Lab 3: Out-Of-Distribution Detection in Federated Learning using principles of Hyperdimensional Computing

February 11, 2026

Course: Secure AI
Duration: 1 Week (Groups of 2-4 students)
Objective: Design and implement a local simulation framework for Federated Learning, along with an Out-of-Distribution detection mechanism to safeguard the global model by identifying and filtering compromised or adversarial local model updates before aggregation.

Contents

List of Tables

List of Figures

Abbreviations

CNN Convolutional Neural Network. 1, 2, 6, 12–15

DL Deep Learning. 1, 14

FL Federated Learning. 1, 2, 4–8, 11, 15–18, 20

HDC Hyperdimensional Computing. 1, 8, 10, 16

HDFF Hyperdimensional Feature Fusion. 16

ID In-Distribution. 2, 5, 9–11, 14, 15, 20–23

OOD Out-of-Distribution. 1, 2, 4, 5, 7–11, 16–23

TF TensorFlow. 2

1. INTRODUCTION

Deep Learning (DL) models have become a cornerstone of modern technology, powering applications across various domains, from speech recognition and natural language processing to predictive analytics and autonomous systems.

In recent years, the demand for privacy-preserving machine learning solutions has grown exponentially, driven by increasing concerns over data security, regulatory requirements, and ethical considerations. Federated Learning (FL) has emerged as a promising approach to address these challenges by enabling collaborative model training across distributed clients without requiring raw data to leave local devices or institutions. Despite its advantages, FL presents unique challenges, particularly in managing data heterogeneity and ensuring the robustness of models when exposed to unexpected or anomalous inputs [1, 2, 3].

One critical issue in FL is handling Out-of-Distribution (OOD) data, data that deviate from the training distribution of participating clients, that can be either malicious or contain imperfections [1, 2]. Malicious data that deviates from the training objective can compromise model performance, leading to unreliable predictions and potential risks in high-stakes applications [3, 4, 5]. This challenge is further compounded by the decentralized nature of FL, where detecting and mitigating the impact of OOD data must be achieved without violating privacy constraints or incurring significant computational costs [1, 2].

Current research proposes different methods for extracting information and implementing FL defense strategies [6]. However, due to the drawbacks of sharing model parameters and information, there are several approaches to defense mechanisms that circumvent these limitations, although they involve certain trade-offs [1, 2, 6, 7, 8, 9].

2. LAB OBJECTIVE

You are tasked with implementing and simulating a FL environment to evaluate how various malicious data scenarios may compromise the integrity and performance of the global model. In addition, you will implement a method to detect and mitigate malicious updates made from local models by leveraging Hyperdimensional Computing (HDC) to compare incoming model updates against the internal state of the global model.

Each participant (a local model engaged in collaborative training within a FL environment) is assigned a Convolutional Neural Network (CNN) model that is trained independently on its respective subset of image data.

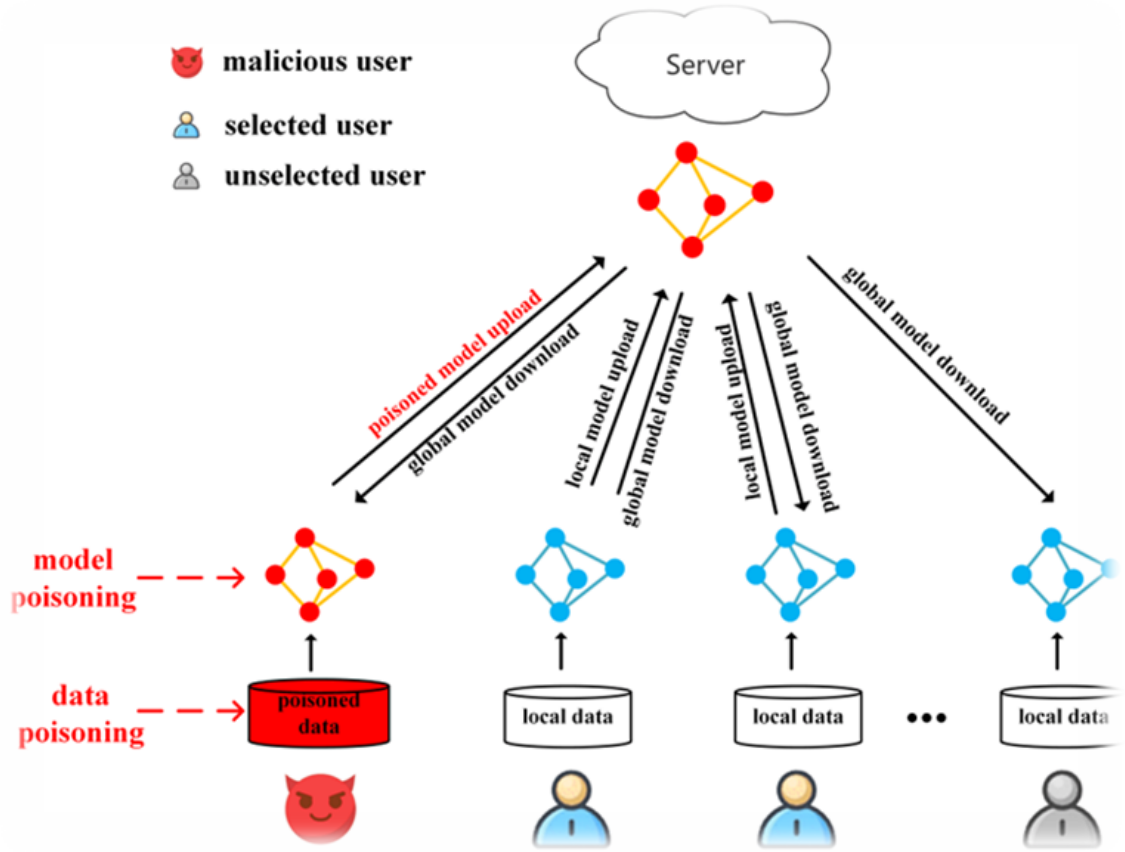

Figure 1: FL Environment

2.1 Resources

You are provided with code as part of the assignment. The supplied code includes a dataset generator for pre-processing and downloading datasets and an implementation of a CNN model. However, you are free to modify the provided code as you see fit. The key requirement is the outcome achieved by adhering to the specifications outlined in Section 2.2, Design & Requirements.

2.1.1 CNN Model & Dependencies

You are provided with a CNN model implemented using TensorFlow (TF). The accompanying folder includes a Makefile, designed for Ubuntu, which assists in creating a Python virtual environment with all required modules installed.

The Makefile is designed for Ubuntu and specifically targets the apt package manager. If you are running the code on Windows or another Linux/Unix distribution not using apt, you can execute the corresponding commands manually and adjust them according to your operating system. All required Python modules are listed in python_requirements.txt.

2.1.2 Datasets

You are free to select any datasets you deem appropriate; however, there must be clear distinctions between them. The provided code includes a dataset generator that downloads four different datasets. Which is sufficient for this laboratory exercise.

