Implementar una red neuronal en Esp32 (Inferencia)

En la primer parte de la práctica se implementó una red neuronal utilizando la librería de Keras y se exportó el modelo a un archivo TensorFlow Lite para posteriormente ser convertido a una matriz de C que pueda ser utilizada en el microcontrolador ESP32.

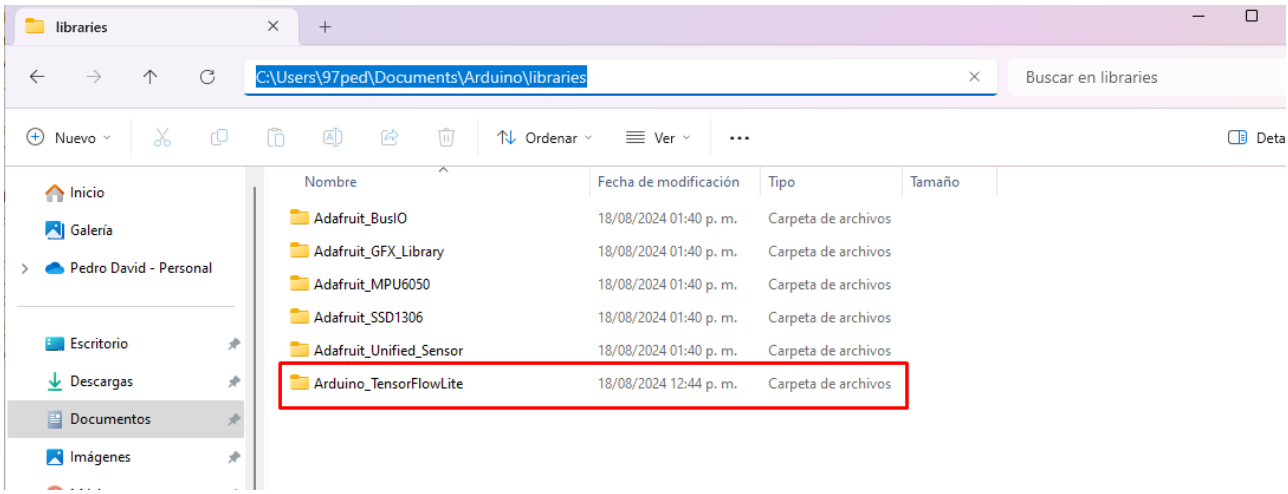
En esta segunda parte de la práctica se realizará la inferencia de la red neuronal en el ESP32. Nos apoyaremos de la librería de TensorFlow Lite para Arduino, la cual nos permite cargar el modelo de la red neuronal y realizar la inferencia en el ESP32.

Instalación de la librería de TensorFlow Lite para Arduino

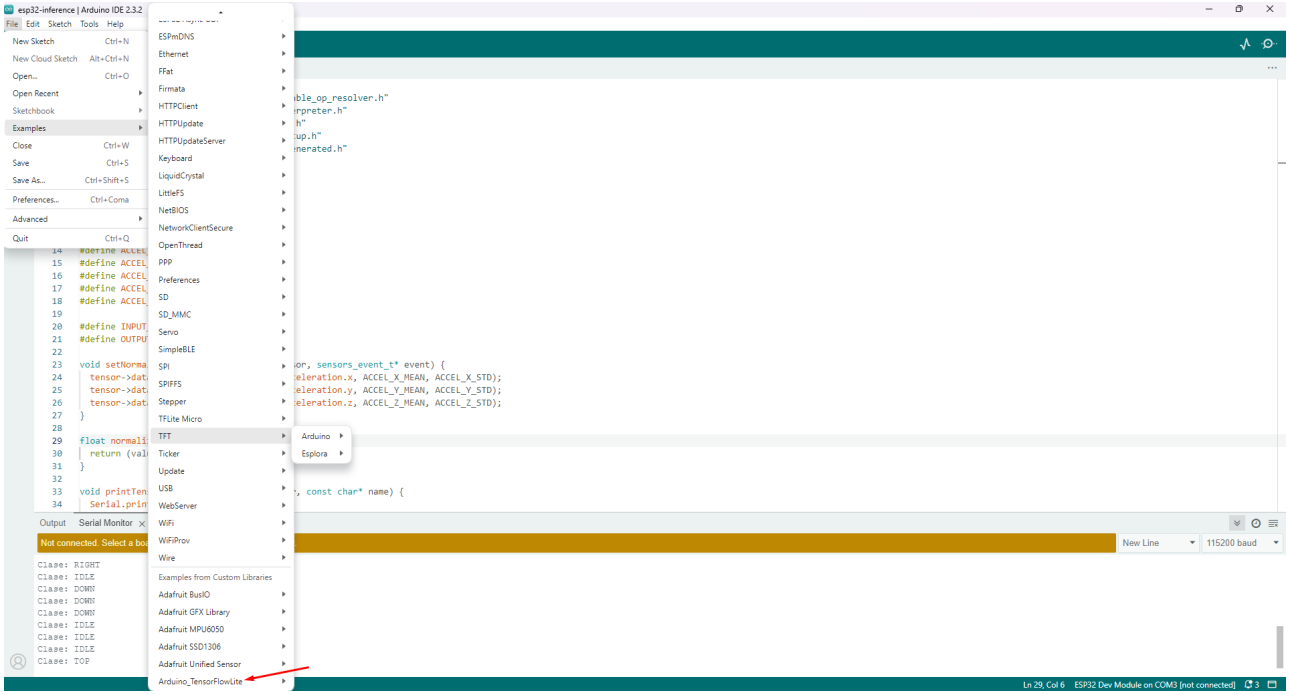
Para instalar la librería de TensorFlow Lite para Arduino, se debe seguir los siguientes pasos:

1. Sigue las instrucciones de la documentación oficial para descargar el repositorio TensorFlow Lite para Arduino o solicite el repositorio a su profesor.

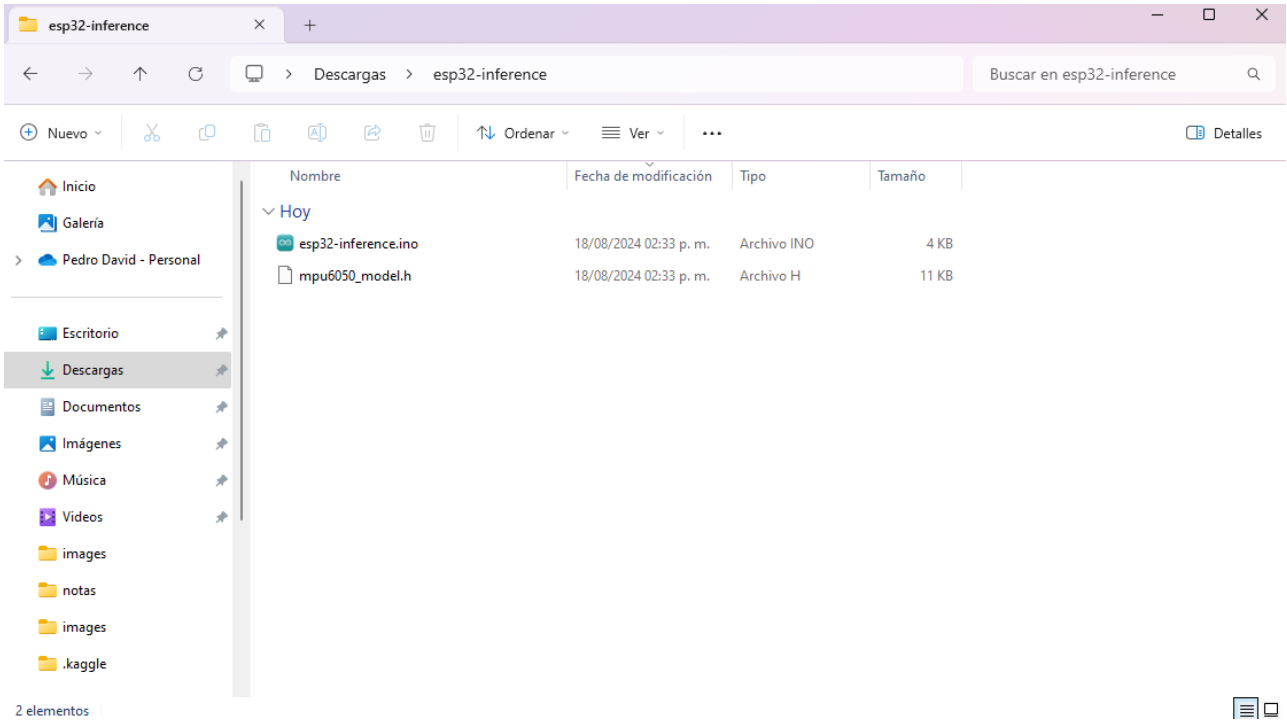
2. Descomprime el archivo descargado y copia la carpeta `Arduino_TensorFlowLite` en la carpeta `libraries` de tu instalación de Arduino.



