Federated Learning Enhancement via Swarm Intelligence Algorithms

CS4099 Project Final Report

Submitted by

Mathew Bino	**B210435CS**
Alvin Gigo	**B210018CS**

Under the Guidance of
Dr. Nirmal Kumar Boran

Department of Computer Science and Engineering
National Institute of Technology Calicut
Calicut, Kerala, India - 673 601

April 15, 2025

NATIONAL INSTITUTE OF TECHNOLOGY
CALICUT, KERALA, INDIA - 673 601

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

2025

CERTIFICATE

Certified that this is a bonafide record of the project work titled

FEDERATED LEARNING ENHANCEMENT VIA SWARM INTELLIGENCE ALGORITHMS

done by

Mathew Bino
Alvin Gigo

of eighth semester B. Tech in partial fulfillment of the requirements for the award of the degree of Bachelor of Technology in Computer Science and Engineering of the National Institute of Technology Calicut

Project Guide	**Head of Department**
Dr. Nirmal Kumar	Dr. Subashini R
Assistant Professor	Associate Professor

DECLARATION

We hereby declare that the project titled, **Federated Learning Enhancement via Swarm Intelligence Algorithms**, is our own work and that, to the best of our knowledge and belief, it contains no material previously published or written by another person nor material which has been accepted for the award of any other degree or diploma of the university or any other institute of higher learning, except where due acknowledgement and reference has been made in the text.

Place : Calicut,Kerala,India	Signature :
Date : 5/5/2025	Name : Mathew Bino
	Reg. No. : B210435CS

Abstract

Federated Learning (FL) enables decentralized model training across distributed edge devices while preserving data privacy by eliminating the need for direct data sharing. This project investigates the performance of two advanced federated optimization algorithms: **Federated Particle Swarm Optimization (FedPSO)** and **Federated Constrained Particle Swarm Optimization (FedCPSO)**. We address critical challenges in FL, including reduced network transmission costs, minimized on-device memory requirements for model storage, and statistical heterogeneity through non-IID data distributions. Through extensive experiments, we evaluate the convergence behavior and accuracy of FedPSO and FedCPSO under varying conditions, comparing them against baseline approaches such as Federated Averaging (FedAvg). Our results show that FedCPSO effectively handles non-IID data by adaptively balancing client contributions during aggregation, significantly outperforming standard approaches. Furthermore, we analyze the impact of optimization techniques on reducing communication overhead and shrinking model footprints while maintaining robustness, particularly in resource-constrained environments. Our results demonstrate significant reductions in bandwidth usage (51% lower) and local memory demands (70% smaller models), with only marginal trade-offs in accuracy , offering practical insights for deploying FL on edge devices with limited computational resources. The implementation leverages the `Flower` framework and is validated on benchmark datasets (e.g., CIFAR-10, MNIST), demonstrating the efficacy of the methods.

ACKNOWLEDGEMENT

Contents

List of Figures

List of Tables

Chapter 1

Introduction

- Machine learning models traditionally rely on centralized training where data is collected and processed in a single location.They use techniques such as Stochastic Gradient Descent or ADAM for training

- Federated Learning (FL) has emerged as a decentralized training paradigm where models are trained locally on edge devices without transferring raw data to a central server.Instead they transfer learned weight matrices which are then processed at the central server

- Federated Averaging (FedAvg) is a fundamental algorithm in Federated Learning that aggregates locally trained model updates from clients to update a global model.

- Flower is an open-source Federated Learning (FL) framework designed to enable flexible and scalable FL research and deployment across various machine learning libraries such as PyTorch, TensorFlow, and Scikit-learn.In this project, Flower is used to implement and test our FL models

