EasyFL

Release 0.1.2

EasyFL Authors

Aug 23, 2023

INTRODUCTION

CHAPTER

ONE

WHY EASYFL?

EasyFL is an easy-to-use federated learning platform that aims to enable users with various levels of expertise to experiment and prototype FL applications with little/no coding.

You can use it for:

- FL Research on algorithm and system
- Proof-of-concept (POC) of new FL applications
- Prototype of industrial applications
- Learning FL implementations

We currently focus on horizontal FL, supporting both cross-silo and cross-device FL. You can learn more about federated learning from these resources.

1.1 Major Features

Easy to Start

EasyFL is easy to install and easy to learn. It does not have complex dependency requirements. You can run EasyFL on your personal computer with only three lines of code (*Quick Start*).

Out-of-the-box Functionalities

EasyFL provides many out-of-the-box functionalities, including datasets, models, and FL algorithms. With simple configurations, you simulate different FL scenarios using the popular datasets. We support both statistical heterogeneity simulation and system heterogeneity simulation.

Flexible, Customizable, and Reproducible

EasyFL is flexible to be customized according to your needs. You can easily migrate existing CV or NLP applications into the federated manner by writing the PyTorch codes that you are most familiar with.

Multiple Training Modes

EasyFL supports **standalone training**, **distributed training**, and **remote training**. By developing the code once, you can easily speed up FL training with distributed training on multiple GPUs. Besides, you can even deploy it to Kubernetes with Docker using remote training.

We have developed many applications and published several *papers* in top-tier conferences and journals using EasyFL. We believe that EasyFL will empower you with FL research and development.

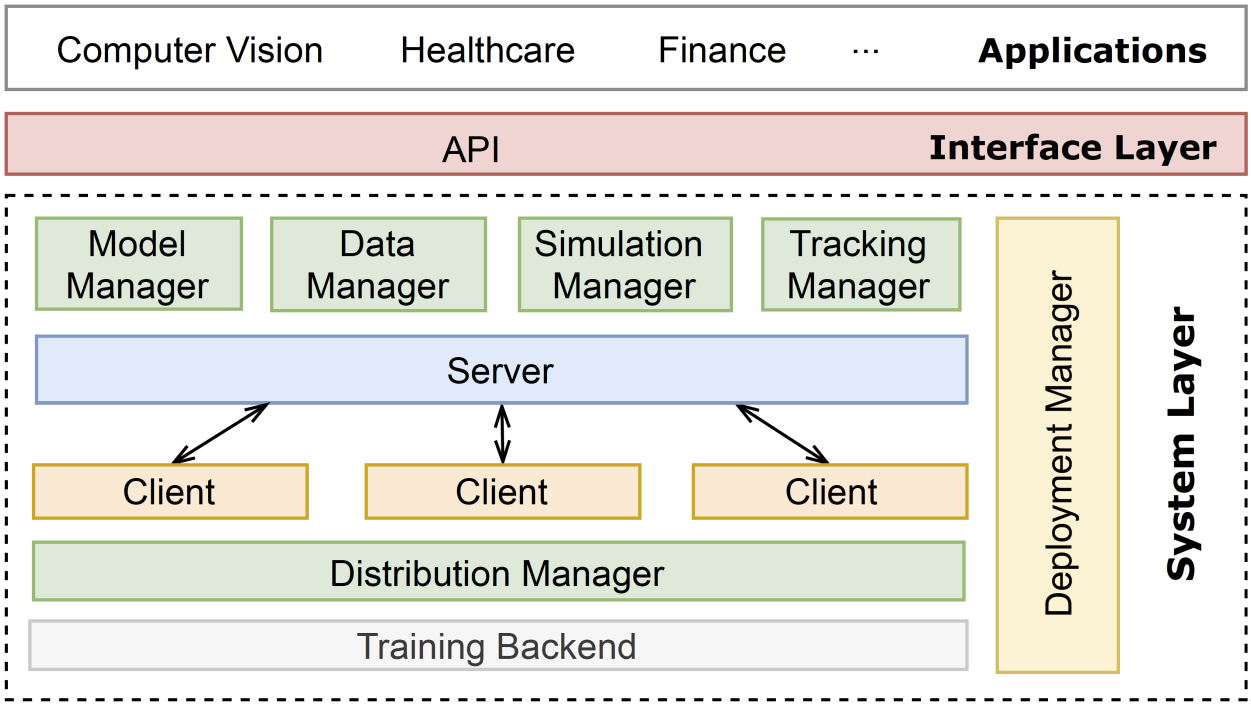
CHAPTER

TWO

ARCHITECTURE OVERVIEW

Here we introduce the architecture of EasyFL. You can jump directly to *Get Started* without knowing these details.

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.



architecture

Interface Layer: The interface layer provides a common interface across FL applications. It contains APIs that are designed to encapsulate complex system implementations from users. These APIs decouple application-specific models, datasets, 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.
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.

To learn more about EasyFL, you can check out our paper.

CHAPTER

THREE

PREREQUISITES

- Linux or macOS (Windows is in experimental support)
- Python 3.6+
- PyTorch 1.3+
- CUDA 9.2+ (If you run using GPU)

CHAPTER

FOUR

INSTALLATION

4.1 Prepare environment

1. Create a conda virtual environment and activate it.

```
conda create -n easyfl python=3.7 -y
conda activate easyfl
```

2. Install PyTorch and torchvision following the official instructions, e.g.,

```
conda install pytorch torchvision -c pytorch
```

or

```
pip install torch==1.10.1 torchvision==0.11.2
```

3. *You can skip the following CUDA-related content if you plan to run it on CPU.* Make sure that your compilation CUDA version and runtime CUDA version match.

Note: Make sure that your compilation CUDA version and runtime CUDA version match. You can check the supported CUDA version for precompiled packages on the PyTorch website.

`E.g.`, 1. If you have CUDA 10.1 installed under `/usr/local/cuda` and would like to install PyTorch 1.5, you need to install the prebuilt PyTorch with CUDA 10.1.

```
conda install pytorch cudatoolkit=10.1 torchvision -c pytorch
```

`E.g.`, 2. If you have CUDA 9.2 installed under `/usr/local/cuda` and would like to install PyTorch 1.3.1., you need to install the prebuilt PyTorch with CUDA 9.2.

```
conda install pytorch=1.3.1 cudatoolkit=9.2 torchvision=0.4.2 -c pytorch
```

If you build PyTorch from source instead of installing the prebuilt package, you can use more CUDA versions such as 9.0.

4.2 Install EasyFL

```
pip install easyfl
```

4.3 A from-scratch setup script

Assuming that you already have CUDA 10.1 installed, here is a full script for setting up MMDetection with conda.

```
conda create -n easyfl python=3.7 -y
conda activate easyfl

# Without GPU
conda install pytorch==1.6.0 torchvision==0.7.0 -c pytorch -y

# With GPU
conda install pytorch==1.6.0 torchvision==0.7.0 cudatoolkit=10.1 -c pytorch -y

# install easyfl
git clone https://github.com/EasyFL-AI/easyfl.git
cd easyfl
pip install -v -e .
```

CHAPTER

FIVE

VERIFICATION

To verify whether EasyFL is installed correctly, we can run the following sample code to test.

```
import easyfl

easyfl.init()
```

The above code is supposed to run successfully after you finish the installation.

CHAPTER

SIX

HIGH-LEVEL INTRODUCTION

EasyFL provides numerous existing models and datasets. Models include LeNet, RNN, VGG9, and ResNet. Datasets include Femnist, Shakespeare, CIFAR-10, and CIFAR-100. This note will present how to start training with these existing models and standard datasets.

EasyFL provides three types of high-level APIs: **registration**, **initialization**, and **execution**. Registration is for registering customized components, which we will introduce in the following notes. In this note, we focus on **initialization** and **execution**.

CHAPTER

SEVEN

SIMPLEST RUN

We can run federated learning with only two lines of code (not counting the import statement). It executes training with default configurations: simulating 100 clients with the FEMNIST dataset and randomly selecting 5 clients for training in each training round. We explain more about the configurations in *another note*.

Note: we package default partitioning of Femnist data to avoid downloading the whole dataset.

```
import easyfl

# Initialize federated learning with default configurations.
easyfl.init()
# Execute federated learning training.
easyfl.run()
```

CHAPTER

EIGHT

RUN WITH CONFIGURATIONS

You can specify configurations to overwrite the default configurations.

```
import easyfl

# Customized configuration.
config = {
    "data": {"dataset": "cifar10", "split_type": "class", "num_of_clients": 100},
    "server": {"rounds": 5, "clients_per_round": 2},
    "client": {"local_epoch": 5},
    "model": "resnet18",
    "test_mode": "test_in_server",
}
# Initialize federated learning with default configurations.
easyfl.init(config)
# Execute federated learning training.
easyfl.run()
```

In the example above, we run training with model ResNet-18 and CIFAR-10 dataset that is partitioned into 100 clients by label `class`. It runs training with 2 clients per round for 5 rounds. In each round, each client trains 5 epochs.

CHAPTER

NINE

TUTORIAL 1: HIGH-LEVEL APIS

EasyFL provides three types of high-level APIs: **initialization**, **registration**, and **execution**. The initialization API initializes EasyFL with configurations. Registration APIs register customized components into the platform. Execution APIs start federated learning process. These APIs are listed in the table below.

`init(config)`: Initialize EasyFL with provided configurations (`config`) or default configurations if not specified.These configurations determine the training hardware and hyperparameters.

`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.

`run, start_<server/client>`: The APIs are commands to trigger execution. `run()` starts FL using standalone training or distributed training. `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.

Next, we introduce how to use these APIs with examples.

9.1 Standalone Training Example

Standalone training means that federated learning (FL) training is run on a single hardware device, such as your personal computer and a single GPU. ***Distributed training*** means conducting FL with multiple GPUs to speed up training. Running distributed training is similar to standalone training, except that we need to configure the number of GPUs and the distributed settings. We explain more on distributed training in another note and focus on standalone training example here.

To run any federated learning process, we need to first call the initialization API and then use the execution API. Registration is optional.

The simplest way is to run with the default setup.

```
import easyfl
# Initialize federated learning with default configurations.
easyfl.init()
# Execute federated learning training.
easyfl.run()
```

You can run it with specified configurations.

```
import easyfl

# Customized configuration.
config = {
```

(continues on next page)

(continued from previous page)

```
    "data": {"dataset": "cifar10", "num_of_clients": 1000},
    "server": {"rounds": 5, "clients_per_round": 2, "test_all": False},
    "client": {"local_epoch": 5},
    "model": "resnet18",
    "test_mode": "test_in_server",
}
# Initialize federated learning with default configurations.
easyfl.init(config)
# Execute federated learning training.
easyfl.run()
```

You can also run federated learning with customized datasets, model, server and client implementations.

