Comparison of Deep Learning Text Generation Models Trained with Song Lyrics

Nathan Stone, Zack Strathe
Kansas State University
CIS 732

Introduction

For intelligent-systems to interact with humans, natural language is the method accessible to most of the world's population. For that reason, text generation is a very important, though also very difficult problem in machine learning. Several different methods of text generation exist, but finding nonsensical results can be commonplace. Currently, state-of-the-field in natural language processing (NLP) is the use of large language models (LLMs) such as OpenAI's GPT-3 or Google's BERT, which do perform well at most natural language processing (NLP) tasks; however, these LLMs do pose some concerning downsides with their use. The largest concern for LLMs is the introduction of bias into the model due to the extremely large corpus used for training. For instance, GPT-3 was trained on the Common Crawl Dataset, which includes documents linked to from Reddit, Wikipedia articles, and a collection of books. With such a large corpus, there may inevitably be some level of biased viewpoints being expressed which could show up in the outputs from the trained model (Bender, et al. 2021). While unintentional bias isn't a major concern for a small-scale student project, it is worth some consideration, and we believe that this ethical issue of bias in LLMs presents an opportunity to evaluate the performance of small language models that have been trained on a custom corpus. In addition, utilizing small language models allows us to explore some novel methods of developing text generation models that have been the focus of recent research in NLP.

For this project, we intend to experiment with training language models with a dataset of song lyrics from different artists. As a baseline, we will implement and train a recurrent neural network (RNN) with long short-term memory (LSTM) and maximum likelihood estimation (MLE). In comparison to this baseline model, we will develop a similar RNN with LSTM, but with hyperparameter tuning in an attempt to improve evaluation results. Further, we will utilize the TextBox (Li, et al. 2021) module– which has implemented a framework for training and evaluating a large number of different language models– to train and evaluate a few different generative adversarial network (GAN) text generation models. Our analysis will include evaluation of the models using both a human-scoring method and a computed score. With the selected set of song lyrics that we will use for training and evaluation, we hope to see that the models incorporate different themes in language usage (such as word choice pertaining to certain topics) that are found in the corpus. For instance, our corpus includes lyrics from Stevie Wonder, which generally contains upbeat themes (example lyric: "Love's in need of love today"), but also includes lyrics from Metallica, which tend to have a much darker tone in general word usage (example lyric: "Rule the midnight air, the destroyer"). While previous research into model-based generation of song lyrics (Potash, Romanov and Rumshisky 2015) has explored developing a metric utilizing inverse document frequency to compare a generated lyric's similarity to a specific artist, our evaluation will instead utilize a much simpler metric. For human-evaluation, we will look at the semantic meaning of the generated text (answering "does it make sense?"), as well as evaluating the grammatical correctness of the generated text. For computer calculated scores, we will utilize the BLEU evaluation, which compares the n-gram similarity between the generated text and an evaluation corpus.

Background and Related Work

Previous Work into Artificial Lyric Generation

In their paper "GhostWriter: Using an LSTM for Automatic Rap Lyric Generation" (Potash, Romanov and Rumshisky 2015) a group from the University of Massachusetts Lowell utilized a RNN with LSTM architecture to artificially generate rap lyrics. This research sought to not only generate lyrics, but to generate the structure of lyrics, taking into account rhyme scheme and similarity to a target rap artist. The results of this paper present promising results of utilizing a RNN-LSTM model for the generation of novel lyrics that are similar in style to a reference corpus.

Recurrent Neural Networks

RNNs and LSTMs have seen implementation in text generation tasks previously, which led us to choose LSTMs for

this project. Due to its previous implementations for song lyrics, and being a simpler network structure, we decided to use an LSTM for our baseline model. We also decided to tune the parameter further from the baseline model to see if we get better results. LSTMs also solve the vanishing error problem present in traditional RNNs (Staudemeyer and Morris 2019). LSTMs work by taking in a vector of word encodings from the corpus and using probabilistic prediction to guess what the next character in the sequence will be. We choose LSTMs over other RNN architectures due to the memory cell present in LSTMS with the hopes that it would produce more convincing and meaningful song lyrics as the memory cell can hold past information longer than a traditional RNN. The memory cell has several important features including input gates, output gates, forget gates, and the cell memory (Potash, Romanov and Rumshisky 2015). The input gates use a sigmoid activation function to control the signals that are sent to the cell memory while the output gates learn how to control access to the memory cell contents (Staudemeyer and Morris 2019). The forget gate is attached to the self-connection in the cell to adjust weights when the stored information is no longer needed (Staudemeyer and Morris 2019). For our RNN LSTM implementations we utilized Keras packages with more detailed explanation below in the Methodology section of this report. For both of the RNN implementations, cross-entropy loss was used. The formula for the cross-entropy loss is calculated after the SoftMax layer and is calculated with this formula:

$$CE = -\sum_{i}^{C} t_i log(f(s)_i)$$

Generative Adversarial Models (GANs)

