Project 4: Real-Time Sensor-Agnostic Posture Detection

A. System Design

a. Motivation:

The goal of this project was to build upon previous work in lying posture tracking to create a complete, on-device machine learning system. The primary challenges were to design a system that was both sensor-agnostic (capable of making predictions using an accelerometer, gyroscope, or magnetometer) and orientation-insensitive (capable of identifying postures like side, supine, etc. regardless of sensor placement). This project covers the full embedded ML pipeline: data collection, offline model training, on-device conversion, and real-time inference with a base station.

b. High-Level Design:

My system is built in three main parts:

1. Data Collection: An Arduino sketch (data_collector.ino) was created to stream 9-axis IMU data (Accel, Gyro, Mag) at 50 Hz over serial monitor. This data was saved it into labeled .csv files.
2. Offline Training: A Google Colab notebook (posture_project.ipynb) was used to load, process, and train the model. This notebook handled data windowing, balancing, normalization, model training, and conversion.
3. Real-Time Deployment: A final Arduino sketch (real_time_inference.ino) was created to run the trained TFLite model. This sketch receives commands from a base_station.py script, collects 2 seconds of new sensor data, performs normalization and inference, and sends the predicted class back to the base station.

c. Key Design Parameters:

- Sampling Frequency: I chose a sampling frequency of 50 Hz (a 20ms period). This is a standard frequency for human activity recognition, as it is fast enough to capture meaningful motion signatures without generating an excessive amount of data.
- Window Size: I selected a 2-second (100-sample) window with a 50% (50-sample) overlap for data augmentation. This window size proved to be a good balance, providing enough data for the model to find patterns, which resulted in a 99.12% test accuracy.
- Model Architecture: A 1D Convolutional Neural Network (CNN) was chosen. This architecture is detailed in the following section.
- Training Parameters: The model was trained using the Adam optimizer, sparse_categorical_crossentropy loss, and a patience of 10 epochs for early stopping, running for a total of 100 epochs.

d. Deployment & Design Difficulties:

Deploying the model to the Arduino presented several technical challenges:

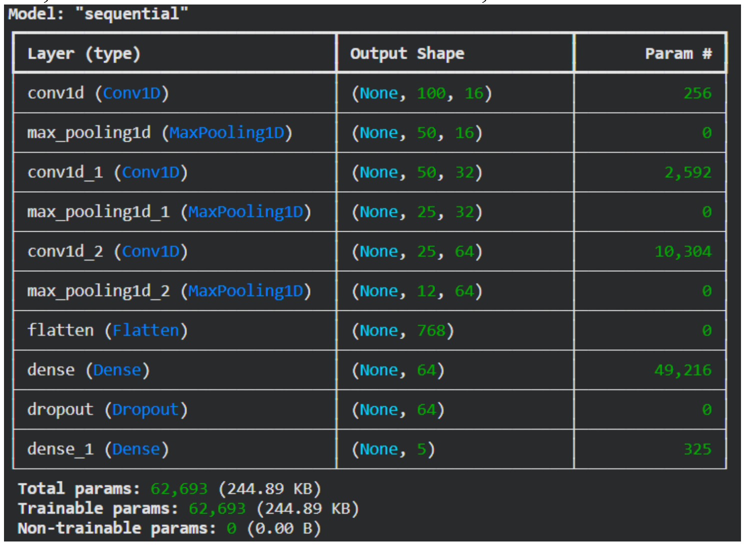
1. **Model Format:** My initial model, converted with converter.optimizations = [tf.lite.Optimize.DEFAULT], was in a "hybrid" format. The Arduino's TFLite library did not support this, leading to an AllocateTensors() failed error. I solved this by removing the optimization, re-converting the model as a pure 32-bit float model.
2. **RAM Overflow:** The new, un-optimized model was too large to fit in the Arduino's SRAM, causing a linker error (cannot move location counter backwards). I fixed this by manually editing the model.h file and adding the const keyword to the model array, which correctly stores the model in the board's larger Flash memory.
3. **Serial Communication Bug:** My initial base-station protocol was unreliable. Commands would get "queued up" or "desynchronized," causing timeouts. I solved this by implementing a robust token-based "handshake" protocol (detailed in Section C).
4. **TFLite Library Versioning:** The TFLite library provided (TensorflowLite_Library_export.zip) had different function definitions than some public examples. I fixed this by rolling back some previous version of tf_lite repository on git hub.

