On-Device Training of Machine Learning Models on Microcontrollers With a Look at Federated Learning

Marc Monfort Grau
Roger Pueyo Centelles
Felix Freitag
marc.monfort@gmail.com
rpueyo@ac.upc.edu
felix@ac.upc.edu
Technical University of Catalonia
Barcelona, Spain

ABSTRACT

Recent progress in machine learning frameworks makes it now possible to run an inference with sophisticated machine learning models on tiny microcontrollers. Model training, however, is typically done separately on powerful computers. There, the training process has abundant CPU and memory resources to process the stored datasets. In this work, we explore a different approach: training the model directly on the microcontroller. We implement this approach for a keyword spotting task. Then, we extend the training process using federated learning among microcontrollers. Our experiments with model training show an overall trend of decreasing loss with the increase of training epochs.

KEYWORDS

machine learning, keyword spotting, embedded systems, federated learning

ACM Reference Format:

Marc Monfort Grau, Roger Pueyo Centelles, and Felix Freitag. 2021. On-Device Training of Machine Learning Models on Microcontrollers With a Look at Federated Learning. In *Conference on Information Technology for Social Good (GoodIT '21), September 9–11, 2021, Roma, Italy*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3462203.3475896

1 INTRODUCTION

Inference with trained machine learning models is now possible on tiny computing devices, while a few years ago this was only run in the cloud [6]. This evolution has been enabled by the extension of popular machine learning frameworks such as TensorFlow and PyTorch to support lower capacity hardware, following the trend of moving computing services from the cloud to the network edge [7].

In common applications using machine learning on microcontrollers, models are typically trained separately, off-line, on more powerful computers. After training, the tools in the machine learning framework allow to optimize the model with regards to the resources consumption (e.g., memory usage on the target device). Some optimizations may involve a minor loss of accuracy, but the off-line training allows leveraging all the potential and facilities of these machine learning frameworks and using large training datasets [3].

In comparison, training on the microcontroller itself has limitations and is currently a niche approach only, but it may better respond to specific needs and enable new application areas. As to [5], training on the microcontroller allows to move from static models into dynamic ones, which could adapt to new data. Scenarios where such capacities have potential include a large number of distributed and remote devices, where performing updates with external models is not feasible. There could also be a potential for energy savings, due to the low energy consumption of microcontrollers. Finally, for artificial intelligence applications development, training on the device reduces the need to have access to higher-end computing devices.

In recent years, the technique of federated learning has raised the interest of the research community, as it provides a means to train machine learning models on distributed devices without sharing the local training data [8]. In federated learning, instead of training a single model with a centralized dataset, local models are trained with local datasets and then merged into a global model. The inconvenience of having fewer data at each device can be compensated by the capacity of the global model built upon the local ones. Federated learning is also seen as a solution to train with data which, for privacy reasons, cannot be sent to the cloud, such as medical records [2].

In this paper, we aim to achieve a better understanding of the feasibility of conducting the training process of machine learning models on microcontrollers, and of the possibility of improving the process via federated learning. We have developed a keyword spotting (KWS) task with an Arduino Nano 33 BLE Sense board to experiment this approach. Different words are spoken by the user and the machine learning model on the device is trained online. In the federated learning process, trained local models are exchanged with a central server.

The final publication is available at ACM via http://dx.doi.org/10.1145/3462203.3475896

2 BACKGROUND

TinyML targets at running machine learning applications on microcontrollers [1]. Typical TinyML applications perform the training phase outside the microcontroller. Popular machine learning frameworks such as TensorFlow provide the tools to train a model on powerful computing machines, such as PCs, or in the cloud. After training, the model is pruned and quantized to reduce its size. Finally, the optimized model is uploaded to the microcontroller board.

In this off-device training approach it is not possible to modify the model once it has been deployed. For instance, users cannot improve the model with their own local data once the model is flashed on the device. Such an option could be of interest for training the model with a user's own voice characteristics, and have an application with more personalized performance. However, model training is a computationally intensive task.

Some of the biggest neural network models can take up to months to be trained, accessing enormous datasets, and spending huge amounts of energy. It is clear that, for microcontrollers, the target scenario must be different, since computing power and energy are typically very restricted. Therefore, instead of training big and general-purpose models, the opportunity seems to be there for training small and specific models. The keyword spotting task which we present in this work is an example for those kind of applications. Given these boundaries, it is computationally possible to train a tiny model using a microcontroller device for a real application.

