TyBox: An Automatic Design and Code Generation Toolbox for TinyML Incremental On-Device Learning

MASSIMO PAVAN and EUGENIU OSTROVAN, Politecnico di Milano, Italy
ARMANDO CALTABIANO, Truesense s.r.l., Italy
MANUEL ROVERI, Politecnico di Milano, Italy

Incremental on-device learning is one of the most relevant and interesting challenges in the field of Tiny Machine Learning (TinyML). Indeed, differently from traditional TinyML solutions where the training is typically carried out on the Cloud and inference only occurs on the tiny devices (e.g., embedded systems or Internet-of-Things units), incremental on-device TinyML allows both the inference and the training of TinyML models directly on tiny devices.

This ability paves the way for TinyML-enabled intelligent devices that can learn directly on the field and adapt to evolving environments, different working conditions, or specific users. The literature in this field is quite limited with very few solutions focusing only on the incremental fine-tuning of machine learning models, whereas a general solution encompassing algorithms and code generation for incremental on-device TinyML is still perceived as missing.

The aim of this article is to introduce, to the best of our knowledge for the first time in the literature, a toolbox called *TyBox* for the automatic design and code generation of incremental on-device TinyML classification models. In more detail, starting from a "static" TinyML model, TyBox is able to (i) automatically design the "incremental" on-device version of the TinyML model that has been suitably designed to take into account the technological constraint on the RAM memory of the target tiny device, and (ii) autonomously provide the C++ codes and libraries to support the inference and learning of the incremental on-device TinyML model directly on the tiny devices.

TyBox has been extensively compared with a state-of-the-art incremental learning solution for TinyML and tested on an off-the-shelf tiny device (i.e., the Arduino Nano 33 BLE) in three relevant TinyML application tasks and scenarios: binary image classification, multi-class image classification, and ultra-wide-band human activity recognition. In addition, TyBox is released to the scientific community as a public repository.

CCS Concepts: • **Computing methodologies** → **Neural networks**; **Supervised learning**; • **Computer systems organization** → **Embedded systems;**

Additional Key Words and Phrases: Tiny Machine Learning, incremental learning, interpreters and code generation for tiny systems, TinyML software generation and mapping

This work was supported by the PNRR-PE-AI FAIR project funded by the NextGeneration EU program.

Authors' addresses: M. Pavan, E. Ostrovan, and M. Roveri, Dipartimento di Elettronica, Informazione e Bioingegneria - DEIB, Politecnico di Milano, Via Giuseppe Ponzio, 34, 20133 Milano MI, Italy IT; emails: massimo.pavan@polimi.it, eugeniu.ostrovan@mail.polimi.it, manuel.roveri@polimi.it; A. Caltabiano, Truesense s.r.l., Via Quadrio Maurizio, 20 20154 Milan MI, Italy IT; email: armando.caltabiano@truesense.it.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

1539-9087/2024/05-ART42 $15.00
https://doi.org/10.1145/3604566

ACM Reference format:
Massimo Pavan, Eugeniu Ostrovan, Armando Caltabiano, and Manuel Roveri. 2024. TyBox: An Automatic Design and Code Generation Toolbox for TinyML Incremental On-Device Learning. *ACM Trans. Embedd. Comput. Syst.* 23, 3, Article 42 (May 2024), 27 pages.
https://doi.org/10.1145/3604566

1 INTRODUCTION

Tiny Machine Learning (TinyML) is paving the way for the pervasive diffusion of smart objects and devices in everyday life. To achieve this ambitious goal, TinyML aims at introducing novel machine and deep learning models and algorithms that are able to be directly executed on tiny devices (e.g, embedded and **Internet-of-Things (IoT)** units), hence bridging the gap between the high computational and memory demands typically required by machine and deep learning algorithms and the severe technological constraints on memory, computation, and energy characterizing tiny devices [9, 45].

A relatively wide amount of literature exists in this field [12, 40], and solutions aim at either introducing tiny architectures for machine and deep learning models (e.g., lightweight versions of **Convolutional Neural Networks (CNNs)**) or approximate computing strategies to reduce the memory and computational demand (e.g., quantization or pruning mechanisms). Interestingly, current solutions assume a technological decoupling between training and inference of the TinyML models. Indeed, having orders of magnitude more memory and being computationally demanding, the training of TinyML models is typically assumed to be carried out in the Cloud where appropriate computing and memory resources are available, whereas inference only is executed on the target tiny devices.

Unfortunately, this approach prevents TinyML solutions from being incrementally trained or adapted during their operational life by exploiting fresh information coming from the field.

For these reasons, incremental on-device TinyML is one of the most promising and challenging research fields in TinyML. Indeed, such an incremental on-device learning would allow (i) making TinyML solutions adaptive over time to deal with additional tasks while they are operating (e.g., a new gesture recognition command should be learned by the TinyML applications); (ii) support in fine-tuning of TinyML models on specific users or settings (e.g., a set of vocal commands is pre-trained on generic users, whereas the final user fine-tunes the model with his or her own specific voice); and (iii) dealing with concept drift that could potentially occur in the process generating the data (e.g., a person recognition TinyML device, trained to operate in indoor conditions, is moved outdoors).

Interestingly, the literature in the incremental learning field that targets tiny COTS (commercial-off-the-shelf) devices (i.e., <1 MB of SRAM, usually few hundreds of kilobytes [12]) is quite limited, with only a few examples addressing specific aspects of the problem [10, 36, 39, 43] (see Section 2 for the analysis of the related literature), whereas a comprehensive perspective integrating design, development, and deployment of incremental on-device TinyML solutions is still missing.

The aim of this article is to introduce *TyBox*, a TinyML toolbox for the automatic design and code generation of incremental on-device TinyML models. TyBox operates on both machine and deep learning models by receiving in input a “static” version of a TinyML model and the technological constrain of the available RAM memory of the target tiny device, and it automatically designs the incremental version of the TinyML model able to satisfy the constraints on the RAM memory, as well as autonomously provides the C++ codes and libraries for the training and inference of this incremental TinyML model to be directly inserted in the firmware and executed on the target tiny devices. TyBox currently supports supervised learning models for classification

tasks, but we plan to actively continue support for the toolbox with the addition of new types of models for different tasks.

Summarizing, the novel content brought in this article is threefold:

- a solution for the automatic design of incremental on-device TinyML models able to take into account the technological constraint on the available RAM memory on the target device,
- an automatic code generation mechanism able to automatically generate the C++ code and libraries for the training and inference of incremental TinyML models directly on the target device, and
- incremental on-device TinyML models able to handle multi-class classification problems.

TyBox has been successfully compared with a state-of-the-art solution for incremental TinyML and extensively tested on an off-the shelf tiny device (i.e., the Arduino Nano 33 BLE sense, equipped with a nRF52840 MCU whose maximum power consumption is <20 mW) in three application scenarios: binary image classification, multi-class image classification, and **Ultra-Wide-Band (UWB)** human activity recognition. In addition, TyBox is made available to the scientific community as a public GitHub repository.[1]

The article is organized as follows. Section 2 introduces the related literature. Section 3 provides an overview of the proposed TyBox toolbox, whereas Sections 4 and 5 detail the automatic design and the automatic code generation of incremental on-device TinyML models, respectively. Section 6 describes the experimental results, whereas challenges and opportunities of on-device learning are analyzed in Section 7. Our conclusion and future works are presented in Section 8.

2 RELATED LITERATURE

In the field of TinyML, most of the solutions presented in the literature focus either on lightweight architectures for machine and deep learning (characterized by reduced computational and memory demands) or on approximate computing mechanisms (e.g., quantization and pruning) to reduce the computational and memory demands.

MobileNet [24] and SqueezeNet [25] are well-known examples of CNN tiny models specifically designed to reduce computational and memory demand. Such deep learning models rely on tiny architectures or suitably defined convolutional filters to account for the severe technological constraints characterizing tiny devices. Further examples in this field include the work of Tan and Le [44] and Alippi et al. [8].

Quantization, pruning, and early-exit neural networks are examples of approximate computing mechanisms to reduce the computational and memory demands of machine and deep learning models. In more detail, quantization [21] allows reduction of the memory and computational demand by reducing the number of bits used for the weight and activation representation (e.g., scaling from a 32-bit floating point to an 8-bit integer). Differently, pruning [28] aims at removing some tasks (e.g., layers or filters) from the processing pipeline of the machine and deep learning models at design time, hence saving the memory and computational needs accordingly. Finally, early-exit neural networks [14, 42] allow skipping the execution of some processing layers at runtime. More specifically, by exploiting the ability of neural networks to learn features characterized by increasing complexity and meaning, early-exit neural networks can incrementally process the input and provide the output even at intermediate classification steps—that is, when enough confidence about the result is achieved (hence saving the computational demand of processing layers that are not executed).

[1] https://github.com/pavmassimo/TyBox.

Interestingly, most of the TinyML solutions presented in the literature focus only on the "inference" of TinyML models on tiny devices, whereas the "training" is assumed to take place on powerful enough computing units (e.g., a Cloud system or computing systems). Very few solutions for on-device training of TinyML models are available in the literature. These solutions can be grouped into two main families according to the type of TinyML models supporting the on-device training: machine learning and deep learning models. These two families of solutions are described in what follows. In addition, in the last part of the section, we will explore machine and deep learning solutions able to deal with concept drift—that is, changes in the process generating the data over time.

We emphasize that, currently, none of the incremental TinyML solutions present in the literature are endowed with a toolbox supporting the automatic design or code generation of the designed solution. In addition, most of the current incremental on-device TinyML models operate only on the binary classification problem.

On-Device Training of TinyML Machine Learning Models. The on-device TinyML training solutions for machine learning models present in the literature mainly focus on incremental learning and learning in the presence of concept drift. For example, Disabato and Roveri [16] proposed a solution addressing the concept drift problem and adapting the algorithms on the basis of new knowledge that is made available from the field, by integrating a deep convolutional feature extractor and a KNN classifier. Due to the nature of the KNN algorithm, the adaptation phase of the model consists of adding the new extracted features/label pairs to the training dataset. The same approach was used in another work by Disabato and Roveri [15] for the incremental case. Differently, Sudharsan et al. [43] introduced a specific not-deep classification algorithm for binary classification problems, called *Train++*. Interestingly, up to now, this is the only work in the literature that has made available the dataset used in the experiments, hence having reproducible results. For this reason, the dataset of Train++ was used also in this work to compare the results of TyBox with the ones of another state-of-the-art solution. Finally, in the field of machine learning, Benatti et al. [10] introduced a multi-class classification approach based on hyperdimensional computing. Unfortunately, this approach cannot be applied to deep learning architectures.

