FEDERATED LEARNING PROJECT REPORT

Jiayi Zhang
Department of Computer Science and Engineering
Shanghai Jiao Tong University
Shanghai
`{jiayizhang}@sjtu.edu.cn`

ABSTRACT

This is a project report for the solo project of CS3511. In this project, our task is to implement a simulation system with horizontal federated learning method for MINIST classification. Our experiment has been devided into three stages. At Stage 1, every client in the system has to participate the aggregation and update process in each round. At Stage 2, we only need M out of N clients participate the aggregation and update in each round. At Stage 1 and 2, client processes communicate with server process using local file IO. At stage 3, client processes communicate with server process using sockets.

***K*eywords** Horizontal Federated Learning

1 Introduction

Federated Learning first proposed by McMahan et al. at 2016 in [1, Communication-efficient learning of deep networks from decentralized data]. In order to utilize privacy sensitive data on modern mobile devices, they designed a decentralized approach named Federated Learning to train a shared model with those sensitive data by sending the model to mobile devices instead of collecting data to the cloud server. In this approach, the cloud server collects updates generated by each mobile devices' local training and aggregates to produce a new global model. With this method, the global model can be trained with those valuable but sensitive data without risks while collecting data from local to cloud.

In this project, we are required to implement a system simulating that we are traing a classification model with federated learning method on MINIST dataset and the dataset has been devided into 20 different clients. Referring to [2, Towards Federated Learning at Scale: System Design], we designed three systems corresponding to the three stages requirement for our project:

- Stage 1: Implement one server process and N (in our experiment $N = 20$) client processes, and all the N clients need to participate in each round of update.
- Stage 2: Implement one server process and N (in our experiment $N = 20$) client processes, only selected M clients have to participate in a round of update.
- Stage3: Implement one server process and N (in our experiment $N = 20$) client processes with socket communication method.

2 System Design

2.1 Stage 1: N-Client FL System

We first design a N-Client FL System, the work flow is as the chart below. As Fig.1, server process and client processes share a folder to save their local models. At the initiation stage, server process initiates the global model and saves it for all the N clients. Client processes load the initial global model as the local model and start to train their local

