Train++: An Incremental ML Model Training Algorithm to Create Self-Learning IoT Devices

Bharath Sudharsan*, Piyush Yadav**, John G. Breslin*, and Muhammad Intizar Ali§
*Confirm SFI Research Centre for Smart Manufacturing, Data Science Institute, NUI Galway, Ireland
{bharath.sudharsan, john.breslin}@insight-centre.org
**Insight SFI Centre for Data Analytics, Data Science Institute, NUI Galway, Ireland, piyush.yadav@insight-centre.org
§School of Electronic Engineering, Dublin City University, Ireland, ali.intizar@dcu.ie

***Abstract*—The majority of Internet of Things (IoT) devices are tiny embedded systems with a micro-controller unit (MCU) as its brain. The memory footprint (SRAM, Flash, and EEPROM) of such MCU-based devices is often very limited, restricting onboard Machine Learning (ML) model training for large trainsets with high feature dimensions. To cope with memory issues, the current edge analytics approaches train high-quality ML models on the cloud GPUs (uses large volume historical data), then deploy the deep optimized version of the resultant models on edge devices for inference. Such approaches are inefficient in concept drift situations where the data generated at the device level vary frequently, and trained models are clueless on how to behave if previously unseen data arrives. In this paper, we present *Train*++, an incremental training algorithm that trains ML models locally at the device level (e.g., on MCUs and small CPUs) using the full *n-samples* of high-dimensional data. *Train*++ transforms even the most resource-constrained MCU-based IoT edge devices into intelligent devices that can locally build their own knowledge base *on-the-fly* using the live data, thus creating smart self-learning and autonomous problem-solving devices. *Train*++ algorithm is extensively evaluated on 5 popular MCU-boards, using 7 datasets of varying sizes and feature dimensions. A few exciting findings when analyzing the evaluation results are: (i) The proposed method reduces the onboard binary classifier training time by $\approx$ 10 - 226 sec across various commodity MCUs; (ii) *Train*++ can infer on MCUs for the entire test set in real-time of 1 ms; (iii) The accuracy improved by 5.15 - 7.3% since the incremental characteristic of *Train*++ enabled the loading of full *n-samples* of the high-dimensional datasets even on MCUs with only a few hundred kBs of memory.**

***Index Terms*—Intelligent Microcontrollers, Online Learning, Optimization, Incremental Learning, Edge Computing.**

I. Introduction

IN the real-world, every new scene generates unseen data patterns. When an ML model deployed over edge devices [1, 2] 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. 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 [3]. In order to achieve truly autonomous local intelligence at the device level, the devices must have the ability to *self-learn* and *understand* the data patterns with no dependency on users or cloud services. In other words, if we provide edge devices the ability to autonomously retrain themselves (self-learning), they become intelligent machines capable of learning to perform analytics in any given environment.

In most real-life IoT scenarios, designing a problem-solving AI is a lengthy and expensive process that demands skills in statistics, data science, and access to complex datasets that are difficult to source (GDPR restrictions and privacy concerns) [4]. Typically, historical data is collected at a central location for years, using which high-quality ML models are trained. Once trained, the models are deeply compressed and deployed on edge devices [5]–[7] across the world to perform edge analytics [8]. In cases where the historical data becomes obsolete, or if it is not truly representative for possible cases, the edge analytics produces inferences at low accuracy (i.e., whenever the device processes the previously unseen data) [9]. In this paper, we propose *Train*++, a resource-friendly algorithm to train ML models at the device level, without the need for any cloud-based ML training services. When devices are equipped with *Train*++, the devices gain the ability to re-train themselves locally and build knowledge *on-the-fly* using the live IoT data streams. Thus, *Train*++ transforms resource-constrained devices into intelligent devices that can train and infer offline at the edge.

The state-of-the-art ML frameworks are not suitable for training models on resource-constrained hardware like commodity MCUs, small CPUs, and FPGAs since executing the frameworks alone requires hundreds of megabytes (MB) for storage, high memory-resource, file system support, high clock speeds, multiple cores, parallel execution units, etc [10]. The majority of IoT devices cannot afford to have such high specifications required by the modern ML frameworks. The memory of MCUs, which is the brain for billions of edge devices (or tiny embedded systems) deployed worldwide, is limited to a few MBs, thus restricting (upper bound is imposed) onboard model training using high features and large trainsets [11].

To enable training ML models offline on edge MCUs without upper bounds, we designed the *Train*++ algorithm to possess both resource-friendly and incremental ML model

training characteristics. For each new data input, *Train++* predicts an output, which is a yes/no decision much similar to the binary classification. After prediction, we show the correct outcome/labels, so the algorithm modifies its classifier, aiming to obtain accurate predictions on the upcoming rounds. We solve a constrained optimization problem to perform this classifier modification while also making the updated classifier stay very close to the current classifier version so as to retain the information learned in previous rounds. When *Train++* is deployed on devices, it reads the live data, learns from it using the incremental method, then discards it, thus saving the memory required to store training data. The main contributions can be summarised as follows:

ML model training on commodity MCUs. To the best of our knowledge, the work presented in this paper is one of the few recent novel approaches enabling high-performance ML model training on MCUs. We provide *Train++* algorithm and open-sourced its C++ implementation, using which users can train binary classifiers on MCUs using a large volume of high dimensional real-time IoT use-case data. We provide *Train++ Pipeline* to show the users how the core *Train++* algorithm can be used to produce self-learning devices capable of learning to perform analytics for any target IoT use cases.

