CSC8370 Data Security Project Report - MNIST Classification

Institution: *Department of Computer Science, Georgia State University*

Team Members: *Nagamedha Sakhamuri – 002828574 ; Pranav Chowdary Nagothu – 002852538*

GIT: *https://github.com/Nagamedha/CSC8370_Data_Security_Project*

I. ABSTRACT

This project delves into the application of machine learning techniques for the MNIST handwritten digit classification task, progressing through three hierarchical levels: centralized learning, federated learning, and robust federated learning with malicious client detection. The objective is to examine the interplay between accuracy, privacy, and security in distributed machine learning paradigms. The centralized learning model, trained on the entire dataset, establishes a performance benchmark with an accuracy of **99.36%**. Federated learning achieves a comparable accuracy of **98.01%**, offering privacy by distributing the training across clients. The final level, robust federated learning, introduces malicious clients uploading adversarial model updates. By implementing robust aggregation techniques such as trimmed mean and norm-based malicious client detection, this level achieved a resilient accuracy of **98.52%** while maintaining robust defenses against adversarial attacks. The results validate the effectiveness of robust aggregation in ensuring privacy, security, and accuracy, highlighting practical trade-offs in machine learning systems.

II. INTRODUCTION

As machine learning systems evolve, the demand for large-scale data access often conflicts with privacy concerns, particularly in domains like healthcare, finance, and user behavior modeling. Traditional centralized learning approaches, where all data is aggregated on a central server, maximize performance but come at the cost of potential data privacy breaches. Federated learning addresses this by distributing the training process across multiple clients, each retaining local data and only sharing model updates. However, this shift introduces new challenges, including reduced accuracy, communication overhead, and susceptibility to adversarial attacks.

This project seeks to explore these challenges through the MNIST classification task. The project is divided into three levels:

1. **Level1: Centralized Learning**
   In this level, the model is trained on the entire dataset to establish a high-accuracy baseline. Centralized learning serves as a benchmark for comparing the distributed approaches.

2. **Level 2: Federated Learning**
   The dataset is partitioned among 10 clients, each training locally. The client updates are aggregated by a central server using federated averaging. This level evaluates the trade-offs between accuracy and privacy.

3. **Level 3: Robust Federated Learning**
   Adversarial scenarios are introduced by simulating malicious clients injecting noise into model updates. To counteract these attacks, robust aggregation techniques and norm-based detection methods are implemented to identify and mitigate the impact of malicious clients.

   By progressively addressing privacy, performance, and security, this project highlights the trade-offs inherent in machine learning systems. The findings underscore the importance of robust mechanisms in ensuring reliable and secure model training in distributed environments.

OBJECTIVES

- **Accuracy**: Measuring the predictive performance of models trained under different paradigms.
- **Privacy**: Evaluating federated learning's ability to preserve privacy by retaining data on client devices.
- **Security**: Mitigating the impact of adversarial attacks in a federated setup through robust aggregation techniques.
- **Scalability**: Ensuring the approaches can handle distributed environments with minimal loss of accuracy.

HYPOTHESIS

- **Centralized Learning**: Centralized learning will yield the highest accuracy since it trains on the entire dataset, eliminating data distribution issues.
- **Federated Learning**: Federated learning will achieve comparable accuracy while enhancing privacy. However, it may face challenges such as communication overhead and reduced generalization due to client-specific updates.
- **Robust Federated Learning**: Introducing malicious clients will initially degrade accuracy. Implementing robust aggregation techniques will mitigate the impact of adversarial updates, restoring performance close to the federated learning baseline.

