Week12-part2-Ml-on-EDGE

Steps for Training the Classification Model

1. **Dataset**: We'll use the temperature and humidity data from the **DHT11 dataset**.
2. **Preprocessing**:
   - We'll categorize the temperature into **comfort levels** like "Cold," "Comfort," and "Hot" based on predefined thresholds.
   - We'll use **temperature** and **humidity** as features for the classification.
3. **Model**: We'll train a classification model to predict the comfort level.
4. **Convert to TensorFlow Lite**: We'll convert the model to TensorFlow Lite format for deployment on ESP32.

Step 1: Data Preprocessing for Classification

We will define the comfort level based on the **temperature**:

- **Cold**: Temperature ≤ 20°C
- **Comfort**: 21°C ≤ Temperature ≤ 30°C
- **Hot**: Temperature ≥ 31°C

We'll create a new column `comfort_level` as the **target variable** for classification.

Code for Data Preprocessing

```
import pandas as pd
import numpy as np
import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import LabelEncoder

# Load the dataset
df = pd.read_csv('dht11_data.csv')

# Define comfort levels based on temperature
def get_comfort_level(temperature):
    if temperature <= 20:
        return 'Cold'
    elif 21 <= temperature <= 30:
        return 'Comfort'
    else:
        return 'Hot'
```

```
# Create a new column 'comfort_level' as the target variable
df['comfort_level'] = df['temperature'].apply(get_comfort_level)

# Features: temperature and humidity
X = df[['temperature', 'humidity']].values

# Labels: comfort level classification (Cold, Comfort, Hot)
y = df['comfort_level'].values

# Encode labels as integers (Cold=0, Comfort=1, Hot=2)
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y)

# Normalize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split into training and testing datasets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_encoded,
test_size=0.2, random_state=42)

# Build a classification model
model = tf.keras.Sequential([
    tf.keras.layers.InputLayer(input_shape=(2,)),  # 2 features (temperature,
humidity)
    tf.keras.layers.Dense(8, activation='relu'),
    tf.keras.layers.Dense(3, activation='softmax')  # 3 classes (Cold,
Comfort, Hot)
])

# Compile the model
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])

# Train the model
model.fit(X_train, y_train, epochs=50, batch_size=32)

# Evaluate the model
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Test Loss: {loss}, Test Accuracy: {accuracy}")
```

Explanation:

- We are classifying the environment into three categories: **Cold**, **Comfort**, and **Hot** based on the **temperature**.
- The **temperature** and **humidity** are the features, while the **comfort level** is the target.
- We use **`LabelEncoder`** to convert the comfort level labels (Cold, Comfort, Hot) into integers (0, 1, 2).
- The model uses **`sparse_categorical_crossentropy`** loss for multi-class classification.

Step 2: Convert the Model to TensorFlow Lite

After training the model, we can convert it to TensorFlow Lite format for deployment on the **ESP32**.

```
# Convert the trained model to TensorFlow Lite format
converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
tflite_model = converter.convert()

# Save the converted TFLite model
with open('dht11_classification_model.tflite', 'wb') as f:
    f.write(tflite_model)
```

Step 3: Prepare for ESP32 Deployment

Next, we will convert the `.tflite` model to a C array that can be embedded into the Arduino sketch.

Use the following command to convert the `.tflite` model into a C array:

