Adaptive ensemble optimization for memory-related hyperparameters in retraining DNN at edge

Xu, Yidong; Han, Rui; Zuo, Xiaojiang; Ouyang, Junyan; Liu, Chi Harold; Chen, Lydia Y.

DOI
10.1016/j.future.2024.107600

Publication date
2025

Document Version
Final published version

Published in
Future Generation Computer Systems

Future Generation Computer Systems 164 (2025) 107600

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Adaptive ensemble optimization for memory-related hyperparameters in retraining DNN at edge

Yidong Xu [a], Rui Han [a,*], Xiaojiang Zuo [a], Junyan Ouyang [a], Chi Harold Liu [a], Lydia Y. Chen [b]

[a] Beijing Institute of Technology, Beijing, China
[b] TU Delft, Delft, Netherlands

ARTICLE INFO

Keywords:
Edge computing
Deep neural networks (DNN)
Memory-related hyperparameters
Model retraining

ABSTRACT

Edge applications are increasingly empowered by deep neural networks (DNN) and face the challenges of adapting or retraining models for the changes in input data domains and learning tasks. The existing techniques to enable DNN retraining on edge devices are to configure the memory-related hyperparameters, termed *m*-hyperparameters, via batch size reduction, parameter freezing, and gradient checkpoint. While those methods show promising results for static DNNs, little is known about how to online and opportunistically optimize all their *m*-hyperparameters, especially for retraining tasks of edge applications. In this paper, we propose, MPOptimizer, which jointly optimizes an ensemble of *m*-hyperparameters according to the input distribution and available edge resources at runtime. The key feature of MPOptimizer is to easily emulate the execution of retraining tasks under different *m*-hyperparameters and thus effectively estimate their influence on task performance. We implement MPOptimizer on prevalent DNNs and demonstrate its effectiveness against state-of-the-art techniques, i.e. successfully find the best configuration that improves model accuracy by an average of 13% (up to 25.3%) while reducing memory and training time by 4.1x and 5.3x under the same model accuracies.

1. Introduction

Edge applications are increasingly powered up by neural networks (DNN) and harvest the benefit of faster communication speed and higher data security, in contrast to the solutions relying on the central intelligence on cloud [1–4]. Typically, an edge intelligence system runs have two types of jobs: *inference* jobs [5] that have stringent latency constraints and higher priority of resource usage, and *retraining* jobs [6] that are triggered when some input distribution shift happens [7]. The retraining tasks consume larger amounts of computational resources and need to be completed within a short retraining window (e.g. 10 or 30 min) to maintain consistent accuracy [8]. For instance, in an edge-side intelligent transportation application, a roadside device runs multiple inference jobs. At the same time, the DNN models used to deliver services (e.g. object detection and image classification) needs to continuously retrained when domain shift happens [9] (e.g. weather, light, or object density changes) or new task arrives (e.g. unseen vehicles or animals in previous mode training [10]). The edge system first needs to guarantee short response time for its inference jobs, then allocates remaining computational and memory resources to the retraining jobs. With the increasing size of modern DNNs (e.g. convolution neural networks (CNNs) and transformer-based networks [11]), there is a plethora of methods [12–14] to minimize the memory consumption of training DNN on resource-constrained edge devices. These methods rely on a set of memory-related hyperparameters, termed as *m*-hyperparameters, to control the tradeoff between the training efficiency and model accuracy, but also cause additional complexity in optimizing the (re)training process.

Example of *m*-hyperparameters. Fig. 1 illustrates three types of *m*-hyperparameters in edge-based DNN retraining: (i) **microbatch** [13, 15] allows a DNN to achieve a similar training effect as large batch size training by accumulating gradients while reducing the input data's batch size, namely reducing the memory footprint for storing these data points' gradients. Microbatch's *m*-hyperparameters include batch size (e.g. 4) and gradient accumulation step size (e.g. 64). (ii) **Parameter freezing** [14,16] reduces computational requirements and memory usage by freezing a portion of model parameters, which are thus not computed/updated during retraining. For instance, 30% of the model's parameters are frozen during the retraining to save 600MB memory. (iii) **Gradient checkpoint** [12,17] saves memory by selecting certain checkpoints and not storing the gradient during forward propagation for non-checkpoint layers' model parameters. Its *m*-hyperparameter is

* Corresponding author.
E-mail addresses: 3120211046@bit.edu.cn (Y. Xu), hanrui@bit.edu.cn (R. Han), 3120195517@bit.edu.cn (X. Zuo), 3220225183@bit.edu.cn (J. Ouyang), chiliu@bit.edu.cn (C.H. Liu), lydiaychen@ieee.org (L.Y. Chen).

https://doi.org/10.1016/j.future.2024.107600
Received 15 January 2024; Received in revised form 17 August 2024; Accepted 3 November 2024
Available online 10 November 2024

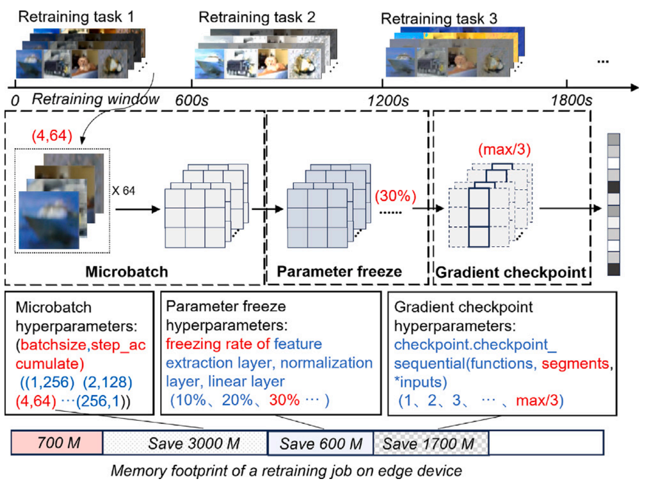

Retraining task 1
Retraining task 2
Retraining task 3
...
0
Retraining window
600s
1200s
1800s
(4,64)
X 64
Microbatch
(30%)
Parameter freeze
(max/3)
Gradient checkpoint
Microbatch hyperparameters: (batchsize,step_accumulate) ((1,256) (2,128) (4,64) ···(256,1))
Parameter freeze hyperparameters: freezing rate of feature extraction layer, normalization layer, linear layer (10%、20%、30% ···)
Gradient checkpoint hyperparameters: checkpoint.checkpoint_sequential(functions, segments, *inputs) (1、2、3、 ··· 、max/3)
700 M
Save 3000 M
Save 600 M
Save 1700 M
Memory footprint of a retraining job on edge device

Fig. 1. Example *m*-hyperparameters in DNN retraining tasks.

the number of segments, typically decided by the model's layer number. Here, 'layer' refers to basic layers in a DNN, such as convolutional layers, fully connected layers, and batch/layer normalization layers. In Fig. 1's example, using one-third of the maximum number (i.e. max/3) of model layers as the segment number can reduce memory usage by 1700MB.

Configuration-sensitive *m*-hyperparameters. Given a retraining task, the choice of *m*-hyperparameters leads to very different model performances, as minor tweaking on such hyperparameters can lead to changes in training steps and thus result in large discrepancies in both model accuracy and training efficiency [18,19]. Our evaluations of the ResNet50 model show that when the microbatch method changes batch size from 8 to 1, it reduces the memory usage from 3,000 MB to 800 MB, but also prolongs the training time by 20 times and leads to 6% accuracy losses [19]. It is certainly *no mean feat to gain potential performance improvement* from setting the best *m*-hyperparameters because the performance impact of different configurations as well as their interdependency depends on a wide range of variable factors. These factors include model architecture and size, input data distribution, and crucially the available memory and computational resources.

Challenges. Most *m*-hyperparameters tuning techniques, either used in practice [13] or described in the prior art [18,19], depend on application owners to manually set them in an offline fashion. However, today's edge applications usually have to tackle *evolving input distributions with big data*, which consist of new domains [9] or tasks [10], and *dynamically changing available resources*. For such applications, static configuration tuning techniques may overlook the potential enhancement through hyper-parameter optimization, because the optimal one is highly volatile. Two major challenges arise in adjusting a retraining task's *m*-hyperparameters for improving application performance.

