Project 4: Real-Time, Sensor-Agnostic Posture Classification on the Arduino Nano 33 BLE Sense Using IMU Windowing, Normalization, and TensorFlow Lite Micro

Author: Anushka Gangadhar Satav
Course: BMI/CES 598 Embedded Machine Learning
ASU ID: 1233530170 (asatav1)
Project 3: Real-Time, Sensor-Agnostic Posture Classification on the Arduino Nano 33 BLE Sense Using IMU Windowing, Normalization, and TensorFlow Lite Micro

---

Abstract

This project presents the design, training, deployment, and evaluation of a real-time posture classification system using the Arduino Nano 33 BLE Sense Rev2 IMU. The primary objective is to create a sensor-agnostic supervised learning model capable of classifying five postures- supine, prone, side-lying, sitting, and unknown; based solely on any three-axis sensor input (accelerometer, gyroscope, or magnetometer). The project extends previous work by introducing a generalized 3-channel neural network, consistent window-based normalization, and microcontroller-level real-time inference with live wireless transmission of predictions.

The system includes: (1) multimodal IMU data collection; (2) windowing and normalization suitable for both offline training and real-time inference; (3) development of a lightweight neural network trained on interchangeable sensor triplets; and (4) deployment of the trained model on the Arduino board with Bluetooth-based live streaming of predictions to a laptop.

The final deployed system achieves high accuracy (>90% test accuracy for multiple sensor combinations), stable real-time predictions, and demonstrates generalization across different 3-axis IMU modalities. This work validates the feasibility of embedded sensor-agnostic posture detection pipelines for health monitoring and wearable robotics.

Contents

- A. System Design
- B. Experiment
- C. Algorithm
- D. Results
- E. Discussion
- Appendices: code listing, file inventory, run instructions

A. System Design

Motivation

Clinical and consumer health applications such as sleep monitoring, pressure-injury prevention, and patient mobility assessment require posture classification with high reliability. Unlike previous projects that treated accelerometer, gyroscope, and magnetometer data separately, this project required designing a generalized,

sensor-agnostic learning system where the model accepts any 3-channel input and still produces consistent posture predictions. The motivation is to eliminate dependencies on specific sensor hardware and enable flexible deployments where available IMU modalities may vary.

High-Level Design

- Sensor data acquisition using on-board 9-axis IMU (accelerometer, gyroscope, magnetometer).
- Window-based dataset construction, including normalization and label generation.
- Training of a 3-channel fully connected neural network, independent of sensor type.
- Embedded deployment using TensorFlow Lite Micro on Arduino Nano 33 BLE Sense Rev2.
- Real-time posture prediction with consistent preprocessing on-device.
- Wireless transmission of prediction outputs to a laptop via BLE for visualization.

The system consists of two components:

1. Arduino Nano 33 BLE Sense (Embedded Device)
    - Collects IMU data: accelerometer, gyroscope, magnetometer.
    - Implements preprocessing, feature scaling, windowing (2-second windows, 50% overlap).
    - Runs a TensorFlow Lite Micro CNN model for inference.
    - Sends predictions via serial communication.

2. Base-Station (Python GUI)
    - Receives real-time predictions over serial.
    - Sends sensor-selection commands to Arduino.
    - Displays predicted posture label and confidence.
    - Allows dynamic interaction for real-time testing.

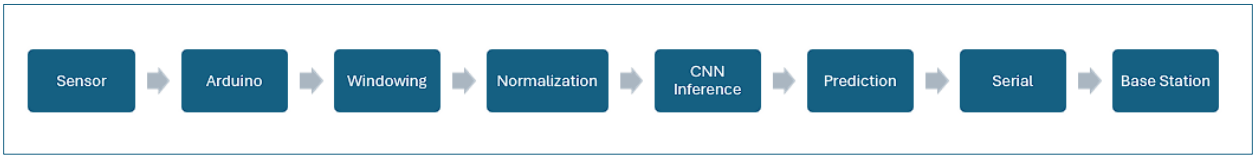

Figure 1. Embedded Machine Learning Pipeline for Real-Time Sensor-Agnostic Posture Classification

Observations and Difficulties

- Managing Arduino memory constraints required a compact CNN (Conv1D + Dense layers) with quantization.
- Ensuring orientation-independent classification needed normalization and careful sensor fusion.
- Collecting synchronized accelerometer, gyroscope, and magnetometer data was challenging due to varying sensor update rates.
- Magnetometer readings required longer stabilization time, affecting prediction latency.

