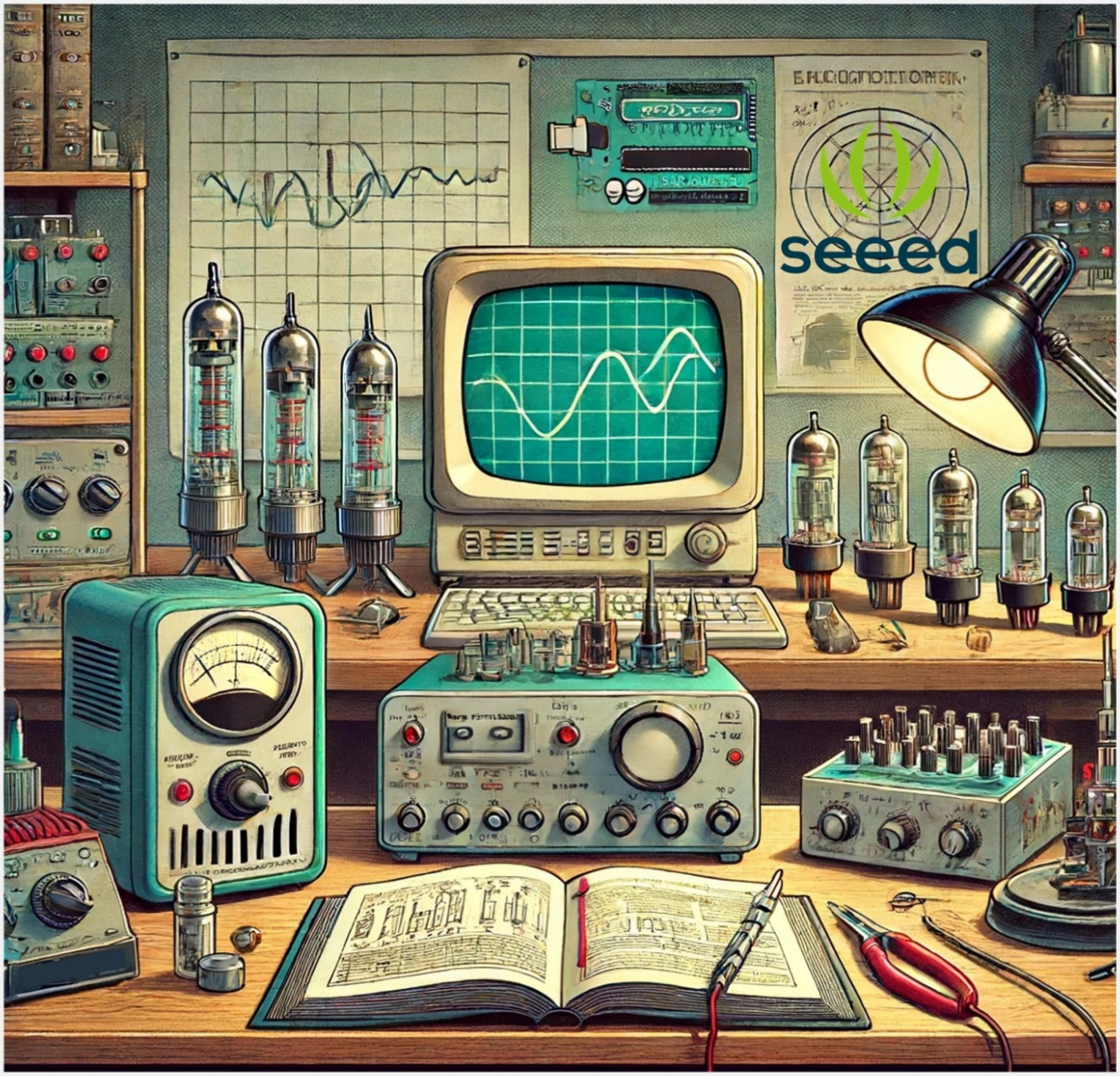
TinyML Made Easy

Hands-On with the XIAO ESP32S3 Sense



UNIFEI

Marcelo Rovai

June, 2024

TinyML Made Easy

Hands-On with the XIAO ESP32S3 Sense

Marcelo Rovai

2024-06-16

Preface

Finding the right info used to be the big issue for people involved in technical projects. Nowadays, there is a deluge of information, but it is not easy to determine what can and what cannot be trusted. A good pointer is to look for the credentials and the affiliation of the author of a document or that of the associated institutions. Since 1992 EsLaRed has been providing training in different aspects of Information Technologies in Latin America and the Caribbean, always striving to provide accurate and timely training materials, which have evolved accordingly to shifts in the technology focus. IoT and Machine Learning are poised to play a pivotal role, so the work of Professor Marcelo Rovai addressing Tiny Machine Learning is both timely and authoritative, in this rapidly changing field.

TinyML Made Easy series is exactly what the title means. Written in a clear and concise style, with emphasis on practical applications and examples, drawing from many years of experience and overwhelming enthusiasm in sharing his knowledge, there is no doubt that Marcelo's work is a very significant contribution to this very interesting field. The knowledge acquired from the exercises will enable the readers to undertake other projects that might interest them.

Ermanno Pietrosemoli, President Fundación Escuela Latinoamericana de Redes

November 2023.

Acknowledgments

We extend our deepest gratitude to the entire TinyML4D Academic Network, comprised of distinguished professors, researchers, and professionals. Notable contributions from Marco Zennaro, Ermanno Petrosemoli, Brian Plancher, José Alberto Ferreira, Jesus Lopez, Diego Mendez, Shawn Hymel, Dan Situnayake, Pete Warden, and Laurence Moroney have been instrumental in advancing our understanding of Embedded Machine Learning (TinyML).

Special commendation is reserved for Professor Vijay Janapa Reddi of Harvard University. His steadfast belief in the transformative potential of open-source communities, coupled with his invaluable guidance and teachings, has served as a beacon for our efforts from the very beginning.

Acknowledging these individuals, we pay tribute to the collective wisdom and dedication that have enriched this field and our work.

Illustrative images on the e-book and chapter's covers generated by OpenAI's DALL-E via ChatGPT

Introduction

Microcontrollers (MCUs) are not just cheap electronic components, but they are the backbone of our modern world. With just a few kilobytes of RAM, they are designed to consume small amounts of power. Today, MCUs can be found embedded in all residential, medical, automotive, and industrial devices. Over 40 billion microcontrollers are estimated to be marketed annually, and hundreds of billions are currently in service. Yet, these devices often go unnoticed, as they are used to replace functionalities in older electromechanical systems in cars, washing machines, or remote controls. Understanding the potential of these microcontrollers is key to unlocking the future of technology.

More recently, with the era of IoT (Internet of Things), a significant part of these MCUs is generating “quintillions” of data, which, in their majority, are not used due to the high cost and complexity of their data transmission (bandwidth and latency).

On the other hand, in the last decades, we have witnessed the development of Machine Learning models (sub-area of Artificial Intelligence) trained with “tons” of data and powerful mainframes. But now, it suddenly becomes possible for “noisy” and complex signals, such as images, audio, or accelerometers, to extract meaning from the same through neural networks. More importantly, we can execute these models of neural networks in microcontrollers and sensors using very little energy and extract much more meaning from the data generated by these sensors, which we are currently ignoring.

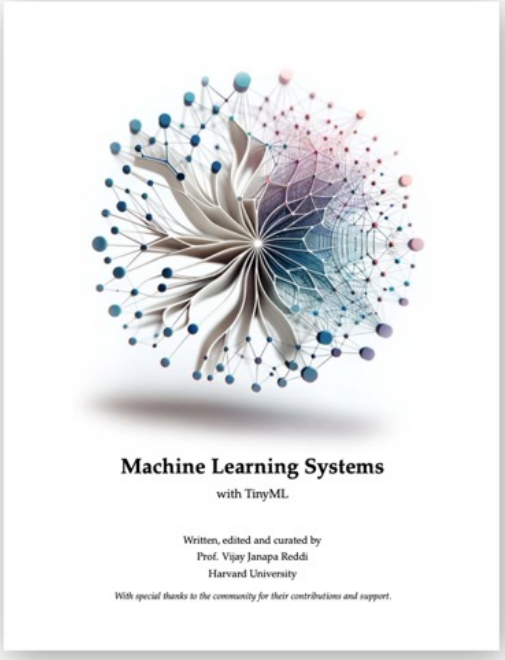
> TinyML, a new area of applied AI, allows for extracting “machine intelligence” from the physical world (where the data is generated).

TinyML Made Ease is a foundational text designed to facilitate the understanding and application of Embedded Machine Learning, or TinyML. In an era where embedded devices boast ultra-low power consumption—measured in mere milliwatts—and machine learning

frameworks such as TensorFlow Lite for Microcontrollers (TF Lite Micro) become increasingly tailored for embedded applications, the intersection of AI and IoT is rapidly expanding. This book demystifies the integration of AI capabilities into these devices, making it accessible and empowering for all, paving the way for a wide-reaching adoption of AI-enabled IoT, or “AioT.”

About this Book

This book is part of the open book Machine Learning Systems, which we invite you to read.



TinyML Made Easy, an open-access eBook and part of the open book Machine Learning Systems, is an accessible resource for enthusiasts, professionals, and engineering students embarking on the transformative path of embedded machine learning (TinyML). This book offers a systematic introduction to integrating ML algorithms with microcontrollers, focusing on practicality and hands-on experiences.

Students are introduced to core concepts of TinyML through the setup and programming of the **Seeed Studio XIAO ESP32-S3**, a state-of-the-art microcontroller designed for ML applications. The book breaks down complex ideas into understandable segments, ensuring foundational knowledge is built from the ground up.

True to the engineering ethos of ‘learning by doing,’ each chapter focuses on a project that challenges students to apply their knowledge in real-world scenarios. The projects are designed to reinforce learning and stimulate innovation, covering:

- Microcontroller setup and sensor tests,
- Image classification,
- Dataset labeling and Object detection,
- Audio processing and keyword spotting,
- Time-series data and Digital Signal Processing,
- Motion classification and Anomaly detection.

TinyML Made Easy provides detailed walkthroughs of industry-standard tools such as **Arduino IDE**, **Seeed SenseCraft**, and **Edge Impulse Studio**. Students will gain hands-on experience with data collection, pre-processing, model training, and deployment, equipping them with the technical skills sought in the embedded systems industry.

As TinyML is poised to be a driving force in the evolution of IoT and smart devices, students will emerge from this book with an introductory theoretical understanding and the practical skills necessary to contribute to and shape the future of technology.

With its straightforward language, clarity, and focus on experiential learning, **TinyML Made Easy** is more than just a book—it’s a comprehensive toolkit for anyone ready to delve into the world of embedded machine learning. It empowers them to move beyond the classroom and into the lab or field, and they are well-prepared to tackle the challenges and opportunities TinyML presents.

Supplementary Online Resources

Chapter 1 - Setup XIAO ESP32S3 Sense

- Arduino Codes

Chapter 2 - Image Classification

- Arduino Codes
- Dataset
- Edge Impulse Project

Chapter 3 - Object Detection

- Edge Impulse Project

Chapter 4 - Audio Feature Engineering

- Audio_Data_Analysis Colab Notebook

Chapter 5 - Keyword Spotting (KWS)

- Arduino Codes
- Subset of Google Speech Commands Dataset
- KWS_MFCC_Analysis Colab Notebook
- KWS_CNN_training Colab Notebook
- Arduino post-processing code
- Edge Impulse Project

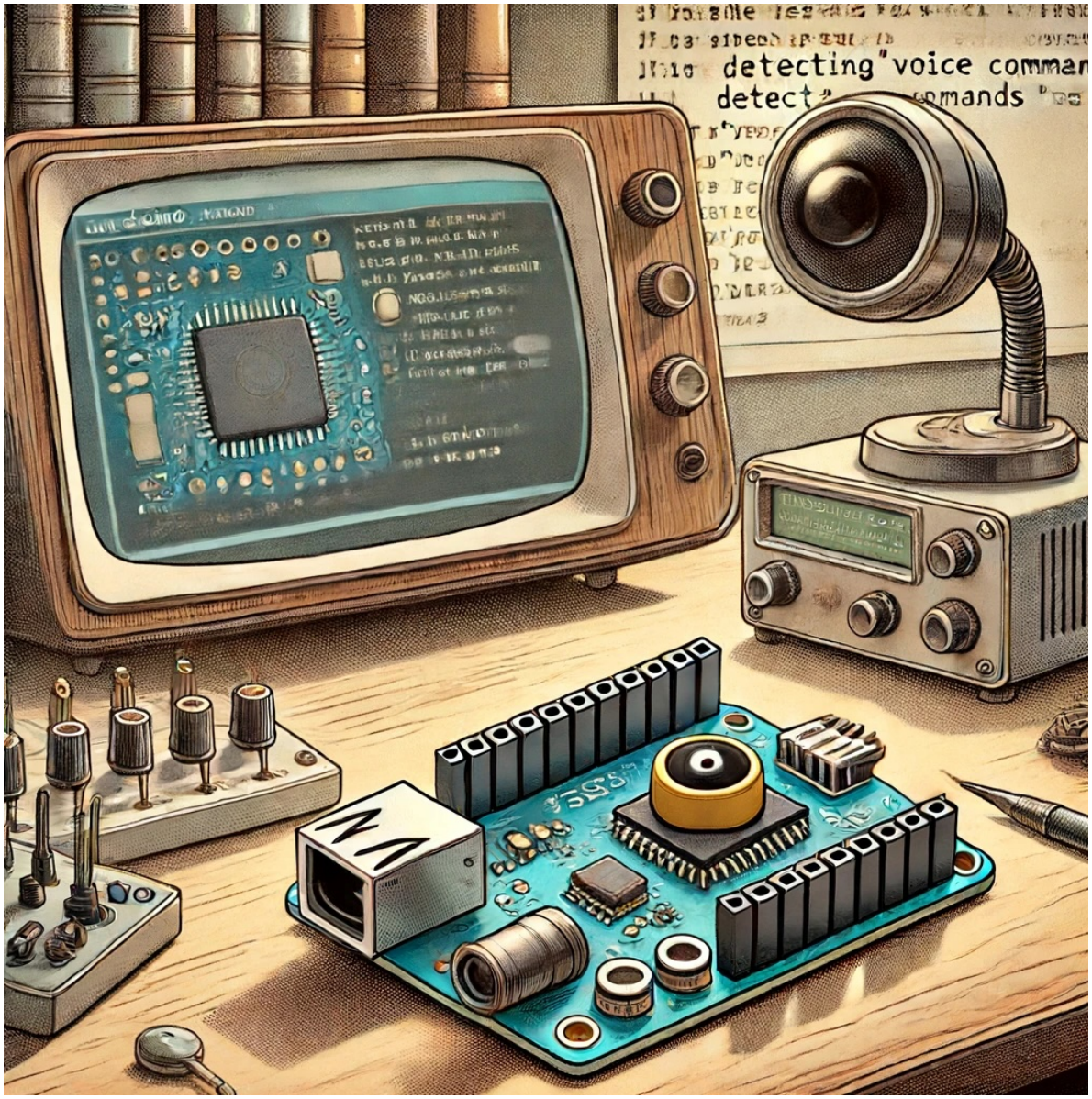
Chapter 6 - DSP Spectral Features

- DSP - Spectral Features Colab Notebook

Chapter 7 - Motion Classification and Anomaly Detection

- Arduino Codes
- Edge_Impulse_Spectral_Features_Block Colab Notebook
- Edge Impulse Project

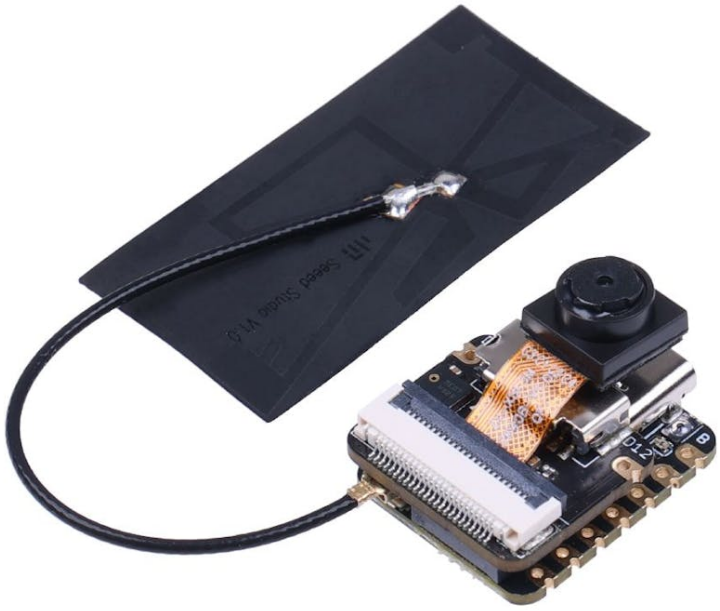
1 XIAO ESP32S3 Sense Setup



DALL·E prompt - 1950s cartoon-style drawing of a XIAO ESP32S3 board with a distincti camera module, as shown in the image provided. The board is placed on a classic lab ta with various sensors, including a microphone. Behind the board, a vintage computer scre displays the Arduino IDE in muted colors, with code focusing on LED pin setups and machine learning inference for voice commands. The Serial Monitor on the IDE showcas outputs detecting voice commands like 'yes' and 'no'. The scene merges the retro charm mid-century labs with modern electronics.

1.1 Introduction

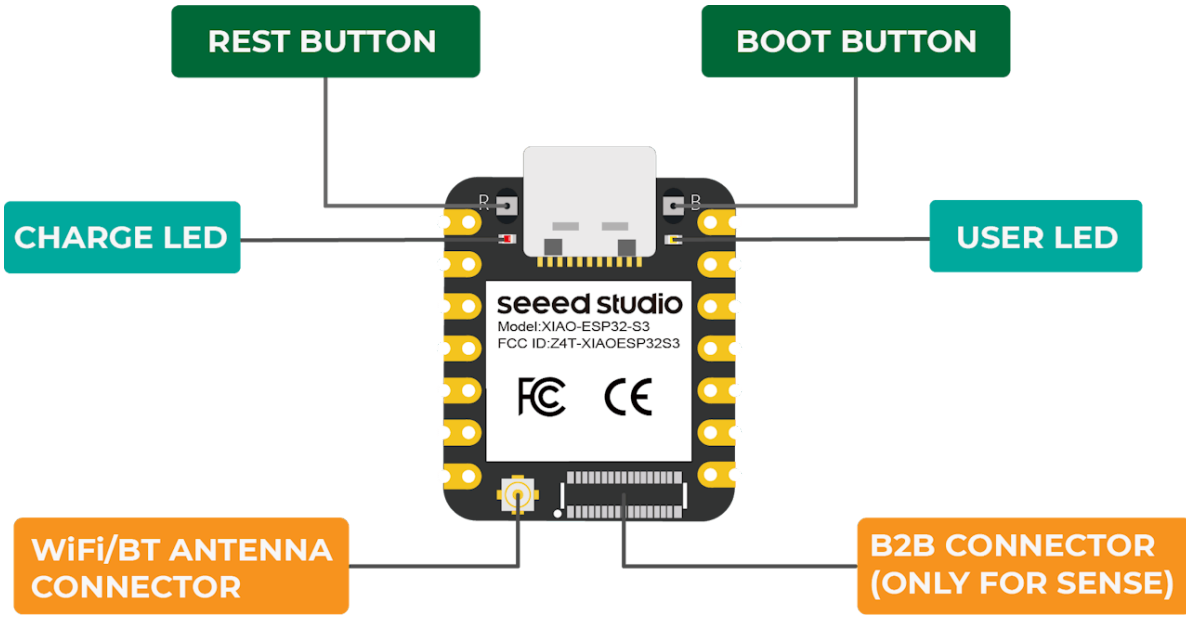
The XIAO ESP32S3 Sense is Seeed Studio's affordable development board, which integrates a camera sensor, digital microphone, and SD card support. Combining embedded ML computing power and photography capability, this development board is a great tool to start with TinyML (intelligent voice and vision AI).



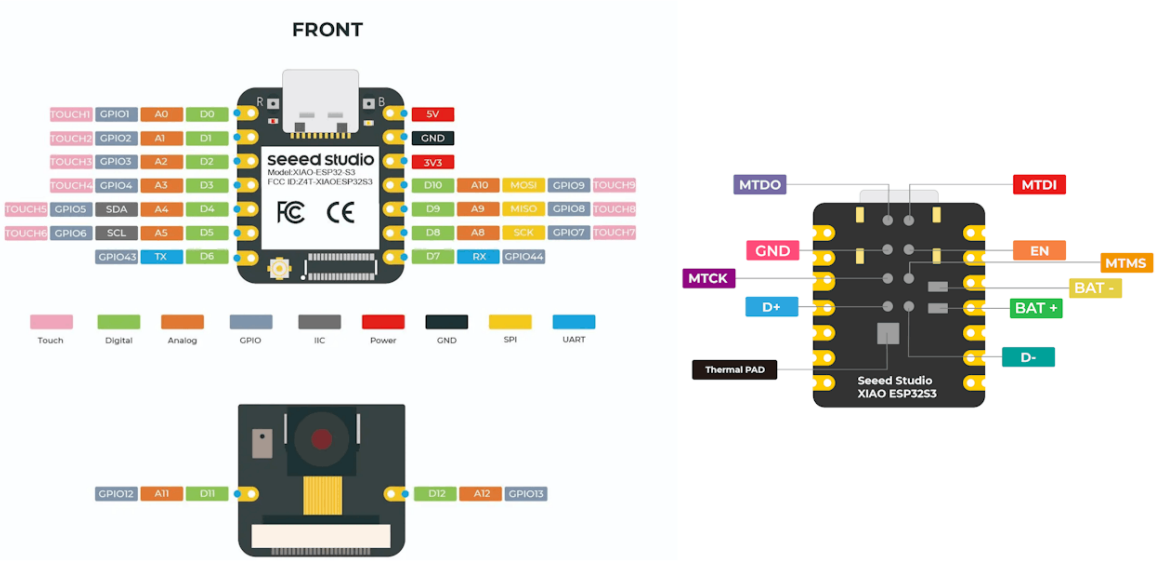
XIAO ESP32S3 Sense Main Features

- **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 offers four power consumption models, 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:



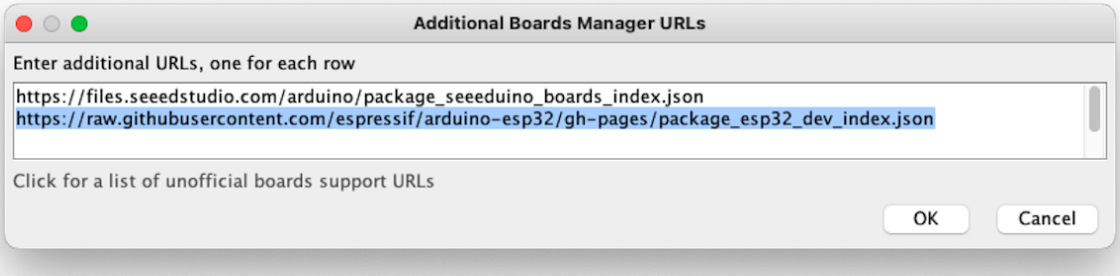
For more details, please refer to the Seeed Studio WiKi page: https://wiki.seeedstudio.com/xiao_esp32s3_getting_started/

1.2 Installing the XIAO ESP32S3 Sense on Arduino IDE

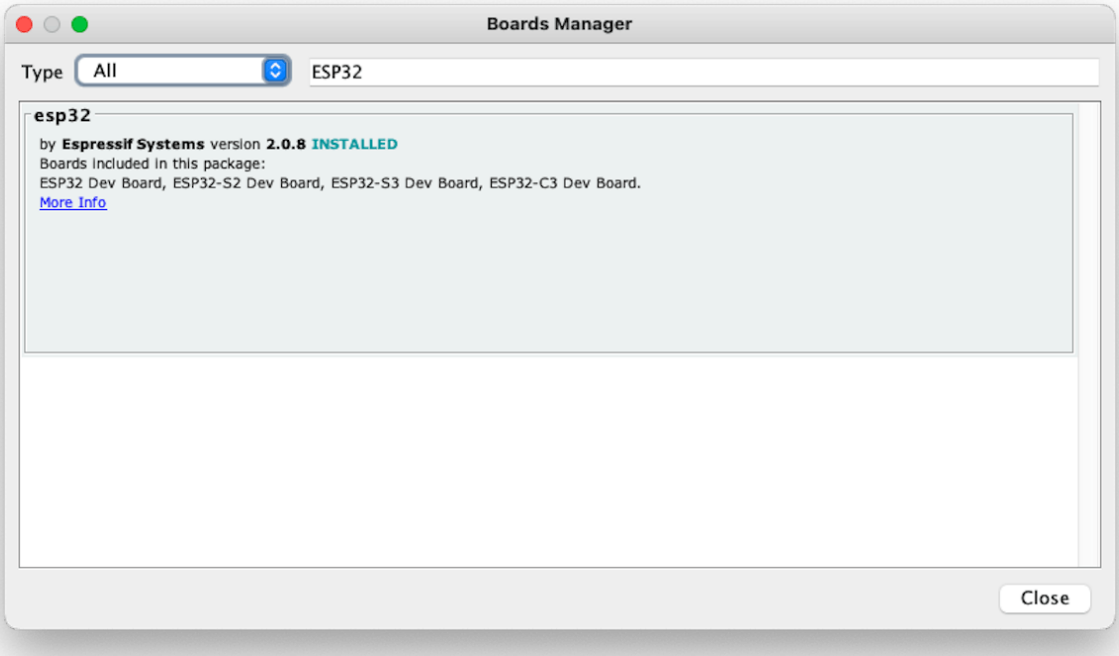
On Arduino IDE, navigate to **File > Preferences**, and fill in the URL:

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

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



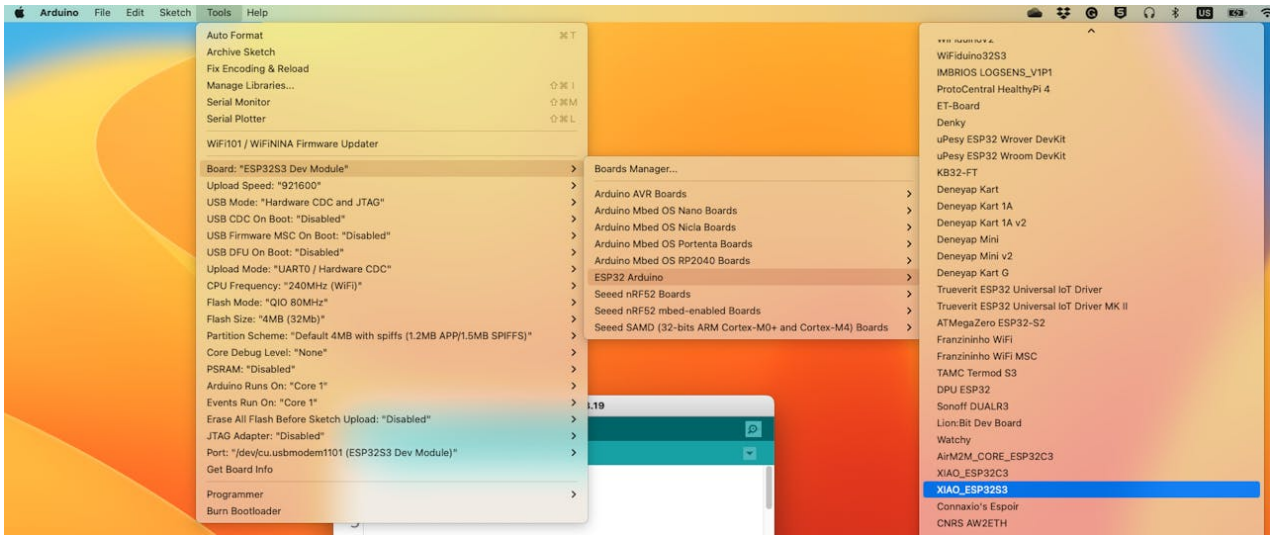
Next, open boards manager. Go to **Tools** > **Board** > **Boards Manager…** and enter with *esp32.* Select and install the most updated and stable package (avoid *alpha* versions) :



⚠ Attention

Alpha versions (for example, 3.x-alpha) do not work correctly with the XIAO and Edge Impulse. Use the last stable version (for example, 2.0.11) instead.

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



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

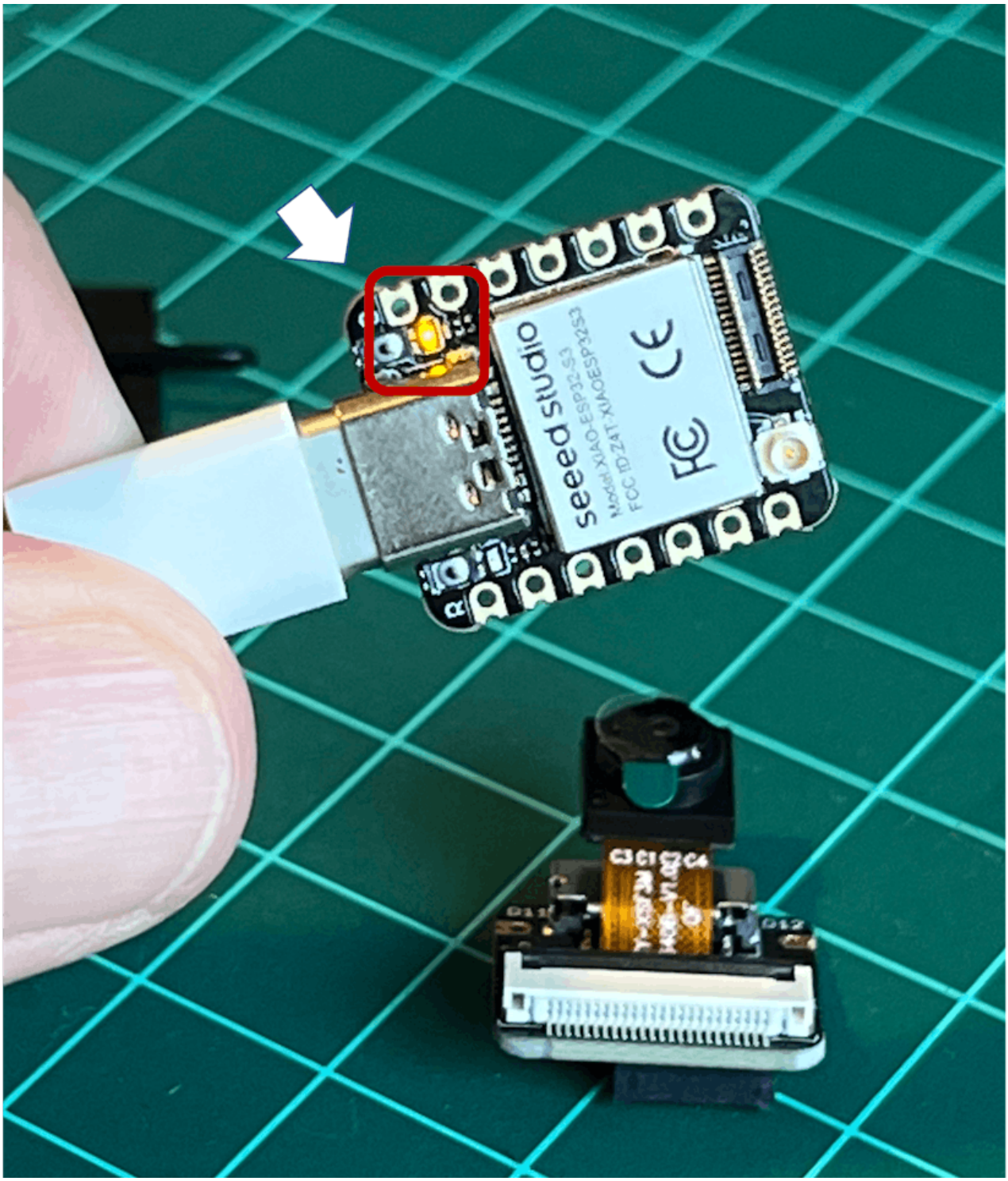
That is it! The device should be OK. Let's do some tests.

1.3 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 is (using the `LED_BUILTIN` Arduino constant) or by changing the Blink sketch accordingly:

```
#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.



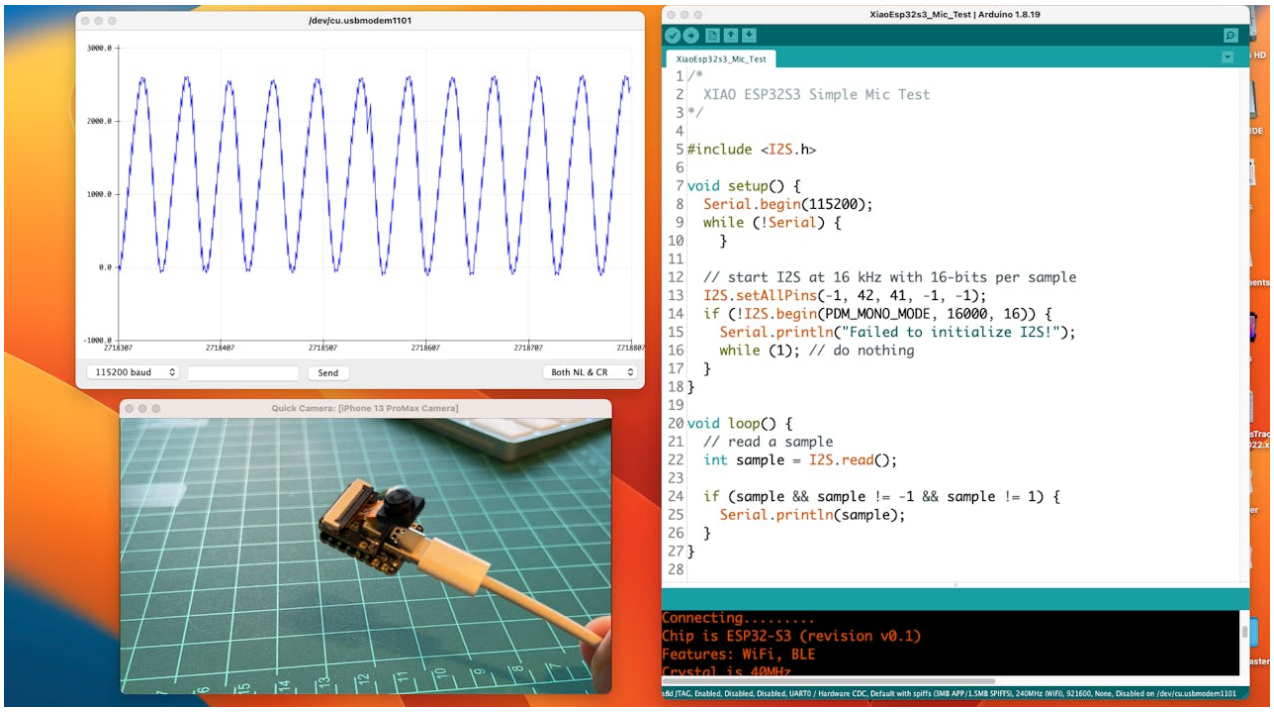
1.4 Connecting Sense module (Expansion Board)

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.

1.5 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:



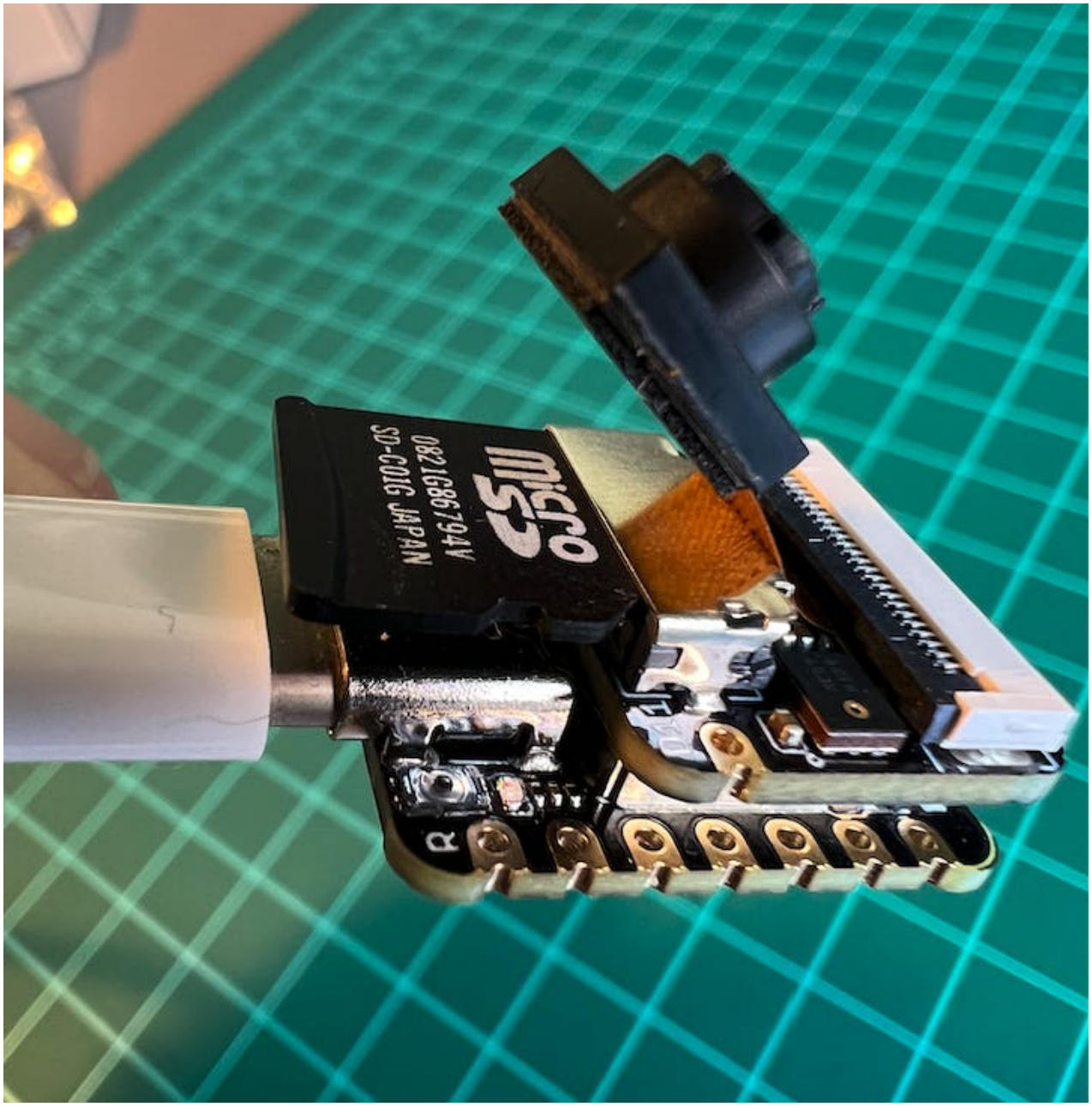
When producing sound, you can verify it on the Serial Plotter.

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

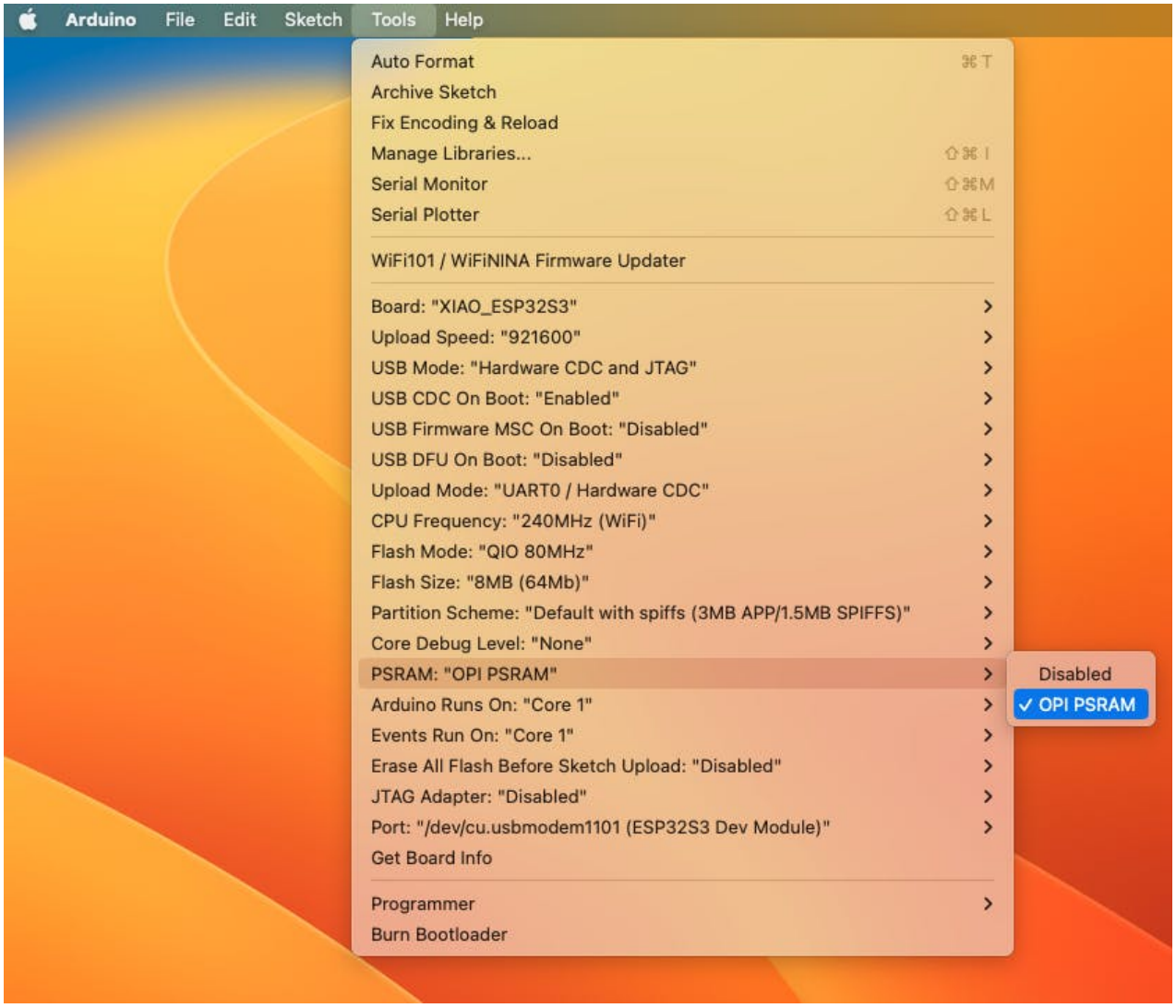
Now, the onboard SD Card reader can save .wav audio files. To do that, we need to habilitate the XIAO PSRAM.

ESP32-S3 has only a few hundred kilobytes of internal RAM on the MCU chip. This can be insufficient for some purposes, so up to 16 MB of external PSRAM (pseudo-static RAM) can be connected 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. It records for 20 seconds and saves 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!

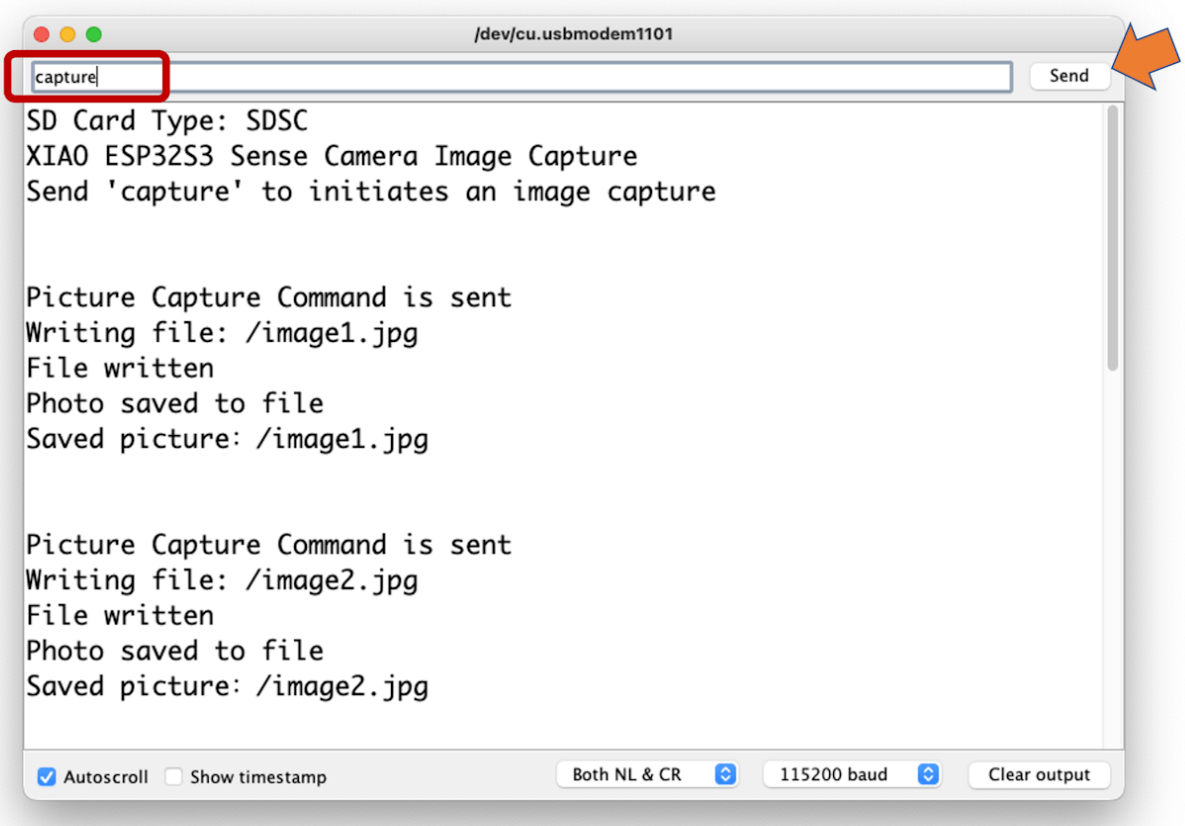
> The explanation of how the code works is beyond the scope of this tutorial, but you can find an excellent description on the wiki page.

1.6 Testing the Camera

To test 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]` are selected on Serial Monitor.



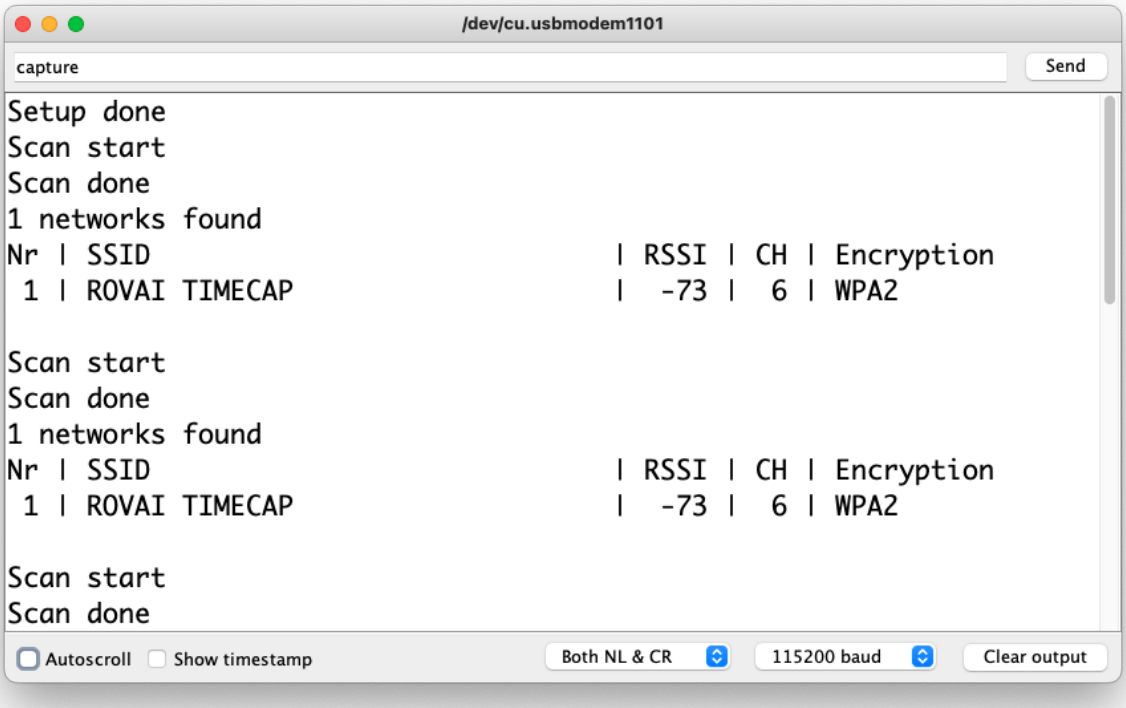
Here is an example of a taken photo:

1.7 Testing WiFi

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

Go to Arduino IDE Examples and look for **WiFI ==> WiFIScan**

You should see the Wi-Fi networks (SSIDs and RSSIs) within your device's range on the serial monitor. Here is what I got in the lab:



Simple WiFi Server (Turning LED ON/OFF)

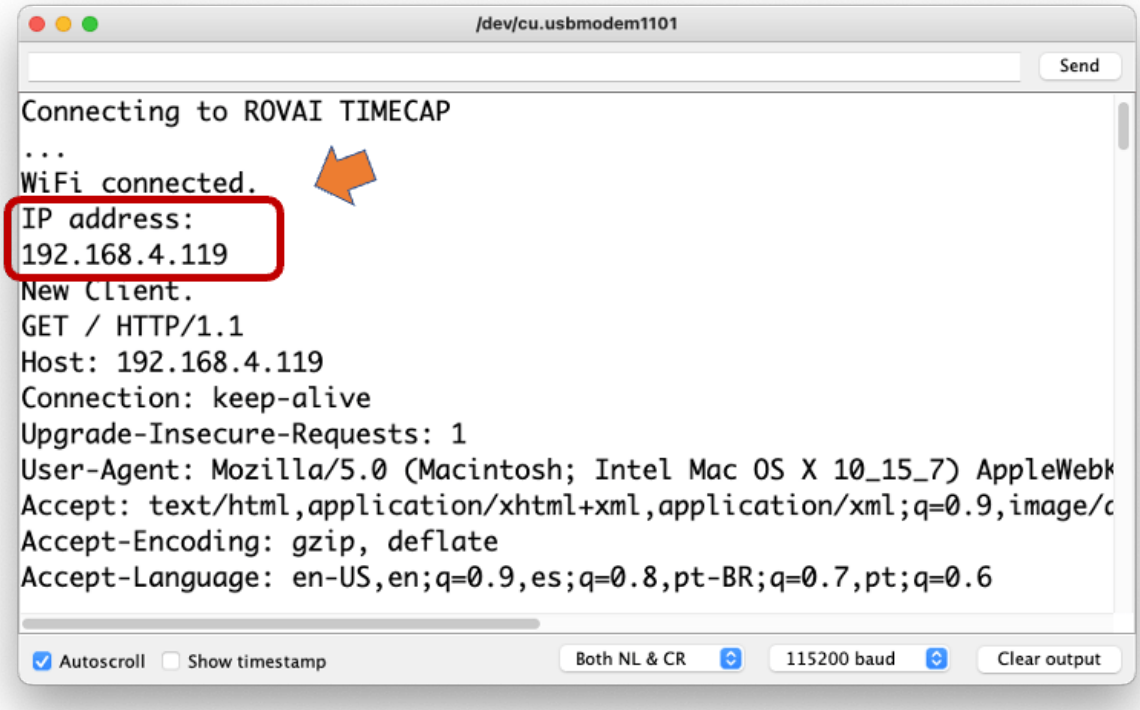
Let's test the device's capability to behave as a WiFi Server. We will host a simple page on the device that sends commands to turn the XIAO built-in LED ON and OFF.

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

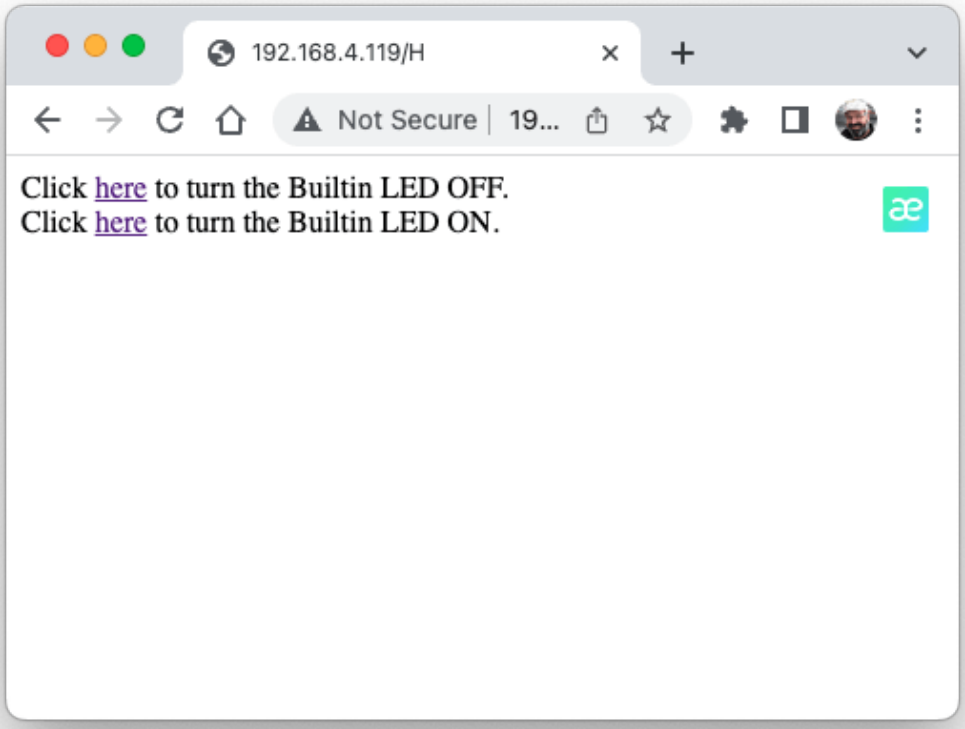
Before running the sketch, you should enter your network credentials:

```
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 the built-in LED of your XIAO ON and OFF.

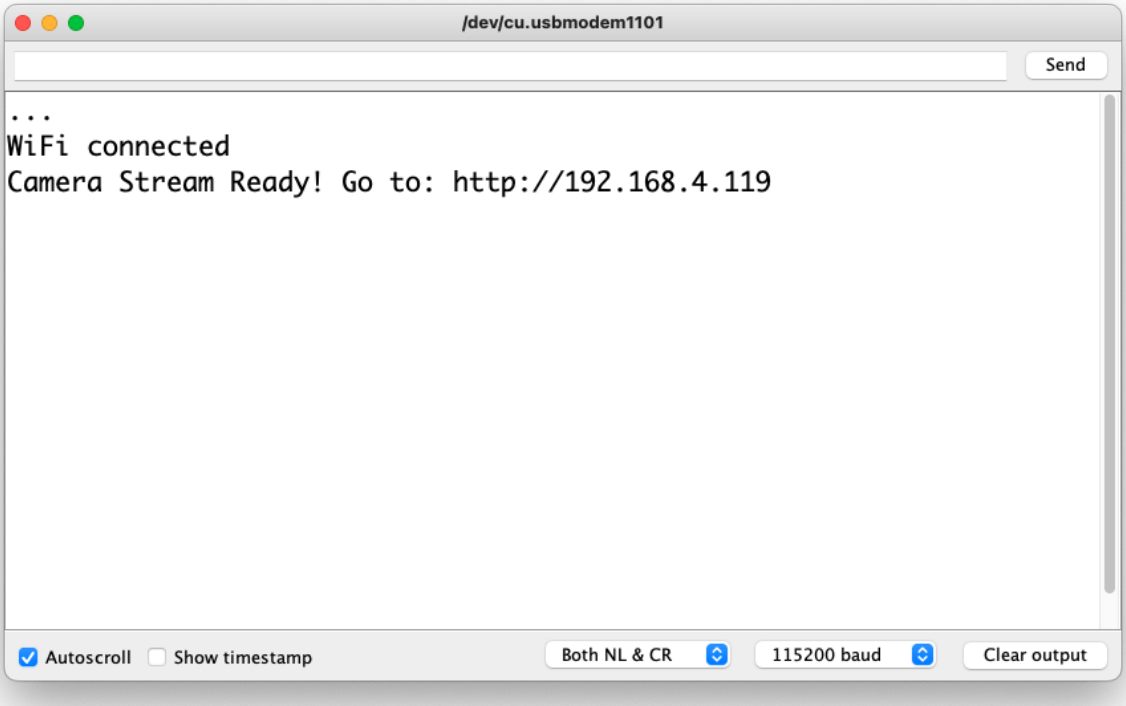
Streaming video to Web

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:

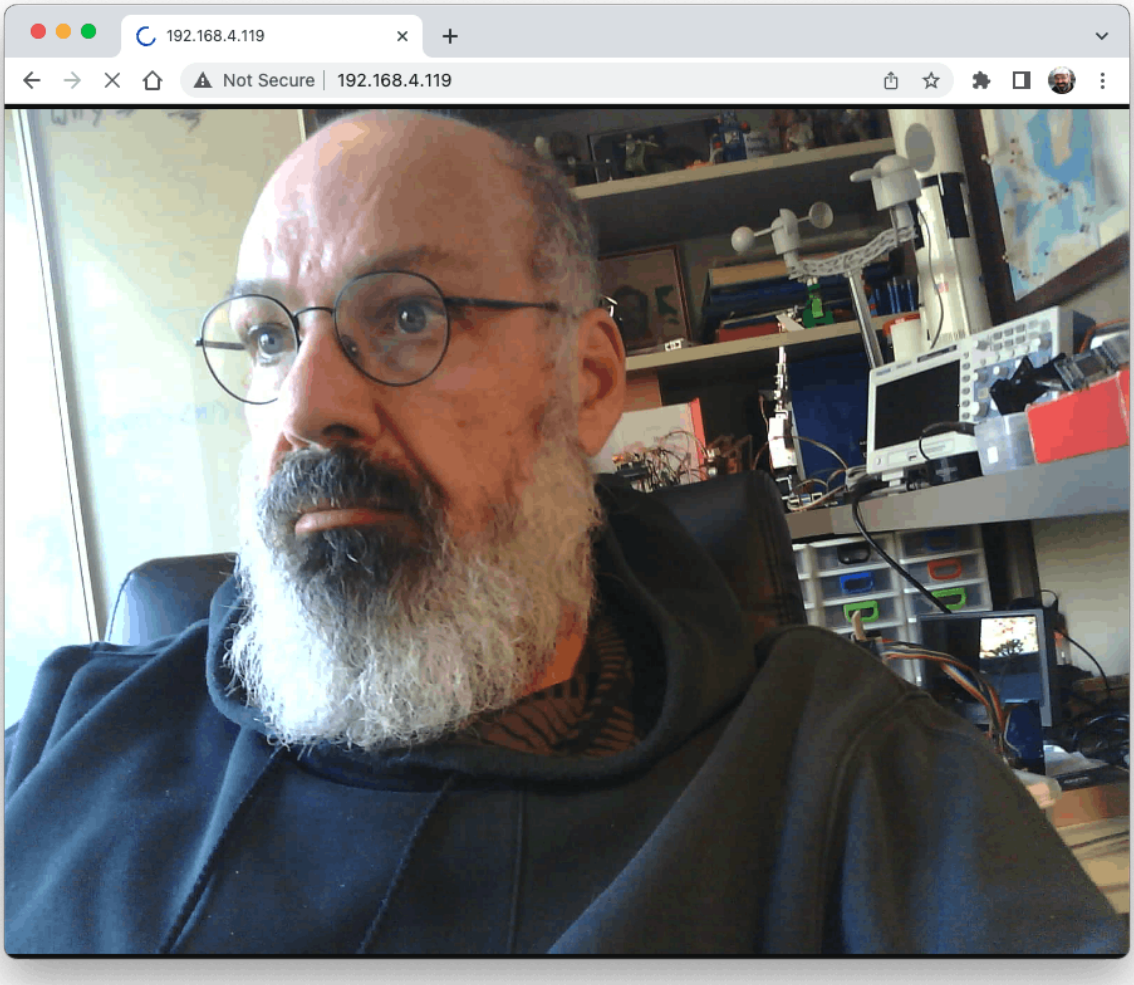
Download from GitHub the folder that contains the code: XIAO-ESP32S3-Streeming_Video.ino.

> Remember that the folder contains the.ino file and 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:

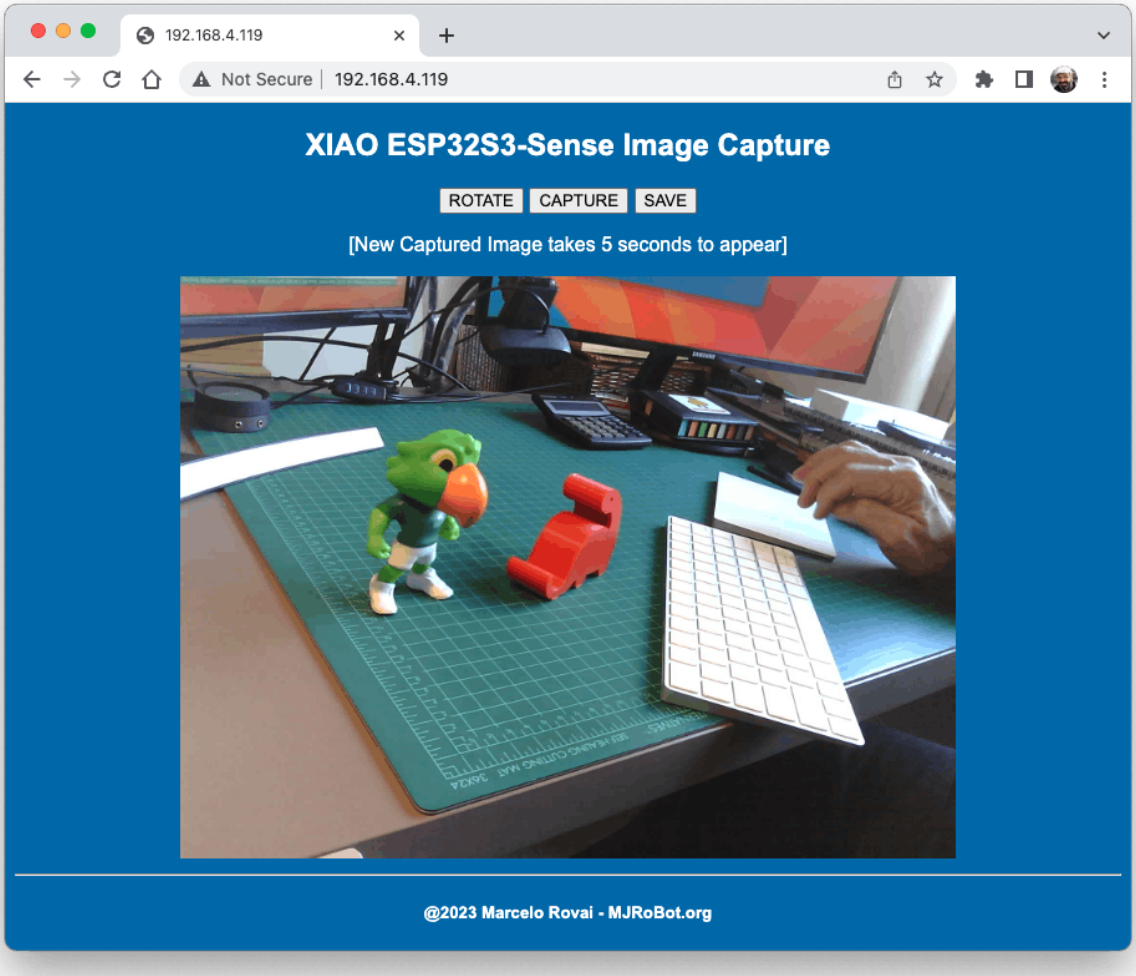


Open the page on your browser (wait a few seconds to start the streaming). That's it.



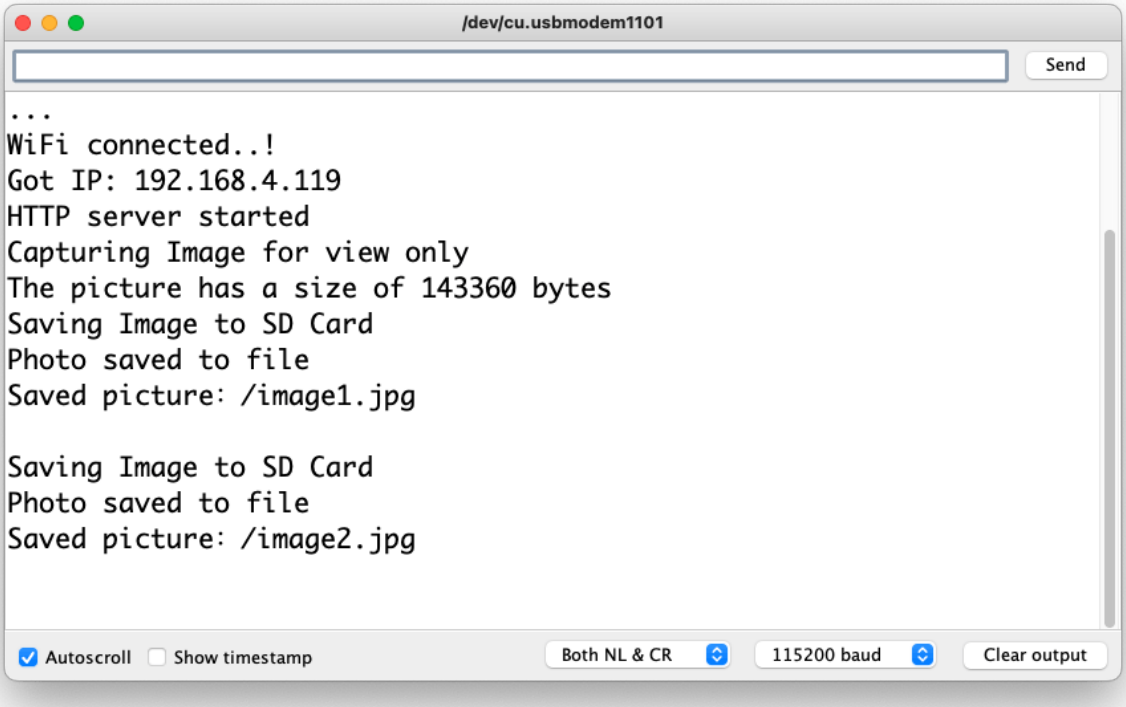
Streamlining what your camera is “seen” can be important when you position it to capture a dataset for an ML project (for example, using the code “take_phots_commands.ino”.

Of course, we can do both things simultaneously: show what the camera sees 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 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 the Serial Monitor
- The `[SAVE]` command will save an image on the SD Card and show the image on the browser.
- Saved images will follow a sequential naming (image1.jpg, image2.jpg.



This program can capture an image dataset 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.

Using the CameraWebServer

In the Arduino IDE, go to `File > Examples > ESP32 > Camera`, and select `CameraWebServer`

You also should comment on all cameras' models, except the XIAO model pins:

```
#define CAMERA_MODEL_XIAO_ESP32S3 // Has PSRAM
```

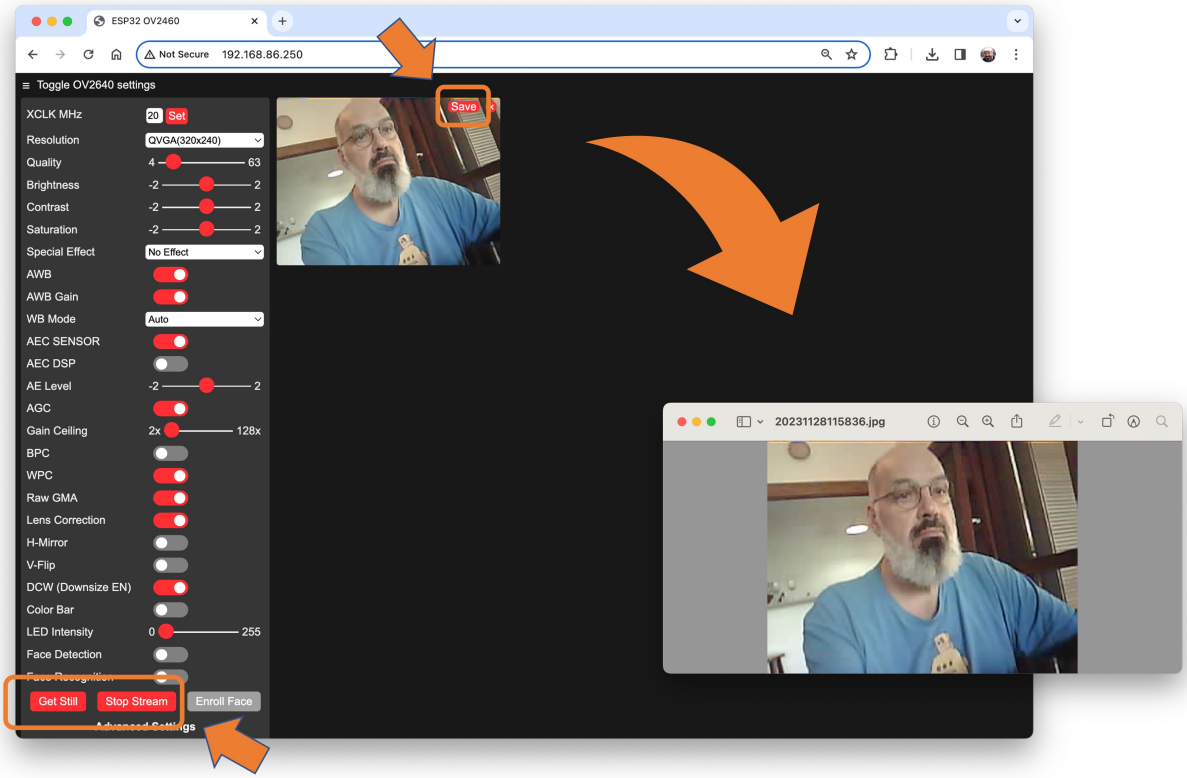
Do not forget the `Tools` to enable the PSRAM.

Enter your wifi credentials and upload the code to the device:

If the code is executed correctly, you should see the address on the Serial Monitor:

```
.
WiFi connected
[  1946][I][app_httpd.cpp:1361] startCameraServer(): Starting web server on port: '80'
[  1948][I][app_httpd.cpp:1379] startCameraServer(): Starting stream server on port: '81'
Camera Ready! Use 'http://192.168.86.250' to connect
```

Copy the address on your browser and wait for the page to be uploaded. Select the camera resolution (for example, QVGA) and select `[START STREAM]`. Wait for a few seconds/minutes, depending on your connection. Using the `[Save]` button, you can save an image to your computer download area.



That's it! You can save the images directly on your computer for use on projects.

1.8 Conclusion

The XIAO ESP32S3 Sense is flexible, inexpensive, and easy to program. With 8 MB of RAM, memory is not an issue, and the device can handle many

post-processing tasks, including communication.

You will find the last version of the codes on the GitHub repository: XIAO-ESP32S3-Sense.

2 Image Classification

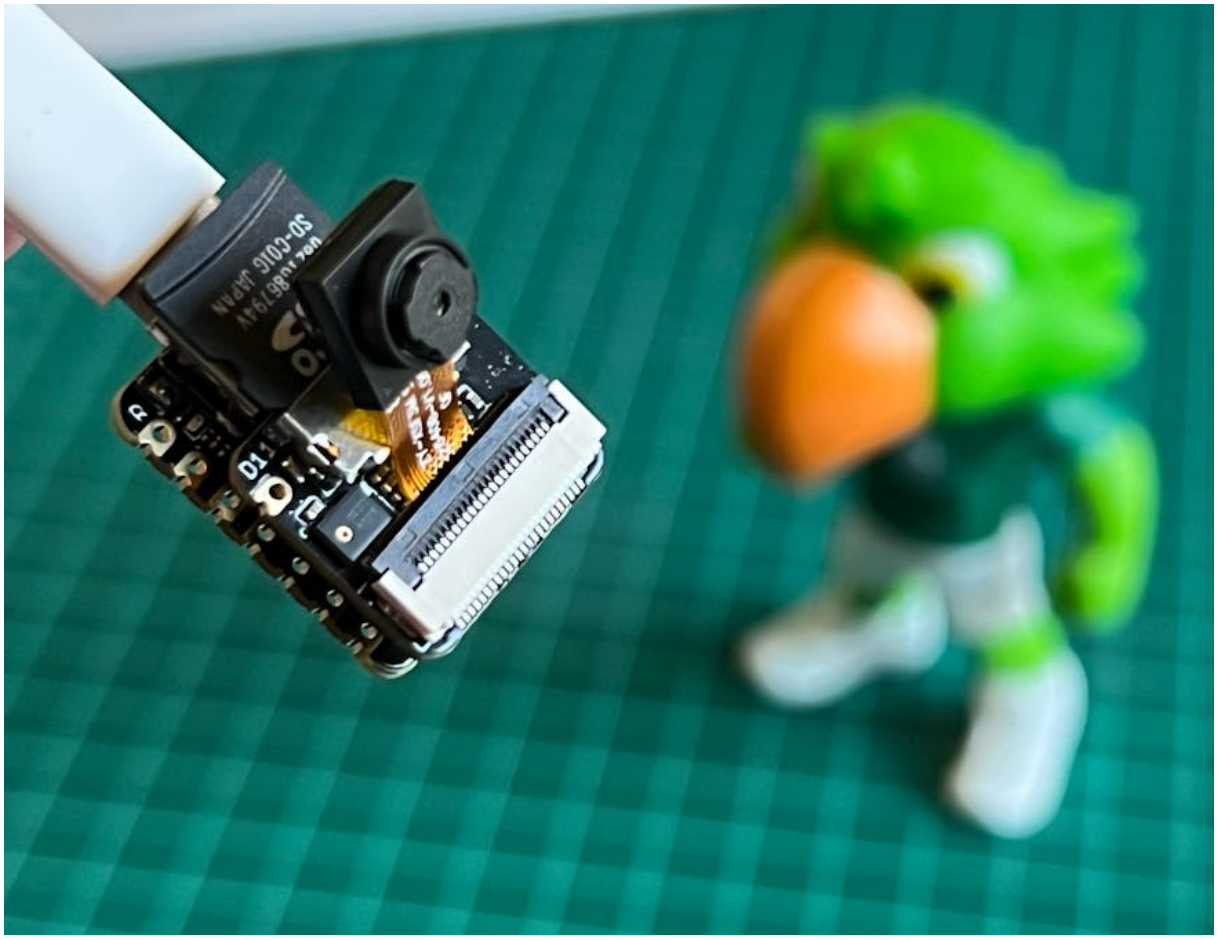


Image by Marcelo Rovai

2.1 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**!

At the forefront of the Emerging Technologies Radar is the universal language of Edge Computer Vision. When we delve into Machine Learning (ML) applied to vision, the first concept that greets us is Image Classification, a kind of ML' Hello World ' that is both simple and profound!

The Seeed Studio XIAO ESP32S3 Sense is a powerful tool that combines camera and SD card support. With its embedded ML computing power and photography capability, it is an excellent starting point for exploring TinyML vision AI.

2.2 A TinyML Image Classification Project - Fruits versus Veggies

The whole idea of our project will be to train a model and proceed 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 techniques associated with machine learning inference on embedded devices, we should limit the classification to three or four categories due to limitations (mainly memory). 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).

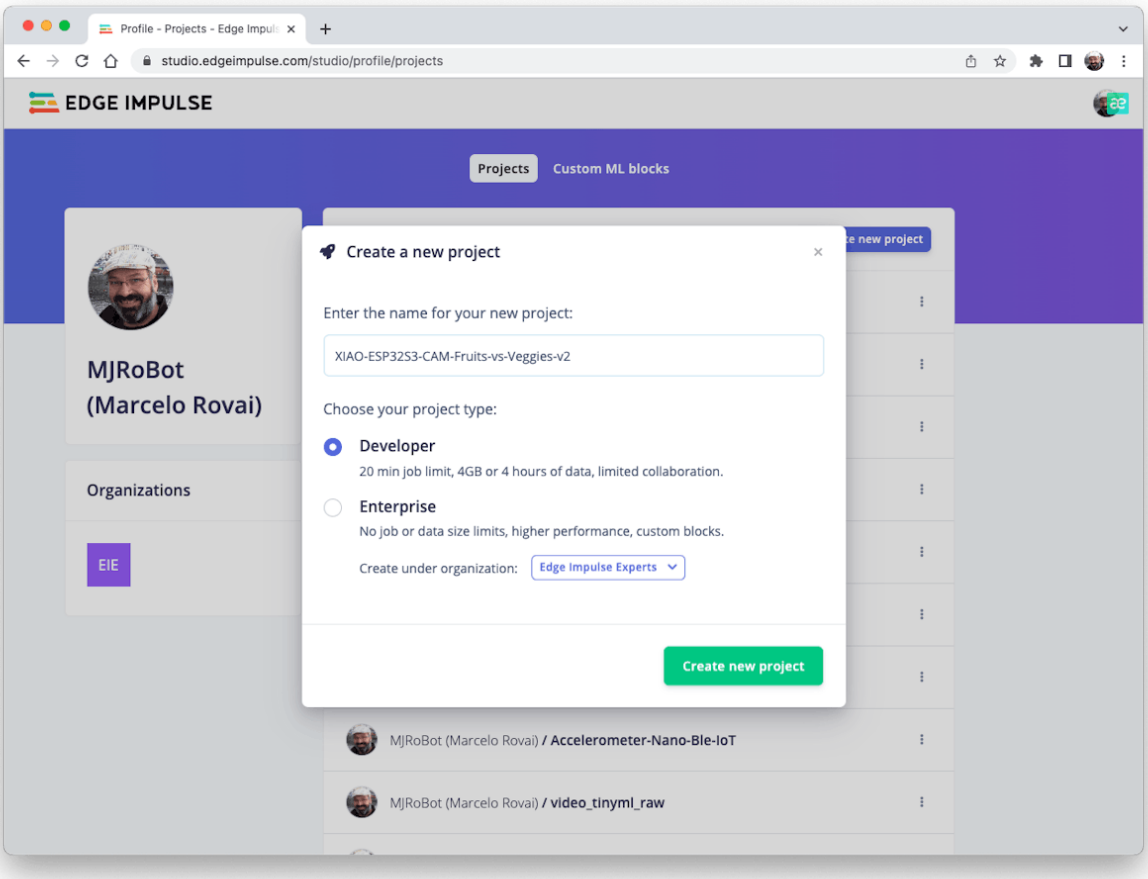
- Download the dataset from the Kaggle website and put it on your computer.

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

2.3 Training the model with Edge Impulse Studio

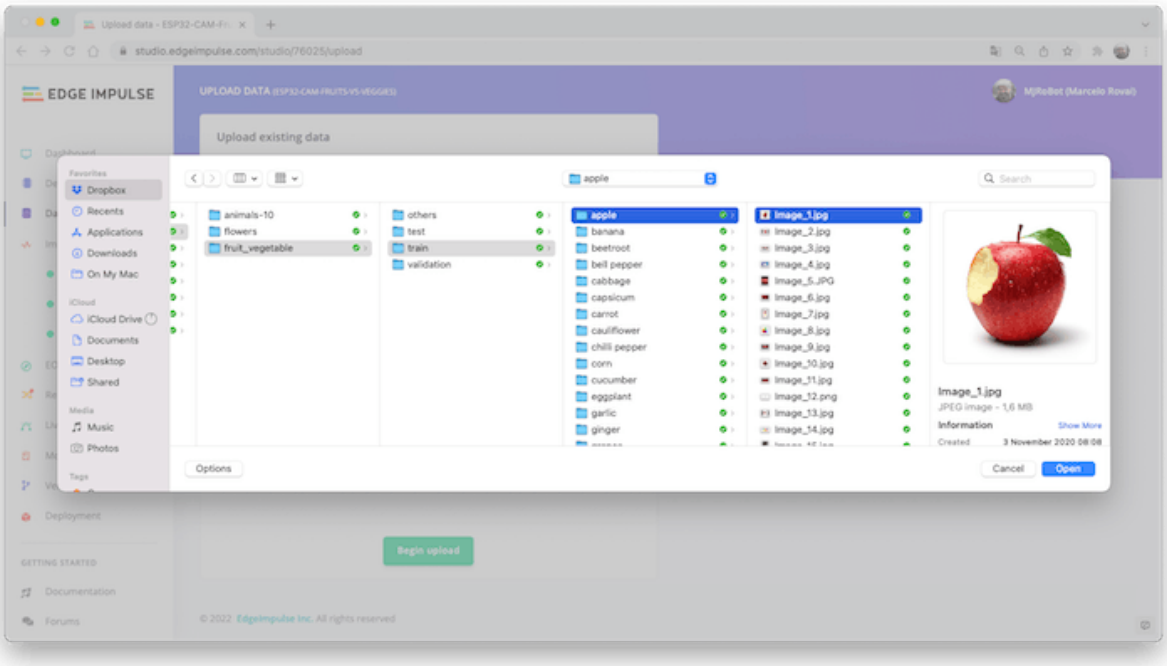
We will use the Edge Impulse Studio to train our model. As you may know, Edge Impulse is a leading development platform for machine learning on edge devices.

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

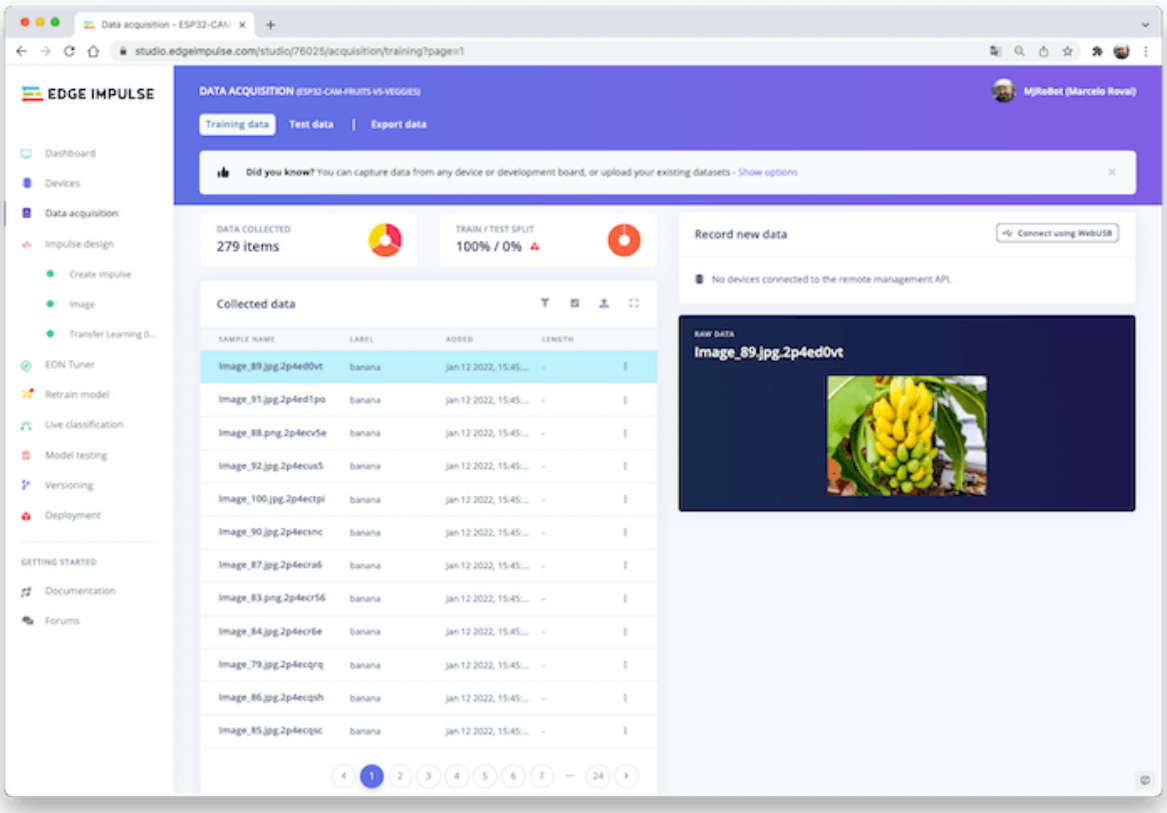


2.3.1 Data Acquisition

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



It would be best if you now had your training dataset split into three classes of data:

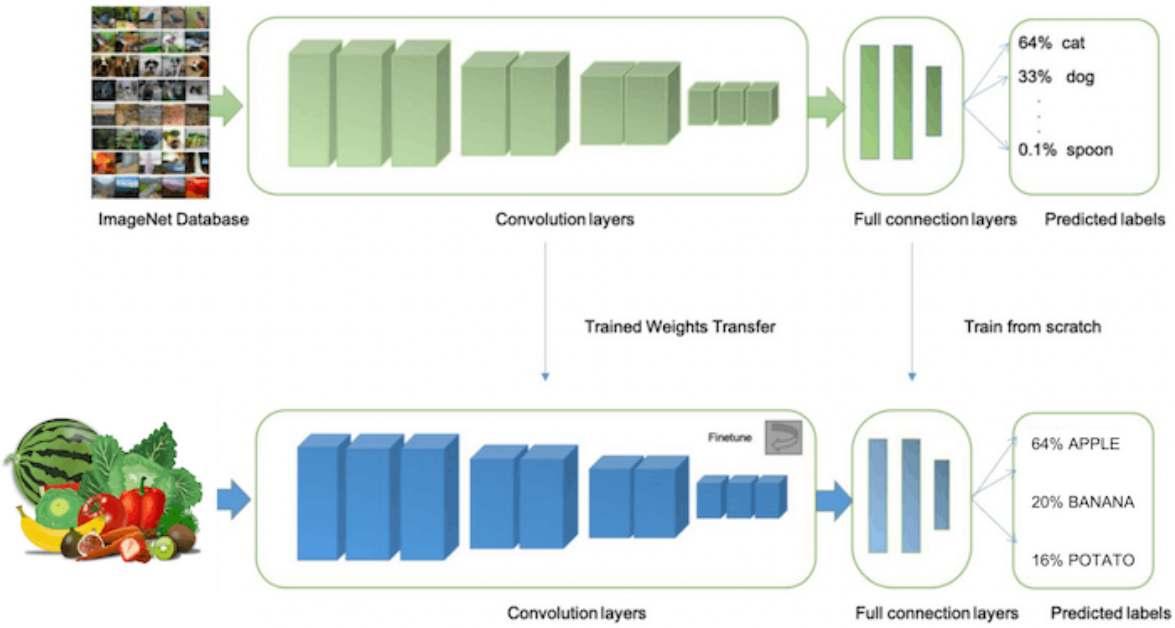


You can upload extra data for further model testing or split the training data. I will leave it as it is to use the most data possible.