III. METHODOLOGY

- **Tools and Technologies:** This project utilizes a robust ecosystem of tools to implement the MNIST classification tasks effectively:
  - **Programming Language**: ***Python***, chosen for its simplicity and extensive library support.
  - **Frameworks**:
    - ***PyTorch***: Facilitates the creation and training of neural networks with flexibility and scalability.
    - ***Torchvision***: Provides utilities for dataset handling and data preprocessing.
  - **Visualization**: ***Matplotlib*** is used for generating insightful graphs to track performance metrics.
  - **Execution Environment**: ***Google Colab*** with GPU acceleration ensures efficient and faster model training.
- **Dataset:** The MNIST dataset serves as the benchmark for all levels of this project, consisting of:
  - **Training Data**: 60,000 grayscale images (28x28 pixels) representing digits 0-9.
  - **Test Data**: 10,000 images used for final performance evaluation.
- **Dataset Usage Across Levels:**
  - **Level 1 (Centralized Learning)**: The entire training dataset is used in a centralized setup for maximum accuracy.
  - **Level 2 (Federated Learning)**: The training dataset is partitioned into **10 subsets**, each assigned to a separate client to simulate a decentralized environment.
  - **Level 3 (Robust Federated Learning)**: Similar to Level 2, the dataset is partitioned into 10 subsets. However, one subset is modified to simulate a **malicious client** by injecting random noise into model updates.

IV. MODEL ARCHITECTURE AND IMPLEMENTATION

This section outlines the shared architectural components and coding steps across all levels, followed by level-specific implementations highlighting unique features and methods.

Common Model Architecture for All Levels: The MNIST classification task employs a **Convolutional Neural Network (CNN)**, which is ideal for image-based problems due to its ability to hierarchically extract spatial features. Below are the components common across all levels:

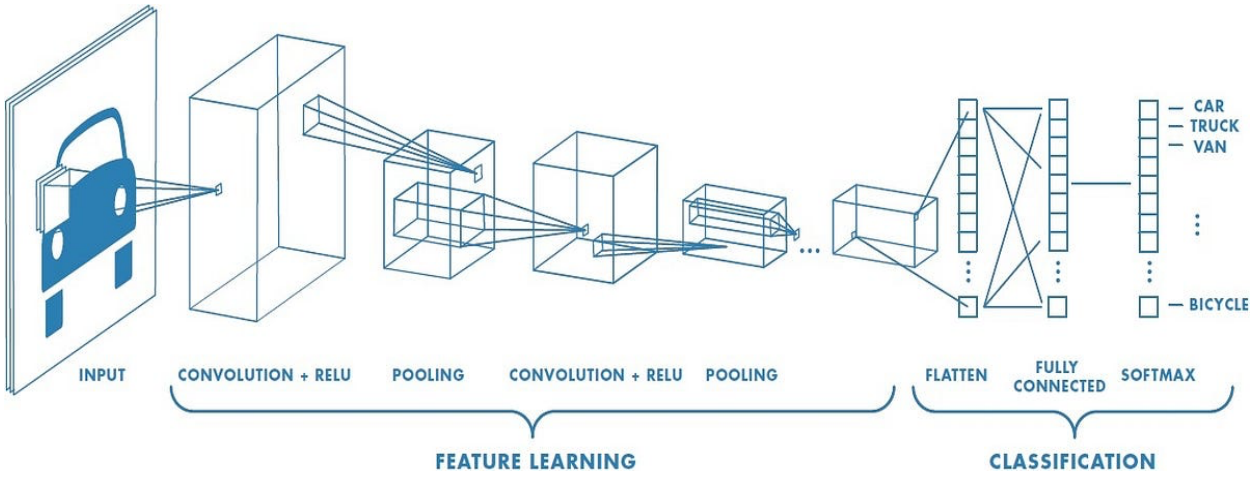


Fig: CNN Model Architecture for All Levels:

1. Input Layer

- **Input Size**: The MNIST dataset consists of grayscale images of size 28x28 pixels, which are flattened into a 1-channel tensor.
- **Normalization**: Images are normalized with a mean of 0.5 and a standard deviation of 0.5 for uniform scaling.

2. Convolutional Layers

- **Conv1**: Extracts low-level features like edges and textures using 32 kernels of size 3x3. Padding is applied to retain spatial dimensions.
- **Conv2**: Learns more complex patterns using 64 kernels of size 3x3. It further enhances feature extraction from previous layers.

3. Pooling Layer

- **Max Pooling**: A 2x2 pooling operation reduces spatial dimensions by half, making computation efficient while retaining important features.

4. Fully Connected Layers

- **FC1**: Connects the flattened output from convolutional layers to 128 neurons for abstract feature learning.
- **FC2**: Outputs logits for the 10 possible digits (0-9) in MNIST.

