An SRAM optimized approach for constant memory consumption and ultra-fast execution of ML classifiers on TinyML hardware

Title	An SRAM optimized approach for constant memory consumption and ultra-fast execution of ML classifiers on TinyML hardware
Author(s)	Sudharsan, Bharath;Yadav, Piyush;Breslin, John G.;Ali, Muhammad Intizar
Publication Date	2021-11-15
Publisher	Institute of Electrical and Electronics Engineers
Repository DOI	10.1109/SCC53864.2021.00045

An SRAM Optimized Approach for Constant Memory Consumption and Ultra-fast Execution of ML Classifiers on TinyML Hardware

Bharath Sudharsan*, Piyush Yadav**, John G. Breslin*, Muhammad Intizar Ali§
*Confirm SFI 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*—With the introduction of ultra-low-power machine learning (TinyML), IoT devices are becoming smarter as they are driven by Machine Learning (ML) models. However, any increase in the training data results in a linear increase in the space complexity of the ML models. It is highly challenging to deploy such ML models on IoT devices with limited memory (TinyML hardware). To alleviate such memory issues, in this paper, we present an SRAM-optimized classifier porting, stitching, and efficient deployment approach. The proposed method enables large classifiers to be comfortably executed on microcontroller unit (MCU) based IoT devices and perform ultra-fast classifications while consuming 0 bytes of SRAM. We tested our SRAM optimized approach by utilizing it to port and execute 7 dataset-trained classifiers on 7 popular MCU boards, and report their inference time and memory (Flash and SRAM) consumption. It is apparent from the experimental results that; (i) the classifiers ported using our proposed approach are of varied sizes but have constant SRAM consumption. Thus, the approach enabled the deployment of larger ML classifier models even on tiny Atmega328P MCU-based Arduino Nano, which has only 8 kB SRAM; (ii) even the resource-constrained 8-bit MCUs performed faster unit inference (in less than a millisecond) than a NVIDIA Jetson Nano GPU and Raspberry Pi 4 CPU; (iii) the majority of models produced 1-4x times faster inference results in comparison with the models ported by the sklearn-porter, m2cgen, and emlearn libraries.

***Index Terms*—IoT Devices, TinyML, Microcontrollers, Offline Inference, SRAM Optimization, Classifiers Porting.**

I. Introduction

The majority of IoT devices like smart plugs, Heating Ventilation Air Conditioning (HVAC) controllers, IoT thermostats, etc. are powered by MCUs and small CPUs that are resource-constrained [1]. Such MCUs lack multiple cores, parallel execution units, no hardware support for floating-point operations (FLOPS), and low clock speed [2]. Still, for decades, the hardware of IoT devices are designed using such resource-constrained MCUs because; (i) MCUs are tiny in form factor as its memory units (Flash, SRAM) and processor unit are contained in a single chip; (ii) MCUs are highly power-efficient and cheaper than the standard laptop CPUs and mobile phone processors. For example, the Arduino Nano is an 8-bit ATmega328 MCU with a 16 MHz clock, 2 kB of SRAM, 32 kB of ISP flash memory. Similarly, the NUCLEO-F303K8 is a 32-bit ARM Cortex-M4 MCU with a 72 MHz clock and 64 kB of flash memory. These two MCU-boards are popular examples of TinyML hardware that are widely used to design IoT devices, and billions of similar specification hardware-based IoT devices have already been deployed in the world [3].

During the design phase of IoT devices, to conserve energy and to maintain high instruction execution speeds, no secondary/backing memory is added. For example, adding a high-capacity SD card or EEPROM can enable storing large Decision Trees (DT) and Random Forest (RF) models even without optimization. But such an approach will highly affect the model execution speed since the memory outside the chipset is slow. It also requires $\approx$ 100x more energy to read the thousands of outside located model parameters. Due to this un-addressable memory constraint, the vast majority of MCU-powered IoT edge devices use simplified versions of DTs and RFs to solve ranking, regression, and classification problems offline at the device level [4]. Currently, edge devices cannot handle complex tree-based ML models with a large number of tree nodes because they are resource-intensive and often cannot fit within the memory of MCUs, resulting in memory overflow issues.

Any increase in training samples increases the model size and inference complexity of the widely used stable scikit-learn classifiers [5, 6]. Multiple studies [7, 8] have shown that tree-based algorithms can only be deployed on embedded sensor systems or tiny IoT devices after reducing inference complexity and model size. To comfortably fit within the specific hardware architecture, either the DTs and RFs are pruned [9, 10], or node parameters in the DTs are shared using a directed acyclic graph [11]. Sometimes users design sparse and shallow tree learners that only require a few kBs of memory [2] to keep a low memory footprint. Such methods of learning shallow trees or aggressive pruning to fit within a few kBs often led to degradation in accuracy. This is due to the approximation of non-linear and complex decision boundaries using a small number of axis-aligned hyperplanes. Other studies [12, 13] have proposed optimization methods, where the models are trained in high resource setups, then a multi-stage MCU-aware

2474-2473/21/$31.00 ©2021 IEEE
DOI 10.1109/SCC53864.2021.00045

optimization (tailored) is performed before deployment.

In contrast to the above-mentioned approaches, in this paper, we present an SRAM optimized approach. The proposed approach does not reduce the ML algorithm complexity since doing so results in highly engineered models that need special consideration and optimization for different datasets and IoT scenarios, which is not practically feasible. The main contributions of the SRAM optimized approach are as follows:

- The proposed method is generic and can efficiently port and execute a wide variety of DT and RF classifiers on different resource-constrained MCUs and small CPUs-based IoT devices with 0 bytes of SRAM consumption.
- The models ported and executed using the proposed method produce ultra-fast classification results on MCUs (1-4x times faster than state-of-the-art libraries). Thus, even the autonomous tiny IoT devices can efficiently control real-world IoT applications by making timely predictions/decisions.
- Despite the reduced memory footprint and ultra-fast classifications, our approach guarantees the same level of performance (accuracy, F1 score, etc.) as its original models (before porting) that were trained and tested on high-resource lab setups.

The rest of the paper is structured as follows: Section II - III briefs the essential concepts and related studies. Section IV presents the SRAM optimized approach, and Section V performs an extensive experimental evaluation that aims to justify claims of the SRAM optimized approach. Section VI, concludes by providing a context for future research.

