electronics

Article

On-Device Training of Machine Learning Models on Microcontrollers with Federated Learning

Nil Llisterri Giménez, Marc Monfort Grau, Roger Pueyo Centelles and Felix Freitag *

Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya (UPC), 08034 Barcelona, Spain; nil.llisterri@estudiantat.upc.edu (N.L.G.); marc.monfort@upc.edu (M.M.G.); roger.pueyo@upc.edu (R.P.C.)
* Correspondence: felix.freitag@upc.edu

Abstract: Recent progress in machine learning frameworks has made it possible to now perform inference with models using cheap, tiny microcontrollers. Training of machine learning models for these tiny devices, however, is typically done separately on powerful computers. This way, the training process has abundant CPU and memory resources to process large stored datasets. In this work, we explore a different approach: training the machine learning model directly on the microcontroller and extending the training process with federated learning. We implement this approach for a keyword spotting task. We conduct experiments with real devices to characterize the learning behavior and resource consumption for different hyperparameters and federated learning configurations. We observed that in the case of training locally with fewer data, more frequent federated learning rounds more quickly reduced the training loss but involved a cost of higher bandwidth usage and longer training time. Our results indicate that, depending on the specific application, there is a need to determine the trade-off between the requirements and the resource usage of the system.

Keywords: machine learning; keyword spotting; embedded systems; federated learning

check for updates

Citation: Llisterri Giménez, N.; Monfort Grau, M.; Pueyo Centelles, R.; Freitag, F. On-Device Training of Machine Learning Models on Microcontrollers with Federated Learning. *Electronics* **2022**, *11*, 573. https://doi.org/10.3390/electronics11040573

Academic Editors: Kah Phooi Seng and Akash Kumar

Received: 23 December 2021
Accepted: 3 February 2022
Published: 14 February 2022

1. Introduction

We can observe an evolution of machine learning (ML) being implemented into ever smaller computing devices. Not too long ago, machine learning applications moved from cloud data centers into Single-Board Computers (SBCs), such as the Raspberry Pi. This tendency has only become stronger, and nowadays, machine learning applications have been brought even into cheap microcontroller boards, which have been coined Tiny Machine Learning (TinyML) [1].

TinyML promises a new opportunity for machine learning at a low cost, in terms of being affordable and having a low material and energy consumption, given that microcontrollers are very cheap compared to higher-end computing devices and have a low material and energy consumption. While TinyML will not replace current high performance AI-based services, it will complement them with machine learning capability within the IoT. This is timely given the growing concern about the energy consumption of machine learning and Artificial Intelligence (AI) in computing devices: it is generally acknowledged that more AI is needed for optimizing resource management everywhere, but the uptake must come at a lower cost in both economic and environmental terms.

Today's most successful machine learning applications with tiny devices focus on doing inference with trained models deployed on microcontrollers [2]. In these applications, the models are trained offline on more powerful computers, applying train-then-deploy design [3]. After training, the tools of the machine learning framework allow optimizing the model with regard to the consumption of resources (e.g., memory usage on the target device). Such optimizations, based on quantization and pruning, may involve a minor loss of accuracy, but the offline training allows leveraging all the potential and facilities of these machine learning frameworks and using large training datasets [4]. 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 [5].

In comparison, executing machine learning model training on the microcontroller itself has limitations and is currently a niche approach only. However, it may better respond to specific needs and enable new application areas. According to [6], training on the microcontroller allows moving 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 the development of machine learning applications, training models directly on tiny devices reduces the need to have access to higher-end computers.

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 [7]. 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 from the local ones. Furthermore, there is the potential of federated learning designs to split a large computing effort into tiny pieces, suitable for many distributed, lower-capacity devices. Federated learning is also seen as a solution to train models involving data that, for privacy reasons, cannot be sent to the cloud, such as medical records [8].

In this paper, we provide new insights extending our previous work [9] on how to train machine learning models on microcontrollers by applying federated learning. The problem we address is that while many research works tackle the theoretical foundations of federated learning, still very little is known about the application of federated learning in on-device training for TinyML. The novelty of our results consists in the demonstration of federated learning applied in embedded devices and the analysis of different parameters for the design of the system. The main contributions are:

- We provide new results on the trade-off between the number of federated learning rounds, resource consumption, the obtained machine learning model accuracy, and the training time.
- We analyze the microcontroller memory usage when the machine learning model is trained on-device with federated learning experiments.
- The experimental results are obtained using real microcontroller boards with reproducible settings, thus facilitating the practical applicability.

The methodology we use to elaborate these contributions is to develop the software and build a prototype to test and analyze different parameters of the design space. For the experimentation with a keyword spotting (KWS) task, we use a real environment with three Arduino Nano 33 BLE Sense boards that implement the federated learning clients and perform model exchange with a central server. The experiments are reproducible, as we created a dataset for the speech samples used in the training, which have been made public along with the open-source code implementation.

2. Background and Related Work

Most of the TinyML solutions currently present in the literature assume that embedded and Internet of Things (IoT) devices only support the inference of ML and deep learning (DL) algorithms, while the training process is performed in more powerful systems (e.g., in a PC or remotely in the cloud), also called off-device training. Popular ML frameworks such as TensorFlowLite (https://www.tensorflow.org/lite/microcontrollers, accessed on 15 November 2021) provide the tools to apply this approach. After training, the model is pruned and quantized to reduce its size [10]. Finally, the optimized model is flashed on the microcontroller board.

In this off-device training approach, it is not possible to modify the model once it has been deployed. This prevents those tiny devices from learning incrementally or directly from the field, to improve accuracy over time, or to adapt to new environmental conditions. In order to provide adaptability, Disabato and Roveri [11] introduced an incremental algorithm based on transfer learning and *k*-nearest neighbor to support not only inference but also on-device learning of ML and DL solutions on embedded devices and IoT units. For hardware, the authors used a Raspberry Pi 3B+ and an STM32F7 board with 512 kB of RAM and 2 MB of flash, which has a larger computing capacity than the Arduino Nano 33 BLE Sense that we use in our work. The authors showed the feasibility with two off-the-shelf hardware platforms via image and audio benchmarking; however, their approach required the availability of the pre-trained feature extractor.

