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

Yequan Zhao* Hai Li† Ian Young† Zheng Zhang*

November 12, 2024

Abstract

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 face multiple major challenges, such as the lack of hardware resources, long time-to-market, and dramatic errors in a low-precision setting. This paper 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.

1 Introduction

On-device training, that is training deep neural networks (DNN) 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) [1] is used in almost all neural network training frameworks for gradient computation. BP is actually a reverse-mode automatic differentiation (AD) [2, 3] 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 are 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

*Department of Electrical and Computer Engineering, UC Santa Barbara, Santa Barbara, California, USA
†Intel Corporation, Hillsboro, Oregon, USA

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 [4]), using real-quantized graphs and a co-designed lightweight back-end (*e.g.,* the TinyEngine [4, 5]), one may perform edge inference with a low memory cost (e.g., 96 KB for the MCUNet-in1 model [4]). 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 [6, 7], bias vectors [8] to reduce the memory cost, yet leads to significant (e.g., ¿10%) accuracy drop. Sparse update [4, 9]) 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 [10] 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 [11, 12].

- **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 automatic differentiation packages [2] 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 [13]. This error-prone process needs numerous debugs and design trade-offs. 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.

Paper 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 paper, 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 zeroth-order 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 [14] or coordinate-wise gradient estimation [15] 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 and node

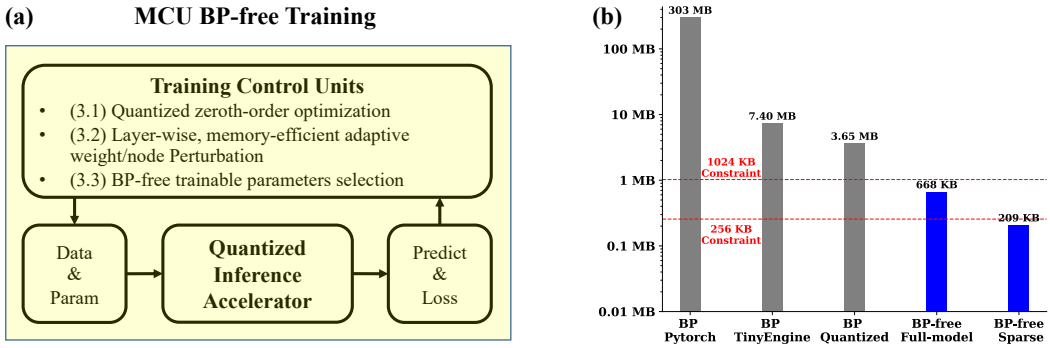

Figure 1: (a): Overview of BP-free training framework. A quantized inference engine is easily converted to a training engine by adding control unit and repeatedly calling the inference accelerator. (b): Training memory comparison of different training methods. The numbers are measured with MCUNet-in1 [4], batch size 1, and resolution 128× 128.

perturbation 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 image corruptions, our BP-free training outperforms current quantized BP-based training with an average 6.4% test accuracy improvement. Our method can also match the performance of back-propagation training on fine-grained vision classification datasets.

Our Key idea is summarized in Fig. 1 (a). As demonstrated in Fig. 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 $> 400\times$ 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 [5] 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 [16]). On resource-constrained edge devices, *real-quantized* graphs are usually leveraged to achieve memory and computation efficiency.

In this paper, 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 [17] 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.

Table 1: Comparison of different quantized training paradigms.

	Model	Forward	Backward
Quantization-aware Training (QAT) [17]	FP	INT	FP
Fully-quantized Training (FQT) [12]	FP	INT	INT
Real-quantized Training (RQT) [5]	INT	INT	INT

2.2 Extra Memory and Computation Cost of Back-propagation

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

We consider a generic neural network with L layers:

$$f(\mathbf{x};\boldsymbol{\theta}) = f^{(L)}\left(f^{(L-1)}\left(\dots f^{(0)}\left(\mathbf{x};\boldsymbol{\theta}^{(0)}\right)\dots;\boldsymbol{\theta}^{(L-1)}\right);\boldsymbol{\theta}^{(L)}\right). \tag{2}$$

Here $\boldsymbol{\theta}^{(i)}$ denotes the parameters of the i-th layer. The training process aims to find the optimal set of $\{\boldsymbol{\theta}^{*(i)}\}_{i=1}^{L}$ by minimizing an objective (loss) function $\mathcal{L}(f(\mathbf{x};\boldsymbol{\theta}), y)$. Here $f^{(i)}(\mathbf{a}^{(i)};\boldsymbol{\theta}^{(i)})$ predicts the activation values $\mathbf{a}^{(i+1)}$ of the next layer based on its inputs $\mathbf{a}^{(i)}$. For instance, in a fully connected layer $\boldsymbol{\theta}^{(i)} = \{\mathbf{W}^{(i)};\mathbf{b}^{(i)}\}$, and $f^{(i)}$ performs the calculation

$$\mathbf{a}^{(i+1)} = h^{(i)}(\mathbf{z}^{(i)}), \quad \mathbf{z}^{(i)} = \mathbf{W}^{(i)}\mathbf{a}^{(i)} + \mathbf{b}^{(i)} \tag{3}$$

where $h^{(i)}$ is the nonlinear activation function. The memory cost of an inference task consists of two parts: (1) the non-volatile storage (*e.g.,* Flash) that stores the pre-trained model parameters $\{\boldsymbol{\theta}^{(i)} \in \mathbb{R}^{d_{\theta^{(i)}}}\}_{i=1}^{L}$, (2) run-time volatile memory (*e.g.,* SRAM) that stores the intermediate computation results. The peak run-time memory at layer i is approximately the summation $\mathbf{a}^{(i)}$, $\boldsymbol{\theta}^{(i)}$, and $\mathbf{a}^{(i+1)}$.

Extra Memory of BP. In the BP process, given the gradients of the output activation $\nabla_{\mathbf{a}^{(i+1)}}\mathcal{L}$ at i-th layer, one needs to first back-propagate through the non-linear activation function:

$$\nabla_{\mathbf{z}^{(i)}}\mathcal{L} = h'(\mathbf{z}^{(i)})\nabla_{\mathbf{a}^{(i+1)}}\mathcal{L} \tag{4}$$

Then back-propagate through the linear transformation to compute the gradients of the input activation $\nabla_{\mathbf{a}^{(i)}}\mathcal{L}$ and the gradients of the parameters $\nabla_{\boldsymbol{\theta}^{(i)}}\mathcal{L}$:

$$\nabla_{\mathbf{a}^{(i)}}\mathcal{L} = {\mathbf{W}^{(i)}}^{T} \cdot \nabla_{\mathbf{z}^{(i)}}\mathcal{L}, \quad \nabla_{\mathbf{W}^{(i)}}\mathcal{L} = \nabla_{\mathbf{z}^{(i)}}\mathcal{L} \cdot (\mathbf{a}^{(i)})^{T}, \quad \nabla_{\mathbf{b}^{(i)}}\mathcal{L} = \nabla_{\mathbf{z}^{(i)}}\mathcal{L} \tag{5}$$

The result $\nabla_{\boldsymbol{\theta}^{(i)}}\mathcal{L} = \{\nabla_{\mathbf{W}^{(i)}}\mathcal{L}, \nabla_{\mathbf{b}^{(i)}}\mathcal{L}\}$ is saved to update model parameters, and $\nabla_{\mathbf{a}^{(i)}}\mathcal{L}$ is propagated to the (i-1)-th layer. Throughout this paper, we use the term **training memory** to denote run-time memory on an edge device. Apart from the memory to store the updated model parameters, we need the following additional memory in BP:

- **Intermediate activation values** of all layers $\{\mathbf{a}^{(i)} \in \mathbb{R}^{B\times d_{a_i}}\}_{i=1}^{L}$ and $\{\mathbf{z}^{(i)} \in \mathbb{R}^{B\times d_{a_i}}\}_{i=1}^{L}$. Here B denotes batch size and d_{a_i} denotes the number of neurons in i-th layer.
- **Gradients** of model parameters $\{\nabla_{\boldsymbol{\theta}_i}\boldsymbol{L} \in \mathbb{R}^{d_{\theta_i}}\}_{i=1}^{L}$. Here d_{θ_i} denotes the dimension of parameters in i-th layer;
- **Optimizer state** (optional). Vanilla SGD [18] does not cost additional memory for the optimizer state. SGD with momentum and Adam optimizer [19] costs 1× and 2× the size of all trainable parameters, respectively.

