BITS F464 MACHINE LEARNING

ASSIGNMENT-2

A REPORT ON

FEDERATED LEARNING SIMULATION

Submitted To:

Dr. Navneet Goyal

Department of Computer Science, BITS Pilani

Submitted By:

Harsh Agrawal (2021A7PS0524P)

Saarthak Vijayvargiya (2021A3PS1044P)

Vineeta Bugalia (2021B3PS2058P)

TABLE OF CONTENTS

1. Introduction to Federated Learning

2. How Federated Learning Works

3. Types Of Federated Learning

4. Advantages & Challenges of Federated Learning

5. Comparison Between Centralized Machine Learning & Federated Learning

6. Applications Of Federated Learning

7. Problem Statement

8. Code Overview

9. Results

10. References

1. Introduction to Federated Learning

Machine Learning (ML) has revolutionized numerous domains by leveraging vast amounts of data to train predictive models. Traditionally, ML models have relied on centralized learning, where data from different sources is aggregated at a central server for training. However, this approach poses two significant challenges:

Communication Overhead: In the era of Big Data, transferring large datasets to a centralized location is costly and inefficient.

Privacy Concerns: Sensitive data (e.g., in healthcare, finance, or personal devices) risks exposure when shared outside the local environment.

Federated Learning (FL) addresses these challenges by proposing a decentralized machine learning paradigm where data remains on local devices, and only model updates (e.g., gradients or weights) are shared with a central server. Thus, FL significantly reduces communication costs and preserves data privacy.

2. How Federated Learning Works

The typical Federated Learning process involves the following steps:

1. **Client Simulation**: Thousands of clients (e.g., mobile devices) participate in training. Each client possesses its own local dataset.
2. **Client Selection**: Clients are selected based on criteria such as battery level, network type (Wi-Fi or mobile data), and computational power.
3. **Skeleton Model Initialization**: A central server initializes a **global model** (called a skeleton model) and sends it to selected clients.
4. **Local Model Training**: Each client independently updates the model using its local data.
5. **Model Update Communication**: Clients send only the updated model parameters (not the raw data) back to the server.
6. **Global Model Aggregation**: The server aggregates the updates (e.g., using weighted averaging) to refine the global model.
7. **Iterative Rounds**: This process repeats across multiple communication rounds until a satisfactory model performance is achieved.
   This workflow allows building robust models without centralizing the data, thus ensuring data privacy and scalability across millions of devices.

Aggregation Techniques

- The **Federated Averaging (FedAvg)** algorithm is commonly used:
  Each client updates the model on its local data and the server **averages** all client updates to improve the global model.
- More advanced aggregation methods exist, such as:
  - **FedProx**: Handles heterogeneity (i.e., different capabilities of clients).

    - **Secure Aggregation**: Ensures that model updates are encrypted and cannot be viewed even by the server.

Privacy Enhancing Technologies

- **Differential Privacy**:
  Adds controlled noise to model updates to ensure that individual data points cannot be inferred from the updates.
- **Homomorphic Encryption**:
  Allows computations to be performed on encrypted data, providing another layer of privacy.
- **Secure Multiparty Computation (SMPC)**:
  Splits data into pieces among different parties so that no single party can reconstruct the original data.

3. Types Of Federated Learning

- **Horizontal Federated Learning**:
  Different clients have datasets with the **same feature space** (e.g., images of animals) but different users.
  Example: Hospitals in different regions training a model for the same disease using similar features (age, symptoms, test results).
- **Vertical Federated Learning**:
  Different clients have datasets with **different feature spaces** but **overlapping users**.
  Example: A bank and an e-commerce company both have data about the same customers but with different attributes (transactions vs. browsing history).
- **Federated Transfer Learning**:
  When clients have **different feature spaces and different user sets**, transfer learning techniques are used to adapt and share knowledge.

