Heterogeneous Federated Learning

STUDY ORIENTED PROJECT

Submitted by:

Name of the Students	BITS ID
Jai Shrree Raam VR	**2022A8PS1264P**
Madhav Krishna	**2022A3PS0508P**
Himanshu Kumar	**2022A8PS0557P**
Lovenya Jain	**2021B4AA1732P**

Submitted to:

Dr. Meetha V. Shenoy, Assistant Professor
Department of Electrical & Electronics Engineering

Pilani Campus
Birla Institute of Technology & Science, Pilani
Rajasthan, India

Table of Contents

CODE: lovenya/federated-learning-sop

Introduction

Objective

The objective of this project is to develop a federated learning system for training a ML model across decentralized clients, including resource-constrained devices like Raspberry Pi and Jetson Nano, while preserving data privacy. The project aims to enhance the model's accuracy and robustness by implementing the FedAvg strategy to aggregate updates from clients and iteratively improve the global model without sharing raw data. By leveraging real-time data processing and edge computing capabilities, the system strives to address challenges in decentralized training, such as limited computational power, network constraints, and maintaining model performance. This approach paves the way for scalable and efficient deployment of federated learning in real-world scenarios.

What is Federated Learning?

Federated Learning (FL) is a decentralized machine learning approach where multiple devices collaboratively train a shared global model while keeping data localized, ensuring privacy and security. Instead of sharing raw data, devices send model updates like weights or gradients to a central server, which aggregates them to improve the global model iteratively. This method enables scalable and privacy-preserving training, leveraging the computational power of edge devices. Widely applied in areas like healthcare, smartphones, and IoT, FL addresses privacy concerns and reduces data transfer needs but faces challenges such as device heterogeneity, non-IID data distributions, and communication overhead.

Significance of Using Edge Devices (Raspberry Pi and Jetson Nano)

Using edge devices like Raspberry Pi and Jetson Nano in Federated Learning is significant due to their ability to process data locally, reducing latency and preserving privacy. They enable real-time computations and are cost-effective, making them ideal for resource-constrained environments.

- Data Privacy: Local processing ensures sensitive data never leaves the device.
- Real-Time Computation: Low latency for quick responses in edge applications.
- Cost Effectiveness: Affordable hardware for scalable deployment in distributed systems.

System Architecture

Federated Learning Framework (Flower)

Flower framework is a comprehensive and versatile framework for federated learning. Its architecture supports heterogeneous environments and aims at effective testing of federated learning systems. In our project, Flower enables us to communicate with the central server and container models used by the clients (Raspberry Pi and Jetson Nano) to aggregate model updates effectively during training. Some salient points of Flower include interaction strategies between clients and servers that can be customized, availability of different ML frameworks, and seamless integration into edge nodes.

Overview of the Client-Server Architecture

The client-server architecture underpins the federated learning setup.

- **Clients**: The edge devices (Raspberry Pi and Jetson Nano) are clients. Each device uses its own dataset to train the model locally and only sends the model's parameters to the server.
- **Server**: The parameters sent to the central server by the clients are combined at the server and sent back to the relevant clients in the round further improving the global model. This process continues in rounds until the model is able to perform to expected standards.This architecture maintains the privacy of data while allowing collaborative learning among the devices that are distributed across locations.

Edge Devices Used: Raspberry Pi and Jetson Nano

Two edge devices were employed in this system:

- **Raspberry Pi**: A cost-effective, widely used microcomputer capable of running basic deep learning models. It was configured to perform local training on its dataset and contribute to the federated learning process.
- **Jetson Nano**: A high-performance edge device with a GPU, suitable for more complex computations. Its role in the system was to accelerate training and evaluate how hardware differences affect model performance.
  Both devices were connected wirelessly to the central server, ensuring real-time communication for parameter exchange.

Model

In this project, we use **MobileNetV3 Small** from the torchvision library as the base model for image classification, specifically for the CIFAR-10 dataset. MobileNetV3 is a highly efficient deep learning architecture designed for mobile and embedded devices, offering a good trade-off between accuracy and computational efficiency. It is particularly well-suited for scenarios where computational resources (e.g., memory, processing power) are constrained.

