Imperial College London

Department of Electrical and Electronic Engineering

M.Eng. Electronic and Information Engineering

Final Year Project - Final Report

Wireless Federated Learning

Author:
Alessandro Serena
alessandro.serena16@imperial.ac.uk
CID: 01188591

Supervisor:
Dr Deniz Gündüz
Imperial College London
EEE Department
Information Processing and
Communications Lab

Second Marker:
Dr Carlo Ciliberto
Imperial College London
EEE Department
Intelligent Systems and
Networks Group

Date: Wednesday 17 June, 2020

I affirm that I have submitted, or will submit, electronic copies of my final year project report to both Blackboard and the EEE coursework submission system.

I affirm that the two copies of the report are identical.

I affirm that I have provided explicit references for all material in my Final Report which is not authored by me and represented as my own work.

Acknowledgements

I would like to thank my supervisor Dr Deniz Gündüz for his help and support throughout the project. I would like to also thank Mehmet Emre Ozfatura and everyone from the Information Processing and Communications Lab.

I would like to express my sincerest gratitude to my mum Giuseppina, dad Gianni, and brother Andrea, for all they have done to support me throughout this voyage, and for having shaped me into the person that I am today.

Last, but not least, I would like to thank my friends for their love and support. A heartfelt thanks goes to the Barkston and Cromwell family for being an everlasting source of inspiration and for making me feel at home every day during these years.

Contents

Abstract

Recent advances in the computing capabilities of edge devices have led to the possibility of training complex models at the network edge. In particular, there is significant interest in the field of federated learning (FL), a technology that enables distributed training of models on data held by multiple parties, while preserving the privacy of such data. To improve the efficiency of FL at the wireless edge, hierarchical FL has been introduced, where local models are trained with the help of small cell base stations, while a global model is updated occasionally through the macro base station. The aim of this project is to assess the impact of client mobility and non-IID data distribution across clients in hierarchical FL. We investigate multiple learning algorithms that have been developed in the FL and distributed learning literature, and discuss strategies to deal with non-IID data.
The results and analysis presented in this report are achieved by executing software simulations of edge model training on a network of devices.

Chapter 1

Introduction

1.1 High-Level Context

Many modern applications rely on models trained using Machine Learning (§2.1) techniques on large-scale datasets [11]. However, in cases when data is generated by edge devices (e.g. mobile phones, IoT smart sensors, wearable devices, etc.), it is often not possible or inefficient to harvest this information and send it to a central server to perform a centralized fitting of the model due to bandwidth constraints and privacy concerns.

In Federated Learning [29] (§2.2) the model is updated in a distributed fashion. Edge devices update the global model on their local datasets, and then send the updates to a central server that aggregates them generating a new global model. This process ensures that the data held by the clients never gets exposed, hence preserving its privacy.

Considering that large-scale wireless implementations of this process can involve millions of devices, it is trivial to see that having only one aggregation server constitutes a huge bottleneck due to wireless communication latency and processing time [1].
To solve this issue, Hierarchical Federated Learning (HFL) [1, 27] has been proposed (§2.3). In this setting, an intermediate layer of small cell base stations (SBSs) is introduced. The clients communicate with one of the local base stations, which then periodically send their local models to the macro base station (MBS) to generate a global model.

1.2 Introduction of Project Objectives

We can now try to place the scope and objectives of this project in the context described in the section above.

1.2.1 Client Mobility

A common shortfall of [1] and [27] is the fact that both papers present HFL assuming that the clients are fixed and always communicate with the same SBS, and that the clients are equally distributed in the clusters. These assumptions clearly do not hold in reality, where FL is meant to be performed by mobile edge devices such as smartphones or wearables. The aim of this project is hence to investigate how wireless HFL systems perform when clients are allowed to move and to communicate with different SBSs, and how the existing HFL algorithms can be modified to take advantage of client mobility.

Understanding the behaviour of a HFL system when clients are allowed to move is not only important from a communications or distributed systems point of view (e.g. how devices can connect to a new SBS when they change geographical zone, how SBSs should handle the connection/disconnection of a user, etc.), but it is also crucial with regards to the learning task. In facts, as presented in [1], SBSs need to collaborate together with the MBS in the inter-cluster model averaging because otherwise they would not have any knowledge learnt from datasets of clients that do not belong to their own pool of devices.

Intuitively, if clients are allowed to move and hence to carry their local datasets to different SBSs, the advantage of performing inter-cluster model averaging decreases. If all devices were moving and ended up in all the clusters, every SBS would have received updates based on all the data held by the clients and therefore would end up with the same model parameters without having to share its model with the MBS.
One intuition is that it might be possible to reduce the frequency of the inter-cluster model averaging (thus saving bandwidth) as devices move more on average.

1.2.2 Data Distribution Across Clients

Another key aspect that is largely studied in the FL literature is the impact that the training data distribution across clients has on the global learning task. [14, 47]

To better explain this, let us take as example a simple classification task, in which we try to identify whether in an image there is a either a dog or a cat. Ideally we would want to train a model which performs equally well on samples of both classes. However, it is trivial to see that if we have many more images of dogs, the model will be skewed in predicting dogs, and the model will have issues in recognizing cats, as simply it has not been trained with enough cat images.

In centralised Machine Learning, since we have all the data available in one place, we are able to balance the amount of samples in each class (e.g. by performing data augmentation) and optimize our models using iterative methods (e.g. gradient descent) on data that represents evenly all classes.
Moreover, if we had to distribute the learning task on multiple machines, since we have control over the data given to each learner, we can draw subsets of samples that have the same (hence balanced) statistical distribution. We refer to this as an IID (Independent and Identically Distributed) distribution of data.

In Federated Learning, training data is held privately by the clients that use it to update the global model. Since we do not have any control over the data distribution at the clients, it is necessary to take into account that data can be heterogeneously distributed across clients. This means that the updates that are received by the server from each client can differ significantly if the data used to generate said updates has a different distribution. In this case we say that the data is non-IID.
Note that having non-IID data is quite a realistic assumption, if for example we are trying to collectively train a dog-cat classifier model with FL, where the data comes from the pictures taken by human users.

Therefore, in this project, we will analyze how the performance of FL learning algorithms differs when the data distribution across clients is non-IID, and we will discuss some approaches to make the learning more robust to skewed data distributions.

1.2.3 Hierarchical Federated Learning Simulator

In order to tackle the questions outlined above, we will need to simulate an HFL system, and analyze the results. For this purpose, we will design and develop a software simulator and a testing suite.

1.3 Overview of Report Structure

To present the work done in this project, we will start by presenting the background knowledge necessary to follow the discussion (Chapter 2).
Then we will move onto describing in more details the technical requirements that we need the HFL simulator to satisfy to answer the questions described above (Chapter 3). This will be followed by a high-level overview of the software design (Chapter 4) and by the description of the implementation (Chapter 5). In Chapter 6 we will show how the simulator was verified for correct operation.
Afterwards, we will describe the simulations run on the HFL system (Chapter 7), and summarize the work done and the results obtained (Chapter 8).
In Chapter 9 we will draw conclusions on the project, and outline a set of tasks that can be performed as future work.
The last section, Chapter 10, presents a user guide on how to use the simulator, and how to extend its functionalities.

Chapter 2

Background

2.1 Machine Learning

The objective of Machine Learning , in the setting of a *supervised learning* task, is to find the parameters of a model that satisfies the an finite-sum objective function of the form

$$\min_{w \in \mathbb{R}^d} f(w) \quad \text{where} \quad f(w) = \frac{1}{n} \sum_{i=1}^{n} f_i(w) \tag{2.1}$$

in which $f_i(w) = l(x_i, y_i; w)$ represents the error of the prediction on input data (x_i, y_i) of the model with parameters w [8].

The model parameters (also called *weights*) are learnt from a training set by performing iteratively a two-step process called *gradient descent*. In the first step, *forward propagation*, each training sample is input into the model and the output is computed by using the current model weights. The model output is then compared with the ground truth label of the training datapoint. The distance between these two values represents the loss (i.e. error) of the current set of weights[1].
The second step, *backpropagation*, consists in calculating $\nabla l(x, y, w_t)$, the gradient of the loss on the datapoint x with ground truth label y with respect to the current set of parameters w_t, and applying the following update to the weights

$$w_{t+1} = w_t - \eta_t \frac{1}{|\mathcal{D}|} \sum_{x \in \mathcal{D}} \nabla l(x, y, w_t) \tag{2.2}$$

[1]There are various ways of computing the loss, depending on the task that we want the model to perform. Generally, *euclidean distance (L2-norm)* or *mean squared error (MSE)* are used to express the loss in regression problems, *categorical cross-entropy* for classification.

where $\mathcal{D}$ is the training set.
Since our aim is to reduce the loss, the formula above shows that, at each backpropagation step, the current weights are modified in the opposite direction of the average gradient information and the amount of variation is regulated by the *learning rate* η.[2]

Minibatch SGD. One of the most common variations of gradient descent optimization used in machine learning is *minibatch stochastic gradient descent* (minibatch SGD, also known simply as SGD). In minibatch SGD, the gradient descent step described in equation (2) is computed on a small batch of datapoints randomly selected from the training set (rather than on the entire dataset). To reuse datapoints in multiple minibatches, minibatch SGD goes over the entire training set more than once; each complete pass over the training set is referred as one *epoch*. This method reduces significantly the computational cost of a backpropagation step because less derivatives have to be computed, and improves generalization accuracy.
It is necessary to note that the performance of this optimization technique is controlled by the hyperparameters *minibatch size* and *number of epochs*.

2.1.1 Large Minibatch SGD

Researchers in [9] have studied how to increase efficiency of minibatch SGD for distributed synchronized learning applications. When the learning task is split among multiple learners, using small batch sizes increases the number of batches per epoch and therefore the number of times that the learners have to communicate with each other. The results in [9] prove that it is possible to increase the batch size up to 8192 and that this reduces drastically the training time required.
In their work, they present some hyperparameter tuning techniques that are necessary to achieve said results:

- **Learning rate scaling:** when the minibatch size is multiplied by k, multiply the learning rate by k.
- **Learning rate warmup:** following the expression in equation (2)

[2]In the expression provided η has a subscript t because the learning rate can change over iterations due to other hyperparameters such as *learning rate decay*, or techniques like *learning rate warmup* (§2.1.1).

we can compare the SGD weight vector update step for a small batch size (n) and a larger one (kn). In the first case, after k iterations the resulting weights are given by

$$w_{t+k} = w_t - \eta \frac{1}{n} \sum_{j<k} \sum_{x \in \mathcal{B}_|} \nabla l(x, w_{t+j}) \tag{2.3}$$

While for a single SGD step with batch size kn we have

$$w_{t+1} = w_t - \hat{\eta} \frac{1}{kn} \sum_{j<k} \sum_{x \in \mathcal{B}_|} \nabla l(x, w_t) \tag{2.4}$$

By setting the learning rate $\hat{\eta} = k\eta$ (following the scaling technique outlined in the point above) we see that the expressions (3) and (4) yield similar parameter vectors, and therefore a similar learning curve, only if we could assume that $\nabla l(x, w_{t+j}) \approx \nabla l(x, w_t)$. This assumption generally holds in most cases but when the batch size is scaled to very large values and at the beginning of training when the network is changing rapidly.
To solve the latter problem, it is suggested to use a warmup strategy to adapt the learning rate in the first epochs of training. *Constant warmup*, used in [11] can be performed by setting a low constant learning rate for the first epochs[3] and then setting it to $k\eta$. This method, however, yields a similar training curve only for low ranges of batch size scaling factors k.
For large k, it is advised to use a *gradual warmup*, where the learning rate is iteratively incremented from a low value to the desired $\hat{\eta} = k\eta$ during the initial epochs.
After warmup the learning process goes back to the original learning rate schedule.

Other parameters that have influence on distributed SGD are *batch normalization* [16, 15], *weight decay* [30], *momentum correction* [26], *gradient aggregation* (find more about this in §2.2 and §2.3), and *data shuffling* (i.e. how a dataset is split and assigned to the distributed learners).

2.1.2 Local SGD

Since using large minibatches in SGD can cause convergence issues and generalization problems, researchers in [25] have demonstrated that

[3]In [9] the warmup phase encompasses the first 5 epochs of training.

participants in a distributed learning system can achieve better accuracy and reduce communication rounds by performing locally several parameter updates with small batch sizes sequentially, before sharing their update.

They also discuss two variations of local SGD:

- **Post-Local SGD:** since in the initial phase of training weights change rapidly, gradient updates need to be communicated as soon as they are computed to achieve better convergence. In post-local SGD, the local SGD algorithm is only started after a few rounds of small minibatch SGD. This lets us take advantage of warmup strategies, as those described in §2.1.1.
  In mathematical terms, given that we want to start executing H local iterations after T rounds, the number of local iterations at round t is given by:
  $$H_{(t)} \leftarrow \begin{cases} 1 & \text{if } t \leq T \qquad \text{(Minibatch SGD)} \\ H & \text{if } t > T \qquad \text{(Local SGD)} \end{cases} \tag{2.5}$$
- **Hierarchical Local SGD:** in a hierarchically organized distributed learning system, local SGD is applied to different layers of the architecture using different numbers of minibatch SGD iterations.

2.2 Federated Learning

Modern applications make use of models trained on large-scale amount of data. However, when this data is generated by edge devices (e.g. text that is typed on a smartphone keyboard, heartbeat information recorded by a wearable device), it is often impossible or inefficient to send it to a central server to be used in the model fitting, due to privacy concerns and bandwidth constraints.
For this reason, Google researchers have developed *Federated Learning* (FL) [29], an **algorithm to perform privacy-preserving distributed machine learning on the edge**.
Federated optimization (i.e. optimization in federated learning) presents several key properties that make it differ from distributed optimization:

- The training data is generated by a particular user acting on a certain device and hence the data of each device would be **non-IID**.

- Some users may use their device more with respect to others, making the data generated **unbalanced**.
- Since FL is thought to be used on all kinds of edge devices, it is expected that the number of devices taking part in a FL system to be much larger than the number of data samples per user. The system can be said to be **massively distributed**.
- Edge devices have **limited communication**, as they can go offline and they use crowded or expensive connections.

In FL the learning task is handled by a loose federation of clients, coordinated by a central base station. The algorithm that controls FL is called *Federated Averaging* (FedAvg), and is described below and defined in algorithm 1.
The initial parameter vector is generated by the server. Then the algorithm proceeds in rounds all composed of the same steps:

1. A subset of the clients is chosen at random[4] to participate in the current learning round.
2. The latest parameter vector is distributed by the server to the selected clients.
3. Each client updates the model parameters by performing minibatch SGD on their local dataset.
4. The new weights are sent by each client to the server.
5. The server aggregates all the received updates by taking a weighted average

$$w_{t+1} = \sum_{k=1}^{K} \frac{n_k}{n} w_{t+1}^{k} \tag{2.6}$$

Since the training data is never shared by the clients, it is intuitive to understand how this process preserves privacy.

[4]A more efficient way to select clients is presented in [31] and discussed in §2.2.1.

Algorithm 1 Federated Averaging (FedAvg).
The K clients are indexed by k, and C is the fraction of clients selected to participate in each round; B is the local minibatch size, E is the number of local epochs, and η is the learning rate

```
1: Server executes:
2: initialize $w_0$
3: for each round $t = 1, 2, \dots$ do
4:     $m \leftarrow \max(C \cdot K, 1)$
5:     $S_t \leftarrow$ (random set of $m$ clients)
6:     for each client $k \in S_t$ in parallel do
7:         $w_{t+1}^k \leftarrow \text{ClientUpdate}(k, w_t)$
8:     end for
9:     $w_{t+1} \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} w_{t+1}^k$
10: end for

11: ClientUpdate($k, w$):                               ▷ Run on selected client $k$
12: $\mathcal{B} \leftarrow$ (split $\mathcal{P}_k$ into batches of size $B$)
13: for each local epoch $i = 1, 2, \dots, E$ do
14:     for batch $b \in \mathcal{B}$ do
15:         $w \leftarrow w - \eta \nabla l(w; b)$
16:     end for
17:     return $w$ to server
18: end for
```

In the following subsections, we will describe some enhancements that have been made to the Federated Learning algorithm.

2.2.1 Client Selection

In [29] the clients are selected randomly to participate in each federated averaging step. This causes the time necessary for each round to be strongly dependent to the random selection. To increase the efficiency of this process, researchers in [31] have added a `Client Selection` step at the beginning of each round before the server sends out the global model.

1. A fraction of clients is selected to participate in the `Client Selection` step.
2. Clients reply to the server with information about the time needed for the update and for the upload of the model.
3. The server chooses the clients to keep by constraining the maximum length of a round to a fixed value, while trying to maximize the number of clients chosen.

After the `Client Selection` step, the FL algorithm continues as described in §2.2.

2.2.2 Secure Aggregation

Secure aggregation [3] is a cryptographic technique that allows the clients to submit their updates encrypted to an *aggregation server*, which is able to aggregate them without decrypting them. The combined update is then sent to the central server by the aggregation server.
This allows for increased security because theoretically it is possible to derive the training data that generated an update if the update vector can be seen unencrypted.

2.2.3 Data Selection

[42] presents a technique for clients to perform data selection to decide which data to use when updating the model weights. This technique can positively influence the convergence rate, but mostly in cases when we deal with real-world data. Since in this project we are using benchmark datasets, which have balanced classes and we are able to perform normalization on the full dataset before assigning data to the clients, we chose not to implement client-level data selection.

2.2.4 SCAFFOLD

In [20] researchers present `SCAFFOLD`, a correction mechanism for weights updates designed for Federated Learning. This procedure attenuates the problem of local (client-level) updates drifting apart due to non-IID and imbalanced distribution of data across clients, as shown in Fig. 2.1.

Alongside with the server weights $\mathbf{x}$, the system keeps track of a variable for each client i, denoted as the control variate $\mathbf{c}_i$, which has the same shape as $\mathbf{x}$. Similarly, we keep track of the server control variate $\mathbf{c}$, and we ensure that $\mathbf{c} = \frac{1}{N}\sum_i \mathbf{c}_i$. All control variates are initialized at 0, and we retain their values across multiple rounds.

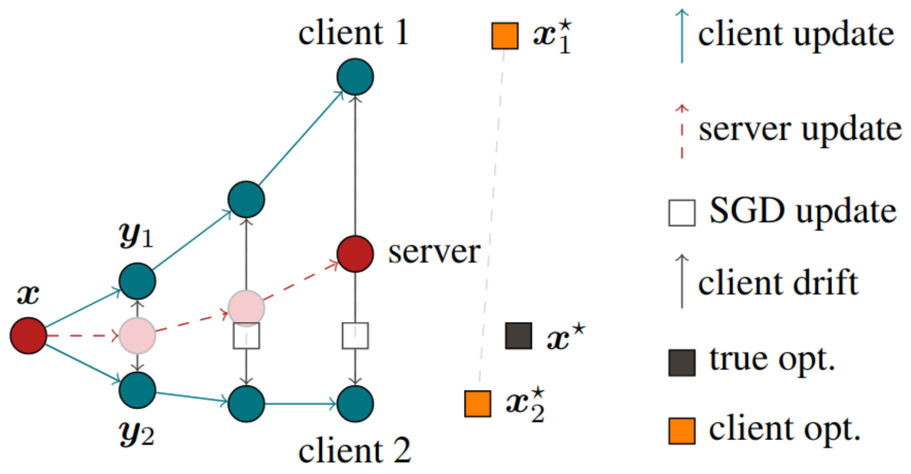


Figure 2.1: Visualization of the update drift. (Image taken from [20])

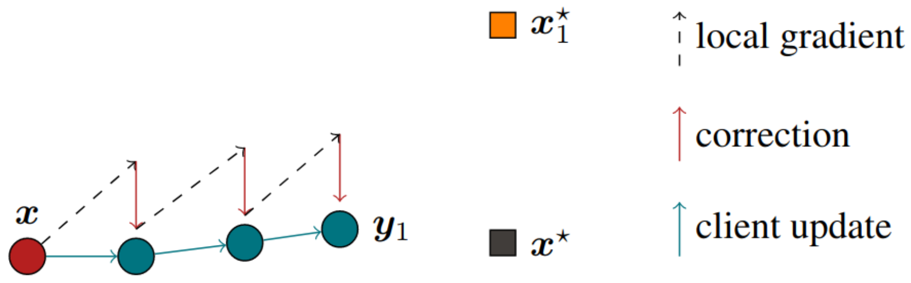


Figure 2.2: SCAFFOLD gradient correction. (Image taken from [20])

In each round, the server distributes $(\mathbf{x}, \mathbf{c})$ to the set of selected clients $\mathcal{S}$. Each client $i \in \mathcal{S}$ copies the server weight to its local model $\mathbf{y}_i \leftarrow \mathbf{x}$, and performs a local pass over its local data by doing K updates[5] as follows:

$$\mathbf{y}_i \leftarrow \mathbf{y}_i + \eta(g_i(\mathbf{y}_i) + \mathbf{c} - \mathbf{c}_i) \tag{2.7}$$

Then we update the client control variate $\mathbf{c}_i$. There are two ways:

$$\mathbf{c}_i^+ \leftarrow \begin{cases} \text{Option I.} & g_i(\mathbf{x}) \\ \text{Option II.} & \mathbf{c}_i - \mathbf{c} + \frac{1}{K\eta}(\mathbf{x} - \mathbf{y}_i) \end{cases} \tag{2.8}$$

The first option involves performing an additional pass over the client dataset to compute the gradient with the server weights; the second option is cheaper to compute and yields similar results by using the updated weights of the local model.

[5]The value of K represent the number of minibatch updates, hence it is dependent on the batch size and number of local epochs.