4. Advantages of Federated Learning

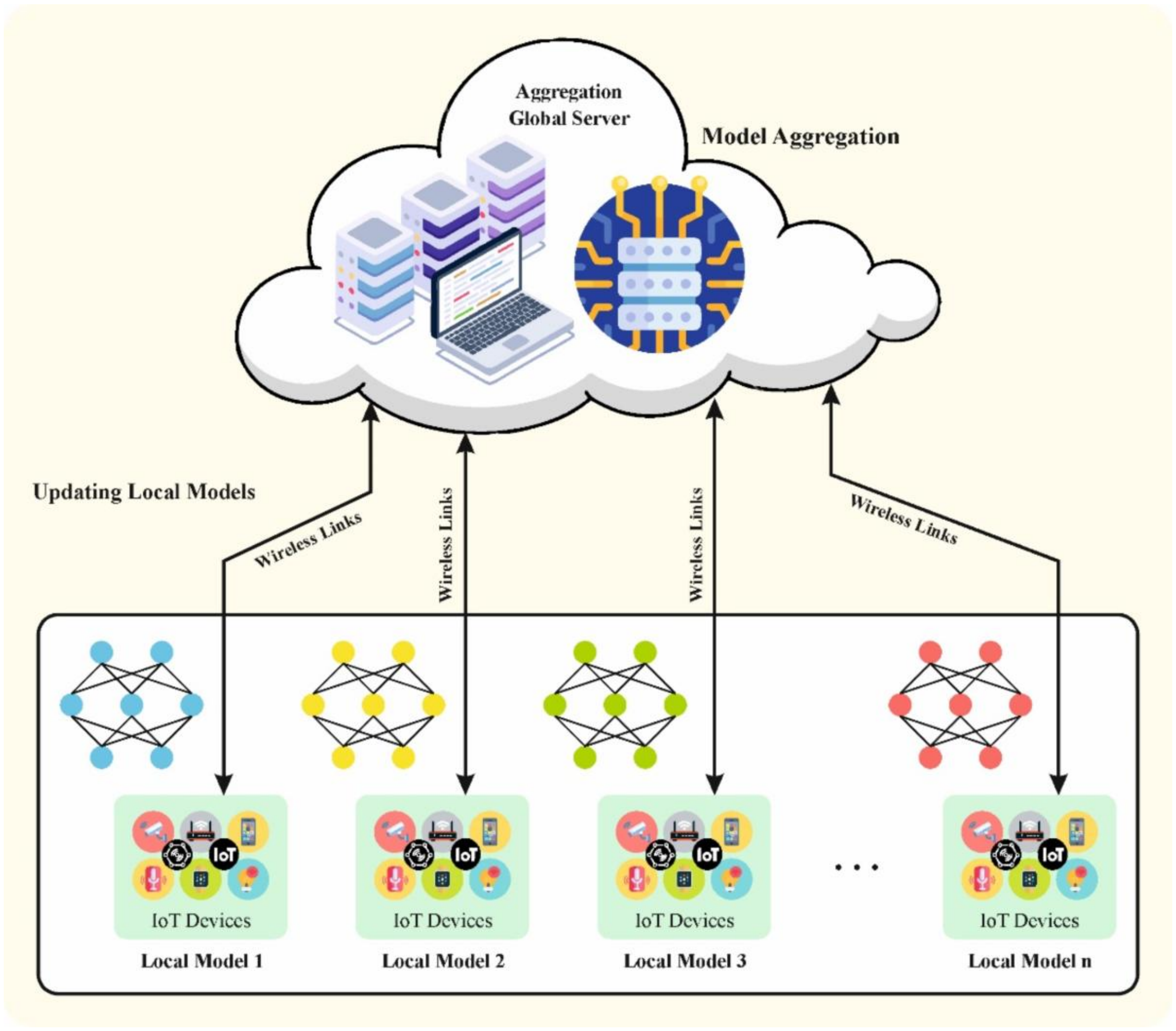
- **Data Privacy**: Raw data never leaves the local device.
- **Reduced Communication Costs**: Only model updates are transmitted, reducing data transfer volume.
- **Personalized Models**: Models can adapt better to local data distributions.
- **Compliance with Regulations**: Useful for environments with strict data privacy laws (e.g., GDPR, HIPAA).

