Text Generation using Generative Adversarial Networks

David Levi
The Cooper Union for the Advancement of Science and Art
ECE 471
levi2@cooper.edu

Cardy Wei
The Cooper Union for the Advancement of Science and Art
ECE 471
wei2@cooper.edu

Abstract

In this paper, we introduce a new way to generate text using Generative Adversarial Networks. In the past, text generation in neural networks has been done using sequential networks, where the word generated is conditioned on previous words. This is generally done using maximum likelihood during training in a method called teacher forcing. However, using this method, there have been problems where the network has been unable to create meaningful sentences due to previous sequences that it has never encountered during training time. Additionally, the network is often unable to generalize well. To alleviate this issue, we look to a different type of neural network architecture, the Generative Adversarial Network (GAN). The advent of the use of GANs in text generation is a recent development and has had various difficulties in working with text. GANs require differentiation to encourage backpropagation of gradients, so it does not work very well with words, which are discrete values. However, GANs work very well on image generation, and based on this, we propose a method of training GANs using word embeddings structured like image pixels. This method is unique in the sense that it does not require any reinforcement learning to allow the GAN to train. Therefore, our model can work like traditional text generation techniques and base generated sentences on training data without any previous knowledge of the language.

1 Introduction

Generative Adversarial Networks, or GANS, are architectures used to generate data based on adversarial training. A GAN consists of two networks, a Generator and Discriminator, each learning off of the other. In this model, the Generator is fed random noise from a latent space and produces synthetic data that the Discriminator will have to learn to differentiate from real data. As the Discriminator learns to distinguish real data from false data, the results are sent back to the Generator and improves on it to produce even higher quality data. This battle for dominance between the two networks should eventually result in a Generator that theoretically could generate data that is indistinguishable from real world data. However, GANS rely on differentiation to propagate gradients. This means that sampling from the generator must be used with some form of continuous function, but text is discrete. This leads to an obvious issue and is one of the reasons that GANs are generally used on images rather than text. Discrete inputs into the generator would make the training of the net unstable [2]. One previous method to deal with discrete data in GANs is to continuously reinforce these discrete inputs [5]. However, this would require some outside knowledge of the language being generated in order to work. Instead, we propose a new method of generating text with GANs using word embeddings. GANs are known to work very well with images, so our work simulates this by using word embeddings in a structure very similar to pixel matrices. This method also does not require pre-training or previous knowledge of the language to condition on.

2 Related Works

Research into using GANs for text generation have recently come across some breakthroughs, most recently in the forms of MaskGAN [5] and SeqGAN [4]. SeqGAN trains its Generator using policy gradients and its discriminator is a CNN-based model. Both the Generator and the Discriminator are pre-trained on real and fake language data beforehand and use Monte Carlo rollouts during training. There has also been work done using RNNs without the pre-training step [4].

On the other hand, MaskGAN generates text to fill in blank spaces between phrases. It creates these blanks by using masks over certain streams of text to create hidden tokens. MaskGAN then conditions the seq2seq model on non-redacted text and real text that fill in the blanks. It then autoregressively goes through text it has generated to continue the training, as is common in many standard language models. The Discriminator additionally uses supervision signals as well to assist in training. If the Generator generates a strong sequence of text except for one token, the Discriminator will recognize this and update the Generator accordingly. While both of these methods have resulted in relatively good results, both require previous knowledge of the generated language which may create a hindrance for generating some lesser known languages.

3 Methods

One of the difficult aspects of GANs is the formatting necessary to have efficient and complete training. Our main goal was to treat each input sentence like an image, which GANs are known to be very good at producing. After formatting the sentences, we created an algorithm using bigrams to smooth the sentence and provide some filler words that may not have been learned by the Generator.

3.1 Sentence Formatting

To accomplish the task of text generation using a normal GAN with no reinforcement learning we needed to find an efficient and intuitive way of expressing the input and output sentences. To do this we used an online news dataset as our training data. We then used the python NLTK library to split the documents into sentences and tokenize them. Once we had each sentence split into lists of words we turned each one into its proper embedding vector using the open source Glove embeddings [7]. Each list was then either cut off or padded depending on a certain max sentence length parameter, which for the results listed in this paper was set to 15. Because of GANs extensive use with images we then formatted each list to be similar to the layout of an image. To do this, we turned the list into a matrix with dimensions [max-sentence-length x n-dim] where max-sentence-length was the length of each input sentence, and n-dim was the size of the embedding dimension. These new "word images" were then used as the training data and input directly into the Discriminator.

The input into the Generator was a noise vector of size [1 x 100] consisting of numbers from 0-10. We tried modifying the range of numbers in the noise vector, but the results were unchanged.