Afterwards, each client transmits the updates $((\mathbf{y}_i - \mathbf{x}), (\mathbf{c}_i^+ - \mathbf{c}_i))$ to the server, which proceeds to aggregate them together:

$$\mathbf{x} \leftarrow \mathbf{x} + \frac{1}{|\mathcal{S}|} \sum_{i \in \mathcal{S}} (\mathbf{y}_i - \mathbf{x}) \tag{2.9}$$

$$\mathbf{c} \leftarrow \mathbf{c} + \frac{1}{N} \sum_{i \in \mathcal{S}} (\mathbf{c}_i^+ - \mathbf{c}_i) \tag{2.10}$$

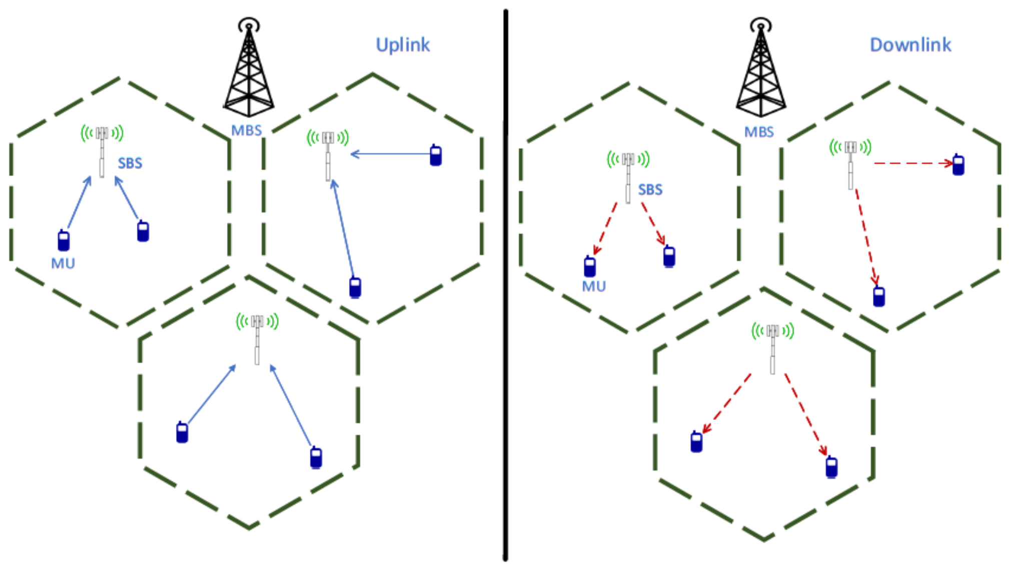
From this process we can see that the clients' and server's control variates are built over time, and that the local ones represent the specific drift given by each client dataset, whilst the server one gives information about the optimal desired direction of the updates.
Note that by setting $\mathbf{c}_i$ always to 0, SCAFFOLD becomes equivalent to Federated Averaging, as no correction is applied in the gradient descent step.
Moreover, the trade-off with using this correction mechanism is that of doubling the size of each message between the server and the clients, as we are required to send both the model parameters and the control variate.

2.3 Hierarchical Federated Learning

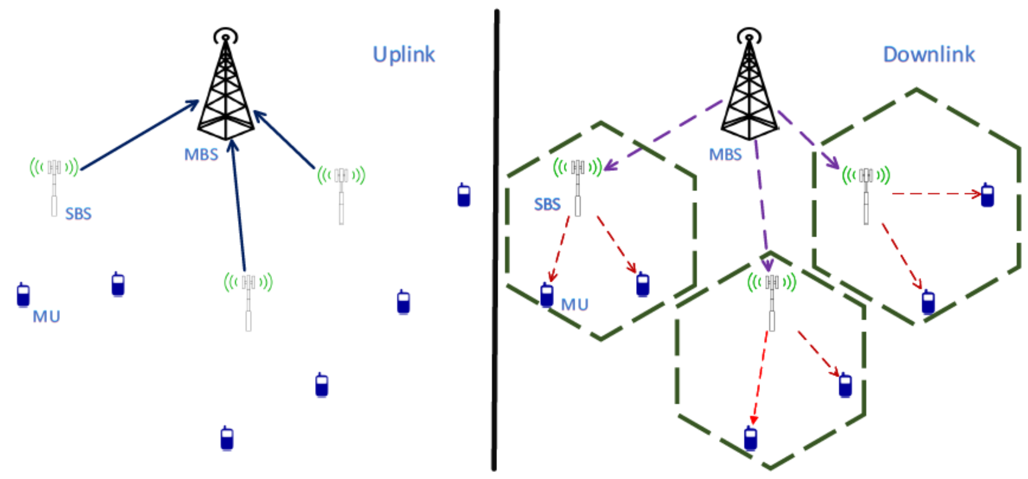
In edge learning, devices (mobile phones, IoT nodes) are often connected to each other and to the servers thanks to a wireless communication channel, an active area of research is that of studying how edge learning systems behave or how they can be improved when coupled with a wireless medium [48, 10, 2, 32, 24, 37]. Federated learning was envisioned to be used on large scale networks of devices. When these clients are in the range of millions, having a cloud-based system with only one aggregation server to which all clients send their updates represents a performance bottleneck. In *Hierarchical Federated Learning* [1, 27], we introduce an intermediate layer of small cell base stations (SBSs). Clients are grouped geographically around the closest SBS, to which they send their updates[6]. Periodically SBSs communicate their current model to a macro cell base station (MBS), which averages them and redistributes a new global model.

This process, together with gradient sparsification [39] reduces considerably the communication latency since it allows for carrier reuse in

[6]the learning process between clients within a cluster and their SBS works as described in §2.2



(a) Local gradient update



(b) Global model averaging

Figure 2.3: Hierarchical Federated Learning (image taken from [1])

different clusters, without sacrificing model accuracy.

The algorithm described in [1] is presented below.

Algorithm 2 Hierarchical Federated Averaging.
$\mathcal{M}$ is the set of clusters, the K clients in each cluster are indexed by k, and C is the fraction of clients in a cluster selected to participate in each round; H is the frequency of the global update, in terms of rounds.
B is the local minibatch size, E is the number of local epochs, and η is the learning rate.

```
1: MBS executes:
2: initialize $w_{MBS}$
3: for each cluster $m \in \mathcal{M}$ do
4:     $w_{SBS_m} \leftarrow w_{MBS}$
5: end for
6: initialize clients datasets
7: for each round $t = 1, 2, \ldots$ do                              ▷ Main Algorithm
8:     for each cluster $m \in \mathcal{M}$ in parallel do              ▷ Learn in each cluster
9:         $w_{SBS_m} \leftarrow$ ClusterUpdate($m, w_{SBS_m}$)
10:    end for
11:    if $\text{mod}_H(t) = 0$ then                                    ▷ Global update step
12:        $w_{MBS} \leftarrow \frac{1}{|\mathcal{M}|} \sum_{m \in \mathcal{M}} (w_{SBS_m})$
13:        for each cluster $m \in \mathcal{M}$ do                     ▷ Distribute new global model
14:            $w_{SBS_m} \leftarrow w_{MBS}$
15:        end for
16:    end if
17: end for

18: ClusterUpdate($m, w_{SBS_m}$):                                    ▷ Run on cluster $m$
19: $m \leftarrow \max(C \cdot K, 1)$
20: $S_t \leftarrow$ (random set of $m$ clients)
21: for each client $k \in S_t$ in parallel do
22:     $w_k \leftarrow w_{SBS_m}$
23:     $w_k \leftarrow$ ClientUpdate($k, w_k$)
24: end for
25: $w_{SBS_m} \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} w_{t+1}^k$

26: ClientUpdate($k, w$):                                             ▷ Run on selected client $k$
27: $\mathcal{B} \leftarrow$ (split $\mathcal{P}_k$ into batches of size $B$)
28: for each local epoch $i = 1, 2, \ldots, E$ do
29:     for batch $b \in \mathcal{B}$ do
30:         $w \leftarrow w - \eta \nabla l(w; b)$
31:     end for
32:     return $w$ to server
33: end for
```

2.4 Communication Latency

In order to analyze the HFL system from a communication standpoint, it is necessary to use a mathematical model to study the latency of the messages exchanged on a wireless medium.

The communication channel of a client is estimated as follows:

$$C_i = \sqrt{\frac{10^{-\frac{L_i}{10}}}{2}} \times |a + bj|^2 \tag{2.11}$$

where a,b are randomly sampled from a normal distribution $\mathcal{N}(0, 1)$, and L_i is the path loss associated with client i. This is computed accordingly to the outdoor path loss model reported by the European Telecommunications Standards Institute (ETSI) for LTE communications [7] which is

$$L_i = 128.1 + 37.6 \log_{10}\left(\frac{d_i}{1000}\right) \tag{2.12}$$

with d_i being the distance between the client and the base station, measured in meters.

The time of a transmission is computed by dividing the size of the message by the bit rate of the communication link. The data rate B_i is computed as follows.
Given a total bandwidth B, the antenna power of the transmitter P_t, the power spectral density of the AWGN channel noise N_0, and the communication channel C_i (computed as in Eq. (2.11)), we use the Shannon-Hartley theorem:

$$B_i = B \log_2\left(1 + \frac{P_t}{N_0}\frac{C_i}{B}\right) \tag{2.13}$$

Then, the time for sending m bits is simply

$$t = \frac{m}{B_i} \tag{2.14}$$

[7] The formula is taken from section 5.3.2.2.2 on page 10 of the RF requirements for LTE Pico NodeB [6]. The document reads "*Macro cell propagation model for urban area is applicable for scenarios in urban and suburban areas outside the high rise core where buildings are of nearly uniform height [...] assuming that the base station antenna height is fixed at 15 m above the rooftop, and a carrier frequency of 2 GHz is used*"

In Chapter 4 we will describe how this channel model is used to compute the latency and the rounds duration of the HFL system, together with the description of the scheduling algorithms used.

Chapter 3

Requirements Capture

As mentioned in the Introduction chapter (Chapter 1), software simulations of a Hierarchical Federated Learning system are a key element of this project, on top of which we are able to carry our analysis on client mobility and robustness of the learning process with respect to non-IID data distributions.
Let us here present the required software capabilities and data requirements of this project, from a high level point of view.

3.1 Software Requirements

From a software point of view, we require three main subsystems in this project, namely a HFL system simulator, a testbench, and a results visualizer. Each of them has the following key functional requirements.

- **Simulator** - We need to be able to do the following.
    1. Set the configuration of the simulation (e.g. learning algorithm, number of users, rate of mobility, etc.).
    2. Partition the training data accordingly to the configuration, and distribute the partitions to the clients.
    3. Run experiments.
    4. Log the state of the simulator at any time during the simulation.
    5. At the end of the experiment, merge the logs with the configuration information. If multiple experiments are run with the same configuration, combine all the results.
- **Testbench** - We need to be able to do the following.

1. Define a set or range of parameters that we want to test.
2. Run the simulations, possibly in parallel.
3. Label the results based on the tests group that is run.

- **Results Visualizer** - We need to be able to do the following.
  1. Load the results files.
  2. Filter the results based on the configuration parameters.
  3. Produce plots for the metrics recorded during the simulations in the results files.

3.2 Data Requirements

From a data point of view, the metrics that the simulator will produce, and that the results visualizer will handle, are:

1. The accuracy of the global model on the complete training set.
2. The accuracy of the global model on the test set.
3. The latency (i.e. duration) of each round.
4. The weight divergence (as described in §4.1.4.C) of the updates generated by the local training procedure.

Moreover, since these values will be generated by simulations that are strongly influenced by random initial conditions, and by random selection of client in each round, we require this data to be as statistically significant as possible.

Chapter 4

Analysis and Design

In this section we will outline a high-level overview of the design of the three subsystems that make up the software side of this project. In each subsection, the discussion will follow the structure of the software requirements described in Chapter 3.

4.1 Simulator

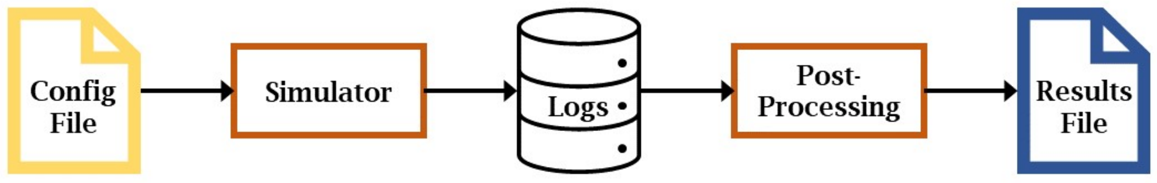


Figure 4.1: System diagram of the simulator.

4.1.1 Configuration

The simulator is configured by loading a JSON file that contains all the settings necessary to run an experiment. The required fields in the configuration files are described below.

- **System settings**
  - `n_clusters` : [integer]
    Specifies the number of clusters to simulate. More information on the arrangement of clusters is given in §4.1.3.B.
  - `n_clients` : [integer]
    Specifies the number of clients in the whole system.

- `server_global_rate` : [integer]
  Number of rounds between each global update step.
- `client_selection_fraction` : [float]
  Fraction of clients that take part in each round. Each cluster randomly selects the users based on its population and the fraction value. More populated clusters will then select more clients.
- `clients_mobility` : [boolean]
  Toggle for client mobility.
- `mobility_rate` : [float]
  Fraction in the range $[0, 1]$ that indicates the probability of clients to move to a different cluster at the end of each round. More information on the client mobility procedure can be found in §4.1.3.B.
- `move_to_neighbours` : [boolean]
  If set to true, clients will be allowed to move only to the cluster immediately adjacent to theirs.
- `clients_distribution` : [string]
  Indicates how the clients should be distributed across the clusters. The only value accepted is `balanced`, which assigns to each cluster $\lfloor$`n_clients/n_clusters`$\rfloor$ clients.

- **Data settings**
  - `model_type` : [string]
    Name of the neural network to use.
  - `model_init` : [string]
    Specifies the technique used to initialize the model weights. The only value supported is `random`, which sets a random initialization.
  - `dataset_name` : [string]
    Name of the dataset to use. The values supported are `"mnist"`, `"fmnist"` (for Fashion-MNIST),and `"cifar10"`.
  - `dataset_distribution` : [string]
    Specifies whether the dataset is distributed in a IID, non-IID, or spatially correlated non-IID fashion. The values supported are `"iid"`, `"non_iid"`, and `"non_iid_spatial"`.

- **Client settings**
  - `client_algorithm` : [string]
    Indicates the optimization algorithm that the clients use to perform local updates. It can be set to `"sgd"` for using minibatch gradient descent (§2.1), or to `"scaffold"` to use the algorithm described in §2.2.4.
  - `client_batch_size` : [integer]
    Batch size to use to perform local updates. Set it to larger values to use the optimization technique outlined in §2.1.1.
  - `client_n_epochs` : [integer]
    Number of local iterations performed by each client before sending the update to its SBS. Set this to a value greater than 1 to use the LocalSGD algorithm (§2.1.2).
  - `epochs_delay_localSGD` : [integer]
    Number of initial epochs for which the clients perform only 1 local update per round. The corresponding number of rounds of delay is computed as

    $$round\left(\frac{\texttt{epochs\_delay\_localSGD}}{\texttt{client\_selection\_fraction}}\right) \tag{4.1}$$

    Set this to a value greater than 0 to use the Post-LocalSGD algorithm (§2.1.2).
  - `client_lr` : [float]
    Learning rate used by the clients. It represents the base learning rate for a batch size of 32 samples. Therefore, if the `client_batch_size` parameter is different, the actual learning rate used is

    $$\texttt{client\_lr}\left(\frac{\texttt{client\_batch\_size}}{32}\right) \tag{4.2}$$

    This learning rate scaling is that described in §2.1.1.
  - `lr_warmup` : [boolean]
    Toggle for linear learning rate warmup. If it is enabled, the learning rate starts at 1/10 of its value, and it is linearly increased in the first 5 epochs of training, to reach its original value. Note that 5 epochs are converted into rounds as

    $$round\left(\frac{5}{\texttt{client\_selection\_fraction}}\right) \tag{4.3}$$

- **Simulator settings**
  - `stop_condition` : [string]
    It represents the metric that is checked to stop the simulation. The only value supported is `"rounds"`.
  - `stop_value` : [integer]
    Value that is used together with `stop_condition` to decide when to terminate the simulation.
  - `log_file` : [string]
    Path to the log file where the results are saved throughout the simulation.
  - `stdout_verbosity` : [integer]
    Level of verbosity of the information printed to the `STDOUT` during the simulation. The levels implemented are {0,1,2}.
  - `debug` : [boolean]
    If enabled, extra debug information is printed to the `STDOUT` during the simulation.

4.1.2 Data Partitioning

Once the configuration is loaded, the simulator proceeds to initialize its operation. The dataset chosen is loaded and distributed to the clients. The partitioning is performed accordingly to the setting specified in the `dataset_distribution` configuration parameter.
In all data distribution settings, each client receives the same amount of samples, computed as

$$\text{Partition size} = \frac{\text{Size of training set}}{\texttt{n\_clients}} \tag{4.4}$$

The **IID** data distribution is performed by selecting randomly for each client a subset of the training set, without replacement so that there are no repeated data samples across clients.

For the **non-IID** case, each class in the training set is divided in smaller partitions and each client receives one partition from 2 different classes. Note that this leads to client datasets that are strongly non-IID. This was decided as most research papers that discuss the problem with non-IID data distributions (e.g. [29, 20, 14]) use this type of partitioning.

In the **spatially correlated non-IID** case, we assign data to the clients so that only one cluster, and possibly its neighbouring clusters, have data samples coming from a class.

To better explain how data is distributed in this case, we first need to mention that the clusters are arranged in a shape which is as square as possible. For example, if the number of clusters is a perfect square (e.g. 9, 25, or 64 clusters) they will be arranged in a square layout (e.g. 3x3, 5x5, 8x8); otherwise (e.g. 6, 28, 54 clusters), the clusters will be arranged in a rectangular shape by minimizing the sum of the dimensions (e.g. 2x3, 4x7, 6x9). From a mathematical point of view this is done by finding all factors of the number of clusters and then choosing the median, if the number of factors is odd, or the two medians, if the number of factors is even.

Then, the data is distributed to the clients by by:

1. sorting the training set accordingly to the class labels;
2. dividing the samples into as many batches as there are clusters;
3. assigning one batch to each cluster, following the pattern shown in Table 4.1;
4. in each cluster, assigning the samples to the clients by sampling randomly without replacement from the cluster batch.

$\mapsto$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\downarrow$
$\downarrow$	$\leftarrow$	$\leftarrow$	$\leftarrow$	$\leftarrow$
$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\downarrow$
$\downarrow$	$\leftarrow$	$\leftarrow$	$\leftarrow$	$\leftarrow$
$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\cdot$

Table 4.1: Pattern for the assignment of spatially correlated non-IID data, starting from the top-left corner.

Table 4.2 shows an example of how the classes get distributed using this method.

0	0	0,1	1	1
3	3	3,2	2	2
4	4	4,5	5	5
7	7	7,6	6	6
8	8	8,9	9	9

Table 4.2: Distribution of 10 equally represented classes for the assignment of spatially correlated non-IID data, for 25 clusters arranged in a 5x5 grid.

4.1.3 Execution of Experiments

After the simulation is initialized, it will start executing the experiment. Each round has a similar structure, composed by the same three steps: model learning, client mobility, metrics computation.
We will now consider the first two steps, while the last one will be described in the following subsection.

4.1.3.A Model Learning

The learning of the model follows the HFL algorithm, as described in §2.3 and Algorithm 2. In each cluster we perform random client selection, the server model is distributed to the selected clients, which update it on their local dataset, and then send the updated model back for the update aggregation. If included in the current round, the SBSs send their learnt models to the MBS for the inter-cluster update step.
A more specific analysis of its software implementation is presented in Chapter 5.

4.1.3.B Client Mobility

Investigating the impact of mobility of clients on the learning task is one of the key objectives of this project. The mobility model is based on the assumption that clusters span a geographical region, and that therefore if a client moves away from its cluster, it is more likely that it will be in one of the adjacent clusters.

Then, given the 2-dimensional grid arrangement of clusters described in §4.1.2 , at the end of each round we iterate through all the clients, and randomly decide whether a client remains in its cluster or leaves

for another cluster. The probability that it will move is given by the `mobility_rate` parameter defined by the user in the configuration.
If a client is selected to migrate to another cluster, the destination cluster is chosen at random, where the probabilities associated to each cluster are equal to the inverse of the euclidean distance (i.e. L2-norm), and then normalized so that they all sum up to 1. An example of the probabilities for a system of 25 clusters is shown in Table 4.3.

5.57%	7.88%	5.57%	3.52%	2.49%
7.88%	X	7.88%	3.94%	2.63%
5.57%	7.88%	5.57%	3.52%	2.49%
3.52%	3.94%	3.52%	2.79%	2.19%
2.49%	2.63%	2.49%	2.19%	1.86%

Table 4.3: Example of probabilities of a client in the cluster marked with X to go to each cluster.

Moreover, we also have implemented a different mobility scheme, in which the clients are only allowed to move to clusters that are immediately adjacent to the one they are currently in. The algorithm works as described above, with the difference that in this case all neighbouring clusters have the same probability of being a client's destination. This mobility pattern is enabled by setting the configuration parameter `move_to_neighbours` to true.

4.1.4 Metrics and Logging

Throughout the simulations we wish to save the state of the system, in order to analyze it later and make evaluations on the impact of the initial parameters. The key metrics that are logged in this simulator are described in the following paragraphs.

4.1.4.A Model Accuracy

The first and most important metric that we capture on the HFL system is the accuracy of the global model. The model accuracy, computed with respect to both the training set and test set, is computed every time an inter-cluster averaging occurs and the global model is updated.

4.1.4.B Round Latency

The latency model is implemented on the assumption that each cluster has its communication channel independent from those of other clusters. This assumption is taken from [1], and it lets us simplify the analysis of latency, as we can treat each cluster as a separate wireless communication entity.