First, retraining tasks need to be completed within a short training window (e.g. 10 min), but profiling time-varying input distributions and resources in a resource-constrained edge environment usually takes a long time. It needs a few iterations to verify the performance difference, e.g., resource consumption and accuracy improvement, between two configurations of *m*-hyperparameters. Finding the optimal combination of *m*-hyperparameters may need to compare hundreds of such hyperparameter values and thus need thousands of training iterations. The *first challenge*, therefore, is how to develop a lightweight on-device profiling approach to optimize *m*-hyperparameters searching while avoiding time-consuming profiling.

Second, each retraining task has an ensemble of *m*-hyperparameters, related to microbatch, parameter freezing, and gradient checkpoint methods, and each of them has a wide set of values. Moreover, depending on the applications' dynamics, a given combination of *m*-hyperparameters can result in different model accuracies. Determining the combination of *m*-hyperparameters that maximize both accuracy improvement and efficiency within an extensive search space poses an NP-hard problem. This raises the *second challenge* about how to efficiently search through the large *m*-hyperparameters space according to the transient dynamics on edge applications at run-time.

In this paper, we propose MPOptimizer, an adaptive ensemble optimizer to tune M̲emory-related hyperP̲arameters in real-time retraining DNNs at the edge. Our run-time tuning approach is designed to find the optimal configuration of *m*-hyperparameters via online profiling and searching, adapting to the latest input distribution and available resources. Note that MPOptimizer differs from traditional training hyperparameter optimization techniques [20], which search for the optimal hyperparameters (e.g. learning rate) to maximize model accuracy [18,21] via offline profiling. In contrast, we focus online configuring the *m*-hyperparameters studied and consider both model accuracy and resource consumptions. In particular, the contributions of this paper are as follows:

▷ **Online resource profiler based on prior knowledge.** MPOptimizer develops a lightweight resource profiler that computes resource utilization for various *m*-hyperparameters configurations. This computation is based on offline analysis conducted during the pre-training phase, as well as online knowledge acquired from prior training rounds. The resource profiler takes the retrained model, the current *m*-hyperparameters configuration, and the present system state and dataset as inputs for each *m*-hyperparameters combination. It then dynamically generates a resource profiling file. Ultimately, the profiler stores mappings of the *m*-hyperparameters and their respective resource profiling files.

▷ **Rule engine-based optimal hyper-parameter searching.** Based on the Drools rule engine [22], MPOptimizer comprehensively describes the searching of *m*-hyperparameters mechanisms using *business rules*, thus automating the search of large hyper-parameter space as judgment conditions of the rule engine and completing searching quickly when input distribution and available resources change.

▷ **Implementation and evaluation**. We implemented MPOptimizer on the prevalent DNNs including CNNs (ResNet-50, ResNet-101 [23], MobileNet-v2 [24]) and transformers (MobileViT [25]) to support retraining tasks of domain adaptation [9] and continual learning [10]. We evaluate our approach in real edge devices against state-of-the-art techniques and the results show: (i) MPOptimizer can effectively cope with changing input distributions and training resources by choosing appropriate *m*-hyperparameters to improve accuracy by an average of 13% (and up to 25.3%) (ii) under the same model accuracies, our approach reduces memory footprint by 4.1x and accelerates the retraining time by 5.3x.

The remainder of this paper is organized as follows: Section 2 briefly introduces the background of m-hyperparameters and related work. Section 3 explains the design of our approach and Section 4 evaluates it. Finally, Section 5 summarizes the work.

2. Background and related work

Performing training tasks on devices with limited memory resources is very challenging [1,3,9,10,26–28]. In this section, we first introduce the main types of memory footprint during training and the corresponding memory optimization strategies [29,30], and then we will focus on optimization techniques related to the activation memory [31] while eliciting the *m*-hyperparameters involved in these techniques. Finally we present relevant existing ensemble optimization techniques.

2.1. Memory footprint of training

The data to be saved for training usually occupies a large amount of memory, and the batch size of the input data affects data memory. The data samples in Fig. 2 illustrate this memory footprint. In addition to

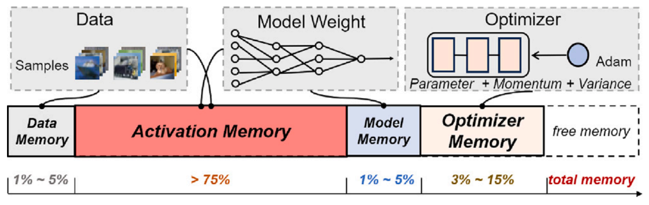


Fig. 2. Memory generated at each stage of the deep neural network training process.

the data memory footprint of the input dataset, there are three types of memory that are typically generated during neural network training:

(1) Model memory. In the context of deep learning models, model memory refers to the memory allocated for storing the model's weights. The model weight in Fig. 2 demonstrates this memory footprint. Pruning [32,33] as well as knowledge distillation [34] are common memory optimizations for this part of the memory.

(2) Optimizer memory. Optimizer memory is the memory used to store the gradient and momentum buffers corresponding to the parameters. The optimizer in Fig. 2 demonstrates the memory required by the Adam [35] optimizer during training, which is three times the model weights. CPU Offload [36] is a strategy that can reduce the memory usage during training by offloading the memory of optimizer from the GPU to the host CPU during training.

(3) Activation memory. As shown in Fig. 2, the forward propagation parameters obtained from the input data by performing calculations with the model weights are stored as activation memory, which tends to occupy the largest memory space. The following section describes related work on shrinking the activation memory, which are orthogonal to the other memory optimization methods described above.

2.2. Memory hyperparameter definition

. There exist numerous optimization techniques for activation memory without losing too much accuracy, which can be broadly categorized into three groups: microbatch/gradient accumulation [13], parameter freezing [14] and gradient checkpoint [12]. These techniques can control the adjustment intensity by setting corresponding hyperparameters. We call these hyperparameters that can adjust memory usage m-hyperparameters. The following will introduce in detail the specific m-hyperparameters corresponding to each technique:

Microbatch (gradient accumulation) is done by using smaller batches than minibatch for the input data stream into the network and accumulating the gradients equal to the number of minibatches. We regulate the impact of microbatch by *the batch size and the step size of gradient accumulation* (b, a) in the m-hyperparameters.

Parameter freezing is one of the common methods used to retrain models, some layers of the model need to be frozen, others positioned as updatable and fine-tuned during retraining. We regulate the impact of parameter freezing by *the parameter freeze rates* r in the m-hyperparameters.

Gradient checkpoint involves storing only a subset of network gradients during the forward propagation of the network, rather than all intermediate outputs. Other gradients are then recalculated from the nearest checkpoint when they are needed for backward propagation. We regulate the impact of gradient checkpoint by *the number of segments* cs in the m-hyperparameters.

2.3. Hyperparameter optimization techniques

With m-hyperparameters, we can easily adjust the impact of above three types of techniques and use them in combination at the same time to achieve better memory optimization. Efficiently selecting hyperparameters is a extensively discussed issue, and numerous classic techniques [37–40] can be employed to tune m-hyperparameters. In the following, we will introduce the existing hyperparameters tuning techniques which can be used for selecting m-hyperparameters:

Static setting based on the knowledge acquired during the pretraining phase is a widely adopted approach for configuring hyperparameters. In a large variety of methods for hyperparameter tuning through static analysis [21], static tuning for m-hyperparameters [13, 18] is a static setup method that analyzes the memory impact of different batch sizes b, different parameter freeze rates r and different number of checkpoint segments cs on training, and picks a set of m-hyperparameters that adapts to the available memory based on the current state of memory resources.

Dynamic adjustment technologies include online grid random search [21], dynamic rule search [19] and other tuning tools to find the optimal hyperparameters [41]. For m-hyperparameters tuning, there are a lot of tuning techniques [19] including dynamic gradient checkpointing, dynamic gradient accumulation, and hybrid strategy Sage. *Dynamic Gradient Checkpointing (DGC)* keeps all intermediate computation results in memory until the upper limit of available memory is reached, at which point it will set gradient checkpoint segments cs in m-hyperparameters and iteratively evict existing non-checkpointed computation results from memory. *Dynamic Gradient Accumulation (DGA)* prioritizes the maximum microbatch size supported under current memory conditions, sets a lower batchsize b and gradient accumulation step a in m-hyperparameters once memory is insufficient, and evicts all intermediate state individual batches from memory. *Sage* is a hybrid m-hyperparameters tuning technique that combines both DGC and DGA strategies.

