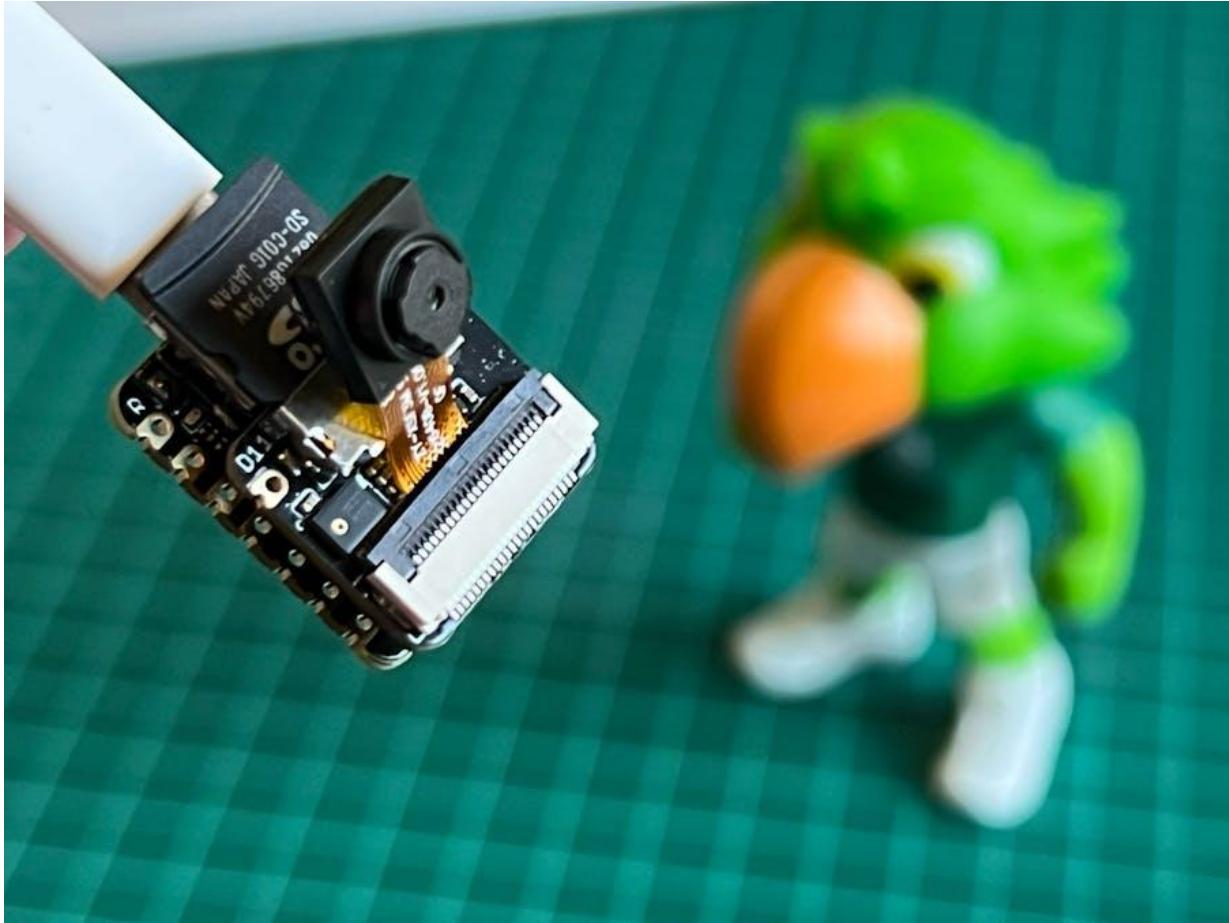


TinyML Made Easy

Image Classification

Exploring Machine Learning on the tremendous new tiny device of the Seeed Studio XIAO family, the ESP32S3 Sense.



MJRoBot (Marcelo Rovai)

Published May 5, 2023, © Apache-2.0

<https://www.hackster.io/mjrobot/tinyml-made-easy-image-classification-cb42ae>

Introduction

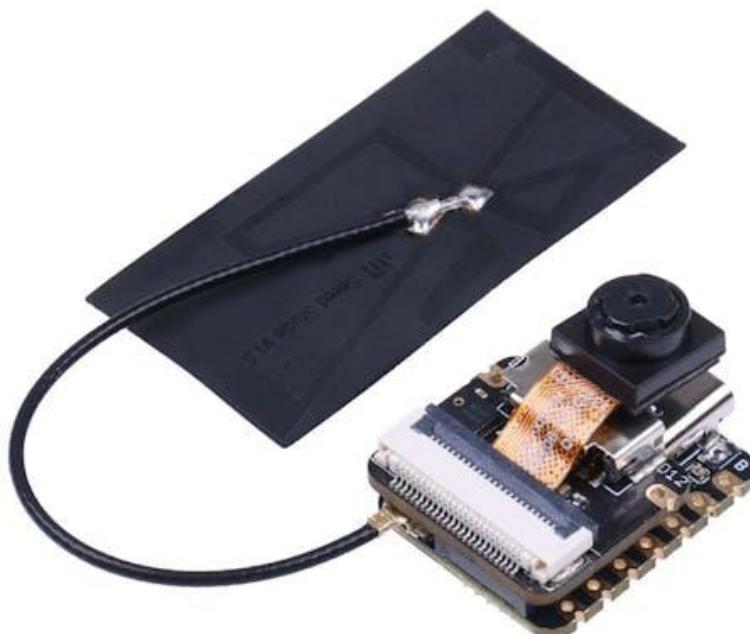
More and more, we are facing an artificial intelligence (AI) revolution where as stated by Gartner, **Edge AI** has a very high impact potential, and **it is for now!**



In the "bull-eye" of emerging technologies, radar is the *Edge Computer Vision*, and when we talk about Machine Learning (ML) applied to vision, the first thing that comes to mind is **Image Classification**, a kind of ML "Hello World"!

Seeed Studio released a new affordable development board, the **XIAO ESP32S3 Sense**, which integrates a camera sensor, digital microphone, and

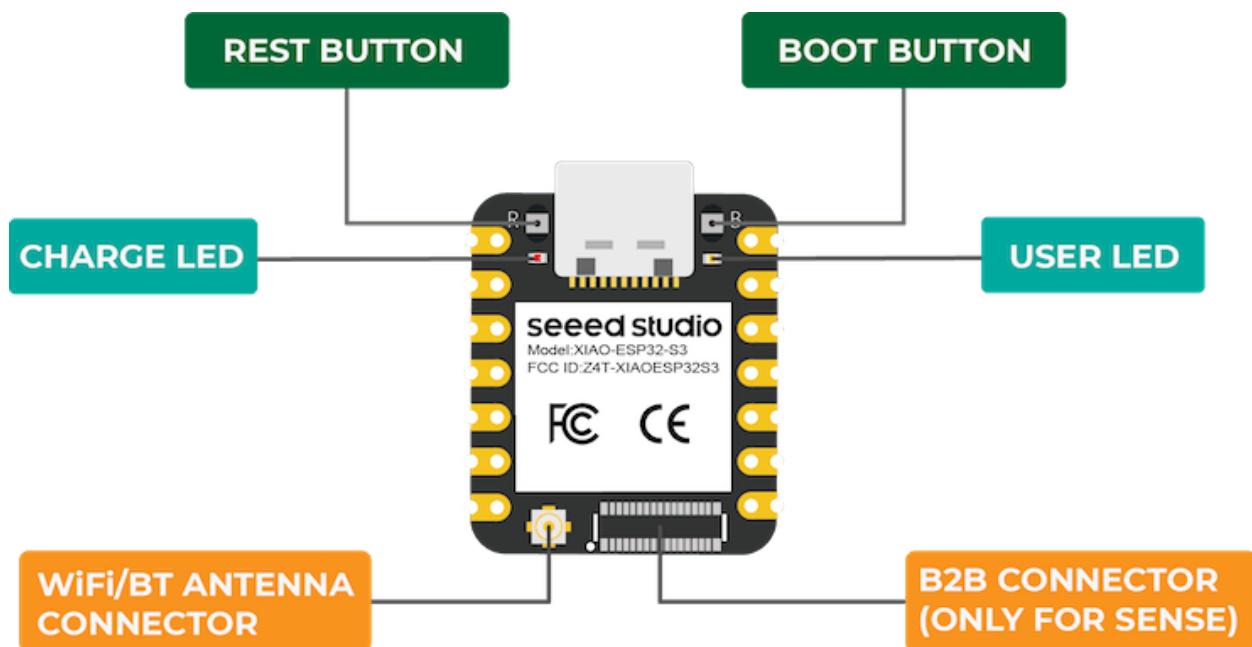
SD card support. Combining embedded ML computing power and photography capability, this development board is a great tool to start with TinyML (intelligent voice and vision AI).



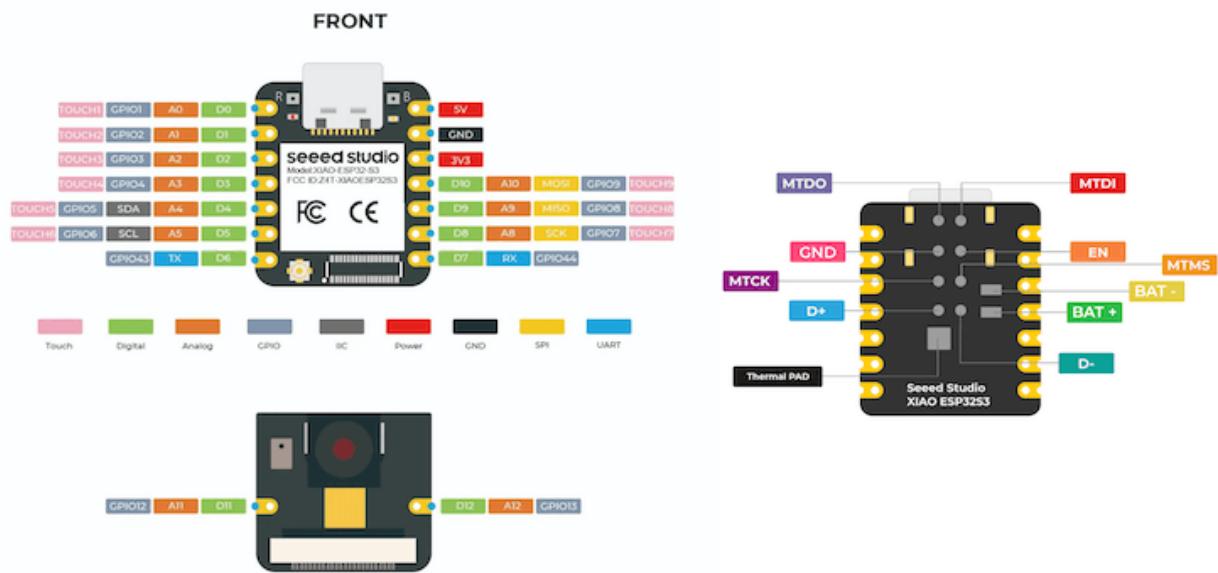
XIAO ESP32S3 Sense Main Features

- **Powerful MCU Board:** Incorporate the ESP32S3 32-bit, dual-core, Xtensa processor chip operating up to 240 MHz, mounted multiple development ports, Arduino / MicroPython supported
- **Advanced Functionality:** Detachable OV2640 camera sensor for 1600*1200 resolution, compatible with OV5640 camera sensor, integrating an additional digital microphone

- **Elaborate Power Design:** Lithium battery charge management capability offer four power consumption model, which allows for deep sleep mode with power consumption as low as 14µA
- **Great Memory for more Possibilities:** Offer 8MB PSRAM and 8MB FLASH, supporting SD card slot for external 32GB FAT memory
- **Outstanding RF performance:** Support 2.4GHz Wi-Fi and BLE dual wireless communication, support 100m+ remote communication when connected with U.FL antenna
- **Thumb-sized Compact Design:** 21 x 17.5mm, adopting the classic form factor of XIAO, suitable for space-limited projects like wearable devices



Below is the general board pinout:



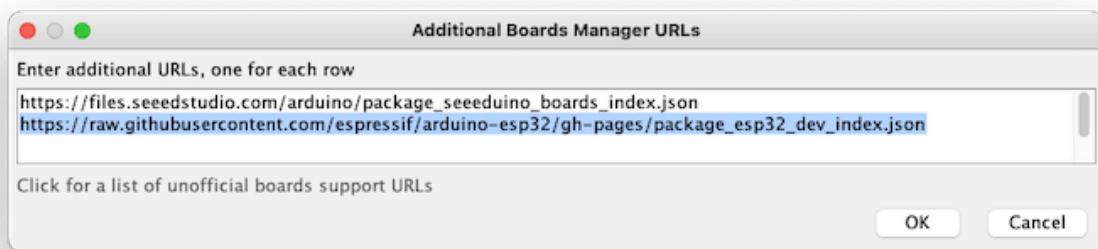
For more details, please refer to Seeed Studio WiKi page:
https://wiki.seeedstudio.com/xiao_esp32s3_getting_started/

Installing the XIAO ESP32S3 Sense on Arduino IDE

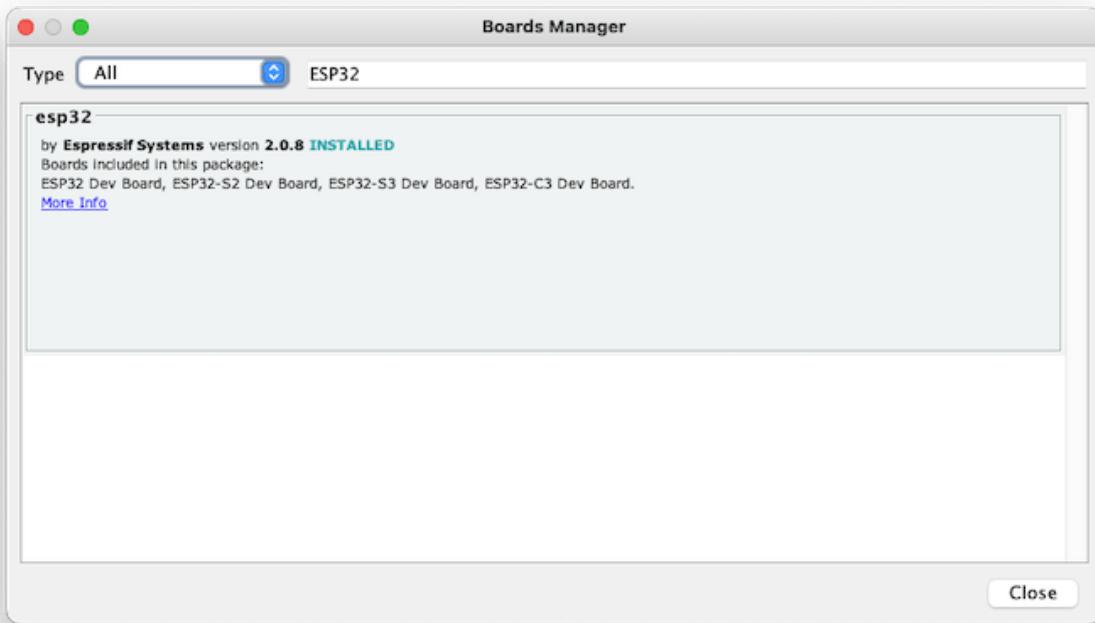
On Arduino IDE, navigate to **File > Preferences**, and fill in the URL:

https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_dev_index.json

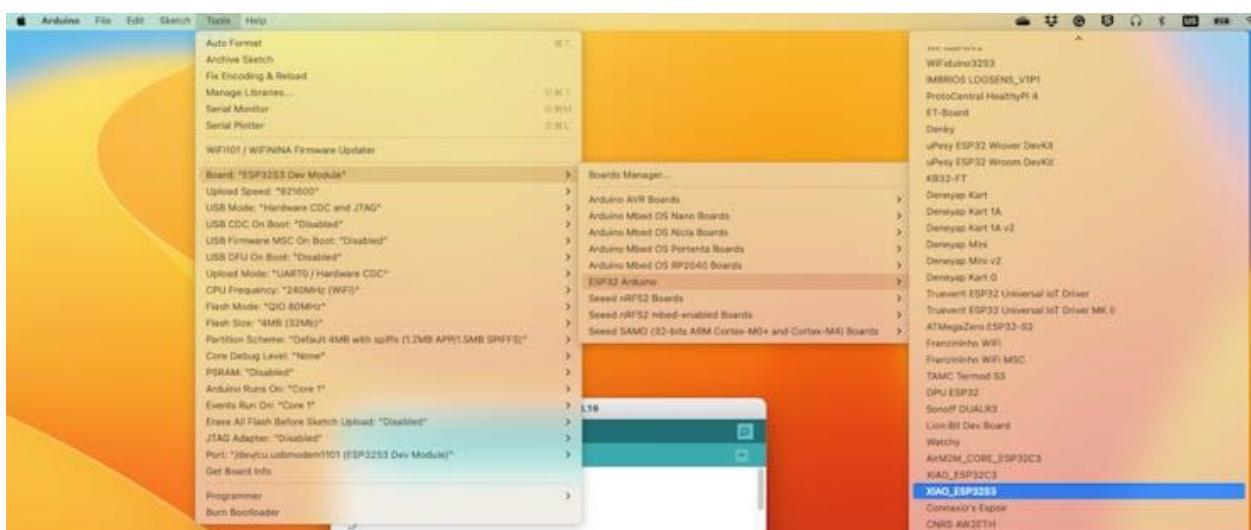
on the field ==> **Additional Boards Manager URLs**



Next, open boards manager. Go to **Tools > Board > Boards Manager...** and enter with **esp32**. Select and install the most updated package:



On **Tools**, select the Board (**XIAO ESP32S3**):



Last, but not least, select the **Port** where the ESP32S3 is connected.

That is it! The device should be OK. Let's do some tests.

Testing the board with BLINK

The XIAO ESP32S3 Sense has a built-in LED that is connected to GPIO21.

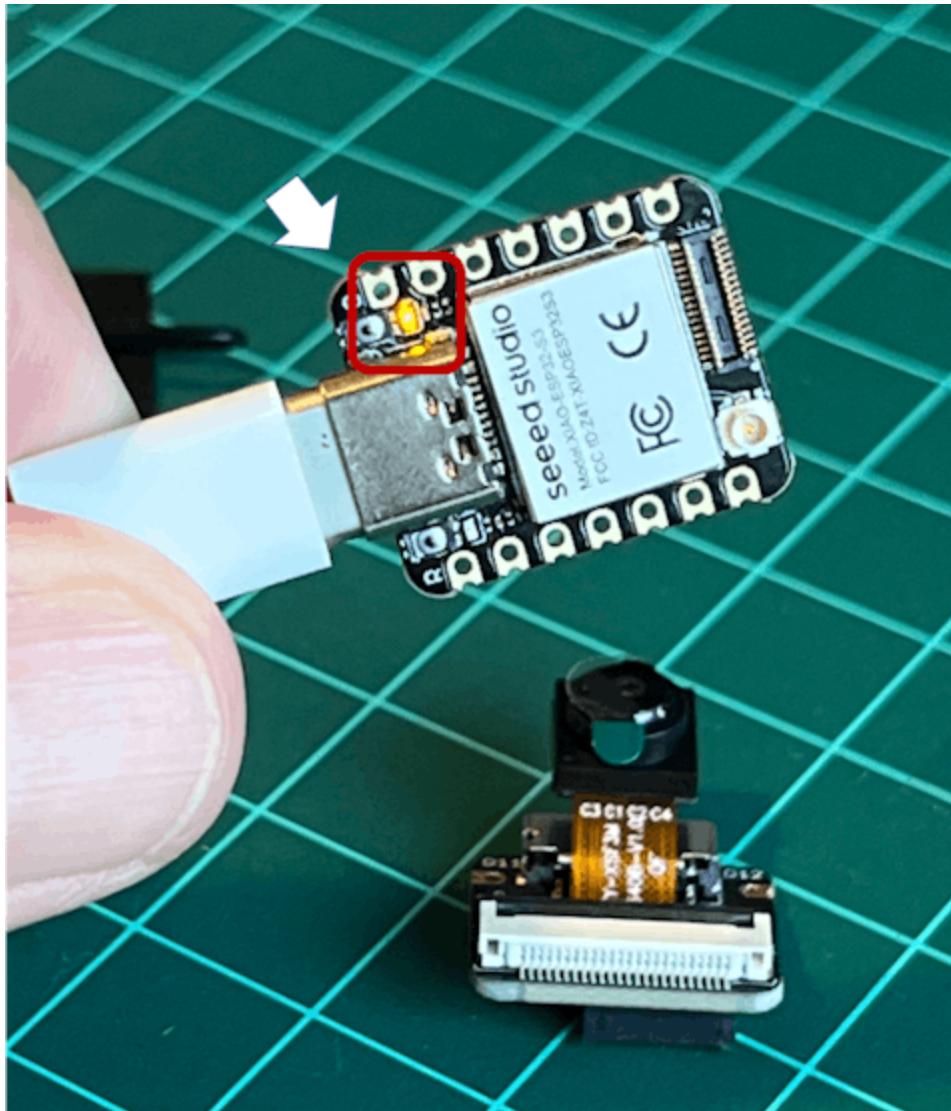
So, you can run the blink sketch as it (using the `LED_BUILTIN` Arduino constant) or by changing the Blink sketch accordantly:

```
#define LED_BUILTIN 21

void setup() {
    pinMode(LED_BUILTIN, OUTPUT); // Set the pin as output
}

// Remember that the pin work with inverted logic
// LOW to Turn on and HIGH to turn off
void loop() {
    digitalWrite(LED_BUILTIN, LOW); //Turn on
    delay (1000); //Wait 1 sec
    digitalWrite(LED_BUILTIN, HIGH); //Turn off
    delay (1000); //Wait 1 sec
}
```

Note that the pins work with inverted logic: LOW to Turn on and HIGH to turn off



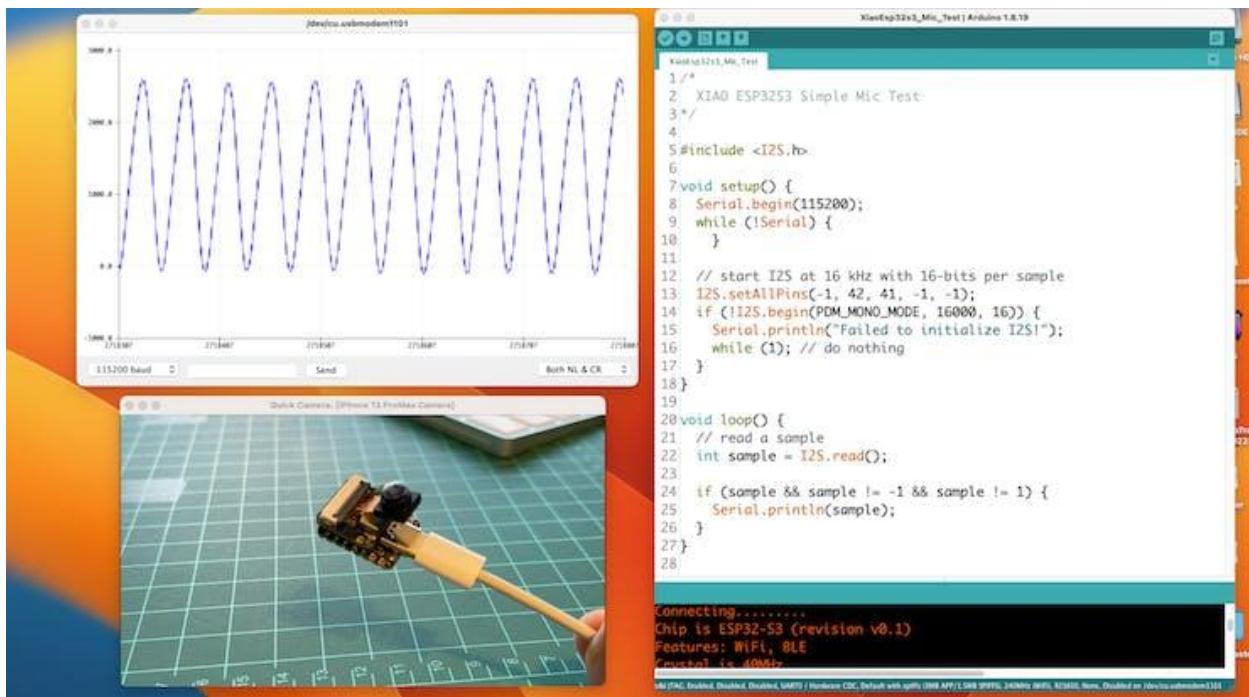
Connecting Sense module (Expansion Board)

When purchased, the expansion board is separated from the main board, but installing the expansion board is very simple. You need to align the connector on the expansion board with the B2B connector on the XIAO ESP32S3, press it hard, and when you hear a "click," the installation is complete.

As commented in the introduction, the expansion board, or the "sense" part of the device, has a 1600x1200 OV2640 camera, an SD card slot, and a digital microphone.

Microphone Test

Let's start with sound detection. Go to the [GitHub project](#) and download the sketch: [XIAOEsp2s3_Mic_Test](#) and run it on the Arduino IDE:



When producing sound, you can verify it on the Serial Plotter.

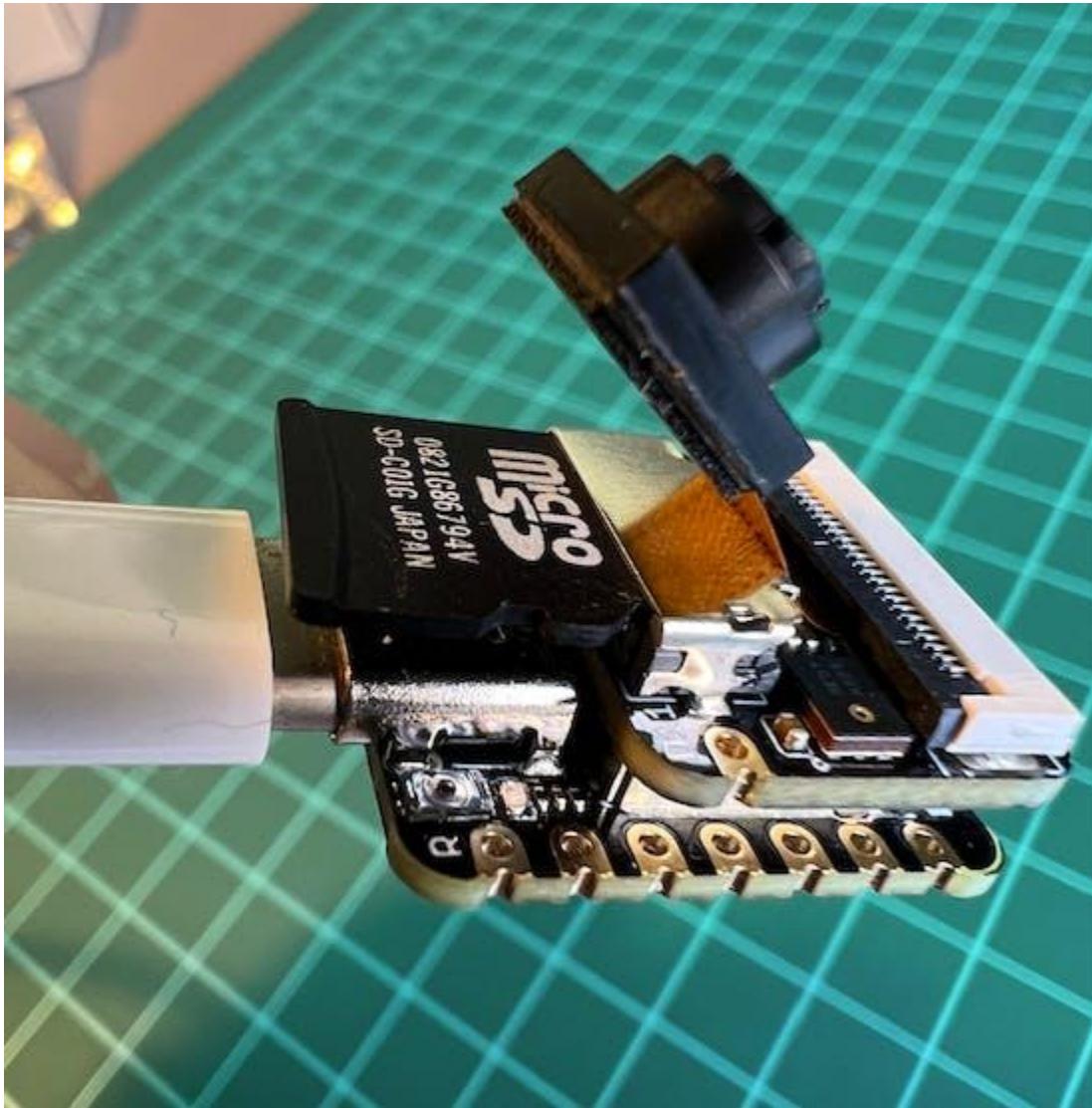
Save recorded sound (.wav audio files) to a microSD card.

Let's now use the onboard SD Card reader to save .wav audio files. For that, we need to habilitate the XIAO PSRAM.

ESP32-S3 has only a few hundred kilobytes of internal RAM on the MCU chip. It can be insufficient for some purposes, so ESP32-S3 can use up to 16 MB of

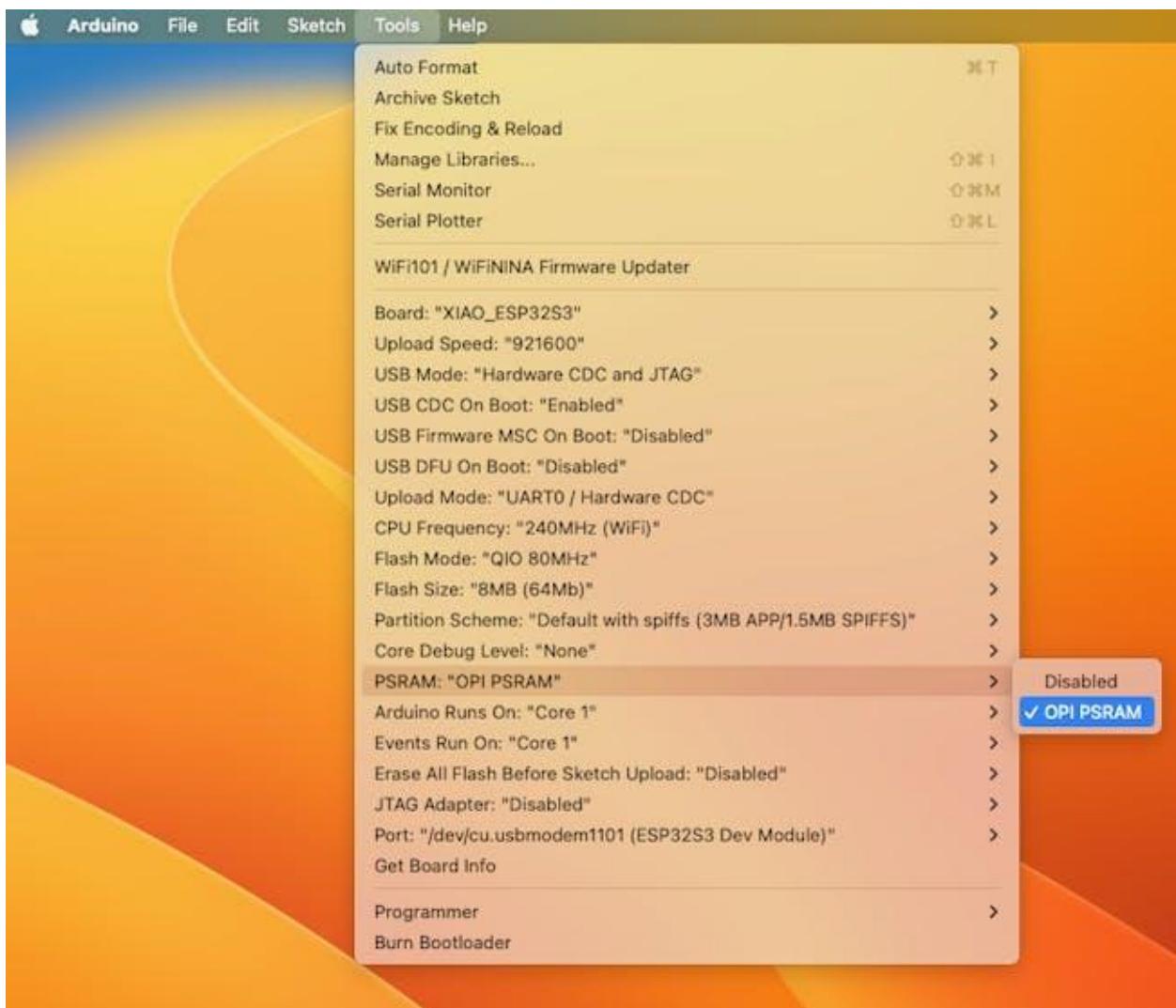
external PSRAM (Psuedostatic RAM) connected in parallel with the SPI flash chip. The external memory is incorporated in the memory map and, with certain restrictions, is usable in the same way as internal data RAM.

For a start, Insert the SD Card on the XIAO as shown in the photo below (the SD Card should be formatted to **FAT32**).



- Download the sketch [Wav_Record](#), which you can find on GitHub.

- To execute the code (Wav Record), it is necessary to use the PSRAM function of the ESP-32 chip, so turn it on before uploading.:
Tools>PSRAM: "OPI PSRAM">OPI PSRAM



- Run the code `Wav_Record.ino`
- This program is executed only once after the user turns on the serial monitor, recording for 20 seconds and saving the recording file to a microSD card as "arduino_rec.wav".

- When the "." is output every 1 second in the serial monitor, the program execution is finished, and you can play the recorded sound file with the help of a card reader.

```
.....Ready to start recording ...
Buffer: 667064 bytes
Record 640000 bytes
Writing to the file ...
The recording is over.
.....
```

Autoscroll Show timestamp Both NL & CR 115200 baud Clear output

The sound quality is excellent!

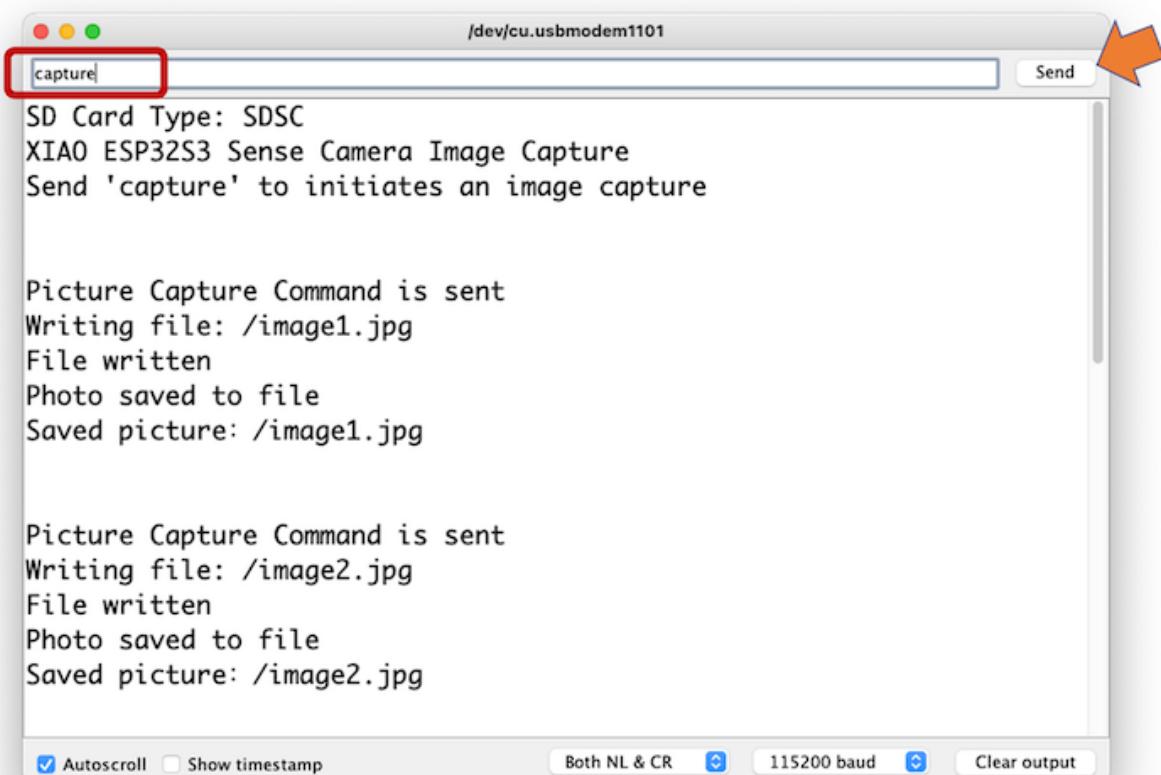
The explanation of how the code works is beyond the scope of this tutorial, but you can find an excellent description on the [wiki](#) page.

Testing the Camera

For testing the camera, you should download the folder [take_photos_command](#) from GitHub. The folder contains the sketch (`.ino`) and two `.h` files with camera details.

- Run the code: `take_photos_command.ino`. Open the Serial Monitor and send the command “`capture`” to capture and save the image on the SD Card:

Verify that [Both NL & CR] is selected on Serial Monitor.



```
/dev/cu.usbmodem1101
capture
Send
SD Card Type: SDSC
XIAO ESP32S3 Sense Camera Image Capture
Send 'capture' to initiates an image capture

Picture Capture Command is sent
Writing file: /image1.jpg
File written
Photo saved to file
Saved picture: /image1.jpg

Picture Capture Command is sent
Writing file: /image2.jpg
File written
Photo saved to file
Saved picture: /image2.jpg

Autoscroll  Show timestamp Both NL & CR 115200 baud Clear output
```

Here is an example of a taken photo:



Testing WiFi

One of the differentiators of the XIAO ESP32S3 is its WiFi capability. So, let's test its radio, scanning the wifi networks around it. You can do it by running one of the code examples on the board.

Go to Arduino IDE Examples and look for **WiFi ==> WiFiScan**

On the Serial monitor, you should see the wifi networks (SSIDs and RSSIs) in the range of your device. Here is what I got in my home:

The screenshot shows a Mac OS X-style terminal window titled "capture". The title bar also displays the path "/dev/cu.usbmodem1101". The window contains several lines of text output from a WiFi scan:

```
Setup done
Scan start
Scan done
1 networks found
Nr | SSID | RSSI | CH | Encryption
1 | ROVAI TIMECAP | -73 | 6 | WPA2

Scan start
Scan done
1 networks found
Nr | SSID | RSSI | CH | Encryption
1 | ROVAI TIMECAP | -73 | 6 | WPA2

Scan start
Scan done
```

At the bottom of the window, there are several configuration buttons: "Autoscroll" (unchecked), "Show timestamp" (unchecked), "Both NL & CR" (selected), "115200 baud" (selected), and "Clear output".

Simple WiFi Server (Turning LED ON/OFF)

Let's test the device's capability to behave as a WiFi Server. We will host a simple page on the device that sends commands to turn the XIAO built-in LED ON and OFF.

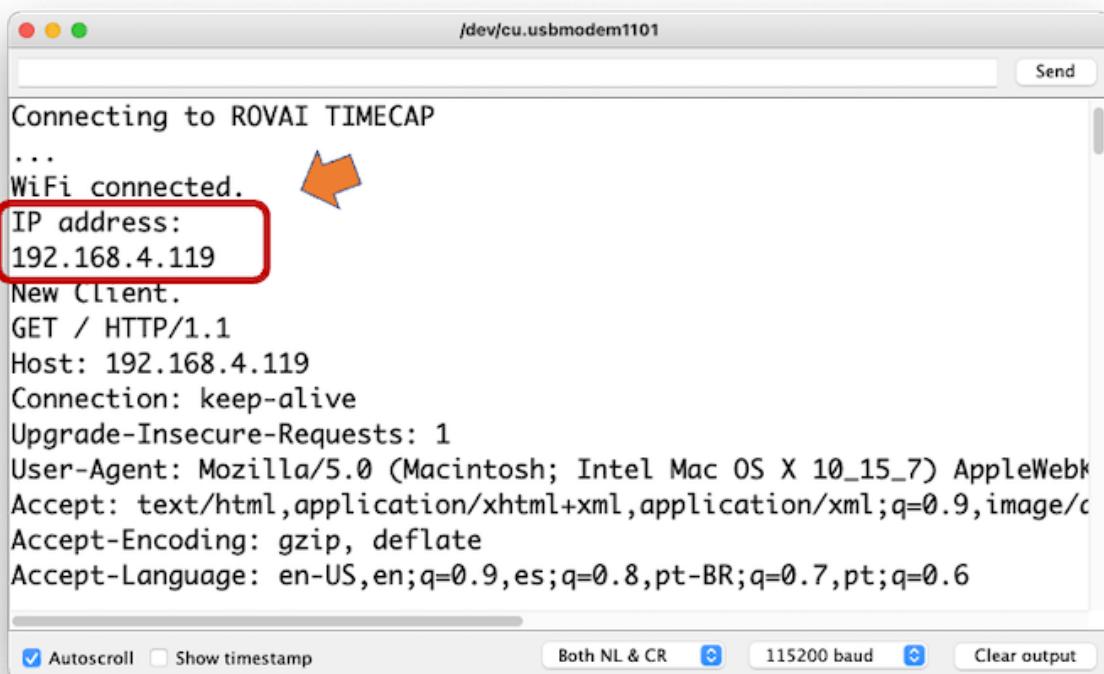
Like before, go to GitHub to download the folder with the sketch:

[SimpleWiFiServer](#).

Before running the sketch, you should enter your network credentials:

```
const char* ssid      = "Your credentials here";
const char* password = "Your credentials here";
```

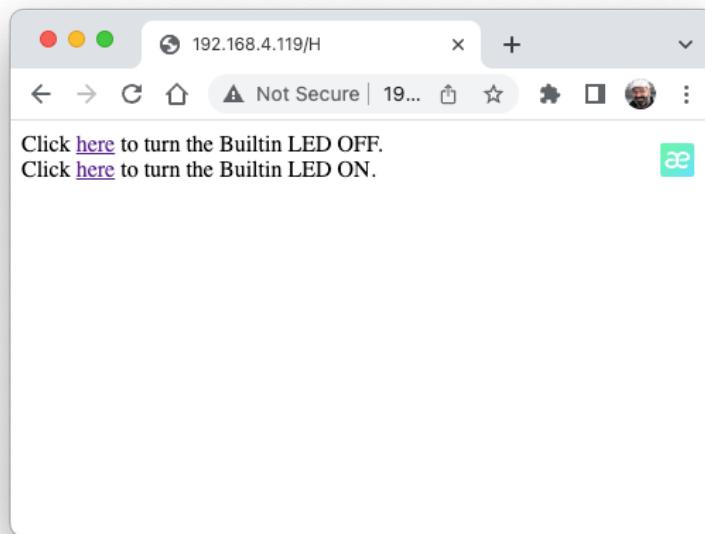
You can monitor how your server is working with the Serial Monitor.



```
Connecting to ROVAI TIMECAP
...
WiFi connected. ←
IP address:
192.168.4.119
New Client.
GET / HTTP/1.1
Host: 192.168.4.119
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/12.0.3 Safari/605.1.15
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9,es;q=0.8,pt-BR;q=0.7,pt;q=0.6
```

Autoscroll Show timestamp Both NL & CR 115200 baud Clear output

Take the IP address and enter it on your browser:



You will see a page with links that can turn ON and OFF the built-in LED of your XIAO.

Streaming video to Web

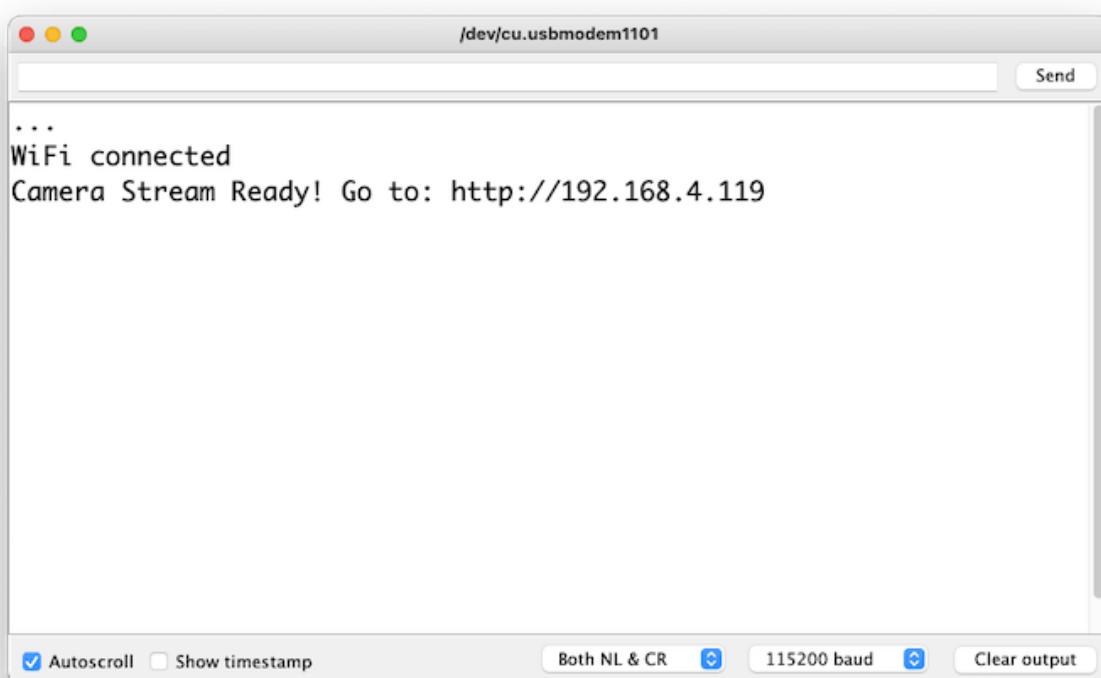
Now that you know that you can send commands from the webpage to your device, let's do the reverse. Let's take the image captured by the camera and stream it to a webpage:

Download from GitHub the [folder](#) that contains the code:

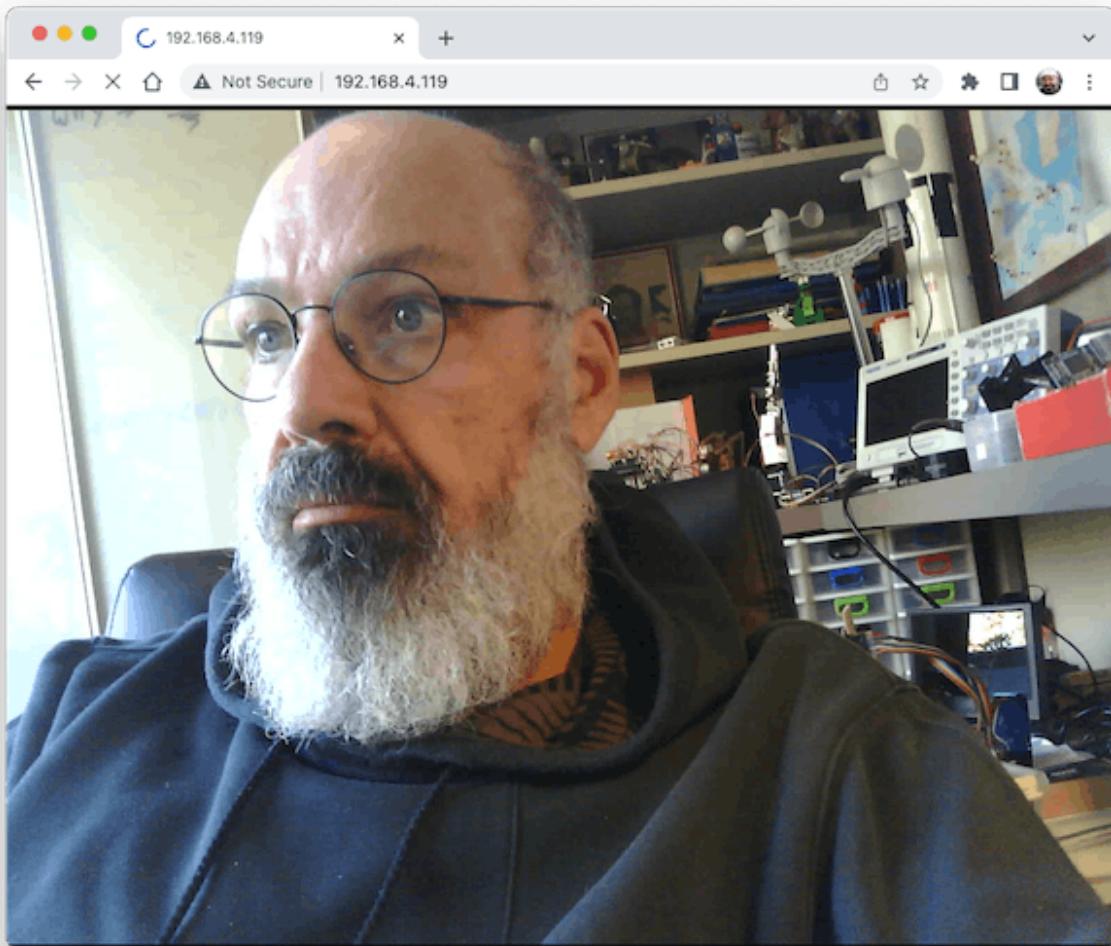
XIAO-ESP32S3-Streeming_Video.ino.

Remember that the folder contains not only the.ino file, but also a couple of.h files, necessary to handle the camera.

Enter your credentials and run the sketch. On the Serial monitor, you can find the page address to enter in your browser:



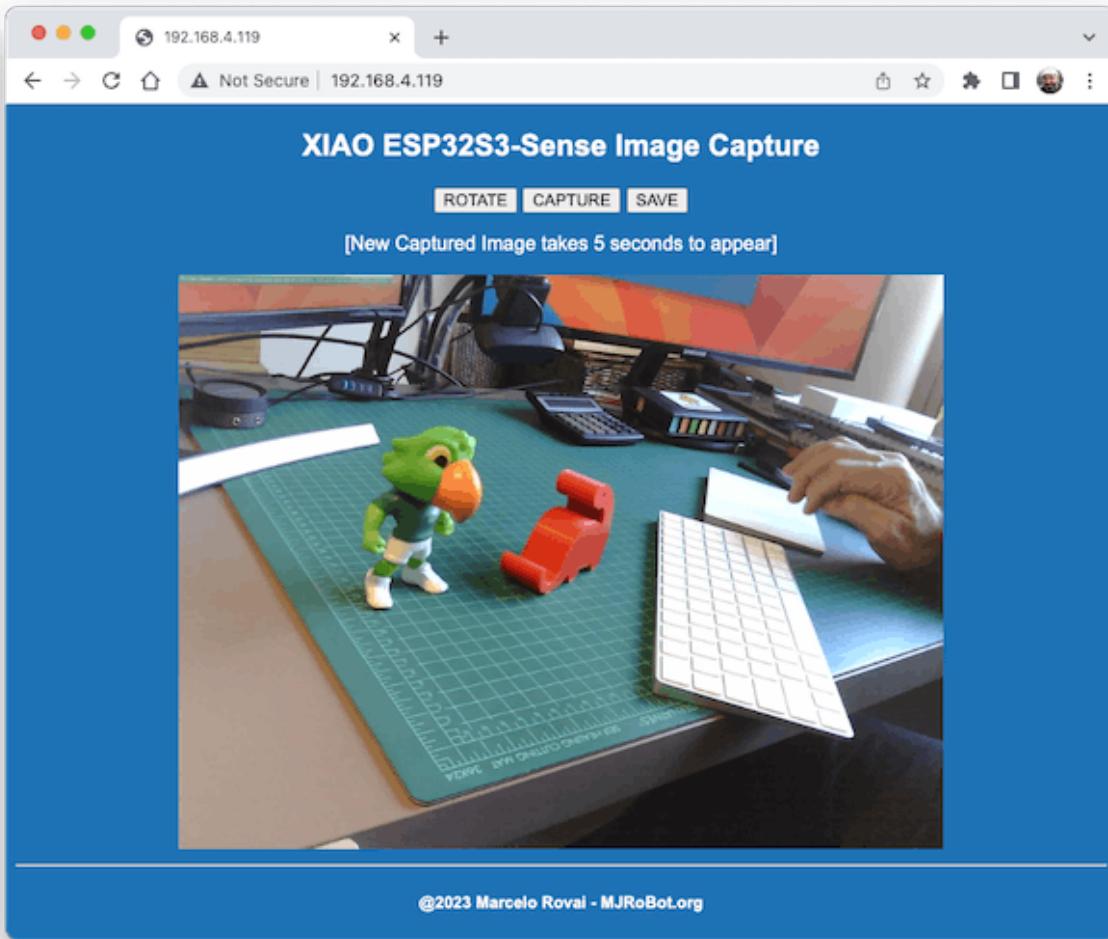
Open the page on your browser (wait a few seconds to start the streaming).
That's it.



Streamlining what your camera is "seen" can be important when you position it to capture a dataset for an ML project (for example, using the code "take_photos_commands.ino").

Of course, we can do both things simultaneously, show what the camera is seeing on the page, and send a command to capture and save the image on

the SD card. For that, you can use the code Camera_HTTP_Server_STA which [folder](#) can be downloaded from GitHub.



The program will do the following tasks:

- Set the camera to JPEG output mode.
- Create a web page (for example ==> <http://192.168.4.119/>). The correct address will be displayed on the Serial Monitor.
- If server.on ("/capture", HTTP_GET, serverCapture), the program takes a photo and sends it to the Web.

- It is possible to rotate the image on webPage using the button [ROTATE]
- The command [CAPTURE] only will preview the image on the webpage, showing its size on Serial Monitor
- The [SAVE] command will save an image on the SD Card, also showing the image on the web.
- Saved images will follow a sequential naming (image1.jpg, image2.jpg).

```

...
WiFi connected..!
Got IP: 192.168.4.119
HTTP server started
Capturing Image for view only
The picture has a size of 143360 bytes
Saving Image to SD Card
Photo saved to file
Saved picture: /image1.jpg

Saving Image to SD Card
Photo saved to file
Saved picture: /image2.jpg

```

This program can be used for an image dataset capture with an Image Classification project.

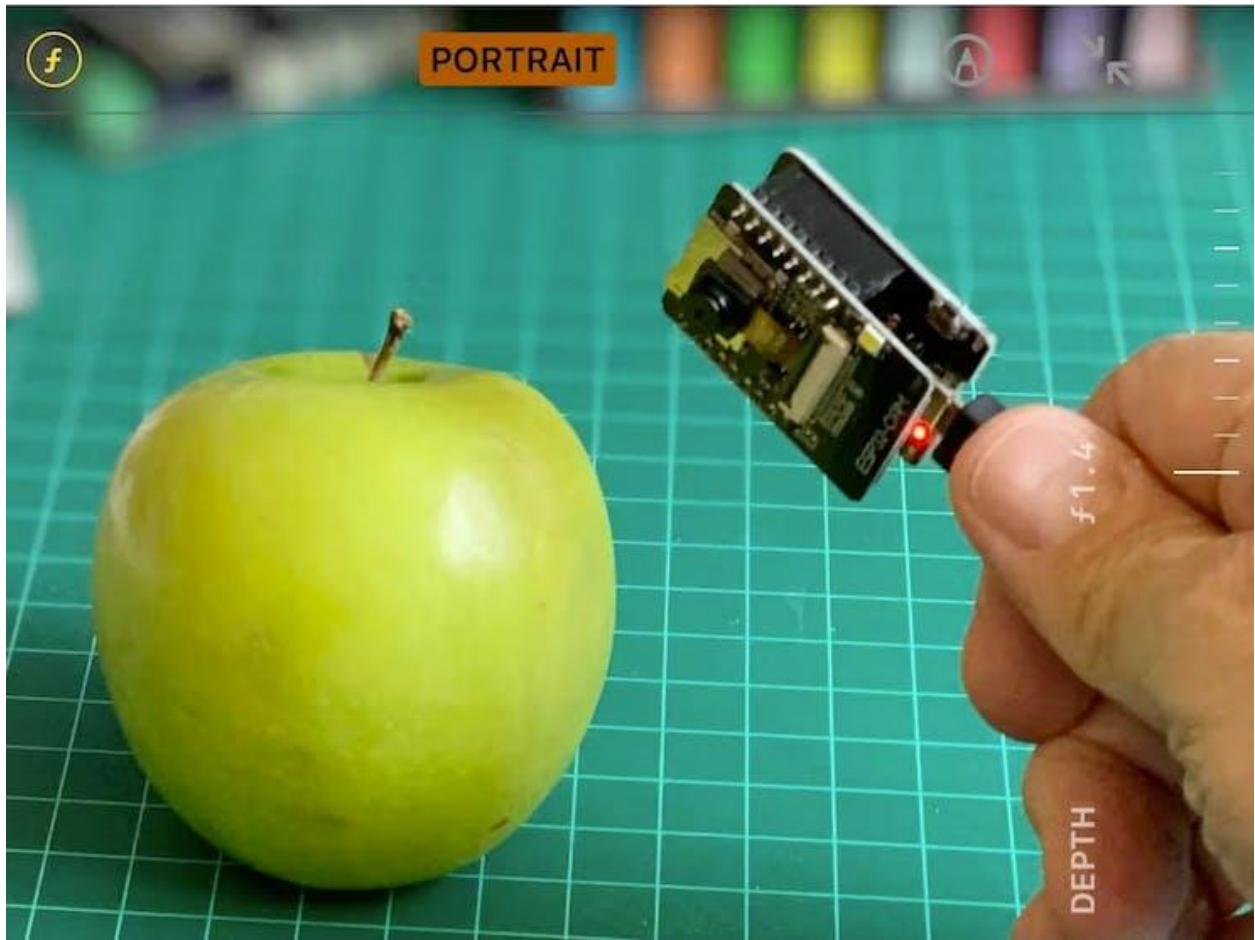
Inspect the code; it will be easier to understand how the camera works..This code was developed based on the great Rui Santos Tutorial: [ESP32-CAM Take Photo and Display in Web Server](#), which I invite all of you to visit.

Fruits versus Veggies - A TinyML Image Classification Project



Now that we have an embedded camera running, it is time to try image classification. For comparative motive, I will replicate the same image classification project developed to be used with an old ESP2-CAM.:

[ESP32-CAM: TinyML Image Classification - Fruits vs Veggies](#)



The whole idea of our project will be training a model and proceeding with inference on the XIAO ESP32S3 Sense. For training, we should find some data (**in fact, tons of data!**).

But first of all, we need a goal! What do we want to classify?

With TinyML, a set of technics associated with machine learning inference on embedded devices, we should limit the classification to three or four categories due to limitations (mainly memory in this situation). We will differentiate **apples** from **bananas** and **potatoes** (you can try other categories).

So, let's find a specific dataset that includes images from those categories.

Kaggle is a good start:

<https://www.kaggle.com/kritikseth/fruit-and-vegetable-image-recognition>

This dataset contains images of the following food items:

- **Fruits** - *banana, apple, pear, grapes, orange, kiwi, watermelon, pomegranate, pineapple, mango.*
- **Vegetables** - *cucumber, carrot, capsicum, onion, potato, lemon, tomato, radish, beetroot, cabbage, lettuce, spinach, soybean, cauliflower, bell pepper, chili pepper, turnip, corn, sweetcorn, sweet potato, paprika, jalepeño, ginger, garlic, peas, eggplant.*

Each category is split into the **train** (100 images), **test** (10 images), and **validation** (10 images).

- Download the dataset from the Kaggle website to your computer.

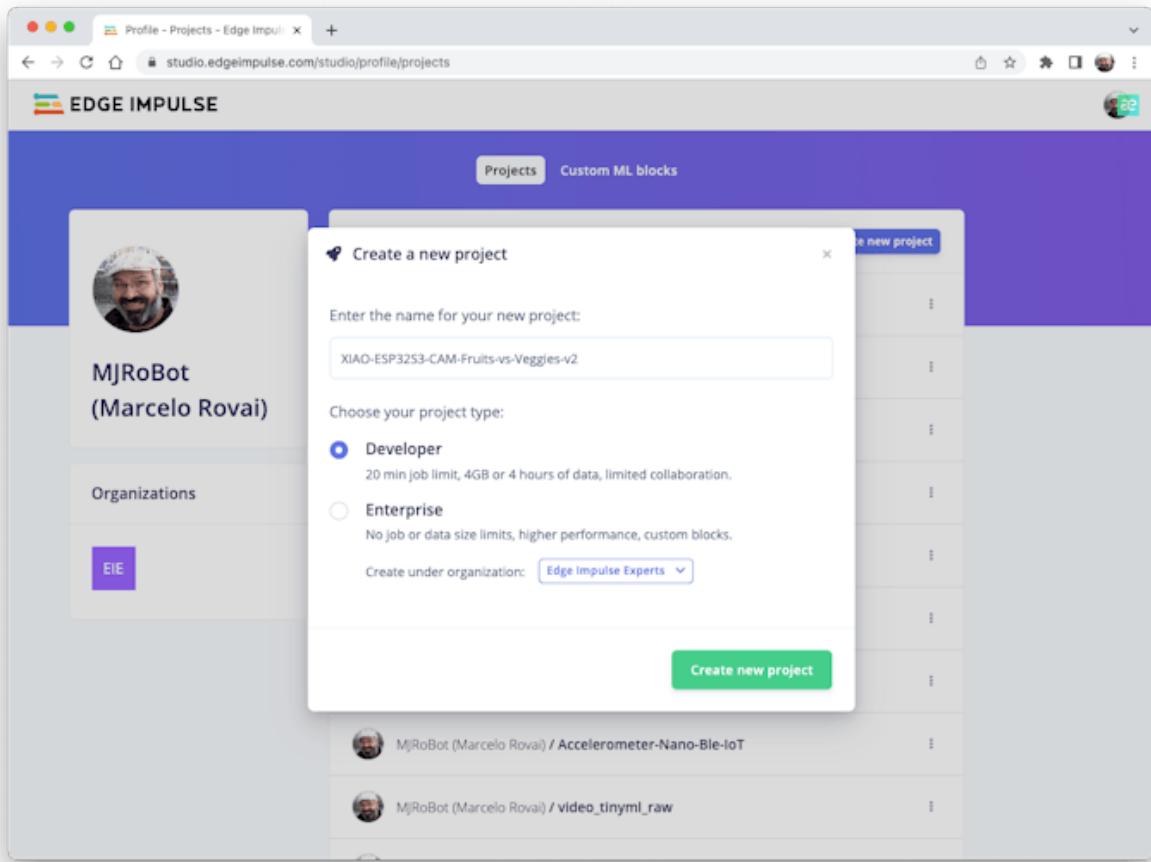
Optionally, you can add some fresh photos of bananas, apples, and potatoes from your home kitchen, using, for example, the sketch discussed in the last section.

Training the model with Edge Impulse Studio

We will use the Edge Impulse Studio for training our model. [Edge Impulse](#) is a leading development platform for machine learning on edge devices.

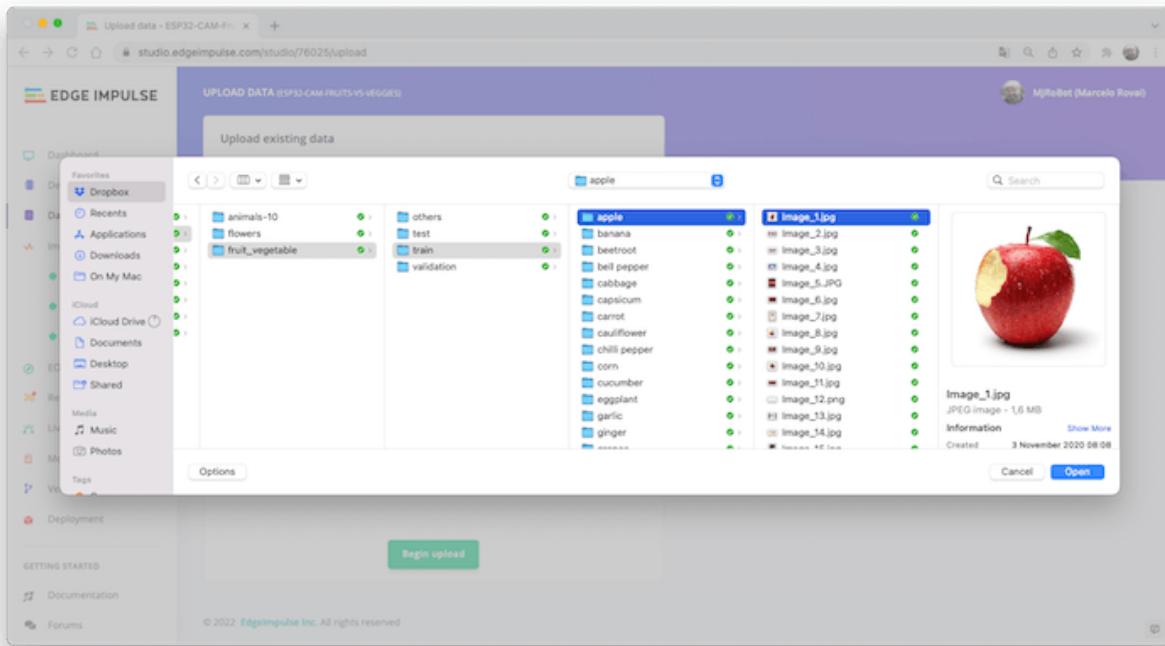
Enter your account credentials (or create a free account) at Edge Impulse.

Next, create a new project:



Data Acquisition

Next, on the **UPLOAD DATA** section, upload from your computer the files from chosen categories:



You should now have your training dataset split into three classes of data:

The screenshot shows the Edge Impulse Data acquisition interface. On the left, a sidebar lists various project management and development tools. The main area displays a summary of collected data: 279 items, with a 100% / 0% train/test split. Below this is a table of 'Collected data' entries, each with a sample name, label (banana), date added, and length. A preview window on the right shows an image labeled 'Image_89.jpg.2p4ed0vt' which is a banana. At the bottom, there are navigation buttons for the data table.

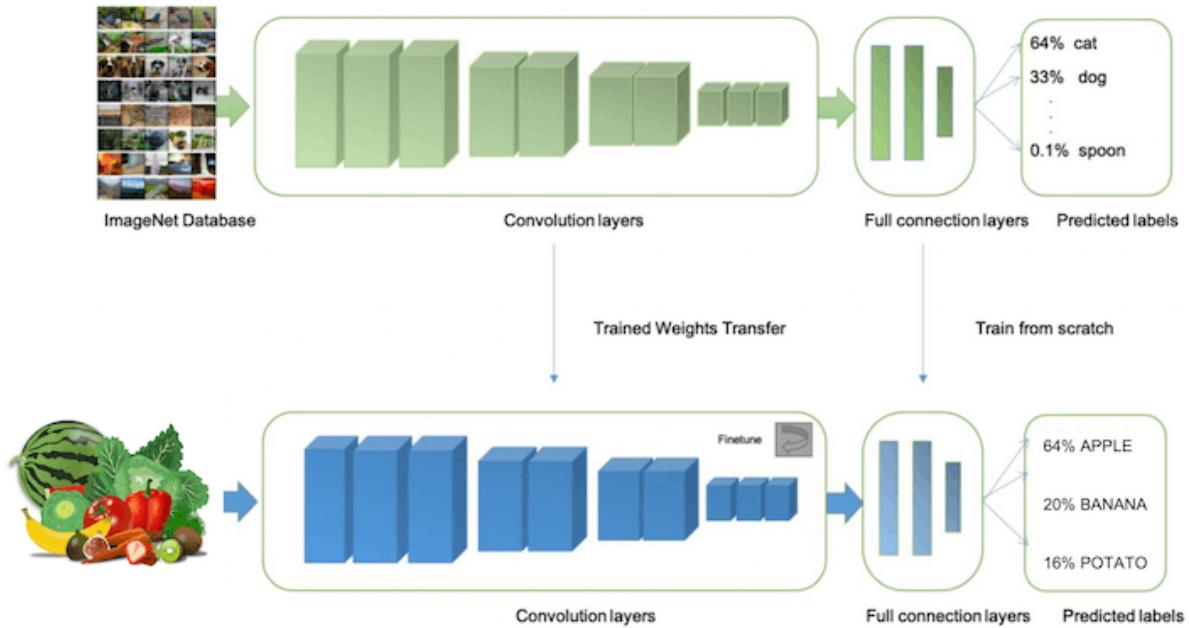
SAMPLE NAME	LABEL	ADDED	LENGTH
Image_89.jpg.2p4ed0vt	banana	jan 12 2022, 15:45...	-
Image_91.jpg.2p4ed1po	banana	jan 12 2022, 15:45...	-
Image_88.png.2p4ecv5e	banana	jan 12 2022, 15:45...	-
Image_92.jpg.2p4ecus5	banana	jan 12 2022, 15:45...	-
Image_100.jpg.2p4ectpi	banana	jan 12 2022, 15:45...	-
Image_90.jpg.2p4ecscnc	banana	jan 12 2022, 15:45...	-
Image_87.jpg.2p4ecraf6	banana	jan 12 2022, 15:45...	-
Image_83.png.2p4ecr56	banana	jan 12 2022, 15:45...	-
Image_84.jpg.2p4ecrf6e	banana	jan 12 2022, 15:45...	-
Image_79.jpg.2p4ecrqq	banana	jan 12 2022, 15:45...	-
Image_86.jpg.2p4ecqsh	banana	jan 12 2022, 15:45...	-
Image_85.jpg.2p4ecqcsc	banana	jan 12 2022, 15:45...	-

You can upload extra data for further model testing or split the training data. I will leave it as it is to use the most data possible.

Impulse Design

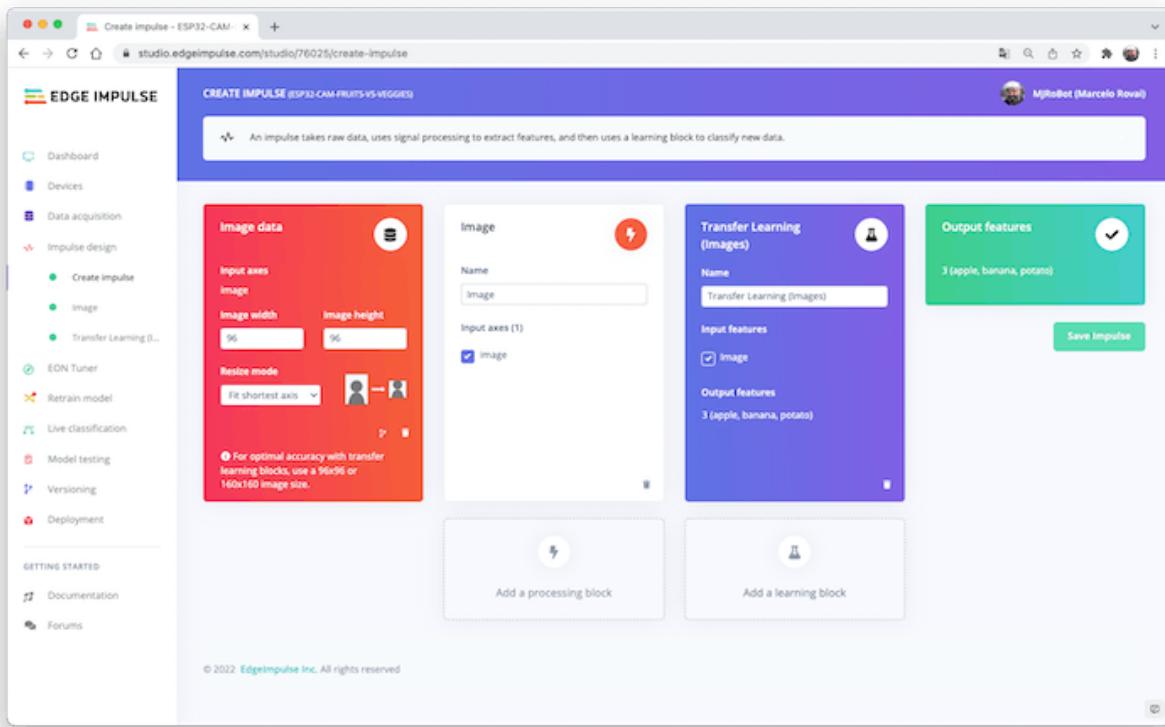
An impulse takes raw data (in this case, images), extracts features (resize pictures), and then use a learning block to classify new data.

As mentioned, classifying images is the most common use of Deep Learning, but much data should be used to accomplish this task. We have around 90 images for each category. Is this number enough? Not at all! We will need thousand of images to "teach or model" to differentiate an apple from a banana. But, we can solve this issue by re-training a previously trained model with thousands of images. We called this technic "Transfer Learning" (TL).



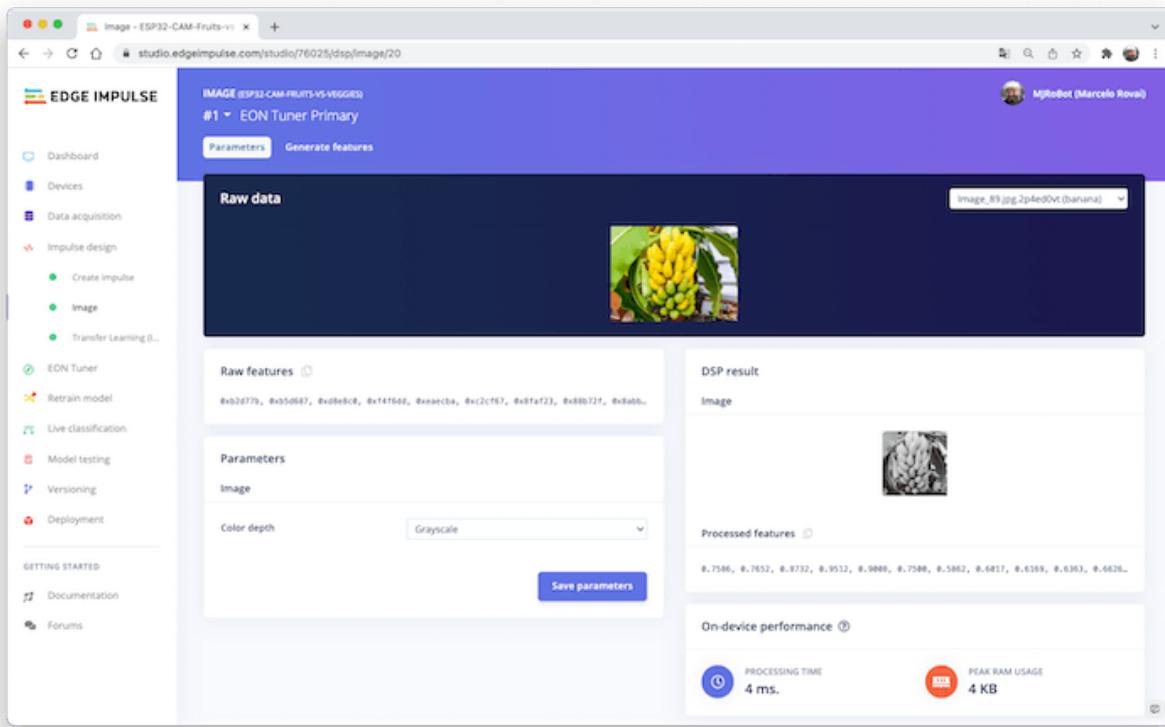
With TL, we can fine-tune a pre-trained image classification model on our data, performing well even with relatively small image datasets (our case).

So, starting from the raw images, we will resize them (96x96) pixels and so, feeding them to our Transfer Learning block:



Pre-processing (Feature generation)

Besides resizing the images, we should change them to Grayscale instead to keep the actual RGB color depth. Doing that, each one of our data samples will have dimension 9, 216 features (96x96x1). Keeping RGB, this dimension would be three times bigger. Working with Grayscale helps to reduce the amount of final memory needed for inference.



Do not forget to "Save parameters." This will generate the features to be used in training.

Training (Transfer Learning & Data Augmentation)

In 2007, Google introduced [MobileNetV1](#), a family of general-purpose computer vision neural networks designed with mobile devices in mind to support classification, detection, and more. MobileNets are small, low-latency, low-power models parameterized to meet the resource constraints of various use cases.

Although the base MobileNet architecture is already tiny and has low latency, many times, a specific use case or application may require the model to be smaller and faster. MobileNet introduces a straightforward parameter α (alpha) called width multiplier to construct these smaller and less computationally

expensive models. The role of the width multiplier α is to thin a network uniformly at each layer.

Edge Impulse Studio has available MobileNet V1 (96x96 images) and V2 (96x96 and 160x160 images), with several different α values (from 0.05 to 1.0). For example, you will get the highest accuracy with V2, 160x160 images, and $\alpha=1.0$. Of course, there is a trade-off. The highest the accuracy, the more memory (around 1.3M RAM and 2.6M ROM) will be needed to run the model and imply more latency.

The smaller footprint will be obtained at another extreme with **MobileNet V1** and $\alpha=0.10$ (around 53.2K RAM and 101K ROM).

When we first published this project to be running on an ESP32-CAM, we stayed at the lower side of possibilities which guaranteed the inference with small latency but not with high accuracy. For this first pass, we will keep this model design (**MobileNet V1** and $\alpha=0.10$).

Another important technic to be used with Deep Learning is **Data Augmentation**. Data augmentation is a method that can help improve the accuracy of machine learning models, creating additional artificial data. A data augmentation system makes small, random changes to your training data during the training process (such as flipping, cropping, or rotating the images).

Under the rood, you can see how Edge Impulse implements a data Augmentation policy on your data:

```
# Implements the data augmentation policy
def augment_image(image, label):
    # Flips the image randomly
    image = tf.image.random_flip_left_right(image)

    # Increase the image size, then randomly crop it down to
    # the original dimensions
```

```

resize_factor = random.uniform(1, 1.2)
new_height = math.floor(resize_factor * INPUT_SHAPE[0])
new_width = math.floor(resize_factor * INPUT_SHAPE[1])
image = tf.image.resize_with_crop_or_pad(image, new_height, new_width)
image = tf.image.random_crop(image, size=INPUT_SHAPE)

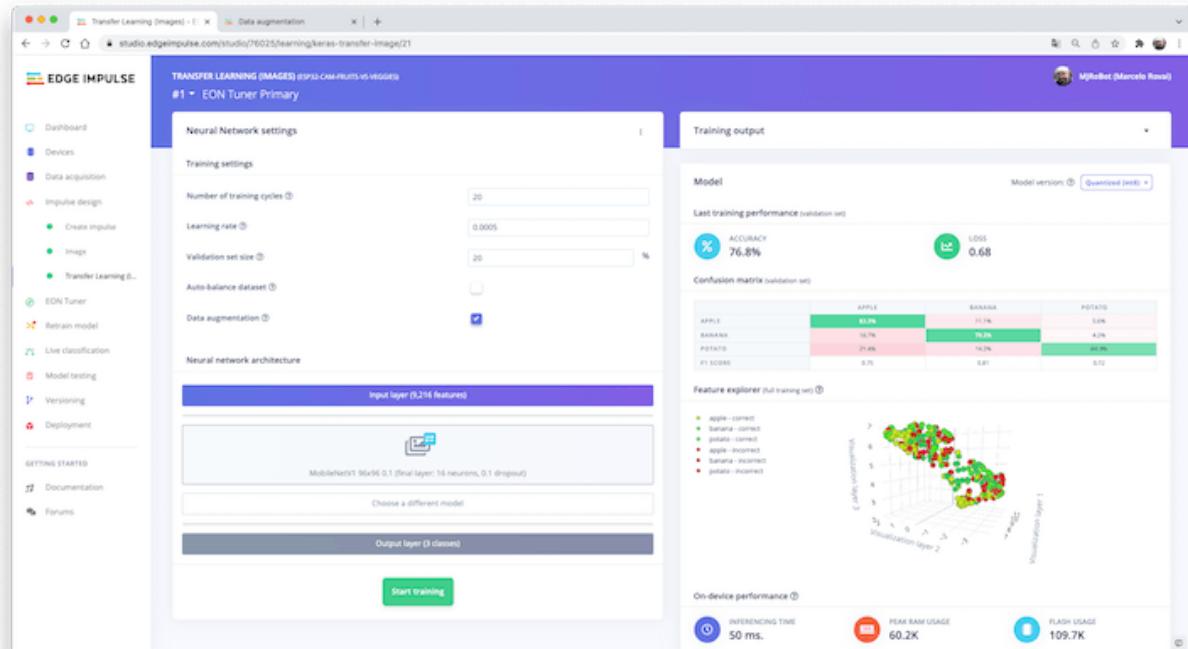
# Vary the brightness of the image
image = tf.image.random_brightness(image, max_delta=0.2)

return image, label

```

Exposure to these variations during training can help prevent your model from taking shortcuts by "memorizing" superficial clues in your training data, meaning it may better reflect the deep underlying patterns in your dataset.

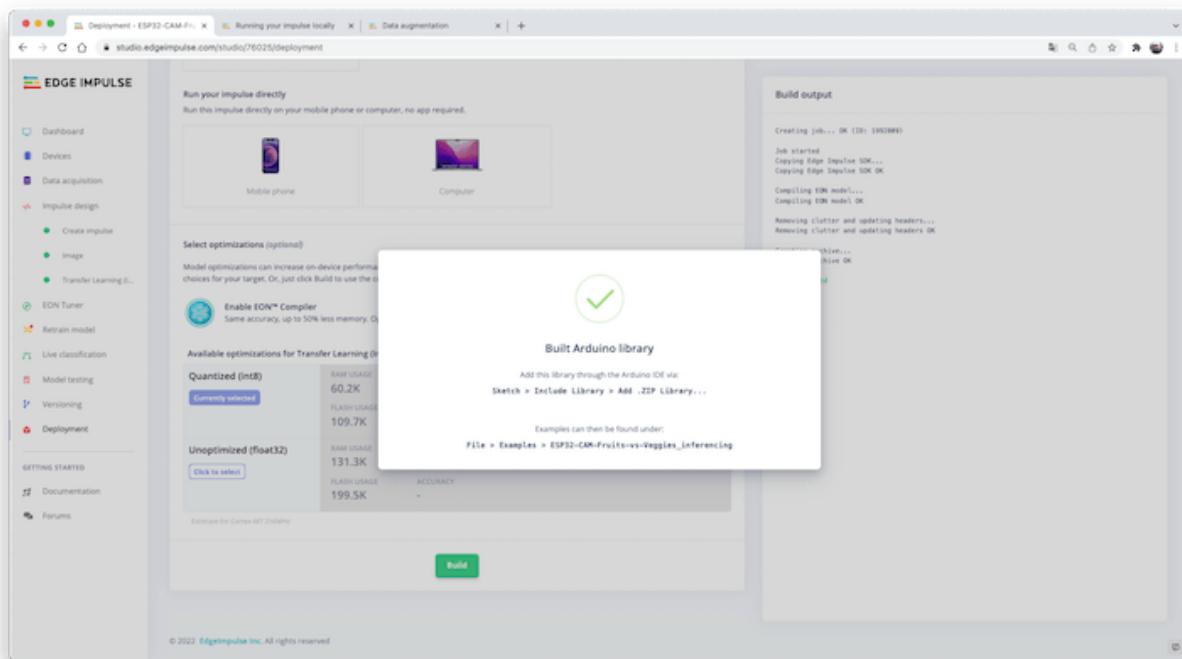
The final layer of our model will have 16 neurons with a 10% of dropout for overfitting prevention. Here is the Training output:



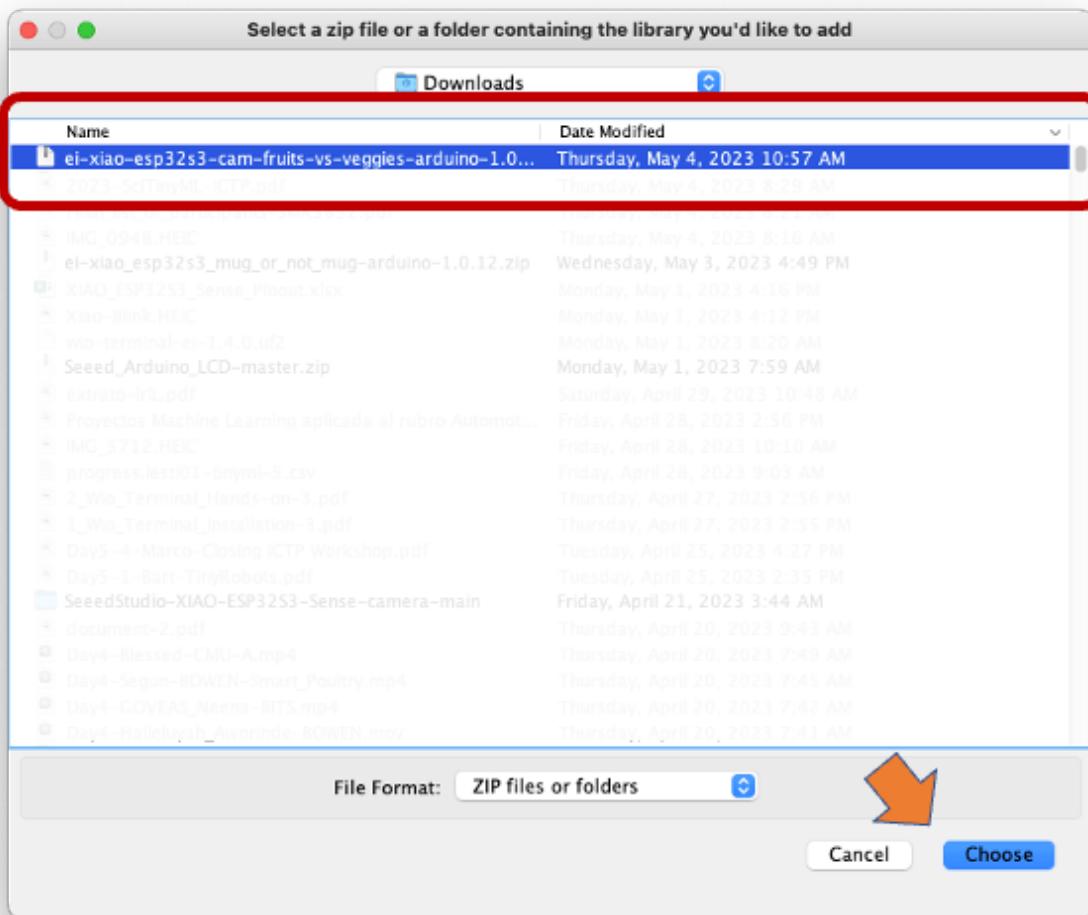
The result is not great. The model reached around 77% of accuracy, but the amount of RAM expected to be used during the inference is relatively small (around 60 KBytes), which is very good.

Deployment

The trained model will be deployed as a.zip Arduino library:



Open your Arduino IDE, and under **Sketch**, go to **Include Library** and **add.ZIP Library**. Select the file you download from Edge Impulse Studio, and that's it!



Under the **Examples** tab on Arduino IDE, you should find a sketch code under your project name.



Open the Static Buffer example:

The screenshot shows the Arduino IDE interface with a window titled "static_buffer | Arduino 1.8.19". The code editor contains C++ code for a library. The code includes comments explaining the purpose of the function and its parameters. The parameters are described as follows:

- offset: The offset
- length: The length
- out_ptr: The out pointer

The code also includes a line of code that imports a header file: "#include <XIAO-ESP32S3-CAM-Fruits-vs-Veggies_inferencing.h>". At the bottom of the screen, there is a status bar with the text: "Mb), Core 1, Core 1, Hardware CDC and JTAG, Enabled, Disabled, UART0 / Hardware CDC, Default with spiffs (3MB APP / 1.5MB SPIFFS), 240MHz (WiFi), 921600, None, Disabled on /dev/cu.usbmodem1101".

```
static_buffer | Arduino 1.8.19
static_buffer
15 */
16
17/* Includes -----
18#include <XIAO-ESP32S3-CAM-Fruits-vs-Veggies_inferencing.h>
19
20static const float features[] = {
21    // copy raw features here (for example from the 'Live classification' page)
22    // see https://docs.edgeimpulse.com/docs/running-your-impulse-arduino
23};
24
25/**
26 * @brief      Copy raw feature data in out_ptr
27 *             Function called by inference library
28 *
29 * @param[in]  offset    The offset
30 * @param[in]  length   The length
31 * @param     out_ptr   The out pointer
32 *
33 * @return     0
-
Mb), Core 1, Core 1, Hardware CDC and JTAG, Enabled, Disabled, UART0 / Hardware CDC, Default with spiffs (3MB APP / 1.5MB SPIFFS), 240MHz (WiFi), 921600, None, Disabled on /dev/cu.usbmodem1101
```

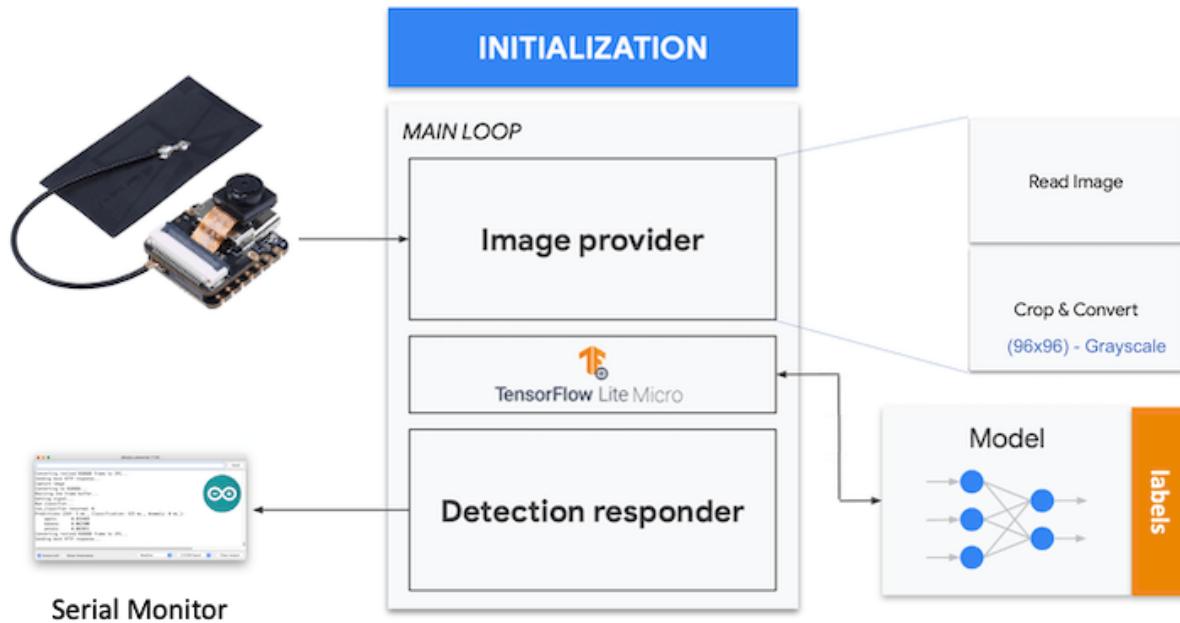
You can see that the first line of code is exactly the calling of a library with all the necessary stuff for running inference on your device.

```
#include <XIAO-ESP32S3-CAM-Fruits-vs-Veggies_inferencing.h>
```

Of course, this is a generic code (a "template"), that only gets one sample of raw data (stored on the variable: *features* = {} and run the classifier, doing the inference. The result is shown on Serial Monitor.

We should get the sample (image) from the camera and pre-process it (resizing to 96x96, converting to grayscale, and flattening it). This will be the

input tensor of our model. The output tensor will be a vector with three values (labels), showing the probabilities of each one of the classes.



Returning to your project (Tab Image), copy one of the Raw Data Sample:

The screenshot shows the Edge Impulse Studio interface. On the left, there's a sidebar with various project management and development tools. The main workspace is titled '#1 EON Tuner Primary'. It displays a 'Raw data' section with a banana image, a 'Parameters' section where 'Color depth' is set to 'Grayscale', and a 'DSP result' section showing a processed version of the banana. A tooltip 'Copy 9216 features to clipboard' is visible over the feature extraction area.

9,216 features will be copied to the clipboard. This is the input tensor (a flattened image of 96x96x1), in this case, bananas. Paste this Input tensor on

```
features[] = { 0xb2d77b, 0xb5d687, 0xd8e8c0, 0xeaecba, 0xc2cf67,  
... }
```

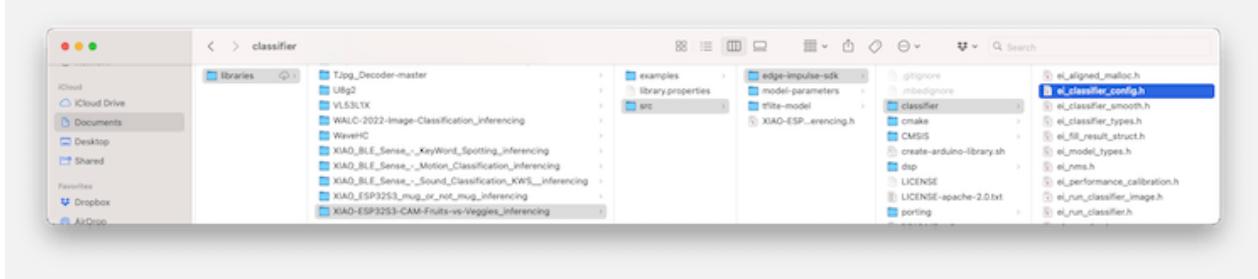
The screenshot shows the Arduino IDE interface with a file named "static_buffer" open. The code includes a section for includes, a static const float array named "features", and a note about memory constraints. The status bar at the bottom indicates the board is set to "Core 1, Core 2, Hardware CDC and JTAG, Enabled, Disabled, UART0 / Hardware CDC, Default with spiffs (3MB APP/1.5MB SPIFFS), 240MHz (WiFi), 921600, None, Disabled on /dev/cu.usbmodem1101".

```
static_buffer | Arduino 1.8.19
static_buffer §
16
17 /* Includes -----
18 #include <XIAO-ESP32S3-CAM-Fruits-vs-Veggies_inferencing.h>
19
20 static const float features[] = {
21     0xb2d77b, 0xb5d687, 0xd8e8c0, 0xf4f6dd, 0xeaecba, 0xc2cf67, 0x8faf23, 0x88b72f
22 };
23
24 /**
 * @brief   ... Camera Feature detection
 */
MB, Core 1, Core 2, Hardware CDC and JTAG, Enabled, Disabled, UART0 / Hardware CDC, Default with spiffs (3MB APP/1.5MB SPIFFS), 240MHz (WiFi), 921600, None, Disabled on /dev/cu.usbmodem1101
```

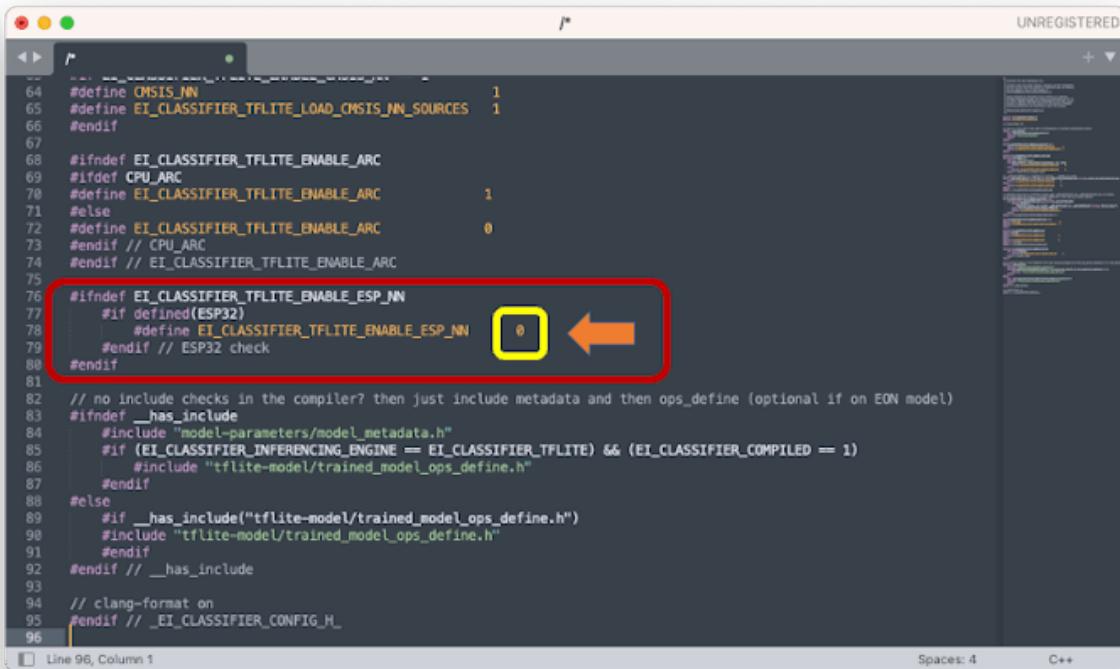
NOTE: Edge Impulse included the [library ESP NN](#) in its SDK, which contains optimized NN (Neural Network) functions for various Espressif chips. Until June 2023, the ESP NN was not working with the ESP32S3 (Arduino IDE).

If you compile the code and get an error, it will be necessary to fix this. EI recommends switching off ESP NN acceleration. To do that, locate `ei_classifier_config.h` in exported Arduino library folder:

```
/scr/edge-impulse-sdk/classifier/:
```

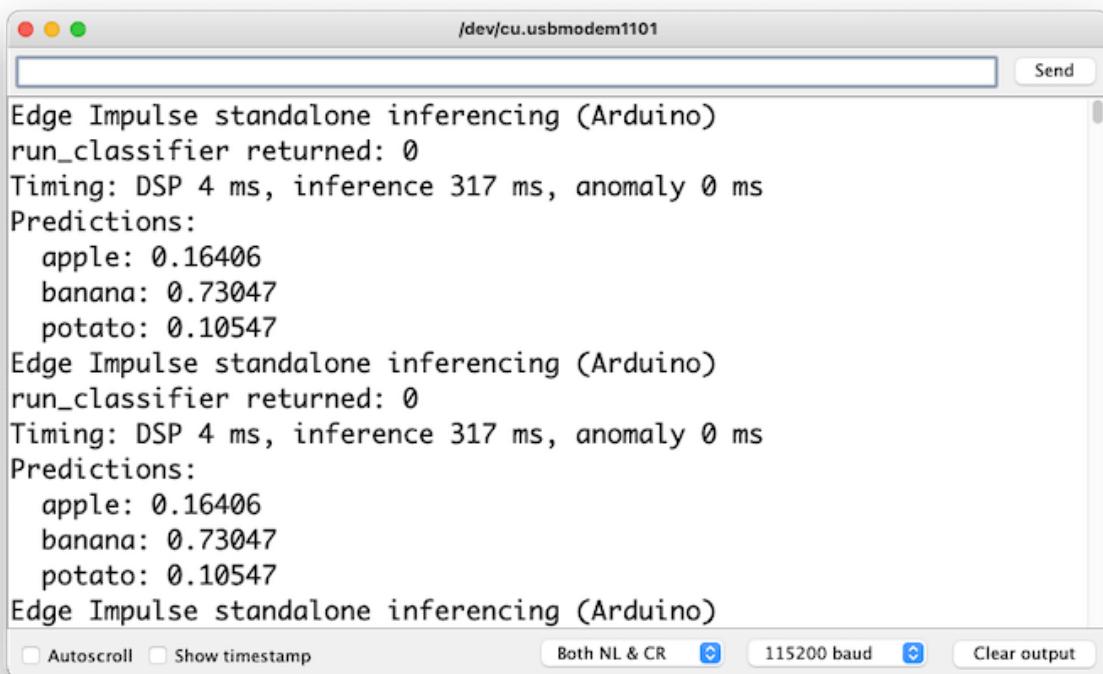


Locate the line with `#define EI_CLASSIFIER_TFLITE_ENABLE_ESP_NN 1`, and change it from 1 to 0:



```
/*  
64 #define CMSIS_NN 1  
65 #define EI_CLASSIFIER_TFLITE_LOAD_CMSIS_NN_SOURCES 1  
66 #endif  
67  
68 #ifndef EI_CLASSIFIER_TFLITE_ENABLE_ARC  
69 #ifdef CPU_ARC  
70 #define EI_CLASSIFIER_TFLITE_ENABLE_ARC 1  
71 #else  
72 #define EI_CLASSIFIER_TFLITE_ENABLE_ARC 0  
73 #endif // CPU_ARC  
74 #endif // EI_CLASSIFIER_TFLITE_ENABLE_ARC  
75  
76 #ifndef EI_CLASSIFIER_TFLITE_ENABLE_ESP_NN  
77 #if defined(ESP32)  
78 #define EI_CLASSIFIER_TFLITE_ENABLE_ESP_NN 0  
79 #endif // ESP32 check  
80 #endif  
81  
82 // no include checks in the compiler? then just include metadata and then ops_define (optional if on EON model)  
83 #ifndef __has_include  
84 #include "model-parameters/model_metadata.h"  
85 #if (EI_CLASSIFIER_INFERENCE_ENGINE == EI_CLASSIFIER_TFLITE) && (EI_CLASSIFIER_COMPILED == 1)  
86 #include "tflite-model/trained_model_ops_define.h"  
87 #endif  
88 #else  
89 #if __has_include("tflite-model/trained_model_ops_define.h")  
90 #include "tflite-model/trained_model_ops_define.h"  
91 #endif  
92 #endif // __has_include  
93  
94 // clang-format on  
95 #endif // _EI_CLASSIFIER_CONFIG_H_
```

Now, when running the inference, you should get; as a result, the highest score for "banana".



The screenshot shows a terminal window titled '/dev/cu.usbmodem1101'. The window displays three sets of inference results from an Edge Impulse model running on an Arduino. Each set includes timing information, prediction counts, and confidence scores for three categories: apple, banana, and potato.

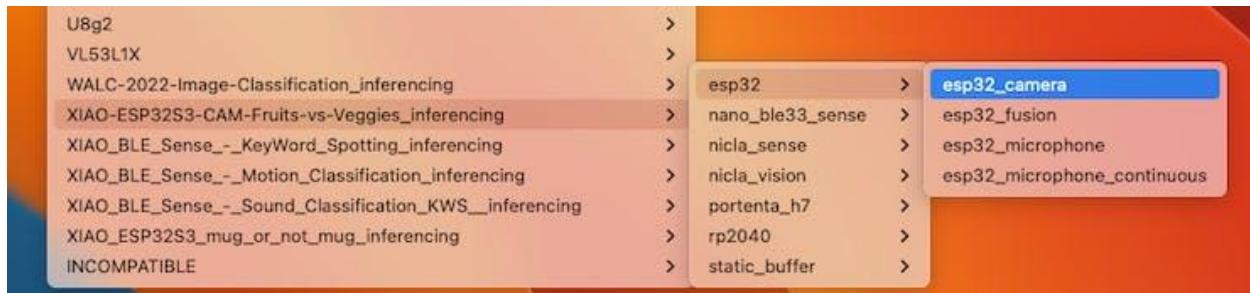
```
Edge Impulse standalone inferencing (Arduino)
run_classifier returned: 0
Timing: DSP 4 ms, inference 317 ms, anomaly 0 ms
Predictions:
  apple: 0.16406
  banana: 0.73047
  potato: 0.10547
Edge Impulse standalone inferencing (Arduino)
run_classifier returned: 0
Timing: DSP 4 ms, inference 317 ms, anomaly 0 ms
Predictions:
  apple: 0.16406
  banana: 0.73047
  potato: 0.10547
Edge Impulse standalone inferencing (Arduino)
```

At the bottom of the terminal window, there are several configuration options: 'Autoscroll' (unchecked), 'Show timestamp' (unchecked), 'Both NL & CR' (selected), '115200 baud' (selected), and 'Clear output'.

Great news! Our device handles an inference, discovering that the input image is a banana. Also, note that the inference time was around 317ms, resulting in a maximum of 3 fps if you tried to classify images from a video. It is a better result than the ESP32 CAM (525ms of latency).

Now, we should incorporate the camera and classify images in real-time.

Go to the Arduino IDE Examples and download from your project the sketch esp32_camera:



You should change lines 32 to 75, which define the camera model and pins, by the data related to our model:

The screenshot shows the Arduino IDE with the esp32_camera example open. The code defines a camera model and pin numbers. A red box highlights the section from line 32 to line 75, which contains the #define statements for the camera model and pin numbers. The code is as follows:

```
esp32_camera.h
23 /* Includes
24 #include <XIAO-ESP32S3-CAM-Fruits-vs-Veggies_inferencing.h>
25 #include "edge-impulse-sdk/dsp/image/image.hpp"
26
27 #include "esp_camera.h"
28
29 // Select camera model - find more camera models in camera_pins.h file here
30 // https://github.com/espressif/arduino-esp32/blob/master/libraries/ESP32/examples/Camera/
31
32 #define CAMERA_MODEL_XIAO_ESP32S3 // Has PSRAM
33
34 #define PWDN_GPIO_NUM      -1
35 #define RESET_GPIO_NUM    -1
36 #define XCLK_GPIO_NUM     10
37 #define SIOD_GPIO_NUM     40
38 #define SIOC_GPIO_NUM     39
39
40 #define Y9_GPIO_NUM        48
41 #define Y8_GPIO_NUM        11
42 #define Y7_GPIO_NUM        12
43 #define Y6_GPIO_NUM        14
44 #define Y5_GPIO_NUM        16
45 #define Y4_GPIO_NUM        18
46 #define Y3_GPIO_NUM        17
47 #define Y2_GPIO_NUM        15
48 #define VSYNC_GPIO_NUM     38
49 #define HREF_GPIO_NUM      47
50 #define PCLK_GPIO_NUM      13
51
52 #define LED_GPIO_NUM       21
```

The modified sketch can be downloaded from GitHub: [xiao_esp32s3_camera](#).

Note that you can optionally keep the pins as an a.h file as we did in previous sections.

Upload the code to your XIAO ESP32S3 Sense, and you should be OK to start classifying your fruits and vegetables! You can check the result on Serial Monitor.

Testing the Model (Inference)



Getting a photo with the camera, the classification result will appear on the Serial Monitor:

```
banana: 0.90234
potato: 0.03906
Predictions (DSP: 4 ms., Classification: 318 ms., Anomaly: 0 ms.):
apple: 0.03906
banana: 0.93359
potato: 0.02734
Predictions (DSP: 4 ms., Classification: 317 ms., Anomaly: 0 ms.):
apple: 0.05469
banana: 0.90625
potato: 0.03906
Predictions (DSP: 4 ms., Classification: 318 ms., Anomaly: 0 ms.):
apple: 0.04297
banana: 0.92578
potato: 0.03125
```

Autoscroll Show timestamp Both NL & CR 115200 baud Clear output

Other tests:

```
banana: 0.14844
potato: 0.12891
Predictions (DSP: 4 ms., Classification: 318 ms., Anomaly: 0 ms.):
apple: 0.78906
banana: 0.06641
potato: 0.14453
Predictions (DSP: 4 ms., Classification: 317 ms., Anomaly: 0 ms.):
apple: 0.71484
banana: 0.06641
potato: 0.21875
Predictions (DSP: 4 ms., Classification: 318 ms., Anomaly: 0 ms.):
apple: 0.79297
banana: 0.05469
potato: 0.14844
```

Autoscroll Show timestamp Both NL & CR 115200 baud Clear output



```

banana: 0.03125
potato: 0.79688
Predictions (DSP: 4 ms., Classification: 318 ms., Anomaly: 0 ms.):
apple: 0.32812
banana: 0.03906
potato: 0.63281
Predictions (DSP: 4 ms., Classification: 318 ms., Anomaly: 0 ms.):
apple: 0.40625
banana: 0.05469
potato: 0.53906
Predictions (DSP: 4 ms., Classification: 318 ms., Anomaly: 0 ms.):
apple: 0.16406
banana: 0.02344
potato: 0.81250 ←

```

Autoscroll Show timestamp Both NL & CR 115200 baud Clear output



```

banana: 0.73047
potato: 0.03125
Predictions (DSP: 4 ms., Classification: 317 ms., Anomaly: 0 ms.):
apple: 0.24219
banana: 0.71484
potato: 0.03906
Predictions (DSP: 4 ms., Classification: 318 ms., Anomaly: 0 ms.):
apple: 0.23828
banana: 0.72266
potato: 0.03906
Predictions (DSP: 4 ms., Classification: 317 ms., Anomaly: 0 ms.):
apple: 0.24609
banana: 0.72266 ←
potato: 0.03125

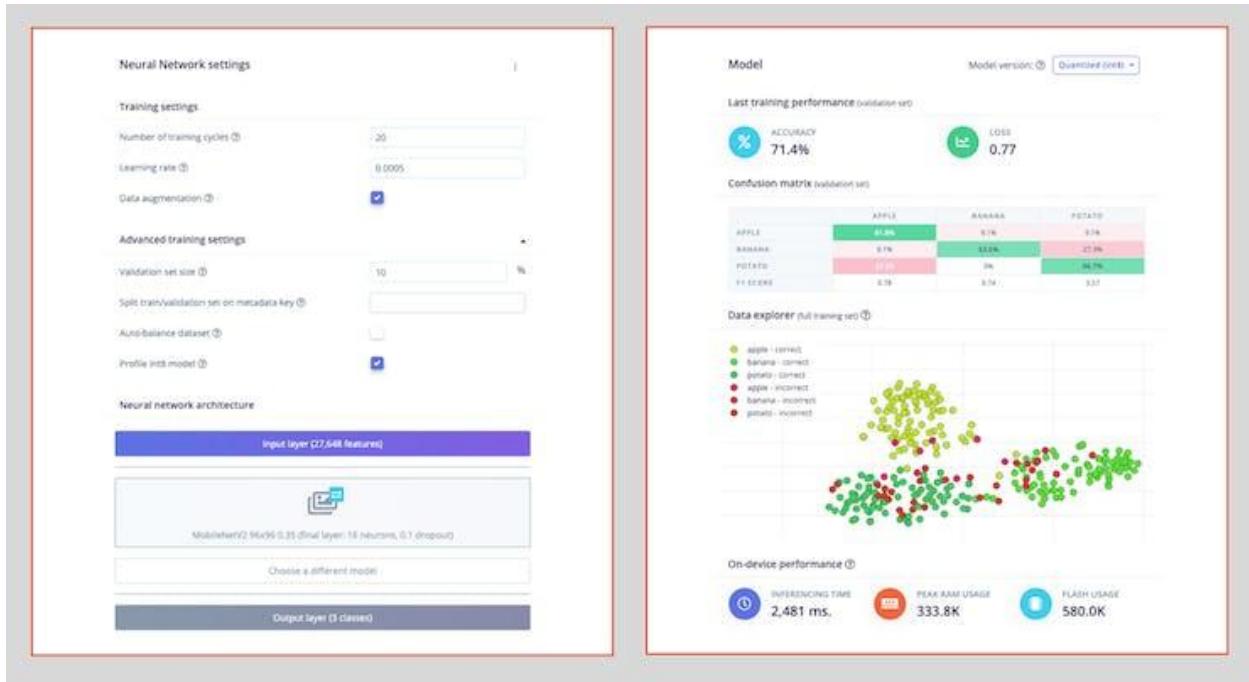
```

Autoscroll Show timestamp Both NL & CR 115200 baud Clear output



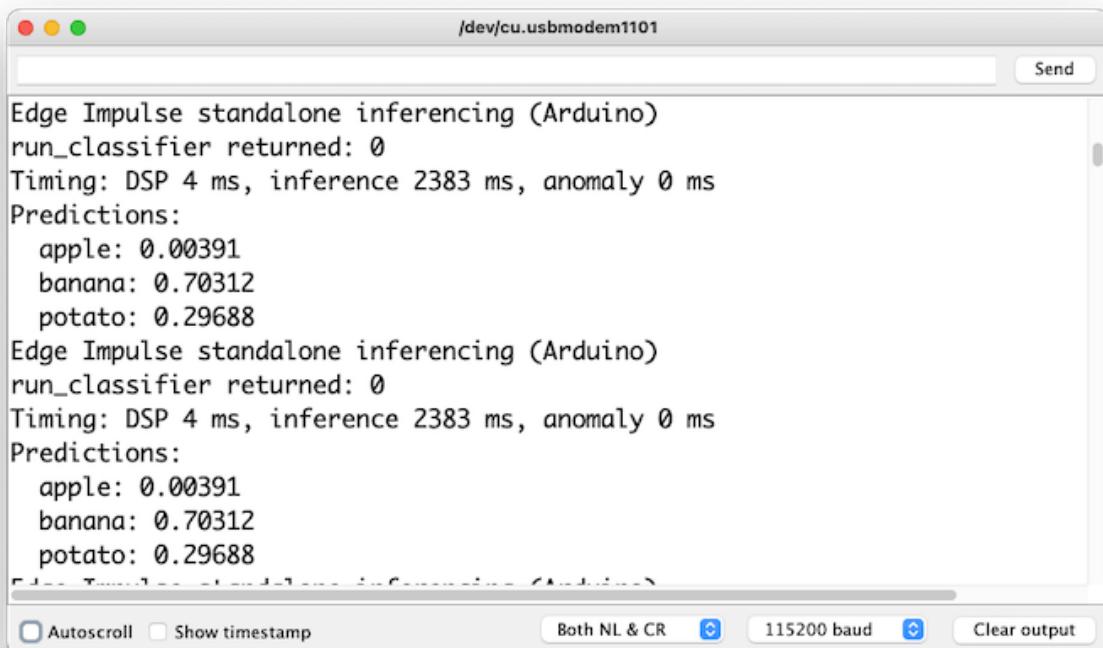
Testing with a bigger model

Now, let's go to the other side of the model size. Let's select a MobilinetV2 96x96 0.35, having as input RGB images.



Even with a bigger model, the accuracy is not good, and worst, the amount of memory necessary to run the model increases five times, with latency increasing seven times. So, to make our model better, we will probably need more images to be trained.

Even though our model did not improve, let's test whether the XIAO can handle such a bigger model. We will do a simple inference test with the Static Buffer sketch.



The screenshot shows a terminal window titled '/dev/cu.usbmodem1101'. The output displays two sets of inference results from an Arduino-based setup. Each set includes the classifier's run time, the total timing (DSP and inference), and the predicted classes with their confidence scores. The predictions are identical for both runs, with 'banana' having the highest probability (~0.70312).

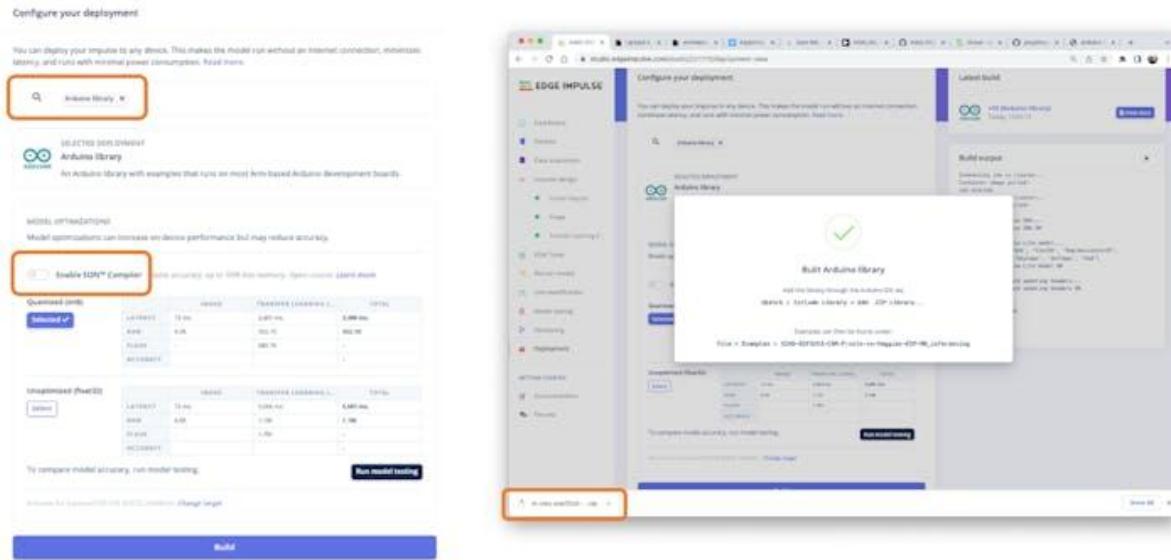
```
Edge Impulse standalone inferencing (Arduino)
run_classifier returned: 0
Timing: DSP 4 ms, inference 2383 ms, anomaly 0 ms
Predictions:
apple: 0.00391
banana: 0.70312
potato: 0.29688
Edge Impulse standalone inferencing (Arduino)
run_classifier returned: 0
Timing: DSP 4 ms, inference 2383 ms, anomaly 0 ms
Predictions:
apple: 0.00391
banana: 0.70312
potato: 0.29688
```

The result is YES! Memory is not an issue here; latency is! See that with a real test, the XIAO took almost 2.5s to perform the inference (compared with the previous 318ms).

Optional use of ESP-NN acceleration

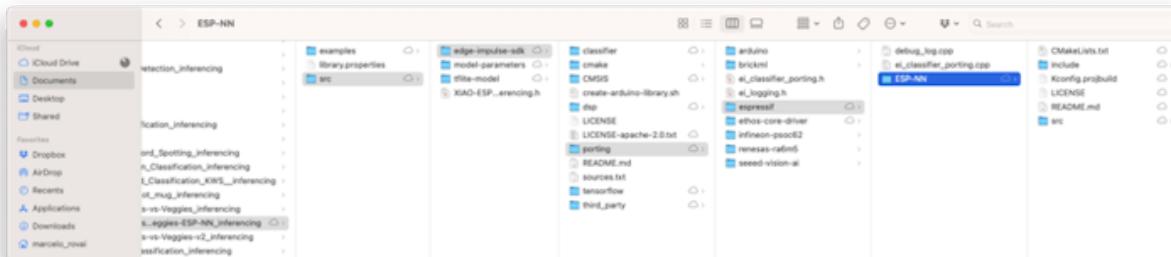
Even though Edge Impulse has not released its SDK for ESP32S3 using the accelerator, thanks to [Dmitry Maslov](#), we can have ESP NN with assembly optimizations restored and fixed for ESP32-S3. This solution is not official yet, being that EI will include it in EI SDK once they fix conflicts with other boards.

For now, this only works with the non-EON version. So, you should redeploy the model if the EON Compiler was enabled when you generate the library.



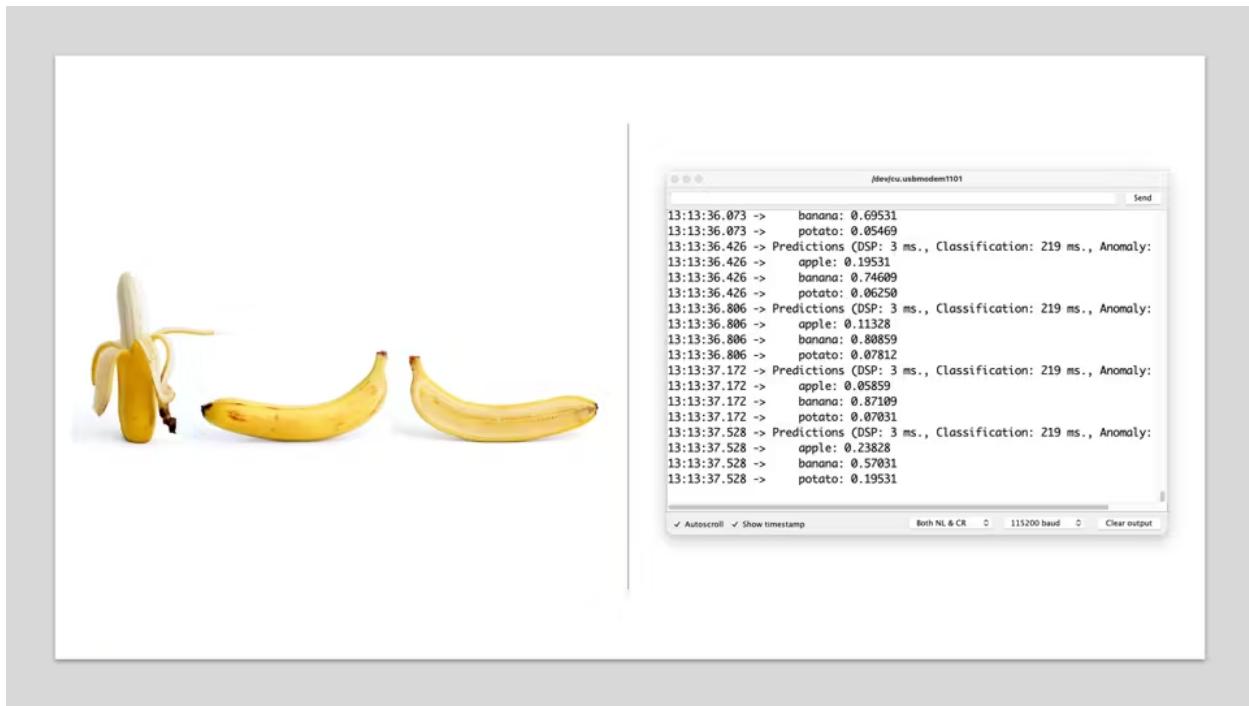
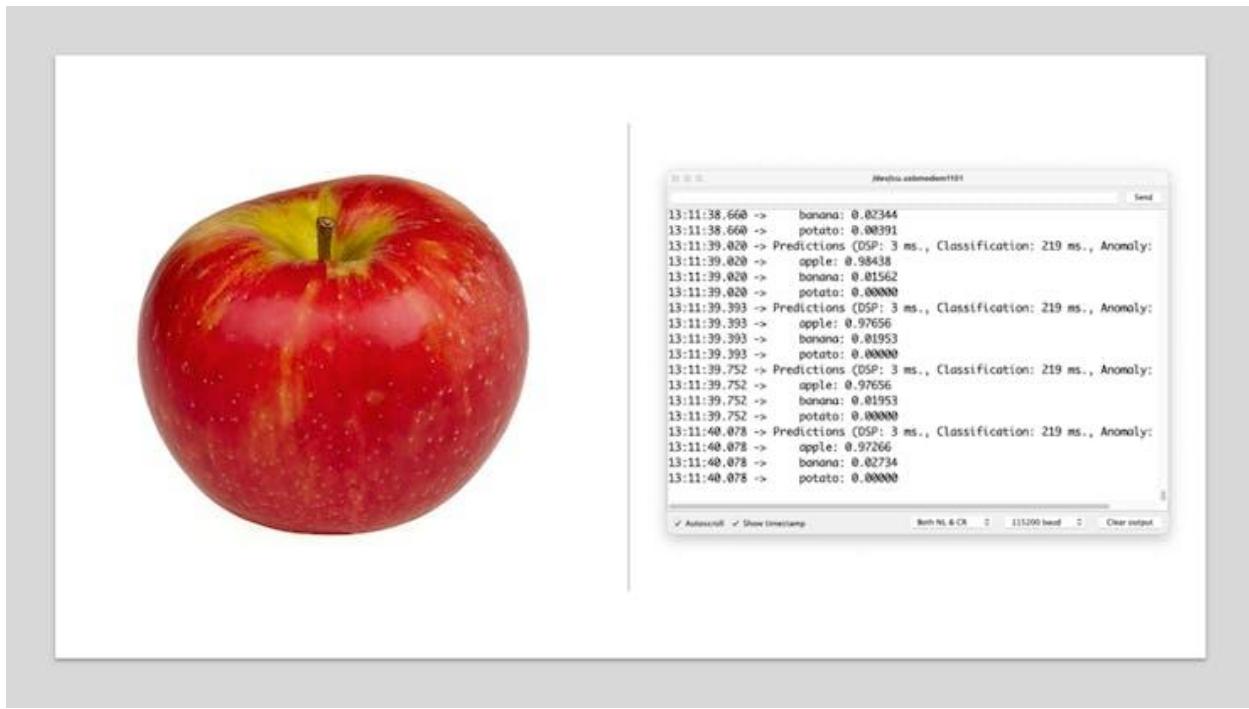
Meanwhile, you can download a preliminary version from the [project GitHub](#), unzip it, and replace the ESP NN folder with it under

`src/edge-impulse-sdk/porting/espressif/ESP-NN`, in your Arduino library folder.

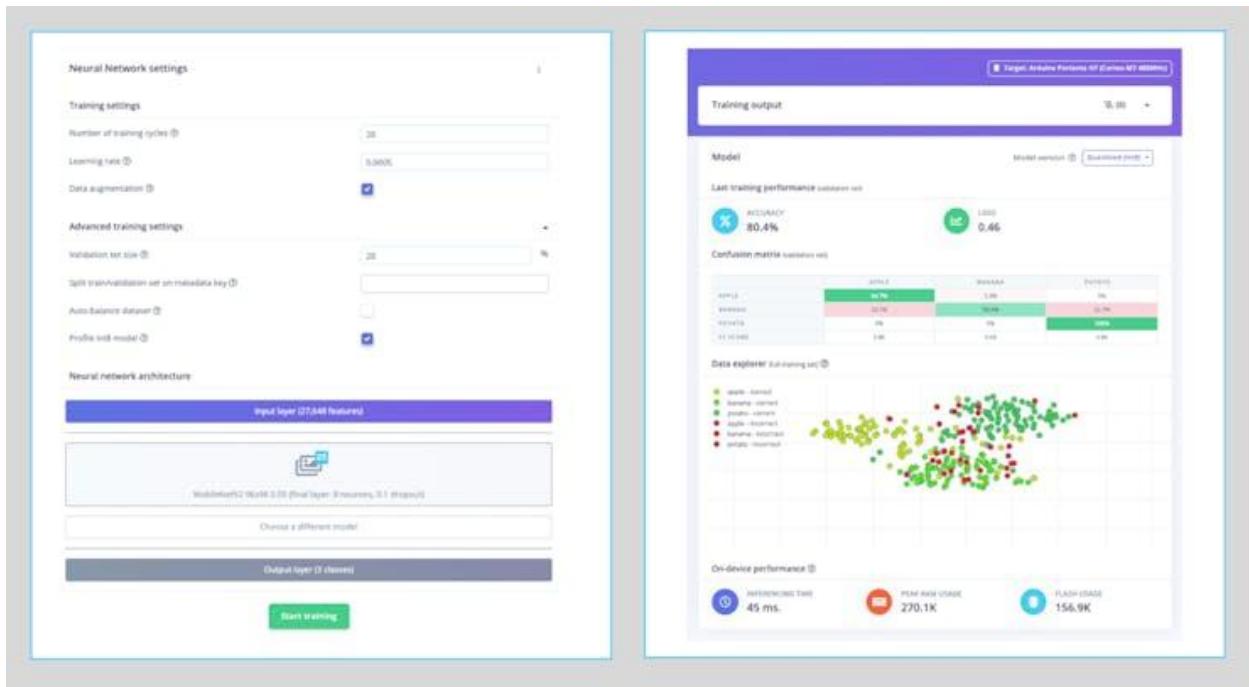


Then compile the sketch. Restarting the IDE after replacing the folder might be helpful.

Doing an inference with MobilinetV2 96x96 0.35, having as input RGB images, the latency was reduced from 2,383ms to 219ms, reducing it by more than ten times!

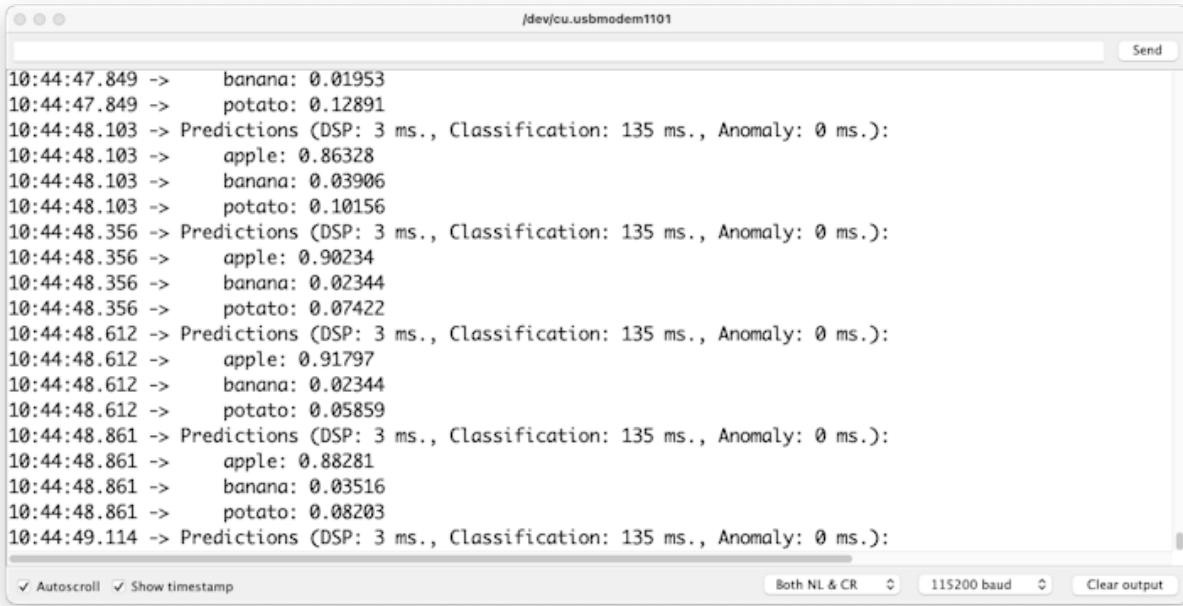


In my tests, this option works with MobileNet V2 but not V1. So, I trained the model again, using the smallest version of MobileNet V2, with an alpha of 0.05.



Note that the estimated latency for an Arduino Portenta (or Nicla), running with a clock of 480MHz, is 45ms.

Deploying the model, and applying the fix, replacing the ESP-NN folder, as explained before, I got an inference of only 135ms, remembering that the XIAO runs with half of the clock used by the Portenta/Nicla (240MHz):



The screenshot shows a terminal window titled '/dev/cu.usbmodem1101'. The window displays a series of log entries from an ESP32S3 Sense device. The entries show timestamped predictions for three classes: banana, potato, and apple. The predictions are accompanied by their respective confidence scores. The terminal interface includes checkboxes for 'Autoscroll' and 'Show timestamp', and a dropdown for 'Both NL & CR' and '115200 baud'. A 'Send' button is located in the top right corner, and a 'Clear output' button is at the bottom right.

```
10:44:47.849 ->    banana: 0.01953
10:44:47.849 ->    potato: 0.12891
10:44:48.103 -> Predictions (DSP: 3 ms., Classification: 135 ms., Anomaly: 0 ms.):
10:44:48.103 ->    apple: 0.86328
10:44:48.103 ->    banana: 0.03906
10:44:48.103 ->    potato: 0.10156
10:44:48.356 -> Predictions (DSP: 3 ms., Classification: 135 ms., Anomaly: 0 ms.):
10:44:48.356 ->    apple: 0.90234
10:44:48.356 ->    banana: 0.02344
10:44:48.356 ->    potato: 0.07422
10:44:48.612 -> Predictions (DSP: 3 ms., Classification: 135 ms., Anomaly: 0 ms.):
10:44:48.612 ->    apple: 0.91797
10:44:48.612 ->    banana: 0.02344
10:44:48.612 ->    potato: 0.05859
10:44:48.861 -> Predictions (DSP: 3 ms., Classification: 135 ms., Anomaly: 0 ms.):
10:44:48.861 ->    apple: 0.88281
10:44:48.861 ->    banana: 0.03516
10:44:48.861 ->    potato: 0.08203
10:44:49.114 -> Predictions (DSP: 3 ms., Classification: 135 ms., Anomaly: 0 ms.):
```

Conclusion

The XIAO ESP32S3 Sense is a very flexible, not expensive, and easy-to-program device. The project proves the potential of TinyML. Memory is not an issue; the device can handle many post-processing tasks, including communication. But you should consider that the high latency (without the ESP NN accelerator) will limit some applications spite the fact that the XIAO is 50% faster than the ESP32-CAM.

On the project GitHub repository, you will find the last version of the codes: [XIAO-ESP32S3-Sense](#).

Knowing more

If you want to learn more about Embedded Machine Learning (TinyML), please see these references:

- "[TinyML - Machine Learning for Embedding Devices](#)" - UNIFEI
- "[Professional Certificate in Tiny Machine Learning \(TinyML\)](#)" – edX/Harvard
- "[Introduction to Embedded Machine Learning](#)" - Coursera/Edge Impulse
- "[Computer Vision with Embedded Machine Learning](#)" - Coursera/Edge Impulse
- "[Deep Learning with Python](#)" by François Chollet
- "[TinyML](#)" by Pete Warden, Daniel Situnayake
- "[TinyML Cookbook](#)" by Gian Marco Iodice
- "[AI at the Edge](#)" by Daniel Situnayake, Jenny Plunkett

On the [TinyML4D website](#), You can find lots of educational materials on TinyML. They are all free and open-source for educational uses – we ask that if you use the material, please cite them! TinyML4D is an initiative to make TinyML education available to everyone globally.

As always, I hope this project can help others find their way in the exciting world of AI, Electronics, and IoT!

For more projects, please visit:

MJRoBot.org

link: MJRoBot.org

Greetings from the south of the world!

See you at my next project!

Thank you

Marcelo

