

# EasyFL: A Low-code Federated Learning Platform For Dummies

Weiming Zhuang, *Student Member, IEEE*, Xin Gan, Yonggang Wen, *Fellow, IEEE*, and Shuai Zhang

**Abstract**—Academia and industry have developed several platforms to support the popular privacy-preserving distributed learning method – Federated Learning (FL). However, these platforms are complex to use and require a deep understanding of FL, which imposes high barriers to entry for beginners, limits the productivity of researchers, and compromises deployment efficiency. In this paper, we propose the first low-code FL platform, *EasyFL*, to enable users with various levels of expertise to experiment and prototype FL applications with little coding. We achieve this goal while ensuring great flexibility and extensibility for customization by unifying simple API design, modular design, and granular training flow abstraction. With only a few lines of code, EasyFL empowers them with many out-of-the-box functionalities to accelerate experimentation and deployment. These practical functionalities are heterogeneity simulation, comprehensive tracking, distributed training optimization, and seamless deployment. They are proposed based on challenges identified in the proposed FL life cycle. Compared with other platforms, EasyFL not only requires just three lines of code (at least 10x lesser) to build a vanilla FL application but also incurs lower training overhead. Besides, our evaluations demonstrate that EasyFL expedites distributed training by 1.5x. It also improves the efficiency of deployment. We believe that EasyFL will increase the productivity of researchers and democratize FL to wider audiences.

**Index Terms**—Federated learning, distributed training, federated learning platform, machine learning system

## I. INTRODUCTION

FEDERATED learning (FL) is attracting considerable attention in recent years. It is a new distributed training technique that provides in-situ model training on decentralized edges. FL has empowered a wide range of applications with privacy-preserving mechanism, including healthcare applications [6]–[8], consumer products [9], [10], recommendation systems [11], [12], and video surveillance [13]. Companies and institutions are exploring these applications mostly in experimental environments [7], [12]. Some of them are building prototypes and then deploying FL applications [14], [15].

However, AI engineering [16] of FL requires tremendous resources and efforts, regardless of levels of expertise. Beginners who are interested in FL face high barriers to entry due to the complex setup and non-trivial concepts. Researchers who lack practical experience in FL would need two to three weeks to build a vanilla FL application from scratch. More advanced features like comprehensive tracking and application-specific optimizations are even more time-consuming, hindering their productivity. Experienced researchers who studied FL would face challenges when building prototypes and deploying FL applications because they may not be familiar with software engineering and infrastructure. Communication and collabora-

tion with software engineers who do not know FL would be a hassle.

Existing FL platforms and frameworks from academia and industry are complex to use and require a deep understanding of FL. Most existing FL systems need at least 100 lines of code to implement a vanilla FL application, as shown in Table I (links of source codes are provided in Appendix A). LEAF [1] is the first FL benchmark, but users need to replicate and learn the whole library to start experimentation. Although TensorFlow Federated (TFF) [4] is a research-oriented platform with two layers of APIs, it imposes prerequisite on knowledge in Tensorflow [17] and Keras [18]. PySyft [2] focuses on secure and private deep learning, which is not straightforward for users interested in FL. All these platforms lack the deployability to build FL prototypes and run FL in production. Industrial FL frameworks such as Federated AI Technology Enabler (FATE) [5] and PaddleFL [3] are deployable with containerization, but they are unfriendly to beginners and researchers due to complex environment setup and heavy system design.

In this paper, we propose a new paradigm of FL system, a *low-code* FL platform termed *EasyFL*, to enable users with various levels of expertise to experiment and prototype FL applications with little coding. Using our platform, beginners can start experiencing FL with only three lines of code, even without prior knowledge of FL. For researchers who are addressing the advanced and open problems in FL [19], we provide them with sufficient flexibility to develop new algorithms with minimal coding by reusing the majority of FL architecture. EasyFL empowers experienced researchers to easily prototype and seamlessly deploy FL applications without further engineering.

Although users write less code, we analyze the challenges in the FL life cycle and empower them with more out-of-box functionalities. We summarize and compare these functionalities with other platforms in Table I. Specifically, to facilitate ease of experimentation and development, EasyFL supports heterogeneity simulation, training flow abstraction, and comprehensive tracking: 1) The out-of-the-box heterogeneity simulation, only partially supported by other platforms, enables researchers to easily start exploring the most important challenges in FL — statistical heterogeneity and system heterogeneity [20]; 2) Instead of using logs to collect results like most other platforms, we design a hierarchical tracking system to organize training metrics for easier result analysis; 3) EasyFL is the first platform that abstracts FL training flow to granular stages for minimum coding to develop new algorithms. Moreover, we exclusively support

TABLE I

COMPARE OUR PROPOSED EASYFL WITH EXISTING FL LIBRARIES, PLATFORMS, AND FRAMEWORKS. EASYFL REQUIRES LITTLE CODING AND PROVIDES MORE OUT-OF-THE-BOX FUNCTIONALITIES TO IMPROVE THE PRODUCTIVITY OF RESEARCHERS. ○ MEANS LIMITED SUPPORT.

	LEAF [1]	PySyft [2]	PaddleFL [3]	TFF [4]	FATE [5]	EasyFL (Ours)
Lines of Code (Vanilla FL App)	~400	~190	~190	~30	~100	3
Heterogeneity Simulation	○	×	○	○	○	✓
Training Flow Abstraction	×	×	×	×	×	✓
Distributed Training Optimization	×	×	×	×	×	✓
Tracking	×	×	×	○	✓	✓
Deployability	×	×	✓	○	✓	✓

distributed training optimization to accelerate and scale FL training. Last but not least, EasyFL provides seamless and scalable deployment of centralized topology-based FL systems with containerization and service discovery, whereas existing platforms either focus on the deployment of multi-party FL (FATE [5] and PaddleFL [3]) or have incomplete deployment features (TFF [4]).

Our evaluation demonstrates that EasyFL effectively reduces the lines code for developing FL applications and accelerates experimentation and deployment. We implement three FL applications with 4.5x to 9.5x fewer codes than the original implementations. Despite that we provide many out-of-the-box functionalities through abstractions, EasyFL has a lower training overhead compared with other popular FL libraries. EasyFL further accelerates distributed training by 1.5x under limited hardware resources. It also allows sub-linear time to deployment and maintains performance in production. We believe that EasyFL will lower the barriers to entry for beginners, increase the productivity of researchers, and bridge the gap between researchers and engineers.

In summary, we make the following contributions:

- We define the FL life cycle and propose the system requirements by analyzing challenges in the life cycle.
- We design and build the first *low-code* FL system, EasyFL, to democratize FL to wider audiences. Complex system implementations are shielded for users, regardless of expertise levels. We unify simple API design and granular training flow abstractions to achieve the low-code capability. Besides, we further facilitate ease of experimentation and development with out-of-the-box heterogeneity simulation and comprehensive tracking.
- We provide sufficient flexibility and extensibility through abstraction and plugin architecture for experienced researchers to easily develop new algorithms and applications to address open problems in FL.
- We propose distributed training optimization to accelerate and scale FL experimentation.
- We support seamless and scalable deployment for fast prototyping and training in production.

The rest of the paper is organized as follows. In Section II, we introduce the background of FL and review related work. We define and analyze the FL life cycle to illustrate the system requirements in Section III. Section IV presents the high-level overview of system design and architecture of EasyFL. In Section V, we introduce the modules to facilitate ease of experimentation and development. We propose distributed

training optimization in Section VI and present seamless and scalable deployment in Section VII. In Section VIII, we provide the evaluation of EasyFL. Section IX summarizes this paper and provides future directions.

## II. BACKGROUND AND RELATED WORK

Federated learning (FL) is a distributed learning paradigm where multiple clients train machine learning (ML) models with the coordination of a central server [19]. Federated Averaging (FedAvg) [21] is a standard FL algorithm. It completes the training with the following three steps: (1) the server selects a fraction of clients and sends them a global model  $w$ ; (2) each client  $k$  trains the model using their datasets for  $E$  local epochs and uploads updates  $w_k$ . (3) the server aggregates these updates to obtain a new global model. EasyFL implements FedAvg as the default algorithm.

**Federated Learning Platforms** Institutions and companies have developed several experimentation-oriented and industrial-level FL platforms and frameworks [1]–[5]. Although experimentation-oriented libraries like LEAF [1] provide sample implementations and simulate statistical heterogeneity with federated datasets, they are lack of deployability. TFF [4] is a deployable research-oriented platform, but it does not optimize distributed training. Although industrial-level frameworks like FATE [5] supports deployment with containers, they are not user-friendly due to high barriers of entry caused by heavy system design. Our proposed low-code platform, EasyFL, is user-friendly and supports efficient experimentation and seamless deployment.

**Heterogeneity in FL** Statistical and system heterogeneity are the key challenges of FL [20]. Statistical heterogeneity is sourced from non-independent and identical distributed (non-IID) and unbalanced data. Many studies propose novel approaches to address it using federated datasets [13], [22], [23]. System heterogeneity is caused by varieties of hardware and networking conditions of clients. Existing system heterogeneity simulations are either hardware-based [24] that are complex and resource-demanding or software-based [23], [25] that are not generic enough. EasyFL simulates statistical heterogeneity with the three most commonly used datasets (Table III) and simulates system heterogeneity in a lightweight manner.

