Activation-Efficient Fine-Tuning via Sequence-Compressed Low-Rank Subspaces

Irene Tenison[1], Stella Ahn[1], Miriam Kim[1,2], Ebtisam Alshehri[1], Lalana Kagal[1]

[1]MIT CSAIL, [2]Harvard SEAS

Abstract

Parameter-Efficient Fine-Tuning (PEFT) has become the standard approach for adapting large language models (LLMs). However, despite reducing trainable parameter counts, many PEFT methods remain memory-bound by token-level activation intermediates that scale linearly with sequence length, limiting their practicality for long-context and on-device adaptation. Unlike prior memory-efficient PEFT approaches that preserve token-level adapter activations, LARS (Low-memory Activation-Rank Subspace) reduces the sequence-length scaling of stored adapter activation intermediates from $\mathcal{O}(BSRL)$ to $\mathcal{O}(BRL)$ by performing adaptation within a sequence-level low-rank subspace. Through pooled feature extraction, low-rank subspace modulations, and residual projection, LARS reduces the overall *memory footprint by 23% on GPUs (Figure 3) and 47.8% on CPUs (Table 3) relative to LoRA at* $S = 1024$*, with relative savings increasing with context length.* Experiments across reasoning, understanding, long-context, retrieval, and token-level tasks show that *LARS maintains competitive performance relative to LoRA* while substantially improving memory efficiency. Beyond GPUs, LARS also *reduces average CPU power and energy consumption per optimization step by 22.43% and 22.06%, respectively*. These results demonstrate that activation-efficient adaptation provides a practical path toward resource-constrained and on-device fine-tuning of LLMs.

1 Introduction

LLMs and transformer-based architectures have become the backbone of modern natural language processing (Vaswani et al., 2023; Touvron et al., 2023). While these models exhibit remarkable zero-shot capabilities, fine-tuning remains essential for specialized tasks, privacy-preserving local adaptation, and low-latency personalization in mobile and edge environments (Wang et al., 2025a). However, the high memory requirements of LLM adaptation pose a significant barrier to deployment on resource-constrained hardware, where available memory is often restricted to a few gigabytes.

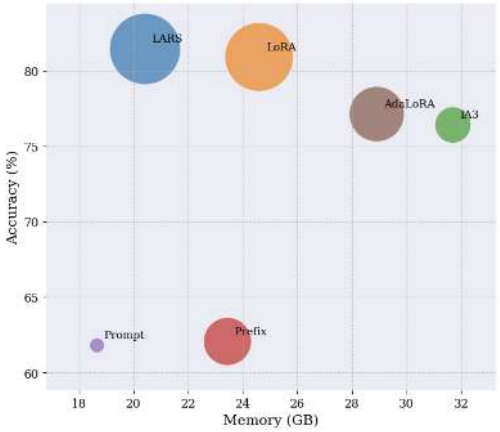


Figure 1: Accuracy vs. peak memory (GB) for state-of-the-art PEFT methods. Bubble size represents the count of trainable parameters. Our analysis reveals a critical disconnect: trainable-parameter count is a poor proxy for actual memory footprint.

To address these constraints, the research community has pivoted toward PEFT(Han et al., 2024). Methods such as LoRA (Hu et al., 2021), prefix tuning (Li and Liang, 2021), and IA3 (Liu et al., 2022) update only a minute fraction of the model's total weights—often by orders of magnitude—while maintaining competitive downstream accuracy. This success has solidified *a fundamental design assumption: that reducing the number of trainable parameters directly translates to improved deployability in memory-limited environments.*

As illustrated in Figure 1, there is a striking lack of correlation between a method's parameter efficiency and its actual physical memory footprint during adaptation. For instance, IA3 — one of the most parameter-efficient methods—requires signif-

Proceedings of the 15th International Joint Conference on Natural Language Processing and the 5th Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics, pages 5668–5687
November 6-10, 2026 ©2026 Association for Computational Linguistics

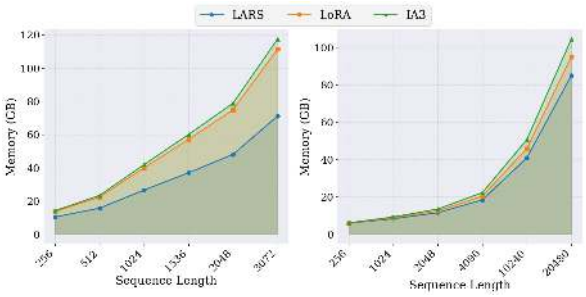


Figure 2: Peak memory scaling vs. sequence length. (Left) Without GC and with GC enabled (Right), LARS defines a more efficient Pareto frontier, enabling the fine-tuning of sequences longer than LoRA within the same hardware constraints.

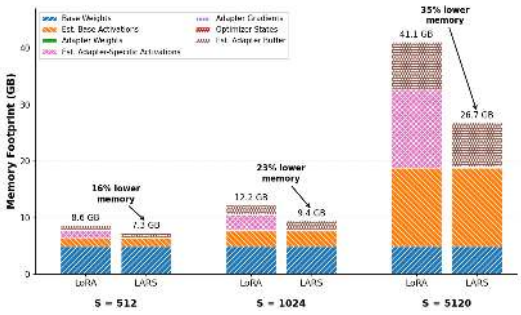


Figure 3: Memory footprint breakdown of Llama 3.2 1B with LoRA and LARS across sequence lengths 512, 1024, and 5120. LARS reduces the sequence-length-dependent activation overhead of the adapter, yielding 12%, 23%, and 34% lower overall memory consumption, respectively. Adapter-specific weights, gradients, and optimizer states are comparatively small and are not visually prominent. Estimated adapter-specific activations are the adapter internals that are reduced by LARS. Adapter buffer includes activations of adapter outputs and this scales with S. More details in Appendix A.2

icantly more peak memory than LoRA, despite having fewer trainable weights.

Following prior works (Chen et al., 2016; Hao et al., 2024; Zhang et al., 2024), we argue that the prevailing focus on parameter count overlooks the primary bottleneck of on-device adaptation - intermediate activation storage. Because most PEFT methods leave the forward computational graph largely unchanged, they still incur massive activation overhead that scales with batch size and sequence length, regardless of how few parameters are updated (Dao, 2024). Consequently, reducing parameter count alone provides diminishing returns when activation memory dominates the peak footprint (Lin et al., 2022).

We introduce LARS (Low-memory Activation-Rank Subspace), an adaptation framework that trades token-level granularity for substantially lower adapter activation memory. U*nlike prior memory-efficient PEFT methods that primarily reduce sparsity or backward-computation overhead (Woo et al., 2024; Pan et al., 2024; Zhang et al., 2024) while preserving token-level adapter activations, LARS reduces the sequence-length scaling of the adapter's retained trainable intermediates themselves.* By operating within a sequence-level low-rank subspace, LARS lowers the hardware barrier for long-context and on-device adaptation while remaining complementary to existing systems-level memory optimizations. Our key contributions are:

- We propose LARS which performs adaptation in a sequence-pooled, low-rank subspace, reducing the size of stored adapter activations as illustrated in Figure 4.
- LARS reduces the overall memory footprint by 23% on GPUs% on GPUs (Figure 3) and 47.8% on CPUs (Table 3) in comparison to LoRA over $S = 1024$. Furthermore, the relative memory savings increase with sequence length as shown in Figure 2.
- Our experiments across reasoning, understanding, long-context, and token-level tasks (Tables 1, 7, 2) using different models further show that LARS maintains comparable accuracy to LoRA.
- LARS reduces average power and energy consumption per step by 22.43% and 22.06% respectively in comparison to LoRA on CPUs (Table 3).

2 Background

Adapting LLMs on-device requires understanding the runtime memory footprint beyond parameter counts. Peak memory is dominated by: $M_{peak} \approx M_{params} + M_{grads} + M_{opt} + M_{acts}$ where $M_{params}, M_{grads}, M_{opt}$, and M_{acts} denote the memory consumed by model parameters, θ, gradients of trainable parameters, optimizer states, and activations respectively.

Parameter-related terms scale with model size; $M_{params} = \mathcal{O}(|\theta|)$ and $M_{grads}, M_{opt} = \mathcal{O}(|\theta_{trainable}|)$. In contrast,activation memory depends on both model and input dimensions. For a transformer with depth L, hidden dimension H,

batch size B, and sequence length S, $M_{acts} = \mathcal{O}(BSHL)$. This creates a fundamental asymmetry: while parameter memory is largely static, activation memory grows directly with sequence length S and batch size B, often becoming the dominant bottleneck during adaptation.

2.1 Memory Consumption in PEFT Methods

PEFT methods operate in the regime where the trainable parameter fraction $\rho = \frac{|\theta_{trainable}|}{|\theta|} << 1$ (Hu et al., 2021; Han et al., 2024). This substantially reduces gradient and optimizer-state memory. However, reducing trainable parameters alone does not significantly lower peak memory usage because activations remain dominant.

Most PEFT methods preserve the original token-level computational graph. For example, LoRA(Hu et al., 2021) applies a low-rank update of the form $Wx + ABx$, where $A \in \mathbb{R}^{H\times R}, B \in \mathbb{R}^{R\times H}$, and $R << H$. Although the trainable parameter count scales with R, the intermediate hidden states must still be retained for gradient computation. Consequently, activation memory scales as $M_{\text{acts}} = \underbrace{\mathcal{O}(BSHL)}_{\text{Base Activations}} + \underbrace{\mathcal{O}(BSRL)}_{\text{Adapter Activations}}$.

As shown in Figure 2, this token-level dependence causes memory usage to grow linearly with sequence length even under system-level optimizations such as gradient checkpointing. Existing PEFT approaches therefore remain constrained by a practical sequence-length ceiling on memory-limited hardware.

Figure 3 and Table 4 further show that adapter weights, gradients, and optimizer states contribute only a negligible fraction of the overall memory footprint and peak memory during adaptation is dominated by activations. Although the base model has more parameters than the adapters or PEFT modules, it is frozen and does not retain as much activations for backpropagation. However, the activation memory of the base model remains substantial and scales with sequence length. Existing PEFT methods introduces additional trainable token-level intermediate tensors whose retained runtime memory grows with sequence length and has to be retained during backpropagation. Consequently, the relative contribution of adapter-specific activation memory becomes increasingly pronounced in long-context settings. These observations motivate a shift from parameter-efficient adaptation toward activation-efficient adaptation.