Validation study and evaluation results. We extensively evaluate *Train++* by loading its C++ implementation on 5 different MCUs and making it perform onboard ML model training and inference using 7 datasets that have various train set sizes and feature dimensions. A few of the exciting findings when analyzing the evaluation results are: (i) *Train++* is compatible with various MCU boards and multiple datasets. (ii) It can load, train, and infer using high features and large-size datasets even on MCU boards with only a few hundred kBs of memory. (iii) The models trained on MCUs using *Train++* show accuracy close to those of Python scikit-learn trained classifiers on high-resource CPUs. (iv) For few datasets, MCUs trained faster than CPUs due to the high-performance characteristics of *Train++*, its algorithmic simplicity (implementation less than 100 lines), and independence of third-party libraries. (v) Across various MCUs, *Train++* consumed 34000 - 65000x times less energy to train and consumed 34 - 66x times less energy for unit inference.

The remainder of this paper is structured as follows; Section II contains a comprehensive review of the state-of-the-art approaches. Section III presents the core contributions of this paper that is the *Train++* algorithm to create self-learning IoT devices. In Section IV, algorithm evaluation and results comparison is performed. Section V discusses the real-world benefits of *Train++*, current limitations and provide a greater context for future research. Finally, Section VI concludes the paper.

II. Related Work

To ensure comprehensiveness, we cover the research works from both the ML model training, inference on MCUs domain, and the classifier training methods domain.

A. Machine Learning on Microcontrollers

Training ML models on MCUs is an emerging area of research. The majority of the existing frameworks like Tensorflow Micro [12], Edge-ML [13], Open-NN [14], etc., do not yet provide methods to enable training models on MCUs. To achieve resource-efficient training, so far, researchers have focused on optimizing existing algorithms to run them on various resource-constrained setups. For example, Lee et al. [15] executed a gaussian mixture model 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. SEFR, a low-power classifier [18], is the most related work to enable a binary classifier training and inference on MCUs. The Artificial Intelligence for Embedded Systems (AIfES) library is a C-based platform-independent tool for generating NNs compatible with a range of open-source MCU boards. AIfES can be used with Windows and embedded Linux platforms by producing efficient code in the form of Dynamic Link Library (DLL). Similar to ML-MCU [19], Edge2Train [8], TinyOL [20] and Globe2Train [21, 22], AIfES permits to implement ML model training process on the embedded devices. Cartesiam NanoEdge AI Studio [23] enables the creation of ML static libraries to embed them in Cortex-M MCUs. It allows integrating the training process within the constrained device. In addition, it also can perform unsupervised algorithm [24] training on MCUs.

However, the above-mentioned work and other similar algorithms [9, 25] 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. *Train++* algorithm is superior to state-of-the-art methods since, even on tiny MCU-based devices, it can load and train faster, using the full *n-samples* of high-dimensional data, thus producing self-learning devices capable of learning to perform analytics for any target IoT use cases.

There is another category of research work presenting algorithms for resource-efficient model inference on MCUs [26]. Here, a set of articles propose compression techniques to reduce the size of the model's weights using quantization and pruning techniques. Condensa [27], a system for users to compose simple operators to build complex model compression strategies. In [28], two new compression methods jointly leverage weight quantization and distillation of larger networks. In [25], a tree-based algorithm for efficient prediction in milliseconds even on slow MCUs was implemented. Similarly, ProtoNN [29], a k-Nearest Neighbor inspired algorithm with several orders of low storage, and prediction complexity addresses the problem of real-time

and accurate prediction on resource-scarce devices. In both [27, 28] and other similar articles proposing compression [30, 31] and optimization [32, 33] methods, the models are trained in high resource setups, then a multi-stage MCU-aware optimization (tailored) is performed before deployment [34]. Similar to the above works, even the *Train++* can infer in super real-time on MCUs. We do not attempt to outperform their state-of-the-art inference methods since they are tailored to be application-specific, whereas ours can train and infer onboard using real-time data from any IoT use-cases.

B. Incremental Classifier Training Algorithms

The concept of updating or modifying a classifier to improve its performance by solving a constrained optimization problem has been presented in [35, 36]. There is widespread literature for online margin-based inference algorithms whose roots date back to the Perceptron algorithm [37, 38]. Recent examples include ROMMA [39], ALMA [40], NORMA [41], and the MIRA [42] algorithms. The NORMA algorithm shares a similar view of classification problems, but its update rule is based on a stochastic gradient approach. The MIRA and Herbster's [43] algorithm for binary classification (both designed to solve separable problems) is closely related to our work. We surpass them by making *Train++* applicable for both separable and non-separable settings and also extending it to solve regression problems.

Another set of articles present an incremental learning approach for different applications such as, for automatic online picture collection [44], incremental and decremental training for linear classification [45]. Also, the online methods have been well studied, a few related articles are, in [46], a robust and efficient algorithm for online classification problems was presented, online active learning techniques for online classification tasks was investigated in [47]. In [48], similar to the proposed approach, an online passive-aggressive algorithm based low-budget online learning algorithm was presented. The proposed *Train++* can also be viewed as a highly-optimized online binary classifier training method that aims to enable training models, but on highly resource-constrained MCU-based IoT devices.

III. Train++ Design

In this section, we present the *Train++ pipeline*, then the core *Train++* algorithm, which is a part of the pipeline that enables ML model training on commodity MCUs.

