Poor Man's Training on MCUs: A Memory-Efficient Quantized Back-Propagation-Free Approach

YEQUAN ZHAO, Electrical and Computer Engineering, University of California Santa Barbara, Santa Barbara, United States
HAI LI, Technology Research - Exploratory Integrated Circuits, Intel Corporation, Hillsboro, United States
IAN YOUNG, Technology Research - Exploratory Integrated Circuits, Intel Corporation, Hillsboro, United States
ZHENG ZHANG, Department of Electrical and Computer Engineering, University of California Santa Barbara, Santa Barbara, United States

Back propagation (BP) is the default solution for gradient computation in neural network training. However, implementing BP-based training on various edge devices such as FPGA, microcontrollers (MCUs), and analog computing platforms faces multiple major challenges, such as the lack of hardware resources, long time-to-market, and dramatic errors in a low-precision setting. This article presents a simple BP-free training scheme on an MCU, which makes edge training hardware design as easy as inference hardware design. We adopt a quantized zeroth-order method to estimate the gradients of quantized model parameters, which can overcome the error of a straight-through estimator in a low-precision BP scheme. We further employ a few dimension reduction methods (e.g., node perturbation, sparse training) to improve the convergence of zeroth-order training. Experiment results show that our BP-free training achieves comparable performance as BP-based training on adapting a pre-trained image classifier to various corrupted data on resource-constrained edge devices (e.g., an MCU with 1024-KB SRAM for dense full-model training, or an MCU with 256-KB SRAM for sparse training). This method is most suitable for application scenarios where memory cost and time-to-market are the major concerns, but longer latency can be tolerated.

CCS Concepts: • **Computing methodologies** → **Artificial intelligence**; • **Computer systems organization** → **Embedded systems;**

Additional Key Words and Phrases: On-device training, tiny machine learning, back-propagation-free training

ACM Reference Format:
Yequan Zhao, Hai Li, Ian Young, and Zheng Zhang. 2025. Poor Man's Training on MCUs: A Memory-Efficient Quantized Back-Propagation-Free Approach. *ACM Trans. Des. Autom. Electron. Syst.* 30, 5, Article 74 (August 2025), 33 pages. https://doi.org/10.1145/3745772

This work is supported by funding of Intel Strategic Research Sector (SRS) - Emerging Technology.
Authors' Contact Information: Yequan Zhao, Electrical and Computer Engineering, University of California Santa Barbara, Santa Barbara, California, United States; e-mail: yequan_zhao@ucsb.edu; Hai Li, Technology Research - Exploratory Integrated Circuits, Intel Corporation, Hillsboro, Oregon, United States; e-mail: hai.li@intel.com; Ian Young, Technology Research - Exploratory Integrated Circuits, Intel Corporation, Hillsboro, Oregon, United States; e-mail: ian.young@intel.com; Zheng Zhang, Department of Electrical and Computer Engineering, University of California Santa Barbara, Santa Barbara, California, United States; e-mail: zhengzhang@ece.ucsb.edu.

ACM 1084-4309/2025/08-ART74
https://doi.org/10.1145/3745772

1 Introduction

On-device training, that is training **deep neural networks** (**DNNs**) on edge devices, enables a DNN model pre-trained on *cloud* to improve itself on newly observed data and adapt to cross-domain or out-of-domain distribution shifts after edge deployment. It also allows the model to adapt to user personalization locally, which protects user privacy over sensitive data (e.g., healthcare and financial data). As physic-informed machine learning has been increasingly used for safety-critical decision-making in autonomous systems, there has been also growing interest in on-device fine-tuning or end-to-end training. In federated learning, a machine learning model also needs to be trained periodically on each local edge node, then updated on a global centralized server.

Backward propagation (**BP**) [51] is used in almost all neural network training frameworks for gradient computation. BP is actually a reverse-mode **automatic differentiation** (**AD**) [3, 60] approach implemented based on the information of a computational graph. While a forward-mode AD is suitable for computing the gradient of a single-input multiple-out function, BP is more suitable for a multiple-input (i.e., many network parameters) and single-output (i.e., training loss) function. With sophisticated AD packages, operating systems, and compilers, BP can be called with just one command (e.g., `loss.backward()` in PyTorch) on a CPU- or GPU-based desktop or cloud computing platform. This has greatly simplified the development and deployment of modern neural network models.

However, training a neural network on resource-constrained edge hardware [e.g., a **microcontroller unit** (**MCU**), FPGA, or photonic platform] is completely different from the training task on a desktop or cloud platform, due to the limited hardware resources and software support. Specifically, implementing a standard BP-based training framework on edge devices is often prevented by three major challenges:

- **Memory Challenge.** Edge devices like MCU have a very limited run-time memory (e.g., STM32F746 with only 256-KB user SRAM, or STM32H7B3 with 1024-KB user SRAM). This budget is often below the memory requirement of storing all network parameters, making full-model BP-based training impossible for most realistic cases. By choosing tailored network models (e.g., MCUNet [54]), using real-quantized graphs and a co-designed lightweight back-end (e.g., the TinyEngine [54, 55]), one may perform edge inference with a low memory cost (e.g., 96 KB for the MCUNet-in1 model [54]). However, the memory cost of a full-model BP-based training (e.g., 7.4 MB for MCUNet-in1) is far beyond the memory capacity. Existing training methods on MCU update only a small subset of model parameters (e.g., only the last layer [62, 75], bias vectors [9]) to reduce the memory cost, yet leads to significant (e.g., >10%) accuracy drop. Sparse update [49, 54] could narrow this gap, yet requires computation-intensive searches and compilation-level optimization on *cloud*.
- **Precision Challenge.** Low-precision quantized computation is often utilized on digital edge hardware (e.g., MCU and FPGA) to reduce latency, memory cost, and energy consumption. However, low-precision operations pose great challenges for BP-based training. BP was originally designed for the gradient computation of smooth functions. Thus, error-prone approximation techniques such as straight-through estimators [5] are required to handle non-differentiable functions in quantized neural network training. The errors introduced by these approximation techniques increase as hardware precision reduces. They also propagate and accumulate through different layers, leading to dramatic accuracy drop, unstable training behaviors, or even divergence [12, 98].
- **Time-to-Market Challenge.** While BP can be done on CPU or GPU with just one line of code (e.g., `loss.backward()` in PyTorch), implementing it on edge devices can be very

challenging. Due to the lack of AD packages [3] and sophisticated operating systems on edge platforms, designers often have to implement the math and hardware of gradient computation manually. On some platforms (e.g., integrated photonics), novel devices must be invented and fabricated to perform BP [70]. This error-prone process needs numerous debugs and design tradeoffs. As a result, designing edge training hardware is more time-consuming than designing inference hardware. For instance, our own experience shows that an experienced FPGA designer can design a high-quality inference accelerator within one week, yet it takes over one year to implement an error-free training accelerator on FPGA. This long time to market is often unacceptable in the industry due to the fast evolution of AI models.

Article Contributions. The above challenges motivate us to ask the following question:

Can we make the edge training hardware design as easy and memory-efficient as inference hardware design?

In this article, we show that the answer is affirmative, with the assumption that memory budget and time to market are given higher priority over runtime latency. Our key idea is to completely bypass the complicated BP implementation by proposing a **quantized zeroth-order (ZO) method** to train a real-quantized neural network model on MCU. This training method only uses quantized forward evaluations to estimate gradients. As a result, we can use a similar memory cost of inference to achieve full-model training under the tiny memory budget of an MCU. This quantized ZO training framework can be used as a plug-and-play tool added to quantized inference hardware, therefore the design complexity and time to market can be dramatically reduced. Our specific contributions are briefly summarized below:

(1) **ZO Quantized Training for Edge Devices.** We propose a BP-free training framework via quantized ZO optimization to enable full-model and real-quantized training on MCUs under extremely low memory budget (e.g., 256-KB SRAM for sparse training or 1024-KB SRAM for dense training). This framework enjoys low memory cost and easy implementation. Furthermore, it shows better accuracy than quantized BP-based training in low-precision (e.g., INT8) settings since no error-prone straight-through estimator is needed.
(2) **Convergence Improvement.** ZO training suffers from slow convergence rates as the number of training variables increases. Previous assumption of low intrinsic dimensionality [58] or coordinate-wise gradient estimation [11] does not work in on-device training. To improve the training convergence, we propose a learning-rate scaling method to stabilize each training step. We also employ a few dimension-reduction methods to improve the training convergence: (i) a generic layer-wise gradient estimation strategy that combines **weight perturbation** (**WP**) and **node perturbation** (**NP**) for ZO gradient estimation, (ii) a sparse training method with task-adaptive block selection to reduce the number of the trainable parameters.
(3) **MCU Implementation.** We implement the proposed BP-free training framework on an MCU (full-model training on an STM32H7B3 with 1024-KB user SRAM, and sparse training on an STM32F746 with 256-KB user SRAM). A quantized inference engine is easily converted to a training engine, with only an additional control unit, a temporary gradient buffer, and a pseudo-random number generator. To our best knowledge, this is the first framework to enable full-model training under such a tiny memory budget (STM32H7B3 with 1024-KB user SRAM).
(4) **Experimental Validation.** We conduct extensive experiments on adapting a pre-trained image classification model to unseen image corruptions and fine-grained vision classification datasets. On adapting noise corruptions, our BP-free training outperforms current quantized

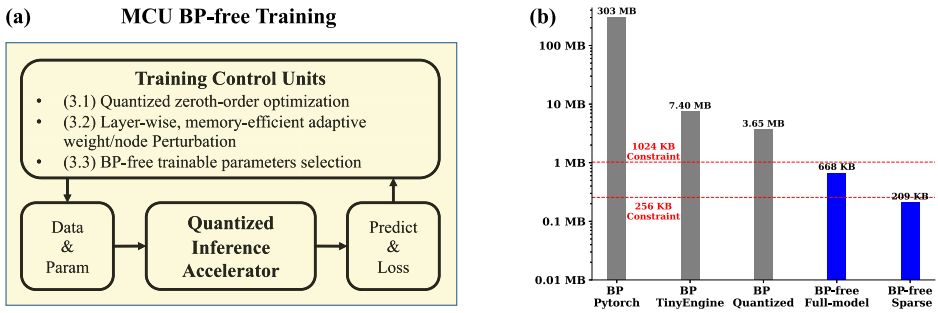

Fig. 1. (a): Overview of BP-free training framework. A quantized inference engine is easily converted to a training engine by adding control unit and repeatedly calling the inference accelerator. (b): Training memory comparison of different training methods. The numbers are measured with MCUNet-in1 [54], batch size 1, and resolution 128 × 128.

BP-based training with an average 3.8% test accuracy improvement. Our method can also match the performance of BP-based training on fine-grained vision classification datasets.

Our Key idea is summarized in Figure 1(a). As demonstrated in Figure 1(b), our method is the only solution to enable full-model training on commodity-level MCU (e.g., STM32H7B3 with 1024-KB user SRAM) without auxiliary memory. This memory cost is the minimum to enable full-model training (478 KB model parameters plus 190 KB peak inference memory), 5.46× more memory-efficient than the memory cost of quantized BP, and > 400× more memory efficient than BP-based training in PyTorch which includes back-end memory overhead. BP-free sparse training further reduces the memory cost to fit a smaller budget (e.g., 256-KB SRAM).

2 Preliminaries

2.1 Real-Quantized Neural Network Model

Given a full-precision linear layer $\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$, the low-precision (e.g., INT8) quantized counterpart is

$$\bar{\mathbf{z}} = \text{clip}(\lceil (s_{\mathbf{W}} s_{\mathbf{x}} \bar{\mathbf{W}} \bar{\mathbf{x}} + \bar{\mathbf{b}}) / s_{\mathbf{z}} \rfloor, -Q_N, Q_P) \tag{1}$$

Here $\bar{\cdot}$ denotes the quantized variables, s is a floating-point scaling factor, $\text{clip}(v, r_1, r_2)$ returns r_1 when $v \leq r_1$ (or r_2 when $v \geq r_2$), and $\lceil v \rfloor$ rounds v to the nearest integer. Given a b-bit integer format, $Q_N = 2^{b-1}$ and $Q_P = 2^{b-1} - 1$ for signed quantization; $Q_N = 0$ and $Q_P = 2^b - 1$ for unsigned quantization. We call it a *real-quantized* graph [55] since all variables are in low-precision formats, and all matrix multiplications are computed with fixed-point arithmetic (the scaling factor s could further be quantized to achieve only fixed-point multiplication [44]). On resource-constrained edge devices, *real-quantized* graphs are usually leveraged to achieve memory and computation efficiency.

In this article, we focus on the training of a *real-quantized* neural network on MCU. We remark that training on a real-quantized graph fundamentally differs from quantization-aware training [41] and fully-quantized training [12]: the latter two maintain a full-precision copy of model parameters and only quantize model parameters and activations to leverage fixed-point computation. The comparison can be found in Table 1.

2.2 Extra Memory and Computation Cost of Back-Propagation

In this subsection, we analyze the extra memory and computation requirements of BP.

Table 1. Comparison of Different Quantized Training Paradigms

	Model	Forward	Backward
Quantization-aware Training (QAT) [41]	FP	INT	FP
Fully-quantized Training (FQT) [12]	FP	INT	INT
Real-quantized Training (RQT) [55]	INT	INT	INT

Consider a generic L-layer network: $f(\mathbf{x};\boldsymbol{\theta}) = f^{(L)} \circ f^{(L-1)} \circ \cdots \circ f^{(0)}(\mathbf{x};\boldsymbol{\theta}^{(0)})$, where each layer computes:

$$\mathbf{a}^{(i+1)} = h^{(i)}(\mathbf{z}^{(i)}), \quad \mathbf{z}^{(i)} = \mathbf{W}^{(i)}\mathbf{a}^{(i)} + \mathbf{b}^{(i)}. \tag{2}$$

To find optimal parameters $\{\boldsymbol{\theta}^{*(i)}\}_{i=1}^{L}$ that minimizes a loss $\mathcal{L}$, BP recursively computes gradients:

$$\nabla_{\mathbf{z}^{(i)}}\mathcal{L} = h'(\mathbf{z}^{(i)})\nabla_{\mathbf{a}^{(i+1)}}\mathcal{L},\ \nabla_{\mathbf{a}^{(i)}}\mathcal{L} = \mathbf{W}^{(i)\top}\nabla_{\mathbf{z}^{(i)}}\mathcal{L}, \quad \nabla_{\mathbf{W}^{(i)}}\mathcal{L} = \nabla_{\mathbf{z}^{(i)}}\mathcal{L}\cdot\mathbf{a}^{(i)\top}, \quad \nabla_{\mathbf{b}^{(i)}}\mathcal{L} = \nabla_{\mathbf{z}^{(i)}}\mathcal{L}. \tag{3}$$

Extra Memory. Consider run-time memory (e.g., SRAM), inference engines only needs to save $\mathbf{a}^{(i)}$ for one layer. BP-based training engine must additionally store (1) ALL intermediate activation values $\{\mathbf{a}^{(i)} \in \mathbb{R}^{B\times d_{a_i}}\}_{i=1}^{L}$ and $\{\mathbf{z}^{(i)} \in \mathbb{R}^{B\times d_{a_i}}\}_{i=1}^{L}$, where B is batch size and d_{a_i} is the number of neurons in layer i, (2) parameter gradients $\nabla_{\boldsymbol{\theta}^{(i)}}\mathcal{L}$, and (3) optimizer states. While techniques like activation recomputation or mask-based compression can help in specific cases (e.g., ReLU, **convolutional neural network** (**CNNs**)), many modern operations (e.g., attention, GeLU) still require full storage, limiting BP's applicability on memory-constrained devices.