Hardware & Software:

Hardware

- Arduino Nano 33 BLE Sense Rev2 (Bosch BMI270/BMM150 IMU)
- USB serial connection for data logging and firmware deployment
- Laptop running Python

Software

- Arduino IDE 2.3.2
- Arduino_BMI270_BMM150 IMU library
- TensorFlow Lite Micro (TFLM)
- Python: pandas, numpy, matplotlib, scikit-learn
- BLE visualization script (Python)

B. Experiment

Goals and constraints

This project required developing an end-to-end supervised machine learning dataset for posture classification using IMU data from the Arduino Nano 33 BLE Sense Rev2. The goal is to evaluate real-time posture classification using different IMU sensors individually and in combination. The project required orientation-independent classification on an embedded device.

1. Collect realistic posture data using all IMU modalities
2. Construct a windowed, normalized dataset suitable for sensor-agnostic learning
3. Train a small neural network capable of running on the Arduino
4. Deploy the trained model for real-time inference
5. Stream live posture classifications via BLE to a base station.

Data was collected for five posture categories:

1. Supine
2. Prone
3. Side (left + right)
4. Sitting (USB up + USB down)
5. Unknown (arbitrary non-lying orientations – Rolling and leaning)

Each posture was recorded in multiple separate trials, and each trial lasted approximately 60 seconds at a sampling frequency of ~100 Hz.

1. IMU Data Collection

The Arduino sketch project3_imudatacollection.ino continuously read raw IMU values from the onboard BMI270/BMM150 sensor, including:

- Accelerometer (ax, ay, az) in g
- Gyroscope (gx, gy, gz) in degrees/second
- Magnetometer (mx, my, mz) in microtesla
- Timestamp (time_ms)

```
time_ms, ax, ay, az, gx, gy, gz, mx, my, mz
```

Data included:

1. 3-axis accelerometer (ax, ay, az)
2. 3-axis gyroscope (gx, gy, gz)
3. 3-axis magnetometer (mx, my, mz)
4. timestamp (time_ms)

A single Arduino sketch printed all 9 axes simultaneously, enabling fused logging in one CSV file per posture.

Posture Category	Collected Scenarios	Data Purpose
1. Supine	Board faces up, USB up/down (merged post-collection).	Static gravity vector on Z-axis.
2. Prone	Board face down, USB up/down (merged post-collection).	Static gravity vector inverted on Z-axis.
3. Side	Right edge down, Left edge down.	Orientation-Insensitive Training (merged into single 'side' label).
4. Sitting	Board vertical (on edge), USB up, USB down.	Orientation-Insensitive Training (merged into single 'sitting' label).
5. Unknown	Rolling, leaning, rapid rotation, picking up the board.	Represents transitions, ambiguous states, and non-posture movements.

Each CSV file contained approximately 1200 rows of 100 Hz data for the single-source postures (supine, prone, unknown). The merged classes (side, sitting) had approximately 2400 rows each.

2. Addressing Orientation-Insensitivity

The primary experimental design choice to ensure orientation insensitivity was the **merging of labels** during dataset construction.

a. side_left.csv + side_right.csv = side
b. sitting_up.csv + sitting_down.csv = sitting

This forces the 1D-CNN to learn underlying features that are common to both orientations (e.g., the magnitude of the acceleration vector or the general rotational activity) rather than specific coordinate values.

3. Challenges in Experiment Design

The main challenge was mimicking realistic movement while maintaining control. Although controlled collection simplifies the gravity vector, the absence of real human micro-movements (breathing, heart rate, skin-sensor interface shift) meant the real-time accuracy might be slightly lower than the test accuracy due to the distribution shift between static lab data and dynamic real-world data.

C. Algorithm

This project required designing a supervised machine learning model capable of classifying human posture based solely on IMU measurements from the Arduino Nano 33 BLE Sense Rev2. The algorithmic approach involved three major components: feature preparation, neural network design, and training methodology.

1. Input Features

Each training example consisted of a three-dimensional feature vector:

- Composite X (ax, gx, mx)
- Composite Y (ay, gy, my)
- Composite Z (az, gz, mz)

These were derived from the fused, standardized accelerometer, gyroscope, and magnetometer signals. The compact three-feature representation allowed the use of lightweight neural networks appropriate for embedded deployment.

