Edgar Allan Poe-try Generator - Written Evaluation

Matthew Sutton (msutton4), Liliana Mack (mmack1), David Heffren (dheffren)

Github Link: https://github.com/Captain525/EdgarAllanPoe-tryGenerator

Introduction:

Poetry is creative, thoughtful, and emotional. It is a hard task for a machine, especially one that can only add zeros and ones, to produce a coherent and creative piece such as a poem. Yet, we've seen massive improvements in generative modeling, to the point that computers can now output creative pieces such as poetry, songs, and pictures/paintings. Thus, in this project, we experimented with generative models and language representations in order to generate poems. More specifically, we built an Edgar Allan Poe-try Generator with the goal of generating poems in the theme/style of the great Edgar Allan Poe. This is a structured prediction learning task where our model produced poems after it was trained on a corpus of Edgar Allan Poe Poetry. We chose to reimplement and expand upon the paper "GPoeT-2: A GPT-2 Based Poem Generator" by Kai-Ling Lo, Rami Ariss, and Phillip Kurz (https://arxiv.org/pdf/2205.08847.pdf). This paper's objectives were to implement a state of the art natural language model capable of generating five line poems with an AABBA rhyming scheme from scratch without an inputted seed, as well as explore different metrics to quantify the model's output as good or bad poetry. We ended up choosing this paper because it piqued our interest, and it also offered a deeper exploration of the language modeling concepts that we have covered in lecture this semester. Furthermore, we wanted to see if we could adapt the paper's underlying model infrastructure by training it only on a single poet's work which consists of many different rhyming schemes (as opposed to only training on a AABBA rhyme scheme).

Methodology:

Data, Preprocessing, and Encoding

For our implementation, the data we used was very different from the original papers. Our implementation of the "GPoeT-2: A GPT-2 Based Poem Generator" did not rely on "Raw Limericks." Instead, we used a repository of written poetic works from Edgar Allan Poe. A summarization of the data and the differences compared to the original papers can be found in the following table:

	Our Implementation	**Original Paper**
One Author:	Yes	No
Number of Poems:	49	0 - Used OEDILF Limericks
Number of Words in Dataset:	14,803	113,722 Approved Limericks
Model Implementation:	Used Pre-Trained GPT-2 Model as an Intermediate Layer in the Full Model	Fine-Tuned the Pre-Trained GPT-2 Model

As far as text formatting goes, GPT-2 is flexible as it allows you to use almost any text file as the corpus. However, formatted text that introduces regularization of inputs works best for generating creative works like poetry. Thus, we implemented the following preprocessing steps to optimize the regularization in both syntax and semantics:

- Introduced consistent capitalization throughout each poem, and removed punctuation.
- Insert special tokens at the beginning/end of each poem ("<BOS>" / "<EOS>") and between lines ("<LINE>").
- Truncated eac poem to a max size of 1024
- We also created a reverse preprocess step in our code like the original paper did, but we did not end up using it in our actual model as it was outside of the scope of what we were trying to achieve

Model Architecture:

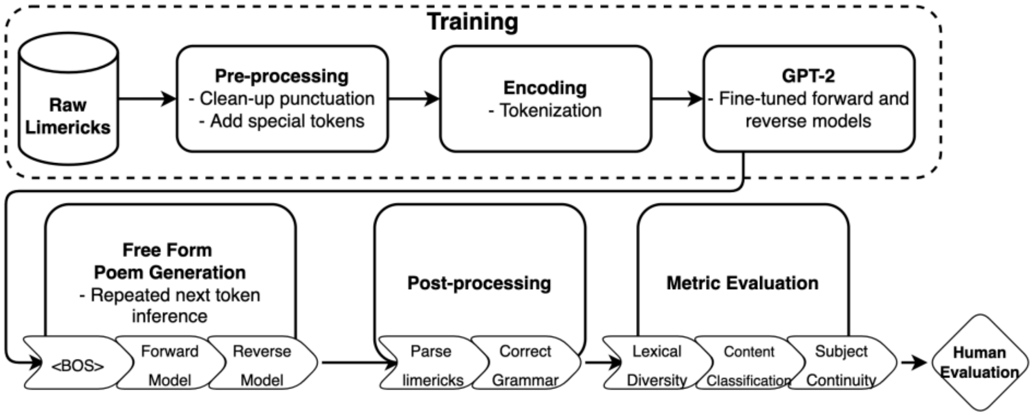


Figure 1: Model training and automatic poem generation process for GPoeT-2.