The first dataset used is an Alzheimer's dataset [10], featuring x-ray images of patients' brains. These images are analyzed to determine whether any signs of Alzheimer's disease are present.

Figure 2: Dataset: Alzheimer dataset, displaying a few samples.

Secondly, two distinct brain tumor datasets [11, 12] is utilized, comprising X-ray images that depict malignant tumor growths in patients' brains.

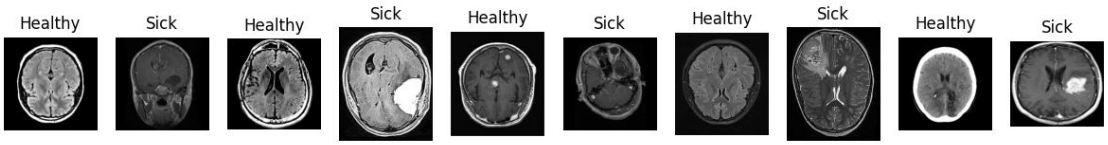

Figure 3: Dataset: Brain tumor dataset 1.

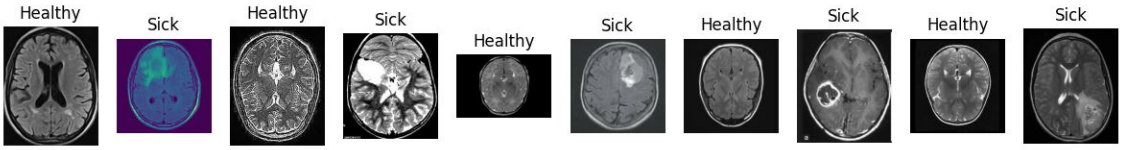

Figure 4: Dataset: Brain tumor dataset 2.

Thirdly, a pneumonia dataset [13] containing X-ray images of patients' lungs is used to identify signs of pneumonia.

Figure 5: Dataset: Pneumonia dataset.

All datasets combined contain approximately 18000 images/features in total. Each category—Alzheimer, brain tumor, and pneumonia—contains approximately 5100, 7600, and 5200 features, respectively.

As part of the attached code, a dataset generator was created to involve diverse data. This generator can import any dataset using a provided URL from a suitable dataset website. Currently, it supports Kaggle[14]. The downloaded dataset is then compressed into organized bundles. These bundles can then be relabeled with custom labels and preprocessed using custom pre-processing functionality and resized to any dimensions, providing a high degree of modifiability and flexibility. This functionality allows the test environment to be adapted as needed, making it more reflective of real-world scenarios and highly beneficial for simulated FL environments.

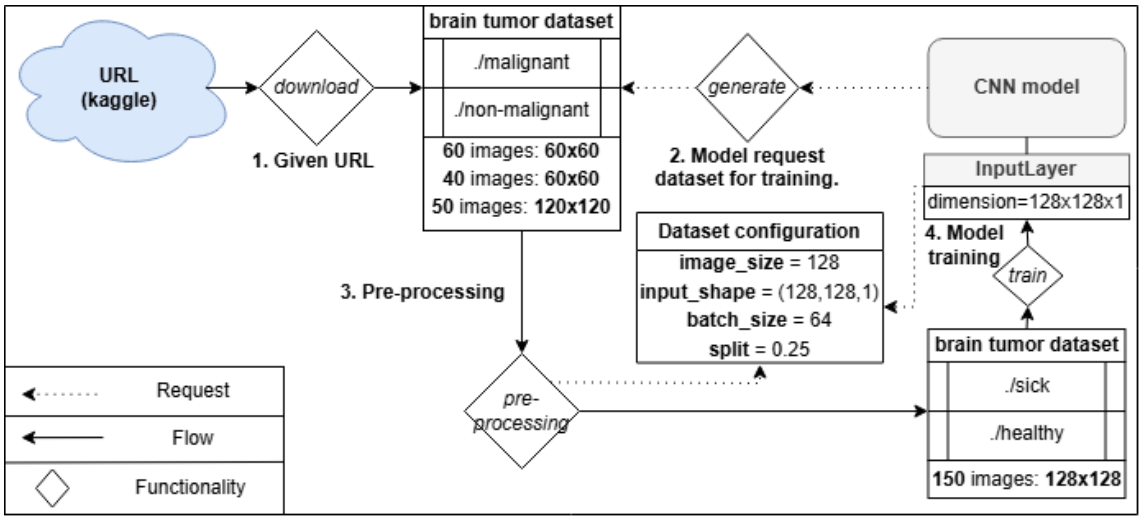

Figure 6: Dataset: Generator Flowchart.

In Figure 6, after step 3, all images are resized to uniform dimensions according to the dataset configuration, and labels are standardized to custom categories, "sick" or "healthy," to align with a common objective, despite potential structural differences in the data across environments. Observe that before the pre-processing procedure, the 150 images of different dimensions all become the same dimensions of $128x128$ matching the input layer (more than 150 images are used in the simulation).

Since the images must match the dimensions of the model's input layer, the dataset configuration serves as a bridge, aligning the datasets with the model requirements. As observed in Figure 6, the dimension of the input layer is set to the *input_shape* of the dataset configuration.

2.1.3 Poisoned & OOD Dataset

For the poisoned dataset, a subset of the complete dataset used by the various local models will first be constructed. Subsequently, label flipping will be applied to introduce poisoned data into the FL environment, compromising the global model's integrity.

This already exists in the provided code as a labeled flipped version of the Alzheimer dataset generated by the dataset generator.

In addition to modifying samples from the In-Distribution (ID) dataset for the local models, the supplementary code also includes an "Animal Face" dataset. This dataset can be used as an additional bonus for evaluating OOD detection on entirely new data.

2.2 Design & Requirements

This section applies important design fundamentals to the implementations in order to emphasize core concepts necessary for comprehension and practical relevance. Implementation-level specifics will be presented in the section Implementation.

Following this design is crucial.

2.2.1 Local

The FL environment will be deployed locally. Meaning that the solution does not require any networking capabilities.

In this laboratory, the server is referred to as the global model, while the client(s) are referred to as local model(s), as they are maintained locally.

It is important, however, to use an appropriate data structure and identifiers to distinguish each local model from one another and to differentiate the global model from the local models.

