Epic

**Interactive Fiction**

by:

The PowerPuff Coders

Jonathon Kastner

Carson Stevens

Nam Do

Scott Oelkers

Skyler McMullen

# Table of Contents

[**Table of Contents**](#_oaplplq2racq) **1**

[**What is Epic?**](#_e0xny9125ch8) **3**

[**Data**](#_324b3xolcg1y) **3**

[**Data Cleaning and Processing**](#_w358uoqxyyjz) **3**

[**Feature Learning**](#_het0fyz5dwek) **3**

[**Special Tokenization**](#_92vdtwq6rilm) **4**

[**Token Selection**](#_qj3c88dj3tda) **4**

[**Simple Cases**](#_mrdye91d5838) **5**

[**BERT Zero-Shot Text Classification**](#_4almwbvgl5up) **5**

[**Blacklist Feature ‘Extraction’**](#_qk8tzz3rw42e) **5**

[**Model & Pipeline**](#_fpdh39w3t4rh) **5**

[**GPT2**](#_ti7ugekeg03r) **5**

[**Overview**](#_wh1jfbcl6xyc) **5**

[**GPT2 Architecture Overview**](#_4vsf9bbvbvh) **7**

[**GPT2’s Main Components**](#_uff407qmqgbs) **8**

[**Encoder**](#_e9i8n2vb3a1n) **8**

[**Decoder**](#_uoou9pe4oisu) **8**

[**Positional Encoding**](#_ljtr01dlv3o0) **8**

[**Optimizer**](#_j3p2i8zi508) **8**

[**Residual Dropout**](#_sc0aq3i23b7t) **8**

[**Label Smoothing**](#_lvqk6gmoajwm) **8**

[**Masking**](#_tm85botzwmpe) **9**

[**Self-Attention Process**](#_qltccrzinlf) **10**

[**Masked Self-Attention Process**](#_70gn3q5zi5fb) **10**

[**Model Output**](#_jf60oko2azlw) **11**

[**Hyperparameter Tuning**](#_3bxmmnkdtwok) **12**

[**Models to Finetune Output**](#_c897iuug7q1h) **13**

[**Roberta-Large-MNLI**](#_4gru4uodb3un) **13**

[**Entailment, Contradiction, Neutral Analysis**](#_gz8fzej2trh8) **13**

[**BERT-Base-Multilingual-Uncased-Sentiment**](#_m8tt9a7bws2t) **13**

[**Sentiment Matching**](#_vz0h9xfyjgic) **13**

[**Word Distance Analysis**](#_tc8uvvgnj669) **13**

[**Evaluation**](#_a7tfzl7233sa) **14**

[**Input Evaluation with Wordnet Hypernyms and Sense Sets**](#_jmcl00sxjrqt) **14**

[**Loss Results**](#_na26gli5ngt5) **14**

[**Sample Outputs**](#_ihadgrdsrapr) **14**

[**Summarizing the Ethic Portion of this Paper (No Special Tokens)**](#_yfifwnhh6ncn) **14**

[**Summarizing the Update Expectations Portion of this Paper (No Special Tokens)**](#_2pbip0derfa7) **14**

[**Interesting Output from Generation Prompt “My cute dog”**](#_9gcbj34bfe1q)14

[**Continuing Timeline**](#_9qi5c52pd5ja) **15**

[**Updated Expectations**](#_w80vq68mcp7q) **16**

[**Deployment and UI**](#_7k4qx8alv9uo) **16**

[**Stretch Goals**](#_4vj05e7a541e) **17**

[**Incorporating Structured Commonsense Knowledge Metrics in Story Completion**](#_5foq8wu2153z) **17**

[**Possible Models to Adapt GPT2 Structure for Improved Storytelling**](#_ckj6q2ekzzfm) **17**

[**Story Cloze Test and ROCStories Corpora**](#_5fmdfllad9yz) **18**

[**Constraints**](#_tmjcsaob41y) **19**

[**Model Selection and Size**](#_9pb9mbfw53jy) **19**

[**Training Time Constraints**](#_jlsr1wg12au4) **19**

[**Deployment Constraints**](#_wn8lxyigmacr) **19**

[**UX**](#_elver14gtr50) **19**

[**Dev/ML Ops**](#_r9yfkul9d3c) **19**

[**Evaluating Output**](#_ez1wyxet6ky3) **20**

[**NLP Generative Metrics**](#_iaai5wt9uis0) **20**

[**Text Generation Ethics**](#_ggy0wlg9p98x) **20**

[**Training Data Ethics**](#_bvox2onqu64g) **20**

[**Generation Ethics**](#_ctfxyylfgzzg) **20**

[**Harmful Bias**](#_pzk3myouyx36) **21**

# 

# What is Epic?

Tired and bored of those lame campfire stories? Epic is an interactive text generation tool for styling a personalized horror, fiction story. By specifying a specific subgenre of horror (Witches, Ghosts, Monsters, etc.), an author’s style (Stephen King, Bram Stoker, Mary Shelly, etc.), and a set of protagonist traits, Epic will create a personalized story to fit your parameters. Epic will generate a paragraph and then ask the user for ideas. The user can either write their own line, give a summary to have the model incorporate the idea into the storyline, ask the model a question, or let the model continue to write. Along with this, the user can add or delete sections of the story the model has produced to be rewritten in the next round. This allows the user to control what happens in the story to a greater extent and provides an interactive component for the user.

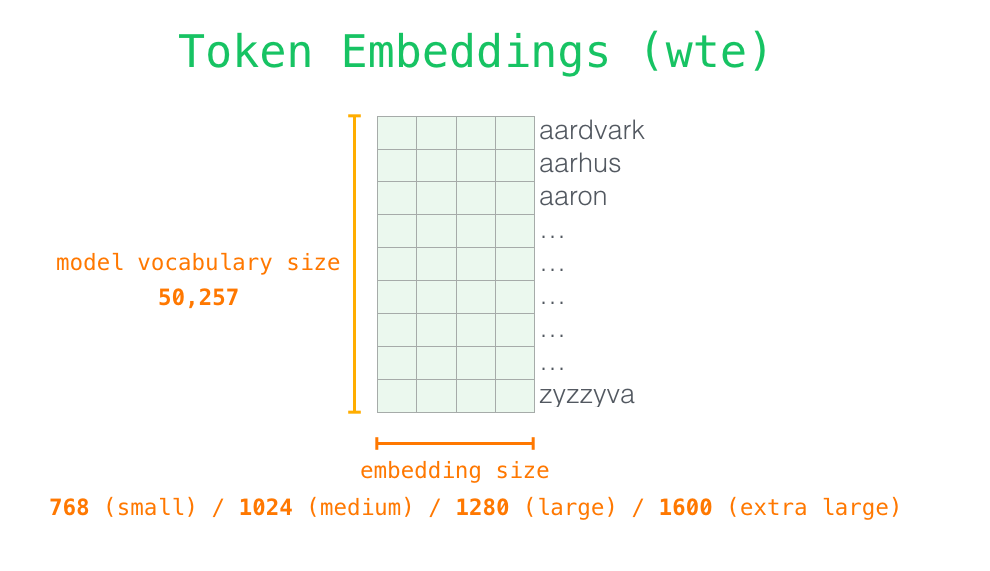
# Data

The final dataset includes 75mb of horror fiction text (~50,000 pages) collected by the group. The data ranges from movie scripts to public domain novels, to short stories from various authors. Although some authors only have one book in the dataset, some of the more prolific authors have 5 or more works. Stephen King for example represents 10% of the dataset. The dataset had to be this large to influence the GPT2 model we were working with. The final dataset was split into 70% training, 20% test, and 10% validation.