The usage of GANs has emerged as a means to alleviate what is known as "exposure bias" in RNN networks, which is a result of the model being trained to predict a word, given a sequence of ground truth words, but at the inference stage, being instead provided with the previous sequence of words generated by the model. As a result of the exposure bias, errors quickly accumulate during the inference stage, so that a generated sentence may initially seem reasonable, but will quickly start to deteriorate as sentence length increases (Yu, et al. 2017, Zhang, et al. 2017). By utilizing elements from reinforcement learning, GANs can alleviate exposure bias in text generation tasks.

With GANs, two models are utilized that are trained adversarially: a discriminator, which is trained to recognize whether a data example is real or not, and a generator, which is trained to generate synthetic data with the goal to trick the discriminator into believing that it is real (Yu, et al. 2017). The objective of training a GAN is to train a generator that "functionally maps samples from a given (simple) prior distribution, to synthetic data that appear to be realistic" (Zhang, et al. 2017). In computer vision tasks, GANs have been very successful, with state-of-the-field image generation algorithms that are capable of creating artificial images indistinguishable from a photo. However, text generation is different from image generation in several ways, but primarily: text generation deals in sequences of discrete tokens, while image generation deals with real-valued, continuous data (Yu, et al. 2017). In a traditional GAN, the loss can be calculated between the output of the discriminator and the generator output to yield the gradient for updating the parameter weights of the generator. However, with discrete sequences like text, a parameter update with respect to a gradient determined from the discriminator output doesn't make any sense because there is "probably no corresponding token for such slight change in the limited dictionary space" (Yu, et al. 2017).

SeqGAN (Yu, et al. 2017)

With SeqGAN, the authors decided to consider text sequence generation as a sequential decision-making process. In this implementation, the generative model is essentially a policy-based reinforcement learning agent, utilizing a RNN with LSTM cells.

The generator directly trains a stochastic parameterized policy via a policy-gradient method, and utilizes a Monte Carlo search to approximate the state-action values. The environment state at each step is the generated sequence of tokens so far, and the action is the next token in the sequence to be generated. The reward values are estimated as the likelihood that the generator will trick the discriminator with a synthetic sample that the discriminator evaluates to be real. The discriminator is a convolutional neural network (CNN) which evaluates every completed sequence of generated tokens. The last layer of the CNN is fully-connected with sigmoid activation and outputs the probability that the sequence is real. The output of the discriminator is then used to guide training of the generator.

For training of the SeqGAN model, the generator is first pre-trained using MLE estimation on a training data set. Next, the discriminator is also pre-trained, by minimizing cross-entropy with negative samples from the pre-trained generator and positive samples from the training data set. Afterward, the RNN-LSTM generator and CNN discriminator are alternatively trained. The generator is trained for *g-steps*, while the discriminator is periodically trained for *d-steps.* At each iteration of the discriminator training loop (*d-steps*), the discriminator is trained to classify using real-data from the training set versus fake data from the generator, with an equal balance between positive and negative examples. During adversarial training, the generator is updated by minimizing cross-entropy loss, utilizing the function:

$$\mathcal{L}(y, \hat{y}) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$$

where "y is the ground truth label of the input sequence, and y-hat is the predicted probability from the discriminative model" (Yu, et al. 2017).

TextGAN (Zhang, et al. 2017)

In their paper detailing TextGAN, the author's note their motivations for building upon the techniques implemented with SeqGAN. First, they note that "mode collapse", where a generator tends to produce a single output for multiple latent representations (encoded text), is an issue with text generation GANs such as SeqGAN. Second, they note the vanishing gradient problem that exists for a generator when the discriminator is close to its local optimum. To alleviate these problems, the TextGAN authors set out to utilize a feature-matching approach to train the generator, instead of directly optimizing the objective as with a standard GAN approach. Rather than the objective of generator training being increasing the probability of the discriminator being fooled by synthetic data, they instead train the generator to more-closely produce a synthetic sentence that matches an encoding from the discriminator. In this model, the discriminator attempts to "produce sentence features that are most discriminative, representative and challenging" (Zhang, et al. 2017). Essentially, the adversarial loop with training TextGAN is that the discriminator attempts to identify most-informative sentence features, while the generator attempts to match those identified features.

