Implementation and Evaluation of the Conventional Federated Learning

Donald Burke, Adam Pang, Justin Ceiley
Group 5
Department of Computer Science
Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, NY 14586

***Abstract*—This paper is a continuation of our Project 1 paper, which elaborates on the core principles of Federated Learning (FL) and the recent advancements within the field. For this part of our paper, we implemented a conventional FL model through the detailed tutorial provided by the Flower framework. We included our own modifications to incorporate the Federated Extended MNIST (FEMNIST) dataset with the Flower FL model. We also implemented Data Poisoning into one client with 50% label flipping to show the differences in evaluation metrics compared to an unpoisoned FL model. A user's guide to our code can be found at the bottom of this paper.**

***Index Terms*—Federated Learning (FL); Federated Extended Modified National Institute of Standards and Technology (FEMNIST); Independent and Identically Distributed (IID)**

I. Introduction

Flower is a strong and simple-to-understand federated learning framework that allows developers and researchers to begin working with a Federated Learning model with ease. This is due to the fact that the Flower framework presents a unified solution to the confusion that is the FL field. With easily accessed evaluation functions, aggregation functions, and analytics support, the Flower framework is a strong starting point for the implementation of our own FL model. As such, we decide to utilize the Flower framework. [1]

The Federated Extended MNIST dataset, also known as FEMNIST dataset, is a dataset of just over 810,000 rows of images. These images are of 62 possible characters written by individual writers. These 62 possible characters are comprised of 26 lower-case English alphabetical characters, 26 upper-case English alphabetical characters, and the 10 decimal digits (0-9). The dataset has one default split (train split) when accessed through the Flower dataset on huggingface. [2]

After incorporating the default FL model as detailed by the Flower setup tutorial, we had to make several adjustments to fit the model with the FEMNIST dataset. Afterwards, we modified the simulation to incorporate one poisoned client to evaluate the differences in evaluation metrics.

II. FL Model Training

We followed a methodical way of approaching our FL model training process. This is the conventional model training process that most default FL models will follow.

A. Initialize Global Model

Given that we are working with the FEMNIST dataset and require image classification tasks, we decided that a simple default CNN model would be the way to go. [3] There are several reasons for this.

Firstly, CNNs are well-suited to work with image classification tasks due to its ability to use convolutional layers to automatically extract features from images. These features can be edges, textures, and shapes, among a plethora of other features. For images like CIFAR10, which is used in the tutorial that we follow, a CNN with 2 convolutional layers proves decently effective. For our FEMNIST dataset, which is a set of greyscale images, capturing strokes, shapes, and curves become essential for accurate classification. [4]

Another reason why we decided to go with a CNN is because CNN models are parameter-efficient, where CNNs are less likely to accidentally overfit to their training data. Compared to fully-connected layers, CNNs use shared weights in convolutional layers that allow the network to learn effectively even when given only a few parameters in the model. [3]

When following the Flower framework, we realized that the framework also utilized a CNN for their CIFAR10 dataset. Given that we are using a FEMNIST dataset, we had to make several adjustments to the model itself. [5]

```python
class Net(nn.Module):
    def __init__(self) -> None:
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 4 * 4, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 62)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 4 * 4)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Fig. 1. Our altered CNN Model for the FEMNIST dataset

1) Major Change 1: Differences in channels: CIFAR10 uses images on the RGB scale. The FEMNIST dataset, on the other hand, uses a greyscale with only black and white colors. As such, the values of our layers had to change slightly. As shown in Figure 1, convolutional layer 1 had to receive an input channel of 1, instead of 3. We decided to continue using 6 output channels and a kernel size of 5, which were appropriate for detecting spatial patterns in our images. Detecting small features like stroke and slight changes in curves of the characters worked well with these settings.

```
self.conv1 = nn.Conv2d(1, 6, 5)
```

Fig. 2. Major Change 1: Differenes in channels

In convolutional layer 2, we decided to remain with 6 input channels (the feature maps from the previous conv layer) and 16 output channels. This layer helps the model learn more complex features, and builds off of the combinations of the simpler features detected in convolutional layer 1. At the same time, we chose to remain with a kernel size of 5 again for this layer. This second layer further enahnces the accuracy and detection capabilities of our CNN model, and makes it an effective model for our FEMNIST dataset.

This is shown in Figure 2.