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

```
Server executes:
1: Initialize $w_0$
2: for each round $t = 1, 2, \dots$ do
3:     $m \leftarrow \max(C \cdot K, 1)$
4:     $S_t \leftarrow$ (random set of $m$ clients)
5:     for each client $k \in S_t$ in parallel do
6:         $w^k_{t+1} \leftarrow$ ClientUpdate$(k, w_t)$
7:     $m_t \leftarrow \sum_{k \in S_t} n_k$
8:     $w_{t+1} \leftarrow \sum_{k \in S_t} \frac{n_k}{m_t} w^k_{t+1}$

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

1.1 Federated Learning Workflow in Flower

The Flower framework follows a structured workflow to facilitate Federated Learning (FL). The key steps involved are:

1. **Client Selection:** The central server selects a subset of available clients (edge devices) to participate in the current training round.
2. **Local Training:** Each selected client trains the model locally using its private dataset and a predefined optimization algorithm
3. **Model Update Transmission:** Once local training is completed, clients send their updated model parameters to the central server.
4. **Aggregation:** The server aggregates the received updates using a specified aggregation strategy such as FedAvg or a custom ap-

proach.

5. **Global Model Update:** The aggregated updates are applied to the global model, which is then shared with the clients for the next training round.
6. **Iteration Until Convergence:** Steps 1–5 are repeated for multiple rounds until the model achieves satisfactory performance.

- Another key concept extensively utilized in this project is the Particle Swarm Optimization (PSO) algorithm.

1.2 Particle Swarm Optimization (PSO)

1.2.1 How PSO Works

PSO maintains a swarm of particles, where each particle represents a candidate solution. The position of each particle is updated based on:

- **Personal Best (pBest):** The best solution found by the individual particle so far.
- **Global Best (gBest):** The best solution discovered by the entire swarm.

Each particle adjusts its velocity and position by balancing exploration (global search) and exploitation (local search) to converge toward an optimal solution.

1.2.2 PSO in This Project

In this project, PSO is integrated into Federated Learning as a supportive optimization technique to enhance model training. By incorporating PSO, we aim to:

  - Reduce the computational burden on edge devices by optimizing certain aspects of the training process.
  - Improve training efficiency without completely replacing gradient-based optimization.
  - Enhance overall model performance by refining weight updates

1.2.3 Combined Particle Swarm Optimization

Combined Particle Swarm Optimization (CPSO) is an enhanced variant of the standard PSO algorithm that introduces an additional factor for the best neighbor. This modification improves performance, particularly in scenarios with non-Independent and Identically Distributed (non-IID) data, which is commonly encountered in real-world applications.

- To make improvements in network cost, computation, and memory efficiency, we have utilized the concepts of magnitude pruning,quantization and compression techniques

1.3 Optimization Techniques

1.3.1 Magnitude Pruning

Magnitude pruning is a technique used to reduce the size of a machine learning model by removing weights with the smallest absolute values. Since these weights contribute the least to the model's performance, their removal leads to a more compact representation without significantly affecting accuracy. This approach helps in reducing memory consumption and computational requirements, making it well-suited for resource-constrained edge devices in Federated Learning.

1.3.2 Quantization

Quantization is a model compression technique that reduces the precision of numerical representations in machine learning models, typically converting high-precision floating-point values into lower-precision formats such as int8 or float16. By doing so, quantization decreases the model's memory footprint and speeds up inference while maintaining acceptable accuracy levels. This is particularly beneficial for deploying models on edge devices where memory is constrained

1.3.3 LZMA Compression

To mitigate communication bottlenecks inherent in federated systems, we have integrated LZMA (Lempel-Ziv-Markov chain Algorithm) lossless compression for model parameter transmission.

Chapter 2

Literature Survey

2.1 *Communication-efficient learning of deep networks from decentralized data*

Authors :Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, Blaise Aguera y Arcas

The seminal work in this paper established the framework for federated learning (FL) as a solution to growing privacy concerns in mobile computing. Modern mobile devices possess powerful sensors (cameras, microphones, GPS) that generate unprecedented volumes of private data [1]. While this data could significantly improve machine learning models (e.g., for speech recognition or image curation), its sensitive nature often prohibits centralized collection due to privacy regulations and security risks.

2.1.1 Core Methodology

This paper proposed a practical approach centered on **iterative model averaging**, where:

- A fixed set of K clients maintain local datasets

- Each round selects a random fraction C of clients for efficiency
- Clients receive global model parameters, perform local updates, and return only model deltas

This design provides two key advantages:

1. **Decoupled training**: Model improvement without raw data access
2. **Risk reduction**: Limits attack surfaces to devices rather than cloud servers

2.1.2 Empirical Contributions

The authors made three primary contributions:

- Identified decentralized mobile data training as a critical research direction
- Introduced FedAvg as a baseline algorithm balancing accuracy and communication costs
- Demonstrated through extensive experiments that:
  - Client sampling ($C < 1$) maintains performance while reducing overhead
  - Model averaging converges despite non-IID data distributions

Table 2.1: Key Properties of Federated Learning

Characteristic	**Benefit**
Decentralized Data	Preserves privacy
Partial Participation	Reduces communication
Model Averaging	Maintains accuracy

The work established FL as a viable alternative to centralized training, particularly for applications handling sensitive user data. Subsequent research has built upon these foundations to address limitations in convergence speed and heterogeneous data handling.

2.2 *Flower: A friendly federated learning research framework*

Authors: Daniel J. Beutel, Taner Topal, Akhil Mathur, Xinchi Qiu, Javier Fernandez-Marques, Yan Gao, Lorenzo Sani, Kwing Hei Li, Titouan Parcollet, Pedro Porto Buarque de Gusmão, Nicholas D. Lane

Current federated learning research faces significant systems bottlenecks, particularly the lack of frameworks supporting scalable execution on real mobile and edge devices. While closed production systems routinely handle thousands to millions of clients, academic research remains limited to simulations with typically fewer than 100 virtual clients. Even studies employing larger client populations rely entirely on software simulations rather than actual device deployments . Existing frameworks like TensorFlow Federated (TFF) and LEAF provide valuable experimentation platforms but critically lack three capabilities:

- Support for heterogeneous mobile and edge hardware
- Tools to evaluate real-world systems challenges
- Transition paths from simulation to production

2.2.1 Flower Framework Design Principles

The Flower framework addresses these limitations through several key design choices:

Core Architecture

The server-side implementation comprises three fundamental components:

1. **ClientManager**: Maintains the set of available clients through `ClientProxy` objects, each abstracting communication with physical devices
2. **Federated Learning Loop**: Orchestrates the training workflow across participation rounds
3. **Strategy**: User-customizable component implementing the aggregation logic (e.g., FedAvg, FedProx)

This architecture explicitly decouples the server from client implementation details, enabling:

- Cross-platform compatibility (Android, iOS, embedded systems)
- Mixed workloads combining different ML frameworks
- Heterogeneous communication protocols

Realistic Simulation Capabilities

Flower provides built-in tools to emulate challenging real-world conditions:

- Device-specific resource constraints (CPU, memory)
- Partial client availability patterns

These simulations operate within cloud environments while maintaining fidelity to edge device behaviors.

2.2.2 Key Innovations

The paper makes three primary contributions:

- **Unified Abstraction**: A framework-agnostic interface supporting PyTorch, TensorFlow, and other ML libraries without modification to existing training pipelines
- **Production Transition**: Capability to deploy the same codebase from small-scale simulations to large-scale device deployments
- **Systems-Aware Evaluation**: Tools to measure both algorithmic convergence and systems metrics (communication costs, compute latency) under realistic conditions

2.2.3 Comparative Advantages

Unlike TFF and LEAF, Flower explicitly bridges the gap between research and production by:

- Supporting actual mobile clients rather than just simulations
- Maintaining compatibility with diverse ML frameworks
- Providing instrumentation for both accuracy and systems performance

The framework's distinctive client-agnostic server design enables this flexibility while preserving research reproducibility.

2.3 *Particle swarm optimization*

Authors:Kennedy, James and Eberhart, Russell

Particle Swarm Optimization (PSO) provides an effective method for optimizing continuous nonlinear functions, particularly in neural network training . The algorithm has dual theoretical foundations:

- **Swarm Intelligence**: Inspired by biological phenomena like bird flocking and fish schooling
- **Evolutionary Computation**: Shares concepts with genetic algorithms but with simpler operations

PSO offers three key advantages for neural network training:

1. **Computational Efficiency**: Requires only basic mathematical operators
2. **Low Memory Usage**: Maintains minimal state (position and velocity vectors)
3. **Empirical Effectiveness**: Demonstrated success across diverse problem domains

Early applications to neural networks showed PSO could effectively train weights without gradient calculations.Intriguing informal indications are that the weights found by particle swarms sometimes generalize from a training set to a test set better than solutions found by gradient descent.

2.4 *FedPSO: Federated learning using particle swarm optimization to reduce communication costs*

Authors:Park, Sunghwan and Suh, Yeryoung and Lee, Jaewoo

Federated Particle Swarm Optimization (FedPSO) is a novel approach introduced to tackle one of the core limitations of federated learning: high communication cost. In traditional artificial neural network (ANN) models, computation time is often the bottleneck. To address this, various solutions such as GPU acceleration and distributed training across multiple GPUs have been explored. However, in the federated learning paradigm, communication overhead significantly outweighs computation time, making it a major efficiency bottleneck.

FedPSO is one of the first methods to explicitly focus on reducing this communication overhead and also integrates Particle Swarm Optimization (PSO) into the federated learning process. Unlike traditional methods such as FedAvg, which aggregate model weights from all participating clients, FedPSO collects only performance metrics—such as accuracy and loss—from each client. These metrics are significantly smaller in size (e.g., loss values are just 4 bytes), drastically reducing the amount of data transmitted during training rounds.

Algorithm 2 FedPSO. The K clients are indexed by k; B is the local minibatch size, E is the number of local epochs, and η is the learning rate.

```
Server executes:
 1: Initialize $w_0$, pbest, gbest, gid
 2: for each round $t = 1, 2, \dots$ do
 3:     $m \leftarrow \max(C \cdot K, 1)$
 4:     $S_t \leftarrow$ (random set of $m$ clients)
 5:     for each client $k \in S_t$ in parallel do
 6:         pbest $\leftarrow$ ClientUpdate($k$, $w_t^{gid}$)
 7:         if gbest > pbest then
 8:             gbest $\leftarrow$ pbest
 9:             gid $\leftarrow$ k
10:     $w_{t+1} \leftarrow$ GetBestModel(gid)

ClientUpdate($k, w_t^{gid}$) :    // Run on client k
 1: Initialize $V$, $w$, $w^{pbest}$, $a$, $c_1$, $c_2$
 2: $B \leftarrow$ (split $\mathcal{P}_k$ into batches of size $B$)
 3: for each weight layer $l = 1, 2, \dots$ do
 4:     $V_l \leftarrow a \cdot V_l + c_1 \cdot \text{rand} \cdot (w^{pbest} - w_l) + c_2 \cdot \text{rand} \cdot (w_t^{gbest} - w_l)$
 5:     $w_l \leftarrow w_l + V_l$
 6: for each local epoch $i$ from 1 to $E$ do
 7:     for batch $b \in B$ do
 8:         $w \leftarrow w - \eta \nabla \ell(w; b)$
 9: Return pbest to server

