Key ideas of Large Language Models

Florian Martel

Abstract: *With the release of ChatGPT in 2022 large language models gained broad public attention. On this report we will discuss and review the underlying architecture of ChatGPT - the Transformer model. As well, we will review the performance and the key architecture of GPT3 as a example for a Transformer model.*

Contents

1 The Transformer Model

The Transformer model, introduced by Vaswani *et al.* (2017), was able to make major advancement in the field of natural language processing. Unlike traditional sequence-to-sequence models that use recurrence or convolution, the Transformer is based solely on attention mechanisms.

The key advancement of the Transformer lies in its self-attention-mechanism, allowing it to adjust the given embeddings of words in a sentence by the context of the sentence. Therefore the Transformer is able not just to understand single words, but rather understand words in the context of a sentence.

Due to their high performance across various tasks, such as translation, summarization, and question answering, Transformers have become the foundation for numerous natural language processing models, such as BERT, GPT, and T5.

1.1 Overview

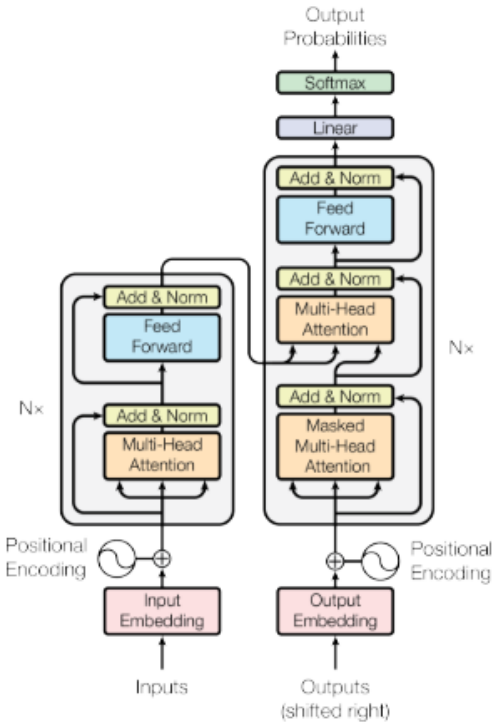

Figure 1: The Transformer - model architecture

Like many other models in natural language processing the Transfomer has a encoder-decoder structure.

Encoder
The Encoder is made up of a stack of 6 identical layers. Each layer consists of two sub-layers. The first sub-layer is a multi-head self-attention mechanism. The second sub-layer is a feed-forward network (multilayer perceptron), which processes each position in the input separately. Each sub-layer uses a residual connection and layer normalization. Residual means the input is added to the output. This is a common approach that helps maintain stability in the learning process as introduced in He *et al.* (2016). All sub-layers in the Encoder produce outputs with a dimension $d_{\text{model}} = 512$.

Decoder
The Decoder also consists of 6 identical layers. Each layer includes the same two sub-layers as the Encoder: multi-head self-attention and feed-forward network. In addition, the Decoder incorporates a third sub-layer that performs multi-head attention over the output of the Encoder stack("encoder-decoder attention"). This enables the model to generate output sequences based on the learned representations from the input. Similar to the Encoder, each sub-layer in the Decoder uses residual connections and layer normalization. To ensure that the model only attends to previous positions during generation, the self-attention sub-layer in the Decoder is modified with masking. This technique prevents positions in the input sequence from attending to previous positions, ensuring that predictions depend only on previously generated outputs.

1.2 Embeddings

Obviously the Transformer Architecture can't process with words. Therefore we need to transform the Inputs into a computable structure. In the case of the Transformer Embeddings we use vectors.

In therory it would work to take the whole alphabet of a language, for example english, and assign a corresponding vector for this word. In the original Transformer in Vaswani *et al.* (2017) this vector has the dimension 512. That would mean, that there are 512 possible "meanings" stored in this vector.

In practice you don't want to take the whole alphabet of a language, because this would take a very large embedding storage. For example there are around 600.000 words in the Oxford English Dictionary. With an embedding dimension of 512 this would result in $N = 600.000 * 512 = 307.200.000$ parameters. This is the reason, why state of the art LLMs use tokenizers like SentencePiece. (Kudo and Richardson (2018))

For example, this is how GPT-3 would tokenize a test sentence:

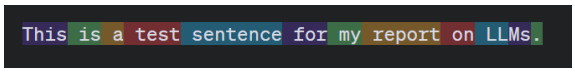

Figure 2: Test tokinization with GPT3 tokenizer

The colored boxes display the tokens. You can see that most words are considered as single tokens, but the word LLMs is made up of two tokens. Each token has a unique id, which maps to the corresponding vector. This Test sentence for example would therefore be translated in this array of unique ids:

$$[1212, 318, 257, 1332, 6827, 329, 616, 989, 319, 27140, 10128, 13]$$

With the usage of tokens the embedding storage rapidly decreases. Using the same tokenizer like GPT-2, GPT3 for example has a vocabulary size of 50.257. (Radford *et al.* (2019))

The embeding vectors are parameters of the transformer. This means, they are chosen randomly in the first place and then will be trained through backpropagation.