Transfer learning is a technique that can significantly reduce the time and the computing power required to train a new model. As such, it is principally relevant for resource-constraint environments such as microcontrollers. This technique takes advantage of the training performed on previous models to produce a new one. This could be as simple as training a model to recognize cats in images, making use of a pre-existing model that recognized dogs. Instead of training a new model from the beginning, it takes some of the weights and biases from an already trained model. Usually, the weights and biases of the first layers are used, since they recognize the most basic patterns on the data. The new model only has to train the layer closer to the output, which is responsible to recognize the most specific patterns. Therefore, the advantage of transfer learning is that only a subset of the total layers of the neural network needs to be trained. However, transfer learning has its limitations: if the purpose of the previously trained model is not related to the purpose of the new model, the result will be very poor. The problem to be solved needs to be similar enough, which sometimes may be hard to discern. If used wisely, transfer learning can suppose a great benefit on the training time of new models.

An important requirement for training a neural network model is to have high floating point precision. The gradient descent technique used in backpropagation is an iterative optimization algorithm for finding a local minimum of a derivative of a function (e.g., the loss function). Each iteration of the gradient descent takes a small step in the opposite direction of the function's gradient. The value of the gradient can be very small and therefore requires a high precision floating point unit to represent it. This is not a problem in an off-device training scenario when a general-purpose computer is used to train the model, since it has high floating point precision available. However, microcontrollers can be of different architectures, and not all of them even have floating point support. The ones with an FPU unit can perform the gradient descent calculations with the required floating point precision while, otherwise, specific solutions need to be given [4]. The microcontroller of the Arduino Nano 33 board we use in this work does have a FPU unit, which enables the training on the device.

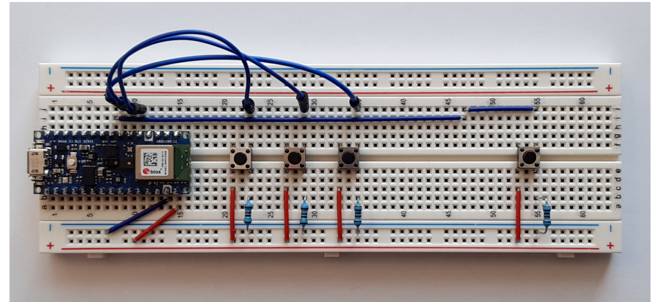

Figure 1: An Arduino Nano 33 BLE Sense board and the buttons to control the training and inference process on a breadboard.

3 APPLICATION DEVELOPMENT

This section describes the development of the keyword spotting application to study the training of a model on the microcontroller. The application shall be able to recognize up to three different keywords, which will be decided by the user when starting to train the model. The user can switch to an inference mode to test the keyword detection with the trained models. Currently, the detected keyword is notified to the user by illuminating the color corresponding to the keyword on the RGB LED of the Arduino board.

3.1 Hardware setup

To interact with the application we need to prepare the hardware. The application is deployed on an Arduino Nano 33 BLE Sense board, which already integrates several of the required components. The board has an integrated microphone, that will be used to record the keywords. It also has an integrated white LED and an RGB one. The white LED will be used to visualize the application stage (e.g. IDLE, busy), and the RGB LED will be used to show the output class of the keyword spotting model. To train the model with three keywords, we connect three button to the digital inputs of the board. Each button allows to train one of the keywords. A fourth button is added for testing the model (i.e., to perform inference). Figure 1 shows how the Arduino board is connected with the buttons.

3.2 Workflow

The application starts when the program is flashed to the Arduino board (or the board is restarted with the program already flashed). Every time the board is powered up, a new model is created. The weights and biases of the model are initialized to random numbers. After the model is initialized, the user can start training their own model using any of the three training buttons. Each button allows

to train one of the three keywords. When a training button is pressed, the RGB LED will light up with a color identifying the button (red, green or blue). When the button is released, the Arduino built-in microphone will start recording audio for one second. The keyword must be spoken within this second. The recorded audio is then processed to obtain the feature vector. The model is trained in a supervised fashion with the feature vector corresponding to the spoken word giving the label (the label is known from the button pressed). The fourth button has the same workflow as the three training buttons, but it does not train the model. Instead, in inference mode, it lights up the RGB LED with the corresponding color upon keyword recognition.

