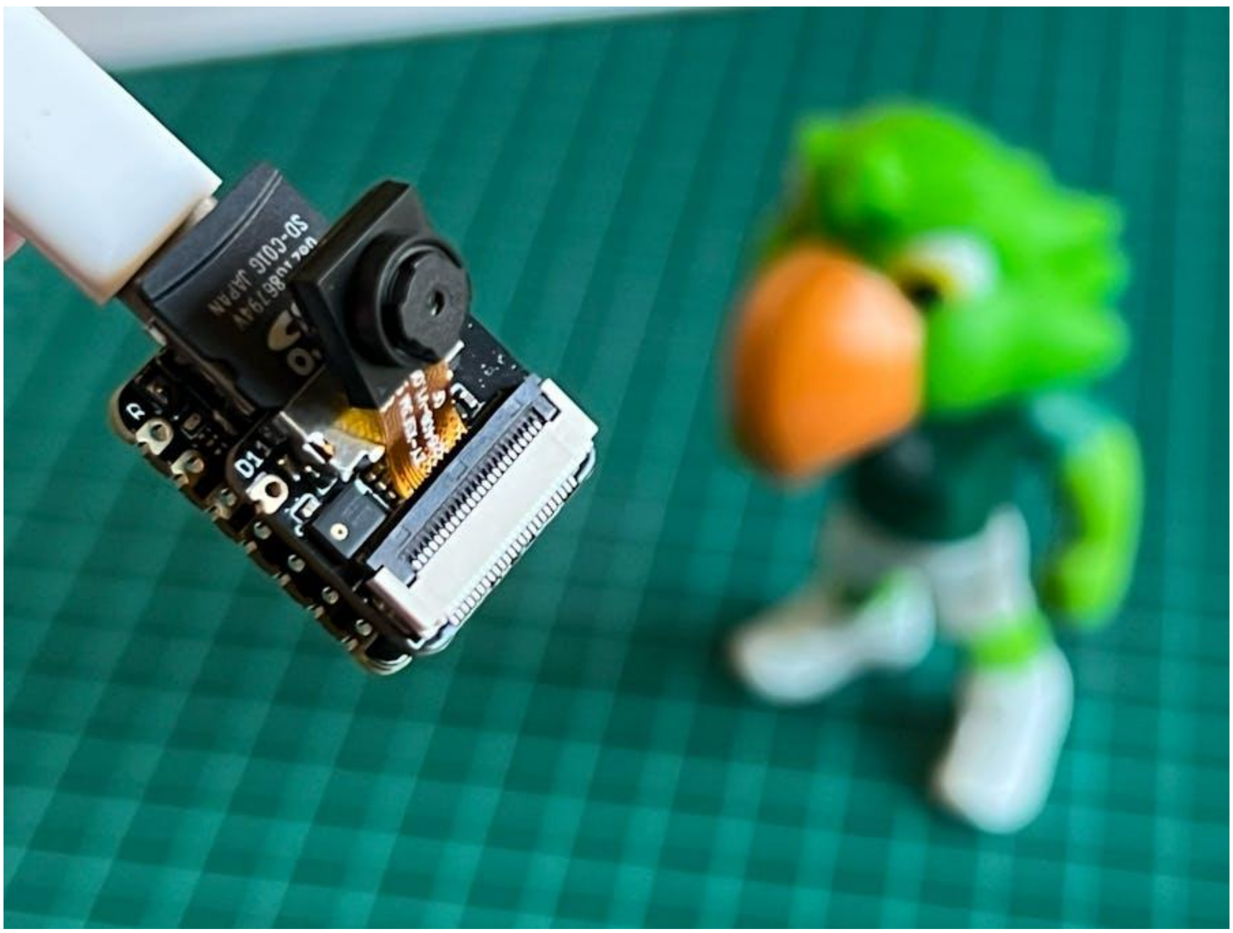
TinyML Made Easy
Image Classification

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"!

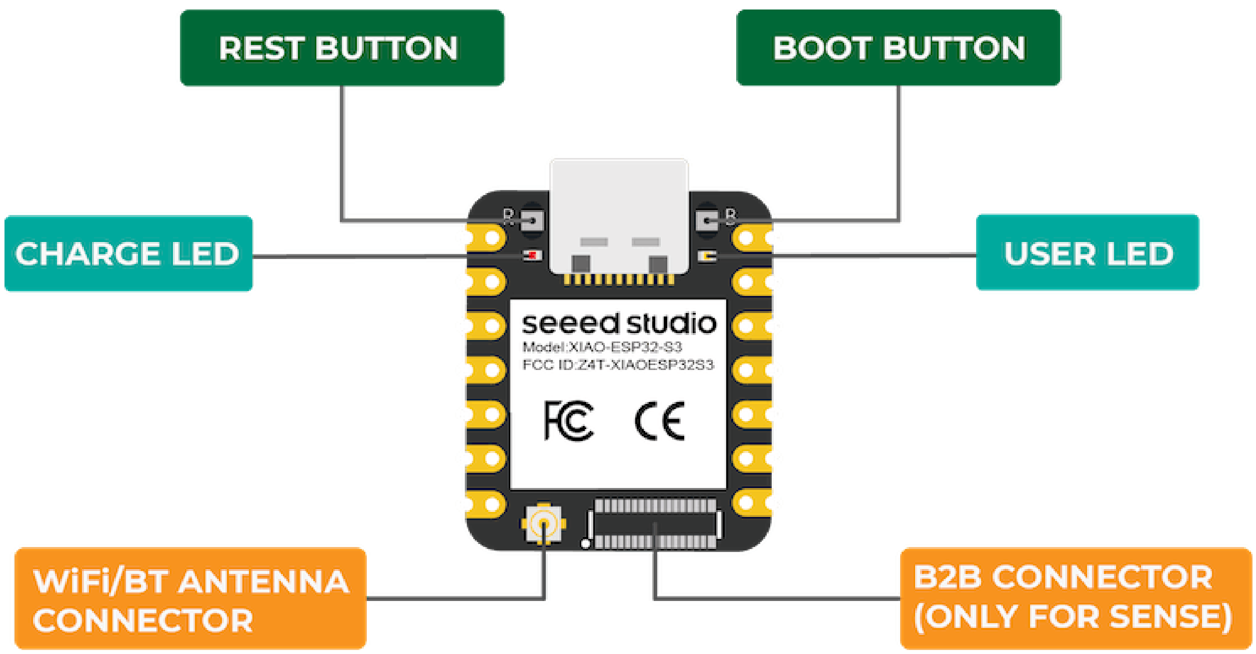
Seeed Studio released a new affordable development board, the XIAO ESP32S3 Sense, which integrates a camera sensor, digital microphone, and

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).

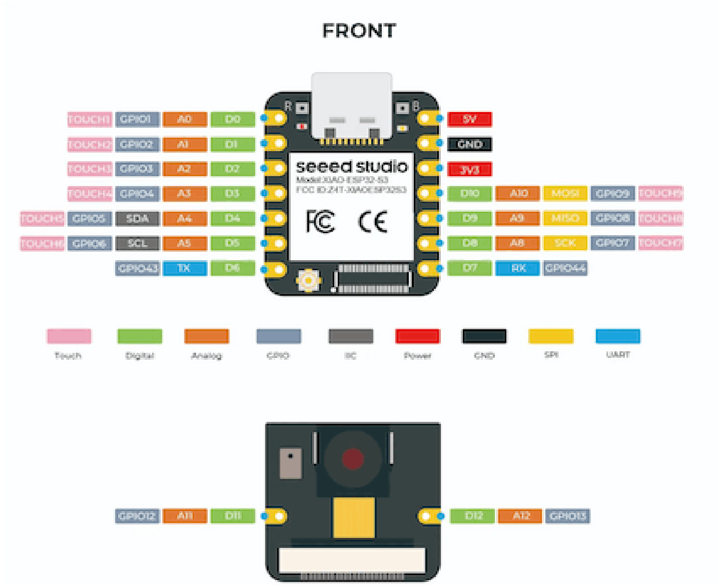
XIAO ESP32S3 Sense Main Features

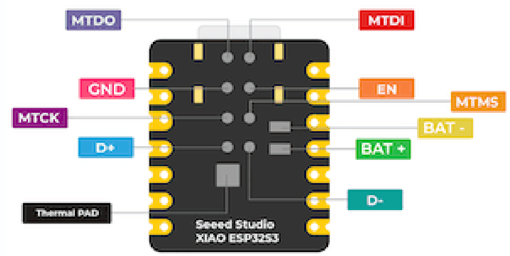
- **Powerful MCU Board**: Incorporate the ESP32S3 32-bit, dual-core, Xtensa processor chip operating up to 240 MHz, mounted multiple development ports, Arduino / MicroPython supported
- **Advanced Functionality**: Detachable OV2640 camera sensor for 1600*1200 resolution, compatible with OV5640 camera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





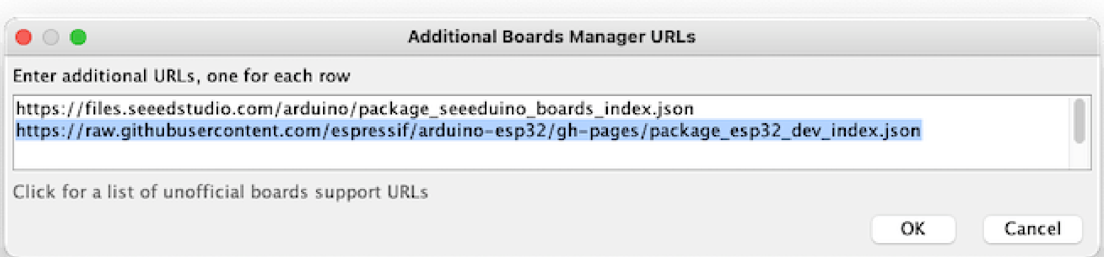
For more details, please refer to Seeed Studio WiKi page:
https://wiki.seeedstudio.com/xiao_esp32s3_getting_started/

Installing the XIAO ESP32S3 Sense on Arduino IDE

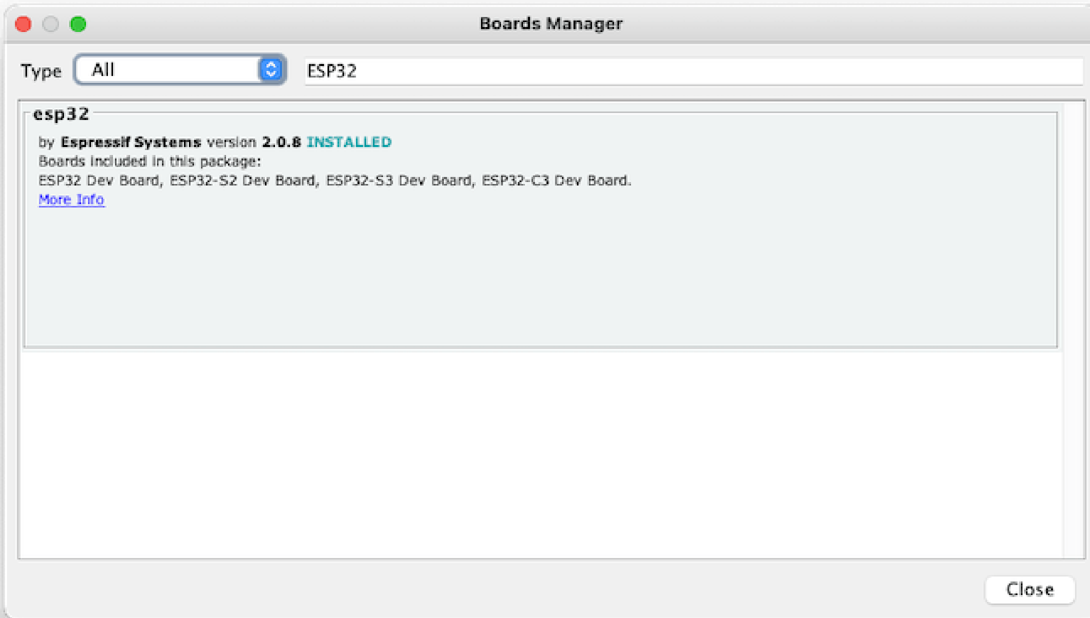
On Arduino IDE, navigate to **File > Preferences**, and fill in the URL:

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

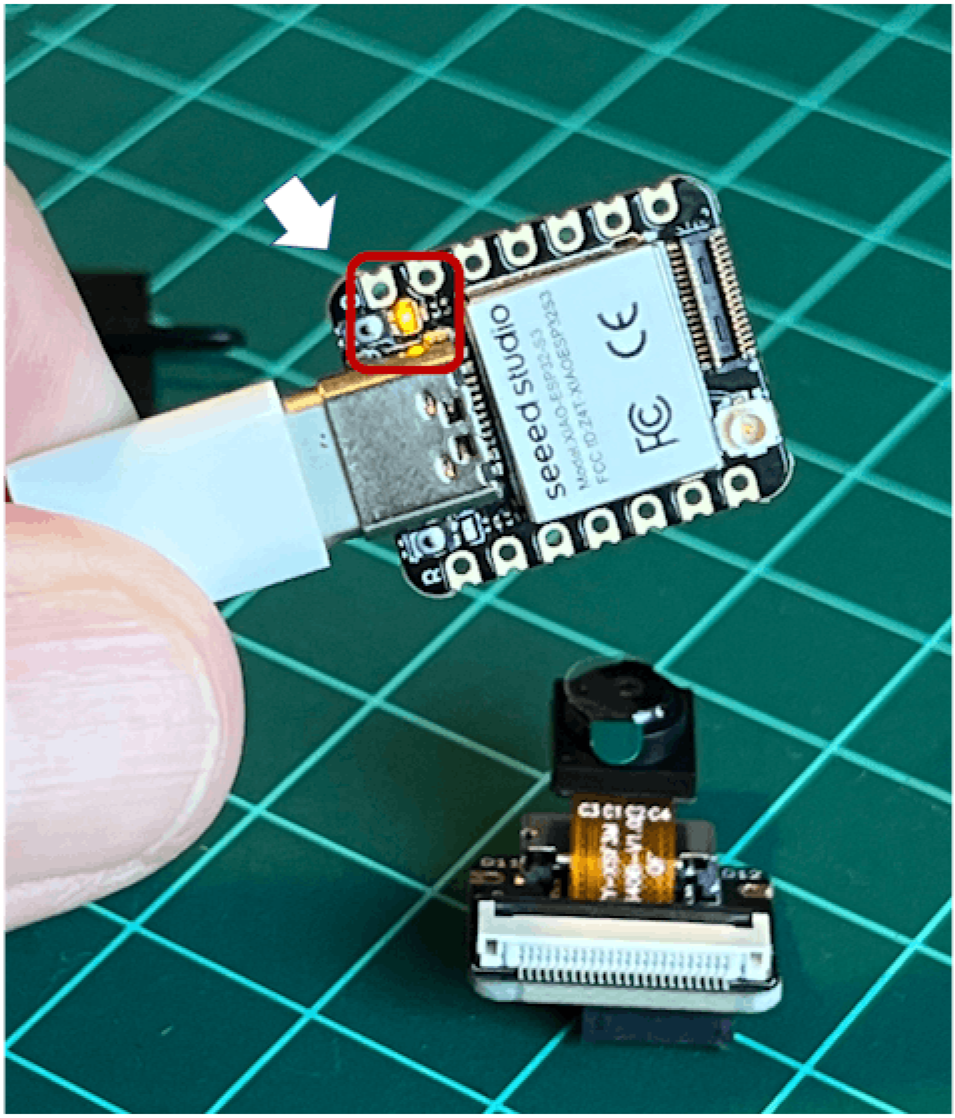