On-Device Training of TinyML Deep Learning Models. The on-device TinyML training solutions for deep learning models present in the literature mainly focus on transfer learning [35]. For example, Cai et al. [11] proposed to learn only the biases of a deep CNN to reduce the memory demand dedicated to activations during training. Another interesting approach to achieve on-device learning on embedded devices was proposed in the work of Pellegrini et al. [34] and Ravaglia et al. [37]. Such an approach relies on latent representation (i.e., the activations of training data at a given point of the neural network) along with new data to partially retrain the neural network to mitigate the well-known effect of catastrophic forgetting [32]. Despite the potential reliability of this approach, the total amount of memory used by the learning algorithms proposed in these works is in the order of tens of megabytes, making them not suitable for embedded systems or IoT units.

Differently, Ramanathan [36] and Ren et al. [39] suggest keeping fixed the feature extraction part of their neural networks and retraining only the last layer. Both works focus on specific binary classification problems (i.e., presence detection and anomaly detection, respectively) and do not provide publicly available code nor datasets (hence lacking in reproducibility).

Recently, an interesting solution for the design of on-device learning solution on embedded devices was proposed by Lin et al. [27]. Such a solution focuses on reducing the memory footprint by applying sparse updates to skip the computation of the gradient for a large part of the network's weights, but differently from what is proposed here, it does not take into account the technological constraints on the available RAM of the tiny devices.

We emphasize that, currently, no incremental TinyML deep learning solutions for multi-class classification problems able to take into account technological constraints of tiny devices (i.e., devices with an available amount of RAM < 1 MB) are available in the literature.

Learning in the Presence of Concept Drift. This relevant research area aims at designing machine and deep learning models able to deal with changes in the data-generating process. This area represent an important field in TinyML since tiny devices (e.g., IoT and embedded systems) typically operate in real-world environments that might change their statistical behavior over time (due to periodicity or seasonality effects, thermal drift, or changes in the users' habits or behavior). In more detail, concept drift occurs when the statistical properties of the data change over time, resulting in a deviation from the original concept on which the model was trained [17]. In this context, the learning phase requires continuously updating the trained model so as to adapt it to the changing distribution of data [29, 46].

The literature in this field converged to two core perspectives: active and passive solutions. The former solutions have the goal of actively detecting the occurrence of a concept drift, by analyzing the accuracy of the model or the statistical properties of the data, and adapting the models only accordingly [6, 7, 19]. The latter solutions focus on continuously adapting the model on the incoming data without explicitly detecting the concept drift [33]. Although there exist some works on on-device training, very few of them address the problem from the point of view of learning in the presence of concept drift, and currently these works can be categorized as passive solutions [37, 39]. We emphasize that the approach used in this work can also be categorized as passive, making TyBox able to continuously adapt to changing conditions without requiring explicit change detection.

Despite being effective with both gradual and abrupt concept drift, passive solutions intrinsically suffer from the "catastrophic forgetting" issue [32]—that is, a machine or deep learning model is not able to retain the previously acquired knowledge while learning from the newly available data. From this perspective, learning and forgetting represent two sides of the same coin, and their relationship strictly depends on the variability of the data-generating process. Indeed, in incremental learning [20] (a passive solution setting in which the learning process takes place whenever new examples emerge and adjusts what has been learned according to the new examples), all of the useful previously acquired knowledge must be retained during the training of new data, whereas generally in the presence of concept drift, forgetting obsolete knowledge is crucial to quickly adapting to new working conditions of the process generating the data. The solutions presented in the literature to deal with catastrophic forgetting can be grouped into three main families [13]: knowledge replay [38] (among which latent replay is a notable "edge version"), regularization based [26], and parameter isolation [30]. These families of solution differ in the strategy used to avoid the forgetting of information: in knowledge replay, old data are used along with the newly collected ones to retrain the network; in regularization-based strategies, an extra regularization term is introduced in the loss function, consolidating previous knowledge when learning on new data; and finally, in parameter isolation, different model parameters are dedicated to each task to prevent any possible forgetting of previously learned information.

3 THE TYBOX TOOLBOX: AN OVERVIEW

The TyBox toolbox aims at automatically designing incremental TinyML models and generating the C++ code and libraries for the training and inference of these models directly on the tiny devices. An overview of TyBox is given in Figure 1, where the inputs, the two main modules, and the outputs of TyBox are detailed.

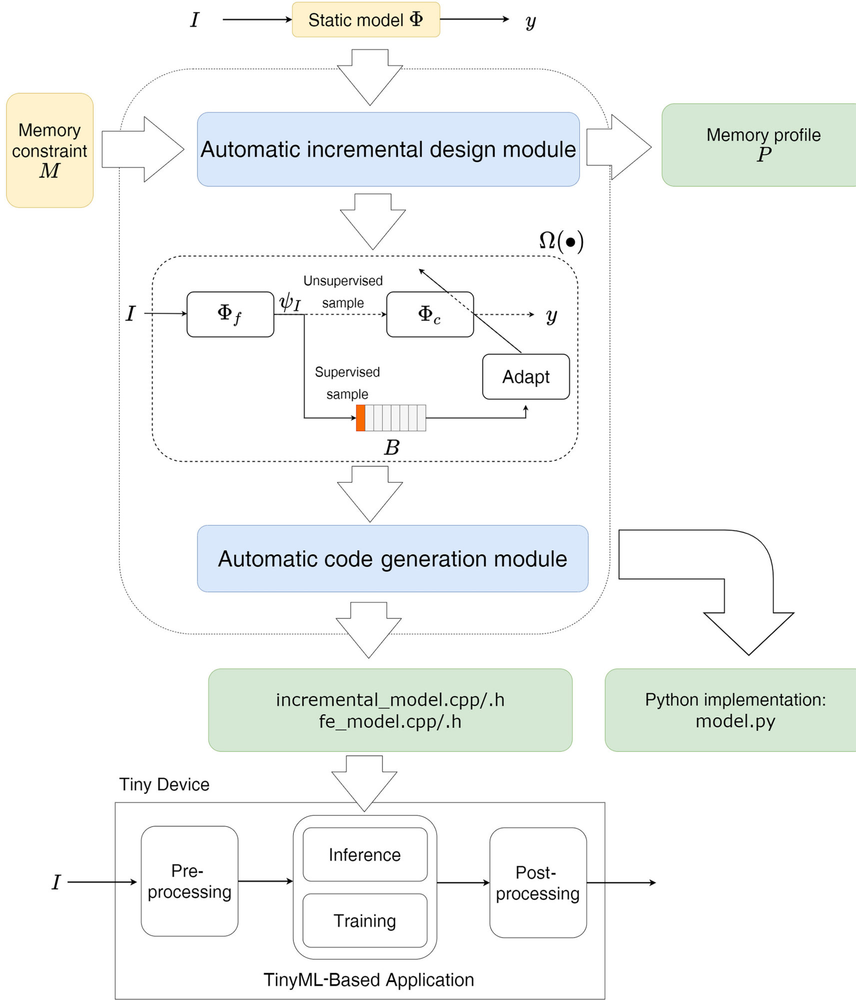

Fig. 1. Overview of TyBox. The required inputs are highlighted in yellow, the two modules composing TyBox are highlighted in blue, and the produced outputs are highlighted in green.

More specifically, TyBox is designed to receive the following in input:

- a "static" (i.e., not incremental) TinyML model $y = \Phi(I)$, where I is the input and y is the output, and
- the technological constraint M on the on-device RAM that must be satisfied by the designed incremental TinyML solution (in both the inference and training phases).

Currently, TyBox supports two different types of TinyML models: **Feed-Forward Neural Networks (FFNNs)** and CNNs, representing two well-known and widely used models in the field of machine learning and deep learning [23], but what here described can be easily extended to other families of machine and deep neural networks.

We emphasize that TyBox is designed to receive in input the TinyML model (either FFNN or CNN) in **TensorFlow (TF)** file format [5]. In the case of CNNs, the TinyML model is assumed to be preliminarily trained on a reference dataset and the on-device incremental learning considers

the pre-trained CNN as weight initialization. Differently, in the case of FFNNs, the TinyML model can be either pre-trained on a reference dataset or trained incrementally from scratch directly on the tiny device. These aspects will be deepened in Section 4. We also emphasize that we currently focus on TF because **TensorFlow Lite for Micro (TFLM)** represents one of the most widely used frameworks for TinyML. Nonetheless, TyBox can be easily extended to operate on different frameworks by modifying its automatic code generation module.

As depicted in Figure 1, TyBox is composed of the following two modules:

- The *automatic incremental design module* receives in input $\Phi(\bullet)$ and M and automatically designs $\Omega(\bullet)$ (i.e., the incremental version of $\Phi(\bullet)$), consisting of a *fixed convolutional feature extraction block* $\Phi_f(\bullet)$ (in the case of CNNs), an *incrementally learnable classification block* $\Phi_c(\bullet)$, and a *buffer* B. Two main actions are carried out by this module: partitioning $\Phi(\bullet)$ into $\Phi_f(\bullet)$ and $\Phi_c(\bullet)$ and sizing the buffer B to maximize the amount of stored data while respecting the constraint on RAM M. This model is detailed in Section 4.
- The *automatic code generation module* receives in input $\Omega(\bullet)$ and generates the C++ codes and library implementing the inference of $\Phi_c \circ \Phi_f$ and the incremental on-device training of $\Phi_c(\bullet)$. These automatically generated files are designed to be directly embedded in the firmware of the target tiny device and used by the TinyML application. This module is detailed in Section 5.

Finally, the output of TyBox is composed of the following three main outcomes:

- the *.cpp* and *.h* files implementing $\Omega(\bullet)$, and as shown in Section 5.2, these files can be directly integrated into the firmware of the tiny device to support the on-device incremental learning and the inference of $\Phi_c(\bullet)$;
- the *.py* version of the *.cpp* and *.h* files implementing $\Omega(\bullet)$ to emulate the on-device inference and learning in Python; and
- a *profiler report* P detailing the memory occupation and number of operations for all processing layers of $\Omega(\bullet)$.

