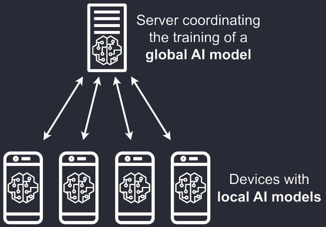
# Federated learning with Docker

Team 01



# Contents

## Federated learning

# What is Federated Learning?

## Introduction

- Federated learning:
  - approach to machine learning
  - allows models to be trained in a *distributed way* across multiple devices or local servers
  - *no need to gather* and transfer the raw data to a central server

# What is Federated Learning?

## Introduction

- Federated learning:
  - approach to machine learning
  - allows models to be trained in a *distributed way* across multiple devices or local servers
  - *no need to gather* and transfer the raw data to a central server
- How does it work?
  1. Models are *sent to local devices* (clients)
  2. They are trained on local data
  3. Model updates are *aggregated* to form a global model

# What is Federated Learning?

## Introduction

- Federated learning:
  - approach to machine learning
  - allows models to be trained in a *distributed way* across multiple devices or local servers
  - *no need to gather* and transfer the raw data to a central server
- How does it work?
  1. Models are *sent to local devices* (clients)
  2. They are trained on local data
  3. Model updates are *aggregated* to form a global model
- Benefits:
  - protection of data *confidentiality*
  - reduced bandwidth requirements
  - ability to process geographically distributed data

# What is Federated Learning?

## Centralized learning

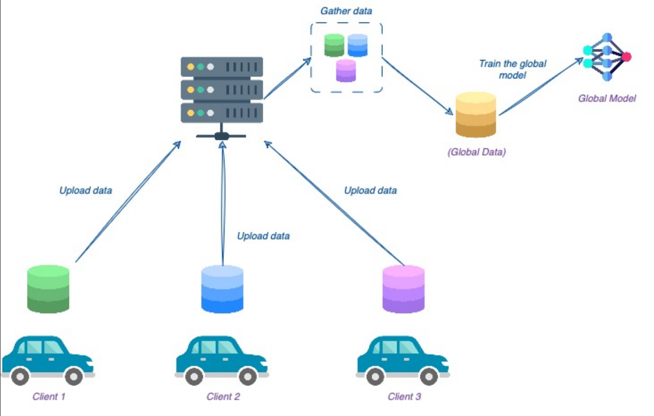



Figure: Centralized learning

# What is Federated Learning?

## Federated learning

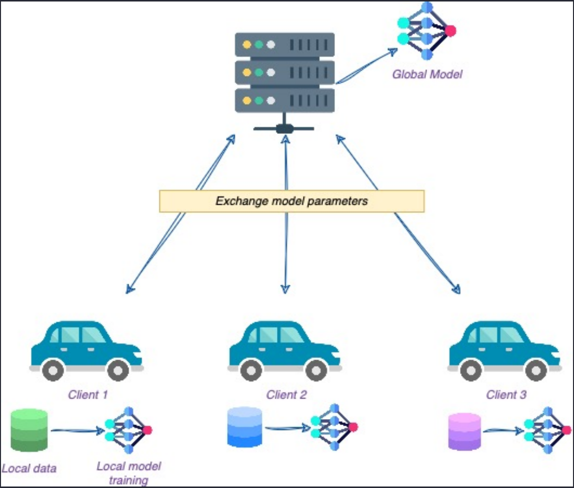



Figure: Federated learning

# Project Setup

## Working environment

1. Clone the repository

```
git clone git@github.com:CCBDA-UPC/Research-Projects-2024.git
```

2. change directory to tutorial-1

```
cd tutorial-1/
```

3. Set up the working environment

```
python -m venv .venv
source .venv/bin/activate

pip install -r requirements.txt
```

# Project Setup

## Flower server

```
from flwr.common import ( EvaluateIns, EvaluateRes, FitIns, FitRes, Parameters, Scalar )
from flwr.server.client_manager import ClientManager
from flwr.server.client_proxy import ClientProxy
from flwr.server.strategy import Strategy


class FedAnalytics(Strategy):
    def initialize_parameters(self, client_manager: Optional[ClientManager] = None) -> Optional[Parameters]:
        ...

    def configure_fit(self, server_round: int, parameters: Parameters, client_manager: ClientManager) -> List[Tuple[ClientProxy, FitIns]]:
        ...

    def aggregate_fit( self, server_round: int, results: List[Tuple[ClientProxy, FitRes]], failures: List[Union[Tuple[ClientProxy, FitRes], BaseException]]) -> Tuple[Optional[Parameters], Dict[str, Scalar]]:
        ...
```