Previous on-device training frameworks explored techniques to alleviate the training memory bottleneck. For example, in convolutional neural networks, $\mathbf{a}^{(i)}$ need not be stored if the parameters in i-th layer are frozen as $\mathbf{a}^{(i)}$ is only involved in computing $\nabla_{\boldsymbol{\theta}^{(i)}}\mathcal{L}$ [8]. For specific non-linear layers (*e.g.,* ReLU [20] and other ReLU-styled), memory for $\mathbf{z}^{(i)}$ can be reduced by storing a binary mask representing whether the value is smaller than 0 [8]. However, such reductions are model-specific. Many operations (e.g., self-attention) involve $\mathbf{a}^{(i)}$ and many non-linear activations (*e.g.,* GeLU [21], Softmax, etc.) involve $\mathbf{z}^{(i)}$ in back-propagation, necessitating additional memory. The memory-consuming nature of BP can only be alleviated, but not fully addressed. The inevitable extra memory to implement BP often prevents full-model training on resource-constrained edge devices.

Extra Computation Graph. The layer-by-layer computation in Eq. (5) needs the knowledge of a computation graph. On cloud servers or personal computers with sufficient hardware resources and sophisticated operating system support, an AD package [22] could automatically generate the computation graph. However, on edge devices running without an operating system (*e.g.,* microcontrollers, FPGAs, ASICs, etc.), the computation graph and the corresponding hardware of Eq. (5) need to be hand-crafted and optimized for different models, tasks, and devices. This inevitably increase the design and manufacturing cost as well as the time to market.

2.3 Zeroth-order Optimization

As shown in Fig. 2, zeroth-order (ZO) optimization [23] 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) [24, 15, 14] 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:*

$$\hat{\nabla}_{\boldsymbol{\theta}} F(\boldsymbol{\theta}) = \frac{1}{Q}\sum_{i=1}^{Q} \frac{[F(\boldsymbol{\theta}+\mu\boldsymbol{\xi}_i) - F(\boldsymbol{\theta})]}{\mu}\boldsymbol{\xi}_i. \tag{6}$$

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 $\nabla_{\boldsymbol{\theta}} F(\boldsymbol{\theta})$ [25, 15].

We utilize the corresponding stochastic gradient descent (SGD) [18] algorithm, ZO-SGD [26, 27] to update model parameters in the training process.

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

$$\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \eta\hat{\nabla}_{\boldsymbol{\theta}} F(\boldsymbol{\theta}) \tag{7}$$

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. (7)] 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

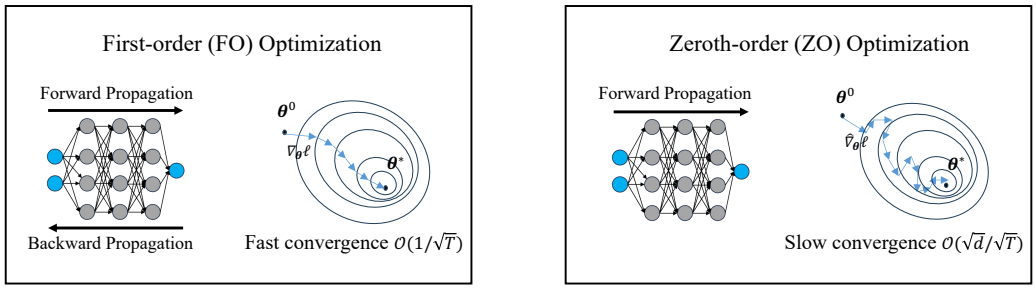

Figure 2: Comparison between first-order (FO) optimization and ZO optimization. FO optimization converges faster as it utilizes exact gradient from BP to update model parameters. ZO optimization, on the other hand, uses only forward function queries to estimate the gradients. ZO method converges more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

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 weight perturbation and node perturbation, 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 $\bar{\boldsymbol{\theta}} = \{\bar{\mathbf{W}}, \bar{\mathbf{b}}\}$ and input/output activation values $\bar{\mathbf{a}}$. Let $\mathcal{L}(\boldsymbol{\theta}; \mathcal{X})$ be the empirical loss function, but we can only access via a sample-wise loss $\ell(\bar{\boldsymbol{\theta}}; \mathbf{x})$ evaluated at a data sample $\mathbf{x} \subset \mathcal{X}$. We estimate the gradient of quantized parameters $\nabla_{\bar{\boldsymbol{\theta}}_i}\mathcal{L}$ 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:

$$\hat{\nabla}_{\bar{\boldsymbol{\theta}}}\mathcal{L} = \frac{1}{NQ}\sum_{n=1}^{N}\sum_{q=1}^{Q}\frac{\left[\ell\left(\bar{\boldsymbol{\theta}} + \mu\boldsymbol{\xi}_{n,q}; \mathbf{x}_n\right) - \ell(\bar{\boldsymbol{\theta}}; \mathbf{x}_n)\right]}{\mu}\boldsymbol{\xi}_{n,q}, \mathbf{x}_n \subset \mathcal{X} \tag{8}$$

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 $\{\{\boldsymbol{\xi}_q\}_{q=1}^Q\}_n$ to each training data sample $\mathbf{x}_n$. We directly add perturbation to the quantized model parameters $\bar{\boldsymbol{\theta}} \in \mathbb{R}^d$ with discrete integer values (*e.g.,* $\{\bar{\boldsymbol{\theta}} \in \mathbb{Z} | -128 \leq \bar{\boldsymbol{\theta}} \leq 127\}$ for signed INT8 format). To ensure that the perturbed values are still in integer format, we sample the perturbation $\boldsymbol{\xi}_q$ 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 Eq. (8) is an unbiased gradient estimator as $\mu \to 0$ [28].

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 [25, 29]. The MSE of gradient estimation based on Eq. (8) derived by [25] is given as:

$$\mathbb{E}\left[\|\hat{\nabla}_{\boldsymbol{\theta}}\mathcal{L} - \nabla_{\boldsymbol{\theta}}\mathcal{L}\|_2^2\right] = \frac{d-1}{NQ}\|\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})\|_2^2 + \frac{d}{NQ}\operatorname{tr}\left(\operatorname{Var}_{\mathbf{x}}\left[\nabla_{\boldsymbol{\theta}}\ell(\boldsymbol{\theta}, \mathbf{x})\right]\right) + \mathcal{O}(\mu^2 d^2) \tag{9}$$

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 [25].

3.1.2 Learning-rate Scaling

Directly updating quantized model parameters with the gradient estimator $\hat{\nabla}_{\bar{\boldsymbol{\theta}}}\mathcal{L}$ in Eq. (8) 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. Lemma 3.1) derived by [14] shows how the gradient norm influences the performance of an SGD optimizer.

