# ML-MCU: A Framework to Train ML Classifiers on MCU-based IoT Edge Devices

Bharath Sudharsan, John G. Breslin, and Muhammad Intizar Ali

***Abstract*—The majority of IoT edge devices are embedded systems with a tiny microcontroller unit (MCU), which acts as its brain. When users want their edge devices to continuously improve for better edge-analytics results, there is a need to equip their devices with algorithms that can learn/train from the continuously evolving real-world data. Currently, such devices are not capable of executing any Machine Learning (ML)-based model training tasks due to their resource constraints such as: limited memory (SRAM, Flash and EEPROM), low operations per second, its inability to perform parallel processing, etc. In this paper, we provide *ML-MCU*, a framework with our novel *Opt-SGD* and *Opt-OVO* algorithms to enable both binary and multi-class ML classifier training directly on MCUs. Thus, *ML-MCU* enables billions of MCU-based IoT edge devices to self learn/train (offline) after their deployment, using live data from a wide range of IoT use-cases. When evaluating our algorithms on multiple popular MCUs, using various datasets of different sizes and feature dimensions, one of the most exciting findings was, our *Opt-OVO* algorithm trained a multi-class classifier using a dataset of size 1250 and class count 50, on a $3 resource-constrained MCU and also performed onboard unit inference for the same 50 class data in super real-time (6.2 ms).**

***Index Terms*—Self-learning IoT devices, Intelligent Microcontrollers, Real-time Machine Learning, Edge Intelligence.**

## I. Introduction

In the real-world, every new scene generates unseen data patterns. When an ML model deployed over edge devices sees any fresh patterns which were not previously exposed during the training phase, it will either not know how to react to that specific scenario or can lead to false or less accurate results [1]. Furthermore, a model trained using data from one context often does not produce the expected results when deployed in another context. Certainly, it is not feasible to train multiple models for multiple environments and contexts. In order to achieve a truly autonomous local intelligence at the device level, the devices must have the ability to *self-learn* and *understand* the data patterns offline, with no dependency on users or cloud services. These devices should be capable of locally gathering knowledge incrementally during the training/learning phase and should self-learn. Thus, transforming edge devices into intelligent machines capable of learning and performing analytics in any given environment [2].

In most real-life IoT scenarios, designing an AI solution and deploying on edge devices to solve a target problem is a lengthy and expensive process that demands statistics, data science skills, and access to complex datasets that are difficult to source (GDPR and privacy concerns) [3]. A large amount of historical data is collected and stored at a central location before training ML models over these large datasets. However, such approaches are only able to train and infer based on the collected data. Once trained, the models are deployed across the board at all devices to perform analytics at the edge. In cases where the historical data becomes obsolete, or it is not truly representative for possible cases, the edge analytics produces inferences at a low accuracy whenever previously unseen data is processed. In our work, we are proposing *Machine Learning - Microcontroller Unit* (*ML-MCU*), which is an incremental method to train and infer at the device level without the need for any cloud-based ML training services. Whenever *ML-MCU* is deployed at the device level, it trains itself locally to build knowledge *on-the-fly* using the live IoT data streams, thus transforming IoT devices into intelligent devices which can train and infer offline at the edge.

The state-of-the-art ML frameworks do not enable training models on resource-constrained devices like MCUs, small CPUs, and FPGAs since executing the frameworks alone requires hundreds of MB for storage, high memory-resource, file system support, high clock speeds, multiple cores & parallel execution units, etc. MCUs are resource-constrained and can not afford to have the high specification required by the modern ML frameworks [4, 5]. Also, since the memory (SRAM, Flash, and EEPROM) of MCUs, which is the brain for billions of edge devices deployed worldwide, is limited to a few MB, an upper bound is imposed, thus restricting onboard model training using high features and large trainsets. Hence, our optimized training algorithms should also be capable of un-restricting this upper bound to enable utilizing the full *n-samples* of the dataset during onboard model training.

To enable MCU-based IoT edge devices to self learn/train (offline) after their deployment, without restrictions, using the full *n-samples* of live IoT use-case data, we provide *ML-MCU* and open-sourced its implementation [1]. To the best of our knowledge, our work is one of the few recent novel approaches enabling a model's training tasks on MCUs. Our main contributions in this paper are as follows:

- We provide an algorithm named *Optimised-Stochastic Gradient Descent* (*Opt-SGD*) as a part of our *ML-MCU* framework to enable high performance, resource-friendly binary classifiers training on small edge MCUs. Our method combines benefits from both Gradient Descent

B. Sudharsan and J.G. Breslin are with the Confirm SFI Centre for Smart Manufacturing, Data Science Institute, National University of Ireland Galway (e-mail: b.sudharsan1@nuigalway.ie; john.breslin@nuigalway.ie). M.I. Ali is with the School of Electronic Engineering, Dublin City University (e-mail: ali.intizar@dcu.ie).

[1]The C++ implementation of ML-MCU is available at: https://github.com/bharathsudharsan/ML-MCU.

(GD) and Stochastic Gradient Descent (SGD) thus, inheriting the stability of GD while retaining the work-efficiency of SGD.

- We provide an algorithm named *Optimized One-Versus-One* (*Opt-OVO*) as a part of our *ML-MCU*. To the best of our knowledge, this is the first novel algorithm to enable multi-class classifiers training on MCUs. Our *Opt-OVO* archives reduced computation by identifying and removing base classifiers that lack significant contributions to the overall multi-class classification result. For further efficiency improvement, we use our *Opt-SGD* method inside *Opt-OVO* for training the *B* number of base learners that decompose one multi-class problem into multiple binary problems. We also provide the flexibility for users to replace our *Opt-SGD* method with a base learner method of their choice, i.e., SVM, LDA, etc.
- We designed both our algorithms to be capable of training classifiers incrementally, thus enabling edge devices to self-learn utilizing the full *n-samples* of high-dimensional IoT use case data. When updating/re-training the existing model with a new set of data, our algorithms do not require the previous training data because it is capable of updating the former version model without erasing the information it learned from the previous datasets.

**Outline.** The rest of the paper is organized as follows. Section II presents the *ML-MCU* framework that contains the *Opt-SGD* and *Opt-OVO* algorithms. In Section III, we perform an extensive evaluation of the framework algorithms. Section IV briefs the related studies. Section V concludes the paper with an outline of future directions.

## II. ML-MCU Framework

Here we present our *ML-MCU* framework, which enables training ML classifiers directly on MCU boards. When users want their edge devices to learn from the new data patterns it sees after deployment, they just need to use any of our framework's algorithms that we present in the upcoming sections [2]. In Fig. 1, we show our *ML-MCU* framework. In such an online, incremental training setup, to facilitate feeding the live data stream directly to the *Opt-SGD* or *Opt-OVO* framework algorithms, we generate data chunks from the data stream then provide it to the algorithm during training and evaluation. This additional step is mandatory since the computation process (classifier training) is slower than the speed of the data stream, which might lead to data loss (no buffer/cache memory on MCUs to compensate for the delay). To better understand the need for training models using *ML-MCU* on IoT edge devices, in the following, we present two use case scenarios.

**Self-learning HVACs for superior thermal comfort.** Currently, HVACs in smart buildings control their internal environment using a standard HVAC control strategy. In most cases, such a standard/one-size-fits-all strategy fails to provide a superior level of thermal comfort for people because every

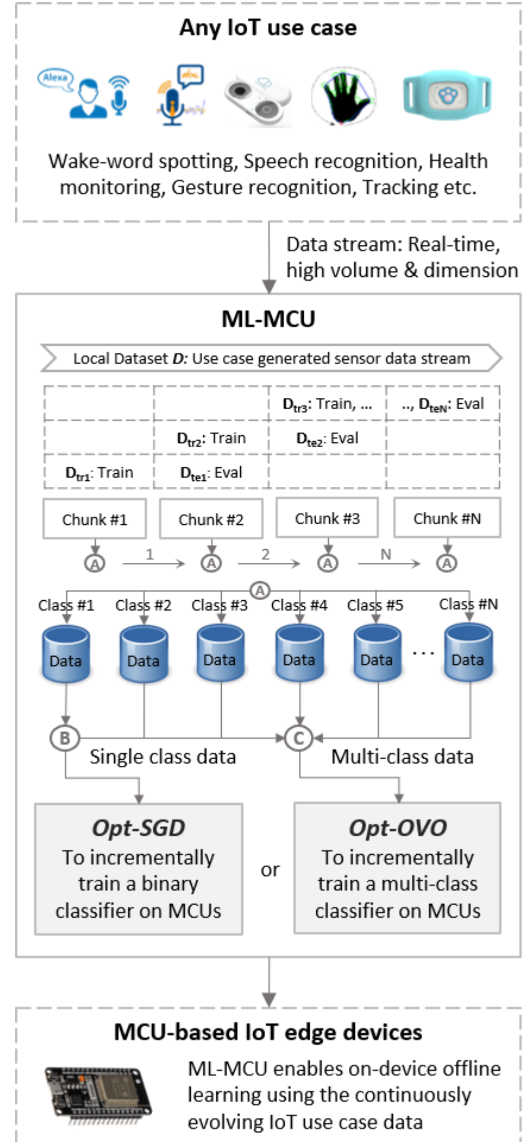


Fig. 1. ML-MCU framework to train ML classifiers on MCUs.

building/infrastructure has differences (e.g., location, size of a building, its thermal confinement, etc.). In this scenario, if the HVAC control edge devices are equipped with any of our training algorithms, they can learn the best strategy (offline) to perform tailored control of the HVAC system for any building types, eliminating the need to find and set distinct HVAC control strategies for each building, in order to provide the desired thermal comfort for people.

