better, as I believe, as this will increase the level of analysis by broadcasters, and the level of play by professionals.

This paper describes my process for building a corpus from a dataset of tweets from Shields, encoding each character as an integer, creating sequences of seventy integers to feed into a neural network. Without going into too much detail already, to train the dataset, the sequences of mapped integers are fed as tensors into a Recurrent Neural Network, which then outputs a sequence of mapped integers, which in turn are decoded from integers to text. For generating text, the model is presented an input sequence of a k-length string, pulled from a list of first words in tweets, and then predicts what the next most likely character will be, given the n previous characters in the sequence.

2. RELATED WORKS

Recurrent Neural Networks come in more than one variety, and it is important to select which is best for this paper. The so-called ”vanilla” Recurrent Neural Network has flaws in handling length lengthy sequences [5], but thankfully alternatives exist, with primary competing flavors, LSTMs and GRUs, and both have been used successfully in Natural Language Processing [5, 6].

Chung et al. and Rana et al. compared the efficiency of the two methods using models that interpreted speech as soundwaves rather than as characters or words. The results looked somewhat favorably on GRUs. When attempting to perform emotion classification on noisy speech data (both literally and technically), Rana et al. was able to achieve similar accuracy for GRUs as with LSTMs, but with nearly 20% shorter run-time [6]. These results inpsired me to use GRUs in my RNN. Other papers I read did not seem to outright condemn or condone their usage in text generation, which should be noted is quite different from emotional classification of soundwaves. Regardless, Rana et al. used an GRU with Natural Language Processing, it led me to believe that for text generation it would still work fine, but maybe not impeccably.

Two different papers offered different sorts of advice for Text Generation. Sutskever et al. nudged me towards character-level encoding. Admitting that while character-level encoding may seem inferior to word-level encoding from an outside perspective, Sutskever et al. asserted that if you can find a form of RNN that learns words very well, you can benefit from the ease-of-use of character-level encoding and still get excellent results. Their use of a Multiplicative Recurrent Neural Network was exciting, but I felt a little too outside of the scope of this project, as while it seems to have good results, there is limited documentation and implemenations of it available. Unfortunately, they did not compare its effacy to GRUs or LSTMs, only vanilla RNNs, so it’s difficult to gauge whether or not an implementation would be worth the headache [7]

Lastly, researchers at Facebook questioned the entire standard learning routine for Text Generation via RNNs. They offered an alternative method, dubbed MIXER, which included gradually exposing the model to more and more of its own predictions over time as a method of training. Most impressive to me was their idea of using a loss function that operates at the sequence level [8]. However, this was far too ambitious to implement for my personal project.

3. DATASET

Two related datasets were used. One contained the text of all tweets by Shields in a 52-day span and the other contained only the first words in each tweet from Shields in that span of time. Both were created using a modified version of a Twitter scraper written by Tom Dickinson [10]. Though there exists a first-party API written by Twitter which can be used for downloading datasets of tweets, I found it unsatisfactory. This is due to Twitter preventing API users from downloading any tweets aside from the last 3200 tweeted by the user [11]. I wanted a larger corpus to work with, and I had chosen Shields as my tweeter of choice partially due to the large number of tweets I’d be able to work with.

3.1. Data Collection

Dickinson’s scraper offered a clever workaround. Twitter has an advanced search function, which allows users to make very strict specifications on which tweets they would like to see. For example, one can search only for tweets from Shields containing the string “hello” from the timespan between March 11th , 2016 and March 17th, 2016. Dickinson exploits this feature to download tweets in batches, with the scraper getting as many tweets as it can from a certain day and iterating through a list of days defined by the user. It should be noted that the scraper fetches tweets starting from the lower bound of the timespan input by the user, so in our example, it would begin with tweets from March 11th, 2016.

However, Dickinson's implementation did not include a method of saving the tweets gather, which I implemented myself. I opted for the usage of two related datasets. For each tweet fetched by the scraper, I first took the tweet, provided it was not in some way null, and split it into a list delimited by the space character, to get myself a list of all words in the tweet. I then took the first element of the list of words, added the '\n' character to it as a string, and concatenated that to a file of all other first words in Shield's tweets, creating a new .txt file if one did not already exist. Thus, I created a dataset which had one first word per line, of all the first words in all tweets scraped, in the order in which they were retrieved. I similarly concatenated the full text of each tweet to a line of a .txt file, adding in a newline after each tweet was written to file. However, this method was not without its flaws. Importantly, tweets can contain newline characters, which means that my delimiter in this case, the newline, was flawed. I had the opportunity to instead opt for a different sort of character in place of the newline, for example, a Unicode Cuneiform character. However, I preferred the readability of this method, which prevented potentially infinite amounts of tweets on a single line. Additionally, my Recurrent Neural Network is fed 70 characters at a time from the full tweets dataset, essentially blind as to where one tweet begins and where one ends, so I deemed it not too significant of an issue, my model would just be more likely to produce double-spaced tweets rather than single-spaced tweets.

