Article Generation and GPT-2 Model

In this article, I will try to

1. Explain what is language model.
2. Discuss How to use language model to generate article.
3. Explain what is GPT-2 and how to use it for language modelling.
4. Visualise GPT-2 model's internal state and which input words affect the next word prediction most.

Language Model

Language model is a probability distribution of a sequence of words.

For example, given a language model of English, we can ask what is the probability of seeing "All roads lead to Rome" in English.

Also we could expect the probability of seeing grammatically wrong non-sense sentence such as "jump hamburger I" will be much lower than grammatically correct meaningful sentence such as "I eat hamburger".

Let's pull in some maths notation to help describing language model.
$P(w_1, w_2, ..., w_n)$ means the probability of having the sentence "w_1 w_2 ... w_n".
Noticed that language model is a probability distribution instead of just a probability. Having a probability distribution means we could tell what is the value of P(All, roads, lead, to, Rome) and P(I, eat, hamburger), as we know $P(w_1, w_2, ..., w_n)$ for any $w_{i=1...n}$, for any n.

Some clarification on the notation, whenever you see P(hello, world), where things inside P() are actual words, this mean we know that $w_{i=1...n}$ and n are, that P() is describing a probability. However when you see $P(w_1, w_2, ..., w_n)$, where things inside inside P() are unknown, that P() is describing a probability distribution.
Most of the time when I use the term "probability" and "probability distribution", they are interchangeable unless with further specification.

Sometime, it is more handy if we express $P(w_1, w_2, ..., w_n)$ as P(w, context).
What this mean is we lump w_1 to w_{n-1} , i.e. all words of a sentence except the last one, to a bulky stuff that we called "context". We then ask what is the chance of being in this "context" (seeing previous n-1 words) and seeing the word "w" at the end.
As you can see, those two expressions are describing the same thing.

Using chain rule, we could write P(w, context) as P(w | context) P(context). The reason why we do that is because P(w | context) is the thing we usually want to know.

P(w | context) is a conditional probability distribution, it is telling the chance of seeing a word w given that we know what content is, i.e. knowing what the previous words are.

As P(w | context) is a probability distribution, and could ask P(apple | context), P(orange | context) or any other words in English dictionary, this mean we could use P(w | context) to somehow predict what is the next word if the sentence goes on, or we could use this probability distribution to sample next word, which is the basis of article generation.

So language model is good thing to have, but how can we obtain a language model? Answering this question deserve another piece of article. One approach is counting the number of w_n comes after w_1 to w_{n-1} on a large text corpus, which will build the n-gram language model.

Another approach is to directly learn the language model using a neural network by feeding lots of text. In our case, we are using GPT-2 model to learn the language model.

Article generation Using Language Model

As mentioned in last section, P(w | context) is the basis for article generation.

P(w | context) telling us the probability distribution of all English words given all seen words (as context). For example, for P(w | “I eat”), we would expect a higher probability when w is noun than w is a verb, and we would expect the probability of w being a noun of food like “bread” will have higher chance than w being “book”.

As we have P(w | context), we could use it to predict or generate next word given all previous words. And we could keep doing this, adding one word at a time, until we have a long enough sentence or reaching some “ending word” like full stop.

There are different approaches on how to pick the next word, and we will discuss some of them.

Greedy Approach

One naive approach for picking next word is picking the word with highest probability. Say for P(w | “I eat”) and w being “hamburger” has is highest probability among all words in dictionary, we will pick “hamburger” as the next word, and now we have “I eat hamburger”. We call this the greedy approach for sentence generation.
This approach is very simple and quick, but the main drawback is, for the same set of previous words, we will always generate the same sentence.