3.3 Feature extraction

In order to train the model, a feature vector needs to be obtained. As mentioned above, just after releasing any of the buttons the microphone starts recording for one second. The recording uses a sample rate of 16 kHz, and the result is stored in an array of 16000 values. Each value is represented by a 16-bit signed integer. Therefore, the recorded audio has a size of 32 kB. Given the recorded audio, we obtain the features that will be used to train the model. A popular way to extract features from human voice is using the Mel Frequency Cepstral Coefficients (MFCCs). Computing the MFCCs (with 13 coefficients) for our recorded audio we obtain a spectrogram of 13 rows and 50 columns. Figure 2 shows an example of the spectrogram obtained from the MFCCs calculation. This spectrogram represents the feature vector used to train the model.

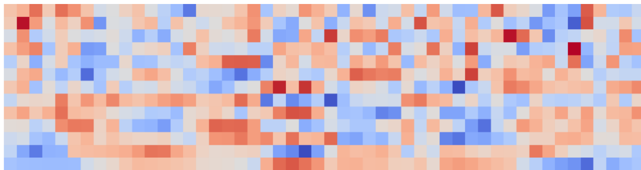

Figure 2: MFCC spectrogram for 1 second recording of the keyword "Montserrat".

3.4 Neural network model

Our application aims at training a model on a microcontroller. For that, the neural network model needs to be small enough to fit in the Arduino board. That being said, the model used in our application is a feedforward neural network with a single hidden layer of 16 nodes. The size of the hidden layer was determined following best practices from similar keyword spotting applications. The input layer has 650 nodes, the same number as the size of the feature vector obtained from the MFCCs algorithm (13*50). The output layer only has 3 nodes, each one representing a keyword. This architecture gives a sum of 10448 weights and 19 biases.

The data type used to represent the weights and biases are 4-byte floats (the maximum precision float allowed by the Arduino board). Therefore, the model has a final size of 41868 bytes. These bytes are not stored on the slow Arduino flash memory (1 MB) because they are constantly modified during the training phase, so they need to be stored in the RAM (preferably in the heap region). This is not an issue in the Arduino Nano 33 BLE Sense board, for it has 256 kB of SRAM. Figure 3 shows the neural network just described.

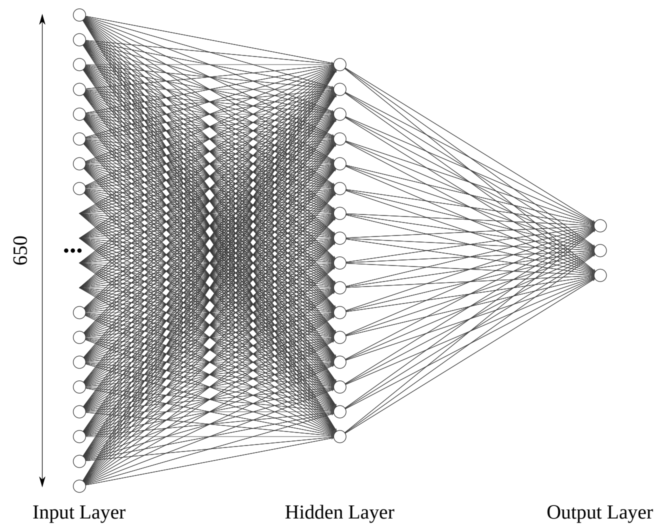

Figure 3: Neural network architecture.

3.5 Model training

The model is trained using an online learning approach. As mentioned, the model is initialized every time the application is restarted. When a training button is pressed and released, the board records a keyword, spoken by the user, and generates the feature vector (as MFCCs). Then, the feature vector is sent to the model to perform a forward propagation and to obtain the values of the three output nodes. With these output values and the expected values (the label is known from the button pressed), the mean square error is obtained. The final step is to calculate the delta (which reflects the magnitude of the error) of each neuron in order to perform the step in the backpropagation algorithm and update the weights and biases.

To optimize the model training, we can fine-tune the hyperparameters. The *learning rate*, which controls how much the model is updated in response to the estimated error of each training epoch, is among the most important ones. Choosing the correct learning rate is quite challenging. Too small a value may result in a long training phase, and too high a value may result in an unstable training process. The default learning rate we use in the application is 0.3. This value is quite high when compared with values used for the training of more complex models. However, since the application needs the user's active participation to become fully trained, it is advisable to not extend the training phase for too long.

The second hyperparameter is the momentum. The *momentum* tries to maintain a consistent direction on the gradient descent algorithm. It works by combining the previous heading vector and the new computed gradient vector. The default momentum value used in our setup is 0.9, which adds 90 % of the previous direction to the new direction. We note that the use of the momentum consumes additional RAM, since it is necessary to store the previously obtained gradient.