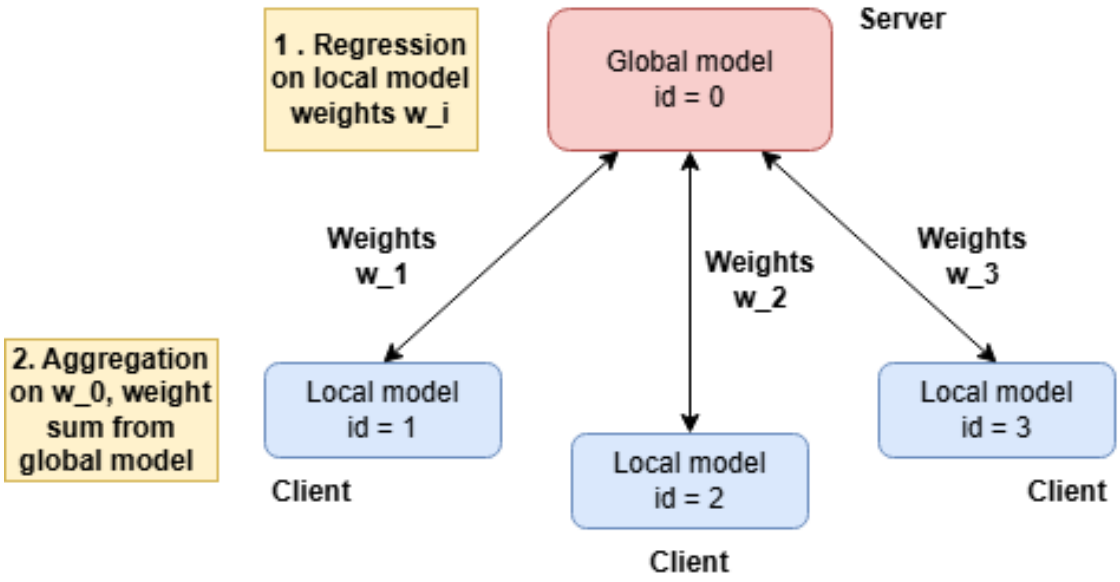

Figure 7: FL Environment, Model Overview.

In Figure 7, each local model is assigned a unique identifier do differentiate each CNN model.

2.2.2 Sequential

The simulation will be implemented sequentially. The different participants in the FL environment do not need to train in parallel; instead, each local model can be iterated and trained separately. Each sequential step of a single simulation round is illustrated as a block with a label in Figure 8 & 9.

The simulation is divided into two distinct phases.

Phase 1 The first phase is a pre-training stage in which each model (the global model and exclusively non-malicious local models) is trained on its respective non-malicious image samples.

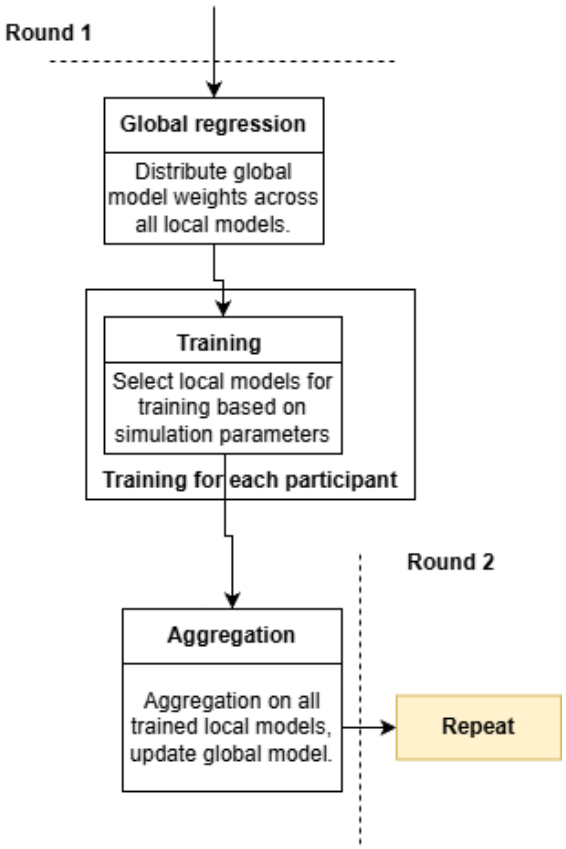

Figure 8: FL Environment, Pre-training

In Figure 8, standard FL functionality is illustrated, where the global model distributes its internal state to the local models. Each local model then performs training, after which all trained local models are aggregated using an aggregation algorithm that combines the updates and integrates them into the global model. This aggregation process is described in more detail in Section 3, implementation.

After pre-training is completed, it is essential to save the models in their entirety, including the full internal model state rather than only the weights, in order to ensure training continuity and reproducibility when the models are resumed or extended in a subsequent phase. Saving only the weights may result in performance degradation when transitioning to Phase 2.

Phase 2 The second phase is a continuation stage in which the OOD detection mechanism is integrated alongside further training of the Phase 1 pre-trained models, with the objective of mitigating OOD updates originating from underlying OOD data. During this phase, malicious local models are also introduced and trained on combinations of malicious data to evaluate the robustness of the OOD detection mechanism.

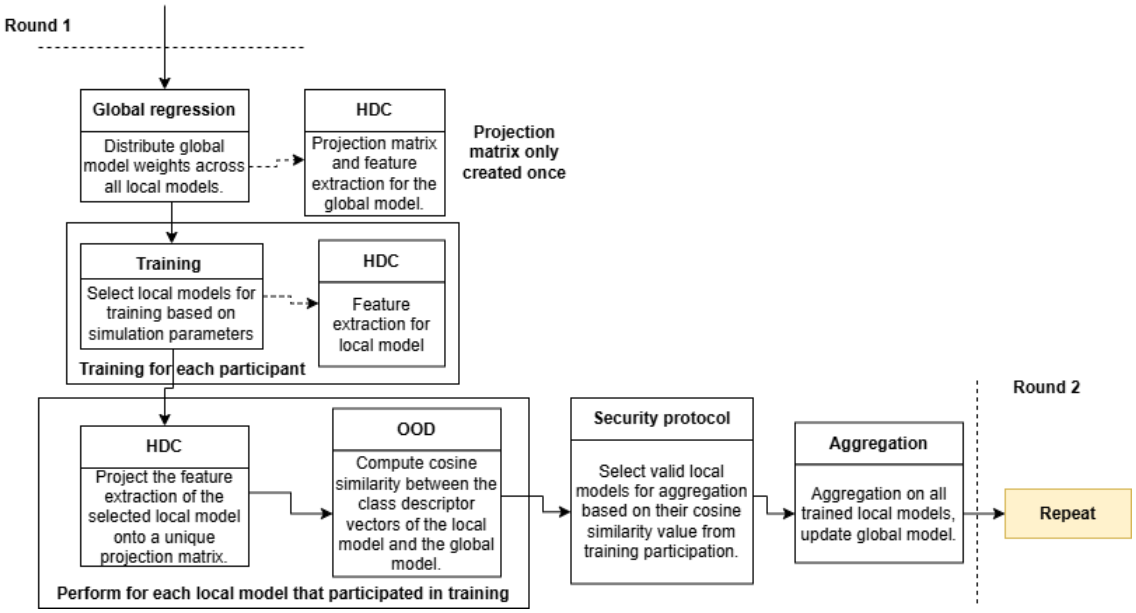

Figure 9: FL Environment, Training & OOD

Figure 9 illustrates a single sequential simulation cycle that your simulation should capture during OOD phase. Each component is executed sequentially; however, the HDC feature extraction step (indicated by the dashed arrow adjacent to training and global regression) must be completed before projection onto the projection matrix and subsequent cosine similarity–based comparison for OOD detection. Subsequently, it is also important that feature extraction for each local model is performed after training so that outdated weights are not used.

Figure 8 & 9 illustrates the resulting simulation procedure for each training round, which your implementation is required to capture in full.

