Creating Poetry with Generative Adversarial Networks and Recurrent Neural Networks

Sam Burtch

Abstract

The main objective for this project was to generate somewhat legible poetry using Generative Adversarial Networks (GANs) and Recurrent Neural Networks (RNNs). To do so, I used a Kaggle dataset of over 500 poems from the Renaissance and Modern eras of poetry to be used as the real poems that the generator portion of the GAN will attempt to imitate. The selection of GANs was done due to their increasing popularity and effectiveness at generating images and other media. As well, I chose to use RNNs as the type of neural network framework for both the generator and discriminator due to its relevance dealing with time-series data, similar to poetry.

In order to convert a poem to a standardized form I converted all the distinct words in each poem to integers and added padding for shorter poems so that all poems are of the same length. Instead of feeding the network integers with arbitrary relation to one another, I used the GloVe vector representation of words to feature engineer higher dimensional space to represent each word. Unfortunately, the network appeared to have difficulty learning the padding feature of each poem, and the generator ended up producing vectors very close to the zero vector.

Introduction

General Adversarial Networks have been increasing in popularity since their introduction by Ian Goodfellow et al. in 2014. [1] Personally, I have become fascinated by the potential impact that generative machine learning algorithms can have moving forward. As most of the applications of GANs I have seen online have used static data, such as image generation [2], I wanted to see if the same logic can be applied to data that had a time-series like structure. Thus, poetry seemed like the easiest domain to apply this philosophy due to its low complexity and strong correlation throughout each word in a poem, such as rhyme schemes and syllable count. Because GANs are notoriously hard to train, my expectations on this project were to simply see if we could get legible output from the final trained generator.

Data

The data came from Kaggle [3] in which over 500 poems from the Renaissance and Modern era of poetry, the likes of Shakespeare, were used as the real data that the generator would attempt to imitate and the discriminator would attempt to classify as ‘real’. Simple hygiene steps were taken to clean out any non-words and punctuation, as well as eliminating any poems with too short of length

(less than 100 in character length) and too long (greater than 1000 in character length). [4] This shortened the list of poems down to 349. From there, using the keras preprocessing library I used the Tokenizer function-set to index each unique word in the set of poems from a string to an integer. For example, ‘the’ would be converted to ‘1’. In order to keep the same length among poems, I padded poems shorter than the longest poem (177 in length) with zeros, since zero doesn’t correspond to any word in the Tokenizer dictionary.

Along with converting words to numbers, since integers might have relation to one another, say 256 and 257, that might not have relation in word form, such as ‘queen’ and ‘frog’, I used a form of feature engineering to convert each integer into a series of higher-dimensional floating point integers. This was done using the GloVe vector representation from Stanford University. [5] This would allow each word that has similar meaning to have similar vector space. For example, ‘frog’ would have a similar vector space to ‘toad’ and dissimilar vector space to ‘queen’. For simplicity, I chose the smaller of the GloVe packages, which had 50 dimensional vectors for each word. Thus, the final dataset of the *real* poems were of *349* x *177* x *50* (# of poems x # of words per poem x # of floats per word).

Network

As discussed prior, GANs are made up of two different networks that are adversaries, the generator and the discriminator. The generator’s job is to take random noise (distributed from a Gaussian distribution) and produce poetry that the discriminator has a hard time distinguishing from the real set of poems. The discriminators job is to then take the generated poems and predict if the input data was real, coming from the poem data, or fake, coming from the generator. For a loss function, both the generator and discriminator use a binary cross-entropy loss, since the final output is binary and the generator is attempting to maximize the loss of the discriminator and vice versa. This is where the adversarial term of the name comes from.

As well, RNNs can use a type of network cell called an LSTM cell, which has three different types of weight matrices, known as gates: input gate, forget gate, and output gate. These gates manipulate a recurrent state known as the hidden state, which gets passed along to each step as inputs, along with the actual input at each step. The input gate determines how much of the input to add to the hidden state, the forget gate determines how much hidden state to forget, and the output gate determines how much of the hidden state to output at each step.

Because GANs act as the outer framework of the network, we can implement the generator and discriminator using an RNN structure, with LSTM cells as nodes in the network. This will allow the network at both ends to read through each poem sequentially and determine the next word at each step in the sequence.

At each layer, I used 50 different LSTM cells corresponding to the 50 floating point integers of each word (coming from the GloVe representation). In the generator, I used 5 different LSTM layers, which I'll explain later, and for the discriminator I just used one layer of LSTM cells. The final layer of the discriminator is a Dense layer, or fully connected layer, with one output from a sigmoid activation, essentially predicting the real or fakeness of the input (i.e. 1 or 0). [6]

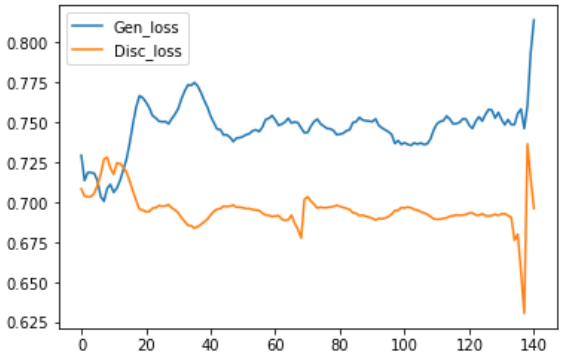
Training

Although trained over 1000 epochs, I put a stop in the training step if the loss of the generator gets over a certain threshold, which will come in handy later in the conclusion. The generator is fed a series of random noise vectors sampled from a Gaussian distribution of the shape 177 x 1 (# of words per poem x 1) which is then combined with a sample of the real poems and fed into the discriminator, along with corresponding labels. To aid the generator, we add in random noise to the labels fed into the discriminator as a way of keeping the loss for the discriminator to be nonzero. The discriminator is then trained on the batch on the loss of the predicted labels. Because the discriminator loss function was explicitly set apart from the larger GAN architecture, we can train only the discriminator, which is separate from the generator.