Also, when we always pick the highest probability, it is very easy to fall in the case of degenerate repetition, i.e. we keep getting the same chunk of text during sentence

generation. For example

> *I eat hamburger for breakfast. I eat hamburger for breakfast. I eat hamburger for breakfast ...*

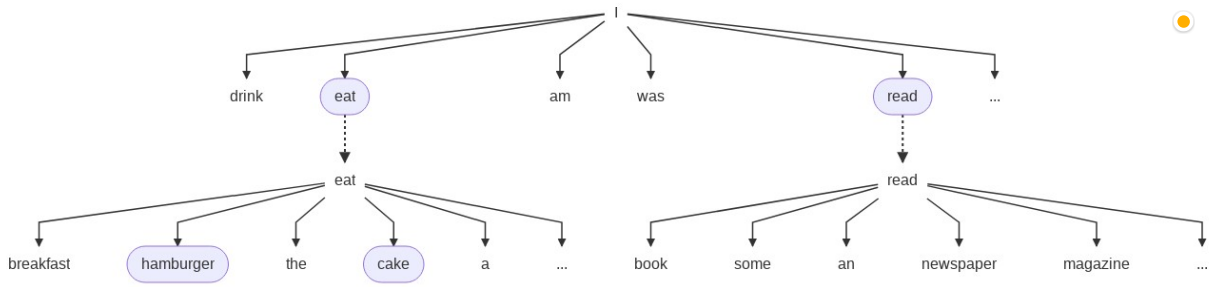
Beam Search

In greedy approach, we pick the highest probability word each time we picking next word, however every we could also generate a lots of sentences first, and pick the sentence with highest probability as well.

Assume there are 20,000 words in dictionary, and we want to generate a sentence with 5 words starting with word "I", the number of all possible sentence that we could generated will be 20000^4 which is one hundred and sixty quadrillion. Clearly we cannot check all those sentences' probability within a reasonable time, even with a fast computer.

Instead of construct all possible sentences, we could, instead, just keep track of top-N partial sentences, so at the end, we just need to check the probability on N sentences. By to so, we hope to search the top-N most likely sentence without trying all combination. This searching is called Beam search, and N is the Beam width.

Following figure illustrate a case of generating a sentence with 3 words, starting with "I" with N = 2, which mean we only keep track of top-2 partial sentences.



For this case, we first check P(w | "I"), among all the possible words, the language model tell use, "eat" and "read" is the most probable words that come next, hence in the next step, we will only consider P(w | "I eat") and P(w | "I read") and ignore other possibility like sentence start with "I drink".

In the next step, we repeat the same procedure, findings most probable 2 words that come after "I eat" or "I read". Among all those sentences that start with "I eat" and "I read", P("hamburger" | "I eat") and P("cake" | "I eat") have highest 2 probabilities, hence we will only expand the search with sentence prefix "I eat hamburger" and "I eat cake". You can see, the whole "I read" branch has died out.

We keep repeating the expand and pick best-N procedure until we have a sentence with desire length. Finally we just need to report the sentence with highest probability.

You may already notice that when beam width is reduce to 1, Beam search will become Greedy approach; when beam width equal to the size of dictionary, beam search become exhaustive search. Beam search give us a way to choose between sentence quality and speed.

With beam width larger than 1, Beam search tends to generate more promising sentence, however greedy approach's issues remain. Same sentence prefix will lead to same sentence and it is easy to result in degenerate repetition when the beam width is not large enough.

Pure Sampling

Issues of Beam search and greedy approach are due to the fact that we are picking the most probable choice during the sentence generation. Instead of picking the most probable word from P(w | context), we could sample a word with P(w | context).

For example, with a sentence start with "I", we can sample a word according to P(w | "I"), as the sampling is random, even P("eat" | "I") > P("read" | "I"), we could still sample the word "read". Using sampling, basically we will have very high chance to get a new sentence in each generation.

Sentence generated from pure sampling will be free from degenerate repetition, but it tends to result in gibberish.

There are 3 common way to try to improve pure sampling.

Top-k Sampling and Sampling with Temperature

First is Top-k sampling. Instead of sample from full P(w | context), we only sample from top K words according to P(w | context).

Second is sampling with temperature. What it mean is we reshape the P(w | context) with a temperature factor **t**. **t** is between 0 and 1.

