MobiZO: Enabling Efficient LLM Fine-Tuning at the Edge via Inference Engines

Anonymous ACL submission

Abstract

Large Language Models (LLMs) are currently pre-trained and fine-tuned on large cloud servers. The next frontier is LLM personalization, where a foundation model can be fine-tuned with user/task-specific data. Given the sensitive nature of such private data, it is desirable to fine-tune these models on edge devices to improve user trust. However, fine-tuning on resource-constrained edge devices presents significant challenges due to substantial memory and computational demands, as well as limited infrastructure support. We observe that inference engines (e.g., ExecuTorch) can be repurposed for fine-tuning by leveraging zeroth-order (ZO) optimization, which uses multiple forward passes to approximate gradients. While promising, direct application of ZO methods on edge devices is inefficient due to the high computational cost of multiple forward passes required for accurate gradient estimation, and their deployment has been largely unexplored in practice. We introduce MobiZO, a resource-efficient fine-tuning framework for LLMs specifically designed for edge devices. MobiZO combines three key innovations: (1) a parallelized randomized gradient estimator that employs both outer-loop and inner-loop parallelism to eliminate sequential forward passes, (2) a specialized Multi-Perturbed LoRA (MP-LoRA) module that enables efficient realization of both inner and outer loop parallelism, and (3) a seamless integration with ExecuTorch for on-device training, requiring no modifications to the runtime. Experiments demonstrate that MobiZO achieves substantial runtime speedups and memory savings while improving fine-tuning accuracy, paving the way for practical deployment of LLMs in real-time, on-device applications. Code available at: anonymous.4open.science/r/MobiZO-DBC6.

1 Introduction

Large Language Models (LLMs) have demonstrated strong performance across varied tasks, chatbots, image generation (OpenAI et al., 2024; Chowdhery et al., 2022; Gemini-Team et al., 2024). Fine-tuning is a crucial step for adapting LLMs to specific tasks, but it demands significant memory resources for storing model parameters, gradients, activations, and optimizer states (Wan et al., 2024). This memory overhead makes fine-tuning infeasible on resource-constrained devices such as smartphones and edge platforms (Yin et al., 2024). Moreover, existing on-device frameworks like ExecuTorch (Meta-AI, 2024a) and TensorFlow Lite (Google, 2020) primarily optimize inference, leaving fine-tuning largely unsupported.

Resource Challenges Despite Recent Advances. Techniques such as parameter-efficient fine-tuning (PEFT) (Hu et al., 2022; Houlsby et al., 2019; Li and Liang, 2021; Lester et al., 2021) and memory-efficient fine-tuning (Dettmers et al., 2023; Lv et al., 2024; Zhao et al., 2024; Malladi et al., 2023) can significantly reduce the memory footprint associated with model weights, gradients, and optimizer states. However, even with these methods, storing internal activations during backpropagation remains a significant challenge. For example, fine-tuning Llama 7B requires up to 45.6 GB of on-chip memory for internal activations (Lv et al., 2024), making it impractical for most edge devices. Current solutions still fall short of meeting the stringent resource constraints of edge environments.

Limitations of On-Device Training Frameworks. While several techniques exist to mitigate the memory costs of intermediate activations during backpropagation, they generally rely on training frameworks that support automatic differentiation to perform backpropagation. For instance, gradient checkpointing (Chen et al., 2016) discards select activations during the forward pass and recomputes them during backpropagation, while gradient accumulation aggregates gradients over smaller batches. PockEngine (Zhu et al., 2023) limits backpropagation to update a subset of layers, reducing

the need to store activations for other layers. However, all these techniques are not well supported by the existing on-device training frameworks on most edge platforms such as Android devices.

Zeroth-Order Optimization as a Potential Solution. Zeroth-order (ZO) optimization has gained attention as a way to eliminate the need to store activations by estimating gradients using only forward passes. Specifically, ZO methods approximate gradients by evaluating the loss function at multiple perturbed versions of the model weights and using these values for gradient estimation. This approach has the potential to solve the memory challenge, as well as avoid the need for backpropagation support. Thus, ZO methods hold promise for on-device fine-tuning by utilizing existing inference frameworks like ExecuTorch (Meta-AI, 2024a). However, applying ZO optimization to fine-tune LLMs on edge devices presents its own set of challenges.

One classic zeroth-order optimizer, the Randomized Gradient Estimator (RGE) (Duchi et al., 2015; Nesterov and Spokoiny, 2017), estimates gradients by computing finite differences of function values along randomly chosen perturbation vectors. With RGE, estimation accuracy for each step of training improves as the number of stochastic perturbations (also referred to as queries) increases (Zhang et al., 2024b; Gautam et al., 2024; Yang et al., 2024), but the computational cost scales linearly with the query count. In addition, its on-device adaptation remains largely unexplored.

In this work, we propose **MobiZO** training framework to address the runtime overhead inherent in multi-query RGE. MobiZO includes a novel **Multi-Perturbed LoRA** (MP-LoRA) design and combines **outer-loop parallelization** and **inner-loop parallelization** to perform multiple forward passes in parallel, substantially reducing per-step latency while harvesting the accuracy benefits of multi-query gradient estimation. Moreover, MobiZO allows seamless adaptation for deploying RGE optimization via inference engines to enable practical on-device fine-tuning. Our contributions are as follows:

- We introduce the MobiZO framework, specialized for on-device training, consisting of outer-loop, inner-loop parallelization, and Multi-Perturbed LoRA designs. By executing multiple forward passes in parallel within each MP-LoRA module, MobiZO effectively amortizes the memory access cost of loading model parameters, thereby reducing training time while improving model performance.
- We demonstrate that the MobiZO framework can be seamlessly integrated into inference engines such as ExecuTorch without requiring any modifications to its runtime code. Our approach is realized through minimal server-side code changes only, making it practical for on-device fine-tuning.
- We empirically validate that our method achieves substantial wall-clock time speedups and memory savings while improving model performance. Our approach results in up to **4.3**$\times$ end-to-end training speedups and up to **8.1**$\%$ improvement in accuracy compared to the MeZO baseline.

2 Background and Related Work

Low-Rank Adaptation. To reduce the resource demands of LLM fine-tuning, parameter-efficient fine-tuning methods update only a small subset of parameters. LoRA (Hu et al., 2022) introduces trainable low-rank matrices $\mathbf{A} \in \mathbb{R}^{k_{in} \times r}$ and $\mathbf{B} \in \mathbb{R}^{r \times k_{out}}$ while freezing the original weight matrix $\mathbf{W}$. Since $r \ll \min(k_{in}, k_{out})$, the number of trainable parameters is significantly reduced. The forward pass is computed as $\mathbf{y} = \mathbf{x}\mathbf{W} + \mathbf{x}\mathbf{A}\mathbf{B}$, where $\mathbf{A}$ is initialized randomly and $\mathbf{B}$ starts at zero, ensuring no initial deviation from the pre-trained model. Variations such as LoRA-FA (Zhang et al., 2023) further reduce trainable parameters by freezing $\mathbf{A}$ and updating only $\mathbf{B}$.

Zeroth-Order Optimization. ZO optimization methods have been widely applied across various machine learning applications (Chen et al., 2017; Sun et al., 2022; Wang et al., 2022; Liu et al., 2024c). Among ZO gradient estimators, the randomized gradient estimator (RGE) is particularly effective, especially for fine-tuning LLMs (Malladi et al., 2023). Given a labeled dataset $\mathcal{D}$ and a model with parameters $\boldsymbol{\theta} \in \mathbb{R}^d$, let the loss function on a minibatch $\mathcal{B} \subset \mathcal{D}$ of size B be denoted as $\mathcal{L}(\boldsymbol{\theta}; \mathcal{B})$. The RGE estimates the gradient of the loss $\mathcal{L}$ with respect to the parameters $\boldsymbol{\theta}$ on a minibatch $\mathcal{B}$ via:

$$\hat{\nabla}\mathcal{L}(\boldsymbol{\theta}; \mathcal{B}) = \frac{1}{q}\sum_{i=1}^{q}\left[\frac{\mathcal{L}(\boldsymbol{\theta} + \epsilon \boldsymbol{z}_i; \mathcal{B}) - \mathcal{L}(\boldsymbol{\theta} - \epsilon \boldsymbol{z}_i; \mathcal{B})}{2\epsilon}\boldsymbol{z}_i\right], \tag{179}$$

where $\boldsymbol{z}_i \sim \mathcal{N}(0, \boldsymbol{I}d)$, q is the query number, and ϵ is the perturbation scale. The choice of q balances the variance of the ZO gradient estimate and the

computational cost, and the variance of the RGE is approximately $O(d/q)$ (Zhang et al., 2024b).

ZO-SGD replaces FO gradients with ZO gradient estimates: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta\hat{\nabla}\mathcal{L}(\boldsymbol{\theta}; \boldsymbol{\mathcal{B}}_t)$, with learning rate η at timestep t. The choice of optimizer (SGD) is orthogonal to ZO optimization methods, but in our preliminary experiments, we find adaptive optimizers such as Adam would not necessarily improve LLM fine-tuning performance.

ZO LLM Fine-Tuning. Conventional RGE training requires storing perturbation noise $\boldsymbol{z}$, effectively doubling inference memory. MeZO (Malladi et al., 2023) eliminates this overhead by storing only the random seed and regenerating $\boldsymbol{z}$ on demand. While MeZO also considers $q > 1$, it compensates for the increased computation per step by proportionally reducing the total number of training steps (e.g., halving the steps when $q = 2$). Under this fixed computational budget, they observe that larger q does not improve accuracy compared to $q = 1$, prompting MeZO to adopt $q = 1$ as the default setting. In contrast, Zhang et al. benchmarked various ZO optimization methods, including RGE with $q > 1$, and confirmed that when computational constraints are lifted, larger q can indeed enhance performance.

