Lora

Introduction

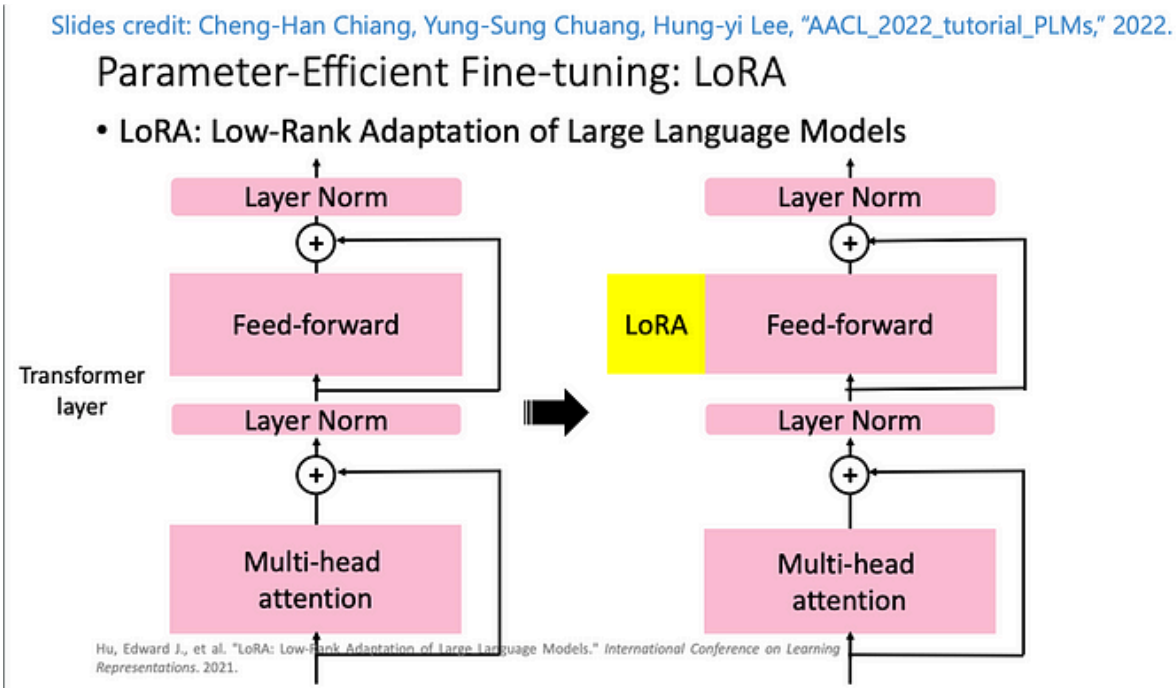
In recent years, Large Language Models (LLMs), also known as Foundational Models, have been trained using large datasets and models with a massive number of parameters, such as the common GPT-3 (175B parameters). The emergence of ChatGPT also indicates the generalization level of LLMs, as they have performed well in common problems.

- **However, when it comes to specific domains, although in-context learning can be achieved through a few examples (few-shot), fine-tuning the model would yield better results.**
- **All dimensions finetuning takes up a lot of memory.**

Model Architecture

As models have grown increasingly larger, directly fine-tuning all parameters incurs significant costs. Therefore, in recent years, researchers have focused on efficient fine-tuning, known as Parameter-Efficient Fine-Tuning (PEFT). This article will introduce Low-Rank Adaptation (LoRA) proposed by the Microsoft team, which involves freezing the weights of the pre-trained model (e.g., GPT-3) and fine-tuning it with a small model, achieving excellent fine-tuning results, similar to the Adapter concept. The idea is to use the small LoRA network inserted into specific layers to make the model adaptable to different tasks.

In addition, Cheng-Han Chiang, Yung-Sung Chuang, Hung-yi Lee, “AACL_2022_tutorial_PLMs,” 2022, provides a detailed explanation in the tutorial, as shown in the following figure.



In the future, instead of fine-tuning the parameters of a large neural network model, the approach may shift towards training a smaller model or weight, and combining it with the specific layer weights of the original LLM. Compared to fine-tuning the GPT-3 model, this method requires 10,000 times fewer training parameters and only 1/3 of GPU usage. **This technique is not only applied to LLMs, but also extensively used in training high-resolution image-generating AIs, such as the Stable-Diffusion generative model.**

Saving Memory

The comparison of memory usage between using and not using LoRA technology:
As shown in the figure below, **the LoRA model only requires a small number of parameters, with 0.5M and 11M parameters, which is much smaller than the original LLM model** (here using GPT-2 Medium with 345M parameters). Moreover, using LoRA technology, the inference efficiency is better than the previous Adapter technology in the case of Batch size = 1.

Note: GPT-2 Medium parameters: 345M

Batch Size Sequence Length	32 512	16 256	1 128
$\|\Theta\|$	0.5M	11M	11M
Fine-Tune/LoRA	1449.4±0.8	338.0±0.6	19.8±2.7
$\text{Adapter}^{\text{L}}$	1482.0±1.0 (+2.2%)	354.8±0.5 (+5.0%)	23.9±2.1 (+20.7%)
$\text{Adapter}^{\text{H}}$	1492.2±1.0 (+3.0%)	366.3±0.5 (+8.4%)	25.8±2.2 (+30.3%)

