Using GPT2 to generate Harry Potter text

Sander Bjerklund Lindberg
MSc Informatics / 2021
sanderbl@stud.ntnu.no

Abstract

Natural language generation is widely used. For example, it is used in smartphones' smart assistants such as Siri and Google Assistant, smartphone keyboards to predict the next word, or email clients to predict the rest of your sentence. One area of natural language generation that has not undergone a lot of research, however, is fictional text generation. This paper aims to fine-tune the pre-trained GPT-2 model to the Harry Potter universe to generate new and funny Harry Potter texts.

1 Introduction

Natural language generation (NGL) is widely used. One can find examples of NGL in smartphones' smart assistants such as Siri and Google Assistant, smartphone keyboards for next word prediction, or email clients predicting the rest of your sentence. One area of NGL that has not undergone a lot of research is that of fictional text generation. Recurrent Neural Networks (RNNs) and Long short term memories (LSTMs) were long used for text generation tasks. However, these are slow architectures, and RNNs suffer from the vanishing gradient problem. Therefore, in 2017, (Vaswani et al., 2017) presented the Transformer architecture for natural language processing, which is a much faster architecture that can "remember" more and minimizes the vanishing gradient problem.

This paper presents a study that uses Generative Pre-trained Transformer 2 (GPT-2) to generate new Harry Potter text, based on the seven Harry Potter books from J.K Rowling. Section 2 contains some background information about Recurrent Neural Networks, LSTMs, and Transformers and is followed by section 3 which briefly presents the architecture of GPT-2. Section 4 outlines the actual experiment, and section 5 evaluates the trained model.

2 Background

This section contains background information for the model used in this experiment.

2.1 Recurrent Neural Networks

In a typical feed-forward neural network, the information flows through the network and a sequence of mathematical operations without any consideration of the order of the inputs. This makes them unsuitable for data in which order matters. That's where Recurrent neural networks (RNNs) come into play. RNNs are a type of neural network. They are widely used for sequential data, such as text, stock prices, sensor data, language translation, and speech recognition.

RNNs work much like traditional feed-forward networks by having nodes that do mathematical operations on the data, activation functions, and gradient descent. They differ, however, in the sense that they are recurrent and can "remember" previous inputs. Where the traditional deep neural networks consider inputs and outputs as independent, recurrent neural networks do not (IBM Cloud Education, 2020). Each output is dependent on the previous elements in a sequence. That is - every output is used in the activation function together with the following input. This way, RNNs can use, e.g., the position of each word in a sentence to better predict the next word.

RNNs do, however, have a pretty short memory. For instance, if it were to predict what kind of drink someone should avoid, given an earlier sentence saying they are lactose intolerant, they might not be able to. Take, for instance, the sentence "I am lactose intolerant. I should not drink". The recurrent neural network would probably predict the correct word *milk*, since "I am lactose intolerant" is close to the word it should predict. However, if it had come in a sentence much earlier, it may have forgotten about it. This is because the model is sequential, and the first inputs have to "travel" through all the hidden layers, which multiply them with weights less than 0, causing the gradients to vanish.

2.2 Long Short-Term Memories

Long short-term memories (LSTMs) is a RNN architecture, introduced by Sepp Hochreiter and Jürgen Schmidhuber in 1997 (Hochreiter and Schmidhuber, 1997). This architecture aims to minimize the vanishing gradient problem. This is done by creating a new cell structure, which introduces multiple gates that control access to the cell's state (Di et al., 2018).

The LSTMs work like regular RNNs, with input to the cells containing the output of the last cell. However, it differs *in* each cell. As mentioned, it incorporates multiple gates, more accurately, three; the forget gate, the input gate, and the output gate. Each gate has its own task.