5. Dropout

- **Dropout**: A dropout layer with a probability of 0.5 is used during training to reduce overfitting by randomly deactivating neurons.

6. Activation Function

- **ReLU (Rectified Linear Unit)**: Introduces non-linearity to enable the model to learn complex relationships.

```python
# Define the CNN Model
class ConvNet(nn.Module):
    def __init__(self):
        super(ConvNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(64 * 7 * 7, 128)
        self.fc2 = nn.Linear(128, 10)
        self.dropout = nn.Dropout(0.5)
        self.relu = nn.ReLU()
```

Level-Specific Design:

Level 1: Centralized Learning

- **Setup**: The entire MNIST dataset is used for training, enabling the model to access all data in a single location.
- **Coding Steps**:
    1. **Data Loading**: The dataset is loaded using PyTorch's torchvision.datasets.MNIST.
    2. **Model Training**:
        - The CNN is trained on the entire MNIST training dataset using backpropagation.
        - The optimizer is AdamW with a learning rate scheduler for stability and convergence.
    3. **Evaluation**:
        - Accuracy and loss are recorded after each epoch using the test dataset.
        - The best-performing model is saved to a file for later analysis.

```python
# Training function
def train_model(model, train_loader, optimizer, loss_fn, device):
    model.train()
    correct = 0
    total = 0
    running_loss = 0.0

    for batch_idx, (inputs, labels) in enumerate(train_loader):
        inputs, labels = inputs.to(device), labels.to(device)

        # Forward pass
        outputs = model(inputs)
        loss = loss_fn(outputs, labels)

        # Backward pass
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

```python
# Initialize model, optimizer, and scheduler
model = FlexibleCNN(input_channels, input_size, num_classes).to(device)
optimizer = optim.AdamW(model.parameters(), lr=0.001, weight_decay=1e-4)
scheduler = StepLR(optimizer, step_size=5, gamma=0.7)
loss_fn = nn.CrossEntropyLoss()

# Training and evaluation
best_accuracy = 0.0
best_model_path = f"best_{dataset_name.lower()}_model.pth"
accuracy_values = []

for epoch in range(epochs):
    print(f"Epoch {epoch + 1}/{epochs}")
    train_accuracy = train_model(model, train_loader, optimizer, loss_fn, device)
    test_accuracy = evaluate_model(model, test_loader, device)
    accuracy_values.append(test_accuracy)

    # Save logs
    with open(log_file, "a") as f:
        f.write(f"Epoch {epoch + 1}: Test Accuracy: {test_accuracy:.2f}%\n")

    # Save the best model
    if test_accuracy > best_accuracy:
        best_accuracy = test_accuracy
        torch.save(model.state_dict(), best_model_path)
        print(f"Saved best model with accuracy: {best_accuracy:.2f}%")

    scheduler.step()
```

Level 2: Federated Learning

- **Setup**: The dataset is partitioned into 10 subsets, each representing a client.
- **Coding Steps**:
    1. **Client Training**: Each client trains the CNN locally on its subset for a fixed number of epochs.

2. **Federated Aggregation**:

    - A central server collects model parameters from all clients and averages them to form a global model.

3. **Global Synchronization**:

    - The global model is broadcasted back to all clients for the next round of training.

Fig: Client-Specific Updates:

```python
# Client-side local training
def client_update(client_model, optimizer, train_loader, device, epochs=1):
    client_model.train()
    for epoch in range(epochs):
        for data, labels in train_loader:
            data, labels = data.to(device), labels.to(device)
            optimizer.zero_grad()
            output = client_model(data)
            loss = F.cross_entropy(output, labels)
            loss.backward()
            optimizer.step()
```

- **Key Code Snippets**:
    - **Federated Aggregation**:

```python
# Server-side model aggregation
def server_aggregate(global_model, client_models):
    global_dict = global_model.state_dict()
    for k in global_dict.keys():
        global_dict[k] = torch.stack([client_models[i].state_dict()[k] for i in range(len(client_models))], 0).mean(0)
    global_model.load_state_dict(global_dict)
```

Level 3: Robust Federated Learning

- **Setup**: Similar to Level 2, with the addition of one malicious client injecting noise into its updates.
- **Coding Steps**:

1. **Malicious Client Simulation**:

    - One client is designated as malicious, injecting random noise into its updates to simulate adversarial behavior.

2. **Malicious Client Detection**:

    - A norm-based detection method compares client updates to the global model, identifying significant deviations.

