Future Generation Computer Systems 133 (2022) 53–67

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Dynamic and adaptive fault-tolerant asynchronous federated learning using volunteer edge devices

José Ángel Morell *, Enrique Alba

Universidad de Málaga, ITIS Software, Spain

ARTICLE INFO

Article history:
Received 21 April 2021
Received in revised form 21 February 2022
Accepted 25 February 2022
Available online 8 March 2022

Keywords:
Federated learning
Edge computing
Internet browser
Distributed computing
Volunteer computing
Deep learning

ABSTRACT

The number of devices, from smartphones to IoT hardware, interconnected via the Internet is growing all the time. These devices produce a large amount of data that cannot be analyzed in any data center or stored in the cloud, and it might be private or sensitive, thus precluding existing classic approaches. However, studying these data and gaining insights from them is still of great relevance to science and society. Recently, two related paradigms try to address the above problems. On the one hand, edge computing (EC) suggests to increase processing on edge devices. On the other hand, federated learning (FL) deals with training a shared machine learning (ML) model in a distributed (non-centralized) manner while keeping private data locally on edge devices. The combination of both is known as federated edge learning (FEEL). In this work, we propose an algorithm for FEEL that adapts to asynchronous clients joining and leaving the computation. Our research focuses on adapting the learning when the number of volunteers is low and may even drop to zero. We propose, implement, and evaluate a new software platform for this purpose. We then evaluate its results on problems relevant to FEEL. The proposed decentralized and adaptive system architecture for asynchronous learning allows volunteer users to yield their device resources and local data to train a shared ML model. The platform dynamically self-adapts to variations in the number of collaborating heterogeneous devices due to unexpected disconnections (i.e., volunteers can join and leave at any time). Thus, we conduct comprehensive empirical analysis in a static configuration and highly dynamic and changing scenarios. The public open-source platform enables interoperability between volunteers connected using web browsers and Python processes. We show that our platform adapts well to the changing environment getting a numerical accuracy similar to today's configurations using a given number of homogeneous (hardware and software) computers as a static platform for learning. We demonstrate the fault-tolerance of the platform in self-recovering from unexpected disconnections of volunteer devices. We then prove that EC, coupled with FL, can lead to scientific tools that can be practical involving real users for final competitive numerical results in real problems for science and society.

1. Introduction

The current advances in Artificial Intelligence (AI) make it possible to train machine learning (ML) models by exploiting daily generated data that was previously considered useless [1,2]. Statista[1] stated that there are 23.8 billion interconnected and active computing devices worldwide and that they will produce 149 zettabytes of data by 2024. Also, Cisco[2] estimated that at least 85 (10%) of the 850 zettabytes created in 2021 are usable, while we will only store seven zettabytes. We cannot store/process most of this data in the cloud due to the exponential increase in data generation [3]. Furthermore, some data are private or confidential, and we should not share them with third parties (e.g., medical images) [4]. Besides, edge devices are becoming more and more powerful, allowing us to use them in training ML models and not just for inference [5,6]. We cannot use them yet for training the larger models. However, we can use medium models that may be very useful for a wide range of tasks such as learning users' activities while maintaining their privacy [7], for federated vehicular networks [8] or for predicting air pollution [9].

Two related proposals [10] have recently emerged that address network congestion of sending all user data to the cloud and data privacy issues. On the one hand, *edge computing* (EC) [3] proposes

* Corresponding author.
E-mail addresses: jamorell@uma.es (J.Á. Morell), eat@lcc.uma.es (E. Alba).

[1] www.statista.com/statistics/1101442/iot-number-of-connected-devices-worldwide.

[2] blogs.cisco.com/sp/five-things-that-are-bigger-than-the-internet-findings-from-this-years-global-cloud-index

https://doi.org/10.1016/j.future.2022.02.024

to increase the amount of processing on edge devices while reducing data exchange with the cloud. On the other hand, *federated learning* (FL) [11–13] deals with training a shared ML model in a distributed (non-centralized) manner while keeping private data locally on edge devices. The combination of both is known as federated edge learning (FEEL) [14]. However, they also add new challenges such as unbalanced and non-identically distributed data (e.g., *non-i.i.d.*) [15], heterogeneity of hardware (HW) and software (SW), dynamicity of the environment (i.e., edge devices are managed by volunteers or are unreliable) [6], and security. The challenges are detailed in Section 2.1. We can see FEEL as a type of Volunteer Computing (VC) [16–20] distributed system where users, and devices (i.e., volunteers), with local and private datasets voluntarily donate their computing resources to a project and collaborate to train a shared ML model.

Despite the increase in FEEL research in recent years and its many real-world applications, there is still no open-source software platform that meets all the challenges and desirable features for a volunteer, decoupled, asynchronous, and scalable FEEL platform that allows researchers to study new techniques using real volunteers and resource-constrained edge devices. A software platform that meets FEEL purposes must be *(i)* modular (to allow easy addition of new algorithms), *(ii)* fault-tolerant, *(iii)* scalable, *(iv)* accessible (to provide easy connection of new volunteer participants with minimal user configuration), *(v)* adaptive (ready to adapt learning to variations in connected edge devices), *(vi)* and prepared to deal with heterogeneous hardware and software.

Most published works use preconfigured parameters in controlled laboratory environments [10–13,21,22]. They use a fixed number of workers or always take a constant subset of all available ones at each iteration, needing a pre-configuration identifying the participant devices before learning starts (devices do not connect and disconnect during learning). They assume that there are always thousands or millions of devices connected, choosing some of them in each iteration. However, they do not consider the possibility that the number of devices may decrease to just a few (or even zero) and how they must adapt the algorithm accordingly. They usually use synchronous training algorithms having to wait for the slowest node to continue learning in each iteration. Nevertheless, this may not be practical in a realistic environment where devices are unreliable (can disconnect at any time) and heterogeneous with diverse performances. We think, in real-world problems, we must use asynchronous training allowing collaborators devices to join or leave during learning. Also, these approaches do not consider the interoperability [23] of the participating devices. Some of them deal with multiple performances, but none with the diverse software, like various learning libraries. Furthermore, they need a centralized node to manage the whole process and take care of the partial results mergers.

Some of the papers address part of these points, but none cover all of them. As a result, there is a need for an open and modular software platform in a decentralized, asynchronous, fault-tolerance, and adaptive way to help researchers explore new techniques and ideas in FEEL. In this work, we define the challenges and desirable features for a volunteer, decoupled, asynchronous, and scalable FEEL platform. We propose an algorithm for FEEL that adapts to asynchronous clients joining and leaving the computation. We propose, implement and evaluate a new software platform for distributed deep learning on volunteer edge devices. We then evaluate its results on problems relevant to FEEL. We performed twenty experiments organized into four case studies and used from 1 to 64 workers. We use two types of workers: Python processes (TensorFlow) limited to one core and 2 GB RAM on HTCondor [24] as constrained devices and up to 24 desktops collaborating from web browsers (TensorFlow.js). We analyze how asynchrony, dynamic connections, and disconnections of devices affect learning and how we can adapt learning to keep high accuracy.

We show that the proposed decentralized and adaptive system architecture for asynchronous learning allows volunteer users to yield their device resources and local data to train a shared ML model. Devices can join and leave the system at any time without stopping the learning process. This architecture does not need to know where or when the participating devices will connect. This system is fault-tolerant, so unexpected disconnections of workers do not interrupt the learning. Besides, our open-source implementation enables collaboration between devices with heterogeneous hardware and software. User devices can join the learning in different ways, such as from a web browser or by running a Python process on the client device, allowing almost any device to participate. All code is available in a public Git repository, and it is packaged in Docker containers to allow for reproducibility[3] (programmability and usability).

Finally, we conducted an exhaustive empirical analysis for the evaluation of the proposal. We analyzed the main features of the platform mentioned above. We organized the study into four case studies:

- **Case study 1:** We analyze the robustness and interoperability of the platform when using heterogeneous hardware and software. We demonstrate that the platform achieves high accuracy despite heterogeneity.
- **Case study 2:** We examine the most suitable setting when using asynchronous FEEL. We show that it is necessary to adapt the aggregation parameter when the number of workers varies to get high accuracy.
- **Case study 3:** We test the fault-tolerance of the software platform to variations in the number of collaborating devices. We also evaluate the learning algorithm self-adaptation to these changes. We prove that the platform is fault-tolerant, can continue its learning, and keep offering a high accuracy despite connections and disconnections of volunteer devices.
- **Case study 4:** We evaluate the self-adaptation and fault-tolerance of the platform using a more extreme *non-i.i.d* data. Also, we perform an analysis of the scalability and random drops. We confirm that when the system uses the adaptive algorithm and random drops, we can get similar results as when the system uses a constant number of workers during learning.

In summary, the main contributions of this work are:

1. We define the challenges and desirable features for a volunteer, decoupled, asynchronous, and scalable FEEL platform.
2. We propose an algorithm for FEEL that adapts to asynchronous clients joining and leaving the computation.
3. We propose, implement and evaluate a software platform for performing FEEL that meets the defined challenges and desirables features. We evaluate our proposal via extensive experimentation in a static configuration and highly dynamic and changing scenarios. We then show that the platform adapts well to this changing environment getting a numerical accuracy similar to today's configurations using a given number of homogeneous (hardware and software) computers as a static platform for learning. Next, we demonstrate the fault-tolerance of the platform that recovers from unexpected disconnections of volunteer devices.

[3] github.com/jsdoop/jsdoop.

Table 1
Desirable features for volunteer computing platforms.

Desirable feature	Description
Accessibility	Users must connect to the platform easily
Adaptability/Dynamicity	The environment is ever-changing, devices join and leave at will
Availability	A highly available environment has minimal service interruption
Fault-Tolerance	The platform should be tolerant of failures and disconnections
Heterogeneity	Connected devices may have different performance, HW and SW
Programmability	Developers should be able to add new features to the platform quickly
Scalability	The platform must handle a growing number of connections
Security	The machines of the volunteers should not be compromised
Data privacy	The access to customer's data must be restricted and remain private
Usability	The platform has to be easy to deploy and use

4. We released a modular open-source library that covers most FEEL and VC desirable features.

The following section briefly presents the technical background and related works. Section 3 introduces the proposed software platform. Section 4 shows the experimental study and the results. Section 5 discusses the threats to validity. Lastly, Section 6 outlines the main conclusions and proposes future work.

2. Technical background and related work

FEEL enables devices to collaboratively learn a shared ML model while keeping all training data on the device. In this work, we use a convolutional neural network (CNN) as a ML model. Sometimes we interchangeably use the term deep learning (DL) instead of neural network (NN). That is commonly accepted when we refer to a NN with multiple nonlinear layers [25]. In this section, we first introduce the main challenges of FEEL and the desirable features of VC. We then briefly describe NN and distributed NN training.