Sparse-MeZO (Liu et al., 2024b) selectively updates parameters but is sensitive to hyperparameters. Extreme-sparse-MeZO (Guo et al., 2025) integrates first-order Fisher-based sparse training. MeZO-SVRG (Gautam et al., 2024) improves variance reduction but occasionally requires full-dataset gradient estimation, increasing cost. AdaZeta (Yang et al., 2024) adaptively schedules queries but still relies on sequential gradient estimations.

On-device LLM Training. Several methods address the memory and compute constraints of on-device LLM training. PockEngine (Zhu et al., 2023) updates only select layers, skipping gradient calculations for less critical parameters. FwdLLM (Xu et al., 2024) applies numerical differentiation to approximate gradients, lowering communication costs in federated learning but is limited to CUDA environment. HETLORA (Cho et al., 2024) enables federated LoRA training across heterogeneous devices but requires further real-world testing due to high activation memory costs. PocketLLM (Peng et al., 2024) evaluates MeZO for on-device fine-tuning but does so in a simulated Linux environment rather than mobile devices.

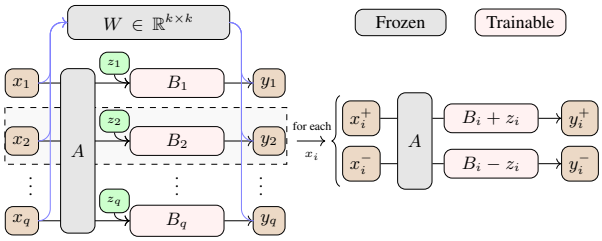
Algorithm 1 MobiZO Algorithm.

Input: learnable parameters $\boldsymbol{\theta}_l \in \mathbb{R}^{d_l}$, frozen parameters $\boldsymbol{\theta}_f \in \mathbb{R}^{d_f}$, loss $\mathcal{L} : \mathbb{R}^{d_l} \times \mathbb{R}^{d_f} \to \mathbb{R}$, step budget T, query budget q, effective batch size E, perturbation scale ϵ, learning rate η
for $t = 1$ to T **do**
  Sample batch $\boldsymbol{\mathcal{B}} \subset \mathcal{D}$
  for $i = 1$ to q **do in parallel** ▷ Outer
    Sample random seed s_i
    $\boldsymbol{z}_i \sim \mathcal{N}(0, \boldsymbol{I}_{d_l})$ using s_i
    for $k \in \{+1, -1\}$ **do in parallel** ▷ Inner
      $\boldsymbol{\theta}_l^{(k)} = \boldsymbol{\theta}_l + k\epsilon\boldsymbol{z}_i$ ▷ MP-LoRA
      $\ell^{(k)} = \mathcal{L}((\boldsymbol{\theta}_l^{(k)}, \boldsymbol{\theta}_f); \boldsymbol{\mathcal{B}})$
    end for
    $g_i = \dfrac{\ell^{(+1)} - \ell^{(-1)}}{2\epsilon}$
    Store s_i and g_i
  end for
  $\boldsymbol{\theta}_l \leftarrow \boldsymbol{\theta}_l - \eta \left(\dfrac{1}{q} \sum_{i=1}^{q} g_i \boldsymbol{z}_i \right)$
end for

3 The MobiZO Framework

To perform a q-query gradient estimation, Randomized Gradient Estimation (RGE) typically requires $2q$ forward passes. The naive implementation of RGE (detailed in Appendix A) executes these passes sequentially. However, these forward passes are inherently independent—the only difference being the random perturbations applied to the trainable parameters. To improve runtime under resource constraints while preserving the accuracy benefits of multi-query RGE, we propose the MobiZO framework. MobiZO enables parallel execution of gradient estimation through a specialized PEFT design called Multi-Perturbed LoRA (MP-LoRA). In addition, we show that MobiZO can be seamlessly integrated into inference engines such as ExecuTorch to enable practical and efficient on-device fine-tuning without modifying the runtime.

MP-LoRA. A straightforward approach to performing multiple forward passes in parallel is to duplicate the model inputs and model weights, perturb each weight copy with distinct perturbations, and then execute the forward passes concurrently. However, this naive duplication incurs substantial memory overhead from replicating weights and managing perturbations, in terms of both storage and I/O access. The random seed trick introduced by MeZO (Malladi et al., 2023) mitigates the mem-



(a) Outer-loop (b) Inner-loop

Figure 1: Overview of the MP-LoRA module in the MobiZO framework that supports both outer-loop and inner-loop parallelization, enabling faster training and improved model accuracy with minimal memory usage and access overhead.

ory overhead from storing the full perturbation vectors from $O(d)$ to $O(1)$. However, the computation time for the parameter perturbation step becomes $O(d)$ as trainable parameters must be sequentially updated using perturbations regenerated from the seed. This sequential process can substantially slow down training for large models, potentially negating the speedups gained from eliminating backpropagation.

To address both the memory overhead of parameter duplication and the $O(d)$ sequential parameter operations inherent in the seed trick, MP-LoRA leverages PEFT methods, which drastically reduce the number of trainable parameters. Our preliminary experiments (see Appendix B) indicate that combining ZO with LoRA-FA yields superior performance compared to alternatives like DoRA (Liu et al., 2024a) and VeRA (Kopiczko et al., 2024). Consequently, LoRA-FA serves as the foundational PEFT method for our MP-LoRA design. We note that, while MP-LoRA is developed upon a LoRA-based PEFT method, its core principles can be adapted to other PEFT techniques.

The MP-LoRA framework is designed to efficiently manage multiple perturbed states of LoRA parameters for concurrent forward passes. Similar to standard LoRA, MP-LoRA modules are applied to specific weights of the pre-trained model, augmenting them with a small number of trainable parameters. The MP-LoRA augmented model takes an input batch $\mathbf{x}$ and a set of n distinct perturbations (which can be the full perturbation vectors or their generating seeds). At the beginning of a model execution, n copies of the input batch are created, $\mathbf{X}^{(n)} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n]$. Each copy $\mathbf{x}_i$ is then associated with the i-th perturbation path for its entire traversal through the model's layers. This collection, $\mathbf{X}^{(n)}$, serves as the effective input that propagates through the network, encountering MP-LoRA layers. Within any given MP-LoRA layer, the original pre-trained model weights $\mathbf{W}$ and the LoRA $\mathbf{A}$ matrix are kept fixed and are shared across all n operational paths passing through that layer. Only the LoRA $\mathbf{B}$ matrix is effectively replicated and distinctly perturbed n times for these paths, resulting in a set of path-specific matrices $\mathbf{B}^{(n)} = [\mathbf{B}_1, \mathbf{B}_2, \ldots, \mathbf{B}_n]$, where each $\mathbf{B}_i$ corresponds to the perturbation path i.

The output of an MP-LoRA layer for these n concurrent paths is computed as:

$$\mathbf{Y}^{(n)} = \mathbf{X}^{(n)}\mathbf{W} + (\mathbf{X}^{(n)}\mathbf{A}) \odot \mathbf{B}^{(n)}$$

Here, $\odot$ denotes a batched operation where each component $(\mathbf{x}_i\mathbf{A})$ from the n paths undergoes matrix multiplication with its respective perturbed LoRA matrix $\mathbf{B}_i$. Since $\mathbf{B}_i$ is orders of magnitude smaller than $\mathbf{W}$, the overall memory overhead is negligible even with the replication demands. MP-LoRA achieves its efficiency by ensuring these n distinct path-specific computations per layer are performed concurrently, while reusing the shared $\mathbf{W}$ and $\mathbf{A}$ components.

Outer-Loop Parallelization. Each RGE step consists of evaluating q distinct stochastic perturbations (queries). We employ MP-LoRA to execute these q queries in parallel, as illustrated in Figure 1(a). This is achieved by invoking the MP-LoRA mechanism once with $n = q$ independent perturbations (or their random seeds) as input. This generates q distinct perturbed model states that are subsequently processed concurrently through the model. However, performing q operations simultaneously would naively increase the computational load per step by a factor of q.

To maintain a computational cost per training step comparable to that of single-query RGE, when performing MobiZO with $q > 1$ queries, we proportionally reduce the input batch size for each of the q paths to $E = B/q$. Here, B is the original batch size used in a $q = 1$ setting, and E is termed the *effective batch size* per query path. For instance, if a baseline setting uses $q = 1$ and $B = 16$, our MobiZO approach can use $q = 4$ with an effective batch size $E = 4$ per query, thereby maintaining the total number of samples processed per step ($qE = B$). As we demonstrate in Section 4, this trade-off of increasing q while reducing E often leads to improved model accuracy under a fixed computational budget. The motivation and theoretical background behind this trade-off are detailed

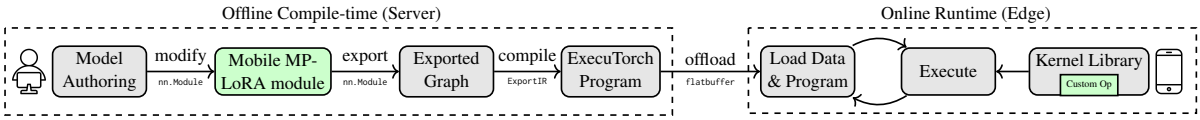


Figure 2: MobiZO on-device training workflow via ExecuTorch with minimal modifications. The green box represents additional procedure in addition to standard steps for inference deployment on edge devices.

in Appendix F.3. Reducing the effective batch size E in a multi-query setting also offers the ancillary benefit of reducing padding tokens, as shown in Appendix C. Typically, larger batch sizes lead to more padding, as sequences of varying lengths are padded to match the maximum sequence length within that larger batch. By adopting a smaller effective batch size per query, the total amount of padding can decrease, limiting wasted computation on these padding tokens, especially during attention operations.

