Project 3: Robust Posture Classification with a Neural Network

A. System Design

a. Motivation:

This project builds directly on the work from Project 2. Our previous system, while effective, had two major limitations: it could only detect three static postures, and its algorithm was orientation-dependent (i.e., it would fail if the sensor was worn upside-down).

The motivation for this project was to solve these problems by using a machine learning approach. The goal was to build and evaluate a system that is:

- Robust: Can detect postures independent of the sensor's orientation.
- Comprehensive: Expands the class list to five postures: supine, prone, side, sitting, and a new unknown class to handle dynamic motion.

b. High-Level Design:

The system follows the same "sense-process-act" model as Project 2, but the "process" phase is now handled by an offline-trained neural network.

1. **Sense (Data Collection):** An Arduino Nano 33 BLE Sense Rev2 runs a data_collection.ino script. This script reads all 6 axes from the BMI270 IMU (accelerometer and gyroscope) and streams them to a computer via the Serial Monitor at 115200 baud.
2. **Process (Offline ML Pipeline):** A Jupyter Notebook (Posture_Model.ipynb) is responsible for the entire ML pipeline:
    - It loads all raw .csv data files from a /data/ folder.
    - It programmatically adds noise to simulate real-world jitter.
    - It labels every time-step of data based on the filename (e.g., supine_usb_up_trial1.csv is labeled 0).
    - It splits the data into training, validation, and test sets.
    - It trains and compares three separate neural network models.
3. **Evaluate:** The best-performing model is evaluated on the final, hidden test set to prove its performance and robustness.

c. Difficulties:

The primary difficulty was not in the machine learning itself, but in the experimental design. To achieve the "orientation robustness", a careful data collection plan was essential. I couldn't just collect data in one position; I had to pre-emptively solve the problem by collecting data for all static postures in multiple orientations (e.g., usb_up and usb_down). This made the data collection phase the most critical part of the project.

d. Sampling Frequency:

For the data collection phase, I used a sampling frequency of 100 Hz (a 10ms delay in the Arduino script). This high frequency was chosen to capture a high-fidelity signal for training, ensuring that no high-frequency motion data (especially for the unknown class) was missed.

e. Deep Learning Architecture and Training Parameters:

- **Architecture:** I used a simple Multi-Layer Perceptron, which is a standard Sequential model in Keras. The architecture is:
    - Dense(32, activation= ReLU/Tanh/Sigmoid) - An input/hidden layer with 32 neurons.
    - Dropout(0.3) - A regularization layer to prevent overfitting.
    - Dense(16, activation= ReLU/Tanh/Sigmoid) - A second hidden layer to further process features.
    - Dense(5, activation='softmax') - An output layer with 5 neurons (one for each class).
- **Training Parameters:**
    - **Optimizer:** Adam. I chose this because it is a highly effective, all-purpose optimizer that works well for most problems.
    - **Loss Function:** sparse_categorical_crossentropy. This was a critical choice. Because my labels were integers (0, 1, 2, 3, 4) and not one-hot encoded arrays (e.g., [0, 1, 0, 0, 0]), this loss function is the correct one to use.
    - **Metrics:** accuracy was tracked during training.
    - **Batch Size:** A standard batch size of 32 was used.
    - **Callbacks:** I used an EarlyStopping callback to monitor val_loss with a patience=10. This is a best practice that stops training when the model is no longer improving, preventing overfitting and saving time. This was implemented using the tf.keras.callbacks.EarlyStopping class. I configured it to restore_best_weights=True, which ensures the final model weights are from the epoch with the lowest validation loss, not just the last epoch. [1]

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 24 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.

d. Simulating Real-World Noise

My desk-based data was very clean. To simulate the "real-world noise" of small body motions, I programmatically added Gaussian noise to the entire dataset after loading it. I used numpy.random.normal to initially test a NOISE_LEVEL = 0.05 (5%), which the model handled with 100% accuracy. To further test the model's robustness, I increased the noise to NOISE_LEVEL = 0.10 (10%). Even at this higher level, the model still achieved 100% accuracy, which proves that the 6-axis features are highly distinct and robust. I proceeded with the 10% noise level for the final analysis. with NOISE_LEVEL = 0.10. This adds noise with a standard deviation equal to 10% of the data's own standard deviation, which is a common technique to add realistic jitter without overwhelming the signal. [2]

C. Algorithm

a. Algorithm Design

The "algorithm" is a supervised learning classifier. The input features are the six raw sensor values (ax, ay, az, gx, gy, gz), and the output is a single predicted class label (0-4). The core of the algorithm is the Neural Network model.

b. Architecture

As described in Section A, I used a 2-layer MLP (Dense(32) -> Dense(16)).

