A Peek into Federated Learning

Name: **Guantao Chen** **ID**: **23336035**

Abstract: As the pioneering work of Federated Learning, the paper *Communication-Efficient Learning of Deep Networks from Decentralized Data* by McMahan et al. has been widely recognized, providing insights into how heterogeneous data can be used to train a global model without actually feeding the data to a central server. In this article, I would like to present my understanding of which and replicate a fraction of the experiments conducted in the paper.

Keywords: Federated Learning, Heterogeneity, Replication

1 Introduction

The benefits of Federated Learning are widely recognized, as it allows for the training of a global model without the need to share data with a central server. Each client performs local computation on its own data over a specified number of epochs, and then sends the model updates to the server. The server aggregates the updates from all clients and updates the global model accordingly. All clients then update their local models with the new global model. Then the process repeats.

In my replication program, several hyperparameters are used in the training process and listed as follows:

- **num_clients**: The number of clients participating in the training process.
- **C**: The fraction of clients that are selected to participate in each round.
- **epoch**: The number of epochs each client trains the model on its local data.
- **batch_size**: The number of samples used in each iteration of the training process.
- **learning_rate**: The step size of the optimization algorithm.
- **rounds**: The number of rounds of training, namely the number of times the server updates the global model.

In comparison, I also trained a global model using the traditional centralized approach, where all data is fed to a central server and the model is trained on the entire dataset. The hyperparameters used in which are basically the same as above, except for the **epoch** parameter. Here it represents the number of epochs the model is trained on the entire dataset, corresponding to the hyperparameter **rounds** in the Federated Learning approach.

Both of the models are trained on the MNIST dataset, consisting of 60,000 training samples and 10,000 test samples. The optimizers are set to be SGD from *torch.optim*, and the loss function is set to be *CrossEntropyLoss* from *torch.nn*.

The results of the losses are both ploted in the same figure using *matplotlib.pyplot*, a line of y=0.99 is also plotted to show the convergence of the models. The complete code implementation can be found at the end of this article.

2 Experimental Results

Due to time limitations, I only trained the models using IID.

I first tried to train the models with C=0.1, num_clients=10, epoch=20, batch_size=64, learning_rate=0.01, rounds=500. The results are shown in Figure 1& 2.

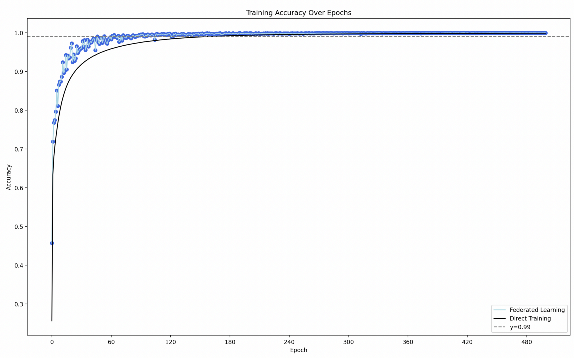

Figure 1: 500 epochs

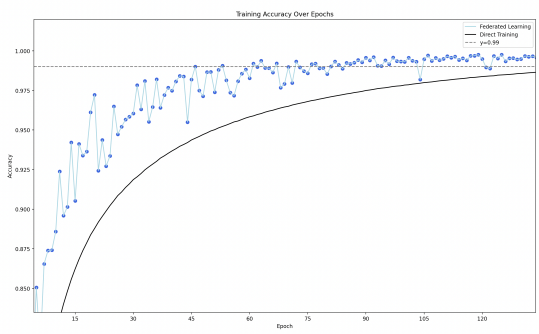

Figure 2: Enlarged figure

We can see that the data points are overwhelmingly above the line of y=0.99 after 80 epochs. A more thorough analysis is listed below.