Key Features of MobileNetV3:

1. **Efficient Architecture**: MobileNetV3 employs lightweight convolutions (Depthwise Separable Convolutions) to reduce computation, making it ideal for edge devices like Raspberry Pi or Jetson Nano in the federated learning setup.
2. **Pre-trained on ImageNet**: MobileNetV3 is often pre-trained on the ImageNet dataset, which helps achieve faster convergence on smaller datasets like CIFAR-10 by leveraging learned feature representations.
3. **Compact Model**: The small variant of MobileNetV3 has a reduced number of parameters compared to the large variant, making it suitable for resource-constrained environments.

Implementation and Setup

Implementation on Raspberry Pi

The implementation on Raspberry Pi involved configuring it as a federated learning client:

1. **Model Training**:
    - The CNN model was trained on the Raspberry Pi using a locally stored dataset. The training loop was optimized to account for the limited computational resources and memory.
    - Data augmentation and preprocessing techniques were applied to enhance model generalizability.
2. **Parameter Updates**:
    - After training for a specified number of epochs, the Raspberry Pi transmitted the model updates (parameters) to the server using the Flower framework.
3. **Integration with the Server**:
    - The server aggregated the updates from the Raspberry Pi and other clients to generate a new global model. This model was then sent back to the Raspberry Pi for the next round of training.
4. **Performance Monitoring**:
    - The training loss and accuracy on the Raspberry Pi were logged for analysis.
5. **Inference:**
    - Code for inference was written to test the updated/received model by performing inference using an RTSP stream from the camera.

This setup demonstrated the feasibility of deploying federated learning on lightweight edge devices while maintaining data privacy and achieving reasonable model performance.

Implementation on Jetson Nano

The corresponding implementation on Jetson Nano is as follows:

1.Model Training:
The CNN model was trained on the Jetson Nano using a locally stored dataset. The training loop was optimized to account for the limited computational resources and memory. Data augmentation and preprocessing techniques were applied to enhance model generalizability.

2.Parameter Updates:
After training for a specified number of epochs, the Jetson Nano transmitted the model updates (parameters) to the server using the Flower framework.

3.Integration with the Server:
The server aggregated the updates from the Jetson Nano and other clients to generate a new global model. This model was then sent back to the Jetson Nano for the next round of training.

4.Performance Monitoring:
The training loss and accuracy on the Server were logged for analysis.

5.Inference:

Code for inference was written to test the updated/received model by performing inference using an RTSP stream from the camera.

Following steps were undertaken for the setup and implementation of code:

Since the Jetson Nano is using a much older version of Ubuntu, we cannot obtain the later python versions as easily. To work around this, docker can be used as it lets us create a container with the correct version of packages required, especially python.

Step 1: Install Docker

1. **Update the package list**:
   ```
   sudo apt-get update
   ```
2. **Install required packages**:
   ```
   sudo apt-get install apt-transport-https ca-certificates curl software-properties-common
   ```
3. **Add Docker's official GPG key**:
   ```
   curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
   ```
4. **Add the Docker repository**:
   ```
   sudo add-apt-repository "deb [arch=arm64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
   ```
5. **Update the package list again**:
   ```
   sudo apt-get update
   ```
6. **Install Docker**:
   ```
   sudo apt-get install docker-ce
   ```
7. **Start and enable Docker**:
   ```
   sudo systemctl start docker
   sudo systemctl enable docker
   ```

Step 2: Run Python 3.11 Container with Named Volume

1. **Pull the Python 3.11 Docker image**:
   ```
   sudo docker pull python:3.11
   ```
2. **Create a named volume**:
   ```
   sudo docker volume create flwr_data
   ```
3. **Run the container**:
   ```
   sudo docker run -d --network host -v flwr_data:/root --name flwr-container python:3.11 tail -f /dev/null
   ```

Step 3: Install Git and Clone the Flower Repo

1. **Access the running container**:
   `sudo docker exec -it flwr-container bash`
2. **Install Git**:
   `apt-get update && apt-get install -y git`
3. **Clone the Flower repository**:
   `git clone https://github.com/adap/flower.git`

Step 4: Run the Client with the Correct Arguments

1. **Navigate to the cloned repository**:
   `cd flower/examples/client_pytorch`
2. **Run the client script with the appropriate arguments**:

   `python client.py --cid <your_client_id> --server_address <your_server_address>`

1. Replace `<your_client_id>` with your client ID and `<your_server_address>` with the server address you intend to connect to.