Another key benefit of this MP-LoRA-enabled outer-loop parallelization is the improved data locality for model weights. By loading the shared weights (such as the base model weights $\mathbf{W}$ and LoRA $\mathbf{A}$ matrices within MP-LoRA layers) once and reusing them across all q concurrent queries, costly external memory accesses are amortized. This can lead to a runtime per training step that is comparable to, or even faster than, the original sequential setting with $q = 1$, despite processing multiple queries.

Inner-Loop Parallelization. While outer-loop parallelization addresses concurrency across multiple queries, each gradient estimation in RGE still requires two forward passes per query: one with a positive perturbation and one with a negative perturbation. In standard RGE, these are typically executed sequentially.

To further accelerate each gradient estimation, MobiZO incorporates inner-loop parallelization, as outlined in Algorithm 1 (line 7). This is also enabled by MP-LoRA, which for a given query i, can perform both the positive and negative perturbation forward passes simultaneously, as illustrated in Figure 1(b). This is achieved by invoking MP-LoRA with $n = 2q$, using perturbations $+\epsilon\mathbf{z}_i$ and $-\epsilon\mathbf{z}_i$ applied to the LoRA $\mathbf{B}$ matrix for the i-th path. By processing positively and negatively perturbed states in a single MP-LoRA invocation, we obtain two corresponding outputs. The loss difference between these outputs can then be used to estimate the gradient component for that query, effectively performing the gradient estimation using a single forward pass. This approach further reduces the external memory bandwidth burden by maximizing the reuse of shared model weights across these two evaluations. Consequently, MobiZO with inner-loop parallelization can achieve an even faster runtime per training step compared to the sequential execution of two forward passes in RGE, even for $q = 1$.

With inner-loop parallelization, the activation size at each layer is doubled, as it forwards two batches at the same time. However, this does not result in significant memory overhead. Unlike first-order methods, ZO methods allow activations from previous layers to be discarded during forward passes, rather than accumulating across all layers. This property, as noted in (Zhang et al., 2024b), enables ZO methods to scale more efficiently with long sequence lengths and large batch sizes compared to FO methods. To minimize memory costs for storing LoRA-$\mathbf{B}$ weight matrices, it is possible to keep a master copy of LoRA-$\mathbf{B}$ and instantiate perturbed copies dynamically during the forward pass. At each LoRA layer, only the master copy is updated with the gradient and learning rate. Perturbed copies of LoRA-$\mathbf{B}$ are then instantiated and deleted once the output is computed, ensuring that the number of additional trainable parameters remains the same as in the conventional ZO method.

A crucial benefit of MP-LoRA enabling inner-loop parallelization in this manner is that it forms the basis for on-device fine-tuning using inference engines, as detailed next.

On-Device Training Adaptation. Deploying a model with inference engine ExecuTorch involves two primary steps: (1) converting a standard PyTorch `nn.Module` into an ExecuTorch program, a serialized computation graph with embedded parameters; and (2) offloading this binary file along with the C++ runtime to the edge device. The edge runtime then interprets and executes the model using a backend-specific operator library.

However, ExecuTorch does not natively support MeZO due to its reliance on complex device-side operations, such as random number generation, weight perturbation, and gradient application, none of which are exposed through standard ExecuTorch

APIs. For instance, line 8 in Algorithm 1 requires direct modification of model weights, an operation currently unsupported. To address this limitation, we design a mobile version of the MP-LoRA that encapsulates all MobiZO logic inside its `forward` function. This approach enables full exportability and execution within the standard ExecuTorch runtime, eliminating the need to modify low-level components.

Algorithm 2 shows the **Mobile MP-LoRA** module for $q = 1$. It maintains two perturbed variants of the LoRA weight matrix $\mathbf{B}$, each scaled by ϵ with positive and negative noise. Since ExecuTorch does not support resetting random seeds between forward passes, we store both perturbed versions of $\mathbf{B}$ in memory instead of regenerating them. During execution, the module computes the gradient using the difference between $\mathbf{B}[0]$ and $\mathbf{B}[1]$, restores the unperturbed weight, and updates the parameter via projected gradient. The final output is computed using the sum of the frozen linear term $\mathbf{xW}$ and the adaptive term $\mathbf{xAB}$.

Figure 2 illustrates the complete workflow. We begin with a pre-trained PyTorch model and replace its linear layers with our Mobile MP-LoRA modules. The modified model is then exported using the standard ExecuTorch pipeline and deployed to the edge device. On-device execution is handled entirely by the ExecuTorch runtime, which runs the fine-tuning process implicitly through the forward pass. Additionally, a lightweight noise generation operator is registered using the ExecuTorch extension API (Meta-AI, 2024b) to support model perturbation.

Algorithm 2 Mobile MP-LoRA Module

1: **Input:** $\mathbf{x} \in \mathbb{R}^{2\times \text{seq\_len}\times \text{k}}$, $\mathbf{A} \in \mathbb{R}^{k\times r}$, $\mathbf{B} \in \mathbb{R}^{2\times r\times k}$, $\mathbf{W}^{k\times k}$, learning rate η, perturbation scale ϵ, projected gradient g
2: $\text{diff} = \frac{\mathbf{B}[0]-\mathbf{B}[1]}{2}$
3: $\text{update} = \eta \cdot g \cdot \frac{\text{diff}}{\epsilon}$
4: $\mathbf{z} = \epsilon \cdot randn\_like(\mathbf{B}[0])$
5: $\mathbf{B}[0] = \mathbf{B}[0] - \text{diff} - \text{update} + \mathbf{z}$
6: $\mathbf{B}[1] = \mathbf{B}[1] + \text{diff} - \text{update} - \mathbf{z}$
7: $\text{output} = \mathbf{xW} + bmm(\mathbf{xA}, \mathbf{B})$
8: **Return:** output

4 Experiments

We conduct comprehensive experiments on the TinyLlama-1.1B (Zhang et al., 2024a) and Llama2-7B (Touvron et al., 2023) models across different systems to evaluate both fine-tuning performance and system efficiency.

4.1 Model Fine-Tuning Performance

We compare two sets of baselines: the first employs an FO-SGD optimizer in both the full and LoRA-FA parameter spaces, while the second uses a ZO-SGD optimizer with MeZO ($q = 1, B = 16$) in the same parameter spaces. For our method, MobiZO, we ensure equivalent computation per training step while varying q by scaling the effective batch size (E) to maintain a fixed $E * q$ value, such that setting $q = 4, E = 4$ or $q = 16, E = 1$ has equal computation load. By using the same number of training steps (i.e., 20,000) for both MobiZO and MeZO, we ensure that MobiZO does not exceed the computational budget of the MeZO baseline for end-to-end training. Experiments are conducted using three random seeds, and we report the average performance. For reference, we also report zero-shot performance without additional fine-tuning.

For the smaller-scale TinyLlama-1.1B model, we evaluate its performance on the GLUE dataset (Wang et al., 2019). The results in Table 1 show that increasing the number of queries while decreasing the batch size outperforms the baseline MeZO by up to 7.5% accuracy. For the larger Llama2-7B model, we evaluate its performance on SST-2 (Wang et al., 2019), SuperGLUE (Wang et al., 2020), WinoGrande (Sakaguchi et al., 2021), ARC-Easy, and ARC-Challenge (Clark et al., 2018) datasets using the same experimental setup. Additional experimental details, including dataset descriptions, training procedures, and hyperparameters, are provided in Appendix D.

From the results in Table 1, we observe that MobiZO consistently outperforms MeZO across nearly all tasks on Llama2-7B model. Notably, MobiZO improves performance over the baseline that updates the full parameter space by up to 8.1% in the WiC task, demonstrating its effectiveness in leveraging multi-query gradient estimation for improved fine-tuning quality under the same computational budget. Although MobiZO introduces an additional hyperparameter q, we find that setting q to 4 or 16 generally yields strong performance, reducing the need for extensive tuning. Additional results exploring a broader range of q values are provided in Appendix F, along with an in-depth discussion of the accuracy gains enabled by MobiZO's

TinyLlama-1.1B	Methods \ Tasks	SST-2	RTE	MRPC	QQP	QNLI	WNLI
Zero-shot		55.5	51.6	68.4	32.8	52.7	43.7
FO-SGD	Full	93.0	80.6	80.0	84.0	83.6	58.2
	LoRA-FA	93.0	77.7	79.1	83.3	84.3	54.9
ZO-SGD	MeZO (Full)	**91.1**	65.3	70.8	73.7	69.2	58.2
	MeZO (LoRA-FA)	86.5	67.1	72.1	74.7	62.4	59.2
	MobiZO ($q = 4$)	88.5	70.5	**73.6**	76.4	75.0	**60.6**
	MobiZO ($q = 16$)	89.7	**72.4**	73.4	**77.0**	**76.7**	59.6

Llama2-7B	Methods \ Tasks	SST-2	RTE	BoolQ	WSC	WiC	MultiRC	COPA	WinoGrande	ARC-E	ARC-C
Zero-shot		58.0	59.2	71.9	52.9	50.0	54.9	79.0	62.7	47.9	35.0
FO-SGD	Full	95.8	86.5	85.3	66.0	72.5	82.4	87.0	67.0	80.6	66.2
	LoRA-FA	95.4	81.7	84.9	62.8	62.6	74.7	84.0	64.2	81.3	60.7
ZO-SGD	MeZO (Full)	92.2	73.5	81.9	64.7	55.6	68.6	84.0	64.5	69.8	47.5
	MeZO (LoRA-FA)	92.4	70.8	81.6	63.5	63.4	72.3	86.3	64.2	73.6	50.6
	MobiZO ($q = 4$)	94.1	**75.5**	**82.7**	**64.1**	63.1	**74.7**	85.0	64.8	**73.8**	50.9
	MobiZO ($q = 16$)	**94.2**	**75.5**	**82.7**	62.5	**63.7**	72.9	**87.3**	**64.9**	73.2	**51.4**

Table 1: Performance of fine-tuning TinyLlama-1.1B and Llama2-7B on different tasks with different optimizers. MobiZO outperforms the baseline MeZO in most tasks under the same computational budget.