2.3.2 Impulse Design

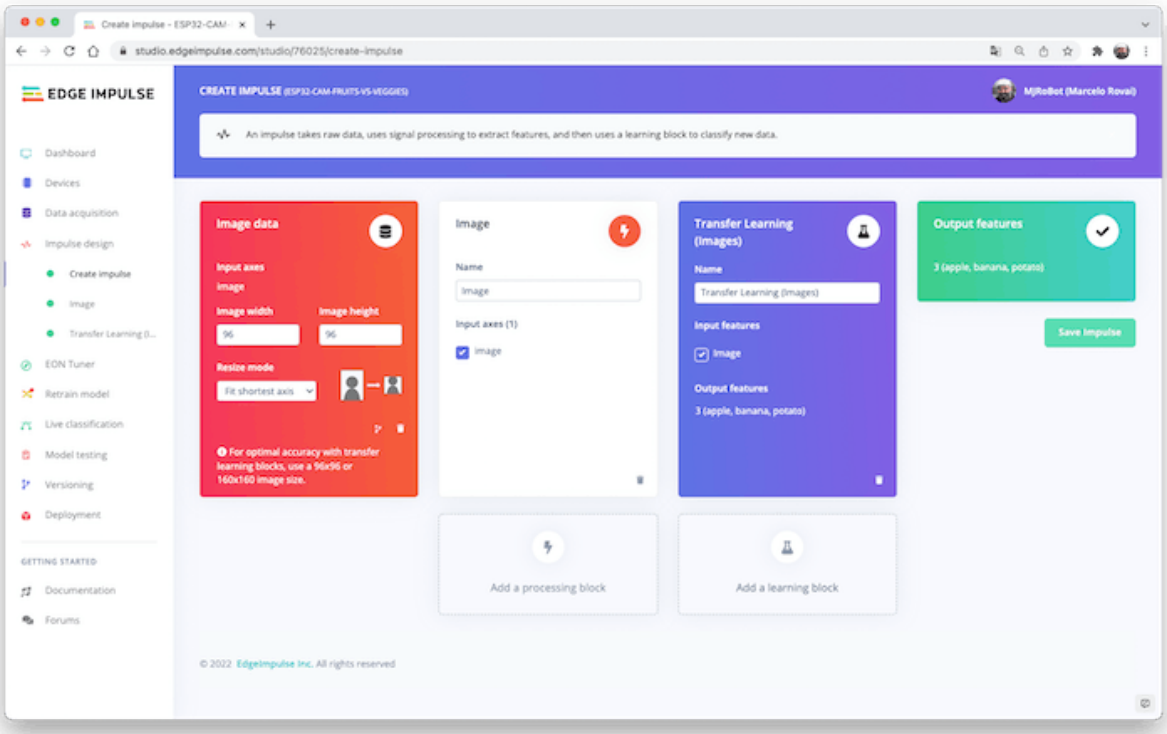
An impulse takes raw data (in this case, images), extracts features (resize pictures), and then uses a learning block to classify new data.

Classifying images is the most common use of deep learning, but a lot of 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 thousands 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 call this technique “Transfer Learning” (TL).



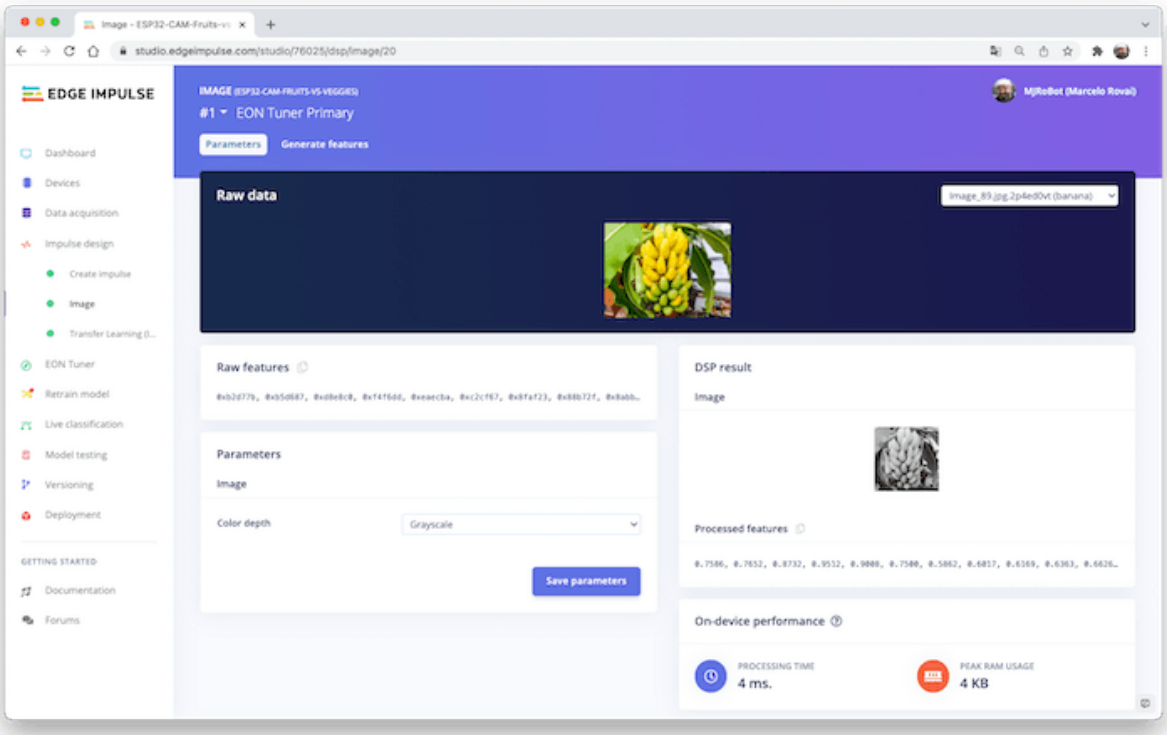
With TL, we can fine-tune a pre-trained image classification model on our data, performing well even with relatively small image datasets (our case).

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



2.3.2.1 Pre-processing (Feature Generation)

Besides resizing the images, we can change them to Grayscale or keep the actual RGB color depth. Let's start selecting `Grayscale`. 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.



Remember to `[Save parameters]`. This will generate the features to be used in training.

2.3.2.2 Model Design

Transfer Learning

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

Edge Impulse Studio has **MobileNet V1 (96x96 images)** and **V2 (96x96 and 160x160 images)** available, 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 higher the accuracy, the more memory (around 1.3M RAM and 2.6M ROM) will be needed to run the model, implying 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).

For this first pass, we will use **MobileNet V1** and α=0.10.

2.3.3 Training

Data Augmentation

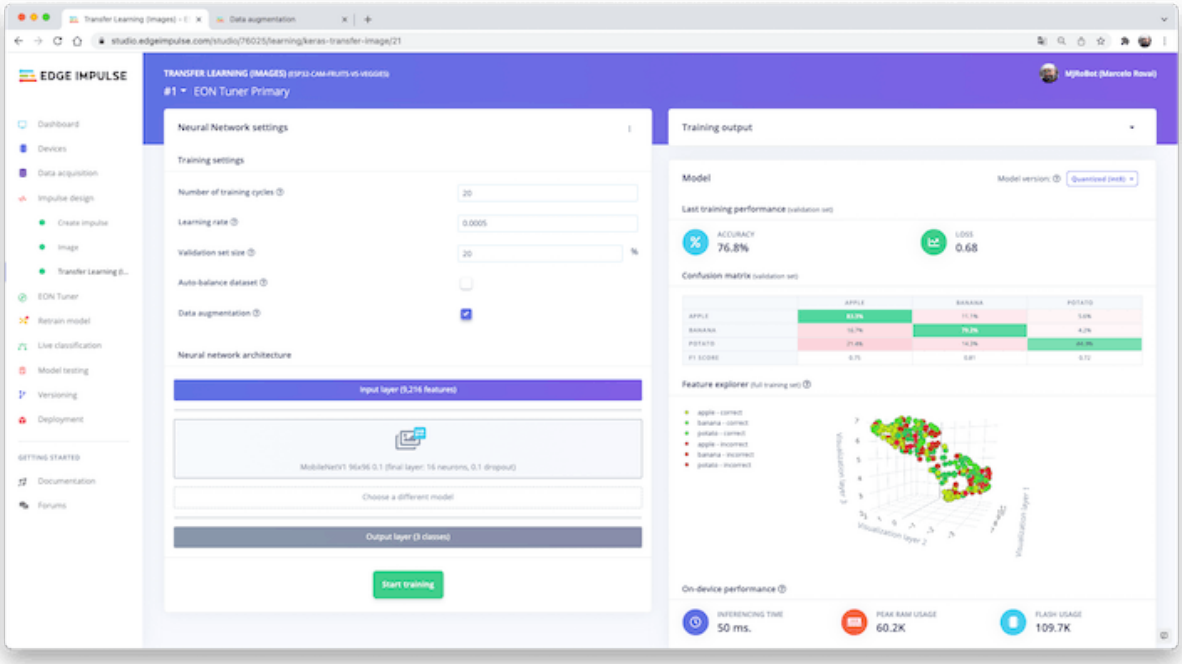
Another necessary technique to use with deep learning is **data augmentation**. Data augmentation is a method that can help improve the accuracy of machine learning models, creating additional artificial data. A data augmentation system makes small, random changes to your training data during the training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

Under the rood, here 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.

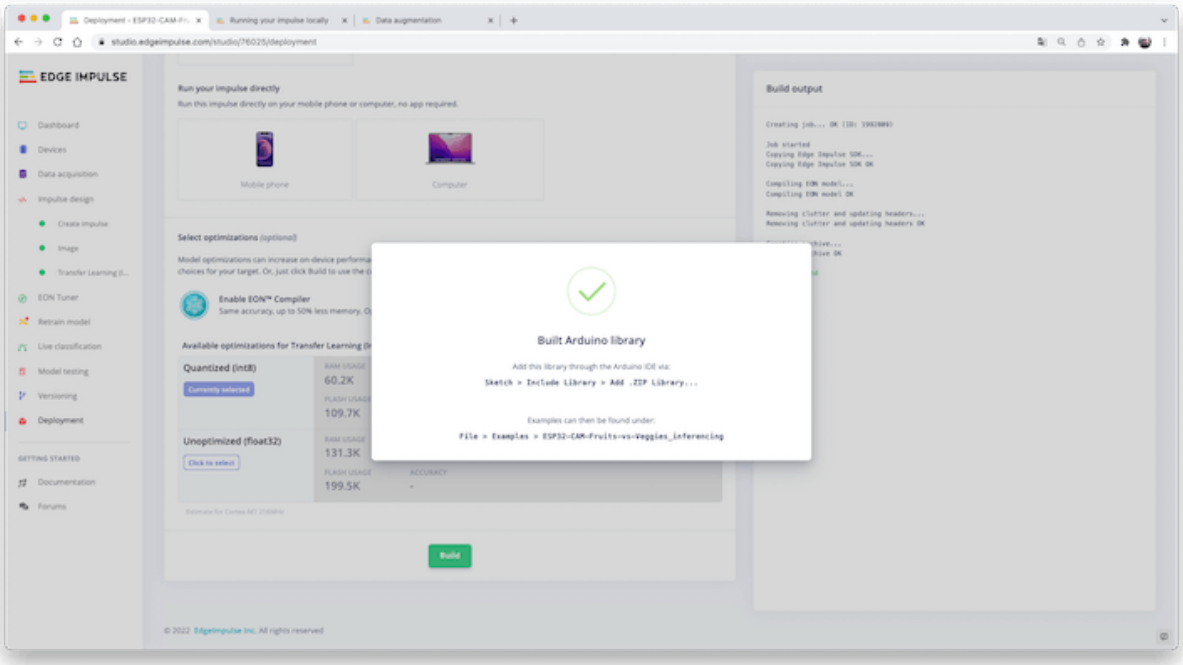
The final layer of our model will have 16 neurons with a 10% dropout for overfitting prevention. Here is the Training output:



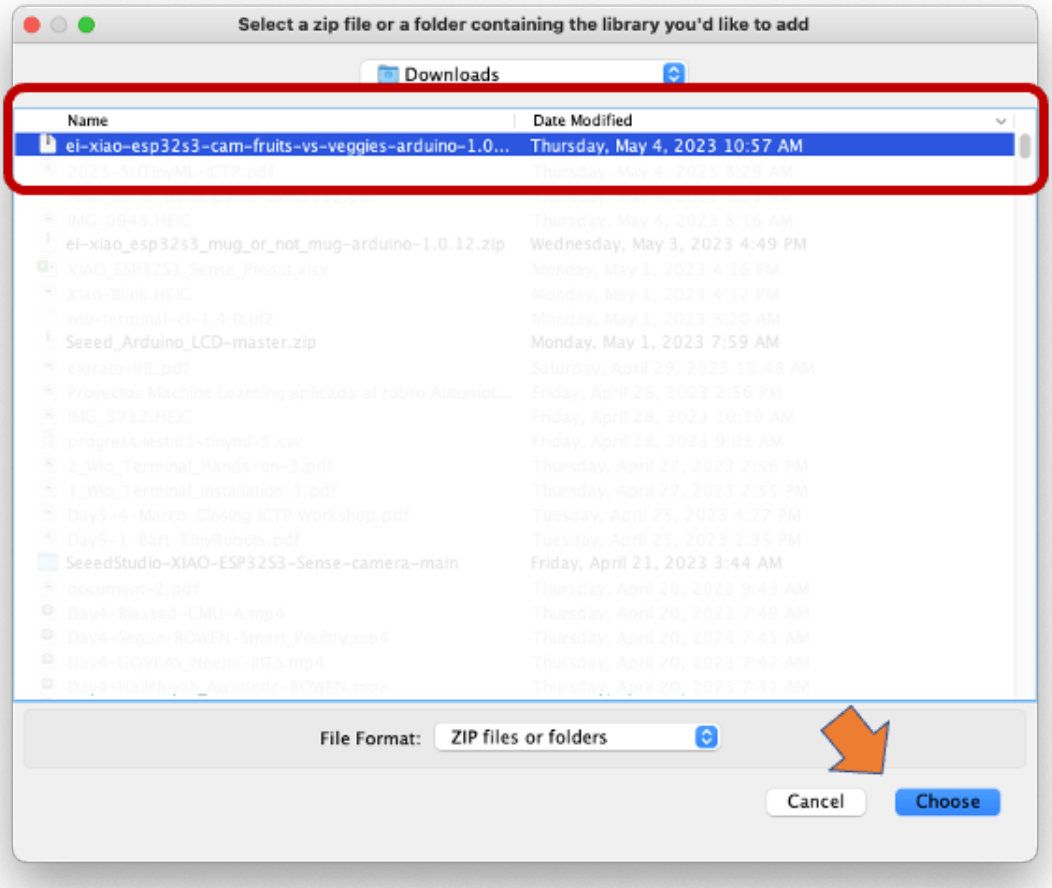
The result could be better. The model reached around 77% accuracy, but the amount of RAM expected to be used during the inference is relatively tiny (about 60 KBytes), which is very good.

2.3.4 Deployment

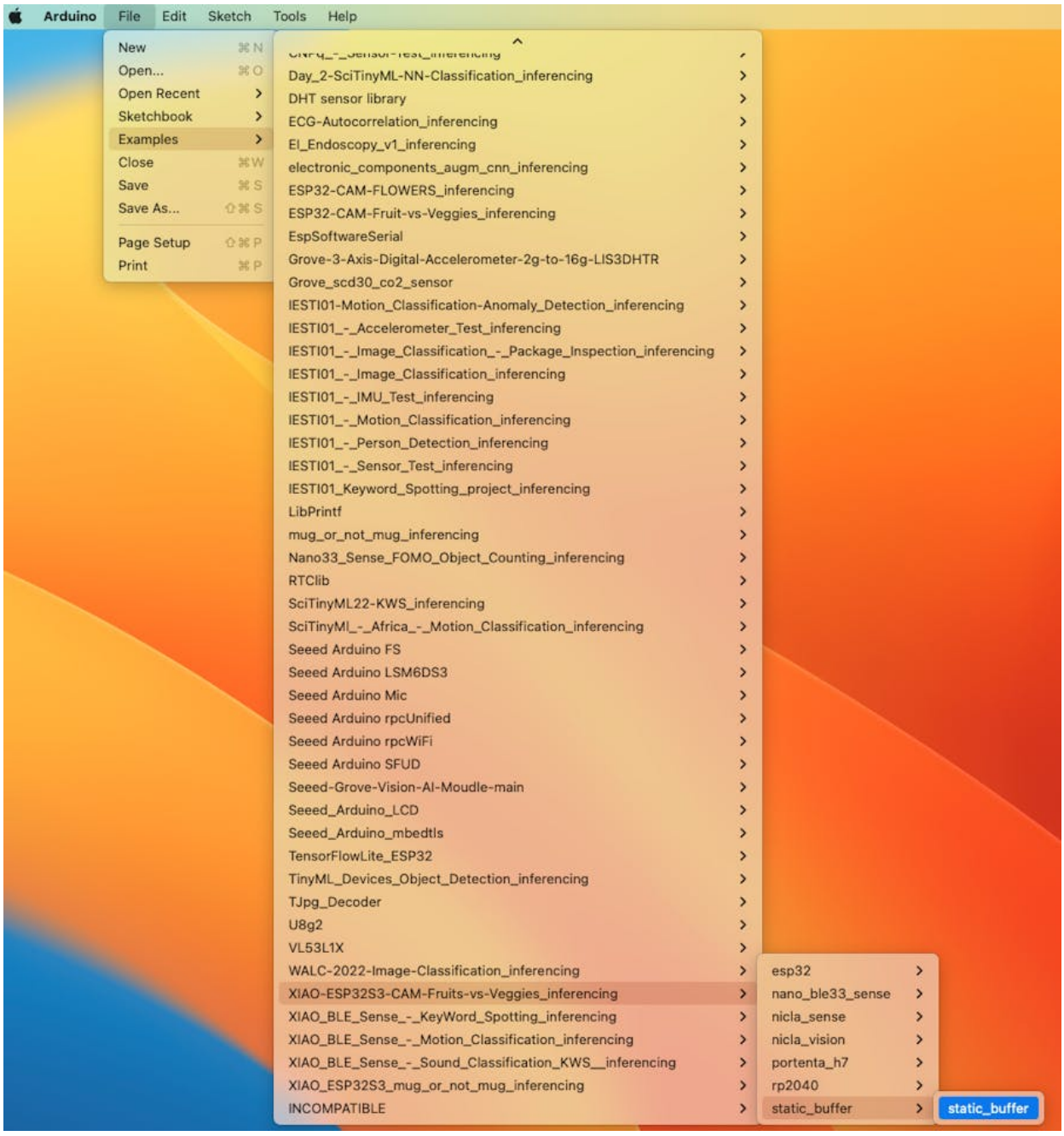
The trained model will be deployed as a .zip Arduino library:



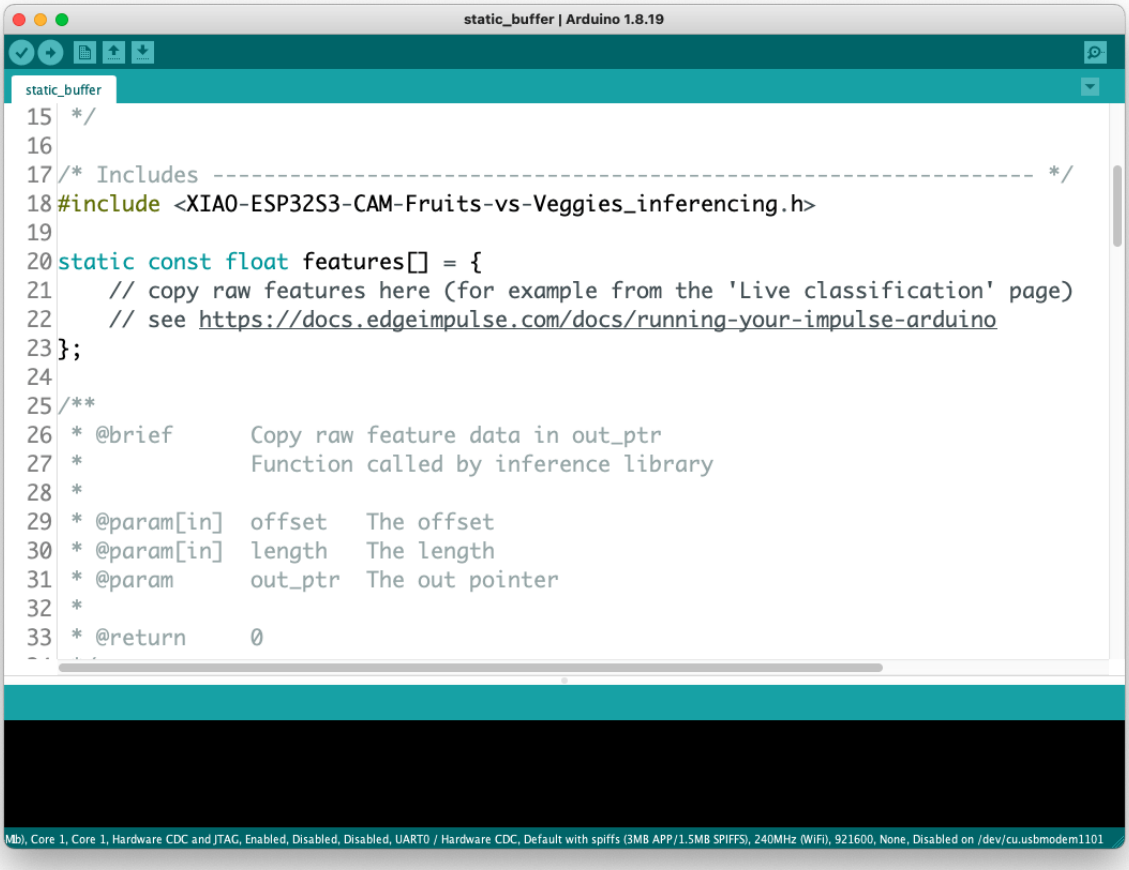
Open your Arduino IDE, and under **Sketch,** go to **Include Library** and **add.ZIP Library.** Please select the file you download from Edge Impulse Studio, and that's it!



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



Open the Static Buffer example:

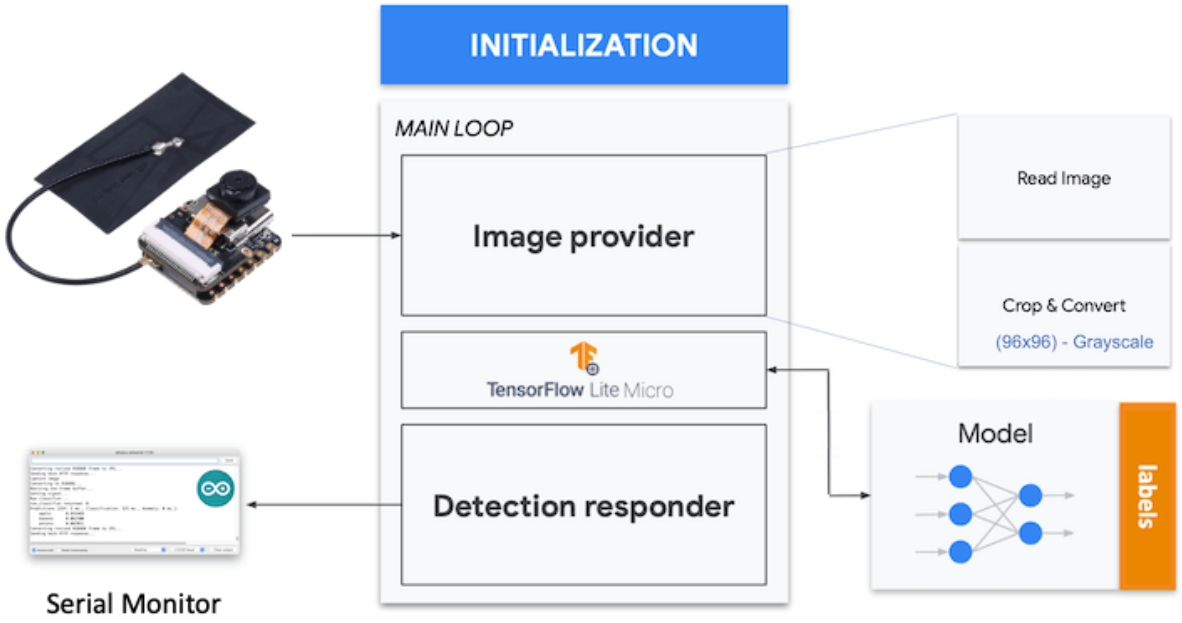


You can see that the first line of code is exactly the calling of a library with all the necessary stuff for running inference on your device.

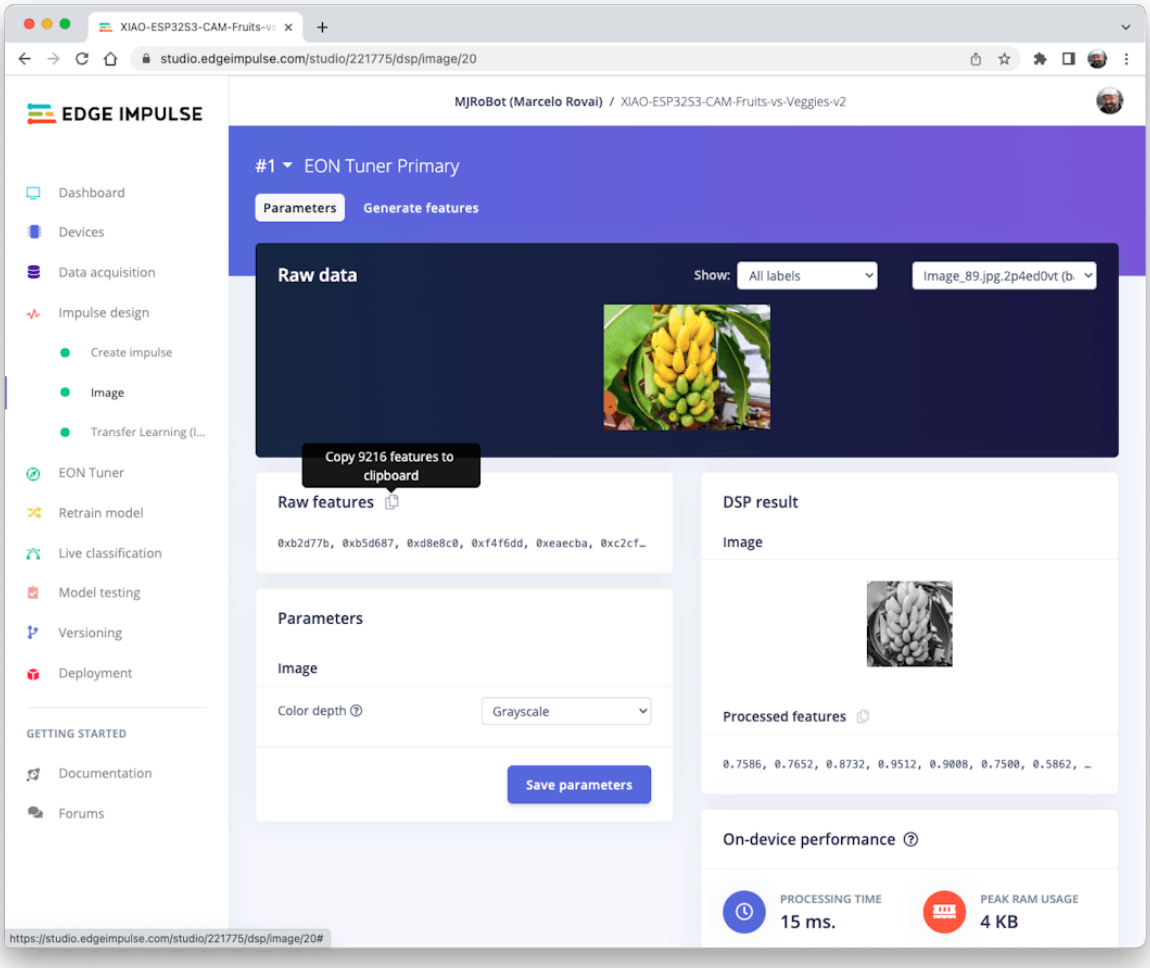
```
#include <XIAO-ESP32S3-CAM-Fruits-vs-Veggies_inferencing.h>
```

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

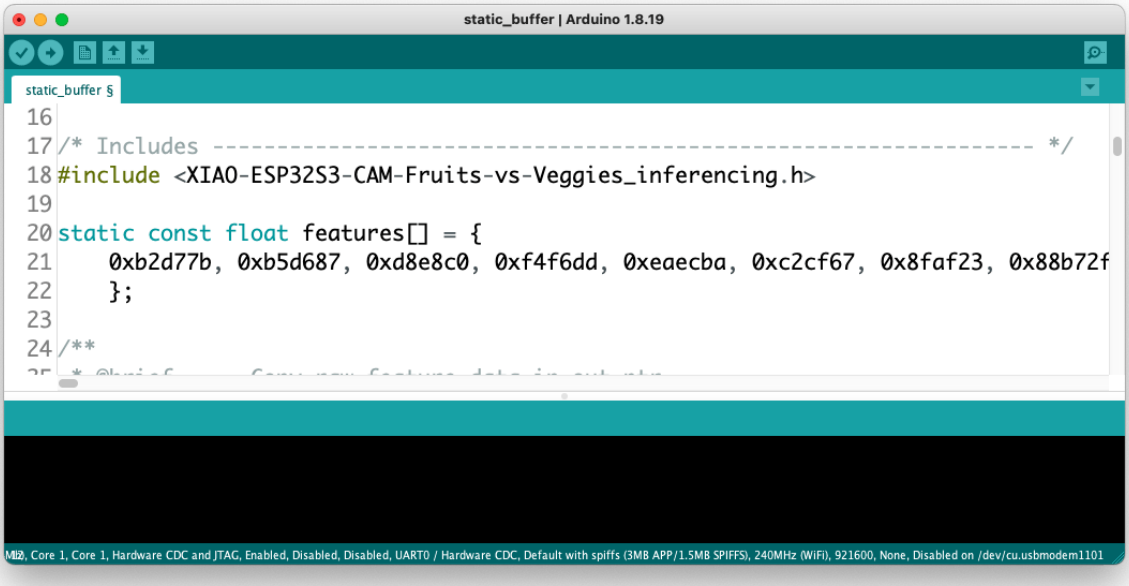
We should get the sample (image) from the camera and pre-process it (resizing to 96x96, converting to grayscale, and flatting it). This will be the input tensor of our model. The output tensor will be a vector with three values (labels), showing the probabilities of each one of the classes.



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

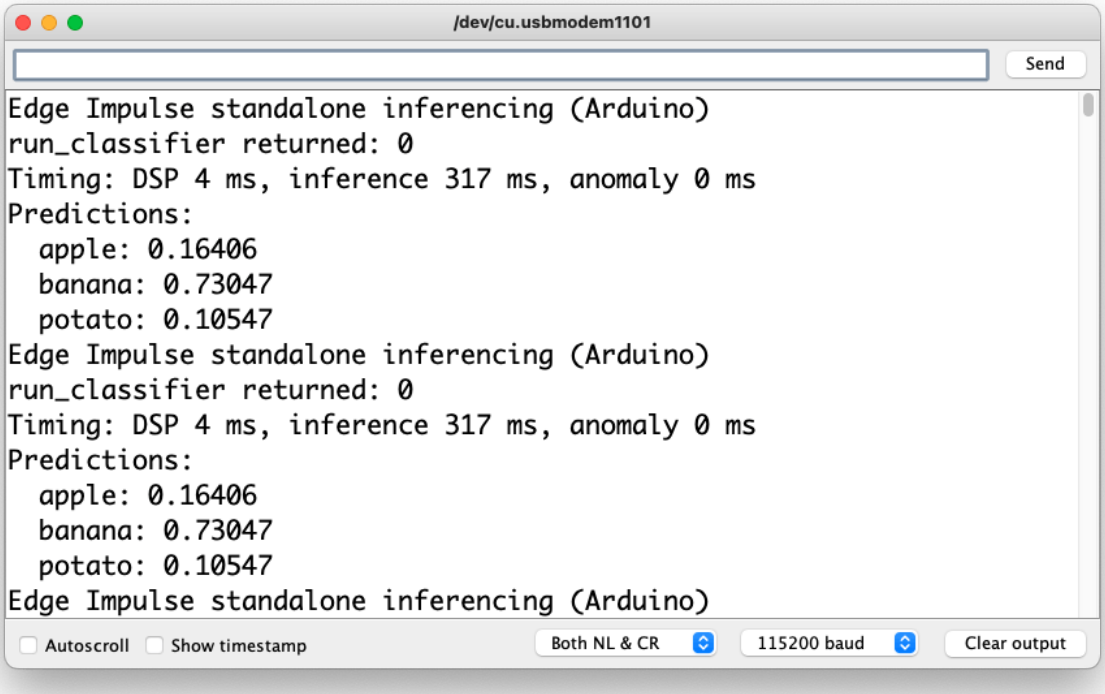


9, 216 features will be copied to the clipboard. This is the input tensor (a flattened image of 96x96x1), in this case, bananas. Past this Input tensor on`features[] = {0xb2d77b, 0xb5d687, 0xd8e8c0, 0xeaecba, 0xc2cf67, ...}`



Edge Impulse included the library ESP NN in its SDK, which contains optimized NN (Neural Network) functions for various Espressif chips, including the ESP32S3 (running at Arduino IDE).

When running the inference, you should get 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.

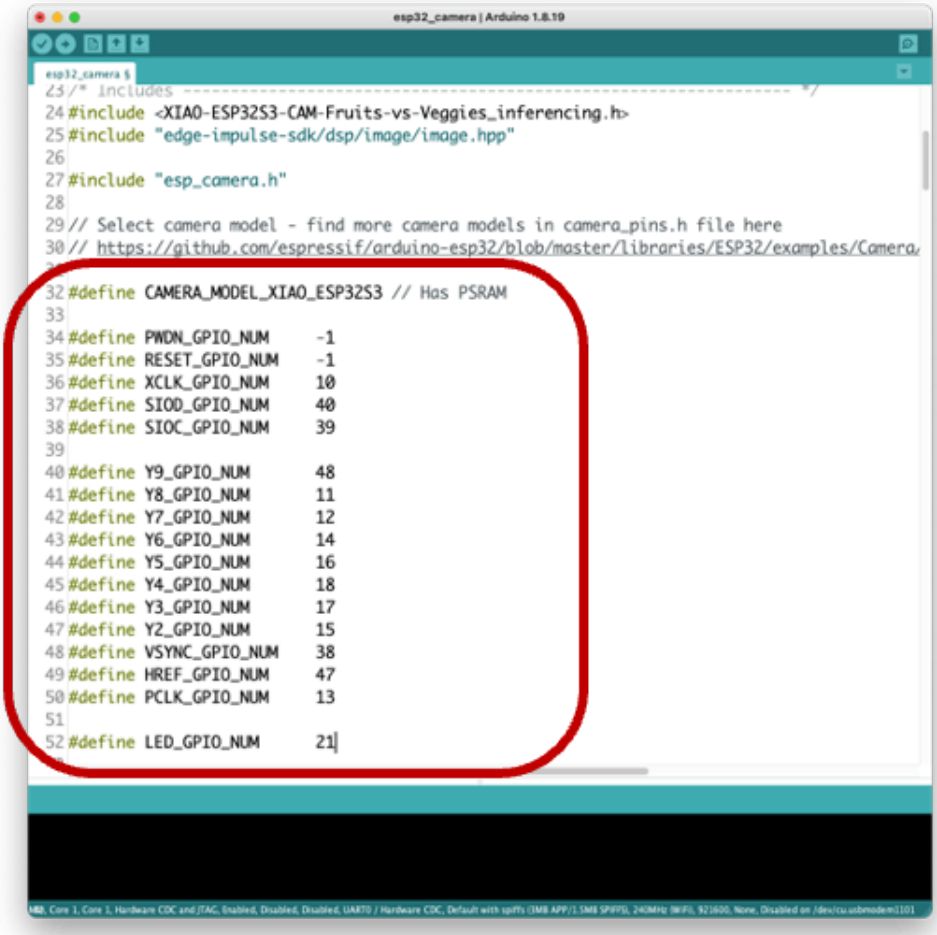
Now, we should incorporate the camera and classify images in real time.

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, using the data related to our model. Copy and paste the below lines, replacing the lines 32-75:

```
#define PWDN_GPIO_NUM     -1
#define RESET_GPIO_NUM    -1
#define XCLK_GPIO_NUM     10
#define SIOD_GPIO_NUM     40
#define SIOC_GPIO_NUM     39
#define Y9_GPIO_NUM       48
#define Y8_GPIO_NUM       11
#define Y7_GPIO_NUM       12
#define Y6_GPIO_NUM       14
#define Y5_GPIO_NUM       16
#define Y4_GPIO_NUM       18
#define Y3_GPIO_NUM       17
#define Y2_GPIO_NUM       15
#define VSYNC_GPIO_NUM    38
#define HREF_GPIO_NUM     47
#define PCLK_GPIO_NUM     13
```

Here you can see the resulting code:

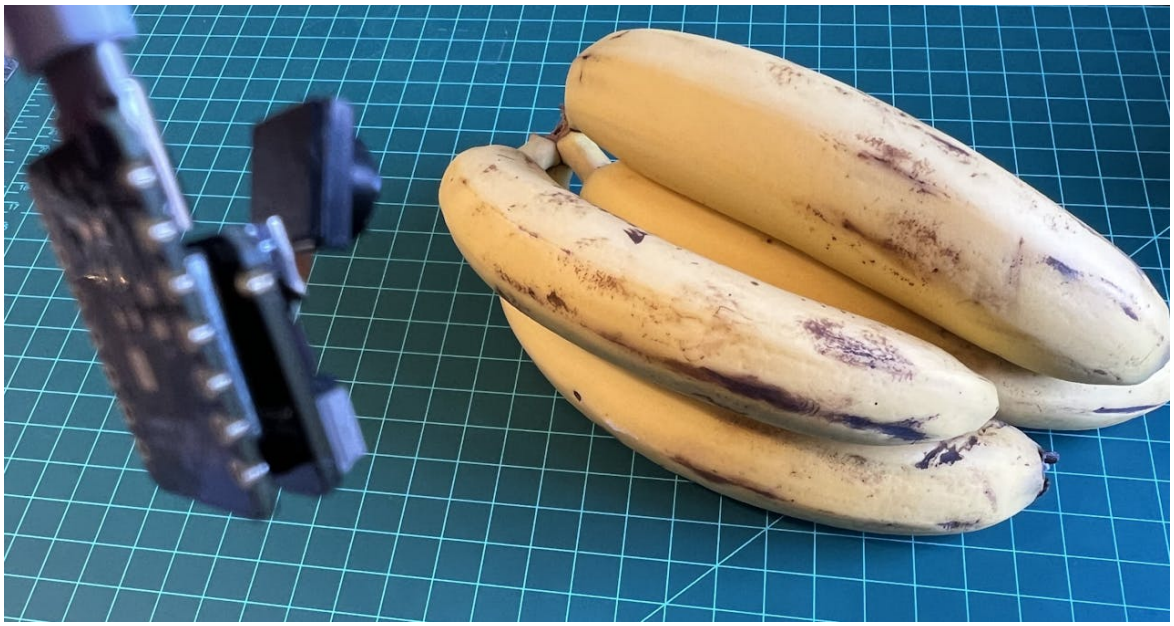


The modified sketch can be downloaded from GitHub: xiao_esp32s3_camera.

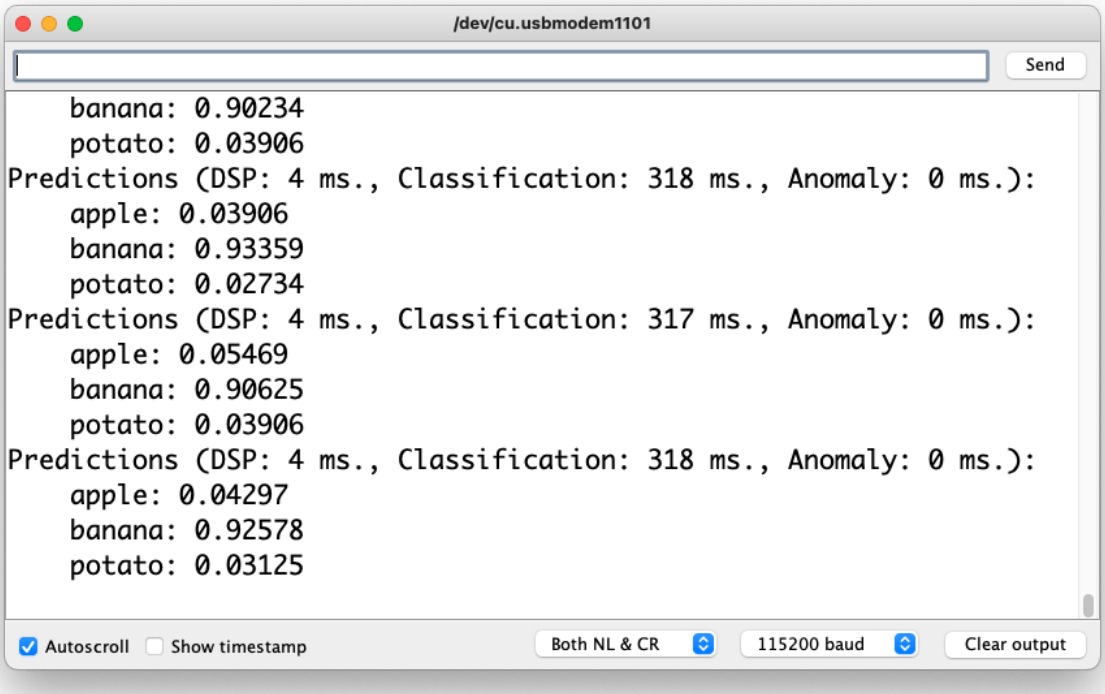
> Note that you can optionally keep the pins as a .h file as we did in the Setup Lab.

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.

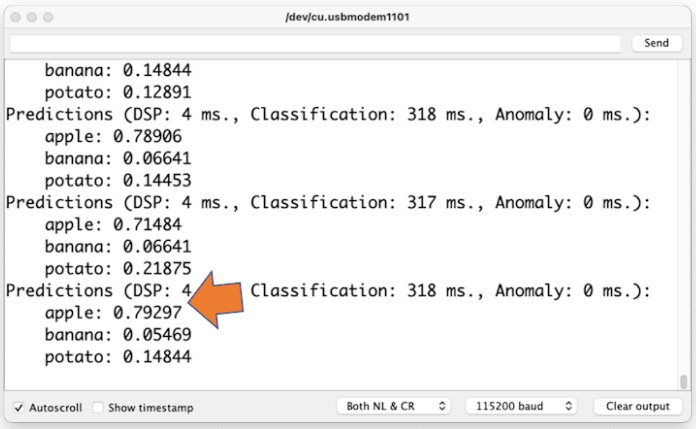
2.4 Testing the Model (Inference)

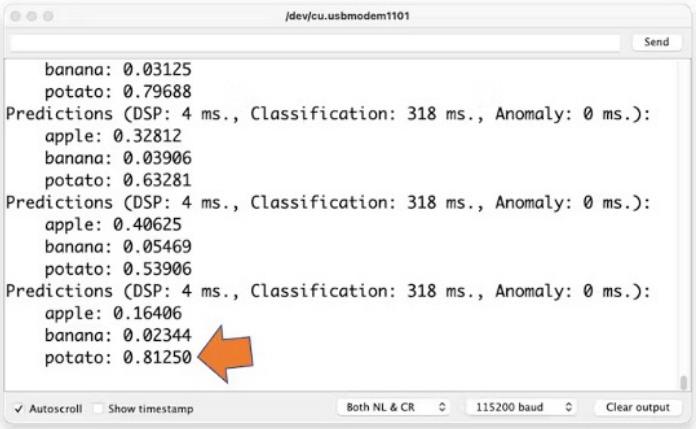


Getting a photo with the camera, the classification result will appear on the Serial Monitor:



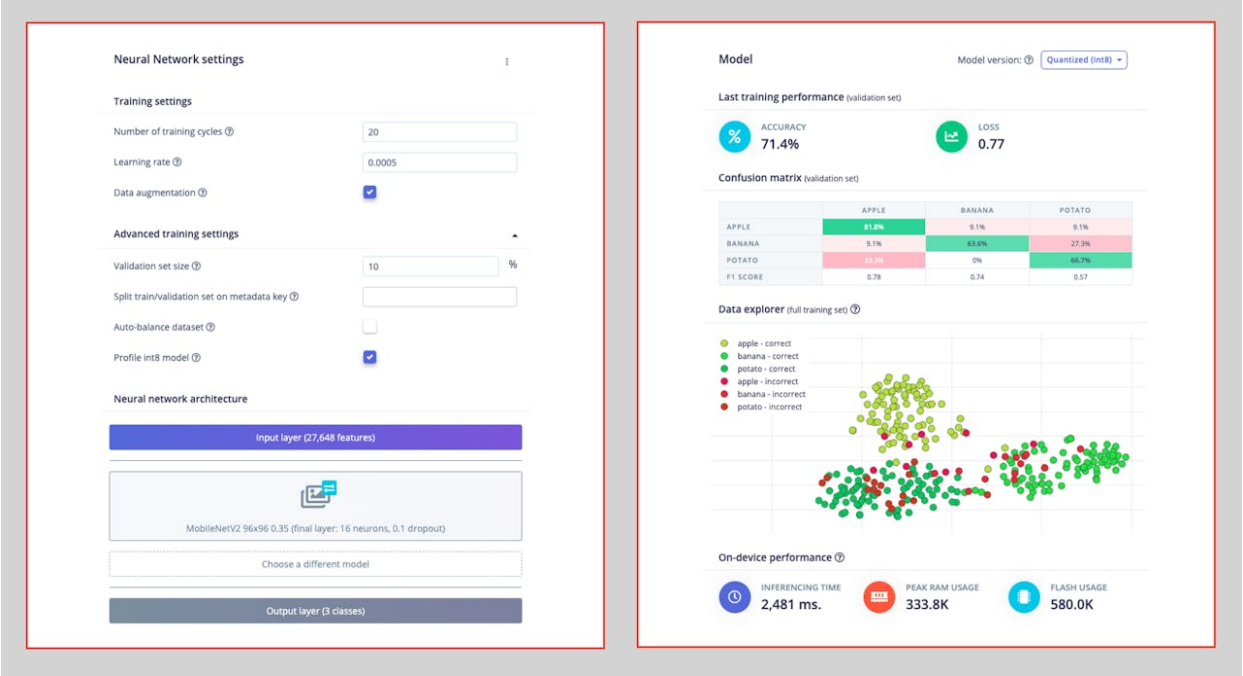
Other tests:





2.5 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 could be better, and the amount of memory necessary to run the model increases five times, with latency increasing seven times.