II. Background

Recent advancements in the field of ultra-low-power machine learning (TinyML) promises to unlock an entirely new class of edge applications [1]. The TinyML segment of libraries, tools, and frameworks is composed of two main elements, i.e., the converter and the interpreter. The converter runs on a high-resource machine and ports the trained model to optimized code that can execute on constrained platforms. The interpreter runs on the target TinyML hardware and executes code of the ML models generated by the converter. Unlike embedded Linux (like Raspberry Pi, BeagleBone families), for MCUs, the generated optimized code is in C++ 11, which requires 32-bit processors (ARM Cortex-M) for reasonable onboard performance. In the upcoming subsections, we present the state-of-the-art that can ease the implementation of ML algorithms on resource-constrained TinyML hardware (embedded Linux, MCUs, small CPUs).

A. TinyML: Libraries, Tools, and Frameworks

GO programming language (TinyGo) is related to TensorFlow Lite for Microcontrollers (TFLM) and Google's contribution to TinyML [14]. Microsoft's contribution in this scene is Embedded Learning Library (ELL) [15], which permits to design and deploy pre-trained ML models on ARM Cortex-A and Cortex-M architectures. ELL's API can be employed from C++/Python to utilize the pre-trained Neural Network (NN) models produced by Darknet [16], Cognitive Toolkit (CNTK) [17], or Open Neural Network Exchange (ONNX) format [18].

ARM has integrated ML on their product line that can be leveraged by Artificial Intelligence of Things (AIoT) researchers/developers to deliver advanced AI use cases/solutions to customers. The ARM-NN toolkit [19] allows the integration of NN workloads with cutting-edge hardware such as ARM Cortex-A CPUs, Mali GPUs, and Ethos Neural Processing Units (NPUs). ARM-NN is open-source and compatible with TensorFlow, Caffe, and ONNX format. To provide support also for the Cortex-M, ARM provides the Cortex Microcontroller Software Interface Standard-NN (CMSIS-NN) [20], a collection of NN kernels optimized for low hardware specification Cortex-M processor series.

STMicroelectronics (STM) delivers intelligent, energy-efficient products and solutions that power the electronics at the heart of everyday life. STM's X-CUBE-AI toolkit [21] can integrate pre-trained NNs with STM32 ARM Cortex-M MCUs by generating STM32-compatible C code from NN models generated by Keras, TensorFlow, or standard ONNX format. The interesting feature of this toolkit is, it enables running large NNs on TinyML hardware by storing weights and activation buffers in external flash memory and RAM respectively (related to the concept of the proposed SRAM optimized approach for ML classifiers).

B. TinyML: Training ML models on MCUs

Besides open-source contributions by tech giants, few institutions and companies have released licensed products. 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 form of Dynamic Link Library (DLL). In contrast to the frameworks reviewed in above-section, and similar to ML-MCU [22], Edge2Train [23] and TinyOL [24], AIfES permits to implement ML model training process on the embedded devices. Cartesiam NanoEdge AI Studio [25] 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 training on MCUs.

C. TinyML: Non Neural Networks

Surprisingly, the majority of modern frameworks focus only on NNs. However, different works released by TinyML enthusiasts, researchers, and developers consider other ML algorithms such as DTs, Naive Bayes classifier, k-Nearest Neighbors (k-NN). For example, to extend the compatibility of TensorFlow models, uTensor [26] produces C code for Mbed boards. Weka-porter [27] is a basic tool (limited features) to covert DTs generated by WEKA into C, Java, and JavaScript codes. EmbML [28] library can convert models trained by Scikit-learn or WEKA into C++ source-code files that can

be compiled and executed in constrained TinyML hardware platforms.

Sklearn-porter [29] transpiles various trained estimators to Java, C, JavaScript, GO, Ruby, and PHP. Given this variety of supported languages, sklearn-porter is a very complete framework as it is also compatible with a range of ML algorithms. Specifically for C language, it can also port Scikit-learn models of SVM, AdaBoost classifier, DTs, RFs. However, the versatility of this library hinders it to generate MCU-optimized code in terms of required RAM. m2cgen [30] is a similar tool that also transpiles Scikit-learn models into a native code. In this case, both the number of compatible algorithms as well as the target programming languages are even greater than those supported by sklearn-porter. Similarly, the emlearn [31] produces portable C99 code from models trained in Scikit-learn or Keras Python libraries. It is compatible with generated models of a range of datasets and ML algorithms. Also, emlearn has been tested on various chipsets like AVR Atmega, ESP8266.

III. Related Work

Here, we outline the concept of the selected classifiers, then compare our approach with the studies that achieve top optimization levels.

A. Optimizing Decision Trees

Like other supervised ML classifiers, DTs can also understand data, perform inference, and can be used for ranking, regression, and classification problems commonly found in IoT settings. When the DTs have a large number of tree nodes, their memory footprint is high and cannot fit within the MCU's memory. Many studies [7, 8] show that such algorithms can be implemented on embedded sensor systems or portable IoT devices only when they are optimized to fit the specific hardware architecture. Currently, to reduce its complexity, constraints are added to the hypothesis class (set of considered possible classification functions), and structures in the hypothesis class are discovered to generate simpler (resource-friendly) hypotheses. Commonly, for achieving reduced memory footprint and to avoid over-fitting, DTs are pruned [10], sparse and shallow tree learners that only require a few KBs of memory are designed [2], and node parameters are shared using a Directed acyclic graph (DAG) [11].

B. Optimizing Random Forests

RFs is based on the concept of *the wisdom of the crowd*, where many DTs are combined in a voting scheme. Here, probably the true predicted class is the class that receives majority votes from the trees. Like RFs, XGBoost is also an ensemble-based algorithm, where a number of trees are chained and each tree learns from the previous errors. For any given m training samples, implementation of the widely used stable DTs [5] has $O(\log(m))$ as its inference complexity and $O(m)$ as its model size. Similarly the stable RFs [6] has $O\left(N_{\text{tree}} \log(m)\right)$ inference complexity and $O\left(N_{\text{tree}} m\right)$ model size. This clearly shows that their complexity grows with increased training samples. For such ensemble-based algorithms, if the critical question of how to select weak learners while preserving the accuracy is answered, it can result in significant computational cost savings. To answer this, [32] presents a method that can automatically trade computation time with accuracy by using the extra time to select from a set of weak learners. In [9], RFs were pruned to obtain predictions within a limited hardware resource budget.

