École Polytechnique Fédérale de Lausanne

Semester Project Report

ALoRA: A Mixed-Batch Multi-LoRA Serving System

presented by

Alexa Ray Lowe

Supervisors:

Prof. Dr. Rafael Pires

Milos Vujasinovic

Martijn de Vos

June 6, 2025

"The real voyage of discovery consists not in seeking new landscapes, but in having new eyes."

– MARCEL PROUST

Contents

Abstract

Fine-tuning large language models (LLMs) using Low-Rank Adaptation (LoRA) is a widely adopted and scalable technique for domain-specific adaptation. However, existing systems typically separate inference and fine-tuning workloads, and many do not support efficient multi-adapter operation on limited hardware such as a single GPU. This project introduces ALoRA, a system that enables concurrent fine-tuning and inference across multiple LoRA adapters that share a single base model. ALoRA supports mixed batches—where some samples are used for training and others for inference—making it well suited for interactive and online learning environments. Designed for efficient execution on a single GPU, ALoRA achieves up to a 4$\times$ throughput improvement over the PEFT implementation, without relying on model parallelism or complex scheduling. By employing a masking-based strategy for adapter computation and optimizing transitions between training and inference phases, ALoRA scales effectively with both batch size and the number of adapters, offering a promising direction for low-latency, multi-task adaptation.

Introduction

Large language models (LLMs) consisting of hundreds of billions of parameters such as Llama 3 [1] or Gemini [2] have become increasingly popular due to their powerful general capabilities. When fine-tuned, these models are highly effective for domain-specific applications. However, fine-tuning the full set of model parameters is often computationally expensive and inefficient.

To address the high resource demands of fine-tuning large language models (LLMs), parameter-efficient fine-tuning (PEFT) techniques have emerged. PEFT enables the adaptation of LLMs by training a much smaller set of parameters than the size of the full model weights—while maintaining high task performance [3]. Among these techniques, Low-Rank Adaptation (LoRA) [4] has become one of the most widely adopted, with support from many major LLM serving platforms [5]. Rather than updating the full model weights, LoRA injects two small, trainable low-rank matrices into selected layers. This significantly reduces both memory and computational overhead.

As fine-tuned LLMs are increasingly deployed in interactive, multi-task serving environments [6, 7], there is a growing need for systems that can dynamically update adapters and efficiently switch between them to handle unique incoming requests.

Supporting multiple LoRA adapters on a shared base model opens the door for efficient serving strategies that leverage shared compute resources for batched inference and fine-tuning. Existing systems introduce two types of optimizations for such scenarios: scheduling or batching techniques for GPU cluster utilization, and low-level memory and compute optimizations. Some scheduling techniques include: token-level fine-tuning in FlexLLM [8], iteration-level scheduling combined with selective-batching in Orca [9], and dynamic adapter merging and unmerging in dLoRA [10]. Memory optimizations introduced for batching operations on different LoRA adapters include: the Segmented Gather Matrix-Vector Multiplication (SGMV) custom CUDA kernel in Punica [11], and Unified Paging for efficient storage of weights and Key-Value (KV) caches in S-LoRA [12].

The listed multi-LoRA systems, except for FlexLLM, are limited to inference only. Although FlexLLM supports training and inference, it is designed for computation on a cluster of GPUs.

This leaves a gap for a simple, single-GPU solution that supports both inference and fine-tuning with multiple adapters. PEFT, although widely used, currently supports batching for multiple adapter workloads only during inference, not training.

This project presents ALoRA—a system designed to enable fine-tuning and inference across many LoRA adapters using a single shared base model. ALoRA's performance through experiments demonstrate how it simplifies multi-adapter workflows with tests on the GPT2 base model.

Contributions:

- ALoRA supports batched inputs for both training and inference.
- It uses a broadcast masking technique to maximize GPU throughput.
- It eliminates re-computation of the key-value cache when switching from training to inference.
- It handles gradient accumulation and per-adapter optimizers, allowing adapters to be fine-tuned as if trained in isolation.

Preliminaries

3.1 Next Token Prediction with Auto-Regressive Generation

Large language models (LLMs) represent and process text as sequences of tokens, where each token is a unique string of characters. On average, a token corresponds to approximately three-quarters of an English word [13]. Most state-of-the-art LLMs are based on the Transformer architecture, which introduced the now-fundamental mechanism of self-attention. Self-attention computes contextualized representations of tokens by multiplying token embeddings with learned query(Q), key(K), and value(V) matrices. These matrices enable the model to weigh the relevance of each token relative to other tokens in the same sequence. Using the attention mask M, tokens can attend to a more specific subset, such as only to tokens in earlier positions [14].

$$\text{MaskedAttention}(Q, K, V) = \text{softmax}\left(QK^T + M\right)V \tag{3.1}$$

The basic transformer consists of two primary components: the encoder and the decoder. The encoder processes the entire input sequence with self-attention to produce a representation of the text. The decoder processes the input text with masked attention, so each token only attends to tokens that come before it in a sequence, and an output sequence is generated one token at a time. The architectures of prominent LLMs today are encoder-only (BERT), decoder-only (Gemini, Llama, GPT), or encoder-decoder (T5) [15, 2, 16, 17].

