Reimplementing TextGAN

Gus Smith (hfs5022)

April 30, 2018

Abstract

In this final report, I discuss the implementation and results of my term project. Note that the title of this term project has changed; due to a number of issues encountered along the way, I had to reduce the scope and goals of my project. These changes will be discussed in detail.

Modification History

date	changes
March 16, 2018	Initial draft.
April 6, 2018	Midpoint. • Added section 4.1, detailing generation of word embeddings. • Added section 4.6, detailing the framework I will use and other technical details of my implementation. • Corrected figure 6. • Updated timetable.
April 30, 2018	Final report. • Added abstract. • Updated existing sections to reflect change in scope. • Restructuring: primarily, the “Methods” section is now the “Implementation” section. • Added results and conclusions sections.

1 Introduction

Problem statement. Generating realistic text based on a corpus of existing text is an open problem with many attempted solutions. Generative Adversarial Networks, or GANs, are a type of network originally developed to improve the generative process. Until recently, GANs had not been applied to discrete data like text; however, recent work has developed a number of ways to use GANs to generate realistic sentences. The use of GANs for generating text must be further investigated, and the creative power of GANs should be improved upon and made more useful.

1.1 Goals

The initial goals of this project were:

1. Reproduce results from Zhang et al. [17] on a new dataset. This primarily involves reimplementing the TextGAN model presented in the paper.

2. Extend the TextGAN model, with the goal of being able to generate text with specific characteristics.

Along the way, I began to realize that I bit off more than I could chew with these goals. My first mistake was in assuming that TextGAN would be easy to implement from scratch; there were a number of fascinating implementation details which first needed to be understood and implemented.

To compound on this first problem, once TextGAN was implemented, I discovered that I had underestimated both the training time and quantity of data I would need to get good results.

In my time remaining, I thus decided to focus on solidifying and understanding my TextGAN implementation, instead of attempting to modify it.

Though I reduced the scope of my original project, this should not indicate that this term project is lacking in content. In fact, it's just the opposite; by implementing TextGAN, I discovered (and will write about) a number of fascinating things which were not touched on in class. In addition, I have interesting results to present in this paper.

The revised goal of this project is thus to reimplement TextGAN from scratch using lower-level TensorFlow APIs, with the goal of gaining real understanding. I will then report in detail on the nuances of this implementation. Finally, I will run the implementation and analyze the results.

2 Related Work

Generative Adversarial Networks (GANs) were originally presented in Goodfellow et al. [4]. The authors developed the concept of a GAN to address the question of how deep learning can be used for *generative* models, rather than *discriminative* ones. To do this, GANs still leverage the discriminative power of deep learning. A GAN is actually two networks combined: a generator network and a discriminator network. The two networks are trained in a sort of competition—the generator is trained to fool the discriminator, and the discriminator is trained to better detect generated data. Technical detail on GANs is presented in section 4 section.

Recently, GANs have been adapted to work for text generation. Zhang et al. [17] just recently presented a model which they call TextGAN, which combines a number of previous ideas for text-generating GANs from the literature and improves upon them. TextGAN will be the core of this term project. The primary contribution of TextGAN is a modification of the standard optimization objective function used in Generative Adversarial Networks. The implementation of TextGAN will also be described in more detail in the methods section.

One of the fundamental steps in building TextGAN is learning fixed-length representations of sentences—that is, mapping a potentially variable-length sentence into a fixed-length feature space. A number of works have proposed ways to do this[7, 8]. Here, we specifically bring attention to [3], not only because it is a unique and interesting approach to sentence representation, but also because this exact approach will be used during our pre-training for TextGAN. Gan et. al. use an unsupervised method to learn sentence representations, by first encoding sentences into fixed-length vectors, and then generating sentences from these encodings. This *autoencoder,* as they call it, is trained to generate the correct sentence from a given word embedding.

3 Data

In this project, I use two different datasets. The first dataset is a text corpus which I have generated on my own. The corpus comes from a group chat which I am a member of. I exported all messages from a three year period. The second dataset is the SIGKDD 2003 "KDD Cup" arXiv paper dataset [12]. This dataset contains the abstracts from physics papers from the 1990s and early 2000s. In this report, I focus on the arXiv dataset, as it is much larger and produced better training results. It is also one of the datasets used in [17].

The data is first pre-processed. Each message or abstract is split into sentences based on punctuation; each sentence is a separate datapoint. Sentences are then filtered—for example, in the chat log dataset, sentences containing URLs are thrown out. The specific intent is to throw out sentences whose meanings would be lost after "cleaning" them—for example, removing the URL from a chat message may make the message meaningless, and thus not a valid datapoint for training. After coarsely filtering out sentences, the remaining sentences are cleaned by removing the "clutter": emojis and emoticons in the chat log, or LaTeX formatting in the abstracts, for example.

There are a total of about 150,000 messages in the chat log dataset; however, after cleaning, only about 30,000 sentences remain. The arXiv dataset produces about 130,000 sentences after pre-processing.

Once the data is pre-processed, the most common 6000 words are put into a dictionary, mapping word strings to numeric IDs. Any words not among these 6000 most common words are assigned the special token UNK (unknown). Using this dictionary, sentences are then converted into lists of word IDs. Another special token, END, is also appended at the end of each sentence.