e. Deep Learning Architecture:

I designed a 1D Convolutional Neural Network (CNN). I chose this architecture for three specific reasons:

- Time-Series Specialization: 1D-CNNs are highly effective at finding local patterns and features within sequential data, making them a perfect fit for my 100-sample (2-second) time-series windows.
- Efficiency: CNNs are computationally lighter and require less memory than other sequence models like RNNs or LSTMs, making them ideal for conversion to TFLite and deployment on a resource-constrained microcontroller.
- Translation Invariance: A CNN can learn a posture's signature (e.g., the specific static pattern of "sitting") and recognize it no matter where it appears within the 2-second window.

My final architecture, as defined in the Colab notebook, is as follows:



Model: "sequential"

Layer (type)	Output Shape	Param #
conv1d (Conv1D)	(None, 100, 16)	256
max_pooling1d (MaxPooling1D)	(None, 50, 16)	0
conv1d_1 (Conv1D)	(None, 50, 32)	2,592
max_pooling1d_1 (MaxPooling1D)	(None, 25, 32)	0
conv1d_2 (Conv1D)	(None, 25, 64)	10,304
max_pooling1d_2 (MaxPooling1D)	(None, 12, 64)	0
flatten (Flatten)	(None, 768)	0
dense (Dense)	(None, 64)	49,216
dropout (Dropout)	(None, 64)	0
dense_1 (Dense)	(None, 5)	325

```
Total params: 62,693 (244.89 KB)
Trainable params: 62,693 (244.89 KB)
Non-trainable params: 0 (0.00 B)
```

Fig. 1. Final Architecture

B. Experiment

a. Experimental Procedure:
I used the data_collection.ino script to gather all experimental data. The board was held steady on a desk to simulate the various static postures. For each scenario, I collected two trials of at least one minute each, resulting in over 23 data files.

b. Addressing Orientation-Independence
This was the most important experimental design. To make the model robust, I collected data for all static classes in multiple orientations:

- **Supine/Prone/Sitting:** Data was collected with the USB port facing UP and with the USB port facing DOWN.
- **Side**: Data was collected for both Left Side and Right Side, and in both USB UP and USB DOWN orientations.
  In the data processing step, all "side" variations (left_side_usb_up, right_side_usb_down, etc.) were given the same label (2). This variance in the training data forces the model to learn that the Y-axis (ay) being +1.0 or -1.0 both mean "side," thus achieving orientation independence.

c. 'Unknown' Class Experiments
To create a meaningful unknown class (Label 4), I performed experiments that were not static postures. This class is designed to catch any "none of the above" motions. My experiments included:

- unknown_walking: Holding the board and walking, which introduces small, periodic motion.
- unknown_shaking: Gently shaking the board, creating high-frequency, noisy data.
- unknown_transition: Slowly rolling the board from a supine to a side position.

The gyroscope data (gx, gy, gz) was essential for this class. For all static postures, the gyro data is near zero or very minute difference can be observed. For the unknown class, the gyro data is non-zero. The neural network learned this pattern perfectly.

C. Algorithm

a. Machine Learning Algorithm
My machine learning algorithm is a 1D Convolutional Neural Network (CNN). The specific architecture and my justification for choosing it are detailed in Section A: Deep Learning Architecture.

b. Training Process & Sensor-Agnostic Design:
The training process was the key to achieving the sensor-agnostic requirement.

1. Load & Window: All 9-axis .csv files were loaded and sliced into 100-sample windows.
2. Balance: To prevent bias from having more side data, I undersampled all classes to match the count of the smallest class.
3. Normalize (The Key): I calculated the mean and std for each axis independently for each sensor type. This resulted in 18 unique values (3 axes * 3 sensors * 2 stats)

which I saved in normalization_params.json.

