Personal Project: Federated Learning

UNIkeEN

May 9, 2024

Contents

1 Introduction

Federated learning was first proposed by McMahan et al. in 2017 [1] as a distributed machine learning approach that utilizes privacy-sensitive data from mobile devices. Instead of aggregating data at a central server, this method involves sending the model to mobile devices which use the data to train a shared model. The cloud server collects updates generated from local training on each mobile device and aggregates them to produce a new global model. By decentralizing computation tasks to the original locations of data, this approach effectively reduces the bandwidth needed for data transmission and minimizes the risk of privacy breaches.

Federated learning can be further categorized into horizontal and vertical federated learning based on data distribution characteristics. In horizontal federated learning, participants possess data that largely overlap in feature space but have small intersections in sample space, making it suitable for collaboration among different sources within the same domain. Vertical federated learning, on the other hand, is applicable where data overlap in sample space but not completely in feature space, typically across different industries or domains. This setup allows institutions to collaboratively build more comprehensive data models while maintaining data privacy.

In this project, we are tasked with implementing a system that simulates training a classification model on the BloodMNIST [2] and CIFAR10 [3] dataset using horizontal federated learning, with the dataset pre-partitioned into a fixed number of clients. Our designed system supports three stages of the project requirements:

- Stage 1: Implement one server and N clients, with all N clients participating in each round of updates, facilitated through the reading and writing of `pth` files.
- Stage 2: Based on stage 1, only a random selection of M clients ($M \leq N$) participates in the updates each round.
- Stage 3: Transit from using `pth` files to using **socket** communication for interactions between clients and the server.

Our system is modular, supporting modifications through YAML configuration files for communication methods, training modes (whether all clients participate in each update), data paths, model selection, and training hyperparameters.

2 Methodology

2.1 Framework Overview

The three stages specified in the project requirements share similar processes and principles. Consequently, we endeavor to implement a universal horizontal federated learning framework aimed at providing: (1) a modular pipeline; (2) flexibility to switch between different datasets and models; and (3) ease of further development, whereby changing the training approach can be achieved by

modifying specified methods. This modification could entail simulating through file read/write operations (stage 1) or utilizing sockets (stage 3) for client-server communication, and deciding whether each training round requires the participation of all clients (stage 2).

Similar to many engineering projects employing deep learning technologies, our framework comprises three critical components: dataset loader, models, and the pipeline.

Dataset Loader The dataset has been pre-divided into multiple training subsets and one testing subset. In practical federated learning scenarios, to ensure privacy, each client should only have access to the portion of the training dataset it possesses, while the server should only have access to the testing subset. In stages 1 and 2 of this project, we employed a single process to load all datasets into memory simultaneously, ensuring that subsequent simulations of training with different clients treat each subset of data as independent.

Models For the BloodMNIST and CIFAR10 datasets used in this project, we employed both a multi-layer perceptron and the classic convolutional network LeNet, which is developed by Yann LeCun et al. in the 1990s [4]. LeNet consists of several layers alternately performing convolution and pooling operations, followed by fully connected layers. It was originally designed for handwritten digit recognition tasks and played a significant role in the early development of deep learning and computer vision.