Permission has been obtained from those members in the group chat who are willing

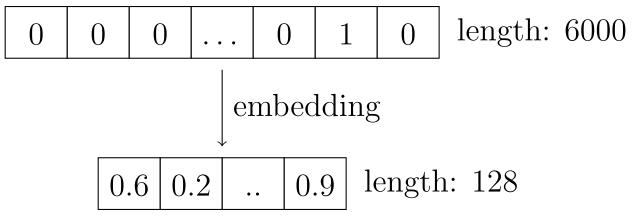

Figure 1: The process of word embedding.

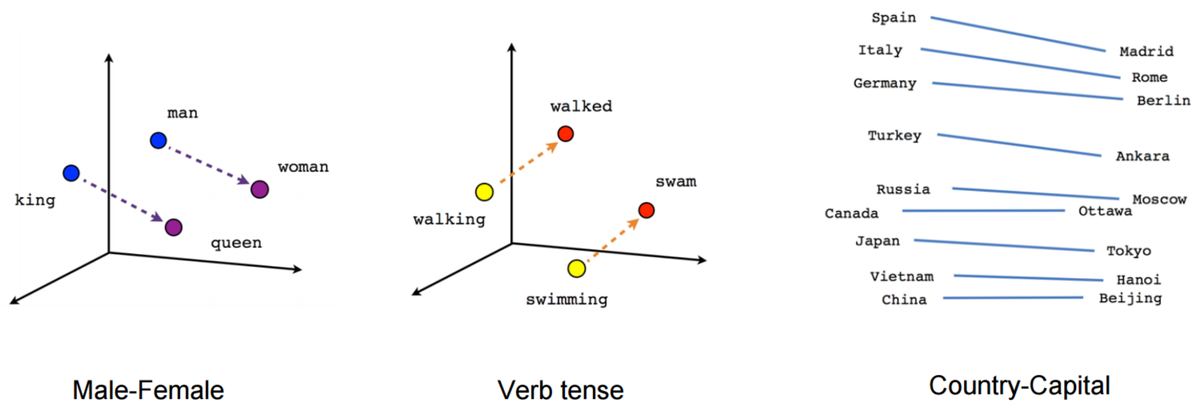

Figure 2: In the embedded space, words show interesting spatial relationships. Reproduced from [15].

to participate in this experiment. For the sake of privacy, I will not release my chat log dataset; I will just release the arXiv dataset. As such, we focus on the arXiv dataset throughout this report.

4 Implementation

In this section, we detail the process of implementing TextGAN, with thorough explanations of each component of the network.

We begin first by explaining in 4.1 the concept of *word embeddings,* a necessary component of most modern language networks. With that out of the way, we then cover the TextGAN implementation in sections 4.2, 4.3, and 4.3.

4.1 Word Embeddings

When feeding sentence data into a network, we need to first decide how we are going to represent words. Generally, we think of input into a network in terms of its *features*—for instance, a simple 10×10 image can be represented as a vector having 100 features. How do we do the same for words and sentences?

One easy way is to encode our vocabulary in a one-hot fashion. For example, in a vocabulary of size 6000, each word would be a vector of 6000 features; each feature but one would be equal to 0. However, this is not ideal, as the data is both high-dimensional and sparse.

Word embeddings solve this problem. As shown in figure 1, word embedding is the process of mapping these high-dimensional, sparse vectors into a lower-dimensional, denser space [15]. However, the dimensionality reduction and density increase are not the only advantages of word embedding. As explained in [15], the process by which we embed words into this lower-dimensional space embeds significance into the resulting vector. For example, words which are nearby in the embedded space often share characteristics (e.g. parts of speech). Interestingly, more complex information is also embedded into the features. For example, as shown in figure 2, it is common to see that the same translation is used to move between related pairs of words—that is, the translation to get from "king" to "queen" is nearly the same translation as the one to get from "man" to "woman".

The state-of-the-art methods for training word embeddings treat it as an *unsupervised* learning task. The core concept behind these methods is to project words into the embeddings space in a way that maximizes the ability to reconstruct snippets of sentences given just a few word embeddings. This is generally implemented in one of two ways: the Skip-gram and Continuous Bag of Words models. These two models are very similar. The Skip-gram model adjusts word embeddings to maximize the ability to predict context (surrounding) words when given the source word, while the Continuous Bag of Words model attempts to maximize the ability to predict the source word given the context words.

In this project, I focus on the Skip-gram model. Though I have implemented both models in my project, my datasets are ultimately embedded using Skip-gram. This is primarily because Skip-gram is the model implemented in [15], which is the original tutorial I was following. I have implemented the Continuous Bag of Words model also, as it is apparently better for smaller datasets; however, I did not have time to train my datasets using this method as well.

We now describe our implementation of Skip-gram, which is based on Mikolov et al. [10] as presented in [15]. A common method for such predictive language tasks is Maximum Likelihood. For example, when predicting the next word given context h, we can find the word w_t which maximizes $P(w_t|h)$. This could be implemented by "scoring" each word against h (for example, by simply taking a dot product) and then computing a softmax over the results, which would give us $P(w_t|h)$ for each w_t.

