Back to Articles

Everything About Long Context Fine-tuning

Community Article Published May 10, 2024

Upvote 52 +46

Belandros Pan
wenbopan Follow

Long Text, Big Models

Most large language models are pre-trained on contexts up to 8K in length. Recently, more and more big models have started supporting contexts longer than 32K. These long-text big models have brought new possibilities for document understanding, code completion, and other scenarios:

- **Reading Comprehension**: The main text of the GPT-4 paper is about 80,000 tokens long. Summarizing, extracting, or analyzing such a paper often requires complex Retrieval-Augmented Generation (RAG) methods. If we could directly pull all 80,000 tokens into the model, we could avoid truncating and extracting the original text. Compared to various complex RAG methods, this is way more pleasing.
- **Code Completion** is also a task that requires cross-referencing different locations far apart. If the model can accept an entire code repository, it can better utilize things like functions definitions in the repository, bringing significant advantages.

Today, the open-source community already has quite a few models that support context lengths exceeding 32K. However, many of these models only have the ability to spit out sensible text under long contexts and have not been specifically optimized for long-context tasks. On the other hand, the output style and capabilities of the out-of-the-box open-source models may not fully meet our needs.

In the above two cases, it becomes meaningful for us to do some long-text fine-tuning on OSS models ourselves. However, long-text fine-tuning is not as straightforward as regular fine-tuning. As the text grows longer, we need to first solve a series of problems caused by it.

The content of this blog is roughly as follows:

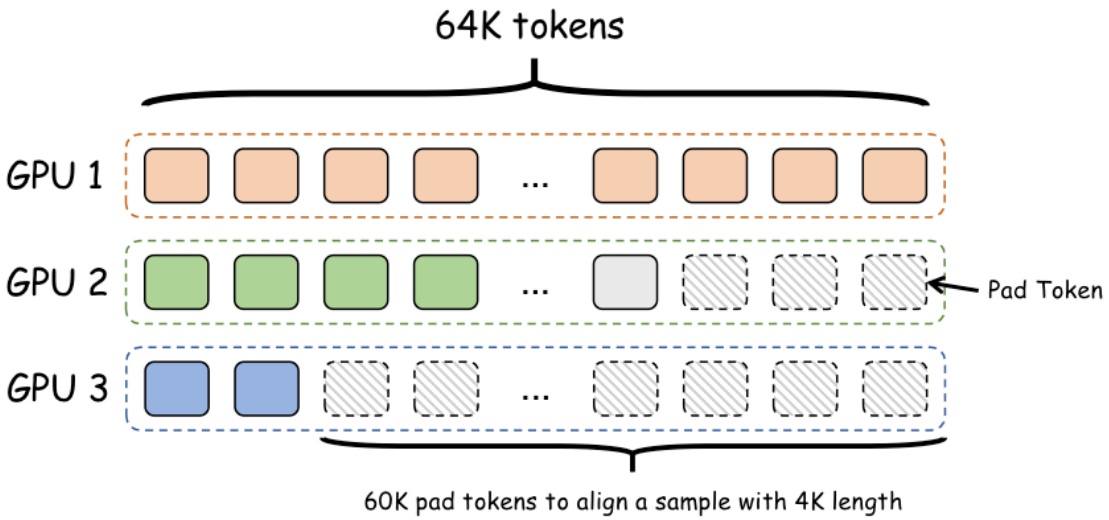
> *"The challenges of long text modeling: memory usage, batch alignment, and attention space complexity;*
> *How to solve the problems of long text fine-tuning*
> *A simple example of long text fine-tuning - the Faro series models"*

Disclaimer ⚠️ : Today, we focus on **the methods for fine-tuning**, not including how to improve long-text performance as much as possible or larger-scale pre-training, although these topic are also important. And as I am a Chinese, my choice of base models and datasets will take Chinese into consideration. You may want to make your modifications for your use case.

What Problems Does Long Text Bring?