When we are using neural network to estimate a language model. Instead of probability values (which are in the range of 0 to 1), we are indeed getting real number that could be in any range, those number we get are called logits, and we can convert logits to probability value using softmax function.

Temperature **t** is taking part in the step of applying softmax function to get the probabilities so as to reshape the resultant P(w | context) by dividing each logit value by **t** before apply softmax function. As **t** is between 0 and 1, dividing it will amplify the logit value, this result in making more probable word more probable, and less probable word even less probable.

Top-k sampling and sampling with temperature tends so be applied together.

Nucleus Sampling

When using Top-k sampling, we need to decide which k to use, and the best k tends to be varied among different context.
The idea of Top-k sampling is to ignore words are that very unlikely to be next word according to P(w | context), we can achieve this goal in another way. Instead of focusing on Top-k words in sampling, we ignore words which sum of their probabilities are less then certain threshold, and we only sample from the remaining words.

This approach is called nucleus sampling, according to The Curious Case of Neural Text Degeneration, the original paper that propose nucleus sampling, we should choose p = 0.95, which imply threshold value is 1-p = 0.05. Solely doing nucleus sampling with p = 0.95, we could generate text that are statistically most similar to human written text.

It is recommended to take a look on the nucleus sampling paper, it gives lots of comparison among text generated using different approach (beam search, top-k sampling, nucleus sampling, etc) and the human written text, measured using different metrics.

Introduction to GPT-2 Model

We have discussed how to generate text using language model. We now continue to discuss how to get a language model.

As we mentioned in the being, we will use the neural network called GPT-2 model from OpenAI to estimate the language model.

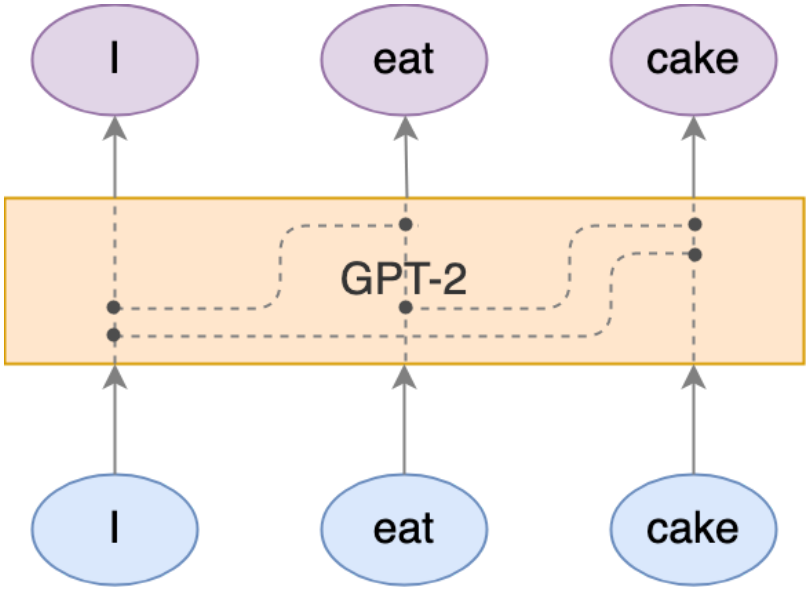
GPT-2 is a Transformer-based model for trained for language modelling, which could easily fine tune to use on other NLP task such as text generation, summarisation, question answering, translation, sentiment analysis, etc.

Again, discussing GPT-2 model deserves a separate article, here I will only focus on few main concepts. For the details, I highly recommend you to read two awesome articles from Jay Alammar on Transformer and GPT-2.

I will explain how GPT-2 model work by building it piece by piece.

Input and output

First, let’s describe the input and output of GPT-2 model.



Given words in its embedding form, GPT-2 could be thought as a transformer to transform the input word embedding vector (blue ellipses) to the output word embedding (purple ellipses). This transformation will not change the dimension of the word embedding (although it could). Output word embedding is also known as the hidden state.