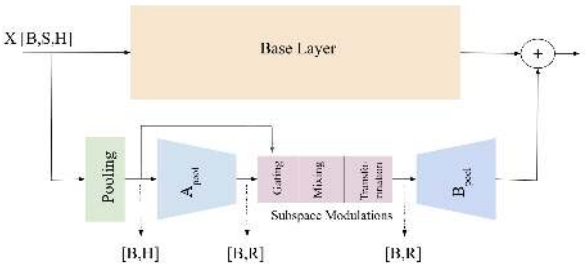


Figure 4: Overview of LARS. Unlike LoRA, which performs token-level low-rank adaptation and retains sequence-dependent intermediates of size $\mathbb{R}^{B\times S\times R}$ during backpropagation, LARS compresses hidden states into a pooled sequence-level representation and performs adaptation entirely within a compact low-rank subspace $\mathbb{R}^{B\times R}$. As a result LARS reduces adapter activation scaling from $\mathcal{O}(BSRL)$ in LoRA to $\mathcal{O}(BRL)$.

These observations motivate a shift from parameter-efficiency toward activation-efficiency. *LARS addresses this bottleneck by decoupling adapter-specific activations from sequence length, asymptotically reducing adapter activation complexity from $O(BSRL)$ to $O(BRL)$.* Notably, LARS reduces overall memory by 12–34% in comparison to LoRA, while its estimated adapter-specific activation memory remains minimal and effectively constant as sequence length S increases (Table 4).

3 Methods

Problem Setting: Let M be a pre-trained Transformer with L layers and hidden dimension H. For an input sequence $X \in \mathbb{R}^{B\times S\times H}$, standard PEFT methods (e.g., LoRA) compute an update $\Delta X = f(X; \theta_{trainable})$. The primary memory overhead in these methods arises from the adapter-specific activations that must be materialized and stored to compute the gradient $\nabla_{\theta_{trainable}}$. While base activations can be managed via GC, the adapter's intermediate tensors — of shape $[B, S, R]$ —still scale linearly with sequence length S. Our objective is to design an adaptation function $f(.)$ such that the adapter-specific activation memory is decoupled from S, reducing its complexity from $\mathcal{O}(BSRL)$ to $\mathcal{O}(BRL)$.

3.1 LARS (Low-memory Activation-Rank Subspace) Adaptation

We propose LARS (Low-memory Activation-Rank Subspace) as illustrated in Figure 4, an adaptation architecture designed to decouple the memory

footprint of trainable modules in PEFT from the input sequence length S. While standard PEFT methods maintain a token-parallel intermediate activations per layer $[B, S, R]$, LARS executes adaptation within a compressed latent manifold, which ensures that the intermediate tensors required for gradient computation in the adapters are S-independent. The LARS execution pipeline consists of three stages: (i) Pooled Feature Extraction, which projects the sequence into a global context vector; (ii) Low-Rank Subspace Adaptation, which performs non-linear modulation in a rank-reduced space; and (iii) Residual Integration, which projects the learned updates back to the original manifold via broadcasting.

3.1.1 Pooled Feature Extraction

In the first stage of LARS, we address the activation bottleneck by projecting the sequence $X \in \mathbb{R}^{B\times S\times H}$ into a condensed global context vector $X_{pool} \in \mathbb{R}^{B\times H}$. This operation is the critical "memory-breaker" that enables LARS to avoid storing token-level hidden states for the adapter's backward pass. We propose two distinct strategies to manage the interplay between representation quality and hardware constraints.

1. **Heuristic-Driven Fixed Pooling** This strategy aggregates global semantic information without introducing new trainable parameters. We employ a hybrid mean-pooling scheme:

$$x_{pool} = \frac{1}{S}\sum_{i=1}^{S} X_i + X_S \tag{1}$$

By augmenting the sequence mean with the final token representation X_S, we ensure the pooled vector captures both the global context and the recency bias inherent in causal transformers (Ruscio et al., 2025). Fixed pooling is $\mathcal{O}(1)$ in terms of additional activation memory, making it the optimal choice for device constraints.

2. **Context-Aware Learned Pooling** For tasks requiring finer semantic precision, we also propose Context-Aware Learned Pooling. This variant adaptively weights "high-information" tokens (e.g., verbs in reasoning tasks) over redundant padding. While this adds a marginal $\mathcal{O}(BH)$ activation term for attention weights, it remains an order of magnitude smaller than the $\mathcal{O}(BSH)$ requirement of token-level adapters (Ennadir et al., 2025). We designate fixed pooling as the default throughout this work to prioritize maximum memory efficiency for on-device constraints.

3.1.2 Low-Rank Subspace Modulations

Given the pooled representation $x_{pool} \in \mathbb{R}^{B\times H}$, LARS projects it into a low-rank subspace of dimension $R << H$.

$$h = x_{pool}A_{pool}; \quad A_{pool} \in \mathbb{R}^{H\times R} \tag{2}$$

While A_{pool} is conceptually similar to the input projection in LoRA (Hu et al., 2021), in LARS its goal is to map a global sequence summary into a feature space that remains informative. To maximize the representational capacity of this singular vector $h \in \mathbb{R}^{B\times R}$, we introduce a tiered modulation framework.

Feature-Space Gating To ensure the model remains sensitive to different input contexts despite the sequence collapse, we apply an instance-conditioned gate $g \in \mathbb{R}^{B\times R}$. The gating signal is computed by integrating the global context with the local subspace features:

$$g = \sigma(\tau_1 \cdot W_x \cdot x_{\text{pool}} + \tau_2 \cdot W_h \cdot LN(h)) \tag{3}$$

where $W_x \in \mathbb{R}^{H\times R}$ and $W_h \in \mathbb{R}^{R\times R}$ are learnable linear projections, $LN(.)$ denote layer normalization, and σ denotes the element-wise sigmoid function. We introduce learnable temperature scalars τ_1 and τ_2 to calibrate the relative influence of the global and local representations before the sigmoid activation. This mechanism allows the model to dynamically amplify or suppress specific subspace coordinates based on the input's global semantic signature.

Inter-Rank Mixing Standard low-rank updates typically treat each subspace dimension as an independent component, limiting the model's capacity to capture feature correlations. To approximate the expressive power of a full-rank transformation, we introduce a learnable mixing transformation $M_{mix} \in \mathbb{R}^{R\times R}$ applied directly to the gating vector. This implements relational gating, where the importance of a specific subspace coordinate is conditioned on the global state of the gating manifold:

$$g_{mix} = gM_{mix}, \quad \text{and} \quad h' = g_{mix} \odot h \tag{4}$$

This facilitates cross-rank communication and higher-order dependencies without expanding the activation footprint beyond $\mathcal{O}(BR)$, providing a significant boost to adaptation capacity on constrained devices.

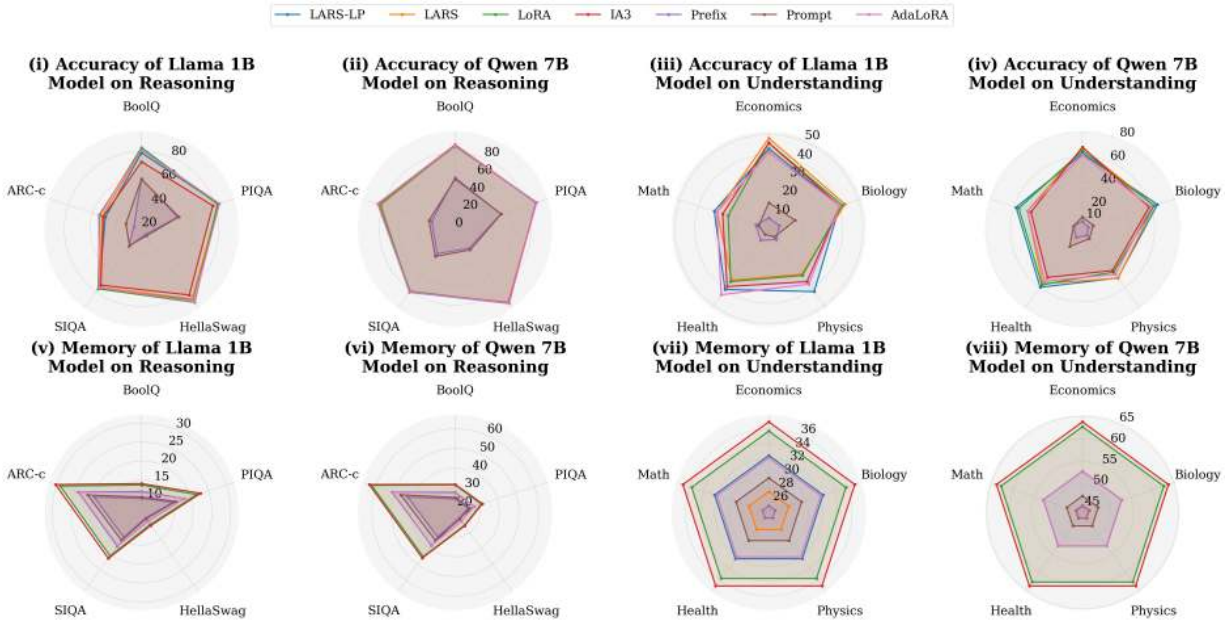


Figure 5: Accuracy and Memory on Llama 1B and Qwen 7B on reasoning and understanding tasks across various datasets. While LARS consumes significantly lower memory than LoRA, IA3, and AdaLoRA, the performance is comparable on reasoning and understanding tasks.

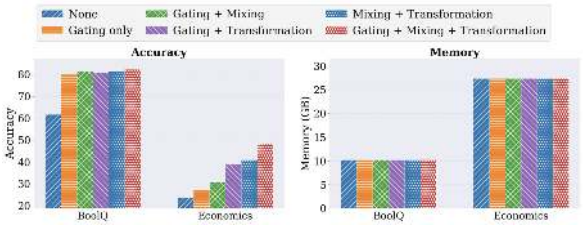


Figure 6: Impact of Gating, Mixing, and Transformation components on Accuracy and Memory Usage (GB) across the BoolQ and Economics datasets.

Subspace Non-Linear Transformation To further enhance the representational capacity of the adaptation module, we introduce a lightweight non-linear bottleneck within GELU activation in the rank-reduced manifold. This operation provides the functional complexity necessary to model non-linear residual updates without reverting to expensive token-level computations.

