Technical Challenges and Solutions

This document provides a technical log of the significant challenges encountered during the development of the temporal hand gesture recognition system, detailing the diagnostic processes and final resolutions.

---

1. TCN Training Stability: Resolving `NaN` Loss

- **Problem:** The C-based training process for the Temporal Convolutional Network (TCN) was unstable, frequently resulting in `NaN` (Not a Number) loss values, which halted model convergence.

- **Investigation and Resolution:**
  1. **Gradient Clipping:** Initial implementation of gradient clipping, a standard technique to mitigate exploding gradients, did not resolve the issue.
  2. **Backpropagation Logic Review:** A manual audit of the backpropagation-through-time implementation revealed minor issues but did not identify the root cause of the instability.
  3. **Root Cause Analysis:** Extensive diagnostic logging identified the “dying ReLU” problem. The standard ReLU activation function was causing neuron gradients to become zero, preventing gradient flow and leading to weight explosion in the remaining active neurons.
  4. **Solution:** A two-part solution was implemented. The ReLU activation was replaced with **Leaky ReLU** to ensure a persistent gradient flow. Additionally, the SGD optimizer was replaced with the **Adam optimizer** to provide adaptive learning rates, which significantly improved training stability.

---

2. System Stability: GUI-Backend Communication Protocol

- **Problem:** The system suffered from severe stability issues during real-time inference, including C server crashes ("Broken pipe” errors) and Python data parsing failures.

- **Investigation and Resolution:**
  1. **Connection Management:** Logging revealed that the Python client was establishing a new TCP connection for each inference frame, overwhelming the server. The protocol was re-architected to use a **single, persistent TCP connection** for the entire session.
  2. **Data Stream Handling:** Data corruption was traced to a TCP streaming issue where multiple server responses were being read simultaneously by the client. This was resolved by implementing a **buffered reader** (`socket.makefile('r')`) in the Python client to correctly parse newline-terminated responses.

3. **Process Management:** To prevent startup failures from zombie processes, the launch script was updated to **automatically detect and terminate** any processes occupying the required network port.

---

3. Model Optimization and Validation

- **Problem:** The initial model was prone to overfitting and exceeded the memory constraints of the target MCU. It also required a formal sanity check to validate that it was learning genuine data patterns.

- **Investigation and Resolution:**
  1. **Model Miniaturization:** The TCN channel count was iteratively reduced from eight to an **ultra-lean two-channel architecture**. This successfully reduced the memory footprint to <1MB while maintaining >98% accuracy.
  2. **Sanity Check Implementation:** To validate learning, a sanity check was performed by shuffling the data-label correlations. To avoid memory corruption, this was implemented by shuffling the label array within the `load_temporal_data` function in C. The resulting near-random accuracy (~33%) confirmed the model was learning from features, not noise.

---

4. Final Pipeline Synchronization: Achieving Production-Ready Inference

- **Problem:** Despite high training accuracy, real-world inference performance was extremely poor, indicating a critical disconnect between the training and inference data pipelines.

- **Investigation and Resolution:** A systematic, multi-stage debugging process identified and resolved several fundamental flaws:
  1. **Double Normalization:** The training data was being normalized twice, while inference data was normalized only once. **Solution:** The redundant normalization step was removed from the data-saving pipeline.
  2. **Temporal Sampling Mismatch:** Training data was sampled with a stride of 5, whereas inference was performed on every frame (stride of 1). **Solution:** The inference logic was updated to sample frames with a stride of 5, matching the training configuration.
  3. **Data Integrity:** Several data corruption issues were identified:
     - **Byte Order:** The C server was not correctly handling network-to-host byte order conversion for incoming float data. **Solution:** Implemented proper byte order conversion.
     - **Model Serialization:** The `save_model` function was using an incorrect struct size (`sizeof(InferenceModel)` instead of `sizeof(Model)`), leading to a corrupted model file. **Solution:** Corrected the `sizeof` reference.

        - **Model Deserialization:** The `InferenceModel` struct in C did not match the memory layout of the saved model. **Solution:** The struct was updated to ensure correct model loading.
    4. **GUI Stability:** The stride-based prediction caused the GUI to flicker. **Solution:** Implemented a caching mechanism to display the last valid prediction during skipped frames.

- **Outcome:** After resolving these pipeline discrepancies and increasing training to 500 epochs, the system achieved production-ready status, with real-world inference performance matching the high accuracy observed during training. This successful outcome underscores the criticality of maintaining absolute consistency between training and inference environments.

---

5. System Reliability: Fixing the Post-Training Workflow

After achieving initial success, a series of architectural changes to improve model capacity and startup robustness introduced two severe, high-priority bugs that broke the core workflow.

- **Problem 1: Stale Model Inference.** After training a new model, the inference page would either crash or produce garbage predictions (e.g., 0.0 confidence). Logs from the C server revealed it was continuously sending a “no model loaded” response, even though a `c_model.bin` file existed.
    - **Investigation:** The root cause was identified as a lifecycle mismatch. The C inference server (`ra8d1_sim`) was designed to load the model file only once at startup. It was never notified when the training process overwrote the `c_model.bin` file. The server process would continue running with its initial state (either no model or an old model), completely unaware of the newly trained weights.
    - **Solution:** The architecture was corrected by making the Python GUI responsible for the C server’s lifecycle. The `TrainingPage` was modified to automatically **kill and restart the C inference server process** after every successful training run. This guarantees that the server always loads the latest model file, ensuring a seamless transition from training to inference.

- **Problem 2: Model Not Learning.** After fixing the stale model issue, a more insidious bug emerged: the training process would run, but the model’s loss and accuracy would remain completely stagnant, indicating that the weights were not being updated.
    - **Investigation:** Diagnostic logs confirmed that the gradients were being calculated correctly during the backward pass. However, a closer look at the `update_weights` function (the Adam optimizer implementation) revealed a critical memory error. The `sizeof()` operator was being used incorrectly, passing the size of the entire `Model` struct instead of the size of the specific weight/bias arrays. This caused the optimizer to read and write out of bounds, corrupting its internal state (the `m` and `v` moments) on every step and preventing any meaningful weight updates.

- **Solution:** The `sizeof()` calls in the `update_weights` function were corrected to reference the specific arrays (e.g., `sizeof(model->tcn_block.weights)`). This restored the integrity of the Adam optimizer, and the model immediately began to learn effectively. This bug served as a critical lesson in how low-level C memory errors can manifest as high-level, non-obvious machine learning failures.

Final Outcome

All major blockers were resolved. The C-based TCN backend is numerically stable, adheres to embedded memory constraints, and communicates flawlessly with the Python frontend. The project successfully demonstrates a production-ready workflow for developing and deploying advanced temporal models on resource-constrained hardware.