4. Combine: I created three normalized datasets (X_accel, X_gyro, X_mag) and stacked them into one large X_final dataset. The y_labels were also tripled to match.
5. Train: I trained my single 1D-CNN on this combined, normalized dataset. This taught the model to associate a posture (e.g., "sitting", "supine", etc.) with the normalized patterns from all three different sensor types.
6. Deploy: The Arduino code stores these 18 normalization values. When the base station requests a prediction with a specific sensor (e.g. 2 for Gyro), the Arduino collects 2 seconds of Gyro data and only applies the GYRO_MEAN and GYRO_STD values before feeding it to the model. Similarly for any other requested sensor.

c. Base-Station Coordination

To solve the serial synchronization bugs, I implemented a robust, token-based handshake protocol:

1. Python Host: Sends a full-line token command, (e.g., SENSOR:ACCEL\n or 2\n).
2. Arduino MCU: Uses Serial.readStringUntil('\n') to safely read the entire command.
3. Arduino MCU: Parses the command (parse_command) and immediately echoes its understanding back to the host (e.g., CMD:ACCEL).
4. Arduino MCU: Enters the perform_inference function, which takes ~2.5 seconds (collecting, normalizing, inferring).
5. Arduino MCU: When finished, it sends two lines: the prediction (e.g., 4) and a "ready" token (Prediction sent. READY).
6. Python Host: Sits in a while loop, reading all lines until it sees the "READY" token. Only then does it break the loop, display the final prediction, and show the menu for the next command. This ensures the host and MCU are always in sync.

D: Results

a. Model Performance:

The model performed exceptionally well, achieving a final offline test accuracy of 99.12%. The training and validation plots below show a healthy training process. The accuracy and validation-accuracy lines track each other closely, as do the loss lines, proving that the model is generalizing well and not overfitting the training data. The use of EarlyStopping with patience=10 and epochs=100 ensured that the notebook saved the model at its peak performance.

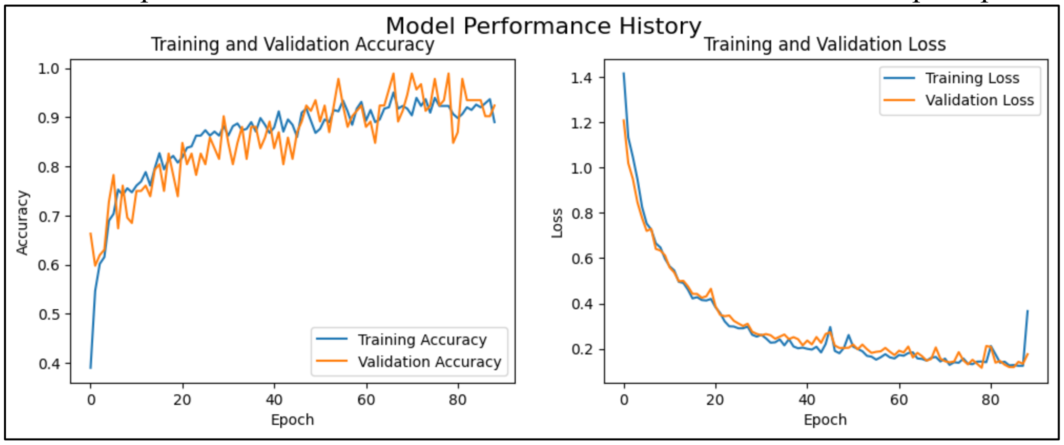
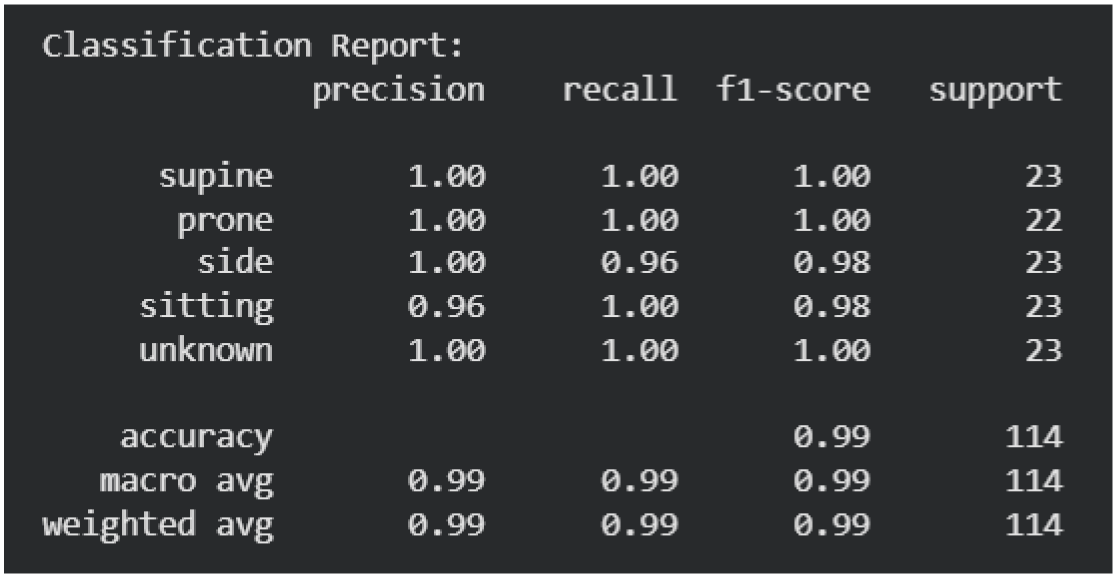