## Data Cleaning and Processing

To clean the dataset, the data was first converted to a uniform UTF8 format. General non-important information such as the publication, page numbers, and extra line breaks were scrubbed from the text with Regex techniques. Other text cleaning methods such as normalization, removing contractions, removing stopwords, and lemmatization were not performed due to the Transformer architecture that GPT2 uses. GPT2 uses word embeddings which have the benefit of encoding each word into a dense vector that captures something about its relative meaning within the training text. This means that variations in words like case, spelling, punctuation, and so on will automatically be learned to be similar in the embedding space. In turn, this means that the amount of cleaning required is less and quite different from classical text cleaning. Tomas Mikolov, one of the developers of word2vec, a popular word embedding method, suggests only very minimal text cleaning is required when learning a word embedding model.

# Feature Learning

The features of the data are represented in the model’s vocabulary as unique word vectors in the model’s token embedding as seen in the figure below.

## Special Tokenization

Word models represent words in a form called a token. Any text that is given to the model has to be tokenized and the output in return has to be decoded. Although all the words in the text will be converted into tokens, GPT2 has a way for us to specify additional ‘special’ tokens. When the model is generating text, by adding in a special token to the input context, the model is directed to format its output like the section that the token encapsulates. Some of the ‘built-in’ special tokens include <BOS> (Beginning of sequence), <EOS> (End of sequence), and <SEP> (A white space padding token). So this left the team with a huge problem: How can we all special tokens to all 50,000 pages of text and what are the most important tokens to add?

### Token Selection

To start, the team identified 5 important features to captivating writing: 1) use of visual language, 2) involving people in the story, 3) creative and surprising use of language, 4) subgenres, and 5) use of sentiment and emotions. The team identified 3 main token classes within these features: Data, Literary, and Subgenre Tokens. The data tokens encapsulate information that the team felt didn’t require interpretation to label. The literary tokens specified literary techniques that the team thought would be useful in story building. And the subgenre tokens declare which subgenre of horror the text is a part of. The tokens the team decided on are as follows:

|  |  |  |
| --- | --- | --- |
| Data Tokens | Literary Tokens | Subgenre Tokens |
| * Author * Chapter * Dialogue * Introduction Sentence * Conclusion Sentence * End of Story | * Characterization * Character * Weapon * Action * Danger * Death * Suspense * Surprise * Emotion * Climax * Problem * Conflict * Plot * Context * Exposition * Narrative * Perspective * Transition * Setting * Dialogue * Resolution * Relief * Metaphor/Simile * Flashback * Inciting Incident (comes before conflict) | * Vampires * Ghosts * Murder * Werewolf * Apocalypse * Haunted House * Witches * Hell * Aliens * Gore * Monster * Zombies * Comedy * Murder * Miscellaneous |

### Simple Cases

For the cases that didn’t require any interpreting (The Data Tokens), Regex was used to add in the tokens into the text files. But when the problem was bigger than just identifying specific words or patterns in the text as these special tokens help to encapsulate a given context. We also had the problem of if we wanted to use special tokens for words that never appear in our training set (or at least not in the needed context). This meant that the text had to be interpreted and then labelled; with the size of the dataset, that didn’t seem feasible. The original idea was to train a BERT Named Entity Recognition (NER) model[[1]](#footnote-0), but this would still result in having to spend countless hours interpreting and labeling text to train a model for our specific task and didn’t allow the model to be as dynamic as the team had hoped. So what was the solution?

### BERT Zero-Shot Text Classification

To solve the problem of interpreting the text, the team selected a BERT model that was pre trained on the task of text classification. By giving the model a context and a set of possible labels, the model assigned independent probabilities that each label was in the given context. By setting a high threshold of 0.9, low probabilities were rejected and the labels with the highest probability were added as tokens to the input context.

### Blacklist Feature ‘Extraction’

Given the way the Transformer and word embedding architecture works, not many features of the text needed to be removed from the dataset. However, the team did decide to add a separate blacklist of words to make sure the model excludes from its output. This is as close as the model got to feature extraction. This feature isn’t applied before the model is trained though. The blacklist words are entered as GPT2’s ‘bad word’ tokens and are not only excluded from the final output, but removed in the decision process when the model is determining the next word to generate. This will be a feature that the user can control to remove harmful or profane words from the model’s output vocabulary.

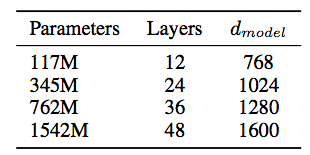
# Model & Pipeline

## GPT2

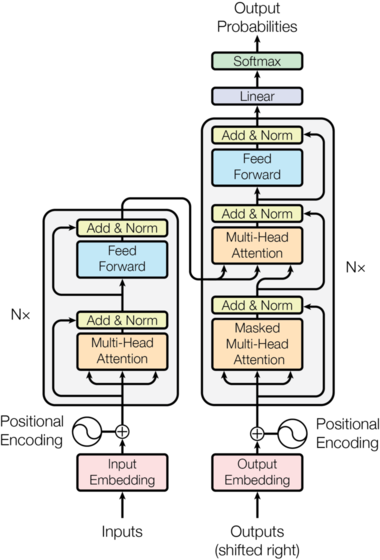
### Overview

Open AI states that their GPT2 model “is a large transformer-based language model with 1.5 billion parameters, trained on a dataset[[2]](#footnote-1) of 8 million web pages. GPT2 is trained with a simple objective: predict the next word, given all of the previous words within some text”. The purpose of having such a large model and feeding it a huge amount of data was to let the model learn and generalize information, patterns, and structure without human interference. After starting with representations of individual words or even pieces of words, GPT2 aggregates information from surrounding words to determine the meaning of a given bit of language in context. 

The result is a model that is capable of many NLP tasks such as: question and answering, fill in the blank, summarization, and text generation. So how do you use this pre trained model for your own task? The answer is finetuning the model on a subset of data that we want the model to imitate; in our case, that’s writing horror fiction.

Unfortunately, it is not feasible to fine-tune the 1558MB model on a single GPU due to RAM constraints. Loading the 1.5B parameter model requires more than 16Gb of RAM not including any other processes or models that need to be loaded or run. This is further discussed in the constraints section, but summed up, due to cost and time constraints, the team chose to start with the 345M “medium” model. Even at this size, GPT2 has twice the amount of parameters than the original GPT. This resulted in the model having a final architecture of 24 layers, 24 heads, and a vocabulary size of 50,257 words.

