Inside GPT — I : Understanding the text generation

A simple explanation of the model behind ChatGPT

Fatih Demirci
.
Follow
Published in

Towards Data Science

.
11 min read
.
Aug 21, 2023
728
7

Generative AI in 19th century (created through midjourney)

Regularly engaging with colleagues across diverse domains, I enjoy the challenge of conveying machine learning concepts to people who have little to no background in data science. Here, I attempt to explain how GPT is wired in simple terms, only this time in written form.

Behind ChatGPT's popular magic, there is an unpopular logic. You write a prompt to ChatGPT and it generates text and whether it is accurate, it resembles human answers. How is it able to understand your prompt and generate coherent and comprehensible answers?

Transformer Neural Networks. The architecture designed to process unstructured data in vast amounts, in our case, text. When we say architecture, what we mean is essentially a series of mathematical operations that were made in several layers in parallel. Through this system of equations, several innovations were introduced that helped us overcome the long-existing challenges of text generation. The challenges that we were struggling to solve up until 5 years ago.

If GPT has already been here for 5 years (indeed GPT paper was published in 2018), isn't GPT old news? Why has it become immensely popular recently? What is the difference between GPT 1, 2, 3, 3.5 (ChatGPT) and 4?

All GPT versions were built on the same architecture. However each following model contained more parameters and trained using larger text datasets. There were obviously other novelties introduced by the later GPT releases especially in the training processes like reinforcement learning through human feedback which we will explain in the 3rd part of this blog series.

Vectors, matrices, tensors. All these fancy words are essentially units that contain chunks of numbers. Those numbers go through a

series of mathematical operations(mostly multiplication and summation) until we reach optimal output values, which are the probabilities of the possible outcomes.

Output values? In this sense, it is the text generated by the language model, right? Yes. Then, what are the input values? Is it my prompt? Yes, but not entirely. So what else is behind?

Before going on to the different text decoding strategies, which will be the topic of the following blog post, it is useful to remove the ambiguity. Let's go back to fundamental question that we asked at the start. How does it understand human language?

Generative Pre-trained Transformers. Three words that GPT abbreviation stands for. We touched the Transformer part above that it represents the architecture where heavy calculations are made. But what do we calculate exactly? Where do you even get the numbers? It is a language model and all you do is to input some text. How can you calculate text?

Data is agnostic. All data is same whether in the form of text, sound or image.[1]

Tokens. We split the text into small chunks (tokens) and assign an unique number to each one of them(token ID). Models don't know words, images or audio recordings. They learn to represent them in huge series of numbers (parameters) that serves us as a tool to illustrate the characteristics of things in numerical forms. Tokens

are the language units that convey meaning and token IDs are the unique numbers that encode tokens.

Obviously, how we tokenise the language can vary. Tokenisation can involve splitting texts into sentences, words, parts of words(sub-words), or even individual characters.

Let's consider a scenario where we have 50,000 tokens in our language corpus(similar to GPT-2 which has 50,257). How do we represent those units after tokenisation?

```
Sentence: "students celebrate the graduation with a big party"
Token labels: ['[CLS]', 'students', 'celebrate', 'the', 'graduation',
'with', 'a', 'big', 'party', '[SEP]']
Token IDs: tensor([[ 101, 2493, 8439, 1996, 7665, 2007, 1037, 2502, 2283,
102]])
```

Above is an example sentence tokenised into words. Tokenisation approaches can differ in their implementation. What's important for us to understand right now is that we acquire numerical representations of language units(tokens) through their corresponding token IDs. So, now that we have these token IDs, can we simply input them directly into the model where calculations take place?

Cardinality matters in math. 101 and 2493 as token representation will matter to model. Because remember, all we are doing is mainly multiplications and summations of big chunks of numbers. So multiplying a number with either with 101 or with 2493 will matter. Then, how can we make sure a token that is represented with

number 101 is not less important than 2493, just because we happen to tokenise it arbitrarily so? How can we encode the words without causing a fictitious ordering?

