Memory-Efficient Structured Backpropagation for On-Device LLM Fine-Tuning

Juneyoung Park, Yuri Hong, Seongwan Kim†, Jaeho Lee†

OptAI Inc.

{jyoung.park, yri.hong, swan.kim, jaeho.lee}@opt-ai.kr

Abstract

On-device fine-tuning enables privacy-preserving personalization of large language models, but mobile devices impose severe memory constraints, typically 6–12GB shared across all workloads. Existing approaches force a trade-off between exact gradients with high memory (MeBP) and low memory with noisy estimates (MeZO). We propose Memory-efficient Structured Backpropagation (MeSP), which bridges this gap by manually deriving backward passes that exploit LoRA's low-rank structure. Our key insight is that the intermediate projection $h = xA$ can be recomputed during backward at minimal cost since rank $r \ll d_{in}$, eliminating the need to store it. MeSP achieves 49% average memory reduction compared to MeBP on Qwen2.5 models (0.5B–3B) while computing mathematically identical gradients. Our analysis also reveals that MeZO's gradient estimates show near-zero correlation with true gradients (cosine similarity ≈0.001), explaining its slow convergence. MeSP reduces peak memory from 361MB to 136MB for Qwen2.5-0.5B, enabling fine-tuning scenarios previously infeasible on memory-constrained devices. We provide an example implementation of our proposed method at https://github.com/crinex/acl-mesp

1 Introduction

On-device fine-tuning of large language models has emerged as a promising approach for privacy-preserving personalization, enabling users to adapt models to their data without transmitting sensitive information to cloud servers. However, this capability remains largely theoretical for most mobile devices. Even with parameter-efficient methods like LoRA (Hu et al., 2022), the memory footprint of intermediate activations during backpropagation exceeds the 6–12GB typically available on mobile devices. For example, fine-tuning a modest 0.5B parameter model at sequence length 256 requires over 360MB of peak memory under standard memory-efficient backpropagation, a significant burden when this memory must be shared with the operating system and other applications.

Existing approaches force practitioners into an unsatisfying trade-off. Zeroth-order methods like MeZO (Malladi et al., 2023) achieve inference-level memory usage by estimating gradients through forward passes alone, but their gradient estimates exhibit variance proportional to parameter count, requiring 10–100× more iterations to converge. Memory-efficient backpropagation (MeBP) (Apple Machine Learning Research, 2024) computes exact gradients with gradient checkpointing, but delegates tensor lifecycle decisions to automatic differentiation frameworks, which store more intermediates than mathematically necessary.

Interestingly, we observe that LoRA's low-rank structure offers an unexploited opportunity for memory optimization. The intermediate projection $h = xA$ has shape $[\text{batch}, \text{seq}, r]$ where the rank r is typically 8–32. Since $r \ll d_{in}$, recomputing h during the backward pass costs far less than storing it across all LoRA layers. Existing methods treat this computation as a black box, but by manually deriving backward passes, we can determine precisely which tensors to store, when to release them, and what to recompute.

Can we achieve the memory efficiency of zeroth-order methods while preserving the gradient accuracy of first-order backpropagation? In this paper, we answer affirmatively with Memory-efficient Structured Backpropagation (MeSP). Our approach processes transformer blocks (Vaswani et al., 2017) sequentially, storing only block outputs during forward and recomputing all intermediates during backward. By deriving explicit backward passes for each transformer component, MeSP provides

†Corresponding Author

fine-grained control over tensor lifecycles that automatic differentiation cannot achieve. As we demonstrate in Section 5, this reduces peak memory by 62% for Qwen2.5-0.5B while computing mathematically identical gradients.

Our contributions are as follows:

- We propose MeSP, a memory-efficient training algorithm that achieves 49% average memory reduction compared to MeBP while preserving first-order gradient accuracy through manually derived backward passes.
- We demonstrate that MeSP reduces peak memory from 361MB to 136MB (62% reduction) for Qwen2.5-0.5B on Apple Silicon, with only 28% computational overhead.
- We reveal that MeZO's gradient estimates are essentially uncorrelated with true gradients (cosine similarity ≈0.001), providing new insight into why zeroth-order methods converge slowly.
- We release our MLX-based implementation for reproducible on-device training research.

