Exploring Fully Decentralized Federated Learning with DeceFL

A report presented to the faculty of

Russ College of Engineering and Technology

In partial fulfillment of the requirements for the degree of

Master in Science

by

Kranthi Chaitanya goud katamouni[*1]

Supervised by Dr. Animesh Yadav

Submitted on June 13, 2025

[1]kk749323@ohio.edu

This report titled

Exploring Fully Decentralized Federated Learning with DeceFL

by

Kranthi Chaitanya goud katamouni

has been approved for

the School of Electrical Engineering and Computer Science

and the Russ College of Engineering and Technology by

Dr. Animesh Yadav

Assistant Professor of Electrical Engineering and Computer Science Department

Dr. Patrick Fox

Dean of Russ College of Engineering and Technology

Abstract

In traditional machine learning, training requires centralizing data on a single location, which is impractical for distributed, privacy-sensitive data like industrial sensor readings. Federated Learning (FL) addresses this by enabling local model training on devices, but most FL systems depend on a central server to coordinate updates. This introduces significant communication overhead and creates a single point of failure, making the system vulnerable to server outages or attacks, which is a critical concern for applications like fault diagnosis in manufacturing.

To overcome these limitations, I propose a novel Bidirectional Long Short-Term Memory (BI-LSTM) model integrated with Decentralized Federated Learning (DeceFL) for fault diagnosis on the Case Western Reserve University (CWRU) dataset. DeceFL eliminates the central server, allowing devices to share model weights directly with nearby peers in a dynamic network topology. The BI-LSTM architecture, designed for time-series vibration signals, excels at capturing temporal dependencies in non-convex optimization tasks, making it ideal for detecting bearing faults across distributed clients without compromising data privacy.

The approach achieves fault diagnosis accuracy is comparable to centralized methods on the CWRU dataset, demonstrating DeceFL's effectiveness for complex, non-convex problems. By reducing communication costs through sparse peer-to-peer updates and enhancing robustness to client failures, this framework offers a scalable, privacy-preserving solution for industrial applications. This work extends DeceFL's applicability to advanced deep learning models, paving the way for secure and efficient AI in healthcare, manufacturing, and beyond.

Dedication

A special thanks to my family and friends for providing the love and support necessary to accomplish my dreams.

Acknowledgements

I am thankful for the invaluable support and guidance that my advisor, Dr. Animesh Yadav, has provided me throughout my coursework and research. His unwavering patience and efforts have made this learning experience truly worthwhile. Furthermore, I express my gratitude to Dr. David Juedes and Dr. Chad Mourning for their role as committee members. Lastly, I thank my family and friends for their unwavering support and belief in me throughout this journey.

Conventions and Notation

AI	Artificial Intelligence
ML	Machine Learning
DL	Deep Learning
FL	Federated Learning
DeceFl	Decentralized Federated Learning
BI-LSTM	Bidirectional LSTM
GDPR	General Data Protection Regulation
IoT	Internet of Things
FedAvg	Federated Averaging
CFL	Centralized Federated Learning
LSTM	Long Short Term Memory
CWRU	Case Western Reserve University
DE	Drive end
FE	Fan end
IID	Independent and Identically Distributed
ViT	Vision Transformers

List of Figures

Contents

Chapter 1

Introduction

Rise of Artificial Intelligence (AI) technologies has significantly transformed how data is processed, interpreted, and used in decision-making across diverse sectors such as healthcare, finance, agriculture, transportation, and industrial automation. At the core of this transformation is the capability of machine learning (ML) models, especially deep learning models, to identify patterns within large datasets. This ability enables intelligent applications, ranging from disease diagnosis and autonomous driving to supply chain optimization and fraud detection [1, 2, 3]. As digital technology grows and sensors are used more often, we're getting more data from things like smartphones, medical records, smart buildings, and IoT devices.

However, with this expansion arises a critical bottleneck: the conflict between the demand for data driven AI systems and the obligation to safeguard data privacy. User-generated and institutionally-held data often include sensitive information such as personal health records, behavioral logs, or financial transactions that are protected by strict regulations. Among these, the European Union's *General Data Protection Regulation (GDPR)* [4] is one of the most comprehensive privacy laws, mandating that individuals must maintain control over their personal data, and that such data must not be transferred or processed without appropriate legal safeguards.

As a result, traditional machine learning approaches that depend on gathering data in cloud storage are becoming less practical in settings where privacy is a big concern. The danger of data leaks, misuse, and not following legal rules has pushed organizations to reduce data sharing, which limits access to varied, high-quality datasets. Models trained under these constraints are likely to be *biased*, less robust, and often lack the generalizability which is needed to deploy into real world especially in critical areas like *medical AI*, where data heterogeneity is essential for effective diagnostics and treatment recommendations [5, 6].

1.1 Problem Statement

To address privacy concerns while maintaining model performance, *Federated Learning (FL)* has emerged as a promising paradigm. In FL, a shared global model is trained across a network of decentralized clients such as hospitals, mobile devices, or edge sensors each with access to their own local data [24]. Instead of sending raw data to a single location, every client performs local updates and sends only model parameters such as gradients or weights, which are aggregated at a central server using techniques such as *Federated Averaging (FedAvg)*, first introduced by McMahan et al. in 2016 [26].

The introduction of FL sparked significant interest across research and industry, leading to the development of practical methods and the identification of key challenges, including *heterogeneous data distributions*, *client dropouts*, *communication constraints*, and *aggregation bias* [7, 9, 10, 11]. Several real-world applications of federated learning have been implemented in privacy critical fields like medical imaging. For instance, Kaissis et al. [12] proposed a secure federated learning framework for training convolutional neural networks on encrypted medical image data over the public internet.

Despite these advances, traditional federated learning architectures suffer from a *central design flaw* the requirement of a *central server*. This central node serves as the point of aggregation for model updates, making it a *bottleneck* for communication, computation, and coordination. Moreover, the server introduces potential *privacy risks*, as it receives aggregated data that could indirectly leak sensitive information. It also represents a *single point of failure*, undermining the system's resilience [13].

To reduce communication overhead and computational complexity, several improvements to the centralized FL model have been proposed. For example, sparse communication strategies have been developed to limit the size of transmitted updates [14], and optimization frameworks have been introduced to balance local computation with global communication under resource constraints [15]. Other works have explored *time-varying leader selection schemes*, where the role of the aggregator rotates dynamically, partially mitigating the drawbacks of static centralized architectures [31].

However, these enhancements do not fully resolve the core limitation: *centralization* remains a dependency.

1.2 The Proposed Solution

To address these fundamental limitations, we propose the adoption of a *Decentralized Federated Learning (DeceFL)* framework. Necessity of central server is eliminated by DeceFL by allowing a complete distributed training process. In this setup, clients are organized in a *connected peer to peer network* and only communicate with their immediate neighbors, rather than a global coordinator.

DeceFL presents numerous unique benefits compared to conventional federated and swarm learning methodologies:

1. **Full Decentralization**: At every stage of training, no single client has access to the global state or to all other clients updates.

2. **Provable Convergence and Optimality**: The framework is built on *principled theoretical foundations*. It has been *mathematically proven* that DeceFL can match the performance of centralized FL under the condition that the global loss function is *smooth and strongly convex*. In such cases, the algorithm achieves *zero performance gap* in terms of both training and test accuracy [19]. Furthermore, empirical studies confirm that similar performance levels are achieved even for *non convex* objectives, such as those arising in deep learning tasks [19].

3. **Flexible Communication Topology**: DeceFL supports *arbitrary connected network structures* (e.g. random graphs, structured graphs..etc), making it suitable for dynamic and heterogeneous networks such as IoT systems or ad hoc device clusters.

4. **Incentive for Participation**: All clients in the network receive a copy of the final trained model, offering a *fair and transparent incentive* to contribute resources to the collective training task. Unlike centralized methods where model access may be limited or hierarchical, DeceFL promotes equal benefit across participants.

Chapter 2

Background

2.1 Machine Learning and its Challenges

Machine learning (ML) has revolutionized numerous fields, including medicine, manufacturing, and finance, by enabling data driven decision making and predictive modeling [3]. Deep learning, a subset of ML, has been particularly impactful, achieving state of the art performance in tasks such as image recognition and natural language processing [3]. However, traditional ML relies on centralized data pipelines, where all data must be aggregated into a single repository for model training. This centralized approach poses significant challenges, particularly in scenarios where data is inherently distributed across multiple entities, such as hospitals or factories [8].

2.2 Privacy Laws and Single Data Store Issues

The centralized data gathering architecture presents significant privacy issues, especially considering rigorous rules such as the General Data Protection Regulation (GDPR). [18]. GDPR and similar laws restrict data sharing by organizations and individuals, limiting their ability to pool sensitive datasets, such as medical records or proprietary industrial data, into a single data store [18]. Transmitting decentralized datasets to a central server risks exposing private information, which can lead to legal violations, fines, and loss of trust among stakeholders [17].

Moreover, a single data store creates a single point of failure, making it vulnerable to cyberattacks, data breaches, or system failures [27].

2.3 Biases in Models Trained on Limited Data

When data sharing is restricted due to privacy concerns, machine learning models are often trained on limited, local datasets, which can introduce biases [20]. For instance, in medical AI, models trained on data from a single hospital often struggle to generalize well to broader patient populations, leading to biased predictions and reduced performance [21]. This issue is particularly pronounced in applications like COVID-19 detection, where models relying on limited radiographic data have been shown to exploit shortcuts rather than learning robust features [20]. The lack of access to diverse, comprehensive datasets is a well-known bottleneck in achieving fair and effective AI systems [21].

2.4 Introduction to Federated Learning

Federated learning (FL), which allows for collaborative model training without sharing raw data, has emerged as a possible solution to machine learning privacy concerns [23, 24]. To update a global model, FL requires clients (such as devices or institutions) to train local models on their own private datasets and share only model updates (such as gradients or parameters) with a central server [25, 26]. The seminal *Federated Averaging (FedAvg)* algorithm, introduced by Google, has become a cornerstone of FL, demonstrating communication-efficient learning across decentralized data [25, 26]. FL has been successfully applied in domains like medical imaging, where secure frameworks train convolutional neural networks over encrypted data [12]. Despite its advantages, FL still relies on a centralized architecture, which introduces significant limitations.

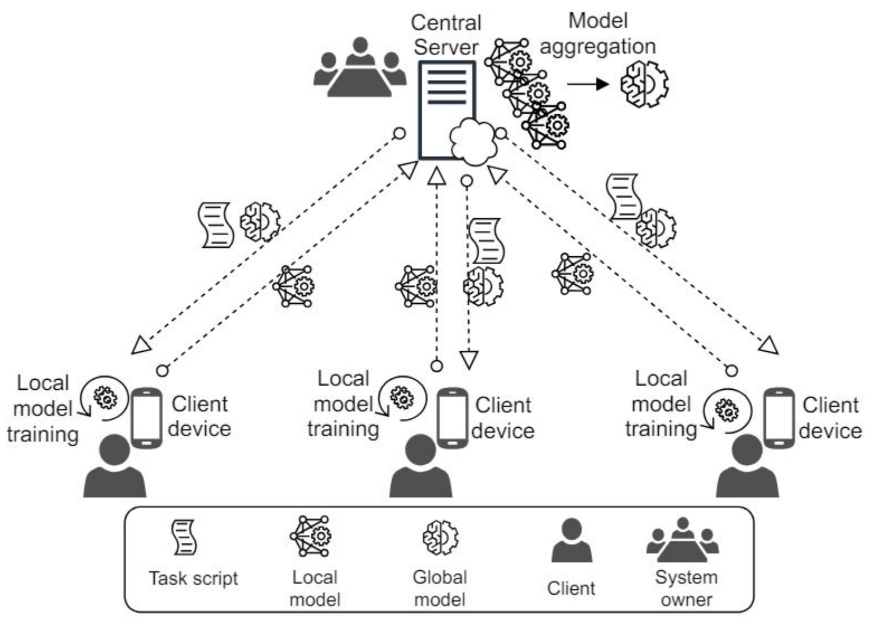


Figure 2.1: Federated Learning Overview, adapted from Lo et al. (2021) [22]

2.5 Problems with Federated Learning: Single Point of Failure

Classical federated learning frameworks, such as *FedAvg*, depend on a central server to collect and distribute model updates, creating a single point of failure [27]. This centralization leads to several drawbacks: (1) high communication burden, as all clients must send updates to the server; (2) vulnerability to attacks, where compromising the central server can disrupt the entire system or leak sensitive information; and (3) privacy risks, as model updates may still reveal information about local data [29]. Additionally, the central server imposes a rigid communication topology, limiting flexibility in dynamic or resource-constrained environments [28]. Efforts to mitigate these issues, such as sparse communication [27] or adaptive aggregation [15], still maintains the centralized system while ignoring the fundamental weaknesses.

2.6 Solution: Decentralized Federated Learning (DeceFL)

Since decentralized federated learning (*DeceFL*) does not require a central server, it provides a logical way to get around the drawbacks of centralized federated learning [19]. In *DeceFL*, clients communicate directly with their neighbors in a connected network, sharing only model parameters and performing local updates using gradient descent [19]. This fully decentralized approach, as proposed in the *DeceFL* framework, ensures: (1) no single point of failure, enhancing robustness against attacks or failures; (2) reduced communication costs, as clients exchange information only with neighbors; (3) flexible topology, supporting time-varying communication graphs; and (4) privacy preservation, as clients do not share gradients directly [19]. It has been theoretically proven that *DeceFL* converges at a rate of ($O(1/T)$), matching that of centralized federated learning (CFL), when the loss functions are smooth and strongly convex, with empirical evidence extending its effectiveness to nonconvex problems, such as deep learning [19]. By extending *DeceFL* to incorporate Long Short-Term Memory (LSTM) architectures, this project aims to leverage its decentralized framework for sequential data tasks, further enhancing its applicability to real-world scenarios like time-series analysis in medicine and manufacturing.