Bayesian optimization (BO) [21] is the state-of-the-art technique among the hyperparameter tuning methods based on the training model. This hyperparameter optimization technique is usually used for the integrated tuning of network training hyperparameters such as learning rate, dropout rate and so on. Bayesian optimization has been widely used for hyperparameter tuning of different computing systems [42–45] discusses hyperparameter tuning under the simultaneous consideration of overall performance of memory, time, and accuracy, and [46] integrates various tuning methods for black-box optimization of hyperparameters on the basis of BO, but there is no work that applies it directly to the tuning of m-hyperparameters.

In summary, static methods [13,18] for setting m-hyperparameters may not always yield the most accuracy efficient configurations when facing changing input data in a dynamic domain. Dynamic adjustment technologies [19,41] offer better memory optimization under varying resource conditions, but their main focus is on improving memory efficiency. However, little attention has been given to the impact of these memory optimization methods on training accuracy. While Bayesian optimization [45] is a general method for hyperparameter tuning, directly applying it to memory tuning can be resource-intensive and may not allow enough time for retraining to improve accuracy. Therefore, our focus is on efficiently improving training accuracy while optimizing memory. To address this, we propose MPOptimizer, a lightweight edge-integrated method for m-hyperparameters tuning.

3. MPOptimizer

3.1. Overview

MPOptimizer is designed to improve performance of continuous model retraining via optimizing m-hyperparameter configurations on resource-constrained edge devices. As shown in Fig. 3, MPOptimizer integrates an offline knowledge extraction scheme with a rule engine, enabling lightweight tuning and retraining. At the online stage, MPOptimizer models the tuning effort of m-hyperparameters as an objective function and aims to obtain the optimal solution for this objective function using its three key components: *Resource Profiler, Rule Engine, and Tuning Controller*. In summary, MPOptimizer remains active on the

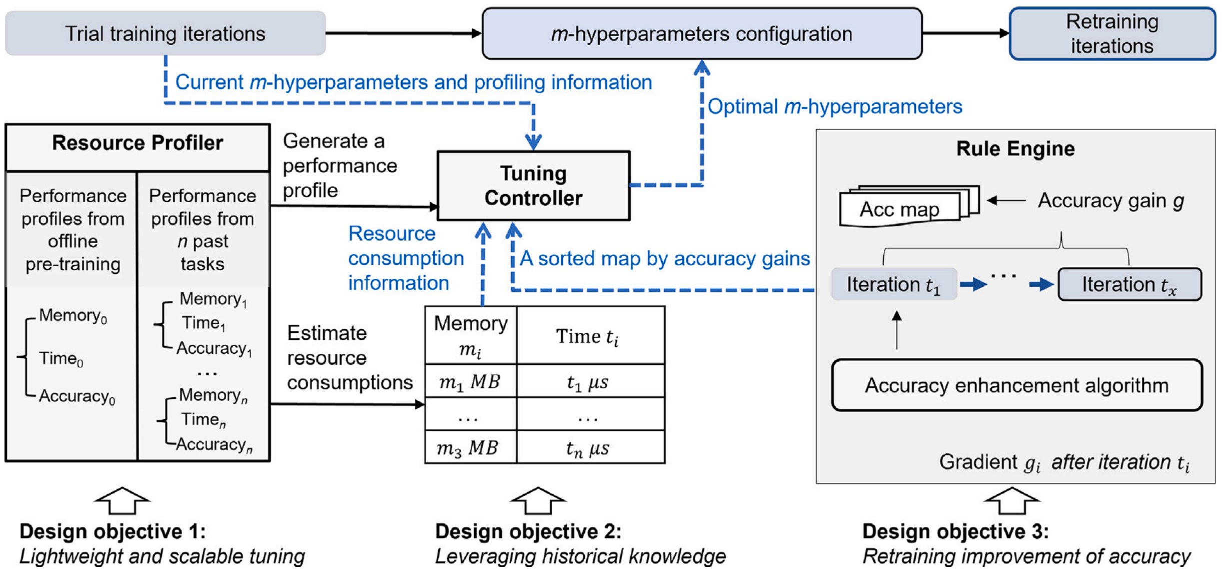

Trial training iterations
m-hyperparameters configuration
Retraining iterations
Current m-hyperparameters and profiling information
Optimal m-hyperparameters
Resource Profiler
Performance profiles from offline pre-training
Performance profiles from n past tasks
Memory$_0$ Time$_0$ Accuracy$_0$
Memory$_1$ Time$_1$ Accuracy$_1$ … Memory$_n$ Time$_n$ Accuracy$_n$
Generate a performance profile
Tuning Controller
Resource consumption information
A sorted map by accuracy gains
Estimate resource consumptions
Memory m_i | Time t_i
m_1 MB | t_1 μs
… | …
m_3 MB | t_n μs
Rule Engine
Acc map
Accuracy gain g
Iteration t_1 … Iteration t_x
Accuracy enhancement algorithm
Gradient g_i after iteration t_i
Design objective 1: Lightweight and scalable tuning
Design objective 2: Leveraging historical knowledge
Design objective 3: Retraining improvement of accuracy

Fig. 3. MPOptimizer and its three design objectives.

edge device throughout the retraining process and it is designed with three objectives.

Lightweight and scalable tuning. On resource-limited edge devices, MPOptimizer serves as a lightweight online integrated method for*m*-hyperparameter tuning. It rapidly completes the search for *m*-hyperparameters by extracting and integrating the necessary tuning knowledge. The **rule engine** in MPOptimizer utilizes the knowledge obtained from a small number of trial training iterations as the basis for updating the *m*-hyperparameters.

Leveraging historical knowledge. In MPOptimizer, the **resource profiler** plays a crucial role in handling each new retraining job. It takes the historical performance profiles stored during the pre-training phase and the retraining phases of previous tasks as input, and outputs a performance profile and a table of resource consumptions according to the current *m*-hyperparameters configuration. This information allows MPOptimizer to reason about the resource requirements of different *m*-hyperparameters configurations, and search for the optimal solution.

Retraining accuracy improvement. The **rule engine** of MPOptimizer develops an accuracy enhancement algorithm that takes the gradient g_i of the current iteration t_i as input and uses a number of iterations to calculate the final accuracy gain g. This information is then passed to the **tuning controller**, which manages the hyperparameter configurations and performs their tuning efficiently.

3.2. Problem statement

MPOptimizer aims to unearth the best overall performance of the system in resource-constrained scenarios at the edge end. To this end, this section presents the formulation of performance and define the performance model of a resource-constrained system.

We first define the three metrics used in optimization: (1) *memory usage* $M_D\left(\vec{\theta}_i\right)$ is decided by the configuration $\vec{\theta}_i$ of *m*-hyperparameters and the retraining data set D; (2) *retraining Time* $T_D\left(\vec{\theta}_i\right)$ denotes the model training time on data set D when using configuration $\vec{\theta}_i$; and (3) *accuracy metric* $A_D\left(\vec{\theta}_i\right)$ is the average accuracy improvement for a given configuration $\vec{\theta}_i$ of *m*-hyperparameters.

In optimization, MPOptimizer first find the range of *m*-hyperparameters under the limited memory $M_D\left(\vec{\theta}_i\right)$, and then searches the optimal configuration of *m*-hyperparameters to maximize the accuracy gain of with resource constraints:

$$\Lambda = \bigcup_{i=0}^{n} \Lambda_i \quad (when\ M_D\left(\vec{\theta}_i\right) \leqslant M_{max}) \tag{1}$$

$$P(\vec{\theta}_i) = \alpha A_D\left(\vec{\theta}_i\right) + \frac{\beta}{T_D\left(\vec{\theta}_i\right)} \tag{2}$$

$$\vec{\theta}^* = arg \max_{\theta\in\Lambda} P(\vec{\theta}_i) \tag{3}$$

In Eq. (1), Λ is the overall tuning space of all *m*-hyperparameters within the maximal memory M_{max}. In Eq. (2), $P(\vec{\theta}_i)$ is the performance objective function, α and β are the weights for accuracy and retraining time in Weighted Sum [47], where $\alpha + \beta = 1$. We note that proper settings of α and β depends on both training dataset and available device resources. That is, a higher value of α prefers accuracy improvement, and a higher value of β means retraining resource is limited. Hence for a specific retraining job, MPOptimizer chooses α and β using Bayesian Optimization (BO) [48] and tunes them together with other hyperparameters in Eq. (3). In Eq. (3), $\vec{\theta}^*$ is the optimal configuration of *m*-hyperparameters.

