AI4CI

Distributed and Federated Learning – TP1
Introduction to Flower Framework

Dmytro Rohovyi NTUU

23/05/2025

TP1: Introduction to Flower Framework

Ahmad Dabaja, Caina Figueiredo Pereira and Rachid Elazouzi

Objectives

The first practical session (TP1) aims to introduce you to the implementation of a Federated Learning system using the Flower framework. In this work, you will:

1. **Generating Distributed Data:** You will create a simulated distributed dataset with different class distributions across clients.
2. **Designing a Model for FL:** You will implement a simple machinelearning model to be used by clients in federated training.
3. **Implementing a Federated Client:** You will extend the abstract class flwr.client.Client to define client-side operations.
4. **Running Individual Clients:** You will create a script run client that creates a client with its assigned dataset and model.
5. **Implementing the Server's Client Manager:** You will extend the abstract class flwr.server.ClientManager that allows the server to manage participating clients.
6. **Implementing a Basic Federated Learning Strategy:** You will extend flwr.server.Strategy to define an basic aggregation strategy (FedAvg).
7. **Running the Server:** You will implement and a script start server that launches the federated learning server.
8. **Running a Full FL Simulation:** You will implement and execute the final script run simulation that starts the server and deploys multiple clients for federated training.
9. **Analyzing FL with Different Configurations:** You will conduct multiple simulations with varying hyperparameters (e.g., number of clients, data heterogeneity) and analyze the effects of them on model performance and convergence.

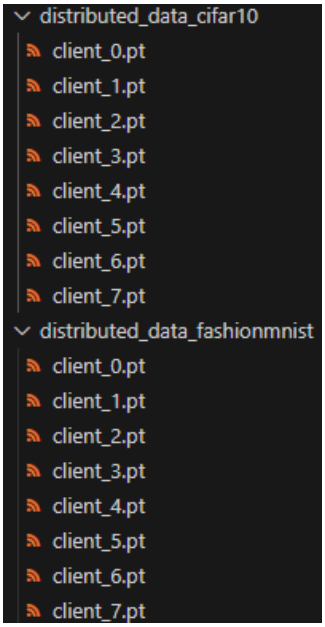
In a nutshell, this TP will provide hands-on experience with Federated Learning, allowing you to explore Flower Framework, client-server interactions, FedAvg training strategy, the effects of different parameters on convergence and performance.

Step 1: Generating and Loading Distributed Datasets

Code with step is located inside data_utils.py

0.1 Task 1: Creating Distributed Datasets

Result of the function is separate folder with dataset divided into different files for different clients. As example this is CIFAR-10 and Fashion MNIST datasets divided for 8 clients. Exact dataset is specified as function's argument and can be passed into code as console line argument when starting server.



Split of data for between users for CIFAR-10 dataset

```
Number of clients: 8
Client 0 has 5154 samples
Client 1 has 8572 samples
Client 2 has 7216 samples
Client 3 has 7203 samples
Client 4 has 9332 samples
Client 5 has 5005 samples
Client 6 has 3656 samples
Client 7 has 3862 samples
```

0.2 Task 2: Loading a Client's Dataset

The function reads data from corresponding .pt file (function gets cid as parameter) splits it into training and validation subsets and returns DataLoader for each subset.

Function don't have any type of output (neither files, nor console) so it's impossible to show any results in the report.

Step 2: Implementing the Federated Learning Model

Since we're going to use multiple datasets with different format of data, 2 separate models were created. The code for this task is located inside model.py

CustomFashionModel is model for Fashion MNIST dataset and its architecture optimized for smaller greyscale images of this dataset:

```
self.network = nn.Sequential(
    nn.Flatten(),
    nn.Linear(28*28, 128),
    nn.ReLU(),
    nn.Linear(128, 64),
    nn.ReLU(),
    nn.Linear(64, 10)
)
```

CustomCifarModel is model for Fashion MNIST dataset and its architecture optimized for bigger RGB images of this dataset:

```
self.network = nn.Sequential(
    nn.Flatten(),
    nn.Linear(3 * 32 * 32, 1024),
    nn.ReLU(),
    nn.Linear(1024, 512),
    nn.ReLU(),
    nn.Linear(512, 128),
    nn.ReLU(),
    nn.Linear(128, 10)
)
```

Step 3: Implementing the Federated Client

Implementation of client is located in client.py

Class contains all basic functions: get_properties(), get_parameters(), fit and evaluate.

Step 4: Running an Individual Client

A file, used to start the client is run_client.py.

It receives cid, batch size and dataset name as console line argument and does this:

1. Reads client's part of dataset via load_client_data() function implemented before, get train and validation data loaders.
2. Initiates model according to dataset name
3. Initiates client class
4. Starts client via fl.client.start_client()

Step 5: Implementing the Server Components

0.3 Implementing the Client Manager

CustomClientManager is implemented in manager.py file and contains all required functions: num_aviable, register, unregister, all, wait_for and sample. In addition to basic functionality some console outputs were added for better logging of whole process and easier debugging.

0.4 Implementing the Strategy

FedAvgStrategy is implemented in strategy.py file and contains all required functions: initialize parameters, configure fit, aggregate_fit, configure_evaluate, aggregate_evaluate and dummy function evaluate(required by flower itself, but never used).

```
def evaluate(
    self,
    server_round: int,
    parameters: Parameters,
) -> Optional[Tuple[float, Dict[str, Scalar]]]:
    return None
```

Same as manager, strategy contains multiple console outputs that allow to follow the learning process and make debug easier.

Step 6: Running the Federated Server

A file, used to start the server is run_server.py, but most of server's code located in server.py in function main_server.

This function receives all training parameters that we will change in tests (number of rounds, number of clients, alpha, number of epochs per round, learning rate, batch size and name of dataset) as arguments.

This function does this:

1. Generates distributed dataset based on amount of clients
2. Initializes client manager
3. Initializes model, depending on dataset
4. Initializes strategy, passing all required parameters to it(number of epochs, and number of clients)
5. Starts the server on localhost:8080 while saving it's history to later pass it to specialized function save_data() that will write it in format, most convenient for analysis.