The two modules implementing TyBox are detailed in the next two sections.

4 TYBOX: THE AUTOMATIC INCREMENTAL DESIGN MODULE

The aim of this section is to describe the automatic incremental design module of TyBox. From the incremental learning perspective, an *algorithmic* and a *technological* challenge need to be jointly addressed to support an effective and efficient on-device training of TinyML models.

The algorithmic challenge concerns the tradeoff between the ability to incrementally learn TinyML models with new data coming from the field might come and a possible forgetting of previously acquired knowledge. This issue is known in the literature as “catastrophic forgetting” as described in Section 2.

Differently, the technological challenge concerns the fact that the incremental learning of TinyML models requires an increased memory and computational demand on tiny devices. The source of this increased demand is twofold. First, the learning algorithms (i.e., in our case, the backpropagation algorithm) typically require a higher computational and memory demand than the inference due to the need to update the weights and store intermediate results. Second, an additional buffer is required to store the samples used to incrementally train the TinyML models. The larger the buffer, the better the learning abilities at the expense of a larger memory demand.

These two challenges are jointly addressed in TyBox. On one hand, a suitably defined memory buffer B storing supervised samples coming from the field is used to mitigate the “catastrophic forgetting” effect by retraining the incremental model on all samples in B every time a new

supervised sample becomes available. On the other hand, to reduce the memory and computational demands, the automatic incremental design module of TyBox decouples $\Phi(\bullet)$ into its two main components—the feature extractor $\Phi_f(\bullet)$ and the classifier $\Phi_c(\bullet)$—and only the latter is retrained on the device.[2] $\Phi_f(\bullet)$ and $\Phi_c(\bullet)$ are assumed to be separated by a flatten layer, which is the point of the model where $\Phi(\bullet)$ is split. This approach also allows storing in the buffer B only the features extracted by $\Phi_f(\bullet)$ and not the original input data I, hence also reducing the memory demand of the buffer (assuming that the dimension of the feature extracted by Φ_f is smaller than the one of I). The drawback of such an approach is that $\Phi_f(\bullet)$ is not incrementally trained over time. This drawback is partially mitigated by the fact that the first convolutional layers of a CNN are typically characterized by coarse-grained extracted features, and these features can be considered as general-purpose feature extractors that can be applied to different CNN tasks [16].

The incremental learning algorithm in TyBox operates as follows. Every time a new supervised sample is received, $\Phi_c(\bullet)$ is trained on all data present in B for γ epochs with batches of $n = 1$ sample.[3] The considered learning algorithm is the backpropagation algorithm [41]. In the experiments described in Section 6, γ has been set to 1 to minimize the computational demand of the incremental learning phase carried out on the device. We emphasize that $n = 1$ allows reduction of the memory demand for storing the activations required to compute the backpropagation, whereas retraining over the entire buffer B ensures that $\Phi_c(\bullet)$ is trained every time on a set of recently acquired samples (hence reducing the risk of catastrophic forgetting). Further information on the sizing and the operational mode of B are provided in Section 4.3.

The automatic incremental design module of TyBox is implemented through the following three steps:

(1) partitioning $\Phi(\bullet)$ into $\Phi_f(\bullet)$ and $\Phi_c(\bullet)$,
(2) memory characterization of $\Phi_f(\bullet)$ and $\Phi_c(\bullet)$, and
(3) sizing buffer B.

These three steps are detailed in the rest of the section, and the output of this module is $\Omega(\bullet)$, which consists of the designed incremental version of $\Phi(\bullet)$ and the buffer B. Without any loss of generality, in the rest of the article we assume that $\Phi(\bullet)$ is a CNN.[4]

4.1 Partitioning Φ into Φ_f and Φ_c

Let $I \in \mathbb{R}^{n \times m \times c}$, where $n, m, c \in \mathbb{N}$, be an input tensor data with m rows, n columns, and c channels, and let $\Phi(\bullet)$ be a CNN to be deployed on the tiny device. $\Phi(\bullet)$ is assumed to include a flatten layer separating $\Phi_f(\bullet)$ and $\Phi_c(\bullet)$, where the former consists of the convolutional feature extraction layers and the latter the final classification layers. In particular, $\Phi_c(\bullet)$ is assumed to be a dense feed-forward network consisting of an input layer, one or more hidden layers, and an output (final classification) layer. All layers are characterized by an activation function, and, currently, the activation functions supported by TyBox are ReLU, sigmoid and softmax.

I is processed by $\Phi_f(\bullet)$ by extracting a feature vector ψ_I of size $|\psi_I|$, with $|\bullet|$ being the cardinality operator. ψ_I is then used as input for $\Phi_c(\bullet)$, which produces the final classification $y = \Phi_c(\psi_I)$.

4.2 Memory Characterization of Φ_f and Φ_c

The second step of the automatic incremental design module aims at computing the memory demand to store the weights and the activations of all processing layers in $\Phi_f(\bullet)$ and $\Phi_c(\bullet)$. This is

[2] $\Phi(\bullet)$, $\Phi_f(\bullet)$ and $\Phi_c(\bullet)$ are stored as TF model Python variables.
[3] In principle, the dimension of the batches could be increased, although at the expense of the layer's memory and computational demands.
[4] FFNNs can be modeled as CNNs where $\Phi_f(\bullet)$ is the identity function and the feed-forward layers are stored in $\Phi_c(\bullet)$.

a crucial step in being able to properly size the buffer B while respecting the memory constraint M on the available on-device RAM. This aspect will be deepened in Section 4.3.

With K being the total number of processing layers in $\Phi(\bullet)$ and with $\Phi_f(\bullet)$ and $\Phi_c(\bullet)$ being the partitions computed as described in Section 4.1, we assume that, without any loss of generality, $\Phi_f(\bullet)$ consists of layers from 1 to l and $\Phi_c(\bullet)$ the ones from $l+1$ to K. Let L_k with $k = 1, \ldots, K$ be a processing layer, and the core of this second step is to characterize the memory requirements of L_k both in terms of weights θ_k and activations a_k (i.e., the output of the L_k layer). L_0 is defined as the input layer, which only stores the input data I, hence $|\theta_0| = 0$ and $|a_0| = |I|$.

Currently, the families of processing layers supported in TyBox are the following:

- *2D convolutional layer*: Assuming L_k to be a 2D convolutional layer characterized by f $r \times s$ convolutional filters and S_x and S_y being the stride along dimension x and y, the cardinality of parameters $|\theta_k|$ and activations $|a_k|$ can be computed as follows:

$$|\theta_k| = (r \cdot s \cdot t + 1) \cdot f$$
$$|a_k| = \left(\frac{m-r}{S_x} + 1\right) \cdot \left(\frac{n-s}{S_y} + 1\right) \cdot f,$$

  where m, n, and t represent the rows, columns, and channels of a_{k-1}, respectively.
- *Dense layers*: This family of layers creates a dense connection between an input layer (whose dimension corresponds to the activation of the previous layer) and an output layer (whose number of output units is set by the designer). More specifically, when L_k is a dense layer, $|\theta_k|$ and $|a_k|$ depend only on the dimensions of the activation of the $k-1$-th layer $|a_{k-1}|$ and the number of output units $|o_k|$ as follows:

$$|\theta_k| = |a_k| \cdot (|a_{k-1}| + 1)$$
$$|a_k| = |o_k|.$$

- *Activation layers*: The activation layers currently implemented in TyBox are the following: ReLU, softmax, and sigmoid. Since these layers only perform mathematical operations on the data, they do not require the need to store any parameter nor change the activation dimensions with respect to the input layer. More specifically, when L_k is an activation layer, $|\theta_k|$ and $|a_k|$ can be computed as follows:

$$|\theta_k| = 0$$
$$|a_k| = |a_{k-1}|.$$

- *Pooling layers*: This family of layers allows one to reduce the dimensionality of the activations while not introducing any parameters to be stored. More specifically, with L_k being an $x \times y$ max pooling or average pooling layer where x and y represent the dimensions of the filter, $|\theta_k|$ and $|a_k|$ can be computed as follows:

$$|\theta_k| = 0$$
$$|a_k| = |a_{k-1}|/(x \cdot y).$$

Table 1 summarizes the equations introduced in TyBox to compute the dimensions of weights and activations of each processing layer in either $\Phi_f(\bullet)$ or $\Phi_c(\bullet)$. The corresponding memory demands to store the parameters m_k^θ and the activations m_k^a can be easily computed as follows:

$$m_k^\theta = |\theta_k| \cdot M_w$$
$$m_k^a = |a_k| \cdot M_w,$$

Table 1. Cardinality of Parameters and Activations for the Different Types of Layers Considered in TyBox

Type of Layer	Cardinality of Parameters $\lvert\theta_k\rvert$	Cardinality of Activations $\lvert a_k\rvert$
Conv2D	$(r \cdot s \cdot t + 1) \cdot f$	$(\frac{m-r}{S_x} + 1) \cdot (\frac{n-s}{S_y} + 1)$
Dense	$\lvert a_k\rvert(\lvert a_{k-1}\rvert + 1)$	$\lvert O\rvert$
Activation	0	$\lvert a_{k-1}\rvert$
Pooling	0	$\lvert a_{k-1}\rvert/(x \cdot y)$

where M_w is the dimension in bytes required to store a single value (either weight or activation). For the purposes of this work, M_w has been set to 4 since weights and activations are stored as 32-bit floating point values. Here, quantization techniques could be considered to further reduce the memory demand of weights and activations, and, in this direction, the next extensions of TyBox will consist of quantization mechanisms by also differentiating the value of M_w associated to the weights from that of the activations.

The total amount of memory M_{Φ_f} and M_{Φ_c} required for the weights and activations of $\Phi_f(\bullet)$ and $\Phi_c(\bullet)$, respectively, can be computed as follows:

$$M_{\Phi_f} = \sum_{k=0}^{k<=l} m_k^{\theta} + \max_{k=0}^{k<l} \left(m_k^a + m_{k+1}^a\right), \tag{1}$$

$$M_{\Phi_c} = \sum_{k=l+1}^{k<=K} m_k^{\theta} + m_k^a. \tag{2}$$

We emphasize that $\Phi_f(\bullet)$ is not trained incrementally and only the forward pass is performed in this part of the network. Hence, there is no need to save the activations for these layers. In addition, M_{Φ_f} relies on the tensor arena optimized by the TFLM interpreter that is able to reuse the same allocated memory for each couple of consecutive layers.[5] Differently, being incrementally trained, M_{Φ_c} cannot be optimized in the same way, as all of the intermediate activations are required to support the backpropagation algorithm.