Fig.2. Model training and validation accuracy/loss over epochs

b. Test Data Analysis:

The final test set accuracy was 99.12%. The classification report and confusion matrix below provide a detailed look at the model's performance on data it had never seen.

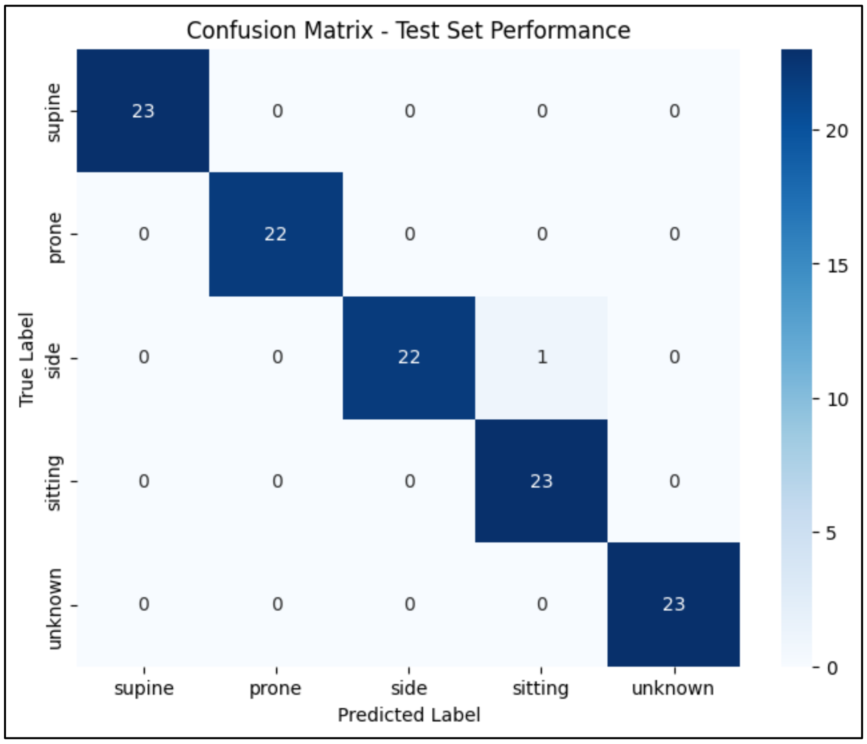


```
Classification Report:
              precision    recall  f1-score   support

      supine       1.00      1.00      1.00        23
       prone       1.00      1.00      1.00        22
        side       1.00      0.96      0.98        23
     sitting       0.96      1.00      0.98        23
     unknown       1.00      1.00      1.00        23

    accuracy                           0.99       114
   macro avg       0.99      0.99      0.99       114
weighted avg       0.99      0.99      0.99       114
```

Fig.3. Classification Report



	supine	prone	side	sitting	unknown
supine	23	0	0	0	0
prone	0	22	0	0	0
side	0	0	22	1	0
sitting	0	0	0	23	0
unknown	0	0	0	0	23

Fig. 4. Confusion Matrix of the final model on the test set.

- **Analysis:** The model's performance on the test set is nearly perfect.
  - Strong Classes: supine, prone, and unknown all achieved perfect 1.00 F1-scores. This shows the model can perfectly differentiate these postures.

