FLSim: An Extensible and Reusable Simulation Framework for Federated Learning

Li Li[1(✉)], Jun Wang[2], and ChengZhong Xu[3]

[1] ShenZhen Institutes of Advanced Technology, Chinese Academy of Sciences, Beijing, China
li.li@siat.ac.cn
[2] Futurewei Technologies, Santa Clara, US
[3] State Key Laboratory of IoTSC, University of Macau, Zhuhai, China

Abstract. Federated learning is designed for multiple mobile devices to collaboratively train an artificial intelligence model while preserving data privacy. Instead of collecting the raw training data from mobile devices to the central server, federated learning coordinates a group of devices to train a shared model in a distributed manner with their local data. However, prior to effectively deploying federated learning on resource-constrained mobile devices in large scale, different factors including the convergence rate, energy efficiency and model accuracy should be well studied. Thus, a flexible simulation framework that can be used to investigate a wide range of problems related to federated learning is urgently required.

In this paper, we propose FLSim, a framework for efficiently building simulators for federated learning. Unlike ad hoc simulators, FLSim is envisioned as an open repository of building blocks for creating simulators. To this end, FLSim consists of a set of software components organized in a well-structured software architecture that provides the foundation for maximizing flexibility and extensibility. With FLSim, creating a simulator generally involves only putting the selected components together, thus allowing users to focus on the problems being studied. We describe the design of the framework in detail and use a few use cases to demonstrate the ease with which various simulators can be constructed with FLSim.

1 Introduction

Mobile devices (e.g., smartphone and wearable devices), powered by batteries, intimately connect users and their environment. Equipped with various types of sensors (e.g., GPS, accelerometer, gyroscope), mobile devices can collect different kinds of data, ubiquitously [19,21]. These data are valuable resources for intelligent applications to efficiently understand user behavior and significantly improve user experience from different perspectives. Acquiring these data raises

H. Song and D. Jiang (Eds.): SIMUtools 2020, LNICST 369, pp. 350–369, 2021.
https://doi.org/10.1007/978-3-030-72792-5_30

a big question mark on preserving users' privacy with any service provided [4]. For example, the scandal swirling around the Facebook App and Cambridge Anaytica has begun to usher in a new era for this once-ignored community of privacy researchers and developers [14,23]. Thus, intelligently analyzing the data being generated from mobile devices while preserving data privacy is a critical challenge.

Federated learning [22] is proposed in order to effectively train the data ubiquitously from mobile devices while removing the concern of privacy. Unlike the cloud-based approach which collects the local training data in the data center, federated learning collaboratively trains a shared model with the data located on each mobile device [16]. Specifically, in a training round, each participating device computes the updates to the current global model based on its local training data. These updates are then sent to the central server. After receiving the updates, the central server aggregates them, updates the shared model and broadcasts the updated model to the mobile participants. This process iterates until the model converges. In this approach, the predictive model can be collaboratively learned while the data privacy is well preserved. This is because the privacy sensitive raw data never leaves the mobile device during the whole training process. Currently, federated learning has been adopted to support various kinds of applications such as human activity recognition, on-device item ranking, and next-word prediction [20].

Despite all the promising benefits, effectively deploying federated learning on mobile devices in large scale is challenging. The whole system can be severely impacted by different factors in a highly dynamic environment. For instance, the training data generated from different mobile devices can have highly different distributions due to different user interaction behaviors which can impact the model convergence in totally different ways. On the other side, the participating devices in a federated learning system usually possess different hardware configurations which lead to totally different training capability. Thus, the training progress of each device can be highly unbalanced which severely impacts the overall training progress of the whole system. Moreover, some uncertainty issues such as out of battery and poor network connection can further affect the convergence rate of a federated learning system in practice. In addition, energy consumption of the on-device learning process is another critical concern that determines whether a specific user is willing to participate in the training process. Prior to deploying a federated learning framework on real mobile devices in large scale, assessing the impacts of different factors is critical for developers and researchers to make corresponding strategies for efficient and effective deployment. ***Thus, a framework that can efficiently create simulators to simulate different usage scenarios in federated learning and quickly obtain corresponding information (e.g., model accuracy, model convergence rate and energy consumption) is urgently required.***

In this paper, we propose FLSim, an extensible and reusable simulation framework for federated learning. FLSim adopts a layered software architecture. It supports commonly used deep learning frameworks such as PyTorch

and Tensorflow. Thus developers familiar with any one of these frameworks can conveniently use FLSim to create corresponding simulators. FLSim can be easily ported to run on different hardware platforms. Moreover, FLSim adopts a highly modular design, allowing different simulators to be easily created through the integration of different components in FLSim. We use different case studies to evaluate the effectiveness of FLSim. The results show that FLSim can effectively simulate different scenarios in Federated Learning and obtain corresponding information accurately. To the best of our knowledge, FLSim is the *first* work that provides a flexible simulation framework that can be efficiently used to study a wide range of problems related to federated learning. Specifically, our major contributions are as follows:

- We propose FLSim, a simulation framework for federated learning, which intelligently helps developers create different simulators to simulate different scenarios in an efficient way.
- FLSim adopts a layered software architecture. Moreover, techniques such as inversion of control are used to make FLSim flexible and extendable.
- We implement the prototype of FLSim and use different use cases to demonstrate the effectiveness of FLSim.

The rest of the paper is organized as follows. Section 2 introduces the background about federated learning and the previous research closely related to our work. Section 3 introduces the system design and the implementation of different components. Section 4 discusses different case studies we adopt to evaluate the effectiveness of FLSim. Section 5 briefly evaluates the performance of FLSim. Finally, Sect. 6 concludes the paper.

2 Background and Related Work

In this section, we introduce the background of federated learning and previous work that is closely related with this paper.

