FL_PyTorch : Optimization Research Simulator for Federated Learning

KONSTANTIN BURLACHENKO, KAUST, KSA
SAMUEL HORVÁTH, KAUST, KSA
PETER RICHTÁRIK, KAUST, KSA

Federated Learning (FL) has emerged as a promising technique for edge devices to collaboratively learn a shared machine learning model while keeping training data locally on the device, thereby removing the need to store and access the full data in the cloud. However, FL is difficult to implement, test and deploy in practice considering heterogeneity in common edge device settings, making it fundamentally hard for researchers to efficiently prototype and test their optimization algorithms. In this work, our aim is to alleviate this problem by introducing FL_PyTorch : a suite of open-source software written in python that builds on top of one the most popular research Deep Learning (DL) framework PyTorch. We built FL_PyTorch as a research simulator for FL to enable fast development, prototyping and experimenting with new and existing FL optimization algorithms. Our system supports abstractions that provide researchers with a sufficient level of flexibility to experiment with existing and novel approaches to advance the state-of-the-art. Furthermore, FL_PyTorch is a simple to use console system, allows to run several clients simultaneously using local CPUs or GPU(s), and even remote compute devices without the need for any distributed implementation provided by the user. FL_PyTorch also offers a Graphical User Interface. For new methods, researchers only provide the centralized implementation of their algorithm. To showcase the possibilities and usefulness of our system, we experiment with several well-known state-of-the-art FL algorithms and a few of the most common FL datasets.

1 INTRODUCTION

Over the past few years, the continual development of DL capabilities has revolutionised the way we interact with everyday devices. Much of this success depends on the availability of large-scale training infrastructures such as large GPU clusters and the ever-increasing demand for vast amounts of training data. In contrary to this, users and providers are becoming increasingly aware of their privacy leakage because of this centralised data collection, leading to the creation of various privacy-preserving initiatives by industry service providers [2] or government regulators [9].

Recently, a viable solution that has the potential to address the aforementioned issue is Federated Learning (FL). FL a term was initially proposed in [24] as an approach to solving learning tasks by a loose federation of mobile devices. However, the underlying concept of training models without data being available in a single location is applicable beyond the originally considered scenario and it turns out to be useful in other practical use-cases. For example, learning from institutional data silos such as hospitals or banks which cannot share data due to confidentiality or legal constraints, or applications in edge devices [16, 37].

The main goal of FL is to provide strong privacy protection which is obtained by storing data locally rather than transferring it to the central storage. To solve the underlying machine learning objective each client provides focused updates intended for immediate aggregation. Stronger privacy properties are possible when FL is combined with other technologies such as differential privacy [8] and secure multiparty computation (SMPC) protocols such as secure aggregation [4].

Authors' addresses: Konstantin Burlachenko, konstantin.burlachenko@kaust.edu.sa, KAUST, Thuwal, KSA; Samuel Horváth, samuel.horvath@kaust.edu.sa, KAUST, Thuwal, KSA; Peter Richtárik, peter.richtarik@kaust.edu.sa, KAUST, Thuwal, KSA.

© 2022 Published in *DistributedML'21, December 7, 2021, Virtual Event, Germany*

Algorithm 1: Generalized Federated Averaging

```
   Input: Initial model $x^{(0)}$, ClientOpt, ServerOpt
 1 Initialize server state $H^0$ = InitializeServerState()
 2 for $t \in \{0,1,\ldots,T-1\}$ do
 3     Sample a subset $\mathcal{S}^{(t)}$ of available clients
 4     Generate state: $s^{(t)}$ = ClientState($H^{(t)}$)
 5     Broadcast ($x^{(t)}$,$s^{(t)}$) to workers
 6     for client $i \in \mathcal{S}^{(t)}$ in parallel do
 7         Initialize local model $x_i^{(t,0)} = x^{(t)}$
 8         for $k = 0,\ldots,\tau_i - 1$ do
 9             Compute local stochastic gradient $g_i$ = LocalGradient($x_i^{(t,k)}, s_t$)
10             Perform local update $x_i^{(t,k+1)}$ = ClientOpt($x_i^{(t,k)}, g_i, k, t$)
11         end
12         Compute local model changes $\Delta_i^{(t)} = x_i^{(t,\tau_i)} - x_i^{(t,0)}$
13         Create local state update: $U_i^{(t)}$ = LocalState($x^{(t)}, x_i^{(t,\tau_i)}$)
14         Send ($\Delta_i^{(t)}$,$U_i^{(t)}$) to Master.
15     end
16     Obtain ($\Delta_i^{(t)}$,$U_i^{(t)}$), $\forall i \in \mathcal{S}^{(t)}$.
17     Compute $G^{(t)}$ = ServerGradient($\{\Delta_i^{(t)},U_i^{(t)}\}_{i\in\mathcal{S}^{(t)}}, H^{(t)}$)
18     Update global model $x^{(t+1)}$ = ServerOpt($x^{(t)}, G^{(t)},\eta_s,t$)
19     Update: $H^{(t+1)}$ = ServerGlobalState($\{\Delta_i^{(t)},U_i^{(t)}\}_{i\in\mathcal{S}^{(t)}}, H^{(t)}$)
20 end
```

Recently, federated learning has seen increasing attention not only in academia but also in the industry which already uses FL in their deployed systems. For instance, Google use it in the Gboard mobile keyboard for applications including next word prediction [12], emoji suggestion [32] or "Hey Google" Assistant [10]. Apple uses FL for applications like the QuickType keyboard and the vocal classifier for "Hey Siri" [3]. In the finance space, FL is used to detect money laundering [36] or financial fraud [19]. In the medical space, federated learning is used for drug discovery [25], for predicting COVID-19 patients' oxygen needs [28] or medical images analysis [29] and others.

To enable research in federated learning, several frameworks have been proposed including `LEAF` [6], `FedML` [13], `Flower` [5]), (`PySyft` [34], `TensorFlow-Federated` (TFF) [17], `FATE` [37], `Clara` [27], `PaddleFL` [23], `Open FL` [18]. These frameworks are mainly built with a focus to be deployed on real world systems while also providing user with an ability to run experiments with the same code. This desired property often comes with the price that the entry bar for researchers to extend or experiment with these frameworks is limited in a sense that they either need to have extensive experience with distributed systems or require assistance from experts on given framework.

In our work, we decided to take one step back and focus on the construction of a framework that, although not aimed for being deployed to edge devices as primary goal, can provide a useful simulation environment for researchers with the following goals:

- *Low Entry Bar/ Simplicity*: We aim our tool to be as simple as possible for the user while providing all necessary functionalities.

