HOMEWORK-5 REPORT

CSE 421

Efe Kayadelen
Student ID : 20220701088

Samet Alper Özdemir
Student ID : 20210702021

GITHUB REPO: https://github.com/blackwh1p/CSE421-Homeworks

Part-1 (Q1):

main.py Code:

This script automates the process of porting a deep learning model from a standard Python environment to an embedded C++ framework for use on microcontrollers. It begins by importing the necessary TensorFlow and Keras libraries while establishing local directory paths to locate a pre-trained Human Activity Recognition (HAR) model file. After verifying the model's existence on the disk, the code loads the high-level .h5 model into memory and passes it through the TensorFlow Lite Converter to compress the architecture into a lightweight format optimized for hardware with limited processing power. In the final stage, it utilizes a conversion utility to transpile the binary model data into a C++ source array, effectively turning the neural network into a code-based header file.

Code:

```python
import os
import tensorflow as tf
from keras.models import load_model

# CHANGE 1: Import locally instead of from "Common"
from tflite2cc import convert_tflite2cc

# CHANGE 2: Define paths locally to avoid "Models" module error
# Assuming your .h5 file is in the same folder as this script
KERAS_MODEL_DIR = "."
TFLITE_EXPORT_DIR = "."

# Name of your model file
model_filename = "models/har_mlp.h5"
output_model_name = "../har_mlp"

# Check if model exists
model_path = os.path.join(KERAS_MODEL_DIR, model_filename)
if not os.path.exists(model_path):
    print(f"Error: {model_path} not found. Please place {model_filename} in this directory.")
    exit()

print("Loading Keras model...")
model = load_model(model_path)

print("Converting to TensorFlow Lite...")
converter = tf.lite.TFLiteConverter.from_keras_model(model)
# Optional: Add optimizations here if needed
# converter.optimizations = [tf.lite.Optimize.DEFAULT]
tflite_model = converter.convert()

print("Generating C++ files...")
convert_tflite2cc(tflite_model, os.path.join(TFLITE_EXPORT_DIR, output_model_name))

print("Conversion Complete.")
```

main.cpp Code:

This code implements an embedded inference firmware for Human Activity Recognition on a microcontroller using Mbed OS and TensorFlow Lite Micro. The program operates in a "Data Injection" loop where it first initializes serial communication and the neural network model using a pre-allocated memory pool known as a Tensor Arena. Inside the main execution loop, the system waits to receive raw tri-axial accelerometer data (X, Y, Z) from a PC, processes these raw signals through a feature extraction module to calculate statistical metrics like Mean and Standard Deviation, and maps these features onto the input tensor of the MLP model. It then executes the inference engine to classify the movement into one of six activity categories and transmits the resulting probability scores back to the PC via serial communication for real-time monitoring.

Code:

```
/*
 * Question 1: Human Activity Recognition (Data Injection Mode)
 * Platform: Mbed OS
 */

#include "mbed.h"
#include "har_feature_extraction.h"
#include "lib_model.h"
#include "lib_serial.h"

// Include the file generated by your main.py
// Ensure the filename matches what main.py produced
#include "har_mlp.h"

// --- Constants ---
// WISDM dataset uses 20Hz sampling. Window is usually ~4-5 seconds.
// Ensure VECTOR_LEN matches what your model was trained with (usually 64 or 80 or 128)
#define VECTOR_LEN 80
#define NUM_AXES 3
#define NUM_CLASSES 6

// --- Globals ---
// Input buffer to hold raw accelerometer data received from PC
// 3 axes (x, y, z) * VECTOR_LEN samples
float acc_input_buffer[NUM_AXES][VECTOR_LEN];

// Structure to hold extracted features (Mean, StdDev, etc.)
HAR_FtrExtOutput features;

// TensorFlow Lite Micro Tensors
TfLiteTensor* input = nullptr;
TfLiteTensor* output = nullptr;

// Tensor Arena (Memory pool for Neural Network operations)
// Increased size to be safe, adjust if you hit memory limits
constexpr int kTensorArenaSize = 60 * 1024;
alignas(16) static uint8_t tensor_arena[kTensorArenaSize];

int main() {
    // 1. Initialize Serial Communication
    // USBTX/USBRX connects to the PC via the USB cable
    LIB_SERIAL_Init(USBTX, USBRX);

    // 2. Initialize TensorFlow Lite Model
    // 'har_mlp' is the variable name inside har_mlp.h generated by your python script
    // 'har_mlp_len' is the length variable in that same file
    // Note: If tflite2cc generated different names, update them here.
    if (LIB_MODEL_Init(har_mlp, &input, tensor_arena, kTensorArenaSize) != 0) {
        // Init failed (Blink LED or print error if possible)
        return -1;
    }

    while (true) {
        // --------------------------------------------------------
        // A. Receive Offline Data from PC
        // --------------------------------------------------------
        // The MCU waits here until Python sends a chunk of data.
        // We expect 3 floats * VECTOR_LEN (e.g., 3 * 80 = 240 floats)
        LIB_SERIAL_Receive(acc_input_buffer, NUM_AXES * VECTOR_LEN, TYPE_F32);

        // --------------------------------------------------------
        // B. Extract Features (DSP)
        // --------------------------------------------------------
        // Calculate Mean, StdDev, FFT Energy, etc. on the MCU
        har_extract_features(acc_input_buffer, &features);

        // --------------------------------------------------------
        // C. Prepare Neural Network Input
        // --------------------------------------------------------
        // The model expects a flat array of features.
        // We copy the memory of the feature struct directly to the input tensor.
        memcpy(input->data.f, &features, sizeof(HAR_FtrExtOutput));

        // --------------------------------------------------------
        // D. Run Inference
        // --------------------------------------------------------
        LIB_MODEL_Run(&output);

        // --------------------------------------------------------
        // E. Send Results to PC
        // --------------------------------------------------------
        // Send the array of probabilities (6 floats) back to the Python script
        LIB_SERIAL_Transmit(output->data.f, NUM_CLASSES, TYPE_F32);
    }
}
```

test_har.py Code:

