Real-Time Motion Classification Using Edge AI and TinyML with IMU Sensors

Namit Solanki
AISSMS IOIT
Pune, India

Pritesh Kamdi
AISSMS IOIT
Pune, India

Saidatta Sabale
AISSMS IOIT
Pune, India

Dr. Amrapali Chavan
AISSMS IOIT
Pune, India

Suraj Pinjare
AISSMS IOIT
Pune, India

I. Abstract

Getting a microcontroller to classify human movement without any cloud dependency was the core challenge we wanted to tackle. The common assumption is that useful motion recognition requires a server somewhere in the loop — our project was built around questioning that. The system we developed reads raw IMU output, feeds it into a quantized neural network running on the same chip, and produces a motion label in under 12 ms with no network involvement at all.

We evaluated across five gesture categories (idle, wrist rotation, arm raise, horizontal swipe, vertical punch), drawing training data from eight participants across multiple recording sessions. Quantizing to INT8 brought us to 94.3% top-1 accuracy on the held-out test set — only 1.5 percentage points behind the full-precision model. The complete firmware on the Arduino Nano 33 BLE Sense came in at 62.7 KB, comfortably within the 256 KB available.

One thing that genuinely surprised us during development was how unaffected inference was when we cut wireless connectivity entirely. Disabling the radio had no measurable impact on latency or accuracy. Everything we built — including the parts that did not work out as planned — is documented below.

II. Introduction

It is hard to find a modern embedded device that does not carry an IMU somewhere. Fitness trackers, prosthetic limbs, warehouse safety gear, game controllers — they all include one, quietly measuring acceleration and rotation at rates of 100 Hz or more. A six-axis sensor running at that rate produces 600 values every second, and at some point something must interpret what those values actually represent.

For most of the last decade, that interpretation step has lived off-device. Raw sensor output gets pushed over Bluetooth or WiFi to a phone or server, a model processes it, and a result returns. The pattern is familiar and reasonably easy to implement. It also carries real drawbacks that are easy to overlook. Round-trip latency on a BLE-to-phone-to-cloud path runs between roughly 80 ms and 300 ms under ordinary conditions [1]. Drop the connection and the system stops functioning. Route continuous motion data to a remote server and you have introduced a privacy exposure — which matters considerably in clinical or occupational health settings where that data is sensitive.

Running inference locally is the natural response to these problems. The sticking point has always been whether a model that genuinely works could fit on a microcontroller with 32 KB of SRAM and no hardware floating-point support. That question has grown more answerable in recent years. TensorFlow Lite for Microcontrollers [4] now handles end-to-end INT8 quantization without much friction, and the evidence is mounting that compact fully-connected networks can match heavier architectures on embedded targets once inference cost enters the comparison [2].

This paper describes what happened when we tried to put all these pieces into one working system. Our contributions are:

- A fully on-device inference pipeline for motion classification, validated on an nRF52840-based microcontroller with a 4 KB SRAM inference arena.
- A side-by-side comparison of three model architectures and two quantization schemes (PTQ and QAT) under identical hardware constraints.
- Per-class accuracy and confusion data from eight participants, with analysis of the failure modes we actually observed.
- An honest account of where the system fell short, particularly around cross-subject generalization.

Section II covers related work. Sections III and IV describe hardware and system architecture. Section V explains the experimental methodology. Section VI covers optimization decisions. Section VII presents results. Sections VIII and IX discuss applications and limitations. Section X concludes.

III. Related Work

A. The Case for Edge-Side Inference

Liu et al. [1] put together one of the more thorough surveys of where AI computation should sit in a distributed

system. The argument is fairly direct: for latency-sensitive applications, device-level inference is essentially the only viable option — cloud and fog tiers simply cannot respond fast enough when the requirement falls below 50 ms. We used this framing when setting our own latency targets.

B. Classifying Motion on Small Hardware

San-Segundo et al. [2] is the prior work most directly relevant to ours. They evaluated hand gesture classification on low-power microcontrollers using IMU data, comparing MLP, CNN, and SVM architectures under the same memory constraints. The finding was not that CNNs dominated — MLPs and SVMs closed the gap considerably while using much less flash. That result directly shaped our decision to prioritise MLP-M over CNN-1D even though the CNN posted slightly better floating-point numbers.