As the context length increases, training efficiency becomes the most formidable challenge we face. I summarize this challenge as coming from the mechanism of fine-tuning when dealing with long inputs:

1. **Memory Usage**: During forward propagation, the model needs to calculate and retain the intermediate results (AKA, activations) of each layer. Specifically, each token in the context will have its own multiple Keys, Values, and Queries in each layer. As the context grows longer, these activations take up a lot of VRAM.
2. **Batch Alignment**: Long-text training with a batch size greater than one may waste a lot of space due to padding tokens, as long texts often span multiple orders of magnitude in length distribution. The image below shows an example.
3. **Attention Space Complexity**: Self-Attention needs to calculate the Attention value of each token to all other tokens in the sequence, and the result of this Attention calculation forms an $N \times N$ matrix. This means that the space complexity of Attention calculation is $O(N^2)$. So if the context length increases by 30 times, the space required for Attention calculation will increase by 900 times!



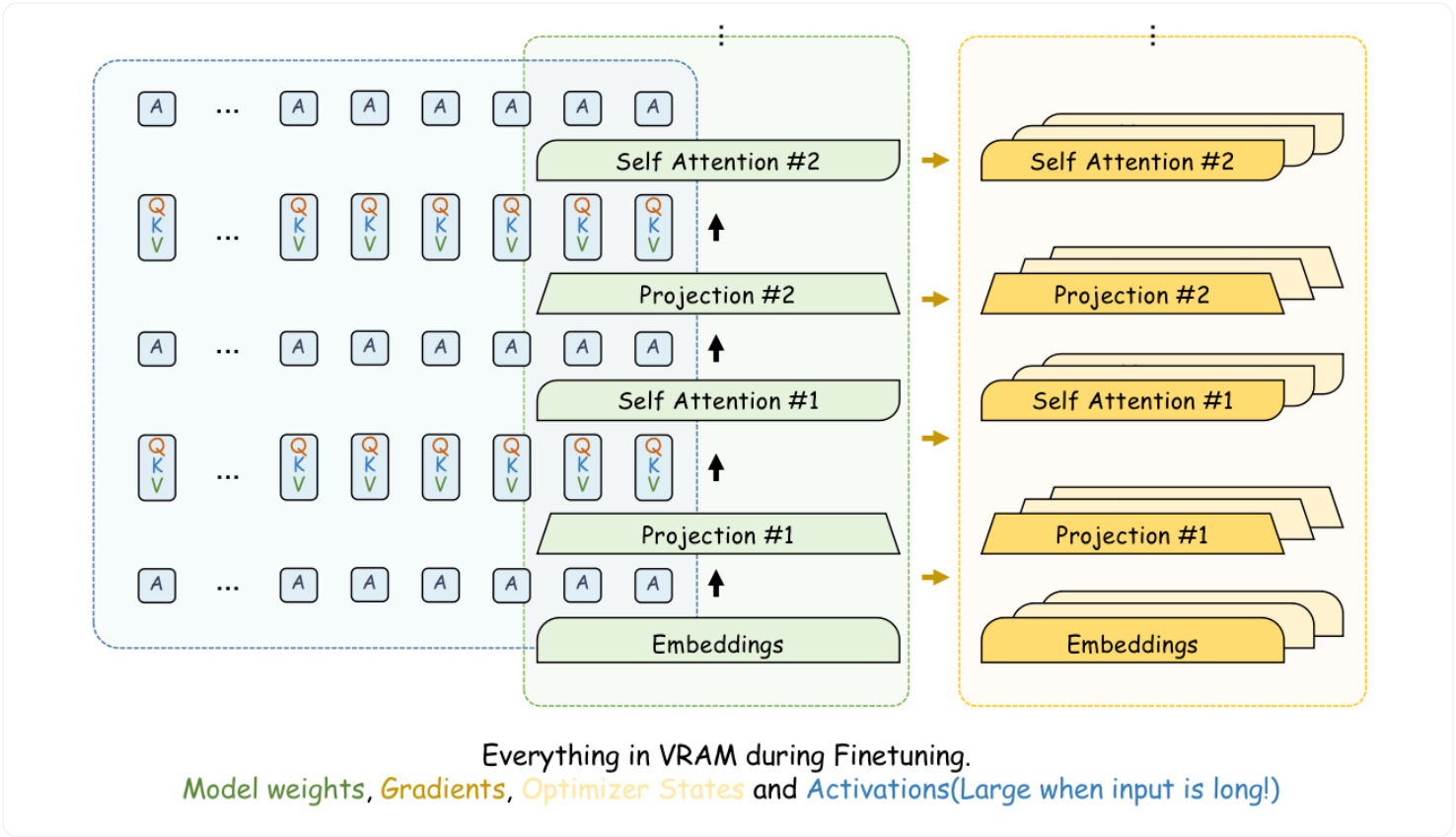
Lots of Tokens are <pad> when Input Length Varys

Fortunately, these problems can be properly solved under the blessing of various training techniques we have nowadays. We may not even encounter the above problems at all, as the default settings of the training framework we use may have already considered these problems. But knowing the existence of these problems and understanding how they are optimized is still very important.