During the transformation, previous words' input embedding will affect the result of current word's output embedding, but not the other way round. In our example, the output embedding of "cake" will depend on the input embedding of "I", "eat" and "cake", but output embedding of "I" will only depend on input embedding of "I".

Due to this, we could also think that the output embedding of the last input word is somehow capturing the essence of the whole input sentence.

To get the language model, we could have a matrix $\mathbf{W_{LM}}$ which has number of column equal to dimension of output embedding, and has number of row equals to the dictionary size; the bias vector $\mathbf{b_{LM}}$ with its dimension being the dictionary size.
We can then compute the logit of each word in dictionary by multiplying $\mathbf{W_{LM}}$ with the output embedding of the last word and add $\mathbf{b_{LM}}$. To convert those logits to probabilities, we just need to apply the softmax function, and its result could be interpreted as P(w | context).

Inside GPT-2 Model

Now, we till discuss how output word embeddings are computed from input word embeddings.

Input word embeddings are vectors, the first steps of the transformation is to create even more vectors from those input word embeddings. To be precise, 3 vectors, namely the **key vector**, **query vector** and **value vector** will be created base on each input word embedding.

Way to produce these vectors are simple, We just need 3 matrices W_{key}, W_{query} and W_{value} By multiplying the input word embedding with these 3 matrices, we will get the corresponding key, query and value vector of corresponding the input word. W_{key}, W_{query} and W_{value} are parts of the parameters of the GPT-2 model.

Let I_{input} be the input word embedding of "I", we have

$$I_{key} = W_{key}\ I_{input},\ I_{query} = W_{query}\ I_{input},\ I_{value} = W_{value}\ I_{input}$$

Noticed that we will use the same W_{key}, W_{query} and W_{value} to compute key, query and value vectors for all other words.

After we know how to compute key, query and value vectors for each input word, it is time to discuss how to use these vectors to compute the output word embedding.

As mentioned in previous section, current word's output embedding will depends on current word's input embedding as well as all the previous words' input embedding. Indeed, the output embedding of a current word is the weighted sum of current word and all its previous words' value vectors. This also explains why value vectors are calling value vectors.

Let eat_{output} be the output embedding of "eat", its value is computed by

$$eat_{output} = {}^{I}A_{eat}\ I_{value} + {}^{eat}A_{eat}\ eat_{value}$$

Here ${}^{I}A_{eat}$ and ${}^{eat}A_{eat}$ are the attention values, they could be interpreted as how much attention should "eat" pay on "I" and "eat" itself when computing its output embedding. To avoid blow up and shrink the output embedding, attentions values need to sum to 1. This imply for first word, its output embedding will be equal to its value vector, for example, I_{output} is equal to I_{value}.

Each attention value ${}^{x}A_{y}$ is computed by the taking dot product between key vector of x and query vector of y, scale down the dot product with square root of dimension of the key vector, finally taking the softmax to ensure related attention values are summing to 1.

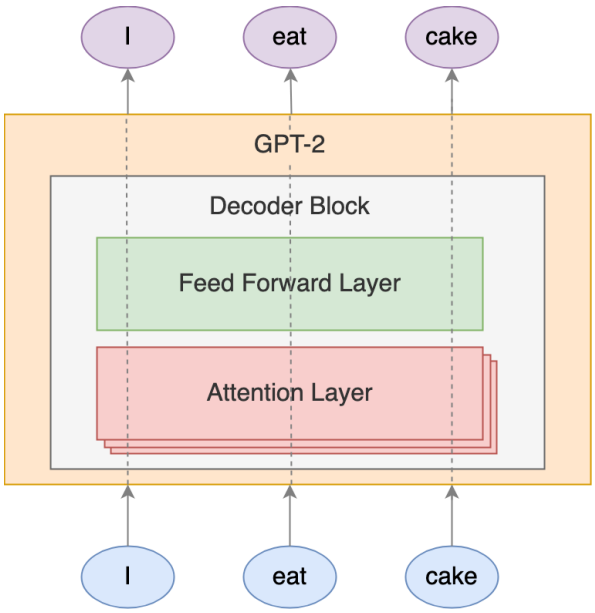
${}^{x}A_{y}$ = softmax($x_{key}^{T}\ y_{query}$ / sqrt(k)), where k is the dimension of key vector.