Figure 6 evaluates the contribution of the proposed subspace modulation components in LARS — Gating, Mixing, and Non-Linear Transformation. The results show that progressively combining these components consistently improves downstream accuracy, particularly on the more challenging Economics dataset, while introducing negligible additional memory overhead. This suggests that adaptation signals can be effectively modeled within a compact sequence-level low-rank manifold, where lightweight modulation operations recover representational expressivity without reintroducing expensive token-level activations. These findings further support the central design principle of LARS: preserving adaptation quality does not require maintaining full sequence-dependent adapter activations throughout training.

3.1.3 Residual Projection and Integration

The final stage of LARS maps the adapted subspace features back to the original model manifold via broadcasting across S. The modulated representation $h' \in \mathbb{R}^{B \times R}$ is projected to the hidden dimension H and integrated into the frozen backbone via a gated residual connection with learnable scalar α to calibrate the adapter's influence on the frozen features.

$$\begin{aligned} \text{out}[b, s, :] &= \text{Base}_{\text{out}}[b, s, :] + \alpha.\text{Adapter}_{\text{out}}[b, :] \\ \text{Adapter}_{\text{out}} &= B_{pool}(h') \end{aligned} \tag{5}$$

The residual formulation (Houlsby et al., 2019) ensures the numerical stability of the pre-trained weights and mitigate catastrophic forgetting, as the adapter shifts the base model's distribution.

Summary By collapsing the sequence dimension before the high-rank backward pass, LARS decouples the adapter's activation memory footprint from input length, enabling scalable fine-tuning of longer contexts on memory-constrained devices.

Table 1: Average memory usage and accuracy of LARS and its learned pooling variant, LARS-LP on various reasoning, understanding, and long context tasks compared against baselines - LoRA, AdaLoRA, IA3, Prefix tuning, and Prompt Tuning. LARS consumes an average of 25% lower memory than the baselines across tasks and models while performing comparably to LoRA.

	(%)	Llama-3.2-1B						Qwen2.5-7B-Instruct					
	Trainable	Reasoning		Understanding		Long Context		Reasoning		Understanding		Long Context	
	Params	Mem.(↓)	Acc.(↑)	Mem.(↓)	Acc.(↑)	Mem.(↓)	Acc.(↑)	Mem.(↓)	Acc.(↑)	Mem.(↓)	Acc.(↑)	Mem.(↓)	Acc.(↑)
LoRA	0.45	19.06	**76.19**	35.72	35.74	25.56	57.91	35.72	35.73	62.73	57	52.1	67.37
AdaLoRA	0.45	16.06	75.8	32.6	37.4	23.44	**58.03**	32.06	38.4	52.81	53.99	46.73	**67.59**
IA3	0.01	19.83	71.66	36.99	37.4	26.49	54.13	36.99	37.44	63.81	52.91	53	62.02
Prefix	0.03	**13.47**	39.23	**25.49**	9.05	22.30	24.62	**25.49**	9.03	**45.22**	9.17	38.89	25.95
Prompt	0.003	14.34	40.79	29.31	12.87	20.79	24.39	29.31	15.47	47.39	14.71	39.04	25.88
LARS	0.67	13.55	75.6	27.44	37.33	**20.27**	57.18	27.44	37.33	45.38	56.97	**38.33**	67.18
LARS-LP	0.67	16.07	75.97	32.39	**39.32**	23.19	54.52	32.39	**39.32**	52.87	**59.19**	44.28	63.9

Combined with gating, inter-rank mixing, and non-linear transformations, LARS recovers the expressive capacity lost during pooling. More details on the implementation including initializations and code snippets are given in Appendix B.

4 Experiments and Results

4.1 Setup

Datasets and Tasks We evaluate LARS primarily across three task categories: (1) Commonsense Reasoning using a subset of the LLM-adapters benchmark (Hu et al., 2023) (BoolQ, PIQA, SIQA, HellaSwag, and ARC-c (Clark et al., 2019; Bisk et al.; Sap et al., 2019; Zellers et al., 2019; Bhakthavatsalam et al., 2021)); (2) General Understanding via five MMLU-Pro subjects (Wang et al., 2024) (Economics, Biology, Physics, Health, and Math); and (3) Long-Context Comprehension using QuALITY (Pang et al., 2022) and RACE (Lai et al., 2017). These tasks measure semantic reasoning and document-level understanding under memory constraints. Additionally, to further understand the performance of LARS, which compresses token-level information in the adapters, we experiment on representative token-level and generation tasks - sequence labeling on CoNLL (Tjong Kim Sang and De Meulder, 2003), extractive QA on SQuAD (Rajpurkar et al., 2016), and summarization on SAMSum (Gliwa et al., 2019). All theses datasets are public and easy to access.

Baselines We evaluate LARS against five standard PEFT methods: LoRA (Hu et al., 2021), its rank-adaptive variant AdaLoRA (Zhang et al., 2023), and the multiplicative scaling method IA3 (Liu et al., 2022). We also compare against the prompt-based approaches of Prefix (Li and Liang, 2021) and Prompt Tuning (Lester et al., 2021).

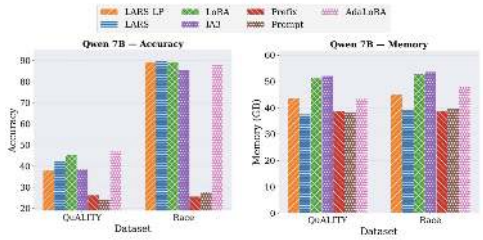


Figure 7: Memory consumption and accuracy of Qwen 7B on long context Quality and RACE datasets

Implementation Our experiments involve publicly available Llama-3.2-1B model that was run on NVIDIA L4OS GPU and Qwen2.5-7B-Instruct model that was run on H200 GPU as the primary base models for all reasoning, understanding, and long-context tasks. All experiments report results from a single run as the primary contribution of this work—reducing adapter-specific activation memory scaling—is determined by the architectural structure of the method and is largely invariant to initialization or random seed choice. The reported accuracy trends are comparative in nature. For most experiments we use LARS with fixed pooling as this paper focuses on a memory-constrained on-device setting. All baseline implementations are from Huggingface PEFT library with all default parameters. Appendix C gives more details on the implementation of LARS, datasets, hardware, hyperparameters, tuning, and memory and power measurement methodology used for various experiments.

4.2 Results and Discussion

4.2.1 Reasoning and Understanding

As shown in Table 1, LARS demonstrates a superior memory-to-accuracy trade-off compared to other baselines. The primary advantage of LARS lies in its significant reduction of the memory footprint while maintaining comparable performance.

Table 2: Additional experiments on representative token-level and generation tasks: sequence labeling (SL) and structured decoding (SD) on CoNLL, Extractive QA on SQuAD, and Summarization on SAMSum, to better understand the effect of sequences-compression in LARS.

	CoNLL(SL)		CoNLL(SD)	SQuAD		SAMSum	
	acc	F1	Entity-F1	EM	F1	Rouge-1	Rouge-L
LARS	***98.12***	***0.78***	0.77	0.556	0.644	0.277	0.204
LoRA	96.11	0.73	***0.82***	***0.558***	***0.646***	***0.302***	***0.230***
IA3	96.89	0.75	0.73	0.414	0.495	0.294	0.220
AdaLoRA	96.68	0.76	0.75	0.556	0.642	0.293	0.224

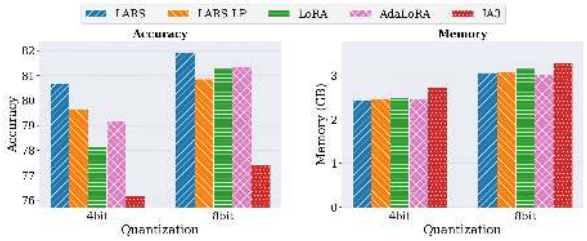


Figure 8: Memory Usage (GB) and Accuracy of LARS and baselines with 4 bit and 8 bit quantization.

On the Qwen 7B and Llama3.2 1B models for understanding tasks, LARS requires only approximately 25% lower memory on average than LoRA at $S = 1024$. Despite this smaller footprint, LARS maintains competitive accuracy compared to LoRA. These results can be further validated with the dataset-specific and visual results in Figure 5. While the memory-usage (bottom row) is visibly minimized for LARS, the accuracy contours (top row) remain largely overlapping with the most baselines. *These findings suggest that LARS provides a favourable solution for deploying LLMs on hardware with limited VRAM, offering a more efficient path for fine-tuning without the typical performance degradation associated with aggressive memory optimization.*

LARS-LP, which is a variant of LARS with the learned pooling strategy introduced in Section 3.1.1 instead of fixed pooling variant, successfully bridges the slight accuracy gap in LARS. In the Qwen 7B Reasoning task, LARS-LP improves the accuracy to 39.32%, outperforming standard LARS (37.33%) and other baselines although it consumes slightly more memory. *This indicates that the learned pooling mechanism effectively retains critical features necessary for complex reasoning and understanding while maintaining the efficiency gains of the LARS architecture.*

Long Context: For long-context datasets, across both models as shown in Figures 7 and 13 and datasets, LARS consistently demonstrates superior memory efficiency and comparable performance relative to all other high-performing adapters similar to reasoning and understanding tasks. Collectively, these results indicate that *LARS provides a favourable trade-off for long-context applications, offering the high-rank representation power of LoRA with a memory profile more closely resembling (or bettering) more restrictive PEFT methods.*

Token-Level Fidelity: Despite operating in a sequence-pooled subspace, LARS remains competitive with standard PEFT methods across all tasks. As shown in Table 2, on CoNLL, LARS achieves the highest accuracy and F1 score, suggesting that compressed sequence representations can still preserve strong token-level semantic information. On SQuAD, LARS performs comparably to LoRA and AdaLoRA, with only a marginal drop in EM and F1. For summarization on SAMSum, LoRA achieves the strongest ROUGE scores, while LARS remains comparable despite its substantially more compressed activation structure. We additionally conducted Needle-In-A-Haystack (NIAH) style passkey retrieval experiments and observed that LARS, LoRA, AdaLoRA, and IA3 achieved near-perfect accuracy at lengths up to 16k. However, performance across all methods declined at the 32k context length as shown in Table 7. Overall, *these results indicate that LARS maintains comparable downstream performance even on tasks that traditionally benefit from token-level adaptation.*

4.2.2 Effect of Quantization

Figure 8 evaluates the effects of 4-bit versus 8-bit quantization on accuracy and memory usage across LARS and baselines. The plots show that LARS consistently achieves the highest performance in both 4 bit and 8 bit settings while consuming lower memory that baselines. Similar characters were observed with other systems-level optimization methods like FlashAttention (Dao, 2024), Checkpointing (Chen et al., 2016), as well as memory optimization methods like DropBP(Woo et al., 2024) and LoRA-FA(Zhang et al., 2024) as shown in Appendix E. Notably these methods are complementary to LARS and they can be combined to achieve better resource efficiency.