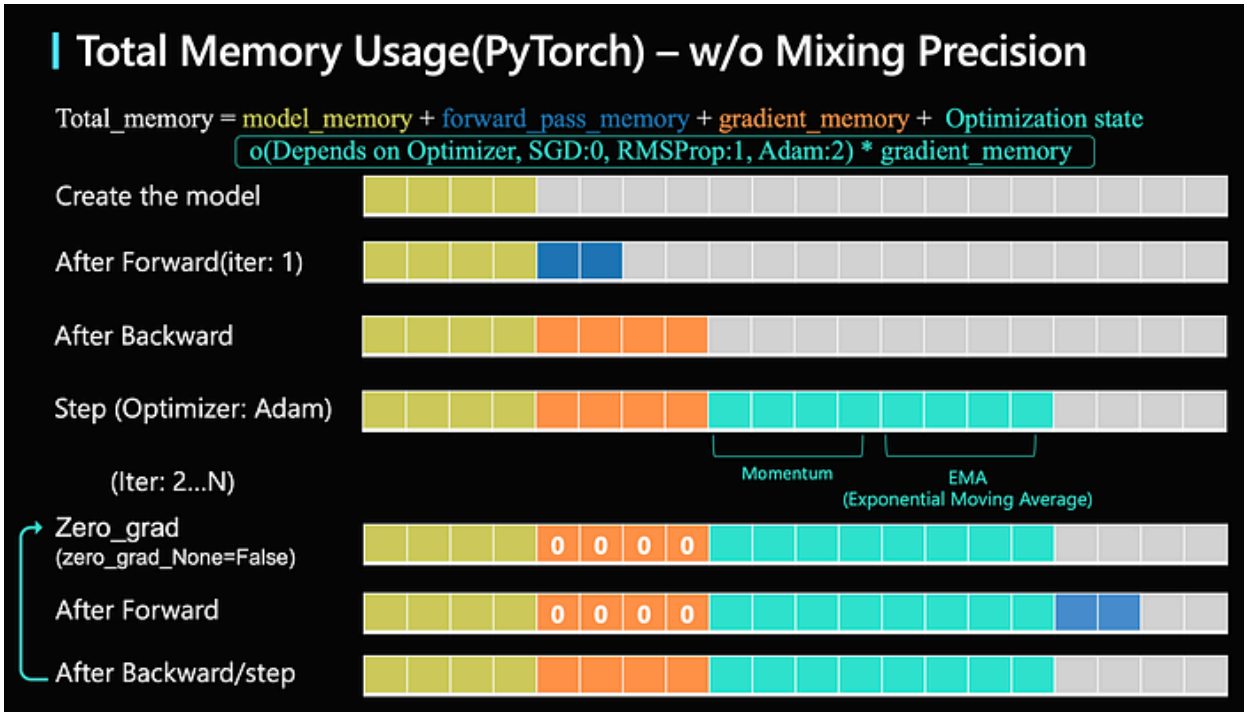
Table 1: Infernece latency of a single forward pass in GPT-2 medium measured in milliseconds, averaged over 100 trials. We use an NVIDIA Quadro RTX8000. "$|\Theta|$" denotes the number of trainable parameters in adapter layers. $\text{Adapter}^{\text{L}}$ and $\text{Adapter}^{\text{H}}$ are two variants of adapter tuning, which we describe in Section 5.1. The inference latency introduced by adapter layers can be significant in an online, short-sequence-length scenario. See the full study in Appendix B.

Why is the parameter/size of a model so important? First, it is important to understand how much GPU memory will be used during model training. For details, please refer to [Jacob Stern's comprehensive guide to memory usage in PyTorch].

The maximum model usage during training (without considering mixing precision) is calculated as follows:

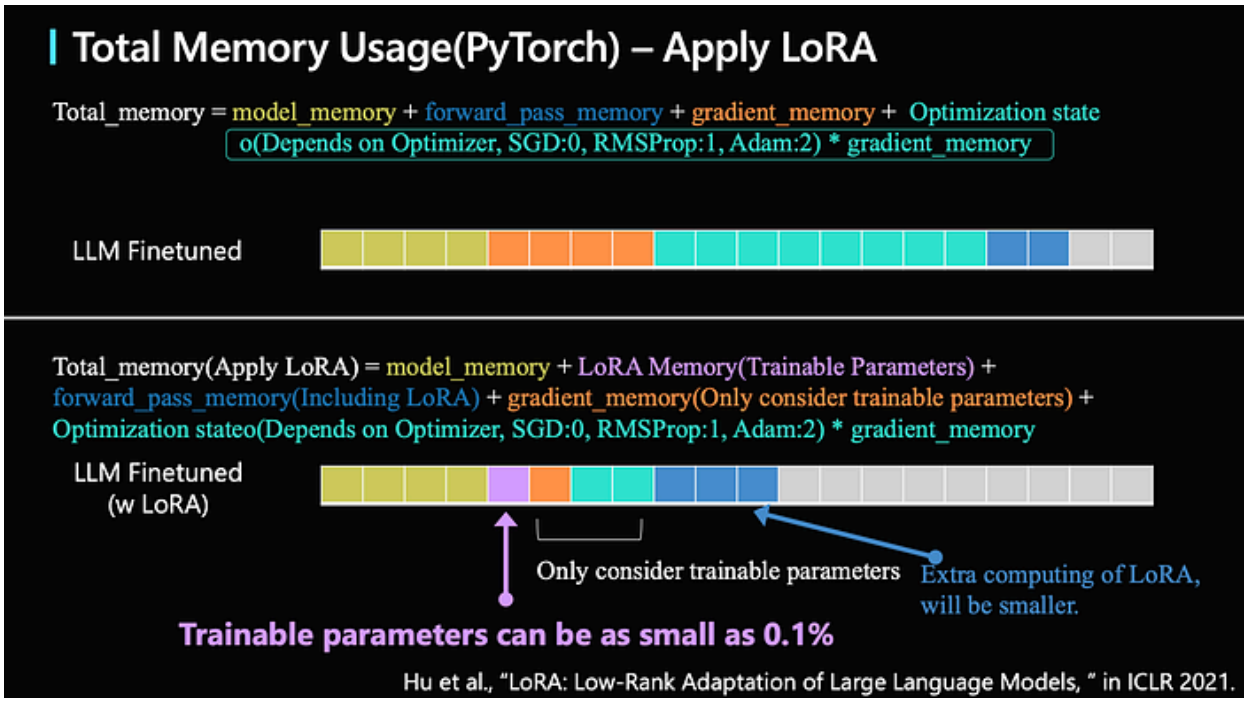
- Formula: Memory required by the **model** + Memory required for **forward calculation** (more flexible) + Memory required for **gradients** (memory required for model training parameters) + Memory required for **optimizer variables** * Memory required for model **training parameters** (usually the largest, Adam can be considered as requiring twice the memory required by the model)
- The forward calculation depends on the **batch size,** the size of the **input content,** and whether or not to use **mixing precision**. The memory consumption of this part can be reduced by using PyTorch's checkpoint mechanism, which is flexible and adjustable.
- The memory required for optimizer variables depends on the different optimizers (SGD: 0, RMSProp: 1, Adam: 2). The common optimizer, Adam, records the EMA and Momentum of the model's previous gradients, so for Adam optimizer, it will store 2 times the size of the model parameters!

The following is an estimate using the Adam optimizer and no mixing precision (corrections are welcome if there are any errors!):



Assuming that the memory consumption of a large model occupies 4 slots, to train the entire model, the Gradient also requires a memory of 4 slots of the model size, and in order to train the model, an optimizer (Adam) is required, which also requires 2 times of memory for the model size, thus occupying 8 slots. This does not yet include the memory required for Forward, which requires 4 times of memory for the model size. It can be imagined that for an extremely large model like GPT-3, with a model size of 175B, the required memory is enormous!!