3.3. Resource profiler

The profiler collects resource utilization data for the pre-training phase and the completed retraining phase, utilizes this information to estimate the memory consumption of the current retraining job under different configurations of *m*-hyperparameters, and sends the estimation result to the rule engine.

As shown in Fig. 3, the profiler maintains three types of information from the pre-training and previous retraining phases: memory, training time, and accuracy. Using this accumulated information, the profiler estimates the impact of the current hyperparameter configuration on the resource consumption (m_i, t_i) of each layer in the network.

3.4. Rule engine

The rule engine utilizes a few quick trial iterations to provide a quick and lightweight solution for simplifying the hyperparameter search process. It contains two components: a memory range searcher and an accuracy enhancement algorithm.

Memory range searcher. To determine the range of the m-hyperparameters set in Eq. (1), the searcher utilizes a dichotomy-based approach for range reduction, and identifies sets of m-hyperparameters that satisfy the memory constraints. For instance, Table 1 shows all the hyperparameters that need tuning. The hyperparameter space size in Table 1 is $256 \times 256 \times 100 \times \frac{N}{3} \times 100 \times (10^{-2} - 10^{-5})$ (according to Eq. (1)) representing a quite large search space. After applying the Rule Engine's reduction process, the compressed space can be reduced to 10.

Accuracy enhancement algorithm. Solving the problem of finding the optimal performance configuration in Eq. (2) is challenging since it involves average accuracy and retraining time metrics, both of which are affected by the m-hyperparameters. This problem equals to the typical NP-hard problem of Resource-Constrained Project Scheduling Problem (RCPSP) [49]. Specifically, each m-hyperparameter's resource usage and accuracy gain correspond to a task in RCPSP; memory constraints and retraining windows correspond to its resource limits; and the configuration of m-hyperparameters corresponds to a complete project schedule in RCPSP.

To simplify the calculation and bring the average accuracy and retraining time under a common scale, MPOptimizer proposes an accuracy enhancement algorithm. This algorithm considers the performance of accuracy improvement when all m-hyperparameters simultaneously act on the training process within a unit time. It sorts this metric, which can then be multiplied by the number of retraining epochs under unit time, thus effectively reflecting the impact on the final performance. Specifically, the *memory range searcher* reduces the tuning space into a to a smaller size (e.g. 10 hyperparameter configurations). It enables an efficient linear search of optimal hyperparameter configuration for retraining. Subsequently, *accuracy enhancement algorithm* linearly searches for the optimal hyperparameter configuration by estimating performance/accuracy improvement for each configuration in the reduced tuning space. The accuracy improvement is measured by the reduction of model gradient value under a specific hyperparameter configuration per unit time. That is, a larger gradient reduction indicates a higher improvement. Finally, the configuration with the highest improvement is selected as the final choice for retraining.

Algorithm 1 equates the time window T used for retraining into smaller units of time t. Microbatch, checkpoint, and parameter freezing are three memory optimization methods corresponding to adjustable m-hyperparameters batchsize_accumulate, checkpoint_segments, and freezing_rate (line 1 to 4). The Acc function uses the gradient reduction value at the unit time of calculation to return the accuracy enhancement efficiency (lines 5 to 11). The algorithm comprises puts m-hyperparameters together for a complete training process per unit of time, records the accuracy improvement performance in this configuration (lines 12 to 18), adds it to the accuracy sorting map Acc_map for sorting (line 19 to 24). Finally, the algorithm returns a sorted map by accuracy performance from high to low.

Algorithm 1 Accuracy enhancement algorithm

```
Input: estimation requests $R$, $m$-hyperparameters $H$, one micro-window epoch seconds $t$, training window of $T$
 1. $budget \leftarrow T$
 2. $ba \leftarrow batchsize\_accumulate \leftarrow H[0]$
 3. $cs \leftarrow checkpoint\_segments \leftarrow H[1]$
 4. $fr \leftarrow freezing\_rate \leftarrow H[2]$
 5. function $Acc\ (r, ba, cs, fr)$ {Accuracy improvement estimation}
 6.     $acc_i \leftarrow r.get\_acc()$
        Using these parameters, retrain the model for request
 7.     $r$ for $t$ seconds
 8.     $acc_f \leftarrow r.get\_acc()$
 9.     $budget \leftarrow budget - t$
10.     return $(acc_f - acc_i)/t$ {Returns the accuracy gain}
11. end function
12. for $r \rightarrow R$ do
13.     $ba, cs, fr \leftarrow H$
14.     $Acc\_map(r, parameter\_list) \leftarrow Acc(r, ba, cs, fr)$
15. end for
16. $sorted(Acc\_map)$ {Accuracy sorted from high to low}
17. while $t < budget$ {Linearly searching from narrowed search space}
18.     $parameter\_list \leftarrow Acc\_map[0]$
19.     $ba, cs, fr \leftarrow parameter\_list$ {Max_gain parameters}
20.     $Acc\_map(r, parameter\_list) \leftarrow Acc(r, ba, cs, fr)$
21.     $sorted(Acc\_map)$
22. end while
```

3.5. Tuning controller

The controller regulates the functioning of the entire system by invoking other components of MPOptimizer and conducts hyperparameter tuning using four steps. Step 1 obtains the edge device's available resource information; step 2 triggers MPOptimizer to get the latest m-hyperparameters configuration; step 3 searches the optimal configuration with the memory and time constraints; and the final step sets the current retraining job according to the optimal configuration.

3.6. Running example

Fig. 4 illustrate the process of MPOptimizer in searching the optimal m-hyperparameter configuration within memory constraint and retraining window T_i. During the trial training iterations, the *resource profiler* obtains historical performance profiles of pre-training and past retraining, and generates performance profiles as well as the memory

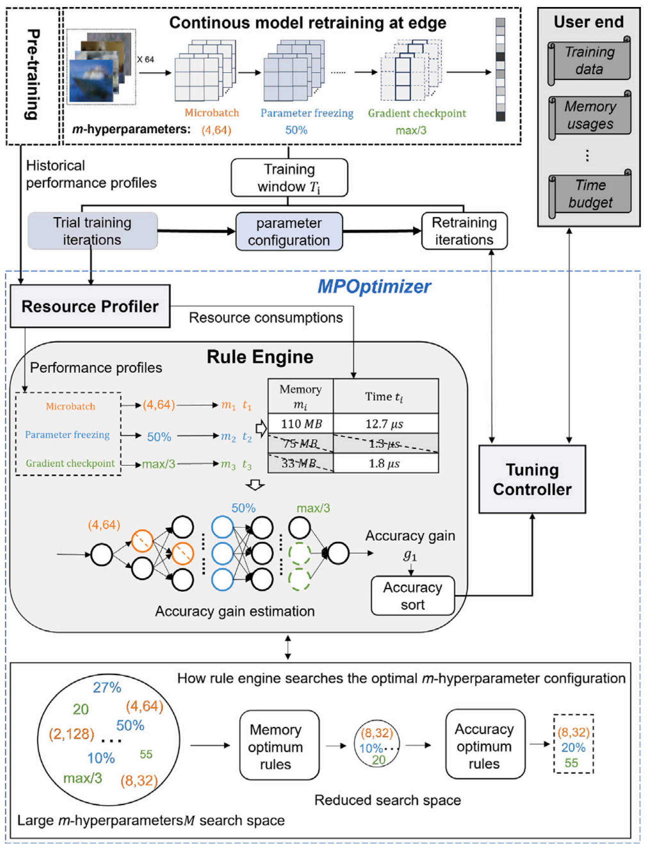


Fig. 4. Example m-hyperparameters tuning process with MPOptimizer.

and time consumptions as input of the *rule engine*. After retraining, the profiler retains the actual resource usages and performance (e.g. $m_1 = 110$ MB, $t_1 = 12.7\ \mu s, t_3 = 1.8\ \mu s$) for future retaining.

Subsequently, the *rule engine* performs emulations to estimate the existing *m*-hyperparameters configurations, feeds the predicted accuracy gain g_1 to an accuracy sorted array, and outputs an accuracy estimation sorted by accuracy improvement efficiency. In searching for the optimal configuration, the *rule engine* first narrows down the large search space, which contains all the ranges in which the *m*-hyperparameters can be tuned, to a reduced space of less than ten sets of *m*-hyperparameters combinations. It then quickly searches this space to find the configuration with the highest accuracy improvement using the *accuracy enhancement algorithm*.