### GPT2 Architecture Overview



### GPT2’s Main Components

#### Encoder

The encoder (left hand side of the architecture) is composed of two blocks (called sub-layers to distinguish from the N blocks composing the encoder and decoder) which are the Multi-Head Self-Attention sub-layer, described in detail below, and a simple feed-forward network compose. Between each sub-layer, there is a residual connection[[3]](#footnote-2) followed by a layer normalization[[4]](#footnote-3). GPT2 uses Byte pair encoding (BPE)[[5]](#footnote-4) because word embeddings are too high level while pure character embeddings are too low level. BPE includes character level, subword level and word level embeddings.

#### Decoder

The decoder (right hand side of the architecture) is very similar to the encoder but has one Multi-Head Masked-Attention sublayer, also described in detail below, instead of the Self-Attention Heads. This network attends over the previous decoder states which is similar to the decoder hidden state in traditional machine translation architectures. The reason this is called the masked multi-head attention block is that we need to mask the inputs to the decoder from future time-steps meaning that the model is trained to predict sentences based on all the words before the current word.

#### Positional Encoding

Positional encodings explicitly encode the relative and absolute positions of the inputs as vectors which are then added to the input embeddings. The multi-head attention network cannot naturally make use of the position of the words in the input sequence. Without positional encodings, the output of the multi-head attention network would be the same for the sentences "I like cats more than dogs" and "I like dogs more than cats".

#### Optimizer

The Adam optimizer with 𝛃1= 0.9 and 𝛃2= 0.98 was used to train the model. Furthermore a learning rate scheduler was used to gradually warmed up the learning rate and then decreased it based on the learned warmup rate.

#### Residual Dropout

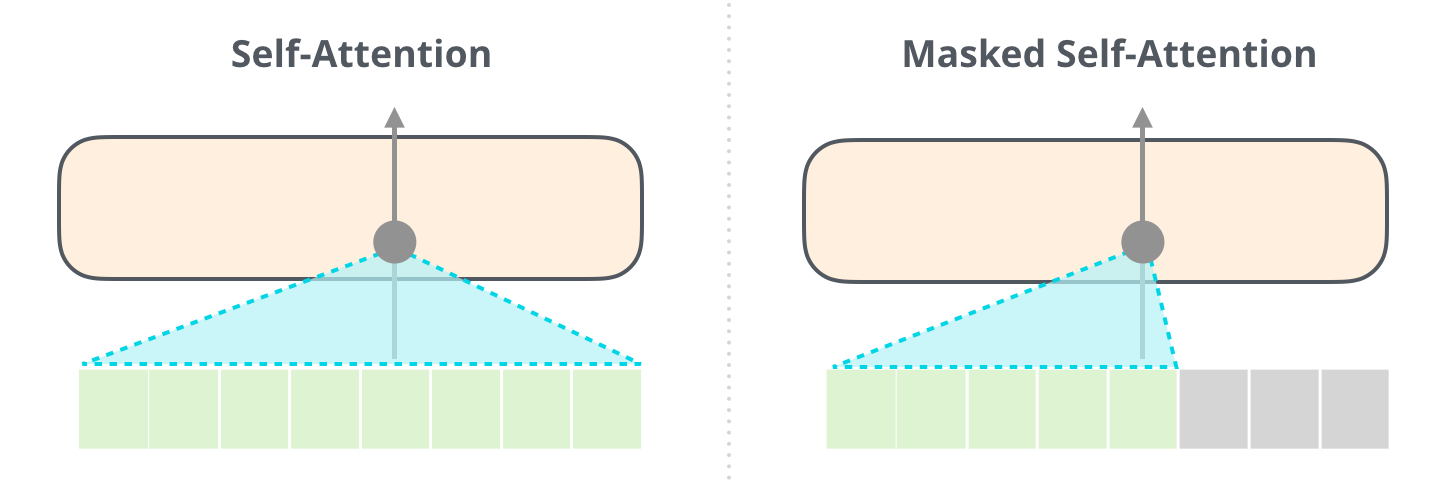
Dropout, 0.1 by default, is applied to each sublayer before adding it to the original input. The residual dropout is also applied to the sum of the embeddings and to the positional encodings.

#### Label Smoothing

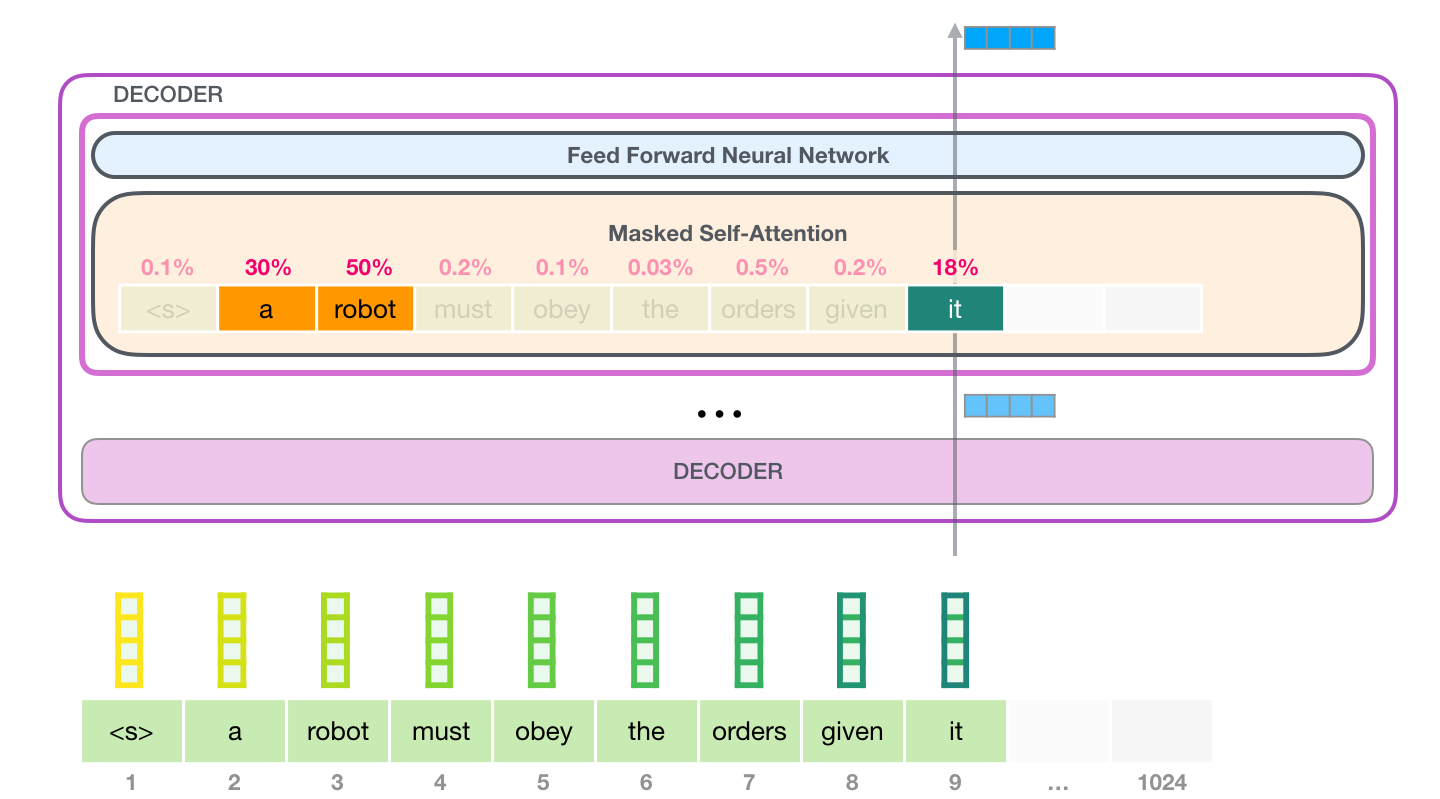
To penalize the model when it becomes too confident in its predictions, label smoothing is performed.

