Federated Learning with Attack Detection

1st Gaddam Abhiram

Panther id:002825416

2nd Sai Sarvagna Beeram

Panther id: 002817491

***Abstract*—Federated learning is a decentralized approach to machine learning that allows models to be trained directly on client devices, protecting privacy by keeping sensitive data local. This project focuses on building a federated learning system using the MNIST dataset, where 10 simulated clients independently train on local data partitions, and a central server aggregates their model updates to form a global model. To strengthen the system's resilience, we introduce attack detection mechanisms that identify and handle malicious clients submitting falsified updates. By applying a deviation threshold-based method, the framework effectively detects outlier contributions, ensuring the global model remains accurate and robust even under adversarial conditions. Experimental results show that the system maintains high classification performance despite attacks, demonstrating the success of the proposed defense strategies. These findings highlight the potential of federated learning to support privacy-sensitive applications in real-world scenarios, even in the presence of adversarial threats.**

I. Introduction

Federated learning is a distributed machine learning approach where multiple clients, such as devices or nodes, work together to train a shared model while keeping their data stored locally. Instead of sending raw data to a central server, each client trains on its own dataset and only shares model updates, like weights or gradients, helping preserve data privacy throughout the process (Wen et al., 2023). While federated learning offers strong privacy benefits, it also introduces new challenges. Malicious clients can intentionally disrupt the training by submitting poisoned or incorrect updates, risking the overall integrity and performance of the global model. These kinds of threats, often referred to as Byzantine faults, highlight the need for robust mechanisms that can detect and handle adversarial behavior during federated learning. In this project, we build a federated learning framework that not only enables secure, decentralized model training but also defends against adversarial attacks. By combining privacy-preserving aggregation with a deviation-based detection method, the system ensures high accuracy and resilience, even when faced with malicious client activities.

A. Objective

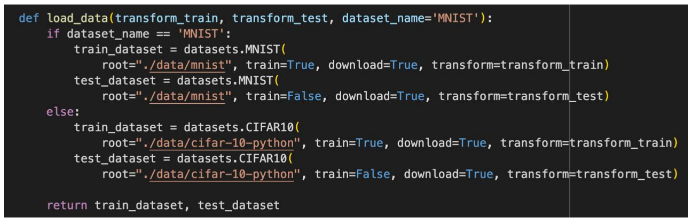
The present project aims at implementing a Federated Learning system, which integrates attack detection with robust aggregation techniques to secure the learning process against adversarial clients.

- Federated Learning Architecture: Federated learning designs a system where several clients in a distributed environment contribute to training the local models with private data and sharing only updates of the various models at the server-side main model.
- Malicious Client Detection: This is an abnormal deviation of model updates away from the global model as a way to detect malicious clients.
- Simulate Poisonous Attacks: It should support the simulation of an attack on malicious client behaviour by introducing updates that are intentionally set apart from the global model, similar to adversarial attacks.
- Employ Robust Aggregation: It employs robust aggregation techniques that reduce the adverse effects of malicious updates in order not to seriously affect global model performance.
- Attack Impact Mitigation: Employ the robust aggregation method, such as trimming extreme updates, to reduce the influence of malicious clients during model aggregation while preserving the integrity of the global model (Rane, et al., 2024).
- System Performance Evaluation: Evaluate the performance of the federated learning system to ensure that the global model maintains high accuracy even with malicious clients.

B. Background

Federated learning plays a critical role in applications where privacy is a major concern, such as mobile devices, healthcare systems, and financial services. In these fields, sharing sensitive data directly is often not an option due to legal, ethical, or security reasons. By allowing model training to happen locally on client devices, federated learning offers a way to protect user data while still enabling collaborative improvements to machine learning models. At the same time, the distributed nature of federated learning introduces new risks. Because the central server only receives model updates — not the actual data — it has limited visibility into whether clients are behaving honestly (Fenoglio et al., 2024). Some clients may act maliciously, intentionally sending manipulated updates to corrupt or mislead the training process. Such adversarial behavior can seriously harm the performance and reliability of the global model. To address these challenges, it becomes essential to develop mechanisms that can detect malicious clients and prevent them from influencing the learning process. Strong detection and mitigation strategies not only protect the integrity of federated learning systems but also ensure that privacy-preserving training remains effective and trustworthy in real-world scenarios.

