REPORT

Implementing Federated Averaging (FedAvg)

Subject	**Bigdata Processing (MS)**	**Professor**	**Jae-Yeon-Park**
ID	**32204292**	**Major**	**Mobile System Eng.**
Name	**Min-hyuk-Cho**	**Submit Date**	**2024/12/20**

Index

1. Introduction

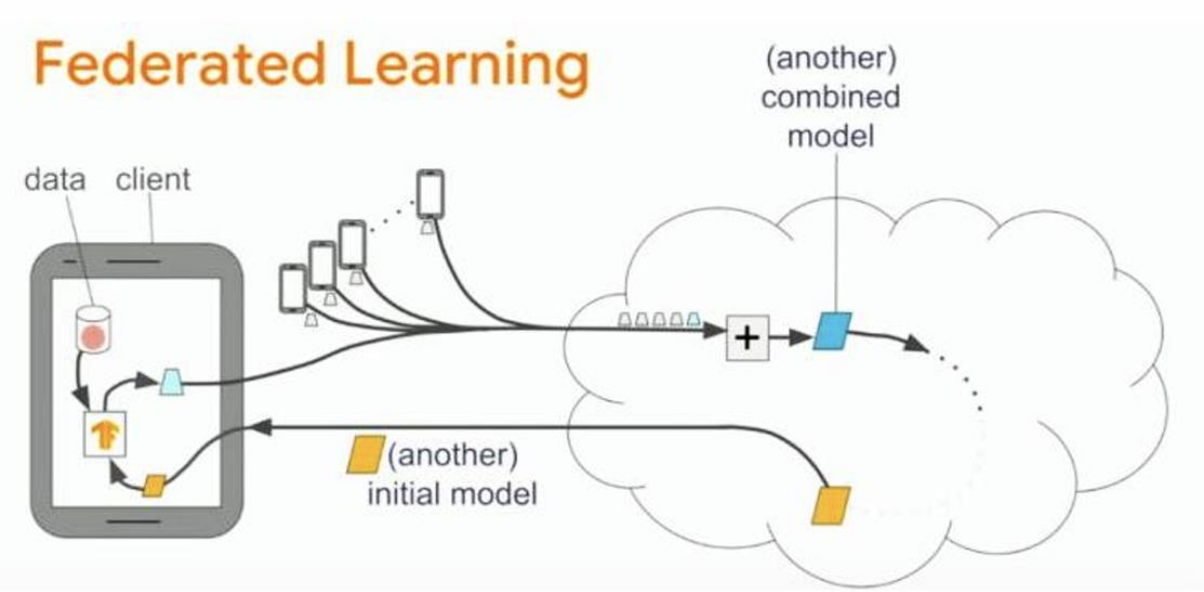

Fig 1. Federated Learning

With the rapid advancements in smartphones and internet technologies, the amount of data generated and accumulated online has been growing exponentially. It is projected that by 2025, approximately 175ZB of data will exist globally. This vast amount of information, often referred to as "big data," is predominantly unstructured, comprising about 90% of the total. Effectively processing such massive datasets and transforming them into forms suitable for AI and machine learning (ML) applications has become increasingly critical.

However, challenges in data processing extend beyond technical efficiency. Protecting user privacy and ensuring data security are paramount concerns. A promising solution to these issues is Federated Learning, which enables decentralized model training without requiring centralized data collection. Federated Learning operates by distributing an initial model to local devices, allowing them to train the model individually, and then sending only the updated model parameters back to a central server for aggregation. This approach reduces communication overhead and effectively preserves user privacy.

This report focuses on implementing the FedAvg (Federated Averaging) algorithm, one of the most widely adopted algorithms in Federated Learning. The structure of this report is as follows: Chapter 2 discusses the requirements for implementing the program. Chapter 3 delves into the fundamental concepts relevant to Federated Learning. Chapter 4 presents the implementation of the FedAvg algorithm. Chapter 5 evaluates the program through a series of experiments and analyzes the results. Finally, Chapter 6 concludes the report with a summary and insights gained from the study.