GetBestModel($gid$) :    // Fetch best model
 1: request to Client(gid)
 2: Return $w$ to server
```

The approach maintains a global model by selecting and updating weights only from the client that reports the best performance score, determined using PSO constructs like pbest (personal best) and gbest (global best).

Experimental evaluations of FedPSO demonstrate its effectiveness both in terms of communication cost and model performance. The proposed method achieves an average accuracy improvement of 9.47% over baseline models and shows a 4% improvement in maintaining accuracy under unstable network conditions. These results highlight FedPSO as a communication-efficient and performance-improving alternative to existing federated learning algorithms, especially in real-world settings with network constraints.

2.5 *FedCPSO: Federated Learning with Combined Particle Swarm Optimization*

Authors:Shi, Hongjian and Ma, Ruhui and Guan, Haibing and Zhang, Weishan

In IoT scenarios, statistical heterogeneity prevents the global model from obtaining good accuracy over the local datasets. Statistical heterogeneity exists when loT devices have different kinds of data distribution, which means that the model trained on one client might not be suitable for the dataset on the other client. Thus researchers turn to personalized FL (pFL) algorithms. The servers in pFL algorithms maintain different models for different clients to fit the local dataset and increase local accuracy. However, pFL methods encounter difficulty separating global and local knowledge , which has the potential for better local accuracy. Furthermore, the communication volume of pFL methods is significant as the communication ability for IoT devices is not always sufficient.

This paper proposes Federated Learning with Com-bined Particle Swarm Optimization (FedCPSO) to handle the statistical heterogeneity and large communication volume. Motivated by PSO, the aggregation process on the server is considered as a multi-agent optimization problem and use PSO to solve it. Moreover,velocity in PSO is designed specifically for FL algorithms, using the best global, client, and neighbor models.Theoretical magnitude pruning is also implemented to reduce communication volume. The experimental results indicate that FedCPSO can reduce up to 50% communication volume.

Furthermore, one previous work adopts PSO to optimize the aggregation process in FL and replaces the aggregation step in FedAvg with PSO. In each round, the server receives the losses from the clients, and only the client with the lowest loss has to upload the entire model. Such a method can reduce the communication volume while slightly improving the accuracy. However, such a method cannot handle statistical heterogeneity as the losses from the clients cannot be used to determine the best model.

The experimental results show that FedCPSO can theoretically reduce up to 50% communication volume and introduce only less than a 2% accuracy drop compared to SOTA pFL algorithms. Such results demonstrate the priority of FedCPSO against statistical heterogeneity and communication overhead.

Algorithm 3 FedCPSO Server-Side Execution

```
    Server executes:                                                   ▷ Run on server
 1: Initialize $\Theta_0$, $\Theta_{best} = \Theta_0$, $A_{best} = 0$, $A^i_{best} = 0$, $p_{i,j} = 1$, and $R^i_t = 0$ for $\forall i, j$
 2: for each round $t = 1, 2, \dots$ do
                                                                       ▷ Local training
 3:     for each client $C_i$ in parallel do
 4:         $\hat{\Theta}^i_t, \hat{A}^i_t \leftarrow$ ClientUpdate($\Theta^i_t$)
                                                                       ▷ Compute best global model
 5:     Aggregate client models to obtain $\Theta_t$
 6:     if avg($\hat{A}^i_t$) $> A_{best}$ then
 7:         $A_{best} =$ avg($\hat{A}^i_t$) and $\Theta_{best} = \Theta_t$
                                                                       ▷ Compute best client model
 8:     for each client model $\hat{\Theta}^i_t$ do
 9:         if $\hat{A}^i_t > A^i_{best}$ then
10:             $A^i_{best} = \hat{A}^i_t$ and $\Theta^i_{best} = \Theta^i_t$
                                                                       ▷ Compute best neighbor model
11:     for each client model $\hat{\Theta}^i_t$ do
12:         if $\hat{A}^i_t < A^i_{t-1}$ then
13:             Update $P_i, R^i_t$ according to 2.2
14:             Update $R^i_t$ according to 2.3
15:             $\Theta^i_t = \Theta^{R^i_t}_t$
                                                                       ▷ Update client model using CPSO
16:     for each client model $\hat{\Theta}^i_t$ do
17:         Update $v^i_t$ according to 2.1
18:         $\Theta^i_{t+1} = \hat{\Theta}^i_t + v^i_t$
19: return $\Theta_T$
```

Algorithm 4 ClientUpdate (Θ^i_t)

```
    ClientUpdate ($\Theta^i_t$):                                       ▷ Local training on client $C_i$
 1: for each local epoch do
 2:     for each data batch $B$ do
 3:         $\Theta^i_t \leftarrow \Theta^i_t - \alpha\nabla\mathcal{L}(\Theta^i_t, B)$
 4: Prune $\Theta^i_t$ to obtain $\hat{\Theta}^i_t$
 5: Using local test dataset to obtain $\hat{A}^i_t$ with $\hat{\Theta}^i_t$
 6: return $\hat{\Theta}^i_t, \hat{A}^i_t$
```

$$v_t^i = w \cdot v_{t-1}^i + (1-w) \cdot \Big[c_0 \cdot (\Theta_{best} - \Theta_t^i) + c_1 \cdot (\Theta_{best}^i - \Theta_t^i) + c_2 \cdot (\Theta_t'^i - \Theta_t^i)\Big] \tag{2.1}$$

$$P_{i,j} = p_{i,j} \cdot A_t^i \tag{2.2}$$

$$R_t^i = \arg\max_j P_{i,j} \tag{2.3}$$

2.6 *Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding*

Authors:Han, Song and Mao, Huizi and Dally, William J

Deep neural networks have become the state-of-the-art technique for computer vision tasks. However, these powerful models come with significant deployment challenges, particularly for mobile systems. The AlexNet model requires over 200MB of storage, while VGG-16 exceeds 500MB, making them impractical for most mobile applications.

While larger models have more storage requirements, the energy consumption of deep neural networks presents another major barrier to mobile deployment. The computational demands of these models are dominated by memory access operations, with energy consumption primarily coming from two sources:

- Frequent fetching of weights from memory
- Intensive dot product computations

These factors make standard deep neural networks particularly power-hungry, which is incompatible with the battery constraints of mobile devices.

2.6.1 Deep Compression Approach

The "Deep Compression" methodology addresses these challenges through a novel three-stage pipeline:

Pruning Stage

The first stage removes redundant connections while preserving the most informative ones. This pruning step significantly reduces the number of parameters that need to be stored and processed, without compromising the network's accuracy.

Trained Quantization and Weight Sharing

In the second stage, weight sharing is done so that multiple connections share the same weight value. This approach means only two elements need to be stored:

- A codebook of effective weights
- Indices mapping connections to these weights

The second stage then implements **trained quantization**, converting float32 weights to int8 precision. Let $x \in [a, b]$ be a float value. The **affine quantization scheme** can be written as:

$$x = S \cdot (x_q - Z) \tag{2.4}$$

where:

- x_q is the quantized int8 value associated to x
- S is the scale (a positive float32 value)
- Z is the zero-point (an int8 value corresponding to 0 in float32)

The zero-point Z is crucial for exactly representing the value 0, which is frequently used in machine learning models. The quantized value x_q of $x \in [a, b]$ is computed as:

$$x_q = \text{round}\left(\frac{x}{S} + Z\right) \tag{2.5}$$

Huffman Coding Stage

The final stage applies Huffman coding to take advantage of the biased distribution of the quantized weights. This entropy coding technique further compresses the network representation by assigning shorter codes to more frequently occurring weights.

2.6.2 Significance of the Work

Deep Compression represents a significant advancement in making deep neural networks practical for mobile deployment. By combining these three techniques, the method achieves:

- Dramatic reduction in model size (35× to 49× compression)
- Preservation of original accuracy
- Reduced energy consumption during inference

The approach is particularly notable for its comprehensive treatment of the problem, addressing both storage and energy constraints through principled compression techniques rather than simple heuristic methods.

Chapter 3

Problem Definition

In traditional machine learning models, data processing follows a centralized approach, where all key tasks—including data collection, preprocessing, validation, training, and retraining—occur at a single location, such as a cloud server or a centralized data center.

However, this centralized model introduces several challenges:

- Privacy Risks & Data Security Concerns – Storing and processing sensitive data in a single location increases the risk of data breaches, unauthorized access, and privacy violations.

- Non-IID Data Challenges – Centralized aggregation of heterogeneous (non-independent and identically distributed) client data leads to biased models and degraded performance for edge devices with unique data distributions.

- High Computational & Storage Costs – Centralized machine learning requires large-scale computational power, expensive hardware, and high bandwidth, making it costly to scale.

- Data Silos & Limited Data Access – Organizations often restrict data sharing due to regulatory and competitive concerns, limiting the diversity and accuracy of AI models.

- Latency & Bandwidth Issues – Transferring large datasets to a central server introduces delays and network congestion, making real-time AI applications inefficient.
- Ethical Concerns: Data Ownership & Control – In a centralized setup, users lose control over their data, leading to potential misuse and ethical dilemmas.

To address these challenges, decentralized machine learning has emerged as an alternative, where AI models are trained and processed across distributed nodes without requiring centralized data storage. Techniques such as Federated Learning, Edge AI, and Blockchain-based AI allow machine learning to operate on decentralized data while preserving privacy, improving scalability, and reducing costs.

Federated Learning has emerged as a promising approach for training machine learning models on decentralized data, particularly in scenarios where data privacy is a concern and direct access to data is restricted..However, one major challenge in this setting is that clients are typically resource-constrained edge devices..These edge devices have limited memory, computational power, and battery life. In addition, they often experience unstable network connections, making efficient communication difficult. Due to these constraints, conventional machine learning algorithms, which assume powerful and stable computing environments, cannot be directly applied.This necessitates the need for algorithms that can still produce results in these environments

Another key challenge is that while PSO-based optimization algorithms have been proposed in the literature, there are no publicly available real-world implementations.While a reference implementation of FedPSO exists, it relies on inefficient Python for-loops rather than optimized framework-level operations, resulting in suboptimal performance for production deployment. This lack of implementations poses difficulties in evaluating and adapting these algorithms for federated learning. Addressing these issues requires implementing and improving existing training strategies that minimize computational overhead, reduce communication costs, and optimize model storage on devices.

In summary, there is a need to make Federated Learning more affordable for edge devices. Additionally, it is essential to implement the selected PSO algorithms, as no publicly available implementations currently exist. Once implemented, further improvements must be made to optimize network costs, computational efficiency, and memory usage.

Chapter 4

Methodology

4.1 Data Partitioning

The data partitioning pipeline begins by dividing the complete dataset into N distinct subsets matching the number of federated clients. Each subset undergoes standard preprocessing transformations (normalization, resizing, etc.) before being split into training and testing sets via scikit-learn's `train_test_split` (80-20 ratio by default). The critical design choice lies in the distribution strategy - we use:

4.1.1 IID Data Partitioning

For controlled experiments with homogeneous data distributions, we implement **uniform random partitioning** using the IidPartitioner provided by Flower to create independent and identically distributed (IID) datasets across clients. Each client k receives an approximately equal number of samples from all classes:

$$\mathbf{p}_k = \left(\frac{1}{C}, \dots, \frac{1}{C}\right) \quad \forall k \in \{1, \dots, K\} \tag{4.1}$$

where C is the total number of classes. This approach ensures:

- **Balanced local datasets**: Each client's dataset $\mathcal{D}_k$ contains:

$$|\mathcal{D}_k^c| \approx \frac{|\mathcal{D}^c|}{K} \quad \forall c \in \{1, \ldots, C\} \tag{4.2}$$

  with $\mathcal{D}^c$ denoting class-c examples in the global dataset.

- **Statistical equivalence**: All clients have identical expected class distributions, simulating ideal conditions where:

$$E[\mathbf{p}_k] = \mathbf{p}_{\text{global}} \tag{4.3}$$

4.1.2 Non-IID Data Partitioning

To simulate realistic federated learning environments with statistical heterogeneity, we employ Dirichlet distribution sampling implemented using Flowers **DirichletPartitioner**, to partition dataset $\mathcal{D}$ across K clients. For a C-class classification task:

$$\mathbf{p}_k \sim \text{Dir}_C(\alpha), \quad \alpha = 0.1 \tag{4.4}$$

where $\mathbf{p}_k = (p_{k,1}, ..., p_{k,C})$ defines the class probability vector for client k, and α controls partition skewness.This approach generates:

- **Controlled heterogeneity**: Lower α values produce more extreme non-IIDness

- **Device-specific bias**: Each client's dataset $\mathcal{D}_k$ is sampled such that:

$$|\mathcal{D}_k^c| \propto p_{k,c} \cdot |\mathcal{D}^c| \tag{4.5}$$

  where $\mathcal{D}^c$ denotes class-c examples in the global dataset.

- **Real-world relevance**: The resulting partitions mimic edge scenarios where devices naturally collect imbalanced data.

4.2 Model Architecture

We employ a simple CNN model trained using the Adam optimizer ($\eta = 0.001$) with CrossEntropy Loss, with a batch size of **10**.

Layer	**Configuration**
conv1	nn.Conv2d(3, 6, kernel_size=5)
pool	nn.MaxPool2d(2, 2)
conv2	nn.Conv2d(6, 16, kernel_size=5)
fc1	nn.Linear(16*5*5, 120)
fc2	nn.Linear(120, 84)
fc3	nn.Linear(84, 10)

Table 4.1: Model Architecture Summary

4.3 Weight Transmission in Flower

The model weights are first converted to NumPy arrays using the get_parameters function