Note: `registration` must be done before `initialization`.

```
import easyfl
from easyfl.client import BaseClient

# Inherit BaseClient to implement customized client operations.
class CustomizedClient(BaseClient):
    def __init__(self, cid, conf, train_data, test_data, device, **kwargs):
        super(CustomizedClient, self).__init__(cid, conf, train_data, test_data, device,␣
↪**kwargs)
        pass  # more initialization of attributes.

    def train(self, conf, device):
        pass # Implement customized training method, overwriting the default one.

# Register customized client.
easyfl.register_client(CustomizedClient)
# Initialize federated learning with default configurations.
easyfl.init()
# Execute federated learning training.
easyfl.run()
```

9.2 Remote Training Example

Remote training is the scenario where the server and the clients are running on different devices. We explain more on remote training in another note. Here we provide examples on how to start client and server services using the APIs.

Start remote server.

```
import easyfl
# Configurations for the remote server.
conf = {"is_remote": True, "local_port": 22999}
# Initialize only the configuration.
easyfl.init(conf, init_all=False)
# Start remote server service.
# The remote server waits to be connected with the remote client.
easyfl.start_server()
```

Start remote client.

```
import easyfl
# Configurations for the remote client.
conf = {"is_remote": True, "local_port": 23000}
# Initialize only the configuration.
easyfl.init(conf, init_all=False)
# Start remote client service.
# The remote client waits to be connected with the remote server.
easyfl.start_client()
```

We expose two additional APIs that wrap starting remote services with customized components. They are `start_remote_server` and `start_remote_client`. You can explore more in the API documentation.

CHAPTER

TEN

TUTORIAL 2: CONFIGURATIONS

Configurations in EasyFL are to control and config the federated learning (FL) training behavior. It instructs data simulation, the model for training, training hyperparameters, distributed training, etc.

We provide *default configs* in EasyFL, while there are two ways you can modify the configs of EasyFL: using Python and using a yaml file.

10.1 Modify Config

EasyFL provides two ways to modify the configurations: using Python dictionary and using a yaml file. Either way, if the new configs exist in the default configuration, they overwrite those specific fields. If the new configs do not exist, it adds them to the EasyFL configuration. Thus, you can either modify the default configurations or add new configurations based on your application needs.

10.1.1 1. Modify Using Python Dictionary

You can create a new Python dictionary to specify configurations. These configs take effect when you initialize EasyFL with them by calling `easyfl.init(config)`.

The examples provided in the previous tutorial demonstrate how to modify config via a Python dictionary.

```
import easyfl

# Define customized configurations.
config = {
    "data": {
        "dataset": "cifar10",
        "num_of_clients": 1000
    },
    "server": {
        "rounds": 5,
        "clients_per_round": 2
    },
    "client": {"local_epoch": 5},
    "model": "resnet18",
    "test_mode": "test_in_server",
}
# Initialize EasyFL with the new config.
easyfl.init(config)
```

(continues on next page)

(continued from previous page)

```
# Execute federated learning training.
easyfl.run()
```

10.1.2 2. Modify Using A Yaml File

You can create a new yaml file named `config.yaml` for configuration and load them into EasyFL.

```
import easyfl
# Define customized configurations in a yaml file.
config_file = "config.yaml"
# Load the yaml file as config.
config = easyfl.load_config(config_file)
# Initialize EasyFL with the new config.
easyfl.init(config)
# Execute federated learning training.
easyfl.run()
```

You can also combine these two methods of modifying configs.

```
import easyfl

# Define part of customized configs.
config = {
    "data": {
        "dataset": "cifar10",
        "num_of_clients": 1000
    },
    "server": {
        "rounds": 5,
        "clients_per_round": 2
    },
    "client": {"local_epoch": 5},
    "model": "resnet18",
    "test_mode": "test_in_server",
}

# Define part of configs in a yaml file.
config_file = "config.yaml"
# Load and combine these two configs.
config = easyfl.load_config(config_file, config)
# Initialize EasyFL with the new config.
easyfl.init(config)
# Execute federated learning training.
easyfl.run()
```

10.2 A Common Practice to Modify Configuration

Since some configurations are directly related to training, we may need to set them dynamically with different values.

For example, we may want to experiment with the effect of batch size and local epoch on federated learning. Instead of changing the value manually each time in configuration, you can pass the value in as command-line arguments and set the value with different commands.

```
import easyfl
import argparse

# Define command line arguments.
parser = argparse.ArgumentParser(description='Example')
parser.add_argument("--batch_size", type=int, default=32)
parser.add_argument("--local_epoch", type=int, default=5)
args = parser.parse_args()
print("args", args)

# Define customized configurations using the arguments.
config = {
    "client": {
        "batch_size": args.batch_size,
        "local_epoch": args.local_epoch,
    }
}
# Initialize EasyFL with the new config.
easyfl.init(config)
# Execute federated learning training.
easyfl.run()
```

10.3 Default Configurations

The followings are the default configurations in EasyFL. They are copied from `easyfl/config.yaml` on April, 2022.

We provide more details on how to simulate different FL scenarios with the out-of-the-box datasets in another note.

```
# The unique identifier for each federated learning task
task_id: ""

# Provide dataset and federated learning simulation related configuration.
data:
  # The root directory where datasets are stored.
  root: "./data/"
  # The name of the dataset, support: femnist, shakespeare, cifar10, and cifar100.
  dataset: femnist
  # The data distribution of each client, support: iid, niid (for femnist and␣
↪shakespeare), and dir and class (for cifar datasets).
    # `iid` means independent and identically distributed data.
    # `niid` means non-independent and identically distributed data for Femnist and␣
↪Shakespeare.
    # `dir` means using Dirichlet process to simulate non-iid data, for CIFAR-10 and␣
↪CIFAR-100 datasets.
```

(continues on next page)

(continued from previous page)

```
    # `class` means partitioning the dataset by label classes, for datasets like CIFAR-10,
↪ CIFAR-100.
  split_type: "iid"

  # The minimal number of samples in each client. It is applicable for LEAF datasets and␣
↪dir simulation of CIFAR-10 and CIFAR-100.
  min_size: 10
  # The fraction of data sampled for LEAF datasets. e.g., 10% means that only 10% of␣
↪total dataset size are used.
  data_amount: 0.05
  # The fraction of the number of clients used when the split_type is 'iid'.
  iid_fraction: 0.1
  # Whether partition users of the dataset into train-test groups. Only applicable to␣
↪femnist and shakespeare datasets.
    # True means partitioning users of the dataset into train-test groups.
    # False means partitioning each users' samples into train-test groups.
  user: False
  # The fraction of data for training; the rest are for testing.
  train_test_split: 0.9

  # The number of classes in each client. Only applicable when the split_type is 'class'.␣
↪
  class_per_client: 1
  # The targeted number of clients to construct.used in non-leaf dataset, number of␣
↪clients split into. for leaf dataset, only used when split type class.
  num_of_clients: 100

  # The parameter for Dirichlet distribution simulation, applicable only when split_type␣
↪is `dir` for CIFAR datasets.
  alpha: 0.5

  # The targeted distribution of quantities to simulate data quantity heterogeneity.
    # The values should sum up to 1. e.g., [0.1, 0.2, 0.7].
    # The `num_of_clients` should be divisible by `len(weights)`.
    # None means clients are simulated with the same data quantity.
  weights: NULL

# The name of the model for training, support: lenet, rnn, resnet, resnet18, resnet50,␣
↪vgg9.
model: lenet
# How to conduct testing, options: test_in_client or test_in_server.
  # `test_in_client` means that each client has a test set to run testing.
  # `test_in_server` means that server has a test set to run testing for the global model.
↪ Use this mode for cifar datasets.
test_mode: "test_in_client"
# The way to measure testing performance (accuracy) when test mode is `test_in_client`,␣
↪support: average or weighted (means weighted average).
test_method: "average"

server:
  track: False  # Whether track server metrics using the tracking service.
  rounds: 10  # Total training round.
```

(continues on next page)

(continued from previous page)

```
  clients_per_round: 5  # The number of clients to train in each round.
  test_every: 1  # The frequency of testing: conduct testing every N round.
  save_model_every: 10  # The frequency of saving model: save model every N round.
  save_model_path: ""  # The path to save model. Default path is root directory of the␣
↪library.
  batch_size: 32  # The batch size of test_in_server.
  test_all: True  # Whether test all clients or only selected clients.
  random_selection: True  # Whether select clients to train randomly.
  # The strategy to aggregate client uploaded models, options: FedAvg, equal.
    # FedAvg aggregates models using weighted average, where the weights are data size␣
↪of clients.
    # equal aggregates model by simple averaging.
  aggregation_strategy: "FedAvg"
  # The content of aggregation, options: all, parameters.
    # all means aggregating models using state_dict, including both model parameters and␣
↪persistent buffers like BatchNorm stats.
    # parameters means aggregating only model parameters.
  aggregation_content: "all"

client:
  track: False  # Whether track server metrics using the tracking service.
  batch_size: 32  # The batch size of training in client.
  test_batch_size: 5  # The batch size of testing in client.
  local_epoch: 10  # The number of epochs to train in each round.
  optimizer:
    type: "Adam"  # The name of the optimizer, options: Adam, SGD.
    lr: 0.001
    momentum: 0.9
    weight_decay: 0
  seed: 0
  local_test: False  # Whether test the trained models in clients before uploading them␣
↪to the server.

gpu: 0  # The total number of GPUs used in training. 0 means CPU.
distributed:  # The distributed training configurations. It is only applicable when gpu >
↪ 1.
  backend: "nccl"  # The distributed backend.
  init_method: ""
  world_size: 0
  rank: 0
  local_rank: 0

tracking:  # The configurations for logging and tracking.
  database: ""  # The path of local dataset, sqlite3.
  log_file: ""
  log_level: "INFO"  # The level of logging.
  metric_file: ""
  save_every: 1

# The configuration for system heterogeneity simulation.
resource_heterogeneous:
  simulate: False  # Whether simulate system heterogeneity in federated learning.
```

(continues on next page)

