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

Anonymous ACL submission

Abstract

Large language models (LLMs) have demonstrated remarkable capabilities across multiple domains and fine-tuning is an essential step in adapting a pre-trained model to downstream tasks with user 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, computational demands, and limited infrastructure support for backpropagation. We observe that inference engines (e.g., ExecuTorch) can be repurposed for fine-tuning by leveraging zeroth-order (ZO) optimization. Memory efficient ZO (MeZO) proposes to estimate gradient using only two forward passes, reducing the memory cost to the same as inference. However, ZO methods require multiple queries of gradient estimations at each training step to achieve good performance. Since multi-query gradient estimation requires multiple independent forward passes, our key insight is that these can be executed in parallel. To this end, we propose *parallelized randomized gradient estimation* (P-RGE), which employs a novel design based on parameter-efficient fine-tuning techniques to achieve high-speed training while still harvesting the model performance boost, without increasing computational cost. Moreover, it seamlessly extends inference engines without altering their underlying runtime code and only minimal server-side modifications are needed. Through extensive experiments, we demonstrate that P-RGE delivers substantial gains in fine-tuning efficiency and accuracy, thereby enabling real-time, on-device personalization of LLMs under strict memory and compute budgets. Code available at: `anonymous.4open.science/r/PRGE-ARR-4FE5`.

1 Introduction

Large Language Models (LLMs) have demonstrated strong performance across various applications, including chatbots and image generation (OpenAI et al., 2024; Chowdhery and et al., 2022; Anil and 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, gradient, activation, and optimizer state (Wan et al., 2024). This memory overhead makes fine-tuning infeasible on resource-constrained devices such as smartphones and edge platforms (Zhu et al., 2023; 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.

Recent advancements in parameter-efficient fine-tuning (PEFT) (Hu et al., 2022; Houlsby et al., 2019) and memory-efficient strategies (Dettmers et al., 2023; Lv et al., 2024) reduce the memory footprint of model weights and optimizer states. However, the storage of activation during backpropagation remains a bottleneck, particularly for large models (Lv et al., 2024). Techniques such as gradient checkpointing (Chen et al., 2016) and mixed-precision training (Micikevicius et al., 2018) help mitigate activations memory usage but still rely on automatic differentiation, which is not well-supported in edge environments.

Unlike conventional first-order (FO) optimization methods, zeroth-order (ZO) methods do not require backpropagation to compute gradients (Duchi et al., 2015; Nesterov and Spokoiny, 2017). The memory-efficient ZO (MeZO) method is proposed to further reduce memory overhead and relies on only two forward passes to estimate the gradient along one random direction (i.e., single-query gradient estimation) (Malladi et al., 2023). Therefore, ZO methods eliminate the need to store activations and gradients, making them particularly attractive for memory and computation-efficient LLM fine-tuning (Zhang et al., 2024b).

However, models trained using ZO methods typ-

ically exhibit lower accuracy than those using FO methods. To bridge this gap, gradient estimation at each training step can be improved by incorporating multiple queries (i.e., averaging the gradient estimations along multiple random directions) (Zhang et al., 2024b). In a multi-query setting, each query is executed sequentially. Given a fixed computational budget (i.e., fixed total FLOPs), ZO methods like MeZO attempt to scale up the number of queries while proportionally reducing the number of training steps and find that this trade-off of using multi-query does not necessarily lead to better final model performance or reduction in total wall-clock training time (Malladi et al., 2023).

In this work, we aim to leverage multi-query gradient estimation to boost model performance while still achieving speedups under a fixed computational budget by enhancing computational efficiency, particularly in resource-constrained scenarios. We observe that reducing batch size while keeping the number of training steps unchanged improves final model performance in a multi-query setting, yet it also prolongs training because each query is processed sequentially. To address this, we eliminate serial processing and maximize parallel execution of forward passes, both across queries and within each query at every training step. Our key contributions are as follows:

- We introduce a *parallelized randomized gradient estimation* (P-RGE) technique that achieves high parallel efficiency by leveraging both **outer-loop** and **inner-loop parallelization**. By executing multiple forward passes in parallel, P-RGE effectively amortizes the memory access cost of loading model parameters, thereby reducing training time while improving model performance.