```
def get_parameters(net) -> List[np.ndarray]:
    return [val.cpu().numpy() for _, val in net.state_dict().items()]
```

These arrays are then converted into bytes and bundled together into a dataclass called Parameters

```
    def ndarray_to_sparse_bytes(ndarray: NDArray) -> bytes:
    """Serialize NumPy ndarray to bytes."""
    bytes_io = BytesIO()
    np.save(bytes_io, ndarray, allow_pickle=False)
    return serialized_data
```

```
def ndarrays_to_sparse_parameters(ndarrays: NDArrays) -> Parameters:
"""Convert NumPy ndarrays to parameters object."""
tensors = [ndarray_to_sparse_bytes(ndarray) for ndarray in ndarrays]
return Parameters(tensors=tensors, tensor_type="numpy.ndarray")
```

At the receivers end they are converted back into the original numpy arrays

4.4 PSO

In Flower simulations, clients are ephemeral by design - they are instantiated per-round and destroyed afterwards, while the server remains persistent. To maintain client-specific state across rounds, Flower provides a `Records` object. This key-value store persists throughout the simulation and is automatically preserved between client executions

Two-types of Records:

- `ConfigsRecord` or MetricRecord: To save simple components(e.g float,boolean,integer,string bytes and lists on these types)
- `ParametersRecord`: For model parameters or generally data arrays

For example, to store a client's best accuracy:

```
client.parameters_records["best_accuracy"] = 0.82 # float
```

The data persists even when the client instance is destroyed between rounds. This mechanism is crucial for FedCPSO to maintain:

- Best accuracy
- Client-specific best models
- Velocity vectors

across the federated learning rounds.

The Records object is not essential in CPSO implementation because all the necessary data and parameters and stored at the server and not maintained by the client.

4.4.1 Client-Side Optimization

- **Model Tracking**:
  - Local and global best accuracies are maintained client-side
  - Global best updated when server model accuracy is received
  - Local model updated if PSO+training yields better accuracy
- **Storage Handling**:
  - Client models saved to disk after training (non-persistent memory)
  - `Records` object stores:
    - Best Accuracy
    - Velocity arrays
- **PSO Hyperparameters**:
  - Local training rounds at client : 1
  - Inertia coefficient (α): 0.3
  - Local acceleration: 0.7
  - Global acceleration: 1.4
- **PSO Execution**:
  1. Convert model weights θ to ndarrays
  2. Initialize per-layer velocities $v \sim \mathcal{U}(-0.1, 0.1)$

3. Update velocities using coefficients from [2]:

$$v_{t+1} = 0.3v_t + 0.7(\theta_{\text{local}} - \theta_t) + 1.4(\theta_{\text{global}} - \theta_t) \tag{4.6}$$

4. Apply updates: $\theta_{t+1} = \theta_t + v_{t+1}$
5. Perform 1 epoch SGD: $\theta_{t+1} = \theta_t - \eta\nabla\mathcal{L}(\theta_t)$

- **Server Communication**:
  - Final accuracy sent to server
  - Updated weights stored via `Records` object

4.4.2 Server-Side Processing

Unlike conventional FL systems that only require rigid synchronization intervals, our implementation uses an event-driven parameter exchange mechanism. Clients initially perform local training without transmitting updates, allowing uninterrupted model refinement. The clients then sent back the accuracies only and the server subsequently queries specific clients via `get_parameters` to retrieve only the highest-performing models based on validation metrics.

4.5 CPSO

4.5.1 Client-Side Processing

Each client performs the following steps:

- Executes local optimization (detailed in Section 4.6)
- Computes gradient updates via:

$$\theta_{t+1} = \theta_t - \eta\nabla\mathcal{L}(\theta_t; \mathcal{D}_k) \tag{4.7}$$

where η is the learning rate and $\mathcal{D}_k$ is the client's local dataset

- Transmits updated weights θ_{t+1} to the server

4.5.2 Server-Side Computation

Most computations (except client gradient descent) are performed on the server. The server maintains:

- A dictionary with client IDs as keys , ClientID can be obtained from the ClientManager
- For each client, stores:
  - Current model parameters
  - Local best model
  - Global best model
  - Best neighbor model (key FedCPSO differentiator)
  - Local best scores and velocity arrays

4.5.3 Model Updates

- Weight aggregation (FedAvg-style):

$$\theta_{\text{agg}} = \sum_{k=1}^{K} \frac{n_k}{N} \theta_k \tag{4.8}$$

- Update global model if $\frac{1}{K} \sum A_k > A_{\text{global}}$
- Update local best models if $A_k > A_{\text{local}}$

where A_k is accuracy of client k

4.5.4 Best Neighbor Selection

- Each client tracks other clients via initialized scores (default=1)
- If current accuracy is less then accuracy from previous iteration $A_t < A_{t-1}$, update best neighbor(j):

$$\text{score}[j] \leftarrow \text{score}[j] \times A_t \tag{4.9}$$

- Client with highest score becomes new best neighbor

4.5.5 Hyperparameters for CPSO

Parameters as used in the original paper

- Local training rounds at client : 1
- Inertia (α): 0.5
- Local acc.: 0.5
- Global acc.: 0.5
- Best neighbor acc.: 0.5

4.5.6 CPSO Update Rule

For each client:

$$\begin{aligned} v_{t+1} &= 0.5v_t + 0.5(\theta_{\text{local}} - \theta_t) \\ &\quad + 0.5(\theta_{\text{neighbor}} - \theta_t) + 0.5(\theta_{\text{global}} - \theta_t) \end{aligned} \tag{4.10}$$

$$\theta_{t+1} = \theta_t + v_{t+1} \tag{4.11}$$

4.6 Optimization Pipeline

We implemented a three-stage optimization process:

Magnitude Pruning

- Sort all model parameters by absolute value
- Calculate threshold τ for target sparsity s:

$$\tau = |\theta|_{(k)}, \quad k = \lfloor s \cdot n \rfloor \tag{4.12}$$

- Set weights with magnitude below τ to zero

In our experiments we have used 0.5 as the pruning ratio.

LZMA Compression

- Modified serialization to compress parameters: In the weights transmission step right after the numpy array is converted into bytes we incorporate LZMA compression and at the receivers end the received bytes are first decompressed after which normal deserialization proceeds
  - Client-side: Compress model with LZMA
  - Server-side: Decompress before processing
- Achieved 51% reduction in communication volume - Increasing sparsity also increases the compression ratio. Without magnitude pruning the compression achieved was 10%

Quantization

4.6.1 Post-training dynamic quantization

PyTorch implements post-training dynamic quantization through `quantize_dynamic()`, which converts model parameters to 8-bit integers during runtime without retraining. The process requires:

```
import torch.quantization

# Original model in evaluation mode
model = NeuralNetwork().eval()

# Apply dynamic quantization to Linear layers
quantized_model = torch.quantization.quantize_dynamic(
    model,
    {torch.nn.Linear},  # Target layer types
    dtype=torch.qint8   # Quantization precision
)
```

Key implementation details:

- Activations remain float32 (only weights quantized)
- No calibration dataset needed (scales/zero-points calculated dynamically)
- **Runtime Conversion**: Scales/zero-points calculated during execution
- **Memory Footprint**: 4$\times$ reduction (32-bit $\rightarrow$ 8-bit)
- Reduced storage by around 70%(float32 to int8)

Post training dynamic quantization implementation is available for only Linear layers currently.

4.7 Datasets

CIFAR10

We have primarily used the CIFAR10 datasets to compare results

- CIFAR-10 contains 60,000 color images in 10 classes such as airplane, automobile, bird, and cat.
- Each image is an RGB image of size 32×32 pixels.
- The dataset is divided into 50,000 training and 10,000 test images.
- Due to the color and object variety, it is more complex than MNIST and FMNIST.

Other can CIFAR10 , the MNIST and FMNIST were also used to solidify our results

MNIST:

- The MNIST dataset consists of 70,000 grayscale images of handwritten digits (0–9).
- Each image is of size 28×28 pixels.
- The dataset is split into 60,000 training images and 10,000 testing images.
- It is widely used for benchmarking models on basic image classification tasks.

Fashion-MNIST (FMNIST):

- Fashion-MNIST is designed as a more challenging replacement for MNIST.
- It contains 70,000 grayscale images of fashion items such as shirts, shoes, and bags.

- Each image is also 28×28 pixels in size and belongs to one of 10 classes.
- The dataset is split into 60,000 training and 10,000 test images.

Chapter 5

Results

All experimental results labelled FedCPSO presented in this section were obtained using our modified implementation of FedCPSO.
All experimental results labelled FedPSO presented in this section were obtained using same algorithm as original FedPSO.

5.1 Experimental Results: IID and Non-IID Performance Evaluation

5.1.1 PSO Results

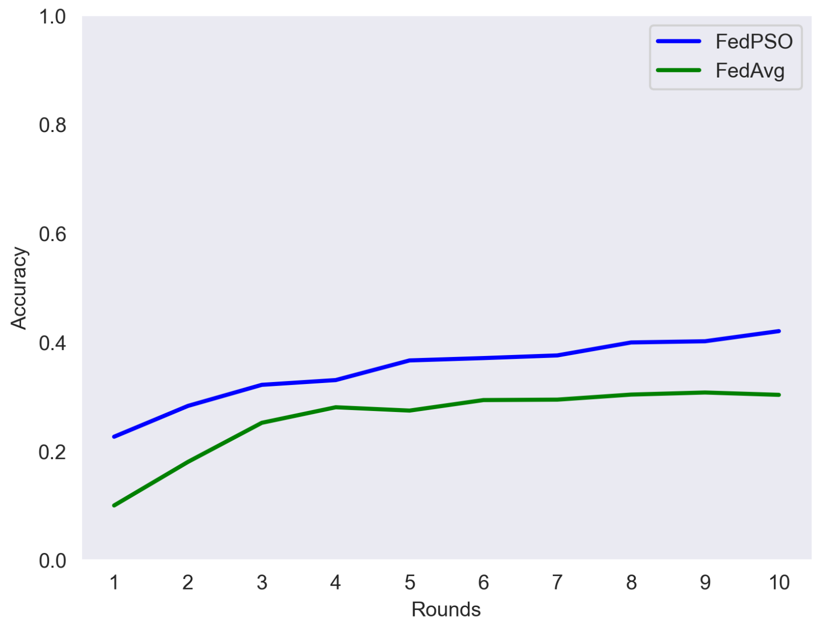

Figure 5.1: PSO vs FedAvg in IID setting with 10 clients and 10 rounds

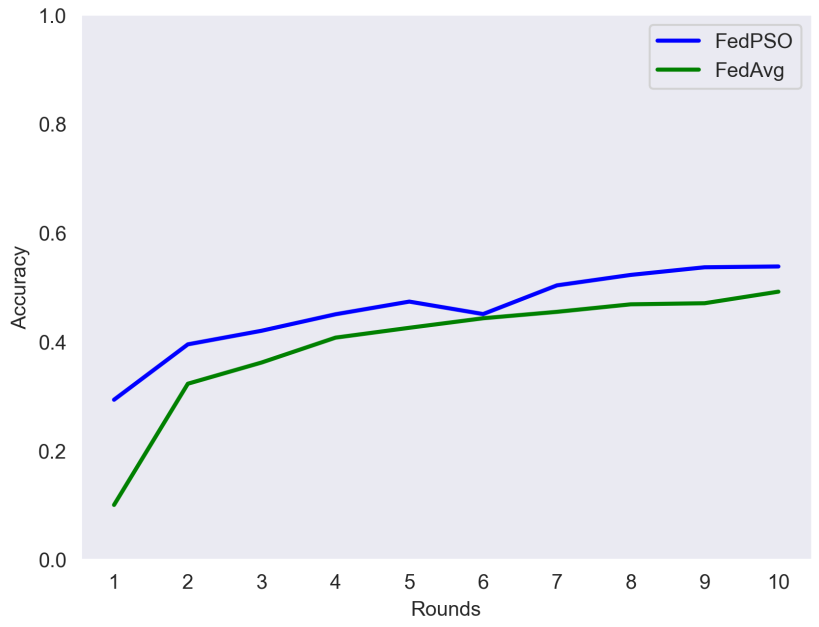

Figure 5.2: PSO vs FedAvg in IID setting with 5 clients and 10 rounds

Due to the stochastic nature of Particle Swarm Optimization, FedPSO's convergence exhibits sensitivity to initial conditions. While the algorithm consistently achieves competitive performance, final accuracy metrics may vary within a ±5% range across different random initializations.