The model architecture of TextGAN is similar to SeqGAN, utilizing a RNN-LSTM generator and a CNN discriminator. The discriminator essentially acts as a feature detector, convolving over the text sequence to produce a latent feature map, which is fed to a max-over-time pooling layer to determine a filter particular to a certain feature. This filter is then convolved over every position of the sentence, allowing features to be extracted regardless of position in the sentence. The model uses multiple filters with varying window sizes for extracting these features. The RNN-LSTM generator of the TextGAN model is trained to convert an encoded feature vector into a synthetic sentence. The first word of the generator sentence is deterministically generated from the encoded feature vector, with the remaining words sequentially generated with the RNN.

For training with TextGAN, the generator is first pre-trained using a CNN-LSTM autoencoder. For discriminator pretraining, the authors utilize a method of "permutation training," where for each sentence in the training corpus, two words are randomly swapped, to create a slightly different example for the discriminator to pre-train to classify real versus synthetic data. The authors claim that the permutation pre-training is necessary because it "requires the discriminator to learn features characteristic of sentences' long dependencies" (Zhang, et al. 2017). For the generator, TextGAN utilizes the loss function:

$$\mathcal{L}_G = \mathcal{L}_{MMD^2}$$

, where MMD is the maximum mean discrepancy "between the empirical distribution of sentence embeddings…for synthetic and real data" (Zhang, et al. 2017).

LeakGAN (Guo, et al. 2017)

The authors of the LeakGAN method note that, while recent GAN methods have been fairly successful in the generative text domain–especially with treating text generation as a sequential decision-making stochastic policy-gradient method– there is a lack of research into text generation of longer form (more than 20 words). They claim that longer form text generation is necessary for "practical tasks such as auto-generation of news articles or product descriptions" (Guo, et al. 2017). They also note that the main drawbacks of other GAN-based text generation methods such as SeqGAN for longer-form text, is that the binary signal from the discriminator is sparse because it is only generated from the full-length sequence, and that the signal is not informative enough for the generator to sufficiently learn for longer-sequence data, as the discriminator signal does not preserve syntactic structure or semantics of the text. As such, the authors propose some novel methods with LeakGAN to attempt to address these problems.

To deal with the problems of sparse and uninformative reward signals, LeakGAN utilizes elements of hierarchical reinforcement learning (RL). In this model, the generator is split into a high-level "manager" module and a low-level "worker" module. The discriminator for this model utilizes a CNN similar to other text-generation GAN models; however, unlike other GANs researched for this report, the CNN discriminator in LeakGAN serves primarily as a feature-extractor, where information from the last layer of the CNN is "leaked" to the generator. The authors claim this to be a much better guiding signal for the generator, "since it tells what the position of currently-generated words is in the extracted feature space" (Guo, et al. 2017). For the generator function, hierarchical RL methods are utilized, where the "manager" is a LSTM network that receives the feature vector input from the discriminator, and outputs a goal-vector to the "worker", another LSTM module, as a guiding signal for training.

For training, LeakGAN utilizes pre-training of the generator: the "manager" LSTM is pre-trained to distinguish the transition of real text samples in the feature space, while the "worker" LSTM is pre-trained with maximum likelihood estimation. Following pre-training, the generator and discriminator are alternatively trained, with the "manager" and "worker" networks also being alternatively trained for every generator training step. The overall goal in training LeakGAN is to generate a gradient from the manager to update the worker model weights. The generator update gradient is defined as:

$$\nabla^{\text{adv}}_{\theta_m} g_t = -Q_{\mathcal{F}}(s_t, g_t)\nabla_{\theta_m} d_{\cos}\Big(f_{t+c} - f_t, g_t(\theta_m)\Big)$$

, where $Q_F(s_t, g_t)$ is the estimated reward of the current policy, which is estimated with Monte Carlo search. d_{cos} "represents the cosine similarity between represents the co-

sine similarity between the change of feature representation, after c-step transitions, and the goal vector $g_t(\theta_m)$" (Guo, et al. 2017).