  Lemma 3.1 (Descent Lemma [14])**.** *Let $\mathcal{L}$ be l-smooth. For any unbiased gradient estimate $\hat{\nabla}\mathcal{L}(\boldsymbol{\theta})$*

  $$\mathbb{E}\left[\mathcal{L}\left(\boldsymbol{\theta}_{t+1}\right) \mid \boldsymbol{\theta}_t\right] - \mathcal{L}\left(\boldsymbol{\theta}_t\right) \leq -\eta \left\|\nabla \mathcal{L}\left(\boldsymbol{\theta}_t\right)\right\|^2 + \frac{1}{2}\eta^2 l \cdot \mathbb{E}\left[\left\|\hat{\boldsymbol{g}}\left(\boldsymbol{\theta}\right)\right\|^2\right] \tag{10}$$

  Lemma 3.1 indicates that the largest permissible learning rate of ZO-SGD should be $\mathbb{E}\left[\|\hat{\nabla}\mathcal{L}_{\text{ZO-SGD}}\|^2\right] / \mathbb{E}\left[\|\hat{\nabla}\mathcal{L}_{\text{SGD}}\|^2\right.$ times smaller than that of SGD to guarantee loss decrease. According to Eq. (9), the squared norm of the ZO stochastic gradient estimation is approximately $(NQ + d - 1)/NQ$ times larger than that of the FO stochastic gradient [14]. 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 [5].** The quantization process also distorts the SGD update. Let $\boldsymbol{\theta}$ denote the full-precision parameter, and $\bar{\boldsymbol{\theta}} = \boldsymbol{\theta}/s_{\boldsymbol{\theta}}$ denote its quantized (*e.g.,* INT8) and scaled representation. Here $s \ll 1$. $\bar{\boldsymbol{\theta}}$ is $1/s_{\boldsymbol{\theta}}$ times larger than $\boldsymbol{\theta}$ in magnitude, while according to the chains rule, the gradient magnitude of quantized representation $\hat{\nabla}_{\bar{\boldsymbol{\theta}}}\mathcal{L}$ is $s_{\boldsymbol{\theta}}$ times smaller than that of $\hat{\nabla}_{\boldsymbol{\theta}}\mathcal{L}$. To ensure the update of a quantized parameter representation follows the expected update in its original scale, we follow [5] 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 $s_{\boldsymbol{\theta}}$ (c.f. Appendix B for details). Consequently, at step t we update quantized model parameters as:

  $$\bar{\boldsymbol{\theta}}^{t+1} \leftarrow \text{clip}\left(\bar{\boldsymbol{\theta}}^t - \left\lceil \frac{NQ}{NQ + d - 1}\frac{1}{s_{\boldsymbol{\theta}}^2} \cdot \eta \hat{\nabla}_{\bar{\boldsymbol{\theta}}}\mathcal{L} \right\rfloor, -Q_N, Q_P\right) \tag{11}$$

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

The convergence speed of ZO training depends on the gradient error [25, 29], which is dominated by the gradient estimation variance [i.e., the first term in Eq. (9)]. Despite various variance reduction methods for ZO optimization, extra memory is needed to save either the co-variance matrix [30], hessian diagonal [31], or history perturbations [32, 24, 33, 34], making them unsuitable for edge devices with tiny memory budget. Ref. [25] 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.

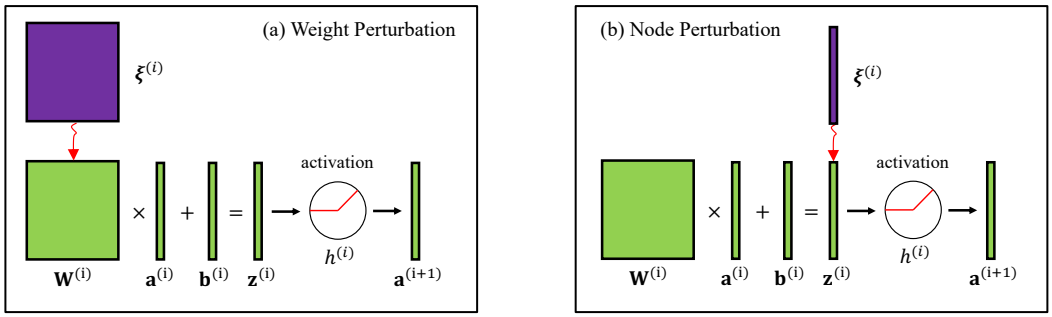

Figure 3: (a) Data flow of weight perturbation. (b) Data flow node perturbation.

3.2 Memory-Efficient Adaptive Weight/Node Perturbation

3.2.1 Background: Dimension Reduction via Node Perturbation

Eq. (8) 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 $\hat{\nabla}_{\mathbf{z}^{(i)}}\mathcal{L}$,

$$\hat{\nabla}_{\mathbf{z}^{(i)}}\mathcal{L} = \frac{1}{NQ}\sum_{n=1}^{N}\sum_{q=1}^{Q}\frac{\ell_q(\bar{\boldsymbol{\theta}};\mathbf{x}_n) - \ell(\bar{\boldsymbol{\theta}};\mathbf{x}_n)}{\mu}\boldsymbol{\xi}_{k,q,n}, \quad \mathbf{x}_n \subset \mathcal{X}. \tag{12}$$

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 Eq. (5), the gradient of the weights and biases is then obtained as [35, 36, 37]:

$$\hat{\nabla}_{\bar{\mathbf{W}}^{(i)}}\mathcal{L} = \hat{\nabla}_{\bar{\mathbf{z}}^{(i)}}\mathcal{L}\cdot(\bar{\mathbf{a}}^{(i)})^T \quad \hat{\nabla}_{\bar{\mathbf{b}}^{(i)}}\mathcal{L} = \hat{\nabla}_{\bar{\mathbf{z}}^{(i)}}\mathcal{L} \tag{13}$$

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

However, the superiority of node perturbation over weight perturbation 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. Fig. 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 weight perturbation or activation perturbation across the whole model does not guarantee a better gradient estimation for all layers.

- **Inter-layer Feature Correlation:** Fig 4 also shows a zig-zag pattern in node perturbation. This is attributed to the simultaneous perturbation and update of all layers in vanilla node perturbation. 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 [37, 36].

- **Memory Inefficiency:** Vanilla implementation of node perturbation 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, weight perturbation needs storing only $2N + 1$ scalars [14].

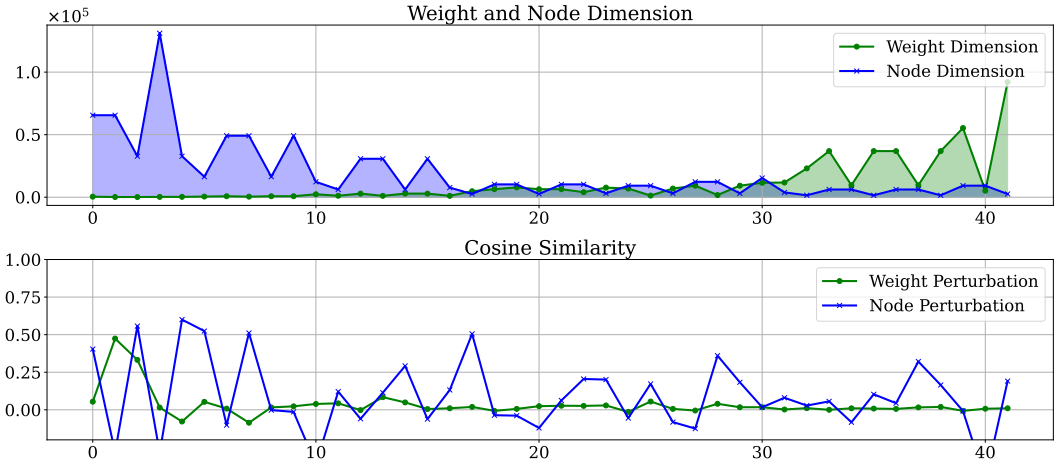

Figure 4: Top: The dimension of weights/nodes at each layer. Bottom: The cosine similarity between zeroth-order gradient estimation and the first-order gradient computed by back-propagation at each layer.

