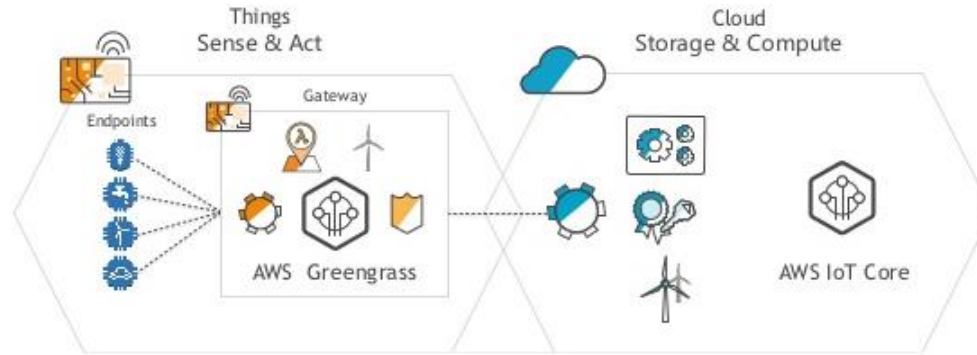


Train++: An Incremental ML Model Training Algorithm to Create Self-Learning IoT Devices

Bharath Sudharsan, Piyush Yadav, John G. Breslin,
Muhammad Intizar Ali

A World Leading SFI Research Centre

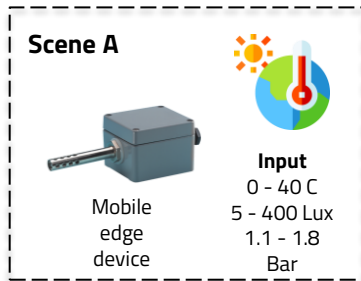




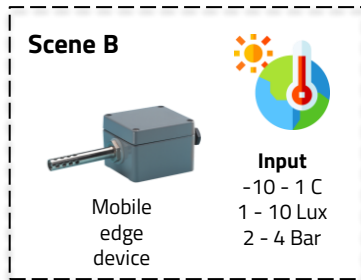
A typical IoT scenario: Edge devices (Endpoints) depends on cloud for inference and model updates

To improve inference accuracy, edge devices log unseen data in cloud. Initial model re-trained then sent as OTA update

- ✓ Edge devices hardware cost increase - wireless module addition
- ✓ Increases cyber-security risks and power consumption
- ✓ Not self-contained ubiquitous systems since they depend on the cloud services for inference and re-training
- ✓ Latency, privacy, other concerns



Accuracy: 92 %

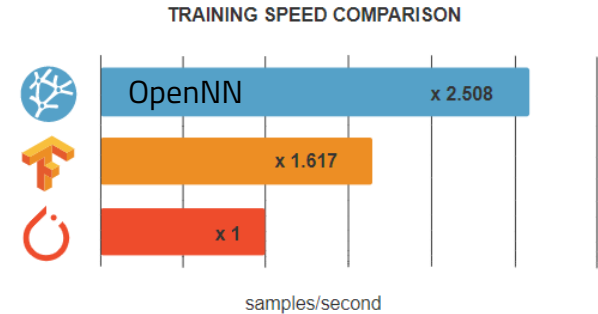


Accuracy: 54 %

IoT edge devices exposed to new scenes/environments

- In the real-world, every new scene generates a fresh, unseen data pattern
- When the model deployed in edge devices sees such fresh patterns, it will either not know how to react to that specific scenario or lead to false or less accurate results
- A model trained using data from one context will not produce the expected results when deployed in another context
- It is not feasible to train multiple models for multiple environments and contexts

- To enable the edge devices to learn offline after deployment - ML model deployed and running on the edge devices need to be re-trained or updated at the device level



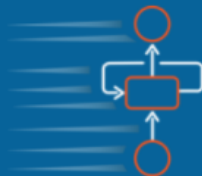
Top ML Frameworks for running ML models on MCUs. It does not yet support training models on MCUs

- ML frameworks like TensorFlow Micro, FeatherCNN, Open-NN, etc. do not yet enable training models directly on MCUs, small CPUs and IoT FPGAs
- Currently models are trained on GPUs then deployed on IoT edge devices after deep compression/optimization



Bonsai

A tree based classification/regression algorithm.