2.1. FEEL and VC: Challenges and desirable features

In this subsection, we present the main challenges of FEEL and the desirable features of VC platforms [20,26,27] (see Table 1). As we will see, both concepts share many similarities. We are faced with heterogeneous hardware and software, asynchronism, fault-tolerance, and devices arbitrarily joining and leaving the system, among other things. In Section 3, we explain how our proposal covers this.

Adaptability/Dynamicity. In FEEL, edge devices usually are managed by volunteers, or if not, their connections are unreliable, so we cannot be sure that they will always be available. In either case, devices can join and leave the learning at any time, thus the platform must be ready for unexpected disconnections and failures. That is the same problem that occurs in traditional VC systems, so we follow the same guidelines. Besides, the variations in connected devices during learning implies that the dataset available for training at any given time also varies. Traditional distributed NN training algorithms use highly controlled environments with high connection speeds where the data is divided into *i.i.d.* (i.e., sufficiently randomly disordered to make the order of the data unrelated) balanced datasets [28]. However, when we perform FEEL, we keep the data locally to maintain data privacy, so we are sure to face unbalanced and not identically distributed data (e.g., *non-i.i.d.*) [15,29].

Availability. A highly available environment has minimal service interruption. Like most distributed systems, a VC platform must be as decoupled as possible. Each element of the system must be independent of the others components. They should communicate with each other in a decoupled manner, e.g., via REST APIs. The platform must be ready to replace actors in case of failures and disconnections. It is necessary to adopt a decentralized approach. Our proposal follows these guidelines (see Section 3).

Fault-Tolerance. The platform should be tolerant of failures and disconnections. As mentioned above, volunteer users' devices can be disconnected at any time without prior notice due to errors, connection failures, or because the user decided to stop collaborating. Therefore, the platform must continue learning despite these obstacles. There are different techniques for tackling fault tolerance, such as checkpointing, redundant processing, and heartbeat monitoring [27]. As for checkpointing and heartbeat, they are useful when the processing time before communication is high. In our work, each processing time before communication is low. Also, we wanted to minimize communications and reduce coupling. Therefore, we preferred to use a maximum time threshold between result communications to determine when a volunteer was disconnected or lagging (see Sections 3 and 2.2.4). Nevertheless, we plan to consider adding both in the future if longer processing times are required. Regarding redundancy, other authors claimed that these techniques have to deal with a large amount of redundant computing and that new methods need to be found to improve the efficiency of big data processing on VC [30]. Although some authors propose to use these techniques with minimal impact on performance [27], distributed training of a NN is slow, and it is necessary to find techniques that avoid redundancy to speed up processing. Furthermore, in FL, data is located on the edge devices and is private, so we cannot guarantee redundancy in processing as each device will compute different data. In our case, we circumvent all the mentioned weaknesses by going for an adaptive algorithm that continues job processing despite the failures of the volunteer nodes.

Scalability. The platform must handle a growing amount of connections. The decentralized approach and the decoupling between the different actors involved are paramount to achieve this. Our proposal is based on these principles. Related to scalability is bottleneck in communications. There are different techniques to reduce the communications bottleneck during distributed DL (in Section 2.2 we detail distributed DL) such as *model quantization*, *quantization of gradients/weights*, *sparsification of gradients/weights* [29,31,32]. They relate to compressing the ML model (before or after the training), compressing gradients/weights, or sending only the most representative values of gradients/weights above a specific threshold. Another way is by increasing *local computation*, allowing nodes to train their local ML model using their local data for more iterations before synchronizing with the global shared model. Besides, to avoid waiting for the slowest node, it is possible to perform asynchronous training allowing the aggregator for summarizing device results of different ages. In this case, we can use a threshold [33] to discard old results (i.e., bounded asynchronous training) or/and a different learning rate depending on how outdated they are. Likewise, data access can be another bottleneck. In traditional distributed DL, data is divided among the participating nodes before training

begins and shuffles after each epoch. In FEEL, the training data is dynamic and remains local. However, local devices may access the same database on a local network while collaborating with other remote devices. The best way to deal with this problem is to use random access to data, the equivalent of sampling with replacement [15,34,35]. It allows accessing data simultaneously and without waiting for each other. Our proposal makes use of several of these techniques (see Section 3).

Heterogeneity. Connected devices may have different performance, hardware and software.

- *SW Heterogeneity.* Using FEEL, interoperability of each node's dataset is a prerequisite. Also, each node may use different software, including a distinct NN learning library. They exchange gradients, weights, and model topologies with the server and the other devices. The Open Neural Network (ONNX)[4] tries to achieve interoperability of the various NN libraries but is not yet fully compatible with all of them. Also, software containers, such as Docker,[5] allow us to package and deploy algorithms into standardized services and applications. Our proposal solves this problem and allows heterogeneous hardware and software to collaborate in learning (see Section 3).
- *HW Heterogeneity.* FEEL deals with a heterogeneous set of clients with different performances, such as smartphones and IoT devices. To address this problem, some researchers [13,36] have proposed algorithms to adapt learning to devices with various performances. For instance, each node may use customized local steps based on its power. Although the vanilla FedAVG [12] uses random node selection, it is also possible to use an intelligent node selection algorithm [27] to deal with HW heterogeneity. In this case, node selection should be based not only on the performance of the edge devices but also on the distribution of the edge devices' data. We do not use this technique in this work but plan to investigate it further in future work. Despite the heterogeneity, most works typically use synchronous learning algorithms. They are often more accurate, simpler to apply and make the results easier to analyze. In these papers, the experimentation uses fixed and preconfigured workers, and no connections and disconnections occur dynamically during learning. We think this does not fit into a real scenario where all devices are not always available but are dynamic, and thus the algorithm must adapt to these changes and continue with learning. We believe that the natural way to deal with this is to use asynchronous learning despite its challenges and difficulties. We are aware of the risks of this scenario. However, we firmly believe that this has to be the ultimate goal of FEEL.

Accessibility. Volunteers must connect to the platform easily while avoiding complex configurations or installations. In FEEL, users yield their device resources and local data to train a shared ML model. However, for this to happen, they must be able to do it simply without much hassle. Besides, it is possible to use gamification [37] to make users stay connected for longer and feel that they are participating in a global project for the good of science and society. Our proposal allows volunteers to easily connect to the platform by just running a simple Python process or through a web browser (see Section 3). Plenty of IoT devices allow running web browsers such as Raspberry Pi, mobile phones, smart TVs, Jetson Nano or Alexa. We indeed emphasize the importance of being able to connect through a web browser [20]. If users do not have to install anything, it is easier for them to collaborate as volunteers in collaborative projects. Allowing workers to connect using Python or web browsers ensures that virtually any device can use the system.

Programmability. Developers should be able to add new features to the platform quickly. For a software platform to be accepted by the research community, it must be open-source to allow for reproducibility. It must also be modular and easily extensible so that researchers can effortlessly add new techniques and algorithms. The proposed platform's code fulfills all of these requirements.

Usability. The platform has to be easy to deploy and use. Researchers must be able to deploy the software platform on their machines quickly. For this purpose, it is essential to use packing containers such as Docker, allowing us to forget the problems of dependencies between different software libraries. The proposed software platform is packaged in Docker containers for ease of deployment and use by researchers.

Data privacy. The access to customers' data must be restricted, and they must remain private. This aspect is the cause of the emergence of FL aiming to learn a global ML model from end-users data without accessing the data directly at any time, these always remaining private and on local devices.

Security. There are many challenges related to user data security and privacy [38]. However, these issues are outside the scope of this work.

2.2. Distributed neural network training

Backpropagation is a widely used algorithm for training feedforward NNs [1,2,39]. Its goal is to minimize the difference between the actual output and the predicted output of the NN (i.e., the error). The loss function computes this error (e.g., mean square error) and propagates it to previous layers. The optimization function (e.g., gradient descent) calculates the gradients, i.e., the partial derivative of the loss function with respect to weights, and the weights are modified in the opposite direction of the calculated gradient.

Gradient descent (GD) is the most common method to optimize NNs [1,2,39]. There are three variants of GD. Vanilla gradient descent (i.e., batch gradient descent) computes the gradient using the entire training dataset. Stochastic gradient descent (SGD) performs a parameter update for each training example. Mini-batch gradient descent approximates the partial derivative over the loss function using a randomly sampled subset of the data (i.e., a mini-batch). The last is the standard for training a NN, and the term SGD is also often used when we use mini-batches. Many algorithms improve on the basic SGD algorithm, such as Adagrad, RMSprop, Adam, and more [39]. In this work, we use distributed mini-batch gradient descent (distributed SGD from now on). In the experimentation, we specifically employ the RMSprop variation.

2.2.1. Mini-batch gradient descent

Here we present a canonical SGD (see Table 2 for main notation summary). We define D as a dataset. For each data example j, we define the loss function [40] as $f(w, x_j, y_j)$, which we write as $f_j(w)$ in short. For each mini-batch of data B where $B \subseteq D$, we define the loss function obtained after applying the mini-batch B_k in iteration t as:

$$F_{k,t}(w_t) = \frac{1}{|B_k|} \sum_{j \epsilon B_k} f_j(w_t) \tag{1}$$

Then, we update the weights of the model using an optimization function, e.g., if we use canonical SGD:

$$w_{t+1} = w_t - \eta \nabla F_{k,t}(w_t) \tag{2}$$

where $\eta > 0$ is the step size (i.e., the learning rate).

4 onnx.ai.

5 www.docker.com.

Table 2
Main notation summary.

$w_{i,t}$	Local model weights of node i at time t
w_t^{global}	Global model weights at time t
$f_j(w)$	Loss function for each data example j
$F_{k,t}(w_t)$	Loss function for minibatch k at time t
η	Learning rate
t	Iteration index
T	Total number of global aggregations
τ	Number of local update steps between two global aggregations
N	Maximum number of edge devices (connected or disconnected)
E	Current set of connected edge devices
P	Number of gradients used to calculate the new weights
D	The entire dataset
D_i	Local dataset in node i
B_k	Minibatch k of data
Z	A threshold variable to accept or discard old gradients
S	Time of inactivity before disconnection happens

2.2.2. Canonical distributed mini-batch gradient descent

A canonical synchronous distributed gradient descent algorithm [12] is shown in Algorithm 1. Assuming we have N edge nodes with local datasets $D_1, D_2, \ldots, D_i, \ldots, D_N$, $\cup_i^N D_i = D$ and $D_i \cap D_{i'} = \emptyset$ for $i \neq i'$.

Algorithm 1 Sync Distributed Gradient Descent.