2. Requirements

Table 1. Requirements Table

Index	Requirements
1	Dataset: MNIST, 60,000 training images and 10,000 testing images (Handwritten digits, 28x28 grayscale image)
2	Predefined Code: Local Client Distribution and Testing 1. For accurate evaluation, code for distributing the MNIST dataset to local clients (or local devices) has already been provided. Each client will have its own local subset of the dataset to simulate federated learning. 2. Additionally, a testing function to evaluate the global model at each training round is also included. This ensures that the accuracy of the model can be monitored as training progresses. Important Note: The code for data distribution to local clients and the testing function must not be modified under any circumstances. These components are critical for the consistency and fairness of the evaluation process. Any alterations will invalidate the results and will not be accepted.
3	Task 1: • Implement the following two functions to complete the FedAvg framework: 1. average_weights(selected_models): A function to aggregate the model weights from selected clients during federated learning. 2. federated_training(num_rounds, num_clients, client_fraction, local_epochs, train_loaders, test_loader, lr=0.001): A function to perform federated training by coordinating clients, updating the global model, and testing it after each round. • After completing the implementation, document the training results for the federated learning process.
4	Task 2: • Set the number of participants (num_clients) to 60, local epochs to 1, and the learning rate to 0.001. • Train the model for 20, 30, and 40 rounds, and plot the training results as a graph with: • x-axis: Number of rounds • y-axis: Accuracy • Discuss the outcomes and explain why these results were observed.
5	Task 3: • Set the number of participants (num_clients) to 60, learning rate to 0.001, and the number of rounds to 20. • Train the model with local epochs set to 1, 5, and 10, and plot the training results as a graph with: • x-axis: Number of rounds • y-axis: Accuracy • Discuss the outcomes and explain the effect of changing the number of local epochs on the learning process.
6	Task 4: • Set the number of participants (num_clients) to 100, learning rate to 0.001, number of rounds to 20, and local epochs to 1. • Train the model with client_fraction set to 0.01, 0.05, and 0.1, and plot the training results as a graph with: • x-axis: Number of rounds • y-axis: Accuracy • Discuss the outcomes and explain how varying the fraction of participating clients affects the learning process.

3. Concepts

3-1. Federated Learning

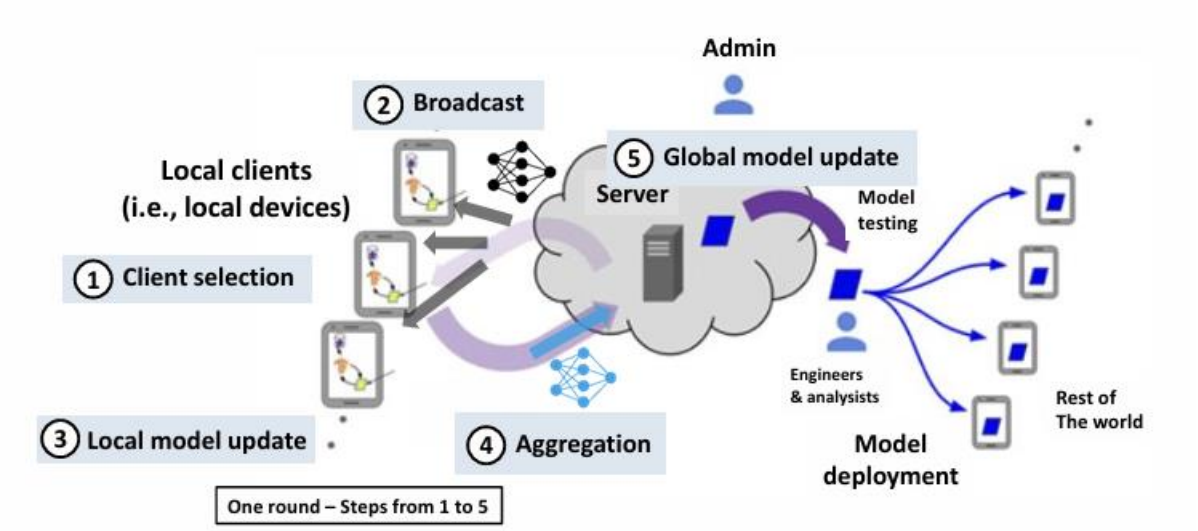

Fig 2. Step of Federated Learning

Federated Learning was briefly introduced in Chapter 1. To better understand its principles, this section delves into the learning process in detail. Fig. 2 visually illustrates the Federated Learning workflow, which comprises the following key steps:

i. Client Selection
The central server selects clients to participate in the training process. These clients can be devices connected to the network, such as smartphones or IoT devices. The selection process considers factors like network conditions, device computational capacity, and energy consumption to optimize participation.