Long Short-Term Memory (*LSTM*) models, renowned for effectively modeling temporal dependencies in sequential data. In sequential data like vibration signals, are increasingly integrated into decentralized federated learning (*DeceFL*) frameworks for tasks such as fault diagnosis. By leveraging *DeceFL*, *LSTM* based models can be trained collaboratively across multiple nodes, enhancing scalability and robustness while addressing challenges like data heterogeneity and privacy concerns. This refined background underscores the combining effect between *DeceFL*, *LSTM* models, and datasets like *CWRU* in advancing intelligent industrial systems.

This decentralized approach is particularly advantageous in scenarios involving sensitive or distributed data, such as industrial fault diagnosis, where datasets like the Case Western Reserve University (*CWRU*) bearing dataset are prevalent. Vibration signals from bearings under various failure scenarios make up the *CWRU* dataset, which is frequently used to assess machine learning models in bearing fault detection and is hence a perfect benchmark for testing *DeceFL* frameworks in industrial applications [30]

Chapter 3

Methodology

3.0.1 Dataset Overview

A well-known standard for diagnosing bearing problems in industrial rotating machinery is the Case Western Reserve University (CWRU) bearing dataset. It provides high-frequency vibration signals acquired from motor bearings under various health conditions, including normal operation and several fault types—such as defects in the inner race, outer race, and ball elements—at multiple severity levels (7 mils, 14 mils, and 21 mils). The dataset was collected at motor speeds ranging from 1730 to 1797 RPM, which is a narrow operational band typical of controlled lab environments. While sufficient for benchmarking, it should be noted that the trained model may not generalize well to machines operating outside this speed range.

In the study by Yuan et al. (2023)[19], DeceFL, a fully decentralized federated learning algorithm, was evaluated using the CWRU dataset. Following a similar setup, this project leverages the same dataset to explore collaborative learning for fault classification using Bi-LSTM architecture trained under decentralized conditions.

This Project is implementation of Global, Local models by proposing Bi-LSTM architecture and 6-clients are used for experimentation. Experiments were conducted by generating random graphs with various connectivity, under varying condition i.e subset of clients in the connections are selected which changes every round resembling real world dynamic connectivity and robust network under IID data distribution.

3.0.2 Data Source and Availability

The dataset is publicly accessible via the Case Western Reserve University Bearing Data Center:

https://engineering.case.edu/bearingdatacenter

Vibration data were initially acquired using accelerometers positioned at both the drive end (DE) and fan end (FE) of a motor test rig setup. Vibration signals were recorded with a sampling rate of 12,000 Hz, ensuring sufficient resolution to capture subtle fault-induced signal variations. The dataset is available in MATLAB format and includes metadata for each run.

3.0.3 Data Preprocessing

The raw time-series vibration signals were preprocessed in multiple stages [19]:

1. **Segmentation**: Raw signals were segmented into windows of 300 data points to generate fixed-length sequences suitable for batch processing.

2. **Fourier Transformation**: Each segment was transformed using the Fast Fourier Transform (FFT), applied separately to both DE and FE channels.A total of 150 frequency-domain features were extracted from each channel, resulting in 300 features per sample.

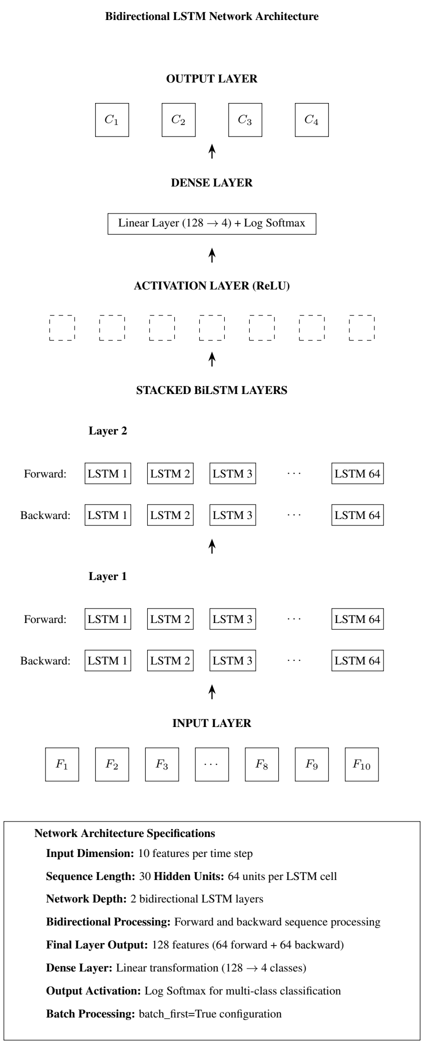
3. **Normalization**: Features were normalized to have zero mean and unit variance. Test data was normalized using training set statistics.

4. **Label Selection**: Four classes were selected (one normal, three fault types) for a simplified classification task.

5. **Client Partitioning**: The data was split into six IID subsets, each receiving a balanced sample distribution across the four selected classes.

3.0.4 Model Architecture

A deep, stacked bidirectional Long Short-Term Memory (Bi-LSTM) network was used to capture temporal dependencies in the time-series data. The model is implemented in PyTorch and defined as follows:



Network Architecture Specifications

Input Dimension: 10 features per time step

Sequence Length: 30 **Hidden Units:** 64 units per LSTM cell

Network Depth: 2 bidirectional LSTM layers

Bidirectional Processing: Forward and backward sequence processing

Final Layer Output: 128 features (64 forward + 64 backward)

Dense Layer: Linear transformation (128 $\rightarrow$ 4 classes)

Output Activation: Log Softmax for multi-class classification

Batch Processing: batch_first=True configuration

Figure 3.1: Architecture of BI-LSTM Model

3.0.5 Input Vector Preparation

Each sample in the CWRU dataset is represented by 300 frequency-domain features, obtained from high-resolution vibration signals collected using accelerometers. To make this data compatible with a sequential model like Bi-LSTM, the 300-dimensional feature vector is reshaped into a sequence of 30 time steps, with 10 features per time step. In this formulation, the input to the LSTM is a tensor of shape $[\text{batch\_size}, \text{sequence\_length} = 30, \text{input\_size} = 10]$.

This reshaping does not involve any form of dimensionality reduction or feature selection—it is simply a rearrangement of the input vector to allow the LSTM to learn temporal patterns over artificially constructed “timesteps” of the feature vector. The model then processes the input using a 2 layer bidirectional LSTM followed by a fully connected layer for classification into four fault categories.

While this setup enables the use of sequence models on static feature vectors, it is important to note that these pseudo-sequences do not reflect actual time progression. They allow the model to capture dependencies across segments of the feature vector, but do not represent a true temporal signal.

3.0.6 Federated Training Setup

To train the model in a decentralized fashion, the DeceFL framework was adopted. Each client only communicates with its neighbors, defined by an Erdős-Rényi (ER) random graph with a given connection probability p.

- $p = 0.2$: Very sparse
- $p = 0.4$: Low connectivity
- $p = 0.6$: Medium
- $p = 0.8$: Dense
- $p = 1.0$: Fully connected

Each client updates its model using local gradients and parameter averaging with neighbors:

$$w_k^{(t+1)} = \sum_{j \in \mathcal{N}_k} W_{kj} w_j^{(t)} - \eta_t \nabla F_k(w_k^{(t)})$$

This update rule is derived from decentralized SGD proposed by Lian et al. [32].

3.0.7 Loss Function Explanation

In this work, we adopt Negative Log-Likelihood Loss (NLLLoss) as the objective function for multi-class classification. This choice is directly tied to the design of our model architecture, in which the final layer applies the log-softmax activation function:

return torch.log-softmax(out, dim=1)

The log-softmax operation transforms the model's raw logits into log-probabilities, which are numerically stable and more suitable for gradient based optimization in certain deep learning contexts, especially when used with sequential models like LSTMs.

The nn.NLLLoss function in PyTorch is explicitly designed to work with such log-probabilities, computing the negative log likelihood of the true class label given the predicted distribution.

While an alternative approach would be to output raw logits and use nn.CrossEntropyLoss which internally applies log-softmax followed by NLL. our explicit use of log-softmax provides the following advantages:

Numerical Stability: Applying log-softmax explicitly allows for better control over floating point behavior in deep networks.

Interpretability: Log-probability outputs are easier to inspect and debug in applications requiring model transparency.

Modularity: Separating the activation from the loss function allows for flexible experimentation and integration with other probabilistic models or objectives.

Therefore, the use of NLLLoss is a deliberate and technically appropriate choice, given the structure of our model and our emphasis on interpretability and stability.

3.0.8 Evaluation Metrics and Reporting

Relying solely on accuracy can be insufficient, especially in multi-class classification problems. Therefore, in addition to accuracy, we evaluate and report precision, recall, F1-score, and AUC metrics, providing a comprehensive assessment of the model's performance across all four fault categories. This ensures that the model's predictive quality is measured not just by overall correctness but also by its ability to correctly identify each class, including less frequent fault types.

Regarding the use of the 1730 RPM motor speed subset, this choice aligns with the standard evaluation protocol used in the original DeceFL decentralized federated learning study [19]. The CWRU dataset includes data at speeds ranging from 1730 to 1797 RPM, but the 1730 RPM subset is often selected for testing to maintain consistency and comparability of results. This speed represents a nominal operating condition of the motor, making it a meaningful and realistic test scenario.

Furthermore, our task is a 4-class classification problem, directly predicting the exact fault category (including normal operation and three fault types - Inner race fault , outer race fault , ball bearing fault). The class labels in our training data, e.g., [0, 0, ..., 3, 3, 3], confirm this multi-class formulation, rather than a binary failure/non-failure distinction.

By adhering to the experimental setup of DeceFL and including comprehensive metrics beyond accuracy, our evaluation is both rigorous and comparable to established benchmarks in decentralized federated fault diagnosis.

3.0.9 Weight Transmission Protocol

In this implementation, each client trains a local BI-LSTM model on its private subset of the CWRU dataset and updates its local copy of the global model by aggregating weights and gradient estimates from its neighbors, as defined by the topology matrices W. The communication is structured by topology, which defines which clients exchange updates.

The protocol operates as follows:

Local Training: Each client in idxs-users (6 clients per round) trains its local model. After training, the client generates two types of updates: its current model weights (the parameters of its local model) and gradient estimates, calculated as the difference between its current and updated weights after local optimization. These updates are prepared for sharing with neighboring clients.

Topology-Based weights Exchange: Clients exchange their weights and estimated gradients with neighbors, as defined by the column W[:, ind] of the topology weight matrix. Each client uses topology, which is represented by adjacency matrices that specify which clients are connected and the influence of their updates.

Connected Neighbor-Based Aggregation: For each client , the protocol aggregates the weights and estimated gradients received from its neighbors, with the influence of each neighbor determined by the topology matrix.

Global Model Synthesis: After all clients have updated their local models, a global model is created by averaging the weights of all selected clients. This global model serves as a reference point, reflecting the collective learning across the network.

Local Communication Scope: The protocol restricts communication to direct neighbors, avoiding global broadcasts. This localized exchange reduces communication overhead, as clients only share updates with those connected in the topology. The process is simulated in-memory, abstracting real-world network complexities like transmission delays, but effectively models decentralized coordination.

This gossip-like protocol ensures that model updates propagate efficiently through the network, leveraging local consensus

3.0.10 Avoiding Multiple Counting of the Same Updates

Preventing the duplicate incorporation of weight updates (double counting) is crucial to ensure the model converges correctly without bias toward certain clients' contributions. The protocol achieves this through a synchronized update process, topology-guided aggregation, and implicit mathematical properties that guarantee balanced information diffusion.

Synchronized Update Process: All clients perform their local training and compute their weight and gradient updates before any aggregation occurs. This synchronization ensures that each client's updates are fresh and represent only the current round's training results. By collecting all updates first, the protocol avoids reusing or reprocessing the same weights within a single round.

Topology-Guided Aggregation: The topology matrix assigns weights to each neighbor's contribution, ensuring that only direct neighbors influence a client's update. These weights are normalized so that the total influence of all neighbors sums to one, preventing any single client's update from being overrepresented. For example, if a client has three neighbors, each might contribute one-third of the update, ensuring a balanced aggregation. This normalization is mathematically enforced to maintain fairness across the network.

Single Aggregation per Client: Each client aggregates its neighbors' updates exactly once per round. The protocol iterates through the clients systematically, ensuring that no client's weights are used multiple times within the same round. This structured approach eliminates the risk of redundant contributions.

Mathematical Convergence and Update Integrity: In our implementation, the weight mixing matrices are intended to be row-stochastic, meaning that rows add up to 1, in accordance with convergence results established in decentralized optimization literature (e.g. Lian et al., 2017). These matrices ensure that, during each communication round, every client integrates its own model update (self-weight) along with those of its direct neighbors.

Since clients explicitly update their models once per round, and because the mixing process is driven by a fixed topology matrix, each client's contribution is accounted for exactly once per round in its local aggregation. There is no risk of redundant updates even in the presence of overlapping neighborhoods, as the row-stochastic formulation implicitly normalizes the contributions.

Moreover, over multiple rounds, the iterative nature of decentralized averaging naturally propagates information throughout the network, ensuring eventual consistency without requiring centralized coordination or duplicate tracking.

In our decentralized learning framework using Erdős–Rényi topologies, each client performs synchronized aggregation in every communication round using only the model updates from its direct neighbors as specified by the topology matrix. There is no transitive relaying or forwarding of updates—each client's model is updated exactly once per round based solely on immediate neighbor interactions. The matrices of topology are intended to be row-stochastic and and include self-weights, ensuring that within cliques or densely connected subgraphs, each neighbor's influence is proportionally balanced (e.g., 1/k in a k-node clique). This normalization prevents any single update from being overrepresented. Furthermore, the graph generation process ensures full connectivity through Laplacian rank checks, allowing model updates to diffuse gradually across the entire network over multiple rounds rather than being redundantly relayed. This iterative and localized aggregation approach, grounded in decentralized optimization theory (e.g. Lian et al., 2017), ensures accurate, balanced weight updates and naturally avoids duplication, even in the presence of small cliques.