A. Train++ Pipeline to Create Self-Learning IoT Devices

In Fig. 1, using a four-step pipeline, we show how to use the proposed *Train++* algorithm to enable the edge MCUs to self-train offline for any IoT use case. This pipeline enables users to build powerful ML models quickly and inexpensively without needing statistics, data science skills, and complex datasets. Here, in *Step One*, the users can select any MCU of choice depending on the target use case. Then in *Step Two*,

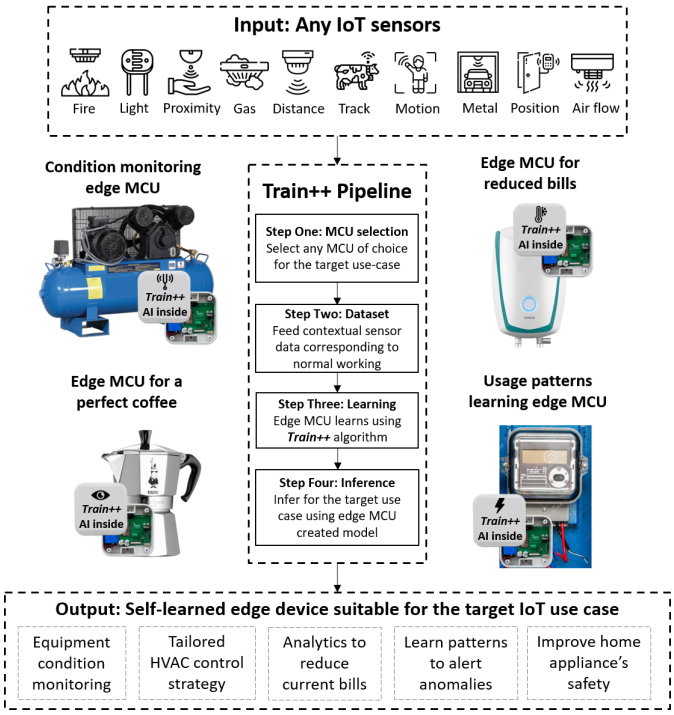

Fig. 1. Train++ pipeline produces devices capable of self-learning to perform analytics for any given target IoT use case.

the contextual sensor data corresponding to normal working will be collected as the local dataset and used for training in the next step. The labels/ground truth for the collected training data rows shall be computed using method from [8]. In *Step Three*, the edge MCUs learns using the*Train++* algorithm. In *Step Four*, the resultant MCU-trained model can start inference over previously unseen data. In the following, we brief a few use-case scenarios where *Train++* can train models offline on edge MCUs, producing devices capable of learning to perform analytics for any target IoT use cases.

Condition monitoring edge MCU. Incrementally learn by monitoring the contextual sensor data corresponding to regular vibration patterns from the pump's crosshead, cylinder and frame. Generate alerts using the learned knowledge if anomaly patterns are predicted or detected.

Edge MCU for a perfect coffee. Learn the data patterns of temperature, time, and material proportion when a person makes his/her best coffee. When brewing new coffees, use this knowledge to detect and alert when there is a considerable deviation from the best coffee patterns. Thus, we can ensure the person gets the coffee of his/her taste all the time.

Usage patterns learning edge MCU. Learn the usual residential electricity consumption patterns and raise alerts in the event of unusual usage or overconsumption. Thus, we can reduce bills, detect leaks, etc. Similarly, the devices can learn to perform continuous diagnostics for detecting fault-causing anomalies, learning to improve home appliance safety.

B. Train++ Algorithm Design

In the above use cases, the live sensor data stream which act as local training dataset is represented as

$$Dataset = \begin{cases} X = \{x_0; x_1 ::: x_t\} \text{ where } x_t \in \mathbb{R}^n \\ Y = \{y_0; y_1 ::: y_t\} \text{ where } y_t \in \{-1; +1\}, \end{cases} \quad (1)$$

Here, in Eqn (1), X is the dataset rows that contain features of input data samples, and Y holds the corresponding labels. We consider t (dimension of time) indefinite since real-time sensor data keeps arriving with indefinite length. Initially the algorithm infers using a binary classification function that updates from round to round and the vector of weights $w \in \mathbb{R}^n$, takes the $sign(x:w)$ form. The magnitude $|x:w|$ is the confidence score of prediction. After prediction, the task of *Train++* becomes to learn the weight vector w in an incremental method. w_t is the weight vector that *Train++* uses on round t and $y_t(x_t:w_t)$ is the margin obtained at t. Whenever the algorithm makes a correct prediction $sign(x_t:w_t) = y_t$. After prediction, we instruct the *Train++* to predict again with a higher confidence score. Hence, the goal of our algorithm becomes to achieve at least 1 as the margin, as frequently as possible. Whenever $y_t(x_t:w_t) < 1$ the following hinge-loss function makes *Train++* suffer an instantaneous loss,

$$l(w; (x; y)) = \begin{cases} 0 \text{ if } y(x:w) \geq 1 \\ 1 - y(x:w) \text{ otherwise}, \end{cases} \quad (2)$$

Here, if the margin exceeds 1, the loss is zero. Else, it is the difference between the margin and 1. Now we require an update rule to modify the weight vector for each round. *Train++* algorithm updates using this rule,

$$w_{t+1} = argmin \frac{1}{2}||w - w_t||^2 + C \xi \text{ s.t } l(w; (x_t; y_t)) \leq \xi; \quad (3)$$

ξ is a slack variable and C is the parameter to control the influence of ξ. Here, ξ is non-negative and C is positive. Whenever a correct prediction occurs, the loss function is 0 and the $argmin$ is w_t, hence the *Train++* algorithm becomes permissive. Whereas on the rounds when misclassifications occur, the loss is positive and *Train++* offensively forces w_{t+1} to satisfy the constrain $l(w; (x_t; y_t)) = 0$. Larger C values produce strong offensiveness, which might increase the risk of destabilization when input data is noisy. Whereas lower C values improve adaptiveness. To provide the ability for edge devices to cope with noisy input samples (wrong labeled), reduce rapid changes that produce consequent higher misclassification rates, finally, to keep the value of w_{t+1} close to w_t, to retain the information learned in previous rounds. We increase our algorithm's robustness. Hence the update rule in its simple closed-form is $w_t + \tau_t y_t x_t$. When substituting τ, it becomes,

Algorithm 1 Train++: A high-performance binary classifier training algorithm for MCU-based IoT devices.

