fedsim

Release 0.9.1

Farshid Varno

Sep 23, 2022

CONTENTS

CHAPTER

ONE

EASY INSTALL AND RUN

Listing 1: Install using pip:

```
pip install fedsim
```

Listing 2: Train MNIST on 500 clients:

```
fedsim-cli fed-learn
```

CHAPTER

TWO

USER GUIDE

This guide is an overview and explains the important features; details are found in *API Reference*.

2.1 Quick User Guide

2.1.1 FedSim

FedSim is a comprehensive and flexible Federated Learning Simulator! It aims to provide the researchers with an easy to develope/maintain simulator for Federated Learning. See documentation at here!

2.1.2 Installation

```
pip install fedsim
```

That's it! You are all set!

2.1.3 Design Architecture

2.1.4 CLI

Minimal example

Fedsim provides powerful cli tools that allow you to focus on designing what is truly important. Simply enter the following command to begin federatively training a model.

```
fedsim-cli fed-learn
```

The "MNIST" dataset is partitioned on 500 clients by default, and the FedAvg algorithm is used to train a minimal model with two fully connected layers. A text file is made that descibes the configuration for the experiment and a summary of results when it is finished. Additionally, a tensorboard log file is made to monitor the scores/metrics of the training. The directory that these files are stored is (reconfigurable and is) displayed while the experiment is running.

Hooking scores to cli tools

In case you are interested in a certain metric you can make a query for it in your command. For example, lets assume we would like to test and report: * the accuracy score of the global model on global test dataset both every 21 rounds and every 43 rounds. * the average accuracy score of the local models every 15 rounds. Here's how we modify the above command:

```
fedsim-cli fed-learn \
    --global-score Accuracy score_name:acc21 split:test log_freq:21 \
    --global-score Accuracy score_name:acc43 split:test log_freq:43 \
    --local-score Accuracy split:train log_freq:15
```

Check Fedsim Scores Page for the list of all other scores like Accyracy or define your custom score.

Changing the Data

Data partitioning and retrieval is controlled by a `DataManager` object. This object could be controlled through *-d* or *--data-manager* flag in most cli commands. In the following we modify the arguments of the default `DataManager` such that `CIFAR100` is partitioned over 1000 clients.

```
fedsim-cli fed-learn \
    --data-manger BasicDataManager dataset:cifar100 num_partitions:1000 \
    --num-clients 1000 \
    --model SimpleCNN2 num_classes:100 \
    --global-score Accuracy split:test log_freq:15
```

Notice that we also changed the model from default to `SimpleCNN2` which by default takes 3 input channels. You can learn about existing data managers at data manager documentation and Custom data managers at this guide to make Custom data managers.

Note: Arguments of the constructor of any component (rectangular boxes in the image of design architecture) could be given in *arg:value* format following its name (or *path* if a local file is provided). Among these component, the algorithm is special, in that the arguments are controlled internally. The only arguments of the algorithm object that could be directly controlled in your commands is the algorithm specific ones (mostly hyper-parameters). Examples:

```
fedsim-cli fed-learn --algorithm AdaBest mu:0.01 beta:0.6 ...
```

Feed CLI with Customized Components

The cli tool can take a locally defined component by ingesting its path. For example, to automatically include your custom algorithm by the a command of the cli tool, you can place your class in a python file and pass the path of the file to *-a* or *--algorithm* option (without .py) followed by colon and name of the algorithm definition (class or method). For instance, if you have algorithm *CustomFLAlgorithm* stored in a *foo/bar/my_custom_alg.py*, you can pass *--algorithm foo/bar/my_custom_alg:CustomFLAlgorithm*.

```
fedsim-cli fed-learn --algorithm foo/bar/my_custom_alg_file:CustomFLAlgorithm mu:0.01 ...
```

The same is possible for any other component, for instance for a Custom model:

```
fedsim-cli fed-learn --model foo/bar/my_model_file:CustomModel num_classes:1000 ...
```

More about cli commands

For help with cli check fedsim-cli documentation or read the output of the following commands:

```
fedsim-cli --help
fedsim-cli fed-learn --help
fedsim-cli fed-tune --help
```

2.1.5 Python API

Fedsim is shipped with some of the most well-known Federated Learning algorithms included. However, you will most likely need to quickly develop and test your custom algorithm, model, data manager, or score class. Fedsim has been designed in such a way that doing all of these things takes almost no time and effort. Let's start by learning how to import and use Fedsim, and then we'll go over how to easily modify existing modules and classes to your liking. Check the following basic example:

```
from logall import TensorboardLogger
from fedsim.distributed.centralized.training import FedAvg
from fedsim.distributed.data_management import BasicDataManager
from fedsim.models import SimpleCNN2
from fedsim.losses import CrossEntropyLoss
from fedsim.scores import Accuracy

n_clients = 1000

dm = BasicDataManager("./data", "cifar100", n_clients)
sw = TensorboardLogger(path=None)

alg = FedAvg(
    data_manager=dm,
    num_clients=n_clients,
    sample_scheme="uniform",
    sample_rate=0.01,
    model_def=partial(SimpleCNN2, num_channels=3),
    epochs=5,
    criterion_def=partial(CrossEntropyLoss, log_freq=100),
    batch_size=32,
    metric_logger=sw,
    device="cuda",
)
alg.hook_local_score(
    partial(Accuracy, log_freq=50),
    split='train,
    score_name="accuracy",
)
alg.hook_global_score(
    partial(Accuracy, log_freq=40),
    split='test,
    score_name="accuracy",
)
report_summary = alg.train(rounds=50)
```

2.1.6 Side Notes

- Do not use double underscores (__) in argument names of your customized classes.

2.2 Guide to data manager

Provided with the simulator is a basic DataManager called `BasicDataManager` which for now supports the following datasets

- MNIST
- CIFAR10
- CIFAR100

It supports the popular partitioning schemes (iid, Dirichlet distribution, unbalanced, etc.).

2.2.1 Custom DataManager

Any Custom data manager class should inherit from `fedsim.data_manager.data_manager.DataManager` (or its children) and implement its abstract methods.

2.2.2 DataManager Template

```
from fedsim.distributed.data_management import DataManager

class CustomDataManager(DataManager)
     def __init__(self, root, seed, save_dir=None, other_args="default value", ...):
         self.other_arg = other_arg
         """
         apply the changes required by the abstract methods here (before calling
         super's constructor).
         """
         super(BasicDataManager, self).__init__(root, seed, save_dir=save_dir)
         """
         apply the operation that assume the abstract methods are performed here
         (after calling super's constructor).
         """



     def make_datasets(self, root: str) -> Tuple[object, object]:
         """makes and returns local and global dataset objects. The created datasets do
         not need a transform as recompiled datasets with separately provided transforms
         on the fly (for vision datasets).

         Args:
             dataset_name (str): name of the dataset.
             root (str): directory to download and manipulate data.

         Raises:
             NotImplementedError: this abstract method should be
```

(continues on next page)

(continued from previous page)

```
                implemented by child classes

        Returns:
            Tuple[object, object]: local and global dataset
        """
        raise NotImplementedError

    def make_transforms(self) -> Tuple[object, object]:
        """make and return the dataset trasformations for local and global split.

        Raises:
            NotImplementedError: this abstract method should be
                implemented by child classes
        Returns:
            Tuple[Dict[str, Callable], Dict[str, Callable]]: tuple of two dictionaries,
                first, the local transform mapping and second the global transform
                mapping.
        """
        raise NotImplementedError

    def partition_local_data(self, datasets: Dict[str, object]) -> Dict[str,␣
↪Iterable[Iterable[int]]]:
        """partitions local data indices into splits and within each split, partition␣
↪in client-indexed Iterable.
        Return a dictionary of these splits (e.g., train, test, ...).

        Args:
            dataset (object): local dataset

        Raises:
            NotImplementedError: this abstract method should be
                implemented by child classes

        Returns:
            Dict[str, Iterable[Iterable[int]]]:
                dictionary of {split:client-indexed iterables of example indices}.
        """
       raise NotImplementedError


    def partition_global_data(
        self,
        dataset: object,
    ) -> Dict[str, Iterable[int]]:
        """partitions global data indices into desired splits (e.g., train, test, ...).

        Args:
            dataset (object): global dataset

        Returns:
            Dict[str, Iterable[int]]:
                dictionary of {split:example indices of global dataset}.
```

(continues on next page)

(continued from previous page)

```
        """
        raise NotImplementedError

    def get_identifiers(self) -> Sequence[str]:
        """ Returns identifiers to be used for saving the partition info.
        A unique identifier for a unique setup ensures the credibility of comparing␣
↪your experiments results.

        Raises:
            NotImplementedError: this abstract method should be
                implemented by child classes

        Returns:
            Sequence[str]: a sequence of str identifing class instance
        """
        raise NotIm
```

Note: scores can be passed to `--criterion` option the same way, however, if the selected score class is not differentiable an error may be raised (if necessary).plementedError

You can use BasicDataManager as a working template.

Integration with fedsim-cli

To automatically include your custom data-manager into the provided cli tool, you can define it in a python file and pass its path to `-a` or `--data-manager` option (without .py) followed by colon and the definition of the data-manager (class or method). For example, if you have data-manager `DataManager` stored in `foo/bar/my_custom_dm.py`, you can pass `--data-manager foo/bar/my_custom_dm:DataManager`.

Note: Arguments of constructor of any data-manager could be given in `arg:value` format following its name (or *path* if a local file is provided). Examples:

```
fedsim-cli fed-learn --data-manager BasicDataManager num_clients:1100 ...
```

```
fedsim-cli fed-learn --data-manager foo/bar/my_custom_dm:DataManager arg1:value ...
```

2.3 Guide to centralized FL algorithms

2.3.1 Included FL algorithms

Alias	Paper
FedAvg	
FedNova	
FedProx	
FedDyn	
AdaBest	
FedDF	

2.3.2 Algorithm interface

Look at the design architecture illustrated in the image below. .. image:: ../_static/arch.svg

2.3.3 Custom Centralized FL Algorithm

Implementing a new fedsim algorithm is very simple. There are on three things to remember: 1. any Custom FL algorithm class has to inherit from a base algorithm (e.g., `CentralFLAlgorithm`) or one of their children classes (e.g., `FedAvg`). 2. the user methods should be implemented (see an algorithm template below) withoud **self** argument (static methods). 3. global models/parameters have to be cloned and detached before local training.

`fedsim.distributed.centralized.CentralFLAlgorithm` (or its children) and implement its abstract methods.

2.3.4 Algorithm Template

```
from fedsim.distributed.centralized.centralized_fl_algorithm import CentralFLAlgorithm


class GreetingAlgorithm(CentralFLAlgorithm):
    def init(server_storage):
        # do operations required prior to training. For exampel you can make your model␣
↪and optimizer here.
        # use read and write methods of server_storage to retrieve definitions and store␣
↪the result of your operation.
        # server_storage.get_keys() returns list of definitions required to build␣
↪objects you like.
        model_def = server_storage.read("model_def")
        model = model_def()
        server_storage.write("model", model)

    def send_to_client(server_storage, client_id):
        # add your message for client with id <client_id> here. This method runs at the␣
↪beginning of each round for each sampled client.
        return f"Hello client {client_id}!"

    def send_to_server(
        id, rounds, storage, datasets, train_split_name, scores, epochs, criterion,␣
↪train_batch_size,
```

(continues on next page)

(continued from previous page)

```
        inference_batch_size, optimizer_def, lr_scheduler_def=None, device="cuda",␣
↪ctx=None, step_closure=None,
    ):
        # this is what client <id> does locally. ``ctx`` is the message send from the␣
↪server.
        print(f"Message received from server on client {id}: {ctx}")
        return f"Hello server, this is {id}!"

    def receive_from_client(server_storage, client_id, client_msg, train_split_name,␣
↪serial_aggregator, appendix_aggregator):
        # this method is to collect information from clients as their messages arrive.
        # use serial_aggregator.add amd appendix_aggregator.append to serially aggregate␣
↪pieces of info received from the client.
        print(f"Message from {client_id}: {client_msg}")
        # return True if message is received without any problems
        return True

    def optimize(server_storage, serial_aggregator, appendix_aggregator):
        # optimize the server parameters here. Additionally, unpack and arrange the␣
↪reports from the aggregators here.
        # return yuor optimization reports (along with those unpacked from the␣
↪aggtegators)
        return f"Nothing to report here!"

    def deploy(server_storage):
        # send the deployment points, so that the report can be made for those points
        return dict(point1="foo", point2="bar")

    def report(server_storage, dataloaders, rounds, scores, metric_logger, device,␣
↪optimize_reports, deployment_points=None):
        # report your findings using metric_logger. Those metrics that are in scalar␣
↪format can be returned in a dictionary (with their name as the key).
        # the entries in the returned dictionary are automatically reported using metric_
↪logger
        return dict(x=1, y=2)
```

2.3.5 Examples

Here's the complete implementation of Federated Averaging (FedAvg) algorithm which could be used as a template:

```
import math
from torch.utils.data import DataLoader
from torch.utils.data import RandomSampler
from fedsim.local.training import local_inference
from fedsim.local.training import local_train
from fedsim.local.training.step_closures import default_step_closure
from fedsim.utils import initialize_module
from fedsim.utils import vectorize_module

from fedsim.distributed.centralized import CentralFLAlgorithm
from fedsim.distributed.centralized.training import serial_aggregation
```

(continues on next page)