Large neural networks can have a huge number of parameters that require lots of memory from the computing infrastructure to store the model and CPU capacity to do the inference. It is clear that for machine learning models running on microcontrollers, the approach must be different, since computing power, memory, and energy are typically very restricted. Therefore, instead of training big and general-purpose models, the opportunity seems to be to develop models for specific needs. In the work of Fedorov et al. [12], progress in this direction has been shown with pruning techniques that reduce the memory consumption of a convolutional Neural Network (CNN) in order to fit into microcontrollers. The application of the proposed technique, however, is a step in the toolchain that takes place off-device, i.e., before flashing the optimized model on the microcontroller, and not when the model is trained on the device.

Many TinyML applications have been developed addressing different purposes [13]. In a typical application, depending on the sensors available to the device, trained machine learning models process individual or combined inputs from audio, video, or motion signals. Speech and image recognition are popular for end user and consumer applications. Gesture recognition and pose estimation applications leverage the device's motion and orientation sensors. Besides being of interest for consumers, this type of applications also finds use in the health domain. Ecological conservation and wildlife classification is another area where embedded systems with machine learning capacity are used [14]. For the industrial sector, predictive maintenance is a very important application field. For structural health monitoring [15], for instance, TinyML solutions can become very relevant, since power supply is often not available to these type of installations, such that low-energy-consuming microcontrollers can be an option. Not only in buildings but also in machines, the vibration sensors in combination with a trained machine learning model in a networked microcontroller board may detect anomalies and become part of a larger IoT application.

Transfer learning in combination with federated learning is explored in the work of Kopparapu et al., who proposed TinyFedTL [16]. The advantage of transfer learning is that only a subset of the total layers of the neural network is trained. As such, transfer learning is principally relevant for resource-constraint environments such as microcontrollers, since it saves computing resources. The technique takes advantage of the training already performed on previous models to produce a new one. Usually, the weights and biases of the first layers are reused, since they recognize the most basic patterns of the data. The new model only has to train the layer closer to the output, which is responsible for recognizing the most specific patterns. When applying this approach in TinyFedTL, only the layer connected to the output layer was trained with backpropagation for an image recognition task, where the previous layers were obtained from a compressed version of TensorFlow's MobileNet. Similarly to in our work, Kopparapu et al. used the popular Arduino Nano 33 BLE Sense board. However, unlike our work, instead of federated transfer learning, we consider the situation where a neural network model is trained from scratch on the device by means of federated learning.

Federated learning on higher-end edge devices such as Android phones, Raspberry Pi, and NVIDIA Jetson was presented in the work of Mathur et al. using the so-called

Flower framework [17]. Given the computing capacity of these devices, the federated learning client for the Android phones was implemented in Java with specific TensorFlow Lite Model Personalization support for Android Studio. The federated learning client for the Raspberry Pi and NVIDIA Jetson was implemented in Python. The model training was done on-device with federated learning, for which TensorFlow Lite was run on the boards. This is a different situation from doing machine learning in much more resource-constraint embedded systems such as the Arduino Nano 33 BLE, where in order to benefit from TensorFlow Lite, the models must be trained off-device and only after training the optimized static model is flashed to the device for inference [18].

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 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 found [19]. The microcontroller of the Arduino Nano 33 BLE Sense board we use in this work does have an FPU unit, which enables the training on this device.

In comparison with the above works, the one nearest to ours is the work of Kopparapu et al., which is the only related work that has also applied federated learning on microcontroller boards. However, Kopparapu et al. do on-device federated transfer learning, and their evaluation combines memory and time measurements at the device with simulations for the performance understanding. In contrast, in our work, we conduct experiments on complete model training with on-device federated learning using three boards and could gain new insights on the trade-offs in the federated learning design space in relation to the resource constraints of the microcontroller boards.

3. Application Development

This section describes the development of the KWS application to study the training of a neural network model on the microcontroller. The application shall be able to recognize different keywords, which will be decided by the user when starting to train the model. The user shall be able to switch to an inference mode to test the keyword detection with the trained models.

3.1. Hardware Setup

For the user 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 is to be used to record the keywords. It also has an integrated white LED and an RGB one. The white LED is used to visualize the application state (e.g., IDLE, busy), and the RGB LED is used to show the output class of the keyword spotting model. To train the model with up to three keywords, we connect three buttons to the digital inputs of the board. Each button allows for training one of the keywords. A fourth button is added for testing the model (i.e., to perform inference). Currently, the detected keyword notifies the user by illuminating the color corresponding to the keyword on the RGB LED of the Arduino board. Figure 1 shows how the Arduino board is connected with the buttons.

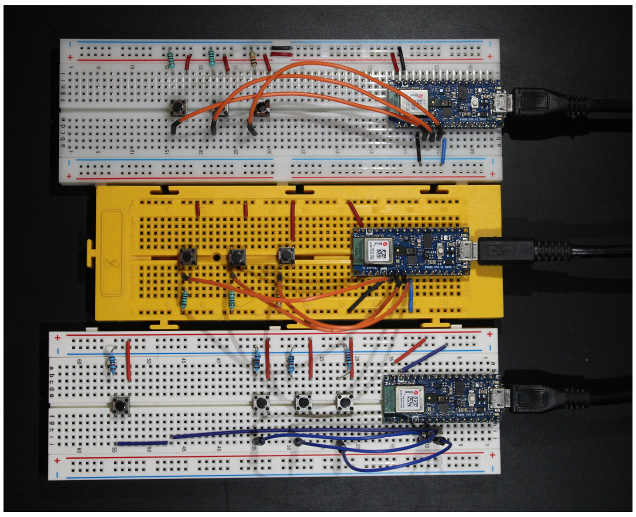


Figure 1. The Arduino Nano 33 BLE Sense board and the buttons on a breadboard to control the training and inference process. The hardware for three federated learning clients is shown.

3.2. Feature Extraction

In order to train the model, a feature vector needs to be obtained. Just after releasing any of the buttons, the microphone starts recording for one second. The recording applies a sampling rate of 16 kHz, which corresponds to the best practice value used in other works on audio recognition tasks with TinyML. Each value is represented by a 16 b signed integer. Therefore, the recorded audio has a size of 32 kB. An increase in the sampling rate would require considering the new memory requirements of the audio signal, given the memory constraints of a TinyML device. Given the recorded audio, we obtain the features that will be used to train the model. A popular way to extract features from a human voice is using the Mel Frequency Cepstral Coefficientss (MFCCs). The board computes the MFCCs (with 13 coefficients) for the recorded audio and obtains a spectrogram of 13 rows and 50 columns. This spectrogram represents the feature vector used to train the model.

