arXiv:2107.14759v1 [cs.LG] 30 Jul 2021

Tiny Machine Learning for Concept Drift

Simone Disabato, and Manuel Roveri, *Senior Member, IEEE*

Abstract—Tiny Machine Learning (TML) is a new research area whose goal is to design machine and deep learning techniques able to operate in Embedded Systems and IoT units, hence satisfying the severe technological constraints on memory, computation, and energy characterizing these pervasive devices. Interestingly, the related literature mainly focused on reducing the computational and memory demand of the inference phase of machine and deep learning models. At the same time, the training is typically assumed to be carried out in Cloud or edge computing systems (due to the larger memory and computational requirements). This assumption results in TML solutions that might become obsolete when the process generating the data is affected by concept drift (e.g., due to periodicity or seasonality effect, faults or malfunctioning affecting sensors or actuators, or changes in the users' behavior), a common situation in real-world application scenarios. For the first time in the literature, this paper introduces a Tiny Machine Learning for Concept Drift (TML-CD) solution based on deep learning feature extractors and a k-nearest neighbors classifier integrating a hybrid adaptation module able to deal with concept drift affecting the data-generating process. This adaptation module continuously updates (in a passive way) the knowledge base of TML-CD and, at the same time, employs a Change Detection Test to inspect for changes (in an active way) to quickly adapt to concept drift by removing the obsolete knowledge. Experimental results on both image and audio benchmarks show the effectiveness of the proposed solution, whilst the porting of TML-CD on three off-the-shelf micro-controller units shows the feasibility of what is proposed in real-world pervasive systems.

Index Terms—Tiny Machine Learning, Concept Drift, Adaptation, Deep Learning, k-Nearest Neighbour.

1 Introduction

Internet-of-Things (IoT) and embedded systems are nowadays part of our everyday life in a wide range of application scenarios (e.g., automotive, medical devices, and smart cities, to name a few). In recent years, the scientific and technological trend about these pervasive devices is to move the processing (and in particular the intelligent processing) as close as possible to where data are generated. The reason is twofold. First, IoT units and embedded systems already operate pervasively in the environment processing large amounts of data acquired by the sensors. Second, machine and deep learning solutions processing these data directly on the pervasive devices are crucial to support real-time applications, prolong the system lifetime, and increase the Quality-of-Service. Nevertheless, machine and deep learning solutions are typically characterized by memory and computational demands that rarely match the constraints on memory, computation, and energy characterizing the IoT units and embedded systems [1], [2], [3].

Tiny Machine Learning (TML) [4] is a relatively new research area aiming at filling this gap by designing "tiny" machine and deep learning solutions able to run on IoT units and embedded systems. Section 2 analyses the related literature, highlighting that most TML solutions focus on approximation, pruning, and quantization mechanisms to reduce memory and computational demand of machine and deep learning models. Although these solutions run on embedded systems and IoT units, their training is typically carried out on high-performing units (such as Cloud or EdgeComputing systems), with very few papers proposing on-device incremental learning mechanisms [5], [6].

- *S. Disabato and M. Roveri are with the Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB), Politecnico di Milano, Milan, Italy. E-mail: {simone.disabato,manuel.roveri}@polimi.it*

The ability to learn TML models directly on the devices is crucial to improve the accuracy over time by exploiting fresh information coming from the field, and to deal with concept drift, i.e., variations in the statistical behavior of the data generating process, a quite common situation in real-world applications (e.g., due to seasonality or periodicity effects, faults affecting sensors or actuators, changes in the user's behavior, or aging consequences). Failing to adapt TML models to concept drift results in a (possibly dramatic) decrease of the accuracy over time [7].

This paper aims at addressing this challenge by introducing, for the first time in the literature, a Tiny Machine Learning algorithm for Concept Drift (TML-CD) that can learn directly on the IoT unit or embedded system and adapt the knowledge base in response to a concept drift (thus tracking the evolution of the data generating process). To achieve this goal, we introduce three different adaptation mechanisms (i.e., passive, active, and hybrid), each of which has its advantages, issues, accuracy, and behavior. Among these mechanisms, we focus on the hybrid solution thanks to its ability to trade-off adaptation with memory demand. The three proposed TML-CD adaptive mechanisms have been tested in two different application scenarios (i.e., image classification and speech command recognition) and ported to three real-world Micro-Controller Units, showing their feasibility and effectiveness. Finally, the code is made available to the scientific community.[1]

The paper is organized as follows. Section 2 revises the related literature. Section 3 formalises the addressed problem, whereas Sections 4, 5, and 6 present the proposed TML-CD solution and its stages. Finally, Section 7 details the experimental results and Section 8 draws the conclusions.

1. The repository link is https://github.com/simdis/Adaptive-TML.

2 Related Literature

This section discusses the related literature about machine and deep learning solutions in presence of concept drift as well as TML solutions.

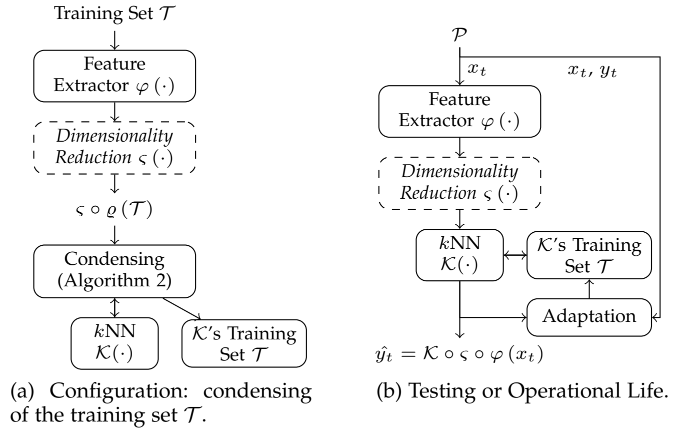
Machine and Deep Learning Solutions in Presence of Concept Drift: The literature about machine and deep learning in presence of concept drift refers to adaptive solutions able to deal with concept drift affecting the data generating process. The related literature usually groups them into two main families: passive and active [7], [8], [9].

Passive solutions adapt the model at each incoming data, disregarding the fact that a concept drift has occurred in the data-generating process (or not). The gradual forgetting classifiers, e .g. [10], [11], w hich r educe t he i mportance of older samples over time, are examples of passive solutions. The Concept Drift Very Fast Decision Tree (CDVFDT) [12] introduces a Decision Tree that learns new subtrees on incoming data. However, most passive solutions employ ensemble methods and their adaptation mechanisms consist in adding, removing, or weighting the ensemble base classifiers, e .g., S treaming E nsemble A lgorithm [13], Dynamic Classifier S election [14], o r t he a daptive e nsemble o f Decision Trees proposed in [15]. Deep learning-based passive solutions examples can be found in [16], [17], [18].

On the contrary, *active solutions* aim at detecting concept drift in the data generation process and, only in that case, they adapt their model to the new conditions. Change Detection Tests (CDT) are statistical techniques meant to sequentially process the incoming data inspecting for concept drift. [19] proposed to use the Hellinger distance between the reference probability distribution and the one estimated on incoming data along with a t-test to detect changes. [20] relies on bootstrapping several windows of data and the Kullback Leibler divergence as a measure of the distance among them. A few works detect changes with density estimation techniques [21], [22]. Other examples of CDT used in active solutions can be found in [23], [24], [25], [26]. In active solutions, the adaptation stage following a concept drift detection is usually carried out in two steps [27]: first, the time instant the concept drift occurred is estimated by ad-hoc mechanisms (e.g., by Change-Point Methods); second, the obsolete knowledge, i.e., that acquired before the concept drift occurred, is discarded. To achieve this goal, the adaptation mechanisms typically rely on a window over the last acquired data, whose size is usually optimized over time to reduce its memory requirements [28], [29], or on all the samples seen so far (suitably weighted) [30]. Finally, deep learning-based active approaches (integrating deep learning solutions with active adaptive solutions) can be found in [31], [32].

Tiny Machine Learning: TML techniques aim at designing machine and deep learning models that take into account the severe technological constraints on memory, computation, and energy characterizing IoT units and embedded systems [2], [4], [6].

To achieve this goal, most solutions employ approximation techniques from the deep learning literature. These approximation mechanisms can be grouped into three main families according to the way the approximation is carried out: pruning of processing layers (and part of them) [33], [34], quantization of parameters and activations with limited precision or binary parameters [35], [36], [37], or solutions integrating both pruning and quantization [1].



(a) Configuration: condensing of the training set $\mathcal{T}$. (b) Testing or Operational Life.

Fig. 1: The proposed architecture of the proposed solution for Tiny Machine Learning for Concept Drift (TML-CD) $\mathcal{D}$ on embedded systems and IoT units.

