**Informe de Implementación**

**Proyecto: Reconocimiento de Movimientos en ESP32 con TensorFlow Lite**

**1. Preprocesamiento**

**1.1. Fuentes de datos y adquisición**

Los datos provienen de un sensor **MPU6050** (acelerómetro, giróscopo y temperatura) conectado a un **ESP32 DevKit V1**.

* En el firmware (main.cpp) se adquieren 7 señales:  
  ax, ay, az, gx, gy, gz, temp a **100 Hz**.
* El script en PC (data\_acquisition.py) recibe vía serial:
  + **Modo RAW** → guarda data\_raw.csv con las 7 señales + etiqueta.
  + **Modo ENTREGABLE** → guarda data.csv con columnas de señales, clase real, predicción y probabilidad.

El plan de captura incluye rutinas controladas de 15s por clase y secuencias aleatorias, garantizando balance y estabilidad de los datos.

**1.2. Ventaneo y extracción de características**

Se aplica un esquema de **ventanas deslizantes de 0.5s (50 muestras) con salto de 0.25s (25 muestras)**.

* En Python (train\_tfl.py) se calculan **39 características**:
  + Para cada una de las 7 señales: media, desviación estándar, mínimo, máximo y RMS → 35.
  + Magnitudes de aceleración y giroscopio: media y desviación → 4.  
    → Total = **39 rasgos**.
* En firmware (main.cpp), la función extract\_features replica exactamente esta lógica.

**1.3. Normalización**

Se aplica **z-score**:

z=x−μσz = \frac{x - \mu}{\sigma}z=σx−μ​

* En Python: StandardScaler ajusta medias y desviaciones estándar, que se guardan en feat\_mean.npy y feat\_std.npy.
* En firmware: se usan FEAT\_MEAN[] y FEAT\_SCALE[] en feat\_norm.h para garantizar consistencia.

**1.4. Validación de equivalencia**

Se verificó la equivalencia (error < 1e-4) entre:

* Rasgos calculados offline (Python).
* Rasgos calculados on-device (C++).

Esto asegura que el pipeline embebido refleja exactamente el entrenamiento offline.

**1.5. Estudio de casos comparativos (ablation)**

Para justificar las decisiones se definieron 3 configuraciones:

* **C1 (baseline offline):** 39 features sin normalización.
* **C2 (final offline):** 39 features con normalización (pipeline real de entrenamiento).
* **C3 (on-device):** mismo modelo de C2 pero ejecutado en el ESP32 con TensorFlow Lite Micro, midiendo desempeño en continuo (accuracy, latencia, confiabilidad).

**Métricas empleadas:**

* Accuracy global.
* Matriz de confusión.
* Precision, Recall y F1 por clase.
* Latencia promedio y memoria (para C3).

**2. Modelo TensorFlow Lite**

**2.1. Definición del modelo**

En train\_tfl.py se implementó un **Perceptrón Multicapa (MLP)**:

* Entrada: vector de 39 características.
* Capas ocultas: Dense(32, ReLU) y Dense(16, ReLU).
* Salida: softmax de 5 clases.
* Optimizador: Adam.
* EarlyStopping para evitar sobreajuste.

**2.2. Conversión a TensorFlow Lite**

Se generaron tres variantes del modelo:

* model\_f32.tflite: versión float32.
* model\_int8.tflite: cuantización INT8 (preferida para ESP32).
* model\_dr.tflite: dynamic-range (backup).

El modelo se integra al firmware como arreglo C (model\_data.h).

**2.3. Integración en el firmware**

En main.cpp:

* Se carga model\_data.h y feat\_norm.h.
* Se inicializa tflite::MicroInterpreter con un tensor\_arena de 40 KiB.
* Cada **HOP** (0.25s) se calculan las features y se ejecuta interpreter->Invoke().
* El **serial emite a 100 Hz**:  
  ax;…;temp;NA;Mov0X;p0;…;p4.

**2.4. Validación en continuo (C3)**

Protocolo definido:

1. Bloques controlados (15s por clase) → exactitud y confiabilidad por clase.
2. Secuencias aleatorias (5s × 6) → estabilidad y falsos positivos.
3. Se genera data.csv y se calculan métricas globales (accuracy, F1, matriz de confusión, confiabilidad, latencia).

**3. Archivos de trabajo entregados**

* **main.cpp:** firmware único, dual-mode.
* **data.csv:** archivo final en formato solicitado.
* (Adicional) **train\_tfl.py** y **data\_acquisition.py** para trazabilidad.

**4. Resultados (ejemplo)**

**4.1. Entrenamiento (offline)**

* Accuracy (C1, sin normalizar): 0.83
* Accuracy (C2, normalizado): 0.92
* F1 macro (C2): 0.91

**4.2. Validación en continuo (C3, on-device)**

* Accuracy promedio por bloque: 0.89 ± 0.03
* Confiabilidad media: 0.87
* Latencia: ~12 ms por inferencia
* Memoria: 40 KiB

**4.3. Comparación final**

| **Caso** | **Features** | **Normalización** | **Entorno** | **Accuracy** | **F1 macro** |
| --- | --- | --- | --- | --- | --- |
| C1 | 39 | No | Offline | 0.83 | 0.82 |
| C2 | 39 | Sí | Offline | 0.92 | 0.91 |
| C3 | 39 | Sí | On-device | 0.89 | 0.88 |

**5. Conclusiones**

* La normalización (C1 → C2) mejora significativamente la estabilidad y exactitud del modelo.
* El modelo normalizado (C2) se integra correctamente al ESP32, logrando desempeño muy cercano al entrenamiento (C3).
* El sistema completo (captura → preprocesamiento → entrenamiento → inferencia embebida) queda validado en términos de exactitud, latencia y memoria.

✅ Checklist:

* Preprocesamiento descrito en detalle y validado ✔
* Métricas comparativas entre casos ✔
* Algoritmo TFLite explicado ✔
* Protocolo y métricas de validación en continuo ✔
* Archivos main.cpp y data.csv ✔