- Minor Confusion: The only misclassifications were minimal:
    - Side: One side sample was misclassified (0.96 recall).
    - Sitting: One other sample was misclassified as sitting (0.96 precision).
- Conclusion: An overall accuracy of 99.12% and a macro F1-score of 0.99 demonstrate a highly robust and production-ready model. The tuning of patience=10 and epochs=100 was highly effective.

c. Real-time vs. Offline Prediction

The real-time performance demonstrated in the submission video aligns perfectly with the 99.12% offline test accuracy. During the demo, the system proved to be fully robust and met all design goals:

1. Posture Identification: It correctly identified all static postures (supine, prone, sitting).
2. Orientation-Insensitivity: It successfully demonstrated this by classifying both left_side and right_side as "SIDE", and also worked flawlessly regardless of the sensor's USB-port orientation (up, down, left, or right).
3. Sensor-Agnosticism: It demonstrated comprehensive sensor-agnostic capabilities. I was able to get accurate predictions for all postures using all three distinct sensors: the Accelerometer (command 1), the Gyroscope (command 2), and the Magnetometer (command 3).
4. Robustness: It correctly rejected dynamic, non-posture motion by classifying "shaking" as "UNKNOWN".

E: Discussions

Summary of Results:

This project was a complete success. I designed and deployed an end-to-end system that achieved an outstanding 99.12% test accuracy for posture detection. The final system correctly implemented both sensor-agnostic and orientation-insensitive features and proved to be robust and reliable during live tests.

Difficulties in Design:

The two main design challenges were sensor-agnosticism and orientation-insensitivity.

- Sensor-Agnostic: The difficulty is that the Accelerometer, Gyroscope, and Magnetometer have completely different physical meanings, units, and data ranges. A model trained on raw accelerometer data would be useless for gyroscope data. I solved this by creating a "universal" model. I trained it on a combined dataset of all three sensor types, where each type was independently normalized first. This taught the model to recognize the normalized patterns for each posture, allowing the Arduino to apply the correct normalization at runtime and use the same model for all three sensors.
- Orientation-Insensitive: The difficulty is ensuring a posture is recognized regardless of the sensor's specific placement on the chest. A user might wear the device "upside down" (USB port down vs. up). To solve this, I used a comprehensive data collection strategy: for all static postures (supine, prone, sitting, left_side, and right_side), I collected data with the sensor in all two orientations (USB up, down). Furthermore, I mapped both left_side and right_side data to the single side class. This robust "data augmentation" approach forced the model to learn the underlying gravitational and magnetic signatures of each posture, making it truly insensitive to the sensor's placement.

Most Difficult Part of the Project:
The most difficult part by far was not the machine learning (which achieved 99% accuracy), but the TFLite setup and on-device deployment. This phase was a multi-step debugging process:

- Library & Header Mismatch: My first attempts failed because I was including the wrong headers. The specific library provided (TensorflowLite_Library_export.zip) required a full set of headers (like .../all_ops_resolver.h, .../micro_interpreter.h, etc.), not just TensorFlowLite.h..
- Model Format Incompatibility: My first model conversion in Colab used tf.lite.Optimize.DEFAULT. This created a "hybrid" model that the Arduino's TFLite library could not handle, resulting in a runtime Hybrid models are not supported and AllocateTensors() failed error. I had to re-run my Colab pipeline without this optimization line to generate a pure 32-bit float model.
- SRAM Overflow (Linker Error): The new, un-optimized 32-bit float model was too large to fit in the Arduino's limited SRAM. This caused a cannot move location counter backwards linker error. The solution was to manually edit the model.h file and add the const keyword to the model array (const unsigned char model_tflite[]...) and its length. This correctly placed the large model into the board's Flash memory instead of SRAM.
- Serial Protocol: Finally, my initial simple serial protocol was unreliable and suffered from read timeouts and command "desyncing." I had to re-architect both the Python and Arduino code to implement a proper, token-based "handshake" protocol (as detailed in Section C) to achieve the smooth, reliable communication seen in the final demo.

Future Improvements:
Given the 99.12% accuracy, the model is excellent and effectively "solved" for this dataset. The confusion matrix shows only minimal, non-systematic errors. Future improvements would move beyond this dataset and focus on real-world robustness:

1. **More unknown data:** I would add more complex transition movements (like rolling over, or sitting-to-standing) to the unknown class to make the model even more robust at rejecting non-static postures.
2. **On-body testing:** The model could be further tested by collecting data from multiple users with different body types to ensure it generalizes well across a wider population.