4.3 Sizing Buffer B

The role of buffer B is crucial to support the incremental learning of $\Phi_c(\bullet)$. Interestingly, since our incremental solution retrains only $\Phi_c(\bullet)$, it is not necessary to store the incoming input samples Is but only the activations ψ_Is that are produced by $\Phi_f(\bullet)$ (together with the corresponding supervised information). This concept is similar to the latent replays mentioned in the related literature, here resulting in a more efficient use of the buffer.

Differently from the previous works on this topic, TyBox does not keep data from the original training set in the buffer. Indeed, TyBox is able to store and exploit data arriving over time after the device is deployed. In addition, in the original works, latent replays are used to mitigate the effects of catastrophic forgetting, whereas in TyBox, they are also exploited to limit the memory required by the backpropagation algorithm. In fact, TyBox does not retrain the network with batches of replays, but it performs backpropagation each time a new supervised sample arrives, hence allowing TyBox to reuse the same space for the activations of every datum.

[5] The actual amount of memory that TFLM requires may vary on the basis of the given target device. Nevertheless, this is the best estimate we can provide for this value, and in our experiment with the Arduino Nano 33 BLE, variations with regard to the actual values are less than 1 KB.

We emphasize that to trade off catastrophic forgetting, memory demand, and learning ability, TyBox relies on batches consisting of only one single datum and one single training epoch. A different approach could have led to use multiple epochs over a single datum and then discarding it, similarly to what happens in online learning. This option was not employed in TyBox because it is quite difficult to balance an appropriate amount of epochs with the potential occurrence of overfitting.

Within TyBox, the dimension of the buffer M_B (in terms of bytes) and the number N_B of vectors of extracted features (that can be stored on the device) are automatically computed on the basis of M and the memory demand for weights and activations computed in Equations (1) and (2). More specifically, the amount of memory M_B allocated to the buffer B is calculated as

$$M_B = \left\lfloor \frac{M}{M_w} \right\rfloor \cdot M_w - M_{\Phi_f} - M_{\Phi_c}, \tag{3}$$

whereas N_B is computed as

$$N_B = \frac{M_B}{M_w \cdot |\psi_I|}. \tag{4}$$

The buffer is filled with a first-in-first-out policy.

5 TYBOX: THE AUTOMATIC CODE GENERATION MODULE

The automatic code generation module of TyBox aims at transforming the incremental solution $\Omega(\bullet)$ designed in the former module into C++ code and libraries to be directly ported to the target tiny devices. We emphasize that $\Omega(\bullet)$ is organized into the following:

- two TF models $\Phi_f(\bullet)$ and $\Phi_c(\bullet)$ stored as Python variables (`model_fe, model_incr`), and
- the memory specification $|\psi|$ (modeled by the constant `FEATURE_DIM` in the code) and N_B (modeled by the constant `BUFFER_SIZE` in the code).

The rest of the section aims at describing the automatic code generation step for tiny devices (Section 5.1), the comprehensive on-device TinyML application integrating on-device inference and training (Section 5.2), and the automatic code generation step for Python projects and the generation of the report profile on the memory occupations (Section 5.3).

5.1 Automatic Code Generation for Tiny Devices

In TyBox, the *automatic code generation for tiny devices* is carried out by the code generation Python function

```
TyBox.create_incr_solution(model_fe, model_incr,
    FEATURE_DIM, BUFFER_SIZE),
```

which, by receiving as input $\Omega(\bullet)$, creates the following two couples of files:

- `incremental_model.h, incremental_model.cc`
- `fe_model.h, fe_model.cc`

The first couple of files stores the data structures of $\Phi_c(\bullet)$ and B, and the functions supporting the inference and incremental training of $\Phi_c(\bullet)$. In more detail, in this couple of files, every variable is allocated at compile time such that the memory occupation of all data structures can be *a priori* computed. The second couple of files refers to fact that TyBox relies on TFLM to support the execution of $\Phi_f(\bullet)$. Hence, the automatic code generation script of TyBox also produces the .h and .cc files to be inserted in the firmware as a standard TFLM model implementing $\Phi_f(\bullet)$.

More specifically, the `create_incr_solution()` function reads the weights and the biases of $\Phi_c(\bullet)$ directly from the TF model `model_incr` by using the TF API and stores them in the data structure `t_dense_network` present in the `incremental_model.c` file. Furthermore, it initially converts `model_fe` into a .tflite model by using the `convert()` function of the TFLiteConverter and, subsequently, into the `fe_model.cc` file by using the `-xxd` shell command for translating the weights into the hex format as requested by the `MicroInterpreter` of TFLM.

Since the `fe_model.cc` file is generated by using the standard TFLM pipeline, we will now focus on `incremental_model.cc`. More specifically, the data structures for $\Phi_c(\bullet)$ and B are reported in the following fragment of C++ code:

```
//An example of the specification of a network with
//one input layer and one fully connected layer.
typedef struct t_dense_network{
        float * bias_list[1];
        float * weight_list[1];
        float * layer_list[2];
        Activation *layer_activations[1];
    }t_dense_network;

//An example of the specification of the
//buffer in the MNIST experiment
const int BUFFER_SIZE = 100;
const int FEATURE_DIM = 200;
const int LABEL_DIM = 1;

float mbp_buffer[BUFFER_SIZE][FEATURE_DIM + LABEL_DIM];
```

The data structure `dense_netwotk` of $\Phi_c(\bullet)$ is designed to store both the bias and the weights for each layer of the neural network (`bias_list`, `weight_list`) as well as the activations of the inputs (`layer_list`), along with the implementation of the used activation function (`layer_activations`).

Furthermore, the `incremental_model.cc` file provides the following three functions to support the inference and learning of $\Phi_c(\bullet)$:

```
void forward_pass(const int & number_of_layers,
                const int * layer_sizes,
                t_dense_network & dense_network)

void get_label(const int &numOutputs,
            t_dense_network &dense_network)

void backpropagate(const int &numOutputs,
                int *targets, float lr,
                const int * layer_sizes,
                t_dense_network &dense_network)
```

More specifically, the `forward_pass` function receives in input only the `dense_network` in which the input features have already been computed by the TFLM `MicroInterpreter` and stored into the `layer_list[0]` variable. The `backpropagate` function performs the backpropagation step directly on the tiny device by assuming that the `forward_pass` has already been carried out

and that the activations for all layers are stored in `layer_activations`. The function receives in input the `dense_network`, the `targets` (expressed in one-hot encoding notation), and the learning rate `lr`. Finally, the function `get_label` is used to retrieve the results of the inference, and for this reason it only receives in input the `dense_network`.

5.2 On-Device TinyML Application

The on-device TinyML applications are typically organized into four main steps:

(1) data acquisition from sensors,
(2) data pre-processing to remove noise or highlight relevant features present in acquired data,
(3) inference of the TinyML model on pre-processed data, and
(4) the output of TinyML model, which post-processed to perform actuations.

Here, we focus on step 3 and highlight that TyBox, when supervised information is available, will be able to carry out both the inference and the incremental training by means of the `forward_pass` and `backpropagate` functions on the TinyML model stored in `incremental_model.h`.

We provide a fragment of the setup function, which includes the standard TFLM setup and the setup of the solution generated by TyBox:

```
static tflite::MicroErrorReporter micro_error_reporter;
error_reporter = &micro_error_reporter;
model = tflite::GetModel(g_model);
[...]
static tflite::AllOpsResolver resolver;
static tflite::MicroInterpreter static_interpreter(
  model, resolver, tensor_arena, kTensorArenaSize,
  error_reporter);
interpreter = &static_interpreter;
[...]
init_network(dense_network);
```

The TFLM setup includes the initialization of `error_reporter`, `model`, `AllOpsResolver`, and `MicroInterpreter`, whereas the setup of our solution consists only of the initialization of `dense_network`, which consists of the loading of the weights and biases present in the `incremental_model.c` file into the `dense_network_t` data structure.

Differently, during the main loop, the acquired and (possibly) pre-processed sample is loaded into `input->data.f`, which is the input buffer for the TFLM network,

```
// set m_f input and invoke TFLM interpreter
for(int input_node_index=0; input_node_index<INPUT_SIZE;
input_node_index++){
  input->data.f[input_node_index] =
    data[input_node_index];
}
```

and the `interpreter` is invoked as follows:

```
TfLiteStatus invoke_status = interpreter->Invoke();
```

If the input sample is provided with its supervised information (the label), both the sample and the label are stored in the `mbp_buffer` and the whole buffer is used for incrementally training `dense_network` by applying `forward_pass` and `backpropagate` functions.

If the input sample is not supervised, only the inference is performed by loading the data in `layer_list` of `dense_network`. The `forward_pass` function is then carried out, and the predicted label is retrieved by using the `get_label` function.

The following fragment of code details the on-device learning and inference of the automatically designed TinyML models by TyBox:

```
if(data_is_labelled){ //train on data
    // put features calculated by convolutional block
    // into the buffer
    store_in_buffer(output->data.f, mbp_buffer);

    // train on all data from buffer
    for(int i=0; i<max(buf_index, BUFFER_SIZE * is_full);
    i++){
        //load data from buffer
        for(int j=0; j<FEATURE_SIZE; j++){
            dense_network.layer_list[0][j] = mbp_buffer[i][j];
      }
    //forward
    forward_pass(number_of_layers, layer_sizes,
        dense_network);
    //backward
    backpropagate(numOutputs, targets, lr, layer_sizes,
        dense_network);
    }
}
else{ // evaluate data
    for(int input_node_index=0;
    input_node_index<FEATURE_SIZE; input_node_index++){
        float value = output->data.f[input_node_index];
        dense_network.layer_list[0][input_node_index] =
            value;
    }
    forward_pass(number_of_layers, layer_sizes,
        dense_network);
    int l = get_label(numOutputs, dense_network);
    [...]
}
```

5.3 Automatic Code Generation for Python and Profiler Output

In addition to the C++ code and libraries to be inserted in the firmware of tiny devices, TyBox generates the corresponding Python code for “off-device” evaluation and testing purposes (i.e., analysis that are carried out not on the device). We emphasize that this Python code is equivalent to the code for the tiny device. The file containing the equivalent Python implementation is called `model.py`, and contains the implementations of both $\Phi_f(\bullet)$ and $\Phi_c(\bullet)$.