- *Extensibility*: It is easy to bring your own algorithm or dataset or extend existing ones. We aim to achieve this by providing universal abstractions with a sufficient level of flexibility to experiment with existing and novel approaches to advance the state-of-the-art.
- *Hardware Utilisation*: Cross-device FL experiments are usually of much smaller scale comparing to the centralized setting. This is mainly because of limited device capabilities. Running such experiments on GPU can lead to the under-utilisation of available hardware. We aim to resolve this via the ability to parallelise clients' computation.
- *Easy Debugging*: Debugging multi-process or multi-thread systems is hard. We only require user to provide a single thread implementation which is automatically adjusted to multi-GPU and multi-node setup.

To the best of our knowledge, there is no such tool that could provide a sufficient level of freedom and is simple to use and therefore, we design `FL_PyTorch` to achieve the aforementioned goal.

`FL_PyTorch` is an optimization research simulator for FL implemented in Python based on `PyTorch` [30]. `FL_PyTorch` is a simple to use tool with its own Graphical User Interface (GUI) implemented in PyQt [31]. During the simulation process, the selected local CPUs/GPUs are accessed in a parallel way. In addition, there is a possibility to use remote CPUs/GPUs. Remote devices are required to have a TCP transport layer for communicating with the master. Regarding supported devices, we target server and desktop stations running on Linux, macOS, or Windows OS for efficient simulations.

For the paper organization, we introduce the general FL minimization objective in Section 2. Subsequently, we provide a deep dive into our `FL_PyTorch` system in Section 3 and, finally, we demonstrate `FL_PyTorch` capabilities by concluding several experiments on multiple FL baselines and the most used FL dataset in Section 4.

2 FL OBJECTIVE AND FEDAVG

In this section, we introduce the FL objective in its general form

$$F(\boldsymbol{x}) = \mathbb{E}_{i\sim\mathcal{P}}[F_i(\boldsymbol{x})] \quad \text{where } F_i(\boldsymbol{x}) = \mathbb{E}_{\xi\in\mathcal{D}_i}[f_i(\boldsymbol{x},\xi)]. \tag{1}$$

The global objective F is an expectation over local objectives F_i over the randomness inherited from the client distribution $\mathcal{P}$, and the local objectives $\{F_i\}$ have the form of an expectation over the local datasets $\{\mathcal{D}_i\}$. In the case of the finite number of clients and local data points, both global objectives F and local losses $\{F_i\}$ can be written as simple weighted averages in the empirical risk minimization form (ERM).

The most common algorithm to solve (1) is federated averaging `FedAVG` [24]. This algorithm divides the training process into communication rounds. At the beginning of the t-th round ($t \geq 0$), the server broadcasts the current global model $\boldsymbol{x}^{(t)}$ to a random subset of clients $\mathcal{S}^{(t)}$ (often uniformly sampled without replacement in simulations). Then, each sampled client performs τ_i local SGD updates on its own local dataset and sends its local model update $\Delta_i^{(t)} = \boldsymbol{x}_i^{(t,\tau_i)} - \boldsymbol{x}^{(t)}$ to the server. Finally, the server uses the aggregated model updates to obtain the new global model as follows:

$$\boldsymbol{x}^{(t+1)} = \boldsymbol{x}^{(t)} + \frac{\sum_{i\in\mathcal{S}^{(t)}} p_i\Delta_i^{(t)}}{\sum_{i\in\mathcal{S}^{(t)}} p_i}. \tag{2}$$

where p_i is the relative weight of client i. The above procedure will repeat until the algorithm converges. In the *cross-silo* setting where all clients participate in the training at every round, we have $\mathcal{S}^{(t)} = \{1,2,\ldots,M\}$.

3 FL OPTIMIZATION SIMULATOR

FL_PyTorch is a system that has been built using Python programming language, and it is based on the DL framework PyTorch.

Its backbone consists of a general form of FedAVG displayed in Algorithm 1 partially inspired by Algorithm 1 in [33]. Our proposed general form preserves the standard structure of federated optimization where in each round t, subset $\mathcal{S}^{(t)}$ of all available clients is selected. As a next step, the master generates its state s_t which is broadcasted together with the current copy of the global solution $\boldsymbol{x}^{(t)}$ to the selected subset of clients $\mathcal{S}^{(t)}$. Afterwards, each participating client initializes its local solution $\boldsymbol{x}_i^{(t,0)}$ to be a copy of the global solution $\boldsymbol{x}^{(t)}$. Each of these clients then performs τ_i steps using the local optimizer ClientOpt with gradient estimated by the LocalGradient function. After this step, each client computes the model and state update, which are then sent back to the master which estimates the global update direction using the ServerGradient function. This estimate is used in ServerOpt that updates the global solution to its new value $\boldsymbol{x}^{(t+1)}$. Lastly, the server global state is updated. In Section 3.2, we show that this general scheme captures all standard algorithms, thus our scheme is sufficient and we believe that it gives researchers a sufficient level of freedom to develop and experiment with each component of our general scheme to push both practical and theoretical FL state-of-the-art.

Our current implementation is fully determined and allows us to configure up to 42 parameters which can be specified either through our easy to use GUI tool or directly via the command line that we discuss in the next subsection. These parameters can be grouped into 4 categories based on their function:

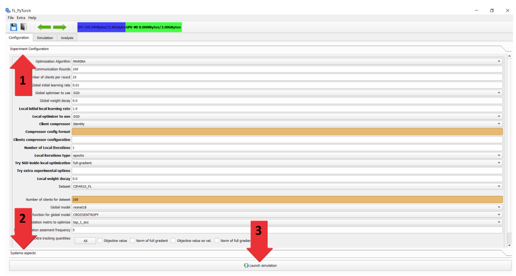
- *Server Optimizer*: the number of communication rounds T; the number of sampled clients per round; global initial local learning rate, global learning rate schedule, and global optimizer with its parameters such as momentum, the name of the algorithm to execute.
- *Local Optimizer*: the number of local epochs or local iterations τ_i; batch size for data loading; local optimizer with its parameters.
- *Model and Data*: model's and dataset's names.
- *System Setup*: directory to store run metadata, target compute devices, usage of remote compute devices, logging level, number of workers for loading dataset, random seed, thread pool sizes, experiments grouping, user defined comment.