FastCells

A new class of RNN cells (FastRNN & FastGRNN) carefully designed for resource efficiency



EMI-RNN

A model independent algorithm for training smaller and faster RNN cells (GRU, LSTM, FastRNN and others).



ProtoNN

A k-nearest neighbours inspired prototype based classification/regression algorithm.

Microsoft EdgeML provides Algorithms and Tools to enable ML inference on the edge devices

- Bonsai: non-linear tree-based classifier which is designed to solve traditional ML problem with 2KB sized models
- EMI-RNN: speeding-up RNN inference up to 72x when compared to traditional implementations
- FastRNN & FastGRNN: can be used instead of LSTM and GRU. 35x smaller and faster with size less than 10KB
- SeeDot: floating-point to fixed-point quantization tool including a new language and compiler

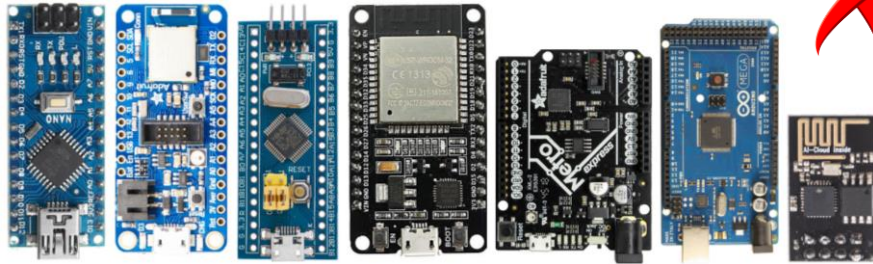
On Device ML Model Training



Powerful CPU + basic GPU based SBCs (single board computers)



Billions of IoT devices are designed using such MCUs and small CPUs



MCU1	MCU2	MCU3	MCU4	MCU5	MCU6	MCU7
ATmega328P	nRF52840	STM32f103c8	ESP32	ATSAMD21G18	ATmega2560	ESP8266
8 kB SRAM	256 kB SRAM	20 kB SRAM	520 kB SRAM	32 kB SRAM	8 kB SRAM	32 kB SRAM
32 kB Flash	1 MB Flash	128 kB Flash	4 MB Flash	256 kB Flash	256 kB Flash	1 MB Flash
@ 16 MHz	@ 64 MHz	@ 72 MHz	@ 240 MHz	@ 48 MHz	@ 16 MHz	@ 80 MHz

- Numerous papers, libraries, algorithms, tools exists to enable self-learning and re-training of ML models on better resourced devices like SBCs
 - ✓ Due to ML framework support i.e., TF Lite can run on SBCs and not on MCUs
- SEFR paper: Classification algorithm specifically designed to perform both training and testing on ultra-low power devices
- ML in Embedded Sensor Systems paper: Enable real-time data analytics and continuous training and re-training of the ML algorithm

- After real-world deployment, can the highly resource-constrained **MCU-based IoT devices** autonomously (offline) improve their intelligence by performing onboard re-training/updating (self-learning) of the local ML model using the live IoT use-case data?

