Ultra-fast Machine Learning Classifier Execution on IoT Devices without SRAM Consumption

Bharath Sudharsan*, Pankesh Patel*, John G. Breslin*, Muhammad Intizar Ali†
*Confirm SFI Research Centre for Smart Manufacturing, Data Science Institute, NUI Galway, Ireland
{bharath.sudharsan, pankesh.patel, john.breslin}@insight-centre.org
†School of Electronic Engineering, Dublin City University, Ireland
ali.intizar@dcu.ie

***Abstract*— With the introduction of edge analytics, IoT devices are becoming smart and ready for AI applications. A few modern ML frameworks are focusing on the generation of small-size ML models (often in kBs) that can directly be flashed and executed on tiny IoT devices, particularly the embedded systems. Edge analytics eliminates expensive device-to-cloud communications, thereby producing intelligent devices that can perform energy-efficient real-time offline analytics. Any increase in the training data results in a linear increase in the size and space complexity of the trained ML models, making them unable to be deployed on IoT devices with limited memory. To alleviate the memory issue, a few studies have focused on optimizing and fine-tuning existing ML algorithms to reduce their complexity and size. However, such optimization is usually dependent on the nature of IoT data being trained. In this paper, we presented an approach that protects model quality without requiring any alteration to the existing ML algorithms. We propose SRAM-optimized implementation and efficient deployment of widely used standard/stable ML-frameworks classifier versions (e.g., from Python scikit-learn). Our initial evaluation results have demonstrated that ours is the most resource-friendly approach, having a very limited memory footprint while executing large and complex ML models on MCU-based IoT devices, and can perform ultra-fast classifications while consuming *0 bytes of SRAM*. When we tested our approach by executing it on a variety of MCU-based devices, the majority of models ported and executed produced *1-4x times* faster inference results in comparison with the models ported by the sklearn-porter, m2cgen, and emlearn libraries.**

***Index Terms*—Offline Inference, Intelligent Microcontrollers, Edge AI, Multi-class Classifiers, Efficient Model Deployment.**

I. Introduction

The vast majority of edge devices use simple supervised ML classifiers such as Decision Trees (DTs) and Random Forest (RFs) to solve ranking, regression, and classification problems locally at the device level. However, when users try to execute complex tree-based models on edge devices such as smart doorbells, HVAC controllers, smart energy meters, etc., these high-quality models with a large number of tree nodes often cannot fit within the memory of MCUs, resulting in memory overflow issues. For any given m training samples, implementation of the widely used stable scikit-learn DTs has $O(\log(m))$ as its inference complexity and $O(m)$ for its model size. Similarly, stable RFs have $O\left(N_{\text{tree}} \log(m)\right)$ inference complexity and $O\left(N_{\text{tree}} m\right)$ model size. Any increase in training samples results in an increase of the complexity of ML models. Multiple studies [1,2] have shown that tree-based algorithms can only be implemented on embedded sensor systems or tiny IoT devices after their refinement and fine-tuning (e.g. by reducing inference complexity and model size) to comfortably fit within the specific hardware architecture. To achieve high refinement levels, either the DTs and RFs are pruned [3,4], or node parameters in the DTs are shared using a directed acyclic graph [5]. Sometimes users design sparse and shallow tree learners that only require a few kBs of memory [6] to keep a low memory footprint. Such methods of learning shallow trees or aggressive pruning to fit within a few kBs often leads to degradation in accuracy due to approximation of non-linear and complex decision boundaries using a small number of axis-aligned hyperplanes. A few other studies have proposed compression and optimization [6] methods, where the models are trained in high resource setups, then a multi-stage MCU-aware optimization (tailored) is performed before deployment. In [7], authors have trained SVMs on MCUs. In [8], authors present a generic pipeline named 'RCE-NN', that can fit, deploy, and execute a broad spectrum of CNN-based models on tiny IoT devices. For example, 'RCE-NN' has been used to run 'COVID-Away' models [9] on smartwatches, as well as DNNs trained for biometric authentication [10] on Alexa smart speakers.

In this paper, in contrast to the aforementioned approaches, we do not reduce the model 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. Instead, we propose a generic method that applies to any dataset or framework trained classifiers. The main contributions of our work are as follows:

- Our method is generic and can efficiently port and execute a wide variety of ML classifiers on different resource-constrained MCU- and small CPU-based devices.
- In order to preserve accuracy, unlike existing methods, ours does not perform pruning, sparsification, compression, or alter any properties and parameters of the high-resource ML framework trained classifiers.
- When any ML classifiers are ported to C and stitched with IoT edge applications using our method, models consume 0 bytes of SRAM when executed on MCUs, thereby clearly superior to related methods.
- Contrary to the existing compression techniques, our approach reduces the size of ML models without any

alterations and the standard classifiers trained using any dataset can be efficiently deployed and executed on MCUs.