C. *Data Handling*

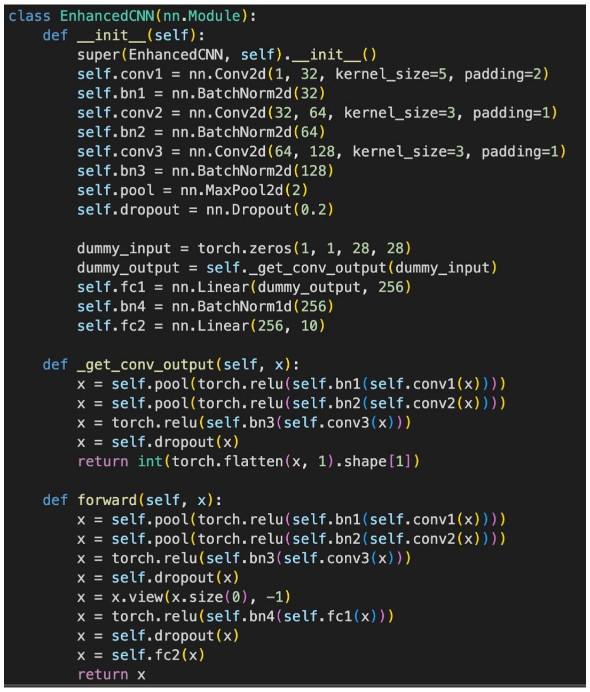


```python
def load_data(transform_train, transform_test, dataset_name='MNIST'):
    if dataset_name == 'MNIST':
        train_dataset = datasets.MNIST(
            root="./data/mnist", train=True, download=True, transform=transform_train)
        test_dataset = datasets.MNIST(
            root="./data/mnist", train=False, download=True, transform=transform_test)
    else:
        train_dataset = datasets.CIFAR10(
            root="./data/cifar-10-python", train=True, download=True, transform=transform_train)
        test_dataset = datasets.CIFAR10(
            root="./data/cifar-10-python", train=False, download=True, transform=transform_test)

    return train_dataset, test_dataset
```

Fig. 1. Data Loader

For this project, we use the MNIST dataset, which consists of images of handwritten digits. The dataset is divided into a training set and a test set, with the training data further partitioned across multiple clients. To simulate a realistic federated learning environment, the training data is first shuffled to ensure randomness and then split into ten equal parts, each assigned to a different client in the network. This setup allows each client to train its model independently on its own private subset of data, without any direct data sharing between clients.