#### 

#### Masking

GPT2 uses normal self-attention for the input and masked self-attention for the output. A normal self-attention block allows a position to peak at tokens to its right. Masked self-attention prevents that from happening. The self attention mask is important when looking at the input so that the entire context of the input is considered. As an example, in Figure 6, the self-attention layer in the top block is paying attention to “a robot” when it processes the word “it”. The vector it will pass to the neural network is a sum of the vectors for each of the three words multiplied by their scores.



On the other hand, in the output, looking at future words to influence the current one would introduce a form of bias to what the model is able to output. When we do this, if we give the decoder access to the entire target sentence, the model can just repeat the target sentence (in other words, it doesn't need to learn anything). 

Attention improves the performance by applying a weight to the hidden states and adding together. But it turns out that that may not be quite enough though as it is still hard for the network to understand what is happening just from a weighted summation. The trick was to introduce heads. Each head is a weighted summation of the hidden states but with different weights. So, essentially, each head looks at a different variety of hidden states. Then, these are summed together into a single context vector which is sent to the decoder! This is called multi-head attention. Basically, the attention mechanism is used as a way for the model to focus on relevant information based on what it is currently processing. Traditionally, the attention weights were the relevance of the encoder hidden states (values) in processing the decoder state (query) and were calculated based on the encoder hidden states (keys) and the decoder hidden state (query).

##### Self-Attention Process

Self-attention bakes in the model’s understanding of relevant and associated words that explain the context of a certain word before processing that word (passing it through a neural network). It does that by assigning scores to how relevant each word in the segment is, and adding up their vector representation. Self-attention is processed along the path of each token in the segment.

Each layer of GPT2 retains its own interpretation of the first token and will use it in processing the second token. GPT2 does not re-interpret the first token in light of the second token. I also recommend running this [GPT2 Layer Visualization Notebook](https://colab.research.google.com/drive/1c9kBsbvSqpKkmd62u7nfqVhvWr0W8_Lx#scrollTo=XLu4wFlC0rK5) to see how scores for each word adjust in each layer.

The first block can now process the token by first passing it through the self-attention process, then passing it through its neural network layer. Once the first transformer block processes the token, it sends its resulting vector up the stack to be processed by the next block. This is called residual networks. These add the input back to the output in order to not lose information and they are, quite surprisingly, very effective! The process is identical in each block, but each block has its own weights in both self-attention and the neural network sublayers.

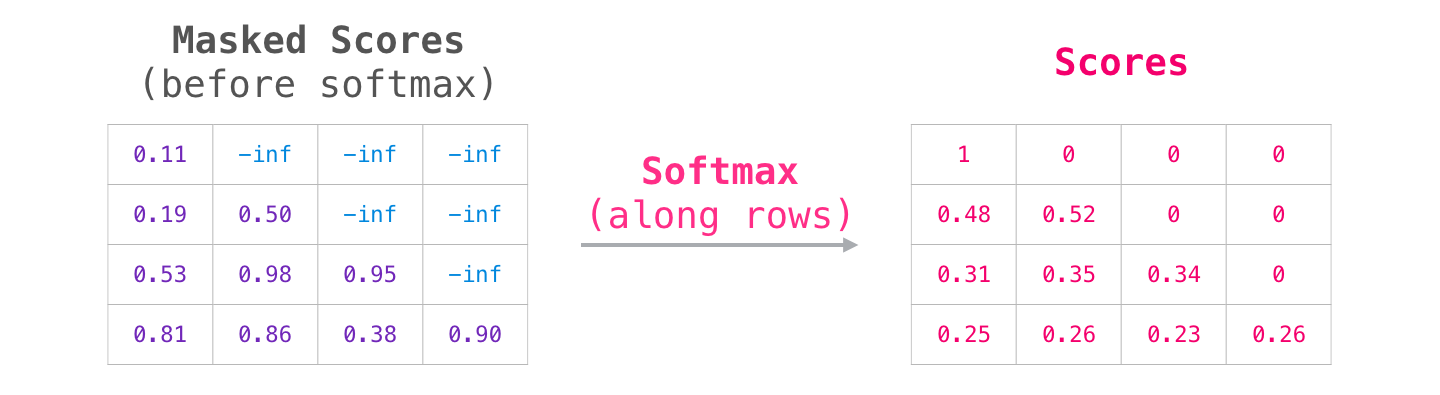
We multiply the current token vector and its query by all the other key vectors resulting in a score for each of the tokens currently generated in the segment. We can now multiply the scores by the value vectors. A value with a high score will constitute a large portion of the resulting vector after we sum them up. If we do the same operation for each path, we end up with a vector representing each token containing the appropriate context of that token.