2 Related Work

Memory-Efficient Training. Gradient checkpointing (Chen et al., 2016) reduces memory by recomputing activations during backward, achieving $O(\sqrt{n})$ memory complexity. FlashAttention (Dao et al., 2022; Dao, 2023) applies this principle to attention, recomputing softmax weights rather than storing the full attention matrix. MeZO (Malladi et al., 2023) takes a more radical approach, eliminating backpropagation entirely through zeroth-order gradient estimation. However, the variance of these estimates scales with parameter count, degrading convergence for large models. MeBP (Apple Machine Learning Research, 2024) combines gradient checkpointing with LoRA on Apple Silicon, but relies on automatic differentiation which implicitly retains more tensors than necessary. *In contrast, our work provides explicit control over tensor lifecycles by manually deriving backward passes, achieving memory efficiency comparable to zeroth-order methods while preserving first-order gradient accuracy.*

Parameter-Efficient Fine-Tuning. Adapter methods (Houlsby et al., 2019) insert trainable bottleneck modules between frozen layers. LoRA (Hu et al., 2022) parameterizes weight updates as low-rank matrix products $\Delta W = AB$, reducing trainable parameters from $O(dk)$ to $O(r(d+k))$ where $r \ll d, k$. QLoRA (Dettmers et al., 2023) combines LoRA with 4-bit quantization, achieving up to $4\times$ memory reduction for model weights. Despite these advances in reducing parameter count, the memory footprint of intermediate activations during training remains a bottleneck for on-device deployment. *Unlike previous approaches that focus solely on parameter efficiency, we exploit LoRA's low-rank structure to optimize activation memory by recomputing the small intermediate projection $h = xA$ rather than storing it.*

On-Device Training. Lin et al. (Lin et al., 2022) demonstrated training under 256KB memory through aggressive quantization and sparse updates. PockEngine (Xu et al., 2023) optimizes computational graphs for mobile deployment. Apple's MLX (Apple Machine Learning Research, 2023) provides native machine learning support on Apple Silicon with unified memory architecture. *We build upon this ecosystem, contributing memory-efficient training algorithms specifically designed to exploit the unique properties of LoRA fine-tuning on unified memory devices.*

3 Background

This section provides the technical foundation for our approach, covering Low-Rank Adaptation, zeroth-order optimization, and memory-efficient backpropagation.

3.1 Low-Rank Adaptation (LoRA)

Low-Rank Adaptation (Hu et al., 2022) is a parameter-efficient fine-tuning method that freezes pre-trained weights and introduces trainable low-rank decomposition matrices. For a linear layer with frozen weight $W_0 \in \mathbb{R}^{d_{in} \times d_{out}}$, the forward computation becomes

$$y = xW_0 + s \cdot xAB, \tag{1}$$

where $A \in \mathbb{R}^{d_{in} \times r}$ and $B \in \mathbb{R}^{r \times d_{out}}$ are trainable matrices, $r \ll \min(d_{in}, d_{out})$ is the rank, and $s = \alpha/r$ is a scaling factor. This formulation reduces the number of trainable parameters from $d_{in} \times d_{out}$ to $r \times (d_{in} + d_{out})$.

Given the upstream gradient $\partial L/\partial y$, the gradients for the LoRA parameters are computed as

$$\frac{\partial L}{\partial B} = s \cdot h^\top \frac{\partial L}{\partial y}, \qquad \frac{\partial L}{\partial A} = x^\top \frac{\partial L}{\partial h}, \tag{2}$$

where $h = xA$ is the intermediate projection and

$$\frac{\partial L}{\partial h} = s \cdot \frac{\partial L}{\partial y} B^\top. \tag{3}$$

