Information Fusion 104 (2024) 102189

Contents lists available at ScienceDirect

Information Fusion

journal homepage: www elsevier com/locate/inffus

Federated learning for IoT devices: Enhancing TinyML with on-board training

M. Ficco [a], A. Guerriero [b], E. Milite [a], F. Palmieri [a,*], R. Pietrantuono [b], S. Russo [b]

[a] Università degli Studi di Salerno Salerno Italy
[b] Università degli Studi di Napoli Federico II Napoli Italy

ARTICLE INFO

Keywords:
Internet of things
Federated learning
Transfer learning
TinyML

ABSTRACT

The spread of the Internet of Things (IoT) involving an uncountable number of applications, combined with the rise of Machine Learning (ML), has enabled the rapid growth of pervasive and intelligent systems in a variety of domains, including healthcare, environment, railway transportation and Industry 4.0. While this opens up favorable scenarios, it also raises new challenges. The huge amount of data collected and processed by ML applications requires efficient and scalable solutions that contrast with the constrained capabilities of IoT devices as for memory, power consumption, processing and network bandwidth. The TinyML technologies foster the adoption of ML algorithms running locally on IoT devices. However, they typically foresee a remote training process (e.g., on cloud servers) combined with local inference – a strategy not always viable, e.g., for privacy and security issues.

We present a technique to enable the on-board training of ML algorithms on IoT devices, through the combination of federated learning (FL) and transfer learning (TL). We experimentally analyze it in classification and regression problems, comparing it to traditional FL solutions, as well as with a consolidated technique based on Tensorflow Lite. Results show that FL with TL reaches accuracy values better than FL without TL in both classification (86.48%) and regression (0.0201). These results are comparable with a model trained on the full dataset. We further analyze training and inference time and power consumption on various devices. Finally, we evaluate how the performance changes with unbalanced training datasets, showing that although they strongly impact accuracy, FL makes models more robust, letting them achieve accuracy comparable to when trained on balanced datasets.

1. Introduction

The Internet of Things (IoT) is experiencing a rapid growth thanks to the spread of smart devices. The number of connected devices was estimated to be less than 9 billions in 2019, whereas in 2023 it reached 15 billions and it is expected to reach 29 billions by 2030. Smart devices generate large amounts of data, nowadays often processed by machine learning (ML) algorithms in many important application fields, like healthcare, environment monitoring and industrial control. In 2019, devices collected less than 20 zettaBytes of data; in 2025 the data volume associated with IoT devices is forecast to reach 79.4 zettaBytes [1].

The implementation of ML-based IoT applications typically - e.g., in the Internet of Medical Things [2] - requires that the collected data is sent for processing to central cloud servers. This may not be practical due to network limitations, privacy, or security issues related to the application context. An alternative approach could be to process data directly at the edge layer. However, traditional ML approaches are not well suited to resource-constrained IoT devices.

TinyML is an ecosystem around hardware, algorithms, and software platforms - supported by a growing community - that allows running ML models on resource-constrained IoT devices. TinyML finds applications in wide a range of fields: e-health, transportation systems, agriculture and Industry 4.0 [3]. Most current TinyML solutions do not support *on-device model training* [4]. Models are trained on powerful servers and then transferred and put into operation on target devices. Therefore, also such an approach may not be viable in sectors where privacy is a concern. The challenge is to train the model directly on devices, without requiring sensitive data to be transferred over the network, and reducing latency and network bandwidth usage. Solutions in the literature represent mainly proofs of concept of the feasibility of on-device training, or they are just simulated or demonstrated in limited scenarios, and are not mature yet for real contexts.

* Corresponding author.
E-mail address: fpalmieri@unisa.it (F. Palmieri).

https://doi.org/10.1016/j.inffus.2023.102189
Received 30 September 2023; Received in revised form 13 November 2023; Accepted 6 December 2023
Available online 8 December 2023

This work proposes a technique for on-board training and for improvement of the model performance, by combining Federated Learning (FL) and Transfer Learning (TL), which separately have been shown to provide benefits in specific contexts. In FL, multiple models are trained on different datasets, and then "merged" into a unique one [5]. This way, privacy is preserved because data are collected and managed only on local devices. This strategy is here combined with TL to improve accuracy.

The proposed technique is experimentally evaluated for classification and regression tasks, through a supervised training approach. The results are compared with consolidated ML approaches based on TensorFlow Lite. We first evaluate two different FL techniques, the *FedAvg* algorithm proposed by Google, and a simplified version of FL, where nodes first complete the training process, and then transmit the model weights to a central server for aggregation purposes. The latter is useful in scenarios where devices have large differences in performance or poor connectivity. We then evaluate the benefits of combining FL and TL. Several metrics are computed, including training time and power consumption, for various tiny devices. In classification experiments, the model is trained on the *ECG heartbeat* dataset [6]. As for regression, the dataset is *Car trips data log* [7]. Additionally, a comparison is presented of three neural network (NN) libraries in C language, with no external dependencies. We show the optimizations made to suit resource-constrained devices.

The main contributions of the paper are summarized as follows:

A solution combining FL and TL for on-board training of ML algorithms;
An analysis of how performances change in unbalanced vs. balanced datasets;
A comparison of the execution cost of the proposed solution with common tiny devices (including, Arduino WiFi Rev2, ESP8266, ESP32, Arduino MKR1010, Raspberry Pi Zero W, and Raspberry Pi3 B+);
An extensive experimental campaign to evaluate the performance of the proposed solution by comparing it with traditional ML approaches (e.g., based on Tensorflow Lite) on both classification and regression tasks;
A preliminary analysis on continuous improvement of implemented solution accuracy with manual sampling of newly collected data.

The rest of the work is structured as follows. Section 2 provides an overview of edge computing and TinyML. Section 3 introduces the proposed technique to run ML models on resource-constrained devices. Section 4 describes the design of both classification and regression experiments. Section 5 presents results for the classification task, and Section 6 for the regression task. Section 7 shows a comparison against traditional TensorFlow Lite-based approaches. Section 8 reports a preliminary analysis of continuous improvement. Section 9 concludes the paper.

2. Machine learning on embedded devices

2.1. TinyML

Edge computing is a paradigm that scales computations from servers down to locations closer to end users: artificial intelligence models can run directly on devices, making low-latency and real-time predictions. TinyML is a set of edge computing technologies enabling tiny devices, such as microcontrollers, to run ML models. TinyML moves the data mining process to the edge by addressing complications like high latency and privacy violations. Ultimately, it allows edge devices to make autonomous decisions without relying on cloud servers. The main differences between TinyML and Traditional ML (Table 1) concern [4]:

Hardware constraints:

- *Traditional ML*: The models are deployed on high-performance servers;
- *TinyML*: The models are deployed on devices, such as microcontrollers, that are both energy- and resource-constrained, with 1 or 2 CPU cores, limited memory, and no access to more advanced frameworks such as TensorFlow;

Model size and complexity:

- *Traditional ML*: There is no model size limit, and it can use large and complex models due to the absence of computational constraints;
- *TinyML*: It generates compressed and small-size ML models, designed to suit minimal run-time resources (e.g., storable within a few KB of RAM);

Privacy:

- *Traditional ML*: Data is sent over the network and stored in a centralized database;
- *TinyML*: Data and computational tasks are kept on the edge device, minimizing the transfer of sensitive data;

Latency:

- *Traditional ML*: Latency is not a priority, as it is used mainly for offline processing, with data distributed over the network;
- *TinyML*: It supports real-time or near-real-time inference, and decisions can be made directly on the device.

Table 1
TinyML vs. traditional ML.

Criterion	TinyML	Traditional ML
Hardware constraints	Resource-constrained devices, energy saving	Unrestricted
Model size and complexity	Small-size models (few KB)	Unrestricted
Privacy	Only local data	Data sent over the network
Latency	Low latency (local)	Not a priority

Two main research fields in ML on microcontrollers focus on: 1) developing algorithms to convert more efficiently resources and pre-trained NNs to be executed on tiny devices, and 2) improving hardware design and developing more power-efficient hardware that can operate for extended periods on battery power [4]. Currently, TinyML does not support training the model directly on edge devices. As shown in Fig. 1, the models are trained on a computer or server, and then a smaller model is generated that can fit the embedded devices. These compressed models, represented as C arrays, are then flashed onto devices. These can then perform inference using the data collected by the sensors [8]. Many libraries support these steps, including TensorFlow Lite Micro and ST X-CUBE-AI.

TinyML models are designed for inference on low-end devices, but this can be challenging as the training task requires significantly more computational power and memory than the inference task. It is also a time-consuming process that can be energy inefficient and may drain faster the battery life. Models require periodic updates with fresh data collected by the sensors. If the device could learn from local data, the model performance might improve. However, the lack of onboard training support forces us to rely on cloud servers, an operation that is not always possible due to privacy or communication performance concerns.

2.2. Related work

Lo et *al*. [9] presented a literature review in the FL field: the analyzed works do not consider the limitations of FL in resources- and

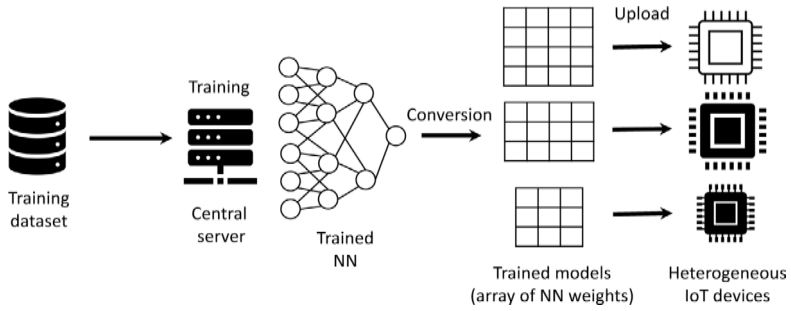

Fig. 1. Traditional model training process.

bandwidth-constrained IoT environments. In particular, most of the literature studied TL through tuning of NN models and privacy-conscious learning; aspects related to NN-based fine-tuning in significantly-constrained tiny development boards used in IoT have not been much investigated [10,11]. In this direction, the few proposed solutions either supported only the deployment of static ML models on tiny devices, or implemented FL solutions in mobile edge networks using at most embedded-Linux devices like Raspberry Pi [12]. TensorFlow Lite, CMSIS-NN, and TVM are libraries developed (from Google, ARM, and Apache, respectively [12–14]) to support ML on tiny-constrained devices. These libraries assume the model is trained in remote servers, and then, uploaded to the tiny device for performing only inference tasks. To update the model with new data, it must be retrained from scratch and re-uploaded to edge devices.

Very few studies discussed the possibility of performing the training process on tiny devices [15,16]. Montiel et al. [17] suggest an approach that does not require training the model on the microcontroller, but focuses on continuous model improvement. An edge node is deployed only for inference, and however, it streams new data to an external server to enhance the model over time. The first two on-board training experiments have been performed in simulation environments without taking into account client heterogeneity, which could lead to slow and unsatisfactory learning progress [18,19]. Ren et al. present a framework that enables incremental on-device training on streaming data, without storing historical data on board. A preliminary implementation has been tested on Arduino Nano 33 BLE Sense.

The first implementation of FL on the resource-constrained real MCU development boards has been presented in [18]. It represents only a proof of concept needed to demonstrate the feasibility of on-device TL with constant storage costs. TinyTrain introduces a methodology for on-device model training [20]. It reduces memory consumption by dynamically selecting layers to be updated and reducing the computation cost of backpropagation. As this solution focuses on complex model training, it suits more powerful boards (like Raspberry Pi) and it is not compatible with most devices. In [21], a modified formulation of the Reptile algorithm is presented [22]. It can be used to implement model-agnostic meta-learning in a federated setting by tiny devices, for finding initial shared models that can quickly adapt by new federated agents. A recent work proposed a tiny FL algorithm to enable the learning of Bayesian classifiers based on distributed tiny data storage. It first performs in parallel by multiple tiny devices and then, a final classification model is computed in a central node, by aggregating the local Bayesian classifiers [23].