As regards the target of the approximation mechanisms, most of TML literature focuses on approximated Convolutional Neural Networks [38], [39], [40], [41], with a few works considering recurrent DL architectures [42], [43]. In particular, [44] introduces a methodology to explore sparse (and pruned) CNN architectures able to be executed on microcontroller units, whereas [5] proposes a Tiny-CNN whose biases can be learned directly on the device. Finally, [45] investigates the impact of quantized networks in TinyML embedded systems.

Although there are very few works proposing on-device learning, e.g., [5], [6], to the best of our knowledge, no work presents a Tiny-ML solution able to adapt over time to concept drift.

3 Problem Formulation

Let $\mathcal{P}$ be a data generating process that, at each time instant t, provides a pair (x_t, y_t) sampled from an unknown probability distribution $p_t(x, y)$, where x is the input of the proposed solution $\mathcal{D}$ (e.g., an image or an audio clip) and $y \in \Lambda$ its classification label.[2]

Moreover, following a *test-then-train* approach [7], the proposed solution $\mathcal{D}$ receives the supervised information (the true label y_t) only after it provides the classification output $\hat{y}_t = \mathcal{D}(x_t)$ on input x_t, at each time instant t.[3]

In a concept drift scenario, the process $\mathcal{P}$ might evolve over time, hence inducing a shift in the distribution $p_t(x, y)$ at an unknown time instant t^*. It is worth noting that the change in $p_t(x, y)$ might affect the input x (e.g., by the introduction of noise), the set Λ (e.g., class change), or both.

The goal of the proposed TML-CD solution is to react and adapt $\mathcal{D}$ to changes in $p_t(x, y)$ so as to guarantee the highest accuracy over time.

2. Let λ be the cardinality of Λ, i.e., the number of classes in the considered classification problem.

3. Without any loss of generality, the supervised information might not be provided at every time instant. In those cases, the proposed solution only provides its classification output.

Algorithm 1: A sketch of the proposed solution.

Input: Training Set $\mathcal{T}$, Feature Extractor $\varsigma \circ \varrho$.
Parameters : Number of neighbors k.

```
1  Preprocess T.                                   ▷ Initialization.
2  Initialize the k-NN classifier K with ς ∘ ϱ(T).
3  Define D = K ∘ ς ∘ ϱ.
                              ▷ Loop over samples arriving at time t.
4  foreach (x_t, y_t) ~ P, t = 1, 2, ... do
5  |  Predict ŷ_t ← D(x_t).
6  |  Adaptation of D.
```

Algorithm 2: The Condensed Nearest Neighbor [49].

Input: Training Set $\mathcal{T}$.
Output: Condensed Representation $\bar{\mathcal{T}} \subseteq \mathcal{T}$.

```
                    ▷ H contains one sample, D all the others.
1  Initialize H ← {t ∈ T} and D ← T \ H.
2  Initialize the kNN K with H.
3  do
4  |  foreach t ← (x, y) ∈ D do
5  |  |  Predict ŷ ← K(t).
6  |  |  if ŷ ≠ y then                     ▷ Condensing Update:
7  |  |  |  D ← D \ {t}.                   ▷ Move t from D to H.
8  |  |  |  H ← H ∪ {t}.
9  |  |  |  (Re–)train the kNN K with H.
10 while H and D are modified in the foreach loop.
11 return T̄ ← H
```

4 The Proposed Tiny Machine Learning for Concept Drift

Figure 1 shows the general architecture of the proposed solution for Tiny Machine Learning for Concept Drift (TML-CD) $\mathcal{D}$, which comprises the following five different modules:

- **Feature Extractor ϱ**. The *Feature Extractor* extracts features from the input x_t. As in [1], [6], the Feature Extractor is a pre-trained DL model approximated by means of Task-Dropping (e.g., pruning of layers), Precision Scaling (e.g., weights precision reduction), or both, to satisfy the constraints on computation, memory, and energy characterizing the embedded systems and IoT units running $\mathcal{D}$.
- **Dimensionality Reduction Operator ς** The *Dimensionality Reduction Operator* ς (that can be optionally activated) reduces the dimensionality of features extracted by ϱ. In this paper, among the approaches presented in [6], we focused on the *Filter-Selection without supervised information*. This technique selects the f out of F filters of the last ϱ convolutional layer (and its subsequent batch-normalization channels, if any) providing the highest mean activation on publicly available benchmarks or datasets.
  It is crucial to point out that the adaptation step of $\mathcal{D}$ does not affect the feature extractor ϱ nor the dimensionality reduction operator ς, which are therefore fixed over time. Moreover, since the choice of the f filters to keep does not rely on the specific data-generation process $\mathcal{P}$, the block $\varsigma \circ \varrho$ can be defined at design time and prior to the porting of $\mathcal{D}$ on the IoT units. This is the reason why $\varsigma \circ \varrho$ is an input to our algorithm, and ς takes part in the choice of the approximated DL-feature extractor.
- ***k*NN Classifier** and **Training Set $\mathcal{T}$**. The kNN [46] *classifier* $\mathcal{K}(\cdot)$, whose input is either the output of the dimensionality-reduction operator $\varsigma \circ \varrho$ or that of the feature extractor ϱ (when no dimensionality reduction is considered), provides the classification $\hat{y}_t$ of the input x_t, while $\mathcal{T}$ is its training set. From the algorithmic point of view, the kNN is a statistical classifier based on majority voting, i.e., the predicted class corresponds to the majority class of the k nearest neighbors of the input sample within $\mathcal{K}$'s training set $\mathcal{T}$. Interestingly, it does not require a training phase, but only the initialization of its training set $\mathcal{T}$. Unless otherwise specified, the parameter k, i.e., the number of neighbors, is set to the ceiling of the square root of the available samples, as suggested in [27].
- **Adaptation Module**. The adaptation module receives as input the sample x_t and its kNN $\mathcal{K}$ prediction $\hat{y}_t$ and, when the supervised information y_t is available, it updates the TML-CD solution $\mathcal{D}$ so as to make it adaptive over time to concept drift. Among the four presented modules, the adaptation involves only the $\mathcal{K}$'s training set $\mathcal{T}$ [47], [48]. The kNN classifier adaption indeed requires to simply add the new supervised information (x_t, y_t) to its training set $\mathcal{T}$.

Algorithm 1, instead, details how the proposed TML-CD $\mathcal{D}$ works. More in detail, the TML-CD $\mathcal{D}$, which receives in input a feature extractor along with a dimensionality reduction operator ($\varsigma \circ \varrho$) and an initial training set $\mathcal{T}$, comprises two different stages: configuration and testing.

The *configuration* stage, detailed in lines 1–3 and shown in Figure 1a, encompasses an initial *preprocessing step* where the training set $\mathcal{T}$ is preprocessed to reduce the memory occupation (line 1) by means of a condensing mechanism (Algorithm 2). Once the preprocessing step has been carried out, the knowledge base of the kNN classifier is initialized on the features extracted from the preprocessed training set $\mathcal{T}$, i.e., the training set of $\mathcal{K}$ is $\varsigma \circ \varrho(\mathcal{T})$. Section 5 will detail the configuration stage.

After the completion of the configuration stage, the TML-CD solution $\mathcal{D} = \mathcal{K} \circ \varsigma \circ \varrho$ enters the *testing* stage where it is able to operate on the novel incoming samples provided by the data-generating process $\mathcal{P}$ (lines 4–6). At each time instant $t = 1, 2, \ldots$, the proposed solution $\mathcal{D}$ receives in input x_t and provides the output $\hat{y}_t = \mathcal{D}(x_t)$ (line 5). Then, when the supervised information y_t about x_t is made available as per the "test-and-train" approach, it activates the adaptation step (line 6). Section 6 will detail the testing stage by describing the proposed three adaptive mechanisms for TML-CD.

5 The Configuration Stage: Condensing $\mathcal{T}$

The kNN classifier has the great advantage of not requiring a proper training phase. However, this advantage comes, in principle, at the expense of the following two drawbacks. First, a kNN-based classifier requires to store all the data

of the training set. Second, the larger the amount of the training data, the higher the time to provide a classification in output. These drawbacks are more severe as the samples within $\mathcal{T}$ increase.

The related literature addresses these two issues from three different perspectives.

First, condensing techniques [49], [50] aim at identifying the smallest subset of training data that can correctly classify all the training samples. Second, editing techniques [51], [52], [53] instead reduce the number of stored samples by removing the noisy ones, i.e., those not agreeing with their neighborhoods. Third, [54] proposed to train a supervised parametric classifier on available data and to remove all the samples having a classification probability below a hard threshold.

In this work, we focus on the first approach and, in particular, on the Condensed Nearest Neighbour algorithm. In more detail, $\mathcal{D}$ applies this algorithm during the preprocessing step (Algorithm 1–Line 1) in order to optimize both the memory and computational requirements of the classifier $\mathcal{K}$. More specifically, given a N-dimensional training set $\mathcal{T} = \{(x_t, y_t), t = 1, \ldots, N\}$, the preprocessing step computes the condensed representation of T, i.e., the minimum subset $\bar{\mathcal{T}} \subseteq \mathcal{T}$ for which $\mathcal{K}$ is able to correctly classify all the samples in $\mathcal{T}$. Algorithm 2 shows the pseudo-code of the condensing algorithm proposed by Hart [49] that is employed in the preprocessing step. Section 7.4 experimentally evaluates the impact on accuracy and memory demand of this condensing stage, highlighting that the significant savings in terms of memory come at the expense of a negligible drop in accuracy (in stationary conditions).

