University of Zurich[UZH]

Master Thesis – Communication Systems Group, Prof. Dr. Burkhard Stiller

FedEP: Tailoring Attention to Heterogeneous Data Distribution with Entropy Pooling

Hongjie Guan
Zurich, Switzerland
Student ID: 20-743-696

Supervisor: Chao Feng
Date of Submission: July 03, 2024

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Master Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Abstract

Federated Learning (FL) is a distributed machine learning paradigm in which multiple clients collaboratively train a global model without sharing their private data. The performance of FL is highly influenced by data heterogeneity, such as the varied statistical distribution of training data across clients. To address this issue, numerous algorithms have been introduced. In this paper, we propose a novel FL algorithm, Federated Entropy Pooling (FedEP). This method mitigates the client-drift problem resulting from data heterogeneity while preserving clients' privacy by incorporating the statistical characteristics of local distributions instead of any actual data. Prior to training, each client conducts a local distribution fitting using a Gaussian Mixture Model (GMM) and communicates the resulting statistical characteristics to a central aggregator in Centralized Federated Learning (CFL) or throughout the network in Decentralized Federated Learning (DFL). The aggregator then uses these statistical characteristics to compute Kullback-Leibler (KL) divergences between the data distributions of clients and the estimated global distribution to construct a new, optimized aggregation function. FedEP can be considered a re-parameterization of FedAvg, incorporating the distribution differences across clients.

Our experiments demonstrate that FedEP can achieve a faster convergence rate and higher accuracy compared to Federated Averaging (FedAvg) and other popular algorithms.

Zusammenfassung

Federated Learning (FL) ist ein verteiltes maschinelles Lernparadigma, bei dem mehrere Clients gemeinsam ein globales Modell trainieren, ohne ihre privaten Daten zu teilen. Die Leistung von FL wird stark durch Datenheterogenität beeinflusst, wie die unterschiedliche statistische Verteilung der Trainingsdaten über die Clients hinweg. Um dieses Problem zu adressieren, wurden zahlreiche Algorithmen eingeführt. In dieser Arbeit schlagen wir einen neuartigen FL-Algorithmus vor, Federated Entropy Pooling (FedEP). Diese Methode mindert das Client-Drift-Problem, das durch Datenheterogenität entsteht, und bewahrt gleichzeitig die Privatsphäre der Clients, indem die statistischen Eigenschaften lokaler Verteilungen anstelle der tatsächlichen Daten einbezogen werden. Vor dem Training führt jeder Client eine lokale Verteilungsanpassung mittels eines Gaussian Mixture Model (GMM) durch und übermittelt die resultierenden statistischen Eigenschaften an einen zentralen Aggregator im Centralized Federated Learning (CFL) oder im gesamten Netzwerk im Decentralized Federated Learning (DFL). Der Aggregator verwendet diese statistischen Eigenschaften, um Kullback-Leibler (KL)-Divergenzen zwischen den Datenverteilungen der Clients und der geschätzten globalen Verteilung zu berechnen und so eine neue, optimierte Aggregationsfunktion zu konstruieren. FedEP kann als eine Re-Parametrisierung von FedAvg betrachtet werden, die die Verteilungsunterschiede zwischen den Clients einbezieht. Unsere Experimente zeigen, dass FedEP eine schnellere Konvergenzrate und höhere Genauigkeit im Vergleich zu Federated Averaging (FedAvg) und anderen populären Algorithmen erreichen kann.

Acknowledgments

I would like to express my special gratitude to Chao Feng, Ph.D. student at the Communication Systems Group at the University of Zurich, for his supervision of my master's thesis. Chao provided me with an excellent introduction to federated learning, offering a comprehensive overview and valuable insights into the field. Throughout the thesis, he consistently offered prompt and crucial guidance. In our regular meetings, he provided valuable insights and critical feedback, effectively guiding me and ensuring I stayed on the right path. His steadfast support and guidance have been essential to my accomplishment. I would also like to extend my thanks to Dr. Alberto Huertas for his high-level and insightful instructions.

Additionally, I extend my thanks to Prof. Dr. Stiller for the opportunity to undertake my Master's thesis at the Communication Systems Group.

Contents

Abstract i

Acknowledgments v

1 Introduction 1

1.1 Motivation 2

1.2 Description of Work 2

1.3 Thesis Outline 2

2 Background 5

2.1 Overview on Federated Learning 5

2.1.1 The Foundational Algorithm 5

2.2 Categorization 7

2.2.1 Centralized FL, Semi-Decentralized FL and Decentralized FL . . . 7

2.2.2 Horizontal FL (HFL) and Vertical FL (VFL) 7

2.3 Open Challenges 8

2.3.1 Heterogeneities 9

2.3.2 Security 10

3 Related Work 13

3.1 Improving Global Model 14

3.1.1 Data-based Methods 14

3.1.2 Parameter-based Methods 16

3.1.3 Algorithm-based Methods 16
3.2 Personalized Federated Learning 17
3.2.1 Local Fine-tuning 17
3.2.2 Meta Learning 18
3.2.3 Multi-task Learning 18

4 FedEP Algorithm **19**
4.1 Problem Formulation 19
4.2 Algorithm Description 19
4.2.1 Pre-Train Distribution Fitting 20
4.2.2 Local Distribution Estimation and Entropy Pooling 24
4.3 Convergence Analysis 24
4.3.1 Assumptions 25
4.3.2 Full Device Participation 25

5 Implementation **27**
5.1 Adaptions to Fedstellar 27
5.1.1 Algorithm Implementation 27
5.1.2 Communication Implementation 32

6 Evaluation **35**
6.1 Experiment Setup 35
6.1.1 Pure Non-IID Scenario 36
6.1.2 Mixed Non-IID Scenario 36
6.2 Results on one dataset 37
6.3 Discussion 38

7 Summary and Conclusions **49**

Abbreviations **51**

List of Figures **51**

List of Tables **54**

List of Algorithms **55**

List of Listings **57**

A Performance Comparison on MNIST **61**

B Performance Comparison on FashionMNIST **71**

Chapter 1

Introduction

Conventional machine learning tasks typically involves collecting data on a centralized server. However, this approach faces challenges in real-world applications due to growing concerns about data privacy, confidentiality, and limitations in computing resources. Federated Learning (FL) emerges as a distributed machine learning paradigm that addresses these concerns by allowing data to remain on its owner's devices, and enabling collaborative machine learning tasks without direct data sharing.

Compared to distributed machine learning in a data center environment, FL faces three distinct challenges: (1)Computational Heterogeneity: The various computational capabilities of different devices lead to discrepancies in training times; (2)Communication Heterogeneity: Devices participate intermittently in FL. Some may be offline or powered off, potentially losing connection during training; and (3)Data Heterogeneity: FL uses datasets collected directly from edge devices. In this case, datasets are often heterogeneous or non-independent and non-identically distributed (Non-IID), as opposed to the independent and identically distributed (IID) datasets in data center environment. These challenges affects the consistency, reliability and performance of FL.

To accommodate machine learning in federated settings, FedAvg[1], the commonly used optimization method in the federated setting, brings training to local device and reduces communication frequency. FedAvg first performs E epochs of local stochastic gradient descent (SGD) on a fraction of client devices in the network each round. The devices then communicate their model parameters to a central server, where they are averaged with a weight with respect to the data amount of this clients. FedAvg has demonstrated empirical effect in mitigating computation and communication heterogeneities, it does not fully address the data heterogeneity. Since training was performed in local client nodes with non-IID data distribution, FedAvg is likely to meet a Client Drift(CD) problem, that the global model is drifted away from its global optimal by other non-IID client. Although there are works prove that FedAvg can converge in Non-IID data setting[]. In practice, FedAvg, converges slower in Non-IID data setting, or in worst cases, cannot converge to the optimal.

1.1 Motivation

The root representative of data heterogeneity, is the various data distributions. The previous related works either share data, compromising privacy, or add penalties or re-parameterize the loss function in training based on the weights difference. In this context, this work aims to quantify the Non-IID data distributions and incorporates this quantity into the FL algorithm.

1.2 Description of Work

In this work, we introduce a novel Federated Entropy Pooling Algorithm (FedEP), designed to alleviate the client drift issue brought by data heterogeneity in extreme Non-IID scenarios. FedEP considers the distribution of local data and customizes the attention allocated to each model. Contrasting with FedAvg, which performs weighted average aggregation based on the volume of local data, FedEP employs a pooling of the Kullback-Leibler (KL) divergence[2]. The optimized aggregation function expends the weights of the models from clients with more unique datasets, while reducing the weights of models from clients with less distinctive data, thereby enhancing the global model training process. Conceptually, FedEP can be viewed as a re-parameterization of the widely used FedAvg algorithm, but with a unique focus on incorporating distribution differences across clients.

The proposed FedEP algorithm was evaluated on the MNIST, FashionMNIST, and CIFAR-10 datasets in decentralized FL networks under two different Non-IID settings: Pure Non-IID and Mixed Non-IID. The performances were compared to those of FedAvg and SCAFFOLD[3].

Our results have shown that under mixed Non-IID scenarios, such as networks comprising nodes with varying levels of data heterogeneity, FedEP effectively enhances overall performance in classification tasks. Specifically, in a network with some nodes exhibiting lower heterogeneity and others with higher heterogeneity, FedEP not only increases the overall accuracy but also significantly aids nodes with heterogeneous data samples in learning more effectively compared to FedAvg and SCAFFOLD.

The entropy pooling technique employed by FedEP shows its ability to handle mixed Non-IID scenarios with extreme outliers, suggesting its potential application in future FL environments characterized by high data variability. This robustness makes FedEP a valuable tool for improving the reliability and performance of FL systems in real-world applications, where data heterogeneity is a common challenge.

1.3 Thesis Outline

The structure of this work is outlined as follows. First, Chapter 2 establishes the theoretical background and describes the fundamental concepts used in this work. This includes

an overview of FL, and a specific focus on Data Heterogeneity.

Chapter 3 delves into Related Work, presenting a comprehensive review of existing methodologies and approaches in the field. This chapter is structured to first introduce the problem formation. It then explores common methods to address data heterogeneity in different categories, highlighting significant contributions like FedProx, FedNova and SCAFFOLD.

Chapter 4 introduces the FedEP Algorithm, the core contribution of this thesis. This chapter not only details the algorithm but also includes an Convergence Analysis to demonstrate its theoretical robustness and efficiency.

In Chapter 5, the focus shifts to Implementation, specifically discussing the adaptations made to integrate FedEP with the existing Fedstellar framework. This chapter provides insights into the practical aspects of applying the FedEP algorithm in real-world scenarios.

Chapter 6 is dedicated to Evaluation, where the experimental setup, results, and a comprehensive discussion are presented. This chapter aims to empirically validate the effectiveness of the FedEP algorithm through various experiments and comparative analyses.

Finally, Chapter 7 provides a Summary and Conclusions, drawing together the main findings of the research and discussing the implications of the results. This chapter also suggests potential avenues for future research, building on the work presented in this thesis.

Chapter 2

Background

This sections will summarize the theoretical foundation of FL and elucidate the taxonomy of common heterogeneity challenges in FL.

2.1 Overview on Federated Learning

Federated Learning represents a distributed machine learning paradigm in which numerous clients(also known as nodes, participants, or entities) work collaboratively to train a unified global model, while ensuring their private data remains unshared. Consider a federation with a set of clients, K, and $|K|$ as the total number of clients, for client $k \in K$, the local dataset $D_k = (\mathbf{X_k}, \mathbf{y_k}) \in (\mathbf{X}, \mathbf{Y})$. The feature matrix $\mathbf{X_k} = \{\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_{N_k}}\}$ where each of the element $\mathbf{x_n}, n \in [1, 2, ..., N]$ is a feature vector and N_k is the number of data samples in client k. $\mathbf{y_k}$ is a label vector $\{y_1, y_2, ..., y_{N_k}\}$. We have $\mathbf{X} = \bigcup_{k=1}^{K} \mathbf{X_k}$ and $\mathbf{Y} = \bigcup_{k=1}^{K} \mathbf{y_k}$ as the theoretical total union of global datasets that are not stored on a single machine. Especially in a classification case, we define $\mathcal{Y}$ as distinct label classes in the global context and $\mathcal{Y}_k$ as the distinct label classes in client k.

2.1.1 The Foundational Algorithm

The foundational algorithm in FL, FedAvg, was first proposed by McMahan[1] in 2017. This algorithm was initially developed for use in centralized federated learning(CFL) settings that consists of a centralized server responsible for aggregation and $|K|$ clients undertaking model training with their respective local dataset.

Problem Formulation

The global optimization model of FedAvg is as follows:

$$\min_{w} \{F(w) = \sum_{k=1}^{|K|} p_k F_k(w)\}$$

where $p_k = \frac{N_k}{\sum_{k=1}^{|K|} N_k}$ is the weight of dataset of client k such that $p_k \leq 0$ and $\sum_{k=1}^{|K|} p_k = 1$. F_k is the local optimization function of client k.

Algorithm Description

FedAvg begins with the server initializing a weight w_0. The training process then proceeds in rounds, each involving three steps:

Step 1: The Server randomly selects a subset of clients, $S_t \subseteq K$,and broadcasts the current weight w_t to S_t;

Step 2: Upon receiving the weight w_t from the server, each client $k \in S_t$ performs a Client Update function in parallel. This involves splitting the local dataset D_k into batches of size of B. For E epochs, each batch of data $\xi_{i,j}$ is used to update w_t via stochastic gradient descent with learning rate η:

$$w_{t+i+1}^k = w_{t+i}^k - \eta \bigtriangledown F_k(w_{t+i}^k; \xi_{i,j}^k), \quad i = 0, 1, ..., E-1, \quad j = 1, 2, ..., \lceil N_k/B \rceil$$

Step 3: The clients send their updated weights back to the server, which then aggregates these weights into w_{t+1} using a weighted average with weights $p_k = \frac{N_k}{\sum_{k=1}^{|S_t|} N_k}$:

$$w_{t+1} = \sum_{k=1}^{|S_t|} p_k w_{t+1}^k$$

The algorithm iterates through these steps until convergence is achieved or a predefined number of rounds is completed. Through this procedure, the central server obtains a model, w_t^*, that has been trained on data from the clients without direct access to the data itself.

Compared with the Distributed SGD[4–6], in which, the gradients instead of the model weights are communicated and aggregated, FedAvg let the clients to do their own model updating, and more importantly, updating for E epochs. This greatly reduced the dependency on communication, making collaborate training among large numbers of clients possible. However, this approach also presents challenges, such as security concerns. Previous studies have shown that FedAvg is highly vulnerable to poisoning attacks[7, 8] since this protocol does not account for adversarial participation[9, 10]. Moreover, due to the heterogeneous nature of clients' local datasets, local training exacerbates the problem of client drift.

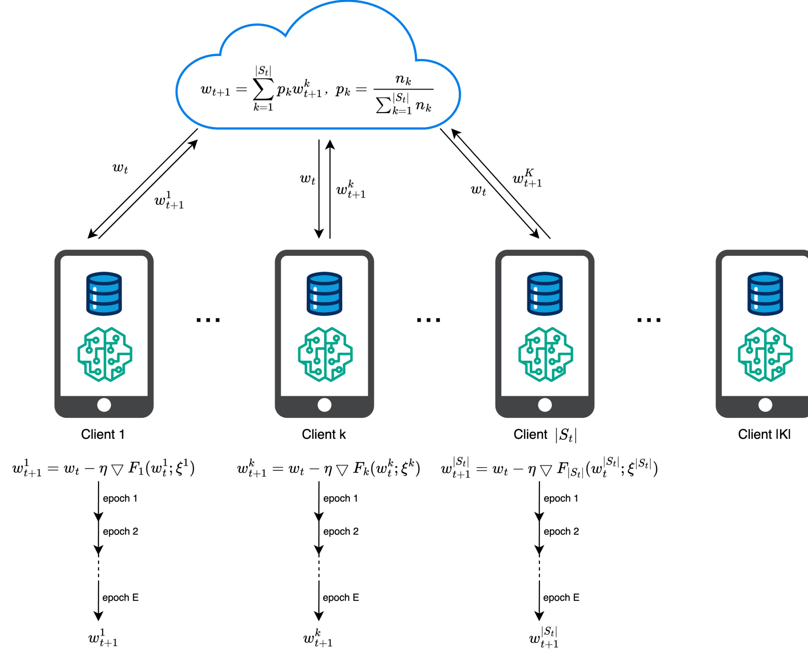

Figure 2.1: Federated Average(FedAvg) ($S_t \neq K$).

2.2 Categorization

2.2.1 Centralized FL, Semi-Decentralized FL and Decentralized FL

According to the variations of FL architectures, FL can be divided into Centralized Federated Learning (CFL), Semi-Decentralized Federated Learning (Semi-DFL), and Decentralized Federated Learning (DFL). In CFL, also referred to as the client-server model, the aggregator role is fixed to the server. This aggregator collects models sent by the clients, aggregates them into a global model, and then broadcasts it to other clients. In Semi-DFL, the role of the aggregator rotates within the network. In DFL, each node in the network acts both as a client to train the model and broadcast their model into the network, and as an aggregator to collect models from others to obtain a better global model.

2.2.2 Horizontal FL (HFL) and Vertical FL (VFL)

Based on differences in data distributions across feature spaces $\mathcal{X}$ and sample spaces, FL can be categorized into Horizontal Federated Learning (HFL), Vertical Federated Learning (VFL), and Federated Transfer Learning (FTL).

HFL refers to cases where multiple clients share a same or similar feature spaces, but the data samples are from distinct users. VFL, on the other hand, applies to cases where the feature spaces do not overlap, but the data samples likely do. In application scenarios,

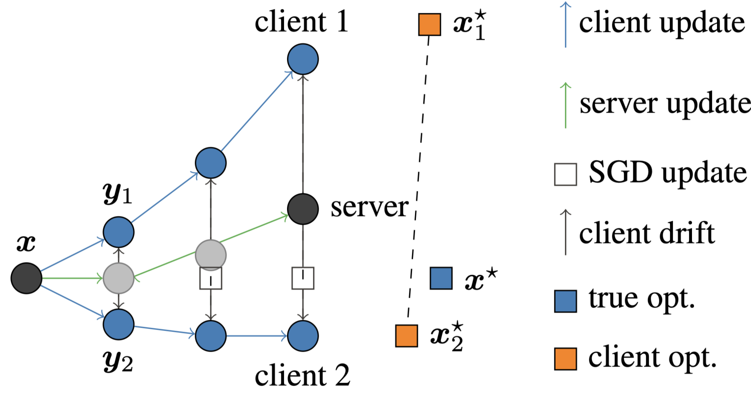

Figure 2.2: Simplified client drift of 2 client nodes in FedAvg[[3]].

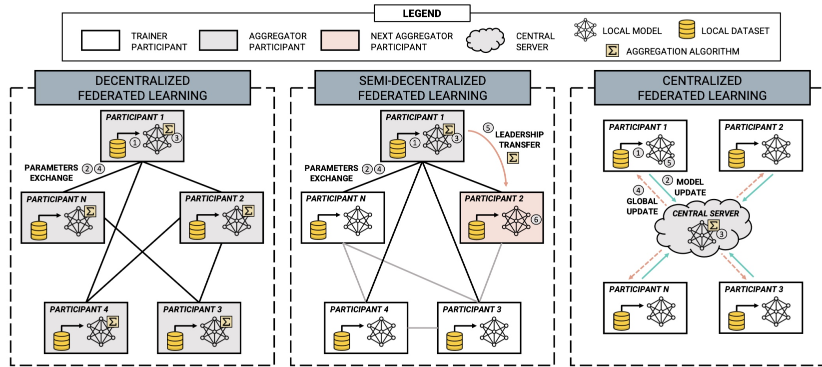

Figure 2.3: Variations of FL Architectures: CFL, Semi-DFL, and DFL [11]

HFL refers to situations where each client collects similar features from distinct users. For example, hospitals in different regions collect the a dataset with the same features from their respective(non-overlapping) local patients. Conversely, VFL refers to situations where each client collects different features from the same or similar group of users. For example, the local government, a bank and a hospital collecting data from the same district are likely to have data on the same group of residents but with different features. Cases where clients have neither common features nor common data samples are usually addressed as Federated Transfer Learning (FTL).

In our work, we focus on the Horizonal scenario for single task and identical features.

2.3 Open Challenges

In FL, there are two main challenges that limit its full potential: heterogeneities and security. Many studies have focused on these two areas[].

2.3.1 Heterogeneities

Heterogeneities in FL can be grouped into data heterogeneity (also called statistical heterogeneity) communication heterogeneity, and computation heterogeneity.

Data Heterogeneity

Data heterogeneity(Statistical Heterogeneity) refers to the differences in data distributions across clients. Traditional machine learning algorithms assumes that data is IID (independently and identically distributed). However, this is not the case in FL where each device collects and stores data independently, leading to non-IID (not independently and not identically distributed) data.

Consider a federation with a set of clients, K, and $|K|$ as the total number of clients, for client $k \in K$, the local dataset $D_k = (\mathbf{X}_k, \mathbf{y}_k) \in (\mathbf{X}, \mathbf{Y})$ and the local data distribution is $\mathcal{P}_k(D_k)$ or $\mathcal{P}_k(\mathbf{X}_k, \mathbf{y}_k)$. In a classification case, we define $\mathcal{Y}$ as distinct label classes in the global context and $\mathcal{Y}_k$ as the distinct label classes in client k.

In a data IID case in FL:

$$P_{k_1}(y = c) \asymp P_{k_2}(y = c), \forall k_1, k_2 \in K, k_1 \neq k_2, \forall c \in \mathcal{Y}$$

For each class c in a $\mathcal{Y}$, the probabilities of y being in class c in two different clients k_1,k_2 are asymptotically equal.

The IID assumption is important for traditional machine learning algorithms to achieve generalization but it is not applicable in FL. Addressing this challenge necessitates the development of novel techniques and models capable of quantifying the level of non-iidness within the data and effectively integrating this insight into the training process.

Communication Heterogeneity and Computation Heterogeneity

Communication heterogeneity comes from the different network conditions of the devices involved. In FL, devices often have varying network speeds and reliability, which can cause delays and inefficiencies. Some devices may be much slower or frequently disconnected. This makes it challenging to design communication methods that can handle these differences and ensure efficient model updates.

Computation heterogeneity is about the varying computational power of the devices. Devices in FL range from powerful servers to less capable mobile phones or IoT devices. This difference means that some devices can process and send data much faster than others, leading to imbalanced training processes. Managing these differences effectively requires algorithms that can adjust the workload based on each device's capabilities.

2.3.2 Security

Security is a major concern in FL due to its decentralized nature. One key security challenge is ensuring the robustness of algorithms against Byzantine failures.

Byzantine-Robust Algorithms Byzantine failures refer to situations where some devices behave incorrectly or maliciously during the FL process. These devices can send incorrect, corrupted, or even harmful updates to the central server, damaging the global model. There are several types of attacks that malicious clients might use:

Poisoning Attacks These involve adding fake data to negatively affect the training process. This can slow down the training rate or, in the worst case, prevent the model from converging altogether[7, 8, 13].

Inference Attacks These attacks aim to compromise data privacy by inferring the datasets of other participants in the training process.[14]

Backdoor Attacks In these attacks, malicious clients insert hidden triggers during training, allowing them to control the model's behavior when specific conditions are met.[15, 16]

Developing Byzantine-robust algorithms is essential to address these issues. These algorithms need to detect and mitigate the impact of malicious or faulty updates, ensuring that the training process remains reliable and secure. Methods such as robust aggregation, anomaly detection, and secure multi-party computation are being explored to tackle these challenges.