##### Masked Self-Attention Process

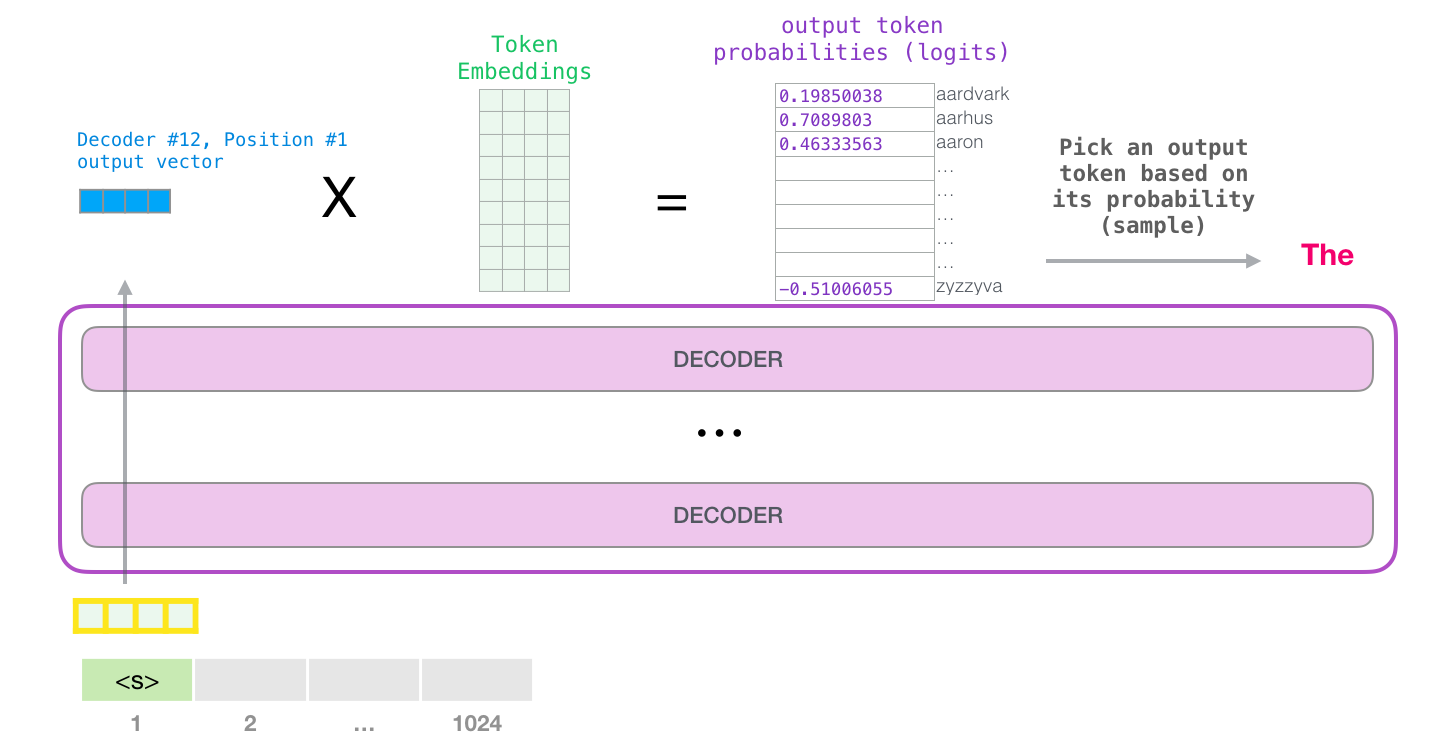
Masked self-attention is identical to self-attention except it basically always scores the future tokens as 0 so the model can’t peak to future words. We calculate the scores by multiplying a queries matrix by a keys matrix.



After the multiplication, we apply the attention mask. It set the cells we want to mask to -infinity or a very large negative number:

Then, applying softmax on each row produces the actual scores we use for self-attention. Multiplying the query vector by each key vector produces a score for each token.

### Model Output

When the top block in the model produces its output vector (the result of its own self-attention followed by its own feed forward network), the model multiplies that vector by the embedding matrix. Recall that each row in the embedding matrix corresponds to the embedding of a word in the model’s vocabulary. Thus the result of this multiplication is interpreted as a score for each word in the model’s vocabulary.

GPT2 uses GELU as its final activation function. With that, the model has completed an iteration resulting in outputting a single word. The model continues iterating until the entire context is generated (1024 tokens) or until an end-of-sequence token is produced.

### Hyperparameter Tuning

|  |  |  |
| --- | --- | --- |
| Hyperparameter | Basic Definition | Importance |
| top\_k | Integer value controlling diversity. 1 means only 1 word is considered for each step (token), resulting in deterministic completions, while 40 means 40 words are considered at each step. 0 (default) is a special setting meaning no restrictions. | Limiting the sample pool to a fixed size K could endanger the model to produce gibberish for sharp word distributions and limit the model's creativity for flat word distribution. This is one of the reasons for GPT2’s success in story generation. |
| top\_p | Top-p sampling chooses from the smallest possible set of words whose cumulative probability exceeds the float probability p. The probability mass is then redistributed among this set of words. | The size of the set of words can dynamically increase and decrease according to the next word's probability distribution. |
| temperature | Float value controlling randomness in boltzmann distribution. Lower temperature results in less random completions and will have a high chance to output data from WebText’s test set. As the temperature approaches zero, the model will become deterministic and repetitive. Higher temperature results in more random completions. | Humans don’t like to read things that don’t surprise them. If the model is deterministic, the outputted results are very boring to the user. |
| num\_beams | Number of hypotheses at each time step and eventually choosing the hypothesis that has the overall highest probability. | Beam search reduces the risk of missing hidden high probability word sequences by keeping the most likely num\_beams |
| no\_repeat\_ngram\_size | ‘n-grams’ (word sequences of n words) The most common n-grams penalty makes sure that no n-gram appears twice by manually setting the probability of next words that could create an already seen n-gram to 0. | We have seen that beam search heavily suffers from repetitive generation. This is especially hard to control with n-gram- or other penalties in story generation since finding a good trade-off between forced "no-repetition" and repeating cycles of identical n-grams requires a lot of finetuning. |
| repetition\_penalty | This is used to penalize words that were already generated or belong to the context. | It can be quite effective at preventing repetitions, but seems to be very sensitive to different models and use cases, |
| num\_return\_sequences | Number of output sequences returned | This is useful for evaluating and determining the optimal response rather than only have one option. |

## Models to Finetune Output

### [Roberta-Large-MNLI](https://huggingface.co/roberta-large-mnli?text=I+like+you.+%3C%2Fs%3E%3C%2Fs%3E+I+love+you.)

#### Entailment, Contradiction, Neutral Analysis

Entailment describes the hypothesis of a sentence with a similar meaning as the premise. If the truth of one sentence entails (or implies) the truth of the other sentence, the relation between those sentences is usually referred to as entailment. While paraphrases have the same truth conditions and always entail each other (symmetrical entailment), entailment in the NLP context refers to when one sentence entails that the other sentence is true, but the reverse does not hold. Thus, many examples of entailment are based on hyponymy between lexical items: Mary loves flowers entails Mary loves roses.

Contradiction is the truth of one sentence implies the falseness of the other: if one sentence is true, the other is necessarily false. This kind of relation is referred to as contradiction or negative entailment. In many cases, the contradiction is based on antonyms between lexical items: ‘It is cold in here’ contradicts ‘It is hot in here’.

Neutral is a term to describe if a hypothesis is a sentence with mostly the same lexical items as the premise but a different meaning. Using these three metrics, the team is able to evaluate the ‘quality’ of an input or output.