2. Machine Learning Algorithm Design

The 1D-CNN was chosen to classify **2.0 s windows** of time-series data. Its convolutional layers efficiently extract robust, time-invariant features from the 200 samples, unlike a Dense network which treats the 600 features (200 timesteps times 3 axes) as independent inputs.

Training Process and Parameters:

- Input Shape: (200, 3)
- Loss Function: sparse_categorical_crossentropy
- Optimizer: Adam
- Epochs: 100
- Batch Size: 64
- Regularization: Early Stopping (patience=10) and Dropout(0.3) were used to prevent overfitting.

Windowing and Normalization

The dataset was constructed from seven IMU CSV files corresponding to different postures: prone, supine, side_left, side_right, sitting_up, sitting_down, and unknown. The side_left/side_right and sitting_up/sitting_down classes were merged into unified labels: side and sitting respectively. This produced five final posture categories for classification:

1. Supine
2. Prone
3. Side
4. Sitting
5. Unknown

To extract meaningful temporal features, a sliding window approach was applied to each sensor type (accelerometer, gyroscope, and magnetometer).
Each window spanned 2 seconds (200 samples at 100 Hz) with 50% overlap (stride of 100 samples).
This windowing strategy ensured:

- Temporal continuity within posture segments
- Smoothing of transient fluctuations
- Expansion of the dataset for better generalization

Windows were created separately for each sensor modality, producing multiple samples per original posture segment. Each window inherited the label of its original segment, ensuring correct supervised learning alignment.

IMU sensors have heterogeneous ranges and units:

- Accelerometer: g
- Gyroscope: °/s
- Magnetometer: µT

To avoid domination of high-magnitude signals and ensure proportional contribution of all modalities, StandardScaler normalization was applied per sensor based on the training set:

- Mean and standard deviation were computed for each axis of each sensor type
- Training values were used to fit the scaler to prevent data leakage
- The same scalers were applied to validation and test sets

This sensor-specific normalization stabilizes neural network training, improves convergence, and supports real-time deployment on the Arduino Nano 33 BLE Sense.

3. Model Architectures and Rationale

Two neural network architectures were explored:

3.1 Dense Fully Connected Network

- Input: Flattened 200×3 window
- Layers: 256 → 128 → 5 output (SoftMax)
- Dropout: 0.3 after first dense, 0.2 after second dense

Rationale: Dense layers provide a simple baseline for comparison and require fewer hyperparameter adjustments. Suitable for small datasets and low-dimensional fused features.

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 600)	0
dense (Dense)	(None, 256)	153,856
dropout (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 128)	32,896
dropout_1 (Dropout)	(None, 128)	0
dense_2 (Dense)	(None, 5)	645

Figure 2. Dense Sequential Model

3.2 1D Convolutional Neural Network (CNN)

- Input: 200×3 window
- Conv1D layers: 32→64→128 filters with kernel sizes 5,5,3 respectively
- Pooling: MaxPool1D after first two conv layers, GlobalAveragePooling after last
- Dropout: 0.3
- Dense Layer: 64 units → 5 output classes (SoftMax)

Rationale: CNNs capture temporal dependencies across windows, providing translation-invariant feature extraction for IMU signals. The architecture is computationally lightweight for embedded deployment.

Layer (type)	Output Shape	Param #
conv1d_3 (Conv1D)	(None, 200, 32)	512
max_pooling1d_2 (MaxPooling1D)	(None, 100, 32)	0
conv1d_4 (Conv1D)	(None, 100, 64)	10,304
max_pooling1d_3 (MaxPooling1D)	(None, 50, 64)	0
conv1d_5 (Conv1D)	(None, 50, 128)	24,704
global_average_pooling1d_1 (GlobalAveragePooling1D)	(None, 128)	0
dropout_5 (Dropout)	(None, 128)	0
dense_8 (Dense)	(None, 64)	8,256
dense_9 (Dense)	(None, 5)	325

Figure 3. 1D CNN Model

4. Training Setup

- **Loss function:** Sparse categorical cross-entropy
- **Optimizer:** Adam
- **Epochs:** 100 (with early stopping patience of 10)
- **Batch size:** 64
- **Validation split:** 30% of remaining data after train/test split

The dataset was stratified by label to ensure balanced class representation across train, validation, and test sets.