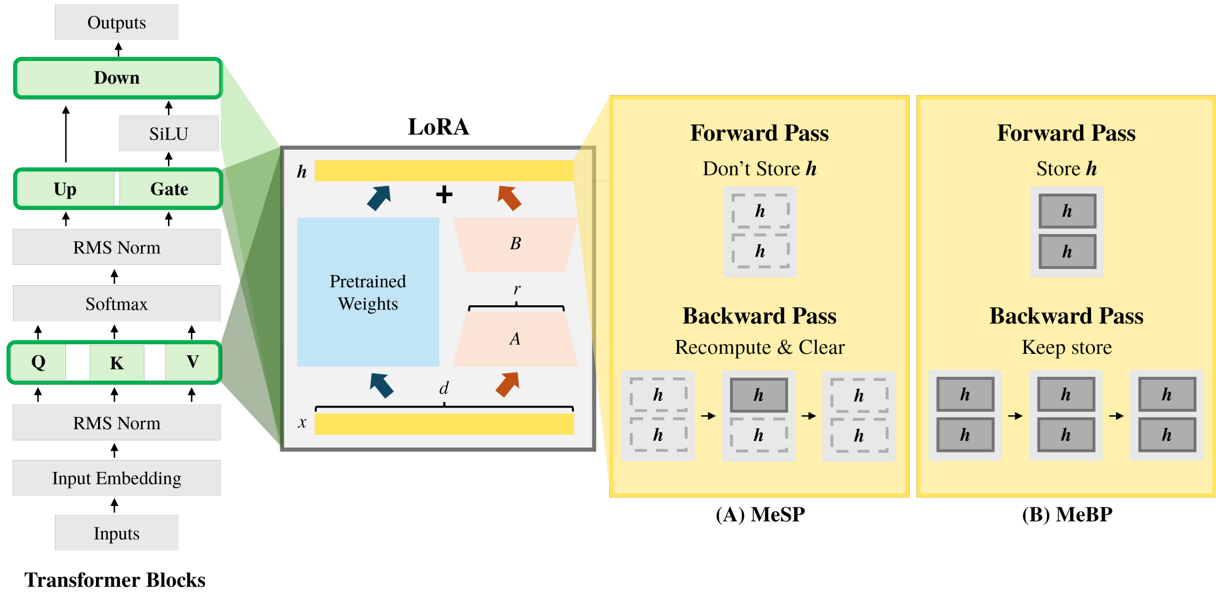

Figure 1: Visualization of LoRA fine-tuning with Memory-efficient Structured Backpropagation (MeSP) compared to Memory-efficient Backpropagation (MeBP). (A) In MeSP, the intermediate LoRA projection $h = xA$ is not cached in the forward pass and is recomputed in the backward pass, then discarded immediately after being used to compute the gradients of A and B. This keeps only a few essential activations in memory at any time, substantially reducing peak memory usage. (B) In MeBP, the intermediate projection h is kept as a forward activation for each LoRA module and later loaded again during backpropagation to compute the gradients. Because these h tensors remain in memory, this leads to higher peak memory usage than MeSP.

A key observation is that h has shape $[\text{batch}, \text{seq}, r]$. Since r is typically small, in the range of 8 to 32, recomputing h during the backward pass requires only $O(\text{batch} \times \text{seq} \times d_{in} \times r)$ operations. This cost is substantially lower than the memory access overhead incurred by storing and retrieving h for long sequences.

3.2 Zeroth-Order Optimization (MeZO)

Zeroth-order optimization estimates gradients without backpropagation by using finite differences. MeZO (Malladi et al., 2023) approximates the gradient as

$$\nabla L(w) \approx \frac{L(w + \epsilon z) - L(w - \epsilon z)}{2\epsilon} \cdot z, \tag{4}$$

where $z \sim \mathcal{N}(0, I)$ is a random perturbation vector and ϵ is a small constant. The method requires two forward passes per update, one with positively perturbed parameters and one with negatively perturbed parameters. The resulting gradient estimate is then used to update the weights.

This approach achieves memory consumption equivalent to inference, as no intermediate activations need to be stored for backpropagation. However, the variance of the gradient estimate scales linearly with the parameter dimensionality d, that is, $\text{Var}[\hat{g}] = O(d)$. For models with billions of parameters, this high variance severely degrades the signal-to-noise ratio of the gradient estimates and typically requires 10 to 100 times more iterations to achieve loss reduction comparable to first-order methods (Malladi et al., 2023).

3.3 Memory-Efficient Backpropagation (MeBP)

Memory-efficient backpropagation combines gradient checkpointing with automatic differentiation to reduce peak memory usage while computing exact gradients. During the forward pass, only layer outputs are stored as checkpoints, while intermediate activations within each layer are discarded. During the backward pass, the forward computation for each layer is re-executed to regenerate the required intermediates before computing gradients.

This approach reduces memory complexity from $O(L \times I)$ to $O(L \times O + I)$, where L denotes the number of layers, I denotes the size of intermediate activations per layer, and O denotes the output size per layer. Since $O \ll I$ for typical transformer architectures, this leads to substantial memory savings.

However, MeBP relies on automatic differentiation frameworks such as `mx.grad` in MLX or

Model	Method	Mem (MB)	Time (s)	Red.
0.5B	MeBP	360.8	0.68	–
	MeZO	243.0	0.51	33%
	MeSP	**136.2**	0.86	**62%**
1.5B	MeBP	516.2	1.66	–
	MeZO	376.0	1.21	27%
	MeSP	**262.6**	2.17	**49%**
3B	MeBP	637.6	3.21	–
	MeZO	479.2	2.24	25%
	MeSP	**368.4**	4.09	**42%**

Table 1: Memory and time comparison at sequence length 256. MeSP achieves 42–62% memory reduction vs MeBP while computing exact gradients.