Methodology

To conduct the experiment of evaluating different models for unconditional text generation, we knew that we would need to use GPU-enabled computing resources, as all of these text generation models utilize deep neural networks, which are significantly faster to train with a GPU since deep learning utilizes parallel matrix multiplication operations which a GPU is capable of handling. Accordingly, we chose to utilize the Google Colab platform for the implementation of this project because it offers free access to a GPU. One downside to the Google Colab platform is that there is a limit to how long it will allow a process to run. A problem that we frequently encountered was a model-training process being terminated before it could complete. As such, to be able to complete these experiments, we had to limit the corpus text size so that model training and evaluation could be completed within the limitations of the Google Colab platform.

Models

Baseline RNN implementation

For the baseline model we used Keras to develop a simple LSTM using the Keras sequential model feature. The model has 2 layers to it, an LSTM layer with 128 units and a dense layer with a SoftMax activation function. The model was then compiled with a categorical cross entropy loss function and Adam optimizer. This model is trained for 40 epochs with a batch size of 128. We used a sequence length of 10 for the baseline with the corpus length, number of characters, and number of patterns being determined by the data we used. The data we used for both of the RNN LSTM models was the training data that was used to train the generator for the GAN to allow for a better comparison of results.

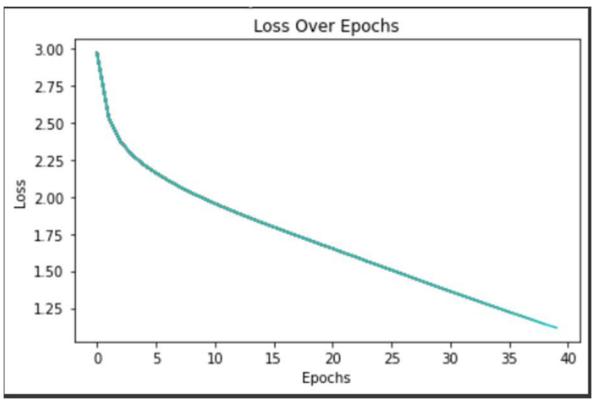

Figure 1: Training loss graph of the baseline RNN with LSTM cells over 40 training epochs

Tuned RNN implementation

For the tuned model, we used Keras sequential model again with 3 layers. We used an LSTM layer with 256 units, a dropout layer with a dropout rate of 0.2, and a dense layer with a SoftMax activation function. This model was also compiled with a categorical cross-entropy loss function and Adam optimizer. We changed the sequence length on this model from 10 on the baseline to 120 so the model would be able to capture much larger sequences of characters. We tried sequence lengths of 40, 80, 100, and 120 and 120 was able to learn more about the structure and patterns from the training data so we continued with that. For this model we also changed the scale factor or temperature from what seems to be the default of 1.0 to 0.2. We tested the temperature at 0.2, 0.35, 0.5, 1.0 and 1.2 and decided to move forward with 0.2 as it produced more convincing song lyrics. With this lowered temperature, the model was less willing to make mistakes with its generation. For this model we trained for 80 epochs with a batch size of 128. We increased the epochs the model was trained on to lower the loss value and for the model to learn to produce better song lyrics by learning more patterns in the training corpus. We did notice that training on too many epochs (about 120) the loss value began to increase significantly.

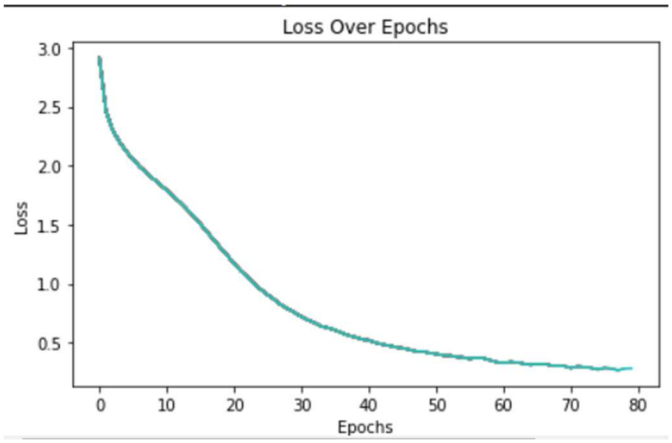

Figure 2: Training loss graph of the RNN with LSTM cells and tuned hyperparameters over 80 training epochs

GAN Implementations

We implemented the GAN models (SeqGAN, TextGAN, and LeakGAN) for this report with the Python module TextBox (Li, et al. 2021) which is a PyTorch based text generation framework developed by a group from Renmin University of China. We chose to use this module because it has implemented many GAN models for text generation uses, and also has implemented multiple evaluation measures. We slightly modified this module for these experiments, to save training loss logs and evaluation scores to .csv files.