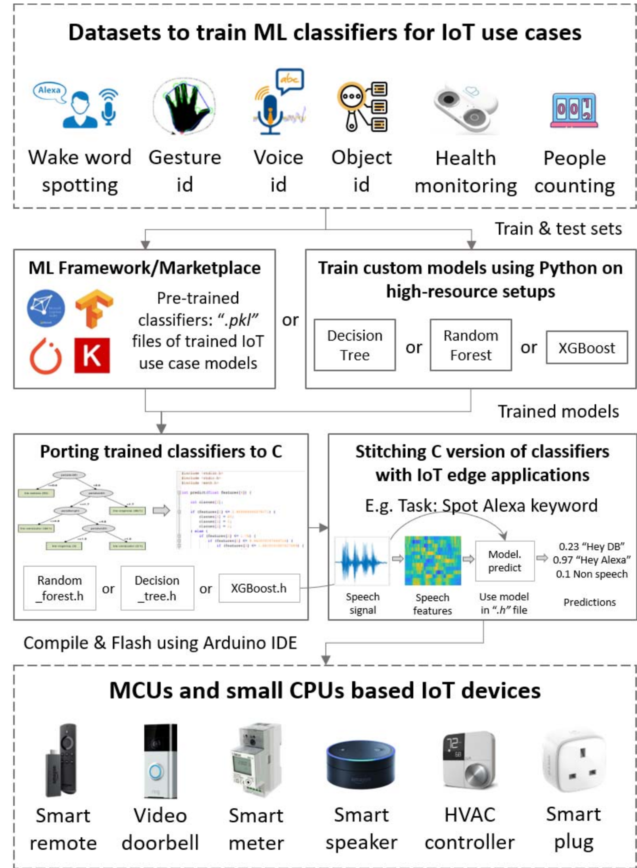

Fig. 1. Design flow to port, stitch, and execute ML classifiers on resource-constrained microcontroller-based IoT devices (TinyML hardware).

As opposed to the family of Neural Networks (NNs), there are only a few resource profiling results reported for the on-device deployment of DTs, RF, XG Boost algorithms [33]. We observed that the review papers and papers with advancement algorithms compare only the accuracy, training & inference complexity of one's method with the previous related ones. For example, in [2], energy and inference time were profiled for their tree-based Bonsai and compared with related techniques such as local deep kernel learning, gradient boosted decision tree, etc. on the Arduino Uno MCU board. However, the memory footprint during onboard execution was not profiled. Similarly, other related works empirically analyze the complexity and performance of ML algorithms [34, 35] on PCs but not on resource-constrained MCU-based IoT devices.

Algorithm 1 SRAM optimized porting of ML classifiers to plain C: runs on a high-resource machine.

```
1:  from sklearn import tree    ▷ can also use from sklearn.ensemble import RandomForestClassifier
2:      import packages: time, from sklearn.metrics import classification_report, train_test_split
3:      dataset ← MNIST Digits    ▷ load required dataset e.g. Iris, Breast Cancer, Titanic
4:      train_features, test_features, train_types, test_types ← train_test_split (dataset) ▷ splits into random train, test subsets
5:          train    ▷ function for ML algorithm training using loaded dataset
6:              clf_DT ← tree.DecisionTreeClassifier ()    ▷ define Decision Tree (DT) classifier
7:              clf_DT ← clf_DT.fit (train_features, train_types)    ▷ trained DT classifier
8:          evaluate    ▷ function to evaluate trained model
9:              predictions ← clf_DT.predict (test_features)
10:             inference time ← use clf_DT.predict () inside time.time ()    ▷ pass data for 1 sample and 100 samples
11:             evaluation ← use module classification_report (test_types, predictions) ▷ f1-score, support, macro avg, accuracy,
    etc. are reported
12:         if evaluation satisfactory then
13:             port    ▷ function that performs SRAM optimized porting of model to C
14:                 for i in node_count do    ▷ node i = 0 is the tree's root, node_count is total number of nodes
15:                     children_left [i], children_right [i], feature [i], threshold [i], classes [i] ← clf_DT ▷ extract entire tree
    structure from trained DT (clf_DT) and store in numerous parallel arrays
16:                 DT_MNIST.h ← clf_DT ported to C    ▷ using f.write write extracted information into .h file
17:         else
18:             re-tune DT parameters (e.g. criterion, splitter, max_depth, etc), then use functions train, evaluate, and port
```

In this paper, like previous works, we do not aim to reduce the ML algorithm complexity since it results in highly engineered models that need special consideration and optimization for different datasets and IoT scenarios, which is not practically feasible. Since both RFs and XGBoost depend on trees, we propose to efficiently port DTs to its C version that in turn results in optimizing many tree-based methods like RFs and XGBoost. During profiling, the standard high-quality DTs, RFs, and XGBoost multi-class classifiers, when ported to C and deployed using our method, consume 0 bytes of RAM when executing on MCUs, thereby clearly superior to the above and other highly engineered methods.

C. Resource-efficient Model Inference on MCUs

A set of articles propose compression techniques to reduce the size of the model's weights using quantization and pruning. Condensa [36], is a system for users to compose simple operators to build complex model compression strategies. In [37], two new compression methods jointly leverage weight quantization and the distillation of larger networks were proposed. ProtoNN [38] is a kNN inspired algorithm with several orders lower storage and prediction complexity that addresses the problem of real-time and accurate prediction on resource-scarce devices. In both [36, 37] and other similar articles proposing compressing [39, 40] and optimization [2, 41] methods, the models are trained in high resource setups, then a multi-stage MCU-aware optimization (tailored) is performed before deployment.

Different types of NNs are optimized and implemented on ARM Cortex-M processors. CMix-NN [42] is an open-source library for deploying mixed-precision NNs on MCUs (supports convolutional kernels with 2, 4, or 8 bits precision, for any of the operands. FANN-on-MCU [43] is an open-source framework that is built upon the Fast Artificial Neural Network (FANN) library [44] that can run light NNs on MCUs. FANN-on-MCU can process Multi-Layer Perceptrons (MLPs) trained with FANN, then generate code that can run on Parallel Ultra-Low Power Platforms (PULP) [45] (besides the mentioned ARM Cortex-M processor).