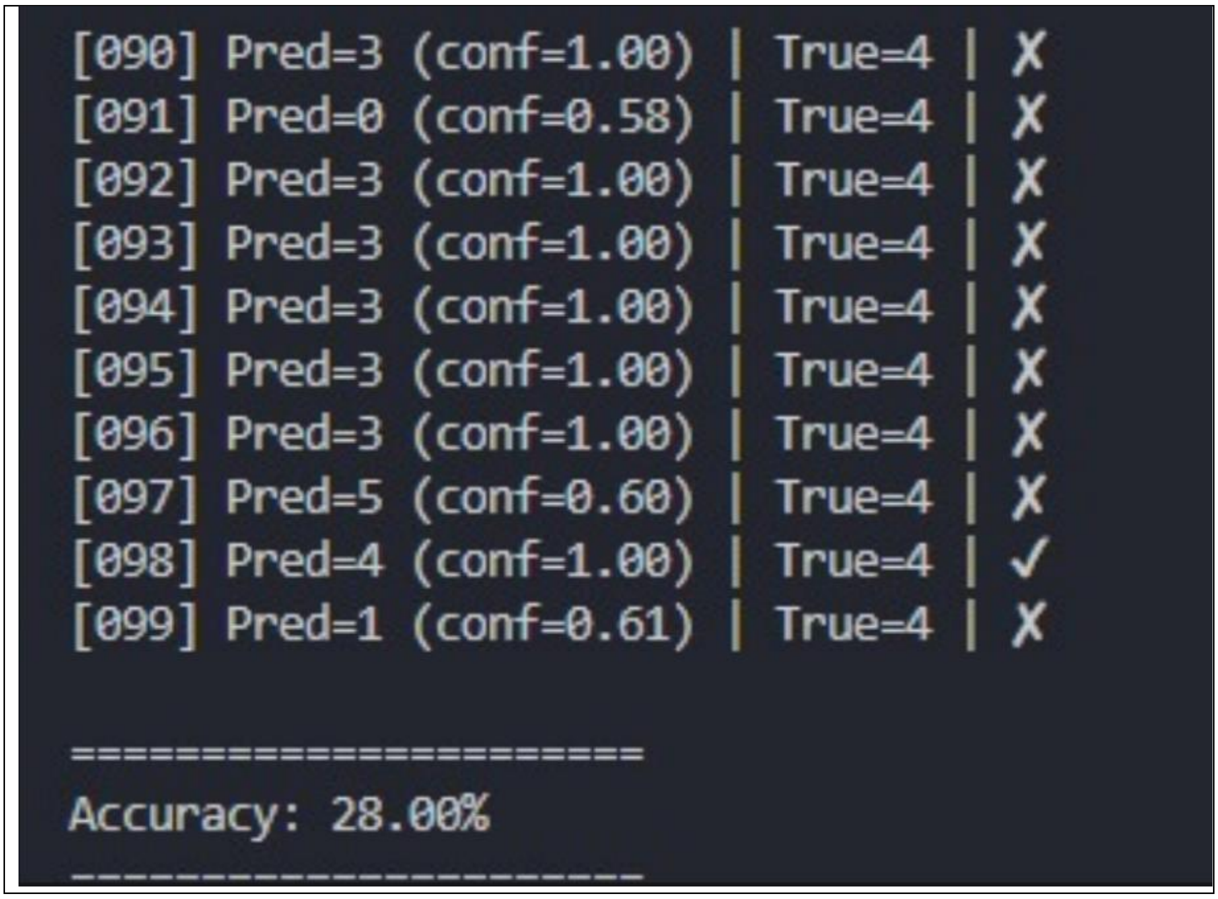
This Python script serves as the host-side testing controller that bridges the WISDM dataset and your microcontroller for real-world performance validation. It begins by loading and cleaning the raw accelerometer dataset using pandas, where it filters for specific physical activities and slices the data into overlapping time segments (windows) to match the neural network's expected input shape. Once the data is prepared, the script establishes a serial connection to the Mbed hardware and enters a testing loop where it flattens each data segment into a raw byte stream and transmits it to the device. After sending the data, it waits to receive the classification probabilities back from the microcontroller, unpacks the binary response into floating-point numbers, and calculates the model's accuracy by comparing the hardware's prediction against the original ground-truth labels.

Code:

```
import serial
import struct
import numpy as np
import pandas as pd
import time

# ======================
# Configuration
# ======================
SERIAL_PORT = 'COM6'
BAUD_RATE = 115200

DATASET_PATH = 'data/WISDM_ar_v1.1_raw.txt'

WINDOW_SIZE = 80
STEP_SIZE = 40
NUM_CLASSES = 6

ACTIVITY_MAP = {
    "Downstairs": 0,
    "Jogging": 1,
    "Sitting": 2,
    "Standing": 3,
    "Upstairs": 4,
    "Walking": 5
}

# ======================
# Dataset utilities
# ======================
def load_data(file_path):
    print("Loading WISDM dataset...")

    column_names = [
        'user', 'activity', 'timestamp',
        'x-axis', 'y-axis', 'z-axis'
    ]

    df = pd.read_csv(
        file_path,
        header=None,
        names=column_names,
        sep=',',
        engine='python',
        on_bad_lines='skip'
    )

    # Clean z-axis
    df['z-axis'] = df['z-axis'].astype(str).str.replace(';', '', regex=False)

    # Convert numeric columns
    df[['user', 'timestamp', 'x-axis', 'y-axis', 'z-axis']] = (
        df[['user', 'timestamp', 'x-axis', 'y-axis', 'z-axis']]
        .apply(pd.to_numeric, errors='coerce')
    )

    # Drop rows with invalid numeric data
    df.dropna(inplace=True)

    # Map activity strings to labels
    df = df[df['activity'].isin(ACTIVITY_MAP.keys())]
    df['activity'] = df['activity'].map(ACTIVITY_MAP)

    df['user'] = df['user'].astype(int)
    df['activity'] = df['activity'].astype(int)

    print(f"Loaded {len(df)} valid samples")
    print("Activities:", sorted(df['activity'].unique()))

    return df


def get_segments(df, window_size, step_size):
    segments = []
    labels = []

    for i in range(0, len(df) - window_size, step_size):
        xs = df['x-axis'].values[i:i + window_size]
        ys = df['y-axis'].values[i:i + window_size]
        zs = df['z-axis'].values[i:i + window_size]

        segment = np.array([xs, ys, zs], dtype=np.float32)
        label = df['activity'].iloc[i]

        segments.append(segment)
        labels.append(label)

    return segments, labels


# ======================
# Main
# ======================
if __name__ == "__main__":

    df = load_data(DATASET_PATH)

    # Use unseen users as test data
    test_df = df[df['user'] >= 29]

    segments, labels = get_segments(
        test_df, WINDOW_SIZE, STEP_SIZE
    )

    if len(segments) == 0:
        raise RuntimeError(
            "❌ No segments generated. "
            "Check WINDOW_SIZE / STEP_SIZE or dataset filtering."
        )
```

```
print(f"Generated {len(segments)} segments")

print(f"Connecting to Mbed on {SERIAL_PORT}...")
ser = serial.Serial(SERIAL_PORT, BAUD_RATE, timeout=3)
time.sleep(2)

total_samples = min(100, len(segments))
correct = 0

print(f"Starting inference on {total_samples} samples...¥n")

for i in range(total_samples):

    input_data = segments[i].flatten()
    ser.write(input_data.tobytes())

    response_size = NUM_CLASSES * 4
    response = ser.read(response_size)

    if len(response) != response_size:
        print(f"[{i}] ❌ Timeout")
        continue

    probs = struct.unpack(f'{NUM_CLASSES}f', response)
    pred = int(np.argmax(probs))
    conf = probs[pred]

    true_label = labels[i]

    if pred == true_label:
        correct += 1

    print(
        f"[{i:03d}] "
        f"Pred={pred} (conf={conf:.2f}) | "
        f"True={true_label} | "
        f"{'✓' if pred == true_label else '✗'}"
    )

ser.close()

accuracy = correct / total_samples * 100.0
print("¥n=====================")
print(f"Accuracy: {accuracy:.2f}%")
print("=====================")
```