5.1.2 CPSO Results

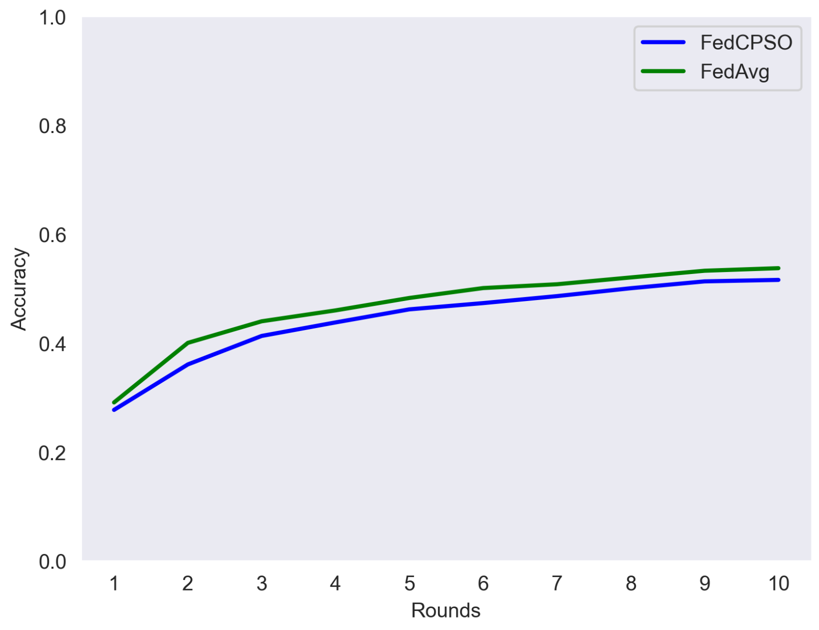

Figure 5.3: CPSO vs FedAvg in IID setting with 10 clients over 10 rounds

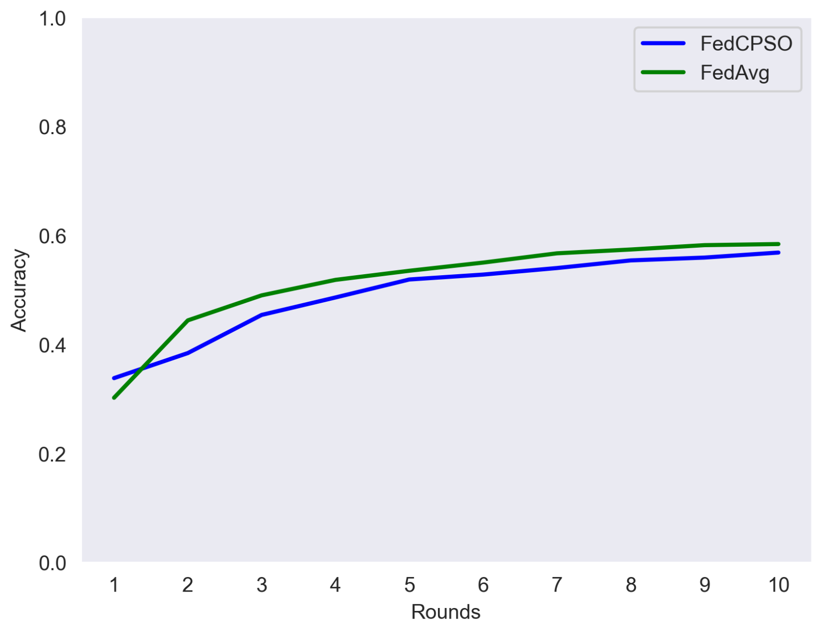

Figure 5.4: CPSO vs FedAvg in IID setting with 5 clients over 10 rounds

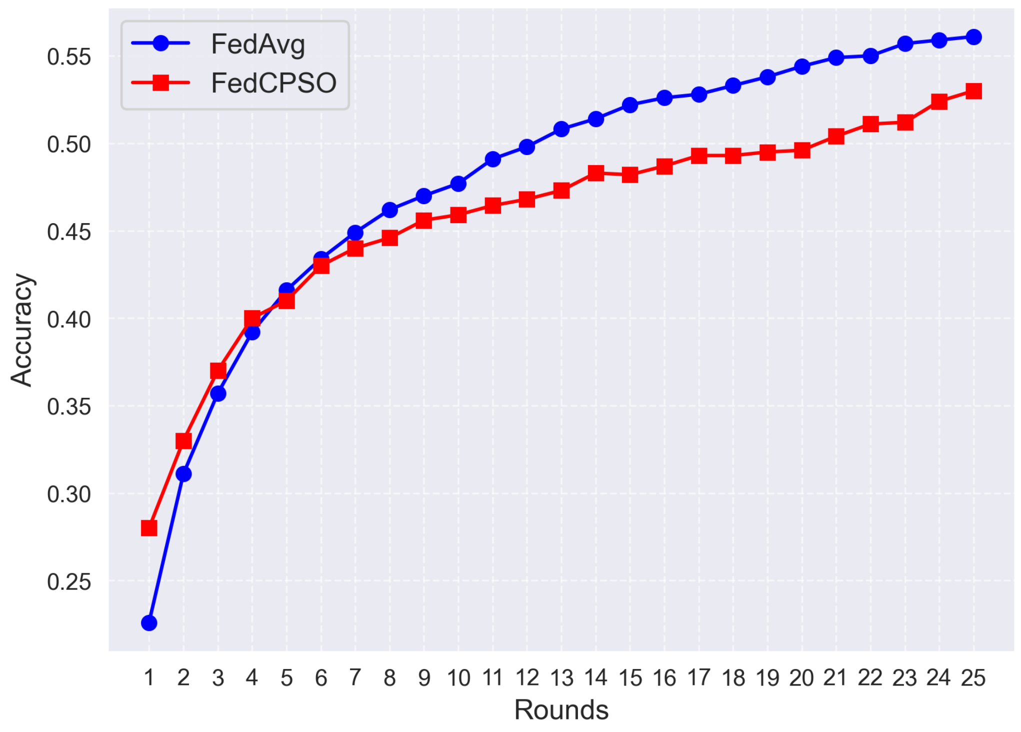

Figure 5.5: CPSO vs FedAvg in IID setting with 20 clients over 25 rounds

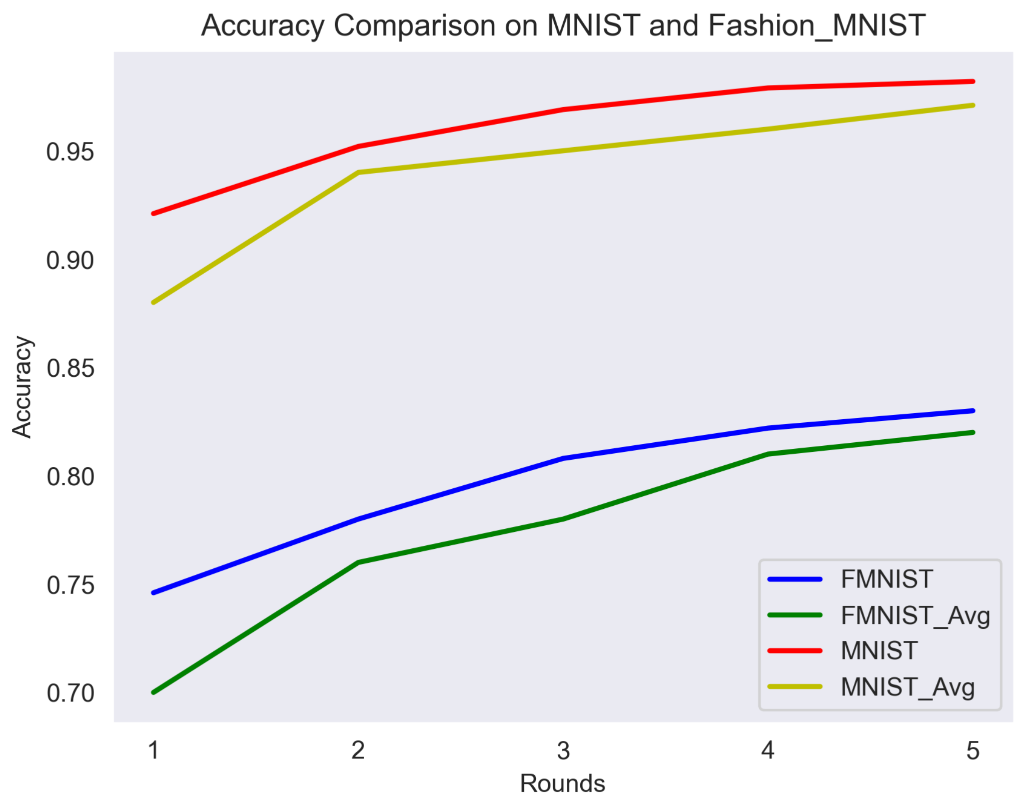

Figure 5.6: Performance on MNIST and FMNIST datasets

FedCPSO demonstrates remarkable robustness, maintaining comparable accuracy to FedAvg (only 2% degradation) and even showing higher accuracies on specifically the MNIST AND FMNIST datasets despite the additional optimization pipeline of pruning, compression, and quantization

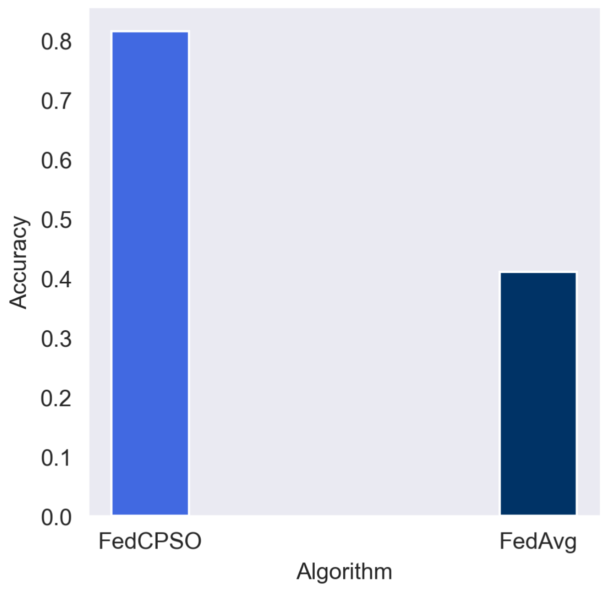

Figure 5.7: CPSO vs FedAvg in non-IID setting with 10 clients over 10 rounds

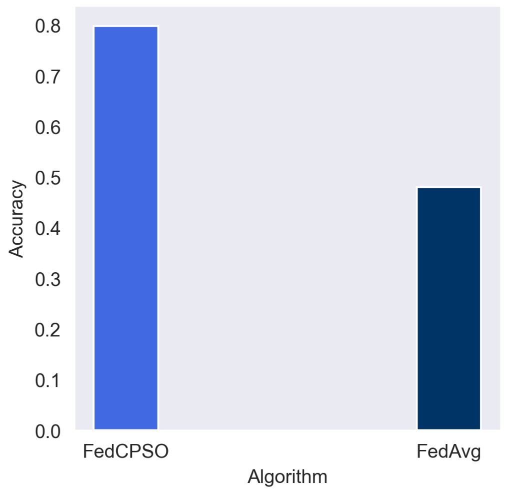

Figure 5.8: CPSO vs FedAvg in non-IID setting with 5 clients over 10 rounds

FedCPSO's 40% accuracy advantage over conventional FedAvg stems

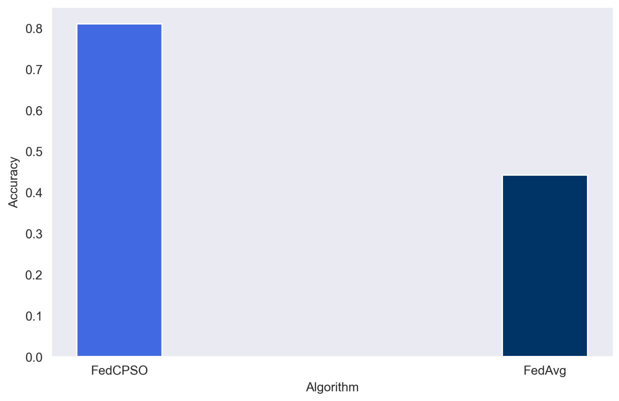

Figure 5.9: CPSO vs FedAvg in non-IID setting with 20 clients over 25 rounds

from two key innovations: (1) client-specific model adaptation for local data characteristics, and (2) combined particle swarm optimization that maintains global model coherence while respecting device limitations.

5.2 Impact of Constraint Handling and Optimizations on Model Accuracy

Building on the Deep Compression, we observed that the optimized CPSO algorithm maintained comparable accuracy to the standard implementation, with a marginal degradation of 2% in federated learning settings.

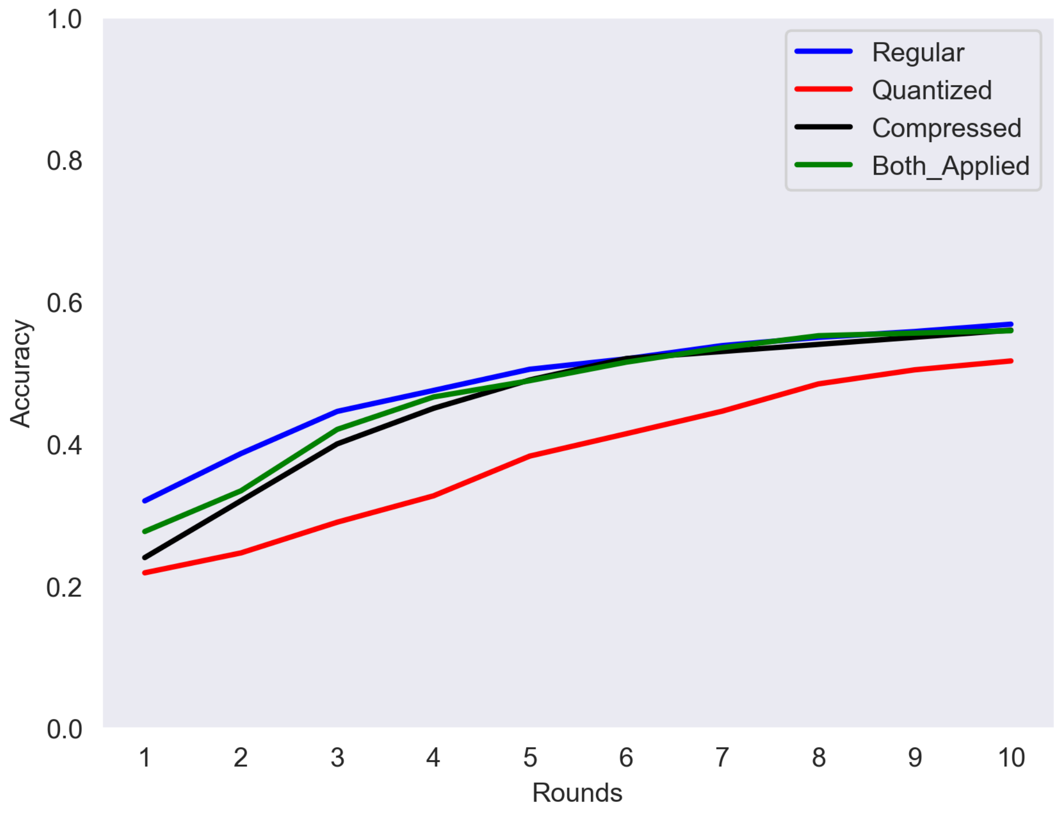

Figure 5.10: Model Accuracy Across Optimization Methods

Using quantization we were able to reduce the model size in memory by approximately 70%

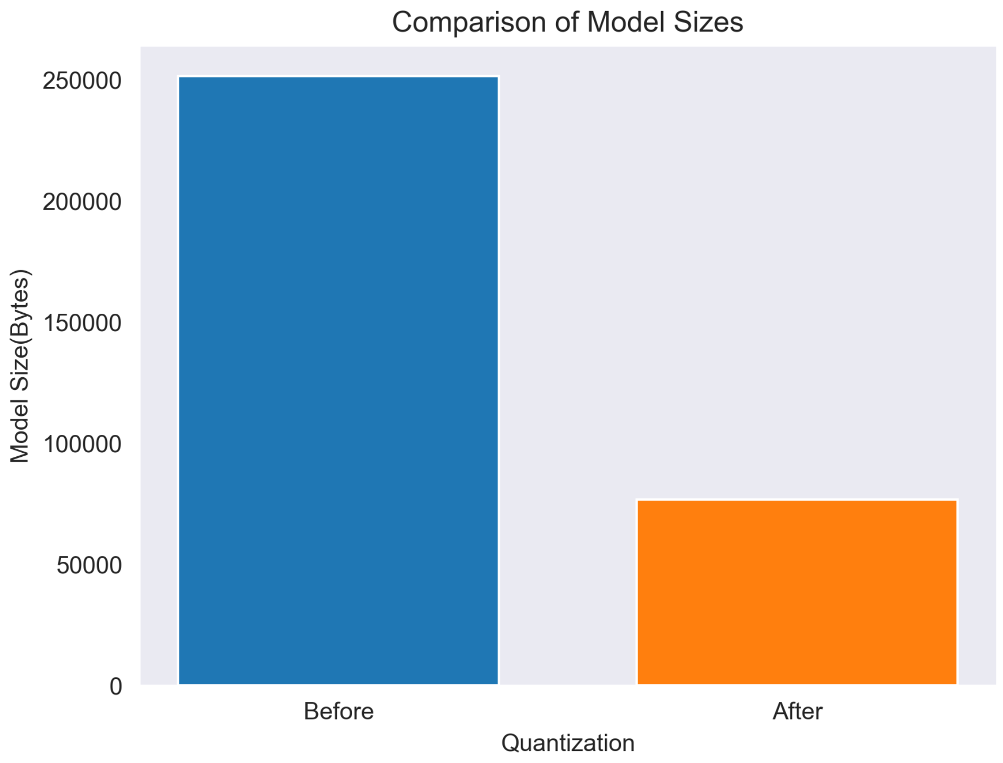

Figure 5.11: Difference in Model Sizes

Beyond reducing model size, we achieved a 51% reduction in network communication costs.

The reported resource savings reflect optimizations applied solely to linear layers; models with different architectural profiles (e.g., CNN-heavy architectures) may demonstrate varying efficiency gains.

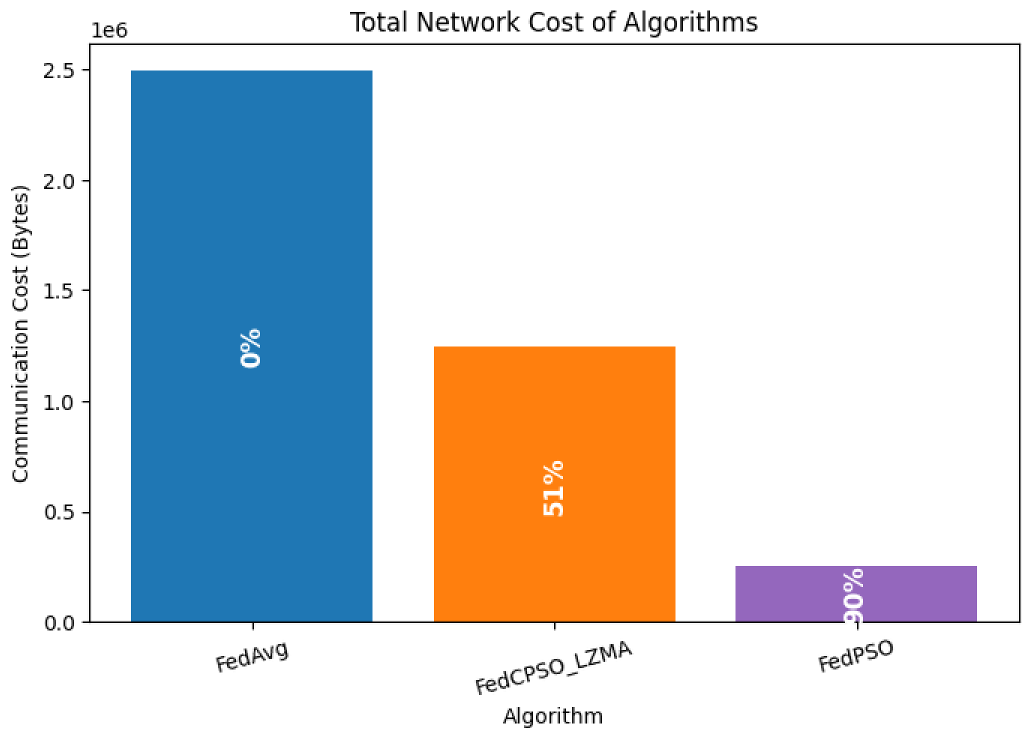

Figure 5.12: Communication Volume

Chapter 6

Conclusion and Future work

This report presented the first practical implementation of Federated PSO (FedPSO) and Federated Constrained PSO (FedCPSO), addressing the absence of publicly available implementations for these algorithms. By leveraging the Flower framework, we developed a scalable FL system capable of handling both IID and non-IID data distributions.