This work presents a technique for collaborative training of ML models on board of resource-constrained IoT devices, leveraging a combination of federated learning and transfer learning. An extensive experimental campaign has been performed on heterogeneous real tiny devices, comparing several NN libraries and traditional ML approaches (e.g., Tensorflow Lite) on both classification and regression tasks, using both unbalanced and balanced datasets.

3. Collaborative on board training of ML models for IoT devices

This section describes the proposed technique to enable embedded devices to train a ML model and make inferences. It introduces FL and TL, the two techniques used for high performance collaborative training. A comparison is presented of three neural network libraries (written in C and without the requirement of external dependencies), and the optimizations made to minimize the memory usage and extend the device compatibility. Finally, the devices used to conduct the tests are presented.

3.1. The proposed TinyML-based technique

The proposed technique allows devices with limited memory and processing capabilities to perform training directly on board. It fosters large compatibility for IoT devices. To this aim, the code that runs on the boards is written in C, the most commonly used low-level language. To simulate the real-world case where boards have different resource constraints, a network of heterogeneous devices is used in the experiments.

Fig. 2 provides an overview of the proposed collaborative learning process. It starts by reading the dataset from an external source. Without loss of generality, the dataset is stored in a Comma-Separated Values (CSV) file, e.g. stored on an SD card. Training is done by devices collaboratively through Federated Learning. FL requires an additional server to enable communication between all boards and the exchange of trained model weights. Communication is carried out via MQTT, a lightweight protocol designed for low power consumption and limited bandwidth scenarios. Clearly, the central server is not a resource-constrained device, as it requires proper hardware to handle complex calculations on multiple numerical matrices. Since general-purpose hardware does not have the same constraints as an IoT device, it can use any external system libraries. In the FL experiments, we use Python libraries (see Section 3.2).

When a new node connects to the federated system, it sends a request to the central server to receive the weights of the final model in few seconds. This offers an advantage compared to training the model from scratch, which may take several hours. In traditional TinyML solutions, the neural network is embedded in the microcontroller firmware. The device is ready to do inference, but it will be limited to the NN version loaded. In case the NN is upgraded, the firmware needs to be updated. With the proposed solution, newly added devices will benefit from the most recent model iteration available.

3.2. Federated learning

Federated learning is a modern approach to ML, that tries to solve the problem of training models on distributed systems without the need of directly sharing the data. Each node has its local dataset and aims to collectively train an ML model without exposing the dataset's data to other nodes. The model is not trained on a centralized single server, but on the edge devices, preserving privacy and security.

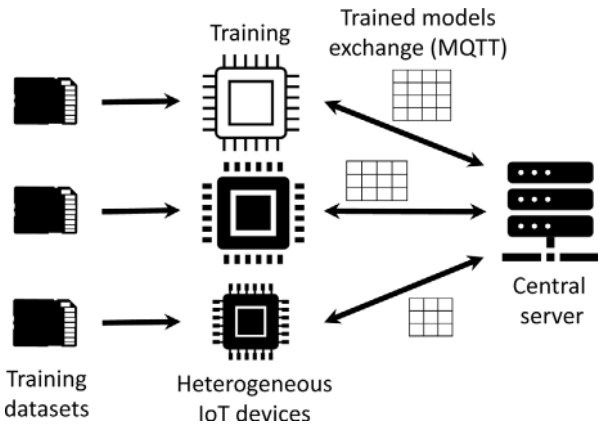

Fig. 2. Proposed on-board collaborative training process.

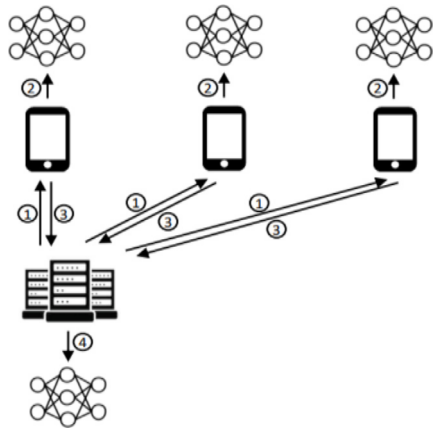

Fig. 3. Aggregation steps of the Federated Averaging (FedAvg) algorithm.

The goal of FL is to obtain a model performance that is similar to or better than the model trained through traditional training methods. Thanks to the nature of the distributed system and using the local data owned by each device, it is possible to train models on different and heterogeneous datasets. The result is a more robust and generalized model. More formally, let us consider N nodes, each characterized by its own dataset D_i, its local model λ_i and its weights vector $\mathbf{w_i}$, where λ_i is a function describing how weights $\mathbf{w_i}$ conform to the local dataset D_i used for training. Once achieving consensus on $\mathbf{w_i}$, we can obtain a global model Λ by aggregating the models trained locally as:

$$\Lambda(\mathbf{w_1} \ \dots \ \mathbf{w}) = \frac{\mathbf{1}}{\mathbf{i=1}} \lambda_{\mathbf{i}}(\mathbf{w_i}). \tag{1}$$

There are three main components in a FL system:

Parties: The parties represent the distributed entities with their dataset. They can be individual devices, such as IoT devices or local servers. Each party has its data not shared with other parties or the manager. They have the role of training the model to preserve privacy and security. The hardware specifications, including power and storage, affect the design of a FL system. Parties with limited hardware capacity may not be able to handle large models and may slow down the entire system.
Manager: A server acts as a coordinator entity and does not have direct access to the raw data held by the parties. It orchestrates the training process, aggregates the model updates from the participants, and creates a global model. The global model represents the knowledge learned from all nodes without exposing the local data.
Communication-Computation Framework: It is the backbone of the FL system. It provides the infrastructure and protocols for the communication and computation between parties and the manager.

Data can be managed in two different ways:

Cross-device FL: The participants are individual devices, such as IoT devices. Each device has its local data.
Cross-silo FL: It involves organizations. Each organization has its own private data and collaborates to train a model with other organizations. Devices from the same organizations share the same dataset. It involves the use of a central server to store the data. For example, hospitals that want to collaborate with other hospitals to create a better model, but cannot share data [5].

Depending on the data distribution scenarios, *horizontal* or *vertical* FL is possible:

Horizontal FL: The participants have similar features, but different datasets;
Vertical FL: The participants train on the same dataset, but for different features.

The most adopted algorithm for aggregating model updates from multiple participants is Federated Averaging (FedAvg). It is applied to various domains, like healthcare and finance. It is an iterative process [24]. As shown in Fig. 3, these are the steps:

1. A global model is initialized on a central server. The manager selects a subset of participants for the training process. The selection is random, based on the devices available or resources.
2. The selected devices train the model locally.
3. The weights from the locally trained model are sent to the central server.
4. The central server aggregates the weights received applying averaging techniques. All the weights are summed and divided by the number of participants. Then, the updated weights are sent back to the edge devices.
5. Steps 2, 3 and 4 are repeated for multiple rounds till the ends of the epochs or a predefined stop criterion.

Some participants may have limited communication bandwidth or intermittent connectivity. Using FedAvg all participants are required to communicate their model updates to the central server each round, but with limited connectivity, there can be bottlenecks. Some devices may be highly unbalanced in performance and may cause a much slower training time for all devices. Thus, performance can be undermined when using heterogeneous devices [25]. Besides FedAvg, we also propose in this work a simplified alternative algorithm. It acts like FedAvg, but rather than updating and generating a global model at each round, the nodes first train the local model for all the epochs, and then send the weights to the central server. In this way, faster nodes do not need to wait for slower nodes to complete the training. The FL solution is developed with Python using the library `Numpy`.

3.3. Transfer learning

Transfer learning is an ML technique in which the knowledge acquired from solving a problem is transferred to improve the performance related to another problem. It can be done by using a pre-trained model. A dataset is used as a starting point for a new task instead of training a model from scratch. More formally, if λ^{t_1} is the model, trained on dataset D_1 to solve the task t_1, transfer learning uses the knowledge from the domain associated with such task to improve the

Table 2
Examples of accuracy of FL without TL and with TL.

	Accuracy without transfer learning	Accuracy with transfer learning
Pre-trained model	–	84.86%
Edge 1	85.01%	84.64%
Edge 2	84.60%	84.97%
Edge 3	84.77%	85.12%
Federated model	46.46%	86.49%

performance of the model λ^{t_2}, trained on dataset D_2 to solve the other task t_2 so that:

$$\lambda^{t_2*} = \lambda^{t_1} \overrightarrow{TL}\, t_2 \quad \text{and} \quad |\lambda^{t_2*}| > |\lambda^{t_2}| \tag{2}$$

where λ^{t_2*} is the model for t_2 improved through transfer learning ($\overrightarrow{TL}$) starting from the knowledge λ^{t_1} related to task t_1 and the $|\ \ |$ operator indicates its performance.

With knowledge transfer, the model obtains benefits from the features and patterns captured by the pre-trained model. The result is faster training and improved performance, especially in cases where the dataset is not complete enough. The use of a pre-trained model on edge nodes can highly increase the performance of the final model. The central server provides a pre-trained model to the edge devices participating in the training process. This model is a starting point and can acquire general features and patterns. Then, the devices fine-tune the pre-trained model using their local data and share the fine-tuned model with the central server. In the end, the server applies FL techniques to generate a final model.

A comparative analysis is shown in Table 2. The models without TL are trained with three distinct datasets with 16,000 entries. We used 8000 elements taken from unused dataset elements for pre-training the network. Then, we adopt TL training the models with three different datasets of 8000 entries.

3.4. Neural network implementation details

The neural networks considered in the experiments are trained according to the supervised learning paradigm. This approach suits IoT and TinyML applications, as it allows us fine-tuning of the model behavior, increasing the accuracy. However, deploying a neural network on devices with limited resources is challenging.

For the implementation of the neural networks, we used Genann, a lightweight library written in pure C, designed for simplicity, speed and extensibility. It implements backpropagation training and supports linear and sigmoid activation functions. It uses a lookup table to store the output of the activation function which helps to obtain faster and more efficient calculations and saves computational resources. The lookup table can have high memory consumption - an issue for some microcontrollers. However, the lookup table can be tuned to the microcontroller. Compared to other available libraries (FANN and Tinn), Genann has a good trade-off between simplicity, performance, and ease of integration.

3.4.1. Enhancements to the Genann library

Several enhancements have been added to the Genann library to improve its functionality and compatibility with resource-constrained devices.

Extended manufactures compatibility: The Genann library has been modified to add compatibility with popular microcontrollers produced by Arduino, Raspberry, and Espressif. The same code can now be compiled on multiple devices without requiring further modifications, allowing faster deployment.
Reading data set from CSV file has been optimized for low-memory devices.
Type conversion from double to float for memory efficiency: To further reduce the memory usage and increase compatibility the library has been updated using float data type instead of double. This conversion significantly decreases memory usage without compromising the accuracy and performance of the model [26].
Export and import weights: To facilitate the deployment in an FL environment, Genann now supports the export and import of the model's weights.
Support for regression and classification tasks: Genann now supports regression and classification tasks, increasing the range of scenarios where the library can be used. Regression functions can predict a continuous value, while classification functions can predict a discrete label or a class.
Extended performance metrics: The library has been improved with proper metrics to assess the effectiveness of the NN. The results are saved in a CSV file.

3.5. MQTT

The proposed technique adopts the MQTT (Message Queuing Telemetry Transport) protocol with the following topics:

/add-participant: Each node participating in the training phase sends a message on this topic, including their ID number in the payload.
/get-weights: The central server is subscribed to the topic. Nodes not participating in training can request the actual global weights by sending a message to this topic.
/receive-weights: Nodes are subscribed to the topic. When a message in the "request-weights" topic is received, the central server sends the weights as a message in the topic "receive-weights". Due to the limited resources of the nodes, the master cannot send the entire model's weights, but it sends individual weight values as separate messages. Since microcontrollers have a small message queue, every 100 messages there is a wait of 0.1s to allow devices to process the queue.
Each node is assigned a unique numeric identifier. After completing a training session, a node sends its weight to a topic corresponding to its identifier. Each weight is sent as a separate message. Once all the weights have been sent, the node sends a special END message, encoded as −1, to indicate the completion of weight transmission.

