Federated Learning Solo Project Report

Jiahao Yao

2024 年 5 月 12 日

1 Introduction

1.1 Project Objectives

- Learn the basic principles and processes of federated learning.
- Simulate and implement the interaction between clients and cloud servers in federated learning.

1.2 Task Description

The project aims to simulate a simple BloodMNIST/CIFAR10 (choose one) classification task under the horizontal federated learning mode. The experiment is divided into three stages:

Stage One:

- The server and multiple clients collaborate to train the classification task and update model parameters.
- All clients participate in each round of updates, completing the sequence of sending the global model, training local models, uploading local models, and aggregating global models.

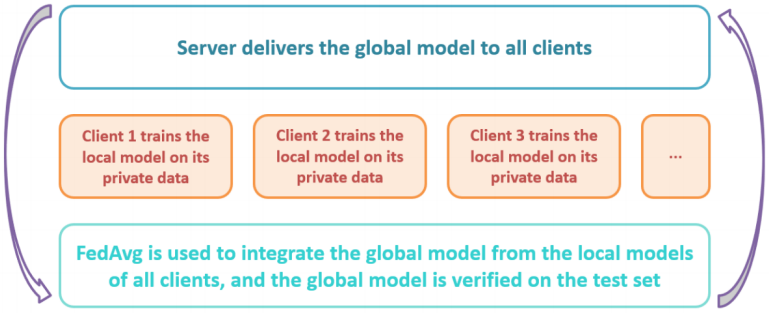

图 1: FL Illustration

Stage Two:

- Based on Stage One, only select a subset of clients to participate in each round of global updates, known as the partial participation mode.
- The program should support randomly selecting different numbers of clients to participate in updates.

Stage Three:

- Interaction between the server and clients is conducted via Socket communication, replacing the use of ".pth" file read and write in Stage One.
- The server listens on a port, receives local models from clients, gradually completes model aggregation, then sends the global model to each client.

- Clients conduct local training on their private datasets, send local model parameters to the server, and receive global model parameters.

2 Project Implementation

2.1 Overall Framework

1) **Dataset Handling (dataset.py):** The Dataset class loads training and testing datasets and provides methods to obtain training data loaders and testing data loaders.

2) **Model Definition (model.py):** Defines the LeNet model for image classification tasks.

3) **Client (client.py):** Connects to the server and receives global model parameters. Trains the model using private data and sends back updated model parameters to the server.

4) **Server (fl_system.py):** Defines the abstract class Pipeline, which contains the basic logic of the training process. Derived classes Offline_Pattern and Online_Pattern implement specific training processes. Offline_-Pattern trains the model on each client, then aggregates the updated parameters. Online_Pattern establishes Socket connections between the server and clients, sending and receiving model parameters in real-time.

5) **Configuration File (param.yaml):** Contains parameters required during the training process, such as devices, ports, data paths, etc.

The overall project flow is as follows:

The server loads the configuration file and selects the corresponding training process according to the mode. In online mode, the server establishes a Socket connection with clients, sends global model parameters to clients, clients receive and train using their private data, then send back the updated parameters to the server. In offline mode, the server sends global model parameters to each client, clients train the model using their private data, then send back the updated parameters to the server, and the server aggregates the updated parameters.

2.2 Introduction to Core Component FL_sys

The abstract class `FL_sys` is one of the core components of the FL training framework. It defines the basic framework of the training process and specifies the methods that derived classes need to implement to support different training modes. It has three key methods:

- `send_and_train(idx, global_state)`

  idx represents the list of client indices participating in the training. In global updates, only a subset of clients may participate in the update. Therefore, the idx list specifies the participating clients. global_state: represents the current state of global model parameters. In federated learning, the global model is periodically sent to each client, and clients train locally based on these global model parameters. The implementation of this method will send global model parameters to the specified clients and train locally on the client side using these parameters.

- `aggregate(idx)`

  This method is used to receive local model parameters from clients and merge them into new global model parameters. Specifically, it accepts one parameter: idx: representing the list of client indices that need to send model parameters. In global updates, only a subset of clients participate, so the idx list specifies the clients that need to send model parameters. The implementation of this method will receive local model parameters from the specified clients, merge these parameters according to a certain strategy (such as averaging), and generate new global model parameters.

- `train()`

  This method controls the entire federated learning training process. In this method, first determine the client indices idx participating in the update, then call the `send_and_train()` method to send global model parameters and train on the client side, then call the `recv_and_merge()` method to receive local model parameters from clients and merge them into new global model parameters, and finally perform model testing and save the best global model parameters.

2.3 Partial Client Participation in Model Updates

In the `Offline_Pattern` class, my `send_and_train()` method First, select a subset of clients to participate in global model updates based on the idx parameter. Then, for each selected client, send the parameters of the global model to the client, allowing the client to train locally. During local training on the client side, the parameters of the global model are used. After training, the client saves the locally trained model parameters in a local file.

