AML Assignment-2 Report

Code Explanation:

Dataset:

- Libraries: Import necessary Python libraries:
- numpy (as np) for numerical operations.
- os for file system operations.
- sys for system-related functionalities.
- random for random number generation.
- torch for PyTorch, a popular deep learning framework.
- torchvision for PyTorch's computer vision library.
- transforms from torchvision for data transformation operations.
- Seed and Variables: Set random seeds and define key variable
- random.seed(1) and np.random.seed(1) set random seeds to ensure reproducibility.
- num_clients is set to 20, representing the number of clients in a federated learning scenario.
- num_classes is set to 10, indicating the number of classes in the MNIST dataset.
- dir_path specifies the directory where the dataset will be stored, which is "mnist/".
- Function generate_mnist: Defines a function for generating and partitioning the MNIST dataset for federated learning. It takes several arguments, including the directory path (dir_path), the number of clients (num_clients), the number of classes (num_classes), whether the data should be non-IID (niid), whether it should be balanced (balance), and a partition scheme (partition).Checking Dataset: This section checks whether the dataset and configuration files already exist in the specified directory (dir_path). If they exist, it skips the dataset download to prevent redundant downloads.Download MNIST Data: It sets up an HTTP request with a user agent and downloads the MNIST dataset. The downloaded data is stored in the "rawdata" subdirectory of the specified directory.
- Data Transformation: The MNIST data is transformed using torchvision transforms, including conversion to PyTorch tensors and normalization of pixel values to the range [-1, 1].
- Data Splitting: The code splits the MNIST dataset into training and testing sets. It creates data loaders for both sets, which are used for batch processing during training and testing.
- Data Separation: This section separates the images and labels from the dataset and converts them into NumPy arrays. The images and labels from both the training and testing sets are combined into two arrays, dataset_image and dataset_label.

- Data Partitioning: The separate_data function is called with the dataset images and labels, along with the specified parameters (num_clients, num_classes, niid, balance, partition). This function partitions the data based on the specified criteria and saves the partitioned data to files.

Main File:

- Import Libraries: Import necessary Python libraries for the experiment.
- Set Hyperparameters: Define hyperparameters for the experiment, such as the dataset, model, batch size, learning rates, number of rounds, and other algorithm-specific parameters.
- Main Function run(args):
- The primary function that orchestrates the federated learning experiment.
- Iterates over multiple runs of the experiment (controlled by args.times) and records execution time.
- For each run, it:
- Selects the appropriate model based on the chosen algorithm and dataset.
- Creates a federated server based on the selected algorithm.
- Trains the federated server using the specified algorithm.
- Algorithm Selection:
- Depending on the specified algorithm (args.algorithm), the code creates an instance of the corresponding federated server class, such as FedAvg, pFedMe, FedProx, etc.
- Model Selection:
- Depending on the specified model (args.model), the code creates an instance of the corresponding model architecture, such as Convolutional Neural Networks (CNN), ResNet, LSTM, etc.
- Dataset Preparation:
- The script assumes that dataset preparation has already been done or is handled separately. It doesn't download or preprocess data within the script.
- Command Line Arguments:
- Parse command-line arguments to configure the experiment.
- Logging and Profiling:
- Sets up logging and performance profiling tools to monitor and measure the execution of the experiment.
- Run Experiments:
- Calls the run(args) function to perform multiple runs of federated learning experiments.
- Report and Print Results:
- The script reports the average time taken for the experiment and may calculate other metrics or results based on the specific goals and algorithms used.

- Cleanup and End:
- The script performs any necessary cleanup or resource release and then prints a completion message.

pFedMe:

- Import Statements: The code begins by importing necessary Python libraries and modules, including os, copy, h5py, and custom modules for federated learning components such as clientpFedMe and Server.
- pFedMe Class Definition: This is a custom server class for pFedMe federated learning. It inherits from the base Server class, which likely contains common federated learning logic. The __init__ method initializes the pFedMe server with various parameters, sets slow clients, and selects clients of the clientpFedMe type.
- Training Logic (train Method): The train method is responsible for executing the pFedMe federated learning algorithm. It consists of several key steps:
- Selecting clients for each global round.
- Sending models to selected clients.
- Evaluating personalized models periodically.
- Iterating through selected clients and calling the train method on each.
- Aggregating parameters, both normally and with a parameter beta (for pFedMe).
- Optionally, calling a distributed lossy gradient evaluation (DLG) function.
- Optionally, breaking early from the training loop if a certain condition is met.
- Beta Aggregation (beta_aggregate_parameters Method): This method aggregates the average model with the previous model using a parameter beta. This is a crucial step in the pFedMe algorithm, as it combines information from previous global models.
- Testing Metrics (test_metrics_personalized Method): This method calculates testing metrics for the personalized models. It involves iterating through clients and obtaining metrics, taking into account new clients if the eval_new_clients flag is set.
- Training Metrics (train_metrics_personalized Method): Similar to testing metrics, this method computes training metrics for personalized models, considering new clients if needed.
- Personalized Evaluation (evaluate_personalized Method): This method performs personalized model evaluation, including testing and training metrics. The results are stored in various arrays (rs_test_acc_per, rs_train_acc_per, and rs_train_loss_per), and the results are printed.
- Results Saving (save_results Method): This function saves the results of the federated learning experiment to an HDF5 file. It records accuracy, loss, and

other metrics for both training and testing. The results are saved under specific filenames based on the dataset, algorithm, goal, and experiment run.