To recap, we should now know how output embedding are computed as weighted sum of value vectors of the current and previous words. The weights used in the sum are called attention v1alue, which is a value between two words, and is computed by taking dot product of key vector of one word and query vector of another word. As weights should sum to 1, we will also taking the softmax on the dot product.

Structure Replication

Indeed, what we discuss so far are just a layer called **Attention Layer** in GPT-2, but this layer cover most of the details, the rest of the GPT-2 model structure is just

replication of the Attention layer.



Let's continue our GPT-2 model construction journey. GPT-2 is not just using one attention layer but multiple of them, this is the so-called multi-head attention.

Those attention layers are run in parallel, they are not depending on each other, and they don't share weights, i.e. there will be different set of $\mathbf{W_{key}}$, $\mathbf{W_{query}}$ and $\mathbf{W_{value}}$ for each attention layer.

As we have multiple attention layers, we will have multiple output word embeddings for each word. To combine all those output word embeddings into one, we first concatenate all the output word embeddings from different attention layers, then multiply the concatenated matrix $\mathbf{W_{project}}$ to make the output word embedding having the same dimension as the input word embedding.

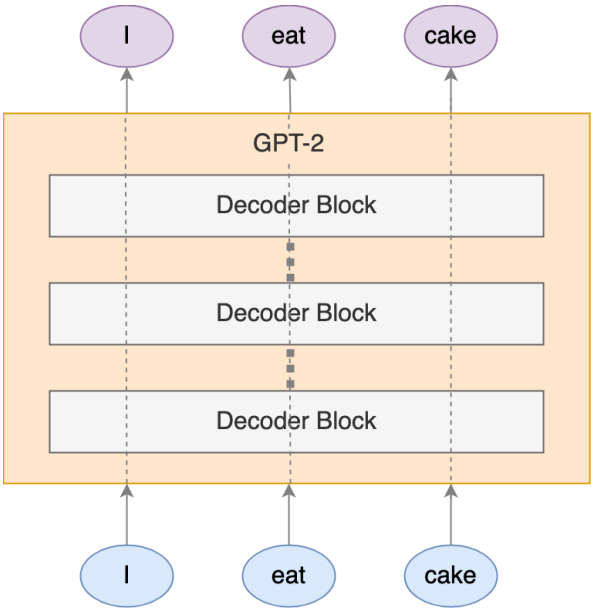
Actually the output word embeddings we got so far is not the final one, the output word embeddings will further go through a feed forward layer, and transform to be the actual output word embeddings.

These parallel running attention layers together with the feed forward layer are grouped to a block called the **Decoder Block**[1].

GPT-2 include not just one Decoder block, but a chain of them. We choose the input word embedding and output word embedding having the same dimensionality so that we could chain up Decoder blocks.

These Decoder blocks has exactly the same structure, but they don't share weight.

GPT-2 model have different size, and they are different in the embedding dimensionality, key, query, value vector's dimensionality, number of Attention layer in each Decoder block, and number of Decoder block in the model.



Omitted Details

Following are some detail I choose to ignore but worth notice, reader can take this list as a pointer to google more about them.

1. GPT-2 is using Byte pair encoding when tokenizing the input string. And one token is not necessary corresponding to one word. And GPT-2 is work in terms of tokens instead of words.
2. Positional embeddings are added to the input embeddings of the first Decoder block, so as to encode the word order information in the word embedding.
3. All residual addition and normalization layer are omitted.

Training of GPT-2 Model

After knowing how GPT-2 work and how to use GPT-2 to estimate language model (by converting last word's output embedding to logits using $\mathbf{W}_{\mathbf{LM}}$ and $\mathbf{b}_{\mathbf{LM}}$, then to probabilities), we can briefly discuss how to train GPT-2 model.