**Distributed ML** Distributed ML is a common practice to train a large-scale model in data centers, using model parallelism and data parallelism [26]. FL differs from distributed ML because it faces unique challenges of expensive communication, statistical and system heterogeneity, and



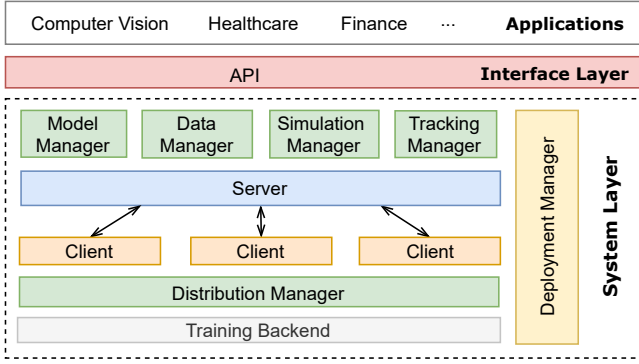


Fig. 2. EasyFL's architecture consists of two parts: an interface player providing simple APIs for applications and a modularized system layer providing flexibility, extensibility, and out-of-the-box functionalities.

the communication channels and the training flow. It provides standalone training by default, optimizes distributed training (Section VI), and supports remote training by transmitting messages through remote communication (Section VII).

**Seamless and Scalable Deployment** Deployment, the last step to run FL in production, is also challenging because of diverse computing environments and the potentially large number of clients. Firstly, the production environment of FL clients could have large variances on the hardware and operating systems [20]. Manually adapting and deploying is time-consuming and unscalable. Moreover, new FL algorithm development usually needs multiple iterations of deployment because the algorithm verified in the experimental environment may not work perfectly well in the production environment. Secondly, to scale up the number of clients in training in production, the server must be aware of the addresses of clients. Using static configuration is a straightforward approach to manually record addresses of clients on the server. However, this method is unstable and unscalable when the number of clients is large because it is common that existing clients would drop out and new clients would join [31].

To tackle these challenges, EasyFL supports seamless and scalable deployment with containerization and service discovery (Section VII).

#### IV. SYSTEM DESIGN AND ARCHITECTURE

In this section, we first provide a high-level overview of the system design and architecture of EasyFL (Fig. 2). Then, we present simple designs of APIs (Table II).

##### A. Overview

EasyFL's architecture comprises of an *interface layer* and a modularized *system layer*. The interface layer provides simple APIs for high-level applications and the system layer has complex implementations to accelerate training and shorten deployment time with out-of-the-box functionalities.

**Interface Layer** The interface layer, providing a common interface across FL applications, is one of the key designs that enable low-code capability. It contains APIs that are designed to encapsulate complex system implementations from users. These APIs decouple application-specific models, datasets,

TABLE II  
EASYFL HAS THREE CATEGORIES OF APIs: INITIALIZATION, REGISTRATION, AND EXECUTION.

Name	Description
<code>init(configs)</code>	Initialize EasyFL with configurations
<code>register_dataset(train, test)</code>	Register an external dataset
<code>register_model(model)</code>	Register an external model
<code>register_server(server)</code>	Register a customized server
<code>register_client(client)</code>	Register a customized client
<code>run(callback)</code>	Start training with an optional callback
<code>start_server(args)</code>	Start server service for remote training
<code>start_client(args)</code>	Start client service for remote training

and algorithms such that EasyFL is generic to support a wide range of applications like computer vision and healthcare.

**System Layer** The system layer supports and manages the FL life cycle. It consists of eight modules to support FL training pipeline and life cycle: 1) The *simulation manager* initializes the experimental environment with heterogeneous simulations. 2) The *data manager* loads training and testing datasets, and the *model manager* loads the model. 3) A *server* and the *clients* start training and testing with FL algorithms such as FedAvg [21]. 4) The *distribution manager* optimizes the training speed of distributed training. 5) The *tracking manager* collects the evaluation metrics and provides methods to query training results. 6) The *deployment manager* seamlessly deploys FL and scales FL applications in production.

The system layer embraces coarse-grained plugin architecture with modular design and fine-grained plugin design with training flow abstraction: Modularity provides users with the flexibility to extend to new datasets and models with application-specific plugins; Training flow abstraction allows users to customize federated algorithms by replacing specific training stages in the server and client modules.

##### B. API

Table II summarizes the most important APIs of EasyFL. We introduce below these APIs in three categories (initialization, registration, and execution), and illustrate their usage scenarios.

`init(configs)`: Initialize EasyFL with provided configurations (`configs`) or default configurations if not specified. It instructs the simulation manager to coordinate with the data manager to set up the simulation environment for experiments. Besides, these configurations determine the training hardware and hyperparameters for both experimental and production scenarios.

`register_module`: Register customized modules to the system. EasyFL supports the registration of customized datasets, models, server, and client, replacing the default modules in FL training. In the experimental phase, users can register newly developed algorithms to understand their performance. In the production phase, users can use it to adapt to real-world datasets.

`run, start_server/client`: The APIs are commands to trigger execution. `run(callback)` starts FL using standalone training or distributed training, with an optional

callback to define execution after training is completed. `start_server` and `start_client` start the server and client services to communicate remotely with `args` variables for configurations specific to remote training, such as the endpoint addresses.

```
# --- Example 1: Quick start ---
configs = {"model": "resnet18"} #optional
easyfl.init(configs) #initialization
easyfl.run() #start training

# --- Example 2: Remote training ---
# Start customized remote server
easyfl.register_server(NewServer) #optional
easyfl.init() #use default configs
easyfl.start_server(args) #start service
# Start customized client server
easyfl.register_client(NewClient) #optional
easyfl.init() #use default configs
easyfl.start_client(args) #start service
```

Listing 1. Usage examples of EasyFL.

These APIs empower users with various levels of expertise to conduct FL experiments or build FL prototypes with minimal coding, as shown in Listing 1. For example, beginners can quickly start FL with no more than three lines of code (Example 1); To build prototypes, experienced researchers can customize federated algorithms in the server and client modules by registering new server and client, and start the services to conduct remote training (Example 2). We further explain how to achieve low-code implementations of the server and client in Section V-B.

## V. EASE OF EXPERIMENTATION AND DEVELOPMENT

To facilitate ease of experimentation and development, EasyFL provides heterogeneity simulation for fast simulation setup, training flow abstraction for minimum coding, and comprehensive tracking for easy results analysis.

### A. Heterogeneity Simulation

To minimize researchers' hassles in setting up the simulation environment to start experiments, EasyFL provides various methods to simulate statistical and system heterogeneity using the simulation manager and the data manager. Users can easily change configurations in initialization to customize simulation methods.

**Statistical Heterogeneity** To simulate statistical heterogeneity, which is closely related with datasets, EasyFL includes three most commonly used datasets in FL research: FEMNIST [1], [32], Shakespeare [21], [33], and CIFAR-10 [34], the statistics of which are shown in Table III. FEMNIST and Shakespeare datasets simulate the non-IID and unbalanced data with realistic scenarios as described in [1]. CIFAR-10 dataset can be flexibly constructed to different numbers of clients with unbalanced data simulation and two different types of non-IID simulation: (1) dividing the dataset by Dirichlet process  $Dir(\alpha)$  [35]; (2) dividing the dataset by class, each client containing  $N$  out of 10 classes [22]. EasyFL provides flexibility for researchers to simulate various numbers of clients, applying non-IID data simulation and unbalanced data simulation, separately or in combination.

Besides these out-of-the-box datasets, researchers can also employ `register_dataset` API to easily integrate new datasets. For datasets that naturally simulate statistical heterogeneity, such as datasets provided in LEAF benchmark [1], EasyFL supports researchers to start training once the datasets are adapted into the system. For datasets that are not simulated yet, such as ImageNet dataset [36], EasyFL provides functionalities to partition the datasets for non-IID and unbalanced data simulations. These three datasets provide a good starting point, we will keep extending to provide more datasets for simulation.

**System Heterogeneity** Although system heterogeneity is associated with hardware resources, EasyFL simulates it in a lightweight and realistic manner. System heterogeneity is caused by varieties of hardware and network connectivity of clients, resulting in varied training times among clients and network transmission times between the server and clients. As a result, the server receives model updates from clients at different times — the slower ones are the stragglers. A significant system heterogeneity simulation should reveal these stragglers with varied time between server model distribution and client model uploads.

We simulate the time differences using training speed variances of mobile devices from AI-Benchmark [37]. We first calculate the training speed ratio of different mobile devices. Then we assign each client a type of mobile device based on speed ratio. In each round of training, clients not only execute the training but also wait for the time proportional to their speed ratios before uploading updates to the central server. Besides, EasyFL can simulate networking conditions (e.g., latency) for communication with an isolated environment provided by containerization (Section VII). As such, EasyFL simulates the system heterogeneity with stragglers.

### B. Training Flow Abstraction

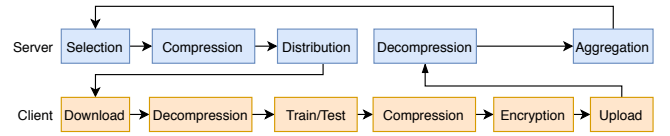


Fig. 3. Training flow abstraction divides the training process in the server and client modules into multiple stages.

