Structured Sparse Back-propagation for Lightweight On-Device Continual Learning on Microcontroller Units

Structured Sparse Back-propagation for Lightweight On-Device Continual Learning on Microcontroller Units

Francesco Paissan* [1], Davide Nadalini* [2,3], Manuele Rusci [4],
Alberto Ancilotto [1], Francesco Conti [3], Luca Benini [3,5], Elisabetta Farella [1]
[1]Fondazione Bruno Kessler, [2]Politecnico di Torino, [3]Universitá di Bologna,
[4]KU Leuven, [5]ETH Zurich

Abstract

With many devices deployed at the extreme edge in dynamic environments, the ability to learn continually on-device is a fast emerging trend for ultra-low-power Microcontrollers (MCUs). The key challenge in enabling Continual Learning (CL) on highly constrained MCUs is to curtail memory and computational requirements. This paper proposes a novel CL strategy based on sparse weight updates coupled with Latent Replay. We reduce the latency and memory requirements of the backpropagation algorithm by computing structured sparse update tensors for the trainable parameters, retaining only partial activations during the forward pass and limiting the per-layer gradient computation to a subset of channels. When applied to lightweight Deep Neural Network (DNN) models for image classification, namely PhiNet and MobileNetV2, our method can reduce up to 1.3× the memory and computation costs of the backpropagation algorithm, with a minor accuracy drop (2%). Furthermore, we evaluate the accuracy-latency-memory trade-off, targeting a class-incremental CL setup on a RISC-V multi-core MCU. The proposed approach allows to learn on-device a new class-incremental task, composed of two unseen classes, in 18 min *with* 4.63 MB *considering the most demanding configuration, i.e., a MobileNetV2 trained on the CORe50 dataset.*

1. Introduction

While the conventional approach to distribute machine learning models on resource-constrained devices involves cloud-based training followed by a static deployment for on-device inference, the surge in smart devices and advancements in the hardware platforms has spurred interest in On-Device Learning (ODL), i.e., training models on-device with environment-specific data. This paradigm shift paves the way for on-device Continual Learning (CL), inspired by algorithms that are designed to incrementally enhance the knowledge of a Deep Neural Network within a new target environment [7].

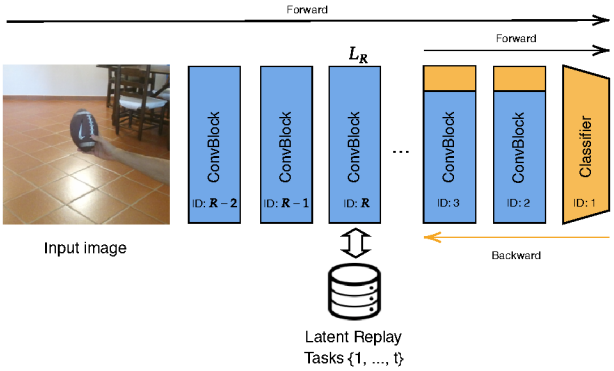

Figure 1. Schema of Latent Replay with sparse update. Layers in blue are frozen while learning the current task (τ_{t+1}), while orange layers are updated. Among the updated layers, only the weights corresponding to a portion of the activations are updated following our sparse update strategy described in Sec. 4.4.

However, devices at the extreme edge, typically powered by ultra-low power Microcontrollers (MCU), often deal with limited resources and connectivity. A new research field, namely Tiny Machine Learning (TinyML), has recently addressed the design of computation and memory-efficient Deep Learning models for on-device computation. Many works in this domain focus on inference optimization [17, 26], with proposals ranging from novel neural architectures [1, 15] to inference engines [17, 39] and Neural Architecture Search (NAS) strategies [5, 16]. Only a few recent works explored transfer learning tasks [4, 20] by learning a new classification task on-device. Conversely, we address a CL scenario differentiating from transfer learning by the ability to learn new concepts without forgetting the past knowledge, i.e. catastrophic forgetting.

*These authors contributed equally to this research.

redupdate thisThis work has been supported by the European Union's Horizon 2020 research and innovation program under grant agreement no. 957337 (MARVEL project).

Corresponding author: fpaissan@fbk.eu

The existing research effort to bring CL on-device is either confined to smartphone-level devices [28], which feature $> 10\times$ power consumption than our target device, or omits the cost implications of learning within resource-constrained devices [36]. Few recent works have started investigating the memory and latency cost of CL on MCUs [24, 29] by leveraging multi-core MCUs that we also adopt in this work. Orthogonally, a set of works [14, 20] proposed to reduce the memory and compute costs of the learning algorithm by updating only a subset of the trainable parameters of every layer, a technique referred to as *sparse update*. Only SparCL [38] applied this sparse logic to a CL setting, by updating a fraction of the weights with the highest relevance for the new task. Due to the irregular update scheme, i.e. not-structured, this approach is not suited for ultra-low-power MCUs.

In this paper, we set out to propose a CL strategy to learn a task stream directly on a ultra-low power MCU device [31]. To this aim, we explore how a state-of-the-art rehearsal-based CL strategy, Latent Replay (LR) [28], performs when coupled with structured sparse update logic. LR is a rehearsal-based CL technique tackling the problem of forgetting by mixing, at training time, the new samples with *latent* representations - i.e., intermediate activations - of a set of past data. For the sparse update, we consider a structured scheme by updating the parameters belonging to a set of input and output channels. The sparse update scheme remains fixed for the entire CL stream, to favor the implementation on-device.

Our contribution can be summarised as follows:

- We propose a structured sparse back-propagation algorithm targeting On-Device CL on resource-constrained MCU platforms;
- We analyze in depth the memory and computational gain deriving from our approach and the methodology to implement an efficient CL pipeline on-device;
- We explore the accuracy-latency-memory tradeoff of our technique on tiny CNNs for image classification while performing a class-incremental CL task.

The code to reproduce the results is available as open-source[1].

2. Related Work

In this section, we discuss relevant works from recent CL literature. We categorize the discussion into two sections: efficient CL strategies and techniques for ODL.