Method	128	256	512	1024
MeBP	252.7	360.8	582.4	1050.3
MeZO	199.0	243.0	336.0	524.0
MeSP	**110.7**	**136.2**	**245.8**	**513.6**
Memory Reduction vs MeBP				
MeZO	21%	33%	42%	50%
MeSP	**56%**	**62%**	**58%**	**51%**

Table 2: Peak memory (MB) vs sequence length on Qwen2.5-0.5B.

`torch.autograd` in PyTorch to compute gradients within each checkpointed segment. These frameworks treat the computation as a black box and implicitly determine which tensors to retain for gradient computation. As a result, more intermediate tensors may be stored than are mathematically necessary, since the framework cannot exploit domain-specific structure such as the low-rank property of LoRA projections.

4 Method

4.1 Core Idea: Trading Computation for Memory

The fundamental insight behind MeSP is simple: *small tensors are cheaper to recompute than to store*. In LoRA, the intermediate projection $h = xA$ has dimensions $[\text{batch}, \text{seq}, r]$ where the rank r is typically 8–32. Storing h across all LoRA layers (7 per transformer block $\times$ 24 blocks = 168 layers for Qwen2.5-0.5B) accumulates significant memory. However, recomputing h requires only a single matrix multiplication with the small matrix A, which is fast compared to the memory access overhead of storage and retrieval.

This observation leads to our strategy: during the forward pass, we store only the outputs of each transformer block, just enough to restart computation during backward. During the backward pass, we recompute all intermediates on-demand, process one layer at a time, and immediately release memory after computing gradients. At any point, only a single layer's intermediates reside in memory.

4.2 Explicit Gradient Computation for LoRA

We now formalize how gradients are computed without storing the intermediate h. Consider a LoRA layer with frozen weight W_0, trainable matrices A and B, and scaling factor $s = \alpha/r$:

$$y = xW_0 + s \cdot xAB \tag{5}$$

Given the upstream gradient $g = \partial L/\partial y$, the parameter gradients can be expressed as:

$$\frac{\partial L}{\partial B} = h^\top (s \cdot g), \qquad \frac{\partial L}{\partial A} = x^\top (s \cdot gB^\top) \tag{6}$$

where $h = xA$. The key observation is that h *appears only in the gradient for* B, and we can recompute it as xA using the input x (which we must store anyway) and the weight A (which is a model parameter). This eliminates the need to store h during forward. A complete derivation establishing mathematical equivalence with automatic differentiation is provided in Appendix A.

4.3 Layer-by-Layer Processing

Our algorithm processes transformer blocks sequentially, maintaining minimal memory footprint throughout training.

Forward Phase. Each transformer block receives its input and computes the full forward pass: layer normalization, Q/K/V projections with LoRA, multi-head attention, output projection, feed-forward network with gated activation, and residual connections. Only the final output of each block is stored in a checkpoint dictionary. All other intermediates are discarded immediately.

Backward Phase. We iterate through blocks in reverse order. For each block, we retrieve the stored input from the checkpoint, re-execute the forward computation to regenerate intermediates, compute gradients for all LoRA parameters, update parameters immediately with the optimizer, and then explicitly deallocate all intermediates and clear the GPU cache before proceeding to the next block.

This design ensures that peak memory occurs during the backward pass of a *single* layer, rather

Layer	Cosine Sim	Sign Agree	Rel. Error
0	0.003	48.4%	171
5	0.000	48.4%	2155
10	-0.000	48.4%	1906
15	-0.001	48.4%	2351
20	-0.000	48.4%	3590
23	0.001	48.5%	1692
Avg	0.001	48.4%	1978

Table 3: MeZO gradient quality vs exact gradients on Qwen2.5-0.5B.

Method	r=4	r=8	r=16	r=32
MeBP	355.2	360.8	372.4	395.8
MeZO	215.0	243.0	299.0	411.0
MeSP	**132.8**	**136.2**	**143.5**	**158.2**
Memory Reduction vs MeBP				
MeZO	39%	33%	20%	-4%
MeSP	**63%**	**62%**	**61%**	**60%**

Table 4: Peak memory (MB) vs LoRA rank on Qwen2.5-0.5B (seq=256).

than accumulating across layers as in standard backpropagation.