**Providing sensitive medical data for research.** The data required for most researches are sensitive in nature, as it revolves around a private individual. Currently, privacy regulations restrict sending sensitive yet valuable medical data from hospitals and imaging centers to biological research conducting pharmaceutical companies. When the resource-constrained medical devices like insulin-delivery devices, electric steam sterilizers, BP apparatus, etc., are equipped with any of *ML-MCU* algorithms, they will become capable of training offline using the sensitive medical data, even without depending on the hospital's local servers. After training, we can extract

[2]Refer to our Edge2Train work [2] for details on how to fuse ML-MCU framework algorithms with the device's IoT application/program for improving edge analytic results by training using the evolving real-world data.

the weights (learned information) of the trained models from multiple similar devices without exposing the data and send it to research firms (securely via https) without voiding the privacy regulations. This method of transmitting trained models (knowledge gained from distributed data) instead of the actual data fuels fine element analysis at various stages of drug development, from identifying target molecules to recruiting patients for clinical trials by providing the much-needed sensitive data. Also, enables manufacturers to know their device field performance, perform analysis for early warnings, etc.

### A. *Opt-SGD for Training Binary Classifiers on MCUs*

In this section, we provide our *Opt-SGD* algorithm to enable training binary classifiers directly on MCUs.

*1) Background and Setup:* Here, we first brief the core concept of the popular GD and SGD, then the setup. We view the optimization method to enable binary classifier training on edge devices as a loss minimization problem with the form,

$$\min_{x \in \mathbb{R}^d}, \text{ where } f(x) = \frac{1}{n}\sum_{i=1}^{n} f_i(x) \ , \tag{1}$$

Where, $d$ is the number of features, $n$ is the number of samples and $f_i(x)$ is the loss for sample $i$. Here, we seek to find a predictor $x \in \mathbb{R}^d$ minimizing the loss $f(x)$. In our ML model training scenarios, this $n$ is very large and we consider the new incoming data patterns as the local dataset,

$$D = \begin{cases} \mathrm{X} = \{x^{(0)}, x^{(1)}, \ldots, x^{(n)}\} \text{ where } x^{(n)} \in \mathbb{R}^n \\ \mathrm{Y} = \{y^{(0)}, y^{(1)}, \ldots, y^{(n)}\} \text{ where } y^{(n)} \in \{0,1\} \ , \end{cases} \tag{2}$$

To solve the problem in Eqn (1), given the model's parameters $x_k$ (e.g. weights of a classier), where $x_k \in \mathbb{R}^d$, the existing GD method sets:

$$x_{k+1} = x_k - \Delta f'(x_k) \ , \tag{3}$$

Where $\Delta$ is the stepsize and $f'(x_k)$ is the gradient of $f$ at $x_k$. To compute $f'(x_k)$ (the full gradient), the batch method computes the gradients of $n$ functions. It is prohibitive to calculate $n$ for every iteration since most ML datasets are large and the hardware we target to compute are MCUs, which are highly memory-constrained. Using this method on MCUs has other limitations such as; it leads to high computation time and sometimes crashing the training process. It causes memory overflows when data does not fit within the MCU's SRAM. Finally, after training, batch methods do not allow online model updations, i.e., using new data on-the-fly.

SGD method addresses these limitations by computing the parameter's gradient only using a single or a few data fields (uniformly picks random $i$'s) from the dataset. The SGD update is given by,

$$x_{k+1} = x_k - \Delta f_i'(x_k) \ , \tag{4}$$

Using this will reduce the computation for each iteration by $n$ factor since $E(f_i'(x_k)) = f'(x_k)$. To summarise, batch methods, over the same dataset, perform redundant computations by recomputing gradients before updating each parameter, which is very expensive on MCUs. To eliminate this redundancy SGD performs one update at a time, resulting in higher learning speeds.

**Algorithm 1** *Opt-SGD* to train binary classifiers on MCUs.

1: **Opt-SGD Parameters**: max: Maximum number of stochastic steps every epoch, $\Delta$: Step size, and $\varphi$: Convexity constant for $f$.
2: **for** s = 0, 1, 2, ..., **do**
3: $\quad g_s \leftarrow \frac{1}{n}\sum_{i=1}^{n} f_i'(x_s)$.
4: $\quad y_{s,0} \leftarrow x_s$.
5: $\quad$ Let $t_s \leftarrow t$ with probability $\frac{(1-\Delta)^{max-t}}{\sigma}$. where $t = 0$ to $max$.
6: $\quad$ **for** $t = 0$ to $t_s - 1$ **do**
7: $\quad\quad$ Pick $i \in 1, 2, \ldots, n$, uniformly at random.
8: $\quad\quad y_{s,t+1} \leftarrow y_{s,t} - \Delta(g_s + f_i'(y_{s,t}) - f_i'(x_s)$.
9: $\quad$ **end for**
10: $\quad x_{s+1} \leftarrow y_{s,t_s}$.
11: **end for**

*2) Opt-SGD Algorithm:* To improve thus briefed existing GD, we need to reduce the gradient computation costs. For existing SGD, we need to reduce the stochastic gradient's variance. In the rest of this section, we present our *Opt-SGD*, which combines benefits from both GD and SGD, thus inheriting the stability of GD while retaining the work-efficiency of SGD. We present our *Opt-SGD* in Algorithm 1 that users can use to enable high performing, resource-friendly binary classifiers training on small edge MCUs. During IoT application design, the users have to set the *OptSGD* parameters, stepsize $\Delta$ and a constant $max$ for limiting the number of stochastic gradients computed for every epoch. We recommend users to experiment training using our method with various $\Delta$s and select the value corresponding to the highest performance for their use-cases. In Algorithm 1, to make the best use of datasets, we use the concept of *epochs* (one pass over the dataset), where we shuffle the dataset in every epoch to prevent cycles. In our algorithm, the outer *for* loop is indexed by epoch counter $s$ and the inner loop is indexed by $t$. In every epoch $s$, we first compute $g_s$, which is the full gradient of $f$ at $x_s$. In every step, a random $t_s \in [1, max]$ number of steps are produced using a geometric law given below:

$$\sigma = \sum_{t=1}^{max} (1-\Delta)^{max-t}. \tag{5}$$

We can compute one stochastic gradient for every single *inner for* iteration, i.e., $f_i'(y_{s,t})$ at the cost of storing $f_i'(x_s)$, where $i = 1, 2, \ldots, n$. But as mentioned, since $n$ is very large for ML datasets, it is not possible to store on MCUs memory. Hence we load the data in batches and compute many stochastic gradient. In the next step of the algorithm, we subtract the stochastic gradient $f_i'(x_s)$ from $g_s$ and $f_i'(y_{s,t})$ is added to $g_s$ to ensure the expectation is w.r.t $i$ (the random variable),

$$E(g_s + f_i'(y_{s,t}) - f_i'(x_s)) = f'(y_{s,t}) \ . \tag{6}$$

Hence, our algorithm is a non-standard execution of the traditional SGD. To the best of our knowledge, *Opt-SGD* is

one of the few recent novel approaches that offer a high performance, resource-friendly method to enable model training on MCUs.