The inclusive timespan I chose was from January 1st, 2018 through February 21st, 2018, or 52 days. After reading in the individual text files in python, I made a list of all the words in each by using the split function and the strip function to eliminate unwanted whitespace characters. To get the size of my corpus, that is, the list of all unique words found in either of my datasets, I made use of the set function, which returns the set of all items in the list, essentially removing any doubles from my list. This amounted to 5536 tweets, 5536 first words, 26123 characters in the first words dataset, and a corpus of 745 unique first words. In regards to the full tweet dataset, it contained 366148 characters, 55307 words, and 8518 unique words in its corpus. My professor, Fr. Joseph Puliparambil, informed me that 5000 tweets would be a good number to use as a dataset. I agree, as a larger dataset may have been much more difficult to train in terms of time-complexity, and a smaller dataset may have led to overfitting [12].

3.2. Character-level Encoding

Next the data needed to be encoded for use with the Recurrent Neural Network. As RNNs take tensors as input, that is, the mappings of relationships between different objects, we must encode our data accordingly. The most effect choice for this would be a form of one-hot encoding, which is very effective for character-level encoding. One-hot encoding encodes each character into a large vector with all values in all dimensions in the vector marked as zero aside from whatever the specific dimension in the vector corresponds exactly to what it is encoding. I do very much the same thing here, except I don't store all the extraneous 0s as data and only store the index of the corresponding dimension.

To do this, we map each of our characters to unique numbers. First, we use the set function to get the set of all unique characters in our text, which is 118 characters. We sort them and create a little function that given a list of characters, will return their indices in the list in order, mapping our characters to values. Given the list of characters $\{$I, t$\}$, the mapping function will return to us a list of integers [42, 83]. We can also define a way of decoding the integers into characters by interpreting our sorted list of unique characters as an array, and just using the integers as indices in the array to return back the character of choice [13].

3.3. Sequences

The Recurrent Neural Network takes in sequences of characters. The beauty of an RNN for Natural Language Processing is its ability to predict based on what it has seen before. I chose to define my sequence length as 70 characters, or one fourth of maximum length of a tweet. Because tweets are variable length, shooting for a full 280-character sequence each time would likely encompass multiple tweets at once. The average length of a tweet in this dataset was the total amount of characters divided by the total number of tweets, which is about 66 characters per tweet. Because of the nature of our data and the fact we must input consistent length sequences into the RNN, it is impossible to balance these and impossible to input in only a single tweet's worth of data each time. To determine how many times we need to feed these sequences of text into the RNN, we simply divide the length of our text by the length of our sequence, and discard the remainder. The RNN must take in data of the exact same shape each epoch, so we cannot use partial sequences of data.

Tensorflow's Dataset.from_tensor_slices method allows us to reformat our encoded text directly into tensor format, which will be readable to the RNN. The batch() method allows us to create an object that will portion our dataset into sequence-length batches, meaning that we can just feed our RNN that object and not have to write any loops to feed the RNN our sequences. Our dataset is now an object that will, in order, iterate in sequences of 70, our text that was encoded as integers. Next, we assign a mapping function to the object that will split the data into two separate chunks, the input sequence and the target sequence, something we can train with. Lastly, we shuffle the data to avoid overfitting, and set a batch size, or sequences being input at a time, for the RNN [13].

4. METHODS

The Recurrent Neural Network is a sequential model, consisting of five layers:

- An embedding layer
- A Gated Recurrent Unit (GRU)
- A dropout layer of 30%
- A second GRU, with more hidden units
- A dense output layer with an output for each character

4.1. Building the model

The embedding layer acts as the input layer in this RNN. An embedding layer takes in a batch size and a sequence length essentially take an input matrix of *n* batches of integers of sequence length *m*. It then transforms these into vectors of 256 dimensions, which will be used as input into the GRU.