2.1 Background About Federated Learning

Federated learning (FL) is proposed to effectively train the data being generated on mobile devices while protecting data privacy. A typical FL system mainly consists of a central server and multiple mobile devices with heterogeneous hardware configurations. The typical workflow of a federated learning procedure contains the following main steps:

1. **Initialization.** At the initialization step of each training round, the central server selects a set of mobile devices to participate in the training process.
2. **Model Download.** The selected participants download the current shared model state (e.g., current model parameters (w_t)).
3. **On-device Training.** Each mobile device conducts local training based on the shared model state and its local training dataset for a certain number of training epochs.

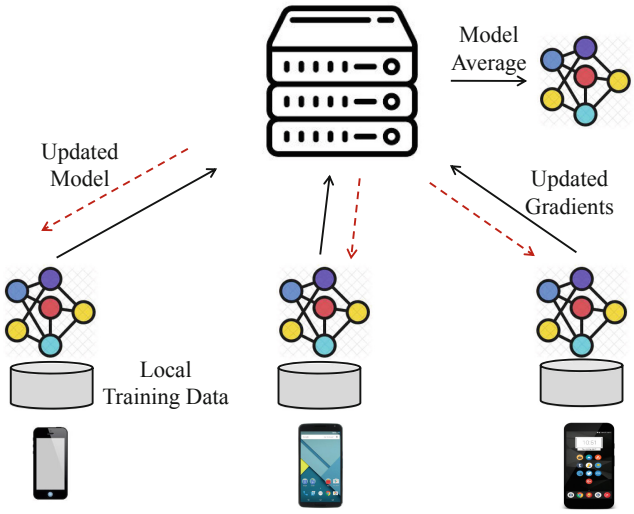


Fig. 1. Workflow of a typical federated learning system.

4. **Gradient Update.** After completing the local training process, each mobile device sends the updated gradients back to the central server.
5. **Model Fusion.** After receiving the updated gradients from all selected participants, the central server aggregates these gradient updates and comes up with the updated global model. Then, the system enters the next training round.
6. **Training Iterates.** The whole process iterates until the global model converges.

We can note that, the raw training data (e.g., raw data generated during user interaction with mobile devices) never leave the mobile device during the whole training process. Data privacy is therefore well preserved which is the key advantage of Federated Learning.

2.2 Related Work

Our work is closely related to the following research topics.

Federated Learning. Federated learning has raised a lot of attention as a machine learning paradigm that effectively trains the data being generated from mobile devices while guaranteeing the data privacy [7,8,11,15,17,22,24,25,28]. Sprague et al. [25] present a new asynchronous federated learning algorithm and study its convergence rate when distributed across many edge devices, with hard data constraints, relative to training the same model on a single device. Wang et al. [28] analyze the convergence bound of distributed gradient descent from a theoretical point of view and propose a control algorithm that determines the

trade-off between local update and global parameter aggregation to minimize the loss function under a given resource budget. Bonawitz et al. [7] build a scalable production system for federated learning in the domain of mobile devices, based on Tensorflow. Additionally, Bonawitz et al. [8] design a novel, communication-efficient, failure-robust protocol for secure aggregation of high-dimensional data. Previous work on federated learning mainly focuses on reducing the communication overhead during the training process and the model convergence from the theoretical perspective. In this paper, we build FLSim which tries to provide a flexible simulation framework that can be used to study a wide range of problems related to federated learning to help efficient deployment in practice.

Simulation System Construction. From the system point of view, the most relevant concept to this work is simulation system construction. Previous work has explored system design principles along the same line that has guided the creation of FLSim. An early work is SimJava [13], which is a library of Java components for building general discrete-event simulation systems. CloudSim [10] is a simulation framework created for simulating various hardware components and software management services in the cloud computing settingss. Like FLSim its main design focus is to provide a toolkit that is highly extensible and customizable. Not surprisingly, it also adopts a layered software architecture. Wang et al. [27] discuss component-based design followed in creating the Manifold simulation framework [26] for multicore computer architectures. They particularly emphasize the importance of dependency management. Although targeting a different application domain, its design principles are similar to what we have adopted for FLSim.

3 FLSim System Design and Implementation

In designing FLSim, our goal is to provide a flexible simulation framework that can be used to study a wide range of problems related to federated learning, instead of an ad hoc simulator that limits itself to a single or a small number of use cases. This idea is captured in our fundamental design rule.

Fundamental Design Rule. FLSim is a simulation framework that provides building blocks for users to easily create simulators tailored for their own federated learning problems.

Another import question we face is whether the training of the neural networks should be simulated or literally carried out. It is obvious that any federated learning simulator will spend most of its time performing the training. If the training can be simulated, it will greatly reduce the simulation time. However, we suspect that in most cases the user would want for the neural networks to be actually trained. Therefore, in the current version of FLSim, training is literally performed.

With the Fundamental Design Rule in mind, we set the following design goals for FLSim.

- FLSim should support all the commonly used deep learning frameworks such as PyTorch [2] and TensorFlow [3], so that users familiar with any one of those frameworks should be able to use FLSim at the same level of ease.
- FLSim should, under the control of one or a few configuration parameters, be able to run on different underlying hardware platforms.
- FLSim should be highly modular such that different simulators can be created simply by choosing different components. On the other hand, if the system does not include a component that fully meets the user's needs, a new one can be easily created and incorporated into the system.

The following sections give a detailed description of the system design of FLSim.

3.1 Design Overview

In order to achieve the design goals outlined above, we adopt a layered software architecture [9]. This type of architectures allows complex systems to be decomposed into different levels of abstraction with cleanly defined interfaces and dependence relationships. To be more precise, our design is a relaxed layered system where components in a given layer can use the services from any layer below it, not just the one immediately below. However, no layer is allowed to use any functionality from any layer above it. Figure 2 shows the software architecture of FLSim.