*3) Opt-SGD Results:* Here, we provide the result of our method, which is the contribution that enables high-performance, resource-friendly training on MCUs. If function $f$ from Eqn. 1 is convex, then *Opt-SGD* requires,

$$C_{work} = C_{fn}((n + c_n)\log(1/\mu)) \ , \tag{7}$$

Here, $C_{work}$ is the complexity of work, which is the measure of the total number of executions of stochastic gradient and also the executions of the total number of full gradient rounds. $n$ is the number of input samples, $c_n = D/\varphi$ is the condition number, where $\varphi$ is the convexity constant for $f$, and $D > 0$ is a constant. The Eqn. 7 can be explained as the work done to produce an $\mu$-approximate solution. We obtained this result by running our *Opt-SGD* Algorithm 1 with the stepsize $\Delta = C_{fn}(1/D)$, epoch $s = C_{fn}(log(1/\mu))$, the value of epoch is the number of full gradient executions, and $max = C_{fn}(c_n)$, the value of $max$ is roughly equal to the number of stochastic gradient executions in one epoch.

In the remainder of this section, we implement our *Opt-SGD* and run an experiment to compare its performance with GD and SGD. In a binary setting, using a zero mean, symmetric covariance Gaussian distribution, we generate data points ranging from $10^2$ to $10^6$, which are of the form shown in Eqn. 2. Here, for each $x^{(n)}$ that is of feature dimension 64, its class label $y^{(n)}$ is generated using a set of known weights. After data generation, we pretend to forget the used weights because our objective is to minimize the objective function, which also means fitting a binary classification model for the generated data.

We use the generated data on each algorithm, including ours. In Fig 2, we show the convergence behavior of each algorithm for input size ranging from $10^2$ to $10^6$. Here, the Y-axis represents the gap between optimality, which we measure and plot after each epoch (one full pass through the dataset). The convergence of GD and *Opt-SGD* is linear, whereas the SGD method shows a sub-linear convergence pattern. GD is the fastest when the data size is small. From analyzing Fig. 2. a to d, as the data size increases, the inefficiency of GD increases since to update the model weights for each iteration, the algorithm needs to pass through the entire dataset. As expected, we noticed the convergence of SGD to be very slow, whereas our *Opt-SGD* was capable to reach high precision despite its stochastic characteristic. In Fig. 2. d, for a very large train set, our method is much less efficient than SGD. But in practical scenarios, we do not train using such large data on MCUs, and anyways our *ML-MCU* framework initially splits the data stream from IoT use-case into data chunks that are small, for which our algorithm performs better than GD (higher precision) and faster than SGD.

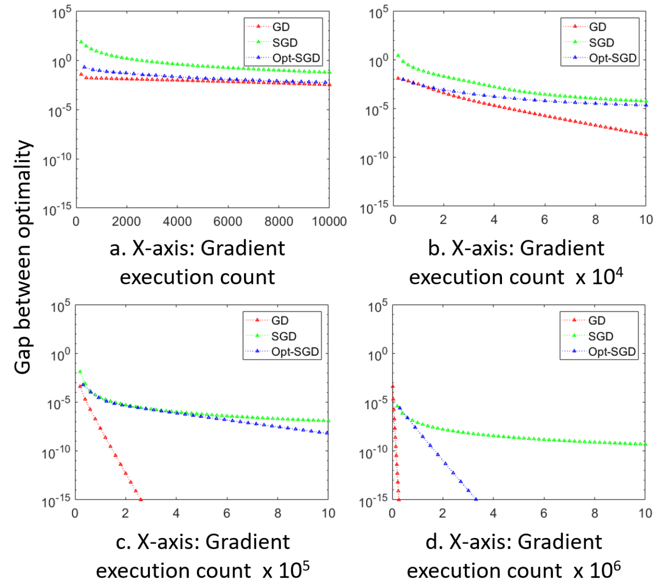


Fig. 2. Comparing the convergence behavior of Opt-SGD with GD and SGD.

### B. Opt-OVO for Training Multi-class Classifiers on MCUs

The majority of real-world IoT use cases such as health monitoring, gesture recognition, and equipment condition monitoring generate multi-class data. Training multi-class classifiers on MCUs, using any existing methods are currently not feasible. In this section, we provide *Opt-OVO*, an algorithm to enable training multi-class classifiers (to distinguish from multiple possible outcomes) directly on MCUs. *Opt-OVO* is superior to *Opt-SGD* since it applies for multi-class scenarios. Still, we provided *Opt-SGD* since its binary classifier training and inference time is much lesser than when using *Opt-OVO*.

*1) Background:* Here, we outline the state-of-the-art OVA and OVO methods before providing our *Opt-OVO*, which we propose as an optimized extension of the OVO method to enable multi-class classifier training on MCUs.

To provide an MCU-executable multi-classifier training method, we initially considered employing $k - 1$ classifiers where each classifier separates points in a particular class $C_k$ from points that do not belong to that class (solves a two-class problem). This approach is the existing *OVA* classifier where the algorithm finds $k-1$ classifiers, i.e., $f_1, f_2, \dots, f_{k-1}$. To explain in detail, the binary classifier $f_1$ classifies 1 from $\{2, 3, \dots, k\}$, the next $f_2$ classifies 2 from $\{1, 3, \dots, k\}$, finally the $f_{k-1}$ classifies $k-1$ from $\{1, 2, \dots, k-2\}$. The input values that are not classified to any classes belong to the $k$ class. There are multiple examples in the literature showing this heuristic method ambiguously classifies regions of input space. This means some input values get classified as they belong to multiple classes. Another drawback is, it creates one model for each class, so $k - 1$ models have to be stored in MCU's limited memory, restraining training using large datasets with multiple classes.

To address these limitations, the existing OVO classifier, introduces $k(k-1)/2$ binary classifiers (one for every possible pair of classes), where it finds $k(k-1)/2$ classifiers, i.e., $f_{(1,2)}, f_{(1,3)}, \dots, f_{(k-1,k)}$. To explain in detail, the binary classifier $f_{(1,2)}$ classifies 1 from 2, the $f_{(1,3)}$ classifies 1 from 3, finally the $f_{(k-1,k)}$ classifies $k-1$ from $k$. Here, the classification result for each multi-class input is based on the majority vote amongst the employed classifiers, or in other words, the final result is the class with the highest votes.

*2) Setup:* We use a set of base classifiers, $B$ to produce classification results for a multi-class input $x^{(n)}$. We store outputs of the entire base classifiers in $R_B$. In other words, each classifier $b_i \in B$ is a base learner that produces a result $\in$ {-1, +1}. Therefore, $R_B$ contains outcomes for all the $k(k-1)/2$ binary classifiers over the entire training chunk $D_{tr}$ shown in Fig. 1. We assign $l$ as the class indicator (label) for our problem with $k(k-1)/2$ classes. To understand our setup better, $R_B$ contains the required information to find out which class a given multi-class input $x^{(n)}$ belongs to and also used to compute $P(l_i \mid R_B)$ (Probablity). For example, $x^{(n)}$ and a set $B$ with three base learners, it's $R_B$ is of the form $R_B(x^{(n)}) = \langle -1, +1, -1 \rangle$. Using Bayes theorem,

$$\begin{aligned} P(y^{(n)} = l_i \mid R_B) &= \frac{P(R_B \mid y^{(n)} = l_i)P(y^{(n)} = l_i)}{P(R_B)} \\ &\propto P(R_B \mid y^{(n)} = l_i)P(y^{(n)} = l_i) \ , \end{aligned} \tag{8}$$

Here, since $P(R_B)$ is common for all classes, it can be suppressed. Hence considering this independence between all the outcomes of base learners in $B$ for a sample input $x^{(n)}$, Eqn. 8 becomes

$$P(y^{(n)} = l_i \mid R_B) \propto \prod_{b_i \in B} P(R_B^{b_i} \mid y^{(n)} = l_i)P(y^{(n)} = l_i) \ , \tag{9}$$

Here, $R_B^{b_i}$ is the classification result of a base learner $b_i \in B$. As shown, using this independence concept simplifies the model, but it might not suite for all cases. Hence we relax it and do not use Eqn. 9 in our algorithm design.

When the results of two base classifiers overlap within a tight area for the same input data, we group such correlated classifiers to reduce the number of base classifiers. Our *Opt-OVO* algorithm finds groups of correlated base classifiers, creates a Probability Table (PT) for each group. Hence, the multi-class classification problem is modeled to be conditioned to groups of correlated base classifiers $Corr_{class}$. Hence the model in Eqn. 8 becomes,

$$\begin{aligned} P(y^{(n)} = l_i \mid R_B, Corr_{class}) &\propto P(y^{(n)} = l_i) \\ & P(R_B, Corr_{class} \mid y^{(n)} = l_i) \ . \end{aligned} \tag{10}$$

We assume independence only among the groups of highly correlated base learners $b_{corr} \subset Corr_{class}$. Therefore, to find the class of an input $x^{(n)}$ we use,

$$\begin{aligned} \text{class}(x^{(n)}) = \arg\max_j & \prod_{b_{corr} \subset Corr_{class}} P(y^{(n)} = l_j) \\ & P(R_B^{b_{corr}}, b_{corr} \mid y^{(n)} = l_j) \ , \end{aligned} \tag{11}$$

Here $R_B^{b_{corr}}$ is the outcome of all base learners that belong to the group of highly correlated base classifiers $b_{corr} \subset Corr_{class}$, for training data $D_{tr}$. To find the groups of correlated base classifiers $Corr_{class}$, we create a correlation matrix $C_m$ to measures the level of correlation between two base classifiers when classifying for a train chunk $D_{tr}$.

*3) Opt-OVO Algorithm:* Here, we present our *Opt-OVO*, which we propose as an optimized extension of the state-of-the-art OVO method. From our analysis, we discovered that when using existing OVO, the $k(k-1)/2$ base learners/classifiers, for a few datasets, contain classifiers that lack significant contributions to the overall multi-class classification result. This occurs when a classifier is already within a big interdependent group. Hence, we provide a method, which is a part of our *Opt-OVO* that identifies then removes the less important base classifiers, thus improving the overall resource-friendliness when executing on MCUs.

**Algorithm 2** *Opt-OVO* with *Opt-SGD* based base learners to incrementally train multi-class classifiers on MCUs.

1: **Inputs**: Train ($D_{tr}$) & test ($D_{te}$) chunks of local dataset $D$ (Eqn. 2). *Opt-SGD* training method from Algorithm 1.
2: **Output**: Incrementally trained multi-class classifier.
3: **for** each $k(k-1)/2$ base classifiers $b_i \in B$ **do**
4: $\quad Model_i \leftarrow$ Train ($D_{tr}$, $b_i$, *Opt-SGD*). Train $b_i$ with $D_{tr}$ using *Opt-SGD* method.
5: $\quad R_i \leftarrow$ Evaluate ($D_{te}$, $Model_i$, *Opt-SGD*). Evaluate trained $Model_i$ with $D_{te}$ using *Opt-SGD*.
6: **end for**
7: $R_B \leftarrow \bigcup R_i$. Store all outcomes $R_i$ of $k(k-1)/2$ binary base classifier $b_i$ in $R_B$.
8: **Create** a correlation matrix $C_m$ for $R_B$.
9: **Find** groups of highly correlated base classifiers $Corr_{class}$ from $C_m$.
10: Using $R_B$ **create** a PT for each highly correlated classifiers groups $b_{corr} \subset Corr_{class}$.
11: **Classifier:** Classify for any new $x^{(n)}$ by using thus obtained set of base classifiers $B$ and $Corr_{class}$ in Eqn. 11.

We present our *Opt-OVO* in Algorithm 2. Here, in Line 3-6, all the $k(k-1)/2$ base classifiers $b_i$ belonging to $B$ are trained with the local data $D_{tr}$ (see Fig. 1) using our *Opt-SGD* Algorithm 1. In the function in Line 4, we provide the flexibility for users to replace our *Opt-SGD* method with the base learner of their choice, i.e., SVM, LDA, etc. But we use our *Opt-SGD* for its resource-friendly characteristics. In Line 5, we evaluate all the thus trained base classifiers. Here, each base classifiers $b_i$ produce a binary output $\in \{-1, +1\}$ for each input vector $x^{(n)}$. In Line 7, for all data in $D_{te}$, we store outcomes of the $k(k-1)/2$ base learners $R_i$ in $R_B$. Next, in Line 8, we create a correlation matrix $C_m$ using the output of base classifiers stored in $R_B$. In Line 9, from $C_m$, we find $Corr_{class}$, which is the group of highly correlated base classifiers. In Line 10, from the groups of this found correlated base learners, we create a PT of each group to know the joint probability of the outcome $R_B$. These PTs provide the joint probabilities of the outcomes $R_B$ and the groups of correlated classifiers $b_{corr} \subset Corr_{class}$ when evaluating using new/unseen data. In the final Line 11, we classify for any new multi-class input $x^{(n)}$ by using the algorithm produced set of base classifiers $B$ and $Corr_{class}$ in Eqn 11.

Next, we explain our method to find the groups of highly correlated base classifiers $Corr_{class}$ from correlation matrix $C_m$. For a multi-class training chunk $D_{tr}$, we measure the level of correlation between two base classifiers by considering their binary classification result $\in \{-1, +1\}, R_i, R_j$. Where $R_i$ & $R_j \in R_B$ are outcomes of base classifiers $b_i$ & $b_j$ and its

correlation matrix $C_m$ is given as,

$$C_{m(i,j)} = \frac{1}{N} \left| \sum_{\forall x^{(n)} \in D_{tr}} R_i(x^{(n)}) R_j(x^{(n)}) \right| , \tag{12}$$

Here, if both the base classifiers produce the same output for all the data points in $D_{tr}$, then the level of correlation between them is one, the highest, so, $R_i = R_j$. In cases when their outputs always don't match $R_i \neq R_j$, again their correlation is one. Whereas if the base classifiers have half outputs matching and rest not equal, then the correlation is zero. We use this method in our *Opt-OVO* Algorithm 2 to group similar output producing base binary classifiers.

In the remainder of this section, we briefly explain how researchers and developers can use our *ML-MCU* algorithms for enabling their IoT devices/products to perform *on-the-fly* offline learning. During the IoT device programming phase using Arduino IDE, Atmel Studio, Keil MDK, etc., our algorithm implementation (code) needs to be fused with the use-case IoT application. Then, when the labeled data fields (generated by following the method from Edge2Train [2]) that correspond to the low accuracy inference performed are passed to our algorithm function, it trains and updates the current classifier version running on the IoT device with a superior performance version. Our *ML-MCU* implementation is also applicable to other self-learning settings where it can locally train a model from scratch without needing cloud-based ML training services or proprietary datasets. Here, instead of passing only the data that correspond to the low accuracy inference, the complete live data stream should be cleaned, labeled, and passed to our algorithm function. An example application of this self-learning setting can be *A coffee machine learning a person's taste*. Here, *ML-MCU* can make a coffee machine learn the data patterns of temperature, time, and material proportion when a person makes his best coffees. When brewing new coffees, the machine can use this knowledge to detect and alert when there is a considerable deviation from the best coffee patterns. Thus, the machine can ensure the person gets the coffee of his taste all the time.

## III. Evaluation

Here, we evaluate both the *Opt-SGD* and *Opt-OVO* algorithms of our *ML-MCU* framework. For each algorithm, we start the evaluation by first explaining the evaluation setup, then for the selected datasets, using our algorithms, we train models on the selected popular MCU boards, whose specification is given in Table I. Next, we tabulate the obtained results (training & unit inference time, model accuracy, and memory requirements) and analyze them to summarize the benefits of our methods. Finally, we compare our methods with the existing results of other methods.

### A. Datasets and Evaluation Procedure for Opt-SGD

For evaluation, we selected four datasets using which the *Opt-SGD* algorithm trains binary classifiers on the selected MCUs. The first dataset we chose is the Iris Flowers dataset[3], which has the least feature dimension of 4 when comparing with the other selected datasets. Here, we extract 50 positive and 100 negative samples of Iris Setosa. Then using *Opt-SGD*, we trained a binary classifier that distinguishes Iris Setosa from other flowers based on the input features. The second round of evaluations was performed using the Heart Disease dataset [4]. Here, after training, based on the input features, the *Opt-SGD* trained classifier should be able to identify the presence of heart disease in the patient. The third round of evaluation was performed using the Breast Cancer dataset [5]. Here we train a binary classifier that can find the class names (malignant or benign) based on the input features. The fourth round of evaluation was performed using the MNIST Handwritten Digits dataset [6], which has the highest feature dimension of 64. Here, we extracted data fields for digit 6, with positive and negative samples. Then using *Opt-SGD* we trained a binary classifier on MCUs, that distinguishes digit 6 from other digits, based on the input features.

TABLE I
Specifications of MCUs chosen to evaluate ML-MCU.

| Board or MCU # | MCU & Board Name | Specification | | | | |
|---|---|---|---|---|---|---|
| | | Bits | SRAM | Flash | Clock (MHz) | FP |
| 1 | nRF52840 Adafruit Feather | 32 | 256kB | 1MB | 64 | ✓ |
| 2 | STM32f103c8 Blue Pill | 32 | 20kB | 128kB | 72 | ✗ |
| 3 | Generic ESP32 | 32 | 520kB | 4MB | 240 | ✓ |
| 4 | ATSAMD21G18 Adafruit METRO | 32 | 32kB | 256kB | 48 | ✗ |

### B. On-board Binary Classifier Training & Inference using Opt-SGD

We uploaded the *Opt-SGD* algorithm's C++ implementation on all boards from Table I. We then power on each board, connect them to a PC via the serial port to feed the training data in chunks, receive training time and classification accuracy from MCUs. For all of the selected datasets, the first 70% of data was used for training, the remaining 30% data for evaluation. When we instruct the board to train, *Opt-SGD* iteratively loads the data chunks and trains the classifier using its method from section II-A. Next, we load the test set, infer using the trained models to evaluate the trained classifiers. We tabulated thus obtained results in Table II, using which we analyze the results in the remainder of this section.

*1) Training and Inference Time:* From Table II, it is apparent that, even for high dimensional Digits dataset, our method achieves real-time unit inference, within a ms, even on the slowest MCU 4. Although CPUs are faster, they cannot be used as IoT edge devices because; they are ≈200x times more expensive than MCUs, they have ≈10x bigger form factor, and consume ≈7x time more energy to train the same models. Since billions of edge devices are MCU-based, it is feasible to train even at lesser speeds. Such offline training using *Opt-SGD* reduces the hardware cost of edge devices since they

[3] https://archive.ics.uci.edu/ml/datasets/iris

[4] https://archive.ics.uci.edu/ml/datasets/heart+Disease

[5] https://www.kaggle.com/uciml/breast-cancer-wisconsin-data

[6] http://yann.lecun.com/exdb/mnist/

TABLE II
ON-BOARD BINARY CLASSIFIER TRAINING USING OPT-SGD: FLASH, SRAM, AND TIME CONSUMED TO TRAIN & INFER ON VARIOUS MCUS.

| MCU # | Dataset Dimension & Size (No. of row) | Train Time (ms) | Accuracy (%) | Inference Time (ms) | Flash & SRAM Req (kB) |
|---|---|---|---|---|---|
| 1 | Iris Flowers 4, 150 | 3 | 84.4 | 0.022 | 42.97, 8.79 |
| | Heart Disease 13, 212 | 6 | 80.1 | 0.015 | 53.63, 18.76 |
| | Breast Cancer Dataset 30, 567 | 273 | 63.0 | 0.005 | 111.864, 77.32 |
| | Handwritten Digits 64, 356 | 341 | 89.7 | 0.018 | 134.90, 100.39 |
| 2 | Iris Flowers 4, 150 | 29 | 75.5 | 0.022 | 30.84, 6.24 |
| | Heart Disease 13, 212 | 77 | 82.0 | 0.03 | 40.832, 16.24 |
| | Breast Cancer Dataset 30, 567 | 3837 | 78.0 | 0.1 | +33.9, +54.33 |
| | Handwritten Digits 64, 356 | 3759 | 95.0 | 0.1 | +56.95, +77.4 |
| 3 | Iris Flowers 4, 150 | 3 | 77.7 | 0.022 | 131.72, 17.38 |
| | Heart Disease 13, 212 | 3 | 83.0 | 0.015 | 228.84, 27.38 |
| | Breast Cancer Dataset 30, 567 | 112 | 62.0 | 0.005 | 287.43, 241.73 |
| | Handwritten Digits 64, 356 | 161 | 70.0 | 0.046 | 310.46, 108.98 |
| 4 | Iris Flowers 4, 150 | 90 | 84.4 | 0.044 | 23.12, 7.21 |
| | Heart Disease 13, 212 | 260 | 83.0 | 0.093 | 35.73, 17.26 |
| | Breast Cancer Dataset 30, 567 | 13806 | 78.0 | 0.33 | +32.6, +40.58 |
| | Handwritten Digits 64, 356 | 14130 | 90.6 | 0.36 | +56.4, +63.62 |

do not need a wireless module (4G or WiFi) to receive the updated models from the cloud. Also when the data for which the model has to be updated is small, then it does not require data center GPUs for training. It can rather be trained on the edge, using our framework, without compromising accuracy.

*2) Accuracy:* From Table II, the highest onboard classification accuracy is 84.4% for the Iris, 83.0 % for Heart Disease, 78.0 % for Breast Cancer, and 95.0% for Digits dataset. Although the training time on MCUs is higher than CPUs, our *Opt-SGD* trained models produce classification accuracies close to those of Python scikit-learn trained models. Unlike other batch methods, since *Opt-SGD* can incrementally load data and train models, we expect to achieve accuracies close to models trained in high-resource setups, even when experimenting with other larger datasets.

*3) Flash & SRAM Requirements:* The run-time variables generated during training need to be stored within the limited SRAM of MCUs. From Table I, the popular open-source boards we chose have only 20 kB to a max of 520 kB of SRAM. We provide the Flash and SRAM requirements calculated by the compiler for target MCUs for all boards in Table II. Here, for MCU3 the largest Digits dataset, including the *Opt-SGD* algorithm consumes only 7.7 % Flash and 20.9 % SRAM. Whereas for MCUs 2 & 5, both the Flash and SRAM overflowed (exceed MCUs capacity) for both the Breast Cancer and Digits datasets. Hence, in such scenarios with large volume and high dimensional data, the upper bound imposed by MCUs memory-constraints restricts training models on such small MCUs. Since we designed *Opt-SGD* to be capable of incrementally training a model, even on small MCUs, with only KBs of memory, we can incrementally load *n-samples* of high-dimensional data, which might range from a few MB to hundreds of MBs, then perform the required model training. Hence, even on the lowest-spec MCU4, we were able to load both the Breast Cancer and Digits datasets incrementally and train in 13.8 ms and 14.1 ms respectively.

From the analysis of *Opt-SGD*'s results presented in this section, it is apparent that developers can leverage *Opt-SGD* to train models offline using real-time data from any of their use cases on such small MCU boards. We also estimate that using *Opt-SGD*, on-board binary classifier training can be performed on thousands of open-source MCU boards supported by Arduino IDE, which have limited Flash, SRAM, and no floating-point support.

### C. Comparing Opt-SGD with Other Methods

The work complexity of our *Opt-SGD*, which is the measured number of stochastic gradient executions required to produce an $\mu$-solution was presented in Eqn 7. Our Eqn is similar to EMGD results [6], where the performance (work complexity) of EMGD is given as $C_{fn}((n + c_n^2)\log(1/\mu))$. This Eqn clearly shows that their method achieves a quadratic dependence on the condition number $c_n$ instead of linear dependence, and also their results hold with high probability. Similarly, the complexity of the next related stochastic coordinate descent (RCDC) method was also investigated in a high probability setup [7]. The SDCA [8] method has work complexity similar to our Eqn 7. But the condition number $c_n$ in their method is complex than ours since it requires complete knowledge of the convexity constant $\varphi$ of $f$ to run their algorithm. We found SAG [9] as the most related method to compare with since it achieves linear convergence only using the stochastic gradient executions (similar to our method). During the close comparison, our method was faster since SAG updates the test points after each execution of a stochastic gradient. Whereas our *Opt-SGD* does not always update during the evaluation of the full gradient. Hence, our method needs lesser time to perform the same amount of passes through data chunks than SAG. Another downside of SAG is, it consumes additional space in MCUs memory to store $n$ gradients, which is necessary for their method.

In Section II-A3, we compared the convergence behavior of *Opt-SGD* with GD and SGD. From dataset to dataset, we expect the performance gap between *Opt-SGD* and related methods to vary. So, to present a fine-grained comparison, we need to conduct computational experiments. Hence, in future work, we shall run experiments, where we make the GD, SGD, SAG, SDCA, RCDC algorithms and *Opt-SGD* to solve the same least-squares problem by iterative refinement. Using the obtained results, we shall perform a fine-grained comparison and also compare the practical performance with their theoretical bound on convergence in expectations.

### D. Datasets and Evaluation Procedure for Opt-OVO

For evaluation, we selected two multi-class datasets using which the *Opt-OVO* algorithm trains multi-class classifiers on the MCUs from Table I. For the first evaluation round, we use the same 64 features Handwritten Digits dataset, but in a different setup. Here, we built 3 train sets of various class counts and sizes. For the first train set, we extract data fields corresponding to the handwritten digits 0 to 2 to build a 3

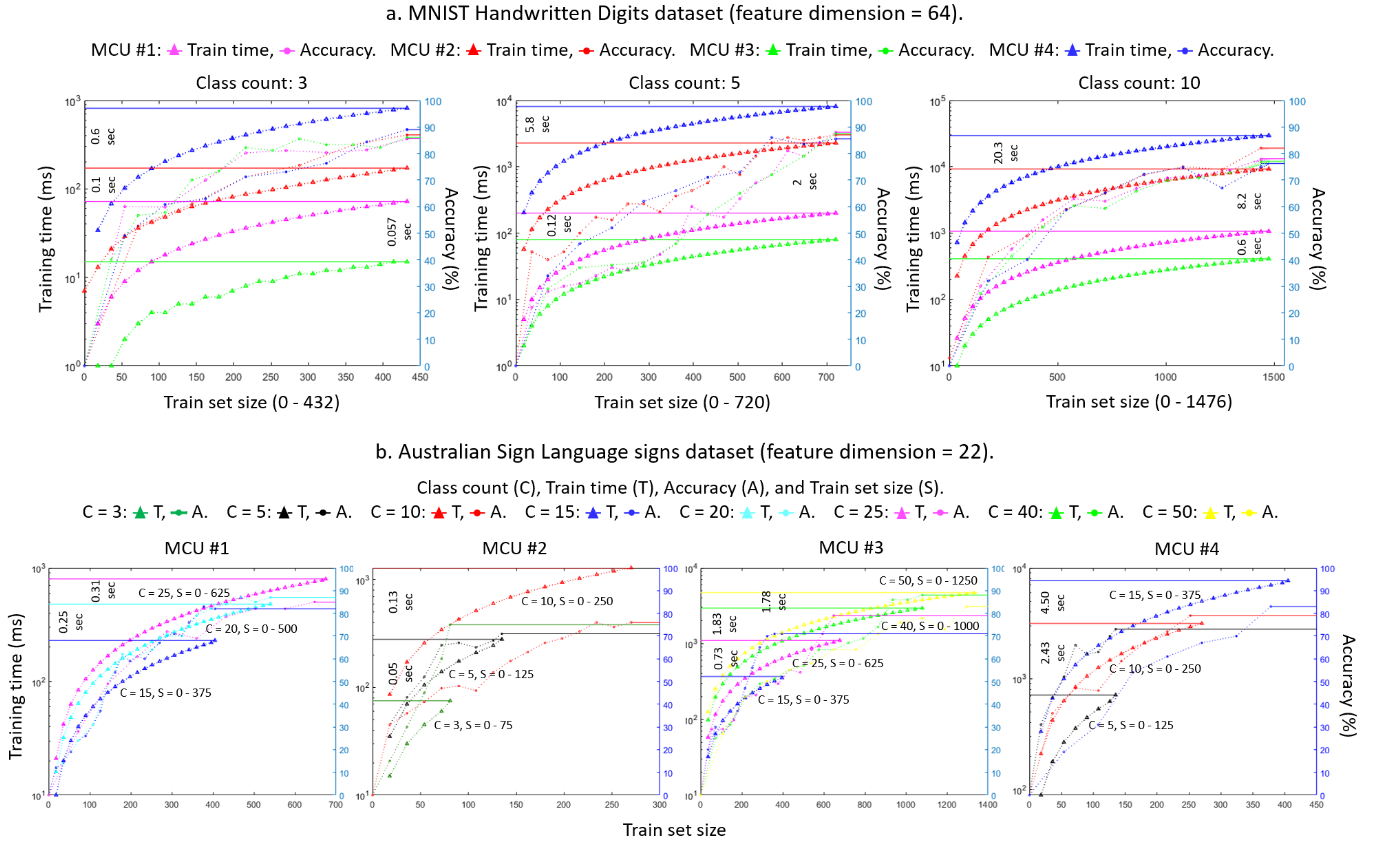


Fig. 3. Training multi-class classifiers on MCUs using *Opt-OVO*: Comparing training set size vs training time & accuracy for a. MNIST Digits dataset; train set size from 0 to 1476 and class count from 3 to 10. b. Australian Sign Language signs dataset; train set size from 0 to 1250 and class count from 3 to 50.

class train set of size 432. The second train set is of class count 5 (digits 0 to 4) and size 720. The last train set of size 1467 contains 10 classes (digits 0 to 9). In all the 3 train sets, each class is of the size 144.

The second round of evaluation was performed using the 22 features Australian Sign Language signs dataset [7]. Here, we built 8 train sets of different class counts and sizes. For the first train set, we extract data fields corresponding to the *alive, all* and *answer* Auslan signs. Hence, the first set is of class count 3 and size 75. The last set is of class count 50 and size 1250 since it contains data of 50 Auslan signs varying from *alive* to *more*. The in-between train sets contain class counts ranging from 3 to 50, with their corresponding train set size ranging from 0 to 1250. In all the 8 train sets, each class is of the size 25.

For both the datasets, as explained, we purposefully built train sets of different class counts and sizes in order to plot and investigate (i) the train set size vs training time & accuracy. (ii) the class size vs onboard inference time. We built a test set for each train set that contains unseen data fields from the corresponding classes in the respective train set and data fields from the remaining classes. We use these test sets to compute the accuracy of classifiers trained on MCUs by *Opt-OVO*. For brevity purposes, we do not present the classifier performance in confusion matrices. If users require performance visualization in a table layout, they can enable the confusion matrix function we provide in the repository.

[7]https://archive.ics.uci.edu/ml/datasets/Australian+Sign+Language+signs+(High+Quality)

### *E. On-board Multi-class Classifier Training & Inference using Opt-OVO*

We follow the same procedure explained in section III-B, starting from uploading the algorithm, until loading the test set and evaluating the trained classifiers. Here, the difference is, we replace the *Opt-SGD* with the *Opt-OVO* algorithm, then use the multi-class data instead of binary. We illustrate the obtained training results in the form of graphs in Fig. 3, which we use to analyze how the training time and accuracy vary w.r.t to the train set size. We do not analyze the Flash and SRAM requirements like in section III-B3 because the datasets we use here exceeded the available onboard memory of all the selected MCUs. But due to the incremental model training design of our *Opt-OVO*, it was able to incrementally load both the high-dimensional, large volume, multi-class datasets via Serial port, then perform the required training and evaluation.

*1) Training Set Size vs Training Time & Accuracy:* The Fig. 3. a & b shows that even on the slowest MCU 4, our *Opt-OVO* was able to train using a 10 class, 1476 size, 64 dimension Digits dataset in 29.6 sec and could train in 7.6 sec using the 15 class, 375 size, 22 features Australian Sign dataset. The same figures also show that the fastest MCU3 trained in 0.4 sec for Digits and in 4.7 sec using the 50 class, 1250 size Sign dataset. In these figures, the gap in the Y-axis (base-10 log scale) is the difference in the training time between the

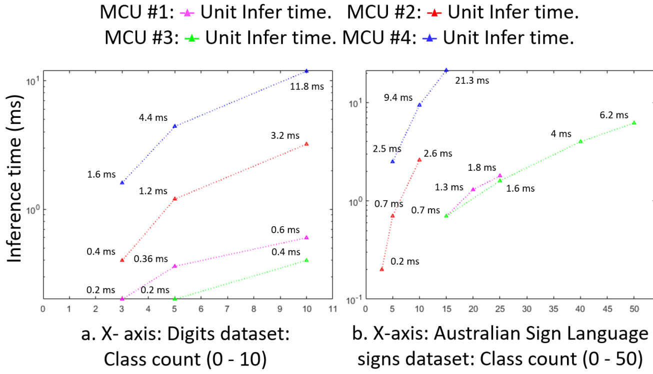


Fig. 4. Inferring for multi-class data on MCUs: Class size (3 to 50) vs inference time of the onboard *Opt-OVO* trained classifiers.

selected MCUs, for fixed class counts. For a clearer view, we marked this gap in seconds. In Fig. 3. b, at the individual MCU level, we show how the training time varies when the class count and train set size increases. In Fig 3, the right side Y-axis is the MCU trained multi-class classifier accuracy. We are not presenting the explicit performance comparing of the classifiers trained using our *Opt-OVO*, with the classifiers trained on high resource setups using *Python scikit-learn* since we achieve similar accuracies when experimenting using the same setup and datasets. Also, we are not comparing the training time on MCUs with CPUs and edge GPUs since they have $\approx 10^6$x times higher hardware resources. Using *Opt-OVO*, users can increase the class count beyond 50 and train without stability issues when they use the emerging better resource MCUs with inbuilt FPU, KPU, FFT hardware capabilities.

*2) Class Size vs Inference Time:* To analyze the impact of increasing class count on inference time, in Fig. 4. a, we feed the *Opt-OVO* trained models a multi-class data (size one) with class count ranging from 0 to 10 and from 0 to 50 in Fig. 4. b. For a clearer view, we label the plotted points with its corresponding values (unit inference time) in seconds. For statistical validation, the plotted inference time corresponds to the average of 5 runs. It is apparent that, even for high dimensional Digits dataset, our method achieves real-time unit inference, 11.8 ms, even on the slowest MCU4. The fastest MCU3 was able to infer for a 50 class input in 6.2 ms. In Fig. 4. b, when the class count increased from 15 to 25, the inference time increased by 0.9 ms, from 25 to 40 by 2.4ms and 40 to 50 by 2.2 ms, showing an almost linear relationship. Overall, it is apparent that our *Opt-OVO* trained classifiers perform onboard unit inference for multi-class data in super real-time, within a second, across various MCUs.

### F. Comparing Opt-OVO with Other Methods

Here, we first brief a method to find the computational requirement of the existing OVA and OVO methods. We then use it on the related papers to investigate their optimization efficiency (computational simplification). In this method, we consider $B$ as the number of base learners that decompose one multi-class problem into multiple binary problems, $N_B$ as the size of $B$, the count of classes in the given multi-class problem as $C$ and $T_c$ is its respective time complexity.

For OVA, $N_B = C = T_c(C)$. Here, the time complexity is $T_c(C)$ since each base learner requires positive examples of the respective class and negative examples of the rest of the classes. During testing, OVA needs $T_c(C)$ binary classifications for each inputs tested. In the case of OVO, $N_B = \binom{C}{2} = T_c(C^2)$. Here, the time complexity is $T_c(C^2)$ since each base learner (binary classifier), each time requires positive and negative samples of the two classes that are considered for training. During testing, OVO needs $T_c(C^2)$ binary classifications for each inputs tested. OVO becomes computationally challenging for high $B$ values.

In [10], authors have combined OVA and OVO to improve the overall classification. But their approach uses $N_B = \binom{C}{2} + C = T_c\left(C^2\right)$ base classifiers during training. The method in [11] is a combination of classifiers, where the output of each base classifier is a probability of the pattern that belongs to the given class. Here, $(C(C+1))/2 = T_c(C^2)$ base classisiers are required. In [12], a dynamic programming approach was used to design a class removal sequence. Their method only uses $N_B = C - 1$ base classifiers during testing, but the class removal policy is very expensive to run on MCUs.

Our *Opt-OVO* method produces a higher level of optimization than the above methods since it can intelligently identify and remove base classifiers that lack significant contributions to the overall multi-class classification result, enabling training up to 50 class data on highly resource-constrained MCUs. Additionally, our method has benefits such as; it does not depend on the number of selected base classifiers, and it also provides the flexibility for users to use any base learner method of their choice, i.e., SVM, LDA, etc.

## IV. Related Work

Since our *ML-MCU* enables incremental binary and multi-class classifier training on MCU-based tiny edge devices, for comprehensiveness, our state-of-the-art review consists of the two following subsections:

### A. Machine Learning on Microcontrollers

Here, we classify the existing literature into two broad categories: training models on MCUs and resource-efficient model inference on MCUs.

Training ML models on MCUs is an emerging area of research [13, 14]. The existing frameworks like Tensorflow Micro, Keras, Edge-ML, Open-NN, etc. do not yet provide methods to enable training models on MCUs. Currently, to achieve a resource-efficient training, authors have optimized existing algorithms to run on various resource-constrained setups. In [15], a Gaussian Mixture Model was executed on an embedded board aiming to re-train an ML algorithm at the edge level. Articles [16, 17] present optimized methods to enable training models on smartphones. Multiple Federated Learning algorithms [18, 19] enable fine-tuning global models offline, at the edge level using local datasets. SEFR, a low-power classifier [20] is the most recent work to enable a binary classifier training and inference on MCUs. However, thus

outlined and other impactful algorithms [21, 22] are tailored for specific applications and do not enable MCU-based IoT edge devices to self learn/train from a wide range of IoT use-case data. Our *ML-MCU's* algorithms are superior to state-of-the-art methods since they can incrementally train both binary and multi-class classifiers iteratively, enabling edge MCUs to continuously improve themselves for better analytics results.

In the category of algorithms for efficient model inference on MCUs, there is a set of articles proposing compression techniques to reduce the size of the model's weights using quantization and pruning. CONDENSA [23], a system for users to compose simple operators to build complex model compression strategies. Two new compression methods jointly leverage weight quantization and the distillation of larger networks was proposed in [24]. Authors in [22] have implemented a tree-based algorithm for efficient prediction in milliseconds even on slow MCUs. Similarly, ProtoNN [25], k-Nearest Neighbor inspired algorithm with several orders lower storage and prediction complexity addresses the problem of real-time and accurate prediction on resource-scarce devices. In [23, 24, 26, 27] and other articles proposing compression [28, 29] and optimization [30, 31] methods, the models are trained in high resource setups, then a multi-stage MCU-aware optimisation (tailored) is performed before deployment.

Similar to the above works, even our framework algorithms can infer in super real-time on MCUs. For example, the MCU3 used *Opt-OVO* to perform onboard unit inference for a 50 class data sample in super real-time of 6.2 ms (in Section III-E). We do not attempt to outperform the state-of-the-art inference methods since they are tailored to be application-specific, whereas our *ML-MCU* algorithms are capable to train and infer using real-time data from any IoT use-cases.

### B. Optimizing the SGD and OVO Methods

SGD is an iterative method for optimizing an objective function with suitable smoothness properties. The Random Coordinate Descent for Composite functions (RCDC) method [7] can be directly applied to the equation of a loss minimization problems or its dual version. Applying an RCDC to the dual formulation is known as Stochastic Dual Coordinate Ascent (SDCA) [8]. The analysis of SDCA in their paper shows that it is comparable or better than SGD. Article [32] study parallel stochastic coordinate descent method where they show block coordinate descent methods when accelerated by parallelization can benefit a type of minimization problem used in their paper. Stochastic Average Gradient (SAG) [9] is the first method relevant to SGD, that is different from the above coordinate descent approaches while exhibiting linear convergence. The Epoch Mixed Gradient Descent (EMGD) [6] is similar to the SAG, SDCA, and our method, but it achieves a quadratic dependence on the condition number instead of linear dependence. Articles [33, 34] combine GD and SGD, vary sample size to achieve reduce variance. However, the benefits obtained as a result of their realization is lesser than the above recent methods. Classical paper [35] on stochastic approximation methods is partially related to our work.

From the literature that aims to optimize the popular OVA and OVO methods, we select and brief the papers that are related to our approach. In [10], authors have combined OVA and OVO to improve the overall classification. But their approach uses $+C$ extra classifiers during training. The method in [11], a combination of classifiers use $C^2$ extra base classifiers. In [36], a class embeddings method to choose the best base classifiers was presented. In [12], a dynamic programming approach was used to design a class removal sequence. Although their method uses fewer $(C-1)$ base classifiers during testing, their class removal policy is very expensive for MCUs. Since the existing methods improve the OVA and OVO at the cost of adding extra base classifiers, it is not feasible to implement them for training models on MCUs.

## V. Conclusion: Discussion and Future Work

We presented *ML-MCU*, a framework with resource-friendly binary and multi-class classifier training algorithms to enable billions of MCU-based IoT edge devices to self learn/train (offline) *on-the-fly*, using live data from a wide range of IoT use-cases. Thus, the devices equipped with any of our algorithms can self-learn to perform analytics for any target IoT use cases without requiring a historical dataset. As explained with evaluations, the few most exciting benefits are; the incremental training characteristics of our algorithms enabled loading the full *n-samples* of large volume high dimensional data without memory overflow issues, also it incrementally updated the trained models without losing any information it learned from the old data streams. Next, models trained on MCUs using *ML-MCU* produced accuracies close to ML framework trained models on high resource setups. Finally, we achieved a significant level of optimization compared to existing methods, thus facilitated training using 50 class data on MCUs and then performed unit inference in super real-time.

Utilizing the open-sourced implementation of *ML-MCU* algorithms, researchers and developers can start to practice split-learning, federated learning, centralized learning, distributed ensemble learning by involving even the most resource-constrained MCU-based IoT devices in complex learning tasks. For example, in a sensitive medical data collection use-case, without disturbing the routine of resource-constrained medical device (such as electric steam sterilizers, BP apparatus, etc.), our framework algorithms can locally learn (data does not leave the device) using the sensitive patient data, then share only the learned parameters to the hospital/research center servers. Since even the light ML frameworks versions like TensorFLow Lite cannot be used in such use-cases involving resource-constrained MCU-based devices, we believe *ML-MCU* to provide the basis for a broad-spectrum of decentralized and collaborative learning applications.

We see the lack of real-world experimental evaluation as the major limitation. Also, the framework behavior and its on-device self-learning performance need to be investigated by deploying the implementation (code) of the framework algorithms on real-world devices. Hence in future work, we plan to; (i) Make a generic IoT device autonomously learn to perform condition monitoring of an industrial paint compressor. Here, the task of *ML-MCU* is to make the device incrementally learn by monitoring the contextual sensor data corresponding to

regular vibration patterns from the pump's crosshead, cylinder and frame. Then, generate alerts using the learned knowledge if anomaly patterns are predicted or detected; (ii) Make the energy/power meters autonomously learn the usual residential electricity consumption patterns and raise alerts in the event of unusual usage or overconsumption. Thus, *ML-MCU* can make the power meters self-learn to perform offline analytics that can reduce bills, detect leaks, etc.

## ACKNOWLEDGEMENT

This publication has emanated from research supported in part by a research grant from Science Foundation Ireland (SFI) under Grant Number SFI/16/RC/3918 (Confirm) and also by a research grant from Science Foundation Ireland (SFI) under Grant Number SFI/12/RC/2289_P2 (Insight), with both grants co-funded by the European Regional Development Fund.

## REFERENCES

[1] C. Savaglio and G. Fortino, "A simulation-driven methodology for iot data mining based on edge computing," *ACM Transactions on Internet Technology (TOIT)*, vol. 21, no. 2, pp. 1–22, 2021.

[2] B. Sudharsan, J. G. Breslin, and M. I. Ali, "Edge2train: A framework to train machine learning models (svms) on resource-constrained iot edge devices," in *International Conference on the Internet of Things*, 2020.

[3] M. Chen, W. Li, G. Fortino, Y. Hao, L. Hu, and I. Humar, "A dynamic service migration mechanism in edge cognitive computing," *ACM Transactions on Internet Technology (TOIT)*, vol. 19, no. 2, pp. 1–15, 2019.

[4] B. Sudharsan, J. G. Breslin, and M. I. Ali, "Rce-nn: a five-stage pipeline to execute neural networks (cnns) on resource-constrained iot edge devices," in *Proceedings of the 10th International Conference on the Internet of Things*, 2020, pp. 1–8.

[5] B. Sudharsan, P. Patel, J. G. Breslin, and M. I. Ali, "Ultra-fast machine learning classifier execution on iot devices without sram consumption," in *IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, 2021, pp. 316–319.

[6] L. Zhang, M. Mahdavi, and R. Jin, "Linear convergence with condition number independent access of full gradients," in *Advances in Neural Information Processing Systems*, 2013, pp. 980–988.

[7] P. Richtárik and M. Takáč, "Iteration complexity of randomized block-coordinate descent methods for minimizing a composite function," *Mathematical Programming*, vol. 144, no. 1-2, pp. 1–38, 2014.

[8] C.-J. Hsieh, K.-W. Chang, C.-J. Lin, S. S. Keerthi, and S. Sundararajan, "A dual coordinate descent method for large-scale linear svm," in *Proceedings of the 25th international conference on Machine learning*.

[9] M. Schmidt, N. L. Roux, and F. Bach, "Minimizing finite sums with the stochastic average gradient," 2013.

[10] N. Garcia-Pedrajas and D. Ortiz-Boyer, "Improving multiclass pattern recognition by the combination of two strategies," *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

[11] M. Moreira and E. Mayoraz, "Improved pairwise coupling classification with correcting classifiers," in *ECML, 1998*.

[12] J. Manikandan and B. Venkataramani, "Design of a modified one-against-all svm classifier," in *2009 IEEE international conference on systems, man and cybernetics*. IEEE, 2009, pp. 1869–1874.

[13] G. Fortino, C. Savaglio, G. Spezzano, and M. Zhou, "Internet of things as system of systems: A review of methodologies, frameworks, platforms, and tools," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 2020.

[14] B. Sudharsan and P. Patel, "Machine learning meets internet of things: From theory to practice," *The European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD)*, 2021.

[15] J. Lee, M. Stanley, A. Spanias, and C. Tepedelenlioglu, "Integrating machine learning in embedded sensor systems for internet-of-things applications," in *IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)*, 2016, pp. 290–294.

[16] G. Huang, S. Liu, L. van der Maaten, and K. Q. Weinberger, "Condensenet: An efficient densenet using learned group convolutions," 2017. [Online]. Available: http://arxiv.org/abs/1711.09224

[17] M. Tan, B. Chen, R. Pang, V. Vasudevan, and Q. V. Le, "Mnasnet: Platform-aware neural architecture search for mobile," *CoRR*, vol. abs / 1807.11626, 2018. [Online]. Available: http://arxiv.org/abs/1807.11626

[18] C. Briggs, Z. Fan, and P. Andras, "A review of privacy preserving federated learning for private iot analytics," *arXiv preprint arXiv:2004.11794*.

[19] Q. Li, Z. Wen, and B. He, "Federated learning systems: Vision, hype and reality for data privacy and protection," *CoRR*, vol. abs / 1907.09693, 2019. [Online]. Available: http://arxiv.org/abs/1907.09693

[20] H. Keshavarz, M. S. Abadeh, and R. Rawassizadeh, "Sefr: A fast linear-time classifier for ultra-low power devices," *arXiv preprint arXiv:2006.04620*, 2020.

[21] G. Kamath, P. Agnihotri, M. Valero, K. Sarker, and W. Song, "Pushing analytics to the edge," in *IEEE Global Communications Conference, GLOBECOM*, 2016.

[22] A. Kumar, S. Goyal, and M. Varma, "Resource-efficient machine learning in 2 KB RAM for the internet of things," in *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, ser. Proceedings of Machine Learning Research, vol. 70. PMLR, 2017, pp. 1935–1944. [Online]. Available: http://proceedings.mlr.press/v70/kumar17a.html

[23] V. Joseph, S. Muralidharan, A. Garg, M. Garland, and G. Gopalakrishnan, "A programmable approach to model compression," *arXiv preprint arXiv:1911.02497*, 2019.

[24] A. Polino, R. Pascanu, and D. Alistarh, "Model compression via distillation and quantization," *CoRR*, vol. abs/1802.05668, 2018. [Online]. Available: http://arxiv.org/abs/1802.05668

[25] C. Gupta, A. S. Suggala, A. Goyal, H. V. Simhadri, B. Paranjape, A. Kumar, S. Goyal, R. Udupa, M. Varma, and P. Jain, "Protonn: Compressed and accurate knn for resource-scarce devices," in *Proceedings of the 34th International Conference on Machine Learning*.

[26] B. Sudharsan, P. Patel, J. G. Breslin, and M. I. Ali, "Sram optimized porting and execution of machine learning classifiers on mcu-based iot devices: demo abstract," in *Proceedings of the ACM/IEEE 12th International Conference on Cyber-Physical Systems (ICCPS)*, 2021.

[27] B. Sudharsan, P. Patel, A. Wahid, M. Yahya, J. G. Breslin, and M. I. Ali, "Porting and execution of anomalies detection models on embedded systems in iot: Demo abstract," in *International Conference on Internet-of-Things Design and Implementation (IoTDI)*, 2021.

[28] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[29] P. Gysel, M. Motamedi, and S. Ghiasi, "Hardware-oriented approximation of convolutional neural networks," *CoRR*, vol. abs/1604.03168, 2016. [Online]. Available: http://arxiv.org/abs/1604.03168

[30] S. Bhattacharya and N. D. Lane, "Sparsification and separation of deep learning layers for constrained resource inference on wearables," in *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems*. [Online]. Available: https://doi.org/10.1145/2994551.2994564

[31] A. Kumar, S. Goyal, and M. Varma, "Resource-efficient machine learning in 2 KB RAM for the internet of things," in *Proceedings of the 34th International Conference on Machine Learning, ICML 2017*.

[32] P. Richtárik and M. Takáč, "Parallel coordinate descent methods for big data optimization," *Mathematical Programming*.

[33] M. P. Friedlander and M. Schmidt, "Hybrid deterministic-stochastic methods for data fitting," *SIAM Journal on Scientific Computing*.

[34] G. Deng and M. C. Ferris, "Variable-number sample-path optimization," *Mathematical Programming*, vol. 117, no. 1-2, pp. 81–109, 2009.

[35] K. Marti and E. Fuchs, "Rates of convergence of semi-stochastic approximation procedures for solving stochastic optimization problems," *Optimization*, vol. 17, no. 2, pp. 243–265, 1986.

[36] V. Athitsos, A. Stefan, Q. Yuan, and S. Sclaroff, "Classmap: Efficient multiclass recognition via embeddings," in *2007 IEEE 11th International Conference on Computer Vision*. IEEE, 2007, pp. 1–8.