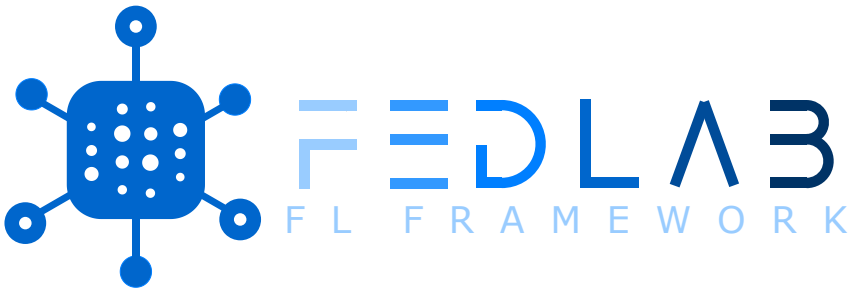
FedLab

Release 1.2.1

SMILE Lab

Apr 25, 2022

CONTENTS:



FedLab provides the necessary modules for FL simulation, including communication, compression, model optimization, data partition and other functional modules. Users can build FL simulation environment with custom modules like playing with LEGO bricks.

CHAPTER

ONE

INTRODUCTION

Federated learning (FL), proposed by Google at the very beginning, is recently a burgeoning research area of machine learning, which aims to protect individual data privacy in distributed machine learning process, especially in finance, smart healthcare and edge computing. Different from traditional data-centered distributed machine learning, participants in FL setting utilize localized data to train local model, then leverages specific strategies with other participants to acquire the final model collaboratively, avoiding direct data sharing behavior.

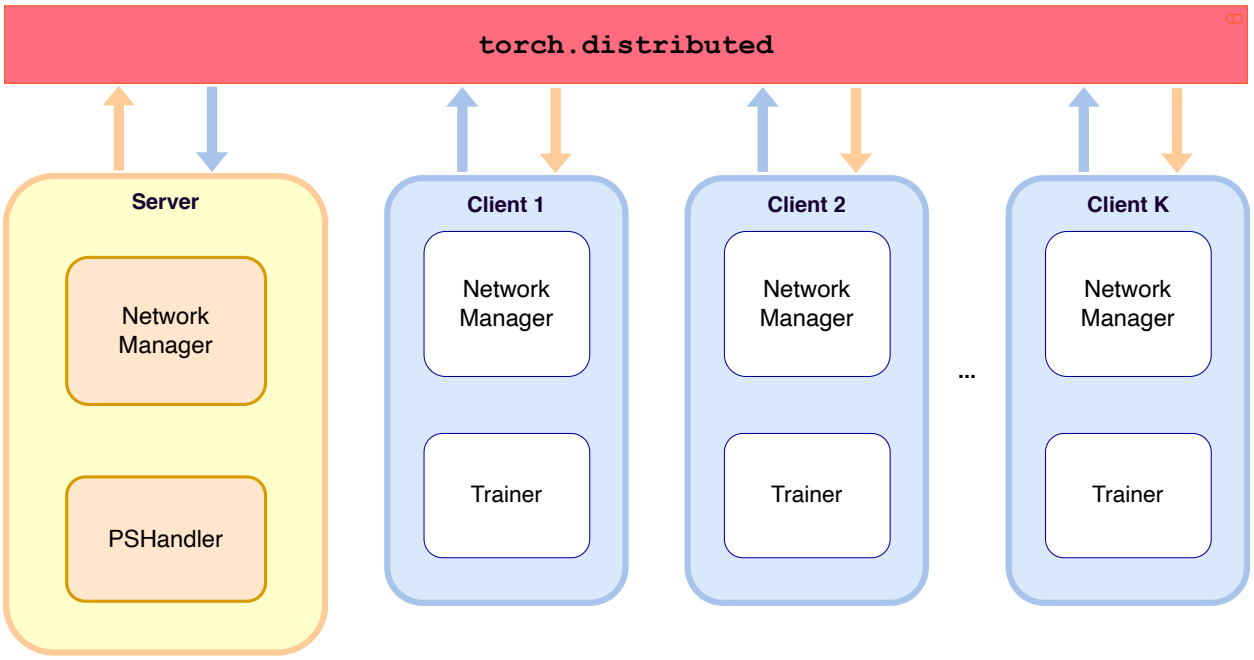
To relieve the burden of researchers in implementing FL algorithms and emancipate FL scientists from repetitive implementation of basic FL setting, we introduce highly customizable framework **FedLab** in this work. **FedLab** is builded on the top of torch.distributed modules and provides the necessary modules for FL simulation, including communication, compression, model optimization, data partition and other functional modules. **FedLab** users can build FL simulation environment with custom modules like playing with LEGO bricks. For better understanding and easy usage, FL algorithm benchmark implemented in **FedLab** are also presented.

For more details, please read our full paper.

CHAPTER

TWO

OVERVIEW

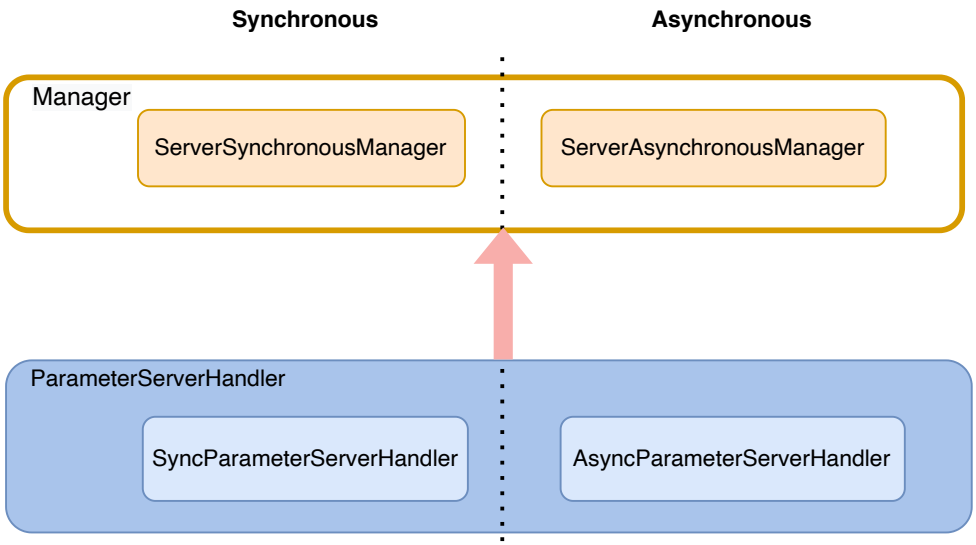


FedLab provides two basic roles in FL setting: `Server` and `Client`. Each `Server`/`Client` consists of two components called `NetworkManager` and `ParameterHandler`/`Trainer`.

- `NetworkManager` module manages message process task, which provides interfaces to customize communication agreements and compression.
- `ParameterHandler` is responsible for backend computation in `Server`; and `Trainer` is in charge of backend computation in `Client`.

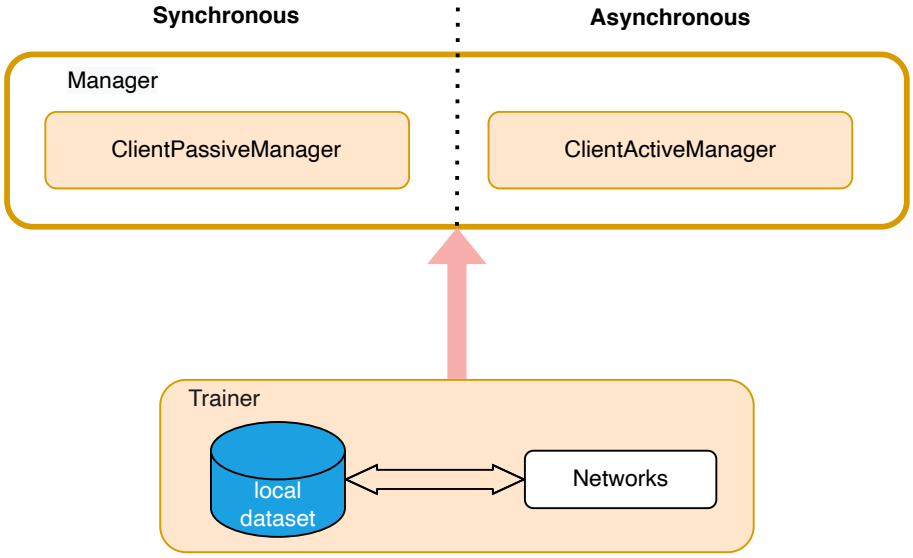
2.1 Server

The connection between `NetworkManager` and `ParameterServerHandler` in `Server` is shown as below. `NetworkManager` processes message and calls `ParameterServerHandler.on_receive()` method, while `ParameterServerHandler` performs training as well as computation process of server (model aggregation for example), and updates the global model.



2.2 Client

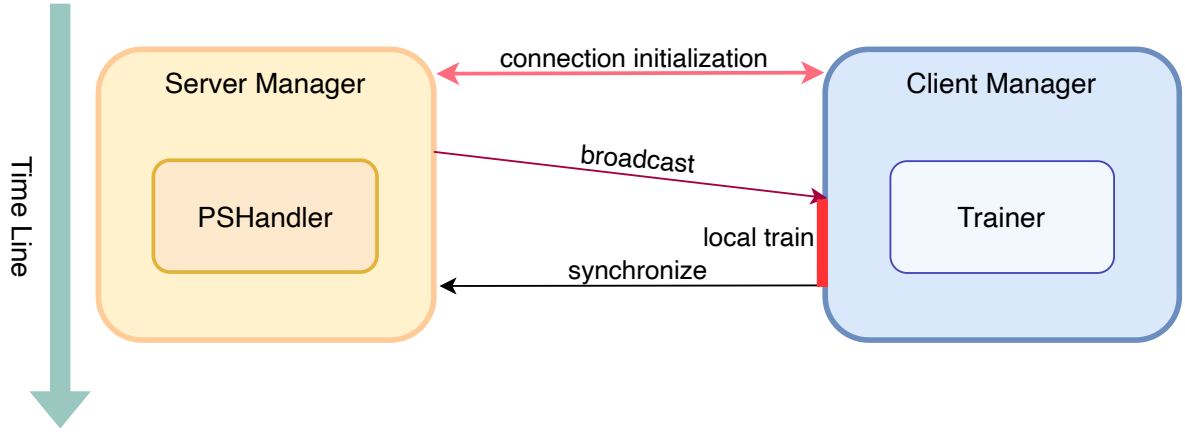
`Client` shares similar design and structure with `Server`, with `NetworkManager` in charge of message processing as well as network communication with server, and *Trainer* for client local training procedure.



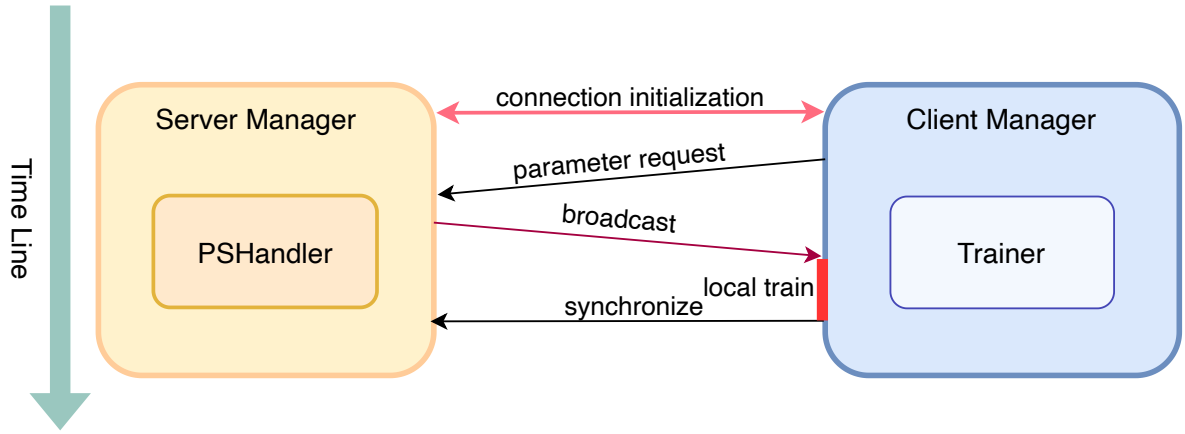
2.3 Communication

FedLab furnishes both synchronous and asynchronous communication patterns, and their corresponding communication logics of `NetworkManager` is shown as below.

1. Synchronous FL: each round is launched by server, that is, server performs clients sampling first then broadcasts global model parameters.



2. Asynchronous FL [1]: each round is launched by clients, that is, clients request current global model parameters then perform local training.



CHAPTER

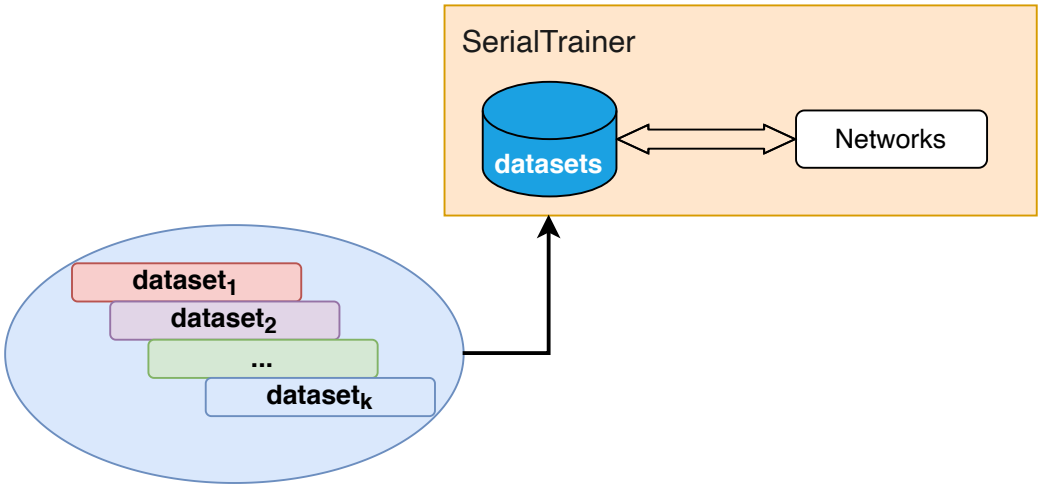
THREE

EXPERIMENTAL SCENE

FedLab supports both single machine and multi-machine FL simulations, with **standalone** mode for single machine experiments, while cross-machine mode and **hierarchical** mode for multi-machine experiments.

3.1 Standalone

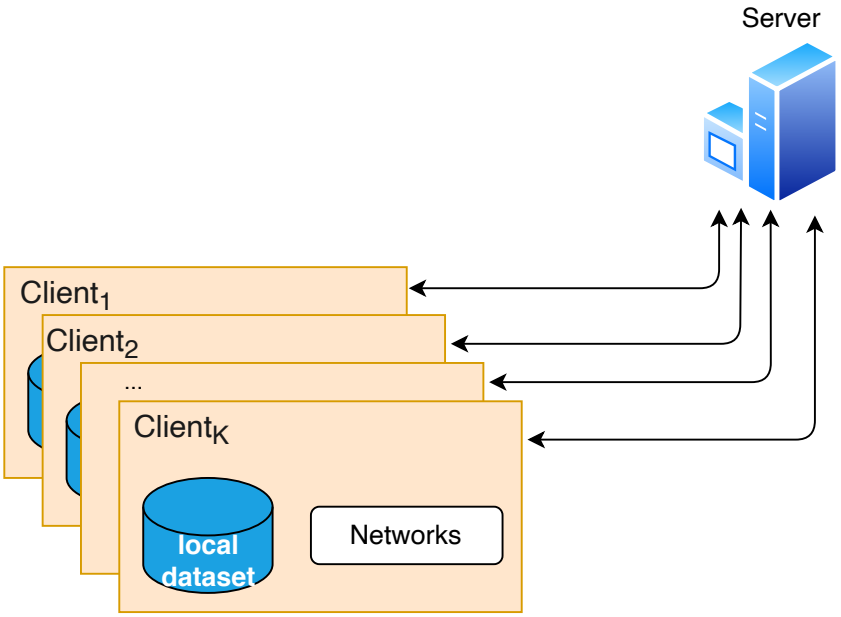
FedLab implements `SerialTrainer` for FL simulation in single system process. `SerialTrainer` allows user to simulate a FL system with multiple clients executing one by one in serial in one `SerialTrainer`. It is designed for simulation in environment with limited computation resources.



3.2 Cross-process

FedLab enables FL simulation tasks to be deployed on multiple processes with correct network configuration (these processes can be run on single or multiple machines). More flexibly in parallel, `SerialTrainer` can replace the regular `Trainer` directly. Users can balance the calculation burden among processes by choosing different `Trainer`. In practice, machines with more computation resources can be assigned with more workload of calculation.

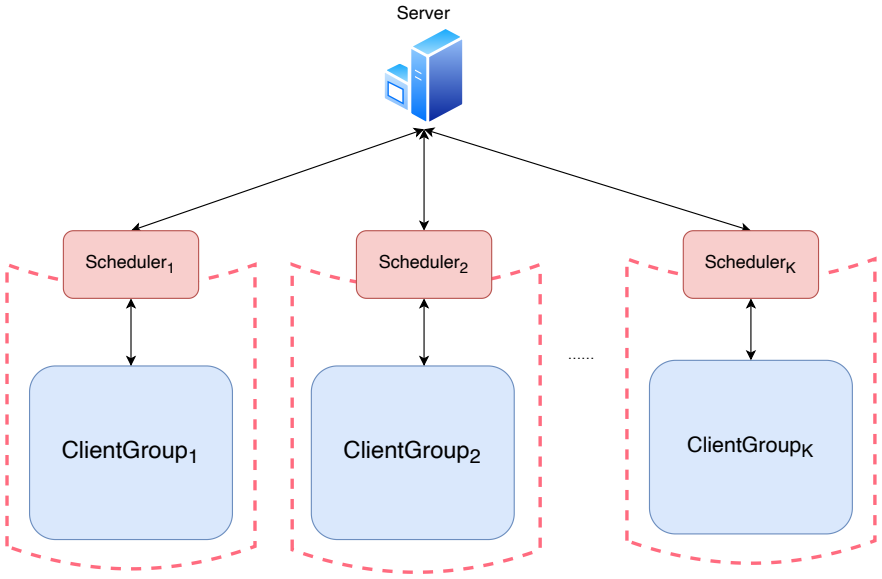
Note: All machines must be in the same network (LAN or WAN) for cross-process deployment.