Using the above method, we would need to score each word in the vocabulary w_t against h at each training step, so that we can generate the full distribution $P(w_t|h)$. However, Mikolov et al. [10] presents a simplification which greatly reduces the amount of computation needed. Mikolov et. al. realized that a full probabilistic model is not needed when performing the Skip-gram optimization task. We do not need to predict the exact set of context words given the source word; instead, given a source word and query word, we need only be able to determine whether the query word is truly a context word, or if it is noise. in this way, we no longer compute the full distribution $P(w_t|h)$, but instead the binary logistic regression $Q_\theta(D = 1|w_t, h)$ (using the notation of [15]) where θ represents our word embeddings.

With this simplification, the number of calculations needed per training step can be reduced. Specifically, Noise-Contrastive Estimation (NCE)[6] can be used in a simplified form. At every training step, we are given a training datapoint (h, w_t), and we draw k noise words $\tilde{w}$. We then maximize

$$J_{NEG} = \log Q_\theta(D = 1|w_t, h) + k\mathbb{E}_{\tilde{w}\sim P_{noise}}[\log Q_\theta(D = 0|\tilde{w}, h)].$$

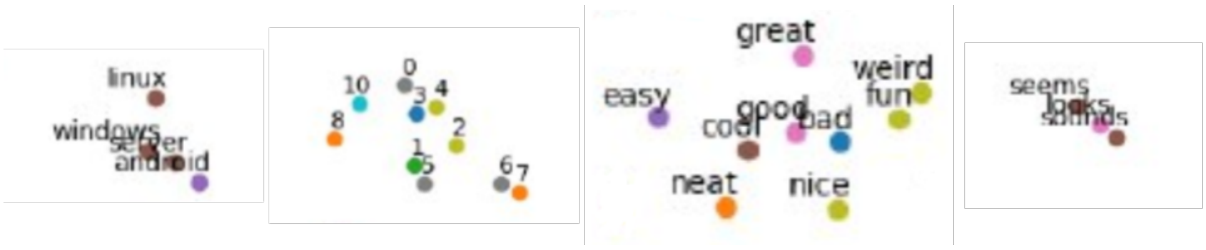

Figure 3: A few interesting groupings in the chat log word embeddings.

Now, instead of having to compute an entire distribution over all words in the vocabulary, we simply need to compute $Q_\theta(D = 0|\tilde{w}, h)$ for each of our k noise words. In our implementation, the size of our vocabulary is 6000, while k is 64; thus, we run two orders of magnitude less operations per training iteration. Mikolov et al. [10] showed that even lower values of k can be used; for large datasets, values as low as 5 still trained successfully.

We can visualize the results of our word embeddings using the t-SNE dimensionality reduction technique (also described in [15]). Figure 3 shows a number of interesting results of the word embeddings from the chat log dataset, while figure 4 shows the word embeddings for the arXiv dataset.

4.2 TextGAN Overview

To explain the implementation of TextGAN, we first give an introduction to GANs, originally presented in [4]; we will then describe how TextGAN modifies and improves this model for text generation.

A GAN is composed of two networks: a generator and a discriminator. The input of the generator network is a noise value z, and we define its distribution as $p_z(z)$. Generally, uniform or Gaussian noise is used. The input noise is then passed into the network; the output of the generator is a synthetic datapoint. We represent this transformation as $G(z; \theta_g)$, where θ_g are the generator's parameters. G can be implemented in a number of ways. The original GAN paper used a simple multilayer perceptron network, while TextGAN will use a more complicated network for generating text. The discriminator network, on the other hand, is a standard network which takes a datapoint as input and produces a classification as output. In this case, the discriminator classifies the datapoint as genuine or synthetic.

Training the two networks then occurs iteratively, alternating between the two networks. In the original GAN paper, the optimization objective is the same for G and D, and is written as

$$\mathcal{L}_{GAN} = \mathbb{E}_{x\sim p_x} \log D(x) + \mathbb{E}_{z\sim p_z} \log[1 - D(G(z))]. \tag{1}$$

This objective is maximized with respect to D. When maximizing with respect to D, the first term causes D to maximize the expected probability of an input being classified as genuine, which results from inputting a value x from the genuine data (whose distribution is represented by p_x). Similarly, the second term causes D to maximize the expected value of $1 - D(G(z))$, which is the probability that the synthetic input $G(z)$ will be classified as synthetic.