EasyFL supports fine-grained plugin design by abstracting the FL training flow in the server and client modules to multiple stages as shown in Fig. 3. It provides researchers with the flexibility to customize any of these stages when developing new FL algorithms.

Each round of training or testing begins from the *selection stage*, which employs strategies to select a fraction of participated clients in the server. The *compression stage* implements data compression algorithms to reduce communication costs and the *decompression stage* decompresses the data. The *distribution stage* transmits data from the server to the clients. Clients download the data in the *download stage* and execute training or testing after decompression. The *encryption stage* encrypts the updates before clients upload them to the server



in the *upload stage*. At the end of each round, the server aggregates these updates in the *aggregation stage*.

We propose training flow abstraction based on analysis of FL open problems [19] and emerging solutions in top publications, as shown in Table VII. We analyzed 33 papers from recent top publications: 10 out of 33 ( $\sim 30\%$ ) publications innovate new algorithms by changing only one stage of the training flow; The majority ( $\sim 57\%$ ) change only two stages of the training flow. However, implementations of these studies mostly need to write the whole FL training process. In contrast, training flow abstraction allows researchers to focus on innovating algorithms, without re-implementing the whole training process. The stages in both the server and client modules serve as templates for the standard FL training process. Users can inherit them and replace one or more stages to implement new algorithms. For example, researchers who focus on improving communication efficiency can develop new compression algorithms to replace the *compression* related stages. We integrate a compression algorithm [38] as an example with around 80 lines of code, whereas the released implementation requires several hundred lines of code.

Moreover, EasyFL encourages users to publish their new federated algorithms as plugins to the community. Others can easily use these plugins to reproduce results and further develop on them with minimal coding. We believe that EasyFL has a great potential to grow and become an open-source community to facilitate the reproducibility of FL research.

### C. Comprehensive Tracking

To support comprehensive and easy analysis of FL training results, EasyFL provides a powerful tracking manager. The tracking manager is specially designed to collect and store three levels of FL metrics: training task metrics, round metrics of a task, and client metrics of a round, equipping researchers with in-depth analysis with comprehensive details. Task metrics include information about the whole training such as configurations and hyperparameters. Round metrics record the metrics in the server for each round of training and testing, such as accuracy, communication cost, and training time. Client metrics contain training and testing metrics of selected clients.

EasyFL supports two forms of tracking for diverse training methods: (1) *Local tracking* logs metrics to local storage, which is efficient for standalone and distributed training; (2) *Remote tracking* starts a tracking service to collect metrics via API calls, required by remote training.

The tracking manager provides command-line tools to query the metrics. It also exposes simple APIs for gRPC calls and HTTP requests, which are extensible to build a visualization dashboard and real-time performance monitoring.

## VI. DISTRIBUTED TRAINING OPTIMIZATION

EasyFL accelerates distributed training under heterogeneous simulations and resource constraints in the distribution manager. Although FL clients are deployed to edge devices such as mobile phones in real-world scenarios, researchers *simulate FL experiments mainly on GPUs* for faster iteration of algorithm

---

### Algorithm 1 Greedy Allocation with Adaptive Profiling

---

```

1: Input: number of GPUs  $M$ ; a set of  $N$  clients  $C_N$ ;
   whether client is profiled  $c.profiled$ ; default client training
   time  $c.time = t$ ; update momentum  $m$ ;
2: for each training round  $r=0$  to  $R-1$  do
3:    $C_K = \text{sort}(\text{random } K \text{ clients of } C_N) \text{ by } c.time \text{ in desc}$ 
4:    $G_M = \text{initialize } M \text{ empty lists}$ 
5:    $T_M = \text{initialize a list of } M \text{ 0s}$ 
6:   for each client  $c \in C_K$  do
7:     if not  $c.profiled$  then
8:        $c.time = t$ 
9:     end if
10:     $\text{index} = \text{argmin}(T_M)$ 
11:     $T_M[\text{index}] += c.time$ 
12:     $G_M[\text{index}].\text{append}(c)$ 
13:  end for
14:   $t = \text{ADAPTIVE\_PROFILING}(G_M, t)$ 
15: end for
16: function ADAPTIVE_PROFILING( $G_M, t$ )
17:   for each group  $G \in G_M$  in parallel in one GPU do
18:     for each client  $c \in G$  do
19:       if not  $c.profiled$  then
20:          $c.profiled = \text{True}$ 
21:          $c.time = \text{training time of client } c$ 
22:       end if
23:     end for
24:   end for
25:    $C_K = \text{aggregate clients } G_M \text{ from } M \text{ GPUs}$ 
26:    $t_{avg} = \text{average}(\text{sum}(c.time \text{ for each client } c \in C_K))$ 
27:    $t_{avg} = t_{avg} * m + t * (1 - m)$ 
28:   return  $t_{avg}$ 
29: end function

```

---

development [1], [4], [5], [23], [24], [39]. To further speed up algorithm development iterations in the experimental phase, we propose distributed training optimization.

EasyFL enables the distributed training optimization with only one line change in configurations. It allows training with the number of selected clients larger than available hardware resources by allocating multiple clients into one hardware to conduct training. The allocation is not trivial because the training time of clients could vary dramatically under system heterogeneity or unbalanced data. We formulate the problem with the goal that the overall training time is shortest.

**Problem Formulation** We assume that the training times of all clients are known and describe the problem as following: Given  $M$  GPUs  $\{G_1, G_2, \dots, G_M\}$  and a set of training time of  $N$  clients  $S_t = \{t_1, t_2, \dots, t_N\}$ , where  $N \geq M$ , find the optimal way of allocating clients to GPUs such that the maximum training time across all GPUs is minimized. Basically, we aim to partition  $S_t$  into  $M$  mutually disjoint subsets  $S_1, S_2, \dots, S_M$  such that the total training time

$$T = \min_{i \in [1, M]} \max_{x \in S_i} \sum x, \quad (1)$$

where  $\sum_{x \in S_i} x$  is the total training time in GPU  $G_i$ .

**Greedy Allocation with Adaptive Profiling** The problem is a variant of multiprocessor scheduling problem [40], [41], which is an NP-hard optimization problem. We adopt the greedy algorithm (Longest Processing Time algorithm) to solve the problem: sort the clients in descending order by training time and allocate the slowest client to the GPU with the shortest total time.

However, the training time for each client is unknown at the beginning of the training. We use an adaptive strategy to update the training times of clients instead of profiling all the clients at the start of training like [24], [25] because profiling would consume intolerable time when total clients are large. We summarize the algorithm of greedy allocation and adaptive profiling in Algorithm 1, which we termed as Greedy Allocation with Adaptive Profiling (GreedyAda). GreedyAda updates the training time of selected clients and marks them as profiled after they complete each round of training. It then updates the training time of not-profiled clients with the running average of selected clients’ training time.

## VII. SEAMLESS AND SCALABLE DEPLOYMENT

To achieve faster iterations between the experimental and production phase, EasyFL supports seamless and scalable deployment. We first introduce remote communication that is the foundation for remote training in production. Next, we introduce containerization and service discovery mechanisms for seamless and scalable deployment.

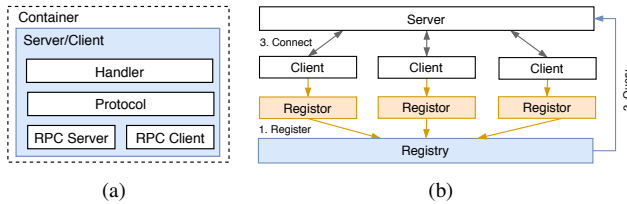


Fig. 4. Illustration of the architectures that support seamless and scalable deployment: (a) remote communication; (b) service discovery.

**Remote Communication** In production, remote communication supports message (model parameters or gradients) transmission between the server and clients, which is the most crucial component as the server and clients may locate in different locations. Fig. 4(a) shows the three-tier architecture of the server and the client that supports remote communication. At the distribution stage of the training flow, the server serializes the operation request (training/testing) with `Protocol` and distributes them to parallel-running clients with `RPC Client`. These requests are asynchronous because clients would take a long time to execute these operations. The `RPC server` receives messages from clients after they complete execution, passing to the `Protocol`. After the `Protocol` deserializes the messages, the `Handler` processes them to streamline the training pipeline.

Since training flow abstraction (Section V-B) decouples the training and communication, EasyFL integrates remote communication by providing alternative implementations for distribution stage and upload stage. When the system is started

TABLE III  
DATASETS AND MODELS PROVIDED BY EASYFL TO SIMULATE STATISTICAL HETEROGENEITY.

Datasets	# of samples	# of clients	Models
FEMNIST	805,263	3,550	CNN (2 Conv + 2 FC)
Shakespeare	4,226,158	1,129	RNN (2 LSTM + 1 FC)
CIFAR-10	60,000	Flexible	ResNet18

with `start_server` or `start_client` API, EasyFL switches on remote communication.

**Containerization** Based on the remote communication, EasyFL containerizes the server, the client, and the tracking service for easy and reliable deployment to the cloud infrastructure and the edge devices, focusing on the edge devices that support containers. EasyFL containerizes using Docker [42], which is the standard industrial containerization tool, to easily adapt to complex software dependencies in diverse computing environments of edge devices. On the one hand, containerization enables the simulation of networking conditions for system heterogeneity with simple configurations when starting the containers. On the other hand, containerization of FL further facilitates seamless deployment and builds the foundation to deploy FL services to edge computing frameworks such as AWS IoT Greengrass and KubeEdge.

