Edge2Train: A Framework to Train Machine Learning Models (SVMs) on Resource-Constrained IoT Edge Devices

Bharath Sudharsan
Confirm SFI Centre for Smart Manufacturing
Data Science Institute
National University of Ireland Galway
b.sudharsan1@nuigalway.ie

John G. Breslin
Confirm SFI Centre for Smart Manufacturing
Data Science Institute
National University of Ireland Galway
john.breslin@nuigalway.ie

Muhammad Intizar Ali
Confirm SFI Centre for Smart Manufacturing
School of Electronic Engineering
Dublin City University
ali.intizar@dcu.ie

ABSTRACT

In recent years, ML (Machine Learning) models that have been trained in data centers can often be deployed for use on edge devices. When the model deployed on these devices encounters unseen data patterns, it will either not know how to react to that specific scenario or result in a degradation of accuracy. To tackle this, in current scenarios, most edge devices log such unseen data in the cloud via the internet. Using this logged data, the initial ML model is then re-trained/upgraded in the data center and then sent to the edge device as an OTA (Over The Air) update. When applying such an online approach, the cost of edge devices increases due to the addition of wireless modules (4G or WiFi) and it also increases the cyber-security risks. Additionally, it also requires maintaining a continuous connection between edge devices and the cloud infrastructure leading to the requirement of high network bandwidth and traffic. Finally, such online devices are not self-contained ubiquitous systems. In this work, we provide Edge2Train, a framework which enables resource-scarce edge devices to re-train ML models locally and offline. Thus, edge devices can continuously improve themselves for better analytics results by managing to understand continuously evolving real-world data on the fly. In this work, we provide algorithms for Edge2Train along with its C++ implementations. Using these functions, on-board, offline SVM training, inference, and evaluation has been performed on five popular MCU boards. The results show that our Edge2Train-trained SVMs produce classification accuracy close to that of SVMs trained on high resource setups. It also performs unit inference for values with 64-dimensional features 3.5x times faster than CPUs, while consuming only $1/350^{th}$ of the energy that CPUs consume.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning**; • **Computer systems organization** → **Embedded hardware**;

KEYWORDS

Intelligent Microcontrollers, Embedded C, Real-time Machine Learning, Self-learning IoT Edge.

ACM Reference Format:
Bharath Sudharsan, John G. Breslin, and Muhammad Intizar Ali. 2020. Edge2Train: A Framework to Train Machine Learning Models (SVMs) on Resource-Constrained IoT Edge Devices. In *Proceedings of 10th International Conference on the Internet of Things, Malmö, Sweden, October 6–9, 2020 (IoT '20),* 8 pages.
https://doi.org/10.1145/3410992.3411014

1 INTRODUCTION

模型需要从采集的未标记的外界数据中不断提升自己

During the past few years, ML models have been used as the principal approach to solve a variety of real-world problems across domains such as machine translation, voice localization [14], handwriting recognition, computer vision [15], etc. Such models exhibit superior performance only when they are trained using a massive, high-quality dataset with data fields containing a variety of patterns. In the real-world, every new scene generates a fresh, unseen data pattern. When the model deployed in edge devices sees such fresh patterns which were not exposed during training, it will either not know how to react to that specific scenario, or lead to false or less accurate results. Also, a model trained using data from one context will not produce the expected results when deployed in another context, and it is not feasible to train multiple models for multiple environments and contexts. Hence, a solution is needed to enable edge devices to continuously improve themselves, by learning from the fresh real-world data patterns they see after their deployment.

Nowadays, to achieve high inference accuracy, edge devices log unseen data in the cloud via the internet. Using this logged data, the initial model is re-trained and upgraded in data centers and then sent to the edge device as an over-the-air (OTA) update. When employing such an online approach, firstly, the cost of edge devices increases due to wireless module addition, while also increasing the cyber-security risks and power consumption. Expensive edge devices are also prone to security issues such as theft or vandalism. Finally, such online devices are not self-contained ubiquitous systems and depend on the internet and cloud services to perform re-training and also inferences. Hence, the solution that enables continuous learning edge devices has to follow an offline learning approach.

通常，模型从
务器经过OTA
发到边缘设备

An edge device is an embedded system with a small microcontroller unit (MCU), which acts as its brain. The Arduino Nano, an 8-bit ATmega328 micro-controller with a 16 MHz clock, 2 kB of