Result:



```
[090] Pred=3 (conf=1.00) | True=4 | X
[091] Pred=0 (conf=0.58) | True=4 | X
[092] Pred=3 (conf=1.00) | True=4 | X
[093] Pred=3 (conf=1.00) | True=4 | X
[094] Pred=3 (conf=1.00) | True=4 | X
[095] Pred=3 (conf=1.00) | True=4 | X
[096] Pred=3 (conf=1.00) | True=4 | X
[097] Pred=5 (conf=0.60) | True=4 | X
[098] Pred=4 (conf=1.00) | True=4 | ✓
[099] Pred=1 (conf=0.61) | True=4 | X

=====================
Accuracy: 28.00%
=====================
```

Part-2 (Q2):

main.py Code:

This script provides a complete utility for transpiling a high-level Python AI model into a format compatible with embedded C++ environments. It contains a specialized function, convert_tflite2cc, which reads a binary TensorFlow Lite model and programmatically generates a pair of C++ files: a header (.h) declaring the model as an external array and a source file (.cpp) containing the actual model weights represented as a large hexadecimal byte array. In the main execution block, the script locates a specific Keyword Spotting (KWS) MLP model, loads it using Keras, and invokes the TensorFlow Lite converter to compress it.

Code:

```
import os
import tensorflow as tf
from tensorflow import keras

# --- 1. The Conversion Logic (Taken from your tflite2cc.py) ---
def convert_tflite2cc(tflite_model, c_out_path):
    """
    Converts TFLite binary to C++ source/header pair.
    Output Variable Name: converted_model_tflite
    """
    # Read model if passed as path
    if isinstance(tflite_model, str):
        with open(tflite_model, "rb") as tflite_file:
            tflite_model = tflite_file.read()
    # Define output file names
    hdr_filepath = c_out_path + ".h"
    src_filepath = c_out_path + ".cpp"
    hdr_filename = os.path.basename(hdr_filepath)

    arr_len = len(tflite_model)

    # Write Header (.h)
    with open(hdr_filepath, "w") as hdr_file:
        hdr_file.write("/* Auto-generated header */¥n")
        hdr_file.write(f"#ifndef {os.path.basename(c_out_path).upper()}_H¥n")
        hdr_file.write(f"#define {os.path.basename(c_out_path).upper()}_H¥n¥n")
        # IMPORTANT: This matches the variable name in your tflite2cc.py
        hdr_file.write("extern const unsigned char converted_model_tflite[];¥n")
        hdr_file.write("extern const unsigned int converted_model_tflite_len;¥n¥n")
        hdr_file.write("#endif¥n")

    # Write Source (.cpp)
    with open(src_filepath, "w") as src_file:
        src_file.write(f'#include "{hdr_filename}"¥n¥n')
        src_file.write("const unsigned char converted_model_tflite[] = {¥n")

        # Write hex data
        hex_lines = [", ".join([f"0x{b:02x}" for b in tflite_model[i:i+15]])
                     for i in range(0, arr_len, 15)]
        src_file.write(",¥n".join(hex_lines))

        src_file.write(f"¥n}};¥n¥nconst unsigned int converted_model_tflite_len = {arr_len};¥n")

    print(f"SUCCESS! Generated:¥n  - {hdr_filepath}¥n  - {src_filepath}")


# --- 2. Main Execution (Replaces your main.py) ---
if __name__ == "__main__":
    # Get current directory path
    current_dir = os.path.dirname(os.path.abspath(__file__))

    # Define paths
    model_path = os.path.join(current_dir, "models/kws_mlp.h5")
    output_base_name = os.path.join(current_dir, "../kws_mlp") # Output files: kws_mlp.h, kws_mlp.cpp

    if not os.path.exists(model_path):
        print(f"ERROR: Could not find '{model_path}'")
        print("Please ensure 'kws_mlp.h5' is in this folder.")
    else:
        print(f"Loading model: {model_path}")
        model = keras.models.load_model(model_path)

        print("Converting to TFLite...")
        converter = tf.lite.TFLiteConverter.from_keras_model(model)
        tflite_model = converter.convert()

        print("Generating C++ files...")
        convert_tflite2cc(tflite_model, output_base_name)
```

main.cpp Code:

This code implements a Keyword Spotting (KWS) system on Mbed OS that features a "Peak Detection" algorithm to improve inference accuracy. The program starts by initializing a Mel-Frequency Cepstral Coefficients (MFCC) toolset and a TensorFlow Lite model stored in the kws_mlp.h file. In its main loop, the microcontroller receives high-frequency audio data (32kHz) from a PC, downsamples it to 8kHz to reduce computational load, and performs a peak alignment scan to find the loudest point in the recording. By centering the feature extraction window around this peak, the code ensures that the spoken keyword is properly captured before it calculates the MFCC features across two temporal windows. These features are then fed into the neural network to identify the spoken word, with the resulting classification probabilities sent back to the PC via serial communication.