3.0.11 Model inversion attacks

While our decentralized federated learning framework enhances privacy by distributing training across multiple clients and using topology-based mixing of updates, it does not fully prevent model inversion attacks. Because clients exchange weights and gradients with neighbors, an adversary controlling one or more nodes—especially with knowledge of the topology matrix and model architecture—could potentially reconstruct sensitive training data using model inversion techniques.

Current limitations include:

No explicit differential privacy or noise addition to obfuscate gradients. Absence of encryption or secure aggregation mechanisms.

Small network size and fixed topology matrices increase the risk of data leakage.

Future enhancements to strengthen privacy and mitigate model inversion risks could include:

Differential Privacy (DP): Incorporate DP by adding calibrated noise to gradients or weight updates before sharing, which provides formal privacy guarantees against reconstruction attacks.

Secure Aggregation: Use cryptographic protocols such as secure multi-party computation or homomorphic encryption to enable clients to aggregate updates without revealing individual contributions.

Dynamic Topologies and Larger Networks: Increase network size and dynamically alter topology graphs more frequently to dilute individual client contributions and reduce inference opportunities.

3.0.12 Experiment Tracking and Results Access

All training logs, evaluation metrics, model checkpoints, and visualizations across different phases and topologies have been systematically tracked using Weights & Biases (WandB). This provides full transparency and reproducibility of the experiments conducted. Detailed results, including loss curves, accuracy trends, AUROC plots, and communication patterns, are available at the associated WandB project dashboard.

Access Link: `project`

3.0.13 Results

Visualizations show that DeceFL achieves strong performance with BI-LSTM Model even under sparse communication conditions, with improvements in accuracy and convergence speed as connectivity increases.

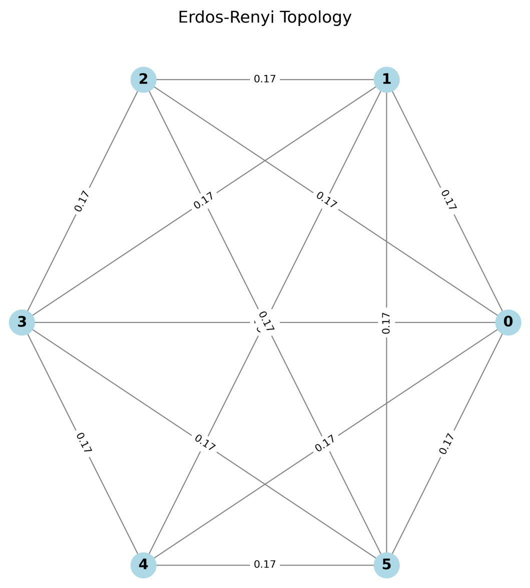


Figure 3.2: Bi-LSTM Model - Static graph topology , Connected

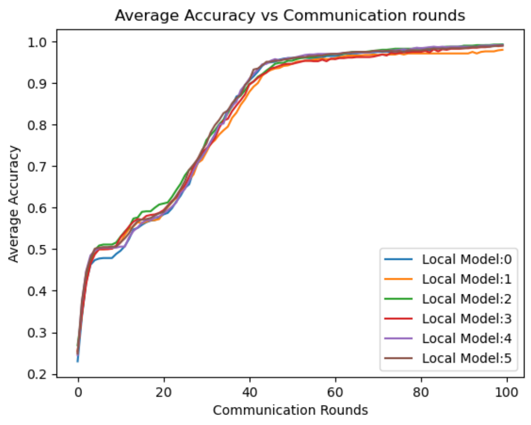


Figure 3.3: Training Accuracy, Bi-LSTM Model with DeceFL - Static graph topology , Connected

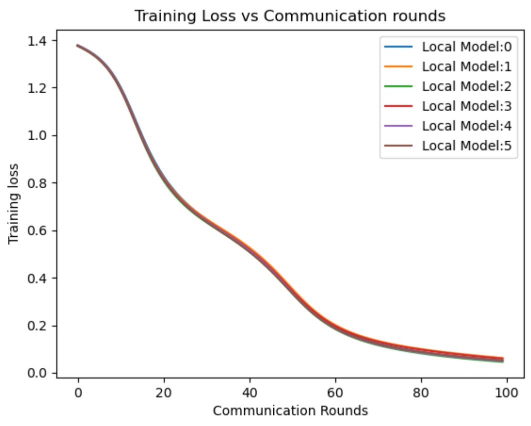


Figure 3.4: Training Loss, Bi-LSTM Model with DeceFL - Static graph topology , Connected

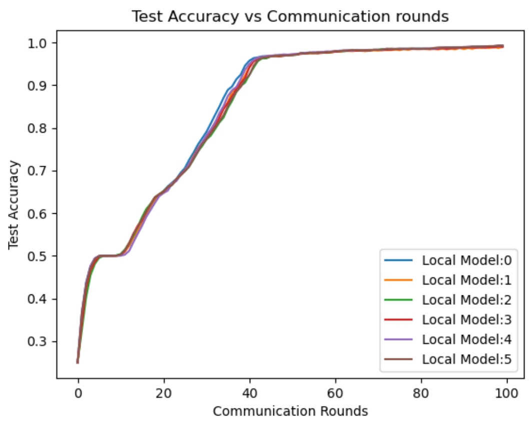


Figure 3.5: Testing Accuracy, Bi-LSTM Model with DeceFL -Static graph topology , Connected

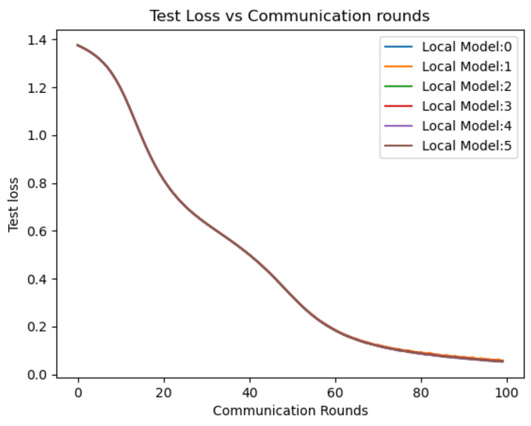


Figure 3.6: Testing Loss, Bi-LSTM Model with DeceFL -Static graph topology , Connected

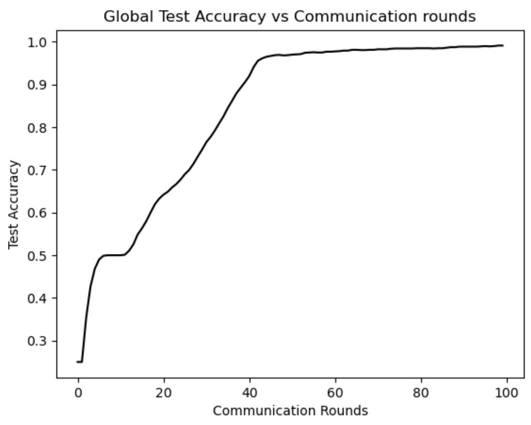


Figure 3.7: Global Accuracy , Bi-LSTM Model with DeceFL -Static graph topology , Connected

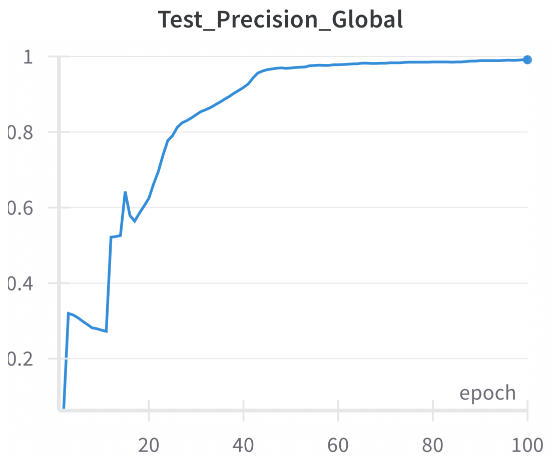


Figure 3.8: Global precision, Bi-LSTM Model with DeceFL -Static graph topology , Connected

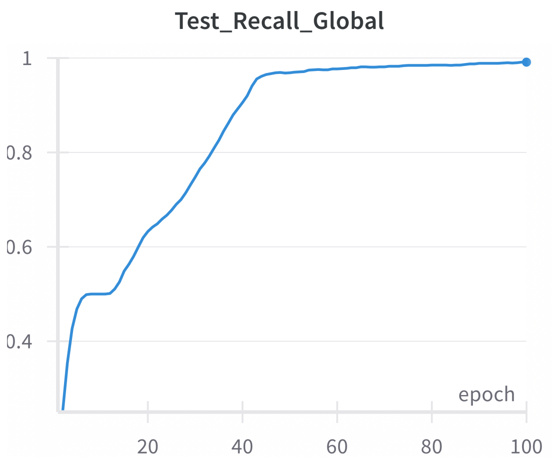


Figure 3.9: Global Recall , Bi-LSTM Model - Static graph topology ,Connected

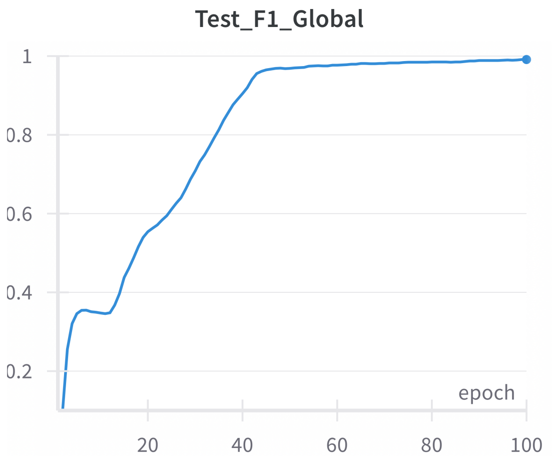


Figure 3.10: Global F1 Bi-LSTM Model - Static graph topology ,Connected

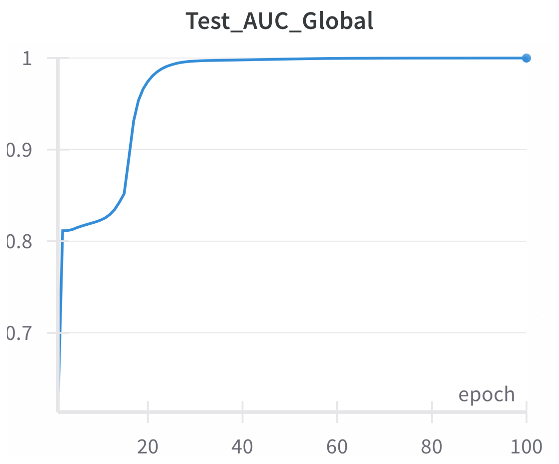


Figure 3.11: Global AUROC Bi-LSTM Model - Static graph topology ,Connected

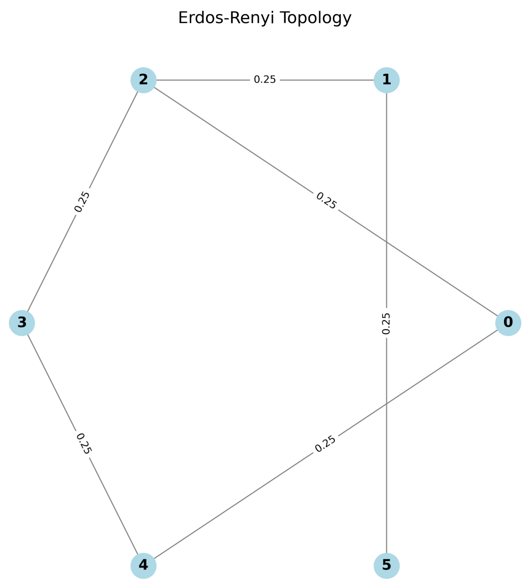


Figure 3.12: Bi-LSTM Model - Static graph topology , 0.2 probability connections

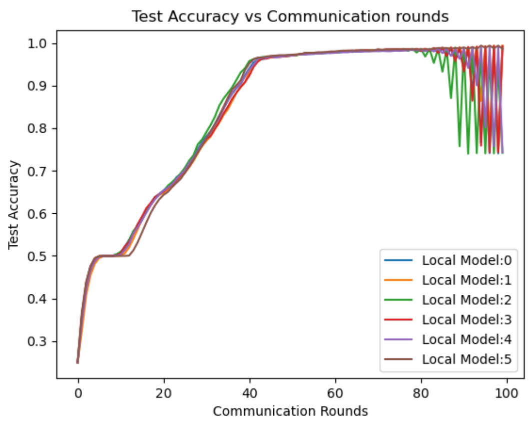


Figure 3.13: Testing acuracy Bi-LSTM Model - Static graph topology , 0.2 probability connections

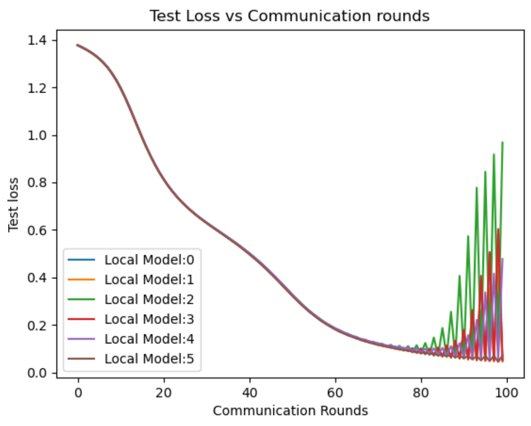


Figure 3.14: Testing loss Bi-LSTM Model - Static graph topology , 0.2 probability connections

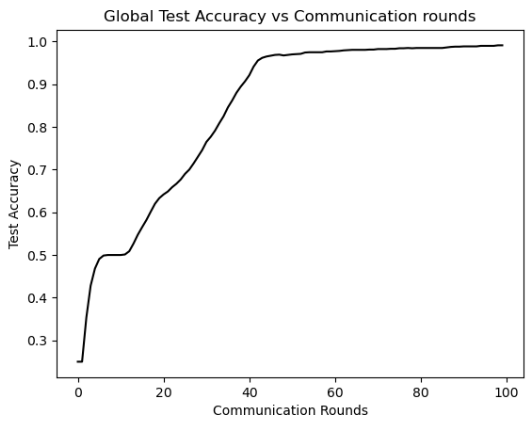


Figure 3.15: Global Model Bi-LSTM Model - Static graph topology , 0.2 probability connections