Input: τ, T
Output: Final model parameter $w_{t=T}^{global}$
1: Initialize: $w_{t=0}^{global}$, $w_{i,t=0}^{local}$ to the same value for all i;
2: **for** $t = 1, 2, ..., T$ **do**
3: **for** $u = 1, 2, ..., \tau$ **do**
4: Each node i, in parallel, compute local update (3)
5: **end for**
6: // Each node sends local changes sync
 // Global aggregation in server (4)
 // Each node load global weights
 Set $w_{i,t} \Leftarrow w_t^{global}$ for all i;
7: **end for**

Each node i has its local model parameter $w_{i,t}$ where t is the iteration index. At $t = 0$, the local parameters for all nodes i are initialized to the same value. In each iteration, a *local update* is performed using an optimization function, e.g., if we use canonical SGD:

$$w_{i,t+1} = w_{i,t} - \eta \nabla F_{i,k,t}(w_{i,t}) \tag{3}$$

After one or multiple local updates (τ), a *global aggregation* (4) is performed through the aggregator to update the local parameter at each node to the weighted average of all nodes parameters.

$$w_t^{global} = \frac{1}{N} \sum_{i=1}^{N} w_{i,t} \tag{4}$$

2.2.3. Adaptive distributed mini-batch gradient descent

In adaptive distributed mini-batch gradient descent, the number of connected edge nodes is dynamic and varies over time. We assume that the maximum number of edge nodes is N (connected or disconnected), that each local dataset D_i is unbalanced and *non-i.i.d*, $\cup_i^N D_i = D$ and $\forall_i \forall_j \left| D_i \cap D_j \right| \geq 0$. We define the current set of connected edge nodes as E where $|E| \leq N$. We dynamically adapt the *global aggregation* to the current connected edge node devices using:

$$w_t^{global} = \frac{1}{P} \sum_{y=1}^{P} w_{local_y} \quad , \quad P \simeq |E| \tag{5}$$

We define $|E|$ as the number of connected edge node devices (workers) and P as the number of gradients the aggregator accumulates each aggregation. The adaptive algorithm dynamically adjusts $P \simeq |E|$. We refer to connected nodes $|E|$ as the nodes that have communicated with the logical server(s) in the last S seconds (see Section 3.1). Also, we use asynchronous training, so devices do not have to wait for each other. Therefore, a new threshold variable Z has been defined to accept or discard old gradients depending on how old they are (see Algorithm 2).

Algorithm 2 Async-Adapt Distributed Gradient Descent.

Input: τ, Z
Output: Final model parameter $w_{t=T}^{global}$
1: Initialize:
 // Each worker loads global weights when connected.
 $w_{i,t} \Leftarrow w_t^{global}$;
 // Aggregator initialize list of weights for aggregation
 W <- [] ;
2: **while** termination criterion is not met (each worker async) **do**
3: **for** $u = 1, 2, ..., \tau$ **do**
4: // Each node i compute local update (3)
5: **end for**
6: // Each node sends local weights $w_{i,t}$
7: // Aggregator receives the local weights async
 // Aggregator receives info of connected workers $|E|$ and update P
8: $P = |E|$
9: // Aggregator checks if received weights are too old
10: **if** $t_i + Z >= t_{global}$ **then**
11: // Aggregator adds weights to list for aggregation
 $W.insert(w_{i,t})$
12: **end if**
13: **if** $W.size() >= P$ **then**
14: // Global aggregation (5)
15: $W = []$
16: **end if**
17: **end while**

2.2.4. Learning communications cost

In vanilla FL [12], the total number of connected workers can be thousands, and the subset of workers participating in each iteration is a fixed number of them chosen randomly. In this paper, we deal with the problem where the number of workers is low and can even be zero during training and how the algorithm must adapt to these changes.

In our adaptive asynchronous approach, when a worker first connects, it downloads the job setting $C_{j_{get}} + C_{j_{res}}$ (get request and response). Later the model topology $C_{t_{get}} + C_{t_{res}}$, and finally the current state of the problem (i.e., weights) $C_{w_{get}} + C_{w_{res}}$. Then iteratively, it processes the local steps, return the results (i.e., gradients/weights) and again downloads the current state of the problem $C_{g_{post}} + C_{w_{res}}$ (post request with results and response with updated weights) until the completion criteria are completed (see Section 3.2). However, the size of the problem settings and the model topology so do the communication time needed for its transmission are negligible compared to the size of the NN weights. Besides, the size of the current state weights $C_{w_{res}}$ is similar to the size of the gradients sent $C_{g_{post}}$. We refer to each one as C.

$$C_{j_{get}} + C_{j_{res}} + C_{t_{get}} + C_{t_{res}} + C_{w_{get}} + C_{w_{res}} \simeq C_{w_{get}} + C_{w_{res}} \tag{6}$$

$$C_{w_{get}} + C_{w_{res}} \simeq C_{w_{res}} \tag{7}$$

$$C_{g_{post}} \simeq C_{W_{res}} = C \tag{8}$$

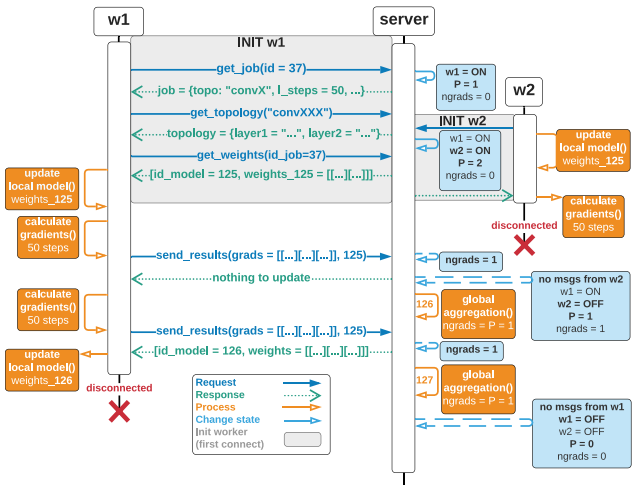


Fig. 1. High-level execution flow. P changes accordingly to the number of connected devices. When the number of gradients ($ngrads$) is equal to P a global aggregation is performed. In this example, two global aggregations occurred using $P = 1$, and C_{total} is equal to 5. $W1$ received weights and sent gradients 2 times. $W2$ received weights once. $W1$ received an empty buffer once because the global model did not change.

When we use a constant and fixed number of workers N without connections and disconnections, and we set P equal to the number of connected workers $|E| = N = P$, the total communication would be:

$$C_{total} = 2 \cdot C \cdot N \cdot T \tag{9}$$

In this case, we can see how the total amount of data transmitted depends on the number of connected nodes $|E| = N$ and the number of global aggregations T we perform. Also, we see that the data batch used in the global aggregation depends on $|E|$ (the more workers used, the larger the data batch used in each global aggregation step).

When we allow connections and disconnections of workers during learning (see Fig. 1), we match P to the number of connected workers dynamically $P \simeq |E|$. In this case, the size of the data batch at each global aggregation step varies during learning according to $|E|$. In addition, if we assume that when a worker goes offline, it has already downloaded the current state of the problem $C_{w_{get}}$ before going offline, then the total amount of communications C_{total} would be:

$$C_{total} \simeq A \cdot C + \sum_{t=1}^{T} |E|_t \cdot 2 \cdot C \tag{10}$$

where A is the total number of disconnections during learning and t is the ID of global aggregation. In this case, C_{total} depends on $|E|$ in each t and A, i.e., it depends on how reliable the connected workers are. That is a problem more related to fault tolerance [41] and is common in volunteer computing. To solve it, we have to find a balance between the probability of failure of a task and the execution time (e.g., using a custom number of local steps). We want to face it more deeply in future. Nevertheless, in our approach, when a disconnection occurs, $|E|$ decreases, fewer gradients are used for a global aggregation, and therefore C_{total} can be lower than when $|E| = N$ with no disconnections.

3. The proposed platform

In this section, we present the design decisions. Next, we describe the flow of execution of the proposed platform.

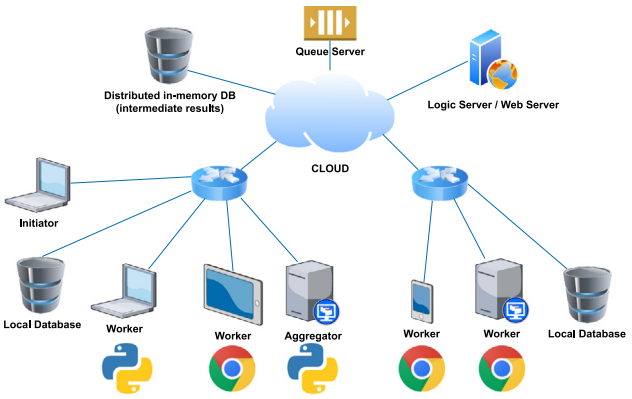


Fig. 2. High-level system architecture.

3.1. Design decisions

In this subsection, we introduce the design decisions we made to design a software platform to meet the FEEL challenges presented above and with the desirable features of VC platforms [20, 26] (see Table 1). We follow a decentralized approach in which the actors are as decoupled as possible. There are six types of actors involved (see Fig. 2).

1. *The initiator* is the user who creates the problem to be solved (i.e., *job*). It participates before starting the training to specify the required parameters.
2. *The workers* are the user edge devices that yield their computing resources and data in learning a shared ML model. These devices may be very different from each other, from smartphones and IoT devices to laptops and desktops.
3. *The aggregator* can be an edge device or a remote machine. It is in charge of averaging the results calculated by the workers and updating the shared ML model.
4. *The logical server* can be in the cloud or the edge network. It is a server in charge of the orchestration and job scheduling.
5. *The distributed in-memory database (DIMDB)* is in the cloud and is the place where the *logical server* stores the intermediate results (i.e., gradients/weights).
6. *The queue server* is in the cloud and is used by the *logical server* to inform the *aggregator* that new results are available.

We distinguish between *jobs* and *tasks*. A *job* is the whole NN training process we want to complete, and it has an ID associated with it. There may be multiple *jobs* being solved at the same time by different workers. A job contains the parameters of the NN training process, such as the NN model topology, the optimization algorithm, the loss function, the learning rate, and the termination criterion (e.g., the number of global aggregations). We call the user who creates a job and uploads it to the platform an *initiator*. We call a *task* each local processing that is performed on a worker to complete the job. Each one has a task ID associated with it.

The *logical server* is in charge of the orchestration and job scheduling. It is only an intermediary that redirects messages between the participants and the *DIMDB*. It can be in the cloud or the edge network. The *DIMDB* is in the cloud and is the place where we store the intermediate results (e.g., *tasks* solved by a worker that are gradients/weights). When the *logical server* receives results from the workers (i.e., gradients/weights), it stores them in the *DIMDB*. Next, it sends a message to a message queue

associated with a job ID in the *queue server* to inform that this result is available.

The *aggregator* is a process subscribed to a message queue of a job ID waiting for results. When it receives a message that a new intermediate result is available, it downloads it from the *DIMDB*. The *aggregator* uses them (i.e., gradients/weights) to calculate the new global shared model. Subsequently, the *aggregator* sends the updated model to the *logical server* who stores it in the *DIMDB*. The *aggregator* can be in the cloud or the edge network.

Workers are volunteer devices that collaborate by solving *tasks*. They can connect and disconnect at any time for different reasons during the execution. Therefore, when a worker connects, it downloads the current state of the job, computes locally for the set number of local steps, and then sends the results to the *logical server*. Thus, they can connect with a job ID that had previously started collaborating.