Now that we have defined the communication model, we will describe the scheduling model for the transmissions. In this simulator, for each cluster, the latency of each round is composed by two components, the downlink latency and the uplink latency.

In the **downlink** phase, the SBS broadcasts its model weights to the clients. We model this as a synchronous process (i.e. each client begins computing the model update only after they all have received the server parameters), so the overall latency of the downlink is determined by the time necessary for the message to be received by the client with the worst channel.
The downlink latency t_{DL} is then computed, using Eq. (2.13) and Eq. (2.14), as

$$B_{DL} = B \log_2 \left(1 + \frac{P_{SBS}}{N_0} \times \frac{C_{min}}{B} \right) \tag{4.5}$$

$$t_{DL} = \frac{m}{B_{DL}} \tag{4.6}$$

where P_{SBS} is the transmitting power of the cluster base station, B is the total bandwidth available, N_0 is the AWG noise power spectral density, and C_{min} is the minimum capacity of the channels of the clients selected to participate in the round.

In the **uplink** phase, we have clients perform their local update and then send the new model parameters to their SBS. However, we know that the clients do not have to wait for all the others to finish computation before they start sending. Therefore, we use a greedy client scheduling algorithm that jointly considers computation time and communication latency for each client.

The algorithm[1] is based on TDMA, and essentially schedules for uplink transmission the client that has the lowest remaining transmission time at any time, among the clients that have finished the model update phase. Therefore, the client that is transmitting will continue until it has finished, or will pause if a client with a much better channel data rate is ready to send[2]. For simplicity of the model, we do not account for any overhead for the process of interrupting a client transmission and letting a new client to start.
We show the scheduling algorithm in Algorithm 3, with the set of selected users in the round denoted by $\mathcal{S}$, $\mathcal{C}_{idle}$ as the set of client completed the computation waits for to be scheduled, and K_{ul} is the set of clients who send their model to the SBS.

Algorithm 3 Greedy Client Scheduling

```
1: Initialize remaining data: $Q_k = Q$, $\forall k \in C_{dl}$
2: Initialize set of idle clients: $\mathcal{C}_{idle} = \{\}$
3: Initialize set of uplink clients: $K_{ul} = \{\}$
4: while $|\mathcal{K}_{ul}| < |\mathcal{S}|$ do
5:     for $k \in \mathcal{C}_{idle}$ do Compute remaining transmission time:
6:         $D_k = Q_k / R_k$
7:     end for
8:     Choose client $k \in \mathcal{C}_{idle}$ with minimum $D_k$
9:     while $Q_k > 0$ and $\mathcal{C}_{idle}$ remains same do
10:        Schedule client $c_k$ and update $Q_k$
11:    end while
12:    if $D_k = 0$ then
13:        Update $\mathcal{C}_{idle}$: $\mathcal{C}_{idle} \leftarrow \mathcal{C}_{idle} \setminus \{k\}$
14:        Update $\mathcal{K}_{ul}$: $\mathcal{K}_{ul} \leftarrow \mathcal{K}_{ul} \cup \{k\}$
15:    end if
16: end while
```

Note that in our implementation we are scheduling for uplink all the clients in $\mathcal{S}$, therefore there is not a clear advantage in interrupting clients which are mid-transmission. However, if we were performing a client selection on the uplink or if we had a maximum time limit on the duration of the round, this scheduling strategy is beneficial, as it lets us give transmission priority to better connected clients at any time.

[1]This algorithm has been developed in collaboration with Mehmet Emre Ozfatura and Junlin Zhao, both members of the Intelligent Systems and Networks Group at Imperial College London.

[2]In a real implementation, we will need the SBS to be able to keep a buffer for each client model, so that transmission from a client can be interrupted and then restarted from where it was paused.

The **computation time** of the update performed by each client, i.e. the time when the client is added to the $\mathcal{C}_{idle}$ set, is obtained as follows:

$$t_{computation} = c_{rate} \times \text{(data samples)} \times \text{(trainable parameters)} \times \text{(epochs)} \tag{4.7}$$

where c_{rate} is the base computation speed of a client, randomly sampled from $\mathcal{N}(5 \times 10^{-9}, 10^{-9})$.[3]

As mentioned before, the latency of a round is then the sum of the duration of the downlink and uplink phase. Moreover, we assume that the MBS is connected to the SBSs with a low-latency, high-bandwidth channel (e.g. an optic fiber wired connection), and hence we assume that there is no latency associated with the inter-cluster global averaging step.

4.1.4.C Weights Divergence

This metric (introduced in [47]) informs us of how much do the local models learnt by the clusters differ on average when compared with the global model held by the MBS. It is computed after each inter-cluster averaging, before the updated global model is downloaded to the SBSs. The weights divergence (D) for a cluster i is computed as

$$D_i = \frac{||w_{SBS_i} - w_{MBS}||}{||w_{MBS}||} \tag{4.8}$$

then we compute the average across n clusters by taking

$$D_{avg} = \frac{1}{n} \sum_{i}^{n} D_i \tag{4.9}$$

It is important to keep track of this metric overtime, as it represents the amount of new information coming from the clusters at each global update step. For example, we expect the weight divergence to be higher in the initial rounds, in which the model parameters are changing rapidly, and to decrease gradually to zero as the model becomes more and more accurate.

[3]These values for mean and standard deviation of the normal distribution have been obtained experimentally, by halving the speed of computation values from model training on a Intel Core i7-6500U CPU.

Moreover, the weight divergence can also inform us on the degree of stability of the learning process. For example, if the data distribution is non-IID, we will see higher values, as updates tend to diverge more given that they are based on only some classes of the complete dataset. On the other hand, we can say that lower divergence values will represent training processes in which the updates are more in agreement with each other, like in the IID data case.

In conclusion, by keeping track of this metric, together with the model accuracy, we can assess the impact of the simulation settings (i.e. learning algorithm and client mobility rate) on the quality of the learning process.

4.1.4.D Status Logging

In order to have the simulator output available for processing even if a simulation fails at any point, we update a log file at the end of each round by appending the the metrics described above. The log files use a JSON format, and the results are arranged as a list of dictionaries, each containing the following fields.

- `round`
- `train_accuracy`
- `test_accuracy`
- `latency`
- `weight_divergence`

Note that the train accuracy, the test accuracy, and the weight divergence are only computed in the rounds where a inter-cluster global update happens. Hence, their values will be the same for rounds between two global updates.

The name of the log file is constructed by appending the number of the experiment that is being run to the name specified by the `log_file` parameter in the configuration. As an example, if the name parameter is `test_log`, the log files will be named `test_log-0.json`, `test_log-1.json`, `test_log-2.json`, and so on.

More information about the logging procedure can be found in Chapter 5.

4.1.5 Post-processing of Results

After all experiments are run for the same configuration file, we proceed to aggregate the results into one single JSON file that contains the combined metrics, as well as the configuration dictionary. This is done so that all the information relative to a simulation are kept in one single file, therefore making it easier to analyse the results.

The result file is structured as follows.

- `config` : [dictionary]
  This field will contain a copy of the configuration dictionary. Its structure has been previously described in §4.1.1.
- `results` : [dictionary]
  This field will hold the aggregated simulation results for the different experiments.
  - `train_accuracy` : [list]
    Average training set accuracy computed across experiments, one entry per round.
  - `test_accuracy` : [list]
    Average test set accuracy computed across experiments, one entry per round.
  - `latency` : [list]
    Average round latency computed across experiments, one entry per round.
  - `latency_median` : [float]
    Median round latency of all rounds of all experiments. This value is much more robust with respect to outliers, compared to the values in `latency`.
  - `weight_divergence` : [list]
    Average computed across experiments of the average weight divergence, one entry per round.

After the combined results are produced, the log files for the individual experiments are deleted.

4.2 Testbench

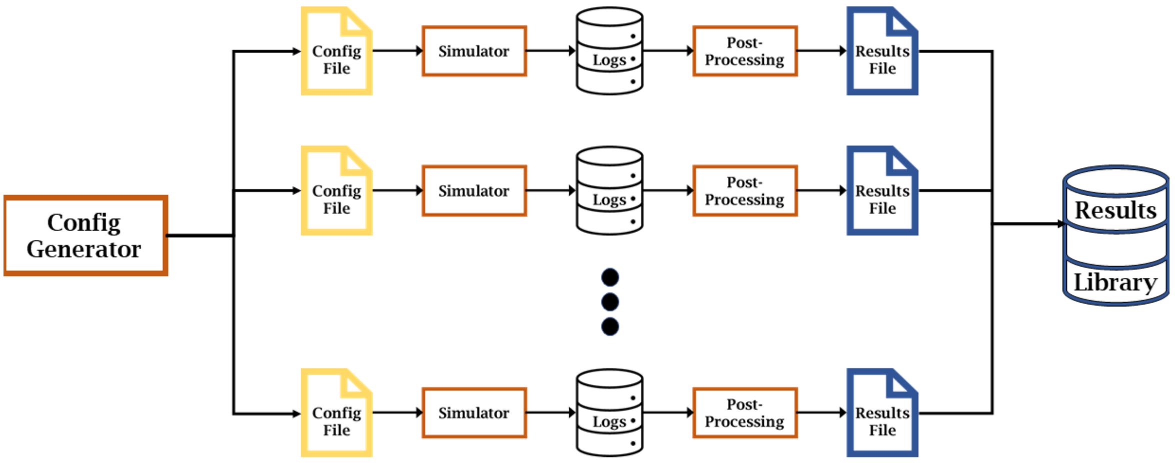


Figure 4.2: System diagram of the testbench.

The testbench is the piece of software that we use to run experiments and gather results in a systematic way. Since we are using it to assess the effect that varying parameters has on the HFL system, the main objective that we had to take into consideration was that of being able to easily set the parameters set that we want to iterate through, and having the testbench execute them.

This program is built on top of the simulator and a library of functions that we implemented to allow for parallel execution of different tests. It takes as input the lists of values that we want to test for each parameter, it generates all the combinations[4], it creates the configuration JSON files for each experiment, and executes them.
More information about the parallel execution of tests is presented in §5.2.

The names of the result files are formed by adding a progressive number to a root string defined by the user. In this way, it is easier to filter and analyze results coming from the same group of tests.

[4]The testbench also allows to select only a random subset of the possible combinations. This is useful, for example, if we have a very large parameter space that we are trying to iterate over, but we are not specifically interested in every single results (e.g. if we are looking for a suitable learning rate for the clients to use).

4.3 Results Visualizer

Because of its intended use, the results visualizer is based on a Interactive Python Notebook (`.ipynb`) as we need it to be as interactive as possible.
It reads the results files, and we can filter them by group of tests, or by defining a set of configuration parameters values that we want to inspect. Then, we can plot the metrics stored in the selected results files, and compute the correlation matrix between the parameters that vary across the selected simulations.

Chapter 5

Implementation

In this chapter we will give details about some key elements in the software implementation of the systems described in Chapter 4.

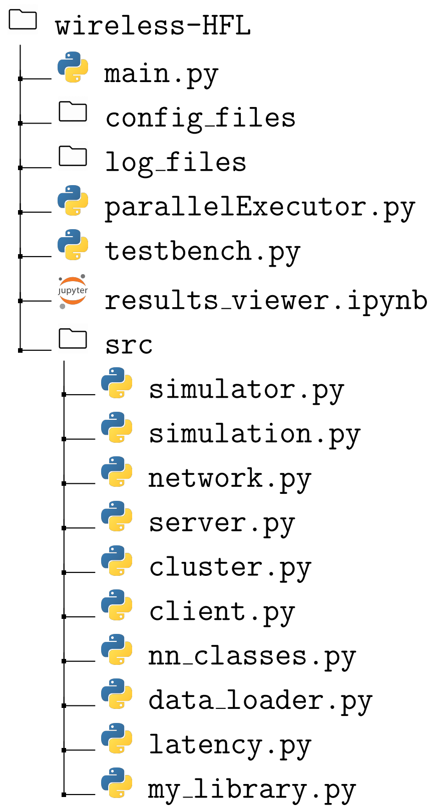


Figure 5.1: Directory structure for the project.

In Fig. 5.1 we show the directory structure of the project. This will be useful to look at when describing implementation in the following sections.

5.1 HFL Simulator

The simulator has been coded in Python, in order to take advantage of the PyTorch[1] library. PyTorch is used for the model training at client-level and for easily accessing and manipulating multiple open-source benchmark datasets (MNIST, CIFAR-10).

5.1.1 Class Model

From an Object-Oriented Programming (OOP) perspective the system has been implemented with the class model drawn in Table 5.1.
The `Simulator` itself runs a list of `Simulations`. This design choice was made to allow for easily linking the results of different experiments for the same configuration.
Each `Simulation` controls a `Network` which is composed by a MBS (a `Server`-type object) and a list of `Clusters`.

Each `Cluster` contains one SBS (again, a `Server` object) and a list of `Clients`.
Some of the methods are common to most of the classes, as they implement the main functionalities described in Chapter 4.

In addition to the Python files that contain the class definitions, we have grouped the rest of the simulator logic in four other Python files (as shownFig. 5.1), grouping the functions accordingly to their area of action:

- `nn_classes.py` contains the definitions of the neural network architectures;
- `dataloader.py` contains the functions for generating the split dataset (IID, non-IID, or spatially correlated non-IID) to assign to the clients;
- `latency.py` presents all the functions related to the scheduling of clients on the communication channel and the computation of latency;

[1]`https://pytorch.org/`

CLASS	ATTRIBUTES	METHODS
`Simulator`	- simulations : `list` - config_files : `list` - log_files : `list`	- **start**()
`Simulation`	- id : `int` - config : `dict` - log_file : `str` - network : `Network` - round_count : `int` - train_accuracy : `list` - test_accuracy : `list` - latency : `list`	- **configure**() - **start**() - **log**()
`Network`	- id : `int` - config : `dict` - round_count : `int` - mbs : `Server` - clusters : `numpy.Array` - clusters_grid_shape : `tuple` - train_set : `DataLoader` - test_set : `DataLoader` - weight_divergence : `float`	- **learn**() - **log**() - **move_clients**() - **evaluate**() - **evaluate_train**()
`Server`	- id : `int` - config : `dict` - model : *`torch model`* - [control_variate : `list`]	- **get_weights**() - **set_weights**(weights) - **set_average_model**(clients, selected_idx, clusters) - **download_model**(recipients)
`Cluster`	- id : `int` - config : `dict` - round_count : `int` - sbs : `Server` - clients : `list` - n_update_participants : `int` - weight_divergence : `float`	- **get_weights**() - **set_weights**(weights) - **learn**() - **learn_sgd**(l_rate, local_iter) - **learn_scaffold**(l_rate, local_iter) - **get_round_latency**(selected_clients)
`Client`	- id : `int` - config : `dict` - model : *`torch model`* - train_data : `DataLoader` - [control_variate : `list`] - [c_variate_update : `list`]	- **get_weights**() - **set_weights**(weights) - **learn**(l_rate, local_iter, server_control_variate) - **learn_sgd**(l_rate, local_iter) - **learn_scaffold**(l_rate, local_iter, server_control_variate) - **update_control_variate**(l_rate, local_iter, server_model, server_control_variate)

Table 5.1: Class model outline. (The attributes enclosed in square brackets are initialize and used only if the learning algorithm used is SCAFFOLD)

- `my_library.py` contains functions related to the logging of the simulation state and the post-processing and production of results files, as well as other miscellaneous functions.

5.1.2 Datasets and Neural Networks

The most important elements in a machine learning project are the datasets and the models used. In this subsection we will describe the datasets and the architectures of the models supported by the simulator, from a software implementation point of view.

5.1.2.A Datasets

The two datasets currently supported by the simulator are the MNIST[2] and the CIFAR-10[3]. Both datasets include 50'000 images in the training set, 10'000 in the test set, and labelled in 10 equally represented classes. The MNIST images are black and white and have a 28x28x1 shape, whilst the CIFAR-10 images are coloured and of shape 32x32x3. We normalize the single-channel MNIST data by using mean 0.1307 and standard deviation 0.3081; for CIFAR-10 we perform channel-wise normalization with means (0.4914, 0.4822, 0.4465) and standard deviations (0.247, 0.243, 0.261). These values were chosen as they represent the mean and standard deviation of each channel.
See §4.1.2 for more information about how the data is distributed to the clients.

5.1.2.B Neural Networks

MNIST CNN. To experiment on the MNIST dataset, we have used a simple convolutional neural network with 56'900 trainable parameters. The architecture (shown in Fig. 5.2) is composed by:

- 5x5 convolutional layer, with stride 1 and no padding, followed by ReLu as activation function;
- 2x2 maxPool layer, with stride 2 and no padding;
- 5x5 convolutional layer, with stride 1 and no padding, followed by ReLu;

[2] `http://yann.lecun.com/exdb/mnist/`
[3] `https://www.cs.toronto.edu/~kriz/cifar.html`

- 2x2 maxPool layer, with stride 2 and no padding;
- a flattening layer;
- a fully connected layer with 100 neurons, with ReLu activation
- a fully connected output layer with 10 neurons.[4]

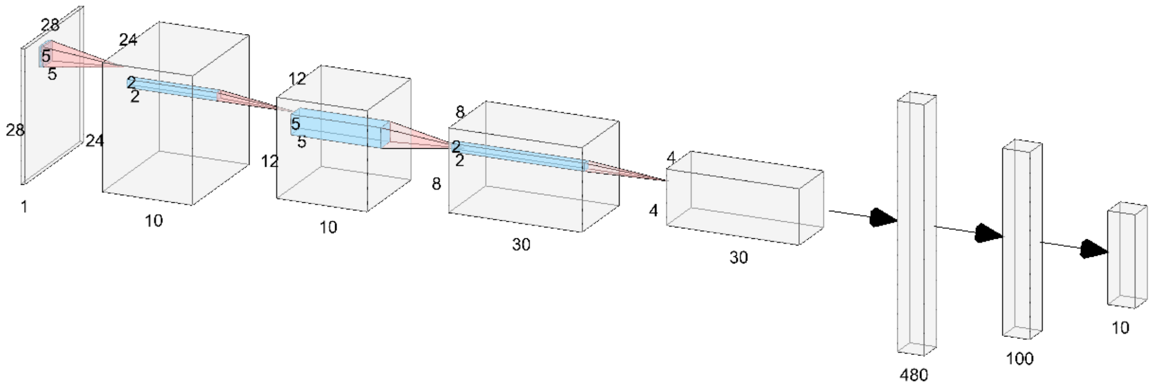


Figure 5.2: Architecture of the CNN used for MNIST.

MobileNetV2. The MobileNet architectures family [12, 34, 13] is the result of the work of Google researchers in developing neural networks architectures that are specifically tailored for mobile and resource constrained environments. The main perk of these models is that they achieve close to state of the art accuracy scores, whilst significantly decreasing the number of operations and memory needed to train and use these models.
We have chosen to use this family of models for our experiments on CIFAR-10 because they represent architectures that can be used on mobile and IoT devices in a federated learning network, without requiring too high computational capabilities. Specifically, we have decided to use the MobileNetV2 [34], over the newer version MobileNetV3-Large [13], as they achieve similar accuracy values but the V2 is significantly smaller in size than the V3-Large since they have 3.5M and 5.5M trainable parameters respectively. The smaller parameter space was preferred in consideration of the necessity for the clients and servers to send the

[4]Note that in most implementations of multiclass classification models, a softmax activation function is applied at the output layer. However, not doing this does not impact the cross-entropy loss computation, and we perform less operations this way. Yet, it is necessary to apply softmax to the output at the inference stage if we want the normalized output probabilities vector; if we are only interested in the top-1 accuracy it is sufficient to do argmax of the output vector.

weight updates over a wireless channel. [5]

What makes training and using the MobileNetV2 particularly computationally efficient is the use of the inverted residual layer with linear bottleneck, shown in Fig. 5.3.

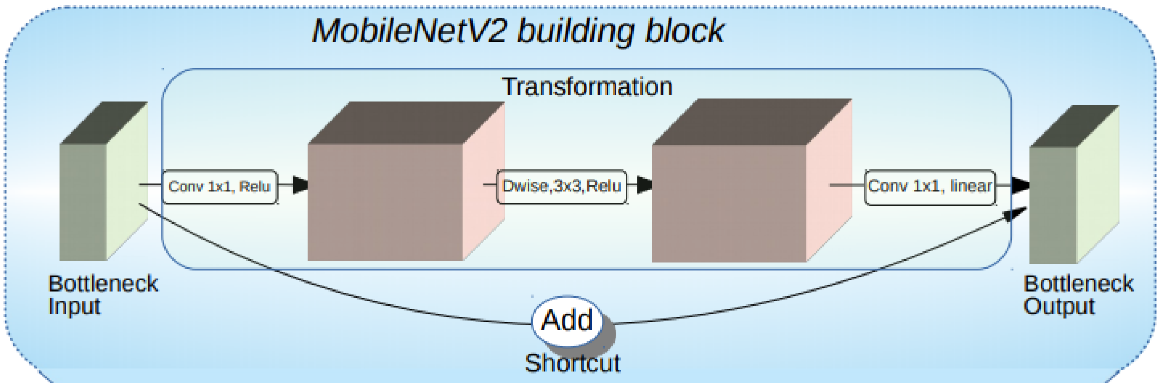


Figure 5.3: Inverted residual layer with linear bottleneck. The fundamental building block of the MobileNetV2 architecture. (Image taken from [33]).

This module takes as an input a low-dimensional compressed representation which is first expanded to high dimension and filtered with a lightweight depth-wise convolution. Features are subsequently projected back to a low-dimensional representation with a linear convolution. Therefore, this convolutional module is particularly suitable for mobile devices or embedded systems, because it reduces the amount of memory needed and the number of main memory accesses during inference as it never fully generates large intermediate tensors.

Since the MobileNetV2 outputs a tensor of shape (1000,1) but the CIFAR-10 data has only 10 classes, we have added an extra fully connected layer of 10 neurons at the output to make it of shape (10,1).