Challenges in Federated Learning

- **Non-IID Data**: Client data may be non-identically distributed (different across users), making model convergence harder.
- **System Heterogeneity**: Devices differ in computation, storage, and network connectivity.
- **Communication Efficiency**: Frequent updates can still cause overhead if not optimized.
- **Security and Trust**: Protecting against malicious updates (e.g., poisoned models) is critical.

5. Comparison Between Centralized Machine Learning & Federated Learning

Aspect	**Centralized Machine Learning**	**Federated Learning**
Data Storage	Data is collected and stored on a central server.	Data stays locally on client devices.
Privacy	High risk of data breaches and privacy violations.	Data privacy is preserved; only model updates are shared.
Communication Overhead	High, since entire datasets need to be transferred.	Low, as only small model updates are communicated.
Scalability	Limited by server storage and processing capabilities.	Highly scalable across millions of devices.
Handling Non-IID Data	Easier, as data is centralized and can be preprocessed.	Challenging, as local data varies across clients.
Infrastructure Cost	High for managing large-scale centralized servers.	Distributed infrastructure reduces central load.
Vulnerability to Attacks	Central server is a single point of failure.	Requires defence against client-side attacks (e.g., poisoning).



Deep Federated Learning Model

6. Applications of Federated Learning

- **Healthcare**:
  Hospitals collaborate to build predictive models without sharing sensitive patient data.
- **Finance**:
  Banks can jointly detect fraud patterns while keeping customer information private.
- **Smartphones**:
  Companies like Google use FL to improve keyboard prediction and speech recognition models without uploading private user text.
- **IoT and Edge Devices**:
  Autonomous vehicles, smart homes, and wearables benefit from local learning and global knowledge sharing.

7. Problem Statement

This assignment aims to simulate a Federated Learning setup for an animal image classification problem and compare its performance against the traditional centralized machine learning approach. Specifically, the objectives are:

- To build and train a classification model using both centralized and federated approaches.
- To simulate a thousand mobile clients with heterogeneous conditions, including battery levels, communication types, and computational capabilities.
- To evaluate and compare both models in terms of classification accuracy and communication efficiency.
- To demonstrate the practical advantages and challenges of Federated Learning over Centralized Machine Learning.

About Dataset:

The **AFHQ (Animal Faces-HQ)** dataset, available on Hugging Face at huggingface.co/datasets/huggan/AFHQ, is a high-quality image dataset designed for various computer vision tasks, including image classification and generative modeling.

Key Features:

- **Classes**: The dataset comprises three categories: **cat**, **dog**, and **wild**. It has approximately equal no. of images for all categories.
- **Image Count**: 16,130
- **Image Resolution**: 512×512 pixels
- **Split**: No splitting comes as a single dataframe.

8. Code Overview

Let's us see how to run the code [Details given in README.md]:

1. First, we need to install dependencies for the project

   If using Conda, you may create an environment first

   ```
   conda create --name FederatedLearning python=3.11 -y
   conda activate FederatedLearning
   pip install -e .
   ```

   OR run directly inside the terminal

   ```
   pip install -e .
   ```

2. Run the simulation:
   [Make Sure your path is "<Path>\fl-simulation" where pyproject.toml is stored]
   In the `fl-simulation` directory, use `flwr run .` to run a local simulation:

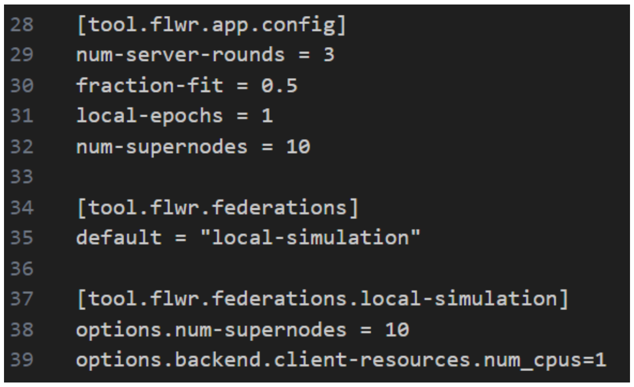
   ```
   flwr run .
   ```

In this project, we have used a library called as flower which enables us for automated simulation of the clients and server. It also helps to create a naïve project which can be tweaked further as our needs.