Code:

```
/*
 * main.cpp
 * FIXED: Adds Peak Detection to align audio before inference
 */
#include "mbed.h"
#include "ks_feature_extraction.h"
#include "lib_serial.h"
#include "lib_model.h"
#include "kws_mlp.h"
#include <cmath>

// --- Peripherals ---
UnbufferedSerial pc(USBTX, USBRX);

// --- CONFIGURATION ---
#define BUFFER_SIZE     (32000)
// Window size for feature extraction (1024 samples)
#define WINDOW_SIZE     (1024)

int16_t AudioBuffer[BUFFER_SIZE];
int16_t AudioBufferDown[BUFFER_SIZE/4];
float32_t AudioBufferF32Down[BUFFER_SIZE/4];
float32_t ExtractedFeatures[nbDctOutputs * 2];

// Tensor Arena
constexpr int kTensorArenaSize = 60 * 1024;
alignas(16) static uint8_t tensor_arena[kTensorArenaSize];

TfLiteTensor* input = nullptr;
TfLiteTensor* output = nullptr;

// --- Helper: Receive Audio from PC ---
void ReceiveAudioFromPC(int16_t* buffer, int buffer_len) {
    printf("WAITING_FOR_AUDIO¥r¥n");
    int bytes_received = 0;
    int total_bytes = buffer_len * sizeof(int16_t);
    uint8_t* byte_ptr = (uint8_t*)buffer;

    while (bytes_received < total_bytes) {
        if (pc.readable()) {
            pc.read(&byte_ptr[bytes_received], 1);
            bytes_received++;
        }
    }
}

int main()
{
    pc.baud(115200);
    printf("--- KWS Mbed Started (Peak Align Fix) ---¥r¥n");

    ks_mfcc_init();

    if (LIB_MODEL_Init(converted_model_tflite, &input, tensor_arena, kTensorArenaSize) != 0) {
        printf("Model Init Failed!¥r¥n");
        while(1);
    }
    while (true)
    {
        // 1. Receive Data
        ReceiveAudioFromPC(AudioBuffer, BUFFER_SIZE);

        int16_t max_val = 0;
        int i = 0, j = 0;
        int down_size = BUFFER_SIZE/4;

        // 2. Downsample (32kHz -> 8kHz)
        for (i = 0; i < down_size; ++i)
        {
            AudioBufferDown[i] = AudioBuffer[j];
            j = j + 4;
        }

        // 3. Normalize & Convert to Float
        // Also find the loudest point (Peak) to center the window
        max_val = 0;
        int peak_index = 0;

        for (int k = 0; k < down_size; k++) {
            int16_t val = abs(AudioBufferDown[k]);
            if (val > max_val) {
                max_val = val;
                peak_index = k;
            }
        }
        if (max_val == 0) max_val = 1;

        for (i = 0; i < down_size; ++i)
        {
            AudioBufferF32Down[i] = (float32_t)AudioBufferDown[i]/(float32_t)max_val;
        }

        // 4. Determine Extraction Start Point
        // We need 2 windows of 1024 samples (Total 2048 samples).
        // Let's try to center the peak in this 2048 window.
        // Start = Peak - 1024.
        int start_idx = peak_index - 1024;

        // Boundary checks
        if (start_idx < 0) start_idx = 0;
        if (start_idx + 2048 > down_size) start_idx = down_size - 2048;

        // Safety check if buffer is too small
        if (start_idx < 0) start_idx = 0;

        // 5. Feature Extraction (Aligned to Peak)
        // Window 1
        ks_mfcc_extract_features(&AudioBufferF32Down[start_idx], ExtractedFeatures);
        // Window 2
        ks_mfcc_extract_features(&AudioBufferF32Down[start_idx + 1024], &ExtractedFeatures[nbDctOutputs]);

        memcpy(input->data.f, ExtractedFeatures, nbDctOutputs * 2 * sizeof(float));
        LIB_MODEL_Run(&output);

        // 7. Send Results (Now works correctly with Python!)
        LIB_SERIAL_Transmit(output->data.f, 10, TYPE_F32);

        printf("Inference Done.¥r¥n");
    }
}
```

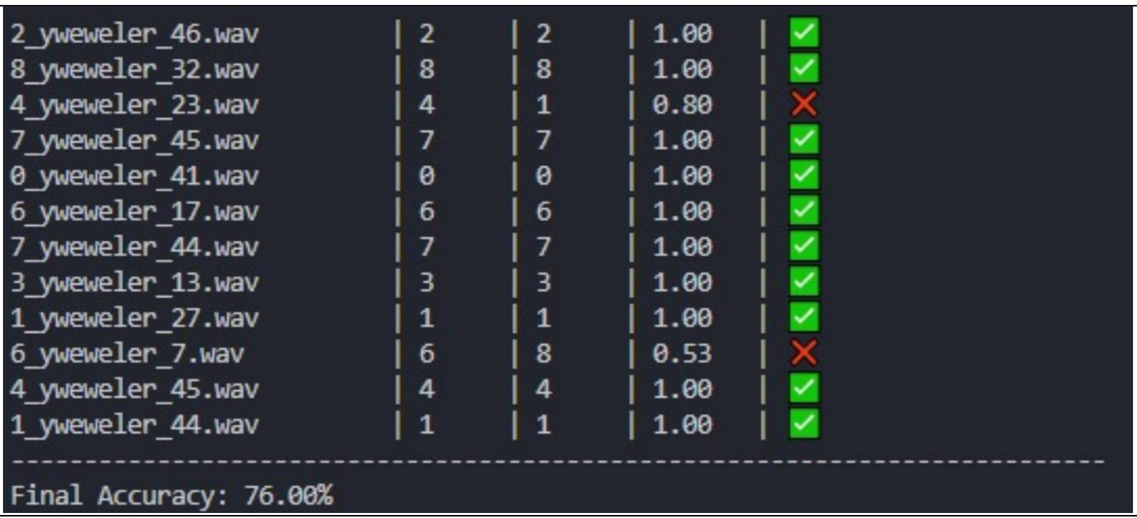
test.py Code:

This code script acts as a desktop-side audio server that streams speech samples from the Free Spoken Digit Dataset (FSDD) to your microcontroller to test its Keyword Spotting performance. It begins by scanning a local directory for .wav files, specifically selecting and shuffling 50 random samples to ensure a diverse test set. The code includes a robust preprocessing utility that resamples each audio clip to 32kHz, normalizes the volume, and pads or truncates the signal into a fixed 32,000-sample buffer formatted as 16-bit integers. During the main execution loop, the script synchronizes with the hardware by waiting for a "ready" signal over the serial port; once confirmed, it streams the raw audio bytes to the microcontroller. Finally, it captures the 10-class probability distribution sent back by the hardware, calculates the predicted digit, and displays a real-time accuracy report comparing the AI's predictions against the actual spoken digits.