2.1. Efficient Continual Learning

Among the various CL techniques [11, 37, 40], rehearsal-based methods [2, 6, 12] have been extensively explored to mitigate catastrophic forgetting with limited computational resources. In general, rehearsal-based methods aim at mitigating forgetting by continuously training a DNN model on a mixed set of data coming from its past experience and newly acquired samples. A popular rehearsal-based method is Experience Replay (ER) [10]. In ER, while adapting the classifier to a new data distribution, we keep in memory a set of elements from the original data distribution, which is called replay. Therefore, the updates are applied to both the new data to learn the new distribution and the old data to preserve performance on the original data. A more efficient alternative, called Latent Replay [28] (LR), has been proposed. In LR, the replay memory is composed of latent representations of the input data, acquired as the output of one of the hidden layers. As hidden representations are generally smaller than the input images, this allows to save storage in terms of replay size, while also keeping the computation lower, as explained in Sec. 4.2. Tremonti et al. [36] study the performance of efficient neural architectures using LR, but do not consider the cost of the update inside the modelling of the constraints. SparCL [38], instead, experiments with many rehearsal-based methods, while focusing on what is the best strategy to perform sparse updates of the network. They suggest using a dynamic sparse-update scheme in which the sparsity mask is computed based on the relevance of each weight to the CL task. This solution comes with a computational overhead needed to perform the relevance estimation, which can be costly on-device.

2.2. On-Device Learning

In recent years, many papers in the literature have proposed strategies to perform learning on resource-constrained devices. These approaches typically fall into two categories. The first focuses on optimizing the backpropagation algorithm to lower its computational requirements. In contrast, the second one proposes new parameter-efficient strategies for adapting models, similar to current trends in large-scale models [9]. We introduce these two categories and we provide an overview of software libraries for ODL.

Optimizing backpropagation. De Vita et al. [8] extend the functionalities of STMicroelectronics X-CUBE-AI[2] to introduce support for on-device training of Echo State Networks for time series analytics. The method was tested on an STM32 MCU featuring less than 100 kB of memory occupation. PocketNN [34] presented a training methodology to exploit integer-only computation based on Direct Feedback Alignment [25]. Tiny Training Engine [18] combined gradient tensors pruning via offline calibration and a novel Quantization-Aware strategy for scaling the gradient magnitude and fitting the limited integer range.

Parameter-efficient ODL. TinyOL [30] proposes to insert a single trainable layer on top of a frozen and quantized

[1] https://github.com/fpaissan/odcl_sparsity

[2] https://www.st.com/en/embedded-software/x-cube-ai.html

model, trained in a few milliseconds using ARM-Cortex-equipped Arduino boards. TinyTL [4] proposes to limit the backpropagation to biases only, reducing the memory requirements by up to one order of magnitude. Train++ [35] implemented ODL for low-footprint devices but targeted shallow single-layer networks for binary classification problems.

Software libraries. To enable model training on ultra-low-power platforms, TyBox [27] proposes a C++ library for ODL with a particular focus on code generation. AIfES [39], instead, proposes a general-purpose library, targeting both inference and training, exploiting the ARM CMSIS libraries. On the other hand, with the aim of minimizing the latency of on-device training, Nadalini et al. [23, 24] propose a hardware-aware-optimized training library for PULP-based platforms, whose details are described in Sec. 5.2.

3. Background

3.1. The Backpropagation Algorithm

The current state-of-the-art Deep Neural Network (DNN) training methods are dominated by the backpropagation algorithm. Initially, a batch of training samples is used to estimate a loss score by comparing the ground-truth labels with the model predictions. Every prediction is obtained by propagating the input data through the DNN layers, known as the *forward step*. During this operation, the outputs of the intermediate layers, i.e., the *activations*, are retained in memory for the following step. Given the loss, the backpropagation algorithm computes the incremental update steps to optimize the DNN parameters. The gradient of the loss with respect to the last DNN layer is the error signal that is backpropagated through the model. Starting from the last layer, this *backward* pass includes two operations. First, the weight gradient (*Weight Grad*) step computes the gradient of the loss function with respect to the parameters by using the output gradient and the stored activations. Then, the input gradient step (*Input Grad*) backpropagates the output gradient towards the input with the weight parameters. Finally, an optimizer, such as Stochastic Gradient Descent (SGD), iteratively adjusts the weights of each layer based on the computed weight gradients.

3.2. Latent Replay

Given a neural network model of L layers, Latent Replay (LR) [28] is a rehearsal-based CL technique aiming at continually train the model on new samples (classes or domains), by mixing new data with a set of latent representation of previous knowledge - i.e., the *Latent Replay*. In particular, the LR data is stored as a set of latent activations collected at the R^{th} layer. While, at training time, the first $L - R$ layers from the input are kept frozen, the last layers update their parameters with backpropagation, by mixing new data with the LRs. Therefore, the mixture of new and old data mitigates forgetting, while leaving the last section of the model free to expand its knowledge.

Model name	MAC	Params	ImageNet Acc [%]
MobileNetV2	185.4M	1.4M	70.01
PhiNet	68.3M	1.0M	64.95

Table 1. Performance of the ConvNets used on the pretraining task. All numbers are referred to the ImageNet benchmark.

4. Hardware-Aware Continual Learning

We address class-incremental CL under limited memory and computational resources for the deployment in an MCU system, exploiting Latent Replay to tackle catastrophic forgetting. Targeting low-resource hardware, we adopt edge-oriented neural architectures. As illustrated in Fig. 1, we explore how to reduce the memory and compute requirements with minimal impact on the final accuracy by (i) changing the layer L_i at which we compute the replay vectors and (ii) by applying a sparse update scheme. After describing the neural network architectures in 4.1, Sec. 4.2 introduces the analytical cost model for the CL scheme with Latent Replay (Sec. 4.3) and Sec. 4.4 discusses the cost reduction achieved with sparse updates.