2.2 Pipeline Design

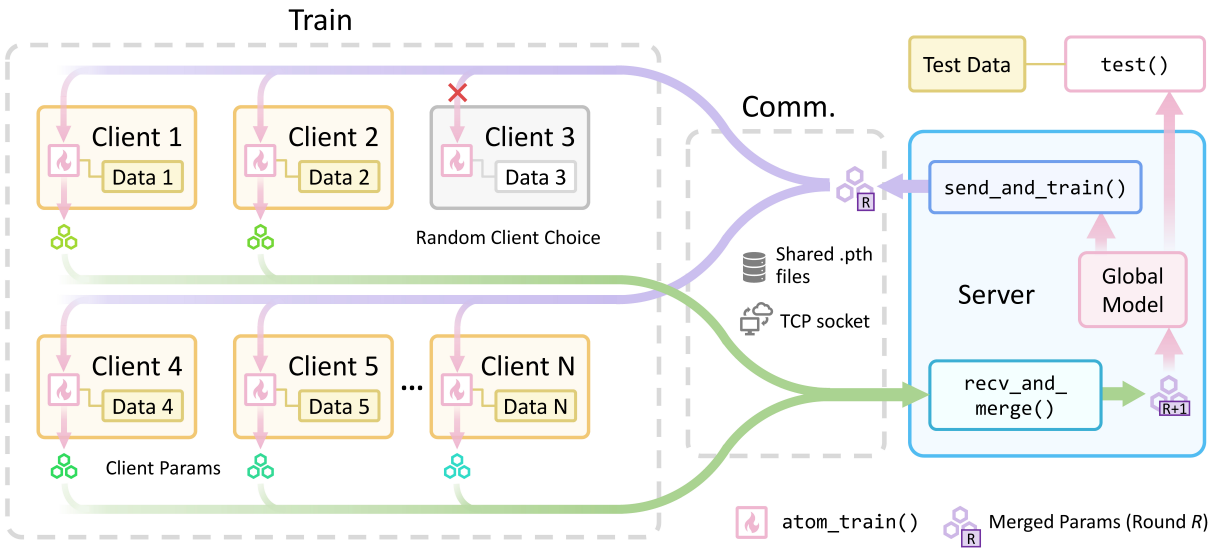


Fig. 1. Pipeline of Federated Learning

The fundamental pipeline of horizontal federated learning is illustrated in Fig. 1. Initially, N clients and the server are initialized based on the training configuration (hyperparameters, model architecture, etc.). In each round (totally `n_rounds`) of global iteration, the parameters of the

global model are first transmitted to randomly selected M clients ($M \leq N$). Subsequently, each selected client conducts several local epochs (i.e. `n_epochs`) of model training using their respective private dataset. Following this, the clients upload their local models to the server, where the server aggregates the model parameters to obtain the global model. The global model is then evaluated on the test dataset. If the test accuracy exceeds the previous highest accuracy, the model parameters are updated and saved; otherwise, discarded.

In our framework, this pipeline is concretely implemented as the `Pipeline` class. To accommodate different communication and training modalities required by various stages, one only needs to inherit from the `Pipeline` class and implement the two functions, `send_and_train()` and `recv_and_merge()`, as depicted in Fig. 1. Additionally, the pre-implemented `atom_train()` function is available for conducting independent training on the client side.

Here are the codes of `Pipeline` and main training process:

```python
class Pipeline(ABC):
    def __init__(self, cfg): ... # initialize

    @abstractmethod
    def send_and_train(self, idx, global_state):
        # send global model's state to client_{id} and start training at client.
        raise NotImplementedError

    @abstractmethod
    def recv_and_merge(self, idx):
        # receive client models' state and do aggregation
        raise NotImplementedError

    def train(self):
        ...
        for r in range(self.n_rounds):
            ...
            global_state = torch.load(global_ckp_path) # send global_model and
                train at client
            self.send_and_train(idx, global_state)
            avg_model = self.recv_and_merge(idx) # calc avg of client parameters
            ... # test model of this round, save if it's better
```

Offline Pipeline The offline pipeline corresponds to stage 1 and 2 of the project requirements, utilizing a single-process simulation. When it is the turn of client i, the process begins by loading the current global model parameters from `global.pth`. After training, client i saves its updated

parameters to `client_i.pth`. Following the sequence dictated by a previously randomized list of client identifiers, each selected client **sequentially** completes its training. Subsequently, the server opens the `.pth` files saved by the clients, sums the parameters, and then computes their average (that is "FedAVG" method).

Online Pipeline The online pipeline, corresponding to stage 3 of the project requirements, initiates by launching N client processes, each automatically assigned a free port for establishing a socket connection with the server. Clients operate within a loop while maintaining continuous connectivity. At the start of each cycle, clients receive parameters from the server, perform independent training, and then send the updated local parameters back to the server's port. This cycle of receiving and sending parameters continues until an `FIN` signal is received, indicating the completion of the entire training process. Upon receipt of this signal, clients disconnect and exit the loop.

The training process in the online pipeline is **parallelized**. On the server side, both sending and receiving processes are managed through multithreading, with the use of locks during parameter reception and aggregation to ensure data integrity.

Here are the key codes of server process:

```
class OnlinePipeline(Pipeline):
    ...
    def recv_and_merge(self, idx):
        ...
        for i in idx: # using muti-threads
            thread = threading.Thread(target=self._recv_model,
                args=(self.client_sockets[i], i, idx))
            threads.append(thread)
            thread.start()
        for thread in threads:
            thread.join()
        for avg_param in self.avg_model.parameters():
            avg_param.data /= self.M
        return self.avg_model

    def _recv_model(self, client_socket, i, idx):
        try:
            ... # receive and save client params
            with self.lock: # using lock
                for avg_param, client_param in ...:
                    avg_param.data += client_param.data
        except socket.error as e: ...
```

3 Implementation Details

This section delves into more details regarding the code implementation.

3.1 Loading Configuration

Firstly, given that deep learning involves numerous hyperparameters and file paths as command-line arguments, we drew inspiration from exemplary projects like HumanGaussian [5] and opted to store training configurations in YAML files, which are accessed via the OmegaConf library.

The configurations include the training mode (online or offline), the number of clients selected per round, the choice of model and the number of input/output channels, the number of iterations per round globally and on the client side, learning rate, batch size, and the operating device (whether to use GPU). Consequently, users can adapt the framework to different datasets, tasks, and training configurations by drafting distinct YAML files. At runtime, the command-line argument needs only to specify the path to the YAML file; additionally, the project also supports inputting other configuration parameters directly from the terminal command line.

Additionally, prior to commencing training, we check whether the paths specified for client/server weight storage in the configuration file exist and are empty. If they exist and are not empty, we prompt the user whether to clear them. If they do not exist, we automatically create them.

3.2 Data Transmission Using Sockets

In our project, the server and clients maintain a continuous connection until the completion of global training. We designed a dedicated function, `receive_data()`, which continuously receives data in an infinite loop via `sock.recv(buffer_size)`. If data packets are received, they are appended together. Specifically, if the end of the data received contains the `END` signal, this indicates the end of the transmission. The function then removes the `END` from the data packet, exits the loop, and returns the data. If the data ends with the `FIN` signal, the function immediately returns `FIN`.

For parameter transmission between the client and the server, `io.BytesIO()` is used to create a buffer from which data is transferred. Once transmission is completed, both sides perform `sendall(b"END")` to indicate the end of that particular transmission. At the conclusion of global training, `sendall(b"FIN")` is sent to all clients to signify the termination of the session. This structured approach to data transmission ensures clarity and efficiency in the communication between the server and clients throughout the training process.

3.3 Learning Rate Scheduling

We can implement dynamic learning rate scheduling akin to conventional deep learning practices. The most straightforward approach involves reducing the learning rate for all clients' optimizer

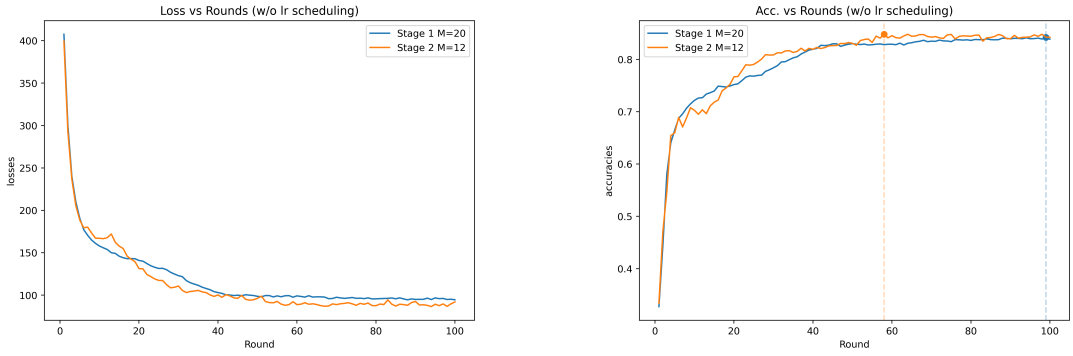
instances if there is no improvement in the global parameters over several rounds. A higher learning rate at the beginning of training can accelerate model convergence and prevent settling into suboptimal local minima. Conversely, a lower learning rate in the later stages of training helps in finding more precise solutions and avoids oscillations.

4 Experiments

Our framework is built upon PyTorch. This section of the experiment utilizes the BloodMNIST dataset with a batch size set to 16. The loss function employed is the cross-entropy function (`nn.CrossEntropyLoss`), and the optimizer used is `optim.SGD`. The experimental platform comprises an `Intel(R) Core(TM) i5-13600KF` and a single `NVIDIA GeForce RTX 4070` GPU, operating under `Windows 11 version 23H2`.

4.1 Offline Pipeline

The experiments employed the LeNet model with a fixed learning rate of 0.005. We set `n_rounds` to 100 for a total of 100 global iterations, and `n_epochs` to 10, allowing each client to perform 10 local iterations per round. The number of clients, N, was set to 20. The offline pipeline was executed under two conditions: $M = 20$ as stage 1 and $M = 12$ as stage 2. The results on server-side test dataset are as follows:



(a) Test Loss (b) Test Accurancy

Fig. 2. Result of Offline Pipeline Experiments

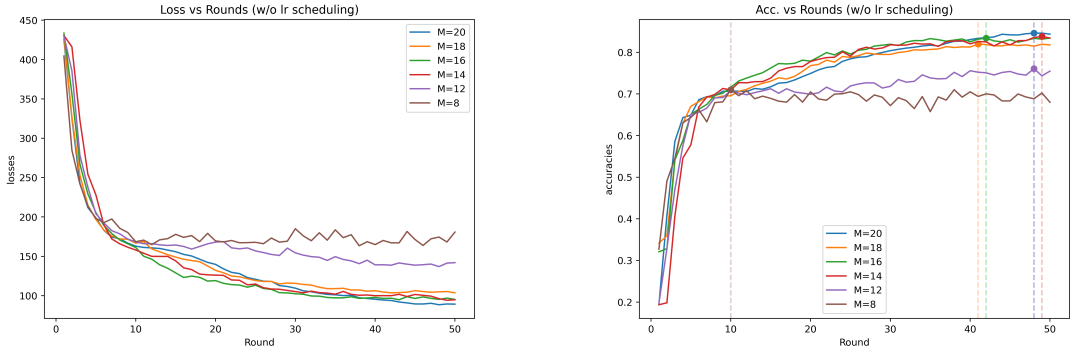
It is important to note that despite the fluctuations in accuracy observed in Fig. 2(b), our strategy involves saving the model only when there is an increase in test accuracy. Moreover, the model parameters sent to the clients in each round are those that previously achieved optimal performance. This approach ensures a stable improvement in model effectiveness and, when $M < N$, reduces the impact of specific data distributions, thereby enhancing generalizability.

From Fig. 2, it is evident that the offline pipeline can converge normally. When $M = 20$ and $M = 12$, the model reached its optimum at round 99 and 58 respectively, with test accuracies

of **84.22%** and **84.80%**. The oscillations in the later results could potentially be attributed to overfitting and an excessively high learning rate.