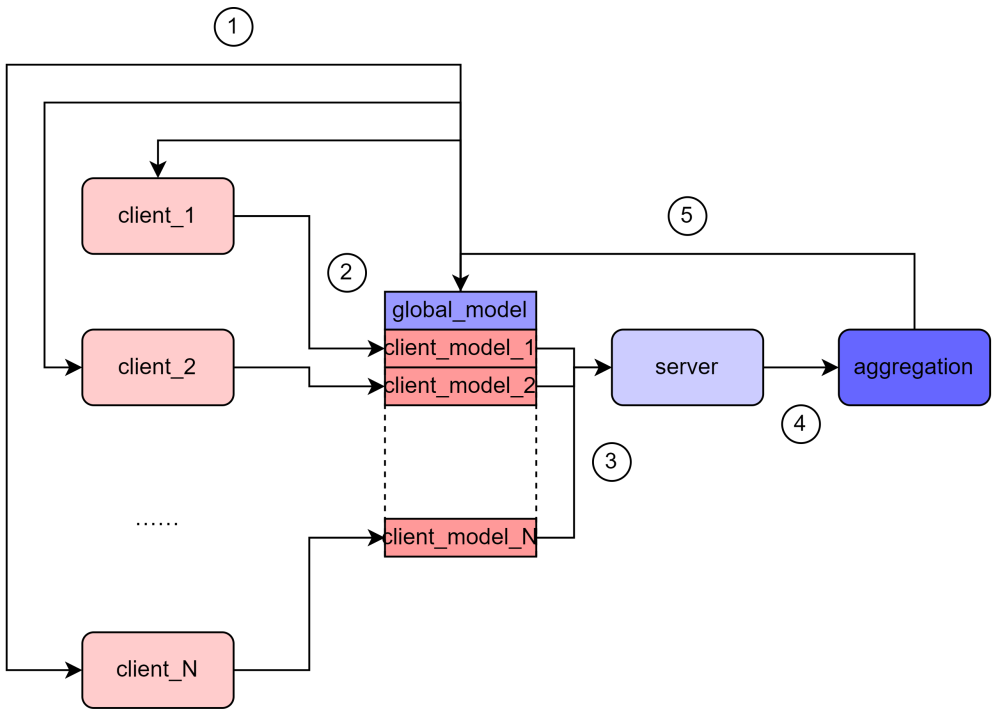

Figure 1: N-Client FL System

model with their own training dataset. At the end of each local training round, client processes save their local models at the shared folder and when all the N local models are ready the server process loads all clients' local models and aggregates parameters with FedAVG algorithm. When server process finish aggregation and save the updated global model, client processes load the newly updated global model as their new local model and start the new round of training. In conclusion, there are 5 steps for a round of training and update:

1. Server process saves the `global_model`, client processes load the `global_model` as their local model.
2. Client processes finish their local training and save their `client_model`.
3. Server process loads all the N `client_model`.
4. Server process aggregates parameters of all the N `client_model`.
5. Server process saves the updated `global_model`, back to step 1.

2.2 Stage 2: M-out-of-N FL System

At Stage 2 we design a M-out-of-N FL System, the work flow is as the chart below: As Fig.2, like the design for Stage 1, server process and client processes share a folder to save their local models. The only difference between Stage 1 and Stage 2 is that the server process at Stage 2 will only randomly choose M clients' local models to aggregate and generate the updated global model each round. In conclusion, there are 5 steps for a round of training and update:

1. Server process saves the `global_model`, client processes load the `global_model` as their local model.
2. Client processes finish their local training and save their `client_model`.
3. Server process randomly chooses and loads M `client_model` from N `client_model`.
4. Server process aggregates parameters of the chosen M `client_model`.
5. Server process saves the updated `global_model`, back to step 1.

2.3 Stage 3: N-Client FL System with Socket

At Stage 3 we design another N-Client FL System, the difference between Stage 1 and Stage 3 is that we use socket communication instead of using shared folder to exchange models, the work flow is as the chart below: As Fig.3, server