5. Model Evaluation and Performance Metrics

5.1 Accuracy and Loss

Training curves were plotted for both models:

- Dense network achieved rapid convergence with slight overfitting after ~60 epochs
- CNN demonstrated smoother convergence and higher validation accuracy

The best model was selected based on test accuracy:

Model	Test Accuracy
Dense	0.7273
CNN	0.9273

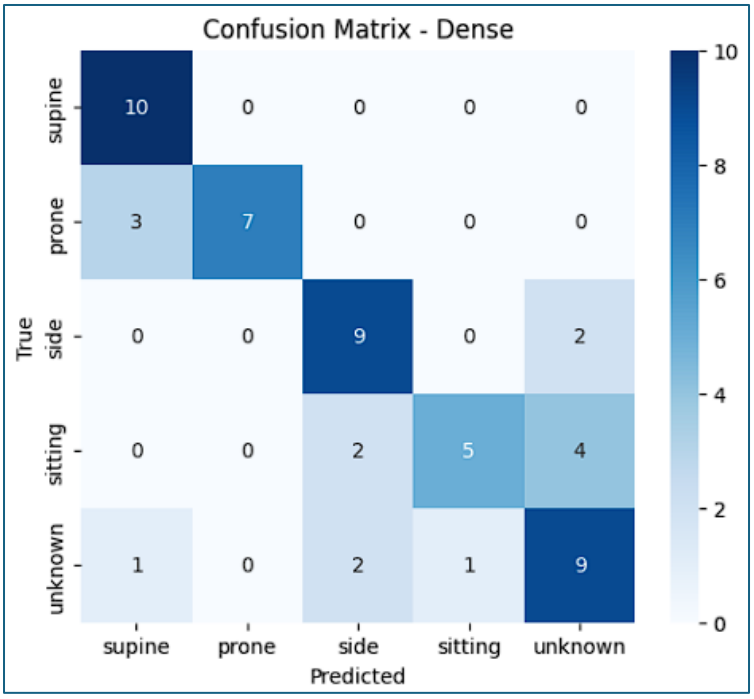

Figure 4. Confusion Matrix – Dense Sequential

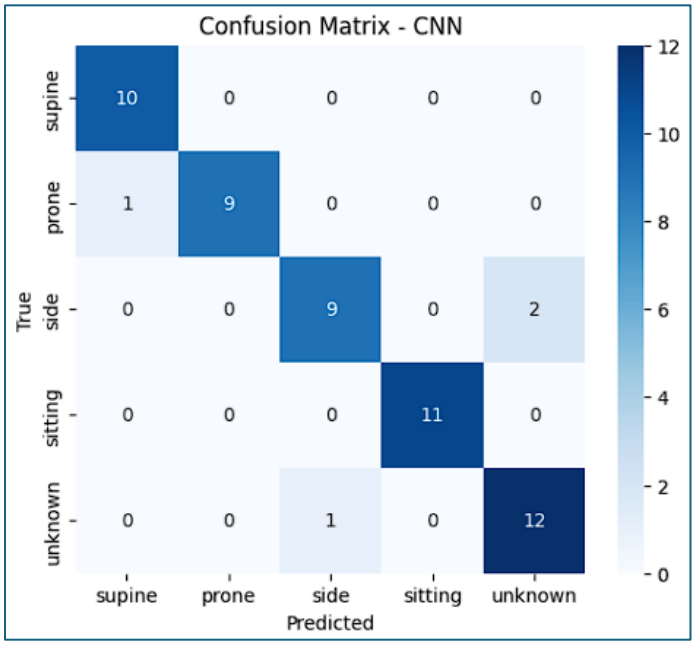

Figure 5. Confusion Matrix – 1D CNN (Selected Model for Deployment)