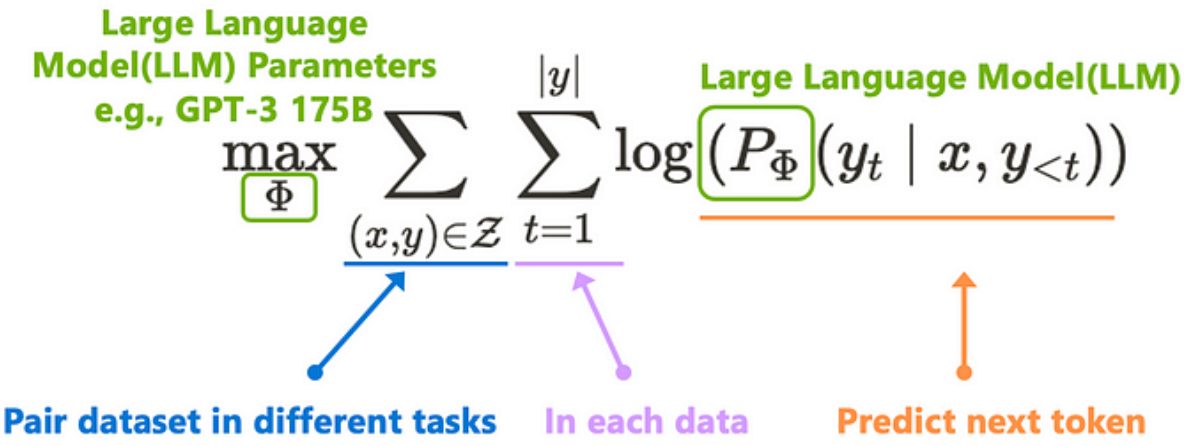
Although in practice, mixing precision and some techniques will be used, the memory size required for training depends on the number of parameters to be trained. In fact, when fine-tuning large models, we usually use adapters to train part of the parameters, rather than training the entire model. By freezing the LLM model weights, we only need to train a small part of the extended model, in which case our optimizer and the number of parameters required for backward will instantly be reduced a lot. We use the following example to show that if we only use the Θ quantity required by the LoRA model (usually, the trained parameters will be < 0.1% of LLM), the memory usage at this time will become the following figure. Of course, in reality, it is even less, because the added trainable parameters will be very few, and the additional computational cost will be very low:



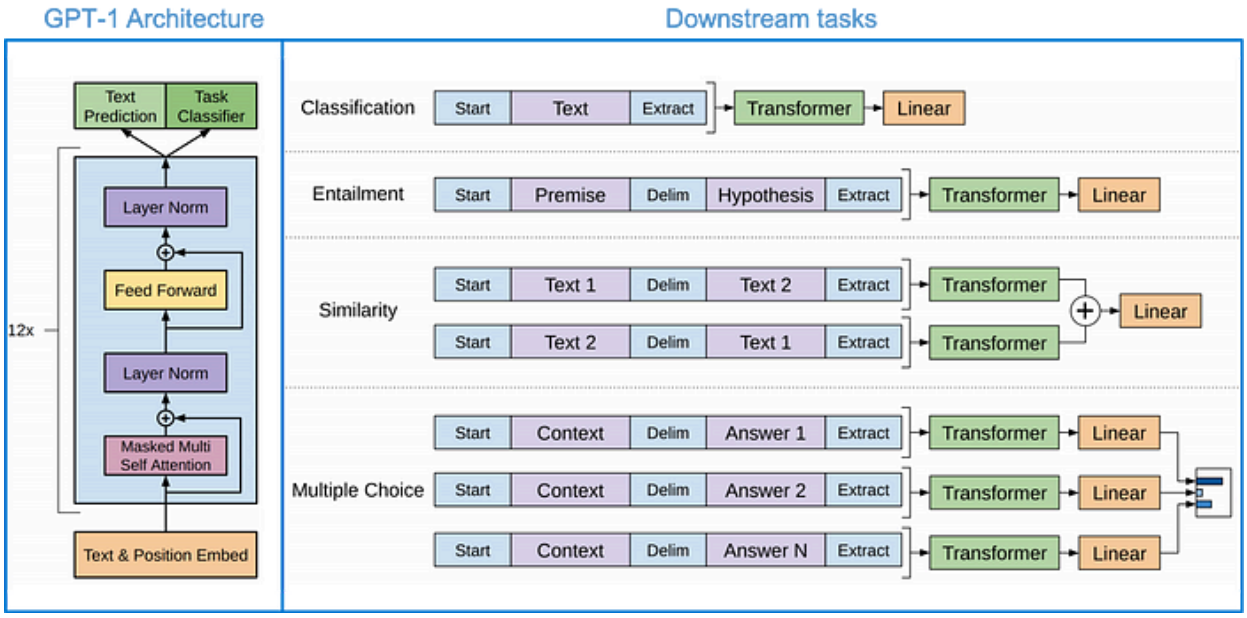
Better Performance

Method — LoRA(Low-Rank Adaptation)

In the past, make LLM or Foundation Models (such as the GPT series) applicable to various downstream tasks, the goal of training the model (Φ) was to ensure that the model performs well in handling multiple different tasks (Z).



The following figure shows the downstream tasks used for GPT-1, which include common NLP tasks such as classification, hypothesis testing, similarity comparison, and multiple-choice questions. The model is trained by providing different prompts as input.



Radford et al., “Improving Language Understanding by Generative Pre-Training”, in 2018.
In the past, there have been two Parameter-Efficient Fine-Tuning approaches for different downstream tasks:

1. **Adapter: By adding a small amount of model architecture and freezing the LLM model parameters, training is performed.**
2. Prefixing: Adding tokens to the beginning of the prompt to allow the model to perform better for specific tasks.

The LoRA introduced in this article belongs to the Adapter type. The concept of LoRA is that since LLM is applicable to different tasks, the model will have different neurons/features to handle different tasks. If we can find the features that are suitable for the downstream task from many features and enhance their features, we can achieve better results for specific tasks.
Therefore, by combining the LLM model — Φ with another set of trainable parameters Trainable Weight — Θ(Rank decomposition matrices), downstream task results can be optimized.