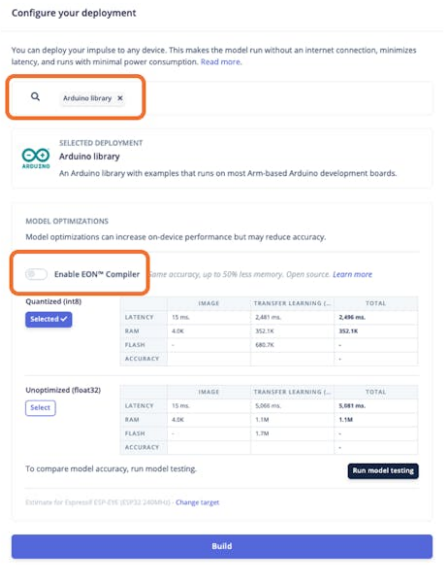
> Note that the performance here is estimated with a smaller device, the ESP-EYE. The actual inference with the ESP32S3 should be better.

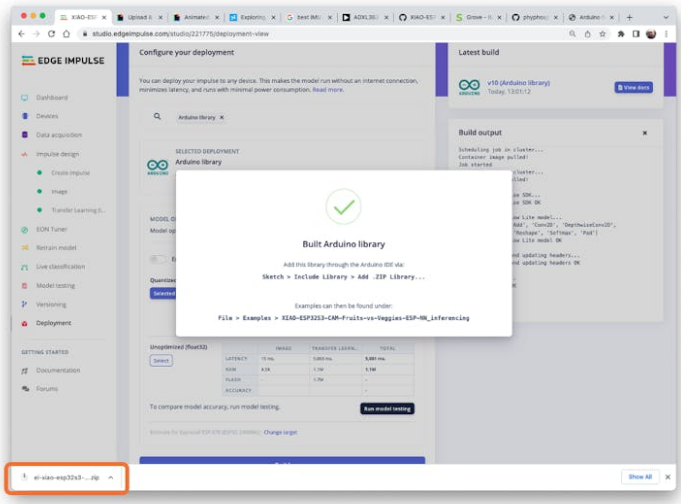
To improve our model, we will need to train more images.

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

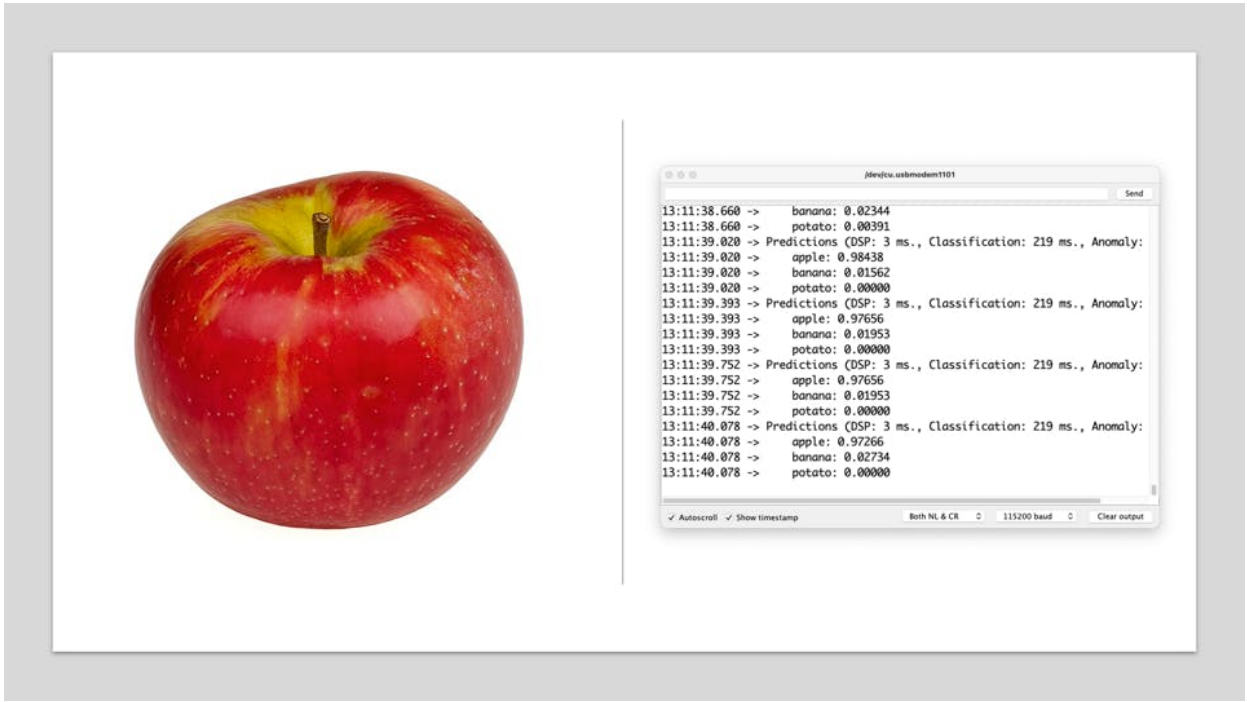
Let's redeploy the model. If the EON Compiler is enabled when you generate the library, the total memory needed for inference should be reduced, but it does not influence accuracy.

> ⚠️ **Attention** - The Xiao ESP32S3 with PSRAM enable has enought memory to run the inference, even in such bigger model. Keep the EON Compiler **NOT ENABLED**.

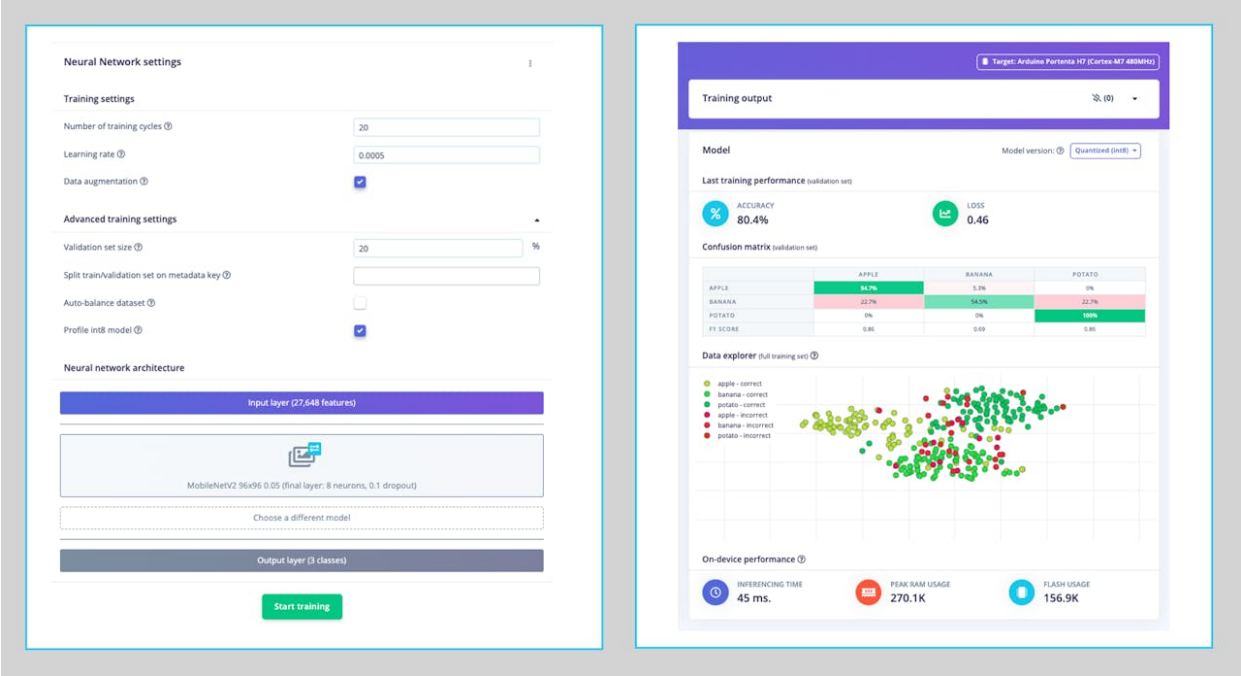




Doing an inference with MobilinetV2 96x96 0.35, having as input RGB images, the latency was 219ms, which is great for such a bigger model.

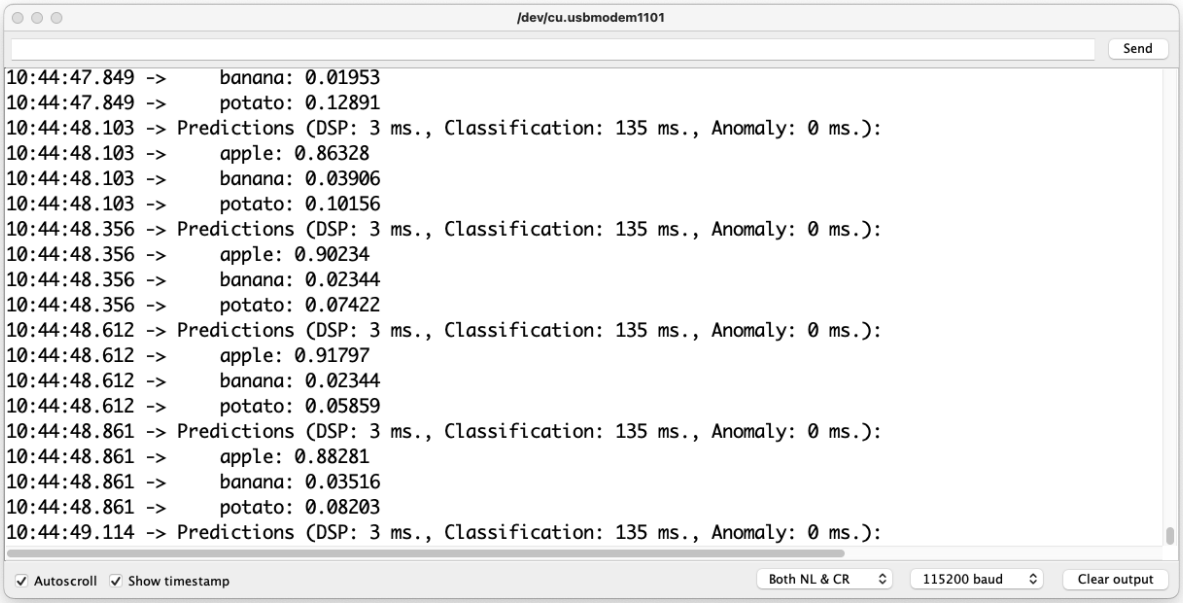


For the test, we can train the model again, using the smallest version of MobileNet V2, with an alpha of 0.05. Interesting that the result in accuracy was higher.



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

Deploying the model, we got an inference of only 135ms, remembering that the XIAO runs with half of the clock used by the Portenta/Nicla (240MHz):



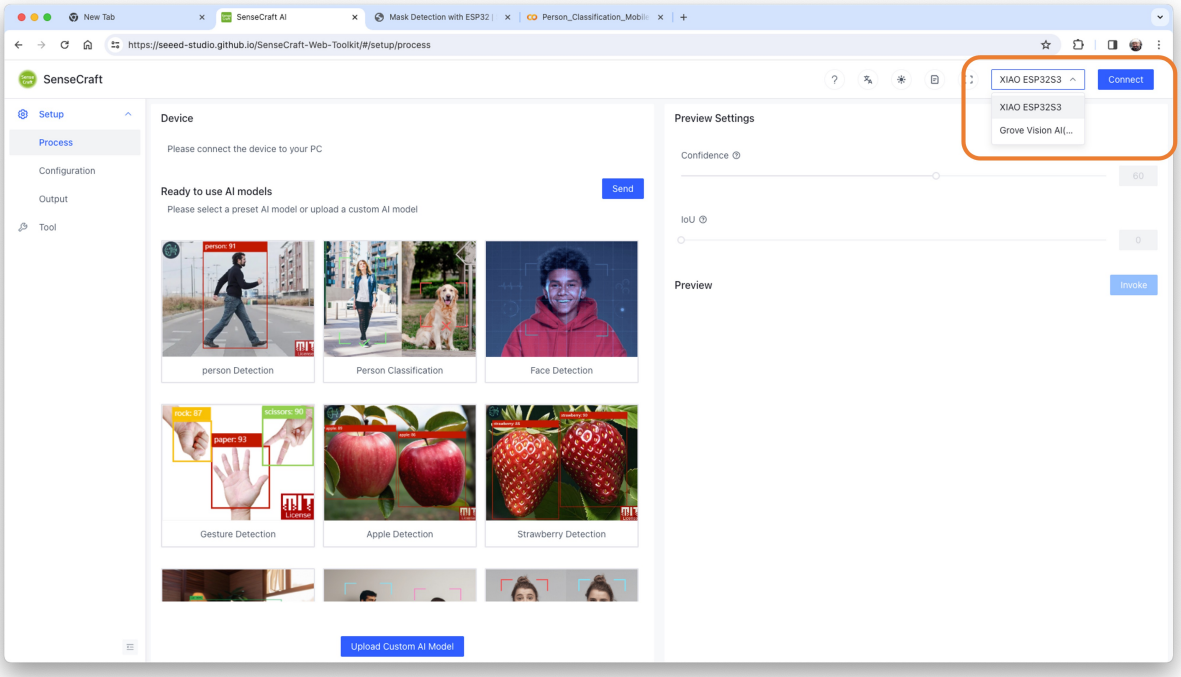
2.6 Running inference on the SenseCraft-Web-Toolkit

One significant limitation of viewing inference on Arduino IDE is that we can not see what the camera focuses on. A good alternative is the **SenseCraft-Web-Toolkit**, a visual model deployment tool provided by SSCMA(Seeed SenseCraft Model Assistant). This tool allows you to deploy models to various platforms easily through simple operations. The tool offers a user-friendly interface and does not require any coding.

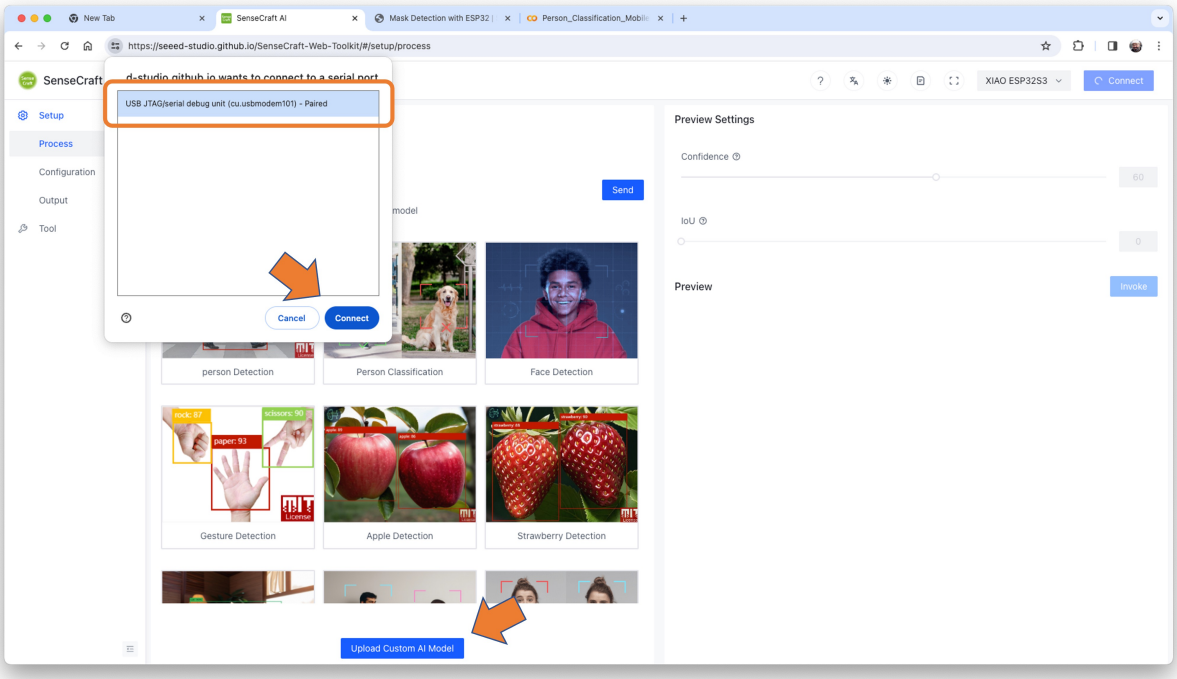
Follow the following steps to start the SenseCraft-Web-Toolkit:

1. Open the SenseCraft-Web-Toolkit website.
2. Connect the XIAO to your computer:

- Having the XIAO connected, select it as below:



- Select the device/Port and press `[Connect]`:



You can try several Computer Vision models previously uploaded by Seeed Studio. Try them and have fun!

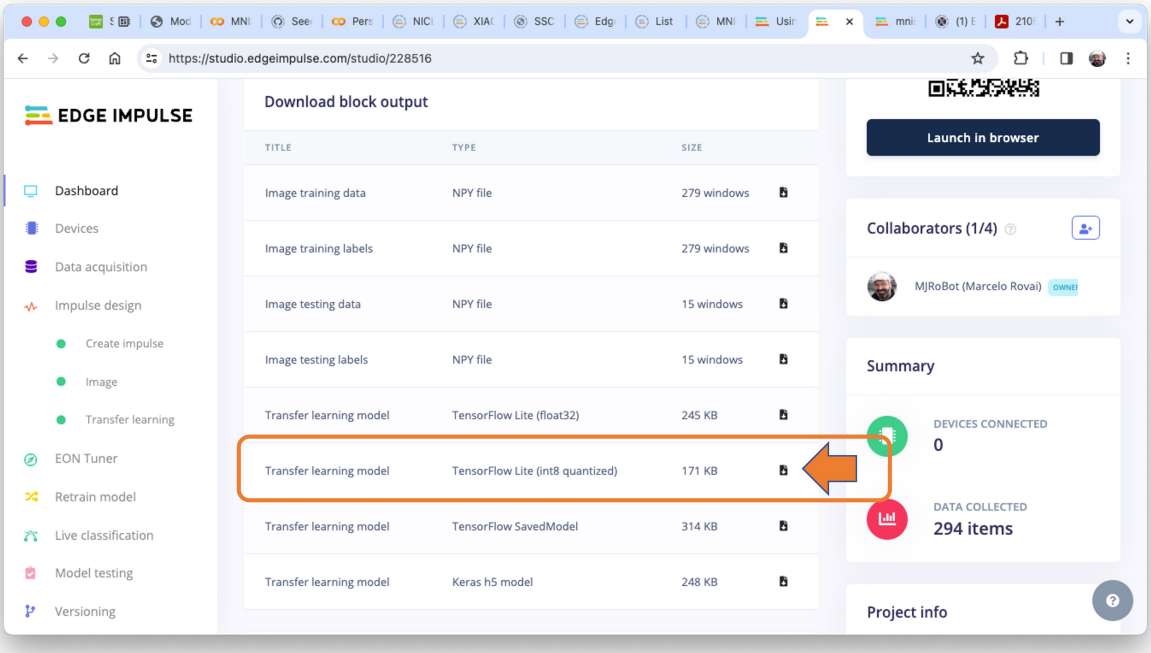
In our case, we will use the blue button at the bottom of the page: `[Upload Custom AI Model]`.

But first, we must download from Edge Impulse Studio our **quantized .tflite** model.

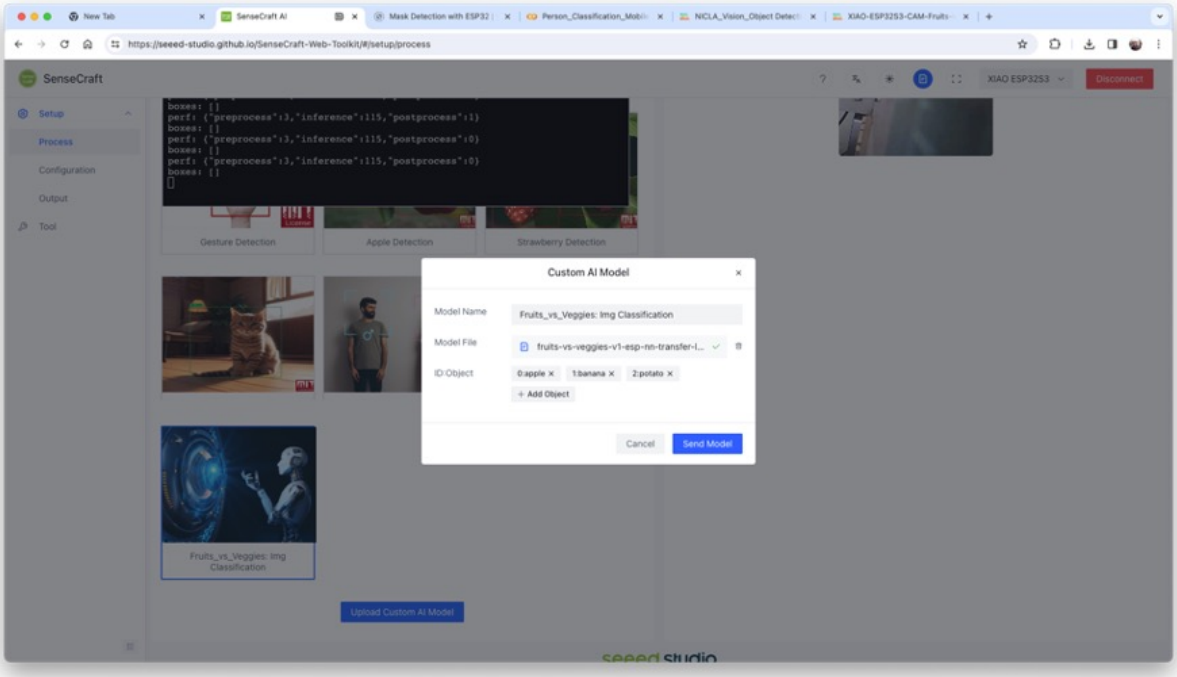
3. Go to your project at Edge Impulse Studio, or clone this one:

- XIAO-ESP32S3-CAM-Fruits-vs-Veggies-v1-ESP-NN

4. On the `Dashboard`, download the model ("block output"): `Transfer learning model - TensorFlow Lite (int8 quantized).`

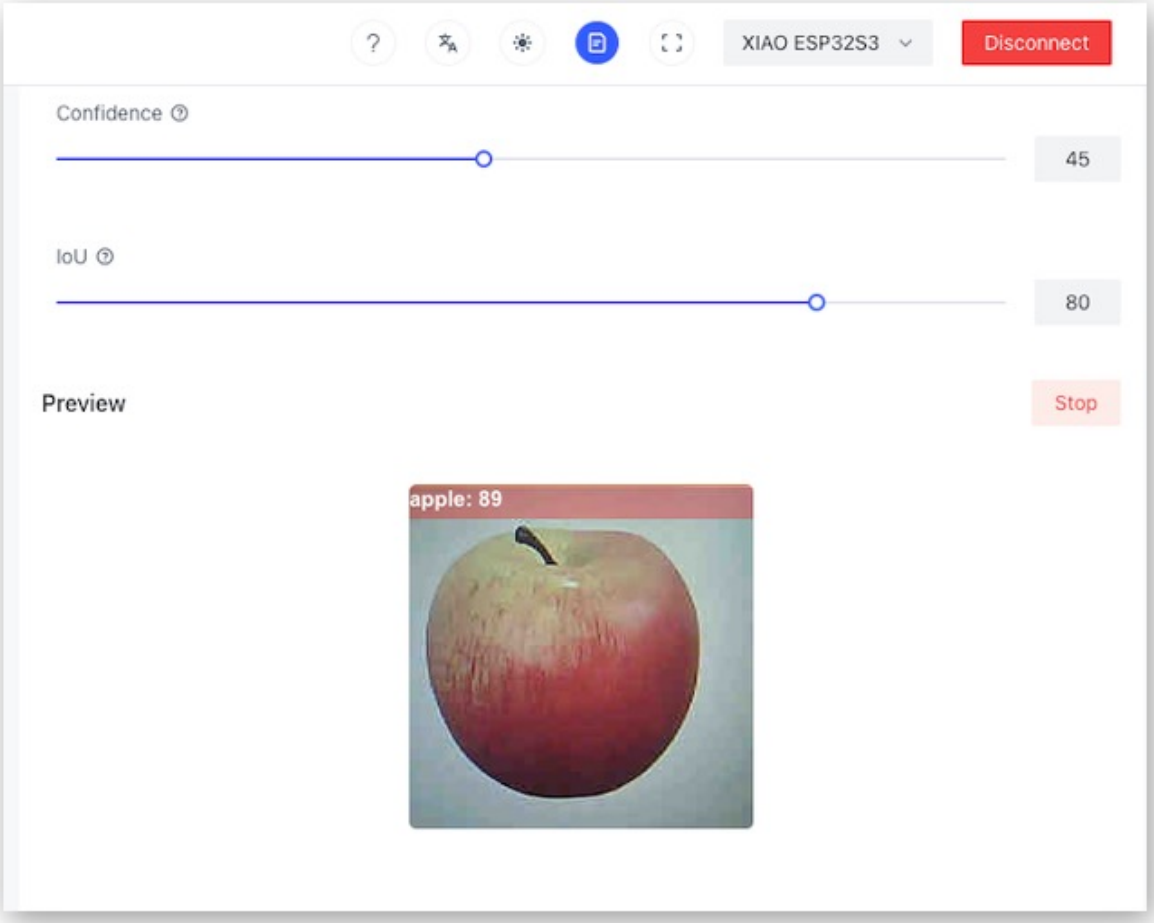


5. On SenseCraft-Web-Toolkit, use the blue button at the bottom of the page: `[Upload Custom AI Model]`. A window will pop up. Enter the Model file that you downloaded to your computer from Edge Impulse Studio, choose a Model Name, and enter with labels (ID: Object):



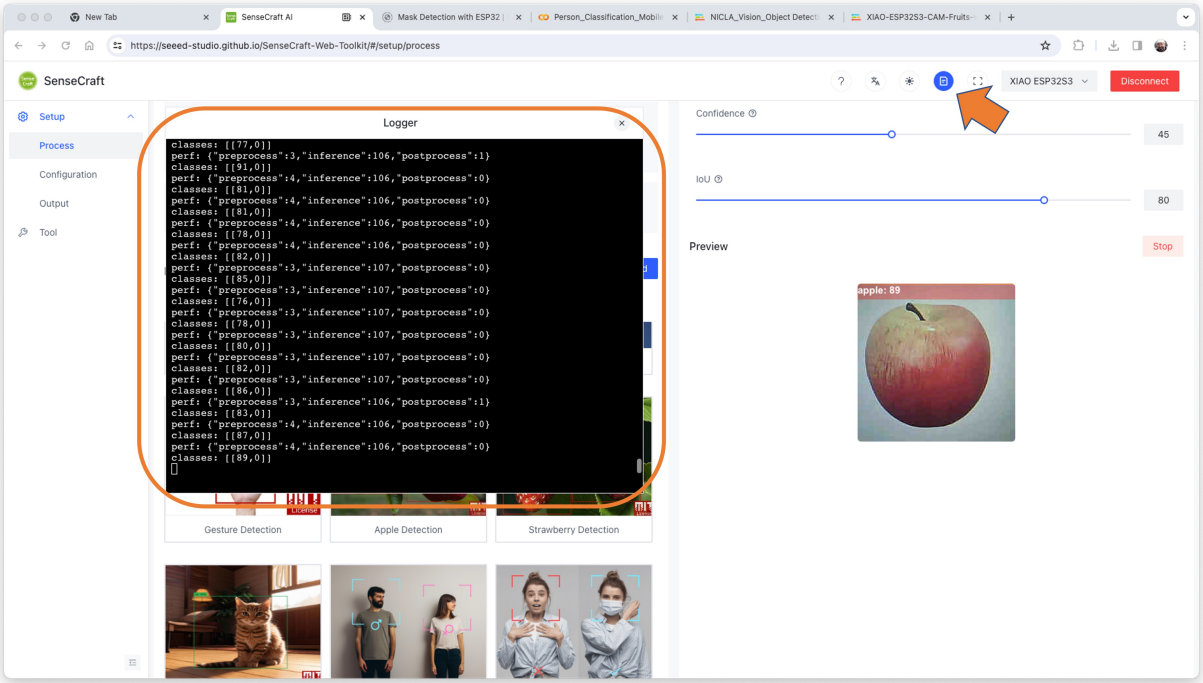
Note that you should use the labels trained on EI Studio, entering them in alphabetic order (in our case: apple, banana, potato).

After a few seconds (or minutes), the model will be uploaded to your device, and the camera image will appear in real-time on the Preview Sector:



The Classification result will be at the top of the image. You can also select the Confidence of your inference cursor `Confidence`.

Clicking on the top button (Device Log), you can open a Serial Monitor to follow the inference, the same that we have done with the Arduino IDE:



On Device Log, you will get information as:

```
perf: {"preprocess":4,"inference":106,"postprocess":0}
classes: [[89,0]]
```

- Preprocess time (image capture and Crop): 4ms;
- Inference time (model latency): 106ms,
- Postprocess time (display of the image and inclusion of data): 0ms.
- Output tensor (classes), for example: [[89,0]]; where 0 is Apple (and 1is banana and 2 is potato)

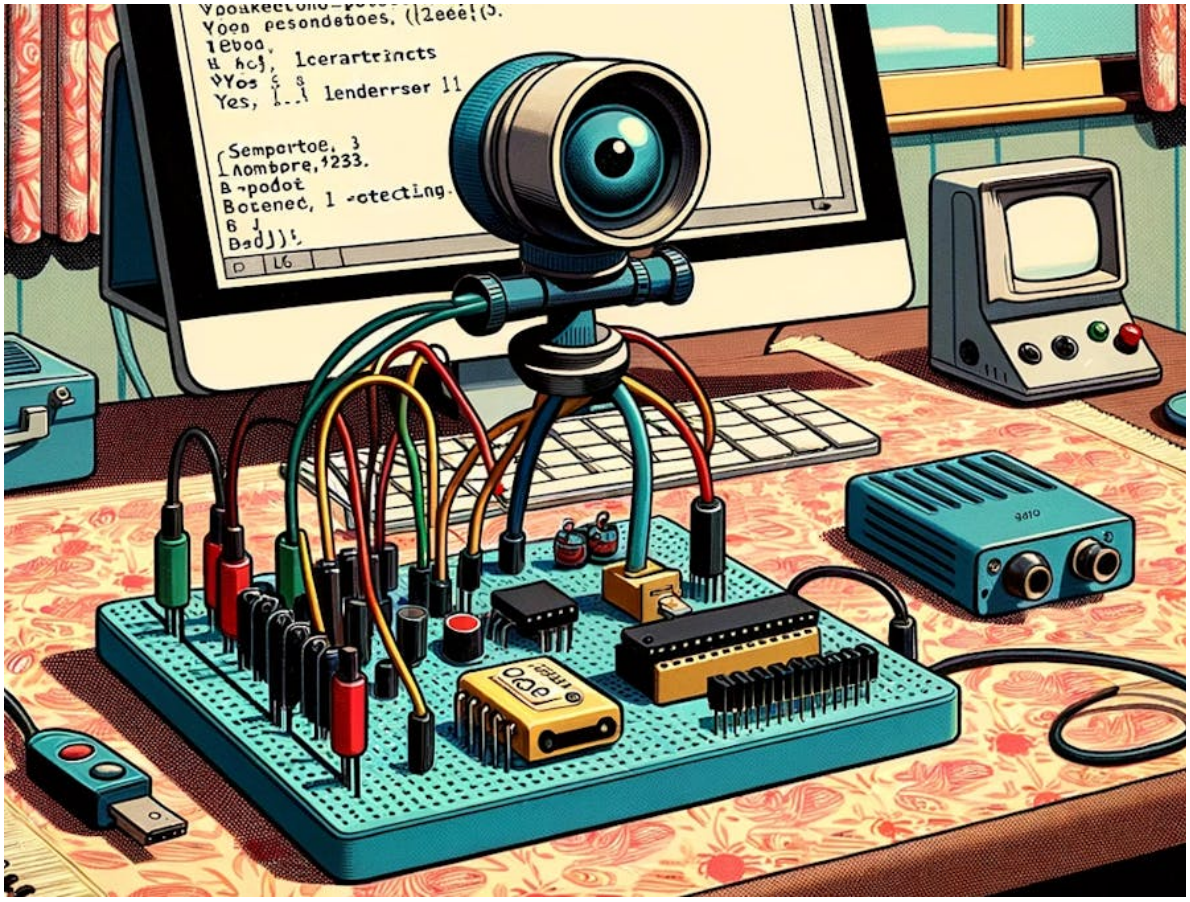
Here are other screenshots:

2.7 Conclusion

The XIAO ESP32S3 Sense is very flexible, inexpensive, and easy to program. The project proves the potential of TinyML. Memory is not an issue; the device can handle many post-processing tasks, including communication.

You will find the last version of the codes on the GitHub repository: XIAO-ESP32S3-Sense.

3 Object Detection



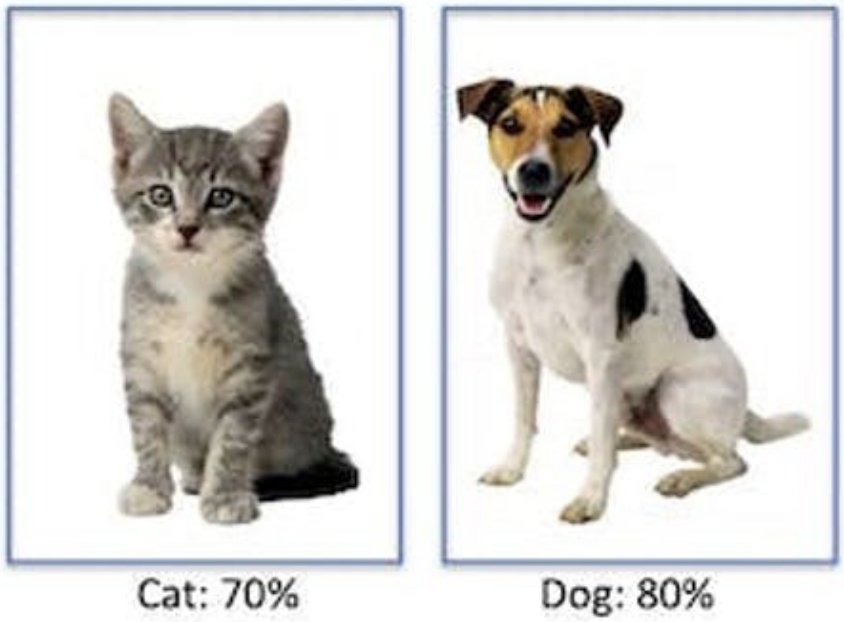
DALL·E prompt - Cartoon styled after 1950s animations, showing a detailed board with sensors, particularly a camera, on a table with patterned cloth. Behind the board, a comp with a large back showcases the Arduino IDE. The IDE's content hints at LED pin assignments and machine learning inference for detecting spoken commands. The Seria Monitor, in a distinct window, reveals outputs for the commands 'yes' and 'no'.

3.1 Introduction

In the last section regarding Computer Vision (CV) and the XIAO ESP32S3, *Image Classification,* we learned how to set up and classify images with this remarkable development board. Continuing our CV journey, we will explore **Object Detection** on microcontrollers.

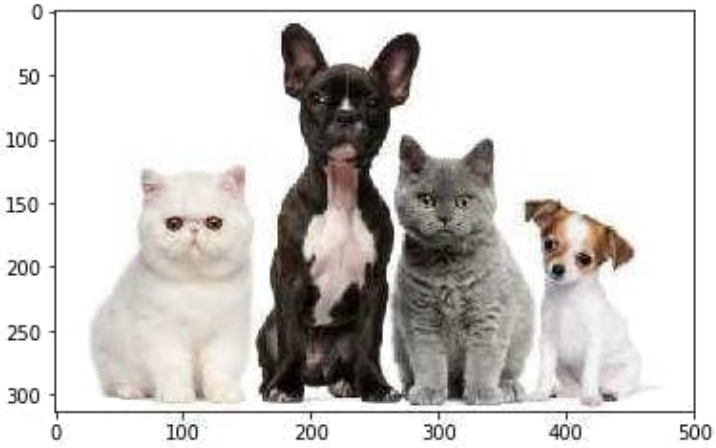
3.1.1 Object Detection versus Image Classification

The main task with Image Classification models is to identify the most probable object category present on an image, for example, to classify between a cat or a dog, dominant “objects” in an image:



But what happens if there is no dominant category in the image?

```
[PREDICTION]        [Prob]

ashcan              : 27%
Egyptian cat        : 19%
hamper              : 13%
```

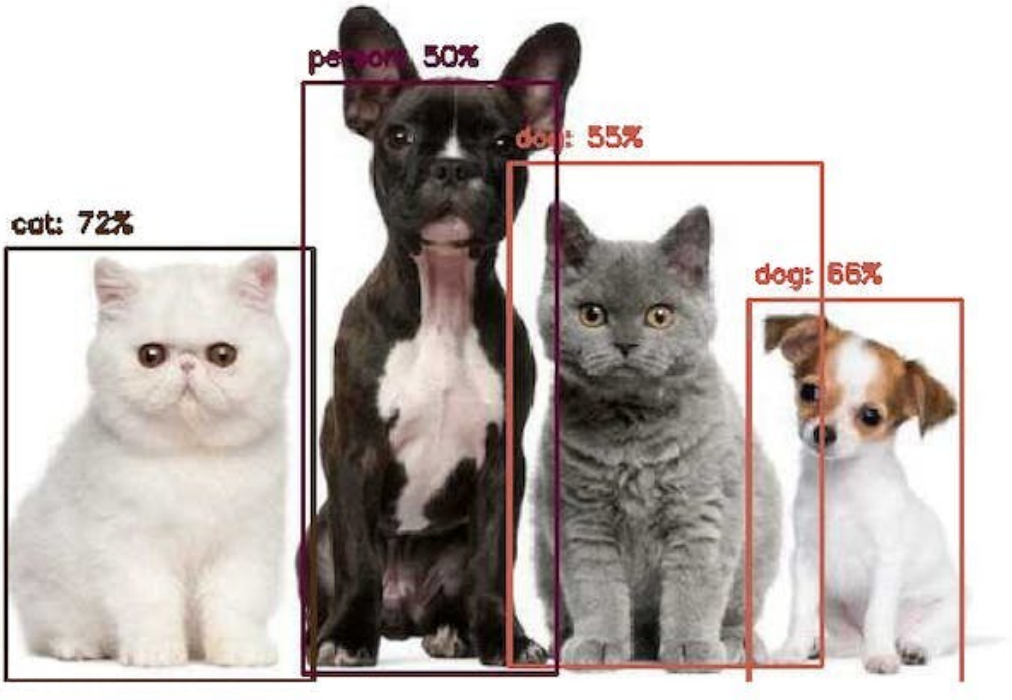


An image classification model identifies the above image utterly wrong as an “ashcan,” possibly due to the color tonalities.

> The model used in the previous images is MobileNet, which is trained with a large dataset, *ImageNet*, running on a Raspberry Pi.

To solve this issue, we need another type of model, where not only **multiple categories** (or labels) can be found but also **where** the objects are located on a given image.

As we can imagine, such models are much more complicated and bigger, for example, the **MobileNetV2 SSD FPN-Lite 320x320, trained with the COCO dataset.** This pre-trained object detection model is designed to locate up to 10 objects within an image, outputting a bounding box for each object detected. The below image is the result of such a model running on a Raspberry Pi:



Those models used for object detection (such as the MobileNet SSD or YOLO) usually have several MB in size, which is OK for use with Raspberry Pi but unsuitable for use with embedded devices, where the RAM usually has, at most, a few MB as in the case of the XIAO ESP32S3.

3.1.2 An Innovative Solution for Object Detection: FOMO

Edge Impulse launched in 2022, **FOMO** (Faster Objects, More Objects), a novel solution to perform object detection on embedded devices, such as the Nicla Vision and Portenta (Cortex M7), on Cortex M4F CPUs (Arduino Nano33 and OpenMV M4 series) as well the Espressif ESP32 devices (ESP-CAM, ESP-EYE and XIAO ESP32S3 Sense).

In this Hands-On project, we will explore Object Detection using FOMO.

> To understand more about FOMO, you can go into the official FOMO announcement by Edge Impulse, where Louis Moreau and Mat Kelcey explain in detail how it works.

3.2 The Object Detection Project Goal

All Machine Learning projects need to start with a detailed goal. Let's assume we are in an industrial or rural facility and must sort and count **oranges (fruits)** and particular **frogs (bugs)**.

In other words, we should perform a multi-label classification, where each image can have three classes:

- Background (No objects)
- Fruit
- Bug

Here are some not labeled image samples that we should use to detect the objects (fruits and bugs):

We are interested in which object is in the image, its location (centroid), and how many we can find on it. The object's size is not detected with FOMO, as with MobileNet SSD or YOLO, where the Bounding Box is one of the model outputs.

We will develop the project using the XIAO ESP32S3 for image capture and model inference. The ML project will be developed using the Edge Impulse Studio. But before starting the object detection project in the Studio, let's create a *raw dataset* (not labeled) with images that contain the objects to be detected.

3.3 Data Collection

You can capture images using the XIAO, your phone, or other devices. Here, we will use the XIAO with code from the Arduino IDE ESP32 library.

3.3.1 Collecting Dataset with the XIAO ESP32S3

Open the Arduino IDE and select the XIAO_ESP32S3 board (and the port where it is connected). On `File > Examples > ESP32 > Camera`, select `CameraWebServer`.