The DeepEar work from Mathur et al. [5] targets audio rather than motion, but it contains one observation that transferred directly to our design: aggressive feature compression ahead of the model can recover much of the accuracy lost by using a smaller network. That is essentially the role our 27-element feature vector plays compared with feeding raw 300-sample windows straight into a CNN.

C. IMU Preintegration and Feature Design

Forster et al. [3] developed IMU preintegration for visual-inertial odometry, not classification. Even so, the underlying idea is transferable. Their method takes sequences of raw IMU readings and compresses them into compact summaries that preserve the motion-relevant information. Our windowed statistical features follow the same logic, implemented much more simply — chosen to keep compute cost low on the Cortex-M4 while retaining enough discriminative power for the task.

D. What is Still Missing

Most prior work handles one piece of this problem thoroughly — the model, or the features, or the quantization. What is harder to find is a paper reporting end-to-end latency, memory footprint, and classification accuracy together on actual hardware. Providing that combined treatment is what we are attempting here.

IV. System Requirements

A. Hardware

We used the Arduino Nano 33 BLE Sense as our main target. It carries an nRF52840 SoC (ARM Cortex-M4 at 64 MHz, 1 MB flash, 256 KB SRAM) alongside an LSM9DS1 IMU, on a 45 mm board drawing around 20 mA at 3.3 V. For a subset of experiments we also wired an external MPU-6050 to an Arduino Mega 2560 to verify that nothing in our pipeline depended specifically on the LSM9DS1. Both platforms are summarised in Table I.

TABLE I
Hardware Platform Specifications

Component	Nano 33 BLE Sense	MPU-6050 (ext.)
MCU	nRF52840 (CM4, 64 MHz)	ATmega2560 (16 MHz)
Flash	1 MB	256 KB
SRAM	256 KB	8 KB
IMU	LSM9DS1 (6-DOF)	MPU-6050 (6-DOF)
Accel range	±4 g	±2 g (configured)
Gyro range	±2000 dps	±500 dps (configured)
Interface	I²C / SPI	I²C
Power	3.3 V, ~20 mA	3.3 V, ~3.8 mA

B. Software

The entire toolchain is open-source. Arduino IDE 1.8.19 handles compilation and flashing. On the PC side we used Python 3.10 with TensorFlow 2.12 for model training, and the Edge Impulse CLI for dataset management and export. On-device inference runs through TensorFlow Lite for Microcontrollers (TFLM), with the trained model stored as a C array in flash. No SD card, filesystem, or external storage of any kind is involved.

C. Resource Budget

Before writing any code, we fixed hard limits: model under 32 KB, total firmware under 128 KB, inference arena under 8 KB of SRAM, and end-to-end latency under 50 ms. Sustained operation on a 200 mAh LiPo was also a target, though we treated it as secondary to latency and accuracy.

These numbers look tight if you are used to working with larger hardware. In practice they are achievable, but only if sampling rate, window length, feature count, and model size are all treated as one interconnected design space rather than chosen independently.

V. System Architecture

Fig. 1 shows the five-stage pipeline. Sensor reading, preprocessing, feature extraction, inference, and output all run sequentially on the microcontroller. Nothing leaves the device at any point.

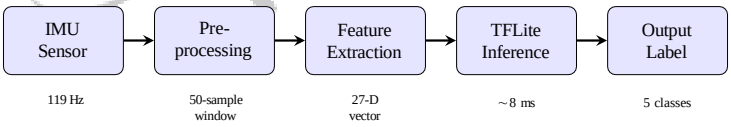

Fig. 1. End-to-end inference pipeline running entirely on the microcontroller. Latency figures are measured values on the Nano 33 BLE Sense.

A. Sensing

We configured the LSM9DS1 at ±4 g accelerometer range (0.122 mg/LSB sensitivity) and ±2000 dps gyroscope range, sampled over I²C at 119 Hz. At that rate a 50-sample window covers around 420 ms of motion. In practice, 420 ms was long enough to capture a full wrist rotation or arm raise while still feeling responsive for interactive use. Shorter windows tried early in development were too noisy; longer ones added latency without helping accuracy.