Furthermore we have made key improvements to FedCPSO such as:

- 51% reduction in network communication costs through optimized parameter transmission.
- 70% decrease in on-device model storage size via compression techniques.
- Minimal accuracy trade-off (5% drop vs. FedAvg in homogeneous settings).
- Superior non-IID performance, achieving 40% higher accuracy than FedAvg on heterogeneous data.

Extensive evaluation on CIFAR-10 and MNIST validated the robustness of our approach under diverse conditions. These results highlight FedCPSO's potential for real-world FL deployments, particularly in resource-constrained environments where communication efficiency and adaptability to data heterogeneity are critical.

While this project has demonstrated significant improvements in efficiency through FedPSO and FedCPSO, several promising directions remain for advancing federated learning systems:

- While current implementations of magnitude pruning and quantization primarily target Linear layers, their impact on Convolutional layers remains limited. More advanced approaches, such as Quantization-Aware Training, could be adopted to further optimize resource efficiency across all layer types.
- The hyperparameters in PSO play a large role in the effectiveness of the algorithm.Methods to determine the best hyperparameters would also be a great direction for improvement
- Combine FedCPSO with differential privacy or homomorphic encryption to enhance privacy without sacrificing efficiency.
- Test on larger edge networks (1000+ devices) with real-world latency and dropout scenarios.

References

[1] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Artificial intelligence and statistics*, pp. 1273–1282, PMLR, 2017.

[2] S. Park, Y. Suh, and J. Lee, "Fedpso: Federated learning using particle swarm optimization to reduce communication costs," *Sensors*, vol. 21, no. 2, p. 600, 2021.

[3] D. J. Beutel, T. Topal, A. Mathur, X. Qiu, J. Fernandez-Marques, Y. Gao, L. Sani, H. L. Kwing, T. Parcollet, P. P. d. Gusmão, and N. D. Lane, "Flower: A friendly federated learning research framework," *arXiv preprint arXiv:2007.14390*, 2020.

[4] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95 - International Conference on Neural Networks*, vol. 4, pp. 1942–1948 vol.4, 1995.

[5] H. Shi, R. Ma, H. Guan, and W. Zhang, "Fedcpso: Federated learning with combined particle swarm optimization," in *2023 China Automation Congress (CAC)*, pp. 3817–3822, IEEE, 2023.

[6] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.