Prompt Engineering, Architecture Analysis, Model Modification

Aleksandr Algazinov
algazinovalexandr@gmail.com

May 6, 2025

1 Prompt Engineering

1.1 Designing a Unique and Challenging Prompt

The goal is to design a prompt, representing a particular problem. The problem must have a clear and verifiable answer. The next step is to give this prompt to two of the advanced reasoning models. One of them must solve the problem for three times in a row, while another is expected to constistently generate the wrong answer.

Until recently, LLMs have been struggling to solve complex mathematical problems. With the recent developments of reasoning capabilities of LLMs, the problem has been partially solved. However, LLMs still struggle with complex problems that require deep understanding and reasoning. Combinatoristics is famous for its complexity, and some of the problems require non-trivial approaches to be solved. Hence, let's check if LLMs can solve a combinatorics problem that requires deep understanding and reasoning. The problem and the prompt are as follows:

Grandma invited six kids to have dinner with her and she prepared 6 pairs of chopsticks with different colors. Two sticks for each pair are the same. Bonnie helped Grandma to hand out the chopsticks to all the other 5 kids and herself. But unfortunately, there is no kid who got a matching pair of chopsticks but Bonnie got a pair. Please figure out how many different ways for Bonnie to distribute the chopsticks?

This problem is one of the problems from the Combinatorics and Algorithms course at Tsinghua University (2019 final exam for the combinatorics part of the course), and the correct answer is 12240. Fig. 1 shows the official solution to the problem. Three models were tested with this prompt: GPT-4 with reasoning capabilities (free version available on the chatgpt website), Deepseek with deep thinking, and Gemini 2.5 Pro. Deepseek took more time to solve the problem than Gemini and GPT-4. We will analyze the results of Gemini and GPT-4, since they were faster in solving the problem. For checking the consistency of the results, we will provide the prompt to each of the model three times, and see whether the models will provide the same answer.

Figures 2-4 show the results of the three experiments with GPT-4 model. As it can be seen, the model provided the same answer in all three cases, and it is correct. Figures 5-7 show the results of the three experiments with Gemini 2.5 Pro model. As it can be seen, the model provided the same answer in all three cases, even though it is incorrect. This makes the experiment successful given the objective.

1.2 In-Context Learning

The goal of this task is to design two prompts: one uses in-context learning with several in-context demonstrations (few-shot learning), while another does not use in-context learning (zero-shot learning) to see how prompt engineering affects the performance of LLMs on tasks that require simple reasoning.

For doing so, we will use the 3 samples from the GSM-8K dataset (we collected the samples with the 1297, 4336, and 5196 indices).

Zero-shot learning prompt: Solve the following three problems:

1. In Professor Plum's biology class there are 40 students. Of those students, 80 percent have puppies. Of those who have puppies, 25

2. Running for 2 hours, Jonah burnt 30 calories every hour. How many more calories would he have lost if he would have run for five hours?

3. Mark is reading books, for 2 hours each day. He decided to increase his time spent on reading books

weekly, by 4 hours. How much time does Mark want to spend during one week on reading books?

Explain your reasoning for each problem separately step by step, but be concise. Return your answer as a json file with the following format:

"Question ID": list, "Reasoning Process": list, "Final Answer": list, "Difficulty Classification": list

Difficulty classification is a number from 1 to 5, where 1 is the easiest and 5 is the hardest. Question IDs are the following: 1297, 4336, 5196.

Few-shot learning prompt: You need to solve the following three problems:

1. In Professor Plum's biology class there are 40 students. Of those students, 80 percent have puppies. Of those who have puppies, 25

2. Running for 2 hours, Jonah burnt 30 calories every hour. How many more calories would he have lost if he would have run for five hours?

3. Mark is reading books, for 2 hours each day. He decided to increase his time spent on reading books weekly, by 4 hours. How much time does Mark want to spend during one week on reading books?

Explain your reasoning for each problem separately step by step, but be concise. Return your answer as a json file with the following format:

"Question ID": list, "Reasoning Process": list, "Final Answer": list, "Difficulty Classification": list

Difficulty classification is a number from 1 to 5, where 1 is the easiest and 5 is the hardest. Question IDs are the following: 1297, 4336, 5196.

