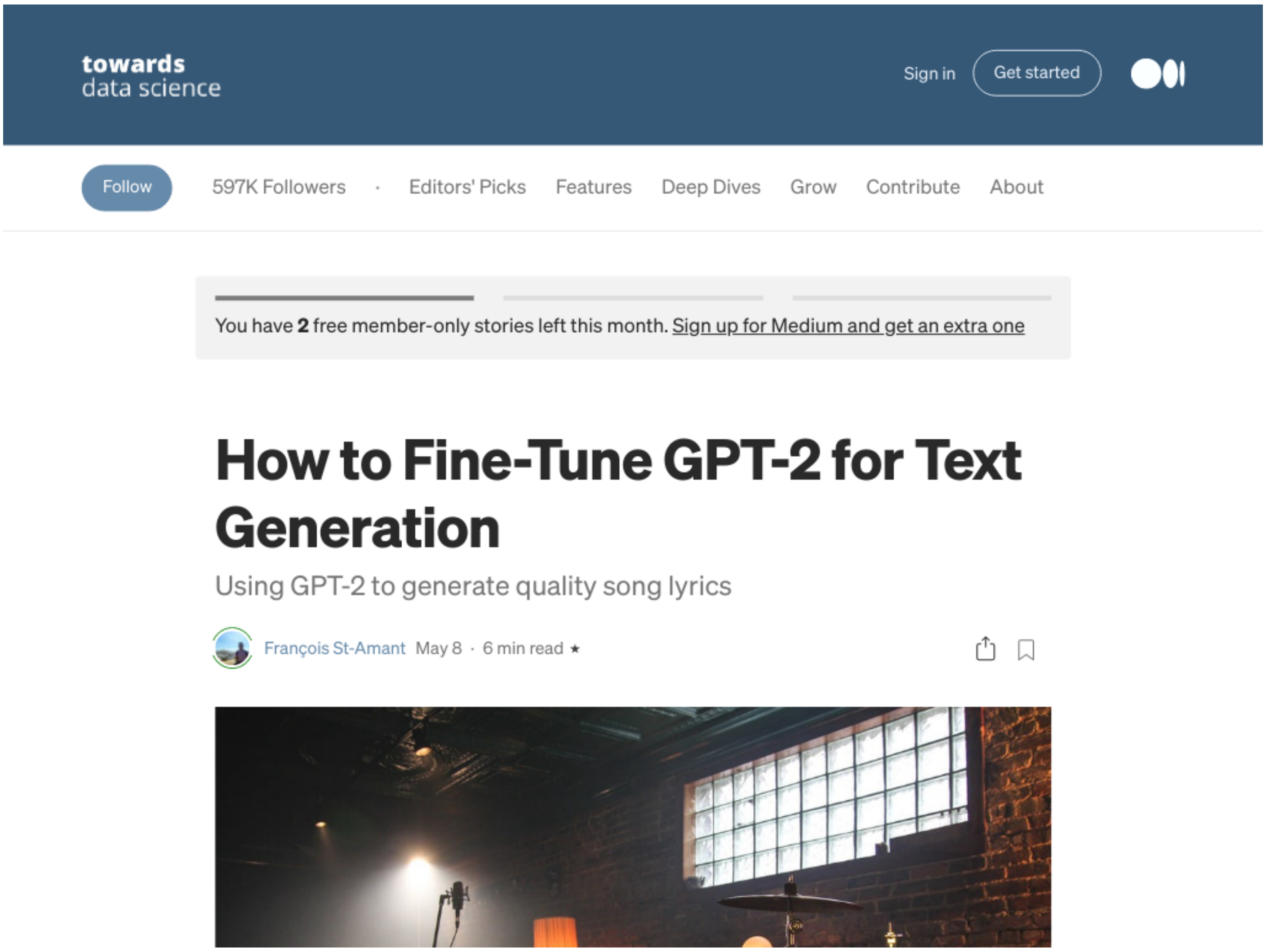
Unknown Title

: 24/5/2021



597K Followers · Editors' Picks Features Deep Dives Grow Contribute About

You have **2** free member-only stories left this month. Sign up for Medium and get an extra one

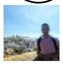
How to Fine-Tune GPT-2 for Text Generation

Using GPT-2 to generate quality song lyrics

François St-Amant May 8 · 6 min read ★

How to Fine-Tune GPT-2 for Text Generation

Using GPT-2 to generate quality song lyrics



François St-Amant

May 8·6 min read★

Source: https://unsplash.com/photos/gUK3lA3K7Yo

Natural Language Generation (NLG) has made incredible strides in recent years. In early 2019, OpenAI released GPT-2, a huge pretrained model (1.5B parameters) capable of generating text of human-like quality.