4.1. Edge-oriented neural networks

Given the limited memory budget of common MCUs (up to a few MB of on-chip memory), we consider lightweight DNN architectures with a number of parameters below 1.5 M [29, 36]. Table 1 reports the total number of parameters and operations, expressed in terms of Millions of Multiply-and-Accumulate (MMAC), for the PhiNet [26] and MobileNetV2 [33] networks, which are also used in [36]. The memory and computation requirements of these DNN architectures can be tuned by acting on several hyper-parameters. For MobileNetV2, we set the width multiplier parameter (α) to 0.75. PhiNet, instead, features a more advanced scaling mechanism based on three hyper-parameters, namely α (here 1.1), which scales the MMACs, β (0.75), scaling the number of parameters, and t_0 (5), scaling the working memory usage. We invite readers to refer to the original manuscripts for more details. We pre-train the two DNN models on Imagenet [32], achieving an accuracy of 64.95% and 70.01% for PhiNet and MobileNet, respectively. On these edge-oriented architectures, we emphasize the high memory cost of the last linear layer, i.e. the classifier, with respect to the total amount of parameters. In the case of MobileNetV2, the final layer carries up to 20% of the overall network parameters, which is reduced to 8% for Phinet thanks to the more advanced scaling knobs.

4.2. Analytical Cost Model

We build an analytical model to estimate the memory, compute, and storage costs of incrementally training a DNN using an LR strategy. Let us consider a neural network stacking N convolutional layers. For a convolutional layer i with a kernel of shape $k_i \times k_i$, input $\mathbf{x} \in \mathbb{R}^{C_{in,i} \times H_i \times W_i}$, and output $\mathbf{y} \in \mathbb{R}^{C_{out,i} \times H_i \times W_i}$, the MAC operations to compute the forward step (O_f) amounts to:

$$O_i^f = W_i H_i k_i^2 C_{in,i} C_{out,i} \tag{1}$$

where H_i, W_i, $C_{in,i}$, and $C_{out,i}$ represent the spatial resolution, number of input channels, and number of output channels. Instead, the number of MAC operations for the backward pass (O_i^b) is:

$$O_i^b = \overbrace{H_i W_i k_i^2 C_{in,i} C_{out,i}}^{\text{Input Grad}} + \overbrace{H_i W_i k_i^2 C_{in,i} C_{out,i}}^{\text{Weight Grad}} \tag{2}$$

where we distinguish the *Weight Grad* and the *Input Grad* operations. Note that, in the above formula, a fully-connected classifier can be viewed as a special case of the convolution where $H_i = 1$, $W_i = 1$, and $k_i = 1$.

On the other side, the memory costs for the backpropagation include the parameters and the weight gradients (M_{weight}) and the intermediate activation values retained from the forward step (M_{act}):

$$M_{weight,i} = \overbrace{k_i^2 C_{in,i} C_{out,i}}^{\text{Weight Grad}} + \overbrace{k_i^2 C_{in,i} C_{out,i}}^{\text{Weight Parameters}} \tag{3}$$

$$M_{act,i} = H_i W_i C_{in,i} \tag{4}$$

When considering deeper layers in the network, the input resolution is generally down-scaled, impacting the operations required for the backward and the memory for the input activations quadratically, as described in Eq. 2 and Eq. 4.

4.3. Latent Replay Costs

Referring to Fig. 1, we indicate with L_R the last frozen layer that we use to generate the latent replay vectors stored in memory. Thus, the layers $\{L_N, ..., L_R\}$ are frozen while the parameters of the layers $\{L_{R-1}, ..., L_1\}$ are updated with LRs. For a batch of N_{batch} elements, comprising the LR and the new processed data, the overall RAM memory costs, derived from Eq. 3 consist of:

$$M = \sum_{i=1}^{R-1} M_{weight,i} + N_{batch} M_{act,i} \tag{5}$$

The total memory cost for the update scales with multiple parameters: first, depending on where L_R is located, which determines the number of free-to-learn layers and the size of the LRs; second, with the batch size N_{batch}. Therefore, in this work we adopt a batch size of 16, comprising 4 LRs and 12 new samples.

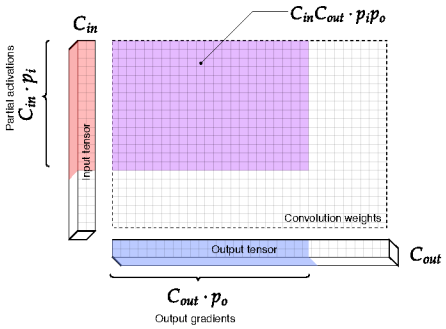

Figure 2. Overview of our sparse update method. During the backward pass, the weight gradients are computed using partial input activations (in red), stored during the forward step, and partially-computed output gradients (in blue).

Furthermore, the selection of the layer from which to extract the LRs influences the storage memory, which contains both the initialization weight parameters and the LR data.

The computational costs to perform CL with LRs can be accounted considering that the forward activation of the new samples are saved, as well as the LRs, after the first epoch of training. Therefore, for each epoch after the first, the computational cost is, for a single sample in a batch:

$$O = \sum_{i=1}^{R-1} O_i^b + O_i^f \tag{6}$$

The overall training cost, including the first epoch, can be obtained by adding the computational contributions O_i^f for $i = N, \ldots, R$ for each sample of the first epoch only.