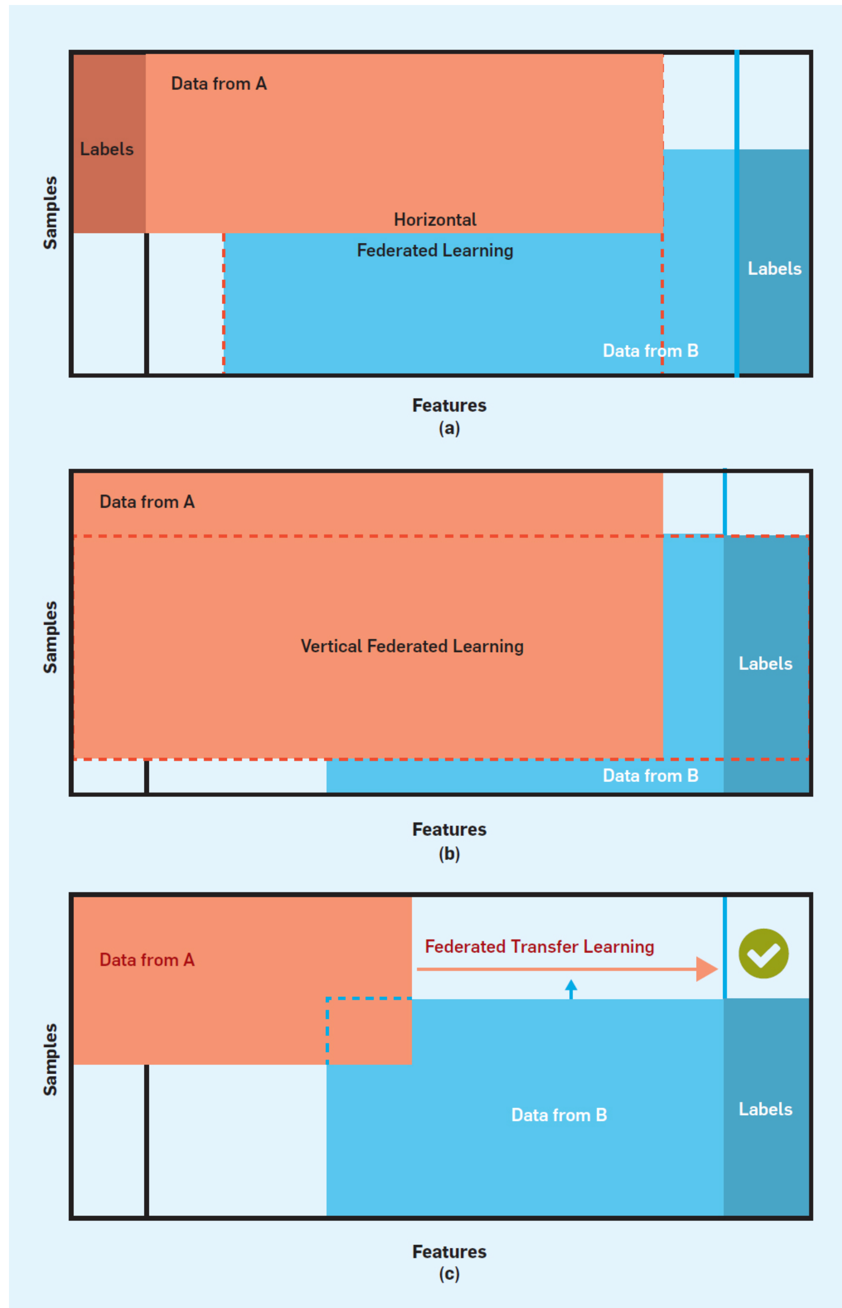

Figure 2.4: Horizontal FL, Vertical FL, and Federated Transfer Learning [12]

Chapter 3

Related Work

Recent literature has extensively explored various approaches to mitigate the influence of Data Heterogeneity in FL, offering numerous categorization methodologies[17]. A critical question to consider before categorization is the problem formulation or the underlying objective. Based on this problem formulation, these methods can be classified into two main categories within a non-i.i.d. setting: enhancing the generalization of the global model or improving the personalization of client models.

To enhance the generalization of the global model, the problem formulation is as follows:

$$\min_{w} \left\{ F(w) = \frac{1}{k} \sum_{k=1}^{|K|} F_k(w) \right\} \tag{3.1}$$

Under problem formulation (3.1), the primary goal is to optimize the global algorithm to ensure its generalization across diverse clients' test sets.

Differently, to improve the personalization of client models, the problem formulation is as follows:

$$\min_{w} \left\{ F(w) = \frac{1}{k} \sum_{k=1}^{|K|} F_k(w - \eta \bigtriangledown F_k(w)) \right\} \tag{3.2}$$

Under problem formulation (3.2), the goal is to prepare a global algorithm such that clients can use this global algorithm as an initial point to fine-tune their respective personalized algorithms. The ultimate focus is on the performance of these personalized algorithms on their distinctive tasks. This scenario is frequently addressed in the context of a multi-task learning setting.

Distinguishing between these problem formulations is crucial, as these two objectives can be contradictory in some methodologies, where a trade-off between the two objectives

might be inevitable. In Federated Meta Learning, an increase in personalization might come at the expense of reduced generalization[18].

In our study, we will focus on a single-task learning setting and problem formulation (3.1) to improve the global model.

3.1 Improving Global Model

In the context of problem formulation (3.1), methods for improving the global model can be categorized into three perspectives: data-based, parameter-based, and algorithm-based. It should be noted that these categories are not mutually exclusive; an algorithm may simultaneously employ methods from several categories.

Category	**Sub-Technique**	**Definition**	**Examples**	**Pros**	**Limitation**
Data-based	data sharing	communicate data	FedDF	Technique simple to employ, does not change the learning model	Consume resource and compromise privacy
	data augmentation	generate fake data	FEDGEN		
	data selection	randomize data selection			
Parameter-based	Adding/adjusting parameters in training	Variation of FedAvg by adding or adjusting parameters	FedProx, FedMA, SCAFFOLD, FedAdagrad, FedYogi, FedAdam	Elegantly solve heterogeneity problem without compromising privacy	Higher computation complexity
	Adding/adjusting parameters in Aggregation				
Algorithm-based	Federated Distillation.	Add new technique to enhance simulation between models	FedDF, FEDGEN	Lift the limitation of model structure	Compromise data accuracy and communication delay
	Federated Personalization Learning	Do further personalization of global model		One more step of personalization	Different problem formation

Table 3.1: Existing Methods in mitigating data heterogeneity

3.1.1 Data-based Methods

Data-based methods primarily focus on enhancing data homogeneity. These methods can be further categorized into data sharing, data augmentation, and data selection.

Data Sharing

Data sharing is a common method for enhancing data homogeneity. Data sharing might seem contradictory to the privacy-preserving nature of FL, but this is not necessarily the case. A fundamental challenge in data sharing is the construction of shared datasets using generated data instead of real private data.

Zhao[19] propose a data-sharing strategy to improve FedAvg with non-IID data by creating a small subset of data, $\mathcal{D}_{\mathcal{G}}$, which is globally shared by all the edge devices. The

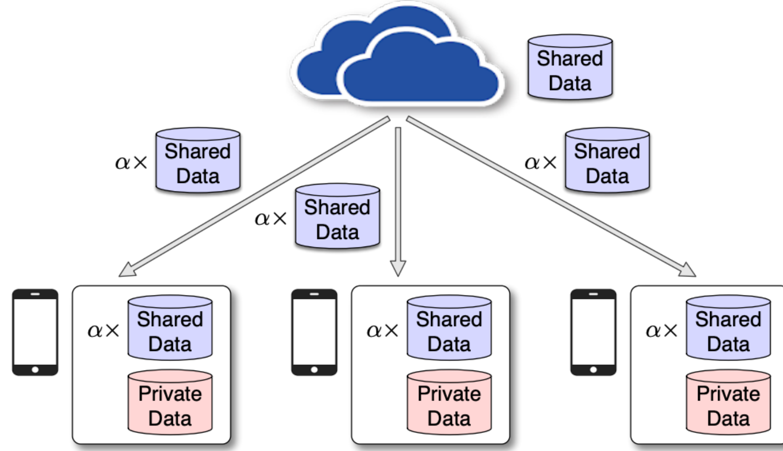

Figure 3.1: The data-sharing Strategy[19]

initial weights, w_0, instead of random initialized, is first trained on $\mathcal{D}_{\mathcal{G}}$ as a warm-up model and later each edge devices is given a random portion of $\mathcal{D}_{\mathcal{G}}$ to aggregate with the local private data. This method attempts to make less compromise to privacy leak by sharing a randomized small portion of private data. However the sharing of real private data is still not ideal in FL. Lin [20] utilized a unlabeled dataset that was shared by clients or by generation

Data Augmentation

Data augmentation methods utilize generative models to locally generate additional data samples at client devices, thereby addressing non-IID challenges while considering privacy and communication overhead constraints. Jeong[21] proposes the FAug algorithm, which enables devices to acquire missing data sample labels (referred to as target labels) by leveraging a small number of data samples uploaded by other devices to balance non-IID data distributions. In the FAug algorithm, the server processes the uploaded small data samples, training them to form a GAN (Generative Adversarial Network) generator capable of supplementing target labels for each device. Subsequently, the trained GAN model is distributed to each client, allowing participants to expand their data by utilizing their own datasets.

Instead of utilizing generative methods like GAN model, Shin[22] proposed a XorMixFL framework that introduces a privacy-preserving XOR-based mixup data augmentation technique called XorMixup. This method aims to mitigate non-IID data challenges by collecting encoded data samples from other devices and decoding them exclusively using each device's own data. The decoding will distort data from other clients to be additional data. This process gradually transforms the dataset into an IID format suitable for model training while preserving data privacy through deliberate distortion using XOR operations.

Data Selection

Data selection strategies in FL involve methods for identifying and utilizing random selected data subsets for model training, minimizing computational complexity, and optimizing model performance across heterogeneous datasets. Wang [23] intelligently chooses the client devices to participate in each round to counterbalance the bias introduced by non-IID data and to speed up convergence is proposed.

3.1.2 Parameter-based Methods

Parameterization methods maintain the traditional FedAvg paradigm and enhance it by adjusting existing parameters or adding new parameters during training or aggregation.

Li [24] added a penalty term to the local objective function $F_k(w)$. Each client k minimizes the following objective $h(k)$:

$$\min_{w} h_k(w; wt) = F_k(w) + \frac{\mu}{2}\|w - w^t\|^2$$

The penalty term added bounds the new weights closer to the global weights to avoid several client drift. Karimireddy [3] proposed SCAFFOLD that add a control variate term c in training and aggregation. During the weights update at training at the local client:

$$w_t = w - \eta(g(w) - c_k + c)$$

where c_k is the client input and c is first initialized and broadcast with the model w and later a global aggregation of c_k. The use of term c,

3.1.3 Algorithm-based Methods

Algorithm-based methods employ approaches that differ from the traditional FedAvg paradigm.

Federated Distillation

Knowledge Distillation (KD), also known as co-distillation(CD) is a learning paradigm of transferring knowledge from a teacher model to a student model[25–27]. In knowledge distillation, the student model updates its weights, not by the discrepancy(usually measured by cross-entropy) between model prediction and label, but by the discrepancy(usually measured by relative entropy, or referred as KL divergence) between the logits of student model and that of a teacher model, where the logits is the last layer output of the neural network before the activation function.

Federated Distillation[21], the application of Knowledge Distillation in FL, has been a heated direction due to several main advantages of distillation. First, unlike classical FedAvg or its adaptation algorithms that aggregates model by weights hence requires all

models has the same structures which conflicts with the computation capacity heterogeneity in real world, Federated Distillation, on the other hand, only ensembles the logits hence allows heterogeneous model structure as long as with a same output dimension. Secondly, exchanging only the logits, which size is much smaller than the weights, greatly reduced communication burdens.

Lin [20] proposes FedDF, an ensemble distillation for model aggregation through unlabeled data or data generated by GAN[28]. Itahara also[29] proposes a Distillation-Based Semi-Supervised Federated Learning(DS-FL) algorithm that update parameters with knowledge distillation from a half private labeled and half shared unlabeled datasets. Typical distillation methods that use a a proxy dataset $\mathcal{D}_{\mathcal{P}}$ to minimize the KL discrepancy, usually measured by Kullback-Leibler divergence, between the logits outputs of teacher model $\mathcal{T}$ and the student model $\mathcal{S}$. No matter this $\mathcal{D}_{\mathcal{P}}$ is real data with labels removed or generated by GAN. FedGen proposed by Zhu[30] can do data-free knowledge distillation without the need of the proxy dataset $\mathcal{D}_{\mathcal{P}}$ by learning a lightweight generator.

As a conclusion, federated distillation reduces communication burdens and lift the strict restriction of identical model structure by communicating the logits, instead of the whole models. While by knowledge distillation, the aggregator's model as a student learns from the aggregated logits from clients as teachers, such that the heterogeneities among various clients can be reduced.

3.2 Personalized Federated Learning

In heterogeneous scenario, the FL global model could perform poorly for some of the clients in the network. For some clients, model solely trained on their local data could perform better than the global model[18]. This situation is reasonable given the classical FedAvg and or its adaptation algorithms focused on minimizing the average loss across all clients. Hence making the global model easier to personalize become very important. Under this goal, the problem formulation becomes (3.2), and this formulation is also commonly referred to as Personalized Federated Learning(PFL).In other word, PFL methods do not study the data heterogeneity under the objective function(3.1), instead they consider this objective function problematic without the consideration of personalization. To solve data heterogeneity, they change the objective function to (3.2) . These methods typically include local fine-tuning[31, 32], meta learning[18, 33, 34],and multi-task learning[35]. These method's algorithm typically contains training of the global models, then use the global model as a initial model and fine-tune it for personalization.

3.2.1 Local Fine-tuning

Local fine-tuning in FL involves refining model parameters on individual devices using after training, following initial training with shared global parameters. This process aims to adapt the global model to local characteristics and enhance performance without compromising data privacy. Wu[36] proposed FedBiOT allowing for resource-efficient Large

language models (LLMs) fine-tuning with FL. Local fine-tuning helps mitigate issues arising from non-IID data distributions across devices, ensuring better convergence and model robustness.

3.2.2 Meta Learning

Meta Learning, or learning to learn, is emerging as a promising approach in FL to improve model adaptation across heterogeneous edge devices. Meta Learning algorithms aim to leverage meta-knowledge acquired from multiple tasks or domains to facilitate faster adaptation and better generalization on new tasks. In FL, Meta Learning techniques can help in initializing models that are more adaptable to diverse local datasets, thereby accelerating convergence and improving overall model performance.

3.2.3 Multi-task Learning

Multi-task Learning (MTL) in FL involves jointly training models to perform multiple related tasks simultaneously. This approach leverages shared representations across tasks to improve model efficiency and effectiveness, especially in scenarios where tasks share underlying patterns or dependencies. Lei [37] proposes a Group-based Federated Meta-Learning framework, called G-FML, which adaptively divides clients into groups based on the similarity of their data distribution. Personalized models are then obtained using meta-learning within each group. MTL in FL aims to enhance data efficiency, reduce computational overhead, and improve privacy preservation by leveraging synergies across multiple tasks while respecting local data constraints.

In summary, Compared with other previous methods thaqt aims to improve the generalization ability of the model, Personalized Federated Learning focus on achieving a higher performance of algorithm on the individual test set instead of a global one. Methods like local fine-tunning, meta learning, multi-task learning does not modify the training itself, however, though operation afterward of the training , these techniques improve the learning outcome.

In summary, Personalized FL diverges from traditional approaches aimed at enhancing model generalization by prioritizing heightened algorithmic performance on individual test sets rather than a global one. Techniques such as local fine-tuning, meta-learning, and multi-task learning do not fundamentally alter the initial training process. Instead, they operate post-training to refine model parameters and adapt them to local data characteristics.

Chapter 4

FedEP Algorithm

4.1 Problem Formulation

The global optimization model of Federated Entropy Pooling(FedEP) is as follows:

$$\min_{w} \{F(w) = \sum_{k=1}^{|K|} \alpha_k F_k(w)\}$$

where F_k is the local optimization function of client k as they are in FedAvg. The attention coefficient, α_k, is the pooled KL-divergence that measures the relative uniqueness of local data D_k in client k to the global datasets. We have $\alpha_k \leq 0$ and $\sum_{k=1}^{|K|} \alpha_k = 1$.

4.2 Algorithm Description

FedEP is designed to be applied to CFL, Semi-DFL, and DFL frameworks. In the following description, the term 'Aggregator' represents a centralized server in the case of CFL, or every node in Semi-DFL and DFL. FedEP supports both full device participation and partial device participation, and for simplicity, it can initially be assumed to have full device participation. FedEP is divided into three phases:

Phase 1 (Pre-train Distribution Fitting): Each clients fits a Gaussian Mixture Model (GMM) with an Expectation-Maximization (EM) method on its own local distribution P_k and then sends the resulting GMM parameters $\boldsymbol{\theta}_k$ and its own number of data samples N_k to the Aggregator(s).

Phase 2 (Global Distribution Estimation and Entropy Pooling): Upon receiving parameters, the Aggregator(s) estimate(s) a global distribution and obtain(s) the corresponding pooled KL-divergence α_k for each client k.

Phase 3 (Training): This phase follows the same procedure as FedAvg, with a modification in Step 3: The clients send their updated weights back to the Aggregator(s), which then

aggregate(s) these weights into w_{t+1} using a weighted average with weights being the pooled KL-divergence α_k:

$$w_{t+1} = \sum_{k=1}^{|S_t|} \alpha_k w_{t+1}^k$$

Algorithm 1 *Federated Entropy Pooling (FedEP).* All K nodes are indexed by k; E is the number of local epochs; η is the learning rate; M is the number of models in GMM; T is the training rounds.

1: **function** AggregatorExecutes(b)
2: Initialize an empty array θ_k [ ]
3: Initialize $i \leftarrow 0$
4: **for** each client $k \in K$ in parallel **do**
5: $\hat{\boldsymbol{\theta}}_k \leftarrow$ Pre-trainDistributionFitting($\boldsymbol{y_k}, \rho = 0.5$)
6: $\theta_k[i] \leftarrow \hat{\boldsymbol{\theta}}_k$
7: $i \leftarrow i + 1$
8: **end for**
9: α_k [ ] $\leftarrow$ EntropyPooling(θ_k [ ])
10: Initialize w_0
11: **for** each round $t = 1, 2, \dots$ **do**
12: $m \leftarrow \max(C \cdot K, 1)$
13: $S_t \leftarrow$ (random set of m clients)
14: **for** each client $k \in S_t$ in parallel **do**
15: $w_{k,t+1} \leftarrow$ ClientUpdate(k, w_t)
16: **end for**
17: $w_{t+1} \leftarrow \sum_{k \in S_t} \alpha_k w_{k,t+1}$
18: **end for**
19: **end function**

20: **function** ClientUpdate(k, w)
21: $B \leftarrow$ (split P_k into batches of size b)
22: **for** each local epoch i from 1 to E **do**
23: **for** each batch $b \in B$ **do**
24: $w \leftarrow w - \eta \nabla L(w; b)$
25: **end for**
26: **end for**
27: **return** w
28: **end function**

4.2.1 Pre-Train Distribution Fitting

A Gaussian Mixture Model(GMM) is a parametric probability density function represented as a weighted sum of Gaussian component densities. Consider this GMM :

$$P(\mathbf{y}_k|\boldsymbol{\theta}_k) = \sum_{m=1}^{M} \pi_m \mathcal{N}(\mu_m, \sigma_m^2), \quad \pi_m > 0, \quad \sum_{m=1}^{M} \pi_m = 1$$

Algorithm 2 *Pre-train Distribution Fitting Algorithm.*The hyper-parameter ρ, representing the maximum component fraction, is used to determine $M_{\max}$, the upper limit for the number of mixture models. Specifically, $M_{\max}$ is set as ρ times the total number of distinct label classes $\mathcal{Y}_k$, where $\rho \in (0, 1]$.

1: **function** PRE-TRAINDISTRIBUTIONFITTING($\boldsymbol{y_k}, \rho = 0.5$)
2: Initialize $M_{\max} = \lceil \rho \times |\mathcal{Y}_k| \rceil$
3: Initialize $\hat{M}_k$ and $\hat{\boldsymbol{\theta}}_k$
4: **for** M in $[1, M_{\max}]$ **do**
5: $L(\boldsymbol{\theta}_M), \boldsymbol{\theta}_M \leftarrow$ EXPECTATIONMAXIMIZATION($M, \boldsymbol{y_k}$)
6: Compute BIC for current model: $BIC_M = -2\ln(L(\boldsymbol{\theta}_M)) + |M| \times \ln(N)$
7: **if** BIC of current model is lower than previous models **then**
8: Update $\hat{M}_k = M$ and $\hat{\boldsymbol{\theta}}_k = \boldsymbol{\theta}_M$
9: **end if**
10: **return** $\hat{\boldsymbol{\theta}}_k$
11: **end for**
12: **end function**

Algorithm 3 *Expectation Maximization Algorithm.* M is the total number of Gaussian models used in the GMM.

1: **function** EXPECTATIONMAXIMIZATION($M, \boldsymbol{y_k}$)
2: Initialize $\boldsymbol{\theta}^0 = \{\pi^0, \mu^0, \sigma^{2(0)}\}$
3: E-step: with $\boldsymbol{\theta}^t$, get the latent variable $\hat{\gamma}_{j,m}$
4: and $Q(\boldsymbol{\theta}^t)$, s.t. $\sum_{n=1}^{M} \pi_m = 1$
5: M-step: $\boldsymbol{\theta}_{i+1} \leftarrow \arg\max \boldsymbol{\theta}(Q + \lambda(\sum_{n=1}^{M} \pi - 1))$
6: Repeat E, M until convergence
7: **return** $L(\boldsymbol{\theta}), \boldsymbol{\theta}$
8: **end function**

where $\mathbf{y}_k$ is the label vector $[y_1, y_2, \ldots, y_{N_k}]$ of client k. $P(\mathbf{y}_k|\boldsymbol{\theta})$ is the probabilistic distribution of the labels of client k given $\boldsymbol{\theta}$. $\boldsymbol{\theta}$ is a $3 \times M$ coefficients matrix $[\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\sigma}^2]$. The hyper-parameter M is the total number of used Gaussian Models in the GMM. The mixture coefficient vector $\boldsymbol{\pi} = [\pi_1, \pi_2, \ldots, \pi_M]$, with each element as the coefficient of the m-th Gaussian distribution. The vector $\boldsymbol{\mu} = [\mu_1, \mu_2, \ldots, \mu_M]$, with each element as the mean of the m-th Gaussian distribution. The vector $\boldsymbol{\sigma}^2 = [\sigma_1^2, \sigma_2^2, \ldots, \sigma_M^2]$, with each element as the variance of the m-th Gaussian distribution. Note that if some observations $\boldsymbol{y} \sim \mathcal{N}(\mu_m, \sigma_m^2)$, their probability density function (PDF) is as follows:

$$f(\boldsymbol{y}|\mu_m, \sigma_m^2) = \frac{1}{\sigma_m\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{y-\mu_m}{\sigma_m}\right)^2}$$

From a statistical perspective, we can model local data distribution P_k of client k by a GMM with the Expectation Maximization (EM) Algorithm.

Expectation Maximization (EM) Algorithm

To fit a dataset with N samples to a GMM with M models, we define a latent variable matrix $\boldsymbol{\gamma}$:

$$\boldsymbol{\gamma}_{n,m} = \begin{pmatrix} \gamma_{11} & \gamma_{12} & \cdots & \gamma_{1m} & \cdots & \gamma_{1M} \\ \gamma_{21} & \gamma_{22} & \cdots & \gamma_{2m} & \cdots & \gamma_{2M} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ \gamma_{n1} & \gamma_{n2} & \cdots & \gamma_{nm} & \cdots & \gamma_{nM} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ \gamma_{N1} & \gamma_{N2} & \cdots & \gamma_{Nm} & \cdots & \gamma_{NM} \end{pmatrix}_{N\times M}$$

with a height of N and a width of M. Each elements $\gamma_{n,m}$ represents whether the n-th data sample belongs to the m-th Gaussian Model:

$$\gamma_{n,m} := \begin{cases} 1 & \text{if } y_n \sim \mathcal{N}(\mu_m, \sigma_m^2), \\ 0 & \text{else.} \end{cases} \qquad \sum_{m=1}^{M} \gamma_{nm} = 1, \quad m \in [1, M], \quad n \in [1, N],$$

Step 1 (Initialization): Initialize the coefficients matrix $\boldsymbol{\theta}^{(0)} = [\boldsymbol{\pi}^{(0)}, \boldsymbol{\mu}^{(0)}, \boldsymbol{\sigma}^{2(0)}]$.

