Transformers learning Graphs

Becsó Gergely

May 2024

1. Introduction

Today the large language models (LLMs) based on the transformer architecture are the main focus of artificial intelligence research. These models are the best solution to various problems, they can be chatbots providing valuable information. Some are thinking, they are the future of programming, as they already can give codes for many simple task and the progress is fast. Generating music, art, videos are more controversial, but just as high-impact tools. They are used in translating, voice processing, machine vision and many more areas.

For this reason, understanding these models inner workings are essential. The topics of AI-safety and interpretability are crucial. Yet, we usually have just educated guesses or even less when questions about these inner mechanisms are asked. From what part of the training data do they conclude their answer? How is the knowledge stored? What is even stored? Is there a model of the world/problem they were trained on? If not, where does the answer come from? If yes, can we somehow access that inner model? And many more similarly interesting questions are still largely unanswered.

In our approach we take a step back from LLMs as they are trained to do a generic problem that is hard to fully comprehend or understand. They have millions of parameters that is untrackable. Starting from a lot simpler task and using tiny, very simple "toy" models might also give us some interesting answers, while they are much more manageable, faster to train. Similarly, choosing a very specific problem and training the models to solve only that can help us, as we only need to identify the inner mechanism for a well known problem and separate them from the initial randomness. That is why we designed a custom, super simplified transformer and trained it on the shortest path problem on simple graphs.

Our approach is to train these *small* networks on *shortest path* questions on some graphs. Many different setups are possible, two of them being *giving the graph as context* as input or *only asking the question* and let the training process teach the graph structure to the model. When only the question is given to the network during training it need to solve the problem: reconstruct this graph, that is only given by an oracle, and one can make queries of shortest path questions. If the question is 'can this small network learn this graph', overfitting is not even a problem. Still, different types of generalizations can be checked, as one training usually takes only a short time — an other big plus for using toy models.

Researches has shown, that until a point these networks can only memorize input-output pairs — what still can be interesting — but after a while they can get an deeper understanding.

2. Transformer Networks, LLMs

Historically the transformer architecture can be derived from sequence processing, specifically natural language processing (NLP). The back then state of the art LSTM networks could be boosted by giving them a so-called attention mechanism, where embedding of a token (word) was modified based on the context — the other tokens around it. In the NLP setting it made a lot of sense, as for example the meaning of the word "take" largely depends on the context. In 2017 Vaswani et al. wrote the paper *Attention is all you need* [Vas+23]. The main take here is that this attention mechanism is so powerful, that it is all you need. And indeed, as they introduced the transformer architecture, which showed great results, the research community tried it out and refined on many different problems and it had a massive success. With that in the NLP field from 2018 the main way of improvements were teaching larger and larger modells on larger and larger datasets — the first ones were GPT and BERT, and many such models are used in the world today.

The general transformer network consists of and embedder, a positional encoder, an encoder network, consisting of transformer blocks, which are an attention layer and a dense feed forward layer (and layernorms, dropouts, etc.), a decoder network, which is almost the same as the encoder, but also has a mask mechanism, which is useful for the causal loss (which is outside of the scope of this essay), and an aggregation layer. In the following section we will discuss what each part does in the network.

The inputs of a transformer network are called 'tokens', they are a unit of whatever sequence the model is trained on. Generally they can be letters, syllables, numbers, pixels, etc. They always can be mapped to the natural numbers, so the domain of an embbeding layer is usually the natural number.

The embedder layer The embedder layer of a transformer gets the tokens as inputs and embeds each into a high dimensional vector space (usually into $\mathbb{R}^{512}$). Lets denote the dimension of this latent space with d. The embeddign layer can be a one-hot encoding, a learnable parameter of the network or some other method.

The positional encoding If the network uses the original sinusoidal positional encoding based on Vaswani et al. [Vas+23] which is based on Fourier transforms, we need an upper bound on the possible input sequence length. Originally it was 10000 so let us use that in this explanation.

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d}}\right)$$
$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d}}\right)$$

Here,

- pos is the position in the sequence,
- i is the dimension index.

This is the positional encoding that is added to the embedded input, and helps the model to differentiate between the same token if they are on different position. It also makes sure, that the model can identify the distance of a token pair, as usually in the context of a given token the close tokens are playing a more important role.