- Our method produce 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.
- Despite the reduced memory footprint, our approach guarantees the same level of performance (accuracy, F1 score, etc.) as its original models (before porting) when compared to high-resource lab setups.

II. Proposed Design

In this section, we present a design flow (as shown in Fig. 1) which can be followed to execute any commercial/standard dataset trained or any pre-trained marketplace models on tiny IoT devices. At first, our proposed method ports the standard Python scikit-learn trained ML classifier models (which are trained in a resource extensive setup) to its MCU executable C versions. Then, it stitches the generated classifier with the IoT use-case application, followed by efficiently deploying and executing models on MCUs and small CPUs of IoT devices. In the rest of this section, we explain our porting method, followed by our IoT application stitching and execution method.

A. SRAM-optimized Porting of Trained Classifiers to C

Here we explain how our method performs SRAM-efficient porting of trained DTs and RFs.

Porting 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 realised, 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 deploying and executing larger classifiers. In other words, we propose to sacrifice flash memory in favor of the limited SRAM since it is the most scarce resource in the majority of MCUs. Our 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 our 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 does not have EEPROM (see Table I) 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,

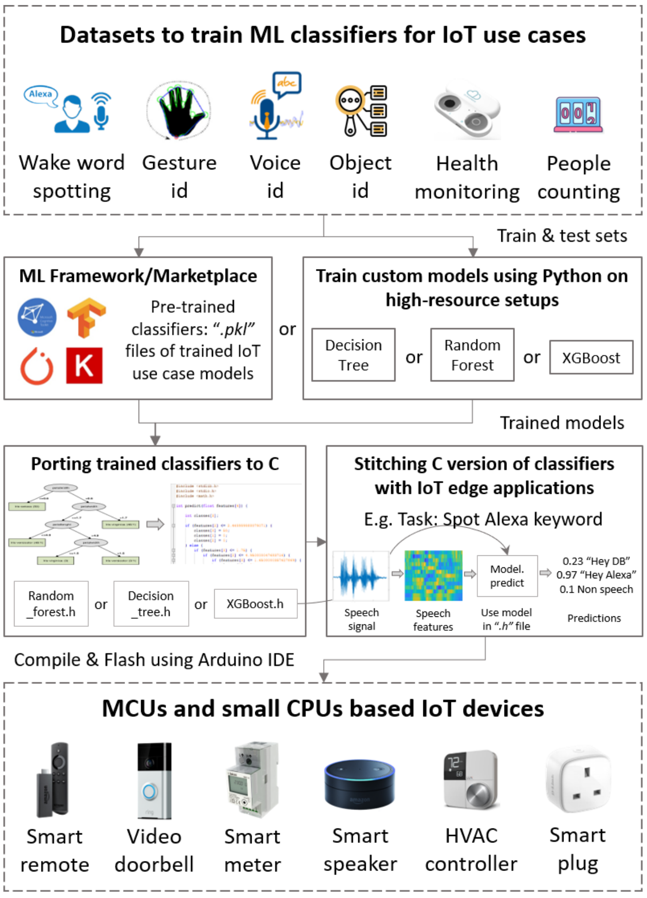

Fig. 1. Design flow to deploy and execute ML models on MCUs of tiny IoT devices: The models ported and stitched using our proposed approach produce ultra-fast classification results while consuming 0 bytes of SRAM.

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 our method to store and execute large DTs containing tens of thousands of splits.

Porting 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, our method that produces 0 bytes consuming C classifiers applies to all algorithms that depend on trees to produce inference results. For example, our method 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.

Porting Comparison. The sklearn-porter [11], m2cgen [12], and emlearn [13] are the popular optimized C code generation libraries. Here, we initially brief their limitations based on our experimental experience, then present our method that performs the same tasks of porting the trained classifier models to C.

To successfully compile models generated by the state-of-the-art 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. Thus, demanding time and a high-level of debugging skills from the users. As shown in Fig. 2, even after fine-tuning, many classifiers crashed, and 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 datatypes of the input data according to the requirement from the ported model that performs inference, which affects the precision, thus resulting in less accurate classifications. We found the emlearn to be the most optimised 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 consumes more memory. For example, when using the Breast Cancer dataset, it produced a 57 x 30 matrix of double data type, resulting in consuming 6.9kB just to store the support vectors.

