DAI Assignment-3 Report

Task: Implement any federated learning algorithm using Pytorch with the MNIST dataset.

Objectives:

- Perform (0-9) digit classification task using federated setup by performing aggregation at the central server.
- Report the class-wise accuracy results for all three datasets at their respective client side and at the central server also. Report overall classification Accuracy and Confusion Matrix.
- Write the mathematical explanation of the function used to perform the aggregation at the central server.
- Write the detailed explanation of the federated learning algorithm with the diagrammatic representation used for the above solution.
- Compare the results of overall accuracy in federated setup with the baseline results calculated by combining all the datasets and training in non-federated setup. Do you observe any decrease/increase in accuracy for both setups? State your answer with proper reasoning.

Procedure:

- Explanations of code file by file
    - Server.py
        - Importing required libraries: The necessary libraries required for building and training the model are imported, such as torch, numpy, matplotlib, torchmetrics, mlxtend, etc.
        - Defining the device: The device used for training the model is set to 'cuda' if it's available; otherwise, it's set to 'cpu'.
        - Data preprocessing: The MNIST dataset is loaded and preprocessed. The images are resized to 32x32 pixels, converted to tensors, and the dimensions are changed to (3,32,32) for compatibility with the pre-trained models.
        - Trimming the dataset: To reduce the training time and ensure class balance, the training and testing datasets are trimmed to contain 1000 and 500 images per class, respectively.
        - Creating dataloaders: DataLoader objects are created for the trimmed datasets for training, testing, and evaluation purposes.
        - Defining the model: The FedModel class is defined, which is a feedforward neural network consisting of three fully connected layers.
        - Training the central model: The central model is trained using the trimmed training dataset. The training and testing losses and accuracies are recorded for each epoch. The Adam optimizer is used for optimization, and the learning rate is set to 0.001.
        - Evaluating the model: The central model is evaluated using the trimmed testing dataset. The evaluation loss and accuracy are recorded for each epoch.

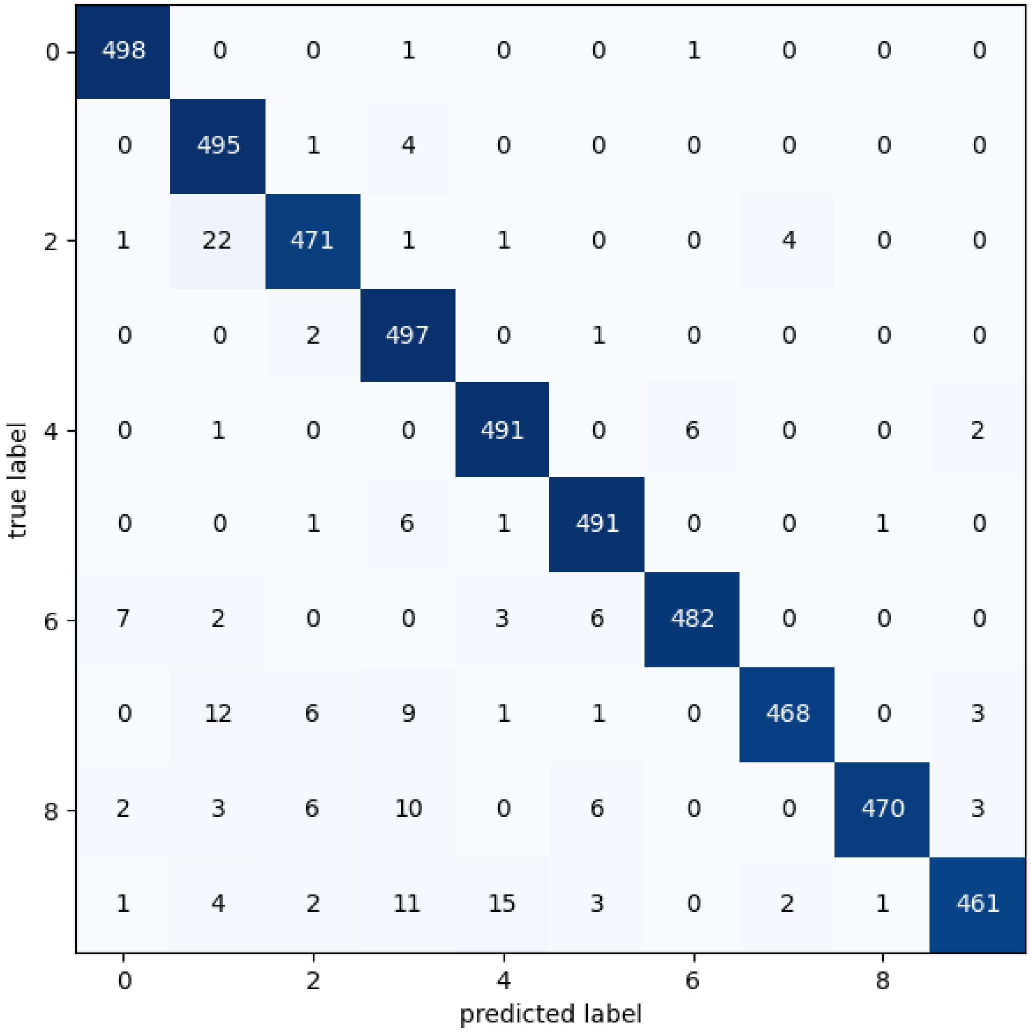
        - Plotting the results: The training and testing losses and accuracies for each epoch are plotted.
        - Generating the confusion matrix: The confusion matrix is generated using the torchmetrics.ConfusionMatrix library, and the predicted labels for the test dataset are recorded.
    - Fedlearn.py
        - This is a Python script for Federated Learning, where several clients train a model locally and then send their updated model weights to a central server, which averages them and then sends the new averaged weights back to each client.
        - Import statements for necessary packages and modules, including PyTorch, NumPy, Matplotlib, and other custom modules.
        - The creation of a device variable to run the code on either CPU or GPU depending on the availability of a GPU.
        - The loading of the MNIST dataset using torchvision, and the creation of a DataLoader object to load the data in batches for testing.
        - A function, get_avg_weight, which takes a list of client models as input and returns the average weight for each layer of the model.
        - A function, update_fedmodel_weights, which takes the average weights returned by get_avg_weight and updates the weights of the FedModel accordingly.
        - A main block of code that creates an instance of the FedModel, runs a loop for several rounds of training, and for each round, iterates over a list of client models, updates the FedModel with the averaged weights from the clients, and tests the updated model on the test dataset. The code also prints out the accuracy and loss for each round of training.
    - Model.py
        - This code defines a neural network model for image classification tasks.
        - It imports necessary modules from PyTorch:
            - torch for tensor operations
            - torch.nn for neural network layers and model construction
            - torchmetrics.classification.Accuracy for computing accuracy metric
        - It checks whether a CUDA-capable GPU is available and sets the device accordingly.
        - It defines a convolutional neural network model FedModel which has:
            - 3 convolutional layers with ReLU activation function, followed by max pooling layer
            - 2 fully connected layers with ReLU activation function
        - It defines a loss function CrossEntropyLoss, which is commonly used for classification problems
        - It defines an accuracy function Accuracy for multiclass classification with 10 classes.
        - It creates an instance of the FedModel class and moves it to the specified device.