5.1.3 Learning Algorithms

As shown in Table 5.1, the implementation of the learning process is contained in various functions across the classes. Let us now describe the learning phase, first by analysing the procedure at the `Network` level,

[5] The MobileNetV3-Small variant of the architecture, was not preferred as, despite having a smaller parameter space (2.5M trainable parameters), achieves worse accuracy values when compared to the V2.

then at the `Cluster` level, and finally at the `Client` level.

5.1.3.A Network-level Learning

In each simulation round, we start the learning phase by calling the `Network.learn()` function, which operates as follows.

1. For each cluster we will run `Cluster.learn()`.
2. If the current round involves a global update[6]:
   (a) Update the MBS model by averaging the models held by the SBSs. However, since clients are allowed to move around clusters, SBSs will learn from a different number of clients[7] leading to an imbalance in the quality of the updates. Then, we need to account for this difference when we aggregate the cluster models. This is performed by calling on the MBS the function `Server.set_average_model` with the clusters as argument and the `clusters` parameter set to true. This will average the model parameters received from the SBSs by using as weights of the average the number of clients that have participated to the transmitted update in each cluster.
   (b) Compute the average weight divergence between the SBSs models and the new global model, as described in §4.1.4.C.
   (c) Copy the global model weights to the SBSs models, executing `Server.download_model(recipients)` on the clusters.
   (d) *Only if we are using SCAFFOLD*, generate a new global server control variate by averaging the SBSs control variates; then, distribute it to the SBSs.

5.1.3.B Cluster-level Learning

For what concerns the intra-cluster learning process, it is executed by running `Cluster.learn()`.
This function, depending on the learning algorithm used, whether it is SGD or SCAFFOLD, will run `Cluster.learn_sgd(l_rate,local_iter)`

[6]This is when the round number modulo the global update rate equals zero.

[7]This happens because in each cluster we select the same fraction of clients. Hence, if a SBS has access to a larger pool of clients, it will select more clients to take part in the update.

or `Cluster.learn_scaffold(l_rate,local_iter)`[8]. Although all clusters and clients have the configuration dictionary as attribute, and hence they know which learning rate and number of local iterations to use, the `l_rate` and `local_iter` parameters are used to specify their values in case they differ from the ones in the configuration. This happens when we are using a learning rate warmup technique (smaller LR in the first rounds) or Post-Local SGD (perform only one local iteration in the initial rounds).
To summarize, `Cluster.learn()` works as follows:

1. Compute `l_rate` and `local_iter` to use, if they differ from the configuration values.
2. Call `Cluster.learn_sgd` or `Cluster.learn_scaffold`, depending on the learning algorithm to use.

`Cluster.learn_sgd` and `Cluster.learn_scaffold` operate by following the regular algorithm of federated learning, although they differ slightly.

1. The clients that will take part in the round are selected.
2. For each selected client:
    (a) Copy the SBS model weights to the client model.
    (b) Compute the update on the client by calling the function `Client.learn` with the learning rate and number of local iterations as parameters.
    In `Cluster.learn_scaffold`, the server control variate is also passed as argument to `Client.learn`.
    (c) Only in `Cluster.learn_scaffold`, we update the client control variate by running `Client.update_control_variate`.
3. We update the SBS model by executing `Server.set_average_model` on the SBS.
4. Only in `Cluster.learn_scaffold`, we update the SBS control variate by adding the sum of the updates divided by the total number of clients in the cluster.

[8]This design choice of having a `Cluster.learn` function which calls `Cluster.learn_sgd` or `Cluster.learn_scaffold` was taken because it increases the modularity of the system. Indeed, if we want to add another learning algorithm, it is sufficient to add a new method to `Cluster` and make sure it can be called by `Cluster.learn`, without the need of editing code in the `Network` class.

Moreover, the simulator supports a `save_memory` mode in which `Client` objects to not keep their model state across rounds. The client model is initialized only if the client is chosen to participate in the round; the model object is then allocated, the learning process is executed, the update is sent to the SBS, and finally the client model is deleted.
This makes the memory footprint of the client models in the simulations as large as the size of the models used by one cluster in one round. This feature is particularly useful for running simulations in which we have a large number of clients, or the model architecture has a very large number of parameters. However, this memory-saving feature comes at the expense of performance, as it introduces the overhead of constantly allocating and deleting the models.
In the current version of the simulator, this mode is only enabled if the model used is the MobileNetV2. This can be easily changed by editing the `simulation.py` file.

5.1.3.C Client-level Learning

Similarly to the cluster learning implementation, also for the `Client` class we have decided to maintain the design of having a top-most `Client.learn` function that calls, depending on the learning algorithm to use, `Client.learn_sgd` or `Client.learn_scaffold`. The differences in the latter function are that we clip the gradient values to a maximum bound of 10^6 using `torch.nn.utils.clip_grad_value_`, and we apply the gradient correction before performing the gradient descent.

5.1.4 Latency Computation

The latency of each round is computed in the `Network.learn` function, by taking the maximum latency of each cluster. The latency value is being returned by each call of `Cluster.learn`, which uses the number of clients that participate in the round to estimate the round duration (downlink + uplink) for the cluster by executing the method `Cluster.get_round_latency`.
We have modelled each round execution as a synchronous process, in which the length of a round is given by the latency of the most straggling cluster. This is a limitation of the simulator, because in reality clusters need to wait for each other only when there is a global update step to be performed. Alternatively, a cluster can proceed to the subse-

quent round immediately upon finishing the current, thus saving time. We will address this issue and propose some improvements as further work in Chapter 9.

In the implementation of the communication channel described in §4.1.4.B, we have set the channel descriptive parameters as follows :

- SBS transmitting power: 6.3 W [43]
- Client transmitting power: 0.2 W [43]
- Noise power spectral density: $10^{-20.4}$ W
- Per cluster bandwidth: 20 MHz
- Cluster radius: 500 metres

These values can be edited in the `latency.py` file by modifying the constants at the top of the script.

5.2 Testbench

From a software implementation point of view, the most important element of the testbench to discuss is the parallel execution of the simulations.
This is achieved by using the `threading` Python library to execute each test inside a separate thread. However, if we are planning to run a lot of simulations, it is inefficient to start them all together, as we would spend a lot of CPU time in switching between threads, or the threads will fail to run because of race conditions on memory resources (RAM, or VRAM since we will be using GPUs [9]).

Because of these reasons, in the testbench we need to set the maximum number of tests that are allowed to run concurrently. The testbench will generate the configuration files and will start to execute them following the scheduling of Algorithm 4. The main feature of this algorithm is the fact that we keep retrying to run a simulation if it fails. Simulations fail for insufficient resources, such as lack of memory (RAM

[9] In our case we used the `bubota` machine from the IPC Lab servers, which features 2 Nvidia GeForce RTX 2080 Ti with 11GB of VRAM each.

and/or VRAM). This usually happen if we are using a shared machine and somebody else is running jobs. This way, we can submit the set of tests to run knowing that they will be executed as soon as enough resources are freed. Moreover, we have the possibility of dividing the load of the simulations across multiple GPUs.

Algorithm 4 Simulations Scheduling

```
1: Initialize simulation list: $\mathcal{S} = [s_1, s_2, \ldots, s_n]$
2: Initialize devices list: $\mathcal{D} = [GPU_1, GPU_2, \ldots, GPU_m]$
3: Initialize $t_{wait}$                      ▷ Time to wait before spawning a thread
4: Initialize $t_{retry}$       ▷ Maximum time to wait before retrying a failed simulation
5: Initialize $T_{max}$                      ▷ Maximum number of concurrent threads
6: Initialize $T_{active} \leftarrow 0$

7: for $s \in \mathcal{S}$ do
8:     while $T_{active} = T_{max}$ do                 ▷ Wait for a slot to be available
9:         Wait $t_{wait}$
10:    end while
11:    Spawn RunSimulation$(s, \mathcal{D})$ in a new thread      ▷ (Non-blocking command)
12:    $T_{active} \leftarrow T_{active} + 1$
13: end for

14: RunSimulation$(s, \mathcal{D})$:
15: Initialize $returnCode \leftarrow -1$
16: while $returnCode \neq 0$ do
17:     Randomly sample $d \in \mathcal{D}$
18:     $reuturnCode \leftarrow$run$(s, d)$             ▷ Execute simulation $s$ on device $d$
19:     if $returnCode \neq 0$ then
20:         Wait $t \sim \mathbf{U}(0, t_{retry})$             ▷ Randomly sample waiting time
21:     end if
22: end while
23: $T_{active} \leftarrow T_{active} - 1$
```

The logic that handles the parallel execution can be found in `parallelExecutor.py`. This module has been designed with a high level of abstraction, so that it can be reused for other projects.

Chapter 6

Testing

Before being able to execute experiments, as well as along the development, we tested the systems to ensure that they operate correctly.

For what concern the functional correctness, we have included in the simulator multiple debug print statements which output to the terminal information about the state and operations executed by the simulator. This debugging feature can be enabled/disabled from the configuration file, and it is possible to add new debugging messages at any point in the simulator code.

On the other hand, to ensure that the results that we get from the experiments are actually correct,

- we have executed simulations with only 1 cluster and 1 client, and compared the learning curves with the results obtained when simply training the model without any federated learning structure;
- we have run experiments on FL and HFL settings that have been used in research papers, and we have compared the results.

Chapter 7

Results

The aim of this chapter is to present the simulations run in order to understand the impact of the configuration parameters on the learning process, as described in Chapter 1.

In the following subsections, we will analyze which experiments have been tested and under which metrics the simulations results have been compared.
In §7.1 we will introduce the set of tests executed. In §7.2 we will define the key metrics that we keep track of during simulations.

Afterwards, the main results will be presented in 3 sections, where each of the sections represent the main aspect that we will take into consideration in our analysis.
In the first (§7.3) we focus on client mobility and its impact on the learning process when data is IID, non-IID, and spatially correlated non-IID.
In the second and third we analyze how different learning algorithms perform with non-IID data (§7.4) and with spatially correlated non-IID data (§7.5).
In the We will present the plots for some of the results obtained, representing the test accuracy of the global model and the weight divergence, as a function of the rounds elapsed.
Moreover, we will include tables with the correlation coefficients between the simulations parameters and the performance metrics described in §7.2.

7.1 Outline of Experiments

Since one application of models trained with FL can be the automated tagging of a photo library, the aim of the experiments is to evaluate the algorithms on a benchmark image classification task (as in [29, 1]) of training a CNN (§5.1.2.B) on the MNIST dataset.

For this model, we tested IID, non-IID [14], and spatially correlated non-IID data distributions among clients. Moreover, we experimented of the impact that the learning algorithm has on the training process, by using

- *Minibatch SGD* (§2.1), with a batch size of 32 samples;
- *Large minibatch SGD* (§2.1.1), with batch sizes of 64 and 128 samples, with and without using a linear learning rate warmup technique;
- *Local SGD* (§2.1.2), by making the clients perform multiple (2 or 4) local updates in the same round;
- *Post-local SGD* (§2.1.2), by delaying the multiple local updates by a user-defined number of epochs.
- *Minibatch SGD with SCAFFOLD* (§2.2.4, [20]), by applying drift correction to the client gradients before performing gradient descent during the local updates.

Note that, in order to reduce the parameters space for the simulations, we have decided to set a constant value for some parameters. These are

- 250 clients;
- 25 clusters;
- 30% of clients are selected for each round;
- 0.01 client learning rate;
- 250 simulated rounds[1];
- the client mobility rate values 0%, 10%, 25%, 50%;
- the global update rates 1, 3, 5, 7;

[1]We extended the simulation length some configurations which did not reach the target test accuracy of 95% in 250 rounds.

- 10 simulations for each configuration.

In addition to these results, in the last section (§7.6) we have reported some result of experiments on a more complex image classification task, performing fine-tuning for the CIFAR-10 dataset of a MobileNetV2 (§5.1.2.B) with weights pretrained on ImageNet[5]. In these simulations, we tested the minibatch SGD algorithm on IID, non-IID, and spatially correlated non-IID data.
We used the same constant parameters of the MNIST experiments, with the exception that we run simulations for 350 rounds.

7.2 Key Performance Indicators

Since the main objective of this project is to analyze and optimize learning algorithms in a HFL setting with moving clients and non-IID data distribution, it is necessary to define a set of key performance indicators (KPIs) to be used to evaluate and compare simulations.
The KPIs that we will use are those found in most papers about Federated Learning [1, 20, 27, 29, 31] and have already been described in Chapter 4:

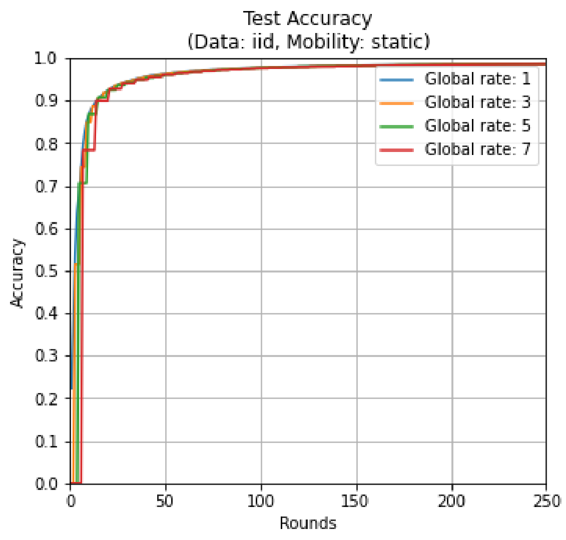
- **Number of learning rounds** needed to achieve a target model accuracy (test for speed of convergence).
- **Number and size of messages** sent around in the network.
  This this is translated into the **latency** analysis of the time needed to send those messages with the modelled communication channel described in §4.1.4.B.
- **Weight divergence** The average weight divergence between the clusters models and the global model, as described in §4.1.4.C.

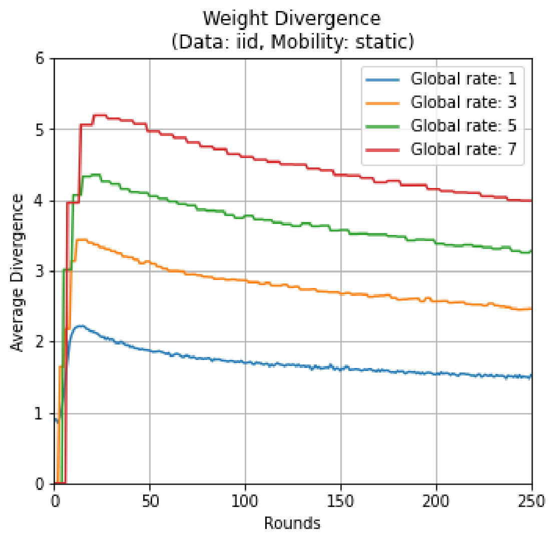
In the following sections below we will include tables with the correlation matrices between the simulation parameters and the following metrics: ***Rounds @ 0.95*** represents the number of rounds needed to reach a test accuracy of 0.95, ***Average Weight Divergence*** is the mean weight divergence value across all simulation rounds, while the ***Median Latency*** is computed across all rounds of all the experiments for the same configuration.

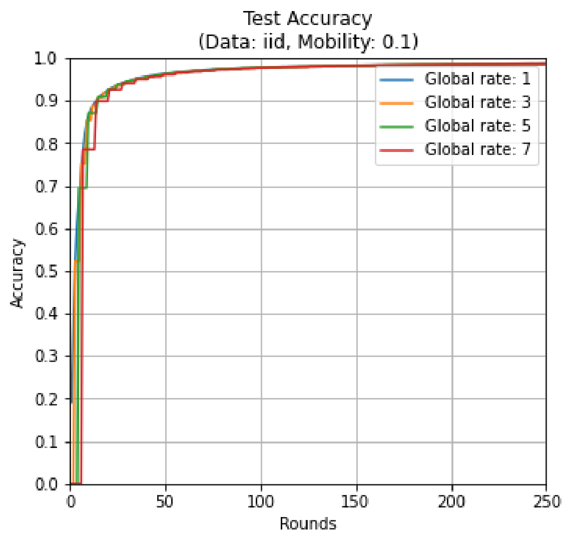
7.3 Client Mobility

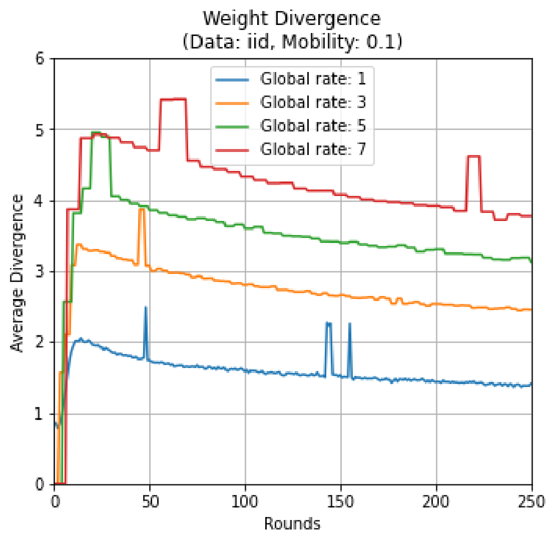
7.3.1 IID & Non-IID

In Fig. 7.1, we want to show the different effect that varying the global update rate has when we have a different client mobility rate, when data is distributed across clients in an IID fashion.









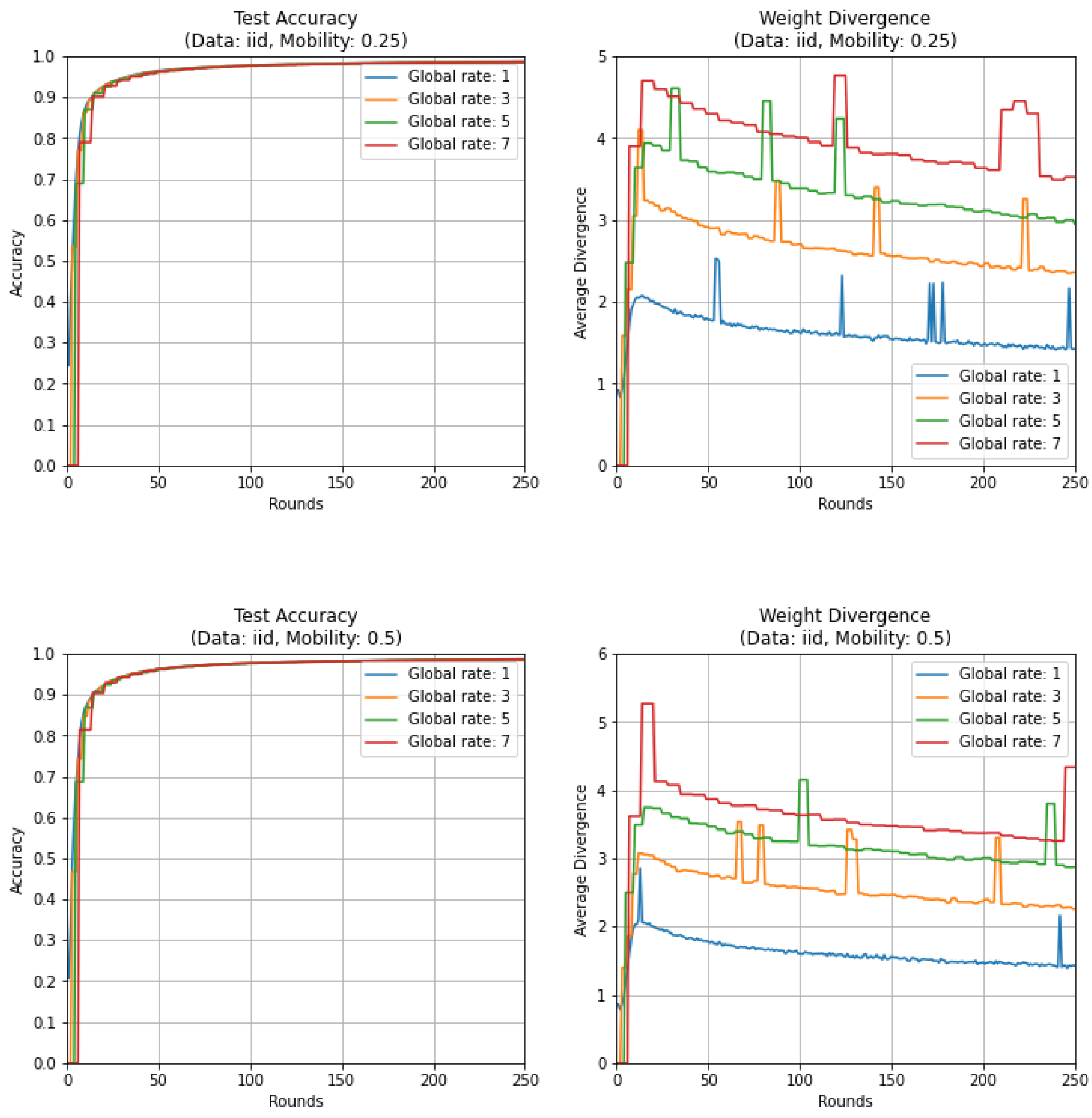
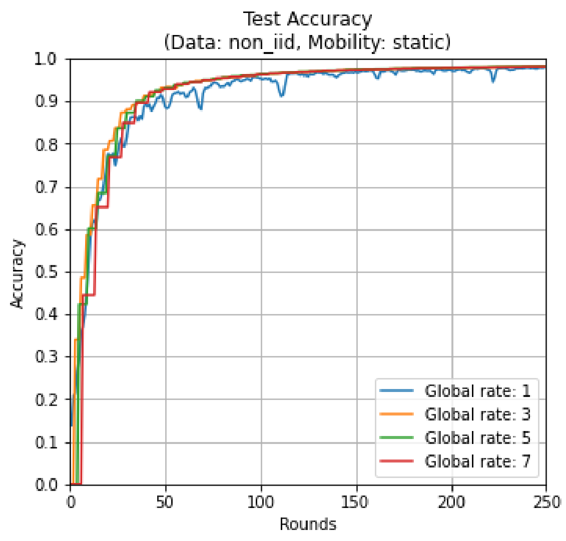
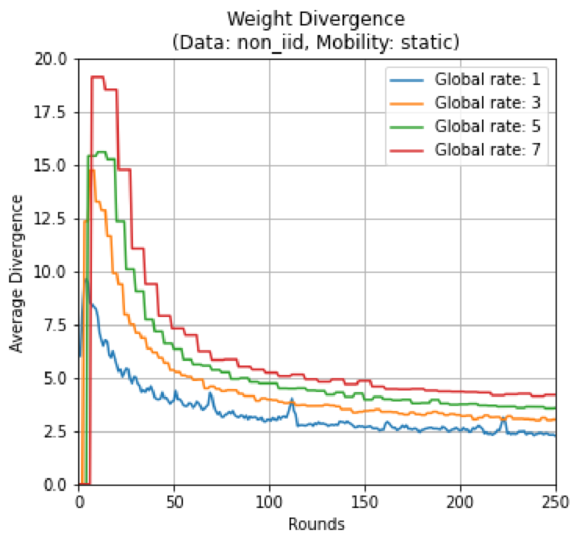
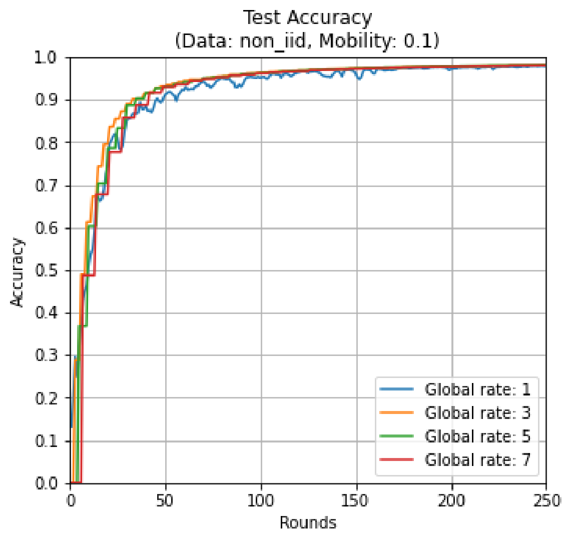