Memory Usage

Ideally, a sample with a batch size of 1 and a length of 64K should occupy the same amount of GPU memory for its activation values, during forward propagation as a sample with a batch size of 32 and a length of 2K. Some may say that a batch size of 32 and a length of 2K is not an unrealistic setting for fine-tuning, especially for multi-GPU training.

However, the problem is that this does not mean that the same multi-GPU training method can be used for training with a batch size of 1 and a length of 64K, because most current parallel training frameworks (Deepspeed and FSDP) do not support distributing a sigle sample across multiple GPUs for training. To train on long texts, we need to use as many optimization methods as possible to save VRAM. From the most to the least memory-saving, we can consider the following techniques.



GQA

During forward propagation, each token needs to retain `num_attention_head` number of Query, Key, and Value vectors at each layer. These QKVs are the biggest contributors to memory usage. Most

models around 10B have `num_attention_head` = 32, which means each token needs to allocate 32 * 3 = 96 vectors.

This default method is Multi-head Attention (MHA). Models like Llama 2 7B, Qwen series, Command R all use this approach.

However, other models adopt a more efficient multi-head attention, namely Grouped Query Attention (GQA). GQA models allocate `num_attention_head` number of Queries to each token but allocate fewer Keys and Values. For example, Yi-9B has `num_key_value_heads = 4`, which means each token in Yi-9B is allocated 32 + 4 + 4 = 40 vectors. Models like Llama 2 70B, Llama 3, Mixtral, Mistral, Yi series models all use GQA.

The advantage of GQA is even more evident during **inference**, because during inference, since there is no need for backward propagation, the query of each token will not be used subsequently and will be discarded after the calculation is completed.

Therefore, in this case, if MHA allocates 32 * 2 = 64 vectors to each token, GQA may only need 4 * 2 = 8 vectors. The memory advantage brought by this is enormous.

Therefore, to achieve faster training and more efficient inference, we should choose models that use GQA as the base for fine-tuning. To determine if a model uses GQA, we only need to check if `num_key_value_heads` is smaller than `num_attention_head` in its configuration file.

Gradient Checkpoint

Gradient Checkpoint reduces the need to save the intermediate results of each layer's calculation during training. Only the intermediate results of some layers are keeped after calculation, such as the 1st, 5th, 15th... layers. During backpropagation, if the calculation results of a certain layer (such as the 7th layer) are needed, they are re-calculated by starting forward propagation again from the nearest Checkpoint (the 5th layer) to compute the intermediate results of the 6th and 7th layers.

Mathematically, to save the most memory, Gradient Checkpoint saves the results of $\sqrt{N}$ layers. For a 16-layer model, it only saves the results of layers 1, 5, 9, and 13 during forward propagation. Therefore, for the intermediate results that take up a large proportion of long texts, i.e., QKV, using Gradient Checkpoint will also reduce memory usage to $\sqrt{N}$ times.

LoRA

LoRA has essentially become the must-have method in the field of fine-tuning in many cases, especially in single-GPU training. Generally, when training a model, in addition to loading all the model weights in the GPU, additional gradients and optimizer states for tracking gradients dynamics are allocated for each trainable parameter. Depending on the type of optimizer, the optimizer state may take up 3 to 6 times the space of the model weights.

LoRA limits most parameters from being trained and introduces an additional small set of trainable parameters (LoRA Adapters), which often account for only 1% of the total weights. Only this 1% of trained parameters are allocated corresponding optimizer states.