Regarding random selection, I used the `numpy.random.choice` method to randomly select a certain number of client indices from all clients. The parameters of the `numpy.random.choice` method include:

- `self.N`: total number of clients, i.e., all clients participating in global model updates.
- `self.M`: number of clients to be selected, i.e., size of the subset of clients participating in global model updates.
- `replace=False`: indicates that the same client index cannot be selected repeatedly.

In param.yaml, you can adjust the number of clients participating in random updates each time by setting the mode parameter to offline and changing the n_update_clients parameter.

2.4 Socket Communication Implementation

In the above code, the client (`client.py`) uses Python's Socket module to communicate with the server. The communication process is roughly as follows:

1. **Establishing Connection**:
   - In the `connect_to_server` function, the client creates a socket object using the `socket.socket` method and connects to the specified server address and port using the `connect` method.
2. **Receiving and Updating Model**:
   - The `receive_and_update_model` function is a loop where the client continuously receives global model parameters from the server.
   - When the received data is `b"FIN"`, it indicates the end of global model updates, and the client closes the connection.
   - When the received data is empty, it indicates that the connection may have been closed, and the client also closes the connection.
   - If valid data is received, it is loaded as global model parameters, and the `single_train` function is used to train the client model locally.
   - The client sends back the trained model parameters to the server via the socket and sends `b"END"` to indicate the completion of data transmission.
3. **Client Main Process**:
   - The `client_process` function is the main execution logic of the client. It loads client-specific data, establishes a connection with the server, and starts receiving and updating model parameters.
   - This function accepts two parameters in the command line: client ID and parameter file path, then loads the configuration information from the parameter file, and starts the client process.

2.5 Model Selection and Training

I chose the LeNet model as the training model for this project. The LeNet model was proposed by Yann LeCun et al. in 1998 and is one of the classic models in the field of deep learning, originally used for handwritten digit recognition. The model has the following structure:

- **Convolutional Layers:** The first convolutional layer: input of 3 channels (color image), output channels of 6, convolution kernel size of 5x5. The second convolutional layer: input channels of 6, output channels of 16, convolution kernel size of 5x5.
- **Pooling Layers:** A max-pooling layer follows each convolutional layer, with a pooling kernel size of 2x2.
- **Fully Connected Layers:** The first fully connected layer: input size of 256, output size of 120. The second fully connected layer: input size of 120, output size of 84. The third fully connected layer: input size of 84, output size of output_channel, which typically corresponds to the number of output categories of the model.
- **Activation Function:** ReLU activation function is used after each convolutional layer and fully connected layer.

Figure 2 shows the structure of LeNet:

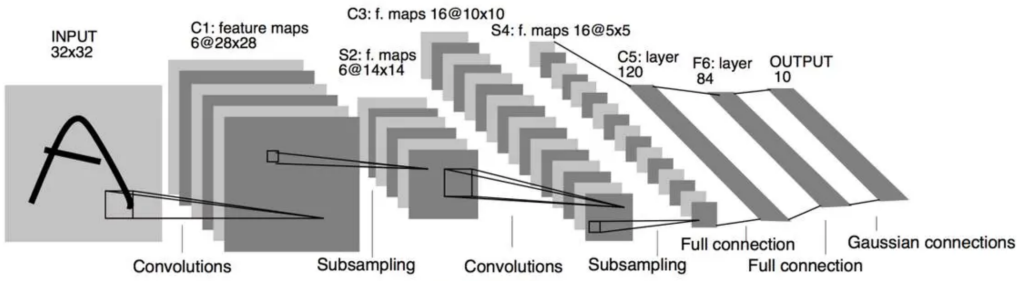

图 2: LeNet Structure

3 Experimental Results

3.1 Batch Update Experiment

Parameter settings: N = 20, M = 10, batch_size: 32, n_rounds: 20, n_epochs: 20, lr: 0.01. Training data is from BloodMNIST.

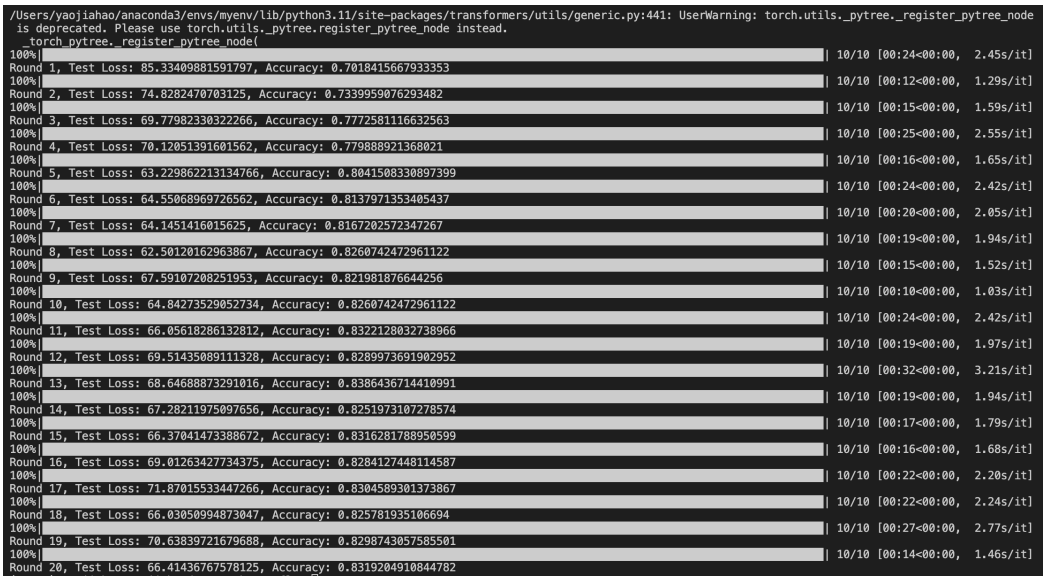

图 3: Batch Update Experiment Results

The final accuracy reached **0.8319** in the 20th round.