Table 2: Gradient estimation variance comparison between weight perturbation (WP) and node perturbation (NP) with their model-wise and layer-wise gradient estimation implementations. S is the squared gradient norm, and V is the variance of stochastic gradient.

	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	$\frac{Ld_a-1}{NQ}S+\frac{L}{NQ}V$	$\frac{Ld_a-L}{NQ}S+\frac{L}{NQ}V$

3.2.2 Proposed Solution: Layer-Wise Memory-Efficient Adaptive Weight/Node Perturbation.

To fully leverage the benefit of node perturbation 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 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, Fig. 5 shows that layer-wise gradient estimation outperforms model-wise gradient estimation for both weight perturbation and node perturbation with a non-negligible gap. This is attributed to the de-correlation between layers in a layer-wise gradient estimation method [36].

- **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 weight perturbation for layer i if $d_w^{(i)} < d_a^{(i)}$; otherwise, we apply node perturbation. We can consistently get a lower gradient estimation variance regardless of the unbalanced weight and node dimension of different layers, as shown in Fig 5. Note that the dimension d in Eq. (11) is set as $d = d_w^{(i)}$ for layer-wise weight perturbation, and $d = d_a^{(i)}$ for layer-wise node perturbation,

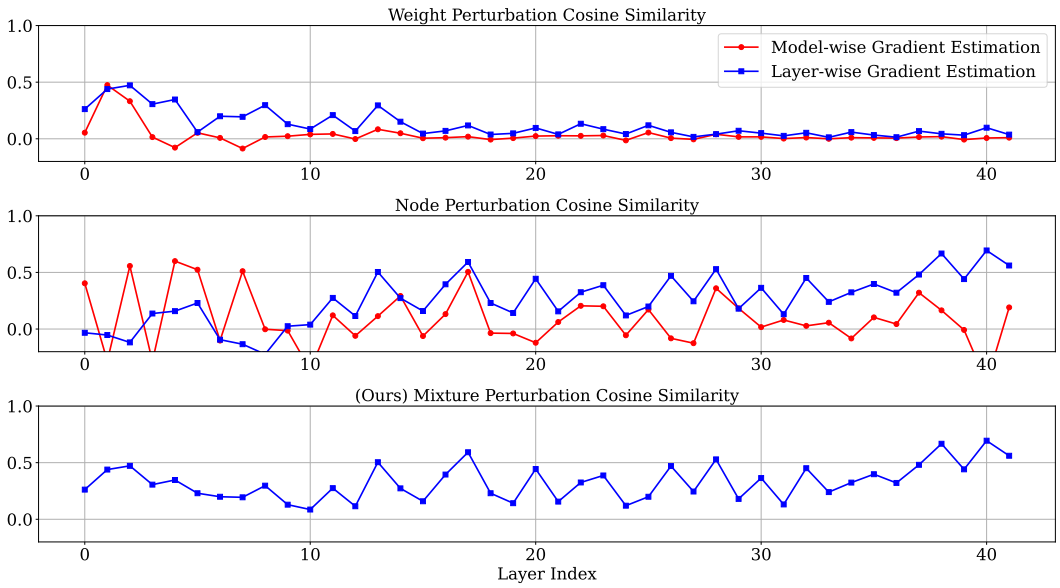

Figure 5: Quality of gradient estimation (measured via Cosine similarity) via weight perturbation (top), node perturbation (middle) and our mixture of weight and node perturbation (bottom).

Table 3: Memory cost comparison. FWD denotes forward propagation, and BWD denotes backward propagation.

	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)	$NLd_a + Ld_w$	/

accordingly.

- **Memory-efficient Layer-wise Gradient Estimation.** We propose a memory-efficient implementation of node perturbation that minimizes the memory overhead. Different from BP, we only need $\mathbf{a}_i$ for the gradient estimation of i-th layer in node perturbation. 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 i-th layer. In this way, $\mathbf{a}_i$ needs not be temporarily stored. Inspired by [14], 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 [38] where different blocks (layers) are updated sequentially. The theoretical study [39] shows that local updates can generalize worse than global updates. The peak training memory comparison between inference-only, BP-based training, memory-efficient weight perturbation, and our memory-efficient node perturbation is provided in Table 3. The peak memory of our method is slightly larger than that of memory-efficient weight perturbation 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 [40, 5, 9], 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 [5, 41, 42] or evaluating gradient-based "importance" metric [9, 43, 44] 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) [15]. 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 first step, we employ a brute-force, yet effective BP-free selection metric. We follow the surgical fine-tuning settings in [40] to split the model into four non-overlapping blocks. We denote them as "Block 1", "Block 2", etc. 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.

Fig. 6 shows the overall training framework. A few key points are summarized below:

- **Quantized Inference Engine.** We employ TinyEngine [5, 45] to deploy tailored neural network model on the MCU. As an example, we consider the tailored CNN model MCUNet-in1 [4] with 0.46M parameters. The model parameter memory usage is 478 KB, and the peak inference runtime memory (SRAM) usage is 190 KB. The inference latency is measured as 239 ms using a single batch size with input resolution 128×128.

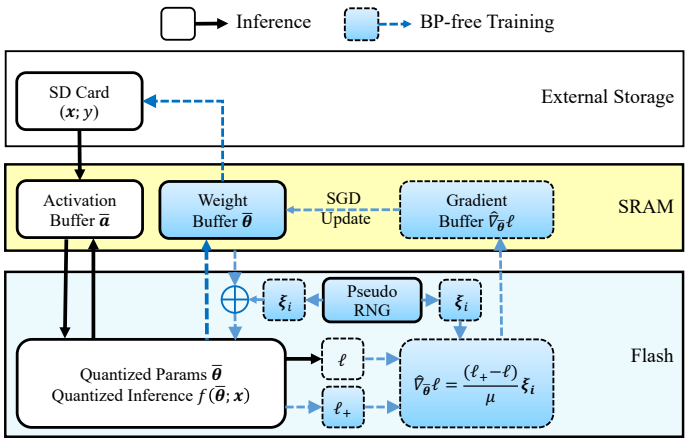

Figure 6: Overview of BP-free training framework on MCU.

- **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.
- **Gradient Estimation and Parameter Update.** We deploy the memory-efficient layer-wise gradient estimation with adaptive weight/node perturbation. The step-by-step implementation is summarized in Algorithm 1.

3.4.2 In-Place Random Perturbation Generation

We generate the quantized perturbation vectors based on a Rademacher distribution and using the XORShift pseudo-random number generator [46, 47]. 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 perturabtion can be re-generated using the same initial seed. Therefore, we do not need a buffer to temporarily store the random perturbations.

4 Experimental Results

4.1 Experiment Setup

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

- **Image Corruption.** We evaluate our BP-free training framework on a widely used out-of-distribution image corruption benchmarks **CIFAR-10-C** [48]. 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 5 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* [5] and *TinyTrain* [9], we also test our BP-free training framework on several Fine-grained Vision Classification (FGVC) datasets including Cars [49], CUB [50], Flowers [51], Food [52], and Pets [53]. The training datasets contain approximately 30-50 samples of each class.