Below are the two examples for your reference. Use these examples to better understand how to perform the task well (after that I provide the contents of the json files):

The detailed analysis of the results can be seen in the Reasoning-Analysis.ipynb file. In short, both the few-shot and zero-shot learning prompts were able to solve the problems correctly. When adding few-shot examples, the output content did not change much. Meanwhile, the answer looked more like the examples in the dataset (mainly due to the calculations in the ¡ and ¿ brackets). Hence, few-shot learning worked, but not sure if it was needed for such simple problems, given that modern reasoning models are already powerful enough. Few-shot is more useful for complex problems, where the model actually needs to learn the pattern from the examples, or when we want to adapt the output style to some particular format (like was done in our case). However, few-shot learning is not always needed, and zero-shot learning can be used as well. Moreover, few-shot requires more time and resources, since we need to provide the examples, and spend more time designing the prompt.

2 NLP Architecture Analysis

In this part, we will calculate the model size, KV cache size and FLOPs in a forward pass of the following models: **GPT, GPT with GQA (Grouped Query Attention), Mamba-2, and GLA (Gated Linear Attention)**. Our assumptions are described as follows:

- Model parameters and KVs are represented in bfloat16 precision.
- The number of FLOPs for matrix multiplication between two tensors with shapes (a, b) and (b, c) is approximated as 2abc.
- The model is a standard GPT-2 model, which means the model:
-
  - Does not use SwiGLU FFN, which is common practice nowadays. GPT-2's FFN consists of two linear layers instead.
  - Uses an attention head size that equals d/n_h.
  - Uses tied embeddings, which means the LM head (a.k.a. output embedding layers) shares the same parameters with the input embedding layer (a.k.a. embedding table).
  - Uses learned position embeddings.
- All the model components that do not make up a singificant part of the total parameters (such as bias terms, layer normalization parameters, etc.) can be ingnored. In addition, FLOPs of the following operations can be ignored. Hence, our calculations might not be exactly correct, but the accuracy of such approximations is high.

All the computations are made in the model-parameter-estimation.ipynb jupyter notebook. Please check it to see the results in terms of the code.

2.1 GPT

Model size: GPT-2 has the following hyperparameters: L = 1024 (input sequence length), V = 50257 (vocabulary size), d = 768 (hidden size), N_{layer} = 12 (number of layers), d_{ff} = 3072 (Intermediate dimension of the feed-forward network (FFN)), n_{head} = 12 (number of attention heads), and $head_{dim}$ = 64 (dimension of each attention head). Each layer has:

- QKV projection: $3 \times (d \times d)$.
- output projection: $(d \times d)$.
- FFN block: $d \times d_{ff} + d_{ff} \times d$.

In addition, we need to take into account the parameters in the embedding layer, and the positional embeddings (V x d and L x d, respectively).

Hence, the model size (give our assumptions) is calculated as the sum of three components: number of parameters per layer times the number of layers, plus the parameters in the embedding layer, plus the parameters in the positional embeddings. Equation 1 shows the calculation of the model size for GPT-2 Small, and the calculated number of parameters is 124,318,464, which is around 237.1 MiB.

$$par_{gpt2} = V \times d + L \times d + N_{layer} \times (3 \times d^2 + 1 \times d^2 + d \times d_{ff} + d_{ff} \times d) = 124318464. \quad (1)$$

In reality, the size of the GPT-2 Small is around 124M parameters, hence, the approximation is accurate.

KV cache size: During the inference phase, the batch size is 1, and the K or V cache size per layer is calculated as the number of attention heads times the head dimension times the sequence length. Then, we multiply the obtained value by the number of layers and 2 (making sure we calculate the elements both for k and v). Equation 2 shows the calculation of the KV cache number of elements for GPT-2 Small. To calculate the size, we multiply the number of elements by 2, and the calculated size is around 36 MiB.

$$KV_{gpt2} = 2 \times N_{layer} \times n_h \times head_{dim} \times L = 18874368. \quad (2)$$

FLOPs: It is known that a matrix A with shape (a, b) and a matrix B with shape (b, c) can be multiplied in 2abc FLOPs. First, it is needed to calculate the number of FLOPs per transformer layer, which consists of the following components:

- QKV projection: project hidden states with shape (L x d) to three matrices with shape (L x D): $3 \times 2(L \times d \times d) = 6Ld^2$.
- Attention core: Attention core FLOPs $= H \times \big(2 \times (L \times D_h) \times L + 2 \times (L \times D_h) \times L\big) = 4\,H\,D_h\,L^2 = 4\,d\,L^2$.
- Output projection: it is needed to project concatenated heads back to d, and the cost is $2(L \times d \times d) = 2Ld^2$.
- FFN block: since we have two linear layers, and the cost for each is $2Ldd_{ff}$, the total cost for this component is $4Ldd_{ff}$.