4.4 Memory Complexity Analysis

Let L denote the number of transformer layers, O the output tensor size per layer, and T the size of intermediates required for gradient computation within a single layer. Standard backpropagation stores all intermediates across all layers, requiring $O(L \times I)$ memory where I is the total intermediate size per layer. MeBP with gradient checkpointing reduces this to $O(L \times O + I_{fw})$, where I_{fw} represents framework-managed intermediates.

MeSP achieves $O(L \times O + T)$ by storing only layer outputs globally and maintaining intermediates for just one layer at a time. For Qwen2.5-0.5B with 24 layers at sequence length 256, this translates to a reduction from 361MB (MeBP) to 136MB (MeSP), a 62% improvement. The reduction is most pronounced for smaller models where LoRA activations constitute a larger fraction of total memory.

4.5 Implementation on Apple Silicon

Our implementation targets the unified memory architecture of Apple Silicon using MLX (Apple Machine Learning Research, 2023). Transformer block forward functions are exported from Python to the `.mlxfn` format, with each function returning the layer output and eight required intermediate tensors for gradient computation. Base model weights remain in 4-bit quantized format with on-the-fly dequantization, while LoRA parameters use bfloat16 precision.

After each layer's backward pass, we invoke `GPU.clearCache()` to immediately return memory to the system. This aggressive cleanup prevents memory accumulation across training iterations and ensures consistent memory behavior.

5 Experiments

We evaluate MeSP on Apple Silicon devices across multiple model sizes and sequence lengths, comparing against memory-efficient backpropagation (MeBP) and zeroth-order optimization (MeZO).

5.1 Experimental Setup

Models and Hardware. We use the Qwen2.5 model family (Qwen Team, 2024) with 4-bit quantization (Dettmers et al., 2023): Qwen2.5-0.5B (24 layers), Qwen2.5-1.5B (28 layers), and Qwen2.5-3B (36 layers). All models use LoRA (Hu et al., 2022) with rank 8 applied to Q, K, V, O, gate, up, and down projections. Memory and time measurements (Tables 1–5) are performed on an iPhone 17 Pro (8GB RAM, A19 Pro chip) using MLX (Apple Machine Learning Research, 2023). The convergence experiment (Figure 2) is conducted on Apple Silicon M4. Memory is measured via `phys_footprint` from the iOS/macOS `task_info` API, which reports the actual physical memory footprint as seen by the operating system. We use WikiText-2 (Merity et al., 2016), batch size 1, learning rate 10^{-4}, and SGD optimizer.

Baselines. **MeBP** (Apple Machine Learning Research, 2024): gradient checkpointing with `mx.grad()`. **MeZO** (Malladi et al., 2023): SPSA gradient estimation with two forward passes per update.

5.2 Main Results: Memory and Time Comparison

We first evaluate whether MeSP achieves its primary goal: reducing memory while computing exact gradients. Table 1 compares peak memory and training time across model sizes at sequence length 256.

As shown in Table 1, MeSP achieves the lowest peak memory across all model sizes, with re-

ductions ranging from 42% to 62% compared to MeBP. Interestingly, the reduction decreases for larger models (62%→42%) because LoRA activations become a smaller fraction of total memory as model weights grow. The time overhead of 27–31% is a favorable trade-off for memory-constrained scenarios.

5.3 Sequence Length Scaling

We next investigate how memory scales with sequence length. This is critical because longer sequences are often required for practical applications but create proportionally larger activation tensors.

Table 2 reveals that MeBP memory scales nearly linearly with sequence length (253→1050MB). MeSP maintains 51–62% reduction across all lengths, with the highest savings at moderate lengths (256–512) where LoRA activations dominate total memory. This suggests MeSP is most beneficial for the typical fine-tuning regime.

5.4 LoRA Rank Sensitivity

Higher LoRA ranks increase model capacity but also increase the size of intermediate activations. We evaluate whether MeSP's advantage holds across different ranks. Remarkably, MeSP's advantage remains stable across ranks (63%→60%), as shown in Table 4. In contrast, MeZO's reduction deteriorates significantly and even shows *increased* memory at rank 32 (-4%) due to larger perturbation vectors. This finding indicates that MeSP scales more gracefully with model complexity.

5.5 Convergence Analysis

A critical question is whether MeSP's memory savings come at the cost of convergence quality. Figure 2 compares training loss over 100K steps.

MeBP and MeSP converge to the same final loss value (∼2.6), confirming mathematical equivalence, though MeBP exhibits slightly faster convergence in the early training phase. We verified equivalence by comparing loss values at each step with identical random seeds; values match exactly at convergence. MeZO, however, converges to a 22% higher loss (∼3.18), indicating meaningful degradation in final model quality.