The *aggregator* uses an adaptive algorithm to adjust the number of gradients to aggregate to the number of connected *workers* (see Section 2.2.3). Therefore, the *aggregator* needs to know how many *workers* are connected to adapt the aggregation parameter *P*. To this end, the *logical server* stores the worker's information from the *workers* who have communicated with it in the last *S* seconds. This information is included in the message that the *logical server* sends to the message queue. Thus, the *aggregator* can know how many *workers* are connected and adapt the aggregation parameter accordingly. Note that the more gradients the *aggregator* accumulates, the bigger is the batch size used in each aggregation. As we can see, this decentralized approach allows multiple *logical servers* as load balancers connected to the same *DIMDB* (availability and scalability).

When *workers* stop communicating with the *logical server*, it considers that they have disconnected, deletes them, and self-adapts the learning to the number of available devices (adaptability/dynamicity). Therefore, *workers* can disconnect unexpectedly, freeze or otherwise fail without stopping global learning (fault-tolerance). Also, asynchronous training allows the platform to continue the learning process without waiting for slower devices. Each job has a threshold parameter for accepting old results from *workers*. Thus, results may arrive late from slower devices, and it is up to the aggregation technique used to decide how best to aggregate them (e.g., discarding old results or using different learning rates in aggregation). Furthermore, since workers in the same local network can access the same local database, we use random access to the local database to avoid synchronization with the rest of the local *workers*.

All participants interact through the *logical server* using REST API protocol. Workers can connect to the platform using a web browser by following a web link (accessibility). That maximizes the number of collaborators by requesting a minimum effort (just a click) from volunteers. The platform also supports connections from devices that do not have a web browser by running a Python process.

In Python, we use the TensorFlow library, and in the web browser, we use TensorFlow.js, a JavaScript library for training and inference directly in the browser or Node.js (interoperability and SW heterogeneity). In recent years, the performance of programs executed in the web browser has improved and is already approaching that achievable with native code. These advances lead us to believe that web browser computing is the way forward for this type of volunteer computing thanks to its ubiquity, sandboxing, and no need for software installation [42]. It is present in desktops, many IoT devices, and smartphones (HW heterogeneity).

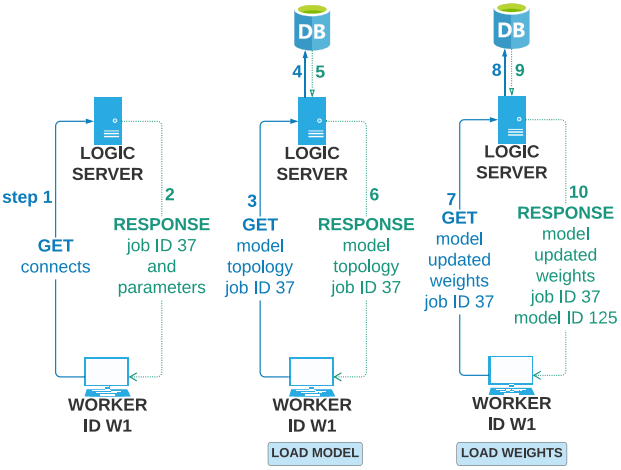


Fig. 3. Initialization execution flow. The worker obtains all the necessary information to start collaborating.

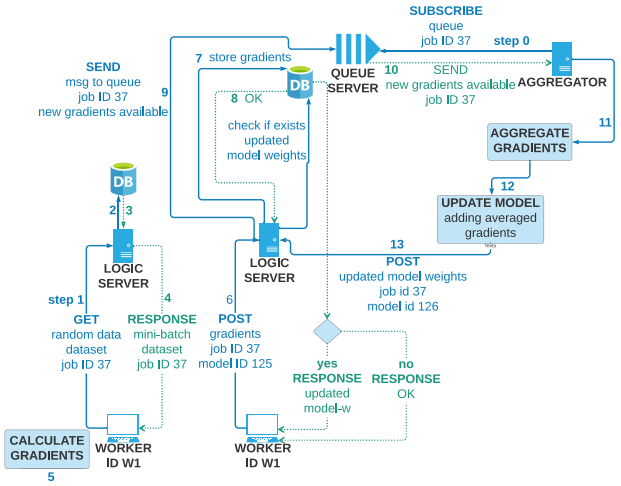


Fig. 4. Execution flow of the collaborative learning process.

We use Redis[6] for *DIMDB*, RabbitMQ[7] (AMQP protocol) for the *queue server*, the HDF5 format[8] to share the model topology, and the NPY file format[9] to share the weights and gradients between the diverse NN libraries in web browsers and Python processes.

The software platform is modular, open-source, packaged in Docker containers, and is publicly available in a Git repository to allow for reproducibility. The flow of execution is detailed in Section 3.2.

3.2. Flow of execution

In this subsection, we explain the platform execution flow of a distributed learning process (see Figs. 3 and 4). Note that we previously used the term gradients/weights because we can follow both approaches. For simplicity, we will use the word *gradients* to refer to the workers' results and *weights* to refer to the model.

An *initiator* setups a problem and creates a *job*. A *job* contains all the information needed to train the NN. If the NN model

6 redis.io.

7 www.rabbitmq.com.

8 www.hdfgroup.org/solutions/hdf5/.

9 https://numpy.org/devdocs/reference/generated/numpy.lib.format.html.

topology associated with the *job* is not stored in the remote *DIMDB*, he/she must first upload it and associate a name to it. Also, he/she initializes the weights of the shared ML model and uploads them to the platform. Now, the *job* is ready for the *workers* to collaborate.

A *worker* requests *job* information from the *logical server* using the job ID (see steps 1 and 2 in Fig. 3). The server responds with the job information needed to know how to solve that *job*. The *worker* then downloads the NN model topology associated with that *job* (see steps 3 to 6). It then downloads the current weights. A *worker* can connect when the training process had already started, so the weights may not be the initial ones. The current weights have an associated ID that indicates the number of global aggregations performed (see steps 7 to 10). Now, the worker has all the necessary information to start collaborating.

The platform allows access to data locally or remotely, allowing in the latter case for a traditional distributed DL (see steps 1 to 4 in Fig. 4). When we use FEEL, the access to data is done through a database deployed in the edge device or another device in the same local network (i.e., keeping data privacy) with the data ready to perform the training and not through the logical server. In either case, data requests use the same REST API through an intermediate module and are transparent to the user.

After the *worker* gets the data, it calculates the gradients for the local steps indicated in the job information. Next, it sends the accumulated gradients in those steps to the *logical server* (see step 6 in Fig. 4). Then the *logical server* stores them in a remote *DIMDB* (see steps 7 and 8). Later, it sends a message to the *queue server* notifying that those results are now available (see step 9).

The *logical server* then checks for updated model weights related to that job ID in the remote *DIMDB*. If there are new weights, the server responds to the *worker* with a buffer containing those weights. Otherwise, the server responds to the *worker* with an empty buffer. Next, if the *worker* receives new up-to-date weights, it updates the model. The *worker* then iterates again to calculate new gradients. This process is repeated as a loop until a termination criterion is satisfied. We use the number of global aggregations as the termination criterion, but the library is modular, and it is easy to add new different termination criteria.

The *aggregator* is another machine that can be an edge device or a remote one and is subscribed to a message queue associated with a job ID (see step 0 in Fig. 4). When messages arrive at the queue notifying newly available results (see step 10), the *aggregator* requests the server for these results from the remote *DIMDB* (we do not show this request in the figure for the sake of simplicity).

When the *aggregator* has the required number of gradients, it calculates the new model weights using the selected aggregation technique (e.g., the most common is averaging) (see step 11 and 12 in Fig. 4). The number of gradients to be summarized can be set or adaptive depending on the number of workers connected. It then sends the new weights from the shared ML model to the *logical server* (see step 13). We use a threshold Z to discard old gradients (see Algorithm 2).

4. Experimental study

In this section, we present a deep experimental study divided into five subsections. The first subsection describes the experimental settings. The next four subsections describe the four case studies. The first analyzes the robustness and interoperability of the platform when using heterogeneous hardware and software. The second one examines the most suitable configuration when using asynchronous FEEL. In the third one, we tested the fault-tolerance of the software platform to variations in the number of collaborating devices during learning. In the fourth one, we evaluate the self-adaptation and fault-tolerance of the platform using a more extreme *non-i.i.d* data. Also, we perform an analysis of the scalability and random drops.

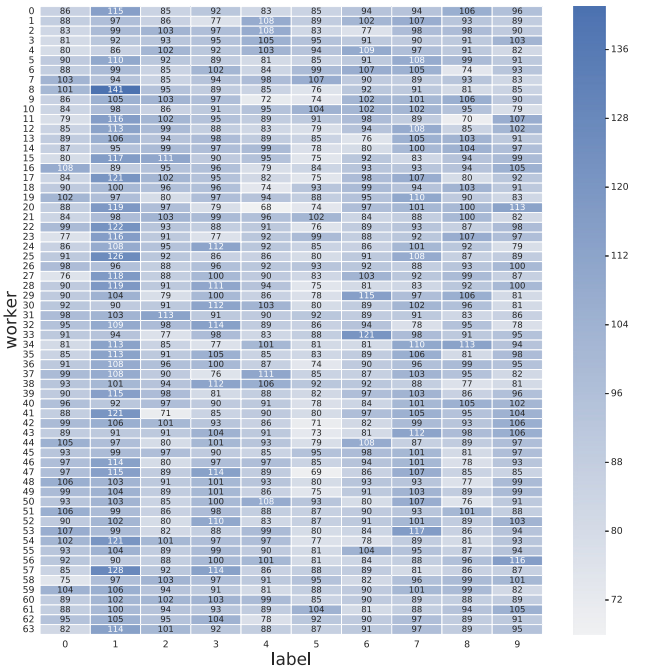


Fig. 5. Number of samples of each label in each worker in case studies 1, 2, and 3. Each worker selects portions of the dataset randomly.

4.1. Experimental settings

For experimentation, we train a CNN [25,43] (12 layers and 887,530 trainable parameters) to predict the Mnist dataset [44]. The Mnist dataset contains 70,000 images of handwritten digits: 60,000 for training and 10,000 for testing. In all experiments, the models were trained using the training set, and then we evaluated them using the test set. All accuracy and loss results shown in all figures refer to the test set. Besides the accuracy and loss, we report the *Cohen's Kappa* coefficient (CK score) [45]. In multi-class classification, metrics such as accuracy do not provide enough information of our classifier's performance. In contrast, the CK coefficient is an extremely valuable metric that can handle very well both multi-class and imbalanced class problems.