```
self.fc1 = nn.Linear(16 * 4 * 4, 120)
```

Fig. 3. Major Change 2: Adjusting fully connected layers

2) Major Change 2: Adjusting fully connected layers: Our CNN model, influenced by the flower framework's model, also uses 3 fully connected layers to further improve its image classification accuracy. Our next change compared to the original CIFAR10 design was to change the first fully connected layer's input size. The input size was initially 400 (16 * 5 * 5). However, the output of our convolutional layers had to be 256 due to our changes in Major Change 1.

In the same way, we needed to make this change within the forward function, where our x.view operation needed to take our flattened fully connected layer size of 256 (16 * 4 * 4).

This change is shown in Figure 3.

```
self.fc3 = nn.Linear(84, 62)
```

Fig. 4. Major Change 3: Difference in classification pool

3) Major Change 3: Difference in classification pool: Compared to CIFAR10's 10 unique classes, FEMNIST has 62 unique classes. Our output of the third fully connected layer had to be changed from (84, 10) to (84, 62) to encompass this difference. This is shown in Figure 4.

B. Global Model Training & Testing

Our training and testing methods remained the same as the Flower framework's training and testing methods. Essentially, their methods simply trained the neural network on the training set, and evaluated the network on the entire test set, respectively.

```
Epoch 1: validation loss 0.018306780960143206, accuracy 0.8109419132997667
Epoch 2: validation loss 0.016721308095481082, accuracy 0.8296696549183348
Epoch 3: validation loss 0.016864216954054007, accuracy 0.8317573375905686
Epoch 4: validation loss 0.01786955998325017, accuracy 0.8303450816652339
Epoch 5: validation loss 0.018778498320088082, accuracy 0.8250644725531131
Final test set performance:
        loss 0.017598521832392543
        accuracy 0.8308708216000207
```

Fig. 5. Initial Global Model training and testing results.

For training and testing our initial global model, we decided to train for 5 epochs and test 5 epochs. The results are shown in Figure 5.

C. Send Model to 10 Clients

The task of Federated Learning is delegated to a number of clients, in our case 10. Usually, these clients exist as different machines all separate from each other and the server. In our implementation, we are putting the server and all of the clients on the same machine. With this comes the added challenge of managing the memory of the machine since we will be running multiple instances of client applications and the server application.

```
def fit(self, parameters, config):
  """ Get parameters from server, train model, return to server.

  This method recieves the model parameters from the server and then
  trains the model on the local data. After that, the updated model
  parameters are returned to the server.
  """
  set_parameters(self.net, parameters)
  train(self.net, self.trainloader, epochs=1)
  return get_parameters(self.net), len(self.trainloader), {}
```

Fig. 6. The fit method from the FlowerClient class

D. Train Model on Clients

The training of the model is delegated to the clients by the server. As mentioned before, the clients and server are all on the same machine. In order to manage memory, the server will spin up clients only when they are needed for training or evaluation. After the client has finished training or evaluation, the client is usually discarded to free up memory. Each client is identified by a partition ID. This ID is used not only to identify each client, but it also specifies a partition of the data to load for each particular client. Since the clients are discarded after they are done training or evaluation, the partition IDs are stored in node_config dictionary within the Context object. This is because the Context object acts as the persistent memory as the training rounds go on. The fit method from the FlowerClient class is shown in Figure 6.

The actual training process starts by receiving the parameters from the server and setting those in the client model. After that, the client employs the training function by passing it the model to use, the trainloader and the number of epochs to use.

E. Return Model Updates to Server

After the training is done, the parameters are retrieved from the model and the returned to the server. In our case, the function simply returns these values from the fit function since the clients and server are on the same machine.