3.3 Hierarchical

Hierarchical mode for **FedLab** is designed for situation tasks on multiple computer clusters (in different LAN) or the real-world scenes. To enable the inter-connection for different computer clusters, **FedLab** develops `Scheduler` as middle-server process to connect client groups. Each `Scheduler` manages the communication between the global server and clients in a client group. And server can communicate with clients in different LAN via corresponding `Scheduler`. The computation mode of a client group for each scheduler can be either **standalone** or **cross-process**.

A hierarchical FL system with `K` client groups is depicted as below.



CHAPTER

FOUR

BENCHMARKS

FedLab also contains data partition settings [2], and implementations of FL algorithms [3]. For more information please see our FedLab-benchmarks repo. More benchmarks and FL algorithms demos are coming. We current provide reproduction of following algorithms and settings:

Optimization Algorithms:

- FedAvg: Communication-Efficient Learning of Deep Networks from Decentralized Data
- FedAsync: Asynchronous Federated Optimization
- FedProx: Federated Optimization in Heterogeneous Networks
- FedDyn: Federated Learning based on Dynamic Regularization
- Personalized-FedAvg: Improving Federated Learning Personalization via Model Agnostic Meta Learning
- qFFL: Fair Resource Allocation in Federated Learning

Compression Algorithms:

- DGC: Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training
- QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding

Datasets:

- LEAF: A Benchmark for Federated Settings
- NIID-Bench: Federated Learning on Non-IID Data Silos: An Experimental Study

CHAPTER

FIVE

INSTALLATION & SET UP

FedLab can be installed by source code or pip.

5.1 Source Code

Install **lastest version** from GitHub:

```
$ git clone git@github.com:SMILELab-FL/FedLab.git
$ cd FedLab
```

Install dependencies:

```
$ pip install -r requirements.txt
```

5.2 Pip

Install **stable version** with pip:

```
$ pip install fedlab==$version$
```

5.3 Dataset Download

FedLab provides common dataset used in FL researches.

Download procedure scripts are available in fedlab_benchmarks/datasets. For details of dataset, please follow README.md.

CHAPTER

SIX

TUTORIALS

FedLab standardizes FL simulation procedure, including synchronous algorithm, asynchronous algorithm [1] and communication compression [4]. **FedLab** provides modular tools and standard implementations to simplify FL research.

6.1 Distributed Communication

6.1.1 Initialize distributed network

FedLab uses torch.distributed as point-to-point communication tools. The communication backend is Gloo as default. FedLab processes send/receive data through TCP network connection. Here is the details of how to initialize the distributed network.

You need to assign right ethernet to `DistNetwork`, making sure `torch.distributed` network initialization works. `DistNetwork` is for quickly network configuration, which you can create one as follows:

```
from fedlab.core.network import DistNetwork
world_size = 10
rank = 0  # 0 for server, other rank for clients
ethernet = None
server_ip = '127.0.0.1'
server_port = 1234
network = DistNetwork(address=(server_ip, server_port), world_size, rank, ethernet)

network.init_network_connection() # call this method to start connection.
network.close_network_connection() # call this method to shutdown connection.
```

- The `(server_ip, server_port)` is the address of server. please be aware of that the rank of server is 0 as default.
- Make sure `world_size` is the same across process.
- Rank should be different (from `0` to `world_size-1`).
- world_size = 1 (server) + client number.
- The ethernet is None as default. torch.distributed will try finding the right ethernet automatically.
- The `ethernet_name` must be checked (using `ifconfig`). Otherwise, network initialization would fail.

If the automatically detected interface does not work, users are required to assign a right network interface for Gloo, by assigning in code or setting the environment variables `GLOO_SOCKET_IFNAME`, for example `export GLOO_SOCKET_IFNAME=eth0` or `os.environ['GLOO_SOCKET_IFNAME'] = "eth0"`.

Note: Check the available ethernet:

```
$ ifconfig
```

6.1.2 Point-to-point communication

In recent update, we hide the communication details from user and provide simple APIs. `DistNetwork` now provies two basic communication APIs: `send()` and `recv()`. These APIs suppor flexible pytorch tensor communication.

Sender process:

```
network = DistNetwork(address=(server_ip, server_port), world_size, rank, ethernet)
network.init_network_connection()
network.send(content, message_code, dst)
network.close_network_connection()
```

Receiver process:

```
network = DistNetwork(address=(server_ip, server_port), world_size, rank, ethernet)
network.init_network_connection()
sender_rank, message_code, content = network.recv(src)
################################
#                              #
#  local process with content.  #
#                              #
################################
network.close_network_connection()
```

Note:

Currently, following restrictions need to be noticed

1. **Tensor list:** `send()` accepts a python list with tensors.
2. **Data type:** `send()` doesn't accept tensors of different data type. In other words, **FedLab** force all appended tensors to be the same data type as the first appended tensor. Torch data types like **[torch.int8, torch.int16, torch.int32, torch.int64, torch.float16, torch.float32, torch.float64]** are supported.

6.1.3 Further understanding of FedLab communication

FedLab pack content into a pre-defined package data structure. `send()` and `recv()` are implemented like:

```
def send(self, content=None, message_code=None, dst=0):
    """Send tensor to process rank=dst"""
    pack = Package(message_code=message_code, content=content)
    PackageProcessor.send_package(pack, dst=dst)

def recv(self, src=None):
    """Receive tensor from process rank=src"""
    sender_rank, message_code, content = PackageProcessor.recv_package(
```

(continues on next page)

(continued from previous page)

```
        src=src)
    return sender_rank, message_code, content
```

Create package

The basic communication unit in FedLab is called package. The communication module of FedLab is in fedlab/core/communicator. `Package` defines the basic data structure of network package. It contains header and content.

```
p = Package()
p.header   # A tensor with size = (5,).
p.content  # A tensor with size = (x,).
```

Currently, you can create a network package from following methods:

1. initialize with tensor

```
tensor = torch.Tensor(size=(10,))
package = Package(content=tensor)
```

2. initialize with tensor list

```
tensor_sizes = [10, 5, 8]
tensor_list = [torch.rand(size) for size in tensor_sizes]
package = Package(content=tensor_list)
```

3. append a tensor to exist package

```
tensor = torch.Tensor(size=(10,))
package = Package(content=tensor)

new_tensor = torch.Tensor(size=(8,))
package.append_tensor(new_tensor)
```

4. append a tensor list to exist package

```
tensor_sizes = [10, 5, 8]
tensor_list = [torch.rand(size) for size in tensor_sizes]

package = Package()
package.append_tensor_list(tensor_list)
```

Two static methods are provided by Package to parse header and content:

```
p = Package()
Package.parse_header(p.header)  # necessary information to describe the package
Package.parse_content(p.slices, p.content) # tensor list associated with the tensor␣
↪sequence appended into.
```

Send package

The point-to-point communicating agreements is implemented in PackageProcessor module. PackageProcessor is a static class to manage package sending/receiving procedure.

User can send a package to a process with rank=0 (the parameter dst must be assigned):

```
p = Package()
PackageProcessor.send_package(package=p, dst=0)
```

or, receive a package from rank=0 (set the parameter src=None to receive package from any other process):

```
sender_rank, message_code, content = PackageProcessor.recv_package(src=0)
```

6.2 Communication Strategy

Communication strategy is implemented by (ClientManager,ServerManager) pair collaboratively.

The prototype of `NetworkManager` is defined in `fedlab.core.network_manager`, which is also a subclass of `torch.multiprocessing.process`.

Typically, standard implementations is shown in `fedlab.core.client.manager` and `fedlab.core.server.manager`. `NetworkManager` manages network operation and control flow procedure.

Base class definition shows below:

```
class NetworkManager(Process):
    """Abstract class

    Args:
        newtork (DistNetwork): object to manage torch.distributed network communication.
    """

    def __init__(self, network):
        super(NetworkManager, self).__init__()
        self._network = network

    def run(self):
        """
        Main Process:
            1. Initialization stage.

            2. FL communication stage.

            3. Shutdown stage, then close network connection.
        """
        self.setup()
        self.main_loop()
        self.shutdown()

    def setup(self, *args, **kwargs):
        """Initialize network connection and necessary setups.

        Note:
```

(continues on next page)

(continued from previous page)

```
            At first, ``self._network.init_network_connection()`` is required to be␣
↪called.
            Overwrite this method to implement system setup message communication␣
↪procedure.
        """
        self._network.init_network_connection()

    def main_loop(self, *args, **kwargs):
        """Define the actions of communication stage."""
        raise NotImplementedError()

    def shutdown(self, *args, **kwargs):
        """Shut down stage"""
        self._network.close_network_connection()
```

FedLab provides 2 standard communication pattern implementations: synchronous and asynchronous. And we encourage users create new FL communication pattern for their own algorithms.

You can customize process flow by: 1. create a new class inherited from corresponding class in our standard implementations; 2. overwrite the functions in target stage. To sum up, communication strategy can be customized by overwriting as the note below mentioned.

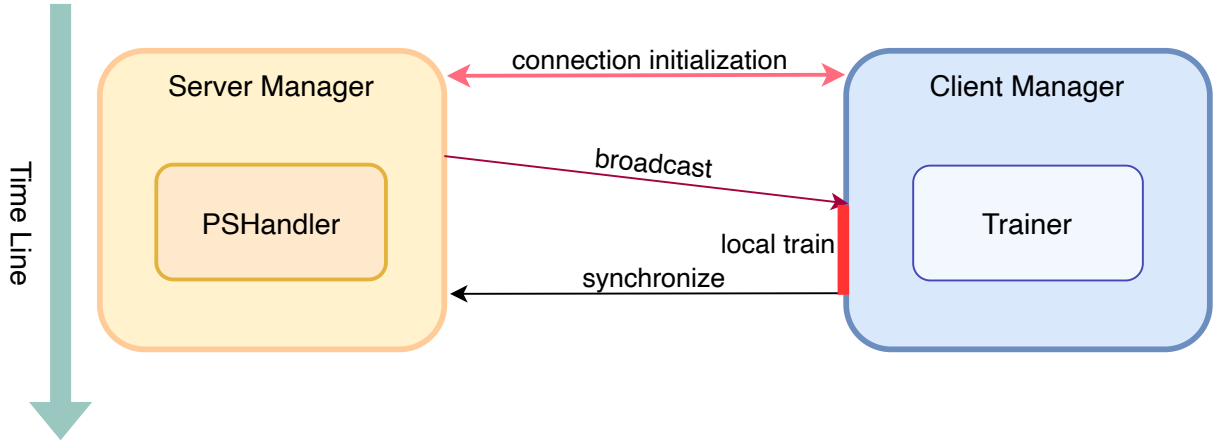
Note:

1. `setup()` defines the network initialization stage. Can be used for FL algorithm initialization.
2. `main_loop()` is the main process of client and server. User need to define the communication strategy for both client and server manager.
3. `shutdown()` defines the shutdown stage.

Importantly, ServerManager and ClientManager should be defined and used as a pair. The control flow and information agreements should be compatible. FedLab provides standard implementation for typical synchronous and asynchronous, as depicted below.

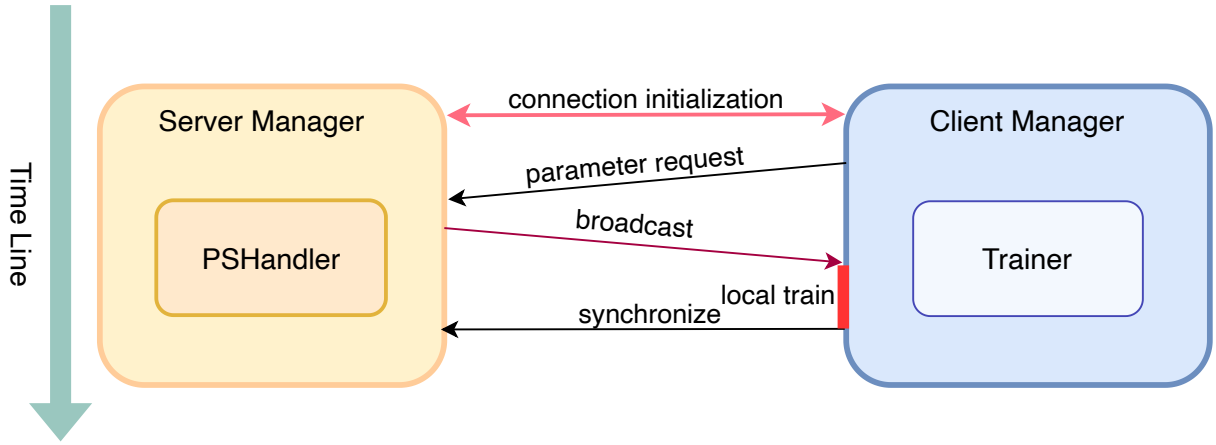
6.2.1 Synchronous mode

Synchronous communication involves `SynchronousServerManager` and `PassiveClientManager`. Communication procedure is shown as follows.



6.2.2 Asynchronous mode

Asynchronous is given by `ServerAsynchronousManager` and `ClientActiveManager`. Communication procedure is shown as follows.



6.2.3 Customization

Initialization stage

Initialization stage is represented by `manager.setup()` function.

User can customize initialization procedure as follows(use ClientManager as example):

```
from fedlab.core.client.manager import PassiveClientManager

class CustomizeClientManager(PassiveClientManager):

    def __init__(self, trainer, network):
        super().__init__(trainer, network)

    def setup(self):
        super().setup()
```

(continues on next page)

(continued from previous page)

```
        ******************************
        *                            *
        *      Write Code Here       *
        *                            *
        ******************************
```

Communication stage

After Initialization Stage, user can define `main_loop()` to define main process for server and client. To standardize **FedLab**'s implementation, here we give the `main_loop()` of `PassiveClientManager`: and `SynchronousServerManager` for example.

Client part:

```
def main_loop(self):
    """Actions to perform when receiving new message, including local training

    Main procedure of each client:
        1. client waits for data from server PASSIVELY
        2. after receiving data, client trains local model.
        3. client synchronizes with server actively.
    """
    while True:
        sender_rank, message_code, payload = self._network.recv(src=0)
        if message_code == MessageCode.Exit:
            break
        elif message_code == MessageCode.ParameterUpdate:
            self._trainer.local_process(payload=payload)
            self.synchronize()
        else:
            raise ValueError("Invalid MessageCode {}.".format(message_code))
```

Server Part:

```
def main_loop(self):
    """Actions to perform in server when receiving a package from one client.

    Server transmits received package to backend computation handler for aggregation or␣
↪others
    manipulations.

    Loop:
        1 activate clients.

        2 listen for message from clients -> transmit received parameters to server␣
↪backend.

    Note:
        Communication agreements related: user can overwrite this function to customize
        communication agreements. This method is key component connecting behaviors of
        :class:`ParameterServerBackendHandler` and :class:`NetworkManager`.
```

(continues on next page)

(continued from previous page)

```
    Raises:
        Exception: Unexpected :class:`MessageCode`.
    """
    while self._handler.stop_condition() is not True:
        activate = threading.Thread(target=self.activate_clients)
        activate.start()
        while True:
            sender_rank, message_code, payload = self._network.recv()
            if message_code == MessageCode.ParameterUpdate:
                if self._handler.iterate_global_model(sender_rank, payload=paylaod):
                    break
            else:
                raise Exception(
                    raise ValueError("Invalid MessageCode {}.".format(message_code))
```

Shutdown stage

`shutdown()` will be called when `main_loop()` finished. You can define the actions for client and server seperately.

Typically in our implementation, shutdown stage is started by server. It will send a message with `MessageCode.Exit` to inform client to stop its main loop.

Codes below is the actions of `SynchronousServerManager` in shutdown stage.

```
def shutdown(self):
    self.shutdown_clients()
    super().shutdown()

def shutdown_clients(self):
    """Shut down all clients.

    Send package to every client with :attr:`MessageCode.Exit` to client.
    """
    for rank in range(1, self._network.world_size):
        print("stopping clients rank:", rank)
        self._network.send(message_code=MessageCode.Exit, dst=rank)
```

6.3 Federated Optimization

Standard FL Optimization contains two parts: 1. local train in client; 2. global aggregation in server. Local train and aggregation procedure are customizable in FedLab. You need to define `ClientTrainer` and `ParameterServerBackendHandler`.