    - It generates a random tensor of shape (1, 3, 32, 32) and moves it to the specified device.
    - It passes the random tensor through the model and prints the shape of the output tensor.
  - Engine.py
    - This is a Python module named engine.py that contains three functions: training_step(), testing_step(), and eval_func(). Here's what each function does:
    - training_step()
    - This function performs a training step for one epoch. It takes in the following parameters:
      - model: a PyTorch model class object
      - dataloader: a training dataloader from the training dataset
      - loss_fn: a loss function (object) of your choice
      - acc_fn: an accuracy function from torchmetrics
      - optimizer: an optimizer function (object) of your choice
      - device: a string indicating the torch device to use ("CPU" or "GPU")
      - profiler: an optional PyTorch profiler object (default is None)
    - The function first puts the model in training mode using model.train(). It then initializes the training loss and accuracy for the epoch to zero.
    - If a profiler object is given, it starts profiling using profiler.start() and loops over the batches in the dataloader. It sends the data and targets to the target device using X.to(device) and y.to(device).
    - For each batch, it performs the following steps:
    - Performs a forward pass of the model on the input data to get the predicted logits (y_pred_logits).
    - Calculates the loss between the predicted logits and the ground truth labels using the given loss function (loss_fn).
    - Zeros out the gradients using optimizer.zero_grad().
    - Backpropagates the loss using loss.backward().
    - Updates the model's parameters using optimizer.step().
    - Calculates the accuracy using the given accuracy function (acc_fn) and adds it to the epoch's accuracy.
    - If a profiler object is given, it records the profiling information using profiler.step().
    - If no profiler object is given, the same process is followed as above, except step 7 is not performed.
    - Finally, the function returns the training loss and accuracy for the epoch divided by the number of batches in the dataloader, as well as the updated model.
    - testing_step()
    - This function performs a testing step for one epoch. It takes in the following parameters:

    - model: a PyTorch model class object
    - dataloader: a testing dataloader from the testing dataset
    - loss_fn: a loss function (object) of your choice
    - acc_fn: an accuracy function from torchmetrics
    - device: a string indicating the torch device to use ("CPU" or "GPU")
- The function first puts the model in evaluation mode using model.eval(). It then initializes the testing loss and accuracy for the epoch to zero.
- It loops over the batches in the dataloader and sends the data and targets to the target device using X.to(device) and y.to(device).
- For each batch, it performs the following steps:
- Performs a forward pass of the model on the input data to get the predicted logits (y_pred_logits).
- Calculates the loss between the predicted logits and the ground truth labels using the given loss function (loss_fn).
- Calculates the accuracy using the given accuracy function (acc_fn) and adds it to the epoch's accuracy.
- Finally, the function returns the testing loss and accuracy for the epoch divided by the number of batches in the dataloader.
- eval_func is used to evaluate the performance of a model on a given dataset. It takes in the following arguments:
    - model: a PyTorch neural network model to evaluate
    - dataloader: a PyTorch dataloader object containing the data to evaluate the model on
    - loss_fn: a PyTorch loss function to use to calculate the loss of the model's predictions
    - accuracy_fn: a function to use to calculate the accuracy of the model's predictions
    - device: a string specifying the device to use for running the model (e.g. 'cpu' or 'cuda')
- The function first sets the model to evaluation mode by calling model.eval(). It then iterates over the batches in the dataloader, performs a forward pass through the model to obtain the model's predictions, calculates the loss and accuracy of the model's predictions, and aggregates these values across all batches. Finally, it calculates the average loss and accuracy over all batches and returns these values.
- train_client is used to train a PyTorch neural network model on a given dataset. It takes in the following arguments:
    - client_model: a PyTorch neural network model to train
    - train_dataloader: a PyTorch dataloader object containing the training data
    - test_dataloader: a PyTorch dataloader object containing the testing data