Before explaining the three gates, I provide some notation I will use. The inputs to a cell in LSTMs consists of the last cell's state (or memory) C_{t-1}, the previous cell's hidden state (or output) h_{t-1} and the current input x_t. h_{t-1} and x_t are concatenated into one input $[h_{t-1}, x_t]$, which I will denote inp_t

The forget gate has the job of choosing what the model can forget from the previous memory. E.g., in a sentence, the model does not have to remember the word "the" since it does not provide any meaningful information. The gate takes C_{t-1} and inp_t as input. inp_1 is run through the Sigmoid activation function, which maps each number in the vector to a number between 0 and 1. Values close to 0 mean to forget, and values close to 1 mean to keep. The resulting vector is multiplied with C_{t-1} to forget the values that should be forgotten, producing the forget gate output f_t.

The next gate is the input gate. It takes f_t and inp_t as input. The goal of this gate is to update this cell's memory. inp_t is being passed to a Sigmoid activation function to filter out again the important values, and a Tanh activation function to squish the values between -1 and 1. Passing the inputs to tanh helps regulate the network and minimize the vanishing gradient or exploding gradient problem. Then, the two activation function outputs are multiplied together. The Sigmoid outputs select what information to keep from the tanh output. The resulting vector is added to f_t to produce this cell's memory, C_t.

The last gate is the output gate. It takes inp_t and C_t as input. Inp_t is again passed to the Sigmoid function, and C_t is passed to Tanh. The two resulting vectors are multiplied together to decide what information the final output of the cell, h_t, should carry.

Even though LSTMs seem pretty great, by being able to "remember" or learn over 1000 time steps (Di et al., 2018), and by minimizing vanishing or exploding gradient problem, they are *incredibly* slow to train due to their sequential nature. That is why we needed a way to parallelize the sequential data.

2.3 Transformers

In 2017, researchers at Google and the University of Toronto presented the Transformer (Vaswani et al., 2017). The full Transformer architecture can be seen in Figure 1. Much like RNNs, the network uses an encoder/decoder architecture. It differs, however, in the sense that the inputs can be passed in parallel. Where an RNN would consider each word in "this is an input" as four individual inputs, the Transformer would consider the whole sentence as one input. As can be seen in the left part of Figure 1, the inputs are passed to an input embedding block. This is simply to map each word to a point in space, where words with similar meanings, such as "dog" and "cat," are close together. This, however, does not account for the word's position in the sentence. Therefore, the next step is *positional encoding*, which applies a function to the embedded input to account for each word's position in the sentence. Then, the positional encoded vector enters the *encoder* block.

The first step of the encoder block is to apply *attention* to the input. That is - how relevant is each input word relevant to the other words in the sentence? This is the main selling point of the Transformer. Attention in the Transformer is done by doing a scaled dot product between three vectors $Q = Query$, $K = Keys$ and $V = Values$. These vectors extract different components about a word. Q and K are multiplied, scaled, and softmaxed before the result is multiplied with V. The resulting vector is an attention vector, which captures the word's contextual relationship with every other word in the sentence. What makes the Transformer better than a standard RNN is that this step is done in parallel in what they call "Multi-Head Attention." The final output of the encoder block is a continuous representation of the sentence with attention information and is later used in the decoder block.

The right part of Figure 1 is called the *decoder*-block. Its job is to generate text sequences. As we can see from the figure, it contains more or less the same blocks as the encoder block, with one difference; the *Masked* Multi-Head Attention block. Since the decoder block is auto-regressive (output is dependent on previous values), the decoder has to mask some of the inputs. That is - each word should only have access to itself and all previous words. The second Multi-Head Attention block takes the output of the encoder as input, in addition to the previous attention block's output, allowing it to determine which of the encoder inputs is relevant to focus on. The output of the decoder is a vector containing softmaxed values for each word in your vocabulary. The index with the highest probability is the predicted word and is later used as input to the decoder block.

3 Architecture

This section contains the architecture of the model used in this experiment.