The XIAO ESP32S3 Sense has a built-in LED that is connected to GPIO21. So, you can run the blink sketch as it (using the `LED_BUILTIN` Arduino constant) or by changing the Blink sketch accordantly:

```
#define LED_BUILT_IN 21

void setup() {
 pinMode(LED_BUILT_IN, 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_BUILT_IN, LOW); //Turn on
 delay (1000); //Wait 1 sec
 digitalWrite(LED_BUILT_IN, 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



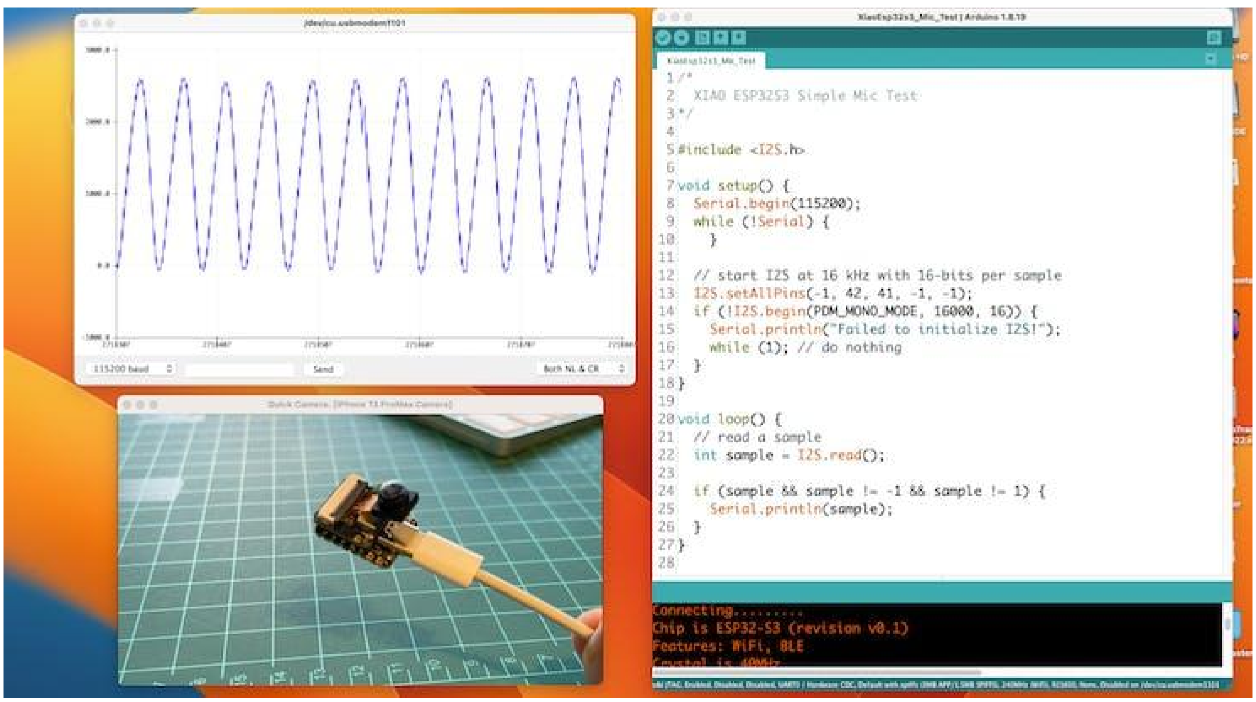
Connecting Sense module (Expansion Board)

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:



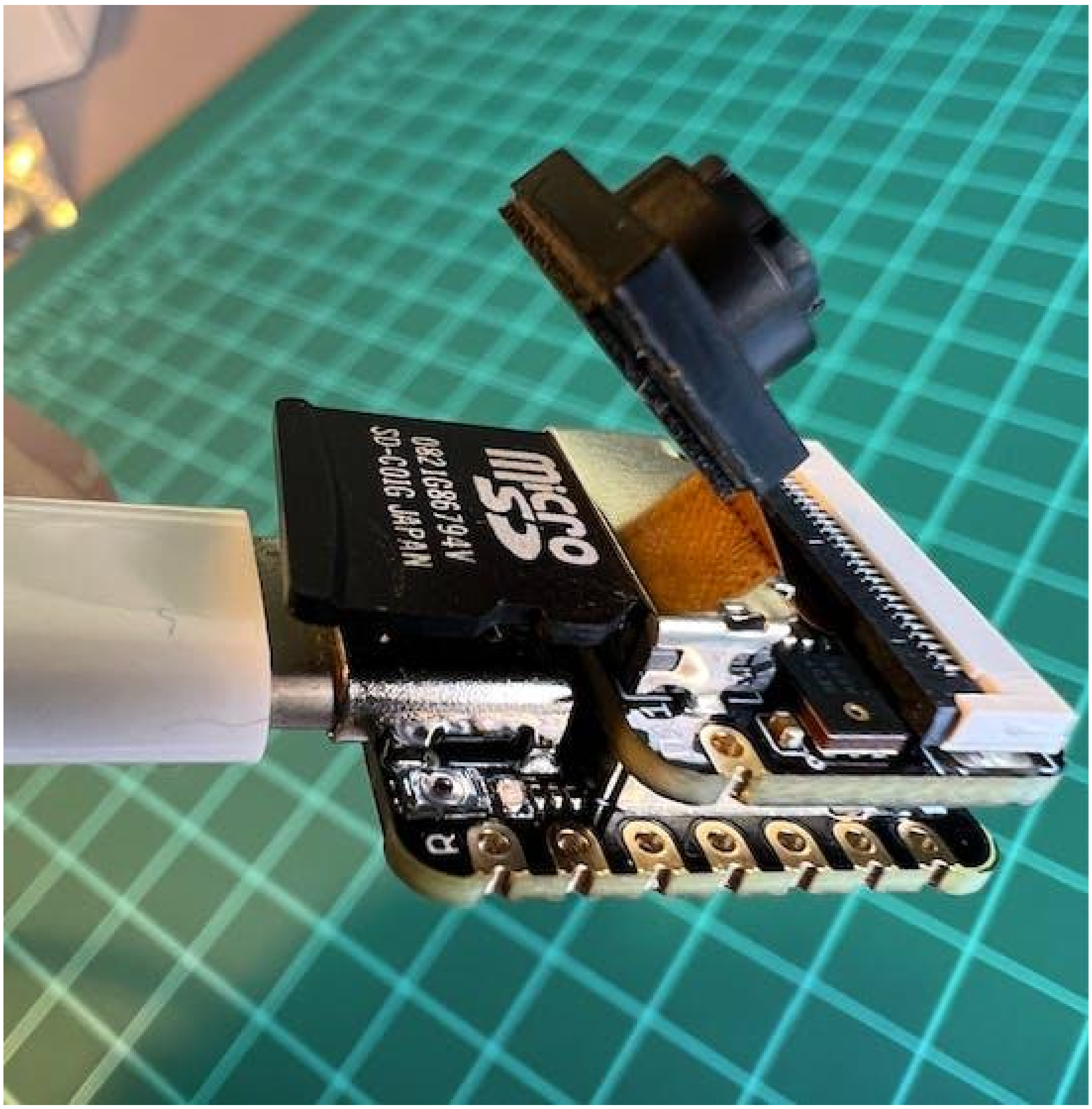
When producing sound, you can verify it on the Serial Plotter.

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.

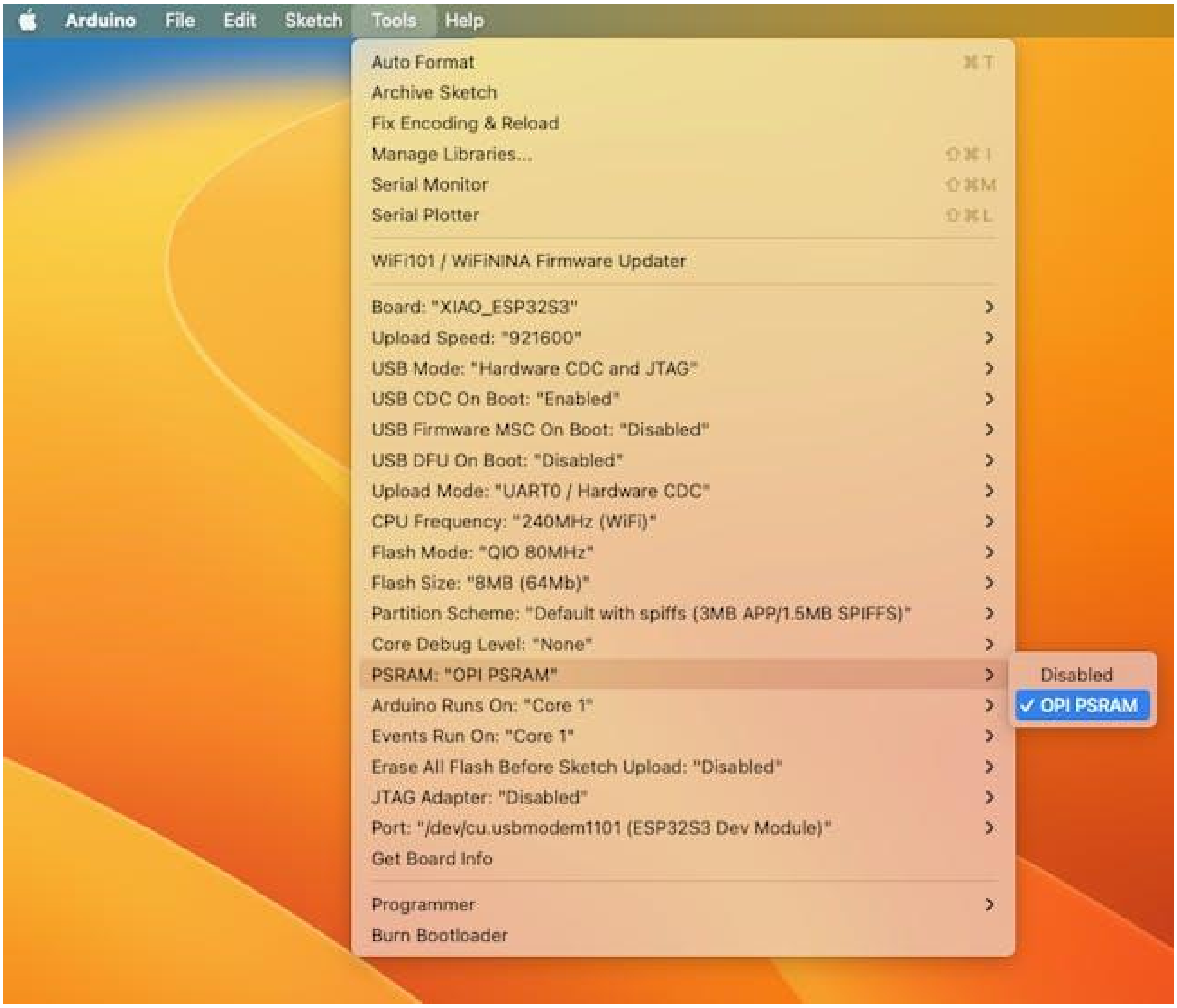
ESP32-S3 has only a few hundred kilobytes of internal RAM on the MCU chip. It can be insufficient for some purposes, so ESP32-S3 can use up to 16 MB of external PSRAM (Psuedostatic RAM) connected in parallel with the SPI flash chip. The external memory is incorporated in the memory map and, with certain 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.

The sound quality is excellent!

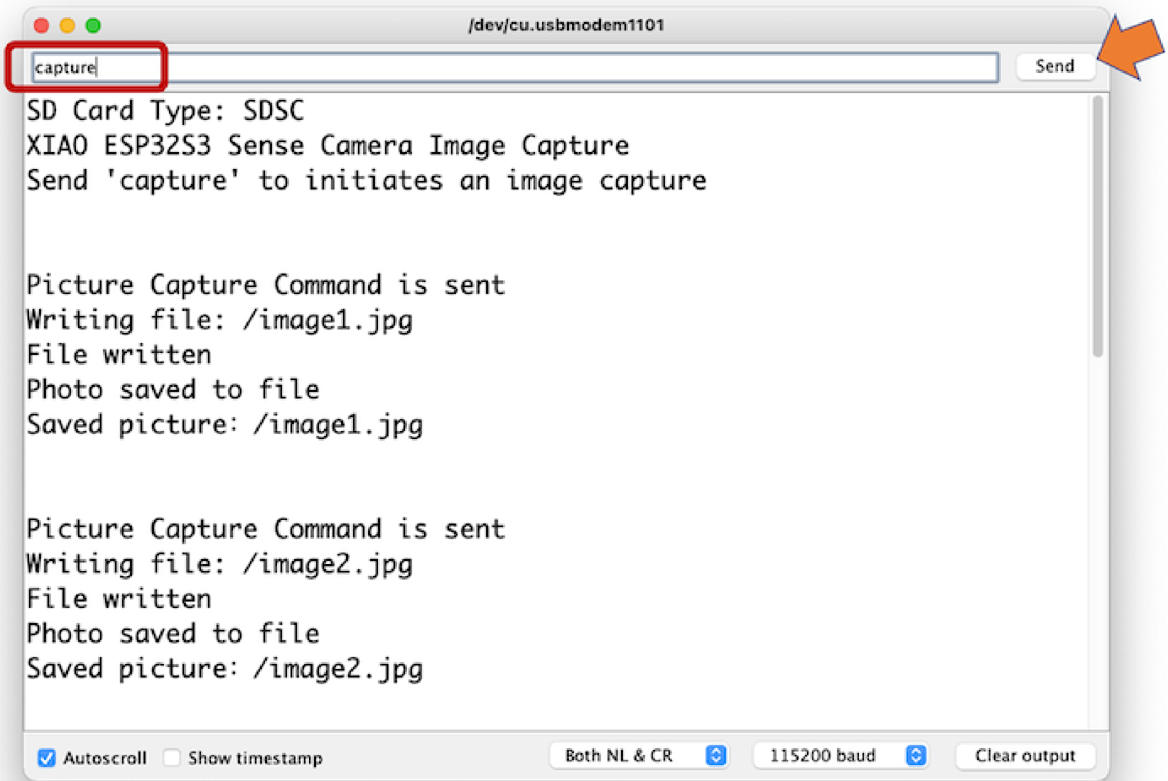
The explanation of how the code works is beyond the scope of this tutorial, but you can find an excellent description on the wiki page.

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.



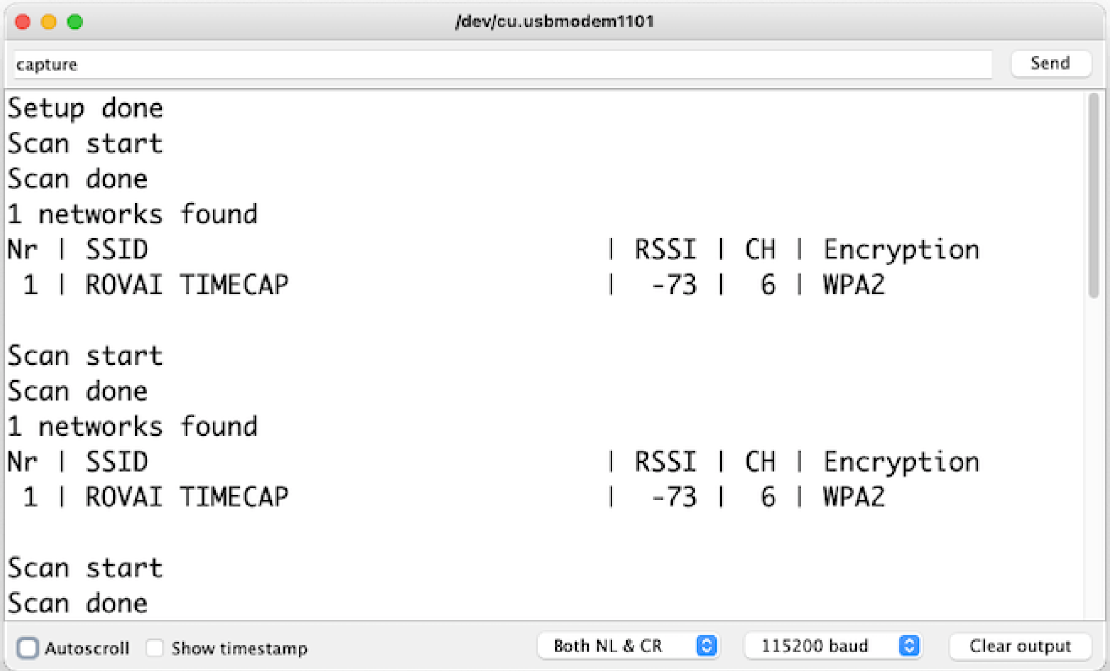
Here is an example of a taken photo:

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:



Simple WiFi Server (Turning LED ON/OFF)

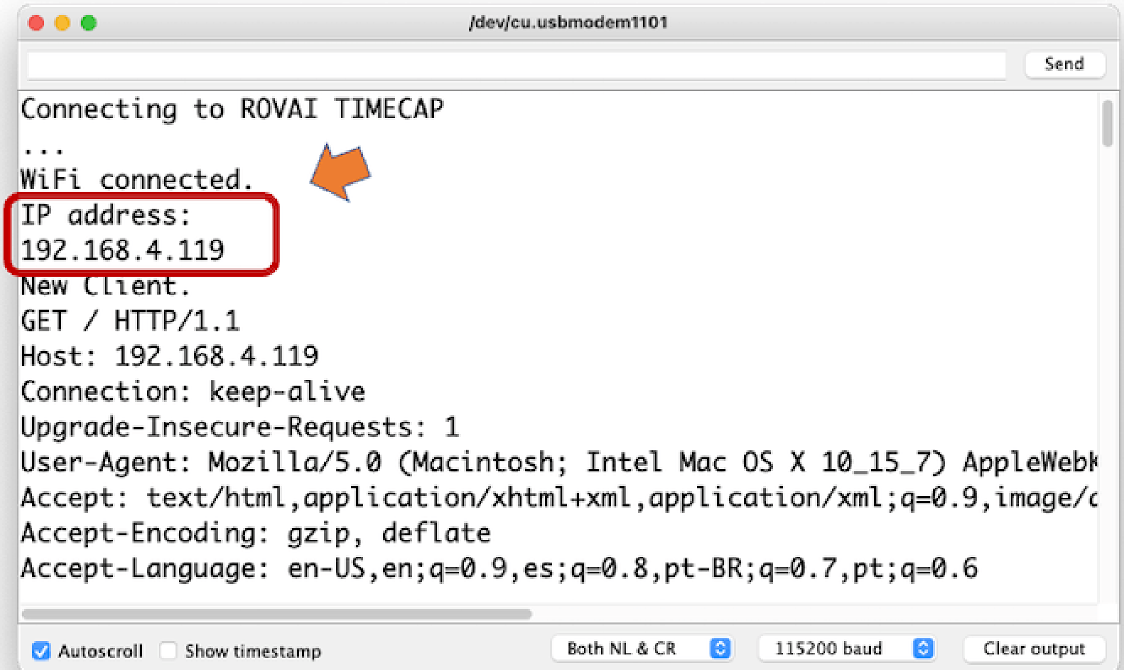
Let's test the device's capability to behave as a WiFi Server. We will host a simple page on the device that sends commands to turn the XIAO built-in LED ON and OFF.

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

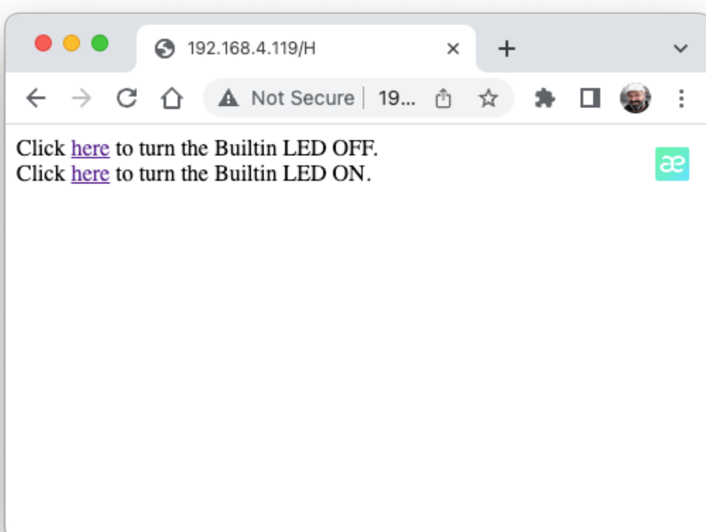
Before running the sketch, you should enter your network credentials:

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

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



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.

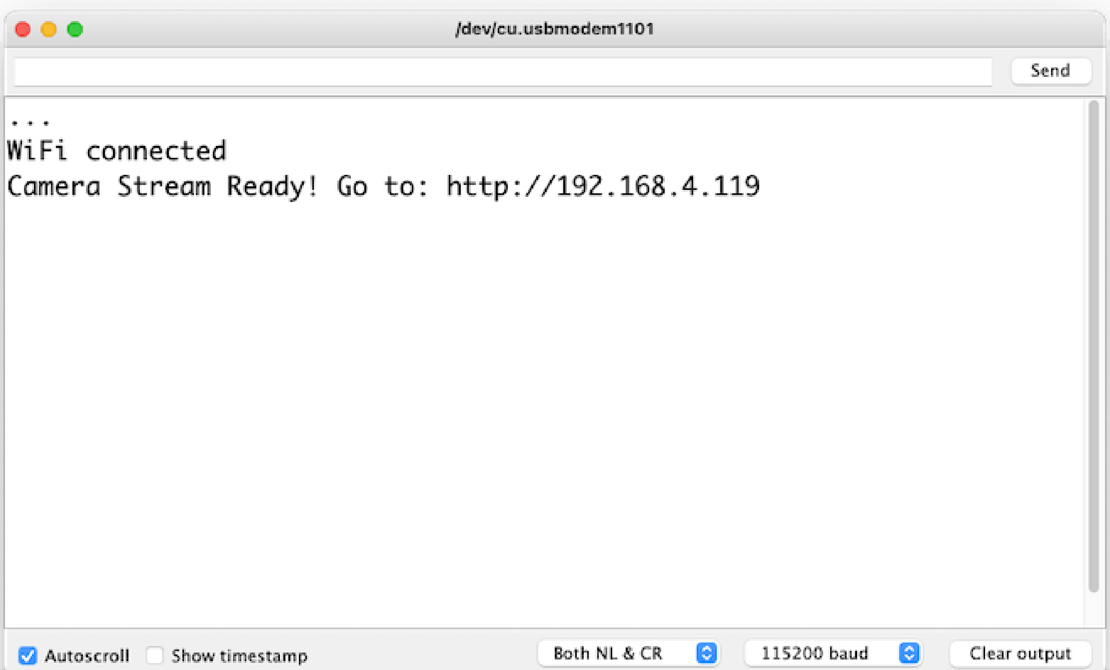
Streaming video to Web

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:

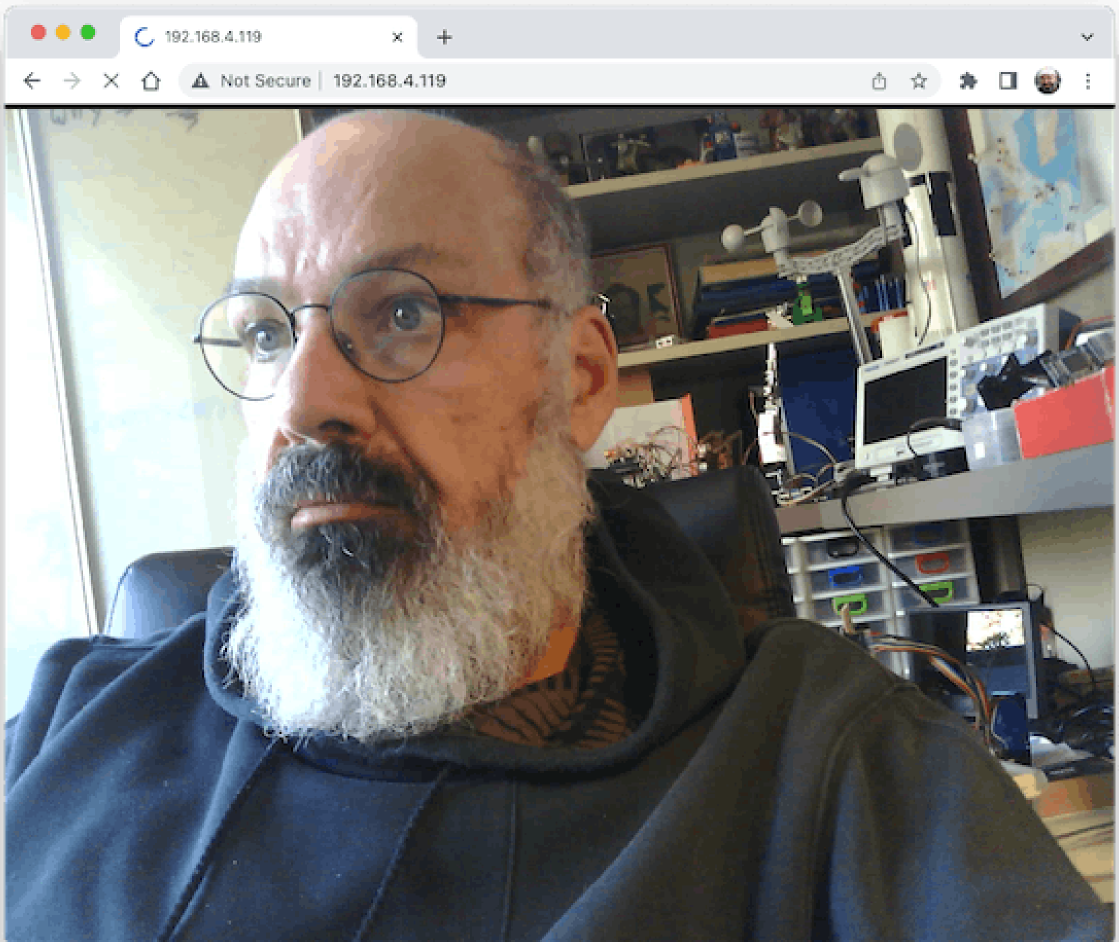
Download from GitHub the folder that contains the code: XIAO-ESP32S3-Streeming_Video.ino.

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.



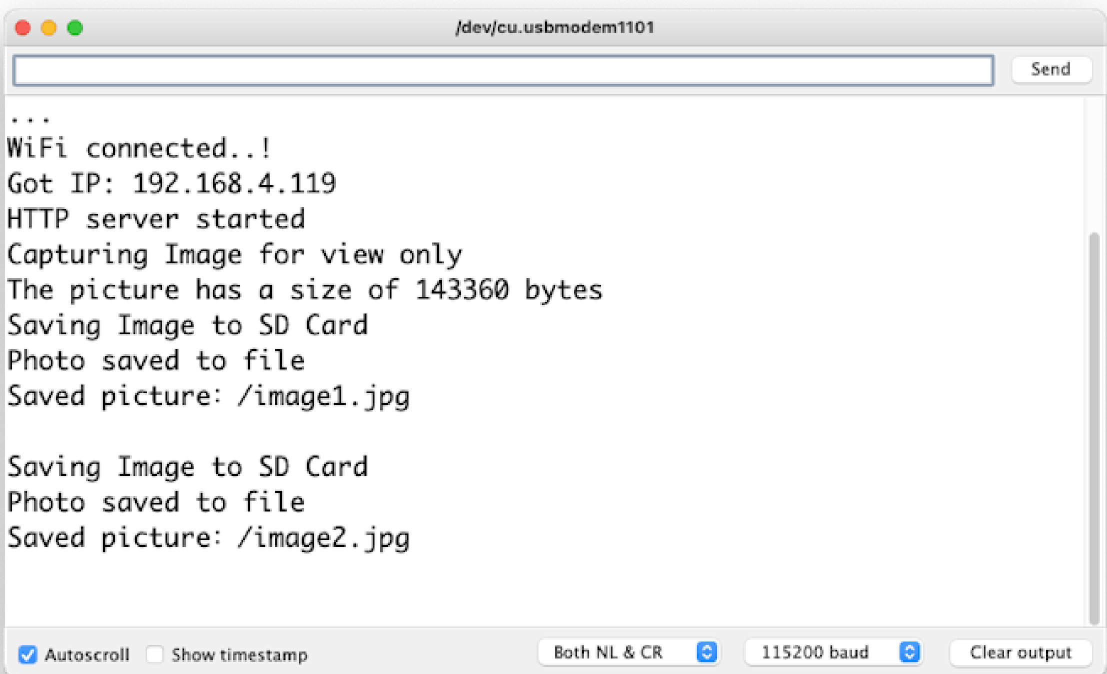
Streamlining what your camera is "seen" can be important when you position it to capture a dataset for an ML project (for example, using the code "take_phots_commands.ino".

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.



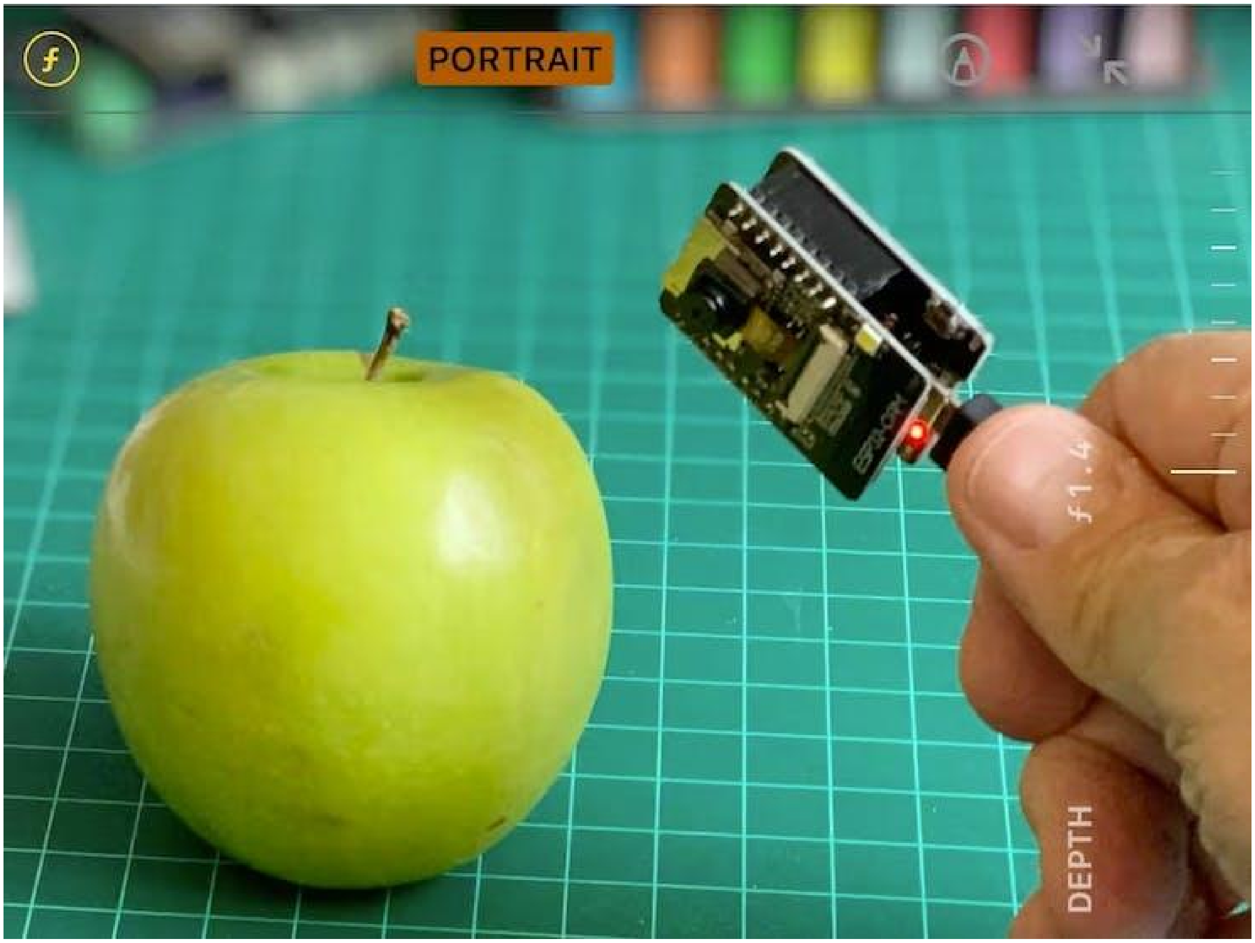
This program can be used for an image dataset capture with an Image Classification project.

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).

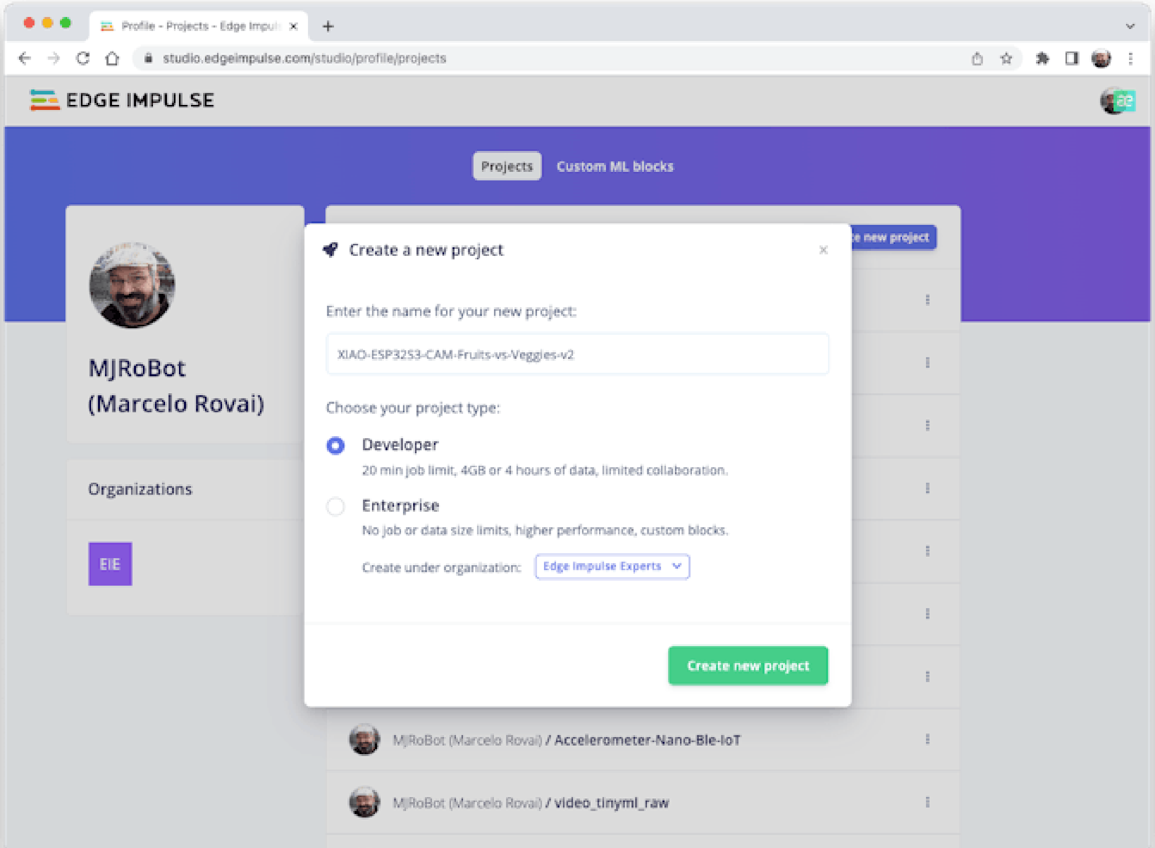
- Download the dataset from the Kaggle website to your computer.

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

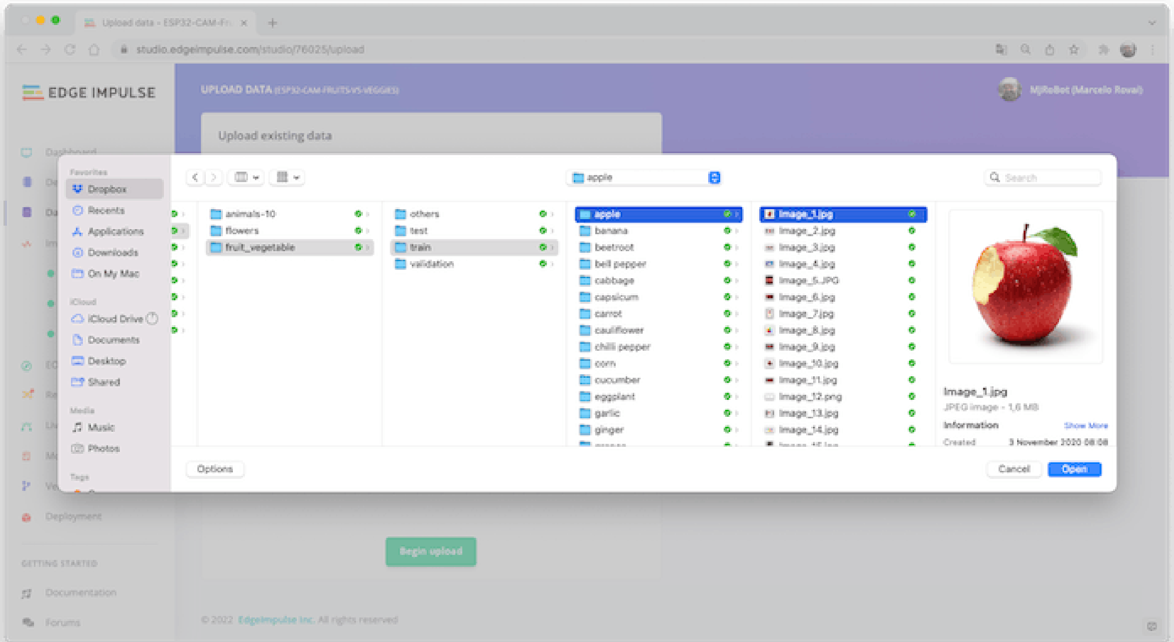
We will use the Edge Impulse Studio for training our model. Edge Impulse is a leading development platform for machine learning on edge devices.

Enter your account credentials (or create a free account) at Edge Impulse. Next, create a new project:

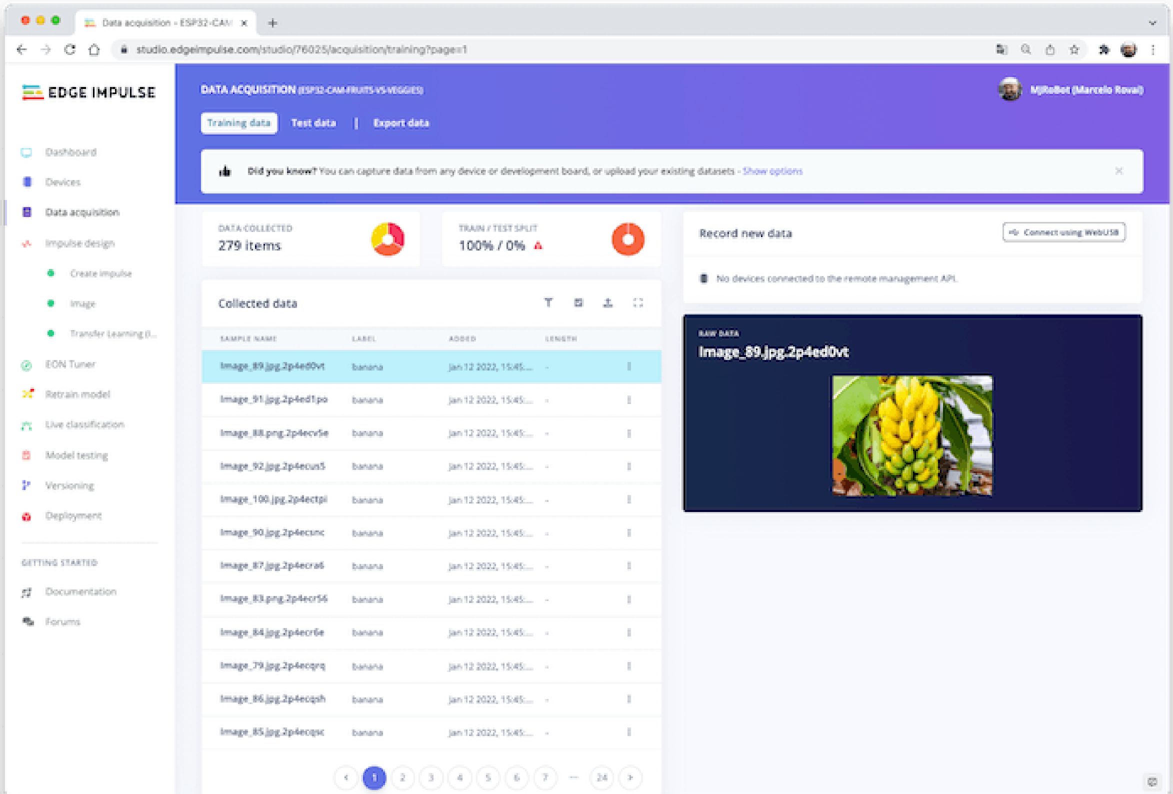


Data Acquisition

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:

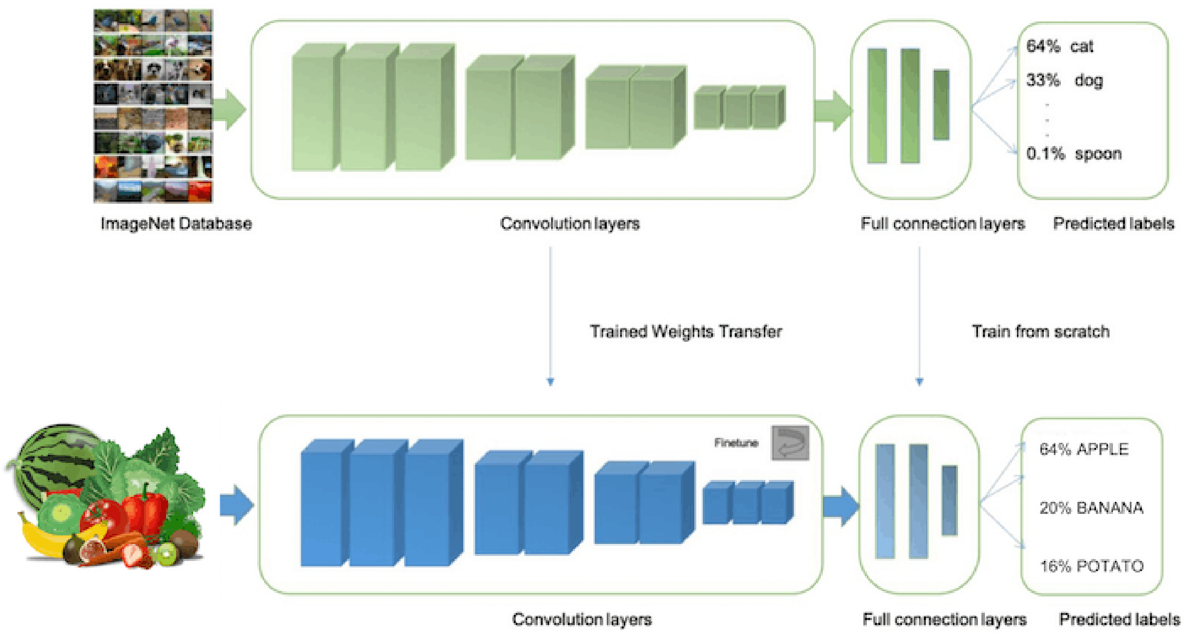


You can upload extra data for further model testing or split the training data. I will leave it as it is to use the most data possible.

Impulse Design

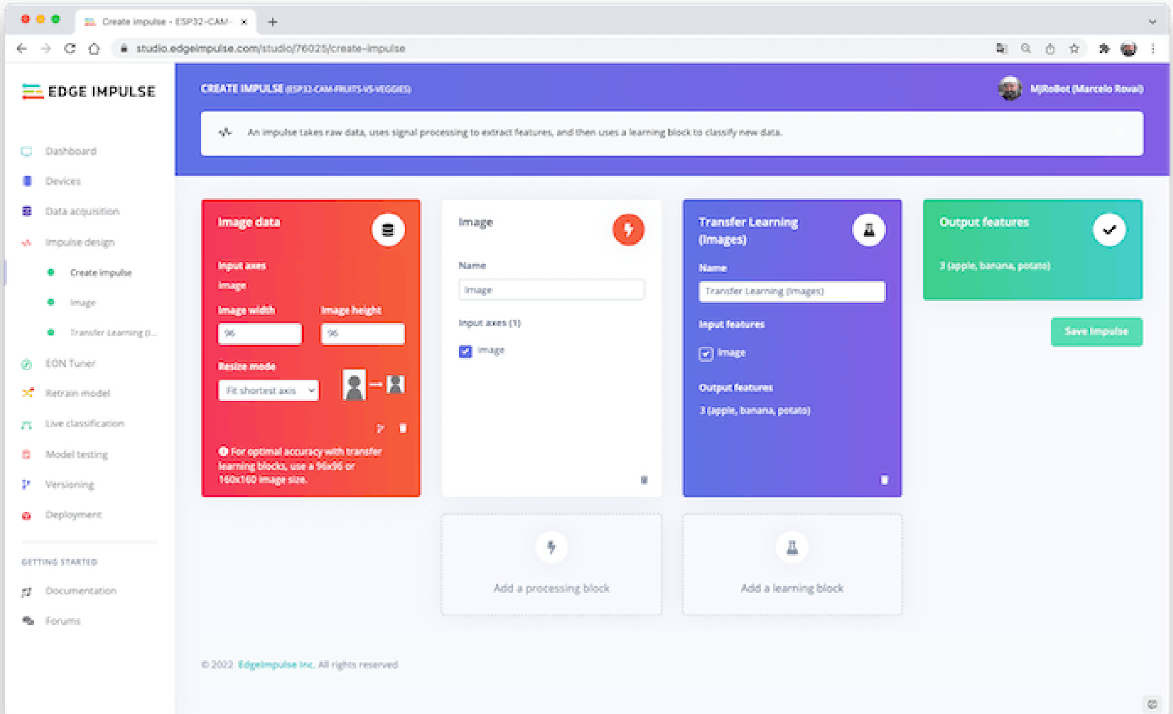
An impulse takes raw data (in this case, images), extracts features (resize pictures), and then use a learning block to classify new data.

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).



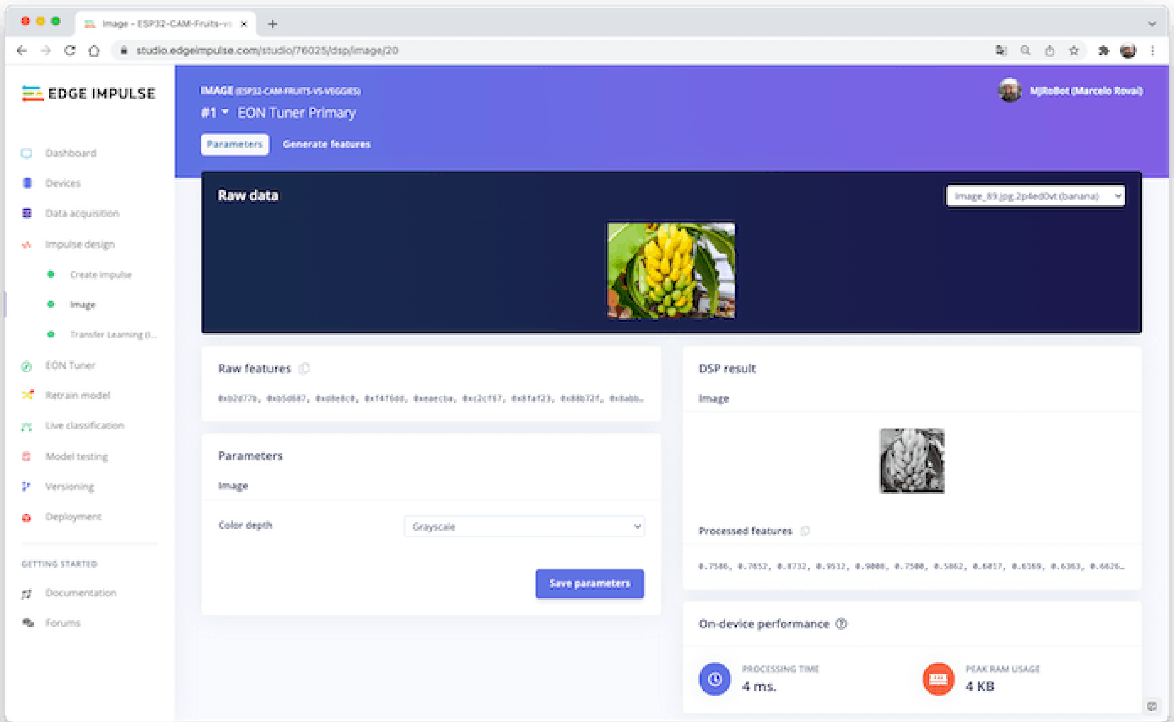
With TL, we can fine-tune a pre-trained image classification model on our data, performing well even with relatively small image datasets (our case).

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 α=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 α=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 α=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
```

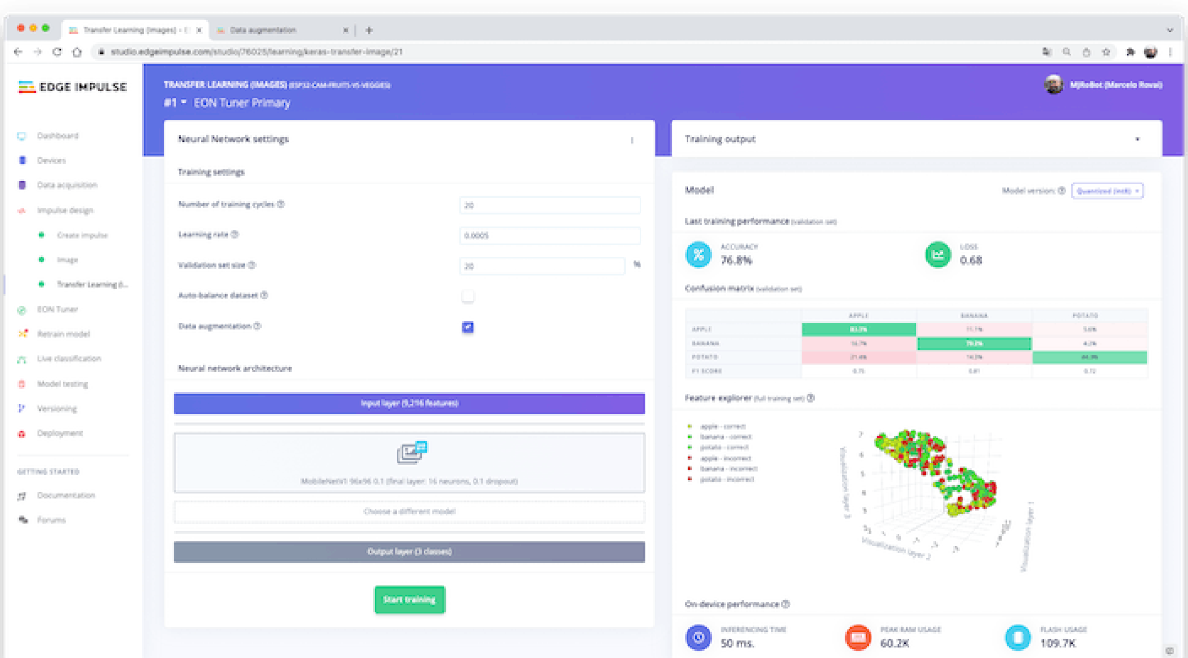
```
    resize_factor = random.uniform(1, 1.2)
    new_height = math.floor(resize_factor * INPUT_SHAPE[0])
    new_width = math.floor(resize_factor * INPUT_SHAPE[1])
    image = tf.image.resize_with_crop_or_pad(image, new_height, new_width)
    image = tf.image.random_crop(image, size=INPUT_SHAPE)

    # 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.

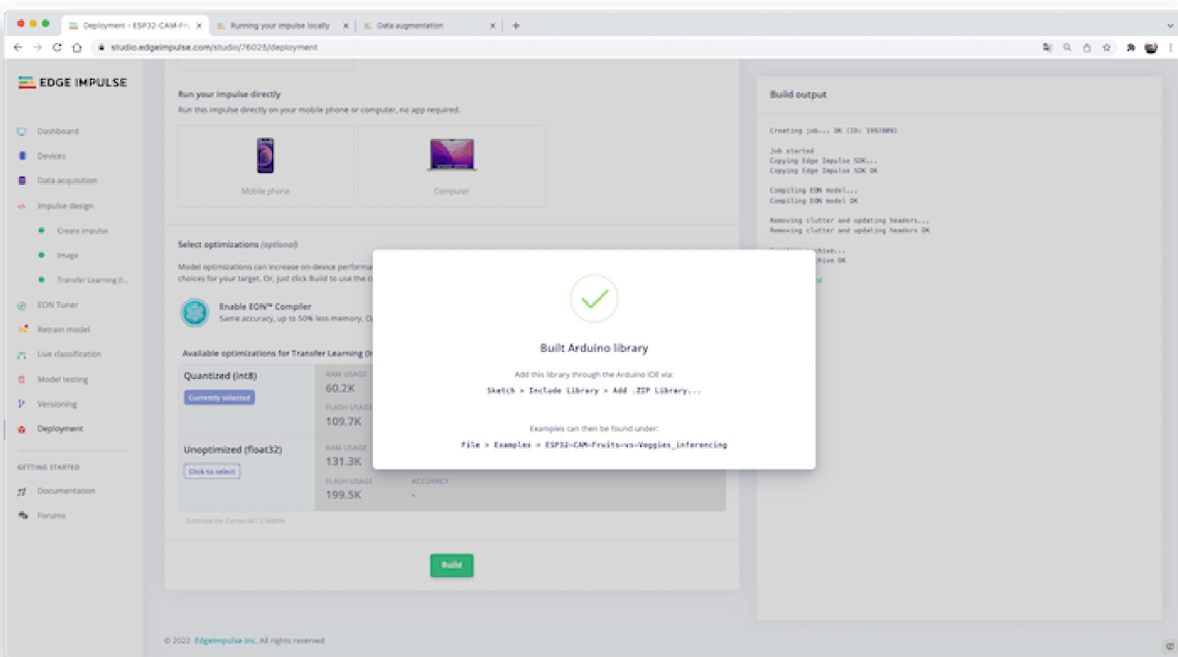
The final layer of our model will have 16 neurons with a 10% of dropout for overfitting prevention. Here is the Training output:



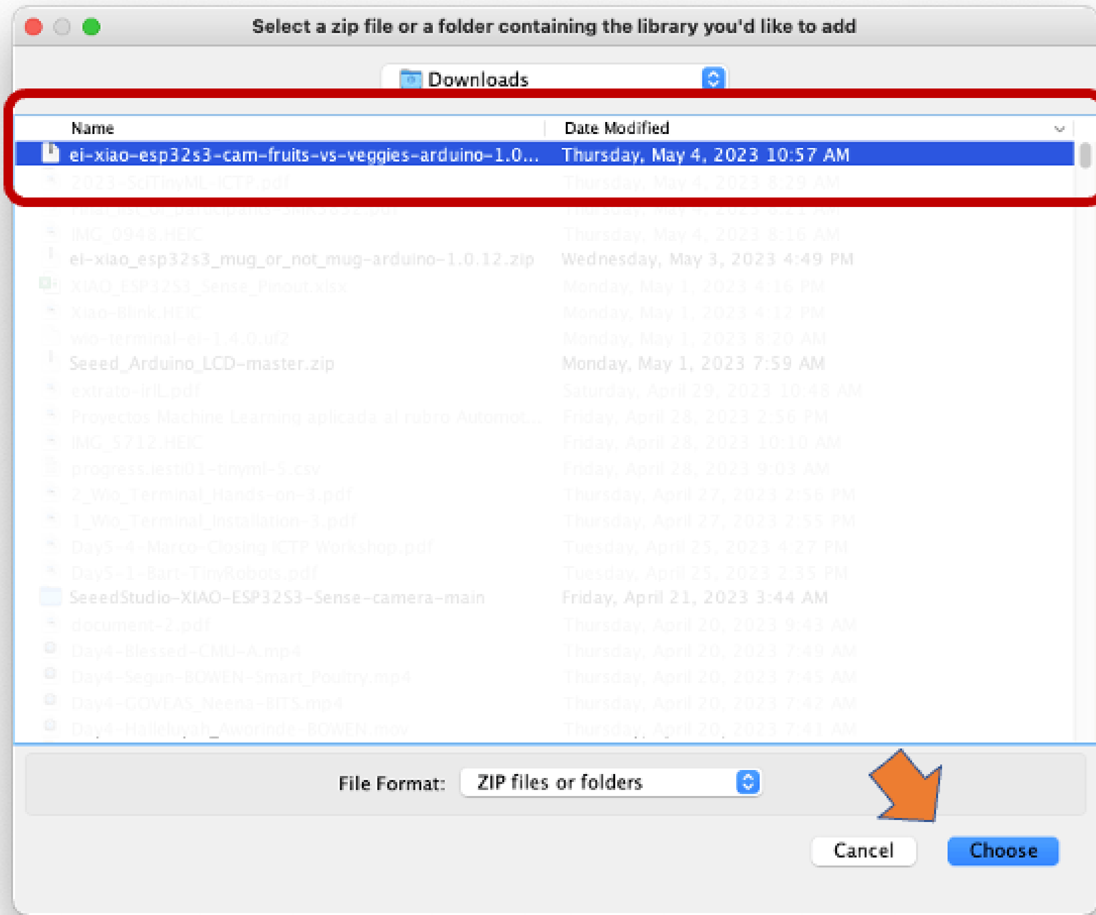
The result is not great. The model reached around 77% of accuracy, but the amount of RAM expected to be used during the inference is relatively small (around 60 KBytes), which is very good.

Deployment

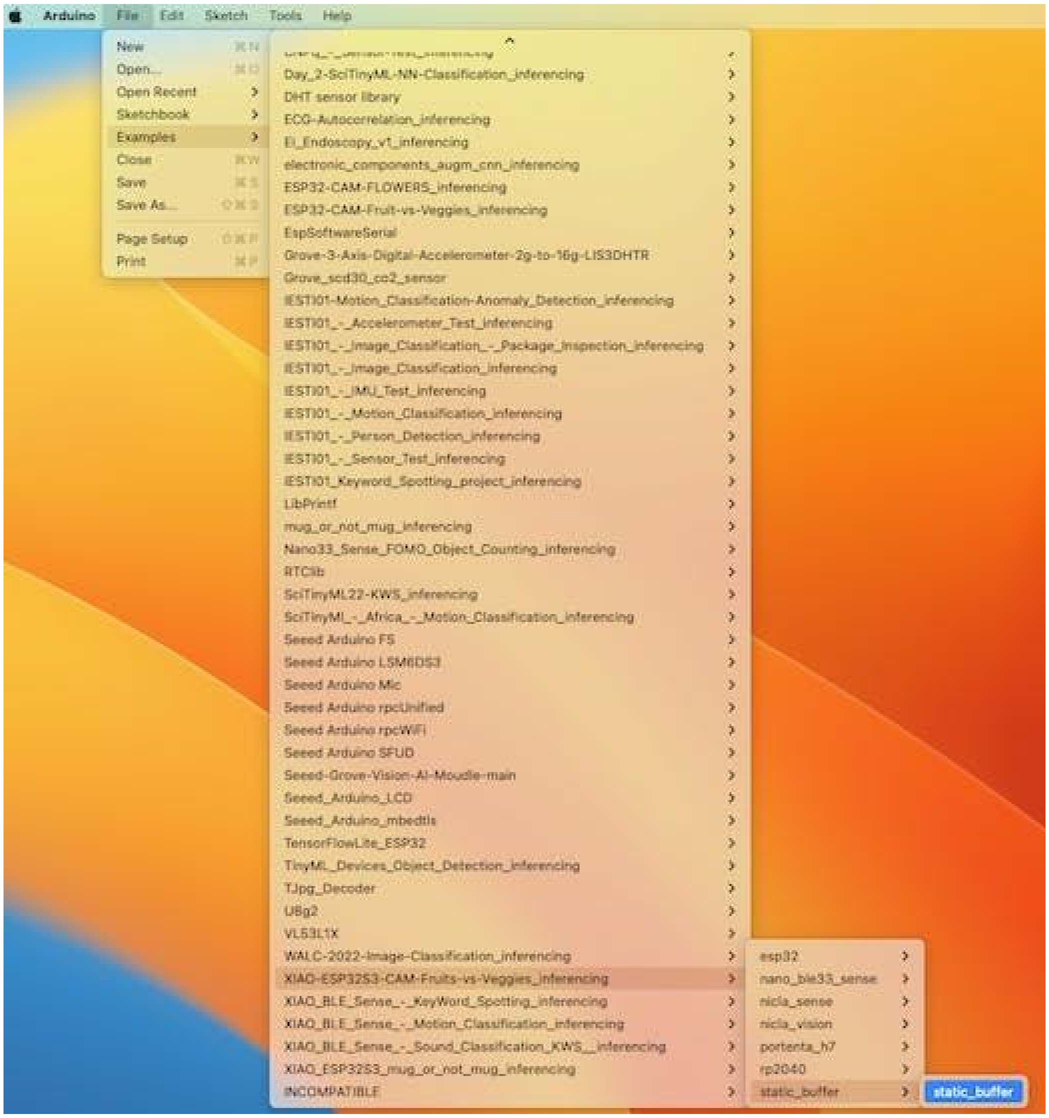
The trained model will be deployed as a.zip Arduino library:



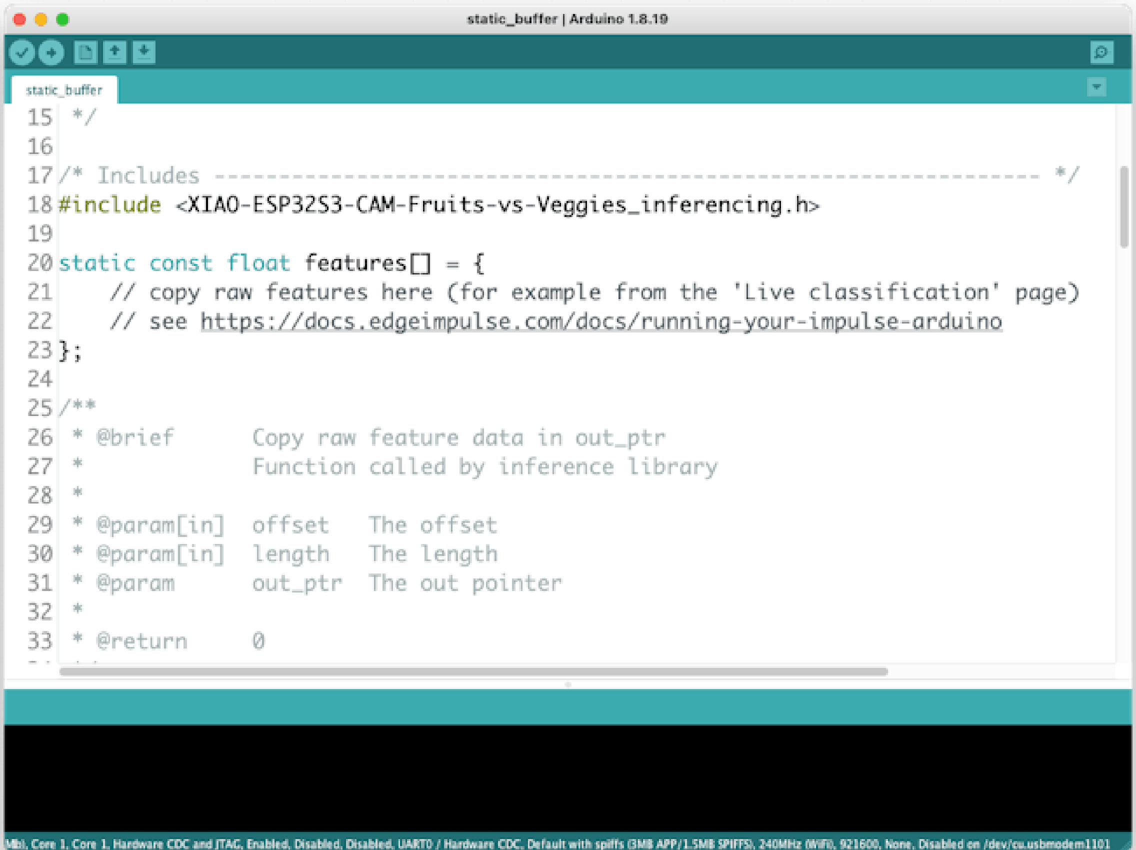
Open your Arduino IDE, and under **Sketch,** go to **Include Library** and **add.ZIP Library.** Select the file you download from Edge Impulse Studio, and that's it!



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



Open the Static Buffer example:

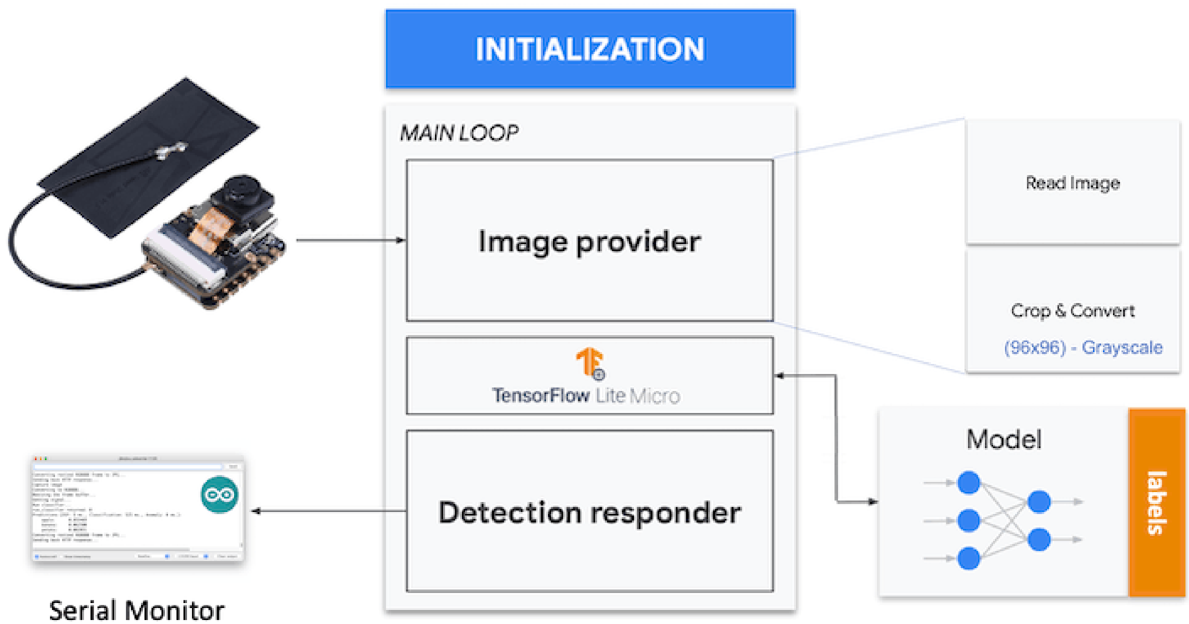


You can see that the first line of code is exactly the calling of a library with all the necessary stuff for running inference on your device.

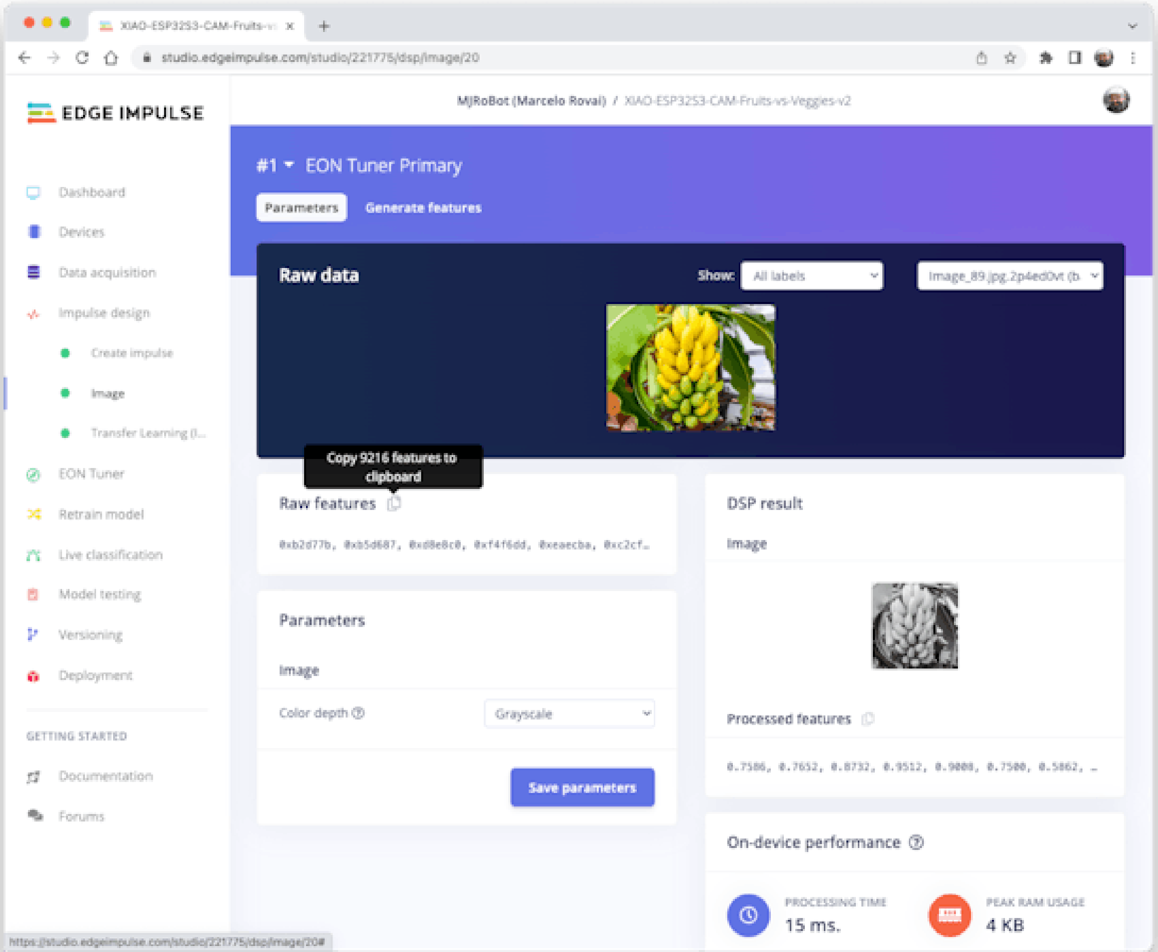
```
#include <XIAO-ESP32S3-CAM-Fruits-vs-Veggies_inferencing.h>
```

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.

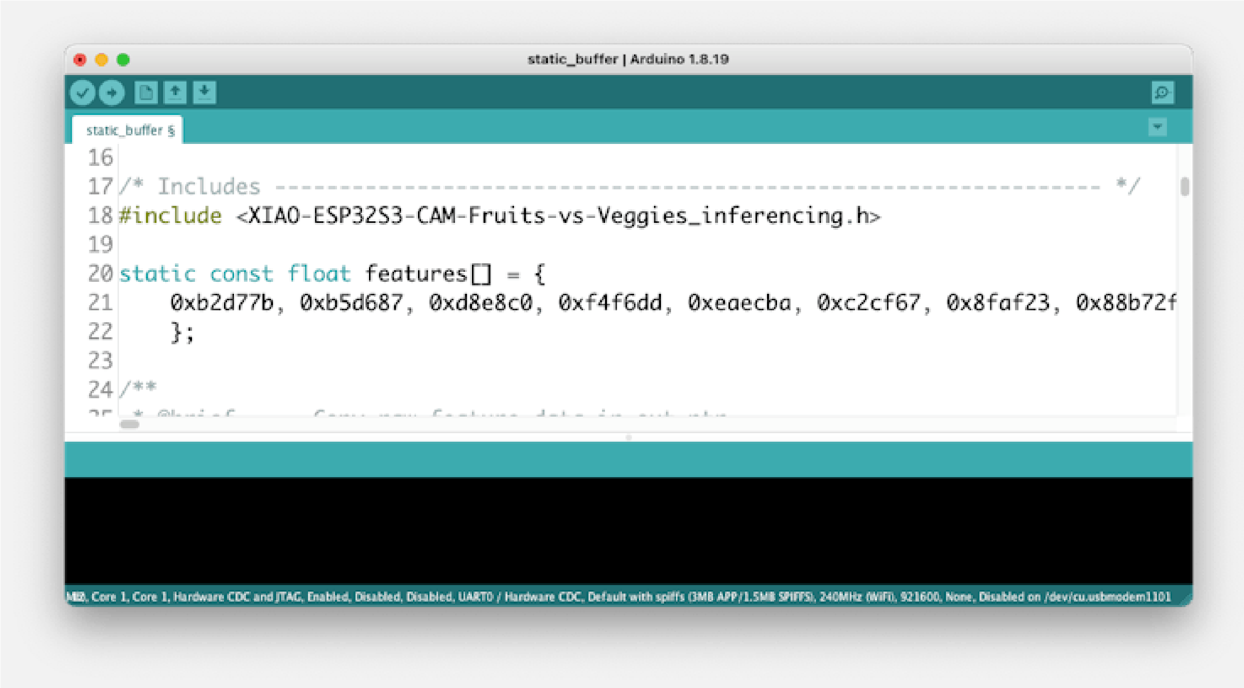
We should get the sample (image) from the camera and pre-process it (resizing to 96x96, converting to grayscale, and flatting it). This will be the input tensor of our model. The output tensor will be a vector with three values (labels), showing the probabilities of each one of the classes.



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

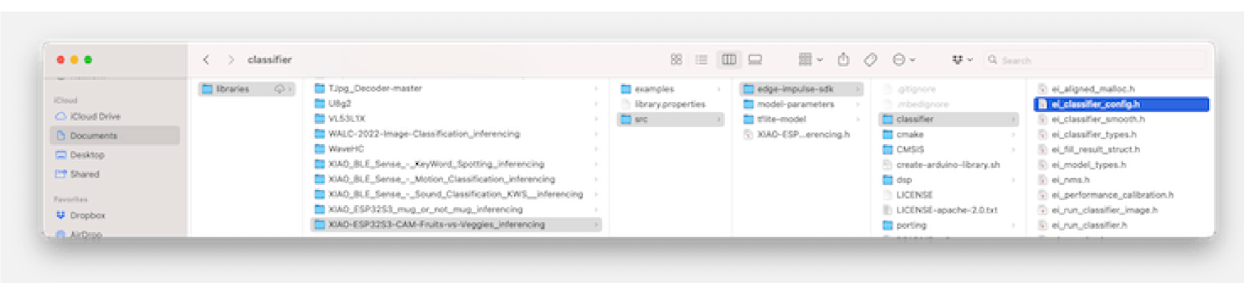


9, 216 features will be copied to the clipboard. This is the input tensor (a flattened image of 96x96x1), in this case, bananas. Past this Input tensor on `features[] = {0xb2d77b, 0xb5d687, 0xd8e8c0, 0xeaecba, 0xc2cf67, ...}`

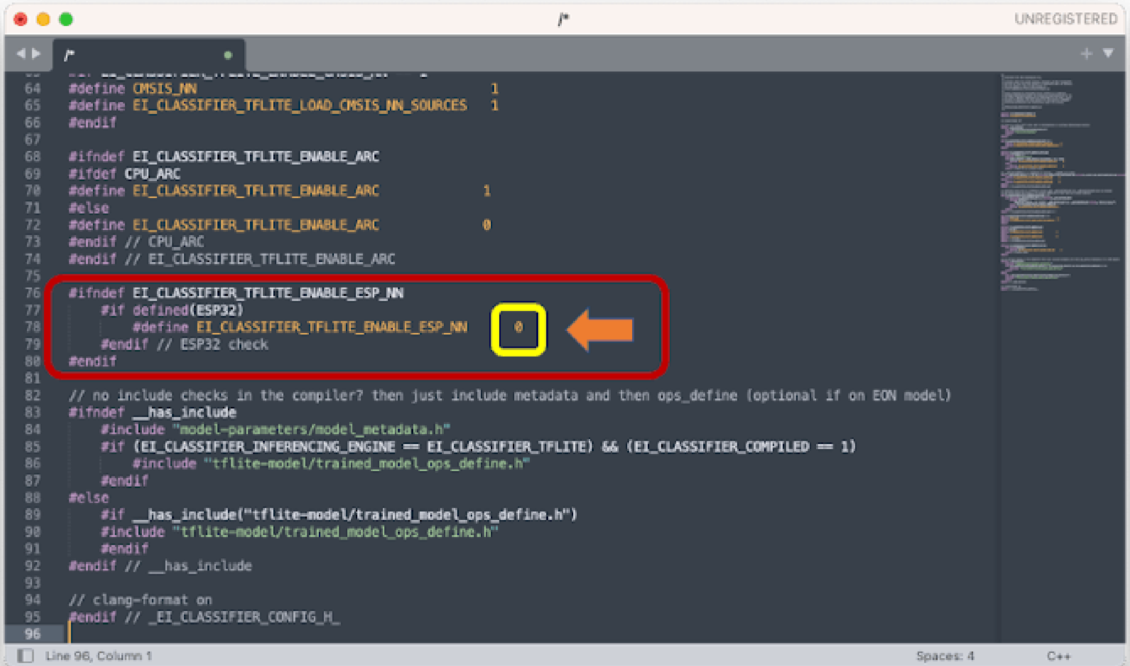


NOTE: Edge Impulse included the [library ESP NN](#) in its SDK, which contains optimized NN (Neural Network) functions for various Espressif chips. Until June 2023, the ESP NN was not working with the ESP32S3 (Arduino IDE).

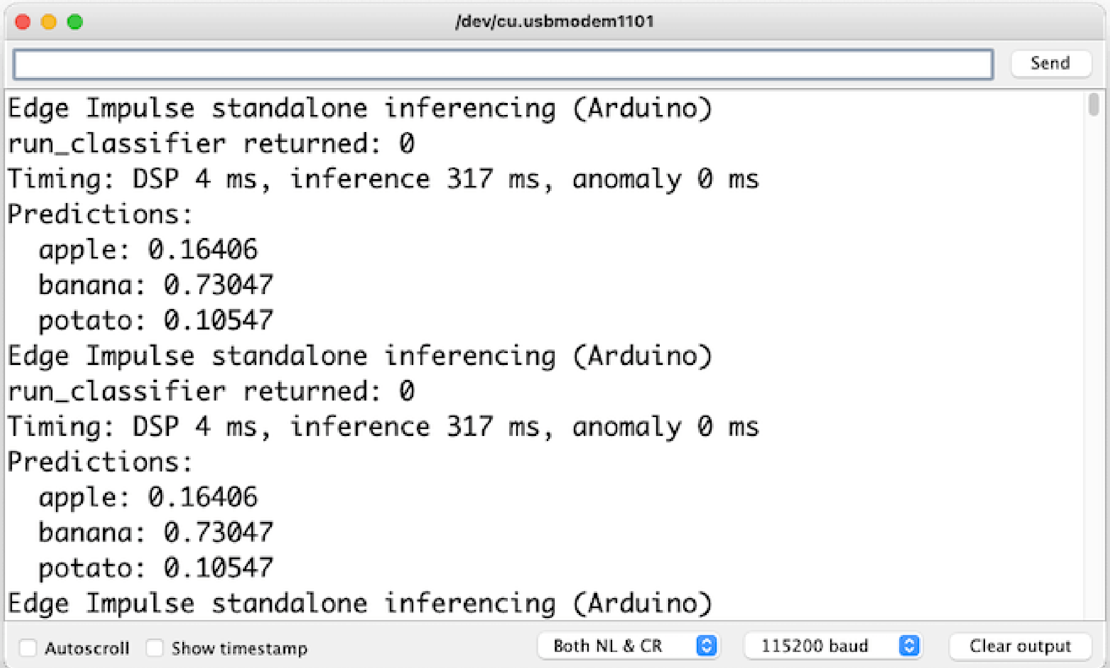
If you compile the code and get an error, it will be necessary to fix this. EI recommends switching off ESP NN acceleration. To do that, locate `ei_classifier_config.h` in exported Arduino library folder: `/scr/edge-impulse-sdk/classifier/:`



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



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

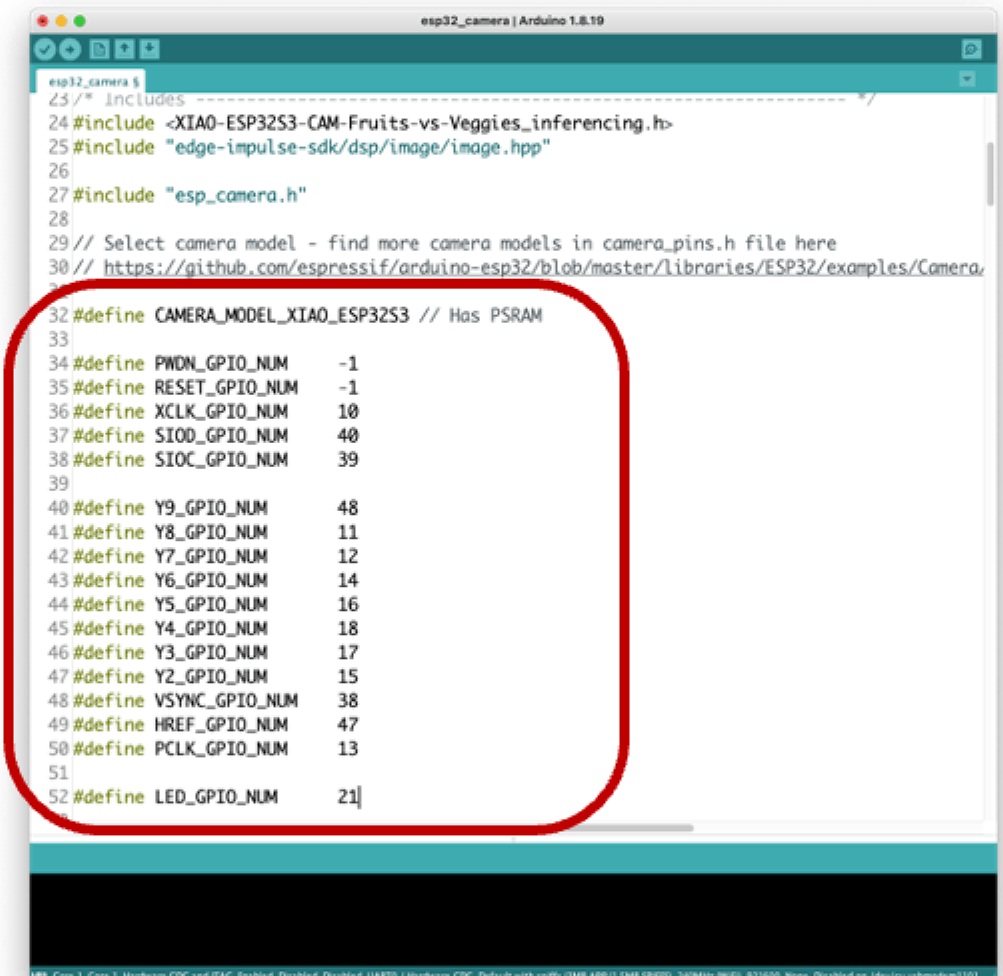


Great news! Our device handles an inference, discovering that the input image is a banana. Also, note that the inference time was around 317ms, resulting in a maximum of 3 fps if you tried to classify images from a video. It is a better result than the ESP32 CAM (525ms of latency).

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:



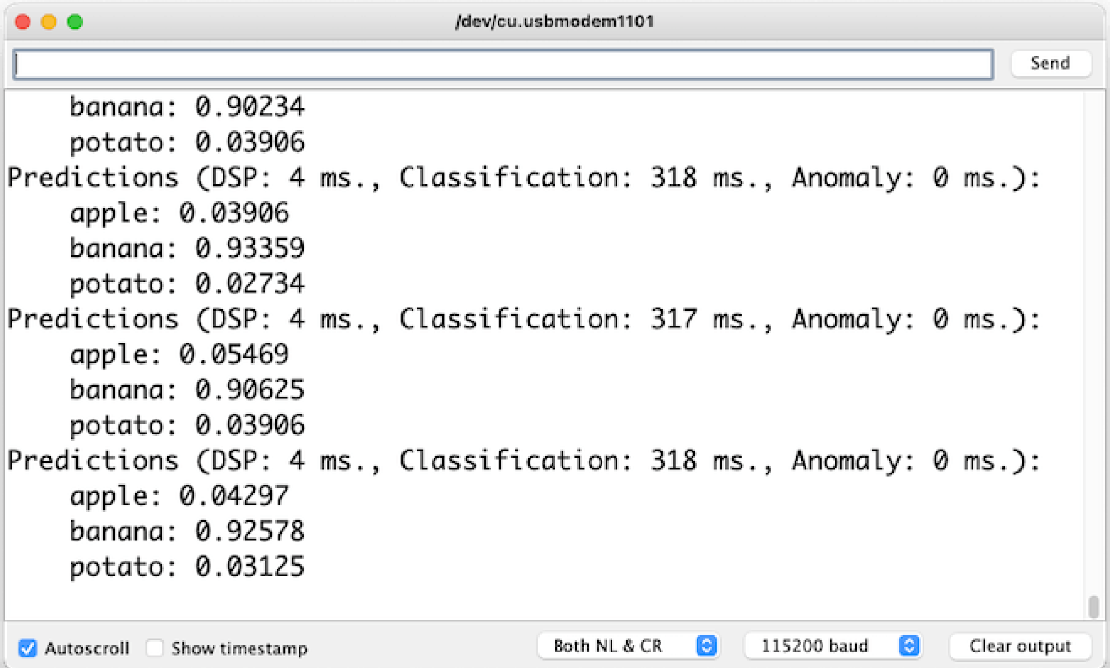
The modified sketch can be downloaded from GitHub: xiao_esp32s3_camera.

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

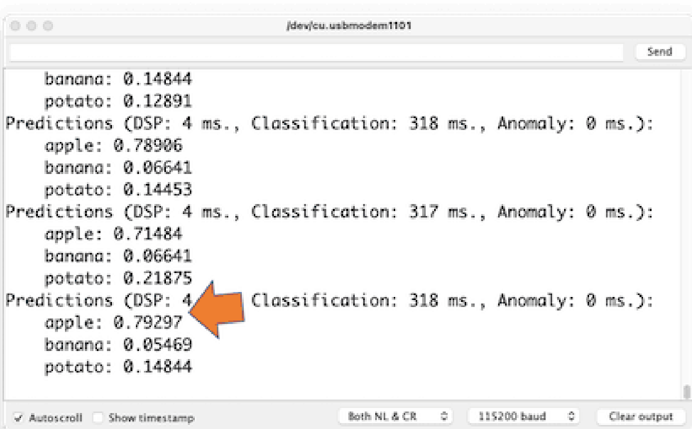
Upload the code to your XIAO ESP32S3 Sense, and you should be OK to start classifying your fruits and vegetables! You can check the result on Serial Monitor.

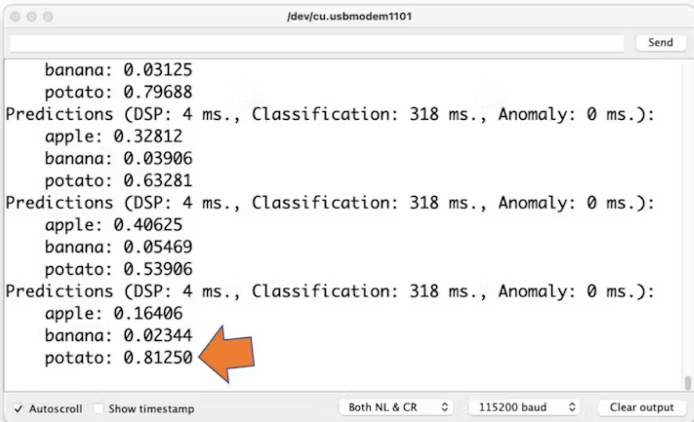
Testing the Model (Inference)

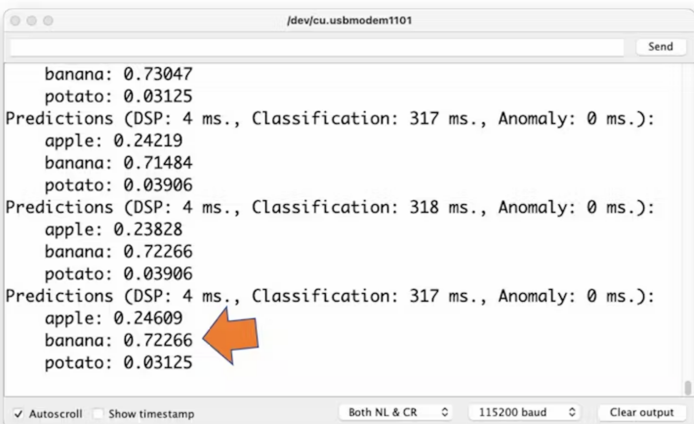
Getting a photo with the camera, the classification result will appear on the Serial Monitor:


/dev/cu.usbmodem1101
Send
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:

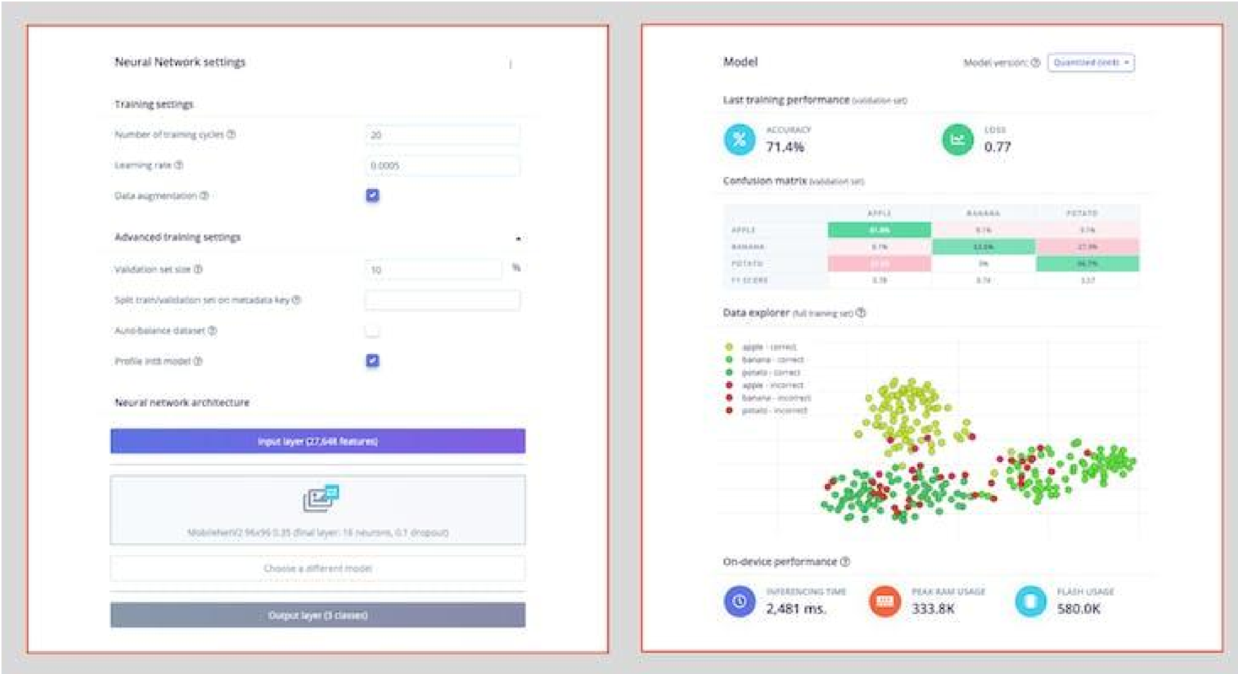

/dev/cu.usbmodem1101
Send
banana: 0.14844
potato: 0.12891
Predictions (DSP: 4 ms., Classification: 318 ms., Anomaly: 0 ms.):
apple: 0.78906
banana: 0.06641
potato: 0.14453
Predictions (DSP: 4 ms., Classification: 317 ms., Anomaly: 0 ms.):
apple: 0.71484
banana: 0.06641
potato: 0.21875
Predictions (DSP: 4
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





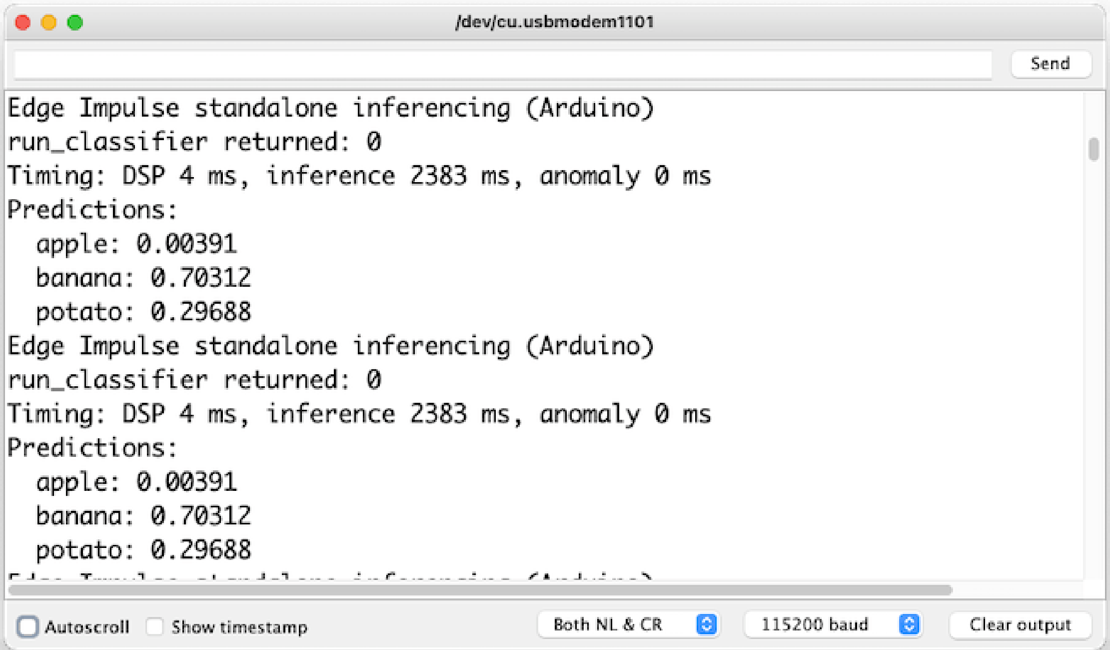
Testing with a bigger model

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.

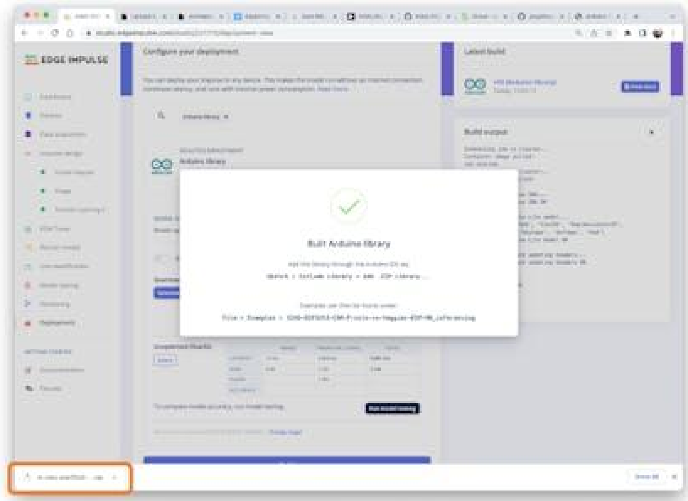


The result is YES! Memory is not an issue here; latency is! See that with a real test, the XIAO took almost 2.5s to perform the inference (compared with the previous 318ms).

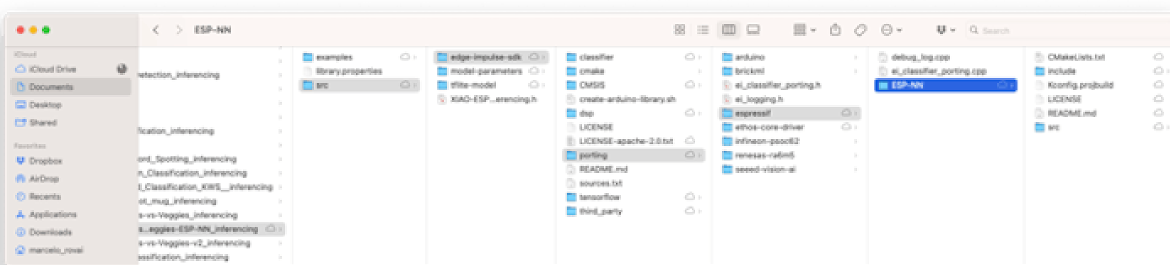
Optional use of ESP-NN acceleration

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.

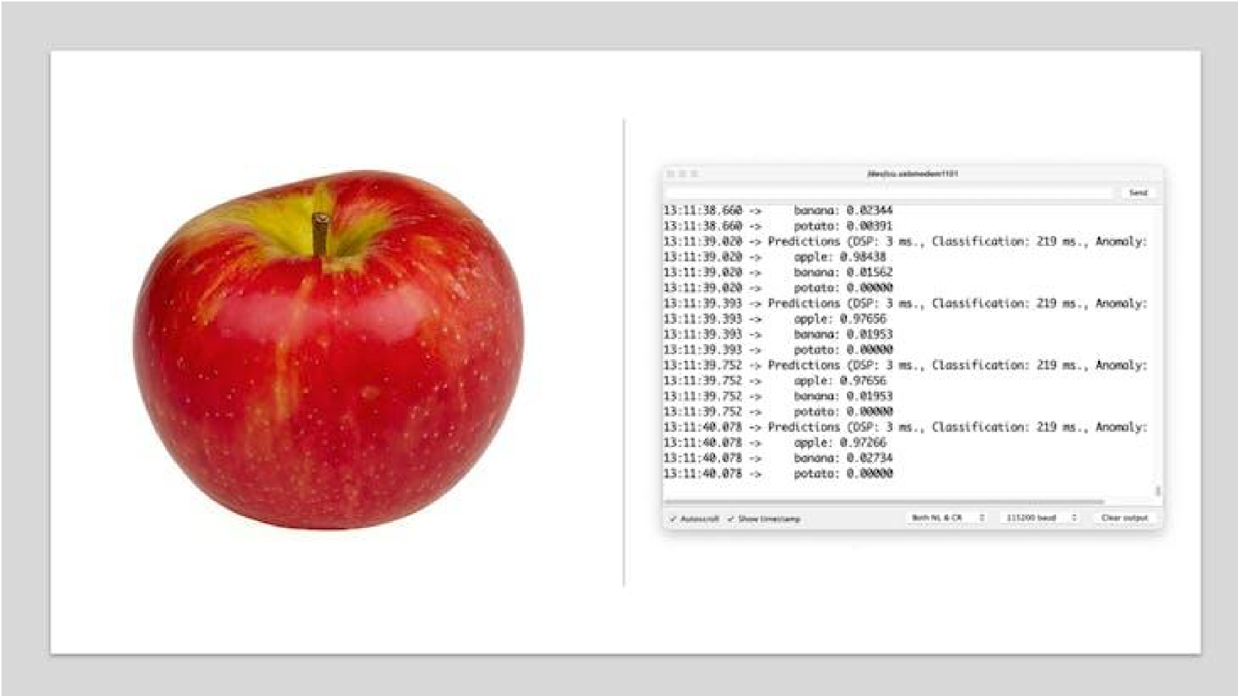


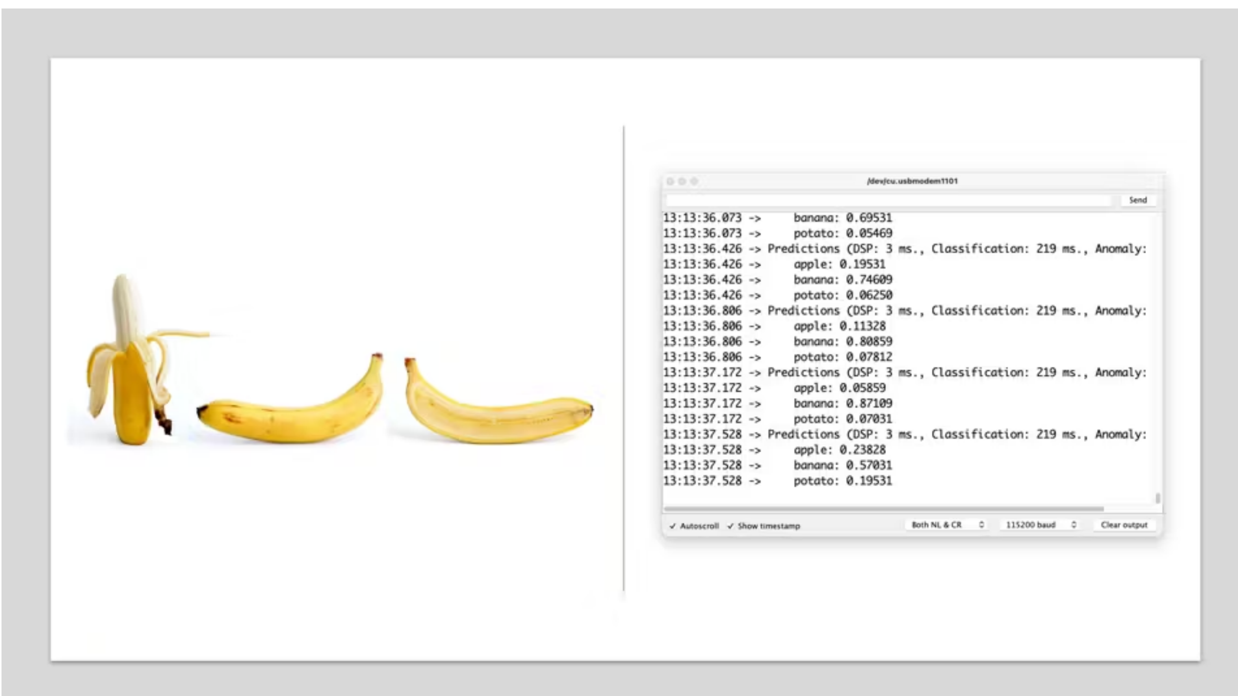
Meanwhile, you can download a preliminary version from the project GitHub, unzip it, and replace the ESP NN folder with it under `src/edge-impulse-sdk/porting/espressif/ESP-NN`, in your Arduino library folder.



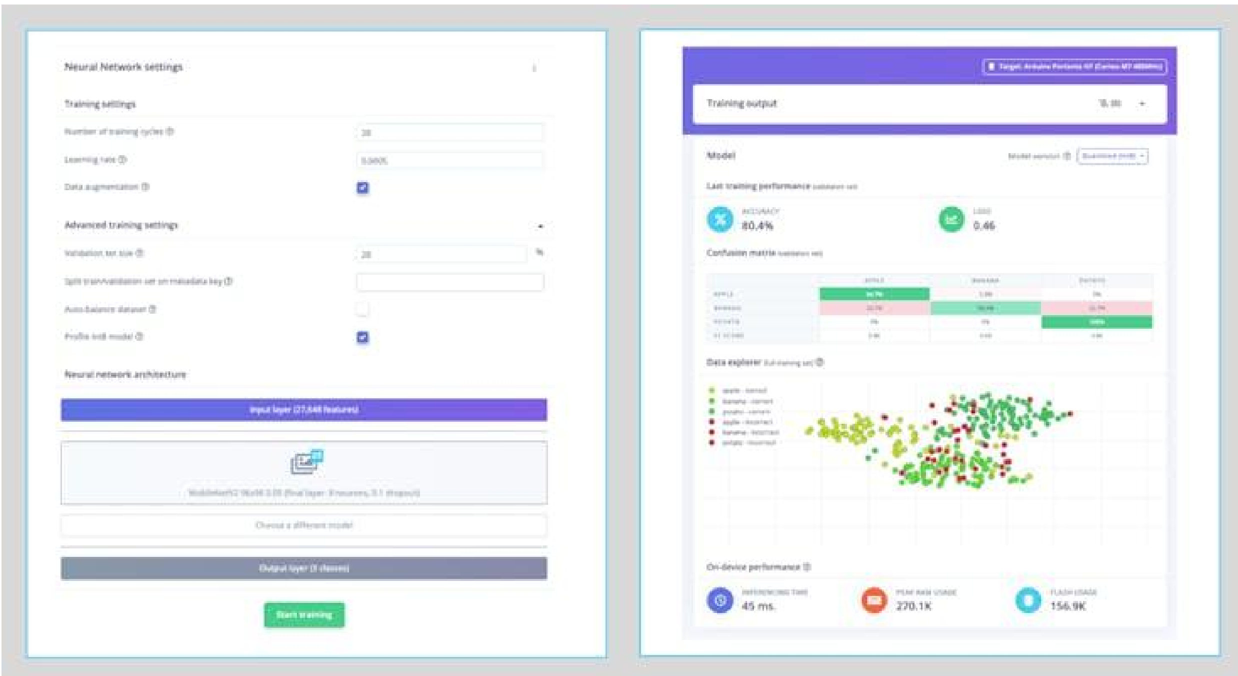
Then compile the sketch. Restarting the IDE after replacing the folder might be helpful.

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!


/dev/cu.usbmodem1101
Send
13:11:38.660 -> banana: 0.02344
13:11:38.660 -> potato: 0.00391
13:11:39.020 -> Predictions (DSP: 3 ms., Classification: 219 ms., Anomaly:
13:11:39.020 -> apple: 0.98438
13:11:39.020 -> banana: 0.01562
13:11:39.020 -> potato: 0.00000
13:11:39.393 -> Predictions (DSP: 3 ms., Classification: 219 ms., Anomaly:
13:11:39.393 -> apple: 0.97656
13:11:39.393 -> banana: 0.01953
13:11:39.393 -> potato: 0.00000
13:11:39.752 -> Predictions (DSP: 3 ms., Classification: 219 ms., Anomaly:
13:11:39.752 -> apple: 0.97656
13:11:39.752 -> banana: 0.01953
13:11:39.752 -> potato: 0.00000
13:11:40.078 -> Predictions (DSP: 3 ms., Classification: 219 ms., Anomaly:
13:11:40.078 -> apple: 0.97266
13:11:40.078 -> banana: 0.02734
13:11:40.078 -> potato: 0.00000
Autoscroll
Show timestamp
Both NL & CR
115200 baud
Clear output

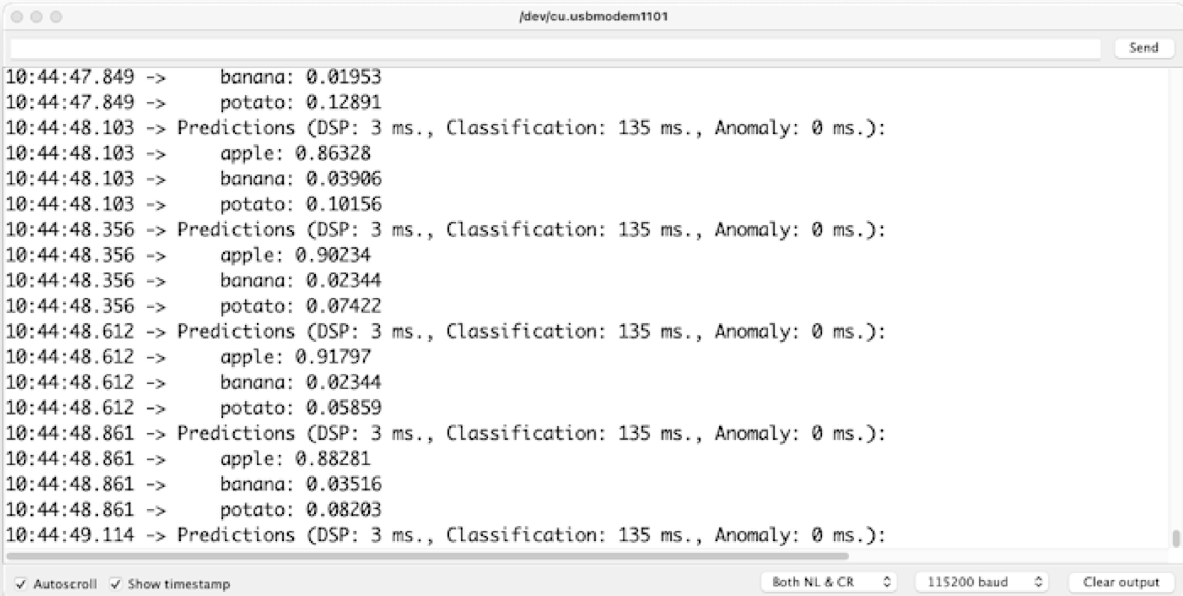

/dev/cu.usbmodem1101
Send
13:13:36.073 -> banana: 0.69531
13:13:36.073 -> potato: 0.05469
13:13:36.426 -> Predictions (DSP: 3 ms., Classification: 219 ms., Anomaly:
13:13:36.426 -> apple: 0.19531
13:13:36.426 -> banana: 0.74609
13:13:36.426 -> potato: 0.06250
13:13:36.806 -> Predictions (DSP: 3 ms., Classification: 219 ms., Anomaly:
13:13:36.806 -> apple: 0.11328
13:13:36.806 -> banana: 0.80859
13:13:36.806 -> potato: 0.07812
13:13:37.172 -> Predictions (DSP: 3 ms., Classification: 219 ms., Anomaly:
13:13:37.172 -> apple: 0.05859
13:13:37.172 -> banana: 0.87109
13:13:37.172 -> potato: 0.07031
13:13:37.528 -> Predictions (DSP: 3 ms., Classification: 219 ms., Anomaly:
13:13:37.528 -> apple: 0.23828
13:13:37.528 -> banana: 0.57031
13:13:37.528 -> potato: 0.19531
Autoscroll
Show timestamp
Both NL & CR
115200 baud
Clear output

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 (ou 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):



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.

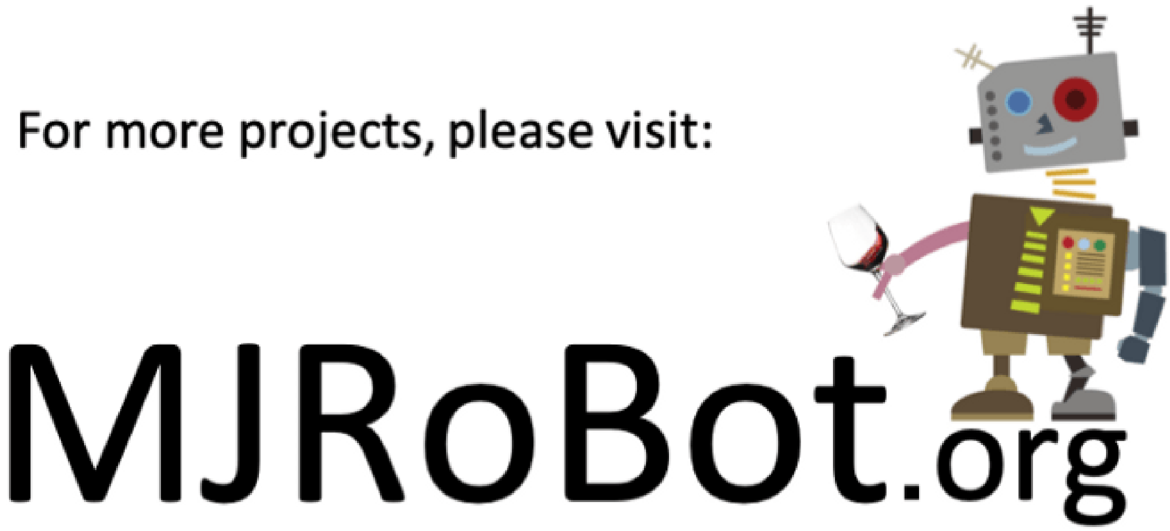
Knowing more

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!



link: MJRoBot.org

Greetings from the south of the world!

See you at my next project!

Thank you

Marcelo