Since `ClientTrainer` and `ParameterServerBackendHandler` are required to manipulate PyTorch Model. They are both inherited from `ModelMaintainer`.

```
class ModelMaintainer(object):
"""Maintain PyTorch model.

    Provide necessary attributes and operation methods.
```

(continues on next page)

(continued from previous page)

```
    Args:
        model (torch.Module): PyTorch model.
        cuda (bool): use GPUs or not.
    """
    def __init__(self, model, cuda) -> None:

        self.cuda = cuda

        if cuda:
            # dynamic gpu acquire.
            self.gpu = get_best_gpu()
            self._model = model.cuda(self.gpu)
        else:
            self._model = model.cpu()

    @property
    def model(self):
        """Return torch.nn.module"""
        return self._model

    @property
    def model_parameters(self):
        """Return serialized model parameters."""
        return SerializationTool.serialize_model(self._model)

    @property
    def shape_list(self):
        """Return shape of parameters"""
        shape_list = [param.shape for param in self._model.parameters()]
        return shape_list
```

6.3.1 Client local training

The basic class of ClientTrainer is shown below, we encourage users define local training process following our code pattern:

```
class ClientTrainer(ModelMaintainer):
    """An abstract class representing a client backend trainer.

    In our framework, we define the backend of client trainer show manage its local␣
↪model.
    It should have a function to update its model called :meth:`local_process`.

    If you use our framework to define the activities of client, please make sure that␣
↪your self-defined class
    should subclass it. All subclasses should overwrite :meth:`local_process`.

    Args:
        model (torch.nn.Module): PyTorch model.
        cuda (bool): Use GPUs or not.
```

(continues on next page)

(continued from previous page)

```
    """

    def __init__(self, model, cuda):
        super().__init__(model, cuda)
        self.client_num = 1  # default is 1.
        self.type = ORDINARY_TRAINER

    @property
    def uplink_package(self):
        """Return a tensor list for uploading to server.

            This attribute will be called by client manager.
            Customize it for new algorithms.
        """
        return [self.model_parameters]

    def local_process(self, payload):
        """Manager of the upper layer will call this function with accepted payload"""
        raise NotImplementedError()

    def train(self):
        """Override this method to define the algorithm of training your model. This␣
↪function should manipulate :attr:`self._model`"""
        raise NotImplementedError()

    def evaluate(self):
        """Evaluate quality of local model."""
        raise NotImplementedError()
```

- Overwrite `ClientTrainer.local_process()` to define local procedure. Typically, you need to implement standard training pipeline of PyTorch.
- Attributes `model` and `model_parameters` is is associated with `self._model`. Please make sure the function `local_process()` will manipulate `self._model`.

A standard implementation of this part is in :class:`SGDClientTrainer`.

6.3.2 Server global aggregation

Calculation tasks related with PyTorch should be define in ServerHandler part. In **FedLab**, our basic class of Handler is defined in `ParameterServerBackendHandler`.

```
class ParameterServerBackendHandler(ModelMaintainer):
    """An abstract class representing handler of parameter server.

    Please make sure that your self-defined server handler class subclasses this class

    Example:
        Read source code of :class:`SyncParameterServerHandler` and␣
↪:class:`AsyncParameterServerHandler`.
    """
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, model, cuda=False):
        super().__init__(model, cuda)

    @property
    def downlink_package(self):
        """Property for manager layer. Server manager will call this property when␣
↪activates clients."""
        return [self.model_parameters]

    @property
    def if_stop(self):
        """:class:`NetworkManager` keeps monitoring this attribute, and it will stop all␣
↪related processes and threads when ``True`` returned."""
        return False

    def _update_global_model(self, *args, **kwargs):
        """Override this function for iterating global model (aggregation or␣
↪optimization)."""
        raise NotImplementedError()
```

User can define server aggregation strategy by finish following functions:

- You can overwrite `_update_global_model()` to customize global procedure.
- `_update_global_model()` is required to manipulate global model parameters (self._model).
- Summarised FL aggregation strategies are implemented in `fedlab.utils.aggregator`.

A standard implementation of this part is in SyncParameterServerHandler.

6.4 Federated Dataset and DataPartitioner

Sophisticated in real world, FL need to handle various kind of data distribution scenarios, including iid and non-iid scenarios. Though there already exists some datasets and partition schemes for published data benchmark, it still can be very messy and hard for researchers to partition datasets according to their specific research problems, and maintain partition results during simulation. FedLab provides *fedlab.utils.dataset.partition.DataPartitioner* that allows you to use pre-partitioned datasets as well as your own data. `DataPartitioner` stores sample indices for each client given a data partition scheme. Also, FedLab provides some extra datasets that are used in current FL researches while not provided by official Pytorch `torchvision.datasets` yet.

Note: Current implementation and design of this part are based on LEAF [2], Acar *et al.* [5], Yurochkin *et al.* [6] and NIID-Bench [7].

6.4.1 Vision Data

CIFAR10

FedLab provides a number of pre-defined partition schemes for some datasets (such as CIFAR10) that subclass *fedlab.utils.dataset.partition.DataPartitioner* and implement functions specific to particular partition scheme. They can be used to prototype and benchmark your FL algorithms.

Tutorial for `CIFAR10Partitioner`: CIFAR10 tutorial.

CIFAR100

Notebook tutorial for `CIFAR100Partitioner`: CIFAR100 tutorial.

FMNIST

Notebook tutorial for data partition of FMNIST (FashionMNIST) : FMNIST tutorial.

MNIST

MNIST is very similar with FMNIST, please check FMNIST tutorial.

SVHN

Data partition tutorial for SVHN: SVHN tutorial

CelebA

Data partition for CelebA: CelebA tutorial.

FEMNIST

Data partition of FEMNIST: FEMNIST tutorial.

6.4.2 Text Data

Shakespeare

Data partition of Shakespeare dataset: Shakespeare tutorial.

Sent140

Data partition of Sent140: Sent140 tutorial.

Reddit

Data partition of Reddit: Reddit tutorial.

6.4.3 Tabular Data

Adult

Adult is from LIBSVM Data. Its original source is from UCI/Adult. FedLab provides both `Dataset` and `DataPartitioner` for Adult. Notebook tutorial for Adult: Adult tutorial.

Covtype

Covtype is from LIBSVM Data. Its original source is from UCI/Covtype. FedLab provides both `Dataset` and `DataPartitioner` for Covtype. Notebook tutorial for Covtype: Covtype tutorial.

RCV1

RCV1 is from LIBSVM Data. Its original source is from UCI/RCV1. FedLab provides both `Dataset` and `DataPartitioner` for RCV1. Notebook tutorial for RCV1: RCV1 tutorial.

6.4.4 Synthetic Data

FCUBE

FCUBE is a synthetic dataset for federated learning. FedLab provides both `Dataset` and `DataPartitioner` for FCUBE. Tutorial for FCUBE: FCUBE tutorial.

LEAF-Synthetic

LEAF-Synthetic is a federated dataset proposed by LEAF. Client number, class number and feature dimensions can all be customized by user.

Please check LEAF-Synthetic for more details.

6.5 Deploy FedLab Process in a Docker Container

6.5.1 Why docker?

The communication APIs of **FedLab** is built on torch.distributed. In cross-process scene, when multiple **FedLab** processes are deployed on the same machine, GPU memory buckets will be created automatically however which are not used in our framework. We can start the **FedLab** processes in different docker containers to avoid triggering GPU memory buckets (to save GPU memory).

6.5.2 Setup docker environment

In this section, we introduce how to setup a docker image for **FedLab** program. Here we provide the Dockerfile for building a FedLab image. Our FedLab environment is based on PytTorch. Therefore, we just need install **FedLab** on the provided PytTorch image.

Dockerfile:

```
# This is an example of fedlab installation via Dockerfile

# replace the value of TORCH_CONTAINER with pytorch image that satisfies your cuda␣
↪version
# you can find it in https://hub.docker.com/r/pytorch/pytorch/tags
ARG TORCH_CONTAINER=1.5-cuda10.1-cudnn7-runtime

FROM pytorch/pytorch:${TORCH_CONTAINER}

RUN pip install --upgrade pip \
    & pip uninstall -y torch torchvision  \
    & conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/
↪free/ \
    & conda config --set show_channel_urls yes \
    & mkdir /root/tmp/

# replace with the correct install command, which you can find in https://pytorch.org/
↪get-started/previous-versions/
RUN conda install -y pytorch==1.7.1 torchvision==0.8.2 cudatoolkit=10.1 -c pytorch

# pip install fedlab
RUN TMPDIR=/root/tmp/ pip install -i https://pypi.mirrors.ustc.edu.cn/simple/ fedlab
```

6.5.3 Dockerfile for different platforms

The steps of modifying Dockerfile for different platforms:

- **Step 1:** Find an appropriate base pytorch image for your platform from dockerhub https://hub.docker.com/r/pytorch/pytorch/tags. Then, replace the value of *TORCH_CONTAINER* in demo dockerfile.
- **Step 2:** To install specific PyTorch version, you need to choose a correct install command, which can be find in https://pytorch.org/get-started/previous-versions/. Then, modify the 16-th command in demo dockerfile.
- **Step 3:** Build the images for your own platform by running the command below in the dir of Dockerfile.

```
$ docker build -t image_name .
```

Warning: Using "–gpus all" and "–network=host" when start a docker container:

```
$ docker run -itd --gpus all --network=host b23a9c46cd04(image name) /bin/bash
```

If you are not in China area, it is ok to remove line 11,12 and "-i https://pypi.mirrors.ustc.edu.cn/simple/" in line 19.

- **Finally:** Run your FedLab process in the different started containers.

Learn Distributed Network Basics Step-by-step guide on distributed network setup and package transmission.

How to Customize Communication Strategy? Use `NetworkManager` to customize communication strategies, including synchronous and asynchronous communication.

How to Customize Federated Optimization? Define your own model optimization process for both server and client.

Federated Datasets and Data Partitioner Get federated datasets and data partition for IID and non-IID setting.

CHAPTER

SEVEN

EXAMPLES

7.1 Quick Start

In this page, we introduce the provided quick start demos. And the start scripts for FL simulation system with FedLab in different scenario. We implement FedAvg algorithm with MLP network and partitioned MNIST dataset across clients.

Source code can be seen in fedlab/examples/.

7.1.1 Download dataset

FedLab provides scripts for common dataset download and partition process. Besides, FL dataset baseline LEAF [2] is also implemented and compatible with PyTorch interfaces.

Codes related to dataset download process are available at `fedlab_benchamrks/datasets/{dataset name}`.

1. Download MNIST/CIFAR10

```
$ cd fedlab_benchamrks/datasets/{mnist or cifar10}/
$ python download_{dataset}.py
```

2. Partition

Run follow python file to generate partition file.

```
$ python {dataset}_partition.py
```

Source codes of partition scripts:

```
import torchvision
from fedlab.utils.functional import save_dict
from fedlab.utils.dataset.slicing import noniid_slicing, random_slicing

trainset = torchvision.datasets.CIFAR10(root=root, train=True, download=True)
# trainset = torchvision.datasets.MNIST(root=root, train=True, download=True)

data_indices = noniid_slicing(trainset, num_clients=100, num_shards=200)
save_dict(data_indices, "cifar10_noniid.pkl")

data_indices = random_slicing(trainset, num_clients=100)
save_dict(data_indices, "cifar10_iid.pkl")
```

`data_indices` is a `dict` mapping from client id to data indices(list) of raw dataset. **FedLab** provides random partition and non-I.I.D. partition methods, in which the noniid partition method is totally re-implementation in paper FedAvg.

3. LEAF dataset process

Please follow the FedLab benchmark to learn how to generate LEAF related dataset partition.

Run FedLab demos

FedLab provides both asynchronous and synchronous standard implementation demos for uses to learn. We only introduce the usage of synchronous FL system simulation demo(FedAvg) with different scenario in this page. (Code structures are similar.)

We are very confident in the readability of FedLab code, so we recommend that users read the source code according to the following demos for better understanding.

1. Standalone

Source code is under fedlab/examples/standalone-mnist. This is a standard usage of `SerialTrainer` which allows users to simulate a group of clients with a single process.

```
$ python standalone.py --total_client 100 --com_round 3 --sample_ratio 0.1 --batch_size␣
↪100 --epochs 5 --lr 0.02
```

or

```
$ bash launch_eg.sh
```

Run command above to start a single process simulating FedAvg algorithm with 100 clients with 10 communication round in total, with 10 clients sampled randomly at each round .

2. Cross-process

Source code is under fedlab/examples/cross-process-mnist

Start a FL simulation with 1 server and 2 clients.

```
$ bash launch_eg.sh
```

The content of `launch_eg.sh` is:

```
python server.py --ip 127.0.0.1 --port 3001 --world_size 3 --round 3 &

python client.py --ip 127.0.0.1 --port 3001 --world_size 3 --rank 1 &

python client.py --ip 127.0.0.1 --port 3001 --world_size 3 --rank 2  &

wait
```

Cross-process scenario allows users deploy their FL system in computer cluster. Although in this case, we set the address of server as localhost. Then three process will communicate with each other following standard FL procedure.

Note: Due to the rank of torch.distributed is unique for every process. Therefore, we use rank represent client id in this scenario.

---

3. Cross-process with SerialTrainer

`SerialTrainer` uses less computer resources (single process) to simulate multiple clients. Cross-pross is suit for computer cluster deployment, simulating data-center FL system. In our experiment, the world size of `torch.distributed` can't more than 50 (Denpends on clusters), otherwise, the socket will crash, which limited the client number of FL simulation.

To improve scalability, FedLab provides scale standard implementation to combine `SerialTrainer` and `ClientManager`, which allows a single process simulate multiple clients.

Source codes are available in fedlab_benchamrks/algorithm/fedavg/scale/{experiment setting name}.

Here, we take mnist-cnn as example to introduce this demo. In this demo, we set world_size=11 (1 ServerManager, 10 ClientManagers), and each ClientManager represents 10 local client dataset partition. Our data partition strategy follows the experimental setting of fedavg as well. In this way, **we only use 11 processes to simulate a FL system with 100 clients.**

To start this system, you need to open at least 2 terminal (we still use localhost as demo. Use multiple machines is OK as long as with right network configuration):

1. server (terminal 1)

```
$ python server.py --ip 127.0.0.1 --port 3002 --world_size 11
```

2. clients (terminal 2)

```
$ bash start_clt.sh 11 1 10 # launch clients from rank 1 to rank 10 with world_size 11
```

The content of `start_clt.sh`:

```
for ((i=$2; i<=$3; i++))
do
{
    echo "client ${i} started"
    python client.py --world_size $1 --rank ${i} &
    sleep 2s # wait for gpu resources allocation
}
done
wait
```

4. Hierachical

Hierarchical mode for **FedLab** is designed for situation tasks on multiple computer clusters (in different LAN) or the real-world scenes. To enable the inter-connection for different computer clusters, **FedLab** develops `Scheduler` as middle-server process to connect client groups. Each `Scheduler` manages the communication between the global server and clients in a client group. And server can communicate with clients in different LAN via corresponding `Scheduler`. The computation mode of a client group for each scheduler can be either **standalone** or **cross-process**.

The demo of Hierachical with hybrid client (standalone and serial trainer) is given in fedlab/examples/hierarchical-hybrid-mnist.

Run all scripts together:

```
$ bash launch_eg.sh
```

Run scripts seperately:

```
# Top server in terminal 1
$ bash launch_topserver_eg.sh

# Scheduler1 + Ordinary trainer with 1 client + Serial trainer with 10 clients in␣
↪terminal 2:
bash launch_cgroup1_eg.sh

# Scheduler2 + Ordinary trainer with 1 client + Serial trainer with 10 clients in␣
↪terminal 3:
$ bash launch_cgroup2_eg.sh
```

7.2 PyTorch version of LEAF

FedLab migrates the TensorFlow version of LEAF dataset to the PyTorch framework, and provides the implementation of dataloader for the corresponding dataset. The unified interface is in ``fedlab_benchmarks/leaf/dataloader.py``

This markdown file introduces the process of using LEAF dataset in FedLab.

7.2.1 Description of Leaf datasets

The LEAF benchmark contains the federation settings of Celeba, femnist, Reddit, sent140, shakespeare and synthetic datasets. With reference to leaf-readme.md , the introduction the total number of users and the corresponding task categories of leaf datasets are given below.

1. FEMNIST
   - **Overview:** Image Dataset.
   - **Details:** 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), 3500 users.
   - **Task:** Image Classification.
2. Sentiment140
   - **Overview:** Text Dataset of Tweets.
   - **Details** 660120 users.
   - **Task:** Sentiment Analysis.
3. Shakespeare
   - **Overview:** Text Dataset of Shakespeare Dialogues.
   - **Details:** 1129 users (reduced to 660 with our choice of sequence length. See bug.)
   - **Task:** Next-Character Prediction.
4. Celeba
   - **Overview:** Image Dataset based on the Large-scale CelebFaces Attributes Dataset.
   - **Details:** 9343 users (we exclude celebrities with less than 5 images).
   - **Task:** Image Classification (Smiling vs. Not smiling).
5. Synthetic Dataset

- **Overview:** We propose a process to generate synthetic, challenging federated datasets. The high-level goal is to create devices whose true models are device-dependant. To see a description of the whole generative process, please refer to the paper.
- **Details:** The user can customize the number of devices, the number of classes and the number of dimensions, among others.
- **Task:** Classification.

6. Reddit

- **Overview:** We preprocess the Reddit data released by pushshift.io corresponding to December 2017.
- **Details:** 1,660,820 users with a total of 56,587,343 comments.
- **Task:** Next-word Prediction.

7.2.2 Download and preprocess data

For the six types of leaf datasets, refer to leaf/data and provide data download and preprocessing scripts in `fedlab _ benchmarks/datasets/data`. In order to facilitate developers to use leaf, fedlab integrates the download and processing scripts of leaf six types of data sets into `fedlab_benchmarks/datasets/data`, which stores the download scripts of various data sets.

Common structure of leaf dataset folders:

```
/FedLab/fedlab_benchmarks/datasets/{leaf_dataset_name}

    ├── {other_useful_preprocess_util}
    ├── prerpocess.sh
    ├── stats.sh
    └── README.md
```

- `preprocess.sh`: downloads and preprocesses the dataset
- `stats.sh`: performs information statistics on all data (stored in `./data/all_data/all_data.json`) processed by `preprocess.sh`
- `README.md`: gives a detailed description of the process of downloading and preprocessing the dataset, including parameter descriptions and precautions.