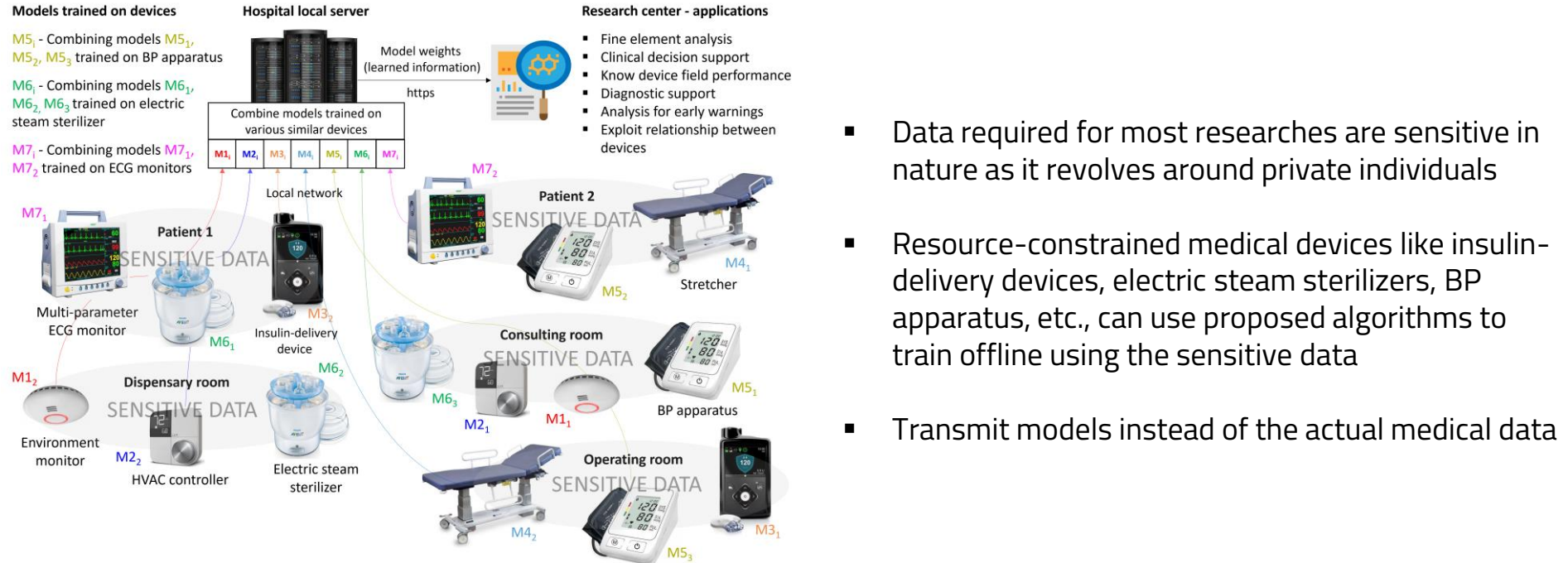
Use case 1: Self-learning HVACs



Create self-learning HVACs for superior thermal comfort

- HVAC controllers are resource-constrained IoT edge devices. For example, Google Nest, Echobee, etc.
- Every building/infrastructure has differences. Standard/one-size-fits-all strategy fails to provide superior level of thermal comfort for people
- Proposed algorithms can make HVAC controllers learn best strategy (offline) to perform tailored control of the HVAC unit for any building types
- Eliminates the need to find and set distinct HVAC control strategies for each building

Use case 2: Sensitive Medical Data



Providing sensitive medical data for research

- Data required for most researches are sensitive in nature as it revolves around private individuals
- Resource-constrained medical devices like insulin-delivery devices, electric steam sterilizers, BP apparatus, etc., can use proposed algorithms to train offline using the sensitive data
- Transmit models instead of the actual medical data