3. Abre el IDE de Arduino y verifica que la librería de TensorFlow Lite para Arduino se haya instalado correctamente. Para ello, ve a `File > Examples` y verifica que aparezca la carpeta `Arduino_TensorFlowLite`.



4. Como paso adicional, crea un nuevo proyecto en el IDE de Arduino y copia dentro de la carpeta del proyecto el archivo `mpu6050_model.h` generado en la primer parte de la práctica.



Programar el ESP32 para realizar la inferencia

Utilizar la librería de TensorFlow Lite para Arduino puede ser un poco complicado, en las siguientes secciones se explicará a grandes rasgos el código necesario para realizar la inferencia de la red neuronal en el ESP32.

Incluir las librerías necesarias

Para realizar la inferencia de la red neuronal en el ESP32, se deben incluir las siguientes librerías:

```
#include "mpu6050_model.h" // Modelo de la red neuronal

#include "tensorflow/lite/micro/micro_mutable_op_resolver.h"
#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/micro/system_setup.h"
#include "tensorflow/lite/schema/schema_generated.h"

#include <Adafruit_MPU6050.h> // Librería para el sensor MPU6050
#include <Adafruit_Sensor.h> // Librería para el sensor MPU6050
#include <Wire.h> // I2C
```

- `micro_mutable_op_resolver.h` proporciona las operaciones utilizadas por el intérprete para ejecutar el modelo.
- `micro_interpreter.h` contiene el código para cargar y ejecutar modelos.
- `schema_generated.h` contiene el esquema para el formato de archivo TensorFlow Lite FlatBuffer.
- `system_setup.h` contiene la configuración del sistema.

Definir constantes simbólicas

Se deben definir las constantes necesarias para la ejecución del modelo de la red neuronal:

```
#define ACCEL_X_MEAN 1.5626
#define ACCEL_X_STD 6.0826
#define ACCEL_Y_MEAN -0.2553
#define ACCEL_Y_STD 5.6
#define ACCEL_Z_MEAN 2.9114
#define ACCEL_Z_STD 4.2686
```

Los valores de `ACCEL_X_MEAN`, `ACCEL_X_STD`, `ACCEL_Y_MEAN`, `ACCEL_Y_STD`, `ACCEL_Z_MEAN` y `ACCEL_Z_STD` se obtienen de la normalización de los datos de entrada de la red neuronal en la primer parte de la práctica.

Funciones auxiliares

Vamos a definir una función auxiliar para normalizar los datos del acelerómetro y cargarlos en un tensor de entrada para la red neuronal:

```
void setNormalizedAccel(TfLiteTensor* tensor, sensors_event_t* event) {
  tensor->data.f[0] = normalize(event->acceleration.x, ACCEL_X_MEAN, ACCEL_X_STD);
  tensor->data.f[1] = normalize(event->acceleration.y, ACCEL_Y_MEAN, ACCEL_Y_STD);
  tensor->data.f[2] = normalize(event->acceleration.z, ACCEL_Z_MEAN, ACCEL_Z_STD);
}

float normalize(float value, float mean, float std) {
  return (value - mean) / std;
}
```

También vamos a definir una función auxiliar para saber la longitud de un tensor:

```
void printTensorShape(TfLiteTensor* tensor, const char* name) {
  Serial.print(name);
  Serial.print(" forma: ");

  for (int i = 0; i < tensor->dims->size; i++) {
    Serial.print(tensor->dims->data[i]);
    if (i < tensor->dims->size - 1) {
      Serial.print(" x ");
    }
  }
  Serial.println();
}
```