**Service Discovery** EasyFL provides a service discovery mechanism for the server to discover the clients when scaling up. Fig. 4(b) shows the architecture of service discovery, containing the `registor` to dynamically register the clients and the `registry` to store the client addresses for the server to query. The `registor` gets the addresses of clients and registers them to the `registry`. Since the clients are unaware of the container environment they are running, they must rely on a third-party service (the `registor`) to fetch their container addresses to complete registration. The `registry` stores the registered client addresses for the server to query. EasyFL supports two service discovery methods targeting different deployment scenarios: (1) deployment using Kubernetes [43], which is an industrial-level container orchestration engine; (2) deployment using only Docker [42] containers.

## VIII. EVALUATION

In this section, we start by presenting the implementation details and experimental setup. We followed by evaluating EasyFL on heterogeneity simulation, distributed training optimization, and remote training. Then we compare the lines of code to EasyFL implementations with original implementations. We end by comparing the training overhead with other FL platforms and presenting a case study of EasyFL.

### A. Implementation

We implement EasyFL as a standalone Python library of  $\approx 6000$  lines of code (LOC). EasyFL uses PyTorch [44] as the backend for deep learning model training and testing.

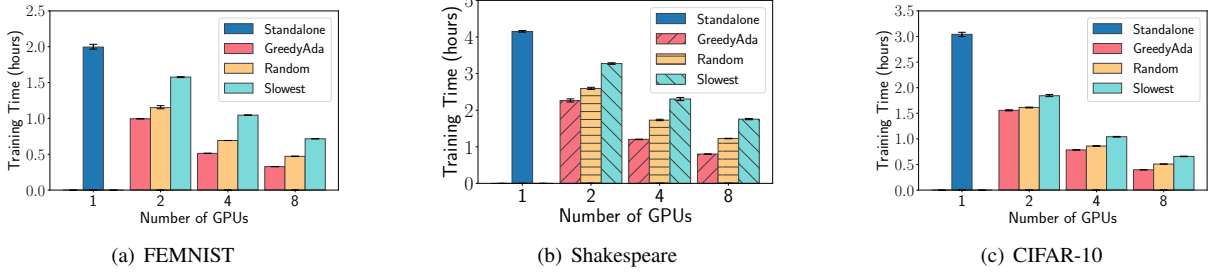


Fig. 5. Training time comparison of standalone and distributed training using three datasets. Greedy Allocation with Adaptive Profiling (GreedyAda) effectively accelerates training on all datasets. We run the experiments with 20 selected clients per round under simulated heterogeneous scenarios.

TABLE IV  
ACCURACY COMPARISON OF IID AND NON-IID SIMULATIONS USING EASYFL. DIFFERENT NON-IID DATA PARTITION METHODS LEAD TO DIFFERENT DEGREES OF PERFORMANCE DEGRADATION.

Datasets	Non-IID accuracy	IID accuracy	Accuracy gap
FEMNIST	78.12%	79.85%	1.73%
Shakespeare	46.15%	50.33%	4.18%
CIFAR-10	93.63% (dir)	94.91%	1.28%
CIFAR-10	89.06% (class(3))	94.91%	5.85%
CIFAR-10	73.66% (class(2))	94.91%	21.25%

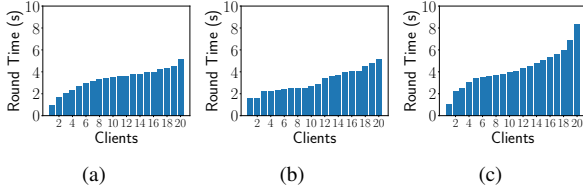


Fig. 6. Impact of heterogeneity simulation on training time per round of sampled 20 clients on CIFAR-10 dataset: (a) unbalanced data, (b) system heterogeneity, and (c) combined simulation of (a) and (b). All simulations cause training time variances, especially the combination method.

Remote communication in EasyFL is based on gRPC<sup>1</sup>, a high-performance RPC framework. We also provide several other RPC methods to start and stop training. Protocol Buffers<sup>2</sup>, a lightweight method for serializing structured data, are used to serialize remote messages.

Targeting two different types of deployment, EasyFL provides different stacks for service discovery. For deployment using Kubernetes [43], EasyFL utilizes Pod, the smallest deployable unit in Kubernetes where containers run on, as the `registor`. The Service in Kubernetes, which connects client Pods and an internal DNS, serves as the `registry`. For deployment using only Docker [42] containers, EasyFL uses `etcd`<sup>3</sup> as `registry` because it is a reliable and consistent key-value store for distributed systems. It provides docker image for docker-gen<sup>4</sup>, which serves as `registor` to get the container metadata including IP addresses.

<sup>1</sup><https://grpc.io/>

<sup>2</sup><https://developers.google.com/protocol-buffers>

<sup>3</sup><https://etcd.io/>

<sup>4</sup><https://github.com/jwilder/docker-gen>

## B. Experimental Setup

**Datasets and Models** We use datasets and models provided by EasyFL shown in Table III. We evaluate the performance under IID and non-IID partition of these datasets.

**Algorithm and Hyperparameters** EasyFL provides FedAvg [21] as the standard algorithm. By default, we set local epoch to be 10 ( $E = 10$ ) and batch size to be 64 ( $B = 64$ ) for all the experiments. We use SGD as the optimizer and tune the learning rate for each dataset. For different experiments, we select different numbers of clients to participate in the training. More experiment settings are provided in Appendix B-A.

**Evaluation Metrics** We evaluate EasyFL both qualitatively and quantitatively. On the one hand, we provide qualitative results of lines of code (LOC) to implement FL applications. On the other hand, we use the evaluation metrics provided by the tracking manager for quantitative results: model accuracy, total training time, processing time each round (round time), and communication cost. To reduce the impact of hardware instability during training or testing, we calculate the round time  $T_{round}$  by averaging end-to-end processing time  $T_{total}$  of  $R$  rounds,  $T_{round} = \frac{T_{total}}{R}$ .

**Experiment Environment** All simulation experiments are run on one NVIDIA<sup>®</sup> 2080Ti GPU or 64 NVIDIA<sup>®</sup> V100 GPUs (8 GPUs per node) with CUDA 10.1 and CuDNN 7.6. We deploy and evaluate EasyFL in production on a Kubernetes cluster with three nodes, where each node has 28 Intel(R) Core(TM) i9-9940X CPUs.

## C. Heterogeneity Simulation

To show the effectiveness of heterogeneity simulation, we present benchmark results on the impact of simulated statistical heterogeneity and system heterogeneity.

**Impact of Statistical Heterogeneity on Performance** We compare the accuracy of models trained on IID and non-IID settings. For FEMNIST and Shakespeare, we simulate non-IID with realistic partition. For CIFAR-10, we simulate three levels of non-IID with increasing heterogeneity: Dirichlet process  $Dir(0.5)$  (10 classes per client), three classes per client, and two classes per client. These experiments are run with 10 selected clients per round.

Table IV shows that the simulated statistical heterogeneity leads to performance degradation. Models trained on simulated non-IID data all perform worse than models trained on the IID data. In particular, the increasing degree of statistical



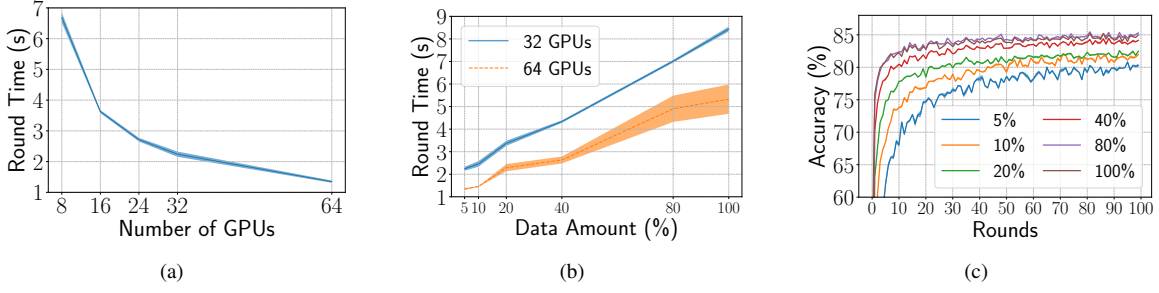


Fig. 7. Performance comparison of EasyFL on (a) round time of different numbers of GPUs, (b) round time of varied data amount, (c) accuracy of varied data amount. These results demonstrate the scalability of EasyFL: (a) the round time is effectively reduced with an increasing number of GPUs; (b) the round time increased (less than 4x) is much lower than the increased data amount (20x from 5% to 100%); (c) the accuracy increased from  $\sim 80\%$  to  $\sim 85\%$  with data amount increased from 5% to 100%. We run the experiments with 100 selected clients per round for 100 rounds under the IID setting.

heterogeneity simulated from *Dir* to two classes per client causes even larger degradation, leading to the largest accuracy gap of 21.25%. Researchers can use these results as a benchmark when they start using EasyFL and have the flexibility to customize the degree of non-IID.