3.1 Frontend

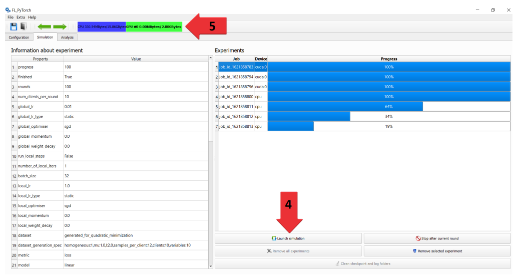
As we mention previously, FL_PyTorch supports two modes for running federated optimization– Graphical User Interface and Command Line Interface. Below, we discuss both of these in detail.

3.1.1 Graphical User Interface (GUI). We implemented our Graphical User Interface using the PyQt5 GUI framework [31]. This GUI framework supports all the standard desktop operating systems such as Windows, Linux, or macOS. In addition, the GUI part of FL_PyTorch has built-in VNC server support. If the target machine has not a native windows manager system one can use this mode and connect to GUI part via VNC Viewer software.

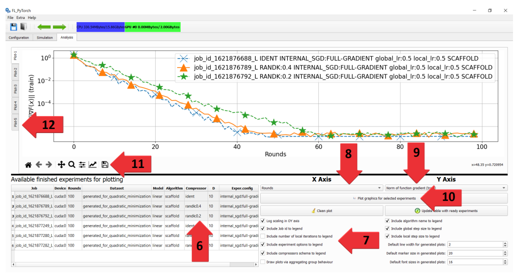
To get a better picture of our GUI, we display its main menu provided in Figure 1, (a), (b) and (c) are the 3 main components of our GUI and they help users to build their run setup with the steps depicted as numbered red arrows. 1.) experiment configuration, 2.) system setup configuration, 3.) thread pool setup, 4.) button to launch experiments, 5.) navigation pane, 6.) system monitoring, 7.) plotting setup – experiments selection, 8.) plotting setup – format 8.) plotting setup – x-axis, 9.) plotting setup – y-axis, 10.) plotting setup – generate plots, 11.) plotting setup – save option, 12.) plotting setup – clean output option.



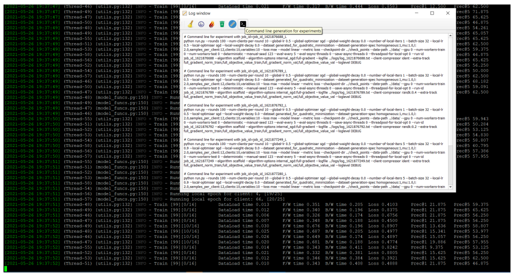
(a) The main tab of the `FL_PyTorch` 's GUI.



(b) The Progress tab of the `FL_PyTorch` 's GUI.


job_id_1621876688_L IDENT INTERNAL_SGD:FULL-GRADIENT global_lr:0.5 local_lr:0.5 SCAFFOLD
job_id_1621876789_L RANDK:0.4 INTERNAL_SGD:FULL-GRADIENT global_lr:0.5 local_lr:0.5 SCAFFOLD
job_id_1621876792_L RANDK:0.2 INTERNAL_SGD:FULL-GRADIENT global_lr:0.5 local_lr:0.5 SCAFFOLD
Rounds
Available finished experiments for plotting
X Axis
Y Axis
6
7
8
9
10
11
12

(c) The Analysis tab of the `FL_PyTorch` 's GUI.



(d) Console output during simulation and GUI log.

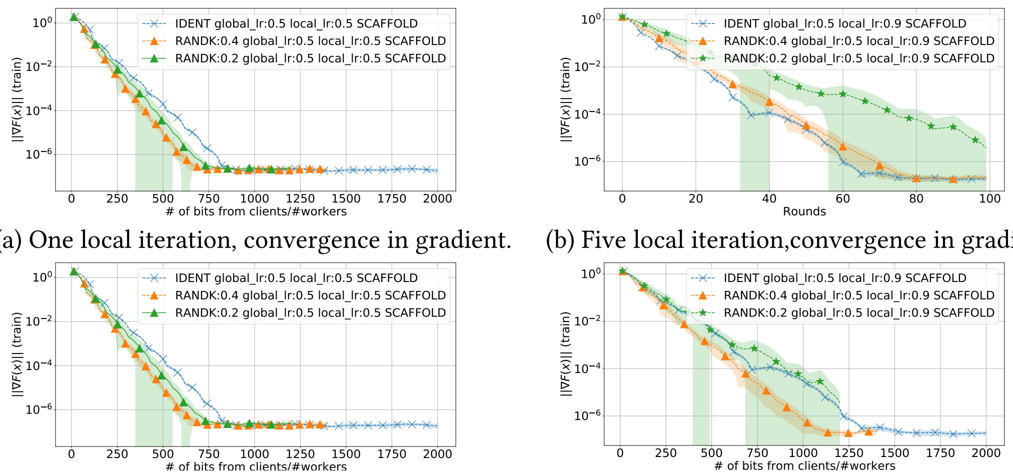
Fig. 1. Graphical User Interface of the simulator

3.1.2 *Command Line Interface (CLI).* Another alternative option to pass the run arguments is directly through the console via Command Line Interface (CLI). We also provide assistance to see the available arguments' options that can be accessed by launching `./run.sh` script provided in the supplementary materials with the`–help` option. To enable an efficient way to switch between GUI and CLI, we provide an option to recover arguments passed via the GUI tool in the command line format, displayed in Figure 1 (d). The CLI interface is used to instantiate remote workers in remote machines, for providing extra remote compute resources for the simulation process.

3.2 Optimization Algorithms

As discussed previously, our general level of abstraction introduced in Algorithm 1 allows for a sufficient level of freedom to implement standard and also more exotic FL optimization algorithms. In the current version, we implemented following state-of-the-arts methods considered in the literature: Distributed Compressed Gradient Descent (`DCGD`) [1, 15, 21] `FedAVG` [24], `SCAFFOLD` [20], `FedProx` [22] `DIANA` [26] and `MARINA` [11].