Por último, vamos a definir una función auxiliar para obtener la clase predicha por la red neuronal:

```
int getOutputMaxIndex(TfLiteTensor* tensor) { //Obtener el índice máximo de la
salida
  int maxIndex = 0;
  float maxValue = tensor->data.f[0];

  for (int i = 1; i < tensor->dims->data[1]; i++) {
    if (tensor->data.f[i] > maxValue) {
      maxIndex = i;
      maxValue = tensor->data.f[i];
    }
  }

  return maxIndex;
}

String getClassLabel(int index){ //Modificar según tus clases de salida
  switch (maxIndex) {
    case 0:
      return "DOWN";
    case 1:
      return "IDLE";
    case 2:
      return "LEFT";
    case 3:
      return "RIGHT";
    case 4:
      return "TOP";
    default:
      return "UNKNOWN";
  }
}
```

Configurar variables globales

Vamos a definir las variables globales necesarias para la ejecución del modelo de la red neuronal:

```
Adafruit_MPU6050 mpu; // Para interactuar con el sensor MPU6050

constexpr int kTensorArenaSize = 2 * 1024;  // Ajusta según la memoria disponible
uint8_t tensor_arena[kTensorArenaSize]; // Espacio de memoria para el tensor, si la memoria es insuficiente, aumenta el tamaño


const tflite::Model* model; // Modelo de la red neuronal
tflite::MicroInterpreter* interpreter;

TfLiteTensor* input; // Tensor de entrada
TfLiteTensor* output; // Tensor de salida
```

Función `setup`

En la función `setup` vamos a inicializar el sensor MPU6050 y cargar el modelo de la red neuronal:

```
Serial.begin(115200);

  while (!Serial)
    delay(10); // Esperar a que el puerto serie esté disponible (Quitar para producción)


model = tflite::GetModel(mpu6050_model); // Cargar el modelo de la red neuronal, mpu6050_model es la variable que contiene el modelo generado y se encuentra en el archivo mpu6050_model.h


if (model->version() != TFLITE_SCHEMA_VERSION) { // Verificar la versión del modelo
    Serial.println("Modelo incompatible con la versión de TensorFlow Lite Micro");
    return;
}

static tflite::MicroMutableOpResolver<3> resolver; // Resolver las operaciones que utiliza el modelo, en este caso 3 operaciones

resolver.AddFullyConnected(); // Capa densa
resolver.AddRelu();
resolver.AddSoftmax();

static tflite::MicroInterpreter static_interpreter(model, resolver, tensor_arena, kTensorArenaSize, nullptr);
interpreter = &static_interpreter; // Inicializar el intérprete
```

```
interpreter->AllocateTensors(); // Asignar memoria para los tensores

input = interpreter->input(0); // Tensor de entrada
output = interpreter->output(0); // Tensor de salida

printTensorShape(input, "Entrada"); //Debemos verificar que la forma del tensor de
entrada sea la correcta, en este caso (1, 3)
printTensorShape(output, "Salida"); //Debemos verificar que la forma del tensor de
salida sea la correcta, en este caso (1, 5)

if (!mpu.begin()) {
    Serial.println("Failed to find MPU6050 chip");
    while (1) {
      delay(10);
    }
}

mpu.setAccelerometerRange(MPU6050_RANGE_8_G);
mpu.setGyroRange(MPU6050_RANGE_500_DEG);
mpu.setFilterBandwidth(MPU6050_BAND_5_HZ);
```

Función `loop`

En la función `loop` vamos a realizar la inferencia de la red neuronal:

```
sensors_event_t a, g, temp;
mpu.getEvent(&a, &g, &temp); // Obtener los datos del acelerómetro

setNormalizedAccel(input, &a); // Normalizar los datos del acelerómetro y
cargarlos en el tensor de entrada

if (interpreter->Invoke() != kTfLiteOk) { // Invocar el intérprete realizando la
inferencia, esperamos que retorne kTfLiteOk
    Serial.println("Error durante la inferencia");
    return;
}

int maxIndex = getOutputMaxIndex(output); // Obtener el índice de la clase
predicha
String label = getClassLabel(maxIndex); // Obtener la etiqueta de la clase
predicha

Serial.print("Clase predicha: ");
Serial.println(label);

delay(100); // Esperar un tiempo antes de realizar la siguiente inferencia, no es
necesario en producción
```