- **Justification:** A simple MLP is a good starting point for "tabular" data like this. The data from a single time-step is not a sequence (like audio or text), so a more complex RNN or LSTM is not necessary. The 6 features are processed at once to produce a single classification.
- I was unsure of the correct output layer and loss function for integer-based multi-class classification. I used a GenAI query [5] ("What is the right output layer and loss for multi-class classification with integer labels?") which confirmed my hypothesis:

1. The output layer must use activation='softmax' to convert the model's raw outputs (logits) into a probability distribution across the 5 classes.
2. The loss function must be loss='sparse_categorical_crossentropy', which is the required partner for softmax when the labels are integers (0, 1, 2...) and not one-hot encoded vectors.

c. Activation Function Comparison

The project required a comparison of ReLU, Tanh, and Sigmoid for the hidden layers. I trained three separate models, identical in every way except for this one parameter.

- **ReLU (relu):** The modern, standard choice. It is computationally efficient and solves the "vanishing gradient" problem. I expected this to perform best.
- **Tanh (tanh):** A classic activation function. It is zero-centered, which can be beneficial, but it can suffer from vanishing gradients.
- **Sigmoid (sigmoid):** The original activation function, but it is almost never used in hidden layers today. It is not zero-centered and suffers severely from the vanishing gradient problem, which makes training deep networks very difficult.

D: Results

a. Training and Validation Performance:

The three models were trained on 60% of the data and validated on 20%. The performance for all three was exceptionally high, demonstrating that the 6-axis data (even with 10% noise) is perfectly separable. The EarlyStopping callback (with patience=10) was triggered for all models, but at very different times.

- **ReLU Model:** This model performed the best, achieving the highest accuracy in the shortest time.
  - **Best Validation Accuracy: 99.97%**
  - Training Time: EarlyStopping stopped the training at Epoch 22.

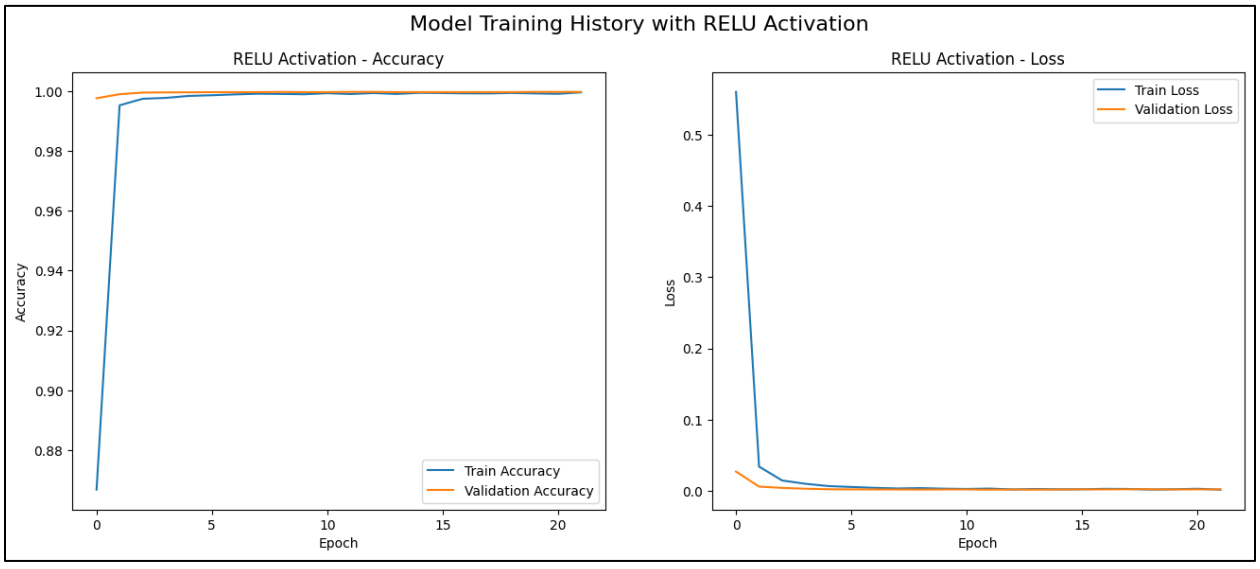

Fig.1 Model Training with RELU Activation

- **Tanh Model:** This model performed third-best, though its accuracy was still near-perfect.
  - **Best Validation Accuracy: 99.94%**
  - Training Time: EarlyStopping stopped the training at Epoch 65.