As an example, we show how Algorithm 1 is adjusted to `FedAVG` and `SCAFFOLD`. For `FedAVG`, both global state is an empty dictionary. Locally, we run τ_i iteration, which is usually set to be a constant T for the theoretical and size of the local dataset for the experiments. LocalGradient returns the unbiased stochastic gradient at $\boldsymbol{x}^{(t,k)}$ and ClientOpt is Stochastic Gradient Descent (`SGD`) with given step-size η_l. Similarly to the global state, local step update is none. ServerGradient is a (weighted) average of local updates and ServerOpt is `SGD` with fixed step-size 1. For the `SCAFFOLD`, there are 2 changes comparing to `FedAVG`. Firstly, the global state $s^{(t)}$ is a non-empty vector of the same dimension as $\boldsymbol{x}^{(t)}$ initialized as zero and each ClientOpt has its own local $s_i^{(t)}$, which is set to be a stochastic local gradient at $\boldsymbol{x}^{(t)}$. LocalGradient then returns the unbiased stochastic gradient at $\boldsymbol{x}^{(t,k)}$ shifted by $s_i^{(t)} - s_i^{(t)}$. Secondly, the local model update is sent to master together with the



(a) One local iteration, convergence in gradient. (b) Five local iteration,convergence in gradient.

(c) One local iteration, average communication. (d) Five local iterations, average communication.

Fig. 2. Function gradient diminishing for 1 and 5 local iterations of SCAFFOLD for quadratic minimization. Mean and variance have been estimated across 10 realizations.

local state update, which is set to the difference between current and previous communication round $U_i^{(t)} = s_i^{(t)} - s_i^{(t-1)}$. The ServerGlobalState is then updated using the average of $U_i^{(t)}$'s.

3.3 Supported Compressors

In federated learning (especially in the cross-device setting), clients may experience severe network latency and bandwidth limitations. Therefore, practical FL algorithms generally use a communication reduction mechanism to speed up the training procedure. Three common methods to reduce the communication cost are:

(1) to reduce the communication frequency by allowing local updates;
(2) to reduce communication traffic at the server by limiting the participating clients per round;
(3) to reduce communication volume by compressing messages.

We naturally support the first and the second option and we also added support for the compression. `FL_PyTorch` allows compressing messages both from the server to local clients and vice-versa. In the current version, we support the following compressor operators: Identical compressor (no compression), Lazy or Bernoulli compressor (update is communicated with probability p rescaled for the update to be unbiased), Rand-K (only random K coordinates are preserved uniformly at random rescaled for the update to be unbiased), Natural compressor [15], Standard dithering [1] and Natural Dithering [15].

3.4 Supported models(or pattern) structures

Current `FL_PyTorch` 's implementation allows users to experiment with the following list of image classification models: ResNets [14] (18, 34, 50, 101, 152), VGGs [35] (11, 13, 16, 19) and WideResNets [38] (28_2, 28_4, 28_8).

Besides DL models, we provide support for simple quadratic loss function to enable users to explore algorithms in this simplistic regime or debug their implementation. We synthetically generate our objective such that local loss is of the form

$$\min_{x\in\mathbb{R}^d} \frac{1}{n_i}\sum_{i=1}^{n_i} \|a_i^\top x - b_i\|^2. \tag{3}$$

Users are able to specify: dimensionality d, strong convexity parameter μ and L smoothness constant for (3) and the number of samples per each client. In addition, we consider two settings wherein the first case all clients optimizes the same objective, i.e., iid setting. In the second scenario, each client has a different objective to optimise, i.e., non-iid setting.

3.5 Tracking metrics and experiments analysis

An important feature to evaluate experiments is the ability to track all important quantities during the run of the algorithms. `FL_PyTorch` allows users to track the number of communication rounds, loss, accuracy, the norm of computed gradient, the norm of the full objective gradient, function values, number of gradient oracle calls, used GPU memory, number of bits which would be sent from workers to master in an actual real-world system. Furthermore, we provide a dedicated visualization tab in our GUI tool which allows user to load and visualize their experiments in an interactive fashion.

3.6 The internals of the system

To reach one of our goals of making `FL_PyTorch` computationally effective, we analyzed various aspects of the `PyTorch` infrastructure, including the place of `PyTorch` at NVIDIA computation stack, `PyTorch` initialization, its forward and backward computation, and speed of typical buses used in a local node setup. This analysis led us to the system design, which asynchronously exploits available hardware.

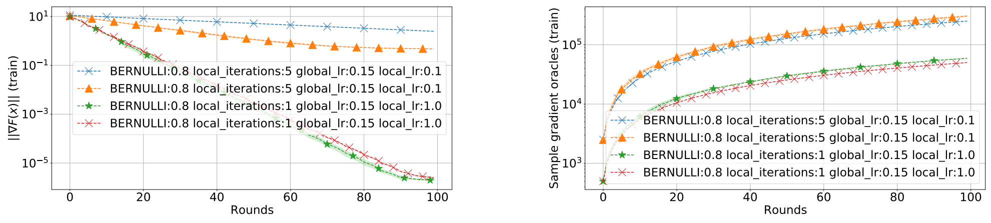
The Algorithm 1 has several independent parts, and instead of sequential execution of these parts in a simulated environment, independent parts can be partition across independent thread-pools of workers. In our implementation, each worker within a thread pool is a separate CPU thread that lives within a Python interpreter process launched in the operation system (OS) if the system's resources allow it. Since `PyTorch` and therefore `FL_PyTorch` allow launching of the computations in a GPUs, for providing separate CPU threads ability to work independently, each thread has its separate GPU CUDA stream to submit its computation work for GPU independently. By design, each worker with the same purpose is assigned to its task-specific thread pool with a thread-safe tasks queue. There are three specific types of tasks – deferred execution, process request for finish execution, and wait on the completion of all submitted work.

Finally, we stress that all this is happening "behind the curtain," and user interaction, including implementation of the new methods, is agnostic to the aforementioned implementation details.

3.7 Bringing custom Algorithm/ Model/ Data

As described in Algorithm 1, each algorithm supported by `FL_PyTorch`
requires implementation of InitializeServerState, ClientState, LocalGradient, ClientOpt, LocalState, ServerOpt, ServerGradient, ServerGlobalState. These are standard centralized-like `PyTorch` functions that need to be provided to `FL_PyTorch` in order to run simulations.

One example of adding new algorithm might be `SCAFFOLD` assuming that we are given implementation of `FedAvg` in the required Algorithm 1 format. As described in Section 3.2, one needs to set ClientState to return global shift as proposed in [20] and update LocalGradient to account for both global and local shifts. One further needs to define LocalState to return local shift update,



(a) Convergence in gradient (b) Gradient oracles

Fig. 3. Experiments with MARINA and DIANA algorithm with making local steps for quadratic minimization. Mean and variance have been estimated across 10 realizations.

see [20] for the detailed formula. The last modification is in SERVERGLOBALSTATE, which updates global shift using the local shift updates.

For the datasets, we require users to provide it in a form that is loadable via `PyTorch`'s `DataLoaders` and for the models, the required format is `PyTorch`'s `nn.Module`.