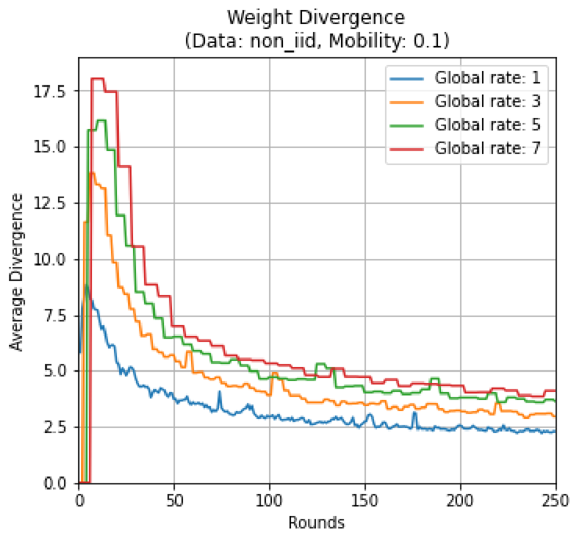
Figure 7.1: Effect of client mobility with IID data.

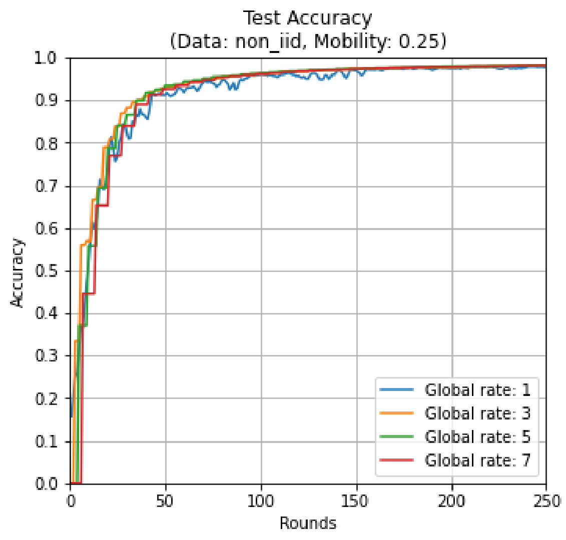
Below, in Fig. 7.2 we show the same plots but for the non-IID dataset distribution.

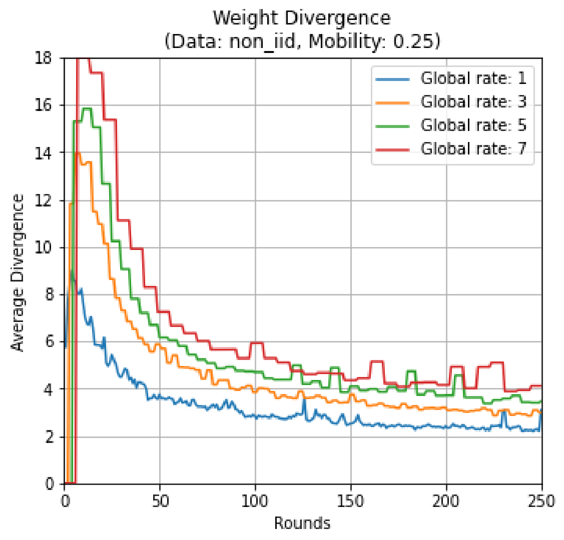

Test Accuracy
(Data: non_iid, Mobility: static)
Accuracy
Rounds
Global rate: 1
Global rate: 3
Global rate: 5
Global rate: 7


Weight Divergence
(Data: non_iid, Mobility: static)
Average Divergence
Rounds
Global rate: 1
Global rate: 3
Global rate: 5
Global rate: 7


Test Accuracy
(Data: non_iid, Mobility: 0.1)
Accuracy
Rounds
Global rate: 1
Global rate: 3
Global rate: 5
Global rate: 7


Weight Divergence
(Data: non_iid, Mobility: 0.1)
Average Divergence
Rounds
Global rate: 1
Global rate: 3
Global rate: 5
Global rate: 7


Test Accuracy
(Data: non_iid, Mobility: 0.25)
Accuracy
Rounds
Global rate: 1
Global rate: 3
Global rate: 5
Global rate: 7


Weight Divergence
(Data: non_iid, Mobility: 0.25)
Average Divergence
Rounds
Global rate: 1
Global rate: 3
Global rate: 5
Global rate: 7

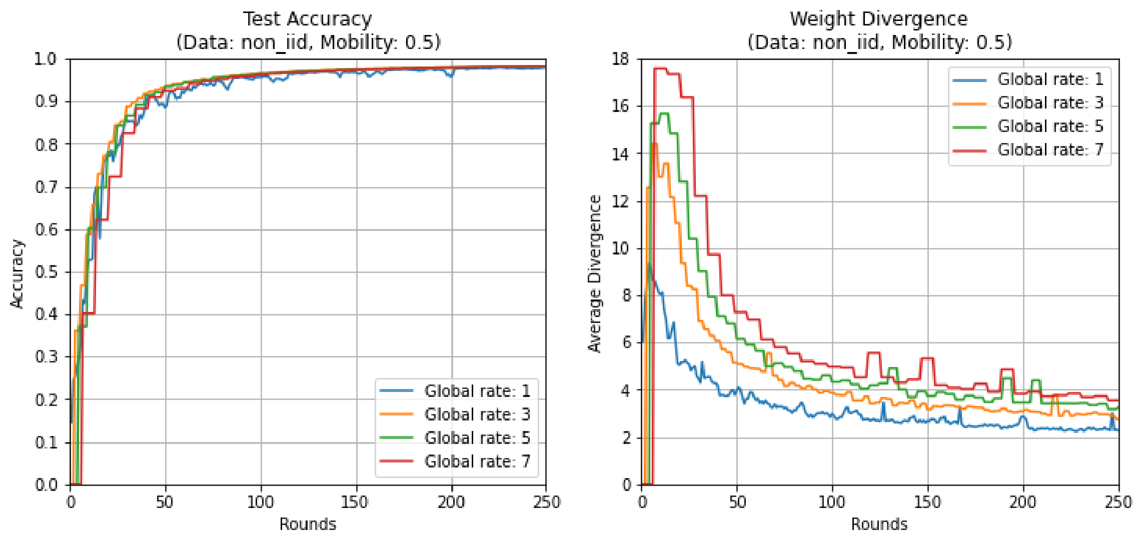


Figure 7.2: Effect of client mobility with non-IID data.

From the results we can see that increasing the mobility rate has a positive effect on the learning process as it helps reducing the weight divergence of the updates. This enhancement becomes clearer when we increase the global update rate, although a higher global update rate leads to higher update divergence.
In Table 7.1 are presented the correlation coefficients between the two varying parameters (mobility rate and global update rate) and the main metrics that we analyze.

IID	**Rounds @ 0.95**	**Average Weight Divergence**	**Median Latency**
Mobility rate	-0.12	-0.16	0.62
Global update rate	0.41	0.97	-0.01

Non-IID	**Rounds @ 0.95**	**Average Weight Divergence**	**Median Latency**
Mobility rate	-0.03	-0.04	0.63
Global update rate	-0.52	0.99	0.22

Table 7.1: Correlations for Minibatch SGD.

From the tables we can note the following.

- The convergence speed is correlated with the global update rate. However, we can see that increasing the global update rate increases the convergences speed only in the non-IID case [2], while it has the opposite effect in the IID case. The contribution of the mobility rate is minimal, as the values are almost uncorrelated especially in the non-IID case.

- The weight divergence is very strongly correlated with the global update rate. This is expected because the more intra-clusters rounds are in between two global updates, the more the local updates will differ.
- Variations in latency are mostly driven by the mobility rate.
  This is because of how we have decided to measure the round latency. Since we have modelled each round execution as a synchronous process, the length of a round is given by the largest round latency (downlink + uplink) of a cluster. Hence, by increasing the mobility rate, we increase the probability that a cluster has a large amount of clients, which drives the total round duration up.

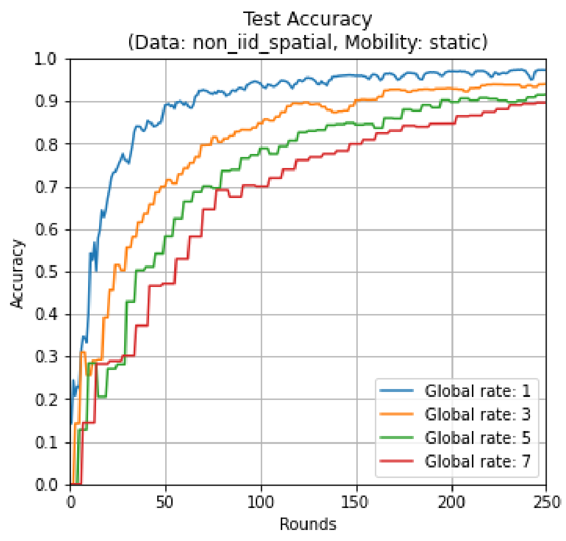
7.3.2 Spatially correlated Non-IID

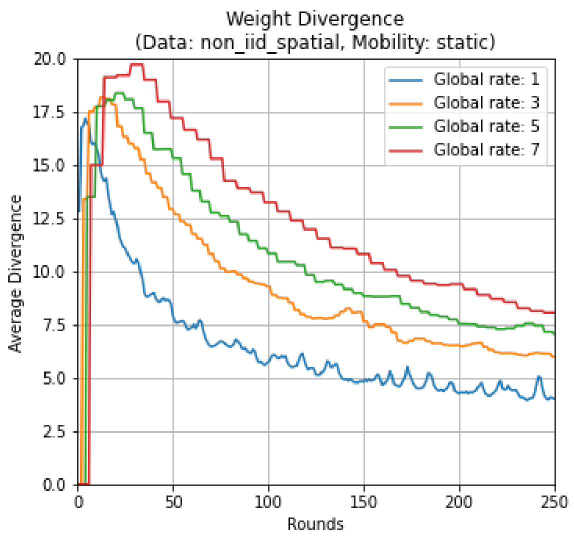
The main differences are seen in the case where data is distributed in a spatially correlated non-IID way. The class distribution in the tests performed is presented in Table 7.2. The results are shown in Fig. 7.3.

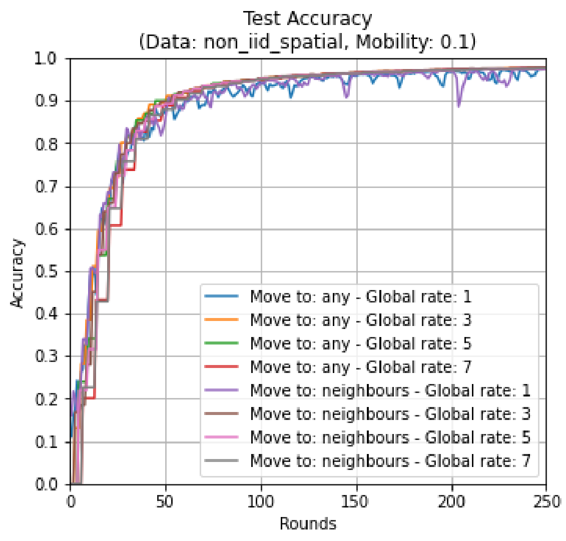
0	0	0,1	1	1
3	3	3,2	2	2
4	4	4,5	5	5
7	7	7,6	6	6
8	8	8,9	9	9

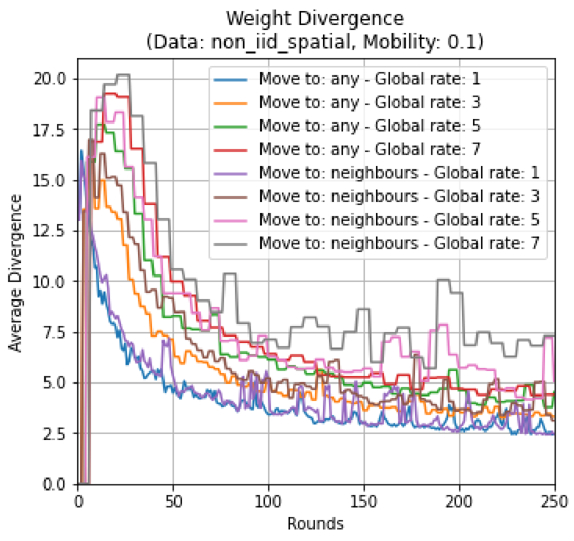
Table 7.2: Distribution of 10 equally represented classes for the assignment of spatially correlated non-IID data, for 25 clusters arranged in a 5x5 grid.

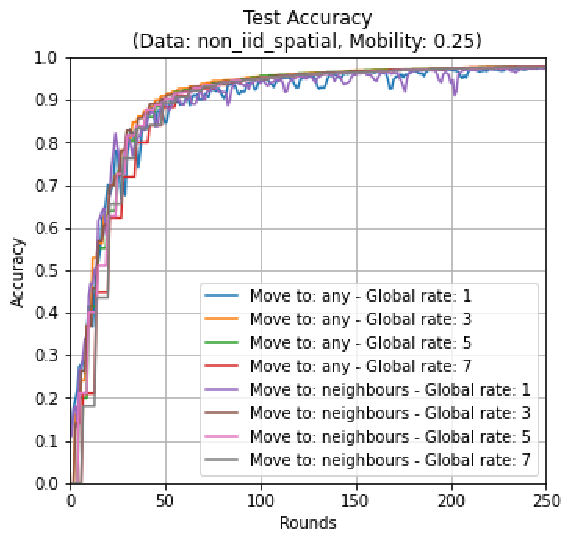
[2]Since the lower the *Rounds @0.95* value, the higher is the convergence speed, note that a negative correlation informs us that increasing the parameter improves the convergence speed.

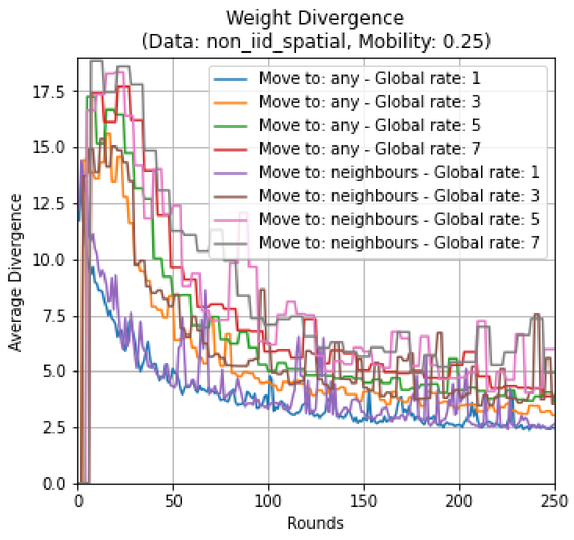

Test Accuracy
(Data: non_iid_spatial, Mobility: static)
Accuracy
Rounds
Global rate: 1
Global rate: 3
Global rate: 5
Global rate: 7


Weight Divergence
(Data: non_iid_spatial, Mobility: static)
Average Divergence
Rounds
Global rate: 1
Global rate: 3
Global rate: 5
Global rate: 7


Test Accuracy
(Data: non_iid_spatial, Mobility: 0.1)
Accuracy
Rounds
Move to: any - Global rate: 1
Move to: any - Global rate: 3
Move to: any - Global rate: 5
Move to: any - Global rate: 7
Move to: neighbours - Global rate: 1
Move to: neighbours - Global rate: 3
Move to: neighbours - Global rate: 5
Move to: neighbours - Global rate: 7


Weight Divergence
(Data: non_iid_spatial, Mobility: 0.1)
Average Divergence
Rounds
Move to: any - Global rate: 1
Move to: any - Global rate: 3
Move to: any - Global rate: 5
Move to: any - Global rate: 7
Move to: neighbours - Global rate: 1
Move to: neighbours - Global rate: 3
Move to: neighbours - Global rate: 5
Move to: neighbours - Global rate: 7


Test Accuracy
(Data: non_iid_spatial, Mobility: 0.25)
Accuracy
Rounds
Move to: any - Global rate: 1
Move to: any - Global rate: 3
Move to: any - Global rate: 5
Move to: any - Global rate: 7
Move to: neighbours - Global rate: 1
Move to: neighbours - Global rate: 3
Move to: neighbours - Global rate: 5
Move to: neighbours - Global rate: 7


Weight Divergence
(Data: non_iid_spatial, Mobility: 0.25)
Average Divergence
Rounds
Move to: any - Global rate: 1
Move to: any - Global rate: 3
Move to: any - Global rate: 5
Move to: any - Global rate: 7
Move to: neighbours - Global rate: 1
Move to: neighbours - Global rate: 3
Move to: neighbours - Global rate: 5
Move to: neighbours - Global rate: 7

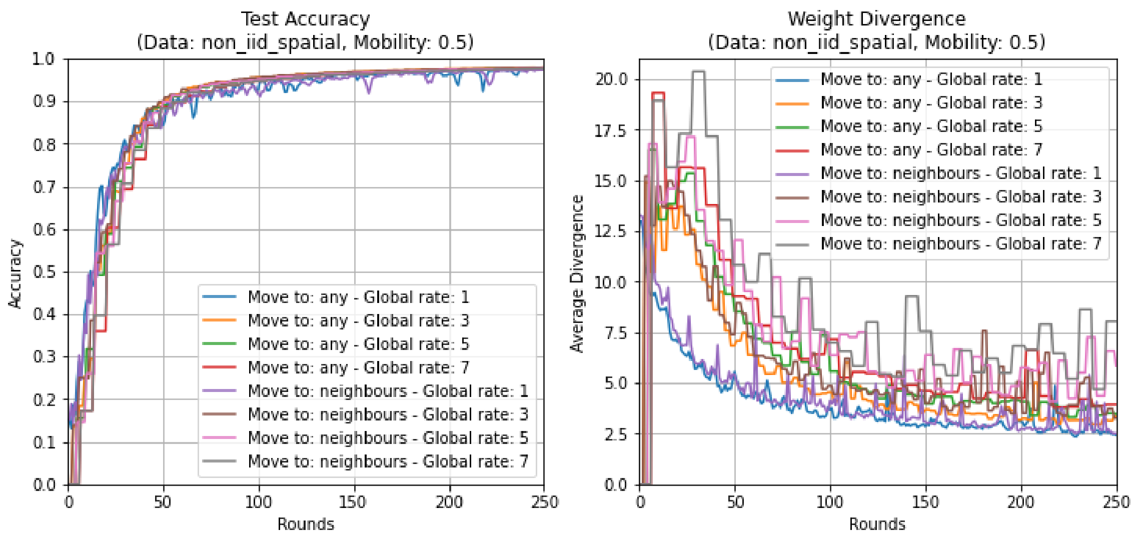


Figure 7.3: Effect of client mobility with spatially correlated non-IID data.

Spatially Correlated Non-IID	**Rounds @ 0.95**	**Average Weight Divergence**	**Median Latency**
Mobility rate	-0.45	-0.47	0.64
Move to neighbours	0.00	0.16	0.13
Global update rate	0.10	0.70	0.03

Table 7.3: Correlations for Minibatch SGD on spatially correlated non-IID data.

The results show that, when data is spatially correlated, the speed of convergence increases drastically if clients are allowed to move among clusters. Let us analyze why by inspecting each of the three data distributions seen so far.

- When data is **IID**, a cluster always sees samples from all the classes, no matter which clients it chooses to participate in the rounds. The client mobility helps by letting the cluster learn from different data samples, but not from a different distribution because all client-held data is IID.