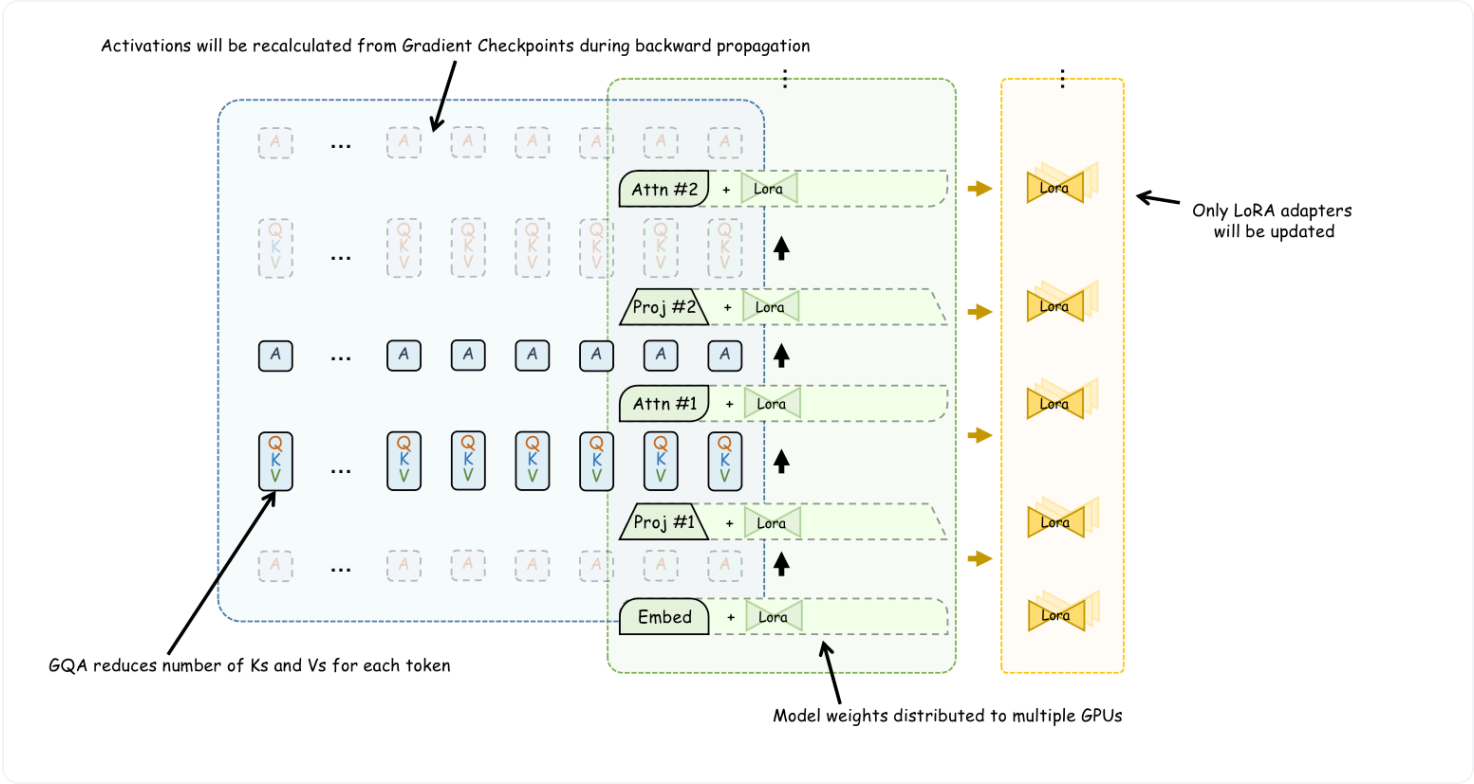
Through LoRA, almost all the memory usage brought by the optimizer state is eliminated. However, it should be noted that the main memory bottleneck of long-text training lies in the intermediate calculation results of the tokens. The advantage brought by LoRA is independent of the sequence length.

Distributed Training

Distributed training can unlock the training of extremely large models by using multiple GPUs. However, it brings much less advantage in long-text training, because commonly used fine-tuning frameworks, including FSDP and Deepspeed, are based on data parallelism, which means that each GPU trains independently. They only offload gradients, optimizer states, and model weights to multiple GPUs or even memory, and aggregate them when needed.

But as mentioned before, the memory burden brought by the optimizer and gradients has basically been removed by LoRA. The real memory load is concentrated on the intermediate calculation results corresponding to each token. However, distributed computing based on data parallelism requires at least one sample on each card, which means they will not share the memory of a single sample across multiple cards.