Sequence length	64			128			256		
Batch size	1	8	16	1	8	16	1	8	16
TinyLlama-1.1B									
FO (Full)	11.32	12.00	12.78	11.44	13.00	14.76	11.77	15.62	20.02
FO (LoRA-FA)	4.15	5.00	5.98	4.27	5.98	7.81	4.51	7.81	11.58
MeZO (LoRA-FA)	2.09	2.19	2.32	2.10	2.32	2.56	2.13	2.56	3.05
MobiZO	2.11	2.32	2.56	2.14	2.56	3.05	2.20	3.05	3.98
Llama2-7B									
FO (Full)	64.31	66.12	69.20	64.6	68.51	72.97	65.32	74.22	84.40
FO (LoRA-FA)	25.16	27.20	29.58	25.46	29.58	34.28	26.05	34.29	43.66
MeZO (LoRA-FA)	12.59	12.70	12.82	12.61	12.82	13.06	12.64	13.06	13.55
MobiZO	12.61	12.82	13.06	12.64	13.07	13.55	12.70	13.55	14.53

Table 2: Peak memory usage (GB) of TinyLlama-1.1B and Llama2-7B for different sequence length and batch size configurations.

design choices.

4.2 System Performance

We conduct measurements on a single NVIDIA A100 GPU to evaluate the server-side system performance of MobiZO compared to its baselines. The ZO-SGD optimizer, including both MeZO and MobiZO, performs forward passes in 16-bit floating-point precision to maximize computational efficiency, leveraging ZO's tolerance for low-precision gradient estimation (Zhang et al., 2024b). We use the FO-SGD optimizer with mixed-precision training enabled for memory and runtime evaluations.

Memory Efficiency. We first evaluate the peak memory usage of MobiZO across different fixed sequence length and batch size configurations. The reported memory footprint includes storage for weights, activations, gradients, CUDA kernels, and other implementation-specific details.

Table 2 shows the memory usage of FO-SGD (LoRA-FA), MeZO (LoRA-FA), and MobiZO. The FO-SGD optimizer requires more memory due to storing activations from all intermediate layers, despite minimal gradient and optimizer state storage through PEFT. In contrast, MobiZO slightly increases memory usage due to the increased size of the largest output tensor during the forward pass and instantiation of multiple sets of LoRA trainable parameters, yet it still demands significantly less memory than the FO optimizer. For instance, with Llama2-7B, a sequence length of 256, and a batch size of 16, memory usage increases from 13.55 GB to 14.53 GB for MobiZO, whereas FO requires over 40 GB. FO over full parameter space requires even much more memory, going beyond the memory capacity of edge devices.

End-to-end Wall-clock Time Speedup. Figure 3 shows the end-to-end wall-clock time for fine-tuning TinyLlama-1.1B and Llama2-7B using MeZO and MobiZO for 20,000 steps across various tasks. By applying PEFT methods, both MeZO and MobiZO reduce training time by minimizing sequential processing of model parameters, a benefit that becomes more pronounced with larger models such as Llama2-7B. MobiZO further improves training runtime through inner-loop and outer-loop parallelization achieving speedups of up to $4.3\times$ over MeZO (Full) and up to $1.9\times$ over MeZO (LoRA-FA) baselines.

Additional system profiling ablation studies, including runtime breakdown under different fixed sequence length and batch size configurations, as well as under different quantization schemes, are available in Appendix G.

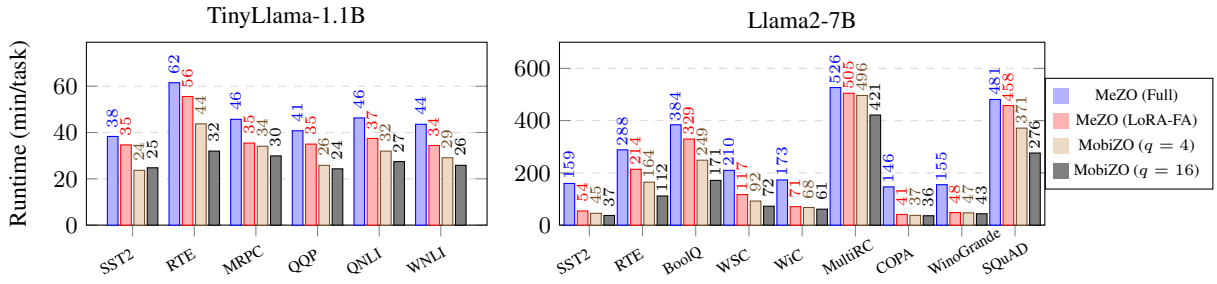


Figure 3: End-to-end wall-clock time of fine-tuning TinyLlama-1.1B and Llama2-7B for various configurations across tasks. MobiZO achieves up to 4.3× speedup compared to MeZO (full) under the same computational budget.

Sequence length	64				128			
Batch size	1	2	4	8	1	2	4	8
TinyLlama-1.1B								
MeZO (LoRA-FA)	0.69	0.71	0.89	1.28	0.70	0.88	1.27	2.18
MobiZO	0.43	0.49	0.69	1.15	0.49	0.69	1.13	2.00
Speedup ratio	1.62	1.45	1.29	1.12	1.42	1.29	1.12	1.09
Llama2-7B								
MeZO (LoRA-FA)	3.10	3.37	4.44	6.46	3.37	4.44	6.47	10.83
MobiZO	1.69	2.22	3.22	5.38	2.22	3.22	5.37	8.60
Speedup ratio	1.83	1.52	1.38	1.20	1.52	1.38	1.21	1.26

Table 3: Runtime (sec/step) and speedup ratio of inner-loop parallelization on Jetson GPU backend for TinyLlama-1.1B and Llama2-7B with NF4 quantization.

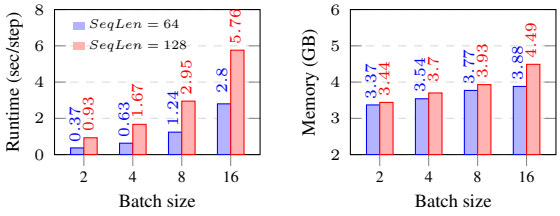


Figure 4: Runtime and memory usage per step on Android NPU backend for TinyLlama-1.1B across different batch sizes and sequence lengths.

4.3 On-Device Training Experiments

For on-device training experiments, we begin with a sanity check to verify per-step loss values on two edge platforms: the NVIDIA Jetson Nano Orin (8GB) GPU and the OnePlus 12 smartphone (12GB) NPU backend. This ensures that both platforms yield the same output given the same input as those observed on the server side. Detailed edge system specifications and experimental setups are provided in Appendix H. After verification, we measure and report the runtime per step of MobiZO across different fixed sequence length and batch size configurations, following the same setup in Section 4.2. Due to out-of-memory issues, FO training is omitted from on-device experiments.

On the Jetson platform, which runs on a Linux system, we use the PyTorch library for model forward passes. Table 3 shows the speedup achieved through MobiZO with inner-loop parallelization with NF4 weight-only quantization, showing up to 1.83× performance improvement. Since MobiZO is fully compatible with Q-LoRA (Dettmers et al., 2023), we also verify that weight-only quantization does not significantly degrade accuracy, as demonstrated in Appendix F.

On the smartphone platform, which operates on Android OS without PyTorch support, we repurpose the latest ExecuTorch library (v0.6.0) to perform MobiZO fine-tuning through integrating the Mobile MP-LoRA module as described in Section 3. Since we do not modify the runtime code on the edge device, vanilla MeZO baseline experiments are omitted due to incompatibility. Additionally, due to current limitations in ExecuTorch's support for weight-only quantization, we run TinyLlama-1.1B in FP16 mode on the NPU backend. While Qualcomm's NPU is optimized for low power rather than raw throughput (Qualcomm, 2024), Executorch on the Android platform achieves comparable efficiency to Pytorch on the Jetson CUDA platform at smaller batch sizes. As shown in Figure 4, with an effective batch size of 16, the NPU backend takes 5.76 seconds for one step with a sequence length of 128, whereas Jetson GPU completes it in 2.00 seconds.

5 Conclusion

This work introduces MobiZO, a parallelized gradient estimation technique for efficient LLM fine-tuning on edge devices. By leveraging outer and inner loop parallelism, MobiZO improves accuracy while reducing training time and memory usage on both server and edge platforms. It enables feasible real-time on-device tuning, and its integration with ExecuTorch via Mobile MP-LoRA demonstrates broad portability on Android NPUs.