- We implement a P-RGE PEFT module that seamlessly integrates P-RGE into **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 demonstrate 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 **9.87**$\%$ improvement in accuracy.

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:

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

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

ZO-SGD replaces FO gradients with ZO gradient estimates: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta\hat{\nabla}\mathcal{L}(\boldsymbol{\theta}; \mathcal{B}_t)$, with learning rate η at timestpe 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.

3 Method

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

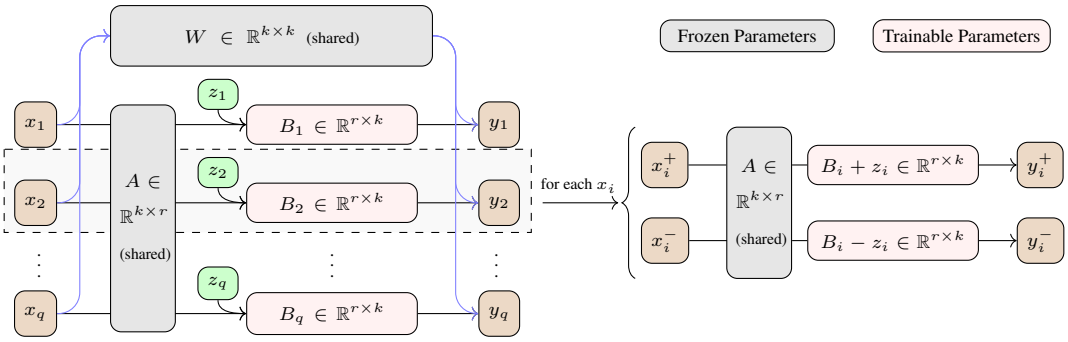
Table 1: Different trade-offs for RGE.

Table 1 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. Here, we introduce an alternative trade-off: increasing the query count while decreasing the batch size, rather than reducing training steps. We later demonstrate that this approach consistently outperforms the strategy in Row 3. 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.

Algorithm 1 Parallelized Randomized Gradient Estimation (P-RGE) 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
      $\boldsymbol{\theta}_l^{(k)} = \boldsymbol{\theta}_l + k\epsilon \boldsymbol{z}_i$
      $\ell^{(k)} = \mathcal{L}((\boldsymbol{\theta}_l^{(k)}, \boldsymbol{\theta}_f); \boldsymbol{\mathcal{B}})$
    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

In general, performing q-query RGE requires $2q$ forward passes. A naive implementation (detailed in Appendix A) would run these forward passes in a two-level nested loop: the outer loop iterates over each random direction, and the inner loop executes the two forward passes required for gradient estimation. Such a sequential approach repeatedly loads model parameters into computational units, causing significant overhead. However, these passes are entirely independent, distinguished only by the random perturbations applied to the trainable parameters. Building on this key observation, we propose *parallelized randomized gradient estimation* (P-RGE) to address the runtime overhead inherent in multi-query RGE. P-RGE 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.



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

Figure 1: Overview of the proposed P-RGE method. Model weights are reused across multiple forward passes, reducing the expensive cost of external memory access and improving runtime per training step.

3.1 Outer-loop Parallelization

One straightforward solution to perform multiple gradient estimations in parallel is to create multiple copies of the model inputs and trainable parameters, then execute all forward passes in parallel (Algorithm 1, line 4). However, this approach incurs large memory overhead from duplicating weights, as well as computational burdens from parallelizing over multiple queries. Particularly, if each query step already maximally uses the available compute resources, then parallelly executing multiple queries is infeasible.

To mitigate memory overhead and avoid sequential parameter operations, we employ PEFT methods, which reduce the number of trainable parameters. The random seed trick from MeZO (Malladi et al., 2023) shrinks the memory overhead from $O(d)$ to $O(1)$ but also increases runtime associated with parameter perturbation from $O(1)$ to $O(d)$, as each parameter must be updated individually. This can substantially slow training for large models, potentially offsetting the speedups gained from removing backpropagation. Hence, reducing the number of trainable parameters using PEFT is crucial for enabling MeZO for overall computational efficiency.