One-hot encoding. Sparse mapping of tokens. One-hot encoding is the technique where we project each token as a binary vector. That means only one single element in the vector is 1 (“hot”) and the rest is 0 (“cold”).

token	world	W_a	W_about	W_after	W_all	W_also	...	...	W_zest	W_zucchin	W_zulu	W_zumba	W_zurich
0	a	1	0	0	0	0	...	...	0	0	0	0	0
1	about	0	1	0	0	0	...	...	0	0	0	0	0
2	after	0	0	1	0	0	...	...	0	0	0	0	0
3	all	0	0	0	1	0	...	...	0	0	0	0	0
4	also	0	0	0	0	1	...	...	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...	...	...	...
49996	zest	0	0	0	0	0	...	...	1	0	0	0	0
49997	zucchini	0	0	0	0	0	...	...	0	1	0	0	0
49998	zulu	0	0	0	0	0	...	...	0	0	1	0	0
49999	zumba	0	0	0	0	0	...	...	0	0	0	1	0
50000	zurich	0	0	0	0	0	...	...	0	0	0	0	1

Image by the author: one-hot encoding vector example

The tokens are represented with a vector which has length of total token in our corpus. In simpler terms, if we have 50k tokens in our language, every token is represented by a vector 50k in which only one element is 1 and the rest is 0. Since every vector in this projection contains only one non-zero element, it is named as sparse representation. However, as you might think this approach is very inefficient. Yes, we manage to remove the artificial cardinality between the token ids but we can’t extrapolate any information about the semantics of the words. We can’t understand whether the word “party” refers to a celebration or to a political organisation by using sparse vectors. Besides, representing every token with a vector of size 50k will mean, in total of 50k vector of length 50k. This is

very inefficient in terms of required memory and computation. Fortunately we have better solutions.

Embeddings. Dense representation of tokens. Tokenised units pass through an embedding layer where each token is transformed into continuous vector representation of a fixed size. For example in the case of GPT 3, each token in is represented by a vector of 768 numbers. These numbers are assigned randomly which then are being learned by the model after seeing lots of data(training).

```
Token Label: "party"
Token : 2283
Embedding Vector Length: 768
Embedding Tensor Shape: ([1, 10, 768])

Embedding vector:

tensor([ 2.9950e-01, -2.3271e-01,  3.1800e-01, -1.2017e-01, -3.0701e-01,
        -6.1967e-01,  2.7525e-01,  3.4051e-01, -8.3757e-01, -1.2975e-02,
        -2.0752e-01, -2.5624e-01,  3.5545e-01,  2.1002e-01,  2.7588e-02,
        -1.2303e-01,  5.9052e-01, -1.1794e-01,  4.2682e-02,  7.9062e-01,
         2.2610e-01,  9.2405e-02, -3.2584e-01,  7.4268e-01,  4.1670e-01,
        -7.9906e-02,  3.6215e-01,  4.6919e-01,  7.8014e-02, -6.4713e-01,
         4.9873e-02, -8.9567e-02, -7.7649e-02,  3.1117e-01, -6.7861e-02,
        -9.7275e-01,  9.4126e-02,  4.4848e-01,  1.5413e-01,  3.5430e-01,
         3.6865e-02, -7.5635e-01,  5.5526e-01,  1.8341e-02,  1.3527e-01,
        -6.6653e-01,  9.7280e-01, -6.6816e-02,  1.0383e-01,  3.9125e-02,
        -2.2133e-01,  1.5785e-01, -1.8400e-01,  3.4476e-01,  1.6725e-01,
        -2.6855e-01, -6.8380e-01, -1.8720e-01, -3.5997e-01, -1.5782e-01,
         3.5001e-01,  2.4083e-01, -4.4515e-01, -7.2435e-01, -2.5413e-01,
         2.3536e-01,  2.8430e-01,  5.7878e-01, -7.4840e-01,  1.5779e-01,
        -1.7003e-01,  3.9774e-01, -1.5828e-01, -5.0969e-01, -4.7879e-01,
        -1.6672e-01,  7.3282e-01, -1.2093e-01,  6.9689e-02, -3.1715e-01,
        -7.4038e-02,  2.9851e-01,  5.7611e-01,  1.0658e+00, -1.9357e-01,
         1.3133e-01,  1.0120e-01, -5.2478e-01,  1.5248e-01,  6.2976e-01,
        -4.5310e-01,  2.9950e-01, -5.6907e-02, -2.2957e-01, -1.7587e-02,
        -1.9266e-01,  2.8820e-02,  3.9966e-03,  2.0535e-01,  3.6137e-01,
         1.7169e-01,  1.0535e-01,  1.4280e-01,  8.4879e-01, -9.0673e-01,
                                            …
                                            …
                                            …                          ])
```