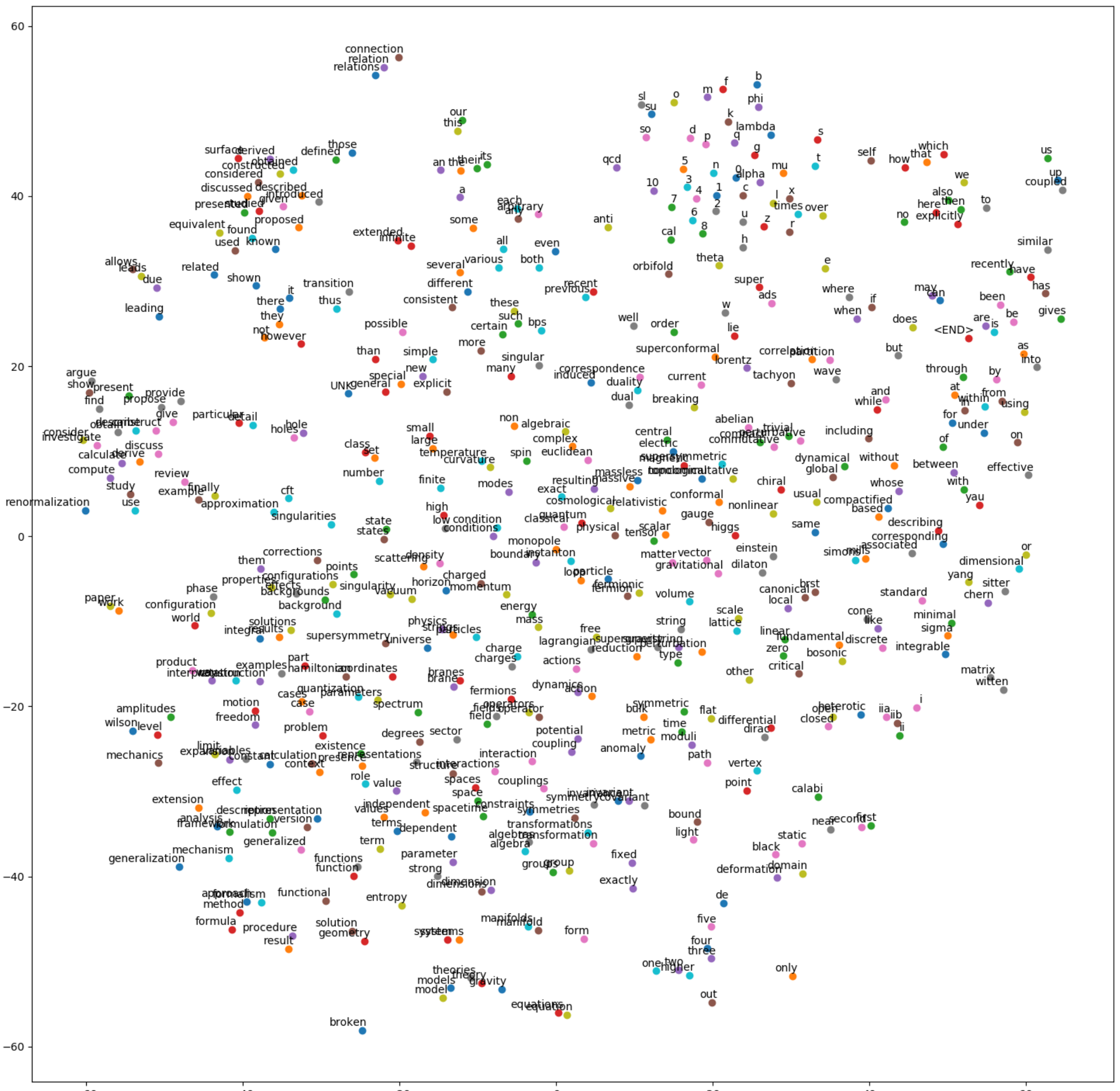

Figure 4: Word embeddings visualized for the arXiv dataset.

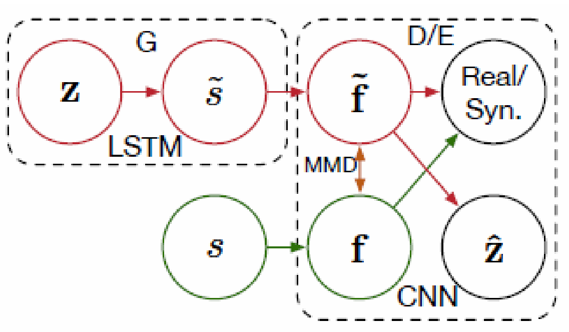

Figure 5: The general structure of TextGAN. Reproduced from Zhang et al. [17].

this	is	a	sentence
0.1	0.4	0.5	0.4
0.3	0.1	0.9	0.3
0.7	0.3	0.3	0.0
0.7	0.2	0.5	0.5
0.8	0.9	0.2	0.2

this	is	another	sentence
0.1	0.4	0.2	0.2
0.3	0.1	0.1	0.4
0.7	0.3	0.6	0.3
0.7	0.2	0.0	0.0
0.8	0.9	0.1	0.5

Figure 6: Example of two different sentences represented as matrices of word embeddings; in this figure, each word is a column.

When training G, this objective is minimized. Only the second term depends on G in this minimization. Minimization of this second term trains G towards producing output which gets misclassified by D.

As its discriminator network, TextGAN uses a standard Convolutional Neural Network (CNN). The network takes sentences (possibly padded) as input, and performs convolutions over them, resulting in a final classification of the sentence as real or synthetic. We will describe the discriminator in more detail in section 4.3.

The generator network is a Long-Short Term Memory (LSTM) network, which is a type of Recurrent Neural Network (RNN). The generator network takes a sample of noise as input, and generates a synthetic datapoint as output. We describe the generator in more detail in section 4.4

Lastly, TextGAN is trained using a modified objective function, which is described in section 4.5.

An overview of TextGAN is shown in figure 5, which is reproduced from Zhang et al. [17]. We will refer back to this figure as we describe the implementation of each of its components.