(continued from previous page)

```
class FedAvg(CentralFLAlgorithm):
    def init(server_storage):
        device = server_storage.read("device")
        model = server_storage.read("model_def")().to(device)
        params = vectorize_module(model, clone=True, detach=True)
        optimizer = server_storage.read("optimizer_def")(params=[params])
        lr_scheduler = None
        lr_scheduler_def = server_storage.read("lr_scheduler_def")
        if lr_scheduler_def is not None:
            lr_scheduler = lr_scheduler_def(optimizer=optimizer)
        server_storage.write("model", model)
        server_storage.write("cloud_params", params)
        server_storage.write("optimizer", optimizer)
        server_storage.write("lr_scheduler", lr_scheduler)

    def send_to_client(server_storage, client_id):
        # load cloud stuff
        cloud_params = server_storage.read("cloud_params")
        model = server_storage.read("model")
        # copy cloud params to cloud model to send to the client
        initialize_module(model, cloud_params, clone=True, detach=True)
        # return a copy of the cloud model
        return dict(model=model)

    # define client operation
    def send_to_server(
        id, rounds, storage, datasets, train_split_name, scores, epochs, criterion,␣
↪train_batch_size,
        inference_batch_size, optimizer_def, lr_scheduler_def=None, device="cuda",␣
↪ctx=None, step_closure=None,
    ):
        # create a random sampler with replacement so that
        # stochasticity is maximiazed and privacy is not compromized
        sampler = RandomSampler(
            datasets[train_split_name], replacement=True,
            num_samples=math.ceil(len(datasets[train_split_name]) / train_batch_size) *␣
↪train_batch_size,
        )
        # # create train data loader
        train_loader = DataLoader(datasets[train_split_name], batch_size=train_batch_
↪size, sampler=sampler)

        model = ctx["model"]
        optimizer = optimizer_def(model.parameters())
        lr_scheduler = None if lr_scheduler_def is None else lr_scheduler_
↪def(optimizer=optimizer)

        # optimize the model locally
        step_closure_ = default_step_closure if step_closure is None else step_closure
        train_scores = scores[train_split_name] if train_split_name in scores else dict()
```

(continues on next page)

(continued from previous page)

```
        num_train_samples, num_steps, diverged, = local_train(
            model, train_loader, epochs, 0, criterion, optimizer, lr_scheduler, device,␣
↪step_closure_,
            scores=train_scores,
        )
        # get average train scores
        metrics_dict = {train_split_name: {name: score.get_score() for name, score in␣
↪train_scores.items()}}
        # append train loss
        if rounds % criterion.log_freq == 0:
            metrics_dict[train_split_name][criterion.get_name()] = criterion.get_score()
        num_samples_dict = {train_split_name: num_train_samples}
        # other splits
        for split_name, split in datasets.items():
            if split_name != train_split_name and split_name in scores:
                o_scores = scores[split_name]
                split_loader = DataLoader( split, batch_size=inference_batch_size,␣
↪shuffle=False)
                num_samples = local_inference(model, split_loader, scores=o_scores,␣
↪device=device)
                metrics_dict[split_name] = {name: score.get_score() for name, score in o_
↪scores.items()}
                num_samples_dict[split_name] = num_samples
        # return optimized model parameters and number of train samples
        return dict(local_params=vectorize_module(model), num_steps=num_steps,␣
↪diverged=diverged,
            num_samples=num_samples_dict,metrics=metrics_dict,
        )

    def receive_from_client(
        server_storage, client_id, client_msg, train_split_name, serial_aggregator,␣
↪appendix_aggregator
    ):
        return serial_aggregation(
            server_storage, client_id, client_msg, train_split_name, serial_aggregator
        )

    def optimize(server_storage, serial_aggregator, appendix_aggregator):
        if "local_params" in aggregator:
            param_avg = aggregator.pop("local_params")
            optimizer = server_storage.read("optimizer")
            lr_scheduler = server_storage.read("lr_scheduler")
            cloud_params = server_storage.read("cloud_params")
            pseudo_grads = cloud_params.data - param_avg
            # update cloud params
            optimizer.zero_grad()
            cloud_params.grad = pseudo_grads
            optimizer.step()
            if lr_scheduler is not None:
                lr_scheduler.step()
            # purge aggregated results
            del param_avg
```

(continues on next page)

(continued from previous page)

```
        return aggregator.pop_all()

    def deploy(server_storage):
        return dict(avg=server_storage.read("cloud_params"))

    def report(
        server_storage, dataloaders, rounds, scores, metric_logger, device, optimize_
↪reports, deployment_points=None,
    ):
        model = server_storage.read("model")
        scores_from_deploy = dict()
        if deployment_points is not None:
            for point_name, point in deployment_points.items():
                # copy cloud params to cloud model to send to the client
                initialize_module(model, point, clone=True, detach=True)

                for split_name, loader in dataloaders.items():
                    if split_name in scores:
                        split_scores = scores[split_name]
                        _ = local_inference(model, loader,scores=split_scores,␣
↪device=device)
                        split_score_results = {
                            f"server.{point_name}.{split_name}." f"{score_name}": score.
↪get_score()
                            for score_name, score in split_scores.items()
                        }
                        scores_from_deploy = {
                            **scores_from_deploy,
                            **split_score_results,
                        }
        return {**scores_from_deploy, **optimize_reports, **norm_reports}
```

You can easily make changes by inheriting from FedAvg or its children classes. For example the following is the implementation of FedProx algorithm:

```
from functools import partial
from torch.nn.utils import parameters_to_vector
from fedsim.local.training.step_closures import default_step_closure
from fedsim.utils import vector_to_parameters_like
from fedsim.utils import vectorize_module
from fedsim.distributed.centralized import FedAvg


class FedProx(FedAvg):
    def init(server_storage, *args, **kwrag):
        default_mu = 0.0001
        FedAvg.init(server_storage)
        server_storage.write("mu", kwrag.get("mu", default_mu))

    def send_to_client(server_storage, client_id):
        server_msg = FedAvg.send_to_client(server_storage, client_id)
        server_msg["mu"] = server_storage.read("mu")
```

(continues on next page)

(continued from previous page)

```
        return server_msg

    def send_to_server(
        id, rounds, storage, datasets, train_split_name, scores, epochs, criterion,␣
↪train_batch_size,
        inference_batch_size, optimizer_def, lr_scheduler_def=None, device="cuda",␣
↪ctx=None, step_closure=None,
    ):
        model = ctx["model"]
        mu = ctx["mu"]
        params_init = vectorize_module(model, clone=True, detach=True)

        def transform_grads_fn(model):
            params = parameters_to_vector(model.parameters())
            grad_additive = 0.5 * (params - params_init)
            grad_additive_list = vector_to_parameters_like(mu * grad_additive, model.
↪parameters())

            for p, g_a in zip(model.parameters(), grad_additive_list):
                p.grad += g_a

        step_closure_ = partial(default_step_closure, transform_grads=transform_grads_fn)
        return FedAvg.send_to_server(
            id, rounds, storage, datasets, train_split_name, scores, epochs, criterion,␣
↪train_batch_size,
            inference_batch_size, optimizer_def, lr_scheduler_def, device, ctx, step_
↪closure=step_closure_,
        )
```

Integration with fedsim-cli

To automatically include your custom algorithm by the provided cli tool, you can define it in a python and pass its path to `-a` or `--algorithm` option (without .py) followed by column and name of the algorithm. For example, if you have algorithm `CustomFLAlgorithm` stored in a `foo/bar/my_custom_alg.py`, you can pass `--algorithm foo/bar/my_custom_alg:CustomFLAlgorithm`.

Note: Non-common Arguments of constructor of any algoritthm (mostly hyper-parameters) could be given in `arg:value` format following its name (or `path` if a local file is provided). Areguments that are common among the desired algorithm and CentralFLAlgorithm are internally assigned. Examples:

```
fedsim-cli fed-learn --algorithm AdaBest mu:0.01 beta:0.6 ...
```

```
fedsim-cli fed-learn --algorithm foo/bar/my_custom_alg:CustomFLAlgorithm mu:0.01 ...
```

2.4 Guide to models

2.4.1 Custom Model

Any custom model class should inherit from `torch.Module` (or its children) and implement its abstract methods.

Integration with fedsim-cli

To automatically include your custom model by the provided cli tool, you can define it in a python file and pass its path to *-m* or *--model* option (without .py) followed by column and name of the model definition (class or method). For example, if you have model `CustomModel` stored in a `foo/bar/my_custom_model.py`, you can pass `--model foo/bar/my_custom_alg:CustomModel`.

Note: Arguments of constructor of any model could be given in *arg:value* format following its name (or *path* if a local file is provided). Examples:

```
fedsim-cli fed-learn --model cnn_mnist num_classes:8 ...
```

```
fedsim-cli fed-learn --model foo/bar/my_custom_alg:CustomModel num_classes:8 ...
```

2.5 Guide to optimziers

2.5.1 Custom optimziers

Any custom optimizer class should inherit from `torch.optim.Optimizer` (or its children) and implement its abstract methods.

Integration with fedsim-cli

To automatically include your custom optimizer by the provided cli tool, you can define it in a python file and pass its path to `--optimzier` or `--local-optimzier` option (without .py) followed by column and name of the optimizer definition (class or method). For example, if you have optimizer `CustomOpt` stored in a `foo/bar/my_custom_opt.py`, you can pass `--optimizer foo/bar/my_custom_opt:CustomOpt` for setting global optimizer or `--local-optimizer foo/bar/my_custom_opt:CustomOpt` for setting the local optimizer.

Note: Arguments of constructor of any optimzier could be given in `arg:value` format following its name (or *path* if a local file is provided). Examples:

```
fedsim-cli fed-learn --optimzier SGD lr:0.1 weight_decay:0.001 ...
```

```
fedsim-cli fed-learn --local-optimizer foo/bar/my_custom_opt:CustomOpt lr:0.2 momentum:
↪True ...
```

2.6 Guide to scores

2.6.1 Custom scores

Any custom score class should inherit from `fedsim.scores.Score` (or its children) and implement its abstract methods.

Integration with fedsim-cli

To automatically include your custom score by the provided cli tool, you can define it in a python file and pass its path to `--global-score` or `--local-score` option (without .py) followed by column and name of the score definition (class or method). For example, if you have score `CustomScore` stored in a `foo/bar/my_custom_score.py`, you can pass `--global-score foo/bar/my_custom_score:CustomScore` for setting global optimizer or `--local-score foo/bar/my_custom_score:CustomScore` for setting the local score.

Note: Arguments of constructor of any score could be given in `arg:value` format following its name (or *path* if a local file is provided). Examples:

```
fedsim-cli fed-learn --global-score Accuracy log_freq:20 split:test ...
```

```
fedsim-cli fed-learn --local-score foo/bar/my_custom_sore:CustomScore log_freq:30 split:
↪train ...
```

Note: scores can be passed to `--criterion` option the same way, however, if the selected score class is not differentiable an error may be raised (if necessary).

2.7 Guide to learning rate schedulers

fedsim-cli fed-learn accepts 3 scheduler objects.

- **lr-scheduler:** learning rate scheduler for server optimizer.
- **local-lr-scheduler:** learning rate scheduler for client optimizer.
- **r2r-local-lr-scheduler:** schedules the initial learning rate that is delivered to the clients of each round.

These arguments are passed to instances of the centralized FL algorithms.

Note: Choose learning rate schedulers from `torch.optim.lr_scheduler` documented at Lr Schedulers Page or define a learning rate scheduler class that has the common methods (`step`, `get_last_lr`, etc.).

Note: For now `fedsim-cli` does not support the learning rate schedulers that require another object in their constructor (such as `LambdaLR`) or a dynamic value in their step function (`ReduceLROnPlateau`). To implement one with similar functionality, you can implement one and assign it to `self.r2r_local_lr_scheduler` inside the constructor of your custom algorithm (after calling super).

2.7.1 Custom Learning Rate Scheduler

Any custom learning rate scheduler class should implement the common methods of torch optim lr schedulers.

Integration with fedsim-cli

To automatically include your custom lr scheduler by the provided cli tool, you can define it in a python file and pass its path to `--lr-scheduler` or `--local-lr-scheduler` or `r2r-local-lr-scheduler` option (without .py) followed by column and name of the lr scheduler definition (class or method). For example, if you have score `CustomLRS` stored in a `foo/bar/my_custom_lr_scheduler.py`, you can pass `--lr-scheduler foo/bar/my_custom_lr_scheduler:CustomLRS` for setting global lr scheduler or `--local-lr-scheduler foo/bar/my_custom_lr_scheduler:CustomLRS` for setting the local lr scheduler or `--r22-local-lr-scheduler foo/bar/my_custom_lr_scheduler:CustomLRS` for setting the round to round lr scheduler. The latter determines the initial learning rate of the local optimizer at each round.

Note: Arguments of constructor of any lr scheduler could be given in `arg:value` format following its name (or *path* if a local file is provided). Examples:

```
fedsim-cli fed-learn --lr-scheduler StepLR step_size:200 gamma:0.5 ...
```

```
fedsim-cli fed-learn --local-lr-scheduler foo/bar/my_custom_lr_scheduler:CustomLRS step_
↪size:10 beta:0.1 ...
```

2.8 Fine-tuning

The cli includes a fine-tuning tool. Under the hood `fedsim-cli fed-tune` uses Bayesian optimization provided by scikit-optimize (skopt) to tune the hyper-parameters. Besides `skopt` argumetns, it accepts all arguments that could be used by `fedsim-cli fed-learn`. The arguments values could be defined as search spaces.