Fig. 4 shows an example of the operations of Node 1 in the collaborative training phase.

4. Experimentation

4.1. Dataset import

Dataset storage needs to be efficiently handled on microcontrollers, as they have a very constrained amount of memory, usually 64 kB. The traditional approach is to save a dataset entirely in two matrices, for the input and for the output features. While this suits training on computers, it can quickly deplete memory on microcontrollers.

In our proposal, datasets are stored in CSV-format files, where each row represents a data sample, with values separated by a comma. The features are the information used to make predictions; a label or target value is the output to predict. For a dataset with X input features and Y labels, an input array of size X and an output array of size Y are created. Two methods are experimented with for importing the dataset. The first involves reading the dataset line-by-line. Every line read is split by a comma to obtain an array of strings; the fields of the line are converted into numbers. If the string index is less than X, it is added to the input array, otherwise to the output array. In the second method, the dataset is read character by character. Each character is inserted into an array until a comma is reached; once a comma is encountered, the array of

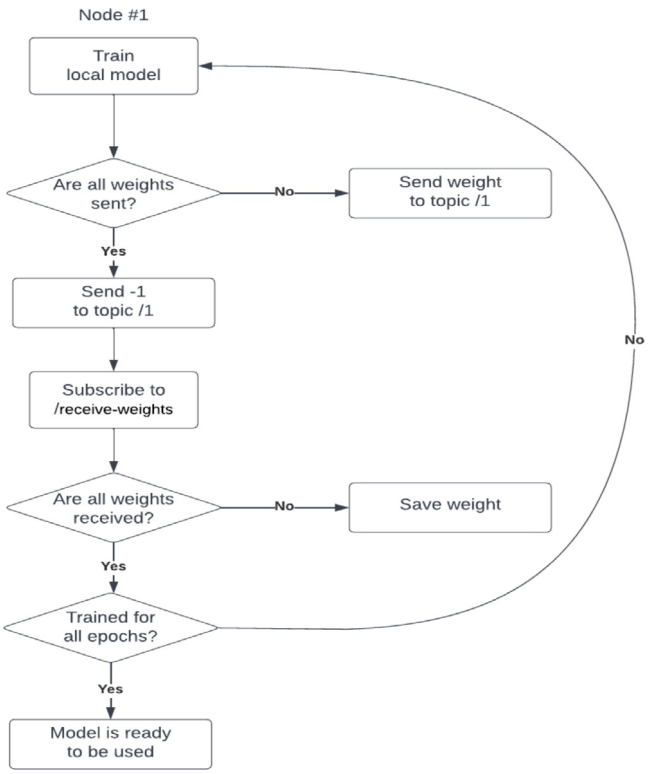

Fig. 4. MQTT-based federated learning algorithm.

Table 3
Dataset memory consumption - Traditional matrix-based import.

Variables	Memory consumption
Input float matrix of size 150 × 1000	600 kB
Output float matrix of size 5 × 1000	20 kB
Char matrix of size 3067 × 1000 to store the dataset	3067 kB
Support variable for number conversion	0.004 kB

Table 4
Dataset memory consumption - Method 1: Read the file line by line and save it in arrays.

Variables	Memory consumption
Input float array of size 150	0.6 kB
Output float array of size 5	0.02 kB
Char array of size 3067 to store one line of the dataset	3067 kB
Support variable for number conversion	0.004 kB

Table 5
Dataset memory consumption - Method 2: Read the file char by char and save it in arrays.

Variables	Memory consumption
Input float array of size 150	0.6 kB
Output float array of size 5	0.02 kB
Char array of size 15 to store one value of the dataset	0.015 kB
Support variable for number conversion	0.004 kB

characters is converted into a number, which is then added to the input or the output array based on the number of commas encountered.

Tables 3 to 5 report results of the traditional import and of the two proposed methods. Table 6 compares the three methods on a dataset - 1000 entries, 150 input features (each of 14 characters), and 5 output features - representative of real TinyML applications. As the last method revealed to be the most efficient, it is the one adopted in subsequent experiments.

Table 6
Dataset memory consumption - Methods comparison.

Variables	Memory consumption
Read the entire file and save it in two matrices	~3095 kB
Read the file line by line and save it in arrays	~3.1 kB
Read the file char by char and save it in arrays	~0.64 kB

4.2. Devices for experiments

Experiments are performed on a network of heterogeneous IoT devices, namely: 1) Arduino WiFi Rev2, 2) ESP8266, 3) ESP32, 4) Arduino MKR1010, 5) Raspberry Pi Zero W, 6) Raspberry Pi3 B+, and 7) a Personal Computer (PC). All devices are equipped with a WiFi board (for collaborative learning). On Raspberry Pi Zero W and Pi3 B+, and on PC, the dataset is saved in internal storage. On Arduino WiFi Rev2 and MKR1010, ESP8266, ESP32, due to the limited flash memory space, it is saved on a micro SD card connected through the HW-125 adapter. The devices characteristics are listed in Table 7.

For each device, we measure the time required to complete a training epoch, the time taken to perform testing on the test set, and the current consumption during training, measured with a USB digital multimeter.

The FL architecture is composed of ESP8266, ESP32, and Arduino MKR1010 nodes, and Raspberry PI3 B+ as the central server. The MQTT broker server is hosted on Raspberry PI3 B+. Fig. 5 shows an example of the designed FL architecture. The boards read from the SD card the dataset as a CSV file; the model is trained; the weights are sent to the server, which aggregates them. Fig. 6 shows the propagation phase of the aggregated weights.

4.3. Experimental settings

Five experimental configurations are defined for both classification and regression tasks; they are listed in Table 8.

Experiments for classification are performed on the *ECG heartbeat dataset* [6]. The task is to classify new ECG input data into categories learned during the training process. Tests are conducted first using a balanced dataset, and then an unbalanced one to check how the network behaves in extreme cases. Once the network is deployed, it is important to continue to improve it with the new data acquired by sensors. We hereafter present a continuous learning solution by exploring a scenario where a specialist corrects the network's wrong results. The network learns from mispredictions to reduce future errors.

The experiments on a regression task use the automotive *Car trips data log* dataset [7].

4.3.1. Metrics

In the following, we report the metrics implemented for the classification task:

Time to train: Time to complete a training epoch.
Time to test: Time to test the classification model on previously unseen data.
Class accuracy: The proportion of correctly predicted instances on the total number of instances. It is an indication of how well the model performs in a class:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}. \quad (3)$$

Class precision: It measures the precision of a class prediction. Precision is the reliability of positive predictions. It is focused on minimizing false positives:

$$Precision = \frac{TP}{TP + FP}. \quad (4)$$

Table 7
Characteristics of devices for experiments.

Device	CPU	of cores	Clock speed	Data bus width	Instruction set	RAM	Storage
Arduino WiFi Rev2	Atmega 4809	1	16 MHz	8 bit	RISC	6 kB	256 kB
ESP8266	Tensilica LX106	1	160 MHz	32 bit	RISC	64 kB	96 kB
ESP32	Tensilica LX6	2	240 MHz	32 bit	RISC	520 kB	512 kB
Arduino MKR1010	SAMD21 Cortex M0+	1	48 MHz	32 bit	ARM	32 kB	256 kB
Raspberry PI Zero W	Broadcom BCM2835	1	1 GHz	32 bit	ARM	512 MB	16 GB
Raspberry PI3 B+	Broadcom BCM2837B0	4	1.4 GHz	64 bit	ARM	1 GB	16 GB
PC	AMD Ryzen 3900X	12	3.8 GHz	64 bit	X86	16 GB	1 TB

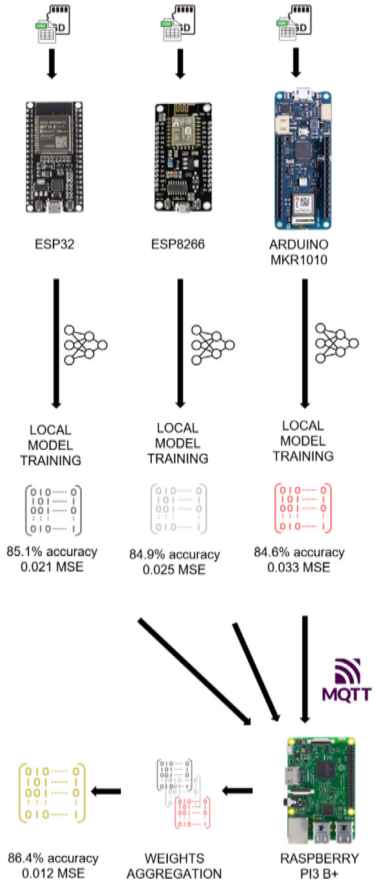

Fig. 5. Federated learning architecture - Training and weights aggregation.

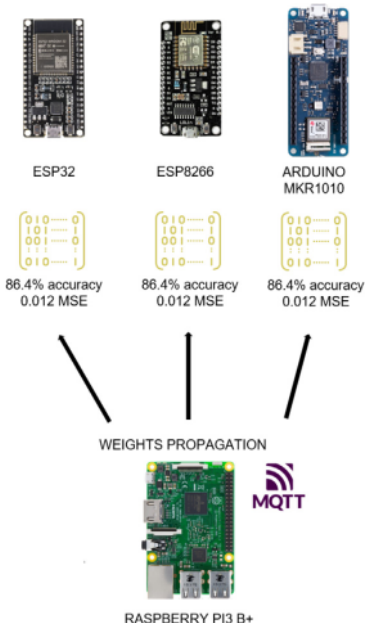

Fig. 6. Federated learning architecture - Weights propagation.

Table 8
Experimental configurations.

Configuration	Federated learning	FedAvg	Transfer learning
1	No	No	No
2	Yes	No	No
3	Yes	No	Yes
4	No	Yes	No
5	No	Yes	Yes

Class recall: It measures the ability of a model to identify positive instances from all instances that truly belong to the positive class:

$$Recall = \frac{TP}{TP + FN}. \tag{5}$$

Class F1: Harmonic mean of precision and recall:

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall}. \tag{6}$$

Macro accuracy: Average accuracy across all classes.
Macro precision: Average precision across all classes.
Mean Squared Error (MSE): Average squared difference between predicted and actual class values:

$$MSE = \frac{1}{n} \quad y - \hat{y})^2. \tag{7}$$

Multi-Class Entropy (MCE): Entropy of the predicted class probabilities:

$$L\ \hat{y}\ y) = - \sum_{k}^{K} y^{k)} log \hat{y}^{k)}. \tag{8}$$

For the regression task, we instead consider the following metrics:

Table 9
Classification (ECG dataset) - Memory consumption.

Variables	Memory consumption
5777 neurons	23.152 kB
Lookup table of size 4096	16.384 kB
(alternative) Lookup table of size 1024	4.096 kB
Input float array of size 187	0.748 kB
Output float array of size 5	0.020 kB
Char array of size 15 to store one value of the dataset	0.015 kB
Support variable for number conversion	0.004 kB

Time to train and time to test.
Accuracy: the proportion of correct predictions within a specific tolerance range.
Mean Absolute Error (MAE): absolute difference between predicted and actual regression values:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|. \quad (9)$$

Mean Squared Error (MSE) between predicted and actual values (Eq. (7)).
Root Mean Squared Error (RMSE):

$$RMSE = \sqrt{\sum_{i=1}^{n}\frac{\hat{y}_i - y_i)^2}{n}}. \quad (10)$$

4.3.2. Settings for classification

Dataset. The *ECG heartbeat dataset* is composed of two collections of heartbeat signals generated from two sanitarian contexts: the *MIT-BIH Arrhythmia Dataset* and *The PTB Diagnostic ECG Database*. The signals correspond to electrocardiogram (ECG) shapes of heartbeats in healthy conditions, arrhythmia, or myocardial infarction.