6 Limitations

While MobiZO enables efficient on-device LLM fine-tuning, it has several limitations. First, MobiZO is tailored for the randomized gradient estimator in ZO optimization. Extending it to other ZO methods, such as variance-reduced optimizers or adaptive query selection, could further improve convergence speed. Second, thermal constraints on edge devices limit sustained compute performance, leading to throttling during prolonged fine-tuning. Future work will explore thermal-aware scheduling to maintain performance under temperature fluctuations. Third, MobiZO assumes static computational settings, whereas edge environments often have dynamic compute and memory resource constraints. Adapting query count, batch size, or precision in response to runtime conditions is a promising direction.

References

Stephen H. Bach, Victor Sanh, Zheng-Xin Yong, Albert Webson, Colin Raffel, Nihal V. Nayak, Abheesht Sharma, and et al. 2022. Promptsource: An integrated development environment and repository for natural language prompts. *Preprint*, arXiv:2202.01279.

Pin-Yu Chen, Huan Zhang, Yash Sharma, Jinfeng Yi, and Cho-Jui Hsieh. 2017. Zoo: Zeroth order optimization based black-box attacks to deep neural networks without training substitute models. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*.

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

Yae Jee Cho, Luyang Liu, Zheng Xu, Aldi Fahrezi, and Gauri Joshi. 2024. Heterogeneous lora for federated fine-tuning of on-device foundation models. *Preprint*, arXiv:2401.06432.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, and et al. 2022. Palm: Scaling language modeling with pathways. *Preprint*, arXiv:2204.02311.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. 2018. Think you have solved question answering? try arc, the ai2 reasoning challenge. *Preprint*, arXiv:1803.05457.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. QLoRA: Efficient finetuning of quantized LLMs. In *Thirty-seventh Conference on Neural Information Processing Systems*.

John C. Duchi, Michael I. Jordan, Martin J. Wainwright, and Andre Wibisono. 2015. Optimal rates for zero-order convex optimization: The power of two function evaluations. *IEEE Transactions on Information Theory*, 61(5):2788–2806.

Tanmay Gautam, Youngsuk Park, Hao Zhou, Parameswaran Raman, and Wooseok Ha. 2024. Variance-reduced zeroth-order methods for fine-tuning language models. In *5th Workshop on practical ML for limited/low resource settings*.

Gemini-Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M. Dai, Anja Hauth, Katie Millican, David Silver, and et al. 2024. Gemini: A family of highly capable multimodal models. *Preprint*, arXiv:2312.11805.

Google. 2020. Tensorflow lite guide.

Wentao Guo, Jikai Long, Yimeng Zeng, Zirui Liu, Xinyu Yang, Yide Ran, Jacob R. Gardner, Osbert Bastani, Christopher De Sa, Xiaodong Yu, Beidi Chen, and Zhaozhuo Xu. 2025. Zeroth-order fine-tuning of LLMs with transferable static sparsity. In *The Thirteenth International Conference on Learning Representations*.

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. 2022. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Dawid Jan Kopiczko, Tijmen Blankevoort, and Yuki M Asano. 2024. VeRA: Vector-based random matrix adaptation. In *The Twelfth International Conference on Learning Representations*.

Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The power of scale for parameter-efficient prompt tuning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

Xiang Lisa Li and Percy Liang. 2021. Prefix-tuning: Optimizing continuous prompts for generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing*.

Shih-Yang Liu, Chien-Yi Wang, Hongxu Yin, Pavlo Molchanov, Yu-Chiang Frank Wang, Kwang-Ting Cheng, and Min-Hung Chen. 2024a. DoRA: Weight-decomposed low-rank adaptation. In *Forty-first International Conference on Machine Learning*.

Yong Liu, Zirui Zhu, Chaoyu Gong, Minhao Cheng, Cho-Jui Hsieh, and Yang You. 2024b. Sparse mezo: Less parameters for better performance in zeroth-order llm fine-tuning. *Preprint*, arXiv:2402.15751.

Z Liu, J Lou, W Bao, Y Hu, B Li, Z Qin, and K Ren. 2024c. Differentially private zeroth-order methods for scalable large language model finetuning. *Preprint*, arXiv:2402.07818.

Kai Lv, Yuqing Yang, Tengxiao Liu, Qipeng Guo, and Xipeng Qiu. 2024. Full parameter fine-tuning for large language models with limited resources. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics.

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 *Thirty-seventh Conference on Neural Information Processing Systems*.

Meta-AI. 2024a. Executorch.

Meta-AI. 2024b. Executorch kernel registration.

Meta-AI. 2024c. Llama 3.2: Revolutionizing edge ai and vision with open, customizable models. Accessed: 2025-05-12.

Yurii Nesterov and Vladimir Spokoiny. 2017. Random gradient-free minimization of convex functions. *Foundations of Computational Mathematics*.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, and et al. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Dan Peng, Zhihui Fu, and Jun Wang. 2024. PocketLLM: Enabling on-device fine-tuning for personalized LLMs. In *Proceedings of the Fifth Workshop on Privacy in Natural Language Processing*. Association for Computational Linguistics.

Qualcomm. 2024. Hexagon dsp sdk documentation. Accessed: 2025-05-12.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. 2021. Winogrande: an adversarial winograd schema challenge at scale. *Commun. ACM*.

Tianxiang Sun, Zhengfu He, Hong Qian, Yunhua Zhou, Xuanjing Huang, and Xipeng Qiu. 2022. BBTv2: Towards a gradient-free future with large language models. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, and et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *Preprint*, arXiv:2307.09288.

Zhongwei Wan, Xin Wang, Che Liu, Samiul Alam, Yu Zheng, Jiachen Liu, Zhongnan Qu, Shen Yan, Yi Zhu, Quanlu Zhang, Mosharaf Chowdhury, and Mi Zhang. 2024. Efficient large language models: A survey. *Transactions on Machine Learning Research*.

Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2020. Superglue: A stickier benchmark for general-purpose language understanding systems. *Preprint*, arXiv:1905.00537.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2019. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *International Conference on Learning Representations*.

Xiaoxing Wang, Wenxuan Guo, Jianlin Su, Xiaokang Yang, and Junchi Yan. 2022. ZARTS: On zero-order optimization for neural architecture search. In *Advances in Neural Information Processing Systems*.

Mengwei Xu, Dongqi Cai, Yaozong Wu, Xiang Li, and Shangguang Wang. 2024. FwdLLM: Efficient federated finetuning of large language models with perturbed inferences. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*.

Yifan Yang, Kai Zhen, Ershad Banijamal, Athanasios Mouchtaris, and Zheng Zhang. 2024. Adazeta: Adaptive zeroth-order tensor-train adaption for memory-efficient large language models fine-tuning. *Preprint*, arXiv:2406.18060.

Wangsong Yin, Mengwei Xu, Yuanchun Li, and Xuanzhe Liu. 2024. Llm as a system service on mobile devices. *Preprint*, arXiv:2403.11805.

Longteng Zhang, Lin Zhang, Shaohuai Shi, Xiaowen Chu, and Bo Li. 2023. Lora-fa: Memory-efficient low-rank adaptation for large language models fine-tuning. *Preprint*, arXiv:2308.03303.

Peiyuan Zhang, Guangtao Zeng, Tianduo Wang, and Wei Lu. 2024a. Tinyllama: An open-source small language model. *Preprint*, arXiv:2401.02385.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, and et al. 2022. Opt: Open pre-trained transformer language models. *Preprint*, arXiv:2205.01068.

Yihua Zhang, Pingzhi Li, Junyuan Hong, Jiaxiang Li, Yimeng Zhang, Wenqing Zheng, Pin-Yu Chen, Jason D. Lee, Wotao Yin, Mingyi Hong, Zhangyang Wang, Sijia Liu, and Tianlong Chen. 2024b. Revisiting zeroth-order optimization for memory-efficient LLM fine-tuning: A benchmark. In *Forty-first International Conference on Machine Learning*.

Jiawei Zhao, Zhenyu Zhang, Beidi Chen, Zhangyang Wang, Anima Anandkumar, and Yuandong Tian. 2024. Galore: Memory-efficient LLM training by gradient low-rank projection. In *Forty-first International Conference on Machine Learning*.

Ligeng Zhu, Lanxiang Hu, Ji Lin, Wei-Chen Wang, Wei-Ming Chen, and Song Han. 2023. Pockengine: Sparse and efficient fine-tuning in a pocket. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

A MeZO Algorithm and Its Limitation

Algorithm 3 MeZO with $q > 1$.