Use case 3: Learning any Patterns



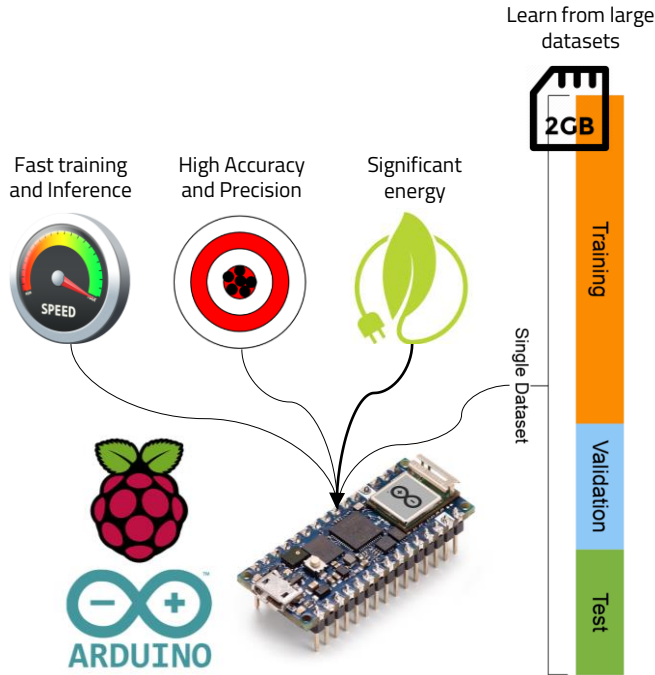
Use case: Learn to monitor vibrations on pump's crosshead, cylinder and frame. Generate alerts if anomaly patterns are detected

Use case: Learn to detect Leaks, monitor efficiency & consumptions



Self-learning the data patterns

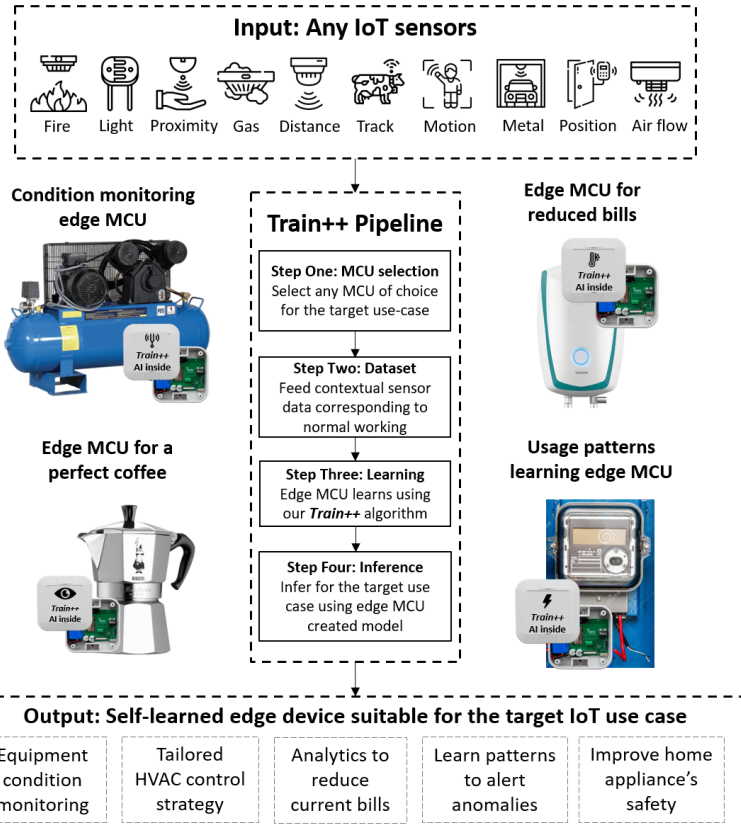
- Proposed algorithms can train models on MCUs. Thus, even the tiny IoT devices can self-learn data patterns
 - ✓ Learn usual residential electricity consumption patterns and raise alerts in the event of unusual usage or overconsumption. Thus can reduce bills, detect leaks, etc.
 - ✓ Learn by monitoring the contextual sensor data corresponding to regular vibration patterns from the pump's crosshead, cylinder and frame
 - ✓ Generate alerts using the learned knowledge if anomaly patterns are predicted or detected



Benefits of Training and Inference
on MCUs using Train++

- Train++ is designed for MCUs to perform on-board binary classifier model training, inference. Train++ algorithm main highlights/benefits
 - ✓ **Dataset size:** Incremental training characteristic enables training on limited Flash and SRAM footprints while also allowing to use full n -samples of the *high-dimensional* datasets
 - ✓ **Training speed and energy:** Achieves significant energy conservations due to its high-performance characteristics i.e., it trained and inferred at much higher speeds than other methods
 - ✓ **Performance:** Train++ trained classifiers show classification accuracy close to those of Python scikit-learn trained classifiers on high-resource CPUs