Therefore, using these distributed methods can significantly speed up training by increasing parallelism, but at the same time, it can only reduce some memory usage (by distributing model parameters to multiple cards).



In summary, after our painstaking efforts, we have minimized memory usage as much as possible. Referring to the image above. Through testing, such optimization, combined with Flash-attention, allows us to perform fine-tuning with a batch size of 1 and a length of 64K on Yi-9B-200K in a **float16 precision, 80GB A100 environment**.

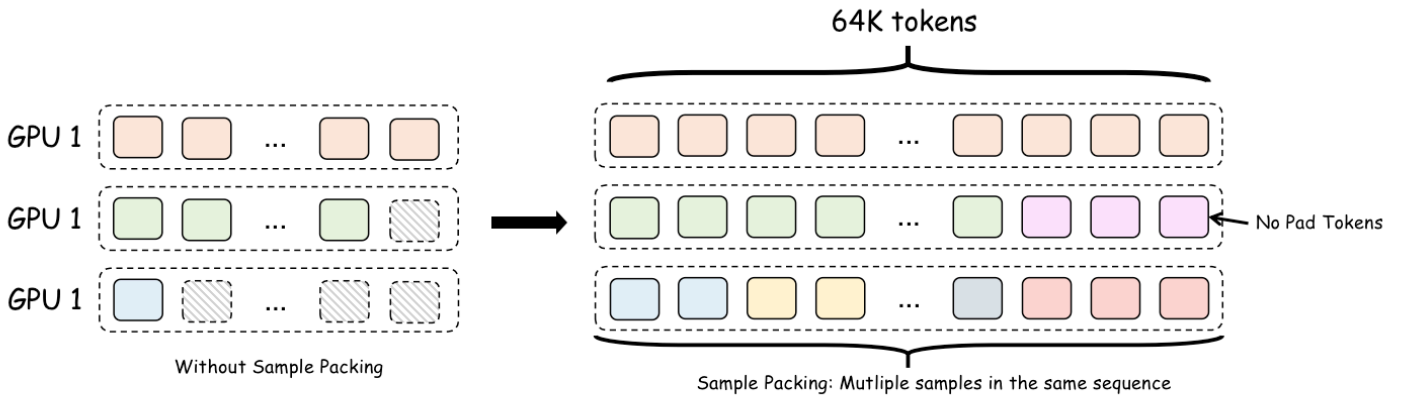
Batch Alignment

Unlike general fine-tuning data, where lengths are concentrated between 200-500 tokens, training samples for long-text fine-tuning may span several orders of magnitude in length.

In long-text training, it is easy to have a 4K sample and a 64K sample appear in the same batch. In this case, a large number of padding tokens will be added after the 4K sample to align the samples, causing a lot of waste.

In the default setting, short samples will be padded with padding tokens to the length of the longest sample in the batch. This means that a 4K sample may be padded to a length of 60K. Fortunately, most current fine-tuning frameworks can solve this problem through **Sample Packing** techniques, and we only need to enable the corresponding option.

Sample Packing actually removes the concept of batch size. A batch containing 3 samples is now concatenated into a single longer sequence. The three samples are connected head-to-tail into one sequence, and the corresponding attention mask changes to prevent different samples in the same sequence from influencing each other. The advantage of this is that there are no more padding tokens: an input may contain 2 long samples or 100 short samples.