> **Developers can directly run the executable script ``create_datasets_and_save.sh`` to obtain the dataset, process and store the corresponding dataset data in the form of a pickle file.** This script provides an example of using the preprocess.sh script, and developers can modify the parameters according to application requirements.

preprocess.sh Script usage example:

```
cd fedlab_benchmarks/datasets/data/femnist
bash preprocess.sh -s niid --sf 0.05 -k 0 -t sample

cd fedlab_benchmarks/datasets/data/shakespeare
bash preprocess.sh -s niid --sf 0.2 -k 0 -t sample -tf 0.8

cd fedlab_benchmarks/datasets/data/sent140
bash ./preprocess.sh -s niid --sf 0.05 -k 3 -t sample

cd fedlab_benchmarks/datasets/data/celeba
```

(continues on next page)

(continued from previous page)

```
bash ./preprocess.sh -s niid --sf 0.05 -k 5 -t sample

cd fedlab_benchmarks/datasets/data/synthetic
bash ./preprocess.sh -s niid --sf 1.0 -k 5 -t sample --tf 0.6

# for reddit, see its README.md to download preprocessed dataset manually
```

By setting parameters for `preprocess.sh`, the original data can be sampled and spilted. The `readme.md` in each dataset folder provides the example and explanation of script parameters, the common parameters are:

1. `-s` := 'iid' to sample in an i.i.d. manner, or 'niid' to sample in a non-i.i.d. manner; more information on i.i.d. versus non-i.i.d. is included in the 'Notes' section.
2. `--sf` := fraction of data to sample, written as a decimal; default is 0.1.
3. `-k` := minimum number of samples per user
4. `-t` := 'user' to partition users into train-test groups, or 'sample' to partition each user's samples into train-test groups
5. `--tf` := fraction of data in training set, written as a decimal; default is 0.9, representing train set: test set = 9:1.

At present, FedLab's Leaf experiment need provided training data and test data, so we needs to provide related data training set-test set splitting parameter for `preprocess.sh` to carry out the experiment, default is 0.9.

If you need to obtain or split data again, make sure to delete `data` folder in the dataset directory before re-running `preprocess.sh` to download and preprocess data.

7.2.3 Pickle file stores Dataset.

In order to speed up developers' reading data, fedlab provides a method of processing raw data into Dataset and storing it as a pickle file. The Dataset of the corresponding data of each client can be obtained by reading the pickle file after data processing.

set the parameters and run `create_pickle_dataset.py`. The usage example is as follows:

```
cd fedlab_benchmarks/leaf/process_data
python create_pickle_dataset.py --data_root "../../datasets" --save_root "./pickle_
↪dataset" --dataset_name "shakespeare"
```

Parameter Description:

1. `data_root` : the root path for storing leaf data sets, which contains all leaf data sets; If you use the `Fedlab_benchmarks/datasets/` provided by fedlab to download leaf data, 'data_root' can be set to this path, a relative address of which is shown in this example.
2. `save_root`: directory to store the pickle file address of the processed Dataset; Each dataset Dataset will be saved in `{save_root}/{dataset_name}/{train,test}`; the example is to create a `pickle_dataset` folder under the current path to store all pickle dataset files.
3. `dataset_name`: Specify the name of the leaf data set to be processed. There are six options {femnist, shakespeare, celeba, sent140, synthetic, reddit}.

7.2.4 Dataloader loading data set

Leaf datasets are loaded by `dataloader.py` (located under `fedlab_benchmarks/leaf/dataloader.py`). All returned data types are pytorch Dataloader.

By calling this interface and specifying the name of the data set, the corresponding Dataloader can be obtained.

Example of use:

```
from leaf.dataloader import get_LEAF_dataloader
def get_femnist_shakespeare_dataset(args):
    if args.dataset == 'femnist' or args.dataset == 'shakespeare':
        trainloader, testloader = get_LEAF_dataloader(dataset=args.dataset,
                                                      client_id=args.rank)
    else:
        raise ValueError("Invalid dataset:", args.dataset)

    return trainloader, testloader
```

7.2.5 Run experiment

The current experiment of LEAF data set is the **single-machine multi-process** scenario under FedAvg's Cross machine implement, and the tests of femnist and Shakespeare data sets have been completed.

Run `fedlab_benchmarks/fedavg/cross_machine/LEAF_test.sh` to quickly execute the simulation experiment of fedavg under leaf data set.

Quick Start PyTorch version of LEAF

CHAPTER

EIGHT

CONTRIBUTING TO FEDLAB

8.1 Reporting bugs

We use GitHub issues to track all bugs and feature requests. Feel free to open an issue if you have found a bug or wish to see a feature implemented.

In case you experience issues using this package, do not hesitate to submit a ticket to the Bug Tracker. You are also welcome to post feature requests or pull requests.

8.2 Contributing Code

You're welcome to contribute to this project through **Pull Request**. By contributing, you agree that your contributions will be licensed under Apache License, Version 2.0

We encourage you to contribute to the improvement of FedLab or the FedLab implementation of existing FL methods. The preferred workflow for contributing to FedLab is to fork the main repository on GitHub, clone, and develop on a branch. Steps as follow:

1. Fork the project repository by clicking on the 'Fork'. For contributing new features, please fork FedLab core repo or new implementations for FedLab benchmarks repo.
2. Clone your fork of repo from your GitHub to your local:

```
$ git clone git@github.com:YourLogin/FedLab.git
$ cd FedLab
```

3. Create a new branch to save your changes:

```
$ git checkout -b my-feature
```

4. Develop the feature on your branch.

```
$ git add modified_files
$ git commit
```

8.3 Pull Request Checklist

- Please follow the file structure below for new features or create new file if there are something new.

```
fedlab
    ├── core
    │   ├── communicator            # communication module
    │   ├── client                  # client related implementations
    │   └── server                  # server related implementations
    │       └── hierarchical        # hierarchical communication pattern␣
↪modules
    └── utils                       # functional modules
        └── dataset                 # functional modules about dataset
```

- The code should provide test cases using *unittest.TestCase*. And ensure all local tests passed:

```
$ python test_bench.py
```

- All public methods should have informative docstrings with sample usage presented as doctests when appropriate. Docstring and code should follow Google Python Style Guide: | English.

CHAPTER

NINE

REFERENCE

CHAPTER

TEN

API REFERENCE

This page contains auto-generated API reference documentation[1].

10.1 fedlab

10.1.1 core

client

manager

Module Contents

ClientManager	Base class for ClientManager.
PassiveClientManager	Passive communication `NetworkManager` for client in synchronous FL pattern.
ActiveClientManager	Active communication `NetworkManager` for client in asynchronous FL pattern.

class ClientManager(*network*, *trainer*)

Bases: *fedlab.core.network_manager.NetworkManager*

Base class for ClientManager.

ClientManager defines client activation in different communication stages.

Parameters

- **network** (DistNetwork) – Network configuration and interfaces.
- **trainer** (ClientTrainer) – Subclass of `ClientTrainer`. Provides `local_process()` and `uplink_package`. Define local client training procedure.

setup(*self*)

Initialization stage.

ClientManager reports number of clients simulated by current client process.

[1] Created with sphinx-autoapi

class PassiveClientManager(*network*, *trainer*, *logger=None*)

Bases: *ClientManager*

Passive communication NetworkManager for client in synchronous FL pattern.

Parameters

- **network** (DistNetwork) – Network configuration and interfaces.
- **trainer** (ClientTrainer) – Subclass of ClientTrainer. Provides local_process() and uplink_package. Define local client training procedure.
- **logger** (Logger, *optional*) – Object of Logger.

main_loop(*self*)

Actions to perform when receiving a new message, including local training.

Main procedure of each client:

1. client waits for data from server (PASSIVELY).
2. after receiving data, client start local model training procedure.
3. client synchronizes with server actively.

synchronize(*self*)

Synchronize with server

class ActiveClientManager(*network*, *trainer*, *logger=None*)

Bases: *ClientManager*

Active communication NetworkManager for client in asynchronous FL pattern.

Parameters

- **network** (DistNetwork) – Network configuration and interfaces.
- **trainer** (ClientTrainer) – Subclass of ClientTrainer. Provides local_process() and uplink_package. Define local client training procedure.
- **logger** (Logger, *optional*) – Object of Logger.

main_loop(*self*)

Actions to perform on receiving new message, including local training

1. client requests data from server (ACTIVELY).
2. after receiving data, client will train local model.
3. client will synchronize with server actively.

request(*self*)

Client request

synchronize(*self*)

Synchronize with server

serial_trainer

Module Contents

SerialTrainer	Base class. Simulate multiple clients in sequence in a single process.
SubsetSerialTrainer	Train multiple clients in a single process.

class SerialTrainer(*model*, *client_num*, *cuda=False*, *logger=None*)

Bases: *fedlab.core.client.trainer.ClientTrainer*

Base class. Simulate multiple clients in sequence in a single process.

Parameters

- **model** (*torch.nn.Module*) – Model used in this federation.
- **client_num** (*int*) – Number of clients in current trainer.
- **cuda** (*bool*) – Use GPUs or not. Default: `False`.
- **logger** (`Logger`, *optional*) – Object of `Logger`.

property uplink_package(*self*)

Return a tensor list for uploading to server.

This attribute will be called by client manager. Customize it for new algorithms.

abstract _train_alone(*self*, *model_parameters*, *train_loader*)

Train local model with `model_parameters` on `train_loader`.

Parameters

- **model_parameters** (*torch.Tensor*) – Serialized model parameters of one model.
- **train_loader** (*torch.utils.data.DataLoader*) – `torch.utils.data.DataLoader` for this client.

abstract _get_dataloader(*self*, *client_id*)

Get `DataLoader` for `client_id`.

local_process(*self*, *id_list*, *payload*)

Train local model with different dataset according to client id in `id_list`.

Parameters

- **id_list** (*list[int]*) – Client id in this training serial.
- **payload** (*list[torch.Tensor]*) – communication payload from server.

class SubsetSerialTrainer(*model*, *dataset*, *data_slices*, *logger=None*, *cuda=False*, *args={'epochs': 5, 'batch_size': 100, 'lr': 0.1}*)

Bases: *SerialTrainer*

Train multiple clients in a single process.

Customize *_get_dataloader()* or *_train_alone()* for specific algorithm design in clients.

Parameters

- **model** (*torch.nn.Module*) – Model used in this federation.

- **dataset** (*torch.utils.data.Dataset*) – Local dataset for this group of clients.
- **data_slices** (*list[list]*) – Subset of indices of dataset.
- **logger** (*Logger, optional*) – Object of Logger.
- **cuda** (*bool*) – Use GPUs or not. Default: False.
- **args** (*dict*) – Uncertain variables. Default: {"epochs": 5, "batch_size": 100, "lr": 0.1}

Note: len(data_slices) == client_num, that is, each sub-index of dataset corresponds to a client's local dataset one-by-one.

_get_dataloader(*self*, *client_id*)

Return a training dataloader used in train() for client with id

Parameters **client_id** (*int*) – client_id of client to generate dataloader

Note: client_id here is not equal to client_id in global FL setting. It is the index of client in current *SerialTrainer*.

Returns DataLoader for specific client's sub-dataset

_train_alone(*self*, *model_parameters*, *train_loader*)

Single round of local training for one client.

Note: Overwrite this method to customize the PyTorch training pipeline.

Parameters

- **model_parameters** (*torch.Tensor*) – serialized model parameters.
- **train_loader** (*torch.utils.data.DataLoader*) – torch.utils.data.DataLoader for this client.

trainer

Module Contents

ClientTrainer	An abstract class representing a client trainer.
SGDClientTrainer	Client backend handler, this class provides data process method to upper layer.

class ClientTrainer(*model*, *cuda*)

Bases: *fedlab.core.model_maintainer.ModelMaintainer*

An abstract class representing a client trainer.

In FedLab, we define the backend of client trainer show manage its local model. It should have a function to update its model called *local_process()*.

If you use our framework to define the activities of client, please make sure that your self-defined class should subclass it. All subclasses should overwrite *local_process()* and property *uplink_package*.

Parameters

- **model** (*torch.nn.Module*) – PyTorch model.
- **cuda** (*bool*) – Use GPUs or not.

property uplink_package(*self*) → List[torch.Tensor]

Return a tensor list for uploading to server.

This attribute will be called by client manager. Customize it for new algorithms.

abstract classmethod local_process(*self*, *payload*) → bool

Manager of the upper layer will call this function with accepted payload

In synchronous mode, return True to end current FL round.

abstract train(*self*)

Override this method to define the algorithm of training your model. This function should manipulate `self._model`

abstract evaluate(*self*)

Evaluate quality of local model.

class SGDClientTrainer(*model*, *data_loader*, *epochs*, *optimizer*, *criterion*, *cuda=False*, *logger=None*)

Bases: *ClientTrainer*

Client backend handler, this class provides data process method to upper layer.

Parameters

- **model** (*torch.nn.Module*) – PyTorch model.
- **data_loader** (*torch.utils.data.DataLoader*) – `torch.utils.data.DataLoader` for this client.
- **epochs** (*int*) – the number of local epoch.
- **optimizer** (*torch.optim.Optimizer*) – optimizer for this client's model.
- **criterion** (*torch.nn.Loss*) – loss function used in local training process.
- **cuda** (*bool, optional*) – use GPUs or not. Default: `False`.
- **logger** (`Logger`, *optional*) – :object of `Logger`.

property uplink_package(*self*)

Return a tensor list for uploading to server.

This attribute will be called by client manager. Customize it for new algorithms.

local_process(*self*, *payload*)

Manager of the upper layer will call this function with accepted payload

In synchronous mode, return True to end current FL round.

train(*self*, *model_parameters*) → None

Client trains its local model on local dataset.

Parameters **model_parameters** (*torch.Tensor*) – Serialized model parameters.

Package Contents

ClientManager	Base class for ClientManager.
ActiveClientManager	Active communication `NetworkManager` for client in asynchronous FL pattern.
PassiveClientManager	Passive communication `NetworkManager` for client in synchronous FL pattern.
SerialTrainer	Base class. Simulate multiple clients in sequence in a single process.
SubsetSerialTrainer	Train multiple clients in a single process.

ORDINARY_TRAINER	
SERIAL_TRAINER	

ORDINARY_TRAINER = 0

SERIAL_TRAINER = 1

class ClientManager(*network*, *trainer*)

Bases: *fedlab.core.network_manager.NetworkManager*

Base class for ClientManager.

ClientManager defines client activation in different communication stages.

Parameters

- **network** (DistNetwork) – Network configuration and interfaces.
- **trainer** (ClientTrainer) – Subclass of `ClientTrainer`. Provides `local_process()` and `uplink_package`. Define local client training procedure.

setup(*self*)

Initialization stage.

ClientManager reports number of clients simulated by current client process.

class ActiveClientManager(*network*, *trainer*, *logger=None*)

Bases: *ClientManager*

Active communication `NetworkManager` for client in asynchronous FL pattern.

Parameters

- **network** (DistNetwork) – Network configuration and interfaces.
- **trainer** (ClientTrainer) – Subclass of `ClientTrainer`. Provides `local_process()` and `uplink_package`. Define local client training procedure.
- **logger** (Logger, *optional*) – Object of `Logger`.

main_loop(*self*)

Actions to perform on receiving new message, including local training

1. client requests data from server (ACTIVELY).
2. after receiving data, client will train local model.

3. client will synchronize with server actively.

request(*self*)

Client request

synchronize(*self*)

Synchronize with server

class PassiveClientManager(*network*, *trainer*, *logger=None*)

Bases: *ClientManager*

Passive communication `NetworkManager` for client in synchronous FL pattern.

Parameters

- **network** (`DistNetwork`) – Network configuration and interfaces.
- **trainer** (`ClientTrainer`) – Subclass of `ClientTrainer`. Provides `local_process()` and `uplink_package`. Define local client training procedure.
- **logger** (`Logger, optional`) – Object of `Logger`.

main_loop(*self*)

Actions to perform when receiving a new message, including local training.

Main procedure of each client:

1. client waits for data from server (PASSIVELY).
2. after receiving data, client start local model training procedure.
3. client synchronizes with server actively.

synchronize(*self*)

Synchronize with server

class SerialTrainer(*model*, *client_num*, *cuda=False*, *logger=None*)

Bases: *fedlab.core.client.trainer.ClientTrainer*

Base class. Simulate multiple clients in sequence in a single process.

Parameters

- **model** (*torch.nn.Module*) – Model used in this federation.
- **client_num** (*int*) – Number of clients in current trainer.
- **cuda** (*bool*) – Use GPUs or not. Default: `False`.
- **logger** (`Logger, optional`) – Object of `Logger`.

property uplink_package(*self*)

Return a tensor list for uploading to server.

This attribute will be called by client manager. Customize it for new algorithms.

abstract _train_alone(*self*, *model_parameters*, *train_loader*)

Train local model with `model_parameters` on `train_loader`.

Parameters

- **model_parameters** (*torch.Tensor*) – Serialized model parameters of one model.
- **train_loader** (*torch.utils.data.DataLoader*) – `torch.utils.data.DataLoader` for this client.

abstract _get_dataloader(*self*, *client_id*)

Get `DataLoader` for `client_id`.

local_process(*self*, *id_list*, *payload*)

Train local model with different dataset according to client id in `id_list`.

Parameters

- **id_list** (*list[int]*) – Client id in this training serial.
- **payload** (*list[torch.Tensor]*) – communication payload from server.

class SubsetSerialTrainer(*model*, *dataset*, *data_slices*, *logger=None*, *cuda=False*, *args={'epochs': 5, 'batch_size': 100, 'lr': 0.1}*)

Bases: *SerialTrainer*

Train multiple clients in a single process.

Customize *_get_dataloader()* or *_train_alone()* for specific algorithm design in clients.

Parameters

- **model** (*torch.nn.Module*) – Model used in this federation.
- **dataset** (*torch.utils.data.Dataset*) – Local dataset for this group of clients.
- **data_slices** (*list[list]*) – Subset of indices of dataset.
- **logger** (*Logger, optional*) – Object of `Logger`.
- **cuda** (*bool*) – Use GPUs or not. Default: `False`.
- **args** (*dict*) – Uncertain variables. Default: `{"epochs": 5, "batch_size": 100, "lr": 0.1}`

Note: `len(data_slices) == client_num`, that is, each sub-index of `dataset` corresponds to a client's local dataset one-by-one.