ii. Broadcast
The central server distributes an untrained initial global model to the selected clients. This initial model may be randomly initialized or derived from a previously trained global model. By starting with the same initial model, Federated Learning avoids biases toward specific client datasets, ensuring a more balanced and generalizable model.

iii. Local Model Update
Each client trains the received model using its local dataset. As the training occurs locally, no raw data is transmitted to the central server, ensuring user privacy. The local training process involves standard machine learning operations, including gradient computation and model updates based on local data.

iv. Aggregation
After local training, clients upload the updated model parameters (weights and biases) to the central server. This step transmits only the learning outcomes, not the raw data itself, minimizing the risk of data leakage.

v. Global Model Update
The central server aggregates the parameters received from clients to update the global model. Methods like weighted averaging (e.g., the FedAvg algorithm) are used to combine client updates into a refined global model. During this process, gradients are computed to minimize the loss function and enhance overall model performance.

1. Iterative Training
The updated global model is redistributed to clients, and the process is repeated iteratively. Over multiple rounds, the model progressively improves, eventually achieving a robust global model capable of generalizing across diverse client datasets.

Federated Learning offers several advantages:

i. Privacy Preservation: Data remains on local devices, preventing sensitive information from being shared with the central server.

ii. Reduced Communication Costs: Only model parameters are exchanged, significantly lowering network overhead compared to traditional centralized learning.

iii. Decentralized Learning: Leveraging local data enables the creation of personalized models tailored to each client's needs.

3-2. Challenges of Federated Learning

Federated Learning is an innovative technology that ensures privacy protection while enabling personalized models. However, several significant challenges must be addressed for its successful implementation and application.

i. Communication Overhead
One of the advantages of Federated Learning is the reduced communication cost, as only model parameters are exchanged rather than raw data. However, the iterative communication process between the central server and clients can become costly as the number of participating clients increases. In environments with limited network bandwidth, this communication overhead may create a bottleneck. Additionally, larger model sizes result in increased data transfer, further escalating the cost.

ii. Computation Overhead - Local and Global Computational Cost
Federated Learning involves computational costs at both the local and global levels. On local devices, limited processing power, energy constraints, and heat generation can be significant concerns. On the central server, as the number of participating clients grows, the computational load for aggregating and optimizing the global model increases, potentially slowing down the overall training process.

iii. Non-Independent and Identically Distributed (Non-IID) Data
In Federated Learning environments, client data is rarely uniformly distributed. Each client may hold data specific to their usage patterns, leading to skewed distributions. Such Non-IID data poses challenges for training a generalized global model and increases the risk of model bias toward specific client datasets. This imbalance can compromise both model fairness and performance.

iv. Heterogeneous Devices
The devices participating in Federated Learning vary widely in terms of hardware capabilities, network speeds, and energy resources. This disparity leads to uneven training speeds among clients, potentially delaying global updates in synchronous learning setups. Client heterogeneity impacts not only the training efficiency but also the overall system stability.

v. Heterogeneous Models
Due to differences in computational capacity, clients may require models with varying architectures. Some devices may handle complex deep learning models, while others may need lightweight models. This structural diversity complicates the aggregation of global models and can result in degraded performance unless efficient layer-sharing and model integration methods are applied.

To overcome these challenges, various algorithms and optimization methods have been proposed. This report introduces two such algorithms detailed in later.

3-3. FedSGD (Federated Stochastic Gradient Descent)

$$\circ\, \theta_{j+1} := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \frac{\partial J(\theta; x_i, y_i)}{\partial \theta_j}$$

Fig 3. Formular of FedSGD

Federated Stochastic Gradient Descent (FedSGD) is one of the foundational algorithms employed in Federated Learning. Similar to the concept of Stochastic Gradient Descent (SGD), it operates probabilistically. The model training process begins with parameters initialized to random values. Gradients are then computed based on these parameters and the optimization objective. Subsequently, the parameters are adjusted in the direction opposite to the gradient to iteratively minimize the loss function. This process continues until the loss function converges to its minimum value. As an optimization algorithm, the ultimate goal of FedSGD is to discover the most optimized model parameters.