2.2.3 Configurability, Modifiability & Simulation Parameters

The simulation shall be configurable, enabling straightforward reproduction of diverse scenarios and results. Although execution within a Python notebook is advantageous for testing individual scenarios, deployment and execution in this case will be streamlined through standard command-line Python execution on the local computer, as multiple scenarios will be evaluated. Save all results (plots, graphs and important printouts) and include them in your final document for reflection.

However, if you prefer to develop in a Python notebook, you are free to do so, provided that the code is kept modular for your own benefit.

The following Table 1 provides an example of parameters that may be used; the parameter names, exact values, and even the number of

parameters do not need to be identical. Its purpose is to illustrate the types of parameters that may be required for your simulations.

Simulation parameters	
Parameter	**Value**
Iterations	1
Rounds	40
Epochs	1
OOD round	*None*
Pre-trained	*False*
Pre-rounds	*None*
Clients	5
Participants	4
Hyper size	1*e*4
OOD protection	*False*
OOD threshold	0.5
ID clients	$[1, 2, 3, 4]$
OOD clients	*None*

Dataset parameters		
Client id	**Dataset**	**Features**
0 (Global model)	All (ID)	18000
1	Brain tumour 1 (ID)	4600
2	Brain tumour 2 (ID)	3000
3	Alzheimer (ID)	5100
4	Pneumonia (ID)	5200

Table 1: Simulation Parameters: Pre-training Example

Iterations: The number of times the simulation is re-executed to compute averaged results. Each iteration starts from a fresh state, with only the results being retained "outside" the simulation.

Rounds: Number of federated simulation rounds.

Epochs: The number of epochs each participant trains for (model-specific). It is generally sufficient to set this to one for the entire federated learning simulation. In some cases, a specific model may be trained for more than one epoch within a single participation round.

Total of number of training rounds

$$\#\,\text{trainings} = \text{rounds} \times \text{epochs} \times \text{participants} \tag{1}$$

OOD round: The round at which OOD detection is activated and malicious clients begin to influence the simulation. OOD protection must be enabled for this parameter to take effect; consequently, OOD detection can remain disabled during

the pre-training phase.

Pre-trained: If enabled, pre-trained models from Phase 1 is loaded for the simulation, as specified in Section 2.2.2.

Pre-rounds: The total number of rounds the model underwent during pre-training. Specify which models to load.

Clients: Number of global models + local models (clients) included in the simulation.

Participants: Number of local models participating in each training round. (global model should be excluded).

Hyper size: Hyperdimensional size of the projection matrix. Used in HDC and OOD detection.

OOD protection: Indicates whether protection against malicious clients during regression is enabled. Should be disabled for Phase 1, enabled for Phase 2.

OOD threshold: Threshold value for cosine similarity at which local models are classified as OOD in the safety mechanism. Local model vs. Global model. If similarity below, exclude model from aggregation into global model.

ID clients: Clients utilizing ID data, used during the evaluation phase to assess anomaly detection success.

OOD clients: Malicious clients using OOD data, evaluated during the assessment phase to measure anomaly detection success.

In Table 1, the table below, displays the client/local model and dataset assignments, indicating which local model has access to which dataset.

Table 1 also depicts a typical simulation setup for pre-training a model on ID datasets. Where OOD detection is disabled for pre-training purposes.

2.2.4 Plot, Diagrams & Producing Results

It is important that your results are presented using appropriate tools that enable clear analysis and reflection. The provided code includes functionality for this purpose through various plotting scripts, which may need to be adapted depending on how the functionality is implemented in your code. Below are example plots illustrating how results are expected to be presented.

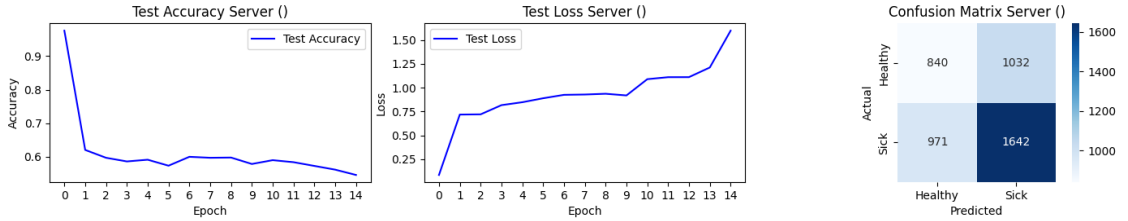

Figure 10: Plot 1: Global model (server) accuracy, loss & confusion matrix.

In Figure 10, each time Phase 1 or Phase 2 is executed, as described in Section 2.2.2, the accuracy of the global model must be reported, along with a plot of its convergence over each training round in the FL simulation. The global model does not perform training; it only evaluates performance on all ID data included in the simulation across the local models.

If desired, plots for each local model may also be implemented; however, it is sufficient to include only the global model test accuracy, loss and confusion matrix in the final report.

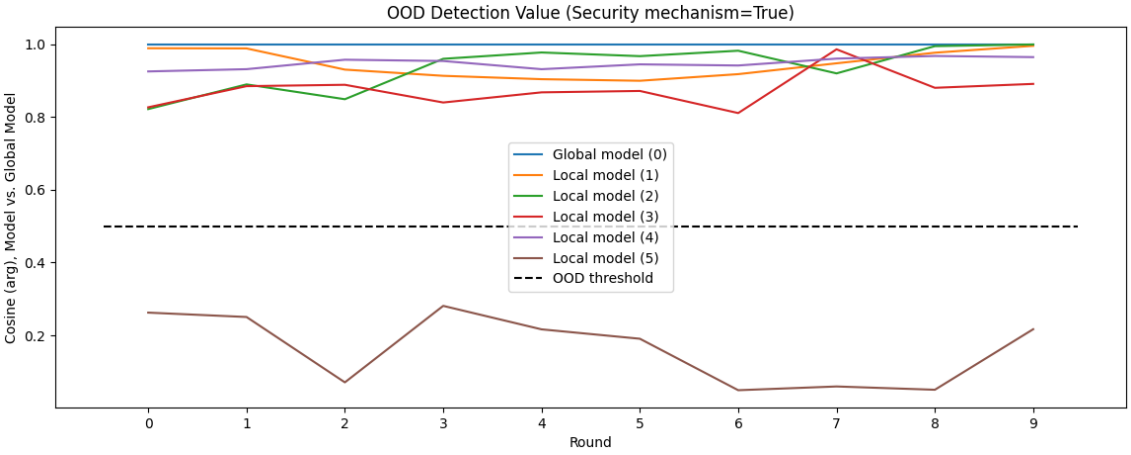

Figure 11: Plot 1: OOD detection, similarity values of local models vs. global model

In the example Figure 11, the similarity values between the class descriptor vectors of the global model and each participating local model are shown. In this example, local model (id=5) is trained on OOD data and is successfully identified by the OOD detection mechanism (since below OOD threshold). The remaining local models in the plot train exclusively on ID data and are correctly treated as ID (above OOD threshold), and are therefore included in the aggregation process in FL with OOD detection, as observed.