- To define a float range to tune use `Real` keyword as the argument value (e.g., `mu:Real:0-0.1`)
- To define an integer range to tune use `Integer` keyword as the argument value (e.g., `arg1:Integer:2-15`)
- To define a categorical range to tune use `Categorical` keyword as the argument value (e.g., `arg2:Categorical:uniform-normal-special`)

Examples

```
fedsim-cli fed-tune --epochs 1 --n-clients 2 --client-sample-rate 0.5 -a AdaBest mu:Real:
↪0-0.1 beta:Real:0.3-1 --maximize-metric --n-iters 20
```

CHAPTER

THREE

API REFERENCE

Release 0.9.1

Date Sep 23, 2022

This reference manual details functions, modules, and objects included in FedSim, describing what they are and what they do. For learning how to use FedSim, see the *complete documentation*.

3.1 FedSim

Comprehensive and flexible Federated Learning Simulator!

3.1.1 Distributed Learning

Centralized Distributed Learning

Centralized Compression

There are no centralized compressions algorithms defined in this version.

Centralized Privacy

There are no centralized privacy algorithms defined in this version.

Centralized Training

Algorithms for centralized Federated training.

AdaBest

class AdaBest(*data_manager*, *metric_logger*, *num_clients*, *sample_scheme*, *sample_rate*, *model_def*, *epochs*, *criterion_def*, *optimizer_def=functools.partial(<class 'torch.optim.sgd.SGD'>, lr=1.0)*, *local_optimizer_def=functools.partial(<class 'torch.optim.sgd.SGD'>, lr=0.1)*, *lr_scheduler_def=None*, *local_lr_scheduler_def=None*, *r2r_local_lr_scheduler_def=None*, *batch_size=32*, *test_batch_size=64*, *device='cpu'*, **args*, ***kwargs*)

Implements AdaBest algorithm for centralized FL.

For further details regarding the algorithm we refer to AdaBest: Minimizing Client Drift in Federated Learning via Adaptive Bias Estimation.

Parameters

- **data_manager** (`distributed.data_management.DataManager`) -- data manager
- **metric_logger** (`logall.Logger`) -- metric logger for tracking.
- **num_clients** (*int*) -- number of clients
- **sample_scheme** (`str`) -- mode of sampling clients. Options are `'uniform'` and `'sequential'`
- **sample_rate** (`float`) -- rate of sampling clients
- **model_def** (`torch.Module`) -- definition of for constructing the model
- **epochs** (`int`) -- number of local epochs
- **criterion_def** (`Callable`) -- loss function defining local objective
- **optimizer_def** (`Callable`) -- derfintion of server optimizer
- **local_optimizer_def** (`Callable`) -- defintoin of local optimizer
- **lr_scheduler_def** (`Callable`) -- definition of lr scheduler of server optimizer.
- **local_lr_scheduler_def** (`Callable`) -- definition of lr scheduler of local optimizer
- **r2r_local_lr_scheduler_def** (`Callable`) -- definition to schedule lr that is delivered to the clients at each round (deterimined init lr of the client optimizer)
- **batch_size** (*int*) -- batch size of the local trianing
- **test_batch_size** (*int*) -- inference time batch size
- **device** (*str*) -- cpu, cuda, or gpu number
- **mu** (*float*) -- AdaBest's μ hyper-parameter for local regularization
- **beta** (*float*) -- AdaBest's β hyper-parameter for global regularization

---

Note:

definition of

- **learning rate schedulers, could be any of the ones defined at** `torch.optim.lr_scheduler` or any other that implements step and get_last_lr methods._schedulers``.
- optimizers, could be any `torch.optim.Optimizer`.
- model, could be any `torch.Module`.
- criterion, could be any `fedsim.scores.Score`.

`deploy()`

return Mapping of name -> parameters_set to test the model

Parameters **server_storage** (*Storage*) -- server storage object.

`init(*args, **kwrag)`

this method is executed only once at the time of instantiating the algorithm object. Here you define your model and whatever needed during the training. Remember to write the outcome of your processing to server_storage for access in other methods.

Note: `*args` and `**kwargs` are directly passed through from algorithm constructor.

Parameters **server_storage** (*Storage*) -- server storage object

`optimize(serial_aggregator, appendix_aggregator)`

optimize server mdoel(s) and return scores to be reported

Parameters

- **server_storage** (*Storage*) -- server storage object.
- **serial_aggregator** (*SerialAggregator*) -- serial aggregator instance of current round.
- **appendix_aggregator** (*AppendixAggregator*) -- appendix aggregator instance of current round.

Raises `NotImplementedError` -- abstract class to be implemented by child

Returns *Mapping[Hashable, Any]* -- context to be reported

`receive_from_client(client_id, client_msg, train_split_name, serial_aggregator, appendix_aggregator)`

receive and aggregate info from selected clients

Parameters

- **server_storage** (*Storage*) -- server storage object.
- **client_id** (*int*) -- id of the sender (client)
- **client_msg** (*Mapping[Hashable, Any]*) -- client context that is sent.
- **train_split_name** (*str*) -- name of the training split on clients.
- **aggregator** (*SerialAggregator*) -- aggregator instance to collect info.

Returns *bool* -- success of the aggregation.

Raises `NotImplementedError` -- abstract class to be implemented by child

`send_to_client(client_id)`

returns context to send to the client corresponding to client_id.

Warning: Do not send shared objects like server model if you made any before you deepcopy it.

Parameters

- **server_storage** (*Storage*) -- server storage object.
- **client_id** (*int*) -- id of the receiving client

Raises `NotImplementedError` -- abstract class to be implemented by child

Returns *Mapping[Hashable, Any]* -- the context to be sent in form of a Mapping

`send_to_server`(*rounds*, *storage*, *datasets*, *train_split_name*, *scores*, *epochs*, *criterion*, *train_batch_size*, *inference_batch_size*, *optimizer_def*, *lr_scheduler_def=None*, *device='cuda'*, *ctx=None*, *step_closure=None*)

client operation on the recieved information.

Parameters

- **id** (*int*) -- id of the client
- **rounds** (*int*) -- global round number
- **storage** (*Storage*) -- storage object of the client
- **datasets** (*Dict[str, Iterable]*) -- this comes from Data Manager
- **train_split_name** (*str*) -- string containing name of the training split
- **scores** -- Dict[str, Dict[str, Score]]: dictionary of form {'split_name':{'score_name': Score}} for global scores to evaluate at the current round.
- **epochs** (*int*) -- number of epochs to train
- **criterion** (*Score*) -- citerion, should be a differentiable fedsim.scores.score
- **train_batch_size** (*int*) -- training batch_size
- **inference_batch_size** (*int*) -- inference batch_size
- **optimizer_def** (*float*) -- class for constructing the local optimizer
- **lr_scheduler_def** (*float*) -- class for constructing the local lr scheduler
- **device** (*Union[int, str], optional*) -- Defaults to 'cuda'.
- **ctx** (*Optional[Dict[Hashable, Any]], optional*) -- context reveived.

Returns *Mapping[str, Any]* -- client context to be sent to the server

FedAvg

`class FedAvg`(*data_manager*, *metric_logger*, *num_clients*, *sample_scheme*, *sample_rate*, *model_def*, *epochs*, *criterion_def*, *optimizer_def=functools.partial(<class 'torch.optim.sgd.SGD'>, lr=1.0)*, *local_optimizer_def=functools.partial(<class 'torch.optim.sgd.SGD'>, lr=0.1)*, *lr_scheduler_def=None*, *local_lr_scheduler_def=None*, *r2r_local_lr_scheduler_def=None*, *batch_size=32*, *test_batch_size=64*, *device='cpu'*, **args*, ***kwargs*)

Implements FedAvg algorithm for centralized FL. For further details regarding the algorithm we refer to Communication-Efficient Learning of Deep Networks from Decentralized Data.

Parameters

- **data_manager** (`distributed.data_management.DataManager`) -- data manager
- **metric_logger** (`logall.Logger`) -- metric logger for tracking.
- **num_clients** (*int*) -- number of clients
- **sample_scheme** (`str`) -- mode of sampling clients. Options are `'uniform'` and `'sequential'`
- **sample_rate** (`float`) -- rate of sampling clients

- **model_def** (`torch.Module`) -- definition of for constructing the model
- **epochs** (`int`) -- number of local epochs
- **criterion_def** (`Callable`) -- loss function defining local objective
- **optimizer_def** (`Callable`) -- derfintion of server optimizer
- **local_optimizer_def** (`Callable`) -- defintoin of local optimizer
- **lr_scheduler_def** (`Callable`) -- definition of lr scheduler of server optimizer.
- **local_lr_scheduler_def** (`Callable`) -- definition of lr scheduler of local optimizer
- **r2r_local_lr_scheduler_def** (`Callable`) -- definition to schedule lr that is delivered to the clients at each round (deterimined init lr of the client optimizer)
- **batch_size** (*int*) -- batch size of the local trianing
- **test_batch_size** (*int*) -- inference time batch size
- **device** (*str*) -- cpu, cuda, or gpu number

Note:

definition of

- **learning rate schedulers, could be any of the ones defined at** `torch.optim.lr_scheduler` or any other that implements step and get_last_lr methods._schedulers``.
- optimizers, could be any `torch.optim.Optimizer`.
- model, could be any `torch.Module`.
- criterion, could be any `fedsim.scores.Score`.

`deploy()`

return Mapping of name -> parameters_set to test the model

Parameters **server_storage** (*Storage*) -- server storage object.

`init()`

this method is executed only once at the time of instantiating the algorithm object. Here you define your model and whatever needed during the training. Remember to write the outcome of your processing to server_storage for access in other methods.

Note: `*args` and `**kwargs` are directly passed through from algorithm constructor.

Parameters **server_storage** (*Storage*) -- server storage object

`optimize`(*serial_aggregator*, *appendix_aggregator*)

optimize server mdoel(s) and return scores to be reported

Parameters

- **server_storage** (*Storage*) -- server storage object.
- **serial_aggregator** (*SerialAggregator*) -- serial aggregator instance of current round.
- **appendix_aggregator** (*AppendixAggregator*) -- appendix aggregator instance of current round.

Raises `NotImplementedError` -- abstract class to be implemented by child

Returns *Mapping[Hashable, Any]* -- context to be reported

`receive_from_client`(*client_id*, *client_msg*, *train_split_name*, *serial_aggregator*, *appendix_aggregator*)

receive and aggregate info from selected clients

Parameters

- **server_storage** (*Storage*) -- server storage object.
- **client_id** (*int*) -- id of the sender (client)
- **client_msg** (*Mapping[Hashable, Any]*) -- client context that is sent.
- **train_split_name** (*str*) -- name of the training split on clients.
- **aggregator** (*SerialAggregator*) -- aggregator instance to collect info.

Returns *bool* -- success of the aggregation.

Raises `NotImplementedError` -- abstract class to be implemented by child

`report`(*dataloaders*, *rounds*, *scores*, *metric_logger*, *device*, *optimize_reports*, *deployment_points=None*)

test on global data and report info. If a flatten dict of str:Union[int,float] is returned from this function the content is automatically logged using the metric logger (e.g., logall.TensorboardLogger). metric_logger is also passed as an input argument for extra logging operations (non scalar).

Parameters

- **server_storage** (*Storage*) -- server storage object.
- **dataloaders** (*Any*) -- dict of data loaders to test the global model(s)
- **round_scores** (*Dict[str, Dict[str, fedsim.scores.Score]]*) -- dictionary of form {'split_name':{'score_name': score_def}} for global scores to evaluate at the current round.
- **metric_logger** (*Any, optional*) -- the logging object (e.g., logall.TensorboardLogger)
- **device** (*str*) -- 'cuda', 'cpu' or gpu number
- **optimize_reports** (*Mapping[Hashable, Any]*) -- dict returned by optimzier
- **deployment_points** (*Mapping[Hashable, torch.Tensor], optional*) -- output of deploy method

Raises `NotImplementedError` -- abstract class to be implemented by child

`send_to_client`(*client_id*)

returns context to send to the client corresponding to client_id.

> **Warning:** Do not send shared objects like server model if you made any before you deepcopy it.

Parameters

- **server_storage** (*Storage*) -- server storage object.
- **client_id** (*int*) -- id of the receiving client

Raises `NotImplementedError` -- abstract class to be implemented by child

Returns *Mapping[Hashable, Any]* -- the context to be sent in form of a Mapping

`send_to_server`(*rounds*, *storage*, *datasets*, *train_split_name*, *scores*, *epochs*, *criterion*, *train_batch_size*, *inference_batch_size*, *optimizer_def*, *lr_scheduler_def=None*, *device='cuda'*, *ctx=None*, *step_closure=None*)

client operation on the recieved information.

Parameters

- **id** (*int*) -- id of the client
- **rounds** (*int*) -- global round number
- **storage** (*Storage*) -- storage object of the client
- **datasets** (*Dict[str, Iterable]*) -- this comes from Data Manager
- **train_split_name** (*str*) -- string containing name of the training split
- **scores** -- Dict[str, Dict[str, Score]]: dictionary of form {'split_name':{'score_name': Score}} for global scores to evaluate at the current round.
- **epochs** (*int*) -- number of epochs to train
- **criterion** (*Score*) -- citerion, should be a differentiable fedsim.scores.score
- **train_batch_size** (*int*) -- training batch_size
- **inference_batch_size** (*int*) -- inference batch_size
- **optimizer_def** (*float*) -- class for constructing the local optimizer
- **lr_scheduler_def** (*float*) -- class for constructing the local lr scheduler
- **device** (*Union[int, str], optional*) -- Defaults to 'cuda'.
- **ctx** (*Optional[Dict[Hashable, Any]], optional*) -- context reveived.