Finally, the *rule engine* outputs the optimal configuration of *m*-hyperparameters ((8,32), 20%, 55) to the *tuning controller*, which conducts the hyperparameter tuning for the current retraining job.

4. Evaluation

In this section, we evaluate the implementation of MPOptimizer, built on the PyTorch framework. We conduct extensive experiments using a list of data benchmarks. First, we demonstrate the accuracy improvement of the *m*-hyperparameters chosen by MPOptimizer. (Section 4.2). Second, we present an analysis of resource consumption, emphasizing the advantages of MPOptimizer in reducing memory usage and training time (Section 4.3). Finally, we discuss the migratability of MPOptimizer (Section 4.4), its capabilities for lightweight configuration (Section 4.5), and its accuracy enhancement algorithm (Section 4.6).

4.1. Evaluation settings

Testbeds. We choose both CPU and GPU edge devices imposing different architectural features to demonstrate the MPOptimizer's tuning capabilities. The following three edge devices are used:

(i) Raspberry-Pi 4B, equipped with a 64-bit quad-core processor running at 1.5 GHz and 8 GB LPDDR4 memory;

(ii) Nvidia Jetson Xavier NX, featuring a 384-core Volta GPU with 48 Tensor Cores and 16 GB LPDDR4 memory;

(iii) Nvidia Jetson Nano, with a 128-core NVIDIA Maxwell GPU and 4 GB 64-bit LPDDR4 memory.

To address potential configuration challenges in ARM framework systems, we provide a docker environment that can directly simulate the experimental setup. In the PyTorch node, the versions of Python, PyTorch, CUDA, cuDNN, and Redis are 3.8.5, 1.7.1, 10.2, 7.6.4, and 6.0.10, respectively.

Models. Four prevalent DNN models are tested: (i) Convolutional Neural Networks (CNNs), including ResNet-50, ResNet-101 [23], and MobileNet-v2 [24]; (ii) a Transformer-based model called MobileViT [25]. These models are commonly employed in retraining tasks for training classifiers that demonstrate strong generalization capabilities across different domains with varying data distributions [50]. In particular, ResNet-50 consists of 50 layers and has a parameter size of 98MB. ResNet-101 is deeper with 101 layers and a parameter size of 171 MB. MobileNet-v2 is composed of 54 layers and requires 28MB of memory for its trainable parameters. MobileViT-XXS comprises 5 primary network layers and has 5MB of parameters.

Tuning space of *m*-hyperparameters. The space includes all hyperparameters associated with the three memory optimization techniques, as listed in Table 1.

MPOptimizer setting. For the *m*-hyperparameters, we initially set the default values for all methods as follows: (batch size, accumulate step size) = (8, 32); the number of checkpoint segments = $\sqrt{N}$ (where N represents the total number of layers in the network); parameter freeze rates = 40%. Regarding other hyperparameters, we employ the Adam optimizer. In the case of reducing the batch size ($batch\_size$) using the microbatch method, we decrease the learning rate by the same factor as the training converges. When the batch size is 256, the learning rate is set to 0.001.

Table 1
The tuning space of MPOptimizer.

m-hyperparameters	Min	Max
Batchsize	1	256
Step Accumulate	256	1
Freezing Rate	0	100%
Segment	1	$N/3$
Epoch	1	100
Learning Rate	10^{-5}	10^{-2}

Compared baseline. We evaluate and compare five state-of-the-art techniques for optimizing *m*-hyperparameters:

(i) The *static m-hyperparameters setting strategy* [18] proposes a static configuration strategy for combined *m*-hyperparameters. It achieves this by analyzing the memory consumption associated with different *m*-hyperparameters settings for three memory reduction techniques: microbatch, gradient checkpoint, and parameter freezing.

(ii) *DGC* [12] introduces a method to dynamically adjust the *m*-hyperparameters specifically for the gradient checkpoint technique, based on the available memory resources.

(iii) *DGA* [13] presents a dynamic adjustment method that adapts the *m*-hyperparameters for the microbatch strength, taking into account the available resources.

(iv) *Sage* [19] integrates the technologies of DGC and DGA. It continuously monitors the changes in available memory resources and dynamically adjusts the current *m*-hyperparameters using a combination of these two techniques. Sage offers a wider range of *m*-hyperparameters adjustment and can be utilized in scenarios with lower memory resources on the device.

(v) *BO* [45] employs the Bayesian Optimization algorithm to efficiently select hyperparameters. We directly apply this technique in the context of *m*-hyperparameters search.

Evaluation metrics. Accuracy is evaluated by quantifying the enhancement in model retraining within a fixed training time window. Resource consumption is measured by the memory and time during the retraining process.

4.2. Comparison of retraining accuracy

4.2.1. Comparison of accuracy improvement

The first evaluation tests the CIFAR-10-C dataset [51]. This dataset contains 15 distinct types of corruption, such as Gaussian noise, blur, snow, frost, fog, contrast, and others. Each corruption category includes 5 severity levels, thereby offering a diverse scenarios in evaluation. The evaluation focuses on the adaptation stage of the first six domains of CIFAR-10-C: natural, gaussian_noise, shot_noise, fog, saturate, and frost. The experiments run on two memory-constrained devices: Raspberry Pi and Nvidia Jetson Nano.

Evaluation settings. We set different retraining windows for the four different DNN models: ResNet-50 (110s), ResNet-101 (200s), MobileNet-v2 (180s), and MobileViT (200s). For all models, we set the initial *m*-hyperparameters batch size, step_accumulate, freezing rate, segments, epoch, and learning rate to 8, 32, 40%, 20, 5, and 0.085, respectively.

Evaluation results. We test an average of 100 retraining processes for each model and report the average percentage of accuracy improvement under different domains. As shown in Fig. 5, MPOptimizer achieves higher accuracy improvements in most of the cases compared to baseline techniques. For ResNet-50, ResNet-101, MobileNet-v2, and MobileViT, the average accuracy improvement is 25.32%, 10.58%, 9.5%, and 6.68%, respectively. The result indicates that MPOptimizer successfully set better configuration of *m*-hyperparameters that leverage limited devices resources to improve model accuracy.

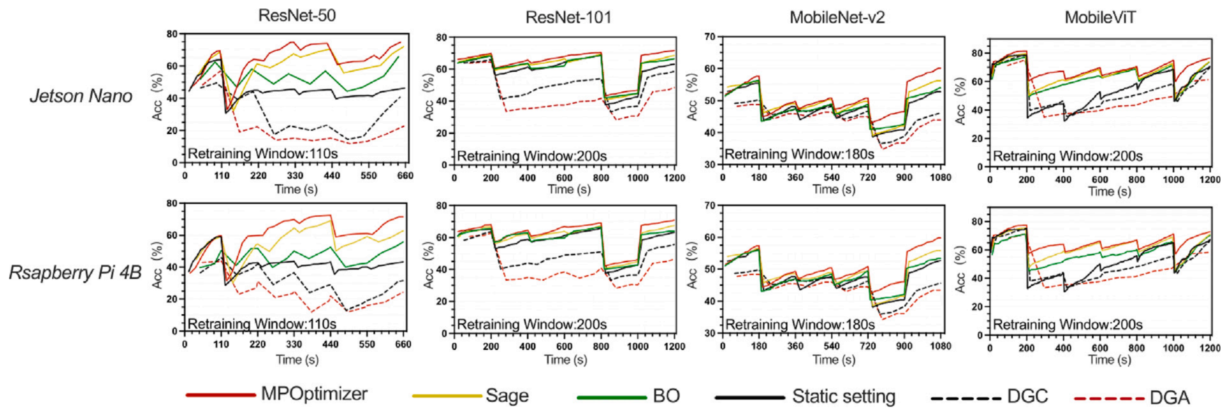


Fig. 5. Percentage of accuracy performance using different *m*-hyperparameters tuning techniques.