IoT '20, October 6–9, 2020, Malmö, Sweden

ACM ISBN 978-1-4503-8758-3/20/10…$15.00
https://doi.org/10.1145/3410992.3411014

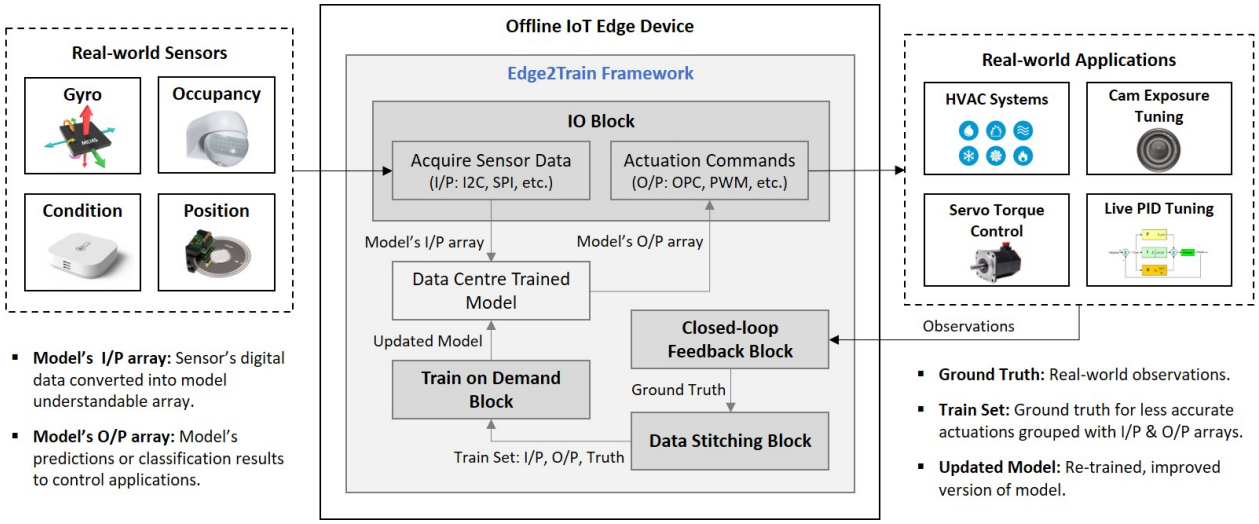

Figure 1: Architecture and components of Edge2Train: training models on resource-scarce MCUs at the IoT edge.

SRAM and 32 kB of ISP flash memory, and the NUCLEO-F303K8, a 32-bit ARM Cortex-M4 micro-controller with a 72 MHz clock and 64 kB of flash memory, are examples of small MCU boards. In such MCU-based edge devices, firstly, the memory footprint (SRAM, FLASH and EEPROM) is limited to a few MB. Next, the computation core (commonly a single ARM Cortex-M CPU) only runs up to few hundred MHz, resulting in low operations per second. Next is the absence of native filesystem support. Finally, there is the issue of its inability to perform parallel processing due to the absence of multiple execution units. These constraints on MCUs restrict the execution of any ML-based tasks on MCU-based edge devices. To the best of our knowledge, our work presented in this paper is one of the few recent novel approaches enabling a model's training tasks on MCUs. Our proposed Edge2Train framework enables offline training of models on resource-scarce edge MCUs. Our main contributions in this paper can be summarised as:

- We provide the functions for Edge2Train, which are realized through C++ implementations of our algorithms. Using these functions, users can train models (SVMs) offline on MCUs using live data from their IoT use cases. These functions also enable on-board inference and model evaluations.
- The implementation blocks of our Edge2Train fuse with the device's IoT application to continuously improve analytic results by training using the evolving real-world data.

2 EDGE2TRAIN FRAMEWORK

We propose an Edge2Train framework that enables MCUs at the IoT edge to learn from a new data pattern it sees after deployment. Initially, the MCU of the edge device will execute the use case's IoT application loaded into its memory. This IoT application follows a routine, starting with acquiring sensor data using onboard peripherals (I2C, SPI, etc.), converting it into a model-understandable array, and feeding it to the deployed model. In the next step of the routine, the model stored in the MCU is executed to obtain an output array containing the inference results of the model. In the final step, the model's output array is used to generate actuation commands (OPC, PWM, etc.) to interact with the real-world applications the edge device is interfaced with.

In the background, Edge2Train monitors this IoT application's routine to compute feedback for each actuation. If the feedback is satisfactory, this routine is not disturbed. If not, a training set is generated by stitching the model's input and output array for the unsatisfactory actuation with the ground truth which was obtained by performing real-world observations. When sufficient training sets are generated, then the Train-on-Demand Block of the framework trains the model using the training sets and updates the existing model. Figure 1 gives an overall architecture of IoT applications on edge devices fused with our framework. The four blocks of Edge2Train are:

收集传感器信息，转换成模型的输入层向量，产生驱动命令

i IO Block: This block acquires sensor data using on-board peripherals (I2C, SPI, etc.), converts it into a model understandable array and feeds it to the input layer of the model. Based on the model's output array, this block generates actuation commands, which are continuous control signals (OPC, PWM, etc.) that interact with real-world applications.

ii Closed-Loop Feedback Block: This block monitors the IoT application routine from the background for computing feedback of each actuation. The output of this block is the ground truth, computed by observing how accurately the actuation commands control the real-world application.

闭环反馈模块这个块的输出是ground truth，通过观察驱动命令控制现实世界应用程序的准确度来计算

iii Data Stitching Block: This block locally builds a new dataset for providing it to the Train-on-Demand Block for training. The dataset is built by stitching the model's input and output array for the unsatisfactory actuation with the ground truth computed by the Closed-loop Feedback Block.

数据集是通过将模型的输入和输出数组与闭环反馈块计算的基本事实拼接起来的

iv Train-on-Demand Block: When the current model is unable to achieve adequate actuation accuracy, the Data Stitching Block starts generating training sets. When sufficient training sets are generated, this block leverages our C++ Edge2Train library to set up, train and evaluate a new model, which is an update for the existing model.

We implement this framework as a C++ library that can be leveraged by developers during the programming phase of their MCUs for their use case. The library files with its program functions that enable offline training on MCUs are described in Section 3. In Section 3.3, we provide an example use case on how Edge2Train can be used to build an offline, self-learning HVAC control system that provides superior thermal comfort. Finally, in Section 4, we perform an end-to-end evaluation of Edge2Train.

3 FRAMEWORK IMPLEMENTATION

When users want their edge devices to learn from a new data pattern it sees after deployment, they just need to call the functions we provide in the C++ Edge2Train library. These functions fuse our framework with their existing IoT applications, enabling edge MCUs to re-train themselves offline, for continuous improvement.