(continued from previous page)

```
  # The type of heterogeneity to simulate, support iso, dir, real.
    # iso means that
  hetero_type: "real"
  level: 3  # The level of heterogeneous (0-5), 0 means no heterogeneous among clients.
  sleep_group_num: 1000  # The number of groups with different sleep time. 1 means all␣
↪clients are the same.
  total_time: 1000  # The total sleep time of all clients, unit: second.
  fraction: 1  # The fraction of clients attending heterogeneous simulation.
  grouping_strategy: "greedy"  # The grouping strategy to handle system heterogeneity,␣
↪support: random, greedy, slowest.
  initial_default_time: 5  # The estimated default training time for each training round,
↪ unit: second.
  default_time_momentum: 0.2  # The default momentum for default time update.

seed: 0  # The random seed.
```

10.3.1 Default Config without Comments

```
task_id: ""
data:
  root: "./data/"
  dataset: femnist
  split_type: "iid"

  min_size: 10
  data_amount: 0.05
  iid_fraction: 0.1
  user: False

  class_per_client: 1
  num_of_clients: 100
  train_test_split: 0.9
  alpha: 0.5

  weights: NULL

model: lenet
test_mode: "test_in_client"
test_method: "average"

server:
  track: False
  rounds: 10
  clients_per_round: 5
  test_every: 1
  save_model_every: 10
  save_model_path: ""
  batch_size: 32
  test_all: True
  random_selection: True
```

(continues on next page)

(continued from previous page)

```
  aggregation_strategy: "FedAvg"
  aggregation_content: "all"

client:
  track: False
  batch_size: 32
  test_batch_size: 5
  local_epoch: 10
  optimizer:
    type: "Adam"
    lr: 0.001
    momentum: 0.9
    weight_decay: 0
  seed: 0
  local_test: False

gpu: 0
distributed:
  backend: "nccl"
  init_method: ""
  world_size: 0
  rank: 0
  local_rank: 0

tracking:
  database: ""
  log_file: ""
  log_level: "INFO"
  metric_file: ""
  save_every: 1

resource_heterogeneous:
  simulate: False
  hetero_type: "real"
  level: 3
  sleep_group_num: 1000
  total_time: 1000
  fraction: 1
  grouping_strategy: "greedy"
  initial_default_time: 5
  default_time_momentum: 0.2

seed: 0
```

CHAPTER

ELEVEN

TUTORIAL 3: DATASETS

In this note, we present how to use the out-of-the-box datasets to simulate different federated learning (FL) scenarios. Besides, we introduce how to use the customized dataset in EasyFL.

We currently provide four out-of-the-box datasets: FEMNIST, Shakespeare, CIFAR-10, and CIFAR-100. FEMNIST and Shakespeare are adopted from LEAF benchmark. We plan to integrate and provide more out-of-the-box datasets in the future.

11.1 Out-of-the-box Datasets

The simulation of different FL scenarios is configured in the configurations. You can refer to the other tutorial to learn more about how to modify configs. In this note, we focus on how to config the datasets with different simulations.

The following are dataset configurations.

```
data:
  # The root directory where datasets are stored.
  root: "./data/"
  # The name of the dataset, support: femnist, shakespeare, cifar10, and cifar100.
  dataset: femnist
    # The data distribution of each client, support: iid, niid (for femnist and␣
↪shakespeare), and dir and class (for cifar datasets).
    # `iid` means independent and identically distributed data.
    # `niid` means non-independent and identically distributed data for FEMNIST and␣
↪Shakespeare.
    # `dir` means using Dirichlet process to simulate non-iid data, for CIFAR-10 and␣
↪CIFAR-100 datasets.
  # `class` means partitioning the dataset by label classes, for datasets like CIFAR-10,␣
↪CIFAR-100.
  split_type: "iid"

  # The minimal number of samples in each client. It is applicable for LEAF datasets and␣
↪dir simulation of CIFAR-10 and CIFAR-100.
  min_size: 10
  # The fraction of data sampled for LEAF datasets. e.g., 10% means that only 10% of the␣
↪total dataset size is used.
  data_amount: 0.05
  # The fraction of the number of clients used when the split_type is 'iid'.
  iid_fraction: 0.1
    # Whether partition users of the dataset into train-test groups. Only applicable to␣
↪femnist and shakespeare datasets.
```

(continues on next page)

(continued from previous page)

```
    # True means partitioning users of the dataset into train-test groups.
  # False means partitioning each users' samples into train-test groups.
  user: False
  # The fraction of data for training; the rest are for testing.
  train_test_split: 0.9

  # The number of classes in each client. Only applicable when the split_type is 'class'.␣
↪
  class_per_client: 1
  # The targeted number of clients to construct.used in non-leaf dataset, number of␣
↪clients split into. for leaf dataset, only used when split type class.
  num_of_clients: 100
  # The parameter for Dirichlet distribution simulation, applicable only when split_type␣
↪is `dir` for CIFAR datasets.
  alpha: 0.5

    # The targeted distribution of quantities to simulate data quantity heterogeneity.
    # The values should sum up to 1. e.g., [0.1, 0.2, 0.7].
    # The `num_of_clients` should be divisible by `len(weights)`.
  # None means clients are simulated with the same data quantity.
  weights: NULL
```

Among them, `root` is applicable to all datasets. It specifies the directory to store datasets.

EasyFL automatically downloads a dataset if it is not exist in the root directory.

Next, we introduce the simulation and configuration for specific datasets.

11.1.1 FEMNIST and Shakespeare Datasets

The following are basic stats of these two datasets.

FEMNIST

- Overview: Image Dataset
- Details: 3500 users, 62 different classes (10 digits, 26 lowercase, 26 uppercase), images are 28 by 28 pixels (with option to make them all 128 by 128 pixels)
- Task: Image Classification

Shakespeare

- Overview: Text Dataset of Shakespeare Dialogues
- Details: 1129 users (reduced to 660 with our choice of sequence length.)
- Task: Next-Character Prediction

The datasets are non-IID (independent and identically distributed) in nature.

`split_type`: There are two options for these two datasets: `iid` and `niid`, representing IID data simulation and non-IID data simulation.

Five hyper-parameters determine the simulated dataset: `min_size`, `data_amount`, `iid_fraction`, `tran_test_split`, and `user`.

`user` is a boolean that determines whether to partition the dataset to train test group by user or samples. `user: True` means partitioning users of the dataset into train-test groups, i.e. some users are for training, some users are for testing.

`user: False` means partitioning each users' samples into train-test groups, i.e. data in each client is partitioned into training set and testing set.

Note: we normally use `test_mode: test_in_clients` for these two datasets.

IID Simulation

In IID simulation, data are randomly partitioned into multiple clients.

The number of clients is determined by `data_amount` and `iid_fraction`.

Non-IID Simulation

Since FEMNIST and Shakespeare are non-IID in nature, each user of the dataset is regarded as a client.

`data_amount` determine the number of clients participate in training.

11.1.2 CIFAR-10 and CIFAR-100 Datasets

> The **CIFAR-10** dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.
>
> The **CIFAR-100** dataset consists of 60000 32x32 colour images in 100 classes, with 600 images per class. There are 50000 training images and 10000 test images.

`split_type`: There are three options for CIFAR datasets: `iid`, `dir`, and `class`.

Three hyper-parameters determine the simulated dataset: `num_of_clients`, `class_per_client`, and `alpha`.

IID Simulation

In IID simulation, the training images of the datasets are randomly partitioned into `num_of_clients` clients.

Non-IID Simulation

We can simulate non-IID CIFAR datasets by Dirichlet process (`dir`) or by label class (`class`).

`alpha` controls the level of heterogeneity for `dir` simulation.

`class_per_client` determines the number of classes in each client.

11.2 Customize Datasets

EasyFL also supports integrating with customized dataset to simulate federated learning.

You can use the following classes to integrate customized dataset: FederatedImageDataset, FederatedTensorDataset, and FederatedTorchDataset.

The following is an example that integrates nine person re-identification datasets, where each client contains one dataset.

```
import easyfl
import os
from torchvision import transforms
from easyfl.datasets import FederatedImageDataset

TRANSFORM_TRAIN_LIST = transforms.Compose([
    transforms.Resize((256, 128), interpolation=3),
    transforms.Pad(10),
    transforms.RandomCrop((256, 128)),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])
TRANSFORM_VAL_LIST = transforms.Compose([
    transforms.Resize(size=(256, 128), interpolation=3),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

DATASETS = ["MSMT17", "Duke", "Market", "cuhk03", "prid", "cuhk01", "viper", "3dpes",
↪"ilids"]

# Prepare customized training data
def prepare_train_data(data_dir):
    client_ids = []
    roots = []
    for db in DATASETS:
        client_ids.append(db)
        data_path = os.path.join(data_dir, db, "pytorch")
        roots.append(os.path.join(data_path, "train_all"))
    data = FederatedImageDataset(root=roots,
                                 simulated=True,
                                 do_simulate=False,
                                 transform=TRANSFORM_TRAIN_LIST,
                                 client_ids=client_ids)
    return data


# Prepare customized testing data
def prepare_test_data(data_dir):
    roots = []
    client_ids = []
    for db in DATASETS:
        test_gallery = os.path.join(data_dir, db, 'pytorch', 'gallery')
        test_query = os.path.join(data_dir, db, 'pytorch', 'query')
        roots.extend([test_gallery, test_query])
        client_ids.extend([f"{db}_gallery", f"{db}_query"])
    data = FederatedImageDataset(root=roots,
                                 simulated=True,
                                 do_simulate=False,
                                 transform=TRANSFORM_VAL_LIST,
                                 client_ids=client_ids)
    return data
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    config = {...}
    data_dir = "datasets/"
    train_data, test_data = prepare_train_data(data_dir), prepare_test_data(data_dir)
    easyfl.register_dataset(train_data, test_data)
    easyfl.init(config)
    easyfl.run()
```

The folder structure of these datasets are as followed:

```
|-- MSMT17
|   |-- pytorch
|   |            |-- gallery
|   |     |-- query
|   |     |-- train
|   |     |-- train_all
|   |     `-- val
|-- cuhk01
|   |-- pytorch
|   |            |-- gallery
|   |     |-- query
|   |     |-- train
|   |     |-- train_all
| ...
```

Please email us if you want to access these datasets with:

1. A short self-introduction.
2. The purposes of using these datasets.

Further distribution of the datasets are prohibited.

11.2.1 Create Your Own Federated Dataset

In case that the provided federated dataset class is not enough, you can implement your own federated dataset by inherit and implement FederatedDataset.

You can refer to FederatedImageDataset, FederatedTensorDataset, and FederatedTorchDataset on how to implement.

CHAPTER

TWELVE

TUTORIAL 4: MODELS

EasyFL supports numerous models and allows you to customize the model.

12.1 Out-of-the-box Models

To use these models, you can set configurations `model: <model_name>`.