Train++: Pipeline



- Four-steps to use our Train++ algorithm to enable the edge MCUs to self-train offline for any IoT use cases
 - ✓ Select MCU of choice depending on the target use case
 - ✓ Contextual sensor data corresponding to normal working will be collected as the local dataset and used for training. The labels/ground truth for the collected training data rows shall be computed using our method from Edge2Train
 - ✓ Edge MCUs learn/train using our core Train++ algorithm
 - ✓ The resultant MCU-trained models can start inference over previously unseen data

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

Inputs:

C : Positive parameter to control the influence of ξ .
 t : The dimension of time for real-time data inputs.
 x_t : Real-time sensor data inputs. Where $x_t \in \mathbb{R}^n$.
 y_t : Correct labels. Where $y_t \in \{-1, +1\}$.

Output:

w_t : Incrementally learned weights.

Receive live data stream, represented as in Eqn (1).

function MCUTrain (x_t, y_t, t, C)

for $t = 0$ to setsize **do**

Predict \hat{y}_t for every x_t . Use $\text{sign}(x_t \cdot w_t)$.

Compute confidence score of prediction. Use $|x_t \cdot w_t|$.

Compute margin at t . Use $y_t(x_t \cdot w_t)$.

Show original label y_t to this algorithm.

if (\hat{y}_t equals to y_t) **then**

Prediction was correct. $y_t = \text{sign}(x_t \cdot y_t)$.

Predict again with a higher confidence score.

if ($\text{sign}(x_t \cdot y_t)$ lesser than 1) **then**

Suffer an instantaneous loss. Use Eqn (2).

if ($\text{sign}(x_t \cdot y_t)$ exceeds 1) **then**

Loss becomes 0.

else

Wrong prediction. Loss becomes $1 - y_t(x_t \cdot w_t)$.

end if

Incrementally learn w_t . Use Eqn (4).

Store the learned weights w_t .

end for

- The full Train++ algorithm is shown. Characteristics
 - ✓ Highly resource-friendly and low complexity design
 - ✓ Implementation less than 100 lines in C++
 - ✓ Does not depend on external or third-party libraries
 - ✓ Executable even by 8-bit MCUs without FPU, KPU, APU support
 - ✓ Incremental data loading and training characteristics
 - ✓ Compatible with datasets that have high feature dimensions
 - ✓ Keeps reading live data -> incrementally learns from it -> discards data (no storing required)

Inputs:

C : Positive parameter to control the influence of ξ .

t : The dimension of time for real-time data inputs.

x_t : Real-time sensor data inputs. Where $x_t \in \mathbb{R}^n$.

y_t : Correct labels. Where $y_t \in \{-1, +1\}$.

Output:

w_t : Incrementally learned weights.

Receive live data stream, represented as in Eqn (1).

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

First part: Dataset, Inputs and Output for Train++

- Input and output for Train++ core algorithm. Eqn (1) is the local dataset that is based on the IoT use-case data
 - ✓ X is the dataset rows that contain features of input data samples
 - ✓ Y holds the corresponding labels
 - ✓ dimension of time (t) is considered indefinite as real-time sensor data keeps arriving with indefinite length

```
function MCUTrain ( $x_t, y_t, t, C$ )  
  for  $t = 0$  to setsize do  
    Predict  $\hat{y}_t$  for every  $x_t$ . Use  $sign(x_t, w_t)$ .  
    Compute confidence score of prediction. Use  $|x_t, w_t|$ .  
    Compute margin at  $t$ . Use  $y_t(x_t, w_t)$ .  
    Show original label  $y_t$  to this algorithm.  
    if ( $\hat{y}_t$  equals to  $y_t$ ) then  
      Prediction was correct.  $y_t = sign(x_t, y_t)$ .  
      Predict again with a higher confidence score.
```