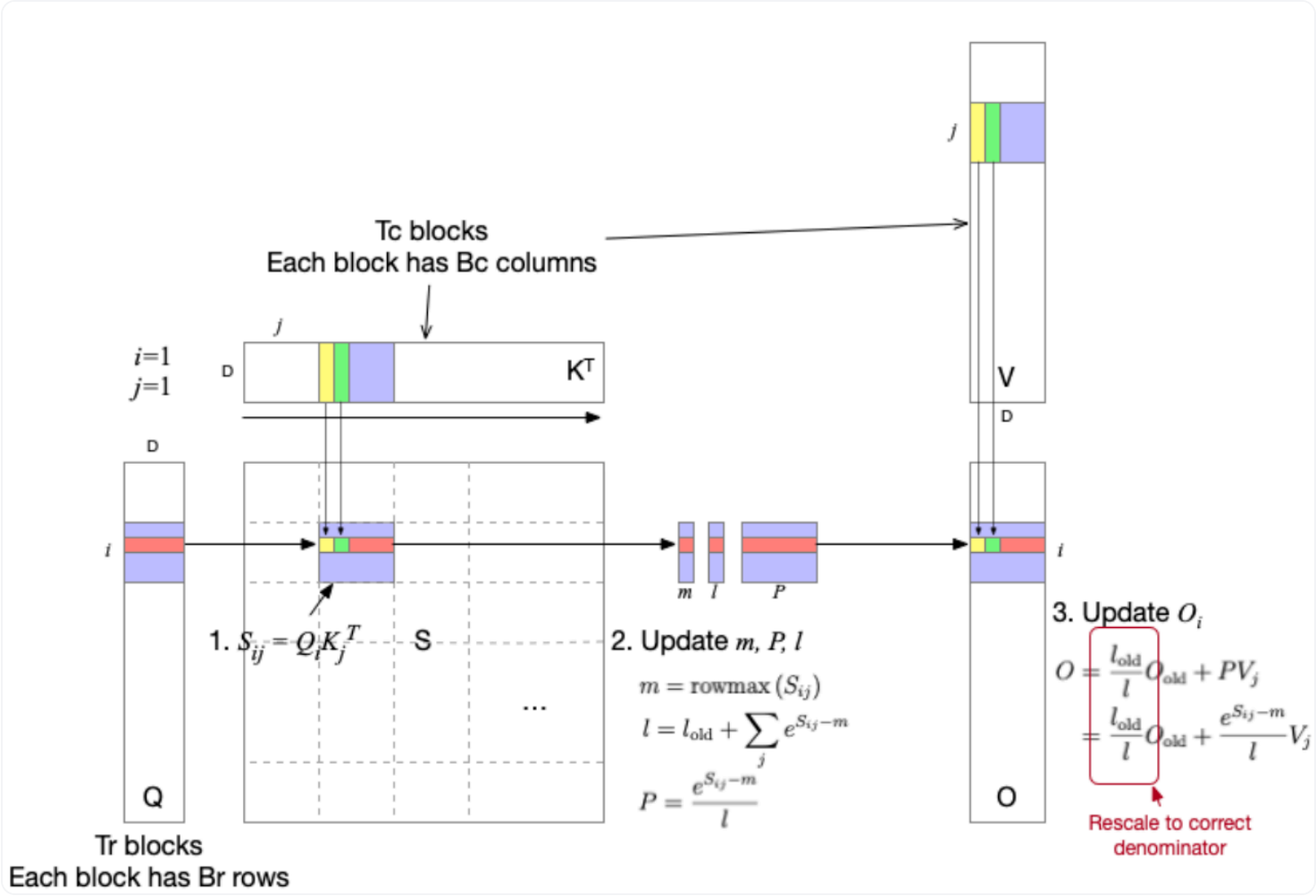


However, in practice, the LongAlign paper mentions that having long samples and extremely short samples in the same batch may affect model convergence. To solve this problem, samples of similar lengths are generally made to appear in the same batch during training. Common training frameworks also provide this option, which may be called `sort_by_length` or something similar.

Attention Space Complexity

In normal Attention computation, the interaction between Query and Key, Value involves N * N matrix multiplication. This makes long-text Attention computation have a space complexity of $O(N^2)$.

But this last problem is actually quite easy to solve: **just use Flash Attention**. Flash Attention designs specific CUDA operators for Attention computation. When updating each Query, other tokens' corresponding Queries and KVs that are not involved in the computation are not loaded. Therefore, the Attention computation of Flash Attention is more of $O(N)$ complexity.



Credit: https://insujang.github.io/2024-01-21/flash-attention

At this point, we have basically bridged the gap between long and short text fine-tuning: now training on a 32K-length sample is effectively like training on 32 1K samples. However, there are some problems that are difficult to solve, such as time complexity. Performing a forward pass requires calculating the representation at each token position, and the calculation of each token representation itself is close to $O(N)$. A long-context sample necessarily brings longer inference time compared to multiple short-context samples.

In actual fine-tuning, we can use any framework that implements the above features properly to perform the training, such as HF Trainer, Axolotl, and Llama Factory, which theoretically should all meet the requirements.