D. *Model Architecture*



```python
class EnhancedCNN(nn.Module):
    def __init__(self):
        super(EnhancedCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=5, padding=2)
        self.bn1 = nn.BatchNorm2d(32)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(64)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm2d(128)
        self.pool = nn.MaxPool2d(2)
        self.dropout = nn.Dropout(0.2)

        dummy_input = torch.zeros(1, 1, 28, 28)
        dummy_output = self._get_conv_output(dummy_input)
        self.fc1 = nn.Linear(dummy_output, 256)
        self.bn4 = nn.BatchNorm1d(256)
        self.fc2 = nn.Linear(256, 10)

    def _get_conv_output(self, x):
        x = self.pool(torch.relu(self.bn1(self.conv1(x))))
        x = self.pool(torch.relu(self.bn2(self.conv2(x))))
        x = torch.relu(self.bn3(self.conv3(x)))
        x = self.dropout(x)
        return int(torch.flatten(x, 1).shape[1])

    def forward(self, x):
        x = self.pool(torch.relu(self.bn1(self.conv1(x))))
        x = self.pool(torch.relu(self.bn2(self.conv2(x))))
        x = torch.relu(self.bn3(self.conv3(x)))
        x = self.dropout(x)
        x = x.view(x.size(0), -1)
        x = torch.relu(self.bn4(self.fc1(x)))
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

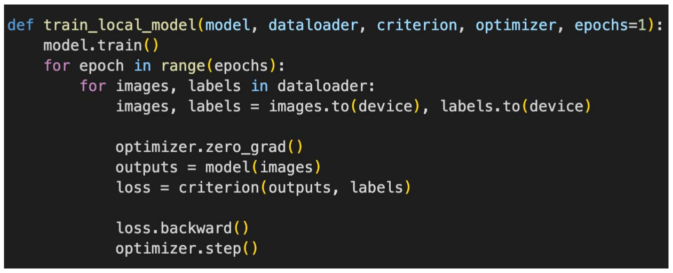
Fig. 2. Model Architecture

We employed CNN that includes three convolutional layers with increasing filters (32, 64, and 128) to capture progressively detailed features, each followed by batch normalization to stabilize and accelerate training. Max pooling layers are used to reduce feature map dimensions, improving efficiency, while a dropout layer prevents overfitting by randomly deactivating neurons. The architecture dynamically calculates the output size of the convolutional layers using a dummy input and connects to two fully connected layers, with batch normalization applied to the first. The final layer outputs predictions for 10 classes.

II. Proposed Methodology

A. *Federated Learning*

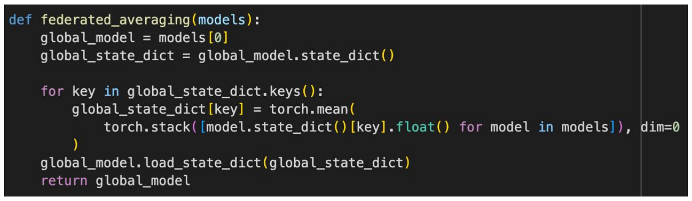
As discussed above, first the MNIST dataset is preprocessed and divided into 10 subsets, with each subset assigned to a unique client. A Data Loader is created for each client to handle batch processing efficiently.



```python
def train_local_model(model, dataloader, criterion, optimizer, epochs=1):
    model.train()
    for epoch in range(epochs):
        for images, labels in dataloader:
            images, labels = images.to(device), labels.to(device)

            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)

            loss.backward()
            optimizer.step()
```

Fig. 3. Local Training

1) Local Training: Each client independently trains a local instance of the EnhancedCNN model using its assigned dataset. The training process begins by initializing the local model with the current global model parameters to ensure consistency across clients. During training, the model processes batches of images through a forward pass, computing predictions using its layers. The CrossEntropyLoss function calculates the loss by comparing predictions with true labels, and the Adam optimizer adjusts the model parameters to minimize the loss. Training proceeds over the specified number of epochs, ensuring each client's model captures the characteristics of its local data before sending the updated parameters to the central server.



```python
def federated_averaging(models):
    global_model = models[0]
    global_state_dict = global_model.state_dict()