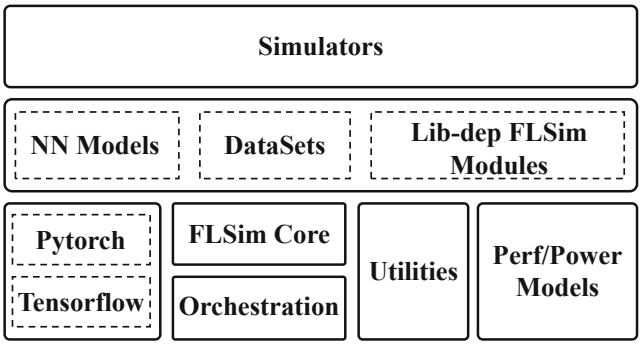


Fig. 2. System architecture.

From the user's standpoint, building a simulator is a simple two-step process. First, the user selects the essential components such as the neural network model, the dataset, and the hardware platform. The user can also create customized components by sub-classing existing classes to meet their requirements. For example, if one needs to select clients in each training round in a particular way, they can simply extend the `FLClientSelector` class.

Once the user has configured the system, in the second step the user instantiates a `Federation` object and starts running the simulator.

In order to realize the clean dependencies as required by the layered software architecture, we adopt a few design guidelines that center on dependency management.

First of all, we adopt the design technique known as inversion of control [12] whenever possible. When an object A directly uses an object B, A has a dependence on B. The coupling between A and B would become too tight if B is created by A. With inversion of control, the dependence of A on B is injected. That is to say, B is created elsewhere and passed to A, while A only requires an interface that B implements.

Another guideline is library-neutrality. Deep learning libraries such as PyTorch and TensorFlow are commonly used. A pitfall we want to avoid is tying the system to any particular library. Therefore, the components in general are designed in a library-neutral manner, with library-dependent extensions provided as a convenience. We next explain the layers individually.

3.2 Library-Independent Layer

At the bottom of the layered architecture is the Library Independent Layer. As the name indicates, components in this layer are independent of deep learning libraries such as PyTorch. This layer contains FLSim Core, Simulation Orchestration, common utilities, and, for studying system performance and energy efficiency, Performance and Power Models. We further divide FLSim Core and Orchestration into two layers, as the latter represents a distinctive layer of abstraction.

Figure 2 also shows at the bottom-left of the layers some commonly used deep learning libraries such as PyTorch and TensorFlow. It is clear from the positions of these components that the FLSim components in this layer are independent of them.

Simulation Orchestration Layer. In the context of FLSim, orchestration means selecting the hardware platform on which to run the simulator, and, when multiprocessing is used, determining the number of processes and how the clients are assigned to the processes.

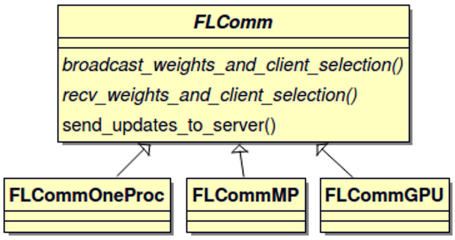


Fig. 3. Orchestration layer

At the core of the Orchestration Layer is the class hierarchy for client-server communications, as shown in Fig. 3. At present, we support using a single CPU process, multiple CPU processes, and GPU SIMT execution, which are respectively implemented in the sub-classes shown in Fig. 3.

These communication classes provide the abstraction for orchestration and insulate the users of these classes from such details as the underlying inter-process communication mechanism, or communication between CPU and GPU. Once a platform is selected, the clients and server simply use the interface functions such as those listed in Fig. 3 to communicate.

FLSim Core. This component includes the basic classes that implement a Federated Learning system, such as server and client. Major classes of this component are illustrated in Fig. 4.

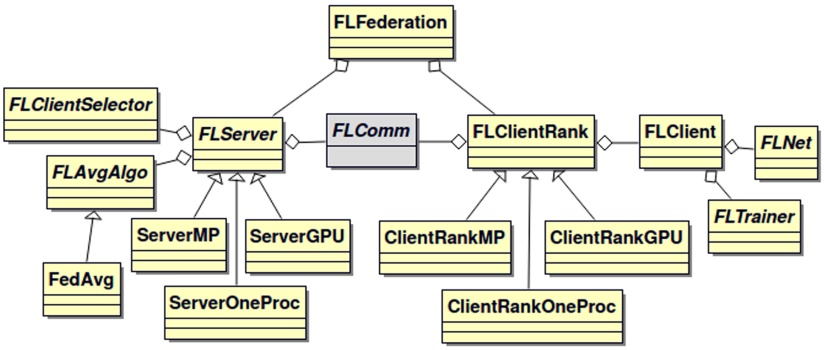


Fig. 4. FLSim Core class diagram.

Clearly the classes form two clusters centered on `FLServer` and `FLClientRank`, the latter representing a group of clients. The server has a few injected dependencies: `FLComm` for communications with the clients, `FLClientSelector` for client selection, and `FLAvgAlgo` for model averaging.

On the client side, injected dependencies include `FLNet` representing a neural network model, `FLTrainer` representing a local trainer that trains the model.

Finally, the class `FLFederation` represents the federation and wraps a server and one or more client ranks.

Utilities. The Utilities component contains utility functions that are independent of deep learning libraries. These include the following: dataset splitter, logging, visualization, and charting.

Performance and Power Models. A special feature of FLSim is its incorporation of performance and power models to support the study of system performance and energy efficiency. At present this is mainly provided in the form of

device profiles represented by the `DevProfile` class. This class basically has a performance table and a power table. The former is an array of execution time required to finish one round of training using different CPU frequencies, and the latter is an array of the same size with the corresponding power consumption values.