1: **Input:** parameters $\boldsymbol{\theta} \in \mathbb{R}^d$, loss $\mathcal{L} : \mathbb{R}^d \to \mathbb{R}$, step budget T, function query budget q, perturbation scale ϵ, learning rate η
2: **for** $t = 1, \ldots, T$ **do**
3: **for** $i = 1, \ldots, q$ **do**
4: `seeds, projected_grads = []`
5: Sample batch $\boldsymbol{\mathcal{B}} \subset \mathcal{D}$ and random seed s
6: $\boldsymbol{\theta} =$ `PerturbParameters`$(\boldsymbol{\theta}, \epsilon, s)$
7: $\ell_+ = \mathcal{L}(\boldsymbol{\theta}; \boldsymbol{\mathcal{B}})$
8: $\boldsymbol{\theta} =$ `PerturbParameters`$(\boldsymbol{\theta}, -2\epsilon, s)$
9: $\ell_- = \mathcal{L}(\boldsymbol{\theta}; \boldsymbol{\mathcal{B}})$
10: $\boldsymbol{\theta} =$ `PerturbParameters`$(\boldsymbol{\theta}, \epsilon, s)$
11: `proj_grads[i]` $= \frac{\ell_+ - \ell_-}{2\epsilon}$
12: `seeds[i]` $= s$
13: **end for**
14: **for** $i = 1, \ldots, q$ **do**
15: Reset random generator with `seeds[i]`
16: **for** $\boldsymbol{\theta}_j \in \boldsymbol{\theta}$ **do**
17: $z \sim \mathcal{N}(0, 1)$
18: $\boldsymbol{\theta}_j = \boldsymbol{\theta}_j - \frac{\eta_t}{q} \times$ `proj_grads[i]` $\times z$
19: **end for**
20: **end for**
21: **end for**
22: **Function** `PerturbParameters`$(\boldsymbol{\theta}, \epsilon, s)$
23: Reset random number generator with seed s
24: **for** $\boldsymbol{\theta}_j \in \boldsymbol{\theta}$ **do**
25: $z \sim \mathcal{N}(0, 1)$
26: $\boldsymbol{\theta}_j = \boldsymbol{\theta}_j + \epsilon z$
27: **end for**
28: **End Function**

We evaluate the runtime efficiency of the MeZO optimizer, outlined in Algorithm 3, which is adapted from the original work. MeZO employs a random seed trick to eliminate the need for storing random noise, reducing peak memory usage.

In each iteration, MeZO proceeds through four distinct loops. First, it introduces positive noise into the trainable parameters (line 6), followed by perturbing the weights in the opposite direction using the same noise (line 8). Next, the weights are restored to their original state before the update (line 10), and finally, the computed gradients are applied to update the weights (line 18).

This method reduces memory overhead from $O(d)$ to $O(1)$ by avoiding the storage of random noise. However, the runtime cost escalates from $O(1)$ to $O(d)$ because each parameter update requires individual processing, which cannot be efficiently parallelized. In practical settings, especially with LLMs, iterating over the full parameter set four times per update can significantly slow down the training process, thus negating the benefits of eliminating backpropagation.

In contrast, PyTorch's FO optimizers utilize a *foreach* implementation by default. This method aggregates all layer weights into a single tensor during parameter updates, which speeds up the computation. However, this approach also increases the memory usage by $O(d)$, as it requires maintaining a copy of the entire gradients for the parameters update.

Table 4 compares the runtime of the Llama2-7B model using both FO-SGD and MeZO-SGD optimizers ($q = 1$) over the full parameter space across various batch sizes and sequence lengths on the same standard benchmark introduced in Section 4.2. The FO optimizer is run with FP16 mixed-precision training, while MeZO uses pure FP16 to maximize computational speed. To avoid out-of-memory errors, we utilize two NVIDIA A100 (40GB) GPUs for the FO optimizer, which incurs additional GPU communication time in a distributed environment.

Sequence length	64			128			256		
Batch size	1	4	8	1	4	8	1	4	8
FO-SGD	0.17	0.21	0.34	0.19	0.33	0.49	0.18	0.49	0.90
MeZO-SGD ($q = 1$)	0.43	0.48	0.56	0.43	0.56	0.73	0.45	0.73	1.05

Table 4: Runtime (sec/step) of Llama2-7B using FO and MeZO optimizers over full parameter space.

When both the batch size and sequence length are small, MeZO exhibits significantly higher runtime due to the overhead of sequential operations required to apply perturbations and gradients. However, as the batch size and sequence length increase, where forward and backward passes, as well as GPU communication, dominate the runtime, the MeZO optimizer demonstrates improved performance. This behavior highlights the importance of applying PEFT methods with MeZO to mitigate the computation overhead caused by the sequential processing of model parameters.

B Preliminary Experiment of ZO with Different PEFT Methods

We conducted a preliminary experiment by fine-tuning the OPT-1.3B model (Zhang et al., 2022) for 10,000 iterations on the SST2 dataset (Wang et al.,

2019) using ZO-SGD optimizer with different PEFT methods. We use hyperparameter grid search with learning rate $\in \{5e-6, 5e-5, 5e-4, 5e-3\}$ and $\epsilon \in \{1e-3, 1e-2\}$. LoRA (Hu et al., 2022), LoRA-FA (Zhang et al., 2023), and DoRA (Liu et al., 2024a) are configured with $r = 16$, and VeRA (Kopiczko et al., 2024) uses $r = 1024$. The results in Table 5 indicate that the LoRA-FA method outperforms other PEFT methods in terms of accuracy.

PEFT Methods	LoRA	LoRA-FA	DoRA	VeRA
Accuracy	90.9	92.0	90.9	91.4

Table 5: ZO accuracy of OPT-1.3B on SST2 dataset using different PEFT methods.

C Padding Statistics

Figure 5 illustrates how smaller batch sizes (e.g., 2) result in less padding compared to larger batch sizes (e.g., 8), thereby reducing wasted computation.

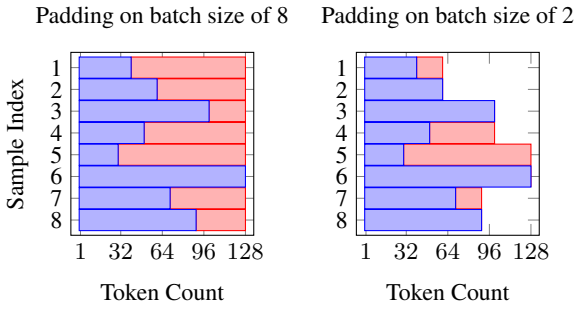


Figure 5: The standard batching approach pads shorter sequences to the maximum sequence length within the batch.

Figure 6 shows the average percentage of padding tokens used across different tasks and batch sizes. A larger batch size of 16 results in a higher percentage of padding tokens across all tasks compared to a batch size of 4. This suggests that smaller batch sizes may help reduce padding overhead, potentially leading to more efficient computation.

D Experiment Setup

D.1 Datasets

We evaluate the performance of the TinyLlama-1.1B model on six tasks from the GLUE dataset (Wang et al., 2019): sentiment analysis (SST2), paraphrase (MRPC and QQP), and natural language inference (QNLI, RTE, and WNLI). For

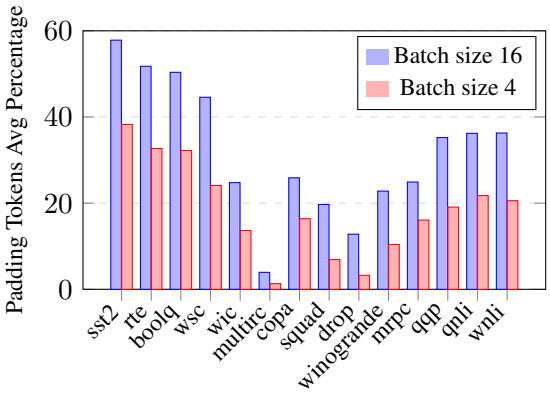


Figure 6: Average percentage of padding tokens for different tasks and batch sizes.

the larger Llama2-7B model, evaluations were performed on two tasks from the GLUE dataset: SST2 and RTE. Additionally, the model was tested on six tasks from the SuperGLUE dataset (Wang et al., 2020), categorized as follows: text classification (BoolQ, WSC, WIC, and MultiRC), and multiple-choice (COPA). We include three additional multiple-choice tasks from the WinoGrande (Sakaguchi et al., 2021), ARC-Easy, and ARC-Challenge (Clark et al., 2018) datasets. For question-and-answering tasks, we utilize the F1 score as a metric, while accuracy metrics are used for the rest. All datasets used in this work are in English.

D.2 Training procedure

We achieve text classification, multiple-choice, and question-and-answering tasks through next-word prediction, using prompt templates based on MeZO (Malladi et al., 2023) and PromptSource (Bach et al., 2022). Table 6 presents the prompt templates used for the datasets in our TinyLlama-1.1B and Llama2-7B experiments. For SST-2, RTE, BoolQ, WSC, WIC, MultiRC, and COPA, we applied the template from MeZO (Malladi et al., 2023). We created templates for MRPC, QQP, QNLI, WNLI, and ARC by following the suggestions from PromptSource (Bach et al., 2022), and we adapted the same template for WinoGrande from (Zhang et al., 2024b).

Unlike MeZO, we compute the loss value of prediction over the entire vocabulary space instead of only the vocabulary space of the ground true. For these tests, we also adopt a low-volume data condition, limiting our samples to 1,000 for training, 500 for validation, and 1,000 for testing, as pro-

posed in the original MeZO work (Malladi et al., 2023). FO-SGD experiments are trained for 1,000 iterations, and performance on the test dataset is evaluated every 100 steps. ZO experiments are trained for 20,000 iterations and performance on the test dataset is evaluated every 500 steps.

D.3 Hyperparameters

We report the hyperparameters searching grids in Table 7. For LoRA hyperparameters, we choose the LoRA rank to be 16 and LoRA alpha to be 32. For MobiZO, with the constant batch size of 16, we search configurations ($q = 1, E = 16$), ($q = 4, E = 4$), and ($q = 16, E = 1$).