            - client_optimizer: a PyTorch optimizer object to use for optimizing the model's parameters during training
            - client_loss_fn: a PyTorch loss function to use to calculate the loss of the model's predictions during training
            - accuracy_fn: a function to use to calculate the accuracy of the model's predictions during training
            - device: a string specifying the device to use for running the model during training (e.g. 'cpu' or 'cuda')
        - The function first sets the number of epochs to train for, initializes lists to store the training and testing loss and accuracy values for each epoch, and sets the random seed for reproducibility. It then iterates over the epochs, performing a single training step and a single testing step for each epoch. The training step involves iterating over the batches in the training dataloader, performing a forward pass through the model to obtain the model's predictions, calculating the loss and accuracy of the model's predictions, and using the optimizer to backpropagate the loss and update the model's parameters. The testing step involves using the eval_func function defined above to evaluate the model's performance on the testing data. The function then appends the training and testing loss and accuracy values for the epoch to the appropriate lists. Finally, the function returns the trained model, optimizer, loss function, and testing accuracy.
    - ClientX.py
        - Import necessary packages/modules from PyTorch and other libraries.
        - Load and preprocess the MNIST dataset with the torchvision package.
        - Trim the training and testing datasets to reduce their sizes to 10,000 and 5,000 samples, respectively, while keeping class distribution balanced.
        - Define a PyTorch model (FedModel), a loss function (CrossEntropyLoss), an optimizer (Adam), and an accuracy function.
        - Define a function (update_client) to update the weights of the local model of a client using its training dataset.
        - Train the local model of the first client (client1) using the function defined in step 5.
        - Save the weights of the trained client model.
        - Make a prediction on the test dataset of the client1 and print its accuracy, along with a confusion matrix and classwise accuracy.
        - Visualize the confusion matrix.
- Results:
- Client 1:
- Overall Accuracy of Client Model: 0.9085
- Class wise Accuracy:
    - Class 0: 0.9980
    - Class 1: 0.9880
    - Class 2: 0.9840

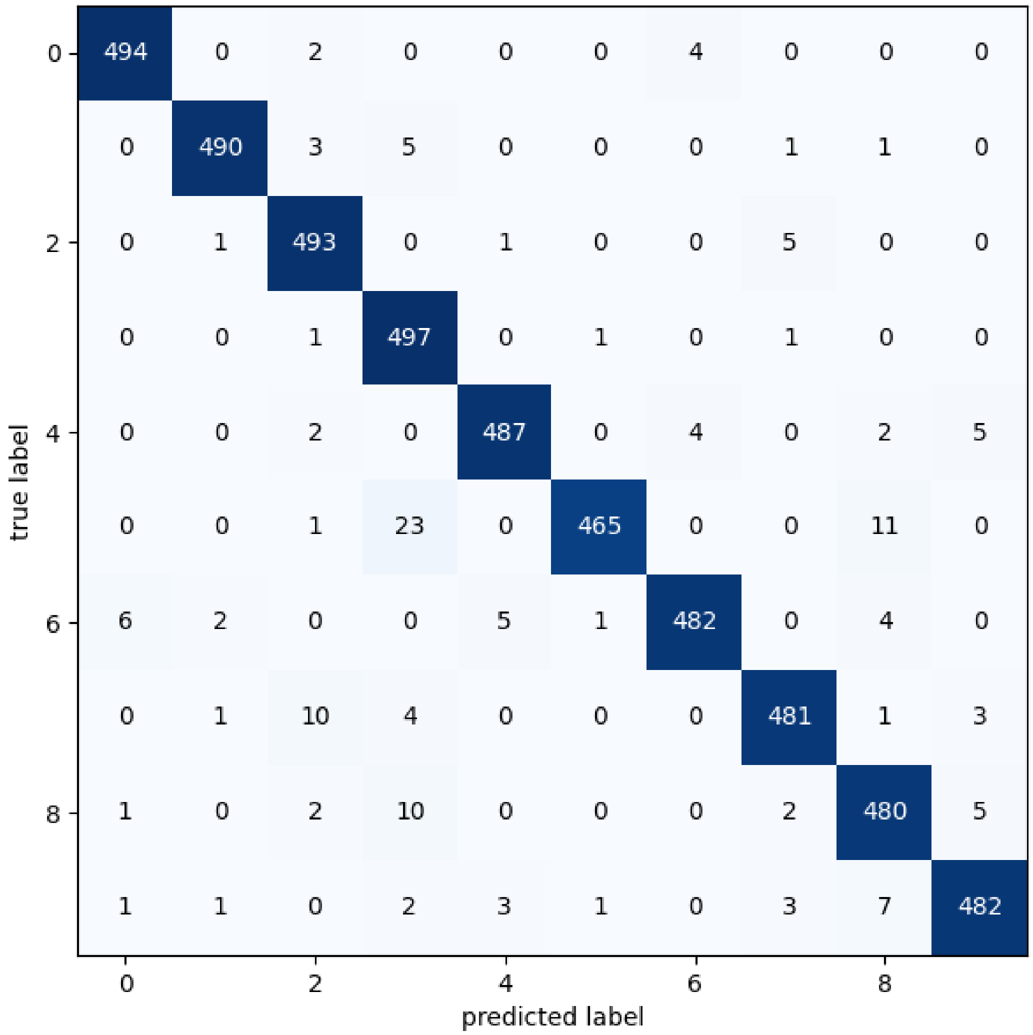
    - Class 3: 0.9900
    - Class 4: 0.9760
    - Class 5: 0.9740
    - Class 6: 0.9680
    - Class 7: 0.9700
    - Class 8: 0.9740
    - Class 9: 0.9500
- Confusion Metric