To use a custom data set with the TextBox module, it needs to be split into three files: a train file, a test file, and an evaluation file. Because we are utilizing unconditional text generation, we do not need a testing file with labeled data (but it is required for the TextBox library to function), so the test and evaluation files for our song lyrics corpus are identical.

The model training steps for each model utilized from the TextBox module is as follows (Li, et al. 2021):

SeqGAN

In the TextBox module's implementation of SeqGAN, the generator is first pre-trained for 80 epochs, where it utilizes cross-entropy loss between the generated output and the target sequence for updating parameter weights. The discriminator is then pre-trained for 50 epochs, where it updates parameter weights by evaluating both real and fake data, then calculates the average cross-entropy loss of the predictions for both the real and fake data. Last, the model is trained adversarially for 80 epochs, where the generator is updated utilizing the cross-entropy loss between the ground truth label of the input sequence and the predicted probability of the generator being able to trick the discriminator (to classify a generated sample as real). During each epoch of adversarial training, the discriminator is trained for an additional 5 epochs, again utilizing cross-entropy loss with real and fake data. A graph displaying the adversarial training losses for the SeqGAN generator is displayed in Figure 3.

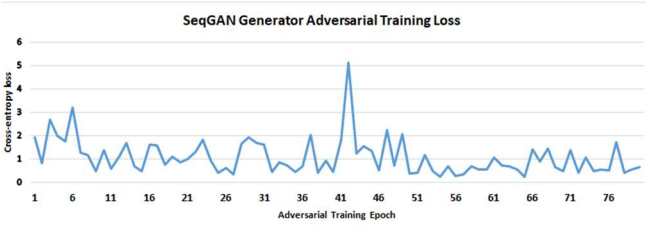

Figure 3: Adversarial training loss graph of the SeqGAN generator, showing cross-entropy loss over 80 adversarial training epochs

TextGAN

In the TextBox module's implementation of TextGAN, the generator is first pre-trained for 80 epochs, where it utilizes cross-entropy loss between the generated output and the target sequence for updating parameter weights. The discriminator is then pre-trained for 50 epochs, where it updates parameter weights by evaluating both real and fake data, then calculates the average cross-entropy loss of the predictions for both the real and fake data, then additionally calculates and adds maximum mean discrepancy and reconstructed loss for the feature encoder. The discriminator loss is also regularized with L2 normalization. Last, the model is trained adversarially for 80 epochs, where the generator is updated utilizing the maximum mean discrepancy (MMD), which "measures the mean squared difference between two sets of samples" (Zhang, et al. 2017), between the ground truth label of the input sequence and the generated sequence. During each epoch of adversarial training, the discriminator is trained for an additional 5 epochs, again utilizing cross-entropy loss with real and fake data, plus maximum mean discrepancy and reconstruction loss. A graph displaying the adversarial training losses for the TextGAN generator is displayed in Figure 4, where the generator attempts to minimize the MMD between generated and real data.

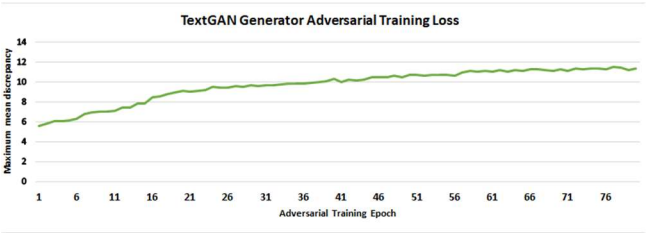

Figure 4: Adversarial training loss graph of the TextGAN generator, showing maximum mean discrepancy over 80 adversarial training epochs