The inference phase, known as the decoding or auto-regressive generation, puts output tokens one-by-one back into the decoder to generate a sequence. First, the model computes a forward-pass (the prefill stage) to encode the input, then the model decodes tokens one at a time. To improve efficiency, the key (K) and value (V) tensors are cached, avoiding recomputation of the entire attention mechanism at every step. The equation below describes the values computed to produce the output at step t.

$$
\begin{aligned}
Q_t &= W_Q x_t \\
K_t &= W_K x_t \\
V_t &= W_V x_t \\
K^{(t)}_{\text{cache}} &= \left[K^{(t-1)}_{\text{cache}}; K_t\right] \\
V^{(t)}_{\text{cache}} &= \left[V^{(t-1)}_{\text{cache}}; V_t\right] \\
\text{Output}_t &= \text{softmax}\left(Q_t (K^{(t)}_{\text{cache}})^\top + M\right) V^{(t)}_{\text{cache}}
\end{aligned}
\tag{3.2}
$$

This work focuses on tests with the GPT2 model and, like all decoder models, it relies on KV caching to achieve efficient auto-regessive generation.

3.2 Low-Rank Adaptation (LoRA)

Low-Rank Adaptation is a parameter-efficient fine-tuning method that updates pairs of low-rank additive matrices, or adapters, to learn from the training samples. LoRA leverages the observation that "there exists a low-dimension reparameterization that is as effective for fine-tuning as the full parameter space" [18]. Thus, changes in a dense layer can be represented by a pair of low-rank matrixes A and B. The original weight matrix $W_0 \in \mathbb{R}^{d\times k}$ is updated to the fine-tuned matrix composed of $W_0 + \Delta W \approx W_0 + AB$ where $B \in \mathbb{R}^{d\times r}$ and $A \in \mathbb{R}^{r\times k}$ and rank $r << min(d, k)$. Only A and B are modified in the training process, and given an input x, the forward pass for a single modified dense layer comprises of:

$$h = W_0 + \Delta W x = W_0 x + BAx \tag{3.3}$$

In this project, one finetuning task uses unique LoRA adapters to modify each attention matrix in all 12 layers of GPT2.

3.3 Memory vs. Compute Tradeoff

The large size of mainstream LLMs, often on the scale of several billion to trillion parameters, results in significant memory and compute demands for performing the forward pass, or prefill stage, and the backward pass. In multi-LoRA setups, the base model weights are typically separated in memory from the smaller adapter modules, creating an opportunity to reduce both compute and memory overhead.

During the prefill stage, outputs from the shared base model layers can be batched, allowing the GPU's computational capacity to be fully utilized. The subsequent backward pass, which involves intensive gradient computation, also remains compute-bound. However, in the auto-regressive generation stage, only one token is processed at a time per sequence, and the KV cache must be stored—leading to low GPU utilization. As the output sequence lengthens, the workload shifts from compute-bound to memory-bound, particularly due to the growing size of the KV cache [11].

To maximize GPU utilization, it is important to store only the minimal set of tensors necessary for the task and to utilize available compute resources. This means minimizing unnecessary tensors stored during the training phase, and batching the computation of LoRA adapters during the auto-regressive phase.

3.4 System Requirements

In this project, a setting in which the base model and all associated LoRA adapters fit entirely within the memory of a single GPU is assumed. Techniques involving model parallelism—where computation is distributed across multiple GPUs—are not considered. Additionally, the system is designed independently of any request- or token-level scheduling strategies, focusing purely on the core mechanisms for batched training and inference.

3.5 Research Question

This work explores how to efficiently support simultaneous training and inference across multiple LoRA adapters in large language models using a unified system. In particular, we aim to address the following research questions:

- How can a single system efficiently process mixed batches containing both training and inference samples for multiple LoRA adapters sharing a common base model?
- Can we design a masking and batching mechanism that enables concurrent computation across adapters, without sacrificing performance or correctness?
- What are the performance trade-offs—such as throughput, latency, and memory usage—of enabling this mixed-mode execution compared to existing adapter-based fine-tuning frameworks like PEFT?