The training of GPT-2 model is indeed doing language model estimation.
Given an input string, such as "I eat cake", GPT-2 can estimate P(eat | "I") and P(cake | "I eat").

For this input string, in training, we will assume
P(eat | "I") = 1, P(w != eat | "I") = 0
P(cake | "I eat") = 1, P(w != cake | "I eat") = 0

Now we have estimated probability distributions and the target probability distributions, we can then compute the cross entropy loss, and use this loss to update

the weights.

As you can see, to train the GPT-2 model, what we need to do it just feeding it with large amount of text.

Try out GPT-2

To quickly try GPT-2 on article generation. We could use Huggingface Transformers. It is a python library for developer to quickly using pre-trained transformers based NLP models. Of course, GPT-2 is supported. It also support both PyTorch and TensorFlow as the underlying deep learning framework.

To fine tune a pre-trained model, we could use the example/run_langauge_modeling.py. What we need is just a text file contain the training text and a text file contain the text for evaluation.

Example usage of run_language_modeling.py for fine-tuning:

```
python run_language_modeling.py \
    --output_dir=output \          # The trained model will be store at ./output
    --model_type=gpt2 \            # Tell huggingface transformers we want to train gp
    --model_name_or_path=gpt2 \    # This will use the pre-trained gpt2 samll model
    --do_train \
    --train_data_file=$TRAIN_FILE \
    --do_eval \
    --eval_data_file=$TEST_FILE \
    --per_gpu_train_batch_size=1   # For GPU training only, you may increase it if you
```

Huggingface transforms have lots of built-in function, text generation is one of them.

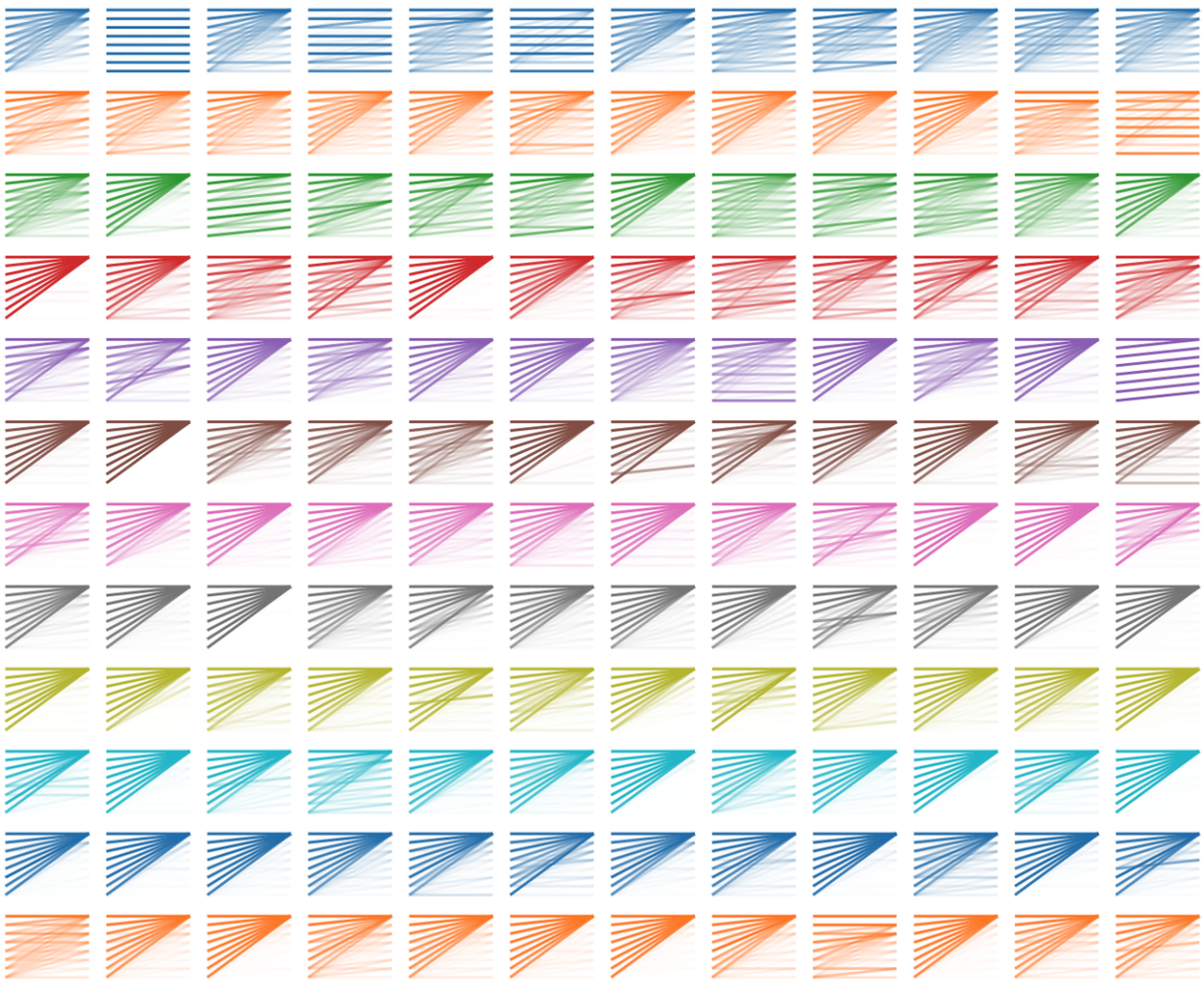
Following is a code snippet to do text generation using pre-trained GPT-2 model.

```
from transformers import (
    GPT2LMHeadModel,
    GPT2Tokenizer,
)

tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
model = GPT2LMHeadModel.from_pretrained("gpt2")

sentence_prefix = "I eat"

input_ids = tokenizer.encode(
    sentence_prefix,
    add_special_tokens=False,
    return_tensors="pt",
    add_space_before_punct_symbol=True
)

output_ids = model.generate(
    input_ids=input_ids,
```