We currently provide `lenet`, `resnet`, `resnet18`, `resnet50`,`vgg9`, and `rnn`.

12.2 Customized Models

EasyFL allows training with a wide range of models by providing the flexibility to customize models. You can customize and register models in two ways: register as a class and register as an instance. Either way, the basic is to **inherit and implement the `easyfl.models.BaseModel`**.

12.2.1 Register as a Class

In the example below, we implement and conduct FL training with a `CustomizedModel`.

It is applicable when the model does not require extra arguments to initialize.

```
from torch import nn
import torch.nn.functional as F
import easyfl
from easyfl.models import BaseModel

# Define a customized model class.
class CustomizedModel(BaseModel):
    def __init__(self):
        super(CustomizedModel, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, 224, padding=(2, 2))
        self.conv2 = nn.Conv2d(32, 64, 5, padding=(2, 2))
        self.fc1 = nn.Linear(64, 128)
        self.fc2 = nn.Linear(128, 42)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, 2, 2)
```

(continues on next page)

(continued from previous page)

```
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2, 2)
        x = x.view(-1, 64)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)

        return x

# Register the customized model class.
easyfl.register_model(CustomizedModel)
# Initialize EasyFL.
easyfl.init()
# Execute FL training.
easyfl.run()
```

12.2.2 Register as an Instance

When the model requires arguments for initialization, we can implement and register a model instance.

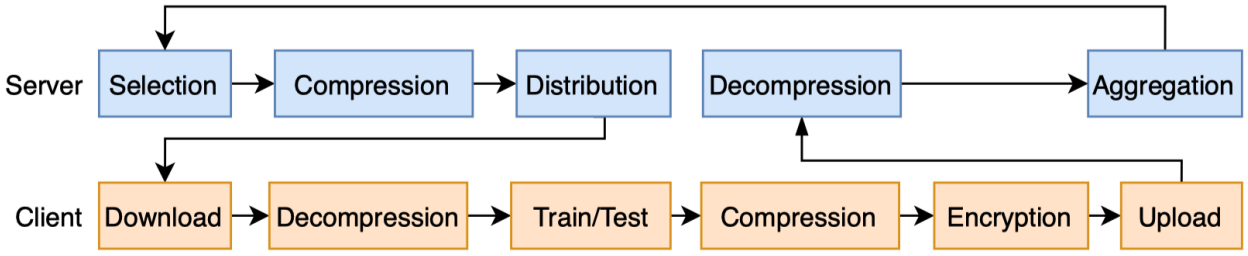
```
from torch import nn
import torch.nn.functional as F
import easyfl
from easyfl.models import BaseModel

# Define a customized model class.
class CustomizedModel(BaseModel):
    def __init__(self, num_class):
        super(CustomizedModel, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, 224, padding=(2, 2))
        self.conv2 = nn.Conv2d(32, 64, 5, padding=(2, 2))
        self.fc1 = nn.Linear(64, 128)
        self.fc2 = nn.Linear(128, num_class)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, 2, 2)
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2, 2)
        x = x.view(-1, 64)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# Register the customized model instance.
easyfl.register_model(CustomizedModel(num_class=10))
# Initialize EasyFL.
easyfl.init()
# Execute FL training.
easyfl.run()
```

CHAPTER

THIRTEEN

TUTORIAL 5: CUSTOMIZE SERVER AND CLIENT

EasyFL abstracts the federated learning (FL) training flow in the server and the client into granular stages, as shown in the image below.



Training Flow

You have the flexibility to customize any stage of the training flow while reusing the rest by implementing a customized server/client.

13.1 Customize Server

EasyFL implements random client selection and Federated Averaging as the aggregation strategy. You can customize the server implementation by inheriting BaseServer and override specific functions.

Below is an example of a customized server.

```
import easyfl
from easyfl.server import BaseServer
from easyfl.server.base import MODEL

class CustomizedServer(BaseServer):
    def __init__(self, conf, **kwargs):
        super(CustomizedServer, self).__init__(conf, **kwargs)
        pass  # more initialization of attributes.

    def aggregation(self):
        uploaded_content = self.get_client_uploads()
        models = list(uploaded_content[MODEL].values())
        # Original implementation of aggregation weights
        # weights = list(uploaded_content[DATA_SIZE].values())
        # We can assign the manipulated customized weights in aggregation.
        customized_weights = list(range(len(models)))
        model = self.aggregate(models, customized_weights)
```

(continues on next page)

(continued from previous page)

```
        self.set_model(model, load_dict=True)

# Register customized server.
easyfl.register_server(CustomizedServer)
# Initialize federated learning with default configurations.
easyfl.init()
# Execute federated learning training.
easyfl.run()
```

Here we list down more useful functions to override to implement a customized server.

```
import easyfl
from easyfl.server import BaseServer

class CustomizedServer(BaseServer):
    def __init__(self, conf, **kwargs):
        super(CustomizedServer, self).__init__(conf, **kwargs)
        pass  # more initialization of attributes.

    def selection(self, clients, clients_per_round):
        pass  # implement customized client selection algorithm.

    def compression(self):
        pass  # implement customized compression algorithm.

    def pre_train(self):
        pass  # inject operations before distribution to train.

    def post_train(self):
        pass  # inject operations after aggregation.

    def pre_test(self):
        pass  # inject operations before distribution to test.

    def post_test(self):
        pass  # inject operations after aggregating testing results.

    def decompression(self, model):
        pass  # implement customized decompression algorithm.

    def aggregation(self):
        pass  # implement customized aggregation algorithm.
```

Below are some attributes that you may need in implementing the customized server.

`self.conf`: Configurations of EasyFL.

`self._model`: The global model in server, updated after aggregation.

`self._current_round`: The current training round.

`self._clients`: All available clients.

`self.selected_clients`: The selected clients.

You may refer to the BaseServer for more functions and class attributes.

13.2 Customize Client

Each client of EasyFL conducts training and testing. The implementation of training and testing is similar to normal PyTorch implementation. We implement training with Adam/SGD optimizer using CrossEntropy loss. You can customize client implementation of training and testing by inheriting BaseClient and overriding specific functions.

Below is an example of a customized client.

```
import time
import easyfl
from torch import nn
import torch.optim as optim
from easyfl.client.base import BaseClient

# Inherit BaseClient to implement customized client operations.
class CustomizedClient(BaseClient):
    def __init__(self, cid, conf, train_data, test_data, device, **kwargs):
        super(CustomizedClient, self).__init__(cid, conf, train_data, test_data, device,␣
↪**kwargs)
        # Initialize a classifier for each client.
        self.classifier = nn.Sequential(*[nn.Linear(512, 100)])

    def train(self, conf, device):
        start_time = time.time()
        self.model.classifier.classifier = self.classifier.to(device)
        loss_fn, optimizer = self.pretrain_setup(conf, device)
        self.train_loss = []
        for i in range(conf.local_epoch):
            batch_loss = []
            for batched_x, batched_y in self.train_loader:
                x, y = batched_x.to(device), batched_y.to(device)
                optimizer.zero_grad()
                out = self.model(x)
                loss = loss_fn(out, y)
                loss.backward()
                optimizer.step()
                batch_loss.append(loss.item())
            current_epoch_loss = sum(batch_loss) / len(batch_loss)
            self.train_loss.append(float(current_epoch_loss))
        self.train_time = time.time() - start_time
        # Keep the classifier in clients and upload only the backbone of model.
        self.classifier = self.model.classifier.classifier
        self.model.classifier.classifier = nn.Sequential()

    # A customized optimizer that sets different learning rates for different model␣
↪parts.
    def load_optimizer(self, conf):
        ignored_params = list(map(id, self.model.classifier.parameters()))
        base_params = filter(lambda p: id(p) not in ignored_params, self.model.
↪parameters())
        optimizer = optim.SGD([
            {'params': base_params, 'lr': 0.1 * conf.optimizer.lr},
            {'params': self.model.classifier.parameters(), 'lr': conf.optimizer.lr}
```

(continues on next page)

(continued from previous page)

```
        ], weight_decay=5e-4, momentum=conf.optimizer.momentum, nesterov=True)
        return optimizer

# Register customized client.
easyfl.register_client(CustomizedClient)
# Initialize federated learning with default configurations.
easyfl.init()
# Execute federated learning training.
easyfl.run()
```

Here we list down more useful functions to override to implement a customized client.

```
import easyfl
from easyfl.client import BaseClient

# Inherit BaseClient to implement customized client operations.
class CustomizedClient(BaseClient):
    def __init__(self, cid, conf, train_data, test_data, device, **kwargs):
        super(CustomizedClient, self).__init__(cid, conf, train_data, test_data, device,␣
↪**kwargs)
        pass  # more initialization of attributes.

    def decompression(self):
        pass  # implement decompression method.

    def pre_train(self):
        pass  # inject operations before training.

    def train(self, conf, device):
        pass  # implement customized training method.

    def post_train(self):
        pass  # inject operations after training.

    def load_loss_fn(self, conf):
        pass  # load a customized loss function.
        return loss

    def load_optimizer(self, conf):
        pass  # load a customized optimizer
        return optimizer

    def load_loader(self, conf):
        pass  # load a customized data loader.
        return train_loader

    def test_local(self):
        pass  # implement testing of the trained model before uploading to the server.

    def pre_test(self):
        pass  # inject operations before testing.
```

(continues on next page)

(continued from previous page)

```
    def test(self, conf, device):
        pass  # implement customized testing.

    def post_test(self):
        pass  # inject operations after testing.

    def encryption(self):
        pass  # implement customized encryption method.

    def compression(self):
        pass  # implement customized compression method.

    def upload(self):
        pass  # implement customized upload method.

    def post_upload(self):
        pass  # implement customized post upload method.
```

Below are some attributes that you may need in implementing the customized client.

`self.conf`: Configurations of client, under key "client" of config dictionary.

`self.compressed_model`: The model downloaded from the server.

`self.model`: The model used for training.

`self.cid`: The client id.

`self.device`: The device for training.

`self.train_data`: The training data of the client.

`self.test_data`: The testing data of the client.

You may refer to the BaseClient for more functions and class attributes.

13.3 Existing Works

We surveyed 33 papers from recent publications of FL from both the machine learning and system community. The following table shows that 10 out of 33 (~30%) publications propose new algorithms with changes in only one stage of the training flow, and the majority (~57%) change only two stages. Training flow abstraction you to focus on the problems, without re-implementing the whole FL process.