498	0	0	1	0	0	1	0	0	0
0	495	1	4	0	0	0	0	0	0
1	22	471	1	1	0	0	4	0	0
0	0	2	497	0	1	0	0	0	0
0	1	0	0	491	0	6	0	0	2
0	0	1	6	1	491	0	0	1	0
7	2	0	0	3	6	482	0	0	0
0	12	6	9	1	1	0	468	0	3
2	3	6	10	0	6	0	0	470	3
1	4	2	11	15	3	0	2	1	461

- Client 2:
- Overall Accuracy of Client Model: 0.8994
- Class wise Accuracy:
    - Class 0: 0.9980
    - Class 1: 0.9860
    - Class 2: 0.9820
    - Class 3: 0.9940
    - Class 4: 0.9700
    - Class 5: 0.9460

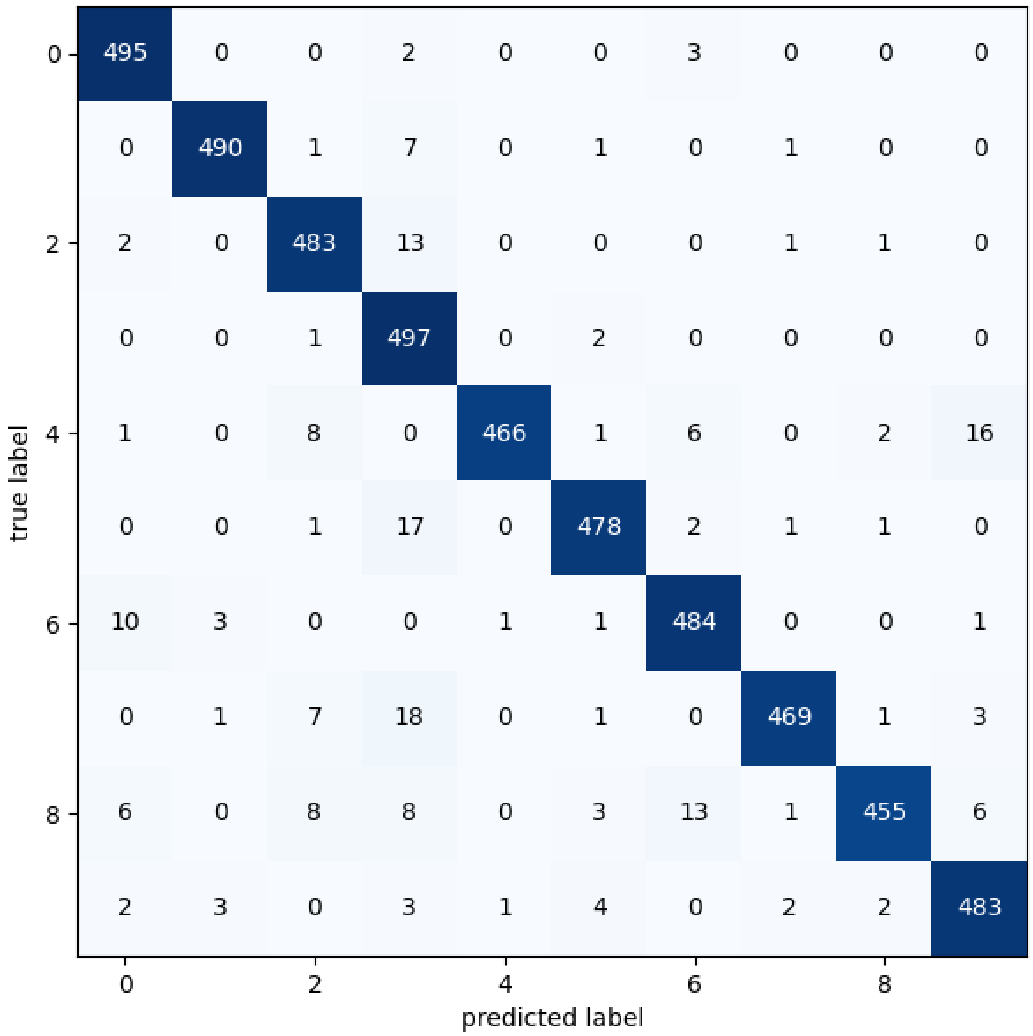
    - Class 6: 0.9640
    - Class 7: 0.9500
    - Class 8: 0.9520
    - Class 9: 0.9580
- Confusion Metric



494	0	2	0	0	0	4	0	0	0
0	490	3	5	0	0	0	1	1	0
0	1	493	0	1	0	0	5	0	0
0	0	1	497	0	1	0	1	0	0
0	0	2	0	487	0	4	0	2	5
0	0	1	23	0	465	0	0	11	0
6	2	0	0	5	1	482	0	4	0
0	1	10	4	0	0	0	481	1	3
1	0	2	10	0	0	0	2	480	5
1	1	0	2	3	1	0	3	7	482

- Client 3:
- Overall Accuracy of Client Model: 0.8899
- Class wise Accuracy:
    - Class 0: 0.9900
    - Class 1: 0.9800
    - Class 2: 0.9660
    - Class 3: 0.9940
    - Class 4: 0.9320
    - Class 5: 0.9560
    - Class 6: 0.9680