_get_dataloader(*self*, *client_id*)

Return a training dataloader used in `train()` for client with `id`

Parameters **client_id** (*int*) – `client_id` of client to generate dataloader

---

Note: `client_id` here is not equal to `client_id` in global FL setting. It is the index of client in current *SerialTrainer*.

Returns `DataLoader` for specific client's sub-dataset

_train_alone(*self*, *model_parameters*, *train_loader*)

Single round of local training for one client.

Note: Overwrite this method to customize the PyTorch training pipeline.

Parameters

- **model_parameters** (*torch.Tensor*) – serialized model parameters.

- **train_loader** (*torch.utils.data.DataLoader*) – torch.utils.data.DataLoader for this client.

communicator

FedLab communication API

package

Module Contents

Package	A basic network package data structure used in FedLab. Everything is Tensor in FedLab.

supported_torch_dtypes

supported_torch_dtypes

class Package(*message_code=None*, *content=None*)

Bases: object

A basic network package data structure used in FedLab. Everything is Tensor in FedLab.

Note: `slice_size_i = tensor_i.shape[0]`, that is, every element in slices indicates the size of a sub-Tensor in content.

Package **maintains 3 variables:**

- `header : torch.Tensor([sender_rank, recv_rank, content_size, message_code, data_type])`
- `slices : list[slice_size_1, slice_size_2]`
- `content : torch.Tensor([tensor_1, tensor_2, ...])`

Parameters

- **message_code** (MessageCode) – Message code
- **content** (*torch.Tensor, optional*) – Tensors contained in this package.

append_tensor(*self*, *tensor*)

Append new tensor to `Package.content`

Parameters **tensor** (*torch.Tensor*) – Tensor to append in content.

append_tensor_list(*self*, *tensor_list*)

Append a list of tensors to `Package.content`.

Parameters **tensor_list** (*list[torch.Tensor]*) – A list of tensors to append to `Package.content`.

to(*self*, *dtype*)

static parse_content(*slices*, *content*)

Parse package content into a list of tensors

Parameters

- **slices** (*list[int]*) – A list containing number of elements of each tensor. Each number is used as offset in parsing process.
- **content** (*torch.Tensor*) – `Package.content`, a 1-D tensor composed of several 1-D tensors and their corresponding offsets. For more details about *Package*.

Returns A list of 1-D tensors parsed from `content`

Return type list[torch.Tensor]

static parse_header(*header*)

Parse header to get information of current package.

Parameters **header** (*torch.Tensor*) – `Package.header`, a 1-D tensor composed of 4 elements: `torch.Tensor([sender_rank, recv_rank, slice_size, message_code, data_type])`.

:param For more details about *Package*.:

Returns A tuple containing 5 elements: `(sender_rank, recv_rank, slice_size, message_code, data_type)`.

Return type tuple

processor

Module Contents

PackageProcessor	Provide more flexible distributed tensor communication functions based on

class PackageProcessor

Bases: `object`

Provide more flexible distributed tensor communication functions based on `torch.distributed.send()` and `torch.distributed.recv()`.

PackageProcessor defines the details of point-to-point package communication.

EVERYTHING is `torch.Tensor` in FedLab.

static send_package(*package*, *dst*)

Three-segment tensor communication pattern based on `torch.distributed`

Pattern is shown as follows: 1.1 sender: send a header tensor containing `slice_size` to receiver

1.2 receiver: receive the header, and get the value of `slice_size` and create a buffer for incoming slices of content

2.1 sender: send a list of slices indicating the size of every content size.

2.2 receiver: receive the slices list.

3.1 sender: send a content tensor composed of a list of tensors.

3.2 receiver: receive the content tensor, and parse it to obtain slices list using parser function

static recv_package(*src=None*)

Three-segment tensor communication pattern based on `torch.distributed`

Pattern is shown as follows: 1.1 sender: send a header tensor containing `slice_size` to receiver

1.2 receiver: receive the header, and get the value of `slice_size` and create a buffer for incoming slices of content

2.1 sender: send a list of slices indicating the size of every content size.

2.2 receiver: receive the slices list.

3.1 sender: send a content tensor composed of a list of tensors.

3.2 receiver: receive the content tensor, and parse it to obtain slices list using parser function

Package Contents

dtype_torch2flab(torch_type)	
dtype_flab2torch(fedlab_type)	

HEADER_SENDER_RANK_IDX

HEADER_RECEIVER_RANK_IDX

HEADER_SLICE_SIZE_IDX

HEADER_MESSAGE_CODE_IDX

HEADER_DATA_TYPE_IDX

DEFAULT_RECEIVER_RANK

DEFAULT_SLICE_SIZE

DEFAULT_MESSAGE_CODE_VALUE

HEADER_SIZE

INT8

INT16

INT32

INT64

FLOAT16

FLOAT32

FLOAT64

HEADER_SENDER_RANK_IDX = 0

HEADER_RECEIVER_RANK_IDX = 1

HEADER_SLICE_SIZE_IDX = 2

HEADER_MESSAGE_CODE_IDX = 3

HEADER_DATA_TYPE_IDX = 4

DEFAULT_RECEIVER_RANK

DEFAULT_SLICE_SIZE = 0

DEFAULT_MESSAGE_CODE_VALUE = 0

HEADER_SIZE = 5

INT8 = 0

INT16 = 1

INT32 = 2

INT64 = 3

FLOAT16 = 4

FLOAT32 = 5

FLOAT64 = 6

dtype_torch2flab(*torch_type*)

dtype_flab2torch(*fedlab_type*)

server

hierarchical

connector

Module Contents

Connector	Abstract class for basic Connector, which is a sub-module of `Scheduler`.
ServerConnector	Connect with server.
ClientConnector	Connect with clients.

class Connector(*network*, *write_queue*, *read_queue*)

Bases: *fedlab.core.network_manager.NetworkManager*

Abstract class for basic Connector, which is a sub-module of `Scheduler`.

Connector inherits `NetworkManager`, maintaining two Message Queue. One is for sending messages to collaborator, the other is for read messages from others.

Note: Connector is a basic component for scheduler, Example code can be seen in `scheduler.py`.

Parameters

- **network** (DistNetwork) – Manage `torch.distributed` network communication.
- **write_queue** (*torch.multiprocessing.Queue*) – Message queue to write.
- **read_queue** (*torch.multiprocessing.Queue*) – Message queue to read.

abstract process_meessage_queue(*self*)

Define the procedure of dealing with message queue.

class ServerConnector(*network*, *write_queue*, *read_queue*, *logger*)

Bases: *Connector*

Connect with server.

this process will act like a client.

This class is a part of middle server which used in hierarchical structure.

Parameters

- **network** (DistNetwork) – Network configuration and interfaces.
- **write_queue** (*torch.multiprocessing.Queue*) – Message queue to write.
- **read_queue** (*torch.multiprocessing.Queue*) – Message queue to read.
- **logger** (*Logger, optional*) – object of `Logger`.

run(*self*)

Main Process:

1. Initialization stage.
2. FL communication stage.
3. Shutdown stage. Close network connection.

setup(*self*, **args*, ***kwargs*)

Initialize network connection and necessary setups.

At first, `self._network.init_network_connection()` is required to be called.

Overwrite this method to implement system setup message communication procedure.

main_loop(*self*, **args*, ***kwargs*)

Define the actions of communication stage.

process_meessage_queue(*self*)

client -> server directly transport.

class ClientConnector(*network*, *write_queue*, *read_queue*, *logger*)

Bases: *Connector*

Connect with clients.

This class is a part of middle server which used in hierarchical structure.

Parameters

- **network** (DistNetwork) – Network configuration and interfaces.
- **write_queue** (*torch.multiprocessing.Queue*) – Message queue to write.
- **read_queue** (*torch.multiprocessing.Queue*) – Message queue to read.

run(*self*)

Main Process:

1. Initialization stage.
2. FL communication stage.
3. Shutdown stage. Close network connection.

setup(*self*, **args*, ***kwargs*)

Initialize network connection and necessary setups.

At first, `self._network.init_network_connection()` is required to be called.

Overwrite this method to implement system setup message communication procedure.

main_loop(*self*)

Define the actions of communication stage.

process_meessage_queue(*self*)

Process message queue

Strategy of processing message from server.

scheduler

Module Contents

Scheduler	Middle Topology for hierarchical communication pattern.

class Scheduler(*net_upper*, *net_lower*)

Middle Topology for hierarchical communication pattern.

Scheduler uses message queues to decouple connector modules.

Parameters

- **net_upper** (DistNetwork) – Distributed network manager of server from upper level.
- **net_lower** (DistNetwork) – Distributed network manager of clients from lower level.

run(*self*)

Package Contents

ClientConnector	Connect with clients.
ServerConnector	Connect with server.
Scheduler	Middle Topology for hierarchical communication pattern.

class ClientConnector(*network*, *write_queue*, *read_queue*, *logger*)

Bases: Connector

Connect with clients.

This class is a part of middle server which used in hierarchical structure.

Parameters

- **network** (DistNetwork) – Network configuration and interfaces.
- **write_queue** (*torch.multiprocessing.Queue*) – Message queue to write.
- **read_queue** (*torch.multiprocessing.Queue*) – Message queue to read.

run(*self*)

Main Process:

1. Initialization stage.
2. FL communication stage.

3. Shutdown stage. Close network connection.

setup(*self*, **args*, ***kwargs*)

Initialize network connection and necessary setups.

At first, `self._network.init_network_connection()` is required to be called.

Overwrite this method to implement system setup message communication procedure.

main_loop(*self*)

Define the actions of communication stage.

process_meessage_queue(*self*)

Process message queue

Strategy of processing message from server.

class ServerConnector(*network*, *write_queue*, *read_queue*, *logger*)

Bases: `Connector`

Connect with server.

this process will act like a client.

This class is a part of middle server which used in hierarchical structure.

Parameters

- **network** (DistNetwork) – Network configuration and interfaces.
- **write_queue** (*torch.multiprocessing.Queue*) – Message queue to write.
- **read_queue** (*torch.multiprocessing.Queue*) – Message queue to read.
- **logger** (*Logger, optional*) – object of `Logger`.

run(*self*)

Main Process:

1. Initialization stage.
2. FL communication stage.
3. Shutdown stage. Close network connection.

setup(*self*, **args*, ***kwargs*)

Initialize network connection and necessary setups.

At first, `self._network.init_network_connection()` is required to be called.

Overwrite this method to implement system setup message communication procedure.

main_loop(*self*, **args*, ***kwargs*)

Define the actions of communication stage.

process_meessage_queue(*self*)

client -> server directly transport.

class Scheduler(*net_upper*, *net_lower*)

Middle Topology for hierarchical communication pattern.

Scheduler uses message queues to decouple connector modules.

Parameters

- **net_upper** (DistNetwork) – Distributed network manager of server from upper level.
- **net_lower** (DistNetwork) – Distributed network manager of clients from lower level.

run(*self*)

handler

Module Contents

ParameterServerBackendHandler	An abstract class representing handler of parameter server.
SyncParameterServerHandler	Synchronous Parameter Server Handler.
AsyncParameterServerHandler	Asynchronous Parameter Server Handler

class ParameterServerBackendHandler(*model*, *cuda=False*)

Bases: *fedlab.core.model_maintainer.ModelMaintainer*

An abstract class representing handler of parameter server.

Please make sure that your self-defined server handler class subclasses this class

Example

Read source code of *SyncParameterServerHandler* and *AsyncParameterServerHandler*.

property downlink_package(*self*) → List[torch.Tensor]

Property for manager layer. Server manager will call this property when activates clients.

property if_stop(*self*) → bool

`NetworkManager` keeps monitoring this attribute, and it will stop all related processes and threads when `True` returned.

abstract _update_global_model(*self*, *payload*)

Override this function to define how to update global model (aggregation or optimization).

class SyncParameterServerHandler(*model*, *global_round*, *sample_ratio*, *cuda=False*, *logger=None*)

Bases: *ParameterServerBackendHandler*

Synchronous Parameter Server Handler.

Backend of synchronous parameter server: this class is responsible for backend computing in synchronous server.

Synchronous parameter server will wait for every client to finish local training process before the next FL round.

Details in paper: http://proceedings.mlr.press/v54/mcmahan17a.html

Parameters

- **model** (*torch.nn.Module*) – Model used in this federation.
- **global_round** (*int*) – stop condition. Shut down FL system when global round is reached.
- **sample_ratio** (*float*) – The result of `sample_ratio * client_num` is the number of clients for every FL round.
- **cuda** (*bool*) – Use GPUs or not. Default: `False`.

- **logger** (Logger, *optional*) – object of Logger.

property downlink_package(*self*)

Property for manager layer. Server manager will call this property when activates clients.

property if_stop(*self*)

NetworkManager keeps monitoring this attribute, and it will stop all related processes and threads when True returned.

property client_num_per_round(*self*)

sample_clients(*self*)

Return a list of client rank indices selected randomly. The client ID is from 1 to self.client_num_in_total + 1.

_update_global_model(*self*, *payload*)

Update global model with collected parameters from clients.

Note: Server handler will call this method when its client_buffer_cache is full. User can overwrite the strategy of aggregation to apply on model_parameters_list, and use SerializationTool.deserialize_model() to load serialized parameters after aggregation into self._model.

Parameters **payload** (*list[torch.Tensor]*) – A list of tensors passed by manager layer.

class AsyncParameterServerHandler(*model*, *alpha*, *total_time*, *strategy='constant'*, *cuda=False*, *logger=None*)

Bases: *ParameterServerBackendHandler*

Asynchronous Parameter Server Handler

Update global model immediately after receiving a ParameterUpdate message Paper: https://arxiv.org/abs/1903.03934

Parameters

- **model** (*torch.nn.Module*) – Global model in server
- **alpha** (*float*) – Weight used in async aggregation.
- **total_time** (*int*) – Stop condition. Shut down FL system when total_time is reached.
- **strategy** (*str*) – Adaptive strategy. constant, hinge and polynomial is optional. Default: constant.
- **cuda** (*bool*) – Use GPUs or not.
- **logger** (Logger, *optional*) – Object of Logger.

property if_stop(*self*)

NetworkManager keeps monitoring this attribute, and it will stop all related processes and threads when True returned.

property downlink_package(*self*)

Property for manager layer. Server manager will call this property when activates clients.

_update_global_model(*self*, *payload*)

Override this function to define how to update global model (aggregation or optimization).

_adapt_alpha(*self*, *receive_model_time*)

update the alpha according to staleness

manager

Module Contents

ServerManager	Base class of ServerManager.
SynchronousServerManager	Synchronous communication
AsynchronousServerManager	Asynchronous communication network manager for server

DEFAULT_SERVER_RANK

DEFAULT_SERVER_RANK = 0

class ServerManager(*network*, *handler*)

Bases: *fedlab.core.network_manager.NetworkManager*

Base class of ServerManager.

Parameters

- **network** (DistNetwork) – Network configuration and interfaces.
- **handler** (ParameterServerBackendHandler) – Performe global model update procedure.

setup(*self*)

Initialization Stage.

- Server accept local client num report from client manager.
- Init a coordinator for client_id -> rank mapping.

class SynchronousServerManager(*network*, *handler*, *logger=None*)

Bases: *ServerManager*

Synchronous communication

This is the top class in our framework which is mainly responsible for network communication of SERVER!. Synchronously communicate with clients following agreements defined in *main_loop()*.

Parameters

- **network** (DistNetwork) – Network configuration and interfaces.
- **handler** (ParameterServerBackendHandler) – Backend calculation handler for parameter server.
- **logger** (Logger, *optional*) – Object of Logger.

setup(*self*)

Initialization Stage.

- Server accept local client num report from client manager.

- Init a coordinator for client_id -> rank mapping.

main_loop(*self*)

Actions to perform in server when receiving a package from one client.

Server transmits received package to backend computation handler for aggregation or others manipulations.

Loop:

1. activate clients for current training round.
2. listen for message from clients -> transmit received parameters to server backend.

Note: Communication agreements related: user can overwrite this function to customize communication agreements. This method is key component connecting behaviors of `ParameterServerBackendHandler` and `NetworkManager`.

Raises `Exception` – Unexpected `MessageCode`.

shutdown(*self*)

Shutdown stage.

activate_clients(*self*)

Activate subset of clients to join in one FL round

Manager will start a new thread to send activation package to chosen clients' process rank. The id of clients are obtained from `handler.sample_clients()`. And their communication ranks are are obtained via coordinator.

shutdown_clients(*self*)

Shutdown all clients.

Send package to each client with `MessageCode.Exit`.

Note: Communication agreements related: User can overwrite this function to define package for exiting information.

---

class AsynchronousServerManager(*network*, *handler*, *logger=None*)

Bases: *ServerManager*

Asynchronous communication network manager for server

This is the top class in our framework which is mainly responsible for network communication of SERVER!. Asynchronously communicate with clients following agreements defined in `mail_loop()`.

Parameters

- **network** (`DistNetwork`) – Network configuration and interfaces.
- **handler** (`ParameterServerBackendHandler`) – Backend computation handler for parameter server.
- **logger** (`Logger`, *optional*) – Object of `Logger`.

setup(*self*)

Initialization Stage.

- Server accept local client num report from client manager.

- Init a coordinator for client_id -> rank mapping.

main_loop(*self*)

Communication agreements of asynchronous FL.

- Server receive ParameterRequest from client. Send model parameter to client.
- Server receive ParameterUpdate from client. Transmit parameters to queue waiting for aggregation.

Raises `ValueError` – invalid message code.

shutdown(*self*)

Shutdown stage.

Close the network connection in the end.

updater_thread(*self*)

Asynchronous communication maintain a message queue. A new thread will be started to keep monitoring message queue.

shutdown_clients(*self*)

Shutdown all clients.