Our preliminary experiments (see Appendix B) indicate that combining ZO with LoRA-FA performs better than alternatives such as DoRA (Liu et al., 2024a) and VeRA (Kopiczko et al., 2024), making LoRA-FA our default choice, but our approach can be adapted to any other PEFT method.

To further improve the memory usage efficiency we propose to carefully schedule the operations to minimize needless data movement between memory and compute units. As illustrated in Figure 1(a), we start by replicating the model input batch (x) q times. We keep both $\mathbf{W}$ and LoRA-$\mathbf{A}$ matrices fixed, and replicate only the LoRA-$\mathbf{B}$ matrix q times. Each $\mathbf{B}$ copy is then perturbed by distinct random noise during its forward pass. Next, the replicated inputs $(x_1, x_2, \ldots, x_q)$ undergo batched matrix multiplication with their respective LoRA-$\mathbf{B}$ copies.

When performing P-RGE with $q > 1$, we proportionally reduce the batch size to $B/q = E$, where E is referred to as the *effective batch size*. For instance, if the original setting uses $q = 1$ and $B = 16$, our approach can increase q to 4 while setting $E = 4$, thereby maintaining $qE = B$.

One benefit of outer-loop parallelization is that it also increases parallelism across queries and improves data locality for model weights. By loading the required weights once and reusing them for all queries, costly external memory accesses are amortized across different queries, and runtime per training step is also reduced to the same level as the original setting of $q = 1$ (Table 1 Row 1).

Reducing B in a multi-query setting has the added benefit of reducing padding tokens as shown in Figure 2. Typically, the number of padding tokens increases with the batch size as the variability of sequence length increases with batch size, and sequences of varying lengths are padded to the maximum sequence length in a batch. By adopting a smaller batch size, the total padding decreases, limiting wasted computation on padding tokens during attention operations (Vaswani et al., 2017). Although batching sequences of similar lengths can also reduce padding, it disrupts random data shuffling, which is crucial for preventing overfitting and enhancing model generalizability (Yun et al.,

2021; Gürbüzbalaban et al., 2021; Bengio, 2012). Detailed statistics on the padding ratio for each task are given in Appendix C.

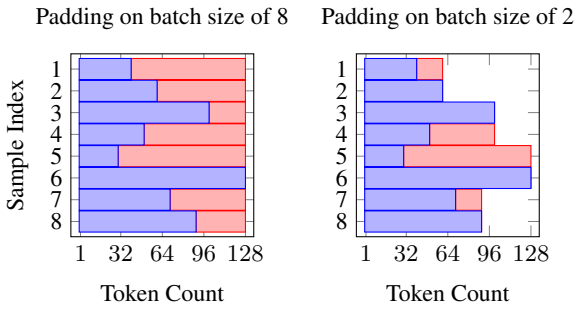


Figure 2: The standard batching approach pads shorter sequences to the maximum sequence length within the batch. A smaller batch size reduces the number of padding tokens, resulting in less wasted computation.

3.2 Inner-loop Parallelization

While outer-loop parallelization increases parallelism across multiple queries, one gradient estimation still requires two forward passes per query: one with positive perturbation and one with negative perturbation, which are executed sequentially in the RGE algorithm.

To further accelerate each gradient estimation, we propose inner-loop parallelization as shown in Algorithm 1 (line 7), which performs both forward passes simultaneously. As illustrated in Figure 1 (b), each input batch and LoRA-$\mathbf{B}$ matrix is duplicated once more. One copy of the LoRA-$\mathbf{B}$ matrix is perturbed with positive noise, and the other with negative noise. By computing the loss difference in parallel, we can estimate the gradient using a single combined forward pass. This approach further reduces the external memory bandwidth burden for loading model weights by reusing the model weights across two forward passes. As a result, P-RGE achieves even faster runtime per training step compared to the sequential two forward passes execution in RGE's original setting of $q = 1$.

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 LoRA-FA method.

3.3 On-Device Training Workflow

For the on-device implementation of our proposed methods, we use ExecuTorch (Meta-AI, 2024a) as the inference engine. As the successor to PyTorch Mobile (Meta-AI, 2024c), ExecuTorch enables model inference across various platforms with different backends (e.g., CPUs, NPUs, DSPs) using the same toolchains and SDKs provided by PyTorch.