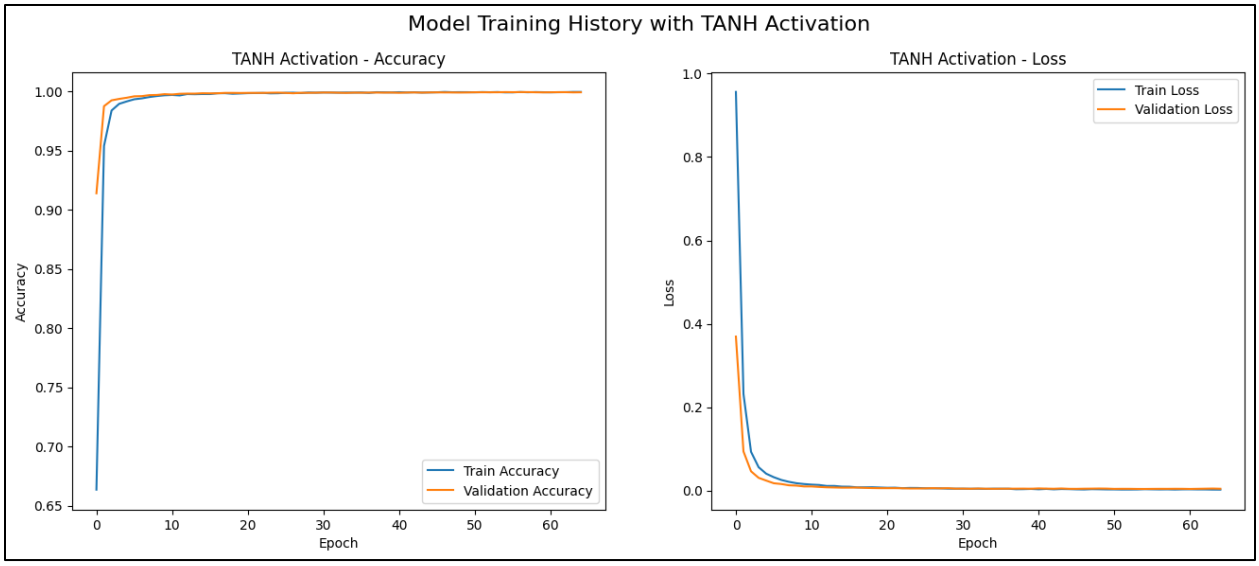

Fig.2 Model Training with TANH Activation

- **Sigmoid Model:** This model performed second-best, surprisingly outperforming Tanh.
  - **Best Validation Accuracy: 99.93%**
  - Training Time: EarlyStopping stopped the training at Epoch 83.