Fig 3 illustrates the mathematical formulation of this process, specifically showcasing how gradients computed by each local device are aggregated and averaged to update the global model parameters. The central principle of FedSGD lies in its efficiency: each local device computes the gradient once and transmits it to the central server. This design allows Federated Learning to be effectively performed on devices with limited computational capabilities.

However, this advantage comes with a trade-off. The frequent communication between the local devices and the central server results in increased communication costs, which can be a bottleneck in practical implementations.

3-4. FedAvg (Federated Averaging)

Algorithm 1 `FederatedAveraging`. The K clients are indexed by k; B is the local minibatch size, E is the number of local epochs, and η is the learning rate.

Server executes:
initialize w_0
for each round $t = 1, 2, \dots$ **do**
  $m \leftarrow \max(C \cdot K, 1)$
  $S_t \leftarrow$ (random set of m clients)
  for each client $k \in S_t$ **in parallel do**
    $w_{t+1}^k \leftarrow \text{ClientUpdate}(k, w_t)$
  $w_{t+1} \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} w_{t+1}^k$

ClientUpdate(k, w): *// Run on client k*
$\mathcal{B} \leftarrow$ (split $\mathcal{P}_k$ into batches of size B)
for each local epoch i from 1 to E **do**
  for batch $b \in \mathcal{B}$ **do**
    $w \leftarrow w - \eta \nabla \ell(w; b)$
return w to server

17

Fig 4. Algorithm of FedAvg

Federated Averaging (FedAvg) is an enhanced algorithm built upon FedSGD, designed to address its communication cost challenges. Unlike FedSGD, where each client sends gradients to the central server after a single local computation, FedAvg allows clients to perform multiple local model updates before transmitting parameters to the central server. Each client executes local gradient descent multiple times based on its local dataset. Once the local computations are complete, clients send their updated parameters to the server, which aggregates them by averaging to update the global model.

Fig 4 provides a detailed illustration of the FedAvg algorithm. At the beginning of each round, the server randomly selects a subset of clients to participate in training. These selected clients then execute Client Update processes in parallel. On the client side, a batch size is allocated, and weights are computed over several epochs based on the local data. After completing the local computations, the clients transmit their model parameters to the server. The server aggregates the parameters using a weighted averaging scheme to update the global model.

FedAvg effectively reduces communication costs by limiting the frequency of interactions between the server and clients. Additionally, it leverages the computational resources of local devices more efficiently by performing multiple updates locally.

Despite its advantages, FedAvg presents several challenges. Selecting optimal hyperparameters, such as the number of local epochs and batch size, is critical for achieving efficient training. Moreover, as the algorithm requires repeated computations on local devices, its practical implementation can be limited by the computational capabilities of resource-constrained devices.

4. Implements

4-1. Build environment

Before explaining the implemented code, the development environment and setup used to run the program are as follows:

- ✔ Development Environment: Colab
- ✔ Programming Language: Python
- ✔ Program Execution Steps:

STEP 1) Enter the Colab -> Open Note -> CTRL+F9 (All of shell execute)

4-2. Implementation FedAvg Program

```
# Function to average weights from selected clients
def average_weights(selected_models):
    avg_state_dict = deepcopy(selected_models[0])

    for key in avg_state_dict.keys():
        for model in selected_models[1:]:
          avg_state_dict[key] += model[key]
        avg_state_dict[key] /= len(selected_models)

    return avg_state_dict
```

Fig 5. average_weights function of FedAvg

Fig 5 illustrates the implementation of the average_weights function, a key component of the FedAvg algorithm. This function is responsible for aggregating model parameters from multiple clients by computing their weighted average.

i. Initialization:
The function begins by receiving a list of models from the clients. It creates a deep copy of the first model to initialize the aggregated model. This ensures that the structure of the resulting model matches that of the input models while maintaining independence from the original model references.

ii. Aggregation:
Using a loop, the function iterates over the remaining models in the list. For each model,

it accesses the parameters (weights) by their keys. The parameters from all models are summed together during this process.

iii. Averaging:
After completing the summation, the function calculates the average of the parameters by dividing the accumulated values by the number of models. This is performed for each parameter key to ensure the proper averaging of all model weights.