pFedMeOptimizer:

- Purpose: It's a custom optimizer class designed for federated learning, specifically for the pFedMe algorithm.
- Inherits from: It extends the base PyTorch Optimizer class.
- Initialization:
- Takes parameters params, lr, lamda, and mu during initialization.
- params: List of model parameters to be optimized.
- lr (Learning Rate): Controls the step size during updates (default is 0.01).
- lamda: Controls the trade-off between local and global model updates (default is 0.1).
- mu: Regulates another aspect of the trade-off between local and global updates (default is 0.001).
- step Method:
- Called during training to update model parameters.
- Takes local_model and device as arguments.
- Updates each parameter based on a combination of:
- Standard gradient descent (first term).
- Regularization (second term) with lamda.
- A term involving mu.
- Updates are applied based on the local model's difference from the global model.
- The updated parameters are returned.
- Usage: Typically used in federated learning setups, especially when implementing the pFedMe algorithm. It balances global and local model updates according to the algorithm's hyperparameters.
- Customization:
- The optimizer is customized for pFedMe and may not be suitable for standard deep-learning tasks without adjustments.

Results:

nohup: ignoring input
==
Algorithm: pFedMe
Local batch size: 64
Local steps: 1
Local learing rate: 0.005
Local learing rate decay: False
Total number of clients: 20

```
Clients join in each round: 1.0
Clients randomly join: False
Client drop rate: 0.0
Client select regarding time: False
Running times: 1
Dataset: mnist
Number of classes: 10
Backbone: dnn
Using device: cuda
Using DP: False
Auto break: False
Global rounds: 2000
Cuda device id: 0
DLG attack: False
Total number of new clients: 0
Fine tuning epoches on new clients: 0
==================================================

============= Running time: 0th =============
Creating server and clients ...
DNN(
  (fc1): Linear(in_features=784, out_features=100, bias=True)
  (fc): Linear(in_features=100, out_features=10, bias=True)
)

Join ratio / total clients: 1.0 / 20
Finished creating server and clients.


-------------Round number: 2000-------------

Evaluate personalized model
Average Test Accurancy: 0.9688
Average Test AUC: 0.9842
Average Train Loss: 0.0614

Best accuracy.
0.9692711903129998

Average time cost: 14999.86s.
```

Length: 2001
std for best accurancy: 0.0
mean for best accurancy: 0.9692711903129998
All done!

Storage on cuda:0

Total Tensors: 6599330   Used Memory: 19.16M
The allocated memory on cuda:0: 35.41M
Memory differs due to the matrix alignment or invisible gradient buffer tensors
----

nohup: ignoring input
==
Algorithm: pFedMe
Local batch size: 128
Local steps: 1
Local learing rate: 0.005
Local learing rate decay: False
Total number of clients: 20
Clients join in each round: 1.0
Clients randomly join: False
Client drop rate: 0.0
Client select regarding time: False
Running times: 1
Dataset: Tiny-imagenet
Number of classes: 200
Backbone: resnet
Using device: cuda
Using DP: False
Auto break: False
Global rounds: 2000
Cuda device id: 0
DLG attack: False
Total number of new clients: 0
Fine tuning epoches on new clients: 0
==

============= Running time: 0th =============
Creating server and clients ...

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),
bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
```

```
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
```

```
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
```

```
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=200, bias=True)
)

Join ratio / total clients: 1.0 / 20
Finished creating server and clients.

-------------Round number: 0-------------

Evaluate personalized model
Average Test Accurancy: 0.0044
Average Test AUC: 0.0041
Average Train Loss: 5.3180

-------------Round number: 1-------------

Evaluate personalized model
Average Test Accurancy: 0.0097
Average Test AUC: 0.0088
Average Train Loss: 5.4337

-------------Round number: 2-------------

Evaluate personalized model
Average Test Accurancy: 0.0149
Average Test AUC: 0.0146
Average Train Loss: 5.3872

-------------Round number: 3-------------

Evaluate personalized model
Average Test Accurancy: 0.0148
Average Test AUC: 0.0151
Average Train Loss: 5.3856