- On the other hand, when data is **non-IID** across clients only, but not at a cluster level, a cluster is able to learn from all the classes by selecting different clients across rounds. Allowing client mobility does not benefit much the learning process because each cluster already has access to a data pool which contains all the classes and is (almost) IID with respect to the data pools of other clusters.

- Lastly, when data is **non-IID and spatially correlated**, and clients are static, a cluster has only access to data representing some of the classes (or even only one class); it will then generate updates that are strongly drifting from the optimal direction. In this context, each cluster needs to use the global update step to gain knowledge on the other classes, hence the more frequent the global update is, the faster the model convergence (as shown in the first plot of Fig. 7.3).
  In this case, moving clients can largely benefit the learning process, because as rounds elapse the data pool available to each cluster will become more class-balanced, hence yielding more stable updates.
  Moreover, we note that introducing client mobility makes a large difference on the speed of convergence, but the improvement does not scale when we further increase the mobility rate.

In Table 7.3 and in the plots of Fig. 7.3 we also see that the weight divergence is reduced by increasing the mobility rate, and that, in accordance to what was discussed in §7.3.1, increasing the number of rounds between global updates makes the cluster-transmitted updates diverge more.
Another interesting insight is that restricting client mobility only to neighbouring clusters is uncorrelated with the speed of convergence, and has a minor influence in increasing the updates divergence.

Lastly, for what concerns the negative impact of client mobility on latency, we can make the same observations that we had in §7.3.1 for the IID and non-IID cases.

7.4 Non-IID Data

In this section we will discuss the impact of using different learning algorithms on the generalization performance of the model and on the stability of the updates submitted by the clusters. We will distribute the data in a non-IID way, in which each client has samples drawn from two classes.

7.4.1 Minibatch SGD

7.4.1.A Large Minibatch SGD.

In Table 7.4 we present the results for experiments with the Large Minibatch SGD (§2.1.1, [9]) learning algorithm. For these experiments we have also tested how the learning rate warmup technique influences the learning process.
Note that the results with no learning rate warmup and a batch size of 32 are those that we refer to as standard minibatch SGD.

The results are shown for the case in which clients move with a probability of 25%. We have experimented as well with static clients and mobility rates of 10%, and 50% but, as shown in Table 7.5 and in the mobility results highlighted above in the mobility section, we have seen that there is not a significant impact of the rate of mobility on the results for this kind of non-IID data distribution.

Global Update Rate	Batch Size	Learning Rate Warmup	Rounds @ 0.95 Accuracy	Average Weight Divergence
1	32	No	**93**	3.3
		Yes	101	3.2
	64	No	101	3.0
		Yes	111	2.8
	128	No	116	2.6
		Yes	130	**2.5**
3	32	No	**75**	4.7
		Yes	78	4.6
	64	No	93	3.9
		Yes	99	4.0
	128	No	117	**3.3**
		Yes	129	3.4
5	32	No	**70**	5.5
		Yes	80	5.5
	64	No	95	4.5
		Yes	105	4.5
	128	No	120	3.9
		Yes	125	**3.8**
7	32	No	**77**	6.2
		Yes	84	6.1
	64	No	98	5.2
		Yes	105	5.1
	128	No	119	**4.3**
		Yes	133	4.4

Table 7.4: Results for Large Minibatch SGD on MNIST; mobility rate 25%. In bold are marked the best results in each *global update rate* group.

Non-IID	Rounds @ 0.95 Accuracy	Average Weight Divergence	Median Latency
Mobility rate	-0.02	-0.01	0.91
Global update rate	-0.08	0.81	0.22
Batch size	0.90	-0.52	-0.04
Learning rate warmup	0.25	-0.03	-0.01

Table 7.5: Correlations for Large Minibatch SGD on MNIST.

In Fig. 7.4 we show an example of the test accuracy and update divergence curves for the minibatches sizes tested and learning rate warmup.

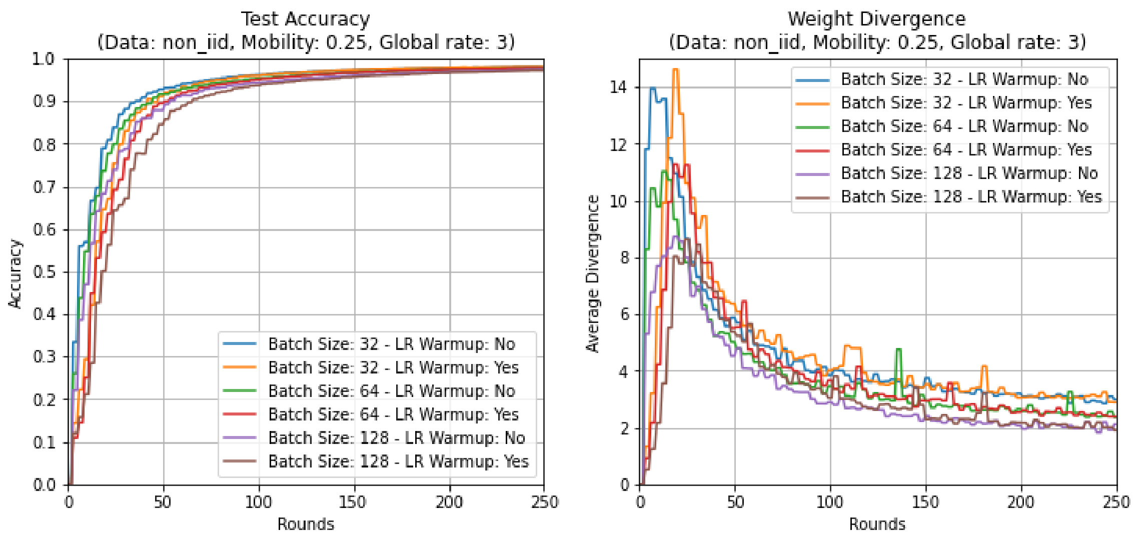


Figure 7.4: Comparison of the impact of minibatches sizes and LR warmup, with 25% client mobility rate and the global update rate set to 3.

The results outline that there is not an advantage in increasing the batch size or using a learning rate warmup for this dataset and data distribution, in terms of speeding up convergence. On the other hand, if we look at the average weight divergence, the use of larger minibatches is an effective strategy for reducing the divergence of the updates.

7.4.1.B Local SGD & Post Local SGD.

In Table 7.8 we show the results for Local SGD and Post Local SGD, where clients have a mobility rate of 25%. Similarly to the large minibatch SGD case, in Local SGD and Post Local SGD the clients mobility rate does not influence much the results for this type of data distribution, as shown in Table 7.6 and Table 7.7 , therefore we have only included the results for one mobility parameter.

Non-IID	**Rounds @ 0.95 Accuracy**	**Average Weight Divergence**	**Median Latency**
Mobility rate	-0.01	-0.04	0.58
Global update rate	0.04	0.82	0.06
Local Iterations	-0.86	0.50	0.12

Table 7.6: Correlations for Local SGD on MNIST.

The results show that for this type of non-IID distribution, there is a large improvement in the speed of convergence when performing multiple

Non-IID	**Rounds @ 0.95 Accuracy**	**Average Weight Divergence**	**Median Latency**
Mobility rate	-0.05	-0.08	0.66
Global update rate	0.75	-0.54	0.11
Local Iterations	-0.22	0.20	0.06
Epochs Delay Local SGD	0.12	0.01	0.06

Table 7.7: Correlations for Post Local SGD on MNIST.

updates at the client level in each round. However, as we can see in Fig. 7.5, if we delay the start of Local SGD the model generalization is severely slowed down.
From the perspective of weight divergence, this metric increases as we increase the number of local iterations. If we adopt Post Local SGD, the weight divergence is lower, especially in the first few rounds if we compare it to the Local SGD counterparts, as clients are able to vary the model weights less as they perform only one local update.

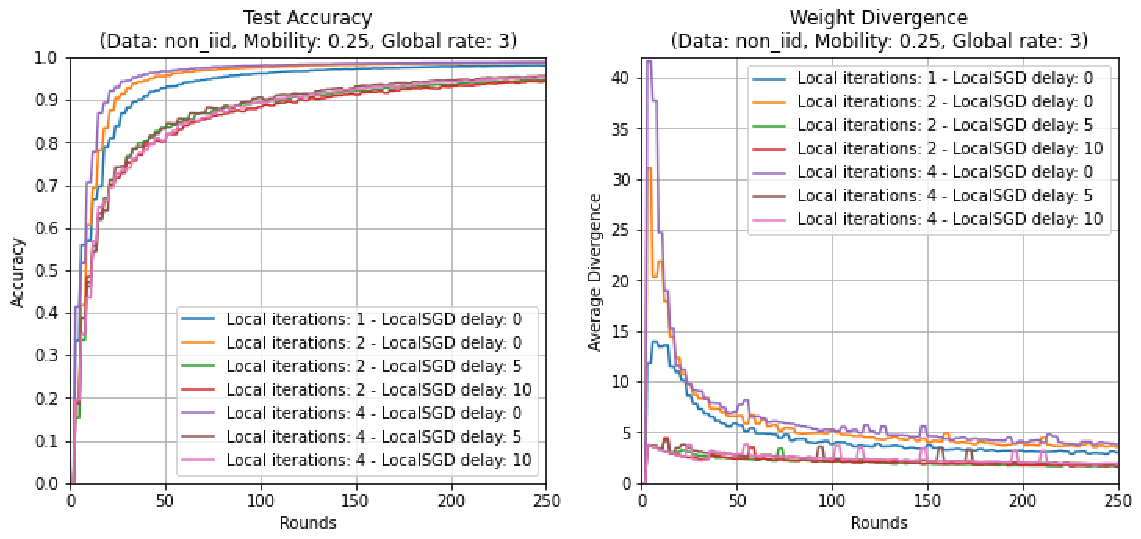


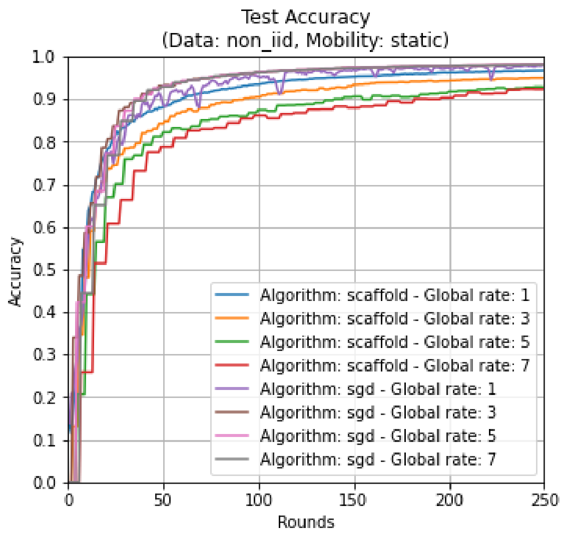
Figure 7.5: Comparison of Local SGD and Post Local SGD, with 25% client mobility rate and the global rate set to 3.

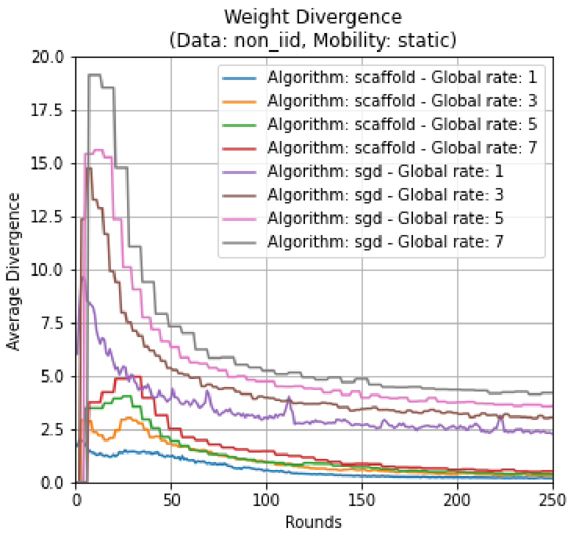
7.4.2 SCAFFOLD

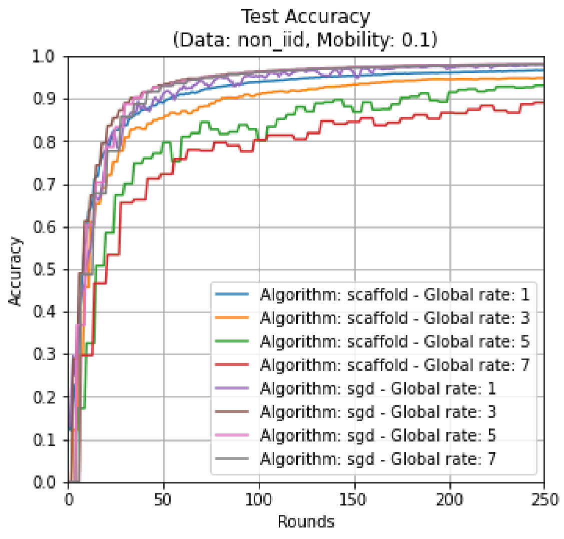
Let us now analyze the results that we obtained using the SCAFFOLD algorithm and compare them to the correspondent results for federated averaging based on SGD.

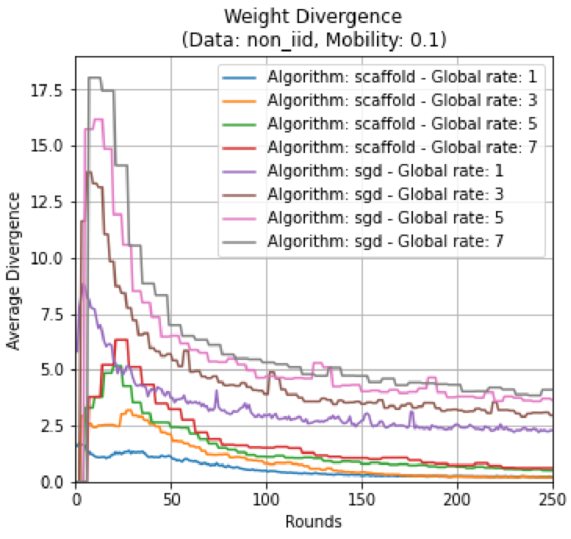
Global Update Rate	**Local Iterations**	**Epochs Delay LocalSGD**	**Rounds @ 0.95 Accuracy**	**Average Weight Divergence**
1	1	-	93	**3.3**
	2	-	53	4.2
		5	66	4.2
		10	79	4.2
	4	-	**34**	4.5
		5	51	4.5
		10	62	4.7
3	1	-	75	4.7
	2	-	42	5.9
		5	252	**2.1***
		10	255	2.2*
	4	-	**33**	6.7
		5	216	2.4
		10	231	2.4
5	1	-	70	5.5
	2	-	55	7.1
		5	255	**2.4***
		10	265	**2.4***
	4	-	**40**	8.4
		5	220	2.8
		10	230	2.8
7	1	-	77	6.2
	2	-	49	8.4
		5	252	**2.6***
		10	259	2.8*
	4	-	**42**	9.8
		5	210	3.1
		10	217	3

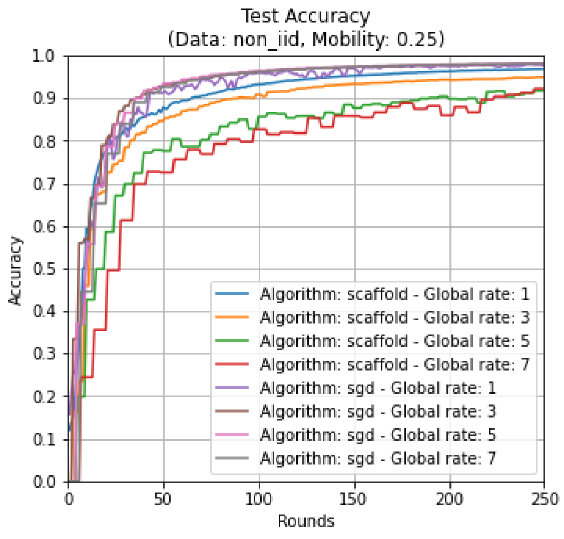
Table 7.8: Results on MNIST for Local SGD and Post Local SGD; mobility rate 25%. In bold are marked the best results in each *global update rate* group.
The values marked with an asterisk in the *Average Weight Divergence* column have been computed on the first 250 rounds (as for all the other simulations) even though we ran the simulation for more rounds than that to reach 95% test accuracy.

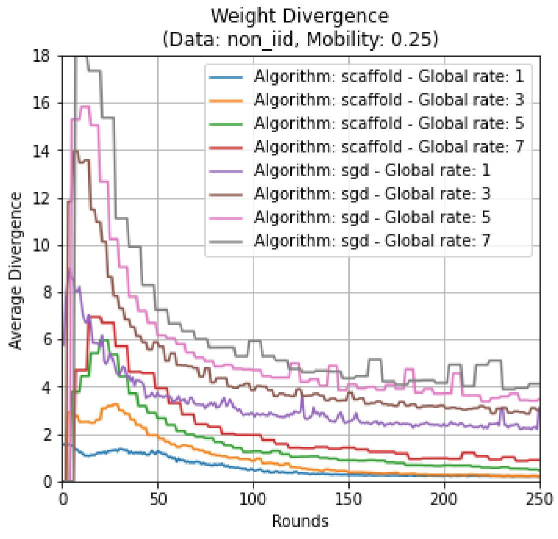

Test Accuracy
(Data: non_iid, Mobility: static)
Accuracy
Rounds
Algorithm: scaffold - Global rate: 1
Algorithm: scaffold - Global rate: 3
Algorithm: scaffold - Global rate: 5
Algorithm: scaffold - Global rate: 7
Algorithm: sgd - Global rate: 1
Algorithm: sgd - Global rate: 3
Algorithm: sgd - Global rate: 5
Algorithm: sgd - Global rate: 7


Weight Divergence
(Data: non_iid, Mobility: static)
Average Divergence
Rounds
Algorithm: scaffold - Global rate: 1
Algorithm: scaffold - Global rate: 3
Algorithm: scaffold - Global rate: 5
Algorithm: scaffold - Global rate: 7
Algorithm: sgd - Global rate: 1
Algorithm: sgd - Global rate: 3
Algorithm: sgd - Global rate: 5
Algorithm: sgd - Global rate: 7


Test Accuracy
(Data: non_iid, Mobility: 0.1)
Accuracy
Rounds
Algorithm: scaffold - Global rate: 1
Algorithm: scaffold - Global rate: 3
Algorithm: scaffold - Global rate: 5
Algorithm: scaffold - Global rate: 7
Algorithm: sgd - Global rate: 1
Algorithm: sgd - Global rate: 3
Algorithm: sgd - Global rate: 5
Algorithm: sgd - Global rate: 7


Weight Divergence
(Data: non_iid, Mobility: 0.1)
Average Divergence
Rounds
Algorithm: scaffold - Global rate: 1
Algorithm: scaffold - Global rate: 3
Algorithm: scaffold - Global rate: 5
Algorithm: scaffold - Global rate: 7
Algorithm: sgd - Global rate: 1
Algorithm: sgd - Global rate: 3
Algorithm: sgd - Global rate: 5
Algorithm: sgd - Global rate: 7


Test Accuracy
(Data: non_iid, Mobility: 0.25)
Accuracy
Rounds
Algorithm: scaffold - Global rate: 1
Algorithm: scaffold - Global rate: 3
Algorithm: scaffold - Global rate: 5
Algorithm: scaffold - Global rate: 7
Algorithm: sgd - Global rate: 1
Algorithm: sgd - Global rate: 3
Algorithm: sgd - Global rate: 5
Algorithm: sgd - Global rate: 7


Weight Divergence
(Data: non_iid, Mobility: 0.25)
Average Divergence
Rounds
Algorithm: scaffold - Global rate: 1
Algorithm: scaffold - Global rate: 3
Algorithm: scaffold - Global rate: 5
Algorithm: scaffold - Global rate: 7
Algorithm: sgd - Global rate: 1
Algorithm: sgd - Global rate: 3
Algorithm: sgd - Global rate: 5
Algorithm: sgd - Global rate: 7

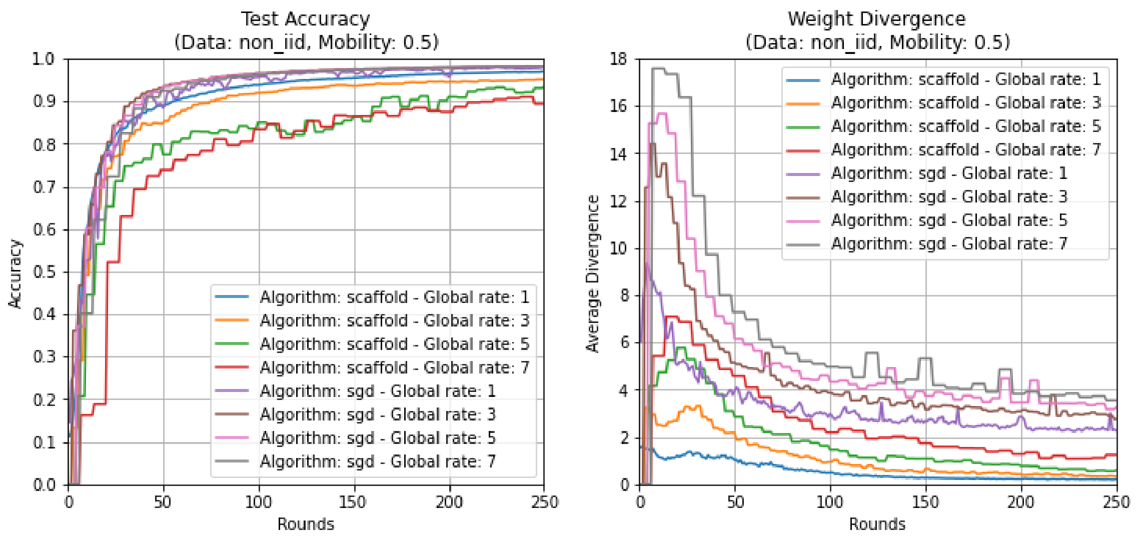


Figure 7.6: Comparison of SCAFFOLD and SGD on non-IID data.