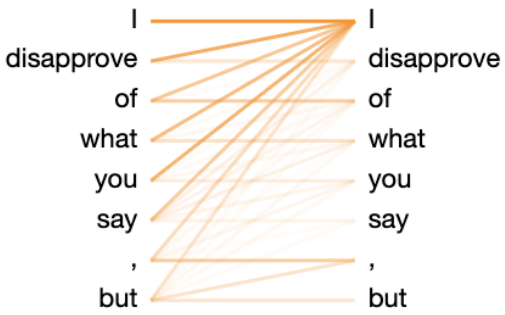
```
    do_sample=True,
    max_length=20,  # desired output sentence length
    pad_token_id=model.config.eos_token_id,
)[0].tolist()

generated_text = tokenizer.decode(
    output_ids,
    clean_up_tokenization_spaces=True)

print(generated_text)
```

Attention Visualisation

Thanks to https://github.com/jessevig/bertviz, we could try to peek how GPT-2 work by visualising the attention values.



The figure above is a visualisation of attentions values on each decoder block (from top to bottom of the grid, with first row being the first block), and each attention head (from left to right) of GPT-2 small model taking “I disapprove of what you say, but” as input.



On the left is the zoom-in of the 2^{nd} block’s 6^{th} attention head’s result.

Words on the left are the output, and words on the right are the input. The opacity of the line indicate how much attention did the output word pay to the input words.

One interesting fact we could see here is, most of the time, first word is being paid the most attention on. This general pattern remain even if we use other input sentence.

Word Importance Visualisation

Purely looking at the attention values seems bring us not much clue on how input sentence is affect GPT-2 model to pick its next word.
One of the reason could be, as there still has $\mathbf{W_{project}}$ feed forward layer to transform attention layer’s output, it is hard to imagine how those attention are being utilise on predicting the next word.