The project consists of a configuration file called as **pyproject.toml** where we can define different parameters regarding the server and the clients.

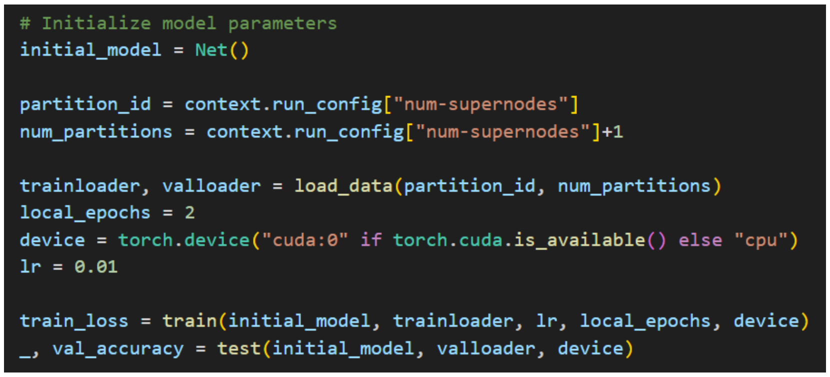
Parameters:

1. **num-server-rounds:** Number of rounds server should perform of federated learning.
2. **Local-epochs:** Number of epochs ran by each client for training.
3. **Num-supernodes:** Number of clients.
4. **Options.backend.client-resources.num-cpus**: Num of cpus allotted to each client.



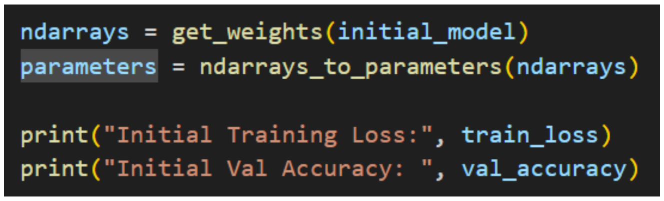
```
[tool.flwr.app.config]
num-server-rounds = 3
fraction-fit = 0.5
local-epochs = 1
num-supernodes = 10

[tool.flwr.federations]
default = "local-simulation"

[tool.flwr.federations.local-simulation]
options.num-supernodes = 10
options.backend.client-resources.num_cpus=1
```

The file **server-app.py** initializes the server model, which is trained for 2 epochs and its loss and validation accuracy is displayed. The data used for this is different from the clients.

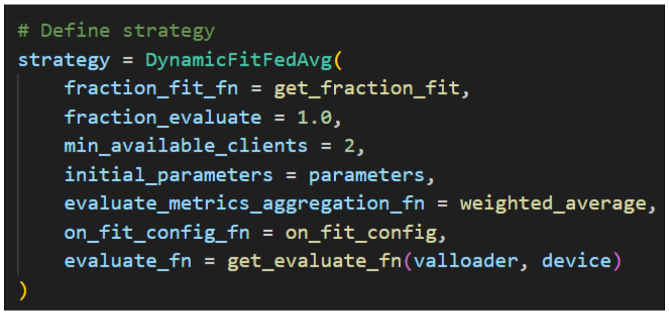


```python
# Initialize model parameters
initial_model = Net()

partition_id = context.run_config["num-supernodes"]
num_partitions = context.run_config["num-supernodes"]+1

trainloader, valloader = load_data(partition_id, num_partitions)
local_epochs = 2
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
lr = 0.01

train_loss = train(initial_model, trainloader, lr, local_epochs, device)
_, val_accuracy = test(initial_model, valloader, device)
```

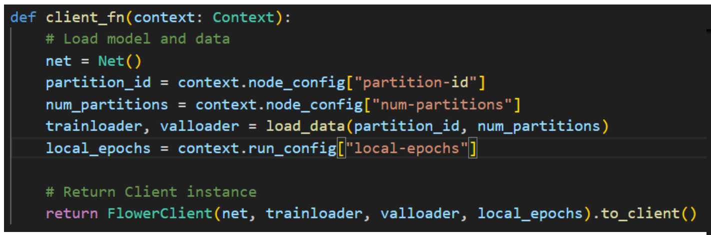
Now the trained model parameters are stored in “parameters” and sent to each client.