2.1 Changing Epochs

The following results are obtained by changing the number of epochs in the range of [1,5,10,20] and [1,5,10,20,50] while other hyperparameters remain the same, which are: C = 0.1, num_clients = 10, learning_rate = 0.01, rounds = 50. The former has batch_size 10 and the latter has batch_size 50. The results are as follows:

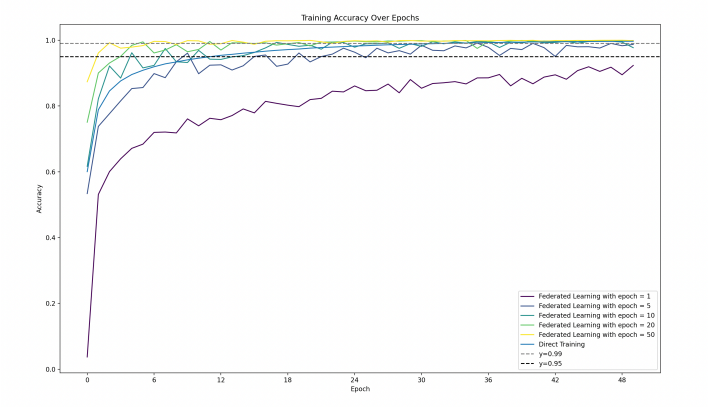

Figure 3: batch_size=10

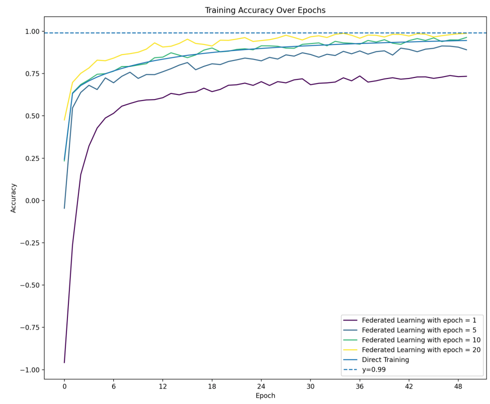

Figure 4: batch_size=50

It is clear that Figure 3 shows a faster convergence than Figure 4, and among all the lines in Figure 3, convergence is the fastest when epoch=50. Not only are the results consistent with the intuition that the more epochs the model is trained, the better the performance, but also the results in the original paper, showing that the model reaches 99% accuracy after 34, 20, 18 rounds respectively for epochs 1, 5, 20.

2.2 Changing *C*

The following results are obtained by changing the fraction of clients that are selected to participate in each round in the range of [0.0,0.1,0.2,0.5,1.0] while other hyperparameters remain the same, which are: C = 0.1, num_clients = 10, learning_rate = 0.01, batch_size = 64, rounds = 50. The results are shown in figure 5.

From the figure, all choices of C converge to 95% accuracy after 25 rounds, but oscillate around 99% even around 50 rounds. It is hard to tell which choice of C converge the fastest, but when computation time is taken into consideration, C = 0.1 seems to be the best choice.