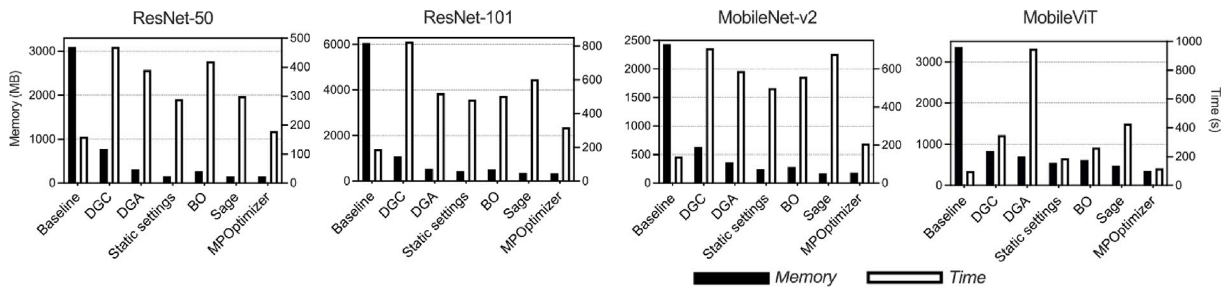


Fig. 6. Comparison of resource consumptions with same accuracy improvement.

Results. *When considering all evaluations, MPOptimizer improves retraining accuracy by 13% on average when using the same memory and training window as other baseline, and up to 25.3% in ResNet-50.*

4.2.2. Impact factors of tuning m -hyperparameters

Following the setting of the previous experiment, this evaluation tests ResNet-50 model on the Gaussian Noise Domain dataset and discusses two important factors that influence *m*-hyperparameter tuning.

(1) *Training time per round.* In MPOptimizer, each round of iterations completes faster compared to other baseline methods, and thus the model can achieve higher accuracy with more rounds within the same training time. This improvement is particularly noticeable with tight retraining window. For example, ResNet-50 achieved the highest average accuracy when the training window is only 110 s. In average, MPOptimizer increases training rounds by 5x compared to other techniques and thus improves the model accuracy by 9.12% with the same retraining window. (2) *Accuracy improvement per training round.* The *m*-hyperparameters in MPOptimizer also brings the highest accuracy improvement at each training round. Table 2 lists the *m*-hyperparameters configurations of all tuning techniques. We can see the combination of these hyperparameters found by MPOptimizer achieves the largest accuracy improvement per round.

Table 2
Example of optimal *m*-hyperparameter configurations and performance by each tuning technique on CIFAR10-C (Gaussian Noise Domain).

Best*m*-hyperparameters Configuration Found by Each Technique						
Tuning technique	MPOptimizer	Sage	BO	Static setting	DGC	DGA
Batch size	4	1	2	8	8	1
Step Accumulate	64	256	128	32	32	256
Freezing Rate	52.60%	40%	27.80%	40%	40%	40%
Segment	44	41	26	20	33	20
Epoch	9	7	3	5	3	2
Learning Rate	0.042	0.0053	0.0081	0.085	0.088	0.0046
The Performance of ResNet-50 by each technique on Gaussian Noise Domain in CIFAR10-C						
Tuning techniques	MPOptimizer	Sage	BO	Static setting	DGC	DGA
Each round's training time(s)	12	16	45	35	47	53
Accuracy improvement (%)	69.4	68.7	63.9	62.5	49.4	56.9

4.3. Comparison of resource consumption

In this section, we compare the resource consumptions between MPOptimizer and five *m*-hyperparameters tuning techniques under the same accuracy improvements. The evaluation tests the office-31 dataset and runs on an Nvidia Jetson Xavier NX device.

Evaluation settings. To ensure a fair comparison, all techniques are configured to achieve an accuracy of over 80% when migrating from the Amazon domain to the Digital domain. The initial *m*-hyperparameters batch size, step accumulation, freezing rate, segments, epoch, and learning rate are set to 32, 8, 0, 1, and 0.01, respectively. These values are obtained from pre-training of ImageNet dataset.

Evaluation results. Fig. 6 illustrates the resource consumptions of different techniques. We can see MPOptimizer outperforms other baselines by reducing training memory by an average of 1.8x, 5.6x, 2.1x, and 6.8x for ResNet-50, ResNet-101, MobileNet-v2, and MobileVit, and reducing training time by an average of 6.5x, 11.6x, 1.5x, and 1.6x, respectively. This is because MPOptimizer comprehensively utilizes multiple memory reduction techniques and hence benefits from their performance improvements to find a global optimal solution.

Results. *Compared to baseline techniques under the same accuracy improvement, MPOptimizer reduces memory footprint and training time by an average of 4.1x and 5.3x, respectively..*

4.4. Discussion of accuracy improvement across different domains

This section's evaluation discusses the migratability of our approach and the two best hyperparameter tuning techniques in previous experiments: Sage and BO.

Evaluation settings. This evaluation tests a baseline method that using sufficient memory resource (2000M) to train the ResNet-50

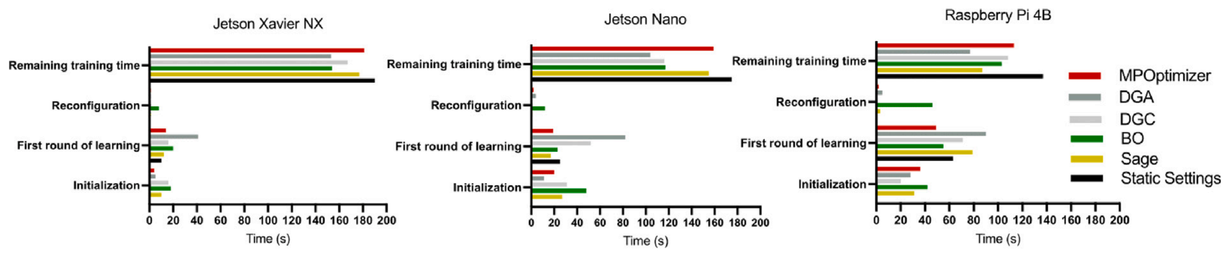


Fig. 7. Time consumed in the first round of configuration.

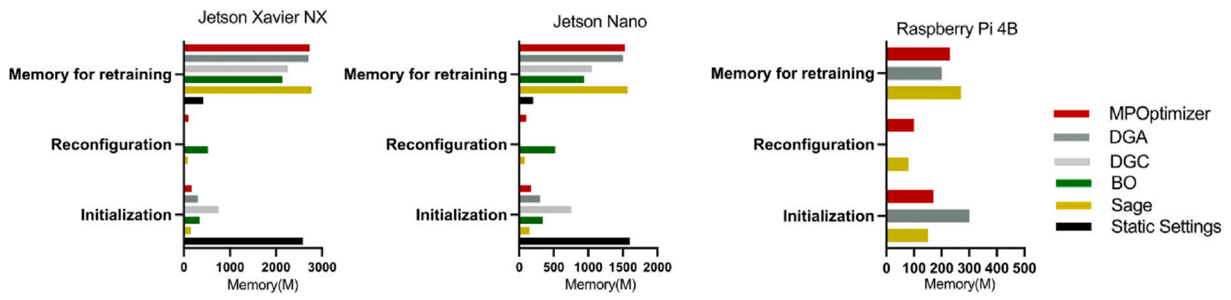


Fig. 8. Memory consumed in the first round of configuration.

Table 3
Classification accuracy (%) on **Digits** dataset. S:SVHN, U:USPS, M:MNIST.

Method (Source→Target)	M→U	U→M	S→M	Avg.
Baseline (sufficient memory) [23]	82.2	69.6	67.1	73.0
BO	69.1	53.4	49.7	57.4
Sage	75.6	64.3	61.2	67.0
MPOptimizer	80.8	68.9	66.3	72.0

model. In contrast, MPOptimizer, BO, and Sage runs on a Raspberry Pi device with less than 500M memory. All four techniques has a retraining window of 10 min for each data domain, and three different domains of the Digits dataset are tested: SVHN, USPS, and MNIST.

Evaluation results. Table 3 presents the accuracy improvements of the four techniques when retraining models for different domains. It is not surprisingly that the baseline technique using the largest memory has the highest accuracy improvement. MPOptimizer outperforms the other two techniques in all tested domains. This is because Sage does not consider accuracy optimization metrics for different datasets, and BO utilizes more memory in hyper-parameter searching, resulting in insufficient memory for running the retraining job. BO thus freezes more parameters and degrades accuracy in retraining. In contrast, MPOptimizer employs a lightweight tuning technique and prioritizes accuracy improvement in optimal hyperparameter searching.

Results. *MPOptimizer achieves the largest accuracy improvement when migrating to different three target domains (15% higher than BO and 9.6% higher than Sage), and only has slightly lower accuracy (1% lower) compared to the baseline method with sufficient memory resource.*

4.5. Resource consumption during initialization

This evaluation test the time and memory usage of MPOptimizer and baseline techniques during the initial round of *m*-hyperparameter configuration using ResNet-50 and CIFAR-10-C.