1.3 Attention

One token can have multiple meanings. For example the german word "Bank" can reference a bank, a bench or a sandbank. Without an attention mechanism the embedding vector for "Bank" stays the same, although it should differ due to different meanings depending on the context of the sentence where the token is used. This is what the attention process provides: It adjusts the given embedding vector by the context of the sentence in order to contain the real meaning of the token.

An attention function takes three parameters: queries q, keys k, and values v. Each token has its own corresponding query, key, and value. They can be computed by multiplying the embedding vector e by a corresponding weight matrix W:

$$q = W_q * e$$
$$k = W_k * e$$
$$v = W_v * e$$

The weight matrix contains parameters which are learned in the training process. Therefore it is difficult to understand, what the Transformer model really does. The most common interpretation is the following: The query of a token can be seen as query to search

for context. If a query finds a fitting key (we'll see that this happens when the vectors point in the same direction) this means the key-token gives context to the query-token by adding the value from the key-token to the embedding-vector of the query-token.

Scaled Dot-Product Attention
Besides the Scaled Dot-Product Attention there are several possible way to compute Attention, for example the more simple Dot-Product Attention (same procedure, but without scaling) or the Additive Attention introduced in Bahdanau *et al.* (2016). It turns out, that Scaled Dot-Product Attention outperforms Additive and Dot-Product Attention.(Britz *et al.* (2017)) Scaled Dot-Product Attention is computed in 5 Steps:

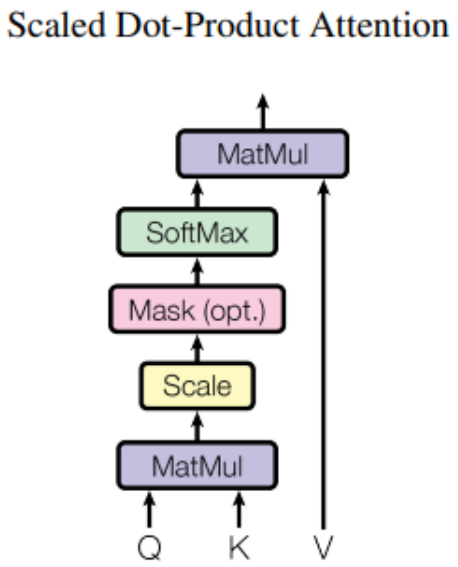

Figure 3: Scaled Dot Product Attention

1. Dot-product of queries and keys
After computing queries and keys as mentioned above, the first step is to perform a dot-product of all query- and keyvectors. To do this, all query- and keyvectors are concatenated to matrix:

$$Q = [q_1, q_2, q_3, ...]$$
$$K = [k_1, k_2, k_3, ...]$$

The result is a matrix QK^T containing all dot-products of query- and keyvectors.

2. Scaling by $\frac{1}{\sqrt{d_k}}$
This step distinguishes the Scaled from the normal Dot-Product Attention and makes it more performant. The reason for this is, that with rising embedding dimension d_k the softmax function (Step 4) is going to return extremely small gradients due to the rising variance of the data. This can be explained as follows:

Assume that the components of the vectors $\mathbf{q}$ and $\mathbf{k}$ are independent random variables. Each of these random variables has a mean of 0 and a variance of 1. Since each q_i and k_i is a random variable with mean 0 and variance 1, the product $q_i k_i$ also has a mean of 0. The variance of $q_i k_i$ is:

$$\text{Var}(q_i k_i) = \text{Var}(q_i) \cdot \text{Var}(k_i) = 1 \cdot 1 = 1$$

When we sum these products to get the dot product, the variances add up. Therefore, the variance of the dot product $\mathbf{q} \cdot \mathbf{k}$ is:

$$\text{Var}\left(\sum_{i=1}^{d} q_i k_i\right) = \sum_{i=1}^{d} \text{Var}(q_i k_i) = \sum_{i=1}^{d} 1 = d$$

Thus, the dot product $\mathbf{q} \cdot \mathbf{k}$ has a mean of 0 and a variance of d.

3. Masking (optional)
During training of Transformer Models, it is necessary to prevent the model from "cheating" by peeking at future tokens. This is achieved by maskinig the future tokens.