Generative Pretrained Transformer 2 (GPT-2) is, like the name says, based on the Transformer. It therefore uses the attention mechanism, which means it learns to focus on previous words that are most relevant to the context in order to predict the next word (for more on this, go here).

The goal of this article is to show you how you can fine-tune GPT-2 to generate context relevant text, based on the data you provide to it.

As an example, I will be generating song lyrics. The idea is to use the already trained model, fine-tune it to our specific data and then, based on what the model observes, generate what should follow in any given song.

Prepare the data

GPT-2 on it's own can generate decent quality text. However, if you want it to do even better for a specific context, you need to fine-tune it on your specific data. In my case, since I want to generate song lyrics, I will be using the following Kaggle dataset, which contains a total of 12,500 popular rock songs lyrics, all in English.

Let's begin by importing the necessary librairies and preparing the data. **I recommend using Google Colab for this project, as the access to a GPU will make things much faster for you.**

```
1  import pandas as pd
2  from transformers import GPT2LMHeadModel, GPT2Tokenizer
3  import numpy as np
4  import random
5  import torch
6  from torch.utils.data import Dataset, DataLoader
7  from transformers import GPT2Tokenizer, GPT2LMHeadModel, AdamW, get_linear_schedule_with_warmup
8  from tqdm import tqdm, trange
9  import torch.nn.functional as F
10 import csv
11
12 ### Prepare data
13 lyrics = pd.read_csv('lyrics-data.csv')
14 lyrics = lyrics[lyrics['Idiom']=='ENGLISH']
15
16 #Only keep popular artists, with genre Rock/Pop and popularity high enough
17 artists = pd.read_csv('artists-data.csv')
18 artists = artists[(artists['Genre'].isin(['Rock'])) & (artists['Popularity']>5)]
19 df = lyrics.merge(artists[['Artist', 'Genre', 'Link']], left_on='ALink', right_on='Link', how='inner')
20 df = df.drop(columns=['ALink','SLink','Idiom','Link'])
21
22 #Drop the songs with lyrics too long (after more than 1024 tokens, does not work)
23 df = df[df['Lyric'].apply(lambda x: len(x.split(' ')) < 350)]
24
25 #Create a very small test set to compare generated text with the reality
26 test_set = df.sample(n = 200)
27 df = df.loc[~df.index.isin(test_set.index)]
28
29 #Reset the indexes
30 test_set = test_set.reset_index()
31 df = df.reset_index()
32
33 #For the test set only, keep last 20 words in a new column, then remove them from original column
34 test_set['True_end_lyrics'] = test_set['Lyric'].str.split().str[-20:].apply(' '.join)
35 test_set['Lyric'] = test_set['Lyric'].str.split().str[:-20].apply(' '.join)
```

view raw lyrics1.py hosted with ❤ by GitHub

As can be seen in lines 26 and 34–35, I created a small test set where I removed the last 20 words of every song. This will allow me to compare the generated text with the actual one to see how well the model performs.

Create the dataset

In order to use GPT-2 on our data, we still need to do a few things. We need to tokenize the data, which is the process of converting a sequence of characters into tokens, i.e. separating a sentence into words.

We also need to ensure that every song respects a maximum of 1024 tokens.

The `SongLyrics` class will do just that for us during the training, for every song in our original dataframe.

```
class SongLyrics(Dataset):
def __init__(self, control_code, truncate=False, gpt2_type="gpt2", max_length=1024):

self.tokenizer = GPT2Tokenizer.from_pretrained(gpt2_type)
self.lyrics = []

for row in df['Lyric']:
self.lyrics.append(torch.tensor(
self.tokenizer.encode(f"<|{control_code}|>{row[:max_length]}<|endoftext|>")
))
if truncate:
self.lyrics = self.lyrics[:20000]
self.lyrics_count = len(self.lyrics)

def __len__(self):
return self.lyrics_count

def __getitem__(self, item):
return self.lyrics[item]

dataset = SongLyrics(df['Lyric'], truncate=True, gpt2_type="gpt2")
```

view raw lyrics2.py hosted with ❤ by GitHub

Training of the model

We can now import the pretrained GPT-2 model, as well as the tokenizer. Also, like I mentionned earlier, GPT-2 is HUGE. It is likely that if you try to use it on your computer, you will be getting a bunch of `CUDA Out of Memory` errors.

An alternative that can be used is to **accumulate the gradients**.