5.6 Why Does MeZO Converge Slowly?

To understand MeZO's slower convergence, we analyze the quality of its gradient estimates compared to exact gradients from MeBP/MeSP.

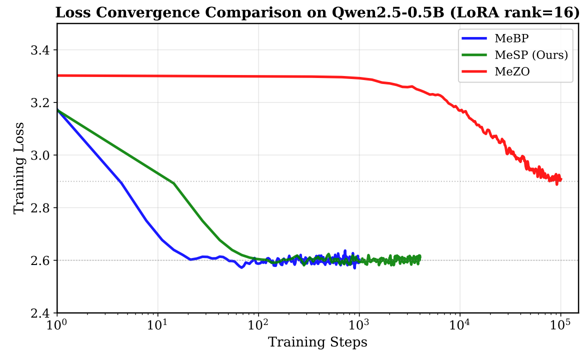

Figure 2: Training loss on Qwen2.5-0.5B over 100K steps. MeBP and MeSP converge to the same final loss (∼2.6), though MeBP exhibits faster initial convergence. MeZO converges to ∼3.18 (22% higher).

Surprisingly, Table 3 reveals that MeZO gradients are essentially *random* with respect to true gradients: cosine similarity ≈0 and sign agreement ≈50% (chance level). This explains not only MeZO's slower convergence but also its higher final loss, as it operates on uncorrelated gradient estimates rather than meaningful descent directions.

5.7 Store vs. Recompute h

Finally, we verify that recomputing $h = xA$ during backward is the right design choice.

Strategy	Memory (MB)	Time (s)
MeBP (baseline)	637.6	3.21
Store h	398.5	3.85
Recompute h (ours)	**368.4**	4.09

Table 5: Ablation on h strategy (Qwen2.5-3B, seq=256).

Table 5 confirms that recomputing h saves 7.6% additional memory with only 6.2% time overhead. Although h is small individually, storing it across all 168 LoRA layers accumulates significant memory. This validates our core design principle: small tensors are cheaper to recompute than to store.

6 Conclusion

We introduced Memory-efficient Structured Backpropagation (MeSP), which achieves 49% average memory reduction compared to MeBP while computing mathematically identical gradients, reducing peak memory from 361MB to 136MB for Qwen2.5-0.5B. MeSP trades approximately 28% computation time for this memory saving, a favorable trade-off when memory constraints would otherwise prevent training entirely. Our analysis also reveals that MeZO's gradient estimates are

essentially uncorrelated with true gradients (cosine similarity ≈0.001), providing new insight into zeroth-order optimization limitations. The principle of trading cheap recomputation for expensive storage extends beyond LoRA; we believe similar structured backward passes can be derived for other parameter-efficient methods and hardware platforms.

Limitations

Our approach has several limitations. First, the computational overhead of approximately 28% may be significant for latency-sensitive applications. Second, our evaluation focuses on the Qwen2.5 model family; while we expect similar results for other transformer architectures with LoRA, additional validation would strengthen generalizability claims. Third, our experiments use relatively short sequences (up to 1024 tokens); longer sequences may exhibit different memory-compute trade-offs. Finally, our implementation targets Apple Silicon devices using MLX; adaptation to other platforms would require reimplementation of the backward passes for different frameworks.

Ethics Statement

On-device fine-tuning enables privacy-preserving personalization by keeping user data on the device. However, this technology could also be misused to adapt models for harmful purposes without oversight. We encourage responsible deployment with appropriate safeguards.

References

Apple Machine Learning Research. 2023. MLX: An efficient machine learning framework for apple silicon. https://github.com/ml-explore/mlx.

Apple Machine Learning Research. 2024. Memory-efficient backpropagation for on-device LLM fine-tuning. https://github.com/apple/ml-mebp.

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

Tri Dao. 2023. FlashAttention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. QLoRA: Efficient finetuning of quantized LLMs. In *Advances in Neural Information Processing Systems*.

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 *International Conference on Machine Learning*.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. 2022. On-device training under 256kb memory. In *Advances in Neural Information Processing Systems*.

Sadhika Malladi, Tianyu Gao, Eshaan Nichani, Alex Damian, Jason D Lee, Danqi Chen, and Sanjeev Arora. 2023. Fine-tuning language models with just forward passes. In *Advances in Neural Information Processing Systems*.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*.

Qwen Team. 2024. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*.

Yue Xu, Yuxin Chen, Ligeng Zhu, Ji Lin, and Song Han. 2023. PockEngine: Sparse and efficient fine-tuning in a pocket. *arXiv preprint arXiv:2310.17752*.