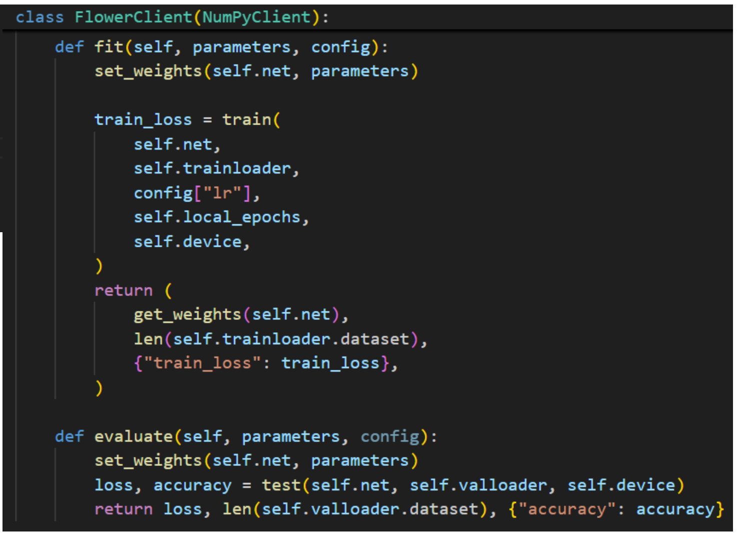
```python
ndarrays = get_weights(initial_model)
parameters = ndarrays_to_parameters(ndarrays)

print("Initial Training Loss:", train_loss)
print("Initial Val Accuracy: ", val_accuracy)
```



```python
# Define strategy
strategy = DynamicFitFedAvg(
    fraction_fit_fn = get_fraction_fit,
    fraction_evaluate = 1.0,
    min_available_clients = 2,
    initial_parameters = parameters,
    evaluate_metrics_aggregation_fn = weighted_average,
    on_fit_config_fn = on_fit_config,
    evaluate_fn = get_evaluate_fn(valloader, device)
)
```



```python
def client_fn(context: Context):
    # Load model and data
    net = Net()
    partition_id = context.node_config["partition-id"]
    num_partitions = context.node_config["num-partitions"]
    trainloader, valloader = load_data(partition_id, num_partitions)
    local_epochs = context.run_config["local-epochs"]

    # Return Client instance
    return FlowerClient(net, trainloader, valloader, local_epochs).to_client()
```

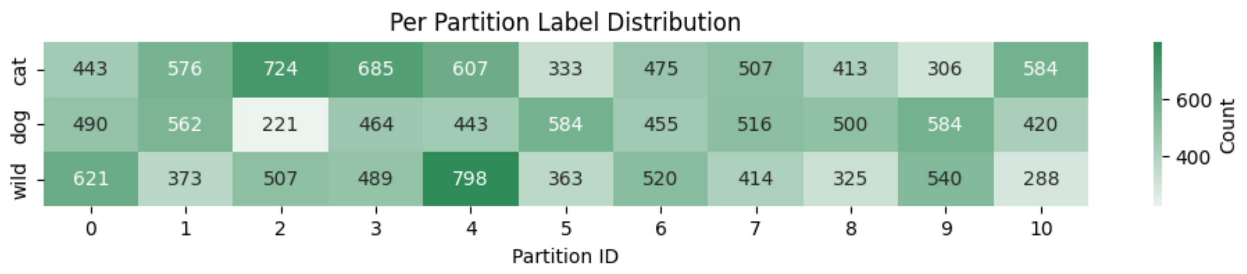


```python
class FlowerClient(NumPyClient):
    def fit(self, parameters, config):
        set_weights(self.net, parameters)

        train_loss = train(
            self.net,
            self.trainloader,
            config["lr"],
            self.local_epochs,
            self.device,
        )
        return (
            get_weights(self.net),
            len(self.trainloader.dataset),
            {"train_loss": train_loss},
        )

    def evaluate(self, parameters, config):
        set_weights(self.net, parameters)
        loss, accuracy = test(self.net, self.valloader, self.device)
        return loss, len(self.valloader.dataset), {"accuracy": accuracy}
```

Now in **client-app.py**, each client will get its own data and load the parameters from the server and get their respective models trained.

task.py contains all the code for train(), test(), set_weights() and the Net() architecture.

load_data() function in **task.py** partitions the data which partions the data in unequal amount by using from DirichletPartitioner.