    In our case this was:
        python3 client_pytorch.py --cid= 0--server_address=192.168.0.110 for Jetson Nano

Note:

- In this demo, we have used CIFAR-10, a well-known image classification dataset that consists of 60,000 RGB images, each measuring 32x32 pixels. The dataset is divided into 10 categories, including examples like cars, birds, and airplanes.
- The server and client will automatically download the data.
- The devices and the camera must be connected to the same network.

Once everything is setup,we can start it from by

1. Sudo docker ps -a
    a. This will list all the dockers.
    b. Copy the container id of the image and run it by -> sudo docker start <container-id>
    c. Then run this command sudo docker exec -it <container-name> bash.In our case container name was my-python-container
    d. Our cloned repo is at root/embedded-devices so run cd root/embedded-devices
    e. There we can find our cloned repo.

Setting up on Raspi

1. Make sure the OS is up to date
2. Run - `sudo apt update` to check for updates and update is required `sudo apt upgrade -y` and then reboot your Rpi `sudo reboot`
3. Run the following command to clone the req repo
4. `git clone --depth=1 https://github.com/adap/flower.git && mv flower/examples/embedded-devices . && rm -rf flower && cd embedded-devices`
5. Since we are using Pytorch we will install the requirements for it
   `pip3 install -r requirements_pytorch.txt`
   If you wish to use tensorflow - `pip3 install -r requirements_tf.txt`
6. Then install flwr using `pip install flwr`
7. We had to debug the client code for our needs, update the client_pytorch.py as given below,
   https://github.com/JaiVR/fedLearn/blob/temp-branch/client_pytorch.py
8. The Raspi client is now ready.
   Run this `python3 client_pytorch.py --cid=<CLIENT_ID> --server_address=<SERVER_ADDRESS>`
   Eg. `<SERVER_ADDRESS> = 192.162.0.110:8080`
   Make sure to include that 8080
   Here make sure you give a unique `CLIENT_ID` for Raspi and Jetson nano.

NOTE: all this is done in a virtual environment and make sure you change the venv in the python interpreter if you're using thonny.

Server Implementation

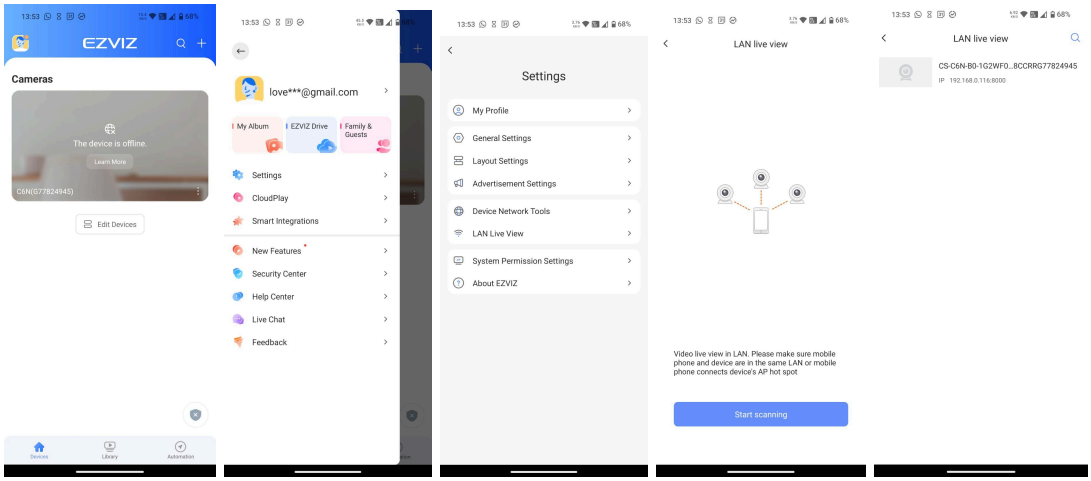
The server was setup in a personal laptop and the server code is given below,
https://github.com/JaiVR/fedLearn/blob/temp-branch/server_fedlearn.py
CLI command
python server.py --rounds **3** --min_num_clients **2** --sample_fraction **1**.0

Camera Setup for rtsp stream

- The devices and the camera must be connected to the same network.