Annotation of the table:

Server stages: **Sel** – Selection, **Com** – Compression, **Agg** – Aggregation

Client stages: **Train**, **Com** – Compression, **Enc** – Encryption

CHAPTER

FOURTEEN

TUTORIAL 6: DISTRIBUTED TRAINING

EasyFL enables federated learning (FL) training over multiple GPUs. We define the following variables to further illustrate the idea:

- K: the number of clients who participated in training each round
- N: the number of available GPUs

When $K == N$, each selected client is allocated to a GPU to train.

When $K > N$, multiple clients are allocated to a GPU, then they execute training sequentially in the GPU.

When $K < N$, you can adjust to use fewer GPUs in training.

We make it easy to use distributed training. You just need to modify the configs, without changing the core implementations. In particular, you need to set the number of GPUs in `gpu` and specific distributed settings in the `distributed` configs.

The following is an example of distributed training on a GPU cluster managed by *slurm*.

```
import easyfl
from easyfl.distributed import slurm

# Get the distributed settings.
rank, local_rank, world_size, host_addr = slurm.setup()
# Set the distributed training settings.
config = {
    "gpu": world_size,
    "distributed": {
        "rank": rank,
        "local_rank": local_rank,
        "world_size": world_size,
        "init_method": host_addr
    },
}
# Initialize EasyFL.
easyfl.init(config)
# Execute training with distributed training.
easyfl.run()
```

We will further provide scripts to set up distributed training using `multiprocess`. Pull requests are also welcomed.

CHAPTER

FIFTEEN

TUTORIAL 7: REMOTE TRAINING

Remote training is the scenario where the server and the clients are running on different devices. Standalone and distributed training are mainly for federated learning (FL) simulation experiments. Remote training brings FL from experimentation to production.

15.1 Remote Training Example

In remote training, both server and clients are started as gRPC services. Here we provide examples on how to start server and client services.

Start remote server.

```
import easyfl

# Configurations for the remote server.
conf = {"is_remote": True, "local_port": 22999}
# Initialize only the configuration.
easyfl.init(conf, init_all=False)
# Start remote server service.
# The remote server waits to be connected with the remote client.
easyfl.start_server()
```

Start remote client 1 with port 23000.

```
import easyfl

# Configurations for the remote client.
conf = {
    "is_remote": True,
    "local_port": 23000,
    "server_addr": "localhost:22999",
    "index": 0,
}
# Initialize only the configuration.
easyfl.init(conf, init_all=False)
# Start remote client service.
# The remote client waits to be connected with the remote server.
easyfl.start_client()
```

Start remote client 2 with port 23001.

```
import easyfl

# Configurations for the remote client.
conf = {
    "is_remote": True,
    "local_port": 23001,
    "server_addr": "localhost:22999",
    "index": 1,
}
# Initialize only the configuration.
easyfl.init(conf, init_all=False)
# Start remote client service.
# The remote client waits to be connected with the remote server.
easyfl.start_client()
```

The client service connects to the remote service via specified `server_address`. The client service users `index` to decide the data (user) of the configured dataset.

To trigger remote training, we can send gRPC requests to trigger the training operation.

```
import easyfl
from easyfl.pb import common_pb2 as common_pb
from easyfl.pb import server_service_pb2 as server_pb
from easyfl.protocol import codec
from easyfl.communication import grpc_wrapper
from easyfl.registry.vclient import VirtualClient

server_addr = "localhost:22999"
config = {
    "data": {"dataset": "femnist"},
    "model": "lenet",
    "test_mode": "test_in_client"
}
# Initialize configurations.
easyfl.init(config, init_all=False)
# Initialize the model, using the configured 'lenet'
model = easyfl.init_model()

# Construct gRPC request
stub = grpc_wrapper.init_stub(grpc_wrapper.TYPE_SERVER, server_addr)
request = server_pb.RunRequest(model=codec.marshal(model))
# The request contains clients' addresses for the server to communicate with the clients.
clients = [VirtualClient("1", "localhost:23000", 0), VirtualClient("2", "localhost:23001
↪", 1)]
for c in clients:
    request.clients.append(server_pb.Client(client_id=c.id, index=c.index, address=c.
↪address))
# Send request to trigger training.
response = stub.Run(request)
result = "Success" if response.status.code == common_pb.SC_OK else response
print(result)
```

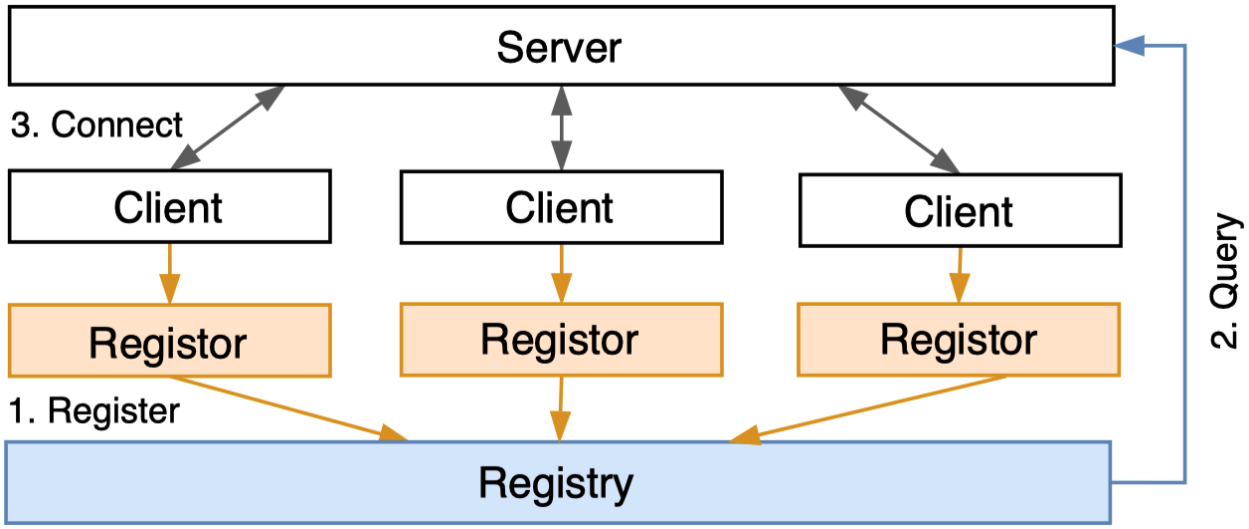
Similarly, we can also stop remote training by sending gRPC requests to the server.

```
from easyfl.communication import grpc_wrapper
from easyfl.pb import common_pb2 as common_pb
from easyfl.pb import server_service_pb2 as server_pb

server_addr = "localhost:22999"
stub = grpc_wrapper.init_stub(grpc_wrapper.TYPE_SERVER, server_addr)
# Send request to stop training.
response = stub.Stop(server_pb.StopRequest())
result = "Success" if response.status.code == common_pb.SC_OK else response
print(result)
```

15.2 Remote Training on Docker and Kubernetes

EasyFL supports deployment of FL training using Docker and Kubernetes.

Since we cannot easily obtain the server and client addresses in Docker or Kubernetes, especially when scaling up the number of clients, EasyFL provides a service discovery mechanism, as shown in the image below.



service_discovery

It contains registors 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: using Docker and using Kubernetes

The following are the deployment manual and the steps to conduct training in Kubernetes.

Note: these commands were tested before refactoring. They may not work as expected now. **Need further testing**.

15.2.1 Deployment using Docker

Important: Adjust the `Memeory` constrain of docker to be > 11 GB (To be optimized)

1. Build docker images and start services with either docker compose or individual docker containers
2. Start training with a grpc message

Build images

```
make base_image
make image
```

Or

```
docker build -t easyfl:base -f docker/base.Dockerfile .
docker build -t easyfl-client -f docker/client.Dockerfile .
docker build -t easyfl-server -f docker/server.Dockerfile .
docker build -t easyfl-run -f docker/run.Dockerfile .
```

Start with Docker Compose

Use docker compose to start all services.

```
docker-compose up --scale client=2 && docker-compose rm -fsv
```

Mac users with Docker Desktop > 2.0 may have port conflict occurs because `bind: address already in use`. The workaround is to run with

```
docker-compose up && docker-compose rm -fsv
```

and start another terminal to scale with

```
docker-compose up --scale client=2 && docker-compose rm -fsv
```

Etcd Setup

```
export NODE1=localhost
export DATA_DIR="etcd-data"
REGISTRY=quay.io/coreos/etcd

docker run --rm \
  -p 23790:2379 \
  -p 23800:2380 \
  --volume=${DATA_DIR}:/etcd-data \
  --name etcd ${REGISTRY}:v3.4.0 \
  /usr/local/bin/etcd \
  --data-dir=/etcd-data --name node1 \
  --initial-advertise-peer-urls http://${NODE1}:2380 --listen-peer-urls http://0.0.0.
↪0:2380 \
  --advertise-client-urls http://${NODE1}:2379 --listen-client-urls http://0.0.0.0:2379 \
  --initial-cluster node1=http://${NODE1}:2380
```

Docker Register

```
docker run --name docker-register --rm -d -e HOST_IP=<172.18.0.1> -e ETCD_HOST=<172.17.0.
↪1>:2379 -v /var/run/docker.sock:/var/run/docker.sock -t wingalong/docker-register
```

- HOST_IP: the ip address of network client runs on: gateway in `docker inspect easyfl-client`
- ETCD_HOST: the ip address of etcd: gateway in `docker inspect etcd`

Start containers

```
# 1. Start clients
docker run --rm -p 23400:23400 --name client0 --network host -v <dataset_path>/femnist/
↪data:/app/<dataset_path>/femnist/data easyfl-client --index=0 --is-remote=True --local-
↪port=23400 --server-addr="localhost:23501"
docker run --rm -p 23401:23401 --name client1 --network host -v <dataset_path>/femnist/
↪data:/app/<dataset_path>/femnist/data easyfl-client --index=1 --is-remote=True --local-
↪port=23401 --server-addr="localhost:23501"

# 2. Start server
docker run --rm -p 23501:23501 --name easyfl-server --network host  easyfl-server --
↪local-port=23501 --is-remote=True
```

Note: you need to replace the `dataset_path` with your actual dataset directory.

Start Training Remotely

```
docker run --rm --name easyfl-run --network host easyfl-run --server-addr 127.0.0.
↪1:23501 --etcd-addr:127.0.0.1:23790
```

It sends a gRPC message to server to start training.

