UNIVERSITÀ DI PISA

MSc in Artificial Intelligence and Data Engineering

Distributed Systems and Middleware Technologies

Project Documentation

FEDERATED LEARNING

Team Members:

Ivan Brillo

Daniel Pipitone

Andrea Bochicchio

Academic Year 2024 - 2025

Contents

1 Federated Learning

Federated Learning is a distributed approach to machine learning that enables collaborative model training across multiple nodes without requiring data centralization. Instead of sharing raw data, participants transmit model updates (e.g. gradients or weights) to a central server for aggregation.

Horizontal Federated Learning (HFL) applies when participants have datasets with the same feature space but different samples. In HFL, each participant trains a local model on its dataset and periodically shares updates with a central server that aggregates the updates. This technique leverages the computational power at the data source while avoiding the sharing of big datasets of sensitive information across the network.

A compelling use case for our application is within a distributed network of hospitals. Using Horizontal Federated Learning, hospitals can collaboratively train AI models on sensitive patient data, such as medical imaging or health records, without ever sharing raw data with the central server.

1.1 Privacy Concerns

Privacy concerns are a significant challenge in machine learning training, especially in models with sensitive data, such as personal information, medical records, or financial transactions. Centralizing such data in a single location for training creates the risks of unauthorized access, data breaches, and non-compliance with privacy regulations. Horizontal Federated Learning mitigates these concerns by keeping data on local devices or servers and only sharing model updates with a central aggregator. This decentralized approach ensures that raw data remain private and never leave its source, significantly reducing the risk of exposure.

1.2 Weights Aggregation

The **Federated Averaging (FedAvg)** algorithm is the most common method for aggregating model weights in HFL. The global model weights $w^{(t+1)}$ in round $t+1$ are computed as a weighted average of the local model weights $w_k^{(t)}$ of the participating nodes.

$$w^{(t+1)} = \sum_{k=1}^{N} \frac{n_k}{n} w_k^{(t)},$$

where:

- N is the total number of participating nodes.
- n_k is the number of training data samples on node k.
- $n = \sum_{k=1}^{N} n_k$ is the total number of training samples across all nodes.
- $w_k^{(t)}$ are the model weights from node k at round t.

2 Requirements

2.1 Functional Requirements

1. The system must allow a neural network to be trained on the available nodes, following the HFL principles.
2. The system must enable the master to specify parameters (epochs and accuracy thresholds) for the training of the neural network.
3. The system must aggregate the parameters sent by the nodes to update the master model using the FedAvg algorithm.
4. The system must provide an interface for the administrator.
5. The system must permit communication only with trusted nodes.
6. The system must allow the administrator to monitor the real-time metrics of the nodes.
7. The system must enable the administrator to run multiple experiments simultaneously, even with different participants.
8. The system must allow the administrator to save or load the model parameters to persistent storage.
9. The system must enable the administrator to provide the TensorFlow model definition for the neural network.
10. The system must allow the administrator to interrupt the training of the model.

2.2 Non-Functional Requirements

1. The system must minimize the data transmitted across nodes to avoid network overload.
2. The system must be fault-tolerant with respect to node crashes.
3. The system must handle network partitions efficiently.
4. The system must ensure the privacy of the data used by the nodes to train the model.
5. The system must be flexible enough to allow the administrator to switch the model to train in a simple manner.

3 The Project and Its Use Cases

We have developed a platform designed to perform and monitor the training of TensorFlow deep learning models within a Horizontal Federated Learning framework. A notable use case for this platform involves a network of hospitals collaboratively training a tumor classifier model. This represents one of several powerful applications of the platform, which can be extended to a wide range of other fields, including autonomous vehicles, financial services, and beyond.

3.1 How to Set Up the Network

After installing all the requirements, we can set up a master or a slave node. In general, we need to:

1. If configuring the master, in the *Erlang/.hosts.erlang* file, add the hostnames of all nodes that can take part in the computation (if they have the right cookie). In a slave node, just the master host will be sufficient

```
'slave1'.
'slave2'.
'slave3'.
'slave4'.
'slave5'.
```

Figure 3.1: A possible master *Erlang/.hosts.erlang* file

2. Update the node */etc/hosts* file with the host-IP mapping of the hosts added in *Erlang/.hosts.erlang* file

```
10.2.1.17  master
10.2.1.18  slave1
10.2.1.15  slave2
10.2.1.16  slave3
10.2.1.19  slave4
10.2.1.20  slave5
```

Figure 3.2: A possible master */etc/hosts* file

3. If the node under configuration is a slave, make sure to add its IP-Host in the master configuration files

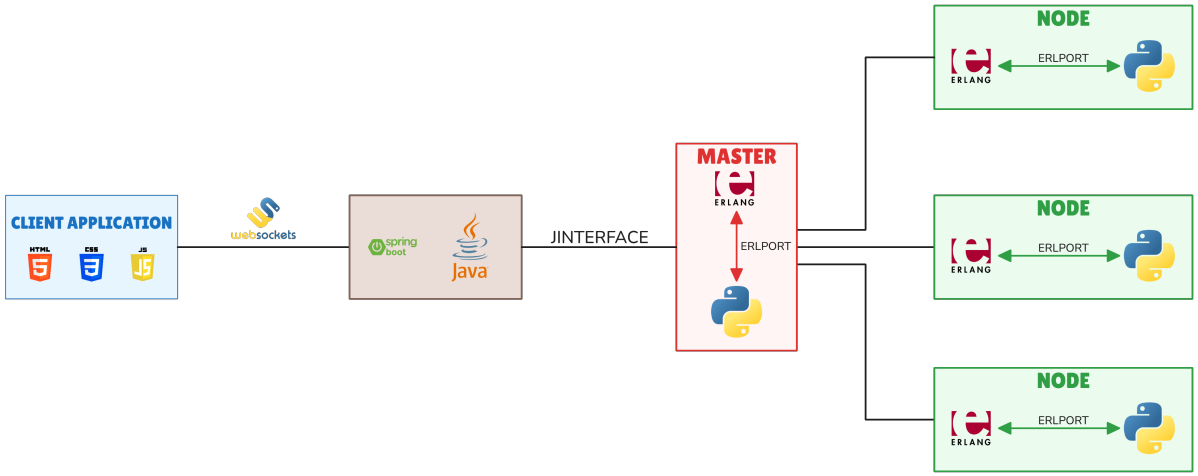
4. If the new node is a slave, implement a Python file called *dataParser.py* in the *priv/* folder with this return structure: *x_train, y_train, x_test, y_test = get_dataset()*. Every element must be a NumPy array of correct dimension, as this will be the input of the TensorFlow *fit* function

5. In the master node, the *tf* model structure can be handled creating a file called *Erlang/priv/modelDefinition.py* containing a sequential object with the name *network_definition*. If a more complex network is needed (for example in VAE), the administrator can modify the *Erlang/priv/networkModel.py* file, implementing the one needed

3.2 Maven and Rebar3 for Build Automation of the Application

Our application is developed using Maven and Rebar3. **Maven** is a robust build automation tool for Java that simplifies project management by handling dependencies, builds, testing, and packaging. **Rebar3** is a widely used build tool for Erlang that helps the programmer by managing dependencies, compiling code, and running tests. Rebar3 also simplifies the integration of Erlang libraries and tools, making it easier to maintain and scale the application over time.