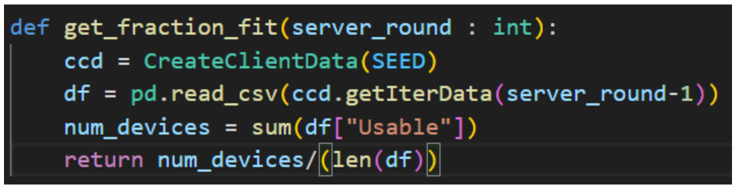


	0	1	2	3	4	5	6	7	8	9	10
cat	443	576	724	685	607	333	475	507	413	306	584
dog	490	562	221	464	443	584	455	516	500	584	420
wild	621	373	507	489	798	363	520	414	325	540	288

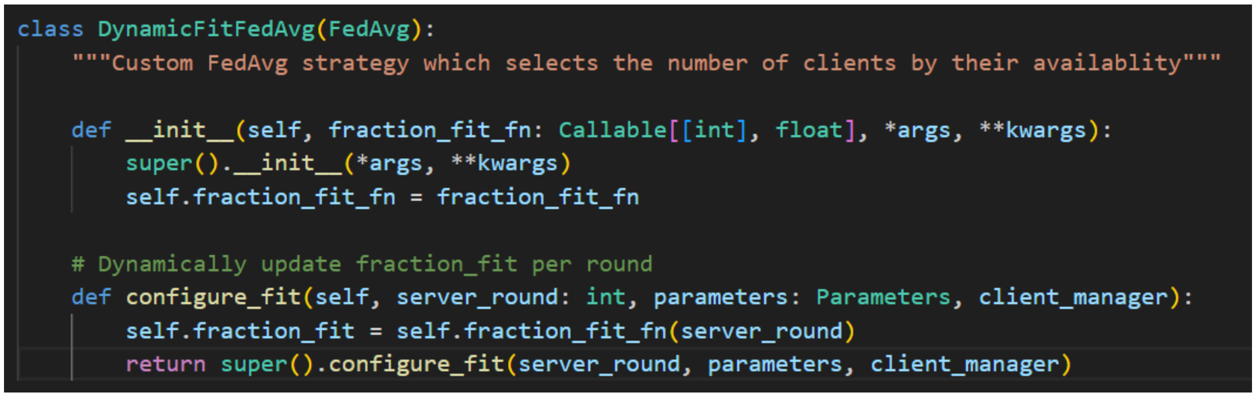
Dirichlet partitioner with 10 clients and a server. Total 11 partitions.

We are also resizing each image to 128 X 128 pixels for efficiency and time saving purposes.

To change the number of clients sampled at each round, we created a class inside **client-data.py** which will get the number of clients from the configuration file and make a data frame which will have the data of the clients for feasibility of battery levels and computing power according to which, the client is selected. This will create a folder named as **iterations** which has a file for each iteration. Now in **server.py,** a function get_fraction_fit() is run for each round and this will change the number of clients sampled in each round by help of a strategy class [`DynamicFitFedAvg`] defined inside **custom-strategy.py.**



```python
def get_fraction_fit(server_round : int):
    ccd = CreateClientData(SEED)
    df = pd.read_csv(ccd.getIterData(server_round-1))
    num_devices = sum(df["Usable"])
    return num_devices/(len(df))
```