The trained model can predict if there is a type of arrhythmia or myocardial infarction. The dataset contains 87,555 entries and 187 input features; each input feature is an input channel for ECG. There are 5 categorical classes:

N: Normal, left/right bundle branch block, atrial escape, or nodal escape;
S: Atrial premature, aberrant atrial premature, nodal premature, supra-ventricular premature;
V: Premature ventricular contraction and ventricular escape;
F: Fusion of ventricular, normal;
Q: Paced, fusion of paced and normal, unclassified.

Preprocessing. The output features are encoded via a one-hot scheme. The dataset is divided into 5 parts, but is not entirely used for training. Training the model with a large dataset makes the model achieve its maximum performance, without leaving any tangible benefit in using FL. The split of the dataset is shown in Fig. 7 and it is divided as follows:

The pre-trained model used for transfer learning is trained on 8000 elements;
Each node trains the local model on 8000 elements;
The test set is composed of 10,000 elements.

Neural network. The neural network is configured with 4 layers: an input layer with 187 neurons, 2 hidden layers with 25 neurons each, and an output layer with 5 neurons. The model is trained for 50 epochs. The NN requires at least ~21 or ~44 kB of memory, depending on the size of the lookup table (Table 9). Additional memory space is required for the metrics computation and for libraries, such as SD, File, and MQTT.

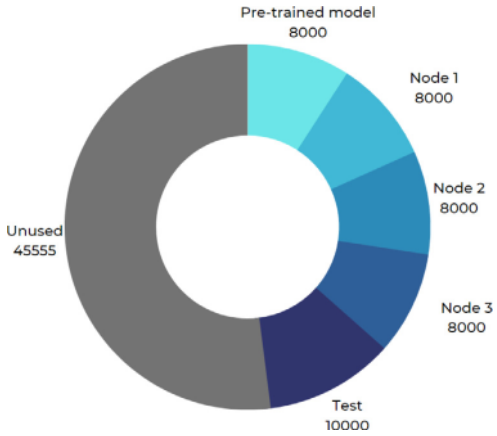

Fig. 7. Classification: ECG dataset split.

Table 10
Regression: Car trips data log dataset input features.

Training input features			
Vehicle's speed	Shift number	Engine load	Total acceleration
Engine RPM	Pitch	Lateral acceleration	Passenger count
Car's Load	Air condit. status	Window opening	Radio volume
Rain intensity	Visibility	Driver's rush	

Table 11
Regression (Car trips data log dataset) - Memory consumption.

Variables	Memory consumption
426 neurons	1.746 kB
Lookup table of size 4096	16.384 kB
(alternative) Lookup table of size 1024	4.096 kB
Input float array of size 187	0.060 kB
Output float array of size 5	0.004 kB
Char array of size 15 to store one value of the dataset	0.015 kB
Support variable for number conversion	0.004 kB

4.3.3. Settings for regression

Dataset. The "Car trips data log" dataset, created by R. F. Vitor [7], contains data acquired from 38 driving sessions in various conditions. The trained model can predict the driver's well-being. The model has potential applications as:

Fatigue detection and prevention: By monitoring the driver's well-being, it is possible to reduce the risk of road accidents.
Calculation of the insurance premium: Insurance companies can adapt the insurance premium value depending on factors, such as stress levels. A driver with a high level of well-being is synonymous of careful driving.

Each entry has 15 input features (Table 10).

Preprocessing. The dataset has been normalized and the values scaled using the Python library `sklearn`. The dataset is divided into 5 parts (Fig. 8). The pre-trained model used for transfer learning is trained on 2240 elements. Each node trains the local model on 2280 elements. The test set is composed of 1520 elements.

Neural network. The neural network is configured with 4 layers: an input layer with 15 neurons, 2 hidden layers with 12 neurons each, and an output layer with 1 neuron. The model is trained for 70 epochs. As shown in Table 11, the network requires at least ~10 kB or ~21.2 kB of memory to be executed, depending on the lookup table size. Additional space is required for metrics computation and for libraries, such as the SD, File, and MQTT.

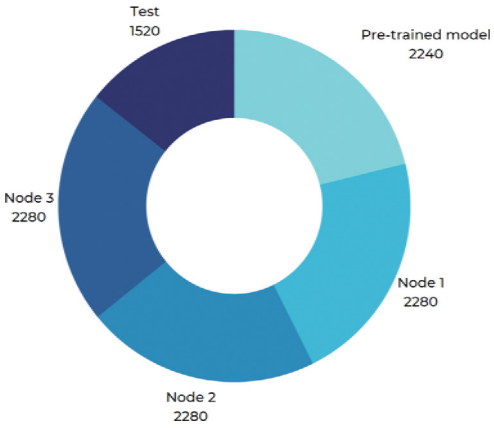

Fig. 8. Regression: Car trips data log dataset split.

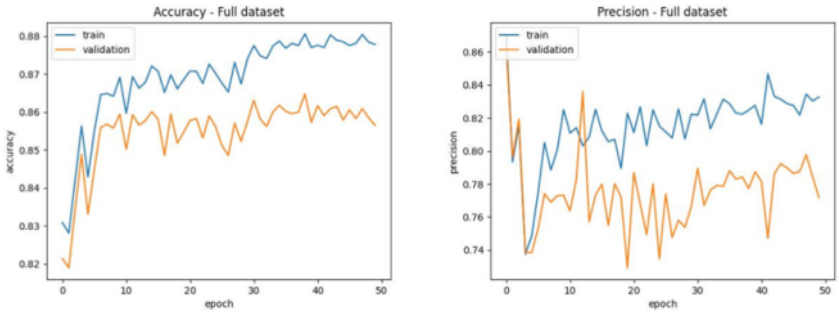

Fig. 9. Classification (ECG) - Full dataset training without federated learning: accuracy and precision.

Table 12
Classification (ECG dataset) - Full dataset training without FL.

Metric	Training	Testing
Accuracy	87.78%	85.65%
Precision	83.27%	77.17%
F1	0.7251	0.6617
MSE	0.0198	0.0210
MCE	0.1148	0.1186

5. Classification

5.1. Results

In the following, the results are discussed taking into account the accuracy of the models achieved on *training* and *testing* datasets, and on the training epochs on the *validation* set (samples left out from training in the corresponding epoch).

Full dataset training without FL. Table 12 reports the results on the full dataset without FL. The difference in precision, accuracy, and F1 between training and testing datasets is more pronounced compared to the difference in the MSE and Multi-Class Entropy (MCE).

Fig. 9 shows that during the first five epochs, a notable decline occurs in precision for both the training and validation datasets, amounting to a 12% reduction. It can be attributed to the model's adjustment to the data. Then, a different trend emerges. The training precision shows consistent improvement, while validation shows a less straightforward path. There are spikes, indicative of temporal improvements, and there is no overall enhancement. This could mean that while the model learns and improves during training, it struggles to generalize information improvements on the validation set.

FL without transfer learning. Fig. 10 shows that the *accuracy* across all nodes is similar for training and validation, and it indicates there are consistent patterns in the learning process.

As for *precision*, Fig. 11 shows that nodes exhibit different behaviors, highlighting variability in the final result. Node 1 and Node 3 experience a sharp drop in the initial 10 epochs with a reduction of ∼21%. Both nodes have small improvements in accuracy in the next epochs. This may indicate there are challenges in improving the model. Unlike Nodes 1 and 3, Node 2 shows a more linear trend in precision during training, without relevant spikes. However, several variations can be noticed in validation, with precision oscillating ∼21%.

As for F1, MSE, and MCE, nodes show similar results in the training and validation phases. Across all metrics, the Node 2 model has more stability and minimal divergence between validation and training; this may indicate the training dataset captures better data patterns. As expected, training shows promising results, but the use of FL reveals a substantial decline in performance (last column of Table 13).

FL with transfer learning. Table 14 shows that the pre-trained model used for the TL phase achieved accuracy and precision scores similar to those obtained in Nodes 1, 2, and 3. However, the nodes gained MCE and MSE scores approximately 30%–40% lower than the pre-trained model. Unlike the previous scenarios without TL, as in Fig. 12 where accuracy is always greater than 0.84 after the 10th epoch, when using TL, the model is consistent across all the training epochs for all the metrics (accuracy, precision, F1, MCE, and MSE). As expected, the use of TL has resulted in marginally improved performances compared to the method without TL. With TL, FL exhibits enhanced performance in all metrics.

FedAvg without transfer learning. The results in Table 15 show the benefits of the FedAvg algorithm compared to the method "FL without TL" in Table 13. FedAvg shows to yield consistent metric trends across

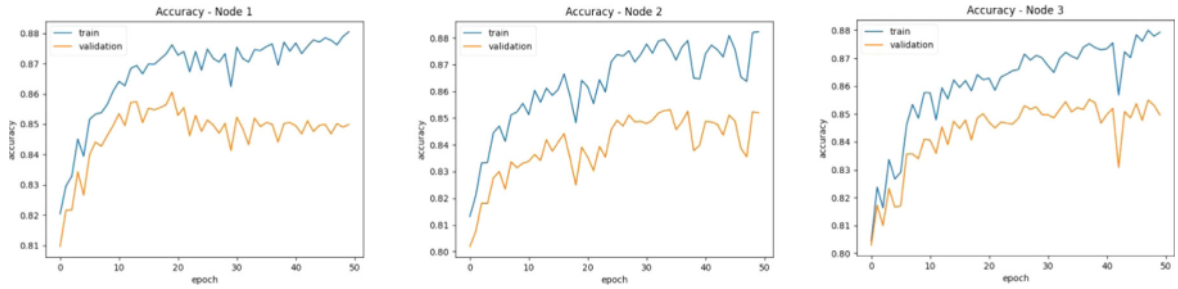

Fig. 10. Classification (ECG) - Federated learning without transfer learning: Accuracy.

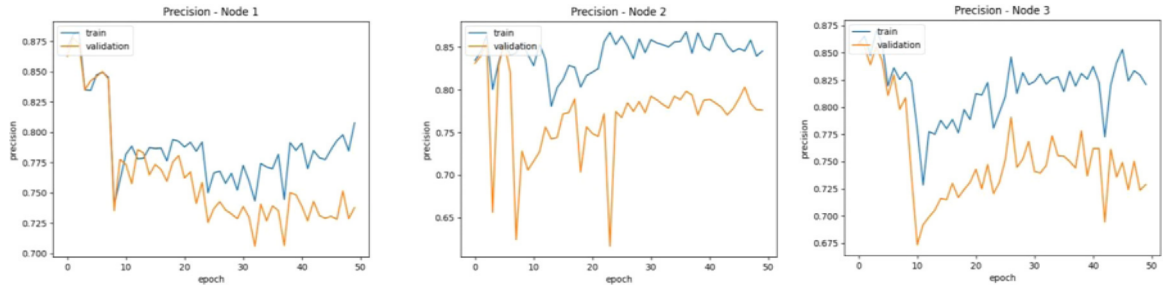

Fig. 11. Classification (ECG) - Federated learning without transfer learning: Precision.

Table 13
Classification (ECG) - FL without transfer learning.

Metric	Node 1		Node 2		Node 3		Federated
	Training	Testing	Training	Testing	Training	Testing	Testing
Accuracy	88.05%	84.99%	88.23%	85.20%	87.93%	84.96%	55.24%
Precision	80.78%	73.79%	84.52%	77.61%	82.10%	72.90%	75.44%
F1	0.7295	0.6814	0.7615	0.6815	0.6828	0.6221	0.3183
MSE	0.0268	0.0292	0.0252	0.0266	0.0221	0.0245	0.3043
MCE	0.1268	0.1332	0.1300	0.1340	0.1208	0.1270	0.8126

Table 14
Classification (ECG) - FL with transfer learning.