Functions save_data() receives the history files plus all hyperparameters we're going to test in changes and saves all data in 3 files:

1. fl_history.json – just raw data from server's history, stored in a way task requires.
2. plot1.json – stores loss_eval, accuracy_eval, loss_fit and accuracy_fit curves plus all hyperparameters(number of rounds, number of epochs, number of clients, batch size, learning rate and alpha) of the simulation. Data from multiple runs is appended to this file, making it easier to store results of series of simulation.
3. plot2.json – stores eval accuracy and loss values of the last round for each run plus all hyperparameters. This will later be used to generate plot that shows how accuracy depends on certain hyperparameter and to build a table of results.

Script run_server.py just parses all command line arguments, and passes them to main_server function.

Step 7: Analyzing results

File vizualizer.py contains both implementation of ResultsVisualizer class and main function which allows to use this file to generate all needed plots and table using console command.

The way task requires us to visualize data isn't convenient for analysis, so I changed it. New representation of data is described in previous Step. ResultsVisualizer have this methods:

1. load_data() – simply loads data from 2 json files and saves it inside the class. There's several checks in case files don't exist or they're empty.
2. plot_single_simulation() – converts data from plot1.json file into 2 plots. One with eval accuracy and loss curves and one with fit accuracy and loss curves. This mode is made for deeper analysis of single simulation and won't work for set of runs.
3. plot_multi_simulations() – generates 5 plots. First 4 are made as subplots and show cures of all 4 metrics for all simulations. This way it's easy to compare results of several simulation within one experiment. Second plot is curve of eval accuracy vs hyperparameter, specified by index.
4. print_results_table() outputs data from second json file in formatted table via prettytables. The way this table look was demonstrated in previous step.

The main function parses console line arguments(name of folder with data, index of parameter, plot_type and boolean print_table and dataset). It loads data from specified folder, generates plots, according to specified plot_type and, if –print_table argument was used, prints table to console. If given dataset parameter it'll add one more parameter to data that corresponds to dataset name.

Step 8. Running the First FL Simulation

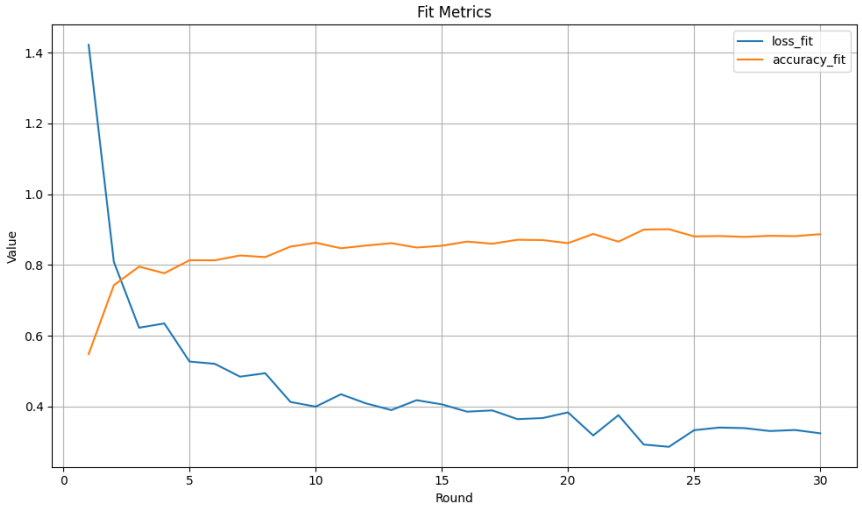
0.5 Hyperparameter Configuration

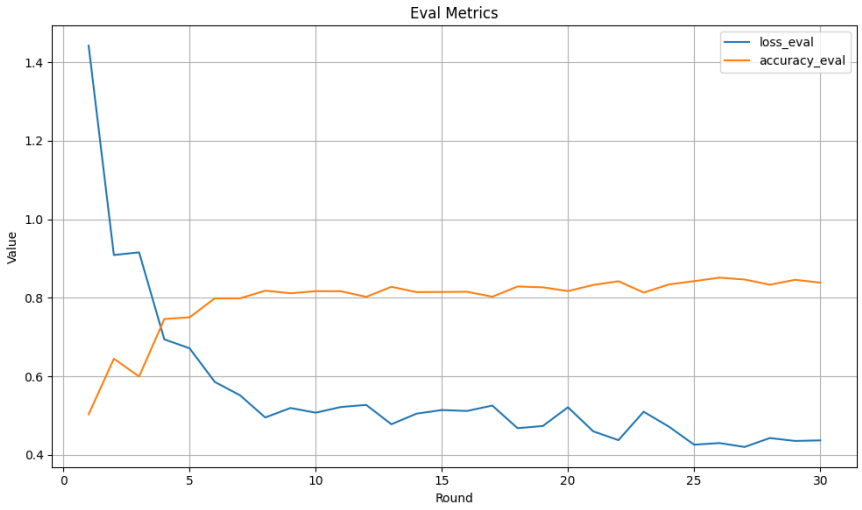
0.6 Running your first Simulation

```
SEED = 42
NUM_ROUNDS = 30
NUM_CLIENTS = 10
EPOCHS = 1
DATASET_NAME = "FashionMNIST"
ALPHA_DIRICHLET = 1
BATCH_SIZE = 32
LEARNING_RATE = 0.01
```

Here’s all results of running simulation with required parameters

rounds	epoch	client	batch	lr	alpha	accuracy	loss
30	1	10	32	0.01	1.0	0.8460910389875484	0.43714689866763773





0.7 Experiment with different Hyperparameters

All the test were automated using experiments.ipynb file. Each experiment is located in 3 code cell within a header. First sell runs the simulation, second sell runs visualizer.py and third cell outputs generated plots. This way tests are much easier to reproduce and data is generated and presented in convenient way for future analysis.

All results of each experiment are stored in separate folder with corresponding name.