4.2.3 Throughput Analysis

Figure 9 illustrates the training and inference throughput across adaptation methods. While LARS is highly efficient for deployment in inference throughput and latency (see Figure 14, Ap-

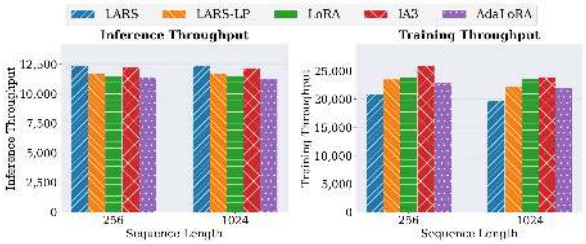


Figure 9: Throughput of LARS and other baselines during inference and fine-tuning across sequence lengths 256 and 1024.

Table 3: Accuracy, memory consumption, throughput, average time per step, average power per step and average energy per step of LARS, LoRA and IA3 on AMD EPYC CPU.

	Accuracy (%)	Memory (GB)	Throughput (tok/sec)	Time (sec)	Power (W)	Energy (J)
S = 256						
LARS	76.02	**15.824**	113.22	31.67	74.64	2238.16
LoRA	77.56	27.059	153.25	32.15	135.96	4231.88
IA3	67.5	18.937	101.78	27.8	89.5	2466.48
S = 512						
LARS	74.58	**15.926**	111.88	30.65	140.81	4731.87
LoRA	75.39	28.164	189.21	33.93	160.86	5071.31
IA3	70.71	19.077	101.95	30.68	161.95	5731.88
S = 1024						
LARS	78.24	**16.866**	111.56	30.24	148.19	4737.42
LoRA	78.85	32.314	174.38	34.37	164.14	5405.82
IA3	70	19.827	97.65	31.07	183.96	6556.87

pendix)—a divergence appears during fine-tuning. We observe that LARS is slightly slower in terms of training tokens per second compared to LoRA. However, this marginal reduction in speed is offset by a substantial gain in hardware accessibility and memory efficiency. This demonstrates that *LARS effectively trades a fraction of computational throughput to mitigate the activation bottleneck* enabling long-context adaptation on devices where standard methods often fail.

4.2.4 Beyond GPUs

To evaluate the practical efficiency and on-device deployment of the proposed LARS method, we compared the convergence accuracy, memory consumption, throughput, average time per step, average power per step and average energy per step of LARS against LoRA and IA3 on both CPUs. Across all settings, LARS consistently achieves the lowest memory consumption while maintaining competitive accuracy and throughput. At $S = 1024$, LARS reduces peak memory usage by nearly 47.8% relative to LoRA. LARS also reduces average power consumption and average energy per step across all sequence lengths by 22.43% and 22.06% compared to LoRA. These results demonstrate that *LARS provides a favorable trade-off between adaptation quality and hardware resource usage, making it particularly suitable for memory- and energy-constrained CPU environments.*

5 Related Works

Parameter-Efficient Adaptation PEFT methods aim to adapt LLMs by updating a minimal fraction of the total weights. Methods like IA3(Liu et al., 2022) and TENT(Wang et al., 2021) moved away from the adapters to achieve better efficiency with a performance drop. Most methods (Chen et al., 2024; Hu et al., 2021; Zhang et al., 2023) remain bound by the "Activation Wall" — where M_{acts} scales linearly with sequence length (Lin et al., 2022; Kwon et al., 2024; Tenison et al., 2026).

System-Level Memory Optimization To mitigate activation overhead, the systems community has introduced GC(Chen et al., 2016), which trades FLOPs for memory by recomputing activations, and FlashAttention(Dao, 2024), which optimizes the attention matrix. More recently, LISA(Pan et al., 2024) reduce memory by freezing specific layers during the backward pass. However, these are optimizations do not alter the scaling of hidden-state tensors. LARS complements these approaches by decoupling adapter-specific activations from the sequence dimension.

On-Device & Edge Intelligence Deploying LLMs on resource-constrained hardware necessitates aggressive compression, typically through quantization (e.g., QLoRA (Dettmers et al., 2023)) or Pruning (Kwon et al., 2024; Tenison et al., 2026). While these target M_{params}, LARS addresses the dynamic peak memory bottleneck. LARS approach aligns with the growing need for "tiny training" (Lin et al., 2022) and local adaptation where RAM is often the limiting factor (Wang et al., 2025b).

6 Conclusion

We introduced an activation-efficient adaptation framework that reduces the sequence-length scaling of adapter activations by operating within a sequence-level low-rank subspace. By collapsing the sequence dimension prior to low-rank adaptation and residual reconstruction, LARS reduces the memory footprint associated with adapter-specific activation intermediates while maintaining competitive downstream performance across reasoning, understanding, long-context, retrieval, and

on-device adaptation tasks. Across multiple models and hardware settings, LARS consistently improves the memory-to-accuracy trade-off relative to existing PEFT methods, while remaining complementary to system-level optimizations. Our results highlight the importance of evaluating activation behavior, rather than parameter count alone, when designing efficient adaptation methods for resource-constrained LLM training.

7 Limitations

LARS achieves its memory efficiency by compressing sequence-level representations prior to adaptation, which can reduce fine-grained token fidelity in tasks requiring precise lexical reconstruction or generation. While LARS remains competitive across sequence labeling, extractive QA, long-context retrieval, and reasoning tasks, modest degradations can appear on generation-oriented metrics such as BLEU, especially at extremely long sequences. Additionally, LARS introduces moderately more trainable parameters than some PEFT baselines at equivalent ranks and has currently been evaluated only on models up to 8B parameters. Future work includes extending activation-efficient adaptation to larger-scale models, improving token-level expressivity through alternative sequence modeling mechanisms, and exploring hybrid architectures that better preserve local token interactions while maintaining favorable activation scaling properties.

8 Ethical Considerations

During the preparation of this manuscript, we used ChatGPT (GPT-4) for assistance with manuscript organization, text editing and phrasing, and debugging; all technical ideas, implementations, experiments, and analyses were developed by the authors. By reducing the hardware and memory requirements of adaptation, LARS can improve accessibility to fine-tuning on resource-constrained devices and lower the energy cost associated with experimentation. However, improved accessibility may also enable adaptation of language models on harmful, biased, or privacy-sensitive datasets. Furthermore, the evaluated models inherit biases and limitations present in their pretraining corpora and downstream benchmarks. We encourage responsible deployment practices, transparent reporting of fine-tuning data, and careful consideration of safety and misuse risks when adapting LLMs in low-resource settings.

References

Sumithra Bhakthavatsalam, Daniel Khashabi, Tushar Khot, Bhavana Dalvi Mishra, Kyle Richardson, Ashish Sabharwal, Carissa Schoenick, Oyvind Tafjord, and Peter Clark. 2021. Think you have solved direct-answer question answering? try arc-da, the direct-answer ai2 reasoning challenge. *Preprint*, arXiv:2102.03315.

Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. Piqa: Reasoning about physical commonsense in natural language. 34.

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *Preprint*, arXiv:1604.06174.

Yukang Chen, Shengju Qian, Haotian Tang, Xin Lai, Zhijian Liu, Song Han, and Jiaya Jia. 2024. Longlora: Efficient fine-tuning of long-context large language models. *Preprint*, arXiv:2309.12307.

Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. 2019. BoolQ: Exploring the surprising difficulty of natural yes/no questions. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 2924–2936, Minneapolis, Minnesota. Association for Computational Linguistics.

Tri Dao. 2024. Flashattention-2: Faster attention with better parallelism and work partitioning. In *The Twelfth International Conference on Learning Representations*.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. Qlora: Efficient finetuning of quantized llms. *Preprint*, arXiv:2305.14314.

Sofiane Ennadir, Levente Zólyomi, Oleg Smirnov, Tianze Wang, John Pertoft, Filip Cornell, and Lele Cao. 2025. Pool me wisely: On the effect of pooling in transformer-based models. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Bogdan Gliwa, Iwona Mochol, Maciej Biesek, and Aleksander Wawer. 2019. SAMSum corpus: A human-annotated dialogue dataset for abstractive summarization. In *Proceedings of the 2nd Workshop on New Frontiers in Summarization*, pages 70–79, Hong Kong, China. Association for Computational Linguistics.

Zeyu Han, Chao Gao, Jinyang Liu, Jeff Zhang, and Sai Qian Zhang. 2024. Parameter-efficient fine-tuning for large models: A comprehensive survey. *Transactions on Machine Learning Research*.

Jitai Hao, Weiwei Sun, Xin Xin, Qi Meng, Zhumin Chen, Pengjie Ren, and Zhaochun Ren. 2024. MEFT: Memory-efficient fine-tuning through sparse adapter. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2375–2388, Bangkok, Thailand. Association for Computational Linguistics.

Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for NLP. In *Proceedings of the 36th International Conference on Machine Learning*.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *Preprint*, arXiv:2106.09685.

Zhiqiang Hu, Lei Wang, Yihuai Lan, Wanyu Xu, Ee-Peng Lim, Lidong Bing, Xing Xu, Soujanya Poria, and Roy Lee. 2023. LLM-adapters: An adapter family for parameter-efficient fine-tuning of large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 5254–5276, Singapore. Association for Computational Linguistics.

Young D. Kwon, Rui Li, Stylianos I. Venieris, Jagmohan Chauhan, Nicholas D. Lane, and Cecilia Mascolo. 2024. Tinytrain: resource-aware task-adaptive sparse training of dnns at the data-scarce edge. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Guokun Lai, Qizhe Xie, Hanxiao Liu, Yiming Yang, and Eduard Hovy. 2017. RACE: Large-scale reading comprehension dataset from examinations. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 785–794.

Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The power of scale for parameter-efficient prompt tuning. *Preprint*, arXiv:2104.08691.

Xiang Lisa Li and Percy Liang. 2021. Prefix-tuning: Optimizing continuous prompts for generation. *Preprint*, arXiv:2101.00190.

Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. 2022. On-device training under 256kb memory.

Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin Raffel. 2022. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning. *Preprint*, arXiv:2205.05638.

NanotronResearch. 2024. Nanotron: A minimalistic library for pretraining transformer models. `https://github.com/huggingface/nanotron`.

Rui Pan, Xiang Liu, Shizhe Diao, Renjie Pi, Jipeng Zhang, Chi Han, and Tong Zhang. 2024. LISA: Layerwise importance sampling for memory-efficient large language model fine-tuning. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Richard Yuanzhe Pang, Alicia Parrish, Nitish Joshi, Nikita Nangia, Jason Phang, Angelica Chen, Vishakh Padmakumar, Johnny Ma, Jana Thompson, He He, and Samuel R. Bowman. 2022. Quality: Question answering with long input texts, yes! *Preprint*, arXiv:2112.08608.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. SQuAD: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2383–2392, Austin, Texas. Association for Computational Linguistics.

Valeria Ruscio, Umberto Nanni, and Fabrizio Silvestri. 2025. What are you sinking? a geometric approach on attention sink. *Preprint*, arXiv:2508.02546.

Maarten Sap, Hannah Rashkin, Derek Chen, Ronan Le Bras, and Yejin Choi. 2019. Social IQa: Commonsense reasoning about social interactions. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 4463–4473, Hong Kong, China. Association for Computational Linguistics.

Irene Tenison, Soumyajit Chatterjee, Fahim Kawsar, and Mohammad Malekzadeh. 2026. Adabet: Gradient-free layer selection for efficient training of deep neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 20233–20242.

Erik F. Tjong Kim Sang and Fien De Meulder. 2003. Introduction to the CoNLL-2003 shared task: Language-independent named entity recognition. In *Proceedings of the Seventh Conference on Natural Language Learning at HLT-NAACL 2003*, pages 142–147.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. Llama: Open and efficient foundation language models. *Preprint*, arXiv:2302.13971.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention is all you need. *Preprint*, arXiv:1706.03762.

Dequan Wang, Evan Shelhamer, Shaoteng Liu, Bruno Olshausen, and Trevor Darrell. 2021. Tent: Fully test-time adaptation by entropy minimization. In *International Conference on Learning Representations*.

Haoming Wang, Boyuan Yang, Xiangyu Yin, and Wei Gao. 2025a. Never start from scratch: Expediting on-device llm personalization via explainable model selection. In *Proceedings of the 23rd Annual International Conference on Mobile Systems, Applications and Services*, MobiSys '25, page 154–168, New York, NY, USA. Association for Computing Machinery.

Xubin Wang, Zhiqing Tang, Jianxiong Guo, Tianhui Meng, Chenhao Wang, Tian Wang, and Weijia Jia. 2025b. Empowering edge intelligence: A comprehensive survey on on-device ai models. *ACM Comput. Surv.*, 57(9).

Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyan Jiang, Tianle Li, Max Ku, Kai Wang, Alex Zhuang, Rongqi Fan, Xiang Yue, and Wenhu Chen. 2024. MMLU-pro: A more robust and challenging multi-task language understanding benchmark. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Sunghyeon Woo, Baeseong park, Byeongwook Kim, Minjung Jo, Se Jung Kwon, Dongsuk Jeon, and Dongsoo Lee. 2024. DropBP: Accelerating fine-tuning of large language models by dropping backward propagation. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Haoyi Wu and Kewei Tu. 2024. Layer-condensed KV cache for efficient inference of large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Bangkok, Thailand. Association for Computational Linguistics.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. 2019. HellaSwag: Can a machine really finish your sentence? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4791–4800, Florence, Italy. Association for Computational Linguistics.

Longteng Zhang, Lin Zhang, Shaohuai Shi, Xiaowen Chu, and Bo Li. 2024. LoRA-FA: Memory-efficient low-rank adaptation for large language models fine-tuning.

Qingru Zhang, Minshuo Chen, Alexander Bukharin, Pengcheng He, Yu Cheng, Weizhu Chen, and Tuo Zhao. 2023. Adaptive budget allocation for parameter-efficient fine-tuning. In *The Eleventh International Conference on Learning Representations*.

A Background and Motivation

A.1 The Fallacy of Parameter Count as a Memory Proxy

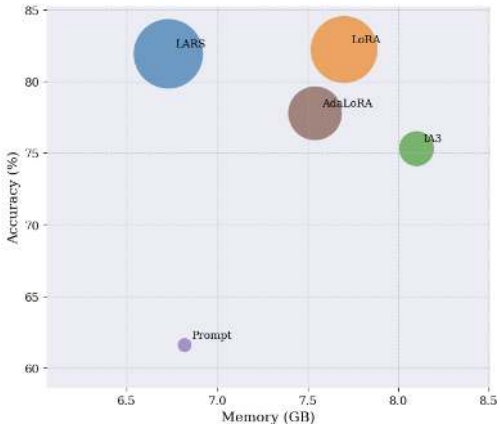


Figure 10: Accuracy vs. peak training memory (GB) for state-of-the-art PEFT methods with CP. Even with checkpointing, the disconnect remains: trainable-parameter count is a poor proxy for actual memory footprint.

While the PEFT literature traditionally uses the number of trainable parameters, $|\theta_{trainable}|$, as the primary metric for efficiency, our work demonstrates that this is a misleading proxy for actual on-device deployability. This subsection provides a detailed analysis of the empirical results presented in Figures 1, 3, and 4 that empirically validates this claim.

In Figure 1, we plot Accuracy vs. Peak Training Memory for several state-of-the-art PEFT methods. The bubble size represents the relative count of trainable parameters. A "parameter-efficient" world would show a clear correlation where smaller bubbles (fewer parameters) appear on the left (lower memory). Instead, we observe a stochastic distribution. IA3 (Liu et al., 2022) is among the most parameter-efficient methods ($\rho \approx 0.003\%$), yet it occupies the highest peak memory region (32 GB). IA3 achieves parameter efficiency by learning multiplicative vectors that scale activations across the entire hidden dimension H for every token S. Because these scaling factors must be applied to the full activation tensors at multiple points in the forward pass, the GPU must preserve these massive tensors for gradient computation, regardless of how few parameters are actually being updated.

Figure 10 repeats this analysis with GC enabled. While GC is the standard "fix" for memory issues, the figure reveals that the disconnect between pa-

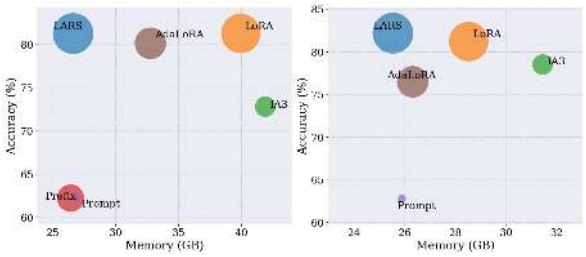


Figure 11: Accuracy vs. peak training memory (GB) for state-of-the-art PEFT methods without (left) with (right) CP. Unlike Figures 1 and 10, these are based on static padding and the disconnect remains.

rameter count and memory footprint remains unchanged. GC reduces the constant factor of memory by recomputing activations during the backward pass. However, as Figure 10 shows, the relative ordering of methods (e.g., LARS being the most efficient and IA3 being the least) remains the same. This shows that system-level tricks like GC shift the baseline but do not solve the structural flaw of token-level adaptation. Even with GC, a "small" method like LoRA can still be "large" in terms of relative peak memory.

One might argue that the memory variability in Figures 1 and 10 is an artifact of dynamic sequence lengths or varying batch sizes. Figure 11 further supports this observation by showing that the disconnect persists under static padding, where sequence length is fixed. Even when the sequence length is held constant, methods with fewer parameters often require more memory. This is due to the "activation density" of the specific adapter. Methods that insert many small adapters throughout the transformer's depth (like IA3) create more "gradient-required" nodes in the computational graph than methods that concentrate updates (like Prefix Tuning), which often under perform.

A.2 Memory Decomposition

Section 2 shows memory decomposition of a standard foundation model. The various terms in the peak memory scales as $M_{params} = \mathcal{O}(|\theta|)$ and $M_{grads} = M_{opt} = \mathcal{O}(|\theta_{trainable}|)$. However activation memory is beyond parameter count. Across layers, hidden-state activations scale at least as $M_{acts} = \mathcal{O}(BSHL)$. Self-attention introduces additional intermediate tensors. In standard implementations, attention logits scale as $\mathcal{O}(BS^2)$ per head (Vaswani et al., 2023). Memory-efficient implementations (e.g., recomputation-based(Chen et al., 2016) or fused attention kernels(Dao, 2024)) reduce this overhead but do not eliminate the need to retain layer-wise hidden states for gradient computation. Thus, regardless of attention implementation, activation memory scales at least linearly with $\mathcal{O}(BSHL)$. Generally, activation memory frequently dominates peak memory usage.

The memory decomposition in Figure 3 and Table 4 were obtained by explicitly measuring the major runtime memory components during training: model weights, gradients, optimizer states, forward-pass activations, and peak training memory. Model parameter memory was computed directly from the total number of parameters and their datatype - float32. While gradient memory was similarly estimated over all trainable parameters. Optimizer-state memory was measured from the internal AdamW optimizer tensors (e.g., momentum and variance buffers) by iterating through the optimizer state dictionary and summing the size of all stored tensors. Peak GPU memory during training was measured using *torch.cuda.max_memory_allocated()* after resetting CUDA memory statistics prior to each optimization step. We have split adapter activations into (1) adapter-specific activations from the adapter internals which LARS specifically attend to and reduce from $O(BSRL)$ to $O(BRL)$ and (2) adapter buffer, which includes activations of adapter output of shape $[BSHL]$. While adapter-specific activations remain constant at 0.035 GB as S scales from 512 to 5120, adapter buffer, which is not targeted by LARS, scales with S.

To isolate activation-related memory, we measured both forward-pass memory and total training memory. Forward-pass memory was estimated by executing inference under *torch.no_grad()* and recording the peak allocated memory, providing a proxy for runtime activations independent of backward graph retention. Additionally, hook-based activation estimates were computed using forward hooks registered across all modules, where the memory footprint of intermediate outputs was accumulated as:

$$M_{\text{acts}} = \sum_i \texttt{numel}(a_i) \times \texttt{element_size}(a_i),$$

where a_i denotes the intermediate activation tensors. The adapter-specific contribution shown in Figure 3 was then estimated by comparing the difference in peak and activation-related memory between the frozen base model and the corresponding PEFT-adapted model under identical batch size and

sequence-length settings.