4 EXPERIMENTS

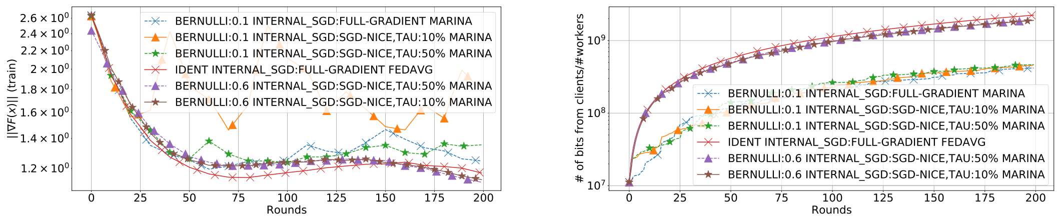
In this section, we provide several experiments to showcase potential use cases of `FL_PyTorch`. In general, `FL_PyTorch` can serve as a simulator for researchers to test and compare different algorithms, extend them and analyze each algorithmic component in a plug-and-play fashion as described in Section 3 and Algorithm 1.

For instance, one might be interested in how to reduce the communication burden of a given algorithm. In the literature, three approaches have been proposed. For the first approach, one performs local updates to reduce the frequency of the global model updates, thus also reducing communication frequency. By this strategy, communication cost per client model update can be effectively reduced by a number of local steps. However, while often in practice this leads to improvement, it is not always guaranteed, as it was theoretically shown that there is a trade-off between communication reduction and convergence [7]. Another possibility is to employ compression techniques that reduce the bits transmitted between the clients and the server. This compression operator should be selected carefully as it comes with an extra variance and it might affect the convergence. The third option is to employ partial participation for which in each round only sampled subset of clients participates. There is also a trade-off for this strategy as although having only a few clients in each round has a positive effect on reducing communication, it might negatively affect the quality of the obtained solution due to inexact updates based only on sub-sampling. `FL_PyTorch` supports experimentation with all 3 approaches and below, we include 3 such examples.

4.1 SCAFFOLD with Compression

In the first example, we used our framework to analyze whether SCAFFOLD can work with compression, i.e., whether both $(\Delta_i^{(t)}, U_i^{(t)})$ can be compressed, without a significant decrease in performance.

This experiment was carried on a synthetically generated quadratic minimization problem 3. We set the dimensionality of the problem to be 20. Both features a_i's and responses b_i are generated using uniform distribution on the interval $[0, 1]$. After this step, we update the data matrix such that the objective is L-smooth with $L = 2$ and strongly convex with $\mu = 1$ using singular value decomposition (SVD). We generate 10 clients and consider a full participation scenario. The number of communications rounds is chosen to be 100. We set the global learning rate to 0.5 and the



(a) Convergence in gradient (b) Average communication from clients to master

Fig. 4. Experiments with MARINA and DIANA algorithm with making local steps for quadratic minimization

local learning rate to 1.0. For the compressor, we choose RAND-K, with 3 values of K: 100%(no compression), 40% and 20%. We provide results of this experiment in Figure 2. One can observe that dropping 60% coordinates at random has minimal effect on the convergence with respect to iterates. Dropping 80% brings a visible slow-down in per iterate convergence, but it is comparable with respect to the number of communicated bits. This experiment demonstrates that it may be worthwhile to consider an extension of `SCAFFOLD` by adding compression.

4.2 Benefits of Local Updates

For the second experiment, we consider a similar setup as for the previous experiment. We set smoothness constant L to 5 and strong convexity parameter $\mu = 1$. As a compressor we select Lazy/Bernoulli compressor described below4 with $p = 0.8$.

$$C(x) = \begin{cases} x/p, & \text{with probability } p \\ 0, & \text{with probability} 1 - p. \end{cases} \tag{4}$$

We run `MARINA` and `DIANA` algorithms with full gradient estimation for 100 rounds with 10 clients and full participation. These methods were not analyzed in the setting with several local iterations that motivates us to consider these methods for this experiment.

Looking into Figure 3, we can observe that both `MARINA` and `DIANA` do not benefit from extra local updates. We also experimented with different problem condition numbers (L/μ), but it seems that local iterations do not speed up the convergence in the strongly convex regime. This might suggest that a naive combination of these algorithms with local updates will not lead to theoretical benefits, too.

4.3 Stochastic Updates and Compression

For the third example, we run `MARINA` and choose the model to be ResNet18 and for the dataset, federated version of CIFAR10 split uniformly at random (u.a.t.) among 100 clients. We run 200 communications rounds and in each round, we randomly select 25 clients. The local learning rate is 1.0 and the global learning rate is 0.001. We perform one local iteration, i.e., $\tau_i = 1$.

We investigate the effect of the combined effect of compression and stochastic gradient estimation since both of these approaches introduce extra variance that might negatively impact the quality of the obtained solution.

For gradient estimation, we employ `SGD-NICE` sampling strategy that estimates gradient via selecting uniformly at random a subset of samples of fixed size. TAU in the experiments reflects the relative subset size with respect to the total size of the local dataset. The compressor operator is Bernoulli compression. Looking into Figure 4, one can see that having Lazy compression with $p = 0.1$ combined with `SGD-NICE` with $TAU = 10\%$ hurts the convergence. On the other hand, having a less aggressive compressor or using a bigger sample size for `SGD-NICE` does not deteriorate the performance.

5 CONCLUSIONS

In this work, we have introduced `FL_PyTorch`, an efficient FL simulator based on `PyTorch` that enables FL researchers to experiment with optimization algorithms to advance current state-of-the-art (SOTA). `FL_PyTorch` is an easy to use tool that supports SOTA FL algorithms and the most used image classification FL datasets. In addition, `FL_PyTorch` uses several levels of parallelism for efficient execution while being simple to extend using only standard `PyTorch` abstractions. In its basic version, it does not require any programming and all its pre-implemented components are accessible through its GUI which allows to set up, run, monitor and evaluate different FL methods.

For the future, we plan to provide users with more pre-implemented algorithms, datasets, and models. We also aim to extend our visualization tool. After open-sourcing `FL_PyTorch`, we expect a significant increase in the number of supported algorithms, datasets, and models via inputs from the FL research community.

REFERENCES

[1] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2017. QSGD: Communication-efficient SGD via gradient quantization and encoding. *Advances in Neural Information Processing Systems* 30 (2017), 1709–1720.