We use the RMSprop optimizer (decay = 0.0, rho = 0.9, momentum = 0.0, epsilon = 1e-08, centered = False), mean squared error as loss function, and a learning rate $\eta = 0.001$. Both the topology and the model parameters are from an example in a Kaggle notebook[10] to allow for reproducibility. We performed twenty experiments organized into four case studies and used from 1 to 64 workers. We use two types of workers: Python processes (TensorFlow) limited to one core and 2 GB RAM on HTCondor [24] as constrained devices and up to 24 desktops collaborating from web browsers (TensorFlow.js). The aggregator runs in a Python process using the TensorFlow library. The logical server that provides access to the database run in a Java process. The dataset, model topology, gradients, and weights are stored in an in-memory database (Redis). The last three are different processes, but they all run on the same machine in our experimentation. All the computers used in the experimentation were interconnected in an Ethernet LAN 100 Mb. That is an average Internet speed similar to that available to most users.[11]

To perform FL, each worker only has access to a fraction of the dataset. In case studies 1 to 3 (see Fig. 5), the different

[10] www.kaggle.com/yassineghouzam/introduction-to-cnn-keras-0-997-top-6.
[11] worldpopulationreview.com/country-rankings/internet-speeds-by-country.

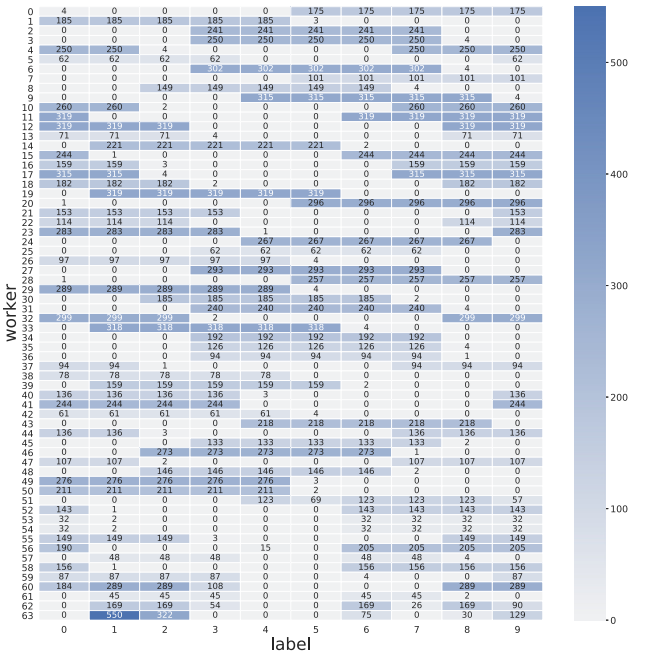


Fig. 6. Number of samples of each label in each worker in case study 4. Each worker has a different number of samples of each class and a maximum of five classes.

fractions may overlap, are unbalanced, and *non-i.i.d*. We divided the dataset into mini-batches of size 8. We use a maximum of 64 workers ($N = 64$). For experimentation, each worker randomly chooses (using a different seed depending on the worker ID) the mini-batches that are part of the local dataset of size 1/64. Hence, some data may be overlapping in local datasets of different workers. That is something that could happen in the real world. In case study 4 (see Fig. 6), we use a more extreme *non-i.i.d* data, partitions do not overlap, each one has a maximum of five classes, and the number of elements in each one can be different.

Each worker performs 50 local steps before communicating with the server. We use bounded asynchronous training. Therefore, workers can have different models for a certain period and then load the shared model. That means that we must add a new parameter to decide which weights will be too outdated to be used. We allow the aggregator to use results from up to 5 previous generations ($Z = 5$) to calculate the new global weights. The aggregation technique used is averaging selected results. Finally, the termination criterion is to reach 300 global aggregations.

4.2. Case study 1: robustness and interoperability

In this case study, we analyze the robustness and the interoperability of the software platform when using heterogeneous HW and SW. For this, we experiment with 64 heterogeneous devices. We use 40 Python processes running TensorFlow on constrained devices (one core and 2 GB of RAM) and 24 desktops running TensorFlow.js in the Chrome web browser. We call this experiment *mixW64P64*.

Then, we compare *mixW64P64* with a new one when using 64 homogeneous constrained devices running a Python process. We call this experiment *W64P64*. In case study 1, the number of workers and the aggregation parameter are fixed and constant throughout the experiment.

As we can see in Figs. 7, and Table 3, in *W64P64* we got 0.9894 accuracy, 0.0016 loss, and 0.9878 CK score, while in *mixW64P64* we got 0.9887 accuracy, 0.0017 loss, and 0.986 CK score. In both experiments, we got almost identical results, slightly better in *W64P64*. The learning was excellent even when we used heterogeneous hardware and software. These results show that the interoperability of the platform works remarkably well.

Table 3
Results of case study 1. The results using homogeneous and heterogeneous hardware and software, W64P64 and mixW64P64, are very similar.

Job	Workers	P	Min Loss	Max Acc.	CK score
W64P64	**64py**	**64**	**0.0016**	**0.9894**	**0.9878**
mixW64P64	24js40py	64	0.0017	0.9887	0.9860

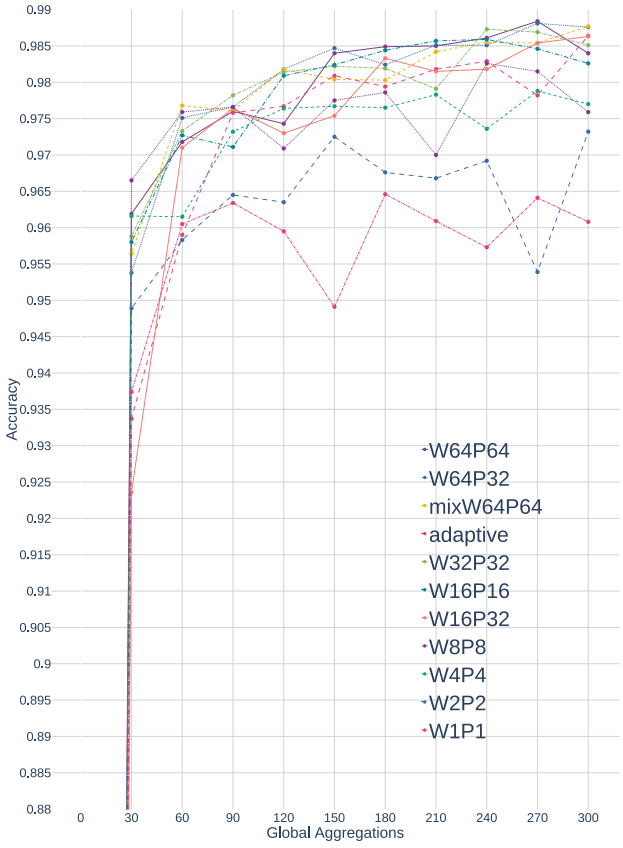


Fig. 7. Accuracy progress in case studies 1 to 3 every 30 aggregations. The larger the number of workers, the better the accuracy.

4.3. Case study 2: asynchronous configuration

In case study 2, we analyze the most suitable configuration to perform asynchronous FEEL. We do not yet employ self-adaptation but study learning using a static and fixed number of workers $|E|$ and the aggregation parameter P in each experiment for comparison. Subsequently, in case 3, we use the results of case study 2 to configure real-time self-adaptation of the algorithm to changes in a dynamic environment.

In this case study, we conduct two groups of related experiments. All of them use up to 64 Python processes.

(a) First, we perform a group of experiments using a given constant number of workers, 64, 32, 16, 8, 4, 2, and 1. In each experiment, the aggregation parameter is constant and is the same as the number of workers used (i.e., $P = |E|$). Both remain constant throughout the learning. We want to analyze what accuracy and CK score we get when

Table 4
Results of case study 2-a. When we repeated the experiment with progressively fewer workers, the accuracy gradually decreased.

Job	Workers	P	Min Loss	Max Acc.	CK score
W64P64	**64py**	**64**	**0.0016**	**0.9894**	**0.9878**
W32P32	32py	32	0.0018	0.9878	0.9842
W16P16	16py	16	0.0019	0.9876	0.9843
W8P8	8py	8	0.0025	0.9844	0.9817
W4P4	4py	4	0.0029	0.9811	0.9770
W2P2	2py	2	0.0043	0.9741	0.9724
W1P1	1py	1	0.0055	0.9663	0.9626

Table 5
Results of case study 2-b. We can obtain high accuracy when the number of workers is different from the aggregation parameter as long as they are not too distant. Otherwise, learning gets worse and worse.

Job	Workers	P	Min Loss	Max Acc.	CK score
W64P32	**64py**	**32**	**0.0017**	**0.9889**	**0.9867**
W32P32	32py	32	0.0018	0.9878	0.9842
W16P32	16py	32	0.0020	0.9865	0.9846
W8P32	8py	32	0.0722	0.6706	0.5116

we use a different number of workers. Each one only has access to a small local dataset. Therefore, we expect to get greater accuracy when using more workers because of having access to more training data.

(b) In the second group of experiments, we use 64, 32, 16, and 8 workers, but we fix the aggregation parameter to 32 in all cases (i.e., $P \neq |E|$). We want to analyze how learning is affected when the number of workers is different from the aggregation parameter. It causes the aggregator to use old results to compute the new shared model. We suppose that when the number of workers and the aggregation parameter are different the final accuracy will be lower.

In Figs. 7, and Table 4, we can see the results of the first group of experiments (2-a). We observe when we repeated the experiment with progressively fewer workers, the accuracy and the CK score gradually decreased as we expected. From 64 (W64P64) to 1 worker (W1P1), we got 0.9894, 0.9878, 0.9876, 0.9844, 0.9811, 0.9741, 0.9663 accuracies and 0.9878, 0.9842, 0.9843, 0.9817, 0.9770, 0.9724, 0.9626 CK scores.

We see that the more workers we use, the better the final accuracy we get. However, we can also see that even if we have a low number of workers, and therefore less dataset available, we can obtain high accuracy. That corroborates our idea that we can continue the learning even if the number of workers varies if we match the aggregation parameter P to the number of workers connected $|E|$.

In Figs. 7, and Table 5 we can see the results of the second group of experiments (2-b). There are some exciting results. When the aggregation parameter P is half or double the number of online workers $|E|$ (*W16P32* and *W64P32*), the accuracy obtained is high (0.9865 and 0.9889, respectively), even *W64P32* is slightly better than when P is equal to $|E|$ (*W32P32*) (0.9878). If we look at the CK score, the *W64P32*, *W32P32*, and *W16P32* obtain very similar results (0.9867, 0.9842, and 0.9846). Using a different value of the aggregation parameter P to the number of workers $|E|$ adds noise to the learning. However, some researchers [46,47] claim that training deep networks by adding some noise allows you to avoid overfitting and even obtain a lower loss. That is what happens in *W64P32* and *W16P32*. The added noise, when it was not too excessive, allowed them to achieve high accuracy and CK score.

However, when P is very distinct from $|E|$ (*W8P32*), we get a lower accuracy and CK score (0.6706 and 0.5116) as we are using too many results from previous old model states to calculate the current weights (i.e., there was too much noise). That leads us to believe that we can obtain high accuracy by adapting the aggregation parameter P to the current online workers $|E|$ in real-time. Also, that allows for new connections and disconnections of workers during training getting a fault-tolerant algorithm.