The results show that using the SCAFFOLD update drift correction consistently underperforms in terms of speed of convergence when compared to standard SGD. However, as expected, SCAFFOLD reduces significantly the weight divergence of the updates.

7.5 Spatially Correlated Non-IID Data

In this section we will analyze the performance of the learning algorithms when applied to a hierarchical federated learning system in which data is distributed to clients in a non-IID fashion but it is spatially correlated at a cluster level. The class distribution in the tests performed is presented in Table 7.2.
Since in §7.3.2 we have shown that there is not a significant difference in restricting client mobility only to adjacent clusters, for conciseness reasons we will present below only the results for simulations in which clients are allowed to move to any cluster.
Moreover, because many simulations did not reach 95% test accuracy due to the more difficult data distribution, we have lowered the target accuracy of these results to 90%.

Global Update Rate	**Batch Size**	**Learning Rate Warmup**	**Rounds @ 0.90 Accuracy**	**Average Weight Divergence**
1	32	No	**54**	4
	64	No	128	2.1
		Yes	145	**2**
	128	No	92	2.7
		Yes	102	2.6
3	32	No	**48**	5.6
	64	No	144	2.7
		Yes	147	**2.6**
	128	No	105	3.3
		Yes	108	3.1
5	32	No	**50**	6.6
	64	No	135	**2.9**
		Yes	150	**2.9**
	128	No	100	3.7
		Yes	105	3.6
7	32	No	**56**	7.3
	64	No	154	3.3
		Yes	154	**3.2**
	128	No	98	4.2
		Yes	112	3.9

Table 7.9: Results for Large Minibatch SGD on MNIST with spatially correlated non-IID data; mobility rate 25%.
In bold are marked the best results in each *global update rate* group.

7.5.1 Minibatch SGD

7.5.1.A Large Minibatch SGD.

In Table 7.10 we present the results for experiments with the Large Minibatch SGD (§2.1.1, [9]) learning algorithm.

In Fig. 7.7 we show an example of the test accuracy and update divergence curves for the minibatches sizes tested and the linear learning rate warmup technique used.

Spatially Correlated Non-IID	**Rounds @ 0.90 Accuracy**	**Average Weight Divergence**	**Median Latency**
Mobility rate	-0.16	-0.41	0.58
Global update rate	0.04	0.39	-0.04
Batch size	0.05	-0.24	0.12
Learning rate warmup	0.37	-0.26	-0.08

Table 7.10: Correlations for Large Minibatch SGD on MNIST with spatially correlated non-IID data distribution.

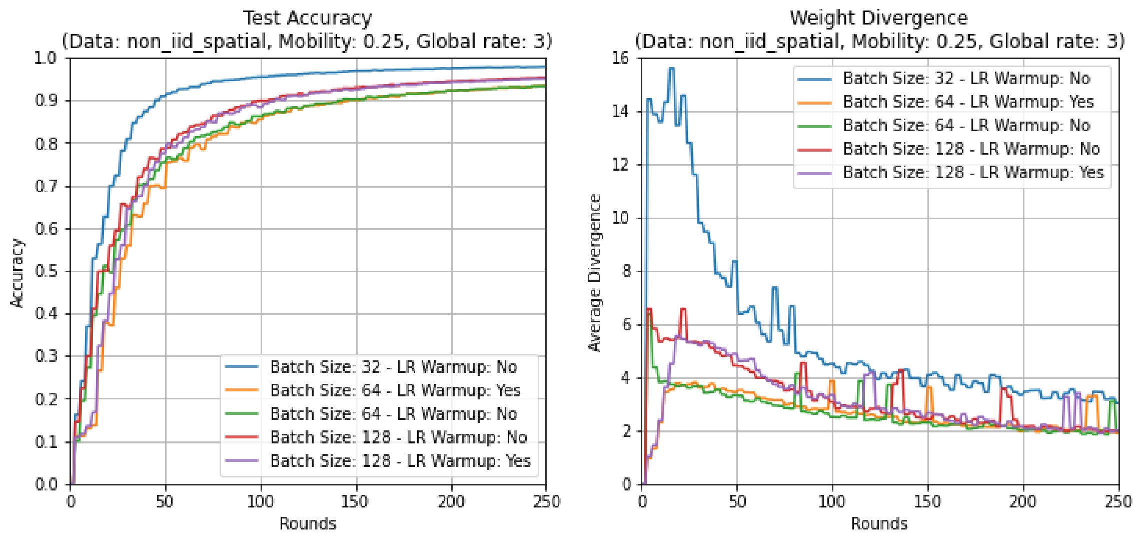


Figure 7.7: Comparison of the impact of minibatches sizes and LR warmup, with 25% client mobility rate and the global update rate set to 3.

The results outline that increasing the batch size or using a learning rate warmup does not improve convergence speed. On the other hand, the use of larger minibatches and learning rate warmup are effective strategies for reducing the size of the updates. These results are in accordance with those found for the client-level non-IID data distribution analyzed above.

7.5.1.B Local SGD & Post Local SGD.

We show the results for Local SGD and Post Local SGD with 25% of client mobility rate in Fig. 7.8 and Table 7.13. We have summarized the results for the mobility rates and global update rates tested in Table 7.11 and Table 7.12.

The results show that performing multiple local updates during the

Spatially Correlated Non-IID	**Rounds @ 0.90 Accuracy**	**Average Weight Divergence**	**Median Latency**
Mobility rate	-0.09	-0.31	0.62
Global update rate	0.05	0.38	0.00
Local Iterations	0.51	-0.55	0.29

Table 7.11: Correlations for Local SGD on MNIST.

Spatially Correlated Non-IID	**Rounds @ 0.90 Accuracy**	**Average Weight Divergence**	**Median Latency**
Mobility rate	-0.08	-0.51	0.61
Global update rate	0.15	0.65	0.01
Local Iterations	-0.78	0.10	0.09
Epochs Delay Local SGD	0.04	-0.02	0.02

Table 7.12: Correlations for Post Local SGD on MNIST.

client learning step does not improve convergence for this type of data distribution. This results is in opposition to what found when using Local SGD on the previously tested non-IID data distribution.
Moreover, the use of Post Local SGD does not influence convergence or the size of the updates as they are actually uncorrelated.

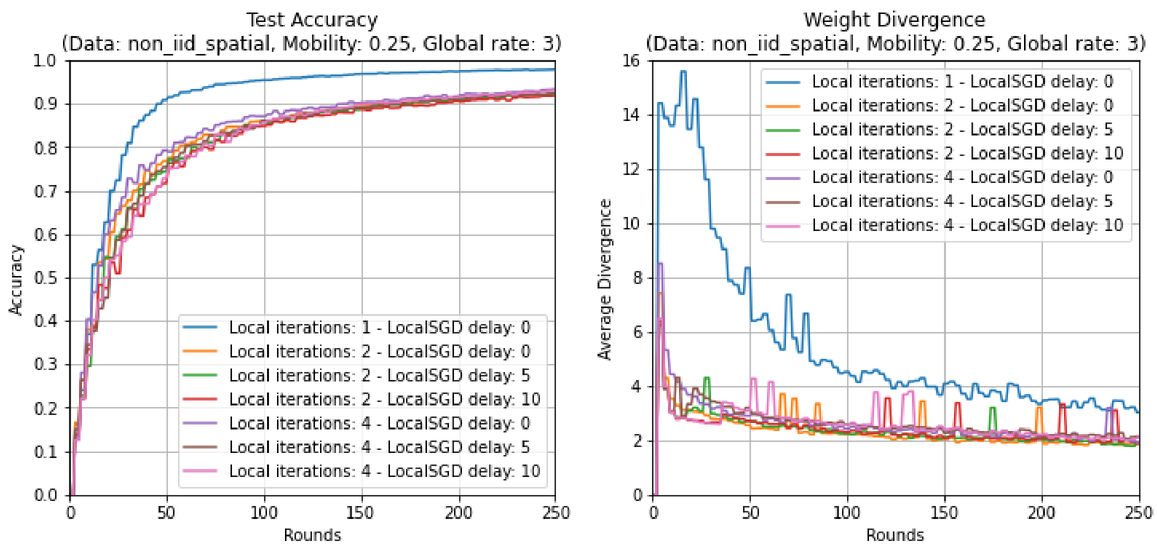


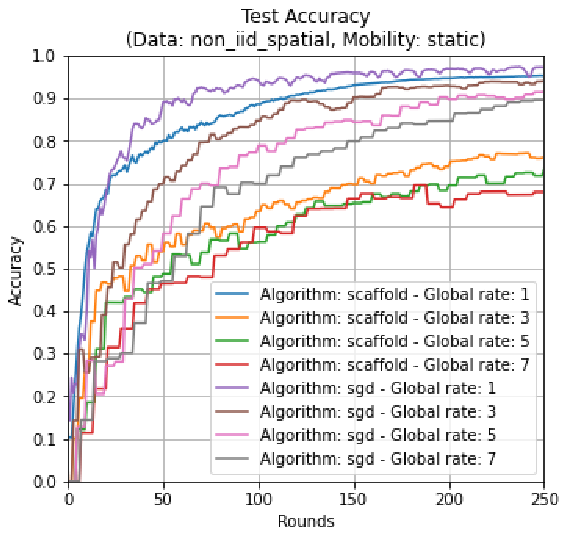
Figure 7.8: Comparison of Local SGD and Post Local SGD, with 25% client mobility rate and the global rate set to 3.

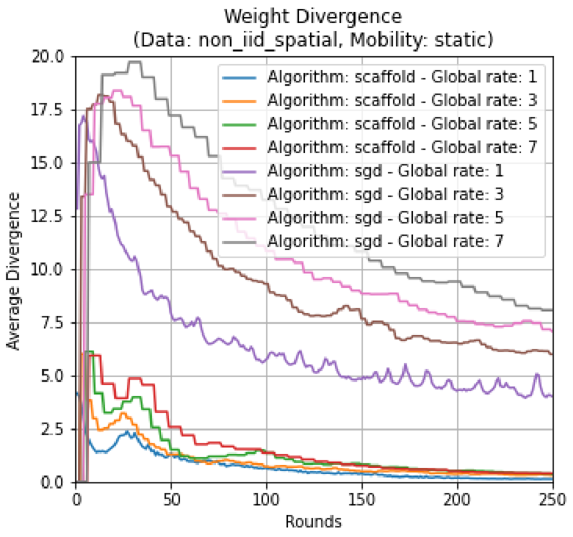
Global Update Rate	**Local Iterations**	**Epochs Delay LocalSGD**	**Rounds @ 0.90 Accuracy**	**Average Weight Divergence**
1	1	-	**54**	4
	2	-	177	1.7
		5	193	1.7
		10	194	**1.6**
	4	-	152	1.8
		5	152	1.7
		10	161	1.7
3	1	-	**48**	5.6
	2	-	168	**2.4**
		5	177	**2.4**
		10	189	**2.4**
	4	-	144	2.6
		5	168	2.6
		10	156	2.6
5	1	-	**50**	6.6
	2	-	170	2.8
		5	190	2.9
		10	190	**2.7**
	4	-	145	3.2
		5	165	3
		10	155	3
7	1	-	**56**	7.3
	2	-	182	3.2
		5	203	3.3
		10	189	**3**
	4	-	161	3.5
		5	161	3.4
		10	161	3.2

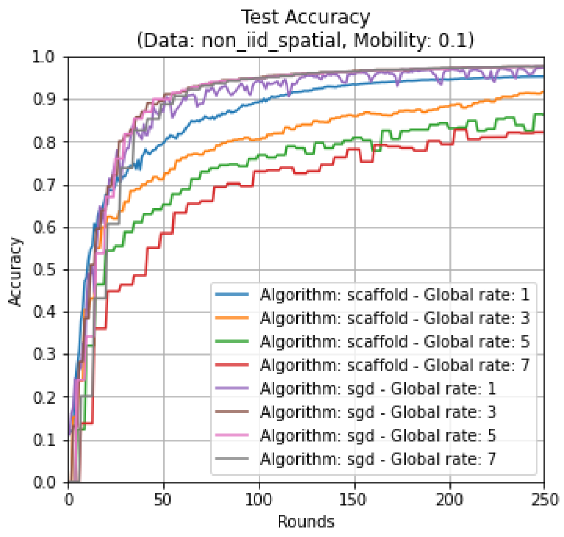
Table 7.13: Results on MNIST for Local SGD and Post Local SGD, for spatially correlated non-IID data; mobility rate 25%. In bold are marked the best results in each *global update rate* group.

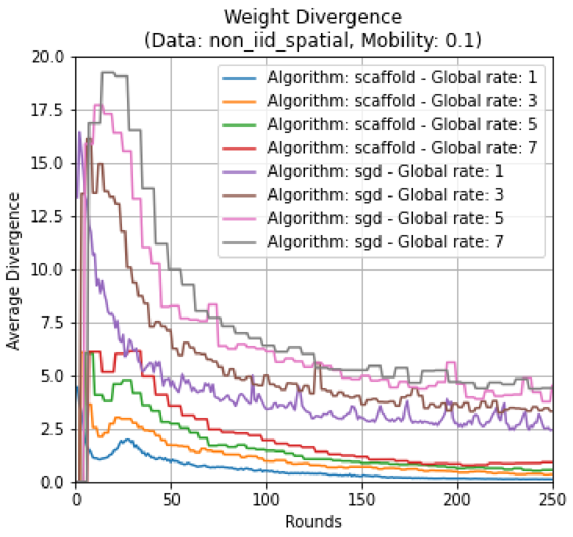
7.5.2 SCAFFOLD

Let us now analyze the results that we obtained using the SCAFFOLD algorithm and compare them to the correspondent results for federated averaging based on SGD.









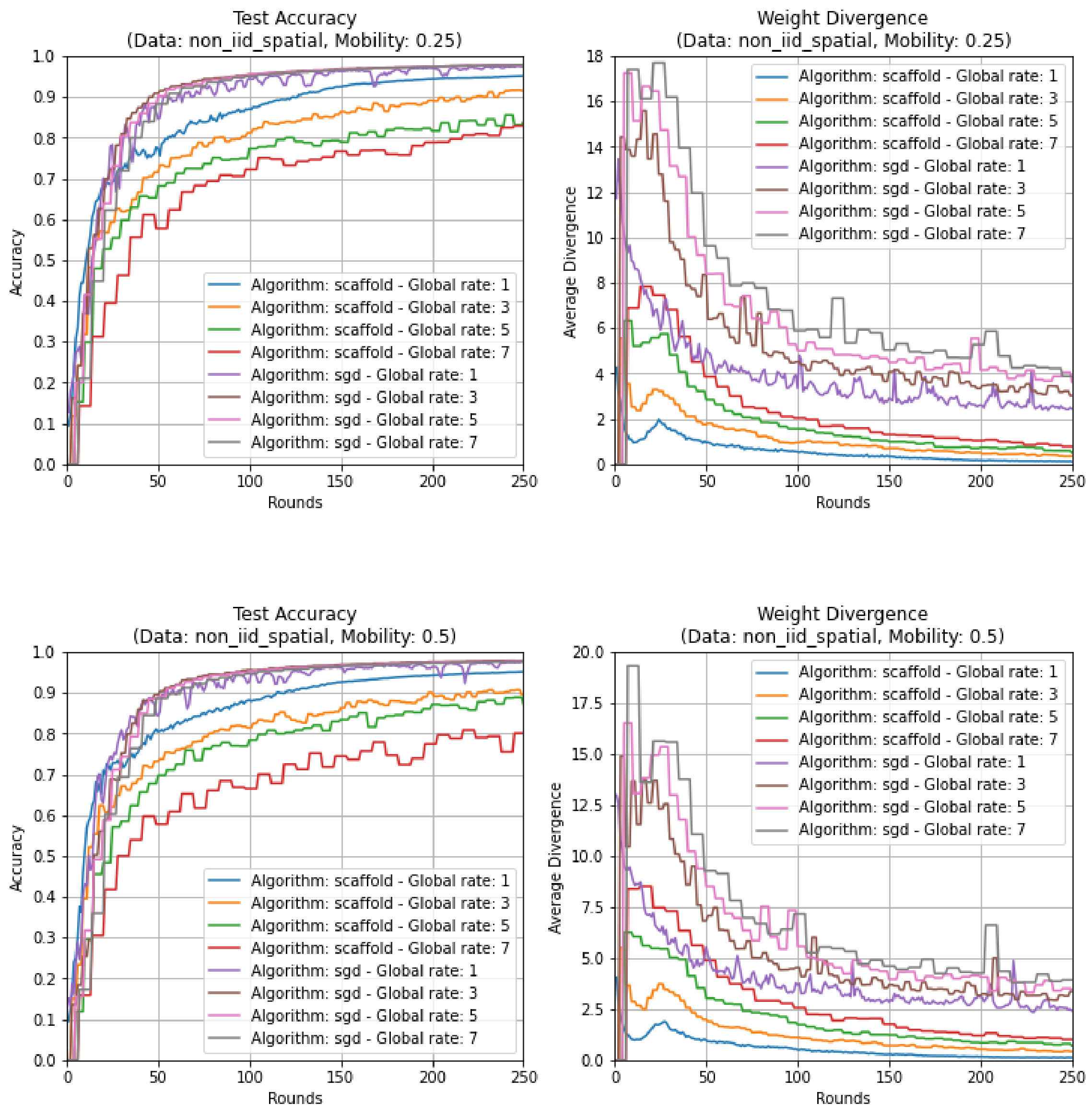


Figure 7.9: Comparison of SCAFFOLD and SGD on spatially correlated non-IID data.

The results show that, also for this type of non-IID distribution, introducing SCAFFOLD does not help convergence, as SGD-based experiments always yield better learning curves. However, applying the drift correction on the gradients reduces the update divergence.

7.6 MobileNetV2 on CIFAR-10

In Fig. 7.10 we show the comparison between the different data distributions of the learning curves obtained with minibatch SGD and static clients, when varying the global update rate. It is clear that this learning algorithm is not able to handle non-IID data for such a large model

(3.5M trainable parameters), and fails to converge to a satisfactory level of test accuracy.

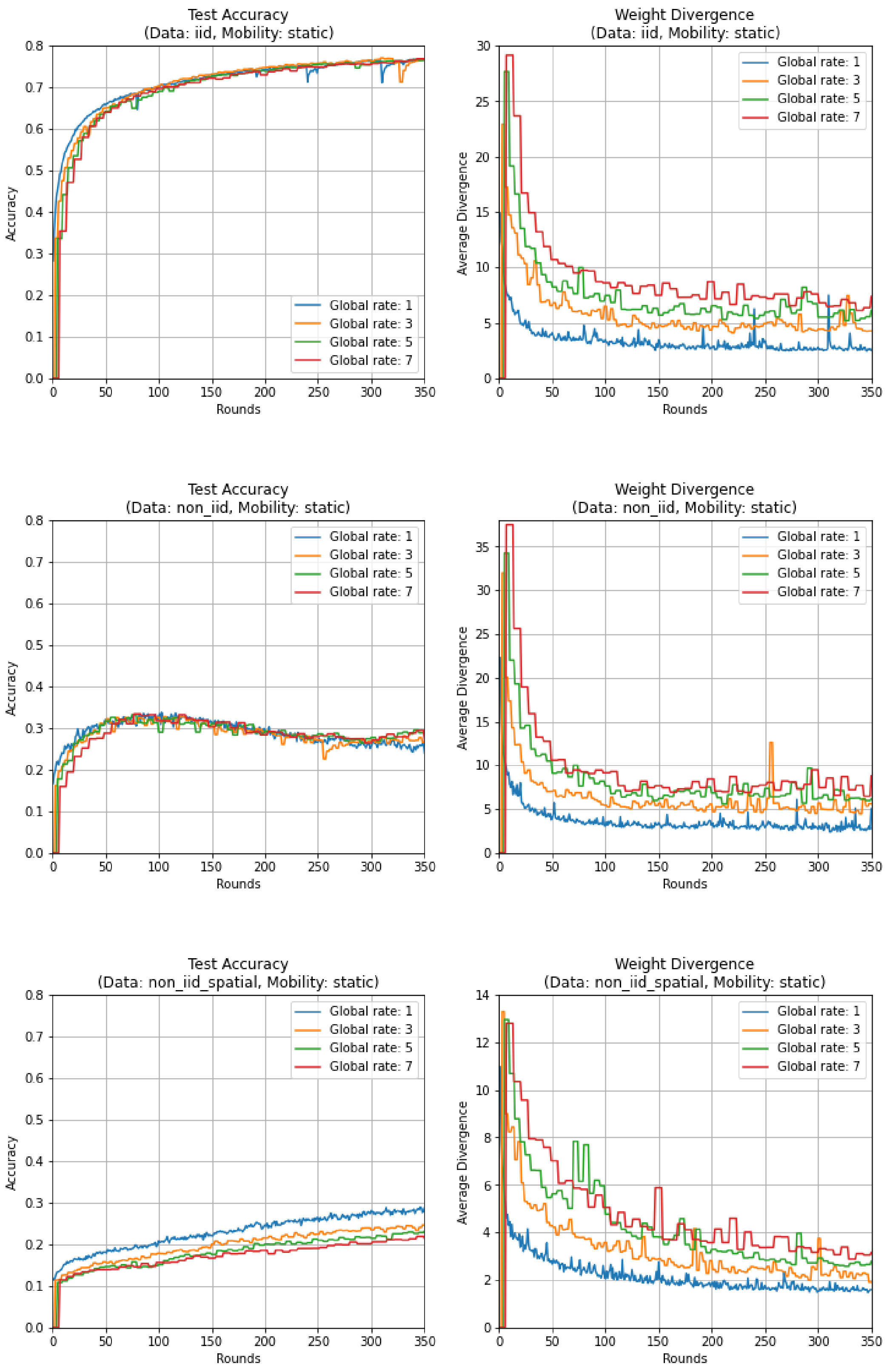


Figure 7.10: Comparison of data distributions for static clients.