iv. Return:
Once the averaging is complete, the function returns the aggregated model containing the averaged weights.

```
# Federated training function with client fraction C and test accuracy measurement
def federated_training(num_rounds, num_clients, client_fraction, local_epochs, train_loaders, test_loader, lr=0.001):
    global_model = SimpleCNN()
    global_model_state = global_model.state_dict()
    accuracy_list = []
    num_selected_clients = max(1, int(client_fraction * num_clients))

    for round_num in range(num_rounds):
      print(f"₩nRound {round_num + 1}/{num_rounds}")

      selected_clients = random.sample(range(num_clients), num_selected_clients)

      selected_models = []

      for client_id in selected_clients:
        local_model = deepcopy(global_model)
        local_weights = train_local(local_model, train_loaders[client_id], epochs = local_epochs, lr = lr)
        selected_models.append(local_weights)

      global_model_state = average_weights(selected_models)
      global_model.load_state_dict(global_model_state)

      accuracy = test_model(global_model, test_loader)
      accuracy_list.append(accuracy)
      print(f"Accuracy: {accuracy:.2f}%")

    return global_model, accuracy_list
```

Fig 6. federated_training function of FedAvg

Fig 6 presents the implementation of the federated_training function, which orchestrates the training process in a Federated Learning setup. Below is a detailed breakdown of the code:

i. Initialization: A global model (global_model) is created and initialized, and its parameters are stored in a state dictionary (global_model_state). An empty list (accuracy_list) is initialized to keep track of the test accuracy after each training round.

ii. Client Selection: For each training round, the number of clients to participate is determined based on the client_fraction and the total number of clients (num_clients). The max function ensures at least one client is selected. A random sample of clients is selected for the current round using random.sample.

iii. Local Training: For each selected client, a local model is initialized as a deep copy of

the global model to ensure independent updates. The local model is trained on the client's dataset using the train_local function, which performs training for the specified number of local epochs (local_epochs) and learning rate (lr). The trained weights from each client's local model are added to a list (selected_models).

iv. Global Model Update: After all selected clients have completed their local training, the global model parameters are updated by averaging the weights of the local models using the average_weights function. The global model's state dictionary is updated with the new aggregated weights.

v. Accuracy Evaluation: The updated global model is evaluated on a test dataset using the test_model function, and the accuracy is calculated. The calculated accuracy is appended to accuracy_list, and it is printed to monitor progress after each round.

vi. Output: At the end of the training process, the function returns the final global model and the list of accuracy values, which can be used for visualization or analysis.

5. Results & Evaluation

5-1. Task 2 - Effect of changing round test

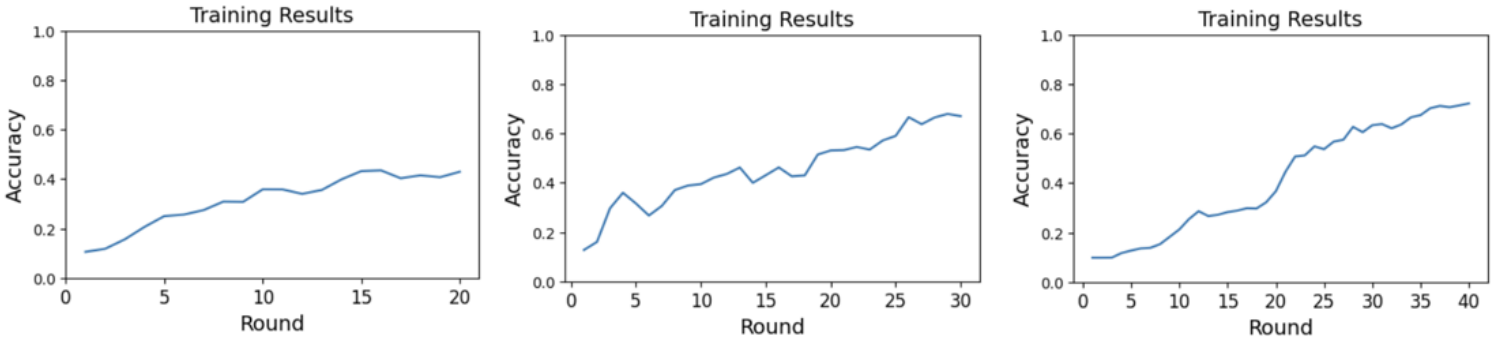

Fig 7. Task 2 - (Round = 20, 30, 40, Left to Right)

Fig 7 illustrates the changes in accuracy for Task 2 as the number of rounds increases. Across all three graphs, accuracy shows a clear upward trend. Initially, all three configurations start with an accuracy of approximately 10%. By the end of 20 rounds, the accuracy reaches about 40%, whereas the 30-round and 40-round configurations achieve accuracy levels of around 70% to 80%. This demonstrates that increasing the number of rounds allows the global model to learn more effectively and refine its understanding of the data. The observed improvement in accuracy with additional rounds highlights that repeated training enables the model to continuously capture and learn essential features.