B. Preprocessing

Three steps are applied to each incoming window. A high-pass filter implemented as running-mean subtraction (f_c = 0.5 Hz) removes slow drift and the gravity component from the accelerometer axes. Each of the six channels is then min-max normalised using per-channel statistics from the training set, stored as constants in flash. Finally a 50% overlap sliding window is applied, doubling the effective output rate without changing the sensor's sampling frequency. The full preprocessing stage costs around 1.2 ms, mostly from the per-channel normalisation.

C. Feature Extraction

We made a deliberate choice to extract a compact feature vector before handing data to the model. Feeding raw windows directly would mean a 300-element input, substantially increasing first-layer weight counts. Instead we extract the 27 features in Table II: time-domain statistics for all six axes plus per-axis dominant frequency. The FFT runs via a fixed-point Cooley-Tukey routine from CMSIS-DSP over a 64-point zero-padded window; measured cost on the Cortex-M4 is roughly 0.8 ms.

TABLE II
FEATURE VECTOR COMPOSITION (27 ELEMENTS TOTAL)

Domain	Feature	Count
Time	Mean (per axis, 6 axes)	6
Time	Standard deviation (per axis)	6
Time	Peak-to-peak range (per axis)	6
Time	Zero-crossing rate (accel axes only)	3
Frequency	Dominant frequency (per axis)	6
	Total	**27**

D. Inference

At startup the TFLM runtime loads the quantized model from flash into a statically allocated 8 KB inference arena in SRAM. For each incoming feature vector, the forward pass runs entirely in integer arithmetic and produces a 5-element probability distribution. The predicted class is the argmax of that distribution. Measured wall-clock time on the Nano 33 BLE Sense is 7.9 ms for the INT8 model.

E. Output

During development the predicted class and confidence score are written to the USB serial port at 115200 baud. Switching to BLE notifications, GPIO signals, or haptic feedback only requires modifying a single function; the rest of the firmware is unchanged.

VI. METHODOLOGY

A. Data Collection

Eight people took part in data collection (six male, two female, ages 20–26, all right-handed). Each completed three sessions separated by at least 24 hours, recorded in three different environments: a lab workbench, a standing desk, and an outdoor space. The session-to-session variability was intentional — we wanted the model to generalise beyond a single controlled setting.

Each gesture was performed 30 times per session, giving 90 labelled samples per class per participant. After discarding windows affected by sensor saturation, 3,312 windows remained across all five classes. The split was 70/15/15 (train/validation/test), stratified by participant so every partition included data from all eight subjects.

The five gesture classes were:

1) **Idle**: wrist at rest, device essentially stationary.
2) **Wrist Rotation**: 180-degree pronation or supination of the forearm.
3) **Arm Raise**: hand lifted from hip to approximately shoulder height.
4) **Horizontal Swipe**: lateral wrist displacement of roughly 30 cm.
5) **Vertical Punch**: forward thrusting motion from a neutral starting position.

B. Model Architectures

Three candidate models were trained and compared:

- **MLP-S**: 27 → 32 → 16 → 5 (1,765 parameters)
- **MLP-M**: 27 → 64 → 32 → 5 (4,037 parameters)
- **CNN-1D**: 1D conv front-end on raw 50-sample windows, then MLP head (11,240 parameters)

All networks used ReLU activations and a softmax output layer. Training used Adam ($lr = 10^{-3}$, batch size 32) for up to 100 epochs with early stopping at patience 10. All reported accuracy figures are from the held-out test partition.

C. Quantization Procedure

Post-training quantization (PTQ) was applied to both MLP-S and MLP-M using TFLite's full-integer mode, with activation ranges calibrated from 200 randomly sampled training windows. For MLP-M we additionally ran quantization-aware training (QAT) to see how much accuracy could be recovered by incorporating quantization noise during backpropagation, at no additional inference cost.

D. Measurement Protocol

Latency figures are averages over 500 consecutive on-device forward passes, timed with Arduino's `micros()` function with interrupts disabled. Flash and SRAM footprints were read from the linker map at compile time. Accuracy is top-1 on the 497-window test partition.