Returns *Mapping[str, Any]* -- client context to be sent to the server

AvgLogits

`class FedDF`(*data_manager*, *metric_logger*, *num_clients*, *sample_scheme*, *sample_rate*, *model_def*, *epochs*, *criterion_def*, *optimizer_def=functools.partial(<class 'torch.optim.sgd.SGD'>, lr=1.0)*, *local_optimizer_def=functools.partial(<class 'torch.optim.sgd.SGD'>, lr=0.1)*, *lr_scheduler_def=None*, *local_lr_scheduler_def=None*, *r2r_local_lr_scheduler_def=None*, *batch_size=32*, *test_batch_size=64*, *device='cpu'*, **args*, ***kwargs*)

Ensemble Distillation for Robust Model Fusion in Federated Learning.

For further details regarding the algorithm we refer to Ensemble Distillation for Robust Model Fusion in Federated Learning.

Parameters

- **data_manager** (`distributed.data_management.DataManager`) -- data manager
- **metric_logger** (`logall.Logger`) -- metric logger for tracking.
- **num_clients** (*int*) -- number of clients
- **sample_scheme** (`str`) -- mode of sampling clients. Options are `'uniform'` and `'sequential'`
- **sample_rate** (`float`) -- rate of sampling clients
- **model_def** (`torch.Module`) -- definition of for constructing the model

- **epochs** (`int`) -- number of local epochs
- **criterion_def** (`Callable`) -- loss function defining local objective
- **optimizer_def** (`Callable`) -- derfintion of server optimizer
- **local_optimizer_def** (`Callable`) -- defintoin of local optimizer
- **lr_scheduler_def** (`Callable`) -- definition of lr scheduler of server optimizer.
- **local_lr_scheduler_def** (`Callable`) -- definition of lr scheduler of local optimizer
- **r2r_local_lr_scheduler_def** (`Callable`) -- definition to schedule lr that is delivered to the clients at each round (deterimined init lr of the client optimizer)
- **batch_size** (*int*) -- batch size of the local trianing
- **test_batch_size** (*int*) -- inference time batch size
- **device** (*str*) -- cpu, cuda, or gpu number
- **global_train_split** (*str*) -- the name of train split to be used on server
- **global_epochs** (*int*) -- number of training epochs on the server

---

Note:

definition of

- **learning rate schedulers, could be any of the ones defined at** `torch.optim.lr_scheduler` or any other that implements step and get_last_lr methods._schedulers``.
- optimizers, could be any `torch.optim.Optimizer`.
- model, could be any `torch.Module`.
- criterion, could be any `fedsim.scores.Score`.

Warning: this algorithm needs a split for training on the server. This means that the global datasets provided in data manager should include an extra split.

`init`(**args*, ***kwrag*)

this method is executed only once at the time of instantiating the algorithm object. Here you define your model and whatever needed during the training. Remember to write the outcome of your processing to server_storage for access in other methods.

Note: `*args` and `**kwargs` are directly passed through from algorithm constructor.

Parameters **server_storage** (*Storage*) -- server storage object

`optimize`(*serial_aggregator*, *appendix_aggregator*)

optimize server mdoel(s) and return scores to be reported

Parameters

- **server_storage** (*Storage*) -- server storage object.
- **serial_aggregator** (*SerialAggregator*) -- serial aggregator instance of current round.

- **appendix_aggregator** (*AppendixAggregator*) -- appendix aggregator instance of current round.

Raises `NotImplementedError` -- abstract class to be implemented by child

Returns *Mapping[Hashable, Any]* -- context to be reported

`receive_from_client`(*client_id*, *client_msg*, *train_split_name*, *serial_aggregator*, *appendix_aggregator*)

receive and aggregate info from selected clients

Parameters

- **server_storage** (*Storage*) -- server storage object.
- **client_id** (*int*) -- id of the sender (client)
- **client_msg** (*Mapping[Hashable, Any]*) -- client context that is sent.
- **train_split_name** (*str*) -- name of the training split on clients.
- **aggregator** (*SerialAggregator*) -- aggregator instance to collect info.

Returns *bool* -- success of the aggregation.

Raises `NotImplementedError` -- abstract class to be implemented by child

FedDyn

`class FedDyn`(*data_manager*, *metric_logger*, *num_clients*, *sample_scheme*, *sample_rate*, *model_def*, *epochs*, *criterion_def*, *optimizer_def=functools.partial(<class 'torch.optim.sgd.SGD'>, lr=1.0)*, *local_optimizer_def=functools.partial(<class 'torch.optim.sgd.SGD'>, lr=0.1)*, *lr_scheduler_def=None*, *local_lr_scheduler_def=None*, *r2r_local_lr_scheduler_def=None*, *batch_size=32*, *test_batch_size=64*, *device='cpu'*, **args*, ***kwargs*)

Implements FedDyn algorithm for centralized FL.

For further details regarding the algorithm we refer to Federated Learning Based on Dynamic Regularization.

Parameters

- **data_manager** (`distributed.data_management.DataManager`) -- data manager
- **metric_logger** (`logall.Logger`) -- metric logger for tracking.
- **num_clients** (*int*) -- number of clients
- **sample_scheme** (`str`) -- mode of sampling clients. Options are `'uniform'` and `'sequential'`
- **sample_rate** (`float`) -- rate of sampling clients
- **model_def** (`torch.Module`) -- definition of for constructing the model
- **epochs** (`int`) -- number of local epochs
- **criterion_def** (`Callable`) -- loss function defining local objective
- **optimizer_def** (`Callable`) -- derfintion of server optimizer
- **local_optimizer_def** (`Callable`) -- defintoin of local optimizer
- **lr_scheduler_def** (`Callable`) -- definition of lr scheduler of server optimizer.
- **local_lr_scheduler_def** (`Callable`) -- definition of lr scheduler of local optimizer

- **r2r_local_lr_scheduler_def** (`Callable`) -- definition to schedule lr that is delivered to the clients at each round (deterimined init lr of the client optimizer)
- **batch_size** (*int*) -- batch size of the local trianing
- **test_batch_size** (*int*) -- inference time batch size
- **device** (*str*) -- cpu, cuda, or gpu number
- **alpha** (*float*) -- FedDyn's α hyper-parameter for local regularization

---

Note:

definition of

- **learning rate schedulers, could be any of the ones defined at** `torch.optim.lr_scheduler` or any other that implements step and get_last_lr methods._schedulers``.
- optimizers, could be any `torch.optim.Optimizer`.
- model, could be any `torch.Module`.
- criterion, could be any `fedsim.scores.Score`.

`deploy()`

return Mapping of name -> parameters_set to test the model

Parameters **server_storage** (*Storage*) -- server storage object.

`init(*args, **kwrag)`

this method is executed only once at the time of instantiating the algorithm object. Here you define your model and whatever needed during the training. Remember to write the outcome of your processing to server_storage for access in other methods.

Note: `*args` and `**kwargs` are directly passed through from algorithm constructor.

Parameters **server_storage** (*Storage*) -- server storage object

`optimize(serial_aggregator, appendix_aggregator)`

optimize server mdoel(s) and return scores to be reported

Parameters

- **server_storage** (*Storage*) -- server storage object.
- **serial_aggregator** (*SerialAggregator*) -- serial aggregator instance of current round.
- **appendix_aggregator** (*AppendixAggregator*) -- appendix aggregator instance of current round.

Raises `NotImplementedError` -- abstract class to be implemented by child

Returns *Mapping[Hashable, Any]* -- context to be reported

`receive_from_client(client_id, client_msg, train_split_name, serial_aggregator, appendix_aggregator)`

receive and aggregate info from selected clients

Parameters

- **server_storage** (*Storage*) -- server storage object.

- **client_id** (*int*) -- id of the sender (client)
- **client_msg** (*Mapping[Hashable, Any]*) -- client context that is sent.
- **train_split_name** (*str*) -- name of the training split on clients.
- **aggregator** (*SerialAggregator*) -- aggregator instance to collect info.

Returns *bool* -- success of the aggregation.

Raises `NotImplementedError` -- abstract class to be implemented by child

`send_to_client`(*client_id*)

returns context to send to the client corresponding to client_id.

> **Warning:** Do not send shared objects like server model if you made any before you deepcopy it.

Parameters

- **server_storage** (*Storage*) -- server storage object.
- **client_id** (*int*) -- id of the receiving client

Raises `NotImplementedError` -- abstract class to be implemented by child

Returns *Mapping[Hashable, Any]* -- the context to be sent in form of a Mapping

`send_to_server`(*rounds*, *storage*, *datasets*, *train_split_name*, *metrics*, *epochs*, *criterion*, *train_batch_size*, *inference_batch_size*, *optimizer_def*, *lr_scheduler_def=None*, *device='cuda'*, *ctx=None*, *step_closure=None*)

client operation on the recieved information.

Parameters

- **id** (*int*) -- id of the client
- **rounds** (*int*) -- global round number
- **storage** (*Storage*) -- storage object of the client
- **datasets** (*Dict[str, Iterable]*) -- this comes from Data Manager
- **train_split_name** (*str*) -- string containing name of the training split
- **scores** -- Dict[str, Dict[str, Score]]: dictionary of form {'split_name':{'score_name': Score}} for global scores to evaluate at the current round.
- **epochs** (*int*) -- number of epochs to train
- **criterion** (*Score*) -- citerion, should be a differentiable fedsim.scores.score
- **train_batch_size** (*int*) -- training batch_size
- **inference_batch_size** (*int*) -- inference batch_size
- **optimizer_def** (*float*) -- class for constructing the local optimizer
- **lr_scheduler_def** (*float*) -- class for constructing the local lr scheduler
- **device** (*Union[int, str], optional*) -- Defaults to 'cuda'.
- **ctx** (*Optional[Dict[Hashable, Any]], optional*) -- context reveived.

Returns *Mapping[str, Any]* -- client context to be sent to the server

FedNova

class FedNova(*data_manager*, *metric_logger*, *num_clients*, *sample_scheme*, *sample_rate*, *model_def*, *epochs*, *criterion_def*, *optimizer_def=functools.partial(<class 'torch.optim.sgd.SGD'>, lr=1.0)*, *local_optimizer_def=functools.partial(<class 'torch.optim.sgd.SGD'>, lr=0.1)*, *lr_scheduler_def=None*, *local_lr_scheduler_def=None*, *r2r_local_lr_scheduler_def=None*, *batch_size=32*, *test_batch_size=64*, *device='cpu'*, **args*, ***kwargs*)

Implements FedNova algorithm for centralized FL.

For further details regarding the algorithm we refer to Tackling the Objective Inconsistency Problem in Heterogeneous Federated Optimization.

Parameters

- **data_manager** (`distributed.data_management.DataManager`) -- data manager
- **metric_logger** (`logall.Logger`) -- metric logger for tracking.
- **num_clients** (*int*) -- number of clients
- **sample_scheme** (`str`) -- mode of sampling clients. Options are `'uniform'` and `'sequential'`
- **sample_rate** (`float`) -- rate of sampling clients
- **model_def** (`torch.Module`) -- definition of for constructing the model
- **epochs** (`int`) -- number of local epochs
- **criterion_def** (`Callable`) -- loss function defining local objective
- **optimizer_def** (`Callable`) -- derfintion of server optimizer
- **local_optimizer_def** (`Callable`) -- defintoin of local optimizer
- **lr_scheduler_def** (`Callable`) -- definition of lr scheduler of server optimizer.
- **local_lr_scheduler_def** (`Callable`) -- definition of lr scheduler of local optimizer
- **r2r_local_lr_scheduler_def** (`Callable`) -- definition to schedule lr that is delivered to the clients at each round (deterimined init lr of the client optimizer)
- **batch_size** (*int*) -- batch size of the local trianing
- **test_batch_size** (*int*) -- inference time batch size
- **device** (*str*) -- cpu, cuda, or gpu number

Note:

definition of

- **learning rate schedulers, could be any of the ones defined at** `torch.optim.lr_scheduler` or any other that implements step and get_last_lr methods._schedulers``.
- optimizers, could be any `torch.optim.Optimizer`.
- model, could be any `torch.Module`.
- criterion, could be any `fedsim.scores.Score`.

`receive_from_client`(*client_id*, *client_msg*, *train_split_name*, *serial_aggregator*, *appendix_aggregator*)

receive and aggregate info from selected clients

Parameters

- **server_storage** (*Storage*) -- server storage object.
- **client_id** (*int*) -- id of the sender (client)
- **client_msg** (*Mapping[Hashable, Any]*) -- client context that is sent.
- **train_split_name** (*str*) -- name of the training split on clients.
- **aggregator** (*SerialAggregator*) -- aggregator instance to collect info.

Returns *bool* -- success of the aggregation.

Raises `NotImplementedError` -- abstract class to be implemented by child

FedProx

`class FedProx`(*data_manager*, *metric_logger*, *num_clients*, *sample_scheme*, *sample_rate*, *model_def*, *epochs*, *criterion_def*, *optimizer_def=functools.partial(<class 'torch.optim.sgd.SGD'>, lr=1.0)*, *local_optimizer_def=functools.partial(<class 'torch.optim.sgd.SGD'>, lr=0.1)*, *lr_scheduler_def=None*, *local_lr_scheduler_def=None*, *r2r_local_lr_scheduler_def=None*, *batch_size=32*, *test_batch_size=64*, *device='cpu'*, **args*, ***kwargs*)

Implements FedProx algorithm for centralized FL.

For further details regarding the algorithm we refer to Federated Optimization in Heterogeneous Networks.