Step 2 (E-step): Based on $\boldsymbol{\theta}^{(t)}$, firstly calculate the estimated latent variables matrix $\hat{\boldsymbol{\gamma}}^{(t)}$:

$$\hat{\boldsymbol{\gamma}}^{(t)} := \begin{pmatrix} \hat{\gamma}_{11} & \hat{\gamma}_{12} & \cdots & \hat{\gamma}_{1m} & \cdots & \hat{\gamma}_{1M} \\ \hat{\gamma}_{21} & \hat{\gamma}_{22} & \cdots & \hat{\gamma}_{2m} & \cdots & \hat{\gamma}_{2M} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ \hat{\gamma}_{n1} & \hat{\gamma}_{n2} & \cdots & \hat{\gamma}_{nm} & \cdots & \hat{\gamma}_{nM} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ \hat{\gamma}_{N1} & \hat{\gamma}_{N2} & \cdots & \hat{\gamma}_{Nm} & \cdots & \hat{\gamma}_{NM} \end{pmatrix}_{N\times M}$$

Each elements $\hat{\gamma}_{nm}$ represents estimated probability of the n-th data sample belongs to the m-th Gaussian Model:

$$\begin{aligned}
\hat{\gamma}_{nm} &:= E(\gamma_{n,m}|y_n, \boldsymbol{\theta}^{(t)}) \\
&= 1 \times P(\gamma_{n,m} = 1|y_n, \boldsymbol{\theta}^{(t)}) + 0 \times P(\gamma_{n,m} = 0|y_n, \boldsymbol{\theta}^{(t)}) \\
&= P(\gamma_{n,m} = 1|y_n, \boldsymbol{\theta}^{(t)}) \\
&= \frac{P(y_n|\gamma_{n,m} = 1, \boldsymbol{\theta}^{(t)})P(\gamma_{n,m} = 1|\boldsymbol{\theta}^{(t)})}{\sum_{m=1}^{M} P(y_n|\gamma_{n,m} = 1, \boldsymbol{\theta}^{(t)})P(\gamma_{n,m} = 1|y_n, \boldsymbol{\theta}^{t})} \qquad \text{(by Bayes' Rule)} \\
&= \frac{\pi_m \mathcal{N}(y_n|\mu_m^{(t)}, (\sigma^2)_m^{(t)})}{\sum_{m=1}^{M} \pi_m \mathcal{N}(y_n|\mu_m^{(t)}, (\sigma^2)_m^{(t)})}
\end{aligned}$$

Secondly, based on $\boldsymbol{\theta}^{(t)}$, the log-likelihood function $Q(\boldsymbol{\theta}^{(t)})$ would be:

$$\begin{aligned}
Q(\boldsymbol{\theta}^{(t)}) &:= ln(L(\boldsymbol{\theta}^{(t)})) \\
&= \sum_{m=1}^{M} \{(\sum_{n=1}^{N} \gamma_{nm}) ln(\pi_m) + \sum_{n=1}^{N} \gamma_{nm} ln(\mathcal{N}(y_n|\mu_m^{(t)}, (\sigma^2)_m^{(t)}))\}, \quad s.t. \sum_{m=1}^{M} \pi_m = 1
\end{aligned}$$

Step 3 (M-step): To maximize the log-likelihood function

$Q(\boldsymbol{\theta}^{(t)})$ found in the E-step, differentiate $Q(\boldsymbol{\theta}^{(t)})$ with respect to $\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\sigma^2}$, we have:

$$\hat{\boldsymbol{\pi}}_m = \frac{\sum_{n=1}^{N} \hat{\gamma}_{nm}}{\sum_{m=1}^{M} \sum_{n=1}^{N} \hat{\gamma}_{nm}}, \quad \hat{\boldsymbol{\mu}}_m = \frac{\sum_{n=1}^{N} \hat{\gamma}_{nm} \boldsymbol{y}_n}{\sum_{n=1}^{N} \hat{\gamma}_{nm}}, \quad \hat{\boldsymbol{\sigma}}_m^2 = \frac{\sum_{n=1}^{N} \hat{\gamma}_{nm} (\boldsymbol{y}_n - \hat{\boldsymbol{\mu}}_m)(\boldsymbol{y}_n - \hat{\boldsymbol{\mu}}_m)^T}{\sum_{n=1}^{N} \hat{\gamma}_{nm}}$$

Step 4 (Iterate until Convergence): Repeat Steps 2 and 3 until convergence: the change of value of $Q(\boldsymbol{\theta}^{(t)})$ between iterations is below a predetermined threshold. Export $\boldsymbol{\theta}^* = [\hat{\boldsymbol{\pi}}_m, \hat{\boldsymbol{\mu}}_m, \hat{\boldsymbol{\sigma}}_m^2]$ as the result.

Model Selection

In GMM application, the hyper-parameter M, representing the total number of Gaussian models used in the GMM, significantly influences the fitting result. For each distribution P_k, to determine the ideal $\hat{M}$, we iterate over a range of $[1, M_{max}]$ and select the model with $\hat{M}$ that yields the lowest Bayesian Information Criterion (BIC) value. A hyper-parameter ρ, representing the maximum component fraction, is employed to determine $M_{\max}$, the upper limit for the number of mixture models. Specifically, $M_{\max}$ is set as ρ times the total number of distinct label classes $\mathcal{Y}_k$, where $\rho \in (0, 1]$ and ρ is by default set to be 0.5.

$$\hat{M} = \arg\min_{M \in \mathbb{Z}} \{-2\ln(\mathcal{L}(\boldsymbol{\theta}^*)) + |M| \times \ln(N)\}, \quad 1 < M < \lceil \rho \times |\mathcal{Y}_k| \rceil, \quad \rho \in (0, 1]$$

The resulting $\boldsymbol{\theta}^*$ would be good summary of the heterogeneous data distributions and can be easily communicated and later used to estimate a global distribution. Compared to directly communicating the data or the data distribution, communicating a set of parameters from a fitted distribution greatly reduces the communication burden while preserving clients' privacy.

4.2.2 Local Distribution Estimation and Entropy Pooling

Upon receiving K fitting results $\boldsymbol{\theta}_k^*$, the Aggregator estimates the global model distribution as:

$$P(Y|\boldsymbol{\theta}_1^*, \ldots, \boldsymbol{\theta}_K^*, N_1, \ldots, N_K) = \sum_{k=1}^{K} p_k \left[\sum_{m=1}^{M} \pi_{mk} \mathcal{N}(\mu_{mk}, \sigma_{mk}^2)\right], \quad \sum_{k=1}^{K} p_k \sum_{m=1}^{M} \pi_{mk} = 1$$

where $p_k = \frac{N_k}{\sum_{k=1}^{K} N_k}$. The global distribution estimation is the weighted average of K GMMs so we have $\sum_{k=1}^{K} p_k \sum_{m=1}^{M} \pi_{mk} = 1$.

To quantifying the difference between two probability distribution, Kullback-Leibler divergence (KL-divergence) and Jensen-Shannon Divergence (JS-divergence) are commonly used. KL-divergence measures the relative entropy between distributions:

$$\mathrm{KLD}_k(P\|P_k) = \sum_{y\in\mathcal{Y}} P(y) \ln\left(\frac{P(y)}{P_k(y)}\right) = \sum_{y\in\mathcal{Y}} P(y) \left(\ln(P(y)) - \ln(P_k(y))\right)$$

We consider P_k, the distribution of client k, as an approximation of the global distribution P. After obtaining K KL-divergences, we calculate the pooled KL-divergence for each client k as the attention coefficient:

$$\alpha_k = \frac{\mathrm{KLD}_k}{\sum_{k=1}^{K} \mathrm{KLD}_k} = \frac{\sum_{y\in\mathcal{Y}} P(y) \left(\ln(P(y)) - \ln(P_k(y))\right)}{\sum_{k=1}^{K} \sum_{y\in\mathcal{Y}} P(y) \left(\ln(P(y)) - \ln(P_k(y))\right)}$$

$$\alpha_k = \frac{\sum_{y\in\mathcal{Y}} P(y) \left(\ln(P(y)) - \ln(P_k(y))\right)}{\sum_{k=1}^{K} \sum_{y\in\mathcal{Y}} P(y) \left(\ln(P(y)) - \ln(P_k(y))\right)}$$

4.3 Convergence Analysis

FedAvg converges to the global optimum at a rate of $\mathcal{O}(1/T)$ for strongly convex and smooth functions and non-iid data [38]. In this section, we show that FedEP converges to the global optimum at a rate of $\mathcal{O}(1/T)$ under a similar situation.

From an intuitive perspective, the aggregation in FedEP uses $\alpha_k = \frac{\text{KLD}_k}{\sum_{k=1}^{|K|}\text{KLD}_k}$ as weights, instead of using the number of data samples. When heterogeneity in a network increases, for a node in FedEP, having a higher KL divergence with the global distribution is similar to having more data samples in FedAvg. The convergence of FedAvg will not be differed from FedAvg.

4.3.1 Assumptions

Assumption 1 (Smoothness). Each local objective function F_k is Lipschitz smooth:

$$F_k(y) \leq F_k(x) + \nabla F_k(x)^T(y-x) + L\|y-x\|^2, \quad \forall k \in \{1,2,\ldots,K\}. \tag{4.1}$$

or in gradient form:

$$\|\nabla F_k(x) - \nabla F_k(y)\| \leq L\|x-y\|, \quad \forall k \in \{1,2,\ldots,K\} \tag{4.2}$$

Assumption 2 (Convexity). Each local objective function F_k is μ-strongly convex:

$$F_k(y) \geq F_k(x) + \nabla F_k(x)^T(y-x) + \frac{\mu}{2}\|y-x\|^2, \quad \forall k \in \{1,2,\ldots,K\}. \tag{4.3}$$

or in gradient form:

$$\|\nabla F_k(x) - \nabla F_k(y)\| \geq \frac{\mu}{2}\|x-y\|, \quad \forall k \in \{1,2,\ldots,K\}. \tag{4.4}$$

4.3.2 Full Device Participation

In full device participation scenario, for each round, all clients participates in training ($S_t = K$). We have the global optimization function $F(w)$

$$F(w) = \sum_{k=1}^{|K|} \alpha_k F_k(w), \sum_{k=1}^{|K|} \alpha_k = 1 \tag{4.5}$$

Now we first prove that given the assumption 1(4.1) and assumption 2(4.3) above, the global optimization function $F(w)$ is also Lipschitz smooth and μ-strongly convex. From (4.5) we have the gradient of $F(w)$:

$$\nabla F(w) = \sum_{k=1}^{|K|} \alpha_k \nabla F_k(w)$$

$$\|\nabla F(w_1) - \nabla F(w_2)\| = \|\sum_{k=1}^{|K|} \alpha_k(\nabla F_k(w_1) - \nabla F_k(w_2))\| \leq \sum_{k=1}^{|K|} \alpha_k \|\nabla F_k(w_1) - \nabla F_k(w_2)\|$$

From assumption 1 (4.1):

$$\sum_{k=1}^{|K|} \alpha_k \|\nabla F_k(w_1) - \nabla F_k(w_2)\| \le \sum_{k=1}^{|K|} \alpha_k L_k \|w_1 - w_2\|$$

let $L = \sum_{k=1}^{|K|} \alpha_k L_k$, we have:

$$\|\nabla F(w_1) - \nabla F(w_2)\| \le L \|w_1 - w_2\|$$

So, the global optimization function $F(w)$ is also Lipschitz smooth with $L = \sum_{k=1}^{|K|} \alpha_k L_k$.

Similarly, from assumption 2 (4.3) and (4.5), we get:

$$\sum_{k=1}^{|K|} \alpha_k F_k(w_1) \ge \sum_{k=1}^{|K|} \alpha_k \left[F_k(w_2) + \nabla F_k(x)^T (w_1 - w_2) + \frac{\mu_k}{2} \|w_1 - w_2\|^2 \right]$$

$$F(w_1) \ge F(w_2) + \sum_{k=1}^{|K|} \alpha_k \nabla F_k(w_2)^T (w_1 - w_2) + \sum_{k=1}^{|K|} \frac{\alpha_k \mu_k}{2} \|w_1 - w_2\|^2$$

Let $\mu = \sum_{k=1}^{|K|} \alpha_k \mu_k$:

$$F(w_1) \ge F(w_2) + \nabla F(x)^T (w_1 - w_2) + \frac{\mu}{2} \|w_1 - w_2\|^2$$

So $F(w)$ is also μ-strongly convex, where $\mu = \sum_{k=1}^{|K|} \alpha_k \mu_k$.

We have now proved that $F(w)$ is also Lipschitz smooth and μ-strongly convex function $F(w)$.

Utilizing the Lipschitz smooth properties of $F(w)$,

$$F(w_{t+1}) \le F(w_t) + \nabla F(w_t)^T (w_{t+1} - w_t) + L \|w_{t+1} - w_t\|^2, \quad \forall k \in \{1, 2, \ldots, K\}.$$

for gradient decent $w_{t+1} = w_t - \eta \nabla F(w_t)$, we have:

$$F(w_{t+1}) \le F(w_t) - \eta \|\nabla F(w_t)\|^2 + L \| - \eta \nabla F(w_t)\|^2, \quad \forall k \in \{1, 2, \ldots, K\}.$$

then simplify as

$$F(w_{t+1}) \le F(w_t) - (\eta - L\eta^2) \|\nabla F(w_t)\|^2, \quad \forall k \in \{1, 2, \ldots, K\}.$$

To achieve descending , we need $(\eta - L\eta^2) > 0$, so we have $0 < \eta < \frac{1}{L}$.

Chapter 5

Implementation

5.1 Adaptions to Fedstellar

Fedstellar [39] was employed as the platform for executing DFL scenarios. Implemented in Python, Fedstellar offers Docker-based containerized simulations and virtual devices for FL. Communication among Docker containers is facilitated using the Google Remote Procedure Call (gRPC) protocol. The platform supports the creation of federations by allowing customization of parameters such as the number and type of devices training FL models, the network topology connecting them, the machine and deep learning algorithms employed, and the datasets utilized by each participant. Fedstellar enables users to execute FL in decentralized, semi-decentralized, and centralized configurations, effectively managing node connectivity and scenario deployment.

5.1.1 Algorithm Implementation

Fedstellar offers a containerized method to simulate real-world FL environments. In this framework, each client operates as a Docker container, enabling isolated and reproducible execution. Each container is instantiated and managed by a **Node** class, which encompasses all the necessary configurations for the selected dataset, training algorithms, and aggregation algorithms.

To implement the FedEP algorithm, we extend the base functionality provided by the **Node** class. This is achieved through the creation of a specialized subclass named **NodeFedEP**. The **NodeFedEP** class inherits from the Node class and introduces additional functions and attributes specific to the FedEP algorithm.

Listing 5.1: NodeFedEP: a class for Node using FedEP(logging.info removed)

```
1 class NodeFedEP(Node):
2 
3     epsilon_prime = 0.1
4 
5     def __init__(self,
6             idx,
```

```
7               experiment_name,
8               model,
9               data,
10              host="127.0.0.1",
11              port=None,
12              config=Config,
13              learner=LightningLearner,
14              encrypt=False,
15              model_poisoning=False,
16              poisoned_ratio=0,
17              noise_type='gaussian',
18      ):
19          Node.__init__(self, idx, experiment_name, model, data, host, port, config,
                learner, encrypt, model_poisoning, poisoned_ratio, noise_type)
20
21          self._labels = np.array([label for _, label in self.learner.data.train_dataloader
                ().dataset])
22          self._distribution_fitting_max_components_fraction = 0.5
23          self._EM_algorithm_max_iterations = 1500
24          self._EM_algorithm_epsilon = 1e-6
25          self._gaussian_epsilon = 1e-2
26          self.theta = None
27
28          # FedEP's locks for communicating distribution
29          self._distribution_communication_lock = threading.Lock()
30
31      def distribution_fitting_and_communication(self):
32          """
33          FedEP's round of sharing distribution characteristics to learn the aggregation
                weights
34          """
35          fitting_thread = threading.Thread(target = self._distribution_fitting)
36          fitting_thread.start()
37          fitting_thread.join()
38
39          broadcast_thread = threading.Thread(target = self._distribution_broadcast)
40          broadcast_thread.start()
41          broadcast_thread.join()
42
43          if self.config.participant["device_args"]["role"] == Role.AGGREGATOR:
44              self.aggregator.pooling(self._labels)
45
46      def _distribution_fitting(self):
47          """
48          FedEP's round of fitting local distribution
49          """
50          Y = np.array(np.sort(self._labels))
51          # deciede the maximum number of mixture components
52          Ms = math.ceil(len(set(Y.tolist())) * self.
                _distribution_fitting_max_components_fraction)
53          theta_hs = np.empty(Ms, dtype=object)
54          likelihood_hs = np.zeros(Ms)
55          BICs = np.zeros(Ms)
56          # AICs = np.zeros(Ms)
57          for M in range(0,Ms):
58              theta_hs[M], likelihood_hs[M] = self._expectation_maximum_algorithm(M+1 , Y)
59              BICs[M] = -2*likelihood_hs[M] + M * np.log(len(Y))
60              # AICs[M] = -2*likelihood_hs[M] + 2 * M
61          min_BIC_index = np.argmin(BICs)
62          self.theta = theta_hs[min_BIC_index]
63
64      def _distribution_broadcast(self):
65          """
66          FedEP's round of broadcasting the distribution characteristics to the other nodes
67          """
68          self.aggregator._lock_to_start_pooling.acquire()
69          # gets aggregator ready to accecpt distributions
70          if self.config.participant["device_args"]["role"] == Role.AGGREGATOR:
71              self.aggregator.set_waiting_distribution(True)
72
73          # get neighbors
```

```
        neighbors = self._neighbors.get_all()
        # broadcast distribution
        try:
            if self.config.participant["device_args"]["role"] != Role.IDLE:
                local_distribution = {"".join(self.addr): (self.theta, self.learner.
                    get_num_samples()[0])}
                self.aggregator.add_distribution(
                    theta= self.theta,
                    addr= self.addr,
                    weight = self.learner.get_num_samples()[0],
                    all_neighbors = neighbors
                )
                for des in neighbors:
                    if des != self.addr:
                        self.send_distribution(des, local_distribution)
        except Exception as e:
            print(f"[NodeFedEP] Error broadcasting distribution: {e}")

    def _expectation_maximum_algorithm(self, M, Y):
        '''
        derived theta by EM algorithm

        Args: M: the number of mixture components
              Y: complete list of lable that is ununique, for example
                 ([0,0,0,....,8,8,8,9,9,9,])
        return:
            theta: the parameters of the GMMs given M and Y
        '''
        theta = self._parameter_initialization(M,Y)
        likelihood_prev = 0
        theta_prev = theta
        iteration = 0
        while iteration <  self._EM_algorithm_max_iterations:
            gamma_lm, n_m = self._E_step(theta,Y)
            theta, likelihood = self._M_step(gamma_lm, n_m, Y)
            iteration += 1
            if likelihood == np.NINF or math.isnan(likelihood):
                return theta_prev, likelihood_prev
            if abs(likelihood - likelihood_prev) < self._EM_algorithm_epsilon:
                break
            likelihood_prev = likelihood
            theta_prev = theta
        return theta, likelihood

    def _parameter_initialization(self, M,Y):
        '''
        Initialized theta

        Args: M: the number of mixture components
              Y: complete list of lable that is ununique, for example
                 ([0,0,0,....,8,8,8,9,9,9,])
        '''
        L=len(Y)
        pi = np.random.rand(M)
        pi /= np.sum(pi)
        mu = Y[np.random.choice(L, M, replace=False)]
        sigma_squared = [np.var(Y.tolist())] * M
        # theta_0
        return np.column_stack((pi, mu, sigma_squared))

    def _gaussian(self, Y, mu, sigma_squared):
        '''
        probability density function of Gaussian distribution
        '''
        return np.exp(-np.square(Y-mu+self._gaussian_epsilon)/(2*(sigma_squared+self.
            _gaussian_epsilon)))/(np.sqrt(2 * np.pi * (sigma_squared + self.
            _gaussian_epsilon)))

    def _E_step(self, theta,Y):
        '''
        given theta, calculate the latent variable gamma_lm and the number of samples n_m
```

```
            for each mixture component m
140         '''
141         M = theta.shape[0]
142         gamma_lm = np.zeros((len(Y),M))
143         n_m = np.zeros(M)
144         sum_gaussians = torch.zeros([len(Y)])
145         for m in range(M):
146             sum_gaussians += theta[m,0] * self._gaussian(Y, theta[m,1], theta[m,2])
147         for m in range(M):
148             gamma_lm[:,m] = theta[m,0] * self._gaussian(Y, theta[m,1], theta[m,2])/
                    sum_gaussians
149             n_m[m] = np.sum(gamma_lm[:,m])
150         return gamma_lm, n_m
151 
152     def _M_step(self, gamma_lm, n_m, Y):
153         '''
154         given gamma and n_m, calculate the new theta
155         '''
156         pi_h = n_m / len(Y)
157         mu_h = np.array([gamma_lm[:,m] @ Y / n_m[m] for m in range(len(n_m))])
158         sigma_squared_h = np.array([gamma_lm[:,m] @ ((Y-mu_h[m])**2 + self.
                _gaussian_epsilon) / n_m[m] for m in range(len(n_m))])
159         theta_h = np.column_stack((pi_h, mu_h, sigma_squared_h))
160 
161         likelihood = np.sum([n_m[m] * np.log(pi_h[m])+ gamma_lm[:,m] @ np.log(self.
                _gaussian(Y, mu_h[m], sigma_squared_h[m])+self._gaussian_epsilon) for m in
                range(len(n_m))])
162         return theta_h, likelihood
```

As shown in Listing 5.1, a Node encompassed FedEP algorithm can be initialized by **NodeFedEP()**. Running **node.distribution_fitting_and_communication()** will initialize a process that consisting of a fitting phase and a communication phase. The fitting phase can started as long as the dataset config is set.

The FedEP algorithm inherits from Aggregator, a predefined fundamental aggregation algorithm class.

Listing 5.2: FedEP: an aggreagtion algorithm class

```
1 
2 class FedEP(Aggregator):
3     """
4 
5     """
6 
7     def __init__(self, node_name="unknown", config=None):
8         super().__init__(node_name, config)
9         self.config = config
10        self.role = self.config.participant["device_args"]["role"]
11        self._waiting_distribution = False
12        self._distributions = {}
13        self._thetas_with_samples_num = {}
14        self._prediction_precision = 1e-3
15        self._prediction_epsilon = 1e-2
16        self._gaussian_epsilon = 1e-1
17        self._theta_global = None
18        self._prob_global = None
19        self._clients_probs = None
20        self._KL_divergence = None
21        self._labels = None
22        self._labels_unique = None
23        self.alpha_k = None
24        self._lock_to_start_pooling = threading.Lock()
25 
26    def set_waiting_distribution(self, TrueOrFalse):
27        self._waiting_distribution = TrueOrFalse
28 
```

```
29      def _gaussian(self, Y, mu, sigma_squared):
30          '''
31          probability density function of Gaussian distribution
32          '''
33          return np.exp(-np.square(Y-mu+self._gaussian_epsilon)/(2*(sigma_squared+self.
                _gaussian_epsilon)))/(np.sqrt(2 * np.pi * (sigma_squared + self.
                _gaussian_epsilon)))
34
35      def aggregate(self, models):
36          """
37          Ponderated average of the models.
38
39          Args:
40              models: Dictionary with the models (node: model,num_samples).
41              model : {layer: tensor, ...}
42          """
43          # Check if there are models to aggregate
44          if len(models) == 0:
45              logging.error(
46                  "[FedEP] Trying to aggregate models when there is no models"
47              )
48              return None
49
50          # Create a shape of the weights use by all nodes
51          accum = {layer: torch.zeros_like(param) for layer, param in list(models.values())
                [-1][0].items()}
52
53          # Add weighted models
54
55          for address, model in models.items():
56              for layer in accum:
57                  accum[layer] += model[0][layer] * self.alpha_k[address]
58          return accum
59
60
61      def _predict_likelihood(self, theta, precision=4):
62          '''
63          given theta and labels, calculate the likelihood of the labels
64          '''
65          prob = np.zeros(len(self._labels_unique))
66          for i in range(len(prob)):
67              prob[i] = np.round(np.sum(theta[:,0] * self._gaussian(self._labels_unique[i],
                     theta[:,1], theta[:,2])), precision)
68          return prob
69
70      def pooling(self,labels):
71
72          self._lock_to_start_pooling.acquire()
73
74          self._labels = labels
75          self._labels_unique = np.unique(labels)
76          # Total Samples
77          total_samples = sum([weight for _, weight in self._thetas_with_samples_num.values
                ()])
78
79          q_k = {[k][0]: v[1]/total_samples for k, v in self._thetas_with_samples_num.items
                ()}
80
81          self._theta_global = {
82              address: np.array([
83                  [round(param[0] * (weight / total_samples), 5), param[1], param[2]]
84                  for param in theta
85              ])
86              for address, (theta, weight) in self._thetas_with_samples_num.items()
87          }
88
89          # Calculate global probability of labels
90          '''
91          self._prob_global = [] with a length equals number of labels space and summing up
                 to 1
92          '''
```

```
        prob_global = []
        for label in self._labels_unique:
            prob_label = 0
            for theta in self._theta_global.values():
                for g in theta:
                    prob_label +=  g[0] * self._gaussian(label, g[1], g[2])
            prob_global.append(prob_label)
        self._prob_global = prob_global

        # Calculate client probabilities
        self._clients_probs = {
            address : self._predict_likelihood(theta)
            for address,theta in self._theta_global.items()
        }

        # Calculate KL divergences
        self._KL_divergence = {
            address: np.sum([self._prob_global[i] * np.log2((self._prob_global[i] + self._prediction_epsilon) / (probs[i] + self._prediction_epsilon)) for i in range(len(self._prob_global))])
            for address,probs in self._clients_probs.items()
        }

        # Calculate alpha_k
        kl_sum = np.sum([kl_div for kl_div in self._KL_divergence.values()])
        self.alpha_k = {
            address: kl_div / kl_sum
            for address, kl_div in self._KL_divergence.items()
        }


    def add_distribution(self, theta, addr, weight, all_neighbors):

        self._thetas_with_samples_num["".join(addr)] = (theta, weight)

         # all distributions recieved
        if len(self._thetas_with_samples_num) > len(all_neighbors):
            # self.set_waiting_distribution(False)
            self._lock_to_start_pooling.release()
        else:
            print(f"({self.node_name}) waiting for more distributions ({len(self._thetas_with_samples_num)}/{len(all_neighbors)+1})")
            pass

        return self._thetas_with_samples_num
```

As shown in Listing 5.2, The FedEP class encapsulates functionality for distributed aggregation and pooling in a collaborative learning context. It manages distributions **thetas_with_samples_num**, calculates probabilities **prob_global** and **clients_probs**, and uses Gaussian functions for likelihood predictions. The use of locks **lock_to_start_pooling** ensures thread-safe operations during aggregation and pooling phases.

5.1.2 Communication Implementation

In the Fedstellar platform, gRPC is used as the communciation protocals among dockers. Besides implemented the algorithm, modifying the communciation is also necesscary.