# Project Setup

## Flower server

```
    def evaluate(self, server_round: int, parameters: Parameters) -> Optional[Tuple[float, Dict[str, Scalar]]]:
        ...

    def configure_evaluate(self, server_round: int, parameters: Parameters, client_manager: ClientManager) -> List[Tuple[ClientProxy, EvaluateIns]]:
        ...

    def aggregate_evaluate(self,server_round: int,results: List[Tuple[ClientProxy, EvaluateRes]],failures: List[Union[Tuple[ClientProxy, EvaluateRes], BaseException]],) -> Tuple[Optional[float], Dict[str, Scalar]]:
        ...
```

# Project Setup

## Flower server

```

# Start Flower server
fl.server.start_server(
    server_address="0.0.0.0:8080",
    config=fl.server.ServerConfig(num_rounds=num_rounds),
    strategy=FedAnalytics(),
)
```

# Project Setup
## Flower client

```python
from flwr_datasets import FederatedDataset
# Define Flower client
class FlowerClient(fl.client.NumPyClient):
    def get_parameters(self, config):
        # return the model weight as a list
        ...
    def set_parameters(self, parameters):
        # update the local model weights with the parameters received from the server
        ...
    def fit(self, parameters, config):
        # set the local model weights, train the model.
        ...
    def evaluate(self, parameters, config):
        # test the local model
        ...
```

# Project Setup

## Flower client

```python
    args = parser.parse_args()
    partition_id = args.partition_id
    server_address = args.server_address

    # Load the partition data
    fds = FederatedDataset(dataset="hitorilabs/iris", partitioners={"train": N_CLIENTS})

    dataset = fds.load_partition(partition_id, "train").with_format("pandas")[:]
    # Use just the specified columns
    X = dataset[column_names]

    # Start Flower client
    fl.client.start_client(
        server_address=server_address,
        client=FlowerClient(X).to_client(),
```

*Train the model* (locally), federated

1. Run the server

```
python server.py
```

2. Run the first client

```
python client.py --partition-id 0 --server-address 127.0.0.1:8080
```

3. Run the second client

```
python client.py --partition-id 1 --server-address 127.0.0.1:8080
```

# Static system

## docker-compose.yml

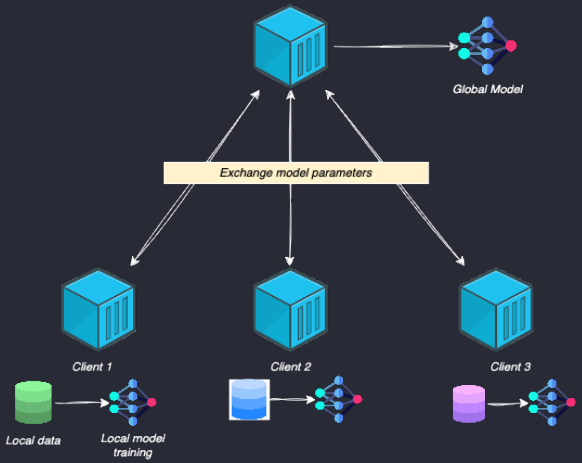



Figure: Federated learning implementation with Docker containers

# Static system

## docker-compose.yml

docker

- *Portability:* consistent performance across various platforms
- *Isolation and Security:* By isolating dependencies: minimizing conflicts and potential risks
- *Ease of Deployment* and Management: simplifies the deployment and management of federated learning clusters
- *Reproducibility:* experiments can be easily shared and replicated

# Static system

## Flower server

- The server service is built from the server/ directory *(uses the script mentioned in the last part)*
- The server service is configured to wait for *3 rounds* before generating the global model.

```
server:
  build: server/.
  container_name: server
  environment:
    - NUM_ROUNDS=3
  networks:
    - federated_learning
```

▸ Dockerfile

# Static system

## Flower client

- The client service is built from the client/ directory.
- The client service is configured *with 2 clients*, each with a unique partition ID and server address.

```
client-0:
  build: client/.
  restart: on-failure
  environment:
    - SERVER_ADDRESS=server:8080
    - PARTITION_ID=0
    - NUMBER_OF_CLIENTS=2
  networks:
    - federated_learning
```