The idea is simply that before calling for optimization to perform a step of gradient descent, it will sum the gradients of several operations. Then, it will divide that total by the number of accumulated steps, in order to get an average loss over the training sample. That means much fewer calculations.

```
#Get the tokenizer and model
tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
model = GPT2LMHeadModel.from_pretrained('gpt2')

#Accumulated batch size (since GPT2 is so big)
```

```
def pack_tensor(new_tensor, packed_tensor, max_seq_len):
if packed_tensor is None:
return new_tensor, True, None
if new_tensor.size()[1] + packed_tensor.size()[1] > max_seq_len:
return packed_tensor, False, new_tensor
else:
packed_tensor = torch.cat([new_tensor, packed_tensor[:, 1:]], dim=1)
return packed_tensor, True, None
```

view raw lyrics3.py hosted with ❤ by GitHub

And now, finally, we can create the training function that will use all our song lyrics to fine-tune GPT-2 so that it can predict quality verses in the future.

```
def train(
dataset, model, tokenizer,
batch_size=16, epochs=5, lr=2e-5,
max_seq_len=400, warmup_steps=200,
gpt2_type="gpt2", output_dir=".", output_prefix="wreckgar",
test_mode=False,save_model_on_epoch=False,
):
acc_steps = 100
device=torch.device("cuda")
model = model.cuda()
model.train()

optimizer = AdamW(model.parameters(), lr=lr)
scheduler = get_linear_schedule_with_warmup(
optimizer, num_warmup_steps=warmup_steps, num_training_steps=-1
)

train_dataloader = DataLoader(dataset, batch_size=1, shuffle=True)
loss=0
accumulating_batch_count = 0
input_tensor = None

for epoch in range(epochs):

print(f"Training epoch {epoch}")
print(loss)
for idx, entry in tqdm(enumerate(train_dataloader)):
(input_tensor, carry_on, remainder) = pack_tensor(entry, input_tensor, 768)

if carry_on and idx != len(train_dataloader) - 1:
continue

input_tensor = input_tensor.to(device)
outputs = model(input_tensor, labels=input_tensor)
```

```
loss = outputs[0]
loss.backward()

if (accumulating_batch_count % batch_size) == 0:
optimizer.step()
scheduler.step()
optimizer.zero_grad()
model.zero_grad()

accumulating_batch_count += 1
input_tensor = None
if save_model_on_epoch:
torch.save(
model.state_dict(),
os.path.join(output_dir, f"{output_prefix}-{epoch}.pt"),
)
return model
```

view raw lyrics4.py hosted with ❤ by GitHub

Feel free to play around with the various hyperparameters (batch size, learning rate, epochs, optimizer).

Then, finally, we can train the model.

```
model = train(dataset, model, tokenizer)
```

Using `torch.save` and `torch.load`, you could also save your trained model for future use.

Lyrics Generation

The time has come to use our brand new fine-tuned model to generate lyrics. With the use of the following two functions, we can generate lyrics for all songs in our test dataset. Remember, I had removed the last 20 words for every song. Our model will now, for a given song, look at the lyrics he has and come up with what should be the end of the songs.

```
def generate(
model,
tokenizer,
prompt,
entry_count=10,
entry_length=30, #maximum number of words
top_p=0.8,
temperature=1.,
):
model.eval()
generated_num = 0
generated_list = []

filter_value = -float("Inf")
```

```

with torch.no_grad():

for entry_idx in trange(entry_count):

entry_finished = False
generated = torch.tensor(tokenizer.encode(prompt)).unsqueeze(0)

for i in range(entry_length):
outputs = model(generated, labels=generated)
loss, logits = outputs[:2]
logits = logits[:, -1, :] / (temperature if temperature > 0 else 1.0)

sorted_logits, sorted_indices = torch.sort(logits, descending=True)
cumulative_probs = torch.cumsum(F.softmax(sorted_logits, dim=-1), dim=-1)

sorted_indices_to_remove = cumulative_probs > top_p
sorted_indices_to_remove[..., 1:] = sorted_indices_to_remove[
..., :-1
].clone()
sorted_indices_to_remove[..., 0] = 0

indices_to_remove = sorted_indices[sorted_indices_to_remove]
logits[:, indices_to_remove] = filter_value

next_token = torch.multinomial(F.softmax(logits, dim=-1), num_samples=1)
generated = torch.cat((generated, next_token), dim=1)

if next_token in tokenizer.encode("<|endoftext|>"):
entry_finished = True

if entry_finished:

generated_num = generated_num + 1

output_list = list(generated.squeeze().numpy())
output_text = tokenizer.decode(output_list)
generated_list.append(output_text)
break

if not entry_finished:
output_list = list(generated.squeeze().numpy())
output_text = f"{tokenizer.decode(output_list)}<|endoftext|>"
generated_list.append(output_text)

return generated_list
```

```
61
62 #Function to generate multiple sentences. Test data should be a dataframe
63 def text_generation(test_data):
64 generated_lyrics = []
65 for i in range(len(test_data)):
66 x = generate(model.to('cpu'), tokenizer, test_data['Lyric'][i], entry_count=1)
67 generated_lyrics.append(x)
68 return generated_lyrics
69
70 #Run the functions to generate the lyrics
71 generated_lyrics = text_generation(test_set)
```