Listing 5.3: GRPC Remote Services

```
    def send_distribution(self, des, distribution):
        try:
            # Initialize channel and stub
            channel = grpc.insecure_channel(des)
```

```
5              stub = node_pb2_grpc.NodeServicesStub(channel)
6              # Encode the distribution using the learner's encode_parameters method
7              encoded_distribution = self.learner.encode_parameters(params=distribution)
8              # Send the encoded distribution to the destination
9              res = stub.add_distribution(
10                 node_pb2.Distributions(
11                     source=self.addr,
12                     distribution=encoded_distribution,
13                 ),
14                 timeout=10,
15             )
16             # Handling errors
17             if res.error:
18                 print(f"[{self.addr}] Error while sending a model: {res.error}")
19             # Close the channel after sending the distribution
20             channel.close()
21
22         except Exception as e:
23             print(f"({self.addr}) Cannot send model to {des}. Error: {str(e)}")
24
25     def add_distribution(self, request, context):
26         """
27         Adds a distribution to the aggregator
28         """
29         try:
30             distribution = self.learner.decode_parameters(request.distribution)
31             print(f"[NodeFedEP] Received distribution: {distribution}")
32             received_theta, received_weight = next(iter(distribution.values()))
33             self.aggregator.add_distribution(
34                 theta=received_theta,
35                 addr=[request.source],
36                 weight=received_weight,
37                 all_neighbors = self._neighbors.get_all()
38             )
39             return node_pb2.ResponseMessage(error="")
40         except Exception as e:
41             return node_pb2.ResponseMessage(error=str(e))
```

In Listing 5.3, **send_distribution** and **add_distribution** is written to facilitate the communication of distributions between nodes in a distributed system. They leverage GRPC for efficient, cross-platform communication and employ encoding/decoding mechanisms to serialize/deserialize distributions.

Chapter 6

Evaluation

6.1 Experiment Setup

In this section, we compare FedEP against FedAvg, the baseline algorithm, and SCAFFOLD[3] across three datasets: MNIST, FashionMNIST, and CIFAR-10. To model non-IID (non-independent and identically distributed) data, we utilize the Dirichlet distribution to generate distribution for data of each classes.

The Dirichlet distribution is a family of continuous multivariate probability distributions parameterized by a vector $\boldsymbol{\alpha}$ of positive reals. It is often denoted as $\mathrm{Dir}(\boldsymbol{\alpha})$.

The probability density function (pdf) of the Dirichlet distribution is given by:

$$f(\boldsymbol{p};\boldsymbol{\alpha}) = \frac{1}{B(\boldsymbol{\alpha})}\prod_{i=1}^{K} p_i^{\alpha_i - 1}, B(\boldsymbol{\alpha}) = \frac{\prod_{i=1}^{K}\Gamma(\alpha_i)}{\Gamma\left(\sum_{i=1}^{K}\alpha_i\right)}$$

where $\mathbf{p} = (p_1, p_2, \ldots, p_k)$ is a point in the probability simplex ($p_i \geq 0$ and $\sum_{i=1}^{k} p_i = 1$),

The variance of p_i is dependent on the concentration parameter α:

$$\mathrm{Var}(p_i) = \frac{\alpha_i(\sum_{j=1}^{k}\alpha_j - \alpha_i)}{(\sum_{j=1}^{k}\alpha_j)^2(\sum_{j=1}^{k}\alpha_j + 1)}$$

As the parameter α increases in the Dirichlet distribution, the term $\alpha_i(\sum_{j=1}^{k}\alpha_j - \alpha_i)$ will increase albeit at a slower rate compared to the denominator, leading to a decreases of $\mathrm{Var}(p_i)$ and a p_i closer to $\mathbf{E}[p_i]$. The Dirichlet distribution is therefore adept at modeling data heterogeneity through varying alpha values, which serve as priors influencing the shape of distribution of the dataset samples.

We examine two scenarios: Pure Non-IID and Mixed Non-IID. We examine two scenarios: Pure Non-IID and Mixed Non-IID. Pure Non-IID considers cases where all nodes in the network contain the same degree of heterogeneity, while Mixed Non-IID explores networks

where two different levels of heterogeneity exist. Comprehensive evaluation over two scenarios allows for rigorously assess the robustness and efficacy of FedEP compared to FedAvg and SCAFFOLD under varying degrees of data heterogeneity in real-world FL environments.

6.1.1 Pure Non-IID Scenario

For the Pure Non-IID scenario, we generate datasets with alpha values of 20, 5, 1, 0.5, and 0.1. Experiments are conducted using FedAvg, FedEP, and SCAFFOLD algorithms across these datasets to assess performance under multiple degrees of non-IID conditions.

Algorithms	**α values**	**Dataset**	**Metrics**
FedEP	(20, 5, 1, 0.5, 0.1)	MNIST	accuracy*, precision*, recall*, F1-Score*, Converge Speed**
		FMNIST	
		CIFAR-10	
FedAvg	(20, 5, 1, 0.5, 0.1)	MNIST	
		FMNIST	
		CIFAR-10	
SCAFFOLD	(20, 5, 1, 0.5, 0.1)	MNIST	
		FMNIST	
		CIFAR-10	

Table 6.1: Experiment Setup of Pure Non-IID Scenario

6.1.2 Mixed Non-IID Scenario

In the Mixed Non-IID scenario, we explore three mixed cases: 20%:80%, 50%:50%, and 80%:20%. Here, the notation represents the proportion of IID to non-IID clients. Specifically, in the 2:8 case, 20% of clients are IID while 80% are non-IID. We generate 2 IID clients with an alpha value of 50, and 8 non-IID clients with alpha values of 5, 1, and 0.5. Similarly, in the 5:5 case, 5 clients are generated with an alpha value of 50, and the remaining 5 with alpha values of 5, 1, and 0.5. In the 8:2 case, 8 clients are generated with an alpha value of 50, and 2 with alpha values of 5, 1, and 0.5. Experiments in this scenario are also conducted using FedAvg, FedEP, and SCAFFOLD to evaluate the algorithms' performance under mixed IID and non-IID conditions.

6.2 Results on one dataset

We utilize experiments on the CIFAR-10 dataset as an example to illustrate performance. For comprehensive results on the other two datasets, MNIST and Fashion-MNIST (FM-NIST), please refer to the appendix.

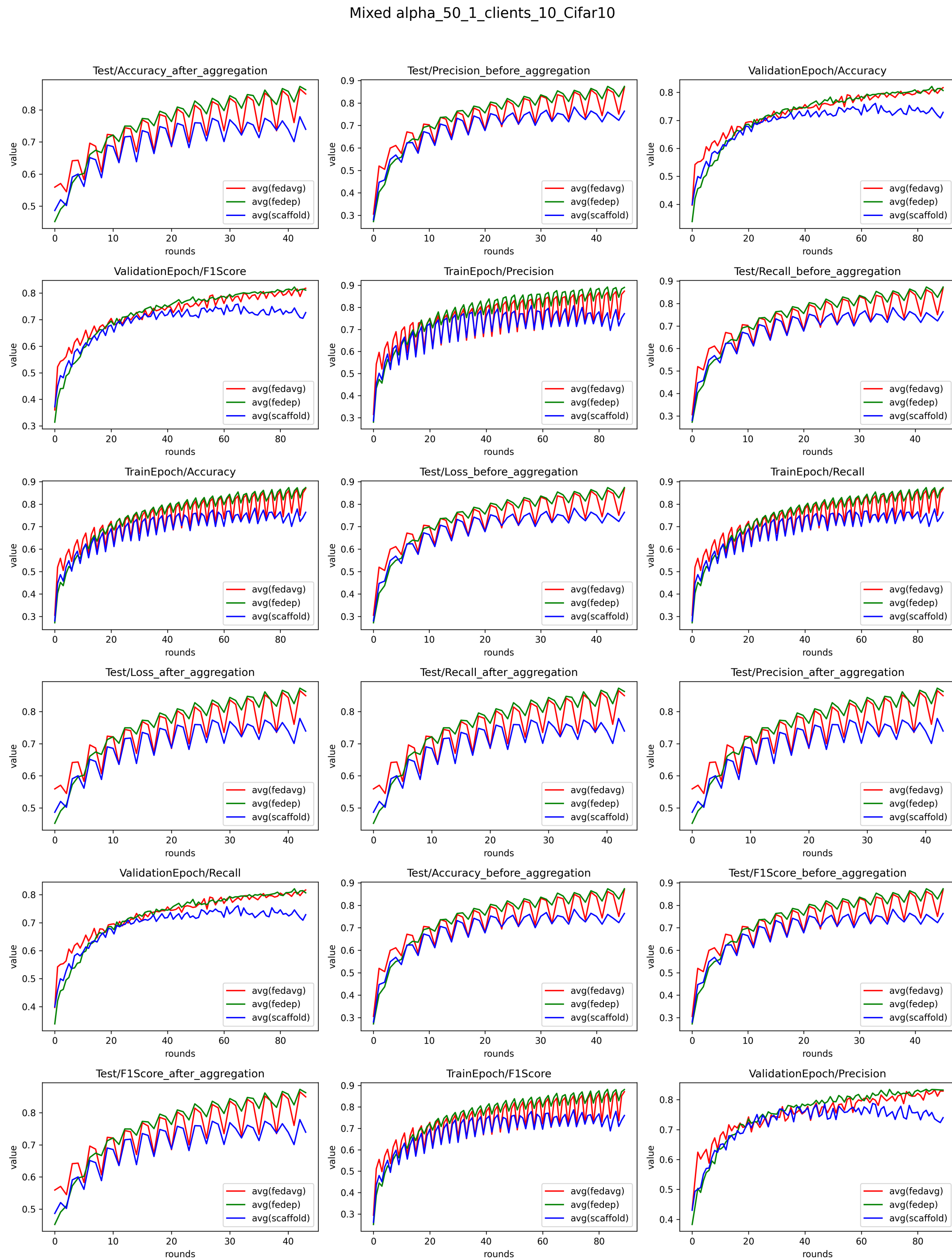

Figure 6.1: Experiments with $\alpha_1 = 50(50\%)$, $\alpha_2 = 1(50\%)$, Cifar-10, 10 clients

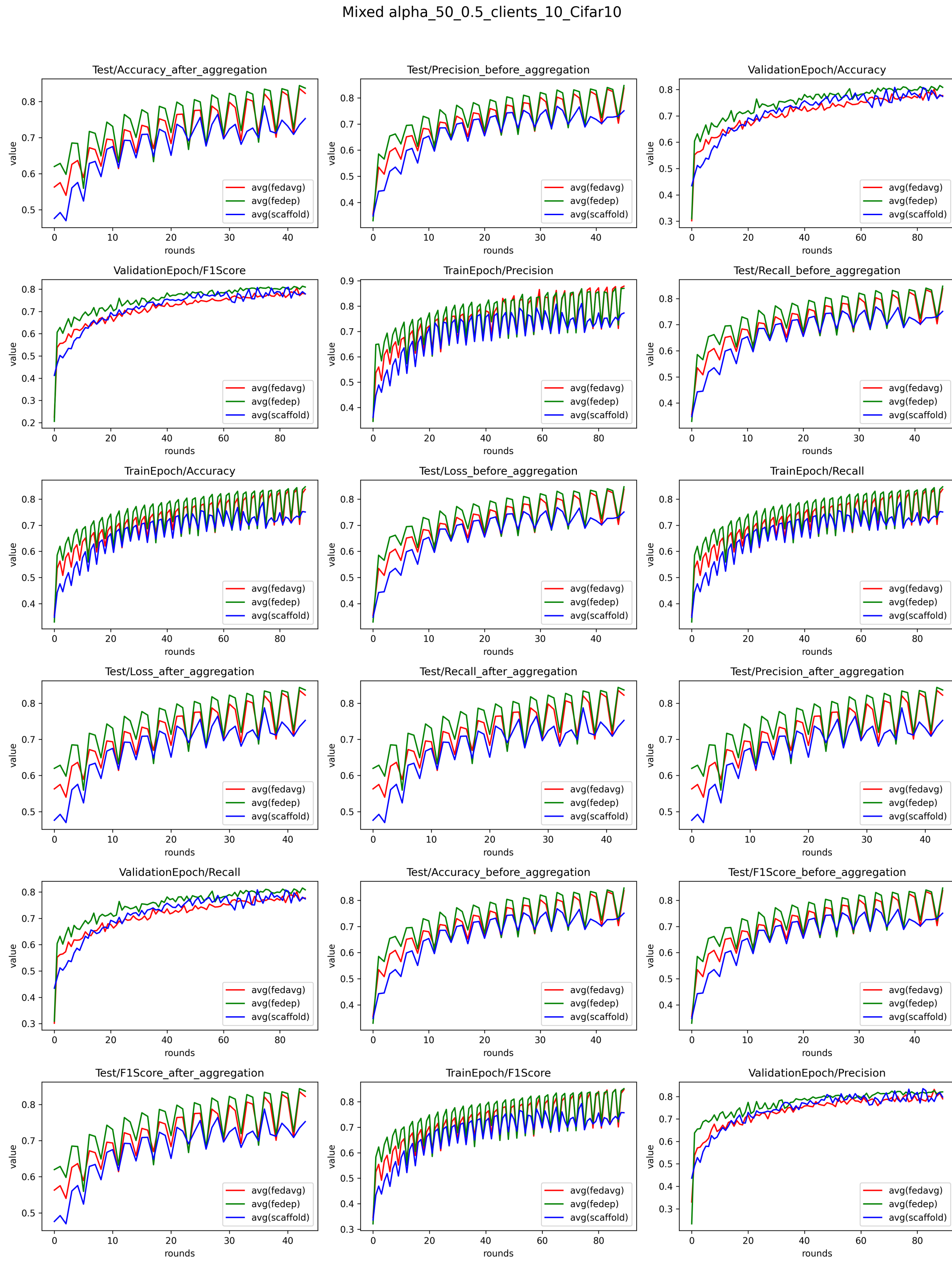

Figure 6.2: Experiments with $\alpha_1 = 50(50\%)$, $\alpha_2 = 0.5(50\%)$, Cifar-10, 10 clients

6.3 Discussion

After testing the performance of FedEP, FedAvg, and SCAFFOLD, we found that FedEP outperforms the other algorithms under a mixed non-IID scenario. As shown in Figure B.1, considering the average accuracy, recall, and F1 score across the 10 nodes, FedEP consistently achieves the most stable and highest outcomes, surpassing both FedAvg and

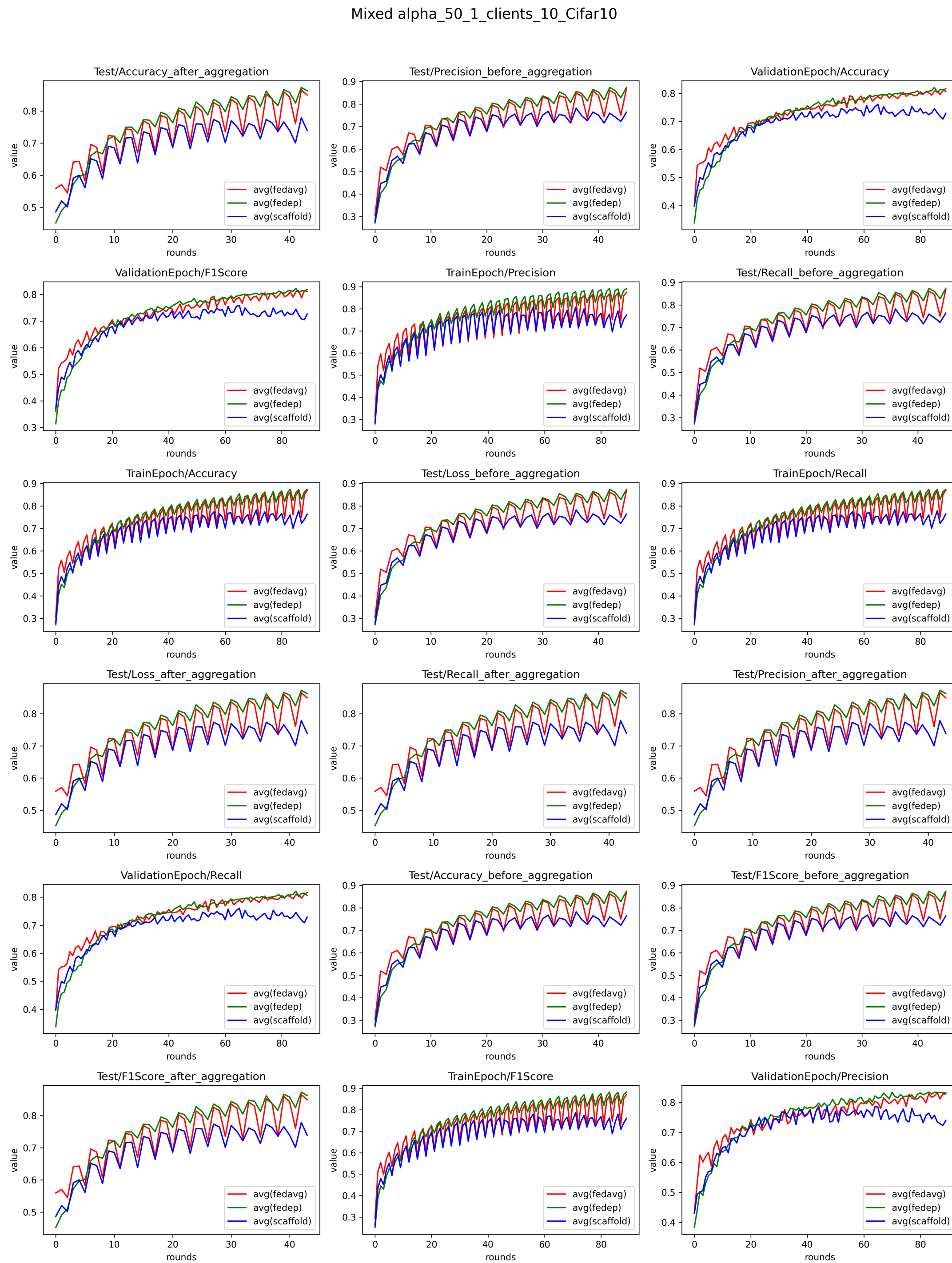

Figure 6.3: Experiments with $\alpha_1 = 50(50\%)$, $\alpha_2 = 1(50\%)$, Cifar-10, 10 clients

SCAFFOLD in most cases.

To examine the performance of individual nodes before averaging6.10, it becomes evident that when the network includes nodes with multiple classes of data and nodes with highly heterogeneous data, FedAvg struggles to manage these variances. In our experiments, these nodes often appear as outliers, adversely affecting the overall accuracy of

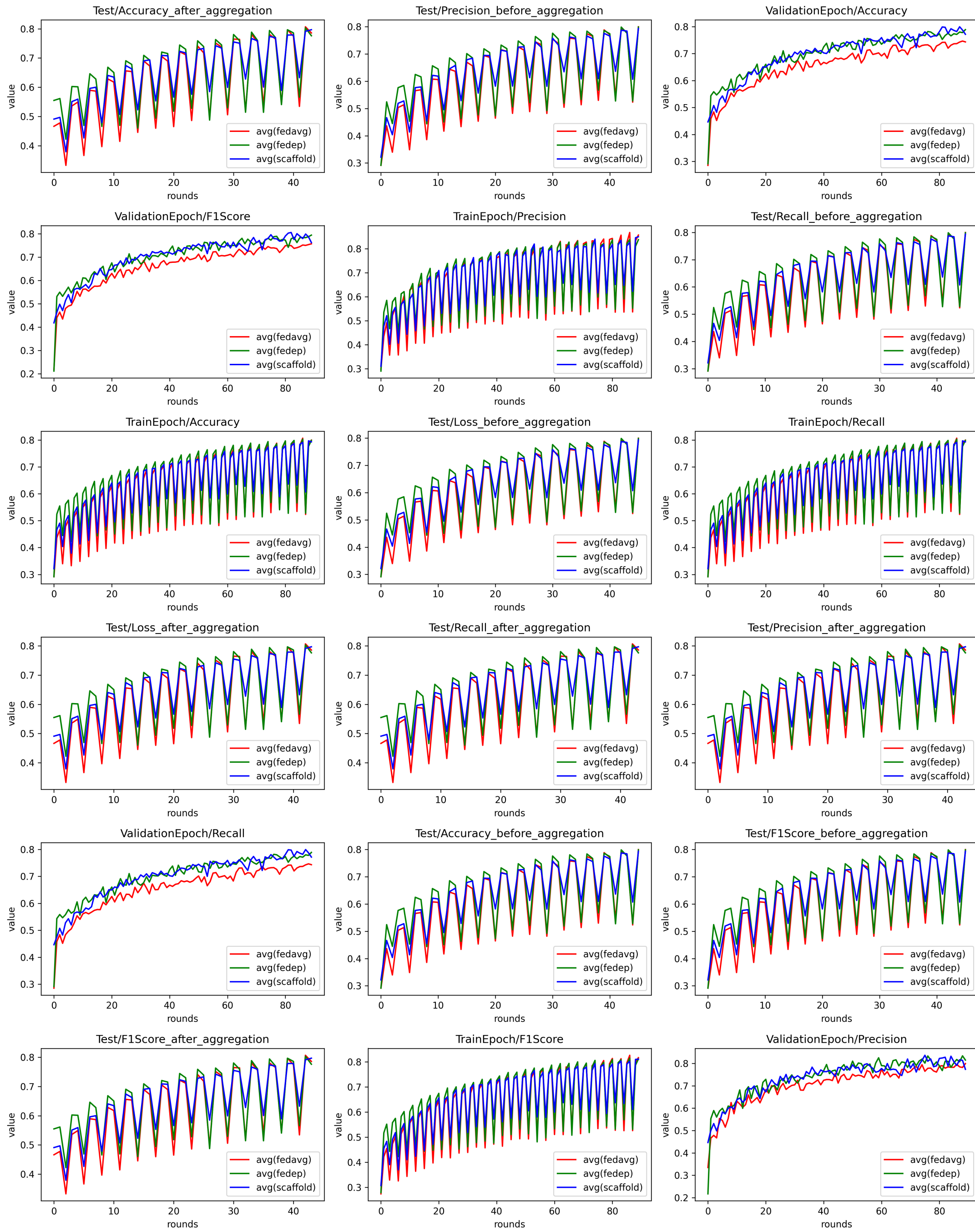

Figure 6.4: Experiments with $\alpha_1 = 50(20\%)$, $\alpha_2 = 0.5(80\%)$, Cifar-10, 10 clients

the system. In contrast, FedEP assigns smaller weights to these outliers, allowing them to converge more quickly by leveraging the influence of other nodes. Consequently, the average performance of the system improves significantly.

By effectively mitigating the impact of outliers, FedEP demonstrates its robustness and adaptability in diverse and heterogeneous FL environments. This highlights the potential

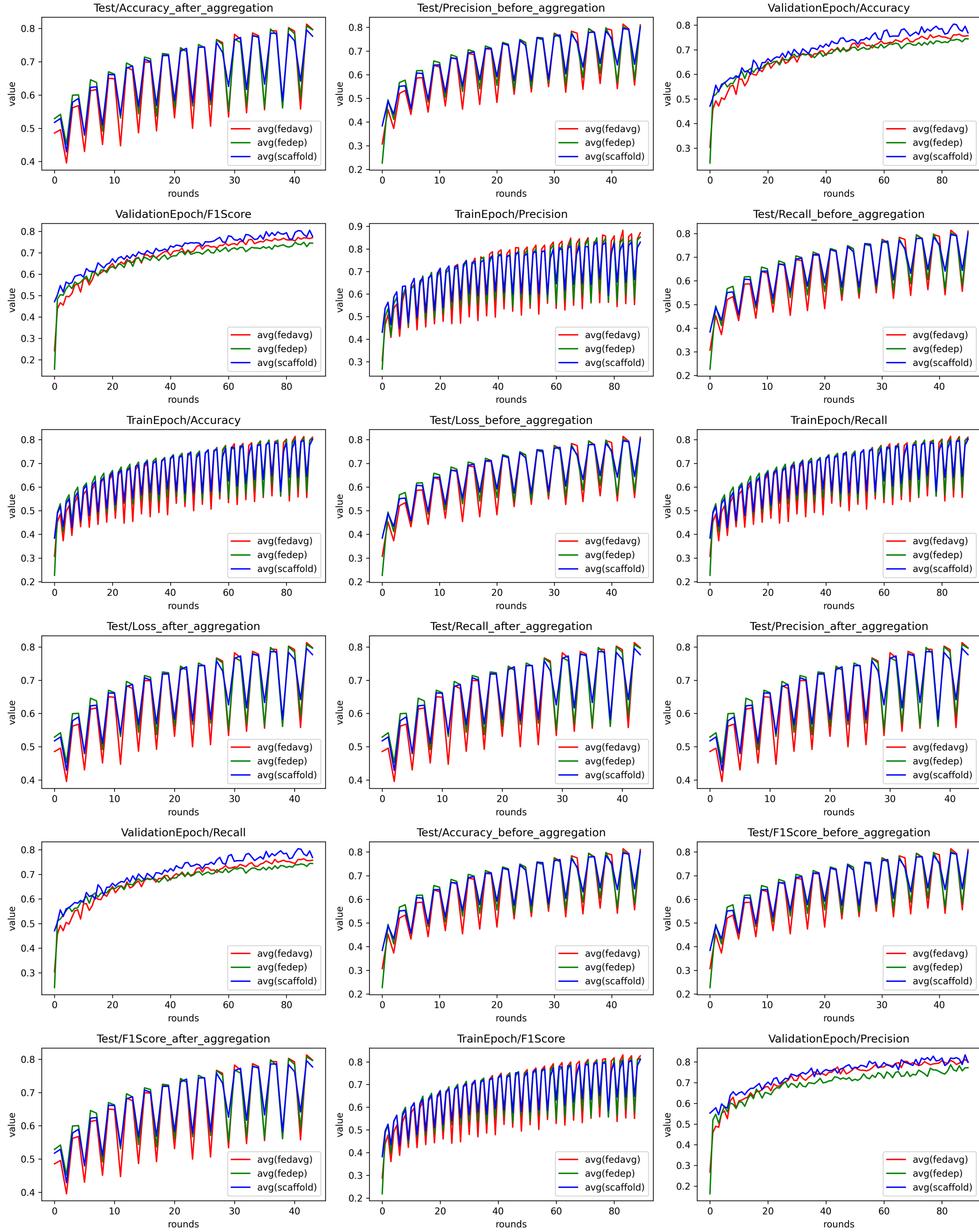

Figure 6.5: Experiments with $\alpha_1 = 50(20\%)$, $\alpha_2 = 1(80\%)$, Cifar-10, 10 clients

of FedEP as a superior algorithm for achieving reliable and high-performing FL outcomes, particularly in scenarios characterized by significant data heterogeneity.