3.3 Library-Dependent Layer

The library-dependent layer contains modules that rely on specific deep learning libraries such as PyTorch. At present this layer focuses on PyTorch.

Datasets. Strictly speaking, datasets should be independent of the deep learning libraries. However, libraries like PyTorch provides utilities for some common datasets, making it easier to deal with the datasets. Obviously we can also include library-neutral datasets in the library independent layer.

Neural Network Models. These are generally bound to a particular library. FLSim includes a few models from simple multi-layer perceptrons to more complex CNN and DNN models. Some of the models will be discussed in the case studies.

Library-Dependent FLSim Modules. As mentioned, to maintain the clean layered architecture, some functionalities are split into a library independent part and a library dependent part. The former is generally in the form of base classes, while the latter leverages the support of commonly used libraries.

An example is the `FLTrainer` class. This base class includes an interface for carrying out training of a neural network. The `PytorchTrainer` class extends the base class and is used to train PyTorch neural network models.

3.4 Simulators

At the top of the layered architecture we find simulators which are built with the components from the lower layers. Some of the FLSim simulators are discussed in the case study section. However, it is important to stress that the goal of FLSim is to provide well-structured components for constructing simulators efficiently.

3.5 Steps for Constructing a Simulator

Using the components provided by FLSim, we can construct a generic simulator for federated learning with the following steps:

- Step 1. Create a dataset partitioner and partition the dataset among the given number of clients.
- Step 2. Create a neural network model.

- Step 3. Create a local trainer object.
- Step 4. Create a client selector object.
- Step 5. Create an averaging algorithm object.
- Step 6. Create an `FLFederation` object with the following parameters:
  - The neural network model
  - The local trainer
  - The dataset and the partitioning
  - The client selector
  - The averaging algorithm object.
  - The platform. If CPU is selected, the number of processes as well.
- Step 7. In a loop call the `run_one_round()` method of the `FLFederation` object.

4 Case Studies

In this section we present a few use cases to demonstrate different types of federated learning simulation that can be realized with the FLSim framework. A brief description of the use cases is given below, followed by more details.

- **Basic case**. In this test case we try to repeat some of the experiments reported in [22] to prove the basic capabilities of FLSim.
- **Federated learning with non-IID data**. In this test case we try to repeat some of the experiments presented in [30], which focuses on the effects of non-IID data.
- **Real-time federated learning**: In this test case we build a simulator that is a simplified version of the work presented in [18] to study problems in real-time federated learning.

4.1 Basic Test Case

We start with a basic test case in which we try to duplicate some experiments reported in [22] in order to perform some basic validation of the implementation of FLSim. We use FLSim to build a neural network that is referred to as *2NN* in [22] for the task of classifying hand-written numerical digits from the MNIST dataset [29], which includes a training set of 60,000 images and a test set of 10,000.

The *2NN* model is a multilayer perceptron (MLP) [6] with 1 input layer, 2 hidden layers and 1 output layer, the sizes of which are respectively 768, 200, 200, and 10, for a total of 199,210 parameters.

Components required to create the simulator include theose listed in Table 1.

Note that all those components are readily available in FLSim. Therefore, building the simulator is a straightforward process.

In the first experiment, we modify a couple of training parameters and try to find the least number of training rounds required to achieve a test accuracy of 97.0%, and we compare the results with what are reported in [22]. Specifically, we fix the number of local training epochs to 20, and set the batch size to three

Table 1. FLSim components for the basic test case.

• The 2NN model based on PyTorch
• The MNIST dataset
• The default federation `FLFederation`
• The `FLRandomClientSelector` to randomly select 10 clients out of 100 in each training round
• The `PytorchLocalTrainer`, a PyTorch based local training module

Table 2. 2NN MNIST with epochs fixed at 20: best of 10 runs.

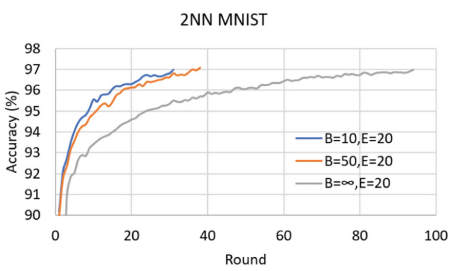
Epoch	BatchSize	#Batches	LearnRate	Rounds	Rounds [22]
20	10	60	0.1	31	32
20	50	12	0.3	38	39
20	50	12	0.1	88	
20	∞	1	0.95	94	92
20	∞	1	0.1	409	

different values (i.e., 10, 50, and all), and select different learning rates to see if we can obtain results similar to [22]. The test results are listed in Table 2.

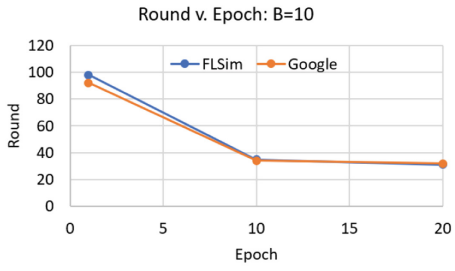
The infinity symbol (∞) means including all the local training data in the batch, i.e., there is only one batch for each client. For the three batch sizes, the best learning rate are respectively 0.1, 0.3, and 0.95. The last column contains the results from [22]. Note that [22] did not state the learning rate used in obtaining the results. We can see that in all three cases our results are very close to those of [22], confirming the soundness of FLSim implementation. Additionally, we have found that the learning rate has a great impact on the number of training rounds required, and it is sensitive to the batch size. For example, while 0.1 is a good learning rate for batch size 10, it produces bad results when the batch size is much larger. Finally, Fig. 5a shows the evolution of the model accuracy with three different batch sizes.

Table 3. 2NN MNIST with fixed batch sizes: best of 10 runs.