In contrast to the above, we provide a generic method that applies to any dataset trained tree-based classifiers. When our method is utilized, without any alterations, the standard/stable models from ML frameworks can be efficiently deployed and executed by MCUs of tiny IoT devices while consuming 0 bytes of SRAM. In [46, 47], we laid a foundation by exploring the porting and execution of ML classifiers, anomalies detection models on embedded systems in IoT.

IV. SRAM Optimized Approach Design

In Fig. 1, we present the design flow of our SRAM-optimized approach. This flow can be followed to execute any commercial/standard dataset trained or any pre-trained marketplace models on tiny IoT devices like HVAC controllers, smart plugs, etc. At first, the developer needs to port the standard Python scikit-learn trained ML classifier models (which are trained in a resource extensive setup) to its MCU executable C versions. Then, we need to stitch the generated classifier with the IoT use-case application, followed by efficiently deploying and executing models on MCUs and small CPUs of IoT devices. The pseudocode of the design flow in Fig. 1, is given in Algorithm 1 and 2. We explain our porting method in the upcoming subsection, followed by our IoT application stitching and execution method.

Algorithm 2 Executing classifiers on IoT devices: runs on TinyML hardware designed using MCUs and small CPUs.

```
1:  load ported DT classifier and test set      ▷ dependencies or external libraries not required for our approach
2:      #include DT_MNIST.h      ▷ file contains ported DT in C that matches human-readable version of trained model
3:          predict (X)                                   ▷ links ported DT model with main IoT application
4:      #include MNIST_test.h          ▷ file contains data samples (test set) to supply during onboard inference
5:          2D array X [test_set_size] [features_dim]    ▷ test_set_size rows containing features_dim features of test set
6:          Y [test_set_size]                             ▷ one row containing labels of all test set samples
7:  for i in test_set_size do
8:      predictions ← predict (X[i])     ▷ data is passed to predict function inside DT_MNIST.h during inference
9:  onboard evaluate                                   ▷ function to evaluate ported classifier on MCUs
10:     inference time ← use predict () inside millis ()          ▷ pass data for 1 sample and 100 samples
11:     accuracy ← compare predictions with Y [test_set_size]
```

A. SRAM Optimized Porting of ML Classifiers to Plain C

In this section, we explain how the proposed method performs SRAM-efficient porting of trained DTs and RFs.

1) SRAM Optimized Porting of Decision Trees: In MCUs and small CPUs based tiny IoT devices, the program space (flash memory) is always much greater than the available SRAM (see Table I). So, we propose a method, that when realized, produces a C version of DTs which does not depend on the SRAM during execution. Instead, it exploits the larger flash memory in order to enable the deployment and execution of bigger classifiers. In other words, we propose to sacrifice flash memory in favor of the limited SRAM since it is the scarcest resource in the majority of MCUs. The proposed SRAM optimized method, *hard codes* the DT splits in C, without storing any reference of the splits and other DTs related parameters/values into variables. Since our method does not allocate any variables, 0 bytes of SRAM will be consumed to execute the C version of the ported classifier to produce inference results.

When using the proposed method, the flash memory consumption will grow almost linearly with the increasing number of splits in DTs. This limitation cannot be addressed since there is no better alternative to store the hard-coded splits. Storing on SRAM is not feasible since the limited available memory restricts executing large-high-quality models, and the majority of MCUs do not have EEPROM to store the models. Although the external I2C peripheral-based EEPROM can be interfaced with MCUs, the model's code stored in such external NAND type flash memory, during the MCU power-up, gets copied to the internal SRAM from which the MCUs execute models. Again this approach leads to an SRAM overflow during runtime. Even in such SRAM-constrained cases, our method is well-suited to execute larger models since we do not store any model-related variables on SRAM. Also, since most of the new generation MCU boards like the ESP32 and ESP01s etc. have at least 1 MB of flash, which is sufficient for the proposed method to store and execute large DTs containing tens of thousands of splits.

2) SRAM Optimized Porting of Random Forests: RFs are based on the concept of *wisdom of the crowd*, where many DTs are combined via voting. Since RFs depend on trees, our core method explained above which efficiently ports the DTs, can in turn result in efficiently porting many other tree-based methods like RFs and XGBoost. Hence, the SRAM optimized method that produces 0 bytes consuming C classifiers applies to all algorithms that depend on trees to produce inference results. For example, it hard codes all composing trees of an RF classifier. But since the class votes have to be stored (proposing or implementing alternatives for class votes will result in altering the standard classifier versions), our method consumes a few bytes of memory for this purpose, which is negligible. Thus explained SRAM optimized porting of ML classifiers to plain C is summarised in Algorithm 1.

B. Executing Classifiers on IoT Devices

The IoT application executed by MCUs receives the input data in different formats such as sensor readings, voice signals, and image frames. When users intend to improve their device's intelligence, we recommend them to train a high-quality ML model that can produce inference results based on the data seen by their edge devices, then port that model to C code using the method from previous subsection. In this section, we first describe the structure of the generated C code. Then we explain how to stitch the C code with the IoT application and perform inference whenever required by the user or the IoT edge application.

To obtain prediction results using the SRAM optimized method, no dependencies or shared libraries are required to be added in the file system along with the C code of the model. In the proposed execution method, just the `.h` file needs to be compiled along with the user's main IoT edge application and flashed via any MCU-supported software such as Arduino IDE, Atmel Studio, Keil MDK, etc. The interior of the `.h` model file generated using the proposed method contains the C code of the user trained model. Here, during the programming or edge application design phase, the users have to just include the generated `.h` model as a header file at the beginning of their program. Inside any of the model files generated using our method, we provide a function named *predict*, to which the main program can pass the values for which it needs predictions. When *predict* is called, the MCU starts to execute the model using its default available C compiler (without requiring any dependencies or external libraries) as a subroutine, without

TABLE I
DATASETS, HARDWARE CHOSEN TO EVALUATE THE SRAM OPTIMIZED APPROACH.

