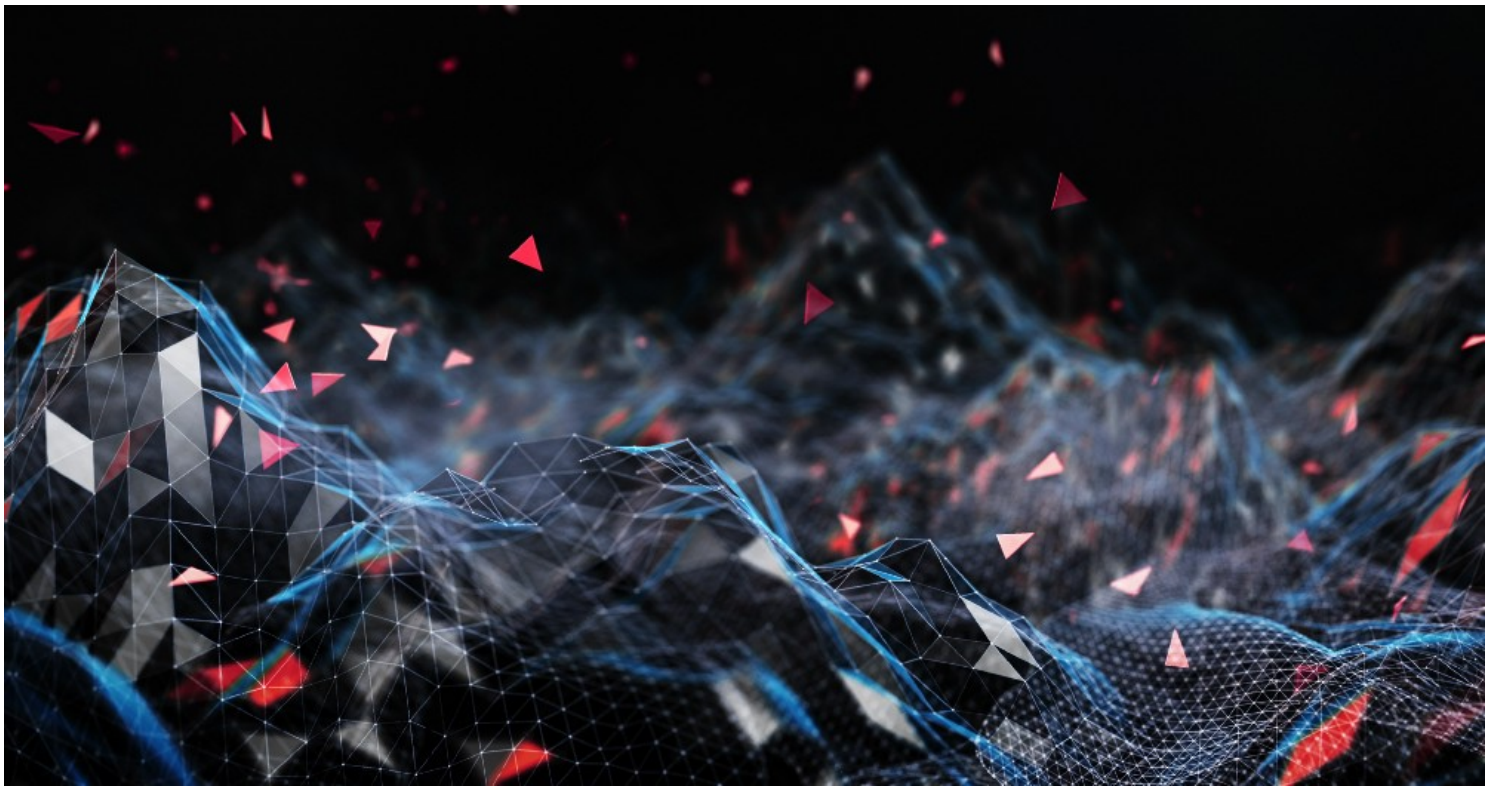
TensorFlow, Meet The ESP32

How to set up a TensorFlow Lite environment for the ESP32

Wezley Sherman Dec 2, 2019 · 6 min read ★



If I had to choose a favorite field of computing, I would choose Embedded Systems. I'm a

fan of the challenge that optimizing code to run in a resource-constrained environment presents.