[2] Apple. 2017. Learning with Privacy at Scale. In *Differential Privacy Team Technical Report.*

[3] Apple. 2019. Designing for Privacy (video and slide deck). Apple WWDC, https://developer.apple.com/videos/play/wwdc2019/708.

[4] James Henry Bell, Kallista A. Bonawitz, Adrià Gascón, Tancrède Lepoint, and Mariana Raykova. 2020. Secure Single-Server Aggregation with (Poly)Logarithmic Overhead. In *CCS.* ACM, 1253–1269.

[5] Daniel J Beutel, Taner Topal, Akhil Mathur, Xinchi Qiu, Titouan Parcollet, and Nicholas D Lane. 2020. Flower: A Friendly Federated Learning Research Framework. *arXiv preprint arXiv:2007.14390* (2020).

[6] Sebastian Caldas, Peter Wu, Tian Li, Jakub Konečný, H Brendan McMahan, Virginia Smith, and Ameet Talwalkar. 2018. Leaf: A benchmark for federated settings. *arXiv preprint arXiv:1812.01097* (2018).

[7] Zachary Charles and Jakub Konečný. 2021. Convergence and Accuracy Trade-Offs in Federated Learning and Meta-Learning. *arXiv preprint arXiv:2103.05032* (2021).

[8] Cynthia Dwork. 2008. Differential privacy: A survey of results. In *International conference on theory and applications of models of computation.* Springer, 1–19.

[9] European Commission. [n. d.]. GDPR: 2018 Reform of EU Data Protection Rules. https://ec.europa.eu/commission/sites/beta-political/files/data-protection-factsheet-changes_en.pdf

[10] Google. 2021. Your voice and audio data stays private while Google Assistant improves. https://support.google.com/assistant/answer/10176224.

[11] Eduard Gorbunov, Konstantin Burlachenko, Zhize Li, and Peter Richtárik. 2021. MARINA: Faster Non-Convex Distributed Learning with Compression. *arXiv preprint arXiv:2102.07845* (2021).

[12] Andrew Hard, Kanishka Rao, Rajiv Mathews, Françoise Beaufays, Sean Augenstein, Hubert Eichner, Chloé Kiddon, and Daniel Ramage. 2018. Federated Learning for Mobile Keyboard Prediction. *arXiv preprint arXiv:1811.03604* (2018).

[13] Chaoyang He, Songze Li, Jinhyun So, Xiao Zeng, Mi Zhang, Hongyi Wang, Xiaoyang Wang, Praneeth Vepakomma, Abhishek Singh, Hang Qiu, Xinghua Zhu, Jianzong Wang, Li Shen, Peilin Zhao, Yan Kang, Yang Liu, Ramesh Raskar, Qiang Yang, Murali Annavaram, and Salman Avestimehr. 2020. FedML: A Research Library and Benchmark for Federated Machine Learning. *arXiv preprint arXiv:2007.13518* (2020). https://fedml.ai

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition.* 770–778.

[15] Samuel Horvath, Chen-Yu Ho, Ludovit Horvath, Atal Narayan Sahu, Marco Canini, and Peter Richtarik. 2019. Natural compression for distributed deep learning. *arXiv preprint arXiv:1905.10988* (2019).

[16] Samuel Horvath, Stefanos Laskaridis, Mario Almeida, Ilias Leontiadis, Stylianos I Venieris, and Nicholas D Lane. 2021. FjORD: Fair and Accurate Federated Learning under heterogeneous targets with Ordered Dropout. *arXiv preprint arXiv:2102.13451* (2021).

[17] Alex Ingerman and Krzys Ostrowski. 2019. *TensorFlow Federated.* https://medium.com/tensorflow/introducing-tensorflow-federated-a4147aa20041

[18] Intel®. 2021. Intel® Open Federated Learning. https://github.com/intel/openfl

[19] Intel and Consilient. 2020. Intel and Consilient Join Forces to Fight Financial Fraud with AI. https://newsroom.intel.com/news/intel-consilient-join-forces-fight-financial-fraud-ai/.

[20] Sai Praneeth Karimireddy, Satyen Kale, Mehryar Mohri, Sashank Reddi, Sebastian Stich, and Ananda Theertha Suresh. 2020. SCAFFOLD: Stochastic controlled averaging for federated learning. In *International Conference on Machine Learning*. PMLR, 5132–5143.

[21] Sarit Khirirat, Hamid Reza Feyzmahdavian, and Mikael Johansson. 2018. Distributed learning with compressed gradients. *arXiv preprint arXiv:1806.06573* (2018).

[22] Tian Li, Anit Kumar Sahu, Manzil Zaheer, Maziar Sanjabi, Ameet Talwalkar, and Virginia Smith. 2018. Federated optimization in heterogeneous networks. *arXiv preprint arXiv:1812.06127* (2018).

[23] Yanjun Ma, Dianhai Yu, Tian Wu, and Haifeng Wang. 2019. PaddlePaddle: An Open-Source Deep Learning Platform from Industrial Practice. *Frontiers of Data and Domputing* 1, 1 (2019), 105–115.

[24] H Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. 2017. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*. 1273–1282. Initial version posted on arXiv in February 2016.

[25] MELLODDY. 2020. MELLODDY Project Meets Its Year One Objective: Deployment Of The World's First Secure Platform For Multi-Task Federated Learning In Drug Discovery Among 10 Pharmaceutical Companies. https://www.melloddy.eu/y1announcement.

[26] K Mishchenko, E Gorbunov, M Takáč, and P Richtárik. 2019. Distributed Learning with Compressed Gradient Differences. CoRR, abs/2019.09269, 2019.

[27] NVIDIA. 2019. NVIDIA Clara. https://developer.nvidia.com/clara

[28] NVIDIA. 2020. Triaging COVID-19 Patients: 20 Hospitals in 20 Days Build AI Model that Predicts Oxygen Needs. https://blogs.nvidia.com/blog/2020/10/05/federated-learning-covid-oxygen-needs/.

[29] Owkin. 2020. Story of the 1st Federated Learning Model at Owkin. https://owkin.com/federated-learning/federated-model/.

[30] Adam et al. Paszke. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[31] PyQT. [n. d.]. ([n. d.]).

[32] Swaroop Ramaswamy, Rajiv Mathews, Kanishka Rao, and Françoise Beaufays. 2019. Federated Learning for Emoji Prediction in a Mobile Keyboard. *arXiv preprint arXiv:1906.04329* (2019).

[33] Sashank Reddi, Zachary Charles, Manzil Zaheer, Zachary Garrett, Keith Rush, Jakub Konečnỳ, Sanjiv Kumar, and H Brendan McMahan. 2020. Adaptive federated optimization. *arXiv preprint arXiv:2003.00295* (2020).