4.2 Online Pipeline

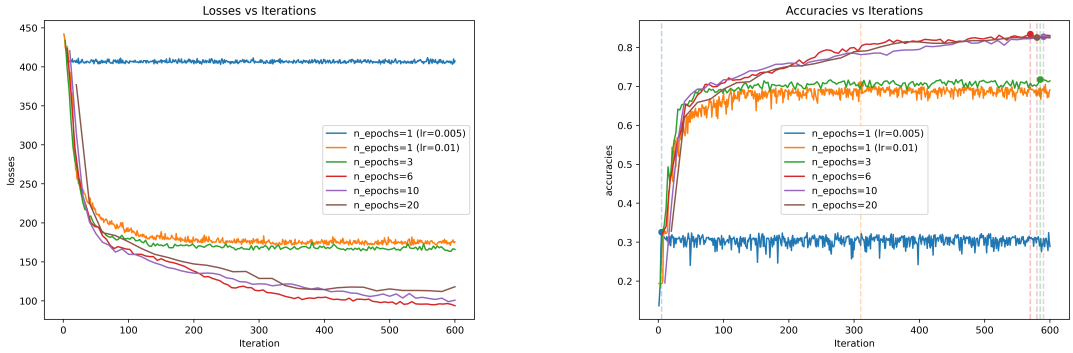
The experiments employed the LeNet model with a fixed learning rate of 0.01. We set `n_rounds` to 50 and `n_epochs` to 5. The server is set to port 23333, with N being 20 and M varying across the series [8, 12, 14, 16, 18, 20] during the execution of the online pipeline. The results are shown in Fig. 3.



(a) Test Loss (b) Test Accurancy

Fig. 3. Result of Online Pipeline Experiments (Using Different M)

As observed in Fig. 3, under the stated parameter conditions, convergence is faster when M is slightly less than N (specifically for $M = 14, 16, 18$) than when $M = N$. When M is too small, convergence slows significantly or even struggles to find an optimal direction, as illustrated by the curve for $M = 8$. The highest final model accuracy is achieved when $M = N$ (reaching **84.60%** in round 48), likely because each client participates in the learning process, thereby enhancing the model's generalizability.



(a) Test Loss (b) Test Accurancy

Fig. 4. Result of Online Pipeline Experiments (Using Different `n_epochs` and `n_rounds`)

Furthermore, we explored the relationship between aggregation strategies and model performance. The learning rate was set at 0.005, with N and M fixed at 20 and 16, respectively. We varied `n_epochs`, which represents the number of local iterations on the client side before returning to the server for aggregation in each round, testing values of 1, 3, 6, 10, and 20. The results are shown in Fig. 4, with the total number of epochs represented on the horizontal axis.

Observations from Fig. 4 indicate that with a lower learning rate, fewer local iterations per round lead to faster optimization in the initial stages of training. However, when `n_epochs` is set to low values such as 1 or 3, the optimization is less effective and accuracy oscillates around lower values. This may be due to the local models not having sufficient iterations to find the correct optimization direction before being frequently merged, resulting in the global model also failing to optimize correctly. Increasing the learning rate could mitigate this issue.

5 Conclusion

In this project, we developed a modular, generic framework to simulate horizontal federated learning, ensuring data privacy through file read/write operations or socket transmission of parameters, and parallel processing to accelerate training. Our experiments demonstrated that the framework can successfully train classification models on the BloodMNIST dataset, achieving a peak validation accuracy of 85% when utilizing the LeNet model. Further experiments indicate that different federated training strategies (frequency of merging) must be selected based on varying learning rates and hyperparameters to avoid issues such as slow convergence and failure to find the correct optimization direction. Future work can involve more extensive experiments to explore the relationships between private data distribution differences, training strategies, and hyperparameters. Our code is available at `https://github.com/UNIkeEN/CS3511-Federated-Learning`.

References

[1] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Artificial intelligence and statistics*, pp. 1273–1282, PMLR, 2017.

[2] J. Yang, R. Shi, and B. Ni, "Medmnist classification decathlon: A lightweight automl benchmark for medical image analysis," in *IEEE 18th International Symposium on Biomedical Imaging (ISBI)*, pp. 191–195, 2021.

[3] A. Krizhevsky, "Learning multiple layers of features from tiny images," tech. rep., 2009.

[4] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[5] X. Liu, X. Zhan, J. Tang, Y. Shan, G. Zeng, D. Lin, X. Liu, and Z. Liu, "Humangaussian: Text-driven 3d human generation with gaussian splatting," *arXiv preprint arXiv:2311.17061*, 2023.