view raw lyrics5.py hosted with ❤ by GitHub

The `generate` function prepares the generation, while `text_generation` actually does it, for the entire test dataframe.

In line 6 is where we specify what should be the maximum length of a generation. I left it at 30 but that is because punctuation counts, and I will later remove the last couple of words, to ensure the generation finishes at the end of a sentence.

Two other hyperparameters are worth mentioning:

- **Temperature** (line 8). It is used to scale the probabilities of a given word being generated. Therefore, a high temperature forces the model to make more original predictions while a smaller one keeps the model from going off topic.
- **Top p filtering** (line 7). The model will sort the word probabilities in descending order. Then, it will sum those probabilities up to p while dropping the other words. This means the model only keeps the most relevant word probabilities, but does not only keep the best one, as more than one word can be appropriate given a sequence.

In the following code, I simply clean the generated text, ensure that it finishes at the end of a sentence (not in the middle of it) and store it in a new column, in the test dataset.

```
#Loop to keep only generated text and add it as a new column in the dataframe
my_generations=[]

for i in range(len(generated_lyrics)):
a = test_set['Lyric'][i].split()[-30:] #Get the matching string we want (30 words)
b = ' '.join(a)
c = ' '.join(generated_lyrics[i]) #Get all that comes after the matching string
my_generations.append(c.split(b)[-1])

test_set['Generated_lyrics'] = my_generations


#Finish the sentences when there is a point, remove after that
final=[]

```

```
for i in range(len(test_set)):
to_remove = test_set['Generated_lyrics'][i].split('.')[-1]
final.append(test_set['Generated_lyrics'][i].replace(to_remove,''))

test_set['Generated_lyrics'] = final
```
view raw lyrics6.py hosted with ❤ by GitHub

Performance Evaluation

There are many ways to evaluate the quality of generated text. The most popular metric is called BLEU. The algorithm outputs a score between 0 and 1, depending how similar a generated text is to reality. A score of 1 indicates that every word that was generated is present in the real text.

Here is the code to evaluate BLEU score for the generated lyrics.