ALoRA

4.1 System Overview

ALoRA enables concurrent processing of training and inference samples across multiple LoRA adapters within a single batch. The system accumulates gradients and updates LoRA adapter weights as if each adapter were trained in isolation using only its designated training samples. Then, it performs auto-regressive generation for the samples marked for inference. Each LoRA adapter maintains its own optimizer, which is triggered once a defined number of training samples (specified by `samples_per_step`) have contributed gradients through the backward pass.

After ALoRA is given inputs, the process follows:

1. The prefill stage uses batched computation on the base model and the masking technique described in section 3.2.

2. The backward pass follows immediately afterward to accumulate gradients. If any adapter reaches its `samples_per_step` threshold, its optimizer is invoked to apply an update.

3. The inference stage is performed just for inference samples and served through auto-regressive generation using key-value (KV) caching.

As illustrated in Figure 3.1, the user specifies how many samples in each batch are used for training: these must occupy the first `num_training` positions in the batch. The user also supplies a masking parameter to assign each sample to a specific adapter or a weighted combination of adapters.

Section 4.2 describes the implementation of batched training and inference using this adapter-masking scheme. Section 4.3 then explores the memory and computational optimizations applied during the transition from the backward pass to the auto-regressive generation phase in mixed batches.

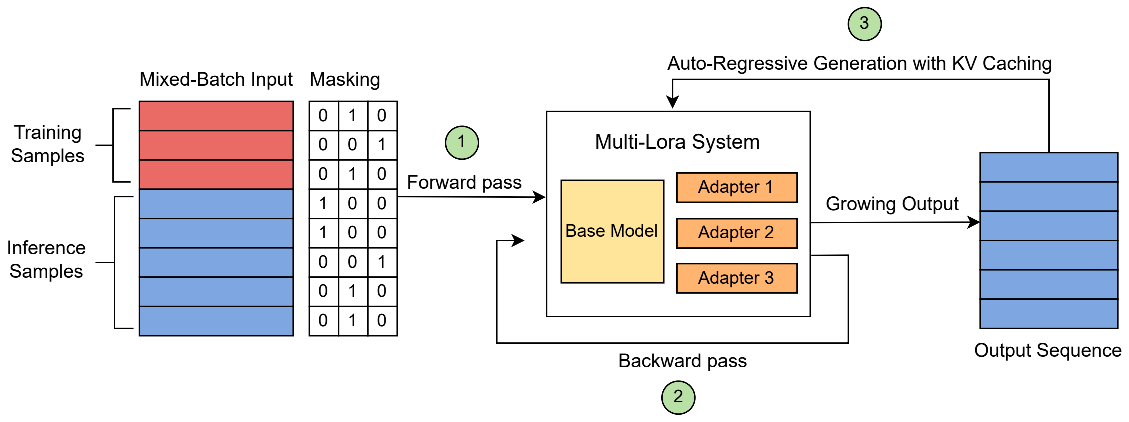


Figure 4.1: This high-level diagram shows the forward pass, backward pass, and auto-regressive generation phases of the model.

4.1.1 Usage

The code excerpt below demonstrates how to instantiate ALoRA. This implementation supports multiple LoRA adapters with a uniform configuration for simplicity: all adapters are instantiated with the same rank and scaling factor (α), use the same learning rate, and performs updates after a fixed number of training samples, defined by `samples_per_step`. The primary arguments required for initialization are:

- `model_name`: the name of the base pretrained model (e.g., `"gpt2"`),
- `num_adapters`: the number of independent LoRA adapters,
- `rank`: the low-rank dimensionality used in LoRA,
- `alpha`: the LoRA scaling factor,
- `learning_rate`: the shared learning rate for all adapters,
- `samples_per_step`: the amount training sample to accumulate between weight update,
- `device`: the target device (e.g., `torch.device("cuda")`).

After the system is initialized, all stages of the system are performed in one call: `generate_and_train`.

```

ALoRA = ALoRA(model_name, num_adapters, rank, alpha, learning_rate,
              samples_per_step, device=torch.device("cuda"))

outputs = AloRA.generate_and_train(input_ids, masking,
                                   num_training, num_tokens,
                                   labels, attention_mask,
                                   use_cache=True)
```

4.2 Parallelized Computation with Masking

A mask *Adapters* is provided along with the input samples, and this mask is used to enable parallel computation of different adapters in one batch. Given batch size b and n adapters, the mask $M \in \mathbb{R}^{b \times n}$ contains rows that sum up to 1. The element $Adapters_{ij}$ describes the weight to apply the adapter j on sample i in the batch. Figure 4.1 shows a masking input with only one adapter applied to each sample, so the mask does one-hot selection.