-------------Round number: 4-------------

Evaluate personalized model
Average Test Accurancy: 0.0243
Average Test AUC: 0.0235
```

Average Train Loss: 5.3749

-------------Round number: 5-------------

Evaluate personalized model
Average Test Accurancy: 0.0328
Average Test AUC: 0.0325
Average Train Loss: 5.3507

-------------Round number: 6-------------

Evaluate personalized model
Average Test Accurancy: 0.0362
Average Test AUC: 0.0374
Average Train Loss: 5.3385

-------------Round number: 7-------------

Evaluate personalized model
Average Test Accurancy: 0.0387
Average Test AUC: 0.0389
Average Train Loss: 5.3466

-------------Round number: 8-------------

Evaluate personalized model
Average Test Accurancy: 0.0475
Average Test AUC: 0.0483
Average Train Loss: 5.3119

-------------Round number: 9-------------

Evaluate personalized model
Average Test Accurancy: 0.0539
Average Test AUC: 0.0530
Average Train Loss: 5.3075

-------------Round number: 10-------------

Evaluate personalized model

Average Test Accurancy: 0.0588
Average Test AUC: 0.0601
Average Train Loss: 5.2887

-------------Round number: 11-------------

Evaluate personalized model
Average Test Accurancy: 0.0677
Average Test AUC: 0.0667
Average Train Loss: 5.3022

-------------Round number: 12-------------

Evaluate personalized model
Average Test Accurancy: 0.0733
Average Test AUC: 0.0708
Average Train Loss: 5.2783

-------------Round number: 13-------------

Evaluate personalized model
Average Test Accurancy: 0.0789
Average Test AUC: 0.0781
Average Train Loss: 5.2594

-------------Round number: 14-------------

Evaluate personalized model
Average Test Accurancy: 0.0826
Average Test AUC: 0.0820
Average Train Loss: 5.2611

-------------Round number: 15-------------

Evaluate personalized model
Average Test Accurancy: 0.0858
Average Test AUC: 0.0853
Average Train Loss: 5.2470

-------------Round number: 16-------------

Evaluate personalized model
Average Test Accurancy: 0.0893
Average Test AUC: 0.0889
Average Train Loss: 5.2407

-------------Round number: 17-------------

Evaluate personalized model
Average Test Accurancy: 0.0900
Average Test AUC: 0.0900
Average Train Loss: 5.2411

-------------Round number: 18-------------

Evaluate personalized model
Average Test Accurancy: 0.0920
Average Test AUC: 0.0917
Average Train Loss: 5.2336

-------------Round number: 19-------------

Evaluate personalized model
Average Test Accurancy: 0.0938
Average Test AUC: 0.0929
Average Train Loss: 5.2167

-------------Round number: 20-------------

Evaluate personalized model
Average Test Accurancy: 0.0946
Average Test AUC: 0.0933
Average Train Loss: 5.2030

-------------Round number: 21-------------

Evaluate personalized model
Average Test Accurancy: 0.0945
Average Test AUC: 0.0938
Average Train Loss: 5.2045

-------------Round number: 22-------------

Evaluate personalized model
Average Test Accurancy: 0.0949
Average Test AUC: 0.0941
Average Train Loss: 5.1986

-------------Round number: 23-------------

Evaluate personalized model
Average Test Accurancy: 0.0955
Average Test AUC: 0.0946
Average Train Loss: 5.1973

Analysis of Result:

Configuration 1: MNIST Dataset with DNN Backbone

- Algorithm and Dataset: The algorithm used is "pFedMe," and the dataset employed is "MNIST." MNIST is a popular dataset for digit recognition.
- Local Model and Training Parameters: The local model uses a Deep Neural Network (DNN) architecture. The configuration uses a local batch size of 64, one local step per round, a local learning rate of 0.005, and no learning rate decay. It performs a total of 2000 global rounds.
- Federated Learning Settings:
- There are a total of 20 clients in the federated learning setup.
- In each round, one client participates, and clients don't join randomly.
- No clients drop out during training, and client selection isn't based on time.
- It doesn't use Differential Privacy (DP) for privacy protection.
- The "Auto break" feature is disabled.
- The server uses a CUDA device with ID 0.
- DLG (Distributed Local Model Update with Gradient Replacement) attack is not enabled.
- There are no new clients introduced, and no fine-tuning is performed on new clients.
- Results: After the 2000 rounds, the personalized model achieves an average test accuracy of 0.9688, an average test AUC of 0.9842, and an average training loss of 0.0614. The best accuracy observed is approximately 0.9693.

Configuration 2: Tiny-Imagenet Dataset with ResNet Backbone

- Algorithm and Dataset: This time, the algorithm remains "pFedMe," but the dataset is "Tiny-imagenet." Tiny-imagenet is an image dataset with 200 classes.
- Local Model and Training Parameters: The local model uses a ResNet architecture. The configuration uses a local batch size of 128, one local step per round, a local learning rate of 0.005, and no learning rate decay. Similar to the previous configuration, it also performs a total of 2000 global rounds.
- Federated Learning Settings: The settings for federated learning are the same as in the previous configuration.
- Results: The output displays results for multiple rounds. Initially, the average test accuracy is low (around 0.0044), but it gradually increases over the rounds. By the 23rd round, the average test accuracy reaches approximately 0.0955. This suggests that the model is improving with additional training rounds, though the final accuracy may not be reached within the provided rounds.

References:

- https://proceedings.neurips.cc//paper/2020/file/f4f1f13c8289ac1b1ee0ff176b56fc60-Paper.pdf
- https://github.com/CharlieDinh/pFedMe
- https://github.com/TsingZ0/PFL-Non-IID