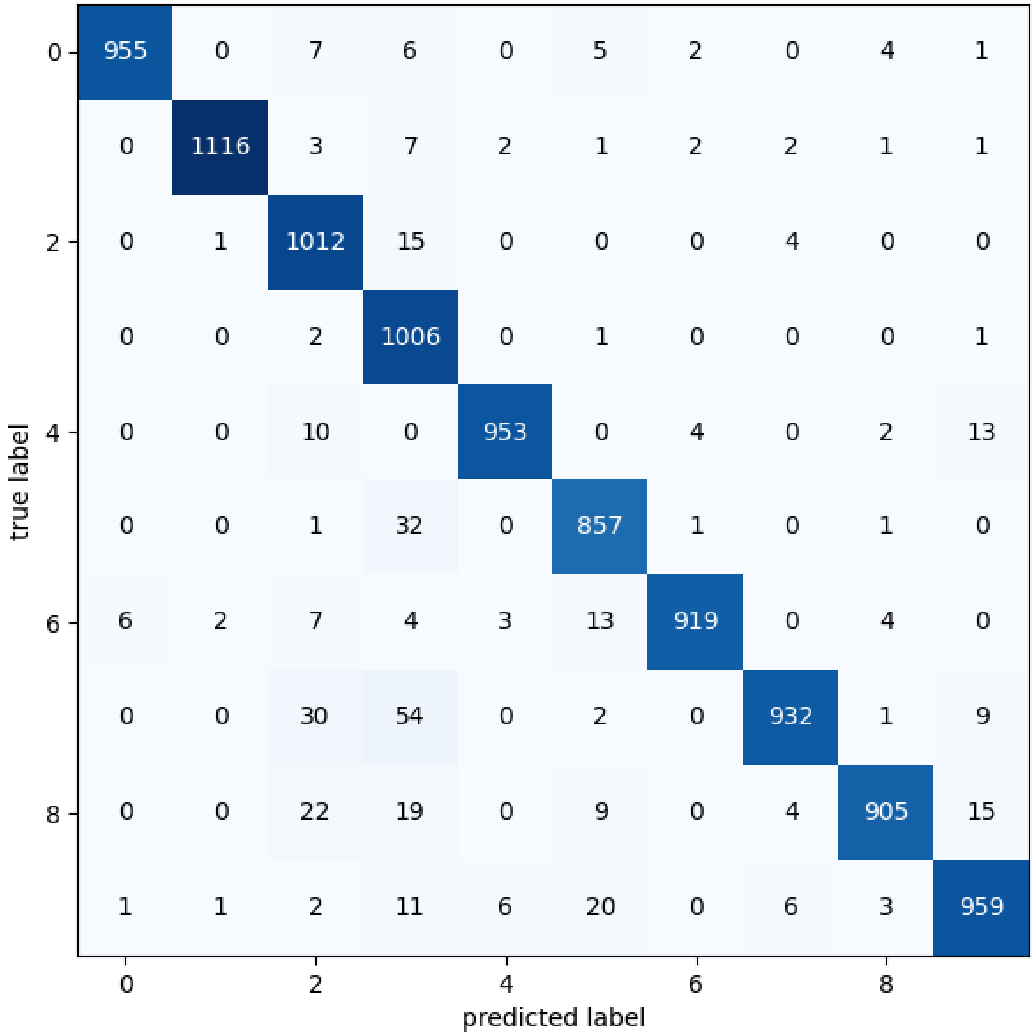
    - Class 7: 0.9380
    - Class 8: 0.9100
    - Class 9: 0.9660
- Confusion Metric



495	0	0	2	0	0	3	0	0	0
0	490	1	7	0	1	0	1	0	0
2	0	483	13	0	0	0	1	1	0
0	0	1	497	0	2	0	0	0	0
1	0	8	0	466	1	6	0	2	16
0	0	1	17	0	478	2	1	1	0
10	3	0	0	1	1	484	0	0	1
0	1	7	18	0	1	0	469	1	3
6	0	8	8	0	3	13	1	455	6
2	3	0	3	1	4	0	2	2	483

- Federated Server Model:
- Overall Accuracy of Client Model: 0.9376
- Class wise Accuracy:
    - Class 0: 0.9745
    - Class 1: 0.9833
    - Class 2: 0.9806
    - Class 3: 0.9960
    - Class 4: 0.9705
    - Class 5: 0.9608
    - Class 6: 0.9593
    - Class 7: 0.9066