Metric	Pre-trained		Node 1		Node 2		Node 3		Federated
	Training	Testing	Training	Testing	Training	Testing	Training	Testing	Testing
Accuracy	88.01%	85.59%	88.80%	85.86%	87.96%	85.16%	89.13%	85.66%	86.48%
Precision	78.55%	72.34%	82.81%	76.34%	84.89%	77.55%	84.07%	75.89%	77.41%
F1	0.7552	0.6996	0.7664	0.7019	0.7247	0.6363	0.7610	0.6800	0.7055
MSE	0.0349	0.0382	0.0191	0.0214	0.0156	0.0157	0.0121	0.0151	0.0117
MCE	0.1608	0.1696	0.0987	0.1052	0.0762	0.0765	0.0758	0.0848	0.0709

Table 15
Classification (ECG) - FedAvg without transfer learning.

Metric	Node 1		Node 2		Node 3		FedAvg
	Training	Testing	Training	Testing	Training	Testing	Testing
Accuracy	87.45%	86.06%	87.20%	85.19%	87.24%	85.57%	85.94%
Precision	79.63%	77.06%	83.24%	79.22%	82.41%	76.16%	77.53%
F1	0.7152	0.6910	0.7187	0.6591	0.6675	0.6342	0.6793
MSE	0.0255	0.0280	0.0286	0.0301	0.0197	0.0221	0.0225
MCE	0.1402	0.1443	0.1493	0.1533	0.1150	0.1215	0.1302

all nodes during training epochs. The final model does not show any drops in accuracy; the MSE and MCE values are substantially lower demonstrating greater stability. However, the precision graph in Fig. 13 shows fluctuations and frequent spikes, ranging from 10% to 30% in most of the epochs before reaching the optimal value.

FedAvg learning with transfer learning. The results in Table 16 indicate a slightly superior performance compared to the FedAvg method without TL. This could indicate that the TL technique is not strictly necessary, but helps to achieve better results during the FL phase. Consistently with the previous method, there is remarkable stability and consistency in the metrics, as shown in Fig. 14 for accuracy. The metrics maintain a uniform trend in the training and validation phase. Additionally, the spikes in the metrics are less frequent and confined only to the initial epochs of the training phase, highlighting the ability of the model to converge faster to the optimal value.

Table 17 compares all the results obtained, giving a summary overview of the above results. As expected, FL without TL has the worst performance. FedAvg and FL with TL have similar performance.

Table 16
Classification (ECG) - FedAvg with transfer learning.

Metric	Pre-trained		Node 1		Node 2		Node 3		FedAvg
	Training	Testing	Training	Testing	Training	Testing	Training	Testing	Testing
Accuracy	87.43%	84.81%	86.78%	85.97%	86.80%	85.99%	87.35%	85.74%	86.31%
Precision	74.72%	68.67%	76.89%	74.80%	83.40%	78.93%	83.61%	76.16%	78.71%
F1	0.7431	0.6846	0.7065	0.6826	0.7089	0.6495	0.6707	0.6396	0.6830
MSE	0.0591	0.0656	0.0286	0.0302	0.0352	0.0366	0.0142	0.0155	0.0208
MCE	0.2404	0.2564	0.1444	0.1489	0.1736	0.1769	0.0965	0.0999	0.1252

Table 17
Classification (ECG) – Results comparison.

Metric	Full dataset – No federated learning	FL without transfer learning	FL with transfer learning	FedAvg without transfer learning	FedAvg with transfer learning
Accuracy	85.65%	53.72%	86.49%	86.18%	86.31%
Precision	77.17%	76.48%	79.78%	81.59%	78.71%
F1	0.6617	0.2361	0.7066	0.6958	0.6830
MSE	0.0210	0.1219	0.0126	0.0193	0.0208
MCE	0.1186	0.3983	0.0683	0.1066	0.1252

Table 18
Classification (ECG) – Devices comparison.

Metric	Arduino WiFi Rev2	ESP8266	ESP32	Arduino MKR1010	Raspberry Pi Zero W	Raspberry Pi3 B+	PC
Time to train (epoch)	FAILED	419 s	321 s	FAILED	22 s	8 s	2.1 s
Time to test	FAILED	198 s	384 s	FAILED	10.9 s	3.7 s	1.2 s
Energy consumption	FAILED	0.07 A	0.0 A	FAILED	0.13 A	0.75 A	21 A/42 W

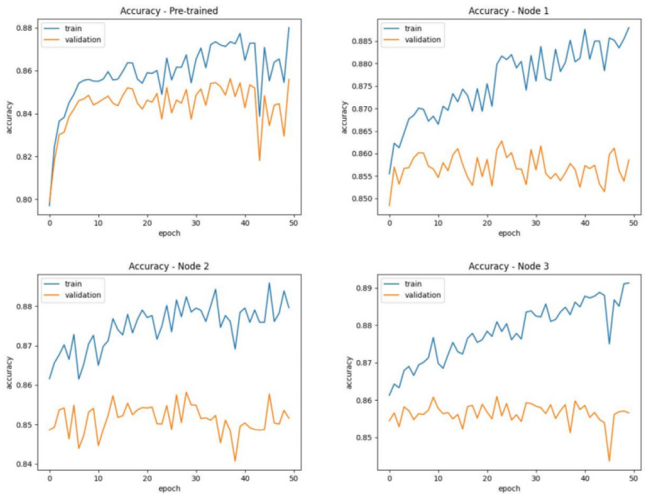

Fig. 12. Classification (ECG) - Federated learning with transfer learning - Accuracy.

5.2. Devices comparison

The two Arduino boards cannot train the model because they do not have enough memory to allocate the NN and the required libraries. As displayed in Table 18, the ESP32 microcontroller is the device with the lowest power consumption (only 0.06 A), which was less demanding compared to the other devices. However, it was slightly slower in training the model than the ESP8266. This discrepancy in speed could be due to a known bug in the ESP32 library, resulting in a longer time to read a file from the SD card. On the other hand, traditional computing devices showed the best results in terms of training speed, at the cost of higher energy consumption: the PC consumed approximately 350% more energy than microcontrollers. The Raspberry Pi Zero W has a good balance between energy consumption and training times, but it is closer to a PC than a microcontroller.

Table 19
Classification (ECG) – Integration of new devices - Time and energy consumption.

Metric	ESP8266	ESP32
Time to download model weights	636 s	637 s
Energy consumption	0.05 A	0.04 A
Energy consumption (IDLE)	0.05 A	0.04 A

5.3. Integration of a new device into an existing architecture

When a new device joins an already deployed FL architecture, it can benefit from the fully trained model without the need to retrain one. Table 19 shows the time and energy consumption for microcontrollers to download the model. The comparison with Table 18 reveals the benefit in the time required for the new device to be in operation. ESP8266 and the ESP32 devices took more than 8 and 9 h to complete training of a new model. The two Arduino boards were excluded because unable to run the network. The two Raspberry Pi were also excluded (as the PC) because of their computer-like architecture and performance. In contrast, downloading only the model weights took approximately 63 s with equivalent power consumption. Table 19 also illustrates how, once the boards acquire the weights, the idle power consumption decreases. The ESP32 consumes 0.04 A, while the ESP8266 0.05 A.

5.4. Unbalanced datasets

The previous tests have been conducted using balanced datasets where all the nodes are trained on datasets with the same class distribution, shown in Table 20. However, in a real-world scenario, it may happen that not all nodes see in operation data similar to those for which they all have been trained. We therefore analyze the case when nodes are trained on unbalanced datasets. Not only global metrics, but also inter-class metrics will be computed to assess if the final model benefits from models with more knowledge of specific classes.

Table 21 illustrates the four datasets prepared for the test with their class distribution. The validation dataset and the one used for TL have the same distribution for all classes.

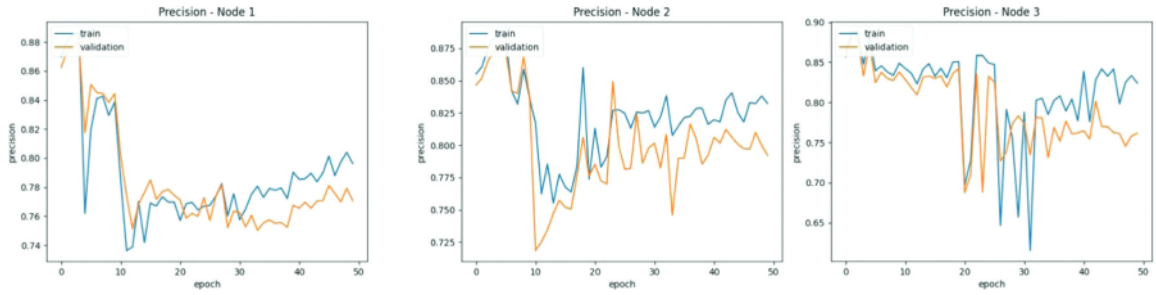

Fig. 13. Classification (ECG) - FedAvg without transfer learning - Precision graph.

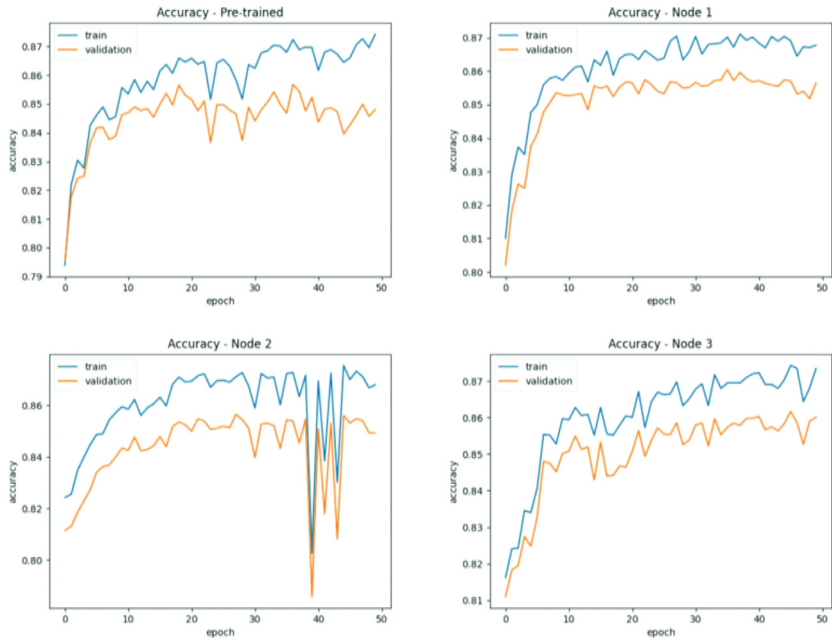

Fig. 14. Classification (ECG) - FedAvg with transfer learning - Accuracy graph.

Table 20
Classification (ECG) - Class distribution in the dataset.

Classes	N	S	V	F	Q
Examples (%)	71	4	11	2	12

Table 22 shows high accuracy results in the training phase, but comparing the training results with the testing ones in Table 22, the accuracy is approximately 10% lower. The model trained using the simplified FL has worse accuracy and MSE. This is caused by the use of a small-sized dataset for TL. On the other hand, the FedAvg model shows better performance compared to the models trained on single datasets.

In Table 23, it can be observed that the simplified version of FL exhibits strongly degraded performance in the N and Q classes. The model trained with FedAvg has better or comparable performance across all classes and it proves to be the best training methodology in contexts with highly unbalanced datasets.

6. Regression

6.1. Results

Full dataset training without federated learning. Table 24 reports the results in the case of no FL being used. The results for training and testing are very close to each other, denoting that the model well generalizes the knowledge acquired during the training.

As shown in Fig. 15, the training and validation lines match for most of the training phase. This may suggest that the data present in the training dataset is similar to the validation dataset. Also in this case, the alignment between training and validation results indicates the model can well generalize the knowledge acquired during the training.