Real-time Prediction Accuracy:
The real-time prediction was just as accurate as I expected from my **99.12% test score**. It performed flawlessly during the demo video, correctly identifying all postures, rejecting unknown motion, and successfully proving both the orientation-insensitive and sensor-agnostic capabilities.

Conclusion

This project was a comprehensive success, demonstrating a complete end-to-end workflow for an embedded machine learning system. I successfully designed, trained, and deployed a 1D Convolutional Neural Network that achieved an outstanding 99.12% accuracy on the test set.

The system fully met the core project requirements. The sensor-agnostic design was proven effective, with the single model correctly identifying postures using data from the accelerometer, gyroscope, and magnetometer. The orientation-insensitive design was also validated, as the model correctly classified all postures regardless of the sensor's physical placement and identified both "left_side" and "right_side" as the general "side" class.

The most significant challenges were not in machine learning, but in the on-device deployment. The final system is robust, highly accurate, and serves as an excellent foundation for any future real-world posture-monitoring application.

References

1. Course Materials: "EML - Week 4 - 03 - Model Deployment.pptx". CEN 598: Embedded Machine Learning, Arizona State University.
2. Tensorflow Lite Micro on an Arduino (https://github.com/tensorflow/tflite-micro-arduino-examples)
3. TensorFlow/Keras: Sequential, Dense, Dropout, EarlyStopping - Used for building, training, and regularizing the neural network. (https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping)
4. NumPy: numpy.random.normal, numpy.concatenate, numpy.argmax - Used for adding noise, combining data, and processing model predictions. (https://numpy.org/doc/2.1/reference/random/generated/numpy.random.normal.html)
5. Scikit-learn: train_test_split, StandardScaler, classification_report, confusion_matrix - Used for robust dataset splitting, feature scaling, and final model evaluation. (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html, https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html)
6. Seaborn: seaborn.heatmap - Used for visualizing the final confusion matrix. (https://seaborn.pydata.org/generated/seaborn.heatmap.html)
7. Assistance from generative AI models was used for debugging complex C++ and Python errors.
8. Pandas: pandas.read_csv - Used for loading and parsing all .csv files.

Final Source Code

1. data_collection.ino

```
/*
 * PROJECT 4 - 9-AXIS DATA COLLECTION SCRIPT
 *
 * This program reads all 9 axes from the onboard IMU
 * (BMI270 Accelerometer, BMI270 Gyroscope, BMM150 Magnetometer)
 * and prints the data to the Serial Monitor in CSV format at 50 Hz.
 */
#include "Arduino_BMI270_BMM150.h"
// Define our sampling period. 1000ms / 20ms = 50 Hz.
const int SAMPLING_PERIOD_MS = 20;
void setup() {
  Serial.begin(115200); // Use a fast baud rate
  while (!Serial);
  // Initialize the IMU (Accel, Gyro, Mag)
  if (!IMU.begin()) {
    Serial.println("Failed to initialize IMU!");
    while (1);
  }
  Serial.println("IMU initialized. Starting data collection...");
  // Print the 9-column CSV header
  Serial.println("ax,ay,az,gx,gy,gz,mx,my,mz");
}
void loop() {
  float ax, ay, az; // Accelerometer
  float gx, gy, gz; // Gyroscope
  float mx, my, mz; // Magnetometer
  // Check if all data is available
  if (IMU.accelerationAvailable() &&
      IMU.gyroscopeAvailable() &&
      IMU.magneticFieldAvailable())
  {
    // Read all 9 axes
    IMU.readAcceleration(ax, ay, az);
    IMU.readGyroscope(gx, gy, gz);
    IMU.readMagneticField(mx, my, mz);
    // Print the data as a single CSV line
    Serial.print(ax); Serial.print(",");
    Serial.print(ay); Serial.print(",");
    Serial.print(az); Serial.print(",");
    Serial.print(gx); Serial.print(",");
    Serial.print(gy); Serial.print(",");
    Serial.print(gz); Serial.print(",");
    Serial.print(mx); Serial.print(",");
    Serial.print(my); Serial.print(",");
    Serial.println(mz);
  }

  // Wait for the next sampling interval
  delay(SAMPLING_PERIOD_MS);
}
```