    for key in global_state_dict.keys():
        global_state_dict[key] = torch.mean(
            torch.stack([model.state_dict()[key].float() for model in models]), dim=0
        )
    global_model.load_state_dict(global_state_dict)
    return global_model
```

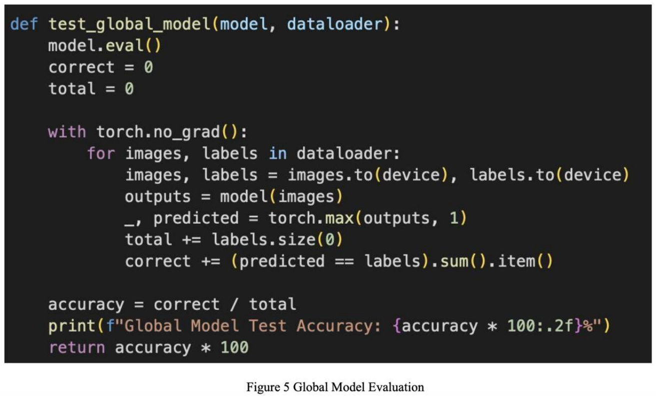
Fig. 4. Federated Averaging

2) Federated Averaging: After local training, each client submits its updated model parameters to the central server. The server performs Federated Averaging, a process that aggregates the parameters by computing the element-wise mean across all clients' updates. This ensures that the global model reflects the collective contributions of all clients while maintaining data privacy. The aggregated parameters are then applied to the global model, updating it with the knowledge gained during local training. This step forms the backbone of federated learning, combining distributed efforts into a unified model.

3) Evaluation: Following each round of aggregation, the updated global model is evaluated on a centralized test dataset to monitor its performance. The evaluation process involves feeding the test data through the global model to compute predictions. These predictions are compared with the true labels to calculate the accuracy, providing a measure of the global model's performance after each round. This step is crucial for tracking the model's convergence and ensuring its effectiveness across all rounds of federated learning.

4) Iterative Training and Aggregation: The entire process of local training, aggregation, and evaluation is repeated over multiple federated learning rounds. In each round, clients train their local models using the latest global model parameters, and the central server aggregates updates to refine the global model further. This iterative process allows the global model to improve progressively, incorporating diverse patterns from each client's data. Over successive rounds, the model achieves better performance, demonstrating the effectiveness of federated learning in leveraging decentralized data for robust model training.

B. Federated Learning with Attack Detection

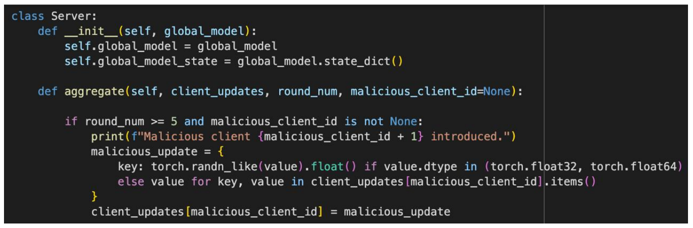


```
def test_global_model(model, dataloader):
    model.eval()
    correct = 0
    total = 0

    with torch.no_grad():
        for images, labels in dataloader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    accuracy = correct / total
    print(f"Global Model Test Accuracy: {accuracy * 100:.2f}%")
    return accuracy * 100
```

Figure 5 Global Model Evaluation

Fig. 5. Local Training

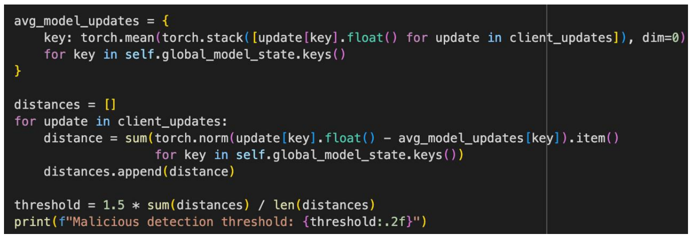
1) Local Training: Similar to Level 2, each client independently trains its local model using its assigned dataset. The local model is initialized with the global model's parameters and updated using the SGD optimizer and CrossEntropyLoss over multiple epochs before submitting updates to the server.



```
class Server:
    def __init__(self, global_model):
        self.global_model = global_model
        self.global_model_state = global_model.state_dict()

    def aggregate(self, client_updates, round_num, malicious_client_id=None):

        if round_num >= 5 and malicious_client_id is not None:
            print(f"Malicious client {malicious_client_id + 1} introduced.")
            malicious_update = {
                key: torch.randn_like(value).float() if value.dtype in (torch.float32, torch.float64)
                else value for key, value in client_updates[malicious_client_id].items()
            }
            client_updates[malicious_client_id] = malicious_update
```

Fig. 6. Malicious Client Introduction

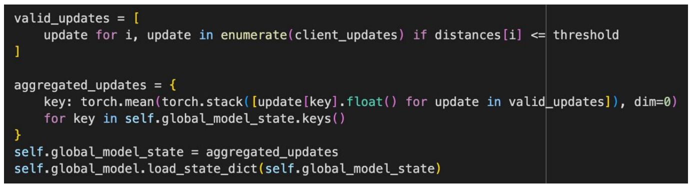
2) Malicious Client Introduction: To simulate an adversarial scenario, a malicious client is introduced after a specific training round (e.g., round 5). This malicious client submits falsified model updates instead of genuine ones. The updates are randomly generated and do not represent valid learning from the dataset. The aim is to disrupt the global model's performance by injecting harmful updates into the aggregation process, reflecting potential real-world attacks in federated learning systems.