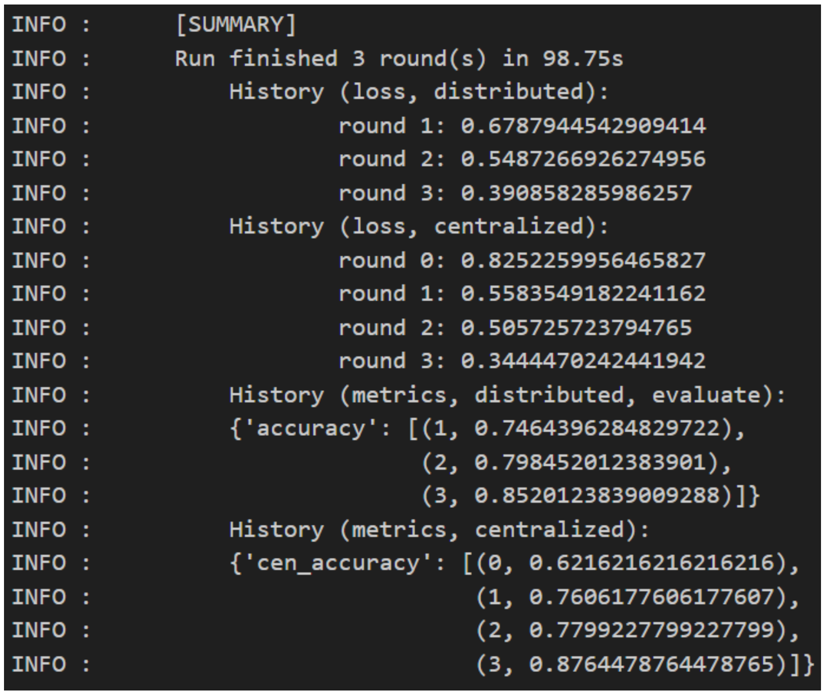
```python
class DynamicFitFedAvg(FedAvg):
    """Custom FedAvg strategy which selects the number of clients by their availablity"""

    def __init__(self, fraction_fit_fn: Callable[[int], float], *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.fraction_fit_fn = fraction_fit_fn

    # Dynamically update fraction_fit per round
    def configure_fit(self, server_round: int, parameters: Parameters, client_manager):
        self.fraction_fit = self.fraction_fit_fn(server_round)
        return super().configure_fit(server_round, parameters, client_manager)
```

`FedAvg` is a class which automatically performs federated averaging, which is provided by the flower library.

For Centralized Machine Learning, you may run **cml-task.py** file inside the cml-simulation directory by using python3 command. This simulates the model trained in a centralized way, where all the data is on the server and model is trained on the server.

9. Results

Flower library also helps in proper formatting of the logs. The summary of the simulation displayed during the federated learning involving 10 clients is given:



```
INFO :      [SUMMARY]
INFO :      Run finished 3 round(s) in 98.75s
INFO :          History (loss, distributed):
INFO :                  round 1: 0.6787944542909414
INFO :                  round 2: 0.5487266926274956
INFO :                  round 3: 0.390858285986257
INFO :          History (loss, centralized):
INFO :                  round 0: 0.8252259956465827
INFO :                  round 1: 0.5583549182241162
INFO :                  round 2: 0.505725723794765
INFO :                  round 3: 0.3444470242441942
INFO :          History (metrics, distributed, evaluate):
INFO :          {'accuracy': [(1, 0.7464396284829722),
INFO :                        (2, 0.798452012383901),
INFO :                        (3, 0.852012383909288)]}
INFO :          History (metrics, centralized):
INFO :          {'cen_accuracy': [(0, 0.6216216216216216),
INFO :                            (1, 0.7606177606177607),
INFO :                            (2, 0.779922779922779),
INFO :                            (3, 0.8764478764478765)]}
```

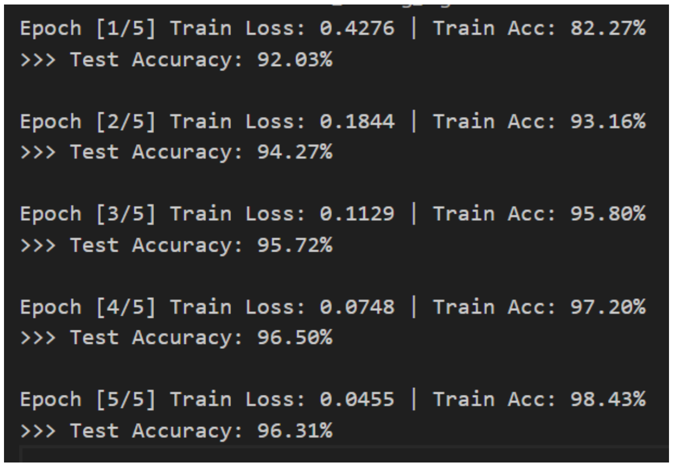
History (loss, distributed)
>> Shows federated average loss among all clients.