4.3 TextGAN Discriminator

For the discriminator network D, TextGAN uses a CNN. This network expects as input sentences represented as matrices of word vectors, examples of which can be seen in figure

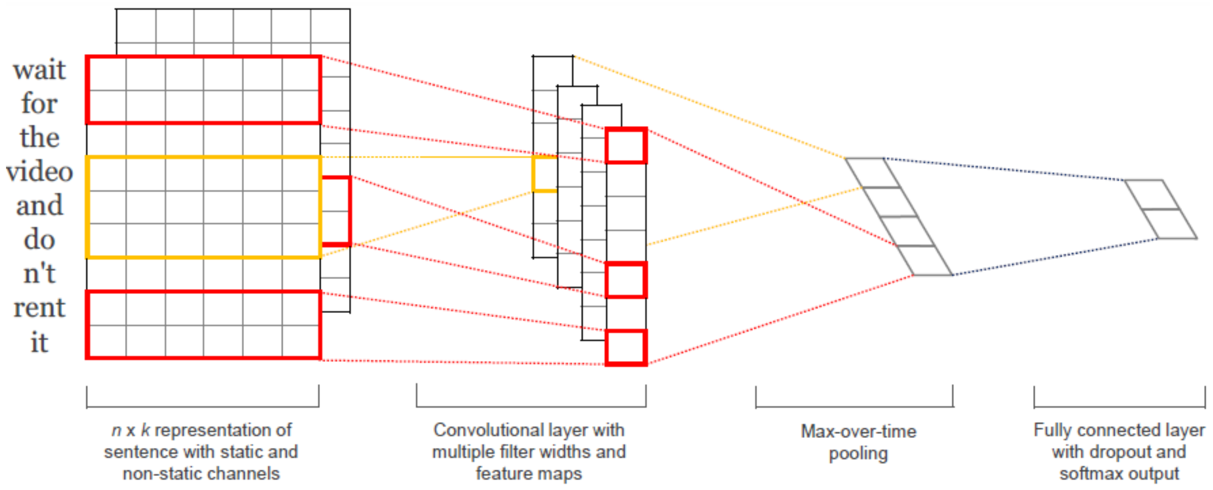
wait
for
the
video
and
do
n't
rent
it
n x k representation of sentence with static and non-static channels
Convolutional layer with multiple filter widths and feature maps
Max-over-time pooling
Fully connected layer with dropout and softmax output

Figure 7: The CNN used by TextGAN, originally presented by Kim [9]. Reproduced from [9].

6. The discriminator network then performs convolutions over these inputs, much like what we are used to in CNNs for images—the primary difference being that each filter has the width of an entire row, and thus produces activations that are single columns. The final output of the network is a classification as genuine or synthetic, in addition to a fixed-length encoding of the sentence $\hat{z}$, which we will describe in a moment.

We will now describe the structure of the network in more detail. The network was originally presented in Kim [9]. I have reproduced figure 7 from [9] as a helpful visual aid.

The first layer of the network performs convolution over sentences. Convolution is performed over a window of h words. Convolving a filter over every h words in a sentence produces a vector; the max of this vector is then taken. By taking the max of the vector, we achieve two important results:

1. The vector which results from the convolution is of variable length, depending on the length of the sentence. By then taking the max, the result is size 1, and thus independent of the length of the sentence.

2. We are able to find the feature which the filter represents at any location within the sentence.

In practice, we use $h \in \{3, 4, 5\}$. Furthermore, each value of h has 300 total filters. Thus, after convolving and pooling each filter over a sentence and concatenating the results, we will have a length 900 vector. Zhang et. al. refer to this vector as the *features* f of sentence s (labeled $\tilde{f}$ if the features come from a generated sentence $\tilde{s}$). These features can be seen in figure 5.

At this point, the network splits, and produces two different outputs from f (or $\tilde{f}$): first, a binary classification of the sentence, and second, a fixed-length encoding $\hat{z}$ of the sentence. The binary classification is generated by passing the f through two fully connected layers of size 200 and 2, respectively. The encoding is also generated by passing f through two fully connected layers, but in this case, both are of size 900. Between each fully connected layer on both branches, the sigmoid activation function is applied. Softmax is applied to the classification output, and tanh activation is applied to the encoding output to produce $\hat{z}$. The need for $\hat{z}$ will be explained in detail in sections 4.4 and 4.5.

4.3.1 Pretraining

Pretraining the discriminator is essential for helping TextGAN train later on. In my early experiments without pretraining, simply connecting the generator to the discriminator and attempting to train resulted in no productive training occurring.

The goal of the discriminator in TextGAN is to distinguish generated from real sentences. Zhang et al. [17] pretrains the discriminator by devising a task that, in theory, is similar to the generated vs. real determination task. Specifically, the task is to distinguished real sentences from "tweaked" sentences, where tweaked sentences are simply real sentences from the dataset with two words randomly swapped.

We pretrained the discriminator until we achieved an average accuracy of 75% on the validation set on the tweaked/not tweaked task.