6 THE TESTING STAGE: ADAPTING $\mathcal{T}$

The adaptation module, which is the core of the proposed $\mathcal{D}$, has been declined from three different perspectives, differing in the type of adaptation mechanism therein employed:

- **Passive Update** (Section 6.1). The adaptation module relies on a fully passive approach where the adaptation is carried out at each new incoming supervised samples, without requiring an explicit detection of a change in the data-generating process $\mathcal{P}$;
- **Active Update** (Section 6.2). This adaptation module relies on a CDT to detects changes in $\mathcal{P}$. Once a change is detected, the algorithm adapts $\mathcal{K}$ accordingly;
- **Hybrid Update** (Section 6.3). The hybrid adaptation module integrates the passive approach with a CDT to speed up the adaptation stage exactly when needed.

6.1 Passive Update: the Condensing-in-Time approach

The passive approach, called *Condensing-in-Time* (CIT) algorithm, updates the training set $\mathcal{T}$ every time a new supervised sample is available. Algorithm 3 presents the CIT algorithm.

It receives in input the feature extractor along with a dimensionality reduction operator $\varsigma \circ \varrho$ and a training set $\mathcal{T}$, whose condensed representation $\bar{\mathcal{T}} \subseteq \mathcal{T}$ (see Algorithm 2) is used to initialize the training set $\mathcal{T}$ of the kNN. Once

Algorithm 3: The Condensing-in-Time (Passive).

Input: Training Set $\mathcal{T}$, Feature Extractor $\varsigma \circ \varrho$.
Parameters : Maximum number of training samples p.
1 Compute $\mathcal{T} \leftarrow \bar{\mathcal{T}}$ with Algorithm 2. ▷ Condense $\mathcal{T}$.
2 Initialize the k-NN classifier $\mathcal{K}$ with $\varsigma \circ \varrho(\mathcal{T})$.
3 Define $\mathcal{D} = \mathcal{K} \circ \varsigma \circ \varrho$.
▷ Loop over samples arriving at time t.
4 **foreach** $(x_t, y_t) \sim \mathcal{P}, t = 1, 2, \ldots$ **do**
5 Predict $\hat{y}_t \leftarrow \mathcal{D}(x_t)$
6 **if** $\hat{y}_t \neq y_t$ **then** ▷ Passive Update.
7 $\mathcal{T} \leftarrow \mathcal{T} \cup (x_t, y_t)$
8 **if** $|\mathcal{T}| > p$ **then** ▷ Window Size Check.
9 $(x_{\bar{t}}, y_{\bar{t}}) \leftarrow \arg\min_{\bar{t}} \{(x_{\bar{t}}, y_{\bar{t}}) \in \mathcal{T}\}$
10 $\mathcal{T} \leftarrow \mathcal{T} \setminus \{(x_{\bar{t}}, y_{\bar{t}})\}$
11 Update $\mathcal{D}$ with $\mathcal{T}$

initialized, the CIT–$\mathcal{D}$, i.e., the $\mathcal{D}$ solution implementing the CIT adaption module, is ready to classify novel incoming samples. The CIT passively updates the $\mathcal{K}$'s knowledge set $\mathcal{T}$ at every time instant t for which the supervised information, i.e., the true label y_t, is available. More in details, CIT–$\mathcal{D}$ adds the sample x_t and its true label y_t (at time instant t) to the kNN $\mathcal{K}$ knowledge set $\mathcal{T}$ if and only if x_t is misclassified, i.e., $\hat{y}_t \neq y_t$ (Algorithm 3, Lines 6–11). This idea is inspired by the condensing algorithm update, shown at Lines 6–9 in Algorithm 2, but it is here tailored to the time evolution of the data-generating process.

It is worth noting that the CIT algorithm can only add a new supervised sample to the knowledge set $\mathcal{T}$ of the kNN $\mathcal{K}$, hence potentially introducing critical issues in the memory and computational demand of the kNN when the number of samples in $\mathcal{T}$ increases. To keep under control the cardinality of $\mathcal{T}$, the CIT algorithm employs two different solutions. The former introduces a maximum number of samples p that can be stored, i.e., $|\mathcal{T}| \leq p$, being $|\cdot|$ cardinality operator. Hence, every time the adaptation stage introduces a sample in $\mathcal{T}$ overcoming this limit, the oldest sample is removed (Algorithm 3, Lines 8–10). As a consequence, the solution $\mathcal{D}$ based on the CIT classifier operates on the last p supervised samples introduced in $\mathcal{T}$. Besides, this mechanism allows also to remove old samples in $\mathcal{T}$ by introducing only misclassified samples, i.e., those bringing more information to the classifier.

The latter introduces a probability for a misclassified sample to be added to the training set $\mathcal{T}$. Ideally, such probability should be close to zero in stationary conditions and close to one immediately after a change. The definition of this probabilistic memory management mechanism is left as future work.

6.2 Active Update: Active Tiny kNN

The Active Tiny kNN, whose pseudocode is shown in Algorithm 4, relies on a Change Detection Test (CDT) ϑ to detect changes in the data generation process $\mathcal{P}$. The core of this algorithm is the ability to adapt the classifier $\mathcal{K}$'s training set $\mathcal{T}$ only after the detection of a concept drift. In addition, the Active Tiny kNN allocates space for a history window W of size ϖ, being ϖ a parameter of the algorithm (described in the sequel).

Algorithm 4: The Active Tiny kNN.

Input: Feature Extractor $\varsigma \circ \varrho$, CDT ϑ, Training Set $\mathcal{T}$.
Parameters : History Window Size ϖ, CDT threshold h.

1 Compute $\mathcal{T} \leftarrow \bar{\mathcal{T}}$ with Algorithm 2. ▷ `Condense` $\mathcal{T}$.
2 Initialize the k-NN classifier $\mathcal{K}$ with $\varsigma \circ \varrho(\mathcal{T})$.
3 Define $\mathcal{D} = \mathcal{K} \circ \varsigma \circ \varrho$.
4 Initialize $W \leftarrow \emptyset$. ▷ `History Window.`
▷ `Loop over samples arriving at time` t.
5 **foreach** $(x_t, y_t) \sim \mathcal{P}, t = 1, 2, \dots$ **do**
6   Predict $\hat{y}_t \leftarrow \mathcal{D}(x_t)$.
7   $W \leftarrow W \cup \{(x_t, y_t)\}$. ▷ `Update History Window.`
8   **if** $|W| \geq \varpi$ **then**
9     $W \leftarrow W \setminus \{(x_{t-\varpi}, y_{t-\varpi})\}$.
10   Compute CDT metric s_t. ▷ `Active Step.`
11   Apply CDT $g_t \leftarrow \vartheta(s_1, \dots, s_t)$.
12   **if** $g_t \geq h$ **then** ▷ `Change Detection Check.`
13   Estimate Real Change Time t_r.
14     $\mathcal{T} \leftarrow \{(x_{\bar{t}}, y_{\bar{t}}) \in W : \bar{t} \geq t_r\}$. ▷ `Novel Samples.`
15   *[Optional] Condense* $\mathcal{T}$.
16   Update $\mathcal{D}$ with $\mathcal{T}$.

In more detail, for each sample (x_t, y_t) provided by $\mathcal{P}$ at time instant t, the Active Tiny kNN predicts the label $\hat{y}_t = \mathcal{D}(x_t)$ and, when the supervised information y_t is available, the active update is activated (Algorithm 4, Lines 7–16).

At first, it adds the pair (x_t, y_t) to the history window W and discards the oldest pair if the window already contains ϖ pairs (Algorithm 4, Lines 7–9). After that, the Active Tiny kNN computes the figure of merit s_t (at time t) and applies the CDT decision function ϑ to inspect for changes in $\mathcal{P}$ (Algorithm 4, Lines 10–11), i.e., $g_t = \vartheta(s_1, \dots, s_t)$. In the most general situation, the computation of g_t at time t takes into account all the figures of merits computed from $t = 1$. A change is detected in the data-generation process $\mathcal{P}$ when g_t overcomes the detection threshold h, being h a parameter of the algorithm (Algorithm 4, Line 12).

Once a change is detected, the adaptation stage starts (Algorithm 4, Lines 13–16). In the first place, it estimates the time t_r the change occurred at (e.g., with a Change Point Method). After that, it discards from the history window W all the samples older than the estimated change time t_r. The updated history window W (optionally condensed through Algorithm 2) becomes the new $\mathcal{K}$'s training set $\mathcal{T}$.