Second part

- Initially the algorithm infers using a binary classification function that updates from round to round and the vector of weights $w \in R^n$ takes the $sign(x, w)$ form
 - ✓ The magnitude $|x, w|$ is the confidence score of prediction
 - ✓ w_t is the weight vector that Train++ uses on round t
 - ✓ $y_t(x_t, w_t)$ is the margin obtained at t
- Whenever algorithm makes a correct prediction $sign(x_t, w_t) = y_t$
 - ✓ After prediction, we instruct Train++ to predict again with a higher confidence score
 - ✓ Hence, the goal becomes to achieve at least 1 as the margin, as frequently as possible


```
if (sign( $x_t, y_t$ ) lesser than 1) then
    Suffer an instantaneous loss. Use Eqn (2).
if (sign( $x_t, y_t$ ) exceeds 1) then
    Loss becomes 0.
else
    Wrong prediction. Loss becomes  $1 - y_t(x_t, w_t)$ .
end if
```

Third part

- Whenever $y_t(x_t, w_t) < 1$ Hinge-loss makes Train++ suffer an instantaneous loss

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

- ✓ If margin exceeds 1 the loss is zero
- ✓ Else, it is the difference between the margin and 1
- ✓ Now we require an update rule to modify the weight vector for each round

- Update rule to modify the weight vector for each round. ξ is a slack variable, C is the parameter to control influence of ξ

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

Incrementally learn w_t . Use Eqn (4).
Store the learned weights w_t .
end for

Fourth part

- ✓ Whenever a correct prediction occurs, the loss function is 0
- ✓ The *argmin* is w_t , hence Train++ algorithm becomes permissive
- ✓ Whereas on the rounds when misclassifications occur, loss is positive and Train++ offensively forces w_{t+1} to satisfy the constrain $l(w; (x_t, y_t)) = 0$
- ✓ Larger C values produce strong offensiveness, which might increase the risk of destabilization when input data is noisy
- ✓ Lower C values improve adaptiveness

- To keep w_{t+1} close to w_t to retain the information learned in previous rounds. The update rule in its simple closed-form is $w_t + \tau_t y_t x_t$. Substituting τ then substituting loss l_t we get Eqn (4)

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

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

Incrementally learn w_t . Use Eqn (4).
Store the learned weights w_t .
end for

Fourth part

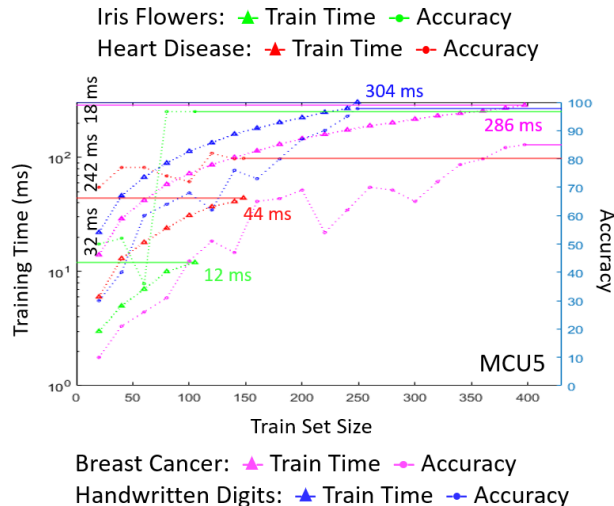
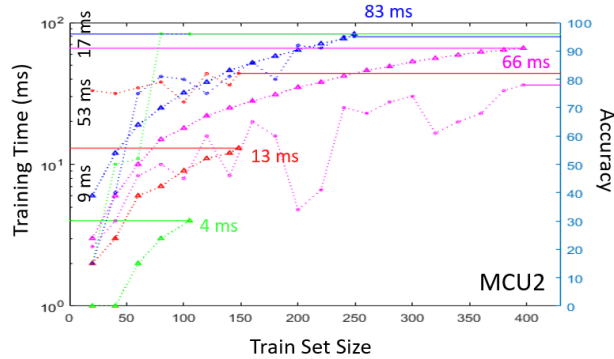
- ✓ This update rule meets our expectations since the weight vector is updated with a value whose sign is determined by y_t (magnitude proportional to error)
- ✓ During correct classification, the nominator of Eqn (4) becomes 0, so $w_{t+1} = w_t$
- ✓ During misclassification the value of the weight vector will move towards x_t
- ✓ After this movement, the dot product in the update rule becomes negative. Hence input is classified correctly as +1