A.3 Effect of Gradient Checkpointing, FlashAttention, and KV Caching

Gradient checkpointing (Chen et al., 2016) reduces memory by discarding intermediate activations during the forward pass and recomputing them during backward propagation, lowering the constant factor in M_{acts} but not its scaling with B, S, H, L. As shown in Figure 2 (Right), even with GC enabled, the slope remains identical for all token-level PEFT methods. GC trades compute (FLOPs) for memory, but it does not change the asymptotic dependence on sequence length. LARS, by contrast, reduces the rank of the tensors that must be stored or recomputed, providing a lower growth rate that compounds with the benefits of GC. The same can be observed in Table 5; while the activation memory consumption is lower than that without checkpointing, the reduction still scales with respect to S for LARS.

Flash-style memory-efficient attention kernels(Dao, 2024) further shrink attention-specific buffers (e.g., logits and softmax intermediates) and optimizes the $\mathcal{O}(S^2)$memory requirement of the attention matrix by computing it in tiles. While FlashAttention solves the memory spike of the attention weights, ($\mathbb{R}^{S\times S}$), it does nothing for the memory footprint of the hidden states ($\mathbb{R}^{B\times S\times R}$). LARS specifically targets these $\mathcal{O}(BSR)$ tensors, which FlashAttention leaves untouched.

During training, KV caching (Wu and Tu, 2024) primarily serves to accelerate autoregressive decoding and does not remove the need to store or reconstruct hidden states needed for backpropagation, offering limited relief for peak activation memory in the full-sequence training regime. Consequently, while checkpointing, optimized attention, and KV caching can substantially reduce constants or improve runtime, they do not alter the asymptotic dependence of training memory on token-level activations and therefore do not eliminate the activation bottleneck.

A.4 Constants vs. Growth Rates

Standard optimizations like GC (Chen et al., 2016) and FlashAttention (Dao, 2024) target the constant factors of M_{acts}. While GC lowers the absolute memory baseline (Figure 2 Right), the growth rate (the slope $\frac{\partial M}{\partial S}$) remains identical for LoRA and IA3. This is because they must still materialize the full activation tensors for the backward pass.

In contrast, LARS reduces this slope by operating in a sequence-pooled subspace. By decoupling the gradient-related activations from S, LARS provides a superior Pareto frontier for long-context adaptation. As S increases, LARS effectively raises the "Sequence Length Ceiling" for memory-constrained hardware, enabling adaptation where traditional PEFT methods would trigger Out-of-Memory (OOM) errors.

A.5 Context-Aware Learned Pooling

While fixed pooling is optimal for memory-constrained environments, uniform averaging may dilute fine-grained nuances in complex documents. To recover this expressivity, we introduce a learnable linear projection $w_{pool} \in \mathbb{R}^H$ that computes per-token importance scores. The pooled representation is calculated as:

$$x_{\text{pool}} = \sum_{i=1}^{S} \alpha_i X_i \quad \text{where,} \quad \alpha_i = \text{Softmax}(\text{pool_proj}(X_i)) \tag{6}$$

This strategy allows the model to prioritize critical semantic anchors while ignoring syntactic markers or padding, aligning with findings that principled pooling is essential for maintaining Transformer capacity (Ennadir et al., 2025). In our evaluations (see Table 1), this variant—denoted as LARS-LP—consistently bridges the accuracy gap on challenging reasoning tasks like MMLU-Pro, offering a tunable Pareto frontier between peak memory efficiency and predictive performance.

B Method Details

This section explains in the detail the implementation specifics and design choices of LARS. In addition to this, the repository will be made public on acceptance.

B.1 Initialization and Stability Design

While the paper focuses on the architectural formulation of LARS, the implementation introduces several stability-oriented initialization choices. The residual output projection B_{pool} is initialized to zeros, ensuring that the adapter initially behaves as an identity perturbation over the frozen backbone. Similarly, the low-rank subspace gating projections W_x and W_h are initialized to zeros so that early training begins from a neutral gating state before progressively learning rank-space modulation patterns. In contrast, the low-rank projection A_{pool} is

Table 4: Memory decomposition of LoRA and LARS at $S = 512$ and $S = 5120$. The adapter activations have been split into adapter-specific activations or internals and adapter buffer, which includes the activations of adapter outputs stored for backpropagation.

		Base Model Weights (GB)	Base Model Est. Activations (GB)	Adapter Est. Activations (GB)	Adapter Weights (MB)	Adapter Grad.(MB)	Adapter Opt. States (MB)	Adapter Buffer (GB)
S=512	LARS	4.94	1.38	0.035	33.38	33.38	67.76	0.78
	LoRA	4.94	1.38	1.39	22.58	22.58	45.16	0.83
S=5120	LARS	4.94	13.84	0.035	33.38	33.38	67.76	7.8
	LoRA	4.94	13.84	13.9	22.58	22.58	45.16	8.3

Table 5: Memory decomposition of LoRA and LARS at $S = 512$ and $S = 5120$ with gradient checkpointing. The reported estimated adapter activations include both adapter internals and buffers as they are significantly smaller in comparison to Table 4 due to gradient checkpointing.

		Base Model Weights (GB)	Base Model Est. Activations (GB)	Adapter Est. Activations (GB)	Adapter Weights (MB)	Adapter Grad.(MB)	Adapter Opt. States (MB)
S=512	LARS	4.94	0.08	0.05	33.38	33.38	67.76
	LoRA	4.94	0.08	0.13	22.58	22.58	45.16
S=5120	LARS	4.94	0.78	0.48	33.38	33.38	67.76
	LoRA	4.94	0.78	1.33	22.58	22.58	45.16

initialized with a small Gaussian distribution (std = 0.01) to preserve low-magnitude feature projections during the initial optimization phase. These initialization strategies improve optimization stability and prevent large early-stage perturbations to the pretrained model representations.

B.2 Rank-Space Pipeline

The implementation of LARS realizes the proposed low-rank subspace adaptation entirely within the compressed rank manifold of dimensionality R. After sequence pooling as in Stage 1 of Figure 12, the pooled representation is projected into rank space and normalized using LayerNorm before gating. The gated features are then processed through three lightweight operations: (i) inter-rank mixing using the learnable matrix M_mix, (ii) multiplicative feature modulation (g_mix * h_norm), and (iii) a residual feed-forward transformation with GeLU activations. Importantly, all non-linear processing occurs in the low-dimensional subspace rather than the original hidden dimension H, preserving the activation-efficient design objective of LARS while improving representational expressivity.

B.3 Residual Broadcasting

A key implementation detail is how the adapter output is reintegrated into the token sequence shown in Stage 3 of Figure 12. The adapter produces a single sequence-level residual vector of shape $[B, H]$, which is broadcast across the sequence dimension using *unsqueeze(1)* before addition to the frozen backbone activations. This avoids constructing token-level adapter activations of shape [B, S, R] during the forward and backward passes. Consequently, the trainable adapter intermediates remain independent of sequence length, enabling the reduction of adapter activation complexity from $\mathcal{O}(BSRL)$ to $\mathcal{O}(BRL)$ as discussed.

C Additional Setup Details

C.1 Tasks and Datasets

To rigorously evaluate the memory-to-accuracy trade-off of LARS (Low-memory Activation-Rank Subspace), we utilize a diverse evaluation suite categorized into three primary domains. These tasks are designed to stress-test the model's ability to maintain high-resolution signals despite the sequence-pooling mechanism.

1. Commonsense Reasoning We evaluate reasoning capabilities using five core benchmarks from the LLM-adapters family. After fine-tuning, these tasks require the model to perform logical inference over everyday scenarios.

- BoolQ: A reading comprehension dataset of 15,942 naturally occurring yes/no questions
- PIQA (Physical Interaction QA): Tests the model's understanding of physical objects and their interactions
- SIQA (Social Interaction QA): Focuses on reasoning about social interactions and social commonsense
- HellaSwag: A dataset that challenges the model to complete sentences by predicting the most plausible continuation of a scene

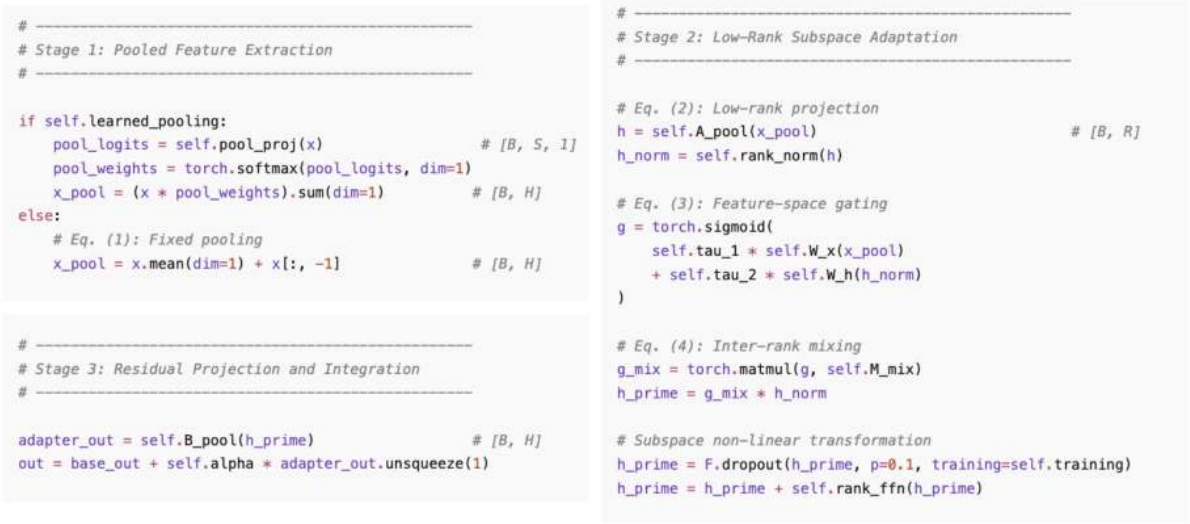


Figure 12: Code snippets corresponding to the 3 stages in LARS - Sequence pooling, low-rank subspace modulations, and residual projection via broadcasting.