An alternative to this is the *Rotary Positional Embedding*, *RoPE* that will be discussed at the next section.

Transformer blocks A transformer block has two main parts are the attention and the feed forward layer. The attention computes a relevance for each token pair, using the attention mechanism, which quantifies the importance of each token for any other tokens context.

The second part of a transformer block is a feed forward neural network. This has one parameter, the expansion factor f. It consists of two dense layer, one being an $\mathbb{R}^{d\times fd}$ and an other one $\mathbb{R}^{fd\times d}$. These are applied to each token in the sequence, leaving the dimensions of the tokens unchanged.

After the attention and the feed forward network normalization layers, dropout layers can be added as needed.

The encoder architecture is such transformer blocks. A decoder block has a mask, that is applied to the attention matrix — it zeroes out the attention score for each token after it. This makes sense from a text generation point of view.

Aggregation layer At the end of the encoder-decoder layers we got a high-dimensional embedding of the original tokens. We need to map them to a format, that is easily interpretable as tokens. As such, if the vocabulary of the problem ha m tokens, an appropriate aggregation layer would be an $\mathbb{R}^{d\times m}$ dense layer. These then can be interpreted as probabilities for the given token.

3. Network Architecture

I designed a costum transformer network for these experiments. As LLMs have a huge parameter space and are containing enormous amount of information, to understand the inner mechanisms first a small toy model is practical.

From a theoretical point of view, the essential part of a transformer are the embedding layer, the positional encoding, the attention layers and the aggregation layer. To get as small of a model as possible we are keeping only these.

Embedding For the embedding layer the standard pytorch Embedder is used. This is a learnable layer, that given a vocabulary size gives an embedding to each possible token. The learnable part is, that these embedding can change during training.

Attention layers To understand the attention mechanism better, let us walk through the inner computations of our attention: at this point, the example input is $X \in \mathbb{R}^{k\times d}$.

We create the *key*, *query* and *value* matrices as follows:

- $Q = XW^q$,
- $K = XW^k$,
- $V = XW^v$

and W^q, $W^k \in \mathbb{R}^{dxd'}$ and $W^v \in \mathbb{R}^{kxd''}$. d' and d'' are choosable parameters. Using them the attention mechanism is computed by the

$$X \leftarrow \mathrm{softmax}\left(\frac{QK^T}{\sqrt{d}_k}\right)V.$$

formula. The dimension of the output of the attention is $k \times d''$. The standard choice is $d' = d''$, and we call it the inner dimension of the attention layer. We call the matrix QK^T the attention matrix, as it holds the information of the pairwise token relations.

Rotary Positional Embedding is a positional embedding from the paper 'RoFormer: Enhanced Transformer with Rotary Position Embedding' by J Su et al. [Su+23]. The high level concept is to rotate the keys and queries. Let R is the rotary matrix of the rotational embedding. If $d' = 2$, this is a rotation matrix. For some $d > 2$ we may assume that d' is even. Then we can create $d'/2$ rotation matrices with different rotation angles. Placing them onto the diagonal of a $\mathbb{R}^{\times}$ matrix and filling the rest with zeros we obtain the R matrix. This rotates an embedding with one unit. Applying this i times to the i-th element of the sequence we get the encoding. We apply this rotation to the Q and K matrices, as their columns are the embeddings.

An important technique here is that this is the one-headed attention layer. Using the 'multi-head attention' we make multiple of these and concatenating the output of them. This gives the model a lot of inner space to work with, sometimes different heads can compute different aspects of the inner-token relations.

In the transformer block the attention can be implemented either as-is or in a residual fashion: the output of the attention is added to the input and this value is forwarded to the next layers.

After the attention mechanism, the dimension might have changed, due to multi-head mechanisms and $d'' \neq d$ choices. For this reason we should map each token-embedding to d dimension. This can be done via a dense layer. In our experiments $d' = d'' = 64$.

For the aggregation layer the standard choice of a dense layer is implemented as well.

4. Data generation

In order to generate input data and labels for the transformers we need to encode the graphs, the graph problem and the solution into a token-series, we need to tokenize them.