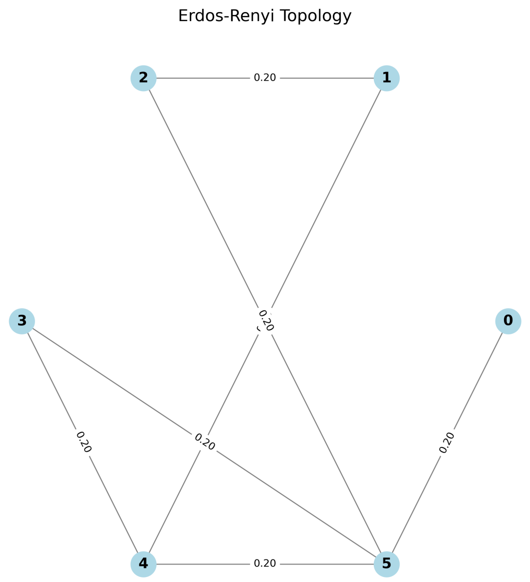


Figure 3.16: Bi-LSTM Model - Static graph topology , 0.4 probability connections

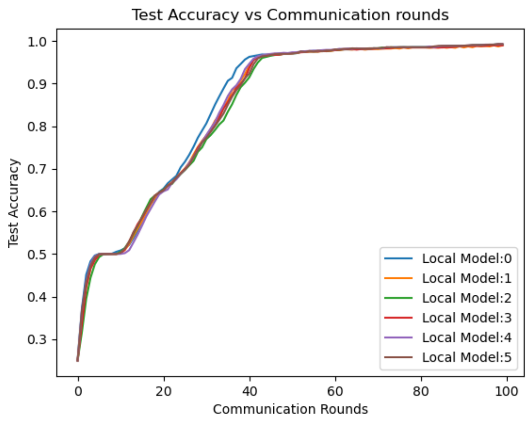


Figure 3.17: Testing acuracy Bi-LSTM Model - Static graph topology , 0.4 probability connections

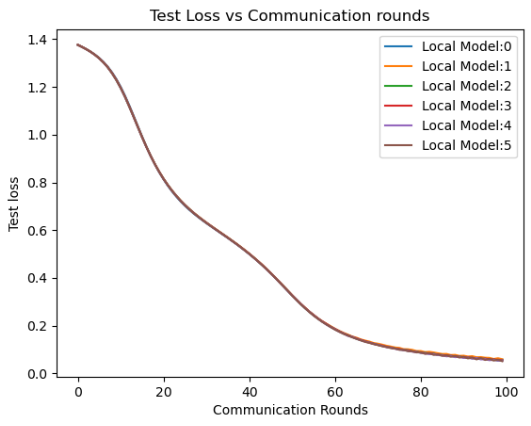


Figure 3.18: Testing loss Bi-LSTM Model - Static graph topology , 0.4 probability connections

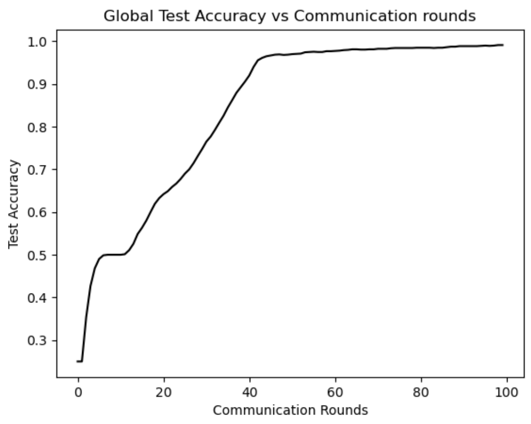


Figure 3.19: Global Model Bi-LSTM Model - Static graph topology , 0.4 probability connections

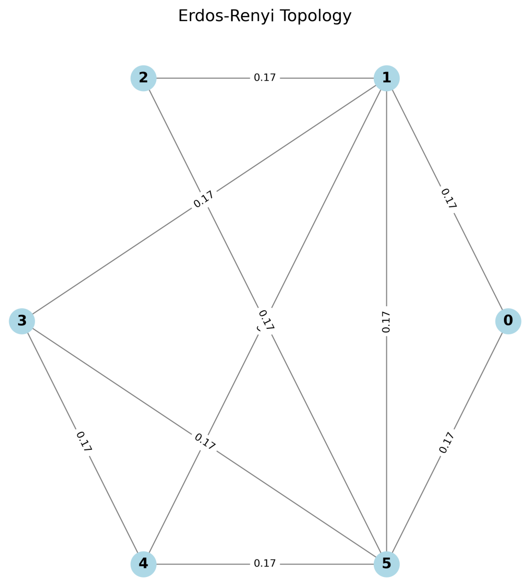


Figure 3.20: Bi-LSTM Model - Static graph topology , 0.6 probability connections

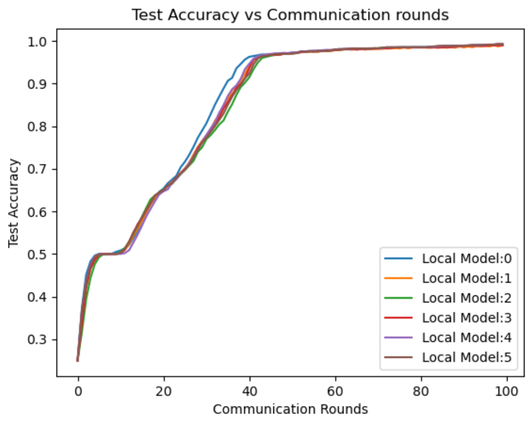


Figure 3.21: Testing acuracy Bi-LSTM Model - Static graph topology , 0.6 probability connections

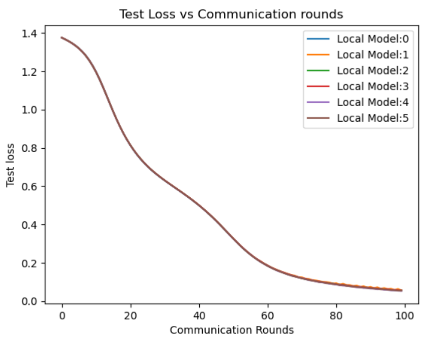


Figure 3.22: Testing loss Bi-LSTM Model - Static graph topology , 0.6 probability connections

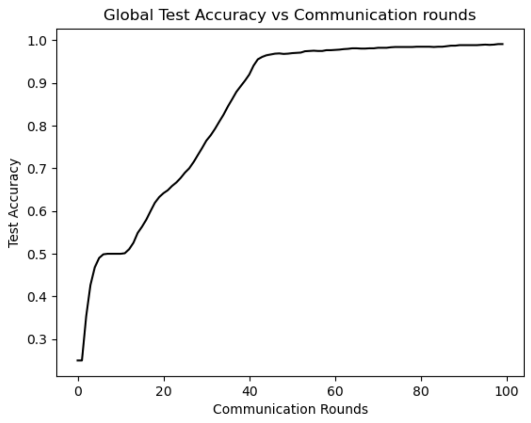


Figure 3.23: Global Model Bi-LSTM Model - Static graph topology , 0.6 probability connections

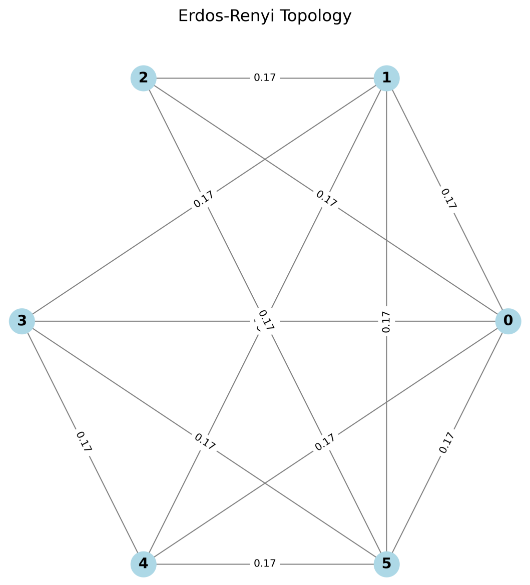


Figure 3.24: Bi-LSTM Model - Static graph topology , 0.8 probability connections

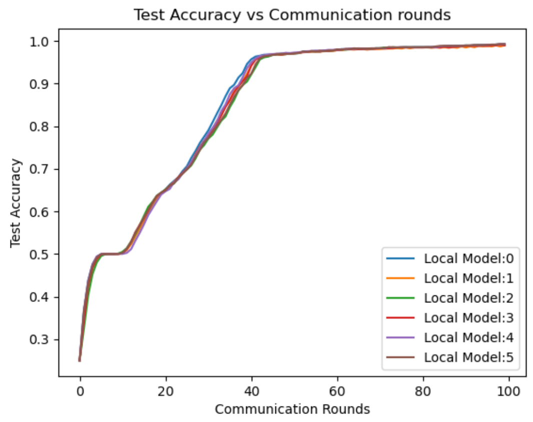


Figure 3.25: Testing acuracy Bi-LSTM Model - Static graph topology , 0.8 probability connections

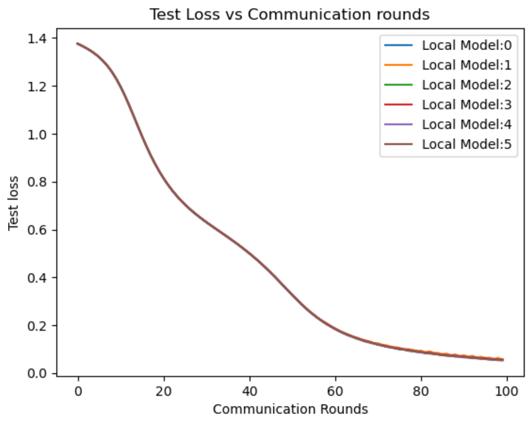


Figure 3.26: Testing loss Bi-LSTM Model - Static graph topology , 0.8 probability connections

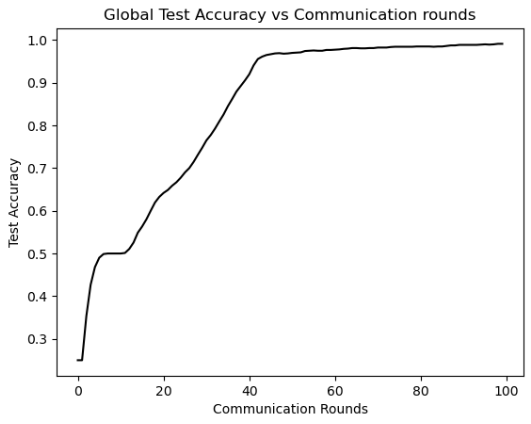


Figure 3.27: Global Model Bi-LSTM Model - Static graph topology , 0.8 probability connections