4.4 TextGAN Generator

The generator network is an LSTM which takes a fixed-length vector z—the so-called "latent code vector", in Zhang et. al.'s terminology—and translates it to a synthetic sentence $\tilde{s}$. The latent code vector z represents a latent encoding of a sentence, exactly as the encoding $\hat{z}$ produced by the discriminator represents the encoding of the input sentence.

A sentence $\tilde{s}$ is generated according to the distribution

$$p(\tilde{s}|z) = p(\tilde{w}^1|z)\prod_{t=2}^{T} p(\tilde{w}^t|\tilde{w}^{<t}, z)$$

where $\tilde{w}^t$ are the generated tokens. Each word is generated by maximizing $p(\tilde{w}^t|\tilde{w}^{<t}, z) = p(\tilde{w}^t|h_t)$, where h_t is an encoding of the history which passes through the LSTM.

In practice, this is implemented as follows. First, the initial h_1 is generated from z as $\tanh(Cz + C_b)$, where C and C_b are parameters of the network. Given h_t, word $\tilde{w}^t$ is generated as $\text{argmax}(Vh_t + V_b)$, where again, V and V_b are parameters of the network. V is a weight matrix that, when multiplied with h_t, creates a distribution over all words in the vocabulary. The history encoding h_t is updated to h_{t+1} according to the LSTM cell's input, output, and forget gate parameters. We continue down the chain of the LSTM.

Note that the expression $\text{argmax}(Vh_t + V_b)$ is not differentiable in any sane manner, and thus, it is impossible to do gradient descent (or other gradient-based optimization) through an argmax. Zhang et. al. mention a number of solutions to this problem. First, one could use gradient estimation approaches, such as the REINFORCE algorithm of Williams [16]. However, Zhang et. al. pursue a simpler solution by employing a "soft argmax" function:

$$W_e\text{softmax}(Vh_{t-1} \odot L).$$

This function approximates argmax, while also being differentiable. Note that, as $L \to \infty$, this approximation becomes argmax.

4.4.1 Pretraining

Like the discriminator, it is important to pretrain the generator. Pretraining for the generator is more complicated, however. To pretrain the generator, we train the autoencoder presented in Gan et al. [3].

The autoencoder has the opposite structure of our GAN network: it is a CNN discriminator which feeds into an LSTM generator. The goal of the autoencoder is to encode each sentence into a vector $\hat{z}$, such that when $\hat{z}$ is passed into the generator, the original sentence is likely to be produced.

Loss is computed during training as follows. First, we encode a sentence in the discriminator. Then, we pass that encoding into the generator network. At each step of the LSTM, we compute the probability of the correct word being generated. We sum the negative logs of these probabilities, and attempt to minimize the resulting quantity.

4.5 Training TextGAN

To train these networks, TextGAN introduces new optimization objectives:

$$\mathcal{L}_D = \mathcal{L}_{GAN} - \lambda_r \mathcal{L}_{recon} + \lambda_m \mathcal{L}_{MMD^2} \tag{2}$$
$$\mathcal{L}_G = \mathcal{L}_{MMD^2} \tag{3}$$

Where $\mathcal{L}_{GAN}$ is the original GAN objective. We will now describe the rest of these objectives in detail.

$\mathcal{L}_{recon}$ is the Euclidean distance between the original noise value z which the generator uses to create the sentence, and the reconstructed noise value $\hat{z}$ which is produced by the discriminator.

$\mathcal{L}_{MMD^2}$ is the Maximum Mean Discrepancy metric, which is described in [5]. MMD is a measure of distance or dissimilarity between two sample sets $\mathcal{X}$ and $\mathcal{Y}$. In this case, $\mathcal{X}$ and $\mathcal{Y}$ are the features f and $\tilde{f}$ produced by the CNN for genuine and synthetic inputs, respectively; thus, the MMD measures dissimilarity between the extracted features. We use an implementation of MMD from GitHub[13] developed by Dougal Sutherland and originally presented in [14].

As described in [17], the model has a number of hyperparameters. The hyperparameters λ_m and λ_r in the loss function simply weight the new objectives. We set these parameters by trial and error based on how the model trains. Furthermore, we clip our gradients at 5; this is needed in the early stages of training, as the largely untrained parameters will produce infinite gradients. The MMD metric has a bandwidth parameter σ; we use a weighted combination of five kernels, with $\sigma \in \{0.5, 1, 5, 15, 25\}$. We also train the generator multiple times per every training iteration of the discriminator. Zhang et. al. uses 5 G iterations per D iteration; we use 25, as our generator is not pretrained as well as the discriminator. In our experience, using less than 25 leads to the discriminator "running away" with the training, forcing the generator's loss to rise endlessly.

When watching the training, we observed a number of interesting patterns and phenomena. Most notably was how the competitive nature of TextGAN is often reflected in the loss functions. Figure 8 shows discriminator (left) and generator (right) loss at one instant of time—we can see that a lowering of the generator loss due to the generator's ability to match features leads to a jump in discriminator loss.