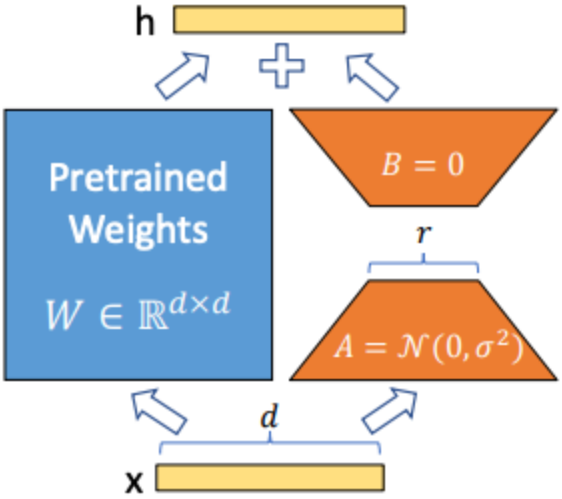
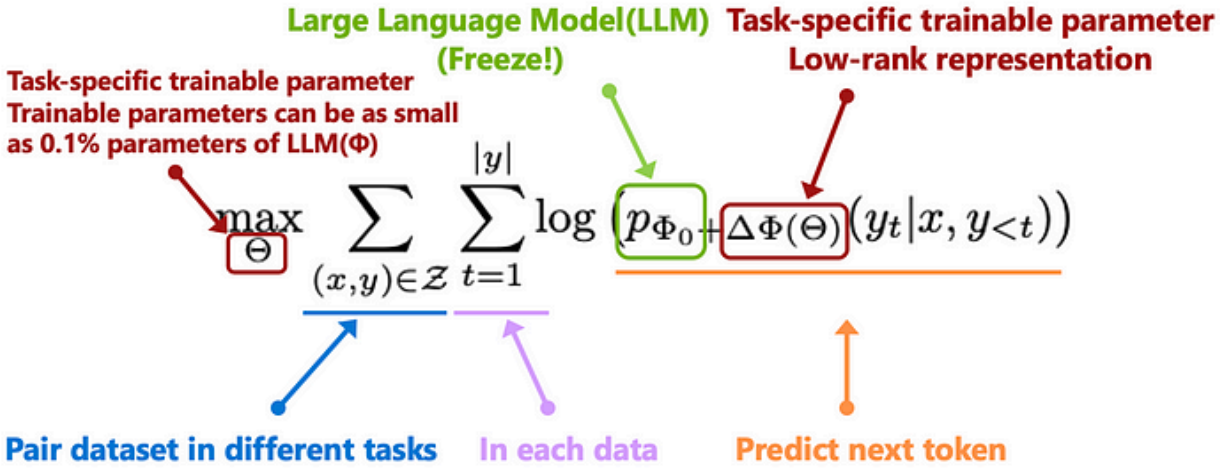


Figure 1: Our reparametrization. We only train A and B.

The orange module on the right represents the LoRA model weights that we want to train. By restricting the rank(r) to smaller in the middle, the number of trainable parameters can be significantly reduced, and the dimensionality of the features can be reduced to "r << d". The overall number of parameters can then be expressed as "|Θ|=2×LoRA ×dmodel ×r". LoRA is the number of LoRA modules used in the entire model, and in the paper, LoRA modules were inserted into the Attention layer of the Transformer architecture. The value of "r" varies depending on the task, but in experiments, a value of 2~4 has been shown to yield good results. Ultimately, we want to optimize the downstream tasks through the LoRA modules, as shown in the formula below.

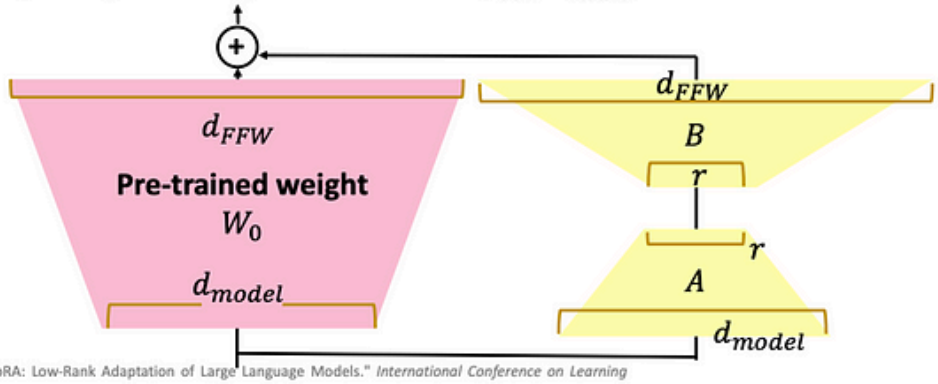


Wrap Up

Slides credit: Cheng-Han Chiang, Yung-Sung Chuang, Hung-yi Lee, "AACL_2022_tutorial_PLMs," 2022.

Parameter-Efficient Fine-tuning: LoRA

- **Low-Rank Adaptation** of Large Language Models
- Motivation: Downstream fine-tunings have low intrinsic dimension
- Weight after fine-tuning = W_0 (pre-trained weight) $+\Delta W$ (updates to the weight)
- Hypothesis: The updates to the weight (ΔW) also gave a low intrinsic rank
- Fine-tuned weight = $W_0 + \Delta W = W_0 + BA$, rank $r \ll \min(d_{FFW}, d_{model})$



Hu, Edward J., et al. "LoRA: Low-Rank Adaptation of Large Language Models." *International Conference on Learning Representations*. 2021.

Experiments Evaluation

LoRA achieved better results than Fine-tuning, and required much fewer parameters to train.