2. real_time_inference.ino

```
/*
 * REAL-TIME POSTURE DETECTION (strict command-driven, timeout-safe)
 */

#include <Arduino.h>

// --- TFLite Core Headers ---
#include <TensorFlowLite.h>
#include <tensorflow/lite/micro/all_ops_resolver.h>
#include <tensorflow/lite/micro/micro_error_reporter.h>
#include <tensorflow/lite/micro/micro_interpreter.h>
#include <tensorflow/lite/schema/schema_generated.h>

// --- Project-Specific Headers ---
#include "model.h"
#include "Arduino_BMI270_BMM150.h"

// --- TFLite Globals ---
tflite::MicroErrorReporter tflErrorReporter;
tflite::AllOpsResolver    tflOpsResolver;
const tflite::Model*      tflModel = nullptr;
tflite::MicroInterpreter* tflInterpreter = nullptr;
TfLiteTensor*             tflInputTensor = nullptr;
TfLiteTensor*             tflOutputTensor = nullptr;

// --- Memory Arena ---
constexpr int kTensorArenaSize = 15 * 1024;
```

```
uint8_t tensor_arena[kTensorArenaSize];

// --- Project Constants ---
const int SAMPLING_FREQUENCY_HZ = 50;
const int SAMPLING_PERIOD_MS    = 1000 / SAMPLING_FREQUENCY_HZ; // 20ms
const int WINDOW_SIZE           = 100; // 2 seconds @ 50Hz
const int NUM_CLASSES           = 5;

// Buffer to hold one window of data (100 samples, 3 axes)
float window_buffer[WINDOW_SIZE][3];

// --- Normalization Constants from normalization_params.json file ---
const float ACCEL_MEAN[] = { 0.001600526, -0.017278421, -0.021210526 };
const float ACCEL_STD[]  = { 0.525893116, 0.519990200, 0.707250449 };

const float GYRO_MEAN[]  = { -1.301315789, 0.011415263, 0.051266315 };
const float GYRO_STD[]   = { 48.513148771, 56.630698703, 40.677223375 };

const float MAG_MEAN[]   = { -30.665157894, 33.622736842, -5.371315789 };
const float MAG_STD[]    = { 100.036291765, 69.151219412, 82.474813816 };

// --- Protocol helpers ---
enum SensorSel { SEL_NONE = 0, SEL_ACCEL = 1, SEL_GYRO = 2, SEL_MAG = 3 };

static inline void clear_serial_rx() {
  while (Serial.available() > 0) (void)Serial.read();
}

static inline void clear_window_buffer() {
  for (int i = 0; i < WINDOW_SIZE; ++i) {
    window_buffer[i][0] = 0.f;
    window_buffer[i][1] = 0.f;
    window_buffer[i][2] = 0.f;
  }
}

static inline SensorSel parse_command_line(const String& cmd) {
  if (cmd == "1" || cmd == "SENSOR:ACCEL") return SEL_ACCEL;
  if (cmd == "2" || cmd == "SENSOR:GYRO")  return SEL_GYRO;
  if (cmd == "3" || cmd == "SENSOR:MAG")   return SEL_MAG;
  return SEL_NONE;
```

```
}

static void echo_cmd(SensorSel sel) {
  Serial.print("CMD:");
  if (sel == SEL_ACCEL)      Serial.println("ACCEL");
  else if (sel == SEL_GYRO)  Serial.println("GYRO");
  else if (sel == SEL_MAG)   Serial.println("MAG");
  else                       Serial.println("UNKNOWN");
}

static bool read_one_sample(SensorSel sel, float &x, float &y, float &z) {
  // Blocking, waits until the selected sensor reports new data
  if (sel == SEL_ACCEL) {
    while (!IMU.accelerationAvailable()) {}
    IMU.readAcceleration(x, y, z);
    return true;

  } else if (sel == SEL_GYRO) {
    while (!IMU.gyroscopeAvailable()) {}
    IMU.readGyroscope(x, y, z);
    return true;

  } else { // SEL_MAG
    while (!IMU.magneticFieldAvailable()) {}
    IMU.readMagneticField(x, y, z);
    return true;
  }
}
static bool collect_window(SensorSel sel) {
  Serial.print("Collecting ");
  if (sel == SEL_ACCEL)      Serial.print("Accel");
  else if (sel == SEL_GYRO)  Serial.print("Gyro");
  else if (sel == SEL_MAG)   Serial.print("Mag");
  else return false;
  Serial.println(" data for 2 seconds...");