The Gated Recurrent Unit layer acts similarly to the more familiar Long Short-Term Memory layer, but there are some key differences. First, it should be noted how each of these work. Both operate recursively. A simple, ”vanilla” Recurrent Neural Network calculates probabilities of a number based on previous items in the sequence. In essence, an RNN calculates

$$P(x_n|x_1, ..., x_{n-1})$$

where *x* is a sequence of *n* items. However, RNNs have their shortcomings, especially when the sequence is a particularly long one; regardless of the importance of the first item in the sequence, it usually is replaced by items that come later on in the sequence [5].

LSTM units and GRUs offer alternative, similar approaches, with key distinctions. Both use additive methods rather than replacement methods, that is, they keep the existing content in the unit and add new information and content on top of it. Traditional RNNs replace the previous information in a unit with new information each time. An important distinction between LSTM units and GRU is that GRU require fewer inputs (and fewer gates) and make fewer calculations. However, a GRU does not have gates to determine how much of its memory to show to other units, how much memory to forget, and how much memory to add, it only has a single gate that just determines how much to replace its memory by. I chose to use GRU layers as they has been shown to potentially be more time-efficient than LSTM layers while yielding similar results [5].

I used two separate GRU layers for higher accuracy, but to prevent over-fitting from the large total amount of RNN units, I introduced a dropout layer between them. Dropout layers randomly exclude a percentage of units during training time.

Lastly, the dense, fully-connected layer receives all of the information from the second GRU layer with as many outputs as there are unique characters in our dataset.

4.2. Running the model

Our loss function is set to be Sparse Categorical Crossentropy, due to the fact that we are using integer encoding, not true one-hot encoding. This makes sense, as our integer encoding is essentially a sparse matrix, where we do not bother tallying all of the zeros and only tally the non-zeros, in this case, the dimension of our character. We use the categorical form of crossentropy because this is a categorical problem; our expected output is not binary, we can have as many possible outputs as there are indices in our lookup table, which in this case was

118. Adam was used as the optimizer as it works similarly to traditional gradient descent, which would work in this context, but is believed to be better [9].

A loop instructs the model to, each epoch:

1. compile the model
2. perform one epoch of fitting
3. save its loss value for the current epoch
4. save its weights
5. make an altered model for testing purposes
6. pick a random first word from the list of first words for use as an input string
7. use the altered model, corpus, and input string with a text generator function
8. print a snippet of generated text and the accuracy for the current epoch
9. save the accuracy for the current epoch

We need to define a way to actually use our model once we have trained it. In order to test our RNN, we will need to alter it a little bit to handle only a single string of input rather than a full batch. Thus, we'll need to save our weights after each fit if we want to get an idea of accuracy for each epoch. The vast majority of RNNs, from what I've seen, require the user to input a string and only produce text, with no attempt at an accuracy metric.

My text generator function (adapted from tensorflow's guide) [13] takes in a corpus in addition to a model and a starting string. It operates by encoding the random first word selected as a list of integers, feeding it into altered version of the model which only expects a single sequence. The model produces a number, which we pull from it using a random categorical distribution weighted by a temperature variable. The temperature variable adds a degree of randomness to the selection, yielding to more variance in text. The number is then decoded back into a character and concatenated onto a string. It generates 10 tweets' worth of characters, makes words out of them by splitting the string with a space character, and then compares the created words to the ones in our corpus. We then make a hit ratio of what percentage of generated words were words actually in our corpus (words tweeted by Shields).

This is by no means a perfect accuracy metric, as it only checks for if the model is stringing together words in the corpus, but does not check if they are being strung together in the right order. Despite this, I thought it would be nice to have something to tell us we're moving in the right direction. We return a single tweet's worth snippet of text and the accuracy. The first is printed for transparency of how the RNN is doing and the accuracy is saved for later.

5. EXPERIMENTS

Unfortunately, due to limited processing power and time-constraints, I was not able to tinker with the implementation of this model as much as I would've liked. I was not able to test single variable in the implementation to achieve a sweet spot. However, I was still able to make adjustments for the better.

Using smaller datasets yielded worse results and using bigger datasets took so long to train that I was not able to train for as many epochs as I did for this specific implementation. Using a dataset of approximately 1500 tweets, I achieved a 72.2% accuracy after 5 epochs, contrasted with an accuracy of 80.9% after 5 epochs using the larger 5000 tweet dataset. However, it should be remembered that my accuracy metric is flawed and should primarily be used to tell if the model is producing actual words, not gibberish. For example, compare some select sentences taken from 3 different epochs and the accuracy metric of the text:

have gitch?t winch the saputicale who of
Accuracy: 56.9%, Epoch: 3
You find at all all the are that The MVP
Accuracy: 89.1%, Epoch: 11
Pleb: "LOL they won the old major qualifier
Accuracy: 93.7%, Epoch: 20

While we can see that the accuracy metric is imperfect, the quality of the output increases with more epochs. Unfortunately, as mentioned before, I could not train the model enough epochs to see when the overfitting began, as this model seemed underfitted to me.