Model & Method	# Trainable Parameters	MNLI	SST-2	MRPC	CoLA	QNLI	QQP	RTE	STS-B	Avg.
RoB_{base} (FT)*	125.0M	**87.6**	94.8	90.2	**63.6**	92.8	**91.9**	78.7	91.2	86.4
RoB_{base} (BitFit)*	0.1M	84.7	93.7	**92.7**	62.0	91.8	84.0	81.5	90.8	85.2
RoB_{base} (Adpt^{D})*	0.3M	$87.1_{\pm.0}$	$94.2_{\pm.1}$	$88.5_{\pm1.1}$	$60.8_{\pm.4}$	$93.1_{\pm.1}$	$90.2_{\pm.0}$	$71.5_{\pm2.7}$	$89.7_{\pm.3}$	84.4
RoB_{base} (Adpt^{D})*	0.9M	$87.3_{\pm.1}$	$94.7_{\pm.3}$	$88.4_{\pm.1}$	$62.6_{\pm.9}$	$93.0_{\pm.2}$	$90.6_{\pm.0}$	$75.9_{\pm2.2}$	$90.3_{\pm.1}$	85.4
RoB_{base} (LoRA)	0.3M	$87.5_{\pm.3}$	$\mathbf{95.1}_{\pm.2}$	$89.7_{\pm.7}$	$63.4_{\pm1.2}$	$\mathbf{93.3}_{\pm.3}$	$90.8_{\pm.1}$	$\mathbf{86.6}_{\pm.7}$	$\mathbf{91.5}_{\pm.2}$	**87.2**
$\text{RoB}_{\text{large}}$ (FT)*	355.0M	90.2	**96.4**	**90.9**	68.0	94.7	**92.2**	86.6	92.4	88.9
$\text{RoB}_{\text{large}}$ (LoRA)	0.8M	$\mathbf{90.6}_{\pm.2}$	$96.2_{\pm.5}$	$\mathbf{90.9}_{\pm1.2}$	$\mathbf{68.2}_{\pm1.9}$	$\mathbf{94.9}_{\pm.3}$	$91.6_{\pm.1}$	$\mathbf{87.4}_{\pm2.5}$	$\mathbf{92.6}_{\pm.2}$	**89.0**
$\text{RoB}_{\text{large}}$ (Adpt^{P})†	3.0M	$90.2_{\pm.3}$	$96.1_{\pm.3}$	$90.2_{\pm.7}$	$\mathbf{68.3}_{\pm1.0}$	$\mathbf{94.8}_{\pm.2}$	$\mathbf{91.9}_{\pm.1}$	$83.8_{\pm2.9}$	$92.1_{\pm.7}$	88.4
$\text{RoB}_{\text{large}}$ (Adpt^{P})†	0.8M	$\mathbf{90.5}_{\pm.3}$	$\mathbf{96.6}_{\pm.2}$	$89.7_{\pm1.2}$	$67.8_{\pm2.5}$	$\mathbf{94.8}_{\pm.3}$	$91.7_{\pm.2}$	$80.1_{\pm2.9}$	$91.9_{\pm.4}$	87.9
$\text{RoB}_{\text{large}}$ (Adpt^{H})†	6.0M	$89.9_{\pm.5}$	$96.2_{\pm.3}$	$88.7_{\pm2.9}$	$66.5_{\pm4.4}$	$94.7_{\pm.2}$	$92.1_{\pm.1}$	$83.4_{\pm1.1}$	$91.0_{\pm1.7}$	87.8
$\text{RoB}_{\text{large}}$ (Adpt^{H})†	0.8M	$90.3_{\pm.3}$	$96.3_{\pm.5}$	$87.7_{\pm1.7}$	$66.3_{\pm2.0}$	$94.7_{\pm.2}$	$91.5_{\pm.1}$	$72.9_{\pm2.9}$	$91.5_{\pm.5}$	86.4
$\text{RoB}_{\text{large}}$ (LoRA)†	0.8M	$\mathbf{90.6}_{\pm.2}$	$96.2_{\pm.5}$	$\mathbf{90.2}_{\pm1.0}$	$68.2_{\pm1.9}$	$\mathbf{94.8}_{\pm.3}$	$91.6_{\pm.2}$	$\mathbf{85.2}_{\pm1.1}$	$\mathbf{92.3}_{\pm.5}$	**88.6**
DeB_{XXL} (FT)*	1500.0M	91.8	**97.2**	92.0	72.0	**96.0**	92.7	93.9	92.9	91.1
DeB_{XXL} (LoRA)	4.7M	$\mathbf{91.9}_{\pm.2}$	$96.9_{\pm.2}$	$\mathbf{92.6}_{\pm.6}$	$\mathbf{72.4}_{\pm1.1}$	$\mathbf{96.0}_{\pm.1}$	$\mathbf{92.9}_{\pm.1}$	$\mathbf{94.9}_{\pm.4}$	$\mathbf{93.0}_{\pm.2}$	**91.3**

Table 2: $\text{RoBERTa}_{\text{base}}$, $\text{RoBERTa}_{\text{large}}$, and $\text{DeBERTa}_{\text{XXL}}$ with different adaptation methods on the GLUE benchmark. We report the overall (matched and mismatched) accuracy for MNLI, Matthew's correlation for CoLA, Pearson correlation for STS-B, and accuracy for other tasks. Higher is better for all metrics. * indicates numbers published in prior works. † indicates runs configured in a setup similar to Houlsby et al. (2019) for a fair comparison.

Compared to other efficient Fine-tuning methods, LoRA achieved the best accuracy.

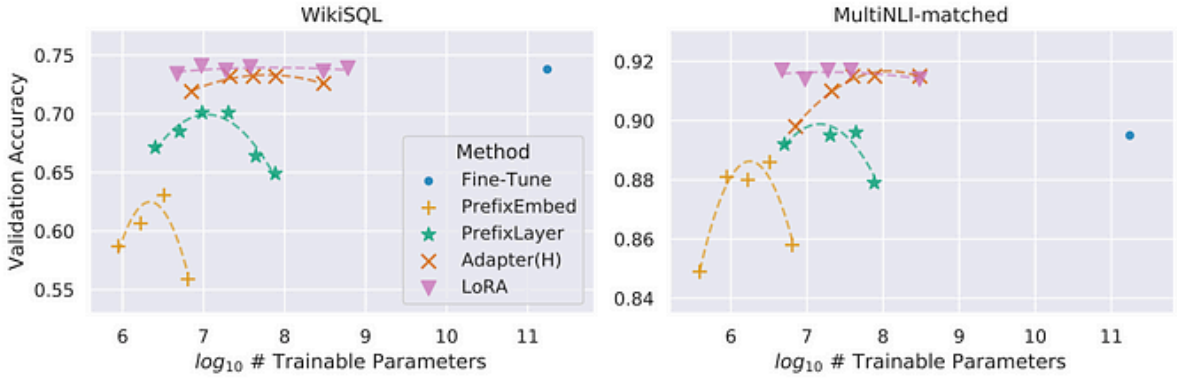


Figure 2: GPT-3 175B validation accuracy vs. number of trainable parameters of several adaptation methods on WikiSQL and MNLI-matched. LoRA exhibits better scalability and task performance. See Section F.2 for more details on the plotted data points.

The experiments only evaluated the performance of adding LoRA modules to the Attention block, and evaluated which block (Q, K, V, or O) achieved the best results while keeping the parameter count fixed.