This process is repeated for the generator but instead of feeding the output into the discriminator, I trained the entire network on the output of the generator, with incorrect labels, in order to train only the generator, because the discriminator was set to not be trainable by the whole network. (We have to explicitly train it, as above.) The loss of each network is monitored and used for when the loss is saturated and on the verge of exploding. This appeared to be an issue with the two networks having dissimilar capabilities, as discussed below.

Errors

Upon training, a trend of the discriminator quickly beating the generator appeared to ruin the adversarial nature of the two networks. As seen in the graph of the loss below, the two networks seem to be competing nicely until some point in which the discriminator loss quickly vanishes and the generator loss explodes.

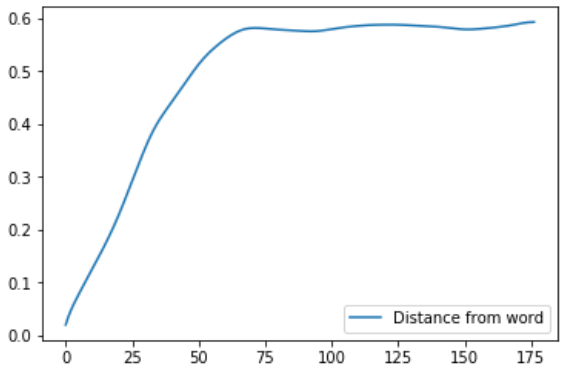


Thus, I needed to put a stopper for the training loop to keep the training from furthering this divide. In order to combat the issue, I was able to delay this occurrence by increasing the batch size to 30, increasing the number of generator samples by batch size squared, and increasing the layers of the generator to 5. This was done due to recommendations made by different blog

postings made online to this type of problem. [7] (I had already been adding noise to the discriminator labels, so this wasn't an option.)

The error was persistent, although these steps did help to delay the step in which this sudden loss change occured. Upon reviewing the output of the generator we can understand further what is happening. After training, I fed in a noise vector similar to training to the generator to show the output. Nearly all of the floating point integers representing the vector space of each word appear to be close to zero. My assumption is that his is because of the padding used to standardize each poem to become the same length, as all but one poem needs padding (the longest poem is already at max length).

As well, in order to take the 177 x 50 size output of the generator and convert it into 177 x 1, I had to take the vector in the GloVe dictionary with the smallest euclidean distance from each row, or word vector. These appear to be the same word, as all the vectors are closest to the zero vector. Looking at the output of the shortest distances, I noticed that the distance steadily grows until saturating at about 0.58 near the midway point in the poem, shown below.



This further confirms my belief that the generator is essentially predicting zero vectors since most of the poems are made up of quite a lot of padding. Because these vectors are all close to zero, it kept outputting the same word at each step, the closest vector to zero, which obviously doesn't resembling a poem.

Conclusion

Unfortunately, the approach of padding and feature engineering ended up to be detrimental to the final outcome. Nevertheless, I still have laid the foundation for what could potentially be a way of generating time-series data using a mixture of GAN and RNN architecture. The mission of being able to produce legible poetry was not a success, but the larger mission to become more fluent in training and iterating the approach of cutting-edge neural networks for time-series data was a success.

For future iterations of this approach, I would recommend changing how the embedding of the integers is calculated. For example, I was unable to incorporate the Embedding layer from keras that would have resulted in adding the 50 dimensional features to each word. This could potentially use built-in methods to avoid calculating the padding, which would result in a more poem-like output.

References

[1] Goodfellow, Ian; Pouget-Abadie, Jean; Mirza, Mehdi; Xu, Bing; Warde-Farley, David; Ozair, Sherjil; Courville, Aaron; Bengio, Joshua (2014). "Generative Adversarial Networks".

[2] Umezawa, Keisuke (2018). “DCGAN: Generate the images with Deep Convolutional GAN”.

[3] Isnoor. (2017). *Poetry Analysis with Machine Learning*. Retrieved from: https://www.kaggle.com/ishnoor/poetry-analysis-with-machine-learning

[4] hsankesara. (2017). *Mr. Poet*. Retrieved from: https://www.kaggle.com/hsankesara/mr-poet

[5] Pennington, Jeffrey; Socher, Richard; Manning, Christopher D. (2014). “GloVe: Global Vectors for Word Representation”.

[6] Chollet, François (2018) Introduction to generative adversarial networks. Retrieved from: https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/8.5-introduction-to-gans.ipynb

[7] Chintala, Soumith (2017) G loss increase, what is this mean?. Retrieved from: https://github.com/soumith/ganhacks/issues/14