To set up the Ezviz app and enable RTSP on your camera, please follow these steps:

1. **Download the Ezviz App**: Install the Ezviz app on your smartphone.
2. **Register the Camera**: If this is your first time using the app, you will need to register your camera. Follow the on-screen instructions to complete the registration process.
3. **Navigate to Settings**: Once registered, open the app and navigate to the settings menu.
4. **Access LAN Live View**: Within the settings menu, locate and select the "LAN Live View" option.
5. **Start Scanning**: Click on "Start Scanning" to detect available devices on your network.
6. **IP Settings**: After scanning, go to "IP Setting" and then select "Local Service Settings."
7. **Enable RTSP**: In the Local Service Settings, enable the RTSP option



Dataset

The CIFAR-10 dataset consists of 60,000 32x32 color images in 10 classes, with 6,000 images per class. The dataset is split into 50,000 training images and 10,000 test images.

For testing accuracy of each class per client

- **Total Images:** 10,000/20=500
- **Number of Classes:** 10
- **Images per Class:** 50

So, for each class in the CIFAR-10 test dataset, there are **50 images**.

But on server testing happens on **all 10,000 images.**

Classes in CIFAR-10:

Airplane, Automobile, Bird, Cat, Deer, Dog, Frog, Horse, Ship, Truck.

Initialization

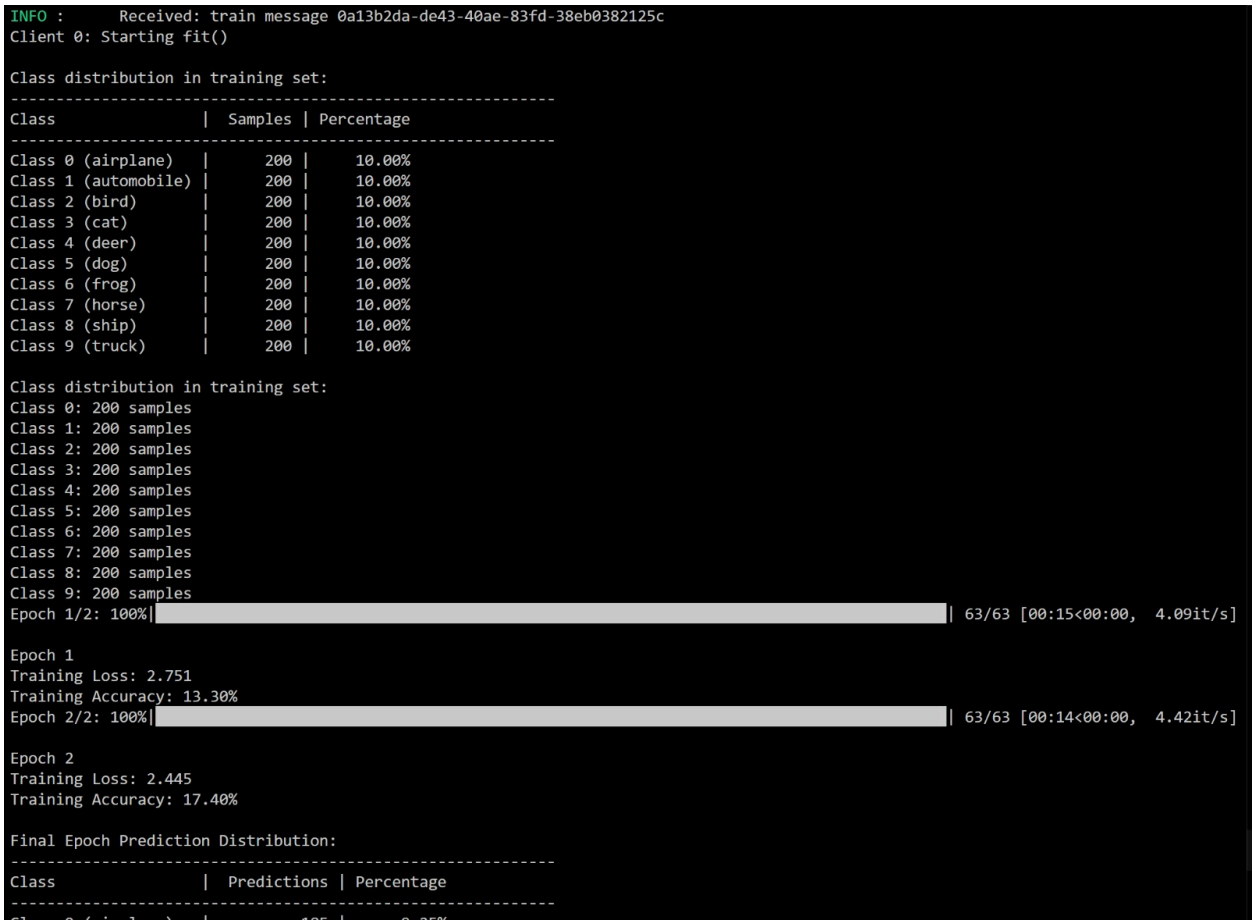
- The server loaded pre-existing model parameters and started the Flower server.
- It performed an initial evaluation of the global model (before training) on the 20% test dataset:

Round 1

- The server sampled 2 clients (as specified) for training.
- After receiving the trained parameters, it aggregated them into a global model.
- The aggregated model was evaluated on the server-side test dataset

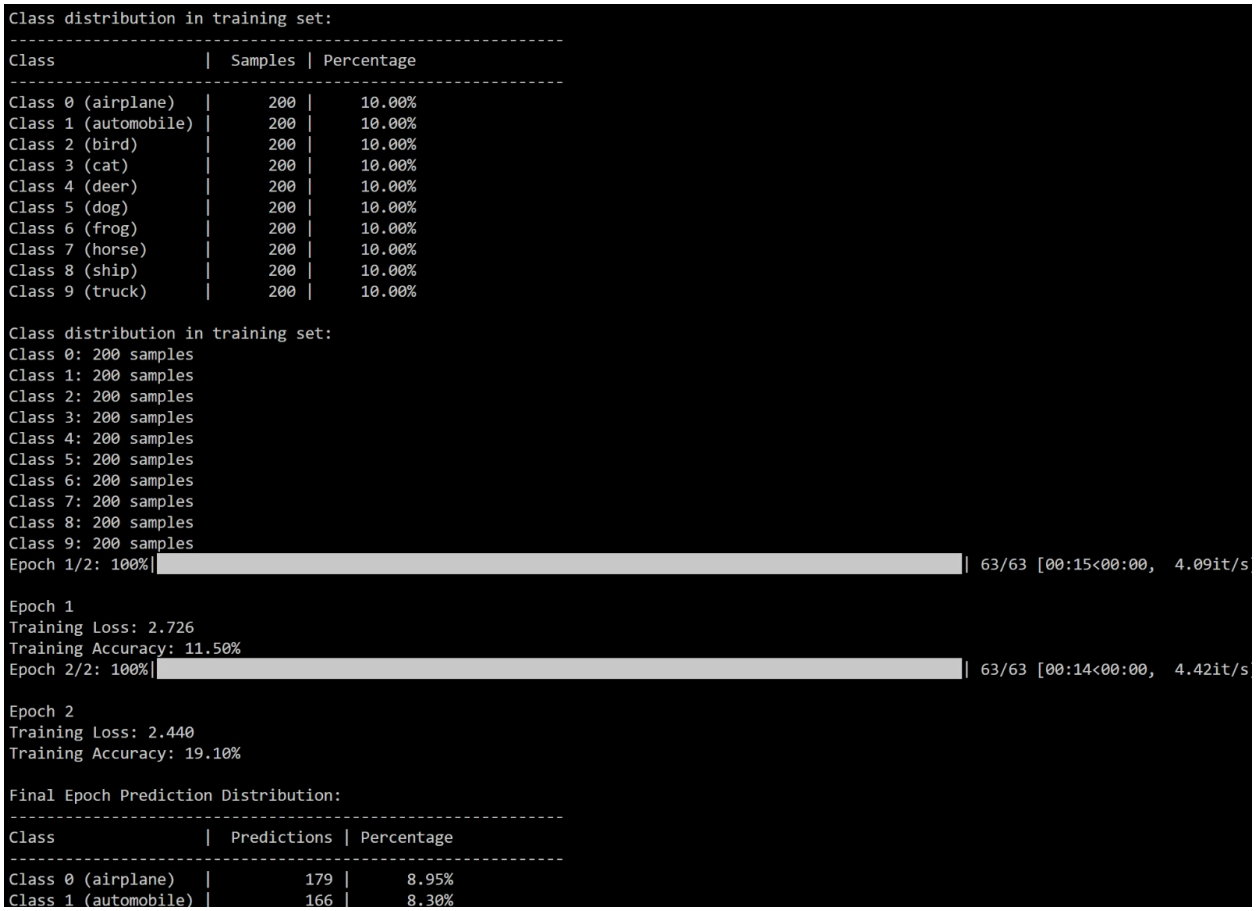
Results

Client 1 Output (Training)



```
INFO :      Received: train message 0a13b2da-de43-40ae-83fd-38eb0382125c
Client 0: Starting fit()

Class distribution in training set:
------------------------------------------------------------
Class               |  Samples | Percentage
------------------------------------------------------------
Class 0 (airplane)  |      200 |     10.00%
Class 1 (automobile) |      200 |     10.00%
Class 2 (bird)      |      200 |     10.00%
Class 3 (cat)       |      200 |     10.00%
Class 4 (deer)      |      200 |     10.00%
Class 5 (dog)       |      200 |     10.00%
Class 6 (frog)      |      200 |     10.00%
Class 7 (horse)     |      200 |     10.00%
Class 8 (ship)      |      200 |     10.00%
Class 9 (truck)     |      200 |     10.00%

Class distribution in training set:
Class 0: 200 samples
Class 1: 200 samples
Class 2: 200 samples
Class 3: 200 samples
Class 4: 200 samples
Class 5: 200 samples
Class 6: 200 samples
Class 7: 200 samples
Class 8: 200 samples
Class 9: 200 samples
Epoch 1/2: 100%| | 63/63 [00:15<00:00,  4.09it/s]

Epoch 1
Training Loss: 2.751
Training Accuracy: 13.30%
Epoch 2/2: 100%| | 63/63 [00:14<00:00,  4.42it/s]

Epoch 2
Training Loss: 2.445
Training Accuracy: 17.40%

Final Epoch Prediction Distribution:
------------------------------------------------------------
Class               |  Predictions | Percentage
------------------------------------------------------------
```

Client 2 Output (Training)



Client 1 Output