Deploying a PyTorch model (`nn.Module`) on edge devices via ExecuTorch involves two main steps. First, we convert the model into an ExecuTorch program, a computation graph containing the model's parameters. This process produces a binary file with ExecuTorch instructions that the runtime interprets and executes. Second, we offload the binary file and runtime library to the target device. The runtime, written in C++ and OS-independent, includes an operator library tailored for the hardware backend and executes the ExecuTorch program.

However, the MeZO implementation is not natively supported in ExecuTorch, as it requires significant device-side modifications (e.g., resetting the random number generator, generating noise, extracting weights, applying gradients). For instance, line 8 in Algorithm 1 requires extracting model weights and applying noise, which is challenging because ExecuTorch does not provide an API for this. To simplify deployment, we leverage inner-loop parallelization and implement a dual-forwarding LoRA module in PyTorch. By defining the P-RGE training procedure within the LoRA module's `forward` function, we ensure full exportability to an ExecuTorch program. This allows us to generate and offload the program with minimal server-side modifications, enabling training without changes to the ExecuTorch runtime on edge devices.

In our dual-forwarding LoRA module, as shown

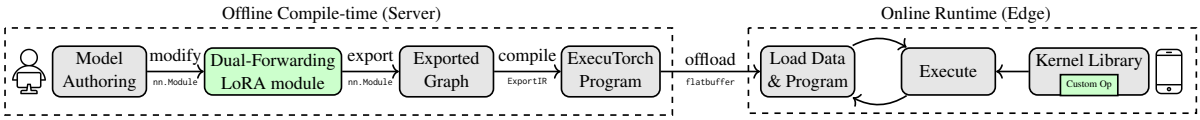


Figure 3: On-device training workflow via ExecuTorch with minimal modifications. The green box represents additional procedure in addition to standard steps for inference deployment on edge devices.

Algorithm 2 Dual-forwarding LoRA-FA Module

1: **Input:** $\mathbf{x} \in \mathbb{R}^{2\times \text{seq\_len}\times k}$, $\mathbf{A} \in \mathbb{R}^{k\times r}$, $\mathbf{B} \in \mathbb{R}^{2\times r\times k}$, $\mathbf{W}^{k\times k}$, 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

in Algorithm 2, we first compute the difference between the perturbed weights $\mathbf{B}[0]$ (positive perturbation) and $\mathbf{B}[1]$ (negative perturbation), which share the same random noise scaled by ϵ. Since resetting the random number generator with a seed is not an exportable operation in ExecuTorch (i.e., line 5 and 6 in Algorithm 1), we store all copies of matrix $\mathbf{B}$ in memory rather than maintaining a single master copy. This allows us to recover the random noise without regenerating it. Lines 5 and 6 in Algorithm 2 restore the original value of $\mathbf{B}$, update the weights with gradients, and apply new random noise. The output is computed by combining the original linear transformation $\mathbf{xW}$ with a batched matrix multiplication between matrices $\mathbf{xA}$ and $\mathbf{B}$. This approach extends to larger batch sizes and integrates with outer-loop parallelization.

Figure 3 illustrates the workflow for enabling on-device fine-tuning using dual-forwarding LoRA module in ExecuTorch. Starting with a pre-trained PyTorch model, we integrate the dual-forwarding LoRA module as in conventional first-order LoRA training and redirect the scalar projected gradient g to each module. Following the standard ExecuTorch process, we export, compile, and offload the model to the edge device. On the device, the ExecuTorch runtime seamlessly executes the binary file, handling data loading and running the inference plan, which includes the dual-forwarding LoRA module to update the model without explicitly recognizing it as a training task. For random noise generation, a custom operator can be integrated into the runtime library using the ExecuTorch API (Meta-AI, 2024b).

4 Experiments

We conduct comprehensive experiments on the TinyLlama-1.1B (Zhang et al., 2024a) and Llama2-7B (Touvron and 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, P-RGE, 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 $q = 16, E = 1$. By using the same number of training steps (i.e., 20,000) for both P-RGE and MeZO, we ensure that P-RGE does not exceed the computational budget of the MeZO baseline for end-to-end training. For reference, we also report zero-shot performance without additional fine-tuning. Additional experimental details, including dataset descriptions, training procedures, and hyperparameters, are provided in Appendix D.