To alleviate such issues we provide the users with our method, that when realised 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 our proposed method aims to simplify the deployment and execution of models on MCUs, the C code generated using our method contains just one function to which the IoT application needs to send the data for which it requires classification results. As presented in the evaluation section, classifiers ported and executed using our method achieve higher levels of SRAM conservation, do not compromise the model accuracy, consume 0-bytes SRAM, and produce inference results for 100 samples faster than the classifiers ported and executed using state-of-the-art libraries.

B. Stitching & Executing Ported C Model

The IoT application executed by the MCUs receives the input data that can be sensor readings, voice signals, and image frames, etc. 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 our method from Section II-A. In this section, we first describe the structure of thus generated C code, then explain how to stitch it with the IoT application and perform inference whenever required by the user or the IoT edge application.

As mentioned earlier, to obtain prediction results using our method, no dependencies or shared libraries are required to be added in the file system along with the C code of the model. In our 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 our 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 disturbing the device's main routine, which is handled by the main IoT edge application.

TABLE I
SPECIFICATION OF MCU BOARDS CHOSEN FOR EVALUATION.

Board or MCU	MCU & Board Name	Specification					
		Bits	EEP ROM	SRAM	Flash	Clock (MHz)	FP
#1	ATmega328P Arduino Nano	8	1kB	8kB	32kB	16	✗
#2	nRF52840 Adafruit Feather	32	-	256kB	1MB	64	✓
#3	STM32f103c8 Blue Pill	32	-	20kB	128kB	72	✗
#4	Generic ESP32	32	-	520kB	4MB	240	✓
#5	ATSAMD21G18 Adafruit METRO	32	-	32kB	256kB	48	✗
#6	ATmega2560 Arduino Mega	8	4kB	8kB	256kB	16	✗
#7	ESP-01S ESP8266	32	-	32kB	1MB	80	✗

III. EXPERIMENTAL EVALUATION

To justify our claims from Section I, here we evaluate our proposed method using standard datasets and popular MCUs that are the brain of billions of tiny IoT devices. The classifiers ported and stitched using our method from Section II, are executed on 7 popular open-source MCU boards whose specification is given in Table I (most boards lack FPU, KPU, and FFT support). 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, for which we train classifiers on high-resource setups using Python scikit-learn (we perform an 80/20 training/testing split for each dataset), then port it to C, stitch it with an IoT application, and finally deploy and execute on all MCUs using our method. To cover a broad range of dataset, we chose the following eight openly available datasets for our evaluation, Iris Flowers, Heart Disease , Breast Cancer , MNIST Handwritten Digits, Banknote Authentication, Haberman's Survival and Titanic dataset.

Model Performance on MCUs. We ported, stitched, and executed all the datasets trained DT and RF models. i.e., 14 models were ported, stitched, and executed on each selected MCUs using our method. We then performed an onboard test for accuracy, F1 score, and simultaneously recorded the time taken by each MCUs to perform unit inference and inference for 100 samples for each of the 14 models. From the obtained experimental results, we report the following observations; (i) All the MCUs invariable of their specifications, for all the

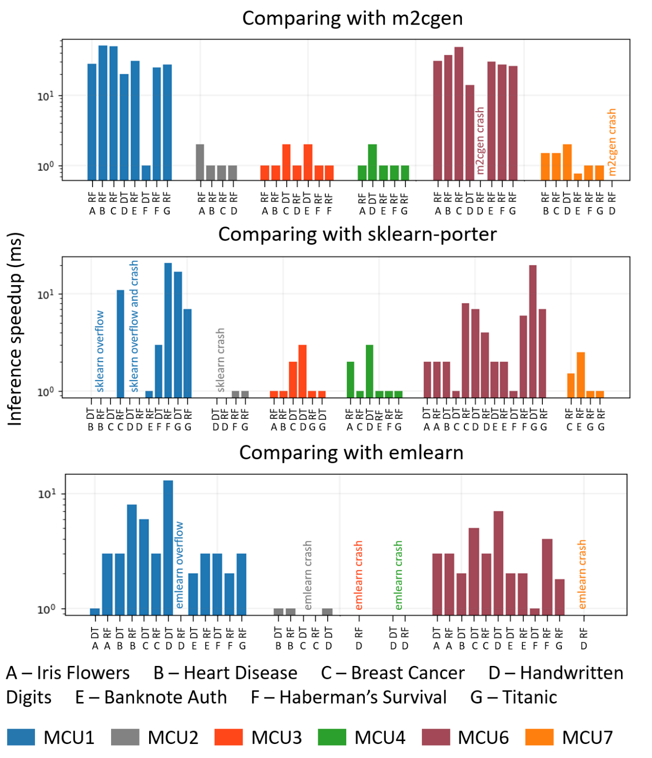

Fig. 2. Comparing inference time for 100 samples: Speedups achieved when porting and executing the models on MCUs using our proposed method.