Each folder contains:

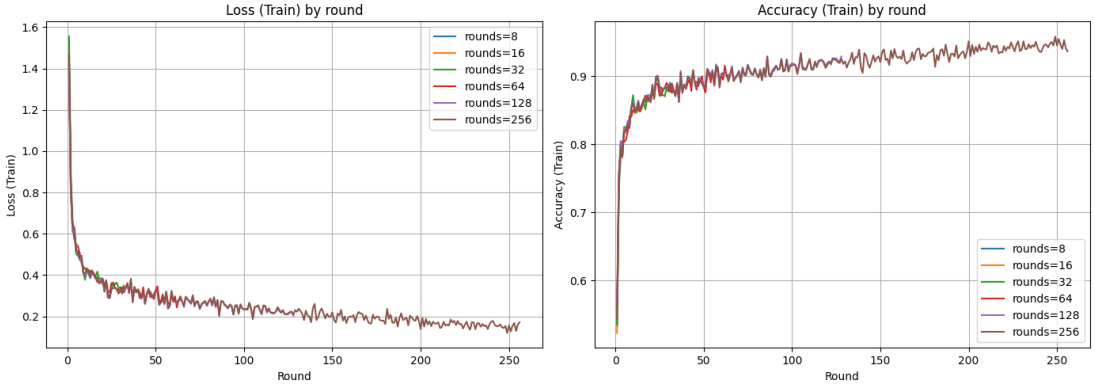
1. Folders with clients' and server's logs, stored in separate files.
2. 3 json files, described in Step 6
3. 2 plots stored as images

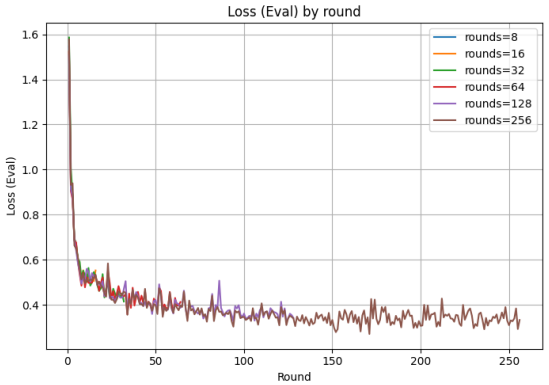
This way it's easy to check results of each experiment even without .ipynb file.

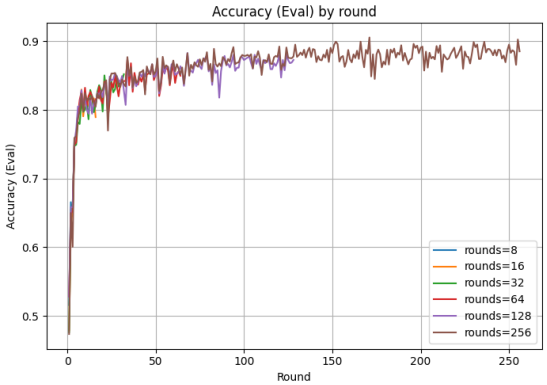
Unfortunately, moodle does not allow to upload such big number of files, so this folders will be removed from folder with code. All results of experiments still will be

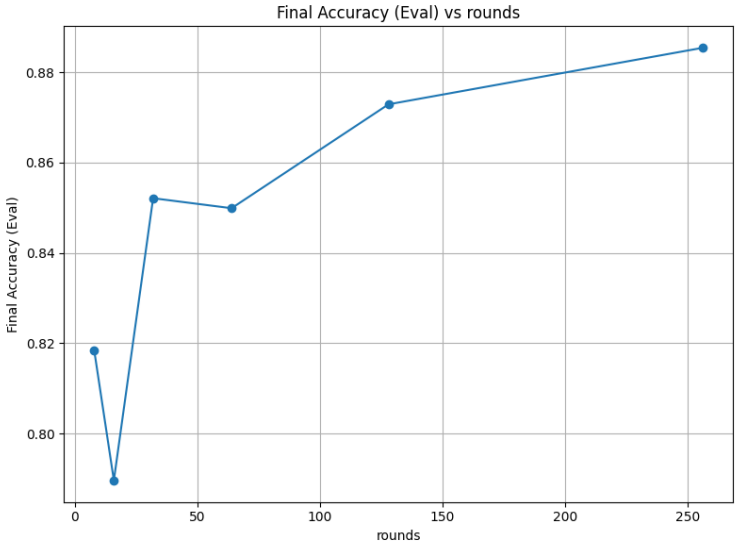
1. Number of rounds

rounds	epoch	client	batch	lr	alpha	accuracy	loss
8	1	10	32	0.01	1	0.8183920313055852	0.4978452095121135
16	1	10	32	0.01	1	0.7896551724137931	0.5527331704723424
32	1	10	32	0.01	1	0.8521008403361344	0.41486575476762627
64	1	10	32	0.01	1	0.8498886414253898	0.4124078724281648
128	1	10	32	0.01	1	0.8728827495477718	0.3417402090965143
256	1	10	32	0.01	1	0.885376756064112	0.33289096984027444