3.3. Neural Network Model

Our application aims to train a neural network model on a microcontroller. For that purpose, the model needs to be small enough to fit into the memory of the microcontroller of the Arduino board, thus excluding complex models with many parameters that require the resource availability of higher-end devices. That being said, the model used in our application is a feed-forward neural network with a single hidden layer of 25 nodes by default. The effect of the size of the hidden layer is a parameter that we evaluate in more detail in Section 4.4. The input layer has 650 nodes, the same number as the size of the feature vector obtained from the MFCCs (13×50). The size of the output layer corresponds to the number of keywords and has three nodes. This neural network architecture gives a sum of 16,325 weights and 28 biases.

The data type used to represent the weights and biases is 4 B floats (the maximum float precision allowed by the Arduino board). Therefore, the model has a final size of 65,412 B. These bytes are not stored on the slow Arduino flash memory (1 MB) because they are constantly modified during the training phase, so they are stored in the RAM. This is not an issue in the Arduino Nano 33 BLE Sense board, as it has 256 kB of SRAM.

3.4. Model Training

The model is trained using an online learning approach. As mentioned, the model is initialized every time the application is restarted. In interactive mode, 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 improve 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 0.6. This value is quite high when compared with values used for the training of more complex models; however, from experimenting with different values of the learning rate, we found this value to be suitable [9]. Nevertheless, 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.

3.5. Workflow Interactive Application

The application starts when the program is flashed to the Arduino board (or the board is restarted with the program already flashed). In our implementation, 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 her own model using any of the three training buttons. Each button allows training one keyword. When one of the training buttons 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. Figure 2 shows the diagram of the interactive application workflow.

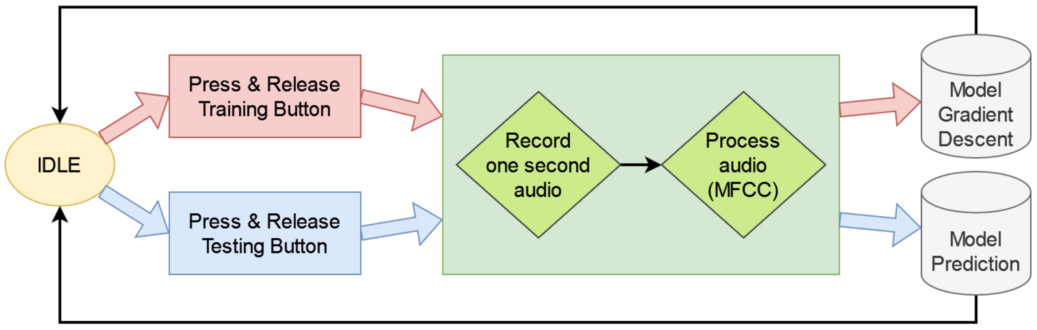


Figure 2. Workflow diagram for interactive on-device training of a single board.

3.6. Software Implementation

We implement the keyword spotting application for the federated learning components shown in Figure 3. This architecture consists of a central server and several clients, as well as their interactions. The central server calculates the model averaging with the received local models after a determined number of training epochs at the client nodes. The model averaging leads to a new global model. The updated global model is sent back to the client nodes, which train their local model again with the data available at each node.

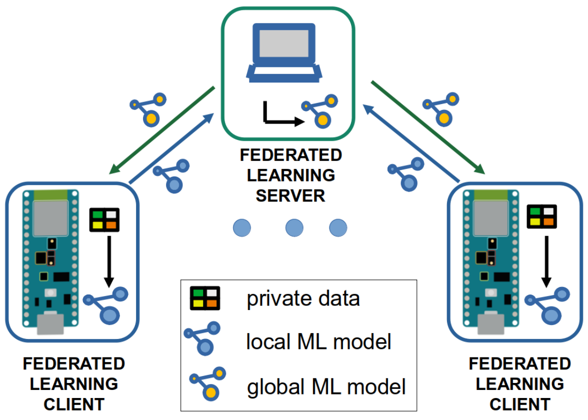

FEDERATED LEARNING SERVER
FEDERATED LEARNING CLIENT
FEDERATED LEARNING CLIENT
private data
local ML model
global ML model

Figure 3. Implemented federated learning components and their interaction.

We implemented the federated learning server in Python. The server can run on a PC. It communicates with the client nodes over serial ports, spawning different threads to handle the communication with each client device in parallel. It sends and receives model data from the client 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 (http://robotics.hobbizine.com/arduinoann.html, accessed on 15 November 2021) and code from EdgeImpulse (https://www.edgeimpulse.com/, accessed on 15 November 2021) 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 software implementation of the application used in this work is publicly available at https://github.com/NilLlisterri/TinyML-FederatedLearning, accessed on 15 November 2021.

3.7. Federated Learning Training

Figure 4 shows the principal diagram of the federated learning workflow. After the configuration, the server creates a neural network model and initializes it with random parameters. This model is sent to all the clients. As soon as the clients receive the model, they start training it with their local data. The local data are generated by the user saying the keywords. In the independent and identically distributed (IID) data case, all the clients must use the same keywords. Meanwhile, the server starts a timer and sets a ten-second countdown for the first federated learning round. When the countdown ends, the server tries to connect with all the clients and asks them to send their trained model. The clients have five seconds to accept the request, or they will be discarded for this iteration. The models that are received from the connected clients are aggregated to create a new global model. This global model will be sent back to the connected clients to be trained again, and the server will restart the countdown for the next round.

The centralized federated learning approach requires frequent communication and high bandwidth. The Python server uses the pySerial library to communicate and exchange data with the Arduino boards. When the server starts, it first opens a communication channel through the ports used by the clients. Then, using the open channels, bytes can be received and sent. The bytes received at the server are the bytes sent by the Arduino through the serial output buffer. The bytes sent from the server are received at the Arduino serial input buffer. The input buffer of the Arduino only holds 64 B. The pySerial input buffer can hold as many bytes as the RAM allows.