ALoRA computes the outputs of all adapters for samples in the batch simultaneously using PyTorch's vectorized operations. Specifically, the outputs of all adapters are stacked in a tensor of shape $[b, n, h]$, where b is batch size, n is number of adapters and h is hidden size. A corresponding masking tensor is expanded and broad-casted to match the output dimensions, and element-wise multiplication followed by summation over the adapter dimension yields the result with only the selected adapter output added.

```
adapter_outputs = torch.stack([adapter(x) for adapter in
                               self.adapters], dim=1)

masking_expanded = masking.unsqueeze(-1).unsqueeze(-1)

result = (masking_expanded * adapter_outputs).sum(dim=1)
```

This approach uses the parallel processing capabilities of the GPU, allowing all adapter computations to be dispatched in a single batched operation. It also allows for parallel backpropagation of gradients across training samples for different adapters in a single batch.

However, this comes at the cost of a higher memory footprint. The approach temporarily allocates memory for the entire tensor of adapter outputs, and needs to store all activations for all adapters, which scales linearly with the number of adapters, rank of the adapter, and the model's hidden size.

4.3 Prefill to Output Generation Optimizations

To reduce latency, ALoRA avoids recomputing key and value projections after the backward pass by reusing the outputs from the prefill stage. Specifically, the keys and values corresponding to the inference samples are extracted from the prefill stage and fed directly into the updated model for the subsequent auto-regressive generation phase. This introduces a minor inconsistency: adapter weights may have been updated during the backward pass, and the reused keys and values may not reflect the latest model parameters. Thus, such an optimization would be appropriate in an interactive on-line learning setting where model changes are incremental and continuous. Additionally, the computational savings from reusing cached activations motivate the cost of this approximation, as demonstrated in our experimental results.

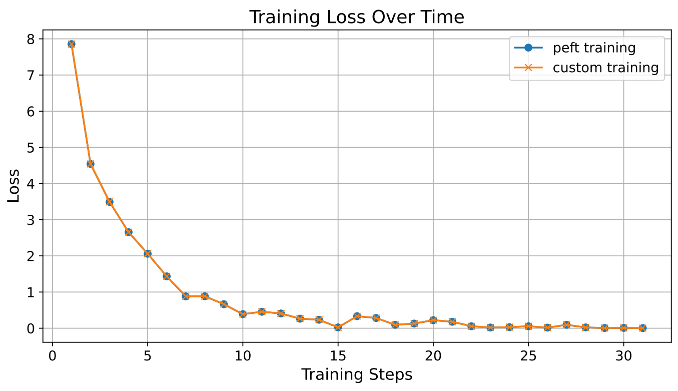
Evaluation

This section evaluates ALoRA using synthetic workloads and compares its performance with a HuggingFace PEFT implementation. Since there is no equivalent of PEFT that does mixed-batch training and inference, the PEFT baselines in the following sections are a naive implementation of the same tasks that ALoRA performs. The PEFT implementation processes each training sample sequentially. For every sample, the backward pass is performed, gradients are accumulated, and weights are updated if enough samples have been accumulated before the next training sample can do the same.

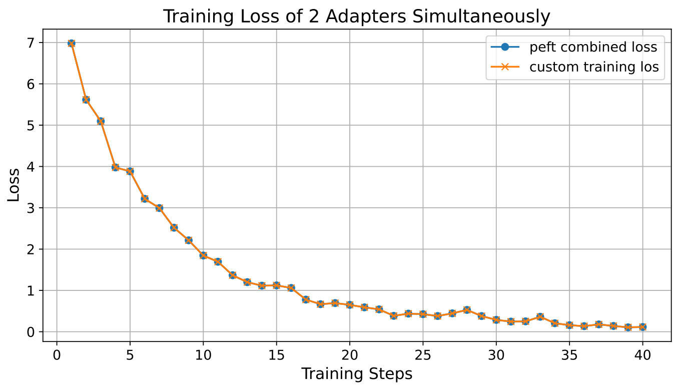
All experiments are conducted using the GPT2 base model(124 million parameters), though the system is designed to be adaptable to other transformer-based architectures. The evaluation explores a range of adapter counts and batch sizes, constrained to configurations that ensure all model weights and intermediate tensors fit within the memory of a single GPU.

5.1 Correctness Tests

The system's correctness was validated by verifying that mixed-batch ALoRA updates and outputs matched those of the PEFT implementation given identical inputs. For multi-adapter training, each adapter was assigned a distinct dataset. Figure 5.1 shows that the training loss curves produced by ALoRA closely align with those from PEFT when training adapters independently. Only one loss curve is shown for the two-adapter test because the system computes loss over the entire batch. The comparable PEFT loss was calculated by concatenating individual inference outputs and computing the loss across the whole batch.



(a) Matching loss curve for one-adapter training



(b) Matching loss curve for two-adapter training

Figure 5.1: Loss curves demonstrating correctness for one and two adapter setups.