  bool warned = false;
  unsigned long next_tick = millis(); // pacing anchor

  for (int i = 0; i < WINDOW_SIZE; i++) {
    // Pace: target every 20 ms; catch up if late
    next_tick += SAMPLING_PERIOD_MS;

    float sx = 0, sy = 0, sz = 0;
    bool fresh = read_one_sample(sel, sx, sy, sz);
    if (!fresh && !warned) {
      Serial.println("Note: sensor timeout on some samples (filling zeros).");
      warned = true;
    }
    window_buffer[i][0] = sx;
    window_buffer[i][1] = sy;
    window_buffer[i][2] = sz;

    long now = (long)millis();
    long wait_ms = (long)next_tick - now;
    if (wait_ms > 0) delay((unsigned long)wait_ms);
  }

  Serial.println("Collection complete. Normalizing...");
  return true;
}

static void normalize_fill_input(SensorSel sel) {
  for (int i = 0; i < WINDOW_SIZE; i++) {
    for (int j = 0; j < 3; j++) {
      float val;
      if (sel == SEL_ACCEL)      val = (window_buffer[i][j] - ACCEL_MEAN[j]) /
ACCEL_STD[j];
      else if (sel == SEL_GYRO)  val = (window_buffer[i][j] - GYRO_MEAN[j])  /
GYRO_STD[j];
      else /* SEL_MAG */         val = (window_buffer[i][j] - MAG_MEAN[j])   / MAG_STD[j];
      tflInputTensor->data.f[i * 3 + j] = val;   // input is [1,100,3] flattened
    }
  }
}

static void run_once(SensorSel sel) {
  // Start from a clean slate each cycle
  clear_serial_rx();          // ignore any typed-ahead input
  clear_window_buffer();      // clear previous samples

  echo_cmd(sel);
  if (!collect_window(sel)) {
    Serial.println("Collection failed");
    Serial.println("READY");
    return;
  }

  Serial.println("Running inference...");
  normalize_fill_input(sel);

  if (tflInterpreter->Invoke() != kTfLiteOk) {
    Serial.println("Invoke failed");
    Serial.println("READY");
    return;
  }

  // Argmax over NUM_CLASSES
  int   predicted_label = 0;
  float max_score       = tflOutputTensor->data.f[0];
  for (int i = 1; i < NUM_CLASSES; i++) {
    float s = tflOutputTensor->data.f[i];
    if (s > max_score) { max_score = s; predicted_label = i; }
  }

  // Send 1..5 to host (compat with Python)
  Serial.println(predicted_label + 1);
  Serial.println("READY");
}

void setup() {
  Serial.begin(115200);
  Serial.setTimeout(2000);
  while (!Serial) {}

  if (!IMU.begin()) {
    Serial.println("Failed to initialize IMU!");
    while (1) {}
  }

  // --- TFLite setup ---
  tflModel = tflite::GetModel(model_tflite);
  if (tflModel->version() != TFLITE_SCHEMA_VERSION) {
    Serial.println("Model schema mismatch!");
    while (1) {}
  }

  tflInterpreter = new tflite::MicroInterpreter(
    tflModel, tflOpsResolver, tensor_arena, kTensorArenaSize, &tflErrorReporter
  );
  if (tflInterpreter->AllocateTensors() != kTfLiteOk) {
    Serial.println("AllocateTensors() failed");
    while (1) {}
  }

  tflInputTensor  = tflInterpreter->input(0);
  tflOutputTensor = tflInterpreter->output(0);

  clear_window_buffer();
  clear_serial_rx();

  Serial.println("Ready");
  Serial.println("Send 1 (ACCEL), 2 (GYRO), or 3 (MAG).");
}
void loop() {
  // STRICT IDLE → RUN-ONCE → IDLE behavior
  if (Serial.available() > 0) {
    String cmd = Serial.readStringUntil('\n');
    cmd.trim();
    if (cmd.length() == 0) return;

    SensorSel sel = parse_command_line(cmd);
    if (sel == SEL_NONE) {
      Serial.println("Unknown command. Use 1 | 2 | 3.");
      Serial.println("READY");
      return;
    }
    // From here until we print READY again, we do not look at Serial.
    run_once(sel);
  }
}
```

3. posture_project.ipynb