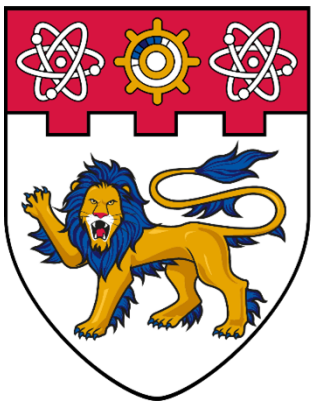

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

FEATHER SENSE LEARNS TO SKATE: REAL-TIME BATTERY POWERED IMU CLASSIFICATION WITH TFLITE AND BLE

SC4172

INTERNET OF THINGS: TINY MACHINE LEARNING

by

Aden Ong Yew

College of Computing and Data Science

2025

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The Internet of Things (IoT) has revolutionized sports by integrating data analytics and deep learning to improve athletic performance. However, deploying advanced deep learning models on low-power embedded systems poses significant challenges, particularly when classifying subtle human motions. Inline skating is a sport that combines physical skill with creativity, especially in classic slalom skating. In this style, skaters navigate through a series of cones, performing a variety of tricks. These moves include techniques in families such as sitting, wheeling, jumping, spinning, and others.

This project focuses on classic slalom inline skating and aims to classify the technical skills into distinct families: **weaving**, **jumping**, and **spinning**. In this context, *weaving* is defined as a combination of sitting, wheeling, and other weaving-related movements, all of which involve a characteristic weaving motion. The specific families are defined as follows:

- **Jumping:** Explosive moves where the skater leaves the ground and crosses at least four cones.
- **Spinning:** Rotational moves requiring the completion of at least three full rotations around or between cones.
- **Weaving:** A composite category (including sitting, wheeling, etc.) that demands a continuous weaving pattern across at least four cones.

To address these challenges, the project implements TensorFlow Lite on the Adafruit Feather Sense, enabling a deep learning model to run directly on the low-power device. The model processes sequences of inertial measurement unit (IMU) data using a sliding window approach, which is critical for continuous inference. Moreover, it incorporates data augmentation via axis permutations, simulating varied sensor orientations and enhances the model's robustness.

Overall, this project demonstrates that it is possible to deploy deep learning on low-power, embedded systems for complex sports applications. It provides a framework for building smart training tools that can deliver real-time feedback through sensor data analysis, efficient model inference, and wireless communication via BLE.

Chapter 2

Features

In this project, the input data is collected using the Adafruit Feather Sense, which is equipped with an Inertial Measurement Unit (IMU) that provides data from the accelerometer, gyroscope, and magnetometer. Two versions of the code were developed: one for serial transmission and another for Bluetooth Low Energy (BLE) transmission.

2.1 Sensor Data Collection

Adafruit Feather Sense's IMU data is captured by the two key sensors:

- **Adafruit LSM6DS33:**
  This sensor is a combined accelerometer and gyroscope. It measures linear acceleration along the x, y, and z axes (in m/s^2) as well as angular velocity (in rad/s).
- **Adafruit LIS3MDL:**
  This sensor is a magnetometer that measures magnetic field strength along the x, y, and z axes (in μTesla).

Each sensor sample is stored with a timestamp, ensuring accurate tracking over time. The code buffers a fixed number of samples (e.g., 900 for 30 seconds at 30 Hz), though actual recordings average around 878 due to latency.

Two methods are implemented for data extraction:

1. **Serial Data Extraction:**

```
// Send a header indicating the CSV data format
Serial.println("Data format: ax,ay,az,gx,gy,gz,mx,my,mz,timestamp");
// Transmit each sample as a comma-separated line
char asciiBuffer[128];
for (int i = 0; i < sampleIndex; i++) {
  snprintf(
    asciiBuffer,
    sizeof(asciiBuffer),
    "%.2f,%.2f,%.2f,%.2f,%.2f,%.2f,%.2f,%.2f,%.2f,%lu",
    sampleBuffer[i].ax,
    sampleBuffer[i].ay,
    sampleBuffer[i].az,
    sampleBuffer[i].gx,
    sampleBuffer[i].gy,
    sampleBuffer[i].gz,
    sampleBuffer[i].mx,
    sampleBuffer[i].my,
    sampleBuffer[i].mz,
    sampleBuffer[i].timestamp
  );
  Serial.println(asciiBuffer);
```