The final output of the model are logits, and the cost of the final projection is 2LdV. To get the final value, we need to add the cost of the last projection and the cost of FLOPs per transformer laye times the number of layers. Calculations are shown in equation 3, and the total FLOPs are equal to 291648307200, which is approximately 291.6 GFLOPs.

$$FL_{gpt2} = N_{layer}(6 \times L \times d^2 + 4 \times d \times L^2 + 2 \times L \times d^2 + 4 \times L \times d \times d_{ff}) + 2LdV = 291648307200. \quad (3)$$

2.2 GPT with GQA

As opposed to multi-head attention (MHA), where there is a key and a value for all the queries, in grouped query attention (GQA) there are G key heads and G value heads (G is smaller than the number of query heads). Hence, each of the query heads is assigned to one of those G groups. Since we decrease the number of key and value heads, we decrease the number of parameters in the model. In this subsection, we will not dive as deep into the details of the calculations as we did in the previous subsection, since the idea of the calculations is similar.

Model size: The only difference between this model and the classical GPT-2 small is that now we

have less key and value heads per layer. Parameters for QKV used to be equal to $3d^2$, but now they are equal to $d^2 + 2 \times d \times (G \times d_k)$, where d_k is $d//n_h$. In our example G = 4. Subtracting the number of new QKV parameters from the number of old QKV parameters, and multiplying by the number of layers, we get the difference in the model size. It turns out that the new model is 9437184 smaller than the previous one, and the new model size is 114,881,280 parameters, which is around 219 MiB.

KV cache size: Using the same logic, the formula for the calculation changes from $L \times N_{layer} \times n_h \times head_{dim} \times 2$ to $L \times N_{layer} \times G \times head_{dim} \times 2$. This results in the KV-Cache equal to around 12 MiB, making it three times lower compared to the original GPT-2 Small (significant reduction).

FLOPs: In this case, only the number of FLOPs for the projection part changes from $4 \times (2 \times L \times d \times d)$ to $(2 \times L \times d \times d) + 2 \times (2 \times L \times d \times (G \times head_{dim})) + (2 \times L \times d \times d)$. This results in the reduction of GFLOPs from around 292 to around 272.

2.3 Mamba-2

Due to the time constraints, the results in this subsection will be described in less details compared to the previous two. Please check the model-parameter-estimation.ipynb jupyter notebook to see the calculations.

Model size:

$$par_{mamba2} = V \times d + N_{layer} \times (4 \times d^2 + 2 \times d \times E \times d). \quad (4)$$

Model size turned out to be around 181.62 MiB, while the number of parameters is 95220480.

KV cache size:

$$KV_{mamba2} = N_{layer} \times L \times 2 \times d. \quad (5)$$

KV-cache size turned out to be around 36 MiB.

FLOPs:

$$FL_{mamba2} = N_{layer} \times (8 \times L \times d^2 + 4 \times L^2 \times d + 4 \times E \times L \times d^2). \quad (6)$$

Number of FLOPs per forward turned out to be around 154.6 GFLOPs.

2.4 GLA (Gated Linear Attention)

Due to the time constraints, the results in this subsection will be described in less details compared to the previous two. Please check the model-parameter-estimation.ipynb jupyter notebook to see the calculations.

Model size:

$$par_{gla} = V \times d + N_{layer} \times (4d^2 + 2 \times d \times d_{ff}). \quad (7)$$

Model size turned out to be around 236 MiB, while the number of parameters is 123532032.

KV cache size:

$$KV_{gla} = N_{layer} \times n_h \times L \times (d_k + d_v). \quad (8)$$

KV-cache size turned out to be around 36 MiB.