	# of Trainable Parameters = 18M						
Weight Type	W_q	W_k	W_v	W_o	W_q, W_k	W_q, W_v	W_q, W_k, W_v, W_o
Rank r	8	8	8	8	4	4	2
WikiSQL (±0.5%)	70.4	70.0	73.0	73.2	71.4	**73.7**	**73.7**
MultiNLI (±0.1%)	91.0	90.8	91.0	91.3	91.3	91.3	**91.7**

Table 5: Validation accuracy on WikiSQL and MultiNLI after applying LoRA to different types of attention weights in GPT-3, given the same number of trainable parameters. Adapting both W_q and W_v gives the best performance overall. We find the standard deviation across random seeds to be consistent for a given dataset, which we report in the first column.

The choice of the number of Ranks was also investigated.

	Weight Type	$r = 1$	$r = 2$	$r = 4$	$r = 8$	$r = 64$
WikiSQL(±0.5%)	W_q	68.8	69.6	70.5	70.4	70.0
	W_q, W_v	73.4	73.3	73.7	73.8	73.5
	W_q, W_k, W_v, W_o	74.1	73.7	74.0	74.0	73.9
MultiNLI (±0.1%)	W_q	90.7	90.9	91.1	90.7	90.7
	W_q, W_v	91.3	91.4	91.3	91.6	91.4
	W_q, W_k, W_v, W_o	91.2	91.7	91.7	91.5	91.4

Table 6: Validation accuracy on WikiSQL and MultiNLI with different rank r. To our surprise, a rank as small as one suffices for adapting both W_q and W_v on these datasets while training W_q alone needs a larger r. We conduct a similar experiment on GPT-2 in Section H.2.

Application Benefits

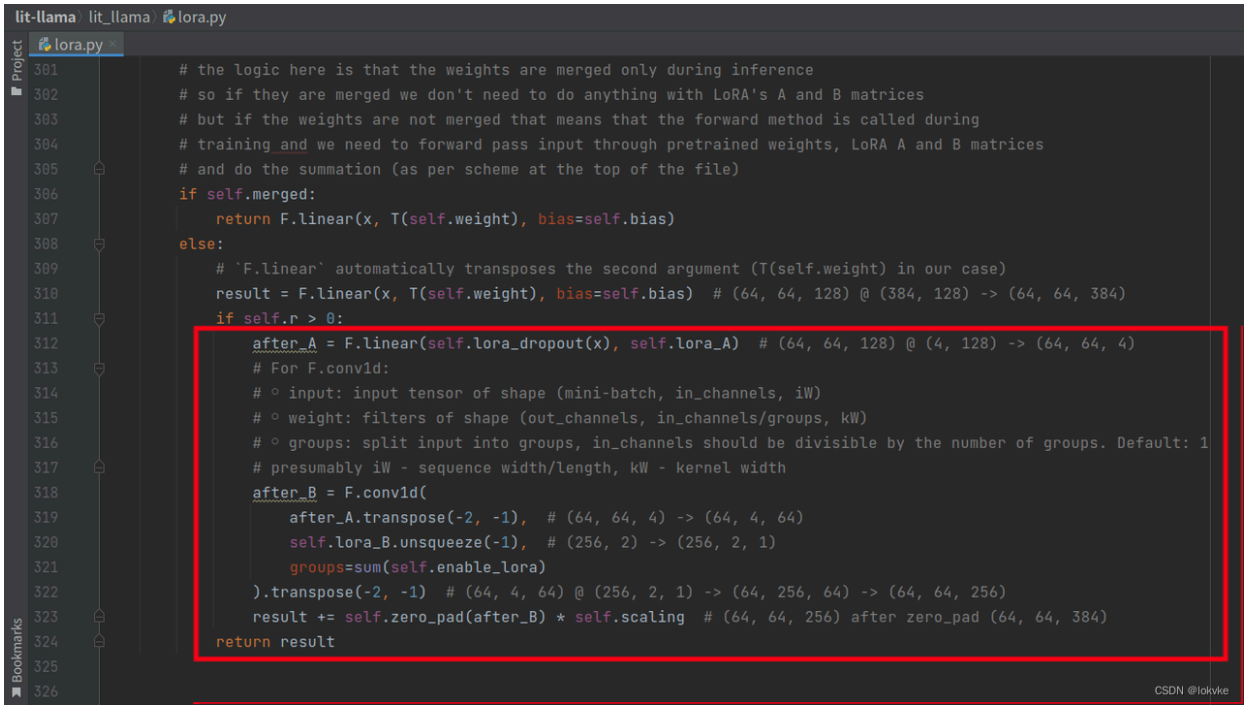
The **Benefits** of LoRA are plentiful as we can probably tell. However, some of the most notable benefits of this approach include the following:
A single pretrained model can be shared by several (much smaller) LoRA modules that adapt it to solve different tasks, which simplifies the deployment and hosting process.

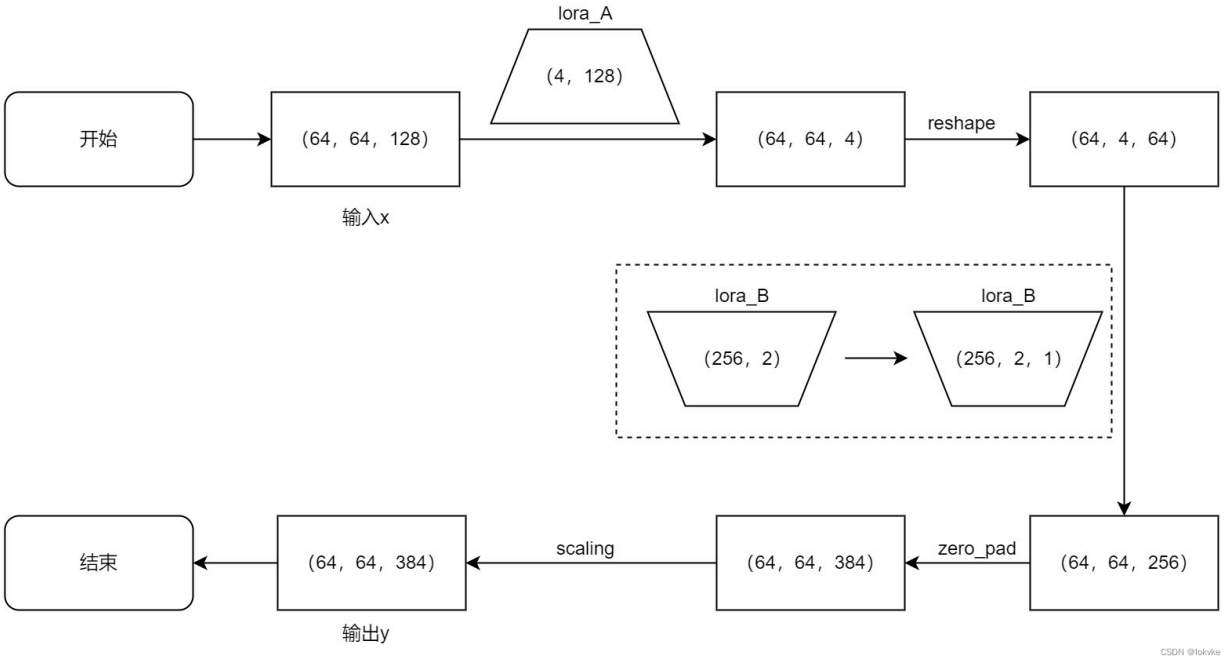
LoRA modules can be "baked in" to the weights of a pretrained model to avoid extra inference latency, and we can quickly switch between different LoRA modules to solve different tasks.

