**Federated Learning**

# **Objective**

The task is to explore and implement Federated Learning (FL) techniques to train models on datasets from the EU and USA networks separately. The aim is to assess the effectiveness of FL in handling data from different regions and compare its performance with centralized learning approaches.

# **Understanding Federated Learning**

## **Introduction**

Federated Learning (FL) is a machine learning approach where multiple decentralized devices (clients) collaboratively train a shared model while keeping their data localized. This method enhances privacy and security by ensuring that raw data never leaves the client's device.

## **Key Concepts**

### **1. Federated Averaging (FedAvg)**

* A key algorithm in FL where local models are trained on client devices, and their updates (model weights) are sent to a central server.
* The server aggregates these updates (often by averaging) to update the global model.
* The updated global model is then sent back to the clients for the next round of training.

### **2. Client-Server Architecture**

* **Clients**: Devices or local nodes that hold local data and perform model training.
* **Server**: A central node that coordinates the training process by aggregating model updates from clients and redistributing the updated model.

### **3. Privacy Preservation**

* **Differential Privacy**: Techniques ensuring that individual data points cannot be inferred from the aggregated data.
* **Secure Aggregation**: Cryptographic methods ensuring that the server cannot see individual updates but only the aggregated result.

## **Benefits and Federated Learning**

* **Data Privacy**: Since data remains on the client devices, it mitigates privacy concerns.
* **Reduced Bandwidth Usage**: Only model updates, not raw data, are transmitted.
* **Compliance with Regulations**: Helps in adhering to data protection laws like GDPR.

## **Common Techniques and Frameworks**

### **1. TensorFlow Federated (TFF)**

* An open-source framework for experimenting with FL.
* Provides tools to simulate the FL environment, including client/server setup.

### **2. PySyft**

* An open-source library for secure and private machine learning.
* Extends PyTorch to enable FL, Differential Privacy, and Encrypted Computation.

### **3. Flwr (Flower)**

* An open-source framework for federated learning.
* Provides a flexible and easy-to-use system for FL with support for various machine learning frameworks, including TensorFlow, PyTorch, and more.

# **Data Preprocessing for both Datasets**

Data preprocessing is an important step, as it refers to the cleaning, transforming, and integrating of data in order to make it ready for analysis. The goal of data preprocessing is to improve the quality of the data and to make it more suitable for the specific task.

**We have used channel 1 (GSNR\_1) as the target variable.**

## **Library Imports**

The code imports necessary libraries, including pandas for data manipulation, MinMaxScaler from sklearn for normalization, and torch components for creating datasets and data loaders.

## **Data Normalization**

The normalize\_data function scales both feature columns and the target column using MinMax scaling. This method transforms the data into a range between 0 and 1, ensuring all features contribute equally to the model's performance. The function returns the normalized feature set and target variable as DataFrames.

## **Custom Train-Test Split**

The custom\_train\_test\_split function splits the data path-wise, recognizing that the dataset changes every 3000 samples. It splits each block into training and testing sets, ensuring that the data is correctly partitioned. This approach is particularly useful for datasets with structured changes or segments.

## **Data Extraction and Preparation**

The get\_data function reads data from an Excel file, normalizes it, and splits it into training and testing sets using the custom split logic. The function identifies specific attribute columns and the target variable (GSNR\_1 in this case), excluding frequency due to its low correlation with the target.

## **Federated Learning Data Preparation**

The prepare\_fldata function prepares data for federated learning by partitioning the training data into several parts (clients), each with its own training and validation split. It also creates DataLoaders for these partitions, facilitating the training process in a federated learning setup. The function returns lists of DataLoaders for training and validation, as well as a DataLoader for testing.

## **Data Preprocessing Overview**

The note outlines the preprocessing steps: examining the dataset's initial rows, checking for null values, splitting the dataset based on structured paths, and scaling features using MinMaxScaler. The emphasis on feature scaling highlights its importance in ensuring all features contribute equally to the model's learning process, which is crucial for models sensitive to feature magnitudes, such as neural networks.

# **Federated Learning Implementation**

## **Introduction**

This report outlines the implementation of a Federated Learning (FL) system using the Flower framework (flwr). The system is designed to train a simple neural network across multiple clients, each holding their local data. The implementation is split into three main components: model.py, main.py, and client.py, each handling different aspects of the FL process.

## **Configuration Management and Data Preparation**

### **Key Libraries**

* **hydra**: Manages and overrides configurations at runtime.
* **omegaconf**: Provides structured configuration handling.