### [BERT-Base-Multilingual-Uncased-Sentiment](https://huggingface.co/nlptown/bert-base-multilingual-uncased-sentiment?text=I+like+you.+I+love+you)

#### Sentiment Matching

The NLP task of sentiment analysis returns a probability of whether an inputted context has a positive or negative tone. In our case, the team theorizes that by selecting the output that has the most similar sentiment as the input context, rapid shifting in tone will be automatically removed and should help the flow of the writing. This boils down to matching the sentiment of a user’s input to the sentiment of the model’s output.

### Word Distance Analysis

Using the word to vec embedded model, the team is able to take an input and determine the primary subject and action in a sentence. From there, we are able to measure the distance between the words in the embedding latent space. Words that have a great distance would be less likely to appear in the same context. If an output’s subject and action have too great a distance, we can reject the output as irrelevant. On top of that, this helps throw out statements where the model confused homographs which would essentially change the context of the story randomly. These confused words are a part of the same word senses[[6]](#footnote-5). To improve the results, the word sense is also considered in the context which helps to not rule out the model’s creativity.

# Evaluation

## Input Evaluation with Wordnet Hypernyms and Sense Sets

The team was quick to find that the quality of the output is highly related to the quality of the input and without the use of special tokens, the generation became repetitive or was simply too random. This left the question of how to call the special tokens the team implemented without having the user manually specify them. To solve this problem, the team employed the Wordnet language model to find hypernyms and sense sets. By employing this, the team can check the user’s input to see if any of the word hypernyms or top *n* senses match any of the special tokens. We can then append the special token to guide the model in the output. The only decision left was to decide if the team should add these words as special tokens or if we should simply refer back to the token that it matched to. The team ended up deciding on the latter as to not dilute the quality and context of the special tokens that we did add.

## Loss Results

Medium Model: 500 Warm-up Steps for Learning Rate Scheduler

["epoch": 0.10888501742160278,

"learning\_rate": 2.5e-05,

"loss": 3.815122314453125,

"step": 500]

["epoch": 4.790940766550523,

"learning\_rate": 2.185792349726776e-06,

"loss": 2.7911875,

"step": 22000]

With the small model at 16000 train steps, a loss of 2.2 was observed before it plateaued (stopped learning)

## Sample Outputs

### Summarizing the Ethic Portion of this Paper (No Special Tokens)

“The team at Generation Ethics OpenAI has developed a language model that can be used to train anti-government, anti-Semitic, and anti-Muslim chatbots, as well as to train anti-government, anti-Semitic, and anti-Muslim chatbots for a variety of other applications, including spam, phishing…”

### Summarizing the Update Expectations Portion of this Paper (No Special Tokens)

“The Huggingface team has been working on an ML Ops Pipeline for the past few months, and has now released the first phase of the project: the Huggingface Transformers Model Hub The team has been working on the Huggingface Transformers Model Hub for the past few months, and has now released the first… The Generative Story Loop interface allows the user to generate their own stories using a dataset of stories that have been written by authors such as Stephen King, Mary Shelly, and many more, all of which can be viewed at…”

### Interesting Output from Generation Prompt “My cute dog”

"My cute dog was so upset and he yelled at me! He came to try and find me!" she said. Cecilia said she did not feel safe for about 2 hours until police came to her side. "I was screaming but I couldn't see because of the sun light," she said. "Then the police said there's nothing they could do. I kept saying my dog has to go but I don't believe that because I live right on top of them."

# Continuing Timeline

|  |  |
| --- | --- |
| October 25 - November 1 | * Finish Data Cleaning   + **Jonathon Kastner**   + **Scott Oelkers**   + **Skyler McMullen** * Add in special tokens to data   + **Jonathon Kastner**   + **Scott Oelkers**   + **Skyler McMullen** * Create Jupyter Widgets Interface   + **Carson Stevens**   + **Nam Do**   + Start with Jupyter interactive HTML widget interface. * Create Project Update Document   + **Carson Stevens**   + **Nam Do**   + *Finalize with team at next meeting (November 1)* |
| November 1 - 15 | * Start final training   + **Carson Stevens** * Fine Tuning Output   + Sentiment Analysis     - **Jonathon Kastner**     - **Scott Oelkers**     - **Skyler McMullen**   + Hyperparameter Tuning     - **Skyler McMullen**   + Word Subject Distance Rejection     - **Carson Stevens** * Fine Tuning Input for Special Tokens (Hypernyms and Senses)   + **Nam Do**   + **Carson Stevens** * Formatting User input into pipeline   + **Carson Stevens** * Finalize Generation Input Loop for Interface   + **Nam Do** |
| November 15 - 22 | * *Finish training* * ***Continued*** Finetune output results * ***Continued*** Finetune input response |
| November 22 - 29 | * Make sure it’s *accessible*   + **Everyone** *get user feedback* * Refactor environments for production   + **Carson** * Optimization/Refactoring/Debugging   + **Everyone** |
| November 29 - December 6 | * Implement User feedback   + **Everyone** * Optimization/Refactoring/Debugging   + **Everyone** * Finalize product for deployment on Transformers Model Hub   + Include both the PyTorch and Tensorflow model for accessibility   + Documentation     - Include model card     - Include instructions to use in pipeline or as a generative model     - **Everyone** * Start Final Presentation   + Divide Presentation Parts   + **Everyone** |
| December 6 -TBD | * Deploy to Transformers Model Hub   + **Carson** * Practice Presentation/Finalize Video   + **Everyone** |
| Final: **TBD** | **Final Presentation/Video** |

# Updated Expectations

## Deployment and UI

As further discussed in the Constraints section below, the team had originally wanted to create an application that we could deploy and have any user be able to access it. Since the team had experience with Flask, this originally felt like a possible idea. Unfortunately, due to the costs of deployment, the inexperience of the ML Ops Pipeline, and the size of deploying the models requiring a more full-stack framework like Django instead of Flask’s microframework, the team decided to reframe the project to simply publish our model (in Pytorch and Tensorflow format) to the Huggingface Transformers Model Hub. The team plans to add details about how to use the model with the Transformers library from the base model and from the pipeline module found in the Transformers library. As discussed in the Ethical section of the document, the team also plans to publish the model card in hopes of helping future users determine bias in the model. Once published to the Model Hub, anyone with the Huggingface Transformer library installed will be able to download and run the model simply by loading the model ID by name from the Hub. By uploading the 0model to the Hub, Huggingface automatically creates an interface for developers to see sample outputs from their inputs.

Although the idea of deploying a full application seems out of reach, the team still wanted to have an easy interface to interact with the model. To find a middle ground between no UI and an application, the team decided to use the Jupyter ipywidgets library to create an interactive interface for the development environment. Part of the reasoning behind choosing this framework for the UI was that the team could code the interface in HTML, CSS, and Javascript which would allow for easier integration into a final app if that became a more reasonable choice. The current interface design contains two main components: 1) Story Initialization and 2) The generative story loop.