When I realized that TensorFlow Lite supports microcontrollers, I flipped my desk — with excitement.

I have a couple of the ESP32-CAM modules that I'm wanting to use with a home security system. My idea is to deploy a model that recognizes people and starts recording as soon as the camera picks up on a person. I don't want to use motion-sensing here because I have a dog who would just trip the sensors.

I see TensorFlow Lite as being a great tool for this use-case. I can train a model to recognize people on my desktop, and then deploy it to my ESP32-CAM modules.

After reading through the TensorFlow Lite documentation it became apparent that using PlatformIO wasn't going to be as easy as calling:

```
platformio lib install tfmicro
```

Unless I want to use an Adafruit library (**spoiler**: I don't).

With some careful trial and error, I was able to get TensorFlow Lite to play nice with PlatformIO on the ESP32.

This guide will go through how I got the "Hello World" example compiling and uploaded to an ESP32-CAM module using PlatformIO with Arduino-ESP32 support.

. . .

Setting up TensorFlow Lite for PlatformIO Deployment

The first thing you'll want to do is install PlatformIO. To do so open up a terminal and type:

```
pip install -U platformio
```

Now, create your project's root directory. This directory should also contain sub-directories for **src**, **lib**, and **include**.

Within your project's root directory, create a file named **platformio.ini**. This file will contain all of the information needed for PlatformIO to initialize your development environment. Mine looks like:

```
[env:esp32doit-devkit-v1]
platform = espressif32
board = esp32doit-devkit-v1
framework = arduino
board_build.partitions = custom.csv
lib_deps=tfmicro
```

platformio.ini hosted with ♥ by GitHub view raw

platformio.ini

Next, you'll need to create a file named **custom.csv**. This is the partition information for the ESP32's flash. You can format the partition table based on your application's needs and ESP32's flash size. More information on ESP32 partition tables can be found here. Below is how my **custom.csv** file is formatted:

Search this file…

# Name	Type	SubType	Offset	Size	Flags
nvs	data	nvs	0x9000	20K	
otadata	data	ota	0xe000	8K	
firm	app	ota_0		3400K	
eeprom	data	0x99		4K	
spiffs	data	spiffs		444K	

custom.csv hosted with ♥ by GitHub view raw

custom.csv — The custom partition table I am using for the ESP32

With all of that set up, go ahead and clone the TensorFlow repository (TensorFlow Lite is included). You can download this as a .zip and extract to a directory of your choosing, or clone using git:

```
git clone https://github.com/tensorflow/tensorflow.git
```

Once you have the TensorFlow repository downloaded, generate one of the sample ESP32 projects from the TensorFlow Lite folder. We want to generate a sample project so we can grab the **tfmicro** library that is generated and the sample model. To generate the sample project, navigate to the root 'tensorflow' folder and run:

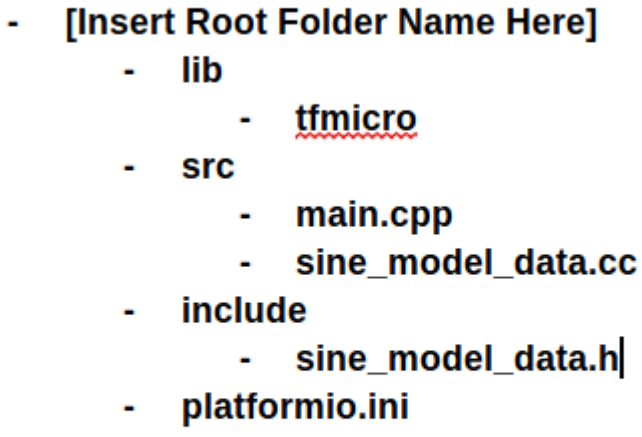
```
sudo make -f tensorflow/lite/experimental/micro/tools/make/Makefile
TARGET=esp generate_hello_world_esp_project
```

This will generate a sample project that'll be located in:

```
tensorflow/lite/experimental/micro/tools/make/gen/esp_xtensa-
esp32/prj/hello_world
```

Navigate to the **hello_world/esp-idf** directory and copy the **tfmicro** folder from **components** into the **lib** folder that you created earlier. If you plan on running the sample program to ensure you set up your environment correctly, then copy over **sin_model_data.cc** and **sine_model_data.h** from the **main** folder to your **src** and **include** folders.