### **Main Functions**

* **@hydra.main**: Initializes the configuration using the Hydra framework, allowing for easy management and modification of parameters. The configuration is loaded from a specified path and printed in YAML format for easy debugging.
* **prepare\_fldata**: A custom function to prepare federated data. It takes the number of clients and batch size from the configuration and returns DataLoaders for training, validation, and testing. This function handles data normalization, splitting, and conversion into PyTorch tensors.

### **Overview**

The model.py script is responsible for managing configurations and preparing data for the federated learning process. It uses Hydra for flexible configuration management, allowing for easy modification of parameters such as the number of clients and batch size. The data preparation function, prepare\_fldata, ensures that data is correctly partitioned and ready for distributed training across multiple clients.

## **Model Definition and Training**

### **Key Libraries**

* **torch**: The main library used for creating and training the neural network.

### **Main Components**

* **Net**: Defines a simple fully connected neural network suitable for regression tasks. The model consists of three linear layers with ReLU activation functions applied between them.
* **train**: A function that handles the training loop for the network. It uses Mean Squared Error (MSE) as the loss function and optimizes the model parameters using the provided optimizer. The function iterates over the data for a specified number of epochs, updating the model weights to minimize the loss.
* **test**: A function to evaluate the network on the test set. It calculates the MSE loss over the entire dataset, providing an average loss metric for model evaluation.

### **Overview**

The main.py script defines the neural network architecture and implements the training and evaluation procedures. The Net class represents the model structure, while the train and test functions facilitate training and validation, respectively. These components are crucial for the local training of models on client devices.

## **Client Implementation**

### **Key Components**

**FlowerClient**: Inherits from fl.client.NumPyClient and represents a client in the FL setup. It includes methods for setting model parameters, local training (fit), and evaluation (evaluate).

* **set\_parameters**: Sets the model parameters using the provided state dictionary.
* **fit**: Trains the model using local data, updates the model parameters, and returns them along with the number of samples used and an optional dictionary for metrics.
* **evaluate**: Evaluates the model's performance on local validation data and returns the loss, number of samples, and an optional metrics dictionary.

**generate\_client\_fn**: A factory function that returns a function to create instances of FlowerClient for each client. This function ensures that each client is initialized with its own local data.

### **Overview**

The client.py script encapsulates the functionality required for a client in the federated learning framework. The FlowerClient class implements the methods necessary for communication with the server, including parameter updates, local training, and evaluation. The generate\_client\_fn function provides a mechanism to create and manage multiple clients in the simulation.

## **Conclusion**

This implementation demonstrates a comprehensive approach to federated learning using the Flower framework. It includes:

* **Configuration Management**: Leveraging Hydra for dynamic configuration handling.
* **Data Preparation**: Custom functions for data normalization and splitting, ensuring data is correctly partitioned across clients.
* **Model Architecture and Training**: A simple neural network model with training and evaluation loops.
* **Client Simulation**: A custom client class for local training and evaluation, along with a factory function to manage multiple clients.

This system is designed for scalability, allowing easy addition of more clients and modification of training parameters. The use of Flower and PyTorch ensures flexibility and compatibility with various machine learning models and datasets. The project serves as a robust foundation for further exploration and experimentation in the field of federated learning.

# **Execution Instructions**

## **Running Federated Learning Setup**

To successfully execute the Federated Learning (FL) implementation, follow these steps:

### **1. Open Terminal in the Project Directory**

* Navigate to the directory where all the files (dataProcessing.ipynb, model.py, server.py, client.py, and client2.py) are located.

### **2. Data Preparation**

* Run the dataProcessing.ipynb Jupyter notebook file to preprocess the data. This step involves loading the dataset DataSet\_EU\_3k\_5k.xlsx and preparing it for the federated learning process. Ensure that all necessary libraries are installed and the data is correctly processed.

### **3. Initialize Configuration**

* In the terminal, run model.py to set up the configuration for the FL process. This script initializes the necessary configurations, including the number of clients, batch size, and data loaders for training, validation, and testing.

### **4. Start the FL Server and Clients**

* **Terminal 1**: Start the FL server by running server.py. This script will initiate the server that coordinates the federated learning process among the clients.
* **Terminal 2**: Start the first client by running client.py. This client will receive model updates, train locally on its data, and send updates back to the server.
* **Terminal 3**: Start the second client by running client2.py. Similar to the first client, this will participate in the FL process, contributing to the overall model improvement.