Dataset#: Name - feature dimension			
Datasets	D1: Iris Flowers [34] - 4	D5: Banknote [35] - 5	
	D2: Heart Disease [36] - 13	D6: Survival [37] - 3	
	D3: Breast Cancer [38] - 30	D7: Titanic [39] - 11	
	D4: MNIST Digits [40] - 64		
MCU#		Name	
		Specs: processor flash, SRAM (kB), clock (MHz)	
MCU boards	1	nRF52840 Feather	Cortex-M4, 1MB, 256, 64
	2	STM32f10 Blue Pill	Cortex-M0 128kB, 20, 72
	3	Adafruit HUZZAH32	Xtensa LX6,
	4	Generic ESP32	4MB, 520, 240
	5	ATSAMD21 Metro	Cortex-M0+, 256kB, 32, 48

Datasets and MCUs used for Train++ evaluation

- Using Train++, for datasets D1-D7, we train a binary classifier on MCUs 1-5. This multiple datasets and MCUs based extensive experimental evaluation aims to answer
 - ✓ Is Train++ compatible with different MCU boards, and can it train ML models on MCUs using various datasets with various feature dimensions and sizes?
 - ✓ Can Train++ load, train, and infer using high features and size datasets on limited memory MCU boards that have low hardware specification and no FPU, APU, KPU support?

Results: Train Time



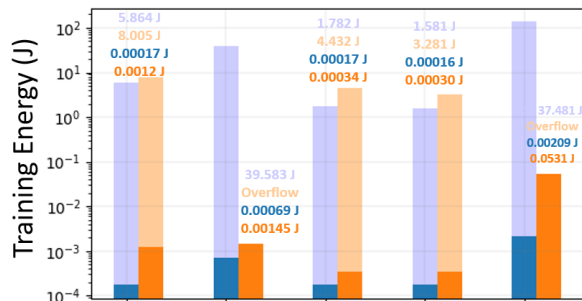
- Training models on MCUs using Train++: Comparing training set size vs training time and accuracy for selected datasets
 - ✓ We show results only for MCUs 2, 5 as other MCUs trained in 2 ms
 - ✓ The gap in y-axis is difference in train time between datasets
 - ✓ Train time grows almost linearly with samples for all datasets
 - ✓ For the Iris dataset MCU2 only took 4 ms to train on 105 samples
 - ✓ It took 83 ms to train on the Digits dataset
 - ✓ MCU5 is slowest since it only has a 48 MHz clock, does not have FPU support. Hence it took 12 ms to train on 105 samples of the Iris dataset (3x times slower than MCU2) and 304 ms to train on the Digits dataset (3.6x times slower than MCU2)

- **Memory:** Flash and SRAM consumption is calculated during compilation by the IDE
 - ✓ For MCU1, Iris dataset and Train++ in total used only 4.1 % of Flash and 3.4 % SRAM. For Digits, the same MCU1 requires 6.54 % and 29.93 %
 - ✓ When using Edge2Train, for MCUs 2, 5, we cannot train using the Digits dataset because SRAM overflowed by +52.0 kB and +63.62 kB. Similarly for Heart Disease both the Flash and SRAM requirements exceed the MCU's capacity
 - ✓ Even after overflow on MCU2, Train++ incremental training characteristics could load the dataset incrementally and complete training in 83 ms
 - ✓ Similarly, even after overflow on MCU5, Train++ was able to load the entire dataset incrementally and train in 304 ms
- **Inference time:** For the selected datasets, on MCUs 1-5, Train++ method infer for the entire test set in lesser time than the Edge2Train model's unit inference time