	Name: feature dimension, class counts		
Datasets	Iris Flowers [48]: 4, 3		Banknote Auth [49]: 5, 2
	Heart Disease [50]: 13, 2		Haberman's Survival [51]: 3, 2
	Breast Cancer [52]: 30, 2		Titanic [53]: 11, 2
	MNIST Digits [54]: 64, 10		
	MCU#	**Name**	**Specs: flash, SRAM, clock**
MCU boards	1	ATmega328P, Arduino Nano	32kB, 8kB, 16MHz
	2	nRF52840, Adafruit Feather	1MB, 256kB, 64MHz
	3	STM32f10, Blue Pill	128kB, 20kB, 72MHz
	4	Generic ESP32	4MB, 520kB, 240MHz
	5	ATSAMD21, Adafruit Metro	256kB, 32kB, 48MHz
	6	ATmega2560, Arduino Mega	256kB, 8kB, 16MHz
	7	ESP-01S, ESP8266	1MB, 32kB, 80MHz
	CPU#	**Name**	**Basic specs**
CPU devices	1	Laptop	Intel Core i7, W10, 1.9GHz
	2	NVIDIA Jetson Nano	Ubuntu,128-core GPU, 1.4GHz
	3	Laptop	Intel Core i5, W10, 1.6GHz
	4	Laptop	Intel Core i7, Ubuntu, 2.4GHz
	5	Raspberry Pi 4	ARM Cortex-A72, Raspbian, 2.4GHz

disturbing the device's main routine, which is handled by the main IoT edge application. Thus explained model execution method on MCUs is summarised in Algorithm 2.

V. EXPERIMENTAL EVALUATION

To justify claims of the SRAM optimized approach, extensive experimental evaluations are performed using standard datasets and popular MCUs that are the brain of billions of resource-constrained IoT devices (TinyML hardware).

A. Devices, Datasets and Experiment Procedure

Table I presents the datasets and hardware (various popular MCUs and CPUs) used for the experimental evaluation of the SRAM optimized approach. To ensure an extensive evaluation, we selected 7 datasets that have feature dimensions ranging from 4 to 64 features and class counts from 2 to 10 classes. Using these datasets, we train DT and RF classifiers on high-resource setups using Python scikit-learn (we perform an 80/20 training/testing split for each dataset). Then, as explained in Section IV, we port it to C, stitch it with an IoT application, finally deploy and execute on MCUs 1-7, whose specifications are given in Table I. In the experiments, the ported classifier, the `.h` dataset file and the main program will be compiled and flashed on the MCU's memory using the Arduino IDE. We selected 5 popular CPUs, whose details are given in Table I. Similar to MCUs, we also execute the same 7 datasets trained DT and RF classifiers (14 models) on CPUs 1-5. The inference performance of both MCU 1-7 and CPUs 1-5 are reported in Fig. 2. The onboard memory consumption of MCUs 1-7 are reported in Fig. 3. For statistical validation, in both figures, the plotted values correspond to the average of 5 runs [1]. In the upcoming subsection, analysis is performed based on the obtained results.

[1]Code, experiment setting and performance report are available at https://github.com/bharathsudharsan/ML-Classifiers-on-MCUs

B. Analyzing the Inference Performance on MCUs

Here we present the time taken by each MCU to perform unit inference and inference for 100 samples for each of the 14 models. The onboard test for accuracy, F1 score is also performed. The inference performance results are shown in Fig. 2, and report the following observations:

- All the MCUs, irrespective of their specifications, for all the datasets, performed unit inference in less than 1 millisecond.
- The resource-constrained 8-bit Atmega328P MCU1 performed faster unit inference than the NVIDIA Jetson Nano GPU and Raspberry Pi 4 CPU. This is because, to execute ML models, the better-resourced devices depend on ML frameworks like TensorFlow Lite, which creates computational overheads. But such devices perform inference for 100 samples at much higher speeds than MCUs because, after initializing the framework, there are no significant overheads, and the advantage of multi-core, multi-thread processors running at high clock speed are utilized efficiently.
- During execution on MCUs, the ported models show the same level of accuracy and F1 score as its original models (before porting) when evaluated on high-resource lab setups.
- For datasets with low features count like Haberman's Survival, Iris Flowers, Titanic, and Heart Disease, the ESP32 (MCU 4) produces faster inference results for 100 samples than CPU class devices. For the largest 64-feature Digits dataset, the 3 $ ESP32 inferred for 100 samples in 7 ms, which is only $\approx$ 5 ms slower than CPUs 3 & 4, which are 200 times more costly than MCUs. During the experiment, we noticed that loading the models and datasets from `.h` files is much faster than loading from `.csv` files, using libraries like Pandas and NumPy.