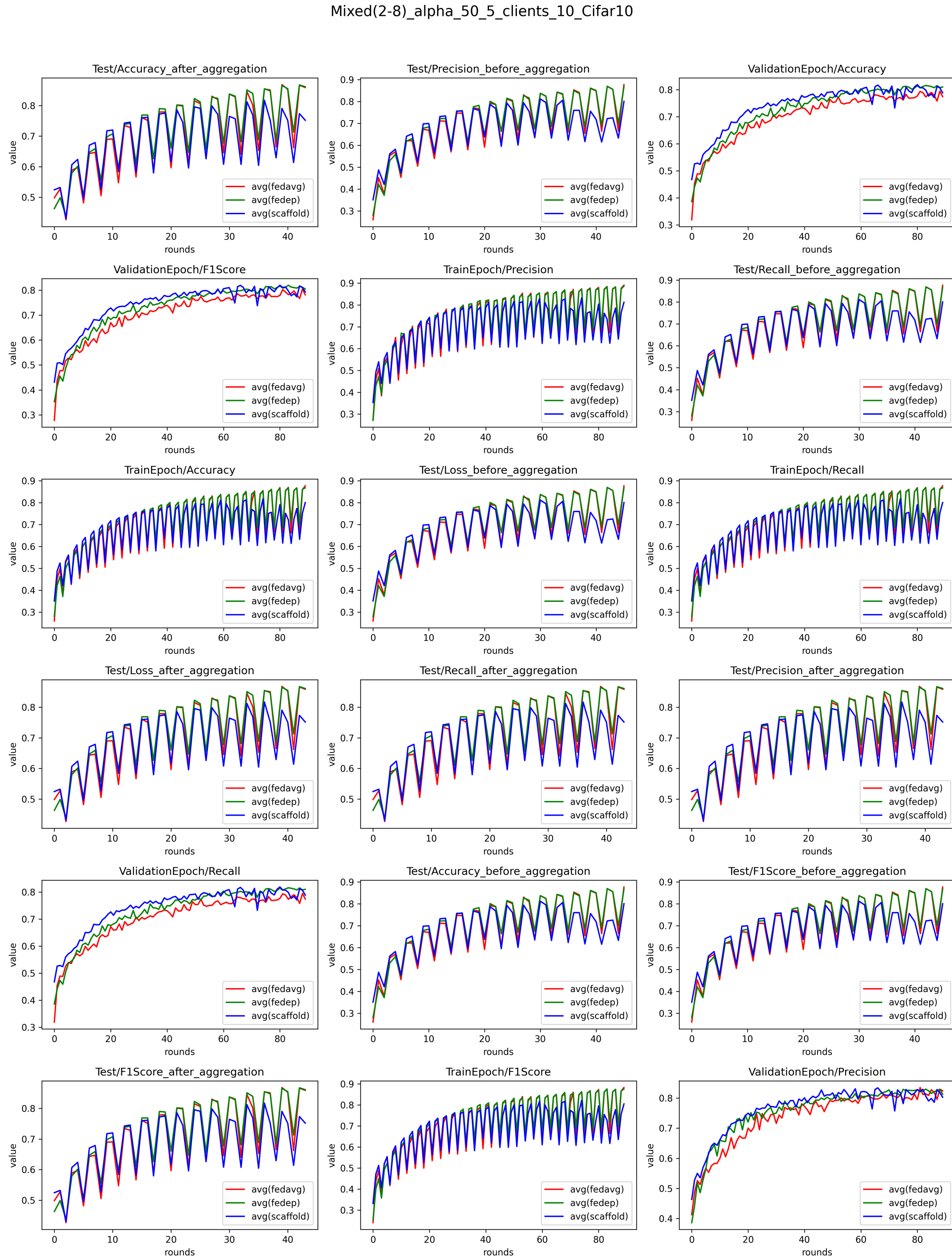

Figure 6.6: Experiments with $\alpha_1 = 50(20\%)$, $\alpha_2 = 5(80\%)$, Cifar-10, 10 clients

Algo	**Dataset**	**Num(G1)***	**Num(G2)***	**α(G1)***	**α(G2)***	**Metrics**
FedEP	MNIST	5	5	50	(5, 1, 0.5)	test accuracy, train accuracy, precision, recall, F1-Score, Converge Speed
		2	8	50	(5, 1, 0.5)	
		8	2	50	(5, 1, 0.5)	
	FMNIST	5	5	50	(5, 1, 0.5)	
		2	8	50	(5, 1, 0.5)	
		8	2	50	(5, 1, 0.5)	
	Cifar-10	5	5	50	(5, 1, 0.5)	
		2	8	50	(5, 1, 0.5)	
		8	2	50	(5, 1, 0.5)	
FedAvg	MNIST	5	5	50	(5, 1, 0.5)	
		2	8	50	(5, 1, 0.5)	
		8	2	50	(5, 1, 0.5)	
	FMNIST	5	5	50	(5, 1, 0.5)	
		2	8	50	(5, 1, 0.5)	
		8	2	50	(5, 1, 0.5)	
	Cifar-10	5	5	50	(5, 1, 0.5)	
		2	8	50	(5, 1, 0.5)	
		8	2	50	(5, 1, 0.5)	
SCAFFOLD	MNIST	5	5	50	(5, 1, 0.5)	
		2	8	50	(5, 1, 0.5)	
		8	2	50	(5, 1, 0.5)	
	FMNIST	5	5	50	(5, 1, 0.5)	
		2	8	50	(5, 1, 0.5)	
		8	2	50	(5, 1, 0.5)	
	Cifar-10	5	5	50	(5, 1, 0.5)	
		2	8	50	(5, 1, 0.5)	
		8	2	50	(5, 1, 0.5)	

Table 6.2: Experiment Setup of Mixed Non-IID Scenario

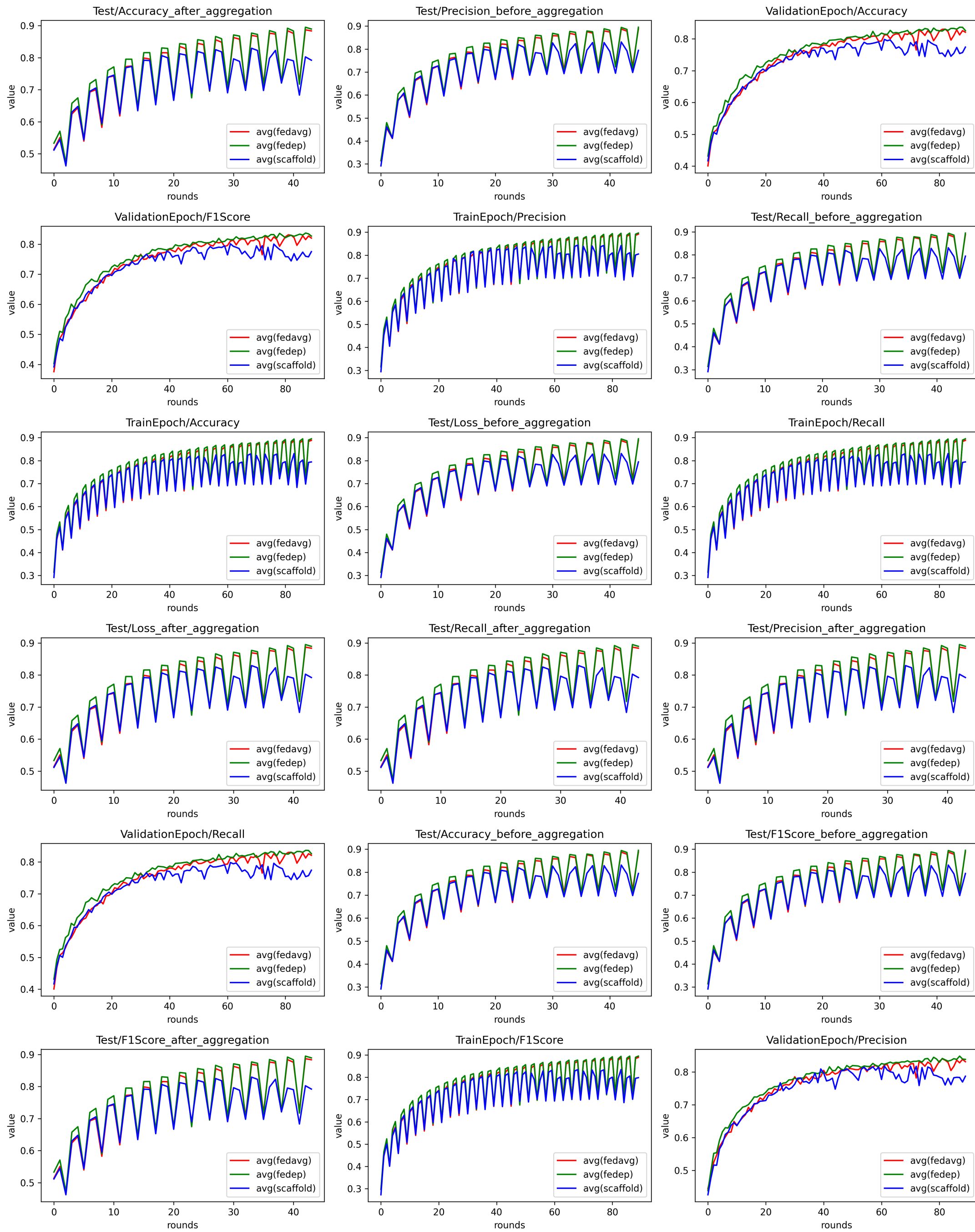

Figure 6.7: Experiments with $\alpha_1 = 50(80\%)$, $\alpha_2 = 0.5(20\%)$, Cifar-10, 10 clients

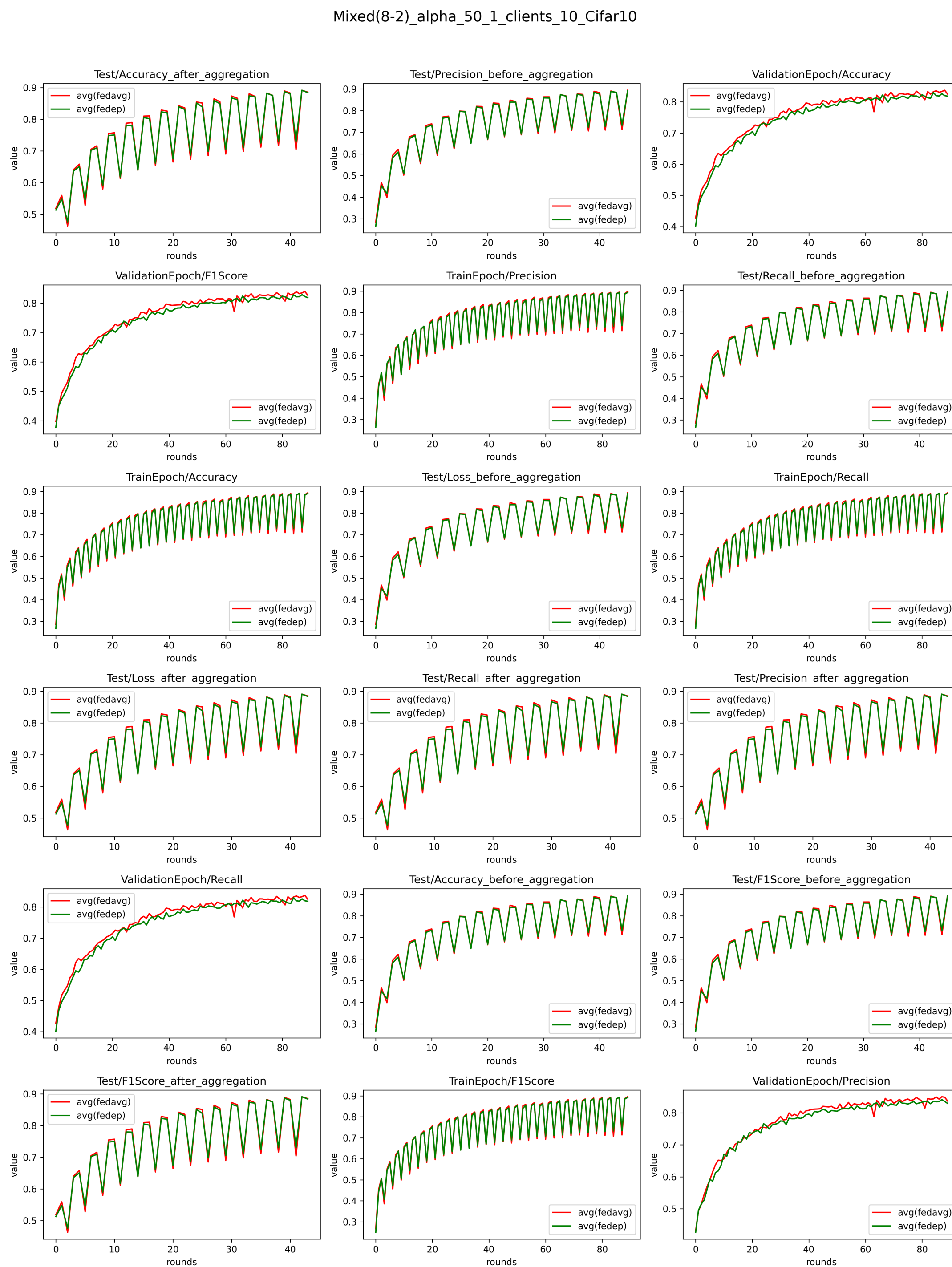

Figure 6.8: Experiments with $\alpha_1 = 50(80\%)$, $\alpha_2 = 1(20\%)$, Cifar-10, 10 clients

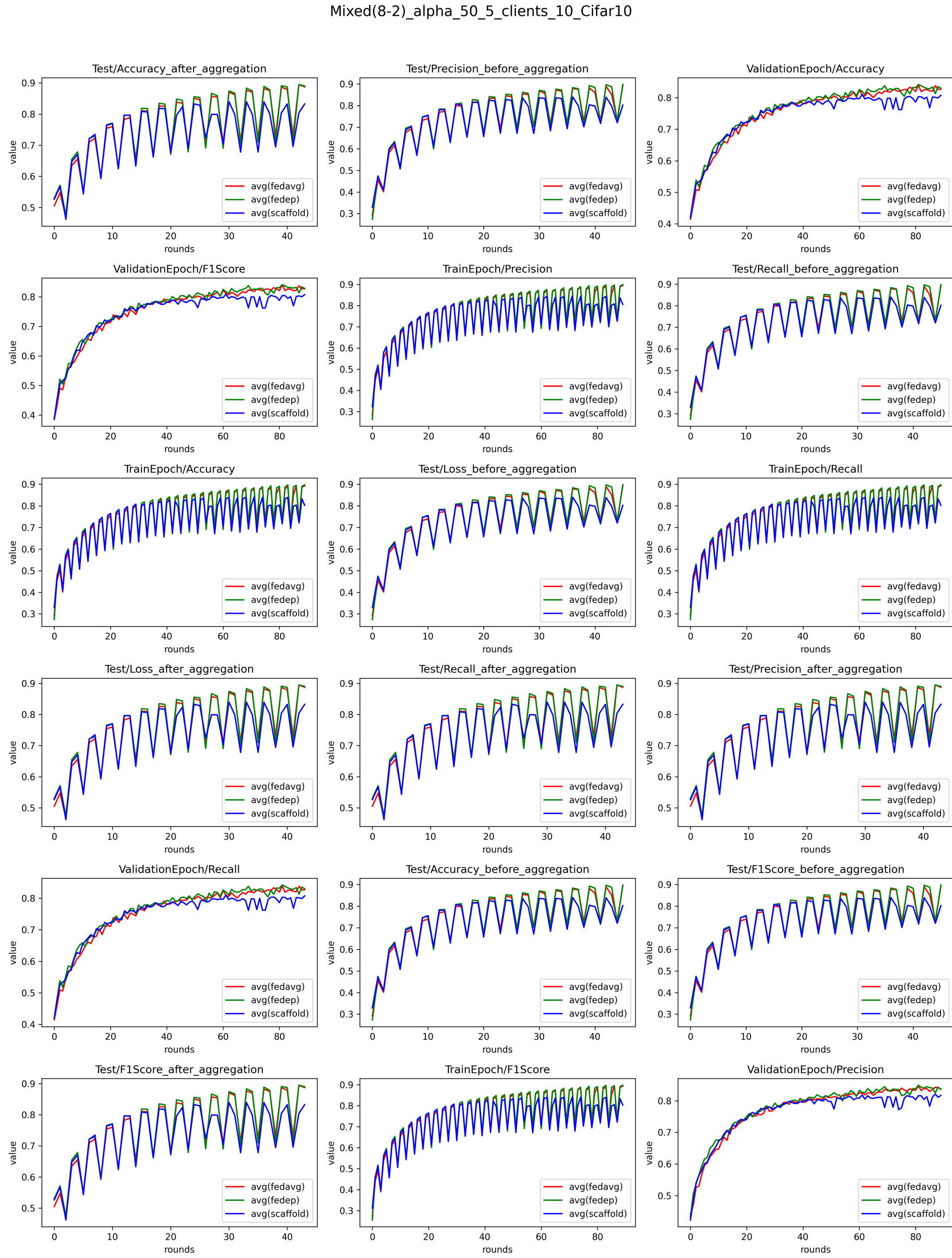

Figure 6.9: Experiments with $\alpha_1 = 50(80\%)$, $\alpha_2 = 5(20\%)$, Cifar-10, 10 clients

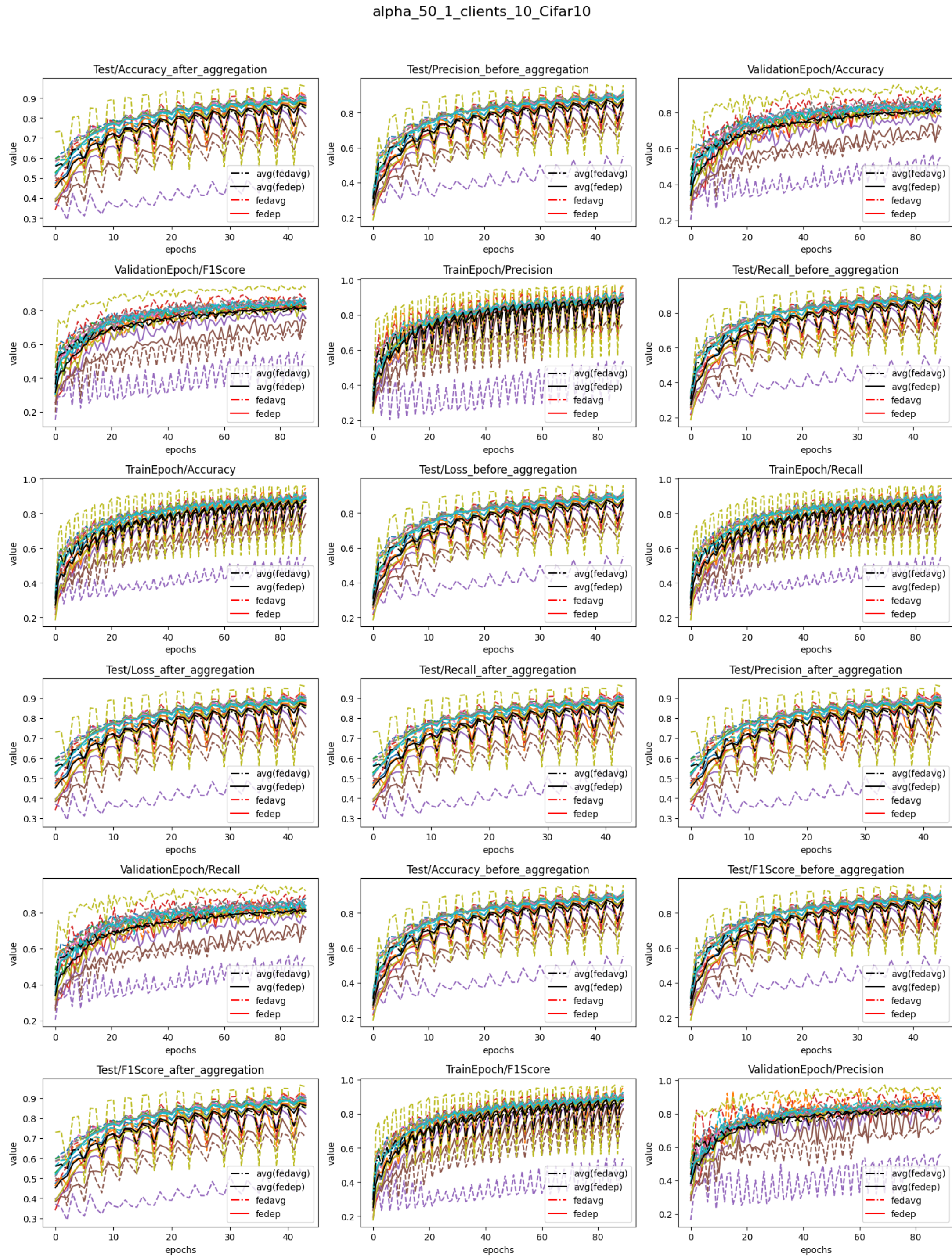

Figure 6.10: Experiments with $\alpha_1 = 50(50\%)$, $\alpha_2 = 1(50\%)$, Cifar-10, 10 clients

Chapter 7

Summary and Conclusions

In this work, we introduced a novel Federated Entropy Pooling Algorithm (FedEP), specifically designed to address the client drift issue caused by data heterogeneity in extreme Non-IID scenarios. Unlike traditional methods such as FedAvg, which aggregate model updates based on the volume of local data, FedEP incorporates the distribution of local data into the aggregation process. By leveraging the Kullback-Leibler (KL) divergence, FedEP adjusts the attention allocated to each model, expanding the weights for clients with more unique datasets and reducing the weights for those with less distinctive data. This optimized aggregation function enhances the global model training process by mitigating the adverse effects of data heterogeneity.

FedEP can be conceptually viewed as a re-parameterization of the widely used Federated Averaging (FedAvg) algorithm, with a distinctive focus on incorporating distributional differences across clients. This approach ensures a more balanced and robust model training process, particularly in environments with significant data variability.

To evaluate the efficacy of FedEP, we conducted experiments on the MNIST, FMNIST, and CIFAR-10 datasets within both CFL and DFL networks. These experiments were performed under two different degrees of Non-IID settings: Moderate Non-IID and Extreme Non-IID. The results of our experiments were compared against the performance of FedAvg and SCAFFOLD algorithms.

Our comprehensive evaluation demonstrates that FedEP outperforms FedAvg and SCAFFOLD in scenarios with high data heterogeneity, providing a more effective solution for FL environments. This highlights the practical applicability and robustness of FedEP in real-world FL settings, where data privacy, computational constraints, and non-IID data distributions are prevalent challenges.