My approach: given a simple graph $G = (V,E)$, first I enumerate/name the vertices of the graph with $i = \{1,\ldots,n\}$ in an arbitrarily ($n = |V(G)|$). If we would like the network to see the graph, then I list them via their names. After that the edges should be listed as well. In order to make the separation between the different part of the input better, lets use 0 as a separator sign and $n+1$ as the empty sign. To list the edges, iterate through the u, v edges and append $0uv$ to the input. This way the graph is tokenized, and is in an understandable format for the network. We need to specify the question to the network, for the shortest path problem this can be done with $s,t \in V(G)$, so appending $0s0t0$ is sufficient. This also makes sure, that it is different from the previous edge descriptions. To simplify the system and keep the meaning of the tokens in different positions, it is suitable to append n piece of $n+1$ token, as the shortest path contains at most n vertices. To generate the labels for the modells, we perform the same, with the modification, that at the last step first the vertices of the shortest path should be listed, then complete the labels with the necessary number of noise tokens.

This way we expect the network to keep the first $|V(G)|+3\cdot|E(G)|+5$ tokens as-is and using the attention between them to show, how did the network learned the graph structure.

5. Experiments / Measurements

Data was generated from the path graph P_{10} and some Barabási-Albert graphs with 10 vertices and 4 as the connection parameter.

During the experiments the AdamW optimizer and cross entropy loss were used. Each training 5000 epochs. Batch sizes were 64.

5.1. Parameter optimisation

Originally we tried the sinusoidal encoding, which couldn't reach perfect accuracy. Thus we implemented *RoPE*, and the network described above has shown much better results.

The models are very sensitive towards the learning rate, $lr = 10^{-4}$ worked.

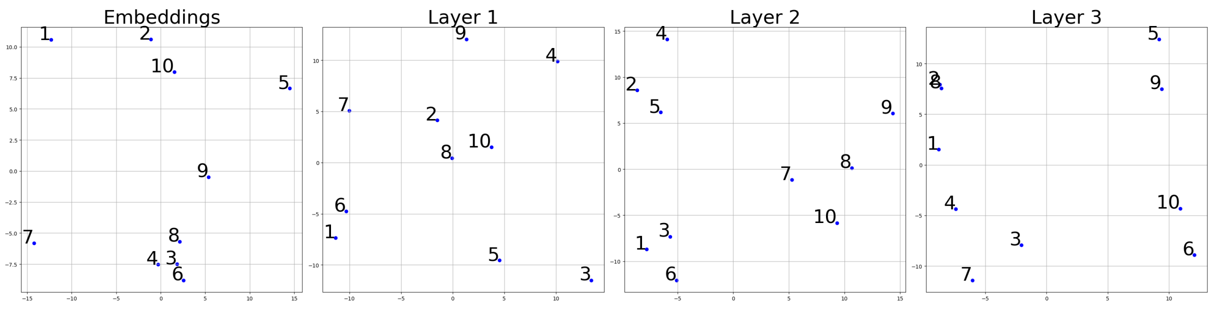
The number of layers did not impacted the learning capabilities highly, so we fixed it for most experiments at 3.

With such parameters the network could reach 100% accuracy on all the graphs tested.

5.2. Visualizing embeddings

Giving the tokenized graph as a context or not did not had big significance from a performance point of view. There is a difference in interpreting the results. We can use *PCA* or $T-SNE$ to interpret the original learnt embeddings and their changes after each layer of the network.

For the one without graph we passed through the node tokens of the graph to see what happens with the respective embeddings. Here we used *PCA* to visualize the flow of the node tokens of a path graph through the embedding and attention layers in a model that can solve the shortest path problem with 100 % accuracy.