Model Configurations: We employ the MCU-compatible convolutional neural network (CNN) model *MCUNet-in1* [4] with 22.5M MACs and 0.46M parameters. The model is pre-trained on ImageNet-1k [54] and quantized to the INT8 format by post-training quantization [17]. The batch normalization [55] 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 paper are in the INT8 format.

Training Configurations: We use a resolution of 128×128 following [5] and gradient accumulation with 100 steps for all datasets, models, and baselines for a fair comparison. We apply vanilla SGD [18] / ZO-SGD [26] 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 [56] 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 and a cosine decay following the setups in [40]. For experiments on FGVC datasets, we train the model for 50 epochs following [5] in both BP-based and our BP-free methods. The initial learning rate is set as 0.1, and a cosine decay method is used. 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 [55]), 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 node perturbation, i.i.d. perturbations are simultaneously applied to the pre-activation vector in a mini-batch at each layer. For weight perturbation, 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.

- **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

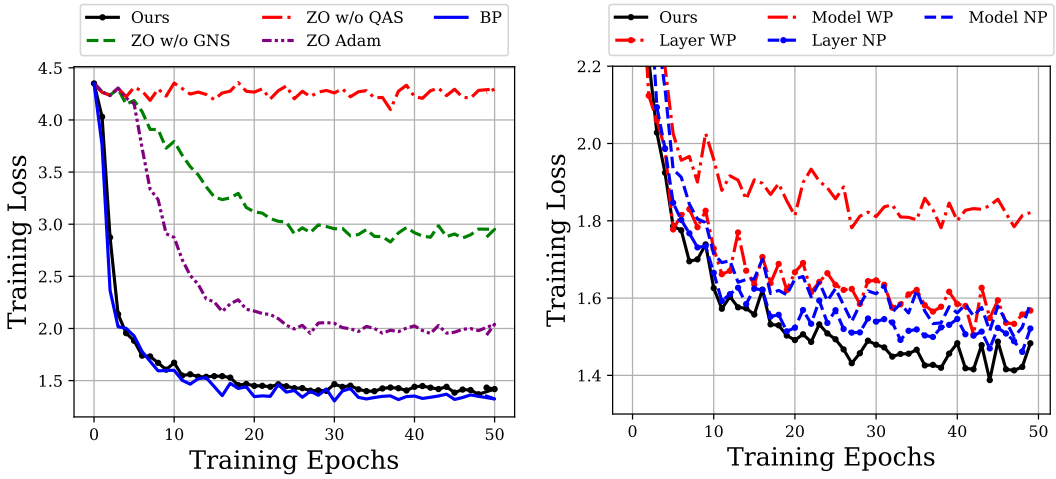

Figure 7: Left: Training loss curves w/ and w/o learning-rate scaling applied. QAS: quantization-aware scaling. GNS: gradient-norm scaling. Right: Training loss curves with different zeroth-order gradient estimation methods applied. We select the best learning rate for each method.

gradient estimation. Without gradient-norm scaling or quantization-aware scaling, the training converges slowly or even cannot converge. Adaptive learning rate methods like Adam [19] 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 weight and node perturbations, showing that dis-entangling the gradient estimation layer-by-layer improves convergence. Our layer-wise gradient estimation with adaptive weight/node perturbation 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.

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 state-of-the-art BP-free training baselines.

	iter. forwards	epochs	total forwards	test accuracy (%)
MeZO [14]	2	10000	**2.00E+07**	18.33
DeepZero [15]	49457	50	2.47E+09	52.78
Ours	400	50	**2.00E+07**	**66.78**

4.2.2 Performance Comparison with Other BP-free Training Methods

Next, we compare our proposed methods with two state-of-the-art BP-free training baselines: *MeZO* [14] and *DeepZero* [15]. *MeZO* [14] uses RGE with memory-efficient model-wise weight perturbation as the gradient estimator. All trainable parameters are perturbed and updated at the same time. *DeepZero* [15] uses 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. The main results are summarized below:

- **MeZO [14].** 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 weight perturbations, and 3) the general behaviour 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 [15].** *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.

Table 6: Comparison between our BP-free training and other state-of-the-art BP-free training baselines when applied the same trainable parameter selection. DeepZero-0 denotes dense training, and DeepZero-0.9 means training with a 90% gradient sparsity.

	iter. forwards	epochs	total forwards	test accuracy (%)
MeZO [14]	2	10000	**2.00E+07**	40.56
DeepZero-0 [15]	18560	50	9.28E+08	60.67
DeepZero-0.9 [15]	1856	50	9.28E+07	54.33
Ours	400	50	**2.00E+07**	**66.78**

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 back-propagation training baselines: (1) *No Adaptation* does not perform any on-device training. (2) *FullTrain* fine-tunes the entire model. (3) *MCUTrain* is the state-of-the-art (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

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

		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 [4]	61.77	63.66	55.55	65.33	**81.55**	83.44	**82.44**	**87.33**	76.22	**89.22**	**71.77**	**81.11**
	BP-free (Ours)	**66.78**	**66.22**	**61.00**	**67.33**	80.11	**83.66**	81.00	86.33	**78.33**	88.22	69.78	79.56

to update based on *offline* evolutionary search on the *cloud* servers before MCU deployment, and updates the selected layers/channels on MCU. (4) *TinyTrain [9]* 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 back-propagation, the extra memory includes the saved intermediate activation, and the gradients of weights. For back-propagation-free training, the extra memory includes loss values, random seeds, and a buffer needed to save the largest gradients of weights (only for node perturbation). The analytic memory determines whether a 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 back-propagation or zeroth-order 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 [57] inference backend and an MCU domain-specific inference backend TinyEngine [5, 45]. 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 back-propagation. We then measure the actual **on-device per-iteration latency** for MCU-deployable training methods.

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.

- **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** [4] is a state-of-the-art (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*. Ref. [40] 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 and Latency).** 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.** Our BP-free training has a longer latency and training time, 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. This trade-off results in a much larger per-step latency and longer end-to-end training time 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 SOTA TinyEngine [4] for inference, the per-iteration latency of our BP-free training could be greatly reduced, leading to a greatly reduced total training time with BP-based training. While it is also possible to optimize the latency of BP, extra memory as well as extra high-precision computation resources are always needed. By further incorporating the sparse training, we could achieve a comparable end-to-end training time as BP full-model training.

- **Remark.** BP sparse training could achieve a low training memory [5] to fit 256-KB SRAM budget. However, this approach is not fully on-device, as it relies on *cloud* computation to select trainable parameters and perform BP computation graph optimization. Besides, the trainable parameters search space is constrained within ending layers, as training starting layers with BP also necessitates larger memory. Our BP-free training is all-on-edge, enables task-adaptive sparse training, and enables training any layer in the network to achieve the best sparse training performance. The trade-off among memory, flexibility, design/deployment difficulty, and end-to-end training time depends on the actual settings.

Table 8: Comparison of the hardware cost of full-model training.

		MCUNet			
Backend	Method	Memory (KB)	Compute (MMACs)	Latency (ms)	End-to-end Time (s)
TF-Lite [58]	BP	3,650	76.2	13,398	6.70E+05
	BP-free	668	7,857.5	428,496	2.14E+07
TinyEngine [4]	BP-free	668	7,857.5	72,839	3.64E+06

Table 9: Comparison of the hardware cost of sparse training. The underlined are measured results on an STM32F746 MCU.