Both Figures 10 and 11 for each simulation scenario should be discussed and included in your final report; however, the plots do not need to be identical.

3. IMPLEMENTATION

This section covers additional implementation details and provides a step-by-step guide for assisting development.

3.1 Task 1: Setup & Run

Run the Makefile, which is designed for Linux systems using the apt package manager (included in Ubuntu). If this setup is not available on your system, you may either adapt the Makefile to your local environment or execute the commands manually. Typically, only the make install target requires such adjustments; all other steps rely on standard Python operations.

If the Makefile executes successfully, all data will be downloaded automatically and a CNN model will be trained. Ensure that this process completes correctly before proceeding to Task 2.

3.1.1 Q&A

Do I have to use the Makefile? No. The Makefile is provided for convenience and automates environment setup, dependency installation, data download, and initial model training. You may run the commands manually if preferred, provided that all required dependencies are installed correctly.

Which operating systems are supported? The Makefile is designed for Linux systems using the apt package manager (as available in Ubuntu). If you are using another Linux distribution, Windows, or macOS, you will need to adapt the installation steps or execute them manually. Windows users may optionally use WSL.

What does the make install target do? The make install target installs required system-level and Python dependencies. This is typically the only part of the Makefile that may need modification for different operating systems or package managers.

Is a virtual environment created automatically? Yes, the Makefile creates and configures a Python virtual environment. If you run the setup manually, you are responsible for creating and activating a suitable virtual environment yourself.

Where are the datasets downloaded? The datasets are downloaded automatically to the directories specified in the dataset generator. You do not need to download them manually unless the process fails.

Do I need to re-download the datasets every time? No. Once downloaded, the datasets are reused unless you explicitly delete them or change the dataset paths.

How do I know if the Makefile ran successfully? A successful run will complete without errors, download all required datasets, and train a CNN model. You should see training logs and generated output files (such as saved models or plots).

How long should the initial training take? Training time depends on your hardware, but it should complete within a reasonable timeframe on a standard CPU-based system. Excessively long runtimes may indicate configuration or dependency issues. If this becomes a problem, you may reduce the model complexity; however, doing so may require adjusting the dataset configuration to ensure compatibility with the modified model's input layer.

What should I do if the Makefile fails? Check the error message carefully. Common issues include missing system dependencies, incompatible Python versions, or permission errors. In such cases, running the commands manually often helps isolate the problem.

Can I skip the training step if I already have a trained model? Yes, provided that the model is compatible with the codebase and configuration. Ensure that the model is correctly executed and validated before proceeding to Task 2.

What must be verified before proceeding to Task 2? You must confirm that the environment is correctly set up, the datasets are downloaded, and the CNN model trains successfully without errors.

The file `python_requirements.txt` specifies `tensorflow[and-cuda]`; what should I do if I do not have access to a GPU? It should be able to run on a CPU-only machine as well; a GPU is not required, although it will speed up the simulation. If you encounter issues with `tensorflow[and-cuda]`, you can replace it with the standard `tensorflow` package.

What if I am unable to run the provided code because my local machine is a brick? You may run the entire workflow in Google Colab; however, the provided codebase may not be fully compatible, particularly the dataset generator, which is designed to run on a local machine. As a result, you may need to select alternative datasets or download the datasets manually. The provided code is intended to save time by avoiding the need to reimplement the CNN model or handle data preprocessing and dataset downloads, as these aspects are outside the scope of this laboratory exercise.

3.2 Task 2: Implementing Phase 1

As described in Section 2.2.2, Phase 1 is defined as the pre-training step in which the global model and local models are pre-trained on ID data. The dimensionality and structure of each DL for the global model and local models are identical.

3.2.1 Initialize

Implement functionality for uniquely identifying each model. For example, the global model may be assigned `id=0`, while also local models are created iteratively based on a client parameter and assigned unique identifiers. Use an appropriate data structure both to distinguish between models and to associate each model with its corresponding dataset.

For global model:

1. Assign an identifier to the model (for example, assuming the global model always has `id=0`) and register it in an appropriate model identifier data structure, such as a list or dictionary.

2. Assign datasets. In the provided code, the mapping `client_to_dataset = [[0,1,2,3],[0],[1],[2],[3]]` can be used to specify which datasets each model has access to. For example, calling `Dataset.get(client_to_dataset[global_model_id])` retrieves all datasets at indices 0, 1, 2, and 3 from `Dataset()` for the global model. Which are all ID datasets that we want to measure performance of global model on. You are free to edit or design this how you want.

3. Create a copy of the CNN model to serve as the global model using, for example, `copy.deepcopy(model)` from the Python standard library. It is important that this copy is a completely separate instance; otherwise, the global model and all local models may reference the same underlying object. You may encounter a warning indicating that the model needs to be recompiled after copying; this warning can be safely ignored. The copied models can be stored in an appropriate data structure, such as a list or dictionary.

For each local model: Identical to previous step, but ids and assigned datasets are different.

You may iterate over this based on the client parameter provided in the supplementary code, or implement an alternative design of your choosing.

The initialization step only needs to be executed once; thereafter, the simulation is started and training proceeds for the specified number of rounds.

3.2.2 Regression (Training)

Distribute global model weights to local models.

For each local model, take the global model weights and just set them for each local model: `model.set_weights(global_model.get_weights())`

3.2.3 Train (Training)

Iterate over all models and set the number of training epochs for each specific model to one. Adjust the number of iterations for the FL simulation according to the desired amount of training (rounds).

If local model: Train and evaluate the model on its assigned dataset.

If global model: Perform testing only, using the assigned dataset (ALL ID datasets included in the FL simulation).

3.2.4 Aggregation (Training)

We will be using Federated Averaging [15] to combine local models weights and update the global model.

Let:

- K: total number of local models trained
- w_t: global model weights at round t
- w_t^k: local model weights from client k after local training

The Federated Averaging (FedAvg) update rule for weight regression, is given by:

$$w_{t+1} = \sum_{k=1}^{K} \frac{1}{K} w_t^k \tag{2}$$

In practice, this can be implemented by iterating through each hidden layer of all trained local models (participants) at round t.

For each layer in the CNN model, collect the corresponding weights from all participating local models and compute their mean using, for example, `layer_mean = tf.math.reduce_mean(weights, axis=0)`. Store the result for the corresponding hidden layer index and repeat this process for each layer.

After iterating through all layers across all participants, use the resulting set of averaged weight vectors to update the global model by calling:

```
global_model.set_weights(resulting_weights)
```

3.2.5 Result

Train the FL environment for a number of rounds and record the test accuracy and test loss of the global model, which can then be visualized as shown in Figure 10.

In `model/model.py`, the functions `plot_all()` and `plot_test()` are available for this purpose. These can be invoked, for example, by calling `global_model.plot_all()` and `global_model.plot_test()`.

You should also include informative console output that allows you to monitor and inspect accuracy during execution. Exists already to some extent.