It is noteworthy to point out that the memory footprint of the Active Tiny kNN is bounded over time since it requires to store the training set $\mathcal{T}$ and history window W of at most ϖ samples (the CDT memory footprint can be neglected). Moreover, since the adaptation stage modifies the knowledge set $\mathcal{T}$ only through copies of the (at most whole) history window W, the total memory footprint cannot overcome twice the memory of the history window W, i.e., that of 2ϖ samples.

Although the solution accepts as input any CDT ϑ, in the context of this paper, ϑ is the well-known and theoretically grounded CUSUM algorithm [24] in its generalized version [55], monitoring the accuracy of the Active Tiny kNN over time. As a consequence, any change in the data-generation process $\mathcal{P}$ is assumed to reflect on the $\mathcal{K}$ classification accuracy.

The generalized CUSUM CDT is designed as follows. Let υ_0 be the stationary classification accuracy (estimated on the first ξ supervised samples in the testing stage, being ξ parameter of the Active Tiny kNN algorithm). A Bernoulli distribution with parameter υ_0 and, in turn, a Binomial distribution with parameters υ_0 and n (with n size of the batches on which the accuracy is computed in the following) model our scenario in stationary conditions. The figure of merit of the CUSUM CDT is the likelihood ratio of the probability distributions modeling the scenario after and before the change, i.e.:

$$s_t = \ln \frac{p_{\upsilon_1}(\zeta_t)}{p_{\upsilon_0}(\zeta_t)}, \tag{1}$$

where υ_1 represents the classification accuracy after the change and ζ_t the realization of the Binomial distribution at time t, i.e., the accuracy on the n supervised samples arrived before time t.[4]

Since the value of υ_1 is a priori unknown, the generalized version of the CUSUM algorithm employs a set Υ_1 containing a grid of possible accuracies υ_1 after change equally spaced from 0 to 1, except for the neighborhood of υ_0.[5] The resulting decision function ϑ is:

$$g_t = \vartheta(s_1, \dots, s_t) = \max_{1 \leq j \leq t} \sup_{\upsilon_1 \in \Upsilon_1} S_j^t(\upsilon_1), \tag{2}$$

where

$$S_j^t(\upsilon_1) = \sum_{i=j}^{t} s_i, \tag{3}$$

represents the sum of the log-likelihood ratio s_ts from time j to time t.

Assuming the parameter n large enough, the considered Binomial distribution can be approximated as a Normal one with mean $n\upsilon_0$ and variance $n\upsilon_0(1-\upsilon_0)$. Hence, the log-likelihood ratio s_t in Equation 1 then becomes:

$$s_t = \frac{\upsilon_1\bar{\upsilon_1} - \upsilon_0\bar{\upsilon_0}}{2n\upsilon_0\bar{\upsilon_0}\upsilon_1\bar{\upsilon_1}}\zeta_t^2 + \frac{\bar{\upsilon_0} - \bar{\upsilon_1}}{\bar{\upsilon_0}\bar{\upsilon_1}}\zeta_t + \\ + \frac{n(\upsilon_0\bar{\upsilon_1} - \upsilon_1\bar{\upsilon_0})}{2\bar{\upsilon_0}\bar{\upsilon_1}} + \ln\sqrt{\frac{\upsilon_0\bar{\upsilon_0}}{\upsilon_1\bar{\upsilon_1}}}, \tag{4}$$

where $\bar{\upsilon_0} = 1 - \upsilon_0$ and $\bar{\upsilon_1} = 1 - \upsilon_1$.

As a final remark, the CUSUM CDT is also endowed with the ability to estimate the change time t_r (and, if desired, of the parameter υ_1 after the change). The estimated change time t_r is indeed the index $\tilde{j}$ maximizing the decision function ϑ in Eq. (2), i.e.:

$$(\tilde{j}, \tilde{\upsilon_1}) = \arg\max_{1 \leq j \leq t} \sup_{\upsilon_1 \in \Upsilon_1} S_j^t(\upsilon_1). \tag{5}$$

6.3 Hybrid Tiny kNN: Integrating Condensing-in-Time and Active Tiny kNN

The core of the proposed hybrid update is to integrate the "condensing-in-time" ability of the passive update with

4. Although the Active Tiny kNN algorithm is general enough to deal with any CDT, in the described CUSUM case with Binomial distribution of size n, the CDT figure of merit is not computed for every supervised samples, but every n. As a consequence, all the $n-1$ s_t values before a window of size n is full are set to zero.

5. The cardinality of Υ_1, i.e., the number of tested values υ_1, is a parameter of the Active Tiny kNN.

Algorithm 5: The Hybrid Tiny kNN.

Input: Feature Extractor $\varsigma \circ \varrho$, CDT ϑ, Training Set $\mathcal{T}$.
Parameters : Maximum $\mathcal{T}$ Size ϖ, CDT threshold h.

```
1  Compute T ← T̄ with Algorithm 2.                  ▷ Condense T.
2  Initialize the k-NN classifier K with ς ∘ ϱ(T).
3  Define D = K ∘ ς ∘ ϱ.
                        ▷ Loop over samples arriving at time t.
4  foreach (x_t, y_t) ~ P, t = 1, 2, ... do
5      Predict ŷ_t ← D(x_t).
6      if ŷ_t ≠ y_t then                             ▷ Passive Update.
7          T ← T ∪ (x_t, y_t)
8          if |T| ≥ ϖ then
9              t_min ← min_t̄ {(x_t̄, y_t̄) ∈ T}
10             T ← T \ {(x_{t_min}, y_{t_min})}.
11         Update D with T.
12     Compute CDT metric s_t.                       ▷ Active Step.
13     Apply CDT g_t ← ϑ(s_1, ..., s_t).
14     if g_t ≥ h then                    ▷ Change Detection Check.
15         Estimate Real Change Time t_r.
16         T ← {(x_t̄, y_t̄) ∈ T : t̄ ≥ t_r}.           ▷ Novel Samples.
17         [Optional] Condense T.
18         Update D with T.
```

the capability to quickly adapt to changes by discarding obsolete knowledge of the active one.

In more detail, the (CIT) passive update continuously adapts $\mathcal{T}$ when supervised information is available, regardless a concept drift occurred (or not). This ability comes at the expense of two weak points. First, there is (in principle) no bound on the memory occupation, although two solutions have been suggested to mitigate the problem. Second, when a change occurs, the passive update does not discard the obsolete knowledge present in $\mathcal{T}$, i.e., samples generated by $\mathcal{P}$ before the concept drift occurred.

On the contrary, the active adaptation provides a bound on the memory occupation (i.e., twice the history window size W) and, in turn, on the required computation. However, similarly to the other active approaches present in the literature [23], [25], [31], the effectiveness of the active adaptation phase is strictly related to the ability to promptly detect the concept drift in $\mathcal{P}$.

The proposed hybrid update aspires at compensating the weak points of passive and active updates by integrating the "condensing-in-time" solution described in Algorithm 2 with the CUSUM-based CDT detailed in Section 6.2. The resulting algorithm, namely the Hybrid Tiny kNN, is shown in Algorithm 5. Here, the inputs and the initialization are the same as Active Tiny kNN. The only difference resides in the fact that the Hybrid Tiny kNN does not allocate a history window, but it relies on the training set $\mathcal{T}$ (whose size is bounded by ϖ) as history window.

Similarly to the algorithms it derives from, the Hybrid Tiny kNN predicts the label $\hat{y}_t = \mathcal{D}(x_t)$ and, when the supervised information y_t is made available, it carries out both a passive (Algorithm 5, Lines 6–11) and an active update (Algorithm 5, Lines 12–18), for each sample (x_t, y_t) generated by $\mathcal{P}$ at time instant t. Although the passive update is equal to that of the "condensing-in-time" algorithm, the active one requires to take into account the effects of the passive updates, which are supposed to increase the classification capability of the algorithm over time (until the accuracy of Hybrid Tiny kNN reaches its maximum value). Consequently, the CUSUM CDT is slightly modified in its set Υ_1, which contains only values that are smaller than υ_0, i.e., the accuracy estimated on an initial window of data. In this way, the hybrid update does not detect as concept drift the increases in the accuracy brought by the passive update (hence focusing on changes inducing a drop in the accuracy). Moreover, the adaptation phase triggered by the CDT involves directly the knowledge set $\mathcal{T}$ of the classifier $\mathcal{K}$, where samples older than the estimated time of change t_r are discarded (Algorithm 5, Lines 15–18).

Summing up, the hybrid update continuously adapts $\mathcal{T}$ over time thanks to the passive adaptation, hence avoiding the risk of non-detecting changes due to false-negative detections of the CDT. At the same time, the active adaptation present in the hybrid update can quickly discard obsolete knowledge when a change is detected and set a bound on the memory footprint of $\mathcal{T}$.