		MCUNet			
Method	Test Accu.	Memory (KB)	Compute (MMACs)	Latency (ms)	End-to-end Time (s)
BP [5]	61.78	3,650	40.4	495	2.48E+04
BP-free	62.33	209	1,162.5	9,939	4.97E+05

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 [40, 5]. Nevertheless, due to the extremely limited memory budget, the BP-based method can only afford training three middle layers [5, 45] 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 10: Validation accuracy comparison on transfer learning to 5 FGVC datasets with our BP-free method and other BP training baselines. 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.

		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 [9]	FP32	55.2	57.8	**89.1**	52.3	80.9	67.1
MCUTrain [5]	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

5 Related Work

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

5.1 On-device Training

On-device inference has been well studied to deploy a pre-trained deep neural network (DNN) for inference on the edge. Driven by the demand to adapt edge-deployed neural network models to new data/new tasks [8, 5, 9] or to unseen distribution shifts at test time [59, 60], 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 automatic differentiation [2]. To implement training with BP on edge devices one has to implement gradient computations by hand, which is time-consuming and needs specific optimization. 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 [6, 7], bias only [8], normalization parameters [61]), 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* [5] enabled training on a microcontroller with merely 256KB SRAM and matched

cloud training results on the VisualWakeWords 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 4 layers (out of 42). *TinyTrain* [9] 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, state-of-the-art 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". Zeroth-order (ZO) optimization [26, 62, 38, 24] plays an important role in signal processing and adversarial machine learning where actual gradient information is infeasible (*e.g.*, black-box attack [63, 64, 32]). 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 [65, 66, 67], prompt/adapter tuning for emerging large language [68, 69, 70], vision [71, 72], and vision-language models [73], but not the backbone parameters of a pre-trained model. Recently, [14, 74] showed that ZO optimization was effective for full-parameter fine-tuning of large language models (LLM) up to 66 Billion parameters. However, this is only applicable when the pre-trained model has a very low intrinsic dimension (200-400) [75, 76], which is only observed in LLMs. For more general cases, the scalability of ZO training remains a fundamental challenge. Ref. [77] scaled up ZO training from scratch by leveraging tensor-compressed training for dimension reduction. DeepZero [15] 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) [78, 79, 80, 81]) 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 [82] for biologically-plausible learning, forward gradient method [83, 84, 35] based on forward-mode AD for memory-efficient gradient computation or to avoid stacked BP [85, 86] when higher-order derivatives are needed, log-likelihood method [87], node perturbation [37, 36], feedback alignment method [88] and input-weight alignment method [89]. Scalability, training stability, and convergence remain the top challenges of all emerging BP-free training frameworks.

6 Conclusion

In this paper, 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 a MCU at the similar memory cost of inference. This approach has addressed the long-standing memory challenge that prevents realistic training on a 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 node perturbation, 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 times. 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., integrated photonics, probabilistic circuits) will further expand its applicability, potentially impacting many application areas.

Acknowledgments

This work is supported by funding of Intel Strategic Research Sector (SRS) - Emerging Technology.

References

[1] Yann LeCun, D Touresky, G Hinton, and T Sejnowski. A theoretical framework for backpropagation. In *Proceedings of the 1988 connectionist models summer school*, volume 1, pages 21–28, 1988.

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

[3] Charles C Margossian. A review of automatic differentiation and its efficient implementation. *Wiley interdisciplinary reviews: data mining and knowledge discovery*, 9(4):e1305, 2019.

[4] Ji Lin, Wei-Ming Chen, Yujun Lin, Chuang Gan, Song Han, et al. Mcunet: Tiny deep learning on iot devices. *Advances in Neural Information Processing Systems*, 33:11711–11722, 2020.

[5] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. On-device training under 256kb memory. *Advances in Neural Information Processing Systems*, 35:22941–22954, 2022.

[6] Pramod Kaushik Mudrakarta, Mark Sandler, Andrey Zhmoginov, and Andrew Howard. K for the price of 1: Parameter-efficient multi-task and transfer learning. *arXiv preprint arXiv:1810.10703*, 2018.

[7] Haoyu Ren, Darko Anicic, and Thomas A Runkler. Tinyol: Tinyml with online-learning on microcontrollers. In *2021 international joint conference on neural networks (IJCNN)*, pages 1–8. IEEE, 2021.

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

[9] Young D Kwon, Rui Li, Stylianos I Venieris, Jagmohan Chauhan, Nicholas D Lane, and Cecilia Mascolo. Tinytrain: Deep neural network training at the extreme edge. *arXiv preprint arXiv:2307.09988*, 2023.

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

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

[12] Jianfei Chen, Yu Gai, Zhewei Yao, Michael W Mahoney, and Joseph E Gonzalez. A statistical framework for low-bitwidth training of deep neural networks. *Advances in neural information processing systems*, 33:883–894, 2020.

[13] Sunil Pai, Zhanghao Sun, Tyler W Hughes, Taewon Park, Ben Bartlett, Ian AD Williamson, Momchil Minkov, Maziyar Milanizadeh, Nathnael Abebe, Francesco Morichetti, et al. Experimentally realized in situ backpropagation for deep learning in photonic neural networks. *Science*, 380(6643):398–404, 2023.

[14] Sadhika Malladi, Tianyu Gao, Eshaan Nichani, Alex Damian, Jason D Lee, Danqi Chen, and Sanjeev Arora. Fine-tuning language models with just forward passes. *Advances in Neural Information Processing Systems*, 36:53038–53075, 2023.

[15] Aochuan Chen, Yimeng Zhang, Jinghan Jia, James Diffenderfer, Jiancheng Liu, Konstantinos Parasyris, Yihua Zhang, Zheng Zhang, Bhavya Kailkhura, and Sijia Liu. Deepzero: Scaling up zeroth-order optimization for deep model training. *arXiv preprint arXiv:2310.02025*, 2023.

[16] Qing Jin, Jian Ren, Richard Zhuang, Sumant Hanumante, Zhengang Li, Zhiyu Chen, Yanzhi Wang, Kaiyuan Yang, and Sergey Tulyakov. F8net: Fixed-point 8-bit only multiplication for network quantization. *arXiv preprint arXiv:2202.05239*, 2022.

[17] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.

[18] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010: 19th International Conference on Computational StatisticsParis France, August 22-27, 2010 Keynote, Invited and Contributed Papers*, pages 177–186. Springer, 2010.

[19] Diederik P Kingma. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[20] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[21] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

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

[23] Sijia Liu, Pin-Yu Chen, Bhavya Kailkhura, Gaoyuan Zhang, Alfred O Hero III, and Pramod K Varshney. A primer on zeroth-order optimization in signal processing and machine learning: Principals, recent advances, and applications. *IEEE Signal Processing Magazine*, 37(5):43–54, 2020.

[24] Sijia Liu, Bhavya Kailkhura, Pin-Yu Chen, Paishun Ting, Shiyu Chang, and Lisa Amini. Zeroth-order stochastic variance reduction for nonconvex optimization. *Advances in Neural Information Processing Systems*, 31, 2018.

[25] Katelyn Gao and Ozan Sener. Generalizing gaussian smoothing for random search. In *International Conference on Machine Learning*, pages 7077–7101. PMLR, 2022.

[26] Yurii Nesterov and Vladimir Spokoiny. Random gradient-free minimization of convex functions. *Foundations of Computational Mathematics*, 17(2):527–566, 2017.

[27] Saeed Ghadimi and Guanghui Lan. Stochastic first-and zeroth-order methods for nonconvex stochastic programming. *SIAM journal on optimization*, 23(4):2341–2368, 2013.

