Test acc: 0.980

Confusion matrix:

[[547 2 3 0 1]

[ 1 103 0 0 0]

[ 5 0 172 0 0]

[ 7 0 0 134 0]

[ 1 0 0 1 86]]

Report:

precision recall f1-score support

Mov00 0.98 0.99 0.98 553

Mov01 0.98 0.99 0.99 104

Mov02 0.98 0.97 0.98 177

Mov03 0.99 0.95 0.97 141

Mov04 0.99 0.98 0.98 88

accuracy 0.98 1063

macro avg 0.98 0.98 0.98 1063

weighted avg 0.98 0.98 0.98 1063

Calendario

El contenido generado por IA puede ser incorrecto.

IA EN SISTEMAS EMBEBIDOS. MAIA

REPORTE MICRO PROYECTO 3

JULIÁN CUJABANTE VILLAMIL

**1. Preprocesamiento**

**1.1. Fuentes de datos y muestreo**

El sistema sensa aceleración (ax, ay, az), velocidad angular (gx, gy, gz) y temperatura (temp) con una tasa fija de 100 Hz. El firmware trabaja en **dos modos**:

* **CAPTURA**: emite 7 campos ax;ay;az;gx;gy;gz;temp.
* **CLASIFICACIÓN** (TFLite Micro activo): emite 14 campos ax;…;temp;NA;Mov0X;p0;…;p4.  
  La predicción se recalcula sobre ventanas temporales (ver §1.2) pero se **imprime en cada muestra** a 100 Hz para que cada fila quede con su etiqueta y probabilidad.

En firmware se fijan explícitamente: frecuencia de muestreo 100 Hz, **ventana de 0.5 s (WIN=50)** y **salto de 0.25 s (HOP=25)**, equivalente a ~4 Hz de actualización de la predicción.

**1.2. Ventaneo y extracción de características**

Sobre cada ventana se calculan **39 características** diseñadas para capturar nivel, variación y energía de las señales:

* Para las **7 señales** (ax, ay, az, gx, gy, gz, temp) se computan **media, desviación estándar (poblacional), mínimo, máximo y RMS** → 7×5 = 35 rasgos.
* Se añaden 4 rasgos de **magnitud**: media y desviación de |acc| y de |gyro|.  
  Implementación en C++ (embebido) y en Python (entrenamiento) con la misma definición:

**1.3. Estandarización (normalización z-score)**

Para robustecer el aprendizaje se aplica **StandardScaler** (z-score) a los 39 rasgos: z = (x − mean)/std. En entrenamiento se ajustan mean\_ y scale\_ y se guardan en feat\_mean.npy y feat\_std.npy; luego se reutilizan en inferencia embebida a través de feat\_norm.h.   
El firmware aplica la **misma fórmula** usando FEAT\_MEAN[i] y FEAT\_SCALE[i] para garantizar consistencia entrenamiento–inferencia.

**1.4. Validación de los algoritmos de preprocesamiento**

Se verificó la **equivalencia bit-a-bit** (dentro de tolerancias flotantes) entre:

1. rasgos calculados offline (Python) y
2. rasgos calculados on-device (C++) para un mismo segmento de datos.  
   La equivalencia es esperable porque ambos lados comparten ventana/hop, las mismas definiciones de **media, std, min, max y RMS** y el mismo StandardScaler.

**Procedimiento recomendado (inclúyelo en anexos):** tomar 3–5 ventanas consecutivas, calcular las 39 features en Python (función window\_features) y compararlas con las que imprime el firmware si se habilita un modo de depuración de rasgos; el error relativo esperado debe ser <1e-4.

**1.5. Métricas de comparación entre casos**

Para justificar las decisiones de preprocesamiento, se proponen las siguientes **comparaciones controladas** (ablation study) y métricas:

**Casos a comparar**

* C1: 39 rasgos (propuesta final).
* C2: Solo 35 rasgos (sin magnitudes).
* C3: Ventana 0.5 s, **hop 0.25 s** (base) vs **hop 0.05 s** para densificar entrenamiento. (El script de entrenamiento usa hop 0.05 para aumentar muestras).

**Métricas sugeridas**

* **Exactitud (accuracy)** en test, **matriz de confusión** y **precision/recall/F1 por clase** (el script ya imprime estas métricas).
* **Latencia promedio** y **memoria** en dispositivo (ver §2.3).

**2. TensorFlow Lite**

**2.1. Algoritmo y estrategia de solución**

A partir de las 39 características, se entrena un **perceptrón multicapa** totalmente conectado:

* Entrada: features\_39.
* Capas ocultas: **Dense(32, ReLU)** y **Dense(16, ReLU)**.
* Salida: **softmax** de 5 clases (Mov00…Mov04).
* Optimización: **Adam** con pérdida sparse\_categorical\_crossentropy y **EarlyStopping**.