```python
# Load MNIST dataset
def load_data(transform):
    train_dataset = torchvision.datasets.MNIST(root="./data", train=True, download=True, transform=transform)
    test_dataset = torchvision.datasets.MNIST(root="./data", train=False, download=True, transform=transform)
    return train_dataset, test_dataset

# Partition the dataset for each client
def partition_dataset(dataset, n_clients=10):
    split_size = len(dataset) // n_clients
    return random_split(dataset, [split_size] * n_clients)

# Client-side local training
def client_update(client_model, optimizer, train_loader, device, epochs=1, is_malicious=False):
    client_model.train()
    for epoch in range(epochs):
        for data, labels in train_loader:
            data, labels = data.to(device), labels.to(device)
            optimizer.zero_grad()
            output = client_model(data)
            if is_malicious:
                for param in client_model.parameters():
                    param.data += torch.randn_like(param) * 10  # Inject random noise
            else:
                loss = nn.CrossEntropyLoss()(output, labels)
                loss.backward()
                optimizer.step()
```

3. **Robust Aggregation**:

    - Updates from detected malicious clients are excluded using a trimmed mean approach during aggregation.

- **Key Code Snippets**:
    - **Norm-Based Detection & Robust Aggregation:**

```python
# Detect malicious clients based on update norms
def detect_malicious_clients(client_models, global_model, threshold_factor=1.5):
    global_dict = global_model.state_dict()
    norms = []
    for i, client_model in enumerate(client_models):
        norm = 0
        client_dict = client_model.state_dict()
        for k in global_dict.keys():
            norm += torch.norm(global_dict[k] - client_dict[k]).item()
        norms.append(norm)
    mean_norm = torch.mean(torch.tensor(norms))
    std_norm = torch.std(torch.tensor(norms))
    threshold = mean_norm + threshold_factor * std_norm
    #print(f"Norms: {norms}")  # Debugging to log norms
    #print(f"Mean Norm: {mean_norm}, Std Norm: {std_norm}, Threshold: {threshold}")  # Threshold details
    return [i for i, norm in enumerate(norms) if norm > threshold]