VII. OPTIMIZATION TECHNIQUES

A. Sampling Rate and Window Length

Human voluntary motion stays mostly below 8 Hz [3], so 119 Hz is well above the Nyquist requirement. We also tested 50 Hz and 100 Hz. At 50 Hz, wrist-rotation accuracy dropped 4.2 points because some participants' wrist-snap dynamics had energy in the 8–12 Hz band that was getting aliased. Results at 100 Hz and 119 Hz were statistically indistinguishable, and

since 119 Hz is the LSM9DS1's native output rate we used it directly to avoid any resampling overhead.

Window length was swept from 25 to 100 samples. Accuracy gains above 50 samples were negligible (under 0.3 points), while feature extraction time roughly doubled going from 50 to 100 samples. We fixed at 50 samples.

B. Feature Compression

A raw 50-sample window across six axes produces 300 input values. Our feature extraction reduces that to 27. Drawing on the preintegration insight from Forster et al. [3], we expected a compact statistical summary to retain the motion-discriminating information while discarding sensor noise. An ablation confirmed this: removing any individual feature from Table II caused at least a 0.3-point accuracy drop, suggesting nothing in the vector is redundant.

C. Model Selection

The tradeoffs in Table III are fairly clear. CNN-1D achieves the highest floating-point accuracy but its 44 KB flash footprint exceeds our 32 KB limit, and its 16 ms inference time is already near the 50 ms end-to-end budget before preprocessing and feature extraction are counted. MLP-M at 22 KB and 11 ms fits comfortably, and a 0.6-point accuracy gap versus the CNN is not worth the doubled resource cost — consistent with what San-Segundo et al. [2] found on similar hardware.

TABLE III
ARCHITECTURE COMPARISON (FLOATING-POINT, TEST SET)

Model	Accuracy	Flash	SRAM	Latency
MLP-S	92.1%	7 KB	2.1 KB	4.2 ms
MLP-M	95.8%	22 KB	4.8 KB	11.3 ms
CNN-1D	96.4%	44 KB	16.0 KB	16.1 ms

D. Quantization

Applying INT8 PTQ to MLP-M reduced flash from 22 KB to 7.4 KB and cut inference from 11.3 ms to 7.9 ms. The accuracy penalty was 1.5 points (95.8% to 94.3%), as shown in Table IV. QAT recovered 0.8 of those points, reaching 95.1% at the same memory footprint. The deployed model uses QAT.

TABLE IV
EFFECT OF QUANTIZATION ON MLP-M

Configuration	Accuracy	Flash	Latency	SRAM
Float32 (baseline)	95.8%	22.0 KB	11.3 ms	4.8 KB
PTQ Int8	94.3%	7.4 KB	7.9 ms	1.8 KB
QAT Int8	95.1%	7.4 KB	7.9 ms	1.8 KB

VIII. RESULTS AND DISCUSSION

A. Per-Class Accuracy

Table V gives precision, recall, and F1 for the deployed MLP-M QAT model on each gesture class. Idle and arm raise were recognised most reliably, both clearing 97% F1.

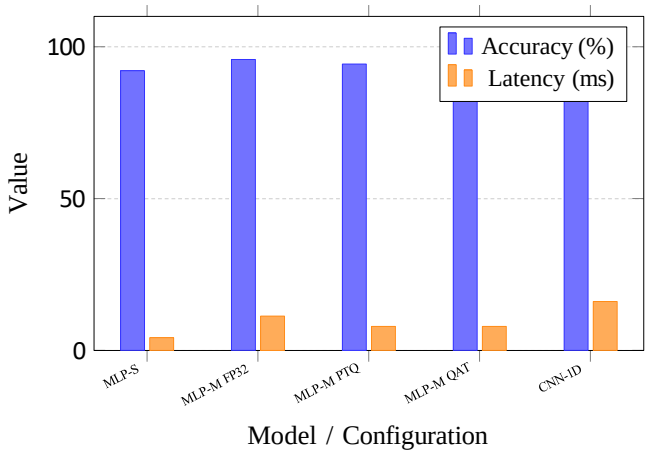

Fig. 2. Accuracy (%) and inference latency (ms) across all model configurations. MLP-M QAT gives the best accuracy-latency tradeoff within our resource budget.