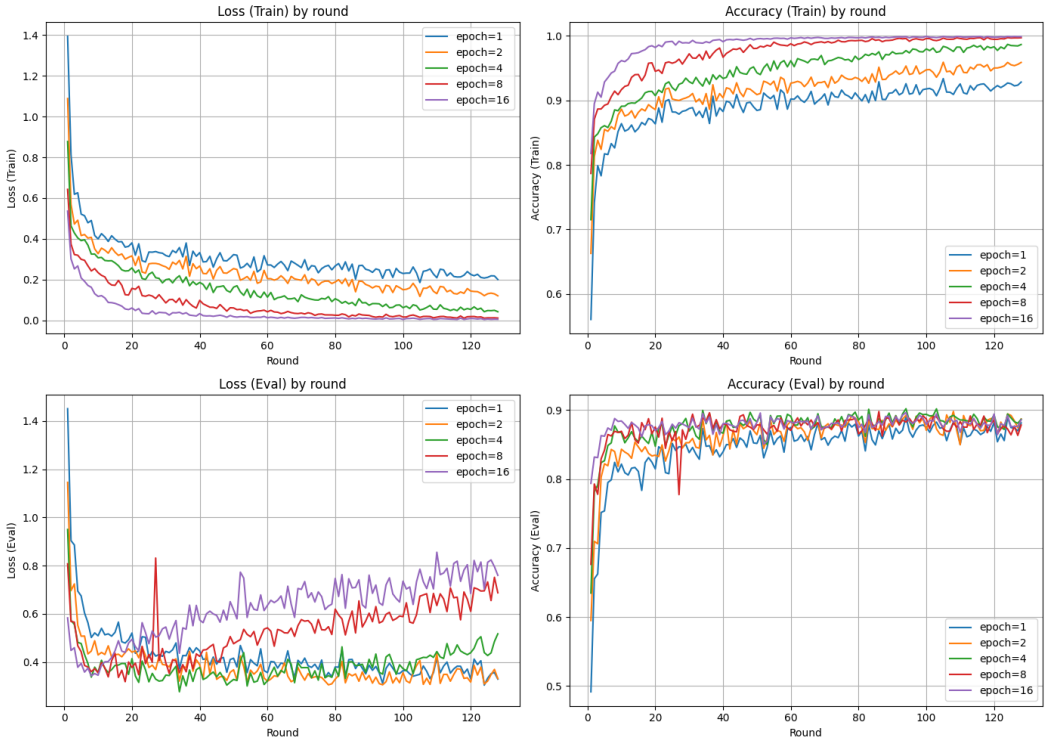


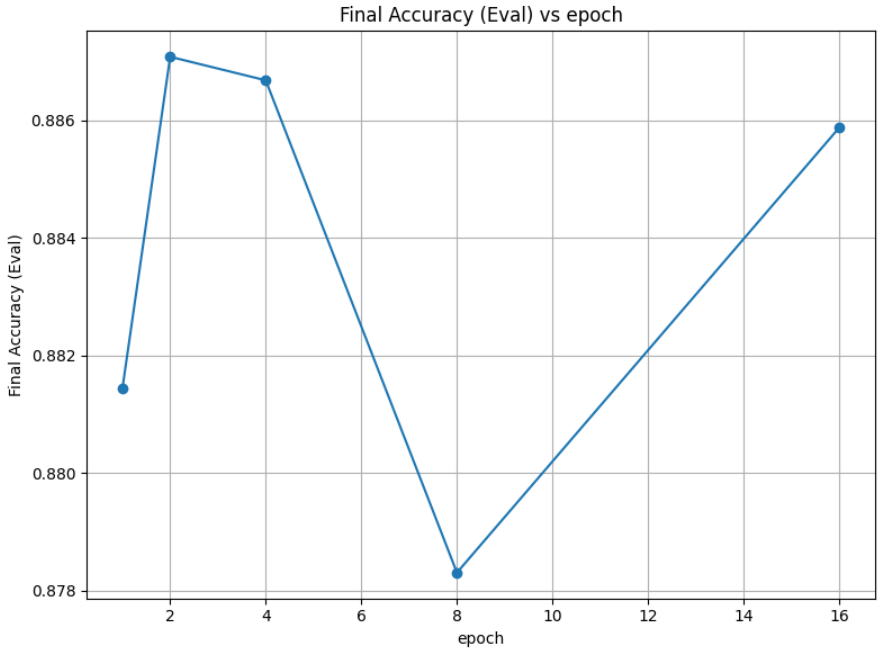
It is evident that higher number of rounds leads to increased accuracy. From both the plot and table we can observe rapid rise in accuracy to 85,2% within first 32 epochs. After this point, model's performance growth begins to slow down, resulting in only 3,3% increase by the 128th epochs. Beyond this, growth become marginal, while training time continues to increase.

The drop in accuracy for 64 rounds can be could be attributed to overfitting, client drift or other common ML issues. Alternatively, since we don't have any recovery mechanisms in place it is possible that overall model's performance was better, but during on 64th round it encountered too much noise, which reduced measured accuracy. We can observe this pattern through whole training process, where accuracy fluctuates by few percentage points.

2. Number of epochs per round

rounds	epoch	client	batch	lr	alpha	accuracy	loss
128	1	10	32	0.01	1	0.8814339746752179	0.329260656054799
128	2	10	32	0.01	1	0.8870779976717112	0.33146457197898915
128	4	10	32	0.01	1	0.8866779089376053	0.5170350580653694
128	8	10	32	0.01	1	0.8783094885709587	0.6874802833108442
128	16	10	32	0.01	1	0.8858740338760073	0.7598646146512859





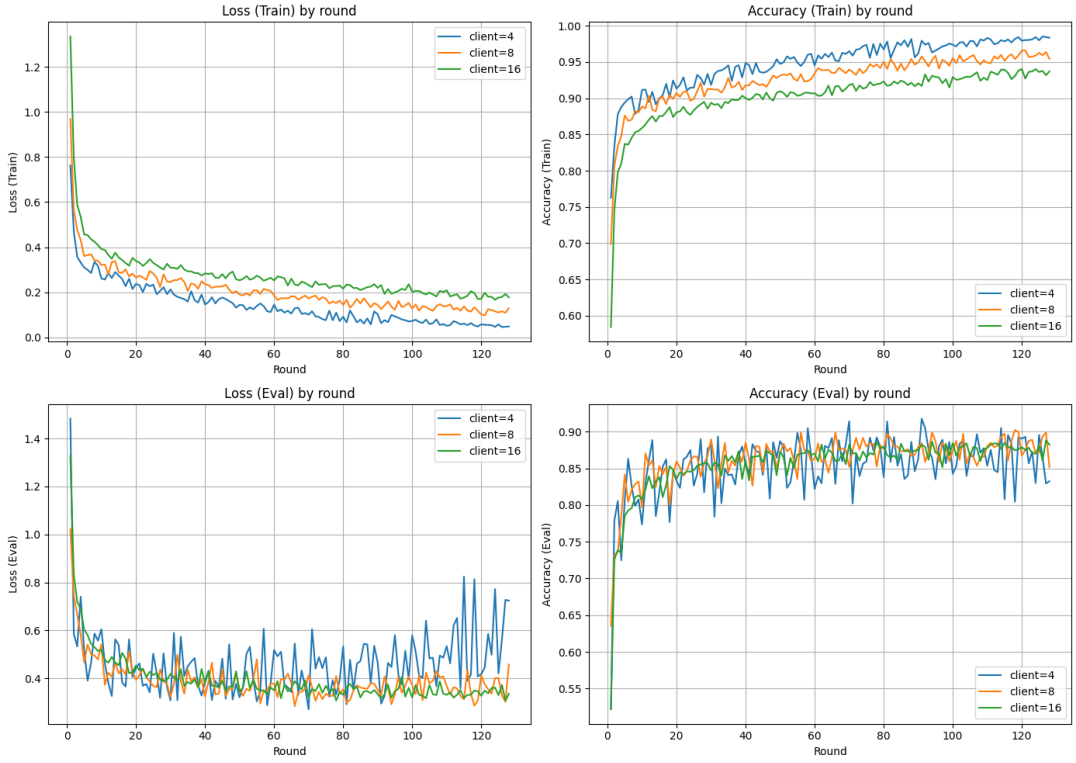
In this case, we observe that accuracy across all tests fluctuates within 87,8-88,7%, which is small, marginal fluctuations. Such limited impact can be explained by overall high accuracy of model, which leaves little room for significant variation.