Federated learning without transfer learning. As Fig. 16 shows, the training and validation curves are closely aligned also in this case. The dataset for the FL test is smaller than the one for the full dataset training test. The model reached its optimal values at approximately 60–70 epochs. This pattern is consistent across all nodes, and there is no spike. Node 3 compiled the model with the best performance, followed by Node 1 and Node 2.

The results in Table 25 have a similar pattern as the ones in the classification training task without TL. The final model compiled with FL has notably poorer performances than the models trained individually on each node. This result suggests that alternative training approaches may be more suitable.

Federated learning with transfer learning. Table 26 reports the results when using a pre-trained model for the TL phase. Despite the use of a smaller dataset, the performance in Table 26 was not significantly different from the nodes. All the three nodes demonstrate higher performance than the previous case and are similar across all the nodes. Fig. 17 reports the accuracy computed on the validation set for each training epoch. The pre-trained model reaches a relatively high accuracy of around 90% after 70 epochs (first plot of Fig. 17). The three

Table 21
Classification (ECG) - Unbalanced class distribution in the split dataset.

Classes	Dataset 1	Dataset 2	Dataset 3	Validation	Transfer learning
Class N	5%	53%	11%	20%	20%
Class S	20%	0.6%	0.3%	20%	20%
Class V	3%	1.2%	76.7%	20%	20%
Class F	62%	0.6%	2%	20%	20%
Class Q	10%	44.6%	10%	20%	20%
Number of elements	3210	8170	7040	4000	1600

Table 22
Classification (ECG) - Unbalanced dataset - Training results.

Metric	Dataset 1		Dataset 2		Dataset 3		Federated	FedAvg
	Training	Testing	Training	Testing	Training	Testing	Testing	Testing
Accuracy	83.83%	73.95%	84.73%	73.48%	85.23%	74.61%	63.27%	79.34%
MSE	0.0500	0.1912	0.0183	0.1502	0.0381	0.1679	0.2508	0.1431

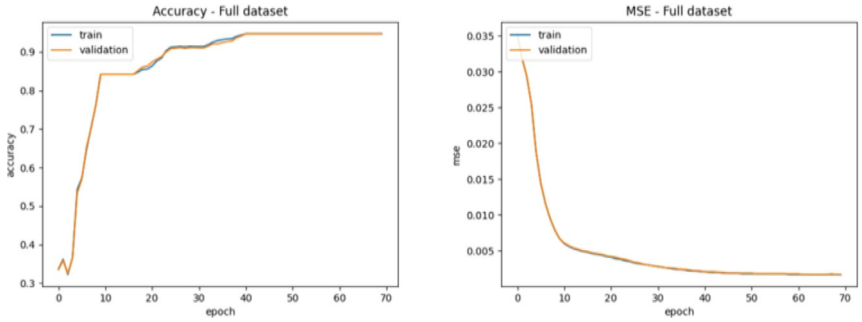

Fig. 15. Regression (Car trips) - Full dataset training without federated learning - Accuracy and MSE.

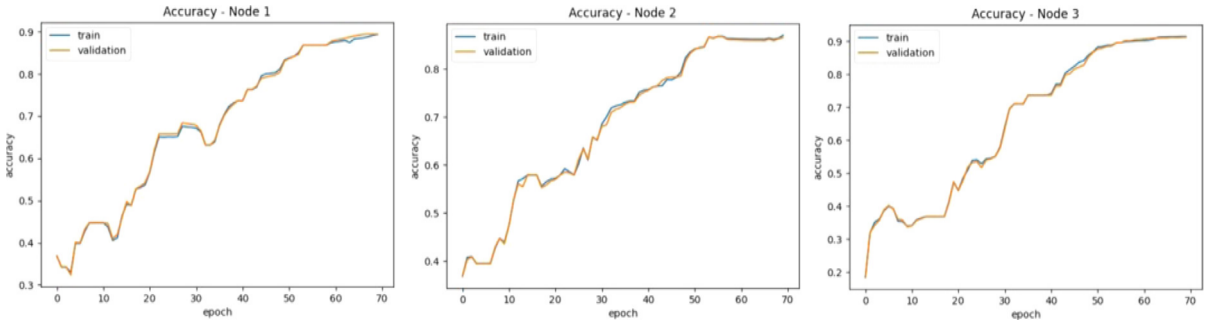

Fig. 16. Regression (Car trips) - FL without transfer learning - Accuracy graph.

Table 23
Classification (ECG) - Unbalanced datasets - Intra-class validation accuracy.

Accuracy	Dataset 1	Dataset 2	Dataset 3	Federated learning	FedAvg
Class N	65.42%	53.88%	67.04%	24.75%	67.98%
Class S	64.54%	75.02%	76.67%	68.35%	78.35%
Class V	75.39%	69.69%	57.66%	67.81%	79.03%
Class F	82.31%	89.00%	88.90%	89.11%	88.69%
Class Q	82.11%	79.83%	82.79%	66.30%	82.65%

Table 24
Regression (Car trips) - Full dataset training without FL.

Metric	Training	Testing
Accuracy	94.74%	94.74%
MAE	0.0206	0.0201
MSE	0.0017	0.0017
RMSE	0.0417	0.0415

nodes exhibit a gradual improvement ranging between 2–4 percentage points and reaching the plateau around the 20 epochs (around 95%). The final model results in a marginally superior performance compared to the individual nodes. This suggests that the final model has likely reached its maximum performance.

FedAvg without transfer learning. Table 27 reports the results when using FEdAvg without transfer learning. In contrast to the classification task, the outcome for regression is different. There are no substantial performance improvements observed in the classification task and the final model did not exhibit a similar level of improvement. In the FedAvg algorithm, each node benefits from the collective knowledge of all nodes during the training phase. However, the final model performance is slightly worse than the most-performing node 27. As shown in Fig. 18 for accuracy, similarly to the previous case, the training and validation curves largely matched for most of the training phase.

FedAvg with transfer learning. Table 28 reports the results when using FedAvg *with* TL. In contrast to the same training methodology applied in the classification task, in which the use of TL had a minimal impact on the final model's performance, this test demonstrated a higher

Table 25
Regression (Car trips) - Federated learning without transfer learning.

Metric	Node 1		Node 2		Node 3		Federated
	Training	Testing	Training	Testing	Training	Testing	Testing
Accuracy	89.39%	89.47%	86.97%	86.51%	91.54%	91.32%	15.79%
MAE	0.0388	0.0378	0.0511	0.0532	0.0377	0.0384	0.2605
MSE	0.0037	0.0036	0.0047	0.0050	0.0035	0.0036	0.0886
RMSE	0.0606	0.0602	0.0687	0.0705	0.0591	0.0597	0.2976

Table 26
Regression (Car trips) - FL with transfer learning.

Metric	Pre-trained		Node 1		Node 2		Node 3		Federated
	Training	Testing	Training	Testing	Training	Testing	Training	Testing	Testing
Accuracy	90.31%	91.05%	94.74%	94.74%	94.74%	94.74%	94.74%	94.74%	94.74%
MAE	0.0388	0.0387	0.0188	0.0191	0. 0243	0.0249	0.0200	0.0203	0.0201
MSE	0.0033	0.0033	0.0017	0.0017	0.0019	0.0019	0.0017	0.0017	0.0017
RMSE	0.0572	0.0574	0.0411	0.0412	0.0435	0.0439	0.0413	0.0414	0.0415

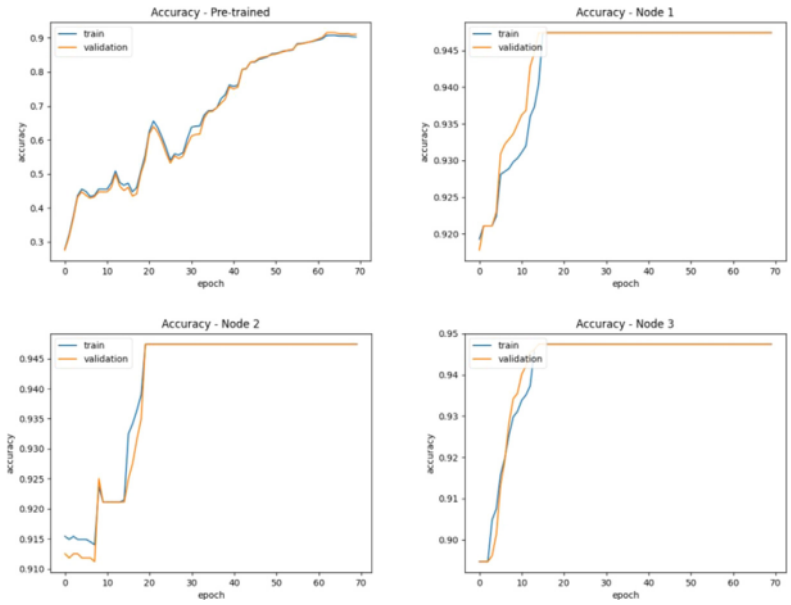

Fig. 17. Regression (Car trips) - FL with transfer learning - Accuracy graph.

Table 27
Regression (Car trips) - FedAvg without transfer learning.

Metric	Node 1		Node 2		Node 3		FedAvg
	Training	Testing	Training	Testing	Training	Testing	Testing
Accuracy	85.26%	85.53%	87.32%	87.57%	86.84%	86.84%	86.84%
MAE	0.0430	0.0422	0.0508	0.0521	0.0407	0.0412	0.0413
MSE	0.0041	0.0041	0.0050	0.0052	0.0040	0.0041	0.0041
RMSE	0.0642	0.0641	0.0704	0.0718	0.0630	0.0638	0.0637

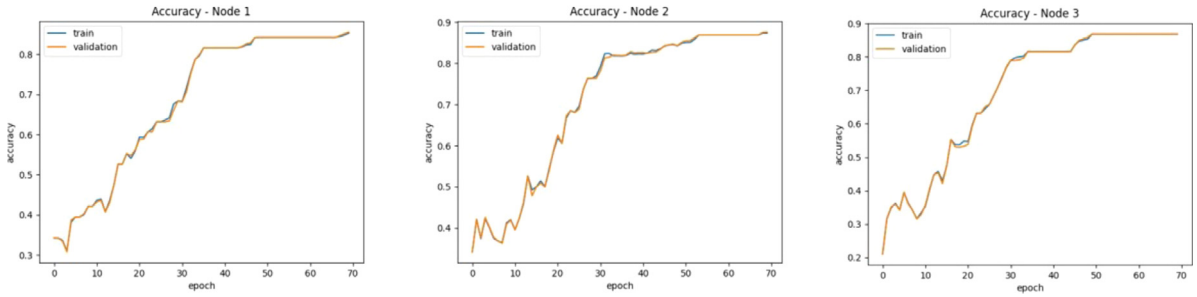

Fig. 18. Regression (Car trips) - FedAvg without transfer learning - Accuracy graph.

Table 28
Regression (Car trips) - FedAvg with transfer learning.