Furthermore, to provide TyBox users with feedback on the designed network and its memory occupations, TyBox also provides an output file P reporting the following:

- θ_k and m_k^θ for each layer L_k,
- a_k and m_k^a for each layer L_k,
- $\sum_{k=0}^{k<=l} m_k^\theta$ and $\max_{k=0}^{k<l}(m_k^a + m_{k+1}^a)$,
- $\sum_{k=l+1}^{k<=K} m_k^\theta$ and $\sum_{k=l+1}^{k<=K} m_k^a$,
- M_{Φ_f} and M_{Φ_c},
- the total memory occupied by the network M_Φ, and
- the buffer parameters M_B and N_B.

This report file is a valuable tool to evaluate the effectiveness of the designed TinyML-based embedded application.

6 EXPERIMENTAL RESULTS

This section details all experiments carried out to validate the effectiveness and efficiency of the proposed TyBox toolbox.

The experimental campaign has been organized into three main experimental tasks for incremental on-device learning:

- image binary classification,
- image multi-class classification, and
- UWB human activity recognition.

A comparison with Train++ [43], a state-of-the-art non-deep TinyML incremental learning solution for binary classification, is provided for the image binary classification tasks; Train++ has been configured following the initialization and instructions provided in the work of Sudharsan et al. [43]. The comparison with Train++ has been included only in the image binary classification experiment since all other experiments consist of multi-class classification tasks. Differently, a comparison with the TF solution (not deployable on tiny devices) is carried out for the image multi-class classification and UWB human activity recognition experiments, as no multi-class TinyML incremental solution is currently available in the literature. In addition, all incremental solutions generated by TyBox have been ported on the Arduino Nano 33 BLE characterized by 256 KB of RAM memory, 1 MB of flash memory, and the nRF52840 microcontroller. Memory occupation and inference and training times (in microseconds) have been measured and reported.

The three experimental settings are detailed in the rest of the section. We emphasize that for the first experimental setting (i.e., the image binary classification experiment), we considered only one value of M since the considered benchmark is not particularly demanding in terms of memory occupation, whereas for the two remaining experimental settings, we considered two different values of M to explore different technological constraints for TyBox.

6.1 Image Binary Classification

6.1.1 Problem Definition. The first experimental setting concerns the incremental on-device learning of a machine learning model for image binary classification. This is the same setting of Sudharsan et al. [43], and the considered dataset refers to the Banknote authentication benchmark [2] consisting of images supporting the authentication procedure for banknotes.

6.1.2 Neural Network Architecture. Since the Banknote authentication dataset [2] already is composed of features based on wavelet transform that are extracted from the images, the considered incremental learning model $\Phi(\bullet)$ is an FFNN. Hence, the convolutional feature extractor $\Phi_f(\bullet)$ is not needed and $\Phi_c(\bullet)$ consists of one dense layer with softmax activation, where $|a_0| = 4$,

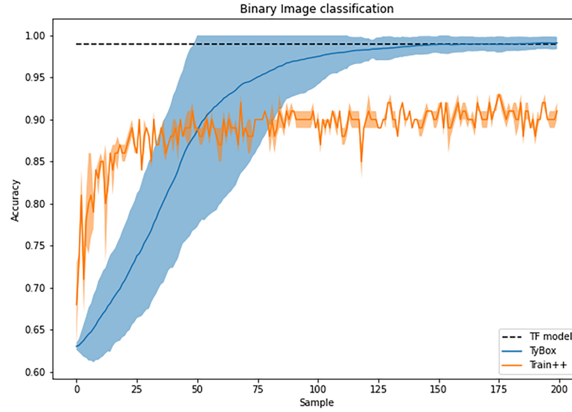

Fig. 2. The estimated accuracy for the image binary classification setting.

$|a_1| = 2$ and consequently $|\theta_1| = 10$. The learning rate ϵ of the backpropagation algorithm for the on-device training is set to 0.01.

6.1.3 Profiler and Buffer. By applying the first module of TyBox, the incremental version $\Omega(\bullet)$ of $\Phi(\bullet)$ is designed by considering a constraint $M = 142$ KB on the available RAM on the tiny device. Following Equations (1) and (2), the memory demand M_Φ of $\Phi(\bullet)$ is 64 bytes. Further details on M_Φ are provided later in Table 3. In addition, TyBox computes M_B and N_B with Equations (3) and (4), resulting in nearly 142 KB and 9,088 samples, respectively. In this experiment, the buffer is never fully filled, since the training set is smaller than 9,088 samples.

6.1.4 Experimental Result. For the purposes of the experiment, the Banknote dataset is split into training (75%) and testing (25%). The parameters of the Train++ model are randomly initialized, whereas for the incremental model designed by TyBox, the Glorot initialization is used [22]. The training samples are provided to the model designed by TyBox and the Train++ model one at a time for the incremental training. Once a new sample is provided, the accuracy of the two models is tested on the testing set.

In addition, as a further comparison, the accuracy of a model characterized by the same architecture of $\Phi(\bullet)$ but trained with a standard train-then-deploy approach on TF is provided. Such a TF model that is used for comparison is compiled with Adam as the optimizer, the loss function is the categorical cross entropy, and the learning rate is set to 0.03. The TF model was trained directly on the whole training set for 50 epochs, and the batch size is 32.

Experimental results are averaged over five runs with a random partitioning of training and testing of the dataset. The confidence intervals for these results are provided with a .95 confidence. Confidence intervals on the TF model are not reported in the figure for clarity. Figure 2 shows the classification accuracy on the image binary classification setting for TyBox, Train++, and the TF model. Three main comments arise:

- After the initial 50 samples, the incremental solution designed by TyBox provides a higher accuracy than Train++ thanks to the ability to manage more complex TinyML models.
- Train++ initially provides a higher accuracy since its model is simpler and requires less data to be trained, and for the same reason, Train++ models are characterized by a lower variance until convergence.

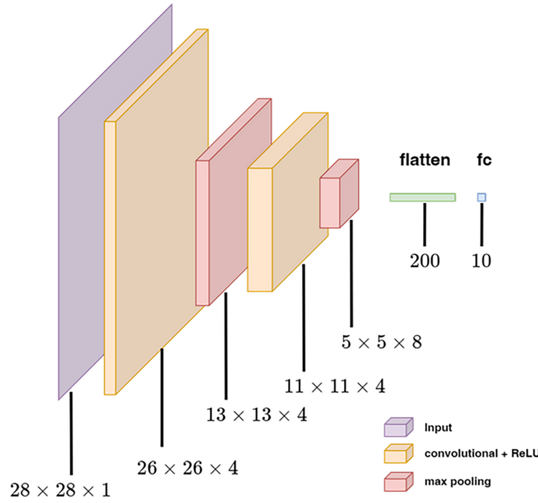

Fig. 3. The architecture of $\Phi(\bullet)$ for the image multi-class experimental setting. The figure also reports the size of the activations of each layer.

- The incremental solution provided by TyBox is able to achieve the same accuracy of the TF model trained in a non-incremental way. This is a crucial ability in the incremental learning setting.

6.2 Image Multi-Class Classification

The second experimental setting concerns the image classification on a multi-class problem, and for this purpose, the MNIST [4] and FMNIST [3] datasets have been considered.

In this experimental setting, we considered three different application scenarios:

- *Transfer learning*, in which the original model $\Phi(\bullet)$ is initially trained to solve a given task, and once deployed it is incrementally retrained on-device to address a different task. This application scenario measures the ability of the designed algorithm to transfer previously acquired knowledge to a different classification task.
- *Abrupt concept drift learning*, in which the distribution of the data changes over time after the model has been deployed on-device. This application scenario measures the ability of the designed algorithm to adapt to changes in the process generating the data.
- *Incremental class concept drift learning*, in which the task to be solved is extended after the model has been deployed on-device. This application scenario measures the ability of the model to learn a new task without forgetting the old one.

6.2.1 Neural Network Architecture. The model $\Phi(\bullet)$ used for this experiment is a CNN consisting of $\Phi_f(\bullet)$ and $\Phi_c(\bullet)$. More specifically, $\Phi_f(\bullet)$ consists of two convolutional blocks, each of which relies on one convolutional layer and a 2×2 max pooling layer. The first and the second convolutional layer consist of four and eight 3×3 convolutional filters, respectively. Differently, $\Phi_c(\bullet)$ consists of one dense layer with $|\theta| = 2{,}010$ that relies on softmax activation. The architecture of $\Phi(\bullet)$ is presented in Figure 3.

6.2.2 Neural Network Parameters. The model $\Phi(\bullet)$ is initially trained on the training set of the MNIST dataset (i.e., 60,000 pictures) by considering sparse categorical cross entropy as the

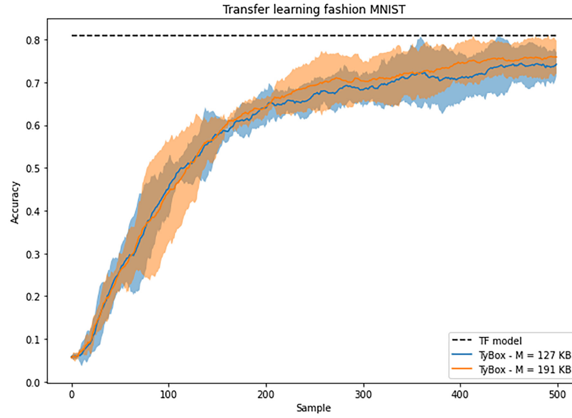

Fig. 4. The classification accuracy on transfer learning for the image multi-class classification setting.

loss function, whereas rmsprop was used as the optimizer. The learning rate was set to $1e^{-3}$. The number of training epochs was set to 25 for all experiments. With regard to on-device learning, the learning rate ϵ was set to 0.01.