FLOPs:

$$FL_{gla} = N_{layer} \times (8 \times L \times d^2 + 2 \times L^2 \times (d_k + d_v) + 4 \times L \times d \times d_{ff}). \quad (9)$$

Number of FLOPs per forward turned out to be around 177 GFLOPs.

Fig.8 shows the table summarizing the results of the calculations. For each model, we have the number of parameters, the size of the model, the size of the KV-cache, and the number of FLOPs per forward pass. As it can be seen, GPT-2 with GQA is the least in terms of KV-Cache, and all the metrics are smaller compared to GPT-2 with classical MHA. It means that the idea of reducing number of keys and values paid off, and, assuming it does not negatively affect the quality of text generation (which probably is the case, although the quality reduction might not be very significant), should be applied to make both the training and inference faster and less computationally expensive. Mamba-2 turned out to be the most light-weight model (in terms of both number of parameters and FLOPs), making it the least computational demanding and the fasters out of the four. Note that in our analysis we cannot compare the model performance, since we are not actually training these models. We are limited to theoretical analysis and comparison of their characteristics. GLA model, in terms of number of parameters, turned out to be almost similar to the GPT-2 small model with MHA, making it more heavy compared to GPT-2 with GQA, and Mamba-2. Meanwhile, in terms of FLOPs it is significantly more efficient than both versions

of GPT models, and not way behind the most optimized Mamba-2 model. Regarding KV-Cache, almost all models have it equal to 36 MiB, except for GPT-2 with GQA (12 MiB).

Note that my calculations for Mamba-2 and GLA might not be very accurate, since computations of the following metrics is a complex task. Hence, the conclusions I made regarding the trade-offs, advantages, and disadvantages of each model might not be very accurate as well. Nevertheless, I tried my best.

3 Model Implementation

For this task, I chose to modify Qwen2.5 architecture, so that the following changes to the architecture are made:

- **Learned Position Encoding:** We replaced the rotary positional encoding in Qwen2.5 with learned position encoding, which is added to the hidden representations right after the input embedding layer.
- **DyT (Dynamic Tanh):** We replaced all normalization layers (RMSNorm) with a Dynamic Tanh (DyT) module, which can be more efficient while maintaining performance. Please refer to the "Transformers without Normalization" paper (https://arxiv.org/pdf/2503.10622) for more details.
- **ReLU-Attention:** We replaced the softmax operation in attention with a ReLU-Attention. Please refer to the "Replacing softmax with ReLU in Vision Transformers" paper (https://arxiv.org/pdf/2309.08586) for more details.
- **Layer-Dependent Attention Mask:** We changed the standard causal masking to the following attention mask: (1) The first layer applies a causal attention mask, restricting each token to attend only to past tokens. (2) The last layer uses dilated sliding window attention with a window size of 128 and a dilation factor of 2, meaning each token attends to a sparse subset of past tokens. Specifically, the token at position t attends to tokens at positions $t, t-2, t-4, t-6, ..., t-126$. (3) Middle layers apply a custom attention mask. For example, for a sequence of 16 tokens divided into 4 blocks, the attention mask is shown in Fig. 9.

We started by creating new conda environment and cloning the transformers repository by running git clone https://github.com/huggingface/transformers.git. After loading the necessary modules, we could import the original model:

```
from transformers.models.qwen2.modeling_qwen2 import
Qwen2ForCausalLM, Qwen2Config
```

Next, it was necessary to make sure it works by running the simple forward pass. Detailed code can be found in the load-and-test-qwen.ipynb jupyter notebook. After we made sure that it works on the dummy forward pass, we started modifying the model. The jupyter notebook summarises the changes that were made. In addition, for more details please refer to the modeling-qwen2.py file, which is the modified implementation of the Qwen2 model, while the modeling-qwen2-original.py file is the original implementation. Every change that was made is commented in the code, and is marked with the comment "CHANGED", so that it is more convenient for the reader to find the changes, and understand the ideas behind them. To make sure that the code was implemented correctly, we performed the test on the dummy forward pass, and it worked without errors. This means that the code was implemented correctly.