Metric	Pre-trained		Node 1		Node 2		Node 3		FedAvg
	Training	Testing	Training	Testing	Training	Testing	Training	Testing	Testing
Accuracy	91.96%	92.11%	94.74%	94.74%	94.74%	94.74%	94.74%	94.74%	94.74%
MAE	0.0380	0.0389	0.0215	0.0220	0.0278	0.0286	0.0224	0.0231	0.0230
MSE	0.0032	0.0033	0.0018	0.0018	0.0021	0.0021	0.0018	0.0018	0.0018
RMSE	0.0568	0.0575	0.0422	0.0425	0.0453	0.0457	0.0426	0.0429	0.0428

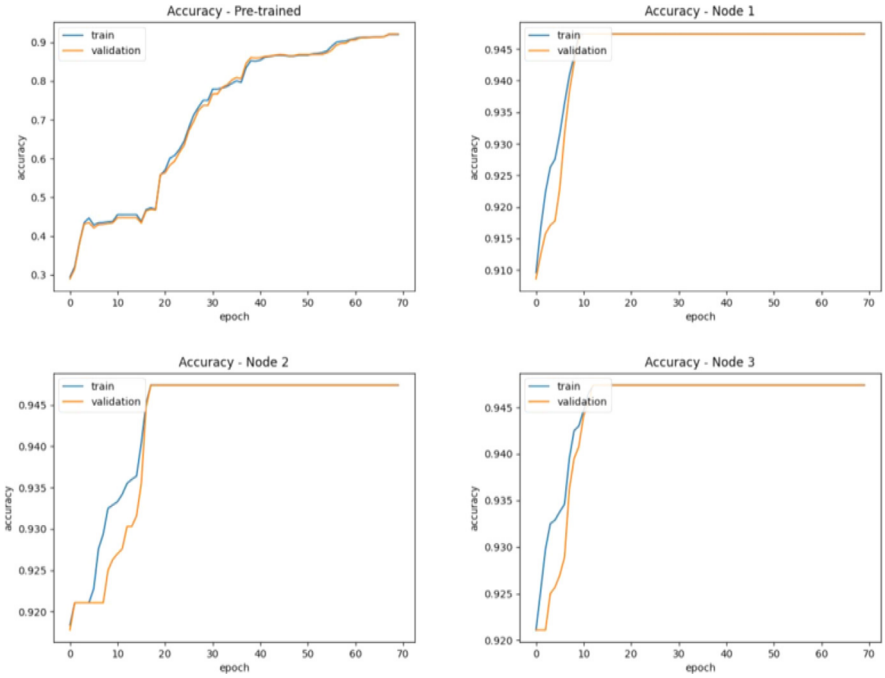

Fig. 19. Regression (Car trips) - FedAvg with transfer learning - Accuracy graph.

Table 29
Regression (Car trips) – Results comparison.

Metric	Full dataset – No federated learning	FL without transfer learning	FL with transfer learning	FedAvg without transfer learning	FedAvg with transfer learning
MAE	0.0206	0.2605	0.0201	0.0413	0.0230
MSE	0.0017	0.0886	0.0017	0.0041	0.0018
RMSE	0.0415	0.2976	0.0415	0.0637	0.0428

benefit. All the nodes exhibited in Table 28 improved and strictly similar performance compared to the non-transfer learning training approach. Each node obtained the same level of accuracy and nearly identical MAE, MSE, and RMSE values.

Despite the use of TL, the final model reached the same level of accuracy, with only slight negative variation in MAE, MSE, and RMSE values, differing by just a few decimal points. The use of TL confirmed the benefit: accuracy during training (Fig. 19) starts at the value of 91%, the value reached by the pre-trained network after 70 epochs (first plot of Fig. 19), and reaches the plateau after around 10 epochs, maintaining the same level (around 95%) throughout the entire training. MSE, MAE, and RMSE values continue to decrease.

Table 29 provides the summary overview. The training methods exhibit similar results as for the classification task (Table 17). The FL task without TL shows lower results. Instead, FedAvg and FL with TL demonstrate similar performance.

6.2. Devices comparison

Table 30 compares performance on the devices. The Arduino WiFi Rev2 board could not train the model because it lacks the required memory to allocate the NN and the required libraries. The ESP8266, despite being a single-core microcontroller, shows slightly higher power consumption compared to the dual-core ESP32 and achieves faster training and testing times. As reported before, this difference in performance may be attributed to a known issue with the ESP32's SD card library. The Arduino MKR1010, due to its limited memory, had the lookup table reduced from the size of 4096 to 1024. It is the device that consumed the least amount of energy, only 0.02 A, which was more than three times less than the ESP8266. However, it took more time to complete the training process, approximately five times more. On the other hand, the PC proved to be the fastest device for compiling the model but was also the least energy-efficient among the devices tested.

6.3. Integration of a new device into a trained model architecture

Hereafter, we analyze the scenario when a new device joins an FL architecture with a fully trained model. The time to train the models for the ESP8266, ESP32, and Arduino MKR1010, was, respectively, 9, 15, and 43 min. The model weights download required 32.1, 33.3, and 3.2 s (Table 31). Examining the power consumption, the ESP8266 and the ESP32 have lower values, 0.05 A and 0.02 A. The Arduino MKR1010 has a higher power consumption of 0.07 A, possibly caused by the accelerated message queue processing.

Table 30
Regression (Car trips) – Devices comparison.

Metric	Arduino WiFi Rev2	ESP8266	ESP32	Arduino MKR1010	Raspberry Pi Zero W	Raspberry Pi3 B+	PC
Time to train (epoch)	FAILED	5.4 s	7.5 s	28.1 s	0.8 s	0.3 s	0.0036 s
Time to test	FAILED	2.4 s	4.9 s	9.3 s	0.18 s	0.065 s	0.015 s
Energy consumption	FAILED	0.07 A	0.06 A	0.02 A	0.15 A	0.7 A	20 A/40 W

Table 31
Regression (Car trips) – Integration of new devices - Time and energy consumption.

Metric	ESP8266	ESP32	Arduino MKR1010
Time to download model weights	32.1 s	33.3 s	3.2 s
Energy consumption	0.05 A	0.02 A	0.07 A
Energy consumption (IDLE)	0.05 A	0.04 A	0.07 A

Once the devices acquire the weights, there is a slight increase in idle power consumption, as shown in Table 31. The ESP32 device registers 0.04 A and the ESP8266 0.05 A. The Arduino MKR1010 energy consumption is consistent.

6.4. Experiments with unbalanced training datasets

We explore the case of unbalanced training datasets. In previous tests, nodes were trained on balanced datasets containing the same portion of data from each session but captured at different times, with approximately each dataset having 2.6% of each session. To add variability, 98% of each dataset consists of data extracted from 13 sessions, with each session contributing with the ∼7.5% of the data. The selection of the 13 sessions varies for each node. The remaining 2% is drawn from all remaining sessions, approximately 0.08% of each session. Each training dataset contains 11,922 entries. Table 32 shows a sample configuration. The testing dataset contains 38,000 entries, equally distributed. The TL model is trained on 1900 entries, 50 per session.

Table 33 shows extremely high accuracy in the training phase, but the first and third models lose around 30% of accuracy with the test dataset, due to overfitting. This indicates the created datasets have a high level of unbalance. The newly generated models exhibit higher performance than the three models trained on individual datasets. The "simplified federated learning" model has a comparable but slightly worse performance than FedAvg.

7. Comparative analysis

We now assess whether the proposed technique performs similarly to traditional pre-compiled models, by comparing it to models trained with `TensorFlow Lite` for Microcontrollers [12]. The same networks configurations used in previous experiments are replicated in these tests, and the models are trained on the entire dataset.

7.1. TensorFlow Lite for Microcontrollers

TensorFlow Lite for Microcontrollers is a widely used ML framework for resource-limited devices. As it does not support on-board training, the steps to run a model are:

1. Train the model on a PC or a server using `Tensorflow`.
2. Convert the trained model into a plain C array, e.g., using the `TinyML gen` library.
3. Deploy and run the converted model on the microcontroller.

One limitation of this approach is the lack of compatibility with most of the boards. Among the boards previously tested, the ESP32 is the only device compatible. All subsequent experiments are performed on this device.

7.2. Classification task comparison

The model is trained on the entire ECG dataset for 50 epochs. In Table 34, the FedAvg and FL with transfer learning show slightly better accuracy than the `TensorFlow` Lite model, but worse precision. The accuracy graph in Fig. 20 shows the same trends as the graph of the model generated from the proposed technique, trained on the full dataset.

7.3. Regression task comparison

The model is trained on the entire *Car trips data log* for 70 epochs. In Table 35, the FedAvg and FL with TL perform overall better than the `TensorFlow Lite` model trained on the entire dataset. As in the previous case, the graphs in Fig. 21 show no significant differences from the ones obtained with the proposed architecture.

7.4. Considerations

The non-significant difference between the local models and the ones trained with `Tensorflow Lite` is attributable mainly to the datasets. The main advantage of remote training is to fine-tune the models without constraints on resources, number of features, and training dataset size. This advantage is very limited in our experiments, due to the nature of the dataset. Local training allows the chosen model to reach a good level of accuracy, close to the one obtained with `Tensorflow Lite`. Different results can be obtained on more complex models or datasets, on which training with `Tensorflow Lite` may be more effective.

8. Post deployment continuous model improvement

Once a model has been deployed on IoT devices, sensors may acquire new (unlabeled) data which may be misclassified. It is important to continue improving the model using previously unseen data, so that the system can keep learning and minimize future mispredictions. Inspection of predictions by domain specialists may spot and report

Table 32
Regression (Car trips) - Unbalanced class distribution in the split dataset.

Session	1	2	3	...	13	14	15	...	38
Class distribution (%)	∼7.5%	∼7.5%	∼7.5%	...	∼7.5%	0.08%	0.08%	...	0.08%

Table 33
Regression (Car trips) - Unbalanced dataset.

Metric	Dataset 1		Dataset 2		Dataset 3		Federated	FedAvg
	Training	Testing	Training	Testing	Training	Testing	Testing	Testing
Accuracy	97.61%	67.46%	98.34%	82.37%	97.49%	66.93%	85.53%	86.68%
MSE	0.0014	0.0141	0.0016	0.0098	0.0017	0.0169	0.0082	0.0053

Table 34
Classification (ECG dataset) - Comparison to TensorFlow Lite.

Metric	Full dataset – No federated learning	FL with transfer learning	FedAvg with transfer learning	TensorFlow Lite Full dataset
Accuracy	85.65%	86.49%	86.31%	85.77%
Precision	77.17%	79.78%	78.71%	86.39%
F1	0.6617	0.7066	0.6830	0.6920
MSE	0.0210	0.0126	0.0208	0.0478

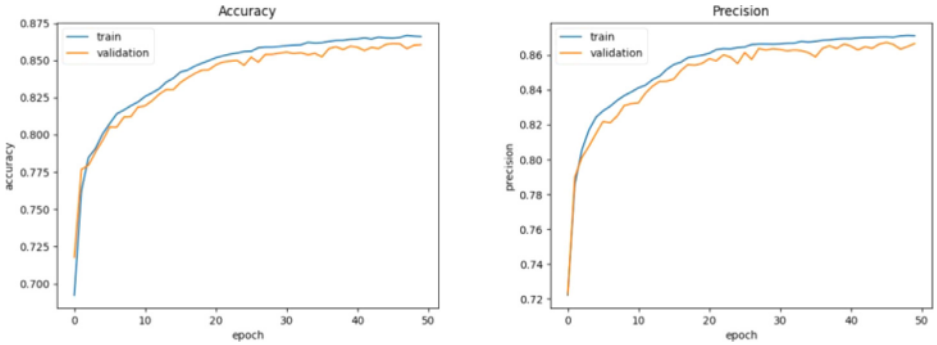

Fig. 20. Classification (ECG dataset) - Comparison to TensorFlow Lite (graphs).

Table 35
Regression (Car trips dataset) - Comparison to TensorFlow Lite.