4 System Architecture and Communication



4.1 Master Erlang

The Master Node, implemented in Erlang, acts as the central coordinator in the distributed system. It manages communication with the Java backend and controls the connected Erlang worker nodes. Using Erlang's native distributed messaging system, the master handles the distribution of tasks, the aggregation of results, and any connectivity issues that might arise during system operation.

4.1.1 Supervisor

The OTP supervisor process is the entry point of our Erlang application. It monitors the OTP gen_server process, managing its life-cycle. If the server crash, the supervisor is responsible for handling the failure by restarting the process, ensuring the system remains operational.
The supervisor will attempt a maximum of 10 restarts within 60 seconds. If the child process continues to crash, the supervisor will consider the situation as an indication of a larger issue and will stop further restarts.
It uses a transient restart type. This means that the child will only be restarted if it terminates with an error, but not when it exits normally. The shutdown time is set to 10 seconds to allow for a graceful shutdown before the supervisor forcefully terminates the process.

4.1.2 Master Server

The master server process is the task coordinator of the entire Erlang application. In particular, its main functionalities are:

- **Initialization of Python Master**:
  - The server initializes a Python process responsible for keeping the latest version of the aggregated model.
  - Their connection is a process link. We will not attempt to restart the Python process if it fails, since the computational cost of doing so is almost comparable to restarting the entire master process (by the supervisor), and the complexity overhead would outweigh the benefits.

- **Node Management**:
  - The server manages the nodes in the system, including initializing, loading, and distributing models and weights. It ensures that nodes are operational and updates the UI about the status of nodes.
  - When a node connects or disconnects, the server updates its state accordingly, re-initializing nodes if necessary and ensuring proper handling of live or dead nodes.
- **Training Management**:
  - The server handles the distribution of training tasks across connected nodes. It checks for training conditions (epochs and target train accuracy) and ensures that training is stopped when requested.
  - The master issues the command to load or save the model to Python.
  - During training, it will periodically save the model, which could be useful for reloading in case of a crash.
- **Fault Handling**:
  - The server includes error handling for node disconnections and other possible crashes, restarting the nodes if necessary.
- **UI Updates**:
  - The server updates the UI with the current status of the system.

4.1.3 Contract Between Master and Trusted Nodes

Every node, in order to take part in the distributed application, must finish initialization within 60 seconds. The same is true for completing the training of a specified epoch; otherwise, the result from the node will not be counted.

4.1.4 Aggregated Functionalities: Node Initialization, Load, and Train Pipeline

At every connected node, a new Erlang process is started. This procedure is called *initialize_nodes*. The process will be monitored to receive a 'DOWN' message in case of a crash.
Then, for each initialized node, we need to load the model structure, model weights, and the dataset for computation. These tasks are placed in a pipeline to achieve optimal performance (4.1). By doing so, we leverage the FIFO message ordering guarantees in Erlang. Specifically, if two messages are sent from node A to node B and both are delivered, their ordering is preserved.
The train pipeline is similar. We need to get the weights from the Python model, distribute them among the nodes that will perform an epoch of training, and then gather the updated weights. This list will finally be handed over to Python for aggregation (4.2).
In the training phase, there are four termination conditions:

1. The specified number of epochs is completed.
2. The target mean training accuracy, specified at the beginning, is reached.
3. An error occurs due to all nodes being disconnected.
4. The stop training button is pressed.

Meanwhile, the master will save the model weights every 3 epochs in a backup file, which can be easily retrieved by the user. An additional check is made to ensure that the epoch of the received message is the correct one and not a previous one from a node that was too late to deliver the result.

```
load_nodes(ListsPidNodes, PythonModelPid, JavaUiPid) ->
    {Pids, _Nodes} = lists:unzip(ListsPidNodes),
    load_db(Pids, JavaUiPid, async),  % the acks sent will be discarded in master loop
    distribute_model(PythonModelPid, Pids, JavaUiPid, async),
    distribute_model_weights(PythonModelPid, Pids, JavaUiPid, sync).


load_db(Pids, _JavaUiPid, async) ->
    lists:foreach(fun(Pid) -> node_api:load_db(Pid) end, Pids),
    ok;

distribute_model(PythonModelPid, Pids, JavaUiPid, async) ->
    Model = message_primitives:synch_message(PythonModelPid, get_model, null, model_definition, JavaUiPid),
    lists:foreach(fun(Pid) -> node_api:initialize_model(Pid, Model) end, Pids),
    ok;

distribute_model_weights(PythonModelPid, Pids, JavaUiPid, async) ->
    Weights = message_primitives:synch_message(PythonModelPid, get_weights, null, model_weights, JavaUiPid),
    lists:foreach(fun(Pid) -> node_api:update_weights(Pid, Weights) end, Pids),
    ok;

distribute_model_weights(PythonModelPid, Pids, JavaUiPid, sync) ->
    distribute_model_weights(PythonModelPid, Pids, JavaUiPid, async),
    NewPids = message_primitives:wait_response(length(Pids), weights_ack, JavaUiPid),
    lists:map(fun(Pid) -> {Pid, erlang:node(Pid)} end, NewPids).
```

Figure 4.1: Load procedure pipeline

4.1.5 Node Discovery, Network Partitions, and Crashes

Every 10 seconds, the master server will call a function that tries to connect to every host specified in the *.erlang.hosts* file. Leveraging this file, we can also whitelist a set of trusted sources to communicate with (4.3). After a connection is made, since we enable node monitoring by *net_kernel:monitor_nodes(true)*, we will receive two different messages: {*nodeup, Node*} and {*nodedown, Node*}. Upon receiving a *nodeup* message, we know that a new node has been found or that an old node has reconnected. We then call the *reconnect* procedure to initialize or just load the node. The order guarantees of these messages can be looked up in the documentation.