Practice: Faro Series Models

With this series of techniques, we can train our own long-text models with not very exaggerated resources (a few A100s). So I first trained some models myself. I named this series of models **Faro** and trained multiple versions, respectively from Qwen1.8B, Qwen4B, Yi-9B-200K, and Yi-34B-200K. You can download my models on Huggingface, and I also provide all the training configuration scripts and Wandb tracking records for those who are interested to follow.

- Models: Faro-Yi-9B Faro-Yi-34B Faro-Yi-9B-DPO
- Wandb training records: Faro-SFT Faro-DPO
- Axolotl training configuration files: SFT.yml DPO.yml

Long Text Data

There are not many open-source long-context SFT datasets. In order to train long-context models, I used the datasets open-sourced by LongAlign and LongLora, and I also synthesized some myself.

- THUDM/LongAlign-10k LongAlign contains 10K long-text task samples, of which 10% are in Chinese.
- Yukang/LongAlpaca-12k LongAlpaca contains 12K long-text tasks, mainly about reading comprehension of papers, and also mixes in some short data for balance.
- wenbopan/RefGPT-Fact-v2-8x The data I synthesized myself. Mutonix/RefGPT-Fact-v2 is a high-quality conversational dataset involving document extraction and understanding, but its length is a bit short, so I expanded it.
- wenbopan/anti-haystack A collection of long-text tasks generated using GPT-4. These tasks are mostly more symbolic in nature and generally involve precise recall of facts and referencing of paragraphs.

The above add up to about 40,000 pieces of data. I also added some short samples for balance, and at the same time, to maintain the model's Chinese ability, I controlled about 10% of the samples to be in Chinese. wenbopan/Fusang-v1 is the final dataset obtained, and its `long` branch is constructed according to the above method. Most of these samples are within 20K in length, so my actual training limits the model's maximum length to 24K. However, in practice, this kind of training can also significantly enhance the model's modeling ability on even longer texts.

Training

Training only needs to be properly configured according to the methods mentioned in this blog post. I use the Axolotl framework for training. The biggest advantage of this framework is that its training is highly configurable, and all the options required for training can be defined through a configuration file. The training of the Faro series models includes SFT and DPO. Only SFT is performed on long text. As for the methods of DPO, you can refer to my Huggingface repository and training script.

At the same time, I also provide Wandb tracking records of the entire training process Faro-SFT Faro-DPO for reference. Since different training runs may used different numbers of GPUs, you will see Loss curves of different lengths on Wandb.

Evaluation

Of course, after completing the training, we need to test how well our model performs on long-text modeling. Here, I chose LongBench. We can see that our long-text fine-tuning is quite effective: Faro-Yi-9B outperforms Yi-9B-200K in most aspects.

Name	Few-shot Learning_en	Synthetic Tasks_en	Single-Doc QA_en	Multi-Doc QA_en	Summarization_en	Few-shot Learning_zh	Synthetic Tasks_zh	Singl Doc QA_z
Yi-9B-200K	60.6	22.8	30.9	38.9	25.8	46.5	28.0	49.6
Faro-Yi-9B	63.8	40.2	36.2	38.0	26.3	30.0	75.1	55.6

What's Next

At this point, I am quite satisfied with our long-text fine-tuning. But if we want to continue to improve, there is still a lot that can be done. This long-text fine-tuning method is actually full of compromises.

In order to run long text on a single card, we have to use LoRA, and the models that can be fine-tuned are also limited to GQA models. For 13B models, the longest length that can be fine-tuned is about 32K, and for 8B, it is 64K. Even longer lengths touch the ceiling of our method.

To train on even longer texts, we need to use training methods based on **tensor parallelism** instead of data parallelism, such as MegatronLM and Jax. However, for the usage scenarios of individual researchers, our method can already simply and quickly produce useful long-text models.

👏 Community

mseeger Feb 18 • edited Feb 18

A simple way to save memory is activation "checkpointing". Instead of storing ALL intermediate results of the forward pass, you just store them once at the end of each transformer layer. You then recompute all missing parts during the backward pass.

❤️ 3 +

Reply

Edit | Preview

Start discussing this article

Upload images, audio, and videos by dragging in the text input, pasting, or clicking here.

Comment