15.2.2 Deployment using Kubernetes

```
# 1. Deploy tracker
kubectl apply -f kubernetes/tracker.yml

# 2. Deploy server
kubectl apply -f kubernetes/server.yml

# 3. Deploy client
kubectl apply -f kubernetes/client.yml

# 4. Scale client
kubectl scale -n easyfl deployment easyfl-client --replicas=6

# 5. Check pods
kubectl get pods -n easyfl -o wide

# 6. Run
```

(continues on next page)

(continued from previous page)

```
python examples/remote_run.py --server-addr localhost:32501 --source kubernetes

# 7. Check logs
kubectl logs -f -n easyfl easyfl-server

# 8. Get results
python examples/test_services.py --task-id task_ijhwqg

# 9. Save log
kubectl logs -n easyfl easyfl-server > server-log.log

# 10. Stop client/server/tracker
kubectl delete -f kubernetes/client.yml
kubectl delete -f kubernetes/server.yml
kubectl delete -f kubernetes/tracker.yml
```

CHAPTER

SIXTEEN

PROJECTS BASED ON EASYFL

We have been doing research on federated learning for several years and published several papers in top-tier conferences and journals. EasyFL is developed based on deep insights from our research. It further facilitated us built other federated learning several projects.

16.1 Applications

We have released the following implementations of federated learning applications:

- Federated Multiple Task Learning: [code] for MAS: Towards Resource-Efficient Federated Multiple-Task Learning (*ICCV'2023*)
- FedReID: [code] for Performance Optimization for Federated Person Re-identification via Benchmark Analysis (*ACMMM'2020*).
- FedSSL: [code] for two papers: Divergence-aware Federated Self-Supervised Learning (*ICLR'2022*) and Collaborative Unsupervised Visual Representation Learning From Decentralized Data (*ICCV'2021*)

16.2 Papers

The following are the projects and papers built on EasyFL:

- EasyFL: A Low-code Federated Learning Platform For Dummies, *IEEE Internet-of-Things Journal*. [paper]
- Divergence-aware Federated Self-Supervised Learning, *ICLR'2022*. [paper]
- Collaborative Unsupervised Visual Representation Learning From Decentralized Data, *ICCV'2021*. [paper]
- Joint Optimization in Edge-Cloud Continuum for Federated Unsupervised Person Re-identification, *ACMMM'2021*. [paper]

If you have built new projects using EasyFL, please feel free to create PR to update this page.

CHAPTER

SEVENTEEN

CHANGELOG

17.1 v0.1.0 (05/04/2022)

- Official release

CHAPTER

EIGHTEEN

FREQUENTLY ASKED QUESTIONS

We list some common troubles faced by many users and their corresponding solutions here. Feel free to enrich the list if you find any frequent issues and have ways to help others to solve them. If the contents here do not cover your issue, please create an issue using the provided templates and make sure you fill in all required information in the template.

18.1 EasyFL Installation

Waiting for your input :)

CHAPTER

NINETEEN

EASYFL

CHAPTER

TWENTY

EASYFL.SERVER

CHAPTER

TWENTYONE

EASYFL.CLIENT

class easyfl.client.**BaseClient**(*cid*, *conf*, *train_data*, *test_data*, *device*, *sleep_time=0*, *is_remote=False*, *local_port=23000*, *server_addr='localhost:22999'*, *tracker_addr='localhost:12666'*)

Default implementation of federated learning client.

Parameters

- **cid** (*str*) – Client id.
- **conf** (*omegaconf.dictconfig.DictConfig*) – Client configurations.
- **train_data** (FederatedDataset) – Training dataset.
- **test_data** (FederatedDataset) – Test dataset.
- **device** (*str*) – Hardware device for training, cpu or cuda devices.
- **sleep_time** (*float*) – Duration of on hold after training to simulate stragglers.
- **is_remote** (*bool*) – Whether start remote training.
- **local_port** (*int*) – Port of remote client service.
- **server_addr** (*str*) – Remote server service grpc address.
- **tracker_addr** (*str*) – Remote tracking service grpc address.

Override the class and functions to implement customized client.

Example

```
>>> from easyfl.client import BaseClient
>>> class CustomizedClient(BaseClient):
>>>     def __init__(self, cid, conf, train_data, test_data, device, **kwargs):
>>>         super(CustomizedClient, self).__init__(cid, conf, train_data, test_data,
↪ device, **kwargs)
>>>         pass  # more initialization of attributes.
>>>
>>>     def train(self, conf, device=CPU):
>>>         # Implement customized client training method, which overwrites the␣
↪default training method.
>>>         pass
```

compression()

Compress the client local model after training and before uploading to the server.

connect_to_server()
Establish connection between the client and the server.

construct_upload_request()
Construct client upload request for training updates and testing results.

Returns The upload request defined in protobuf to unify local and remote operations.

Return type UploadRequest

decompression()
Decompressed model. It can be further implemented when the model is compressed in the server.

download(*model*)
Download model from the server.

Parameters **model** (*nn.Module*) – Global model distributed from the server.

encryption()
Encrypt the client local model.

load_loader(*conf*)
Load the training data loader.

Parameters **conf** (*omegaconf.dictconfig.DictConfig*) – Client configurations.

Returns Data loader.

Return type torch.utils.data.DataLoader

load_optimizer(*conf*)
Load training optimizer. Implemented Adam and SGD.

operate(*model*, *conf*, *index*, *is_train=True*)
A wrapper over operations (training/testing) on clients.

Parameters

- **model** (*nn.Module*) – Model for operations.
- **conf** (*omegaconf.dictconfig.DictConfig*) – Client configurations.
- **index** (*int*) – Client index in the client list, for retrieving data. TODO: improvement.
- **is_train** (*bool*) – The flag to indicate whether the operation is training, otherwise testing.

post_test()
Postprocessing after testing.

post_train()
Postprocessing after training.

post_upload()
Postprocessing after uploading training/testing results.

pre_test()
Preprocessing before testing.

pre_train()
Preprocessing before training.

pretrain_setup(*conf*, *device*)
Setup loss function and optimizer before training.

run_test(*model*, *conf*)
Conduct testing on clients.

Parameters

- **model** (*nn.Module*) – Model to test.
- **conf** (*omegaconf.dictconfig.DictConfig*) – Client configurations.

Returns Testing contents. Unify the interface for both local and remote operations.

Return type UploadRequest

run_train(*model*, *conf*)
Conduct training on clients.

Parameters

- **model** (*nn.Module*) – Model to train.
- **conf** (*omegaconf.dictconfig.DictConfig*) – Client configurations.

Returns Training contents. Unify the interface for both local and remote operations.

Return type UploadRequest

save_metrics()
Save client metrics to database.

simulate_straggler()
Simulate straggler effect of system heterogeneity.

start_service()
Start client service.

test(*conf*, *device='cpu'*)
Execute client testing.

Parameters

- **conf** (*omegaconf.dictconfig.DictConfig*) – Client configurations.
- **device** (*str*) – Hardware device for training, cpu or cuda devices.

test_local()
Test client local model after training.

track(*metric_name*, *value*)
Track a metric.

Parameters

- **metric_name** (*str*) – The name of the metric.
- **value** (*str|int|float|bool|dict|list*) – The value of the metric.

train(*conf*, *device='cpu'*)
Execute client training.

Parameters

- **conf** (*omegaconf.dictconfig.DictConfig*) – Client configurations.
- **device** (*str*) – Hardware device for training, cpu or cuda devices.

upload()
Upload the messages from client to the server.

Returns

The upload request defined in protobuf to unify local and remote operations. Only applicable for local training as remote training upload through a gRPC request.

Return type UploadRequest

upload_remotely(*request*)
Send upload request to remote server via gRPC.

Parameters **request** (UploadRequest) – Upload request.

class easyfl.client.**ClientService**(*client*)
"Remote gRPC client service.

Parameters **client** (*BaseClient*) – Federated learning client instance.

Operate(*request*, *context*)
Perform training/testing operations.

CHAPTER

TWENTYTWO

EASYFL.DISTRIBUTED

easyfl.distributed.**dist_init**(*backend*, *init_method*, *world_size*, *rank*, *local_rank*)
Initialize PyTorch distribute.

Parameters

- **backend** (*str or Backend*) – Distributed backend to use, e.g., *nccl*, *gloo*.
- **init_method** (*str, optional*) – URL specifying how to initialize the process group.
- **world_size** (*int, optional*) – Number of processes participating in the job.
- **rank** (*local*) – Rank of the current process.
- **rank** – Local rank of the current process.

Returns Rank of current process. int: Total number of processes.

Return type int

easyfl.distributed.**gather_value**(*value*, *world_size*, *device*)
Gather the value from devices to a list.

Parameters

- **value** (*float|int*) – The value to gather.
- **world_size** (*int*) – The number of processes.
- **device** (*str*) – The device where the value is on, either cpu or cuda devices.

Returns A list of gathered values.

Return type list[torch.Tensor]

easyfl.distributed.**get_device**(*gpu*, *world_size*, *local_rank*)
Obtain the device by checking the number of GPUs and distributed settings.

Parameters

- **gpu** (*int*) – The number of requested gpu.
- **world_size** (*int*) – The number of processes.
- **local_rank** (*int*) – The local rank of the current process.

Returns Device to be used in PyTorch like *tensor.to(device)*.

Return type str

easyfl.distributed.**get_ip**(*node_list*)
Get the ip address of nodes.

Parameters **node_list** (*str*) – Name of the nodes.

Returns The first node in the nodes.

Return type str

easyfl.distributed.**grouping**(*clients*, *world_size*, *default_time=10*, *strategy='random'*, *seed=1*)
Divide clients into groups with different strategies.

Parameters

- **clients** (list[BaseClient]) – A list of clients.
- **world_size** (*int*) – The number of processes, it represent the number of groups here.
- **default_time** (*float, optional*) – The default training time for not profiled clients.
- **strategy** (*str, optional*) – Strategy of grouping, options: random, greedy, worst. When no strategy is applied, each client is a group.
- **seed** (*int, optional*) – Random seed.

Returns Groups of clients, each group is a sub-list.

Return type list[list[BaseClient]]

easyfl.distributed.**reduce_models**(*model*, *sample_sum*)
Aggregate models across devices and update the model with the new aggregated model parameters.

Parameters

- **model** (*nn.Module*) – The model in a device to aggregate.
- **sample_sum** (*int*) – Sum of the total dataset sizes of clients in a device.

easyfl.distributed.**reduce_models_only_params**(*model*, *sample_sum*)
Aggregate models across devices and update the model with the new aggregated model parameters, excluding the persistent buffers like BN stats.

Parameters

- **model** (*nn.Module*) – The model in a device to aggregate.
- **sample_sum** (*torch.Tensor*) – Sum of the total dataset sizes of clients in a device.

easyfl.distributed.**reduce_value**(*value*, *device*)
Calculate the sum of the value across devices.