Epoch	BatchSize	#Batches	LearnRate	Rounds	Rounds [22]
1	10	60	0.1	98	92
5	10	60	0.1	39	–
10	10	60	0.1	35	34
20	10	60	0.1	31	32
30	10	60	0.1	31	–
1	50	12	0.3	131	144
10	50	12	0.3	40	45
20	50	12	0.3	38	39



(a) (a) Epoch=20; different batch (B) sizes



(b) (b) Round v. epoch

Fig. 5. 2NN results: (a) Epoch (E) set to 20, numbers of rounds with different batch sizes (B). (b) Comparison with [22]: batch size set to 10, epoch set to 1, 10, and 20.

We next try to find the best numbers of rounds with different parameters and compare the results with [22]. Table 3 shows the results of fixing the batch size and changing the epoch. The last column contains results from [22]. It can be seen that in most cases the results are very close.

Table 4 shows the results of the best numbers of rounds obtained with different numbers of clients. In all the four cases here, our results differ from those in [22] by about 10 rounds. One possible reason for this difference is the learning rate. Unlike [22], we did not test a large set of different learning rates. Note that for the case E=1, B=10, two different results are reported in [22], namely 92 and 87, as listed in Tables 3 and 4.

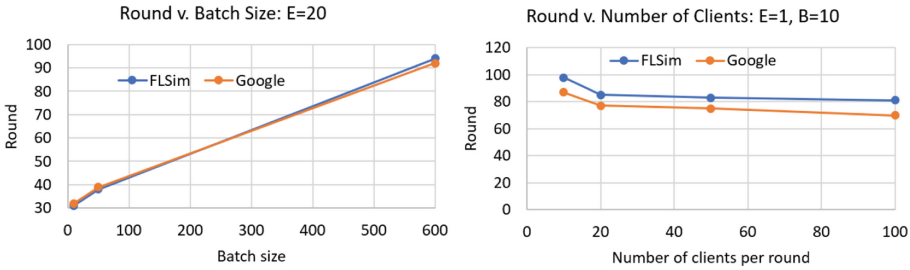
Table 4. 2NN MNIST with different federation sizes; Batch = 10; Epoch = 1

#Clients	LearnRate	Rounds	Rounds [22]
10	0.1	98	87
20	0.1	85	77
50	0.1	83	75
100	0.1	81	70

The numbers of rounds with different epochs, batch sizes, and number of clients are graphically depicted in Figs. 5(b), 6(a), and 6(b), respectively.

4.2 Federated Learning with Non-IID Data

In a federated learning system, when the clients have IID (independent and identically distributed) data, high model accuracy can generally be achieved, as demonstrated in the Basic Case above (e.g., Fig. 5(a). Zhao et al. [30] studied the impact on model accuracy when clients have non-IID data instead. They demonstrate that the model accuracy is significantly reduced, especially in the



(a) (a) Round v. batch size (b) (b) Round v. number of clients

Fig. 6. 2NN results in comparison with [22]: (a) Epoch (E) set to 20, batch size (B) set to 10, 50, 600. (b) Epoch set to 1, batch size set to 10, number of clients set to 10, 20, 50, 100.

extreme case where each client has data from only one class. To address this problem, they propose creating a common set of data and giving it to all the clients. It has been shown that even with a small amount of shared data, the model accuracy can be greatly improved.

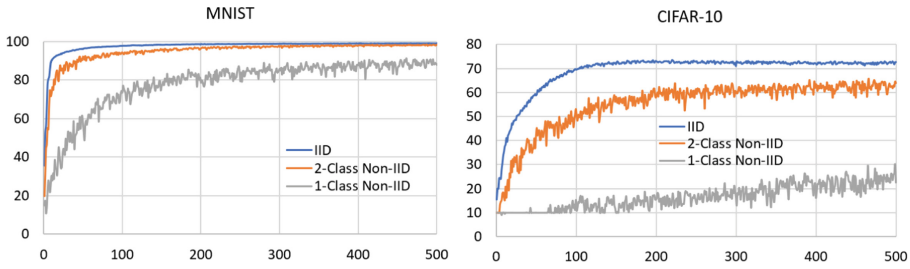
In this case study, we show how this problem can be studied with FLSim. Two CNNs are used in this test case: one for MNIST and one for CIFAR-10 [5]. The CIFAR-10 is also an image dataset with 10 classes. It has a training set of 50,000 images and a test set of 10,000. The CNN for MNIST is the same as the one in [22], with 1,663,370 parameters. However, [30] does not provide the parameters of the CNN for CIFAR-10 in that work. Therefore we have created our own network using the PyTorch library. This CNN has three convolution layers with sizes 3×64, 64×128, and 128×256 respectively, and three fully connected layers with sizes 1024×128, 128×256, and 256×10 respectively. The total number of trainable parameters is 537,610.

Components for this test case are mostly available from the FLSim framework. The only extension required is to create two utility classes that create non-IID distributions for MNIST and CIFAR-10 respectively. We list the major components for this test case in Table 5.

Table 5. FLSim components for the non-IID test case.