Parameters

- **data_manager** (`distributed.data_management.DataManager`) -- data manager
- **metric_logger** (`logall.Logger`) -- metric logger for tracking.
- **num_clients** (*int*) -- number of clients
- **sample_scheme** (`str`) -- mode of sampling clients. Options are `'uniform'` and `'sequential'`
- **sample_rate** (`float`) -- rate of sampling clients
- **model_def** (`torch.Module`) -- definition of for constructing the model
- **epochs** (`int`) -- number of local epochs
- **criterion_def** (`Callable`) -- loss function defining local objective
- **optimizer_def** (`Callable`) -- derfintion of server optimizer
- **local_optimizer_def** (`Callable`) -- defintoin of local optimizer
- **lr_scheduler_def** (`Callable`) -- definition of lr scheduler of server optimizer.
- **local_lr_scheduler_def** (`Callable`) -- definition of lr scheduler of local optimizer
- **r2r_local_lr_scheduler_def** (`Callable`) -- definition to schedule lr that is delivered to the clients at each round (deterimined init lr of the client optimizer)
- **batch_size** (*int*) -- batch size of the local trianing
- **test_batch_size** (*int*) -- inference time batch size
- **device** (*str*) -- cpu, cuda, or gpu number

- **mu** (*float*) -- FedProx's μ hyper-parameter for local regularization

---

Note:

definition of

- **learning rate schedulers, could be any of the ones defined at** `torch.optim.lr_scheduler` or any other that implements step and get_last_lr methods._schedulers``.
- optimizers, could be any `torch.optim.Optimizer`.
- model, could be any `torch.Module`.
- criterion, could be any `fedsim.scores.Score`.

`init`(**args*, ***kwrag*)

this method is executed only once at the time of instantiating the algorithm object. Here you define your model and whatever needed during the training. Remember to write the outcome of your processing to server_storage for access in other methods.

Note: `*args` and `**kwargs` are directly passed through from algorithm constructor.

Parameters **server_storage** (*Storage*) -- server storage object

`send_to_client`(*client_id*)

returns context to send to the client corresponding to client_id.

Warning: Do not send shared objects like server model if you made any before you deepcopy it.

Parameters

- **server_storage** (*Storage*) -- server storage object.
- **client_id** (*int*) -- id of the receiving client

Raises `NotImplementedError` -- abstract class to be implemented by child

Returns *Mapping[Hashable, Any]* -- the context to be sent in form of a Mapping

`send_to_server`(*rounds*, *storage*, *datasets*, *train_split_name*, *scores*, *epochs*, *criterion*, *train_batch_size*, *inference_batch_size*, *optimizer_def*, *lr_scheduler_def=None*, *device='cuda'*, *ctx=None*, *step_closure=None*)

client operation on the recieved information.

Parameters

- **id** (*int*) -- id of the client
- **rounds** (*int*) -- global round number
- **storage** (*Storage*) -- storage object of the client
- **datasets** (*Dict[str, Iterable]*) -- this comes from Data Manager
- **train_split_name** (*str*) -- string containing name of the training split

- **scores** -- Dict[str, Dict[str, Score]]: dictionary of form {'split_name':{'score_name': Score}} for global scores to evaluate at the current round.
- **epochs** (*int*) -- number of epochs to train
- **criterion** (*Score*) -- citerion, should be a differentiable fedsim.scores.score
- **train_batch_size** (*int*) -- training batch_size
- **inference_batch_size** (*int*) -- inference batch_size
- **optimizer_def** (*float*) -- class for constructing the local optimizer
- **lr_scheduler_def** (*float*) -- class for constructing the local lr scheduler
- **device** (*Union[int, str], optional*) -- Defaults to 'cuda'.
- **ctx** (*Optional[Dict[Hashable, Any]], optional*) -- context reveived.

Returns *Mapping[str, Any]* -- client context to be sent to the server

Distributed Centralized Trainign Utils

`serial_aggregation`(*server_storage*, *client_id*, *client_msg*, *train_split_name*, *aggregator*, *train_weight=None*, *other_weight=None*, *purge_msg=True*)

To serially aggregate received message from a client

Parameters

- **server_storage** (*Storage*) -- server storage object
- **client_id** (*int*) -- client id.
- **client_msg** (*Mapping*) -- client message.
- **train_split_name** (*str*) -- name of the training split on clients
- **aggregator** (*SerialAggregator*) -- a serial aggregator to accumulate info.
- **train_weight** (*float, optional*) -- aggregation weight for trianing parameters. If not specified, uses sample number. Defaults to None.
- **other_weight** (*float, optional*) -- aggregation weight for any other factor/metric. If not specified, uses sample number. Defaults to None.

Returns *bool* -- success of aggregation.

Centralized Federated Learnming Algorithm

`class CentralFLAlgorithm`(*data_manager*, *metric_logger*, *num_clients*, *sample_scheme*, *sample_rate*, *model_def*, *epochs*, *criterion_def*, *optimizer_def=functools.partial(<class 'torch.optim.sgd.SGD'>, lr=1.0)*, *local_optimizer_def=functools.partial(<class 'torch.optim.sgd.SGD'>, lr=0.1)*, *lr_scheduler_def=None*, *local_lr_scheduler_def=None*, *r2r_local_lr_scheduler_def=None*, *batch_size=32*, *test_batch_size=64*, *device='cpu'*, **args*, ***kwargs*)

Base class for centralized FL algorithm.

Parameters

- **data_manager** (`distributed.data_management.DataManager`) -- data manager
- **metric_logger** (`logall.Logger`) -- metric logger for tracking.

- **num_clients** (*int*) -- number of clients
- **sample_scheme** (`str`) -- mode of sampling clients. Options are `'uniform'` and `'sequential'`
- **sample_rate** (`float`) -- rate of sampling clients
- **model_def** (`torch.Module`) -- definition of for constructing the model
- **epochs** (`int`) -- number of local epochs
- **criterion_def** (`Callable`) -- loss function defining local objective
- **optimizer_def** (`Callable`) -- derfintion of server optimizer
- **local_optimizer_def** (`Callable`) -- defintoin of local optimizer
- **lr_scheduler_def** (`Callable`) -- definition of lr scheduler of server optimizer.
- **local_lr_scheduler_def** (`Callable`) -- definition of lr scheduler of local optimizer
- **r2r_local_lr_scheduler_def** (`Callable`) -- definition to schedule lr that is delivered to the clients at each round (deterimined init lr of the client optimizer)
- **batch_size** (*int*) -- batch size of the local trianing
- **test_batch_size** (*int*) -- inference time batch size
- **device** (*str*) -- cpu, cuda, or gpu number

Note: definition of * learning rate schedulers, could be any of the ones defined at `torch.optim.lr_scheduler` or any other that implements step and get_last_lr methods. * optimizers, could be any `torch.optim.Optimizer`. * model, could be any `torch.Module`. * criterion, could be any `fedsim.losses`.

Architecture:

`at_round_end`(*score_aggregator:* fedsim.utils.aggregators.AppendixAggregator) → None

to inject code at the end of rounds in training loop

Parameters

- **server_storage** (*Storage*) -- server storage object.
- **score_aggregator** (*AppendixAggregator*) -- contains the aggregated scores

`at_round_start`() → None

to inject code at the beginning of rounds in training loop.

Parameters **server_storage** (*Storage*) -- server storage object.

`deploy`() → Optional[Mapping[Hashable, Any]]

return Mapping of name -> parameters_set to test the model

Parameters **server_storage** (*Storage*) -- server storage object.

`get_device`() → str

To get the device name or number

Returns *str* -- device name or number

`get_global_loader_split`(*split_name*) → Iterable

To get the data loader for a specific global split.

Parameters **split_name** (*Hashable*) -- split name.

Returns *Iterable* -- data loader for global split <split_name>

`get_global_scores`() → Dict[str, Any]

To instantiate and get global scores that have to be measured in the current round (log frequencies are matched).

Returns *Dict[str, Any]* -- mapping of name:score

`get_global_split_scores`(*split_name*) → Dict[str, Any]

To instantiate and get global scores that have to be measured in the current round (log frequencies are matched) for a specific data split.

Parameters **split_name** (*Hashable*) -- name of the global data split

Returns

Dict[str, Any] --

mapping of name:score. If no score is listed for the given split, None is returned.

`get_local_scores`() → Dict[str, Any]

To instantiate and get local scores that have to be measured in the current round (log frequencies are matched).

Returns

Dict[str, Any] --

mapping of name:score. If no score is listed for the given split, None is returned.

`get_local_split_scores`(*split_name*) → Dict[str, Any]

To instantiate and get local scores that have to be measured in the current round (log frequencies are matched) for a specific data split.

Parameters **split_name** (*Hashable*) -- name of the global data split

Returns *Dict[str, Any]* -- mapping of name:score

`get_model_def`()

To get the definition of the model so that one can instantiate it by calling.

Returns *Callable* -- definition of the model. To instantiate, you may call the returned value with paranthesis in front.

`get_round_number`()

To get the current round number, starting from zero.

Returns *int* -- current round number, starting from zero.

`get_server_storage`()

To access the public configs of the server.

Returns *Storage* -- public server storage.

`get_train_split_name`()

To get the name of the split used to perform local training.

Returns *Hashable* -- name of the split used for local training.

`hook_global_score`(*score_def*, *score_name*, *split_name*) → None

To hook a score measurment on global data.

Parameters

- **score_def** (*Callable*) -- definition of the score used to make new instances of. the list of existing scores could be found under `fedsim.scores`.
- **score_name** (*Hashable*) -- name of the score to show up in the logs.
- **split_name** (*Hashable*) -- name of the data split to apply the measurement on.

`hook_local_score`(*score_def*, *score_name*, *split_name*) → None

To hook a score measurment on local data.

Parameters

- **score_def** (*Callable*) -- definition of the score used to make new instances of. the list of existing scores could be found under `fedsim.scores`.
- **score_name** (*Hashable*) -- name of the score to show up in the logs.
- **split_name** (*Hashable*) -- name of the data split to apply the measurement on.

`init`(**args*, ***kwargs*) → None

this method is executed only once at the time of instantiating the algorithm object. Here you define your model and whatever needed during the training. Remember to write the outcome of your processing to server_storage for access in other methods.

Note: `*args` and `**kwargs` are directly passed through from algorithm constructor.

Parameters **server_storage** (*Storage*) -- server storage object

`optimize`(*serial_aggregator:* fedsim.utils.aggregators.SerialAggregator, *appendix_aggregator:* fedsim.utils.aggregators.AppendixAggregator) → Mapping[Hashable, Any]

optimize server mdoel(s) and return scores to be reported

Parameters

- **server_storage** (*Storage*) -- server storage object.
- **serial_aggregator** (*SerialAggregator*) -- serial aggregator instance of current round.
- **appendix_aggregator** (*AppendixAggregator*) -- appendix aggregator instance of current round.

Raises `NotImplementedError` -- abstract class to be implemented by child

Returns *Mapping[Hashable, Any]* -- context to be reported

`receive_from_client`(*client_id: int*, *client_msg: Mapping[Hashable, Any]*, *train_split_name: str*, *serial_aggregator:* fedsim.utils.aggregators.SerialAggregator, *appendix_aggregator:* fedsim.utils.aggregators.AppendixAggregator) → bool

receive and aggregate info from selected clients

Parameters

- **server_storage** (*Storage*) -- server storage object.
- **client_id** (*int*) -- id of the sender (client)
- **client_msg** (*Mapping[Hashable, Any]*) -- client context that is sent.

- **train_split_name** (*str*) -- name of the training split on clients.
- **aggregator** (*SerialAggregator*) -- aggregator instance to collect info.

Returns *bool* -- success of the aggregation.

Raises `NotImplementedError` -- abstract class to be implemented by child

`report`(*dataloaders: Dict[str, Any]*, *round_scores: Dict[str, Dict[str, Any]]*, *metric_logger: Optional[Any]*, *device: str*, *optimize_reports: Mapping[Hashable, Any]*, *deployment_points: Optional[Mapping[Hashable, torch.Tensor]] = None*) → Dict[str, Union[int, float]]

test on global data and report info. If a flatten dict of str:Union[int,float] is returned from this function the content is automatically logged using the metric logger (e.g., logall.TensorboardLogger). metric_logger is also passed as an input argument for extra logging operations (non scalar).

Parameters

- **server_storage** (*Storage*) -- server storage object.
- **dataloaders** (*Any*) -- dict of data loaders to test the global model(s)
- **round_scores** (*Dict[str, Dict[str, fedsim.scores.Score]]*) -- dictionary of form {'split_name':{'score_name': score_def}} for global scores to evaluate at the current round.
- **metric_logger** (*Any, optional*) -- the logging object (e.g., logall.TensorboardLogger)
- **device** (*str*) -- 'cuda', 'cpu' or gpu number
- **optimize_reports** (*Mapping[Hashable, Any]*) -- dict returned by optimzier
- **deployment_points** (*Mapping[Hashable, torch.Tensor], optional*) -- output of deploy method

Raises `NotImplementedError` -- abstract class to be implemented by child

`send_to_client`(*client_id: int*) → Mapping[Hashable, Any]

returns context to send to the client corresponding to client_id.

> **Warning:** Do not send shared objects like server model if you made any before you deepcopy it.

Parameters

- **server_storage** (*Storage*) -- server storage object.
- **client_id** (*int*) -- id of the receiving client

Raises `NotImplementedError` -- abstract class to be implemented by child

Returns *Mapping[Hashable, Any]* -- the context to be sent in form of a Mapping

`send_to_server`(*rounds: int*, *storage: Dict[Hashable, Any]*, *datasets: Dict[str, Iterable]*, *train_split_name: str*, *scores: Dict[str, Dict[str, Any]]*, *epochs: int*, *criterion: torch.nn.modules.module.Module*, *train_batch_size: int*, *inference_batch_size: int*, *optimizer_def: Callable*, *lr_scheduler_def: Optional[Callable] = None*, *device: Union[int, str] = 'cuda'*, *ctx: Optional[Dict[Hashable, Any]] = None*, **args*, ***kwargs*) → Mapping[str, Any]

client operation on the recieved information.