```
avg_model_updates = {
    key: torch.mean(torch.stack([update[key].float() for update in client_updates]), dim=0)
    for key in self.global_model_state.keys()
}

distances = []
for update in client_updates:
    distance = sum(torch.norm(update[key].float() - avg_model_updates[key]).item()
                   for key in self.global_model_state.keys())
    distances.append(distance)

threshold = 1.5 * sum(distances) / len(distances)
print(f"Malicious detection threshold: {threshold:.2f}")
```

Fig. 7. Malicious Client Detection

3) Malicious Client Detection: The central server employs a mechanism to identify and isolate malicious updates by analyzing the deviation of each client's update from the average (mean) update. The process begins by computing the average of all received client updates for each model parameter. This average update represents the consensus across the client contributions. For each client, the server calculates the Euclidean distance between their update and the mean update. These distances are used to identify outliers—clients whose updates deviate significantly from the consensus. A deviation threshold is calculated as 1.5 times the average distance of all updates from the mean. This threshold acts as a boundary for identifying malicious updates. Any client whose distance exceeds this threshold is flagged as malicious.



```
valid_updates = [
    update for i, update in enumerate(client_updates) if distances[i] <= threshold
]

aggregated_updates = {
    key: torch.mean(torch.stack([update[key].float() for update in valid_updates]), dim=0)
    for key in self.global_model_state.keys()
}
self.global_model_state = aggregated_updates
self.global_model.load_state_dict(self.global_model_state)
```

Fig. 8. Malicious Client Removal

4) Malicious Client Removal: Once malicious clients are identified through the detection mechanism, their updates are excluded from the aggregation process. The server aggregates only the updates from non-malicious clients by computing the element-wise mean of their parameters. This ensures that the global model is not adversely affected by the malicious updates. The aggregation process focuses on valid contributions, enabling the global model to improve iteratively while maintaining robustness against adversarial behaviour.

C. Experiments and Results

1) Federated Learning: A federated learning framework was implemented where 10 clients independently trained local instances of a Convolutional Neural Network (CNN) on their partitioned MNIST datasets. Each client used a batch size of

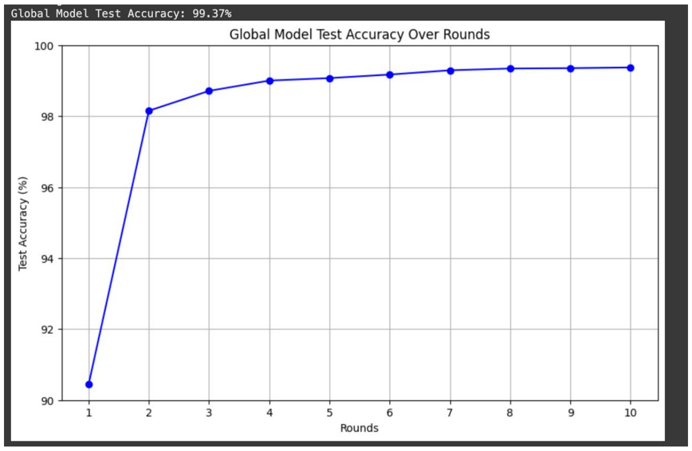


Fig. 9. Federated Learning

32, trained for 1 epoch per round, with the Adam optimizer and a learning rate of 0.001, while minimizing the CrossEntropy-Loss function. After local training, the server aggregated the client updates using Federated Averaging to update the global model. This process was repeated for 10 federated training rounds, with the global model evaluated on a centralized test dataset after each round to monitor performance improvements. The experiments showed a steady improvement in the global model's performance over successive rounds. Starting with a test accuracy of 90.44consistently improved, reaching 99.37gains, such as 98.15decentralized data. The improvement plateaued in later rounds, reflecting the model's convergence as it approached optimal performance. Hyperparameters such as the batch size, learning rate, and number of epochs per round were chosen to balance computational efficiency and learning progress. The results confirmed the effectiveness of federated learning, leveraging decentralized client data to train a highly accurate global model while preserving privacy.