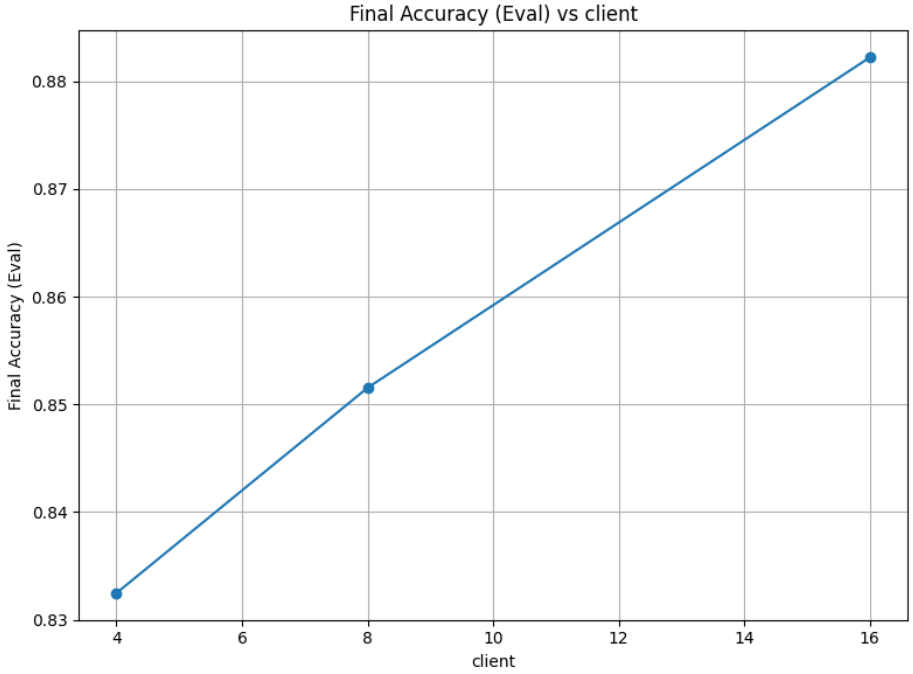
However, when examining evaluation loss curves for higher number of epochs we can see how it starts rising after 20-30 epochs. Loss functions “punishes” model more heavily for high-confidence false classifications than accuracy. This means that even though accuracy remains largely unchanged, higher number of batches leads to more severe overfitting on local clients’ data. As a result, we observe rising evaluation loss curves.

On evaluation accuracy curve we see that models trained with fewer epochs per round take longer to reach plateau. This is most noticeable in the case of 1 epoch per round, where model reaches plateau after around 100 rounds, while model trained with 16 epochs reaches it in less than 20 rounds.

3. Number of Clients

rounds	epoch	client	batch	lr	alpha	accuracy	loss
128	2	4	32	0.01	1	0.8324705882352941	0.7241713698331047
128	2	8	32	0.01	1	0.851534210024776	0.4577907424995916
128	2	16	32	0.01	1	0.8822141846202233	0.33571495553020186