Send package to clients with `MessageCode.Exit`.

Package Contents

SyncParameterServerHandler	Synchronous Parameter Server Handler.
AsyncParameterServerHandler	Asynchronous Parameter Server Handler
SynchronousServerManager	Synchronous communication
AsynchronousServerManager	Asynchronous communication network manager for server

class SyncParameterServerHandler(*model*, *global_round*, *sample_ratio*, *cuda=False*, *logger=None*)

Bases: `ParameterServerBackendHandler`

Synchronous Parameter Server Handler.

Backend of synchronous parameter server: this class is responsible for backend computing in synchronous server.

Synchronous parameter server will wait for every client to finish local training process before the next FL round.

Details in paper: http://proceedings.mlr.press/v54/mcmahan17a.html

Parameters

- **model** (*torch.nn.Module*) – Model used in this federation.
- **global_round** (*int*) – stop condition. Shut down FL system when global round is reached.
- **sample_ratio** (*float*) – The result of `sample_ratio * client_num` is the number of clients for every FL round.
- **cuda** (*bool*) – Use GPUs or not. Default: `False`.
- **logger** (`Logger`, *optional*) – object of `Logger`.

property downlink_package(*self*)

Property for manager layer. Server manager will call this property when activates clients.

property if_stop(*self*)

NetworkManager keeps monitoring this attribute, and it will stop all related processes and threads when True returned.

property client_num_per_round(*self*)

sample_clients(*self*)

Return a list of client rank indices selected randomly. The client ID is from 1 to self.client_num_in_total + 1.

_update_global_model(*self*, *payload*)

Update global model with collected parameters from clients.

Note: Server handler will call this method when its client_buffer_cache is full. User can overwrite the strategy of aggregation to apply on model_parameters_list, and use SerializationTool.deserialize_model() to load serialized parameters after aggregation into self._model.

Parameters **payload** (*list[torch.Tensor]*) – A list of tensors passed by manager layer.

class AsyncParameterServerHandler(*model*, *alpha*, *total_time*, *strategy='constant'*, *cuda=False*, *logger=None*)

Bases: ParameterServerBackendHandler

Asynchronous Parameter Server Handler

Update global model immediately after receiving a ParameterUpdate message Paper: https://arxiv.org/abs/1903.03934

Parameters

- **model** (*torch.nn.Module*) – Global model in server
- **alpha** (*float*) – Weight used in async aggregation.
- **total_time** (*int*) – Stop condition. Shut down FL system when total_time is reached.
- **strategy** (*str*) – Adaptive strategy. constant, hinge and polynomial is optional. Default: constant.
- **cuda** (*bool*) – Use GPUs or not.
- **logger** (Logger, *optional*) – Object of Logger.

property if_stop(*self*)

NetworkManager keeps monitoring this attribute, and it will stop all related processes and threads when True returned.

property downlink_package(*self*)

Property for manager layer. Server manager will call this property when activates clients.

_update_global_model(*self*, *payload*)

Override this function to define how to update global model (aggregation or optimization).

_adapt_alpha(*self*, *receive_model_time*)

update the alpha according to staleness

class SynchronousServerManager(*network*, *handler*, *logger=None*)

Bases: ServerManager

Synchronous communication

This is the top class in our framework which is mainly responsible for network communication of SERVER!. Synchronously communicate with clients following agreements defined in *main_loop()*.

Parameters

- **network** (DistNetwork) – Network configuration and interfaces.
- **handler** (ParameterServerBackendHandler) – Backend calculation handler for parameter server.
- **logger** (Logger, *optional*) – Object of Logger.

setup(*self*)

Initialization Stage.

- Server accept local client num report from client manager.
- Init a coordinator for client_id -> rank mapping.

main_loop(*self*)

Actions to perform in server when receiving a package from one client.

Server transmits received package to backend computation handler for aggregation or others manipulations.

Loop:

1. activate clients for current training round.
2. listen for message from clients -> transmit received parameters to server backend.

Note: Communication agreements related: user can overwrite this function to customize communication agreements. This method is key component connecting behaviors of ParameterServerBackendHandler and NetworkManager.

Raises Exception – Unexpected MessageCode.

shutdown(*self*)

Shutdown stage.

activate_clients(*self*)

Activate subset of clients to join in one FL round

Manager will start a new thread to send activation package to chosen clients' process rank. The id of clients are obtained from handler.sample_clients(). And their communication ranks are are obtained via coordinator.

shutdown_clients(*self*)

Shutdown all clients.

Send package to each client with MessageCode.Exit.

Note: Communication agreements related: User can overwrite this function to define package for exiting information.

---

class AsynchronousServerManager(*network*, *handler*, *logger=None*)

Bases: `ServerManager`

Asynchronous communication network manager for server

This is the top class in our framework which is mainly responsible for network communication of SERVER!. Asynchronously communicate with clients following agreements defined in `mail_loop()`.

Parameters

- **network** (`DistNetwork`) – Network configuration and interfaces.
- **handler** (`ParameterServerBackendHandler`) – Backend computation handler for parameter server.
- **logger** (`Logger, optional`) – Object of `Logger`.

setup(*self*)

Initialization Stage.

- Server accept local client num report from client manager.
- Init a coordinator for client_id -> rank mapping.

main_loop(*self*)

Communication agreements of asynchronous FL.

- Server receive ParameterRequest from client. Send model parameter to client.
- Server receive ParameterUpdate from client. Transmit parameters to queue waiting for aggregation.

Raises `ValueError` – invalid message code.

shutdown(*self*)

Shutdown stage.

Close the network connection in the end.

updater_thread(*self*)

Asynchronous communication maintain a message queue. A new thread will be started to keep monitoring message queue.

shutdown_clients(*self*)

Shutdown all clients.

Send package to clients with `MessageCode.Exit`.

coordinator

Module Contents

Coordinator	Deal with the mapping relation between client id and process rank in FL system.

class Coordinator(*setup_dict*, *mode='LOCAL'*)

Bases: `object`

Deal with the mapping relation between client id and process rank in FL system.

Note Server Manager creates a Coordinator following: 1. init network connection. 2. client send local group info (the number of client simulating in local) to server. 4. server receive all info and init a server Coordinator.

Parameters

- **setup_dict** (*dict*) – A dict like {rank:client_num ...}, representing the map relation between process rank and client id.
- **mode** (*str, optional*) – "GLOBAL" and "LOCAL". Coordinator will map client id to (rank, global id) or (rank, local id) according to mode. For example, client id 51 is in a machine which has 1 manager and serial trainer simulating 10 clients. LOCAL id means the index of its 10 clients. Therefore, global id 51 will be mapped into local id 1 (depending on setting).

map_id(*self*, *id*)

a map function from client id to (rank,local id)

Parameters **id** (*int*) – client id

Returns rank in distributed group and local id.

Return type rank, id

map_id_list(*self*, *id_list*)

a map function from id_list to dict{rank:local id}

This can be very useful in Scale modules.

Parameters **id_list** (*list(int)*) – a list of client id.

Returns contains process rank and its relative local client ids.

Return type map_dict (dict)

switch(*self*)

property total(*self*)

__str__(*self*) → str

Return str(self).

__call__(*self*, *info*, **args*, ***kwds*)

model_maintainer

Module Contents

ModelMaintainer	Maintain PyTorch model.

class ModelMaintainer(*model*, *cuda*)

Bases: object

Maintain PyTorch model.

Provide necessary attributes and operation methods. More features with local or global model will be implemented here.

Parameters

- **model** (*torch.nn.Module*) – PyTorch model.
- **cuda** (*bool*) – use GPUs or not.

property model(*self*)

Return `torch.nn.module`

property model_parameters(*self*)

Return serialized model parameters.

property model_gradients(*self*)

Return serialized model gradients.

property shape_list(*self*)

Return shape of model parameters.

Currently, this attributes used in tensor compression.

network

Module Contents

DistNetwork	Manage `torch.distributed` network.

type2byte

type2byte

class DistNetwork(*address*, *world_size*, *rank*, *ethernet=None*, *dist_backend='gloo'*)

Bases: `object`

Manage `torch.distributed` network.

Parameters

- **address** (*tuple*) – Address of this server in form of `(SERVER_ADDR, SERVER_IP)`
- **world_size** (*int*) – the size of this distributed group (including server).
- **rank** (*int*) – the rank of process in distributed group.
- **ethernet** (*str*) –
- **dist_backend** (*str or torch.distributed.Backend*) – backend of `torch.distributed`. Valid values include `mpi`, `gloo`, and `nccl`. Default: `"gloo"`.

init_network_connection(*self*)

Initialize `torch.distributed` communication group

close_network_connection(*self*)

Destroy current `torch.distributed` process group

send(*self*, *content=None*, *message_code=None*, *dst=0*, *count=True*)

Send tensor to process rank=dst

recv(*self*, *src=None*, *count=True*)

Receive tensor from process rank=src

__str__(*self*)

Return str(self).

network_manager

Module Contents

NetworkManager	Abstract class

class NetworkManager(*network*)

Bases: `torch.multiprocessing.Process`

Abstract class

Parameters **network** (`DistNetwork`) – object to manage torch.distributed network communication.

run(*self*)

Main Process:

1. Initialization stage.
2. FL communication stage.
3. Shutdown stage. Close network connection.

setup(*self*, **args*, ***kwargs*)

Initialize network connection and necessary setups.

At first, `self._network.init_network_connection()` is required to be called.

Overwrite this method to implement system setup message communication procedure.

abstract main_loop(*self*, **args*, ***kwargs*)

Define the actions of communication stage.

shutdown(*self*, **args*, ***kwargs*)

Shutdown stage.

Close the network connection in the end.

Package Contents

DistNetwork	Manage `torch.distributed` network.
NetworkManager	Abstract class

class DistNetwork(*address*, *world_size*, *rank*, *ethernet=None*, *dist_backend='gloo'*)

Bases: `object`

Manage `torch.distributed` network.

Parameters

- **address** (*tuple*) – Address of this server in form of `(SERVER_ADDR, SERVER_IP)`

- **world_size** (*int*) – the size of this distributed group (including server).
- **rank** (*int*) – the rank of process in distributed group.
- **ethernet** (*str*) –
- **dist_backend** (*str or torch.distributed.Backend*) – backend of `torch.distributed`. Valid values include `mpi`, `gloo`, and `nccl`. Default: `"gloo"`.

init_network_connection(*self*)

Initialize `torch.distributed` communication group

close_network_connection(*self*)

Destroy current `torch.distributed` process group

send(*self*, *content=None*, *message_code=None*, *dst=0*, *count=True*)

Send tensor to process rank=dst

recv(*self*, *src=None*, *count=True*)

Receive tensor from process rank=src

__str__(*self*)

Return str(self).

class NetworkManager(*network*)

Bases: `torch.multiprocessing.Process`

Abstract class

Parameters **network** (`DistNetwork`) – object to manage torch.distributed network communication.

run(*self*)

Main Process:

1. Initialization stage.
2. FL communication stage.
3. Shutdown stage. Close network connection.

setup(*self*, **args*, ***kwargs*)

Initialize network connection and necessary setups.

At first, `self._network.init_network_connection()` is required to be called.

Overwrite this method to implement system setup message communication procedure.

abstract main_loop(*self*, **args*, ***kwargs*)

Define the actions of communication stage.

shutdown(*self*, **args*, ***kwargs*)

Shutdown stage.

Close the network connection in the end.

10.1.2 utils

dataset

functional

Module Contents

split_indices(num_cumsum, rand_perm)	Splice the sample index list given number of each client.
balance_split(num_clients, num_samples)	Assign same sample sample for each client.
lognormal_unbalance_split(num_clients, num_samples, unbalance_sgm)	Assign different sample number for each client using Log-Normal distribution.
dirichlet_unbalance_split(num_clients, num_samples, alpha)	Assign different sample number for each client using Log-Normal distribution.
homo_partition(client_sample_nums, num_samples)	Partition data indices in IID way given sample numbers for each clients.
hetero_dir_partition(targets, num_clients, num_classes, dir_alpha, min_require_size=None)	Non-iid partition based on Dirichlet distribution. The method is from "hetero-dir" partition of
shards_partition(targets, num_clients, num_shards)	Non-iid partition used in FedAvg paper.
client_inner_dirichlet_partition(targets, num_clients, num_classes, dir_alpha, client_sample_nums, verbose=True)	Non-iid Dirichlet partition.
label_skew_quantity_based_partition(targets, num_clients, num_classes, major_classes_num)	Label-skew:quantity-based partition.
fcube_synthetic_partition(data)	Feature-distribution-skew:synthetic partition.
samples_num_count(client_dict, num_clients)	Return sample count for all clients in `client_dict`.

split_indices(*num_cumsum*, *rand_perm*)

Splice the sample index list given number of each client.

Parameters

- **num_cumsum** (*np.ndarray*) – Cumulative sum of sample number for each client.
- **rand_perm** (*list*) – List of random sample index.

Returns `{ client_id: indices}`.

Return type dict

balance_split(*num_clients*, *num_samples*)

Assign same sample sample for each client.

Parameters

- **num_clients** (*int*) – Number of clients for partition.
- **num_samples** (*int*) – Total number of samples.

Returns A numpy array consisting `num_clients` integer elements, each represents sample number of corresponding clients.

Return type numpy.ndarray

lognormal_unbalance_split(*num_clients*, *num_samples*, *unbalance_sgm*)

Assign different sample number for each client using Log-Normal distribution.

Sample numbers for clients are drawn from Log-Normal distribution.

Parameters

- **num_clients** (*int*) – Number of clients for partition.
- **num_samples** (*int*) – Total number of samples.
- **unbalance_sgm** (*float*) – Log-normal variance. When equals to `0`, the partition is equal to `balance_partition()`.

Returns A numpy array consisting `num_clients` integer elements, each represents sample number of corresponding clients.

Return type numpy.ndarray

dirichlet_unbalance_split(*num_clients*, *num_samples*, *alpha*)

Assign different sample number for each client using Log-Normal distribution.

Sample numbers for clients are drawn from Log-Normal distribution.

Parameters

- **num_clients** (*int*) – Number of clients for partition.
- **num_samples** (*int*) – Total number of samples.
- **alpha** (*float*) – Dirichlet concentration parameter

Returns A numpy array consisting `num_clients` integer elements, each represents sample number of corresponding clients.

Return type numpy.ndarray

homo_partition(*client_sample_nums*, *num_samples*)

Partition data indices in IID way given sample numbers for each clients.

Parameters

- **client_sample_nums** (*numpy.ndarray*) – Sample numbers for each clients.
- **num_samples** (*int*) – Number of samples.

Returns `{ client_id: indices}`.

Return type dict

hetero_dir_partition(*targets*, *num_clients*, *num_classes*, *dir_alpha*, *min_require_size=None*)

Non-iid partition based on Dirichlet distribution. The method is from "hetero-dir" partition of Bayesian Nonparametric Federated Learning of Neural Networks and Federated Learning with Matched Averaging.

This method simulates heterogeneous partition for which number of data points and class proportions are unbalanced. Samples will be partitioned into J clients by sampling $p_k \sim \text{Dir}_J(\alpha)$ and allocating a $p_{p,j}$ proportion of the samples of class k to local client j.

Sample number for each client is decided in this function.

Parameters

- **targets** (*list or numpy.ndarray*) – Sample targets. Unshuffled preferred.
- **num_clients** (*int*) – Number of clients for partition.
- **num_classes** (*int*) – Number of classes in samples.

- **dir_alpha** (*float*) – Parameter alpha for Dirichlet distribution.
- **min_require_size** (*int, optional*) – Minimum required sample number for each client. If set to None, then equals to num_classes.

Returns { client_id: indices}.

Return type dict

shards_partition(*targets*, *num_clients*, *num_shards*)

Non-iid partition used in FedAvg paper.

Parameters

- **targets** (*list or numpy.ndarray*) – Sample targets. Unshuffled preferred.
- **num_clients** (*int*) – Number of clients for partition.
- **num_shards** (*int*) – Number of shards in partition.

Returns { client_id: indices}.

Return type dict

client_inner_dirichlet_partition(*targets*, *num_clients*, *num_classes*, *dir_alpha*, *client_sample_nums*, *verbose=True*)

Non-iid Dirichlet partition.

The method is from The method is from paper Federated Learning Based on Dynamic Regularization. This function can be used by given specific sample number for all clients client_sample_nums. It's different from *hetero_dir_partition()*.

Parameters

- **targets** (*list or numpy.ndarray*) – Sample targets.
- **num_clients** (*int*) – Number of clients for partition.
- **num_classes** (*int*) – Number of classes in samples.
- **dir_alpha** (*float*) – Parameter alpha for Dirichlet distribution.
- **client_sample_nums** (*numpy.ndarray*) – A numpy array consisting num_clients integer elements, each represents sample number of corresponding clients.
- **verbose** (*bool, optional*) – Whether to print partition process. Default as True.

Returns { client_id: indices}.

Return type dict

label_skew_quantity_based_partition(*targets*, *num_clients*, *num_classes*, *major_classes_num*)

Label-skew:quantity-based partition.

For details, please check Federated Learning on Non-IID Data Silos: An Experimental Study.

Parameters

- **targets** (*np.ndarray*) – Labels od dataset.
- **num_clients** (*int*) – Number of clients.
- **num_classes** (*int*) – Number of unique classes.
- **major_classes_num** (*int*) – Number of classes for each client, should be less then num_classes.

Returns { client_id: indices}.

Return type dict

fcube_synthetic_partition(*data*)

Feature-distribution-skew:synthetic partition.

Synthetic partition for FCUBE dataset. This partition is from Federated Learning on Non-IID Data Silos: An Experimental Study.

Parameters **data** (*np.ndarray*) – Data of dataset FCUBE.

Returns { client_id: indices}.

Return type dict

samples_num_count(*client_dict*, *num_clients*)

Return sample count for all clients in client_dict.