Our architecture uses the GPT-2 generative language model, which is a generator based transformer. It can be used for both conditional and unconditional poetry generation. For our specific architecture, we used a pre-trained GPT2 model, with our own custom layers before and after. Before the pretrained GPT2 layer, we put a custom trainable embedding layer, which is supposed to fine tune to the specifics of Edgar Allan Poe's poetry. Then, this embedded text is passed into the GPT2 Transformer, using the pretrained weights. Then, we put that output into a linear softmax layer to get the probabilities of each vocab word. The above figure represents the paper's implementation of the generation process. For us, the difference between our model and the one described in the figure above is the "GPT-2" block where we had a different underlying model architecture from the fine-tuned one (as described before).

Results:

Random Model Implementation:

Our random model averages a lexical diversity of 1.0, which makes intuitive sense because there are 14,803 words in our vocabulary. Thus, if every word has equal probability of being selected, it is unlikely that we will get repeats. Since this is a random model, there is nothing to train on, so we do not have perplexity training statistics for this model, but for validation accuracy, the perplexity is going to be about 14,803. This is because perplexity = 14,803 means that there is approximately 1 in 14,803 chance of correctly predicting the next word in the sequence, and since this is uniformly random, there is a chance equal to 1 over the vocabulary size.

Example of Random Model Output:

welltuned crept soaring brood side of triumphantly

didst coming flower standing endymion weariness withal resplendency partner cities

mendez below impotent heard wind sir wake searing define sob

kind lovers environs twelve desire unburthend eyelids shone ramparts

commanding abysses lazily child in bent giant bud frown

Example of the Pre-Trained GPT-2 Output (no added layers):

Lexical Diversity: 0.811

Once upon a midnight dreary

Zippy on the page era, legend to last movie ever to blip

I know industries do well, and sucks! My parents drive only and vote for corporate cronies

and they've never even paid anything for

Our Model's Implementation:

For our specific architecture, we used a pre-trained GPT2 model, with our own custom layers before and after. Before the pretrained GPT2 layer, we put a custom trainable embedding layer, which is supposed to fine tune to the specifics of Edgar Allan Poe's poetry. Then, this embedded text is passed into the GPT2 Transformer, using the pretrained weights. Then, we put that output into a linear softmax layer to get the probabilities of each vocab word.

To generate text, we call the model with the last word of our sequence. Next, after getting the model's output, we then have a probability distribution according to the last softmax layer. From here, we have one of two options - we can either pick the argmax of the output probabilities, or sample from the vocab based on a probability distribution of the model's output probabilities. Lastly, we can specify how deep the model searches/try to filter out repeated words by removing the previous word in the sequence's probability, and then scale the rest of the model's distributions so that they add up to 1.

Example of Our Model's Output:

Lexical Diversity = 0.83

With a prompt (first line):

Once upon a midnight dreary

oth len from and to me to the danger down *but the nebabel my angels lot of lake did at fair and a heaven at kind*

a amid ch he my throne and with you well

a autumn no bes stars, light, half cried the night

Without a prompt:

stir, the child of in and ago of gentlyesar

in'd that (that the when which the lies so, strange the!ils came angel

upon in height late choir heaven butued upon,

a ly in heed when! all r and in fire but upon restier mround thrive only thee

With special tokens and prompt:

Once upon a midnight drearyracted anding eyes thee her heart

i stillupon the se the kn to!

to wild my redill minuteorings with such lowly'

With special tokens without prompt:

orn not hour that boythe at thei been hath he,IORthe a and define thatalone for but to of thethough

the hung heartfor is that when, innocenceof we soul

Embedding + GPT with LM Head:

Lexical Diversity: 0.7266

Once upon a midnight dreary <LINE> the (

Dul (

(- (

a

in the path

the genitals the Mutant the a bashing the an the binge eulchre shrive

a m, a ball the in an eight a andlicense theCatholic a motiv the

GPT + Dense

The heart beat beneath the floorboards

as my heart was rill, with the stilly, all the heavens that herself,

her soul the heart a sad soul my heart a sad heart for

'mid the night i my heart a more than its a more than its a more

Comparison of Model Implementations:

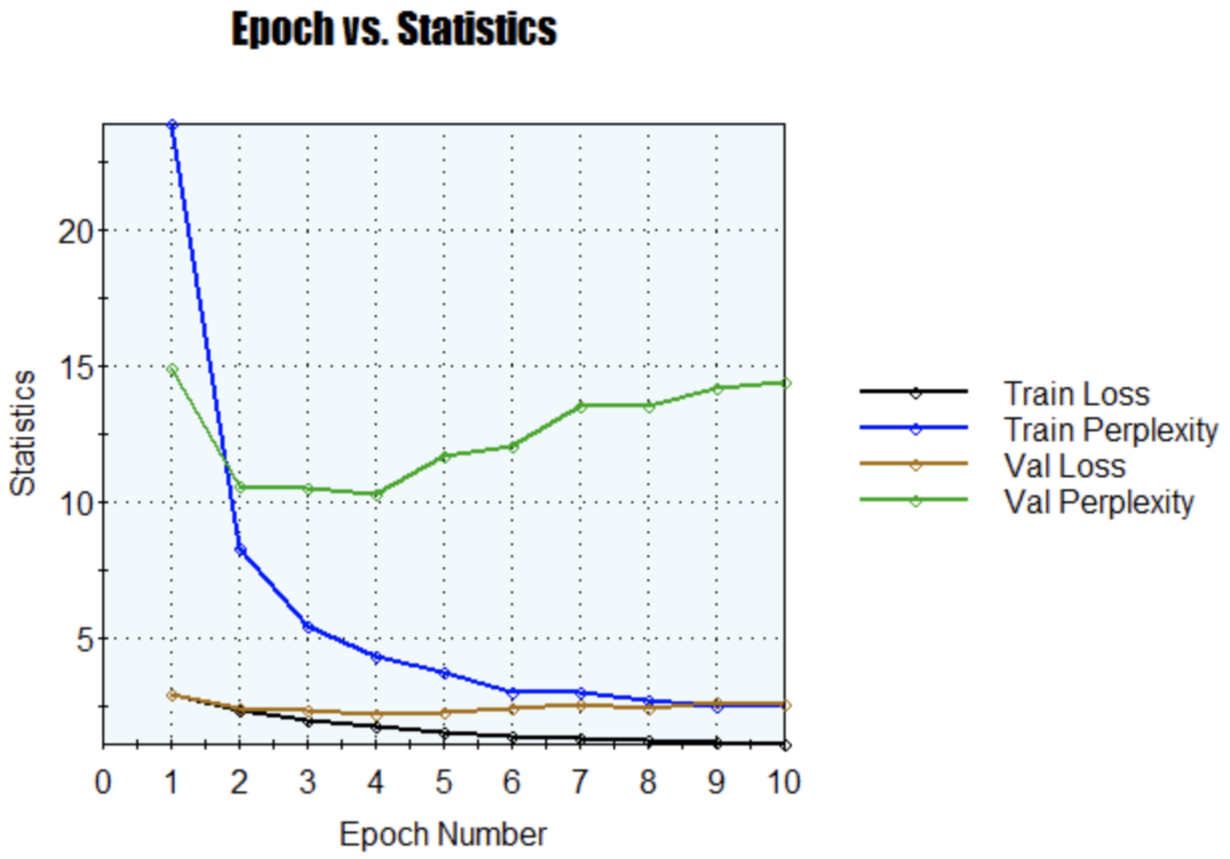
We used a batch size of 2 (90 steps) with 10 epochs, learning rate of .005, the Adam optimizer for all of these.

	Final epoch train loss	Final epoch train perplexity	Final epoch validation loss	Final epoch validation perplexity	Lexical Diversity
Embedding + GPT w/ LM head	1.2137	2.6174	3.6095	5.7474	0.7266
Embedding + GPT + Dense	**1.1324**	**2.5890**	**2.5872**	**14.4569**	**0.7786**
Embedding + GPT + Dense w/ special tokens	0.8824	1.9462	1.9339	8.5046	0.6984
GPT + Dense	2.5048	9.246	7.6918	32.2737	0.607

The Embedding + GPT with LM head is horrible with its generation, as it seems to get random outputs, not even learning to sound like Poe. This is most likely because the embedding layer is trying to learn to be more like Poe, but the outputs are generated as from the training set, which is much less like poe. Earlier layers in the model tend to focus on more general language understanding, which should be the same for the original GPT and the new one, whereas the later ones should focus on the specific task at hand, which this doesn't.