Abbreviations

ML Machine Learning

SGD Stochastic Gradient Descent

FL Federated Learning

CFL Centralized Federated Learning

DFL Decentralized Federated Learning

IID Independent and Identically Distributed

HFL Horizontal Federated Learning

VFL Vertical Federated Learning

EM Expectation Maximization

GMM Gaussian Mixture Model

CD Client Drift

EP Entropy Pooling

KL Kullback-Leibler

List of Figures

2.1 Federated Average(FedAvg) ($S_t \neq K$). 7
2.2 Simplified client drift of 2 client nodes in FedAvg[[3]]. 8
2.3 Variations of FL Architectures: CFL, Semi-DFL, and DFL [11] 8
2.4 Horizontal FL, Vertical FL, and Federated Transfer Learning [12] 11

3.1 The data-sharing Strategy[19] 15

6.1 Experiments with $\alpha_1 = 50(50\%)$, $\alpha_2 = 1(50\%)$, Cifar-10, 10 clients 37
6.2 Experiments with $\alpha_1 = 50(50\%)$, $\alpha_2 = 0.5(50\%)$, Cifar-10, 10 clients 38
6.3 Experiments with $\alpha_1 = 50(50\%)$, $\alpha_2 = 1(50\%)$, Cifar-10, 10 clients 39
6.4 Experiments with $\alpha_1 = 50(20\%)$, $\alpha_2 = 0.5(80\%)$, Cifar-10, 10 clients 40
6.5 Experiments with $\alpha_1 = 50(20\%)$, $\alpha_2 = 1(80\%)$, Cifar-10, 10 clients 41
6.6 Experiments with $\alpha_1 = 50(20\%)$, $\alpha_2 = 5(80\%)$, Cifar-10, 10 clients 42
6.7 Experiments with $\alpha_1 = 50(80\%)$, $\alpha_2 = 0.5(20\%)$, Cifar-10, 10 clients 44
6.8 Experiments with $\alpha_1 = 50(80\%)$, $\alpha_2 = 1(20\%)$, Cifar-10, 10 clients 45
6.9 Experiments with $\alpha_1 = 50(80\%)$, $\alpha_2 = 5(20\%)$, Cifar-10, 10 clients 46
6.10 Experiments with $\alpha_1 = 50(50\%)$, $\alpha_2 = 1(50\%)$, Cifar-10, 10 clients 47

A.1 Experiments with $\alpha_1 = 50(50\%)$, $\alpha_2 = 1(50\%)$, MNIST, 10 clients 62
A.2 Experiments with $\alpha_1 = 50(50\%)$, $\alpha_2 = 0.5(50\%)$, MNIST, 10 clients 63
A.3 Experiments with $\alpha_1 = 50(50\%)$, $\alpha_2 = 1(50\%)$, MNIST, 10 clients 64
A.4 Experiments with $\alpha_1 = 50(20\%)$, $\alpha_2 = 0.5(80\%)$, MNIST, 10 clients 65
A.5 Experiments with $\alpha_1 = 50(20\%)$, $\alpha_2 = 1(80\%)$, MNIST, 10 clients 66

A.6 Experiments with $\alpha_1 = 50(20\%)$, $\alpha_2 = 5(80\%)$, MNIST, 10 clients 67

A.7 Experiments with $\alpha_1 = 50(80\%)$, $\alpha_2 = 0.5(20\%)$, MNIST, 10 clients 68

A.8 Experiments with $\alpha_1 = 50(80\%)$, $\alpha_2 = 1(20\%)$, MNIST, 10 clients 69

A.9 Experiments with $\alpha_1 = 50(80\%)$, $\alpha_2 = 5(20\%)$, MNIST, 10 clients 70

B.1 Experiments with $\alpha_1 = 50(50\%)$, $\alpha_2 = 1(50\%)$, FashsionMNIST, 10 clients 72

B.2 Experiments with $\alpha_1 = 50(50\%)$, $\alpha_2 = 0.5(50\%)$, FashsionMNIST, 10 clients 73

B.3 Experiments with $\alpha_1 = 50(50\%)$, $\alpha_2 = 1(50\%)$, FashsionMNIST, 10 clients 74

B.4 Experiments with $\alpha_1 = 50(20\%)$, $\alpha_2 = 0.5(80\%)$, FashsionMNIST, 10 clients 75

B.5 Experiments with $\alpha_1 = 50(20\%)$, $\alpha_2 = 1(80\%)$, FashsionMNIST, 10 clients 76

B.6 Experiments with $\alpha_1 = 50(20\%)$, $\alpha_2 = 5(80\%)$, FashsionMNIST, 10 clients 77

B.7 Experiments with $\alpha_1 = 50(80\%)$, $\alpha_2 = 0.5(20\%)$, FashsionMNIST, 10 clients 78

B.8 Experiments with $\alpha_1 = 50(80\%)$, $\alpha_2 = 1(20\%)$, FashsionMNIST, 10 clients 79

B.9 Experiments with $\alpha_1 = 50(80\%)$, $\alpha_2 = 5(20\%)$, FashsionMNIST, 10 clients 80

List of Tables

3.1 Existing Methods in mitigating data heterogeneity 14

6.1 Experiment Setup of Pure Non-IID Scenario 36

6.2 Experiment Setup of Mixed Non-IID Scenario 43

List of Algorithms

1 *Federated Entropy Pooling (FedEP)*. All K nodes are indexed by k; E is the number of local epochs; η is the learning rate; M is the number of models in GMM; T is the training rounds. 20

2 *Pre-train Distribution Fitting Algorithm.*The hyper-parameter ρ, representing the maximum component fraction, is used to determine $M_{\max}$, the upper limit for the number of mixture models. Specifically, $M_{\max}$ is set as ρ times the total number of distinct label classes $\mathcal{Y}_k$, where $\rho \in (0, 1]$. 21

3 *Expectation Maximization Algorithm.* M is the total number of Gaussian models used in the GMM. 21

List of Listings

5.1 NodeFedEP: a class for Node using FedEP(logging.info removed) 27
5.2 FedEP: an aggreagtion algorithm class 30
5.3 GRPC Remote Services 32

Appendix A

Performance Comparison on MNIST

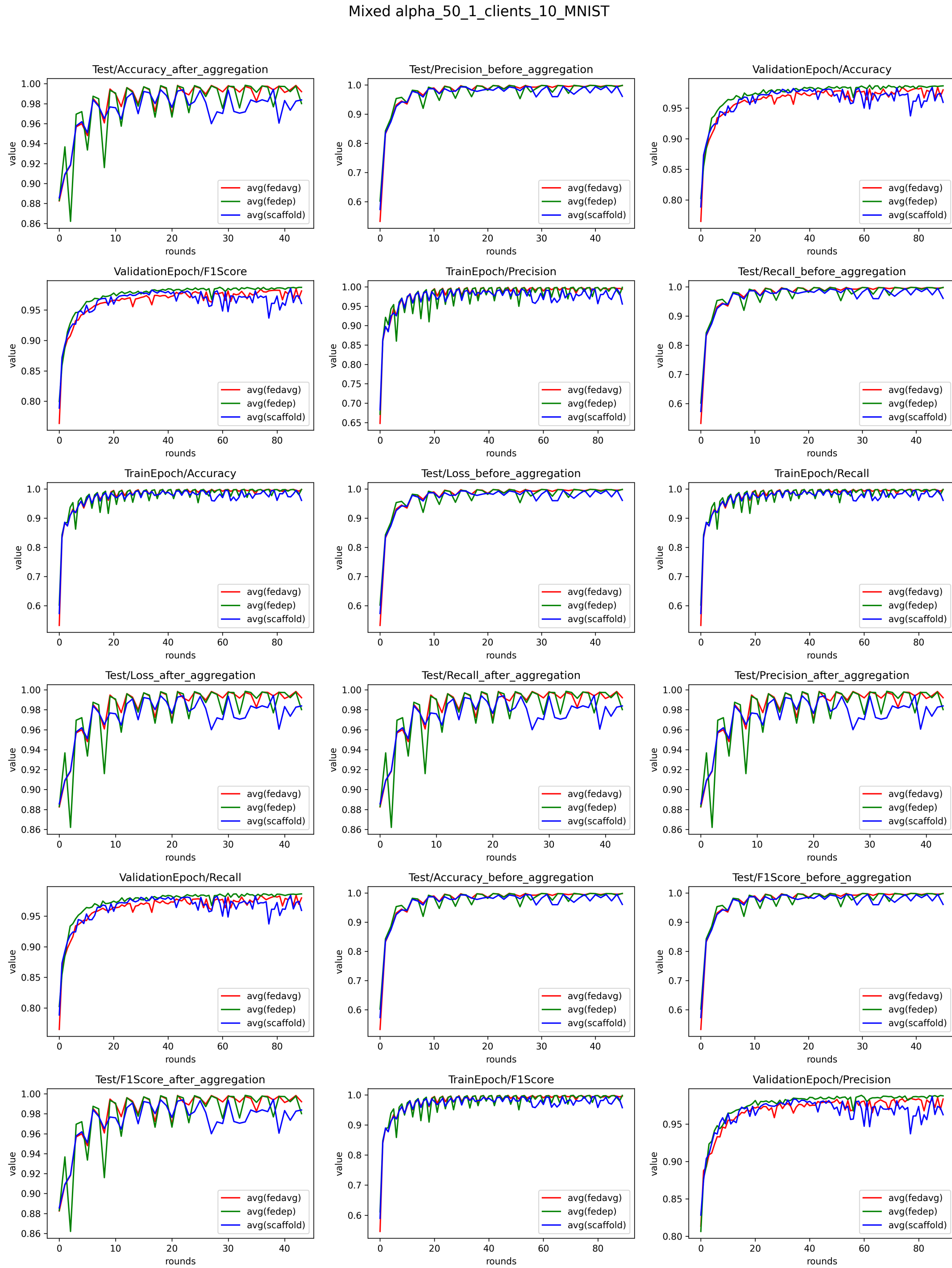

Figure A.1: Experiments with $\alpha_1 = 50(50\%)$, $\alpha_2 = 1(50\%)$, MNIST, 10 clients

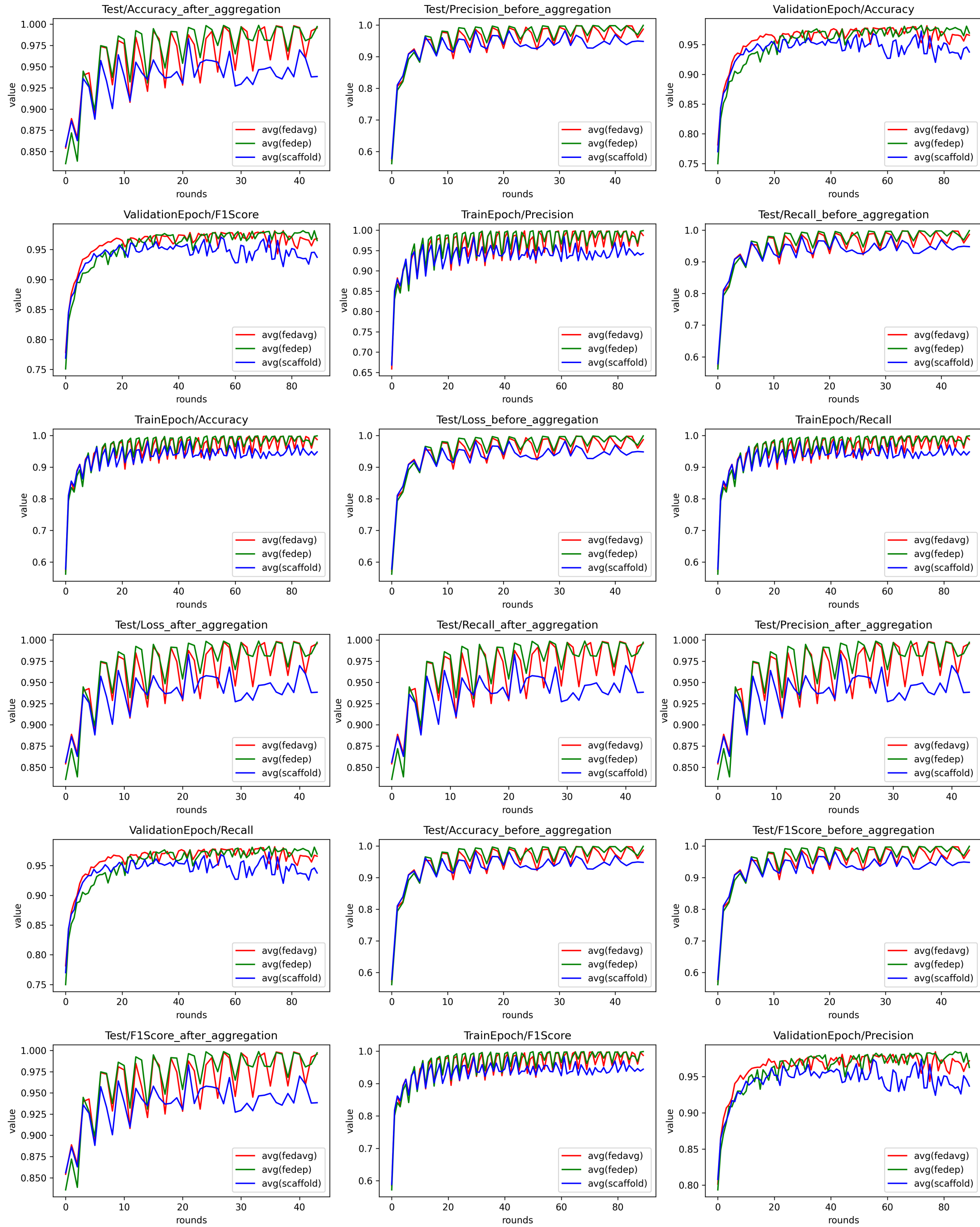

Figure A.2: Experiments with $\alpha_1 = 50(50\%)$, $\alpha_2 = 0.5(50\%)$, MNIST, 10 clients

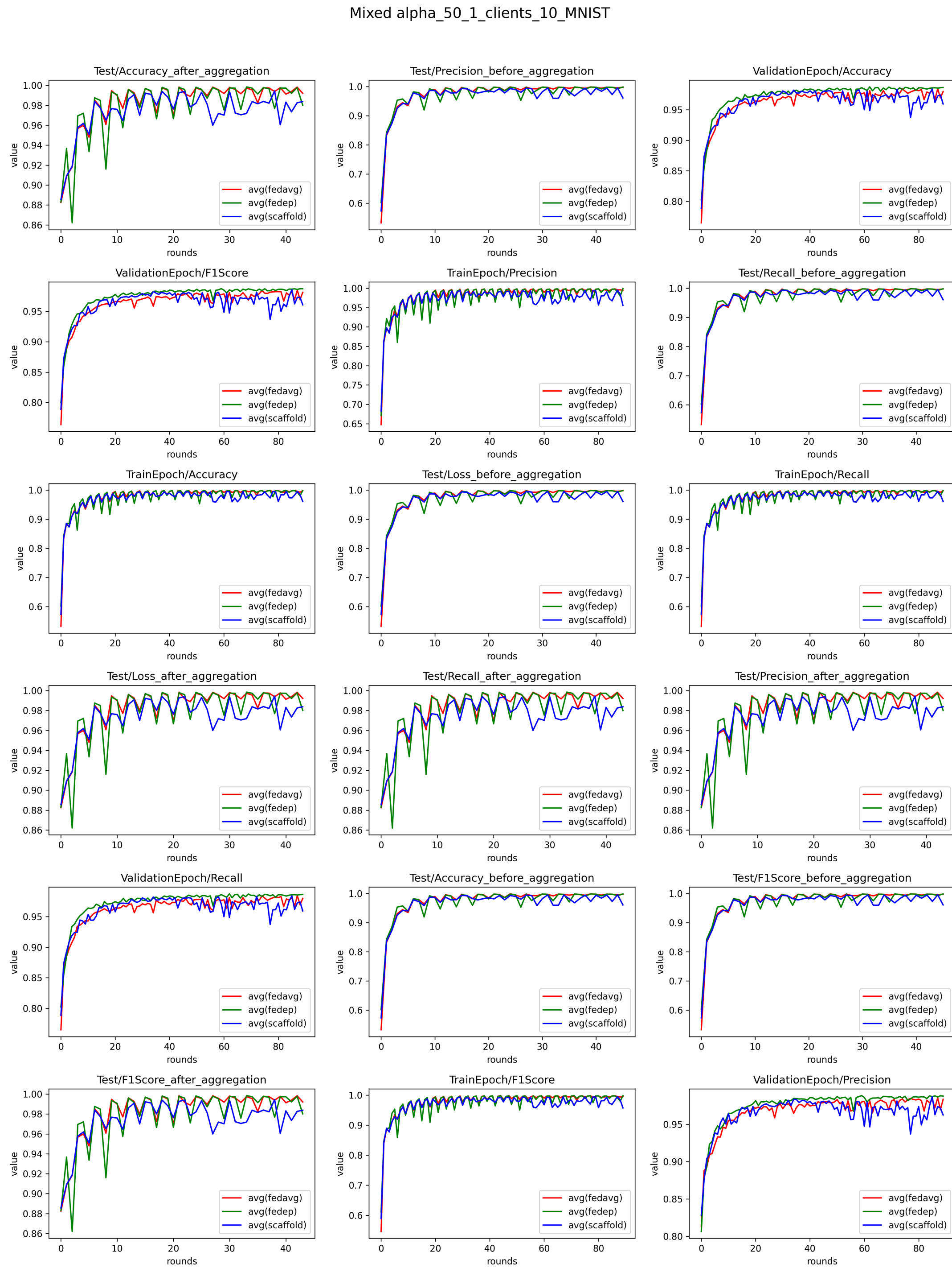

Figure A.3: Experiments with $\alpha_1 = 50(50\%)$, $\alpha_2 = 1(50\%)$, MNIST, 10 clients

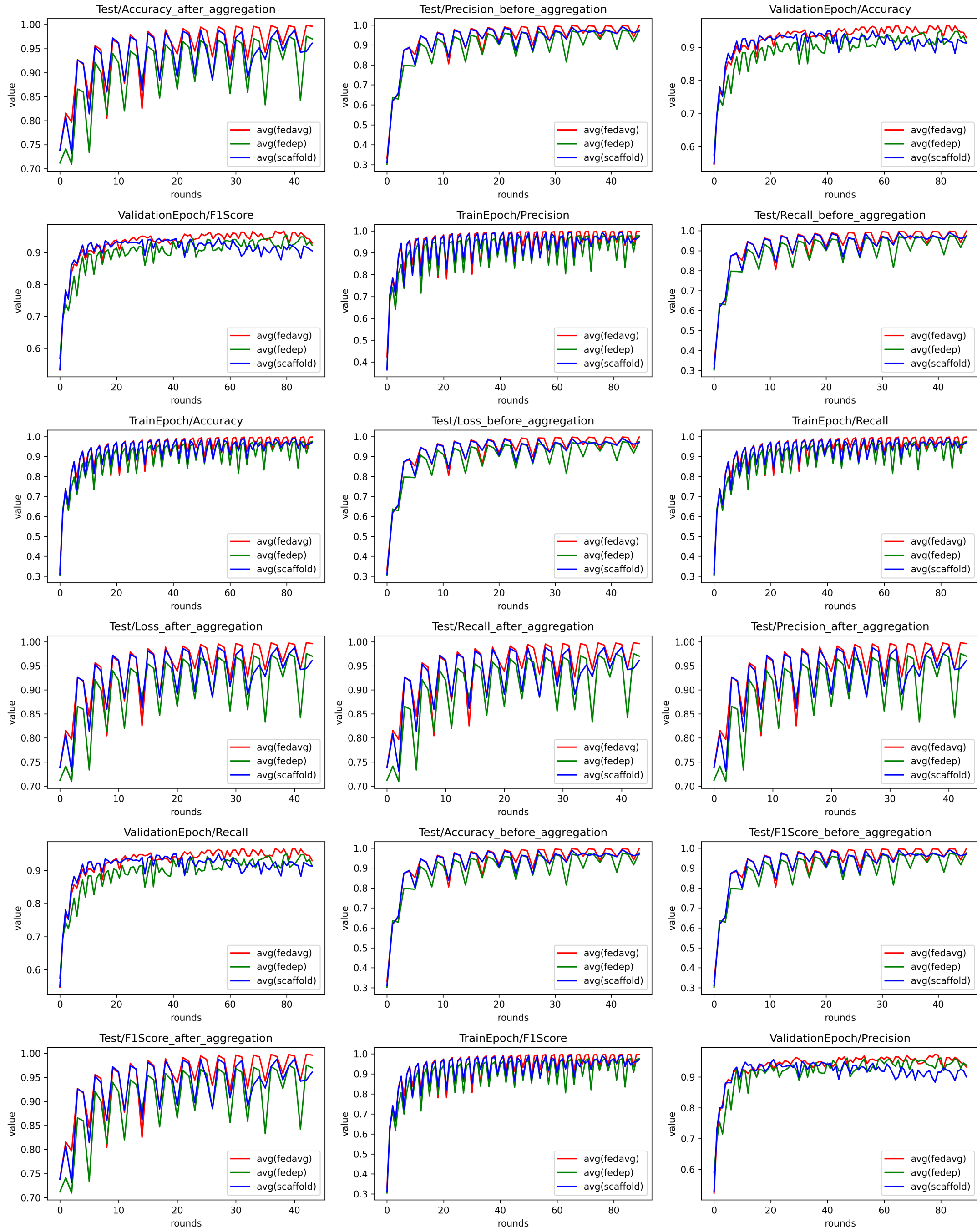

Figure A.4: Experiments with $\alpha_1 = 50(20\%)$, $\alpha_2 = 0.5(80\%)$, MNIST, 10 clients

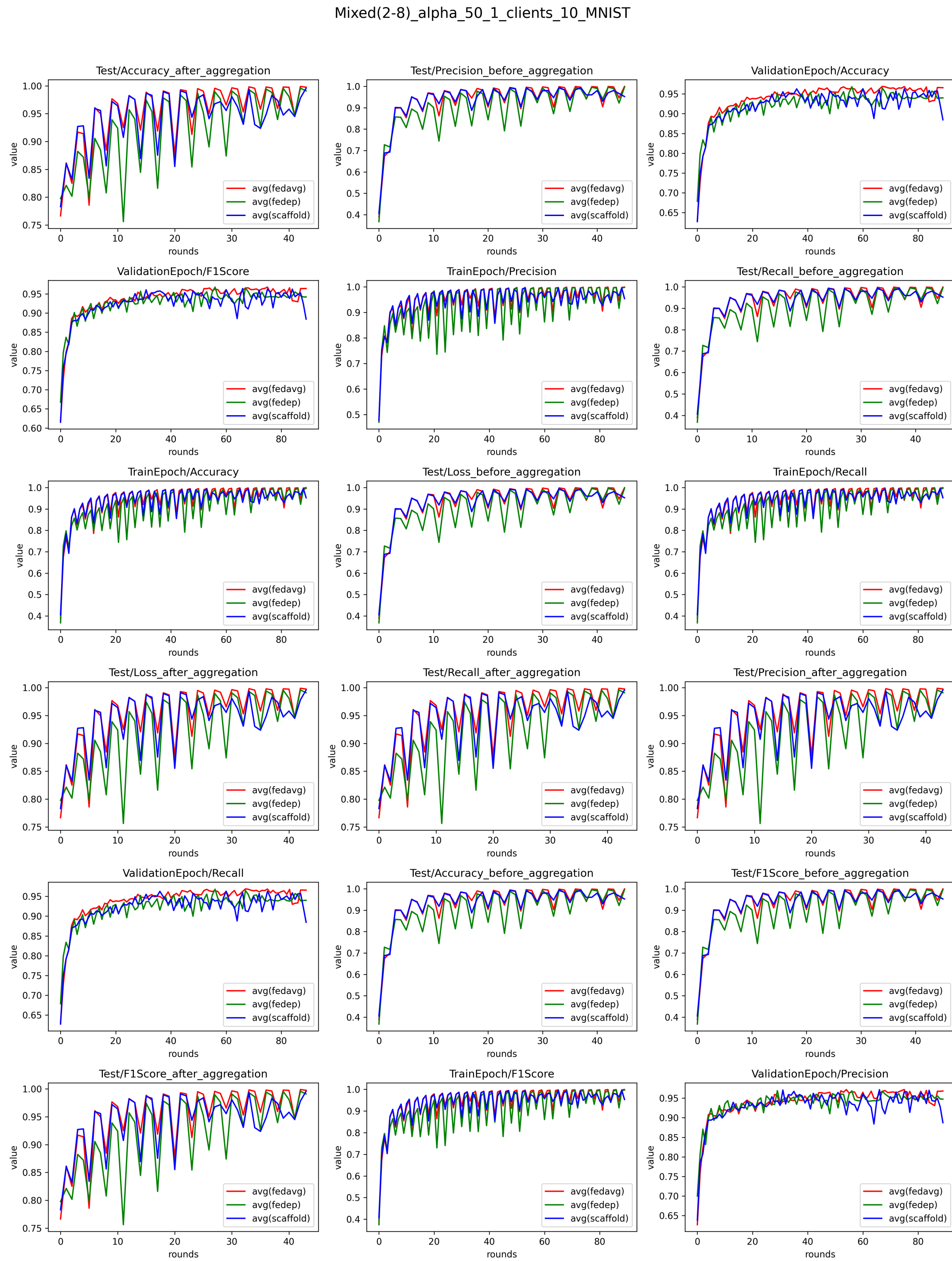

Figure A.5: Experiments with $\alpha_1 = 50(20\%)$, $\alpha_2 = 1(80\%)$, MNIST, 10 clients