Inputs:
C: Positive parameter to control the influence of ξ.
t: The dimension of time for real-time data inputs.
x_t: Real-time sensor data inputs. Where $x_t \in \mathbb{R}^n$.
y_t: Correct labels. Where $y_i \in \{-1; +1\}$.
Output:
w_t: Incrementally learned weights.
Receive live data stream, represented as in Eqn (1).
function MCUTrain ($x_t; y_t; t; C$)
for t = 0 to setsize **do**
Predict $\hat{y}_t$ for every x_t. Use $sign(x_t:w_t)$.
Compute confidence score of prediction.
Use $|x_t:w_t|$.
Compute margin at t. Use $y_t(x_t:w_t)$.
Show original label y_t to this algorithm.
if ($\hat{y}_t$ equals to y_t) **then**
Prediction was correct. $y_t = sign(x_t:y_t)$.
Predict again with a higher confidence score.
if ($sign(x_t:y_t)$ lesser than 1) **then**
Suffer an instantaneous loss. Use Eqn (2).
if ($sign(x_t:y_t)$ exceeds 1) **then**
Loss becomes 0.
else
Wrong prediction. Loss becomes 1 - $y_t(x_t:w_t)$.
end if
Incrementally learn w_t. Use Eqn (4).
Store the learned weights w_t.
end for

$$w_{t+1} = w_t + \frac{l_t}{||x_t||^2 + \frac{1}{2C}} y_t x_t,$$

Then substituting the loss l_t, we get,

$$w_{t+1} = w_t + \frac{max\{0; 1 - y_t(x_t:w_t)\}}{||x_t||^2 + \frac{1}{2C}} y_t x_t, \quad (4)$$

Now, this update rule meets our expectations since the weight vector is updated, with a value whose sign is determined by y_t, with a magnitude proportional to the error. During correct classification, the nominator of this equation becomes 0, so $w_{t+1} = w_t$. During misclassification, the value of the weight vector will move towards x_t and stops with a loss of l . This movement is usually very tiny. After this movement, the dot product in the update rule becomes negative, hence the input is classified correctly as +1.

In the remainder of this section, we modify *Train++* algorithm for regression problems. Here, all the inputs x_t are associated with their corresponding labels $y_t \in R$ (labels/target/ground truth are computed using the method from [8]), where we try to predict $\hat{y}_t$ on every round. In the

linear regression function, $y_t = x_t.w_t$, w_t is the vector that is learned incrementally using *Train++* algorithm. Similar to the binary classification scenario explained above, after inference, we show the true label y_t to the algorithm, but here we use a new loss function,

$$l\ (w;(x;y)) = \begin{cases} 0 \text{ if } |x \quad y.w| \\ |x \quad y.w| \quad \text{otherwise} \end{cases}, \tag{5}$$

Here, is the parameter to control the algorithm's sensitivity to wrong predictions. If the predictions deviate from the true labels by less than , the loss is zero. Else, the loss grows linearly with value of deviation, $|\hat{y}_t \quad y_t|$. At the end of every round, *Train++* uses w_t, the input (x_t), and its label (y_t) to generate a new weight vector w_{t+1}, which will be used as weights for the next round. This new weight vector is set using:

$$w_{t+1} = argmin\frac{1}{2}||w \quad w_t||^2 \text{ s.t } l(w;(x_t;y_t)) \quad ,$$

Similar to binary classification, this update rule can also be written in its closed-form as $w_t + sign(y_t \quad \hat{y}_t)\ _t x_t$. When substituting and l_t, it becomes,

$$w_{t+1} = w_t + \frac{max\{0;1 \quad y_t(x_t.w_t)\}}{||x_t||^2 + \frac{1}{2C}} x_t sign(y_t \quad \hat{y}_t) \ , \tag{6}$$

The *Train++* classifier training method is summarised in Algorithm 1. We implemented [1] this algorithm in C++ and open-sourced it so users can train models on MCU-based edge devices using their live IoT use case data. After training, when the MCUs want to infer, we pass new inputs, and the incrementally learned weights w_t to a function that performs sign(w_t.new inputs) to predict the output for the newly fed inputs. *Train++* is applicable for regression problems too, when its loss function in Algorithm 1 is replaced with Eqn (5), then the incrementally learn function with Eqn (6).

From our experimental experience, we report that training NNs on commodity MCUs is not feasible since it requires hardware resources (particularly computational power and memory) orders of magnitude higher than what is available on MCU-based IoT devices. Even if we manage to implement the training of a basic network (only a few layers) on MCUs that usually has less than 1 MB memory (see Table I), the resultant models will show less accuracy than simple algorithms like SVM, Random Forests, etc. In [8], we had already explored training SVMs on various popular MCU boards.

IV. Train++ Algorithm Performance Evaluation

Here we perform multiple datasets and MCUs based extensive experimental evaluation that aims to answer:

- Is *Train++* compatible with different MCU boards, and can it train ML models on MCUs using datasets with various feature dimensions and sizes?

[1]Train++ code: https://github.com/bharathsudharsan/Train_plus_plus

TABLE I
Datasets and hardware chosen for Train++ evaluation.

	Dataset#: Name - feature dimension		
Datasets	D1: Iris Flowers [49] - 4		D5: Banknote [50] - 5
	D2: Heart Disease [51] - 13		D6: Survival [52] - 3
	D3: Breast Cancer [53] - 30		D7: Titanic [54] - 11
	D4: MNIST Digits [55] - 64		
	MCU#	**Name**	**Specs: processor flash, SRAM (kB), clock (MHz)**
MCU boards	1	nRF52840 Feather	Cortex-M4, 1MB, 256, 64
	2	STM32f10 Blue Pill	Cortex-M0 128kB, 20, 72
	3	Adafruit HUZZAH32	Xtensa LX6,
	4	Generic ESP32	4MB, 520, 240
	5	ATSAMD21 Metro	Cortex-M0+, 256kB, 32, 48
	CPU#	**Name**	**Basic specs**
CPU devices	1	W10 Laptop	Intel Core i7 @1.9GHz
	2	NVIDIA Jetson Nano	128-core GPU @1.4GHz
	3	W10 Laptop	Intel Core i5 @1.6GHz
	4	Ubuntu Laptop	Intel Core i7 @2.4GHz
	5	Raspberry Pi 4	Cortex-A72 @1.5GHz

- Can *Train++* load, train, and infer using high features and size datasets on limited memory MCU boards that have low hardware specification and no floating point unit (FPU), accelerated processing unit (APU), convolution operation accelerator (KPU) support?
- What is the impact on accuracy when training ML models on MCUs using *Train++*, and how much does the accuracy vary in comparison with models trained on high resource CPU/GPU setups?