[28] James C Spall. Multivariate stochastic approximation using a simultaneous perturbation gradient approximation. *IEEE transactions on automatic control*, 37(3):332–341, 1992.

[29] Yury Demidovich, Grigory Malinovsky, Igor Sokolov, and Peter Richtárik. A guide through the zoo of biased sgd. *Advances in Neural Information Processing Systems*, 36, 2024.

[30] Yao Shu, Zhongxiang Dai, Weicong Sng, Arun Verma, Patrick Jaillet, and Bryan Kian Hsiang Low. Zeroth-order optimization with trajectory-informed derivative estimation. In *The Eleventh International Conference on Learning Representations*, 2023.

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

[32] Shuyu Cheng, Yinpeng Dong, Tianyu Pang, Hang Su, and Jun Zhu. Improving black-box adversarial attacks with a transfer-based prior. *Advances in neural information processing systems*, 32, 2019.

[33] Kaiyi Ji, Zhe Wang, Yi Zhou, and Yingbin Liang. Improved zeroth-order variance reduced algorithms and analysis for nonconvex optimization. In *International conference on machine learning*, pages 3100–3109. PMLR, 2019.

[34] Hao Di, Haishan Ye, Yueling Zhang, Xiangyu Chang, Guang Dai, and Ivor W Tsang. Double variance reduction: A smoothing trick for composite optimization problems without first-order gradient. *arXiv preprint arXiv:2405.17761*, 2024.

[35] Mengye Ren, Simon Kornblith, Renjie Liao, and Geoffrey Hinton. Scaling forward gradient with local losses. *arXiv preprint arXiv:2210.03310*, 2022.

[36] Sander Dalm, Marcel van Gerven, and Nasir Ahmad. Effective learning with node perturbation in deep neural networks. *arXiv preprint arXiv:2310.00965*, 2023.

[37] Naoki Hiratani, Yash Mehta, Timothy Lillicrap, and Peter E Latham. On the stability and scalability of node perturbation learning. *Advances in Neural Information Processing Systems*, 35:31929–31941, 2022.

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

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

[40] Yoonho Lee, Annie S Chen, Fahim Tajwar, Ananya Kumar, Huaxiu Yao, Percy Liang, and Chelsea Finn. Surgical fine-tuning improves adaptation to distribution shifts. *arXiv preprint arXiv:2210.11466*, 2022.

[41] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*, 2018.

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

[43] Kai Huang, Hanyun Yin, Heng Huang, and Wei Gao. Towards green ai in fine-tuning large language models via adaptive backpropagation. *arXiv preprint arXiv:2309.13192*, 2023.

[44] Yi-Lin Sung, Varun Nair, and Colin A Raffel. Training neural networks with fixed sparse masks. *Advances in Neural Information Processing Systems*, 34:24193–24205, 2021.

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

[46] François Panneton and Pierre L'ecuyer. On the xorshift random number generators. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 15(4):346–361, 2005.

[47] George Marsaglia. Xorshift rngs. *Journal of Statistical software*, 8:1–6, 2003.

[48] Dan Hendrycks and Thomas Dietterich. Benchmarking neural network robustness to common corruptions and perturbations. *arXiv preprint arXiv:1903.12261*, 2019.

[49] Jonathan Krause, Michael Stark, Jia Deng, and Li Fei-Fei. 3d object representations for fine-grained categorization. In *Proceedings of the IEEE international conference on computer vision workshops*, pages 554–561, 2013.

[50] Peter Welinder, Steve Branson, Takeshi Mita, Catherine Wah, Florian Schroff, Serge Belongie, and Pietro Perona. Caltech-ucsd birds 200. 2010.

[51] Maria-Elena Nilsback and Andrew Zisserman. Automated flower classification over a large number of classes. In *2008 Sixth Indian conference on computer vision, graphics & image processing*, pages 722–729. IEEE, 2008.

[52] Lukas Bossard, Matthieu Guillaumin, and Luc Van Gool. Food-101–mining discriminative components with random forests. In *Computer vision–ECCV 2014: 13th European conference, zurich, Switzerland, September 6-12, 2014, proceedings, part VI 13*, pages 446–461. Springer, 2014.

[53] Omkar M Parkhi, Andrea Vedaldi, Andrew Zisserman, and CV Jawahar. Cats and dogs. In *2012 IEEE conference on computer vision and pattern recognition*, pages 3498–3505. IEEE, 2012.

[54] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[55] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. pmlr, 2015.

[56] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[57] L Lai, N Suda, and V CMSIS-NN Chandra. Efficient neural network kernels for arm cortex-m cpus. arxiv 2018. *arXiv preprint arXiv:1801.06601*.

[58] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, et al. Tensorflow lite micro: Embedded machine learning for tinyml systems. *Proceedings of Machine Learning and Systems*, 3:800–811, 2021.

[59] Dequan Wang, Evan Shelhamer, Shaoteng Liu, Bruno Olshausen, and Trevor Darrell. Tent: Fully test-time adaptation by entropy minimization. *arXiv preprint arXiv:2006.10726*, 2020.

[60] Qin Wang, Olga Fink, Luc Van Gool, and Dengxin Dai. Continual test-time domain adaptation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7201–7211, 2022.

[61] Jonathan Frankle, David J Schwab, and Ari S Morcos. Training batchnorm and only batchnorm: On the expressive power of random features in cnns. *arXiv preprint arXiv:2003.00152*, 2020.

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

[63] Pin-Yu Chen, Huan Zhang, Yash Sharma, Jinfeng Yi, and Cho-Jui Hsieh. 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*, pages 15–26, 2017.

[64] Chun-Chen Tu, Paishun Ting, Pin-Yu Chen, Sijia Liu, Huan Zhang, Jinfeng Yi, Cho-Jui Hsieh, and Shin-Ming Cheng. Autozoom: Autoencoder-based zeroth order optimization method for attacking black-box neural networks. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 742–749, 2019.

[65] Yun-Yun Tsai, Pin-Yu Chen, and Tsung-Yi Ho. Transfer learning without knowing: Reprogramming black-box machine learning models with scarce data and limited resources. In *International Conference on Machine Learning*, pages 9614–9624. PMLR, 2020.

[66] Li Yang, Adnan Siraj Rakin, and Deliang Fan. Rep-net: Efficient on-device learning via feature reprogramming. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12277–12286, 2022.

[67] Zige Wang, Yonggang Zhang, Zhen Fang, Long Lan, Wenjing Yang, and Bo Han. Soda: robust training of test-time data adaptors. *Advances in Neural Information Processing Systems*, 36:44017–44038, 2023.

[68] Zixian Guo, Yuxiang Wei, Ming Liu, Zhilong Ji, Jinfeng Bai, Yiwen Guo, and Wangmeng Zuo. Black-box tuning of vision-language models with effective gradient approximation. *arXiv preprint arXiv:2312.15901*, 2023.

[69] Yifan Yang, Jiajun Zhou, Ngai Wong, and Zheng Zhang. Loretta: Low-rank economic tensor-train adaptation for ultra-low-parameter fine-tuning of large language models. *arXiv preprint arXiv:2402.11417*, 2024.

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

[71] Changdae Oh, Hyeji Hwang, Hee-young Lee, YongTaek Lim, Geunyoung Jung, Jiyoung Jung, Hosik Choi, and Kyungwoo Song. Blackvip: Black-box visual prompting for robust transfer learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 24224–24235, 2023.

[72] Shuaicheng Niu, Chunyan Miao, Guohao Chen, Pengcheng Wu, and Peilin Zhao. Test-time model adaptation with only forward passes. *arXiv preprint arXiv:2404.01650*, 2024.