We are particularly interested in how the input sentence is affecting the probability distribution of next word, or more precisely, we want to know which word in the input sentence will affect the next word probability distribution the most.

Measure word importance through input perturbation

In Towards a Deep and Unified Understanding of Deep Neural Models in NLP, the author propose a way to answer this question. They also provide the code which we could use to analyse the GPT-2 model as well

The idea mentioned in this paper for measuring the importance of input word in this paper is simple.

The idea is assign a value σ_i to each input word, σ_i is initially randomly to value between 0 and 1.

Later on we will generate some noise vector with the size of input word embedding. This noise vector will be added to the input word embedding with the weight specified in σ_i. So σ_i are telling use how much noise is added to the corresponding input word.

With the original and perturbed input word embeddings, we feed both of them to our GPT-2 model and get 2 set of logit from the last output embeddings.

We then measure the difference (using L^2 norm) between this two logits. This difference tell use how severe the perturbation is affecting the resultant logits we use to construct the language model.
We then optimise σ_i on the minimising the difference between two logits.

We keep repeating generate new noise vector and added them to the original input word embedding using **updated** σ_i, compute the difference between the resultant logits, and use this difference to guide the update of the σ_i.
During the iteration, we will keep track of the best σ_i which lead to the smallest difference in the resultant logits, and report it as the result after we reaching the max number of iteration.

The reported σ_i are telling us how much noise the corresponding input word could withstand and not lead to significant change in the resultant logits.

If a word is importance to the resultant logits, we would expect small perturbation on that word's input embedding will lead to significant change in the logits, hence reported σ_i are inversely proportional to the importance of the words.

The smaller the the reported σ_i, the more important the corresponding input word is.

Code Snippet

Following is the code snippet for visualising the word importance. Interpreter.py could be found here

```
import torch
from transformers import GPT2Tokenizer, GPT2LMHeadModel
from Interpreter import Interpreter

def Phi(x):
    global model
    result = model(inputs_embeds=x)[0]
    return result[-1,:] # return the logit of last word

model_path = "gpt2"
model = GPT2LMHeadModel.from_pretrained(model_path, output_attentions=True)
tokenizer = GPT2Tokenizer.from_pretrained(model_path)

input_embedding_weight_std = (
    model.get_input_embeddings().weight.view(1,-1)
    .std().item()
)

text = "I disapprove of what you say , but"
inputs = tokenizer.encode_plus(text, return_tensors='pt',
```

```
                                add_special_tokens=True,
                                add_space_before_punct_symbol=True)
input_ids = inputs['input_ids']

with torch.no_grad():
    x = model.get_input_embeddings()(input_ids).squeeze()

interpreter = Interpreter(x=x, Phi=Phi,
                          scale=10*input_embedding_weight_std,
                          words=text.split(' ')).to(model.device)

# This will take sometime.
interpreter.optimize(iteration=1000, lr=0.01, show_progress=True)
interpreter.get_sigma()
interpreter.visualize()
```

Following are the reported σ_i and their visualisation.
The smaller the value, the darker the color.

```
array([0.8752377, 1.2462736, 1.3040292, 0.55643  , 1.3775877, 1.2515365,
       1.2249271, 0.311358 ], dtype=float32)
```

From the figure, we can now know P(w | “I disapprove of what you sat, but”) will be affected by the word “but” most and followed by “what”, then followed by “I”.

Conclusion

In this article, we have discussed what language model is and how use it to do article generation with different approach to get human-written like text.

We also have a brief introduction to GPT-2 model and some of its internal working.
We have also seen how to use Huggingface transforms for applying GPT-2 model in text generation.

We have visualised the attention values in GPT-2 model, as well as using
input perturbation approach to see which word(s) in the input sentence would affect the next word prediction the most.

Footnotes

1. The actual structure of Decoder Block is consist of one attention layer only, what

we describe in this article as attention layer should be called attention head. One attention layer include multiple attention heads AND the $\mathbf{W_{project}}$ for combining the attention heads' output.