In this document, I'll provide a comprehensive, file-by-file technical explanation of my temporal hand gesture recognition system. I'll cover the mathematical foundations, machine learning algorithms, and software engineering principles I used to build the project.

Core Concepts

Before dissecting each file, I should explain the core concepts that govern the system.

1. **Temporal Convolutional Network (TCN)**: I chose a TCN for the model, a specialized neural network architecture designed for sequence data. Unlike RNNs, TCNs use causal, dilated convolutions to process sequences in parallel, which made them a computationally efficient choice for capturing long-range dependencies in the gesture data.

2. **Hybrid Architecture (Python + C)**: I designed a hybrid architecture to intentionally separate high-level orchestration (Python) from performance-critical computation (C). I used Python's rich ecosystem for the GUI and data handling, while I wrote the ML engine in C to be minimal, high-performance, and memory-efficient, making it suitable for my embedded target.

3. **Static Memory Allocation**: For embedded readiness, I used 100% static memory allocation in the C backend. I allocate all required memory for the model, its gradients, and optimizer states at compile time. This eliminates the risk of memory fragmentation or allocation failures on a resource-constrained device.

4. **Consistent Data Pipeline**: This was the most critical concept for achieving high accuracy. The normalization and temporal sampling methods I used during inference had to be mathematically identical to those used during training. To prevent discrepancies, I implemented a shared normalization function and a synchronized temporal stride for both training data augmentation and real-time inference.

C Backend: The Machine Learning Engine

I built the C backend as a self-contained, high-performance ML engine. It's composed of two executables: `train_c` for training and `ra8d1_sim` for inference.

`RA8D1_Simulation/training_logic.h`

This header file is the data blueprint for my entire C engine. Here, I defined the core data structures and constants.

- **`Model` vs. `InferenceModel`**: I defined two key structs. `Model` is a comprehensive struct containing everything needed for training: weights, biases, gradients, and Adam optimizer states.

`InferenceModel` is a lean version with only the weights and biases necessary for a forward pass, which minimizes the memory footprint for the inference engine.
- **Static Allocation**: All struct members are statically-sized arrays (e.g., `float weights[SIZE]`). This was the foundation of my static memory strategy.
- **`static_assert`**: I added a compile-time assertion, `static_assert(sizeof(Model) < APP_SRAM_LIMIT, ...)` to ensure the `Model` struct never exceeds the 1MB SRAM budget of the Renesas RA8D1. This was a critical safeguard for my embedded development workflow.

`RA8D1_Simulation/training_logic.c`

This file is the mathematical core of the project, where I implemented the TCN and its training algorithms from scratch.

- **`initialize_weights`**: I implemented **He Initialization**, drawing weights from a normal distribution with a standard deviation of `sqrt(2.0 / fan_in)`. This is a standard practice for networks using ReLU, and it helped prevent gradients from vanishing or exploding during training.
- **`forward_pass`**: Here, I execute the TCN's forward pass. It performs a **causal convolution**, followed by a **Leaky ReLU** activation (`max(0.01*x, x)`) to prevent the "dying ReLU" problem. Finally, I apply **Global Average Pooling** across the time dimension to create a fixed-size feature vector for the output layer.
- **`backward_pass`**: My implementation of backpropagation through time for the TCN. It calculates the gradients of the loss function with respect to the weights and biases of the entire network.
- **`update_weights`**: My from-scratch implementation of the **Adam Optimizer**. It updates the model's weights using the calculated gradients and the optimizer's internal state, including the necessary bias correction.
- **`load_temporal_data`**: Here, I implemented **temporal data augmentation**. My function reads sequences from CSV files and generates multiple, overlapping 20-frame windows using a `WINDOW_STRIDE` of 5. This dramatically increased the size and diversity of my training dataset.
- **`save_model` / `load_inference_model`**: I wrote these functions to perform safe, portable model serialization by writing and reading each array field-by-field. This approach avoids C struct padding and alignment issues that would otherwise make the binary model file non-portable.

`RA8D1_Simulation/train_in_c.c`

This is the main entry point for my training executable, which orchestrates the entire training process.

- **Hyperparameters**: I defined all key training parameters (`NUM_EPOCHS`, `LEARNING_RATE`, etc.) as constants at the top of the file for easy tuning.
- **Training Loop**: In the `main` function, I created the primary training loop that iterates for a fixed number of epochs, running a training and validation phase in each.
- **Loss Calculation**: I wrote the `calculate_loss` function to compute the **Cross-Entropy Loss**, which is standard for multi-class classification problems.

`RA8D1_Simulation/main.c`

This is my inference server, which I designed for real-time performance and robust communication.

- **TCP Server**: I implemented a persistent, single-client TCP server that listens on port `65432`.
- **Length-Prefix Protocol**: To handle TCP's stream-based nature, I designed a simple protocol where every message is prefixed with a 4-byte unsigned integer specifying the payload length. The server reads this header first to ensure it receives a complete data frame.
- **Network Byte Order**: I made sure the server correctly converts the incoming byte stream from network byte order to the host system's byte order using `ntohl`. This was critical for cross-platform compatibility with the Python client.
- **Model Loading and Lifecycle**: The server attempts to load `c_model.bin` only once at startup. Because it doesn't automatically reload, I made the Python GUI responsible for restarting this server process after training to force it to load the new model file.

Python GUI: The System Orchestrator

I built the Python application with PyQt6 to manage the user workflow, handle data processing, and communicate with the C backend.

`gui_app/logic.py`

This file contains the core business logic, connecting the UI to the underlying data processing and communication protocols.

- **`normalize_landmarks`**: This function is the heart of my Python data pipeline. I implemented a two-step normalization on the raw landmark data from MediaPipe:
  1. **Translation Invariance**: I subtract the wrist's coordinates from all other landmarks, making the gesture independent of its position in the camera frame.
  2. **Scale Invariance**: I calculate the average distance of all landmarks from the new origin (the wrist) and divide all coordinates by this scale factor. This makes the gesture robust to changes in hand size or distance from the camera.
- **`HandTracker` Class**: I created this as a wrapper around the MediaPipe library to process video frames, detect hand landmarks, and pass the raw data to my `normalize_landmarks` function.
- **`GesturePredictor` Class**: This class manages all communication with the C inference server.
  - **Persistent Connection**: I implemented logic to establish and maintain a persistent TCP connection, with automatic reconnection in case of errors.
  - **Data Buffering**: It maintains a `sequence_buffer` that collects the last 20 normalized landmark frames.
  - **Stride-Based Prediction**: This was my key to ensuring pipeline consistency. I only request predictions from the server every 5 frames (`window_stride`). On frames in between, it

returns the last known prediction. This perfectly mirrors the data augmentation I used during training.

- **Binary Packing**: I used `struct.pack` with network byte order (`'!'`) to pack data into a binary stream, matching the C server’s expectations.

`gui_app/main_app.py` (and other page files)

These files define the PyQt6 UI. They display the camera feed, render predictions, and connect UI buttons to my backend logic. The `training_page.py` module has the key responsibility of restarting the C inference server after a training run to ensure the new model is loaded.