In Section IV-A, we explain the experiments and results comparison procedure. In Sections IV-B - IV-F, we present, analyze and compare the obtained results.

A. Datasets and Experimentation Procedure

Table I presents the datasets and hardware chosen to evaluate the performance of *Train++* algorithm. Using *Train++*, for datasets D1-D4, we train a binary classifier on MCUs 1-5. Datasets D5-D7 are used in the latter part of this section. For the first dataset D1, all the classifiers trained on MCUs 1-5 (using Train++) should distinguish Iris Setosa from other flowers based on the input features. Similarly for D2, the MCU-trained classifiers should identify the presence of heart disease in the patient. Similarly, for D3, the classifiers should be able to predict the class names based on the input features from the test set. For D4, digit six should be recognized from other digits.

As explained in Section II, training ML models on MCUs is an emerging research area. During the time of writing, *Edge2Train* [8] is the only work that enables training of ML models (SVMs) on MCUs. Also, its code is made available online that can be used to reproduce experimental results. We compare the evaluation results of *Train++* trained models with *Edge2Train* trained models. During comparison, we use the same datasets, MCUs, and procedure as from *Edge2Train*.

TABLE II
HIGH-PERFORMANCE ML MODEL TRAINING ON MCUS USING TRAIN++: ACCURACY, MEMORY AND TIME CONSUMED BY MCUS FOR ON-BOARD TRAINING AND INFERENCE.

MCU #	Dataset Dimension & Size (No. of row)	Train Time (ms)	Accuracy (%)	Inference Time (ms)	Flash & SRAM Req (kB)
1	Iris Flowers 4, 150	< 1	97.33	< 1	41.62, 8.76
		< 1	80.18	< 1	51.95, 18.74
		5	85.0	1	110.41, 77.32
		7	98.0	1	133.46, 100.36
2	Heart Disease 13, 212	4	96.0	1	269.36, 6.24
		13	82.08	1	36.93, 16.23
		66	78.0	9	+53.6, +29.0
		83	95.0	11	+76.7, +52.0
3	Breast Cancer Dataset 30, 567	< 1	96.67	< 1	217.27, 17.37
		< 1	80.0	< 1	227.35, 27.35
		2	78.0	< 1	285.94, 859.32
		2	98.0	< 1	308.97, 108.97
4	Handwritten Digits 64, 356	< 1	97.33	< 1	217.27, 17.37
		< 1	80.0	< 1	227.35, 27.35
		2	73.0	< 1	285.94, 859.32
		2	95.0	< 1	308.97, 108.97
5		12	96.67	1	20.46, 9.20
		44	80.18	6	20.47, 19.17
		286	85.0	36	+46.7, +40.58
		304	97.75	45	+69.7, +63.62

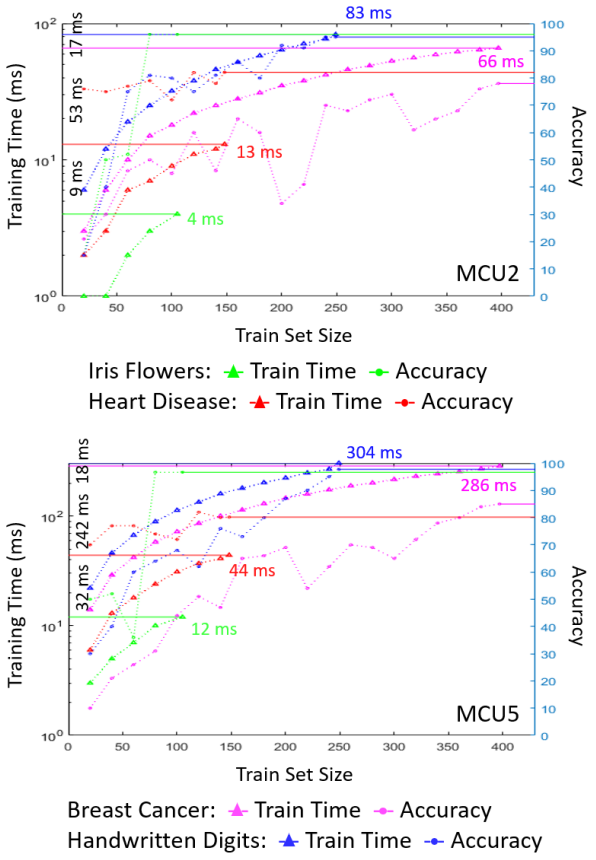

Fig. 2. Training models on MCUs using Train++: Comparing training set size vs training time and accuracy for selected datasets.

The difference is, we use *Train++* instead of *Edge2Train*'s classifier training algorithm.

We upload the *Train++* algorithm's C++ implementation on all MCU boards from Table I. 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 MCU boards. We perform a 70/30 training/testing split for each of the above datasets. When we instruct the boards to train, *Train++* iteratively loads the train set and trains the classifier using its method from Section III. Next, we load the test set on MCUs, make the MCU trained classifier models perform inference in order to evaluate them. The results are shown in Table II, using which we analyze and compare results in the upcoming subsections.