- ARC-c (AI2 Reasoning Challenge): A "Challenge" set consisting of difficult, grade-school science questions that require more than simple retrieval

2. General Understanding (MMLU-Pro)

- Subjects: Economics, Biology, Physics, Health, and Math. These subjects are chosen at random.
- Task: These subjects measure the model's ability to maintain competitive accuracy

3. Long-Context & Retrieval Analysis A critical component of our evaluation is the "Sequence Length Ceiling" test, where we evaluate if LARS can handle long inputs without the linear memory growth typical of LoRA or IA3.

- QuALITY: A multiple-choice QA dataset featuring long input texts that require deep reasoning
- RACE: A large-scale reading comprehension dataset derived from middle and high school English exams
- Nanotron NIAH (Needle-In-A-Haystack): We perform passkey retrieval tasks across context lengths of 1024, 16k, and 32k tokens. This validates that LARS's pooling mechanism does not cause "catastrophic information loss" and can retrieve localized, high-resolution signals with near-perfect accuracy

3. Token-Level Fidelity To evaluate whether LARS generalizes beyond classification and retrieval tasks, we additionally benchmark it on structured prediction, span extraction, and generative summarization workloads.

- CoNLL-2003: A sequence labeling benchmark for named entity recognition (NER), where models must preserve token-level contextual representations across long token sequences. This evaluates whether LARS's pooled adaptation mechanism can maintain fine-grained token discrimination required for accurate labeling.
- SQuAD: An extractive question answering benchmark requiring precise span localization within a context passage. This tests whether LARS can retain localized contextual information necessary for accurate answer boundary prediction despite compressed adaptation activations.
- SAMSum: A dialogue summarization benchmark focused on conversational summarization. This evaluates whether LARS can preserve long-range conversational dependencies and salient contextual information needed for coherent abstractive summaries while maintaining memory efficiency.

C.2 Hyperparameters and Fine-Tuning Setup

Models We evaluate two base models - Llama 3.2 1B and Qwen2.5 7B Instruct. The Llama model

is used for most experiments, while Qwen serves as a larger model comparison. For classification tasks we use sequence classification heads, whereas long-context, reasoning, and token-level fidelity benchmarks are trained using causal language modeling. These models are open access models available on Huggingface.

Hyperparameters All models are fine-tuned for 1500 steps using the AdamW optimizer with weight decay 0.01, Cosine decay with 100 warmup steps, and gradient clipping 1.0. Other baseline-specific hyperparameters like LoRA α, LoRA dropout, AdaLoRA init rank etc were chosen as the default values provided by HuggingFace PEFT Library. Training uses dynamic token counting to monitor throughput and GPU utilization. Other fine-tuning dataset specific hyperparameters are as listed below:

- BoolQ: batch size (BS) = 8, accumulation steps (AS) = 4
- PIQA: BS = 8, AS = 4
- SIQA: BS = 8, AS = 4
- HellaSwag: BS = 2, AS = 16
- ARC-c: BS = 8, AS = 4
- MMLU-Pro: BS = 2, AS = 8
- QuALITY: BS = 2, AS = 16
- RACE: BS = 2, AS = 16
- Nanotron NIAH: BS = 1

The above dataset-specific BS and AS were chosen specifically to run the experiments on at max a single GPU of L40S for Llama or H200 for Qwen models.

Learning rates for each baseline including LARS were obtained via tuning on BoolQ and the same were used for all experiments as the primary contribution of this work is memory reduction with comparable, which is determined by architectural design and is invariant to hyperparameter choices. While tuning can influence accuracy, it does not affect the observed memory reductions. To ensure that this choice do not adversarially affect the baselines, we conducted learning rate sweeps (5 values) for LARS, LoRA, AdaLoRA, and IA3 on SIQA and HellaSwag as shown in Table 6. The results show that the learning rates used in the paper (italics) achieve performance close to the best (bold) for each method, with only minor deviations.

While variance across random seeds can influence downstream accuracy metrics, the primary contribution of this work is reducing the sequence-length scaling of adapter-specific activation memory, which is determined by the architectural structure of the method and remains invariant to initialization or seed choice. Consequently, our evaluation prioritizes consistent memory, throughput, and activation-scaling comparisons across methods and hardware settings. We nevertheless include additional experiments across multiple datasets, tasks, and model scales to demonstrate that the observed performance trends are stable and not isolated to a single benchmark configuration. Additionally, we recommend careful tuning for applications where accuracy gains are of extreme importance.

Hardware For our experiments we had access to 4 hardwares:

- NVIDIA L40S: All Llama 1B experiments were run on this. GPU memory available was 45GB.
- NVIDIA H200: All Qwen 7B and model scaling experiments were run on this. GPU memory available was 145GB.
- AMD EPYC: This powerful CPU was used only for Table 2
- Raspberry Pi 5: This 8GB edge device was used only validation

Memory, Throughput, and Power Measurement Methodology To accurately measure memory efficiency and model throughput, we implemented the following procedure:

- GPU Peak Memory: Before training, GPU memory statistics are reset with *torch.cuda.reset_peak_memory_stats()* and *torch.cuda.empty_cache()*. During training, peak memory usage is logged after each optimizer step using *torch.cuda.max_memory_allocated()*.
- CPU Peak Memory: Peak CPU memory is tracked using Python's *psutil* library - *process = psutil.Process(); cpu_mem_mb = process.memory_info().rss / 1e6*. The maximum of this provides the peak memory usage.

Table 6: Learning rate sweep for LARS, LoRA, AdaLoRA, and IA3 on SIQA and HellaSwag.

	SIQA					Hellaswag				
lr	**1e-2**	**1e-3**	**1e-4**	**1e-5**	**1e-6**	**1e-2**	**1e-3**	**1e-4**	**1e-5**	**1e-6**
LARS	76.61	**78.85**	*78.04*	78.75	70.21	25	85.62	***90.2***	90.2	75.62
LoRA	42.94	76.25	*78.77*	**79.2**	75	30	70.11	***90.45***	90.21	75.62
IA3	71.25	*73.75*	**74.92**	48.75	45	85.62	***85.77***	85.26	65.2	45
AdaLoRA	71.25	**78.5**	76.61	63.75	25	30	***90.21***	85.85	75.2	25

- Throughput Measurement: Token throughput is computed dynamically as

$$\text{tokens/sec} = \frac{\text{total tokens processed per AS}}{\text{elapsed wall-clock time in sec.}}$$

  For inference throughput, we consider the total tokens in the evaluation set and total time taken for evaluation.

- Power Measurement: For CPU experiments, average power consumption is estimated using real-time CPU utilization from the *psutil* library together with a linear power model:

$$P_{\text{cpu}} = P_{\text{idle}} + u \cdot (P_{\text{max}} - P_{\text{idle}})$$

  where $u \in [0, 1]$ denotes CPU utilization, P_{idle} is the idle CPU power, and P_{max} is the maximum processor power draw.

- Energy Measurement: Energy consumption per optimization step is computed as:

$$E_{\text{step}} = P_{\text{cpu}} \times t_{\text{step}}$$

  where P_{cpu} is the average power consumption during the step and t_{step} is the elapsed wall-clock time for that optimization step.

On the resource-constrained Raspberry Pi 5 (8GB), LARS achieved a memory footprint of 3.008 GB and a throughput of 3.20 tokens/sec, demonstrating performance parity with established methods while maintaining a lower memory profile than LoRA and IA3. This efficiency scales effectively to more performant CPUs. On the AMD EPYC 9474F, LARS consistently outperformed LoRA in memory efficiency. Notably, at a sequence length of 1024, LARS showed nearly 2× reduction—while maintaining competitive throughput. *These results confirm that LARS is a highly viable candidate for deploying large-scale activations on CPU-only environments, including edge devices.*

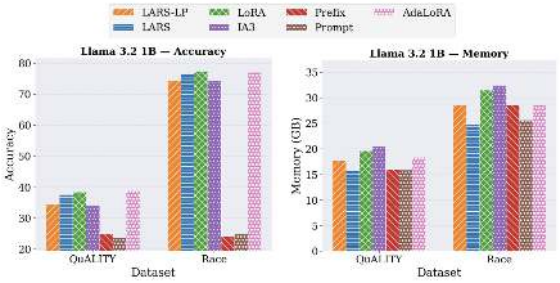


Figure 13: Comparison of accuracy and memory on long context tasks using QuALITY and Race datasets for models Llama 3.2 1B model.

D Additional Results

D.1 Long Context Results

Figures 7 and 13 illustrate the memory consumption and predictive accuracy across two long-context benchmarks, QuALITY and Race, using Qwen 7B models and Llama 3.2 1B respectively. We compare our proposed LARS and LARS-LP methods against standard PEFT baselines, including LoRA, IA3, Prefix Tuning, Prompt Tuning, and AdaLoRA. Across both model scales and datasets, LARS consistently demonstrates superior memory efficiency relative to all other high-performing adapters. In terms of accuracy, LARS and LARS-LP achieve performance parity with LoRA and AdaLoRA, despite their reduced memory requirements across both datasets and models. Collectively, these results indicate that *LARS provides a favorable trade-off for long-context applications, offering the high-rank representation power of LoRA with a memory profile more closely resembling (or bettering) more restrictive parameter-efficient methods.*

D.2 Latency of LARS

Figure 14 follows the inference throughput shown in Figure 9. The results demonstrate that LARS consistently achieves the lowest inference latency (the fastest processing time) at both sequence lengths of 256 and 1024. As expected, latency

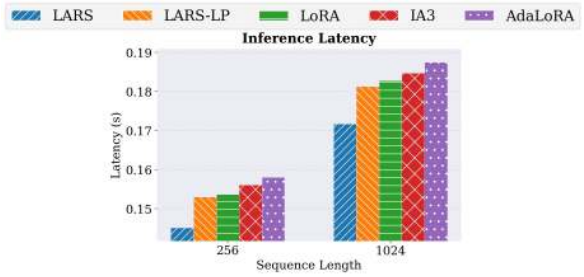


Figure 14: Inference latency of LARS and other baselines.

increases for all methods as the sequence length grows, but LARS maintains its performance advantage, while AdaLoRA consistently exhibits the highest latency. This indicates that LARS is not only memory-efficient, but also faster in terms of inference speed.