5.2 Throughput Analysis

Throughput is measured as the total number of tokens processed during training and generated for output, divided by the total end-to-end execution time. This measurement encompasses all stages, including input tokenization, generation of training labels, gradient computation, weight updates, and generation of output sequence labels.

Figure 5.2 presents throughput comparisons between ALoRA and PEFT as the proportion of training samples within a batch increases. To ensure that both training and inference phases are executed regardless of batch size, each test includes at least one training sample per batch. Each graph corresponds to a different number of unique adapters managed by the system. Within each batch, samples are randomly assigned to an adapter with equal chance to each adapter. For the inference phase, 20 new tokens are generated per sample.

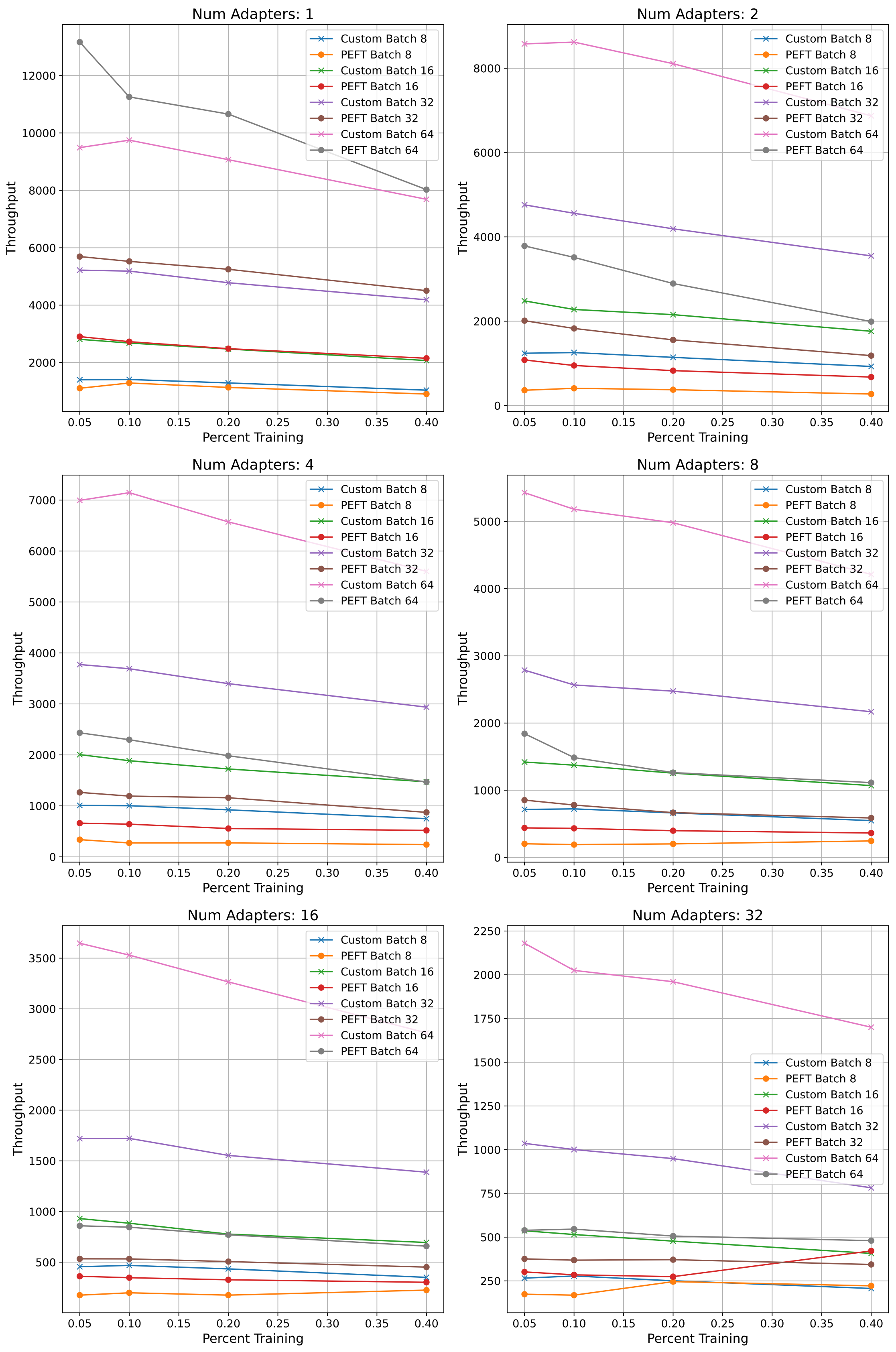


Figure 5.2: Throughput comparisons for different number of adapters between ALoRA and PEFT.