GPT + Dense gets pretty good results, but not quite as good as Embedding + GPT + Dense. I think that this is because the dense layer allows it to more specifically work for our task.

Graph of Our Chosen Model's (Embedding + GPT + Dense) Training Loss/Perplexity:



<u>Challenges</u>:

Our primary challenge in this project was the issues we had with implementing what the paper had done. We found that their method of just fine tuning the pretrained GPT2 model didn't work for us, as it was too computationally expensive. Half of the work they implemented was specific to that exact method of generating poetry, which meant that we were stuck trying to reinterpret and analyze their code to see how that could apply to our scenario. Even though the task was supposed to be "reimplement an existing paper" it ended up being closer to doing a new project, due to how inapplicable their existing implementation was for our task (data and computational resource constraints). We ended up having to redesign our own techniques to do a lot of tasks, which took up a lot of time.

An example of where we had to redesign our own techniques is with generation, where in the paper, since they just fine tuned the existing model, they could use the built in generation function with beam search and all the fancy techniques to prevent word repetition and penalize for length. However, we weren't able to use that, so we had to design our own generation methods, lacking a lot of the complexity of the built in model. This is part of the reason why our model generates so many repetitive, strange phrases.

Another problem we ran into was that our tokenizer wouldn't allow us to add special tokens, as it wouldn't work with the embedding layer of the pretrained model. This took us very long to wrap our heads around, and this also may have negatively affected the results.

It was difficult to tell whether problems were coming from our preprocessing, post processing or our training itself, which made it hard to know where to put our focus/energy. The model was learning something, but our results suggested that it wasn't learning properly/adequately enough.

The paper wasn't clear either, since it lacks explanations and information in the paper body of what they were doing, and the pytorch code of their implementation had almost no comments describing what each method was supposed to do. We wasted a lot of time trying to implement things from their code which weren't even used in the final paper, yet we couldn't tell which was which.

Overall, the most difficult challenge that we may have faced was the constant pivots that we had to make. Two of the three group members reached out to our mentor TA throughout the project, describing the challenges that we were facing and what had to be done because of it. The constant pivoting made the project more stressful and resulted in mediocre results (in our opinion) because we spent too much time trying to get something running as opposed to training and optimizing the model.

Reflection:

How do you feel your project ultimately turned out? How did you do relative to your base/target/stretch goals?

In our opinion, our results are mediocre at best as the model's output isn't very coherent (especially after observing the pre-trained GPT-2 outputs), and we wanted to generate good poetry. Although, when looking at our base/target/stretch goals from our first TA check-in (which are slightly different goals then the ones on the project outline), we actually did okay. More specifically, as discussed in our TA check-in, our base/target/stretch goals were: base - implement a model that is better at generating poetry than a random word picker, target - implement a model that produced coherent outputs, and stretch - implement a model that implemented coherent outputs that contained rhyme schemes. The fact that the paper's implementation didn't work in our case, due to computation deficiencies, meant we had to improvise and try out a different technique, which didn't really work out. Although, with all the pivots we had to make, we still achieved our base goal of implementing a model that produced results that were better than the random generators. As far as our target goal, our model will produce somewhat coherent poems in the tone of Edgar Allan Poe from time to time, but overall, does not do it on a consistent enough basis for us to conclude that we reached that target goal.

Did your model work out the way you expected it to?

No it didn't really work out the way we expected it to, as our model didn't reach our goal of consistently outputting coherent phrases or poetry. We didn't really expect it to make great poems, but we hoped to get some sort of logical coherence or consistency. I think that this has something to do with the fact that we added a linear layer at the bottom of the network, rather than using a pre-trained one, which meant that the inevitable predictions would be much less fine tuned, and have much worse results. The paper didn't have to worry about designing their own layers, or picking the architecture, which took up a lot of time for us.

How did your approach change over time? What kind of pivots did you make, if any? Would you have done differently if you could do your project over again?