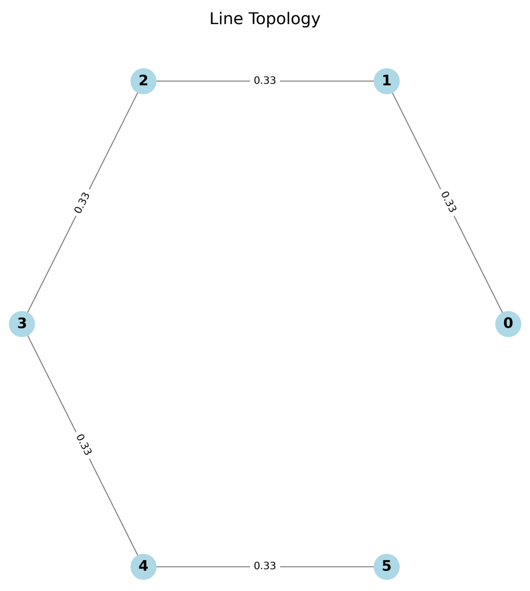


Figure 3.28: Bi-LSTM Model - Line graph topology

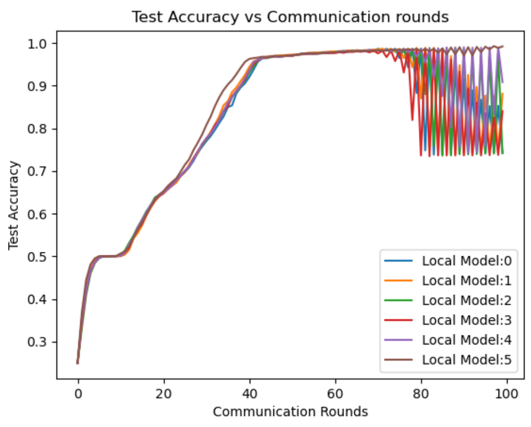


Figure 3.29: Testing acuracy Bi-LSTM Model - Line graph topology

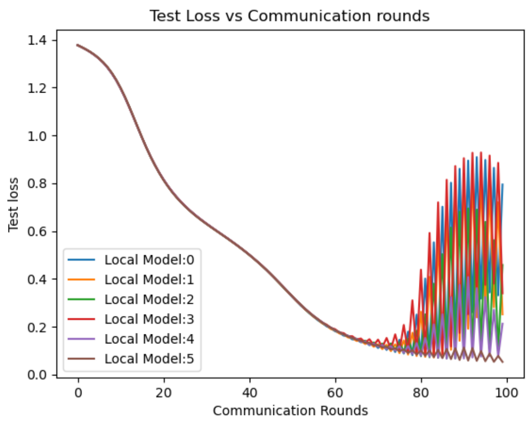


Figure 3.30: Testing loss Bi-LSTM Model - Line graph topology

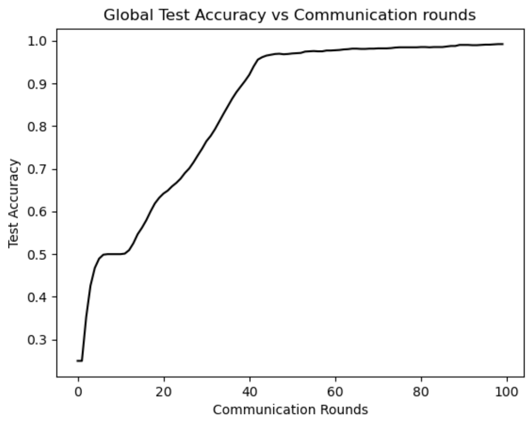


Figure 3.31: Global Model Bi-LSTM Model - Line graph topology

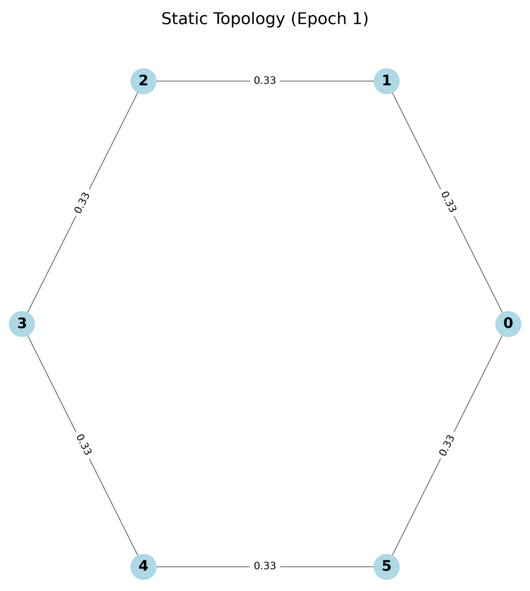


Figure 3.32: Bi-LSTM Model - ring graph topology

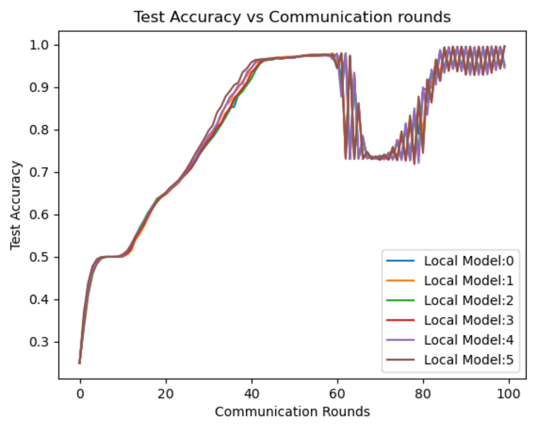


Figure 3.33: Testing acuracy Bi-LSTM Model - ring graph topology

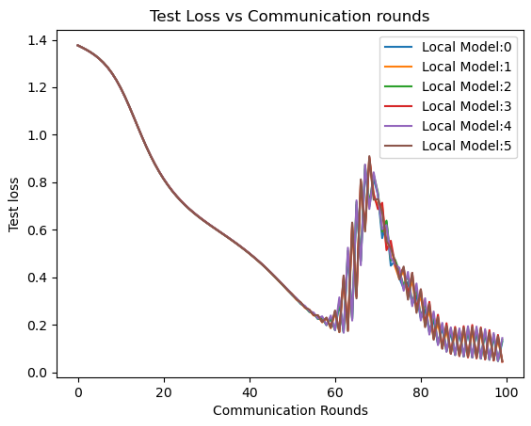


Figure 3.34: Testing loss Bi-LSTM Model - ring graph topology

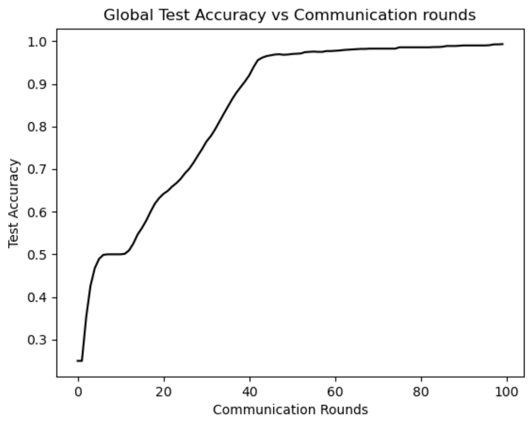


Figure 3.35: Global Model Bi-LSTM Model - ring graph topology

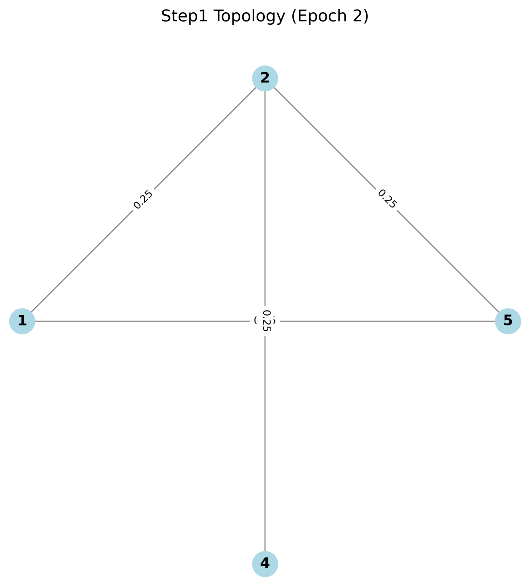


Figure 3.36: Bi-LSTM Model - Dyanamic graph topology - 1 , p = 0.2

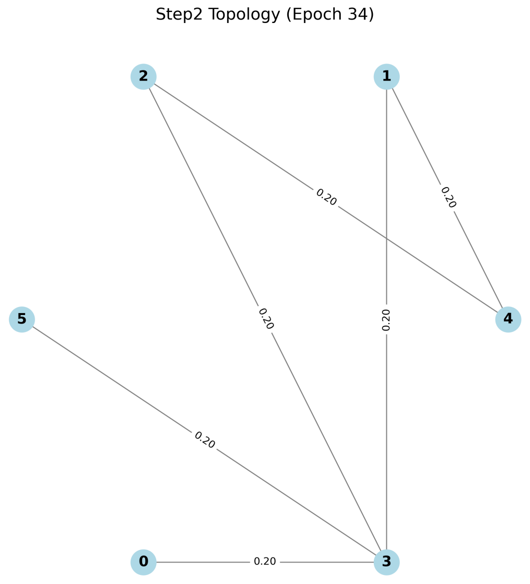


Figure 3.37: Bi-LSTM Model - Dyanamic graph topology - 2, added clients in training - p= 0.2

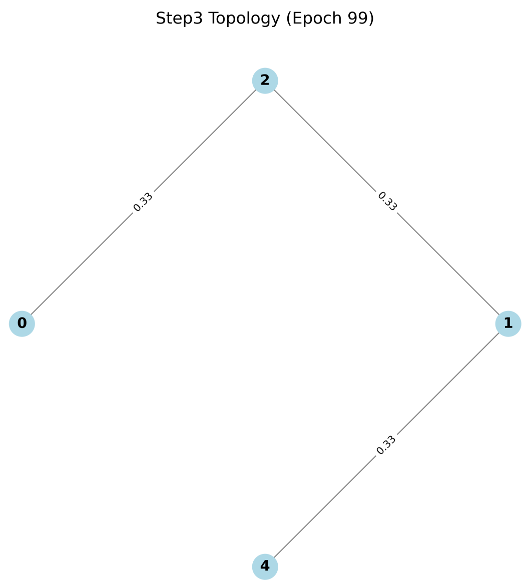


Figure 3.38: Bi-LSTM Model - Dyanamic graph topology -3 , dropped clients in training - p = 0.2

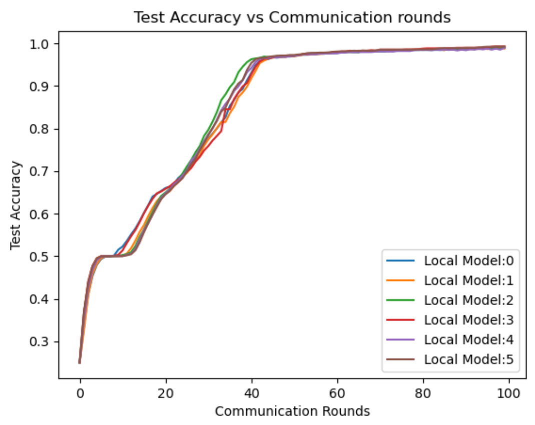


Figure 3.39: Testing accuracy Bi-LSTM Model - Dyanamic graph topology , p = 0.2

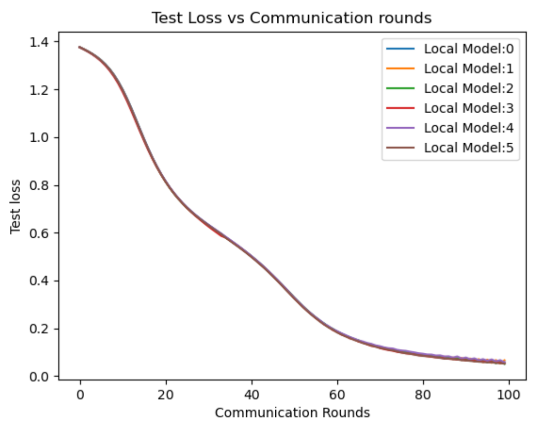


Figure 3.40: Testing loss Bi-LSTM Model - Dyanamic graph topology , p = 0.2

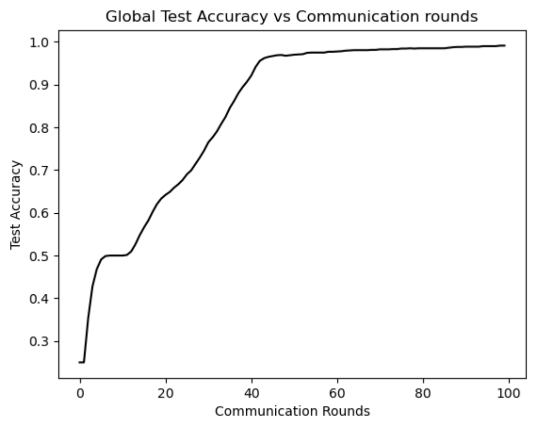


Figure 3.41: Global model Bi-LSTM Model - Dyanamic graph topology , p = 0.2

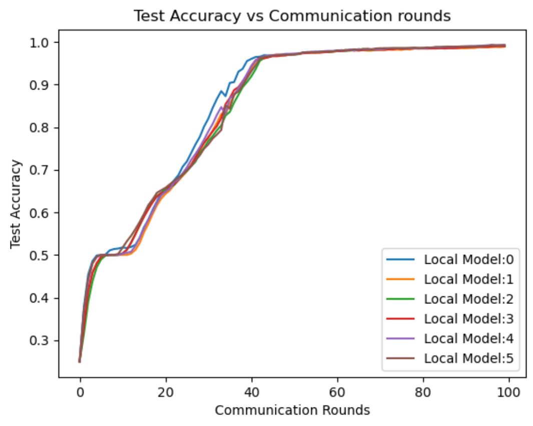


Figure 3.42: Testing Accuracy , Bi-LSTM Model - Dyanamic graph topology , p = 0.4

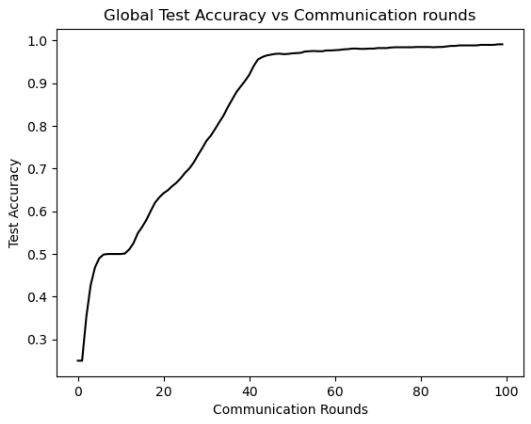


Figure 3.43: Global Model , Bi-LSTM Model - Dyanamic graph topology , p = 0.4

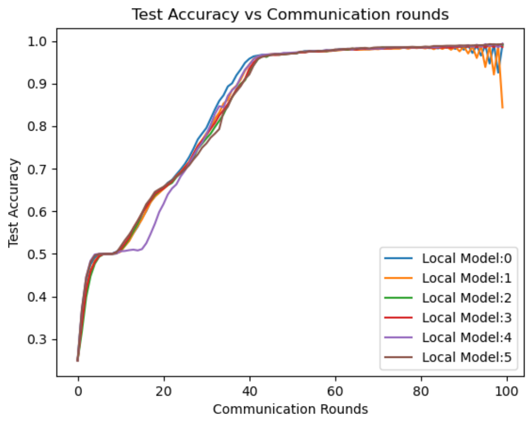


Figure 3.44: Testing accuracy ,Bi-LSTM Model - Dyanamic graph topology , p = 0.6

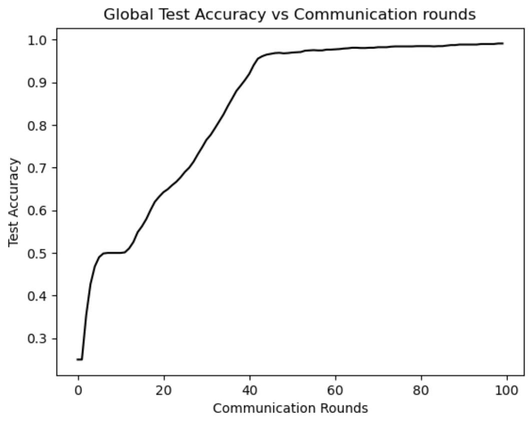


Figure 3.45: Global model Bi-LSTM Model - Dyanamic graph topology , p = 0.6

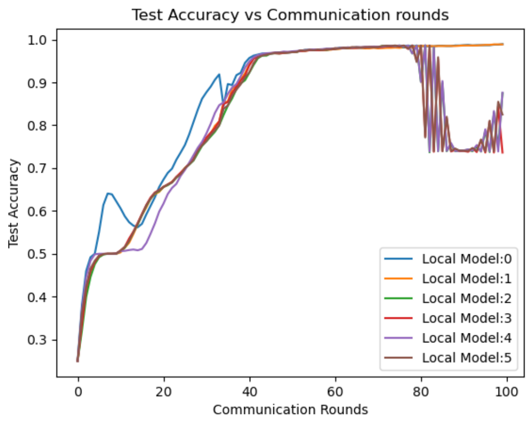


Figure 3.46: Testing accuracy, Bi-LSTM Model - Dyanamic graph topology , p = 0.8

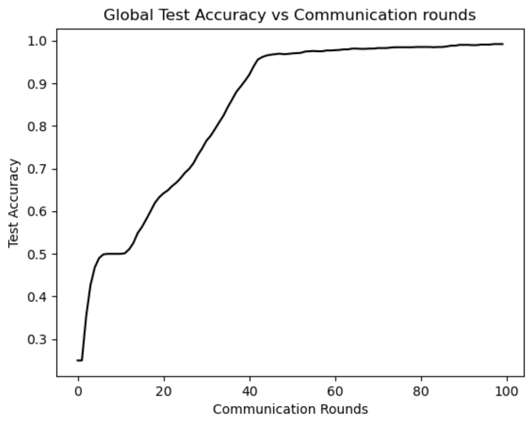


Figure 3.47: Global model Bi-LSTM Model - Dyanamic graph topology , p = 0.8

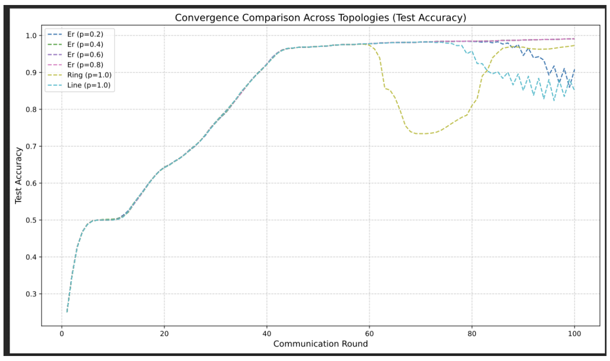


Figure 3.48: Comparision Graph of Testing Accuracy - Various p's [0.2,0.4,0.6,0.8] ER Graph Topology , p [1.0]- Line and Ring Topology

Chapter 4

Discussion, Conclusion and Future Work

Discussion

The Term Global in the context of Result section figures refers to average of local models results of all 6 nodes.