**Impact of Heterogeneity on Training Time** We simulate unbalanced data by *Dir*(0.5) and system heterogeneity using the EasyFL simulation manager. Fig. 6 shows that unbalanced data, system heterogeneity, and their combination cause a huge discrepancy in training time of each round among clients in the CIFAR-10 dataset. The fastest client is four times faster than the slowest client because of unbalanced data as shown in Fig. 6(a). This gap is larger under system heterogeneity in Fig. 6(b) and their combination in Fig. 6(c). FL training normally requires hundreds of rounds, further amplifying the impact of the stragglers. We provide the results of training time per round of Shakespeare and CIFAR-10 datasets in Appendix B-B.

#### D. Distributed Training Optimization

In this section, we demonstrate the effectiveness of our proposed GreedyAda and the scalability of distributed training in EasyFL.

**Performance of GreedyAda** EasyFL accelerates training with GreedyAda (Greedy Allocation with Adaptive Profiling) under resource constraints of  $M$  available GPUs and heterogeneous scenarios. The heterogeneous scenarios are simulated with the combination of unbalanced data and system heterogeneity described in Section VIII-C. We run the experiments with 20 selected clients per round using standalone training and distributed training with different client allocation strategies: (1) GreedyAda; (2) random allocation that randomly allocates around  $\frac{20}{M}$  clients to a GPU; (3) slowest allocation that allocates around  $\frac{20}{M}$  slowest clients to a GPU. Fig. 5 demonstrates that GreedyAda achieves the fastest training speed — up to 1.5x faster than random allocation and up to 2.2x faster than slowest allocation — in all datasets with different number of GPUs.

**Scalability** We further evaluate the scalability of distributed training with different numbers of GPUs  $\{8, 16, 24, 32, 64\}$  and varied amount of data  $\{5\%, 10\%, 20\%, 40\%, 80\%, 100\%\}$ . Data amount means the percentage of samples used training in clients. We run these experiments with 100 selected clients

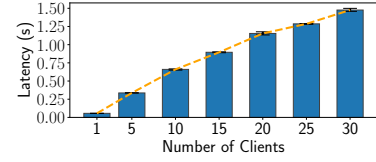


Fig. 8. The distribution latency from the server to clients increases almost linearly when scaling up the number of clients in the production phase. The latency is low compared to training time.

each round under the IID setting for 100 rounds on FEMNIST and measure the processing time per round (round time).

Fig. 7(a) demonstrates that the round time is effectively reduced with increasing numbers of GPUs. While compared with 1.84x (optimal 2x) speedup from 8 GPUs to 16 GPUs, the speedup from 8 GPUs to 64 GPUs is only 4.96x (optimal 8x). The underutilization of hardware resources is mainly because the data amount is small (5%) for these experiments, causing the communication overhead among GPUs to outweigh the training time when the number of GPUs is large. We further investigate it by increasing the data amount.

Fig. 7(b) illustrates that the round time increases as data amount increases, either using 32 or 64 GPUs. However, the increase in round time is much smaller than the increase in data amount. In particular, the data amount increases by 20x from 5% to 100%, but the round time only increases less than 4x. These results suggest that when the data amount is small, the smaller number of GPUs can handle training effectively; when the data amount is large, EasyFL can scale up to the large number of GPUs for training. Besides, we also present the evaluation accuracy of different data amounts in Fig. 7(c). The accuracy increased from  $\sim 80\%$  to  $\sim 85\%$  with data amount increased from 5% to 100%.

#### E. Remote Training

We evaluate remote training on time to deployment and performance of training in containers.

**Deployment Time** EasyFL enables sub-linear time to deployment by leveraging the power of containerization and container orchestration engine, Kubernetes. Without containerization, the time to deployment grows linearly as we increase the number of clients, where the deployment of one client

TABLE V  
COMPARISON OF LINES OF CODE OF THE ORIGINAL AND EASYFL IMPLEMENTATIONS ON THREE DIFFERENT FL APPLICATIONS.

Name	Application	Original	EasyFL
FedProx [23]	Optimization framework	~380	~40
STC [38]	Compression framework	~560	~80
FedReID [13]	Video surveillance	~640	~140

would need hours because of diverse and complex computing environments. In contrast, EasyFL only needs a one-time setup of the Docker environment and a Kubernetes cluster. It builds docker images of FL components in seconds and deploys clients to each node in Kubernetes clusters in minutes, dramatically reducing the deployment time and effort.

**Accuracy and Distribution Latency** EasyFL supports remote training to achieve the same accuracy as experiments in Table IV under the same settings. Besides, Fig. 8 presents the distribution latency of the server when scaling up the number of clients using FEMNIST dataset. Although the distribution latency increases almost linearly using multi-threading, the latency is relatively low compared to the training time needed. To further optimize this distribution latency, we can consider replicating servers to load balance the requests.

#### F. Applications

EasyFL enables easy implementation of FL to a wide range of applications by allowing seamless migration from existing training codes. We demonstrate its capability by using it to implement three applications: (1) FedProx [23] is an optimization framework to address statistical and system heterogeneity; (2) STC [38] is a compression framework to reduce communication cost; (3) FedReID [13] implements FL to person re-identification, an important computer vision task for video surveillance. Table V compares lines of code (LOC) of EasyFL implementation with the original implementations. EasyFL greatly reduces the efforts of researchers in writing codes, 4.5x to 9.5x less coding. Fewer codes result in fewer bugs and faster iterations of development. Besides, the codes are similar to non-FL implementation that researchers are familiar with. We provide more details of LOC counting and implementation of STC [38] using EasyFL in Appendix A.

#### G. Training Overhead

Despite that EasyFL reduces the LOC significantly with abstractions, we further investigate whether these abstractions would lead to extra training overhead by comparing with other FL libraries: LEAF [1] and TFF [4]. We conduct the experiments with 10 selected clients per round under IID simulation for  $R = 150$  rounds on FEMNIST and  $R = 100$  rounds on Shakespeare and CIFAR-10.

Table VI compares the round time of EasyFL, LEAF, and TFF<sup>5</sup> on three datasets. These results indicate that EasyFL,

<sup>5</sup>To train LEAF and TFF on GPUs with CUDA10.1 and CuDNN 7.6, we use TFF v0.17.0 (latest v0.19.0) and upgrade implementation of LEAF from TensorFlow 1.3 to TensorFlow 2.2.

TABLE VI  
COMPARISON OF TRAINING OVERHEAD (ROUND TIME) OF DIFFERENT FL LIBRARIES ON DIFFERENT HARDWARE. EASYFL IS MORE EFFICIENT THAN LEAF AND TFF IN THE MAJORITY OF SETTINGS.

Platforms	Hardware	FEMNIST	Shakespeare	CIFAR-10
LEAF [1]	2080Ti	7.9s	181.6s	— <sup>a</sup>
TFF [4]	2080Ti	5.5s	960.5s	<b>30.4s</b>
EasyFL (Ours)	2080Ti	<b>3.9s</b>	<b>31.5s</b>	39.8s
LEAF [1]	V100	56.3s	959.3s	— <sup>a</sup>
TFF [4]	V100	8.7s	1574.4s	83.2s
EasyFL (Ours)	V100	<b>2.5s</b>	<b>33.2s</b>	<b>36.1s</b>

<sup>a</sup> LEAF does not support CIFAR-10 dataset.

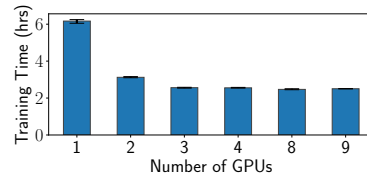


Fig. 9. EasyFL achieves near-optimal training speed using 3 GPUs instead of 9 GPUs for training FedReID [13] with 9 clients, reducing the demand for hardware resources.

instead of incurring extra training overhead, achieves a smaller round time than other libraries in the majority of settings. Surprisingly, we found that LEAF and TFF are more efficient on NVIDIA 2080Ti, compared with the round times on NVIDIA V100 that supposed to have stronger computation capability. Also, the round times of TFF on Shakespeare are unexpectedly large on both types of GPUs. We suspect that these results are caused by implementations of models (e.g., LSTM) in TFF and different internal implementations for CuDNN operations in different versions of TensorFlow [17] and PyTorch [44]<sup>6</sup>.

#### H. Case Study

We conduct a case study of EasyFL on developing new federated computer vision applications with FedReID [13]. FedReID implements FL to person re-identification (ReID), which is an important retrieval task in computer vision. Since FedReID is a new application that uses nine heterogeneous datasets to simulate nine clients, we adapt these datasets into EasyFL via `register_dataset` API. As the training and testing of ReID are different from image classification, we customize the `train` and `test` in clients and integrate it with `register_client` API. The codes for training and testing are almost the same as the ones used for normal person ReID training. With these two registrations, we then initialize EasyFL with `init` API with configurations like training rounds and learning rates. After these steps, we can start training with `run` API.

We further leverage distributed training optimization in EasyFL to accelerate FedReID training and achieve near-optimal training speed with 6 fewer GPUs, as shown in Fig. 9. We train FedReID with 9 clients containing unbalanced data. The client with the largest dataset is the bottleneck in training. Instead of training with 9 GPUs by allocating each client to

<sup>6</sup>EasyFL uses PyTorch, whereas LEAF and TFF use TensorFlow.

one of them, EasyFL saves hardware resources by achieving similar training speeds with only 3 GPUs.

Moreover, we build a prototype of FedReID using EasyFL with containerization and seamless deployment to Kubernetes. It imitates the real-world scenario that a client is deployed to an edge device, training with data collected from cameras.