However, while accuracy significantly improves from 20 to 30 rounds, the improvement from 30 to 40 rounds is relatively smaller. This indicates diminishing returns as the number of rounds increases. Given the negligible difference in accuracy between 30 and 40 rounds, it can be

concluded that 30 rounds strike a balance between achieving high accuracy and minimizing communication and computation costs. Therefore, 30 rounds can be considered the optimal number of rounds in this scenario.

5-2. Task 3 - Effect of changing local epochs test

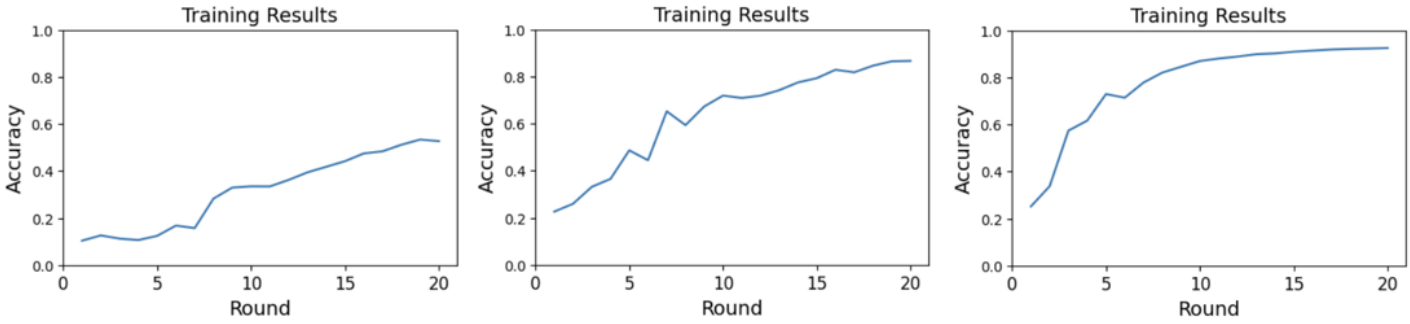

Fig 8. Task 3 – (Local Epochs = 1, 5, 10, Left to Right)

In all the graphs of Fig 8, accuracy consistently increases as the number of rounds progresses. This demonstrates that the iterative global model updates in Federated Learning continuously improve learning performance. Notably, the greater the number of Local Epochs, the faster the accuracy improves in the initial rounds, and the convergence speed also increases.

With Local Epoch = 1, accuracy increases gradually as the rounds progress, eventually reaching about 50%. Local Epoch = 5 shows faster learning compared to Local Epoch = 1, achieving a higher accuracy of around 80%. Local Epoch = 10 reaches a high accuracy of approximately 90% in the early rounds, showing near-convergence even with fewer rounds.

As the number of Local Epochs increases, each client learns local data more deeply, significantly improving the performance of the global model. This indicates that the number of Local Epochs directly affects learning performance and convergence speed. Particularly, Local Epoch = 10 shows the fastest convergence and the highest final performance.

However, increasing the number of Local Epochs also raises computational costs on the client side. In resource-constrained environments, high numbers of Local Epochs may become inefficient, so a trade-off between accuracy and computational costs must be considered.

Local Epoch = 5 strikes a balance between accuracy and computational efficiency, making it a suitable choice even in resource-constrained environments. Local Epoch = 10 provides the best performance in environments with sufficient computational resources, while Local Epoch = 1 can be used in environments where minimizing computational costs is the priority.