This study evaluates the performance of a Decentralized Federated Learning (DeceFL) framework using a Bidirectional Long Short-Term Memory (Bi-LSTM) model for bearing fault classification on the Case Western Reserve University (CWRU) dataset, across topologies: Erdős-Rényi (ER) with connection probabilities (p=[0.2, 0.4, 0.6, 0.8]) static and dynamic network , and line and ring topologies with connection probablity (p=1.0). The evaluation encompasses test accuracy of local models , test loss of local models , training accuracy of local models , training loss of local models , and average mean accuracy of local models, as visualized in Figures 3.13, 3.17, 3.21, 3.25 (test accuracy, ER topologies, pages 37–39), 3.14, 3.18, 3.22, 3.26 (test loss, ER topologies, pages 37–43), 3.29, 3.33 (test accuracy, line and ring, pages 44, 46), 3.30, 3.34 (test loss, line and ring, pages 45, 47), 3.4 (training loss, page 30), 3.5 (training accuracy, page 30) and the aggregated comparision plot in convergence of various topologies, provide insights into the convergence behavior and performance of the DeceFL framework under varying connectivity structures. These graphs plot the mean test accuracy across six clients over 100 communication rounds, with each client training on an independent and identically distributed (IID) samples, preprocessed into 300 frequency-domain features (page 23). The discussion below interprets the trends, particularly the scalloping (oscillatory) behavior in sparse topologies, and explores the underlying factors driving these patterns.

Topologies of Erdős-Rényi (Figures 3.13, 3.17, 3.21, 3.25): The connection probabilities of the ER topologies show different convergence patterns. For (p=0.2) (Figure 3.13, page 37), the test accuracy starts at approximately 0.25, reflecting random initialization, and rises to a moderate range by round 100. However, the curve displays noticeable scalloping in the later stages (after 80 rounds), with oscillations of approximately 0.05–0.10 (e.g., dropping from 0.95 to 0.75)range. This sparse topology, with few connections, results in slow information diffusion, causing delayed model consensus across clients. As (p) increases to 0.4 (Figure 3.17) and 0.6 (Figure 3.21), the test accuracy improves, reaching 0.90–0.96 with reduced oscillations, reflecting faster update propagation due to increased connectivity. For (p=0.8) (Figure 3.25), the accuracy achieves the highest range (0.95–0.98, Table 3.1), with minimal scalloping, indicating near-optimal convergence due to dense connections facilitating rapid consensus. The aggregated comparision plot in convergence confirms this trend, showing smoother curves for higher (p)-values, with (p=0.8) approaching the performance of a fully connected topology. The scalloping in sparse ER graphs ((p=0.2)) is likely due to delayed updates from distant clients, which introduce significant corrections when aggregated, disrupting local models in the non-convex Bi-LSTM loss landscape.

Line and Ring Topologies (Figures 3.29, 3.33): The line topology (Figure 3.29, page 44) exhibits the most pronounced oscillations among the topologies studied. The test accuracy, starting at 0.25, reaches a peak of 0.98 by round 75 but shows severe drops, such as from 0.98625 to 0.748125 , resulting in fluctuations of 0.20–0.25 in the later stages. This behavior stems from the linear structure, where each client communicates only with one or two neighbors (Figure 3.28, page 44), causing updates to propagate slowly across the chain (e.g., from client 1 to client 6). The delayed updates lead to divergent local models, and when distant weights are finally incorporated, they cause significant disruptions, as evidenced by the sharp accuracy drops. The ring topology (Figure 3.33, page 46) performs better, achieving high accuracy (0.95–0.98) with milder oscillations . Its cyclic structure (Figure 3.32, page 46) allows bidirectional update flow, enabling faster circulation of model weights compared to the line topology, though still slower than dense ER graphs. The aggregated comparision plot in convergence highlights that both line and ring topologies exhibit scalloping, with the line topology's oscillations being the most severe due to its restrictive connectivity.

Explaining the Scalloping Behavior: The oscillatory behavior, particularly pronounced in sparse topologies (ER with (p=0.2), line, and to a lesser extent, ring), is primarily driven by oscillations in model updates due to sparse connectivity, non-convex optimization, and local training dynamics. In sparse topologies, the limited number of connections (e.g., low (p) in ER, linear structure in line) slows the diffusion of model weights, as described by the DeceFL update rule (page 25). This delay causes clients to train on divergent local models, and when updates from distant clients arrive, they introduce large corrections, leading to accuracy fluctuations (e.g. line topology's drop from 0.986 to 0.748). The Bi-LSTM's non-convex loss landscape exacerbates this, as weight averaging may push models toward different local minima, causing instability. Additionally, the fixed learning rate with minimal decay amplifies overshooting, particularly in sparse graphs where updates are sporadic. Numerical instabilities in weight averaging may add minor perturbations, but their impact is secondary to connectivity and optimization challenges.

Comparative Insights and Implications: The comparison across topologies reveals that connectivity is a critical determinant of convergence stability. Dense ER topologies ((p=0.8)) achieve smoother convergence and higher accuracy due to rapid update diffusion, as seen in Figure 3.25, which notes "Excellent" accuracy and "Fast" convergence. In contrast, sparse topologies (ER with (p=0.2), line) exhibit slower convergence and significant oscillations, as delayed updates disrupt consensus ("Slow" for (p=0.2)). The ring topology's cyclic structure mitigates oscillations compared to the line, but its sparsity limits performance relative to dense ER graphs. These findings align with theoretical results (e.g., Lian et al., 2017, page 29), which suggest that sparse graphs increase variance in decentralized optimization. The scalloping behavior indicates that sparse topologies struggle to maintain stable global models, particularly in non-convex settings like Bi-LSTM. The document's exploration of dynamic topologies (Figures 3.36–47, pages 48–52) suggests that time-varying graphs could reduce oscillations by improving connectivity over time (Table 3.2, page 27), offering a potential mitigation strategy.

Other solutions include adaptive learning rate schedules.

Conclusion

This report presented a robust methodology for bearing fault classification using a deep Bidirectional LSTM (Bi-LSTM) model trained under a fully decentralized federated learning framework, DeceFL. The proposed system utilizes the Case Western Reserve University (CWRU) bearing dataset for performance evaluation and experimentation which simulates real-world constraints where data privacy and communication efficiency are critical. By distributing training across six clients using Erdős-Rényi communication topologies. The study demonstrated the impact of varying graph densities on convergence speed and model accuracy.

The architecture and training strategy highlight the effectiveness of combining temporal deep learning models with decentralized optimization algorithms. Notably, the Bi-LSTM network showed strong classification performance across multiple experimental conditions, even in low-connectivity scenarios. This confirms the viability of decentralized federated learning for industrial fault detection tasks where data cannot be centrally pooled due to privacy, bandwidth, or policy constraints.

The results affirm that DeceFL, with its peer-to-peer learning paradigm, offers a scalable, privacy-preserving solution to collaborative machine learning without relying on a central aggregator.

This project serves as a foundational proof of concept that such decentralized architectures can support advanced temporal models for critical applications like predictive maintenance in industrial settings.

Future Work

While the current system demonstrates the capabilities of LSTM-based models within decentralized settings, several promising avenues for future exploration and enhancement remain:

- **Transformer-based Architectures**: Future work will explore replacing LSTM modules with Transformer-based models, such as Vision Transformers (ViT) or Time Series Transformers. These architectures have demonstrated state-of-the-art performance in various sequence modeling tasks due to their ability to capture long-range dependencies through attention mechanisms. Their inclusion could lead to better generalization and faster convergence, especially under Non-IID data settings.
- **Hybrid Models**: Investigating hybrid architectures that combine convolutional layers (for local feature extraction) with Transformer blocks (for global context modeling) may yield a more expressive model suitable for vibration-based signal classification tasks.
- **Federated Hyperparameter Tuning**: Introducing decentralized or federated strategies for hyperparameter tuning, such as learning rate adaptation or layer-wise freeze schedules, could improve model convergence and stability in heterogeneous environments.

In summary, this project lays the groundwork for developing decentralized, privacy-aware diagnostic systems in smart manufacturing. The transition from recurrent to attention-based models represents a logical and exciting progression that could unlock higher accuracy, scalability, and robustness for future industrial AI systems.

Bibliography

[1] Topol, E. (2019). *Deep Medicine: How Artificial Intelligence Can Make Healthcare Human Again*. Basic Books.

[2] Chen, M., Mao, S., & Liu, Y. (2014). Big Data: A Survey. *Mobile Networks and Applications*, 19(2), 171–209.

[3] LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, 521(7553), 436–444.

[4] Voigt, P., & Von dem Bussche, A. (2017). *The EU General Data Protection Regulation (GDPR)*. Springer.

[5] Mehrabi, N., Morstatter, F., Saxena, N., Lerman, K., & Galstyan, A. (2021). A Survey on Bias and Fairness in Machine Learning. *ACM Computing Surveys (CSUR)*, 54(6), 1–35.

[6] Esteva, A., et al. (2019). A Guide to Deep Learning in Healthcare. *Nature Medicine*, 25, 24–29.

[7] Kairouz, P., McMahan, H. B., et al. (2021). Advances and Open Problems in Federated Learning. *Foundations and Trends® in Machine Learning*.

[8] Bonawitz, K., et al. (2019). Towards Federated Learning at Scale: System Design. *SysML*.

[9] Li, T., Sahu, A. K., Talwalkar, A., & Smith, V. (2020). Federated learning: Challenges, methods, and future directions. *IEEE Signal Processing Magazine*, 37(3), 50–60.

[10] Wang, H., et al. (2020). Federated learning with matched averaging. *ICLR*.

[11] Zhao, Y., Li, M., Lai, L., & Suda, N. (2018). Federated learning with non-IID data. *arXiv preprint arXiv:1806.00582*.

[12] Kaissis, G., Makowski, M. R., Rückert, D., & Braren, R. F. (2020). Secure, privacy-preserving and federated machine learning in medical imaging. *Nature Machine Intelligence*, 2, 305–311.

[13] Nasirigerdeh, R., et al. (2021). Overcoming centralized data challenges in healthcare federated learning. *NPJ Digital Medicine*, 4(1), 1–9.

[14] Lin, Y., Han, S., Mao, H., Wang, Y., & Dally, W. J. (2017). Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*.

[15] Wang, S., Tuor, T., Salonidis, T., Leung, K. K., Makaya, C., He, T., & Chan, K. (2019). Adaptive federated learning in resource constrained edge computing systems. *IEEE Journal on Selected Areas in Communications*, 37(6), 1205–1221.

[16] Yuan, Y., Tang, X., Zhou, W., et al. (2019). Data driven discovery of cyber physical systems. *Nature Communications*, 10, 4894.

[17] Yan, L., Zhang, H.-T., Goncalves, J., et al. (2020). An interpretable mortality prediction model for COVID-19 patients. *Nature Machine Intelligence*, 2, 283–288.

[18] Price II, W. N., & Cohen, I. G. (2019). Privacy in the age of medical big data. *Nature Medicine*, 25, 37–43.

[19] Yuan, Y., Liu, J., Jin, D., et al. (2023). DeceFL: A principled fully decentralized federated learning framework. *National Science Open*, 2, 20220043.

[20] DeGrave, A. J., Janizek, J. D., & Lee, S.-I. (2021). AI for radiographic COVID-19 detection selects shortcuts over signal. *Nature Machine Intelligence*, 3, 610–619.

[21] Roberts, M., Driggs, D., Thorpe, M., et al. (2021). Common pitfalls and recommendations for using machine learning to detect and prognosticate for COVID-19 using chest radiographs and CT scans. *Nature Machine Intelligence*, 3, 199–217.

[22] Lo, S. K., Lu, Q., Zhu, L., Paik, H.-Y., Xu, X., & Wang, C. (2021). Architectural patterns for the design of federated learning systems. *arXiv preprint arXiv:2101.02373*.

[23] Konečný, J., McMahan, B., & Ramage, D. (2015). Federated optimization: Distributed optimization beyond the datacenter. *arXiv preprint arXiv:1511.03575*.