In theory masking involves setting specific positions in the attention matrix to a very large negative value (typically $-\infty$) so that when the Softmax function is applied in the next step, these positions will have the vale zero. This would ensure that the model does not consider these masked positions when computing the attention weights. In practice it is not possible to set values to minus infinity. Therefore the values are set to the smallest possible value, which causes a small blurring, because the values computed by Softmax are not exact zero.

4. Softmax
The Softmax function is defined as:

$$\sigma(\mathbf{z_i}) = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

It is used to convert the values of the scaled matrix into probabilities. For z we use all vectors inside the matrix. Softmax normalizes the vector-values so that they lie in the

range (0, 1) and sum to 1. This allows the model to weight the importance of different tokens. The computed values therefore are an indicator of how well the query fits to the key. They are called attention weights.

5. Matrix-Vector Multiplication with values
After computing the attention weights using the softmax function, the next step is to use these weights to combine the value vectors.
Let $V = [v_1, v_2, v_3, \ldots]$ be the matrix of value vectors, where each v_i corresponds to the value vector associated with the key k_i. The resulting context matrix C is then obtained by multiplying the attention weight matrix A with the value matrix V:

$$C = AV$$

This context matrix will then be added to the input embedding matrix in order to change the meaning of the tokens.

Putting all steps together, we end up with the following equation:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

This process forms one Attention Head.

Multi Head Attention
In practice it turned out to be beneficial to use multiple attention heads in one model. This

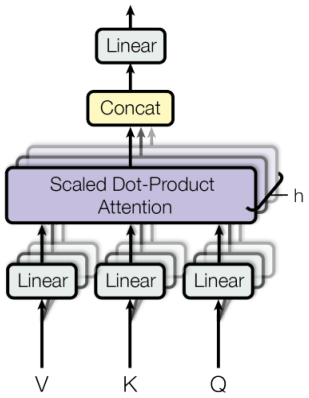

Figure 4: Multi Head Attention

means there are multiple query, key and value matrices allowing the model to attend from different representation subspaces at different positions. The computed context matrices of the attention heads are concatenated at the end to a single multi head context matrix.

2 Analysis of GPT3

In "Language Models are Few Shot learners" by Brown *et al.* (2020) GPT3 was introduced, a large language model focussing on general understanding rather than beeing finetuned n a specific task.

2.1 Overview

Before GPT3 there were other approaches for solving NLP tasks. At first, there were single-layer word vectors (Mikolov *et al.* (2013)) that fed into specific task architectures. Then RNNs came across, that added multiple layers and context but still needed specific architectures for different tasks. (Dai and Le (2015)) The latest trend before GPT3 was using big pre-trained transformer models, which were fine-tuned directly for different tasks. (Radford *et al.* (2018))

This approach has led to progress in challenging tasks like reading comprehension and question answering. However, there is a problem: these models still need a lot of task-specific data for fine-tuning. For each new task, you need thousands to hundreds of thousands of labeled examples. That is why GPT3 uses another approach, instead of finetuning: Few-Shot Learning.

In order to perform well on different tasks without finetuning GPT3 has to develop a broad understanding of different tasks. This is reffered as meta learning and takes a lot of training data. This training data was used to train GPT3.

Dataset	Quantity (tokens)	Weight in training mix	Epochs elapsed when training for 300B tokens
Common Crawl (filtered)	410 billion	60%	0.44
WebText2	19 billion	22%	2.9
Books1	12 billion	8%	1.9
Books2	55 billion	8%	0.43
Wikipedia	3 billion	3%	3.4

Figure 5: Training Data used to train GPT3

Using training data, different versions of GPT3 were trained in order to be able to test, whether performance depends on model size. We will see, that this correlates.

2.2 Few Shot Learning vs. Finetuning

Finetuning
A model is finetuned by giving it a a supervised dataset of the task it should be finetuned on. Processing the examples, the weights are updated through backpropagation. That

Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}
GPT-3 Small	125M	12	768	12	64
GPT-3 Medium	350M	24	1024	16	64
GPT-3 Large	760M	24	1536	16	96
GPT-3 XL	1.3B	24	2048	24	128
GPT-3 2.7B	2.7B	32	2560	32	80
GPT-3 6.7B	6.7B	32	4096	32	128
GPT-3 13B	13.0B	40	5140	40	128
GPT-3 175B or "GPT-3"	175.0B	96	12288	96	128

Figure 6: Different archtitectures of GPT3

means finetuning a model is changing the parameters of the model. Besides the disadvantage of needing a large trining dataset, finetuning also potencially makes the model perform worse on other tasks. As well there ist the potential to exploit spurious features, resulting in an unfair advantage to human performance. The main advantage on the other hand is a strong performance on a specific benchmark. GPT3 is not finetuned for specific tasks.