Extra Computation Graph. Implementing BP on edge devices (e.g., micro-controllers, FPGAs, ASICs) requires manually optimizing the BP computation graphs (Equation (3)), as these devices often lack operating systems and AD [6] support. This inevitably increases the design and manufacturing costs and the time to market.

2.3 Zeroth-Order Optimization

As shown in Figure 2, ZO optimization [57] uses only function queries, instead of exact gradient information, to solve optimization problems. In this work, we use the multi-point variant of **randomized gradient estimator** (**RGE**) [11, 56, 58] as a ZO gradient estimator.

Definition 2.1 (Randomized Gradient Estimator, RGE). For a finite-sum optimization problem $\min_{\boldsymbol{\theta}\in\mathbb{R}^d} F(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} f_i(\boldsymbol{\theta})$, RGE estimates the gradients of F with respect to variables $\boldsymbol{\theta} \in \mathbb{R}^d$ as

$$\hat{\nabla}_{\boldsymbol{\theta}} F(\boldsymbol{\theta}) = \frac{1}{Q}\sum_{i=1}^{Q} \frac{[F(\boldsymbol{\theta} + \mu\boldsymbol{\xi}_i) - F(\boldsymbol{\theta})]}{\mu}\boldsymbol{\xi}_i. \tag{4}$$

Here $\{\boldsymbol{\xi}_i\}_{i=1}^{Q}$ are Q *i.i.d.* random perturbation vectors drawn from a zero-mean and unit-variance distribution (*e.g.*, multivariate normal distribution $\mathcal{N}(\mathbf{0}, \boldsymbol{I})$ or multivariate uniform distribution $\mathcal{U}(\mathcal{S}(0,1)$ on a unit sphere centered at zero with a radius of one). $\mu > 0$ is the sampling radius, which is typically small. RGE is an unbiased estimation of $\nabla_{\boldsymbol{\theta}} F_{\boldsymbol{\xi}}(\boldsymbol{\theta})$, here $F_{\boldsymbol{\xi}}(\boldsymbol{\theta})$ denotes the random smoothed version of $F(\boldsymbol{\theta})$. However, RGE is a biased estimation to $\nabla_{\boldsymbol{\theta}} F(\boldsymbol{\theta})$. With $\mu \to 0$, $\hat{\nabla}_{\boldsymbol{\theta}} F(\boldsymbol{\theta})$ is asymptotically unbiased to $\nabla_{\boldsymbol{\theta}} F(\boldsymbol{\theta})$ [11, 29].

We utilize the corresponding **stochastic gradient descent** (**SGD**) [8] algorithm, ZO-SGD [30, 63] to update model parameters in the training process.

Definition 2.2 (ZO-SGD). The variables $\boldsymbol{\theta}$ are iteratively updated as

$$\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \eta\hat{\nabla}_{\boldsymbol{\theta}} F(\boldsymbol{\theta}) \tag{5}$$

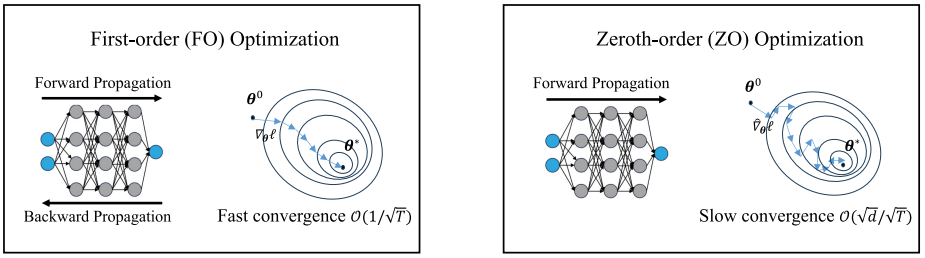

Fig. 2. Comparison between **first-order** (**FO**) optimization and ZO optimization. FO optimization converges faster as it utilizes exact gradient from BP to update model parameters. ZO optimization, on the other hand, uses only forward function queries to estimate the gradients. ZO method converges more slowly due to the large variance of gradient estimation, but it is much more memory-efficient, since no extra computation graph is needed.

Here the descent direction is computed using the ZO estimation rather than a BP method.

3 Poor Man's Training on MCU

This section presents a completely BP-free training framework on MCU with tiny memory budget. Our goal is two-fold: (1) to enable ultra memory-efficient on-device full-model training, (2) to greatly simplify the design complexity of training hardware, making it as easy as inference hardware design.

To achieve the above goals, we propose a quantized ZO optimization in Section 3.1 that employs only quantized inferences (forward evaluations) to optimize quantized model parameters. This allows us to reuse an inference hardware accelerator and convert it to a training engine with minimal changes. The ZO gradient estimation avoids the additional memory associated with storing activation values in BP. Here we employ the vanilla ZO-SGD [c.f. (5)] without momentum to avoid the memory overhead caused by optimizer states. However, training a real-quantized model with ZO gradient estimation faces slow or even no convergence. This is caused by the dimension-dependent variance of ZO gradient estimation as well as the quantization error. We stabilize the algorithm by properly scaling the learning rate (c.f. Section 3.1.2) of each layer to mitigate the distorted gradient norms. Further, we employ a few dimension reduction methods to reduce the ZO gradient variance, including a combination of WP and NP, as well as a BP-free sparse training method with task-adaptive trainable parameters selection. These techniques combined enable a stable and accelerated BP-free training framework on an MCU.

3.1 Training Real Quantized Models via Quantized Zeroth-Order Optimization

Given the resource constraints on an MCU, we consider a **real-quantized** setting, where we only have access to the low-precision (integer) quantized and scaled representation of model parameters $\bar{\theta} = \{\bar{\mathbf{W}}, \bar{\mathbf{b}}\}$ and input/output activation values $\bar{\mathbf{a}}$. Let $\mathcal{L}(\theta; \mathcal{X})$ be the empirical loss function, but we can only access via a sample-wise loss $\ell(\bar{\theta}; \mathbf{x})$ evaluated at a data sample $\mathbf{x} \subset \mathcal{X}$. We estimate the gradient of quantized parameters $\nabla_{\bar{\theta}_i} \mathcal{L}$ by a quantized ZO method and directly update the quantized model parameters.

3.1.1 Quantized ZO Gradient Estimation. Given a batch of data samples $\{\mathbf{x}_n\}_{n=1}^N \subset \mathcal{X}$, the quantized ZO randomized gradient estimation (quantized-RGE) is given as

$$\hat{\nabla}_{\bar{\theta}} \mathcal{L} = \frac{1}{NQ} \sum_{n=1}^{N} \sum_{q=1}^{Q} \frac{\left[\ell\left(\bar{\theta} + \mu \xi_{n,q}; \mathbf{x}_n\right) - \ell(\bar{\theta}; \mathbf{x}_n)\right]}{\mu} \xi_{n,q}, \mathbf{x}_n \subset \mathcal{X} \tag{6}$$

We jointly consider the double stochasticity of ZO-SGD, i.e., the stochasticity in sub-sampling training data and the stochasticity in random perturbation, by applying an independent set of perturbations $\{\{\xi_q\}_{q=1}^{Q}\}_n$ to each training data sample $\mathbf{x}_n$. We directly add perturbation to the quantized model parameters $\bar{\theta} \in \mathbb{R}^d$ with discrete integer values (e.g., $\{\bar{\theta} \in \mathbb{Z} | -128 \leq \bar{\theta} \leq 127\}$ for signed INT8 format). To ensure that the perturbed values are still in integer format, we sample the perturbation ξ_q from the Rademacher distribution of which the entries are integers +1 or -1 with equal probability. The perturbations sampled from a Rademacher distribution are zero-mean and unit-variance, ensuring that Equation (6) is an unbiased gradient estimator as $\mu \to 0$ [79].

However, in a quantized setting, we must restrict the smoothing parameter μ to be the smallest integer (i.e., $\mu = 1$). This leads to a biased gradient estimation. For SGD with a biased gradient estimator, the convergence rate depends on the **mean squared error** (**MSE**) of gradient estimation [22, 29]. The MSE of gradient estimation based on Equation (6) derived by [29] is given as

$$\mathbb{E}\left[\|\hat{\nabla}_\theta \mathcal{L} - \nabla_\theta \mathcal{L}\|_2^2\right] = \frac{d-1}{NQ}\|\nabla_\theta \mathcal{L}(\theta)\|_2^2 + \frac{d}{NQ}\operatorname{tr}\left(\operatorname{Var}_{\mathbf{x}}\left[\nabla_\theta \ell(\theta, \mathbf{x})\right]\right) + O(\mu^2 d^2) \tag{7}$$

On the right-hand side, the first term comes from the gradient estimation variance, the second term comes from the gradient variance from data sampling, and the last term is the remainder when μ does not go to 0. Reducing the MSE of the gradient estimation, especially the variance of gradient estimation that dominate MSE, provably improves the convergence speed [29].

3.1.2 Learning-Rate Scaling. Directly updating quantized model parameters with the gradient estimator $\hat{\nabla}_{\bar{\theta}} \mathcal{L}$ in Equation (6) and using a global learning rate η leads to slow or even no convergence. Under this setting, we show that the SGD-style update is distorted by the high variance of gradient estimation and the quantization process. To address this issue, we incorporate two scaling methods to adjust the learning rate.

— **Gradient-Norm Scaling.** The high variance of a ZO gradient estimation distorts the gradient update. The descent lemma derived by [58] indicates that the largest permissible learning rate of ZO-SGD should be $\mathbb{E}\left[\|\hat{\nabla}_\theta \mathcal{L}\|^2\right] / \mathbb{E}\left[\|\nabla_\theta \mathcal{L}\|^2\right]$ times smaller than that of SGD to guarantee loss decrease. According to Equation (7), the squared norm of the ZO stochastic gradient estimation is approximately $(NQ + d - 1)/NQ$ times larger than that of the FO stochastic gradient [58]. To avoid complicated hyperparameter tuning of the learning rate according to different N, Q, or d, we propose to fix a global learning rate and scale it by $NQ/(NQ + d - 1)$ at each step.

— **Quantization-Aware Scaling [55].** The quantization process also distorts the SGD update. Let θ denote the full-precision parameter, and $\bar{\theta} = \theta / s_\theta$ denote its quantized (e.g., INT8) and scaled representation. Here $s \ll 1$. $\bar{\theta}$ is $1/s_\theta$ times larger than θ in magnitude, while according to the chains rule, the gradient magnitude of quantized representation $\hat{\nabla}_{\bar{\theta}} \mathcal{L}$ is s_θ times smaller than that of $\hat{\nabla}_\theta \mathcal{L}$. To ensure the update of a quantized parameter representation follows the expected update in its original scale, we follow [55] to apply a quantization-aware scaling to the learning rate of each parameter. Specifically, we fix a global learning rate and divide the learning rate of each quantized parameter by the square of its scaling factor s_θ (c.f. Appendix B for details).

Consequently, at step t we update quantized model parameters as

$$\bar{\theta}^{t+1} \leftarrow \operatorname{clip}\left(\bar{\theta}^t - \left\lfloor \frac{NQ}{NQ + d - 1} \frac{1}{s_\theta^2} \cdot \eta \hat{\nabla}_{\bar{\theta}} \mathcal{L} \right\rceil, -Q_N, Q_P\right) \tag{8}$$

The update is rounded and clipped to maintain the updated parameters in its low-precision format.

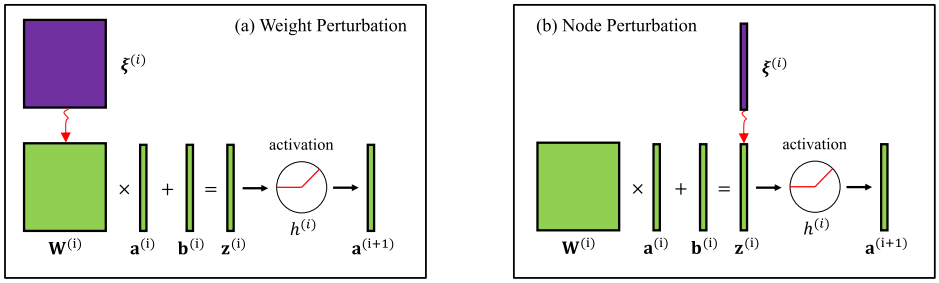

Fig. 3. (a) Data flow of WP. (b) Data flow of NP.

The convergence speed of ZO training depends on the gradient error [22, 29], which is dominated by the gradient estimation variance [i.e., the first term in Equation (7)]. Despite various variance reduction methods for ZO optimization, extra memory is needed to save either the covariance matrix [78], hessian diagonal [91], or history perturbations [15, 24, 42, 56], making them unsuitable for edge devices with tiny memory budget. Reference [29] proposed to find the optimal perturbation distribution that minimizes the MSE error, but it requires the shrinkage of the perturbation scale, making it infeasible for real-quantized graphs. As the ZO gradient variance depends on the dimension d of the optimization variables, we employ dimension-reduction methods to reduce the MSE error and improve the training convergence.

3.2 Memory-Efficient Adaptive Weight/Node Perturbation

We proposed to leverage both layer-wise gradient estimation and NP as dimension-reduction techniques to improve the overall training performance. Our final applied solution is:

(1) **Layer-wise gradient estimation**: perturb and estimate the gradients for only 1 layer at a time.
(2) **Adaptive weight perturbation and node perturbation**: use WP for layers with smaller weight dimension, and NP for layers with smaller node dimension.
(3) **Memory-efficient implementation** of both WP and NP.

In the following, we first introduce the background of NP and its challenges, then explain the details of each technique in our proposed solution.