3.6 Software implementation

We implement the keyword spotting application for a federated learning architecture as shown in Figure 4. This architecture consists of a central server and several workers. The central server calculates the model averaging with the received local models after a determined number of training epochs in the worker nodes. This model averaging leads to a new global model. This updated global model is sent back to the worker nodes, which train again their local model with the data available on each node.

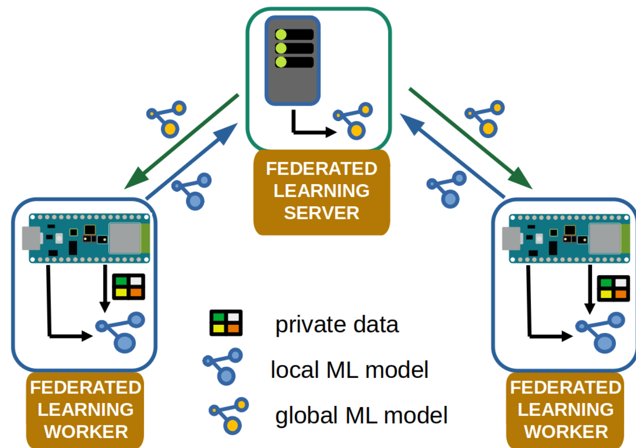

Figure 4: Implemented federated learning components.

To build the components, we have implemented from scratch the federated learning server in Python. The server runs on a PC. It communicates with the worker nodes over serial lines. It sends and receives model data from the worker nodes. Furthermore, for the experimental evaluation, the server receives the model accuracy given by the loss which is computed in the forward path when the model processes new data (i.e., a spoken word). The development of the worker nodes uses publicly-available code of a neural network implementation [1] and code from EdgeImpulse [2] for the speech processing and MFCCs calculation.

An important part of federated learning is the aggregation of the models. The technique used in our implementation is FedAvg, which stand for federated average. In contrast to FedSGD (stochastic gradient descent) where the aggregation is done on the gradients, with FedAvg, the aggregation is done on the model's parameters (weights and biases). After the server receives all the models from the clients, the parameters are averaged in order to produce the new global model. The implementation of the application used in this work is publicly available at https://github.com/MarcMonfort/TinyML-FederatedLearning.

4 EXPERIMENTAL RESULTS

The experiments presented in this section are divided into two groups: the first one focuses on how the training of the model on the device performs; the second group of experiments performs the model training in combination with federated learning.

[1] http://robotics.hobbizine.com/arduinoann.html
[2] https://www.edgeimpulse.com/

4.1 Single device training

Model training on device with two keywords. In order to evaluate the performance of on-device training, the model is trained with a set of keywords. This set of keywords contains two spoken words, namely *Montserrat* and *Pedraforca* (two iconic mountains in Catalonia) and a third type to classify silence. The keywords are spoken to the device in alternate order. Figure 5 shows the result of the training process. It can be seen that the overall trend is that loss decays with the increase of training epochs. The apparently erratic small-scale behavior is produced by the online learning approach, where each epoch is using a single audio recording and, therefore, the accuracy obtained can vary significantly between one word and the next. However, the important observation is that loss progressively decays over the epochs.

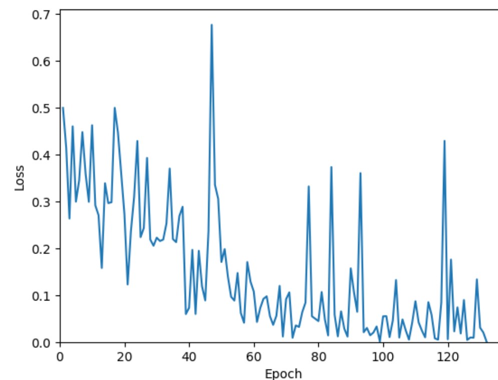

Figure 5: Loss vs. epochs during the training of the three keywords (*Montserrat*, *Pedraforca* and silence). Number of observations is 130, learning rate 0.1, momentum 0.9.

Model training on device with three keywords. We use a second set of keywords with tree keywords, which are *vermell*, *verd* and *blau* (which stand for red, green and blue, in Catalan). We trained the model with the hyperparameters learning rate 0.3 and momentum 0.9. Figure 6 shows the results of the training. It can also be seen how loss decays over the training epochs. As in the first experiment, the learning rate is relatively large. A reason for choosing a high learning rate was that the training data is scarce, since we use just one spoken words for each training epoch.