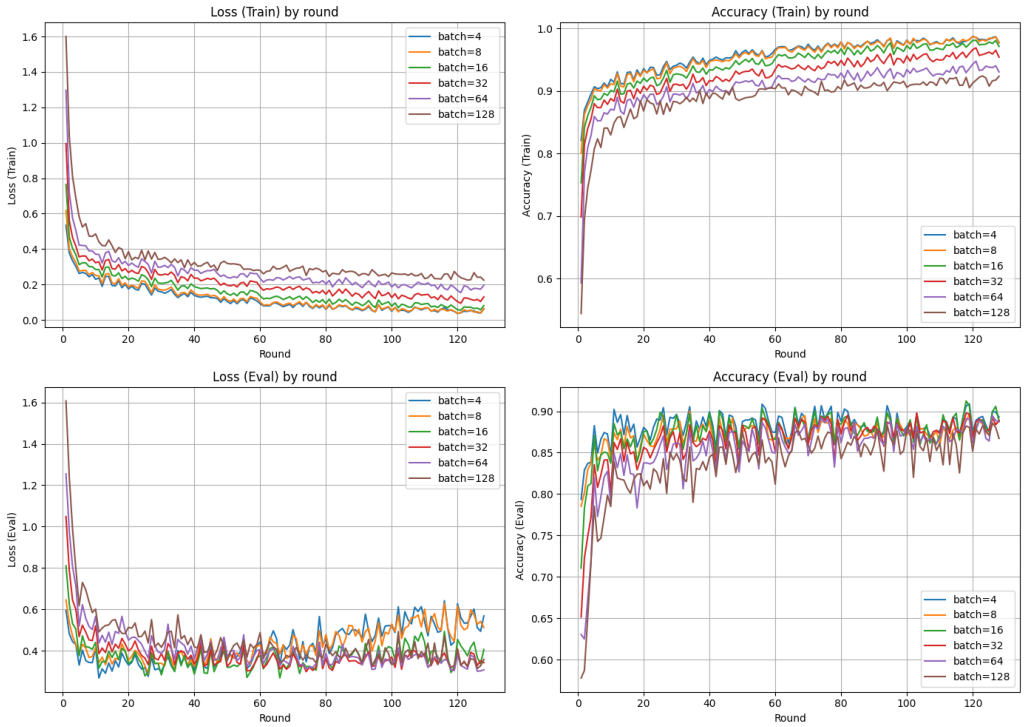


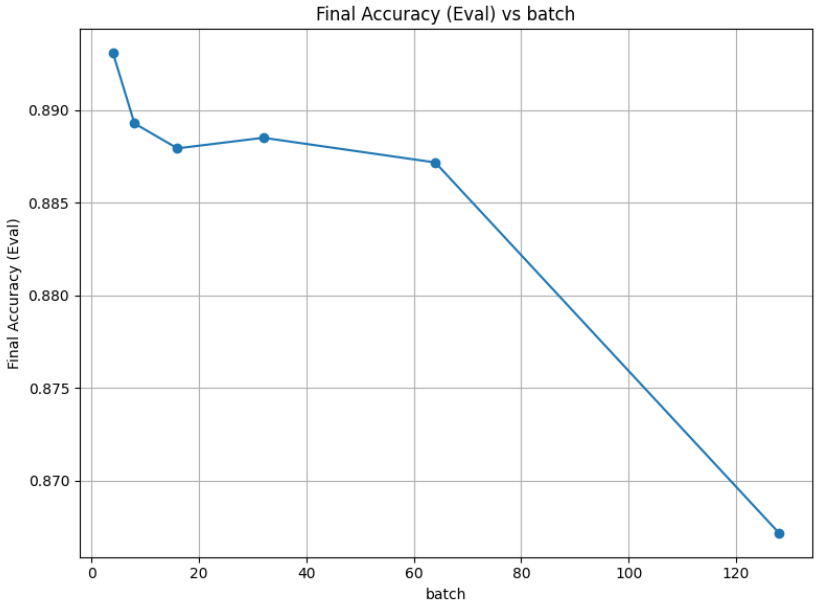
During this test we can see that increasing amount of clients results in higher accuracy. Bigger amount of clients reduces “weight” of each client’s results, thus lowering impact of any noise, leading to more stable convergence. Lowering amount of clients, on the other hand, will increase this impact, causing bigger fluctuations during training process and lower overall performance.

Even though 16 clients yield better results, for further tests only 8 clients will be used. Large amount of clients causes instabilities in simulation process. 32 clients were not tested due to same issue. After each round one or two clients were always shutting down. Logs of falied attempt is saved in corresponding folder. This is error of one of “dead” clients:

4. Batch size

rounds	epoch	client	batch	lr	alpha	accuracy	loss
128	2	8	4	0.01	1	0.8930817610062893	0.5691088164207116
128	2	8	8	0.01	1	0.8892830470500374	0.5128787283279986
128	2	8	16	0.01	1	0.8879359634076616	0.40703663461681366
128	2	8	32	0.01	1	0.888507718696398	0.34380871578639227
128	2	8	64	0.01	1	0.8871736230226797	0.3094604679271337
128	2	8	128	0.01	1	0.8671621879169049	0.35903816650157794





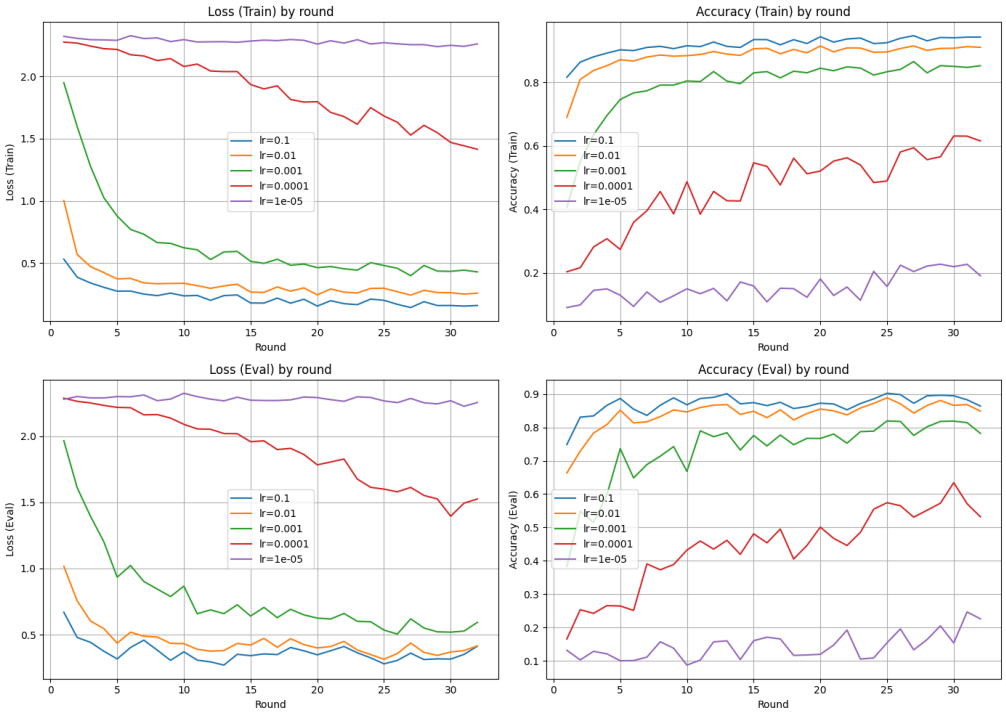
Based on the data, lower batch size yields higher accuracy, however, just like with higher number of epochs, it causes more severe overfitting, judging by rising loss curves for batch size 4 and 8. One more problem with smaller batches is higher training time, since it takes more runs to cover the dataset.

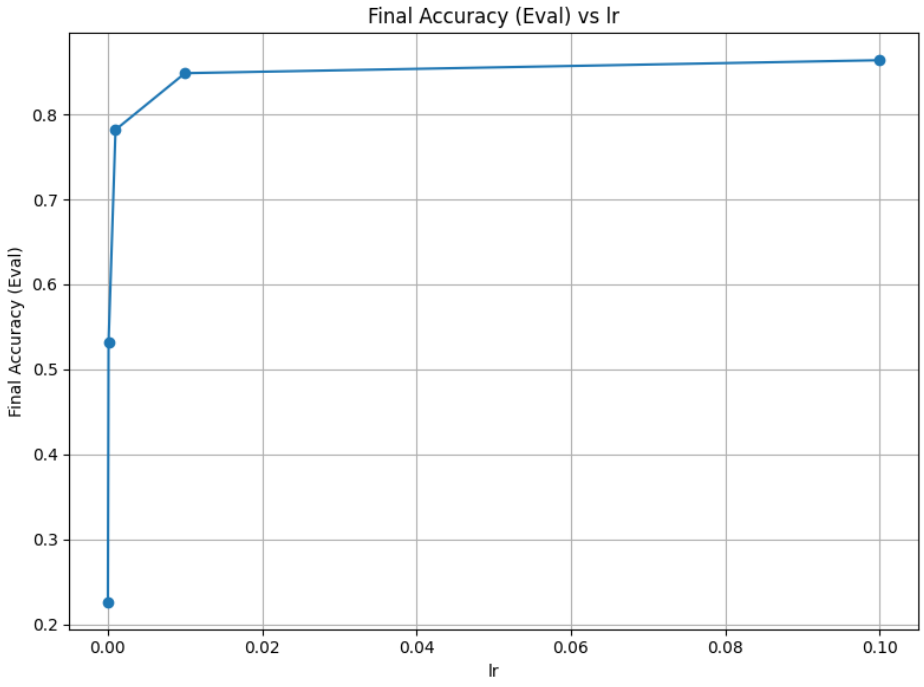
Larger batches, on the other hand, provide much faster training in exchange for lower overall accuracy and instability of this training. Eval accuracy curve clearly depicts this instability.