3.1 Generative Pre-trained Transformer 2

Generative Pre-trained Transformer 2 (GPT-2) is a pre-trained Transformer and successor to GPT created by researches at OpenAI in 2019 (Radford et al., 2019). It can be used for a great number of tasks, such as text generation, language translation and question answering systems. It is built with the Transformer architecture outlined in subsection 2.3 as its building blocks. GPT-2, however, only relies on the *decoder* block of the Transformer, more specifically, a block of decoders. It takes a series of tokens (or words) as input and outputs the next, most probable, word in the sentence. It can do this "out of the box", because it has already been trained on a large dataset (40GB) called WebText, which was generated by scraping Reddit[1] for web pages. In fact - GPT-2 does not actually require any start text at all, other than the "<|endoftext|>"-token. If this is passed as input, GPT-2 will ramble on its own, generating unconditional text (Alammar, 2019). The (simplified) architecture of GPT-2 can be seen in Figure 2. From the figure,

[1] `https://reddit.com`

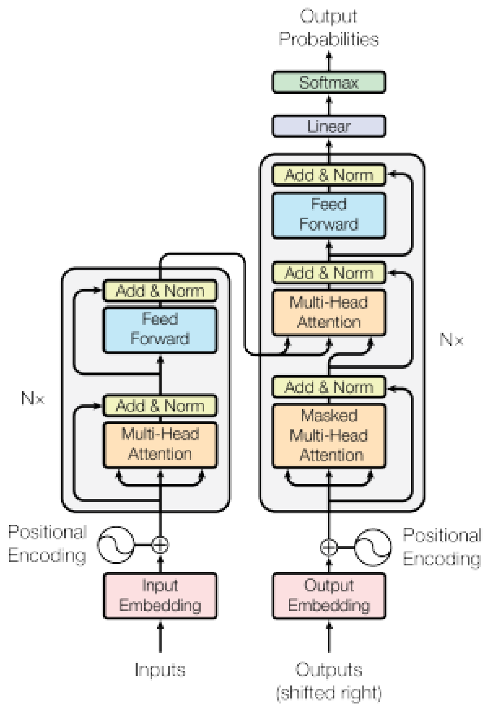


Figure 1: The Transformer model architecture (borrowed from Vaswani et al. (2017)).

we see that GPT-2 takes up to 1024 words as input. The input is embedded and added with the positional encoding of the input, as seen in the decoder block in Figure 1. Then, it enters the first decoder block, passes through a masked self-attention layer and a feed-forward neural network, before outputting a vector and passing this to the next decoder block. Each of the decoder blocks have its own weights with regards to the self-attention and neural networks. Eventually, after the last decoder block, the resulting vector is multiplied with the model's token embeddings and softmaxed, resulting in a probability table for all the models tokens. The chosen prediction word (the one with highest probability in the softmax table) is then added to the previous input and used in another run through the decoders.