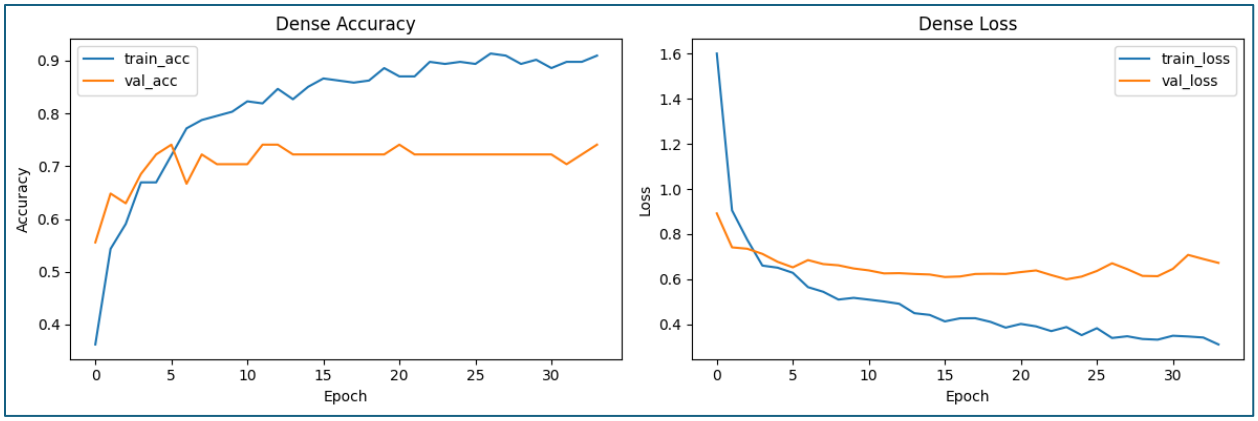

Figure 6. Final Test Accuracy Comparison – Dense Sequential

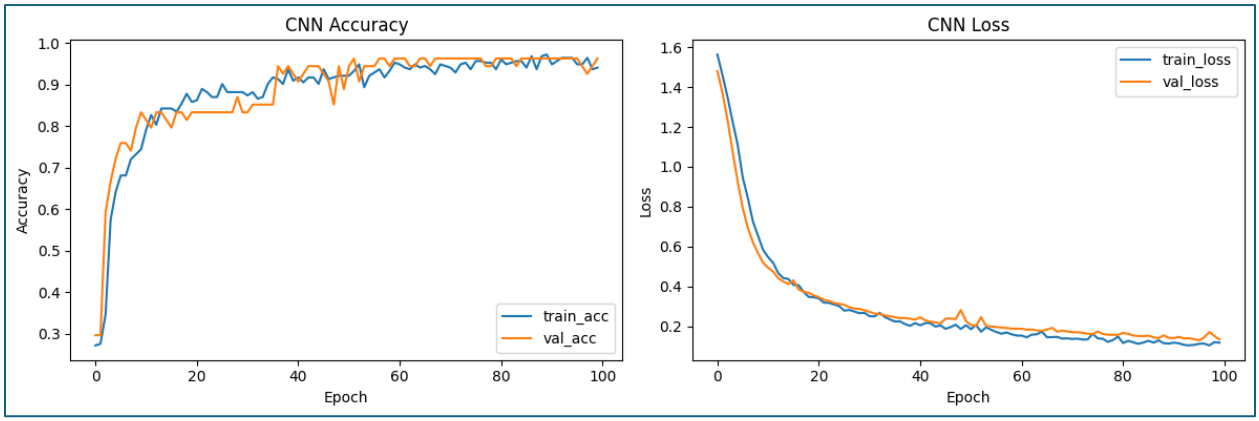

Figure 7. Final Test Accuracy Comparison – 1D CNN

The Figures generated for both the Dense and CNN models illustrate the learning process across training epochs. For both architectures, the Accuracy plots show the training accuracy increasing rapidly and stabilizing near the maximum possible value, while the validation accuracy follows closely, indicating the model learned effectively. Simultaneously, the Loss plots show a quick decrease in both training and validation loss, demonstrating that the models were successfully optimizing the weights. Since the validation curves tracked the training curves without a significant or prolonged separation, this confirms that minimal overfitting occurred, and the models generalized well to unseen data. The **1D-CNN model was chosen** because it achieved the highest test accuracy and is structurally better suited than the Dense network for extracting robust, time-invariant features from the windowed time-series data.

Model Deployment Steps for Arduino Integration

After the best model (the 1D-CNN) was trained and evaluated offline, the following critical steps were taken to prepare it for deployment on the resource-constrained Arduino Nano 33 BLE Sense:

1. Model Selection and Conversion (TFLite)

The best-performing Keras model (best_cnn.h5) was selected and converted into the TensorFlow Lite (TFLite) format.