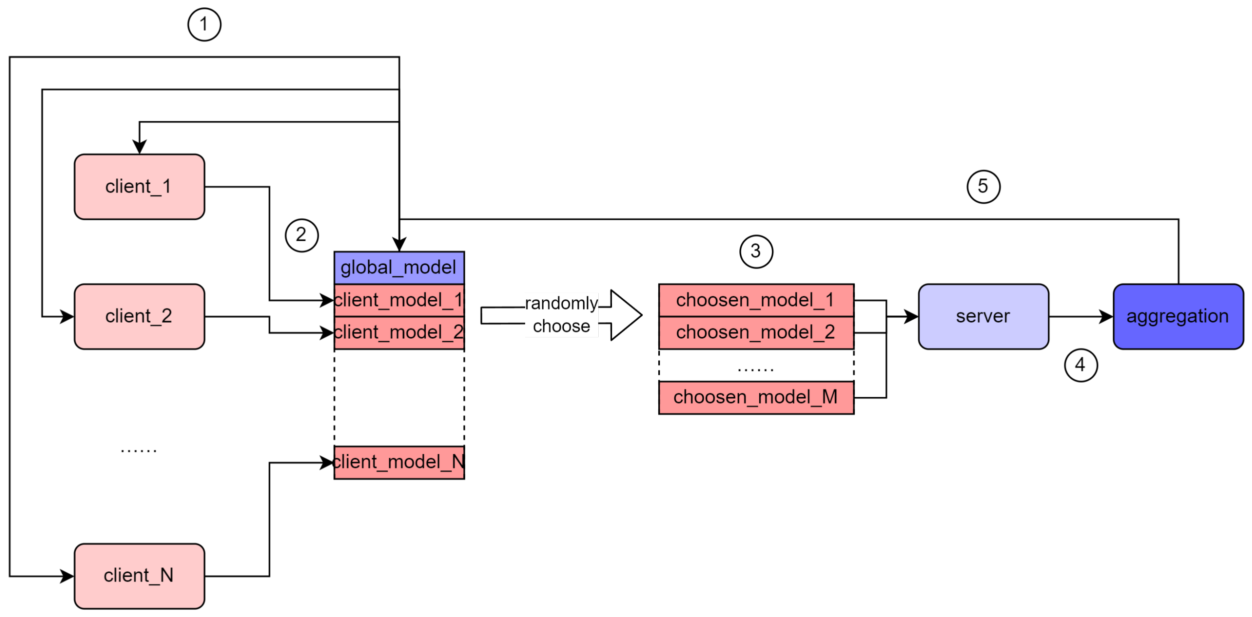

Figure 2: M-out-of-N FL System

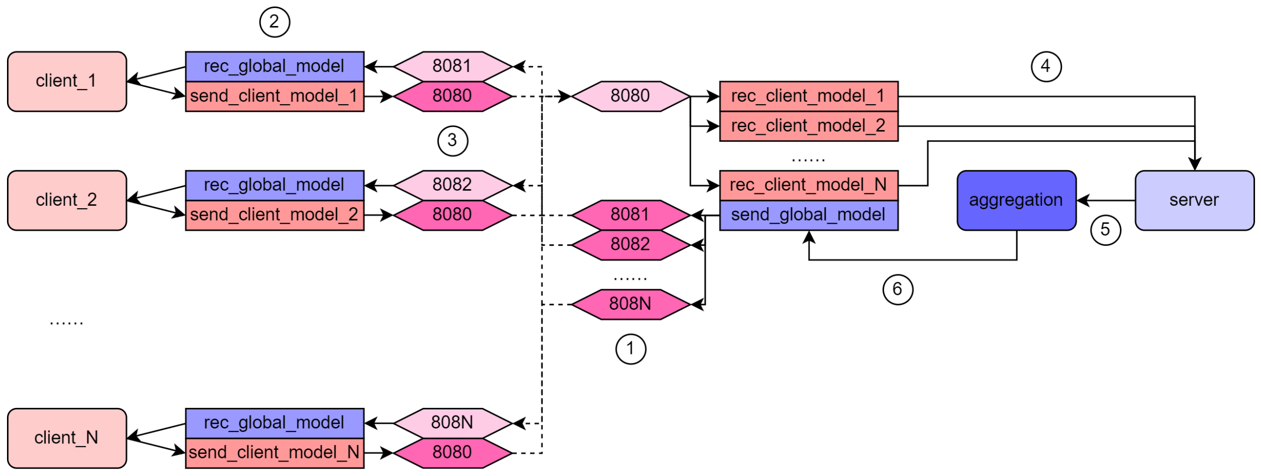

Figure 3: N-Client FL System with Socket

process receives model files from each client processes at port 8080 and each client process i receives model file from server process at port $8080 + i$. Like stage1, there are 6 steps for a round of training and update.

1. Server process sends the `global_model` to listening ports of client processes.
2. Client processes load the received `global_model` as their local model, finish their local training and save their `client_model`.
3. Client processes send the saved `client_model` to the server process's listening port 8080.
4. Server process receives all `client_model` from each client process and loads them.
5. Server process aggregates parameters of all the N `client_model`.
6. Server process saves the updated `global_model`, back to step 1.

3 Implementation

For all 3 stages, they share the same code architecture. We implement a `model.py` to define a convolution neural network for the classifier. We implement a `dataset_manager.py` to process and provide train dataset and test dataset. We also implement `client_process.py` and `server_process.py` for the training and update process. There are mainly two difficulties for our implementation. The first is the implementation of FedAVG algorithm. The second is the implementation of socket communication.

3.1 Implementation of FedAVG

Algorithm 1 FedAVG

```
1: sumTensor = sum(clientModels)
2: sumTensor /= clientNum
3: return sumTensor
```

As pseudo-code 1, we simply sum parameters of all client models and divide by the number of clients. With this method, we get a simply averaged model from client models.