Step 4: Arduino Code to Run the Model on ESP32

Here's an updated version of the **Arduino code** (`ESP32_DHT_TFLite.ino`) to work with both **temperature and humidity** for classification:

```
#include <TensorFlowLite.h>
#include <DHT.h>

// DHT sensor configuration
#define DHTPIN 4  // Pin connected to DHT11
#define DHTTYPE DHT11

DHT dht(DHTPIN, DHTTYPE);

// Define the model
extern "C" const unsigned char model_data[];  // TFLite model array
extern "C" const int model_data_len;

tflite::MicroInterpreter* interpreter;
tflite::Model* model;
tflite::ops::micro::OpResolver<4> resolver;  // Resolving operations for the
model
```

```
void setup() {
  Serial.begin(115200);
  dht.begin();

  // Load the TensorFlow Lite model
  model = tflite::GetModel(model_data);
  interpreter = new tflite::MicroInterpreter(model, resolver, model_data_len,
4 * 1024);  // 4KB memory buffer
  interpreter->AllocateTensors();

  Serial.println("TensorFlow Lite model loaded successfully!");
}

void loop() {
  // Read temperature and humidity
  float temperature = dht.readTemperature();
  float humidity = dht.readHumidity();

  if (isnan(temperature) || isnan(humidity)) {
    Serial.println("Failed to read from DHT sensor!");
    return;
  }

  // Input the data to the model
  float* input = interpreter->input(0)->data.f;
  input[0] = temperature;  // Temperature
  input[1] = humidity;     // Humidity

  // Run the model
  interpreter->Invoke();

  // Get the output
  float* output = interpreter->output(0)->data.f;

  // Get the predicted class (Cold, Comfort, Hot)
  int predicted_class = std::distance(output, std::max_element(output, output
+ 3));

  if (predicted_class == 0) {
    Serial.println("Predicted: Cold");
  } else if (predicted_class == 1) {
    Serial.println("Predicted: Comfort");
  } else {
    Serial.println("Predicted: Hot");
  }

  delay(2000);  // Wait for 2 seconds
}
```

Step 5: Upload to ESP32

1. **Prepare your ESP32**:
    - Select your ESP32 board model in Arduino IDE (`Tools` > `Board` > `ESP32 Dev Module`).

    - Set the correct port (`Tools` > `Port`).
2. **Upload** the code.
3. **Open the Serial Monitor** to see the predicted comfort level based on temperature and humidity.

Conclusion

By using both **temperature** and **humidity** as input features, this classification model predicts whether the environment is **Cold**, **Comfort**, or **Hot**.

the **full Python script** to train a simple TensorFlow classification model using synthetic DHT11-like temperature and humidity data. It saves the model as `model.h5`—ready to be used with your Firebase Cloud Function.

✅ Python Script: `train_dht_model.py`

```
import numpy as np
import pandas as pd
import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler

# Step 1: Generate Synthetic DHT11-like Data
np.random.seed(42)
data_size = 300

temperature = np.random.uniform(15, 35, size=data_size)  # Celsius
humidity = np.random.uniform(30, 90, size=data_size)      # Percent

# Step 2: Create Rule-Based Labels
labels = []
for t, h in zip(temperature, humidity):
    if t < 20 or h < 40:
        labels.append("Cold")
    elif 20 <= t <= 28 and 40 <= h <= 70:
        labels.append("Comfort")
```

```
    else:
        labels.append("Hot")

df = pd.DataFrame({
    "temperature": temperature,
    "humidity": humidity,
    "label": labels
})

# Step 3: Encode Labels
label_encoder = LabelEncoder()
df["label_encoded"] = label_encoder.fit_transform(df["label"])

# Step 4: Prepare Features and Target
X = df[["temperature", "humidity"]].values
y = tf.keras.utils.to_categorical(df["label_encoded"].values, num_classes=3)

# Step 5: Scale Features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Step 6: Train/Test Split
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y,
test_size=0.2, random_state=42)

# Step 7: Build TensorFlow Model
model = tf.keras.Sequential([
    tf.keras.layers.Dense(16, activation='relu', input_shape=(2,)),
    tf.keras.layers.Dense(8, activation='relu'),
    tf.keras.layers.Dense(3, activation='softmax')
])

model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])

# Step 8: Train Model
model.fit(X_train, y_train, validation_split=0.2, epochs=50, verbose=1)

# Step 9: Evaluate and Save Model
loss, accuracy = model.evaluate(X_test, y_test, verbose=0)
print(f"Test Accuracy: {accuracy:.2f}")

model.save("model.h5")
print("✅ model.h5 saved successfully!")
```

✅ Updated Script to Convert `.h5` → `.tflite`

Add the following **after** training and saving the `model.h5` in your existing script:

```
# Step 11: Convert to TensorFlow Lite format
converter = tf.lite.TFLiteConverter.from_keras_model(model)
```

```
tflite_model = converter.convert()

# Step 12: Save the .tflite file
with open("model.tflite", "wb") as f:
    f.write(tflite_model)

print("✅ Saved model.tflite for ESP32")
```

🧠 Notes for ESP32 Deployment

To deploy this `.tflite` model on ESP32:

1. Use **TensorFlow Lite for Microcontrollers (TFLM)** — not regular TensorFlow Lite.
2. You'll need to:
   - Convert `model.tflite` to a C array using `xxd` or Python.
   - Include that array in your Arduino sketch.
   - Use the TFLM Arduino library.