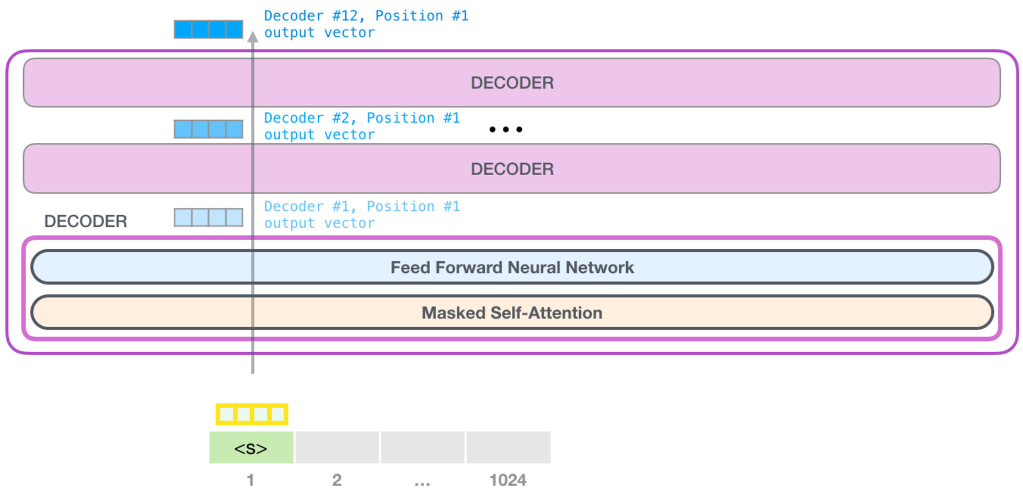
4 Experiments and Results

I am a big Harry Potter fan. Therefore, I wanted to create a model that, given start input, could generate new Harry Potter text. Luckily, GPT-2 (see section 3) was already created and trained to generate sequences of text in addition to being open source!

Transfer learning is the art of using what you have learned in one setting to learn something new—for instance, using your knowledge of Python as a programming language to learn how to write Java. Transfer learning is not limited to "real life" and human beings; it is also used in computing. Examples of transfer learning in, e.g., image classification is to take a model that is previously trained to classify images of cars to recognize motorcycles. Or, in my example, fine-tuning the pretrained GPT-2 model to generate Harry Potter text.

4.1 Data

The dataset used to train this experiment's model consists of all 7 Harry Potter books by J.K. Rowling. The books were collected from the GitHub repository Owlreader (Deng (2014)) as plain txt files. The files contain a lot of what I would consider noise, such as page numbers and chapter divisions, and names.



Decoder #12, Position #1 output vector
DECODER
Decoder #2, Position #1 output vector
DECODER
DECODER
Decoder #1, Position #1 output vector
Feed Forward Neural Network
Masked Self-Attention
<s>
1
2
...
1024

Figure 2: Simplified GPT-2 architecture. (Alammar, J (2019). The Illustrated GPT-2 [Blog post]. Retrieved from `https://jalammar.github.io/illustrated-gpt2/` Licenced under CC BY-NC-SA 4.0)

The main goal of this experiment is to generate new Harry Potter text based on an input string, so this metadata seemed redundant. Therefore, the first step in the preprocessing was to remove this noise. The chapter division and names were removed manually by going through each txt file. Then, I carefully removed the page numbers by splitting the text into sentences and removing the lines containing only numbers. The sentences would work as a training examples for the model. I then built a test set for each train set using the following sentence as the correct output for a given sentence. The last sentence of each book was succeeded by "The end" as the correct output in the test set. The train and test sets were then concatenated to create one extensive train set and one big test set for all seven books.

4.2 Model

As mentioned, the model used in this experiment was the pretrained GPT-2 model with a Language Model head, obtained from the python module Transformers (Wolf et al., 2020). It was trained with Huggingface's trainer (Huggingface, 2020b) class for 40 epochs on the dataset as described in subsection 4.1, with 400 eval steps, 800 warmup steps, and batch size of 8 per device, for both train and eval. The model was trained on one Nvidia Tesla P100 GPU for 3 hours.

The model also uses the GPT2Tokenizer from Huggingface (Huggingface, 2020a), which is based on byte-level Byte-Pair-Encoding. The tokenizer takes raw text as input and tokenizes it into id's the GPT2 model can understand.

4.3 Experimental Results

Figure 3 shows the loss graph during training of the model. The loss was plotted every 128 step of the training by using the Wandb package (Biewald, 2020) and the `report_to="wandb"` argument for the trainer. The figure shows that the loss is steadily decreasing, eventually beginning to flatten out around one. The loss is calculated using CrossEntropy loss.