```
strategy = AggregateCustomMetricStrategy(
    fraction_fit=1.0,  # Sample 100% of available clients for training
    fraction_evaluate=1.0,  # Sample 100% of available clients for evaluation
    min_fit_clients=10,  # Never sample fewer than 10 clients for training
    min_evaluate_clients=10,  # Never sample fewer than 10 clients for evaluation
    min_available_clients=10,  # Wait until all 10 clients are available
)
```

Fig. 7. The strategy object from the server_fn function

F. Aggregate Model Updates

For our federated learning approach, we are using a custom subclass of the Federated Averaging class, FedAvg, that is provided by Flower. With this custom subclass, AggregateCustomMetricStrategy, we are able to customize the method of aggregation to gather the required metrics and implement a weighted average when aggregating our accuracy scores. We set the fraction_fit parameter to 1.0 to mean that we want to sample 100% of the clients that are available for training. fraction_evaluate is set to 1.0 as well, to mean that we want to sample 100% of available clients for evaluation. min_fit_clients and min_evaluate_clients are both set to 10 to mean that we never want to sample less than 10 clients for training and evaluation. The min_available_clients parameter is set to 10 so that we wait until all 10 clients are available. The strategy object is shown in Figure 7.

G. Iterate Updates Until Model Converges

The training that has been going on thus far continues until the model's performance no longer significantly improves or simply put, converges. Each time a client finishes training, it sends it's model back up to the server. The server then aggregates this model into the global model. In our case we are using Federated Averaging. This updated global model is then sent to the clients for another round of training. This process continues until the model converges.[6]

III. Simulation (Without Data Poisoning)

For the simulation without any data poisoning present, we see promising results for the CNN's ability to accurately classify the individual handwritten numbers. With 5 epochs of training and 5 epochs of testing, we notice a consistent increase in the average evaluation accuracy among all clients. The average loss also consistently decreases across all rounds in all clients. At the end, we result at an average evaluation accuracy of 84.24%, and an average loss of 1.47%.

A. Evaluation Results

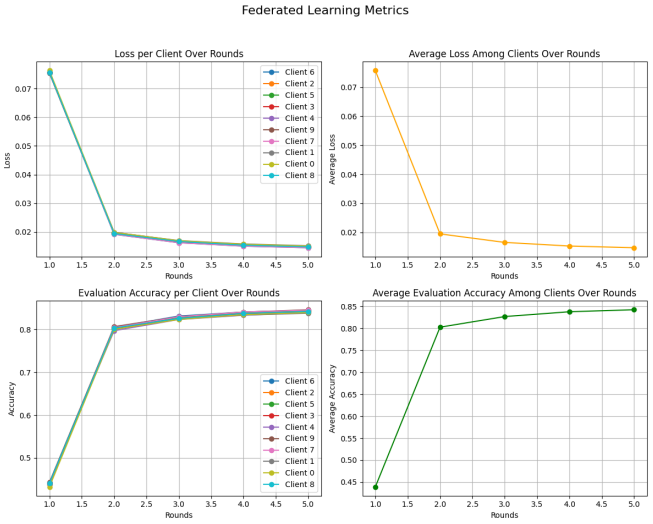

Fig. 8. Collected metrics from our traditional FL simulation

Figure 8 depicts the resulting metrics of our traditional Federated Learning setup. The top left plot shows each individual client's loss over the course of our 5 rounds of training and evaluation. The top right shows the aggregated/averaged loss of all our clients across the 5 rounds. Finally, the bottom left and right graphs show each individual client's accuracy and the aggregated/average accuracy respectively.

By looking into the loss and accuracy metrics, we can gather just how well our model is being trained (or learning). Loss is a numerical representation of "how wrong a model's predictions are". This is important to us to prove that our model is getting better and better at making correct predictions [7]. Similarly, but not quite exactly the same, accuracy is the "proportion of all classifications that were correct". In our case, we are using a weighted average to assure that clients with less data are not underrepresented in the training of our model.

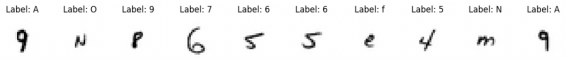

Fig. 9. The testing of our data poison approach. This shows the poisoned/flipped labels (above the images) and their corresponding images (below the labels)