🔧 Convert `.tflite` to C Array (example):

✅ Python Script (Recommended)

Cross-platform and reliable. Just install Python (if not already):

Python Script (copy-paste-ready):

```
def bin_to_c_array(input_file, output_file, array_name="model_tflite"):
    with open(input_file, "rb") as f:
        data = f.read()

    with open(output_file, "w") as f:
        f.write(f"const unsigned char {array_name}[] = {{\n")
        for i in range(0, len(data), 12):
            chunk = data[i:i+12]
            line = ", ".join(f"0x{b:02x}" for b in chunk)
            f.write("  " + line + ",\n")
        f.write("};\n")
        f.write(f"const unsigned int {array_name}_len = {len(data)};\n")

# Use like this:
bin_to_c_array("dht_classifier_model.tflite", "model_data.h")
```

To **load a TensorFlow Lite model into PSRAM** on an **ESP32-S3** for inference (instead of internal SRAM), you need to follow these key steps:

✅ 1. Use an ESP32-S3 with PSRAM Enabled

- Ensure your ESP32-S3 board has PSRAM (most dev boards do, like ESP32-S3-DevKitC-1 with 8MB PSRAM).
- In the Arduino IDE:
  - Tools → "PSRAM" → **Enabled**
  - Board → Select **ESP32S3 Dev Module**

✅ 2. Store the TensorFlow Lite model in PSRAM

Use the `PROGMEM` keyword **without** forcing it into internal SRAM. Also, **dynamically copy it** to a pointer in PSRAM before loading.

Example:

```
#include <esp_heap_caps.h>
#include "model_data.h"  // Your .tflite model as a C array

const unsigned char* tflite_model_data = nullptr;

void setup() {
  Serial.begin(115200);

  // Allocate space in PSRAM
  tflite_model_data = (const unsigned char*)heap_caps_malloc(
      dht_classifier_model_tflite_len, MALLOC_CAP_SPIRAM);

  if (tflite_model_data == nullptr) {
    Serial.println("Failed to allocate model in PSRAM!");
    while (1);
  }

  // Copy model from flash to PSRAM
  memcpy((void*)tflite_model_data, dht_classifier_model_tflite,
dht_classifier_model_tflite_len);

  // Load model from PSRAM
  const tflite::Model* model = tflite::GetModel(tflite_model_data);
  if (model->version() != TFLITE_SCHEMA_VERSION) {
```

```
    Serial.println("Model version mismatch!");
    while (1);
  }

  // Continue with interpreter setup...
}
```

✅ 3. Use TensorArena in PSRAM too

```
constexpr int kTensorArenaSize = 40 * 1024;
uint8_t* tensor_arena = (uint8_t*)heap_caps_malloc(kTensorArenaSize,
MALLOC_CAP_SPIRAM);
```

This ensures **both your model** and **tensor buffers** are offloaded to external PSRAM, avoiding internal memory limitations.

🔥 Summary:

- Use `heap_caps_malloc(..., MALLOC_CAP_SPIRAM)` to allocate PSRAM.
- Use `memcpy` to move `model_data` from flash to PSRAM.
- Pass that pointer to `tflite::GetModel()`.

```
// DHT11 + TensorFlow Lite on ESP32-S3 using PSRAM

#include <Arduino.h>
#include <DHT.h>
#include <WiFi.h>
#include <esp_heap_caps.h>

// TensorFlow Lite Micro
```

```
#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/micro/all_ops_resolver.h"
#include "tensorflow/lite/schema/schema_generated.h"
#include "tensorflow/lite/version.h"

#include "model_data.h"  // Your converted .h model file

#define DHTPIN 4
#define DHTTYPE DHT11
DHT dht(DHTPIN, DHTTYPE);

constexpr int kTensorArenaSize = 40 * 1024;
uint8_t* tensor_arena;

const unsigned char* model_data_ptr = nullptr;
const tflite::Model* model = nullptr;
tflite::MicroInterpreter* interpreter = nullptr;

void setup() {
  Serial.begin(115200);
  dht.begin();

  // Allocate PSRAM for model
  model_data_ptr = (const unsigned
char*)heap_caps_malloc(model_tflite_len,
MALLOC_CAP_SPIRAM);
  if (!model_data_ptr) {
    Serial.println("Failed to allocate PSRAM for
model!");
    while (1);
  }
```

```
  memcpy((void*)model_data_ptr, model_tflite,
model_tflite_len);

  // Load model
  model = tflite::GetModel(model_data_ptr);
  if (model->version() != TFLITE_SCHEMA_VERSION) {
    Serial.println("Model version mismatch!");
    while (1);
  }

  // Allocate PSRAM for tensor arena
  tensor_arena =
(uint8_t*)heap_caps_malloc(kTensorArenaSize,
MALLOC_CAP_SPIRAM);
  if (!tensor_arena) {
    Serial.println("Failed to allocate tensor arena in
PSRAM!");
    while (1);
  }

  // Set up the interpreter
  static tflite::AllOpsResolver resolver;
  static tflite::MicroInterpreter
static_interpreter(model, resolver, tensor_arena,
kTensorArenaSize);
  interpreter = &static_interpreter;

  if (interpreter->AllocateTensors() != kTfLiteOk) {
    Serial.println("Tensor allocation failed");
    while (1);
  }
```

```
  Serial.println("Setup complete. Starting
inference...");
}

void loop() {
  float temperature = dht.readTemperature();
  float humidity = dht.readHumidity();

  if (isnan(temperature) || isnan(humidity)) {
    Serial.println("Failed to read from DHT sensor!");
    delay(2000);
    return;
  }

  TfLiteTensor* input = interpreter->input(0);
  input->data.f[0] = temperature;
  input->data.f[1] = humidity;

  if (interpreter->Invoke() != kTfLiteOk) {
    Serial.println("Inference failed");
    return;
  }

  TfLiteTensor* output = interpreter->output(0);
  Serial.print("Temperature: ");
  Serial.print(temperature);
  Serial.print(" C, Humidity: ");
  Serial.print(humidity);
  Serial.print(" %, Prediction: ");
  Serial.println(output->data.f[0]);

  delay(5000);
```

```
}
```

TensorFlow Lite (TFLite) model on the **ESP32-S3** using **PSRAM** (external RAM) to make predictions based on data from a **DHT11 sensor** (temperature and humidity). Below is an explanation of the important portions of the code.

1. Library Inclusions

```
#include <Arduino.h>
#include <DHT.h>
#include <WiFi.h>
#include <esp_heap_caps.h>
```

- **Arduino.h**: Required for using the Arduino functions (ESP32-specific).
- **DHT.h**: Library to read temperature and humidity data from the DHT11 sensor.
- **WiFi.h**: Provides Wi-Fi connectivity (though it's not directly used in this snippet, it could be for features like time synchronization or data upload).
- **esp_heap_caps.h**: Allows you to allocate memory from specific regions like **PSRAM** (external RAM) using `heap_caps_malloc()`.

2. TensorFlow Lite Micro Libraries

```
#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/micro/all_ops_resolver.h"
#include "tensorflow/lite/schema/schema_generated.h"
#include "tensorflow/lite/version.h"
```

These libraries are for integrating **TensorFlow Lite Micro** into the ESP32, which is a lightweight version of TensorFlow Lite designed to run on embedded devices with limited resources.

- **micro_interpreter.h**: Manages the model's execution and handles tensor allocation and interpretation.
- **all_ops_resolver.h**: Resolves the operations needed to run the model.
- **schema_generated.h**: Contains the schema of the model, generated during model conversion.
- **version.h**: Contains the TensorFlow Lite version.

3. Model Data Header File

```
#include "model_data.h"  // Your converted .h model file
```

- This header file contains the **converted TensorFlow Lite model** in a format that can be used by the ESP32. The model is converted into a byte array (from `.tflite` to `.h`) using tools like `xxd` or the `xxd` tool.
- The model is stored in a `model_tflite` byte array in the header file (`model_tflite` is a part of the conversion).