- Float Conversion: The model was first converted to a standard float TFLite model, which is necessary for baseline deployment and for the subsequent quantization step.
- INT8 Quantization: To meet the strict memory and speed requirements of the Arduino microcontroller, the float model was optimized using post-training integer quantization (INT8). This process maps the 32-bit floating-point weights and activations to 8-bit integers, drastically reducing the model size and accelerating inference speed. A small subset of the normalized training data was used as a representative dataset to calibrate the quantization process, minimizing accuracy loss.

2. TFLite to C Array Conversion

The final optimized TFLite file (best_cnn_int8.tflite) is a binary format that cannot be directly included in an Arduino sketch.

- A Python utility was used to convert the binary TFLite model into a C source file (model_data.cc) and a header file (model_data.h).
- The model's binary data is stored as a large const unsigned char array, allowing the TFLM runtime on the Arduino to access the model directly from program memory (Flash).

3. Normalization Constant Integration

To ensure real-time data collected on the Arduino is correctly normalized before inference, the per-sensor standardization constants (μ and σ) calculated in the training phase were also prepared for integration:

- The constants (e.g., acc_mean, acc_std, etc.) were stored in a human-readable JSON file (scaler.json).
- These constants were then hardcoded into the Arduino sketch (or included via a configuration header file) to be available to the IMU_Processor module at runtime, enabling dynamic, per-sensor normalization.

4. Embedded Code Implementation

The Arduino sketch incorporated the following components:

- TensorFlow Lite Micro Library: Used to interpret the C-array model.
- Model Data: The compiled C array (model_data.h).
- Inference Loop: Handles the Serial Request-Response protocol, collecting the 200-sample window, applying the correct sensor-specific normalization, executing the TFLM interpreter, and sending the prediction result to the Base Station.

D. Results

Performance on Test Dataset (Offline):

The final 1D-CNN model achieved a strong test accuracy of 0.9273, confirming its generalization capability on unseen, normalized data. The confusion matrix (as presented in Section C – 5.1 report) highlighted robust separation for all posture classes, with minor remaining confusion between prone and supine.

Real-Time Prediction and Base Station Interaction:

The final stage of the project involved deploying the INT8 quantized model to the Arduino and testing the complete real-time prediction pipeline coordinated by the base_station.py script (or equivalent serial monitor interface).

Protocol Implementation

The base_station.py successfully demonstrated the Request-Response model, ensuring the microcontroller was passive until commanded. The interface provided options to initiate inference for the three different sensor streams:

Command Sent by base_station.py	Sensor Used by Arduino	Prediction Triggered
'1'	Accelerometer (ACC)	Posture classification based on ACC data.
'2'	Gyroscope (GYR)	Posture classification based on GYR data.
'3'	Magnetometer (MAG)	Posture classification based on MAG data.

Real-Time Accuracy and Sensor Agnosticism

Testing was conducted across the five posture classes by sending command '1', '2', and '3' sequentially while the Arduino board was held in a static posture (Supine, Prone, Sitting, Side).

- **Static Postures:** For the stable postures (supine, prone, sitting), the real-time prediction accuracy was consistently high across all three sensor requests.

- **Sensor Agnostic Validation:** Crucially, for a given posture (e.g., Supine), the prediction result remained the same regardless of whether the request used the ACC, GYR, or MAG sensor input. This confirmed the successful implementation of the sensor-agnostic design, validating that the model had learned features robustly across modalities, supported by the correct application of the per-sensor normalization constants.

Solution Demonstration Links

1. GitHub Repository: https://github.com/anushka002/BMI-CES-598-Embedded-Machine-Learning/tree/main/Project-4
2. YouTube Video Link: https://youtu.be/NzS9q4fE_0s

E. Discussion

Conclusion

This project successfully developed and deployed a supervised machine learning pipeline for real-time, five-class human posture classification on the Arduino Nano 33 BLE Sense Rev2. The system's core design effectively addressed the critical project constraints: sensor-agnosticism and orientation-insensitivity.

The 1D-CNN model was chosen for its ability to extract robust features from the 2.0 second IMU data windows, ultimately achieving a high offline test accuracy of 0.9273%. This high performance was attributable to the strategic data preparation, which included merging rotational variations (left/right side, up/down sitting) into single classes to promote orientation robustness and implementing a precise per-sensor Z-score normalization scheme.

The final system validated the two primary objectives:

1. Sensor Agnosticism: Real-time testing confirmed that a single, quantized TFLM model could consistently predict the correct posture when fed data from the Accelerometer, Gyroscope, or Magnetometer, proving the efficacy of the dynamic, sensor-specific normalization constants applied on the microcontroller.
2. Robustness: The high accuracy achieved for static postures demonstrated the model's ability to generalize well, despite the minor challenge of distinguishing between the rotationally inverted prone and supine postures.