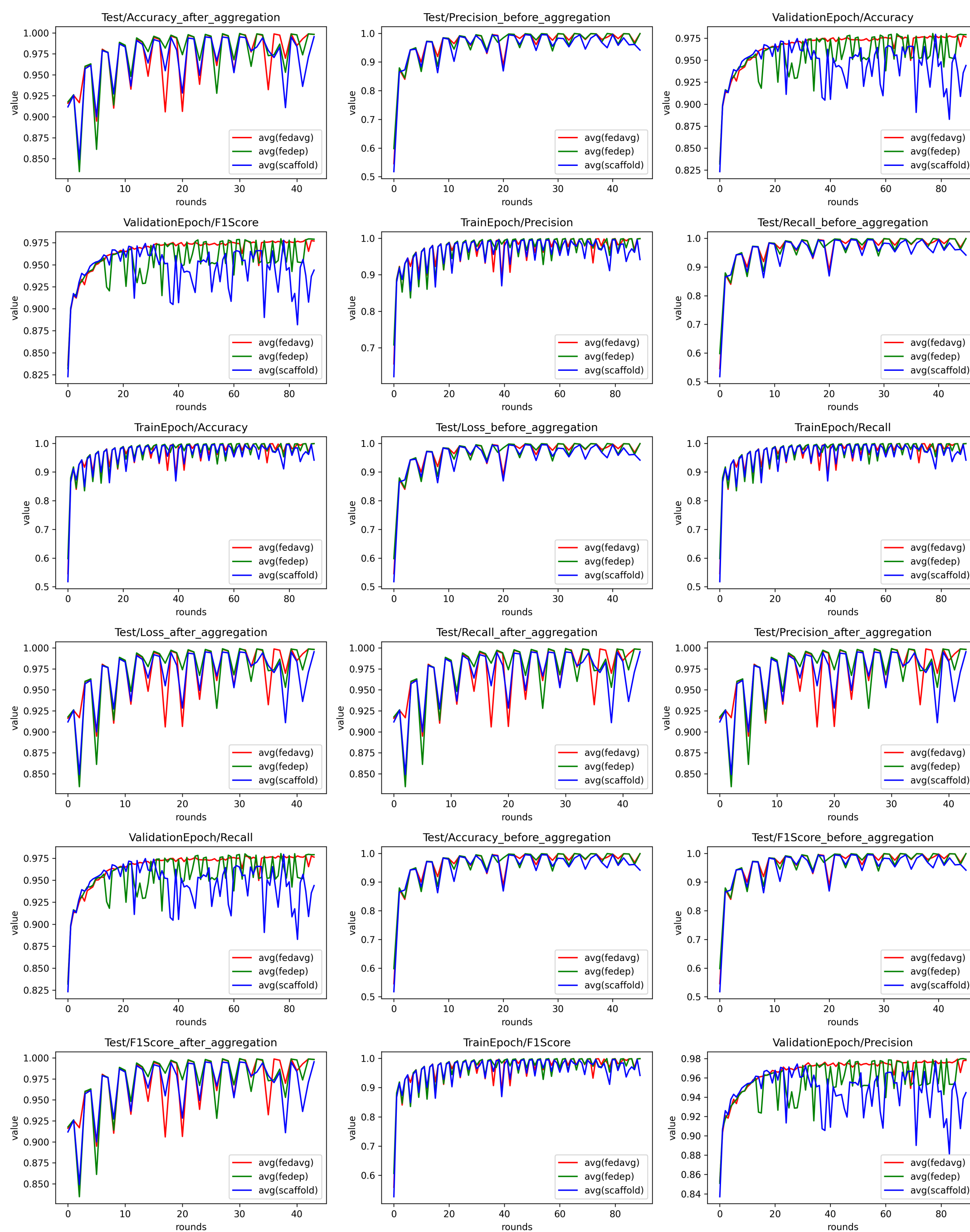

Figure A.6: Experiments with $\alpha_1 = 50(20\%)$, $\alpha_2 = 5(80\%)$, MNIST, 10 clients

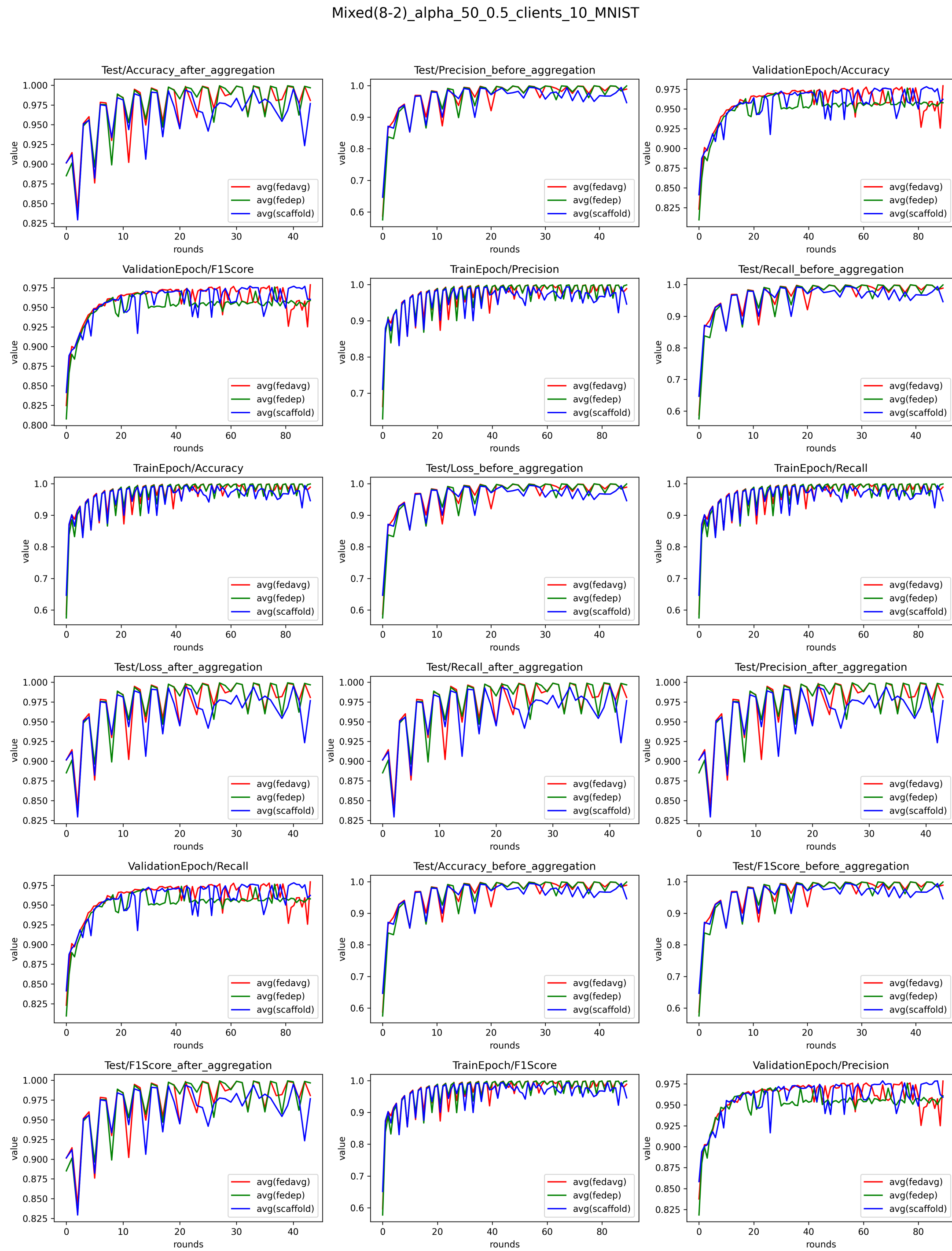

Figure A.7: Experiments with $\alpha_1 = 50(80\%)$, $\alpha_2 = 0.5(20\%)$, MNIST, 10 clients

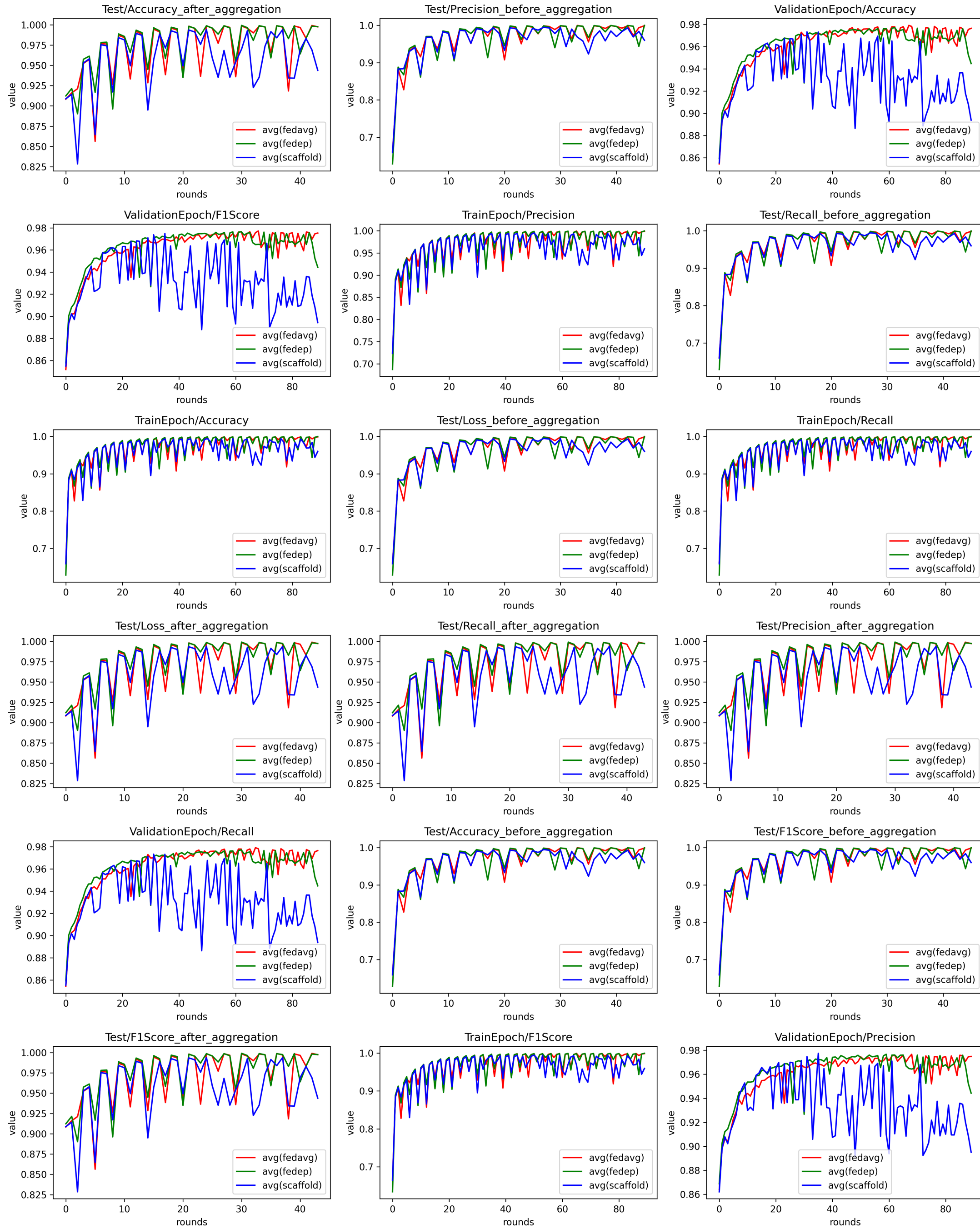

Figure A.8: Experiments with $\alpha_1 = 50(80\%)$, $\alpha_2 = 1(20\%)$, MNIST, 10 clients

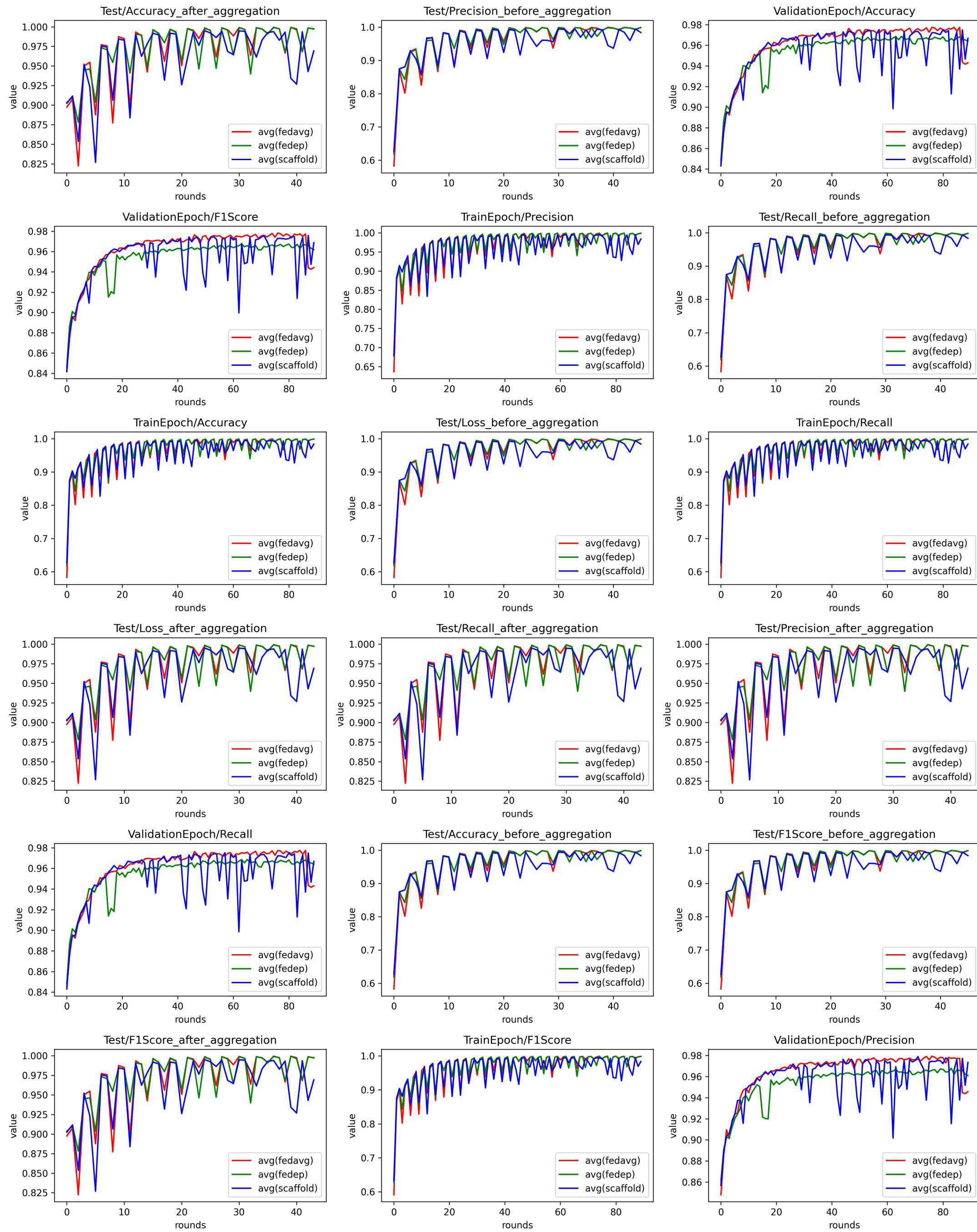

Figure A.9: Experiments with $\alpha_1 = 50(80\%)$, $\alpha_2 = 5(20\%)$, MNIST, 10 clients

Appendix B

Performance Comparison on FashionMNIST

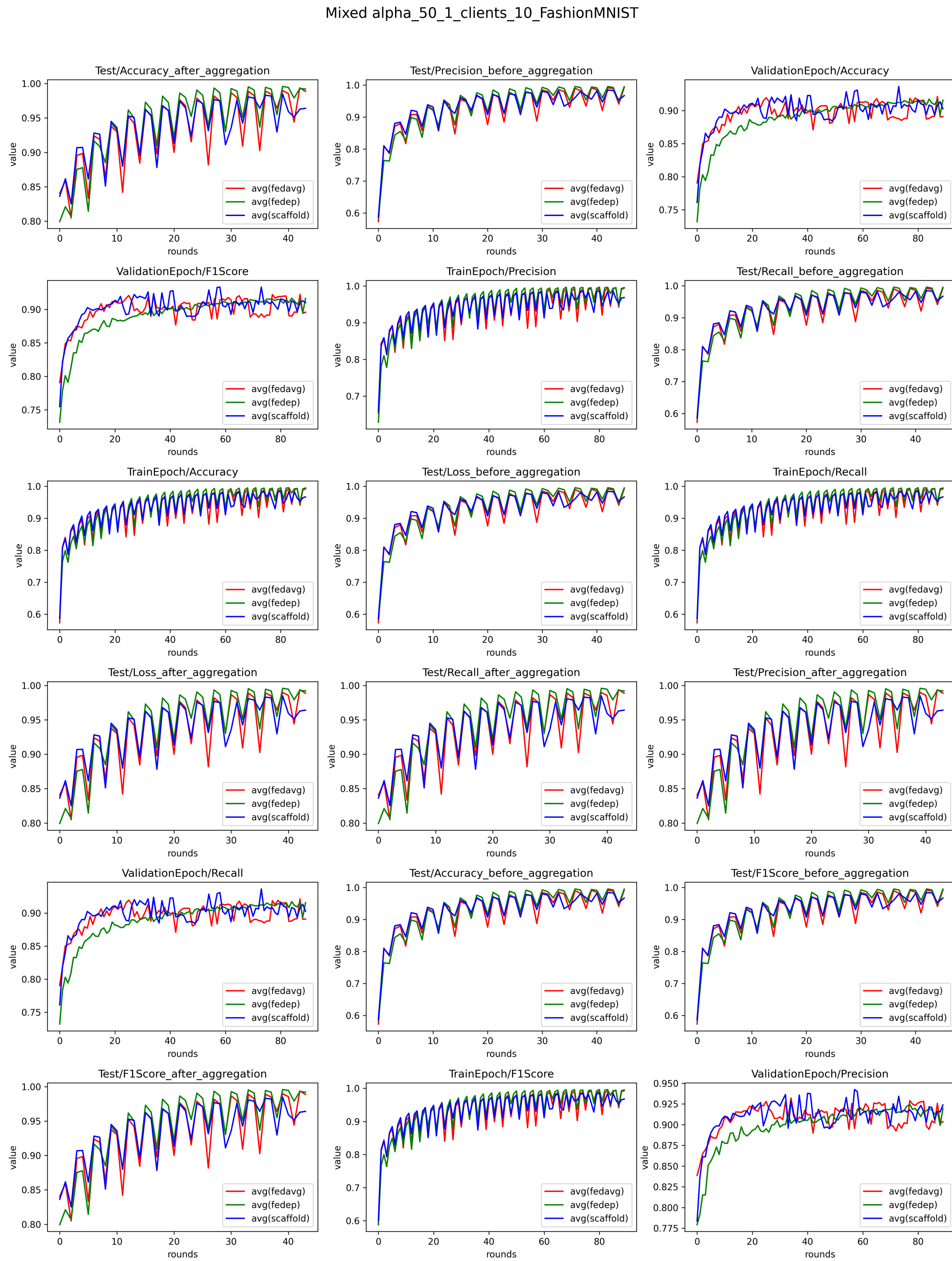

Figure B.1: Experiments with $\alpha_1 = 50(50\%)$, $\alpha_2 = 1(50\%)$, FashsionMNIST, 10 clients

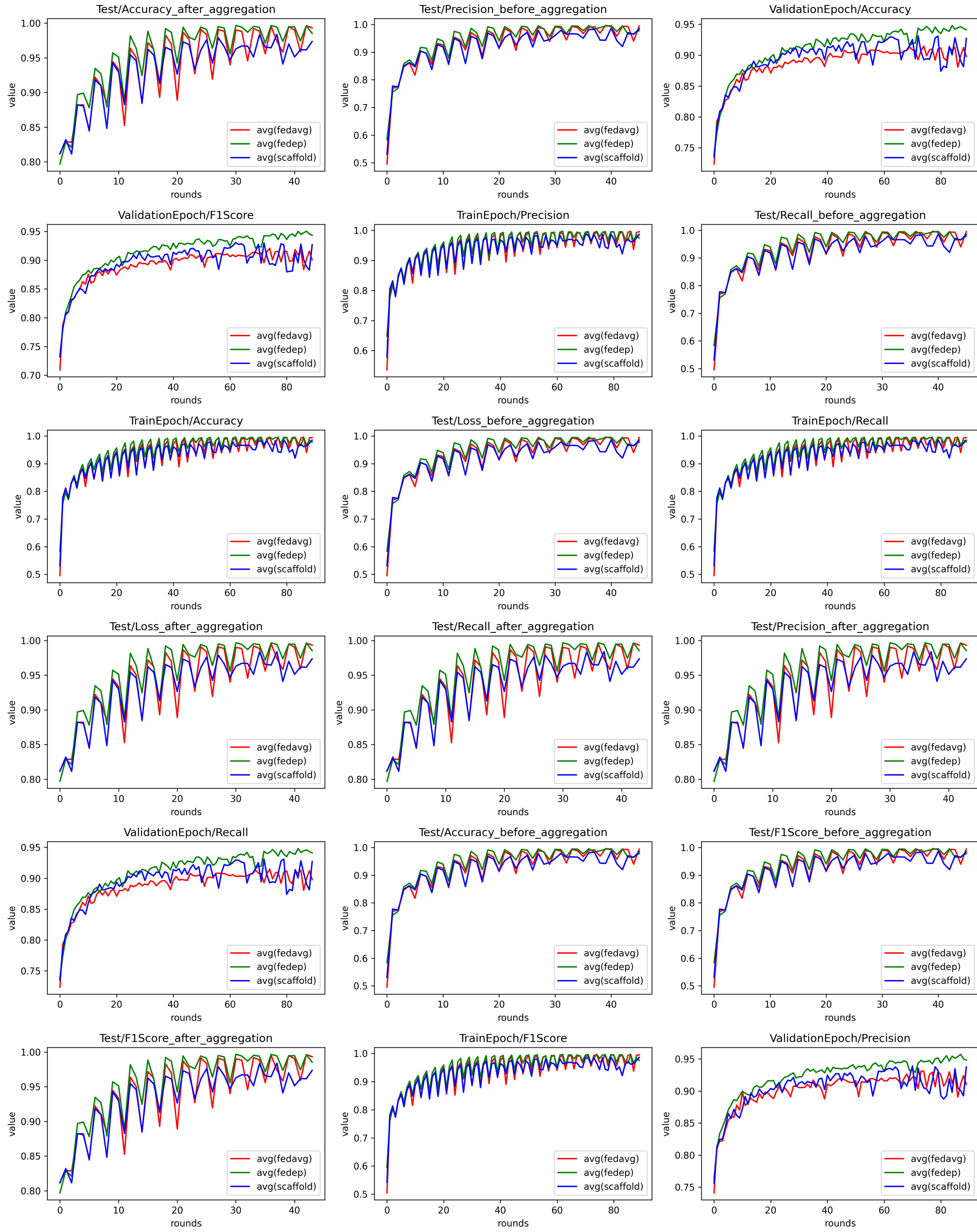

Figure B.2: Experiments with $\alpha_1 = 50(50\%)$, $\alpha_2 = 0.5(50\%)$, FashsionMNIST, 10 clients

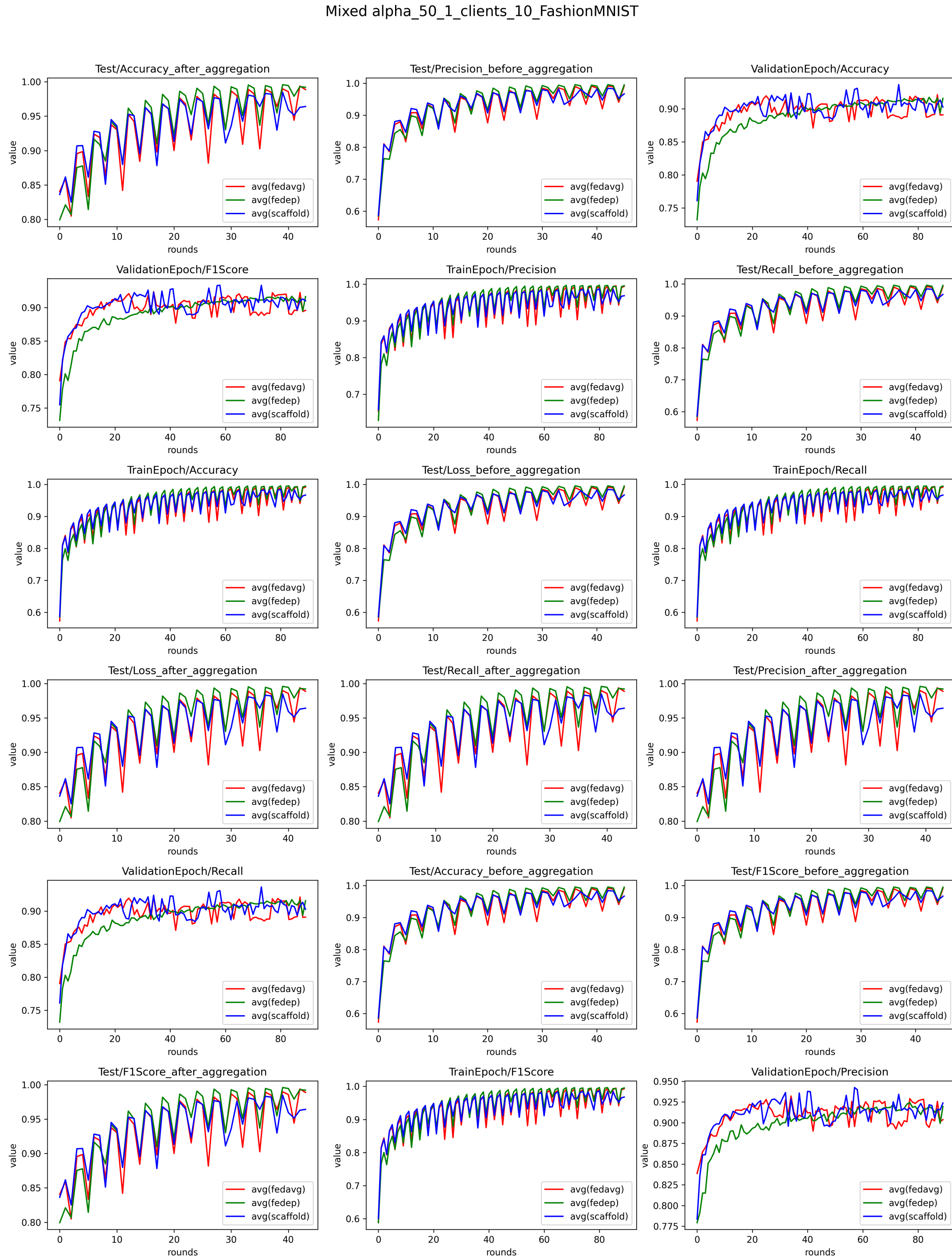

Figure B.3: Experiments with $\alpha_1 = 50(50\%)$, $\alpha_2 = 1(50\%)$, FashsionMNIST, 10 clients