6.2.3 Profiler and Buffer. For these experiments, we considered two different configurations for $M = \{127 \text{ KB}, 191 \text{ KB}\}$ modeling different memory occupations of the firmware on the tiny device.

The memory footprint M_Φ of the incremental solution $\Phi(\bullet)$ designed by TyBox for this experiment is 26 KB. Further details on M_Φ are provided later in Table 3.

In addition, TyBox computed M_B equal to 101 KB and 165 KB for the two values of M, respectively, resulting in N_B equal to 126 and 210, respectively.

6.2.4 Experiment Results and Comparison. In each experiment, the algorithm has been tested on the entire test set after each new supervised sample is provided.

In this case, as a comparison, we considered the results of the original network $\Phi(\bullet)$ retrained with a standard train-then-deploy approach on TF. We recall that this TF model does not provide any incremental learning on-device capabilities.

Experimental results are averaged over five runs with randomly initialized partitioning of the dataset. The classification accuracy is provided with a .95 confidence for the TyBox model, whereas confidence intervals on the TF model are not reported in figures for clarity.

6.2.5 Transfer Learning. In this application scenario, to test the incremental learning ability of the proposed solution, we initially trained $\Phi(\bullet)$ on MNIST and applied the incremental learning procedure on FMNIST. Even in this case, FMNIST has been divided into training (500 samples) and testing (500 samples) sets. The training set is provided in an incremental manner during the experiments, whereas the test set is used to evaluate the classification accuracy after each supervised sample is provided.

The TF model used for comparison is compiled with Adam as the optimizer, the loss function is categorical cross entropy, and ϵ is set to 0.01. To obtain results comparable to the ones of TyBox, the TF model was trained with 500 samples for 50 epochs, and the batch size was set to 32.

The classification accuracy for the two configurations of the TyBox incremental solutions (i.e., with $M = 127$ KB and $M = 191$ KB) and for the TF model are provided in Figure 4. These experimental results are particularly interesting since, even in this case, the incremental solutions designed by TyBox are able to increase their accuracy over time, providing results that, at the end of the experiments, approach the ones provided by the TF model that is trained directly on all training

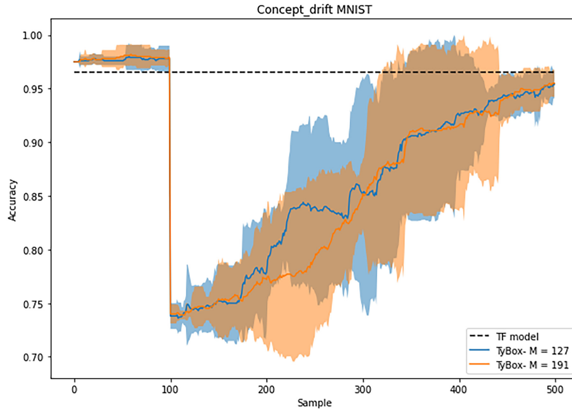

Fig. 5. The classification accuracy on the abrupt concept drift learning experiment for the image multi-class classification setting.

data. We emphasize that the TF model cannot be trained on the target device due to the lack of on-device retraining functions in TFLM.

6.2.6 Learning with Abrupt Concept Drift. Following the formalization of learning in the presence of concept drift introduced in the work of Ditzler et al. [18], we defined an application scenario where the process generating the data evolves over time. In particular, we considered an abrupt concept drift affecting the MNIST multi-class classification problem where classes 4 and 6 are swapped at sample 100.

Even in this case, we considered for comparison a TF model using Adam as the optimizer with categorical cross entropy as the loss function. ϵ was set to 0.01. To obtain results comparable to the solution designed by TyBox, the TF model was trained on the same amount of data (400 samples) for 50 epochs with a batch size equal to 32. The TF model was trained only on data collected after the abrupt concept drift.

The classification accuracy for the two different configurations of the TyBox incremental solutions and the TF model are provided in Figure 5. Two main comments arise. First, both TyBox solutions are able to adapt to a concept drift by increasing the accuracy over the acquired samples after the change occurs. Second, the TyBox solution characterized by the smaller buffer is faster in adapting to the concept drift. This is reasonable since a smaller dimension of the buffer allows quick removal of obsolete samples following a first-in-first-out policy. A further experiment in the field of abrupt concept drift performed on the Fashion-MNIST dataset can be found in Appendix A.

6.2.7 Incremental Class Concept Drift Learning. This application scenario models the setting where the classification task is incrementally extended to model a more complex classification problem [31]. More specifically, the classification problem on the MNIST dataset is initially configured only with digits from 0 to 7, whereas at sample 100, digits 8 and 9 are included into the classification problem. A total of 500 samples are used as the incremental training set, whereas the test set consists of 500 samples composed of all 10 classes.

Even in this case, the TF model used for comparison is compiled with Adam as the optimizer, and the loss function is categorical cross entropy. ϵ was set to 0.01. To obtain results comparable to the ones of TyBox, the TF model was trained with 500 samples for 10 epochs, and the batch size is 32. The TF model has been trained only on the data coming from the dataset composed of all 10 classes.

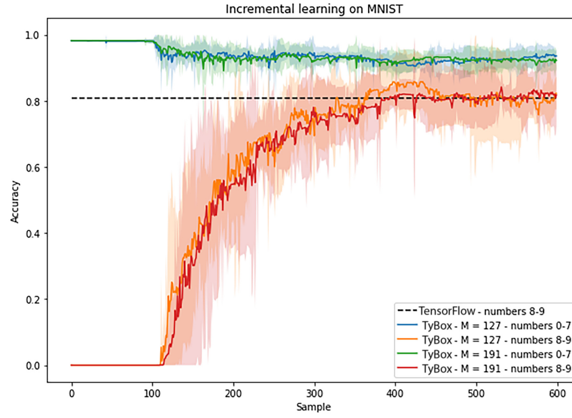

Fig. 6. The classification accuracy on the incremental class concept drift learning experiment for the image multi-class classification setting.

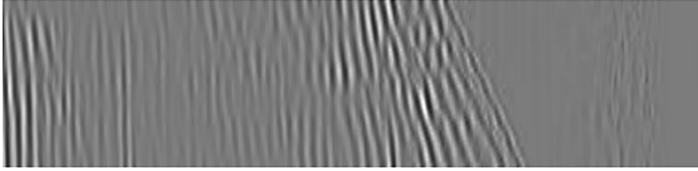

Fig. 7. An example of the radar data used as input for the network. The *x*-axis represents the distance from the radar, whereas the *y*-axis represents time.

The classification accuracy for the two configurations of the TyBox solutions and the TF model are depicted in Figure 6. In particular, two distinct curves are provided for the accuracy on digits 0 through 7 and that on digits 8 and 9. Interestingly, high classification accuracies can be achieved on the digits that the TinyML model has never seen before while maintaining acceptable performance even on the previously learned digits.

6.3 UWB Human Activity Recognition

6.3.1 Problem Definition. The third experimental setting concerns UWB human activity recognition. For this purpose, we considered the IR-UWB radar measurements of the human activity dataset [47] consisting of radar records of three people. The records were collected inside of a building and include both through-air (nothing between the radar device and the target) and through-wall (the radar device is positioned behind a wall). This problem is formalized as a multi-class classification problem, where the goal is to correctly classify if a person is absent, walking, or standing. We modeled the incremental learning problem as follows: we initially defined the dataset composed only of the records of the first two persons and then, during the experiments, included samples of the third person after sample 100. This can be framed as an incremental instances concept drift adaptation task (new training patterns of the same classes become available during the incremental training), as the data distribution of the used data changes over time.

The data are organized as a Distance × Time matrix, where at each pulse of the radar a vector of 32 distance bins is collected and this measurement is repeated 768 times in a 4-second time span, resulting in a matrix I of dimension 768×32. No particular pre-processing technologies (but a standard normalization) have been applied to the input data. An example of the data used as input for the network can be found in Figure 7.

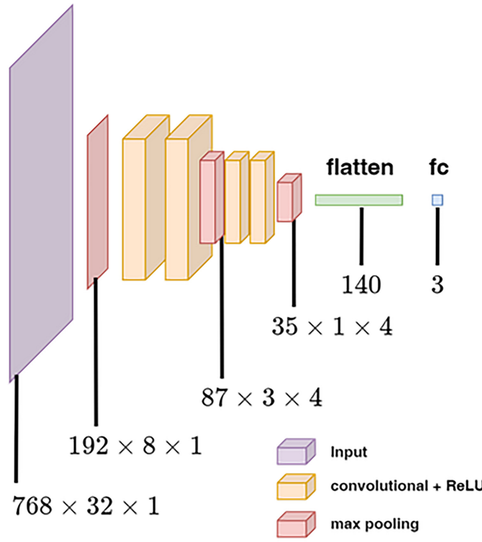

Fig. 8. The neural network architecture Φ considered in the UWB human activity recognition experimental setting. The dimension of activations for each layer are also reported.

6.3.2 Neural Network Architecture. The model $\Phi(\bullet)$ used for this experiment consists of both $\Phi_f(\bullet)$ and $\Phi_c(\bullet)$. In more detail, $\Phi_f(\bullet)$ consists of an initial 4×4 max pooling layer, meant to reduce the dimensions of the input data, which is followed by two convolutional blocks, each of which consists of two convolutional layers and a 2×2 max pooling layer. Each convolutional layer consists of four rectangular convolutional filters of dimension 10. Finally, $\Phi_c(\bullet)$ consists of a dense layer with softmax activation. The architecture of the network is presented in Figure 8.

6.3.3 Neural Network Parameters. The model $\Phi(\bullet)$ is initially trained on the dataset composed of records of the first two persons. For this first training phase, we considered categorical cross entropy as the loss function, whereas rmsprop was used as the optimizer. The ϵ was set to $1e^{-3}$, whereas the number of training epochs was set to 20.

For on-device incremental learning, we considered a dataset consisting of data from all three persons and relied on the backpropagation algorithm with ϵ set to 0.001.

6.3.4 Profiler and Buffer. Two different values of M (i.e., 136 KB and 200 KB) have been considered to model two versions of the firmware running on the target device. The total memory footprint M_Φ required by $\Phi(\bullet)$ in this experiment is 112 KB. Further details on the memory demands are provided later in Table 3. The two corresponding values of M_B computed by TyBox are 24 KB and 88 KB, respectively, whereas the corresponding values of N_B are 43 samples and 160 samples, respectively.