One of the main issues with GANs are their inability to deal with discrete inputs. The main reason that GANs deal so poorly with discrete inputs is because you cannot find the gradient of a discrete set. So, to get around this issue, we transformed the output back to its closest word embedding. To implement this efficiently, we used the Gensim library and the open source Glove embeddings to search for similar word vectors to the output. Each output from the Generator was then turned into a matrix with the proper dimensions before going into the Discriminator. After training, the output from the Generator was turned into sentences by finding the most similar word to each row in the output image.

3.2 Bigram Method

Our goal for this project was to see if GANs could generate text. However, we knew that the text generation would be limited in scope. Therefore, we aimed to have the GAN generate sequences of words that sound similar to a sentence and decided to smooth out the result. To do this we created an algorithm using bigrams from an open source corpus [8]. The algorithm looked at sets of two words and tried to see if those words logically came after each other by looking at a corpus of bigrams. If they did not, a word was added between them that is common to both words. The pseudocode for the algorithm is pictured below:

Algorithm 1 Bigram Method

```
procedure BIGRAM($sentence, bigramList$) ▷ Returns modified sentence using Bigram Method
    $NewSent$ ← Empty string
    $words$ ← words in *sentence*

    for *wordIndex,word* in length of *words* do
        $NextWord$ ← words[wordIndex + 1]
        $dictInd$ ← 0
        $flag$ ← True
        $NewSent$ ← *NewSent* + *word* + " "

        if *NextWord* in *bigramList[word]* then Continue
        while flag do
            if *dictInd* $\geq$ length of *bigramList[word]* then
                *flag* ← False

            if *word* in *bigramList[word]* then
                *NewSent* ← *bigramList[wordIndex][dictInd]*$+$" "
                *flag* ← False


        $dictInd$ ← *dictInd* + 1

        Return $NewSent$
        =0
```

The Bigram method accomplished a few things. Firstly, it added some "filler" words to the sentences such as words like "and, the, it, etc." These types of words were lacking in the output of the Generator because, as stated before, the Generator attempts to maximize the loss of the Discriminator which is most easily accomplished by primarily producing nouns and verbs. This method was also able to create a better flow in the sentence structure. Essentially it creates a sliding window across the sentence that provides some sort of context to the sentences. Therefore, when reading the sentences modified using this method, it seems like it is built up of small chunks of sentences that make perfect sense, but when looked at from a wider range, the context usually changes. Examples produced from the bigram method are shown in the results section.

4 GAN Architectures

The two main GAN architectures used were a very simple GAN [6] and the Wasserstein GAN [1] with gradient penalty also known as WGAN-GP [3]. The first GAN is composed of a small sigmoid output generator and two convolutional layers in the Discriminator. The Wasserstein GAN is composed of a 10 layer convolutional Generator and an 18 layer convolution discriminator. What makes this GAN special is the use of the Wasserstein loss, which encourages the smoothing out of outputs in a given metric. In addition, this implementation of WGAN uses gradient penalty which assists in training stability without losing much information.

Table 1: Text Generation Output after 0 epochs

Type of GAN	Generated text
Simple GAN	"Palladium sandhya 120/80 swivels awakens zinho unfeeling markakis build oxidoreductase devgan accredits quaternary earner urinalysis"
WGAN	"Academie selzer tamora adamses sawhorses haitian recant sec legwold staunton alvin profiled horrorfest zk wagram"
WGAN + Bigram Method	"Academie selzer tamora adamses sawhorses haitian recant sec legwold staunton alvin profiled horrorfest zk wagram"

Table 2: Text Generation Output after 100 epochs

Type of GAN	Generated text
Simple GAN	"Four years ago while everything putting wanted directly app a mobile app now rampages"
WGAN	"1989 Jefferson Portland blazers games considering NBA chance shortfall Cardinals attendance fans quickly seasons serving"
WGAN + Bigram Method	"1989 Jefferson Portland blazers games after considering NBA chance the shortfall Cardinals attendance and fans and quickly the seasons and serving"

5 Results

The results are organized into 3 different parts: the simple GAN, the WGAN, and the WGAN after the bigram method has been applied. Table 1 shows the results after 0 epochs of training. As can be seen from the results, most words are gibberish and the bigram method had no effect on the output. There also does not seem to be any difference between the simple GAN and the WGAN. Table 2 shows the results after 100 epochs of training. We tried training for more epochs, but the results were not any better. Clearly it can be seen that better sentences are generated with an increase in the number of epochs. It also seems that the WGAN's generated sentences are a bit more complex and diverse than that of the simple GAN's sentences. Additionally, after the bigram method is applied, the sentence gains some readability. Because the sentence mostly contains nouns and verbs, the bigram method adds some additional words to help the sentence flow.