For the smaller-scale TinyLlama-1.1B model, we evaluate its performance on the GLUE dataset (Wang et al., 2019). The results in Table 2 show that increasing the number of queries while decreasing the batch size outperforms the baseline MeZO by up to 8.8% accuracy. For the larger Llama2-7B model, we evaluate its performance on SST-2 (Wang et al., 2019), WinoGrande (Sakaguchi et al., 2021), and the SuperGLUE (Wang et al., 2020) dataset using the same experimental setup. As shown in Table 2, P-RGE consistently improves performance over the baseline that updates the full parameter space by up to 9.87%. While P-RGE introduces an additional hyperparameter q

TinyLlama-1.1B	Methods \ Tasks	SST-2	RTE	MRPC	QQP	QNLI	WNLI
Zero-shot		55.3	52.3	68.3	32.8	52.7	43.6
FO-SGD	Full	93.7	80.5	80.4	83.2	83.3	56.3
	LoRA-FA	92.7	78.7	80.6	83.4	83.7	59.2
ZO-SGD	MeZO (Full)	**91.2**	67.9	70.6	72.0	67.9	57.8
	MeZO (LoRA-FA)	87.5	67.9	71.6	74.6	67.2	**60.6**
	P-RGE ($q = 4$)	89.1	**70.8**	75.5	77.9	**76.0**	**60.6**
	P-RGE ($q = 16$)	90.8	68.6	**76.7**	**79.1**	75.0	57.8

Llama2-7B	Methods \ Tasks	SST-2	RTE	BoolQ	WSC	WiC	MultiRC	COPA	WinoGrande
Zero-shot		58.0	59.2	71.9	51.9	50.0	54.6	79.0	62.7
FO-SGD	Full	96.2	86.6	86.4	69.2	73.8	83.7	87.0	66.9
	LoRA-FA	95.6	81.2	85.5	62.5	63.5	80.4	86.0	66.1
ZO-SGD	MeZO (Full)	93.7	72.6	81.1	64.4	54.6	69.9	81.0	64.4
	MeZO (LoRA-FA)	94.4	72.9	80.7	64.4	60.0	73.1	86.0	64.5
	P-RGE ($q = 4$)	**94.9**	76.2	**83.0**	63.5	**64.4**	**74.5**	85.0	65.4
	P-RGE ($q = 16$)	94.2	**77.6**	82.5	**65.4**	62.9	74.9	**87.0**	**65.7**

Table 2: Performance of fine-tuning TinyLlama-1.1B and Llama2-7B on different tasks with different optimizers. P-RGE 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
P-RGE	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
P-RGE	12.61	12.82	13.06	12.64	13.07	13.55	12.70	13.55	14.53

Table 3: Peak memory usage of TinyLlama-1.1B and Llama2-7B for different sequence length and batch size configurations.

for improved accuracy, setting q to either 4 or 16 is recommended in practice to minimize the need for extensive hyperparameter searching.

4.2 System Performance

We conduct measurements on a single NVIDIA A100 GPU to evaluate the server-side system performance of P-RGE compared to its baselines. The ZO-SGD optimizer, including both MeZO and P-RGE, 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 P-RGE 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 3 shows the memory usage of FO-SGD (LoRA-FA), MeZO (LoRA-FA), and P-RGE with both outer-loop and inner-loop parallelization. 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, P-RGE 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 P-RGE, 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 4 shows the end-to-end wall-clock time for fine-tuning TinyLlama-1.1B and Llama2-7B using MeZO and P-RGE for 20,000 steps across various tasks. By applying PEFT methods, both MeZO and P-RGE reduce training time by minimizing sequential processing of model parameters, a benefit that becomes more pronounced with larger models such as Llama2-7B. P-RGE 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).