2) Federated Learning with Attack Detection: The federated learning framework incorporated a mechanism to detect and mitigate the impact of malicious client updates. The experimental setup included 10 clients, each independently training a local CNN on their respective partitions of the MNIST dataset. The training process was conducted over 10 federated rounds, with each client using a batch size of 32, the SGD optimizer with a learning rate of 0.01, and the CrossEntropyLoss function. Starting from Round 6, a malicious client (Client 4) was introduced, submitting randomly generated, falsified updates to disrupt the learning process. The server detected malicious updates using a distance-based deviation threshold and excluded them from the aggregation process. During the initial rounds (Rounds 1–5), when no malicious updates were present, the global model demonstrated steady improvements in test accuracy, starting at 96.47and reaching 98.60introduced, the detection mechanism effectively flagged and excluded the malicious updates. Despite the adversarial behavior, the global model's performance continued to improve, achieving a test accuracy of 98.96The deviation thresholds observed during the initial rounds ranged between 4.45 and 6.49, reflecting consistent client contributions. However, after the malicious client's introduction, the threshold spiked significantly, exceeding 536, highlighting the successful detection of the outlier update. The results demonstrated the effectiveness of the attack detection mechanism in maintaining model robustness and accuracy, even in the presence of adversarial behavior.

D. Conclusion

This project highlights how federated learning can successfully balance privacy, collaboration, and model performance. By allowing 10 clients to train models on their own data and only share updates with a central server, the system preserved user privacy while steadily improving the global model's accuracy across training rounds. No raw data ever left the client devices, yet the collective learning process remained strong and effective. To make the system more robust, we introduced an adversarial scenario where a malicious client submitted falsified updates. Using a simple but powerful deviation-based detection method, the system was able to identify and exclude malicious updates before aggregation. As a result, the global model continued to improve without being compromised, even when under attack. Overall, this work shows that federated learning has the potential to build secure, privacy-first AI systems for real-world applications like healthcare, finance, and mobile services. Future work could explore more advanced aggregation strategies, smarter detection methods, and defenses against stronger or coordinated attacks to make the system even more resilient.

References

[1] Agrawal, S., Sarkar, S., Aouedi, O., Yenduri, G., Piamrat, K., Alazab, M., Bhattacharya, S., Maddikunta, P.K.R. and Gadekallu, T.R., (2022), 'Federated learning for intrusion detection system: Concepts, challenges and future directions', Computer Communications, 195, pp.346-361. https://arxiv.org/pdf/2106.09527

[2] Beilharz, J., Pfitzner, B., Schmid, R., Geppert, P., Arnrich, B. and Polze, A., (2021), 'Implicit model specialization through dag-based decentralized federated learning', In Proceedings of the 22nd International Middleware Conference (pp. 310-322). https://dl.acm.org/doi/pdf/10.1145/3464298.3493403

[3] Fenoglio, D., Dominici, G., Barbiero, P., Tonda, A., Gjoreski, M. and Langheinrich, M., (2024), 'Federated Behavioural Planes: Explaining the Evolution of Client Behaviour in Federated Learning', arXiv preprint arXiv:2405.15632. https://arxiv.org/pdf/2405.15632

[4] Ghavamipour, A.R., Zhao, B.Z.H., Ersoy, O. and Turkmen, F., (2024), 'Privacy-Preserving Aggregation for Decentralized Learning with Byzantine-Robustness', arXiv preprint arXiv:2404.17970. https://arxiv.org/pdf/2404.17970

[5] Kim, Y ., Yun, J., Shon, H. and Kim, J., (2021), 'Joint negative and positive learning for noisy labels', In Proceedings of the IEEE/CVF conference on computer vision and pattern recognition (pp. 9442-9451).

[6] Wen, J., Zhang, Z., Lan, Y ., Cui, Z., Cai, J. and Zhang, W., (2023), 'A survey on federated learning: challenges and applications', International Journal of Machine Learning and Cybernetics, 14(2), pp.513-535.