(**Post-evaluation**, before sending weights to the server for aggregation)

```
============================================================
CLIENT-SIDE EVALUATION METRICS
============================================================

Overall Test Accuracy: 21.80%
Average Loss: 2.1311

Class-wise Performance:
------------------------------------------------------------
Class                |  Samples |  Correct |  Accuracy
------------------------------------------------------------
Class 0 (airplane)   |       50 |       16 |    32.00%
Class 1 (automobile) |       50 |       16 |    32.00%
Class 2 (bird)       |       50 |       12 |    24.00%
Class 3 (cat)        |       50 |       10 |    20.00%
Class 4 (deer)       |       50 |        7 |    14.00%
Class 5 (dog)        |       50 |       10 |    20.00%
Class 6 (frog)       |       50 |       11 |    22.00%
Class 7 (horse)      |       50 |       11 |    22.00%
Class 8 (ship)       |       50 |        7 |    14.00%
Class 9 (truck)      |       50 |        9 |    18.00%

Model Prediction Distribution:
------------------------------------------------------------
Class                |  Predictions | Percentage
------------------------------------------------------------
Class 0 (airplane)   |           59 |     11.80%
Class 1 (automobile) |           49 |      9.80%
Class 2 (bird)       |           69 |     13.80%
Class 3 (cat)        |           47 |      9.40%
Class 4 (deer)       |           59 |     11.80%
Class 5 (dog)        |           40 |      8.00%
Class 6 (frog)       |           50 |     10.00%
Class 7 (horse)      |           58 |     11.60%
Class 8 (ship)       |           20 |      4.00%
Class 9 (truck)      |           49 |      9.80%

Client 0 Class-wise Accuracy:
```