7 Experimental Results

The proposed solutions have been validated on two different application scenarios (described in Section 7.1), two types of concept drift (defined in Section 7.2) and three different Micro-Controller Units (MCUs) from STMicroelectronics (whose technological details are given in Section 7.6). In addition, the rest of the Section is organized as follows. Section 7.3 discusses the experimental settings, whereas Sections 7.4 and 7.5 provides the experimental results. Finally, Section 7.6 presents the porting of the Hybrid Tiny kNN algorithm on the three considered MCUs.

It is crucial to point out that TML in presence of concept drift is a completely new research area and, to the best of our knowledge, this is the first work in the related literature proposing adaptive mechanisms for TML running on MCUs.

7.1 Application Scenarios and Datasets

In the experimental section, the following two application scenarios have been considered:

- the *speech-command identification* scenario whose goal is to correctly recognize an user-speech command present in a one-second long audio clip. For this purpose, the *Synthetic Speech Commands Dataset* [56], [57] has been considered. This dataset comprises 30 classes of commands, corresponding, for example, to "*up*", "*left*", "*yes*", "*go*", or a number from "*zero*" to "*nine*". Moreover, the audio files within the dataset comprise different kinds of voices as well as different types of noisy classes.
- the *image classification* scenario whose goal is to classify an image containing exactly one object. The well-known ImageNet [58] dataset, comprising 1000 classes, has been considered.

7.2 The Considered Concept Drift affecting $\mathcal{P}$

Two different kinds of concept drift affecting the data-generation process $\mathcal{P}$ have been considered:

TABLE 1: The impact of condensing techniques (C) in both the application scenarios and in stationary conditions, when no update is done. A SVM and Neural Network (with one fully-connected layer) classifiers have been added as baselines. The results represent the mean $\pm$ standard deviation accuracy (α_λ) and memory (m_λ) over 20 experiments with $\lambda = \{2, 3, 5\}$ classes. The memory is expressed in terms of number of training samples within $\mathcal{T}$, with $|\mathcal{T}| = 100 \cdot \lambda$.

Algorithm	Speech Command Identication						Image Classification					
	α_2	m_2	α_3	m_3	α_5	m_5	α_2	m_2	α_3	m_3	α_5	m_5
SVM	**0.93±0.05**	138±17	**0.88±0.04**	244±16	**0.81±0.05**	447±18	**0.73±0.07**	188±9	**0.61±0.05**	292±6	**0.46±0.05**	496±4
NN–FC1	0.71±0.14	2	0.75±0.06	3	0.49±0.12	5	0.60±0.08	2	0.45±0.05	3	0.29±0.05	5
kNN	0.89±0.06	200	0.82±0.06	300	0.74±0.07	500	0.66±0.07	200	0.49±0.07	300	0.33±0.06	500
kNN + C	0.88±0.06	**64±20**	0.81±0.04	**128±22**	0.73±0.06	**255±33**	0.66±0.07	**119±18**	0.51±0.06	**214±19**	0.35±0.04	**414±16**

- the addition of noise on x. This type of concept drift models the scenario where a failure on the microphone acquiring the audio clip occurs. To achieve this goal, the noisy variant of each class within the *Synthetic Speech Commands Dataset* is considered after the change;
- a change in the classification problem, i.e., a variation in the set of classes Λ.

7.3 Experimental Settings

In this experimental analysis, the considered feature extractor ϱ refers to the first layer of the well-known ResNet-18 CNN [59]. This layer comprises a convolutional layer with 64 7x7 three-dimensional filters with stride 2, a batch-normalization layer, a ReLU non-linearity, and a 3x3 max-pooling layer with stride 2. The dimensionality reduction operator ς discards 63 out of 64 filters by keeping only the one with the highest mean activation on the ImageNet benchmark. Consequently, the resulting DL model $\varsigma \circ \varrho$ is a single 7x7x3 filter that has 147 parameters and occupies 588B with a 32-bit floating-point representation.

In the *speech-command identification* scenario, the audio waveform (sampled at $f_a = 22050$ Hz) is converted into a spectrogram through a Short-Time Fourier Transform with windows of size $n_{fft} = 512$ and a step $h_l = 512$ and then converted into a colored one by means of a colormap. In the *image classification*, instead, the images are resized to 224x224x3 before being passed as input to $\mathcal{D}$. The resulting one-second audio has a memory footprint of 88 200B, the image 602 112B, whereas the resulting colored spectrogram of size 257x44x3 requires 135 696B.

The change always occurs after half of the available data, i.e., 500 samples per class in the *image classification scenario* and 750 in the *speech command identification* one. Finally, 100 samples per class are provided to all the algorithm as initial training set $\mathcal{T}$, i.e., $|\mathcal{T}| = 100 \cdot \lambda$. Experimental results are averaged over twenty runs.

7.4 Evaluating Effects of the Pre-Processing Through Condensing

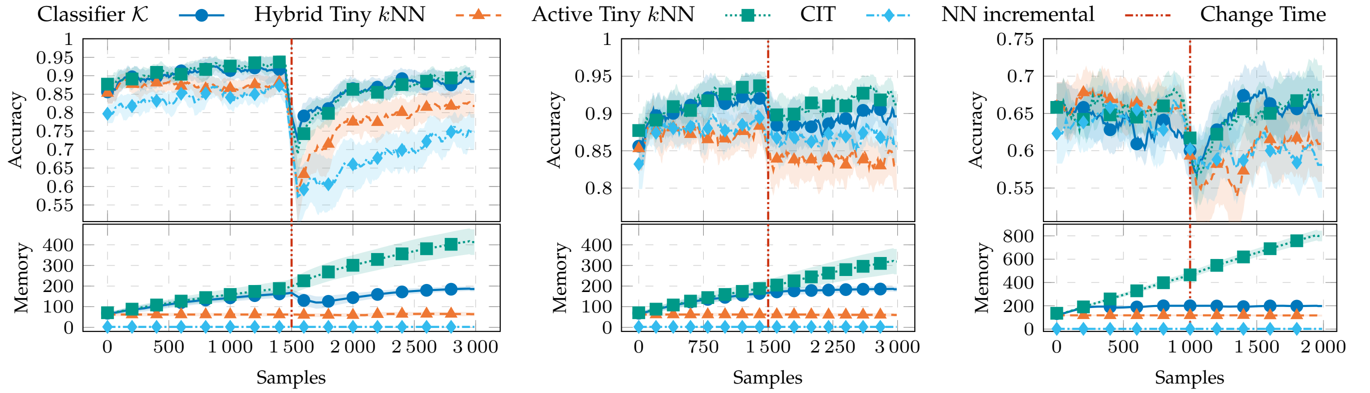
The first aspect considered in this experimental analysis aims at studying the impact of condensing the kNN $\mathcal{K}$ training set $\mathcal{T}$ with Algorithm 2. To achieve this goal, the proposed TML-CD $\mathcal{D}$ is configured with the block $\varsigma \circ \varrho$ presented in Section 7.3 and an initial training set $\mathcal{T}$ with size $|\mathcal{T}| = 100 \cdot \lambda$. Then, the classification capabilities of $\mathcal{D}$ when condensing is employed are evaluated in stationary conditions, i.e., no adaptation is carried out during the operational life of $\mathcal{D}$. In addition to $\mathcal{D}$ without or with the initial condensing (referred to as kNN and kNN+C in the following, respectively), the comparison comprises two well-known classifiers, i.e., a Support Vector Machine (SVM) and a single fully-connected layer neural network classifier (NN–FC1). Both the classifiers are applied on the same features of the kNN $\mathcal{K}$, i.e., those extracted by $\varsigma \circ \varrho$ (on the initial training set $\mathcal{T}$). Moreover, the SVM is trained until convergence, whereas the NN–FC1 is trained for 3 epochs with stochastic gradient descent, no momentum, and a learning rate $\eta \in \left[1e^{-2}, 5e^{-3}, 1e^{-3}, \ldots, 1e^{-5}\right]$. In our experiments, only the best performing NN–FC1 classifier is shown. We emphasize that both the SVM and the NN are characterized by an unfeasible training procedure in MCUs, so they cannot be considered for the on-device training phase.

Table 1 shows the result in the proposed application scenarios with a different number of classes. The SVM and the NN–FC1 classifiers present the highest and worst accuracy in all the considered application scenarios, respectively. The proposed TML solution $\mathcal{D}$ (without condensing) shows accuracies smaller than the SVM classifier by 4 to 8% in *speech command identification* and 7 to 13% in *image classification* scenarios. As expected, condensing the training set $\mathcal{T}$ has a limited impact on the accuracy (at most 1% drop in *speech command identification* scenario), but it allows to reduce the memory requirements significantly. Without considering the NN classifier, $\mathcal{D}$ with condensing is the algorithm providing the lowest memory demand.[6] In the *speech command identification* scenario, the number of stored samples indeed ranges from 32 to 50% of the provided samples (with λ from 2 to 5), representing the 46 to 57% of the ones required by the SVM, i.e., its support vectors. In the *image classification* scenario, the memory saving is significantly lower, with the SVM retaining almost all the samples as support vectors and the condensed $\mathcal{D}$ storing 60 to 82% of them. From now on, the proposed TML-CD $\mathcal{D}$ is assumed to always rely on condensing algorithm in the configuration and testing stages (when available).

7.5 Experimental Results in Presence of Concept Drift

Figure 2 compares the three proposed adaptive algorithms, i.e., CIT, Active Tiny kNN, and Hybrid Tiny kNN, in the two considered scenarios with different numbers of classes

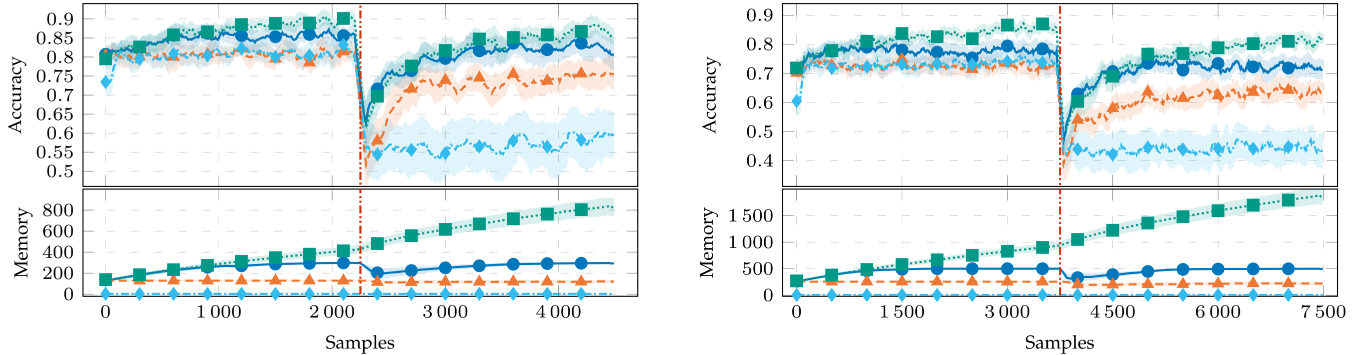
6. The NN–FC1 memory footprint corresponds to that of its weights, which are equal to the number of classes multiplied by the size of classifier inputs. Since the input size is the same for all the classifiers, the NN-FC1 memory is that of λ samples.



(a) 2-class *Speech Command Identification* where a class changes after half samples.

(b) 2-class *Speech Command Identification* where noise affects data after half samples

(c) 2-class *Image Classification* where a class changes after half samples.



(d) 3-class *Speech Command Identification* where a class changes after half samples.

(e) 5-class *Speech Command Identification* where a class changes after half samples.

Fig. 2: The mean accuracy and the number of samples to be kept in $\mathcal{K}$'s training set $\mathcal{T}$ over time of the three proposed adaptive algorithms and a continuously learning Neural Network (with one fully-connected classifier). The plots represent the mean $\pm$ standard deviation over 20 experiments with $\lambda = \{2, 3, 5\}$ classes and the considered scenarios/changes. The algorithms receive as input a training set $\mathcal{T}$, with $|\mathcal{T}| = 100 \cdot \lambda$.

λ. We considered two different figures of merit: the mean $\pm$ std *accuracy* (the curve of each experiment is the convolution of the correct predictions of each experiment with a 100-dimensional filter with all values of 0.01) and *memory footprint*, measured as the number of samples within the training set $\mathcal{T}$, over all the experiments. It is crucial to point out that the memory footprint does not include any other auxiliary source of memory, e.g., the history window W of the Active Tiny-kNN algorithm (that has a size of $\varpi = 100 \cdot \lambda$).

As a comparison, this experimental analysis includes also a continuously learning single-layer fully-connected classifier (NN–FC1) operating on the features extracted by $\varsigma \circ \varrho$ and performing a back-propagation step for each incoming sample. The NN–FC1 is trained for 3 epochs with stochastic gradient descent, no momentum, and a learning rate η, with $\eta \in \left[1e^{-2}, 5e^{-3}, 1e^{-3}, \ldots, 1e^{-5}\right]$. Moreover, during the testing stage, the learning rate for back-propagation might be reduced ($\eta \in \left[1e^{-3}, 5e^{-4}, 1e^{-4}, \ldots, 1e^{-6}\right]$). Among all the possible combinations, Figure 3 shows only the one with the largest accuracy.

In more detail, Figure 2 shows the accuracy and memory footprint in five different configurations: the *Speech Command Identification scenario* where one-class changes with $\lambda = \{2, 3, 5\}$ classes (Figures 2a, 2d, and 2e) and with the introduction of noise with $\lambda = 2$ (Figure 2b), and the *Image Classification* scenario where one-class changes with $\lambda = 2$ (Figure 2c).

The NN-FC1 baseline is the worst algorithm in almost all the cases, with low capabilities of recovering after change when $\lambda > 2$ (Figures 2d and 2e). The noise has a limited impact on the accuracy, as shown in Figure 2b, whose small degradation is detected neither by the Active nor by the Hybrid Tiny kNN algorithms. With the other changes, the proposed algorithms work as expected. On the one hand, the (passive) CIT algorithm continuously improves over time, at the expense of an unbounded memory growth (in these experiments, none of the approaches detailed in Section 6.1 to control it has been considered). Moreover, in all the considered scenarios, the slope of the samples' curve increases at the change time, highlighting the accuracy drop due to the change itself. On the other hand, the Active kNN algorithm is able to recover after a change keeping its memory footprint nearly constant and significantly lower than the size of history window W (not shown in the Figure 2) due to condensing. Finally, the Hybrid Tiny kNN algorithm combines the advantages of both the CIT and the Active Tiny kNN. It can recover faster than the two

TABLE 2: The detailed memory footprint (with a 32-bit data type) of the Hybrid Tiny kNN on the STM32 MCUs. The size of the training set $\mathcal{T}$ includes those of samples (see $\varsigma \circ \varrho$ output), their labels (32bit), and their timestamps (32bit).

(a) STM32H743 and STM32F767.

	Size	Memory Footprint (B)
Audio ($t_a = 1$ s, $f_a = 22050$ Hz)	1x22050	88 200
Spectrogram ($n_{fft} = h_l = 512$)	257x44x3	135 696
$\varsigma \circ \varrho$ (1 convolutional filter 7x7x3)	7x7x3	588
$\varsigma \circ \varrho$ output	65x11	2 860
$\mathcal{K}$'s Training Set $\mathcal{T}$	50	143 400
Total		370 744

(b) STM32F401.

	Size	Memory Footprint (B)
Audio ($t_a = 1$ s, $f_a = 4410$ Hz)	1x4410	17 640
Spectrogram ($n_{fft} = h_l = 128$)	64x35x3	27 300
$\varsigma \circ \varrho$ (1 convolutional filter 7x7x3)	7x7x3	588
$\varsigma \circ \varrho$ output	17x9	612
$\mathcal{K}$'s Training Set $\mathcal{T}$	50	31 000
Total		77 140

TABLE 3: The experimental execution times, measured in milliseconds, on the STM32 MCUs. The measured times are: the spectrogram processing t_p, the feature extraction $t_{\varsigma \circ \varrho}$, and the $\mathcal{K}$ prediction with 10 and 50 samples within $\mathcal{T}$. The time $t_{\mathcal{K},50}$ also shows the worst prediction+adaptation time.

MCU	t_p	$t_{\varsigma \circ \varrho}$	$t_{\mathcal{K},10}$	$t_{\mathcal{K},50}$
STM32H743ZI	22.9	18.6	2.0	2.9–6.2
STM32F767ZI	53.8	41.5	2.2	4.0–12.1
STM32F401RE	34.6	43.1	2.2	4.6–136.0

other algorithms in all the considered scenarios and keep the memory footprint under control (by taking into account also the Active Tiny kNN history window memory footprint, the Hybrid Tiny kNN has the lowest footprint). Moreover, it shows the best accuracy, being overcome by CIT only when it saturates the maximum size of its training set $\mathcal{T}$ that is here fixed to $\varpi = 100 \cdot \lambda$. This effect is visible in particular in Figures 2d and 2e.