3.2 Implementation of Socket Communication

Algorithm 2 SendWithSocket

```
 1: mySocket=socket()
 2: mySocket.connect(ip, port)
 3: file = open(filePath)
 4: line=file.read(1024)
 5: while line do
 6:     mySocket.send(line)
 7:     line=file.read(1024)
 8: end while
 9: file.close()
10: mySocket.close()
```

As pseudo-code 2 and 3, we implement two algorithms to send files and receive files through socket. In order to avoid data error during transmission, we send 1024 bytes each turn until the whole file is sent. In order to avoid missing data, we keep spinning until receive the first 1024 byte.

4 Experiment

Since in the experiment we have 20 client processes and each process may cost about 900 MB VRAM if we try to train on GPU, we have no choice but conduct our experiment on CPU. Our experiment is conducted on `Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz` with 24 cores in total and 128 GB RAM, which means we can assign a core for each process (include 20 client processes and 1 server process). In practice, we assign core 0 to server process and core 1 to 20 to client processes.

Algorithm 3 RecvWithSocket

```
mySocket=socket()
mySocket.bind(ip, port)
mySocket.listen()
con, addr = mySocket.accept()
file = open(filePath)
line = con.recv(1024)
while len(line)==0 do
    line = con.recv(1024)
end while
while line do
    file.write(line)
    line = con.recv(1024)
end while
file.close()
mySocket.close()
```

Our experiment focuses on the difference between difference FL systems, so the model we use and the hyper parameters for the model are not changed between difference experiments. With module `torchsummary`, the model's architecture is as below:

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 26, 26]             320
            Conv2d-2           [-1, 64, 24, 24]          18,496
         Dropout2d-3           [-1, 64, 12, 12]               0
            Linear-4                  [-1, 128]       1,179,776
         Dropout2d-5                  [-1, 128]               0
            Linear-6                   [-1, 10]           1,290
================================================================
Total params: 1,199,882
Trainable params: 1,199,882
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.52
Params size (MB): 4.58
Estimated Total Size (MB): 5.10
----------------------------------------------------------------
```

We use `nn.CrossEntropyLoss` to compute the loss for our model, use `optim.SGD` as the model's optimizer. The learning rate we use is 0.0001 and the momentum is 0.5.

4.1 Stage 1

For Stage 1, we conduct the experiment with 20 client processes and 50 epochs. By analyzing log files of the experiment, we can plot the Fig.4. The chart at left is the loss for local model at each client, the chart at middle is accuracy for the global model, and the chart at right is loss for global model at server.

As Fig.4, for the global model on server, we can see that the accuracy increases continuously and the loss drops continuously until the model converges. For client models, the reason why their loss curves have violent gyrations is that we record the loss for local models at the end of each batch. With more frequent sampling, loss curves for local models have more oscillation but still continuously drop in general.

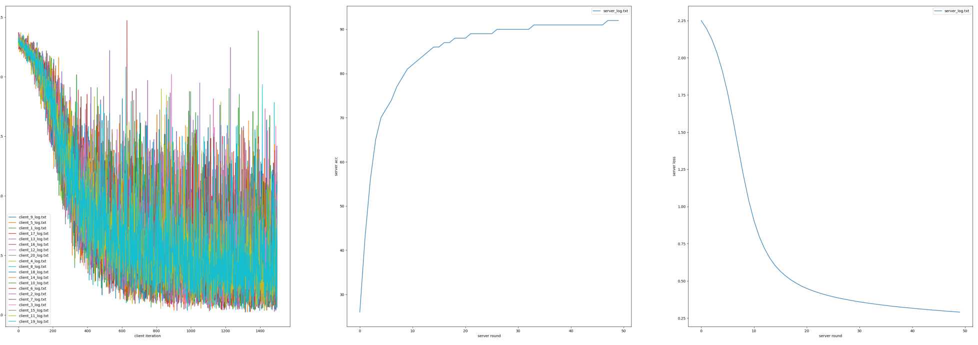

Figure 4: Experiment Result for Stage 1

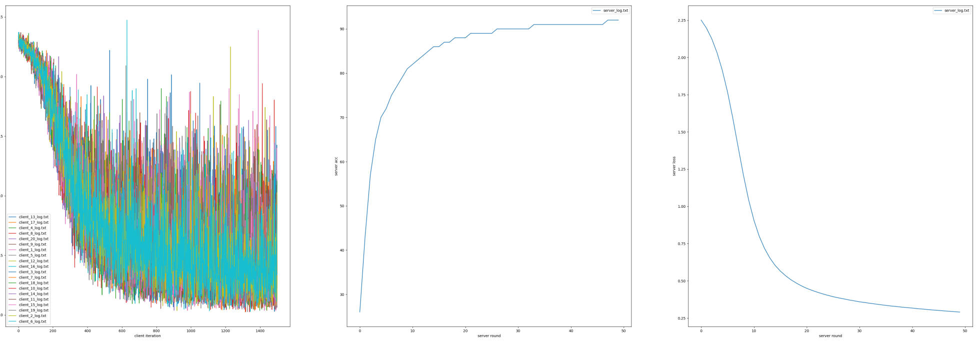

Figure 5: Experiment Result for Stage 2 $m = 0.8$

4.2 Stage 2

For Stage 2, we still conduct the experiment with 20 client processes and 50 epochs. In order to compare the difference between different M, we conduct 4 experiments with the selection ratio $m = 0.8(M = 0.8 \times 20 = 16)$, $m = 0.4(M = 8)$, $m = 0.2(M = 4)$ and $m = 0.1(M = 2)$.