4.2 Training in a federated learning scenario

In order to perform the training process with federated learning we deploy the scenario shown in Figure 4. Two Arduino Nano 33 BLE Sense boards are used, with identical hardware setup as previously shown in Figure 1. The devices are connected via the USB serial interface to a PC, where the federated learning server runs.

We run two experiments in which, after 10 training epochs on each worker, the local model is sent to the server, to be averaged. Then, a new global model is sent back to both workers for the next local training round. Two spoken keywords are used for training,

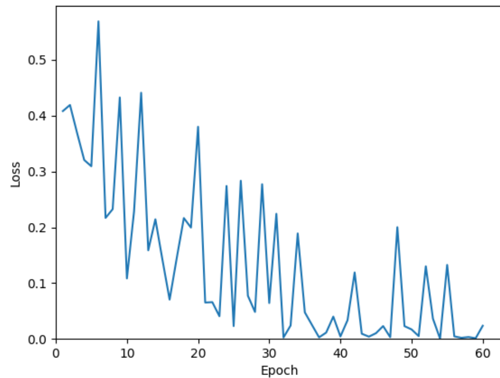

Figure 6: Loss vs. epochs during the training of the three keywords ("vermell", "verd" and "blau"). Number of observations is 60. learning rate 0.3, momentum 0.9.

Montserrat and *Pedraforca*. The difference lays in which of the local training data are spoken to each of the boards.

Federated learning with IID data . In this experiment, the two keywords are spoken in alternating order to both workers (nodes), to represent the scenario of training with independent and identically distributed (IID) data. Figure 7 shows the obtained results for the training loss. It can be seen that, in both nodes, the loss decreases over the training epochs.

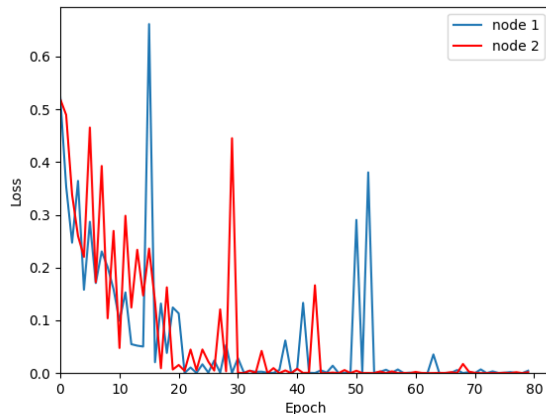

Figure 7: Loss vs. epochs during training with federated learning in IID data scenario (10 training epoch per aggregation).

Federated learning with non-IID data . In this experiment, the two keywords are split among the workers (i.e., each keyword is spoken to only one of the workers). This setup aims at presenting the scenario of training with non-IID data. It can be seen in Figure 8 how, after averaging the model every 10 epochs, loss increases. This can be explained by the fact that the model averaging merges the characteristics of the two models which were trained with different keywords.

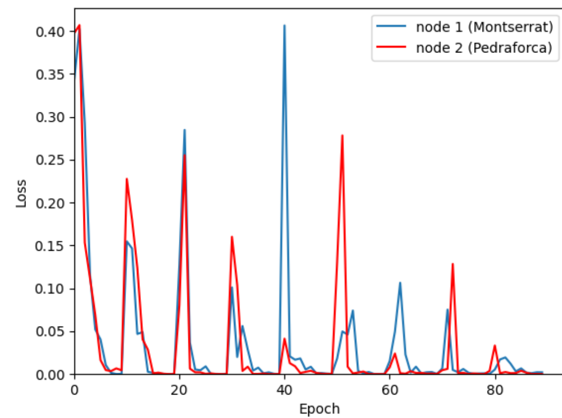

Figure 8: Loss vs. epochs during training with federated learning in non-IID data scenario (10 training epoch per aggregation).

4.3 Discussion

The results from the experiments showed the feasibility of combining on-device training on the microcontrollers with federated learning. We could train a global keyword spotting model, by aggregating local models trained on distributed devices. However, we also identified several improvements or changes that can be made for the application. For example, the devices could be allowed to start with a pre-trained model. Such a configuration would reduce the training time if the keywords in the application are kept unchanged. Another issue to tackle is found in the current communication protocol, which is slow due to the small size of the Arduino's USB port buffer (64 bytes). In order to avoid a buffer overflow, in the current implementation the server only sends four bytes at once, before checking that the board (i.e. client) has received them. Therefore, the sending time in the application from server to client is significantly longer (≈3 sec) than the receiving time from client to server (≈1 sec).