Figure 7: Finetuning process

Few-Shot learning

Few-Shot Learning does not change the weights of a model. Rather it gives a few examples oft the task at interference to the context oft the model. Few Shot learning turns out

to be performing worse than state-of-the art finetuned models, but you only need a very small amount of supervised data and the model can perform well on diffrent tasks.

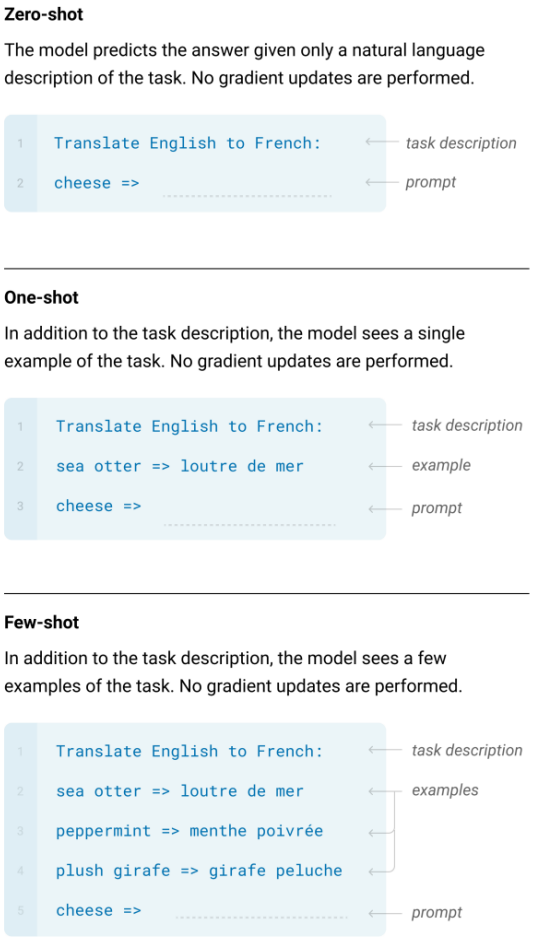

Figure 8: Few shot learning

2.3 Performance On Benchmarks

Putting together different results on different benchmarks it turns out that Few-Shot learning tends to be better than Zero- (no examples given) and One-Shot Learning (Only one example given). As well the performance on tasks tends to increase with the amount of parameters of the model.

The task-performance increases with rising amount of examples, but the growth rate turns out to be decreasing when increasing the rate of examples.

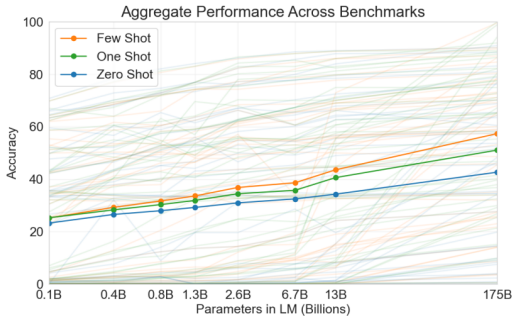

Figure 9: Aggregate performance on benchmarks

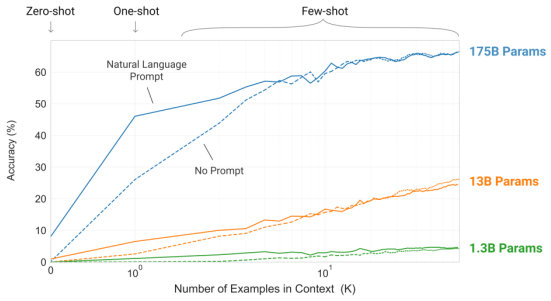

Figure 10: Few Shot Learning with rising examples

3 Conclusion

One can say the development of the transformer model is a breakthrough for large language models and machine learning in general. The introduction of the self-attention mechanism and moveing away from recurrence and convolution led to significant improvements in various NLP tasks. GPT-3's ability to perform a wide array of tasks without finetuning showcases the robustness and adaptability of transformer-based models.

References

Bahdanau, D., Cho, K., and Bengio, Y. (2016). Neural machine translation by jointly learning to align and translate.

Britz, D., Goldie, A., Luong, M.-T., and Le, Q. (2017). Massive exploration of neural machine translation architectures.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). Language models are few-shot learners.

Dai, A. M. and Le, Q. V. (2015). Semi-supervised sequence learning.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

Kudo, T. and Richardson, J. (2018). Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint arXiv:1808.06226*.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space.

Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. (2018). Improving language understanding by generative pre-training.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., *et al.* (2019). Language models are unsupervised multitask learners. *OpenAI blog*, **1**(8), 9.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, **30**.