B. Training and Inference Time on MCU Boards

Here, in Table II, we record the ML model training and inference time consumed on MCUs when using *Train++*. Comparing with *Edge2Train* results, it is apparent that using Train++, MCU1 trained 33.5 sec faster for the Iris dataset, 45.7 sec faster for Digits. Likewise MCU2 trained 226.1 sec faster for Iris, and *Edge2Train* could not load the Digits dataset due to SRAM overflow. Since *Train++* has an incremental training characteristic, even after the SRAM requirement exceeds by +29 kB (Table II), we were able to load the dataset incrementally, to perform the training and complete training in 83 ms. MCUs 3, 4 trained 10 sec faster for Iris and 25 sec faster for Digits. The slowest MCU5 trained 785.5 sec faster for Iris, and although the SRAM requirement exceeds by +63.62 kB, using *Train++*, MCU5 was able to load the entire dataset incrementally and train in 304 ms. We are not comparing the inference time in detail since, for the same datasets, *Train++* models infer for the entire test set in lesser time than the *Edge2Train* model's unit inference time.

Next, to explain the relationship between training time, train set size, and feature dimension, using *Train++*, we trained binary classifiers on all MCUs 1-5 by providing training sets of varying sizes. We illustrate the results only for MCUs 2, 5 in Fig. 2 since other MCUs trained using the largest Breast Cancer dataset and the highest features Digits dataset very fast, in 2 ms. In this figure, the gap in the y-axis is the difference in the training time between the selected datasets, and for a clearer view, we marked this gap in ms. For MCUs 2, 5, we noticed that the training time grows almost linearly with the number of training samples for all the datasets. For the Iris dataset with 4-dimensional features, MCU2 only took 4 ms to train on 105 samples, whereas it took 83 ms to train on the Digits dataset with 64-dimensional features. MCU5

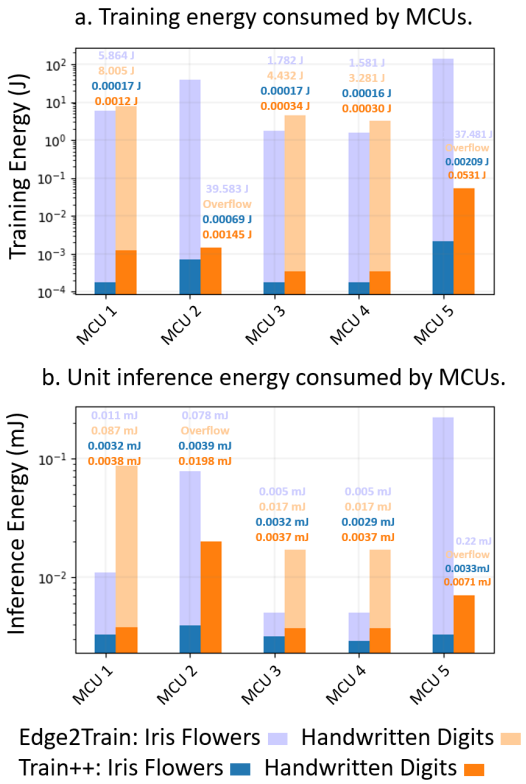

Fig. 3. Edge2Train vs Train++: Comparison of energy consumed for on-board classifier training and unit inference on MCUs 1-5.

is the slowest since it only has a 48 MHz clock and does not have FPU support. Hence it took 12 ms to train on 105 samples of the Iris dataset (3x times slower than MCU2) and 304 ms to train on the Digits dataset (3.6x times slower than MCU2).

C. ML Model Accuracy on MCU Boards

From Table II, the highest onboard classification accuracy is 97.33% for the Iris (D1), 82.08% for Heart Disease (D2), 85.0% for Breast Cancer (D3), and 98.0% for Digits dataset (D4). In Fig. 2, we illustrate the training sample size vs accuracy (accuracy scale in the right side y-axis). When comparing the accuracy of *Train++* trained models with *Edge2Train* trained models, for the same Iris dataset, the accuracy improved by 7.3% and by 5.15% for the Digits dataset. This improvement is because our training algorithm enabled incremental loading of the full dataset. Other algorithms like SVMs work in batch mode, requiring full training data to be available in the limited MCU memory, thus sets an upper bound on the train set size. Hence, as shown in Fig. 2, our models achieved overall improved accuracy compared to the *Edge2Train* models, which were trained with limited data (unable to load full dataset due to memory constraints).

D. Flash and SRAM Usage on MCU Boards

Most embedded systems have limited kBs of SRAM, which restricts training models using high feature dimensions and large train sets. *Train++* unrestricted this upper bound, thus enabling us to train using the full *n-samples* of the high-dimensional datasets. We provide the Flash and SRAM requirements (calculated by the compiler for target MCUs) in Table II. For MCU1, Iris dataset and *Train++* algorithm in total used only 4.1% of Flash and 3.4% SRAM. For the Digits, the same MCU1 requires 6.54% and 29.93%. When we use the *Edge2Train*, in the case of MCUs 2, 5, we cannot train using the Digits dataset because SRAM overflowed by +52.0 kB and +63.62 kB. Similarly, for Heart Disease, both the Flash and SRAM requirements exceed the MCU's capacity. The results from Table II shows that the incremental training characteristic of *Train++* enables training on limited Flash and SRAM footprints while also allowing to use full *n-samples* of the high-dimensional datasets.

E. Energy Consumption on MCU Boards

We follow the same procedure as is *Edge2Train* to calculate the energy (in Joules) consumed by MCUs to train and infer using the proposed method. We multiply the Current (Amperes) rating of MCUs with its Potential/Voltage (Volts) and corresponding task time (seconds). In this formula, the task time is the training and inference time (values are from Table II) consumed by *Train++* when executing on MCUs 1-5. Then, for comparison purposes, we plot thus calculated energy along with *Edge2Train* energy consumption results, in the form of a stacked bar chart.