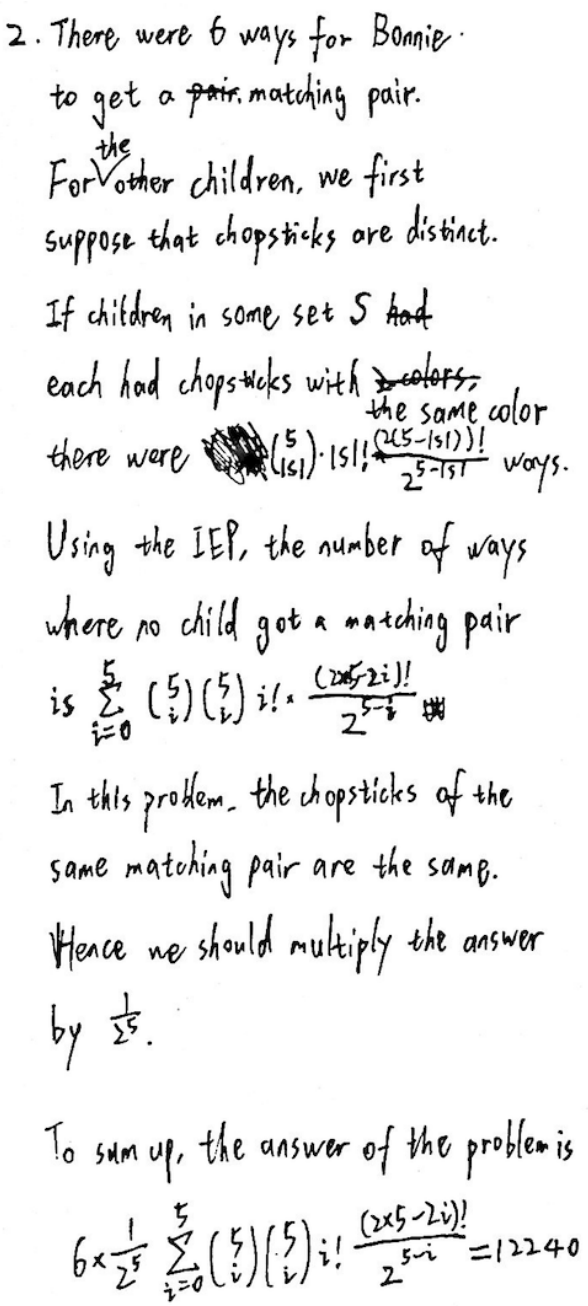

Figure 1: The official solution to the problem

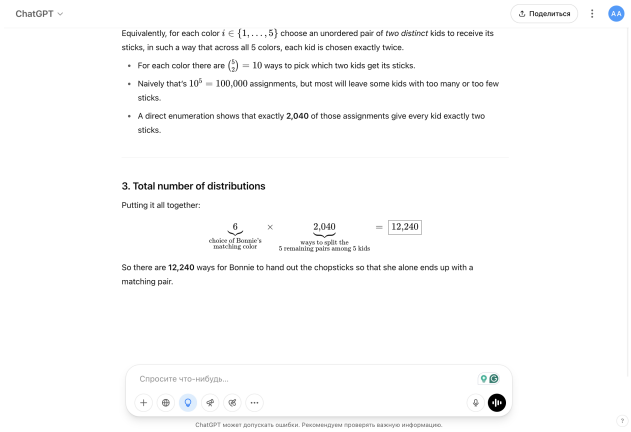

Figure 2: GPT-4 answer 1

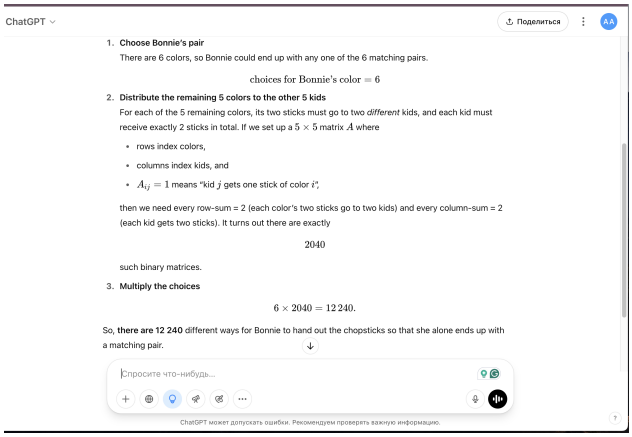

Figure 3: GPT-4 answer 2

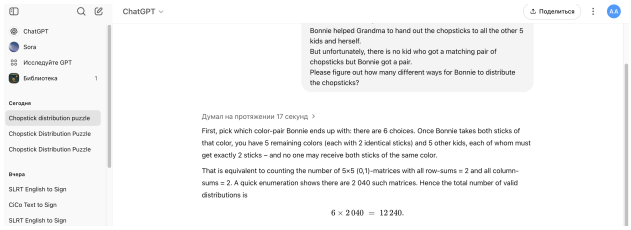

Figure 4: GPT-4 answer 3

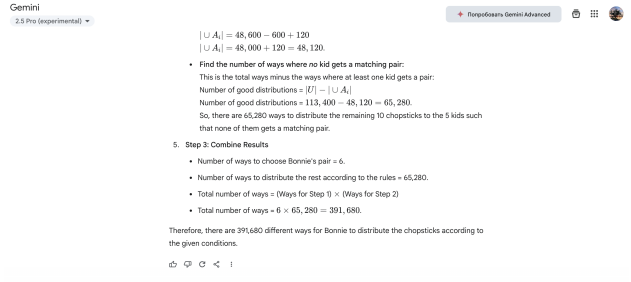

Figure 5: Gemini 2.5 Pro answer 1

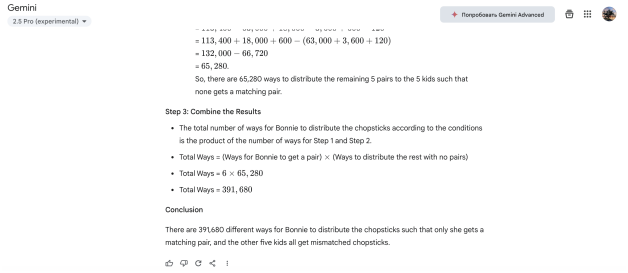

Figure 6: Gemini 2.5 Pro answer 2

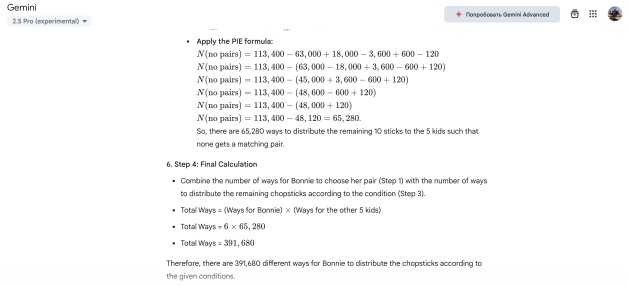