Parameters

- **value** (*float/int*) – Value to sum.
- **device** (*str*) – The device where the value is on, either cpu or cuda devices.

Returns Sum of the values.

Return type torch.Tensor

easyfl.distributed.**reduce_values**(*values*, *device*)
Calculate the average of values across devices.

Parameters

- **values** (*list[float|int]*) – Values to average.
- **device** (*str*) – The device where the value is on, either cpu or cuda devices.

Returns The average of the values across devices.

Return type torch.Tensor

easyfl.distributed.**reduce_weighted_values**(*values*, *weights*, *device*)
Calculate the weighted average of values across devices.

Parameters

- **values** (*list[float|int]*) – Values to average.
- **weights** (*list[float|int]*) – The weights to calculate weighted average.
- **device** (*str*) – The device where the value is on, either cpu or cuda devices.

Returns The average of values across devices.

Return type torch.Tensor

easyfl.distributed.**setup**(*port=23344*)
Setup distributed settings of slurm.

Parameters **port** (*int, optional*) – The port of the primary server. It respectively auto-increments by 1 when the port is in-use.

Returns The rank of current process. int: The local rank of current process. int: Total number of processes. str: The address of the distributed init method.

Return type int

CHAPTER

TWENTYTHREE

EASYFL.DATASET

class easyfl.datasets.**BaseDataset**(*root*, *dataset_name*, *fraction*, *split_type*, *user*, *iid_user_fraction*, *train_test_split*, *minsample*, *num_class*, *num_of_client*, *class_per_client*, *setting_folder*, *seed=- 1*, ***kwargs*)

The internal base dataset implementation.

Parameters

- **root** (*str*) – The root directory where datasets stored.
- **dataset_name** (*str*) – The name of the dataset.
- **fraction** (*float*) – The fraction of the data chosen from the raw data to use.
- **num_of_clients** (*int*) – The targeted number of clients to construct.
- **split_type** (*str*) – The type of statistical simulation, options: iid, dir, and class. *iid* means independent and identically distributed data. *niid* means non-independent and identically distributed data for Femnist and Shakespeare. *dir* means using Dirichlet process to simulate non-iid data, for CIFAR-10 and CIFAR-100 datasets. *class* means partitioning the dataset by label classes, for datasets like CIFAR-10, CIFAR-100.
- **minsample** (*int*) – The minimal number of samples in each client. It is applicable for LEAF datasets and dir simulation of CIFAR-10 and CIFAR-100.
- **class_per_client** (*int*) – The number of classes in each client. Only applicable when the split_type is 'class'.
- **iid_user_fraction** (*float*) – The fraction of the number of clients used when the split_type is 'iid'.
- **user** (*bool*) – A flag to indicate whether partition users of the dataset into train-test groups. Only applicable to LEAF datasets. True means partitioning users of the dataset into train-test groups. False means partitioning each users' samples into train-test groups.
- **train_test_split** (*float*) – The fraction of data for training; the rest are for testing. e.g., 0.9 means 90% of data are used for training and 10% are used for testing.
- **num_class** – The number of classes in this dataset.
- **seed** – Random seed.

class easyfl.datasets.**Cifar10**(*root*, *fraction*, *split_type*, *user*, *iid_user_fraction=0.1*, *train_test_split=0.9*, *minsample=10*, *num_class=80*, *num_of_client=100*, *class_per_client=2*, *setting_folder=None*, *seed=- 1*, *weights=None*, *alpha=0.5*)

class easyfl.datasets.**Cifar100**(*root*, *fraction*, *split_type*, *user*, *iid_user_fraction=0.1*, *train_test_split=0.9*, *minsample=10*, *num_class=80*, *num_of_client=100*, *class_per_client=2*, *setting_folder=None*, *seed=- 1*, *weights=None*, *alpha=0.5*)

class easyfl.datasets.**FederatedDataset**

The abstract class of federated dataset for EasyFL.

abstract loader(*batch_size*, *shuffle=True*)

Get data loader.

Parameters

- **batch_size** (*int*) – The batch size of the data loader.
- **shuffle** (*bool*) – Whether shuffle the data in the loader.

abstract size(*cid*)

Get dataset size.

Parameters **cid** (*str*) – client id.

property users

Get client ids of the federated dataset.

class easyfl.datasets.**FederatedImageDataset**(*root*, *simulated*, *do_simulate=True*, *extensions=('.jpg', '.jpeg', '.png', '.ppm', '.bmp', '.pgm', '.tif', '.tiff', '.webp')*, *is_valid_file=None*, *transform=None*, *target_transform=None*, *client_ids='default'*, *num_of_clients=10*, *simulation_method='iid'*, *weights=None*, *alpha=0.5*, *min_size=10*, *class_per_client=1*)

Federated image dataset, data of clients are in format of image folder.

Parameters

- **root** (*str|list[str]*) – The root directory or directories of image data folder. If the dataset is simulated to multiple clients, the root is a list of directories. Otherwise, it is the directory of an image data folder.
- **simulated** (*bool*) – Whether the dataset is simulated to federated learning settings.
- **do_simulate** (*bool, optional*) – Whether conduct simulation. It is only effective if it is not simulated.
- **extensions** (*list[str], optional*) – A list of allowed image extensions. Only one of *extensions* and *is_valid_file* can be specified.
- **is_valid_file** (*function, optional*) – A function that takes path of an Image file and check if it is valid. Only one of *extensions* and *is_valid_file* can be specified.
- **transform** (*torchvision.transforms.transforms.Compose, optional*) – Transformation for data.
- **target_transform** (*torchvision.transforms.transforms.Compose, optional*) – Transformation for data labels.
- **num_of_clients** (*int, optional*) – number of clients for simulation. Only need if doing simulation.
- **simulation_method** (*optional*) – split method. Only need if doing simulation.
- **weights** (*list[float], optional*) – The targeted distribution of quantities to simulate quantity heterogeneity. The values should sum up to 1. e.g., [0.1, 0.2, 0.7]. The *num_of_clients* should be divisible by *len(weights)*. None means clients are simulated with the same data quantity.
- **alpha** (*float, optional*) – The parameter for Dirichlet distribution simulation, only for dir simulation.

- **min_size** (*int, optional*) – The minimal number of samples in each client, only for dir simulation.
- **class_per_client** (*int, optional*) – The number of classes in each client, only for non-iid by class simulation.
- **client_ids** (*list[str], optional*) – A list of client ids. Each client id matches with an element in roots. The client ids are ["f0000001", "f00000002", ...] if not specified.

loader(*batch_size*, *client_id=None*, *shuffle=True*, *seed=0*, *num_workers=2*, *transform=None*)

Get dataset loader.

Parameters

- **batch_size** (*int*) – The batch size.
- **client_id** (*str, optional*) – The id of client.
- **shuffle** (*bool, optional*) – Whether to shuffle before batching.
- **seed** (*int, optional*) – The shuffle seed.
- **transform** (*torchvision.transforms.transforms.Compose, optional*) – Data transformation.
- **num_workers** (*int, optional*) – The number of workers for dataset loader.

Returns The data loader to load data.

Return type torch.utils.data.DataLoader

size(*cid=None*)

Get dataset size.

Parameters **cid** (*str*) – client id.

property users

Get client ids of the federated dataset.

class easyfl.datasets.**FederatedTensorDataset**(*data*, *transform=None*, *target_transform=None*, *process_x=<function default_process_x>*, *process_y=<function default_process_x>*, *simulated=False*, *do_simulate=True*, *num_of_clients=10*, *simulation_method='iid'*, *weights=None*, *alpha=0.5*, *min_size=10*, *class_per_client=1*)

Federated tensor dataset, data of clients are in format of tensor or list.

Parameters

- **data** (*dict*) – A dictionary of data, e.g., {"id1": {"x": [[], [], ...], "y": [...]]}}. If simulation is not done previously, it is in format of {'x':[[],[], ...], 'y': [...]}.
- **transform** (*torchvision.transforms.transforms.Compose, optional*) – Transformation for data.
- **target_transform** (*torchvision.transforms.transforms.Compose, optional*) – Transformation for data labels.
- **process_x** (*function, optional*) – A function to preprocess training data.
- **process_y** (*function, optional*) – A function to preprocess testing data.
- **simulated** (*bool, optional*) – Whether the dataset is simulated to federated learning settings.

- **do_simulate** (*bool, optional*) – Whether conduct simulation. It is only effective if it is not simulated.
- **num_of_clients** (*int, optional*) – number of clients for simulation. Only need if doing simulation.
- **simulation_method** (*optional*) – split method. Only need if doing simulation.
- **weights** (*list[float], optional*) – The targeted distribution of quantities to simulate quantity heterogeneity. The values should sum up to 1. e.g., [0.1, 0.2, 0.7]. The *num_of_clients* should be divisible by *len(weights)*. None means clients are simulated with the same data quantity.
- **alpha** (*float, optional*) – The parameter for Dirichlet distribution simulation, only for dir simulation.
- **min_size** (*int, optional*) – The minimal number of samples in each client, only for dir simulation.
- **class_per_client** (*int, optional*) – The number of classes in each client, only for non-iid by class simulation.

loader(*batch_size*, *client_id=None*, *shuffle=True*, *seed=0*, *transform=None*, *drop_last=False*)
Get dataset loader.

Parameters

- **batch_size** (*int*) – The batch size.
- **client_id** (*str, optional*) – The id of client.
- **shuffle** (*bool, optional*) – Whether to shuffle before batching.
- **seed** (*int, optional*) – The shuffle seed.
- **transform** (*torchvision.transforms.transforms.Compose, optional*) – Data transformation.
- **drop_last** (*bool, optional*) – Whether to drop the last batch if its size is smaller than batch size.

Returns The data loader to load data.

Return type torch.utils.data.DataLoader

size(*cid=None*)
Get dataset size.

Parameters **cid** (*str*) – client id.

property users
Get client ids of the federated dataset.

class easyfl.datasets.**FederatedTorchDataset**(*data*, *users*)
Wrapper over PyTorch dataset.

Parameters **data** (*dict*) – A dictionary of client datasets, format {"client_id": dataset1, "client_id2": dataset2}.

loader(*batch_size*, *client_id=None*, *shuffle=True*, *seed=0*, *num_workers=2*, *transform=None*)
Get data loader.

Parameters

- **batch_size** (*int*) – The batch size of the data loader.

- **shuffle** (*bool*) – Whether shuffle the data in the loader.

size(*cid=None*)
Get dataset size.