Here, from Fig. 3 (y-axis in base-10 log scale), it is apparent that *Train++* consumed 34000 - 65000x times less energy to train and consumed 34 - 66x times less energy for unit inference. For a given task that needs to be completed by using the same datasets on the same MCUs, *Train++* achieved such significant energy consumption due to its high-performance characteristics (i.e., it trained and inferred at much higher speeds than other methods). Hence, when *Train++* is used, MCUs can perform onboard model training and inference at the lowest power costs, thus enabling offline learning and model inference without affecting the IoT edge application routine and operating time of battery-powered devices.

F. MCUs vs CPUs: ML Model Performance

Here, additional experiments are performed to extensively evaluate the training performance of *Train++* on MCUs. After training using the datasets D5 - D7, we analyze and compare the *Train++* results produced on MCUs with the model training results produced on CPUs. The CPUs selected for experimentation are given in Table I. An 80/20 training/testing split is performed for datasets D5 - D7, using which *Train++* performs onboard model training. Using Arduino IDE, the train sets of split dataset and C++ implementation of the *Train++* algorithm is uploaded on all 5 MCUS, using the

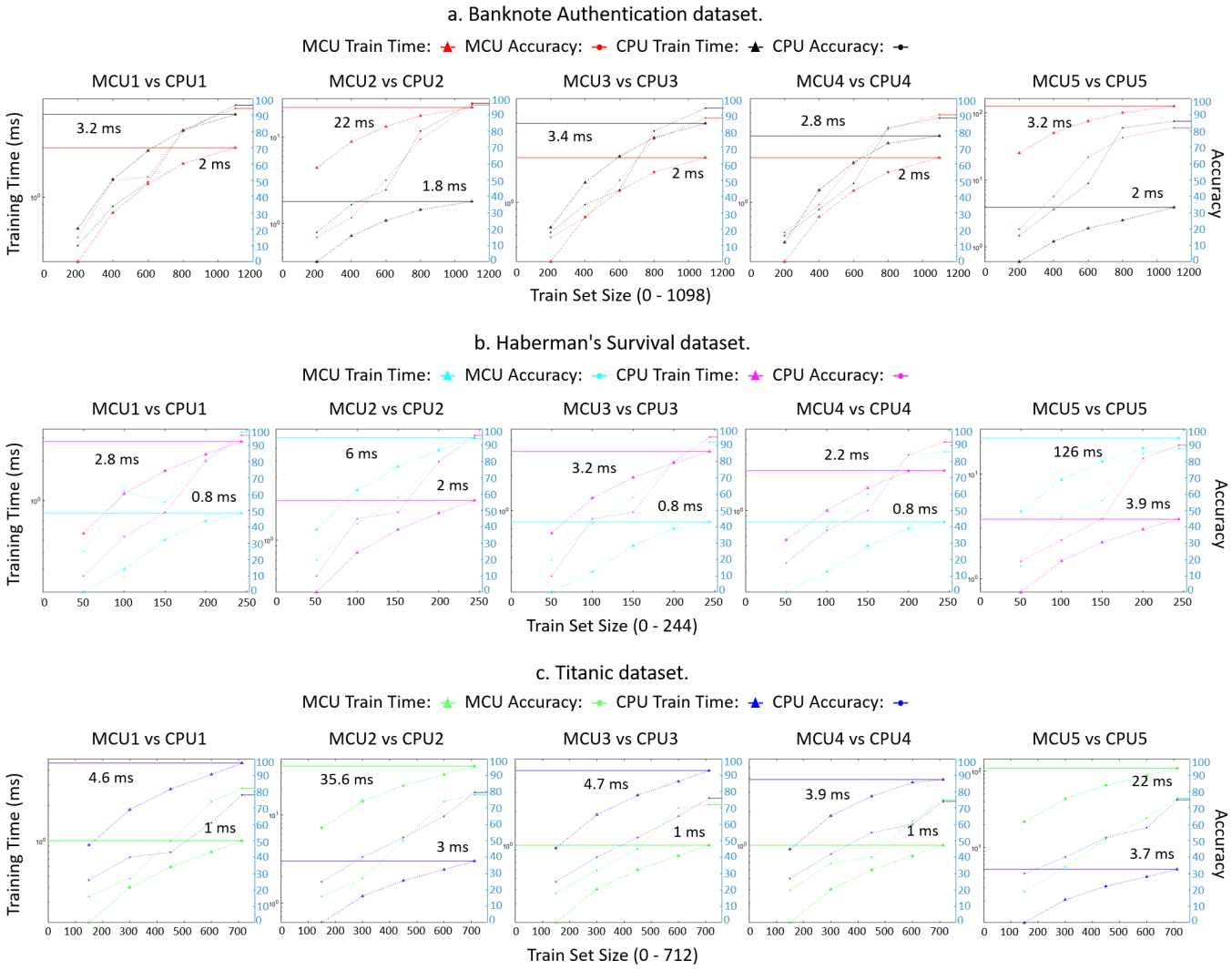

Fig. 4. Onboard training of ML models on various MCUs (uses C++ implementation of Train++) and CPUs (uses standard Python scikit-learn) using the same selected datasets: Comparing training time and accuracy (training results) produced by MCUs 1-5 with the results of CPUs 1-5.

Arduino IDE. Each MCU board is then powered and connected to PC via serial port to receive training time and classification accuracy.

In Fig. 4 (left y-axis in base-10 log scale), we present the training time and accuracy (right y-axis in percentage) obtained during training across MCUs 1-5. We follow the same setup and procedure when training on high-resource CPUs 1-5. The results are then shown in the same Fig. 4 to compare the *Train++* trained models with Python scikit-learn models. For statistical validation, the plotted time corresponds to the average of 5 runs. In Fig. 4, we use labels to help quickly identify the time consumed by MCUs and CPUs to train using the complete train set of the selected datasets. In the remainder of this section, using Fig. 4, we analyze and compare the accuracy and training time of the MCU models with CPU models.