Speedup of (ALoRA throughput / PEFT throughput)		
Num Adapters	**5% training**	**40% training**
1	0.72	0.96
4	2.87	3.81
16	4.25	4.20
32	4.04	3.54

Table 5.1: Throughput speedup of ALoRA over PEFT at batch size **64** for different percentages of the batch as training samples.

ALoRA demonstrates over 4× higher throughput compared to the PEFT implementation with batch size of 64. The performance gains generally improve as both the number of adapters and batch size increase. To further investigate the sources of these improvements, the contribution of each phase of computation to the overall latency is displayed in the following section.

5.3 Latency Analysis

To understand the amount of time spent in each phase of the ALoRA pass, time of each phase of was recorded. In Figure 5.3, ”Fwd” represents the prefill stage, ”Bkwd” represents the gradient calculation and model weights updating phase, and ”Inf” represents the generation phase for just one new token for inference samples.

ALoRA achieves lower first-token inference latency by reusing past key values from the prefill stage, avoiding redundant computation. While these values may be slightly outdated due to recent weight updates, the effect is limited to a single token of the output. In contrast, PEFT lacks support for passing past key values during batched multi-adapter inference, requiring full recomputation for the first token. However, after the initial recalculation of the KV cache, it grows like usual for future token predictions.

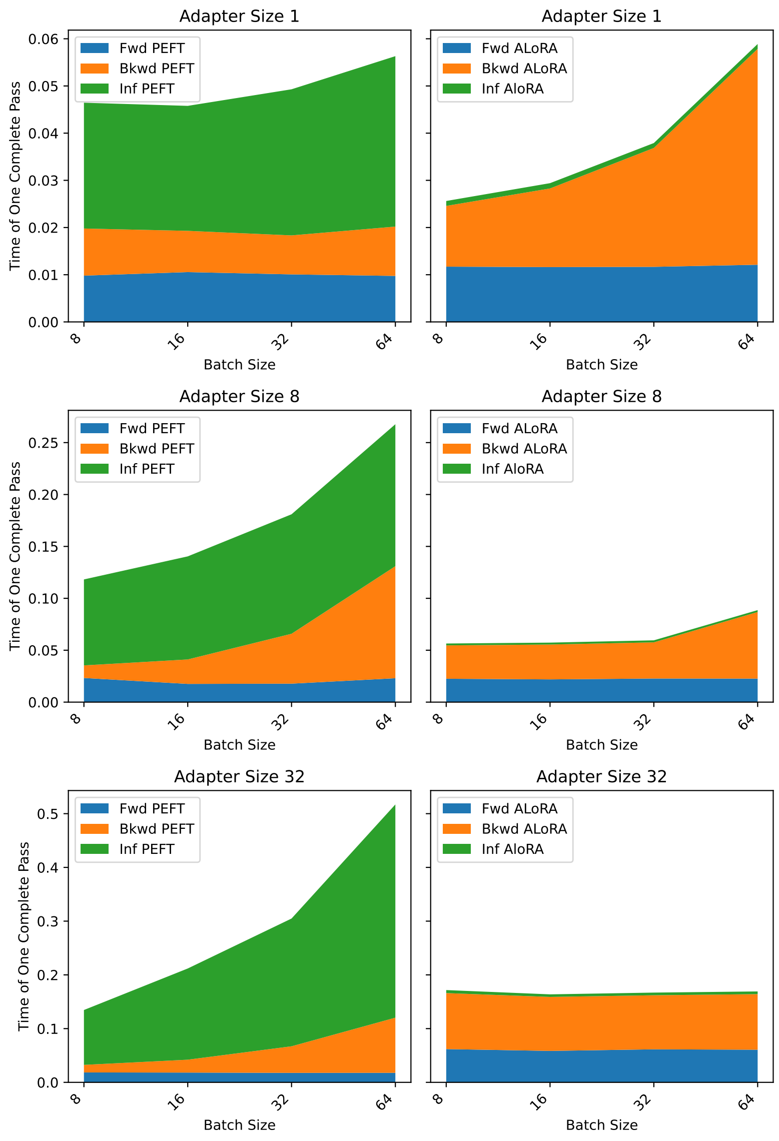


Figure 5.3: Time for each phase for ALoRA and PEFT.

5.4 Memory analysis

ALoRA's memory footprint is compared to the PEFT's footprint when performing the same training samples followed by inference. Four categories of memory footprint are displayed in the area graph:

1. **Model size:** sum of all base model weights and all LoRA adapter weights.

2. **Forward-Training pass:** additional memory to perform a forward pass and store activations for gradient calculation.