Parameters **cid** (*str*) – client id.

property users
Get client ids of the federated dataset.

class easyfl.datasets.**Femnist**(*root*, *fraction*, *split_type*, *user*, *iid_user_fraction=0.1*, *train_test_split=0.9*, *minsample=10*, *num_class=62*, *num_of_client=100*, *class_per_client=2*, *setting_folder=None*, *seed=- 1*, ***kwargs*)

FEMNIST dataset implementation. It gets FEMNIST dataset according to configurations. It stores the processed datasets locally.

base_folder
The base folder path of the datasets folder.

Type str

class_url
The url to get the by_class split FEMNIST.

Type str

write_url
The url to get the by_write split FEMNIST.

Type str

class easyfl.datasets.**Shakespeare**(*root*, *fraction*, *split_type*, *user*, *iid_user_fraction=0.1*, *train_test_split=0.9*, *minsample=10*, *num_class=80*, *num_of_client=100*, *class_per_client=2*, *setting_folder=None*, *seed=- 1*, ***kwargs*)
Shakespeare dataset implementation. It gets Shakespeare dataset according to configurations.

base_folder
The base folder path of the datasets folder.

Type str

raw_data_url
The url to get the *by_class* split shakespeare.

Type str

write_url
The url to get the *by_write* split shakespeare.

Type str

easyfl.datasets.**construct_datasets**(*root*, *dataset_name*, *num_of_clients*, *split_type*, *min_size*, *class_per_client*, *data_amount*, *iid_fraction*, *user*, *train_test_split*, *quantity_weights*, *alpha*)
Construct and load provided federated learning datasets.

Parameters

- **root** (*str*) – The root directory where datasets stored.

- **dataset_name** (*str*) – The name of the dataset. It currently supports: femnist, shakespeare, cifar10, and cifar100. Among them, femnist and shakespeare are adopted from LEAF benchmark.
- **num_of_clients** (*int*) – The targeted number of clients to construct.
- **split_type** (*str*) – The type of statistical simulation, options: iid, dir, and class. *iid* means independent and identically distributed data. *niid* means non-independent and identically distributed data for Femnist and Shakespeare. *dir* means using Dirichlet process to simulate non-iid data, for CIFAR-10 and CIFAR-100 datasets. *class* means partitioning the dataset by label classes, for datasets like CIFAR-10, CIFAR-100.
- **min_size** (*int*) – The minimal number of samples in each client. It is applicable for LEAF datasets and dir simulation of CIFAR-10 and CIFAR-100.
- **class_per_client** (*int*) – The number of classes in each client. Only applicable when the split_type is 'class'.
- **data_amount** (*float*) – The fraction of data sampled for LEAF datasets. e.g., 10% means that only 10% of total dataset size are used.
- **iid_fraction** (*float*) – The fraction of the number of clients used when the split_type is 'iid'.
- **user** (*bool*) – A flag to indicate whether partition users of the dataset into train-test groups. Only applicable to LEAF datasets. True means partitioning users of the dataset into train-test groups. False means partitioning each users' samples into train-test groups.
- **train_test_split** (*float*) – The fraction of data for training; the rest are for testing. e.g., 0.9 means 90% of data are used for training and 10% are used for testing.
- **quantity_weights** (*list[float]*) – The targeted distribution of quantities to simulate data quantity heterogeneity. The values should sum up to 1. e.g., [0.1, 0.2, 0.7]. The *num_of_clients* should be divisible by *len(weights)*. None means clients are simulated with the same data quantity.
- **alpha** (*float*) – The parameter for Dirichlet distribution simulation, applicable only when split_type is *dir*.

Returns Training dataset. *FederatedDataset*: Testing dataset.

Return type *FederatedDataset*

easyfl.datasets.**data_simulation**(*data_x*, *data_y*, *num_of_clients*, *data_distribution*, *weights=None*, *alpha=0.5*, *min_size=10*, *class_per_client=1*, *stack_x=True*)

Simulate federated learning datasets by partitioning a data into multiple clients using different strategies.

Parameters

- **data_x** (*list[Object]*) – A list of data.
- **data_y** (*list[Object]*) – A list of dataset labels.
- **num_of_clients** (*int*) – The number of clients to partition to.
- **data_distribution** (*str*) – The ways to partition the dataset, options: *iid*: Partition dataset into multiple clients with equal quantity (difference is less than 1) randomly. *dir*: partition dataset into multiple clients following the Dirichlet process. *class*: partition dataset into multiple clients based on classes.
- **weights** (*list[float], optional*) – list, for simulating data quantity heterogeneity If None, each client are simulated with same data quantity Note: num_of_clients should be divisible by len(weights)

- **weights** – The targeted distribution of data quantities. The values should sum up to 1. e.g., [0.1, 0.2, 0.7]. When *weights=None*, the data quantity of clients only depends on data_distribution.
- **alpha** (*float, optional*) – The parameter for Dirichlet process simulation. It is only applicable when data_distribution is *dir*.
- **min_size** (*int, optional*) – The minimum number of data size of a client. It is only applicable when data_distribution is *dir*.
- **class_per_client** (*int*) – The number of classes in each client. It is only applicable when data_distribution is *class*.
- **stack_x** (*bool, optional*) – A flag to indicate whether using np.vstack or append to construct dataset. It is only applicable when data_distribution is *class*.

Raises **ValueError** – When the simulation method *data_distribution* is not supported.

Returns A list of client ids. dict: The partitioned data, key is client id, value is the client data. e.g., {'client_1': {'x': [data_x], 'y': [data_y]}}.

Return type list[str]

easyfl.datasets.**equal_division**(*num_groups*, *data_x*, *data_y=None*)

Partition data into multiple clients with equal quantity.

Parameters

- **num_groups** (*int*) – THe number of groups to partition to.
- **data_x** (*list[Object]*) – A list of elements to be divided.
- **data_y** (*list[Object], optional*) – A list of data labels to be divided together with the data.

Returns A list where each element is a list of data of a group/client. list[list]: A list where each element is a list of data label of a group/client.

Return type list[list]

Example

```
>>> equal_division(3, list[range(9)])
>>> ([[0,4,2],[3,1,7],[6,5,8]], [])
```

easyfl.datasets.**iid**(*data_x*, *data_y*, *num_of_clients*, *x_dtype*, *y_dtype*)

Partition dataset into multiple clients with equal data quantity (difference is less than 1) randomly.

Parameters

- **data_x** (*list[Object]*) – A list of data.
- **data_y** (*list[Object]*) – A list of dataset labels.
- **num_of_clients** (*int*) – The number of clients to partition to.
- **x_dtype** (*numpy.dtype*) – The type of data.
- **y_dtype** (*numpy.dtype*) – The type of data label.

Returns A list of client ids. dict: The partitioned data, key is client id, value is the client data. e.g., {'client_1': {'x': [data_x], 'y': [data_y]}}.

Return type list[str]

easyfl.datasets.**non_iid_class**(*data_x*, *data_y*, *class_per_client*, *num_of_clients*, *x_dtype*, *y_dtype*, *stack_x=True*)

Partition dataset into multiple clients based on label classes. Each client contains [1, n] classes, where n is the number of classes of a dataset.

Note: Each class is divided into *ceil(class_per_client * num_of_clients / num_class)* parts and each client chooses *class_per_client* parts from each class to construct its dataset.

Parameters

- **data_x** (*list[Object]*) – A list of data.
- **data_y** (*list[Object]*) – A list of dataset labels.
- **class_per_client** (*int*) – The number of classes in each client.
- **num_of_clients** (*int*) – The number of clients to partition to.
- **x_dtype** (*numpy.dtype*) – The type of data.
- **y_dtype** (*numpy.dtype*) – The type of data label.
- **stack_x** (*bool, optional*) – A flag to indicate whether using np.vstack or append to construct dataset.

Returns A list of client ids. dict: The partitioned data, key is client id, value is the client data. e.g., {'client_1': {'x': [data_x], 'y': [data_y]}}.

Return type list[str]

easyfl.datasets.**non_iid_dirichlet**(*data_x*, *data_y*, *num_of_clients*, *alpha*, *min_size*, *x_dtype*, *y_dtype*)

Partition dataset into multiple clients following the Dirichlet process.

Parameters

- **data_x** (*list[Object]*) – A list of data.
- **data_y** (*list[Object]*) – A list of dataset labels.
- **num_of_clients** (*int*) – The number of clients to partition to.
- **alpha** (*float*) – The parameter for Dirichlet process simulation.
- **min_size** (*int*) – The minimum number of data size of a client.
- **x_dtype** (*numpy.dtype*) – The type of data.
- **y_dtype** (*numpy.dtype*) – The type of data label.

Returns A list of client ids. dict: The partitioned data, key is client id, value is the client data. e.g., {'client_1': {'x': [data_x], 'y': [data_y]}}.

Return type list[str]

easyfl.datasets.**quantity_hetero**(*weights*, *data_x*, *data_y=None*)

Partition data into multiple clients with different quantities. The number of groups is the same as the number of elements of *weights*. The quantity of each group depends on the values of *weights*.

Parameters

- **weights** (*list[float]*) – The targeted distribution of data quantities. The values should sum up to 1. e.g., [0.1, 0.2, 0.7].
- **data_x** (*list[Object]*) – A list of elements to be divided.

- **data_y** (*list[Object], optional*) – A list of data labels to be divided together with the data.

Returns A list where each element is a list of data of a group/client. list[list]: A list where each element is a list of data label of a group/client.

Return type list[list]

Example

```
>>> quantity_hetero([0.1, 0.2, 0.7], list(range(0, 10)))
>>> ([[4], [8, 9], [6, 0, 1, 7, 3, 2, 5]], [])
```

CHAPTER

TWENTYFOUR

EASYFL.MODELS

CHAPTER

TWENTYFIVE

EASYFL.COMMUNICATION

easyfl.communication.**init_stub**(*typ*, *address*)

Initialize gRPC stub.

Parameters

- **typ** (*str*) – Type of service, option: client, server, tracking
- **address** (*str*) – Address of the gRPC service.

Returns stub of the gRPC service.

Return type (ClientServiceStub`|:obj:`ServerServiceStub`|:obj:`TrackingServiceStub)

easyfl.communication.**start_service**(*typ*, *service*, *port*)

Start gRPC service. :param typ: Type of service, option: client, server, tracking. :type typ: str :param service: gRPC service to start. :type service: ClientService`|:obj:`ServerService`|:obj:`TrackingService :param port: The port of the service. :type port: int

CHAPTER

TWENTYSIX

EASYFL.REGISTRY

CHAPTER

TWENTYSEVEN

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

e

INDEX

P

Q

R

S

T

U

W