Figure 7: Gemini 2.5 Pro answer 3

	Model	Total Parameters	Model Size (MiB)	KV-Cache (MiB)	FLOPs per Forward (GFLOPs)
0	GPT-2 Small	124318464	237.118652	36.0	291.648307
1	GPT-2 Small + GQA (G=4)	114881280	219.118652	12.0	272.320954
2	Mamba-2	95220480	181.618652	36.0	154.618823
3	GLA	123532032	235.618652	36.0	177.167401

Figure 8: Example of the attention mask for middle layers.

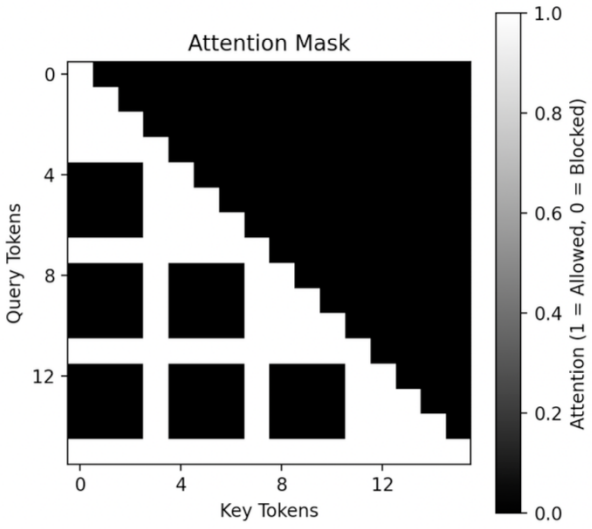

Figure 9: Example of the attention mask for middle layers.