LeakGAN

In the TextBox module's implementation of LeakGAN, the generator is first pre-trained for 80 epochs, where the "manager" and "worker" sub-modules are each updated. The manager sub-module calculates cosine similarity loss between its hidden state output and the real features. The worker sub-module is updated during each pre-training epoch with the cross-entropy loss between training targets and the "leaked" representation from the discriminator model. The discriminator is then pre-trained for 50 epochs, where it updates parameter weights by evaluating both real and fake data, then calculates the average cross-entropy loss of the predictions for both the real and fake data which is regularized with L2 normalization. Last, the model is trained adversarially for 10 epochs of interleaved adversarial training, where the generator and discriminator are trained adversarially for 8 epochs, followed by 5 epochs of generator pre-training steps repeated, then 5 epochs of discriminator pre-training steps. During each adversarial training epoch, the generator is updated with the "worker" loss calculated as the cosine similarity between its output and the training data, which is multiplied by the cross-entropy loss between training targets and the leaked representation of the targets. Following the generator training during each adversarial epoch, the discriminator is additionally trained for 15 epochs, utilizing the same cross-entropy loss function from pre-training steps. Finally, each adversarial training epoch ends with 5 more iterations, utilizing the same loss functions from pre-training, for both the gen-

erator and the discriminator. A graph displaying the adversarial training losses for the LeakGAN generator worker sub-module is displayed in Figure 5.

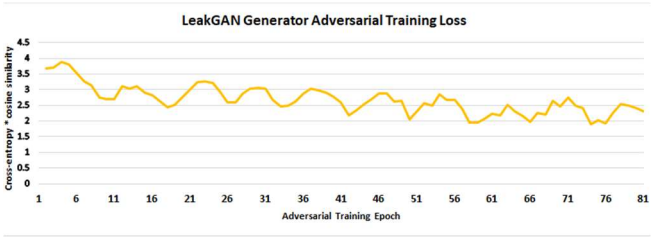

Figure 5: Adversarial training loss graph of the LeakGAN generator, showing the cross-entropy*cosine similarity loss over 80 adversarial training epochs

Experiment Design

Data Set

The data set that we're using is a corpus of song lyrics collected from azlyrics.com, with 31 randomly selected songs from each of the musical artists: Stevie Wonder, David Bowie, Tool, Nine Inch Nails, Metallica, Black Sabbath, Jay-Z, and Frank Zappa. In total, the corpus includes lyrics for 248 total songs. The corpus was further processed to split the data into a train and a test set. To ensure an even distribution of both musical artists and songs in both sets, the data was sequentially split, after first removing all duplicate entries to ensure that the same sentence was not inadvertently included in both sets. In total, the data set consists of 7,842 unique sentences, with an even split between training and testing sets, with 3,921 sentences each. Each sentence in this set contains 7.42 words on average, and to approximately match this we set parameters in the models to generate a maximum length of 10 words per sentence.

In regard to word makeup of the data corpus, the set consists of 6,411 unique tokens in total, with the training set containing 4,317 unique tokens and the testing set containing 4,294 unique tokens. Based on this makeup of unique tokens counts, it can be estimated that approximately 1,050 unique tokens included in the training set are not seen in the testing set, and vice versa for 1,050 unique tokens in the testing set that are not seen in the training set. Since this difference in unique tokens accounts for approximately 24% of each training and testing set, there may be poor evaluation performance relative to evaluation results from literature that utilize data sets approximately 50-10 times larger (because we assume that a larger data set overall would result in a smaller percentage of tokens unique to each of the training and testing sets). This would be because evaluation of a model would be based partly on tokens that the language model has never encountered. Though we expect that the actual quality of generated text won't be significantly impacted by this issue with the corpus.

Evaluation Methods

For evaluation of the models, we will primarily utilize human evaluation of individual sentences. In similar experiments (Yu, et al. 2017, Guo, et al. 2017) with evaluation of text generation models, human evaluation was conducted with Turing tests, utilizing human subjects to score a 1 for samples that they believe to be real, and 0 for samples that they believe to be machine generated. For this experiment, two human evaluators will evaluate each sample with a score of between 1 and 5 for coherence and grammar. To ensure there is no bias in the evaluation, each evaluator is shown each example in a randomized order, with no information identifying the source model of the text. The evaluation scores are defined as:

1: example contains no real words
2: example contains some real words but lacks structure or meaning
3: example contains some real words and has some meaning (such as, contains a subject and a verb)
4: example contains all real words, but lacks some coherence and meaning
5: example contains all real words, and seems like it could be a real song lyric

Because the corpus is made of song lyrics, each sample doesn't need to be a full sentence, so this evaluation will largely consider whether the generated sample would make sense as a song lyric (even one- or two-word statements are acceptable, such as examples from the testing corpus which would receive a score of 5: "Right", "Goblin girl", "Attack"). And finally, the mean of both human annotated scores will be calculated for the final combined score. The best-performing model will be identified as the one with the highest mean score.