Altering the temperature variable greatly affected my results. Having a temperature variable of 1.0 yielded results similar to epoch 3 above, for many epochs longer. A temperature of 0.5, meanwhile, produced something completely on the other side of the spectrum, with the

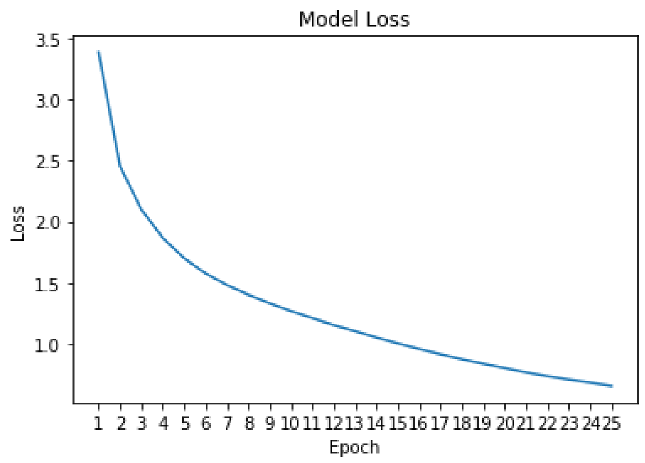

Fig. 1. Loss function over time.

majority of generated text after only a handful of epochs being only the most popular English words, such as ”a the the are why is the.” A value of 0.75 for the temperature produced my best results, as though they had a lower accuracy according to my metric, they seemed to be the most human-like.

Altering the dropout layer percentage from 15% to 30% did not seem to strongly affect my results. Similarly, adjusting the batch size from 64 to 128 did not seem to noticeably affect results either.

Reducing the total number of hidden nodes to 128 in the first GRU layer and 256 in the second GRU layer achieved similar accuracy ratings in terms of the model's ability to produce English text, but qualitatively, they were much worse than ones with more nodes, though they trained much quicker. Qualitatively, their results after 25 epochs were similar to the results using 512 and 1024 units in the layers, respectively, after approximately 6-7 epochs. This aligns very well with the value of the loss functions. After 7 epochs in the more complex RNN, the loss function was at 1.48. After 25 epochs in the simpler RNN, the loss function was at 1.46.

6. RESULTS

My best results, the ones shown in figures 1 and 2, were achieved after 25 epochs of training, the most I was able to complete. They used a batch size of 64, 256 embedding dimensions, 512 nodes in the first GRU layer, 30% dropout, 1024 nodes in the second GRU layer, and a temperature of 0.75. These were, on epoch 25, 93.4% accuracy and a loss value of 0.66.

The shaky curvature of the model accuracy metric over the course of the training shows its imperfections, though it does seem to gradually increase after its very rapid ascent in early epochs, which does reflect the improvement of the model. However, it does not accurately depict the increase in Shields-esque characteristics in the generated text. This was to be expected, and the accuracy metric still, I feel served a purpose, though limited, to inform me as to whether or not the model could at least produce english words somewhat consistently.

It is notable that by the later epochs, the model was very clearly attempting to imitate Duncan ”Thorin” Shields's tweets. However, they still possessed far from perfect grammar and spelling, and used words in inappropriate contexts, such as mentioning a player from an incorrect esport when talking about a completely different esport. I showed the results to a few other fans of his, and they were not fooled into believing that Shields himself wrote them, they were very impressed at how similar the text was to his tweets. The topics and phrases all seem to be there for his signature style.

The question of the strength of this specific model over other similar implementations remains unanswered, due to hardware limitations that prevented me from training a variety of different implementations to the point where they were very clearly overfitted. None of the parameter combinations seemed to strongly approach overfitting, and while we can say that some qualitatively were closer to the goal than others, not enough testing was done to see which parameter set was most efficient.