7.6 Porting the Hybrid Tiny kNN on the STM32 MCUs

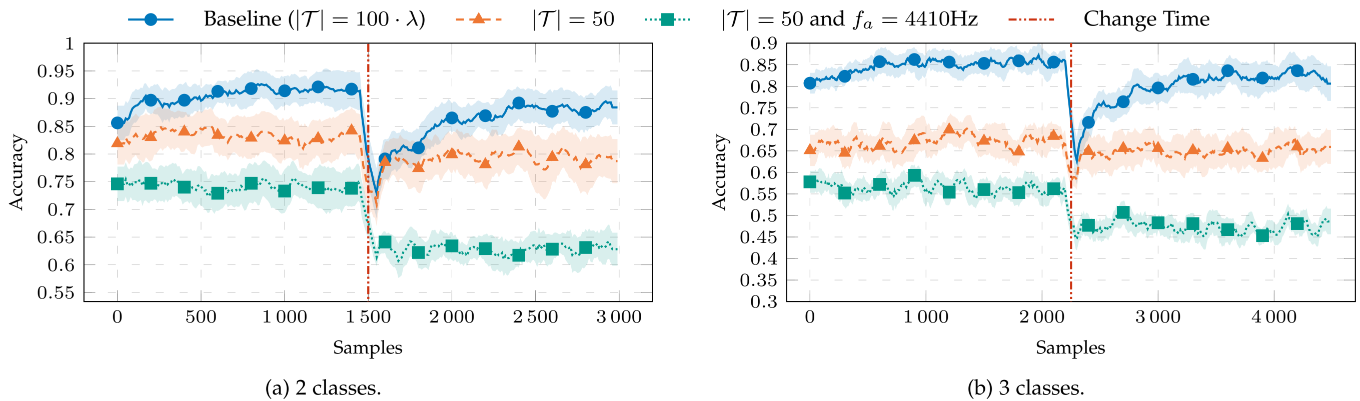
The aim of this section is to show the technological feasibility of the proposed Hybrid Tiny kNN algorithm in the *Speech command identification* scenario. To achieve this goal, we considered the following three different MCUs:

- The *STM32H743* is a high-performance MCU having a 480 MHz Cortex-M7 processor, 1024 KB of RAM (split into five blocks of different speed), and 2048 KB of Flash memory;
- The *STM32F767* is a high-performance MCU having a 216 MHz Cortex-M7 processor, 512 KB of RAM, and 2048 KB of Flash memory;
- The *STM32F401* is a general-purpose MCU having a 84 MHz Cortex-M4 processor, 96 KB of RAM, and 512 KB of Flash memory.

The main technological constraint imposed by such board is the one on the memory, i.e., the maximum memory footprint of the Hybrid kNN algorithm cannot overcome the available RAM of each MCU (in the case of *STM32H743* that limit is lowered to 512KB, i.e., the size of the fastest RAM block). To satisfy this memory constraint, we set the maximum training set size of the Hybrid Tiny kNN algorithm to $|\mathcal{T}| = 50$. In addition, for the *STM32F401* board, the sampling frequency f_a is reduced from $f_a = 22050$Hz to $f_a = 4410$Hz as suggested in [60]. This guarantees a strong reduction in the memory footprint of the input audio (17 640B), the generated spectrogram with windows of size $n_{fft} = 128$ and step $h_l = 128$ (65x35x3 occupying 27 300B), and on the output of the feature extractor (17x9 occupying 588B). Table 2 details the memory footprint of the Hybrid Tiny kNN deployed for the STM32H743 and STM32F767 (Table 2a) and the STM32F401 (Table 2b).

Figure 3 shows the effects of such technological choices in the considered scenario when a class change after half samples with $\lambda = \{2, 3\}$. In that figure, the baseline is the Hybrid Tiny kNN algorithm introduced in Section 7.5, where $\varpi = 100 \cdot \lambda$. The algorithms deployed on the MCUs exhibit a constant accuracy until the change with a minimum gain due to passive adaptation (limited by the constraint on the training set size). The gap w.r.t. the baseline algorithm is between 4 and 10% (15% to 25% when varying also the acquisition frequency). After the change, the algorithm with $|\mathcal{T}| = 50$ recovers as fast as the baseline, then the effects of the passive updates create a gap in the accuracy. The algorithm that considers a variation also in the sampling frequency, instead, shows a larger drop in accuracy (about 10 to 15%), but its passive updates are able to recover after the change, slightly improving the accuracy over time.

Table 3 reports the experimental execution times of the $\mathcal{D}$ blocks in the considered MCUs. More in detail, the measured quantities are: the processing time t_p needed to transform the acquired 1s audio into a spectrogram, the feature extractor and dimensionality reduction blocks' execution $t_{\varsigma \circ \varrho}$, and the kNN $\mathcal{K}$ prediction time $t_{\mathcal{K},|\mathcal{T}|}$ with two different cardinalities of its training set, 10 and 50 (the latter also shows the worst measured prediction time when an adaptation has been made). Results are particularly interesting. In particular, the processing and the feature extraction are the two predominant times, requiring 41.5ms and 95.3ms on a high-performance MCUs (the STM32H7 and the STM32F7) and 77.7ms on a general-purpose one (the STM32F4, although on a smaller spectrogram). The $\mathcal{K}$'s prediction and the adaptation, when employed, are negligible w.r.t. the other times, being the 7 and the 15% on the STM32H7, the 4% and the 13% on the STM32F7, and the 6 and the 175% on the STM32F4 (whose adaptation is the only exception). As a final remark, the total time required from the processing of the acquired audio to the final prediction, including the possible adaptations, is significantly lower than that of the acquisition, showing the effectiveness of the proposed Hybrid Tiny kNN algorithm on three real MCUs.



(a) 2 classes.

(b) 3 classes.

Fig. 3: The mean accuracy of the Hybrid Tiny kNN when satisfying the technological requirements of the STM32 boards. The plots represent the mean $\pm$ standard deviation over 20 experiments in the 2-class *Speech Command Identification* where a class changes after half samples. The algorithms receive as input a training set $\mathcal{T}$, with $|\mathcal{T}| = 50$.

8 Conclusion

For the first time in the literature, this paper introduced an adaptive Tiny Machine Learning solution for Concept Drift. This solution, characterized by a hybrid approach integrating an active and a passive adaptation step, takes into account the technological constraints on memory, computation, and energy typically characterizing embedded systems and IoT units it runs on. The proposed solution has been deployed to three different micro-controller units with 96 to 512KB of RAM, showing its feasibility in real-world scenarios and on off-the-shelf technological units.

Future work will encompass the definition of advanced memory control mechanisms on passive updates (e.g., by deepening the suggested probabilistic approach), further optimization of the kNN memory requirements, the definition of learning mechanisms at the feature extractor block, and the exploration of sparse or quantized solutions for the TML algorithms.

References

[1] C. Alippi, S. Disabato, and M. Roveri, “Moving Convolutional Neural Networks to Embedded Systems: The AlexNet and VGG-16 Case,” in *2018 17th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. Porto: IEEE, apr 2018, pp. 212–223.

[2] R. Sanchez-Iborra and A. F. Skarmeta, “Tinyml-enabled frugal smart objects: Challenges and opportunities,” *IEEE Circuits and Systems Magazine*, vol. 20, no. 3, pp. 4–18, 2020.

[3] J. Tang, D. Sun, S. Liu, and J.-L. Gaudiot, “Enabling deep learning on iot devices,” *Computer*, vol. 50, no. 10, pp. 92–96, 2017.

[4] C. R. Banbury, V. J. Reddi, M. Lam, W. Fu, A. Fazel, J. Holleman, X. Huang, R. Hurtado, D. Kanter, A. Lokhmotov *et al.*, “Benchmarking tinyml systems: Challenges and direction,” *arXiv preprint arXiv:2003.04821*, 2020.

[5] H. Cai, C. Gan, L. Zhu, and S. Han, “Tinytl: Reduce memory, not parameters for efficient on-device learning,” *Advances in Neural Information Processing Systems*, vol. 33, 2020.

[6] S. Disabato and M. Roveri, “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*, 2020, pp. 7–13.

[7] G. Ditzler, M. Roveri, C. Alippi, and R. Polikar, “Learning in nonstationary environments: A survey,” *IEEE Computational Intelligence Magazine*, vol. 10, no. 4, pp. 12–25, November 2015.

[8] J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia, “A survey on concept drift adaptation,” *ACM Computing Surveys (CSUR)*, vol. 46, no. 4, p. 44, 2014.

[9] J. Lu, A. Liu, F. Dong, F. Gu, J. Gama, and G. Zhang, “Learning under concept drift: A review,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 12, pp. 2346–2363, 2018.

[10] R. Elwell and R. Polikar, “Incremental learning of concept drift in nonstationary environments,” *IEEE Transactions on Neural Networks*, vol. 22, no. 10, pp. 1517–1531, 2011.

[11] B. Krawczyk and M. Woźniak, “One-class classifiers with incremental learning and forgetting for data streams with concept drift,” *Soft Computing*, vol. 19, no. 12, pp. 3387–3400, 2015.

[12] G. Hulten, L. Spencer, and P. Domingos, “Mining time-changing data streams,” in *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD ’01. New York, NY, USA: ACM, 2001, pp. 97–106.

[13] W. N. Street and Y. Kim, “A streaming ensemble algorithm (sea) for large-scale classification,” in *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD ’01. New York, NY, USA: ACM, 2001, pp. 377–382.

[14] P. R. Almeida, L. S. Oliveira, A. S. Britto Jr, and R. Sabourin, “Adapting dynamic classifier selection for concept drift,” *Expert Systems with Applications*, vol. 104, pp. 67–85, 2018.