4.4. Sparse Update

Inspired by recent advancements in the field [14, 19], we propose a **structured** sparse training approach applied to both the stored input activations and the activation gradients computed during the backward step. The term "structured" describes the choice of weights updated for each layer, which belong to a set of adjacent channels, differently from SparCL [38]. To describe the behavior of this update strategy analytically, we define two *partiality* coefficients: one for the input activations ($p_i \in [0, 1]$) and one for the output gradient ($p_o \in [0, 1]$), as depicted in Fig. 2. These partiality coefficients define the number of channels that are kept in memory and used for the computation of the weight gradients for, respectively, the stored input activation and the output gradient of a given layer. The computational and memory costs for updating of a single layer are, therefore, reduced proportionally with respect to the full update, determining a memory requirement of:

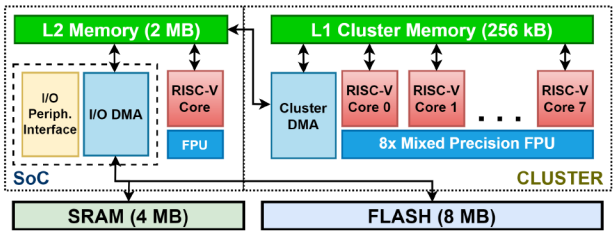

Figure 3. Architecture of the simulated PULP MCU.

$$M_{weight,i} = \overbrace{k^2(p_i \cdot C_{in})(p_o \cdot C_{out})}^{\text{Weight Grad}} + \overbrace{k_i^2 C_{in,i} C_{out,i}}^{\text{Weight Parameters}} \quad (7)$$

$$M_{act,i} = H_i W_i (p_i \cdot C_{in}) \quad (8)$$

while the number of operations scales accordingly, following

$$O_i^b = H_i W_i k^2 (\overbrace{(p_i \cdot C_{in})(p_o \cdot C_{out})}^{\text{Input Grad}} + \overbrace{(p_i \cdot C_{in})(p_o \cdot C_{out})}^{\text{Weight Grad}}) \quad (9)$$

Specifically, for a pointwise convolution ($k = 1$), applying the sparsity coefficient to the input and output produces a weight gradient that scales proportionally to the product of the two sparsity factors $p_i \cdot p_o$ (Eq. 7). For a depthwise convolution, instead, since the number of input channels is the same as the number of output channels ($C_{in} = C_{out}$), it is only possible to define a single sparsity factor $p = p_i = p_o$. The same scaling behaviour is preserved for the number of operations, as described in Eq. 9.

To simplify our analysis, we use a constant sparse update factor $p = p_i = p_o$ for all the layers except the classifier, which is always fully optimized. The partiality factor is also shared between input activations and gradient outputs. Finally, we emphasize that our sparse update approach is different in nature from the ones present in the literature [38], as it optimizes the memory computation starting from the number of channels kept in the input activations and output gradients rather than directly changing the number of updated weights.

5. Embedded Platform Deployment

In this section, we describe the target hardware platform and the design choices to deploy CL on the edge.

5.1. MCU Target Platform

As a reference MCU platform, we take a modular architecture template from the open-source PULP Platform[3], which is illustrated in Fig. 3. The processing unit features a single RISC-V core, 2 MB of on-chip SRAM memory, and a set of peripherals for interfacing to a wide variety of external components. Among them, we consider a low-power volatile RAM memory and a non-volatile FLASH memory with capacities of 4 MB and 8 MB, respectively. A peripheral DMA engine is then used to copy data between the internal and the external memories in the background of the core operation.

To accelerate compute-intensive tasks, e.g. DNN training, the platform also includes a compute cluster (denoted in short as the *cluster* in the following) with 8 general-purpose RISC-V. All the cores feature the RV32IMFC Instruction Set (ISA) with DSP-oriented extensions, e.g., post-increment load/store instructions and 2-level hardware loops. Every core can also access a private mixed-precision Floating-Point Unit (FPU), operating on 32-bit (FP32) and 16-bit (FP16) floating-point data. For the FP16 datatype, the ISA includes Single-Instruction-Multiple-Data (SIMD) MAC instructions that operate on 2× FP16 vectors, resulting in the maximum throughput of 2 MACs per clock cycle. The cluster features a 256kB fast-access memory tightly coupled to the cores and a DMA engine for transferring data between this memory and the bigger off-cluster memory.

5.2. On-Device Continual Learning Deployment

We refer to the PULP-TrainLib [23] framework to assess the compute and memory costs of the CL task on the reference PULP platform. PULP-TrainLib is an On-Device Learning library comprising latency-optimized training software functions based on Linear Algebra operators. To benefit from the compute cluster's acceleration capabilities (8 cores with SIMD MAC instructions), we consider an FP16 datatype for the parameters and intermediate activations.

We map the training task on the PULP platforms by taking into account the available on-chip and external memories. First, we distinguish between non-trainable and trainable parameters. While the former are initially stored in the off-chip FLASH memory, a copy of both parameters is kept in external RAM memory, along with the memory arrays for accumulating the computed gradients of the trainable weights. Additionally, the external RAM also retains (a batch of) the intermediate results of the trainable layers during the forward step, which are then used to compute the gradients during the backward pass. On the other hand, the replay data are preserved in the FLASH memory. The pool of replay data is also updated after learning a new class, replacing several samples from the old distribution with newly acquired ones and keeping the total to a fixed number (e.g., 3000 samples).

At runtime, the On-Device Learning task performs the forward and backward steps for all the training data (including the replays). The training loop is repeated for a total of 4

[3]PULP Platform: https://pulp-platform.org/

epochs. The computation is operated layer-by-layer during either the forward or backward steps. We account for the total latency, measured as the number of elapsed clock cycles (clk), by benchmarking the training operators (with a total number of operations in Eq. 9) on the target platform. For this purpose, we use the cycle-accurate open-source software simulator GVSoC [3].

The layer-wise software routines initially copy the input data from the external memories to the on-chip memory, with the peripheral DMA, and then into the cluster memory with the DMA. The training functions, i.e., forward and backward functions of every layer, take data from the low-level memory and use parallelization and SIMD instruction to speed up the computation. Note that processing efficiency, expressed as the ratio between the MAC operations and the clock cycles to execute (MAC/clk), varies between different types of layers and steps. For a PointWise layer, we measure peaks of 3.07, 3.29, and 2.78 MAC/clk during the forward, weight grad, and input grad steps, respectively. On the contrary, the efficiency reduces up to 0.86 MAC/clk for the DepthWise layers, whose core computations are less efficient to be optimized with linear algebra kernels.