6.3.5 Experimental Result. To evaluate the effectiveness of the automatically designed incremental solutions of TyBox, we explicitly computed both the classification accuracy for the first two subjects (the ones present in the initial dataset) and the classification accuracy for the third person (the one introduced in the dataset that is incrementally learned during the operational life).

Similarly to the other experimental settings, as a comparison, we considered the retraining of the original model $\Phi(\bullet)$ in TF on the dataset [47] composed of data from all three persons. The TF model used for comparison was compiled with Adam as the optimizer, the loss function was the

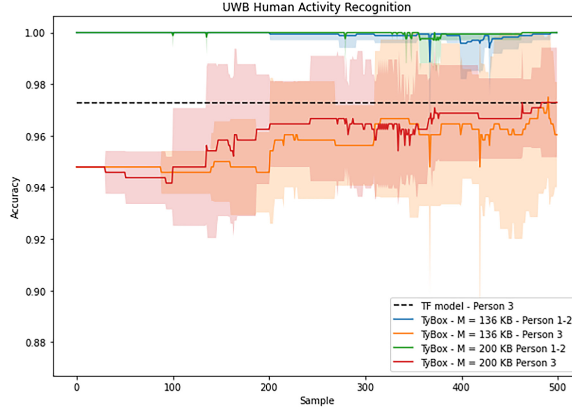

Fig. 9. The classification accuracy for the UWB human activity recognition experimental setting.

categorical cross entropy, and ϵ was set to 0.01. To obtain results comparable to the ones of TyBox, the same amount of data (i.e., 400 samples) was used for training of the TF model for 50 epochs, and the batch size was set to 32.

Experimental results, averaged over five runs, are shown in Figure 9. The confidence intervals for these results are provided with a .95 confidence. Confidence intervals on the TF model are not reported in the figure for clarity. Two main comments arise. First, as expected, the incremental learning solution is able to effectively improve the recognition on the third person during the experiment. Second, such an ability does not come at the expense of a reduction in the classification ability of the first two persons.

6.4 Porting of the Designed Solutions on the Arduino Nano 33 BLE

All incremental learning solutions automatically designed by TyBox for the previously described experimental settings have been ported on the Arduino Nano 33 BLE [1]. To evaluate the efficiency of these solutions, we computed the execution time (measured directly on the device by means of the Arduino software tool) for the forward pass and the backpropagation of a single datum and for the whole training procedure when the buffer is full for all experiments (Table 2). As a comparison, for the forward pass, we also report the execution times of $\Phi(\bullet)$ deployed with TFLM. We emphasize that TFLM does not implement any procedure of on-device training, hence it is not possible to provide a comparison for the backpropagation times. Finally, we also provide the execution time obtained with the Train++ algorithm, which has been reimplemented to run on the Arduino Nano 33 BLE.

Furthermore, we computed (and reported in Table 3) the memory demands for all incremental learning solutions automatically designed by TyBox.

Three main comments arise. First, during the inference, TyBox does not add any significant overhead with respect to the non-incremental version of the model deployed with TFLM, and guarantees comparable performance even with a much simpler solution like Train ++ on the single datum. Second, depending on the experimental setting, the on-device training phase on the whole buffer requires execution times comparable to the execution of the feature extraction (multi-class image) or even negligible with respect to it (UWB recognition). Of course, when $\Phi_f(\bullet)$ is not performed, the overhead becomes relevant. Finally, as expected by the automatic incremental design module of TyBox introduced in Section 4, from Table 3 it is possible to see that the sum of M_Φ and M_B is less than or equal to M, as designed in Section 4.3.

Table 2. Execution Times of Inference and Training on a Single Datum and on the Full Buffer for TyBox, for the Non-Incremental TF model Deployed with TFLM, and for Train++

				TyBox			TF Model	TRAIN++
Experiment	M	$\|\psi\|$	N_B	**Inference** $\Phi_f + \Phi_c$ **1 Datum**	**Backpropagation** Φ_c **1 Datum**	**Full Buffer Training Time**	**Inference** Φ **1 Datum**	**Inference+Train 1 Datum**
Binary image	142 KB	4	200	0 + 100 μs	25 μs	26 ms	91 μs	39 μs
Multi-class image	191 KB	200	210	63 ms + 170 μs	182 μs	76 ms	64 ms	–
	127 KB	200	126	63 ms + 170 μs	182 μs	45 ms	64 ms	–
UWB recognition	200 KB	140	160	466 ms + 152 μs	145 μs	50 ms	467 ms	–
	136 KB	140	43	466 ms + 152 μs	145 μs	14 ms	467 ms	–

Table 3. Memory Profile for the Automatically Designed and Generated TinyML Models by TyBox in All Experimental Settings

Experiment	M	$\|\psi\|$	N_B	M_B	$M^p_{\Phi_f}$	$M^p_{\Phi_c}$	$M^a_{\Phi_f}$	$M^a_{\Phi_c}$	M_Φ
Binary image	142 KB	4	9,088	≈142 KB	–	40 B	–	24 B	64 B
Multi-class image	191 KB	200	210	165 KB	1,344 B	840 B	15,500 B	8,040 B	≈26 KB
	127 KB	200	126	101 KB	1,344 B	840 B	15,500 B	8,040 B	≈26 KB
UWB recognition	200 KB	140	160	88 KB	4,224 B	1,692 B	104,500 B	572 B	≈112 KB
	136 KB	140	43	24 KB	4,224 B	1,692 B	104,500 B	572 B	≈112 KB

7 ON-DEVICE TINYML: CHALLENGES AND OPPORTUNITIES

On-device learning, in the context of TinyML, is currently an open research area. A lot of work is needed in the definition of widely used benchmarks and in the formalization of open challenges. Indeed, currently, all papers in this area focus on demonstrating the feasibility of the execution of learning algorithms on a constrained device, by training those algorithms on the same tasks that are used in batch learning. Although extremely interesting from the technical point of view, from the applicative point of view, the benchmarks and challenges defined for batch learning are hardly suitable for a tiny, distributed learning environment.

In this framework, the two main differences with the batch learning environment concern the amount of data available for the training and the availability of labels for the collected data. Interestingly, on one hand, it is easy for tiny devices to collect data. On the other hand, it is quite challenging to locally create a large training set (due to the memory constraints) and to have supervised information of collected data (due to the need of a supervisor). For this reasons, we believe that few-shot, semi-supervised learning on-device will represent a game-changer in the field of learning on tiny devices.

Finally, on-device learning on tiny devices opens the possibility for the final user to easily train a machine learning algorithm on the data they collect. The ability to obtain highly customized algorithms comes with the dangers that appear every time a machine learning algorithm is learned on uncontrolled data sources. Indeed, without proper knowledge of how to collect the data, the algorithms could lead to unreliable performance, or even worse, they could be derailed on purpose by a malevolous actor. For this reason, in the future, it will be important for the community working in this field to develop methodologies to drive a "controlled" evolution of the original non-incremental model. This will allow the models to improve, with data coming directly from the environment in which the devices operate, by also limiting the ways in which the model can evolve to the ones defined by the designer of the application.

8 CONCLUSION

The aim of this article was to introduce, to the best of our knowledge for the first time in the literature, a toolbox for the automatic design and code generation of incremental on-device TinyML models. The effectiveness and efficiency of the proposed solution have been successfully evaluated on a state-of-the-art solution and on three publicly available benchmarks by also porting all designed solutions on an off-the-shelf embedded device.

Future works will include the use of post-training quantization or quantization-aware mechanisms to reduce the memory demands of weights and activations, the extension of TyBox to different families of deep learning models, and the introduction of concept drift detection mechanisms to support an active adaptation of the TinyML models in time-varying scenarios.

APPENDIX

A ADDITIONAL ABRUPT CONCEPT DRIFT EXPERIMENT

A.0.1 Neural Network Architecture. The model $\Phi(\bullet)$ used for this experiment is a CNN consisting of $\Phi_f(\bullet)$ and $\Phi_c(\bullet)$, the same as presented in Section 6.2.1. The architecture of $\Phi(\bullet)$ is presented in Figure 3.

A.0.2 Neural Network Parameters. The model $\Phi(\bullet)$ is initially trained on the training set of the FMNIST dataset (i.e., 60,000 pictures), by considering sparse categorical cross entropy as the loss function, whereas rmsprop was used as the optimizer. The learning rate was set to $1e^{-3}$. The number of training epochs was set to 10. With regard to on-device learning, the learning rate ϵ was set to 0.005.

A.0.3 Profiler and Buffer. For these experiments, we considered the same configurations for $M = \{127 \text{ KB}, 191 \text{ KB}\}$ that were used in the experiments of Section 6.2.

For this reason, the same considerations on M_Φ (equal to 26 KB), M_B (equal to 101 KB and 165 KB in the two configurations, respectively), and N_B (equal to 126 and 210, respectively) done in that section are also valid for this experiment.

A.0.4 Experiment Results and Comparison. As done previously, the algorithm is tested on the entire test set after each new supervised sample is provided.

In this case, as a comparison, we considered the results of the original network $\Phi(\bullet)$ retrained with a standard train-then-deploy approach on TF with no incremental learning on-device capabilities.

Experimental results are averaged over five runs with randomly initialized partitioning of the dataset. The classification accuracy is provided with a .95 confidence for the TyBox model, whereas confidence intervals on the TF model are not reported in figures for clarity.

A.0.5 Learning with Abrupt Concept Drift on FMNIST. Similarly to what was proposed in the experimental scenario proposed in Section 6.2.6, the process generating the data of this experiment evolves over time. In particular, we considered an abrupt concept drift affecting the Fashion-MNIST multi-class classification problem where classes “coat” and “shirt” are swapped at sample 100.

As a comparison, we considered a TF model using Adam as the optimizer with categorical cross entropy as the loss function. ϵ was set to 0.005. To obtain results comparable to the solution designed by TyBox, the TF model was trained on the same amount of data (500 samples) for 50 epochs with a batch size equal to 32. The TF model was trained only on data collected after the abrupt concept drift.

The classification accuracy for the two different configurations of the TyBox incremental solutions and the TF model are provided in Figure A.1.