  - Class 8: 0.9292
  - Class 9: 0.9504
- Confusion Metric

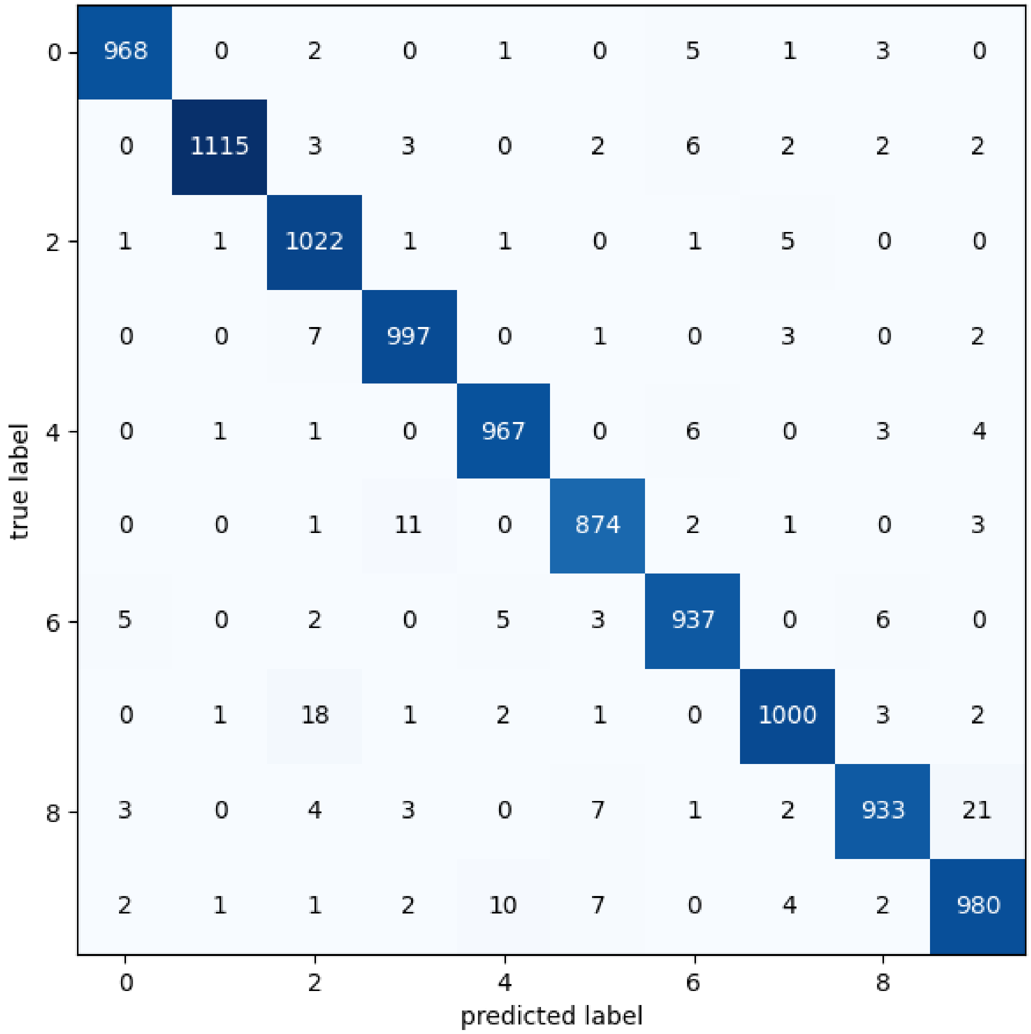


955	0	7	6	0	5	2	0	4	1
0	1116	3	7	2	1	2	2	1	1
0	1	1012	15	0	0	0	4	0	0
0	0	2	1006	0	1	0	0	0	1
0	0	10	0	953	0	4	0	2	13
0	0	1	32	0	857	1	0	1	0
6	2	7	4	3	13	919	0	4	0
0	0	30	54	0	2	0	932	1	9
0	0	22	19	0	9	0	4	905	15
1	1	2	11	6	20	0	6	3	959

- Centralized Based Server:
- Overall Accuracy of Client Model: 0.9552
- Class wise Accuracy:
  - Class 0: 0.9878
  - Class 1: 0.9824
  - Class 2: 0.9903
  - Class 3: 0.9871
  - Class 4: 0.9847
  - Class 5: 0.9798
  - Class 6: 0.9781
  - Class 7: 0.9728
  - Class 8: 0.9579