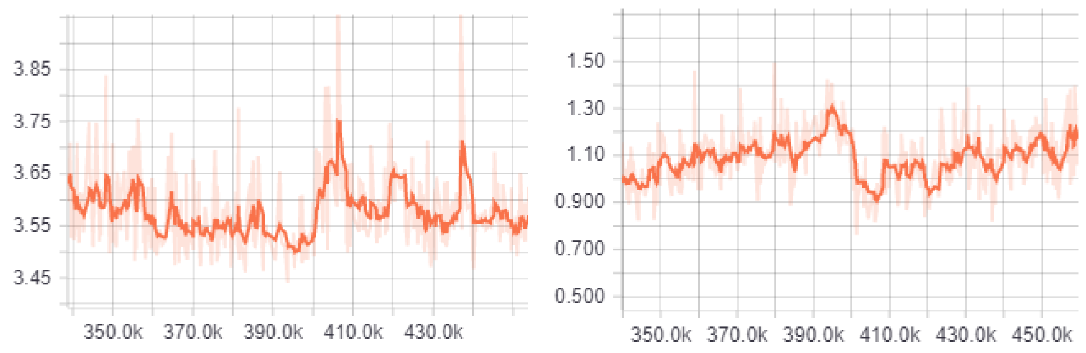

Figure 8: Discriminator loss (left) reacts to a decrease in generator loss (right).

In the original paper, Zhang et. al. train TextGAN for a total of three days, in addition to the pretraining time. This, by far, was the biggest shortcoming of my reimplementation. Having never built a large network before, I did not realize how much time training, bugfixing, and retraining would take. As a result, my version of TextGAN was pretrained for 8 (discriminator) and 4 (generator) hours each, and the core network was trained for 20 hours. Though I used a much smaller subset of the arXiv dataset, this still was not enough time to get adequate results.

In addition, the hyperparameters needed more tuning. With the wrong weighting of the three terms in $\mathcal{L}_D$, or too few G iterations, the training was often heavily biased towards the discriminator, and $\mathcal{L}_{MMD}$ would rise unflaggingly.

4.6 Technical Details

This project is built entirely with TensorFlow. I chose TensorFlow as it seems to be a popular choice for GAN implementations I had seen. In addition, TensorFlow's documentation is spectacular.

The networks are built from scratch. The only external scripts I rely on is the previously mentioned code for computing MMD, and code for computing the BLEU metric, described in the next section.

I train the networks on my home machine using my Nvidia GTX 970.

5 Quantitative Evaluation

I quantitatively evaluate my results primarily using the BLEU method, originally presented in Papineni et al. [11]. I use an open-source implementation of BLEU from the TensorFlow team [1].

BLEU was originally intended for judging machine translation, but nowadays is also often used for judging machine-generated text. The algorithm behind BLEU, though not described in full detail here, is relatively simple: given a piece of generated (or translated) text, count the number of n-grams (usually bigrams, trigrams, and 4-grams) in the text that are also found in the corpus; the BLEU score of the text is based on the proportion of n-grams that were also found in the corpus.

	BLEU-4	BLEU-3	BLEU-2
AE	0.01	0.11	0.39
VAE	0.02	0.16	0.54
seqGAN	0.04	0.30	0.67
TextGAN(MM)	0.09	0.42	0.77
TextGAN(CM)	0.12	0.49	0.84
TextGAN(MMD)	0.13	0.49	0.83
TextGAN(MMD-L)	0.11	0.52	0.85
ours, 300000	0.01	0.14	0.51
ours, 450000	0.00	0.09	0.46

Table 1: Comparison of our implementation with the implementations presented in Zhang et al. [17]. First seven rows reproduced from [17]. The rest of the rows show the results of our implementation after different numbers of total training batches.

To test my implementation of TextGAN, I sample 10 latent codes and generate sentences from them. I then run BLEU for $n \in \{2, 3, 4\}$, using a sampling of the dataset as the reference corpus. Zhang et. al. generate 320 sentences and test them against their corpus, but we cut down the numbers due to time constraints. For reference: generating a BLEU scores on my machine can take up to 30 minutes per sentence, even when using the sampled corpus.

Table 1 shows the results of our reimplementation. The first rows are reproduced from [17], while the final rows are the scores of our implementation at different stages of training. Clearly, while our reimplementation is competitive with other generative techniques proposed in the past (namely the autoencoder of [3] which we discussed earlier, and the Variational Autoencoder implemented by Bowman et al. [2]) it is not competitive with Zhang et. al.'s multiple TextGAN implementations.

It is also interesting to see how the performance changes with time. Seemingly, with more training, the model produced worse results. In my experience, from watching the results coming out of the model, I can say that the results varied significantly with time, and that more iterations did not always mean instantly better results.

Tables 2 and 3 present some sample sentences from our TextGAN implementation. A number of things can be inferred by looking at these sentences.

First, we see a lot of repetition of words, especially at the end of sentences. I noticed that, while training, sentences generally learned to be more varied and interesting at their beginnings, while the ends of sentences were generally filled with repetitions of one or two words. At later stages in the training, we can see a lot of variety in the first five to ten words of the sentence, after which point the sentence degrades.

Second, it seems the generator's vocabulary is rather limited; we see the same words (or symbols) popping up over and over. This was another thing I noticed during training: the sentences produced within a batch would tend to re-use the same words. However, over the course of a training epoch, the words used would change. From this, we can hypothesize that the most recently trained sentences can have an outsized effect on the generator.