7. CONCLUSION

This paper attempted to test the ability of a Recurrent Neural Network using multiple Gated Recurrent Unit layers to accurately imitate tweets written by a specific individual, Duncan ”Thorin” Shields. After scraping twitter for his tweets and using a subset of his tweets as a dataset, I encoded the entirety of his tweets at a character-level using integer encoding. Batches of sequences of characters were created and used to train a Recurrent Neural Network consisting of an embedding layer, a GRU layer, a dropout layer, another GRU layer, and a dense output layer. The model was fit for number of epochs, generating text after each epoch run, randomly selecting a first word in a tweet from the dataset

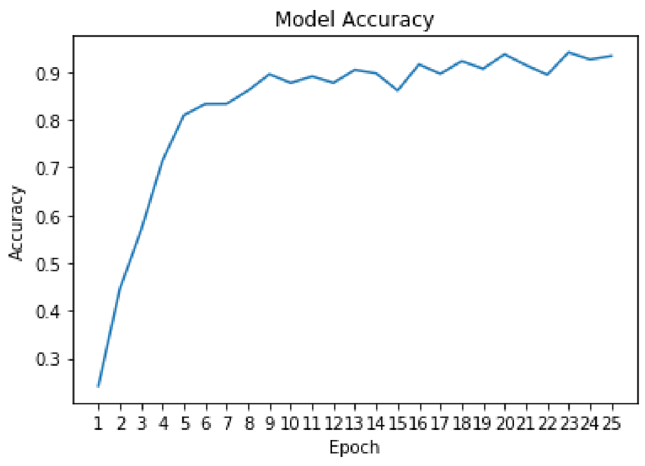

Fig. 2. Accuracy of model over time.

as an input string. The model was able to clearly imitate Shields's tweets, but struggled with making believable tweets.

Had I possessed more time and more resources, I strongly believe I could make significantly more convincing tweets. More training and more tinkering with parameters will almost certainly yield more exciting results. Additionally, I would have liked to have tried other methods of text generation, including comparing the efficiency of an LSTM model to that of a GRU model, and see if word-level encoding produced better results than character-level encoding. Experimenting with larger datasets would also likely yield model improvements.

8. REFERENCES

[1] Millard, J. (2019). Garkov – Garfield + Markov chains – Josh Millard. [online] Joshmillard.com. Available: http://joshmillard.com/garkov/ [Accessed 27 Sep. 2019].

[2] "Humans of New York," Humans of New York. [Online]. Available: https://www.humansofnewyork.com/. [Accessed: 02-Dec-2019].

[3] "Bots of New York - Home." [Online]. Available: https://www.facebook.com/botsofnewyork. [Accessed: 02-Dec-2019].

[4] K. Webb, "Esports revenue is expected to top $1 billion worldwide for the first time during 2019," Business Insider. [Online]. Available: https://www.businessinsider.com/esports-revenue-billion-2019-2. [Accessed: 02-Dec-2019].

[5] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling," arXiv:1412.3555 [cs], Dec. 2014.

[6] R. Rana, "Gated Recurrent Unit (GRU) for Emotion Classification from Noisy Speech," arXiv:1612.07778 [cs], Dec. 2016.

[7] I. Sutskever, J. Martens, and G. Hinton, "Generating Text with Recurrent Neural Networks," p. 8.

[8] M. Ranzato, S. Chopra, M. Auli, and W. Zaremba, "Sequence Level Training with Recurrent Neural Networks," arXiv:1511.06732 [cs], May 2016.

[9] J. Brownlee, "Gentle Introduction to the Adam Optimization Algorithm for Deep Learning," Machine Learning Mastery, 02-Jul-2017.

[10] "GitHub - tomkdickinson/Twitter-Search-API-Python: A Twitter search client mining tweets using their advanced search implemtation." [Online]. Available: https://github.com/tomkdickinson/Twitter-Search-API-Python. [Accessed: 02-Dec-2019].

[11] "GET statuses/user_timeline." [Online]. Available: https://developer.twitter.com/en/docs/tweets/timelines/api-reference/get-statuses-user_timeline. [Accessed: 02-Dec-2019].

[12] enrique a, "Word-level LSTM text generator. Creating automatic song lyrics with Neural Networks.," Medium, 24-Sep-2019. [Online]. Available: https://medium.com/coinmonks/word-level-lstm-text-generator-creating-automatic-song-lyrics-with-neural-networks-b8a1617104fb. [Accessed: 02-Dec-2019].

[13] "Text generation with an RNN — TensorFlow Core," TensorFlow. [Online]. Available: https://www.tensorflow.org/tutorials/text/text_generation. [Accessed: 02-Dec-2019].