[24] Yang, Q., Liu, Y., Chen, T., et al. (2019). Federated machine learning: Concept and Applications.*ACM Transactions on Intelligent Systems and Technology*, 10(2), 1–19.

[25] Konečný, J., McMahan, H. B., Yu, F. X., et al. (2016). Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492*.

[26] McMahan, B., Moore, E., Ramage, D., et al. (2017). Communication-efficient learning of deep networks from decentralized data. *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, 54, 1273–1282.

[27] Tang, Z., Shi, S., & Chu, X. (2020). Communication-efficient decentralized learning with sparsification and adaptive peer selection. *arXiv preprint arXiv:2002.09692*.

[28] Warnat-Herresthal, S., Schultze, H., Shastry, K. L., et al. (2021). Swarm Learning for decentralized and confidential clinical machine learning. *Nature*, 594, 265–270.

[29] Zhu, L., Liu, Z., & Han, S. (2019). Deep leakage from gradients. *Proceedings of the 33rd Conference on Neural Information Processing Systems (NeurIPS)*, 14774–14784.

[30] Zhang, W., et al. (2018). A deep convolutional neural network for bearing fault diagnosis using the *CWRU* dataset. *IEEE Transactions on Industrial Informatics*, 14(12), 5430–5438.

[31] Tang, Z., Shi, S., & Chu, X. (2020). Communication-Efficient Decentralized Learning with Sparsification and Adaptive Peer Selection. *Proceedings of the 40th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 1207–1208.

[32] Lian, X., Zhang, C., Zhang, H., Hsieh, C.-J., Zhang, W., & Liu, J. (2017). Can decentralized algorithms outperform centralized algorithms? A case study for decentralized parallel stochastic gradient descent. *Advances in Neural Information Processing Systems*, 30.

Appendix A

Decentralized Federated Learning Implementation

Note: This code is adapted from the official implementation of DeceFL by Yuan et al. (2023). changes were made to the Global, Local models by proposing Bi-LSTM architecture and 6-clients are used for experimentation. Experiments were conducted by generating random graphs with various connectivity, under varying condition i.e subset of clients in the connections are selected which changes every round resembling real world dynamic connectivity and robust network under IID data distribution.

A.1 Proposed Model Architecture

```
import torch.nn
from torch import nn
from torch.nn import Module
import torch.nn.functional as F

class LSTMNet(nn.Module):
    def __init__(self, input_size=1, hidden_size=64, num_layers=3,
                 num_classes=4):
        super(LSTMNet, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.bidirectional = True
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers,
                            batch_first=True,
                            bidirectional=self.bidirectional)
        self.fc = nn.Linear(hidden_size * 2, num_classes)

    def forward(self, x):
        device = x.device
        num_directions = 2 if self.bidirectional else 1
        h0 = torch.zeros(self.num_layers * num_directions,
              x.size(0), self.hidden_size).to(device)
        c0 = torch.zeros(self.num_layers * num_directions,
              x.size(0), self.hidden_size).to(device)

        out, _ = self.lstm(x, (h0, c0))
        out = out[:, -1, :]
        out = self.fc(out)
        return torch.log_softmax(out, dim=1)
```

A.2 Decentralized Federated Learning - Main Training Loop

```
for epoch in tqdm(range(args.epochs)):
        local_weights = []
        local_weights_grad = []

        print(f'\n | Global Training Round : {epoch+1} |\n')
```

```
for i in range(Num_model):
    global_model_i[i].train()

global_model.train()

for ind,idx in enumerate(idxs_users):
    local_model = LocalUpdate(args=args, dataset=train_dataset,
                              idxs=user_groups[idx])
                              #, logger=logger
    model_i_update, loss = local_model.update_weights(
        model=copy.deepcopy(global_model_i[ind]),
                            global_round=epoch)

    #.state_dict(), model.grad.state_dict()
    w_update = model_i_update.state_dict()
    w_i = global_model_i[ind].state_dict()

    # local_weights_new.append(copy.deepcopy(w_update))

    grad_i = copy.deepcopy(w_update)

    for key_idx, key in enumerate(grad_i.keys()):
        grad_i[key] = (w_i[key] - grad_i[key]).float()

    local_weights.append(copy.deepcopy(w_i))
    local_weights_grad.append(copy.deepcopy(grad_i))
    # train_loss[ind].append(copy.deepcopy(loss))

if args.varying == 1:
    Num_model_sel = int(Num_model/2)
    if args.method == 'er':
        W = erdos_renyi(Num_model_sel, p)
        print('W:\n', W)
    user_sel = random.sample(range(len(idxs_users)),
               Num_model_sel)
    user_else = set(range(len(idxs_users))) - set(user_sel)
    print(f'user_sel:{user_sel}')
    local_weights_sel = [local_weights[i] for i in user_sel]
    local_weights_grad_sel = [local_weights_grad[i]
                             for i in user_sel]
    for ind, idx in enumerate(user_sel):
        w_weights = W[:, ind]
        mu = 1
        global_weights = average_weights_new(local_weights_sel, l
        global_model_i[idx].load_state_dict(global_weights)
    for idx in user_else:
        mu = 1
        global_weights = unsel_weights_new(local_weights[idx]
                         , local_weights_grad[idx], mu)
        global_model_i[idx].load_state_dict(global_weights)

else:
    for ind in range(len(idxs_users)):
        # update global weights
        w_weights = W[:,ind]
        mu = 1
        global_weights = average_weights_new
        (local_weights, local_weights_grad, w_weights, ind, mu)
        # global_weights = average_weights_w(local_weights_new,
        w_weights)
        # update global weights
        global_model_i[ind].load_state_dict(global_weights)

    local_weights_grad_old = local_weights_grad



# Calculate avg training accuracy over all users at every epoch

# update global weights
global_weights_mean = average_weights(local_weights)

# update global weights
global_model.load_state_dict(global_weights_mean)
global_model.eval()



for ind, idx in enumerate(idxs_users):
    list_acc, list_loss = [], []
    global_model_i[ind].eval()
    local_model = LocalUpdate(args=args, dataset=train_dataset,
                              idxs=user_groups[idx])
                              # , logger=logger
    acc, loss = local_model.inference(model=global_model_i[ind])

    train_accuracy[ind].append(acc)
    train_loss[ind].append(loss)
    # print global training loss after every 'i' rounds
    if (epoch+1) % print_every == 0:
        print(f' \nAvg Training Stats after {epoch+1}
              global rounds:')
        print(f'Training Loss :
               {np.mean(np.array(train_loss[ind]))}')
        print('Train Accuracy:
              {:.2f}% \n'.format(100*train_accuracy[ind][-1]))

list_mean_acc, list_mean_loss = [], []
for c in range(args.num_users):
    local_model = LocalUpdate(args=args, dataset=train_dataset,
                              idxs=user_groups[c])
                              #, logger=logger
    acc_mean, loss_mean = local_model.inference
                            (model=global_model)
    list_mean_acc.append(acc_mean)
    list_mean_loss.append(loss_mean)

train_mean_accuracy.append(sum(list_mean_acc)/len(list_mean_acc))
train_mean_loss.append(sum(list_mean_loss)/len(list_mean_loss))

for ind in range(len(idxs_users)):
    # Test inference after completion of training
    test_acc_i, test_loss_i = test_inference(args,
                     global_model_i[ind], test_dataset)
    test_accuracy[ind].append(copy.deepcopy(test_acc_i))
    test_loss[ind].append(copy.deepcopy(test_loss_i))
```

```
            print(f' \n Local Model:{ind} Results after
                        {args.epochs} global rounds of training:')
            print("|---- Avg Train Accuracy: {:.2f}%".
                         format(100*train_accuracy[ind][-1]))
            print("|---- Test Accuracy: {:.2f}%"
                        .format(100*test_acc_i))

        test_acc_global[epoch], test_loss_global[epoch] =
        test_inference(args, global_model, test_dataset)
        print("|---- Test Accuracy Global:
            {:.2f}%".format(100 * test_acc_global[epoch]))
```

A.3 Local Model Training and Testing

```
class LocalUpdate(object):
    def __init__(self, args, dataset, idxs): #, logger
        self.args = args
        # self.logger = logger
        self.trainloader, self.testloader = self.train_val_test(
            dataset, list(idxs))
        self.device = 'cuda' if args.gpu else 'cpu'
        # Default criterion set to NLL loss function
        self.criterion = nn.NLLLoss().to(self.device)

    def train_val_test(self, dataset, idxs):
        """
        Returns train, validation and test dataloaders for a
        given dataset and user indexes.
        """
        # split indexes for train, validation, and test (80, 20)
        idxs_train = idxs[:] #int(0.8*len(idxs))
        idxs_test = idxs[:]

        trainloader = DataLoader(DatasetSplit(dataset, idxs_train),
                                 batch_size=self.args.local_bs,
                                 shuffle=True)
        testloader = DataLoader(DatasetSplit(dataset, idxs_test),
                                batch_size=int(len(idxs_test)/10),
                                shuffle=False)
        return trainloader, testloader

    def update_weights(self, model, global_round):
        # Set mode to train model
        model.train()
        epoch_loss = []

        # Set optimizer for the local updates
        if self.args.optimizer == 'sgd':
            optimizer = torch.optim.SGD(model.parameters(),

            lr=self.args.lr,
            weight_decay=1e-4) #momentum=0.5
        elif self.args.optimizer == 'adam':
            optimizer = torch.optim.Adam(model.parameters(),
            lr=self.args.lr,
            weight_decay=1e-4)

        step_size = self.args.step_size
```

```
StepLR_optimizer = torch.optim.lr_scheduler.
StepLR(optimizer, step_size=step_size, gamma=0.2)

for iter in range(self.args.local_ep):
    batch_loss = 0.0
    for batch_idx, (data, labels) in enumerate
                                (self.trainloader):
        data, labels = data.to(self.device)
                        , labels.to(self.device)

        model.zero_grad()
        log_probs = model(data)
        loss = self.criterion(log_probs, labels.long())
        # loss = self.criterion(log_probs,
                    labels.float().view(-1, 1))
        loss.backward()

        #for name,param in model.named_parameters():
        #    param.grad = grad_avg.grad
        nn.utils.clip_grad_norm_(model.parameters(),
        max_norm=10, norm_type=2)

        optimizer.step()

        if self.args.verbose and (batch_idx % 5 == 0):
            print('| Global Round : {} | Local Epoch : {} |
            [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                global_round, iter+1, (batch_idx+1)
                * len(images),
                    len(self.trainloader.dataset),
                    100. * (batch_idx+1) / len(self.trainloader)
                    , loss.item()))
                # self.logger.add_scalar('loss', loss.item())
                batch_loss += loss.item()
                # batch_loss.append(loss.item())
            epoch_loss.append(batch_loss/(batch_idx + 1))

            StepLR_optimizer.step()

        return model, epoch_loss[-1] #sum(epoch_loss) / len(epoch_loss)

    def inference(self, model):
        """ Returns the inference accuracy and loss.
        """

        model.eval()
        loss, total, correct = 0.0, 0.0, 0.0

        for batch_idx, (data, labels) in enumerate(self.testloader):
            data, labels = data.to(self.device),
            labels.to(self.device)

            # Inference
            outputs = model(data)
            batch_loss = self.criterion(outputs, labels.long())
            # batch_loss = self.criterion(outputs,
                    labels.float().view(-1, 1))
            loss += batch_loss.item()
            # Prediction
            _, pred_labels = torch.max(outputs, 1)
            pred_labels = pred_labels.view(-1)
            correct += torch.sum(torch.eq(pred_labels, labels)).item()
            total += len(labels)

        loss = loss / (batch_idx + 1)
        accuracy = correct/total
        return accuracy, loss
```

A.4 Global model accuracy

```
def test_inference(args, model, test_dataset):
    """ Returns the test accuracy and loss for time-series data. """
    model.eval()
    loss, total, correct = 0.0, 0.0, 0.0
    device = 'cuda' if args.gpu else 'cpu'
    criterion = nn.CrossEntropyLoss().to(device)
    testloader = DataLoader(test_dataset, batch_size=128, shuffle=False)

    with torch.no_grad():
        for batch_idx, (data, labels) in enumerate(testloader):
            data, labels = data.to(device), labels.to(device,
            dtype=torch.long)
            outputs = model(data)
            batch_loss = criterion(outputs, labels)
            loss += batch_loss.item()
            _, pred_labels = torch.max(outputs, 1)
            correct += torch.sum(torch.eq(pred_labels, labels)).item()
            total += len(labels)

    loss = loss / (batch_idx + 1)
    accuracy = correct / total
    return accuracy, loss
```

A.5 Erdős-Rényi graph - generates random graphs with p probability connections

```
def erdos_renyi(n,p):
    while True:
        A = np.random.random((n,n))
        A[A<p] = 1
        A[A<1] = 0
        #symmetrize A, get adjacency matrix
        A = np.triu(A,1); A = A + A.T
        #get laplacian
        L = -A
        for k in range(n):
            L[k,k] = sum(A[k,:])

        eig_L = np.linalg.eig(L)[0]
        pos_eig_0 = np.where(np.abs(eig_L) <1e-5)[0]
        if len(pos_eig_0)==1:
            break
    degrees = np.diag(L)
    D = np.diag(degrees)
    max_degree = np.max(degrees)=
    W = np.eye(n) - 1/(max_degree+1)* (D-A)

    return W
```

A.6 Local Model updation - selected clients and unselected clients

```
def average_weights_new(w, s_avg, w_weight, idx, mu):
    w_avg = copy.deepcopy(w[idx])
    for key in w_avg.keys():
        for i in range(len(w)):
            w_avg[key] = w_avg[key].float() + w[i][key] * w_weight[i]
        w_avg[key] = w_avg[key] - w[idx][key]
        w_avg[key] = w_avg[key] - mu * s_avg[idx][key]
    return w_avg

def unsel_weights_new(w, s_avg, mu):
    w_avg = copy.deepcopy(w)
    for key in w_avg.keys():
        w_avg[key] = w_avg[key] - mu * s_avg[key]
    return w_avg
```