Above is the embedding vector example of the word "party".

Now we have 50,000x786 size of vectors which is compare to 50,000x50,000 one-hot encoding is significantly more efficient.

Embedding vectors will be the inputs to the model. Thanks to dense numerical representations we will able to capture the semantics of words, the embedding vectors of tokens that are similar will be closer to each other.

How can you measure the similarity of two language unit in context? There are several functions that can measure the similarity between the two vectors of same size. Let's explain it with an example.

Consider a simple example where we have the embedding vectors of tokens "cat" , "dog", "car" and "banana". For simplification let's use an embedding size of 4. That means there will be four learned numbers to represent the each token.

```
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity

# Example word embeddings for "cat" , "dog", "car" and "banana"
embedding_cat = np.array([0.5, 0.3, -0.1, 0.9])
embedding_dog = np.array([0.6, 0.4, -0.2, 0.8])
embedding_car = np.array([0.5, 0.3, -0.1, 0.9])
embedding_banana = np.array([0.1, -0.8, 0.2, 0.4])
```

Using the vectors above lets calculate the similarity scores using the cosine similarity. Human logic would find the word "dog" and "cat" more similar to each other than the words banana a car. Can we expect math to simulate our logic?

```
# Calculate cosine similarity
similarity = cosine_similarity([embedding_cat], [embedding_dog])[0][0]

print(f"Cosine Similarity between 'cat' and 'dog': {similarity:.4f}")

# Calculate cosine similarity
similarity_2 = cosine_similarity([embedding_car],
[embedding_banana])[0][0]

print(f"Cosine Similarity between 'car' and 'banana': {similarity:.4f}")
"Cosine Similarity between 'cat' and 'dog': 0.9832"
"Cosine Similarity between 'car' and 'banana': 0.1511"
```

We can see that the words “cat” and “dog” have very high similarity score whereas the words “car” and “banana” have very low. Now imagine embedding vectors of length 768 instead of 4 for each 50000 token in our language corpus. That’s how we are able find the words that are related to each other.

Now, let’s have a look at the two sentences below which have higher semantic complexity.

```
"students celebrate the graduation with a big party"

"deputy leader is highly respected in the party"
```

The word “party” from the first and second sentence conveys different meanings. How are large language models capable of mapping out the difference between the “party” as a political organisation and “party” as celebrating social event?

Can we distinguish the different meanings of same token by relying on the token embeddings? The truth is, although embeddings

provide us a range of advantages, they are not adequate to disentangle the entire complexity of semantic challenges of human language.

Self-attention. The solution was again offered by transformer neural networks. We generate new set of weights(another name for parameters) that are namely query, key and value matrices. Those weights learn to represent the embedding vectors of tokens as a new set of embeddings. How? Simply by taking the weighted average of the original embeddings. Each token “attends” to every other token(including to itself) in the input sentence and calculates set of attention weights or in other word the new so called “*contextual embeddings*”.

All it does really is to map the importance of the words in the input sentence by assigning new set of numbers(attention weights) that are calculated using the token embeddings.