Evaluation settings. The initial round of the training task is conducted with a fixed time duration of 200 s on three different devices: Nvidia Jetson Xavier NX, Nvidia Jetson Nano, and Raspberry Pi 4B. The *m*-hyperparameters are uniformly set to their default values. The product of the batch size and accumulate_step are kept constant (256). When the batch size increases, the learning rate also increases proportionally to ensure an optimal training process.

Evaluation results. Fig. 7 shows that compared to baseline techniques, MPOptimizer completes the initialization faster and thus leaves more time for retraining. The static setting method has the longest remaining retraining time because it has no hyperparameter tuning.

In addition, Fig. 8 shows that when running on devices whose memory capacities are 3000MB, 1800MB, and 500MB, MPOptimizer and Sage consumes the least amount of memories during initialization, which take 9% and 7.6% of the total memory, thus remaining the most resources for retraining.

Results. *During the initialization phase, MPOptimizer consumes 60.88% of the time and 20.95% of memory compared to baseline tuning techniques.*

4.6. Discussion of the accuracy enhancement algorithm

This section's evaluation tests the overheads and accuracy improvement of the accuracy enhancement algorithm in MPOptimizer.

Evaluation settings. ResNet-50 is trained using each of the 15 domains in the CIFAR-10-C dataset. The baseline techniques employ a memory-intensive training configuration, including a batch size of 256, step accumulation of 1, freezing rate of 0, and a single segment. MPOptimizer is tested using the accuracy enhancement algorithm or not.

Evaluation results. Table 4 lists the comparison results. We can see that using the algorithm indeed increases retraining accuracies by 12.76% when considering all domains, while only slightly increasing memory usage and training time.

5. Conclusion

In this paper, we present MPOptimizer to optimize configuration of *m*-hyperparameters for evolving input distributions in edge-based retraining jobs. The core part of MPOptimizer is a rule engine that makes trade-off between three dimensions: memory, time, and accuracy, and quickly search for the optimal *m*-hyperparameters configuration for the current input distribution. MPOptimizer is implemented on PyTorch and evaluated against state-of-the-art *m*-hyperparameters optimization techniques to demonstrate its improvement in both model accuracy and training performance.

Table 4
Discussion of accuracy enhancement algorithm using 15 domains.

Task	Baseline			MPOptimizer without accuracy enhancement algorithm			MPOptimizer		
	Memory (M)	Time (s)	Accuracy (%)	Memory (M)	Time (s)	Accuracy (%)	Memory (M)	Time (s)	Accuracy(%)
natural	3113	160	62.6	155	181	68.2	178	207	83.9
gaussian_noise	3328	163	44.1	195	275	58.1	211	291	71.3
shot_noise	3224	162	48.8	157	185	60.6	182	221	74.7
specke_noise	3009	159	48.8	152	175	60.5	176	210	74
impuluse_noise	3138	160	36.7	155	180	51.6	178	206	63
defocus_blur	3206	160	55.8	156	180	62	180	214	74.8
gaussian_blur	3179	160	52.9	153	176	59.3	178	211	70.8
motion_blur	3442	168	51.3	158	192	55.4	185	212	64.6
zoom_blur	3120	160	51.7	155	180	56	178	217	66.4
snow	3544	169	58	162	199	64.3	185	235	78.9
fog	3100	160	49.6	152	177	50.5	176	212	59
brightness	3240	162	64.5	154	184	73.9	180	220	92.1
contrast	3155	160	40.1	155	182	39.5	177	215	47.9
saturate	3782	176	56.1	169	204	59.8	194	278	72.2
frost	3161	156	63.2	154	179	73.3	180	221	92.9

CRediT authorship contribution statement

Yidong Xu: Writing – original draft, Methodology. **Rui Han:** Writing – review & editing, Supervision, Investigation, Funding acquisition. **Xiaojiang Zuo:** Methodology, Data curation. **Junyan Ouyang:** Writing – original draft, Software. **Chi Harold Liu:** Writing – original draft, Funding acquisition. **Lydia Y. Chen:** Writing – review & editing, Validation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This paper was supported by the National Natural Science Foundation of China (Grant No. 62272046, 62132019, 61872337). Equal contributors: Yidong Xu and Xiaojiang Zuo.

Data availability

Data will be made available on request.

References

[1] S. Dhar, J. Guo, J. Liu, S. Tripathi, U. Kurup, M. Shah, A survey of on-device machine learning: An algorithms and learning theory perspective, ACM Trans. Internet Things 2 (3) (2021) 1–49.
[2] J. Chen, X. Ran, Deep learning with edge computing: A review, Proc. IEEE 107 (8) (2019) 1655–1674.
[3] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, J. Zhang, Edge intelligence: Paving the last mile of artificial intelligence with edge computing, Proc. IEEE 107 (8) (2019) 1738–1762.
[4] M.S. Murshed, C. Murphy, D. Hou, N. Khan, G. Ananthanarayanan, F. Hussain, Machine learning at the network edge: A survey, ACM Comput. Surv. 54 (8) (2021) 1–37.
[5] X. Yao, N. Chen, X. Yuan, P. Ou, Performance optimization of serverless edge computing function offloading based on deep reinforcement learning, Future Gener. Comput. Syst. 139 (2023) 74–86.
[6] T. Veiga, H.A. Asad, F.A. Kraemer, K. Bach, Towards containerized, reuse-oriented AI deployment platforms for cognitive IoT applications, Future Gener. Comput. Syst. 142 (2023) 4–13.
[7] R. Bhardwaj, Z. Xia, G. Ananthanarayanan, J. Jiang, Y. Shu, N. Karianakis, K. Hsieh, P. Bahl, I. Stoica, Ekya: Continuous learning of video analytics models on edge compute servers, in: 19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 22, 2022, pp. 119–135.
[8] H. Zhang, M. Shen, Y. Huang, Y. Wen, Y. Luo, G. Gao, K. Guan, A serverless cloud-fog platform for DNN-based video analytics with incremental learning, 2021, ArXiv, arXiv:2102.03012.
[9] M. Wang, W. Deng, Deep visual domain adaptation: A survey, Neurocomputing 312 (2018) 135–153.
[10] M. De Lange, R. Aljundi, M. Masana, S. Parisot, X. Jia, A. Leonardis, G. Slabaugh, T. Tuytelaars, A continual learning survey: Defying forgetting in classification tasks, IEEE Trans. Pattern Anal. Mach. Intell. 44 (7) (2021) 3366–3385.
[11] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need, in: Advances in Neural Information Processing Systems, vol. 30, 2017.
[12] T. Chen, B. Xu, C. Zhang, C. Guestrin, Training deep nets with sublinear memory cost, 2016, arXiv preprint arXiv:1604.06174.
[13] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q.V. Le, Y. Wu, et al., Gpipe: Efficient training of giant neural networks using pipeline parallelism, in: Advances in Neural Information Processing Systems, vol. 32, 2019.
[14] Q. Zhou, Z. Qu, S. Guo, B. Luo, J. Guo, Z. Xu, R. Akerkar, On-device learning systems for edge intelligence: A software and hardware synergy perspective, IEEE Internet Things J. 8 (15) (2021) 11916–11934.
[15] A. Harlap, D. Narayanan, A. Phanishayee, V. Seshadri, N. Devanur, G. Ganger, P. Gibbons, Pipedream: Fast and efficient pipeline parallel dnn training, 2018, arXiv preprint arXiv:1806.03377.
[16] C. Chen, H. Xu, W. Wang, B. Li, B. Li, L. Chen, G. Zhang, Communication-efficient federated learning with adaptive parameter freezing, in: 2021 IEEE 41st International Conference on Distributed Computing Systems, ICDCS, 2021, pp. 1–11.
[17] J. Feng, D. Huang, Optimal gradient checkpoint search for arbitrary computation graphs, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2021, pp. 11433–11442.
[18] N.S. Sohoni, C.R. Aberger, M. Leszczynski, J. Zhang, C. Ré, Low-memory neural network training: A technical report, 2019, arXiv preprint arXiv:1904.10631.
[19] I. Gim, J. Ko, Memory-efficient DNN training on mobile devices, in: Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services, 2022, pp. 464–476.
[20] A. Zhou, J. Yang, Y. Gao, T. Qiao, Y. Qi, X. Wang, Y. Chen, P. Dai, W. Zhao, C. Hu, Brief industry paper: Optimizing memory efficiency of graph neural networks on edge computing platforms, in: 2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium, RTAS, IEEE, 2021, pp. 445–448.
[21] T. Yu, H. Zhu, Hyper-parameter optimization: A review of algorithms and applications, 2020, arXiv preprint arXiv:2003.05689.
[22] S. Balcerek, V. Karovič, V. Karovič, Application of business rules mechanism in IT system projects, in: Developments in Information & Knowledge Management for Business Applications, vol. 2, 2021, pp. 33–112.
[23] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 770–778.
[24] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, L.-C. Chen, Mobilenetv2: Inverted residuals and linear bottlenecks, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 4510–4520.
[25] S. Mehta, M. Rastegari, MobileViT: Light-weight, general-purpose, and mobile-friendly vision transformer, 2021, arXiv e-prints, arXiv–2110.
[26] X. Qiang, Y. Hu, Z. Chang, T. Hamalainen, Importance-aware data selection and resource allocation for hierarchical federated edge learning, Future Gener. Comput. Syst. (2023).