The Story Initialization section allows the user to set up the story by choosing one or more tokens from the author and subgenres that are in the dataset. For example, a user could choose the options of Stephen King and Mary Shelly with the sub genres ‘werewolf’ and ‘apocalypse’ to start creating an apocalypse based werewolf novel in the style of both Stephen King and Mary Shelly. The user can also define any blacklist words they don’t want to appear in the text. After that, the user defines the characters in the story by giving a small summary of the character and their attributes. The user can also define any additional aspects that they would want to appear in their text such as a scene or situation. This input will just be context and not actually direct input for the story. This input is fed into a summarization pipeline that will take the users information and create the initial storyline given the information. After the user has entered all possible characters and any other details, they can begin their story and we enter into the second interface.

The Generative Story Loop interface displays the current storyline in an editable textbox that will allow the user to directly edit the current story. This means that they can add in or remove lines as well as change any of the currently produced lines. The next part is a set of inputs that allow the user to choose from asking the model a question about the story, summarizing a new aspect of the story, providing the next line of the story, or simply let the model continue to generate where it left off. Additional aspects to the interface such as a microphone input and text to speech for reading the story aloud has also been implemented. With this simple interface, users should be able to generate their stories simply by running the notebook. While this puts the pressure on the user to provide the necessary computational resources, the team felt that this was the best option given our experience.

## Stretch Goals

### Incorporating Structured Commonsense Knowledge Metrics in Story Completion[[7]](#footnote-6)

The paper introduced in the footnote above describes three new metrics for evaluating a model and a ‘Combination Gate’ to weight the metrics. Since the cross-entropy loss that is currently used isn’t great for the purpose of evaluating longform writing, by implementing the proposed algorithms as a new loss function. This would help guide the model toward story generation instead of ‘simple’ text generation. The proposed metrics are as follows:

* Narrative Sequence: To describe a consistent story, plots should be planned in a logically reasonable sequence; that is there should be a narrative chain between different characters in the story.
* Sentiment Evolution: Besides narrative sequence, getting a good sentiment prediction model is also important for choosing the correct endings. Note that stories are different from other objective texts(e.g., articles), as they have emotions within the context. Usually there is a sentiment evolution when a storyline is being revealed.
* Commonsense Knowledge: Narrative sequence and sentiment evolution, though useful, are not sufficient to make correct predictions. In a typical story, newly introduced key-words may not be explained in the story because story-writers are not given enough narrative space and time to develop and describe them (Martin and George 2000). In fact, there are many hidden relationships among key-words in natural stories.

### Possible Models to Adapt GPT2 Structure for Improved Storytelling

* Narrative Chain Model(Chambers, Jurafsky et al. 2008)[[8]](#footnote-7): The Narrative Chain Model induces a new representation of structured knowledge called narrative event chains (or narrative chains). Narrative chains are partially ordered sets of events centered around a common protagonist. They are related to structured sequences of participants and events that have been called scripts (Schank and Abelson, 1977) or Fillmorean frames. These participants and events can be filled in and instantiated in a particular text situation to draw inferences. Chains focus on a single actor to facilitate learning, and thus this paper addresses the three tasks of chain induction: narrative event induction, temporal ordering of events and structured selection (pruning the event space into discrete sets).
* Msap(Schwartz et al. 2017): Msap uses a linear classifier based on language modeling probabilities of the entire story, and utilizes linguistic features of the ending sentences. These ending style features include sentence length, word and character n-gram in each candidate ending (independent of story).
* HCM(Chaturvedi, Peng, and Roth 2017): HCM uses FCSemLM (Peng and Roth 2016) in order to represent events in the story, learns sentiment trajectories in a form of N-gram language model, and uses topic-words’ GloVe to extract topical consistency feature. It uses Expectation-Maximization for training.
* DSSM(Huang et al. 2013): DSSM first uses two deep neural networks to project the context and the candidate endings into the same vector space, and ending choices based on the cosine similarity of the context.
* Cai(Cai, Tu, and Gimpel 2017): Cai uses BiLSTM RNN with attention mechanisms to encode the body and ending of the story separately and uses a cosine similarity between their representations to calculate the score for each ending during the selection process.
* SeqMANN(Li et al. 2018): SeqMANN uses a multi-attention neural network and introduces semantic sequence information extracted from FC-SemLM as external knowledge. The embedding layer concatenates five representations including word embedding, character feature, part-of-speech (POS) tagging, sentiment polarity and negation. The model uses DenseNet to match the body with an ending.
* FTLM(Radford et al. 2018): FTLM solves the stories cloze test by pre-training a language model using a multilayer transformer on a diverse corpus of unlabeled text, followed by discriminative fine-tuning.

### Story Cloze Test and ROCStories Corpora

'Story Cloze Test'[[9]](#footnote-8) is a commonsense reasoning framework for evaluating story understanding, story generation, and script learning. This test requires a model to choose the correct ending to a four-sentence story. To enable the Story Cloze Test, researchers created a new corpus of five-sentence commonsense stories, 'ROCStories'. This corpus is unique in two ways: (1) it captures a rich set of causal and temporal common sense relations between daily events, and (2) it is a high quality collection of everyday life stories that can also be used for story generation.



The current problem with directly implementing this[[10]](#footnote-9) is that the dataset requires a specific input of 5 sentences and our model’s current output would be a long narrative. The proposed solution is to take our final trained model and then continue to fine-tune a new model on the task of summarizing stories into the ROCStories format of 5 sentences. From that point we could use the Cloze Test along with a plethora of other tests developed based on the ROCStories Corpora. Almost all of the proposed model improvements above were evaluated with the Cloze Test or a similar metric.

# Constraints

## Model Selection and Size

Everything about GPT2 is big; even a large 70mb dataset was needed to influence the model in fine tuning. As mentioned earlier, GPT2 requires a lot of RAM and this didn’t include the extra memory needed to run the other models for evaluating output. Problems can arise when the GPU runs out of memory. This was a common problem in training, but this can also happen when users pass longer prompts (> 200 words). PyTorch raises an Memory exception, but unfortunately results in a large memory leak.

To properly monitor GPU usage though, you have to use nvidia-smi to output the results on a set interval running on a separate thread. But even then, nvidia-smi isn't useful since memory usage stats aren't real-time precise (it only shows the peak memory usage of a process). In short, we’d have to write a microservice to monitor a microservice which didn’t feel like it fit into the timeline or our knowledge scope. One solution to handle this was to catch PyTorch Memory exceptions and force releasing unused memory. Even with this, the memory problem has interrupted training and has been a real headache to continually run and restart from checkpoints. During usage, both CPU and GPU usage spike somewhat linearly. The lowest amount of vCPUs could spike to 80-100% with auto-scaling based on CPU usage. But with too many vCPUs, the CPU usage % won't budge while the GPU is hammered. Another solution the team tried was using gradient checkpointing to handle memory issues, but even still, fine-tuning GPT2-large (774M parameters) isn't possible on single GPUs without memory issues.