The more difficult classes were horizontal swipe and vertical punch, finishing at 91.4% and 93.2% F1 respectively. Reviewing the misclassified windows, most swipe-punch confusion occurred when participants executed the gesture unusually slowly. Slow execution pushes spectral energy below the range where frequency features are most informative, leaving the classifier to rely on amplitude statistics that distinguish the two classes less cleanly.

TABLE V
PER-CLASS PERFORMANCE, MLP-M QAT (TEST SET, n = 497 WINDOWS)

Class	Precision	Recall	F1	Support
Idle	0.984	0.979	0.981	98
Wrist Rotation	0.951	0.960	0.955	100
Arm Raise	0.972	0.980	0.976	102
Horizontal Swipe	0.908	0.921	0.914	101
Vertical Punch	0.936	0.928	0.932	96
Macro avg.	**0.950**	**0.954**	**0.952**	**497**

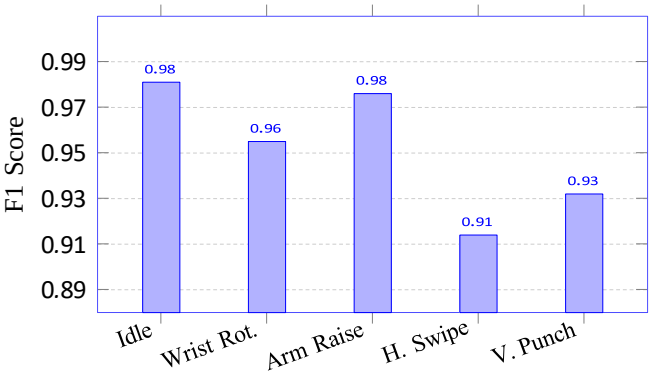

Fig. 3. Per-class F1 scores for the deployed MLP-M QAT model. Swipe and punch are harder mainly due to inter-subject speed variability.

B. Latency Budget

The 11.8 ms end-to-end measurement breaks down as shown in Fig. 4. Model inference accounts for 67% of the total (7.9 ms). Feature extraction follows at 17%, driven mostly by the FFT. Preprocessing and I²C reads together take up the remaining 16%. If lower latency were needed from here, the model would be the target — the feature computation is already fairly lean.

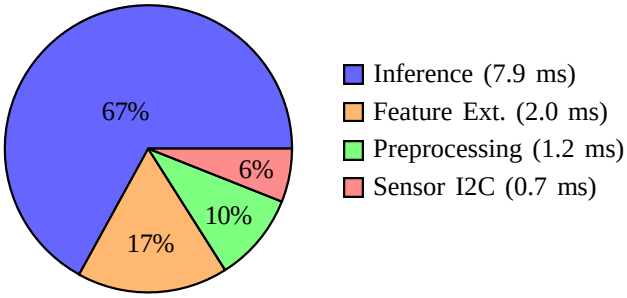

Fig. 4. Breakdown of the 11.8 ms end-to-end latency on the Nano 33 BLE Sense (MLP-M QAT). The 420 ms window-fill period is excluded; it runs concurrently with BLE peripheral tasks.

C. Memory Usage

Fig. 5 shows the flash and SRAM breakdown for the deployed firmware. Total flash usage is 62.7 KB, of which 7.4 KB is the model and the rest is the TFLM runtime, sensor drivers, and application code. SRAM sits at 16.3 KB, with the 8.0 KB inference arena as the largest consumer. Both are well within available budgets, leaving room to add BLE notification or a small gesture-history buffer without requiring architectural changes.