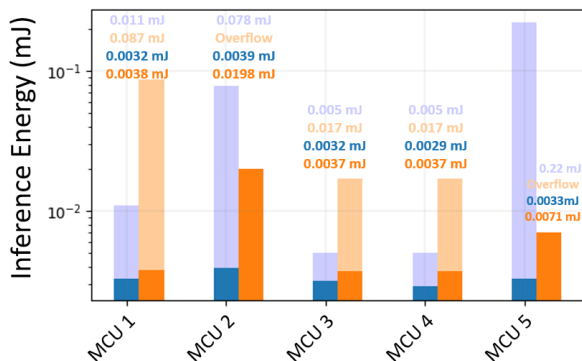
- **Accuracy:** The highest onboard classification accuracy is 97.33 % for the Iris (D1), 82.08 % for Heart Disease (D2), 85.0 % for Breast Cancer (D3), 98.0 % for Digits dataset (D4)
- **Accuracy comparison:** When comparing the accuracy of Train++ trained models with Edge2Train trained models
 - ✓ For the same Iris dataset, the accuracy improved by 7.3% and by 5.15% for the Digits dataset. This improvement is because our training algorithm enabled incremental loading of the full dataset
 - ✓ Other algorithms like SVMs work in batch mode, requiring full training data to be available in the limited MCU memory, thus sets an upper bound on the train set size
 - ✓ Train++ trained models achieved overall improved accuracy compared to the Edge2Train models, which were trained with limited data (unable to load full dataset due to memory constraints)

Results: Edge2Train vs Train++

a. Training energy consumed by MCUs.



b. Unit inference energy consumed by MCUs.



Edge2Train: Iris Flowers ■ Handwritten Digits ■

Train++: Iris Flowers ■ Handwritten Digits ■

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

- Incremental ultra-fast @ Train++: https://github.com/bharathsudharsan/Train_plus_plus


master ▾

1 branch

0 tags

Go to file

Code ▾

 bharathsudharsan Update README.md 88b54dd on Aug 2 71 commits

Train_plus_plus	readme with results	4 months ago
LICENSE	Initial commit	12 months ago
README.md	Update README.md	2 months ago
energy_comparison_for_train_and_inf...	image resize	4 months ago
setsize_vs_train_time_and_accuracy.png	image resize	4 months ago

☰ README.md

Resource-friendly, High-performance ML Classifier Training and Inference on Arduino MCUs

About

Repo and code of the IEEE UIC paper: Train++: An Incremental ML Model Training Algorithm to Create Self-Learning IoT Devices

machine-learning

microcontroller

optimization

esp32

adafruit

arduino-ide

online-learning

stm32f103

incremental-learning

edge-computing

classifier-training

armcortexm4

armcortexm0

📖 Readme

📄 MIT License

Languages

- Summary of Train++ benefits
 - ✓ The proposed method reduces the onboard binary classifier training time by $\approx 10 - 226$ sec across various commodity MCUs
 - ✓ Train++ can infer on MCUs for the entire test set in real-time of 1 ms
 - ✓ The accuracy improved by 5.15 - 7.3 % since the incremental characteristic of Train++ enabled loading of full n-samples of high-dimensional datasets even on MCUs with only a few hundred kBs of memory
 - ✓ Train++ is compatible with various MCU boards and multiple datasets
 - ✓ Across various MCUs, Train++ consumed $\approx 34000 - 65000$ x times less energy to train and consumed $\approx 34 - 66$ x times less energy for unit inference

Confirm

Smart Manufacturing

Confirm
Smart Manufacturing



Contact: Bharath Sudharsan

Email: bharath.sudharsan@insight-centre.org

www.confirm.ie


Science
Foundation
Ireland For what's next