History (loss, centralized)
>> Shows loss calculated on centralized validation dataset

'accuracy'
>> Shows federated average accuracy on validation dataset.

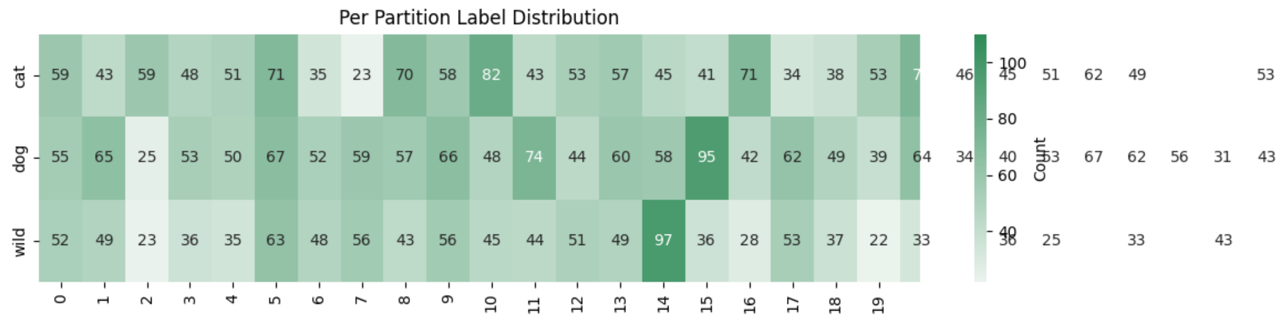
History (loss, distributed)
>> Shows accuracy on central validation dataset. The numbers 0-3 indicate the accuracy of the model after initial phase and 3 rounds.

We were able to achieve around 87% of accuracy on the validation dataset in case of federated learning. Likewise, when we trained the same model in centralized way where 80% of dataset was used for training and 20% was used for validation, we were able to achieve 96% of accuracy. This shows that Federated learning offers us a trade-off between accuracy and communication costs.



```
Epoch [1/5] Train Loss: 0.4276 | Train Acc: 82.27%
>>> Test Accuracy: 92.03%

Epoch [2/5] Train Loss: 0.1844 | Train Acc: 93.16%
>>> Test Accuracy: 94.27%

Epoch [3/5] Train Loss: 0.1129 | Train Acc: 95.80%
>>> Test Accuracy: 95.72%

Epoch [4/5] Train Loss: 0.0748 | Train Acc: 97.20%
>>> Test Accuracy: 96.50%

Epoch [5/5] Train Loss: 0.0455 | Train Acc: 98.43%
>>> Test Accuracy: 96.31%
```

Please note that, we were able to use only 10 clients as our dataset had only around 16000 images. So, if we divide the dataset into 11 parts (10 clients and a server), then we are able to get 1454 images in each part which gets divided further into 3 categories.
If we simulate 1000 clients, then each partition will have only around 16 images and this will deteriorate the accuracy to less than 20 %.
A sample of number of images in each partition, if number of clients is 100:



Even 100 clients led to an accuracy of about 50%.

The main purpose of using a library such as flower for simulation of federated learning is its ability to handle a large number of clients automatically. Otherwise, each client has to be manually connected to the server by running ssh commands.

10. References

1. Dataset: https://huggingface.co/datasets/huggan/AFHQ
2. https://www.researchgate.net/figure/Architecture-diagram-of-Deep-Federated-Learning_fig1_376767476
3. https://research.ibm.com/blog/what-is-federated-learning
4. https://www.v7labs.com/blog/federated-learning-guide
5. https://medium.com/@jamsherbhanbhro/federated-learning-vs-classical-machine-learning-6b8592eb02b4