3.1 File System of Edge2Train

The entire framework is implemented in *Fl1- Edge2Train*, a single .h file shown in Figure 2. This file contains five functions that setup an SVM, train the SVM, evaluate the trained SVM, and use real-time data to carry out inferencing using the trained SVM. As shown in Figure 2, the *Fl1* file uses operators from the *Fl2- all_ops_resolver* file to run the model and uses the *Fl3- micro_interpreter* file for initializing the interrupter to load the model and pass variables. The *Fl2* and *Fl3* files are imported from TensorFlow Micro library [1].

3.2 Edge2Train C++ Library

In this section, we provide and explain the algorithms of the five functions that constitute our C++ Edge2Train library. These functions are:

i Fn1- SVM Setup: Sets up the parameters that tune the SVM. These parameters are sent by the IoT application, to tune the SVM for the task specified for the edge device.
ii Fn2- SVM Train: This function uses the newly built training set to train SVMs offline on MCUs.
iii Fn3- SVM Infer: When sensor data is fed to this function, it calls the newly trained SVM, performs classifications, and sends the results to the IoT application.
iv Fn4- SVM Eval: Evaluate the classification accuracy of the MCU-trained SVM.
v Fn5- Kernel: Produce implicit dot products of two values. On MCUs, it is used during training SVMs and while inferring using trained SVMs.

First we provide and explain the *Fn2- SVM Train* function of Edge2Train that uses Algorithm 1 to train SVMs on MCUs. The novelty of this algorithm is that it optimizes the training method of SVMs to enable training on resource-constrained MCUs. This optimization also speeds up the algorithm on large training sets while ensuring that there is convergence even under degenerate conditions. We perform this optimized training of SVMs on MCUs by leveraging Sequential Minimal Optimization (SMO) [11] since it consumes minimal MCU resources while adapting and scaling efficiently to diverse real-world applications. In this work, we use an SVM to compute a linear classifier as shown in Eqn. 1. Since, we are training a binary classification SVM, we will classify $y = 1$ if $f(x) \geq 0$ and $y =$ -1 if $f(x) < 0$.

Algorithm 1 To train SVMs on MCUs using SMO.

Input:
C: Regularization parameter.
alphaTol: Numerical tolerance.
max passes: Max # of times to iterate over $\alpha's$.
(x(1), y(1)),..,(x(m), y(m)) is Training set (x_train & y_train).
Output:
$\alpha \in R^m$: Lagrange multipliers used during inference.
$b \in R$: Threshold.
Initialize $\alpha_i = 0, \forall i, b = 0, passes$.
function SVM Train (C, alphaTol, max passes, Training set)
while (passes < max passes) **do**
num_changed_alphas = 0.
for i = 1 to m **do**
Calculate Ei = $f(x_i)$ - y_i. Use Eqn. 1.
if (($y_i E_i < -alphaTol$ && $\alpha_i < C$) || ($y_i E_i > alphaTol$ && $\alpha_i > 0$)) **then**
Randomly select $j \neq i$.
Calculate $Ej = f(x_j) y_j$. Use Eqn. 1.
Save old $\alpha's$: $\alpha_i^{old} = \alpha_i$,$\alpha_j^{old} = \alpha_j$.
Compute L and H. Use Eqn. 2 or 3.
if (L == H) **then**
Continue to next i.
Compute η. Use Eqn. 4.
if (η >= 0) **then**
Continue to next i.
Compute and clip new value for α_j. Use Eqn. 5 and 6.
if ($\alpha_j \alpha_j^{old} < 105$) **then**
Continue to next i.
Determine value for α_i. Use Eqn. 7.
Compute b_1 and b_2. Use Eqn. 8 and 9.
Compute b. Use Eqn. 10.
num changed alphas = num changed alphas + 1.
end if
end for
if (num changed alphas == 0) **then**
passes = passes + 1.
else
passes = 0.
end while

$$f(x) = \sum_{i=1}^{m} \alpha_i y_i (x_i, x) + b \quad (1)$$

We find the bounds L and H using Eqn. 2 and Eqn. 3, such that $L \leq \alpha_j \leq H$ holds in order for α_j to satisfy $0 \leq \alpha_j \leq C$, where α_i and α_j are the Lagrange multipliers chosen to optimize the training process. To choose, we iterate over all α_i, where $i = 1, 2...m$. During this iteration, if α_i does not satisfy the Karush-Kuhn-Tucker (KKT) conditions, we select α_j randomly from the remaining $m - 1$ $\alpha's$ followed by joint optimization of α_i and α_j.

$$\text{If } y_i \neq y_j, L = max(0, \alpha_j - \alpha_i), H = min(C, C + \alpha_j - \alpha_i) \quad (2)$$

$$\text{If } y_i = y_j, L = max(0, \alpha_i + \alpha_j - C), H = min(C, \alpha_i + \alpha_j) \quad (3)$$

[1] https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/micro/

用SMO优化SVM算法，用二进制分类SVM

Algorithm 2 To infer using the MCU trained SVM.

Input:
x_train[][D]: Training set used to train SVM. D is feature dimen.
setsize: The size of test set.
y_train[]: Training set's true output.
x_test[D]: Test set input. Trained SVM will infer for this array.
var_alphas[i]: Lagrange multipliers obt using Algorithm 1.
alphaTol: Numerical tolerance received from *Fn1- SVM Setup*.
Output:
classifications: Inferences provided by MCU trained SVM.
function SVMInfer (x_train[][D], y_test, D, setsize, skip)
initialize classifications = 0.
for i = 0 to setsize
if ((!skip && var_alphas[i] != 0) || (skip && var_alphas[i] > alphaTol)) **then**
classifications += var_alphas[i] * y_train[i] * Kernel (x_test, x_train, D). Use Kernel fn from Algorithm 3.
return classifications.