## IX. CONCLUSION

In this paper, we propose a low-code FL platform, *EasyFL*, to empower users with different levels of expertise to conduct FL experiments efficiently and deploy FL applications seamlessly with minimal coding. We achieve it with a two-tier system architecture: the interface layer contains simple APIs that shield complex system implementations for users; the system layer embraces modular design and abstracts the training flow to provide flexibility and reusability. Eight modules in the system layer provide out-of-the-box functionalities to accelerate the iteration and increase the productivity of users. EasyFL only needs 3 lines of code (LOC) to implement a vanilla FL application, which is at least 10x less than other platforms. We also implement several FL applications with 4.5x to 9.5x less LOC than the original implementations. Our evaluation demonstrates that EasyFL increases the training speed of distributed training by 1.5x. It also reduces the demand for resources and decreases the time to deployment.

In the future, we will support out-of-the-box encryption methods in EasyFL and provide mechanisms to capture the production environment for more representative simulations. Besides, we consider a *no-code* FL platform to further liberate FL for wider adoption.

## APPENDIX A APPLICATIONS

In this section, we provide more details on lines of code comparison and present an implementation example of EasyFL.

We supplement the source codes for implementations of the FL vanilla application using various frameworks<sup>7 8 9 10 11</sup>.

We compare EasyFL implementations with original implementations on lines of code for specific applications in Table V, not counting the lines of the import statements. We refer to the original implementations in Github<sup>12 13 14</sup>. As for EasyFL implementation, due to space limit, we only present the implementation of STC [38] in Listing 2. (Function `stc` is from the original implementation.) We are in the internal process of getting approval to open-source EasyFL and all the implementations.

<sup>7</sup>LEAF: <https://github.com/TalwalkarLab/leaf>

<sup>8</sup>TFF: <https://www.tensorflow.org/federated>

<sup>9</sup>PaddleFL: [https://github.com/PaddlePaddle/PaddleFL/tree/master/python/paddle\\_fl/paddle\\_fl/examples/femnist\\_demo](https://github.com/PaddlePaddle/PaddleFL/tree/master/python/paddle_fl/paddle_fl/examples/femnist_demo)

<sup>10</sup>FATE: [https://github.com/FederatedAI/FATE/blob/master/examples/pipeline/hetero\\_nn/pipeline-hetero-nn-train-binary.py](https://github.com/FederatedAI/FATE/blob/master/examples/pipeline/hetero_nn/pipeline-hetero-nn-train-binary.py)

<sup>11</sup>PySyft: <https://github.com/OpenMined/PySyft/blob/dev/examples/experimental/madhava/MNIST.ipynb>

<sup>12</sup>STC: <https://github.com/feliasat/federated-learning>

<sup>13</sup>FedProx: <https://github.com/litian96/FedProx>

<sup>14</sup>FedReID: <https://github.com/cap-ntu/FedReID>

```
class STCClient(BaseClient):
    def compression(self):
        weights = self.model.state_dict()
        old_weights = self.downloaded_model.state_dict()
        dw = {} # parameter updates
        comp_dw = {} # compressed parameter updates
        for i in weights:
            dw[i] = weights[i] - old_weights[i]
        for i in dw:
            comp_dw[i] = stc(dw[i])
        # construct data for upload stage
        self.upload_holder = server_pb.UploadContent(
            data=codec.marshal(copy.deepcopy(comp_dw)),
            data_type=common_pb.DATA_TYPE_PARAMS,
            data_size=self.train_data.size(self.cid)
        )

class STCServer(BaseServer):
    def decompression(self, model):
        '''param model: compressed model from client'''
        new_model = copy.deepcopy(self.model)
        weights = new_model.state_dict()
        for i in weights:
            weights[i].data += model[i].data.clone()
        new_model.load_state_dict(weights)
        return new_model

config = {"data": {"dataset": "femnist"}}
easyfl.register_server(STCServer)
easyfl.register_client(STCClient)
easyfl.init(config)
easyfl.run()
```

Listing 2. Implementation of STC compression algorithm.

## APPENDIX B EXPERIMENTS

This section provides detailed experiment settings and two complementary experiment results.

### A. Experiment Settings

By default, we use batch size  $B = 64$  and local epoch  $E = 10$ . We use SGD with momentum 0.9 as the optimizer with learning rates  $\eta = 0.01$  for FEMNIST dataset and CIFAR-10 dataset and  $\eta = 0.8$  for Shakespeare dataset.

For experiments on *Impact of Statistical Heterogeneity* in Section VIII-C, we use the number of selected clients  $C = 10$  and total training rounds  $R = 150$  for FEMNIST dataset [1], [32], and  $R = 100$  for both Shakespeare [1], [33] and CIFAR-10 [34] datasets.

For experiments on *Performance of GreedyAda* in Section VIII-D, we run experiments with 20 selected clients each round and train FEMNIST with  $R = 500$  rounds, and Shakespeare and CIFAR-10 with  $R = 100$  rounds.

For experiments on *FedReID*, we use batch size  $B = 32$ ,  $E = 1$ , and learning rate setting same as the original paper.

### B. Experiment Results

**Impact of Heterogeneity on Training Time** Fig. 10 and 11 show the impact of heterogeneity simulation on training time of each round, using FEMNIST and Shakespeare datasets. They simulate the training time discrepancy and stragglers with unbalanced data and system heterogeneity. The combination of unbalanced data and system heterogeneity simulation results has the largest training time variances. Depending on the datasets, the value and scale of round time are different.

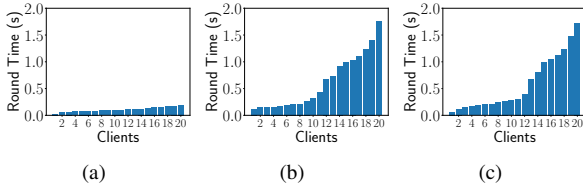


Fig. 10. Impact of heterogeneity simulation on training time of sampled 20 clients in one round of training using FEMNIST dataset: (a) unbalanced data simulated by  $Dir(0.5)$ , (b) system heterogeneity, and (c) combination effect of (a) and (b). All simulations cause training time variances.

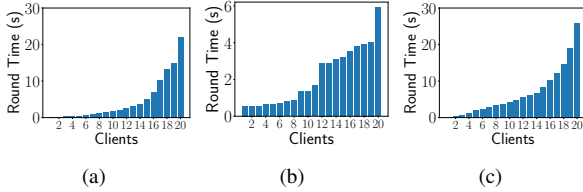


Fig. 11. Impact of heterogeneity simulation on training time of sampled 20 clients in one round of training using Shakespeare dataset: (a) unbalanced data simulated by  $Dir(0.5)$ , (b) system heterogeneity, and (c) combination effect of (a) and (b). All simulations cause training time variances.

**Impact of Statistical Heterogeneity** Fig. 12 shows the performance comparison of IID and non-IID data simulated using EasyFL with three datasets: FEMNIST, Shakespeare, and CIFAR-10. These figures further illustrate the accuracy gap between IID and non-IID data.

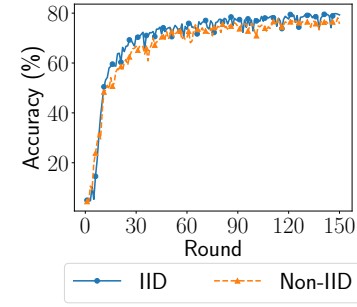
## APPENDIX C

### RECENT PUBLICATIONS OF FEDERATED LEARNING

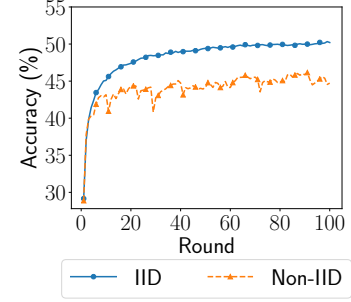
We surveyed 33 papers from recent publications of FL from both the machine learning and system community. Table VII shows that 10 out of 33 ( $\sim 30\%$ ) publications propose new algorithms with changes in only one stage of the training flow, and the majority ( $\sim 57\%$ ) change only two stages. Training flow abstraction (Section V-B) allows researchers to focus on the problems, without re-implementing the whole FL process.