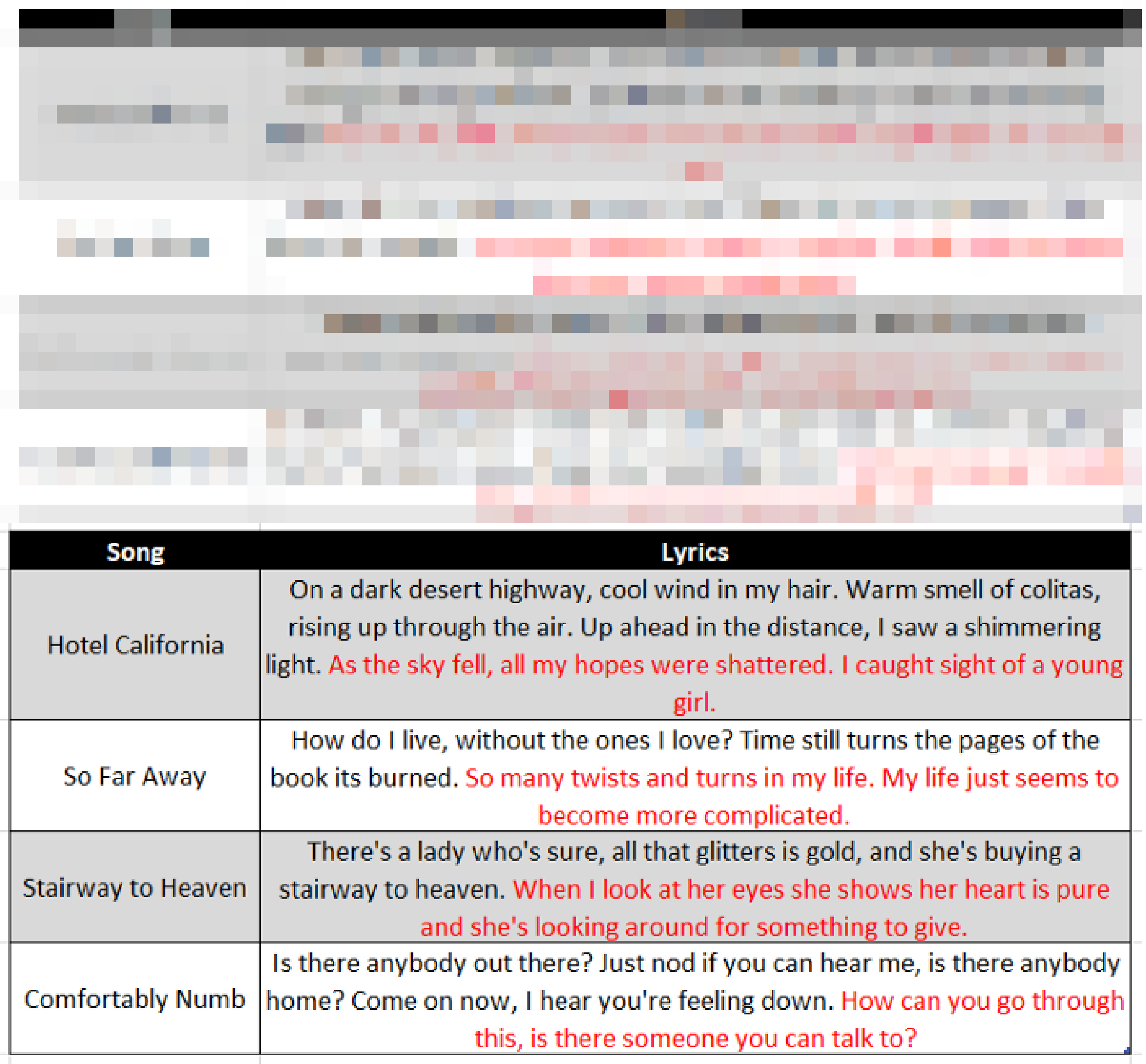
```
#Using BLEU score to compare the real sentences with the generated ones
import statistics
from nltk.translate.bleu_score import sentence_bleu

scores=[]

for i in range(len(test_set)):
reference = test_set['True_end_lyrics'][i]
candidate = test_set['Generated_lyrics'][i]
scores.append(sentence_bleu(reference, candidate))

statistics.mean(scores)
```
view raw lyrics7.py hosted with ❤ by GitHub

We obtain an average BLEU score of 0.685, which is pretty good. In comparison, the BLEU score for the GPT-2 model without any fine-tuning was of 0.288.

However, BLEU has it’s limits. It was originally created for machine translation and only looks at the vocabulary used to determine the quality of a generated text. That is a problem for us. Indeed, it is possible to generate high quality verses that use entirely different words than reality.

That is why I will do a subjective evaluation of the performance of the model. To do that, I created a small web interface (using Dash). The code is available on my Github repository.

The way the interface works is that you provide the app with some input words. Then, the model will use that to predict what the next couple of verses should be. Here are a few example outcomes.



Song	Lyrics
Hotel California	On a dark desert highway, cool wind in my hair. Warm smell of colitas, rising up through the air. Up ahead in the distance, I saw a shimmering light. As the sky fell, all my hopes were shattered. I caught sight of a young girl.
So Far Away	How do I live, without the ones I love? Time still turns the pages of the book its burned. So many twists and turns in my life. My life just seems to become more complicated.
Stairway to Heaven	There's a lady who's sure, all that glitters is gold, and she's buying a stairway to heaven. When I look at her eyes she shows her heart is pure and she's looking around for something to give.
Comfortably Numb	Is there anybody out there? Just nod if you can hear me, is there anybody home? Come on now, I hear you're feeling down. How can you go through this, is there someone you can talk to?

In red is what the GPT-2 model predicted, given the input sequence in black. You see that it has managed to generate verses that make sense and that respect the context of what came prior! Also, it generates sentences of similar length, which is extremely important when it comes to keeping the rythmn of a song. **In that regard, punctuation in the input text is absolutely essential when generating lyrics.**

Conclusion

As the article shows, by fine-tuning GPT-2 to specific data, it is possible to generate context relevant text fairly easily.

For lyrics generation, the model can generate lyrics that respect both the context and the desired length of a sentence. Of course, improvements could be made to the model. For instance we could force it to generate verses that rhyme, something often necessary when writing song lyrics.

Thanks a lot for reading, I hope I could help!

The repository with all the code and models can be found right here: https://github.com/francoisstamant/lyrics-generation-with-GPT2