[27] X. Wang, Y. Han, V.C. Leung, D. Niyato, X. Yan, X. Chen, Convergence of edge computing and deep learning: A comprehensive survey, IEEE Commun. Surv. Tutor. 22 (2) (2020) 869–904.
[28] H. Li, K. Ota, M. Dong, Learning IoT in edge: Deep learning for the Internet of Things with edge computing, IEEE Netw. 32 (1) (2018) 96–101.
[29] B. Steiner, M. Elhoushi, J. Kahn, J. Hegarty, Model: memory optimizations for deep learning, in: International Conference on Machine Learning, PMLR, 2023, pp. 32618–32632.
[30] M. Katsaragakis, L. Papadopoulos, M. Konijnenburg, F. Catthoor, D. Soudris, A memory footprint optimization framework for Python applications targeting edge devices, J. Syst. Archit. 142 (2023) 102936.
[31] A. Dorri, S.S. Kanhere, R. Jurdak, MOF-BC: A memory optimized and flexible blockchain for large scale networks, Future Gener. Comput. Syst. 92 (2019) 357–373.
[32] S. Vadera, S. Ameen, Methods for pruning deep neural networks, IEEE Access 10 (2022) 63280–63300.
[33] B.J. Eccles, P. Rodgers, P. Kilpatrick, I. Spence, B. Varghese, DNNShifter: An efficient DNN pruning system for edge computing, Future Gener. Comput. Syst. 152 (2024) 43–54.
[34] J. Gou, B. Yu, S.J. Maybank, D. Tao, Knowledge distillation: A survey, Int. J. Comput. Vis. 129 (2021) 1789–1819.
[35] D.P. Kingma, J. Ba, Adam: A method for stochastic optimization, 2014, arXiv preprint arXiv:1412.6980.
[36] J. Ren, S. Rajbhandari, R.Y. Aminabadi, O. Ruwase, S. Yang, M. Zhang, D. Li, Y. He, $\{ZeRO-Offload\}$: Democratizing $\{Billion-Scale\}$ model training, in: 2021 USENIX Annual Technical Conference, USENIX ATC 21, 2021, pp. 551–564.
[37] M. Merenda, C. Porcaro, D. Iero, Edge machine learning for ai-enabled iot devices: A review, Sensors 20 (9) (2020) 2533.
[38] R. Zaheer, H. Shaziya, A study of the optimization algorithms in deep learning, in: 2019 Third International Conference on Inventive Systems and Control, ICISC, IEEE, 2019, pp. 536–539.
[39] L. Yang, A. Shami, On hyperparameter optimization of machine learning algorithms: Theory and practice, Neurocomputing 415 (2020) 295–316.
[40] S. Liu, T. Ju, Apapo: An asynchronous parallel optimization method for DNN models, Future Gener. Comput. Syst. 152 (2024) 317–330.
[41] W. Chen, X. Dong, X. Chen, S. Liu, Q. Xia, Q. Wang, pommDNN: Performance optimal GPU memory management for deep neural network training, Future Gener. Comput. Syst. (2023).
[42] J. Snoek, H. Larochelle, R.P. Adams, Practical bayesian optimization of machine learning algorithms, in: Advances in Neural Information Processing Systems, vol. 25, 2012.
[43] K. Kandasamy, W. Neiswanger, J. Schneider, B. Poczos, E.P. Xing, Neural architecture search with bayesian optimisation and optimal transport, in: Advances in Neural Information Processing Systems, vol. 31, 2018.
[44] X. Ma, A.R. Triki, M. Berman, C. Sagonas, J. Cali, M.B. Blaschko, A Bayesian optimization framework for neural network compression, in: Proceedings of the IEEE/CVF International Conference on Computer Vision, 2019, pp. 10274–10283.
[45] X. Li, G. Zhang, W. Zheng, SmartTuning: selecting hyper-parameters of a ConvNet system for fast training and small working memory, IEEE Trans. Parallel Distrib. Syst. 32 (7) (2020) 1690–1701.
[46] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, D. Sculley, Google vizier: A service for black-box optimization, in: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2017, pp. 1487–1495.
[47] R.T. Marler, J.S. Arora, The weighted sum method for multi-objective optimization: new insights, Struct. Multidisc. Optim. 41 (2010) 853–862.
[48] S. Falkner, A. Klein, F. Hutter, BOHB: Robust and efficient hyperparameter optimization at scale, in: International Conference on Machine Learning, PMLR, 2018, pp. 1437–1446.
[49] F. Habibi, F. Barzinpour, S. Sadjadi, Resource-constrained project scheduling problem: review of past and recent developments, J. Project Manag. 3 (2) (2018) 55–88.
[50] D. Wang, E. Shelhamer, S. Liu, B.A. Olshausen, T. Darrell, Fully test-time adaptation by entropy minimization, 2020, ArXiv, arXiv:2006.10726.
[51] A. Krizhevsky, G. Hinton, et al., Learning multiple layers of features from tiny images, 2009.

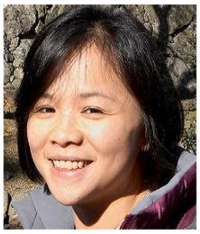
Yidong Xu is a Master student at the School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China. His research interests include edge computing and deep learning optimization.

Rui Han is an Associate Professor at the School of Computer Science and Technology, Beijing Institute of Technology, China. Before joining BIT, He received M.Sc. with honor in 2010 from Tsinghua University, China, and obtained his Ph.D. degree in 2014 from Imperial College London, UK. His research interests are system optimization for cloud data center workloads (in particular highly parallel services and deep learning applications). He has over 40 publications in these areas, including papers at MobiCOM, TPDS, TC, TKDE, INFOCOM, and ICDCS.

Xiaojiang Zuo is currently pursuing the Ph.D. degree with the School of Computer Science and Technology, Beijing Institution of Technology, Beijing, China. His research interests include federated learning and edge computing.

Junyan Ouyang is currently pursuing the Ph.D. degree with the School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China. His research interests include deep learning, federated learning and privacy.

Chi Harold Liu (SM'15) received the B.Eng. degree from Tsinghua University, Beijing, China, and the Ph.D. degree from the Imperial College London, London, U.K. He is currently a Full Professor and the Vice Dean with the School of Computer Science and Technology, Beijing Institute of Technology, Beijing. Before that, he worked for IBM Research - China and Deutsche Telekom Laboratories, Berlin, Germany, and IBM T. J. Watson Research Center, USA. He is now an Associate Editor for IEEE Trans. Network Science and Engineering. His current research interests include the big data analytics, mobile computing, and deep learning. Dr. Liu is a fellow of IET, and a fellow of Royal Society of the Arts.



Lydia Y. Chen is a full professor in the Department of Computer Science at the Technology University Delft. Prior to joining TU Delft, she was a research staff member at the IBM Zurich Research Lab from 2007 to 2018. She received Ph.D. from the Pennsylvania State University and B.A from National Taiwan University Her research interests center around dependability management, resource allocation and privacy enhancement for large scale data processing systems and services. She has published more than 80 papers in journals, e.g., IEEE Transactions on Distributed Systems, IEEE Transactions on Service Computing, and conference proceedings, e.g., INFOCOM, Sigmetrics, DSN, and Eurosys. She was a co-recipient of the best paper awards at CCgrid'15 and eEnergy'15. She is a senior IEEE member.