In the case of a tiebreaker being necessary with the human evaluation scores, we also computed bilingual evaluation understudy (BLEU) for the generated text from each model. BLEU calculates the n-gram similarity between a sentence and a set of reference texts, where a score of 1 is a perfect match in n-gram similarity. While, as (Tatman 2019) notes, BLEU doesn't account for meaning nor sentence structure, we believe that using these scores in addition to human evaluation (and only as a tiebreaker between models), is a reasonable use of BLEU. Similar to the evaluation method utilized in (Yu, et al. 2017, Zhang, et al. 2017) we will use the entire test set as a reference corpus. For each sentence in the text output of each model, the individual sentence BLEU score is calculated, and the sum of the calculated scores are then averaged to yield the score for each model. This was done for BLEU-1, BLEU-2, BLEU-3, and BLEU-4 scores to yield the 1-gram, 2-gram, 3-gram, and 4-gram similarity between the generated text and reference corpus. While we are reporting all BLEU n-gram scores of 1 through 4, we will utilize BLEU-3 as the

score to use in case of a tiebreaker needed from the human evaluation results. We selected BLEU-3 because we believe this n-gram balance is fitting for a sequence length of 10 (where 2-grams may not be able to capture semantic relationships well enough, and 4-grams may be too sparse within the training and generated texts to achieve a meaningful comparison between models).

Results and Conclusions

Generated Examples

Figure 6 shows examples of unconditional generated text output from the trained models. Each of the seven samples for each model was randomly selected from the model outputs. It appears that the baseline RNN does not tend to generate real English words, while the RNN with tuned hyperparameters does much better (but still generates words that are not real). The GAN models all appear to perform fairly well with unconditional text generation of real words. In regard to generation of meaningful text, the output of both RNN models seem to lack coherence, while the GAN models do much better in generating text that could be interpreted as a genuine song lyric (though these models do still generate some lines that are incomprehensible).

Model	Generated Text
RNN (baseline)	folling world,
	as a 1selly murise freez labming ah please halls skirsendled
	a foor the marst
	your mocawint on bling the 'reer
	ift cruplifes lough
	oh by he vererestiy
	which see beending west is...., in't cauneed undi and yseeay
RNN (tuned hyperparameters)	don't let me to live to like i take you
	please the ruck if the stare
	the chused and shear
	here a must bord i'm on a pellong
	and i was the way your bay you've got
	ther that they cruice from me
	what you get me to be
SeqGAN	and we draw fire
	to feel it like and turn out
	hate
	and kill his joy
	chase back
	listen of love
	put me when i know the family
TextGAN	it 's just a fella ?
	the show the potential
	well it belongs my money ?
	y'all worries them burden
	refugees of crushin , lovers
	hands bags through me now
	wrap the damage when and listen to fight away up
LeakGAN	i want to date i got my back
	you ca n't got ta claim if you
	well , she hid down clouds midnight hot
	i 'm in cuba , love , still get , surprise
	i 'm not such a stranglehold , i 'm down
	i 'm eating
	i 'm not mind girl

Figure 6: Randomly selected samples of generated text sequences from each trained model

Model Evaluation Results

In our primary evaluation of the models, utilizing a human evaluation score, each of the alternative models performed significantly better than the baseline RNN. These evaluation scores are displayed in Figure 7. The SeqGAN model was evaluated to be the best, and no tiebreaker was necessary for utilizing the BLEU scores. In general, the human evaluation scores for the GAN models were higher than the RNN models, so we may conclude that these models do generate more meaningful text with unconditional generation. The SeqGAN performed the best, though not by a large margin compared to the other GAN models trained and evaluated.

While we aren't utilizing the BLEU scores for evaluation, since no tiebreaker was needed, they are also displayed in Figure 7, for each of the n-gram similarity calculations of BLEU-1 through BLEU-4. One observation from these BLEU scores is that the higher-scoring model changes depending on the n-gram metric utilized, and that there aren't large differences in scores between the models, especially for the3-gram and 4-gram evaluations. Therefore, it seems inconclusive whether the BLEU metric is useful for a meaningful evaluation of unconditionally generated text.