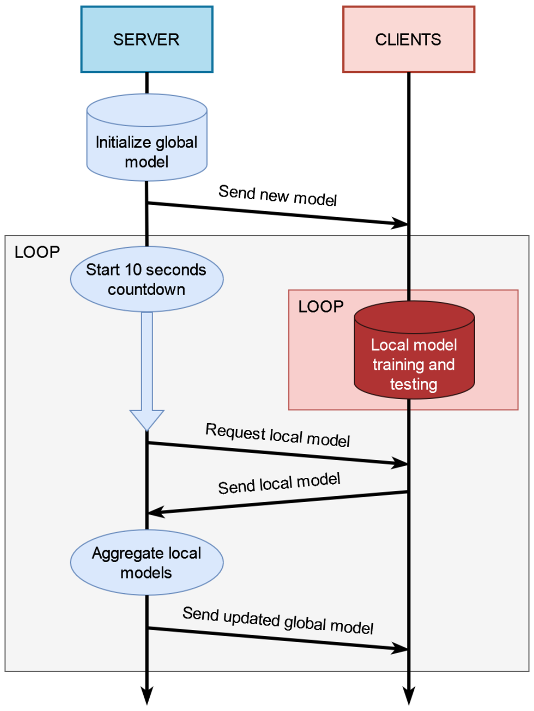


Figure 4. Sequence of interactions between server and clients in the federated learning implementation.

Sending the models back and forth sequentially from the clients in the Arduino boards to the federated learning server on the PC proved to be very inefficient when the number of boards increased. Therefore, we applied Python's threading library at the server in order to send and receive the clients' models in parallel. When the server sends the initial model, receives the devices' models, or sends the updated ones, one thread is started for each device to handle the transmission.

For the purpose of better controlling the experimentation, we also implemented a few additional features in the server and clients: instead of using the timer at the server to control the federated learning rounds, we implemented this control according to an experiment configuration. For each experiment, the server reads a configuration that specifies the number of samples used for the training at each client in each round and other hyperparameters. In order to send these control data to the client, additional messages were created.

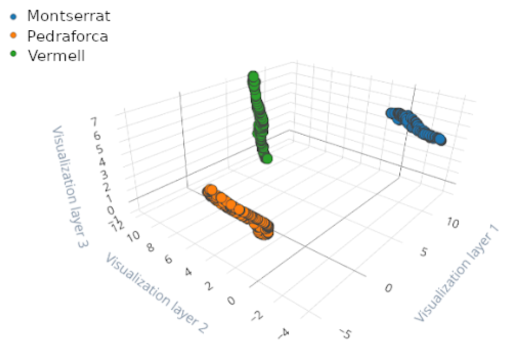
To avoid variations in the training data, we introduced the possibility that the client received the raw speech signals for the training samples (pre-recorded previously with the client device's microphone) through a message from the server, instead of using the board's microphone as input. This way, the local training for federated learning is performed at the client device as if the speech sample was received through the microphone, but taking the speech from a previously created dataset (see Section 4.1) allows the reproducibility of the obtained results.

4. Experimental Results

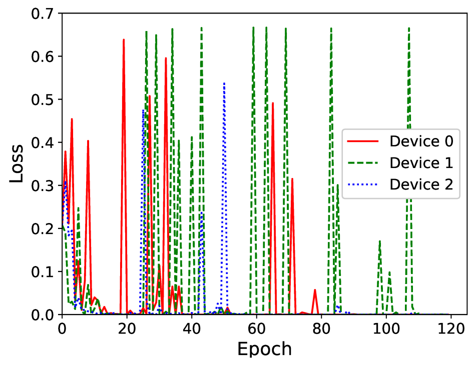
The experiments presented in this section are divided into two sets. First, the experiments focus on the performance of on-device training of machine learning models, considering different system configurations. The second group of experiments also tackles on-device model training, now in combination with federated learning.

4.1. Dataset Creation

Before starting the experimentation, we create a dataset with several voiced keywords, so that the same data can be used through all the experiments. The spoken keywords are captured with the microphone of an Arduino Nano 33 Bluetooth Low Energy (BLE) board, following the push buttons training procedure described in Section 3.4. The recording of the keywords took place in a quiet room in a city location. The keywords were spoken close to the microphone, but the samples may have background noise from a nearby street. For the experiments in following Sections 4.2–4.4, there were two keywords chosen: *Montserrat* and *Pedraforca* (both after two iconic mountains in Catalonia). For the experiment on non-IID data later, in Section 4.5, we add a third keyword. Raw data from the captured speech signal are transmitted from the Arduino board to the PC, where the speech samples are collected. Repeating this procedure, we create a dataset consisting of 180 samples for each of the keywords ,(the dataset of keywords is publicly available at https://github.com/NilLlisterri/TinyML-FederatedLearning/tree/master/datasets, accessed on 30 November 2021). Figure 5a shows the separation of the keywords visually in a three-dimensional space, obtained using UMAP, to reduce the 650 features obtained with MFCC to only 3 artificial features.



(**a**)



(**b**)

Figure 5. On-device training of the two keywords Montserrat and Pedraforca, without federated learning. (**a**) Separation of the three keywords. (**b**) Loss vs. epoch during training (1 sample for 1 epoch).

4.2. Model Training without Federated Learning

In the first experiment, we aimed to assess on-device training performance, using the dataset for keywords *Montserrat* and *Pedraforca*. Each board trains its own model independently without exchanging any information. We used the three boards shown in Figure 1 and randomly divided the total number of 360 samples for both keywords among the three boards. Hence, each board received 120 samples for training, containing 60 samples of each of the keywords. We applied a learning rate of 0.6 and a momentum of 0.9. The hidden layer size was 25 neurons. During the training process, the keywords are presented as inputs to the models in alternate order.

The evolution of the model training process can be seen in Figure 5b. There, we can observe an overall trend indicating that the training error decreases as the number of training epochs grows.

4.3. Evaluation of the Number of Federated Learning Rounds

In this set of experiments, we aim to assess the influence that the number of federated learning rounds has on the training of the models. We perform several repetitions, each time using a different number of rounds, ranging from very frequent federated learning rounds to very infrequent ones.