Additional system profiling ablation studies, including runtime breakdown under different fixed

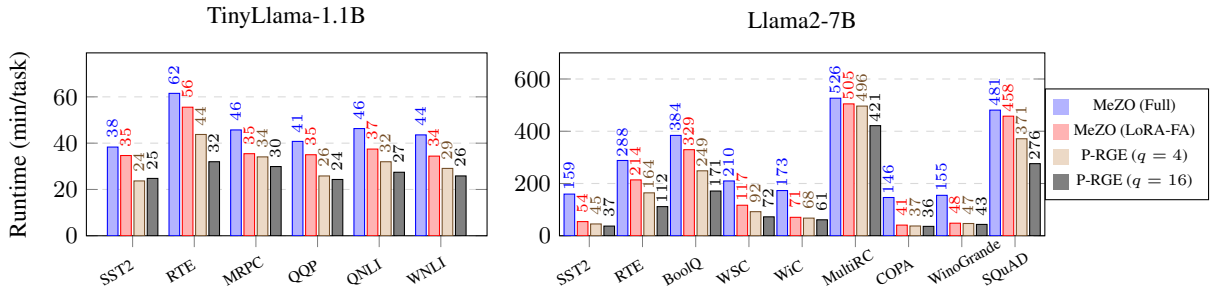


Figure 4: End-to-end wall-clock time of fine-tuning TinyLlama-1.1B and Llama2-7B for various configurations across tasks.

sequence length and batch size configurations, as well as under different quantization schemes, are available in Appendix F.

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 are provided in Appendix G. After verification, we measure and report the runtime per step of P-RGE 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 4 shows the speedup achieved through P-RGE with inner-loop parallelization with NF4 weight-only quantization, showing up to 1.83× performance improvement.

On the smartphone platform, which operates on Android OS without PyTorch support, we use the ExecuTorch workflow to perform ZO fine-tuning, integrating the dual-forwarding LoRA module as described in Section 3.3. Since we do not modify the runtime code on the edge device, vanilla MeZO baseline experiments are omitted. 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. ExecuTorch shows lower runtime efficiency for multi-batch inference compared to CUDA platforms, as it is primarily optimized for single-prompt processing, typical in chat-based LLMs. As shown in Table 5, with an effective batch size of 16, the NPU backend takes 15.73 seconds for one step with a sequence length of 128, whereas Jetson completes it in 8.60 seconds.

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
P-RGE	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
P-RGE	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 4: Runtime (sec/step) and speedup ratio of inner-loop parallelization on Jetson GPU backend for TinyLlama-1.1B and Llama2-7B with NF4 quantization. The results demonstrate a consistent performance boost across different batch sizes and sequence lengths.

Sequence length	64				128			
Batch size	2	4	8	16	2	4	8	16
Runtime (sec/step)	1.04	2.34	4.70	10.43	2.49	4.83	10.36	15.73
Memory (GB)	3.36	3.53	3.75	3.88	3.43	3.68	3.91	4.46

Table 5: Runtime and memory usage of dual-forwarding implementation on Android NPU backend for TinyLlama-1.1B.

5 Conclusion

This work introduces parallelized randomized gradient estimation (P-RGE) to address the computational and memory challenges of fine-tuning LLMs in resource-constrained edge environments. P-RGE leverages outer-loop and inner-loop parallelization for efficient multi-query gradient estimation, improving model accuracy without extra computational overhead. Experiments show P-RGE significantly enhances training speed and reduces memory usage on both server and edge platforms, enabling real-time, on-device fine-tuning. By integrating P-RGE with inference engines like ExecuTorch, we validate its versatility across diverse hardware such as Android NPU and Jetson GPU.

6 Limitations

While P-RGE enables efficient on-device LLM fine-tuning, it has several limitations. First, P-RGE 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, Android's NPU backend lacks native support for large matrix multiplications, limiting batch processing efficiency. Future work will explore alternative backends, such as Vulkan for GPU acceleration. Third, P-RGE assumes static computational settings, whereas edge environments often have dynamic resource constraints. Adapting query count, batch size, or precision in response to runtime conditions is a promising direction.

References

Rohan Anil and et al. 2024. Gemini: A family of highly capable multimodal models. *Preprint*, arXiv:2312.11805.

Stephen H. Bach and et al. 2022. Promptsource: An integrated development environment and repository for natural language prompts. *Preprint*, arXiv:2202.01279.

Yoshua Bengio. 2012. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade: Second edition*. Springer.