5.3. On-Device Sparse Update

The sparse update execution differentiates from the baseline training functions as follows. The forward step is not affected by the *partiality* coefficients for what concerns the number of operations (Eq. 1). On the other hand, only $p_i \cdot C_{in,i}$ channels of each layer's activations are retained in the RAM memory, instead of the whole tensor.

During the backward phase, using a sparse update logic reduces instead the number of operations w.r.t. the full-update baseline. Starting from the last layer of the model, the input gradient is backpropagated by computing, for every layer, only a part of the gradient channels. More in detail, the operator takes $p_i \cdot C_{out,i}$ channels of the output gradient and the full weight parameters. This reduces the RAM memory requirements (Eq. 7) and the computational requirements (Eq. 9) proportionally.

Similarly, the sparse update brings computational benefit for the weight gradient computation. This backward step is fed by $p_i \cdot C_{out,i}$ channels of the output gradient and the partial input activations stored during the forward step. Finally, only $(p_i \cdot C_{in})(p_o \cdot C_{out})$ channels of the weights are updated for each layer.

6. Experimental Setup

Datasets. We use Split CIFAR-10 [13], referred to as CIFAR-10 in the remainder of the paper, and CORe50 [21]. For CIFAR10, we use the implementation presented in Avalanche [22], which consists of a total of five tasks, each one carrying two new categories. Each task contains around 5000 images. CORe50, instead, includes 50 objects belonging to 10 macro-categories. Following the same principle used for CIFAR10, we split the ten categories into five tasks of two categories each. We normalize the images by using the ImageNet statistics for both datasets before feeding them through the network.

Hyperparameters. We fix the number of epochs to 4 after observing a minimal final accuracy improvement (2%) in the case of 8 epochs. We set a replay buffer of 3000 and 5000 elements, for CIFAR-10 and CORe50, respectively. We use a Stochastic Gradient Descent optimizer with a learning rate of 1×10^{-3} for CIFAR10 and 1×10^{-4} for CORe50, which is found with a grid-search optimization.

7. Results

We analyze the trade-off between accuracy, latency, and memory required for the on-device learning task with the optimizations considered, i.e., varying the latent replay layer and the sparse update coefficient.

Our findings are summarized in Fig. 4, which shows the average accuracy over all five tasks after training the model on the entire continual stream. We experimented with nine layers for PhiNet and five layers for MobileNetv2 for the generations of the latent replays. We use partiality coefficients of $p = \{1, 0.9, 0.8, 0.7, 0.6\}$ for each layer, represented by different colors in the Figure. For each architecture and dataset, we report the memory occupation M, computed from Eq. 5 with $N_{batch} = 16$, and the latency required for the backward step. In Fig. 4, we report the latency in terms of clock cycles estimated using our simulator. Then, we convert the results into latency by scaling the number of clock cycles for the update with respect to a clock frequency of $450\,\mathrm{MHz}$, which is the maximum frequency measured on a recent silicon prototype of the PULP platform [31].

7.1. Performance-complexity tradeoff of PhiNet

On CIFAR10, the PhiNet configurations without sparse updates (blue curve) span from the backpropagation until the sole classifier, with a memory cost of $2.11\,\mathrm{MB}$, a latency of $0.6\,\mathrm{ms}$, and an accuracy of 78.64%, to the 5^{th}-to-last layer, demanding $5.38\,\mathrm{MB}$ of memory and $8\,\mathrm{ms}$ for the backward step, and achieving an accuracy of 84.45%. If also updating the layers after the 5^{th}-to-last layer layer, the accuracy decreases, suggesting a forgetting effect in the low-level features.

By introducing the sparse updates, we gain a more fine-grained control of the computational requirement of the backward step. In particular, we add five points in the Pareto curve (circled points in the plots), characterized by intermediate accuracy values and requirements. On both CIFAR10 and CORe50 datasets, we observe sharp Pareto curves for PhiNet. With a partiality factor of $p = 0.6$, the inference

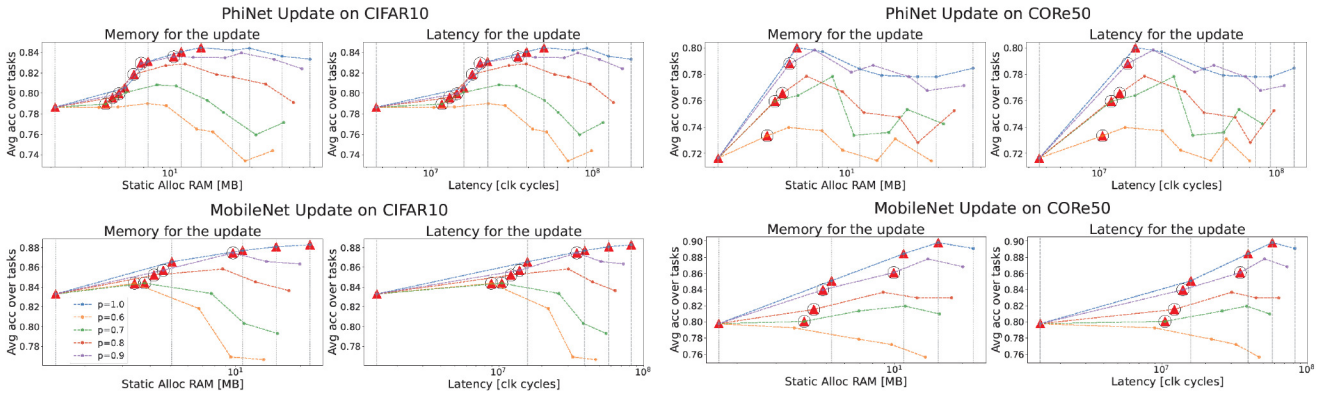

Figure 4. Pareto frontier for the CL performance with respect to different computational requirements. Each color is associated with a different sparsity level, while each dot describes a different layer. The linear classifier is depicted as the left-most dot. Going from left to right along the x-axis, each point represents a Latent Replay extracted from a layer closer to the input. Grey vertical lines describe the computational cost for each studied layer without sparsity ($p = 1$). The red triangles in the plot indicate an element of the Pareto frontier. Black circles highlight elements of the Pareto curve that are obtained when using sparsity.

time and memory are reduced by a factor of 1.1x and 1.4x, respectively, while a minor accuracy drop (2%) is measured with respect to retraining the model up to the 2^{nd} layer. In this same setting, we achieve a $\sim$2% improvement vs. updating only the classifier, with a computational overhead of 1.06x the memory and 1.15x the latency.