The α_j is found using Eqn. 5, where the value of η and E_k is found out using Eqn. 4. If α_j is found outside the L and H bounds, it is clipped using the Eqn. 6. Using this α_j, the value of α_i is found out using Eqn. 7 where $\alpha_j^{(old)}$ is the value of α_j before optimization by Eqn. 5 and Eqn. 6.

$$\eta = 2(x_i, x_j) - (x_i, x_i) - (x_j, x_j) \text{ and } E_k = f(x_k) - y_k \quad (4)$$

When calculating η we use a kernel function K, from Algorithm 3 to obtain inner products for values.

$$\alpha_j = \alpha_j - \frac{y_i(E_i - E_j)}{\eta} \quad (5)$$

$$\alpha_j = \begin{cases} \text{H if } \alpha_j > H \\ \alpha_j \text{ if } L \leq \alpha_j \leq H \\ L \text{ if } \alpha_j < L \end{cases} \quad (6)$$

$$\alpha_i = \alpha_i + y_i y_j (\alpha_j^{(odd)} - \alpha_j) \quad (7)$$

After optimizing α_i and α_j, we select the threshold b such that the KKT conditions are satisfied for the i^{th} and j^{th} values. The KKT conditions here are used to check if the optimized values converge to the defined optimal point. If the optimized α_i is not at the bounds $0 < \alpha_i < C$, then Eqn. 8 is used to calculate the threshold, and the thus-found threshold is $b1$. This threshold $b1$ forces the SVM to output y_i when the input is x_i. Similarly, if $0 < \alpha_j < C$ then Eqn. 9 is used to calculate the threshold, the threshold found is $b2$.

$$\text{b1 = b -E}_i - y_i(\alpha_i - \alpha_i^{(odd)})(x_i, x_i) - y_j(\alpha_j - \alpha_j^{odd})(x_i, x_j) \quad (8)$$

$$\text{b2 = b -E}_j - y_i(\alpha_i - \alpha_i^{(odd)})(x_i, x_j) - y_j(\alpha_j - \alpha_j^{(odd)})(x_j, x_j) \quad (9)$$

In cases when both new $\alpha's$ are at the bounds, then all the thresholds between b1 and b2 satisfy the KKT conditions. So, in such cases, the threshold b is $(b1 + b2)/2$. Hence the complete equation for b is given as Eqn. 10.

$$b = \begin{cases} b_1 \text{ if } 0 < \alpha_i < C \\ b_2 \text{ if } 0 < \alpha_j < C \\ \frac{b_1 + b_2}{2} \text{ otherwise} \end{cases} \quad (10)$$

Algorithm 3 Kernel for SVM during training and inferring on MCUs.

Input:
x[], y[]: The values for which sum of the products is needed.
D: Feature dimension.
Output:
sum: Dot products of input.
function Kernel (x, y, D)
initialize sum = 0
for i = 0 to D **do**
sum += x[i] * y[i].
return sum.

We use a kernel function K whenever an inner product is required during training or inferring using SVMs on MCUs. We substitute a kernel $K(x(i), x)$ in place of the inner product. We provide this kernel function in Algorithm 3.

After training, users can next call the *Fn4- SVM Eval*. This function uses our Algorithm 4 to evaluate the classification accuracy of the MCU trained SVM. The accuracy here is the ratio of the number of correct classifications to the total number of input samples. To conserve the MCU's Flash memory (very limited), we do not use additional metrics such as the F1 score, Matthews Correlation Coefficient (MCC), confusion matrix, etc. The Algorithms 3 and 4 are the standard C++ implementations of the methods used in Deep Learning frameworks for computing dot products and accuracy. Finally, when users want to use their MCU trained SVM to perform onboard offline inference, they just need to call the Fn- SVM Infer and pass their sensor readings. This function uses our Algorithm 2 to infer for the input values.

Algorithm 4 To find SVM classification accuracy on MCUs.

Input:
x_test[][D]: Test set input. D is feature dimension.
y_test[]: Test set's true output.
setsize: The size of test set.
Output:
accuracy: Classification accuracy of MCU trained SVM.
function SVMEval (x_test[][D], y_test[], setsize)
Initialize correct, accuracy, inference = 0.
for i = 0 to setsize **do**
inference = infer for x_test[i]. Use Algorithm 2.
if (inference == y_test[i]) **then**
correct = correct + 1.
accuracy = 1.0 * correct / setsize.
return accuracy.

3.3 Fusing IoT applications with Edge2Train

In this section, we provide an example use-case on how Edge2Train can be used to build an offline, self-learning HVAC control system that provides superior thermal comfort. Here, we consider a typical smart building with a sensor-based HVAC control system to control its internal heating, ventilation, and air conditioning. During the design time of this HVAC control system, it is programmed

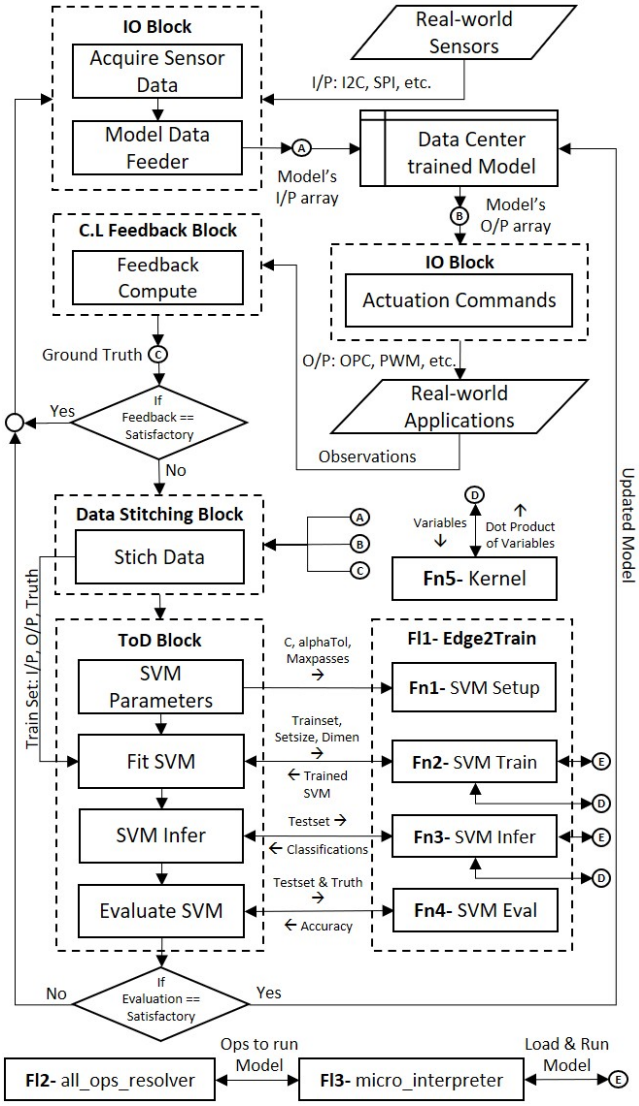

Figure 2: IoT application fused with functions of the Edge2Train.

with a standard HVAC control strategy. In many cases, such a standard/one-size-fits-all HVAC control strategy cannot work well to provide thermal comfort for many people because every building/infrastructure has differences (e.g. location, size of a building, its thermal confinement, etc.). Due to such multiple differences, every building needs a separate HVAC control strategy to provide the desired thermal comfort to different people in their respective buildings. In this section, using Figure 2, we provide and explain a method to fuse the IoT application of an HVAC control use case with our Edge2Train. The fusing starts with the Closed-Loop Feedback Block of Edge2Train. This block monitors the application routine of the HVAC control system, to compute feedback for each actuation. If this feedback is satisfactory, the HVAC routine is not disturbed. Else, a training set is generated by the Data Stitching Block as shown in Figure 2. When sufficient training sets are generated, then the Train on Demand (ToD) block trains a model leveraging

Table 1: Specifications of boards chosen to evaluate Edge2Train.

Board	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 #4	Adafruit HUZZAH32, Generic ESP32	32	520kB	4MB	240	✓
#5	ATSAMD21G18 Adafruit METRO	32	32kB	256kB	48	✗

the *Fl1: Edge2Train* file. As explained in Section. 3.2, this file contains functions that setup, train, evaluate, and also use real-time data to inference using trained SVM. To summarise, when the IoT application keeps updating its model by training using the generated training sets, the edge devices can learn the best strategy to perform tailored control of the HVAC system for any building types.

4 EVALUATION

In this section, we focus on the end-to-end evaluation of the framework using five MCU-based boards given in Table 1. This evaluation answers:

- Can Edge2Train be used to train models on a wide variety of MCU based development boards?
- Can models be trained on MCUs with limited Flash and SRAM space?
- What is the relationship between a feature vector's dimension, training set size, and training time?
- How much time and energy is consumed to train models and infer using this trained model on MCUs?

4.1 Datasets and Evaluation Procedure

We selected two datasets for which on-board SVM training and evaluation were performed for the selected devices. From the first Iris flower [2] dataset, we extracted 100 data fields (feature dimension of 4), with 50 positive and 50 negative samples of Iris Setosa. We use such small samples since the MCU boards usually lack EEPROM (space and cost cutting) or any additional memory units (SD card modules or external Flash) to store the data fields. Then we trained a binary classification SVM that distinguishes Iris Setosa from other flowers based on the input features. The second round of evaluations was performed using the MNIST database of handwritten digits [3]. Here we extracted 120 data fields (feature dimension of 64) for digit 6 (we chose digit 6 to ensure there would be complications rather than using 0 or 1), with 60 positive and 60 negative samples. Then we trained a binary classification SVM that distinguishes digit 6 from other digits, based on the input features. For evaluation, in addition to the MCU boards from Table 1, a standard Ubuntu laptop with Intel (R) Core (TM) i7-5500 CPU @ 2.40 GHz was selected as CPU1 and the CPU2 is a Windows laptop with Intel (R) Core

[2] https://archive.ics.uci.edu/ml/datasets/iris
[3] http://yann.lecun.com/exdb/mnist/

Table 2: On-board SVM training using Edge2Train: Time and energy consumed to train and infer on MCUs (uses Edge2Train) and CPUs (uses Python scikit-learn). Flash and SRAM consumed by Edge2Train for various MCU boards.

Board #	Datasets	Training & Unit Inference Time (ms)			Accuracy (%)			Training & Unit Inference Energy (mJ)			Flash & SRAM Requirement (kB)
		CPU1	CPU2	MCU	CPU1	CPU2	MCU	CPU1	CPU2	MCU	MCU
1	Iris Flowers Features Dim=4 Training set size=100			33512.0, 0.067 45744.0, 0.5			90.0 92.85			5864.6, 0.011 8005.2, 0.087	43.889, 12.392 65.488, 76.632
2				226190.0, 0.4 -			83.34 -			39583.2, 0.078 -	30.112, 9.880 53.008 overflow
3	Handwritten Digits Features Dim=64 Training set size=120	1.193, 0.006 2.516, 0.341	1.738, 0.004 2.671, 0.366	10187.0, 0.031 25326.0, 0.1	93.37 91.66	96.67 88.89	86.67 86.12	21.474, 0.108 45.288, 6.308	32.153, 0.074 49.413, 6.771	1782.7, 0.005 4432.0, 0.017	219.165, 21.012 240.545, 85.252
4				9037.0, 0.034 18754.0, 0.1			90.0 83.34			1581.4, 0.005 3281.9, 0.017	219.165, 21.012 240.545, 85.252
5				785607.0, 1.3 -			90.0 -			137481.2, 0.22 -	29.31, 10.650 39.896 overflow

(TM) i7-8650U CPU @ 1.90 GHz. The same SVM parameters, C (regularization parameter) = 1, alphaTol (numerical tolerance) = 1e-7, maxIter = 10000 (max # of times to iterate without changing), linear kernel and same datasets were used across all the MCUs and CPUs. We also compare the time and energy consumed to train and inference on MCUs that use our Edge2Train with CPUs that use Python scikit-learn to perform the same tasks. We calculated the energy (in Joules) consumed by devices during training and inferring by multiplying the Current (Amperes) rating of MCUs with its Potential/Voltage (Volts) and task time (seconds). For CPUs, we used the htop process viewer and powerstat tool to calculate energy.

4.2 Training and Executing SVMs on MCUs

We uploaded the dataset and our Edge2Train-fused IoT application on all five MCUs from Table 1 using the Arduino IDE. Then we powered on each board, connected them to a PC via the serial port to receive training time and classification accuracy from MCUs. For both of the selected datasets, the first 70% of data was used to train the binary classification SVM, the remaining 30% data was used for evaluation. When we instruct the board to train, the SVM Setup and SVM Train functions as described in Section 3.2 uses the Algorithms 1 and 3 to train on the loaded datasets. Next, the SVM Infer and SVM Eval functions use our Algorithms 2 and 4 to inference using the test sets and to evaluate (find accuracy of) the trained SVMs. We tabulated the obtained results in Table 2. From this, it is apparent that developers can leverage the functions of Edge2Train for training models offline using real-time data from any of their use cases on such small MCU boards. We also estimate that using Edge2Train, on-board model 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.

From Table 2, the highest classification accuracy is 96.67% for the Iris dataset on CPUs and 90.0% on MCUs. For the MNIST digits dataset, 92.85% on MCUs and 91.66% on CPUs were the highest accuracies. Although the training time on MCUs was higher than CPUs, our Edge2Train-trained SVMs produce classification accuracies close to those of Python scikit-learn trained SVMs.

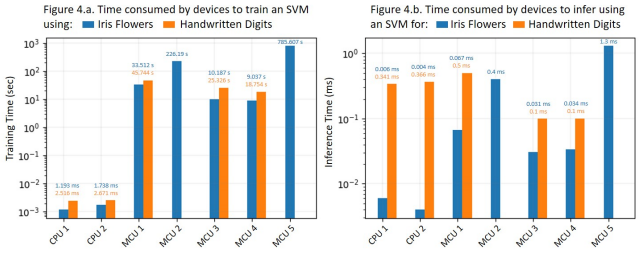

Figure 3: Comparing training and inference time of SVMs trained using Edge2Train on MCUs vs Python scikit-learn trained SVMs on CPUs.

4.3 Training and Inference Time

In this section, we compare the training and inference time of SVMs on MCUs (uses Edge2Train) with standard CPUs (uses Python scikit-learn). The CPUs we chose for comparison have approximately 1000x times better specifications over the chosen MCUs. In Figure 3 (y-axis in Log scale) we plotted the time consumed by devices to train and inference for both datasets. We noticed that CPU1 and MCU4 are the fastest in their respective classes. Although CPU1 has 1000x times better spec, it is only 7.57 seconds faster for Iris, and 7.45 seconds for the MNIST digits datasets when comparing the CPU1 versus MCU4 training times from Figure 3. Although CPUs are faster, they cannot be used as IoT edge devices due to their cost (CPU1 is 200x times more expensive than MCU4), form factor (5x times more area), and energy consumption (7x times). Since billions of edge devices are MCU-based, it is feasible to train even at lesser speeds. Such offline training using Edge2Train 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. 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 the model accuracy. Whereas, when comparing the inference time using Figure 3, MCUs 3 and 4 that used our Edge2Train performed unit inference for the digits data (64-dimensional features) approximately 3.5x times faster than CPUs.

Next, to explain the relationship between training time, training set size, and feature dimension, using Edge2Train we trained SVMs on all five MCU boards by providing training sets of various sizes.

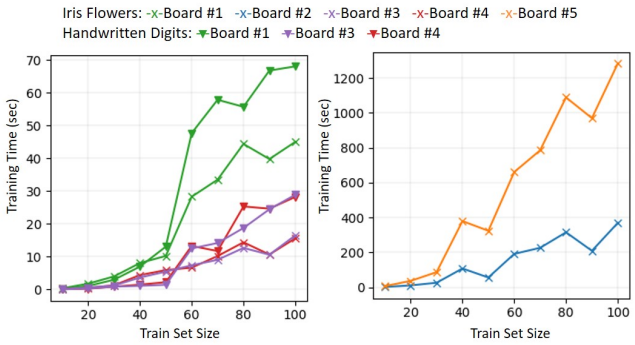

Figure 4: Training SVMs on MCUs using Edge2Train: Comparing training set size vs training time for selected datasets.

The results are shown in Figure 4. Here we noticed that the training time grows swiftly with the number of training samples. For the Iris dataset with 4-dimensional features, MCU4 only took 15.58 seconds to train on 100 samples, whereas, it took 28.24 seconds to train on the MNIST digits dataset with 64-dimensional features. Board 5 is the slowest since it only has a 48 MHz clock and does not have floating-point support. Hence it took 21.39 minutes to train on 100 samples of the Iris dataset (82x times slower than MCU4). In such cases, to reduce training time and also to save SRAM, the precision of the data should be reduced. In Figure 4, sudden peaks are observed when the algorithm consumes more time to find the optimal hyperplane, and trough points are observed when the optimal hyperplane is found soon. Hence the on-board training time does not strictly increase linearly with training set size.

4.4 SRAM Requirement

The run-time variables generated during training need to be stored in the SRAM of MCUs. This SRAM space in MCUs is restricted, since adding more leads to higher power leakage and manufacturing costs. From Table 1, the popular open-source boards we chose have only 20 kB to a max of 520 kB of SRAM which restricts training models using high feature dimensions and large training set sizes in a single run. When compiling Edge2Train with IoT application for boards, the memory requirements for target boards are calculated by the compiler. We provide the calculated FLASH and SRAM requirements for all boards in Table 2. For board 1, Iris dataset, our Edge2Train fused IoT application and dataset in total uses only 4.38% of FLASH and 4.84% SRAM. For the digits dataset, the same board one requires 6.54% and 29.93%. Board 2, for Iris, uses 23.52% and 49.4%. Here, board 2 cannot train using the digits dataset because SRAM overflowed by 53.008 kB. Similarly for board 5, SRAM overflowed by 39.896% for the digits. Boards 3 and 4 have the highest memory resources. They have 83.61% free SRAM space even after loading it with the digits, a 64 dimension features dataset. In such cases, we can train using a larger training set and also implement polynomial kernel-based SVMs. From Table 2, we noticed that when the input feature dimensions increase from 4 to 64, for board 1 the SRAM requirement increase 6x times and FLASH by 1.4x times. Similarly, 4x times of SRAM and 1x time of FLASH for boards 3 and 4. It is apparent from Table 2 that Edge2Train can

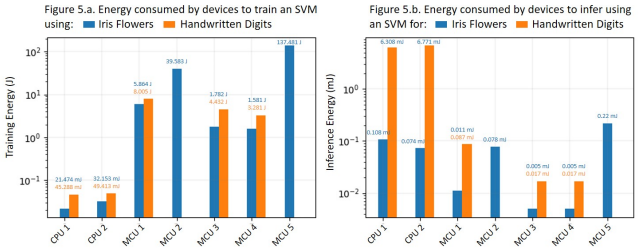

Figure 5: Comparing energy consumed to train and infer on MCUs using Edge2Train vs Python scikit-learn on CPUs.

train models on MCUs with limited Flash and SRAM space. But when the feature dimensions of datasets are large, then the SVMs should be trained by providing data in batches.

4.5 Energy Efficiency

In this section, using Figure 5 the energy consumed by MCU boards to train and inference using Edge2Train is compared with standard CPUs that use Python scikit-learn to perform the same tasks. The values used for the plot were calculated by applying the method described in Section 4.1 on the values from Table 2. CPU1 and MCU4 consumed the least energy to train. From Figure 5 (y-axis in Log scale) we observed that CPU1 consumed 2x times more energy to train on the digits dataset (64 dimension features) than to train on the iris dataset (4 dimension features). Similarly, MCU4 also consumed 2x times more energy to train on the digits dataset. To perform a unit inference using 64 dimension digits data, the CPU1 consumed 58x more energy than to inference for a 4-dimension input, whereas MCU4 only consumed 3.4x times more energy to infer for digits data. On the whole, MCUs consume approximately 20x times less energy for the iris data and 350x times less energy for digits data than CPUs did to perform a unit inference. Using our Edge2Train, MCUs can perform inference at the lowest power costs and also can perform complete offline training and inference. This increases the operating time of battery-powered devices while also eliminating the need for a wireless chipset that transmits data to the cloud for training or inference.

5 RELATED WORK

There are only a few articles in the domain of enabling on-the-fly offline training on the resource-constrained MCUs at the edge. Also, since this is a yet to emerge area of research there are no existing frameworks like Tensorflow, Keras, etc. for enabling training models on MCUs. We are grouping the related research into three sets.

In the first set, we present efforts that aim to push analytics to the edge platform. In the article [7], a decentralized stochastic gradient descent method to solve a large linear regression problem on the edge network was proposed and applied to the seismic imaging use-case. In [13], authors present a framework for coordinated processing between edge and cloud. They utilize network-wide knowledge and historical information from the cloud servers to guide edge nodes for improving the performance of heterogeneous wireless IoT networks. To minimize the communication overhead and to maximize the quality of analytics results, the article [5] provides a quality-aware, time optimized edge analytics model that enables

edge devices to intelligently decide when and what data to process and deliver to the above fog and cloud layers. In [10], to improve the data security without much impact on throughput, authors first process the large streams of IoT data using their edge stream analytics engine, which is a Trusted Execution Environment (TEE), then stream the data to the cloud. Articles [5, 7, 10, 13] provides edge nodes executable models/algorithms to achieve configuration adaptation, privacy-preservation, useful features extraction from heterogeneous data, balanced computation, robustness towards node/link failure, etc. Although such models solve real-life problems, when they get exposed to unseen data patterns, they will not know how to react to that specific scenario or produce results with degraded accuracy. Also, such models do not continuously improve themselves, by learning from the fresh real-world data patterns they see after their deployment.

The second set of articles proposes compression techniques to reduce the size of the model's weights using quantization and weight pruning techniques resulting in a reduction of both model size and inference time without considerable accuracy loss. CONDENSA [6], 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 [12]. In both [6], [12] and other similar articles proposing compressing [3, 4] and optimization [1, 8] the model is trained in high resource environments, followed by using multi-stage MCU-aware optimisation to enabling deploying models at edge. Whereas in our Edge2Train, we provided a c++ library with multiple functions that developers can leverage to train models offline using real-time data from any of their use-case.