3.2.6 Saving & Loading Model

Implement functionality for saving models after training and for loading models prior to training. This can be added once the simulation is functioning correctly. You may use the provided example code for saving and loading models; for loading, the existing IF/THEN logic will need to be adjusted accordingly.

For saving:

```
model.save(self.federated_config.path + "model" + str(id) + "_" + "round"
+ str(round) + ".keras")
```

For loading:

```
model_path = f¨{self.federated_config.path}model{id}_round{self.federated_
config.load_round}.keras"
IF os.path.exists(model_path):
THEN model.model = tf.keras.models.load_model(model_path)
```

You can use any other structure for saving / loading if you want.

If a simulation is later executed with five local models while only four local models were previously saved, it is important that the saving functionality stores each model using the identifier it had during training. When loading, the implementation should also handle this mismatch gracefully and avoid crashing if fewer models are available than requested.

It is also very important that the models are saved in their entirety, rather than saving only the weights.

3.3 Task 3: Implementing Phase 2

First, implement the functionality for the OOD detection mechanism, and then integrate it into the environment.

The OOD detection mechanism is inspired by Hyperdimensional Feature Fusion (HDFF)[16] and HDC. In that work, the same underlying principle is applied to

input samples; however, in FL such access is typically not available in real-world deployments. Instead, this approach constructs a single high-dimensional representation of a model by extracting layer outputs, projecting them into a common hypervector space, and bundling them together. This representation enables comparison between two models (e.g., global versus local) by computing similarity between their fused hypervectors, which is useful for OOD detection or drift analysis.

You are provided with utility functions in `ood/VSA.py` to support operations on these high-dimensional vectors. In `HDFF.py` you have under each function more in detail of how you can implement this, some hints.

3.3.1 Creating Projection Matrices

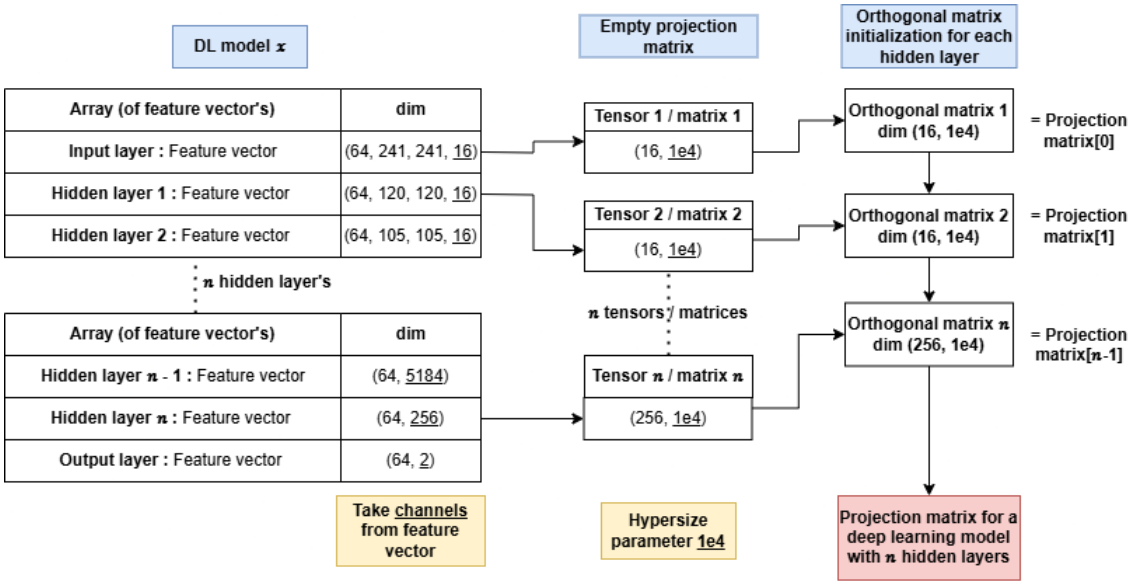

Figure 12: OOD: Generating a Projection Matrix from the Global Model.

In Figure 12, the projection matrix is generated for the global model and later used to project the model weights from each layer in the local models, resulting in projected feature maps.

The projection matrix is essential for enabling algebraic operations between the holistic representations of each hidden layer by projecting them into a high-dimensional space.

3.3.2 Creating Feature Vectors

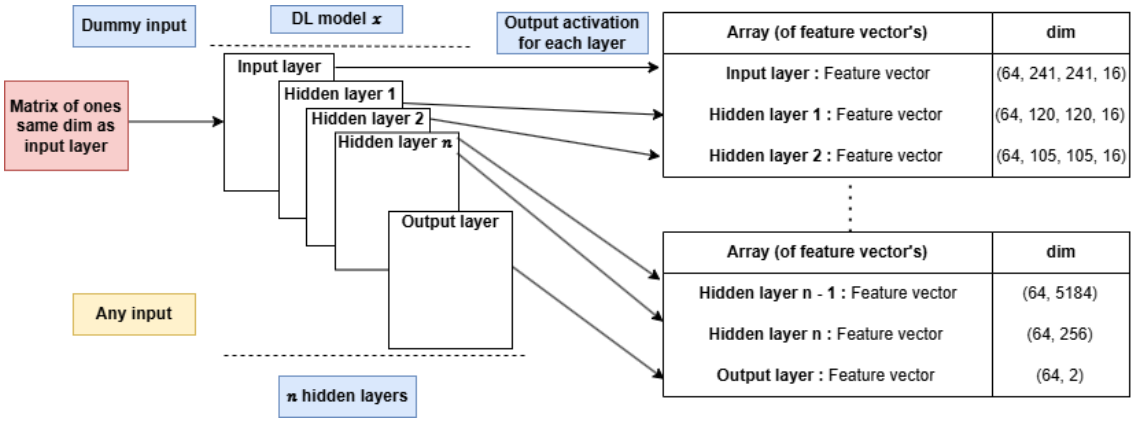

Figure 13: OOD: Feature Map Generation from a Deep Learning Model.

In Figure 13, a feature vector is generated by capturing the output from a hidden layer, obtained when the neural network processes a given input.

To facilitate this process, a dummy input—typically a matrix of ones or randomly initialized values matching the input layer's dimensions—is used. Since actual training samples from local models cannot be reused due to access restrictions (coherent with FL). Consequently, the CNN model performs a feedforward pass using the dummy input, and the output from each hidden layer is collected by a feature extractor, which generates the corresponding feature vectors. The feature extractor copies each output and appends it to its corresponding position in the feature vector.

In this simulation, all local models are accessed locally, making it easy to extract features from the hidden layers since they are readily available. However, in practice, a temporary model could be created using the weights sent to the global model during regression. Feature vectors could then be extracted from this temporary model.