Parameters

- **id** (*int*) -- id of the client

- **rounds** (*int*) -- global round number
- **storage** (*Storage*) -- storage object of the client
- **datasets** (*Dict[str, Iterable]*) -- this comes from Data Manager
- **train_split_name** (*str*) -- string containing name of the training split
- **scores** -- Dict[str, Dict[str, Score]]: dictionary of form {'split_name':{'score_name': Score}} for global scores to evaluate at the current round.
- **epochs** (*int*) -- number of epochs to train
- **criterion** (*Score*) -- citerion, should be a differentiable fedsim.scores.score
- **train_batch_size** (*int*) -- training batch_size
- **inference_batch_size** (*int*) -- inference batch_size
- **optimizer_def** (*float*) -- class for constructing the local optimizer
- **lr_scheduler_def** (*float*) -- class for constructing the local lr scheduler
- **device** (*Union[int, str], optional*) -- Defaults to 'cuda'.
- **ctx** (*Optional[Dict[Hashable, Any]], optional*) -- context reveived.

Returns *Mapping[str, Any]* -- client context to be sent to the server

`train`(*rounds: int*, *num_score_report_point: Optional[int] = None*, *train_split_name='train'*) → Optional[Dict[str, Optional[float]]]

loop over the learning pipeline of distributed algorithm for given number of rounds.

Note:

- The clients metrics are reported in the form of clients.{metric_name}.
- **The server metrics (scores results) are reported in the form of** server.{deployment_point}.{metric_name}

Parameters

- **rounds** (*int*) -- number of rounds to train.
- **num_score_report_point** (*int*) -- limits num of points to return reports.
- **train_split_name** (*str*) -- local split name to perform training on. Defaults to 'train'.

Returns *Optional[Dict[str, Union[float]]]* -- collected score metrics.

Data Management

A Basic Data Manager

`class BasicDataManager`(*root='data'*, *dataset='mnist'*, *num_partitions=500*, *rule='iid'*, *sample_balance=0.0*, *label_balance=1.0*, *local_test_portion=0.0*, *global_valid_portion=0.0*, *seed=10*, *save_dir='partitions'*)

A basic data manager for partitioning the data. Currecntly three rules of partitioning are supported:

- **iid:** same label distribution among clients. sample balance determines quota of each client samples from a lognorm distribution.

- **dir:** Dirichlete distribution with concentration parameter given by label_balance determines label balance of each client. sample balance determines quota of each client samples from a lognorm distribution.
- **exclusive:** samples corresponding to each label are randomly splitted to k clients where k = total_sample_size * label_balance. sample_balance determines the way this split happens (quota). This rule also is know as "shards splitting".

Parameters

- **root** (*str*) -- root dir of the dataset to partition
- **dataset** (*str*) -- name of the dataset
- **num_clients** (*int*) -- number of partitions or clients
- **rule** (*str*) -- rule of partitioning
- **sample_balance** (*float*) -- balance of number of samples among clients
- **label_balance** (*float*) -- balance of the labels on each clietns
- **local_test_portion** (*float*) -- portion of local test set from trian
- **global_valid_portion** (*float*) -- portion of global valid split. What remains from global samples goes to the test split.
- **seed** (*int*) -- random seed of partitioning
- **save_dir** (*str, optional*) -- dir to save partitioned indices.

`get_identifiers()`

Returns identifiers to be used for saving the partition info.

Returns *Sequence[str]* -- a sequence of str identifing class instance

`make_datasets(root)`

makes and returns local and global dataset objects. The created datasets do not need a transform as recompiled datasets with separately provided transforms on the fly.

Parameters

- **dataset_name** (*str*) -- name of the dataset.
- **root** (*str*) -- directory to download and manipulate data.

Returns *Tuple[object, object]* -- local and global dataset

`make_transforms()`

make and return the dataset trasformations for local and global split.

Returns

Tuple[Dict[str, Callable], Dict[str, Callable]] --

tuple of two dictionaries, first, the local transform mapping and second the global transform mapping.

`partition_global_data(dataset)`

partitions global data indices into splits (e.g., train, test, ...).

Parameters **dataset** (*object*) -- global dataset

Returns *Dict[str, Iterable[int]]* -- dictionary of {split:example indices of global dataset}.

`partition_local_data`(*dataset*)

partitions local data indices into client-indexed Iterable.

Parameters **dataset** (*object*) -- local dataset

Returns *Dict[str, Iterable[Iterable[int]]]* -- dictionary of {split:client-indexed iterables of example indices}.

Data Manager

`class DataManager`(*root*, *seed*, *save_dir=None*)

DataManager base class. Any other Data Manager is inherited from this class. There are four abstract class methods that child classes should implement: get_identifiers, make_datasets, make_transforms, partition_local_data.

> **Warning:** when inheritted, super should be called at the end of the constructor because the abstract classes are called in super's constructor!

Parameters

- **root** (*str*) -- root dir of the dataset to partition
- **seed** (*int*) -- random seed of partitioning
- **save_dir** (*str, optional*) -- path to save partitioned indices.

`get_global_dataset`() → Dict[str, torch.utils.data.dataset.Dataset]

returns the global dataset

Returns *Dict[str, Dataset]* -- global dataset for each split

`get_global_splits_names`()

returns name of the global splits (train, test, etc.)

Returns *List[str]* -- list of global split names

`get_group_dataset`(*ids: Iterable[int]*) → Dict[str, torch.utils.data.dataset.Dataset]

returns the local dataset corresponding to a group of given partition ids

Parameters **ids** (*Iterable[int]*) -- a list or tuple of partition ids

Returns *Dict[str, Dataset]* -- a mapping of split_name: dataset

`get_identifiers`() → Sequence[str]

Returns identifiers to be used for saving the partition info.

Raises `NotImplementedError` -- this abstract method should be implemented by child classes

Returns *Sequence[str]* -- a sequence of str identifing class instance

`get_local_dataset`(*id: int*) → Dict[str, torch.utils.data.dataset.Dataset]

returns the local dataset corresponding to a given partition id

Parameters **id** (*int*) -- partition id

Returns *Dict[str, Dataset]* -- a mapping of split_name: dataset

`get_local_splits_names()`

returns name of the local splits (train, test, etc.)

Returns *List[str]* -- list of local split names

`get_oracle_dataset()` → Dict[str, torch.utils.data.dataset.Dataset]

returns all of the local datasets stacked up.

Returns *Dict[str, Dataset]* -- Oracle dataset for each split

`get_partitioning_name()` → str

returns unique name of the DataManager instance. .. note:: This method can help store and retrieval of the partitioning indices, so the experiments could reproduced on a machine.

Returns *str* -- a unique name for the DataManager instance.

`make_datasets`(*root: str*) → Tuple[object, object]

makes and returns local and global dataset objects. The created datasets do not need a transform as recompiled datasets with separately provided transforms on the fly.

Parameters

- **dataset_name** (*str*) -- name of the dataset.
- **root** (*str*) -- directory to download and manipulate data.

Raises `NotImplementedError` -- this abstract method should be implemented by child classes

Returns *Tuple[object, object]* -- local and global dataset

`make_transforms()` → Tuple[object, object]

make and return the dataset trasformations for local and global split.

Raises `NotImplementedError` -- this abstract method should be implemented by child classes

Returns

Tuple[Dict[str, Callable], Dict[str, Callable]] --

tuple of two dictionaries, first, the local transform mapping and second the global transform mapping.

`partition_global_data`(*dataset: object*) → Dict[str, Iterable[int]]

partitions global data indices into splits (e.g., train, test, ...).

Parameters **dataset** (*object*) -- global dataset

Returns *Dict[str, Iterable[int]]* -- dictionary of {split:example indices of global dataset}.

`partition_local_data`(*dataset: object*) → Dict[str, Iterable[Iterable[int]]]

partitions local data indices into client-indexed Iterable.

Parameters **dataset** (*object*) -- local dataset

Raises `NotImplementedError` -- this abstract method should be implemented by child classes

Returns *Dict[str, Iterable[Iterable[int]]]* -- dictionary of {split:client-indexed iterables of example indices}.

Data Management Utils

class Subset(*dataset*, *indices*, *transform=None*)

Subset of a dataset at specified indices.

Parameters

- **dataset** (*Dataset*) -- The whole Dataset
- **indices** (*sequence*) -- Indices in the whole set selected for subset.

Decentralized Distributed Learning

This package is emptt in this version.

3.1.2 Local

Local Training and Inference

Provides the basic definitions for local trainign and inference.

Local Inference

Inference for local client

local_inference(*model*, *data_loader*, *scores*, *device='cpu'*, *transform_y=None*)

to test the performance of a model on a test set.

Parameters

- **model** (*Module*) -- model to get the predictions from
- **data_loader** (*Iterable*) -- inference data loader.
- **scores** (*Dict[str, Score]*) -- scores to evaluate
- **device** (*str, optional*) -- device to load the data into ("cpu", "cuda", or device ordinal number). This must be the same device as the one model parameters are loaded into. Defaults to "cpu".
- **transform_y** (*Callable, optional*) -- a function that takes raw labels and modifies them. Defaults to None.

Returns *int* -- number of samples the evaluation is done for.

Step Closures

default_step_closure(*x*, *y*, *model*, *criterion*, *optimizer*, *scores*, *max_grad_norm=1000*, *device='cpu'*, *transform_grads=None*, *transform_y=None*, ***kwargs*)

one step of local training including: * prepare mini batch of the data * forward pass * loss calculation * backward pass * transfor and modify the gradients * take optimization step * evaluate scores on the training mini-batch batch.

Parameters

- **x** (*Tensor*) -- inputs

- **y** (*Tensor*) -- labels
- **model** (*Module*) -- model
- **criterion** (*Callable*) -- loss criterion
- **optimizer** (*Optimizer*) -- optimizer chosen and instanciated from classes under `torch.optim`.
- **scores** -- Dict[str, Score]: dictionary of form str: Score to evaluate at the end of the closure.
- **max_grad_norm** (*int, optional*) -- to clip the norm of the gradients. Defaults to 1000.
- **device** (*str, optional*) -- device to load the data into ("cpu", "cuda", or device ordinal number). This must be the same device as the one model parameters are loaded into. Defaults to "cpu".
- **transform_grads** (*Callable, optional*) -- A function the takes the model and modified the gradients of the parameters. Defaults to None.
- **transform_y** (*Callable, optional*) -- a function that takes raw labels and modifies them. Defaults to None.

Returns *Tensor* -- loss value obtained from the forward pass.

Local Training

Training for local client

local_train(*model*, *train_data_loader*, *epochs*, *steps*, *criterion*, *optimizer*, *lr_scheduler=None*, *device='cpu'*, *step_closure=<function default_step_closure>*, *scores=None*, *max_grad_norm=1000*, ***step_ctx*)

local training

Parameters

- **model** (*Module*) -- model to use for getting the predictions.
- **train_data_loader** (*Iterable*) -- trianing data loader.
- **epochs** (*int*) -- number of local epochs.
- **steps** (*int*) -- number of optimization epochs after the final epoch.
- **criterion** (*Callable*) -- loss criterion.
- **optimizer** (*Optimizer*) -- a torch optimizer.
- **lr_scheduler** (*Any, optional*) -- a torch Learning rate scheduler. Defaults to None.
- **device** (*str, optional*) -- device to load the data into ("cpu", "cuda", or device ordinal number). This must be the same device as the one model parameters are loaded into. Defaults to "cpu".
- **step_closure** (*Callable, optional*) -- step closure for an optimization step. Defaults to default_step_closure.
- **scores** (*Dict[str, Score], optional*) -- a dictionary of str:Score. Defaults to None.
- **max_grad_norm** (*int, optional*) -- to clip the norm of the gradients. Defaults to 1000.

Returns

Tuple[int, int, bool] --

tuple of number of training samples, number of optimization steps, divergence.

3.1.3 Models

Simple Model Architectures

In this file, you can find a number of models that are commonly used in FL community. These models are used in Communication-Efficient Learning of Deep Networks from Decentralized Data.

`class SimpleCNN(`*num_classes=10*, *num_channels=1*, *in_height=28*, *in_width=28*, *num_filters1=32*, *num_filters2=64*, *feature_size=512*`)`

A simple two layer CNN Perceptron.

Parameters

- **num_classes** (*int, optional*) -- number of classes. Defaults to 10. Assigning None or a negative integer means no classifier.
- **num_channels** (*int, optional*) -- number of channels of input. Defaults to 1.
- **in_height** (*int, optional*) -- input height to resize to. Defaults to 28.
- **in_width** (*int, optional*) -- input width to resize to. Defaults to 28.
- **feature_size** (*int, optional*) -- number of features. Defaults to 512.

`forward(`x`)`

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`get_features(`x`)`

Gets the extracted features. Goes through all cells except the classifier.

Parameters **x** (*Tensor*) -- input tensor with shape $(N \times C \times D_1 \times D_2 \times \cdots \times D_n)$ where `N` is batch size and `C` is dtermined by `num_channels`.

Returns

Tensor --

output tensor with shape $(N \times O)$ where `O` is determined by `feature_size`

`training: bool`

`class SimpleCNN2(`*num_classes=10*, *num_channels=3*, *in_height=24*, *in_width=24*, *num_filters1=64*, *num_filters2=64*, *hidden_size=384*, *feature_size=192*`)`

A simple two layer CNN Perceptron. This is similar to CNN model in McMahan's FedAvg paper.