Lastly, and to the credit of our implementation, we do see some sensible phrases

1	resulting introduced will make that the topological algebra of introduced of fermat five r r r introduced at introduced at
2	oppositely introduced at successfully make that fermat algebra of the topological fermat END
3	resulting introduced will make that the topological algebra of introduced at r r r r r r at r r
4	which make that the fermat algebra of fermat term END
5	oppositely u u fermat fermat fermat algebra of introduced introduced with the five r r algebra at introduced at r
6	oppositely u u fermat fermat fermat algebra of the introduced with the five r r algebra at introduced at r
7	oppositely u u fermat fermat fermat algebra of introduced introduced at the five r r algebra at introduced at r
8	resulting introduced will make that the topological algebra of introduced at r five r r r introduced at r r
9	which make the fermat of the topological term END
10	oppositely u u fermat fermat fermat algebra of the five of r r r r r r r r

Table 2: Sample sentences generated by our implementation after 300,000 training steps.

1	by graded END
2	and END
3	string features of the and string string END
4	including an discreteness sum further rightarrow the green and the string string introduced to the introduced to functional functional string
5	including lefschetz lefschetz lefschetz lefschetz lefschetz discreteness sum further rightarrow the green green the and string string and the functional
6	we coupled green green graded END
7	by an discreteness discreteness sum further rightarrow the green of the and string introduced and the introduced to functional functional
8	the string features to the and string string END
9	including coupled green green graded END
10	lefschetz coupled green green graded END

Table 3: Sample sentences generated by our implementation after 450,000 training steps.

within otherwise nonsensical sentences—for example, sentence 9 in table 2. Much like the limited vocabulary, though, the "phrase vocabulary" is also limited; we see the same or similar short phrases cropping up again and again.

6 Conclusion

This term project originally had a wider scope: we were going to implement TextGAN first, and then make a modification on top of that. Instead, as the task of reimplementing TextGAN proved to be much more time-intensive than originally expected, we narrowed the scope of the project to just reimplementing TextGAN. We gained thorough enough comprehension of TextGAN to successfully build the model by hand in TensorFlow, but did not allocate enough training time to get high-quality results.

Though we failed to train our implementation of TextGAN to the level of performance of Zhang et. al.'s original implementation, I personally feel very satisfied with this project. Before starting this project, I knew nothing about TensorFlow; moreover, I knew nothing about the practicalities of constructing neural networks. We have discussed neural networks in class, and even trained them in projects—yet this was the first time I'd ever built a more complex network using a powerful framework like TensorFlow. Not only am I confident that I would be able to field questions about TextGAN—I also feel far more confident about neural networks in general.

References

[1] bleu.py. `https://github.com/tensorflow/nmt/blob/master/nmt/scripts/bleu.py`. Accessed: 2018-04-29. 12

[2] Samuel R. Bowman, Luke Vilnis, Oriol Vinyals, Andrew M. Dai, Rafal Józefowicz, and Samy Bengio. Generating sentences from a continuous space. *CoRR*, abs/1511.06349, 2015. URL `http://arxiv.org/abs/1511.06349`. 13

[3] Zhe Gan, Yunchen Pu, Ricardo Henao, Chunyuan Li, Xiaodong He, and Lawrence Carin. Unsupervised learning of sentence representations using convolutional neural networks. *arXiv preprint arXiv:1611.07897*, 2016. 3, 11, 13

[4] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2672–2680. Curran Associates, Inc., 2014. URL `http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf`. 2, 6

[5] Arthur Gretton, Karsten M Borgwardt, Malte J Rasch, Bernhard Schölkopf, and Alexander Smola. A kernel two-sample test. *Journal of Machine Learning Research*, 13(Mar):723–773, 2012. 11

[6] Michael U Gutmann and Aapo Hyvärinen. Noise-contrastive estimation of unnormalized statistical models, with applications to natural image statistics. *Journal of Machine Learning Research*, 13(Feb):307–361, 2012. 5

[7] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997. 3

[8] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. A convolutional neural network for modelling sentences. *CoRR*, abs/1404.2188, 2014. URL `http://arxiv.org/abs/1404.2188`. 3

[9] Yoon Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014. 9

[10] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013. 5, 6

[11] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics, 2002. 12

[12] sigkdd. Kdd cup 2003 dataset. `http://www.cs.cornell.edu/projects/kddcup/datasets.html`. 3

[13] Dougal J Sutherland. opt-mmd. `https://github.com/dougalsutherland/opt-mmd`. 11

[14] Dougal J Sutherland, Hsiao-Yu Tung, Heiko Strathmann, Soumyajit De, Aaditya Ramdas, Alex Smola, and Arthur Gretton. Generative models and model criticism via optimized maximum mean discrepancy. *arXiv preprint arXiv:1611.04488*, 2016. 11

[15] TensorFlow. `https://www.tensorflow.org/tutorials/word2vec`. 4, 5, 6

[16] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Reinforcement Learning*, pages 5–32. Springer, 1992. 10

[17] Yizhe Zhang, Zhe Gan, Kai Fan, Zhi Chen, Ricardo Henao, Dinghan Shen, and Lawrence Carin. Adversarial feature matching for text generation. *arXiv preprint arXiv:1706.03850*, 2017. 2, 3, 8, 10, 11, 13