In the third set of related articles, authors in [9] have implemented a tree-based algorithm for efficient prediction in milliseconds even on slow MCUs with only kBs of memory. Similarly, ProtoNN, k-Nearest Neighbor (KNN) inspired algorithm with several orders lower storage and prediction complexity was proposed in [2] to address the problem of real-time and accurate prediction on resource-scarce devices. Articles from this set provided algorithms that tailor models to fit in MCUs. Such methods and are again not generic across all MCUs, and also do not enable edge devices to learn from the new data pattern it sees after deployment.

6 CONCLUSION

We presented Edge2Train, a framework that enables a wide variety of open-source MCUs that have limited Flash, SRAM, and no floating-point support to re-train themselves locally and offline. Thus, MCU-based edge devices can continuously improve themselves for better analytics results by managing to understand continuously evolving real-world data on the fly. We also provided algorithms of Edge2Train with its C++ implementation, which enables users to train models offline on MCUs using live data from their IoT use-cases, and also enables performing on-board inference and model evaluations. In future work, we plan to provide algorithms and functions to enable users to perform, on-board, offline multi-class SVM training and inference.

7 ACKNOWLEDGEMENTS

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

REFERENCES

[1] Sourav Bhattacharya and Nicholas D. Lane. 2016. 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, SenSys 2016, Stanford, CA, USA, November 14-16, 2016*. ACM, 176–189. https://doi.org/10.1145/2994551.2994564

[2] Chirag Gupta, Arun Sai Suggala, Ankit Goyal, Harsha Vardhan Simhadri, Bhargavi Paranjape, Ashish Kumar, Saurabh Goyal, Raghavendra Udupa, Manik Varma, and Prateek Jain. 2017. ProtoNN: Compressed and Accurate kNN for Resource-scarce Devices. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017 (Proceedings of Machine Learning Research, Vol. 70)*. PMLR, 1331–1340. http://proceedings.mlr.press/v70/gupta17a.html

[3] Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. 2016. Hardware-oriented Approximation of Convolutional Neural Networks. *CoRR* abs/1604.03168 (2016). arXiv:1604.03168 http://arxiv.org/abs/1604.03168

[4] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[5] Natascha Harth and Christos Anagnostopoulos. 2017. Quality-aware aggregation & predictive analytics at the edge. In *2017 IEEE International Conference on Big Data, BigData 2017, Boston, MA, USA, December 11-14, 2017*. IEEE Computer Society, 17–26. https://doi.org/10.1109/BigData.2017.8257907

[6] Vinu Joseph, Saurav Muralidharan, Animesh Garg, Michael Garland, and Ganesh Gopalakrishnan. 2019. A Programmable Approach to Model Compression. *CoRR* abs/1911.02497 (2019). arXiv:1911.02497 http://arxiv.org/abs/1911.02497

[7] Goutham Kamath, Pavan Agnihotri, Maria Valero, Krishanu Sarker, and Wen-Zhan Song. 2016. Pushing Analytics to the Edge. In *2016 IEEE Global Communications Conference, GLOBECOM 2016, Washington, DC, USA, December 4-8, 2016*. IEEE, 1–6. https://doi.org/10.1109/GLOCOM.2016.7842181

[8] Ashish Kumar, Saurabh Goyal, and Manik Varma. 2017. 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 (Proceedings of Machine Learning Research, Vol. 70)*. PMLR, 1935–1944. http://proceedings.mlr.press/v70/kumar17a.html

[9] Ashish Kumar, Saurabh Goyal, and Manik Varma. 2017. 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 (Proceedings of Machine Learning Research, Vol. 70)*. PMLR, 1935–1944. http://proceedings.mlr.press/v70/kumar17a.html

[10] Heejin Park, Shuang Zhai, Long Lu, and Felix Xiaozhu Lin. 2019. StreamBox-TZ: Secure Stream Analytics at the Edge with TrustZone. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*. USENIX Association, 537–554. https://www.usenix.org/conference/atc19/presentation/park-heejin

[11] Fernando Pérez-Cruz, Pedro Luis Alarcón-Diana, Ángel Navia-Vázquez, and Antonio Artés-Rodríguez. 2000. Fast Training of Support Vector Classifiers. In *Advances in Neural Information Processing Systems 13, Papers from Neural Information Processing Systems (NIPS) 2000, Denver, CO, USA*. MIT Press, 734–740. http://papers.nips.cc/paper/1855-fast-training-of-support-vector-classifiers

[12] Antonio Polino, Razvan Pascanu, and Dan Alistarh. 2018. Model compression via distillation and quantization. *CoRR* abs/1802.05668 (2018). arXiv:1802.05668 http://arxiv.org/abs/1802.05668

[13] Shree Krishna Sharma and Xianbin Wang. 2017. Live Data Analytics With Collaborative Edge and Cloud Processing in Wireless IoT Networks. *IEEE Access* 5 (2017), 4621–4635. https://doi.org/10.1109/ACCESS.2017.2682640

[14] Bharath Sudharsan and Manigandan Chockalingam. 2019. A Microphone Array and Voice Algorithm based Smart Hearing Aid. *CoRR* abs/1908.07324 (2019). arXiv:1908.07324 http://arxiv.org/abs/1908.07324

[15] Bharath Sudharsan, Sree Prem Kumar, and Rakesh Dhakshinamurthy. 2019. AI Vision: Smart speaker design and implementation with object detection custom skill and advanced voice interaction capability. In *2019 11th International Conference on Advanced Computing (ICoAC)*. IEEE, 97–102.