[34] Theo Ryffel, Andrew Trask, Morten Dahl, Bobby Wagner, Jason Mancuso, Daniel Rueckert, and Jonathan Passerat-Palmbach. 2018. A generic framework for privacy preserving deep learning. *arXiv preprint arXiv:1811.04017* (2018).

[35] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[36] WeBank. 2020. Utilization of FATE in Anti Money Laundering Through Multiple Banks. https://www.fedai.org/cases/utilization-of-fate-in-anti-money-laundering-through-multiple-banks/.

[37] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. 2019. Federated machine learning: Concept and applications. *ACM Transactions on Intelligent Systems and Technology (TIST)* 10, 2 (2019), 1–19.

[38] Sergey Zagoruyko and Nikos Komodakis. 2016. Wide residual networks. *arXiv preprint arXiv:1605.07146* (2016).

A FL_PYTORCH: TEMPLATE METHODS OF ALGORITHM 1

Here we present an overview of templated methods under research responsibilities for Algorithm 1. For subtle technical details please familiarize yourself with the provided readme, tutorial, automatically generated code documentation in the code repository, and well-documented code.

Table 1. Brief description of template methods of Algorithm 1

Template Method	Description
InitializeServerState	This method should return a dictionary that initializes the server state. The method obtains as dimension of the problem, and the constructed and initialized model.
ClientState	By our design client state is stateless. The client state is instantiated at the beginning of each round for each of the selected clients. You should reconstruct client state based on initialized or updated server state.
LocalGradient	This method should evaluate the local gradient that is optimization algorithm specific.
ClientOpt	Local classical optimizer provided out of the box by `PyTorch` used in client for local steps.
LocalState	The default implementation of this step in presented in some sense in *local_training* method. This step happens automatically, and in rare cases there is a need for customization this step.
ServerOpt	Classical optimizer provided out of the box by `PyTorch` used in sever for global steps.
ServerGradient	Server gradient is the method that estimates the direction of the global server model update, which should return a flat vector with d elements.
ServerGlobalState	This logic is dedicated of the global server state update. This method as input obtains collected and ready to use information about clients responses and clients in that communication round, previous global model and new model with updates parameters model.

B FL_PYTORCH: INTERNAL MECHANISMS

In this section, we would like to depict FL_PyTorch main components in a schematic way.

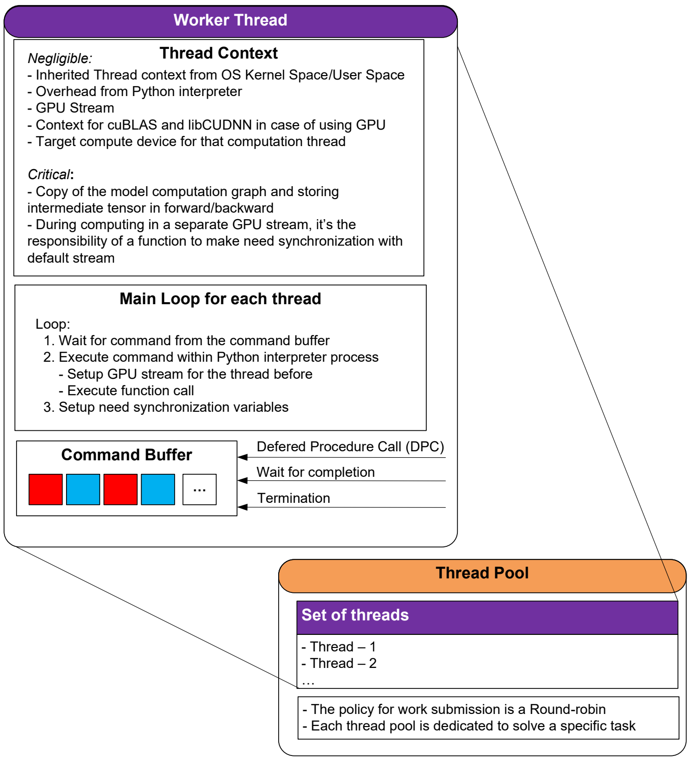

Worker Thread
Thread Context
Negligible:
- Inherited Thread context from OS Kernel Space/User Space
- Overhead from Python interpreter
- GPU Stream
- Context for cuBLAS and libCUDNN in case of using GPU
- Target compute device for that computation thread
Critical:
- Copy of the model computation graph and storing intermediate tensor in forward/backward
- During computing in a separate GPU stream, it's the responsibility of a function to make need synchronization with default stream
Main Loop for each thread
Loop:
1. Wait for command from the command buffer
2. Execute command within Python interpreter process
- Setup GPU stream for the thread before
- Execute function call
3. Setup need synchronization variables
Command Buffer
...
Defered Procedure Call (DPC)
Wait for completion
Termination
Thread Pool
Set of threads
- Thread – 1
- Thread – 2
…
- The policy for work submission is a Round-robin
- Each thread pool is dedicated to solve a specific task

Fig. 5. A single worker thread structure and it's role in a thread pool.

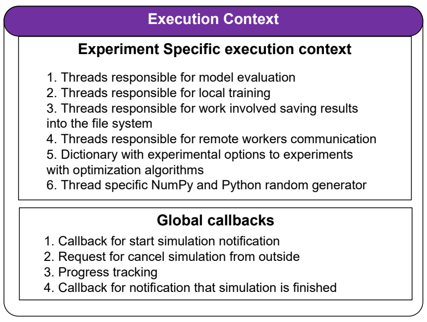

Execution Context
Experiment Specific execution context
1. Threads responsible for model evaluation
2. Threads responsible for local training
3. Threads responsible for work involved saving results into the file system
4. Threads responsible for remote workers communication
5. Dictionary with experimental options to experiments with optimization algorithms
6. Thread specific NumPy and Python random generator
Global callbacks
1. Callback for start simulation notification
2. Request for cancel simulation from outside
3. Progress tracking
4. Callback for notification that simulation is finished

Fig. 6. FL_PyTorch execution context for a single experiment. The GUI can handle several experiments at the same time.