Parameters

- **client_dict** (*dict*) – Data partition result for different clients.
- **num_clients** (*int*) – Total number of clients.

Returns pandas.DataFrame

partition

Module Contents

DataPartitioner	Base class for data partition in federated learning.
CIFAR10Partitioner	CIFAR10 data partitioner.
CIFAR100Partitioner	CIFAR100 data partitioner.
BasicPartitioner	Basic data partitioner.
VisionPartitioner	Data partitioner for vision data.
MNISTPartitioner	Data partitioner for MNIST.
FMNISTPartitioner	Data partitioner for FashionMNIST.
SVHNPartitioner	Data partitioner for SVHN.
FCUBEPartitioner	FCUBE data partitioner.
AdultPartitioner	Data partitioner for Adult.
RCV1Partitioner	Data partitioner for RCV1.
CovtypePartitioner	Data partitioner for Covtype.

class DataPartitioner

Bases: abc.ABC

Base class for data partition in federated learning.

Examples of *DataPartitioner*: *BasicPartitioner*, *CIFAR10Partitioner*.

Details and tutorials of different data partition and datasets, please check Federated Dataset and DataPartitioner.

abstract _perform_partition(*self*)

abstract __getitem__(*self*, *index*)

abstract __len__(*self*)

class CIFAR10Partitioner(*targets*, *num_clients*, *balance=True*, *partition='iid'*, *unbalance_sgm=0*, *num_shards=None*, *dir_alpha=None*, *verbose=True*, *seed=None*)

Bases: *DataPartitioner*

CIFAR10 data partitioner.

Partition CIFAR10 given specific client number. Currently 6 supported partition schemes can be achieved by passing different combination of parameters in initialization:

- `balance=None`
  - `partition="dirichlet"`: non-iid partition used in Bayesian Nonparametric Federated Learning of Neural Networks and Federated Learning with Matched Averaging. Refer to *fedlab.utils.dataset.functional.hetero_dir_partition()* for more information.
  - `partition="shards"`: non-iid method used in FedAvg paper. Refer to *fedlab.utils.dataset.functional.shards_partition()* for more information.
- `balance=True`: "Balance" refers to FL scenario that sample numbers for different clients are the same. Refer to `fedlab.utils.dataset.functional.balance_partition()` for more information.
  - `partition="iid"`: Random select samples from complete dataset given sample number for each client.
  - `partition="dirichlet"`: Refer to *fedlab.utils.dataset.functional.client_inner_dirichlet_partition()* for more information.
- `balance=False`: "Unbalance" refers to FL scenario that sample numbers for different clients are different. For unbalance method, sample number for each client is drown from Log-Normal distribution with variance `unbalanced_sgm`. When `unbalanced_sgm=0`, partition is balanced. Refer to `fedlab.utils.dataset.functional.lognormal_unbalance_partition()` for more information. The method is from paper Federated Learning Based on Dynamic Regularization.
  - `partition="iid"`: Random select samples from complete dataset given sample number for each client.
  - `partition="dirichlet"`: Refer to *fedlab.utils.dataset.functional.client_inner_dirichlet_partition()* for more information.

For detail usage, please check Federated Dataset and DataPartitioner.

Parameters

- **targets** (*list or numpy.ndarray*) – Targets of dataset for partition. Each element is in range of [0, 1, ..., 9].
- **num_clients** (*int*) – Number of clients for data partition.
- **balance** (*bool, optional*) – Balanced partition over all clients or not. Default as `True`.
- **partition** (*str, optional*) – Partition type, only `"iid"`, `shards`, `"dirichlet"` are supported. Default as `"iid"`.
- **unbalance_sgm** (*float, optional*) – Log-normal distribution variance for unbalanced data partition over clients. Default as `0` for balanced partition.
- **num_shards** (*int, optional*) – Number of shards in non-iid `"shards"` partition. Only works if `partition="shards"`. Default as `None`.
- **dir_alpha** (*float, optional*) – Dirichlet distribution parameter for non-iid partition. Only works if `partition="dirichlet"`. Default as `None`.
- **verbose** (*bool, optional*) – Whether to print partition process. Default as `True`.
- **seed** (*int, optional*) – Random seed. Default as `None`.

Returns { client_id: indices}.

Return type dict

num_classes = 10

_perform_partition(*self*)

__getitem__(*self*, *index*)

Obtain sample indices for client `index`.

Parameters **index** (*int*) – Client ID.

Returns List of sample indices for client ID `index`.

Return type list

__len__(*self*)

Usually equals to number of clients.

class CIFAR100Partitioner(*targets*, *num_clients*, *balance=True*, *partition='iid'*, *unbalance_sgm=0*, *num_shards=None*, *dir_alpha=None*, *verbose=True*, *seed=None*)

Bases: *CIFAR10Partitioner*

CIFAR100 data partitioner.

This is a subclass of the *CIFAR10Partitioner*. For details, please check Federated Dataset and DataPartitioner.

num_classes = 100

class BasicPartitioner(*targets*, *num_clients*, *partition='iid'*, *dir_alpha=None*, *major_classes_num=1*, *verbose=True*, *seed=None*)

Bases: *DataPartitioner*

Basic data partitioner.

Basic data partitioner, supported partition:

- label-distribution-skew:quantity-based
- label-distribution-skew:distributed-based (Dirichlet)
- quantity-skew (Dirichlet)
- IID

For more details, please check Federated Learning on Non-IID Data Silos: An Experimental Study and Federated Dataset and DataPartitioner.

Parameters

- **targets** (*list or numpy.ndarray*) – Sample targets. Unshuffled preferred.
- **num_clients** (*int*) – Number of clients for partition.
- **partition** (*str*) – Partition name. Only supports `"noniid-#label"`, `"noniid-labeldir"`, `"unbalance"` and `"iid"` partition schemes.
- **dir_alpha** (*float*) – Parameter alpha for Dirichlet distribution. Only works if `partition="noniid-labeldir"`.
- **major_classes_num** (*int*) – Number of major class for each clients. Only works if `partition="noniid-#label"`.
- **verbose** (*bool*) – Whether output intermediate information. Default as `True`.

- **seed** (*int*) – Random seed. Default as None.

Returns { client_id: indices}.

Return type dict

num_classes = 2

_perform_partition(*self*)

__getitem__(*self*, *index*)

__len__(*self*)

class VisionPartitioner(*targets*, *num_clients*, *partition='iid'*, *dir_alpha=None*, *major_classes_num=None*, *verbose=True*, *seed=None*)

Bases: *BasicPartitioner*

Data partitioner for vision data.

Supported partition for vision data:

- label-distribution-skew:quantity-based
- label-distribution-skew:distributed-based (Dirichlet)
- quantity-skew (Dirichlet)
- IID

For more details, please check Federated Learning on Non-IID Data Silos: An Experimental Study.

Parameters

- **targets** (*list or numpy.ndarray*) – Sample targets. Unshuffled preferred.
- **num_clients** (*int*) – Number of clients for partition.
- **partition** (*str*) – Partition name. Only supports "noniid-#label", "noniid-labeldir", "unbalance" and "iid" partition schemes.
- **dir_alpha** (*float*) – Parameter alpha for Dirichlet distribution. Only works if partition="noniid-labeldir".
- **major_classes_num** (*int*) – Number of major class for each clients. Only works if partition="noniid-#label".
- **verbose** (*bool*) – Whether output intermediate information. Default as True.
- **seed** (*int*) – Random seed. Default as None.

Returns { client_id: indices}.

Return type dict

num_classes = 10

class MNISTPartitioner(*targets*, *num_clients*, *partition='iid'*, *dir_alpha=None*, *major_classes_num=None*, *verbose=True*, *seed=None*)

Bases: *VisionPartitioner*

Data partitioner for MNIST.

For details, please check *VisionPartitioner* and Federated Dataset and DataPartitioner.

num_features = 784

class FMNISTPartitioner(*targets*, *num_clients*, *partition='iid'*, *dir_alpha=None*, *major_classes_num=None*, *verbose=True*, *seed=None*)

Bases: *VisionPartitioner*

Data partitioner for FashionMNIST.

For details, please check *VisionPartitioner* and Federated Dataset and DataPartitioner

num_features = 784

class SVHNPartitioner(*targets*, *num_clients*, *partition='iid'*, *dir_alpha=None*, *major_classes_num=None*, *verbose=True*, *seed=None*)

Bases: *VisionPartitioner*

Data partitioner for SVHN.

For details, please check *VisionPartitioner* and Federated Dataset and DataPartitioner

num_features = 1024

class FCUBEPartitioner(*data*, *partition*)

Bases: *DataPartitioner*

FCUBE data partitioner.

FCUBE is a synthetic dataset for research in non-IID scenario with feature imbalance. This dataset and its partition methods are proposed in Federated Learning on Non-IID Data Silos: An Experimental Study.

Supported partition methods for FCUBE:

- feature-distribution-skew:synthetic
- IID

For more details, please refer to Section (IV-B-b) of original paper. For detailed usage, please check Federated Dataset and DataPartitioner.

Parameters

- **data** (*numpy.ndarray*) – Data of dataset FCUBE.
- **partition** (*str*) – Partition type. Only supports *'synthetic'* and *'iid'*.

num_classes = 2

num_clients = 4

_perform_partition(*self*)

__getitem__(*self*, *index*)

__len__(*self*)

class AdultPartitioner(*targets*, *num_clients*, *partition='iid'*, *dir_alpha=None*, *major_classes_num=1*, *verbose=True*, *seed=None*)

Bases: *BasicPartitioner*

Data partitioner for Adult.

For details, please check *BasicPartitioner* and Federated Dataset and DataPartitioner

num_features = 123

num_classes = 2

class RCV1Partitioner(*targets*, *num_clients*, *partition='iid'*, *dir_alpha=None*, *major_classes_num=1*, *verbose=True*, *seed=None*)

Bases: *BasicPartitioner*

Data partitioner for RCV1.

For details, please check *BasicPartitioner* and Federated Dataset and DataPartitioner

num_features = 47236

num_classes = 2

class CovtypePartitioner(*targets*, *num_clients*, *partition='iid'*, *dir_alpha=None*, *major_classes_num=1*, *verbose=True*, *seed=None*)

Bases: *BasicPartitioner*

Data partitioner for Covtype.

For details, please check *BasicPartitioner* and Federated Dataset and DataPartitioner

num_features = 54

num_classes = 2

sampler

Module Contents

SubsetSampler	Samples elements from a given list of indices, always in the same order once initialized.
RawPartitionSampler	Partition dataset according to `num_replicas`.
DictFileSampler	Get data sample indices given client id from data file with dict.

class SubsetSampler(*indices*, *shuffle=False*)

Bases: `torch.utils.data.Sampler`

Samples elements from a given list of indices, always in the same order once initialized.

It is a `Sampler` used in `Dataloader`, that each partition will be fixed once initialized.

Parameters

- **indices** (*list[int]*) – Indices in the whole set selected for subset
- **shuffle** (*bool*) – shuffle the indices or not.

__iter__(*self*)

__len__(*self*)

class RawPartitionSampler(*dataset*, *client_id*, *num_replicas=None*)

Bases: `torch.utils.data.Sampler`

Partition dataset according to `num_replicas`.

Every client get a equal shard of dataset.

Parameters

- **dataset** (*torch.utils.data.Dataset*) –
- **client_id** (*int*) –
- **num_replicas** (*int, optional*) – Number of data replications. Default None means total number of client processes.

__iter__(*self*)

__len__(*self*)

class DictFileSampler(*dict_file*, *client_id*)

Bases: torch.utils.data.Sampler

Get data sample indices given client id from data file with dict.

__iter__(*self*)

__len__(*self*)

slicing

functions associated with data and dataset slicing

Module Contents

noniid_slicing(dataset, num_clients, num_shards)	Slice a dataset for non-IID.
random_slicing(dataset, num_clients)	Slice a dataset randomly and equally for IID.

noniid_slicing(*dataset*, *num_clients*, *num_shards*)

Slice a dataset for non-IID.

Parameters

- **dataset** (*torch.utils.data.Dataset*) – Dataset to slice.
- **num_clients** (*int*) – Number of client.
- **num_shards** (*int*) – Number of shards.

Notes

The size of a shard equals to `int(len(dataset)/num_shards)`. Each client will get `int(num_shards/num_clients)` shards.

Returns dict: `{ 0: indices of dataset, 1: indices of dataset, ..., k: indices of dataset }`

random_slicing(*dataset*, *num_clients*)

Slice a dataset randomly and equally for IID.

Args dataset (torch.utils.data.Dataset): a dataset for slicing. num_clients (int): the number of client.

Returns dict: `{ 0: indices of dataset, 1: indices of dataset, ..., k: indices of dataset }`

Package Contents

SubsetSampler	Samples elements from a given list of indices, always in the same order once initialized.
RawPartitionSampler	Partition dataset according to `num_replicas`.
DictFileSampler	Get data sample indices given client id from data file with dict.
DataPartitioner	Base class for data partition in federated learning.
BasicPartitioner	Basic data partitioner.
VisionPartitioner	Data partitioner for vision data.
CIFAR10Partitioner	CIFAR10 data partitioner.
CIFAR100Partitioner	CIFAR100 data partitioner.
FMNISTPartitioner	Data partitioner for FashionMNIST.
MNISTPartitioner	Data partitioner for MNIST.
SVHNPartitioner	Data partitioner for SVHN.
FCUBEPartitioner	FCUBE data partitioner.
AdultPartitioner	Data partitioner for Adult.
RCV1Partitioner	Data partitioner for RCV1.
CovtypePartitioner	Data partitioner for Covtype.

noniid_slicing(dataset, num_clients, num_shards)	Slice a dataset for non-IID.
random_slicing(dataset, num_clients)	Slice a dataset randomly and equally for IID.

class SubsetSampler(*indices*, *shuffle=False*)

Bases: `torch.utils.data.Sampler`

Samples elements from a given list of indices, always in the same order once initialized.

It is a `Sampler` used in `Dataloader`, that each partition will be fixed once initialized.

Parameters

- **indices** (*list[int]*) – Indices in the whole set selected for subset
- **shuffle** (*bool*) – shuffle the indices or not.

__iter__(*self*)

__len__(*self*)

class RawPartitionSampler(*dataset*, *client_id*, *num_replicas=None*)

Bases: `torch.utils.data.Sampler`

Partition dataset according to `num_replicas`.

Every client get a equal shard of dataset.

Parameters

- **dataset** (*torch.utils.data.Dataset*) –
- **client_id** (*int*) –
- **num_replicas** (*int, optional*) – Number of data replications. Default `None` means total number of client processes.

__iter__(*self*)

__len__(*self*)

class DictFileSampler(*dict_file*, *client_id*)

Bases: torch.utils.data.Sampler

Get data sample indices given client id from data file with dict.

__iter__(*self*)

__len__(*self*)

noniid_slicing(*dataset*, *num_clients*, *num_shards*)

Slice a dataset for non-IID.

Parameters

- **dataset** (*torch.utils.data.Dataset*) – Dataset to slice.
- **num_clients** (*int*) – Number of client.
- **num_shards** (*int*) – Number of shards.

Notes

The size of a shard equals to `int(len(dataset)/num_shards)`. Each client will get `int(num_shards/num_clients)` shards.

Returns dict: `{ 0: indices of dataset, 1: indices of dataset, ..., k: indices of dataset }`

random_slicing(*dataset*, *num_clients*)

Slice a dataset randomly and equally for IID.

Args dataset (torch.utils.data.Dataset): a dataset for slicing. num_clients (int): the number of client.

Returns dict: `{ 0: indices of dataset, 1: indices of dataset, ..., k: indices of dataset }`

class DataPartitioner

Bases: abc.ABC

Base class for data partition in federated learning.

Examples of *DataPartitioner*: *BasicPartitioner*, *CIFAR10Partitioner*.

Details and tutorials of different data partition and datasets, please check Federated Dataset and DataPartitioner.

abstract _perform_partition(*self*)

abstract __getitem__(*self*, *index*)

abstract __len__(*self*)

class BasicPartitioner(*targets*, *num_clients*, *partition='iid'*, *dir_alpha=None*, *major_classes_num=1*, *verbose=True*, *seed=None*)

Bases: *DataPartitioner*

Basic data partitioner.

Basic data partitioner, supported partition:

- label-distribution-skew:quantity-based

- label-distribution-skew:distributed-based (Dirichlet)
- quantity-skew (Dirichlet)
- IID

For more details, please check Federated Learning on Non-IID Data Silos: An Experimental Study and Federated Dataset and DataPartitioner.

Parameters

- **targets** (*list or numpy.ndarray*) – Sample targets. Unshuffled preferred.
- **num_clients** (*int*) – Number of clients for partition.
- **partition** (*str*) – Partition name. Only supports "noniid-#label", "noniid-labeldir", "unbalance" and "iid" partition schemes.
- **dir_alpha** (*float*) – Parameter alpha for Dirichlet distribution. Only works if partition="noniid-labeldir".
- **major_classes_num** (*int*) – Number of major class for each clients. Only works if partition="noniid-#label".
- **verbose** (*bool*) – Whether output intermediate information. Default as True.
- **seed** (*int*) – Random seed. Default as None.

Returns { client_id: indices}.

Return type dict

num_classes = 2

_perform_partition(*self*)

__getitem__(*self*, *index*)

__len__(*self*)

class VisionPartitioner(*targets*, *num_clients*, *partition='iid'*, *dir_alpha=None*, *major_classes_num=None*, *verbose=True*, *seed=None*)

Bases: *BasicPartitioner*

Data partitioner for vision data.

Supported partition for vision data:

- label-distribution-skew:quantity-based
- label-distribution-skew:distributed-based (Dirichlet)
- quantity-skew (Dirichlet)
- IID

For more details, please check Federated Learning on Non-IID Data Silos: An Experimental Study.

Parameters

- **targets** (*list or numpy.ndarray*) – Sample targets. Unshuffled preferred.
- **num_clients** (*int*) – Number of clients for partition.
- **partition** (*str*) – Partition name. Only supports "noniid-#label", "noniid-labeldir", "unbalance" and "iid" partition schemes.