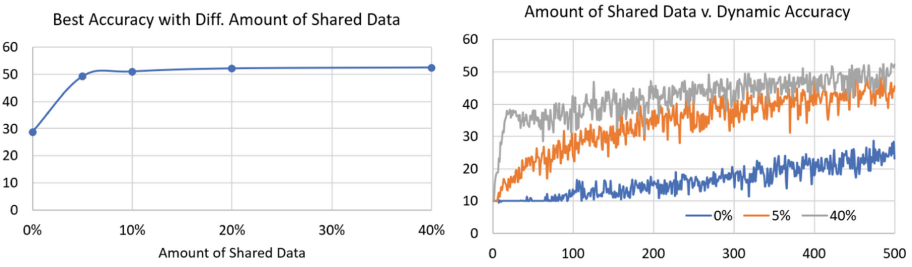
• The CNN models for both MNIST and CIFAR-10 based on PyTorch
• The MNIST and CIFAR-10 datasets
• The default federation `FLFederation`
• The `FLRandomClientSelector` to randomly select 10 clients out of 100 in each training round
• The `PytorchLocalTrainer`, a PyTorch based local training module
• Two utility classes that respectively divide the MNIST and CIFAR-10 into non-IID distributions

In the first step of this test case, we run the training tasks for 500 rounds on the two data sets with three different distributions: IID, 2-class non-IID where each client has data from 2 classes, and 1-class non-IID where each client has data from only 1 class. Using the same parameters as in [30], we set the batch size B to 10, and epoch E to 1. Learning rates are respectively 0.01 and 0.1 for MNIST and CIFAR-10.



(a) (a) CNN from [22]: 500 rounds (b) (b) CNN for CIFAR-10; 500 rounds

Fig. 7. Impact of training data distribution on model accuracy: (a) MNIST on CNN. (b) CIFAR-10 on CNN. Batch size = 10, epoch = 1.



(a) (a) Best accuracy v. amount of shared data (b) (b) Accuracy evolution w. different shared data

Fig. 8. CNN on CIFAR-10, 1-class non-IID data: impact of amount of shared data on model accuracy: (a) best accuracy (b) accuracy evolution

Figures 7 (a) and (b) show the evolution of test set accuracy for MINST and CIFAR-10 respectively. For MNIST, the trend is very similar to [30]. For CIFAR-10, because our CNN likely has fewer parameters than the one used in [30], the achieved accuracy level is lower, even in the IID case. However, the impact of data distribution is still clearly demonstrated.

In the second part of this test case we evaluate the effect of adding a set of shared data to all the clients. This is done with the CIFAR-10 dataset using the 1-class non-IID distribution.

In this test, each client has 500 images from 1 class only. We create a common set of data that includes images from all the 10 classes. The size of the common set range from 25 to 200, representing 5% to 40% of each client's data.

Table 6 lists the best accuracy obtained in 500-round runs using different sizes of shared data. The trend is graphically illustrated in Fig. 8(a) as well. Without the shared data, the best accuracy we have achieved is 28.7%. With only 5% of shared data, this is raised to 49.2%, a 71% improvement. However, adding more shared data appears to have limited benefits, especially beyond 20% in this case.

Table 6. CNN on 1-class non-IID CIFAT-10: best accuracy with different amount of shared data

Shared data	0%	5%	10%	20%	40%
Best accuracy (%)	28.7	49.2	51.0	52.2	52.5

Figure 8(b) shows the accuracy evolution of this test. We compare the original test case that includes no shared data with adding respectively 5% and 40% shared data. It is clear that, by adding just a small amount of shared data we can dramatically increase the accuracy level. These results conform very well with the findings in [30].

4.3 Real-Time Federated Learning

When federated learning was first proposed, it was expected that the clients would only participate in the training task when the device is connected to a power source [22]. This is likely because deep learning training is very compute intensive, and therefore could seriously affect the battery life of mobile devices.

This requirement, however, goes against the ubiquitous nature of mobile devices and prevents timely utilization of user data. It is conceivable that removing this restriction can expand the application scope of federated learning, particularly for targeting real-time machine learning tasks [1]. Recently, Li et al. [18] proposed SmartPC, a framework that allows on-device federated learning to take place when the device is not being charged. Among the key features of that work are the following:

- Instead of making the server wait for all clients to respond, SmartPC ends a training round when a certain proportion (e.g., 80%) of clients have responded.
- The server estimate a training deadline in each round. Based on the deadline, an on-device controller adjusts the device CPU frequency to meet the deadline and to minimize energy consumption of the training process.

In this case study, we use FLSim to create a simulator to study a simplified version of the main problems addressed in [18]. Specifically, we focus on two

problems: 1) how it would impact model accuracy and training time if the server uses only a subset of client updates, and 2) how a device can meet the training time requirement while saving energy.

As presented previously, a special feature of FLSim is its incorporation of performance and power models for studying system performance and energy efficiency. This feature is key to this test case.

We make the following changes to FLSim in order to implement the simulator.

- We extend the orchestration layer shown in Fig. 3 to add two additional data items in the client-server communications. From the server to the clients, a training deadline is appended to the model parameters, and from the clients to the server, we add the simulated completion time.
- A new client type is created which includes a device profile for computing performance and energy consumption.
- A new server type is also created because now it has the additional job of selecting the subset of clients as well as determining the training deadline.

This real-time federated learning system works as follows:

- Step 1. The server broadcasts the model parameters and the training deadline to all clients. The deadline for the very first round is based on the device profiles.
- Step 2. Clients perform local training, and compute the associated time and energy using their device profiles. Clients then send model updates along with training time to the server.
- Step 3. The server selects the first $p\%$ of clients (p is predefined) based on their completion time and performs model averaging using their updates only. The deadline is updated using a mechanism such as exponential weighted moving average.
- Repeat the above steps for the next round.