Results from the trained model can be seen in Figure 4

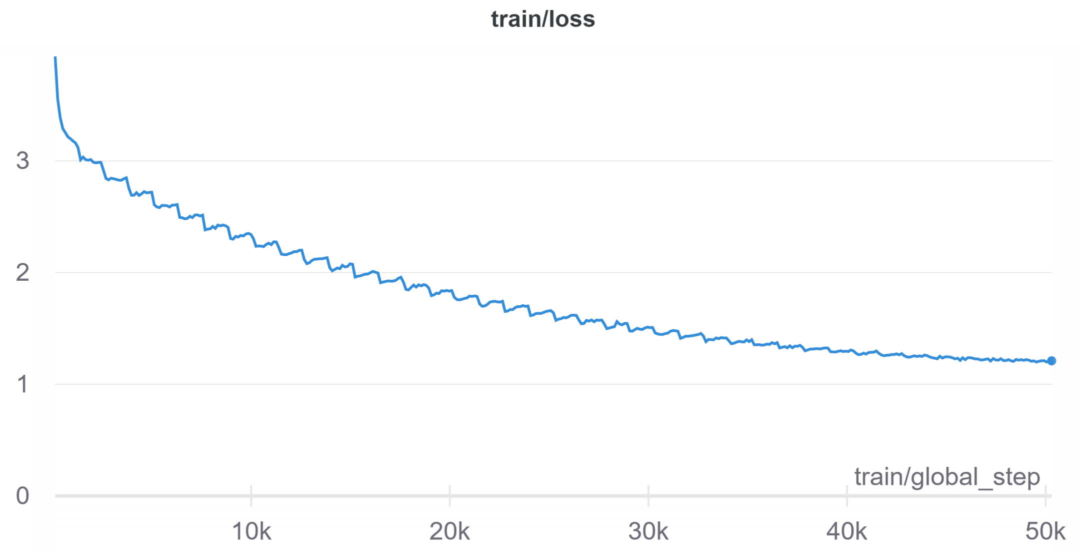
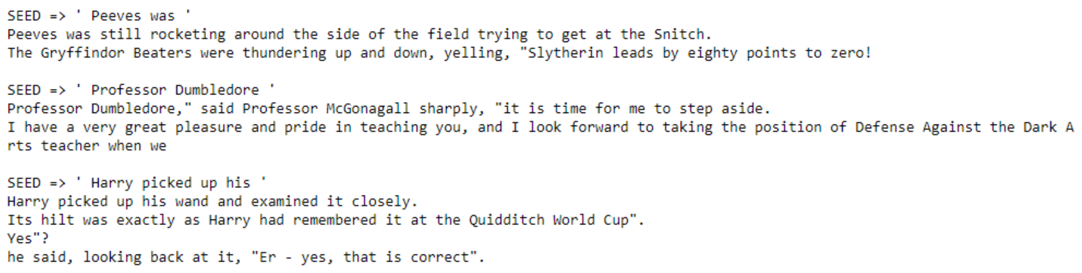


Figure 3: Loss during training of the GPT2 model. Loss is plotted every 128 step of the training.



```
SEED => ' Peeves was '
Peeves was still rocketing around the side of the field trying to get at the Snitch.
The Gryffindor Beaters were thundering up and down, yelling, "Slytherin leads by eighty points to zero!

SEED => ' Professor Dumbledore '
Professor Dumbledore," said Professor McGonagall sharply, "it is time for me to step aside.
I have a very great pleasure and pride in teaching you, and I look forward to taking the position of Defense Against the Dark A
rts teacher when we

SEED => ' Harry picked up his '
Harry picked up his wand and examined it closely.
Its hilt was exactly as Harry had remembered it at the Quidditch World Cup".
Yes"?
he said, looking back at it, "Er - yes, that is correct".
```

Figure 4: Generated Harry Potter texts from the trained model

5 Evaluation and Discussion

The CrossEntropy compares two probability distributions - the closer it is to zero, the better. A CrossEntropy of zero means the predicted probability distribution is equal to the target distribution. This means, based on the loss in Figure 3, my model is not doing the best. I believe that the loss would get even lower had I let it run for more than 40 epochs. However, this could lead to overfitting. In addition, I did not *want* the loss to be as low as possible since I wanted the model to generate *new* Harry Potter text, and not just what it had "read" from the books.