[73] Yassine Ouali, Adrian Bulat, Brais Matinez, and Georgios Tzimiropoulos. Black box few-shot adaptation for vision-language models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 15534–15546, 2023.

[74] Yihua Zhang, Pingzhi Li, Junyuan Hong, Jiaxiang Li, Yimeng Zhang, Wenqing Zheng, Pin-Yu Chen, Jason D Lee, Wotao Yin, Mingyi Hong, et al. Revisiting zeroth-order optimization for memory-efficient llm fine-tuning: A benchmark. *arXiv preprint arXiv:2402.11592*, 2024.

[75] Armen Aghajanyan, Luke Zettlemoyer, and Sonal Gupta. Intrinsic dimensionality explains the effectiveness of language model fine-tuning. *arXiv preprint arXiv:2012.13255*, 2020.

[76] Sadhika Malladi, Alexander Wettig, Dingli Yu, Danqi Chen, and Sanjeev Arora. A kernel-based view of language model fine-tuning. In *International Conference on Machine Learning*, pages 23610–23641. PMLR, 2023.

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

[78] Jiaqi Gu, Zheng Zhao, Chenghao Feng, Wuxi Li, Ray T Chen, and David Z Pan. Flops: Efficient on-chip learning for optical neural networks through stochastic zeroth-order optimization. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[79] Jiaqi Gu, Chenghao Feng, Zheng Zhao, Zhoufeng Ying, Ray T Chen, and David Z Pan. Efficient on-chip learning for optical neural networks through power-aware sparse zeroth-order optimization. In *Proceedings of the AAAI conference on artificial intelligence*, volume 35, pages 7583–7591, 2021.

[80] Jiaqi Gu, Hanqing Zhu, Chenghao Feng, Zixuan Jiang, Ray Chen, and David Pan. L2ight: Enabling on-chip learning for optical neural networks via efficient in-situ subspace optimization. *Advances in Neural Information Processing Systems*, 34:8649–8661, 2021.

[81] Yequan Zhao, Xian Xiao, Xinling Yu, Ziyue Liu, Zhixiong Chen, Geza Kurczveil, Raymond G Beausoleil, and Zheng Zhang. Real-time fj/mac pde solvers via tensorized, back-propagation-free optical pinn training. *arXiv preprint arXiv:2401.00413*, 2023.

[82] Geoffrey Hinton. The forward-forward algorithm: Some preliminary investigations. *arXiv preprint arXiv:2212.13345*, 2022.

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

[84] Louis Fournier, Stéphane Rivaud, Eugene Belilovsky, Michael Eickenberg, and Edouard Oyallon. Can forward gradient match backpropagation? In *International Conference on Machine Learning*, pages 10249–10264. PMLR, 2023.

[85] Junwoo Cho, Seungtae Nam, Hyunmo Yang, Seok-Bae Yun, Youngjoon Hong, and Eunbyung Park. Separable physics-informed neural networks. *Advances in Neural Information Processing Systems*, 36, 2024.

[86] Xinling Yu, Sean Hooten, Ziyue Liu, Yequan Zhao, Marco Fiorentino, Thomas Van Vaerenbergh, and Zheng Zhang. Separable operator networks. *arXiv preprint arXiv:2407.11253*, 2024.

[87] Jinyang Jiang, Zeliang Zhang, Chenliang Xu, Zhaofei Yu, and Yijie Peng. One forward is enough for neural network training via likelihood ratio method. *arXiv preprint arXiv:2305.08960*, 2023.

[88] Arild Nøkland. Direct feedback alignment provides learning in deep neural networks. *Advances in neural information processing systems*, 29, 2016.

[89] Ping-yeh Chiang, Renkun Ni, David Yu Miller, Arpit Bansal, Jonas Geiping, Micah Goldblum, and Tom Goldstein. Loss landscapes are all you need: Neural network generalization can be explained without the implicit bias of gradient descent. In *The Eleventh International Conference on Learning Representations*, 2022.

Appendix

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
        $\hat{\nabla}_{\bar{\boldsymbol{\theta}}_i} \leftarrow \sum_q (\boldsymbol{l}_q - \boldsymbol{l})\boldsymbol{\xi}_q$
      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 $\hat{\nabla}_{\bar{\mathbf{a}}^{(i+1)}}\ell$
          $\hat{\nabla}_{\bar{\boldsymbol{\theta}}_i}\ell \leftarrow \hat{\nabla}_{\bar{\boldsymbol{\theta}}_i}\ell + (\bar{\mathbf{a}}_n^{(i)})^T\boldsymbol{\xi}_{q,n}(\boldsymbol{l}_{q,n} - \boldsymbol{l}_n)/\mu$ ▷ $(\bar{\mathbf{a}}^{(i)})^T\bar{\boldsymbol{\xi}}_n$ can be reduced to INT8 additions
        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 [5]

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

$$\bar{\boldsymbol{\theta}}_{t+1} = \text{clip}\left(\lceil \bar{\boldsymbol{\theta}}_t - \eta_{\bar{\boldsymbol{\theta}}}\hat{\nabla}_{\bar{\boldsymbol{\theta}}}\mathcal{L}(\bar{\boldsymbol{\theta}}, s_{\boldsymbol{\theta}}) \rfloor\right) \tag{14}$$

The update rule of full-precision parameter $\boldsymbol{\theta}$ with a global learning rate η:

$$\begin{aligned}
\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \eta \hat{\nabla}_{\boldsymbol{\theta}} \mathcal{L} \\
&= s_{\boldsymbol{\theta}} \left(\frac{\boldsymbol{\theta}_t}{s_{\boldsymbol{\theta}}} - \eta \frac{\hat{\nabla}_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})}{s_{\boldsymbol{\theta}}} \right) \\
\bar{\boldsymbol{\theta}}_{t+1} \approx \boldsymbol{\theta}_{t+1}/s_{\boldsymbol{\theta}} &= \left(\bar{\boldsymbol{\theta}}_t - \frac{\eta}{s_{\boldsymbol{\theta}}} \hat{\nabla}_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) \right) \\
\bar{\boldsymbol{\theta}}_{t+1} \approx \boldsymbol{\theta}_{t+1}/s_{\boldsymbol{\theta}} &= \left(\bar{\boldsymbol{\theta}}_t - \frac{\eta}{s_{\boldsymbol{\theta}}^2} \hat{\nabla}_{\bar{\boldsymbol{\theta}}} \mathcal{L}(\bar{\boldsymbol{\theta}}, s_{\boldsymbol{\theta}}) \right)
\end{aligned} \tag{15}$$

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

Scaling the learning rate is equivalent to scale the gradient $\hat{\nabla}_{\bar{\boldsymbol{\theta}}} \mathcal{L}(\bar{\boldsymbol{\theta}}, s_{\boldsymbol{\theta}})$ by $s_{\boldsymbol{\theta}}^{-2}$ 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 $s_{\boldsymbol{\theta}}$ follows the update of full-precision parameters.

C Extented Details of MCU Implementation

The following steps outline the implementation of XORShift to generate random numbers following the Rademacher distribution:

1. **Initialize the seed:** Set the seed for the XORShift generator, which will serve as the initial state.
2. **Update the state:** Apply the XORShift recurrence relations to the current state to generate a new pseudo-random number.
3. **Generate Rademacher-distributed values:** The generated pseudo-random number is transformed into a Rademacher-distributed value by examining the least significant bit (LSB) of the number. If the LSB is 1, return -1; otherwise, return $+1$.

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 least significant bit (LSB).