### Training Time Constraints

Training the medium model on a K80 with all of our data and a batch size of 10 for 3 epochs resulted in an estimated training time of 600 hours (~25 days). Upgrading to the P100 cut the training time by 3 which the team felt was reasonable given the timeline. The team would have loved to try training the GPT2-XL (1.5B parameters), but due to the even smaller batch size and size of the model in general, the training time would have been months. The team has considered upgrading the system to reduce time, but due to the memory leaks and complexity of distributed training decided to not go this route. I feel it is important to note that the largest model can be trained in only a few hours using 8 V100s. Using the same setup can train a small model in 25 minutes for $25. So even though training is feasible, it is out of the range of resources the team has on hand.

### Deployment Constraints

#### UX

On top of the above constraints, the team also had to settle on what a minimum rate of generation was. If the generation rate was too slow, users would feel frustrated and not be able to enjoy the novel experience. Running on just a CPU resulted in a measly 3-7 words/second, but using a “medium” model, the team was able to generate text at a rate of 24-36 words/second with a P100 which was acceptable, but faster generation might be able to be achieved with a distilled model. The current time to load a GPT2 model in PyTorch is (1-minutes)

#### Dev/ML Ops

It is not hard to find the horror stories of unfortunate cases where people tried to deploy their models to production, but failed to limit the number of micro instances that the server can spawn. This meant that every time a new user joined while another was on the site, another GPU was instantiated. Considering the size of needed GPUs, the cost can very quickly add up. One GPC user reported having a $40,000 bill after deploying. Due to the limited experience of the team in ML Ops pipeline deployment, the team no longer feels it would be wise to deploy a model publicly.

|  |  |  |  |
| --- | --- | --- | --- |
| **GCP Chip** | **Speed (words/sec)** | **Memory** | **VM Instance**  **Cost/hr** |
| Cascade Lake (CPU) | 8-11 | 26 gb | $0.03398 / vCPU  $0.00455 / GB |
| K80 (GPU) | 12-24 | 11 gb | $0.45 |
| P100 (GPU) | 32-64 | 16 gb | $1.46 |
| V100 (GPU) | 32-64 | 16 gb | $2.48 |

## Evaluating Output

### NLP Generative Metrics

The main problem with evaluating the model was determining performance. In general, scoring cross-entropy loss with a language model in the task of filling in the blank or question & answering is straightforward. However, longform generative output is hard to gauge and typically subjective to the reader. It isn’t hard to find examples where users thought a smaller model produced better output or where the data used to fine tune a larger model resulted in incoherent output. The cross-entropy loss was still used to evaluate as this was the metric that was used to evaluate the original model’s performance. In general this means that we are comparing how similar the generated text is to the validation and test text sets, but not how good a narrative the story is.

As mentioned above, there is a proposed Commonsense Knowledge Metrics for Story Completion that we could change the loss function to instead of the standard cross-entropy loss, but this would require modifying the complex GPT2 architecture. Since this would also require retraining our models on this metric, the idea has been put on hold, but is a stretch goal for the team.

## Text Generation Ethics

### Training Data Ethics

Since the dataset we used to finetune the GPT2 model is a composition of copyrighted works there was an additional check that the team has added to the output validation. The general consensus is that as long as the information can be found, it can be used for training. The limitation is that the model can’t produce any text that can be reconstructed from the training data; so as long as the model doesn’t generate phrases from the training set, the output should be ethical. To ensure this is the case, the generated text is passed to a plagiarism API.

### Generation Ethics

OpenAI’s “partners at the Middlebury Institute of International Studies’ Center on Terrorism, Extremism, and Counterterrorism (CTEC) found that extremist groups can use GPT-2 for misuse, specifically by fine-tuning GPT-2 models on four ideological positions: white supremacy, Marxism, jihadist Islamism, and anarchism.” There is also GPT-2’s potential to augment high-volume/low-yield operations like spam and phishing. We acknowledge that we cannot be aware of all threats, and that motivated actors can replicate language models without model release.

### Harmful Bias

Language models have biases which are introduced by the data they are trained on. Working out how to study these biases, discuss them, and address them, is a challenge for the AI research community. The team approached the challenge of bias in two ways: Publishing a model card, and performing a qualitative, in-house evaluation of some of the biases in GPT-2: We probed GPT-2 for some gender, race, and religious biases, tokens (mentioned in the Black List) to include as invalid word tokens. These probes are not comprehensive and raise the need for collaboration on bias analysis frameworks. [Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings](http://papers.nips.cc/paper/6228-man-is-to-computer-programmer-as-woman-is-to-homemaker-d) was the paper that exposed gender bias in word embedding models.

1. NER is “a sub-task of information extraction that seeks to locate and classify named entities in text into predefined categories such as the names of persons, organizations, locations, expressions of times, quantities, monetary values, percentages, etc.” [Source](https://towardsdatascience.com/a-review-of-named-entity-recognition-ner-using-automatic-summarization-of-resumes-5248a75de175) [↑](#footnote-ref-0)
2. The dataset “emphasizes diversity of content, by scraping content from the Internet. In order to preserve document quality, we used only pages which have been curated/filtered by humans—specifically, we used outbound links from Reddit which received at least 3 karma. This can be thought of as a heuristic indicator for whether other users found the link interesting (whether educational or funny), leading to higher data quality than other similar datasets.” [Open AI GPT2 Blog](https://openai.com/blog/better-language-models/) [↑](#footnote-ref-1)
3. A residual connection is basically just taking the input and adding it to the output of the sub-network, and is a way of making training deep networks easier. [↑](#footnote-ref-2)
4. Layer normalization is a normalization method in deep learning that is similar to batch normalization [↑](#footnote-ref-3)
5. BPE is a way of compression originally. A list of subwords will be calculated by using the following algorithm: Split word to sequence of characters. Join the highest frequency pattern Keeping doing the previous step until it hits the predefined maximum number of sub-word of iterations. [↑](#footnote-ref-4)
6. In linguistics, a word sense is one of the meanings of a word. Words are in two sets: a large set with multiple meanings (word senses) and a small set with only one meaning (word sense). For example, a dictionary may have over 50 different senses of the word "play", each of these having a different meaning based on the context of the word's usage in a sentence, as follows:

   * We went to see the play Romeo and Juliet at the theater.
   * The coach devised a great play that put the visiting team on the defensive.
   * The children went out to play in the park.

   [↑](#footnote-ref-5)
7. [Here is the paper](https://arxiv.org/pdf/1811.00625.pdf) that proposes theIncorporating Structured Commonsense Knowledge in Story Completion. [↑](#footnote-ref-6)
8. [Here is the paper](https://nlp.stanford.edu/pubs/narrative-chains08.pdf) that proposes the Narrative Chain Model. [↑](#footnote-ref-7)
9. Linked is the homepage for the [Story Cloze Test](https://cs.rochester.edu/nlp/rocstories/) and ROCStories Corpora. [↑](#footnote-ref-8)
10. Huggingface Transformers details how to train GPT2 on ROCStories [here](https://huggingface.co/transformers/v1.1.0/examples.html#fine-tuning-openai-gpt-on-the-rocstories-dataset). [↑](#footnote-ref-9)