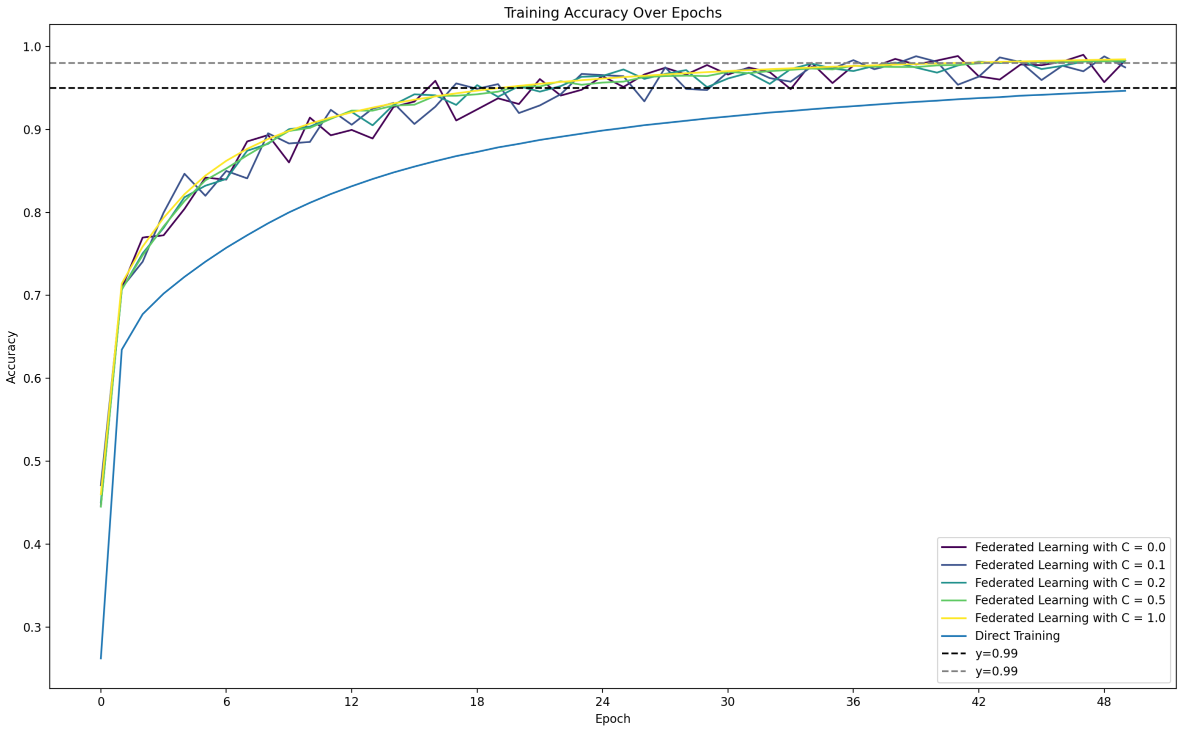

Figure 5: Changes in C

3 Reflections and Future Work

In my replication, I have only trained the models using IID, whereas the quaintessence of Federated Learning lies in the heterogeneity of the data, namely the non-IID setting. Also, it is surprising to see that the Federated Learning approach converges faster than the traditional centralized approach, the reason for which is still unclear to me.

The mathematical approach with which to calculate the accuracy rate is also open to discussion. In this article I robustly used $Accuracy = 1 - CrossEntropyLoss$ as the accuracy rate, which now seems to me naive and inaccurate. It maybe better to use the accuracy rate calculated from the test set.

Code optimization is also a problem. When compared to the direct SGD training, my Federated Learning approach converges much slower in reaching the desired accuracy rate.

Finally, I haven't completely grasped how to utilize my gpu to accelerate the training process. In one training, it took RTX 4070ti 634 seconds to train 50 rounds, while i7-13700kf and Apple M2 respectively spent 519 and 419 seconds.

References

[1] McMahan, H. Brendan, et al. "Communication-efficient learning of deep networks from decentralized data." *arXiv preprint arXiv:1602.05629* (2017).

[2] Sebastian Raschka, Vahid Mirjalili. *Machine Learning with Pytorch and Scikit-Learn*. China Machine Press, 2023.

A Complete Code Implementation

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader, random_split
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator
import itertools
import copy
import random
import time

class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN,self).__init__()
        self.fc1 = nn.Linear(28*28,128)
        self.fc2 = nn.Linear(128,10)

    def forward(self,x):
        x = x.view(-1,28*28)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

def get_data_loaders(batch_size,num_clients):
    transform = transforms.Compose([transforms.ToTensor(),transforms.
        Normalize((0.5,),(0.5,))])
```

```
    dataset = datasets.MNIST(root='./data',train=True,download=True,
        transform=transform)

    client_datasets = random_split(dataset, [len(dataset) // num_clients] *
         num_clients)
    client_loaders = [DataLoader(ds,batch_size=batch_size,shuffle=True) for
         ds in client_datasets]
    return client_loaders

def client_update(client_model,optimizer,train_loader,epochs):
    client_model.train()
    epoch_losses = []
    for epoch in range(epochs):
        total_loss = 0
        for data,target in train_loader:
            optimizer.zero_grad()
            output = client_model(data)
            loss = nn.CrossEntropyLoss()(output,target)
            loss.backward()
            optimizer.step()
            total_loss += loss.item()
        average_loss = total_loss/len(train_loader)
        epoch_losses.append(average_loss)
    return epoch_losses