Another observation can be made by looking at fit curves. They clearly show the difference between different batch sizes, and, considering pros and cons of small and big batches, most optimal one would be “golden middle” – 16. It provides more stable training compared to 32, 64 or 128 while overfitting less than 4 and 8. In fact, fit curves show that’s model’s performance on training data doesn’t vary much for 4,8 and 16 batch sizes, which makes it even better.

5. Learning rate

rounds	epoch	client	batch	lr	alpha	accuracy	loss
32	2	8	16	0.1	1	0.8636993685507469	0.41237206517704533
32	2	8	16	0.01	1	0.8484521792699831	0.4145404621465317
32	2	8	16	0.001	1	0.7818731117824773	0.5930671574365937
32	2	8	16	0.0001	1	0.5319234642497482	1.5257412074435754
32	2	8	16	1e-05	1	0.22639765901740336	2.254350487227141





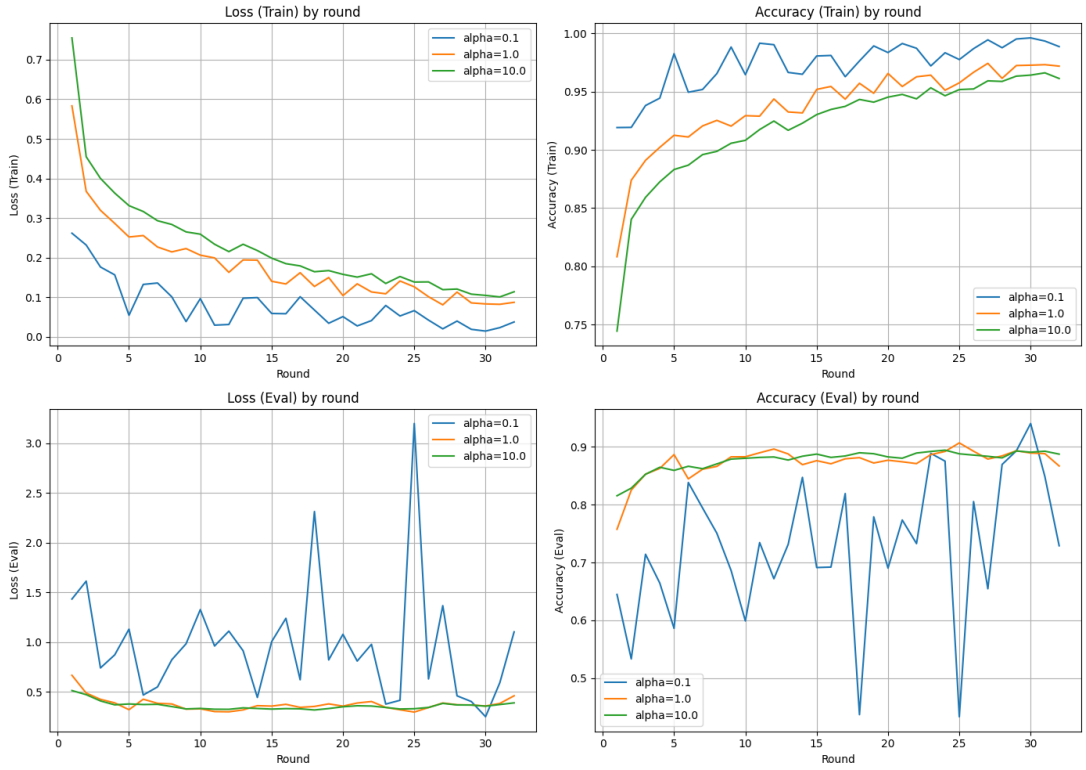
It’s is evident that lowering learning rate severely impacts model’s ability to learn. Low learning rate lowers results in tinier changes to the model, which means that model fails to learn anything, which is most obvious for 0.00001 lr, were all model curves are mostly straight lines, showing no improvements at all.

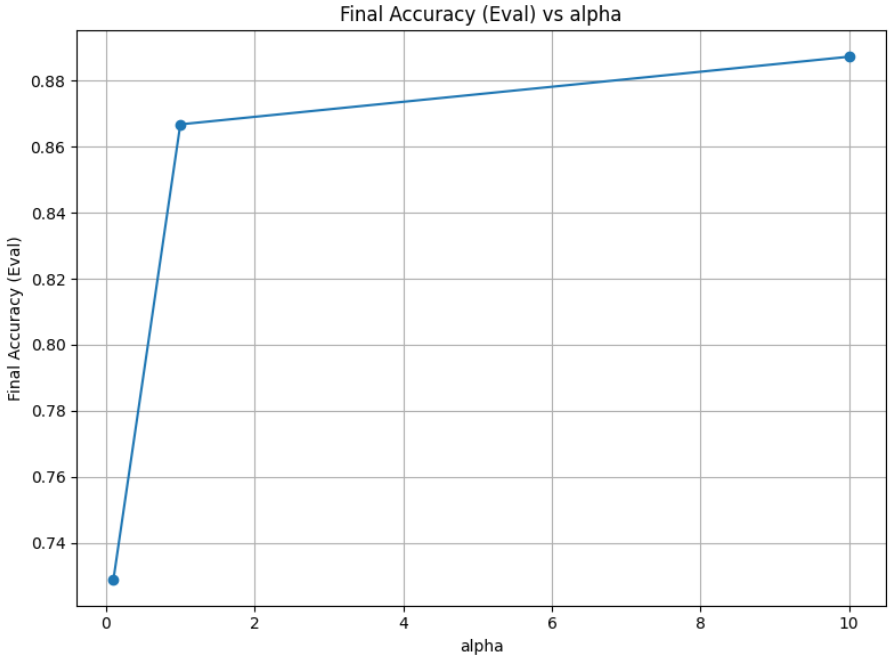
On the other hand we see that learning rate 0.1 yields best accuracy. Even though it’s not intuitive, since high learning rate usually causes overshooting of local minima, in our case this can be result of good initialization of model or wide and shallow minima due to specifics of dataset. One more possible explanation is other hyperparameters might provide better regularization, stabilizing high learning rate.