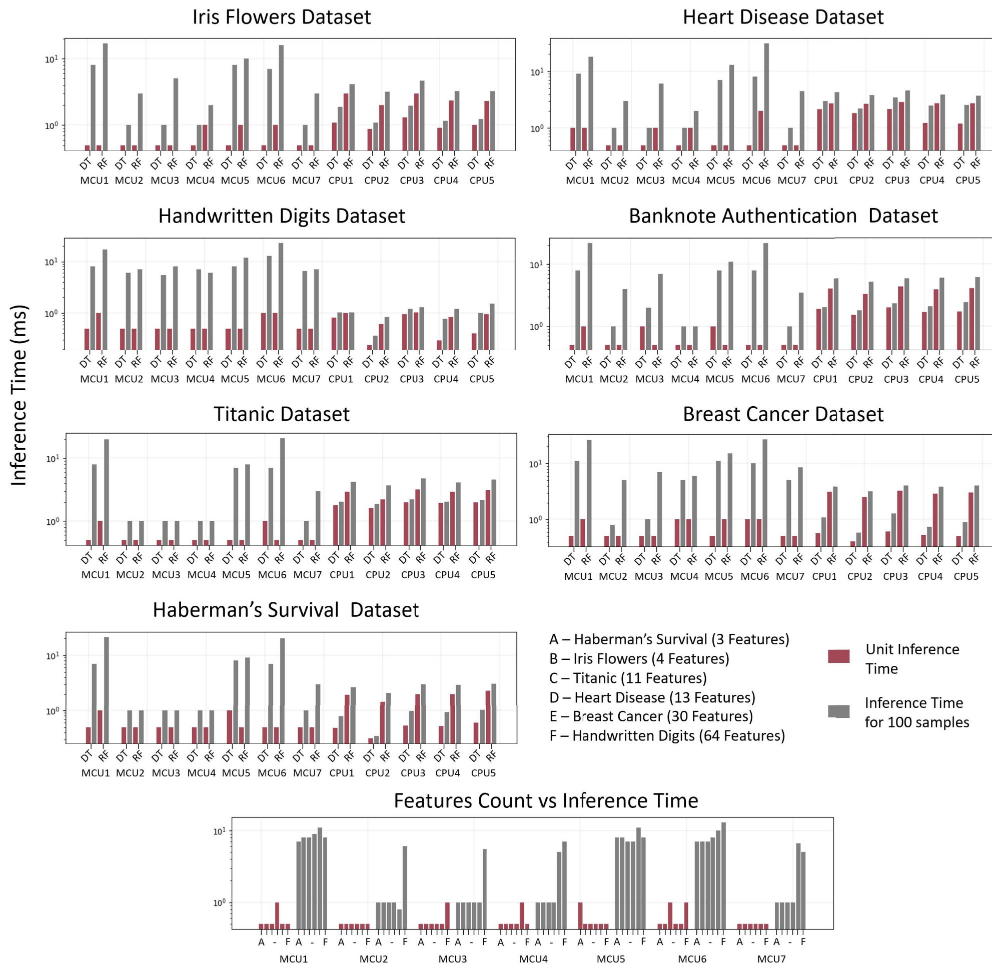

Fig. 2. Experiment results: Comparing time consumed by MCUs 1-7 (uses SRAM optimized approach) and CPUs 1-5 (uses Python scikit-learn) when performing inference for datasets of various features dimensions (3-64), class counts (1-10).

- From the subfigure titled Features count vs inference time in Fig. 2, we can observe that using high-feature data as input has only a few ms of impact on the unit inference time. Whereas for 100 input samples, the slower MCUs 1,5 & 6 show a logarithmic growth in inference time, the fast MCUs 2, 3, 4 & 7 show time growth only for the Cancer and Digits dataset.

It is apparent from the observations that the SRAM optimized method produces ultra-fast classification results on MCUs. Thus, even the autonomous tiny IoT devices can efficiently control real-world IoT applications by making timely predictions/decisions. Also, we report the ported ML classifiers, during execution, preserved model accuracy. This is because, unlike existing methods, the SRAM optimized approach does not perform pruning, sparsification, compression, or alter any properties and parameters of the high-resource ML framework trained classifiers. When users perform the same experiments or deploy their IoT use case models on advanced MCUs or Artificial Intelligence of Things (AIoT) boards like Sipeed MAIX Bit, M5 StickV, Sipeed Maix Amigo, they will obtain even faster inference results due to the onboard FPU, KPU, and FFT support.

C. Analyzing SRAM Consumption

In the same experimental setup, we execute the 7 datasets trained DT, RF classifiers (14 models) on MCUs 1-7, then report the Flash, SRAM consumption in Fig. 3, (y-axis in base-10 log scale) and report the following observations:

- It is apparent that all the DT and RF classifiers ported using the proposed method are of varied sizes (varying Flash consumption) but have constant SRAM consumption. For example, the RF classifier trained using the largest Digits dataset (64 features and 10 classes) has the largest model size after porting (so it occupies more Flash memory). But it consumes the same amount of SRAM as other classifiers produced by training using smaller datasets like Iris Flowers (4 features and 3 classes). Thus, our

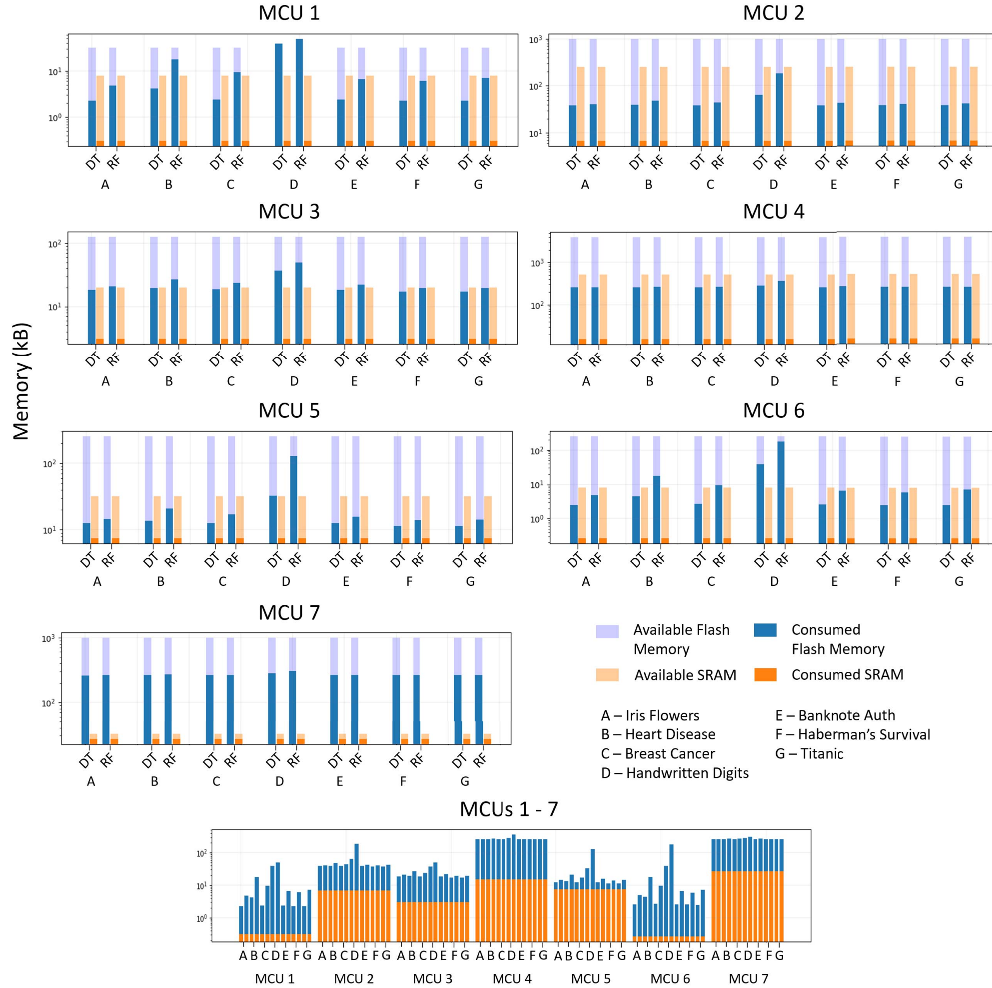

Fig. 3. Experiment results: Memory consumed by MCUs to execute various datasets trained DT and RF classifiers. As shown, SRAM optimized approach ported classifiers have varying Flash usage but constant SRAM usage.

SRAM optimized method is a promising way/method to fit and execute large models on MCU-based IoT devices.

- The MCU 7 has ≈ 31 times more Flash memory than the available 8 kB SRAM (see Table I). Similarly, other IoT hardware also has a significantly larger Flash than the SRAM in its Flash to SRAM ratio. Our approach is best suited for such scenarios since it does not store any model-related parameters/values in variables, so the ported models do not consume SRAM. Instead, it sacrifices the larger Flash memory in favor of the limited SRAM.
- Although a constant SRAM consumption was achieved, in MCU1, we faced a memory overflow issue since the model size was greater than the available Flash memory. Similarly, the Flash is almost full in MCU6. In such cases, we need to reduce the maximum tree depth during the model design phase and then perform porting.

The above results show that the proposed SRAM optimized method is applicable for a broad spectrum of various datasets trained ML classifiers. Also, the ported models are compatible to be executed on billions of TinyML hardware like resource-constrained embedded systems, IoT devices that have small CPUs and MCUs as their brain.

D. *Comparison with Existing Libraries*

As described in Section II-C, the emlearn [29], sklearn-porter [31], m2cgen [30] are popular C code generation libraries. To successfully, without errors, compile models generated by these libraries, we had to perform manual fine-tuning of their ported C code, which usually spans thousands of lines in the case of large models. This, demands time and a high level of debugging skills from the developer. As shown in Fig. 4, even after fine-tuning, many classifiers crashed. Few faced memory overflow issues when the Arduino IDE compiled the C code of the classifiers ported using these libraries for the target MCUs. Next, we also had to alter the data types of the input data according to the requirement from the ported model that performs inference, which affects the precision, thus resulting

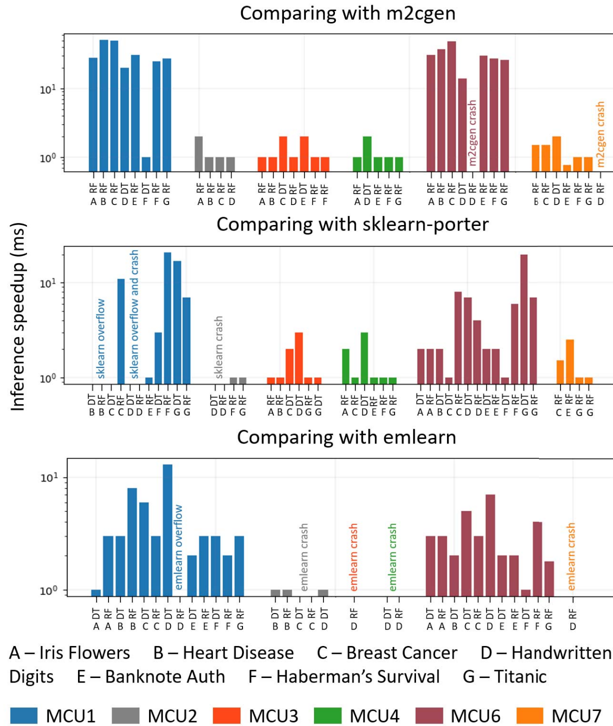

Fig. 4. Comparing inference time for 100 samples: Speedups achieved when porting and executing models on MCUs using SRAM optimized approach.

in less accurate classifications. We found the emlearn to be the most optimized library for MCUs, but still, to execute Tree-based models require an `eml_trees.h` file that consumes additional memory which is already at the peak utilization. We faced more SRAM overflow issues when using sklearn-porter since it declares all the model parameters like support vectors as variables that consume more memory. For example, when using the Breast Cancer dataset, it produced a 57 x 30 matrix of double data type, resulting in consuming 6.9 kB just to store the support vectors. To alleviate such issues, in Sections IV, we presented our SRAM optimized approach. When users realize this method to port trained classifiers, the generated C code will be stored in a `.h` file as shown in Fig. 1, and can readily execute on all the Arduino IDE supported MCU boards without requiring any fine-tuning or datatype conversions. Since the SRAM optimized method aims to simplify the deployment and execution of models on MCUs, the generated C code contains just one function to which the IoT application needs to send the data for which it requires classification results.

In Fig. 4, we compare the inference performance (for 100 data samples) of the models ported using the SRAM optimized method with the performance of the same models ported using m2cgen, sklearn-porter, and emlearn libraries. We report that the models ported and executed using the SRAM optimized method can infer 1-4 x faster. The highly resource-constrained MCUs 1 & 7 benefited the most since they achieved higher inference speedups than other boards.

VI. Conclusion

In this paper, an SRAM-optimized ML classifier porting, stitching, and efficient execution approach is presented. When researchers and developers apply this method to port and execute any use-case ML models on their IoT devices/products, similar to the experiment results, they can; (i) deploy and execute large problem-solving ML classifiers on low-cost, low-power MCU-based IoT devices that have only a few kB of SRAM; (ii) make even the small MCUs perform ultra-fast classifications (1-4x times faster than state-of-the-art libraries). In the future, we plan to run an integrity test (to ensure model quality preservation after porting) on the presented method and all its supported classifiers before packaging the code and releasing it as an open-source library.

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] B. Sudharsan, S. Salerno, D.-D. Nguyen, M. Yahya, A. Wahid, P. Yadav, J. G. Breslin, and M. I. Ali, "Tinyml benchmark: Executing fully connected neural networks on commodity microcontrollers," in *IEEE 7th World Forum on Internet of Things (WF-IoT)*, 2021.

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

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

[4] B. Sudharsan, P. Patel, J. G. Breslin, and M. I. Ali, "Ultra-fast classification on iot devices without sram consumption," in *IEEE International Conference on Pervasive Computing and Communications (PerCom)*, 2021.

[5] "Stable decision trees," 2021. [Online]. Available: https://scikit-learn.org/stable/modules/tree.html

[6] "Stable ensemble methods," 2021. [Online]. Available: https://scikit-learn.org/stable/modules/ensemble.html

[7] K. Z. Haigh, A. M. Mackay, M. R. Cook, and L. G. Lin, "Machine learning for embedded systems: A case study," in *BBN Technologies: Cambridge, MA, USA*, 2015.

[8] 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.

[9] F. Nan, J. Wang, and V. Saligrama, "Pruning random forests for prediction on a budget," in *Advances in neural information processing systems (NIPS)*, 2016.