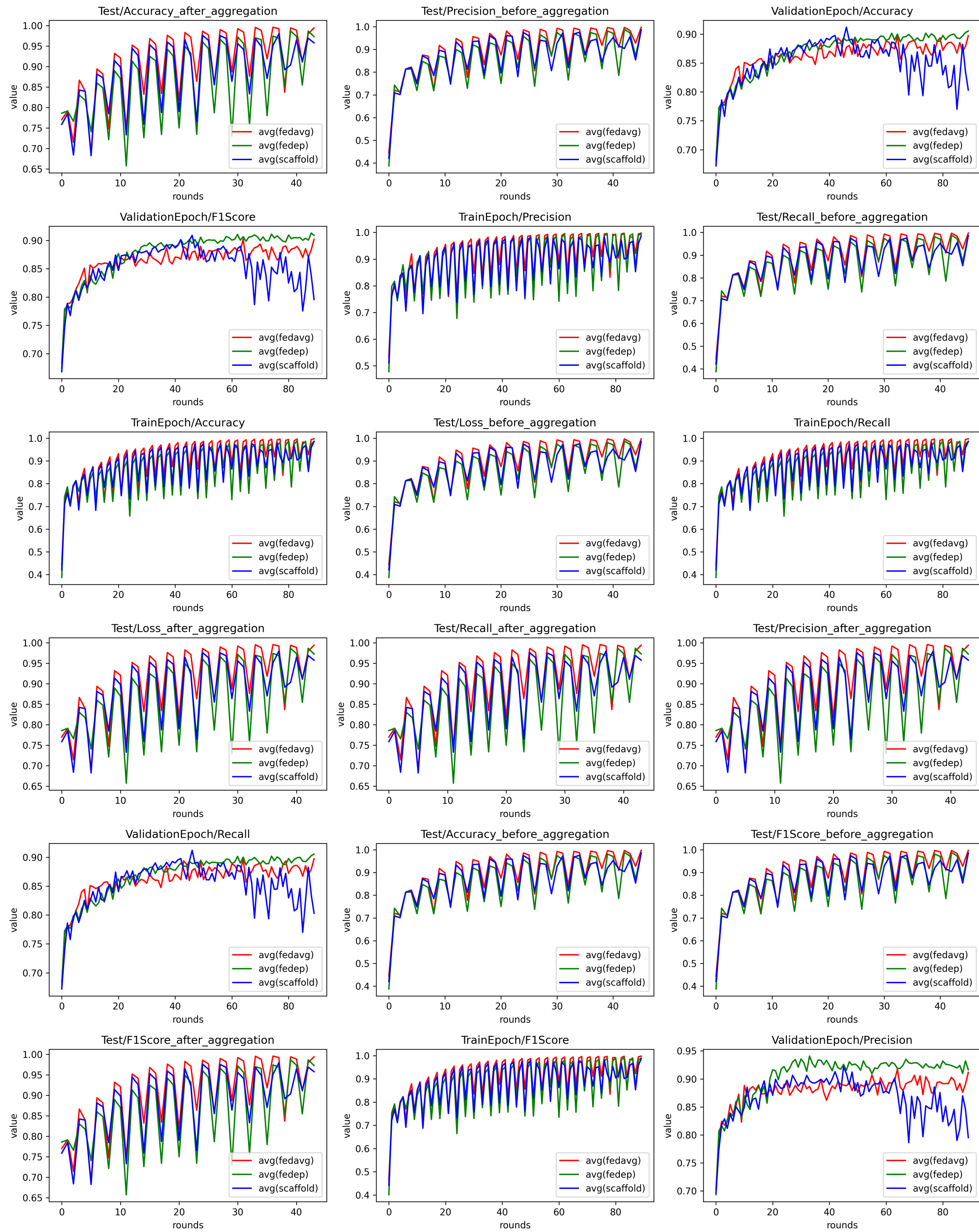

Figure B.4: Experiments with $\alpha_1 = 50(20\%)$, $\alpha_2 = 0.5(80\%)$, FashsionMNIST, 10 clients

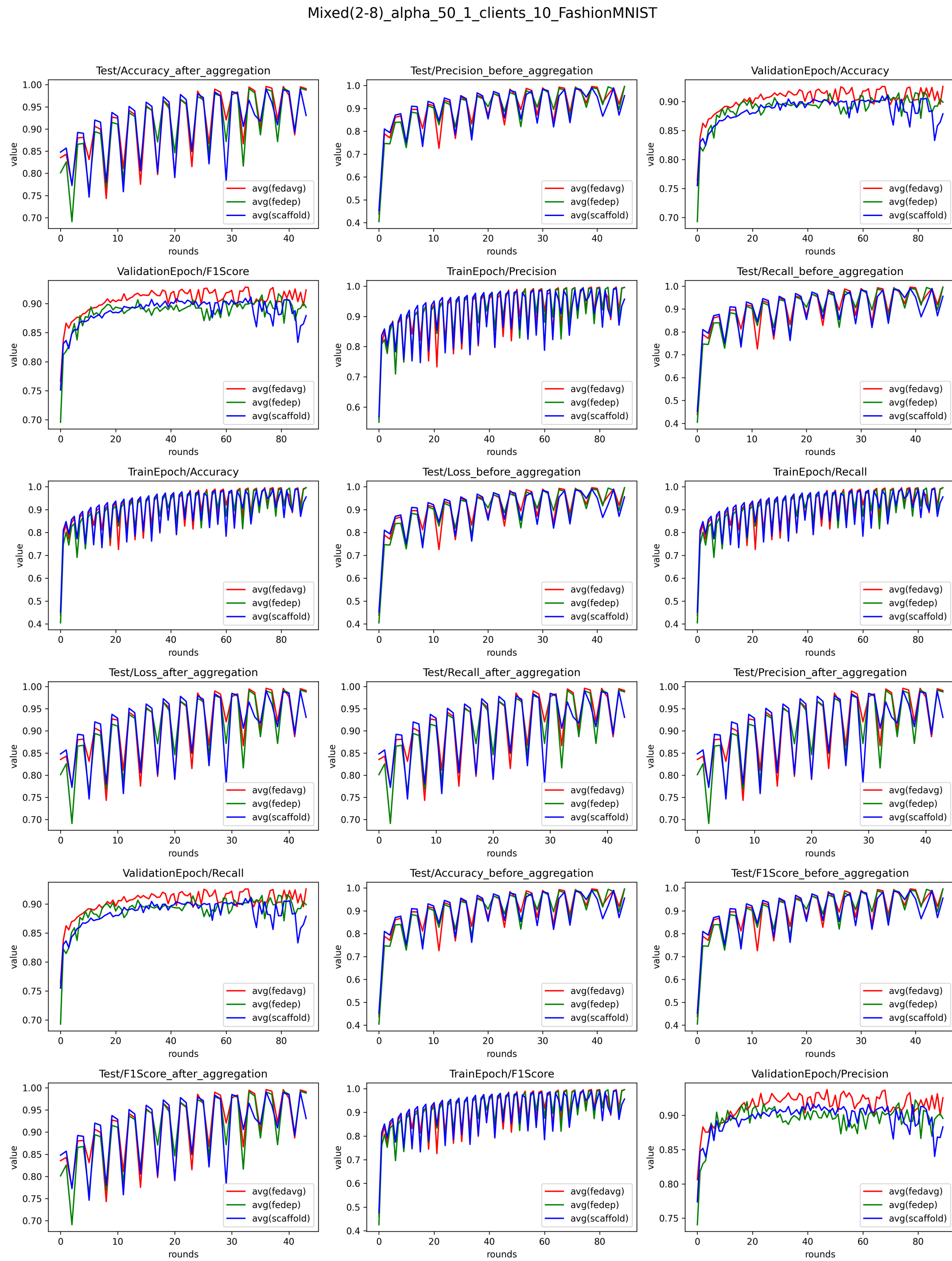

Figure B.5: Experiments with $\alpha_1 = 50(20\%)$, $\alpha_2 = 1(80\%)$, FashsionMNIST, 10 clients

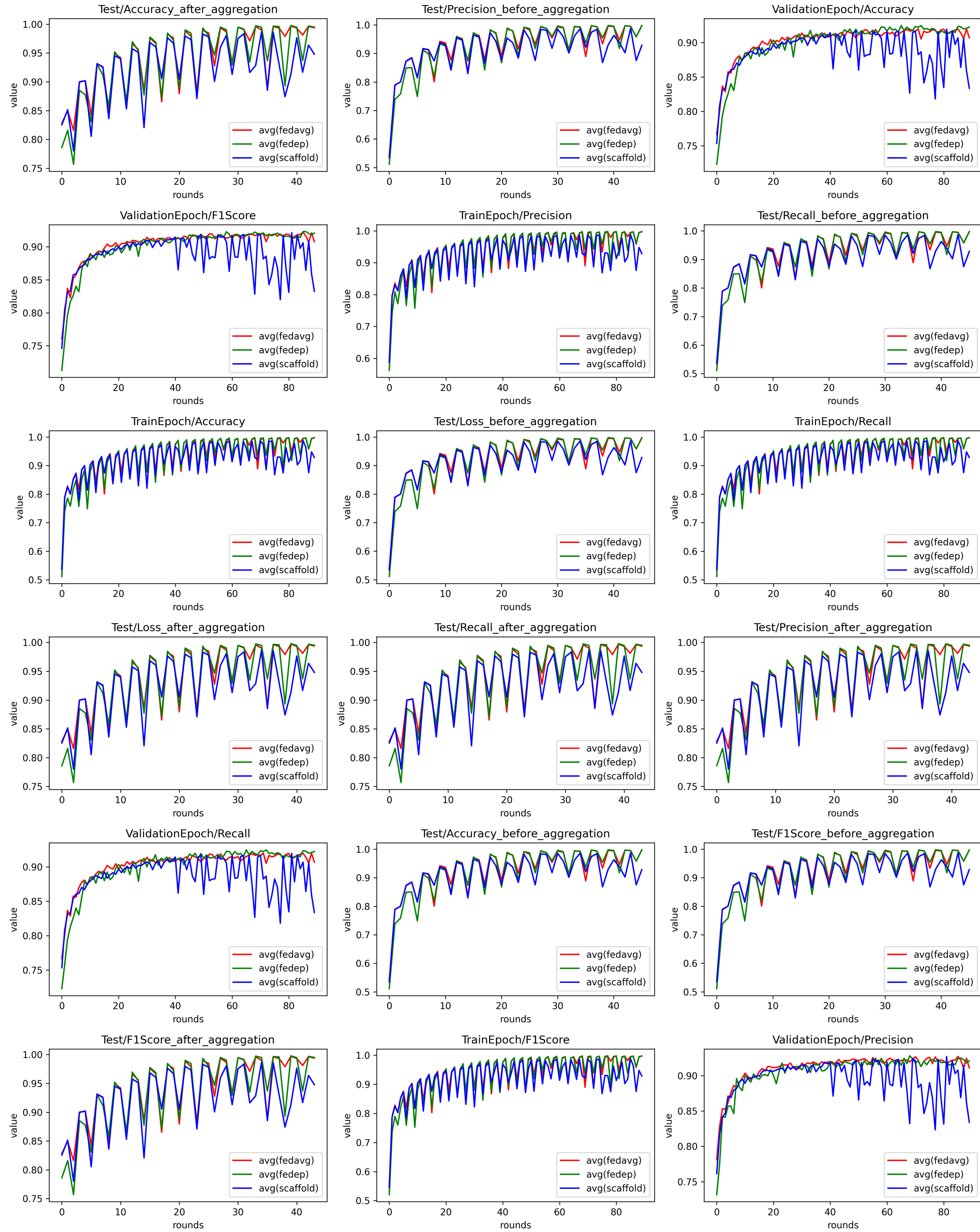

Figure B.6: Experiments with $\alpha_1 = 50(20\%)$, $\alpha_2 = 5(80\%)$, FashsionMNIST, 10 clients

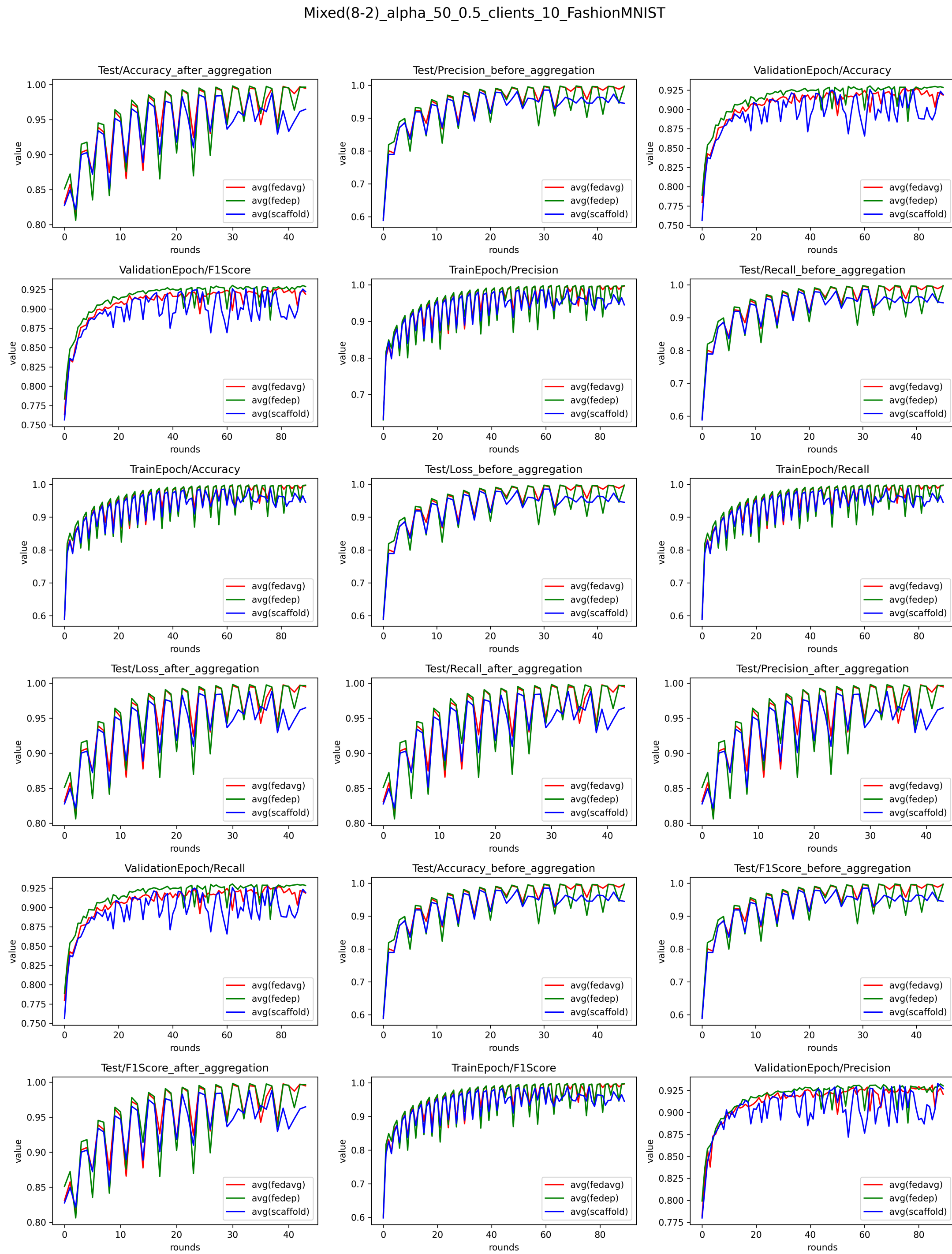

Figure B.7: Experiments with $\alpha_1 = 50(80\%)$, $\alpha_2 = 0.5(20\%)$, FashsionMNIST, 10 clients

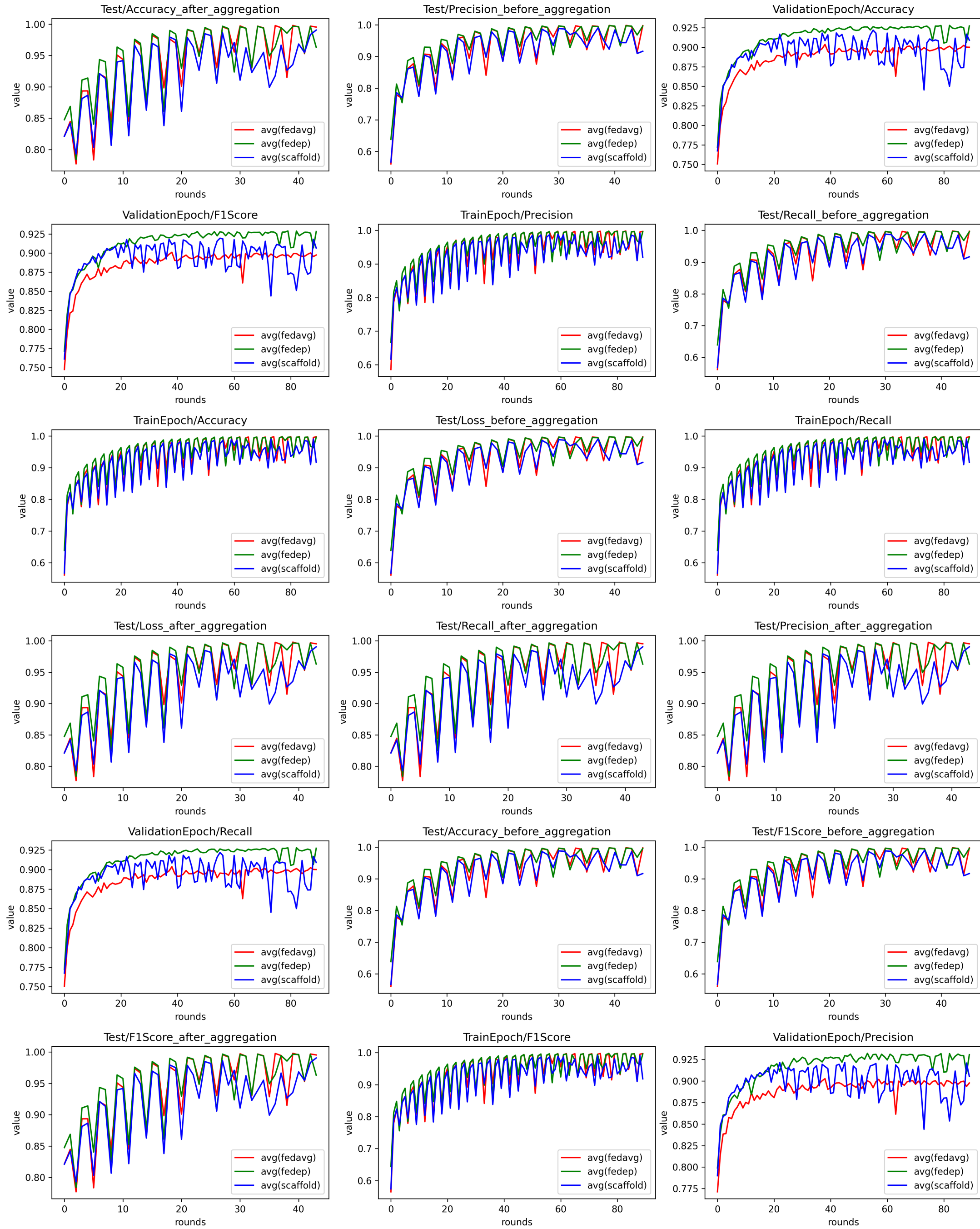

Figure B.8: Experiments with $\alpha_1 = 50(80\%)$, $\alpha_2 = 1(20\%)$, FashsionMNIST, 10 clients

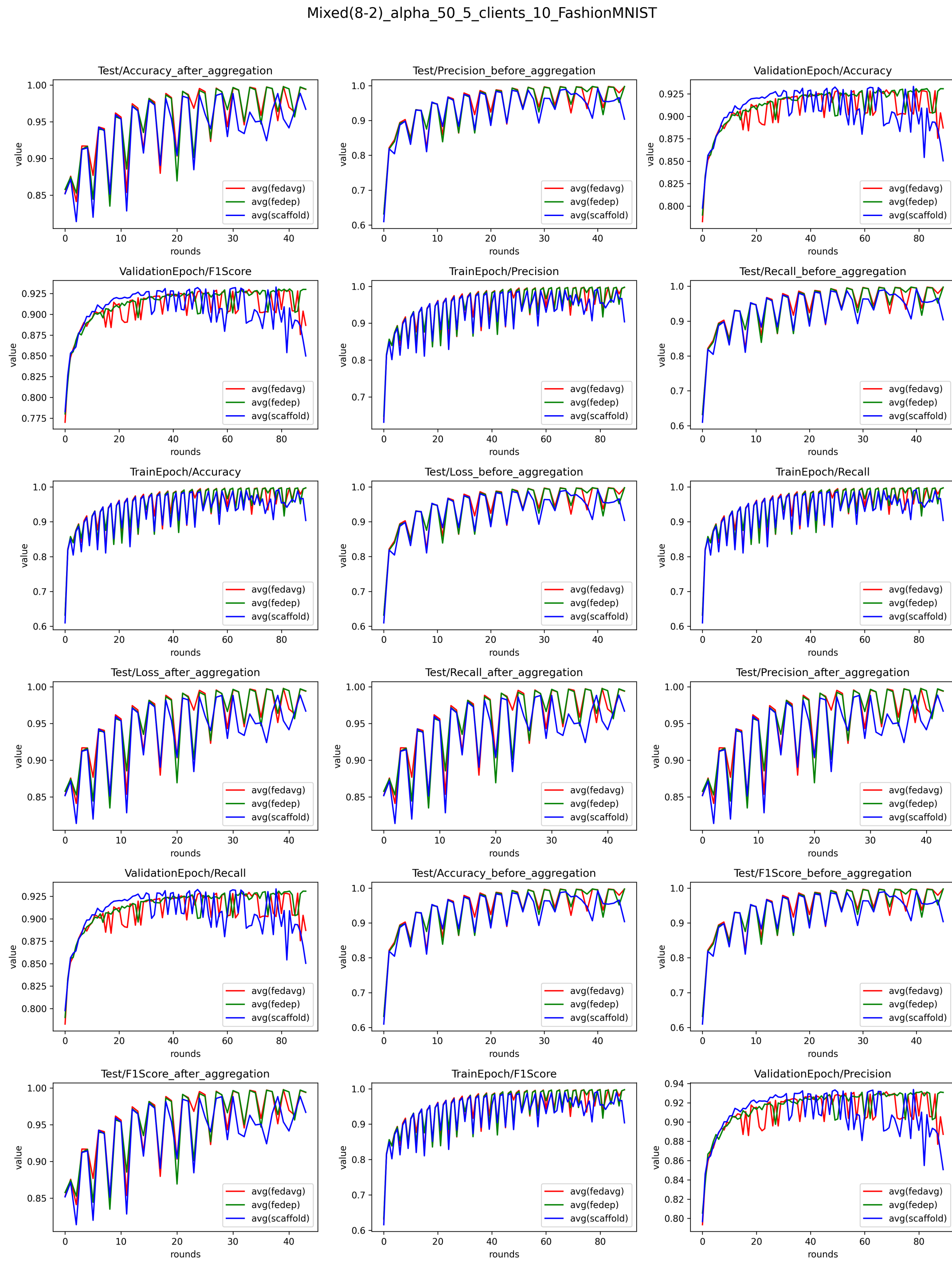

Figure B.9: Experiments with $\alpha_1 = 50(80\%)$, $\alpha_2 = 5(20\%)$, FashsionMNIST, 10 clients