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 and et al. 2022. Palm: Scaling language modeling with pathways. *Preprint*, arXiv:2204.02311.

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*.

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*.

Mert Gürbüzbalaban, Asu Ozdaglar, and Pablo A Parrilo. 2021. Why random reshuffling beats stochastic gradient descent. *Mathematical Programming*.

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*.

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. Pytorch mobile.

Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. 2018. Mixed precision training. In *International Conference on Learning Representations*.

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

OpenAI, Josh Achiam, 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.

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 and et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *Preprint*, arXiv:2307.09288.

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

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.

Chulhee Yun, Suvrit Sra, and Ali Jadbabaie. 2021. Open problem: Can single-shuffle sgd be better than reshuffling sgd and gd? In *Proceedings of Thirty Fourth Conference on Learning Theory*.

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 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*.

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 6 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 6: 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 and 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 7 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 7: ZO accuracy of OPT-1.3B on SST2 dataset using different PEFT methods.

C Padding Statistics

Figure 5 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.

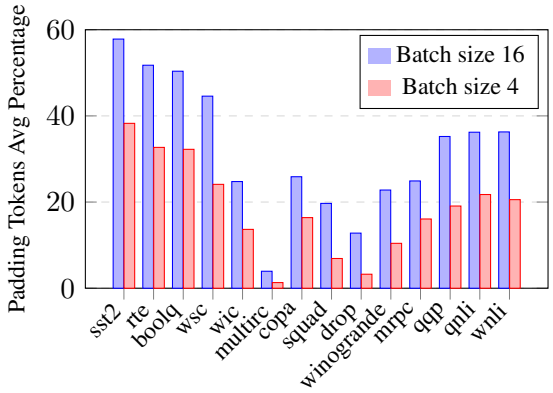


Figure 5: Average percentage of padding tokens for different tasks and batch sizes.

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 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), multiple-choice (COPA), and question-and-answering (SQuAD). We include one additional multiple-choice task from WinoGrande (Sakaguchi et al., 2021) dataset. 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 and et al., 2022). Table 8 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, COPA, and SQuAD, we applied the template from MeZO (Malladi et al., 2023). We created templates for MRPC, QQP, QNLI, and WNLI by following the suggestions from PromptSource (Bach and 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 proposed 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 9. For LoRA hyperparameters, we choose the LoRA rank to be 16 and LoRA alpha to be 32. For P-RGE, 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>
SQuAD	QA	Title: <title> Context: <context> Question: <question> Answer:

Table 8: The prompt template of the datasets used in the experiments.

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

Table 9: Hyperparameters used for TinyLlama-1.1B and Llama2-7B experiments. Note that MeZO (LoRA-FA) is a special case of P-RGE with $q = 1$.

E Additional FO Experiments

We also provide additional experimental results on FO-Adam in Tables 10 and 11. 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 10: Performance of fine-tuning TinyLlama-1.1B on different tasks with FO-Adam optimizers.

Tasks	**SST-2**	**RTE**	**BoolQ**	**WSC**	**WiC**	**MultiRC**	**COPA**	**WinoGrande**	**SQuAD**
Full	92.5	78.7	80.6	63.4	67.2	71.7	81.0	68.2	79.2
LoRA-FA	96.0	88.1	85.7	79.8	75.1	84.2	87.0	71.8	77.2

Table 11: Performance of fine-tuning Llama2-7B on different tasks with FO-Adam optimizers.

F Ablation Studies on System Performance of P-RGE

F.1 Efficiency of outer-loop parallelization

We measure the runtime and memory usage of P-RGE, implemented using outer-loop parallelization only for the Llama2-7B model across different effective batch size and fixed sequence lengths configurations. As shown in Table 12, 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 12: System performance of outer-loop parallelization for Llama2-7B under the same batch size of 16.

F.2 Efficiency of inner-loop parallelization

We measure the runtime and memory usage of P-RGE, implemented using inner-loop parallelization only for the Llama2-7B model across fixed different sequence length and batch size configurations. As shown in Table 13, 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
P-RGE ($q = 1$, inner)	0.04	0.10	0.18	0.04	0.18	0.34	0.06	0.34	0.67

Table 13: 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 6, inner-loop parallelization achieves the greatest speedup in conjunction with NF4 quantization, reaching up to a $1.97\times$ 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.