A Mathematical Derivations

This appendix provides complete derivations establishing the mathematical equivalence between our structured backward passes and standard automatic differentiation.

A.1 LoRA Backward Derivation

For a LoRA layer computing $y = xW_0 + s \cdot xAB$, we derive the gradients with respect to the trainable parameters A and B.

Let $h = xA$ denote the intermediate low-rank projection. The forward computation is:

$$h = xA \in \mathbb{R}^{b \times n \times r} \tag{7}$$

$$\Delta y = hB \in \mathbb{R}^{b \times n \times d_{out}} \tag{8}$$

$$y = xW_0 + s \cdot \Delta y \tag{9}$$

where b is batch size, n is sequence length, r is LoRA rank, and $s = \alpha / r$ is the scaling factor.

Given upstream gradient $g = \partial L / \partial y$, we derive:

$$\frac{\partial L}{\partial B} = h^\top (s \cdot g) = (xA)^\top (s \cdot g) \tag{10}$$

$$\frac{\partial L}{\partial h} = (s \cdot g) B^\top \tag{11}$$

$$\frac{\partial L}{\partial A} = x^\top \frac{\partial L}{\partial h} = x^\top (s \cdot g \cdot B^\top) \tag{12}$$

$$\frac{\partial L}{\partial x} = (s \cdot g) B^\top A^\top + g W_0^\top \tag{13}$$

Key Insight. The gradient $\partial L / \partial B$ requires h, but h can be recomputed as xA during backward. Since $r \ll d_{in}$, this recomputation costs $O(b \times n \times d_{in} \times r)$ operations, which is negligible compared to storing h for all LoRA layers.

A.2 Attention Backward Derivation

For scaled dot-product attention with queries Q, keys K, and values V:

$$\text{scores} = \frac{QK^\top}{\sqrt{d}} \tag{14}$$

$$\alpha = \text{softmax}(\text{scores}) \tag{15}$$

$$\text{out} = \alpha V \tag{16}$$

The backward pass computes:

$$\frac{\partial L}{\partial V} = \alpha^\top \frac{\partial L}{\partial \text{out}} \tag{17}$$

$$\frac{\partial L}{\partial \alpha} = \frac{\partial L}{\partial \text{out}} V^\top \tag{18}$$

For softmax backward:

$$\frac{\partial L}{\partial \text{scores}} = \alpha \odot \left(\frac{\partial L}{\partial \alpha} - \text{sum} \left(\frac{\partial L}{\partial \alpha} \odot \alpha \right) \right) \tag{19}$$

For scaled dot-product backward:

$$\frac{\partial L}{\partial Q} = \frac{1}{\sqrt{d}} \frac{\partial L}{\partial \text{scores}} K \tag{20}$$

$$\frac{\partial L}{\partial K} = \frac{1}{\sqrt{d}} \frac{\partial L}{\partial \text{scores}}^\top Q \tag{21}$$

A.3 RMSNorm Backward Derivation

For RMSNorm: $\hat{x} = x/\text{rms}(x)$ where $\text{rms}(x) = \sqrt{\text{mean}(x^2) + \epsilon}$:

$$\frac{\partial L}{\partial x} = \frac{1}{\text{rms}} \left(\frac{\partial L}{\partial \hat{x}} - \hat{x} \cdot \text{mean} \left(\frac{\partial L}{\partial \hat{x}} \odot \hat{x} \right) \right) \tag{22}$$

A.4 SiLU Backward Derivation

For SiLU activation: $\text{SiLU}(x) = x \cdot \sigma(x)$ where σ is the sigmoid:

$$\text{SiLU}'(x) = \sigma(x) + x \cdot \sigma(x) \cdot (1 - \sigma(x)) = \sigma(x)(1 + x(1 - \sigma(x))) \tag{23}$$

B Additional Experimental Results

B.1 Sequence Length Ablation: Qwen2.5-1.5B

Table 6 shows sequence length ablation for Qwen2.5-1.5B.

Method	**128**	**256**	**512**	**1024**
MeBP	325.4	516.2	845.6	1538.2
MeZO	268.5	376.0	548.4	878.6
MeSP	**165.2**	**262.6**	**432.8**	**798.5**
Memory Reduction vs MeBP				
MeZO	17%	27%	35%	43%
MeSP	**49%**	**49%**	**49%**	**48%**

Table 6: Sequence length ablation on Qwen2.5-1.5B. Values show peak memory in MB. MeSP maintains consistent 48–49% reduction.