3.2.1 Background: Dimension Reduction via Node Perturbation. Equation (6) perturbs the model parameters of every layer to obtain a ZO gradient. An alternative of obtaining the ZO gradient estimation for $\mathbf{W}_i$ and $\mathbf{b}_i$ is to firstly perturb the output nodes of linear transformation $\mathbf{z}^{(i)} = \mathbf{W}^{(i)}\mathbf{a}^{(i)} + \mathbf{b}^{(i)}$ and estimate the gradient of nodes $\hat{\nabla}_{\mathbf{z}^{(i)}}\mathcal{L}$,

$$\hat{\nabla}_{\mathbf{z}^{(i)}}\mathcal{L} = \frac{1}{NQ}\sum_{n=1}^{N}\sum_{q=1}^{Q}\frac{\ell_q(\bar{\boldsymbol{\theta}};\mathbf{x}_n) - \ell(\bar{\boldsymbol{\theta}};\mathbf{x}_n)}{\mu}\xi_{k,q,n}, \quad \mathbf{x}_n \subset \mathcal{X}. \tag{9}$$

Here $\ell_q(\bar{\boldsymbol{\theta}};\mathbf{x}_n)$ is the sample-wise loss associated with perturbing the linear transformation output $\mathbf{z}^{(i)}$ via $\bar{\mathbf{a}}_q^{(i+1)} = h^{(i)}(\bar{\mathbf{z}}^{(i)} + \mu\boldsymbol{\xi}_q^{(i)})$. According to Equation (3), the gradient of the weights and biases is then obtained as [19, 37, 76]:

$$\hat{\nabla}_{\bar{\mathbf{W}}^{(i)}}\mathcal{L} = \hat{\nabla}_{\bar{\mathbf{z}}^{(i)}}\mathcal{L}\cdot(\bar{\mathbf{a}}^{(i)})^T \quad \hat{\nabla}_{\bar{\mathbf{b}}^{(i)}}\mathcal{L} = \hat{\nabla}_{\bar{\mathbf{z}}^{(i)}}\mathcal{L} \tag{10}$$

We term these two gradient estimation schemes as *WP* and *NP*, respectively, which are illustrated in Figure 3. In cases where the activation dimension d_a is smaller than weight dimension d_w, NP benefits from a smaller gradient estimation variance of $\hat{\nabla}_{\bar{\mathbf{W}}_i}\mathcal{L}$.

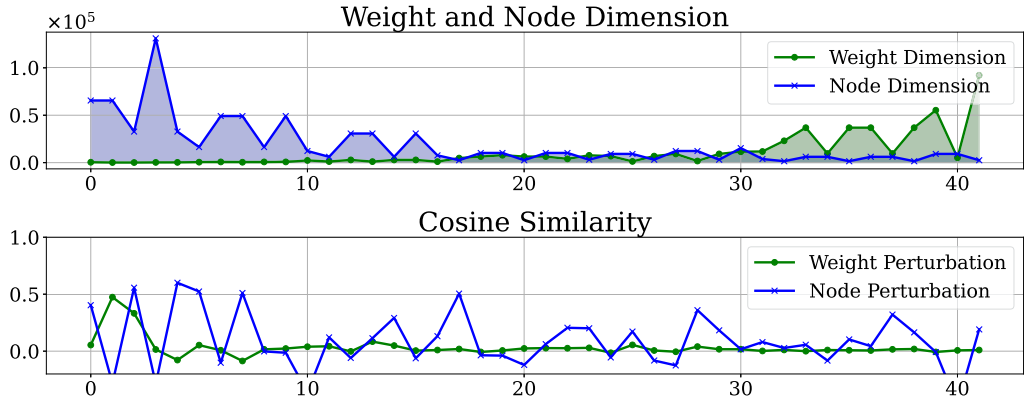

Fig. 4. Top: The dimension of weights/nodes at each layer. Bottom: The cosine similarity between ZO gradient estimation and the FO gradient computed by BP at each layer. NP is not always better than WP due to unbalanced weight/node dimension across layers.

However, the superiority of NP over WP could be hindered:

— **Unbalanced Weight/Node Dimension:** Convolution neural networks have smaller weight dimensions and larger node dimensions in starting layers while having larger weight dimensions and smaller node dimensions in ending layers. We take a preliminary investigation into the gradient estimation variance at each layer. Figure 4 shows the cosine similarity between the ZO gradient estimation and the FO gradient computed by BP of each layer. The larger cosine similarity indicates a better alignment with the true gradient, *i.e.*, smaller gradient estimation MSE. Applying WP or activation perturbation across the whole model does not guarantee a better gradient estimation for all layers.

— **Inter-layer Feature Correlation:** Figure 4 also shows a zig-zag pattern in NP. This is attributed to the simultaneous perturbation and update of all layers in vanilla NP. The perturbations added at each layer accumulate and get amplified to the following layers, which potentially introduces additional variance and affect the gradient estimation performance [19, 37].

— **Memory Inefficiency:** Vanilla implementation of NP has the same memory overhead as BP since the input activation values $\mathbf{a}_i$ of each layer need to be temporarily saved in the forward propagation. In contrast, by leveraging a pseudo-random number generator that could generate the same perturbation given the same random seed, WP needs storing only $2N + 1$ scalars [58].

3.2.2 Proposed Solution: Layer-Wise Memory-Efficient Adaptive Weight/Node Perturbation. To fully leverage the benefit of NP and improve the overall training performance, we propose the following techniques:

— **Layer-wise Gradient Estimation.** We propose to leverage a layer-wise gradient estimation strategy. Specifically, at each step, we only add perturbation to, and estimate the gradients of the parameters in one single layer. We theoretically and experimentally show that layer-wise gradient estimation contributes to a lower gradient variance for each layer than model-wise gradient estimation. Table 2 summarizes the theoretical results. For simplicity, we consider a neural network with L identical layers. The dimension of weight is d_w and the dimension of output activation is d_a. The mini-batch (number of data samples) is N. We fix the computation cost (evaluated by the number of forward passes) of different methods for

Table 2. Gradient Estimation Variance Comparison between WP and NP with Their Model-Wise and Layer-Wise Gradient Estimation Implementations

	Computation	Model-wise Variance	Layer-wise Variance
Weight Perturbation	NQ	$\frac{Ld_w-1}{NQ}S + \frac{L}{NQ}V$	$\frac{Ld_w-L}{NQ}S + \frac{L}{NQ}V$
Node Perturbation	NQ	$\frac{Ld_a-1}{NQ}S + \frac{L}{NQ}V$	$\frac{Ld_a-L}{NQ}S + \frac{L}{NQ}V$

S is the squared gradient norm, and V is the variance of stochastic gradient.

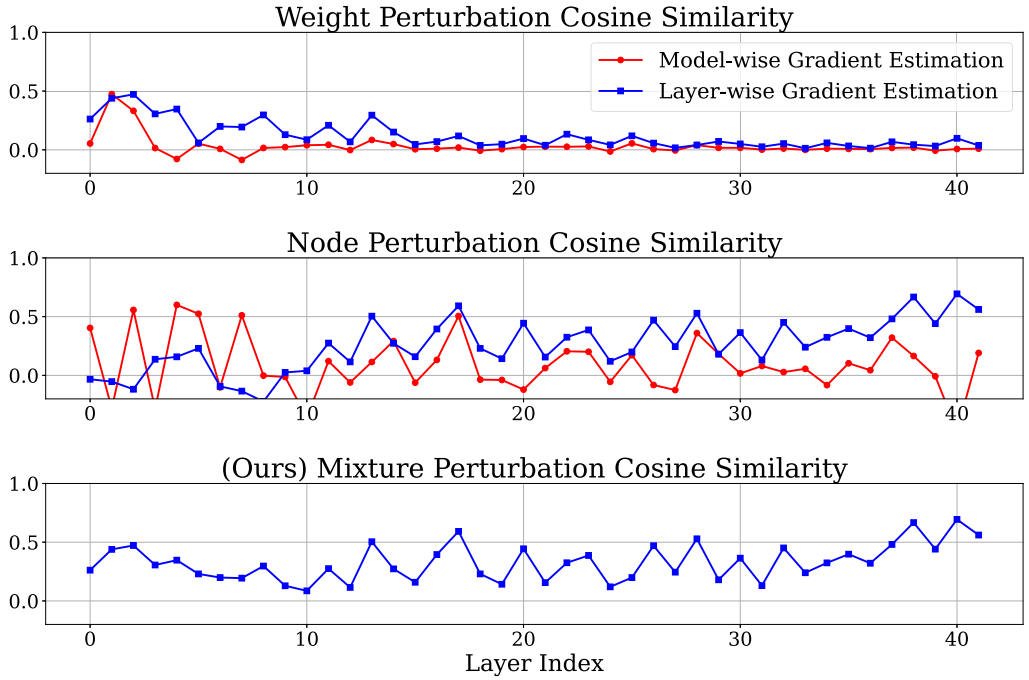

Fig. 5. Quality of gradient estimation (measured via Cosine similarity) via WP (top), NP (middle) and our mixture of WP and NP (bottom). Layer-wise perturbation outperforms model-wise perturbation; by applying layer-wise adaptive WP and NP, our solution enjoys the benefits of the two, and achieves the larger cosine similarity (better gradient estimation) across all layers.

a fair comparison. The number of forward evaluations is Q in model-wise perturbation and Q/L for each layer in layer-wise perturbation. Theoretically, the variance of layer-wise gradient estimation is just marginally smaller than that of model-wise gradient estimation, but this is true only if we neglect the possible cross-layer correlation between weights/nodes. Empirically, Figure 5 shows that layer-wise gradient estimation outperforms model-wise gradient estimation for both WP and NP with a non-negligible gap. This is attributed to the de-correlation between layers in a layer-wise gradient estimation method [19].

— **Adaptive Weight Perturbation and Node Perturbation.** With a disentangled layer-wise gradient estimation scheme, we can minimize the gradient variance of each layer by applying WP for layer i if $d_w^{(i)} < d_a^{(i)}$; otherwise, we apply NP. We can consistently get a lower gradient estimation variance regardless of the unbalanced weight and node dimension of different layers, as shown in Figure 5. Note that the dimension d in Equation (8) is set as $d = d_w^{(i)}$ for layer-wise WP, and $d = d_a^{(i)}$ for layer-wise NP, accordingly.

Table 3. Memory Cost Comparison

	Computation	Peak Memory (Vanilla)	Peak Memory (Efficient)
Inference Only	$N\times$ FWD	$N(2d_a + d_w)$	/
Weight Perturbation	$NQ\times$ FWD	Ld_w	LQ
Node Perturbation	$NQ\times$ FWD	$NLd_a + Ld_w$	$NLQ + d_w$
Back-propagation	$N\times$ (FWD + BWD)	$NLd_a + Ld_w$	/

FWD denotes forward propagation, and BWD denotes backward propagation.

— **Memory-efficient Layer-wise Gradient Estimation.** We propose a memory-efficient implementation of NP that minimizes the memory overhead. Different from BP, we only need $\mathbf{a}_i$ for the gradient estimation of ith layer in NP. With the aforementioned layer-wise gradient estimation, we estimate $\hat{\nabla}_{\mathbf{a}_{i+1}}\mathcal{L}$, compute the gradients of weights $\hat{\nabla}_{\mathbf{W}_i}\mathcal{L}$, and update $\mathbf{W}_i$ instantly after the forward computation of the ith layer. In this way, $\mathbf{a}_i$ needs not be temporarily stored. Inspired by [58], we only store the random seed that generates the perturbation instead of the whole perturbation matrix. When regenerating $\hat{\nabla}_{\mathbf{a}_{i+1}}\mathcal{L}$, we could reuse the same memory that temporarily stored $\mathbf{a}_{i+1}$. Note that the old values of $\mathbf{W}_i^{t-1}$ should be stored (via reusing the memory of $\hat{\nabla}_{\mathbf{W}_i}\mathcal{L}$ by in-place swap) when $\mathbf{W}_i$ is updated. In this way, we could re-compute $\mathbf{a}_{i+1}^{t-1}$, which guarantees the activations of the following layers are still in the $(t-1)$-th step. This ensures a *global* update (the updates of all layers are synchronized), compared with a *local* update like layer-wise block-coordinate descent [10] where different blocks (layers) are updated sequentially. The theoretical study [2] shows that local updates can generalize worse than global updates. The peak training memory comparison between inference-only, BP-based training, memory-efficient WP, and our memory-efficient NP is provided in Table 3. The peak memory of our method is slightly larger than that of memory-efficient WP but is much smaller than that of BP.

Algorithm 1 in the appendix gives the pseudo-code for the whole gradient estimation and update procedure.

3.3 Further Dimension Reduction via Sparse Training

We propose to further reduce the dimension by integrating sparse training with our BP-free training framework. Sparse training identifies and updates only the "most important" parameters while freezing others, significantly reducing the number of trainable parameters. By incorporating sparse training with BP-free training, we expect to effectively reduce the variance in ZO gradient estimation. While previous work has shown that properly selected trainable parameters can match or surpass full-model training in fine-tuning pre-trained models [49, 52, 55], existing methods are ill-suited for resource-constrained on-device training on MCUs for two main reasons:

— **Incompatibility with on-device training.** Current methods either involve model training [27, 55, 95] or evaluating gradient-based "importance" metric [39, 49, 80] to determine the "effective" parameters for each target dataset. Both necessitates a full BP graph which is not supported on MCUs.

— **Additional Memory Overhead.** Current methods use a fine selection granularity (e.g., parameter selection) [11]. It introduces additional memory as large as a model copy to maintain an importance score for all parameters.

3.3.1 Proposed BP-Free Sparse Training. To address these challenges, we propose a BP-free sparse training method compatible with MCUs' constrained memory resources. Our approach consists of two steps: BP-free trainable parameters selection and BP-free sparse training. In the

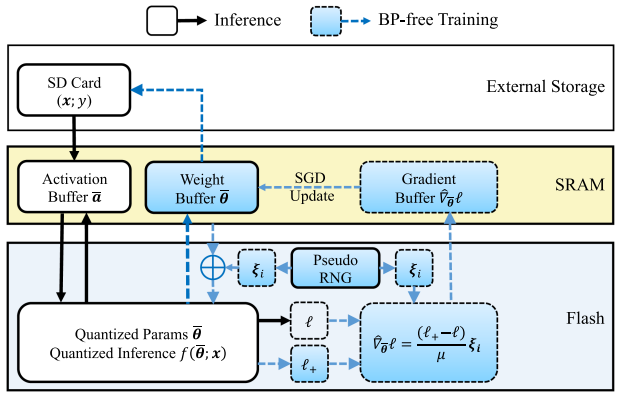

Fig. 6. Overview of BP-free training framework on MCU.

first step, we employ a brute-force, yet effective BP-free selection metric. We follow the surgical fine-tuning settings in [52] to split the model into four non-overlapping blocks. We denote them as "Block 1", "Block 2", and so on, in the order of input to output. For a new target dataset, we conduct a BP-free test training (1 epoch) for each block. The block achieving the highest accuracy gain on a held-out subset of the training data is selected. Then, we proceed with BP-free sparse training, where we only estimate the gradients of and update the parameters in the selected block, while keeping other blocks frozen. Our solution comes with the following advantages:

— **Easy Implementation.** BP-free gradient computation shares the same computation graph regardless of which parameters are selected for training. After deploying the BP-free training framework on MCUs, changing the selection of trainable parameters is purely a **software** effort.
— **MCU-compatible Overhead.** The coarse selection granularity introduces a minimal memory overhead (several scalars) and minimal computation overhead for selection (4 epochs).
— **On-device Task-adaptive Selection.** The MCU-compatible overhead allows fully on-device evaluation of the selection process and adapting to different datasets without external computation.