In the server's state, two {*Pid, Node*} lists are kept: the list of *currentUpNodes*, used to identify participants in a future training task, and the *previousInitializedNodes*, which helps handle network partitions more efficiently. When recovering from a partition, the node is just loaded with the last system changes, rather than also initialized again.

The same procedure (*reconnect* (4.4)) is called after a crash when a 'DOWN' message is received. A small trick is used to avoid an infinite loop of crash-recovery events (4.5).

4.1.6 Handling Crashes and Disconnections During Blocking Wait

Instead of waiting for the timeout every time, if a node is disconnected or crashes during a blocking wait, we will dynamically adjust the number of responses required from the command to continue.

This approach leverages the fact that if a process is in the list of *currentUpNodes* and there is a 'DOWN' message in the mailbox associated with its PID, the node cannot respond, either because it has crashed and not been re-spawned or because the entire node is disconnected (4.6).

There is a third option: a node disconnected and reconnected during the same operation. In this case, if a response is returned from the given node, that response must be considered outdated and discarded, as other events could have occurred during the node's absence.

The code snippet also shows a particular way of handling timeouts, since the *node_metrics* message would interfere (it has a frequency of 1 message per second for each node). The *EndTime* is calculated as: *EndTime = erlang:monotonic_time(millisecond) + ?TIMEOUT*.

```
handle_cast({new_train, EpochsLeft, CurrentEpoch, AccuracyThreshold}, State) ->
    message_primitives:notify_ui(State#mstate.javaUiPid, {new_train, {EpochsLeft,AccuracyThreshold}}),
    gen_server:cast(erlang_master, {train, EpochsLeft, CurrentEpoch, AccuracyThreshold}),
    {noreply, State#mstate{terminateTraining = false}};   % clear the terminateTraining flag

handle_cast({train, EpochsLeft, CurrentEpoch, AccuracyThreshold}, State) when EpochsLeft > 0, CurrentEpoch ⩾ 0,
length(State#mstate.currentUpNodes) > 0 ->

    {PidNodes, _} = lists:unzip(State#mstate.currentUpNodes),
    {Nodes, TrainMeanAccuracy} = master_utils:train(CurrentEpoch, State#mstate.pythonModelPID, PidNodes,
State#mstate.javaUiPid),

    case {EpochsLeft > 1, AccuracyThreshold ⩾ TrainMeanAccuracy, length(Nodes) > 0,
State#mstate.terminateTraining } of
        {true, true, true, false} ->
           if
            CurrentEpoch rem 3 == 2 ->
                gen_server:cast(erlang_master, {save_model, backup}); % Call save_model every 3 epochs
            true -> ok
            end,
            gen_server:cast(erlang_master, {train, EpochsLeft - 1, CurrentEpoch + 1, AccuracyThreshold});
        {_, _, true, _} ->
            io:format("—— MASTER: training completed ——~n"),
            message_primitives:notify_ui(State#mstate.javaUiPid, {training_total_completed, TrainMeanAccuracy});
        {_, _, false, _} ->
            io:format("—— MASTER: training error ——~n"),
            message_primitives:notify_ui(State#mstate.javaUiPid, {training_error})
    end,

    {noreply, State};

handle_cast({train, _EpochsLeft, _CurrentEpoch, _AccuracyThreshold}, State) ->
    message_primitives:notify_ui(State#mstate.javaUiPid, {train_refused}),
    io:format("—— MASTER: training refused, possible causes: no nodes connected or illegal param ——~n"),
    {noreply, State};


train(CurrentEpoch, PythonModelPid, Nodes, JavaUiPid) ->
    Weights = message_primitives:synch_message(PythonModelPid, get_weights, null, model_weights, JavaUiPid),
    lists:foreach(fun(Pid) -> node_api:train_pipeline(Pid, Weights, CurrentEpoch) end, Nodes),
    ResponseList = message_primitives:wait_response(length(Nodes), {train_pipeline_ack, CurrentEpoch},
JavaUiPid, Nodes),

    case ResponseList of
        [] -> {[], 0.0};
        _ ->
            {PidList, Messages} = lists:unzip(ResponseList),
            {NewWeights, TrainSizes, Accuracy} = lists:unzip3(Messages),
            {TrainAccuracy, TestAccuracy} = lists:unzip(Accuracy),

            message_primitives:synch_message(PythonModelPid, update_weights, NewWeights, update_weights_ack,
JavaUiPid),
            TrainWeightedAccuracy = lists:sum(lists:zipwith(fun(Acc, Size) -> Acc * Size end, TrainAccuracy,
TrainSizes)) / lists:sum(TrainSizes),
            {PidList, TrainWeightedAccuracy}
    end.
```

Figure 4.2: Train procedure pipeline

4.1.7 Terminating Procedure of the OTP Master Server

During the termination procedure, which is started by the supervisor if the process crashes or the user interrupts it, the following actions will occur (4.7):

1. The UI will be notified about the possible termination or restart.
2. The Python model process will be terminated.
3. All processes connected to the master will be stopped.

```
get_cluster_nodes() ->
    Nodes = net_adm:world(),
    MasterNode = node(),
    lists:filter(fun(N) -> N =/= MasterNode end, Nodes).
```

Figure 4.3: Node discovery function, called every 10 seconds by master

```
initialize_node(State, Node) ->
    % Determine the new PID for the node
    case lists:keyfind(Node, 2, State#mstate.previousInitializedNodes) of
    {Pid, _Node} ->
        case master_utils:check_node_alive(Pid, Node) of
            true ->
                    io:format("--- MASTER: Node ~p server is still alive ---~n", [Node]),
                    erlang:monitor(process, Pid),   % monitor is deactivated when a node is disconnected
                    Pid;
            _ ->    io:format("--- MASTER: Node ~p server is dead, initializing ---~n", [Node]),
                    [{PidN, Node}] = network_helper:initialize_nodes([Node], State#mstate.javaUiPid),
                    PidN
        end;
    false -> io:format("--- MASTER: Node ~p is newly connected, initializing ---~n", [Node]),
        [{PidN, Node}] = network_helper:initialize_nodes([Node], State#mstate.javaUiPid),
        PidN
end.

load_node(State, Pid, Node) ->
    case master_utils:load_nodes([{Pid, Node}], State#mstate.pythonModelPID, State#mstate.javaUiPid) of
    [LoadedPidNodeNew] ->
        NewPidNodes = lists:keydelete(Node, 2, State#mstate.previousInitializedNodes),
        % remove to avoid two processes for the same node
        PidNodes1 = [LoadedPidNodeNew | State#mstate.currentUpNodes],
        PidNodes2 = [LoadedPidNodeNew | NewPidNodes],
        message_primitives:notify_ui(State#mstate.javaUiPid, {node_up, Node}),
    State#mstate{currentUpNodes = PidNodes1, previousInitializedNodes = PidNodes2};
    [] ->
        % List is empty, no PID was loaded -> meaning the node has an error on the initialization routine
        io:format("--- MASTER: Node ~p cannot be initialized ---~n", [Node]),
        message_primitives:flush_msg({nodeup, Node}),
        % avoid to retry the initialization if the error is in the startup routine
        State
    end.

reconnect(Node, State) ->
    try
        PidNew = initialize_node(State, Node),   % could throw badmatch if unable to load the node
        load_node(State, PidNew, Node)
    catch
        Error:Reason ->
            io:format("--- MASTER: Node ~p failed to initialize (~p:~p) ---~n", [Node, Error, Reason]),
            message_primitives:flush_msg({nodeup, Node}),
            State
    end.

flush_msg(Msg) ->
    receive
        Msg -> flush_msg(Msg)
    after 0 -> ok
    end.
```

Figure 4.4: Reconnect procedure to handle new nodes, reconnections, and restoring from crashes

```erlang
reconnect(Node, State, Pid, Reason, _MonitorRef) ->
    NewState = master_utils:reconnect(Node, State),
    message_primitives:flush_msg('DOWN', Pid, Reason),
    % avoid to retry the initialization if the error is in the startup routine
    NewState.

flush_msg('DOWN', Pid, Reason) ->  % flush DOWN msgs of processes that are on the same node of Pid
    receive
        {'DOWN', _MonitorRef, process, Pid2, Reason} when node(Pid) == node(Pid2) -> flush_msg('DOWN', Pid2,
Reason)
    after 0 -> ok
    end.
```

Figure 4.5: Reconnect procedure wrapper, called after a crash occurs

```erlang
wait_response(N, RespList, _, _, _EndTime, Crashed, _Pids) when N == 0 ->
    lists:foreach(fun(CrashedMsg) -> self() ! CrashedMsg end, Crashed),
    % retransmit exceptions for proper handling
    RespList;

% crashed is the list of msgs read but not handled, that need retrasmission
wait_response(N, RespList, AckCode, DestinationNodeMetrics, EndTime, Crashed, Pids) ->
    Now = erlang:monotonic_time(millisecond),
    TimeLeft = case EndTime - Now < 0 of
        true -> 0;
        false -> EndTime - Now
    end,

    receive
        {AckCode, _Pid, Response} when Pids == [] ->  % no Pids check required
            wait_response(N-1, RespList ++ [Response], AckCode, DestinationNodeMetrics, EndTime, Crashed, []);
        {AckCode, Pid, Response} ->
            case lists:member(Pid, Pids) of
            true ->
                NewPids = lists:delete(Pid, Pids),
                wait_response(N-1, RespList ++ [Response], AckCode, DestinationNodeMetrics, EndTime, Crashed,
NewPids);
            false ->
                wait_response(N, RespList, AckCode, DestinationNodeMetrics, EndTime, Crashed ++ [{AckCode, Pid,
Response}], Pids)
            end;
        {node_metrics, Metrics} ->   % has high frequency and will interfere with normal timeout handling
            DestinationNodeMetrics ! {node_metrics, Metrics},
            wait_response(N, RespList, AckCode, DestinationNodeMetrics, EndTime, Crashed, Pids);
        {'DOWN', _MonitorRef, process, Pid, Reason} when length(Pids) > 0 ->  % read only if Pids check enabled
            handleErrorMsg({'DOWN', _MonitorRef, process, Pid, Reason}, N, RespList, AckCode,
DestinationNodeMetrics, EndTime, Crashed, Pids)

    after TimeLeft ->
        io:format("-- ERROR: timeout occurs --~n"),
        lists:foreach(fun(CrashedMsg) -> self() ! CrashedMsg end, Crashed),  % retransmit exceptions
        RespList
    end.

handleErrorMsg(Msg, N, RespList, AckCode, DestinationNodeMetrics, EndTime, Crashed, Pids) ->
    {'DOWN', _MonitorRef, process, Pid, Reason} = Msg,
    case {lists:member(Pid, Pids), Reason == noconnection} of
        {true, false} ->
            io:format("-- WARNING: exception occurred during waiting the responses, Pid = ~p --~n", [Pid]),
            NewPids = lists:delete(Pid, Pids),
            wait_response(N-1, RespList, AckCode, DestinationNodeMetrics, EndTime, Crashed ++ [Msg], NewPids);
        {true, true} ->
            io:format("-- WARNING: nodedown occurred during waiting the responses, Pid = ~p --~n", [Pid]),
            NewPids = lists:delete(Pid, Pids),
            wait_response(N-1, RespList, AckCode, DestinationNodeMetrics, EndTime, Crashed, NewPids);
        {_, _} ->
            wait_response(N, RespList, AckCode, DestinationNodeMetrics, EndTime, Crashed ++ [Msg], Pids)
    end.
```

Figure 4.6: Waiting for responses from N nodes, while forwarding node metrics and listening for crashes or disconnections

```
terminate(_Reason, State) ->    % called from supervisor shutdown or stop function
    io:format("—— MASTER: Terminating Procedure ——~n"),
    lists:foreach(fun({Pid, _Node}) -> node_api:stop(Pid) end, State#mstate.currentUpNodes),
    python:stop(State#mstate.pythonModelPID),
    message_primitives:notify_ui(State#mstate.javaUiPid, {master_terminating, "possible restarting by
supervisor"}),
    ok.
```

Figure 4.7: Termination procedure of the master server

4.2 Master Python

The Erlang master node interfaces with a Python instance via Erlport. This Python process is responsible for defining the neural network model, aggregating the weights received from the worker nodes, and sending the aggregated weights back to the Erlang master node for further processing. Moreover, it handles model saving and loading from files. The module's core is a callback function that processes incoming messages from Erlang (4.8).

```
def handler(message):
        code, pid, payload = message
        code = code.decode('utf-8')

        match code:
            case "get_model":
                response = federatedController.get_definition()
                send_message(master_pid, encode_status_code("model_definition"), response)
            case "get_weights":
                response = federatedController.get_weights()
                send_message(master_pid, encode_status_code("model_weights"), response)
            case "update_weights":
                federatedController.update_weights(payload)
                send_message(master_pid, encode_status_code("update_weights_ack"), None)
            case "save_model":
                result = federatedController.save_model(payload.decode('utf-8'))
                send_message(master_pid, encode_status_code("model_saved"), encode_status_code(result))
            case "load_model":
                result = federatedController.load_model(payload.decode('utf-8'))
                send_message(master_pid, encode_status_code("model_loaded"), encode_status_code(result))
            case _:
                send_message(master_pid, encode_status_code("python_unhandled"), "Invalid message code " + code)

    setmessage_handler(handler)
    return (encode_status_code("ok"), "MODEL")
```

Figure 4.8: Setting up the callback function for the master Python process

4.2.1 Basic Model Definition

The basic structure of the model defined by the administrator must follow the format shown below. The admin can then modify certain aspects, such as the *sequential* definition or the *call* implementation (4.9).

```
class NetworkModel(tf.keras.Model):
    def __init__(self, sequential: tf.keras.Sequential):
        super().__init__()
        self.network: tf.keras.Sequential = sequential

    def call(self, x):
        network_output = self.network(x)
        # Possible to add specific behavior (eg. custom penality as in VAE)
        return network_output
```

Figure 4.9: Basic structure of the *NetworkModel* object

4.2.2 Marshalling and Unmarshalling of Data

To efficiently pass model data across nodes, we use the *pickle* module. Pickle serializes our model object into a byte array, which can be easily recovered and deserialized on another Python instance. This makes *pickle* the most efficient and straightforward way to transport TensorFlow objects (4.10).

```python
def get_definition(self) -> str:
        return pickle.dumps(self.model.network.get_config())

def get_weights(self) -> str:
    return pickle.dumps(self.model.get_weights())

def update_weights(self, node_outputs: list):
    node_outputs = [pickle.loads(output) for output in node_outputs]
    new_weights = FederatedController.federated_weight_average(node_outputs)
    self.model.set_weights(new_weights)

@staticmethod
def federated_weight_average(node_outputs) -> list:
    """
    Args:
        node_outputs: A list of tuples, containing the node weights and dataset size.
    """
    n_nodes = len(node_outputs)
    if n_nodes == 0:
        raise ValueError("No outputs to average")

    # Calculate total dataset size
    total_size = sum(output[1] for output in node_outputs)

    # Get the structure of weights from first node
    first_weights = node_outputs[0][0]
    averaged_weights = [np.zeros_like(np.array(w)) for w in first_weights]

    # Average weights across nodes
    for node_output in node_outputs:
        weight = node_output[1] / total_size
        node_weights = node_output[0]

        # Add weighted contribution from this node
        for layer_idx, layer_weights in enumerate(node_weights):
            averaged_weights[layer_idx] += np.array(layer_weights) * weight

    return averaged_weights
```

Figure 4.10: Pickle use and FedAvg implementation

4.2.3 Erlang Communication - Erlport

The Erlang master starts this Python process. A call is then made to the *register_handler* function to set the message callback. To enable seamless communication between these two languages, we use the *erlport* module.

Erlport is a library designed to enable communication between Python and Erlang (with Ruby also supported). It allows inter-language function calls, message passing, and type mapping.

When we call the *start* function from Erlang, the Erlport library is delegated the task of starting a new OS process and connecting it to the current Erlang instance via a port. For every function call or message sent, a new message is created and sent to the port. The language-specific Erlport library on the other side of the port processes the message by invoking the appropriate function or message handler.

At the end of the session, when the *stop* function is called, the process is terminated, and the port is closed.

4.3 Erlang Nodes

The Erlang worker nodes are distributed processes that communicate directly with the Erlang master node. Their primary role is to transmit commands received from the master to their respective local Python nodes. These nodes are also responsible for error handling and mitigating issues caused by network partitions.

4.3.1 Handling Network Partitions and Process Leaks

A process spawned by the master server could leak if a network partition occurs and, in the meantime, the master terminates or crashes. The solution is to set a timer that, in case the node and the master are disconnected for too long, will terminate the slave process. When the master node is reconnected, the timer is canceled. Before a node terminates, it will also kill the Python process (4.11).

```erlang
% the process will terminate if the master node is down for more than TIMEOUT milliseconds
handle_info({nodedown, MasterNode}, State) when MasterNode == State#nstate.masterNode ->
    io:format("—— NODE ~p: MASTER disconnected ——~n", [MasterNode]),
    TimerRef = erlang:send_after(?TIMEOUT, self(), terminate),  % default: 5 minutes
    {noreply, State#nstate{termination_timer = TimerRef}};

handle_info(terminate, State) ->
    {stop, normal, State};

handle_info({nodeup, MasterNode}, State) when MasterNode == State#nstate.masterNode ->
    io:format("—— NODE ~p: MASTER node reconnected ——~n", [node()]),
    erlang:cancel_timer(State#nstate.termination_timer),
    {noreply, State};

terminate(Reason, State) ->
    io:format("—— NODE ~p: Terminating with reason: ~p ——~n", [node(), Reason]),
    python:stop(State#nstate.pythonPid),  % abort python process
    ok.
```

Figure 4.11: Network Partitions Handling

4.4 Python Worker Nodes

The Python worker nodes are responsible for performing federated training tasks. Each Python node loads the local dataset for training, loads the model configuration and weights provided by the master node, trains the model locally, and finally shares the updated weights with the master node for aggregation.

4.4.1 *load_db* and *train_pipeline* Outputs

The *load_db* command produces an output of the form {x_train_shape, x_test_shape}, which is useful for updating the UI accordingly. The *train_pipeline* response has the form {weights, train_accuracy, test_accuracy}. The corresponding Erlang process then adds the *CurrentEpoch* variable to the message, enabling the Erlang master to easily discard late responses.

4.4.2 Node Metrics

The Python node is also responsible for repeatedly sending node metrics such as CPU and RAM utilization, and the response time to the master. The output is sent in JSON format, as the JS script needs to handle it. This is not a significant overhead, as the metrics messages are compact compared to TensorFlow objects (4.12).

```python
def start_metrics_thread():
    metrics_thread = threading.Thread(target=send_system_metrics)
    metrics_thread.daemon = True   # terminates when the python process terminates
    metrics_thread.start()

def send_system_metrics():
    while True:
        cpu_usage = psutil.cpu_percent(interval=1)  # CPU usage in percentage
        memory_info = psutil.virtual_memory()
        memory_usage = memory_info.percent  # Memory usage in percentage

        response_time = ping(nodeController.master_ip, timeout=2)
        response_time_ms = response_time * 1000 if response_time is not None else None  # ms

        node_details = {
            "Node": nodeController.node_id,
            "CPU": f"{cpu_usage}%",
            "Memory": f"{memory_usage}%",
            "Response Time": f"{response_time_ms:.2f} ms" if response_time_ms else "Unreachable",
        }

        cast(nodeController.erlang_pid, (encode_status_code("node_metrics"), json.dumps(node_details)))
        time.sleep(1)
```

Figure 4.12: Metrics Payload Generation in Python Node

4.5 Java Back-End

This layer serves as a bridge between the client application and the Erlang Master Node. It processes incoming requests from the client and forwards them to the Erlang system using JInterface. Additionally, it routes training results and system status updates back to the client. The approach uses two Consumer-Producer patterns, handled by *BlockingQueue* objects.

4.5.1 Spring Boot

Spring Boot simplifies the setup and configuration process, providing a convention-over-configuration approach that reduces the need for boilerplate code. Spring Boot comes with an embedded server, eliminating the need for an external server setup. This self-contained nature of Spring Boot makes it more flexible, enabling quicker deployment, improved productivity, and easier integration with various Spring ecosystem tools, like spring-security.

4.5.2 General Structure

In our Java application, there are two always-running threads. Their lifecycles are directly tied to the lifecycle of Spring Boot beans.

The first one, ***WebSocketForwarder***, is responsible for taking an Erlang message (blocking semantics) from its *BlockingQueue* and broadcasting it to all active sessions (4.13).

The ***ErlangController*** thread performs the opposite function: it takes a WebSocket message from its queue and executes the corresponding command, obtained by a *CommandFactory* method. This call must be non-blocking, as it would otherwise interfere with the thread's other task of taking an Erlang message and placing it inside its designated queue (4.14).

The WebSocket message queue is populated by ***ErlangWebSocketHandler*** that will be examined in the following sections.

```java
@Override
public void run() {
    while (!Thread.currentThread().isInterrupted()) {
        try {

            String erlangMessage = queues.getErlangMessage();   // blocking call

            if (!erlangMessage.startsWith("{node_metrics"))
                log.info("[WebSocket] Send erlang message to active sessions {}", erlangMessage);

            // Broadcast to all active WebSocket sessions
            sessionRegistry.broadcastMessage(erlangMessage);

        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();   // restore the flag
            log.error("[WebSocket] Thread interrupted during take()");
        } catch (IOException e) {
            log.error("[WebSocket] Error sending Erlang message: {}", e.getMessage());
        }
    }
}
```

Figure 4.13: WebSocketForwarder body

```java
@Override
public void run() {
    while (!Thread.currentThread().isInterrupted()) {
        try {
            receiveErlangMessage();
        } catch (InterruptedException e) {
            log.error("ErlangController interrupted during addErlangMessage(msg)");
            Thread.currentThread().interrupt();
        }
        executeWebSocketCommand();
    }
}

private void receiveErlangMessage() throws InterruptedException {
    try {
        String msg = erlangContext.getNextMessage();     // non-blocking

        if (msg != null && !msg.startsWith("{rex,")) {   // RPC return value discarded
            queues.addErlangMessage(msg);

            if (finishedTrainCode.stream().anyMatch(code -> msg.startsWith("{" +
code)))          erlangContext.setTraining(false);
        }

    } catch (OtpAuthException | OtpErlangExit | IOException e) {
        log.error("Error message discarded from Erlang, reason: {}", e.getMessage());
    }
}

private void executeWebSocketCommand() {
    String commandJSON = queues.getWebSocketMessage();
    if (commandJSON == null)
        return;

    try {
        Map<String, String> commandMap = parseCommand(commandJSON);
        Command command = commandFactory.createCommand(commandMap.get("command"));
        command.execute(commandMap.get("parameters"));
    } catch (RuntimeException e) {
        log.error("Error executing erlang command: {}", e.getMessage());
    }
}
```

Figure 4.14: ErlangController body

4.5.3 Authentication

In our application, we use cookies to manage authentication. When a user logs in, their credentials are validated, and a session is created. This session is then stored in a cookie, allowing the user to remain authenticated across different requests.

The Spring Boot *securityFilterChain* is a filter applied after every HTTP request to the server to prevent unauthorized access (4.15). We have not implemented a registration mechanism since no particular data needs to be saved. Instead, we provided the system with two users with different roles: an admin who can issue Erlang commands, and a user who is just a spectator. These accounts can have multiple active sessions simultaneously, enabling the system to support multiple spectators or administrators connected under the same profile.

```java
@Bean
public UserDetailsService users(@Value("${security.user.password}") String userPassword,
                                @Value("${security.admin.password}") String adminPassword) {
    UserDetails user = User.builder()
            .username("user")
            .password(passwordEncoder().encode(userPassword))
            .roles("USER")
            .build();
    UserDetails admin = User.builder()
            .username("admin")
            .password(passwordEncoder().encode(adminPassword))
            .roles("ADMIN")
            .build();
    return new InMemoryUserDetailsManager(user, admin);
}

@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
            .authorizeHttpRequests(auth → auth
                    .requestMatchers("/login").permitAll() // Allow access to log in
                    .anyRequest().authenticated() // Authentication for every other request
            )
            .formLogin(withDefaults()) // Use the default Spring Security login page
            .httpBasic(withDefaults()) // Authorization header encoded in b64
            .sessionManagement(session → session
                    .sessionCreationPolicy(SessionCreationPolicy.IF_REQUIRED)
                    .invalidSessionUrl("/login")  // redirect when cookie expires
            );

    return http.build();
}
```

Figure 4.15: HTTP Authentication filter

4.5.4 Websocket (SockJS)

Using WebSocket to develop a live dashboard is ideal for applications requiring real-time updates and interactive features. WebSocket provides a full-duplex communication channel between the client and server, enabling instant data exchange without the overhead of repeated HTTP requests.

We used SockJS in our application to enhance the reliability of WebSocket connections. SockJS leverages the initial HTTP handshake for connection upgrades, enabling us to integrate the existing HTTP authentication mechanisms directly. By using HTTP cookies during the handshake, we can authorize users securely before the connection is upgraded. Moreover, we will retrieve the user role to authenticate them and handle the two use cases described earlier (4.16).

```java
@Override
public void afterConnectionEstablished(@NonNull WebSocketSession session) {

    if (erlangContext.isConnected()) {
        try {
            queues.restoreSession(session);   // show current state of executions to new client
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            log.error("Session restoration interrupted");
            throw new RuntimeException("Session restoration interrupted", e);
        } catch (RuntimeException e) {
            log.error("Error restoring session {}", e.getMessage());
            return;
        }
    }

    sessionRegistry.addSession(session);
    String role = getRoleString(session);

    if (role != null && role.equals("ADMIN")) {
        try {
            session.sendMessage(new TextMessage("{operator}"));  // show buttons on the UI
        } catch (IOException e) {
            log.error("Error enabling admin command in UI: {}", e.getMessage());
            throw new RuntimeException(e);
        }
    }

    log.info("[SockJS] New session connected: {}", session.getId());

}

@Override
protected void handleTextMessage(@NonNull WebSocketSession session, TextMessage message) throws Exception {

    String role = getRoleString(session);

    if (role == null || !role.equals("ADMIN")) {
        log.warn("[SockJS] Received unauthorized message: {}", message.getPayload());
        return;
    }

    String receivedMessage = message.getPayload();
    queues.addWebSocketMessage(receivedMessage);
    log.info("[SockJS] Received message: {}", receivedMessage);

}

private static String getRoleString(WebSocketSession session) {
    SecurityContext securityContext = (SecurityContext)
session.getAttributes().get("SPRING_SECURITY_CONTEXT");
    Authentication authentication = securityContext.getAuthentication();

    return authentication == null ? null : authentication.getAuthorities().stream()
            .map(grantedAuthority -> grantedAuthority.getAuthority().replace("ROLE_", ""))
            .findFirst()
            .orElse(null);

}
```

Figure 4.16: WebSocket controller

4.5.5 Session Restoration

We handled the challenge of updating the new connections to the current computation state (e.g., current up nodes, or training history) in the following way (4.17). We keep a queue for all the relevant UI messages, called *cacheSession*. When the client finishes processing all these messages, its UI will be the same as that of a client present from the beginning.

In particular, when a new client connects to the WS and the Erlang session is active, we wait at most for *timeoutMillis* for the *erlangQueue* to be empty. If that queue is not empty at the moment of restoring the session, some messages could be received twice by the new client, since they would be present in both *erlangQueue* and *cacheSession*. Note the *synchronized* keyword makes it impossible to add new messages while trying to restore the session. Once this is completed, the messages are sent to the new client and the session is successfully restored.

This solution isn't the most efficient in terms of space and load on the client. However, we cleared all the unnecessary train logs at each train execution. This reduces almost entirely the redundant messages that

the client needs to process (just reconnection of nodes).

```java
private final BlockingQueue<String> cacheSession = new LinkedBlockingQueue<>();
private final List<String> cacheMsgCodes = List.of("initialized_nodes", "train",
        "node_up", "node_down", "db_ack", "stopped");

public synchronized void addErlangMessage(String message) throws InterruptedException {
    erlangQueue.put(message);

    if (cacheMsgCodes.stream().anyMatch(code -> message.startsWith("{" + code)))
        cacheSession.put(message);

    if(message.startsWith("{new_train")) {   // clear all old train messages
        cacheSession.removeIf(msg -> msg.startsWith("{train"));
        cacheSession.put(message);
    }
}

public synchronized void restoreSession(WebSocketSession session) throws RuntimeException, InterruptedException
{   long startTime = System.currentTimeMillis();
    long timeoutMillis = 500;
    // Wait for erlangQueue to empty otherwise could have some message duplication on the new client
    // no new msgs can be added when performing restoreSession
    while (!erlangQueue.isEmpty()) {
        if (System.currentTimeMillis() - startTime > timeoutMillis)
            throw new RuntimeException("Timeout waiting for erlang queue to empty");
        Thread.sleep(5);
    }

    cacheSession.forEach(msg -> {
        try {
            session.sendMessage(new TextMessage(msg));
        } catch (IOException e) {
            throw new RuntimeException(e);
        } });
}

public synchronized void clearCache() {   // called during Stop command
    cacheSession.clear();
    try {
        addErlangMessage("{stopped}");   // clear the UI
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        throw new RuntimeException(e);
    }
}
```

Figure 4.17: Cache Handling

4.5.6 Erlang Start

Once the Start command is issued on the UI, the Java application will start a new *rebar3* process with the cookie and name specified in the *application.properties* file. Then its output is redirected both to the application console and to a specified file for logging purposes. After the process is correctly initialized, we can use the *JInterface* module to establish a connection between Java and the newly created Erlang node. After a connection is established, we can proceed with the initialization of the master supervisor that will initialize the master server, which in turn will initialize and load the connected nodes (4.18).

How JInterface Works

The JInterface package provides a set of tools for enabling communication between Java and Erlang. In particular, it handles the type mapping between the two languages and the processes communication.

The class *OtpPeer* represents an Erlang node. When *OtpSelf.connect(OtpPeer)* is used to connect to an Erlang node, a connection is first made to epmd and, if the node is known, a connection is then made to the Erlang node, connecting to the Host IP via the socket port specified by epmd.

```java
public static Process startErlangNode(String path, String cookie, String name, long timeout) throws
InterruptedException, IOException {
      ProcessBuilder builder = new ProcessBuilder("rebar3", "shell", "--sname", name, "--setcookie", cookie);
      builder.directory(new File(path));
      builder.environment().put("TF_CPP_MIN_LOG_LEVEL", "3");  // disable tf warnings

      File logFile = new File("logs/erlang.log");
      builder.redirectErrorStream(true); // Merge stdout and stderr
      Process process = builder.start();

      new Thread(() -> {  // Start background thread to handle output
          try (BufferedReader reader = new BufferedReader(new InputStreamReader(process.getInputStream()));
               FileWriter writer = new FileWriter(logFile, true)) {
              String line;
              while (process.isAlive() && (line = reader.readLine()) != null) {
                  writer.write(line + "\n");
                  writer.flush();
                  System.out.println(line);
              }
          } catch (IOException e) {...}
      }).start();

      Thread.sleep(timeout);
      if (!process.isAlive()) {
          process.destroyForcibly();
          throw new RuntimeException("[Java] Erlang node did not start correctly");
      }

      return process;
}

private void createConnection(ErlangContext context) {
      try {
          OtpSelf self = new OtpSelf(javaNodeName, cookie);
          OtpPeer master = new OtpPeer(erlangNodeName);
          context.setOtpConnection(self.connect(master));
          context.setJavaPid(self.pid());
      } catch (IOException | OtpAuthException e) {
          context.getErlangProcess().destroyForcibly();
          sendErrorMessage();
          throw new RuntimeException("Cannot start correctly the connection with erlang", e);
      }
}

ErlangHelper.call(context.getOtpConnection(), new OtpErlangObject[]{context.getJavaPid()}, "master_supervisor",
"start_link_shell");
```

Figure 4.18: Erlang Start

4.5.7 Handling Multiple Application Instances

The application supports multiple instances of the Java application running on the same machine or on different machines. If running on the same machine, it is important to change the server port and the master and Java node names. All this information is stored in the *application.properties* file.

4.5.8 Logging

Logging is performed using the default log framework for Spring Boot: *SLF4J*. It is also saved to a file, whose location is specified in the *application.properties* file. Spring Boot automatically appends to that file the console log, and each day it saves the resulting log inside an archive and cleans the file.

4.6 Client Application

The client application is developed using HTML, CSS, and JavaScript, leveraging the *Chart.js* library to dynamically display the training accuracy graph. The login page uses the default Spring Security page.

User Roles

After logging in, the interface adapts based on the user's role. If the login is performed by an admin, buttons for executing various operations are displayed, whereas standard users do not see these options. Apart from this distinction, the two screens are identical, serving the purpose of monitoring activity within the network.

Homepage Screen

The first screen is the Homepage, which includes a dashboard displaying statistics for each active node: CPU and memory usage, response time, training and testing accuracy, and the sizes of the training and testing datasets. It also features a console that shows the main messages sent and received via WebSocket between the front-end and back-end, along with buttons for sending messages to the back-end to perform various operations based on the user's actions.

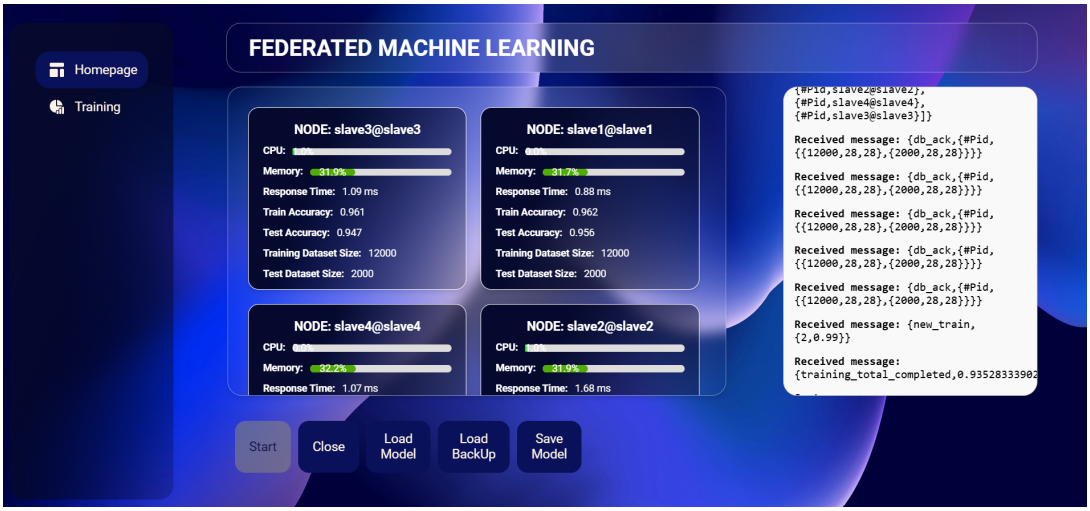


Figure 4.19: UI Administrator

Training Screen

The second screen is dedicated to Training. After clicking the "Start Training" button, a graph shows the average training and testing accuracy of the nodes for each epoch.

WebSocket and Message Processing

Communication between the client and the backend is established via WebSockets, enabling bidirectional communication, specifically using SockJS as mentioned earlier 4.5.4.
When a message is received, the function *processingInput(event.data)* is used, which analyzes the message and is responsible for updating the frontend. Here is how the JavaScript part for communication with WebSocket was implemented.

```
// Connection to the backend using SockJS after log-in
socket = new SockJS(url);

// Event handler when a message is received
socket.onmessage = function (event) {
    processingInput(event.data);
};

socket.onclose = function () {
    console.log('SockJS connection closed');
    addLogMessage("received", "SockJS connection closed");
    uiDisabled();
    const container = document.getElementById('containerNodes');
    const oldElements = container.querySelectorAll('.elemNode');
    oldElements.forEach(elem => elem.remove());
    clearChart();
};
```

Figure 4.20: SockJS connection from client side

4.7 Test of the Federated Learned Model

For the system test, we set up our network with 5 different nodes. We partitioned the MNIST dataset into 5 chunks and loaded each node with a different chunk.

$$\bigcup_{i=1}^{5} C_i = D, \quad C_i \cap C_j = \emptyset, \quad \forall i \neq j, \quad |C_i| = \frac{|D|}{5}, \quad \forall i = 1, 2, 3, 4, 5$$

We trained the model with the following options and save the Keras model.

```
model.compile(optimizer="adam", loss="categorical_crossentropy", metrics=["accuracy"])

train_result = model.fit(
    x_train, y_train,
    validation_data=(x_test, y_test),
    epochs=20,
    batch_size=32,
    verbose=1
)
```

Figure 4.21: System test parameters for training

In the meantime, we wrote a Python script that takes the entire MNIST dataset and trains the same model using the same modalities. We then saved this model as well.
Afterward, we validated both models using a test dataset that was not used in either of the previous trainings and compared the results.

Centralized training results	0.9878
HFL training results	0.9859

Table 4.1: Validation accuracy of the two training methodologies

We can conclude that the HFL approach in our application, at least for the MNIST dataset, produces results similar to the centralized training of the model.