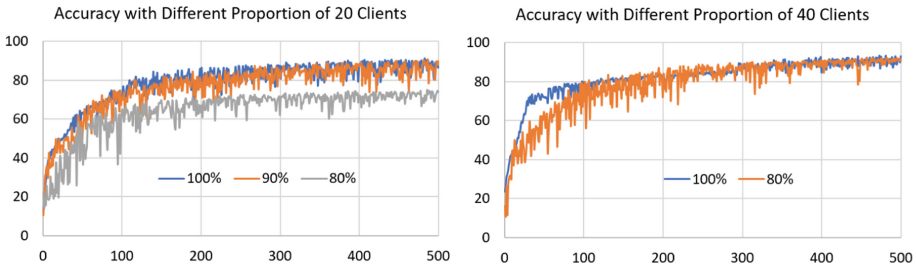
The process for a client to compute its training time and energy is as follows. As explained, each client has a device profile which includes a performance table and a power table. The performance table is basically an array of training times under different CPU frequencies, $[t_1, t_2, ..., t_n]$, in ascending order, while the power table contains the corresponding power consumptions $[p_1, p_2, ..., p_n]$ in descending order. Given a deadline d, we first compute a base training time T and base energy E as follows. If $d < t_1$, set $T = t_1, E = p_1 * t_1$. If $d > t_n$, set $T = t_n, E = p_n * t_n$. Otherwise set $T = d$. Assuming $t_i < d < t_{i+1}$, we find α such that $d = \alpha * t_i + (1-\alpha) * t_{i+1}$. The corresponding power is $p = \alpha * p_i + (1-\alpha) * p_{i+1}$, and set $E = p * d$. Finally, to simulate random factors that affect the training time and energy, random values are added to T and E.

In this experiment, we use the CNN and MNIST dataset with 1-class non-IID distribution as described in Sect. 4.2. Two different device profiles are used in the federations with a 40–60 split. We summarize the components used in Table 7.

Test results for this use case are illustrated in Figs. 9 and 10. In Fig. 9 we show the accuracy in 500-round runs when the server uses a certain proportion

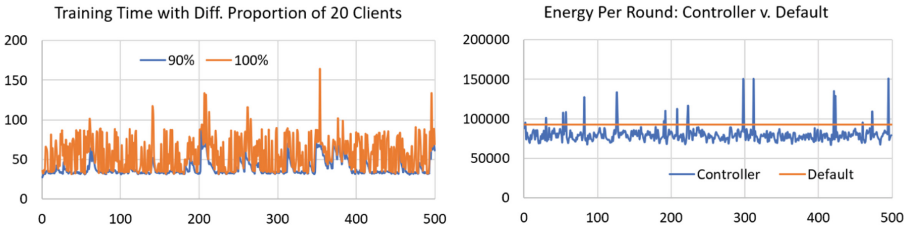
Table 7. FLSim components for the non-IID test case.

• The CNN model for MNIST
• The MNIST dataset
• The utility class that divides MNIST into non-IID distributions
• The real-time federation `FLFedRT`, including corresponding server and client components
• Two device profiles representing two device models
• The `FLAllClientSelector` that selects all the clients in a given set
• The `PytorchLocalTrainer`, a PyTorch based local training module



(a) (a) 20 clients (b) (b) 40 clients

Fig. 9. Model accuracy in 500-round runs with different proportions of clients selected in each round: (a) 20 clients (b) 40 clients



(a) (a) Training time: 90 percentile v. longest (b) (b) Energy consumption: controller v. default

Fig. 10. Training time and energy consumption.

of client updates to update the global model. Figure 9(a) shows the results from a small federation of 20 clients. Although setting the proportion to 80% has a clear gap of accuracy level compared with using all clients, increasing the proportion to 90% results in the same accuracy level as using all clients. This means that the server can ignore the 10% slowest clients in each round and will not lose model accuracy. Clearly this would reduce the overall training time. In Fig. 10(a) we compare the 90 percentile training time in each round with the

longest time among the 20 clients. The corresponding energy results are shown in Fig. 10(b) for one particular client. Here we compare the case of dynamically adjusting CPU frequency against the default, which is to always use the highest frequency for the training task. It is important to point out that the time and energy results here are not really important. What we are demonstrating here is the capability of FLSim for this type of research. Quantified studies obviously require more accurate models.

Intuitively, as the size of the federation increases, the proportion of clients selected can be reduced to get the highest levels of accuracy. This indeed is the case. Figure 9(b) shows the results for the same test as in Fig. 9(a) except that the number of clients is increased to 40. Now we can see setting the proportion of clients to 80% can achieve the highest level of accuracy after about 200 rounds.

5 Evaluation of Using Multiple CPU Processes

In this section we briefly present some performance results of FLSim when it uses multiple CPU processes. That is to say, the federation is built with `ServerMP` and `ClientRankMP`. Experiments here are run on a server with a 16-core Intel Xeon E5-2620 CPU. The OS is Ubuntu 18.04.

Table 8. Multi-Process Performance: Time in Seconds for One Training Round.

Test Case	#Clients	1 Proc	2 Proc	5 Proc	10 Proc
CNN MINST E=1, B=10	10	27.8	18.8 (1.5×)	13.7 (2.0×)	13.3 (2.1×)
	20	46.0	28.0 (1.6×)	19.2 (2.4×)	17.9 (2.6×)
	40	83.0	47.0 (1.8×)	27.6 (3.0×)	23.4 (3.5×)
CNN CIFAR E=1, B=10	10	40.7	30.2 (1.3×)	23.9 (1.7×)	21.7 (1.9×)
	20	62.7	40.8 (1.5×)	28.7 (2.2×)	24.9 (2.5×)
	40	105.1	63.1 (1.7×)	38.1 (2.8×)	31.9 (3.3×)
2NN MINST E=10, B=10	10	26.7	13.1 (2.0×)	6.6 (4.0×)	4.8 (5.6×)
	20	45.1	24.1 (1.9×)	11.4 (4.0×)	7.1 (6.4×)
	40	94.7	45.6 (2.1×)	21.1 (4.5×)	12.1 (7.8×)

Table 8 shows the time in seconds to run one training round with three different models. For each model we tested three different federation sizes and four different numbers of CPU processes. Speedup numbers for the multi-process cases are also displayed. A couple of observations can be made. First, while the two CNN models have similar speedup trends, the simpler 2NN model has a very different trend. Second, as we increase the number of clients, the workload in a training round is increased, and this leads to higher speedup numbers. Third, for the CNN models, using more than five processes has limited effect on speedup.