Metric	Full dataset – No federated learning	FL with transfer learning	FedAvg with transfer learning	TensorFlow Lite Full dataset
Accuracy	94.74%	94.74%	94.74%	94.50%
MAE	0.0206	0.0201	0.0230	0.0276
MSE	0.0017	0.0017	0.0018	0.0022
RMSE	0.0415	0.0415	0.0428	0.0476

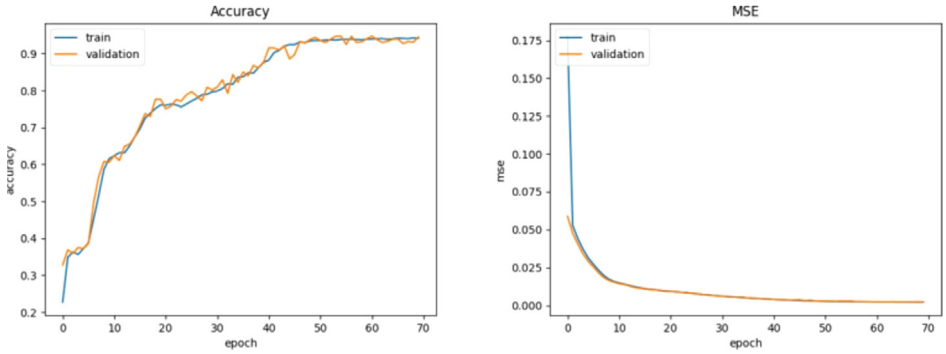

Fig. 21. Regression (Car trips dataset) - Comparison to TensorFlow Lite (graphs).

incorrect operational results. As final contribution, we present the results of experiments for continuous improvement of ML models after deployment, based on the eepEST technique [27].

eepEST (*Deep neural networks Enhanced Sampler for operational Testing*) is a technique for DNN assessment and improvement over delivery cycles. It selects from operational inputs samples with high probability of being misclassified. After their labeling (e.g., by domain experts), it provides an estimate of the ML model accuracy in operation. The selected examples are then used to improve the model accuracy before next deployment.

In the experiment, we consider the 45,500 unused samples from the classification dataset as inputs collected in operation, pretending the actual labels are provided by medical professionals. These samples are split into 13 batches of 3500 samples each. The dataset for eepEST requires as input features: 1) ID - as row number; 2) Outcome - a binary outcome indicating whether the network prediction is correct or not; 3) SUT (System Under Test) - label generated by the network; and 4) Confidence - the confidence level of the prediction, obtained from the output layer nodes. For each batch, eepEST is configured to select 500 predictions with the worst results. Then, the selected samples are added to the training set to improve the network and to the test dataset used to evaluate the performance.

During each cycle, the following data is recorded to monitor improvements: *a*) Accuracy of the base model on the test set; *b*) Accuracy of the fine-tuned model on the test set; *c*) Delta, the difference in accuracy between the fine-tuned model and the base model; and *d*) Accuracy estimated by eepEST. Evaluations have been conducted using the following configurations:

Two training configurations for the initial model. Both trained using FedAvg, one trained for 25 epochs and another for 50 epochs. The 25 epoch configuration is included to ensure the model was not overfitting during the initial training phase.

Table 36
Classification (ECG): tuning with a model trained for 50 epochs and fine-tuned for 10 epochs.

Cycle	Failures	Accuracy base model	Accuracy of retrained model	Delta	DeepEST estimated accuracy
1	97	0.843333	0.829238	−0.014095	0.865048
2	124	0.832182	0.834091	0.001909	0.820258
3	114	0.836696	0.842174	0.005478	0.844781
4	110	0.844583	0.852917	0.008334	0.845922
5	110	0.854240	0.853360	−0.000880	0.837340
6	129	0.854231	0.852231	−0.002000	0.806063
7	112	0.854296	0.861407	0.007111	0.860654
8	102	0.862643	0.852643	−0.010000	0.856618
9	108	0.854552	0.860345	0.005793	0.825758
10	123	0.861067	0.860200	−0.000867	0.853910
11	130	0.860645	0.858903	−0.001742	0.818480
12	135	0.860563	0.861812	0.001249	0.809783
13	89	0.862242	0.859333	−0.002909	0.867330

Three fine-tuning configurations, with 5, 10 and 20 epochs.

Table 36 shows the results. The best performance has been achieved with an initial model trained for 50 epochs and fine-tuned for 10 epochs. In 5 epochs the model does not learn enough from the new data, while in 20 epochs it overfits. These results show that by applying continuous learning techniques it is possible to improve the model knowledge: in the experiments, the overall accuracy of the network increased from 84.3% to 86.22%.

9. Conclusions

We presented the design and experimentation of a technique for enabling a network of heterogeneous IoT devices with a variety of hardware constraints to collaboratively train machine learning applications directly on board of devices. The proposal uses Federated Learning, possibly combined with Transfer Learning. The experiments with devices such as Arduino WiFi Rev2, ESP8266, ESP32, Raspberry Pi, and a PC, evaluating the model performance in classification and regression tasks.

FL combined with TL has been proven to be a valid starting point for enabling microcontrollers to do training and inference on the board. We have also evaluated scenarios with unbalanced datasets, with the integration of new nodes, and, for classification tasks, with a mechanism to enable continuous (re)training. The results of FL and TL, in different configurations, have also been compared to traditional TinyML models trained using `TensorFlow Lite` for Microcontrollers, and the performance of the created federated system turned out to be similar or even better, without sacrificing data privacy.

We plan to conduct additional tests in specific scenarios, in order to experiment the proposed technique on a wider variety of configurations and conditions. We are especially interested in continuous training scenarios, to deal with the frequent case of operational data drifting from training data, a phenomenon which is known to potentially cause severe performance drops. In future work, we will: *i*) extend the neural network library with additional activation functions, so it can be applied in more fields, *ii*) evaluate more secure model aggregation approaches and increase privacy during the training process; *iii*) apply the proposed technique in real-world scenarios to evaluate how it performs in various domains.

CRediT authorship contribution statement

M. Ficco: Supervision, Conceptualization, Methodology, Experimentals, Software Supervision, Writing – reviewing. **A. Guerriero:** Investigation, Methodology, Writing – original draft. **E. Milite:** Software, Development, Experimentation. **F. Palmieri:** Supervision, Conceptualization, Methodology, Writing – reviewing. **R. Pietrantuono:** Investigation, Conceptualization, Methodology, Experimentals, Software, Supervision, Writing – reviewing. **S. Russo:** Supervision, Conceptualization, Methodology, Writing – reviewing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

The work by S. Russo was supported by the Sustainable Mobility National Research Center (MOST), Italy under Piano Nazionale di Ripresa e Resilienza (PNRR) – Missione 4 Componente 2, Investimento 1.4 – D.D. 1033 17/06/2022, CN00000023. The work by M. Ficco and F. Palmieri was partially supported by project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU. The work by M. Ficco is also part of the research activity realized within the project Federated Learning for Generative Emulation of Advanced Persistent Threats (FLEGREA), Bando PRIN 2022. All authors have read and agreed to the published version of the manuscript.

References

[1] L.S. Vailshery, Number of IoT connected devices worldwide 2019–2021, with forecasts to 2030, 2022.
[2] S.F. Ahmed, M.S.B. Alam, S. Afrin, S.J. Rafa, N. Rafa, A.H. Gandomi, Insights into Internet of Medical Things (IoMT): Data fusion, security issues and potential solutions, Inf. Fusion 102 (2024).
[3] R. Sanchez-Iborra, A.F. Skarmeta, TinyML-enabled frugal smart objects: Challenges and opportunities, IEEE Circuits Syst. Mag. 20 (3) (2020) 4–18.
[4] P.P. Ray, A review on TinyML: State-of-the-art and prospects, J. King Saud Univ. - Comput. Inf. Sci. 34 (4) (2022) 1595–1623.
[5] Q. Li, Z. Wen, Z. Wu, S. Hu, N. Wang, X. Liu, B. He, A survey on federated learning systems: Vision, hype and reality for data privacy and protection, IEEE Trans. Knowl. Data Eng. 35 (2023) 3347–3366.
[6] M. Kachuee, S. Fazeli, M. Sarrafzadeh, ECG heartbeat classification: A deep transferable representation, in: IEEE International Conference on Healthcare Informatics, IEEE, 2018, pp. 443–444.
[7] R.F. Vitor, Car trips data log, 2017, https://www.kaggle.com/datasets/vitorrf/cartripsdatamining.
[8] V. Tsoukas, E. Boumpa, G. Giannakas, A. Kakarountas, A review of machine learning and TinyML in healthcare, in: PCI '21: Proc. 25th Pan-Hellenic Conference on Informatics, ACM, 2022, pp. 69–73.
[9] S.K. Lo, C.W. Q. Lu, H.-Y. Paik, L. Zhu, A systematic literature review on federated machine learning: From a software engineering perspective, ACM Comput. Surv. 54 (5) (2021) 1–39.
[10] H. Ren, D. Anicic, T. Runkler, TinyOL: TinyML with online-learning on microcontrollers, in: International Joint Conference on Neural Networks (IJCNN), IEEE, 2021, pp. 1–8.
[11] C.R. Banbury, V.J. Reddi, M. Lam, et al., Benchmarking tinyml systems: Challenges and direction, 2020, arXiv:2003.04821.
[12] R. David, J. Duke, A. Jain, et al., TensorFlow lite micro: Embedded machine learning on TinyML systems, in: Proceedings of the 4th Machine Learning and Systems (MLSys 2021), 2021.

[13] L. Lai, N. Suda, V. Chandra, CMSIS-NN: Efficient neural network kernels for arm Cortex-M CPUs, 2018, arXiv preprint arXiv:1801.06601.
[14] Apache TVM Project, 2021, https://tvm.apache.org/ Accessed: Nov. 9, 2023.
[15] L. Ravaglia, M. Rusci, D. Nadalini, A. Capotondi, F. Conti, L. Benini, A tinyml platform for on-device continual learning with quantized latent replays, IEEE J. Emerg. Sel. Top. Circuits Syst. 11 (4) (2021) 789–802.
[16] J. Lin, W.-M. Chen, Y. Lin, J. Cohn, C. Gan, S. Han, Mcunet: Tiny deep learning on iot devices, Adv. Neural Inf. Process. Syst. 33 (2020) 11711–11722.
[17] J. Montiel, M. Halford, S.M. Mastelini, G. Bolmier, R. Sourty, R. Vaysse, A. Zouitine, H.M. Gomes, J. Read, T. Abdessalem, A. Bifet, River: machine learning for streaming data in Python, 2020, arXiv:2012.04740.
[18] K. Kopparapu, E. Lin, J.G. Breslin, B. Sudharsan, TinyFedTL: Federated transfer learning on ubiquitous tiny IoT devices, in: IEEE Int. Conf. on Pervasive Computing and Communications Workshops and Other Affiliated Events, IEEE, 2022, pp. 79–81.
[19] M.M. Grau, R.P. Centelles, F. Freitag, On-device training of machine learning models on microcontrollers with a look at federated learning, in: Conf. on Information Technology for Social Good, 2021, pp. 198–203.
[20] Y.D. Kwon, R. Li, S.I. Venieris, J. Chauhan, N.D. Lane, C. Mascolo, TinyTrain: Deep neural network training at the extreme edge, 2023, arXiv:2307.09988.
[21] H. Ren, D. Anicic, A. Thomas, TinyReptile: TinyML with federated meta-learning, in: Int. Joint Conf. on Neural Networks (IJCNN), 2023.
[22] A. Nichol, J. Achiam, J. Schulman, On first-order meta-learning algorithms, 2018, arXiv:1803.02999.
[23] N. Xiong, S. Punnekkat, Tiny federated learning with Bayesian classifiers, in: IEEE 32nd Int. Symp. on Industrial Electronics (ISIE), 2023.
[24] H.B. McMahan, E. Moore, D. Ramage, B.A. y Arcas, Communication-efficient learning of deep networks from decentralized data, 2016, CoRR abs/1602.05629, arXiv:1602.05629.
[25] C.R. Banbury, V.J. Reddi, M. Lam, W. Fu, A. Fazel, J. Holleman, X. Huang, R. Hurtado, D. Kanter, A. Lokhmotov, D.A. Patterson, D. Pau, J. Seo, J. Sieracki, U. Thakker, M. Verhelst, P. Yadav, Benchmarking TinyML systems: Challenges and direction, 2020, CoRR abs/2003.04821.
[26] D. Nadalini, M. Rusci, L. Benini, F. Conti, Reduced precision floating-point optimization for deep neural network on-device learning on MicroControllers, 2023, arXiv:2305.19167.
[27] A. Guerriero, R. Pietrantuono, S. Russo, Operation is the hardest teacher: estimating DNN accuracy looking for mispredictions, in: 43rd International Conference on Software Engineering, IEEE, 2021, pp. 348–358.