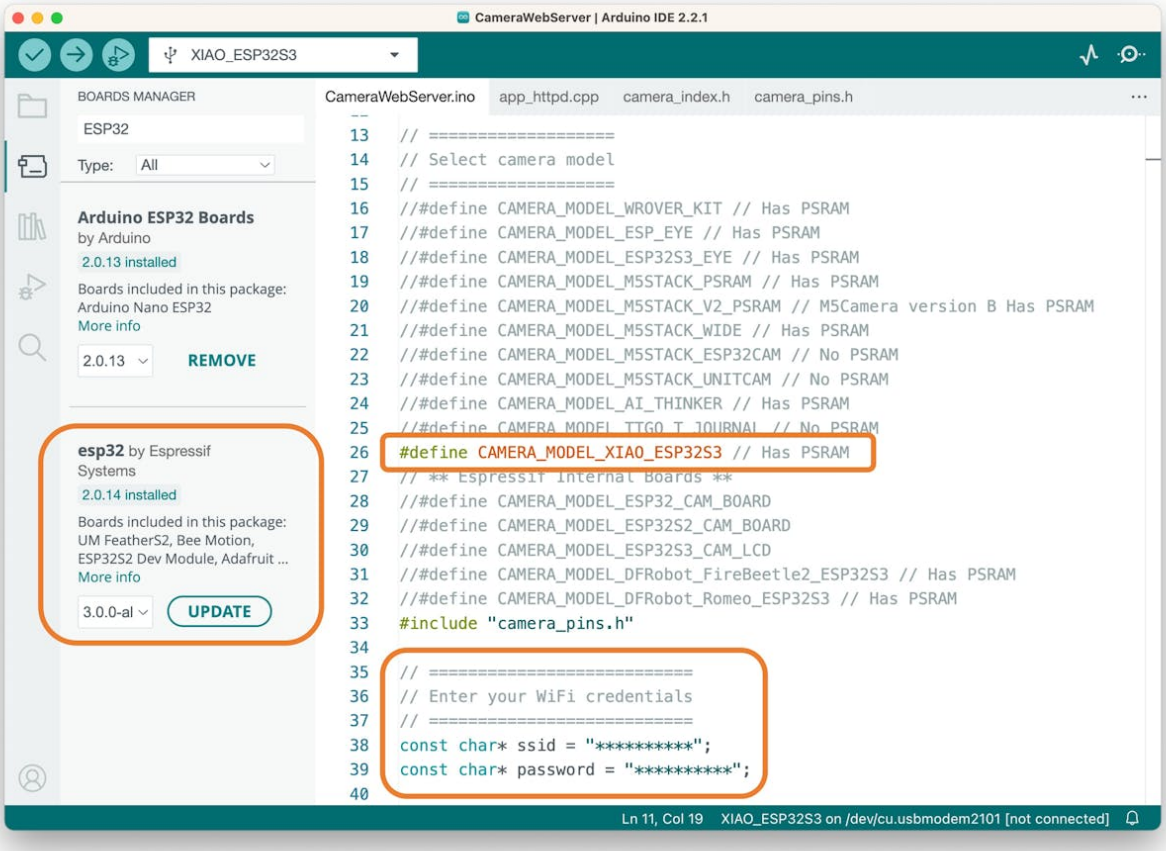
On the BOARDS MANAGER panel, confirm that you have installed the latest “stable” package.

> ⚠️ **Attention**
>
> Alpha versions (for example, 3.x-alpha) do not work correctly with the XIAO and Edge Impulse. Use the last stable version (for example, 2.0.11) instead.

You also should comment on all cameras’ models, except the XIAO model pins:

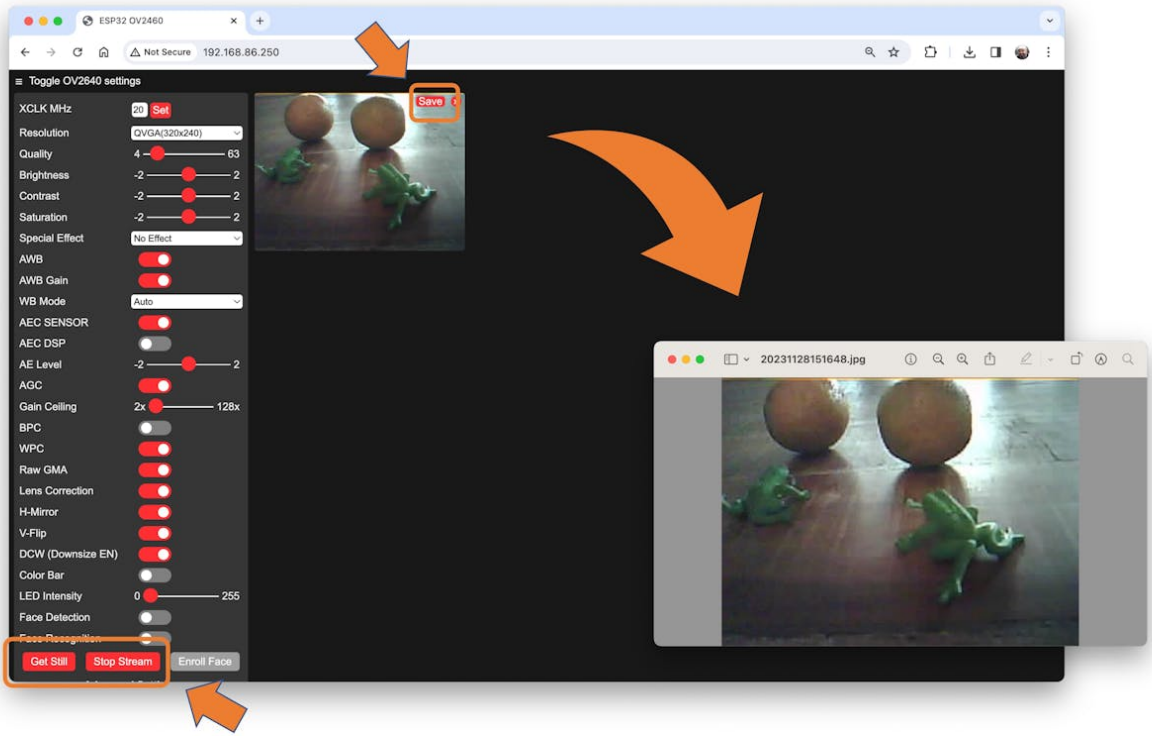
```
#define CAMERA_MODEL_XIAO_ESP32S3 // Has PSRAM
```

And on `Tools`, enable the PSRAM. Enter your wifi credentials and upload the code to the device:



If the code is executed correctly, you should see the address on the Serial Monitor:

Copy the address on your browser and wait for the page to be uploaded. Select the camera resolution (for example, QVGA) and select `[START STREAM]`. Wait for a few seconds/minutes, depending on your connection. You can save an image on your computer download area using the [Save] button.



Edge impulse suggests that the objects should be similar in size and not overlapping for better performance. This is OK in an industrial facility, where the camera should be fixed, keeping the same distance from the objects to be detected. Despite that, we will also try using mixed sizes and positions to see the result.

> We do not need to create separate folders for our images because each contains multiple labels.

We suggest using around 50 images to mix the objects and vary the number of each appearing on the scene. Try to capture different angles, backgrounds, and light conditions.

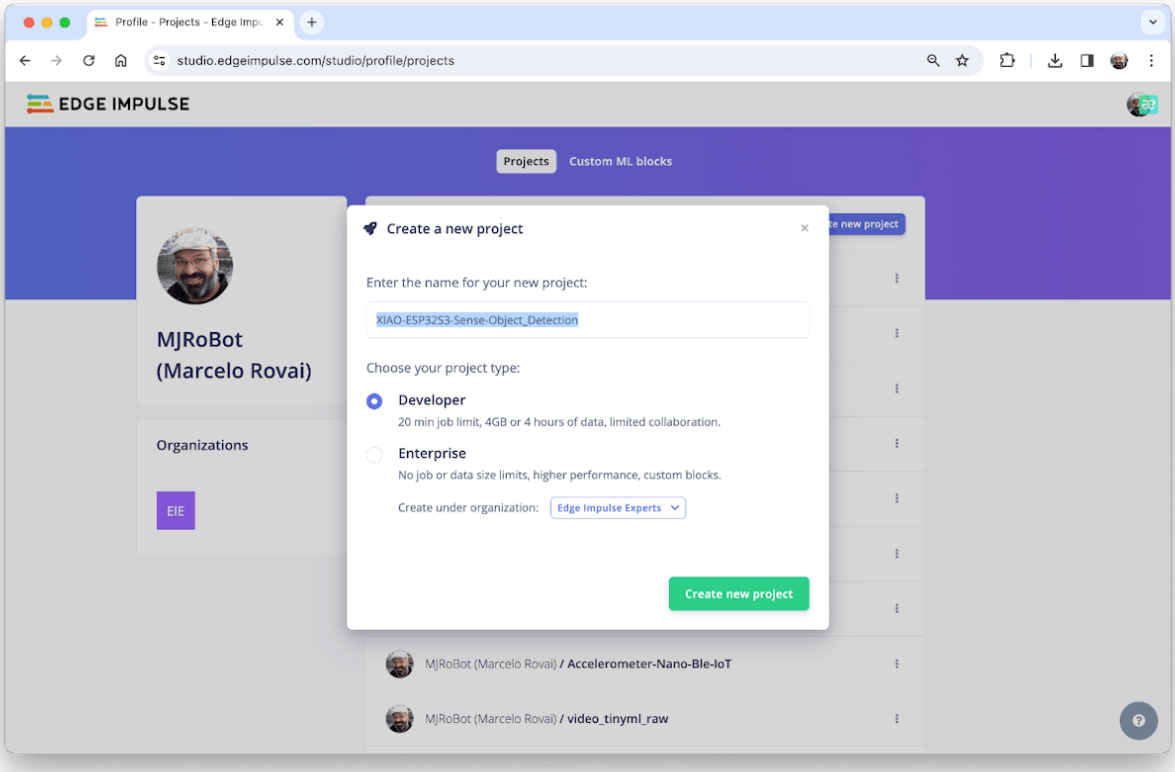
> The stored images use a QVGA frame size of 320x240 and RGB565 (color pixel format).

After capturing your dataset, `[Stop Stream]` and move your images to a folder.

3.4 Edge Impulse Studio

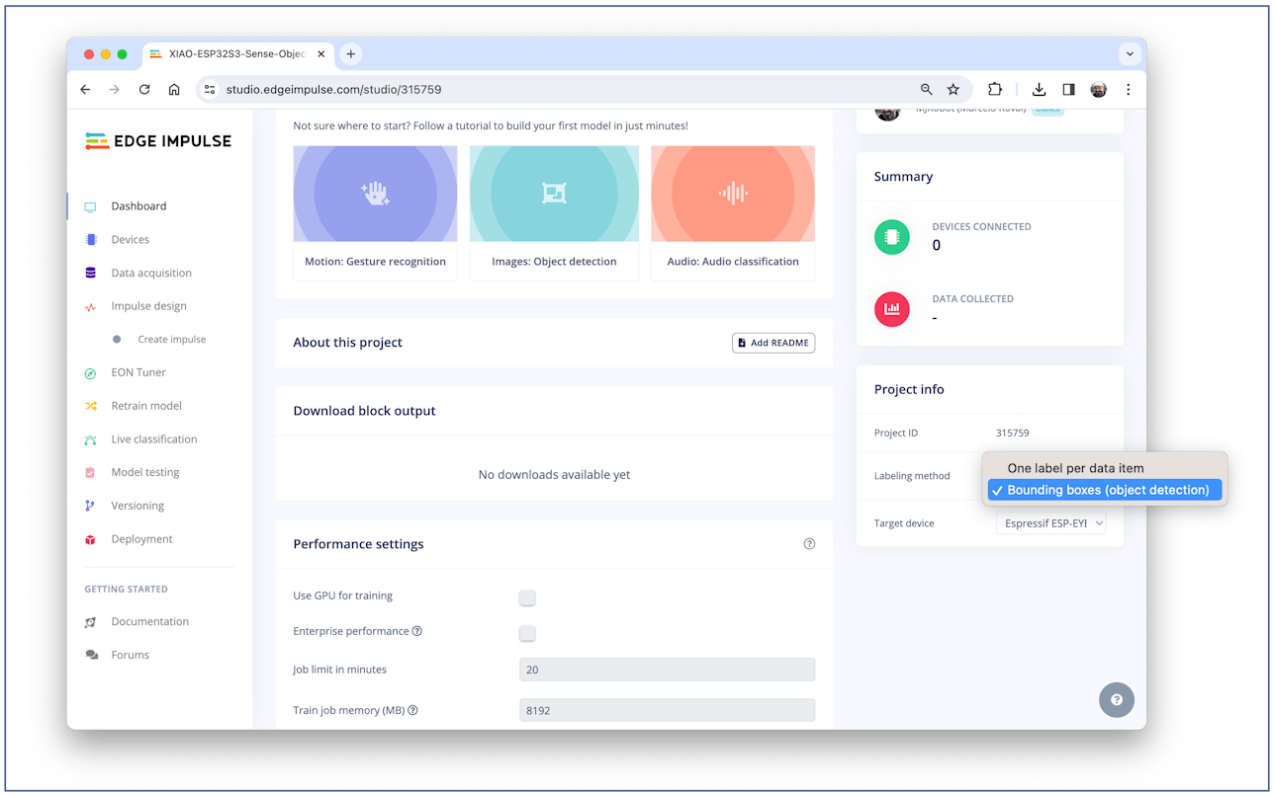
3.4.1 Setup the project

Go to Edge Impulse Studio, enter your credentials at **Login** (or create an account), and start a new project.



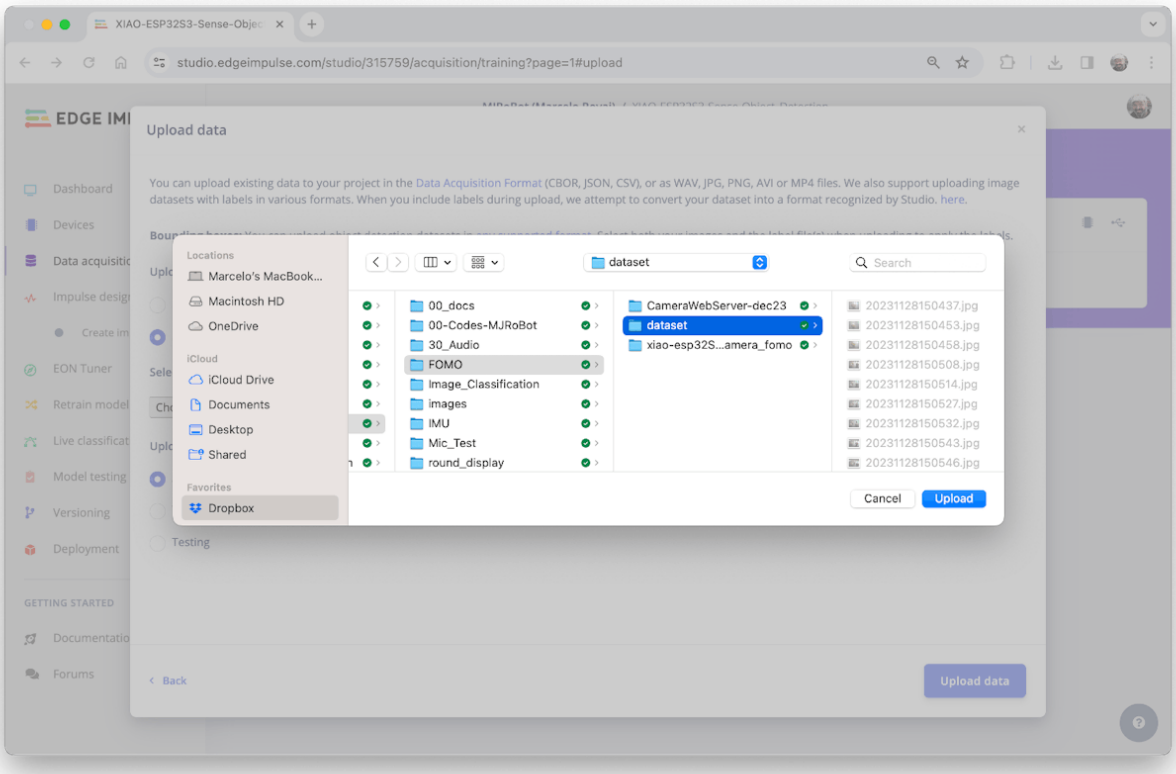
Here, you can clone the project developed for this hands-on: XIAO-ESP32S3-Sense-Object_Detection

On your Project Dashboard, go down and on **Project info** and select **Bounding boxes (object detection)** and **Espressif ESP-EYE** (most similar to our board) as your Target Device:

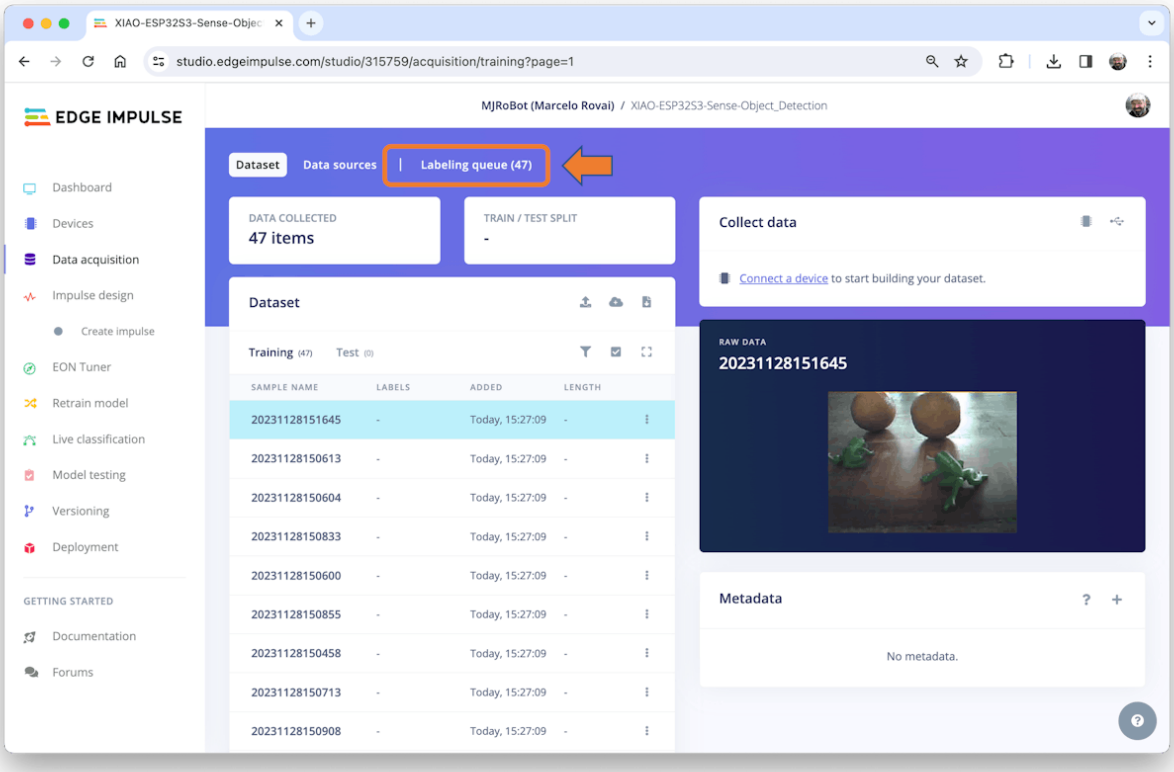


3.4.2 Uploading the unlabeled data

On Studio, go to the `Data acquisition` tab, and on the `UPLOAD DATA` section, upload files captured as a folder from your computer.



You can leave for the Studio to split your data automatically between Train and Test or do it manually. We will upload all of them as training.



All the not-labeled images (47) were uploaded but must be labeled appropriately before being used as a project dataset. The Studio has a tool for that purpose, which you can find in the link Labeling queue (47).

There are two ways you can use to perform AI-assisted labeling on the Edge Impulse Studio (free version):

- Using yolov5
- Tracking objects between frames

Edge Impulse launched an auto-labeling feature for Enterprise customers, easing labeling tasks in object detection projects.

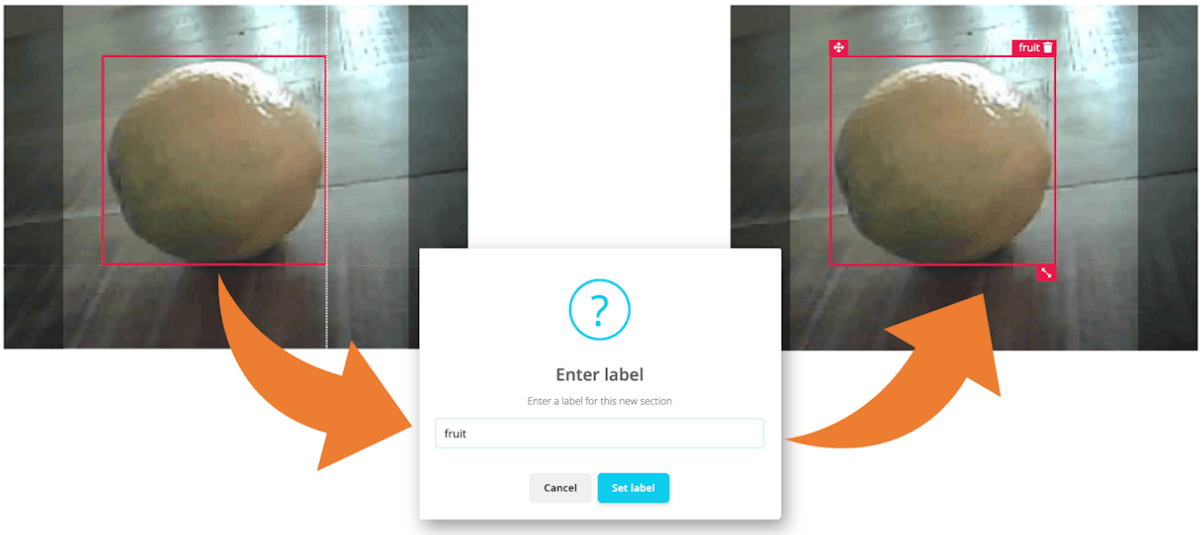
Ordinary objects can quickly be identified and labeled using an existing library of pre-trained object detection models from YOLOv5 (trained with the COCO dataset). But since, in our case, the objects are not part of COCO datasets, we should select the option of tracking objects. With this option, once you draw bounding boxes and label the images in one frame, the

objects will be tracked automatically from frame to frame, *partially* labeling the new ones (not all are correctly labeled).

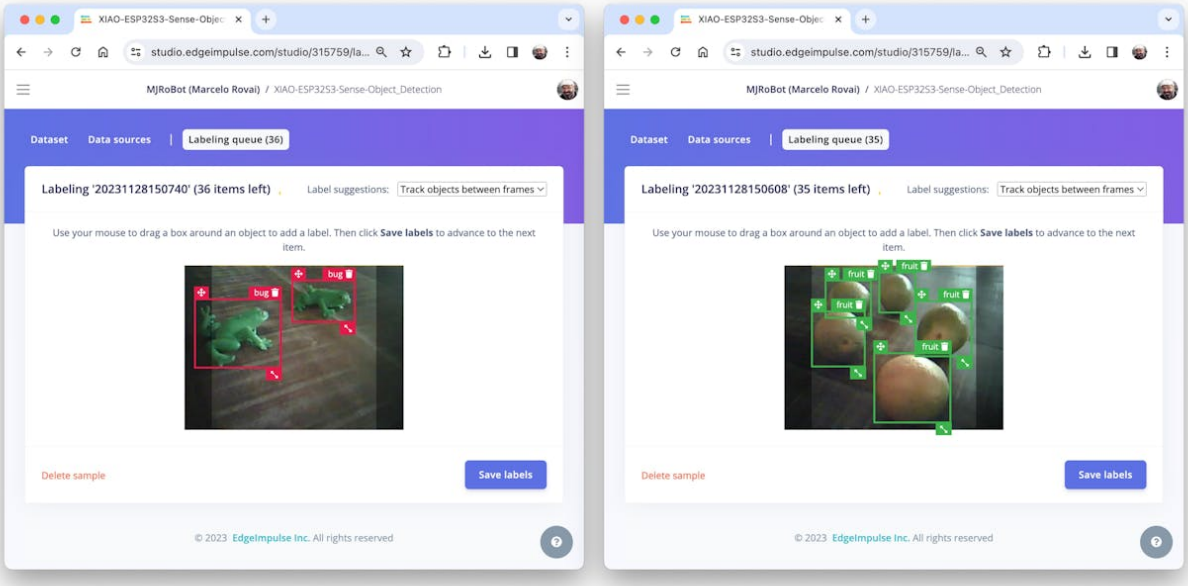
> You can use the EI uploader to import your data if you already have a labeled dataset containing bounding boxes.

3.4.3 Labeling the Dataset

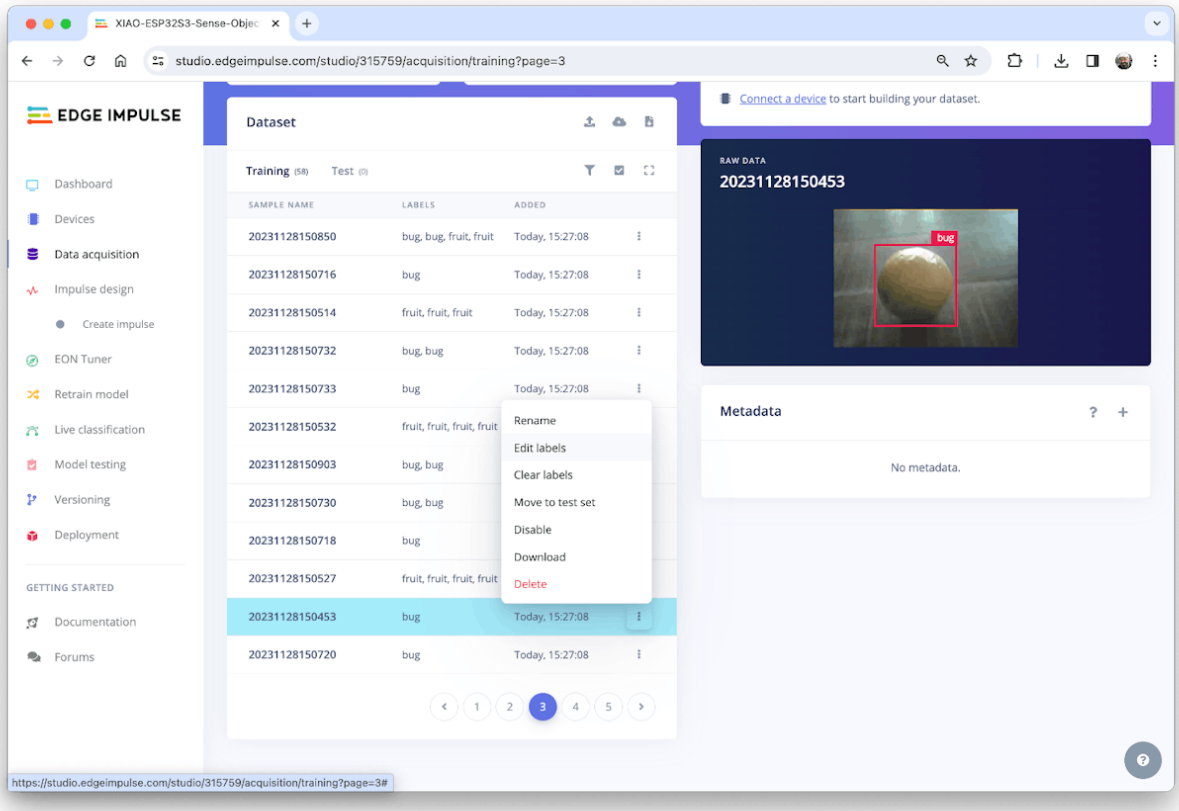
Starting with the first image of your unlabeled data, use your mouse to drag a box around an object to add a label. Then click **Save labels** to advance to the next item.



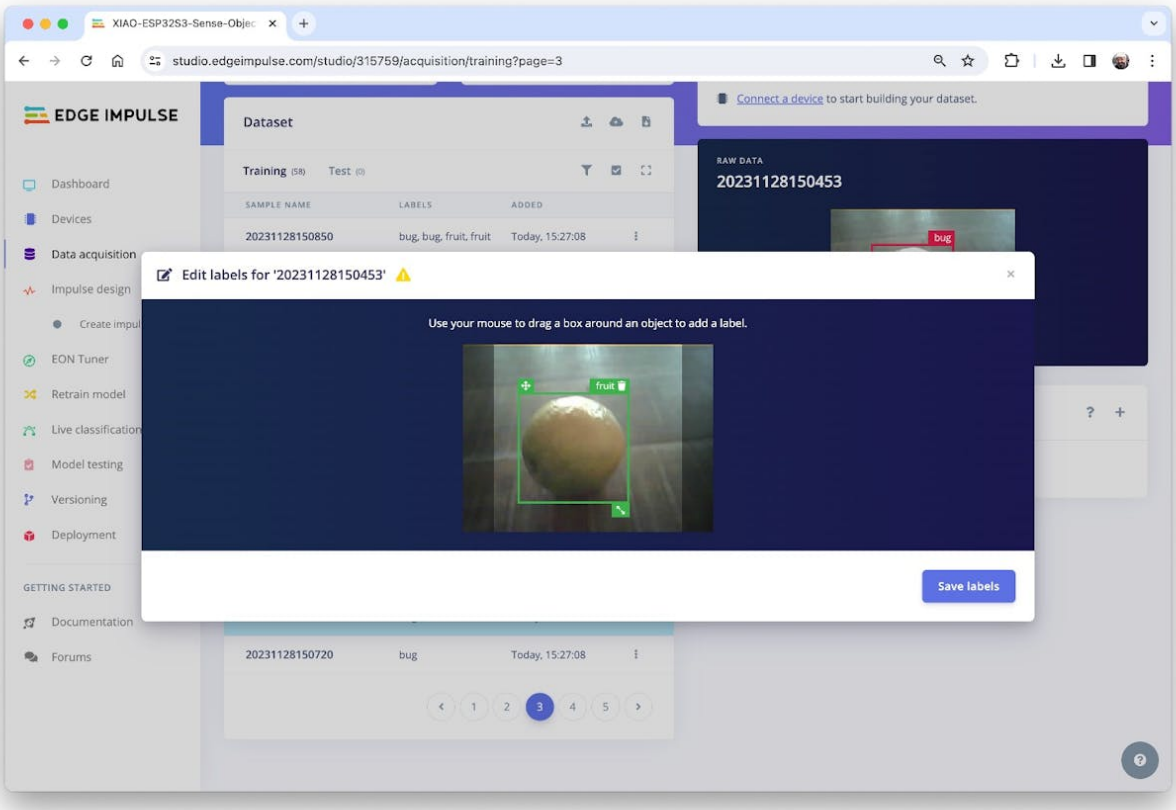
Continue with this process until the queue is empty. At the end, all images should have the objects labeled as those samples below:



Next, review the labeled samples on the `Data acquisition` tab. If one of the labels is wrong, you can edit it using the *three dots* menu after the sample name:

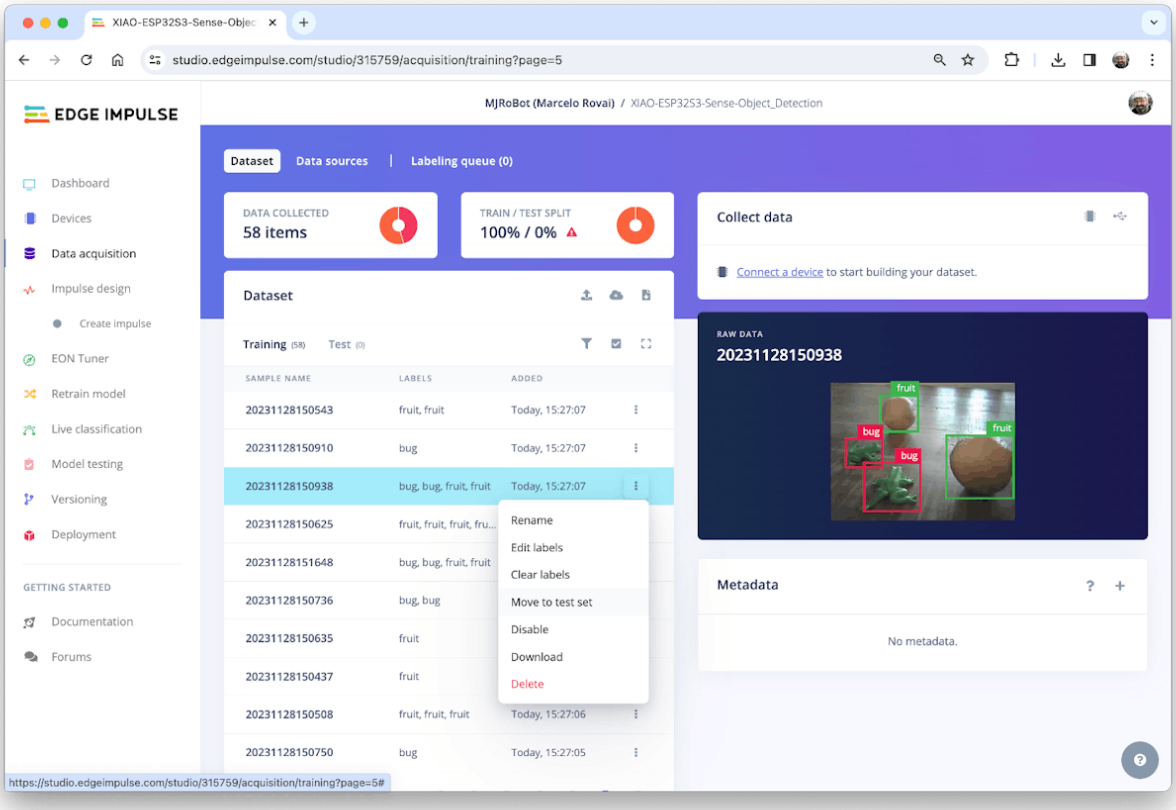


You will be guided to replace the wrong label and correct the dataset.



3.4.4 Balancing the dataset and split Train/Test

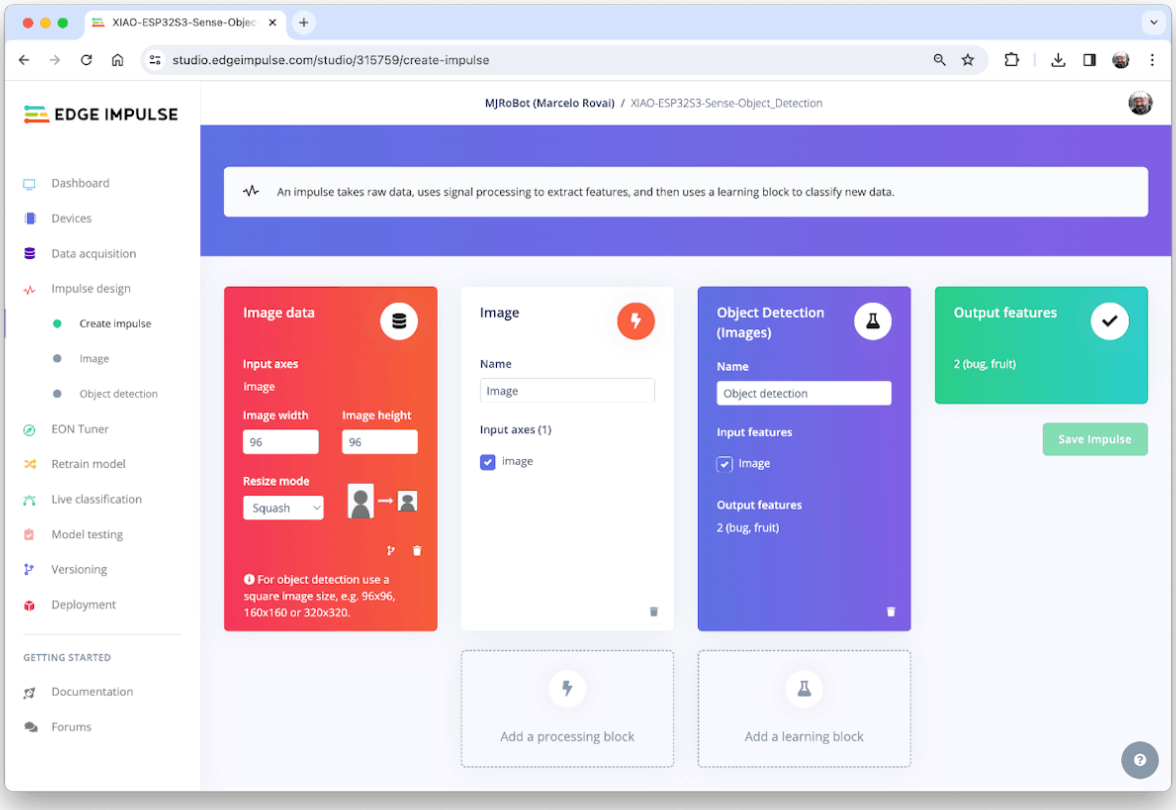
After labeling all data, it was realized that the class fruit had many more samples than the bug. So, 11 new and additional bug images were collected (ending with 58 images). After labeling them, it is time to select some images and move them to the test dataset. You can do it using the three-dot menu after the image name. I selected six images, representing 13% of the total dataset.



3.5 The Impulse Design

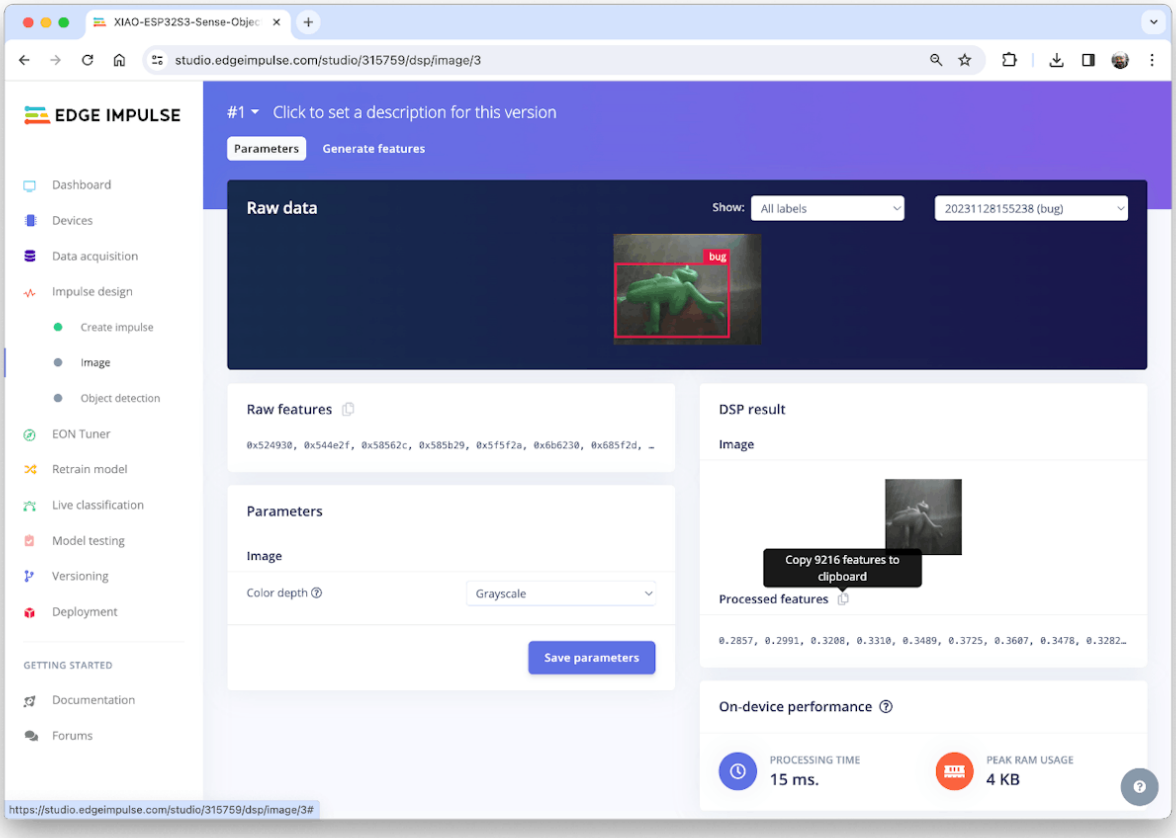
In this phase, you should define how to:

- **Pre-processing** consists of resizing the individual images from 320 x 240 to 96 x 96 and squashing them (squared form, without cropping). Afterward, the images are converted from RGB to Grayscale.
- **Design a Model,** in this case, “Object Detection.”

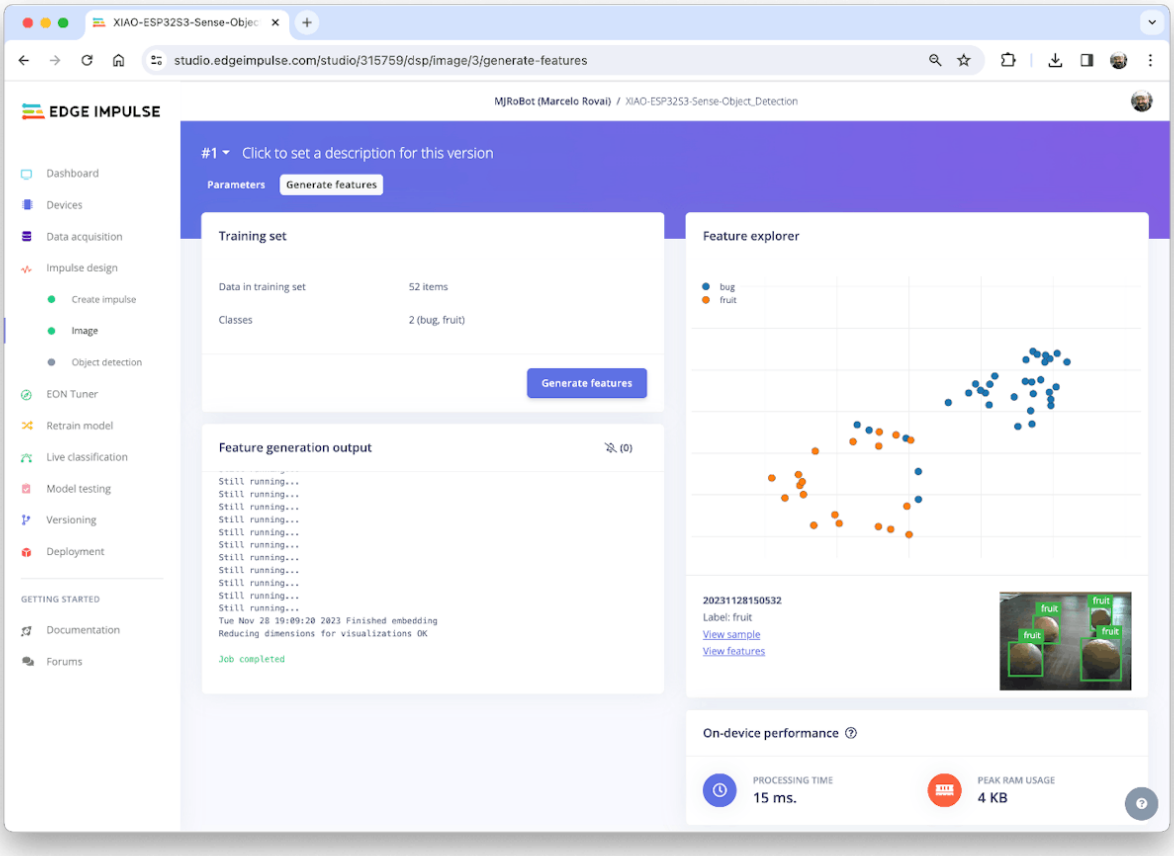


3.5.1 Preprocessing all dataset

In this section, select **Color depth** as Grayscale, suitable for use with FOMO models and Save parameters.



The Studio moves automatically to the next section, Generate features, where all samples will be pre-processed, resulting in a dataset with individual 96x96x1 images or 9,216 features.



The feature explorer shows that all samples evidence a good separation after the feature generation.

> Some samples seem to be in the wrong space, but clicking on them confirms the correct labeling.