This approach accelerates training convergence and achieves shorter end-to-end training times. As shown in Table 7, sparse BP-free training can match or even surpass full-model BP-based training (FullTrain) in certain cases.

3.4 MCU Implementation Details

In this section, we describe the implementation details of our BP-free training on MCUs. We deploy full-model training on STM32H7B3 MCU (1184-KB SRAM including 1024-KB user SRAM and 2-MB Flash) and deploy sparse training on STM32F746 MCU (320-KB SRAM including 256-KB user SRAM and 1-MB Flash).

3.4.1 Overall Framework. Figure 6 shows the overall training framework. A few key points are summarized below:

— **Quantized Inference Engine.** We employ TinyEngine [55, 99] to deploy tailored neural network model on the MCU. As an example, we consider the tailored CNN model MCUNet-in1 [54] with 0.46M parameters. The model parameter memory usage is 478 KB, and the peak

inference run-time memory (SRAM) usage is 190 KB. The inference latency is measured as 239 ms using a single batch size with input resolution 128×128.

— **Run-time Memory:** The pre-trained parameters are hard-coded in the read-only Flash. We first store the trainable parameters in the weight buffer on SRAM. The activation buffer temporarily stores the intermediate activation between layers until they are not needed for future computation. The gradient buffer temporarily stores the gradients, and it is released once the parameter update is complete. In case we use gradient accumulation over multiple training data samples, the gradient buffer is kept between different forward (inference) runs.

— **External Memory:** We store the training data in an SD card. During training, no intermediate values are saved in the external memory. Once the training is complete, we store the checkpoint of updated model parameters in the SD card.

3.4.2 Training Process. We summarize the main characteristics of our BP-free training pipeline below.

Initialization: Each layer is assigned a perturbation mode (weight or node) based on its weight or activation dimensionality. If sparse training is enabled, layer selection is performed as described in Section 3.3.1. The parameters of selected trainable layers are copied into the SRAM weight buffer.

Training Loop: We deploy the memory-efficient layer-wise gradient estimation with adaptive WP/NP. The complete algorithm for a single training iteration is provided in Algorithm 1. Key steps include:

— **Layerwise Gradient Estimation**: For each trainable layer, the framework performs multiple perturbed forward passes to estimate gradients.
  – **Quantized Perturbation Generation**: Rademacher-distributed perturbation vectors are generated using a lightweight XORShift pseudo-random number generator [61, 71], fully implemented on-device (Appendix F).
  – **Perturbed Forward Pass**: In WP mode, perturbations are directly added to parameters in the weight buffer. The input activation of the current layer is saved in the activation buffer. The perturbed forward passes start from the current layer. In NP mode, perturbations are added to the output activation stored in the activation buffer. The perturbed forward passes start from the subsequent layer.

— **Parameter Update**: After gradient estimation, the output activation of the current layer (i.e., input to the next layer) is saved in the activation buffer, and the parameters of the current layer are updated in-place in the weight buffer. This ensures synchronized updates across all trainable layers.

3.4.3 Broader Applicability to Other Resource-Constrained Hardware. Our framework is not constrained within MCU, but also applicable to other resource-constrained hardware like FPGAs or ASICs, or emerging computing platforms (e.g., integrated photonics, probabilistic circuits). By reusing a well-developed inference accelerator at hand, only minimal additional modules are needed to implement BP-free training. These additional modules are way easier to design compared with a BP computation graph, and can be implemented efficiently with an external software-based control system (MCU/edge CPU) or integrated in the hardware system.

4 Experimental Results

4.1 Experiment Setup

Datasets. We evaluate various training methods on multiple image classification datasets of three types:

— **Image Corruption.** We evaluate our BP-free training framework on a widely used out-of-distribution image corruption benchmarks **CIFAR-10-C** [35]. The task is to classify images from the target datasets, which consist of images corrupted by different kinds of corruptions un-seen in the source pre-training dataset with five severity levels. We run experiments over 10 of the corruptions (gaussian noise, impulse noise, shot noise, fog, frost, snow, defocus blur, elastic transform, brightness, contrast, and defocus blur). We tune on a training dataset with 1000 images and evaluate on a held-out test dataset from each of the corruptions.
— **Fine-grained Vision Classification (FGVC).** Following *MCUTrain* [55] and *TinyTrain* [49], we also test our BP-free training framework on several **Fine-grained Vision Classification** (**FGVC**) datasets including Cars [47], CUB [87], Flowers [64], Food [7], and Pets [72]. The training datasets contain approximately 30–50 samples of each class.
— **Visual Wake Words (VWW)**. We further demonstrate our BP-free training framework on real-life IoT application VWW [18] which represents a common microcontroller vision use-case of identifying whether a person is present in the image.

Model Configurations: We employ the MCU-compatible CNN model *MCUNet-in1* [54] with 22.5M MACs and 0.46M parameters. The model is pre-trained on ImageNet-1k [23] and quantized to the INT8 format by post-training quantization [41]. The batch normalization [40] layers are fused into the adjacent convolution layer. Only quantized model parameters and quantized inference computation graphs are implemented on the MCU. Unless otherwise stated, all models used in this article are in the INT8 format.

Training Configurations: We use a resolution of 128×128 following [55] and gradient accumulation with 100 steps for all datasets, models, and baselines for a fair comparison. We apply vanilla SGD [8] / ZO-SGD [63] without momentum as the optimizer to avoid additional memory cost of optimizer states. No weight decay was applied. For experiments on the CIFAR-10-C datasets, the MCUNet model pre-trained on ImageNet-1k is transferred to the CIFAR-10 [48] dataset with 90.17% test accuracy. In our BP-free training and the BP-based training baselines, we train the model for 50 epochs with an initial learning rate of 0.01 following the setups in [52]. For experiments on FGVC datasets, we train the model for 50 epochs following [55] in both BP-based and our BP-free methods. The initial learning rate is set as 0.1. For experiments on VWW, we train the model for 20 epochs following [55] in both BP-based and our BP-free methods. The initial learning rate is set as 0.01. For all experiments, we use a cosine learning rate decay. In the BP-free training scenarios, the smoothing factor is set as $\mu = 1$, the query budget is $Q = 100$ for each layer. The trainable parameter selection is evaluated only once at the beginning of training. To obtain experiment results of multiple downstream datasets faster, we simulate the training process on GPU with batch size 100. As our model does not have any batch-dependent operations (e.g., batch normalization [40]), training with batch size 100 is equivalent to training with batch size 1 along with 100 steps of gradient accumulations, with learning rate scaled accordingly. For NP, i.i.d. perturbations are simultaneously applied to the pre-activation vector in a mini-batch at each layer. For WP, the i.i.d. perturbations are shared across a mini-batch.

4.2 Algorithmic Performance

4.2.1 Effectiveness of Various ZO Optimization Improvement Techniques. We first validate the effectiveness of the key convergence improvement techniques used in our method. To simplify the comparison, we consider fine-tuning the MCUNet on the CIFAR-10-C with a Gaussian noise corruption at severity 5 and fix the trainable parameters as the weight matrices and biases of four point-wise convolution layers in block 1. We provide the results over multiple datasets in Appendix D.

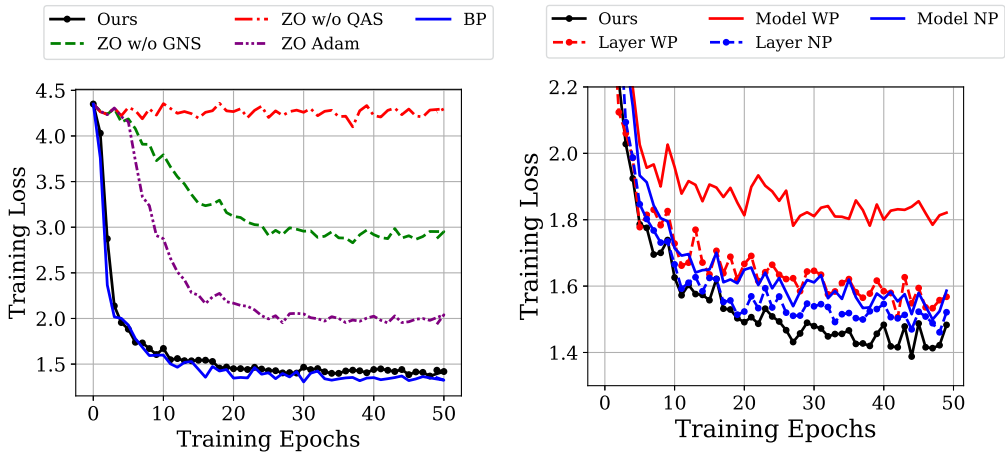

Fig. 7. Left: Training loss curves w/ and w/o learning-rate scaling applied. QAS: quantization-aware scaling. GNS: gradient-norm scaling. Right: Training loss curves with different ZO gradient estimation methods applied. We select the best learning rate for each method.

— **Effectiveness of Learning Rate Scaling.** The training curves with and without learning rate scaling are provided in Figure 7 (Left). We apply layer-wise activation perturbation for the ZO gradient estimation. Without gradient-norm scaling or quantization-aware scaling, the training converges slowly or even cannot converge. Adaptive learning rate methods like Adam [46] cannot fully address this challenge while costing 3× memory to save the optimizer states. With both scaling methods applied, our BP-free training follows a similar training dynamics as full-precision BP-based training without using any extra memory.

— **Effectiveness of the Layer-Wise Mixture of Weight and Node Perturbations.** We further compare the effectiveness of different perturbation methods. The first two layers have larger node dimensions, while the other two layers have larger weight dimensions. The training results are provided in Figure 7 (Right). For a fair comparison, we consider the same computation budget (measured by the number of forward passes) for gradient estimation at each step. Layer-wise gradient estimation outperforms model-wise gradient estimation in both WP and NP, showing that dis-entangling the gradient estimation layer-by-layer improves convergence. Our layer-wise gradient estimation with adaptive WP/NP outperforms all other methods and achieves the best training convergence.

— **Effectiveness of Task-Adaptive Sparse Training.** Table 4 compares the average testing accuracy across various corruption types after training different blocks. Since different corruption types introduce varying distribution shifts from the pre-trained data, they benefit from fine-tuning different layers. A fixed selection of trainable blocks cannot guarantee optimal performance across all corruption types. Therefore, a task-adaptive selection is crucial, and our on-device selection metric is able to identify the most effective block at the onset of training. Consequently, our adaptive approach consistently outperforms all fixed selection methods.

4.2.2 Performance Comparison with Other BP-Free Training Methods. Next, we compare our proposed methods with two **state-of-the-art** (**SOTA**) BP-free training baselines: *MeZO* [58] and *DeepZero* [11]. *MeZO* [58] uses RGE with memory-efficient model-wise WP as the gradient estimator. All trainable parameters are perturbed and updated at the same time. *DeepZero* [11] uses

Table 4. Average Test Accuracy Comparison of Training a Fixed Selection of Layers for All Corruptions and Applying the Task-Adaptive Layer Selection for Different Corruptions

		noise	blur	weather	digital
Fixed Selection	Block 1	55.17	78.91	79.25	75.76
	Block 2	**65.33**	80.31	81.33	77.45
	Block 3	59.75	79.51	81.33	79.32
	Block 4	57.11	78.56	79.31	77.11
Adative Selection		**65.33**	**80.91**	**82.00**	**80.56**

Table 5. Comparison between Our BP-Free Training and Other SOTA BP-Free Training Baselines

	iter. forwards	epochs	total forwards	test accuracy (%)
MeZO [58]	2	10000	**2.00E+07**	18.33
DeepZero [11]	49457	50	2.47E+09	52.78
Ours	400	50	**2.00E+07**	**66.78**

sparsity-assisted **coordinate-wise gradient estimator** (**CGE**). Each single parameter (coordinate) is perturbed sequentially. Note that the proposed gradient sparsity pattern exploration of *DeepZero* needs additional memory as large as three model copies, which is prohibitive on tiny edge devices. Therefore, we only apply a random gradient sparsity. Table 5 shows the model adaptation results of fine-tuning the MCUNet on the CIFAR-10-C with a Gaussian noise corruption at severity 5. The main results are summarized below:

- **MeZO [58].** With the same same total number of forward evaluations, *MeZO* attains the least performance improvement. MeZO fails to converge due to (1) lack of proper learning-rate scaling, (2) large variance of model-wise ZO gradient estimation with WPs, and (3) the general behavior learned in pre-training (e.g., feature extracting, classifier) being corrupted in ZO fine-tuning. In summary, MeZO is not an effective generic BP-free training method. We would like to remark that MeZO still fails to converge well even if our gradient scaling method is used.
- **DeepZero [11].** *DeepZero* attains low-variance ZO gradient by coordinate-wise gradient estimation, at the cost of two orders-of-magnitude more forward evaluations per iteration. *DeepZero* still fails to match the testing accuracy of our method. We posit that *DeepZero*'s parameter-wise sparsity pattern scheme is less effective in detecting the important parameters for image corruption datasets.
- **Our Method.** Our proposed BP-free training method achieves the best efficiency and accuracy compared with the above baseline methods.

We further point out that DeepZero and MeZO still cannot achieve the same performance as our method, even if these two baseline methods are improved by applying the same trainable parameter selection, as shown in Table 6.

4.3 On-Device Training Results

Now we implement our method on the MCU, and compare its performance with existing on-device training methods on two benchmarks: CIFAR-10-C datasets and FGVC datasets.

Baselines. We compare our method with the following four on-device BP training baselines: (1) *No Adaptation* does not perform any on-device training. (2) *FullTrain* fine-tunes the entire model.

Table 6. Comparison between Our BP-Free Training and Other SOTA BP-Free Training Baselines when Applied the Same Trainable Parameter Selection

	iter. forwards	epochs	total forwards	test accuracy (%)
MeZO [58]	2	10000	**2.00E+07**	40.56
DeepZero-0 [11]	18560	50	9.28E+08	60.67
DeepZero-0.9 [11]	1856	50	9.28E+07	54.33
Ours	400	50	**2.00E+07**	**66.78**

DeepZero-0 denotes dense training, and DeepZero-0.9 means training with a 90% gradient sparsity.