A.7 Averaging weights across all clients for global model

```
def average_weights(w):
    """
    Returns the average of the weights.
    """
    w_avg = copy.deepcopy(w[0])
    for key in w_avg.keys():
        for i in range(1, len(w)):
            w_avg[key] += w[i][key]
        w_avg[key] = torch.true_divide(w_avg[key], len(w)) #true_divide
    return w_avg
```

A.8 Data preprocessing

```
import os
import random
from os import listdir
import numpy as np
import scipy.io as sio
from sklearn.preprocessing import StandardScaler
import torch
from torch.utils import data as Data



def load_data(path, label, load_num):
    totalFileList = [item for item in listdir(path)
        if os.path.splitext(item)[0].endswith(str(load_num))]

    for i, item in enumerate(totalFileList):
```

```
data_dict = sio.loadmat(path+item)

DE_names = [n for n in data_dict.keys()
            if n.endswith('DE_time')]
FE_names = [n for n in data_dict.keys()
            if n.endswith('FE_time')]
if len(DE_names) == 2:
    DE_names = [n for n in DE_names if '99' in n]
    FE_names = [n for n in FE_names if '99' in n]
assert len(DE_names) == 1 and len(FE_names) == 1
DE_name, FE_name = DE_names[0], FE_names[0]

data_single_DE, data_single_FE = data_dict[DE_name]
                                , data_dict[FE_name]

data_single_DE = data_single_DE[:-(len(data_single_DE) % 6000)]
                                    .reshape(-1, 300)
data_single_FE = data_single_FE[:-(len(data_single_FE) % 6000)]
                                    .reshape(-1, 300)

# FFT
l = data_single_DE.shape[1]
data_single_DE = np.abs
                (np.fft.fft(data_single_DE))[:, :l//2] / l * 2
l = data_single_FE.shape[1]
data_single_FE = np.abs
                (np.fft.fft(data_single_FE))[:, :l//2] / l * 2

data_single = np.concatenate(
                    (data_single_DE, data_single_FE), axis=1)

        X.append(data_single)
        Y.append(label * np.ones(data_single.shape[0]))

    X, Y = np.concatenate(X, axis=0), np.concatenate(Y)

    return X, Y


def main():

    # 4 classes , multi class classification
    cats = [
        'normal',
         'B007', # 'B014',  'B021',
        'IR007', #'IR014', 'IR021',
        'OR007', #'OR014', 'OR021'
    ]
    labels = list(range(len(cats)))

    # Train
    X_train, Y_train = [], []
    for p, l in zip(cats, labels):
        for i in range(3):
            X, Y = load_data(path=f'../data/CWRU/{p}/',
                                label=l, load_num=i)
            X_train.append(X[:400])
            Y_train.append(Y[:400])
    # Test
    X_test, Y_test = [], []
    for p, l in zip(cats, labels):
        X, Y = load_data(path=f’../data/CWRU/{p}/’,
                          label=l, load_num=3)
        X_test.append(X[:400])
        Y_test.append(Y[:400])

    X_train = np.concatenate(X_train, axis=0)
    Y_train = np.concatenate(Y_train, axis=0)
    X_test  = np.concatenate(X_test,  axis=0)
    Y_test  = np.concatenate(Y_test,  axis=0)

    ss = StandardScaler()
    X_train = ss.fit_transform(X_train)
    X_test  = ss.transform(X_test)

    np.savez(’../data/CWRU/train’, X_train=X_train, Y_train=Y_train)
    np.savez(’../data/CWRU/test’,  X_test=X_test,   Y_test=Y_test)


if __name__ == ’__main__’:
    main()
    print(’Data processed successfully’)
```

A.9 Dataset preparation

```
def get_dataset(args):
    """
    Returns train and test datasets and a user group dictionary
        for IID distribution.
    """
    if args.dataset == 'cwru':
        data_train = np.load
                    ('../Users/kk749323/my-project/data/CWRU/train.npz')
        data_test = np.load
                    ('../Users/kk749323/my-project/data/CWRU/test.npz')

        x_train, y_train = data_train['X_train'], data_train['Y_train']
        x_test, y_test = data_test['X_test'], data_test['Y_test']

        # Reshape for LSTM input (num_samples,
                                 sequence_length, input_size=1)
        x_train = x_train.astype(np.float32)[:, :, np.newaxis]
        x_test = x_test.astype(np.float32)[:, :, np.newaxis]
        y_train = y_train.astype(np.int64)
                        # Integer labels for classification
        y_test = y_test.astype(np.int64)

        X_train = torch.from_numpy(x_train)
        y_train = torch.from_numpy(y_train)
        X_test = torch.from_numpy(x_test)
        y_test = torch.from_numpy(y_test)

        train_dataset = TensorDataset(X_train, y_train)
        test_dataset = TensorDataset(X_test, y_test)

        # IID data distribution
        user_groups = data_iid(train_dataset, args.num_users)
        print(f'user_groups: {user_groups}')
        return train_dataset, test_dataset, user_groups
    else:
        raise ValueError(f"Dataset {args.dataset} not supported.")
```

A.10 Dataset Distribution - IID

```
# IID data splitting
def data_iid(dataset, num_users):
    """
    Sample I.I.D. client data from dataset
    :param dataset: TensorDataset
    :param num_users: Number of clients
    :return: dict of sample indices
    """
    num_items = int(len(dataset) / num_users)
    dict_users, all_idxs = {}, [i for i in range(len(dataset))]
    for i in range(num_users):
        dict_users[i] = set(np.random.choice
                        (all_idxs, num_items, replace=False))
        all_idxs = list(set(all_idxs) - dict_users[i])

    # Print class distribution for each client
    idx2label = {i: int(y) for i, (x, y) in enumerate(dataset)}
    num_node_class = {}
    for i in range(num_users):
        c = Counter([idx2label[j] for j in dict_users[i]])
        num_node_class[f'C{i+1}'] = list(zip(*sorted(c.items())))[1]
    print("Per node class distribution:\n", num_node_class)

    return dict_users
```

A.11 Main project code

```
import os
import copy
import time
import pickle
import numpy as np
from tqdm import tqdm
import torch
import torch.nn as nn
from torch.utils.data import DataLoader, TensorDataset
import matplotlib
import matplotlib.pyplot as plt
matplotlib.use('Agg')
import random
from collections import Counter

# Arguments parser
def args_parser():
    class Args:
        dataset = 'cwru'
        model = ’lstm’
        gpu = torch.cuda.is_available()
        seed = 42
        num_users = 6
        frac = 1
        epochs = 10
        local_ep = 5
        local_bs = 128
        lr = 0.01
        optimizer = ’sgd’
        num_classes = 4
        iid = 1
        varying = 1
        p = 0.9
        method = ’er’  # Erdős-Rényi method
    return Args()

def exp_details(args):
    print(’\nExperimental details:’)
    print(f’    Model     : {args.model}’)
    print(f’    Optimizer : {args.optimizer}’)
    print(f’    Learning  : {args.lr}’)
    print(f’    Global Rounds   : {args.epochs}\n’)
    print(’    Federated parameters:’)
    print(’    IID’ if args.iid else ’    Non-IID’)
    print(f’    Fraction of users  : {args.frac}’)
    print(f’    Local Batch size   : {args.local_bs}’)
    print(f’    Local Epochs       : {args.local_ep}’)
    print(f’    Connection Probability (p) : {args.p}’)
    print(f'    Varying Connectivity : {args.varying}\n')
    return



def main():
    start_time = time.time()
    path_project = os.path.abspath('..')
    args = args_parser()
    exp_details(args)

    if args.gpu and not torch.cuda.is_available():
        print("CUDA requested but not available. Falling back to CPU.")
        args.gpu = False
    elif args.gpu:
        print(f"Using GPU: {torch.cuda.get_device_name(0)}")
    else:
        print("Using CPU")
    device = 'cuda' if args.gpu else 'cpu'
    args.device = device

    if args.seed:
        np.random.seed(args.seed)
        torch.manual_seed(args.seed)
        torch.cuda.manual_seed(args.seed)
        torch.cuda.manual_seed_all(args.seed)
        torch.backends.cudnn.deterministic = True
        print(f'Set seed {args.seed}\n')

    train_dataset, test_dataset, user_groups = get_dataset(args)
```

```
global_model = LSTMNet(input_size=1, hidden_size=64,
                num_layers=2, num_classes=args.num_classes)
global_model.to(device)
global_model.train()
print(global_model)

train_loss, train_accuracy = [], []
test_loss, test_accuracy = [], []
train_mean_loss, train_mean_accuracy = [], []
print_every = 2
Num_model = int(args.frac * args.num_users)

global_model_i = []
for i in range(Num_model):
    global_model_i.append(copy.deepcopy(global_model))
    global_model_i[i].to(device)
    global_model_i[i].train()
    train_loss.append([])
    train_accuracy.append([])
    test_loss.append([])
    test_accuracy.append([])

p = args.p
if p > 0 and args.method == 'er':
    W = erdos_renyi(Num_model, p)
    print('W:\n', W)

m = max(int(args.frac * args.num_users), 1)
```

```
idxs_users = np.random.choice(range(args.num_users),
                m, replace=False)
groups_num = [len(user_groups[idx]) for idx in idxs_users]
print('\nGroups size:\n', groups_num)

local_weights_grad_old = []
test_acc_global, test_loss_global = {}, {}

{ Main Training loop starts }
{Main Training loop ends }

# Save results
save_dir = f'../save/node{args.num_users}'
os.makedirs(save_dir, exist_ok=True)

file_name = f'{save_dir}/model/defed_global_model_
{args.dataset}_{args.model}_{args.epochs}_C{args.frac}_
iid{args.iid}_E{args.local_ep}_B{args.local_bs}_M{args.method}
_p{args.p}.pt'
os.makedirs(os.path.dirname(file_name), exist_ok=True)
torch.save(global_model.state_dict(), file_name)

for ind, idx in enumerate(idxs_users):
    file_name = f'{save_dir}/model/defed_model{idx}_{args.dataset}_
    {args.model}_{args.epochs}_C{args.frac}_iid{args.iid}_
    E{args.local_ep}_B{args.local_bs}_M{args.method}_p{args.p}.pt'
    os.makedirs(os.path.dirname(file_name), exist_ok=True)
    torch.save(global_model_i[ind].state_dict(), file_name)
```

```
file_name = [
    f'{save_dir}/objects/New_fed_test_each_{args.dataset}
    _{args.model}
    _{args.epochs}_C{args.frac}_iid{args.iid}_E{args.local_ep}
    _B{args.local_bs}_M{args.method}_p{args.p}.pkl',
    f'{save_dir}/objects/New_fed_test_acc_vs_round_
    {args.dataset}_{args.model}_{args.epochs}_C{args.frac}
    _iid{args.iid}_E{args.local_ep}_B{args.local_bs}_M{args.method}
    _p{args.p}.pkl',
    f'{save_dir}/objects/New_fed_train_{args.dataset}
    _{args.model}_{args.epochs}_C{args.frac}
    _iid{args.iid}_E{args.local_ep}_B{args.local_bs}_M{args.method}
    _pkl',
    f'{save_dir}/objects/New_fed_mean_{args.dataset}
    _{args.model}_{args.epochs}_C{args.frac}_iid{args.iid}
    _E{args.local_ep}_B{args.local_bs}_M{args.method}_p{args.p}
    .pkl'
]

for fname in file_names:
    os.makedirs(os.path.dirname(fname), exist_ok=True)

with open(file_names[0], 'wb') as f:
    pickle.dump([test_accuracy, test_loss], f)
with open(file_names[1], 'wb') as f:
    pickle.dump([test_acc_global, test_loss_global], f)
with open(file_names[2], 'wb') as f:
    pickle.dump([train_loss, train_accuracy], f)
with open(file_names[3], 'wb') as f:
    pickle.dump([train_mean_loss, train_mean_accuracy], f)

print(f'\nTotal Run Time: {time.time() - start_time:.2f} seconds')

# Plotting
plt.figure()
plt.title('Training Loss vs Communication Rounds')
for ind in range(len(idxs_users)):
    plt.plot(range(len(train_loss[ind])), train_loss[ind],
    label=f'Client {ind + 1}')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend()
file_name = f'{save_dir}/figures/train_loss_{args.dataset}
_{args.model}_{args.epochs}
_C{args.frac}_iid{args.iid}_E{args.local_ep}_B{args.local_bs}
_M{args.method}_p{args.p}.png'
os.makedirs(os.path.dirname(file_name), exist_ok=True)
plt.savefig(file_name)
plt.close()

plt.figure()
plt.title('Training Accuracy vs Communication Rounds')
for ind in range(len(idxs_users)):
    plt.plot(range(len(train_accuracy[ind])), train_accuracy[ind],
    label=f'Client {ind + 1}')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend()
```

```
file_name = f'{save_dir}/figures/train_acc_{args.dataset}
_{args.model}_{args.epochs}_C{args.frac}_iid{args}_E{args.local}
_B{args.local}_M{args.method}_p{args.p}.png'
os.makedirs(os.path.dirname(file_name), exist_ok=True)
plt.savefig(file_name)
plt.close()

plt.figure()
plt.title('Global Training Loss vs Communication Rounds')
plt.plot(range(len(train_mean_loss)), train_mean_loss, color='r')
plt.ylabel('Loss')
plt.xlabel('Epoch')
file_name = f'{save_dir}/figures/global_train_loss_{args.dataset}
_{args.model}_{args.epochs}_C{args.frac}
_iid{args.iid}_E{args.local_ep}_B{args.local_bs}_M{args.method}
_p{args.p}.png'
os.makedirs(os.path.dirname(file_name), exist_ok=True)
plt.savefig(file_name)
plt.close()

plt.figure()
plt.title('Global Training Accuracy vs Communication Rounds')
plt.plot(range(len(train_mean_accuracy)), train_mean_accuracy,
color='b')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
file_name = f'{save_dir}/figures/global_train_acc_{args.dataset}
_{args.model}
_{args.epochs}_C{args.frac}_iid{args.iid}
```

```
    _E{args.local_ep}_B{args.local_bs}_M{args.method}_p{args.p}.png’
    os.makedirs(os.path.dirname(file_name), exist_ok=True)
    plt.savefig(file_name)
    plt.close()

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