In conclusion, we showed in case study 2 that we can continue learning despite the number of workers, even when we only have a small percentage of the theoretical global dataset still getting a high accuracy and CK score. Also, we saw that even if the number of workers and the aggregation parameter are not identical, we can still get high accuracy and a high CK. Therefore, using outdated model results to calculate the new shared model may not be detrimental for model performance if the added noise is moderate.

Table 6
Results of case study 3. Using the adaptive algorithm and random drops, we get a similar accuracy to when the system used a given constant number of workers during learning (i.e., W64P64 and W32P32).

Job	Workers	P	Min Loss	Max Acc.	CK score
W64P64	**64py**	**64**	**0.0016**	**0.9894**	**0.9878**
adaptive	dynamic	adaptive	0.0017	0.9884	0.9859
W32P32	32py	32	0.0018	0.9878	0.9842

4.4. Case study 3: self-adaptation and fault-tolerance

In this case study, we analyze the self-adaptation and fault-tolerance of the platform. We analyze learning performance in a changing environment where workers can connect and disconnect at any time for any reason. We use up to 64 Python processes on constrained devices. However, the devices are connected and disconnected stochastically during learning following two exponential distributions [48]. In the beginning, half of the workers are connected. Then, each one uses an exponential distribution to get how long they will be online. We do the same for the disconnected ones using the other exponential distribution. When a worker has to be disconnected, we kill the process. When a worker has to be connected, we create a new process. Therefore, there are connections and disconnections without any prior negotiation. When a worker reconnects, it must reload the current state of the problem to start collaborating. On average, half of them were online, but not always the same ones. The algorithm automatically adapts the aggregation parameter P in real-time to the current number of connected workers $|E|$.

In Figs. 7, and Table 6 we present the result of the case study 3 and we compare it to the *W64P64* and *W32P32*. Table 7 shows the comparison of all the experiments performed. The adaptive algorithm got a maximum accuracy of 0.9884 and a CK score of 0.9859, while the best results obtained in the previous experiments was 0.9894 accuracy and 0.9878 CK score (*W64P64*). It is important to note that the adaptive algorithm does not have access to the entire dataset the whole time since not all workers are online at the same time. Therefore, we expected the accuracy of *W64P64* to be slightly higher than that of the adaptive experiment, although we can see from the results that this difference is minimal. Besides, it is interesting to show that the adaptive experiment achieved better accuracy and CK score than the experiment in which we had 32 fixed workers (*W32P32* 0.9878 accuracy and 0.9842 CK score). That proves the adaptive algorithm was able to adapt perfectly to the changing environment and at the same time take advantage of the extra information (new local dataset) provided by the new workers when they connected. Taking all these constraints into account, the adaptive algorithm achieved excellent results.

Table 7
Comparison of experiments in case studies 1 to 3.

Job	Workers	*P*	Min Loss	Max Acc.	CK score
W64P64	**64py**	**64**	**0.0016**	**0.9894**	**0.9878**
W64P32	64py	32	0.0017	0.9889	0.9867
mixW64P64	24js40py	64	0.0017	0.9887	0.9860
adaptive	dynamic	adaptive	0.0017	0.9884	0.9859
W32P32	32py	32	0.0018	0.9878	0.9842
W16P16	16py	16	0.0019	0.9876	0.9843
W16P32	16py	32	0.0020	0.9865	0.9846
W8P8	8py	8	0.0025	0.9844	0.9817
W4P4	4py	4	0.0029	0.9811	0.9770
W2P2	2py	2	0.0043	0.9741	0.9724
W1P1	1py	1	0.0055	0.9663	0.9626
W8P32	8py	32	0.0722	0.6706	0.5116

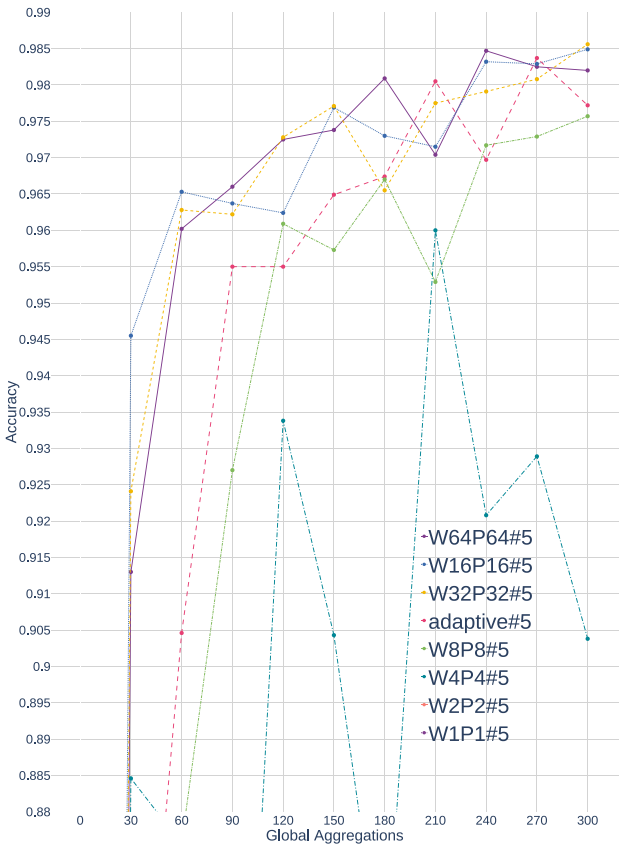


Fig. 8. Accuracy progress in case study 4 every 30 aggregations. The larger the number of workers, the better the accuracy.

4.5. Case study 4: extreme non-i.i.d data

In this case study, we evaluate the self-adaptation and fault-tolerance of the platform using a more extreme *non-i.i.d* data. Also, we perform an analysis of the scalability and random drops. We use an even more truncated data distribution than we would handle in a real-world scenario (see Fig. 6). First, we create a random distribution for the clients. We use a range between 10 and 100 on each client that corresponds to the proportion of data holds on that client. Next, we split data among clients ensuring that every client can hold five classes, worker's datasets do not overlap, and the number of samples is proportional to the random number selected before.[12] In each experiment, we choose workers in order from 0 to 63. For instance, if we use four workers in an experiment, we select workers with IDs from 0 to 3.

Table 8
Results of case study 4. Each worker has a different number of samples of each class and a maximum of five classes. The larger the number of workers, the better the accuracy.

Job	Workers	*P*	Min Loss	Max Acc.	CK score
W64P64#5	**64py**	**64**	**0.0020**	**0.9866**	**0.9833**
W16P16#5	16py	16	0.0022	0.9859	0.9840
W32P32#5	32py	32	0.0023	0.9856	0.9834
adaptive#5	dynamic	adaptive	0.0022	0.9854	0.9822
W8P8#5	8py	8	0.0035	0.9782	0.9728
W4P4#5	4py	4	0.0047	0.9701	0.9618
W2P2#5	2py	2	0.0088	0.9431	0.9228
W1P1#5	1py	1	0.0758	0.4776	0.2001

We perform first a group of experiments using a given constant number of workers, 64, 32, 16, 8, 4, 2, and 1. In each experiment, the aggregation parameter is constant and is the same as the number of workers used (i.e., $P = |E|$). The number of workers and the aggregation parameter remain constant throughout the learning. Next, we perform the adaptive experiment of case study 3 but using the more extreme data *non-i.i.d* data to compare with the previous experiments. Our purpose is to analyze whether the adaptive algorithm can be competitive in the case of an extreme *non-i.i.d* distribution of data and an unpredictable and dynamic scenario compared to experiments using constant workers in a static environment.

Fig. 8 and Table 8 show the accuracy, CK score and loss of the experiments of the case study 4. The experiments that obtained the best results were W64P64#5, W16P16#5, W32P32#5, and adaptive#5. Their accuracies were 0.9866, 0.9859, 0.9856 and 0.9854, their CK scores were 0.9833, 0.9840, 0.9834, 0.9822, and their losses were 0.0020, 0.0022, 0.0023 and 0.0022. In these four experiments, we got similar results. We observe that the adaptive algorithm is competitive in this challenging case and obtains similar accuracy, CK score, and loss to the best of the other results.

On the other side, experiments using few workers obtained low accuracy and CK score. That was due to little or no information on some classes. W8P8#5, W4P4#5, W2P2#5, W1P1#5 got progressively worse results because they had fewer data available. Their accuracies were 0.9782, 0.9701, 0.9431 and 0.4776, their CK scores were 0.9728, 0.9618, 0.9228, 0.2001, and their losses were 0.0035, 0.0047, 0.0088 and 0.0758. These results show that if the data are distributed strongly differently among the nodes, we need more than eight workers to obtain good training results. However, six of the 300 aggregations performed by our adaptive algorithm used eight gradients or less. Despite this, the adaptive algorithm performed well. In future work, we want to study in more depth whether we should stop the adaptive algorithm in these cases, whether these extreme variations are positive for training, or how we can use different aggregations techniques in these cases.

The purpose of our adaptive and asynchronous design is not to improve performance but to adapt to changes in connections and disconnections from initially unknown volunteer devices. However, we also measured the scalability, i.e., the number of processed tasks to the number of connected workers. The results are shown in Fig. 9. We can observe that the scalability is logarithmic and this is due to the fact that in our experiments all the actors were connected through the same 100 Mb Ethernet network. Despite this, the system scales as we add workers up to 64, which we consider a good result given the communication channel constraints [49]. In future work, we plan to perform a

[12] towardsdatascience.com/preserving-data-privacy-in-deep-learning-part-2-6c2e9494398b.

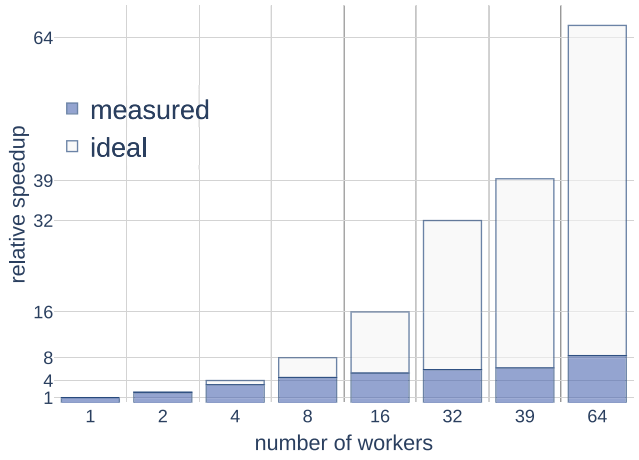


Fig. 9. Case study 4: Relative speedup when devices share the same 100 Mb Ethernet network. The adaptive algorithm with random drops (39.3±11.1) scales proportionally in the same way as the others.