2. **BLE Data Extraction:**

```
// Send data in BLE UART chunks
void bleSendDataChunked(const char* data) {
  int len = strlen(data);
  const int chunkSize = 20; // Standard BLE MTU is 23, less 3 for ATT header leaves 20 for payload
  for (int i = 0; i < len; i += chunkSize) {
    int bytesToSend = min(chunkSize, len - i); // Calculate bytes for this chunk
    char chunk[chunkSize + 1];
    memcpy(chunk, data + i, bytesToSend);
    chunk[bytesToSend] = '\0'; // Null-terminate the chunk
    // Check connection and notification status before sending
    if (ble_connected && bleuart.notifyEnabled()) {
      bleuart.write(chunk, bytesToSend); // Send the exact number of bytes
      delay(15); // Short delay to allow BLE stack processing, prevent flooding
    } else {
      break; // Stop sending if not connected/notifiable
    }
  }
}
```

2.2 Feature Engineering

Prior to feeding the data into the deep learning model, several preprocessing steps are carried out:

- **Normalization and Scaling:** The raw sensor readings are normalized using calculated means and standard deviations to ensure that all features contribute equally to the model.

```
scaler = StandardScaler()
scaler.fit(X_train_reshaped)
X_train_scaled_reshaped = scaler.transform(X_train_reshaped)
X_val_scaled_reshaped = scaler.transform(X_val_reshaped)
X_test_scaled_reshaped = scaler.transform(X_test_reshaped)
```

- **Sliding Window Segmentation:** The continuous data stream is segmented into fixed-length windows. This approach preserves the temporal dynamics and patterns of the movements, which is essential for classifying different skating techniques.

```python
# Calculate window and slide size in samples
window_size = math.ceil(WINDOW_DURATION_S * sampling_rate)
slide_size = math.ceil(SLIDE_DURATION_S * sampling_rate)
print(f"Window size: {window_size} samples ({WINDOW_DURATION_S}s)")
print(f"Slide size: {slide_size} samples ({SLIDE_DURATION_S}s)")
```

- **Sampling:** The sensor readings are uniformly sampled at 30 Hz. This step ensures that the data from different sensors are aligned and that the integrity of the temporal relationships is preserved between readings.

```cpp
// Recording phase: capture sensor data for 30 seconds at ~30Hz
if (recording && (millis() - recordStart < 30000)) {
  sensors_event_t accel, gyro, dummy;
  lsm6ds33.getEvent(&accel, &gyro, &dummy);
  sensors_event_t magEvent;
  lis3mdl.getEvent(&magEvent);
  // Store sensor data in the buffer
  sampleBuffer[sampleIndex].ax = accel.acceleration.x;
  sampleBuffer[sampleIndex].ay = accel.acceleration.y;
  sampleBuffer[sampleIndex].az = accel.acceleration.z;
  sampleBuffer[sampleIndex].gx = gyro.gyro.x;
  sampleBuffer[sampleIndex].gy = gyro.gyro.y;
  sampleBuffer[sampleIndex].gz = gyro.gyro.z;
  sampleBuffer[sampleIndex].mx = magEvent.magnetic.x;
  sampleBuffer[sampleIndex].my = magEvent.magnetic.y;
  sampleBuffer[sampleIndex].mz = magEvent.magnetic.z;
  sampleBuffer[sampleIndex].timestamp = millis();
  sampleIndex++;
  delay(33);  // Approximately 30Hz sampling rate (1000ms/30 ≈ 33ms)
}
```