3.2 Real-time Update Experiment

Parameter settings: N = 20, M = 15, batch_size: 32, n_rounds: 20, n_epochs: 50, lr: 0.01. Training data is from BloodMNIST.

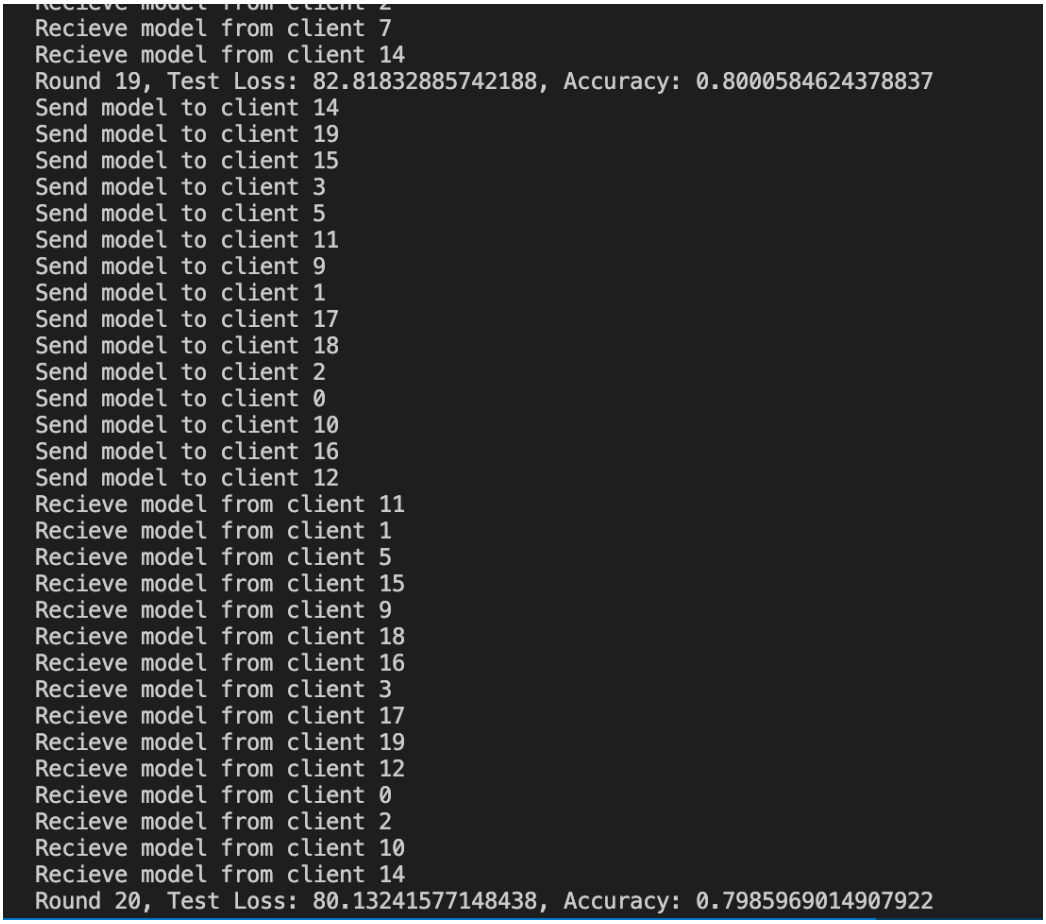

图 4: Real-time Update Experiment Results

The final accuracy reached **0.7986** in the 20th round.

4 Conclusion

This project aims to learn and implement the basic principles and processes of federated learning by simulating the interaction between clients and cloud servers in federated learning. The project task includes simulating a simple BloodMNIST classification task under the horizontal federated learning mode.

The implementation framework of the entire project includes dataset processing, model definition, clients, servers, and configuration files. Among them, the abstract class FL_sys is one of the core components of the project training framework, defining the basic framework of the training process and controlling the update and training of the global model through the send_and_train, aggregate, and train methods. In the Offline_Pattern, partial client participation in model updates is achieved by randomly selecting a subset of client indices using the numpy.random.choice method from all clients. In terms of Socket communication, clients communicate with servers using Python's Socket module, establishing connections and continuously receiving and updating model parameters.

In terms of experimental results, batch update experiments and real-time update experiments were conducted under different parameter settings, obtaining model accuracy results. The final experimental results show that under the given parameter settings, the model performs well in federated learning tasks.