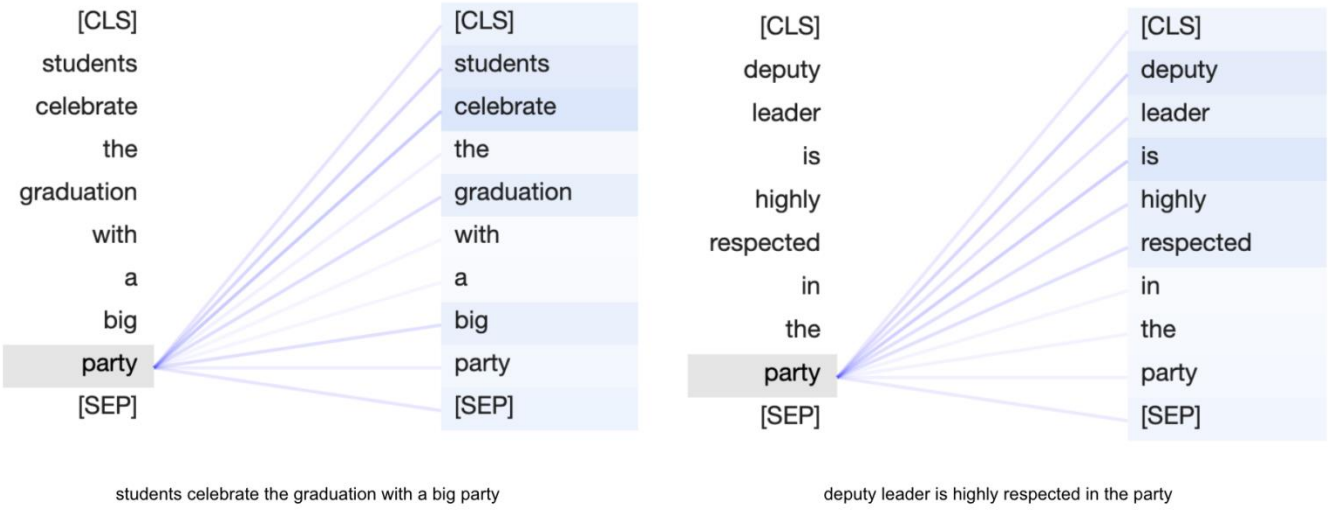

Image by the author: Attention weights of a token in different contexts (BertViz attention-head view)

Above visualisation demonstrates the “attention” of the token “party” to the rest of the tokens in two sentences. The boldness of the connection signals to the importance or the relevance of the tokens. Attention and “attending” are the terms that refer to a new series of numbers(attention parameters) and their magnitude, that we use to represent the importance of words numerically. In the first sentence the word “party” attends to the word “celebrate” the most, whereas in the second sentence the word “deputy” has the highest attention. That’s how the model is able to incorporate the context by examining surrounding words.

As we mentioned in the attention mechanism we derive new set of weight matrices, namely: Query, Key and Value (simply q,k,v). They are cascading matrices of same size(usually smaller than the embedding vectors) that are introduced to the architecture to capture complexity in the language units. Attention parameters are learned in order to demystify the relationship between the words, pairs of words, pairs of pairs of words and pairs of pairs of pairs of words and so on. Below is the visualisation of the query, key and value matrices in finding the most relevant word.