On CORe50, we confirm that training only the last two layers of the network leads to the highest accuracy, up to 79.81%, as shown in Fig. 4. This setting demands 3 ms to run the backward step and 2.54 MB of memory. On the other hand, the backpropagation on the sole classifier demands the same memory and latency as the CIFAR10 case (the number of input and output features are the same). Using the sparse update, the Pareto curve is extended from two to six points, enabling fine-grained control of the complexity of the backward step. With a factor of $p = 0.9$, we observe a 7% accuracy increase with respect to updating the classifier using 2.47 MB of memory (1.17x) and a latency cost of 3 ms.

7.2. Performance-complexity tradeoff of MobileNet

The Pareto frontier obtained with MobileNet is smoother than the PhiNet case. On CIFAR10, MobileNet achieves the highest accuracy of up to 88.28% when updating the entire network. In this case, the memory cost is 10.02 MB and the backward latency is 66 ms. On the other side, updating only the last layer is the fastest and low-memory configuration (0.8 ms and 1.46 MB) but achieves the lowest accuracy (83.29%). Using sparse updates with $p = 0.9$, instead, we gain 2% accuracy with respect to updating only the classifier, with a memory cost of 2.3 MB (1.1x) and a backward latency of 5 ms (6.25x).

On CORe50, the best accuracies range from 80%, when updating only the linear, to 89%, when updating the second-to-last layer. The latter setting demands 3.04 MB and 18 ms. Similarly to the other scenarios, we can define intermediate requirements thanks to the sparse update logic. By updating the weights up to the second layer, with a sparse update coefficient of $p = 0.6$, we observe a 1.5% accuracy increase with respect to backpropagating only up to the linear layer, with a computational overhead of 1.07x the memory, and 12.5x the latency.

For both architectures, we noted that the sparse update negatively affects the accuracy when considering the latent layer closer to the input, as showcased in Fig. 4. For example, on CIFAR10, for the 5^{th} layer starting from the classifier in MobileNet, we observe an accuracy drop of 2% when using a sparse update factor $p = 0.6$, while for the 8^{th} layer, this gap increases to 6%.

7.3. End-to-end LR on-device

We analyze now the overall requirements for incrementally training a DNN model on-device with out CL setting. Table 2 reports the latency and memory requirements needed to learn one task of the CL scenarios described in Sec. 6. This requires rehearsing information from the replay elements while learning from the elements of the new task. The statistics consider $N_{batch} = 16$ and a replay size of 3000 samples for CIFAR10 and 5000 samples for CORe50. In the latency estimation, we account for the forward and backward steps for the trainable layers along with the forward steps of the first layers in the case of non-replay data. In the table, we only include the most efficient update settings present in the Pareto curves: updating only the classifier and the backpropagation up to the layers with second-to-last for PhiNet and 4^{th} from the classifier for MobileNet.

	Layer ID	p	CIFAR10					CORe50				
			M (MB)	S_{replay} (MB)	Latency FW (s)	Latency Update (s)	Acc [%]	M (MB)	S_{replay} (MB)	Latency FW (s)	Latency Update (s)	Acc [%]
PhiNet	2	1	4.95	5.76	158.53	116.79	80.51	6.41	23.04	514.29	307.29	78.35
	2	0.8	4.1	5.76	158.53	88.34	79.59	5.26	23.04	514.29	231.23	76.57
	2	0.7	3.39	5.76	158.53	62.59	78.95	4.26	23.04	514.29	163.1	75.57
	1	1	2.11	1.25	61.56	0.06	78.64	2.11	2.10	222.22	0.07	71.65
MobileNet	4	1	3.9	4.61	113.88	356.16	86.55	6.3	18.43	464.69	1050.03	85.09
	4	0.8	3.25	4.61	113.88	271.36	85.20	5.17	18.43	464.69	785.34	81.54
	4	0.7	2.95	4.61	113.88	231.65	84.39	4.63	18.43	464.69	663.72	80.06
	1	1	1.46	7.68	57.56	3.33	83.29	1.46	12.8	212.00	12.27	79.78

Table 2. Computational cost of the entire CL stream ($N_{batch} = 16$) for MobileNet and PhiNet on CIFAR10 and CORe50. Total CL time to learn 2 new classes (task) is the sum of the FW and Update.

For PhiNet, as observed from the previous results, updating only the parameters of the last layer is the fastest option, leading to an overall latency cost of $61.62\,\mathrm{s}$ for CIFAR10 and $222.29\,\mathrm{s}$ for CORe50. These configurations reach the lowest accuracy of 78.64% and 71.65% for the two benchmarks, respectively. Retraining more layers (up to the 2^{nd}), the cost increases to $275.32\,\mathrm{s}$ and $821.58\,\mathrm{s}$ for an accuracy improvement of 2% and 7% on CIFAR10 and CORe50. MobileNet shows a similar trend where the latency for the entire CL stream goes from $60.89\,\mathrm{s}$ for CIFAR10 and $224.27\,\mathrm{s}$ for CORe50 to $470.04\,\mathrm{s}$ and $1514.72\,\mathrm{s}$ when updating the 4^{th} layer from the classifier.

By using a sparse update logic, we observe a smooth reduction in terms of computational cost for the LR strategy, while the accuracy shows a minimal drop vs. the non-sparse case. For the storage memory, we do not observe any changes with respect to the partiality factor, as all elements of the replay are needed to compute the forward step. When using a partiality factor $p = 0.7$, the memory required for the update scales down by a factor of 1.46x and 1.5x for PhiNet on CIFAR10 and CORe50, and by a factor of 1.3x and 1.36x for MobileNet on the two benchmarks. An analogous trend is observed for the latency requirement on the CL stream.