While federated learning has an important potential in avoiding to share the local training data (e.g., private health records), it also faces challenges when applied to embedded devices. In the learning phase, there is frequent communication between the server and the clients and the amount of data corresponding to the model parameters can be high. Depending on the aggregation methods, it may be necessary to exchange several times the model's parameters between the clients and the server. Depending on the wireless communication link available on the embedded device, this technology may not be designed for an intense data payload and the communication link may become the bottleneck of the training phase. Moreover, the clients on embedded systems involved in federated learning may be unreliable. For instance, they commonly use

less powerful radio communication chips and usually depend on batteries. As a consequence, they may drop out from the training phase more frequently than devices in other environments. Lastly, in real environments, the local data at each device will probably not be independent and identically distributed (IID) and, furthermore, the amounts of local data available for training at each device may be very heterogeneous. Such circumstances will require to apply more advanced algorithms in the server to properly determine the new global model.

5 CONCLUSIONS AND FUTURE WORK

A keyword spotting application, which performs on-device training of a neural network model using an Arduino Nano 33 BLE Sense boards, was presented. We described how a user can train this application by speaking keywords to the board's microphone and, using the buttons connected to the digital inputs, indicate the label for each of the keywords. This way, supervised learning is carried out. We observed, by evaluating the loss function in the forward path of the training process, how the model trained on the device improves during the training of tens of spoken words. Afterwards, the training was extended to a federated learning scenario with a central server and two worker nodes. The keyword spotting was trained again, now in a distributed way. The experiments showed also a decreasing loss trend as the training rounds increased.

The work gave practical insights on how on-device training can be implemented on a recent microcontroller board. Future work will explore further the raised scenario of non-IID data and federated learning training to understand deeper its capacity to obtain versatile models if there is only limited local training data.

ACKNOWLEDGMENTS

This work was partially funded by the Spanish Government under con-tracts PID2019-106774RB-C21, PCI2019-111850-2 (DiPET CHIST-ERA), PCI2019-111851-2 (LeadingEdge CHIST-ERA), and the Generalitat de Catalunya as Consolidated Research Group 2017-SGR-990.

REFERENCES

[1] Colby R. Banbury, Vijay Janapa Reddi, Max Lam, William Fu, Amin Fazel, Jeremy Holleman, Xinyuan Huang, Robert Hurtado, David Kanter, Anton Lokhmotov, David Patterson, Danilo Pau, Jae sun Seo, Jeff Sieracki, Urmish Thakker, Marian Verhelst, and Poonam Yadav. 2021. Benchmarking TinyML Systems: Challenges and Direction. arXiv:2003.04821 [cs.PF]

[2] Olivia Choudhury, Aris Divanis, Theodoros Salonidis, Issa Sylla, Yoonyoung Park, Grace Hsu, and Amar Das. 2020. Differential Privacy-enabled Federated Learning for Sensitive Health Data. arXiv:1910.02578 [cs.LG]

[3] Yuang Jiang, Shiqiang Wang, Victor Valls, Bong Jun Ko, Wei-Han Lee, Kin K. Leung, and Leandros Tassiulas. 2020. Model Pruning Enables Efficient Federated Learning on Edge Devices. arXiv:1909.12326 [cs.LG]

[4] Ashish Kumar, Saurabh Goyal, and Manik Varma. 2017. Resource-efficient Machine Learning in 2 kB RAM for the Internet of Things. In *Proceedings of the 34th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 70)*, Doina Precup and Yee Whye Teh (Eds.). PMLR, http://proceedings.mlr.press/v70/kumar17a.html, 1935–1944. http://proceedings.mlr.press/v70/kumar17a.html

[5] Haoyu Ren, Darko Anicic, and Thomas Runkler. 2021. TinyOL: TinyML with Online-Learning on Microcontrollers. arXiv:2103.08295 [cs.LG]

[6] Fouad Sakr, Francesco Bellotti, Riccardo Berta, and Alessandro De Gloria. 2020. Machine Learning on Mainstream Microcontrollers. *Sensors* 20, 9 (2020), 1–25. https://doi.org/10.3390/s20092638

[7] M. Satyanarayanan. 2017. The Emergence of Edge Computing. *Computer* 50, 1 (Jan 2017), 30–39. https://doi.org/10.1109/MC.2017.9

[8] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. 2019. Federated Machine Learning: Concept and Applications. *ACM Trans. Intell. Syst. Technol.* 10, 2, Article 12 (Jan. 2019), 19 pages. https://doi.org/10.1145/3298981