The effect of executing a federated learning round every 2 and every 30 local training epochs can be seen, respectively, in Figure 6a,b. As it can be observed there, performing federated learning every 2 epochs reduces the training loss much earlier than doing it more infrequently, i.e., every 30 epochs. This result shows a trend coherent with the observations from experiments in our previous work, in which two clients conducted federated learning every 10 local training epochs [9].

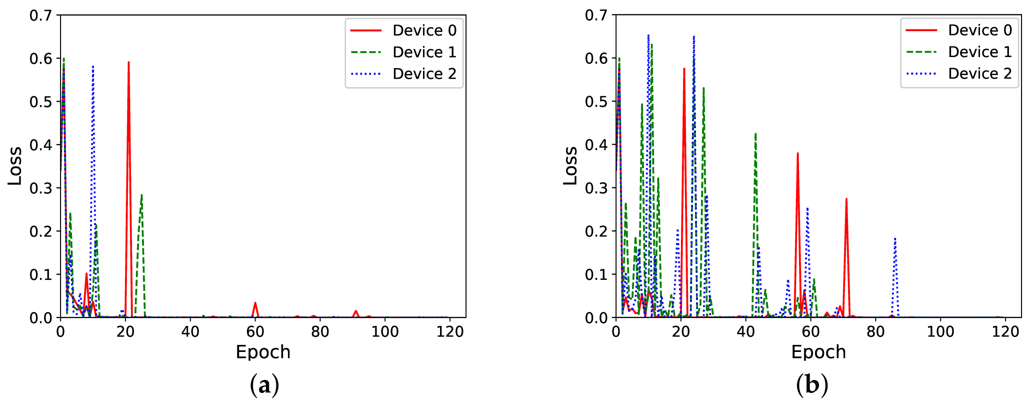


Figure 6. Loss vs. epochs during training for frequent and infrequent federated learning rounds. (**a**) Federated learning round performed every 2 epochs. (**b**) Federated learning round performed every 30 epochs.

During the execution of the experiments, we also measured how the number of federated learning rounds affects the overall training time. Table 1 shows the results for different numbers of rounds. We observe that, for our application, running a federated learning round increases the time of the training process. When doing federated learning, each training round requires using the communication network to transmit the weights of the neural network from the clients to the federated learning server, then computing the averages of the received models at the server, and finally sending the new global model back to the clients through the communication network. While the implemented threading in the server (see Section 3.6) allows the server to receive models from the clients in parallel, most of the additional time spent during each of the federated learning rounds seems to be due to the limited speed provided by the Arduino boards' serial port, used to transmit the neural network model (sized 65 kB, approximately) from the clients to the server and back to them once merged with the others. This totals 130 kB of traffic, for each client, per round. Specific to our experimental setup, we also send data samples from the server to the client. We measured in detail the composition of the time it takes: Sending a data sample always takes 2.49 s. After sending n samples, federated learning takes 7.12 s for one client, 7.85 s for two, and 8.55 s for three clients. For the concrete case of doing a federated learning round every 10 epochs and hence sending 10 data samples, 3 clients, and 120 samples, we obtain a time of $120 \times 2.49\,\mathrm{s} + 12 \times 8.55\,\mathrm{s} = 401.4\,\mathrm{s}$.

Table 1. Federated learning rounds vs. time for training 120 data samples.

Federated Learning Rounds	Training Time
0	309 s
2	320 s
4	336 s
12	401 s
60	777 s
120	1253 s

4.4. Evaluation of the Influence of the Hidden Layer Size

We aim to assess to what extent that model training performance is influenced by the size of the hidden layer of the neutral network. In general, the size of the neural network model is an important parameter for the memory-constraint microcontrollers we are working with. Therefore, in this experiment, we tried to reduce the model's size by using a smaller hidden layer. We conducted this experiment with federated learning training, with a round taking place every 10 local training epochs.

Figure 7 shows the results for this experiment. It can be seen that, for all the four hidden layer sizes chosen, the loss during the training process decreases with the 120 training samples. However, in those experiments with bigger hidden layers, loss is reduced much earlier than in those experiments with smaller ones.

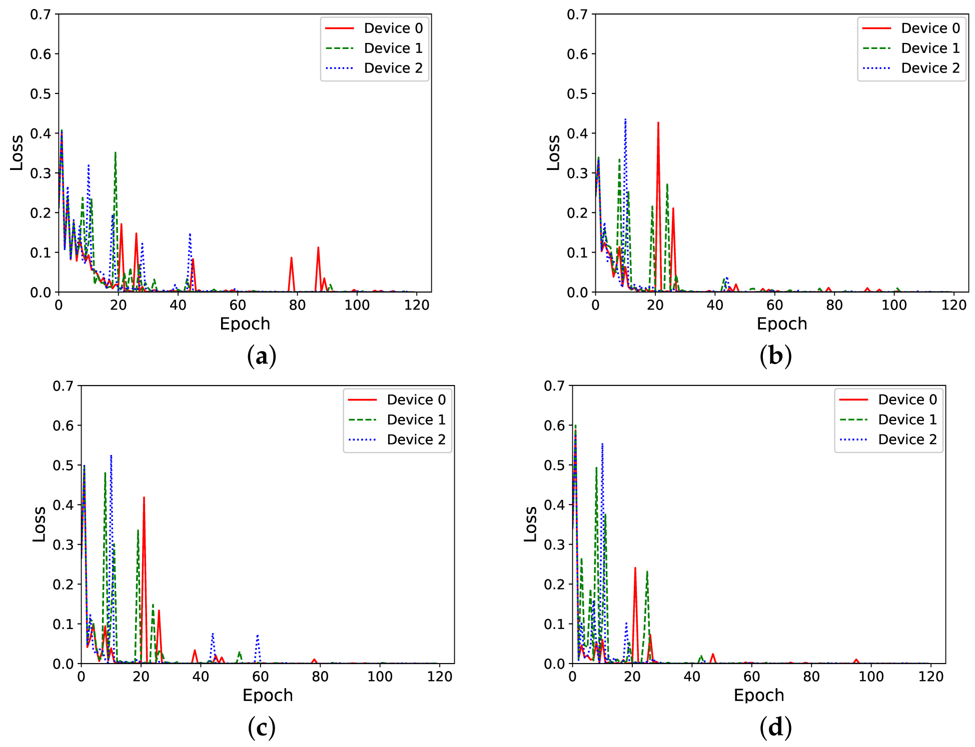


Figure 7. Loss vs. epochs during training for different hidden layer sizes. (**a**) Hidden layer of 5 neurons. (**b**) Hidden layer of 10 neurons. (**c**) Hidden layer of 20 neurons. (**d**) Hidden layer of 25 neurons.

