1. OBJECTIVE

In this project, I have taken up the Aptos 2019 Blindness Detection Challenge available on Kaggle (https://www.kaggle.com/c/aptos2019-blindness-detection/overview).
The objective is to create an Neural Net architecture which can be used to determine the severity of Diabetic Retinopathy from a retinal image. There are 5 classes and they are as follows:

- 0: No DR
- 1: Mild DR
- 2: Moderate DR
- 3: Severe DR
- 4: Proliferative DR

DR: Diabetic Retinopathy

2. ABOUT THE DATASET

The dataset consists of 2 folders (train & test) and 2 csvs (train.csv & test.csv). Both the folders consist of retinal images which the file "train.csv" has labels beside each of the file names in the train folder.
For this exercise, I have taken only a subset of the train dataset and divided it for training and testing purposes.
I have taken a total set of 3100 images. 3000 images have been set for training and 100 images for testing.
These 3000 images have further been divided equally into 4 devices/clients (750 images each) for displaying the process of learning using Federated Learning.

3. SOFTWARES/TECHNIQUES USED

- Federated Learning
- Pytorch
- ResNet-101 (Core Model)
- Pillow
- OpenCV
- Pandas
- Shutil
- Torch
- Torchvision
- ML Flow (Visualisation)

4. DATA REPOSITORY/LOCATION

https://drive.google.com/file/d/1nR3GUr4Nu8FVRtG3bteE1OYDWA4dU09f/view?usp=sharing

5. WHAT IS FEDERATED LEARNING?

Federated learning is a machine learning technique that trains an algorithm across multiple decentralized edge devices or servers holding local data samples, without sending them to the central server. This way, the privacy of the data is maintained.

FL allows for machine learning algorithms to gain experience from a broad range of data sets located at different locations. The approach enables multiple organizations to collaborate on the development of models, but without needing to directly share secure data with each other. Over the course of several training iterations, the shared models get exposed to a significantly wider range of data than what any single organization possesses in-house. In other words, FL decentralizes machine learning by removing the need to pool data into a single location. Instead, the model is trained in multiple iterations at different locations.

6. BRIEF OVERVIEW OF EACH MODULE/FILE

a. Sender

The Sender Module is used to fetch data from each device for every round based on the specified sample size. Each image is preprocessed in this module itself and sent to the Client for training to maintain differential privacy.

- **get_data.py :** This file houses the GetData class which fetches the data from the individual devices based on the sample size each round. It gets activated every time the main function sends a signal before each round of training. After fetching the data and the labels, it sends to the Preprocessing module to preprocess the images and then clubs them into a pickle file and then sends in to the Client module for further training
- **preprocess.py:** This file houses the Preprocess class which is called by the GetData class when every image needs to be preprocessed using Pillow, OpenCV and Torchvision Transforms.

b. Client

The Client Module is used for training the client models after it receives the data for each round from the Sender Module. The Client Module is also responsible for saving each model in the specified directory after each training.

- **train.py:** This file does the major core part of the training for every sample of data received from each device. The core model used is ResNet-101 for training. The metrics that we receive after training are “loss”, “val_loss”, “acc”, “val_acc”. The loss function used is “nn.CrossEntropyLoss” and the optimizer used is “RAdam” along with a learning rate of 0.1
- **radam.py:** This file is a custom file inspired from the newly launched RAdam optimizer is Keras. RAdam stands for Rectified Adam Optimizer.

c. Server

The Server module is used for the final and most critical process of Federated Learning i.e. Aggregation of Client models and individual Client updates from the Aggregated Model.

- **aggregate.py:** This file houses the Aggregate class which receives all the individual client models and then executes simple aggregation (in this example) with equal weightage on all the client models and then computes a Global Aggregated Model. Other sophisticated methods of aggregation include Weighted Averaging, Stochastic Averaging, etc.
- **test.py:** This is an optional file which houses the Test function. This file can be included in the code to test the performance of the aggregated model over a separate set of test data. This file has been created but not activated in the code because it is advised to be run on a Cloud server and it takes up a lot of Cuda memory, such constraints would hold me back while testing the code.

- **updatemodels.py:** This file is an integral part of the Federation process which houses the UpdateModels class. After the aggregation is done and we get a global model, this module helps update the weights of each client model for a better performance in the next round. In this exercise, I have given 70% weightage to the client weights and 30% weightage to the global aggregate weights.

VISUALISATION

The visualisation of losses and accuracies have been integrated into the MLFlow platform which internally uses Plotly for real-time updates on the graphs

NOTE:
The folder “client_models” contains the individual client models and “federated_models” contains the aggregated model

RESULTS:

I ran the whole process for around 6 rounds (out of a total 9 rounds possible with this reduced dataset). Client 4 gave the best performance peaking at around 75-80% accuracy while the other 3 clients averaged around 45-60% accuracy. Federated Learning is slow but steady process and the performance was fairly on the upside.