Model	Human Eval	BLEU-1	BLEU-2	BLEU-3	BLEU-4
RNN (baseline)	1.55	0.415	0.074	0.038	0.046
RNN (tuned)	2.80	0.744	**0.266**	**0.058**	0.045
SeqGAN	**4.55**	0.786	0.176	0.056	**0.063**
TextGAN	3.90	**0.811**	0.187	0.046	0.049
LeakGAN	3.85	0.802	0.227	0.045	0.034

Figure 7: Display of evaluation results for baseline and alternative models. The primary evaluation method (human evaluated with a metric) is shaded. The best score for each metric is in bold.

Future Work

While the alternative models that we trained and evaluated did outperform the baseline model, the generated text from these models still seemed to lack in quality and coherence. Therefore, we have identified a couple of priorities for future work that could improve the quality of these models. First, we would like to utilize a larger corpus of data, because we were limited in the corpus size by the computing resources available and were only able to utilize a corpus containing 7,842 sequences of text. In comparison, many examples in literature, such as (Zhang, et al. 2017), utilize a corpus of 50,000+ sentences. So, we would expect that the quality of the output from our models would significantly improve with a much larger corpus. Second, we were limited in how much time was available for hyperparameter fine-tuning. Therefore, we believe that further fine-tuning of hyperparameters could boost the output quality of these models. For future fine tuning we would use exhaustive grid search which would allow us to use every combination of parameters. With the exhaustive grid search

method, we could use a metric like cross-validation to determine which combination of parameters would produce the best model. This would be an improvement on the technique of manual parameter tuning that we performed for this project.

A further consideration for future work would be to further evaluate these models with consideration for how much computing resources they take to train. We informally observed that most of the GAN models took significantly longer to train than the RNN models, and that there was some variation in how long each particular model took. And while the output of the GAN models does seem to be higher in quality, it would be interesting to further analyze the quality of the models relative to the resources required to train them.

References

Bender, M Emily, Timnit Gebru, Angelina McMillan-Major, and Schmargaret Schmitchell. 2021. "On the Dangers of Stochastic Parrots: Can Language Models Be Too Big? 🦜." *2021 ACM Conference on Fairness, Accountability, and Transparency (FAccT '21).* New York, NY: Association for Computing Machinery. 610-623.

Guo, Jiaxian, Sidi Lu, Han Cai, Weinan Zhang, Yong Yu, and Jun Wang. 2017. "Long Text Generation via Adversarial Training with Leaked Information." December 8. Accessed March 15, 2022. https://arxiv.org/abs/1709.08624.

Li, Junyi, Tianyi Tang, Gaole He, Jinhao Jiang, Xiaoxuan Hu, Puzhao Xie, Zhipeng Chen, Zhuohao Yu, Wayne Xin Zhao, and Ji-Rong Wen. 2021. "TextBox: A Unified, Modularized, and Extensible Framework for Text Generation." April 19. Accessed February 22, 2022. https://arxiv.org/abs/2101.02046.

Potash, Peter, Alexey Romanov, and Anna Rumshisky. 2015. "GhostWriter: Using an LSTM for Automatic Rap Lyric Generation." *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, September: 1919-1924.

Staudemeyer, Ralf C, and Eric Rothstein Morris. 2019. "Understanding LSTM -- a tutorial into Long Short-Term Memory Recurrent Neural Networks." September 12. Accessed March 6, 2022. https://arxiv.org/abs/1909.09586.

Tatman, Rachael. 2019. *Evaluating Text Output in NLP: BLEU at your own risk.* Towards Data Science. January 15. Accessed April 23, 2022. https://towardsdatascience.com/evaluating-text-output-in-nlp-bleu-at-your-own-risk-e8609665a213.

Yu, Lantao, Weinan Zhang, Jun Wang, and Yong Yu. 2017. "SeqGAN: Sequence Generative Adversarial Nets with Policy Gradient." August 25. Accessed March 15, 2022. https://arxiv.org/abs/1609.05473.

Zhang, Yizhe, Zhe Gan, Kai Fan, Zhi Chen, Ricardo Henao, Dinghan Shen, and Lawrence Carin. 2017. "Adversarial Feature Matching for Text Generation." November 18. Accessed March 15, 2022. https://arxiv.org/abs/1706.03850.