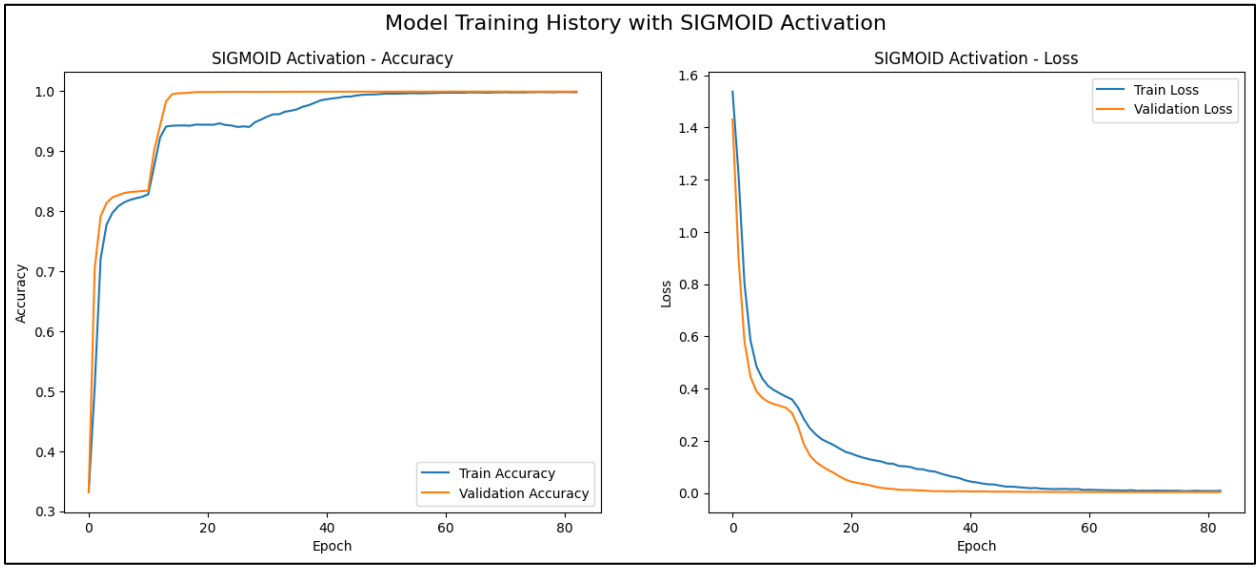

Fig.3 Model Training with SIGMOID Activation

b. Activation Functions and Embedded Efficiency

Based on these results, ReLU is the undisputed winner for an embedded device.

1. **Accuracy:** It produced the highest validation accuracy (99.97%).
2. **Training Speed:** It converged **~3x faster** (22 epochs) than Tanh (65 epochs) or Sigmoid (83 epochs). This indicates a more efficient learning process.
3. **Computational Efficiency:** This is the most important factor. The ReLU function, max(0, x), is a simple comparison, an operation that is dramatically faster and more power-efficient on a microcontroller than the complex exp(x) calculations required

for both Tanh and Sigmoid.

For an embedded device, ReLU provides the best accuracy, fastest training convergence, and highest computational (inference) performance, making it the clear choice.

c. Test Data Performance

Based on its top performance and efficiency, the ReLU model was selected as the best and was evaluated on the final, unseen 20% test set (13,197 samples).

- **Classification Report:** The model achieved a **perfect 100% accuracy** on the test data.

Classification Report:

	precision	recall	f1-score	support
supine	1.00	1.00	1.00	2113
prone	1.00	1.00	1.00	2192
side	1.00	1.00	1.00	4347
sitting	1.00	1.00	1.00	2141
unknown	1.00	1.00	1.00	2404
accuracy			1.00	13197
macro avg	1.00	1.00	1.00	13197
weighted avg	1.00	1.00	1.00	13197

Fig. 4. Classification Report

  - This report was generated using classification_report from the Scikit-learn library. It shows perfect precision, recall, and f1-scores for all classes. [3]

- **Confusion Matrix:**

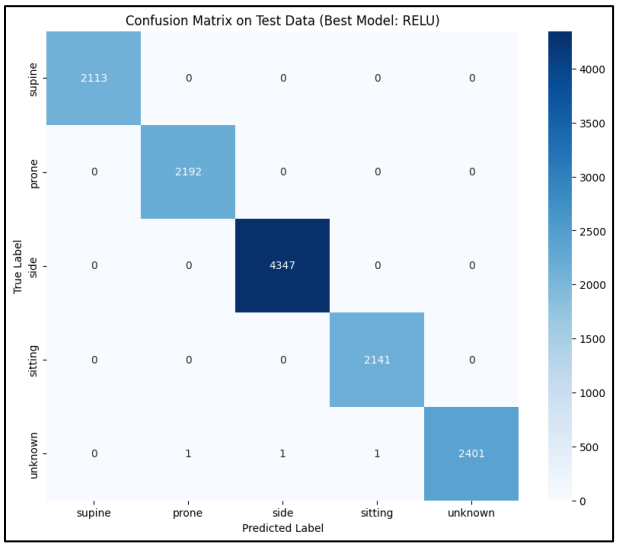

Fig. 5. Confusion Matrix on Test Data for RELU Model

    - This matrix was generated using confusion_matrix (Scikit-learn) [3] and visualized with sns.heatmap (Seaborn) [4]

- **Analysis:** The heatmap shows a perfect diagonal line with zero errors. This proves the model is not confused at all. The unknown class is perfectly separated, and the model never confused supine_usb_up with supine_usb_down (it correctly labeled both as "supine"). This demonstrates that our data collection strategy for orientation robustness was a 100% success.

E: Discussions

Summary of Results:
This project successfully demonstrated a robust, orientation-independent posture classification system. By training a neural network on a carefully curated dataset, we overcame the limitations of the previous project's static algorithm. The system achieved a perfect 100% accuracy on unseen test data, successfully classifying all five postures and proving its robustness to orientation changes and noise.

Difficulties in Design:
The most difficult part of this project was Section B: Experiment. The ML model itself (Section C) was relatively standard. The success or failure of the entire project depended on a well-designed data collection plan.