## REFERENCES

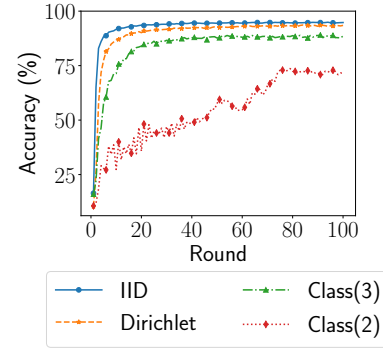
- [1] S. Caldas, S. M. K. Duddu, P. Wu, T. Li, J. Konečný, H. B. McMahan, V. Smith, and A. Talwalkar, “Leaf: A benchmark for federated settings,” *arXiv preprint arXiv:1812.01097*, 2018.
- [2] T. Ryffel, A. Trask, M. Dahl, B. Wagner, J. Mancuso, D. Rueckert, and J. Passerat-Palmbach, “A generic framework for privacy preserving deep learning,” *arXiv preprint arXiv:1811.04017*, 2018.
- [3] Y. Ma, D. Yu, T. Wu, and H. Wang, “Paddlepaddle: An open-source deep learning platform from industrial practice,” *Frontiers of Data and Computing*, vol. 1, no. 1, pp. 105–115, 2019.
- [4] Tensorflow.org, “Tensorflow federated,” 2019. [Online]. Available: <https://github.com/tensorflow/federated>
- [5] WeBank, “Federated ai technology enabler (fate),” 2019. [Online]. Available: <https://github.com/FederatedAI/FATE>
- [6] M. J. Sheller, G. A. Reina, B. Edwards, J. Martin, and S. Bakas, “Multi-institutional deep learning modeling without sharing patient data: A feasibility study on brain tumor segmentation,” in *International MICCAI Brainlesion Workshop*. Springer, 2018, pp. 92–104.
- [7] W. Li, F. Milletari, D. Xu, N. Rieke, J. Hancox, W. Zhu, M. Baust, Y. Cheng, S. Ourselin, M. J. Cardoso *et al.*, “Privacy-preserving federated brain tumour segmentation,” in *International Workshop on Machine Learning in Medical Imaging*. Springer, 2019, pp. 133–141.
- [8] Y. Chen, X. Qin, J. Wang, C. Yu, and W. Gao, “Fedhealth: A federated transfer learning framework for wearable healthcare,” *IEEE Intelligent Systems*, 2020.
- [9] A. Hard, K. Rao, R. Mathews, S. Ramaswamy, F. Beaufays, S. Augenstein, H. Eichner, C. Kiddon, and D. Ramage, “Federated learning for mobile keyboard prediction,” *arXiv preprint arXiv:1811.03604*, 2018.
- [10] D. Leroy, A. Coucke, T. Lavril, T. Gisselbrecht, and J. Dureau, “Federated learning for keyword spotting,” in *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2019, pp. 6341–6345.
- [11] C. Niu, F. Wu, S. Tang, L. Hua, R. Jia, C. Lv, Z. Wu, and G. Chen, “Billion-scale federated learning on mobile clients: A submodel design with tunable privacy,” in *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, 2020, pp. 1–14.
- [12] K. Muhammad, Q. Wang, D. O’Reilly-Morgan, E. Tragos, B. Smyth, N. Hurley, J. Geraci, and A. Lawlor, “Fedfast: Going beyond average for faster training of federated recommender systems,” in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 1234–1242.
- [13] W. Zhuang, Y. Wen, X. Zhang, X. Gan, D. Yin, D. Zhou, S. Zhang, and



(a) FEMNIST



(b) Shakespeare



(c) CIFAR-10

Fig. 12. Performance comparison of IID and Non-IID simulated using EasyFL with selected clients per round  $C = 10$ . Different non-IID data partition methods lead to different degrees of performance degradation. FEMNIST and Shakespeare use the realistic non-IID partition. CIFAR-10 partitions non-IID by Dirichlet process (dir) and class with  $n$  classes per client (class( $n$ )).



- S. Yi, "Performance optimization of federated person re-identification via benchmark analysis," in *Proceedings of the 28th ACM International Conference on Multimedia*, 2020, pp. 955–963.
- [14] T. Yang, G. Andrew, H. Eichner, H. Sun, W. Li, N. Kong, D. Ramage, and F. Beaufays, "Applied federated learning: Improving google keyboard query suggestions," *arXiv preprint arXiv:1812.02903*, 2018.
- [15] N. Rieke, J. Hancox, W. Li, F. Milletari, H. R. Roth, S. Albarqouni, S. Bakas, M. N. Galtier, B. A. Landman, K. Maier-Hein *et al.*, "The future of digital health with federated learning," *NPJ digital medicine*, vol. 3, no. 1, pp. 1–7, 2020.
- [16] K. Panetta. (2020) Gartner top strategic technology trends for 2021. [Online]. Available: <https://www.gartner.com/smarterwithgartner/gartner-top-strategic-technology-trends-for-2021/>
- [17] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.
- [18] F. Chollet *et al.*, "Keras," <https://keras.io>, 2015.
- [19] P. Kairouz, H. B. McMahan, B. Avent, A. Bellet, M. Bennis, A. N. Bhagoji, K. Bonawitz, Z. Charles, G. Cormode, R. Cummings *et al.*, "Advances and open problems in federated learning," *arXiv preprint arXiv:1912.04977*, 2019.
- [20] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith, "Federated learning: Challenges, methods, and future directions," *IEEE Signal Processing Magazine*, vol. 37, pp. 50–60, 2020.
- [21] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Artificial Intelligence and Statistics*. PMLR, 2017, pp. 1273–1282.
- [22] Y. Zhao, M. Li, L. Lai, N. Suda, D. Civin, and V. Chandra, "Federated learning with non-iid data," *arXiv preprint arXiv:1806.00582*, 2018.
- [23] T. Li, A. K. Sahu, M. Zaheer, M. Sanjabi, A. Talwalkar, and V. Smith, "Federated optimization in heterogeneous networks," in *Proceedings of Machine Learning and Systems 2020*, 2020, pp. 429–450.
- [24] Z. Chai, A. Ali, S. Zawad, S. Truex, A. Anwar, N. Baracaldo, Y. Zhou, H. Ludwig, F. Yan, and Y. Cheng, "Tifl: A tier-based federated learning system," in *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, 2020, pp. 125–136.
- [25] C. Yang, Q. Wang, M. Xu, S. Wang, K. Bian, and X. Liu, "Heterogeneity-aware federated learning," *arXiv preprint arXiv:2006.06983*, 2020.
- [26] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang *et al.*, "Large scale distributed deep networks," in *Advances in neural information processing systems*, 2012, pp. 1223–1231.
- [27] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, and M. Young, "Machine learning: The high interest credit card of technical debt," in *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)*, 2014.
- [28] A. Burkov, *Machine learning engineering*. True Positive Incorporated, 2020.
- [29] H. Zhang, Y. Li, Y. Huang, Y. Wen, J. Yin, and K. Guan, "Mlmodelci: An automatic cloud platform for efficient mlaas," in *Proceedings of the 28th ACM International Conference on Multimedia*, 2020, pp. 4453–4456.
- [30] B. Karlaš, M. Interlandi, C. Renggli, W. Wu, C. Zhang, D. Mukunthu Iyappan Babu, J. Edwards, C. Lauren, A. Xu, and M. Weimer, "Building continuous integration services for machine learning," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 2407–2415.
- [31] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konečný, S. Mazzocchi, B. McMahan, T. Van Overveldt, D. Petrou, D. Ramage, and J. Roselander, "Towards federated learning at scale: System design," in *Proceedings of Machine Learning and Systems*, A. Talwalkar, V. Smith, and M. Zaharia, Eds., 2019, vol. 1, pp. 374–388. [Online]. Available: <https://proceedings.mlsys.org/paper/2019/file/bd686fd640be98efaae0091fa301e613-Paper.pdf>
- [32] G. Cohen, S. Afshar, J. Tapson, and A. Van Schaik, "Emnist: Extending mnist to handwritten letters," in *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2017, pp. 2921–2926.
- [33] W. Shakespeare, *The complete works of William Shakespeare*. Wordsworth Editions, 2007.
- [34] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.
- [35] H. Wang, M. Yurochkin, Y. Sun, D. Papailiopoulos, and Y. Khazaeni, "Federated learning with matched averaging," in *International Conference on Learning Representations*, 2020. [Online]. Available: <https://openreview.net/forum?id=BkluqlSFDS>
- [36] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR09*, 2009.
- [37] A. Ignatov, R. Timofte, W. Chou, K. Wang, M. Wu, T. Hartley, and L. Van Gool, "Ai benchmark: Running deep neural networks on android smartphones," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 0–0.
- [38] F. Sattler, S. Wiedemann, K.-R. Müller, and W. Samek, "Robust and communication-efficient federated learning from non-iid data," *IEEE transactions on neural networks and learning systems*, 2019.
- [39] F. Lai, X. Zhu, H. V. Madhyastha, and M. Chowdhury, "Oort: Informed participant selection for scalable federated learning," *arXiv preprint arXiv:2010.06081*, 2020.
- [40] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, 1969.
- [41] D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham, "Worst-case performance bounds for simple one-dimensional packing algorithms," *SIAM Journal on computing*, vol. 3, no. 4, pp. 299–325, 1974.
- [42] Docker, "Docker," 2013. [Online]. Available: <https://www.docker.com/>
- [43] Kubernetes.io, "Kubernetes," 2014. [Online]. Available: <https://kubernetes.io/>
- [44] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.
- [45] S. AbdulRahman, H. Tout, A. Mourad, and C. Talhi, "Fedmccs: multi-criteria client selection model for optimal iot federated learning," *IEEE Internet of Things Journal*, vol. 8, no. 6, pp. 4723–4735, 2020.
- [46] H. Wang, Z. Kaplan, D. Niu, and B. Li, "Optimizing federated learning on non-iid data with reinforcement learning," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 2020, pp. 1698–1707.
- [47] F. Sattler, S. Wiedemann, K.-R. Müller, and W. Samek, "Robust and communication-efficient federated learning from non-iid data," *IEEE transactions on neural networks and learning systems*, vol. 31, no. 9, pp. 3400–3413, 2019.
- [48] T. Lin, L. Kong, S. U. Stich, and M. Jaggi, "Ensemble distillation for robust model fusion in federated learning," *arXiv preprint arXiv:2006.07242*, 2020.
- [49] W. Luping, W. Wei, and L. Bo, "Cmfl: Mitigating communication overhead for federated learning," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 954–964.
- [50] D. Rothchild, A. Panda, E. Ullah, N. Ivkin, I. Stoica, V. Braverman, J. Gonzalez, and R. Arora, "Fetchsgd: Communication-efficient federated learning with sketching," in *International Conference on Machine Learning*. PMLR, 2020, pp. 8253–8265.
- [51] S. P. Karimireddy, S. Kale, M. Mohri, S. Reddi, S. Stich, and A. T. Suresh, "Scaffold: Stochastic controlled averaging for federated learning," in *International Conference on Machine Learning*. PMLR, 2020, pp. 5132–5143.
- [52] X. Wu, X. Yao, and C.-L. Wang, "Fedscr: Structure-based communication reduction for federated learning," *IEEE Transactions on Parallel and Distributed Systems*, 2020.
- [53] Z. Tao and Q. Li, "esgd: Communication efficient distributed deep learning on the edge," in *{USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 18)*, 2018.
- [54] S. Reddi, Z. Charles, M. Zaheer, Z. Garrett, K. Rush, J. Konečný, S. Kumar, and H. B. McMahan, "Adaptive federated optimization," *arXiv preprint arXiv:2003.00295*, 2020.
- [55] W. Gao, S. Guo, T. Zhang, H. Qiu, Y. Wen, and Y. Liu, "Privacy-preserving collaborative learning with automatic transformation search," *arXiv preprint arXiv:2011.12505*, 2020.
- [56] Y. Deng, M. M. Kamani, and M. Mahdavi, "Distributionally robust federated averaging," *arXiv preprint arXiv:2102.12660*, 2021.
- [57] C. He, M. Annavaram, and S. Avestimehr, "Group knowledge transfer: Federated learning of large cnns at the edge," *Advances in Neural Information Processing Systems*, vol. 33, 2020.
- [58] C. T. Dinh, N. H. Tran, and T. D. Nguyen, "Personalized federated learning with moreau envelopes," *arXiv preprint arXiv:2006.08848*, 2020.
- [59] T. Li, M. Sanjabi, A. Beirami, and V. Smith, "Fair resource allocation in federated learning," *arXiv preprint arXiv:1905.10497*, 2019.
- [60] X. Y. Felix, A. S. Rawat, A. K. Menon, and S. Kumar, "Federated learning with only positive labels," *arXiv preprint arXiv:2004.10342*, 2020.