Parameters

- **num_classes** (*int, optional*) -- number of classes. Defaults to 10. Assigning None or a negative integer means no classifier.
- **num_channels** (*int, optional*) -- number of channels of input. Defaults to 1.
- **in_height** (*int, optional*) -- input height to resize to. Defaults to 28.
- **in_width** (*int, optional*) -- input width to resize to. Defaults to 28.

- **hidden_size** (*int, optional*) -- number of hidden neurons. Defaults to 384.
- **feature_size** (*int, optional*) -- number of features. Defaults to 192.

`forward`(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`get_features`(*x*)

Gets the extracted features. Goes through all cells except the classifier.

Parameters **x** (*Tensor*) -- input tensor with shape $(N \times C \times D_1 \times D_2 \times \cdots \times D_n)$ where `N` is batch size and `C` is dtermined by `num_channels`.

Returns

Tensor --

output tensor with shape $(N \times O)$ where `O` is determined by `feature_size`

`training: bool`

`class SimpleMLP`(*num_classes=10*, *num_channels=1*, *in_height=28*, *in_width=28*, *feature_size=200*)

A simple two layer Multi-Layer Perceptron. This is referred to as 2NN in McMahan's FedAvg paper.

Parameters

- **num_classes** (*int, optional*) -- number of classes. Defaults to 10. Assigning None or a negative integer means no classifier.
- **num_channels** (*int, optional*) -- number of channels of input. Defaults to 1.
- **in_height** (*int, optional*) -- input height to resize to. Defaults to 28.
- **in_width** (*int, optional*) -- input width to resize to. Defaults to 28.
- **feature_size** (*int, optional*) -- number of features. Defaults to 200.

`forward`(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`get_features`(*x*)

Gets the extracted features. Goes through all cells except the classifier.

Parameters **x** (*Tensor*) -- input tensor with shape $(N \times C \times D_1 \times D_2 \times \cdots \times D_n)$ where `N` is batch size and `C` is dtermined by `num_channels`.

Returns

Tensor --

output tensor with shape $(N \times O)$ where 0 is determined by `feature_size`

`training: bool`

Model Utils

`class ModelReconstructor(`*feature_extractor*, *classifier*, *connection_fn=None*`)`

reconstructs a model out of a feature_extractor and a classifier.

Parameters

- **feature_extractor** (*Module*) -- feature-extractor module
- **classifier** (*Module*) -- classifier module
- **connection_fn** (*Callable, optional*) -- optional connection function to apply on the output of feature-extractor before feeding to the classifier. Defaults to None.

`forward(`*input*`)`

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`training: bool`

`get_output_size(`*in_size*, *pad*, *kernel*, *stride*`)`

Calculates the output size after applying a kernel (for one dimension).

Parameters

- **in_size** (*int*) -- input size.
- **pad** (*int*) -- padding size. If set to `same`, input size is directly returned.
- **kernel** (*int*) -- kernel size.
- **stride** (*int*) -- size of strides.

Returns *int* -- output size

3.1.4 Utils

Small handy functions and classes used in FedSim package

Aggregators

class AppendixAggregator(*max_deque_lenght=None*)

This aggregator hold the results in a deque and performs the aggregation at the time querying the results instead. Compared to SerialAggregator provides the flexibility of aggregating within a certain number of past entries.

Parameters **max_deque_lenght** (*int, optional*) -- maximum lenght of deque to hold the aggregation entries. Defaults to None.

append(*key*, *value*, *weight=1*, *step=0*)

Appends a new weighted entry timestamped by step.

Parameters

- **key** (*Hashable*) -- key to the aggregation entry.
- **value** (*Any*) -- value of the aggregation entry.
- **weight** (*int, optional*) -- weight of the aggregation for the current entry. Defaults to 1.
- **step** (*int, optional*) -- timestamp of the current entry. Defaults to 0.

append_all(*entry_dict: Dict[str, float]*, *weight=1*, *step=0*)

To apply append on several entries given by a dictionary.

Parameters

- **entry_dict** (*Dict[Hashable, Any]*) -- dictionary of the entries.
- **weight** (*int, optional*) -- weight of the entries. Defaults to 1.
- **step** (*int, optional*) -- timestamp of the current entries. Defaults to 0.

get(*key: str*, *k: Optional[int] = None*)

fetches the weighted result

Parameters

- **key** (*str*) -- the name of the variable
- **k** (*int, optional*) -- limits the number of points to aggregate.

Returns *Any* -- the result of the aggregation

get_steps(*key*)

fetches the timestamps of the aggregation.

Parameters **key** (*Hashable*) -- aggregation key.

Raises **Exception** -- key not in the aggregator.

Returns *List[Any]* -- list of timestamps appended up to the maximum lenght of the internal deque.

get_values(*key*)

fetches the values of the aggregation.

Parameters **key** (*Hashable*) -- aggregation key.

Raises **Exception** -- key not in the aggregator.

Returns *List[Any]* -- list of values appended up to the maximum lenght of the internal deque.

`get_weights`(*key*)

fetches the weights of the aggregation.

Parameters **key** (*Hashable*) -- aggregation key.

Raises **Exception** -- key not in the aggregator.

Returns *List[Any]* -- list of weights appended up to the maximum lenght of the internal deque.

`items`()

Generator of (key, result) to get aggregation result of all keys in the aggregator.

Yields *Tuple[Hashable, Any]* -- pair of key, aggregation result.

`keys`()

fetches the keys of entries aggregated so far.

Returns *Iterable* -- all aggregation keys.

`pop`(*key*)

Similar to `get` method except that the entry is removed from the aggregator at the end.

Parameters **key** (*Hashable*) -- key to the entry.

Raises **Exception** -- key does not exist in the aggregator.

Returns *Any* -- result of the aggregation.

`pop_all`()

Collects all the aggregation results in a dictionary and removes everything from the aggregator at the end.

Returns *Dict[Hashable, Any]* -- mapping of key to aggregation result.

`class SerialAggregator`

Serially aggregats arbitrary number of weighted or unweigted variables.

`add`(*key*, *value*, *weight=None*)

adds a new item to the aggregation

Parameters

- **key** (*Hashable*) -- key of the entry
- **value** (*Any*) -- current value of the entry. Type of this value must support addition. Support for division is required if the aggregation is weighted.
- **weight** (*float, optional*) -- weight of the current entry. If not specified, aggregation becomes unweighted (equal to accumulation). Defaults to None.

`get`(*key*)

Fetches the current result of the aggregation. If the aggregation is weighted the returned value is weighted average of the entry values.

Parameters **key** (*Hashable*) -- key to the entry.

Raises **Exception** -- key does not exist in the aggregator.

Returns *Any* -- result of the aggregation.

`get_sum`(*key*)

Fetches the weighted sum (no division).

Parameters **key** (*Hashable*) -- key to the entry.

Raises **Exception** -- key does not exist in the aggregator.

Returns *Any* -- result of the weighted sum of the entries.

`get_weight`(*key*)

Fetches the sum of weights of the weighted averaging.

Parameters **key** (*Hashable*) -- key to the entry.

Raises `Exception` -- key does not exist in the aggregator.

Returns *Any* -- sum of weights of the aggregation.

`items`()

Generator of (key, result) to get aggregation result of all keys in the aggregator.

Yields *Tuple[Hashable, Any]* -- pair of key, aggregation result.

`keys`()

fetches the keys of entries aggregated so far.

Returns *Iterable* -- all aggregation keys.

`pop`(*key*)

Similar to `get` method except that the entry is removed from the aggregator at the end.

Parameters **key** (*Hashable*) -- key to the entry.

Raises `Exception` -- key does not exist in the aggregator.

Returns *Any* -- result of the aggregation.

`pop_all`()

Collects all the aggregation results in a dictionary and removes everything from the aggregator at the end.

Returns *Dict[Hashable, Any]* -- mapping of key to aggregation result.

Parameters Conversion

`initialize_module`(*module: torch.nn.modules.module.Module*, *vec: torch.Tensor*, *clone=True*, *detach=True*)

initializes a module's parameters with a 1-D vector

Parameters

- **module** (*Module*) -- module to initialize weights
- **vec** (*Tensor*) -- a 1-D Tensor
- **clone** (*bool, optional*) -- clones the vector before initilization. Defaults to True.
- **detach** (*bool, optional*) -- detaches the output before the initialization. Defaults to True.

`vector_to_named_parameters_like`(*vec: torch.Tensor*, *named_parameters_like: collections.OrderedDict*) → collections.OrderedDict

Convert one vector to new named parameters like the ones provided

Parameters

- **vec** (*Tensor*) -- a single vector represents the parameters of a model.
- **parameters** (*OrderedDict*) -- a dictioanry of Tensors that are the parameters of a model. This is only used to get the sizes and keys. New parametere are defined.

vector_to_parameters_like(*vec*, *parameters_like*)

Convert one vector to new parameters like the ones provided

Parameters

- **vec** (*Tensor*) -- a single vector represents the parameters of a model.
- **parameters** (*Iterable[Tensor]*) -- an iterator of Tensors that are the parameters of a model. This is only used to get the sizes. New parametere are defined.

vectorize_module(*module: torch.nn.modules.module.Module*, *clone=True*, *detach=True*)

convert parameters of a module to a vector

Parameters

- **module** (*Module*) -- module to convert the parameters of
- **clone** (*bool, optional*) -- clones the output. Defaults to True.
- **detach** (*bool, optional*) -- detaches the output. Defaults to True.

Returns *Module* -- 1-D Tensor of all parameters in the module

vectorize_module_grads(*module: torch.nn.modules.module.Module*, *clone=True*, *detach=True*)

convert parameters gradients of a module to a vector

Parameters

- **module** (*Module*) -- module to convert the parameters of
- **clone** (*bool, optional*) -- clones the output. Defaults to True.
- **detach** (*bool, optional*) -- detaches the output. Defaults to True.

Returns

Module --

1-D Tensor of gradients of all parameters in the module. None if at least grad of one children deos not exist.

Dict Ops

apply_on_dict(*dict_obj*, *fn*, *return_as_dict=False*, **args*, ***kwargs*)

Applies an operation defined by `fn` on all the entries in a dectionary.

Parameters

- **dict_obj** (*_type_*) -- _description_
- **fn** (*Callable*) -- method to apply on dictionary entries. The signature must be `fn(key, value, *args, **kwargs)`. where `*args` and `**kwargs` are forwarded from `apply_on_dict` method to `fn`.
- **return_as_dict** (*bool, optional*) -- If True a new dictionary with modified entries is returned.

Returns *_type_* -- _description_

Import Utils

`get_from_module`(*module_name*, *entry_name*)

Imports a module and returns it desired member if existed.

Parameters

- **module_name** (*str*) -- name of the module
- **entry_name** (*str*) -- name of the definition within the module.

Returns *Any* -- the desired definition in the given module if existed; None otherwise.

Random Utils

`set_seed`(*seed*, *use_cuda*) → None

sets default random generator seed of `numpy`, `random` and `torch`. In case of using cuda, related randomness is also taken care of.

Parameters

- **seed** (*_type_*) -- _description_
- **use_cuda** (*_type_*) -- _description_

Storage

`class Storage`

storage class to save and retrieve objects.

`change_protection`(*key*, *read_protected=False*, *write_protected=False*, *silent=False*)

changes the protection policy of an entry

Parameters

- **key** (*Hashable*) -- key to the entry
- **read_protected** (*bool, optional*) -- read protection. Defaults to False.
- **write_protected** (*bool, optional*) -- write protection. Defaults to False.
- **silent** (*bool*) -- if False and and any protection changes, a warning is printed. Defaults to False.

`get_all_keys`()

Fetches the keys of all the objects written to the storage so far including read protected ones.

Returns *Iterable[str]* -- an iterable of the keys to the

`get_keys`()

Fetches the keys of the objects written to the storage so far.

Note: to get keys of all entries included read protected ones call `get_all_keys` instead.

Returns *Iterable[str]* -- an iterable of the keys to the

`get_protection_status`(*key*)

fetches the protection status of an entry.

Parameters **key** (*Hashable*) -- key to the entry

Returns *Tuple[bool, bool]* -- read and write protection status respectively.

`read`(*key*, *silent=False*)

read from the storage.

Parameters

- **key** (*Hashable*) -- key to fetch the desired object.
- **silent** (*bool*) -- if False and entry is read protected, a warning is printed. Defaults to False.

Returns *Any* -- the desired object. If key does not exist, None is returned.

`remove`(*key*, *silent=False*)

removes an entry from the storage.

Parameters

- **key** (*Hashable*) -- key to the entry.
- **silent** (*bool, optional*) -- if False and entry is write protected a warning is printed. Defaults to False.

`write`(*key*, *obj*, *read_protected=False*, *write_protected=False*, *silent=False*)

writes to the storage.

Parameters

- **key** (*Hashable*) -- key to access the object in future retrievals
- **obj** (*Any*) -- object to store
- **read_protected** (*bool*) -- prints warning if in future key accessed by a read call. Defaults to False.
- **write_protected** (*bool*) -- print warning if in future key accessed by a write call. Defaults to False.
- **silent** (*bool*) -- if False and entry is write protected, a warning is printed. Defaults to False.

3.1.5 Fedsim Scores

`class Accuracy`(*log_freq: int = 1*, *split='test'*, *score_name='accuracy'*, *reduction: str = 'micro'*)

updatable accuracy score

`__call__`(*input*, *target*) → torch.Tensor

updates the accuracy score on a mini-batch detached from the computational graph. It also returns the current batch score without detaching from the graph.

Parameters

- **input** (*Tensor*) -- Predicted unnormalized scores (often referred to aslogits); see Shape section below for supported shapes.
- **target** (*Tensor*) -- Ground truth class indices or class probabilities; see Shape section below for supported shapes.