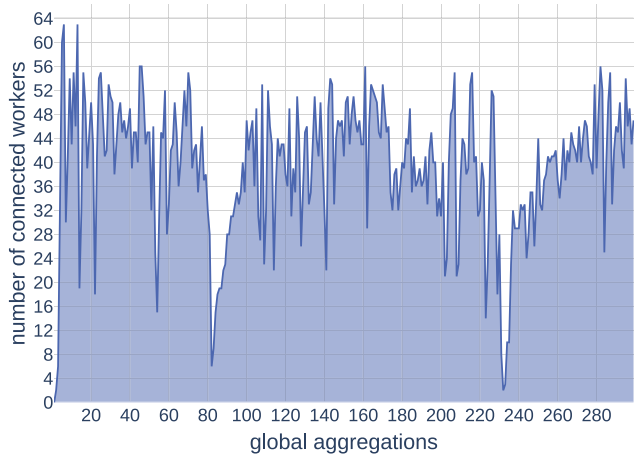


Fig. 10. Case study 4: Evolution of worker's connections and disconnections in the experiment adaptive#5.

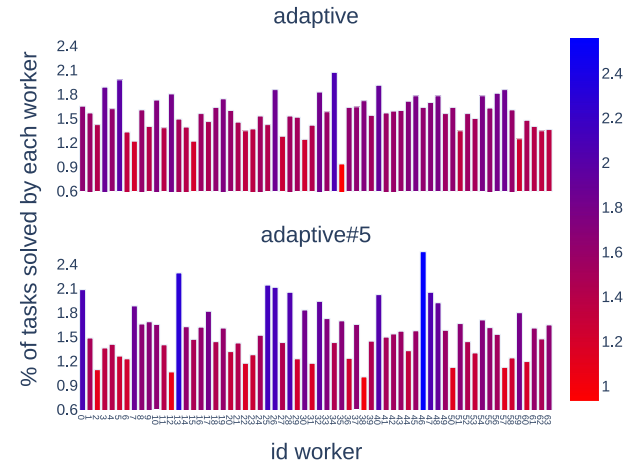


Fig. 11. Percentage of tasks solved by each worker when using the adaptive algorithm. The upper graph is from case study 3, and the lower is from case study 4.

more comprehensive scalability analysis using different communication channels (shared and not shared) and network speeds to get a more accurate result on scalability.

In *adaptive#5*, an average of 39.3 workers were connected concurrently with a standard deviation of 11.1 (see Fig. 10). Furthermore, the scalability when using random drops of workers (i.e., adaptive algorithm) is proportional to the case of using a given constant number of workers (see Fig. 9). Therefore, the impact of random drops of connected workers on the performance of the adaptive algorithm was negligible, which is a great result. In Fig. 11, we see the percentage of tasks solved by each worker in the experiments where we use the adaptive algorithm. We note that random disconnections have caused some workers to end up performing up more than twice as many tasks as others. Despite this, the adaptive algorithm adapted to these changes and got a result similar to the best results using a static scenario.

4.6. Summary of results

The results presented in Section 4.5 show that the implementation of our adaptive proposal is effective for FEEL in very different scenarios even with extreme *non-i.i.d* data, unreliable dis/connecting volunteers and the difficulties mentioned in Section 2.1. The main findings are as follows:

1. Modifying the variable that aggregates the results of workers P according to the available workers $|E|$ together with using a threshold Z for old results computed by workers is effective for achieving a high level of accuracy and CK score in highly dynamic scenarios of FEEL with the participation of a crowd of unreliable devices.
2. The noise generated by the above techniques, if it is not too great, not only does not worsen the resulting accuracy and CK score but can improve them. Experiments show that the most suitable value for P is between $\frac{|E|}{2}$ and $E \cdot 2$. Therefore, it is necessary for the adaptive algorithm to always maintain P within these margins to get a high final accuracy and CK score.
3. When the number of workers is kept too low and unchanged for too long (in this case, less than 8), little or no data available of some classes limits learning. Recall that the local dataset of each worker may be unbalanced and *non-i.i.d*. Therefore, it is necessary to find new techniques to cope with this problem, such as decreasing the learning rate or even temporarily interrupting training if this situation continues for a long time. That does not happen if the available workers shift or the number of workers temporarily decreases and later increases again because then the model learns using different data, obtaining a high level of accuracy and CK score.
4. Connections and disconnections of both new and old volunteers, without the need for pre-configuration, do not compromise the resulting accuracy and CK score if we dynamically adapt P to $|E|$. Nonetheless, further research is still needed to analyze the best trade-off between the parameters used and the best aggregation technique according to the distribution of the data and the available workers.
5. To the authors' knowledge, this is the first work on distributed NN training in which web browsers collaborate with processes executed from a terminal (i.e., Python processes), then proving the interoperability [42,50] of our proposal. We believe web browser computing is the way forward for this type of volunteer computing thanks to its ubiquity, sandboxing, and no need for software installation [20].

5. Threats to validity

This section discusses the main threats to the validity, with special attention to issues that were not reviewed in this study. First, we refer to the communications bottleneck. In the experiments conducted in this work, we spent most of the time on communication rather than processing using a 100 Mb Ethernet. The small communication channel resulted in lower than the desired scalability [49]. Communication cost is often a bottleneck in FL and other distributed computing. Some systems like Horovod [51] use MPI to minimize the IO costs requiring to use of static workers, defining how many workers will participate and their addresses. Yet, in VC, users should be able to join and leave the collaboration at any time without prior configuration.

Some authors have proposed to use process redundancy with minimal impact on performance [27] to deal with unreliable volunteers. However, other authors claimed that these techniques are more suited to different VC scenarios [30]. In FEEL, the training of the distributed NN is slow, we have to reduce communications, and the user data is private. In addition, each worker processes different data, so redundancy in processing is simply not possible, so we think that our proposal of an adaptive algorithm is better suited to deal with this specific VC scenario. Regarding BOINC, its long computation cycle [30] and the software installation need are not the most suitable approaches for volunteer FEEL. Alternatively, Ray [52] is also used in a static computer cluster and does not address the possibility of not knowing which nodes and when will participate in learning. Configuring devices when we do not know which ones or when they will be connected is completely impossible in VC systems. We propose a decoupled and asynchronous system that allows volunteer participating nodes to connect and disconnect during execution without stopping the learning. We believe that these frameworks must implement our contributions in this paper for performing volunteer, adaptive and dynamic FL on constrained devices in the future. On the other side, from a learning point of view, we use different techniques to alleviate communication bottlenecks, like incrementing local steps before communicating. There are also other methods to reduce IO cost that we want to research in future work, such as quantization and sparsification [29,31,32]. We consider that data communications optimization during learning is a topic of great interest and needs specific work. We hope to explore this topic in more depth in the future.

Working with *non-i.i.d.* and unbalanced dataset is another problem. The literature proposes different strategies to meet this concern [15]. In this work, we have shown how despite *non-i.i.d.* data, even in extreme cases, the adaptive algorithm can get similar results to the best results of the static scenario. In future work, we plan to investigate other aggregation strategies depending on the statistical distribution of the data in each worker. Moreover, we face heterogeneous devices with different performances. We use asynchronous training to deal with this problem using a threshold to discard too old results. There are other techniques to address this issue. For example, we can use a different number of local steps depending on the device's performance, or we can use a different learning rate, among other possibilities [15]. We plan to investigate them in future work. We are aware of the difficulties and challenges of performing asynchronous and adaptive FEEL using volunteer devices. However, we firmly believe that this should be the ultimate goal of FL. We think this work is a promising first step for future research in which researchers can easily explore new techniques in a real environment with our tool.

6. Conclusions and future work

In this work, we propose an algorithm for FEEL that adapts to asynchronous clients joining and leaving the computation. We propose, implement and evaluate a new software platform for distributed deep learning on volunteer edge devices. We then evaluate its results on problems relevant to FEEL. We analyzed the main features of the platform *(i)* decentralized computing, *(ii)* asynchronous learning, *(iii)* dynamicity of connected devices, *(iv)* collaboration between heterogeneous devices in hardware and software, *(v)* the ability of the platform and algorithm to self-adapt to changes, *(vi)* and fault-tolerance.

We show that the platform can adapt to changes and continue learning in a changing environment where volunteer devices connect and disconnect at any time. We test the system using different data distributions. Also, the platform enables interoperability and obtains high accuracy when using heterogeneous devices in hardware and software with different NN training libraries collaborating (i.e., web browsers using TensorFlow.js and Python processes using TensorFlow). Next, we release a modular open-source library publicly available in a Git repository that enables FEEL in a decentralized, asynchronous, and fault-tolerant way. There is no need for the previous configuration of participating workers. Instead, the platform allows workers to join and leave at any time. That opens up an exciting avenue for research allowing researchers to experiment with new FEEL techniques on real edge devices and real users easily. In future work, we plan to research communication reduction techniques such as quantification and sparsification. Also, analyze how to improve learning when using *non-i.i.d.* datasets, and study new aggregation techniques, among others.

CRediT authorship contribution statement

José Ángel Morell: Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Project administration, Software, Validation, Visualization, Writing – original draft. **Enrique Alba:** Conceptualization, Investigation, Funding acquisition, Resources, Supervision, Validation, Writing – review & editing.

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.

Acknowledgments

This research is partially funded by the Universidad de Málaga, Spain, Consejería de Economía *y* Conocimiento de la Junta de Andalucía, Spain and FEDER, Spain under grant number UMA18-FEDERJA-003 (PRECOG); under grant PID 2020-116727RB-I00 (HUmove) funded by MCIN/AEI/10.13039/501100011-033, Spain; and TAILOR ICT-48 Network (No 952215) funded by EU Horizon 2020 research and innovation programme. José Ángel Morell is supported by an FPU grant from the Ministerio de Educación, Cultura y Deporte, Gobierno de España, Spain (FPU16/02595). Funding for open access charge is supported by the Universidad de Málaga/CBUA, Spain. The views expressed are purely those of the writer and may not in any circumstances be regarded as stating an official position of the European Commission.

References

[1] T. Hastie, R. Tibshirani, J. Friedman, The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Springer Science & Business Media, 2009.

[2] S. Shalev-Shwartz, S. Ben-David, Understanding Machine Learning: From Theory to Algorithms, Cambridge University Press, 2014.

[3] W. Shi, J. Cao, Q. Zhang, Y. Li, L. Xu, Edge computing: Vision and challenges, IEEE Internet Things J. 3 (2016) 637–646.

[4] M.J. Sheller, B. Edwards, G.A. Reina, J. Martin, S. Pati, A. Kotrotsou, M. Milchenko, W. Xu, D. Marcus, R.R. Colen, et al., Federated learning in medicine: facilitating multi-institutional collaborations without sharing patient data, Sci. Rep. 10 (2020) 1–12.

[5] J. Chen, X. Ran, Deep learning with edge computing: A review, Proc. IEEE 107 (2019) 1655–1674.

[6] W.Y.B. Lim, N.C. Luong, D.T. Hoang, Y. Jiao, Y.C. Liang, Q. Yang, D. Niyato, C. Miao, Federated learning in mobile edge networks: A comprehensive survey, IEEE Commun. Surv. Tutor. 22 (2020) 2031–2063.