It is hard to evaluate how good the generated text is, based on metrics. Human evaluation is the best evaluation method for this kind of prediction. Therefore, I will now evaluate the model from my impressions of the generated texts. The first generated text in Figure 4 was seeded with "Peeves was". Peeves is the Hogwart's poltergeist. Based on Peeves' nature in the books, I expected the model to generate text that displays Peeves causing trouble. However, the generated text is about Peeves being on the Quidditch pitch, looking for the Snitch. This is not something Peeves would typically do unless he was actually looking to ruin the match. I do not think this is what the model meant when outputting this particular bit of text, making it seem as though it has not grasped the idea of who Peeves is. However, it has connected the Snitch, Gryffindor Beaters, and that Slytherin leads, which makes me think it knows what Quidditch is.

The second generated text was seeded with "Professor Dumbledore". The generated text includes a conversation between him and Professor McGonagall. McGonagall is the vice-principal, and Dumbledore is the principal, making a conversation between the two natural. The content of the conversation, however, is questionable. McGonagall is implying that she has been teaching Dumbledore, which I think is unlikely. At the same time, she is saying she is ready to step aside and look forward to taking the Defense Against the Dark Arts teacher position, which I think could make sense since she already is the transfiguration teacher.

The last generated text was seeded with "Harry picked up his". The model correctly predicted "wand" as what Harry picked up, which makes sense as Harry is a wizard. The rest of the text, however, does not make that much sense from a human perspective.

All in all, after playing around with the model, I think it has grasped the overall theme of the Harry Potter universe and books. Its "writing style" is somewhat similar to J.K Rowling's, with, for instance, "thundering up and down, yelling," or "rocketing around the side of the field" being something I would expect to find in one of the Harry Potter books. In addition, it seems to have understood that Quidditch is a sport and that there are different houses in the school.

6 Conclusion and Future Work

This paper has presented a model that will generate new Harry Potter text given a starting text. After human evaluation, it became clear that the model could have been better. Even though it grasped the overall theme and atmosphere of the Harry Potter universe, it makes some errors concerning the bi characters such as Peeves and Professor McGonagall and sometimes does not make much sense. Thus, it is not perfect; however, it is a step toward a fully automatic fantasy author.

The model was only trained for 3 hours, with the standard Transformers (Wolf et al., 2020) loss calculation. Furthermore, the plain pretrained GPT2Tokenizer (Huggingface, 2020a) was used. Future work should look into implementing custom loss and training the pretrained tokenizer on the Harry Potter vocabulary as well. Another idea for future work is to combine the GPT-2 model with Bert to generate even more sensible and Harry Potter-like texts.

References

Jay Alammar. The illustrated gpt-2 (visualizing transformer language models), Aug 2019. URL `https://jalammar.github.io/illustrated-gpt2/`.

Lukas Biewald. Experiment tracking with weights and biases, 2020. URL `https://www.wandb.com/`. Software available from wandb.com.

Bob Deng. Owlreader. `https://github.com/bobdeng/owlreader`, 2014.

Wei Di, Anurag Bhardwaj, and Jianing Wei. *Deep Learning Essentials*. Packt Publishing Ltd, Birmingham, United Kingdom, 2018.

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

Huggingface. Openai gpt2. `https://huggingface.co/transformers/model_doc/gpt2.html#gpt2tokenizer`, 2020a. Accessed: 2021-11-19.

Huggingface. Trainer. `https://huggingface.co/transformers/main_classes/trainer.html`, 2020b. Accessed: 2021-11-19.

IBM Cloud Education, 2020. Recurrent neural networks. URL `https://www.ibm.com/cloud/learn/recurrent-neural-networks`. Accessed on 2021-11-15.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics. URL `https://www.aclweb.org/anthology/2020.emnlp-demos.6`.