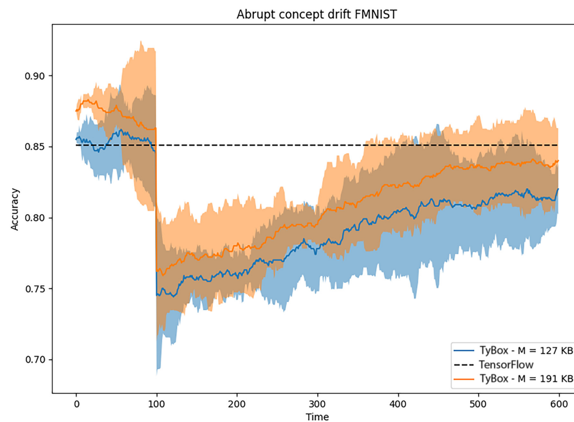

Fig. A.1. The classification accuracy on the abrupt concept drift learning experiment with the FMINST dataset for the image multi-class classification setting.

ACKNOWLEDGMENTS

The authors would like to thank Ing. P. Lento, and Dr. A. Bassi from Trusense s.r.l. and Ing. G. Viscardi from Politecnco di Milano for their support in this work.

REFERENCES

[1] Arduino. n.d. Arduino Nano 33 BLE Sense. Retrieved June 21, 2023 from https://docs.arduino.cc/hardware/nano-33-ble-sense.

[2] UC Irvine. n.d. Banknote Authentication Dataset. https://archive.ics.uci.edu/ml/datasets/banknote+authentication.

[3] GitHub. n.d. Fashion MNIST Dataset. Retrieved June 21, 2023 from https://github.com/zalandoresearch/fashion-mnist.

[4] GitHub. n.d. MNIST Dataset. Retrieved June 21, 2023 from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz.

[5] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, et al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. Retrieved June 21, 2023 from https://www.tensorflow.org.

[6] Cesare Alippi, Giacomo Boracchi, and Manuel Roveri. 2011. An effective just-in-time adaptive classifier for gradual concept drifts. In *Proceedings of the 2011 International Joint Conference on Neural Networks*. IEEE, Los Alamitos, CA, 1675–1682.

[7] Cesare Alippi, Giacomo Boracchi, and Manuel Roveri. 2013. Just-in-time classifiers for recurrent concepts. *IEEE Transactions on Neural Networks and Learning Systems* 24, 4 (2013), 620–634.

[8] Cesare Alippi, Simone Disabato, and Manuel Roveri. 2018. Moving convolutional neural networks to embedded systems: The AlexNet and VGG-16 case. In *Proceedings of the 2018 17th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN'18)*. IEEE, Los Alamitos, CA, 212–223.

[9] Cesare Alippi and Manuel Roveri. 2017. The (not) far-away path to smart cyber-physical systems: An information-centric framework. *Computer* 50, 4 (2017), 38–47. https://doi.org/10.1109/MC.2017.111

[10] Simone Benatti, Fabio Montagna, Victor Kartsch, Abbas Rahimi, Davide Rossi, and Luca Benini. 2019. Online learning and classification of EMG-based gestures on a parallel ultra-low power platform using hyperdimensional computing. *IEEE Transactions on Biomedical Circuits and Systems* 13, 3 (June 2019), 516–528. https://doi.org/10.1109/TBCAS.2019.2914476

[11] Han Cai, Chuang Gan, Ligeng Zhu, and Song Han. n.d. Tiny transfer learning: Towards memory-efficient on-device learning. *arXiv:2007.11622 [cs]* (n.d.).

[12] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, et al. 2020. TensorFlow lite micro: Embedded machine learning on TinyML systems. *arXiv:2010.08678* (2020).

[13] Matthias De Lange, Rahaf Aljundi, Marc Masana, Sarah Parisot, Xu Jia, Aleš Leonardis, Gregory Slabaugh, and Tinne Tuytelaars. 2021. A continual learning survey: Defying forgetting in classification tasks. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44, 7 (2021), 3366–3385.

[14] Simone Disabato and Manuel Roveri. 2018. Reducing the computation load of convolutional neural networks through gate classification. In *Proceedings of the 2018 International Joint Conference on Neural Networks (IJCNN'18)*. IEEE, Los Alamitos, CA, 1–8.

[15] Simone Disabato and Manuel Roveri. 2020. Incremental on-device tiny machine learning. In *Proceedings of the 2nd International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things (AIChallengIoT'20)*. 7–13.

[16] Simone Disabato and Manuel Roveri. 2021. Tiny machine learning for concept drift. *arXiv:2107.14759 [cs]* (2021). http://arxiv.org/abs/2107.14759.

[17] Gregory Ditzler, Manuel Roveri, Cesare Alippi, and Robi Polikar. 2015. Learning in nonstationary environments: A survey. *IEEE Computational Intelligence Magazine* 10, 4 (2015), 12–25.

[18] Gregory Ditzler, Manuel Roveri, Cesare Alippi, and Robi Polikar. 2015. Learning in nonstationary environments: A survey. *IEEE Computational Intelligence Magazine* 10 (Nov. 2015), 12–25. https://doi.org/10.1109/MCI.2015.2471196

[19] Joao Gama, Pedro Medas, Gladys Castillo, and Pedro Rodrigues. 2004. Learning with drift detection. In *Advances in Artificial Intelligence—SBIA 2004*. Lecture Notes in Computer Science, Vol. 3171. Springer, 286–295.

[20] Alexander Gepperth and Barbara Hammer. 2016. Incremental learning algorithms and applications. In *Proceedings of the European Symposium on Artificial Neural Networks (ESANN'16)*.

[21] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. 2021. A survey of quantization methods for efficient neural network inference. *arXiv:2103.13630 [cs]* (2021).

[22] Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics*. 249–256.

[23] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press, Cambridge, MA.

[24] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient convolutional neural networks for mobile vision applications. *arXiv:1704.04861* (2017).

[25] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size. *arXiv:1602.07360* (2016).

[26] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, et al. 2017. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences* 114, 13 (2017), 3521–3526.

[27] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. 2022. On-d training under 256KB memory. *arXiv:2206.15472 [cs]* (2022). http://arxiv.org/abs/2206.15472.

[28] Jiayi Liu, Samarth Tripathy, Unmesh Kurup, and Mohak Shah. 2020. Pruning algorithms to accelerate convolutional neural networks for edge applications: A survey. *arXiv:2005.04275 [cs, stat]* (2020).

[29] Jie Lu, Anjin Liu, Fan Dong, Feng Gu, Joao Gama, and Guangquan Zhang. 2018. Learning under concept drift: A review. *IEEE Transactions on Knowledge and Data Engineering* 31, 12 (2018), 2346–2363.

[30] Arun Mallya and Svetlana Lazebnik. 2018. PackNet: Adding multiple tasks to a single network by iterative pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 7765–7773.

[31] Marc Masana, Xialei Liu, Bartlomiej Twardowski, Mikel Menta, Andrew D. Bagdanov, and Joost van de Weijer. 2020. Class-incremental learning: Survey and performance evaluation on image classification. *arXiv preprint arXiv:2010.15277* (2020).

[32] Michael McCloskey and Neal J. Cohen. 1989. Catastrophic interference in connectionist networks: The sequential learning problem. *Psychology of Learning and Motivation* 24 (1989), 109–165.

[33] German I. Parisi, Ronald Kemker, Jose L. Part, Christopher Kanan, and Stefan Wermter. 2019. Continual lifelong learning with neural networks: A review. *Neural Networks* 113 (2019), 54–71.

[34] Lorenzo Pellegrini, Gabrielle Graffieti, Vincenzo Lomonaco, and Davide Maltoni. 2020. Latent replay for real-time continual learning. *arXiv:1912.01100 [cs, stat]* (2020).

[35] Visal Rajapakse, Ishan Karunanayake, and Nadeem Ahmed. 2023. Intelligence at the extreme edge: A survey on reformable TinyML. *ACM Computing Surveys*. Just Accepted.

[36] Vikram Ramanathan. 2020. Online On-Device MCU Transfer Learning. Retrieved June 21, 2023 from https://vikramramanathan.com/files/CS249R_Final_Report_Transfer_Learning.pdf.

[37] Leonardo Ravaglia, Manuele Rusci, Davide Nadalini, Alessandro Capotondi, Francesco Conti, Luca Benini, and Luca Benini. 2021. A TinyML platform for on-device continual learning with quantized latent replays. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 11 (2021), 789–802. https://doi.org/10.1109/JETCAS.2021.3121554

[38] Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl, and Christoph H. Lampert. 2017. iCaRL: Incremental classifier and representation learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2001–2010.

[39] Haoyu Ren, Darko Anicic, and Thomas Runkler. 2021. TinyOL: TinyML with online-learning on microcontrollers. *arXiv:2103.08295 [cs, eess]* (2021). http://arxiv.org/abs/2103.08295.

[40] Manuel Roveri. 2023. Is tiny deep learning the new deep learning? In *Computational Intelligence and Data Analytics*. Springer, 23–39.

[41] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. 1986. Learning representations by back-propagating errors. *Nature* 323, 6088 (1986), 533–536.

[42] Simone Scardapane, Danilo Comminiello, Michele Scarpiniti, Enzo Baccarelli, and Aurello Uncini. 2020. Differentiable branching in deep networks for fast inference. In *Proceedings of the 2020 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP'20)*. 4167–4171.

[43] Bharath Sudharsan, Piyush Yadav, John G. Breslin, and Muhammad Initzar Ali. 2021. Train++: An incremental ML model training algorithm to create self-learning IoT devices. In *Proceedings of the 2021 IEEE SmartWorld Conference*. 97–106.

[44] Mingxing Tan and Quoc V. Le. 2019. EfficientNet: Rethinking model scaling for convolutional neural networks. *arXiv:1905.11946* (2019). https://arxiv.org/abs/1905.11946.

[45] Pete Warden and Daniel Situnayake. 2020. *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. O'Reilly Media.

[46] Gerhard Widmer and Miroslav Kubat. 1996. Learning in the presence of concept drift and hidden contexts. *Machine Learning* 23 (1996), 69–101.

[47] Zhengliang Zhu and Al. 2020. A dataset of human motion status using IR-UWB through-wall radar. *arXiv:2008.13598* (2020). https://doi.org/10.48550/ARXIV.2008.13598

Received 30 September 2022; revised 11 April 2023; accepted 4 June 2023