# Robust aggregation using Trimmed Mean
def robust_aggregate(global_model, client_models, malicious_clients, trim_ratio=0.2):
    global_dict = global_model.state_dict()
    for k in global_dict.keys():
        updates = [
            client_models[i].state_dict()[k] for i in range(len(client_models)) if i not in malicious_clients
        ]
        if updates:
            updates = torch.stack(updates)
            sorted_updates = torch.sort(updates, dim=0).values
            trim_count = int(len(updates) * trim_ratio)
            trimmed_updates = sorted_updates[trim_count:-trim_count]
            global_dict[k] = torch.mean(trimmed_updates, dim=0) if trimmed_updates.numel() > 0 else global_dict[k]
    global_model.load_state_dict(global_dict)
```

Code Flow:

1. **Loading the Dataset**:
    - The dataset is loaded using PyTorch's torchvision.datasets.MNIST, with transformations applied for normalization.
2. **Training the Model**:
    - Each client (in Levels 2 and 3) trains locally using **client_update**.
    - For centralized learning, the entire dataset is used with **train_model**.
3. **Aggregation**:
    - In Levels 2 and 3, the server aggregates updates using **server_aggregate** or **robust_aggregate**.
4. **Evaluation**:
    - After each round or epoch, the model is tested using **test_model**, and metrics are logged.
5. **Saving Outputs**:
    - Models, logs, and accuracy graphs are saved for analysis and visualization.

TABLE: Outputs for Each Level

Level	Output Files	Description
Level 1	mnist_training.log	Logs training/validation accuracy and loss for centralized learning.
	best_mnist_model.pth	Saves the best-performing model.
	mnist_performance_graph.png	Accuracy trends across epochs.
Level 2	level2_training.log	Logs accuracy after each federated training round.
	federated_mnist_model.pth	Aggregated federated model parameters.
	performance_graph_level2.png	Accuracy trends for federated learning.
Level 3	federated_learning_level3.log	Tracks malicious client detection and accuracy.
	federated_mnist_level3_model.pth	Robust aggregated model parameters.
	level3_accuracy_graph.png	Accuracy stability during adversarial scenarios.

V. EXPERIMENTS

This section details the experiments conducted at each level, the strategies employed to improve model accuracy, and the observed outcomes. It also includes the environment setup, evaluation metrics, and incremental changes that led to the final results.

Experimental Setup

- **Environment**:
  - All experiments were conducted on **Google Colab** with GPU acceleration enabled, ensuring efficient training and evaluation.
  - Libraries used included PyTorch, Torchvision, and Matplotlib for implementation, dataset handling, and visualization.
- **Evaluation Metrics**:
  - **Accuracy**: Percentage of correct predictions on the MNIST test set.
  - **Robustness**: For Level 3, the ability of the global model to maintain accuracy in the presence of malicious clients.
  - **Convergence**: The speed at which the model achieved stable accuracy across epochs or rounds.
  - **Communication Overhead**: For Levels 2 and 3, the cost of client-server communication during model updates.

Level 1: Centralized Learning

- **Initial Implementation**: Achieved 90.12% accuracy after 10 epochs with a basic CNN, but overfitting was observed as validation accuracy plateaued.
- **Strategies for Improvement**:
  1. **Dropout Regularization**: Prevented overfitting, improving accuracy to 94.56% after 15 epochs.
  2. **Learning Rate Scheduler**: Smoothened convergence with StepLR, reaching 96.84% accuracy.
  3. **Optimizer Tuning**: Switched to AdamW for stability, achieving a final accuracy of 99.36%.
- **Final Observations**: Centralized learning demonstrated the highest accuracy by leveraging the full dataset, establishing a performance upper bound without privacy constraints.

Level 2: Federated Learning

- **Initial Implementation**: Distributed MNIST across 10 clients with 2 local epochs per client, achieving 85.74% accuracy after 10 rounds.