MCUs vs CPUs: Training Time Comparison. Although the CPUs 1, 3, 4 have 1000x times better specifications and 200x times more expensive than MCUs. It is apparent from Fig. 4 a-c that, across all the datasets, MCUs 1, 3, 4 trained faster than their competitor CPUs. In the CPU class, the Jetson Nano with the highest hardware resource trained using Banknote dataset (D5) in 1.8 ms, in 2 ms for Haberman's survival dataset (D6), and 3ms for the Titanic dataset (D7). In the MCU class, the MCUs 3, 4, based on the same LX6 processor have the highest resource, trained in 2 ms, 0.8 ms, and 1 ms (same dataset sequence as above).

MCUs vs CPUs: Accuracy Comparison. For all the datasets, after training using various train set sizes, the resultant models were evaluated using the test set (20% of whole data), and the corresponding accuracy is plotted in Fig. 4. For D5, the highest accuracy was 96% on CPU2 and 95% on MCU2. For

D6, MCU1 performed the best, producing the highest accuracy of 98%, followed by CPU2 that showed 96% accuracy. In D7, we removed non-numeric features, then the rest of the procedure remains the same. For D7, MCU1 trained classifier produced the highest accuracy of 82%, and CPU2 showed 79.5% from its class. Although the training time on a few MCUs was higher than CPUs, *Train++* trained classifiers show classification accuracy close to those of Python scikit-learn trained classifiers on high-resource CPUs.

During this experiment, in default settings, the MCU2 could not load the full D5 and D7 datasets because SRAM overflowed by +8.71 kB and +11.20 kB. But the incremental training characteristic of *Train++* enabled training using the full *n-samples* of both the datasets. In addition to the offline model training and inference demonstration from Section IV, our algorithm trained using three new datasets with different sizes and feature dimensions.

V. Discussion

From the extensive evaluation of *Train++*, it is apparent that developers can utilize its C++ implementation to train models offline using real-time data from any of their use cases on commodity MCUs. We also estimate that using *Train++*, onboard model training, and super real-time unit inference can be performed on thousands of open-source MCU boards supported by Arduino IDE, which have limited Flash, SRAM, and no FPU, APU, KPU support. When the same experiments are run on the latest Artificial Intelligence of Things (AIoT) hardware, *Train++* can perform onboard training at much higher speeds and produce unit inference in microseconds.

As demonstrated with results, such tiny resource-constrained devices could train faster than CPUs because the*Train++* algorithm does not depend on external or third-party libraries, and its core implementation is just *less than 100 lines*. Hence, the devices using our method consume lesser storage and time (to load and execute models) than Python or other ML framework classifiers. But, for high-features and large datasets with heterogeneous contents, server-class GPUs are preferred over MCUs and CPUs. Our focus is on the MCUs since, in the real world, billions of IoT edge devices are MCU-based, where it is feasible to train even at lesser speeds. Such offline training using *Train++* reduces the hardware cost of edge devices since they do not need a wireless module (4G or WiFi) to receive the updated models from the cloud [56, 57]. Also, when the data for which the model has to be updated is small, it does not require data center GPUs for training. Models can rather be trained on the edge, using the proposed method, without compromising the model accuracy.

We see the lack of real-world experimental evaluation as the major limitation. Also, the algorithm behavior and its on-device self-learning performance need to be investigated by deploying the implementation (code) of *Train++* 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 by monitoring and learning from 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, *Train++* can make the power meters self-learn to perform offline analytics that can reduce bills, detect leaks, etc.

VI. Conclusion

We presented *Train++*, a resource-friendly binary classifier training algorithm that enables the onboard training of high-performance ML models on commodity MCUs. When MCU-based IoT devices (tiny embedded systems) are equipped with *Train++*, they get transformed into intelligent devices that can self-learn (locally re-train themselves) using large volume, high dimensional, real-time data. Thus, even the resource-constrained tiny IoT hardware with *Train++* can self-learn to perform analytics for any target IoT use cases. The extensive evaluation using 7 datasets shows that *Train++* can guarantee superior onboard model training performance, accuracy, and perform ultra-fast inference while showing a higher level of memory conservation. In future work, we plan to implement the *Train++* algorithm for regression problems which was outlined in this paper.

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 SFI under Grant Number SFI/12/RC/2289_P2 (Insight), with both grants co-funded by the European Regional Development Fund.

References

[1] B. Sudharsan, S. Malik, P. Corcoran, P. Patel, J. G. Breslin, and M. I. Ali, "Owsnet: Towards real-time offensive words spotting network for consumer iot devices," in *IEEE 7th World Forum on Internet of Things*, 2021.

[2] B. Sudharsan, P. Corcoran, and M. I. Ali, "Smart speaker design and implementation with biometric authentication and advanced voice interaction capability." in *27th AIAI Irish Conference on Artificial Intelligence and Cognitive Science (AICS)*, 2019.

[3] P. Yadav, D. Salwala, and E. Curry, "Vid-win: Fast video event matching with query-aware windowing at the edge for the internet of multimedia things," *IEEE Internet of Things Journal*, 2021.

[4] J. R. Zhao, R. Mortier, J. Crowcroft, and L. Wang, "Privacy-preserving machine learning based data analytics on edge devices," *Conference on AI, Ethics, and Society (AIES)*, 2018.

[5] B. Sudharsan, D. Sundaram, J. G. Breslin, and M. I. Ali, "Avoid touching your face: A hand-to-face 3d motion dataset (covid-away) and trained models for smartwatches," in *10th International Conference on the Internet of Things Companion*, 2020.

[6] B. Sudharsan, J. G. Breslin, and M. I. Ali, "Adaptive strategy to improve the quality of communication for iot edge devices," in *IEEE 6th World Forum on Internet of Things (WF-IoT)*, 2020.