Client 2 Output

(**Post-evaluation**, before sending weights to the server for aggregation)

```
============================================================
CLIENT-SIDE EVALUATION METRICS
============================================================

Overall Test Accuracy: 22.60%
Average Loss: 2.1236

Class-wise Performance:
------------------------------------------------------------
Class                |  Samples |  Correct |  Accuracy
------------------------------------------------------------
Class 0 (airplane)   |       50 |       13 |    26.00%
Class 1 (automobile) |       50 |       19 |    38.00%
Class 2 (bird)       |       50 |       11 |    22.00%
Class 3 (cat)        |       50 |       11 |    22.00%
Class 4 (deer)       |       50 |       14 |    28.00%
Class 5 (dog)        |       50 |        7 |    14.00%
Class 6 (frog)       |       50 |        9 |    18.00%
Class 7 (horse)      |       50 |       10 |    20.00%
Class 8 (ship)       |       50 |        8 |    16.00%
Class 9 (truck)      |       50 |       11 |    22.00%

Model Prediction Distribution:
------------------------------------------------------------
Class                |  Predictions | Percentage
------------------------------------------------------------
Class 0 (airplane)   |           55 |     11.00%
Class 1 (automobile) |           64 |     12.80%
Class 2 (bird)       |           56 |     11.20%
Class 3 (cat)        |           50 |     10.00%
Class 4 (deer)       |           63 |     12.60%
Class 5 (dog)        |           29 |      5.80%
Class 6 (frog)       |           48 |      9.60%
Class 7 (horse)      |           49 |      9.80%
Class 8 (ship)       |           43 |      8.60%
Class 9 (truck)      |           43 |      8.60%

Client 1 Class-wise Accuracy:
```

Server Output

(Post-evaluation, after aggregation)

```
============================================================
SERVER-SIDE EVALUATION METRICS
============================================================

Overall Test Accuracy: 11.20%
Average Loss: 2.3026

Class-wise Performance:
------------------------------------------------------------
Class                |  Samples |  Correct |   Accuracy
------------------------------------------------------------
Class 0 (airplane)   |     1000 |       28 |      2.80%
Class 1 (automobile) |     1000 |       59 |      5.90%
Class 2 (bird)       |     1000 |        5 |      0.50%
Class 3 (cat)        |     1000 |      524 |     52.40%
Class 4 (deer)       |     1000 |        2 |      0.20%
Class 5 (dog)        |     1000 |       21 |      2.10%
Class 6 (frog)       |     1000 |        9 |      0.90%
Class 7 (horse)      |     1000 |      273 |     27.30%
Class 8 (ship)       |     1000 |      172 |     17.20%
Class 9 (truck)      |     1000 |       27 |      2.70%

Model Prediction Distribution:
------------------------------------------------------------
Class                |  Predictions | Percentage
------------------------------------------------------------
Class 0 (airplane)   |          186 |      1.86%
Class 1 (automobile) |          678 |      6.78%
Class 2 (bird)       |          149 |      1.49%
Class 3 (cat)        |         4053 |     40.53%
Class 4 (deer)       |           34 |      0.34%
Class 5 (dog)        |          357 |      3.57%
Class 6 (frog)       |          207 |      2.07%
Class 7 (horse)      |         3060 |     30.60%
Class 8 (ship)       |          970 |      9.70%
Class 9 (truck)      |          306 |      3.06%

================================================
ROUND 1 COMPLETE
================================================
```

Federated Learning Workflow

For **each round** (total of 3 rounds):

1. **Server**:
    - Sends the current global model parameters to all clients.
2. **Clients**:
    - Receive the global model parameters.
    - Train the model locally using their local dataset for 10 epochs.
    - **Local Training**:
        - Each client goes through their 5,000 images 10 times (due to 10 epochs).
        - Total images processed per client per round: 5,000 images×10 epochs=50,000 images
    - Send updated model parameters back to the server.
3. **Server**:
    - Receives updated parameters from all clients.
    - Aggregates the parameters (e.g., using Federated Averaging).
    - Updates the global model.

Total Training Over 3 Rounds

- **Per Client**:
    - Total images processed: 50,000 images per round×3 rounds=150,000 images (with repetitions).
- **Entire Federated System**:
    - Total images processed across all clients: 150,000 images per client×10 clients=1,500,000 images (with repetitions).