While we can observe that increasing the number of neurons yields better results, in resource-constrained systems such as the Arduino Nano 33 BLE Sense board, we have to be very conscious about the available RAM memory. Analyzing in more detail the memory consumption, we identify the following main memory blocks required by the program running on the device.

- Inference buffer: this array holds one second of audio to run the inference on it. The audio is sampled at 16 MHz and 16 b/sample, so it contains 16,000 values, each one being a 2 B integer. This reserves 32 kB of memory.
- Neural network weights and biases: these parameters are stored as floats. With a hidden layer size of 25 nodes and an output layer of 3 nodes, the neural network amounts to 16,353 values, which use about 65 kB of memory.
- Neural network weights changes: each value is stored as a float to represent the error change in respect to the loss in the previous training, for the gradient descent. This also takes 65 kB of memory.
- Other static variables: both our code and the libraries used have static memory allocation for other smaller data structures, such as the microphone sample buffer, the twiddle coefficients for the Fast Fourier Transformation (FFT), etc. These allocations consume around 48 kB of memory. We also calculate that each new neuron in the hidden layer costs around 5.2 kB of RAM. Therefore, using a hidden layer with 25 neurons, we consume 80, 2 % of the board's RAM, i.e., 210 kB of the 262 kB available.

The memory consumption specific to the speech recognition application is the inference buffer that captures the audio sample and the libraries used to compute the MFCC. Other applications using the input from different sensors do not apply such sampling and will have less memory demand for storing the input data. Then, other components of the machine learning application, such as the neural network model, may be increased while being kept within the available memory of the microcontroller.

Comparing this memory consumption with off-device training, we see that on-device training requires an additional amount of memory corresponding to the size of the neural network model used, which is necessary to store the gradients obtained when doing the backpropagation on the device. Furthermore, with off-device training, techniques such as quantization and pruning can be used additionally, after the training, to obtain a memory-efficient representation of the neural network model.

4.5. Training with Non-IID Data

Last, we aimed to explore the case of model training with non-IID data. To do so, we used the third keyword *vermell*(i.e., red color, in Catalan), in order to train a different keyword on each of the three clients. For the training process, we used a dataset of 120 samples per keyword.

Figure 8 shows that, within very few epochs, each device became quickly trained to the specific keyword (Figure 8a). When federated learning is enabled in the training process (Figure 8b–d, we notice that the new global model received by each client produced a high loss when trained again, locally, with the specific keyword. This observation is consistent with the results obtained in our previous work, in an experiment consisting of only two clients [9]. Figure 8d, which corresponds to the case of doing federated learning after every single epoch, seems to suggest that for these very frequent federated learning rounds, the loss for each local model decreased as more of the training epochs were carried out.

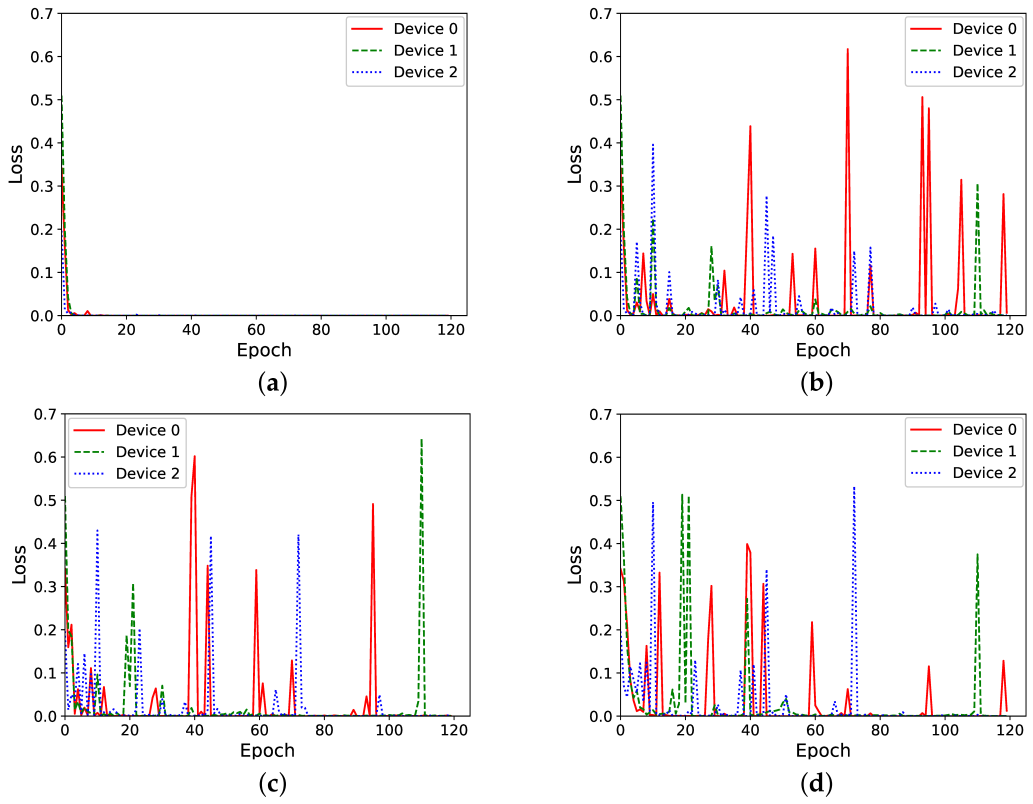


Figure 8. Loss in function of the number of epochs during the non-IID data training process, for different numbers of federated learning rounds. (**a**) One different keyword per client, no federated learning. (**b**) Non-IID data, federated learning every 5 epochs. (**c**) Non-IID data, federated learning every 2 epochs. (**d**) Non-IID data, federated learning every 1 epoch.

5. Discussion

The results obtained from the diverse experiments offer new insights about several aspects of combining on-device training on the microcontrollers with federated learning. As the main result, we could train a global KWS model by aggregating local models trained separately on different microcontroller boards. However, we also identified several issues, which we discuss in the following, as well as directions that could be pursued further.