[15] L. Pietruczuk, L. Rutkowski, M. Jaworski, and P. Duda, “How to adjust an ensemble size in stream data mining?” *Information Sciences*, vol. 381, pp. 46–54, 2017.

[16] H. Li, P. Barnaghi, S. Enshaeifar, and F. Ganz, “Continual learning using bayesian neural networks,” *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[17] G. I. Parisi, R. Kemker, J. L. Part, C. Kanan, and S. Wermter, “Continual lifelong learning with neural networks: A review,” *Neural Networks*, vol. 113, pp. 54–71, 2019.

[18] B. Pérez-Sánchez, O. Fontenla-Romero, and B. Guijarro-Berdiñas, “A review of adaptive online learning for artificial neural networks,” *Artificial Intelligence Review*, vol. 49, no. 2, pp. 281–299, 2018.

[19] G. Ditzler and R. Polikar, “Hellinger distance based drift detection for nonstationary environments,” in *Computational Intelligence in Dynamic and Uncertain Environments (CIDUE), 2011 IEEE Symposium on*. IEEE, 2011, pp. 41–48.

[20] T. Dasu, S. Krishnan, S. Venkatasubramanian, and K. Yi, “An information-theoretic approach to detecting changes in multi-dimensional data streams,” in *In Proc. Symp. on the Interface of Statistics, Computing Science, and Applications*, 2006.

[21] L. Bu, C. Alippi, and D. Zhao, “A pdf-free change detection test based on density difference estimation,” *IEEE transactions on neural networks and learning systems*, vol. 29, no. 2, pp. 324–334, 2016.

[22] P. Duda, L. Rutkowski, M. Jaworski, and D. Rutkowska, “On the parzen kernel-based probability density function learning procedures over time-varying streaming data with applications to pattern classification,” *IEEE transactions on cybernetics*, vol. 50, no. 4, pp. 1683–1696, 2018.

[23] M. Baena-Garcıa, J. del Campo-Ávila, R. Fidalgo, A. Bifet, R. Gavalda, and R. Morales-Bueno, “Early drift detection

method," in *Fourth international workshop on knowledge discovery from data streams*, vol. 6, 2006, pp. 77–86.

[24] E. S. Page, "Continuous inspection schemes," *Biometrika*, vol. 41, no. 1/2, pp. 100–115, 1954.

[25] X. Wang, Q. Kang, M. Zhou, L. Pan, and A. Abusorrah, "Multiscale drift detection test to enable fast learning in nonstationary environments," *IEEE Transactions on Cybernetics*, 2020.

[26] D. Zambon, C. Alippi, and L. Livi, "Concept drift and anomaly detection in graph streams," *IEEE transactions on neural networks and learning systems*, vol. 29, no. 11, pp. 5592–5605, 2018.

[27] C. Alippi, G. Boracchi, and M. Roveri, "Just-in-time classifiers for recurrent concepts," *IEEE transactions on neural networks and learning systems*, vol. 24, no. 4, pp. 620–634, 2013.

[28] C. C. Aggarwal, "On biased reservoir sampling in the presence of stream evolution," in *Proceedings of the 32nd international conference on Very large data bases*. VLDB Endowment, 2006, pp. 607–618.

[29] J. S. Vitter, "Random sampling with a reservoir," *ACM Transactions on Mathematical Software (TOMS)*, vol. 11, no. 1, pp. 37–57, 1985.

[30] R. Klinkenberg, "Learning drifting concepts: Example selection vs. example weighting," *Intelligent data analysis*, vol. 8, no. 3, pp. 281–300, 2004.

[31] S. Disabato and M. Roveri, "Learning convolutional neural networks in presence of concept drift," in *2019 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2019, pp. 1–8.

[32] Z. Yang, S. Al-Dahidi, P. Baraldi, E. Zio, and L. Montelatici, "A novel concept drift detection method for incremental learning in nonstationary environments," *IEEE transactions on neural networks and learning systems*, vol. 31, no. 1, pp. 309–320, 2019.

[33] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *The IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[34] S. Lin, R. Ji, C. Chen, D. Tao, and J. Luo, "Holistic cnn compression via low-rank decomposition with knowledge transfer," *IEEE transactions on pattern analysis and machine intelligence*, vol. 41, no. 12, pp. 2889–2905, 2018.

[35] A. Bulat and G. Tzimiropoulos, "Bit-mixer: Mixed-precision networks with runtime bit-width selection," *arXiv preprint arXiv:2103.17267*, 2021.

[36] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *International Conference on Machine Learning*, 2015, pp. 1737–1746.

[37] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European conference on computer vision*. Springer, 2016, pp. 525–542.

[38] C. Banbury, C. Zhou, I. Fedorov, R. Matas, U. Thakker, D. Gope, V. Janapa Reddi, M. Mattina, and P. Whatmough, "Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers," *Proceedings of Machine Learning and Systems*, vol. 3, 2021.

[39] S. Gopinath, N. Ghanathe, V. Seshadri, and R. Sharma, "Compiling kb-sized machine learning models to tiny iot devices," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 79–95.

[40] A. Kumar, S. Goyal, and M. Varma, "Resource-efficient machine learning in 2 kb ram for the internet of things," in *International Conference on Machine Learning*, 2017, pp. 1935–1944.

[41] J. Lin, W.-M. Chen, Y. Lin, C. Gan, S. Han *et al.*, "Mcunet: Tiny deep learning on iot devices," *Advances in Neural Information Processing Systems*, vol. 33, pp. 11 711–11 722, 2020.

[42] I. Fedorov, M. Stamenovic, C. Jensen, L.-C. Yang, A. Mandell, Y. Gan, M. Mattina, and P. N. Whatmough, "Tinylstms: Efficient neural speech enhancement for hearing aids," *arXiv preprint arXiv:2005.11138*, 2020.

[43] M. Venzke, D. Klisch, P. Kubik, A. Ali, J. D. Missier, and V. Turau, "Artificial neural networks for sensor data classification on small embedded systems," *arXiv preprint arXiv:2012.08403*, 2020.

[44] I. Fedorov, R. P. Adams, M. Mattina, and P. Whatmough, "Sparse: Sparse architecture search for cnns on resource-constrained microcontrollers," in *Advances in Neural Information Processing Systems*, 2019, pp. 4977–4989.

[45] M. Rusci, M. Fariselli, A. Capotondi, and L. Benini, "Leveraging automated mixed-low-precision quantization for tiny edge microcontrollers," in *IoT Streams for Data-Driven Predictive Maintenance and IoT, Edge, and Mobile for Embedded Machine Learning*. Springer, 2020, pp. 296–308.

[46] N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992.

[47] V. Losing, B. Hammer, and H. Wersing, "Knn classifier with self adjusting memory for heterogeneous concept drift," in *2016 IEEE 16th international conference on data mining (ICDM)*. IEEE, 2016, pp. 291–300.

[48] M. Roseberry and A. Cano, "Multi-label knn classifier with self adjusting memory for drifting data streams," in *Second International Workshop on Learning with Imbalanced Domains: Theory and Applications*. PMLR, 2018, pp. 23–37.

[49] P. Hart, "The condensed nearest neighbor rule (corresp.)," *IEEE transactions on information theory*, vol. 14, no. 3, pp. 515–516, 1968.

[50] I. Tomek, "Two modifications of cnn," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-6, no. 11, pp. 769–772, 1976.

[51] J. Laurikkala, "Improving identification of difficult small classes by balancing class distribution," in *Conference on Artificial Intelligence in Medicine in Europe*. Springer, 2001, pp. 63–66.

[52] I. Tomek, "An experiment with the edited nearest-neighbor rule," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-6, no. 11, p. 448–452, 1976.

[53] D. L. Wilson, "Asymptotic properties of nearest neighbor rules using edited data," *IEEE Transactions on Systems, Man, and Cybernetics*, no. 3, pp. 408–421, 1972.

[54] M. R. Smith, T. Martinez, and C. Giraud-Carrier, "An instance level analysis of data complexity," *Machine learning*, vol. 95, no. 2, pp. 225–256, 2014.

[55] G. Lorden *et al.*, "Procedures for reacting to a change in distribution," *The Annals of Mathematical Statistics*, vol. 42, no. 6, pp. 1897–1908, 1971.

[56] J. Buchner, "Synthetic speech commands: A public dataset for single-word speech recognition." *Dataset available from https://www.kaggle.com/jbuchner/synthetic-speech-commands-dataset/*, 2017.

[57] P. Warden, "Speech commands: A dataset for limited-vocabulary speech recognition," *arXiv preprint arXiv:1804.03209*, 2018.

[58] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[59] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[60] S. Disabato, G. Canonaco, P. G. Flikkema, M. Roveri, and C. Alippi, "Birdsong detection at the edge with deep learning," in *2021 IEEE International Conference on Smart Computing (SMARTCOMP)*. IEEE, 2021.