D.3 Impact of Target Modules

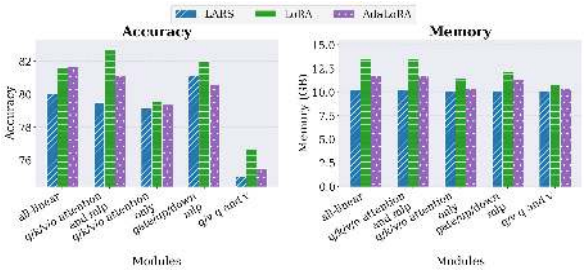


Figure 15: Comparison of Memory Usage (GB) and Accuracy across different target modules for the LARS, LoRA, and IA3 fine-tuning methods.

Figure 15 evaluates how targeting different architectural modules—such as attention layers, MLP components, or all linear layers—affects the performance and resource usage of LARS, LoRA, and AdaLoRA. The accuracy plot on the left shows that while LoRA often achieves the highest overall accuracy, LARS remains a strong competitor, particularly when targeting the "gate/up/down mlp" modules. On the right, the memory plot highlights LARS's primary advantage: its GPU footprint remains remarkably stable and low (around 10 GB) regardless of which modules are modified. In contrast, LoRA and AdaLoRA exhibit much higher memory demands and greater sensitivity to the specific modules being tuned, making LARS the more predictable choice for hardware-constrained environments..

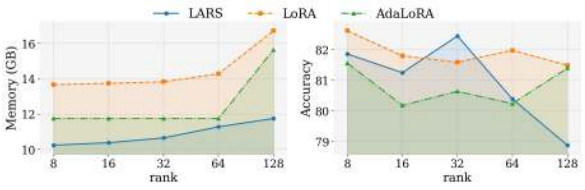


Figure 16: Comparison of Memory Usage (GB) and Accuracy for the LARS, LoRA, and AdaLoRA methods across different ranks (r)

D.4 Effect of Rank

Figure 16 compares the trade-offs between hardware resource efficiency and model performance for three fine-tuning methods: LARS, LoRA, and AdaLoRA. The left plot illustrates that LARS maintains a consistently low and stable GPU memory footprint even as the rank increases. In contrast, both LoRA and AdaLoRA require significantly more memory, which scales up to nearly 17 GB at higher ranks. However, the accuracy plot on the right reveals a performance trade-off; while LoRA and AdaLoRA maintain relatively stable accuracy across all rank values, LARS is highly sensitive to the rank configuration. It reaches a competitive peak at $r = 32$ but suffers a sharp decline in accuracy after $r = 128$. Additionally, LARS demonstrates superior inference speed across all tested ranks as shown in Appendix D.4.1. These results suggest that while *LARS is ideal for memory-constrained environments*, it requires more precise hyperparameter tuning than its counterparts.

D.4.1 Throughput with Increasing Rank

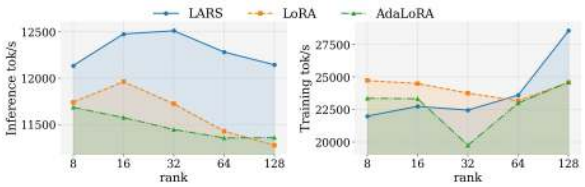


Figure 17: Inference and training throughput of LARS and baselines with increasing ranks.

Figure 17 adds onto Figure 16, from a throughput efficiency measured in tokens per second. The left plot shows that LARS consistently achieves higher inference throughput than the baselines, maintaining a clear speed advantage even as the rank increases. On the right, while LARS begins with slightly lower training throughput at smaller ranks, it experiences a dramatic performance surge at $r = 128$, significantly outperforming both LoRA

and AdaLoRA, which tend to slow down as rank complexity grows.

D.5 Effect of Dataset Size

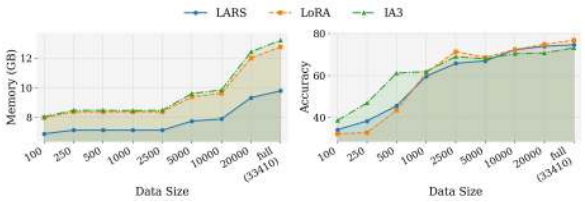


Figure 18: Comparison of Memory Usage (GB) and Accuracy across increasing Data Sizes for the LARS, LoRA, and IA3 fine-tuning methods.

Figure 18 examines how training data size affects both memory consumption and model accuracy across the LARS, LoRA, and IA3 methods. The left plot highlights that LARS is the most resource-friendly option, consistently requiring the least amount of GPU memory as the dataset scales. On the right, the accuracy plot shows that while IA3 takes an early lead in low-data scenarios, LoRA eventually reaches the highest accuracy at the "full" dataset size. *LARS serves as a strong middle ground, providing competitive accuracy while maintaining a significantly smaller memory footprint* than its counterparts throughout the entire scaling process.

D.6 Needle-In-A-Haystack (NIAH) Retrieval

Table 7: Accuracy of Needle-In-A-Haystack (NIAH) style retrieval of a passkey.

	LARS	LoRA	AdaLoRA	IA3
1024	99.2	99.2	99.2	99.2
16k	98.4	98.4	98.4	98.4
32k	72.6	73.2	72.6	72.6

To evaluate whether LARS's sequence-pooling mechanism results in catastrophic information loss, we conducted a passkey retrieval task in a Needle-In-A-Haystack style across context lengths of 1024, 16k, and 32k tokens using the Nanotron NIAH dataset (NanotronResearch, 2024). LARS, LoRA, AdaLoRA, and IA3 achieved near-perfect accuracy at lengths up to 16k; however, performance across all methods declined at the 32k context length as shown in Table 7. These results demonstrate that LARS's memory efficiency maintains parity with standard PEFT methods.

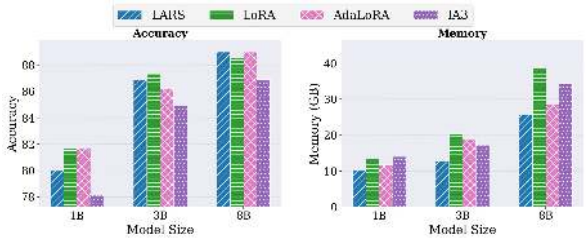


Figure 19: Impact of different model sizes on Accuracy and Memory Usage (GB) of LARS and baselines.

D.7 Scaling across Model Sizes

We further evaluate LARS on Llama models ranging from 1B to 8B parameters to examine whether its memory advantages persist across scales. As shown in Figure 19, LARS consistently reduces peak training memory while maintaining accuracy comparable to LoRA across all model sizes. While recent work often focuses on much larger models, smaller models remain the primary target for on-device and resource-constrained adaptation, where memory and compute budgets are limited. Evaluating within this range therefore reflects the practical deployment regime for edge devices, demonstrating that LARS can lower the hardware requirements for adapting models in such settings.

E Complementary methods

In this section we explore the complementary nature of existing widely-used memory-reduction methods including systems-level optimization methods like FlashAttention (Dao, 2024), Gradient Checkpointing (Chen et al., 2016), as well as algorithmic methods like DropBP(Woo et al., 2024) and LoRA-FA(Zhang et al., 2024). Note that these methods are complementary, not competing, to LARS. These methods do not specifically target the sequence-length scaling of adapters and the adapter's activation memory that grows with it, which is the core contribution of LARS. Additionally, we show that LARS can be combined with these methods for additional resource reduction.

E.1 FlashAttention

Figure 20 examines the impact of FlashAttention on accuracy and memory consumption across the LARS, LoRA, AdaLoRA, and IA3 methods. The left plot indicates that while FlashAttention consistently boosts accuracy for all methods, its impact on LARS is particularly significant, helping it bridge the performance gap with LoRA and

Table 8: Accuracy and memory consumption of LoRA and LARS in combination with DropBP(Woo et al., 2024) and LoRA-FA (Zhang et al., 2024). These results reinstate theire complementary nature to the proposed method, LARS.

	LoRA		LARS		LoRA +FA		LARS +FA		LoRA+ DropBP		LARS+ DropBP	
	acc (%)	mem (GB)	acc (%)	mem (GB)	acc (%)	mem (GB)	acc (%)	mem (GB)	acc (%)	mem (GB)	acc (%)	mem (GB)
BoolQ	83	13.67	81.1	10.24	81.77	10.99	81.94	9.97	70.72	11.89	74.79	9.67
SIQA	77.28	20.55	76.74	14.45	77.17	14.52	77.04	10.58	73.44	12.34	73.95	11.85

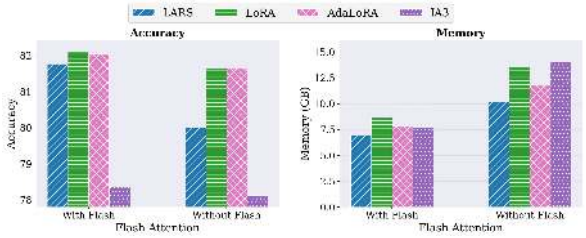


Figure 20: Accuracy and Memory Consumption of LARS and baselines with and without FlashAttention.

AdaLoRA. On the right, the memory plot highlights that FlashAttention substantially reduces the GPU footprint for every technique. Notably, LARS maintains its status as the most resource-efficient option in both scenarios, requiring the least amount of memory whether FlashAttention is enabled or not.

E.2 Gradient Checkpointing

Figure 10 provides a comparison of the accuracy vs. memory trade-off for several parameter-efficient fine-tuning (PEFT) methods, including LARS, LoRA, AdaLoRA, IA3, and Prompt tuning under gradient checkpointing. Compared to Figure 1, while gradient checkpointing slightly reduced accuracy significantly saves memory for all methods, LARS still consumes relatively lower memory among all baselines while maintaining comparable performance. Similar results can be expected from methods like MEFT (Hao et al., 2024) which trades off recomputation for memory savings. Notably, LARS maintains its status as the most resource-efficient option in both scenarios, requiring the least amount of memory whether Gradient Checkpointing is enabled or not.

E.3 Algorithmic Methods

Besides systems-level optimization methods described above, there exists some prior work like DropBP (Woo et al., 2024) and LoRA-FA (Zhang et al., 2024) that target activation memory from the algorithmic side. Specifically, DropBP drops backprop from randomly chosen pre-fixed fraction of layers to save activation memory and LoRA-FA freezes low-rank A matrix in LoRA style adapters. These methods are complementary to the asymptotic reduction of adapter activation memory and can be combined with LARS easily to achieve better efficiency. Figure 8 confirms that (i) plain LARS is capable of achieving more memory reduction than LoRA+FA[2] and LoRA+DropBP[3] and (ii) LARS can be combined with these methods to achieve more efficiency.