Hence, we can learn tasks from two common CL benchmarks on-device with fewer than $6.3\,\mathrm{MB}$ and less the $25\,\mathrm{min}$ for the most demanding configuration, which is MobileNet of CORe50. Using sparse updates, we can lower this requirement to $4.63\,\mathrm{MB}$ and $18\,\mathrm{min}$.

7.4. Comparison with the state of the art

Comparing our on-device continual learning strategy with the state of the art is not trivial, being our work the first targeting deployment on MCUs, as explained in Sec. 2. However, we try to contextualize the results of our method by first quantifying the overall cost of the update for the entire CL pipeline. Then, to have a reference, we compare it with the results reported in SparCL [38] on CIFAR-10. It should be noted that SparCL and our pipeline present two very different objectives. Specifically, SparCL shows how a dynamic sparse update strategy performs when learning a model on a CL from scratch. Conversely, we start from a pre-trained model and continually learn from the data stream, achieving comparable results with SparCL by updating the fully-connected classifier only, as shown in Table 3. With a pre-training on ImageNet and a fine-tuning of the sole classifier, we achieve $\sim 2\%$ accuracy improvement, with two orders of magnitude less computation. However, this comes with a larger computational overhead on the pre-training side.

	FLOPS Train ↓	Mem ↓	Acc [%] ↑
SparseCL [38]	2.50×10^{15}	177 MB	74.9
Ours	8.68×10^{11}	2.11 MB	77.3

Table 3. Numerical comparison of our approach with SparCL [38].

8. Conclusion

This paper presented an empirical evaluation of a novel sparse update strategy applied to a Latent Replay-based CL pipeline, as a viable lightweight solution for extreme-edge devices. We validated the proposed approach on two common benchmarks, observing that the optimization of the classifier only results in the lowest accuracy for both benchmarks, while constituting the fastest learning paradigm. Conversely, introducing sparse updates on deeper layers enhances the performances, while reducing the computational cost of learning, with respect to classic backpropagation.

References

[1] Alberto Ancilotto, Francesco Paissan, and Elisabetta Farella. Xinet: Efficient neural networks for tinyml. *2023 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 16922–16931, 2023. 1

[2] Benedikt Bagus and Alexander Gepperth. An investigation of replay-based approaches for continual learning. In *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–9, 2021. 2

[3] Nazareno Bruschi, Germain Haugou, Giuseppe Tagliavini, Francesco Conti, Luca Benini, and Davide Rossi. Gvsoc: a highly configurable, fast and accurate full-platform simulator for risc-v based iot processors. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*, pages 409–416. IEEE, 2021. 6

[4] Han Cai, Chuang Gan, Ligeng Zhu Massachusetts Institute of Technology, and Song Han Massachusetts Institute of Technology. Tinytl: reduce memory, not parameters for efficient on-device learning. Red Hook, NY, USA, 2020. Curran Associates Inc. 1, 3

[5] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once for all: Train one network and specialize it for efficient deployment. In *International Conference on Learning Representations*, 2020. 1

[6] Arslan Chaudhry, Marcus Rohrbach, Mohamed Elhoseiny, Thalaiyasingam Ajanthan, Puneet Kumar Dokania, Philip H. S. Torr, and Marc'Aurelio Ranzato. On tiny episodic memories in continual learning. *arXiv: Learning*, 2019. 2

[7] Matthias De Lange, Rahaf Aljundi, Marc Masana, Sarah Parisot, Xu Jia, Aleš Leonardis, Gregory Slabaugh, and Tinne Tuytelaars. A continual learning survey: Defying forgetting in classification tasks. *IEEE transactions on pattern analysis and machine intelligence*, 44(7):3366–3385, 2021. 1

[8] Fabrizio De Vita, Giorgio Nocera, Dario Bruneo, Valeria Tomaselli, and Mirko Falchetto. On-device training of deep learning models on edge microcontrollers. In *2022 IEEE International Conferences on Internet of Things (iThings) and IEEE Green Computing Communications (GreenCom) and IEEE Cyber, Physical Social Computing (CPSCom) and IEEE Smart Data (SmartData) and IEEE Congress on Cybermatics (Cybermatics)*, pages 62–69, 2022. 2

[9] Ning Ding, Yujia Qin, Guang Yang, Fu Wei, Zonghan Yang, Yusheng Su, Shengding Hu, Yulin Chen, Chi-Min Chan, Weize Chen, Jing Yi, Weilin Zhao, Xiaozhi Wang, Zhiyuan Liu, Haitao Zheng, Jianfei Chen, Y. Liu, Jie Tang, Juanzi Li, and Maosong Sun. Parameter-efficient fine-tuning of large-scale pre-trained language models. *Nature Machine Intelligence*, 5:220–235, 2023. 2

[10] William Fedus, Prajit Ramachandran, Rishabh Agarwal, Yoshua Bengio, Hugo Larochelle, Mark Rowland, and Will Dabney. Revisiting fundamentals of experience replay. In *Proceedings of the 37th International Conference on Machine Learning*, pages 3061–3071. PMLR, 2020. 2

[11] Alexander Rainer Tassilo Gepperth and Barbara Hammer. Incremental learning algorithms and applications. In *The European Symposium on Artificial Neural Networks*, 2016. 2

[12] Tyler L. Hayes, Nathan D. Cahill, and Christopher Kanan. Memory efficient experience replay for streaming learning. *2019 International Conference on Robotics and Automation (ICRA)*, pages 9769–9776, 2018. 2

[13] Alex Krizhevsky. Learning multiple layers of features from tiny images, 2009. 6

[14] 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. 2, 4

[15] Yunsheng Li, Yinpeng Chen, Xiyang Dai, Dongdong Chen, Mengchen Liu, Lu Yuan, Zicheng Liu, Lei Zhang, and Nuno Vasconcelos. Micronet: Improving image recognition with extremely low flops. *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 458–467, 2021. 1