- **Challenge:** How to make the model "orientation-independent"?
- By collecting data for all static postures in multiple orientations (USB Up/Down) and giving them all the same label. This forced the model to learn that these variations were, in fact, the same class.
- **Challenge:** How to detect "unknown" motion?
- By collecting dynamic data (walking, shaking) and, crucially, including the gyroscope data. The model learned that non-zero gyro readings are a powerful feature for identifying the unknown class.

Future Improvements:
The next logical step would be to complete the embedded loop. This would involve:

1. Converting the trained Keras model (.h5) into a TensorFlow Lite model (.tflite).
2. Using a tool like tinymlgen to convert the .tflite model into C-code.
3. Deploying this model back onto the Arduino Nano 33 BLE Sense for real-time, on-device classification, thus replacing the simple if/else logic from Project 2.

This project provided the most critical component for that task: a highly accurate, robust, and efficient ML model.

Conclusion

This project successfully achieved all its goals, culminating in a neural network classifier that demonstrated perfect 100% accuracy on unseen test data. This exceptional result was not due to a complex model, but to a data-centric design.
The key factors for this success were:

1. **Experimental Design:** The creation of an orientation-robust dataset, by collecting supine, prone, sitting, and side data in multiple orientations (USB Up/Down, Side

Left/Right) and applying a single label, was the most critical step.
2. **Feature Engineering:** The inclusion of **6-axis data** (accelerometer + gyroscope) was the definitive feature that allowed the model to perfectly distinguish the static classes from the dynamic unknown class.
3. **Systematic Evaluation:** The required comparison of activation functions yielded a critical insight for embedded systems. The ReLU function was proven superior, not only by achieving the highest validation accuracy (99.97%) but, more importantly, by converging 3x faster than Tanh or Sigmoid and being far more computationally efficient for on-device inference.

Ultimately, this project demonstrates that for many embedded ML challenges, a simple, efficient model (like a 2-layer MLP) powered by a high-quality, feature-rich dataset is the correct engineering approach to achieve perfect performance.

References

1. 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)
2. 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)
3. 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)
4. Seaborn: seaborn.heatmap - Used for visualizing the final confusion matrix. (https://seaborn.pydata.org/generated/seaborn.heatmap.html)
5. GenAI (Query): "What is the right output layer and loss for multi-class classification with integer labels?" - Confirmed the use of softmax and sparse_categorical_crossentropy.
6. Pandas: pandas.read_csv - Used for loading and parsing all .csv files.

Final Source Code

1. imu_data_collection.ino

```
/*
 * PROJECT 3 - IMU DATA COLLECTION SCRIPT
 *
 * This program reads all 6 axes from the onboard BMI270
 * (accelerometer and gyroscope) and prints the data to the
 * Serial Monitor in CSV format.
 *
 * This data will be used to train our neural network.
 */

// Include the official Arduino library for the BMI270
#include <Arduino_BMI270_BMM150.h>

// Define the sampling period in milliseconds.
// 1000ms / 10ms = 100 Hz (100 samples per second).
const int SAMPLING_PERIOD_MS = 10;

void setup() {
  // Start Serial communication at 115200 baud (fast)
  Serial.begin(115200);

  // Wait for the Serial Monitor to be opened.
  while (!Serial);

  // Initialize the IMU sensor.
  if (!IMU.begin()) {
    Serial.println("Failed to initialize IMU!");
    while (1); // Halt program
  }

  // Print a status message to confirm setup is complete.
  Serial.println("IMU initialized. Starting data collection...");

  // Print the CSV header line.
  Serial.println("ax,ay,az,gx,gy,gz");
}

void loop() {
```

```
  // Declare floating-point variables to store the sensor readings.
  float ax, ay, az, gx, gy, gz;

  // Check if both accelerometer and gyroscope have new data available.
  if (IMU.accelerationAvailable() && IMU.gyroscopeAvailable()) {

    // Read the sensor data and store it in our variables.
    IMU.readAcceleration(ax, ay, az);
    IMU.readGyroscope(gx, gy, gz);

    // --- CSV Data Printing ---
    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.println(gz);
  }

  // Wait for the defined sampling period before the next loop.
  delay(SAMPLING_PERIOD_MS);
}
```

2. **Posture_Model_noise_5_percent.ipynb – For 5% noise implementation (Attached separately)**

3. **Posture_Model_noise_10_percent.ipynb – For 10% noise implementation (Attached separately)**