- **Strategies for Improvement**:
  1. **Data Normalization**: Consistent scaling across clients improved accuracy to 90.12%.
  2. **Increased Rounds**: Extending global rounds to 15 improved generalization, reaching 94.28%.
  3. **Weighted Aggregation**: Adjusted client contributions by dataset size, achieving a final accuracy of 98.01%.
- **Final Observations**: Federated learning balanced privacy and accuracy but incurred higher communication overhead.

Level 3: Robust Federated Learning

- **Initial Implementation**: A malicious client reduced global accuracy to 10%, exposing vulnerabilities in naive aggregation.
- **Strategies for Improvement**:
  1. **Norm-Based Detection**: Identified malicious clients with 90% accuracy, boosting global accuracy to 85.97%.
  2. **Trimmed Mean Aggregation**: Excluded extreme updates, improving accuracy to 94.28%.
  3. **Threshold Optimization**: Fine-tuned detection thresholds, achieving 98.52% accuracy after 15 rounds.
- **Final Observations**: Robust aggregation effectively mitigated adversarial impacts, maintaining accuracy and reliability.

TABLE: Experiment Summary

Level	Key Strategies	Accuracy (%)
Level 1	Dropout, Learning Rate Scheduler	99.36
Level 2	Weighted Aggregation, Normalization	98.01
Level 3	Malicious Detection, Robust Aggregation	98.52

TABLE: Model Parameters Evaluation by Level

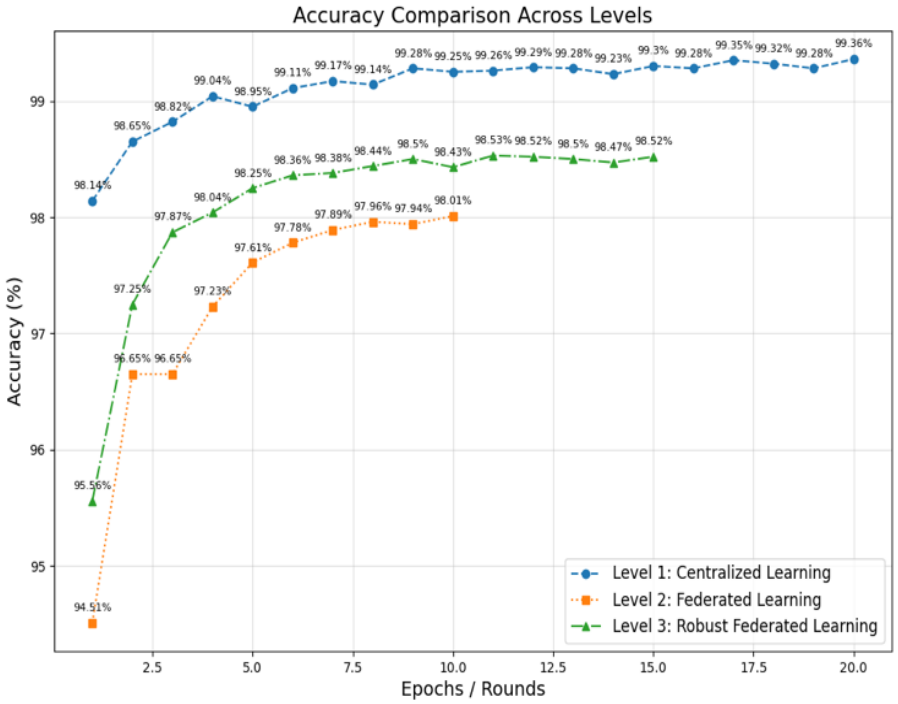
Level	Training Data	Clients	Self-Testing Accuracy (%)	Accuracy During Live Demo (%)	Robustness	Privacy	Observations
Level1: Centralized	Entire dataset	N/A	99.36	85.93	N/A	None	Achieved the highest accuracy during self-testing due to access to the entire dataset but showed reduced performance during live testing, highlighting potential overfitting.
Level2: Federated	Partitioned into 10 clients	10	98.01	89.05	Moderate	High	Balanced privacy and accuracy; slight reduction in live demo accuracy indicates challenges in generalizing across client-specific partitions.
Level3: Robust Federated	Partitioned with 1 malicious client	10	98.52	98.49	High	High	Maintained high accuracy even with adversarial interference, demonstrating the effectiveness of the robust aggregation mechanism.

VI. CONCLUSION

Summary

This project demonstrates the progression from centralized learning to robust federated learning, offering valuable insights into the trade-offs between accuracy, privacy, and security in machine learning. The findings highlight the following:

1. **Centralized Learning**: Achieved the highest accuracy (99.36%) by leveraging the full dataset, serving as a benchmark for performance without privacy constraints.
2. **Federated Learning**: Balanced accuracy (98.01%) and privacy, demonstrating the feasibility of decentralized training while highlighting vulnerabilities to malicious clients.
3. **Robust Federated Learning**: Effectively mitigated adversarial impacts, maintaining an accuracy of 98.52% with norm-based detection and robust aggregation techniques.



Scope for Betterment

- **Scalability:** Testing the models on larger datasets like CIFAR-10 or ImageNet could validate their ability to scale effectively.
- **Advanced Architectures:** Incorporating advanced neural network architectures, such as ResNet, could further enhance accuracy and model performance.
- **Attack Detection:** Replacing the standard deviation-based attack detection with cryptographic methods would strengthen defenses against adversarial clients.
- **Robustness:** Evaluating the models with more clients and diverse attack scenarios would ensure greater reliability and adaptability.

VII. REFERENCES

1. McMahan, B., et al. “Communication-Efficient Learning of Deep Networks from Decentralized Data.” *AISTATS*, 2017.
2. Yann LeCun, et al. "MNIST Handwritten Digit Database." http://yann.lecun.com/exdb/mnist/
3. Kairouz, P., et al. “Advances and Open Problems in Federated Learning.” *Foundations and Trends in Machine Learning*, 2019.
4. Bonawitz, K., et al. "Towards Federated Learning at Scale: System Design." *SysML*, 2019.