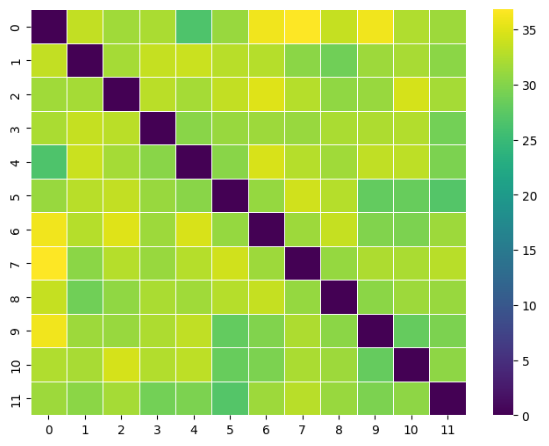


Embeddings of vertice tokens through a model

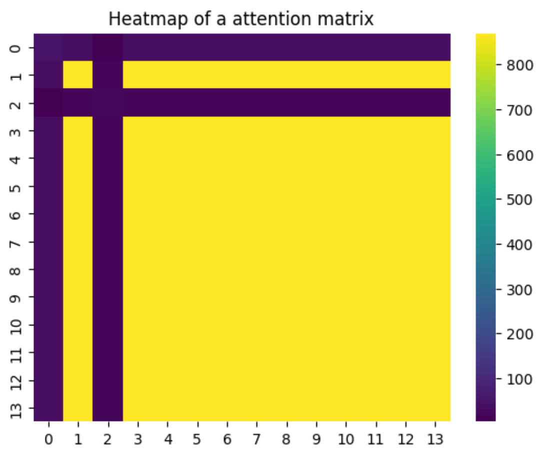
As we can see, the visualization did not show a recognizable P_{10} structure.

With the graph as context we can do the same, but we also have other options. We can read out the graph vertice embeddings from any input, which can give us more hope a priori to show some nice structure.

We can always take the embeddings of all of the tokens and compute their pairwise similarities. High similarity here can by high scalar product or low distance by some distance function. The hope would be for this matrix to show some similarity to the adjacency matrix of the graph.



(a) Distance heatmap of token embeddings



(b) Attention matrix without graph context

As we can see, the distance heatmap 1a does not resemble the adjacency of the path graph on 10 vertices. On picture 1b we can see the attention heatmap for a question without the tokenized graph. It shows, that each path token pays equal attention to each other, but learned to ignore the separator signs in the question.

In summary the simplest forms of checking the embeddings did not gave us proof, that the model stores edge relations in the way it encodes node tokens.

5.3. One graph-one model generalization

Even though generalization is not necessary to our approach, understanding a model involves checking, whether it can generalize to unseen problems, or not. Splitting the data into train and test datasets we can check, whether the network can generalize or not. For this split we have many options. The standard method to this is to **randomly split** the dataset into train and test data. In our case we could also **filter the dataset** by the length of the path — train on certain lengths and test on the rest. A third option is to use the inherent symmetry in the dataset via **reversing paths**. We would split the data by that: if a question s,t is in the training data, then t,s should be in the test set.

The second approach under the hood touches a theoretical question: if a graph is given through an oracle and we can get queries about shortest paths — which sequence of queries provides enough information to know everything about the graph? If the queries can be only questions, for which the answer is a 5 long path, what can we say? A better formulation of this question is how much information is carried by all the 5 long shortest paths of a graph?

Random split We splitted the training data randomly into training and test data with their ratios being 0.8 : 0.2. Repeated the experiment 5 times. Surprisingly one time the model did not reached accuracy 1 on the training set. Validating on the test data gave the following results:

Training Accuracy	Correct Tokens	All Tokens	Test Accuracy	Correct Tokens	All Tokens
0.95	213	224	0.87	244	280
1.0	224	224	0.91	255	280
1.0	224	224	0.89	250	280
1.0	224	224	0.90	252	280
1.0	224	224	0.87	246	280

Table of accuracies

As in average half of the tokens are noise or separator tokens this result is much higher than random.

Path length split We ran numerous experiments with generating data from different path lengths and testing them on others. The length of a path is measured by the number of vertices contained.

Training Lengths	Testing Lengths	Accuracy
2	3 4 5 6 7 8 9 10	0.66
2	1	0.85
2	3	0.82
5	7	0.61
5	1	0.65
1 2 3 4 5	6 7 8 9 10	0.61
1 2 3 4 8 9 10	5 6 7	0.74
6 7 8 9 10	1 2 3 4 5	0.64