3.3.3 Projection, Bundling & Cosine Similarity

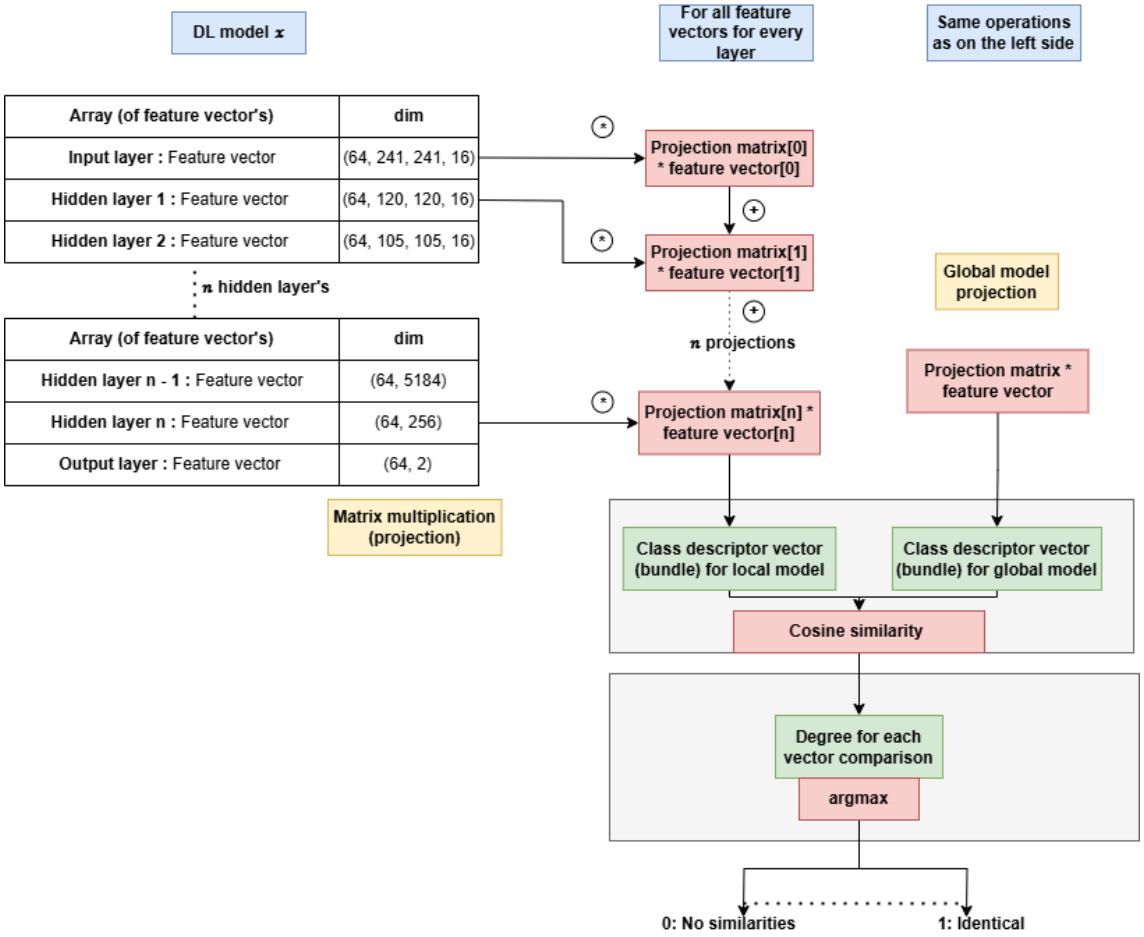

Figure 14: OOD: Projecting Feature Vectors, Bundling, and Applying Cosine Similarity to the Result.

In Figure 14, feature vectors from each hidden layer are projected onto a corresponding projection matrix.

Given a feature vector $\mathbf{f}^{(h)} \in \mathbb{R}^d$ from hidden layer h, it is projected into a high-dimensional space using a semi-orthogonal projection matrix $\mathbf{P}^{(h)} \in \mathbb{R}^{D \times d}$:

$$\mathbf{v}^{(h)} = \mathbf{P}^{(h)} \mathbf{f}^{(h)} \tag{3}$$

where $D \gg d$ and $\mathbf{v}^{(h)} \in \mathbb{R}^D$ is the projected hypervector.

These projected vectors are then bundled to form class descriptor vectors, based on Hyperdimensional Computing (HDC) principles described in Section **??**—specifically, using binding during projection and superposition (addition) during bundling.

Projected vectors from all considered layers $h \in H$ are bundled into a single class descriptor vector via superposition (vector addition):

$$\mathbf{v}_{\text{model}} = \sum_{h \in H} \mathbf{v}^{(h)} \tag{4}$$

To detect potential alterations, cosine similarity is used to compare the global model's class descriptor vector with those from local models in the Federated Learning (FL) environment. Deviations in similarity indicate possible tampering or the influence of malicious data on local model updates during backpropagation.

To compare a local model's descriptor vector $\mathbf{v}_{\text{local}}$ to the global model's descriptor vector $\mathbf{v}_{\text{global}}$, cosine similarity is computed as follows:

$$\text{sim}(\mathbf{v}_{\text{local}}, \mathbf{v}_{\text{global}}) = \frac{\mathbf{v}_{\text{local}} \cdot \mathbf{v}_{\text{global}}}{\|\mathbf{v}_{\text{local}}\| \, \|\mathbf{v}_{\text{global}}\|} \tag{5}$$

Therefore, a cosine similarity close to zero indicates that the local model is OOD, while a value close to one signifies that the local model is ID.

3.3.4 Integration into FL environment

Once the OOD detection mechanism is implemented, it can be integrated into the aggregation process. Before applying Federated Averaging to all participating local models, compare the class descriptor vector of each local model (after training) with that of the global model. Based on the resulting cosine similarity, create a very simple security protocol, and discard local models whose similarity falls below a defined OOD threshold, and exclude them from the aggregation step.

To visualize similarity score results, plotting functionality is available in `federated/math/plot.py`, which may need to be modified depending on your implementation.

It is also advisable to include a configuration variable that can disable both the OOD detection and the security protocol components, as these should be turned off during the pre-training phase.

3.4 Task 4: Experimentation

Pre-train all models on all ID data for 30–40 rounds, with the OOD detection mechanism disabled during this phase. Refer to Table 1, which presents a configuration similar to the pre-training setup. Save the trained models to be used in the subsequent experiments.

3.4.1 Experiment 1: 1 OOD local model with complete poisoned dataset (OOD), OOD detection disabled

1. Global model(id=0), testing on all ID datasets used in pre-training for all local models. `Btumor4600()` `Btumor3000()` `Balzheimer5100()` `Lpneumonia5200()`

2. Local model(id=1), training on OOD data using complete label-flipped, poisoned dataset. `Balzheimer5100_poisoned()`

Disable the OOD detection mechanism.

Load the models using the previously pre-trained models and disable the OOD detection mechanism. It should function correctly, provided that your implementation supports loading previously pre-trained models even when the number of local models differs.

Use the complete `Balzheimer5100_poisoned()` dataset for the single local model. In the supplementary code, the dataset `Balzheimer5100_poisoned()` is provided, which consists exclusively of label-flipped samples.

Run the simulation with only one local model for five rounds and record your result based on Section 2.2 and include in your report.

You should observe a deterioration in the accuracy of the global model.