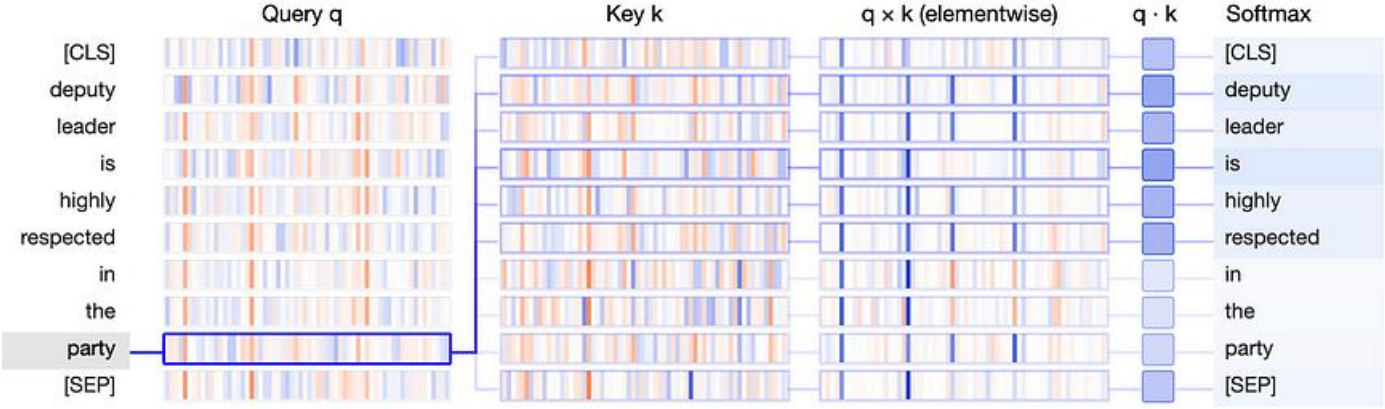

Image by the author: illustration of query key and value matrices and their final probabilities(BertViz q,k,v view)

The visualisation illustrates the q and k vectors as vertical bands, where the boldness of each band reflects its magnitude. The connections between tokens signify the weights determined by attention, indicating that the q vector for “party” aligns most significantly with the k vector for “is”, “deputy” and “respected”.

To make the attention mechanism and the concepts of q, k and v less abstract, imagine that you went to a party and heard an amazing song that you fell in love with. After the party you are dying to find the song and listen again but you only remember barely 5 words from the lyrics and a part of the song melody(query). To find the song, you decide to go through the party playlist(keys) and listen(similarity function) all the songs in the list that was played at the party. When you finally recognise the song, you note the name of the song(value).

One last important trick that transformers introduced is to add the positional encodings to the vector embeddings. Simply because we would like to capture the position information of the word. It enhances our chances to predict the next token more accurately towards to the true sentence context. It is essential information because often swapping the words changes the context entirely. For instance, the sentences *“Tim chased clouds all his life”* vs *“clouds chased Tim all his life”* are absolutely different in essence.

All the mathematical tricks that we explored at a basic level so far, have the objective of predicting the next token, given the sequence of input tokens. Indeed, GPT is trained on one simple task which is the

text generation or in other words the next token prediction. At its core of the matter, we measure the probability of a token, given the sequence of tokens appeared before it.

You might wonder how do models learn the optimal numbers from randomly assigned numbers. It is a topic for another blog post probably however that is actually fundamental on understanding. Besides, it is a great sign that you are already questioning the basics. To remove unclarity, we use an optimisation algorithm that adjusts the parameters based on a metric that is called loss function. This metric is calculated by comparing the predicted values with the actual values. The model tracks the changes of the metric and depending on how small or large the value of loss, it tunes the numbers. This process is done until the loss can not be smaller given the rules we set in the algorithm that we call hyperparameters. An example hyperparameter can be, how frequently we want to calculate the loss and tune the weights. This is the rudimentary idea behind learning.

I hope in this short post, I was able to clear the picture at least a little bit. The second part of this blog series will focus on decoding strategies namely on why your prompt matters. The third and the last part will be dedicated to key factor on ChatGPT's success which is the reinforcement learning through human feedback. Many thanks for the read. Until next time.

References:

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention Is All You Need,” in Advances in Neural Information Processing Systems 30 (NIPS 2017), 2017.

J. Vig, “A Multiscale Visualization of Attention in the Transformer Model,” In Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations, pp. 37–42, Florence, Italy, Association for Computational Linguistics, 2019.

L. Tunstall, L. von Werra, and T. Wolf, “Natural Language Processing with Transformers, Revised Edition,” O’Reilly Media, Inc., Released May 2022, ISBN: 9781098136796.

1 -Lazy Programmer Blog