The results show that these networks were indeed able to recognize and learn context using word embeddings and have generated sentences with words that make sense when seen with each other. However, there seem to be several problems with our results. Firstly, it seems that the text is mostly generated on the immediately preceding phrase, so the sequence of text is not entirely coherent. In addition, there is no way for this model to smooth out the sentence and create a common sentence flow. The model does not "know" what grammar is and therefore cannot utilize it. In addition, looking at the sentences the model generated during training, it can be seen that it creates many sentences that are similar by category. This is known as the unimodal problem, and is a common problem amongst GANS, where the network begins creating data that is solely similar to data it has been trained on. While observing the loss function of the model, we see that the loss of the Generator rises very rapidly and the loss of the Discriminator decreases very rapidly. This could signal that the model has been trained too quickly and has not been refined very well, which could be resolved by using different, more complex models or adding more levels of regularization in the architecture.

6 Future Work

When testing and training the GANs we ran into a few issues. Firstly, the continuous nature of GANs is still a big roadblock when trying to deal with discrete values like words. In our method we chose to treat each sentence as continuous numbers until we changed them back into discrete words at the output. This led to sentences that made sense when looked at through a small window, for example a few words, but did not make as much sense when looked at as a whole. For example, certain sentences would start off on one topic and then slowly morph into a different topic. We believe that this is because the Generator found that the way to best fool the Discriminator was to create sentences that contextually sounded similar to the training data, even if the sentence was dealing with two different topics. We attempted to get around this by using the Bigram method but that was only a temporary fix and could use some improvement in the future.

Additionally, one common issues with GANs is the potential for unimodal outputs. Unimodality is when the output is generally all the same. This is a common problem with GANs and arises when the Generator finds a way to fool the Discriminator and continues to generate the same type of output no matter what the noise vector is. In our case, each time we trained the GAN the output was a different topic. However, each generated sentence looked very similar and many used similar words. We believe that with all of the ongoing research into the topic of unimodality in GANs this will be an easy fix in the future.

7 Conclusion

In this paper, we discussed a new method for text generation using Generative Adversarial Networks. This method is superior to some previous GAN text generation methods because it does not rely on any previous knowledge of the language to be generated. While other methods employ mostly reinforcement learning, which requires a metric based on the validity of a sentence, our method generates text using knowledge from the training data only. The results were taken from the Generator and turned back into discrete words using the Gensim library. Most of the sentences consisted of the same sentence topics, for example, after one set of training all of the results would talk about technology, with many of them using similar sentence structures. Additionally, many of the sentences did not read as well as typical English sentences. To get around this issue, we used bigrams to smooth out the sentences and make them a bit more readable. Overall, we were very happy with the results we have gotten, but there is still a lot more to explore in the field of GANs and text generation.

Acknowledgments

We would like to thank Chris Curro for all of his help and encouragement throughout this project as well as for teaching us and expanding our knowledge of deep learning this semester. We would also like to thank all of our fellow classmates for their constructive criticism which led us to some unique solutions to our problems, many of which we would not have seen otherwise.

References

[1] Charlie Frogner, Chiyuan Zhang, Hossein Mobahi, Mauricio Araya-Polo, Tomaso A. Poggio: *Learning with a Wasserstein Loss.* CoRR abs/1506.05439 (2015)

[2] Desai, Utkarsh. *"Keep Calm and Train a GAN. Pitfalls and Tips on Training Generative Adversarial Networks."* Medium.com, Medium, 29 Apr. 2018, medium.com/@utk.is.here/keep-calm-and-train-a-gan-pitfalls-and-tips-on-training-generative-adversarial-networks-edd529764aa9.

[3] Ishaan Gulrajani, Faruk Ahmed, Martín Arjovsky, Vincent Dumoulin, Aaron C. Courville: *Improved Training of Wasserstein GANs.* CoRR abs/1704.00028 (2017)

[4] Lantao Yu, Weinan Zhang, Jun Wang, Yong Yu: *SeqGAN: Sequence Generative Adversarial Nets with Policy Gradient.* AAAI 2017: 2852-2858

[5] William Fedus, Ian J. Goodfellow, Andrew M. Dai: *MaskGAN: Better Text Generation via Filling in the* —-. CoRR abs/1801.07736 (2018)

[6] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, et al. *Generative adversarial nets.* In NIPS, 2014.

[7] Pennington, Jeffrey. Single-Link and Complete-Link Clustering, nlp.stanford.edu/projects/glove/.

[8] "N-Grams Data." N-Grams: Based on 520 Million Word COCA Corpus, www.ngrams.info/.