F.3 End-to-end training efficiency

Tables 15 - 18 provide additional details on per-task runtime and memory usage to complement the experimental results in Table 2. 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 P-RGE, 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, P-RGE achieves speedups of up to $4.3\times$ over MeZO (Full) and up to $1.9\times$ over MeZO (LoRA-FA).

Regarding memory usage, enabling both inner

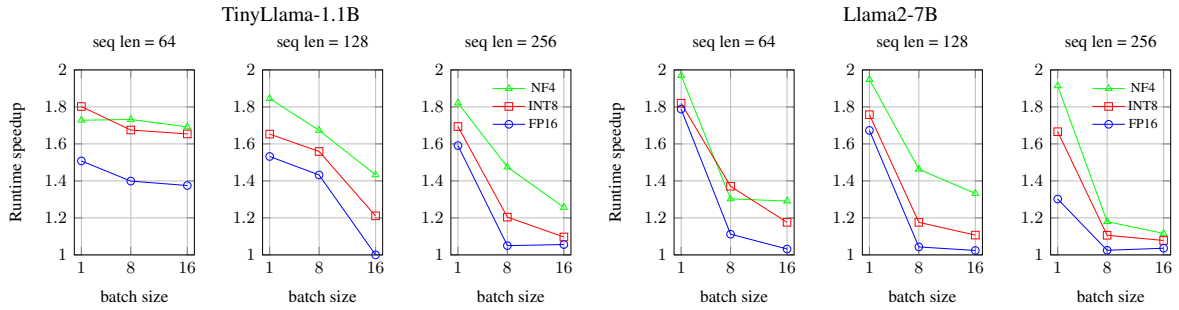


Figure 6: Runtime speedup per training step of TinyLlama-1.1B and Llama2-7B for different quantization methods, sequence lengths, and batch sizes.

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 SQuAD and 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.

G Edge Devices Specifications

Table 14 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 14: Edge devices used in the experiments.

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
P-RGE ($q = 4$)						
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
P-RGE ($q = 16$)						
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 15: 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	SQuAD
MeZO (Full) ($q = 1$)	159.44	288.10	384.07	209.72	173.01	526.49	146.40	154.74	480.90
MeZO (LoRA-FA) ($q = 1$)									
standard	54.20	213.81	329.46	116.79	70.55	504.74	40.77	48.07	457.69
inner	55.22	210.30	322.64	118.03	72.75	505.54	36.57	48.62	440.63
P-RGE ($q = 4$)									
outer only	49.11	165.53	251.63	91.87	66.55	505.70	44.65	49.01	376.34
inner + outer	45.17	164.21	248.55	92.17	67.52	496.32	37.38	46.89	371.29
P-RGE ($q = 16$)									
outer only	43.91	111.80	171.84	71.14	60.31	438.24	41.96	46.41	281.15
inner + outer	36.99	111.54	171.14	72.40	61.10	421.41	35.91	43.41	275.98

Table 16: 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
P-RGE ($q = 4$)						
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
P-RGE ($q = 16$)						
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 17: 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	SQuAD
MeZO (Full)	13.64	16.23	18.39	14.51	13.82	18.39	13.60	13.60	18.39
MeZO (LoRA-FA) ($q = 1$)									
standard	13.41	16.00	18.16	14.27	13.58	18.16	12.98	13.15	18.16
inner	14.23	19.41	23.73	15.96	14.57	23.73	13.37	13.71	23.73
P-RGE ($q = 4$)									
outer only	13.53	16.12	18.28	14.40	13.71	18.28	13.10	13.27	18.28
inner + outer	14.47	19.65	23.97	16.20	14.82	23.97	13.61	13.95	23.97
P-RGE ($q = 16$)									
outer only	14.03	16.10	18.77	14.92	14.20	18.77	13.59	13.77	18.77
inner + outer	15.45	19.59	24.95	17.17	15.79	24.95	14.58	14.93	24.95

Table 18: Peak memory usage (GB) of fine-tuning Llama2-7B across different tasks using different ZO methods.