- **Data Augmentation via Axis Permutations:** This feature engineering pipeline augments the labeled IMU data by generating additional training samples through axis swapping. This process diversifies the microcontroller's orientation, thereby enhancing the system's robustness to variations in sensor data acquisition.

```python
def augment_swap_axes(segment_data, permutation_indices):
    """
    Augments a single segment by swapping axes according to the permutation.
    Args:
        segment_data (np.ndarray): Shape (timesteps, 9)
                                   [ax, ay, az, gx, gy, gz, mx, my, mz]
        permutation_indices (tuple): A tuple of 3 indices (e.g., (1, 0, 2))
                                     representing the new order of axes (e.g., Y, X, Z).
    Returns:
        np.ndarray: The augmented segment data with swapped axes.
    """
    augmented_data = np.zeros_like(segment_data)
    p = permutation_indices # e.g., (1, 0, 2) -> means new X is old Y, new Y is old X, new Z is old Z
    for sensor, base_idx in SENSOR_BASE_INDICES.items():
        # Extract original X, Y, Z columns for this sensor
        original_x = segment_data[:, base_idx + AXIS_INDICES['x']]
        original_y = segment_data[:, base_idx + AXIS_INDICES['y']]
        original_z = segment_data[:, base_idx + AXIS_INDICES['z']]
        original_axes = [original_x, original_y, original_z]

        # Assign to new positions based on permutation
        augmented_data[:, base_idx + AXIS_INDICES['x']] = original_axes[p[0]] # New X = Original[p[0]]
        augmented_data[:, base_idx + AXIS_INDICES['y']] = original_axes[p[1]] # New Y = Original[p[1]]
        augmented_data[:, base_idx + AXIS_INDICES['z']] = original_axes[p[2]] # New Z = Original[p[2]]
    return augmented_data
```

Chapter 3

Model Training Process

3.1 Microcontroller Model Design

A specialized CNN was developed to classify IMU sensor data directly on the resource-constrained Adafruit Feather Sense microcontroller using TensorFlow Lite Micro (TFLM). To ensure compatibility with TFLM, the model uses Conv2D layers instead of Conv1D, reshaping input data into 4D tensors to avoid unsupported operations. Two Conv2D layers extract hierarchical temporal features, followed by global average pooling to summarize patterns across time. A Flatten layer and a Dense output layer with softmax activation complete the classification of four classes, while Dropout is employed during training to reduce overfitting. The model is compiled using the Adam optimizer and categorical crossentropy loss, which are standard choices for multi-class classification tasks due to their efficiency and effectiveness. The resulting model is compact, TFLM-compatible, and optimized for efficient and fast device inference.

Architecture

- **Input:** Segmented sensor data is reshaped into a 4D tensor: `(none, 1, 45, 9)`.
- **Convolutions:** Two Keras `Conv2D` layers perform 1D convolutions along time with kernel size `(1,5)`:
  - First layer: 8 filters.
  - Second layer: 16 filters.

  Both layers use ReLU activation and `'same'` padding.
- **Pooling & Classification:**
  - `GlobalAveragePooling2D` averages the feature maps over the spatial dimensions, reducing them to a fixed-length feature vector.
  - `Dense` layer with softmax activation, producing a probability distribution over the 4 target classes.