- Class 9: 0.9713

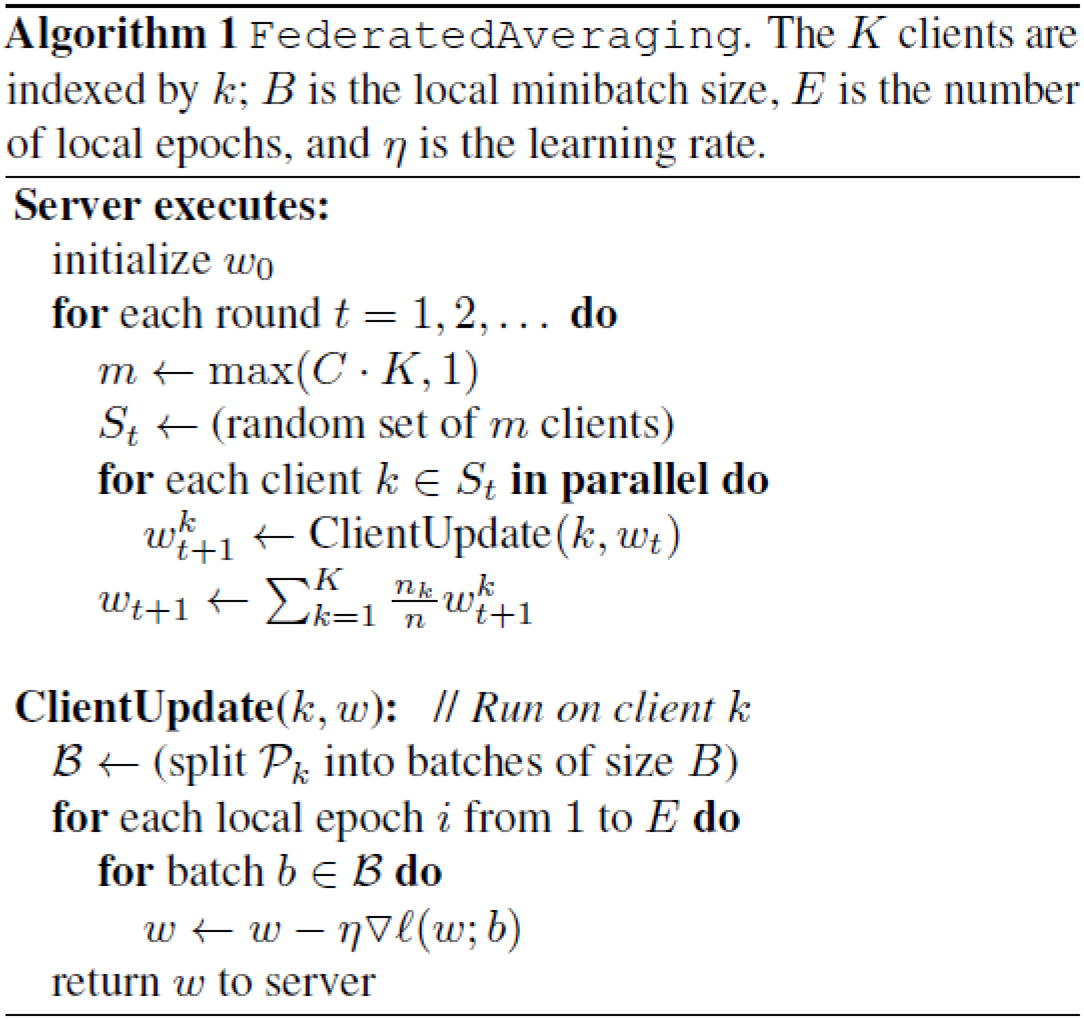
- Confusion Metric



968	0	2	0	1	0	5	1	3	0
0	1115	3	3	0	2	6	2	2	2
1	1	1022	1	1	0	1	5	0	0
0	0	7	997	0	1	0	3	0	2
0	1	1	0	967	0	6	0	3	4
0	0	1	11	0	874	2	1	0	3
5	0	2	0	5	3	937	0	6	0
0	1	18	1	2	1	0	1000	3	2
3	0	4	3	0	7	1	2	933	21
2	1	1	2	10	7	0	4	2	980

Mathematical Explanation:

- The FedAVG (Federated Averaging) technique is frequently employed in federated learning to combine the model updates from the client devices at the central server. According on the quantity of training instances on each client device, the FedAVG algorithm calculates the weighted average of the model parameters across all participating client devices. The following is a mathematical explanation of the FedAVG algorithm:
- Let N represent the overall number of client devices participating in the federated learning system, and let w_t represent the model parameters (such as weights and biases) learned by the i-th client device using local data. A portion of the total training data, designated as n_k, is used for training by each client device. The quantity of training examples on the i-th device is given as n = |n_k|.



Algorithm 1 FederatedAveraging. The K clients are indexed by k; B is the local minibatch size, E is the number of local epochs, and η is the learning rate.

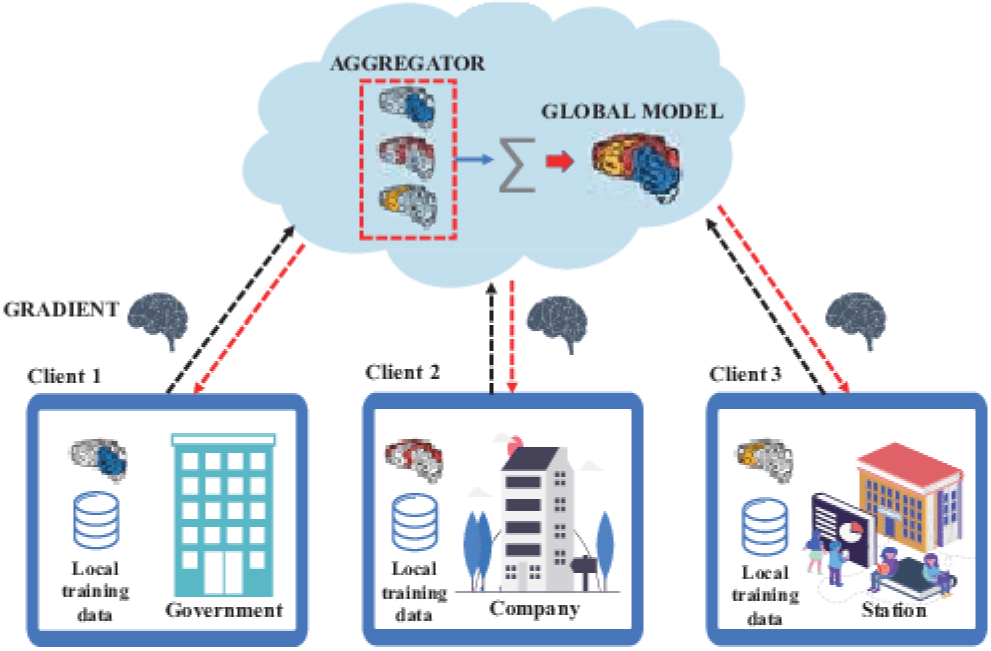
```
Server executes:
  initialize $w_0$
  for each round $t = 1, 2, \dots$ do
    $m \leftarrow \max(C \cdot K, 1)$
    $S_t \leftarrow$ (random set of $m$ clients)
    for each client $k \in S_t$ in parallel do
      $w_{t+1}^k \leftarrow \text{ClientUpdate}(k, w_t)$
    $w_{t+1} \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} w_{t+1}^k$

ClientUpdate($k, w$):   // Run on client k
  $\mathcal{B} \leftarrow$ (split $\mathcal{P}_k$ into batches of size $B$)
  for each local epoch $i$ from 1 to $E$ do
    for batch $b \in \mathcal{B}$ do
      $w \leftarrow w - \eta \nabla \ell(w; b)$
  return $w$ to server
```