(3) *MCUTrain* is the SOTA method for training under an extremely low memory budget (*e.g.*, MCU with 256-KB SRAM), which statically determines the set of layers/channels to update based on *offline* evolutionary search on the *cloud* servers before MCU deployment, and updates the selected layers/channels on MCU. (4) *TinyTrain [49]* is the most recent SOTA on-device training method that employs *on-device* layer/channel selection. For *MCUTrain* and *TinyTrain*, we select the best layer selection setting of each method that could fit into the memory budget of an MCU with 256-KB SRAM.

Performance Evaluation. Through our experiments, we evaluate the following performance metrics:

— **Memory Cost.** We profile **analytic memory** to reflect the memory cost of different training methods. We count the memory required for training, including the peak memory of inference, trainable parameters, and the extra memory for gradient computation and parameter update. For training with BP, the extra memory includes the saved intermediate activation, and the gradients of weights. For BP-free training, the extra memory includes loss values, random seeds, and a buffer needed to save the largest gradients of weights (only for NP). The analytic memory determines whether the training setting can be deployed under the memory budget. We then measure **on-device memory** to measure the actual memory cost for MCU-deployable training methods.

— **Computation Analysis.** We count the number of MACs to reflect the computation cost of different training methods. For each training iteration, the computation consists of 1 forward propagation, gradient computation (by BP or ZO gradient estimation), and parameter update. The analytic computation cost is implementation-agnostic.

— **End-to-end Training Time.** We measure the end-to-end training time as per iteration latency times the number of iterations. The per-iteration latency depends on the implementation backends. We consider a general TFLite Micro [50] inference backend and an MCU domain-specific inference backend TinyEngine [55, 99]. Note that TFLite Micro does not support training, TinyEngine has support for sparse training but the update space is constrained within some ending layers. We used the kernel implementations to profile the **projected per-iteration latency** to perform full-model BP. We then measure the actual **on-device per-iteration latency** for MCU-deployable training methods.

— **End-to-end Training Energy.** We measure the power consumption using a YOJOCK USB power meter. The power remains stable ($\pm 0.01W$) during the training. Therefore, we first derive the energy consumption for each iteration following the equation: Energy = Power × Time, then we measure the end-to-end training energy as per iteration energy times the number of iterations.

4.3.1 Results on the CIFAR-10-C Dataset. Here we consider on-MCU training to adapt a pre-trained model to the CIFAR-10 dataset with various levels of input image corruptions.

Table 7. Test Accuracy of CIFAR-10-C on 12 Representative Corruptions at Severity Level 5. The Training Settings that Could be Implemented on an MCU with 256 KB SRAM are underlined

		Noise				Blur				Weather		Digital	
Model	Method	Gauss.	Shot	Impul.	Speckle	Gauss.	Defoc.	Motion	Zoom	Fog	Brit.	Contr.	Elas.
No Adaptation		17	21	25	27	72	76	67	80	73	83	36	71
FP32	FullTrain	73.89	72.44	76.78	78.44	77.89	81.00	78.44	80.78	82.67	86.67	86.33	76.44
INT8	FullTrain	44.33	37.78	41.55	54.00	44.11	65.11	42.78	55.57	34.22	73.00	20.27	77.78
	MCUTrain [54]	61.78	63.67	55.55	65.33	**81.55**	83.44	**82.44**	**87.33**	76.22	**89.22**	**71.78**	**81.11**
	BP-free (Ours)	**66.78**	**66.22**	**61.00**	**67.33**	80.11	**83.67**	81.00	86.33	**78.33**	88.22	69.78	79.56

Other training settings are out-of-memory and only listed for comparison.

— **Accuracy.** Table 7 summarizes the results of our method and various baseline methods.
  – **No Adaptation** attains very poor accuracy, showing that on-device training is necessary.
  – **FullTrain** fine-tunes the whole model with BP, and it serves as a strong baseline with the assumption of unlimited memory and computation resources. While *FullTrain* achieves excellent accuracy in the FP32 format, its INT8 variant experiences remarkable accuracy drop due to the difficulty of handling a real-quantized model in BP.
  – **MCUTrain** [54] is a SOTA on-device training method under an extremely low memory budget. This baseline method determines the set of layers/channels to update before an MCU deployment. The *MCUTrain* method performs better than the IN8 *FullTrain* approach, but still experiences remarkable (around 7–10%) accuracy drop in certain corruption types (shot noise, impulsive noise, etc) compared with FP32 *FullTrain*. Reference [52] showed that fine-tuning the layers where distribution shifts happen can achieve the best performance. Since corruptions lead to distribution shifts in the starting layers, one has to fine-tune these early layers with a significant memory overhead in the BP process.
  – **Our BP-free method** achieves the best accuracy among all on-device training methods in the INT8 format, and it even can match the performance of FP32 *FullTrain*. Since our method train the **full model**, it significantly outperforms the *MCUTrain* method.

— **Hardware Cost (Memory, Computing, Latency, and Energy).** Next, we evaluate the hardware cost of training the MCUNet on an MCU. The peak memory, computation cost, and latency of one MCUNet inference are 190 KB, 22.5 MMACs, and 190 ms, respectively. We evaluate single-batch training with no gradient accumulation. The query budget of BP-free training is $Q = 50$ for each layer. We perform the same on-device task-adaptive layer selection (Block 3) for both BP-based and BP-free sparse training. The detailed results of full-model training and sparse training are listed in Table 8 and Table 9, respectively. We summarize the key information below.
  – **Peak Memory.** The peak memory of our BP-free training remains the minimum possible training memory (trainable parameters + peak inference memory) in both full-model and sparse training. In contrast, full-model BP training easily runs out of memory (3,650 KB) due to the extra memory for saving large activation values in starting layers.
  – **Latency, Training Time and Energy.** Our BP-free training has a longer per-iteration latency as it needs $Q = 50$ forward evaluations per layer to reduce the ZO gradient variance so that our method can converge after the same training iterations of a BP-based method. Reducing Q can reduce the latency per iteration, but will result in more iterations to converge. We provide detailed tradeoffs for accuracy versus efficiency in Appendix C. This results in a much longer end-to-end training time and energy consumption when using the general TF-Lite backend for training.
  – **Accelerated Performance.** Due to the BP-free nature, our training framework can be easily accelerated by using any existing inference accelerators. For instance, by using the

Table 8. Comparison of the Hardware Cost of Full-Model Training

		MCUNet				
Backend	Method	Memory (KB)	Compute (MMACs)	Latency (ms)	E2E Time (s)	E2E Energy (kJ)
TFLite [20]	BP	3,650	76.2	13,398	6.70E+05	/
	BP-free	668	7,857.5	428,496	2.14E+07	4.24E+04
TinyEngine [54]	BP-free	668	7,857.5	72,839	3.64E+06	6.30E+03

E2E: end-to-end.

Table 9. Comparison of the Hardware Cost of Sparse Training

		MCUNet				
Method	Test Accuracy (%)	Memory (KB)	Compute (MMACs)	Latency (ms)	E2E Time (s)	E2E Energy (kJ)
BP [55]	61.78	3,650	40.4	495	2.48E+04	4.36E+01
BP-free	62.33	209	1,162.5	9,939	4.97E+05	8.60E+02

The underlined are measured results on an STM32F746 MCU. E2E: end-to-end.

Table 10. Validation Accuracy Comparison on Transfer Learning to 5 FGVC Datasets with Our BP-Free Method and Other BP Training Baselines

		Accuracy (%) (MCUNet with 480KB paramemeters)					
Method	BP type	Cars	CUB	Flowers	Food	Pets	Average
Full	FP32	56.7	56.2	88.8	**55.7**	79.5	**67.4**
TinyTrain [49]	FP32	55.2	57.8	**89.1**	52.3	80.9	67.1
MCUTrain [55]	INT8	54.6	57.3	88.1	52.1	81.5	66.7
BP-free (Ours)	/	**60.1**	**58.1**	85.0	45.5	**81.6**	66.1

The training settings that could be implemented on an MCU with 256 KB SRAM are underlined. Other training settings are out-of-memory and only listed for comparison.

SOTA TinyEngine [54] for inference, the per-iteration latency of our BP-free training could be greatly reduced, leading to a greatly reduced total training time and training energy. By further incorporating the sparse training, we could achieve a comparable end-to-end training time and energy cost as BP full-model training.

— **Remark.** It is possible to further reduce the on-device BP-free training energy/latency if we can improve the convergence of ZO optimization by incorporating variance reduction [56] or improving the query efficiency in gradient estimation [73, 77]. We leave them for future works.

4.3.2 Results on the FGVC Dataset. Table 10 summarizes the testing the accuracy of our BP-free method and various BP-based training. The FGVC dataset mainly perceive output-level distribution shifts, thus training some last layers could match or even surpass full model training [52, 55]. Nevertheless, due to the extremely limited memory budget, the BP-based method can only afford training three middle layers [55, 99] even if exhaustive optimization tricks (e.g., compile-time differentiation, backward graph pruning, operator reordering) applied. This is enough for some easy datasets like Flowers, but insufficient for more challenging datasets including Cars. Our BP-free method can train all layers with no extra memory cost. Therefore, even with a large ZO gradient estimation variance, our method still outperforms BP-based training on most datasets (Cars, CUB, and Pets).

Table 11. Validation Accuracy Comparison on Transfer Learning to IoT Application VWW

Model	Method	Memory (KB)	Accuracy (%)
FP32	FullTrain	3,650	86.71
INT8	ClsOnly	191	83.20
INT8	MCUTrain [55]	268	86.68
INT8	BP-free (ours)	240	86.16

The training settings that could be implemented on an MCU with 256 KB SRAM are underlined. Other training settings are out-of-memory and only listed for comparison.

4.3.3 Results on IoT Application Visual Wake Words (VWW). Table 11 summarizes the memory cost and test accuracy of our BP-free method and BP-based method on the VWW dataset. For fair comparisons, we set the same trainable parameters for MCUTrain [55] and our BP-free training. Only training the last classifier (ClsOnly) is memory-efficient, but can only converge to a suboptimal result. Training backbone weight matrices are needed. However, implementing the whole BP computation graph exceeds the memory budget. Our proposed BP-free training method greatly reduces the memory budget (only 190 KB peak inference memory + 49 KB to store updated parameters) and achieves a similar validation accuracy (0.5% lower) compared with BP-based method MCUTrain [55]. The on-device accuracy is above the common requirements for edge ML (>80%) and matches the industry inference-only solution MobileNetV2 (86.2% under 256 KB) [54]. The above results demonstrate our BP-free training framework's applicability to practical IoT applications.

4.3.4 Comparison between BP-Free and BP-Based Training. We provide comparisons in various resource constraints between BP-free and BP-based training to help users to identify whether the BP-free method performs better or worse.

Energy-efficiency tradeoff between BP-based training and BP-free training. The benefit of our current BP-free training is not on-device energy saving. Instead, it makes full-model training feasible in MCU, which is impossible via conventional BP-based training. This eliminates the off-device energy. We discuss the tradeoffs in detail below.

BP-based training:

- **Lower on-device training energy and latency** since exact stochastic gradient estimation will ensure faster convergence in training.
- **High off-device energy and latency overhead**: Full-model BP is infeasible on MCUs due to memory constraints. One has to optimize the sparse BP training graph to fits the memory budget before deployment, which requires off-device computation (e.g., evolutionary search and computation reordering [55]). This leads to significant energy and data transmission costs.
- **Higher data/task transfer overhead:** The optimized graph is fixed and tailored to a specific dataset. One has to repeat the above off-device optimization when transferring to new data/task.

BP-free training (ours):

- **Higher on-device training energy and latency** due to slower convergence of ZO optimization. However, we remark that BP-free training may show advantages in both latency and energy saving on emerging platforms such as integrated photonics, due to the parallel perturbations using wavelength multiplexing [96].

— **No off-device overhead** as BP-free enables full-model training on MCUs.
— **Lower data/task transfer overhead** Our BP-free framework can adapt to new task/data on-device via task-specific parameter selection (see Section 3.3).

Where the BP-free method performs better or worse compared to BP-based approaches.

— Under **extreme memory constraints**, the BP-free method performs **better**. BP-free method enables full-model training with minimal memory overhead (peak inference memory + trainable parameters), while BP-based training can only update some ending layers due to memory constraints. In certain cases, BP-free achieves better performance than BP-based method, as shown in Table 7.
— Under **privacy constraints**, the BP-free method performs **better**. The BP-free framework enables complete on-device training. This presents a significant advantage over BP-based frameworks, which require cloud server communication for BP computation graph pruning and optimization, potentially compromising data security and data privacy.
— Under **time-to-market constraints**, the BP-free method performs **better**. Our proposed BP-free training framework greatly reduces the training accelerator design difficulty with maximum flexibility in task adaptation.
— Under **extreme latency/energy constraints**, the BP-free method performs **worse**. Compared with optimized BP-based training, BP-free training takes longer on-device training time and larger training energy cost due to the inherent slow convergence of ZO optimization. However, we remark that BP-based methods require additional latency/energy costs (data transmission between edge and cloud, BP graph pruning search, BP graph optimization, etc.).

At the current stage, BP-free training is more suitable for application scenarios where memory, data privacy, and time-to-market (rather than energy and latency) are the major concerns. The tradeoff among memory, flexibility, design/deployment difficulty, and end-to-end training time and energy costs depends on the actual settings. We hope that the above comparison can enhance the understanding of the strengths and limitations of our proposed BP-free training framework, and help the decision under different settings.

5 Related Work

In this section, we review some prior works related to this article.

5.1 On-Device Training

On-device inference has been well studied to deploy a pre-trained DNN for inference on the edge. Driven by the demand to adapt edge-deployed neural network models to new data/new tasks [9, 49, 55] or to unseen distribution shifts at test time [84, 85], there have been increasing interests in DNN training on edge devices. However, edge devices usually have tight memory and computation resources, and run without an operating system, making it infeasible to implement standard deep learning training frameworks that rely on AD [3]. To implement training with BP on edge devices one has to implement gradient computations by hand, which is time-consuming and needs exhaustive memory/computation optimization, or add external devices to perform gradient computation [45, 88]. Moreover, BP-based training requires extra memory to save intermediate results as well as high-precision computation. The most memory-efficient scheme is to skip all weight updates (e.g., fine-tuning only the last layer [62, 75], bias only [9], normalization parameters [28]), yet such a scheme leads to considerable accuracy drop compared with full-model training. As a result, it is almost infeasible to support training on the same device that only supports inference.

MCUTrain [55] enabled training on a microcontroller with merely 256 KB SRAM and matched cloud training results on the VWW dataset. However, *MCUTrain* has a significant cloud computation overhead, requiring thousands of runs of evolutionary search to find the best weight update scheme that fits the memory budget. *MCUTrain* also needs compilation-time optimization to reduce the peak memory during BP. As a result, this method is not fully on-device training: with a large cloud overhead, it can only manage to train four layers (out of 42). *TinyTrain* [49] further enabled on-device parameter selection and showed comparable performance to full-model training on some vision classification tasks. However, to implement the update scheme on edge devices without an operating system (e.g., MCU, FPGA), compilation-time optimization is still needed. As a result, to achieve comparable training performance, the current *TinyTrain* method is still not fully on-device training. In summary, SOTA BP-based on-device training can only afford training some last layers due to the memory constraints.

5.2 BP-Free Training

Due to the challenge of implementing BP on edge devices, several BP-free training algorithms have been proposed. These methods have gained more attention in recent years as BP is also considered "biologically implausible". ZO optimization [10, 25, 56, 63] plays an important role in signal processing and adversarial machine learning where actual gradient information is infeasible (*e.g.*, black-box attack [13, 15, 82]). Recently, ZO optimization is also applied in neural network training. However, due to the high variance of ZO gradient estimation, previous work focusing on adapting low-dimensional auxiliary modules including input/feature reprogramming [81, 86], prompt/adapter tuning for emerging large language [34, 89], vision [65, 68], and vision-language models [69], but not the backbone parameters of a pre-trained model. Recently, [58, 94] showed that ZO optimization was effective for full-parameter fine-tuning of **large language models** (**LLMs**) up to 66 Billion parameters. However, this is only applicable when the pre-trained model has a very low intrinsic dimension (200–400) [1, 59], which is only observed in LLMs. For more general cases, the scalability of ZO training remains a fundamental challenge. Reference [97] scaled up ZO training from scratch by leveraging tensor-compressed training for dimension reduction. DeepZero [11] further scaled up ZO training to train a ResNet-20 with 270K parameters from scratch. However, DeepZero relies on low-variance coordinate-wise gradient estimation which is extremely computation-inefficient thus not suitable for on-device training. ZO optimization also provides a promising solution to training neural networks on emerging computing platforms (e.g., **optical neural networks** (**ONN**) [31–33, 96]) where BP is infeasible due to the non-differentiability or limited observability of the analog hardware. Other BP-free training methods include forward-forward algorithm [36] and PEPITA [21] for biologically-plausible learning, forward gradient method [4, 26, 76] based on forward-mode AD for memory-efficient gradient computation or to avoid stacked BP [17, 92] when higher-order derivatives are needed, log-likelihood method [43], NP [19, 37], feedback alignment method [66], and input-weight alignment method [16]. Binarized PEPITA and forward-forward methods are further proposed to reduce the memory overhead [38]. However, this method needs to store full-precision model parameters, making it not suitable for MCUs' extremely constrained memory budget that admits only quantized model parameters. Scalability, training stability, and convergence remain the top challenges of all emerging BP-free training frameworks.

6 Conclusion

In this article, we have proposed a quantized BP-free training framework on MCU. With this framework, a quantized inference engine can be easily converted to a training engine. Our method leverages quantized ZO optimization to achieve full-model training on an MCU at the similar memory

cost of inference. This approach has addressed the long-standing memory challenge that prevents realistic training on an MCU. To tackle the slow convergence commonly associated with ZO optimization, we have introduced several innovations, including learning-rate scaling, dimension-reduction techniques such as layer-wise NP, and task-adaptive sparse training. These enhancements have stabilized the training process and improved the convergence.

Our BP-free methods have demonstrated superior training performance over existing BP-based training solutions, primarily due to its full-model training capability under low memory budget and the ability to adapt the layer selection dynamically to specific tasks on-device. This makes the framework highly suitable for vast real applications, where minimal hardware complexity and maximum flexibility in task adaptation are essential, opening the door to broader adoption of on-device training in resource-limited environments.

However, our BP-free training method needs more training iterations than BP-based methods, resulting in longer end-to-end training time. Future research directions include the development of various novel techniques to further improve the convergence. Additionally, investigating the adaptability of BP-free training across various neural network architectures, application domains, and platforms (e.g., FPGA, ASIC, integrated photonics, probabilistic circuits) will further expand its applicability, potentially impacting many application areas. We refer readers to Appendix E for an extended discussion.

References

[1] Armen Aghajanyan, Sonal Gupta, and Luke Zettlemoyer. 2021. Intrinsic dimensionality explains the effectiveness of language model fine-tuning. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. 7319–7328.

[2] Pierre Baldi and Peter Sadowski. 2016. A theory of local learning, the learning channel, and the optimality of backpropagation. *Neural Networks* 83 (2016), 51–74.

[3] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2018. Automatic differentiation in machine learning: A survey. *Journal of Machine Learning Research* 18, 153 (2018), 1–43.

[4] Atılım Güneş Baydin, Barak A. Pearlmutter, Don Syme, Frank Wood, and Philip Torr. 2022. Gradients without backpropagation. *arXiv preprint arXiv:2202.08587* (2022).

[5] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. 2013. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432* (2013).

[6] Jérôme Bolte and Edouard Pauwels. 2020. A mathematical model for automatic differentiation in machine learning. *Advances in Neural Information Processing Systems* 33 (2020), 10809–10819.

[7] Lukas Bossard, Matthieu Guillaumin, and Luc Van Gool. 2014. Food-101–mining discriminative components with random forests. In *ECCV 2014: Proceedings of the 13th European Conference on Computer vision, part VI 13*. Springer, 446–461.

[8] Léon Bottou. 2010. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010: 19th International Conference on Computational Statistics*. Springer, 177–186.

[9] Han Cai, Chuang Gan, Ligeng Zhu, and Song Han. 2020. Tinytl: Reduce memory, not parameters for efficient on-device learning. *Advances in Neural Information Processing Systems* 33 (2020), 11285–11297.

[10] HanQin Cai, Yuchen Lou, Daniel McKenzie, and Wotao Yin. 2021. A zeroth-order block coordinate descent algorithm for huge-scale black-box optimization. In *Proceedings of the International Conference on Machine Learning*. PMLR, 1193–1203.

[11] Aochuan Chen, Yimeng Zhang, Jinghan Jia, James Diffenderfer, Jiancheng Liu, Konstantinos Parasyris, Yihua Zhang, Zheng Zhang, Bhavya Kailkhura, and Sijia Liu. 2023. Deepzero: Scaling up zeroth-order optimization for deep model training. In *The Twelfth International Conference on Learning Representations*.

[12] Jianfei Chen, Yu Gai, Zhewei Yao, Michael W. Mahoney, and Joseph E. Gonzalez. 2020. A statistical framework for low-bitwidth training of deep neural networks. *Advances in Neural Information Processing Systems* 33 (2020), 883–894.

[13] 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*. 15–26.

[14] Tianlong Chen, Jonathan Frankle, Shiyu Chang, Sijia Liu, Yang Zhang, Zhangyang Wang, and Michael Carbin. 2020. The lottery ticket hypothesis for pre-trained bert networks. *Advances in Neural Information Processing Systems* 33 (2020), 15834–15846.

[15] Shuyu Cheng, Yinpeng Dong, Tianyu Pang, Hang Su, and Jun Zhu. 2019. Improving black-box adversarial attacks with a transfer-based prior. *Advances in Neural Information Processing Systems* 32 (2019).
[16] Ping-yeh Chiang, Renkun Ni, David Yu Miller, Arpit Bansal, Jonas Geiping, Micah Goldblum, and Tom Goldstein. 2022. Loss landscapes are all you need: Neural network generalization can be explained without the implicit bias of gradient descent. In *Proceedings of the 11th International Conference on Learning Representations*.
[17] Junwoo Cho, Seungtae Nam, Hyunmo Yang, Seok-Bae Yun, Youngjoon Hong, and Eunbyung Park. 2024. Separable physics-informed neural networks. *Advances in Neural Information Processing Systems* 36 (2024).
[18] Aakanksha Chowdhery, Pete Warden, Jonathon Shlens, Andrew Howard, and Rocky Rhodes. 2019. Visual wake words dataset. *arXiv preprint arXiv:1906.05721* (2019).
[19] Sander Dalm, Marcel van Gerven, and Nasir Ahmad. 2023. Effective learning with node perturbation in deep neural networks. *arXiv preprint arXiv:2310.00965* (2023).
[20] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, et al. 2021. Tensorflow lite micro: Embedded machine learning for tinyml systems. *Proceedings of Machine Learning and Systems* 3 (2021), 800–811.
[21] Giorgia Dellaferrera and Gabriel Kreiman. 2022. Error-driven input modulation: Solving the credit assignment problem without a backward pass. In *Proceedings of the International Conference on Machine Learning*. PMLR, 4937–4955.
[22] Yury Demidovich, Grigory Malinovsky, Igor Sokolov, and Peter Richtárik. 2024. A guide through the zoo of biased SGD. *Advances in Neural Information Processing Systems* 36 (2024).
[23] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *Proceedings of the 2009 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 248–255.
[24] Hao Di, Haishan Ye, Yueling Zhang, Xiangyu Chang, Guang Dai, and Ivor W. Tsang. 2024. Double variance reduction: A smoothing trick for composite optimization problems without first-order gradient. In *Proceedings of the 41st International Conference on Machine Learning*. 10792–10810.
[25] 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 (2015), 2788–2806.
[26] Louis Fournier, Stéphane Rivaud, Eugene Belilovsky, Michael Eickenberg, and Edouard Oyallon. 2023. Can forward gradient match backpropagation?. In *Proceedings of the International Conference on Machine Learning*. PMLR, 10249–10264.
[27] Jonathan Frankle and Michael Carbin. 2018. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations*.
[28] Jonathan Frankle, David J. Schwab, and Ari S. Morcos. 2020. Training batchnorm and only batchnorm: On the expressive power of random features in CNNs. In *International Conference on Learning Representations*.
[29] Katelyn Gao and Ozan Sener. 2022. Generalizing Gaussian smoothing for random search. In *Proceedings of the International Conference on Machine Learning*. PMLR, 7077–7101.
[30] Saeed Ghadimi and Guanghui Lan. 2013. Stochastic first-and zeroth-order methods for nonconvex stochastic programming. *SIAM Journal on Optimization* 23, 4 (2013), 2341–2368.
[31] Jiaqi Gu, Chenghao Feng, Zheng Zhao, Zhoufeng Ying, Ray T. Chen, and David Z. Pan. 2021. Efficient on-chip learning for optical neural networks through power-aware sparse zeroth-order optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 7583–7591.
[32] Jiaqi Gu, Zheng Zhao, Chenghao Feng, Wuxi Li, Ray T. Chen, and David Z. Pan. 2020. FLOPS: Efficient on-chip learning for optical neural networks through stochastic zeroth-order optimization. In *Proceedings of the 2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.
[33] Jiaqi Gu, Hanqing Zhu, Chenghao Feng, Zixuan Jiang, Ray Chen, and David Pan. 2021. L2ight: Enabling on-chip learning for optical neural networks via efficient in-situ subspace optimization. *Advances in Neural Information Processing Systems* 34 (2021), 8649–8661.
[34] Zixian Guo, Yuxiang Wei, Ming Liu, Zhilong Ji, Jinfeng Bai, Yiwen Guo, and Wangmeng Zuo. 2023. Black-box tuning of vision-language models with effective gradient approximation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*. 5356–5368.
[35] Dan Hendrycks and Thomas Dietterich. 2019. Benchmarking neural network robustness to common corruptions and perturbations. In *International Conference on Learning Representations*.
[36] Geoffrey Hinton. 2022. The forward-forward algorithm: Some preliminary investigations. *arXiv preprint arXiv:2212.13345* (2022).
[37] Naoki Hiratani, Yash Mehta, Timothy Lillicrap, and Peter E. Latham. 2022. On the stability and scalability of node perturbation learning. *Advances in Neural Information Processing Systems* 35 (2022), 31929–31941.
[38] Baichuan Huang and Amir Aminifar. 2025. Binary forward-only algorithms. *IEEE Design & Test* (2025), 1–1.

[39] Kai Huang, Hanyun Yin, Heng Huang, and Wei Gao. 2023. Towards green AI in fine-tuning large language models via adaptive backpropagation. In *The Twelfth International Conference on Learning Representations*.
[40] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the International Conference on Machine Learning*. PMLR, 448–456.
[41] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2704–2713.
[42] Kaiyi Ji, Zhe Wang, Yi Zhou, and Yingbin Liang. 2019. Improved zeroth-order variance reduced algorithms and analysis for nonconvex optimization. In *Proceedings of the International Conference on Machine Learning*. PMLR, 3100–3109.
[43] Jinyang Jiang, Zeliang Zhang, Chenliang Xu, Zhaofei Yu, and Yijie Peng. 2023. One forward is enough for neural network training via likelihood ratio method. In *The Twelfth International Conference on Learning Representations*.
[44] Qing Jin, Jian Ren, Richard Zhuang, Sumant Hanumante, Zhengang Li, Zhiyu Chen, Yanzhi Wang, Kaiyuan Yang, and Sergey Tulyakov. 2023. F8Net: Fixed-point 8-bit only multiplication for network quantization. In *International Conference on Learning Representations*.
[45] Biresh Kumar Joardar, Nitthilan Kannappan Jayakodi, Janardhan Rao Doppa, Hai Li, Partha Pratim Pande, and Krishnendu Chakrabarty. 2020. GRAMARCH: A GPU-ReRAM based heterogeneous architecture for neural image segmentation. In *Proceedings of the 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 228–233.
[46] Diederik P. Kingma. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
[47] Jonathan Krause, Michael Stark, Jia Deng, and Li Fei-Fei. 2013. 3D object representations for fine-grained categorization. In *Proceedings of the IEEE International Conference on Computer Vision Workshops*. 554–561.
[48] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).
[49] Young D. Kwon, Rui Li, Stylianos I. Venieris, Jagmohan Chauhan, Nicholas D. Lane, and Cecilia Mascolo. 2024. TinyTrain: Resource-aware task-adaptive sparse training of DNNs at the data-scarce edge. In *Proceedings of the 41st International Conference on Machine Learning*. 25812–25843.
[50] L. Lai, N. Suda, and V. CMSIS-NN Chandra. [n. d.]. Efficient neural network kernels for arm cortex-M CPUs. arXiv 2018. *arXiv preprint arXiv:1801.06601* ([n. d.]).
[51] Yann LeCun, D. Touresky, G. Hinton, and T. Sejnowski. 1988. A theoretical framework for back-propagation. In *Proceedings of the 1988 Connectionist Models Summer School*, Vol. 1. 21–28.
[52] Yoonho Lee, Annie S. Chen, Fahim Tajwar, Ananya Kumar, Huaxiu Yao, Percy Liang, and Chelsea Finn. 2023. Surgical fine-tuning improves adaptation to distribution shifts. In *The Eleventh International Conference on Learning Representations*.
[53] Tao Li, Lei Tan, Zhehao Huang, Qinghua Tao, Yipeng Liu, and Xiaolin Huang. 2022. Low dimensional trajectory hypothesis is true: DNNs can be trained in tiny subspaces. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 45, 3 (2022), 3411–3420.
[54] Ji Lin, Wei-Ming Chen, Yujun Lin, Chuang Gan, and Song Han. 2020. MCUNet: Tiny deep learning on IoT devices. *Advances in Neural Information Processing Systems* 33 (2020), 11711–11722.
[55] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. 2022. On-device training under 256kb memory. *Advances in Neural Information Processing Systems* 35 (2022), 22941–22954.
[56] Sijia Liu, Jie Chen, Pin-Yu Chen, and Alfred Hero. 2018. Zeroth-order online alternating direction method of multipliers: Convergence analysis and applications. In *Proceedings of the International Conference on Artificial Intelligence and Statistics*. PMLR, 288–297.
[57] Sijia Liu, Pin-Yu Chen, Bhavya Kailkhura, Gaoyuan Zhang, Alfred O. Hero III, and Pramod K. Varshney. 2020. A primer on zeroth-order optimization in signal processing and machine learning: Principals, recent advances, and applications. *IEEE Signal Processing Magazine* 37, 5 (2020), 43–54.
[58] 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. *Advances in Neural Information Processing Systems* 36 (2023), 53038–53075.
[59] Sadhika Malladi, Alexander Wettig, Dingli Yu, Danqi Chen, and Sanjeev Arora. 2023. A kernel-based view of language model fine-tuning. In *Proceedings of the International Conference on Machine Learning*. PMLR, 23610–23641.
[60] Charles C. Margossian. 2019. A review of automatic differentiation and its efficient implementation. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 9, 4 (2019), e1305.
[61] George Marsaglia. 2003. Xorshift RNGs. *Journal of Statistical Software* 8 (2003), 1–6.
[62] Pramod Kaushik Mudrakarta, Mark Sandler, Andrey Zhmoginov, and Andrew Howard. 2018. K for the price of 1: Parameter-efficient multi-task and transfer learning. *arXiv preprint arXiv:1810.10703* (2018).
[63] Yurii Nesterov and Vladimir Spokoiny. 2017. Random gradient-free minimization of convex functions. *Foundations of Computational Mathematics* 17, 2 (2017), 527–566.

[64] Maria-Elena Nilsback and Andrew Zisserman. 2008. Automated flower classification over a large number of classes. In *Proceedings of the 2008 6th Indian Conference on Computer Vision, Graphics & Image Processing*. IEEE, 722–729.
[65] Shuaicheng Niu, Chunyan Miao, Guohao Chen, Pengcheng Wu, and Peilin Zhao. 2024. Test-time model adaptation with only forward passes. In *International Conference on Machine Learning*. PMLR, 38298–38315.
[66] Arild Nøkland. 2016. Direct feedback alignment provides learning in deep neural networks. *Advances in Neural Information Processing Systems* 29 (2016).
[67] Alexander Novikov, Dmitrii Podoprikhin, Anton Osokin, and Dmitry P. Vetrov. 2015. Tensorizing neural networks. *Advances in Neural Information Processing Systems* 28 (2015).
[68] Changdae Oh, Hyeji Hwang, Hee-young Lee, YongTaek Lim, Geunyoung Jung, Jiyoung Jung, Hosik Choi, and Kyungwoo Song. 2023. Blackvip: Black-box visual prompting for robust transfer learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 24224–24235.
[69] Yassine Ouali, Adrian Bulat, Brais Matinez, and Georgios Tzimiropoulos. 2023. Black box few-shot adaptation for vision-language models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 15534–15546.
[70] Sunil Pai, Zhanghao Sun, Tyler W. Hughes, Taewon Park, Ben Bartlett, Ian A. D. Williamson, Momchil Minkov, Maziyar Milanizadeh, Nathnael Abebe, Francesco Morichetti, et al. 2023. Experimentally realized in situ backpropagation for deep learning in photonic neural networks. *Science* 380, 6643 (2023), 398–404.
[71] François Panneton and Pierre L'ecuyer. 2005. On the xorshift random number generators. *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 15, 4 (2005), 346–361.
[72] Omkar M. Parkhi, Andrea Vedaldi, Andrew Zisserman, and C. V. Jawahar. 2012. Cats and dogs. In *Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 3498–3505.
[73] Ruizhong Qiu and Hanghang Tong. 2024. Gradient compressed sensing: A query-efficient gradient estimator for high-dimensional zeroth-order optimization. In *Proceedings of the 41st International Conference on Machine Learning*. 41717–41748.
[74] John Rachwan, Daniel Zügner, Bertrand Charpentier, Simon Geisler, Morgane Ayle, and Stephan Günnemann. 2022. Winning the lottery ahead of time: Efficient early network pruning. In *Proceedings of the International Conference on Machine Learning*. PMLR, 18293–18309.
[75] Haoyu Ren, Darko Anicic, and Thomas A. Runkler. 2021. Tinyol: Tinyml with online-learning on microcontrollers. In *Proceedings of the 2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.
[76] Mengye Ren, Simon Kornblith, Renjie Liao, and Geoffrey Hinton. 2022. Scaling forward gradient with local losses. In *The Eleventh International Conference on Learning Representations*.
[77] Tao Ren, Zishi Zhang, Jinyang Jiang, Guanghao Li, Zeliang Zhang, Mingqian Feng, and Yijie Peng. 2024. FLOPS: Forward learning with optimal sampling. In *The Thirteenth International Conference on Learning Representations*.
[78] Yao Shu, Zhongxiang Dai, Weicong Sng, Arun Verma, Patrick Jaillet, and Bryan Kian Hsiang Low. 2023. Zeroth-order optimization with trajectory-informed derivative estimation. In *Proceedings of the 11th International Conference on Learning Representations*.
[79] James C. Spall. 1992. Multivariate stochastic approximation using a simultaneous perturbation gradient approximation. *IEEE Transactions on Automatic Control* 37, 3 (1992), 332–341.
[80] Yi-Lin Sung, Varun Nair, and Colin A. Raffel. 2021. Training neural networks with fixed sparse masks. *Advances in Neural Information Processing Systems* 34 (2021), 24193–24205.
[81] Yun-Yun Tsai, Pin-Yu Chen, and Tsung-Yi Ho. 2020. Transfer learning without knowing: Reprogramming black-box machine learning models with scarce data and limited resources. In *Proceedings of the International Conference on Machine Learning*. PMLR, 9614–9624.
[82] Chun-Chen Tu, Paishun Ting, Pin-Yu Chen, Sijia Liu, Huan Zhang, Jinfeng Yi, Cho-Jui Hsieh, and Shin-Ming Cheng. 2019. Autozoom: Autoencoder-based zeroth order optimization method for attacking black-box neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 742–749.
[83] Chaoqi Wang, Guodong Zhang, and Roger Grosse. 2020. Picking winning tickets before training by preserving gradient flow. In *International Conference on Learning Representations*.
[84] Dequan Wang, Evan Shelhamer, Shaoteng Liu, Bruno Olshausen, and Trevor Darrell. 2021. Tent: Fully test-time adaptation by entropy minimization. In *International Conference on Learning Representations*.
[85] Qin Wang, Olga Fink, Luc Van Gool, and Dengxin Dai. 2022. Continual test-time domain adaptation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 7201–7211.
[86] Zige Wang, Yonggang Zhang, Zhen Fang, Long Lan, Wenjing Yang, and Bo Han. 2023. SODA: Robust training of test-time data adaptors. *Advances in Neural Information Processing Systems* 36 (2023), 44017–44038.
[87] Peter Welinder, Steve Branson, Takeshi Mita, Catherine Wah, Florian Schroff, Serge Belongie, and Pietro Perona. 2010. Caltech-UCSD birds 200. (2010).
[88] Logan G. Wright, Tatsuhiro Onodera, Martin M. Stein, Tianyu Wang, Darren T. Schachter, Zoey Hu, and Peter L. McMahon. 2022. Deep physical neural networks trained with backpropagation. *Nature* 601, 7894 (2022), 549–555.

[89] 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. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. 977–995.

[90] Zi Yang, Samridhi Choudhary, Xinfeng Xie, Cao Gao, Siegfried Kunzmann, and Zheng Zhang. 2024. CoMERA: Computing-and memory-efficient training via rank-adaptive tensor optimization. *Advances in Neural Information Processing Systems* 37 (2024), 77200–77225.

[91] Haishan Ye, Zhichao Huang, Cong Fang, Chris Junchi Li, and Tong Zhang. 2018. Hessian-aware zeroth-order optimization for black-box adversarial attack. *arXiv preprint arXiv:1812.11377* (2018).

[92] Xinling Yu, Sean Hooten, Ziyue Liu, Yequan Zhao, Marco Fiorentino, Thomas Van Vaerenbergh, and Zheng Zhang. 2024. Separable operator networks. *Transactions on Machine Learning Research* (2024).

[93] Kaiqi Zhang, Cole Hawkins, Xiyuan Zhang, Cong Hao, and Zheng Zhang. 2021. On-FPGA training with ultra memory reduction: A low-precision tensor method. *arXiv preprint arXiv:2104.03420* (2021).

[94] Yihua Zhang, Pingzhi Li, Junyuan Hong, Jiaxiang Li, Yimeng Zhang, Wenqing Zheng, Pin-Yu Chen, Jason D. Lee, Wotao Yin, Mingyi Hong, et al. 2024. Revisiting zeroth-order optimization for memory-efficient LLM fine-tuning: A benchmark. In *International Conference on Machine Learning*. PMLR, 59173–59190.

[95] Yihua Zhang, Yuguang Yao, Parikshit Ram, Pu Zhao, Tianlong Chen, Mingyi Hong, Yanzhi Wang, and Sijia Liu. 2022. Advancing model pruning via bi-level optimization. *Advances in Neural Information Processing Systems* 35 (2022), 18309–18326.

[96] Yequan Zhao, Xian Xiao, Xinling Yu, Ziyue Liu, Zhixiong Chen, Geza Kurczveil, Raymond G. Beausoleil, and Zheng Zhang. 2023. Real-Time FJ/MAC PDE solvers via tensorized, back-propagation-free optical PINN training. *arXiv preprint arXiv:2401.00413* (2023).

[97] Yequan Zhao, Xinling Yu, Zhixiong Chen, Ziyue Liu, Sijia Liu, and Zheng Zhang. 2023. Tensor-compressed back-propagation-free training for (physics-informed) neural networks. *arXiv preprint arXiv:2308.09858* (2023).

[98] Feng Zhu, Ruihao Gong, Fengwei Yu, Xianglong Liu, Yanfei Wang, Zhelong Li, Xiuqi Yang, and Junjie Yan. 2020. Towards unified int8 training for convolutional neural network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 1969–1979.

[99] Ligeng Zhu, Lanxiang Hu, Ji Lin, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. 2023. Pockengine: Sparse and efficient fine-tuning in a pocket. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 1381–1394.

Appendices

A Pseudo-Algorithm of Memory-Efficient Layer-Wise Weight/Node Perturbation

ALGORITHM 1: Memory-Efficient Layer-Wise Weight/Node Perturbation

Require: Loss function $\ell(\cdot)$, training dataset $\mathcal{X}$, batch size N, total iterations T, learning rate schedule η_t, trainable layers $\{f^{(i)}(\mathbf{a}^{(i)}, \boldsymbol{\theta}^{(i)})\}_{i=0}^{L}$

for $t \leftarrow 0 \cdots T-1$ **do**
  Mini-batch training data samples $\mathbf{x} : \{\mathbf{x}_n\}_{n=1}^{N} \in \mathcal{X}$
  $\bar{\mathbf{a}}_0 = \mathbf{x}$
  $\boldsymbol{l} \leftarrow \ell(\boldsymbol{\theta}_t; \mathbf{x})$ ▷ Clean Forward
  for $i \leftarrow 0 \cdots L-1$ **do**
    $\bar{\mathbf{a}}^{(i+1)} = f^{(i)}(\bar{\mathbf{a}}^{(i)}, \bar{\boldsymbol{\theta}}^{(i)})$
    if $d_W < d_a$ **then** ▷ Weight Perturbation
      Sample random seed S
      for $q \leftarrow 0 \cdots Q-1$ **do**
        $\boldsymbol{l}_q \leftarrow \ell(\boldsymbol{\theta}_t^{(i)} + \mu\boldsymbol{\xi}_q; \bar{\mathbf{a}}^{(i)})$ ▷ Generate $\bar{\boldsymbol{\xi}}_q$ with S, in-place addition
      end for
      Reset random number generator with seed S
      for $q \leftarrow 0 \cdots Q-1$ **do**
        Re-generate $\bar{\boldsymbol{\xi}}_n$ with S
        $\hat{\nabla}_{\bar{\theta}_i} \leftarrow \sum_q (\boldsymbol{l}_q - \boldsymbol{l})\boldsymbol{\xi}_q$
      end for
    else ▷ Node Perturbation
      Sample random seed S
      for $q \leftarrow 0 \cdots Q-1$ **do**
        for $n \leftarrow 0 \cdots N-1$ **do**
          $\bar{\mathbf{a}}_n^{(i+1)} \leftarrow \bar{\mathbf{a}}_n^{(i+1)} + \mu\boldsymbol{\xi}_{q,n}$ ▷ Generate $\bar{\boldsymbol{\xi}}_n$ with S, in-place addition
        end for
        $\boldsymbol{l}_{q,n} \leftarrow \ell(\boldsymbol{\theta}^t; \bar{\mathbf{a}}_n^{(i+1)})$ ▷ Partial Forward
      end for
      Reset random number generator with seed S
      for $q \leftarrow 0 \cdots Q-1$ **do**
        for $n \leftarrow 0 \cdots N-1$ **do**
          Re-generate $\bar{\boldsymbol{\xi}}_n$ with S ▷ Reuse $\bar{\mathbf{a}}^{(i+1)}$ to store $\hat{\nabla}_{\bar{\mathbf{a}}^{(i+1)}}\ell$
          $\hat{\nabla}_{\bar{\theta}_i}\ell \leftarrow \hat{\nabla}_{\bar{\theta}_i}\ell + (\bar{\mathbf{a}}_n^{(i)})^T\boldsymbol{\xi}_{q,n}(\boldsymbol{l}_{q,n} - \boldsymbol{l}_n)/\mu$
        end for
      end for
    end if
    $\boldsymbol{\theta}_{t+1}^{(i)} \leftarrow \boldsymbol{\theta}_t^{(i)} - \eta_t\hat{\nabla}_{\bar{\boldsymbol{\theta}}^{(i)}}\ell$
    $\bar{\mathbf{a}}^{(i+1)} \leftarrow f^{(i)}(\bar{\mathbf{a}}^{(i)}, \boldsymbol{\theta}_t^{(i)})$ ▷ Recover activation at step t
  end for
end for

B Quantization-Aware Scaling [55]

The update rule of quantized parameter $\bar{\theta}$ with a specific learning rate $\eta_{\bar{\theta}}$:

$$\bar{\theta}_{t+1} = \text{clip}\left(\lceil \bar{\theta}_t - \eta_{\bar{\theta}} \hat{\nabla}_{\bar{\theta}} \mathcal{L}(\bar{\theta}, s_\theta) \rfloor\right) \tag{11}$$

The update rule of full-precision parameter θ with a global learning rate η:

$$\begin{aligned} \theta_{t+1} &= \theta_t - \eta \hat{\nabla}_\theta \mathcal{L} \\ &= s_\theta \left(\frac{\theta_t}{s_\theta} - \eta \frac{\hat{\nabla}_\theta \mathcal{L}(\theta)}{s_\theta} \right) \\ \bar{\theta}_{t+1} \approx \theta_{t+1}/s_\theta &= \left(\bar{\theta}_t - \frac{\eta}{s_\theta} \hat{\nabla}_\theta \mathcal{L}(\theta) \right) \\ \bar{\theta}_{t+1} \approx \theta_{t+1}/s_\theta &= \left(\bar{\theta}_t - \frac{\eta}{s_\theta^2} \hat{\nabla}_{\bar{\theta}} \mathcal{L}(\bar{\theta}, s_\theta) \right) \end{aligned} \tag{12}$$

Given $\hat{\nabla}_\theta \mathcal{L}(\theta)$, update θ with a global learning rate η is equivalent to update $\bar{\theta}$ with a learning rate η/s_θ, neglecting the rounding error of clip.