[7] A. Hard, K. Rao, R. Mathews, S. Ramaswamy, F. Beaufays, S. Augenstein, H. Eichner, C. Kiddon, D. Ramage, Federated learning for mobile keyboard prediction, 2018, arXiv preprint arXiv:1811.03604.

[8] J. Posner, L. Tseng, M. Aloqaily, Y. Jararweh, Federated learning in vehicular networks: Opportunities and solutions, IEEE Netw. (2021).

[9] Y. Liu, J. Nie, X. Li, S.H. Ahmed, W.Y.B. Lim, C. Miao, Federated learning in the sky: Aerial-ground air quality sensing framework with uav swarms, IEEE Internet Things J. (2020).

[10] Y. Ye, S. Li, F. Liu, Y. Tang, W. Hu, Edgefed: optimized federated learning based on edge computing, IEEE Access 8 (2020) 209191–209198.

[11] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konecny, S. Mazzocchi, H.B. McMahan, et al., Towards federated learning at scale: System design, 2019, arXiv preprint arXiv:1902.01046.

[12] B. McMahan, E. Moore, D. Ramage, S. Hampson, B.A. y Arcas, Communication-efficient learning of deep networks from decentralized data, in: Artificial Intelligence and Statistics, PMLR, 2017, pp. 1273–1282.

[13] S. Wang, T. Tuor, T. Salonidis, K.K. Leung, C. Makaya, T. He, K. Chan, Adaptive federated learning in resource constrained edge computing systems, IEEE J. Sel. Areas Commun. 37 (2019) 1205–1221.

[14] A. Tak, S. Cherkaoui, Federated edge learning: design issues and challenges, IEEE Netw. (2020).

[15] H.B. McMahan, et al., Advances and open problems in federated learning, Found. Trends® Mach. Learn. 14 (2021).

[16] D.P. Anderson, Boinc: A platform for volunteer computing, J. Grid Comput. 18 (2020) 99–122.

[17] R. Das, B. Qian, S. Raman, R. Vernon, J. Thompson, P. Bradley, S. Khare, M.D. Tyka, D. Bhat, D. Chivian, et al., Structure prediction for casp7 targets using extensive all-atom refinement with rosetta@ home, Proteins Struct. Funct. Bioinform. 69 (2007) 118–128.

[18] IBM, World community grid, 2019, https://www.worldcommunitygrid.org/ Online; (Accessed 28 May 2019).

[19] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, M. Lebofsky, Seti@ home—massively distributed computing for seti, Comput. Sci. Eng. 3 (78) (2001).

[20] J.Á. Morell, A. Camero, E. Alba, Jsdoop and tensorflow. js: Volunteer distributed web browser-based neural network training, IEEE Access 7 (2019) 158671–158684.

[21] Y. Chen, Y. Ning, M. Slawski, H. Rangwala, Asynchronous online federated learning for edge devices with non-iid data, in: 2020 IEEE International Conference on Big Data, Big Data, IEEE, 2020, pp. 15–24.

[22] Y. Chen, X. Sun, Y. Jin, Communication-efficient federated deep learning with layerwise asynchronous model update and temporally weighted aggregation, IEEE Trans. Neural Netw. Learn. Syst. 31 (2019) 4229–4238.

[23] M. Noura, M. Atiquzzaman, M. Gaedke, Interoperability in internet of things: Taxonomies and open challenges, Mob. Netw. Appl. 24 (2019) 796–809.

[24] D. Thain, T. Tannenbaum, M. Livny, Distributed computing in practice: the condor experience, Concurr. Comput.: Pract. Exper. 17 (2005) 323–356.

[25] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT Press, 2016.

[26] M.N. Durrani, J.A. Shamsi, Volunteer computing: requirements, challenges, and solutions, J. Netw. Comput. Appl. 39 (2014) 369–380.

[27] J. Subhlok, H. Nguyen, E. Gabriel, M.T. Rahman, Resilient parallel computing on volunteer PC grids, Concurr. Comput.: Pract. Exper. 30 (2018) e4478.

[28] H.B. Konečỳ J. McMahan, F.X. Yu, A.T. Richtárik, D. Bacon, Federated learning: Strategies for improving communication efficiency, 2016, arXiv preprint arXiv:1610.05492.

[29] R. Mayer, H.A. Jacobsen, Scalable deep learning on distributed infrastructures: Challenges, techniques, and tools, ACM Comput. Surv. 53 (2020) 1–37.

[30] Z. Lv, D. Chen, A.K. Singh, Big data processing on volunteer computing, ACM Trans. Internet Technol. 21 (2021) 1–20.

[31] T. Ben-Nun, T. Hoefler, Demystifying parallel and distributed deep learning: An in-depth concurrency analysis, ACM Comput. Surv. 52 (2019) 1–43.

[32] J. Xu, W. Du, Y. Jin, W. He, R. Cheng, Ternary compression for communication-efficient federated learning, IEEE Trans. Neural Netw. Learn. Syst. (2020).

[33] R. Zhang, J. Kwok, Asynchronous distributed admm for consensus optimization, in: International Conference on Machine Learning, 2014, pp. 1701–1709.

[34] Q. Meng, W. Chen, Y. Wang, Z.M. Ma, T.Y. Liu, Convergence analysis of distributed stochastic gradient descent with shuffling, Neurocomputing 337 (2019) 46–57.

[35] O. Shamir, Without-replacement sampling for stochastic gradient methods, in: Advances in neural information processing systems, 2016, pp. 46–54.

[36] Y. Lu, X. Huang, K. Zhang, S. Maharjan, Y. Zhang, Blockchain empowered asynchronous federated learning for secure data sharing in internet of vehicles, IEEE Trans. Veh. Technol. 69 (2020) 4298–4311.

[37] Z. Chen, Z. Liu, K.L. Ng, H. Yu, Y. Liu, Q. Yang, A gamified research tool for incentive mechanism design in federated learning, in: Federated Learning, Springer, 2020, pp. 168–175.

[38] V. Mothukuri, R.M. Parizi, S. Pouriyeh, Y. Huang, A. Dehghantanha, G. Srivastava, A survey on security and privacy of federated learning, Future Gener. Comput. Syst. 115 (2021) 619–640.

[39] S. Ruder, An overview of gradient descent optimization algorithms, 2016, arXiv preprint arXiv:1609.04747.

[40] K. Janocha, W.M. Czarnecki, On loss functions for deep neural networks in classification, 2017, arXiv preprint arXiv:1702.05659.

[41] K. Wolter, Stochastic models for fault tolerance: Restart, in: Rejuvenation and Checkpointing, Springer Science & Business Media, 2010.

[42] T. Mikkonen, C. Pautasso, A. Taivalsaari, Isomorphic Internet of Things architectures with web technologies, Computer 54 (2021) 69–78.

[43] W. Rawat, Z. Wang, Deep convolutional neural networks for image classification: A comprehensive review, Neural Comput. 29 (2017) 2352–2449.

[44] Y. LeCun, The mnist database of handwritten digits, 1998, http://yann.lecun.com/exdb/mnist/.

[45] J. Sim, C.C. Wright, The kappa statistic in reliability studies: use, interpretation, and sample size requirements, Phys. Ther. 85 (2005) 257–268.

[46] S. Chaturapruek, J.C. Duchi, C. Ré, Asynchronous stochastic convex optimization: the noise is in the noise and sgd don't care, Adv. Neural Inf. Process. Syst. 28 (2015) 1531–1539.

[47] A. Neelakantan, L. Vilnis, Q.V. Le, I. Sutskever, L. Kaiser, K. Kurach, J. Martens, Adding gradient noise improves learning for very deep networks, 2015, arXiv preprint arXiv:1511.06807.

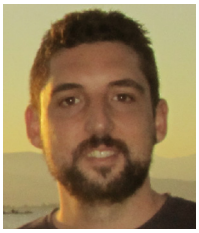
[48] A.M. Law, W.D. Kelton, W.D. Kelton, Simulation Modeling and Analysis, Vol. 3, McGraw-Hill New York, 2000.

[49] Z. Zhang, C. Chang, H. Lin, Y. Wang, R. Arora, X. Jin, Is network the bottleneck of distributed training? in: Proceedings of the Workshop on Network Meets AI & ML, 2020, pp. 8–13.

[50] Y. Liu, C. Chen, R. Zhang, T. Qin, X. Ji, H. Lin, M. Yang, Enhancing the interoperability between deep learning frameworks by model conversion, in: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020, pp. 1320–1330.

[51] A. Sergeev, M. Del Balso, Horovod: fast and easy distributed deep learning in tensorflow, 2018, arXiv preprint arXiv:1802.05799.

[52] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M.I. Jordan, et al., Ray: A distributed framework for emerging *AI* applications, in: 13th *USENIX* Symposium on Operating Systems Design and Implementation *OSDI* 18, 2018, pp. 561–577.



José Ángel Morell received the B.Eng. degree (Hons.) in computer science from the Universidad Nacional de Educación a Distancia (UNED), in 2016, and the M.Sc. degree (Hons.) in software engineering and artificial intelligence from the Universidad de Málaga, Spain, in 2017, where he is currently pursuing the Ph.D. degree. His research interests include design of optimization and machine learning distributed algorithms adapted to edge computing.

Prof. Enrique Alba had his degree in engineering and Ph.D. in Computer Science in 1992 and 1999, respectively, by the University of Málaga (Spain). He works as a Full Professor in this university with varied teaching duties: data communications, distributed programming, software quality, and also evolutionary algorithms, bases for R+D+i and smart cities, both at graduate and master/doctoral programs. Prof. Alba leads an international team of researchers in the field of complex optimization/learning with applications in smart cities, bioinformatics, software engineering, telecoms, and others. In addition to the organization of international events (ACM GECCO, IEEE IPDPS-NIDISC, IEEE MSWiM, IEEE DS-RT, smart-CT…) Prof. Alba has offered dozens postgraduate courses, more than 70 seminars in international institutions, and has directed many research projects (9 with national funds, 7 in Europe, and numerous bilateral actions). Also, Prof. Alba has directed 12 projects for innovation in companies (OPTIMI, Tartessos, ACERINOX, ARELANCE, TUO, INDRA, AOP, VATIA, EMERGIA, SECMOTIC, ArcelorMittal, ACTECO, CETEM, EUROSOTERRADOS) and has worked as invited professor at INRIA, Luxembourg, Ostrava, Japan, Argentina, Cuba, Uruguay, and Mexico. He is editor in several international journals and book series of Springer-Verlag and Wiley, as well as he often reviews articles for more than 30 impact journals. He is included in the list of most prolific DBLP authors, and has published 130 articles in journals indexed by ISI, 11 books, and hundreds of communications to scientific conferences. He is included in the top ten most relevant researchers in Informatics in Spain (fifth position in ISI), and is the most influent researcher of UMA in engineering (webometrics), with 14 awards to his professional activities. Pr. Alba's H index is 65, with more than 19,000 cites to his work.