As shown in the experiments in Table 1, a significant overhead in the training time was produced by the federated learning rounds. A plausible explanation can be found in the fact that the communication protocol we used resulted in a very limited bandwidth link, due to the small size of the Arduino board's USB port buffer (64 B). In order to avoid a buffer overflow, in the current implementation, the server sends only four bytes at once before checking that the board (i.e., the client) has received them. The time required to send our model from the server to the client is approximately 5.5 s, more than three times longer than the time required to send a model from the client to the server (≈1.8 s).

The need for data communication to exchange a model is a feature of federated learning, involving the necessary bandwidth consumption. The frequency of the federated learning rounds, as we investigated in Section 4.3, is a design parameter that has to balance different aspects: Frequent federated learning rounds will imply considerable numbers of data to be interchanged, corresponding to the size of the neural network model, which may not always be feasible due to energy consumption or bandwidth limitations. On the other hand, if the dataset locally available at the client is too small, the quality of the local model may benefit from being trained with federated learning for gaining accuracy but at the cost of bandwidth consumption and energy consumption of the network interface.

Data transmission due to federated learning also has an energy cost and will not be suitable for purely battery-powered IoT devices, considering that the energy consumption of the radio will reduce the life-time of the device. Additional energy supply for the IoT device such as leveraging solar-power could be considered for a more long term operation. Nevertheless, while a model could be trained from scratch, on-device training with federated learning can also be used to occasionally adapt a trained model in an IoT device. In that case, continuous learning will be available to the device, but over long periods of time, the device might just do inference with the trained model and therefore would not require any additional energy consumption for a training process.

With regard to converting the keyword spotting task of the experiment into an application, the inference, i.e., the forward propagation of the speech input over the trained model, must be changed to be done continuously. Currently, for the experiment, the inference phase is specifically indicated to the device through a test button on the breadboard (Figure 1). Changing to a continuous inference, however, does not seem to be a computational challenge for the Arduino Nano 33 BLE Sense with our neural network model, as we measured that the forward path was computed in around 3–4 ms (while for training the backward propagation took 9–10 ms).

An embedded device may have one or more wireless network interfaces, such as Wi-Fi (e.g., available in the Arduino Portenta H7), Bluetooth (including BLE, like the Arduino boards used), or LoRa. Each of the available technologies may be suitable for a given application, depending on the required transmission distance, the data payload size, the energy budget, the spectrum regulation, etc. LoRa, for instance, allows IoT devices for communication over several kilometers. However, its low throughput (typically a few kb/s) and the eventual duty cycle regulations make such a link layer technology a bottleneck for exchanging machine learning models among devices to perform federated learning. Nonetheless, researchers have found ways to increase the available bandwidth and, therefore, the number of data that a LoRa-enabled device can transmit [20]. Another important aspect to consider is that embedded devices working as clients in a federated learning may be unreliable. For instance, they commonly use less powerful radio communication chips (e.g., power-constraint, single-stream Wi-Fi with an omnidirectional antenna) and usually depend on batteries. As a consequence, they may drop out from the training phase more frequently than clients in higher end devices working in other environments (e.g., a datacenter cloud).

Lastly, in real environments, the local data at the clients of a federated learning network will not always fulfill the conditions of being independent and identically distributed (IID). Even if the hardware used by the clients was exactly the same, the sensors that capture the training data may simply be placed in different local environments with their particular acoustic characteristics, background noise, etc., which influence the training data. Furthermore, if different sensors are used by the same client devices, the data captured will be influenced by the specific sensor used (e.g., directivity and sensitivity of the microphone, quality of the camera). Furthermore, the numbers of local data available for the training at each device may be heterogeneous. These practical considerations suggest looking deeper into federated learning solutions for non-IID data, which seems to be representative of real-life situations.

6. Conclusions and Future Work

This paper presented the experimental work undertaken with a keyword spotting (KWS) application that performs on-device training with federated learning of a neural network model, using Arduino Nano 33 BLE Sense boards. The work gives practical insights into how on-device training can be implemented and analyzes the performance issues encountered when combining model training and federated learning on a recent microcontroller board. Training is enabled in a supervised learning mode: the user can train KWS spotting by speaking words to the board's microphone and, by means of the push buttons connected to the digital inputs, indicating the corresponding label to each of the keywords voiced.

We evaluated the loss function in the forward path of the training process, and we observed how the machine learning model trained on the device improves during the training after tens of spoken words. Afterwards, the experimentation was extended to perform the training in a federated learning scenario, with a central server and three client nodes. Regarding the frequency of the federated learning rounds, we observed that more frequent rounds more quickly reduced the training loss, but at a cost of higher bandwidth usage and longer training time. In the comparison of different hidden layer sizes, our experiments showed that more training samples were useful when the models had a smaller hidden layer size. In the training with non-IID data, we observed that very frequent federated learning rounds were useful in order to reduce the loss of the local model.

Future work will further explore the raised scenario of non-independent and identically distributed (IID) data and federated learning training to more deeply understand its capacity to train versatile models if there are only limited local training data at the clients. Furthermore, for integrating on-device federated learning in IoT applications, we will address the trade-offs of the different network interfaces available at an embedded device with regard to the energy and bandwidth consumption.

Author Contributions: Conceptualization, N.L.G., M.M.G., R.P.C. and F.F.; methodology, N.L.G., M.M.G., R.P.C. and F.F.; software, N.L.G. and M.M.G.; validation, N.L.G.; formal analysis, N.L.G. and M.M.G.; investigation, N.L.G., M.M.G., R.P.C. and F.F.; resources, N.L.G. and F.F.; data curation, N.L.G.; writing—original draft preparation, N.L.G., M.M.G., R.P.C. and F.F.; writing—review and editing, N.L.G., R.P.C. and F.F.; visualization, N.L.G. and F.F.; supervision, R.P.C. and F.F.; project administration, F.F.; funding acquisition, F.F. All authors have read and agreed to the published version of the manuscript.