Dockerfile

# Static system

## Run the project

- Deploy the *docker-compose* stack {server, client-0, client-1}

```
docker compose up --build
```

- Check the *logs* to get the results of the process

  *It is possible to mount a volume and extract the stdout in a file*

```
docker compose logs -f
```

```
federated_learning_pandas
INFO :
INFO :      [ROUND 2]
INFO :      configure_fit: strategy sampled 2 clients (out of 2)
INFO :      aggregate_fit: received 2 results and 0 failures
INFO :      fit progress: (2, 0, {'Aggregated histograms': ['Length:', '4', '3', '2', '4', '2', '5', '3',
 '2', '2', '3', 'Width:', '2', '4', '2', '4', '3', '5', '2', '3', '0', '5']}, 0.4847832919913344)
INFO :      configure_evaluate: no clients selected, skipping evaluation
INFO :
INFO :      [ROUND 3]
INFO :      configure_fit: strategy sampled 2 clients (out of 2)
INFO :      aggregate_fit: received 2 results and 0 failures
INFO :      fit progress: (3, 0, {'Aggregated histograms': ['Length:', '4', '3', '2', '4', '2', '5', '3',
 '2', '2', '3', 'Width:', '2', '4', '2', '4', '3', '5', '2', '3', '0', '5']}, 0.4872107499977574)
INFO :      configure_evaluate: no clients selected, skipping evaluation
INFO :
INFO :      [SUMMARY]
INFO :      Run finished 3 rounds in 0.49s
INFO :      History (loss, centralized):
INFO :          '\tround 0: 0\n\tround 1: 0\n\tround 2: 0\n\tround 3: 0\n'History (metrics, centralized):
INFO :          {'Aggregated histograms': [(0, []),
```

# Dynamic system

## Running the Server

Content of *run_server.sh*

1. Create a Docker Network

```
docker network create federated_learning
```

2. Build the Server Docker Image

```
docker build -t federated_learning_server:
    latest server/.
```

3. Run the Server Docker Container

```
docker run --name server \
    --env NUM_ROUNDS=8 \
    --network federated_learning \
    federated_learning_server
echo "Server started."
```

In the same way, create a Bash Script *run_random_clients.sh*

1. Build the image
2. Declare a `start_client()` function to *create one client on demand*
3. Declare `stop_client()` to stop a client
4. Main function `create_random_clients()` to *start and stop randomly clients*

```
./run_random_clients.sh
```

Bash script

# Appendix

## Server Dockerfile

```
FROM python:3.12-slim
WORKDIR /app
# Copy the requirements file
COPY ../requirements.txt requirements.txt
# Install the dependencies
RUN pip install -r requirements.txt --no-cache-dir
# Copy the rest of the code
COPY server.py /app
# Run the server
CMD ["python", "server.py"]
```

# Appendix

## Client Dockerfile

```
FROM python:3.12-slim
WORKDIR /app
# Copy the requirements file
COPY requirements.txt requirements.txt
# Install the dependencies
RUN pip install -r requirements.txt --no-cache-dir
# Copy the rest of the code
COPY client.py /app

# Add the argument
ENV PARTITION_ID=0
ENV SERVER_ADDRESS="server:8080"

# Run the server
CMD python client.py --partition-id $PARTITION_ID --
    server-address $SERVER_ADDRESS
```

# Appendix

## Run clients - 1

```bash
#!/bin/bash
docker build -t federated_learning_client:latest client/.

start_client() {
    client_id=$1
    partition_id=$2
    client_name="client-${client_id}"
    docker run -d --name "$client_name" \
        --env SERVER_ADDRESS=server:8080 \
        --env PARTITION_ID="$partition_id" \
        --env NUMBER_OF_CLIENTS=2 \
        --network federated_learning \
        --restart on-failure \
        federated_learning_client
    echo "Started $client_name"
}
```

# Appendix

## Run clients - 2

```
stop_client() {
    client_id=$1
    client_name="client-${client_id}"
    docker stop "$client_name" > /dev/null
    docker rm "$client_name" > /dev/null
    echo "Stopped $client_name"
}

create_random_clients() {
    for ((i=0; i<10; i++)); do
        client_id=$((RANDOM % 1000))
        start_client "$client_id" "$i"
        sleep 1
    done
}

create_random_clients
```