[10] V. Y. Kulkarni and P. K. Sinha, "Pruning of random forest classifiers: A survey and future directions," in *IEEE International Conference on Data Science & Engineering (ICDSE)*, 2012.

[11] J. Shotton, T. Sharp, P. Kohli, S. Nowozin, J. Winn, and A. Criminisi, "Decision jungles: Compact and rich models for classification," in *Advances in neural information processing systems (NIPS)*, 2013.

[12] 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 *10th International Conference on the Internet of Things (IoT)*, 2020.

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

[14] "Tinygo: Go compiler for small places. microcontrollers, webassembly, and command-line tools. based on llvm." [Online]. Available: https://github.com/tinygo-org/tinygo

[15] "Embedded learning library (ell) an open source library for embedded ai and machine learning from microsoft," 2021. [Online]. Available: https://github.com/microsoft/ELL

[16] "Darknet: open source neural network framework written in c and cuda," 2021. [Online]. Available: https://github.com/pjreddie/darknet

[17] "Microsoft cognitive toolkit (cntk), an open source deep-learning toolkit," 2021. [Online]. Available: https://github.com/microsoft/CNTK

[18] "Open neural network exchange (onnx): The open standard for machine learning interoperability," 2021. [Online]. Available: https://github.com/onnx/onnx

[19] "Arm nn ml software," 2021. [Online]. Available: https://github.com/ARM-software/armnn

[20] L. Lai, N. Suda, and V. Chandra, "Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus," in *arXiv preprint*, 2018.

[21] "Ai expansion pack for stm32cubemx," 2021. [Online]. Available: https://www.st.com/en/embedded-software/x-cube-ai.html

[22] B. Sudharsan, J. G. Breslin, and M. I. Ali, "Ml-mcu: A framework to train ml classifierson mcu-based iot edge devices," *IEEE Internet of Things Journal*, 2021.

[23] 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 *10th International Conference on the Internet of Things (IoT)*, 2020.

[24] H. Ren, D. Anicic, and T. Runkler, "Tinyol: Tinyml with online-learning on microcontrollers," in *arXiv preprint*, 2021.

[25] "Empower your teams to quickly, easily and cost-effectively integrate ai into your projects," 2021. [Online]. Available: https://cartesiam.ai/

[26] "Tinyml ai inference library," 2021. [Online]. Available: https://github.com/uTensor/uTensor

[27] "Transpile trained decision trees from weka to c, java or javascript." 2021. [Online]. Available: https://github.com/nok/weka-porter

[28] L. T. da Silva, V. M. Souza, and G. E. Batista, "Embml tool: Supporting the use of supervised learning algorithms in low-cost embedded systems," in *IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*, 2019.

[29] D. Morawiec, "sklearn-porter: Transpile trained scikit-learn models." 2021. [Online]. Available: https://github.com/nok/sklearn-porter

[30] "m2cgen: Code-generation for various ml models into native code." 2021. [Online]. Available: https://pypi.org/project/m2cgen/

[31] "emlearn: Machine learning inference engine for microcontrollers and embedded devices," GitHub, 2021. [Online]. Available: https://github.com/emlearn/

[32] A. Grubb and D. Bagnell, "Speedboost: Anytime prediction with uniform near-optimality," in *Artificial Intelligence and Statistics*, 2012.

[33] S. Dhar, J. Guo, J. Liu, S. Tripathi, U. Kurup, and M. Shah, "On-device machine learning: An algorithms and learning theory perspective," in *arXiv preprint*, 2019.

[34] Y. Yao, Z. Xiao, B. Wang, B. Viswanath, H. Zheng, and B. Y. Zhao, "Complexity vs. performance: empirical analysis of machine learning as a service," in *Internet Measurement Conference*, 2017.

[35] T.-S. Lim, W.-Y. Loh, and Y.-S. Shih, "A comparison of prediction accuracy, complexity, and training time of thirty-three old and new classification algorithms," in *Machine learning*. Springer, 2000.

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

[37] A. Polino, R. Pascanu, and D. Alistarh, "Model compression via distillation and quantization," in *arXiv preprint*, 2018.

[38] 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 *International Conference on Machine Learning (ICML)*, 2017.

[39] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," in *International Conference on Learning Representations (ICLR)*, 2016.

[40] P. Gysel, M. Motamedi, and S. Ghiasi, "Hardware-oriented approximation of convolutional neural networks," in *arXiv preprint*, 2016.

[41] S. Bhattacharya, "Sparsification and separation of deep learning layers for constrained resource inference on wearables," in *14th ACM Conference on Embedded Network Sensor Systems*, 2016.

[42] A. Capotondi, M. Rusci, M. Fariselli, and L. Benini, "Cmix-nn: Mixed low-precision cnn library for memory-constrained edge devices," in *IEEE Transactions on Circuits and Systems*, 2020.

[43] X. Wang, M. Magno, L. Cavigelli, and L. Benini, "Fann-on-mcu: An open-source toolkit for energy-efficient neural network inference at the edge of the internet of things," in *IEEE Internet of Things Journal*, 2020.

[44] "Fast artificial neural network library (fann)," 2021. [Online]. Available: https://github.com/libfann/fann

[45] D. Rossi, F. Conti, A. Marongiu, A. Pullini, I. Loi, M. Gautschi, G. Tagliavini, A. Capotondi, P. Flatresse, and L. Benini, "Pulp: A parallel ultra low power platform for next generation iot applications," in *IEEE Hot Chips 27 Symposium (HCS)*, 2015.

[46] 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 *ACM/IEEE 12th International Conference on Cyber-Physical Systems (ICCPS)*, 2021.

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

[48] "Iris flowers dataset." [Online]. Available: https://archive.ics.uci.edu/ml/datasets/iris

[49] "Banknote authentication dataset." [Online]. Available: https://archive.ics.uci.edu/ml/datasets/banknote+authentication

[50] "Heart disease dataset." [Online]. Available: https://archive.ics.uci.edu/ml/datasets/heart+Disease

[51] "Haberman's survival dataset." [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Haberman's+Survival

[52] "Breast cancer dataset." [Online]. Available: https://www.kaggle.com/uciml/breast-cancer-wisconsin-data

[53] "Titanic survivor dataset." [Online]. Available: https://www.kaggle.com/c/titanic/data

[54] "Mnist handwritten digits dataset." [Online]. Available: http://yann.lecun.com/exdb/mnist/