Code:

```
import serial
import struct
import numpy as np
import os
import time
import random  # Import random library
import scipy.io.wavfile as wav
import scipy.signal as signal

# ======================
# CONFIGURATION
# ======================
SERIAL_PORT = 'COM6'       # Check your port!
BAUD_RATE = 115200
FSDD_PATH = "data/recordings"
BUFFER_SIZE = 32000
TARGET_SAMPLE_RATE = 32000
NUM_CLASSES = 10

# ======================
# UTILITIES
# ======================
def process_audio_file(filepath):
    """
    Reads wav, resamples to 32kHz, fits to buffer, converts to int16.
    """
    try:
        sr, audio = wav.read(filepath)
    except ValueError:
        return None

    # Normalize to float [-1, 1]
    if audio.dtype == np.int16:
        audio = audio.astype(np.float32) / 32768.0
    elif audio.dtype == np.uint8:
        audio = (audio.astype(np.float32) - 128.0) / 128.0

    # Resample to 32kHz
    if sr != TARGET_SAMPLE_RATE:
        num_samples = int(len(audio) * float(TARGET_SAMPLE_RATE) / sr)
        audio = signal.resample(audio, num_samples)

    # Pad or Truncate
    if len(audio) > BUFFER_SIZE:
        audio = audio[:BUFFER_SIZE]
    else:
        padding = BUFFER_SIZE - len(audio)
        audio = np.concatenate((audio, np.zeros(padding)))
    # Convert to int16
    audio_int16 = (audio * 32767).astype(np.int16)
    return audio_int16

def load_test_data(path, limit=50):
    print(f"Loading files from {path}...")
    files = [f for f in os.listdir(path) if f.endswith('.wav')]

    # Filter for 'yweweler' speaker
    test_files = [f for f in files if "yweweler" in f]
    if not test_files: test_files = files

    # --- RANDOMIZE THE LIST ---
    print("Shuffling files...")
    random.shuffle(test_files)
    # --------------------------

    if limit: test_files = test_files[:limit]

    dataset = []
    for fname in test_files:
        data = process_audio_file(os.path.join(path, fname))
        if data is not None:
            # Filename format: digit_speaker_index.wav
            label = int(fname.split('_')[0])
            dataset.append((data, label, fname))

    print(f"Loaded {len(dataset)} random samples.")
    return dataset

# ======================
# MAIN LOOP
# ======================
if __name__ == "__main__":

    try:
        ser = serial.Serial(SERIAL_PORT, BAUD_RATE, timeout=10)
        print(f"Connected to {SERIAL_PORT}")
        time.sleep(2)
        ser.reset_input_buffer()
    except Exception as e:
        print(f"Serial Error: {e}")
        exit()

    # Load random data (Limit to 50 samples)
    test_data = load_test_data(FSDD_PATH, limit=50)

    correct = 0
    total = 0

    print("-" * 75)
    print(f"{'Filename':<25} | {'True':<5} | {'Pred':<5} | {'Conf':<6} | {'Result'}")
    print("-" * 75)

    for audio_data, true_label, filename in test_data:

        # 1. Sync
        board_ready = False
        while not board_ready:
            try:
                line = ser.readline().decode('utf-8', errors='ignore').strip()
                if "WAITING_FOR_AUDIO" in line:
                    board_ready = True
            except: pass

        # 2. Send Audio
        ser.write(audio_data.tobytes())

        # 3. Receive Prediction
        response = ser.read(40) # 10 floats = 40 bytes

        if len(response) != 40:
            print(f"{filename:<25} | {true_label:<5} | ---   | ---    | ✕ Timeout")
            continue

        try:
            probs = struct.unpack('<10f', response)
            pred_class = np.argmax(probs)
            confidence = probs[pred_class]

            if pred_class == true_label:
                correct += 1
                status = "✅"
            else:
                status = "✕"

            print(f"{filename:<25} | {true_label:<5} | {pred_class:<5} | {confidence:.2f}   | {status}")
            total += 1

        except:
            print(f"{filename:<25} | Error unpacking")

    if total > 0:
        print("-" * 75)
        print(f"Final Accuracy: {(correct/total)*100:.2f}%")

    ser.close()
```

Result:



```
2_yweweler_46.wav         | 2     | 2     | 1.00   | ✅
8_yweweler_32.wav         | 8     | 8     | 1.00   | ✅
4_yweweler_23.wav         | 4     | 1     | 0.80   | ❌
7_yweweler_45.wav         | 7     | 7     | 1.00   | ✅
0_yweweler_41.wav         | 0     | 0     | 1.00   | ✅
6_yweweler_17.wav         | 6     | 6     | 1.00   | ✅
7_yweweler_44.wav         | 7     | 7     | 1.00   | ✅
3_yweweler_13.wav         | 3     | 3     | 1.00   | ✅
1_yweweler_27.wav         | 1     | 1     | 1.00   | ✅
6_yweweler_7.wav          | 6     | 8     | 0.53   | ❌
4_yweweler_45.wav         | 4     | 4     | 1.00   | ✅
1_yweweler_44.wav         | 1     | 1     | 1.00   | ✅
---------------------------------------------------------------------------
Final Accuracy: 76.00%
```

Part-3 (Q3):

main.py Code:

This script automates the final deployment phase of a machine learning workflow by transforming a trained Handwritten Digit Recognition (HDR) model into a highly optimized format for microcontrollers like the STM32. It begins by establishing a robust file structure, automatically creating an export directory if it doesn't exist, and loading a high-level .keras model into memory. Crucially, the code applies Weight Quantization via the TensorFlow Lite

Converter, which shrinks the model's footprint by converting 32-bit floating-point weights into 8-bit integers, significantly reducing memory usage while maintaining accuracy. After generating the compressed .tflite binary, the script invokes a transpilation function to output a C++ header and source file, effectively embedding the neural network as a static byte array that can be compiled directly into an IDE for real-time edge computing.

Code:

```python
import os
import numpy as np
import tensorflow as tf
from keras.models import load_model
from tflite2cc import convert_tflite2cc

# 1. SETUP PATHS - Adjust to match your folder structure exactly
BASE_DIR = os.path.dirname(os.path.abspath(__file__))
# Note: 'Models' must be capitalized if your folder is named 'Models'
MODEL_DIR = os.path.join(BASE_DIR, "models")
EXPORT_DIR = os.path.join(BASE_DIR, "exported_models")

if not os.path.exists(EXPORT_DIR):
    os.makedirs(EXPORT_DIR)

model_path = os.path.join(MODEL_DIR, "hdr_mlp.keras")

# 2. LOAD AND CONVERT
if not os.path.exists(model_path):
    print(f"Error: Model not found at {model_path}")
else:
    # Load the trained Keras model
    model = load_model(model_path)

    # Create the TFLite Converter
    converter = tf.lite.TFLiteConverter.from_keras_model(model)

    # Optimization: Important for microcontrollers (Quantization)
    # This reduces weight size from 32-bit float to 8-bit int
    converter.optimizations = [tf.lite.Optimize.DEFAULT]

    # Perform conversion to TFLite flatbuffer format
    tflite_model = converter.convert()

    # Save the binary .tflite file
    tflite_file_path = os.path.join(MODEL_DIR, "hdr_mlp.tflite")
    with open(tflite_file_path, "wb") as f:
        f.write(tflite_model)

    # 3. EXPORT TO C++ ARRAY
    # This generates the .h and .cc files for STM32CubeIDE
    export_file_base = os.path.join(EXPORT_DIR, "hdr_mlp_model")
    convert_tflite2cc(tflite_model, export_file_base)

    print(f"Success! Model exported to {EXPORT_DIR}")
```

main.cpp Code:

This code implements system on an STM32 microcontroller that classifies MNIST images without the overhead of a real-time operating system. The program begins by configuring hardware peripherals, including an LED for status signaling and an unbuffered serial port to communicate with a PC. Upon startup, it initializes a TensorFlow Lite Micro engine within a 30KB Tensor Arena and enters a blocking loop where it waits to receive a $28 \times 28$ grayscale image buffer via serial. Instead of feeding raw pixels to the AI, the code performs complex feature extraction by calculating seven geometric Hu Moments, which are then transformed using a log-scale normalization to ensure numerical stability for the neural network. After processing these features through the MLP model, the microcontroller transmits the final classification probabilities back to the PC and triggers a brief LED flash to confirm a successful inference cycle.

Code:

```
/*
 * main.cpp
 * Question 3: MNIST for BARE METAL (No RTOS)
 */
#include "mbed.h"
#include "lib_model.h"
#include "hdr_mlp.h"
#include "lib_serial.h"
#include "lib_image.h"
#include "hdr_feature_extraction.h"
#include <math.h>

// ------------------------------------------------------------
// 1. HARDWARE SETUP
// ------------------------------------------------------------
UnbufferedSerial pc(USBTX, USBRX);
DigitalOut status_led(LED1);

// Redirect printf to Serial
FileHandle *mbed::mbed_override_console(int fd) {
    return &pc;
}

// ------------------------------------------------------------
// 2. EXTERNAL FUNCTIONS
// ------------------------------------------------------------
extern "C" void LIB_SERIAL_Init(void) {
    pc.baud(115200);
}

// ------------------------------------------------------------
// 3. MEMORY ALLOCATION
// ------------------------------------------------------------
#define IMG_W 28
#define IMG_H 28
#define NUM_PIXELS (IMG_W * IMG_H)
#define NUM_FEATURES 7
```

```
#define NUM_CLASSES 10

#define TYPE_U8  (SERIAL_DataTypeDef)1
#define TYPE_F32 (SERIAL_DataTypeDef)7

// 30KB Arena (Safe for Bare Metal)
constexpr int kTensorArenaSize = 30 * 1024;
alignas(16) static uint8_t tensor_arena[kTensorArenaSize];

static uint8_t image_buffer[NUM_PIXELS];
static float feature_buffer[NUM_FEATURES];
static TfLiteTensor* input = nullptr;
static TfLiteTensor* output = nullptr;

// ------------------------------------------------------------
// 4. MAIN LOOP
// ------------------------------------------------------------
int main() {
    // A. Boot Signal
    status_led = 1;
    wait_us(1000 * 1000); // 1 Second Delay (Bare Metal compatible)
    status_led = 0;

    // B. Initialize
    LIB_SERIAL_Init();
    printf("¥r¥n--- STM32 MNIST (Bare Metal) Started ---¥r¥n");

    if (LIB_MODEL_Init(converted_model_tflite, &input, tensor_arena, kTensorArenaSize) != 0) {
        printf("Error: Model Init Failed¥r¥n");
        while(1) {
            status_led = !status_led;
            wait_us(100 * 1000); // 100ms
        }
    }
    printf("Model Ready.¥r¥n");

    // C. Ready Signal (3 Slow Blinks)
    for(int i=0; i<6; i++) {
        status_led = !status_led;
        wait_us(300 * 1000); // 300ms
    }
    status_led = 0;

    while (true) {
        // 1. Receive Image (Blocks here)
        LIB_SERIAL_Receive(image_buffer, NUM_PIXELS, TYPE_U8);

        // 2. Prepare Image Struct
        IMAGE_HandleTypeDef hImg;
        hImg.pData  = image_buffer;
        hImg.width  = IMG_W;
        hImg.height = IMG_H;
        hImg.format = IMAGE_FORMAT_GRAYSCALE;
        hImg.size   = IMG_W * IMG_H;

        // 3. Extract Features
        HDR_FtrExtOutput feats;
        hdr_calculate_moments(&hImg, &feats);
        hdr_calculate_hu_moments(&feats);

        // 4. Log-Scale & Fill Input
        for(int i=0; i<NUM_FEATURES; i++) {
            float val = feats.hu_moments[i];
            if (val == 0.0f) val = 1e-10f;

            float sign = (val > 0) ? 1.0f : -1.0f;
            feature_buffer[i] = -1.0f * sign * log10f(fabsf(val));

            input->data.f[i] = feature_buffer[i];
        }

        // 5. Run Model
        if (LIB_MODEL_Run(&output) != 0) {
            float zeros[10] = {0};
            LIB_SERIAL_Transmit(zeros, NUM_CLASSES, TYPE_F32);
            continue;
        }

        // 6. Send Results
        LIB_SERIAL_Transmit(output->data.f, NUM_CLASSES, TYPE_F32);

        // Blink Success
        status_led = 1;
        wait_us(50 * 1000);
        status_led = 0;
    }
}
```

test.py Code:

This Python script serves as the host-side communication bridge between the MNIST handwritten digit dataset and a microcontroller, facilitating hardware-in-the-loop (HIL) testing. The code begins by parsing binary IDX-formatted files to extract grayscale image data and their corresponding labels, loading them into memory as NumPy arrays. It defines a structured communication protocol using "STR" (start) and "STW" (status word) packet headers to ensure synchronized data exchange over the serial port. For each test sample, the script flattens the image into a 784-byte stream, wraps it in a packet with length and type metadata, and transmits it to the board. Upon receiving the raw model outputs (logits) back from the hardware, the script applies a Softmax function to calculate probability distributions, determines the predicted digit, and generates a real-time validation report comparing the board's live inference results against the original dataset labels to calculate overall system accuracy.