- The FedAVG algorithm calculates the weighted average of the model parameters as follows: where |n| is the total number of training examples across all client devices, and w_avg is the averaged model parameters at the central server.
- The weight given to the i-th device in the aggregation process is denoted by the expression (|n-k|/|n|), which indicates the percentage of training examples on that device in relation to the total number of training examples across all devices. This weighting is used by the FedAVG algorithm to make sure that devices with more data have a bigger impact on the aggregated model than devices with fewer data.
- The FedAVG algorithm is an iterative procedure that keeps going back and forth until convergence or a halting requirement is satisfied. The client devices compute their model updates, train their local models on their respective data, and submit them to the central server for aggregation using the FedAVG algorithm in each iteration. The global model is then updated by the central server with the averaging of the parameters, and the modified model is then sent back to the client devices for additional training. This procedure is repeated until the global model reaches an acceptable degree of accuracy.

A detailed explanation of the federated learning algorithm

- In federated learning, a number of client devices train a single machine learning model independently of one another.
- The parameters of the model are transferred to a central server for aggregation after each client has trained its local model on its specific set of data.
- In federated learning, the FedAVG algorithm is a popular aggregation method that calculates the weighted average of the model parameters across all client devices.

$$w_{t+1} \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} w_{t+1}^{k}$$



- The weighted average is calculated by giving each client device a weight based on the quantity of training examples stored on that device.
- Each device is given a weight, which guarantees that devices with more data will have a bigger impact on the aggregated model than devices with fewer data.
- The global model is then created using the aggregated model parameters and delivered back to each client for additional training.

- The FedAVG algorithm is an iterative procedure that keeps going back and forth until convergence or a halting requirement is satisfied.
- The FedAVG algorithm is used by the client devices to compute their model updates, submit them to the central server for aggregation, and train their local models on the relevant data at the conclusion of each iteration.
- The global model is then updated by the central server with the averaging of the parameters, and the modified model is then sent back to the client devices for additional training.
- This procedure is repeated until the global model reaches an acceptable degree of accuracy.
- In order to aggregate model updates in federated learning, the FedAVG algorithm, in essence, computes a weighted average of the model parameters over all participating client devices. The various data sets' privacy and security are upheld while making sure that devices with more data have a greater impact on the aggregated model.

Compare the results of overall accuracy in federated setup with the baseline

	Federated	**Centralized**
Overall Accuracy	0.9393	0.9552
Class wise accuracy		
Class 0	0.9733	0.9878
Class 1	0.9843	0.9604
Class 2	0.9854	0.9797
Class 3	0.9965	0.9861
Class 4	0.9765	0.9725
Class 5	0.9623	0.9552
Class 6	0.9565	0.9572
Class 7	0.9076	0.9698
Class 8	0.9275	0.9405
Class 9	0.9553	0.9713

- Similar results could be obtained, or it might even be able to obtain better accuracy than with a centralised model, if we attempted to train client's models in real time and then transfer the weight to a federated server.

- Just 1% to 2% separates each class' accuracy, which is similar across the board. However, we get a 7% difference for class 7.
- Due to the fact that in the centralised model we train, test, and evaluate in the same system at a serial instance, whereas in the federated model there is a security issue to clients so in order to maintain it they send the train weights to the server, we have compared both federated and centralised models and have observed that there is a difference of 2% accuracy between them. After combining all of the n values in terms of the average, a common final weight is produced.
- As a result, accuracy of 1-3% is regarded as trivial and can be disregarded because it is the average of all client model weights.
- For several reasons, including the following, a federated model may be less precise than a centralised model:
    - Distribution of the data: In federated learning, the data is divided across a number of client devices, and each one creates a local model with the data that pertains to it. The data distribution might not be an accurate representation of the distribution of data as a whole, as opposed to a centralised model that has access to all data, which could result in decreased accuracy.
    - Only a certain number of communication rounds are permitted during federated learning between client devices and the central server. If there aren't enough rounds of communication to achieve a good response, the federated technique might be less accurate than the centralised form.
    - Heterogeneous devices: In federated learning, client devices may have different hardware configurations, network configurations, or training data sizes. This heterogeneity may result in inconsistent updates to the model and slower convergence compared to a centralised model.
    - Privacy protection: Federated learning aims to safeguard the privacy and confidentiality of distinct data sets. The training data are not accessible by either the central server or the client devices. This privacy-preserving technique may limit the quantity of data that is accessible for training and result in reduced accuracy when compared to a centralised model having access to all data.
- It's important to keep in mind that some activities will benefit from federated learning more than others. Federated learning attempts to make collaborative model training possible while upholding privacy and security. A federated strategy that preserves privacy while retaining a respectable level of accuracy would be considered successful.

References:

- https://towardsdatascience.com/federated-learning-a-simple-implementation-of-fedavg-federated-averaging-with-pytorch-90187c9c9577
- https://www.kaggle.com/code/puru98/federated-learning-pytorch
- https://github.com/AshwinRJ/Federated-Learning-PyTorch/tree/master/src
- https://flower.dev/docs/example-pytorch-from-centralized-to-federated.html

- https://shreyansh26.github.io/post/2021-12-18_federated_optimization_fedavg/
- https://www.dam.brown.edu/drp/talks/OhJoonKwon.pdf
- Also from class videos recording.