[16] Yinan Liang, Ziwei Wang, Xiuwei Xu, Yansong Tang, Jie Zhou, and Jiwen Lu. Mcuformer: Deploying vision tranformers on microcontrollers with limited memory. *Advances in Neural Information Processing Systems*, 36, 2024. 1

[17] Ji Lin, Wei-Ming Chen, Yujun Lin, Chuang Gan, and Song Han. Mcunet: Tiny deep learning on iot devices. *Advances in Neural Information Processing Systems*, 33, 2020. 1

[18] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. On-device training under 256kb memory. In *Advances in Neural Information Processing Systems*, pages 22941–22954. Curran Associates, Inc., 2022. 2

[19] 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. 4

[20] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. On-device training under 256kb memory. 2022. 1, 2

[21] Vincenzo Lomonaco and Davide Maltoni. Core50: a new dataset and benchmark for continuous object recognition. In *Conference on Robot Learning*, 2017. 6

[22] Vincenzo Lomonaco, Lorenzo Pellegrini, Andrea Cossu, Antonio Carta, Gabriele Graffieti, Tyler L. Hayes, Matthias De Lange, Marc Masana, Jary Pomponi, Gido M. van de Ven, Martin Mundt, Qi She, Keiland W Cooper, Jérémy Forest, Eden Belouadah, Simone Calderara, German Ignacio Parisi, Fabio Cuzzolin, Andreas Savas Tolias, Simone Scardapane, Luca Antiga, Subutai Amhad, Adrian Daniel Popescu, Christopher Kanan, Joost van de Weijer, Tinne Tuytelaars, Davide Bacciu, and Davide Maltoni. Avalanche: an end-to-end library for continual learning. *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 3595–3605, 2021. 6

[23] Davide Nadalini, Manuele Rusci, Giuseppe Tagliavini, Leonardo Ravaglia, Luca Benini, and Francesco Conti. Pulp-trainlib: Enabling on-device training for risc-v multi-core mcus through performance-driven autotuning. In *International Conference on Embedded Computer Systems*, pages 200–216. Springer, 2022. 3, 5

[24] Davide Nadalini, Manuele Rusci, Luca Benini, and Francesco Conti. Reduced precision floating-point optimization for deep neural network on-device learning on microcontrollers. *Future Generation Computer Systems*, 149:212–226, 2023. 2, 3

[25] Arild Nøkland. Direct feedback alignment provides learning in deep neural networks. In *Neural Information Processing Systems*, 2016. 2

[26] Francesco Paissan, Alberto Ancilotto, and Elisabetta Farella. Phinets: a scalable backbone for low-power ai at the edge. *ACM Transactions on Embedded Computing Systems*, 21(5): 1–18, 2022. 1, 3

[27] Massimo Pavan, Eugeniu Ostrovan, Armando Caltabiano, and Manuel Roveri. Tybox: an automatic design and code-generation toolbox for tinyml incremental on-device learning. *ACM Transactions on Embedded Computing Systems*, 2023. 3

[28] Lorenzo Pellegrini, Gabriele Graffieti, Vincenzo Lomonaco, and Davide Maltoni. Latent replay for real-time continual learning. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 10203–10209. IEEE, 2020. 2, 3

[29] Leonardo Ravaglia, Manuele Rusci, Davide Nadalini, Alessandro Capotondi, Francesco Conti, and Luca Benini. A tinyml platform for on-device continual learning with quantized latent replays. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 11(4):789–802, 2021. 2, 3

[30] 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, 2021. 2

[31] Davide Rossi, Francesco Conti, Manuel Eggiman, Alfio Di Mauro, Giuseppe Tagliavini, Stefan Mach, Marco Guermandi, Antonio Pullini, Igor Loi, Jie Chen, Eric Flamand, and Luca Benini. Vega: A ten-core soc for iot endnodes with dnn acceleration and cognitive wake-up from mram-based state-retentive sleep mode. *IEEE Journal of Solid-State Circuits*, 57(1):127–139, 2022. 2, 6

[32] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115:211 – 252, 2014. 3

[33] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018. 3

[34] Jae-Su Song and Fangzhen Lin. Pocketnn: Integer-only training and inference of neural networks via direct feedback alignment and pocket activations in pure c++. *ArXiv*, abs/2201.02863, 2022. 2

[35] Bharath Sudharsan, Piyush Yadav, John G. Breslin, and Muhammad Intizar Ali. Train++: An incremental ml model training algorithm to create self-learning iot devices. In *2021 IEEE SmartWorld, Ubiquitous Intelligence Computing, Advanced Trusted Computing, Scalable Computing Communications, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/IOP/SCI)*, pages 97–106, 2021. 3

[36] Matteo Tremonti, Davide Dalle Pezze, Francesco Paissan, Elisabetta Farella, and Gian Antonio Susto. An empirical evaluation of tinyml architectures for class-incremental continual learning. *Pervasive and Resource-Constrained Artificial Intelligence Workshop at Pervasive Computing and Communications Conference*, 2024. 2, 3

[37] Liyuan Wang, Xingxing Zhang, Hang Su, and Jun Zhu. A comprehensive survey of continual learning: Theory, method and application. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1–20, 2024. 2

[38] Zifeng Wang, Zheng Zhan, Yifan Gong, Geng Yuan, Wei Niu, Tong Jian, Bin Ren, Stratis Ioannidis, Yanzhi Wang, and Jennifer Dy. Sparcl: Sparse continual learning on the edge. *Advances in Neural Information Processing Systems*, 35:20366–20380, 2022. 2, 4, 5, 8

[39] Lars Wulfert, Johannes Kühnel, Lukas Krupp, Justus Viga, Christian Wiede, Pierre Gembaczka, and Anton Grabmaier. Aifes: A next-generation edge ai framework. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2024. 1, 3

[40] Wei Zhou and Yiying Li. A fixed version of quadratic program in gradient episodic memory. *ArXiv*, abs/2107.07384, 2021. 2