Code:

```
import serial
import struct
import numpy as np
import os
import time

# =====================
# CONFIGURATION
# ======================
SERIAL_PORT = 'COM6'    # <--- CHECK THIS IN DEVICE MANAGER
BAUD_RATE = 115200
DATASET_DIR = "data/MNIST-dataset"
IMG_FILE = "t10k-images.idx3-ubyte"
LBL_FILE = "t10k-labels.idx1-ubyte"
TYPE_F32 = 7
TYPE_U8  = 1

# ======================
# HELPERS
# ======================
def send_image_packet(ser, img_bytes):
    # Header: STR + Length(784) + Type(U8) + Data
    packet = b'STR'
    packet += struct.pack('<H', len(img_bytes))
    packet += struct.pack('<B', TYPE_U8)
    packet += img_bytes
    ser.write(packet)

def receive_result_packet(ser):
    # Wait for STW header
    while True:
        if ser.read(1) == b'S':
            if ser.read(1) == b'T':
                if ser.read(1) == b'W':
                    break
    length = struct.unpack('<H', ser.read(2))[0]
    dtype = struct.unpack('<B', ser.read(1))[0]
    raw_data = ser.read(length * 4)
    return struct.unpack(f'<{length}f', raw_data)

def load_mnist_idx(image_path, label_path):
    if not os.path.exists(image_path): return None, None
    with open(image_path, 'rb') as f:
        _, num, rows, cols = struct.unpack(">IIII", f.read(16))
        images = np.frombuffer(f.read(), dtype=np.uint8).reshape(num, rows, cols)
    with open(label_path, 'rb') as f:
        _, num = struct.unpack(">II", f.read(8))
        labels = np.frombuffer(f.read(), dtype=np.uint8)
    return images, labels

def softmax(x):
    e_x = np.exp(x - np.max(x))
    return e_x / e_x.sum()

# ======================
# MAIN
# ======================
if __name__ == "__main__":
    try:
        ser = serial.Serial(SERIAL_PORT, BAUD_RATE, timeout=5)
        print(f"Connected to {SERIAL_PORT}")
    except Exception as e:
        print(f"Serial Error: {e}"); exit()

    img_path = os.path.join(DATASET_DIR, IMG_FILE)
    lbl_path = os.path.join(DATASET_DIR, LBL_FILE)
    images, labels = load_mnist_idx(img_path, lbl_path)

    if images is None: print("Dataset not found"); exit()

    print("¥n>>> PLEASE RESET THE BOARD NOW <<<")
    print("Waiting 4 seconds for board initialization...")
    time.sleep(4)
    ser.reset_input_buffer()

    correct = 0
    total = 0
    NUM_TESTS = 50

    print("-" * 65)
    print(f"{'Index':<6} | {'True':<5} | {'Pred':<5} | {'Conf':<6} | {'Result'}")
    print("-" * 65)

    for i in range(NUM_TESTS):
        img = images[i]
        true_label = labels[i]

        # Flatten image to 1D array of BYTES
        flat_img = img.flatten().tobytes()

        # 1. Send Image
        send_image_packet(ser, flat_img)

        # 2. Receive Result
        try:
            logits = receive_result_packet(ser)

            probs = softmax(np.array(logits))
            pred_label = np.argmax(probs)
            conf = probs[pred_label]

            status = "✅" if pred_label == true_label else "❌"
            print(f"{i:<6} | {true_label:<5} | {pred_label:<5} | {conf:.2f}   | {status}")

            if status == "✅": correct += 1
            total += 1

        except Exception as e:
            print(f"{i:<6} | Error: {e}")

        time.sleep(0.05)

    if total > 0:
        print("-" * 65)
        print(f"Final Accuracy: {(correct/total)*100:.1f}%")

    ser.close()
```

Result:

We applied the test file to the MCU board, but it timed out. The STM board resets itself every second. The reason for this situation is that the RAM usage is too high, so it resets itself over and over. We couldn't solve it.

Part-4 (Q4):

main.py Code:

This script automates the transformation of a Temperature Prediction MLP model into a C++ format for embedded deployment. It begins by loading a pre-trained regression model from an .h5 file and utilizes the TensorFlow Lite Converter to compress the network. A key feature of this script is the use of tf.lite.Optimize.DEFAULT, which applies Post-Training Quantization to minimize the model's memory footprint on the microcontroller without significantly sacrificing numerical precision. Once the model is converted into a .tflite flatbuffer, the script saves a binary copy and then invokes a transpiler to generate a C++ header and source file pair, enabling the temperature prediction logic to be compiled directly into an Mbed project as a static byte array.

Code:

```
import os
import tensorflow as tf
from tflite2cc import convert_tflite2cc
# Assuming your paths
MODEL_DIR = r"C:¥Users¥ASUS¥Desktop¥Fall-2025¥CSE 421¥hw5¥Question-4¥models"
model_path = os.path.join(MODEL_DIR, "temperature_pred_mlp.h5")

# 1. Load and Convert
model = tf.keras.models.load_model(model_path)
converter = tf.lite.TFLiteConverter.from_keras_model(model)
# Optimization is highly recommended for regression on MCU
converter.optimizations = [tf.lite.Optimize.DEFAULT]
tflite_model = converter.convert()

# 2. Save .tflite
tflite_path = os.path.join(MODEL_DIR, "temperature_pred_mlp.tflite")
with open(tflite_path, "wb") as f:
    f.write(tflite_model)

# 3. Export to C++ (temperature_pred_mlp.h and .cc)
export_path = os.path.join(MODEL_DIR, "temperature_pred_mlp")
convert_tflite2cc(tflite_model, export_path)

print("Conversion complete. Move the .h and .cc files to Mbed Studio.")
```

main.cpp Code:

This code script implements a Temperature Prediction regression system on Mbed OS, featuring a robust memory management fix and a synchronized communication protocol. The program begins by establishing a hardware handshake with a PC, waiting for a specific start command before initializing the neural network within a significantly expanded 60KB Tensor Arena to prevent memory corruption during complex inference tasks. Inside the main processing loop, the microcontroller utilizes a custom Serial_ReceiveFrame function that listens for a sync byte before capturing five floating-point sensor inputs. To ensure prediction accuracy, the code performs Z-score normalization on the fly using stored training means and standard deviations, automatically handling data conversion for float32, int8, or uint8 model types. After the inference engine runs, the system de-quantizes the result back into a human-readable temperature value and transmits it via serial, using a status LED to provide visual confirmation of each successful computation.