When finetuning an LLM with LoRA, we only have to maintain the optimizer state for a very small number of parameters 10, which significantly reduces memory overhead and allows finetuning to be performed with more modest hardware (i.e., smaller/fewer GPUs with less memory).

Finetuning with LoRA is significantly faster than end-to-end finetuning (i.e., roughly 25% faster in the case of GPT-3).

Code





QLORA

Background

一个重要的讨论点是 LoRA 在训练期间的内存要求，无论是在使用的适配器的数量和大小方面。由于 LoRA 的内存占用非常小，我们可以使用更多的适配器来提高性能，而不会显着增加使用的总内存。虽然 LoRA 被设计为一种参数高效的微调 (PEFT) 方法，但 LLM 微调的大部分内存占用来自激活梯度，而不是来自学习的 LoRA 参数。对于在 FLAN v2 上训练的 7B LLaMA 模型，批量大小为 1，LoRA 权重等效于原始模型权重中常用的 0.2%，而 RA 输入梯度的内存占用为 567 MB，而 LoRA 参数仅占 26 MB。通过梯度检查点，输入梯度平均减少到每个平均 18 MB，而 LoRA 被设计为一个序列，使它们比所有 LoRA 权重组合更多的内存密集。相比之下，4 位基础模型消耗了 5048 MB 的内存。这突出了梯度检查点很重要，但也表明积极减少 LoRA 参数的数量只会产生很小的内存好处。这意味着我们可以使用更多的适配器，而不会显着增加整体训练内存占用。如前所述，这对于恢复完整的 16 位精度性能至关重要。

Background

Due to its practical utility, the proposal of LoRA catalyzed the development of an entire

field of research devoted to parameter-efficient finetuning—and LoRA in particular—that is quite active. Within this section, we will (attempt to) overview most of the notable LoRA variants that have been recently proposed.

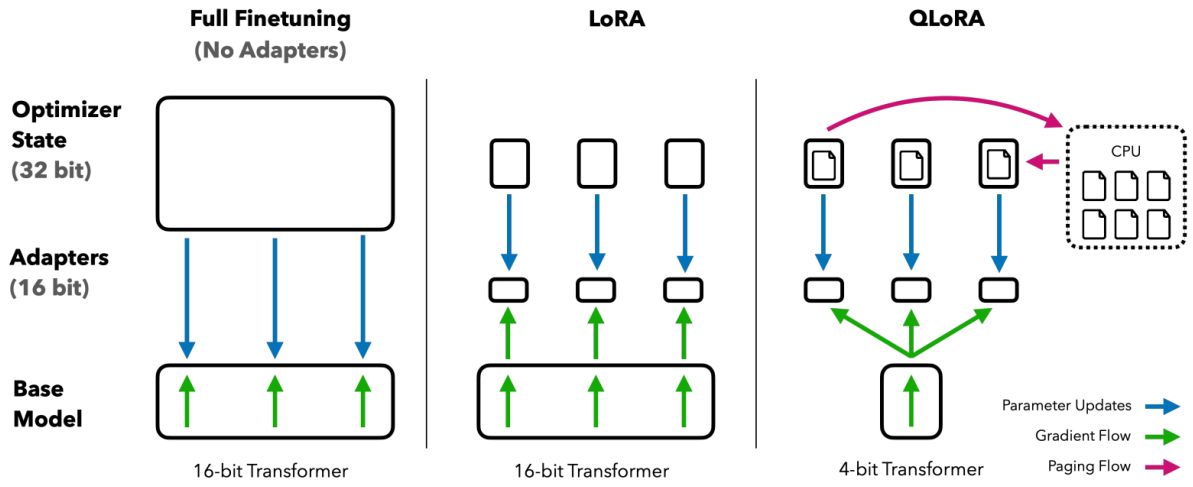


Figure 1: Different finetuning methods and their memory requirements. QLoRA improves over LoRA by quantizing the transformer model to 4-bit precision and using paged optimizers to handle memory spikes.

4-Bit NormalFloat (NF4) Format: a new (quantized) data type that works better for weights that follow a normal distribution.

4 位 Normalfloat，一种理论上最佳量化数据类型，该数据类型对正态分布数据产生比 4 位整数和 4 位 Float 更好的实证结果。这是一种改进量化的方法。它确保每个量化箱中的值数量相等。这避免了计算问题和异常值的错误。

将 32 位浮点 (FP32) 张量量化为范围为 int8 张量。

$$\mathbf{X}^{\text{Int8}} = \text{round}\left(\frac{127}{\text{absmax}(\mathbf{X}^{\text{FP32}})}\mathbf{X}^{\text{FP32}}\right) = \text{round}(c^{\text{FP32}} \cdot \mathbf{X}^{\text{FP32}}),$$

其中 c 是量化常数或量化尺度。去量化是可逆的：

$$\text{dequant}(c^{\text{FP32}}, \mathbf{X}^{\text{Int8}}) = \frac{\mathbf{X}^{\text{Int8}}}{c^{\text{FP32}}} = \mathbf{X}^{\text{FP32}}$$

Strength

Double Quantization: reduces memory footprint by quantizing both model weights and their corresponding quantization constants.
双量化，一种量化量化常数的方法，**QLoRa**的作者将其定义为：“对量化常数进行量化以节省更多内存的过程。”每个参数保存平均约 0.37 位（65B 模型大约 3 GB）。更具体地说，双量化将第一个量化的量化常数cFP322视为第二个量化的输入。第二步产生量化量化常数cFP82和第二级量化常数。我们使用块大小为 256 的 8 位 Floats 进行第二次量化，因为 8 位量化没有观察到性能下降，这与 Dettmers 和 Zettlemoyer 的结果一致。