As Fig.5, Fig.6, Fig.7 and Fig.8, it's obvious that with different selection ratio, we can achieve about the same accuracy for our model 20 client processes and 50 epochs. However, if we focus on the time cost on aggregation at the server process, we will find a great difference between difference selection ratio. The Table.1 below are average time cost on model aggregation for servers with different selection ratio. It's easy to understand that with smaller selection ration,

Table 1: Average Time Cost on Aggregation

m	0.8	0.4	0.2	0.1
time cost	0.2132	0.1130	0.0576	0.0321

the average time cost on aggregation becomes smaller.

4.3 Stage 3

For Stage 3, we still conduct the experiment with 20 client processes and 50 epochs which is the same setup for the experiment at Stage 1. Since the only difference between Stage 1 and Stage 3 is that we use socket communication

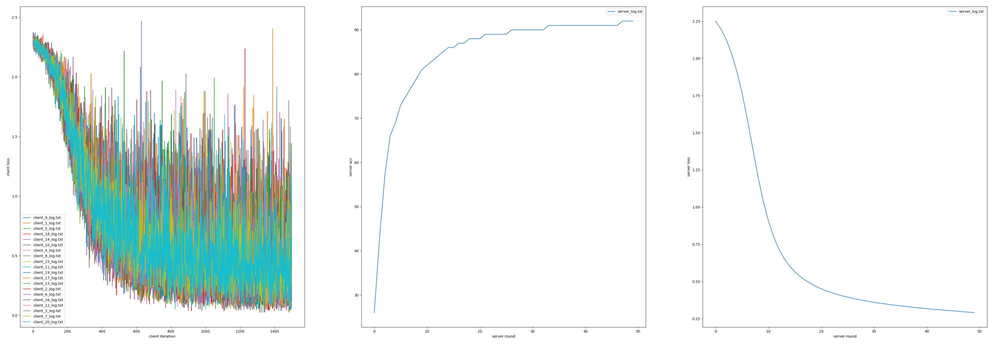

Figure 6: Experiment Result for Stage 2 $m = 0.4$

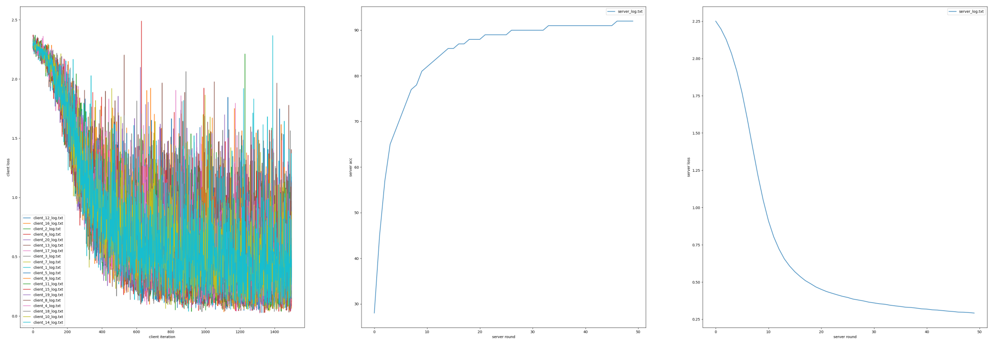

Figure 7: Experiment Result for Stage 2 $m = 0.2$

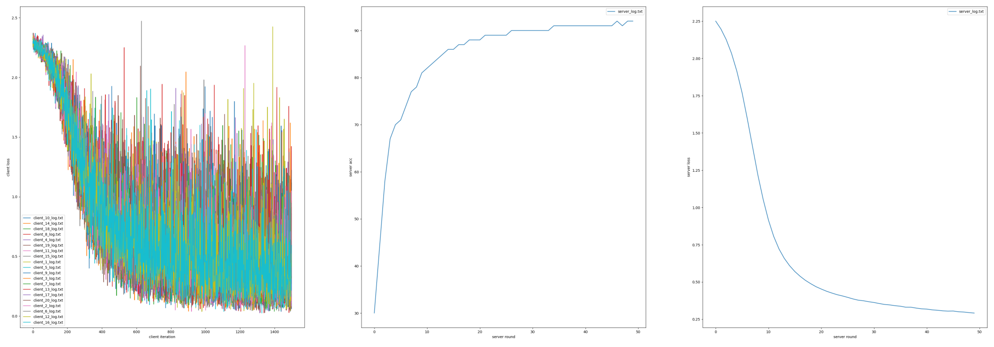

Figure 8: Experiment Result for Stage 2 $m = 0.1$

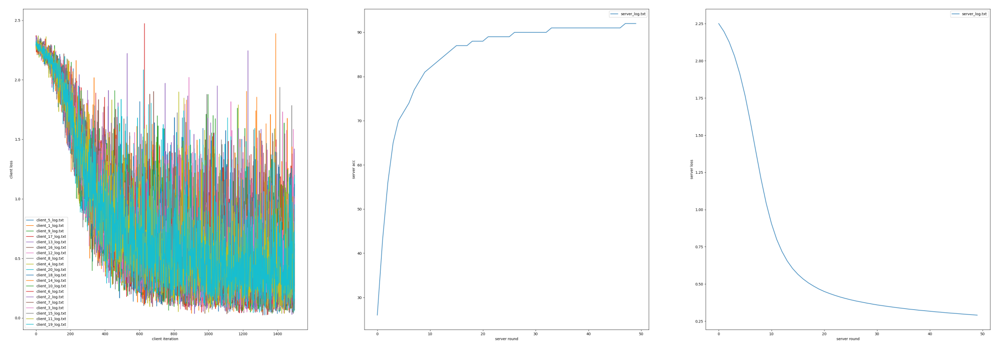

Figure 9: Experiment Result for Stage 3

instead of shared folder to exchange model files, the ultimate result of experiment at Stage 3 is quite similar to the result of Stage 1. The result is as Fig.9.

5 Conclusion

In this project, we implement 3 different FL systems. With our experiments, we have proved that those systems can train the required MINIST classification model properly. The loss drops continuously and the accuracy increases continuously and ultimately achieves about 92%. In conclusion, we implement and prove the feasibility of those 3 stages systems. The source code can be found at `https://github.com/LaGrange151235/CS3511_FL_Project`. And training records can be found at `https://jbox.sjtu.edu.cn/v/link/view/c98a613a3f40452c93514bd3c1a8d941`.

References

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

[2] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloé Kiddon, Jakub Konečný, Stefano Mazzocchi, Brendan McMahan, Timon Van Overveldt, David Petrou, Daniel Ramage, and Jason Roselander. Towards federated learning at scale: System design. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 374–388, 2019.