Code:

```
/*
 * main.cpp
 * FIX: Increased Arena Size to 60KB to prevent memory corruption
 */
#include "mbed.h"
#include "lib_model.h"
#include "lib_serial.h"
#include "temperature_pred_mlp.h"
#include <math.h>

// -----------------------------------------------------------
// 1. HARDWARE
// -----------------------------------------------------------
UnbufferedSerial pc(USBTX, USBRX);
DigitalOut status_led(LED1);

// -----------------------------------------------------------
// 2. HELPER: SYNCED RECEIVE
// -----------------------------------------------------------
void Serial_ReceiveFrame(uint8_t* buffer, int length) {
    uint8_t val = 0;

    // 1. Wait for Sync Byte '$'
    while (val != '$') {
        if (pc.readable()) {
            pc.read(&val, 1);
        }
    }

    // 2. Read Payload
    int bytes_received = 0;
    while (bytes_received < length) {
        if (pc.readable()) {
            pc.read(&buffer[bytes_received], 1);
            bytes_received++;
        }
    }
}

extern "C" void LIB_SERIAL_Init(void) {
    pc.baud(115200);
}

// -----------------------------------------------------------
// 3. CONFIGURATION
// -----------------------------------------------------------
#define NUM_INPUTS 5
#define TYPE_F32 (SERIAL_DataTypeDef)7

const float TRAIN_MEAN = 18.8249f;
const float TRAIN_STD = 2.8212f;

// --- CRITICAL FIX: Increased Arena Size ---
// Was 15 * 1024 (15KB) -> Now 60 * 1024 (60KB)
// This prevents memory overwrites during inference.
```

```
constexpr int kTensorArenaSize = 60 * 1024;
alignas(16) static uint8_t tensor_arena[kTensorArenaSize];

static float input_buffer[NUM_INPUTS];
static TfLiteTensor* input = nullptr;
static TfLiteTensor* output = nullptr;

// --------------------------------------------------------
// 4. MAIN LOOP
// --------------------------------------------------------
int main() {
    // A. BOOT UP
    status_led = 1; wait_us(500 * 1000); status_led = 0;
    LIB_SERIAL_Init();

    // B. HANDSHAKE
    char cmd = 0;
    while (cmd != 'G') {
        pc.write("READY", 5);
        if (pc.readable()) {
            pc.read(&cmd, 1);
        }
        wait_us(500 * 1000);
        status_led = !status_led;
    }
    status_led = 0;

    // C. INIT MODEL
    // Note: If 60KB is too large for your specific board's RAM,
    // try reducing to 40 * 1024.
    if (LIB_MODEL_Init(converted_model_tflite, &input, tensor_arena, kTensorArenaSize) != 0) {
        // Error: Fast Blink
        while(1) { status_led = !status_led; wait_us(100 * 1000); }
    }

    // Safety Check: Input pointer valid?
    if (input == nullptr) {
        while(1) { status_led = !status_led; wait_us(1000 * 1000); } // Slow Blink Error
    }

    // D. PROCESS LOOP
    while (true) {
        // 1. Receive Synced Frame (Waits for '$')
        Serial_ReceiveFrame((uint8_t*)input_buffer, NUM_INPUTS * sizeof(float));

        // 2. Normalize & Populate Input
        for (int i = 0; i < NUM_INPUTS; i++) {
            float normalized = (input_buffer[i] - TRAIN_MEAN) / TRAIN_STD;

            if (input->type == kTfLiteFloat32) {
                input->data.f[i] = normalized;
            }
            else if (input->type == kTfLiteInt8) {
                int32_t val = (normalized / input->params.scale) + input->params.zero_point;
                val = (val < -128) ? -128 : (val > 127 ? 127 : val);
                input->data.int8[i] = (int8_t)val;
            }
            else if (input->type == kTfLiteUInt8) {
                 int32_t val = (normalized / input->params.scale) + input->params.zero_point;
                 val = (val < 0) ? 0 : (val > 255 ? 255 : val);
                 input->data.uint8[i] = (uint8_t)val;
            }
        }

        // 3. Inference
        if (LIB_MODEL_Run(&output) != 0 || output == nullptr) {
            float err = -999.0f;
            LIB_SERIAL_Transmit(&err, 1, TYPE_F32);
            continue;
        }

        // 4. Read Output
        float raw_output = 0.0f;

        if (output->type == kTfLiteFloat32) {
            raw_output = output->data.f[0];
        }
        else if (output->type == kTfLiteInt8) {
            raw_output = (output->data.int8[0] - output->params.zero_point) * output->params.scale;
        }
        else if (output->type == kTfLiteUInt8) {
            raw_output = (output->data.uint8[0] - output->params.zero_point) * output->params.scale;
        }

        // 5. Final Calculation
        float predicted = (raw_output * TRAIN_STD) + TRAIN_MEAN;
        LIB_SERIAL_Transmit(&predicted, 1, TYPE_F32);

        status_led = 1; wait_us(50 * 1000); status_led = 0;
    }
}
```

test.py Code:

This code script acts as a hardware-in-the-loop test bench for validating a temperature prediction model running on a microcontroller. It begins by establishing a strict synchronization handshake via the sync_with_board function, which waits for the board to broadcast a "READY" signal before replying with a 'G' (Go) command to start the test. Once synchronized, the script simulates real-world sensor behavior by generating twenty sets of synthetic temperature data, consisting of five floating-point values slightly randomized around a base trend. For each test iteration, it transmits a synchronization byte followed by the binary-packed sensor data to ensure the hardware aligns its data buffer correctly. Finally, it captures the resulting 4-byte floating-point prediction sent back by the microcontroller, unpacks it into a Celsius value, and prints a side-by-side comparison of the inputs and the model's predicted temperature.

Code:

```
import serial
import struct
import time
import random

SERIAL_PORT = 'COM6'
BAUD_RATE = 115200

def sync_with_board(ser):
    print("Waiting for board... (Press Reset on Board)")
    while True:
        if ser.in_waiting:
            try:
                line = ser.read(ser.in_waiting)
                if b"READY" in line:
                    print("✅ Board Found! Sending 'GO'...")
                    ser.write(b'G')
                    time.sleep(0.5)
                ser.reset_input_buffer()
                return True
        except:
            pass
    time.sleep(0.1)

if __name__ == "__main__":
    try:
        ser = serial.Serial(SERIAL_PORT, BAUD_RATE, timeout=2)
        print(f"Connected to {SERIAL_PORT}")
    except:
        print("Check Port"); exit()

    sync_with_board(ser)

    print("-" * 50)
    print(f"{'Inputs':<35} | {'Pred':<6}")
    print("-" * 50)

    base_temps = [18.0, 18.2, 18.5, 18.9, 19.2]

    for i in range(20):
        # Generate Data
        inputs = [x + random.uniform(-0.5, 0.5) for x in base_temps]

        # --- FIX: Send Sync Byte '$' FIRST ---
        ser.write(b'$')
        # Send 5 Floats
        ser.write(struct.pack('<5f', *inputs))

        # Read Result
        raw_data = ser.read(4)

        if len(raw_data) == 4:
            pred = struct.unpack('<f', raw_data)[0]
            print(f"{str([round(x,1) for x in inputs]):<35} | {pred:.2f}° C")
        else:
            print("❌ Timeout (Sync Error?)")

        time.sleep(0.1)

    ser.close()
```

Results:

We applied the test file to the MCU board, but it timed out. The STM board resets itself every second. The reason for this situation is that the RAM usage is too high, so it resets itself over and over. We couldn't solve it.