6 Conclusions

In this paper, we propose FLSim, a simulation framework for federated learning in order to efficiently build different simulators to investigate different scenarios in federated learning. Different from the ad hoc simulators, FLSim can be envisioned as an open repository of building blocks for creating simulators. Specifically, FLSim consists of a set of software components organized in a well-structured software architecture that provides the foundation for maximizing flexibility and extensibility. Developers can create different simulators through easily putting the selected components together, thus allowing developers/researchers to focus on the problems being studied. In addition, we use different case studies to demonstrate the effectiveness of FLSim.

Acknowledgement. This work is supported by the National Key R&D Program of China (No. 2019YFB2102100), Science and Technology Development Fund of Macao S.A.R (FDCT) under number 0015/2019/AKP, Guangdong Key R&D Project (No. 2020B010164003), Shenzhen Discipline Construction Project for Urban Computing and Data Intelligence.

References

1. Real Time Applications of Machine Learning. https://www.redalkemi.com/blog/post/5-real-time-applications-of-machine-learning
2. Pytorch. https://pytorch.org
3. Tensorflow. https://www.tensorflow.org
4. Ram, A.: How smartphone apps track users and share data (2018). https://ig.ft.com/mobile-app-data-trackers/
5. Krizhevsky, A., Nair, V., Hinton, G.: The cifar-10 dataset. https://www.cs.toronto.edu/~kriz/cifar.html
6. Bishop, C.M., et al.: Neural Networks for Pattern Recognition. Oxford University Press, Oxford (1995)
7. Bonawitz, K., Eichner, H., Grieskamp, W., Huba, D., Ingerman, A., Lvanov, V.: Towards federated learning at scale: system design. arXiv preprint arXiv:1902.01046 (2019)
8. Bonawitz, K., et al.: Practical secure aggregation for privacy-preserving machine learning. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 1175–1191. ACM (2017)
9. Buschmann, F., Meunier, R., Hans, R., Peter, S., Michael, S.: Pattern-Oriented Software Architecture: A System of Patterns, vol. 1. Wiley, New Jersey (1996)
10. Calheiros, R.N., Ranjan, R., Beloglazov, A., De Rose, C.A., Buyya, R.: Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Softw. Pract. Experience **41**(1), 23–50 (2011)
11. Mo, F., Haddadi, H.: Efficient and private federated learning using tee. In: EuroSys (2019)
12. Fowler, M.: Inversion of control containers and the dependency injection pattern (2004). https://martinfowler.com/articles/injection.html

13. Howell, F., McNab, R.: Simjava: a discrete event simulation library for java. Simul. Ser. **30**, 51–56 (1998)
14. Wong, J.C.: The Cambirdge Analytica scandal changed the world, but it didn't change Facebook (2018). https://www.theguardian.com/technology
15. Konečnỳ, J., McMahan, H.B., Yu, F.X., Richtárik, P., Suresh, A.T., Bacon, D.: Federated learning: strategies for improving communication efficiency. arXiv preprint arXiv:1610.05492 (2016)
16. Hautala, L.: Google tool lets any AI app learn without taking all your data (2018). https://www.cnet.com/news/google-ai-tool-lets-outside-apps-get-smart-without-taking-all-your-data/
17. Lalitha, A., Shekhar, S., Javidi, T., Koushanfar, F.: Fully decentralized federated learning. In: Third Workshop on Bayesian Deep Learning (NeurIPS) (2018)
18. Li, L., Xiong, H., Guo, Z., Wang, J., Xu, C.: Smartpc: hierarchical pace control in real-time federated learning system. In: 2019 IEEE Real-Time Systems Symposium (RTSS) (2019)
19. Hamblen, M.: Mobile users prefer Wi-Fi over cellular for lower cost, speed, and reliability (2012). https://www.computerworld.com/article/2506011/
20. Hamblen, M.: Google AI Blog: Federated Learning: Collaborative Machine Learning for Mobile Devices (2017). https://ai.googleblog.com/2017/04/federated-learning-collaborative.html
21. Marketing-Schools: Marketing Mobile Phones (2018). http://www.marketing-schools.org/consumer-psychology
22. McMahan, H.B., Moore, E., Ramage, D., Hampson, S., Arcas, B.A.y.: Communication-efficient learning of deep networks from decentralized data. In: Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (2017)
23. Halpern, S.: The champaign for mobile phone voting is getting a midterm test (2018). https://www.newyorker.com/tech/annals-of-technology/
24. Smith, V., Chiang, C.K., Sanjabi, M., Talwalkar, A.: Federated multi-task learning. In: Advances in Neural Information Processing Systems, pp. 4424–4434 (2017)
25. Sprague, M.R., et al.: Asynchronous federated learning for geospatial applications. In: Monreale, A., et al. (eds.) ECML PKDD 2018. CCIS, vol. 967, pp. 21–28. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-14880-5_2
26. Wang, J., et al.: Manifold: a parallel simulation framework for multicore systems. In: 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 106–115 (2014)
27. Wang, J., Beu, J., Yalamanchili, S., Conte, T.: Designing configurable, modifiable and reusable components for simulation of multicore systems. In: 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, pp. 472–476 (2012)
28. Wang, S., et al.: Adaptive federated learning in resource constrained edge computing systems. Learning, vol. 8, p. 9 (2018)
29. Lecun, Y., Cortes, C., Burges, J.C.: The mnist database of handwritten digits. http://yann.lecun.com/exdb/mnist/
30. Zhao, Y., Li, M., Lai, L., Suda, N., Civin, D., Chandra, V.: Federated learning with non-iid data. arXiv preprint arXiv:1806.00582 (2018)