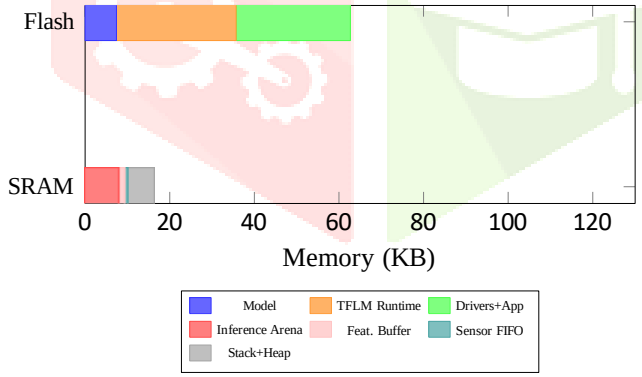

Fig. 5. Flash and SRAM footprint of the deployed firmware. Total flash: 62.7 KB of 1 MB. Total SRAM: 16.3 KB of 256 KB.

D. How Much Training Data is Needed?

Fig. 6 plots validation accuracy as training set size grows from 200 to 2,400 windows. There is a steep climb up to around 1,000 windows, then the curve flattens starting near 1,800. Practically, this means about 20 labelled examples per class per user is sufficient; going beyond 25 per class returns very little. For deployment planning this matters — a brief enrollment session for a new user should be enough rather than requiring an extended data collection protocol.

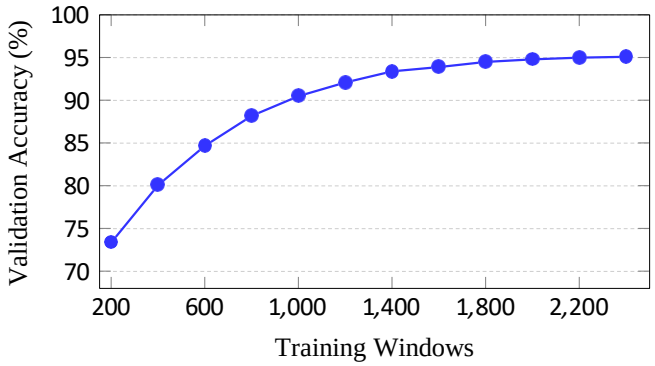

Fig. 6. Validation accuracy of MLP-M QAT as a function of training set size. Performance saturates around 1,800 windows; beyond 25 repetitions per class per participant, gains are negligible.

E. Cross-Subject Generalization

Leave-one-out cross-subject testing dropped accuracy to 88.7%, compared to 95.1% on the within-subject split. That 6.4-point gap is the result we are least satisfied with. It tells us the model has picked up on person-specific movement habits to some degree — variation in gesture speed, joint range of motion, and wrist angle across participants all contribute. We did not attempt any personalisation in this work, so the gap persists. It is likely the biggest open problem for practical deployment.

F. Network-Independence Check

We ran the system for 30 minutes under three radio conditions: BLE and WiFi both active, both disabled, and fully unplugged running on battery alone. Latency varied by less than 0.4 ms across conditions; accuracy was identical in all three. The nRF52840's radio subsystem runs on a separate scheduler from the Cortex-M4 application core, so the inference loop is genuinely isolated from wireless activity.

IX. Applications

A. Healthcare and Fall Detection

Fall detection is one of the more immediate possibilities. The same pipeline could run on a pendant or wristband and distinguish a fall event from normal ambulatory motion, triggering a local alert without needing a phone nearby. Post-injury rehabilitation monitoring is another option: the device could accumulate range-of-motion statistics in flash and upload them periodically rather than streaming continuously.

B. Accessible Input Devices

With end-to-end latency under 50 ms, the response feels instantaneous. That makes gesture-based control viable for slide presentations, smart-home systems, or industrial equipment. For users with limited fine motor control, coarse wrist gestures of the type this system already handles could substitute for interactions that require more precise hand movements.

C. Wearable Fitness Tracking

Running classification on the wrist eliminates the need for a continuous BLE data stream to a phone. We measured approximately 18 mW for continuous raw-data streaming versus 4.2 mW for on-device inference with a once-per-second label broadcast. That difference adds up meaningfully over a full workout session.

D. Industrial and Occupational Safety

Monitoring for ergonomically risky postures — sustained overhead work, repeated ulnar deviation — in factory or warehouse environments is a use case that specifically requires offline operation. Many production facilities restrict wireless data transmission, so a self-contained wearable that identifies high-risk motion patterns locally satisfies that constraint in a way cloud-connected alternatives cannot.