- **Regularization:** Dropout (20% and 25%) is applied during training

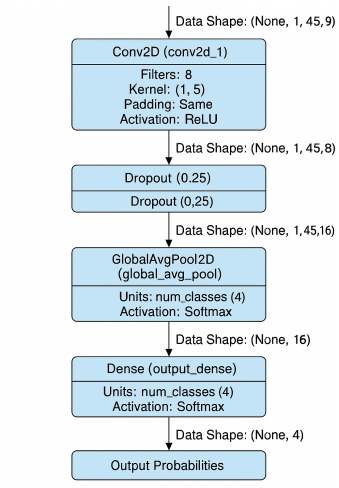

Figure 1: CNN architecture

3.2 Training Pipeline

- **Data:** IMU segments are loaded from `labeled_imu_segments.pkl` and augmented by applying all 6 axis permutations, resulting in `augmented_labeled_imu_segments.pkl`.
- **Splitting & Preprocessing:** Data is stratified into training (65%), validation (20%), and testing (15%) sets. A `StandardScaler` is fitted on the training data to normalize the features. Next, the data is reshaped from its original 3D format (`(samples, timesteps, features)`) into the explicit 4D format required by the model's `Conv2D` input layer (`(samples, 1, timesteps, features)`), specifically `(samples, 1, 45, 9)`.
- **Model Compilation:** The model is compiled using the Adam optimizer and categorical crossentropy loss, with accuracy as the performance metric.
- **Training:** The model is trained for up to 100 epochs with a batch size of 16. To counteract class imbalances, balanced class weights are computed using `compute_class_weight` and applied during training. Two callbacks are employed:
  - `EarlyStopping`: Monitors the validation loss and stops training if no improvement is observed for 15 consecutive epochs, restoring the best weights.
  - `ModelCheckpoint`: Saves the model weights whenever the validation loss improves, resulting in the final model being saved as `imu_cnn_model_explicit2d.h5`.
- **Deployment Preparation:** After training, the best model is converted into the TensorFlow Lite format using the default optimizations provided by the TFLiteConverter. The resulting `.tflite` file is then converted into a C header file (`imu_model.h`) via the `xxd` utility, making it ready for integration into the microcontroller firmware. Additionally, the normalization parameters (means and scales) obtained from the `StandardScaler` are saved for use during on-device inference.

Chapter 4

Optimization

Optimizing the deep learning model and its embedded implementation is critical for ensuring that the system meets real-time performance and power efficiency requirements on a low-power microcontroller. This section outlines the techniques used for code size minimization, real-time performance improvements, and power efficiency optimization.

4.1 Code Size Minimization

- **Techniques Used:** To minimize the model size, the following techniques were applied:
  - **Model Quantization:** The TensorFlow Lite Converter was configured with default optimizations (`tf.lite.Optimize.DEFAULT`), reducing model size and memory usage by converting weights and activations to lower precision.
  - **Conversion to C Array:** After converting the model to TensorFlow Lite format, the resulting binary is transformed into a C header file using the `xxd` tool. This header file contains the model as a C array (`g_imu_model_data`), enabling its direct inclusion in the firmware.
  - **Model Pruning:** Magnitude pruning was tested using the TensorFlow Model Optimization Toolkit (TFMOT). The standard Keras model was wrapped with `tfmot.sparsity.keras.prune_` which automatically adds pruning logic to layers such as `Conv2D` and `Dense`. A pruning schedule was defined using `tfmot.sparsity.keras.PolynomialDecay`, gradually increasing the sparsity from an initial 50% to a final 80% over the course of training, with pruning updates every 100 steps. Pruning-aware callbacks, including `UpdatePruningStep` and `PruningSummaries`, were employed during model fitting to manage and monitor the sparsity progression. After training, the pruning wrappers were removed using `tfmot.sparsity.keras.strip_pruning`, resulting in a standard Keras model with a significant portion of its weights zeroed out.
- **Size Comparison:** The pruning process zeroed many weights, but when converting to TFLite, those zeros were stored explicitly. The pruned model's TFLite file size was 7,088

bytes compared to 7,072 bytes for the original model. This small difference is due to minimal overhead from the pruning workflow rather than a reduction in data storage. Without specialized sparse kernel support on the target platform, pruning does not significantly reduce TFLite file size and mainly serves to improve quantization outcomes.

4.2 Inference Performance and Latency Analysis

- **Latency Benchmarks:** Using the microcontroller's `micros()` timer, key latencies per inference cycle were measured:
  - Sensor Data Acquisition: $\sim$1.95 ms (1953–1954 µs)
  - Model Inference: $\sim$11.72 ms (11718–11719 µs)
  - BLE Transmission: $\sim$14.65 ms (14648–14649 µs)

  Total inference cycle time is approximately 28.32 ms.
- **Sliding Window Implementation:** Sensor data is collected at 30 Hz (33 ms intervals) into a circular buffer. Inference is triggered after 16 new samples are acquired and the buffer (45 samples) is full, yielding a target slide interval of 528 ms. The observed 528 ms interval confirms that the processing time (28–30 ms) is well within the available window.
- **Input Processing & Real-Time Viability:** The model accepts data in an explicit 4D shape (`(1, 1, 45, 9)`), with efficient copying and normalization performed by `LoadCircularBufferToInput`. With an inference cycle of $\sim$28–30 ms and a slide interval of 528 ms, the system reliably meets its real-time requirements, ensuring timely feedback during practice or competition.