# **Results Overview**

## **Loss and Accuracy Over Rounds**

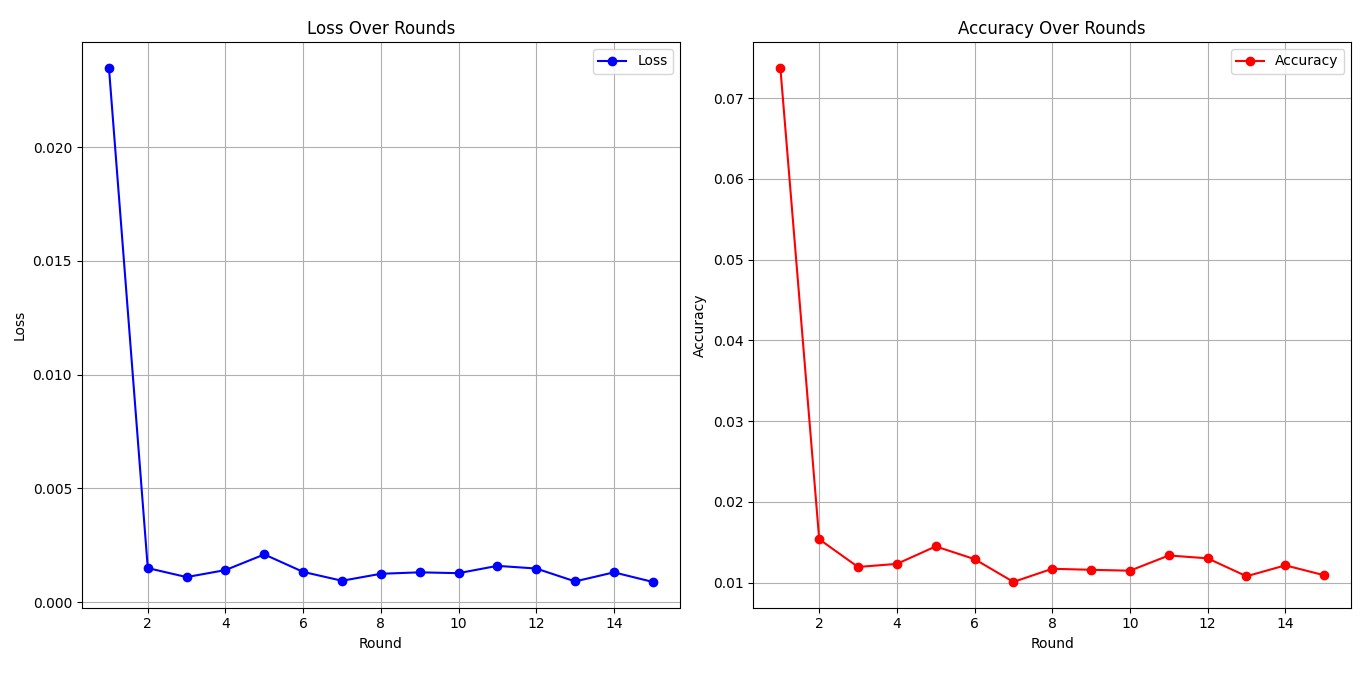
The training results are depicted in the screenshot below, showing the metrics over multiple communication rounds:

### **1. Loss Over Rounds**

The left graph displays the loss metric over several FL rounds. Initially, the loss starts relatively high but quickly decreases, indicating effective learning and convergence. The loss stabilizes with minimal fluctuation after the second round, suggesting that the model has learned a reasonable representation of the data.

### **2. Accuracy Over Rounds**

The right graph shows the accuracy metric over the same rounds. Like the loss graph, accuracy improves significantly after the first round, though the initial accuracy value is low. The accuracy stabilizes quickly, reflecting consistent performance in the subsequent rounds.



## **Interpretation**

* The quick reduction in loss and stabilization of accuracy suggest that the model efficiently learned from the distributed data across the clients. The results imply successful coordination between the server and clients, with each client contributing to the model's improvement.
* The stable values in the later rounds indicate that the model has reached a point where further training does not significantly improve the performance, suggesting a potential convergence.

## **Conclusion**

This federated learning setup successfully demonstrates the potential of distributed training across multiple clients. The results, shown in the graphs, confirm that the model effectively reduces loss and improves accuracy, stabilizing as training progresses. This setup can be further extended with more clients or additional rounds to enhance performance or adapt to new data.

**THE END**