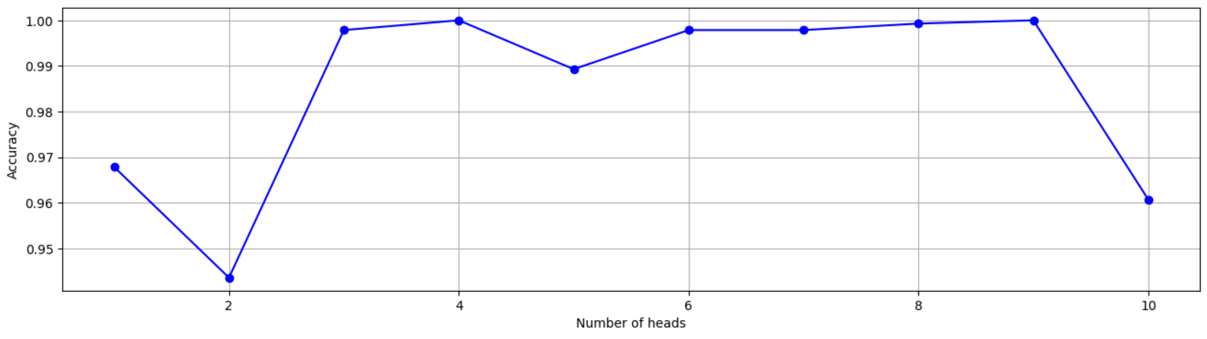
Path length generalization

As the labels are containing the questions as well, the seen accuracies are not showing noticeable ability for generalization. Looking at input-output pairs of the models the correct guesses are mostly the question tokens, noise tokens at the end of the answer and some actual correct guesses at the path.

Such experiments on Barabási-Albert graphs are not viable, as their diameter is relatively small.

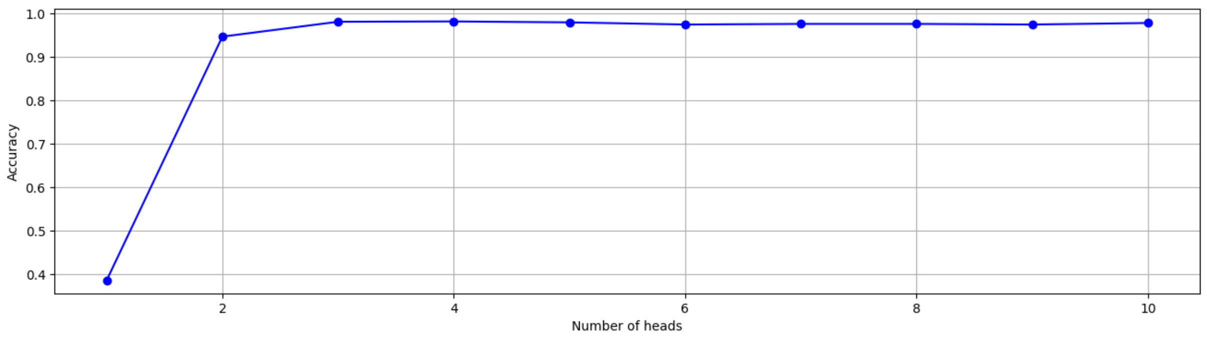
5.4. Number of heads

While using multi-head attention is the standard choice, we checked how much of an impact it actually made.



Accuracy with respect to number of heads

We can see, that the model can easily learn the problem with any number of heads. Looking at the impact for generalization abilities with 80-20 random train-test split:



Generalization accuracy with respect to number of heads

We can clearly see, that adding more attention heads gives more understanding ability to the model and it generalizes better to unseen questions.

6. Conclusions

We implemented a toy model with *RoPE*, ran multiple experiments, gained a basic understanding of the behaviour of the problem.

Generally across all experiments it seems that seeding has a big impact on the capabilities of the models. The models can show ability of generalization if the training data contains examples of all path lengths.

We learned, that in these data formats the embeddings of the graph vertices are not resembling a embedding of the graph, where connected vertices are tending to be closer.

7. Future plans

As the topic is just started, many more questions and experiments are yet to be answered or ran. The most ambitious is to teach many networks on different graphs until perfect accuracy, then use these networks and their graphs as data to yet an other graph. If on this dataset with some appropriate train-test split the model learns to generalize, then we know, that it is possible to reconstruct the graph from the weights of the model and we have a network, that does exactly that.

A more thorough examination of the embeddings are needed. Can the distance matrix show some resemblance of the adjacency matrix if the input data is of different format?

Experimenting with different graph problems can be also interesting in this topic.

Sources

[Su+23] Jianlin Su és tsai. *RoFormer: Enhanced Transformer with Rotary Position Embedding*. 2023. arXiv: `2104.09864 [cs.CL]`.

[Vas+23] Ashish Vaswani és tsai. *Attention Is All You Need*. 2023. arXiv: `1706.03762 [cs.CL]`.