Dataset	Type	Prompt
SST-2	cls.	<text> It was terrible/great
RTE	cls.	<premise> Does this mean that "<hypothesis>" is true? Yes or No? Yes/No
MRPC	cls.	Do the following two sentences mean the same thing? Yes or No? Sentence 1: <sentence1> Sentence 2: <sentence2> Yes/No
QQP	cls.	Are these two questions asking the same thing? Yes or No? Question 1: <question1> Question 2: <question2> Yes/No
QNLI	cls.	Does this sentence answer the question? Yes or No? Sentence 1: <sentence1> Sentence 2: <sentence2> Yes/No
WNLI	cls.	Given the first sentence, is the second sentence true? Yes or No? Sentence 1: <sentence1> Sentence 2: <sentence2> Yes/No
BoolQ	cls.	<passage> <question> <answer>? Yes/No
WSC	cls.	<text> In the previous sentence, does the pronoun "<span2>" refer to <span1>? Yes/No
WIC	cls.	Does the word "<word>" have the same meaning in these two sentences? <sent1> <sent2> Yes, No?
MultiRC	cls.	<paragraph> Question: <question> I found this answer "<answer>". Is that correct? Yes or No?
COPA	mch.	<premise> so/because <candidate>
WinoGrande	mch.	<context> <subject> <object>
ARC	cls.	Pick the most correct option to answer the following question: <question> Options: <op1>, <op2>, <op3>, <op4> Answer: <label>

Table 6: The prompt template of the datasets used in the experiments.

TinyLlama-1.1B		
FO (Full)	Batch size	16
	Learning rate	{1e-4, 5e-5, 1e-5}
FO (LoRA-FA)	Batch size	16
	Learning rate	{5e-3, 1e-3, 5e-4}
MeZO (Full)	Batch size	16
	Learning rate	{1e-6, 5e-7, 1e-7}
	ϵ	1e-3
MobiZO	Batch size	16
	q	{1, 2, 4, 8, 16}
	Learning rate	{5e-4, 1e-4, 5e-5, 1e-5}
	ϵ	1e-2
Llama2-7B		
Experiment	Hyperparameters	Values
FO (Full)	Batch size	8
	Learning rate	{1e-4, 5e-5, 1e-5}
FO (LoRA-FA)	Batch size	8
	Learning rate	{5e-3, 1e-3, 5e-4}
MeZO (Full)	Batch size	16
	Learning rate	{1e-6, 5e-7, 1e-7}
	ϵ	1e-3
MobiZO	Batch size	16
	q	{1, 2, 4, 8, 16}
	Learning rate	{5e-4, 1e-4, 5e-5, 1e-5}
	ϵ	1e-2

Table 7: Hyperparameters used for TinyLlama-1.1B and Llama2-7B experiments. Note that MeZO (LoRA-FA) is a special case of MobiZO with $q = 1$.

E Additional FO Experiments

We also provide additional experimental results on FO-Adam in Tables 8 and 9. While FO-Adam can enhance model performance, it introduces a significantly higher memory overhead, particularly when updating all model parameters. This is because Adam maintains two state variables, moment estimates of the first and second order, for each parameter, effectively tripling the memory requirement compared to storing only the model parameters. Therefore, FO-Adam is typically deployed in distributed multi-GPU environments, which further increases runtime due to the overhead of inter-device communication.

Tasks	SST-2	RTE	MRPC	QQP	QNLI	WNLI
Full	91.9	72.5	77.4	82.4	80.8	56.3
LoRA-FA	94.2	82.6	82.3	84.4	86.5	56.3

Table 8: Performance of fine-tuning TinyLlama-1.1B on different tasks with FO-Adam optimizers.

Tasks	SST-2	RTE	BoolQ	WSC	WiC	MultiRC	COPA
Full	92.5	78.7	80.6	63.4	67.2	71.7	81.0
LoRA-FA	96.0	88.1	85.7	79.8	75.1	84.2	87.0

Table 9: Performance of fine-tuning Llama2-7B on different tasks with FO-Adam optimizers.

F Additional ZO Experiments

F.1 OPT model

We conducted additional zeroth-order optimization experiments on the OPT-1.3B model (Zhang et al., 2022), which was also evaluated in the original MeZO study (Malladi et al., 2023). Given our emphasis on deployment in resource-constrained edge environments, we exclude larger models such as OPT-13B and OPT-70B from consideration. As shown in Table 10, MobiZO consistently outperforms MeZO across most tasks, highlighting its effectiveness beyond the LLaMA model family. These results complement our primary evaluations on LLaMA-based models and further demonstrate the general applicability of MobiZO across diverse model architectures.

Tasks	SST-2	RTE	MRPC	QQP	QNLI	WNLI
MeZO (Full)	**92.4**	57.8	70.6	67.7	56.2	59.2
MeZO (LoRA-FA)	89.8	**62.8**	70.6	69.3	58.9	59.2
MobiZO ($q = 4$)	92.0	**62.8**	**71.3**	73.1	65.4	**60.6**
MobiZO ($q = 16$)	91.5	**62.8**	**71.3**	**74.3**	**65.8**	**60.6**

Table 10: Evaluation results on GLUE tasks using OPT-1.3B with different ZO methods.

F.2 Llama3.2-1B model

Llama3.2-1B (Meta-AI, 2024c) is the first small-scale model from the LLaMA family designed specifically for edge deployment. As shown in Table 11, MobiZO continues to outperform the baseline MeZO on the majority of GLUE tasks.

Methods	SST-2	RTE	MRPC	QQP	QNLI	WNLI
MeZO (Full)	91.9	63.9	70.6	75.6	67.7	60.6
MeZO (LoRA-FA)	92.8	64.6	70.3	73.6	60.4	**63.4**
MobiZO ($q = 4$)	**93.8**	64.6	**74.8**	76.2	63.7	**63.4**
MobiZO ($q = 16$)	93.7	**65.3**	73.0	**77.9**	**67.8**	**63.4**

Table 11: Evaluation results on GLUE tasks using Llama3.2-1B with different ZO methods.

F.3 In-depth analysis of trade-offs in RGE

Table 12 summarizes different trade-offs in RGE under a fixed computational budget. One strategy (Row 2) suggested by MeZO compensates for the increased number of queries by reducing the total number of training steps. In this work, we introduce an alternative trade-off (Row 3): increasing the query count while decreasing the batch size, rather than reducing training steps. As demonstrated in Section D, this approach consistently outperforms the trade-offs in Rows 1 and 2. Nonetheless, each training step takes longer than it would with a single-query ($q = 1$) because the gradient estimations are executed sequentially, even though the total compute remains the same. While this trade-off improves final model accuracy, our objective is to maintain high accuracy while minimizing per-step execution time via the parallelism introduced in MobiZO.

Query	Batch size	Training steps	Performance	Wall-clock time
1	B	T	✗	✓
q	B	T/q	✗	✓
q	B/q	T	✓	RGE ✗ / MobiZO ✓

Table 12: Different trade-offs for RGE.

Our motivation for this adjustment stems from the convergence analysis in the MeZO work (Malladi et al., 2023), which shows that reaching ϵ-suboptimality requires

$$t = \mathcal{O}\left(\frac{\ell}{\mu}\left(1+\frac{r}{n}\right)\left(1+\frac{\alpha}{\mu B}\right)\log\left(\frac{\mathcal{L}(\theta_0)-\mathcal{L}^*}{\epsilon}\right)\right),$$

where n is the number of zeroth-order queries, B is the batch size, r is the local effective rank of the Hessian, and α bounds the trace of the gradient-covariance matrix. Among these variables, ℓ, μ, and α are properties of the loss landscape that are

generally not controllable, but n and B are hyperparameters that can be tuned. We hypothesize that increasing n has a greater impact on convergence speed than increasing B, especially in large language models with low effective Hessian rank (Malladi et al., 2023). Given the complexity of these models, analytically estimating r, α, or μ is intractable; we therefore validate this hypothesis through empirical results.

To support our analysis, Table 13 presents averaged performance across different values of q for both TinyLlama-1.1B and Llama2-7B on their corresponding tasks described in Section 4. As q increases and E decreases proportionally (keeping total B fixed), model performance consistently improves compared to single query RGE, confirming the benefit of our trade-off strategy. However, the improvement does not persist uniformly as q continues to grow. Therefore, in practice, we recommend using $q = 4$ or $q = 16$, which offer a favorable balance between convergence quality and hyperparameter search.

q	1	2	4	8	16
TinyLlama-1.1B	70.33	73.62	74.10	74.82	74.80
Llama2-7B	71.62	71.75	72.87	72.81	72.83

Table 13: Average performance of TinyLlama-1.1B and Llama2-7B models under varying numbers of queries q in MebiZO.

F.4 MobiZO with weight-only quantization

To evaluate the compatibility of MobiZO with quantized models, we apply weight-only quantization at both 8-bit and 4-bit precision levels to the TinyLlama-1.1B model, building on Q-LoRA (Dettmers et al., 2023), which has shown that NF4 quantization is effective for freezing pretrained weights. In this setup, weights are stored in low-bit formats to reduce memory usage and are dequantized to FP16 during the forward pass. As shown in Table 14 and Table 1, the fine-tuning performance remains comparable to that of full FP16 training, confirming that low-bit quantization does not significantly affect model accuracy.

G Ablation Studies on System Performance of MobiZO

G.1 Efficiency of outer-loop parallelization

We measure the runtime and memory usage of MobiZO, implemented using outer-loop parallelization only for the Llama2-7B model across different

Methods \ Tasks	**SST-2**	**RTE**	**MRPC**	**QQP**	**QNLI**	**WNLI**
8-bit weights						
MobiZO ($q = 4$)	88.3	70.3	73.4	76.2	74.8	60.3
MobiZO ($q = 16$)	89.4	72.2	73.5	76.9	76.5	59.7
4-bit weights						
MobiZO ($q = 4$)	88.0	70.1	73.2	76.1	74.6	59.6
MobiZO ($q = 16$)	88.9	72.1	73.0	77.1	76.4	58.2

Table 14: Performance of TinyLlama-1.1B on GLUE tasks under 8-bit and 4-bit weight-only quantization.

effective batch size and fixed sequence lengths configurations. As shown in Table 15, the runtime remains nearly identical across different combinations of the number of queries q and effective batch size E, given that the batch size remains constant at $B = 16$, which indicates our outer-loop parallelization implementation does not incur computation overhead. Peak memory usage increases slightly due to the instantiation of multiple LoRA trainable parameters at each layer.