Chapter 8

Evaluation

In this chapter we summarize the work done for this project and compare it with the technical requirements stated in Chapter 3 and the research objectives contained in Chapter 1.

8.1 Software Infrastructure

From a software point of view, the original requirements were to create a HFL simulator, an automated testbench, and a results visualizer.

We have succeeded in creating a fully working **simulator** which satisfies the following functional specifications: we can configure the simulator by inputting a JSON file containing the parameters values to use; we have implemented logic to load a dataset, partition the training data in various ways, and distribute the partitions to the clients; we are able to run experiments that implement a variety of federated learning algorithms and log the state of the simulator along the execution; finally, the simulator can run multiple experiments for the same configuration, and then combine the results together into a single results file.
Moreover, we have taken into account and implemented also some non-functional requirements, such as making the simulator software architecture as modular as possible so that it is easy to introduce new features or editing existing ones, and making the simulator computationally efficient and minimizing its memory footprint.

The **testbench** created is able to automatically generate the configuration files for the desired tests, and execute multiple simulations in parallel to maximise the throughput of the hardware that we are using.

The **results visualizer** based on IPython achieves the requirements of being an interactive tool to easily select and filter results, and produce plots and tables to visually analyze the performance of the tests.

8.2 Experimental Results

For what concerns the research objectives of this project, we aimed at assessing the impact of client mobility in a hierarchical federated learning system, and studying how different learning algorithms perform on different examples of non-IID data distributions.

Client mobility. The simulations results showed that when data is distributed across the HFL system in an IID fashion, moving clients do not influence the learning process.
Similarly, if data is non-IID across clients (i.e. each client has samples from two classes), client mobility is not a key factor in varying the learning performance of the system.
Lastly, we have shown that, when data is non-IID and spatially correlated, client mobility helps largely in accelerating the model convergence; this is because, thanks to moving clients, SBSs can directly learn from data samples of classes that they did not have access to initially, and can rely less on the global inter-cluster updates.

Non-IID data distributions. We have found that, for both distributions tested, using Large Minibatch SGD and/or learning rate warmup causes model generalization issues. This result is in accordance with what is found in the related literature [25, 21, 28], which describes as *generalization gap* this problem that arises when using large batches for training neural networks.
On the other hand, Local SGD can improve the convergence speed when data is non-IID, but has the opposite effect if data is spatially correlated. This is because with spatially-correlated non-IID data, it does not benefit convergence to perform multiple local iterations especially at the beginning of the learning process (i.e. when model weights change rapidly), when SBSs only have access to a very skewed data distribution.
We have found that the effects of Post-Local SGD are almost negligible

when data is non-IID and spatially correlated; otherwise, in the other non-IID distribution tested, we found that delaying the start of Local SGD has a negative effect on the model convergence.
The last algorithm tested was SCAFFOLD. The experimental results that we obtained on MNIST showed that there is not an advantage in implementing this update drift correction technique, in terms of helping to speed up the learning process.

To summarize our experimental findings regarding the convergence speed, we suggest to use Local SGD when data is non-IID, as performing multiple local iterations has yielded better results than the other algorithms tested. However, when we have spatially correlated non-IID data, standard minibatch SGD is the best performing algorithm.

Moreover, for what concerns the weight divergence, we have seen that SCAFFOLD, Post Local SGD, Large Minibatch SGD, and the linear learning rate warmup technique, all help reducing the average divergence of the updates, in both the non-IID scenarios. On the other hand, the effect of Local SGD differs in the two cases: executing multiple local iterations increases the weight divergence metric in the non-IID case, but it has an opposite effect when data in spatially correlated.

The latency analysis showed us that client mobility increases the time needed for each round. This is because when clients are allowed to move, we might end up with an unbalanced distribution of clients across clusters, and more populated clusters will take longer to execute one learning round. However, this result is strongly driven by the way we have modelled the latency computation and the scheduling algorithm that we have used. For this reason, we have included suggestions in §9.2 on how to further develop the latency model, so that a more insightful and realistic analysis can be performed.

Chapter 9

Conclusions and Further Work

9.1 Conclusions

In this project we have explored the field of federated learning, by focusing on its hierarchical variant. To support our research, we have built a versatile and modular software infrastructure.

We have studied how the learning process is affected by client mobility across clusters, which we think is a key element to investigate before implementing HFL with real-world devices. Indeed, since hierarchical federated learning is envisaged to be performed on networks of devices such as smartphones, wearables, and IoT nodes, it is necessary to assess whether we can reduce the impact, or even take advantage, of client mobility.

Furthermore, the second aspect that we have discussed is how non-IID data distributions affect the learning process and how different algorithms are influenced by this. This is a fundamental open issue in the field of federated learning [14, 47, 36, 35, 17, 29], because of the intrinsic non-IIDness of real world data generated by users, together with the impossibility of getting information on such data due to privacy constraints.

Along this project, we understood and showed that there is a very large range of parameters that influence the learning process of a hierarchical federated learning system, and that, for conciseness, we have only been able to inspect some of them. We include some suggestions of interesting experiments that can be run as further work.

9.2 Further Work

Federated learning is an immense field that encompasses a variety of research areas, such as machine learning, optimization, information theory, communications systems, computer networks, and many more. At the moment there is significant interest in this field and many are the open issues and research possibilities, as discussed in [19].
We will list in this section some possible enhancements that can be made to this project and some suggestions for continuing to research in this topic.

For what concerns the client mobility, we suggest to experiment on different scenarios, for example by varying the number, the arrangement and the size of the clusters, as well as the number of clients and their initial positions across clusters. It would also be interesting to have different rates of mobility for different groups of devices.

On the topic of data distributions, another case that would be interesting to consider is that of unbalanced sized clients datasets, or, moreover, a case in which the client-held datasets evolve overtime. This latter experiment will be a way to also research into the area of model personalization [45]. An extension of federated learning that could be tested in this domain is data selection (§2.2.3, [42]).

About the robustness of the learning algorithms with respect to non-IID data, interesting strategies to study are Federated Distillation and Augmentation [17], Clustered Federated Learning [35, 36], as well as implementing the minimal data sharing technique described in [47].

For what concerns the domain of wireless communications, there are many possible improvements. We can evolve the scheduling algorithm by performing client selection (§2.2.1, [31]) or even perform client selection in 2 stages (both in the downlink and the uplink). We could develop further the latency computation, by allowing clusters to proceed to the next round independently without needing to wait for all the other clusters if there is not a global update step, or also by taking into consideration the communication overhead of performing the global updates. Moreover, we could improve the computation time model, by

accounting for different classes of devices with different computational capabilities.
It would also be interesting to include in the simulator communication-efficient features such as gradient sparsification or update compression [37, 1, 40, 24, 23, 44, 41].

In addition, we could run more experiments on the MobileNetV2, and the performance of HFL on other learning tasks could be investigated: for example on next-word prediction with a LSTM, or other tasks related to NLP, like machine translation, sentiment analysis, or next-character prediction. Moreover, we could explore the suitability of FL and HFL in training very large models like the VGG family and the Resnet family.

To conclude, another area to explore is the application of meta-learning in federated learning [22, 18, 7] which could help in determining the impact of the system parameters on the final model accuracy and on the latency. It would be interesting for example to develop a system to automatically set the learning parameters based on the state of the learning process and the position or rate of mobility of the clients, in order to maximize the convergence speed or minimize latency.

Chapter 10

User Guide

In this chapter we will describe how to use the simulator, the test-bench and the results visualizer. The project is hosted on GitHub and can be found at `www.github.com/alexander3605/wireless-HFL`.
In this chapter we will assume that all paths are relative to the root directory of the repository `wireless-HFL/`.

10.1 Simulator

The simulator can be used as a standalone programme; it is executed by running

```
python3 main.py [--config=file_name]
                [--is_folder=bool]
                [--device=device_name]
                [--n_experiments=int]
```

All parameters are optional and they take the following default values, if they are not specified:

```
--config = ""
--is_folder = False
--device = torch.device("cuda:0" if torch.cuda.is_available()
                        else "cpu")
--n_experiments = 1
```

In `config` we specify the configuration file or folder that we want to execute. The exact path is constructed by concatenating

```
wireless-HFL/config_files/{$config}
```

If `is_folder` is set to true, all files in the specified folder are executed sequentially. With `device` we can specify which GPU/CPU to use, and by setting `n_experiments` we can run multiple tests for the same configurations and have the results aggregated together in a single results file.
One example of this command is

```
python3 main.py --config=test_config.json --device=cuda:1
        --n_experiments=10
```

For what concerns possible **improvements** that have been outlined along the report and in §9.2, they fall in the following categories:

- Learning Algorithms: to implement new learning schemes, edit, in the `src` folder, `network.py`, `cluster.py`, `server.py`, and `client.py`.
- Data Distribution: to implement new data distributions add an extra function in `src/data_loader.py`. For new cluster and client arrangements edit
- Neural Network Architectures: to add new architectures include their class definitions in `src/nn_classes.py`.

10.2 Testbench

To use the testbench follow the steps listed below.

1. Open `testbench.py`.
2. Set the `MAX_THREADS` and `N_EXPERIMENTS constants`.
3. Edit the simulation parameters by setting for each the list of parameters that you want to iterate through. See an example below:

```
n_clusters = [25]
n_clients = [250]
mobility_rate = [0.0, 0.1, 0.25, 0.5]
move_to_neighbours = [False]
model_type = ["mnist"]
dataset_name = ["mnist"]
dataset_distribution = ["iid", "non_iid", "non_iid_spatial"]
client_algorithm = ["sgd"]
```

```
client_batch_size = [32]
client_n_epochs = [2,4]
client_lr = [0.0100]
server_global_rate = [1]
client_selection_fraction = [0.3]
lr_warmup = [False, True]
epochs_delay_localSGD = [5,10]
log_verbosity = [2]
log_frequency = [1]
stdout_verbosity = [2]
stdout_frequency = [1]
debug = [False]
```

4. Set the fraction of all the possible combinations that you want to run, by editing the variable `fraction_to_run`. If a value lower than 1 is chosen, the combinations are shuffled and the fraction is chosen to run.

5. Set the root string used to make the names of the generated results files, by setting the variable `results_name_root`.
   For example `results_name_root = "mnist_post-local-sgd"`.

6. Execute the testbench by running
   `python3 testbench.py`
   Or alternatively, if you wand to execute the testbench as a background process in Linux (and log off from the ssh session if you are working on a remote server for example), run
   `screen python3 testbench.py`
   and then press `ctrl+A, ctrl+D` to send the process to the background. To return to the process, type into the terminal `screen -r`.

The results files will be save into `wireless-HFL/log_files/`.

10.3 Results Visualizer

For using the results visualizer, open in the root directory the Jupyter notebook `results_viewer.ipynb`. In this notebook it is possible to filter the results by specifying the values that we want to have for each parameter; if an empty list of values if given for a parameter, no resutls will be filtered out depending on that parameter. An example of a filter and result sorting is presented below.

```
#####################################
#    DEFINE FILTERS FOR RESULTS
#####################################
filters = {}
filters["dataset_distribution"] = ["non_iid_spatial"]
filters["client_batch_size"] = [32,64,128]
filters["client_n_epochs"] = [1]
filters["mobility_rate"] = [0.25]
filters["server_global_rate"] = []
filters["lr_warmup"] = []
filters["epochs_delay_localSGD"] = [0]

filters["dataset_name"] = ["mnist"]
filters["model_type"] = []
filters["client_algorithm"] = ["sgd"]
filters["n_clusters"] = [25]
filters["move_to_neighbours"] = [False]

filters["client_lr"] = []
filters["client_selection_fraction"] = []
#####################################
#####################################
selected_metadata, selected_results = filter_results(metadata,
    results, filters,
    sort_by = ["server_global_rate","client_batch_size"])
```

After setting this, simply execute all the cells to produce the accuracy and weight divergence plots, as well as the correlation matrix for the parameters that vary in the selected experiments.

Bibliography

[1] Mehdi Salehi Heydar Abad et al. *Hierarchical Federated Learning Across Heterogeneous Cellular Networks.* 2019. arXiv: `1909.02362 [cs.LG]`.

[2] Mohammad Mohammadi Amiri, Tolga M. Duman, and Deniz Gündüz. "Collaborative Machine Learning at the Wireless Edge with Blind Transmitters". In: *CoRR* abs/1907.03909 (2019). arXiv: `1907.03909`. URL: `http://arxiv.org/abs/1907.03909`.

[3] Keith Bonawitz et al. "Practical Secure Aggregation for Privacy-Preserving Machine Learning". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.* CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 1175–1191. ISBN: 9781450349468. DOI: `10.1145/3133956.3133982`. URL: `https://doi.org/10.1145/3133956.3133982`.

[4] Kai Chen and Qiang Huo. "Scalable Training of Deep Learning Machines by Incremental Block Training with Intra-block Parallel Optimization and Blockwise Model-Update Filtering". In: *ICASSP-2016.* Mar. 2016. URL: `https://www.microsoft.com/en-us/research/publication/scalable-training-deep-learning-machines-incremental-block-training-intra-block-parallel-optimization-blockwise-model-update-filtering/`.

[5] J. Deng et al. "ImageNet: A large-scale hierarchical image database". In: *2009 IEEE Conference on Computer Vision and Pattern Recognition.* 2009, pp. 248–255.

[6] ETSI. *RF requirements for LTE Pico NodeB.* URL: `https://www.etsi.org/deliver/etsi%5C_tr/136900%5C_136999/136931/09.00.00%5C_60/tr%5C_136931v090000p.pdf`.

[7] Chelsea Finn, Pieter Abbeel, and Sergey Levine. "Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks". In: *CoRR* abs/1703.03400 (2017). arXiv: `1703.03400`. URL: `http://arxiv.org/abs/1703.03400`.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* `http://www.deeplearningbook.org`. MIT Press, 2016.

[9] Priya Goyal et al. *Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour.* 2017. arXiv: `1706.02677 [cs.CV]`.

[10] Deniz Gündüz et al. "Machine Learning in the Air". In: *CoRR* abs/1904.12385 (2019). arXiv: `1904.12385`. URL: `http://arxiv.org/abs/1904.12385`.

[11] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *CoRR* abs/1512.03385 (2015). arXiv: `1512.03385`. URL: `http://arxiv.org/abs/1512.03385`.

[12] Andrew G. Howard et al. “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications”. In: *CoRR* abs/1704.04861 (2017). arXiv: `1704.04861`. URL: `http://arxiv.org/abs/1704.04861`.

[13] Andrew Howard et al. *Searching for MobileNetV3*. 2019. arXiv: `1905.02244 [cs.CV]`.

[14] Kevin Hsieh et al. *The Non-IID Data Quagmire of Decentralized Machine Learning*. 2019. arXiv: `1910.00189 [cs.LG]`.

[15] Sergey Ioffe. “Batch Renormalization: Towards Reducing Minibatch Dependence in Batch-Normalized Models”. In: *CoRR* abs/1702.03275 (2017). arXiv: `1702.03275`. URL: `http://arxiv.org/abs/1702.03275`.

[16] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *CoRR* abs/1502.03167 (2015). arXiv: `1502.03167`. URL: `http://arxiv.org/abs/1502.03167`.

[17] Eunjeong Jeong et al. “Communication-Efficient On-Device Machine Learning: Federated Distillation and Augmentation under Non-IID Private Data”. In: *CoRR* abs/1811.11479 (2018). arXiv: `1811.11479`. URL: `http://arxiv.org/abs/1811.11479`.

[18] Yihan Jiang et al. “Improving federated learning personalization via model agnostic meta learning”. In: *arXiv preprint arXiv:1909.12488* (2019).

[19] Peter Kairouz et al. *Advances and Open Problems in Federated Learning*. 2019. arXiv: `1912.04977 [cs.LG]`.

[20] Sai Praneeth Karimireddy et al. *SCAFFOLD: Stochastic Controlled Averaging for On-Device Federated Learning*. 2019. arXiv: `1910.06378 [cs.LG]`.

[21] Nitish Shirish Keskar et al. “On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima”. In: *CoRR* abs/1609.04836 (2016). arXiv: `1609.04836`. URL: `http://arxiv.org/abs/1609.04836`.

[22] Mikhail Khodak, Maria-Florina Balcan, and Ameet Talwalkar. “Adaptive Gradient-Based Meta-Learning Methods”. In: *CoRR* abs/1906.02717 (2019). arXiv: `1906.02717`. URL: `http://arxiv.org/abs/1906.02717`.

[23] Jakub Konecný and Peter Richtárik. “Randomized Distributed Mean Estimation: Accuracy vs Communication”. In: *CoRR* abs/1611.07555 (2016). arXiv: `1611.07555`. URL: `http://arxiv.org/abs/1611.07555`.

[24] Jakub Konecný et al. “Federated Learning: Strategies for Improving Communication Efficiency”. In: *CoRR* abs/1610.05492 (2016). arXiv: `1610.05492`. URL: `http://arxiv.org/abs/1610.05492`.

[25] Tao Lin et al. *Don’t Use Large Mini-Batches, Use Local SGD*. 2018. arXiv: `1808.07217 [cs.LG]`.

[26] Yujun Lin et al. “Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training”. In: *CoRR* abs/1712.01887 (2017). arXiv: `1712.01887`. URL: `http://arxiv.org/abs/1712.01887`.

[27] Lumin Liu et al. “Edge-Assisted Hierarchical Federated Learning with Non-IID Data”. In: *CoRR* abs/1905.06641 (2019). arXiv: `1905.06641`. URL: `http://arxiv.org/abs/1905.06641`.

[28] Sam McCandlish et al. "An empirical model of large-batch training". In: *arXiv preprint arXiv:1812.06162* (2018).

[29] H. Brendan McMahan et al. "Federated Learning of Deep Networks using Model Averaging". In: *CoRR* abs/1602.05629 (2016). arXiv: 1602.05629. URL: http://arxiv.org/abs/1602.05629.

[30] Kensuke Nakamura and Byung-Woo Hong. "Adaptive Weight Decay for Deep Neural Networks". In: *CoRR* abs/1907.08931 (2019). arXiv: 1907.08931. URL: http://arxiv.org/abs/1907.08931.

[31] Takayuki Nishio and Ryo Yonetani. "Client Selection for Federated Learning with Heterogeneous Resources in Mobile Edge". In: *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)* (May 2019). DOI: 10.1109/icc.2019.8761315. URL: http://dx.doi.org/10.1109/ICC.2019.8761315.

[32] Jihong Park et al. "Wireless Network Intelligence at the Edge". In: *CoRR* abs/1812.02858 (2018). arXiv: 1812.02858. URL: http://arxiv.org/abs/1812.02858.

[33] Mark Sandler and Andrew Howard. *MobileNetV2: The Next Generation of On-Device Computer Vision Networks.* 2018. URL: https://ai.googleblog.com/2018/04/mobilenetv2-next-generation-of-on.html.

[34] Mark Sandler et al. "Inverted Residuals and Linear Bottlenecks: Mobile Networks for Classification, Detection and Segmentation". In: *CoRR* abs/1801.04381 (2018). arXiv: 1801.04381. URL: http://arxiv.org/abs/1801.04381.

[35] Felix Sattler, Klaus-Robert Muller, and Wojciech Samek. "Clustered Federated Learning: Model-Agnostic Distributed Multi-Task Optimization under Privacy Constraints". In: *arXiv preprint arXiv:1910.01991* (2019).

[36] Felix Sattler et al. *On the Byzantine Robustness of Clustered Federated Learning.* 2020. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=9054676.

[37] Felix Sattler et al. "Robust and Communication-Efficient Federated Learning from Non-IID Data". In: *CoRR* abs/1903.02891 (2019). arXiv: 1903.02891. URL: http://arxiv.org/abs/1903.02891.

[38] Pulkit Sharma, Farah E Shamout, and David A Clifton. *Preserving Patient Privacy while Training a Predictive Model of In-hospital Mortality.* 2019. arXiv: 1912.00354 [cs.LG].

[39] Shaohuai Shi et al. "A Distributed Synchronous SGD Algorithm with Global Top-k Sparsification for Low Bandwidth Networks". In: *CoRR* abs/1901.04359 (2019). arXiv: 1901.04359. URL: http://arxiv.org/abs/1901.04359.

[40] Ananda Theertha Suresh et al. "Distributed mean estimation with limited communication". In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 3329–3337.

[41] Hanlin Tang et al. "Communication Compression for Decentralized Training". In: *NeurIPS*. 2018.

[42] Tiffany Tuor et al. *Data Selection for Federated Learning with Relevant and Irrelevant Data at Clients.* 2020. arXiv: 2001.08300 [cs.LG].

[43] Wieslawa Wajda et al. "INFSO-ICT-247733 EARTH Deliverable D 2 . 3 Energy efficiency analysis of the reference systems , areas of improvements and target breakdown". In: 2012.

[44] Hongyi Wang et al. "ATOMO: Communication-efficient Learning via Atomic Sparsification". In: *NeurIPS*. 2018.

[45] Kangkang Wang et al. "Federated evaluation of on-device personalization". In: *arXiv preprint arXiv:1910.10252* (2019).

[46] Howard H Yang et al. "Age-based scheduling policy for federated learning in mobile edge networks". In: *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2020, pp. 8743–8747.

[47] Yue Zhao et al. "Federated Learning with Non-IID Data". In: *CoRR* abs/1806.00582 (2018). arXiv: `1806.00582`. URL: `http://arxiv.org/abs/1806.00582`.

[48] Guangxu Zhu et al. "Towards an Intelligent Edge: Wireless Communication Meets Machine Learning". In: *CoRR* abs/1809.00343 (2018). arXiv: `1809.00343`. URL: `http://arxiv.org/abs/1809.00343`.