4.3 Power Efficiency Optimization

Power efficiency is primarily addressed through the selection of the low-power Adafruit Feather Sense (nRF52840) platform. Optimizing the inference latency—achieved at approximately 11.7 ms minimizes the active processing time per classification cycle. Although the continuous sampling design limits the use of deep sleep modes, the non-blocking timer checks allow for potential low-power idle states between operations. Further optimization could involve fine-tuning the BLE parameters or further quantization to reduce computational load and consequently inference computation time.

Chapter 5

Results & Conclusion

5.1 Inference Performance

The unpruned and pruned models were evaluated on the test set, and their performance metrics are summarized in Tables 1 and 2, respectively.

Table 1: Performance of the Unpruned Model

Class	Precision	Recall	F1-score	Support
jump	0.95	0.92	0.94	39
null	0.74	0.79	0.76	57
spin	0.85	0.90	0.88	62
weave	0.89	0.76	0.82	51
Test Accuracy	84.21% (209 samples)			
Macro Avg	0.85	0.85	0.85	209
Weighted Avg	0.85	0.84	0.84	209

Table 2: Performance of the Pruned Model (Stripped)

Class	Precision	Recall	F1-score	Support
jump	0.88	0.95	0.91	39
null	0.73	0.72	0.73	57
spin	0.91	0.69	0.79	62
weave	0.70	0.88	0.78	51
Test Accuracy	79.43% (209 samples)			
Macro Avg	0.81	0.81	0.80	209
Weighted Avg	0.81	0.79	0.79	209

5.2 Confusion Matrix and Error Analysis

The confusion matrices for both models reveal distinct patterns in classification errors, as summarized by the performance metrics in Tables 1 and 2. For the unpruned model, the **jump** class is very well-classified, achieving high precision (0.95) and recall (0.92), while the **null** class shows relatively lower metrics (precision 0.74, recall 0.79), indicating some misclassification with other classes. The **spin** and **weave** classes exhibit reasonable performance with F1-scores of 0.88 and 0.82, respectively, although there is room for improvement in distin-

guishing between similar movements.

In contrast, the pruned model (after stripping the pruning wrappers) demonstrates a shift in error distribution. Although the **jump** class maintains a high recall (0.95), its precision drops to 0.88. The **null** class continues to underperform, with both precision (0.73) and recall (0.72) being lower. Notably, the pruned model shows more frequent confusion between the **spin** and **weave** classes: the **spin** class achieves a higher precision (0.91) but a lower recall (0.69), while the **weave** class displays the opposite trend (precision 0.70, recall 0.88). This suggests that although pruning can help reduce model complexity, it may slightly affect the model's ability to differentiate between closely related classes.

5.3 Conclusion

This project successfully demonstrated the feasibility of deploying a deep learning model on a low-power microcontroller, the Adafruit Feather Sense, for the real-time classification of classic slalom inline skating movements. By leveraging IMU sensor data, implementing a tailored CNN architecture compatible with TensorFlow Lite Micro (TFLM), and incorporating essential optimizations, the system provides timely feedback via BLE communication.

The unpruned CNN model achieved a commendable test accuracy of 84.21

Optimization efforts focused on both model complexity and real-time performance. Magnitude pruning, while not yielding significant code size reduction on the target platform due to the lack of specialized sparse kernels, was explored, resulting in a model with 80% sparsity and a slightly reduced accuracy of 79.43%. Analysis of the pruned model's performance indicated a trade-off, particularly affecting the differentiation between the similar **spin** and **weave** classes. Latency analysis confirmed the system's real-time capability, with sensor reading (2ms), inference (12ms), and BLE transmission (15ms) collectively fitting well within the target sample interval (33ms), enabling a consistent sliding window update rate of 528ms. The selection of the low-power nRF52840 platform and the relatively fast inference contribute positively to power efficiency, although aggressive power-saving modes were not implemented due to the continuous sampling requirement.

In summary, the developed system effectively classifies inline skating moves on the edge, showcasing the potential of embedded AI for sports analytics. It validates the chosen design methodology, including TFLM-specific architectural choices and data augmentation, as crucial for practical deployment on resource-constrained hardware.