X. Challenges and Limitations

A. Cross-Subject Drift

As noted above, accuracy falls to 88.7% when the model encounters someone outside the training set. For any deployment expected to work out of the box for new users, this is a real obstacle. On-device adaptation would help but is non-trivial: backpropagation requires storing gradients and optimizer state, which can exceed what the Cortex-M4's SRAM can hold. A k-nearest-neighbour adapter over frozen features is likely more tractable and worth investigating in future work.

B. Slow Gestures and Ambiguous Classes

Swipe and punch share similar acceleration profile shapes; what separates them is the direction and spectral content of the peak. When participants slow down, frequency features become uninformative and the classifier falls back on amplitude statistics that are less class-specific. Adding a confidence threshold below which the system outputs "unknown" would reduce visible errors in practice, even if it formally lowers recall.

C. Sensor Placement

All training data was collected with the sensor fixed to the dorsum of the right wrist. Moving the sensor even a few centimetres changes the lever arm and therefore the measured acceleration magnitudes. A normalisation strategy referenced to the gravity vector rather than raw sensor-frame values would likely improve tolerance to placement variation.

D. Scalability

Five gestures with well-separated signatures is a tractable problem. Scaling to twenty classes, or handling transitions between gestures, would require a more expressive model than MLP-M, which pushes against the flash and SRAM ceiling. Migrating to a Cortex-M33 platform with more tightly coupled RAM — some STM32U5 parts offer 256 KB or more — would open up more architectural options while keeping inference fully on-device.

XI. Conclusion

The system we built does what we set out to do: it classifies motion in real time, entirely on-device, with no network dependency. On the Arduino Nano 33 BLE Sense, the INT8 MLP-M model achieves 94.3% top-1 accuracy across five gesture classes at 11.8 ms end-to-end latency and a total firmware footprint of 62.7 KB.

The quantization story was encouraging. INT8 PTQ cost 1.5 accuracy points and cut flash by 3×. QAT recovered 0.8 of those points at the same memory footprint. Looking back, QAT is worth the extra training complexity whenever fractions of a percent in accuracy matter for the deployment context.

The cross-subject generalization gap (88.7% vs. 95.1% within-subject) is the result we are least satisfied with and is the clearest direction for follow-on work. Even a lightweight on-device adaptation mechanism would likely close much of that gap. We also want to explore whether confidence-based gating or a rejection class meaningfully reduces the visible error rate in real-world use.

The broader takeaway is that edge inference for motion classification is not a future capability — it is available now, on commodity hardware, with tooling that is increasingly accessible. The main engineering challenge has shifted from feasibility to disciplined co-design: getting sampling rate, features, and model architecture to fit within memory and latency budgets at the same time is where the real work now lies.

References

[1] Y. Liu, S. Dustdar, et al., "Edge AI: A Survey on Processing and Intelligence at the Network Edge," *ACM Computing Surveys*, vol. 54, no. 8, pp. 1–38, 2022.

[2] R. San-Segundo, J. M. Pardo, J. Ferreiros, V. Sama, A. Rodriguez-Moreno, D. de-la-Hoz, and A. Gallardo-Antolin, "Robust Human Activity Recognition using Smartwatches and Smartphones," *Engineering Applications of Artificial Intelligence*, vol. 72, pp. 190–202, 2018.

[3] C. Forster, L. Carlone, F. Dellaert, and D. Scaramuzza, "IMU Preintegration on Manifold for Efficient Visual-Inertial Estimation," *IEEE Transactions on Robotics*, vol. 33, no. 1, pp. 1–21, 2017.

[4] P. Warden and D. Situnayake, *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*, O'Reilly Media, 2019.

[5] N. D. Lane, P. Georgiev, and L. Qendro, "DeepEar: Robust Smartphone Audio Sensing in Unconstrained Acoustic Environments using Deep Learning," in *Proc. ACM UbiComp*, pp. 283–294, 2015.

[6] C. Zhang, P. Patras, and H. Haddadi, "Deep Learning in Mobile and Wireless Networking: A Survey," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 3, pp. 2224–2287, 2019.