Since it’s not regular to see such good performance from high lr, it will be safer to use 0.01, especially since it’s results are very close to 0.1, judging by both fit and evaluation curves.

6. Dirichlet distribution parameter(alpha)

rounds	epoch	client	batch	lr	alpha	accuracy	loss
32	8	8	32	0.01	0.1	0.7288973384030418	1.1028718532491546
32	8	8	32	0.01	1.0	0.8667796088094871	0.4599040016249664
32	8	8	32	0.01	10.0	0.8872793872793873	0.3886151044340162





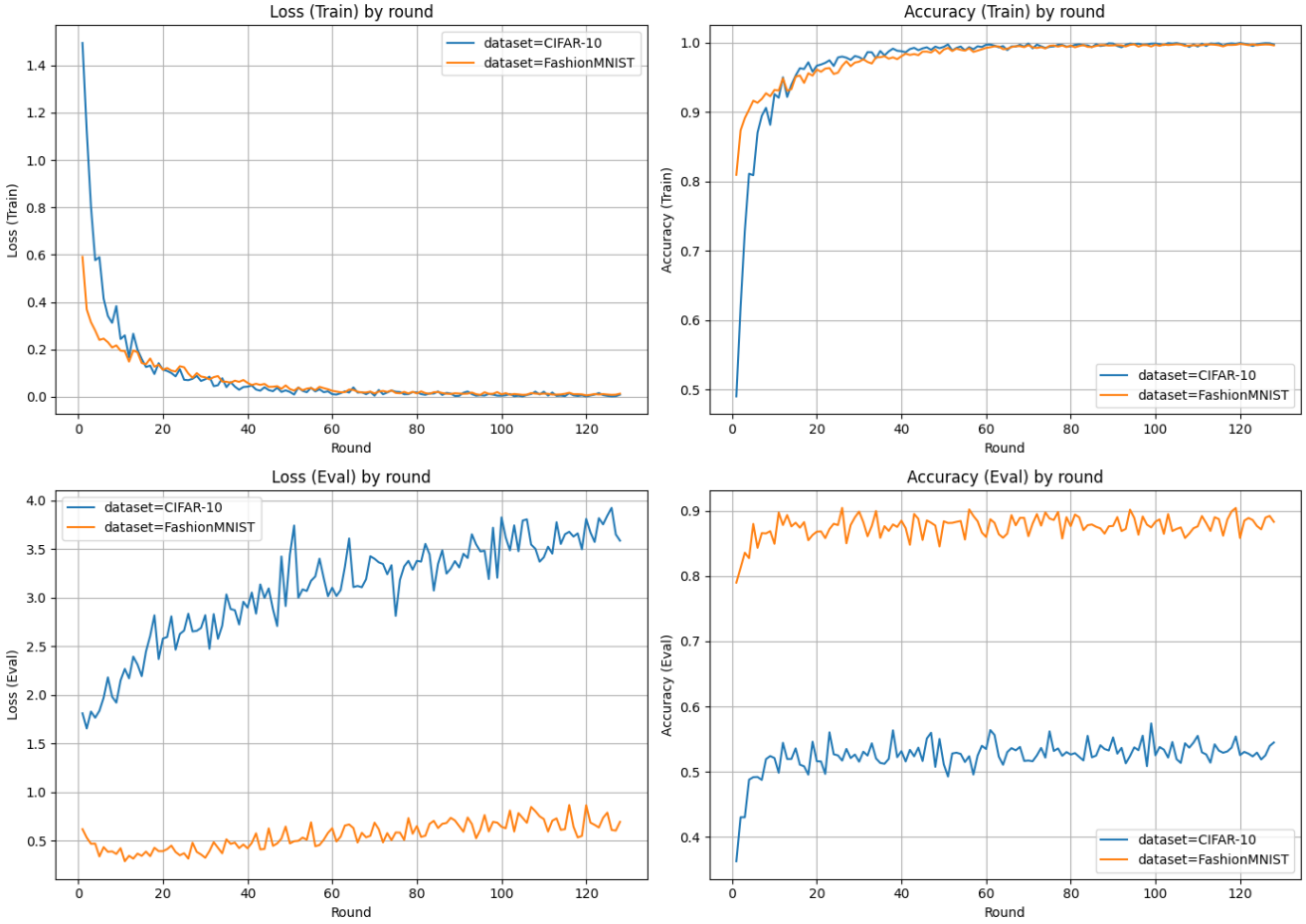
It’s evident that lower alpha results in much worse performance. What’s interesting is that fit curves for alpha=0.1 is relatively stable, compared to evaluation. This can mean that model fail to generalize and overfits on certain data patterns, causing such high volatility on evaluation.

It’s also clear that higher alpha results in much smoother training, which is best visible on fit curves.

Same can be said about final accuracy – highly homogenous data yields much better performance by lowering client drift, while highly heterogeneous data yields causes sever client drift, causing worse results.

7. Dataset and model's architecture

rounds	epoch	client	batch	lr	alpha	dataset	accuracy	loss
128	8	8	32	0.01	1.0	CIFAR-10	0.5449474361724952	3.5875571851846972
128	8	8	32	0.01	1.0	FashionMNIST	0.8829807509052792	0.6961387031210566



It is evident that model struggles to achieve significant results on CIFAR-10. Evaluation accuracy plateaus around 55%, and evaluation loss curve is rising, while training (fit) metrics remain comparable to those of FashionMNIST. This indicates that model, trained on CIFAR-10 fails to generalize and suffers from overfitting. The likely cause is clients' drift, as the model architecture performs well on CIFAR-10 in traditional(centralized) machine learning. While some improvement may be possible through hyperparameter tuning, the primary issue is client drift. Therefore, the most effective solution would be to adopt a more robust aggregation algorithm.