Your file structure should now look something like:



- custom.csv

Project Directory Structure

Pretty easy so far, huh?

You are almost done! You just need to tweak a few things in the **tfmicro** library folder so PlatformIO can see all the third-party libraries TensorFlow Lite requires.

Navigate into the **tfmicro** folder. You should see two folders: **tensorflow** and **third_party**.

Go into **third_party/flatbuffers/include** and copy the **flatbuffers** sub-directory into the **tfmicro** root.

Next, go into **third_party/gemmlowp** and copy both the **fixedpoint** and **internal** directories into the **tfmicro** root directory.

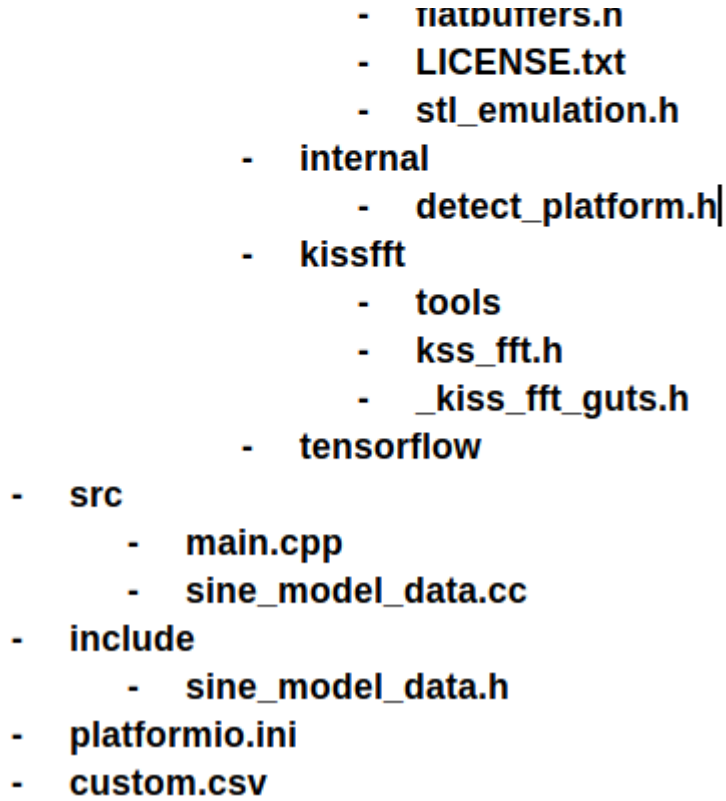
Finally, copy over the **kissfft** directory from **third_party** into the **tfmicro** root directory.

At this point, you can go ahead and delete the **third_party** directory.

By moving all of the third-party libraries into the **tfmicro** root, PlatformIO can recognize and use them.

The final structure of your project should look like:

- [Insert Root Folder Name Here]
  - lib
    - tfmicro
      - fixedpoint
        - fixedpoint.h
        - fixedpoint_sse.h
        - LICENSE
      - flatbuffers
        - base.h



File structure for a TensorFlow Lite ESP32 Project

With the file structure finished, navigate to **lib/tfmicro/flatbuffers** and open **base.h**.Within **base.h**, change line 34 from:

```
#if defined(ARDUINO) && !defined(ARDUINOSTL_M_H)
  #include <utility.h>
#else
  #include <utility>
#endif
```

to:

```
#include <utility>
```

You are now done! The only thing that's left to do is import and use TensorFlow Lite in a project.

. . .

Building A Sample Project

To test that TensorFlow Lite was working correctly, I adapted the "Hello World!" sine model to be used in an Arduino-ESP32 sketch.

I'll guide you through what I did below!

The Includes

The first thing I did was import all of the libraries the project will use.

```
#include <Arduino.h>
#include <math.h>
#include "tensorflow/lite/experimental/micro/kernels/all_ops_resolver.h"
#include "tensorflow/lite/experimental/micro/micro_error_reporter.h"
#include "tensorflow/lite/experimental/micro/micro_interpreter.h"
#include "sine_model_data.h"
```

TF Lite Imports hosted with ♥ by GitHub view raw

Needed Libraries for TensorFlow Lite Sample Project

The libraries are as follows:

- **Arduino .h**— Arduino support! :)
- **math.h** — Specifically used here for definition of M_PI (Will discuss later).
- **all_ops_resolver.h** — Defines operations used to run the model.
- **micro_error_reporter.h** — Used for error reporting.
- **micro_interpreter.h** — Runs the model.
- **sine_model_data.h** — The sample model we are using.

Global Variables

Next, I set my variables:

```
// Create a memory pool for the nodes in the network
constexpr int tensor_pool_size = 2 * 1024;
uint8_t tensor_pool[tensor_pool_size];

// Define the model to be used
```

```
const tflite::Model* sine_model;

// Define the interpreter
tflite::MicroInterpreter* interpreter;

// Input/Output nodes for the network
TfLiteTensor* input;
TfLiteTensor* output;
```

TFLite Vars hosted with ♥ by GitHub view raw

TensorFlow Lite Variables

The first global variable I defined was the memory pool to store the arrays generated by the model. TensorFlow's documentation states that you may have to come up with the pool size from experimentation for different models. For this, I just went what was in the sample code that TensorFlow provided for running the sine model.

Then I defined the model, interpreter, and input/output nodes.

Setup Design

With global variables defined, it's time to set up the environment.

Within the 'setup()' function, I started serial at 115200 baud and imported the sine model.

```
// Start serial at 115200 baud
Serial.begin(115200);

// Load the sample sine model
Serial.println("Loading Tensorflow model....");
sine_model = tflite::GetModel(g_sine_model_data);
Serial.println("Sine model loaded!");
```

Serial and Sine hosted with ♥ by GitHub view raw

Start Serial and Import Sine

Then I instantiated the TensorFlow Lite Micro Interpreter

```
// Define ops resolver and error reporting
static tflite::ops::micro::AllOpsResolver resolver;
```

```
static tflite::ErrorReporter* error_reporter;
static tflite::MicroErrorReporter micro_error;
error_reporter = &micro_error;

// Instantiate the interpreter
static tflite::MicroInterpreter static_interpreter(
  sine_model, resolver, tensor_pool, tensor_pool_size, error_reporter
);

interpreter = &static_interpreter;
```

Instantiate hosted with ♥ by GitHub view raw

Finally, I allocated the model's tensors into the memory pool that was defined as a global and I set the input and output variables to their corresponding nodes.

```
// Allocate the the model's tensors in the memory pool that was created.
Serial.println("Allocating tensors to memory pool");
if(interpreter->AllocateTensors() != kTfLiteOk) {
  Serial.println("There was an error allocating the memory...ooof");
  return;
}

// Define input and output nodes
input = interpreter->input(0);
output = interpreter->output(0);
Serial.println("Starting inferences... Input a number! ");
```

Set Vars hosted with ♥ by GitHub view raw

That concludes the 'setup()' function. In a gist: I imported the model, created the interpreter, and loaded the model into memory.

Control Loop Design

With the 'setup()' function finished, I started implementing the control loop logic.

For the control loop, I waited for user input from serial and converted it to a float. Then I checked to make sure the user input was within the model's params (0–2*PI):

```
// Wait for serial input to be made available and parse it as a float
if(Serial.available() > 0) {
```

```
    float user_input = Serial.parseFloat();

    /* The sample model is only trained for values between 0 and 2*PI
     * This will keep the user from inputting bad numbers.
     */
    if(user_input < 0.0f || user_input > (float)(2*M_PI)) {
      Serial.println("Your number must be greater than 0 and less than 2*PI");
      return;
    }
```

Control Loop 1 hosted with ♥ by GitHub view raw

I then set the input node for the model to parsed user input, invoked the interpreter, and then printed out the result:

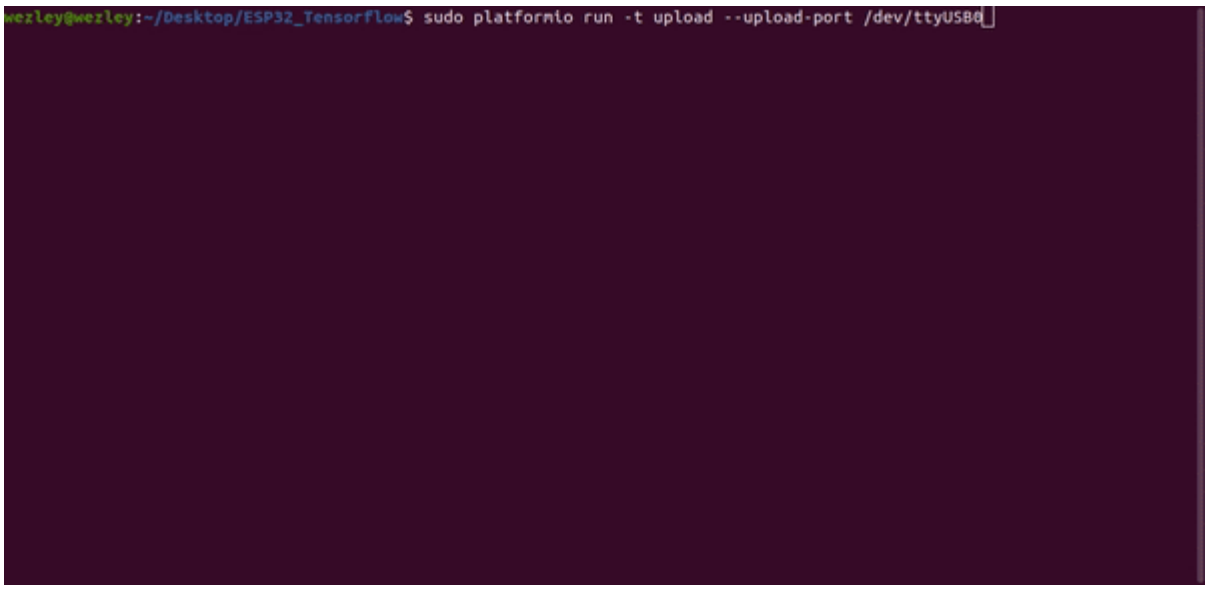
```
  // Set the input node to the user input
  input->data.f[0] = user_input;

  Serial.println("Running inference on inputted data...");

  // Run inference on the input data
  if(interpreter->Invoke() != kTfLiteOk) {
    Serial.println("There was an error invoking the interpreter!");
    return;
  }

  // Print the output of the model.
  Serial.print("Input: ");
  Serial.println(user_input);
  Serial.print("Output: ");
  Serial.println(output->data.f[0]);
  Serial.println("");

}
```

Control Loop 2 hosted with ♥ by GitHub view raw

Deploying to the ESP32

With the code finished, you can deploy to the ESP32 using PlatformIO with:

```
platformio run -t upload --upload-port /dev/ttyUSB0
```



Uploading to the ESP32

You can then give the program input through serial by using:

```
screen /dev/ttyUSB0 115200
```

Running Sine Model on ESP32

That's it! I'm genuinely surprised at how easy TensorFlow makes it to deploy models to microcontrollers. My next article will be on how to create a model using TensorFlow and how to convert it to something that can be run by TensorFlow Lite!

GitHub Project Repo: https://github.com/wezleysherman/ESP32-TensorFlow-Lite-Sample

Programming Esp32 Artificial Intelligence Deep Learning TensorFlow