3.6 Model Design, Training, and Test

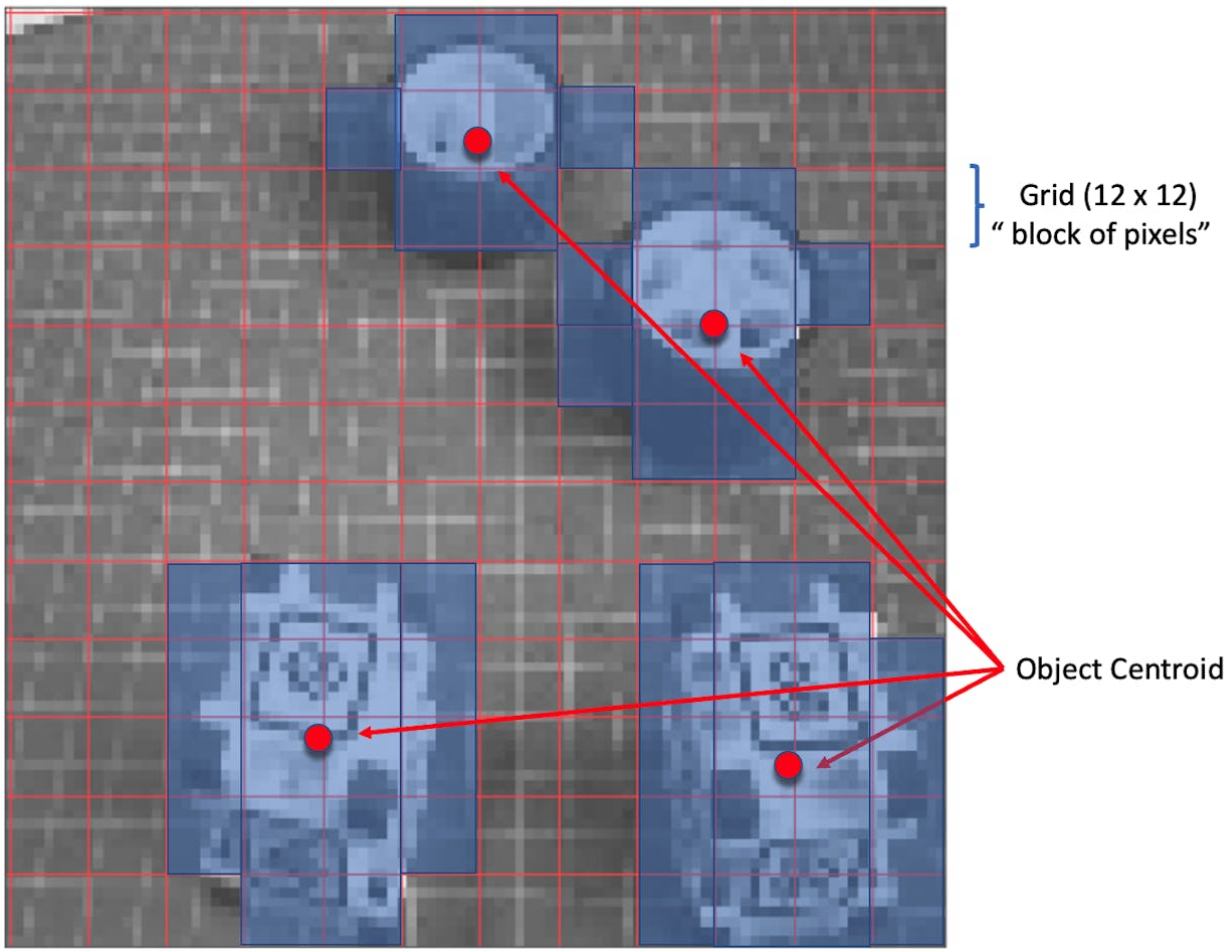
We will use FOMO, an object detection model based on MobileNetV2 (alpha 0.35) designed to coarsely segment an image into a grid of **background** vs **objects of interest** (here, *boxes* and *wheels*).

FOMO is an innovative machine learning model for object detection, which can use up to 30 times less energy and memory than traditional models like Mobilenet SSD and YOLOv5. FOMO can operate on microcontrollers with less than 200 KB of RAM. The main reason this is possible is that while other

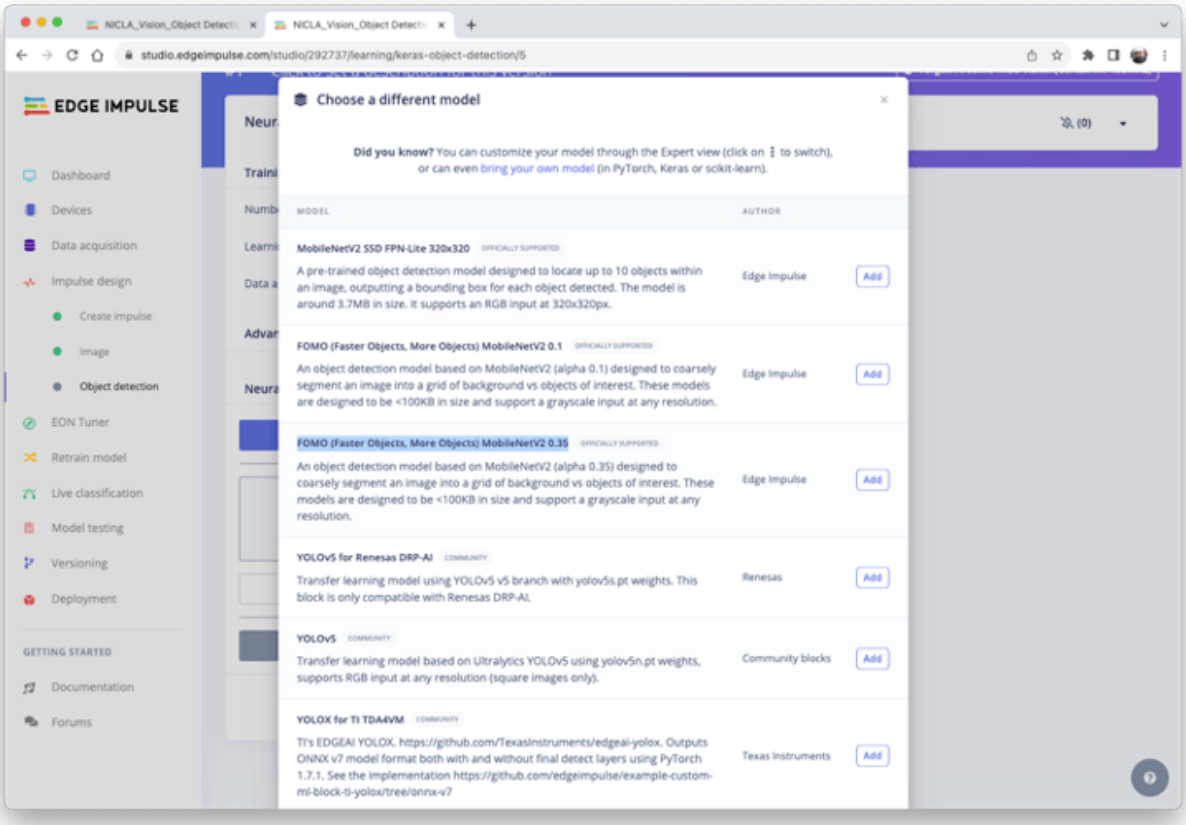
models calculate the object's size by drawing a square around it (bounding box), FOMO ignores the size of the image, providing only the information about where the object is located in the image through its centroid coordinates.

How FOMO works?

FOMO takes the image in grayscale and divides it into blocks of pixels using a factor of 8. For the input of 96x96, the grid would be 12x12 (96/8=12). Next, FOMO will run a classifier through each pixel block to calculate the probability that there is a box or a wheel in each of them and, subsequently, determine the regions that have the highest probability of containing the object (If a pixel block has no objects, it will be classified as *background*). From the overlap of the final region, the FOMO provides the coordinates (related to the image dimensions) of the centroid of this region.



For training, we should select a pre-trained model. Let's use the **FOMO (Faster Objects, More Objects) MobileNetV2 0.35.** This model uses around 250KB of RAM and 80KB of ROM (Flash), which suits well with our board.



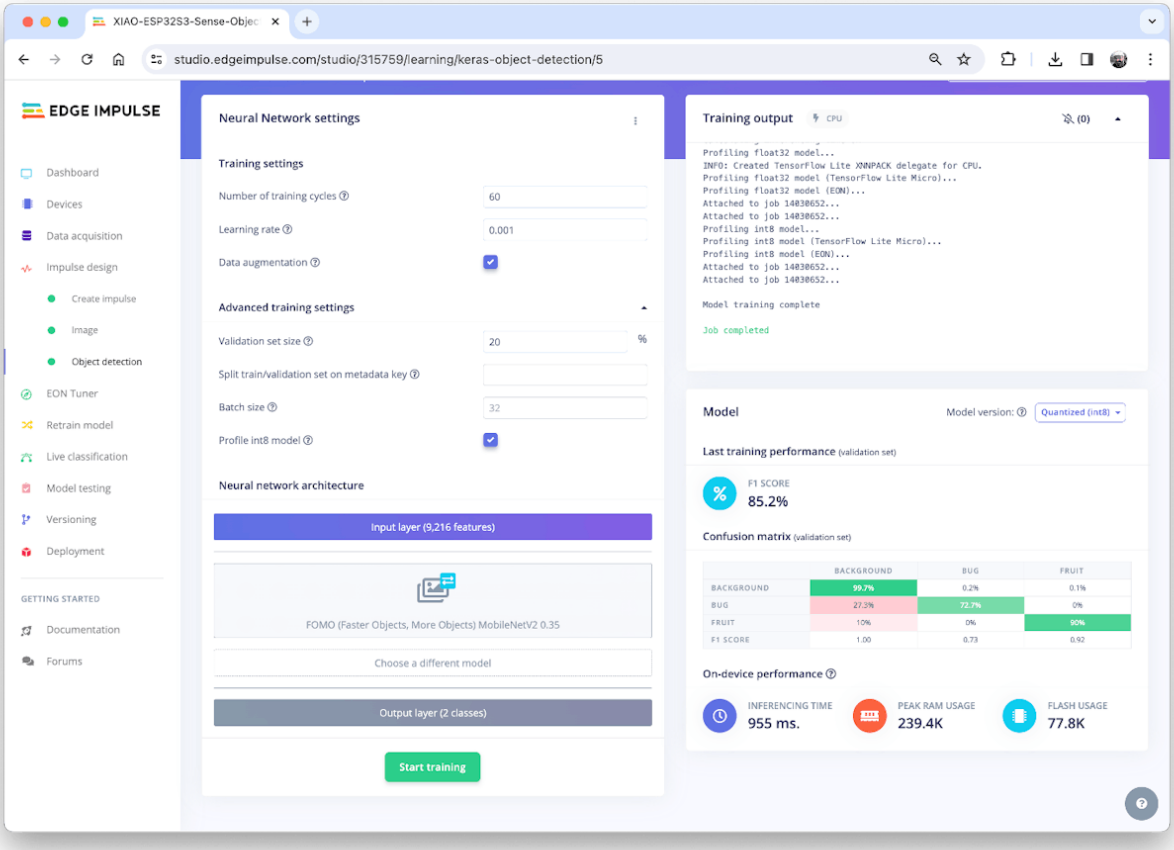
Regarding the training hyper-parameters, the model will be trained with:

- Epochs: 60
- Batch size: 32
- Learning Rate: 0.001.

For validation during training, 20% of the dataset (*validation_dataset*) will be spared. For the remaining 80% (*train_dataset*), we will apply Data Augmentation, which will randomly flip, change the size and brightness of the image, and crop them, artificially increasing the number of samples on the dataset for training.

As a result, the model ends with an overall F1 score of 85%, similar to the result when using the test data (83%).

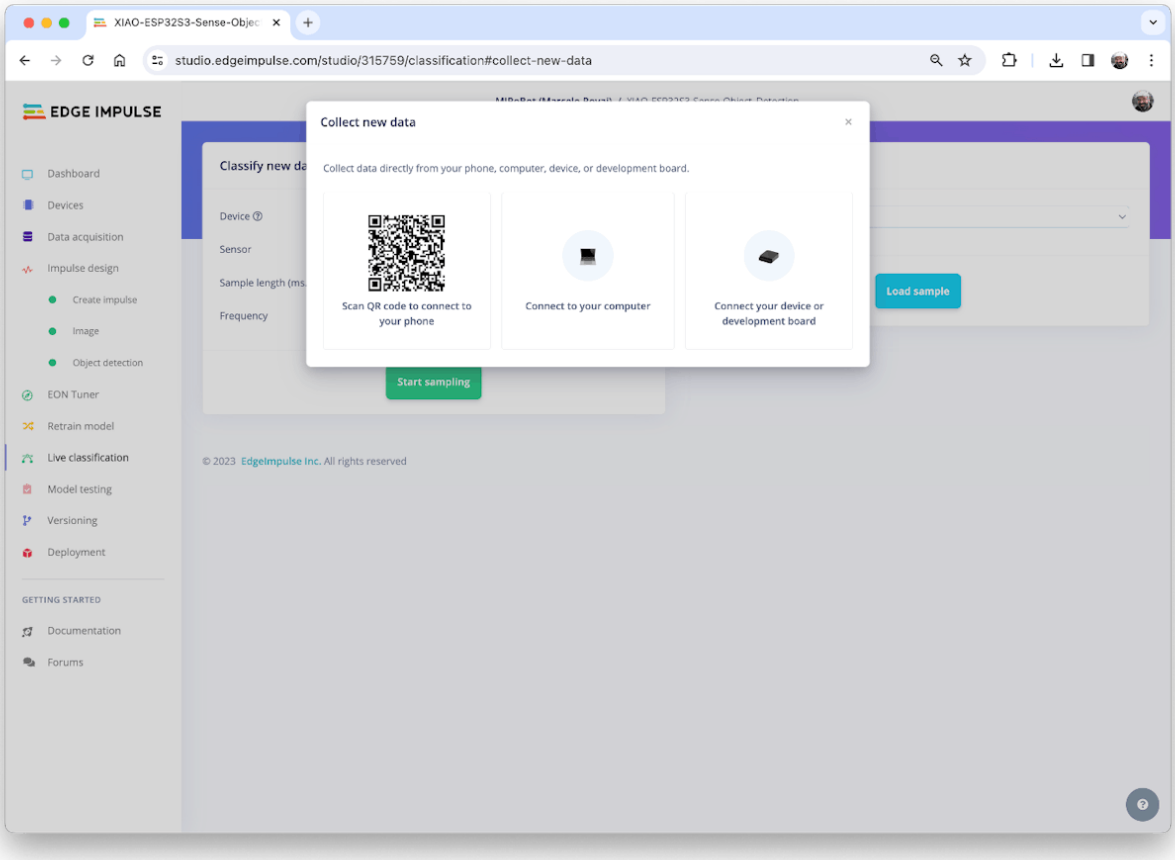
Note that FOMO automatically added a 3rd label background to the two previously defined (*box* and *wheel*).



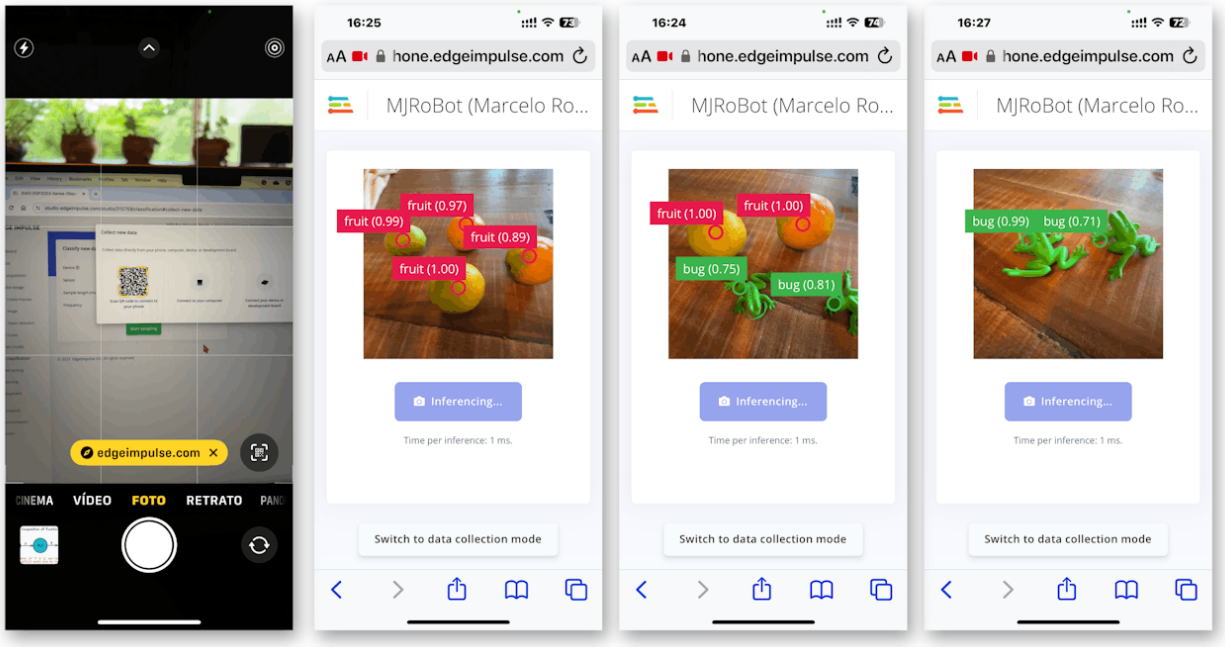
In object detection tasks, accuracy is generally not the primary evaluation metric. Object detection involves classifying objects and providing bounding boxes around them, making it a more complex problem than simple classification. The issue is that we do not have the bounding box, only the centroids. In short, using accuracy as a metric could be misleading and may not provide a complete understanding of how well the model is performing. Because of that, we will use the F1 score.

3.6.1 Test model with "Live Classification"

Once our model is trained, we can test it using the Live Classification tool. On the correspondent section, click on Connect a development board icon (a small MCU) and scan the QR code with your phone.



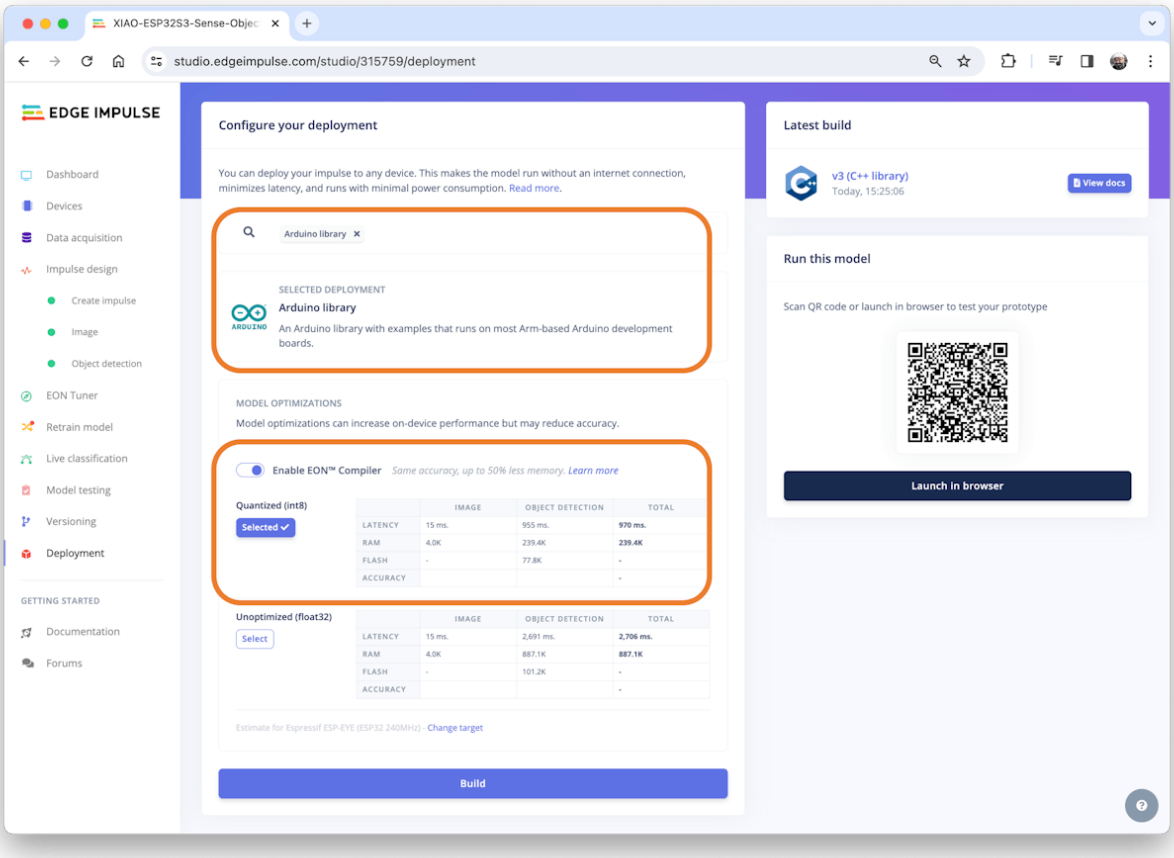
Once connected, you can use the smartphone to capture actual images to be tested by the trained model on Edge Impulse Studio.



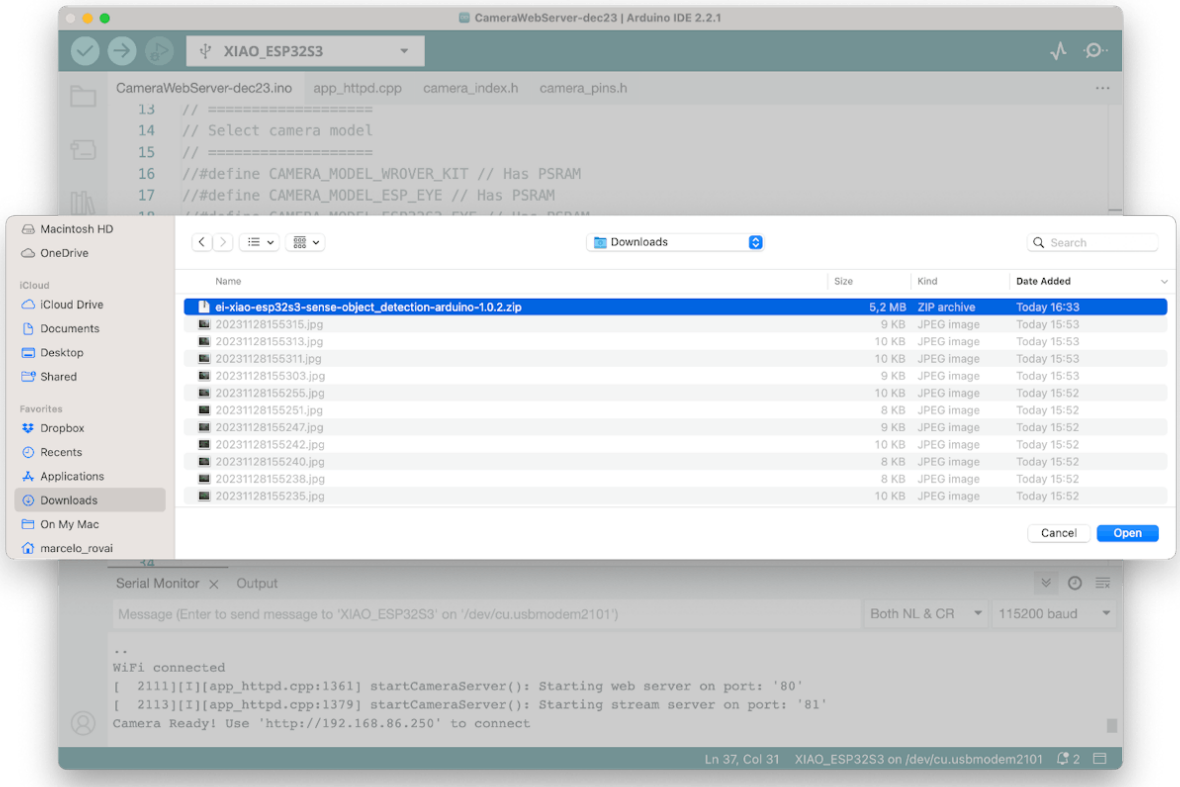
One thing to be noted is that the model can produce false positives and negatives. This can be minimized by defining a proper Confidence Threshold (use the Three dots menu for the setup). Try with 0.8 or more.

3.7 Deploying the Model (Arduino IDE)

Select the Arduino Library and Quantized (int8) model, enable the EON Compiler on the Deploy Tab, and press [Build].



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!



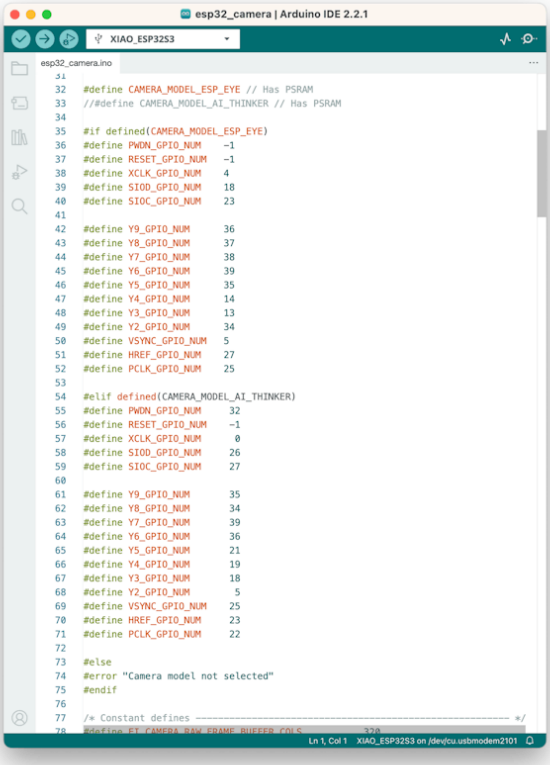
Under the Examples tab on Arduino IDE, you should find a sketch code (`esp32 > esp32_camera`) under your project name.

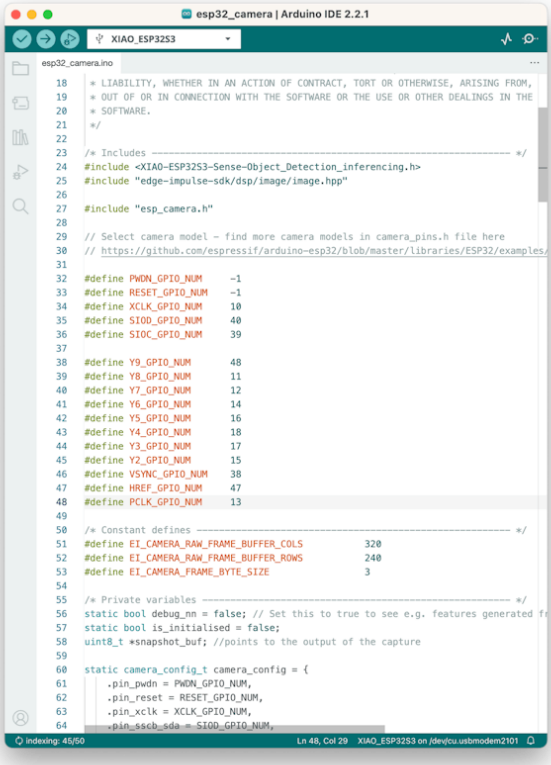
You should change lines 32 to 75, which define the camera model and pins, using the data related to our model. Copy and paste the below lines, replacing the lines 32-75:

```
#define PWDN_GPIO_NUM     -1
#define RESET_GPIO_NUM    -1
```

```
#define XCLK_GPIO_NUM     10
#define SIOD_GPIO_NUM     40
#define SIOC_GPIO_NUM     39
#define Y9_GPIO_NUM       48
#define Y8_GPIO_NUM       11
#define Y7_GPIO_NUM       12
#define Y6_GPIO_NUM       14
#define Y5_GPIO_NUM       16
#define Y4_GPIO_NUM       18
#define Y3_GPIO_NUM       17
#define Y2_GPIO_NUM       15
#define VSYNC_GPIO_NUM    38
#define HREF_GPIO_NUM     47
#define PCLK_GPIO_NUM     13
```

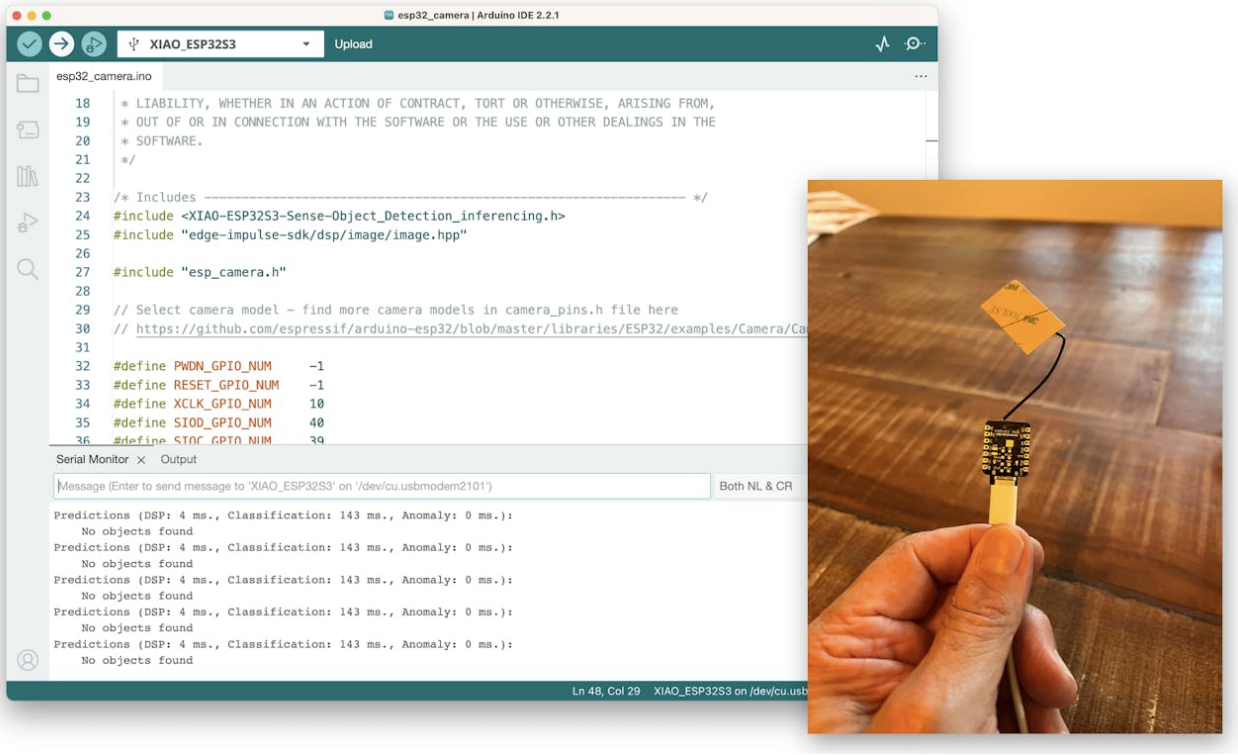
Here you can see the resulting code:



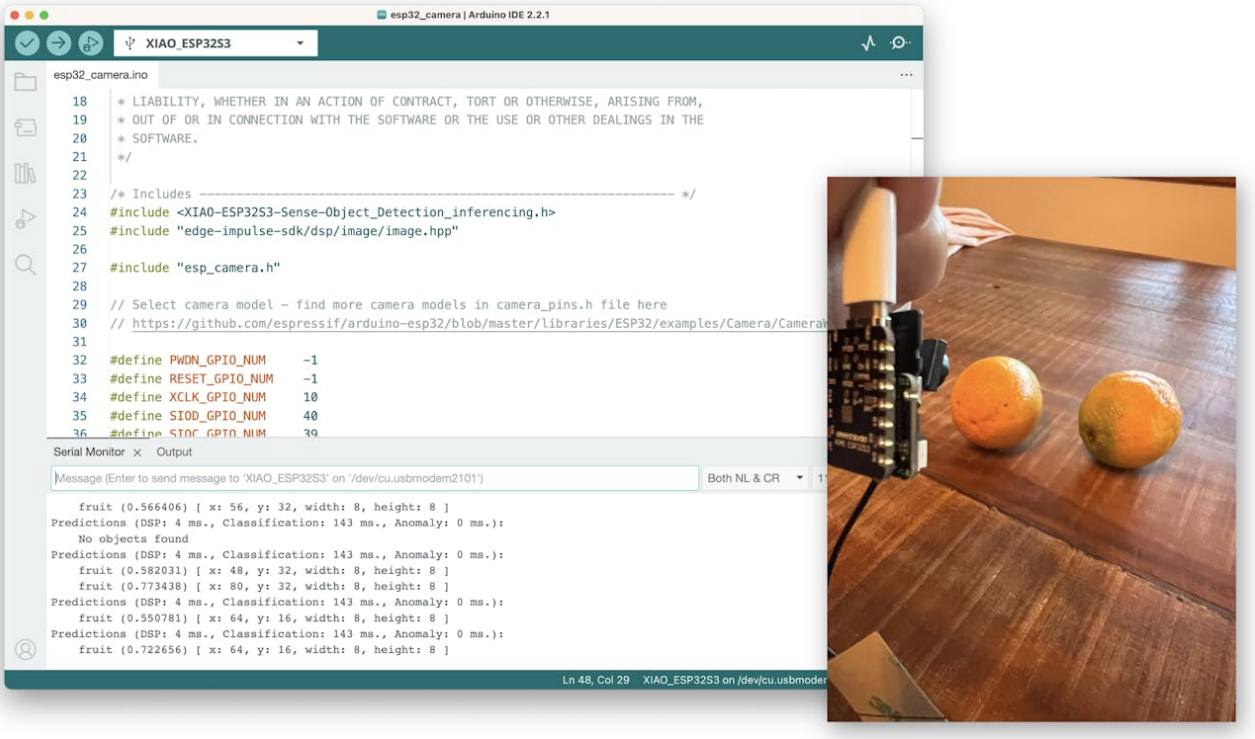


Upload the code to your XIAO ESP32S3 Sense, and you should be OK to start detecting fruits and bugs. You can check the result on Serial Monitor.

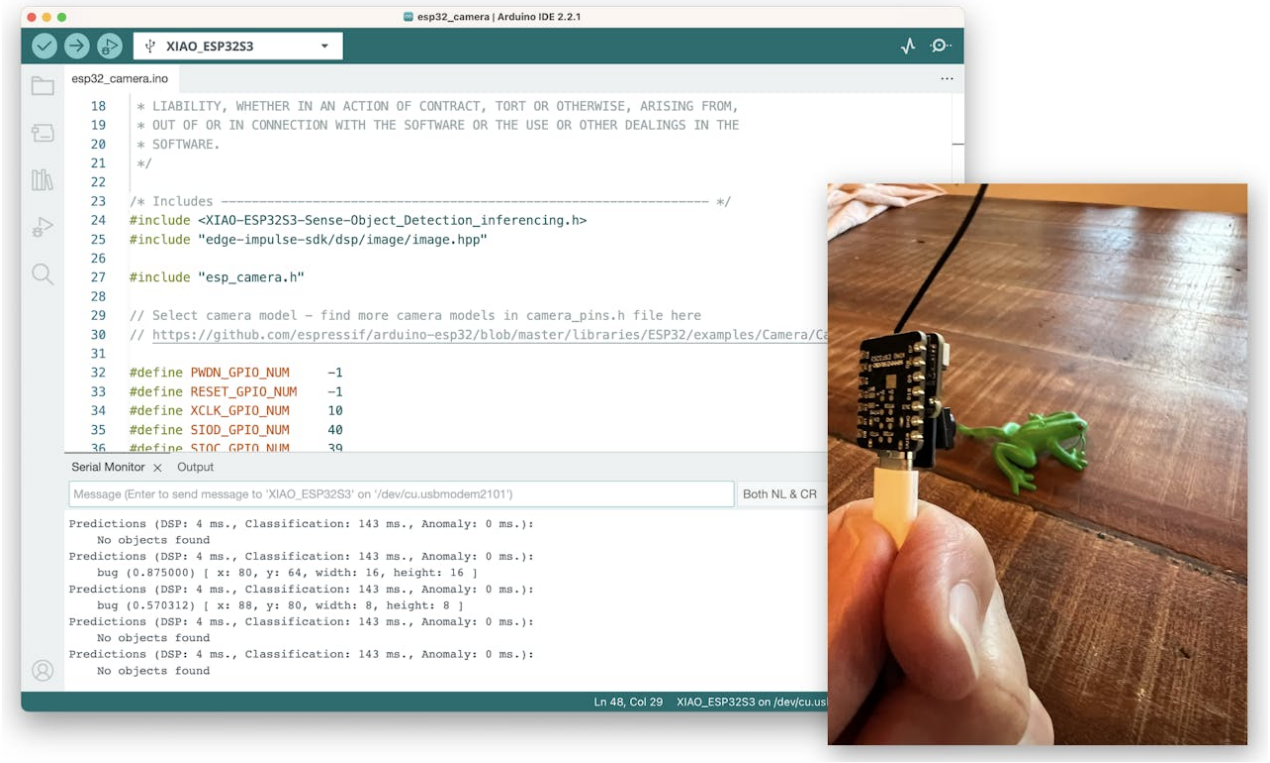
Background


esp32_camera | Arduino IDE 2.2.1
XIAO_ESP32S3
Upload
esp32_camera.ino
* LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
* OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
* SOFTWARE.
*/
/* Includes ---------- */
#include <XIAO-ESP32S3-Sense-Object_Detection_inferencing.h>
#include "edge-impulse-sdk/dsp/image/image.hpp"
#include "esp_camera.h"
// Select camera model - find more camera models in camera_pins.h file here
// https://github.com/espressif/arduino-esp32/blob/master/libraries/ESP32/examples/Camera/Ca
#define PWDN_GPIO_NUM -1
#define RESET_GPIO_NUM -1
#define XCLK_GPIO_NUM 10
#define SIOD_GPIO_NUM 40
Serial Monitor
Output
Message (Enter to send message to 'XIAO_ESP32S3' on '/dev/cu.usbmodem2101')
Both NL & CR
Predictions (DSP: 4 ms., Classification: 143 ms., Anomaly: 0 ms.):
No objects found
Predictions (DSP: 4 ms., Classification: 143 ms., Anomaly: 0 ms.):
No objects found
Predictions (DSP: 4 ms., Classification: 143 ms., Anomaly: 0 ms.):
No objects found
Predictions (DSP: 4 ms., Classification: 143 ms., Anomaly: 0 ms.):
No objects found
Predictions (DSP: 4 ms., Classification: 143 ms., Anomaly: 0 ms.):
No objects found
Ln 48, Col 29
XIAO_ESP32S3 on /dev/cu.usb

Fruits


esp32_camera | Arduino IDE 2.2.1
XIAO_ESP32S3
esp32_camera.ino
* LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
* OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
* SOFTWARE.
*/
/* Includes ---------- */
#include <XIAO-ESP32S3-Sense-Object_Detection_inferencing.h>
#include "edge-impulse-sdk/dsp/image/image.hpp"
#include "esp_camera.h"
// Select camera model - find more camera models in camera_pins.h file here
// https://github.com/espressif/arduino-esp32/blob/master/libraries/ESP32/examples/Camera/CameraW
#define PWDN_GPIO_NUM -1
#define RESET_GPIO_NUM -1
#define XCLK_GPIO_NUM 10
#define SIOD_GPIO_NUM 40
Serial Monitor
Output
Message (Enter to send message to 'XIAO_ESP32S3' on '/dev/cu.usbmodem2101')
Both NL & CR
fruit (0.566406) [x: 56, y: 32, width: 8, height: 8]
Predictions (DSP: 4 ms., Classification: 143 ms., Anomaly: 0 ms.):
No objects found
Predictions (DSP: 4 ms., Classification: 143 ms., Anomaly: 0 ms.):
fruit (0.582031) [x: 48, y: 32, width: 8, height: 8]
fruit (0.773438) [x: 80, y: 32, width: 8, height: 8]
Predictions (DSP: 4 ms., Classification: 143 ms., Anomaly: 0 ms.):
fruit (0.550781) [x: 64, y: 16, width: 8, height: 8]
Predictions (DSP: 4 ms., Classification: 143 ms., Anomaly: 0 ms.):
fruit (0.722656) [x: 64, y: 16, width: 8, height: 8]
Ln 48, Col 29
XIAO_ESP32S3 on /dev/cu.usbmoden

Bugs



Note that the model latency is 143ms, and the frame rate per second is around 7 fps (similar to what we got with the Image Classification project). This happens because FOMO is cleverly built over a CNN model, not with an object detection model like the SSD MobileNet. For example, when running a MobileNetV2 SSD FPN-Lite 320x320 model on a Raspberry Pi 4, the latency is around five times higher (around 1.5 fps).

3.8 Deploying the Model (SenseCraft-Web-Toolkit)

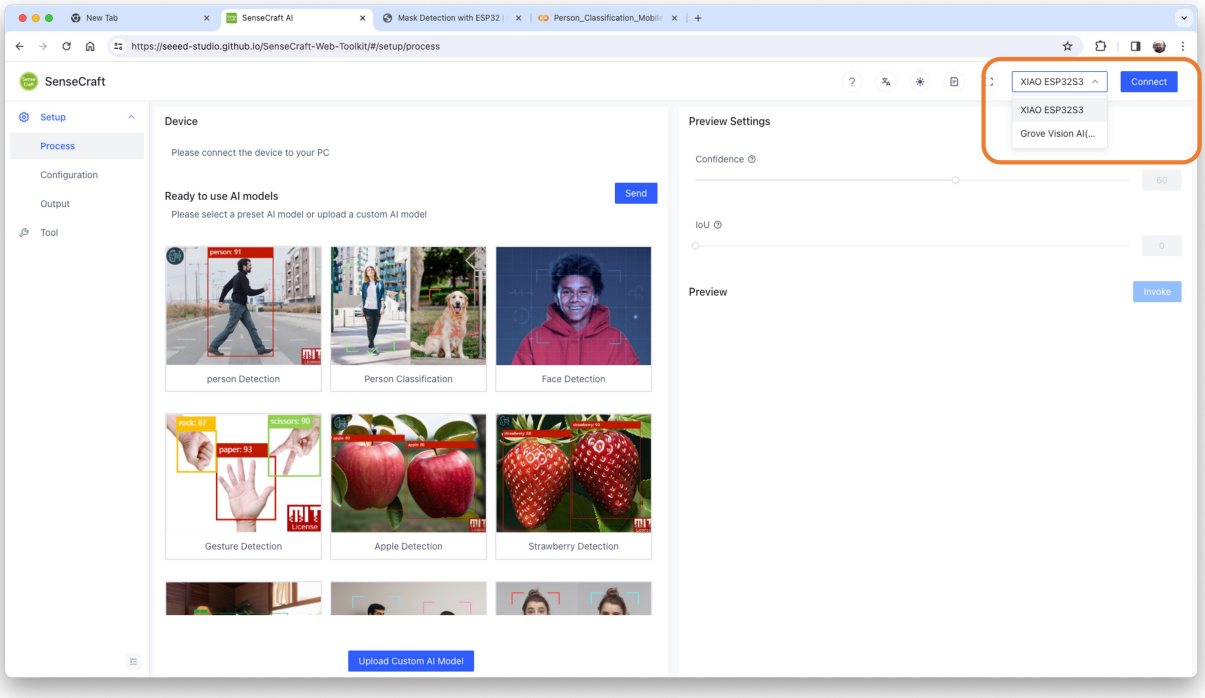
As discussed in the Image Classification chapter, verifying inference with Image models on Arduino IDE is very challenging because we can not see what the camera focuses on. Again, let's use the **SenseCraft-Web Toolkit**.

Follow the following steps to start the SenseCraft-Web-Toolkit:

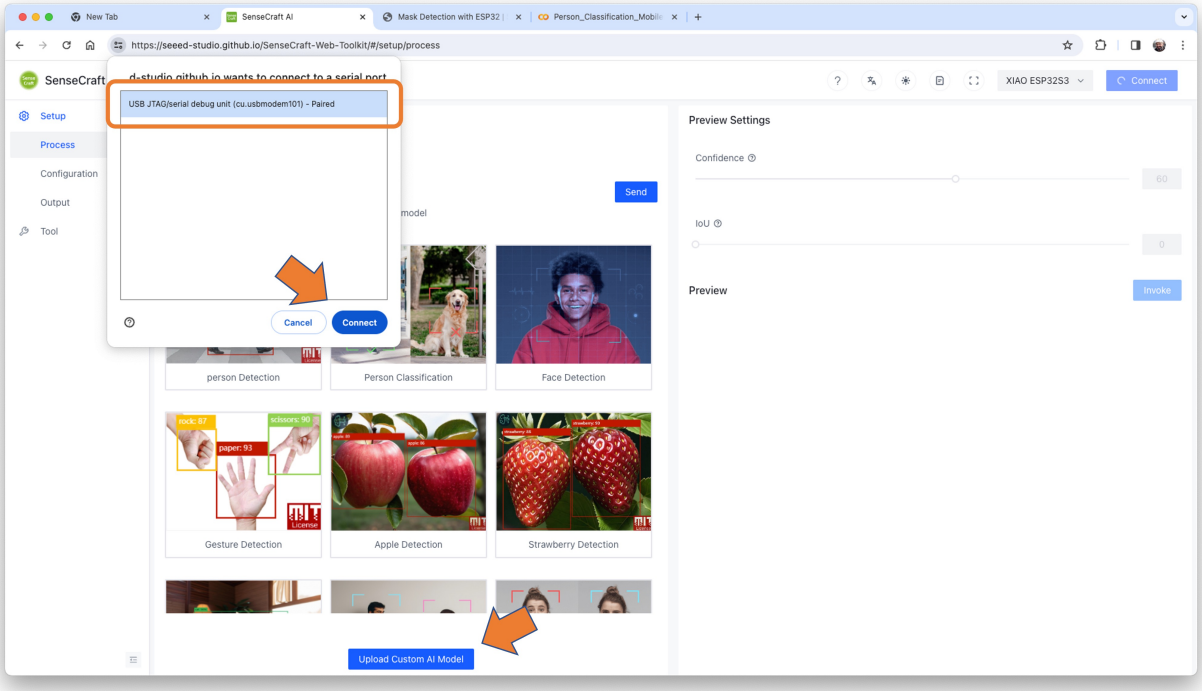
1. Open the SenseCraft-Web-Toolkit website.

2. Connect the XIAO to your computer:

- Having the XIAO connected, select it as below:



- Select the device/Port and press `[Connect]`:



You can try several Computer Vision models previously uploaded by Seeed Studio. Try them and have fun!

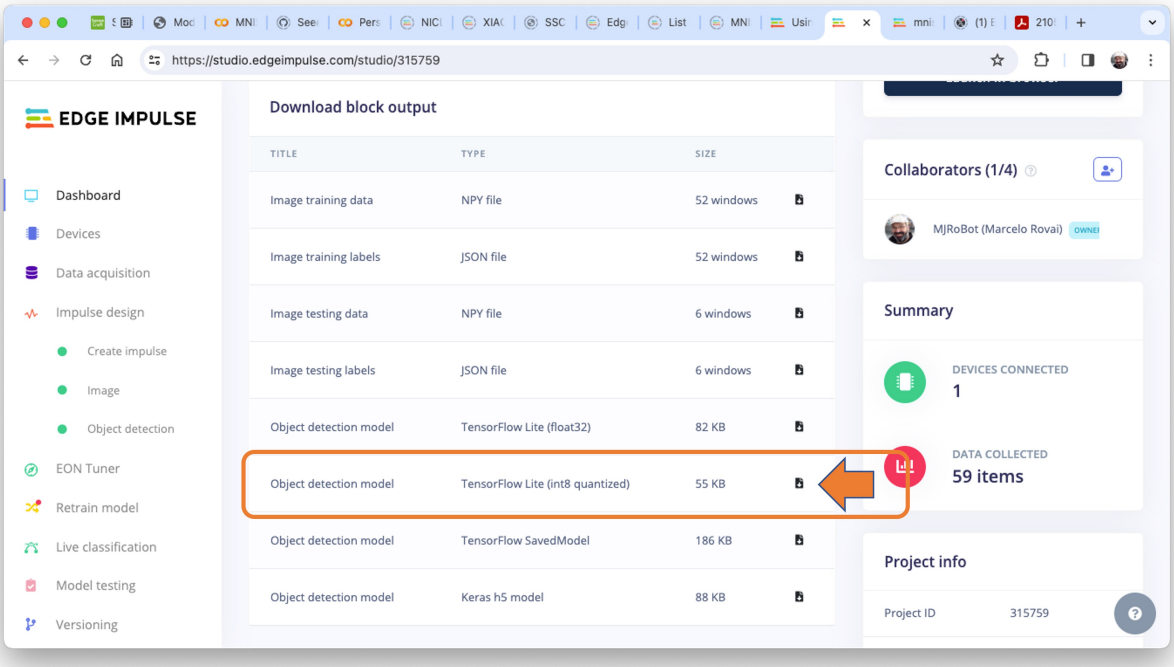
In our case, we will use the blue button at the bottom of the page: `[Upload Custom AI Model]`.

But first, we must download from Edge Impulse Studio our **quantized .tflite** model.

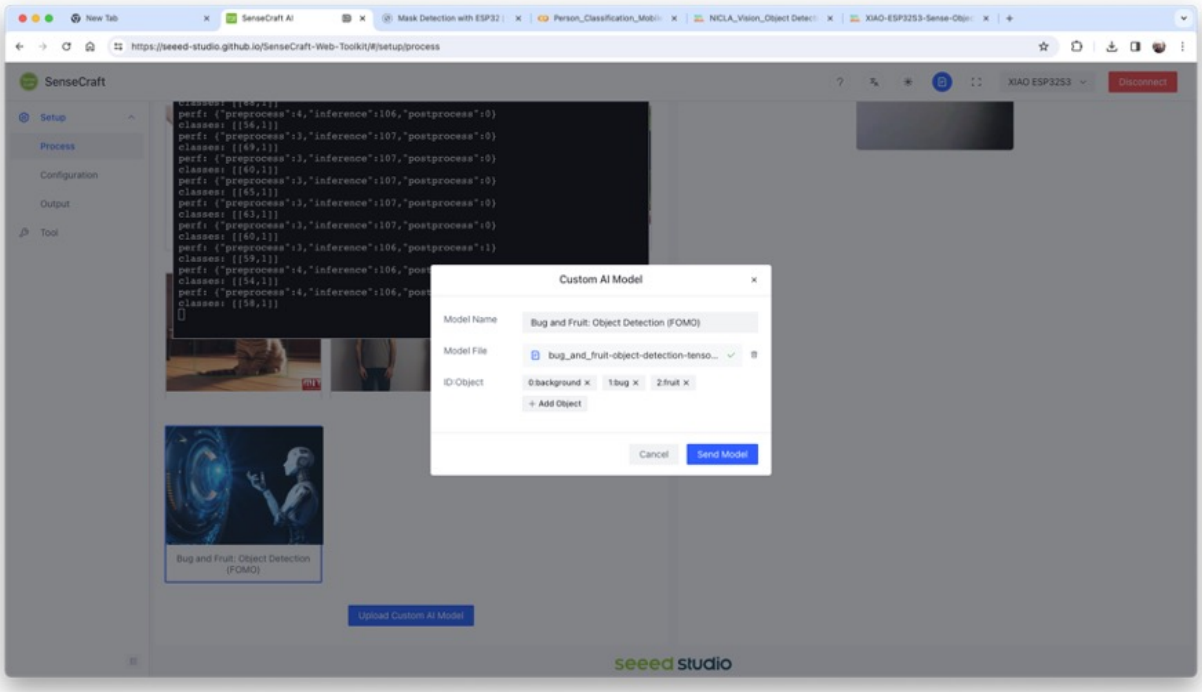
3. Go to your project at Edge Impulse Studio, or clone this one:

- XIAO-ESP32S3-CAM-Fruits-vs-Veggies-v1-ESP-NN

4. On `Dashboard`, download the model ("block output"): `Object Detection model - TensorFlow Lite (int8 quantized)`

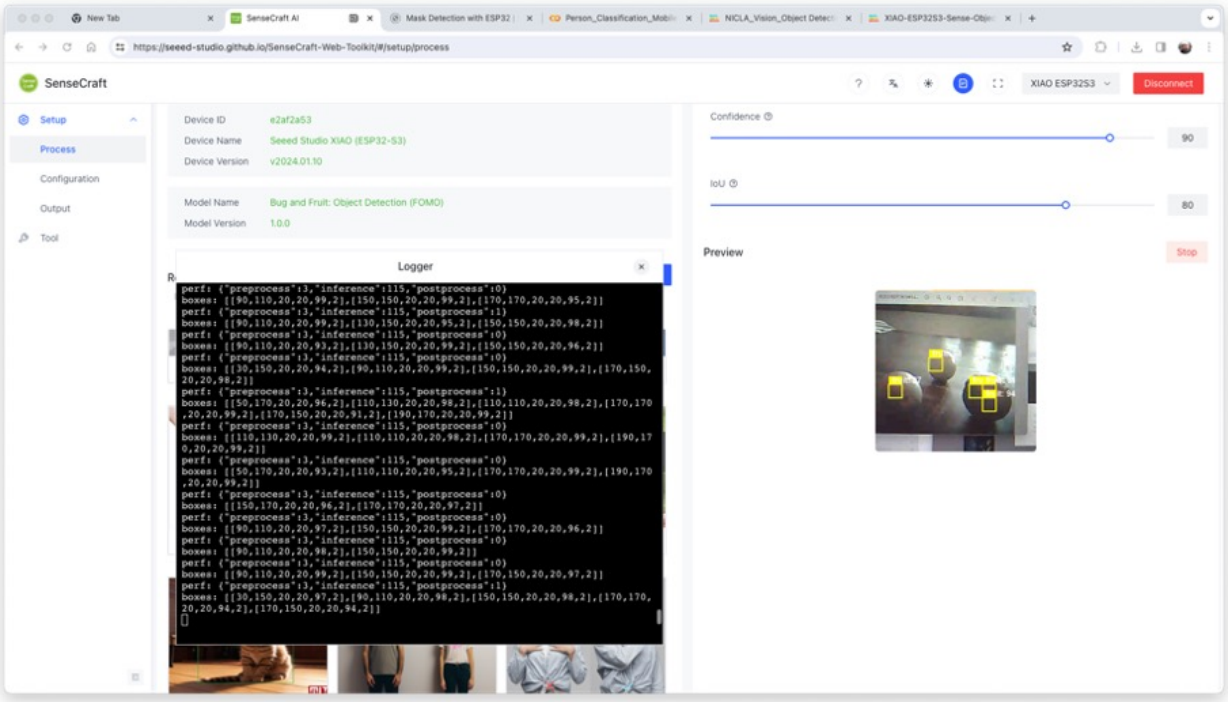


5. On SenseCraft-Web-Toolkit, use the blue button at the bottom of the page: `[Upload Custom AI Model]`. A window will pop up. Enter the Model file that you downloaded to your computer from Edge Impulse Studio, choose a Model Name, and enter with labels (ID: Object):



Note that you should use the labels trained on EI Studio, entering them in alphabetic order (in our case: background, bug, fruit).

After a few seconds (or minutes), the model will be uploaded to your device, and the camera image will appear in real-time on the Preview Sector:



The detected objects will be marked (the centroid). You can select the Confidence of your inference cursor `Confidence`. and `IoU`, which is used to assess the accuracy of predicted bounding boxes compared to truth bounding boxes

Clicking on the top button (Device Log), you can open a Serial Monitor to follow the inference, as we did with the Arduino IDE.

```
perf: {"preprocess":3,"inference":115,"postprocess":1}
boxes: [[30,150,20,20,97,2],[90,110,20,20,98,2],[150,150,20,20,98,2],[170,170,
20,20,94,2],[170,150,20,20,94,2]]
```

On Device Log, you will get information as:

- Preprocess time (image capture and Crop): 3 ms;
- Inference time (model latency): 115 ms,
- Postprocess time (display of the image and marking objects): 1 ms.
- Output tensor (boxes), for example, one of the boxes: [[30,150, 20, 20,97, 2]]; where 30,150, 20, 20 are the coordinates of the box (around the centroid); 97 is the inference result, and 2 is the class (in this case 2: fruit)

> Note that in the above example, we got 5 boxes because none of the fruits got 3 centroids. One solution will be post-processing, where we can aggregate close centroids in one.

Here are other screen shots:

3.9 Conclusion

FOMO is a significant leap in the image processing space, as Louis Moreau and Mat Kelcey put it during its launch in 2022:

> FOMO is a ground-breaking algorithm that brings real-time object detection, tracking, and counting to microcontrollers for the first time.

Multiple possibilities exist for exploring object detection (and, more precisely, counting them) on embedded devices.

4 Audio Feature Engineering

DALL·E 3 Prompt: 1950s style cartoon scene set in an audio research room. Two scientists one holding a magnifying glass and the other taking notes, examine large charts pinned to wall. These charts depict FFT graphs and time curves related to audio data analysis. The r has a retro ambiance, with wooden tables, vintage lamps, and classic audio analysis tools.

4.1 Introduction

In this hands-on tutorial, the emphasis is on the critical role that feature engineering plays in optimizing the performance of machine learning models applied to audio classification tasks, such as speech recognition. It is essential to be aware that the performance of any machine learning model relies heavily on the quality of features used, and we will deal with “under-the-hood” mechanics of feature extraction, mainly focusing on Mel-frequency Cepstral Coefficients (MFCCs), a cornerstone in the field of audio signal processing.

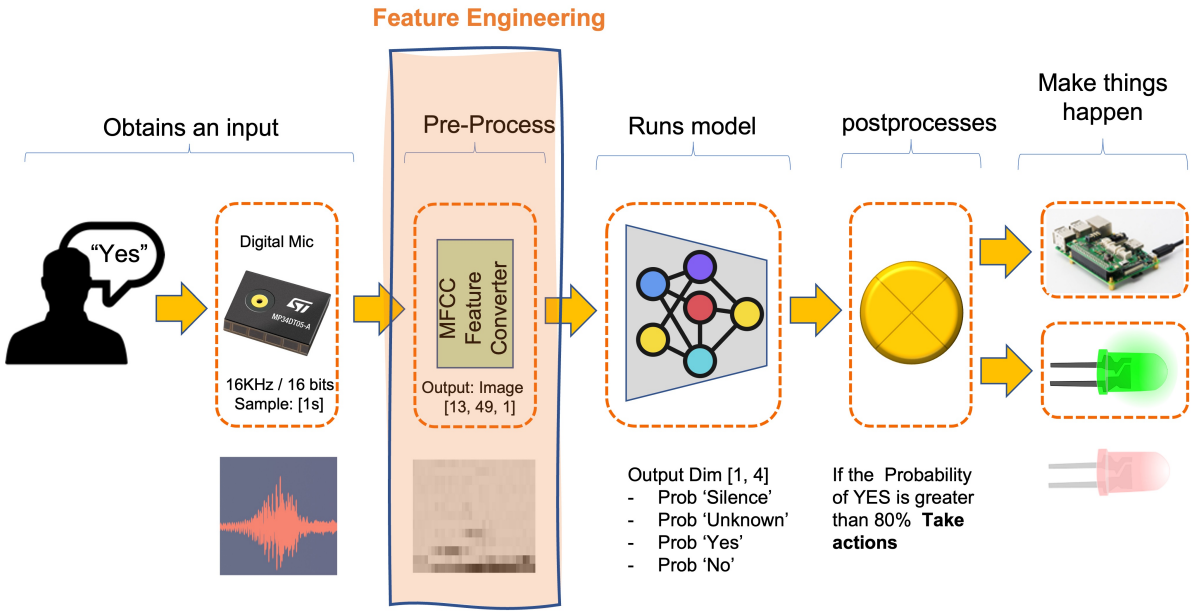
Machine learning models, especially traditional algorithms, don’t understand audio waves. They understand numbers arranged in some meaningful way, i.e., features. These features encapsulate the characteristics of the audio signal, making it easier for models to distinguish between different sounds.

> This tutorial will deal with generating features specifically for audio classification. This can be particularly interesting for applying machine learning to a variety of audio data, whether for speech recognition, music categorization, insect classification based on wingbeat sounds, or other sound analysis tasks

4.2 The KWS

The most common TinyML application is Keyword Spotting (KWS), a subset of the broader field of speech recognition. While general speech recognition aims to transcribe all spoken words into text, Keyword Spotting focuses on detecting specific “keywords” or “wake words” in a continuous audio stream. The system is trained to recognize these keywords as predefined phrases or words, such as *yes* or *no*. In short, KWS is a specialized form of speech recognition with its own set of challenges and requirements.

Here a typical KWS Process using MFCC Feature Converter:



4.2.0.1 Applications of KWS:

- **Voice Assistants:** In devices like Amazon's Alexa or Google Home, KWS is used to detect the wake word ("Alexa" or "Hey Google") to activate the device.
- **Voice-Activated Controls:** In automotive or industrial settings, KWS can be used to initiate specific commands like "Start engine" or "Turn off lights."
- **Security Systems:** Voice-activated security systems may use KWS to authenticate users based on a spoken passphrase.
- **Telecommunication Services:** Customer service lines may use KWS to route calls based on spoken keywords.

4.2.0.2 Differences from General Speech Recognition:

- **Computational Efficiency:** KWS is usually designed to be less computationally intensive than full speech recognition, as it only needs to recognize a small set of phrases.
- **Real-time Processing:** KWS often operates in real-time and is optimized for low-latency detection of keywords.
- **Resource Constraints:** KWS models are often designed to be lightweight, so they can run on devices with limited computational resources, like microcontrollers or mobile phones.

- **Focused Task:** While general speech recognition models are trained to handle a broad range of vocabulary and accents, KWS models are fine-tuned to recognize specific keywords, often in noisy environments accurately.

4.3 Introduction to Audio Signals

Understanding the basic properties of audio signals is crucial for effective feature extraction and, ultimately, for successfully applying machine learning algorithms in audio classification tasks. Audio signals are complex waveforms that capture fluctuations in air pressure over time. These signals can be characterized by several fundamental attributes: sampling rate, frequency, and amplitude.

- **Frequency and Amplitude:** Frequency refers to the number of oscillations a waveform undergoes per unit time and is also measured in Hz. In the context of audio signals, different frequencies correspond to different pitches. Amplitude, on the other hand, measures the magnitude of the oscillations and correlates with the loudness of the sound. Both frequency and amplitude are essential features that capture audio signals' tonal and rhythmic qualities.

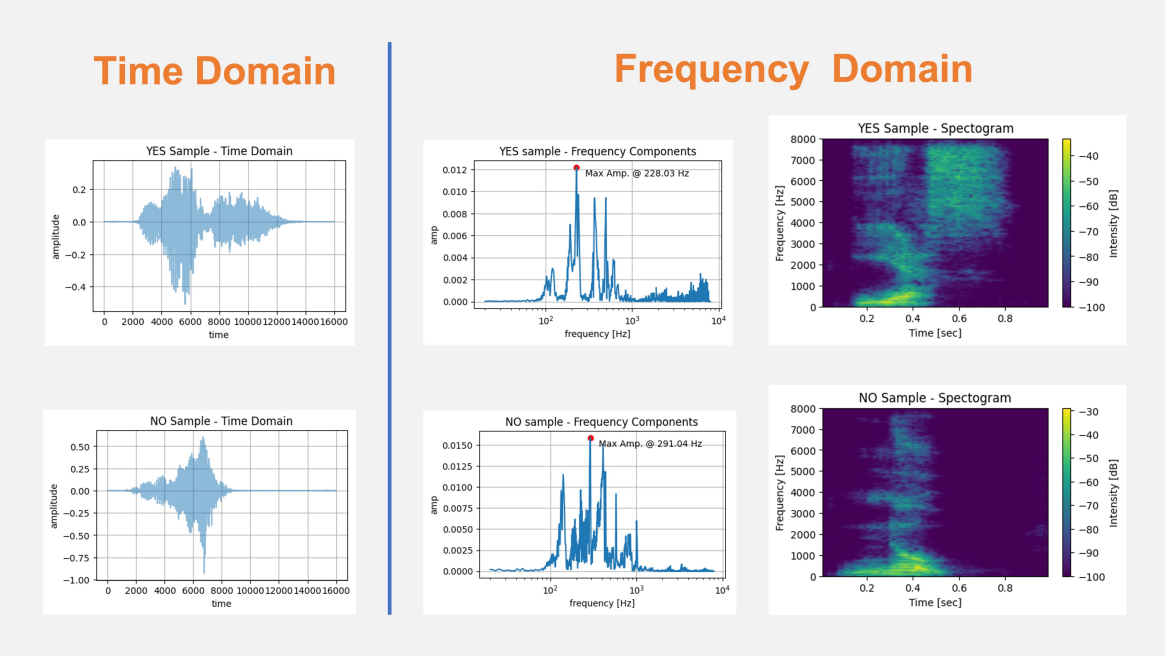
- **Sampling Rate:** The sampling rate, often denoted in Hertz (Hz), defines the number of samples taken per second when digitizing an analog signal. A higher sampling rate allows for a more accurate digital representation of the signal but also demands more computational resources for processing. Typical sampling rates include 44.1 kHz for CD-quality audio and 16 kHz or 8 kHz for speech recognition tasks. Understanding the trade-offs in selecting an appropriate sampling rate is essential for balancing accuracy and computational efficiency. In general, with TinyML projects, we work with 16KHz. Altough music tones can be heard at frequencies up to 20 kHz, voice maxes out at 8 kHz. Traditional telephone systems use an 8 kHz sampling frequency.

For an accurate representation of the signal, the sampling rate must be at least twice the highest frequency present in the signal.

- **Time Domain vs. Frequency Domain:** Audio signals can be analyzed in the time and frequency domains. In the time domain, a signal is

represented as a waveform where the amplitude is plotted against time. This representation helps to observe temporal features like onset and duration but the signal's tonal characteristics are not well evidenced. Conversely, a frequency domain representation provides a view of the signal's constituent frequencies and their respective amplitudes, typically obtained via a Fourier Transform. This is invaluable for tasks that require understanding the signal's spectral content, such as identifying musical notes or speech phonemes (our case).

The image below shows the words YES and NO with typical representations in the Time (Raw Audio) and Frequency domains:



4.3.1 Why Not Raw Audio?

While using raw audio data directly for machine learning tasks may seem tempting, this approach presents several challenges that make it less suitable for building robust and efficient models.

Using raw audio data for Keyword Spotting (KWS), for example, on TinyML devices poses challenges due to its high dimensionality (using a 16 kHz sampling rate), computational complexity for capturing temporal features, susceptibility to noise, and lack of semantically meaningful features, making

feature extraction techniques like MFCCs a more practical choice for resource-constrained applications.

Here are some additional details of the critical issues associated with using raw audio:

- **High Dimensionality:** Audio signals, especially those sampled at high rates, result in large amounts of data. For example, a 1-second audio clip sampled at 16 kHz will have 16,000 individual data points. High-dimensional data increases computational complexity, leading to longer training times and higher computational costs, making it impractical for resource-constrained environments. Furthermore, the wide dynamic range of audio signals requires a significant amount of bits per sample, while conveying little useful information.

- **Temporal Dependencies:** Raw audio signals have temporal structures that simple machine learning models may find hard to capture. While recurrent neural networks like LSTMs can model such dependencies, they are computationally intensive and tricky to train on tiny devices.

- **Noise and Variability:** Raw audio signals often contain background noise and other non-essential elements affecting model performance. Additionally, the same sound can have different characteristics based on various factors such as distance from the microphone, the orientation of the sound source, and acoustic properties of the environment, adding to the complexity of the data.

- **Lack of Semantic Meaning:** Raw audio doesn't inherently contain semantically meaningful features for classification tasks. Features like pitch, tempo, and spectral characteristics, which can be crucial for speech recognition, are not directly accessible from raw waveform data.

- **Signal Redundancy:** Audio signals often contain redundant information, with certain portions of the signal contributing little to no value to the task at hand. This redundancy can make learning inefficient and potentially lead to overfitting.

For these reasons, feature extraction techniques such as Mel-frequency Cepstral Coefficients (MFCCs), Mel-Frequency Energies (MFEs), and simple Spectograms are commonly used to transform raw audio data into a more manageable and informative format. These features capture the essential

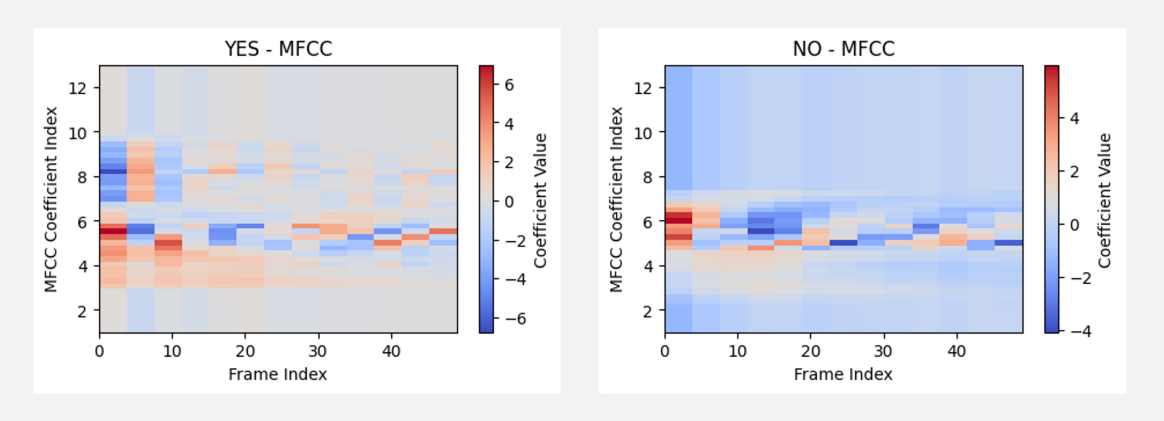
characteristics of the audio signal while reducing dimensionality and noise, facilitating more effective machine learning.

4.4 Introduction to MFCCs

4.4.1 What are MFCCs?

Mel-frequency Cepstral Coefficients (MFCCs) are a set of features derived from the spectral content of an audio signal. They are based on human auditory perceptions and are commonly used to capture the phonetic characteristics of an audio signal. The MFCCs are computed through a multi-step process that includes pre-emphasis, framing, windowing, applying the Fast Fourier Transform (FFT) to convert the signal to the frequency domain, and finally, applying the Discrete Cosine Transform (DCT). The result is a compact representation of the original audio signal's spectral characteristics.

The image below shows the words `YES` and `NO` in their MFCC representation:



This video explains the Mel Frequency Cepstral Coefficients (MFCC) and how to compute them.

4.4.2 Why are MFCCs important?

MFCCs are crucial for several reasons, particularly in the context of Keyword Spotting (KWS) and TinyML:

- **Dimensionality Reduction:** MFCCs capture essential spectral characteristics of the audio signal while significantly reducing the dimensionality of the data, making it ideal for resource-constrained TinyML applications.
- **Robustness:** MFCCs are less susceptible to noise and variations in pitch and amplitude, providing a more stable and robust feature set for audio classification tasks.
- **Human Auditory System Modeling:** The Mel scale in MFCCs approximates the human ear's response to different frequencies, making them practical for speech recognition where human-like perception is desired.
- **Computational Efficiency:** The process of calculating MFCCs is computationally efficient, making it well-suited for real-time applications on hardware with limited computational resources.

In summary, MFCCs offer a balance of information richness and computational efficiency, making them popular for audio classification tasks, particularly in constrained environments like TinyML.

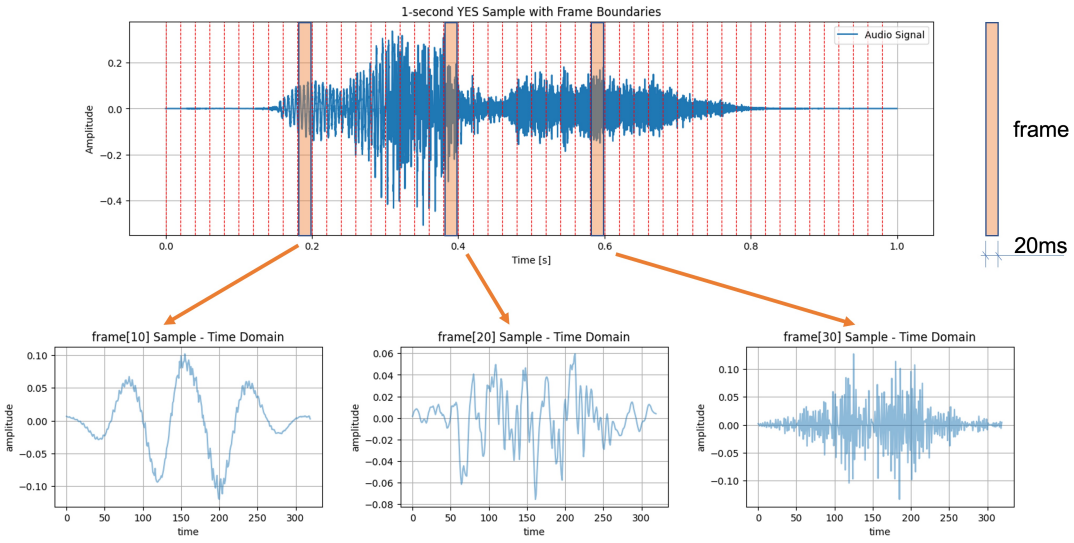
4.4.3 Computing MFCCs

The computation of Mel-frequency Cepstral Coefficients (MFCCs) involves several key steps. Let's walk through these, which are particularly important for Keyword Spotting (KWS) tasks on TinyML devices.

- **Pre-emphasis:** The first step is pre-emphasis, which is applied to accentuate the high-frequency components of the audio signal and balance the frequency spectrum. This is achieved by applying a filter that amplifies the difference between consecutive samples. The formula for pre-emphasis is: $y(t) = x(t) - \alpha\, x(t-1)$, where α is the pre-emphasis factor, typically around 0.97.

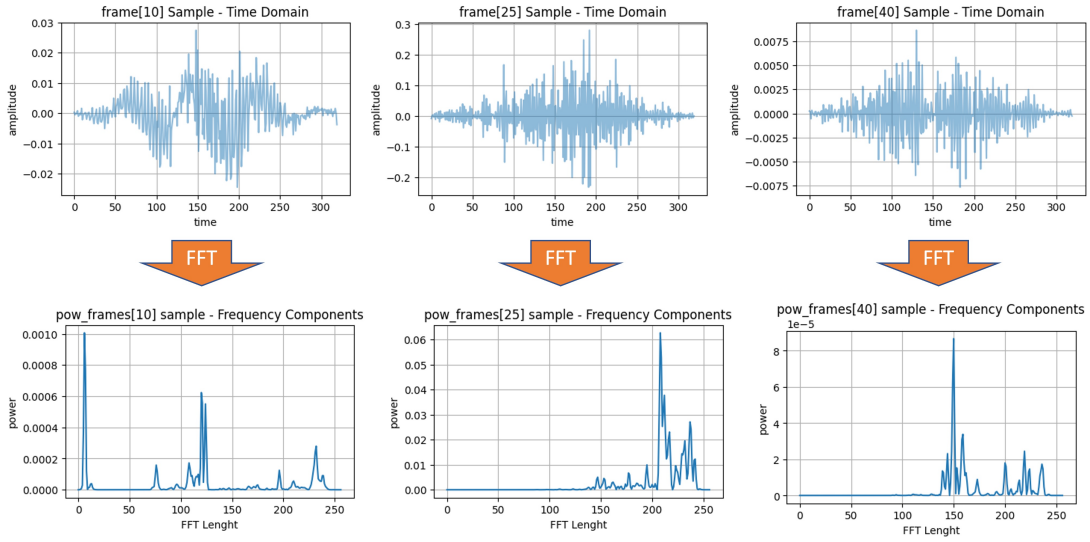
- **Framing:** Audio signals are divided into short frames (the *frame length*), usually 20 to 40 milliseconds. This is based on the assumption that frequencies in a signal are stationary over a short period. Framing helps in analyzing the signal in such small time slots. The *frame stride* (or step) will displace one frame and the adjacent. Those steps could be sequential or overlapped.

- **Windowing:** Each frame is then windowed to minimize the discontinuities at the frame boundaries. A commonly used window function is the Hamming window. Windowing prepares the signal for a Fourier transform by minimizing the edge effects. The image below shows three frames (10, 20, and 30) and the time samples after windowing (note that the frame length and frame stride are 20 ms):

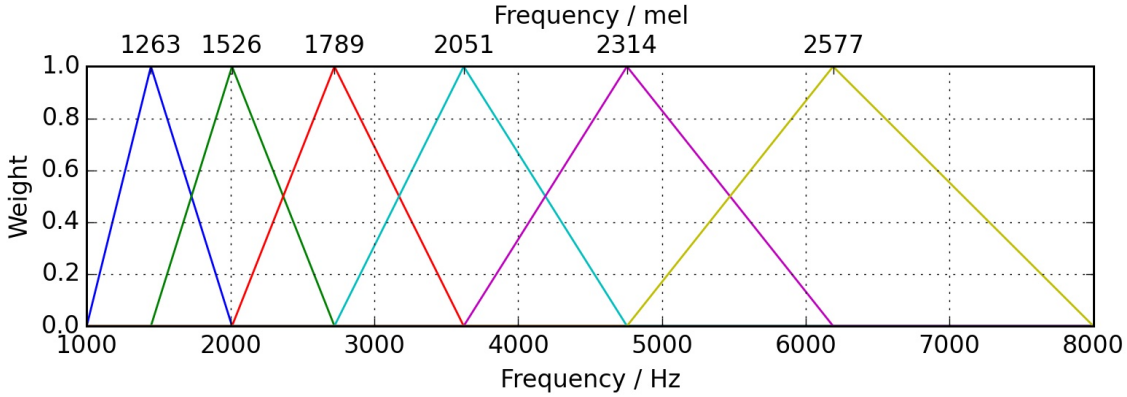


- **Fast Fourier Transform (FFT)** The Fast Fourier Transform (FFT) is applied to each windowed frame to convert it from the time domain to the frequency domain. The FFT gives us a complex-valued representation that includes both magnitude and phase information. However, for MFCCs, only the magnitude is used to calculate the Power Spectrum. The power spectrum is the square of the magnitude spectrum and measures the energy present at each frequency component.

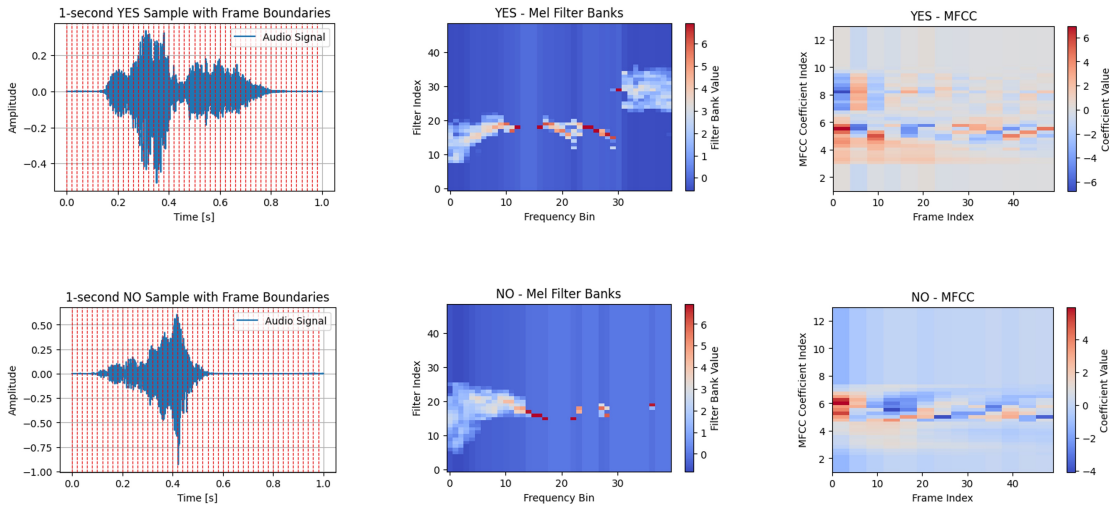
The power spectrum $P(f)$ of a signal $x(t)$ is defined as $P(f) = |X(f)|^2$, where $X(f)$ is the Fourier Transform of $x(t)$. By squaring the magnitude of the Fourier Transform, we emphasize *stronger* frequencies over *weaker* ones, thereby capturing more relevant spectral characteristics of the audio signal. This is important in applications like audio classification, speech recognition, and Keyword Spotting (KWS), where the focus is on identifying distinct frequency patterns that characterize different classes of audio or phonemes in speech.



- **Mel Filter Banks:** The frequency domain is then mapped to the Mel scale, which approximates the human ear's response to different frequencies. The idea is to extract more features (more filter banks) in the lower frequencies and less in the high frequencies. Thus, it performs well on sounds distinguished by the human ear. Typically, 20 to 40 triangular filters extract the Mel-frequency energies. These energies are then log-transformed to convert multiplicative factors into additive ones, making them more suitable for further processing.



- **Discrete Cosine Transform (DCT):** The last step is to apply the Discrete Cosine Transform (DCT) to the log Mel energies. The DCT helps to decorrelate the energies, effectively compressing the data and retaining only the most discriminative features. Usually, the first 12-13 DCT coefficients are retained, forming the final MFCC feature vector.



4.5 Hands-On using Python

Let's apply what we discussed while working on an actual audio sample. Open the notebook on Google CoLab and extract the MLCC features on your audio samples: [Open In Colab]

4.6 Conclusion

4.6.1 What Feature Extraction technique should we use?

Mel-frequency Cepstral Coefficients (MFCCs), Mel-Frequency Energies (MFEs), or Spectrogram are techniques for representing audio data, which are often helpful in different contexts.

In general, MFCCs are more focused on capturing the envelope of the power spectrum, which makes them less sensitive to fine-grained spectral details but more robust to noise. This is often desirable for speech-related tasks. On the other hand, spectrograms or MFEs preserve more detailed frequency information, which can be advantageous in tasks that require discrimination based on fine-grained spectral content.

4.6.1.1 MFCCs are particularly strong for:

1. **Speech Recognition:** MFCCs are excellent for identifying phonetic content in speech signals.
2. **Speaker Identification:** They can be used to distinguish between different speakers based on voice characteristics.
3. **Emotion Recognition:** MFCCs can capture the nuanced variations in speech indicative of emotional states.
4. **Keyword Spotting:** Especially in TinyML, where low computational complexity and small feature size are crucial.

4.6.1.2 Spectrograms or MFEs are often more suitable for:

1. **Music Analysis:** Spectrograms can capture harmonic and timbral structures in music, which is essential for tasks like genre classification, instrument recognition, or music transcription.
2. **Environmental Sound Classification:** In recognizing non-speech, environmental sounds (e.g., rain, wind, traffic), the full spectrogram can provide more discriminative features.
3. **Birdsong Identification:** The intricate details of bird calls are often better captured using spectrograms.
4. **Bioacoustic Signal Processing:** In applications like dolphin or bat call analysis, the fine-grained frequency information in a spectrogram can be essential.
5. **Audio Quality Assurance:** Spectrograms are often used in professional audio analysis to identify unwanted noises, clicks, or other artifacts.

5 Keyword Spotting (KWS)

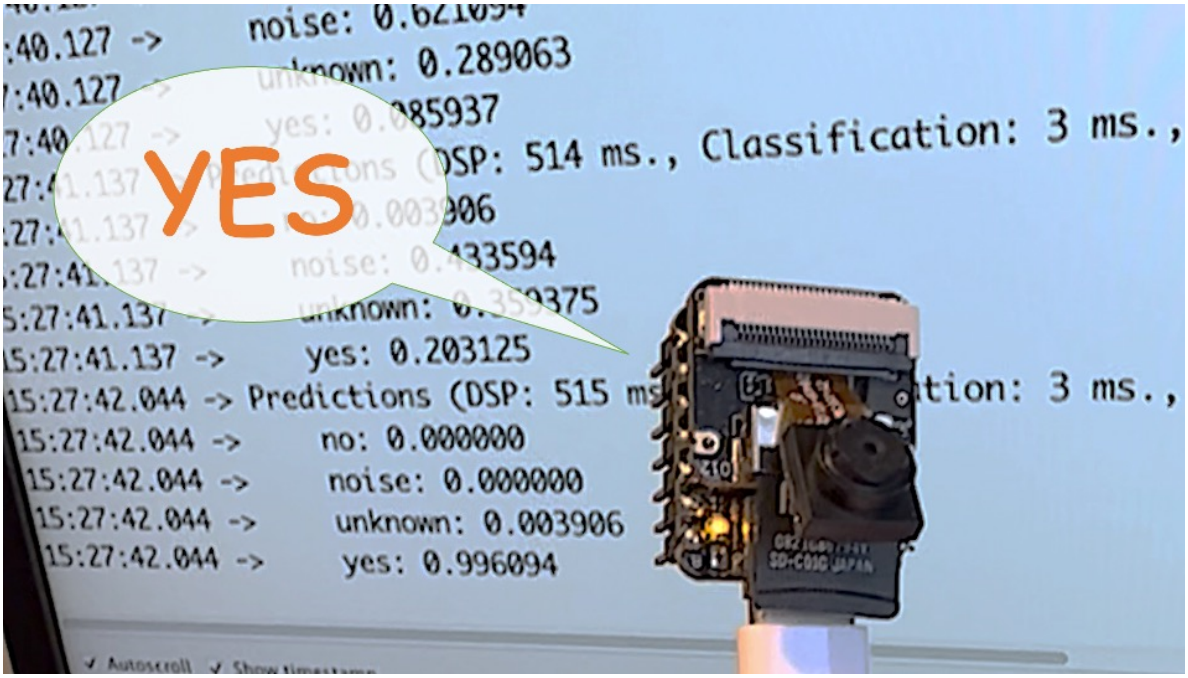


Image by Marcelo Rovai

5.1 Introduction

Keyword Spotting (KWS) is integral to many voice recognition systems, enabling devices to respond to specific words or phrases. While this technology underpins popular devices like Google Assistant or Amazon Alexa, it's equally applicable and achievable on smaller, low-power devices. This lab will guide you through implementing a KWS system using TinyML on the XIAO ESP32S3 microcontroller board.

The XIAO ESP32S3, equipped with Espressif's ESP32-S3 chip, is a compact and potent microcontroller offering a dual-core Xtensa LX7 processor, integrated Wi-Fi, and Bluetooth. Its balance of computational power, energy efficiency, and versatile connectivity make it a fantastic platform for TinyML applications. Also, with its expansion board, we will have access to the "sense" part of the device, which has a 1600x1200 OV2640 camera, an SD

card slot, and a **digital microphone**. The integrated microphone and the SD card will be essential in this project.

We will utilize the Edge Impulse Studio, a powerful, user-friendly platform that simplifies creating and deploying machine learning models onto edge devices. We'll train a KWS model step-by-step, optimizing and deploying it onto the XIAO ESP32S3 Sense.