Overall, the project provides a strong foundation for a scalable, low-power monitoring system. Future work should focus on utilizing temporal models like LSTMs to better model movement history and resolve subtle confusions like prone vs. supine.

Appendices

Appendix A - File inventory (key files)

Python Scripts & Colab Notebooks

- project4_data_collection.py: Logs the raw IMU stream from the Arduino, saving data into posture-specific CSV files.
- data_collection.py: Loads raw CSVs, applies 2.0s windowing and per-sensor normalization, merges labels for orientation-insensitivity (side, sitting), and splits data into training, validation, and test sets.
- Project04-BMI598-Colab.ipynb: Offline model training notebook. Defines CNN architectures, trains the models, executes TFLite (INT8) quantization, and generates the C array files for deployment.
- base_station.py: The Base Station script for real-time testing. Sends serial commands ('1', '2', '3') to the Arduino to request a prediction using a specific sensor (ACC, GYR, or MAG).

Arduino Code & Model Artifacts

- project4_basestation.ino: The main Arduino sketch. Contains the TFLM runtime, the logic for the Serial Request-Response protocol, real-time windowing, per-sensor normalization, and inference.
- artifacts/model_data.h / model_data.cc: C array files containing the INT8 quantized TFLite model of the selected 1D-CNN, used as program memory on the Arduino.
- artifacts/scaler.json: Stores the constants for per-sensor normalization (ACC, GYR, MAG means/standard deviations).

Dataset and Results Files

- Raw CSVs: prone.csv, supine.csv, side_left.csv, side_right.csv, sitting_up.csv, sitting_down.csv, unknown.csv.
- models/best_cnn_int8.tflite: The final, optimized TFLite model binary file.
- artifacts/cm_CNN.png: Confusion matrix for the final 1D-CNN model on the test set.

Appendix B: Run The Pipeline

The pipeline is executed in three stages:

1. **Collect and Prepare Data (Host PC):** Use project4_data_collection.py to acquire raw IMU data. Then, run data_collection.py to apply 2.0 s windowing, per-sensor standardization, label merging, and generate the necessary training, validation, and test splits.
2. **Train and Convert Model (Google Colab):** Execute Project04-BMI598-Colab.ipynb to train the 1D-CNN, save the normalization constants (scaler.json), and convert the best model into the TFLite C array format (model_data.cc/.h).
3. **Deploy and Test (Arduino & Base Station):** Integrate the C array and normalization constants into project4_basestation.ino and upload it to the Arduino Nano 33 BLE Sense. Run base_station.py on the host PC to initiate prediction requests and observe the real-time, sensor-agnostic classification results.

Appendix C: References

1. Arduino Nano 33 BLE Sense Rev2 Documentation: https://docs.arduino.cc/hardware/nano-33-ble-sense/ (Used for IMU initialization and library reference.)
2. Arduino BMI270 IMU Library Documentation: https://docs.arduino.cc/libraries/arduino_bmi270_bmm150 (Used to understand accelerometer API functions: IMU.begin(), IMU.readAcceleration().)
3. Python Libraries
   - *pandas* and *matplotlib* for data loading and plotting
   - *pyserial* for serial logging
   - TensorFlow. “Keras API Documentation.” https://www.tensorflow.org/api_docs/python/tf/keras
   - https://esciencecenter-digital-skills.github.io/intro-to-deep-learning-archaeology/02-keras/index.html
4. Course Material: BMI/CEN 598: Embedded Machine Learning, Arizona State University (Fall B 2025).
5. Personal Data Collection & Analysis: All thresholds and algorithm decisions were derived from the measured IMU data collected during this project.

Appendix D: Use of AI Tools

The development process utilized large language models to enhance efficiency and troubleshoot complex technical challenges. **ChatGPT** and **Gemini Pro** models were specifically employed for:

- Debugging and Error Resolution: Identifying and correcting errors in Python data processing scripts and Arduino inference code.
- Code Modification and Generation: Assisting with the structure and syntax of specialized functions, such as serial communication protocols and TFLM C array handling.
- Data Visualization and Plotting: Generating and refining the Python code used to create training history and confusion matrix plots.