- **dir_alpha** (*float*) – Parameter alpha for Dirichlet distribution. Only works if `partition="noniid-labeldir"`.
- **major_classes_num** (*int*) – Number of major class for each clients. Only works if `partition="noniid-#label"`.
- **verbose** (*bool*) – Whether output intermediate information. Default as `True`.
- **seed** (*int*) – Random seed. Default as `None`.

Returns `{ client_id: indices}`.

Return type dict

num_classes = 10

class CIFAR10Partitioner(*targets*, *num_clients*, *balance=True*, *partition='iid'*, *unbalance_sgm=0*, *num_shards=None*, *dir_alpha=None*, *verbose=True*, *seed=None*)

Bases: *DataPartitioner*

CIFAR10 data partitioner.

Partition CIFAR10 given specific client number. Currently 6 supported partition schemes can be achieved by passing different combination of parameters in initialization:

- `balance=None`
  - `partition="dirichlet"`: non-iid partition used in Bayesian Nonparametric Federated Learning of Neural Networks and Federated Learning with Matched Averaging. Refer to *fedlab.utils.dataset.functional.hetero_dir_partition()* for more information.
  - `partition="shards"`: non-iid method used in FedAvg paper. Refer to *fedlab.utils.dataset.functional.shards_partition()* for more information.
- `balance=True`: "Balance" refers to FL scenario that sample numbers for different clients are the same. Refer to `fedlab.utils.dataset.functional.balance_partition()` for more information.
  - `partition="iid"`: Random select samples from complete dataset given sample number for each client.
  - `partition="dirichlet"`: Refer to *fedlab.utils.dataset.functional.client_inner_dirichlet_partition()* for more information.
- `balance=False`: "Unbalance" refers to FL scenario that sample numbers for different clients are different. For unbalance method, sample number for each client is drown from Log-Normal distribution with variance `unbalanced_sgm`. When `unbalanced_sgm=0`, partition is balanced. Refer to `fedlab.utils.dataset.functional.lognormal_unbalance_partition()` for more information. The method is from paper Federated Learning Based on Dynamic Regularization.
  - `partition="iid"`: Random select samples from complete dataset given sample number for each client.
  - `partition="dirichlet"`: Refer to *fedlab.utils.dataset.functional.client_inner_dirichlet_partition()* for more information.

For detail usage, please check Federated Dataset and DataPartitioner.

Parameters

- **targets** (*list or numpy.ndarray*) – Targets of dataset for partition. Each element is in range of [0, 1, ..., 9].
- **num_clients** (*int*) – Number of clients for data partition.
- **balance** (*bool, optional*) – Balanced partition over all clients or not. Default as `True`.

- **partition** (*str, optional*) – Partition type, only "iid", shards, "dirichlet" are supported. Default as "iid".
- **unbalance_sgm** (*float, optional*) – Log-normal distribution variance for unbalanced data partition over clients. Default as 0 for balanced partition.
- **num_shards** (*int, optional*) – Number of shards in non-iid "shards" partition. Only works if partition="shards". Default as None.
- **dir_alpha** (*float, optional*) – Dirichlet distribution parameter for non-iid partition. Only works if partition="dirichlet". Default as None.
- **verbose** (*bool, optional*) – Whether to print partition process. Default as True.
- **seed** (*int, optional*) – Random seed. Default as None.

Returns { client_id: indices}.

Return type dict

num_classes = 10

_perform_partition(*self*)

__getitem__(*self*, *index*)

Obtain sample indices for client index.

Parameters **index** (*int*) – Client ID.

Returns List of sample indices for client ID index.

Return type list

__len__(*self*)

Usually equals to number of clients.

class CIFAR100Partitioner(*targets*, *num_clients*, *balance=True*, *partition='iid'*, *unbalance_sgm=0*, *num_shards=None*, *dir_alpha=None*, *verbose=True*, *seed=None*)

Bases: *CIFAR10Partitioner*

CIFAR100 data partitioner.

This is a subclass of the *CIFAR10Partitioner*. For details, please check Federated Dataset and DataPartitioner.

num_classes = 100

class FMNISTPartitioner(*targets*, *num_clients*, *partition='iid'*, *dir_alpha=None*, *major_classes_num=None*, *verbose=True*, *seed=None*)

Bases: *VisionPartitioner*

Data partitioner for FashionMNIST.

For details, please check *VisionPartitioner* and Federated Dataset and DataPartitioner

num_features = 784

class MNISTPartitioner(*targets*, *num_clients*, *partition='iid'*, *dir_alpha=None*, *major_classes_num=None*, *verbose=True*, *seed=None*)

Bases: *VisionPartitioner*

Data partitioner for MNIST.

For details, please check *VisionPartitioner* and Federated Dataset and DataPartitioner.

num_features = 784

class SVHNPartitioner(*targets*, *num_clients*, *partition='iid'*, *dir_alpha=None*, *major_classes_num=None*, *verbose=True*, *seed=None*)

Bases: *VisionPartitioner*

Data partitioner for SVHN.

For details, please check *VisionPartitioner* and Federated Dataset and DataPartitioner

num_features = 1024

class FCUBEPartitioner(*data*, *partition*)

Bases: *DataPartitioner*

FCUBE data partitioner.

FCUBE is a synthetic dataset for research in non-IID scenario with feature imbalance. This dataset and its partition methods are proposed in Federated Learning on Non-IID Data Silos: An Experimental Study.

Supported partition methods for FCUBE:

- feature-distribution-skew:synthetic
- IID

For more details, please refer to Section (IV-B-b) of original paper. For detailed usage, please check Federated Dataset and DataPartitioner.

Parameters

- **data** (*numpy.ndarray*) – Data of dataset FCUBE.
- **partition** (*str*) – Partition type. Only supports *'synthetic'* and *'iid'*.

num_classes = 2

num_clients = 4

_perform_partition(*self*)

__getitem__(*self*, *index*)

__len__(*self*)

class AdultPartitioner(*targets*, *num_clients*, *partition='iid'*, *dir_alpha=None*, *major_classes_num=1*, *verbose=True*, *seed=None*)

Bases: *BasicPartitioner*

Data partitioner for Adult.

For details, please check *BasicPartitioner* and Federated Dataset and DataPartitioner

num_features = 123

num_classes = 2

class RCV1Partitioner(*targets*, *num_clients*, *partition='iid'*, *dir_alpha=None*, *major_classes_num=1*, *verbose=True*, *seed=None*)

Bases: *BasicPartitioner*

Data partitioner for RCV1.

For details, please check *BasicPartitioner* and Federated Dataset and DataPartitioner

num_features = 47236

num_classes = 2

class CovtypePartitioner(*targets*, *num_clients*, *partition='iid'*, *dir_alpha=None*, *major_classes_num=1*, *verbose=True*, *seed=None*)

Bases: *BasicPartitioner*

Data partitioner for Covtype.

For details, please check *BasicPartitioner* and Federated Dataset and DataPartitioner

num_features = 54

num_classes = 2

aggregator

Module Contents

Aggregators	Define the algorithm of parameters aggregation

class Aggregators

Bases: object

Define the algorithm of parameters aggregation

static fedavg_aggregate(*serialized_params_list*, *weights=None*)

FedAvg aggregator

Paper: http://proceedings.mlr.press/v54/mcmahan17a.html

Parameters

- **serialized_params_list** (*list[torch.Tensor]*) – Merge all tensors following FedAvg.
- **weights** (*list, numpy.array or torch.Tensor, optional*) – Weights for each params, the length of weights need to be same as length of `serialized_params_list`

Returns torch.Tensor

static fedasync_aggregate(*server_param*, *new_param*, *alpha*)

FedAsync aggregator

Paper: https://arxiv.org/abs/1903.03934

functional

Module Contents

AverageMeter	Record metrics information

evaluate(model, criterion, test_loader)	Evaluate classify task model accuracy.
read_config_from_json(json_file: str, user_name: str)	Read config from *json_file* to get config for *user_name*
get_best_gpu()	Return gpu (`torch.device`) with largest free memory.
save_dict(dict, path)	
load_dict(path)	
partition_report(targets, data_indices, class_num=None, verbose=True, file=None)	Generate data partition report for clients in `data_indices`.
accuracy(output, target, topk=(1,))	Computes the top-k accuracy for the specified values of k, in range of [0, 1]

class AverageMeter

Bases: `object`

Record metrics information

reset(*self*)

update(*self*, *val*, *n=1*)

evaluate(*model*, *criterion*, *test_loader*)

Evaluate classify task model accuracy.

read_config_from_json(*json_file: str*, *user_name: str*)

Read config from *json_file* to get config for *user_name*

Parameters

- **json_file** (*str*) – path for json_file
- **user_name** (*str*) – read config for this user, it can be 'server' or 'client_id'

Returns a tuple with ip, port, world_size, rank about user with *user_name*

Examples

read_config_from_json('../../../tests/data/config.json', 'server')

Notes

config.json example as follows {

"server": { "ip" : "127.0.0.1", "port": "3002", "world_size": 3, "rank": 0

}, "client_0": {

"ip": "127.0.0.1", "port": "3002", "world_size": 3, "rank": 1

}, "client_1": {

"ip": "127.0.0.1", "port": "3002", "world_size": 3, "rank": 2

}

}

get_best_gpu()

Return gpu (`torch.device`) with largest free memory.

save_dict(*dict*, *path*)

load_dict(*path*)

partition_report(*targets*, *data_indices*, *class_num=None*, *verbose=True*, *file=None*)

Generate data partition report for clients in `data_indices`.

Generate data partition report for each client according to `data_indices`, including ratio of each class and dataset size in current client. Report can be printed in screen or into file. The output format is comma-separated values which can be read by `pandas.read_csv()` or `csv.reader()`.

Parameters

- **targets** (*list or numpy.ndarray*) – Targets for all data samples, with each element is in range of `0` to `class_num-1`.
- **data_indices** (*dict*) – Dict of `client_id: [data indices]`.
- **class_num** (*int, optional*) – Total number of classes. If set to `None`, then `class_num = max(targets) + 1`.
- **verbose** (*bool, optional*) – Whether print data partition report in screen. Default as `True`.
- **file** (*str, optional*) – Output file name of data partition report. If `None`, then no output in file. Default as `None`.

Examples

First generate synthetic data labels and data partition to obtain `data_indices` (`{ client_id: sample indices}`):

```
>>> sample_num = 15
>>> class_num = 4
>>> clients_num = 3
>>> num_per_client = int(sample_num/clients_num)
>>> labels = np.random.randint(class_num, size=sample_num)  # generate 15 labels,␣
↪each label is 0 to 3
>>> rand_per = np.random.permutation(sample_num)
>>> # partition synthetic data into 3 clients
>>> data_indices = {0: rand_per[0:num_per_client],
...                 1: rand_per[num_per_client:num_per_client*2],
...                 2: rand_per[num_per_client*2:num_per_client*3]}
```

Check `data_indices` may look like:

```
>>> data_indices
{0: array([8, 6, 5, 7, 2]),
 1: array([ 3, 10, 14,  4,  1]),
 2: array([13,  9, 12, 11,  0])}
```

Now generate partition report for each client and each class:

```
>>> partition_report(labels, data_indices, class_num=class_num, verbose=True,␣
↪file=None)
Class frequencies:
client,class0,class1,class2,class3,Amount
Client   0,0.200,0.00,0.200,0.600,5
Client   1,0.400,0.200,0.200,0.200,5
Client   2,0.00,0.400,0.400,0.200,5
```

accuracy(*output*, *target*, *topk=(1,)*)

Computes the top-k accuracy for the specified values of k, in range of [0, 1]

logger

Module Contents

Logger	record cmd info to file and print it to cmd at the same time

class Logger(*log_name=None*, *log_file=None*)

Bases: `object`

record cmd info to file and print it to cmd at the same time

Parameters

- **log_name** (*str*) – log name for output.
- **log_file** (*str*) – a file path of log file.

info(*self*, *log_str*)

Print information to logger

warning(*self*, *warning_str*)

Print warning to logger

message_code

Module Contents

MessageCode	Different types of messages between client and server that we support go here.

class MessageCode

Bases: `enum.Enum`

Different types of messages between client and server that we support go here.

ParameterRequest = 0

GradientUpdate = 1

ParameterUpdate = 2

EvaluateParams = 3

Exit = 4

SetUp = 5

Activation = 6

serialization

Module Contents

SerializationTool	

class SerializationTool

Bases: object

static serialize_model_gradients(*model: torch.nn.Module*) → torch.Tensor

static serialize_model(*model: torch.nn.Module*) → torch.Tensor

Unfold model parameters

Unfold every layer of model, concate all of tensors into one. Return a *torch.Tensor* with shape (size,).

Parameters **model** (*torch.nn.Module*) – model to serialize.

static deserialize_model(*model: torch.nn.Module*, *serialized_parameters: torch.Tensor*, *mode='copy'*)

Assigns serialized parameters to model.parameters. This is done by iterating through `model.parameters()` and assigning the relevant params in `grad_update`. NOTE: this function manipulates `model.parameters`.

Parameters

- **model** (*torch.nn.Module*) – model to deserialize.
- **serialized_parameters** (*torch.Tensor*) – serialized model parameters.
- **mode** (*str*) – deserialize mode. "copy" or "add".

Package Contents

Aggregators	Define the algorithm of parameters aggregation
Logger	record cmd info to file and print it to cmd at the same time
MessageCode	Different types of messages between client and server that we support go here.
SerializationTool	

class Aggregators

Bases: object

Define the algorithm of parameters aggregation

static fedavg_aggregate(*serialized_params_list*, *weights=None*)

FedAvg aggregator

Paper: http://proceedings.mlr.press/v54/mcmahan17a.html

Parameters

- **serialized_params_list** (*list[torch.Tensor]*) – Merge all tensors following FedAvg.
- **weights** (*list, numpy.array or torch.Tensor, optional*) – Weights for each params, the length of weights need to be same as length of `serialized_params_list`

Returns torch.Tensor

static fedasync_aggregate(*server_param*, *new_param*, *alpha*)

FedAsync aggregator

Paper: https://arxiv.org/abs/1903.03934

class Logger(*log_name=None*, *log_file=None*)

Bases: `object`

record cmd info to file and print it to cmd at the same time

Parameters

- **log_name** (*str*) – log name for output.
- **log_file** (*str*) – a file path of log file.

info(*self*, *log_str*)

Print information to logger

warning(*self*, *warning_str*)

Print warning to logger

class MessageCode

Bases: `enum.Enum`

Different types of messages between client and server that we support go here.

ParameterRequest = 0

GradientUpdate = 1

ParameterUpdate = 2

EvaluateParams = 3

Exit = 4

SetUp = 5

Activation = 6

class SerializationTool

Bases: `object`

static serialize_model_gradients(*model: torch.nn.Module*) → torch.Tensor

static serialize_model(*model: torch.nn.Module*) → torch.Tensor

Unfold model parameters

Unfold every layer of model, concate all of tensors into one. Return a *torch.Tensor* with shape (size,).

Parameters **model** (*torch.nn.Module*) – model to serialize.

static deserialize_model(*model: torch.nn.Module*, *serialized_parameters: torch.Tensor*, *mode='copy'*)

Assigns serialized parameters to model.parameters. This is done by iterating through `model.parameters()` and assigning the relevant params in `grad_update`. NOTE: this function manipulates `model.parameters`.

Parameters

- **model** (*torch.nn.Module*) – model to deserialize.
- **serialized_parameters** (*torch.Tensor*) – serialized model parameters.
- **mode** (*str*) – deserialize mode. "copy" or "add".

10.1.3 Package Contents

__version__ = 1.2.1

CHAPTER

ELEVEN

CITATION

Please cite **FedLab** in your publications if it helps your research:

```
@article{smile2021fedlab,
title={FedLab: A Flexible Federated Learning Framework},
author={Dun Zeng, Siqi Liang, Xiangjing Hu and Zenglin Xu},
journal={arXiv preprint arXiv:2107.11621},
year={2021}
}
```

CHAPTER

TWELVE

CONTACTS

Contact the **FedLab** development team through Github issues or email:

- Dun Zeng: zengdun@foxmail.com
- Siqi Liang: zszxlsq@gmail.com

BIBLIOGRAPHY

[1] Cong Xie, Sanmi Koyejo, and Indranil Gupta. Asynchronous federated optimization. *arXiv preprint arXiv:1903.03934*, 2019.

[2] Sebastian Caldas, Sai Meher Karthik Duddu, Peter Wu, Tian Li, Jakub Konečn\`y, H Brendan McMahan, Virginia Smith, and Ameet Talwalkar. Leaf: a benchmark for federated settings. *arXiv preprint arXiv:1812.01097*, 2018.

[3] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*, 1273–1282. PMLR, 2017.

[4] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep gradient compression: reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*, 2017.

[5] Durmus Alp Emre Acar, Yue Zhao, Ramon Matas, Matthew Mattina, Paul Whatmough, and Venkatesh Saligrama. Federated learning based on dynamic regularization. In *International Conference on Learning Representations*. 2020.

[6] Mikhail Yurochkin, Mayank Agarwal, Soumya Ghosh, Kristjan Greenewald, Nghia Hoang, and Yasaman Khazaeni. Bayesian nonparametric federated learning of neural networks. In *International Conference on Machine Learning*, 7252–7261. PMLR, 2019.

[7] Qinbin Li, Yiqun Diao, Quan Chen, and Bingsheng He. Federated learning on non-iid data silos: an experimental study. *arXiv preprint arXiv:2102.02079*, 2021.

[8] Hongyi Wang, Mikhail Yurochkin, Yuekai Sun, Dimitris Papailiopoulos, and Yasaman Khazaeni. Federated learning with matched averaging. *arXiv preprint arXiv:2002.06440*, 2020.

PYTHON MODULE INDEX

f

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W