- [61] L. Wang, S. Xu, X. Wang, and Q. Zhu, "Addressing class imbalance in federated learning," *arXiv preprint arXiv:2008.06217*, 2021.
- [62] R. Wu, A. Scaglione, H.-T. Wai, N. Karakoc, K. Hreinsson, and W.-K. Ma, "Federated block coordinate descent scheme for learning global and personalized models," *arXiv preprint arXiv:2012.13900*, 2020.
- [63] H. Sun, S. Li, F. R. Yu, Q. Qi, J. Wang, and J. Liao, "Toward communication-efficient federated learning in the internet of things with edge computing," *IEEE Internet of Things Journal*, vol. 7, no. 11, pp. 11 053–11 067, 2020.
- [64] Z. Li, D. Kovalev, X. Qian, and P. Richtárik, "Acceleration for compressed gradient descent in distributed and federated optimization," *arXiv preprint arXiv:2002.11364*, 2020.
- [65] S. Wang, T. Tuor, T. Salonidis, K. K. Leung, C. Makaya, T. He, and K. Chan, "When edge meets learning: Adaptive control for resource-constrained distributed machine learning," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 63–71.
- [66] C. Zhang, S. Li, J. Xia, W. Wang, F. Yan, and Y. Liu, "Batchcrypt: Efficient homomorphic encryption for cross-silo federated learning," in *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, 2020, pp. 493–506.
- [67] R. Liu, Y. Cao, H. Chen, R. Guo, and M. Yoshikawa, "Flame: Differentially private federated learning in the shuffle model," *arXiv preprint arXiv:2009.08063*, 2020.
- [68] K. Wei, J. Li, M. Ding, C. Ma, H. H. Yang, F. Farokhi, S. Jin, T. Q. Quek, and H. V. Poor, "Federated learning with differential privacy: Algorithms and performance analysis," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 3454–3469, 2020.
- [69] D. Wu, M. Pan, Z. Xu, Y. Zhang, and Z. Han, "Towards efficient secure aggregation for model update in federated learning," in *GLOBECOM 2020-2020 IEEE Global Communications Conference*. IEEE, 2020, pp. 1–6.
- [70] C. Niu, F. Wu, S. Tang, L. Hua, R. Jia, C. Lv, Z. Wu, and G. Chen, "Billion-scale federated learning on mobile clients: A submodel design with tunable privacy," in *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, 2020, pp. 1–14.
- [71] C. Zhou, A. Fu, S. Yu, W. Yang, H. Wang, and Y. Zhang, "Privacy-preserving federated learning in fog computing," *IEEE Internet of Things Journal*, vol. 7, no. 11, pp. 10 782–10 793, 2020.

TABLE VII

CHANGES IN STAGES OF TRAINING FLOW ABSTRACTION IN RECENT PUBLICATIONS FROM BOTH MACHINE LEARNING COMMUNITY AND SYSTEM COMMUNITY. AROUND 30% OF THEM CHANGE ONLY ONE STAGE IN FEDERATED LEARNING AND MAJORITY CHANGE (57%) ONLY TWO STAGES IN THE FEDAVG ALGORITHM.

Conference / Journal	Paper Title	Server			Client		
		Selection	Compression	Aggregation	Train	Compression	Encryption
IoT 2021	FedMCCS: Multicriteria Client Selection Model for Optimal IoT Federated Learning [45]	✓					
INFOCOM 2020	Optimizing Federated Learning on Non-IID Data with Reinforcement Learning [46]	✓					
OSDI 2021	Oort: Informed Participant Selection for Scalable Federated Learning [39]	✓					
HPDC 2020	TiFL: A Tier-based Federated Learning System [24]	✓					
KDD 2020	FedFast: Going Beyond Average for Faster Training of Federated Recommender Systems [12]	✓		✓			
TNNLS 2019	Robust and Communication-Efficient Federated Learning From Non-i.i.d. Data [47]		✓			✓	
NIPS 2020	Ensemble Distillation for Robust Model Fusion in Federated Learning [48]			✓			
ICDCS 2019	CMFL: Mitigating Communication Overhead for Federated Learning [49]					✓	
ICML 2020	FetchSGD: Communication-Efficient Federated Learning with Sketching [50]			✓		✓	
ICML 2020	SCAFFOLD: Stochastic Controlled Averaging for Federated Learning [51]			✓		✓	
TPDS 2020	FedSCR: Structure-Based Communication Reduction for Federated Learning [52]			✓		✓	
HotEdge 2018	eSGD: Communication Efficient Distributed Deep Learning on the Edge [53]					✓	
ICML 2020	Adaptive Federated Optimization [54]				✓		
CVPR 2021	Privacy-preserving Collaborative Learning with Automatic Transformation Search [55]				✓		
MLSys 2020	Federated Optimization in Heterogeneous Networks [23]				✓		
ICLR 2020	Federated Learning with Matched Averaging [35]			✓	✓		
ACMMM 2020	Performance Optimization for Federated Person Re-identification via Benchmark Analysis [13]			✓	✓		
NIPS 2020	Distributionally Robust Federated Averaging [56]			✓	✓		
NIPS 2020	Group Knowledge Transfer: Federated Learning of Large CNNs at the Edge [57]			✓	✓		
NIPS 2020	Personalized Federated Learning with Moreau Envelopes [58]			✓	✓		
ICLR 2020	Fair Resource Allocation in Federated Learning [59]			✓	✓		
ICML 2020	Federated Learning with Only Positive Labels [60]			✓	✓		
AAAI 2021	Addressing Class Imbalance in Federated Learning [61]			✓	✓		
AAAI 2021	Federated Block Coordinate Descent Scheme for Learning Global and Personalized Models [62]			✓	✓		
IoT 2020	Toward Communication-Efficient Federated Learning in the Internet of Things With Edge Computing [63]			✓	✓	✓	
ICML 2020	Acceleration for Compressed Gradient Descent in Distributed and Federated Optimization [64]			✓	✓	✓	
INFOCOMM 2018	When Edge Meets Learning: Adaptive Control for Resource-Constrained Distributed Machine Learning [65]			✓	✓		
ATC 2020	BatchCrypt: Efficient Homomorphic Encryption for Cross-Silo Federated Learning [66]			✓			✓
AAAI 2021	FLAME: Differentially Private Federated Learning in the Shuffle Model [67]			✓			✓
TIFS 2020	Federated Learning with Differential Privacy: Algorithms and Performance Analysis [68]			✓			✓
GLOBECOM 2020	Towards Efficient Secure Aggregation for Model Update in Federated Learning [69]			✓			✓
MobiCom 2020	Billion-Scale Federated Learning on Mobile Clients: A Submodel Design with Tunable Privacy [70]			✓	✓		✓
IoT 2020	Privacy-Preserving Federated Learning in Fog Computing [71]			✓	✓		✓