Shape:

- Input: Shape (N, C).
- **Target: shape** (N) **where each** value should be between $[0, C)$.

where:

$$C = \text{number of classes}$$
$$N = \text{batch size}$$

Returns *Tensor* -- accuracy score of current batch

Parameters

- **log_freq** (*int, optional*) -- how many steps gap between two evaluations. Defaults to 1.
- **split** (*str, optional*) -- data split to evaluate on . Defaults to 'test'.
- **score_name** (*str*) -- name of the score object
- **reduction** (*str*) -- Specifies the reduction to apply to the output: `'micro' | 'macro'`. `'micro'`: as if mini-batches are concatenated. `'macro'`: mean of accuracy of each mini-batch (update). Default: `'micro'`

`get_score()` → float

returns the score

Raises `NotImplementedError` -- This abstract method should be implemented by child classes

Returns *float* -- the score

`is_differentiable()` → bool

to check if the score is differentiable (to for ex. use as loss function).

Raises `NotImplementedError` -- This abstract method should be implemented by child classes

Returns *bool* -- True if the output of the call is differentiable.

`reset()` → None

resets the internal buffers, makes it ready to start collecting

Raises `NotImplementedError` -- This abstract method should be implemented by child classes

`class CrossEntropyScore(`*log_freq: int = 1*, *split='test'*, *score_name='cross_entropy_score'*, *weight=None*, *reduction: str = 'micro'*, *label_smoothing: float = 0.0*`)`

updatable cross entropy score

`__call__(`*input*, *target*`)` → torch.Tensor

updates the cross entropy score on a mini-batch detached from the computational graph. It also returns the current batch score without detaching from the graph.

Parameters

- **input** (*Tensor*) -- Predicted unnormalized scores (often referred to aslogits); see Shape section below for supported shapes.
- **target** (*Tensor*) -- Ground truth class indices or class probabilities; see Shape section below for supported shapes.

Shape:

- Input: shape (C), (N, C).

- **Target: shape** $()$**,** (N) **where each** value should be between $[0, C)$.

where:

$$C = \text{number of classes}$$
$$N = \text{batch size}$$

Returns *Tensor* -- cross entropy score of current batch

Parameters

- **log_freq** (*int, optional*) -- how many steps gap between two evaluations. Defaults to 1.
- **split** (*str, optional*) -- data split to evaluate on . Defaults to 'test'.
- **score_name** (*str*) -- name of the score object
- **reduction** (*str*) -- Specifies the reduction to apply to the output:
- **``'micro'`` | ``'macro'``. ``'micro'``** -- as if mini-batches are
- **concatenated. ``'macro'``** -- mean of cross entropy of each mini-batch
- **(update). Default** -- `'micro'`

`get_score()` → float

returns the score

Raises `NotImplementedError` -- This abstract method should be implemented by child classes

Returns *float* -- the score

`is_differentiable()` → bool

to check if the score is differentiable (to for ex. use as loss function).

Raises `NotImplementedError` -- This abstract method should be implemented by child classes

Returns *bool* -- True if the output of the call is differentiable.

`reset()` → None

resets the internal buffers, makes it ready to start collecting

Raises `NotImplementedError` -- This abstract method should be implemented by child classes

`class KLDivScore(`*log_freq: int = 1*, *split='test'*, *score_name='kl_dic_score'*, *reduction: str = 'micro'*, *log_target=False*`)`

updatable pointwise KL-divergence score

`__call__(`*input*, *target*`)` → torch.Tensor

updates the KL-divergence score on a mini-batch detached from the computational graph. It also returns the current batch score without detaching from the graph.

Parameters

- **input** (*Tensor*) -- Predicted unnormalized scores (often referred to aslogits); see Shape section below for supported shapes.
- **target** (*Tensor*) -- Ground truth class indices or class probabilities; see Shape section below for supported shapes.

Shape:

- Input: $(*)$, where $*$ means any number of dimensions.

- Target: $(*)$, same shape as the input.
- **Output: scalar by default. If `reduction` is *'none'*,** then $(*)$, same shape as the input.

Returns *Tensor* -- KL-divergence score of current batch

Parameters

- **log_freq** (*int, optional*) -- how many steps gap between two evaluations. Defaults to 1.
- **split** (*str, optional*) -- data split to evaluate on . Defaults to 'test'.
- **score_name** (*str*) -- name of the score object
- **reduction** (*str*) -- Specifies the reduction to apply to the output:
- **``'micro'`` | ``'macro'``. ``'micro'``** -- as if mini-batches are
- **concatenated. ``'macro'``** -- mean of cross entropy of each mini-batch
- **(update). Default** -- `'micro'`

`get_score`() → float

returns the score

Raises `NotImplementedError` -- This abstract method should be implemented by child classes

Returns *float* -- the score

`is_differentiable`() → bool

to check if the score is differentiable (to for ex. use as loss function).

Raises `NotImplementedError` -- This abstract method should be implemented by child classes

Returns *bool* -- True if the output of the call is differentiable.

`reset`() → None

resets the internal buffers, makes it ready to start collecting

Raises `NotImplementedError` -- This abstract method should be implemented by child classes

`class Score`(*log_freq: int = 1*, *split='test'*, *score_name=''*, *reduction='micro'*)

Score base class.

`__call__`(*input*, *target*)

updates the score based on a mini-batch of input and target

Parameters

- **input** (*Tensor*) -- Predicted unnormalized scores (often referred to aslogits); see Shape section below for supported shapes.
- **target** (*Tensor*) -- Ground truth class indices or class probabilities; see Shape section below for supported shapes.

Raises `NotImplementedError` -- This abstract method should be implemented by child classes

Parameters

- **log_freq** (*int, optional*) -- how many steps gap between two evaluations. Defaults to 1.
- **split** (*str, optional*) -- data split to evaluate on . Defaults to 'test'.
- **score_name** (*str*) -- name of the score object

- **reduction** (*str*) -- Specifies the reduction to apply to the output: `'micro'` | `'macro'`. `'micro'`: as if mini-batches are concatenated. `'macro'`: mean of score of each mini-batch (update). Default: `'micro'`

`get_name()` → str

gives the name of the score

Returns *str* -- score name

`get_score()` → float

returns the score

Raises `NotImplementedError` -- This abstract method should be implemented by child classes

Returns *float* -- the score

`is_differentiable()` → bool

to check if the score is differentiable (to for ex. use as loss function).

Raises `NotImplementedError` -- This abstract method should be implemented by child classes

Returns *bool* -- True if the output of the call is differentiable.

`reset()` → None

resets the internal buffers, makes it ready to start collecting

Raises `NotImplementedError` -- This abstract method should be implemented by child classes

3.2 FedSim cli

3.2.1 fed-learn

fedsim-cli fed-learn

Simulates a Federated Learning system.

```
fedsim-cli fed-learn [OPTIONS]
```

Options

`-r, --rounds` <rounds>

number of communication rounds.

Default 100

`-d, --data-manager` <data_manager>

name of data manager.

Default BasicDataManager

`--train-split-name` <train_split_name>

name of local split to train train on

Default train

-n, --n-clients <n_clients>
number of clients.

Default 500

--client-sample-scheme <client_sample_scheme>
client sampling scheme (uniform or sequential for now).

Default uniform

-c, --client-sample-rate <client_sample_rate>
mean portion of num clients to sample.

Default 0.01

-a, --algorithm <algorithm>
federated learning algorithm.

Default FedAvg

-m, --model <model>
model architecture.

Default SimpleMLP

-e, --epochs <epochs>
number of local epochs.

Default 5

--criterion <criterion>
loss function to use (any differentiable fedsim.scores.Score).

Default CrossEntropyScore, log_freq:50

--batch-size <batch_size>
local batch size.

Default 32

--test-batch-size <test_batch_size>
inference batch size.

Default 64

--optimizer <optimizer>
server optimizer

Default SGD, lr:1.0

--local-optimizer <local_optimizer>
local optimizer

Default SGD, lr:0.1, weight_decay:0.001

--lr-scheduler <lr_scheduler>
lr scheduler for server optimizer

Default StepLR, step_size:1, gamma:1.0

--local-lr-scheduler <local_lr_scheduler>

lr scheduler for server optimizer

Default StepLR, step_size:1, gamma:1.0

--r2r-local-lr-scheduler <r2r_local_lr_scheduler>

lr scheduler for round to round local optimization

Default StepLR, step_size:1, gamma:1

-s, --seed <seed>

seed for random generators after data is partitioned.

--device <device>

device to load model and data one

--log-dir <log_dir>

directory to store the logs.

--n-point-summary <n_point_summary>

number of last score report points to store and get the final average performance from.

Default 10

--local-score <local_score>

hooks a score object to a split of local datasets. Choose the score classes from *fedsim.scores*. It is possible to call this option multiple times.

--global-score <global_score>

hooks a score object to a split of global datasets. Choose the score classes from *fedsim.scores*. It is possible to call this option multiple times.

Options

Arguments

3.2.2 fed-tune

fedsim-cli fed-tune

Tunes a Federated Learning system.

```
fedsim-cli fed-tune [OPTIONS]
```

Options

--n-iters <n_iters>

number of iterations to ask and tell the skopt optimizer

Default 10

--skopt-n-initial-points <skopt_n_initial_points>

number of initial points for skopt optimizer

Default 10

--skopt-random-state <skopt_random_state>

random state for skopt optimizer

Default 10

--skopt-base-estimator <skopt_base_estimator>

skopt estimator

Default GP

Options GP | RF | ET | GBRT

--eval-metric <eval_metric>

complete name of the metric (returned from train method of algorithm) to minimize (or maximize if --maximize is passed)

Default server.avg.test.cross_entropy_score

--maximize, **--minimize**

complete name of the metric (returned from train method of algorithm) to minimize or maximize

-r, **--rounds** <rounds>

number of communication rounds.

Default 100

-d, **--data-manager** <data_manager>

name of data manager.

Default BasicDataManager

--train-split-name <train_split_name>

name of local split to train train on

Default train

-n, **--n-clients** <n_clients>

number of clients.

Default 500

--client-sample-scheme <client_sample_scheme>

client sampling scheme (uniform or sequential for now).

Default uniform

-c, **--client-sample-rate** <client_sample_rate>

mean portion of num clients to sample.

Default 0.01

-a, --algorithm <algorithm>
federated learning algorithm.

Default FedAvg

-m, --model <model>
model architecture.

Default SimpleMLP

-e, --epochs <epochs>
number of local epochs.

Default 5

--criterion <criterion>
loss function to use (defined under fedsim.losses).

Default CrossEntropyScore, log_freq:50

--batch-size <batch_size>
local batch size.

Default 32

--test-batch-size <test_batch_size>
inference batch size.

Default 64

--optimizer <optimizer>
server optimizer

Default SGD, lr:1.0

--local-optimizer <local_optimizer>
local optimizer

Default SGD, lr:0.1, weight_decay:0.001

--lr-scheduler <lr_scheduler>
lr scheduler for server optimizer

Default StepLR, step_size:1, gamma:1.0

--local-lr-scheduler <local_lr_scheduler>
lr scheduler for server optimizer

Default StepLR, step_size:1, gamma:1.0

--r2r-local-lr-scheduler <r2r_local_lr_scheduler>
lr scheduler for round to round local optimization

Default StepLR, step_size:1, gamma:0.999

-s, --seed <seed>
seed for random generators after data is partitioned.

--device <device>
device to load model and data one

--log-dir <log_dir>

directory to store the logs.

--n-point-summary <n_point_summary>

number of last score report points to store and get the final average performance from.

Default 10

--local-score <local_score>

hooks a score object to a split of local datasets. Choose the score classes from *fedsim.scores*. It is possible to call this option multiple times.

--global-score <global_score>

hooks a score object to a split of global datasets. Choose the score classes from *fedsim.scores*. It is possible to call this option multiple times.

Options

Arguments

CHAPTER

FOUR

CONTRIBUTOR GUIDE

4.1 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

4.1.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

4.1.2 Documentation improvements

fedsim could always use more documentation, whether as part of the official fedsim docs, in docstrings, or even on the web in blog posts, articles, and such.

4.1.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/varnio/fedsim/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

4.1.4 Development

To set up *fedsim* for local development:

1. Fork fedsim (look for the "Fork" button).
2. Clone your fork locally:

```
git clone git@github.com:YOURGITHUBNAME/fedsim.git
```

3. Create a branch for local development:

```
git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

4. When you're done making changes run all the checks and docs builder with tox one command:

```
tox
```

5. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`).
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

Tips

To run a subset of tests:

```
tox -e envname -- pytest -k test_myfeature
```

To run all the test environments in *parallel*:

```
tox -p auto
```

4.1.5 Authors

- Farshid Varno - https://fvarno.github.io/
- William Taylor-Melanson - https://github.com/wtaylor17

FedSim is a comprehensive and flexible Federated Learning Simulator. It aims to provide the researchers with an easy to develope/maintain simulator for Federated Learning.

Getting Started

Install FedSim and simulate your first Federated Learning Simulation in 2 lines.

Download & Train

User Guide

The user guide provides in-depth information on what you can do with FedSim.

To the user guide

API Reference

The reference guide contains a detailed description of the functions, modules, and objects included in FedSim. The reference describes how the methods work and which parameters can be used.

To the reference guide

Contributor's Guide

FedSim is an open source project. We are open to community contributions and are thankful of all the efforts made. Check here for how to develepe/contribute.

To the contributor's guide

PYTHON MODULE INDEX

f

r

INDEX

Symbols

A

G

H

I

K

L

M

O

P

R

S

T

V

W