3.4.2 Experiment 2: 1 OOD local model with complete poisoned dataset (OOD), OOD detection enabled

1. Global model(id=0), testing on all ID datasets used in pre-training for all local models.

2. Local model(id=1), training on OOD data using complete label-flipped, poisoned dataset. `Balzheimer5100_poisoned()`

Enable the OOD detection mechanism.

Use the complete `Balzheimer5100_poisoned()` dataset for the single local model.

Run the simulation with only one local model for five rounds and record your result based on Section 2.2 and include in your report.

The OOD detection mechanism should be capable of identifying the OOD local model; if it fails to do so, there is likely an issue in the implementation. This behavior has been thoroughly validated through prior experimentation and research.

3.4.3 Experiment 3: 1 new OOD local model with complete poisoned dataset (OOD), OOD detection enabled

1. Global model(id=0), testing on all ID datasets used in pre-training for all local models.
2. Local model(id=1) `Btumor4600()`
3. Local model(id=2) `Btumor3000()`
4. Local model(id=3) `Balzheimer5100()`
5. Local model(id=4) `Lpneumonia5200()`
6. Local model(id=5), training on OOD data using complete label-flipped, poisoned dataset. `Balzheimer5100_poisoned()`

Load the local models using the previously pre-trained models and use the previously ID data samples. Then introduce an additional model which will be training on OOD data, use the fully poisoned version of any ID dataset (labeled flipped).

Run the simulation for five rounds and record your result based on Section 2.2 and include in your report.

The OOD detection mechanism should be capable of identifying the OOD local model; if it fails to do so, there is likely an issue in the implementation. This behavior has been thoroughly validated through prior experimentation and research.

3.4.4 Experiment 4: 1 new OOD local model with half samples of poisoned data and half ID data, OOD detection enabled

1. Global model(id=0), testing on all ID datasets used in pre-training for all local models.
2. Local model(id=1) `Btumor4600()`
3. Local model(id=2) `Btumor3000()`
4. Local model(id=3) `Balzheimer5100()`
5. Local model(id=4) `Lpneumonia5200()`
6. Local model(id=5), training on OOD data using complete label-flipped, poisoned dataset. `Balzheimer5100_poisoned()` and 500-1000 samples from `Lpneumonia5200()`

Use 500-1000 ID samples from any ID dataset and the OOD poisoned dataset for the new OOD local model and run the simulation for five rounds and record your result based on Section 2.2 and include in your report.

3.4.5 Experiment 5: 1 new OOD local model with new OOD dataset, OOD detection enabled

In this experiment, a new dataset is introduced for training the OOD local model.

1. Global model(id=0), testing on all ID datasets used in pre-training for all local models.
2. Local model(id=1) `Btumor4600()`
3. Local model(id=2) `Btumor3000()`
4. Local model(id=3) `Balzheimer5100()`
5. Local model(id=4) `Lpneumonia5200()`
6. Local model(id=5), training on OOD data using the new OOD dataset.`Afaces16000()`.

Run the simulation for five rounds and record your result based on Section 2.2 and include in your report.

3.4.6 (Bonus) Experiment 6: Design and evaluate an experiment scenario of your choice with OOD detection enabled.

Run the simulation for five rounds and record your result based on Section 2.2 and include in your report.

3.5 Submission

For submission, you should provide both your codebase and a final report that reflects on the experimental results presented in Section 3.4. The report should clearly and concisely discuss why the obtained results align with or differ from the expected outcomes.

References

[1] Subhash Sagar, Chang-Sun Li, Seng W. Loke, and Jinho Choi. Poisoning attacks and defenses in federated learning: A survey. *arXiv:2301.05795*, 2022. doi: 10.48550/arXiv.2301.05795. URL https://arxiv.org/abs/2301.05795.

[2] Lingjuan Lyu, Han Yu, and Qiang Yang. Threats to federated learning: A survey. *arXiv:2003.02133*, 2020. doi: 10.48550/arXiv.2003.02133. URL https://arxiv.org/abs/2003.02133.

[3] Hengtong Zhang, Jing Gao, and Lu Su. Data poisoning attacks against outcome interpretations of predictive models. *KDD '21: Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 2165–2173, 2021. doi: 10.1145/3447548.346745. URL https://dl.acm.org/doi/10.1145/3447548.3467405.

[4] Xinsong Ma, Zekai Wang, and Weiwei Liu. Truth serum: Poisoning machine learning models to reveal their secrets. *ACM CCS 2022*, 2022. doi: 10.48550/arXiv.2204.00032. URL https://arxiv.org/abs/2204.00032.

[5] Reza Shokri, Martin Strobel, and Yair Zick. On the privacy risks of model explanations. *arXiv:1907.00164*, 2021. doi: 10.48550/arXiv.1907.00164. URL https://arxiv.org/abs/1907.00164.

[6] Yao Chen, Yijie Gui, Hong Lin, Wensheng Gan, and Yongdong Wu. Federated learning attacks and defenses: A survey. *IEEE BigData*, page 10, 2022. doi: 10.48550/arXiv.2211.14952. URL https://arxiv.org/abs/2211.14952.

[7] Liwei Song, Reza Shokri, and Prateek Mittal. Privacy risks of securing machine learning models against adversarial examples. *ACM CCS 2019*, 2019. doi: 10.48550/arXiv.1905.10291. URL https://arxiv.org/abs/1905.10291.

[8] Han Xu, Xiaorui Liu, Yaxin Li, Anil K. Jain, and Jiliang Tang. To be robust or to be fair: Towards fairness in adversarial training. *arXiv:2010.06121*, 2020. doi: 10.48550/arXiv.2010.06121. URL https://arxiv.org/abs/2010.06121.

[9] Florian Tramèr, Reza Shokrim, Ayrton San, Joaquin Hoang, Le Matthew, Jagielski Sanghyun, and Hong Nicholas Carlini. On the tradeoff between robustness and fairness. *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=LqGA2JMLwBw.

[10] Yasir Hussein Shakir. Dataset_alzheimer, 2021. URL https://www.kaggle.com/datasets/yasserhessein/dataset-alzheimer.

[11] Preet Viradiya. Brian tumor dataset, 2021. URL https://www.kaggle.com/datasets/preetviradiya/brian-tumor-dataset.

[12] Abhranta Panigrahi. Brain_tumor_detection_mri, 2021. URL https://www.kaggle.com/datasets/abhranta/brain-tumor-detection-mri.

[13] Paul Mooney. Chest x-ray images (pneumonia), 2018. URL https://www.kaggle.com/datasets/paultimothymooney/chest-xray-pneumonia.

[14] Google LLC. Dataset platform, 2010. URL https://www.kaggle.com/.

[15] Tao Sun, Dongsheng Li, and Bao Wang. Decentralized federated averaging, 2021. URL https://arxiv.org/abs/2104.11375.

[16] Samuel Wilson, Tobias Fischer, Niko Sunderhauf, and Feras Dayoub. Hyperdimensional feature fusion for out-of-distribution detection. *arXiv:2112.05341*, v3, 2022. doi: 10.48550/arXiv.2112.05341. URL https://arxiv.org/abs/2112.05341.