5-3. Task 4 - Effect of changing client fraction test

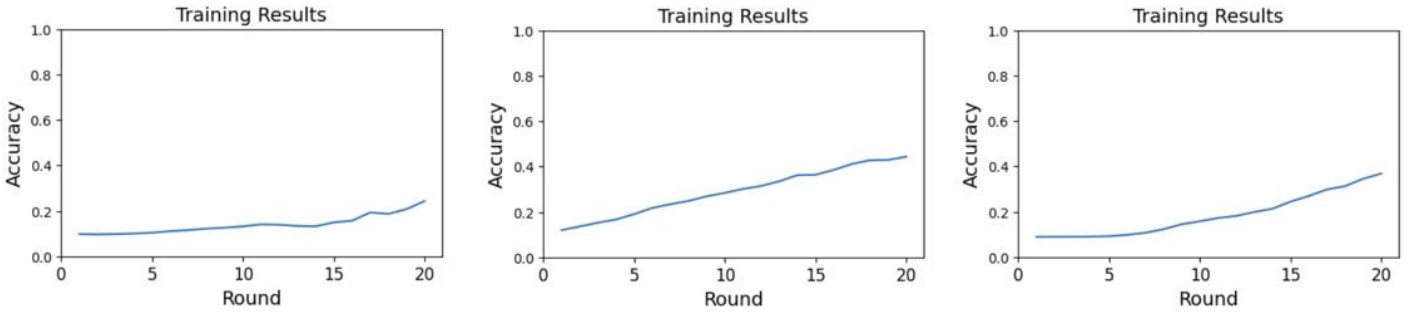

Fig 9. Task 4 – (Client Fraction = 0.01, 0.05, 01, Left to Right)

In all the graphs of Fig 9, accuracy consistently increases as the number of rounds progresses. However, when the Client Fraction is 0.01, the number of participating clients is very small, preventing the global model from sufficiently learning the entire dataset. As a result, the final accuracy remains low at approximately 20%. This demonstrates that a lack of data diversity can significantly limit model performance.

As the Client Fraction increases, the number of participating clients grows, allowing the model to learn from more diverse data. Consequently, with a Client Fraction of 0.05, the accuracy reaches approximately 40%, and with a Client Fraction of 0.1, similar performance is observed. However, these accuracy levels are still significantly lower compared to the results in Fig 7 and Fig 8. This suggests that the number of participating clients (Client Fraction) has less impact on improving accuracy compared to the number of rounds or local epochs.

While increasing the number of participating clients positively impacts accuracy, the effect is limited. Furthermore, as the number of clients increases, communication costs and training time also grow, and the overall model size is likely to expand. Considering these additional resource demands, and given the minimal difference in accuracy between Client Fraction = 0.05 and 0.1, Client Fraction = 0.05 appears to be the optimal choice in this case.

In conclusion, while Client Fraction affects learning performance, the number of rounds and local epochs play a more significant role in improving model accuracy. Balancing communication costs and training efficiency, Client Fraction = 0.05 provides a practical choice, offering a good trade-off between performance and resource usage.

6. Conclusion

In this report, we explored the implementation of FedAvg, the most widely used algorithm in Federated Learning. In Chapter 2, we reviewed the requirements necessary to implement the program, and in Chapter 3, we examined the relevant concepts. Building on this foundation, we explained the code for the implemented program in Chapter 4 and visualized and analyzed the results based on the graphs generated in Chapter 5.

Through three experiments, it became evident that accuracy improved consistently across all experiments as the number of rounds increased. Additionally, we observed that parameter selection requires careful consideration. Increasing the parameter values indiscriminately, without accounting for communication and computational costs, proved to be suboptimal. Instead, moderate adjustments to the parameter values produced the most optimal results across all three experiments.

By taking the Big Data Processing course, I had the opportunity to theoretically study various data processing methods and practice these concepts through coding. This included learning about traditional parallel data processing using C/C++ as well as machine learning-based approaches for data handling. These experiences provided valuable insights and effective strategies for addressing the exponential growth of data in modern society.

The transition from traditional serial code to parallel processing significantly improved program performance. Furthermore, adopting machine learning-based data processing aligned with current trends and equipped me with practical and relevant technical skills. The combination of theoretical knowledge and hands-on experience through this course has fostered a differentiated skill set and deeper insights, which I believe will be instrumental in the future.

Lastly, I would like to express my gratitude to Professor Jae-Yeon Park for the dedication and effort invested in every lecture. The lessons taught throughout the course have been incredibly valuable, and I deeply appreciate the knowledge and guidance shared. Thank you for all your hard work this semester.