Federated Learning Implementation: Technical Documentation

1. Hyperparameter Choices & Concepts

1.1 Optimizer Configuration (AdamW)

The implementation uses AdamW optimizer with specific configurations:

```
optimizer = torch.optim.AdamW(
        self.model.parameters(),
        lr=0.0001,
        weight_decay=0.01
)
```

Key Concepts:

- **AdamW vs Adam**:
  - AdamW is a variant of Adam that implements correct weight decay regularization
  - It decouples weight decay from gradient updates, leading to better generalization
  - More effective than L2 regularization in Adam

Parameter Analysis:

- **Learning Rate (0.0001)**:
  - Conservative learning rate chosen for stability
  - Small enough to ensure convergence in federated setting
  - Helps prevent client models from diverging too much from global model
- **Weight Decay (0.01)**:
  - Moderately strong regularization
  - Helps prevent overfitting on individual client datasets
  - Encourages simpler models by penalizing large weights

1.2 Learning Rate Scheduling (ReduceLROnPlateau)

```
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
        optimizer,
        mode="max",
        factor=0.1,
        patience=2,
        verbose=True
)
```

Concept Explanation:

- **ReduceLROnPlateau**:
  - Adaptive learning rate scheduler that responds to model performance
  - Reduces learning rate when model stops improving
  - Helps overcome plateaus in training

Parameters Explained:

- **Mode="max"**:
  - Monitors metrics for maximization (accuracy in this case)
  - Alternative would be "min" for loss monitoring
- **Factor=0.1**:
  - Multiplication factor for learning rate reduction
  - When triggered, LR is multiplied by 0.1 (reduced to 10% of current value)
  - Example: 0.0001 → 0.00001
- **Patience=2**:
  - Number of epochs to wait before reducing learning rate
  - Prevents premature LR reduction due to temporary fluctuations
  - Balance between responsiveness and stability

1.3 Label Smoothing

criterion = nn.CrossEntropyLoss(label_smoothing=0.1)

Concept Explanation:

- **Label Smoothing**:
  - Technique to prevent model overconfidence
  - Instead of hard labels (0,1), uses soft labels
  - Example: [0, 1, 0] → [0.03, 0.94, 0.03]

Benefits:

- Improves model calibration
- Better generalization
- Reduces overfitting
- More robust predictions

2. Dataset Partitioning & Preparation

2.1 IID Partitioning Strategy

total_train_samples = 50000
samples_per_client = total_train_samples // NUM_CLIENTS
samples_per_class_per_client = samples_per_client // 10

Concept Explanation:

- **IID (Independent and Identically Distributed)**:
  - Each client gets data from similar distribution
  - Balanced representation of all classes
  - Ensures fair learning opportunity across clients

Implementation Details:

- Equal samples per client
- Balanced class distribution
- 80-20 train-validation split

- Controlled randomization

2.2 Data Normalization

```
norm = Normalize(
    mean=[0.485, 0.456, 0.406],
    std=[0.229, 0.224, 0.225]
)
```

Concept Explanation:

- **Image Normalization**:
  - Standardizes pixel values across dataset
  - Uses ImageNet statistics (transfer learning benefit)
  - Helps with model convergence

3. Training Optimizations

3.1 Batch Normalization Momentum

```
layer.momentum = 0.01  # BatchNorm layers
```

Concept Explanation:

- **Momentum in BatchNorm**:
  - Controls running statistics update rate
  - Lower momentum = slower updates
  - More stable in federated setting
  - Helps with client-server synchronization

3.2 Gradient Clipping

```
torch.nn.utils.clip_grad_norm_(net.parameters(), max_norm=1.0)
```

Concept Explanation:

- **Gradient Clipping**:
  - Prevents exploding gradients
  - Limits gradient magnitude to max_norm
  - Ensures training stability
  - Particularly important in deep networks

Implementation:

- max_norm=1.0:
  - Conservative clipping threshold
  - Prevents extreme parameter updates
  - Maintains stable training

3.3 Memory Optimizations

Pin Memory

```
DataLoader(
        dataset,
        pin_memory=True
)
```

Concept Explanation:

- **Pin Memory**:
  - Pins memory in RAM
  - Faster data transfer to GPU
  - Better GPU utilization
  - Important for GPU training

Benefits:

- Reduced data transfer time
- Better memory management
- Improved training speed
- Efficient resource utilization