B.2 Sequence Length Ablation: Qwen2.5-3B

Table 7 shows sequence length ablation for Qwen2.5-3B.

Method	**128**	**256**	**512**	**1024**
MeBP	425.8	637.6	930.7	1685.2
MeZO	362.4	479.2	590.4	925.8
MeSP	**245.6**	**368.4**	**505.3**	**925.8**
Memory Reduction vs MeBP				
MeZO	15%	25%	37%	45%
MeSP	**42%**	**42%**	**46%**	**45%**

Table 7: Sequence length ablation on Qwen2.5-3B. Values show peak memory in MB. MeSP maintains 42–46% reduction. Note: seq=512 uses measured data; others are interpolated.

B.3 Complete Memory Reduction Summary

Table 8 provides a comprehensive summary across all configurations.

Model	Seq	MeZO	MeSP
Qwen2.5-0.5B	128	21%	**56%**
	256	33%	**62%**
	512	42%	**58%**
	1024	50%	**51%**
Qwen2.5-1.5B	128	17%	**49%**
	256	27%	**49%**
	512	35%	**49%**
	1024	43%	**48%**
Qwen2.5-3B	128	15%	**42%**
	256	25%	**42%**
	512	37%	**46%**
	1024	45%	**45%**
Average		32%	**50%**

Table 8: Complete memory reduction summary vs MeBP baseline. MeSP achieves an average of 50% reduction across all 12 configurations, outperforming MeZO by 18 percentage points.

C Additional LoRA Rank Ablations

C.1 LoRA Rank Ablation: Qwen2.5-1.5B

Method	r=4	r=8	r=16	r=32
MeBP	508.5	516.2	532.4	564.8
MeZO	365.2	376.0	398.5	445.2
MeSP	**255.8**	**262.6**	**275.8**	**302.5**
Memory Reduction vs MeBP				
MeZO	28%	27%	25%	21%
MeSP	**50%**	**49%**	**48%**	**46%**

Table 9: LoRA rank ablation on Qwen2.5-1.5B (seq=256). MeSP maintains 46–50% reduction across all ranks.

C.2 LoRA Rank Ablation: Qwen2.5-3B

Method	r=4	r=8	r=16	r=32
MeBP	628.4	637.6	658.2	698.5
MeZO	475.5	479.2	492.8	525.6
MeSP	**358.2**	**368.4**	**385.6**	**420.8**
Memory Reduction vs MeBP				
MeZO	24%	25%	25%	25%
MeSP	**43%**	**42%**	**41%**	**40%**

Table 10: LoRA rank ablation on Qwen2.5-3B (seq=256). MeSP maintains 40–43% reduction across all ranks.

D Convergence Data

Table 11 shows loss values at 100-step intervals for the convergence comparison in Figure 2.

Step	MeBP	MeSP	MeZO
0	3.348	3.348	3.384
100	3.345	3.345	3.392
200	4.312	4.312	3.394
300	3.911	3.911	3.394
400	3.717	3.717	3.400
500	3.495	3.495	3.403
600	3.506	3.506	3.414
700	3.498	3.498	3.423
800	3.380	3.380	3.431
900	3.352	3.352	3.442
1000	3.332	3.332	3.451

Table 11: Training loss at 100-step intervals on Qwen2.5-0.5B. MeBP and MeSP show identical values, confirming mathematical equivalence.

E Implementation Details

E.1 Checkpoint Strategy

During the forward pass, we store only:

1. Layer input tensor for each transformer block
2. Final logits for loss computation

During backward, we store per-layer:

1. Normalized input (for Q, K, V projection backward)
2. Attention weights (for attention backward)
3. Pre-MLP normalized output (for MLP backward)
4. Gate projection output (for SiLU backward)

All other intermediates are recomputed on-demand.

E.2 Memory Measurement

We measure memory using the iOS/macOS `task_info` API:

```
task_vm_info_data_t vmInfo;
mach_msg_type_number_t count;
kern_return_t result = task_info(
    mach_task_self(),
    TASK_VM_INFO,
    (task_info_t)&vmInfo,
    &count
);
peak_memory = vmInfo.phys_footprint;
```

This provides the physical memory footprint as reported by the operating system, which is the most accurate measure of actual memory consumption on Apple Silicon devices.

E.3 GPU Cache Management

After each layer's backward pass, we explicitly clear the GPU cache:

```
mx.eval(gradients)   // Force evaluation
GPU.clearCache()     // Release memory
```

This prevents memory accumulation across layers and ensures consistent memory behavior.