datasets, performed unit inference in less than one millisecond, and show the same level of accuracy and F1 as its original models (before porting) when evaluated on high-resource lab setups. (ii) Even the slowest MCUs performed faster unit inference than NVIDIA Jetson Nano GPU and Raspberry Pi 4 CPU. (iii) For all datasets, the SRAM consumption remains constant, and only the flash consumption changes according to the size of the ported C model.

Comparing our Results with Existing Libraries. We take the same 14 trained classifiers, then using m2cgen, sklearn-porter, and emlearn libraries, we port it to C and execute them on MCUs 1-7 using the respective library recommended method. As shown in Fig. 2, we compare onboard inference time for 100 samples of these libraries ported classifiers with our results to report the speedups achieved when using our method. It is apparent from the comparison that the models ported and executed using our method can infer 1-4x times faster than existing libraries, and the highly resource-constrained MCUs 1 and 7 gets benefited the most since they achieved higher speedups than other boards.

IV. Conclusion: Discussion and Future Work

In this paper, we briefly presented our SRAM-optimized classifier porting, stitching, and efficient deployment method which is currently the most resource-friendly approach that enables ML framework trained standard/stable classifier versions to comfortably execute them within the limited memory footprint of MCU-based tiny IoT devices and perform ultra-fast classifications (1-4x times faster than state-of-the-art libraries) while consuming 0 bytes of SRAM.

In the future, we plan to extend our initial results to achieve the following: (i) Perform a comparison of the time consumed by MCUs (using our method) vs popular CPUs (using standard methods from scikit-learn) to execute and infer using the same classifiers; (ii) Profile the onboard memory consumption of our method in order to present its Flash and SRAM consumption patterns on MCUs during the execution of different classifiers; (iii) We plan to extend our approach to a broad range of classifiers such as SVM, SVR [14], Naive Bayes, and KNN, etc.; (iv) In order to get benefits from the memory-friendly and ultra-fast classifier implementations presented in this paper, we plan to apply our method to the lightweight video object detection models we created as a part of work at the Jupyter [15] consumer electronics company, then deploy it on video doorbell products; (v) Finally, we plan to run an integrity test (to ensure model quality preservation after porting) on our method and all its supported classifiers before packaging the code of our method 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] "Machine learning for embedded systems: A case study."
[2] J. Lee, M. Stanley, A. Spanias, and C. Tepedelenlioglu, "Integrating machine learning in embedded sensor systems for internet-of-things applications," in *2016 IEEE ISSPIT*.
[3] F. Nan, J. Wang, and V. Saligrama, "Pruning random forests for prediction on a budget," in *Advances in neural information processing systems*.
[4] V. Y. Kulkarni and P. K. Sinha, "Pruning of random forest classifiers: A survey and future directions," in *2012 IEEE ICDSE*.
[5] 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*, 2013.
[6] A. Kumar, S. Goyal, and M. Varma, "Resource-efficient machine learning in 2 kb ram for the internet of things," in *ICML*, 2017.
[7] 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*.
[8] 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*.
[9] B. Sudharsan, D. Sundaram, J. G. Breslin, and M. I. Ali, "Avoid touching your face: A hand-to-face 3d motion dataset (covid-away) and trained models for smartwatches," in *10th International Conference on the Internet of Things Companion*, ser. IoT '20 Companion.
[10] B. Sudharsan, P. Corcoran, and M. I. Ali, "Smart speaker design and implementation with biometric authentication and advanced voice interaction capability," in *Artificial Intelligence and Cognitive Science*.
[11] "emlearn," GitHub. [Online]. Available: https://github.com/emlearn/
[12] "m2cgen: Code-generation for various ml models into native code."
[13] D. Morawiec, "sklearn-porter: Transpile trained scikit-learn models."
[14] B. Sudharsan, J. G. Breslin, and M. I. Ali, "Adaptive strategy to improve the quality of communication for iot edge devices," in *2020 IEEE 6th World Forum on Internet of Things (WF-IoT)*.
[15] "Intelligent doorbell," jupyter. [Online]. Available: https://www.jupyter.com.au/