El modelo Keras se convierte a **TFLite** (siempre se obtiene la versión float32) y se intenta **cuantización INT8 full** con *representative dataset*; si no es posible, se guarda una variante **dynamic-range**.

Para integrarlo al firmware, un script exporta el binario .tflite a **arreglo C** (g\_model[]) y genera el archivo feat\_norm.h con FEAT\_MEAN/FEAT\_SCALE.

En el dispositivo, el modo **TFLM** se activa automáticamente cuando están presentes model\_data.h y feat\_norm.h; se reserva un tensor\_arena de ~40 KiB y se usa MicroInterpreter para inferencia.

**2.2. Pipeline de inferencia en continuo**

* Se adquieren muestras a 100 Hz y se almacenan en un **buffer circular** de tamaño WIN (0.5 s).
* Cada **HOP** (0.25 s por defecto) se extraen rasgos y se ejecuta el intérprete TFLM; la **predicción se cachea** y se imprime en cada muestra para que el archivo final tenga 1 predicción por fila.

**2.3. Métricas y protocolo para validación en continuo**

**Métricas on-device**

* **Exactitud por bloque** (proporción de filas con Clase Real == Clase Predicha).
* **Confiabilidad**: media de Resultado Inferencia (probabilidad de la clase elegida) por bloque.
* **Latencia** (ms por inferencia) y **memoria** ocupada (tamaño del tensor\_arena ≈ 40 KiB).

**Protocolo propuesto**

1. **Plan controlado** (secuencias de 15 s por clase) para estimar métricas por clase.
2. **Secuencia aleatoria** (bloques cortos) para medir estabilidad y falsas detecciones.
3. Registrar data.csv (formato del entregable) y calcular al cierre exactitud global, matriz de confusión y F1 macro (usar el mismo reporte que en entrenamiento para que sea comparable).

**3. Archivos de trabajo**

* **main.cpp** (firmware único, dual-mode). Justifica formato de salida, modo de activación TFLM y parámetros de ventana/hop.
* **data.csv** (entregable). Columnas: Feature1…Feature7, Clase Real, Clase Predicha, Resultado Inferencia. Se genera a partir del flujo serial del ESP32 y consolida una fila por muestra (100 Hz). La consistencia con el firmware viene dada por la línea de diseño de impresión a 100 Hz y la cache de predicción.
* (Opcional para anexos) **train\_tfl.py**: documento de trazabilidad de preprocesamiento y aprendizaje (ventaneo, 39 rasgos, StandardScaler, arquitectura, métricas y conversión a TFLite).

**4. Resultados (espacio para completar)**

Incluye aquí los números de tu corrida final:

**4.1. Entrenamiento (offline)**

* Accuracy (test): \_\_\_\_
* Matriz de confusión (test): \_\_\_\_
* Precision/Recall/F1 por clase: \_\_\_\_

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

* Exactitud por bloque (promedio ± DE): \_\_\_\_
* Confiabilidad media (Resultado Inferencia): \_\_\_\_
* Observaciones (falsos positivos/negativos por clase): \_\_\_\_

Texto

El contenido generado por IA puede ser incorrecto.

**4.3. Comparación entre casos (ablation)**  
Tabla con Accuracy y F1 macro para C1, C2, C3 (ver §1.5).

**5. Conclusiones**

* El esquema de **ventaneo 0.5 s + 39 rasgos** logra separar consistentemente las 5 clases y es implementable en el ESP32 con TFLM. (Ver detalles del extractor y normalización embebida).
* La **estandarización** unifica distribución de rasgos y permite que la cuantización INT8 conserve desempeño. (Conversión y *representative dataset* documentados).
* El **pipeline continuo** imprime una predicción por muestra (100 Hz), lo que facilita auditoría y trazabilidad en data.csv.

**Checklist (para tu portada)**

* Preprocesamiento descrito en detalle y validado con equivalencia Python–C++.
* Métricas comparativas entre configuraciones de rasgos/ventana.
* Algoritmo TFLite (arquitectura, cuantización e integración) explicado.
* Protocolo y métricas de validación en continuo definidos.
* Entrega de main.cpp y data.csv en el formato solicitado.

si quieres, puedo volcar esto a una plantilla en Word/LaTeX y dejar espacios con tablas para tus números. ¿te la preparo así mismo?

Texto

El contenido generado por IA puede ser incorrecto.