```
def apply_transforms(batch):
  '''
  Applies transformations of pytorch_transforms on every image in place
  '''
  batch["image"] = [pytorch_transforms(img) for img in batch["image"]]

  if (partition_id == 0):
    for i in range(len(batch["character"]) // 2):
      batch["character"][i] = (batch["character"][i] + 1) % 62

  return batch
```

Fig. 10. The changes we made to the 0th client's partitioned training data.

IV. Data Poisoning Process

According to Table 1 of Project 2 specifications, Group 5 needed to poison 1 client with 50% label flipping. We did this by choosing client #0 to be our poisoned client. With the way that we handle our partitioning of the original FEMNIST dataset, we simply check for partition_id 0 and alter the data directly in our code. This is shown in Figure 10.

To do the data poisoning process, we added 1 to the original truth value of the image as defined in batch["character"]. The FEMNIST dataset holds the truth value label of each image in the "character" field as an integer. We added 1 to the integer and performed a modulus operation by 62 to guarantee that each character would be given another character in the FEMNIST character knowledge. Our testing results of the data poisoning is shown in Figure 9.

V. Simulation (With Data Poisoning)

The simulation process for the FL model with Data Poisoning was the same process. The training and testing methods have remained identical to the FL model without Data Poisoning. The default global model has remained the same as well. The only thing altered was the partition of FEMNIST data in the individual client 0 for training. When handling client 0's data, we decided to flip the labels and ensure that the truth values were not correct.

A. Evaluation Results

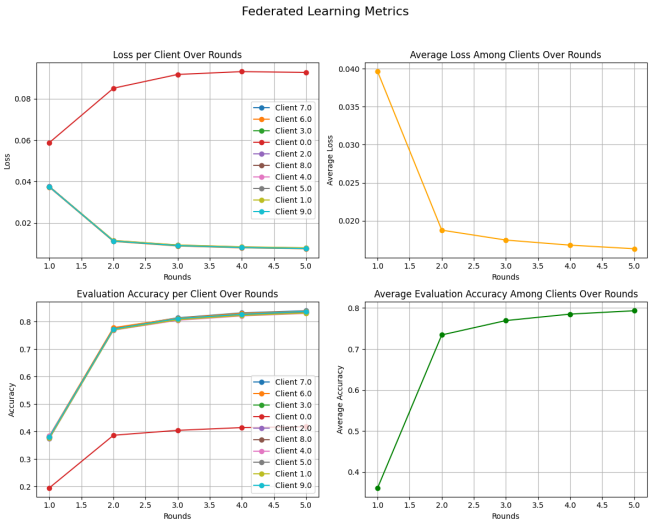

Fig. 11. Collected metrics from our traditional FL simulation with Poisoning

In Figure 11, we see the substantial changes in Client 0. The client has been poisoned with our data poisoning attack, where 50% of the labels have been flipped. As such, we see that the evaluation accuracy grows as the model is trained, but the evaluation accuracy steadily increases towards the maximum of 50% accuracy. This makes sense, since we flipped half of the training set that the client was given. Furthermore, we see that the loss evaluation metric also portrays an increase in loss, rather than the inverse decrease in loss in all the other clients.

When compared with the FL model without data poisoning, we see all 10 clients acting very similar to one another. Each client is reaching just over 80% accuracy, with low loss rates. However, with the data poisoning, we see client 0 act entirely different from the others. Client 0, as shown in Figure 11, has a drastically increasing loss percentage and extremely low accuracy percentage. All other clients act the same as they did in the original FL model.

VI. Conclusion

Federated Learning is a great method to achieving a high amount of accuracy. Compared to the original conventional centralized CNN accuracy of 83%, our FL model without the data poisoning attack reached a CNN accuracy of 84.2%. Although seemingly insignificant, this change in accuracy is very promising. With an increase in clients, batch size, and training epochs, we are confident the difference between the two methodologies will be significantly more apparent. The incorporation of additional security enhancements that come with FL are also beneficial.

We also clearly see the effect of the data poisoning attack on client 0, and its subsequent affect on the global model as well. This is a risk to FL models, and how they can be infiltrated or manipulated. The overall aggregated average of the global model was approximately 4.9% lower in the FL model with the data poisoning compared to the FL model without data poisoning. With a deficit of nearly 5% in accuracy, an increase in almost 0.2% loss was also apparent. This means that FL models are not fool-proof, and they need security measures as well.

VII. User's Guide

Our user's guide will take you through running through our project, installing necessary dependencies, and where you can locate all of our results and reproduce them yourself.

A. Dependencies

- Python3.12, or any later version
- Pip: Install and upgrade pip. We are using the most up to date pip version. This can be done with 'python -m pip install –upgrade pip' on linux and MacOS.
- Flower: flwr[simulation]
- Flower Datasets: flwr-datasets[vision]
- PyTorch: torch
- TorchVision: torchvision
- MatPlotlib: matplotlib
- Pandas: pandas

The dependencies can be installed with one call, without a newline:

```
pip install -r requirements.txt
```

B. Running the Federated Learning Model

Both models (Poisoned and unpoisoned) are run with the same python script 'run_simulation.py'. You can view the help command by including a -h flag.

The program takes one argument, which takes the characters 'p' or 'u' as its inputs.

The following line will run the model with client 0 as a poisoned client.

```
python3 run_simulation.py p
```

The following line will run the model withOUT any data poisoning.

```
python3 run_simulation.py u
```

Note: If rerunning the program, you must delete the metrics.csv and metrics.png files to generate new plots and CSVs. Otherwise, the files will become concatenated and built on top of one another.

VIII. Work Division

For Project 2, we split up the work between the three of us evenly. We believe that everyone contributed a fair amount to the project and each of us dedicated around 8 hours to the project.

A. Adam

- Initialized Global Model: Set up the initial global model - 2 hours
- Altered Global Model: Calculated changes to matrix size, input sizes of the conv layers, and fixed all bugs related to the global model - 1 hours
- Report: Wrote Introduction, FL Model Training (Sections A and B) - 1 hour
- Report: Wrapped up Conclusion, Data Poisoning Section - 1 hour
- Data Poisoning: Wrote data poisoning code - 1 hour
- Report: Wrote Data Poisoning Process, Simulation, Conclusion - 2 hours

B. Justin

- Federated Learning Code: Followed Flower tutorial to integrate FL model with global model - 2 hours
- FL Model Testing: Ran FL model code with different epochs and testing scenarios - 2 hours
- Report: Wrote Sections C-G - 2 hours
- Data Poisoning: Helped with data poisoning approach - 2 hours

C. Donald

- Report: Helped with Data Poisoning writeup - 2 hours
- CSV Handling: Generated CSVs for metrics - 2 hours
- Metrics: Created metrics for run-time evaluation output - 2 hours
- Visualization Plots: Created plots from metrics - 2 hours

References

[1] Flower, "Flower: A friendly federated learning framework," 2024, accessed: 2024-10-16. [Online]. Available: https://flower.dev/

[2] F. Labs, "Femnist dataset on hugging face," 2024, accessed: 2024-10-16. [Online]. Available: https://huggingface.co/datasets/flwrlabs/femnist

[3] S. Chiang, "Convolutional neural networks — why are they so good for image related learning?" 2018, accessed: 2024-10-16. [Online]. Available: https://towardsdatascience.com/convolutional-neural-networks-why-are-they-so-good-for-image-related-learning-2b202a25d757

[4] TensorFlow, "Convolutional neural network (cnn) tutorial," 2024, accessed: 2024-10-16. [Online]. Available: https://www.tensorflow.org/tutorials/images/cnn

[5] (2024) Flower documentation. [Online]. Available: https://flower.ai/docs/framework/tutorial-series-get-started-with-flower-pytorch.html

[6] K. Martineau, "What is federated learning?" 2022, accessed: 2024-10-16. [Online]. Available: https://research.ibm.com/blog/what-is-federated-learning

[7] (2024) The google developers ml concepts crash course. [Online]. Available: