Federated Learning Application Architecture Overview

1. System Architecture

This codebase implements a federated learning system with a client-server architecture in Rust. The system allows multiple client devices to collaboratively train a machine learning model while keeping their training data local.

2. File Structure and Functions

main.rs - Application Entry Point

Purpose: Serves as the launcher for either client or server mode

- Functions:
- main(): Parses command-line arguments and spawns either client or server process
- CommandLineArgs struct: Defines CLI parameters (--server, --client, --server-addr)

server.rs - Federated Learning Server Implementation

Purpose: Coordinates the federated learning process across clients

- Key Components:
- FederatedServer struct: Maintains client state and model data
- MODEL_NAME constant: Defines the name of the model (mnist)
- Client Management Methods:
- register(): Adds new clients to the system
- mark_ready(): Updates client status when ready for training
- remove_client(): Handles client disconnection
- Model Management Methods:
- init(): Initializes LinearModel and loads evaluation dataset
- get_model(): Returns reference to the current model
- aggregate_updates(): Implements federated averaging algorithm
- Training Coordination:
- train(): Orchestrates training rounds across selected clients

client.rs - Federated Learning Client Implementation

Purpose: Performs local training on client devices

- Key Components:
- FederatedClient struct: Maintains client state with:
- coordinator_address: Server address
- local_model: Current model instance, parameters and status
- training_data: Local subset of MNIST dataset
- client_endpoint: Client's listening address
- Connection Methods:
- new(): Creates client with random subset of MNIST data
- join_federation(): Connects to server and registers as participant
- Training Methods:
- train_local_model(): Trains model locally using SGD optimizer
- get(): Returns current model weights, biases and status
- test(): Evaluates model accuracy on test data
- Communication Handlers:
- run_inner(): Main loop handling server commands via TCP

3. Communication Protocol

The system implements a text-based protocol with pipe (|) delimiters:

Server to Client:

- TRAIN|model_name|weights_base64|bias_base64|epochs: Request to train model
- GET|model_name: Request to get current model
- TEST|model_name: Request to evaluate model
- COMPLETE: Notification that training is complete

Client to Server:

- REGISTER|client_address: Register with server
- READY: Indicate readiness for training
- UPDATE|weights_base64|bias_base64: Send model updates
- MODEL|weights_base64|bias_base64|status: Response to GET request
- ACCURACY|value: Response to TEST request

4. Machine Learning Components

- Uses candle_core and candle_nn libraries for ML operations
- Implements a linear model for MNIST classification
- Server aggregates model updates via federated averaging
- Client uses SGD optimizer for local training
- Model weights and biases encoded in base64 for transmission

5. Execution Flow

1. User launches application via *main.rs* with --server or --client flag

2. Server initializes and waits for client connections

3. Clients load a random subset of MNIST data and connect to server

4. Each client:

- Registers with server and marks itself as ready
- Sets up a TCP listener for receiving commands
- Processes training requests by updating local model
- Trains locally using SGD optimizer
- Reports model updates back to server

5. Server:

- Tracks available clients
- Selects subset of clients for training rounds
- Distributes current model to selected clients
- Aggregates client updates using federated averaging
- Evaluates model accuracy on test dataset

This architecture enables distributed machine learning across multiple clients while maintaining data privacy since raw training data never leaves the clients.