Scaling the learning rate is equivalent to scale the gradient $\hat{\nabla}_{\bar{\theta}} \mathcal{L}(\bar{\theta}, s_\theta)$ by s_θ^{-2} at each step so as to unify the hyperparameter η as a global learning rate for all layers. The update of real-quantized parameters with different s_θ follows the update of full-precision parameters.

C Error Analysis and Tradeoffs for Accuracy Versus Efficiency

Figure 8 demonstrates the accuracy of quantized ZO gradient estimation across different network layers. We evaluate the cosine similarity between ZO and FO gradients for varying numbers of forward passes (Q). As predicted by Equation (7), larger Q values yield gradient estimates with smaller MSE/variance, indicating larger cosine similarity to FO gradients. While this improved estimation theoretically accelerates convergence, it incurs additional computational overhead per iteration. Figure 9 illustrates this tradeoff by plotting validation accuracy against the total number of forward passes. Notably, different Q values achieve comparable validation accuracy given the same computational budgets, with larger Q values showing marginally superior performance. Based on these empirical results, we adopt $Q = 100$ in subsequent experiments to ensure stable BP-free training.

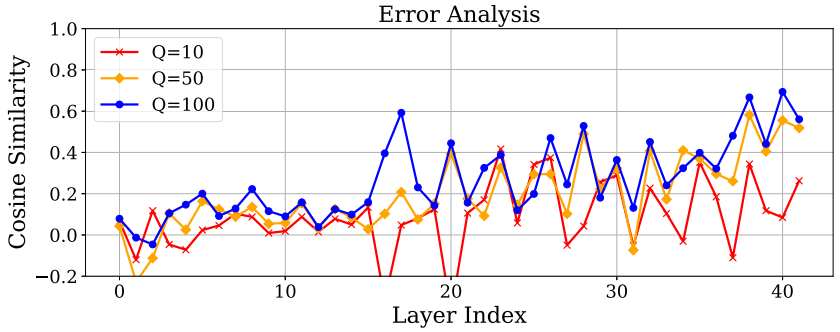

Fig. 8. Error analysis for quantized ZO gradient estimation.

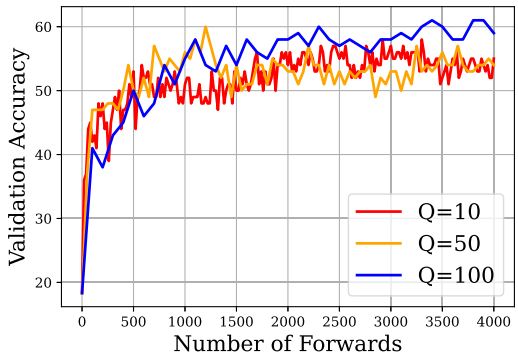

Fig. 9. Validation accuracy against the total number of forward passes.

D Algorithmic Performance Comparison over Multiple Datasets

We extensively evaluated the effectiveness of various ZO optimization improvement techniques over multiple datasets, including all four noise types on the CIFAR-10-C dataset (noise, blur, weather, digital), Pets dataset, and VWW dataset. The trainable parameters are the weight matrices and biases of four point-wise convolution layers in block 1 for CIFAR-10-C datasets, block 2 for the Pets dataset, and block 3 for the VWW dataset.

The conclusions we observed in Section 4.2.1 are consistent over multiple datasets. The training curves with and without learning rate scaling are provided in Figure 10. With both scaling methods applied, our BP-free training follows a similar training dynamics as full-precision BP-based training without using any extra memory. The training curves of different perturbation methods are provided in Figure 11. Our layer-wise gradient estimation with adaptive WP/NP outperforms all other methods and achieves the best training convergence.

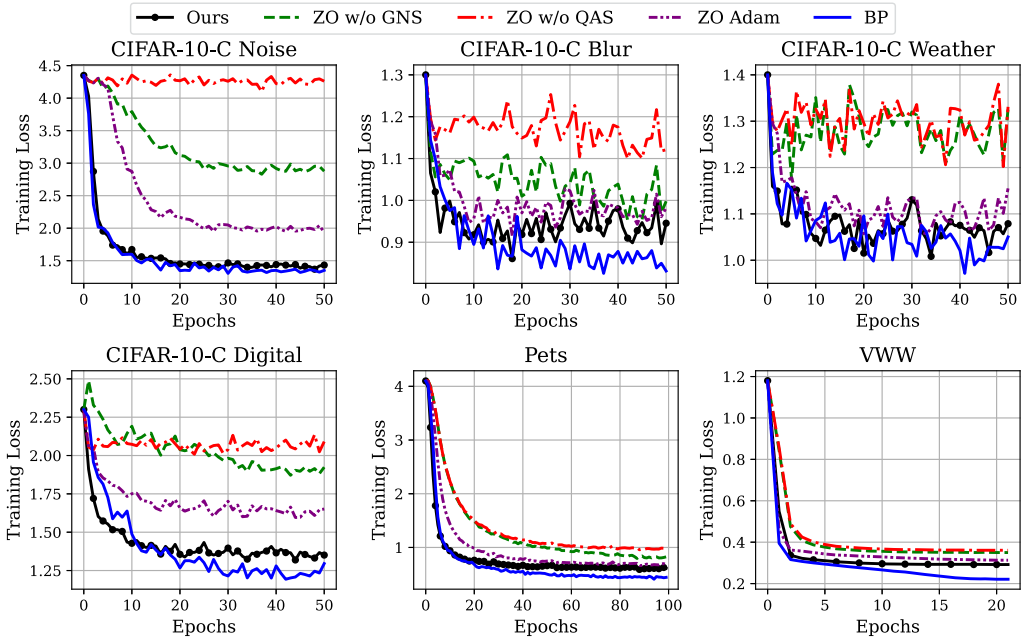

Fig. 10. Training loss curves w/ and w/o learning-rate scaling applied. QAS: quantization-aware scaling. GNS: gradient-norm scaling.

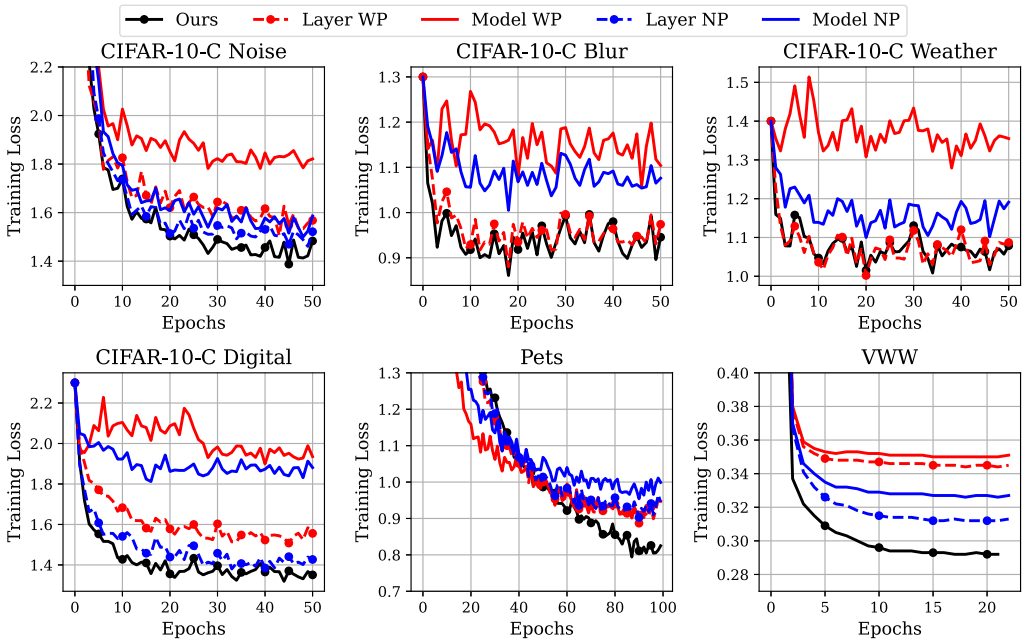

Fig. 11. Training loss curves with different ZO gradient estimation methods applied.

E Extended Discussion

In this section, we extend the discussion to show the advantages of our proposed BP-free on-device training method.

E.1 Comparison with Related Lightweight Training Paradigms

Comparison with federated learning: Our method focuses on personalization through local on-device training only. This is particularly advantageous over federated learning when:

— Data privacy is of vital importance and even encrypted model updates cannot leave the device.
— Edge devices working offline without network access, or when network access are unreliable.
— User patterns differ significantly, making global model aggregation less beneficial.
— Real-time adaptation to user behavior is needed.

Our on-device training approach reduces communication overhead and infrastructure costs while enabling faster personalization to individual usage patterns.

Comparison with Model Pruning: Model pruning contributes to efficient training by finding a more lightweight model with fewer trainable parameters. The lottery ticket hypothesis [14, 27, 53] demonstrates the existence of compact trainable representations but requires resource-intensive pre-training phases. Picking winning tickets at initialization [74, 83] provides benefits in terms of GPU memory and training time, while often facing performance degradation compared with unpruned training. Tensorized training [67, 90, 93] trains a tensor-compressed representation with tens to hundreds of times fewer model parameters end to end. However, it is non-trivial to implement FO tensorized training on edge devices due to the complex gradient computation. Note that model pruning or sparse training can only reduce the memory overhead needed to implement FO training. However, on edge devices with extremely constrained memory (MCUs, FPGAs, etc.), only training the last two blocks could exceed the memory budget [55].

E.2 Discussion of Potential Vulnerabilities

Our framework can tolerate the case where the collected training data is noisy/poisoned, which is one of the motivations of on-device training. We refer the readers to Section 4.3.1 and Table 7, where the training dataset CIFAR-10-C are corrupted images from CIFAR-10 dataset. The corruption covers a wide range of real-life scenarios (noise, blur, weather, etc.), which mimic the potential unknown noises in real-life settings. Through on-device training with our framework, we greatly improve the model performance on these corrupted data, and even surpass BP-based training in certain cases. This is attributed the memory-efficiency nature of BP-free training that enables searching for the best sparse training pattern across the whole model.

F Extented Details of MCU Implementation

In-Place Random Perturbation Generation. We generate the quantized perturbation vectors based on a Rademacher distribution and using the XORShift pseudo-random number generator [61, 71]. XORShift uses bitwise **exclusive OR** (**XOR**) and bit shifts (left and right) to produce a sequence of pseudo-random numbers efficiently in an iterative way:

$$\text{state} \leftarrow \text{state} \oplus (\text{state} \ll a)$$
$$\text{state} \leftarrow \text{state} \oplus (\text{state} \gg b)$$
$$\text{state} \leftarrow \text{state} \oplus (\text{state} \ll c)$$

Here a, b, and c are constants; $\oplus$ denotes the bitwise XOR operator; $\leftarrow$ and $\rightarrow$ denote left and right bit shifts, respectively. These operations ensure sufficient mixing of bits in the state, generating pseudo-random numbers with desirable statistical properties. In practice, the constants are commonly chosen as $a = 13$, $b = 17$, and $c = 5$. We use the following steps to generate a sequence of random numbers following the Rademacher distribution:

(1) **Initialize the seed.** We set the seed for the XORShift generator, which will serve as the initial state.
(2) **Update the state.** We apply the XORShift recurrence relations to the current state to generate a new pseudo-random number.
(3) **Generate Rademacher-distributed samples.** The pseudo-random number is transformed into a Rademacher-distributed value based on the **least significant bit** (**LSB**). If the LSB is 1, return −1; otherwise, return +1.

The random perturbations only need to be generated on-demand and added to the weights/activations in-place. The same random perturbation can be re-generated using the same initial seed. Therefore, we do not need a buffer to temporarily store the random perturbations.

The C++ code snippet below demonstrates how to implement XORShift to generate Rademacher-distributed random numbers:

```
unsigned int xor_seed; // XORShift seed

// Set the XORShift seed
void set_xor_seed(unsigned int s) {
    xor_seed = s;
}

// XORShift pseudo-random number generator
unsigned int xor_rand() {
    xor_seed ^= xor_seed << 13;
    xor_seed ^= xor_seed >> 17;
```

```
    xor_seed ^= xor_seed << 5;
    return xor_seed;
}

// Generate a Rademacher-distributed value
int rademacher() {
    unsigned int rand_num = xor_rand();
    return (rand_num & 1) ? -1 : 1;
}
```

In this implementation, the function `xor_rand()` generates a pseudo-random number using the XORShift algorithm, and the function `rademacher()` converts this number into a Rademacher-distributed value by checking the LSB.

Received 7 November 2024; revised 9 May 2025; accepted 4 June 2025