3. **Backward pass:** additional memory required to perform the backward pass on adapters, and store optimizer state

4. **Inference:** additional memory required to generate 2 new tokens for inference samples

In Figure 3.4, the changing memory footprint is shown as batch size and num adapters increase. In this experiment, to measure the worst-case memory usage, every single sample uses a different LoRA Adapter. Thus, batch size is equal to how many adapters the system supports. The ratio of data designated for training was held constant at 0.2 for these experiments.

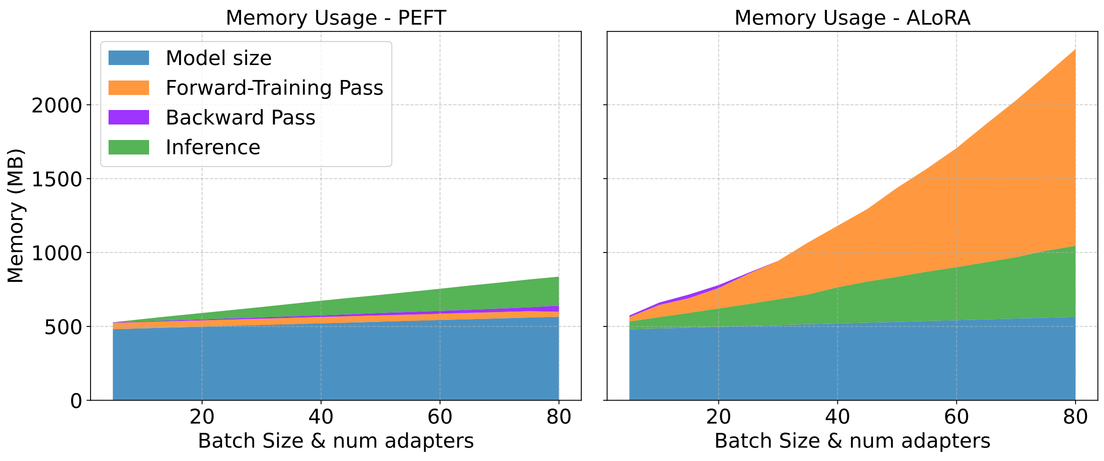


Figure 5.4: Memory footprint comparison between ALoRA and PEFT

The experiments confirm that ALoRA's prefill stage is significantly more memory-intensive than PEFT's, as it stores activations for all samples in the batch, while PEFT trains sequentially with minimal memory overhead. In contrast, during the inference phase, ALoRA's memory footprint doesn't differ much from that of PEFT, since intermediate broadcasted results are discarded, limiting memory growth to be linear to batch size. Additionally, while ALoRA's forward training pass results in high memory usage, the backward pass doesn't contribute to total memory usage, especially in batch sizes beyond 50. As shown in Figure 3.4, the overall training memory cost is dominated by the forward pass.

Discussion and Future Work

ALoRA addresses the gap of having a simple, single-GPU system that supports mixed-batch inference and training on multiple LoRA adapters. ALoRA effectively leverages batching to improve throughput in mixed training and inference workloads, with throughput gains increasing alongside the number of adapters.

However, its memory footprint scales linearly with both batch size and number of adapters. This limitation arises because PyTorch manages activations and gradients at the tensor level, requiring retention of intermediate states for all samples in the batch—even when only a subset is used for training. Consequently, ALoRA must store activations for the entire batch, even if most samples are inference-only.

Future work could explore custom kernel implementations that enable selective activation tracking for partial batch training. Other custom kernels like SGMV from Punica [11] could be used to perform efficient batched inference on adapters. Furthermore, graph-pruning techniques, such as those used in FlexLLM [8], may offer additional strategies to reduce memory usage in single GPU settings.

Related Work

Systems aimed to support efficient multiple LoRAs inference or fine-tuning on LLMs often combine scheduling techniques with memory and compute optimizations to improve performance. Since ALoRA explores changes independent of scheduling considerations, the following explanation of related work will be focused on the memory or compute optimizations of each system.

PEFT [19] implementation of mixed-batch inference attempts to make the most out of batching requests by grouping requests of the same adapter. If every single request in a batch uses a different adapter, there is no opportunity for batched inference. PEFT currently does not support mixed batch training.

Punica [11] presents a novel CUDA kernel for Segmented Gather Matrix-Vector Multiplication (SGMV). This enables batched adapter matrix operations across different LoRAs, storing only one copy of the base model on GPU. Additionally, incorporating a KV cache organization technique from vLLM[20] allows Punica to remove samples from the batch that have finished earlier.

S-LoRA [12] is designed for massively concurrent LoRA serving. It introduces Unified Paging, which maintains a shared memory pool for adapter weights and KV caches to minimize GPU fragmentation. S-LoRA also develops a custom CUDA kernel to support heterogeneous batching with LoRA adapters of variable rank.

FlexLLM [8] reduces GPU memory usage by pruning unecessary activations within the attention graph—essentially keeping only parts of the model that are needed for gradient calculation.