Paged Optimizers: prevents memory spikes due to gradient checkpointing that cause out-of-memory errors when processing long sequences or training a large model.
使用 NVIDIA 统一内存来避免在处理具有长序列的小批量时发生的梯度检查点内存峰值。它确保**GPU**处理无误，特别是在**GPU**可能内存不足的情况下。

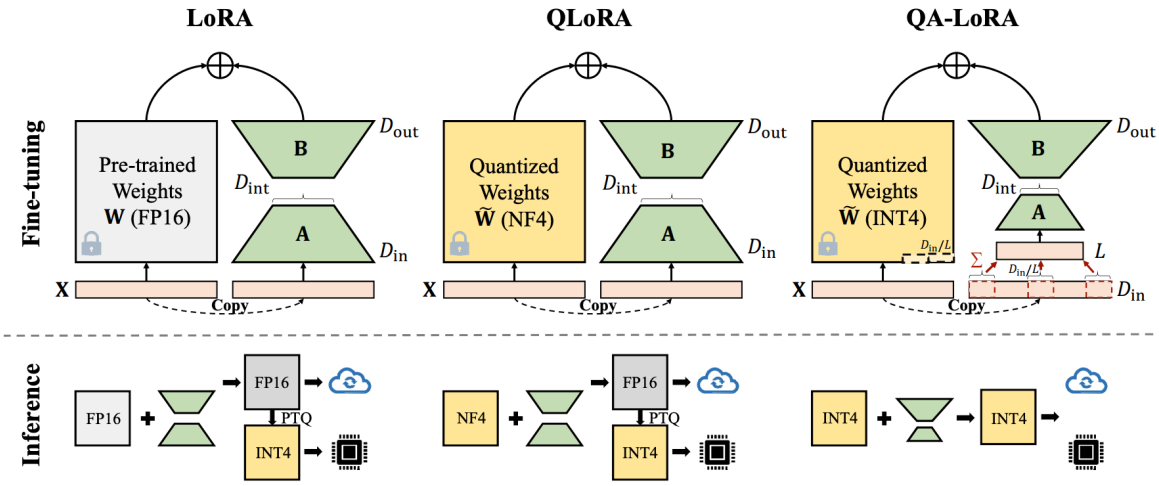


Figure 2: An illustration of the goal of QA-LoRA. Compared to prior adaptation methods, LoRA and QLoRA, our approach is computationally efficient in both the fine-tuning and inference stages. More importantly, it does not suffer an accuracy loss because post-training quantization is not required. We display INT4 quantization in the figure, but QA-LoRA is generalized to INT3 and INT2.

QA-Lora

让LLM更快更小
QA-LoRA旨在实现两个目标。首先，在微调阶段，预训练的权重W被量化为低位表示，使得LLMs可以在尽可能少的GPU上进行微调。其次，在微调阶段之后，经过微调和合并的权重W′仍然是量化形式，因此LLMs可以具有较高效率地被部署。

QLoRA仅仅考虑训练时候的资源，没有考虑推理。虽然QLoRA在训练过程中把模型量化，但是由于训练的LoRA参数是FP16类型的，在推理时，量化后的模型与LoRA参数融合，量化会被破坏，回到未量化的形式，那么如果想要高效推理，就必须再进行一步PTQ。但是我们都知道，PTQ多少都会引入误差。

LongLoRA

LongLoRA [21] attempts to cheaply adapt LLMs to longer context lengths using a parameter-efficient (LoRA-based) finetuning scheme; see above. Training LLMs with long context lengths is expensive because the cost of self-attention is quadratic with respect to the length of the input sequence. However, we can avoid some of this cost by i) starting with a pretrained LLM and ii) expanding its context length via finetuning. LongLoRA does exactly this, making the extension of a pretrained LLM' s context length via finetuning
cheaper by:
1. Using sparse local attention instead of dense global attention (optional at inference time).
2. Using LoRA (authors find that this works well for context extension). Put simply, LongLoRA is just an efficient finetuning technique that we can use to adapt a pretrained LLM to support longer context lengths.

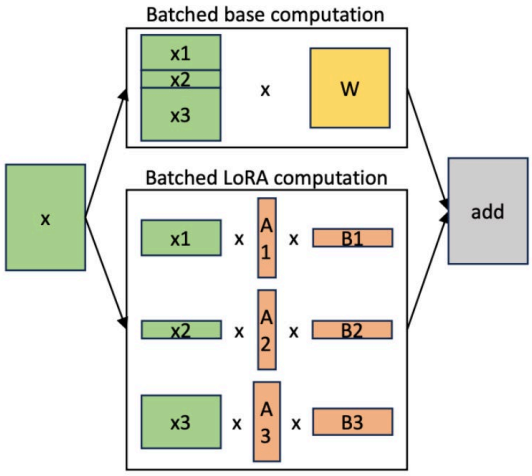


Figure 1. Separated batched computation for the base model and LoRA computation. The batched computation of the base model is implemented by GEMM. The batched computation for LoRA adapters is implemented by custom CUDA kernels which support batching various sequence lengths and adapter ranks.

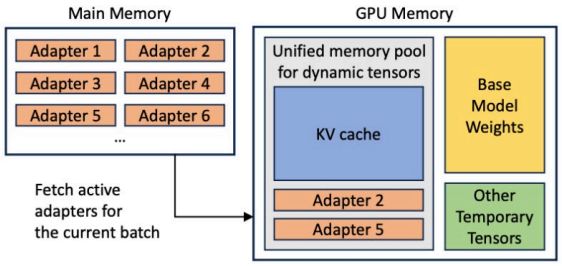


Figure 2. Overview of memory allocation in S-LoRA. S-LoRA stores all adapters in the main memory and fetches the active adapters for the current batch to the GPU memory. The GPU memory is used to store the KV cache, adapter weights, base model weights, and other temporary tensors.

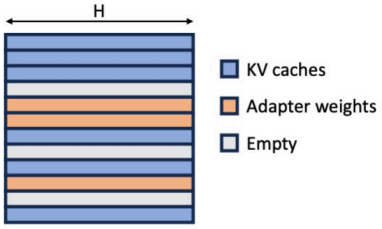


Figure 3. Unified memory pool. We use a unified memory pool to store both KV caches and adapter weights in a non-contiguous way to reduce memory fragmentation. The page size is H elements.