4. Pin Definition and DHT Sensor Initialization

```
#define DHTPIN 4
#define DHTTYPE DHT11
DHT dht(DHTPIN, DHTTYPE);
```

- The **DHT11 sensor** is connected to **GPIO pin 4** on the ESP32.
- An instance of the **DHT** class is created using the defined pin and sensor type.

5. Memory Allocation for PSRAM (Model and Tensor Arena)

```
model_data_ptr = (const unsigned char*)heap_caps_malloc(model_tflite_len,
MALLOC_CAP_SPIRAM);
```

- **PSRAM** is used to store the **model** and **tensor arena** because the ESP32 has limited internal SRAM.
- The `heap_caps_malloc()` function allocates memory from PSRAM using the `MALLOC_CAP_SPIRAM` flag.
- The model data is copied from the `model_tflite` byte array into **PSRAM** for later use.

```
tensor_arena = (uint8_t*)heap_caps_malloc(kTensorArenaSize,
MALLOC_CAP_SPIRAM);
```

- Allocates a **tensor arena** in PSRAM to hold the tensors (input and output data for the model) during inference.
- `kTensorArenaSize` is the size of the memory buffer used for the tensor data. It's set to 40 KB in this example, but the exact size will depend on your model.

6. Model Loading and Interpreter Setup

```
model = tflite::GetModel(model_data_ptr);
```

- Loads the **TensorFlow Lite model** from **PSRAM**. `model_data_ptr` points to the model data stored earlier.
- Checks if the model version matches the **TensorFlow Lite version** to ensure compatibility.

```
static tflite::AllOpsResolver resolver;
static tflite::MicroInterpreter static_interpreter(model, resolver,
tensor_arena, kTensorArenaSize);
interpreter = &static_interpreter;
```

- **AllOpsResolver** is used to resolve all the operations (e.g., add, multiply) that the model needs to perform.
- **MicroInterpreter** manages the execution of the TensorFlow Lite model. It is initialized with the **model**, **operation resolver**, **tensor arena**, and **arena size**.
- `interpreter` is the pointer to the interpreter object that will run the inference.

```
if (interpreter->AllocateTensors() != kTfLiteOk) {
    Serial.println("Tensor allocation failed");
    while (1);
}
```

- Allocates tensors for input and output. If tensor allocation fails, the program enters an infinite loop.

7. Inference and Data Handling in `loop()`

```
float temperature = dht.readTemperature();
float humidity = dht.readHumidity();
```

- Reads temperature and humidity data from the **DHT11** sensor.
- If reading fails, the function returns and retries after a 2-second delay.

```
TfLiteTensor* input = interpreter->input(0);
input->data.f[0] = temperature;
input->data.f[1] = humidity;
```

- Sets the input data for the model. The temperature and humidity values are stored in the **input tensor** at indices `0` and `1`.

```
if (interpreter->Invoke() != kTfLiteOk) {
    Serial.println("Inference failed");
    return;
}
```

- **Invoke** runs the model inference with the provided input data.
- If the inference fails, the function exits.

8. Output Handling and Result Display

```
TfLiteTensor* output = interpreter->output(0);
Serial.print("Temperature: ");
Serial.print(temperature);
Serial.print(" C, Humidity: ");
Serial.print(humidity);
Serial.print(" %, Prediction: ");
Serial.println(output->data.f[0]);
```

- **Gets the output tensor** after invoking the model.
- The output tensor contains the prediction (e.g., a classification result).
- **Outputs** the temperature, humidity, and the prediction to the Serial Monitor.

9. Delay Between Measurements

```
delay(5000);
```

- A **5-second delay** before the next loop, allowing time for the DHT11 sensor to provide new data.

Summary of Important Portions:

1. **PSRAM Usage**: The model and tensor arena are allocated in **PSRAM** to efficiently manage memory on the ESP32-S3.
2. **Model Loading**: The **TensorFlow Lite model** is loaded from PSRAM into the `MicroInterpreter` for inference.
3. **Sensor Data**: **Temperature and humidity** readings from the DHT11 sensor are passed to the model as input.
4. **Inference**: The model performs inference with the sensor data, and the result is outputted to the Serial Monitor.
5. **Memory Management**: PSRAM is used for both the model data and tensor arena to avoid exhausting the limited internal SRAM.