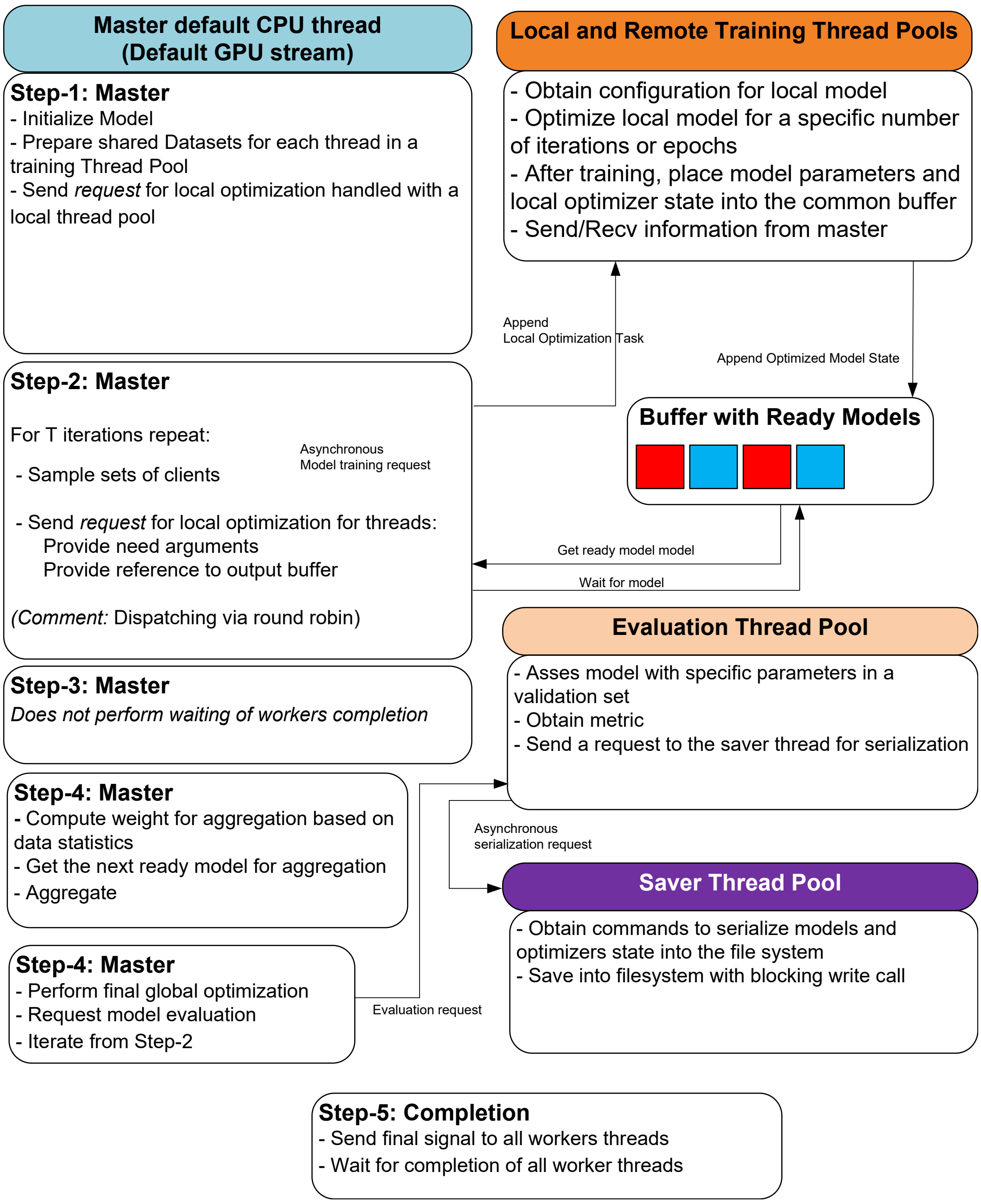

Master default CPU thread
(Default GPU stream)
Step-1: Master
- Initialize Model
- Prepare shared Datasets for each thread in a training Thread Pool
- Send *request* for local optimization handled with a local thread pool
Step-2: Master
For T iterations repeat:
- Sample sets of clients
- Send *request* for local optimization for threads:
Provide need arguments
Provide reference to output buffer
(*Comment:* Dispatching via round robin)
Step-3: Master
Does not perform waiting of workers completion
Step-4: Master
- Compute weight for aggregation based on data statistics
- Get the next ready model for aggregation
- Aggregate
Step-4: Master
- Perform final global optimization
- Request model evaluation
- Iterate from Step-2
Step-5: Completion
- Send final signal to all workers threads
- Wait for completion of all worker threads
Local and Remote Training Thread Pools
- Obtain configuration for local model
- Optimize local model for a specific number of iterations or epochs
- After training, place model parameters and local optimizer state into the common buffer
- Send/Recv information from master
Append Local Optimization Task
Append Optimized Model State
Asynchronous Model training request
Buffer with Ready Models
Get ready model model
Wait for model
Evaluation Thread Pool
- Asses model with specific parameters in a validation set
- Obtain metric
- Send a request to the saver thread for serialization
Asynchronous serialization request
Saver Thread Pool
- Obtain commands to serialize models and optimizers state into the file system
- Save into filesystem with blocking write call
Evaluation request

Fig. 7. Communication between different threads during Algorithm 1 execution

C COMPARISON TO RELATED FRAMEWORKS

Table 2. Comparison of `FL_PyTorch`, `FedML`, and `Flower` in terms of important functionality.

#	Comparison criteria	`FL_PyTorch`	`FedML.ai`	`Flower.dev`
1	Home page	...	fedml.ai	flower.dev
2	Support of stochastic compressors	Yes	No	No
3	Support of local steps	Yes	Yes	Yes
4	Create plots w/ (w/0) internet connection	Yes (Yes)	Yes (No)	Yes (No)
5	Create highly customizable plots	Easy	Harder	Harder
7	Serialization of the results from numerical experiments	Yes	No	No
8	Support of standalone and distributed mode	Yes	Yes	Yes
9	The communication protocols for clients in multi-node setup	TCP/IP	MPI, gRPC	gRPC
10	Synthetically controlled optimization problems	Yes	No	No
11	Number of supported models and datasets	Modest	High	High
12	Client paralelization in a single GPU	Yes	No	No
13	Debugging	Easy[1]	Distributed is hard	Distributed is hard
14	Parallelization across several GPUs (standalone)	Yes	No	No
15	GUI interface for launch experiments	Yes[2]	No	No
16	Console interface for launching experiments	Yes	Yes	Yes
17	Automatic testing	Yes[3]	No	No
18	Built-in mechanism to load/save experiments	Yes	No	No

[1]computations can be reduced to a single thread systems setup
[2]PyQt5 based cross-platform GUI compatible with macOS, Linux, and Windows
[3]Unit-tests with *pytest*