Our approach changed over time mostly due to trial and error. First, our base/target/stretch goals were revised as in our first TA check-in as outlined in the first reflection question above. Further, we considered pivoting to a new project altogether because we felt that fine-tuning a pre-trained model like GPT-2 was not enough to submit as a final project. Although, after we started implementing the underlying model to replicate the paper, we realized that we had to pivot because of other issues. More specifically, we found that it was not going to be an option for us to fine-tune the pre-trained GPT-2 model. This is because GPT-2 is MASSIVE with about 1.5 billion trainable parameters. It was trained with a dataset which included about 45 million website links. Fine-tuning this pre-trained model would not only be very long and drawn out, but it also wasn't feasible for us as it crashed multiple group members IDE's (Google Colab, VSCode, and Jupyter Notebook) over the course of the project. Thus, we pivoted to using the pre-trained GPT-2 as an intermediate layer (parameters were frozen so no new training took place) in our overall model, while adding layers on top of it which we trained ourselves on the Edgar Allan Poe dataset. This made it so we were able to train/optimize our own unique model architecture with our own data, while still keeping the performance benefits of the large and well trained GPT-2 model. Furthermore, because we had to implement our own unique model architecture by building off of the GPT-2 base model, we also had to implement our own generate/sample functions.

As far as what we would do differently if we did the project over again, we all agreed that if we started the project from scratch, we would have probably picked a different paper on the same topic. We agreed on this because a lot of time was spent on just trying to understand the GPT-2 / "transformers" library and API. Furthermore, once we figured out the API and library stencil, a lot of it went to waste because of the memory/crashing issue that we had. On the other hand, if we did this project over again with the same paper, we ideally would have focused more time on building our own implementation around the pre-existing model, instead of desperately trying to fine-tune it. Of course, this is only in hindsight as we didn't know we would run into the fine-tuning issues that we experienced.

What do you think you can further improve on if you had more time?

We think that the main thing we could improve on is filtering generated text by grammar to generate semi-reasonable results, as well as training the model for longer. We didn't spend as much time training the model as we would have liked, due to the problems with implementing everything else. We think that if we had more time to experiment with more advanced preprocessing techniques, as well as more time to train/optimize the model, we likely could have gotten a relatively decent poetry generator. Our dataset was much smaller than theirs, and we trained our model for fewer epochs than them, due to our lack of time and compute resources.

What are your biggest takeaways from this project/what did you learn?

First off, one of the biggest takeaways from this project was the architecture of the pre-trained GPT-2 model and how its implementation had fantastic results. Over the course of this semester, all of our group members noticed how the performance of a deep learning model was heavily influenced by the underlying assumptions and the architecture that the engineer(s) chose. We also saw how adapting a model's implementation to a specific task could give better results. Although, during this project we realized through GPT-2 that if you have a model architecture that has been proven to work very well in a specific context, then scaling that model and training it on a megaton of data can wildly increase the performance of the model. The base GPT-2 model was incredibly good at generating coherent and structured text. As mentioned earlier, GPT-2 has 1.5 billion parameters and was trained on a corpus of over 45 million web pages. Thus, to make the performance of the model great, the engineers of this model scaled it to be very large, and we noticed how well this worked. Therefore, in summary, we learned that when an existing architecture is proven to work in a specific context, sometimes the best bet at increasing the performance of the model can simply be done by scaling the model and the underlying computational power that you have.

Another takeaway from this project is about the effective management of time when dealing with big projects like these. When you have to spend a lot of time rewriting the code you're implementing to work for your specific case, it makes work on the meat of the project more difficult. It's also easy to waste a lot of time on failed implementations, on things you shouldn't be doing, when you aren't sure what you should focus your efforts on. That's one of the most valuable skills a deep learning engineer can learn, to effectively prioritize which parts of the model to focus on, and to effectively manage their time. Sometimes, we spent too long waiting for a fundamentally flawed model to train, which could've been spent doing something more productive to the final result. However, knowing what will contribute to the final result ahead of time is quite difficult, yet an essential skill to develop.

One last thing we noticed was that sometimes the loss / perplexity values of the models we tested out were lower than the implementation we ended up using, yet the generated outputs were really bad in comparison. Thus, at least in this project, we saw that there was a susceptibility for the model to drastically overfit the data, which resulted in very funky generation outputs.