Sequence length	64			128			256		
q	1	4	16	1	4	16	1	4	16
Effective batch size	16	4	1	16	4	1	16	4	1
Runtime (sec/step)	0.18	0.20	0.19	0.35	0.37	0.32	0.69	0.67	0.71
Memory (GB)	12.61	12.69	12.81	12.64	12.80	13.14	12.70	13.04	13.53

Table 15: System performance of outer-loop parallelization for Llama2-7B under the same batch size of 16.

G.2 Efficiency of inner-loop parallelization

We measure the runtime and memory usage of MobiZO, implemented using inner-loop parallelization only for the Llama2-7B model across fixed different sequence length and batch size configurations. As shown in Table 16, the runtime speedup is up to $1.79\times$ at a sequence length of 64 and batch size of 1. This improvement is primarily due to reusing model weights across two forward passes, which reduces cache access and increases operation intensity. However, the benefits diminish as operation intensity increases and the system becomes compute-bound.

Sequence length	64			128			256		
batch size	1	8	16	1	8	16	1	8	16
MeZO ($q = 1$, LoRA-FA)	0.07	0.11	0.18	0.07	0.19	0.35	0.07	0.35	0.69
MobiZO ($q = 1$, inner)	0.04	0.10	0.18	0.04	0.18	0.34	0.06	0.34	0.67

Table 16: Runtime (sec/step) of inner-loop parallelization for Llama2-7B under different sequence length and batch size configurations.

Additionally, we evaluate the speedup achieved by inner-loop parallelization under weight-only INT8 and NF4 quantization. As illustrated in Figure 7, inner-loop parallelization achieves the greatest speedup in conjunction with NF4 quantization,

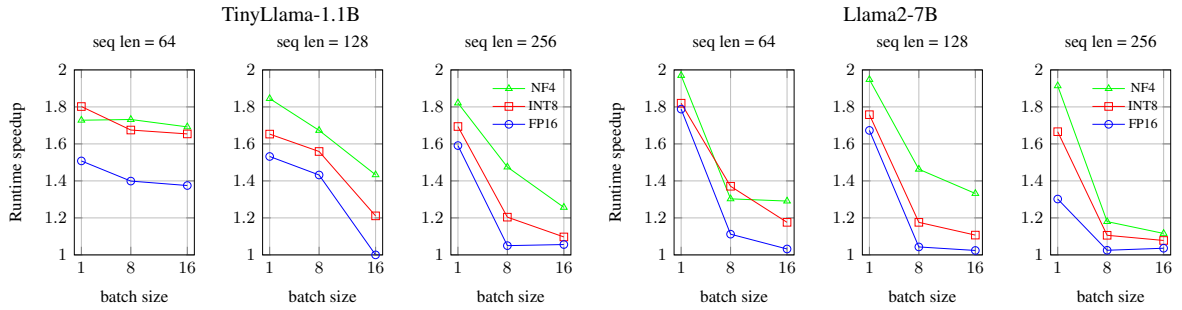


Figure 7: Runtime speedup per training step of TinyLlama-1.1B and Llama2-7B for different quantization methods, sequence lengths, and batch sizes.

reaching up to a 1.97× improvement over the sequential execution of two forward passes. Since NF4 dequantization is more computationally intensive than INT8 during forward passes, inner-loop parallelization enhances efficiency by dequantizing weights only once per training step, reducing the overhead from repeated dequantization.

G.3 End-to-end training efficiency

Tables 18 - 21 provide additional details on per-task runtime and memory usage to complement the experimental results in Table 1. In these tables, MeZO (Full) represents the baseline configuration in which all model parameters are updated during training. For MeZO (LoRA-FA), results are presented for both the standard implementation without optimizations and a variant enhanced with inner-loop parallelization. For MobiZO, results are shown for two setups: one using only outer-loop parallelization and another that combines both inner and outer-loop parallelization strategies. As noted in Section 4.2, when both parallelization strategies are enabled, MobiZO achieves speedups of up to 4.3× over MeZO (Full) and up to 1.9× over MeZO (LoRA-FA).

Regarding memory usage, enabling both inner and outer-loop parallelization results in higher memory consumption for both models compared to configurations using only outer-loop parallelization. This increase is due to the concurrent computation of two forward passes when inner-loop parallelization is enabled. Specifically, for Llama2-7B, tasks like MultiRC see an increase in memory usage of up to 33% when using inner-loop parallelization due to larger sequence length. Despite this increase, the memory efficiency remains within acceptable bounds.

H Edge Devices Specifications

Table 17 presents the specifications of the edge computing devices used in the experiments, detailing the CPU, memory, and accelerator components.

Device	CPU	Memory	Accelerator
NVIDIA Jetson Orin Nano	6× 1.5GHz Cortex-A78AE	8GB 68GB/s LPDDR5	1024-core Ampere GPU 625MHz
OnePlus 12	1× 3.3GHz Cortex-X4 3× 3.2GHz Cortex-A720 2× 3.0GHz Cortex-A720 2× 2.3GHz Cortex-A520	12GB 77GB/s LPDDR5	Hexagon NPU

Table 17: Edge devices used in the experiments.

For experiments on the Android NPU backend, we use the Qualcomm AI Engine Direct SDK (v2.28.0.241029) and Android NDK (r26d) to compile the kernel library.

Methods \ Tasks	SST-2	RTE	MRPC	QQP	QNLI	WNLI
MeZO (Full) ($q = 1$)	38.31	61.51	45.71	40.76	46.30	43.57
MeZO (LoRA-FA) ($q = 1$)						
standard	34.66	55.53	35.45	35.00	37.44	34.40
inner	23.55	54.07	35.72	28.76	36.59	33.22
MobiZO ($q = 4$)						
outer only	36.27	45.22	36.90	36.19	35.33	37.23
inner + outer	23.68	43.75	34.07	25.83	31.97	29.09
MobiZO ($q = 16$)						
outer only	35.57	38.18	35.38	35.19	35.86	35.34
inner + outer	24.77	31.98	29.90	24.31	27.43	25.84

Table 18: Runtime (min/task) of fine-tuning TinyLlama-1.1B across different tasks using different ZO methods.

Methods \ Tasks	SST-2	RTE	BoolQ	WSC	WiC	MultiRC	COPA	WinoGrande
MeZO (Full) ($q = 1$)	159.44	288.10	384.07	209.72	173.01	526.49	146.40	154.74
MeZO (LoRA-FA) ($q = 1$)								
standard	54.20	213.81	329.46	116.79	70.55	504.74	40.77	48.07
inner	55.22	210.30	322.64	118.03	72.75	505.54	36.57	48.62
MobiZO ($q = 4$)								
outer only	49.11	165.53	251.63	91.87	66.55	505.70	44.65	49.01
inner + outer	45.17	164.21	248.55	92.17	67.52	496.32	37.38	46.89
MobiZO ($q = 16$)								
outer only	43.91	111.80	171.84	71.14	60.31	438.24	41.96	46.41
inner + outer	36.99	111.54	171.14	72.40	61.10	421.41	35.91	43.41

Table 19: Runtime (min/task) of fine-tuning Llama2-7B across different tasks using different ZO methods.

Methods \ Tasks	SST-2	RTE	MRPC	QQP	QNLI	WNLI
MeZO (Full) ($q = 1$)	2.56	3.38	2.74	2.74	3.17	2.77
MeZO (LoRA-FA) ($q = 1$)						
standard	2.35	3.27	2.63	2.63	3.06	2.66
inner	2.63	4.46	3.18	3.18	4.04	3.24
MobiZO ($q = 4$)						
outer only	2.37	3.29	2.65	2.65	3.07	2.68
inner + outer	2.67	4.50	3.22	3.22	4.07	3.28
MobiZO ($q = 16$)						
outer only	2.44	3.18	2.72	2.69	3.14	2.75
inner + outer	2.81	4.28	3.36	3.30	4.22	3.42

Table 20: Peak memory usage (GB) of fine-tuning TinyLlama-1.1B across different tasks using different ZO methods.

Methods \ Tasks	SST-2	RTE	BoolQ	WSC	WiC	MultiRC	COPA	WinoGrande
MeZO (Full)	13.64	16.23	18.39	14.51	13.82	18.39	13.60	13.60
MeZO (LoRA-FA) ($q = 1$)								
standard	13.41	16.00	18.16	14.27	13.58	18.16	12.98	13.15
inner	14.23	19.41	23.73	15.96	14.57	23.73	13.37	13.71
MobiZO ($q = 4$)								
outer only	13.53	16.12	18.28	14.40	13.71	18.28	13.10	13.27
inner + outer	14.47	19.65	23.97	16.20	14.82	23.97	13.61	13.95
MobiZO ($q = 16$)								
outer only	14.03	16.10	18.77	14.92	14.20	18.77	13.59	13.77
inner + outer	15.45	19.59	24.95	17.17	15.79	24.95	14.58	14.93

Table 21: Peak memory usage (GB) of fine-tuning Llama2-7B across different tasks using different ZO methods.