def average_weights(global_model,client_models):
    global_dict = global_model.state_dict()
    for k in global_dict.keys():
        global_dict[k] = torch.stack([client_models[i].state_dict()[k].
            float() for i in range(len(client_models))],0).mean(0)
    global_model.load_state_dict(global_dict)  # Update the global model

def plot_losses(federated_losses, direct_losses):
    num_rounds = len(federated_losses)
    num_clients = len(federated_losses[0])

    # Plot the curve using Federated Learning
    avg_losses = []
    avg_accuracies = []
```

```
    for round in range(num_rounds):
        round_losses = federated_losses[round]
        flat_round_losses = list(itertools.chain(*round_losses))
        avg_loss = sum(flat_round_losses) / len(flat_round_losses)
        avg_accuracy = 1 - avg_loss
        avg_accuracies.append(avg_accuracy)
        avg_losses.append(avg_loss)
        plt.scatter(round,1-avg_loss, color='royalblue', marker='o')


    plt.plot(range(num_rounds), avg_accuracies, color='lightblue',
        linestyle='-', label='Federated Learning')


    for i in range(len(direct_losses)):
        direct_losses[i] = 1 - direct_losses[i]
    # Plot the curve using traditional SGD
    plt.plot(direct_losses, label='Direct Training', linestyle='-', color='
        black')

    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.title('Training Accuracy Over Epochs')
    # plt.ylabel('Loss')
    # plt.title('Training Loss Over Epochs')

    plt.axhline(y=0.99, color='grey', linestyle='--', label='y=0.99')

    plt.legend()
    plt.gca().xaxis.set_major_locator(MaxNLocator(integer=True))
    plt.show()

def direct_training(epochs=20):
    batch_size = 64
    learning_rate = 0.01

    model = SimpleNN()
    train_loader = get_data_loaders(batch_size, 1)[0]
    optimizer = optim.SGD(model.parameters(), lr=learning_rate)
```

```
    criterion = nn.CrossEntropyLoss()
    losses = []

    for epoch in range(epochs):
        total_loss = 0
        for data, target in train_loader:
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()
            total_loss += loss.item()
        average_loss = total_loss / len(train_loader)
        losses.append(average_loss)
        print(f"SGD training epoch {epoch+1} comlete, average loss:{average_loss}")

    return losses

def federated_learning(rounds=20):

    num_clients = 10
    batch_size = 64
    learning_rate = 0.01
    epoch = 20
    C = 0.1

    global_model = SimpleNN()
    client_loaders = get_data_loaders(batch_size, num_clients)
    all_client_losses = []

    for round in range(rounds):
        client_models = [SimpleNN() for _ in range(num_clients)]
        for client_model in client_models:
            client_model.load_state_dict(global_model.state_dict())

        optimizer = [optim.SGD(model.parameters(), lr=learning_rate) for model in client_models]
        round_losses = []
```

```

        # Randomly select clients to participate in the round
        m = max(int(C * num_clients), 1)  # Choosing at least 1 client
        selected_clients = random.sample(range(num_clients), m)

        client_losses = []
        for i in selected_clients:
            client_losses = client_update(client_models[i], optimizer[i],
                client_loaders[i], epoch)
            round_losses.append(client_losses)

        average_weights(global_model, [client_models[i] for i in
            selected_clients])
        print(f"Federated Learning round {round+1} complete, average loss:
             {sum(client_losses)/len(client_losses)}")
        all_client_losses.append(round_losses)

    return all_client_losses

def main():
    rounds = 500

    start_time = time.time()
    federated_losses = federated_learning(rounds)
    direct_losses = direct_training(rounds)
    print("Time elapsed: ", time.time() - start_time)
    plot_losses(federated_losses, direct_losses)

if __name__ == "__main__":
    main()
```

Listing 1: Complete Code Implementation