Conclusion

This work illustrates the potential for significant software-level optimizations in enabling mixed-batch training and inference with LoRA adapters on a single-GPU system. These are capabilities current systems such as PEFT or S-LoRA do not support.

ALoRA leverages masking and vectorized computation to enable efficient mixed-batch processing on a single GPU, achieving 4$\times$ throughput gains without relying on complex scheduling mechanisms or or multi-GPU infrastructure. By reusing intermediate outputs across phases, ALoRA also reduces redundant computation and lowers latency.

These findings suggest that with further kernel-level optimizations and thoughtful system design, interactive, transformer-based, multi-adapter platforms can be both performant and feasible on local hardware.

Ultimately, ALoRA highlights how targeted architectural changes can unlock new usage patterns in LLMs, enabling scalable multi-LoRA learning and inference systems.

Bibliography

[1] A. G. et al., "The llama 3 herd of models," 2024. [Online]. Available: https://arxiv.org/abs/2407.21783

[2] G. T. et al., "Gemini: A family of highly capable multimodal models," 2025. [Online]. Available: https://arxiv.org/abs/2312.11805

[3] C. C. S. Balne, S. Bhaduri, T. Roy, V. Jain, and A. Chadha, "Parameter efficient fine tuning: A comprehensive analysis across applications," 2024. [Online]. Available: https://arxiv.org/abs/2404.13506

[4] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," 2021. [Online]. Available: https://arxiv.org/abs/2106.09685

[5] Google AI, "Fine-tuning with the gemini api," https://ai.google.dev/gemini-api/docs/model-tuning, Apr. 2025, updated: 2025-04-28; accessed 2025-06-06.

[6] Apple Machine Learning Research, "Introducing apple's on-device and server foundation models," June 2024, accessed: 2025-06-05. [Online]. Available: https://machinelearning.apple.com/research/introducing-apple-foundation-models

[7] D. Soldatkin, G. Zappia, and R. Vegiraju, "Easily deploy and manage hundreds of lora adapters with sagemaker efficient multi-adapter inference," https://aws.amazon.com/blogs/machine-learning/easily-deploy-and-manage-hundreds-of-lora-adapters-with-sagemaker-efficient-multi-adapter-inference/, 2024, accessed: 2025-06-05.

[8] G. Oliaro, X. Miao, X. Cheng, V. Kada, R. Gao, Y. Huang, R. Delacourt, A. Yang, Y. Wang, M. Wu, C. Unger, and Z. Jia, "Flexllm: A system for co-serving large

language model inference and parameter-efficient finetuning," 2025. [Online]. Available: https://arxiv.org/abs/2402.18789

[9] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A distributed serving system for Transformer-Based generative models," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 521–538. [Online]. Available: https://www.usenix.org/conference/osdi22/presentation/yu

[10] B. Wu, R. Zhu, Z. Zhang, P. Sun, X. Liu, and X. Jin, "dLoRA: Dynamically orchestrating requests and adapters for LoRA LLM serving," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, Jul. 2024, pp. 911–927. [Online]. Available: https://www.usenix.org/conference/osdi24/presentation/wu-bingyang

[11] L. Chen, Z. Ye, Y. Wu, D. Zhuo, L. Ceze, and A. Krishnamurthy, "Punica: Multi-tenant lora serving," 2023. [Online]. Available: https://arxiv.org/abs/2310.18547

[12] Y. Sheng, S. Cao, D. Li, C. Hooper, N. Lee, S. Yang, C. Chou, B. Zhu, L. Zheng, K. Keutzer, J. E. Gonzalez, and I. Stoica, "S-lora: Serving thousands of concurrent lora adapters," 2024. [Online]. Available: https://arxiv.org/abs/2311.03285

[13] OpenAI, "What are tokens and how to count them?" 2025, accessed: 2025-05-30. [Online]. Available: https://help.openai.com/en/articles/4936856-what-are-tokens-and-how-to-count-them

[14] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023. [Online]. Available: https://arxiv.org/abs/1706.03762

[15] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019. [Online]. Available: https://arxiv.org/abs/1810.04805

[16] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[17] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," 2023. [Online]. Available: https://arxiv.org/abs/1910.10683

[18] A. Aghajanyan, L. Zettlemoyer, and S. Gupta, "Intrinsic dimensionality explains the effectiveness of language model fine-tuning," 2020. [Online]. Available: https://arxiv.org/abs/2012.13255

[19] Hugging Face, "Lora · developer guides · peft documentation," https://huggingface.co/docs/peft/main/en/developer_guides/lora, Jun 2025, accessed: 2025-06-06.

[20] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 611–626. [Online]. Available: https://doi.org/10.1145/3600006.3613165