Funding: This work has received funding from the Spanish Government under contracts PID2019-106774RB-C21, PCI2019-111851-2 (LeadingEdge CHIST-ERA), PCI2019-111850-2 (DiPET CHIST-ERA).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Acknowledgments: We acknowledge the support of this work by the Spanish Government under contracts PID2019-106774RB-C21, PCI2019-111851-2 (LeadingEdge CHIST-ERA), PCI2019-111850-2 (DiPET CHIST-ERA).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Banbury, C.R.; Reddi, V.J.; Lam, M.; Fu, W.; Fazel, A.; Holleman, J.; Huang, X.; Hurtado, R.; Kanter, D.; Lokhmotov, A.; et al. Benchmarking TinyML Systems: Challenges and Direction. *arXiv* **2021**, arXiv:2003.04821.
2. Sakr, F.; Bellotti, F.; Berta, R.; De Gloria, A. Machine Learning on Mainstream Microcontrollers. *Sensors* **2020**, *20*, 2638. [CrossRef] [PubMed]
3. Ravaglia, L.; Rusci, M.; Nadalini, D.; Capotondi, A.; Conti, F.; Benini, L.; Benini, L. A TinyML Platform for On-Device Continual Learning with Quantized Latent Replays. *IEEE J. Emerg. Sel. Top. Circuits Syst.* **2021**, *11*, 789–802. [CrossRef]
4. Jiang, Y.; Wang, S.; Valls, V.; Ko, B.J.; Lee, W.H.; Leung, K.K.; Tassiulas, L. Model Pruning Enables Efficient Federated Learning on Edge Devices. *arXiv* **2020**, arXiv:1909.12326.
5. Satyanarayanan, M. The Emergence of Edge Computing. *Computer* **2017**, *50*, 30–39. [CrossRef]
6. Ren, H.; Anicic, D.; Runkler, T. TinyOL: TinyML with Online-Learning on Microcontrollers. *arXiv* **2021**, arXiv:2103.08295.
7. Yang, Q.; Liu, Y.; Chen, T.; Tong, Y. Federated Machine Learning: Concept and Applications. *ACM Trans. Intell. Syst. Technol.* **2019**, *10*, 1–19. [CrossRef]
8. Choudhury, O.; Divanis, A.; Salonidis, T.; Sylla, I.; Park, Y.; Hsu, G.; Das, A. Differential Privacy-enabled Federated Learning for Sensitive Health Data. *arXiv* **2020**, arXiv:1910.02578.
9. Monfort Grau, M.; Pueyo Centelles, R.; Freitag, F. On-Device Training of Machine Learning Models on Microcontrollers with a Look at Federated Learning. In Proceedings of the Conference on Information Technology for Social Good, GoodIT '21, Rome, Italy, 9–11 September 2021; Association for Computing Machinery: New York, NY, USA, 2021; pp. 198–203. [CrossRef]
10. Choi, J.; Wang, Z.; Venkataramani, S.; Chuang, P.I.J.; Srinivasan, V.; Gopalakrishnan, K. PACT: Parameterized Clipping Activation for Quantized Neural Networks. *arXiv* **2018**, arXiv:1805.06085.
11. Disabato, S.; Roveri, M. Incremental On-Device Tiny Machine Learning. In Proceedings of the 2nd International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things, AIChallengeIoT '20, Virtual Event, 16–19 November 2020; Association for Computing Machinery: New York, NY, USA, 2020; pp. 7–13. [CrossRef]
12. Fedorov, I.; Adams, R.P.; Mattina, M.; Whatmough, P.N. SpArSe: Sparse Architecture Search for CNNs on Resource-Constrained Microcontrollers. In Advances in Neural Information Processing Systems 32, Proceedings of the Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, Vancouver, BC, Canada, 8–14 December 2019; Wallach, H.M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E.B., Garnett, R., Eds.; pp. 4978–4990. Available online: https://papers.nips.cc/paper/2019 (accessed on 30 November 2021).
13. Ray, P.P. A Review on TinyML: State-of-the-art and Prospects. *J. King Saud Univ. Comput. Inf. Sci.* **2021**. [CrossRef]
14. Yin, M.S.; Haddawy, P.; Nirandmongkol, B.; Kongthaworn, T.; Chaisumritchoke, C.; Supratak, A.; Sa-ngamuang, C.; Sriwichai, P. A Lightweight Deep Learning Approach to Mosquito Classification from Wingbeat Sounds. In Proceedings of the Conference on Information Technology for Social Good, GoodIT '21, Rome, Italy, 9–11 September 2021; Association for Computing Machinery: New York, NY, USA, 2021; pp. 37–42. [CrossRef]
15. Arcadius Tokognon, C.; Gao, B.; Tian, G.Y.; Yan, Y. Structural Health Monitoring Framework Based on Internet of Things: A Survey. *IEEE Internet Things J.* **2017**, *4*, 619–635. [CrossRef]
16. Kopparapu, K.; Lin, E. TinyFedTL: Federated Transfer Learning on Tiny Devices. *arXiv* **2021**, arXiv:2110.01107.
17. Mathur, A.; Beutel, D.J.; de Gusmão, P.P.B.; Fernandez-Marques, J.; Topal, T.; Qiu, X.; Parcollet, T.; Gao, Y.; Lane, N.D. On-device Federated Learning with Flower. *arXiv* **2021**, arXiv:2104.03042.
18. Kwon, J.; Park, D. Toward Data-Adaptable TinyML Using Model Partial Replacement for Resource Frugal Edge Device. In Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, HPC Asia 2021, Virtual Event, 20–22 January 2021; Association for Computing Machinery: New York, NY, USA, 2021; pp. 133–135. [CrossRef]
19. Kumar, A.; Goyal, S.; Varma, M. Resource-efficient Machine Learning in 2 kB RAM for the Internet of Things. In Proceedings of the 34th International Conference on Machine Learning (ICML'17), Sydney, NSW, Australia, 6–11 August 2017; JMLR: Brookline, MA, USA, 2017; Volume 70, pp. 1935–1944. Available online: http://proceedings.mlr.press/v70/kumar17a.html (accessed on 30 November 2021).
20. Altayeb, M.; Zennaro, M.; Pietrosemoli, E.; Manzoni, P. TurboLoRa: Enhancing LoRaWAN Data Rate via Device Synchronization. In Proceedings of the 2021 IEEE 18th Annual Consumer Communications Networking Conference (CCNC), Las Vegas, NV, USA, 9–12 January 2021; pp. 1–4. [CrossRef]