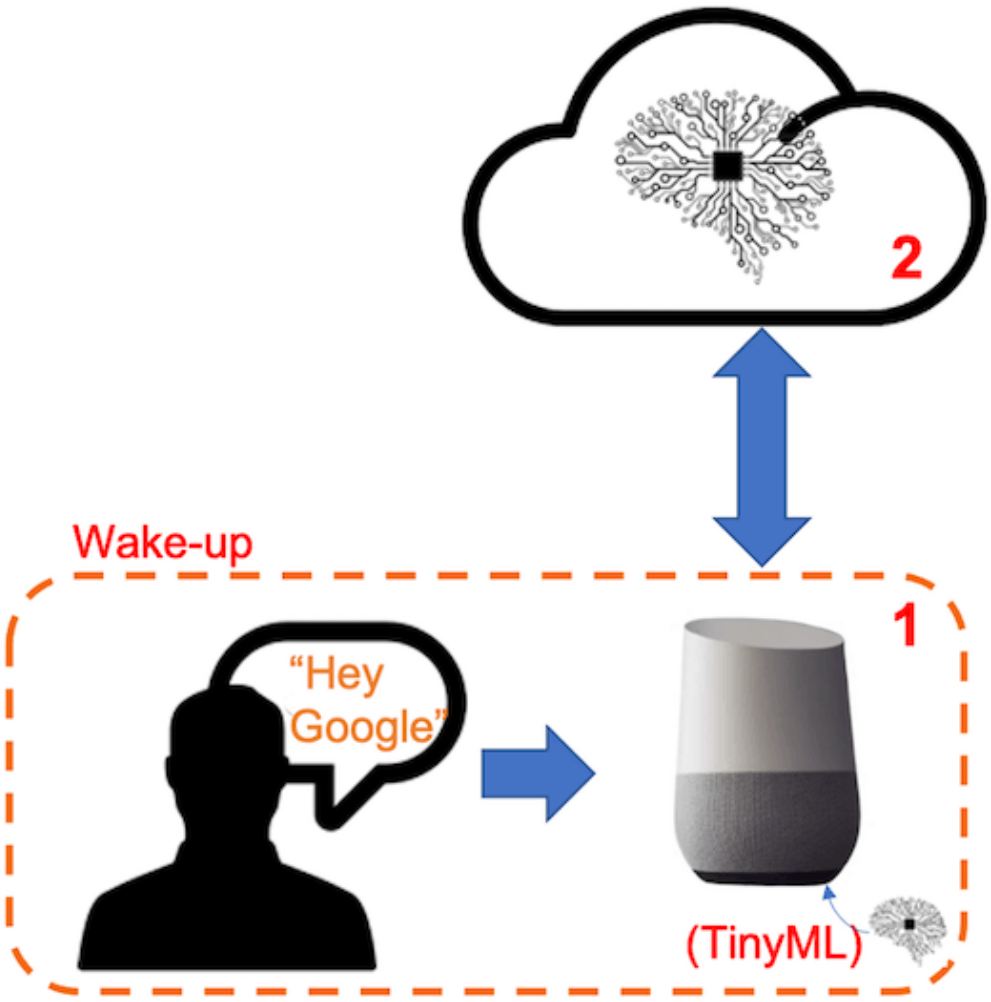
Our model will be designed to recognize keywords that can trigger device wake-up or specific actions (in the case of "YES"), bringing your projects to life with voice-activated commands.

Leveraging our experience with TensorFlow Lite for Microcontrollers (the engine "under the hood" on the EI Studio), we'll create a KWS system capable of real-time machine learning on the device.

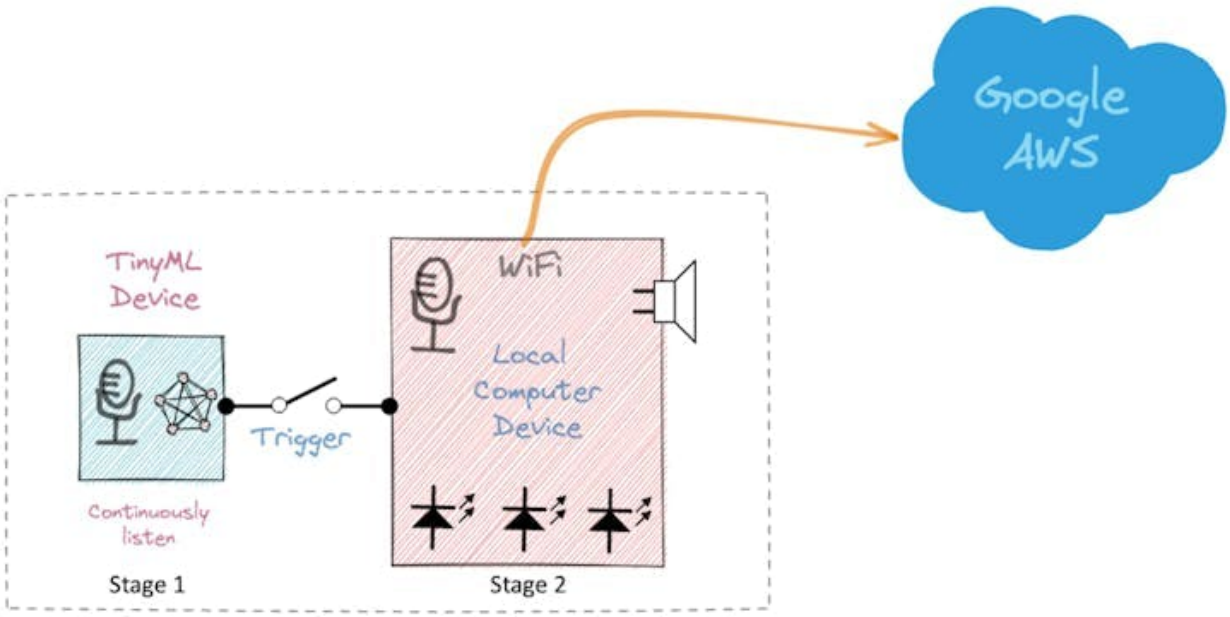
As we progress through the lab, we'll break down each process stage - from data collection and preparation to model training and deployment - to provide a comprehensive understanding of implementing a KWS system on a microcontroller.

5.1.1 How does a voice assistant work?

Keyword Spotting (KWS) is critical to many voice assistants, enabling devices to respond to specific words or phrases. To start, it is essential to realize that Voice Assistants on the market, like Google Home or Amazon Echo-Dot, only react to humans when they are "waked up" by particular keywords such as " Hey Google" on the first one and "Alexa" on the second.



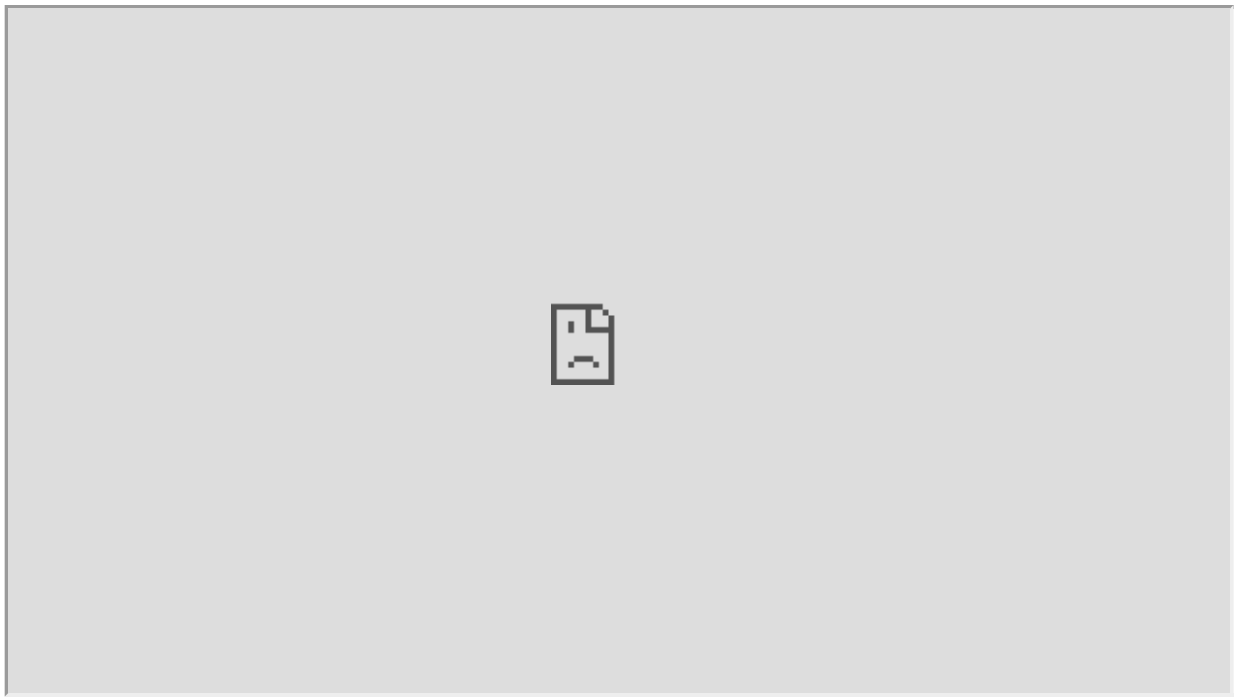
In other words, recognizing voice commands is based on a multi-stage model or Cascade Detection.



Stage 1: A smaller microprocessor inside the Echo Dot or Google Home **continuously** listens to the sound, waiting for the keyword to be spotted. For such detection, a TinyML model at the edge is used (KWS application).

Stage 2: Only when triggered by the KWS application on Stage 1 is the data sent to the cloud and processed on a larger model.

The video below shows an example where I emulate a Google Assistant on a Raspberry Pi (Stage 2), having an Arduino Nano 33 BLE as the tinyML device (Stage 1).

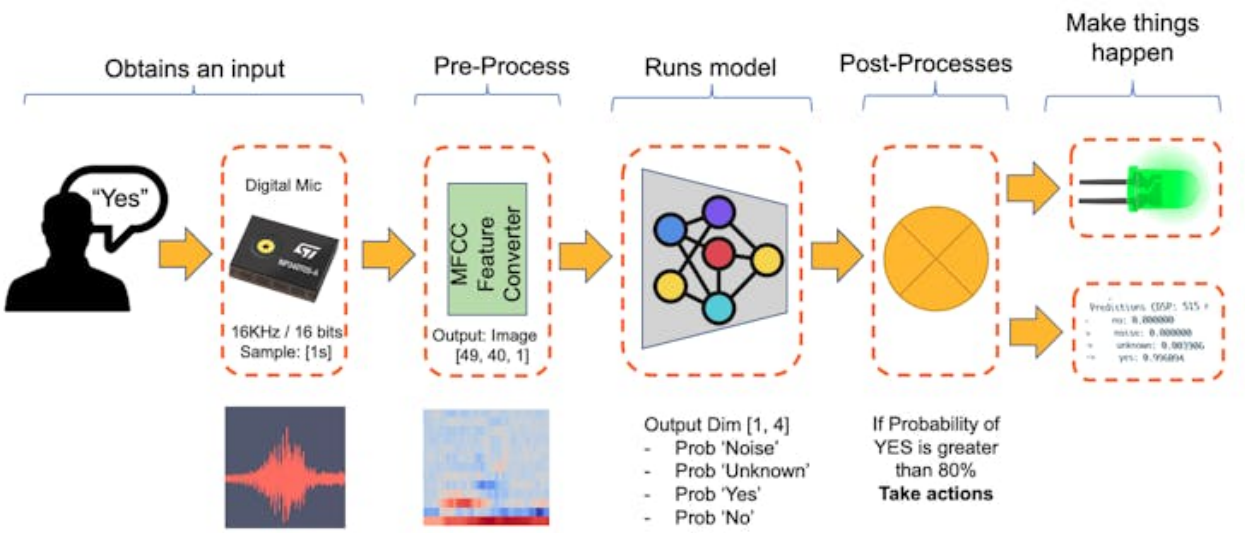


If you want to go deeper on the full project, please see my tutorial: Building an Intelligent Voice Assistant From Scratch.

In this lab, we will focus on Stage 1 (KWS or Keyword Spotting), where we will use the XIAO ESP2S3 Sense, which has a digital microphone for spotting the keyword.

5.1.2 The KWS Project

The below diagram will give an idea of how the final KWS application should work (during inference):



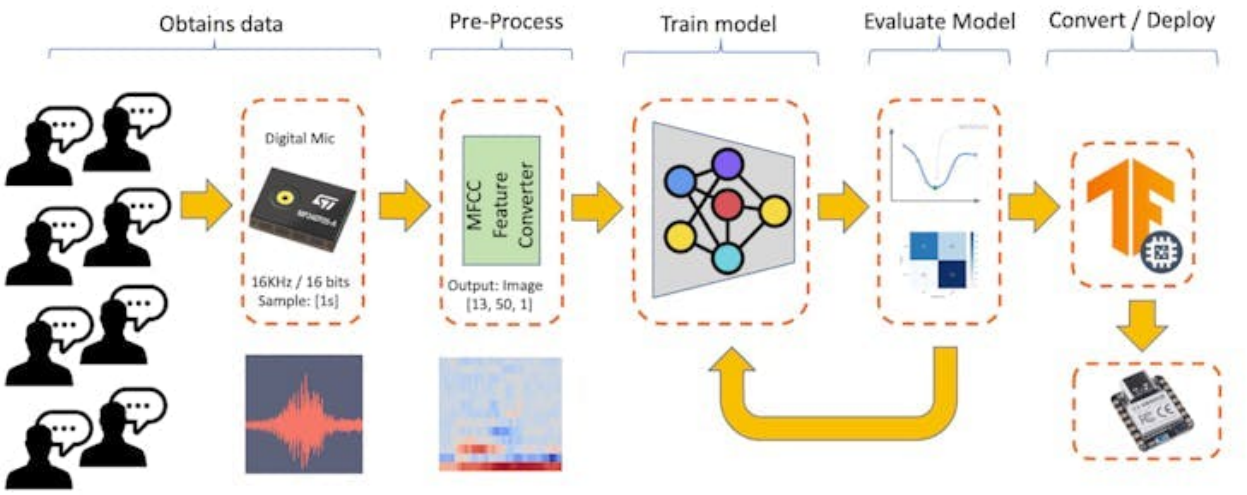
Our KWS application will recognize four classes of sound:

- **YES** (Keyword 1)
- **NO** (Keyword 2)
- **NOISE** (no keywords spoken, only background noise is present)
- **UNKNOW** (a mix of different words than YES and NO)

Optionally for real-world projects, it is always advised to include different words than keywords, such as “Noise” (or Background) and “Unknow.”

5.1.3 The Machine Learning workflow

The main component of the KWS application is its model. So, we must train such a model with our specific keywords, noise, and other words (the “unknown”):



5.2 Dataset

The critical component of Machine Learning Workflow is the **dataset**. Once we have decided on specific keywords (*YES* and NO), we can take advantage of the dataset developed by Pete Warden, "Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition." This dataset has 35 keywords (with +1,000 samples each), such as yes, no, stop, and go. In other words, we can get 1,500 samples of *yes* and *no*.

You can download a small portion of the dataset from Edge Studio (Keyword spotting pre-built dataset), which includes samples from the four classes we will use in this project: yes, no, noise, and background. For this, follow the steps below:

- Download the keywords dataset.
- Unzip the file in a location of your choice.

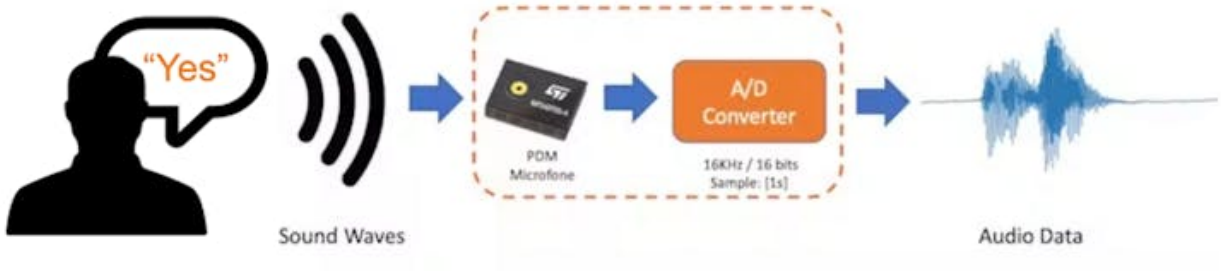
Although we have a lot of data from Pete's dataset, collecting some words spoken by us is advised. When working with accelerometers, creating a dataset with data captured by the same type of sensor was essential. In the case of *sound*, it is different because what we will classify is, in reality, *audio* data.

> The key difference between sound and audio is their form of energy. Sound is mechanical wave energy (longitudinal sound waves) that

propagate through a medium causing variations in pressure within the medium. Audio is made of electrical energy (analog or digital signals) that represent sound electrically.

The sound waves should be converted to audio data when we speak a keyword. The conversion should be done by sampling the signal generated by the microphone in 16KHz with a 16-bit depth.

So, any device that can generate audio data with this basic specification (16Khz/16bits) will work fine. As a device, we can use the proper XIAO ESP32S3 Sense, a computer, or even your mobile phone.



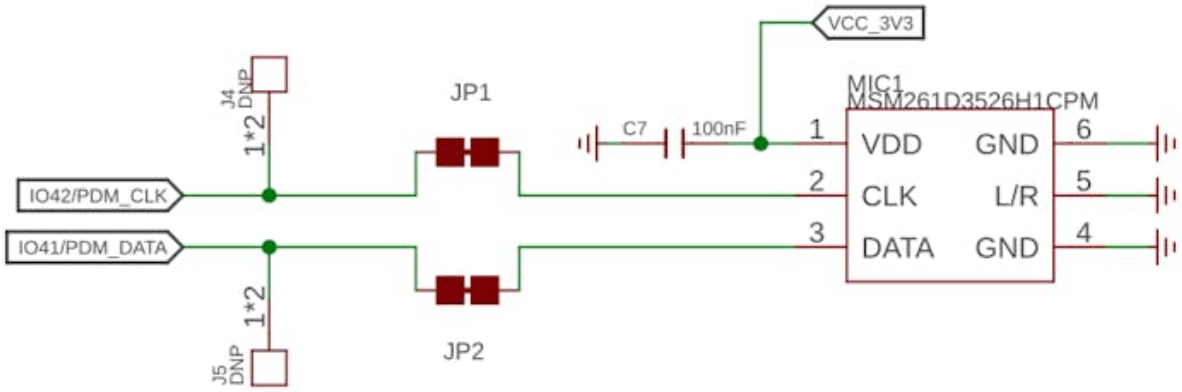
Capturing online Audio Data with Edge Impulse and a smartphone

In the lab Motion Classification and Anomaly Detection, we connect our device directly to Edge Impulse Studio for data capturing (having a sampling frequency of 50Hz to 100Hz). For such low frequency, we could use the EI CLI function *Data Forwarder,* but according to Jan Jongboom, Edge Impulse CTO, *audio (*16KHz) *goes too fast for the data forwarder to be captured.* So, once we have the digital data captured by the microphone, we can turn *it into a WAV file* to be sent to the Studio via Data Uploader (same as we will do with Pete's dataset).

If we want to collect audio data directly on the Studio, we can use any smartphone connected online with it. We will not explore this option here, but you can easily follow EI documentation.

5.2.1 Capturing (offline) Audio Data with the XIAO ESP32S3 Sense

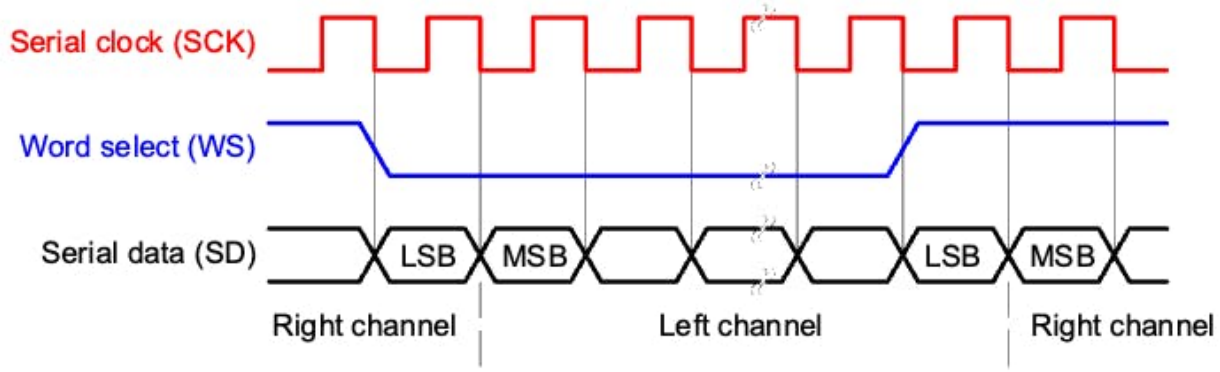
The built-in microphone is the MSM261D3526H1CPM, a PDM digital output MEMS microphone with Multi-modes. Internally, it is connected to the ESP32S3 via an I2S bus using pins IO41 (Clock) and IO41 (Data).



What is I2S?

I2S, or Inter-IC Sound, is a standard protocol for transmitting digital audio from one device to another. It was initially developed by Philips Semiconductor (now NXP Semiconductors). It is commonly used in audio devices such as digital signal processors, digital audio processors, and, more recently, microcontrollers with digital audio capabilities (our case here).

The I2S protocol consists of at least three lines:



1. Bit (or Serial) clock line (BCLK or CLK): This line toggles to indicate the start of a new bit of data (pin IO42).

2. Word select line (WS): This line toggles to indicate the start of a new word (left channel or right channel). The Word select clock (WS) frequency defines the sample rate. In our case, L/R on the microphone is set to ground, meaning that we will use only the left channel (mono).

3. Data line (SD): This line carries the audio data (pin IO41)

In an I2S data stream, the data is sent as a sequence of frames, each containing a left-channel word and a right-channel word. This makes I2S particularly suited for transmitting stereo audio data. However, it can also be used for mono or multichannel audio with additional data lines.

Let's start understanding how to capture raw data using the microphone. Go to the GitHub projectand download the sketch: XIAOEsp2s3_Mic_Test:

```
/*
  XIAO ESP32S3 Simple Mic Test
*/

#include <I2S.h>

void setup() {
  Serial.begin(115200);
  while (!Serial) {
  }

  // start I2S at 16 kHz with 16-bits per sample
  I2S.setAllPins(-1, 42, 41, -1, -1);
  if (!I2S.begin(PDM_MONO_MODE, 16000, 16)) {
    Serial.println("Failed to initialize I2S!");
    while (1); // do nothing
  }
}

void loop() {
  // read a sample
  int sample = I2S.read();

  if (sample && sample != -1 && sample != 1) {
    Serial.println(sample);
  }
}
```

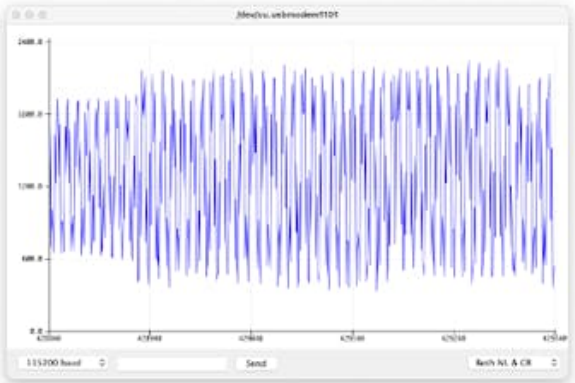
This code is a simple microphone test for the XIAO ESP32S3 using the I2S (Inter-IC Sound) interface. It sets up the I2S interface to capture audio data at a sample rate of 16 kHz with 16 bits per sample and then continuously reads samples from the microphone and prints them to the serial monitor.

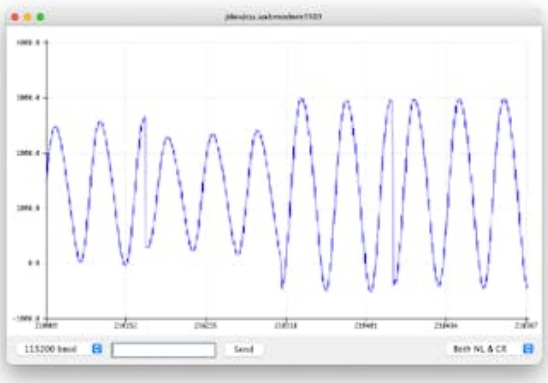
Let's dig into the code's main parts:

- Include the I2S library: This library provides functions to configure and use the I2S interface, which is a standard for connecting digital audio devices.
- I2S.setAllPins(-1, 42, 41, -1, -1): This sets up the I2S pins. The parameters are (-1, 42, 41, -1, -1), where the second parameter (42) is the PIN for the I2S clock (CLK), and the third parameter (41) is the PIN for the I2S data (DATA) line. The other parameters are set to -1, meaning those pins are not used.
- I2S.begin(PDM_MONO_MODE, 16000, 16): This initializes the I2S interface in Pulse Density Modulation (PDM) mono mode, with a sample rate of 16 kHz and 16 bits per sample. If the initialization fails, an error message is printed, and the program halts.
- int sample = I2S.read(): This reads an audio sample from the I2S interface.

If the sample is valid, it is printed on the Serial Monitor and Plotter.

Below is a test "whispering" in two different tones.



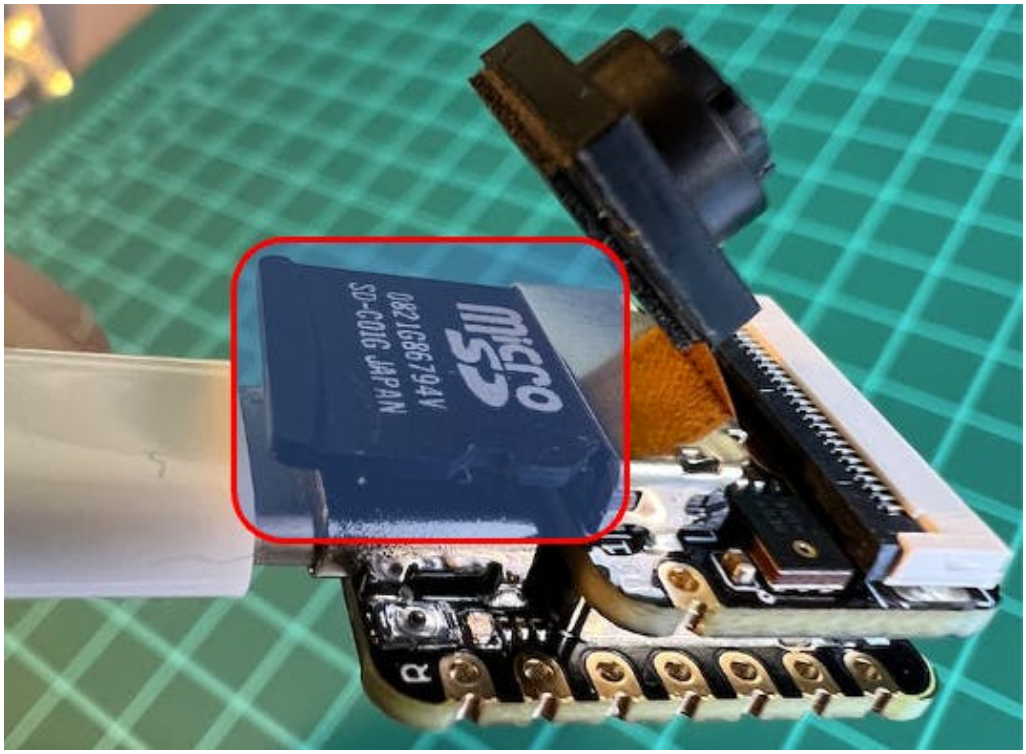


5.2.2 Save recorded sound samples (dataset) as .wav audio files to a microSD card.

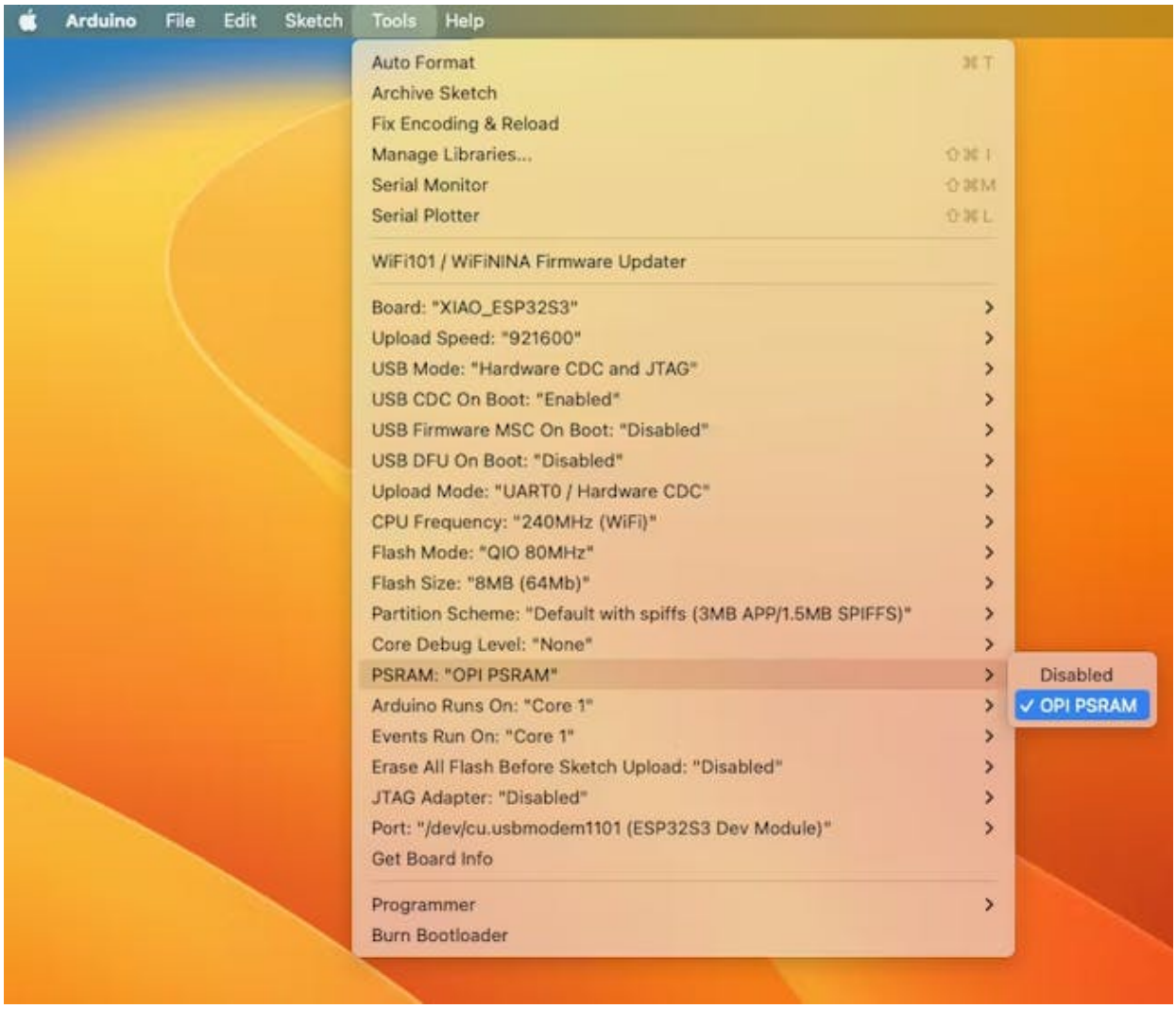
Let's use the onboard SD Card reader to save .wav audio files; we must habilitate the XIAO PSRAM first.

> ESP32-S3 has only a few hundred kilobytes of internal RAM on the MCU chip. It can be insufficient for some purposes so that 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).



Turn the PSRAM function of the ESP-32 chip on (Arduino IDE):
Tools>PSRAM: "OPI PSRAM">OPI PSRAM



- Download the sketch Wav_Record_dataset,which you can find on the project's GitHub.

This code records audio using the I2S interface of the Seeed XIAO ESP32S3 Sense board, saves the recording as a.wav file on an SD card, and allows for control of the recording process through commands sent from the serial monitor. The name of the audio file is customizable (it should be the class labels to be used with the training), and multiple recordings can be made, each saved in a new file. The code also includes functionality to increase the volume of the recordings.

Let's break down the most essential parts of it:

```
#include <I2S.h>
#include "FS.h"
#include "SD.h"
#include "SPI.h"
```

Those are the necessary libraries for the program. I2S.h allows for audio input, FS.h provides file system handling capabilities, SD.h enables the program to interact with an SD card, and SPI.h handles the SPI communication with the SD card.

```
#define RECORD_TIME   10
#define SAMPLE_RATE 16000U
#define SAMPLE_BITS 16
#define WAV_HEADER_SIZE 44
#define VOLUME_GAIN 2
```

Here, various constants are defined for the program.

- **RECORD_TIME** specifies the length of the audio recording in seconds.
- **SAMPLE_RATE** and **SAMPLE_BITS** define the audio quality of the recording.
- **WAV_HEADER_SIZE** specifies the size of the .wav file header.
- **VOLUME_GAIN** is used to increase the volume of the recording.

```
int fileNumber = 1;
String baseFileName;
bool isRecording = false;
```

These variables keep track of the current file number (to create unique file names), the base file name, and whether the system is currently recording.

```
void setup() {
  Serial.begin(115200);
  while (!Serial);

  I2S.setAllPins(-1, 42, 41, -1, -1);
  if (!I2S.begin(PDM_MONO_MODE, SAMPLE_RATE, SAMPLE_BITS)) {
    Serial.println("Failed to initialize I2S!");
    while (1);
  }

  if(!SD.begin(21)){
    Serial.println("Failed to mount SD Card!");
    while (1);
```

```
  }
  Serial.printf("Enter with the label name\n");
}
```

The setup function initializes the serial communication, I2S interface for audio input, and SD card interface. If the I2S did not initialize or the SD card fails to mount, it will print an error message and halt execution.

```
void loop() {
  if (Serial.available() > 0) {
    String command = Serial.readStringUntil('\n');
    command.trim();
    if (command == "rec") {
      isRecording = true;
    } else {
      baseFileName = command;
      fileNumber = 1; //reset file number each time a new basefile
name is set
      Serial.printf("Send rec for starting recording label \n");
    }
  }
  if (isRecording && baseFileName != "") {
    String fileName = "/" + baseFileName + "." + String(fileNumber) +
".wav";
    fileNumber++;
    record_wav(fileName);
    delay(1000); // delay to avoid recording multiple files at once
    isRecording = false;
  }
}
```

In the main loop, the program waits for a command from the serial monitor. If the command is rec, the program starts recording. Otherwise, the command is assumed to be the base name for the .wav files. If it's currently recording and a base file name is set, it records the audio and saves it as a.wav file. The file names are generated by appending the file number to the base file name.

```
void record_wav(String fileName)
{
  ...

  File file = SD.open(fileName.c_str(), FILE_WRITE);
  ...
  rec_buffer = (uint8_t *)ps_malloc(record_size);
```

```
  ...

  esp_i2s::i2s_read(esp_i2s::I2S_NUM_0,
                    rec_buffer,
                    record_size,
                    &sample_size,
                    portMAX_DELAY);
  ...
}
```

This function records audio and saves it as a.wav file with the given name. It starts by initializing the sample_size and record_size variables. record_size is calculated based on the sample rate, size, and desired recording time. Let's dig into the essential sections;

```
File file = SD.open(fileName.c_str(), FILE_WRITE);
// Write the header to the WAV file
uint8_t wav_header[WAV_HEADER_SIZE];
generate_wav_header(wav_header, record_size, SAMPLE_RATE);
file.write(wav_header, WAV_HEADER_SIZE);
```

This section of the code opens the file on the SD card for writing and then generates the .wav file header using the generate_wav_header function. It then writes the header to the file.

```
// PSRAM malloc for recording
rec_buffer = (uint8_t *)ps_malloc(record_size);
if (rec_buffer == NULL) {
  Serial.printf("malloc failed!\n");
  while(1) ;
}
Serial.printf("Buffer: %d bytes\n", ESP.getPsramSize() -
ESP.getFreePsram());
```

The ps_malloc function allocates memory in the PSRAM for the recording. If the allocation fails (i.e., rec_buffer is NULL), it prints an error message and halts execution.

```
// Start recording
esp_i2s::i2s_read(esp_i2s::I2S_NUM_0,
                                    rec_buffer,
                                    record_size,
                                    &sample_size,
                                    portMAX_DELAY);
if (sample_size == 0) {
```

```
  Serial.printf("Record Failed!\n");
} else {
    Serial.printf("Record %d bytes\n", sample_size);
  }
```

The i2s_read function reads audio data from the microphone into rec_buffer. It prints an error message if no data is read (sample_size is 0).

```
// Increase volume
for (uint32_t i = 0; i < sample_size; i += SAMPLE_BITS/8) {
  (*(uint16_t *)(rec_buffer+i)) <<= VOLUME_GAIN;
}
```

This section of the code increases the recording volume by shifting the sample values by VOLUME_GAIN.

```
// Write data to the WAV file
Serial.printf("Writing to the file ...\n");
if (file.write(rec_buffer, record_size) != record_size)
  Serial.printf("Write file Failed!\n");

free(rec_buffer);
file.close();
Serial.printf("Recording complete: \n");
Serial.printf("Send rec for a new sample or enter a new label\n\n");
```

Finally, the audio data is written to the .wav file. If the write operation fails, it prints an error message. After writing, the memory allocated for rec_buffer is freed, and the file is closed. The function finishes by printing a completion message and prompting the user to send a new command.

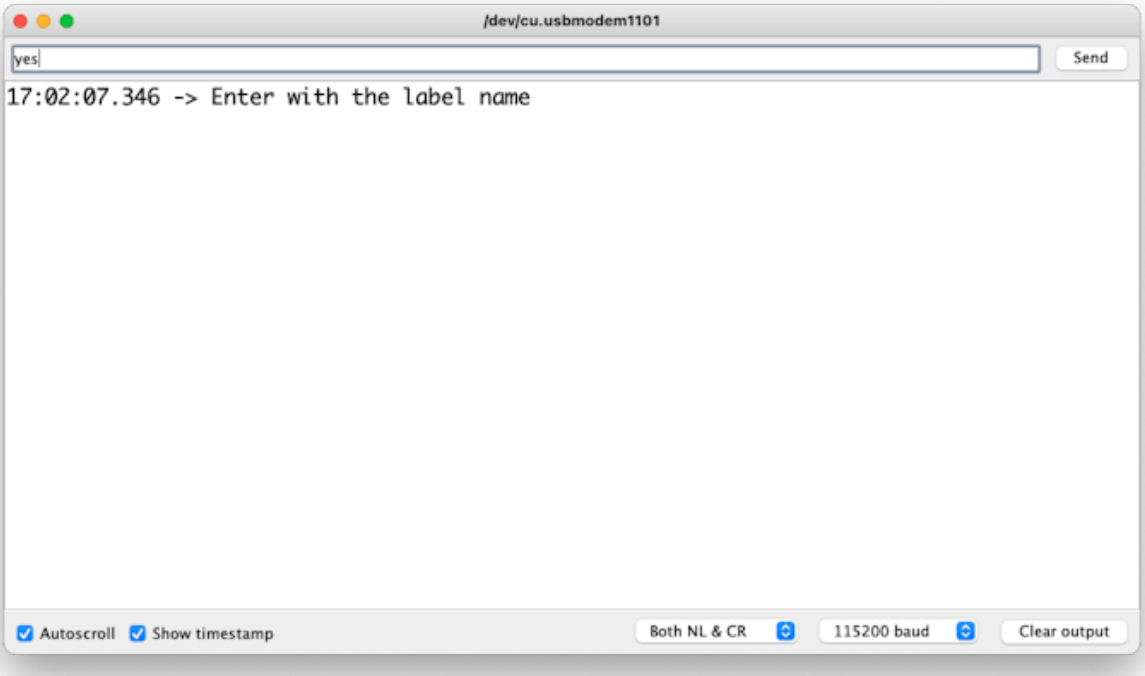
```
void generate_wav_header(uint8_t *wav_header,
                                                uint32_t wav_size,
                                                uint32_t sample_rate)
{
  ...
  memcpy(wav_header, set_wav_header, sizeof(set_wav_header));
}
```

The generate_wav_header function creates a.wav file header based on the parameters (wav_size and sample_rate). It generates an array of bytes according to the .wav file format, which includes fields for the file size, audio format, number of channels, sample rate, byte rate, block alignment, bits per

sample, and data size. The generated header is then copied into the wav_header array passed to the function.

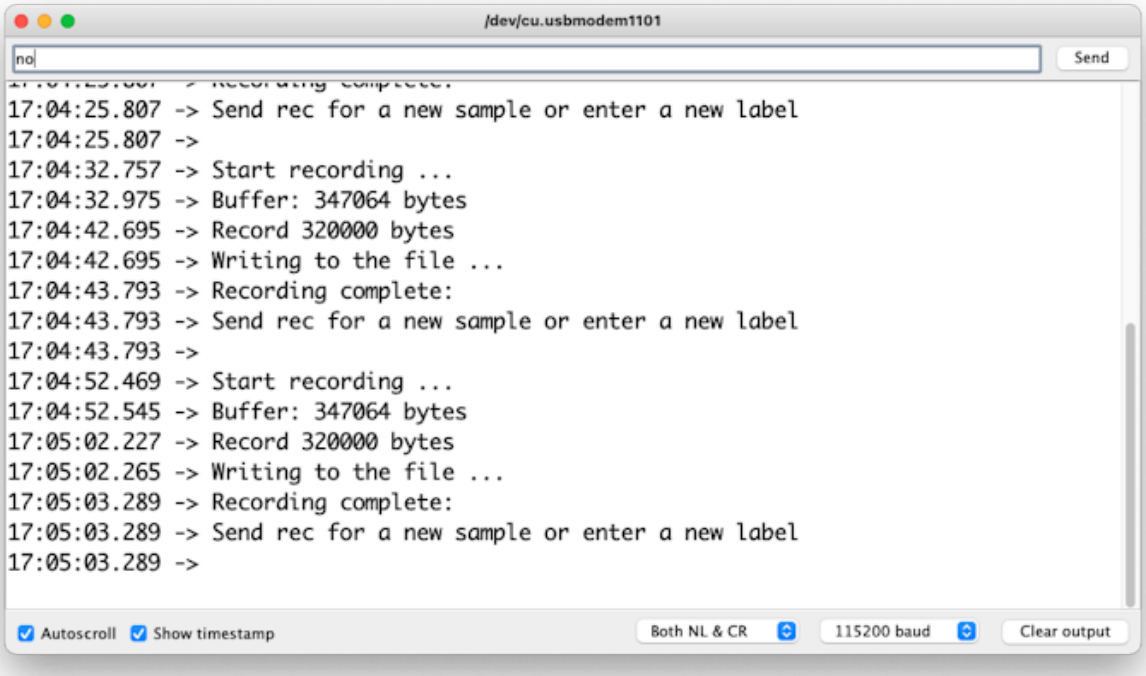
Now, upload the code to the XIAO and get samples from the keywords (yes and no). You can also capture noise and other words.

The Serial monitor will prompt you to receive the label to be recorded.

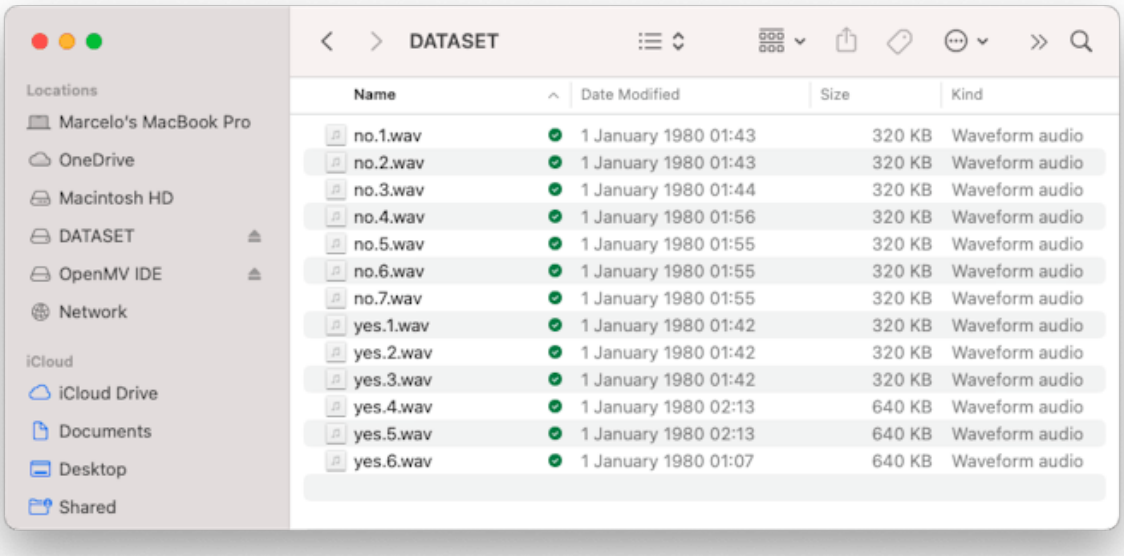


Send the label (for example, yes). The program will wait for another command: rec

And the program will start recording new samples every time a command rec is sent. The files will be saved as yes.1.wav, yes.2.wav, yes.3.wav, etc., until a new label (for example, no) is sent. In this case, you should send the command rec for each new sample, which will be saved as no.1.wav, no.2.wav, no.3.wav, etc.



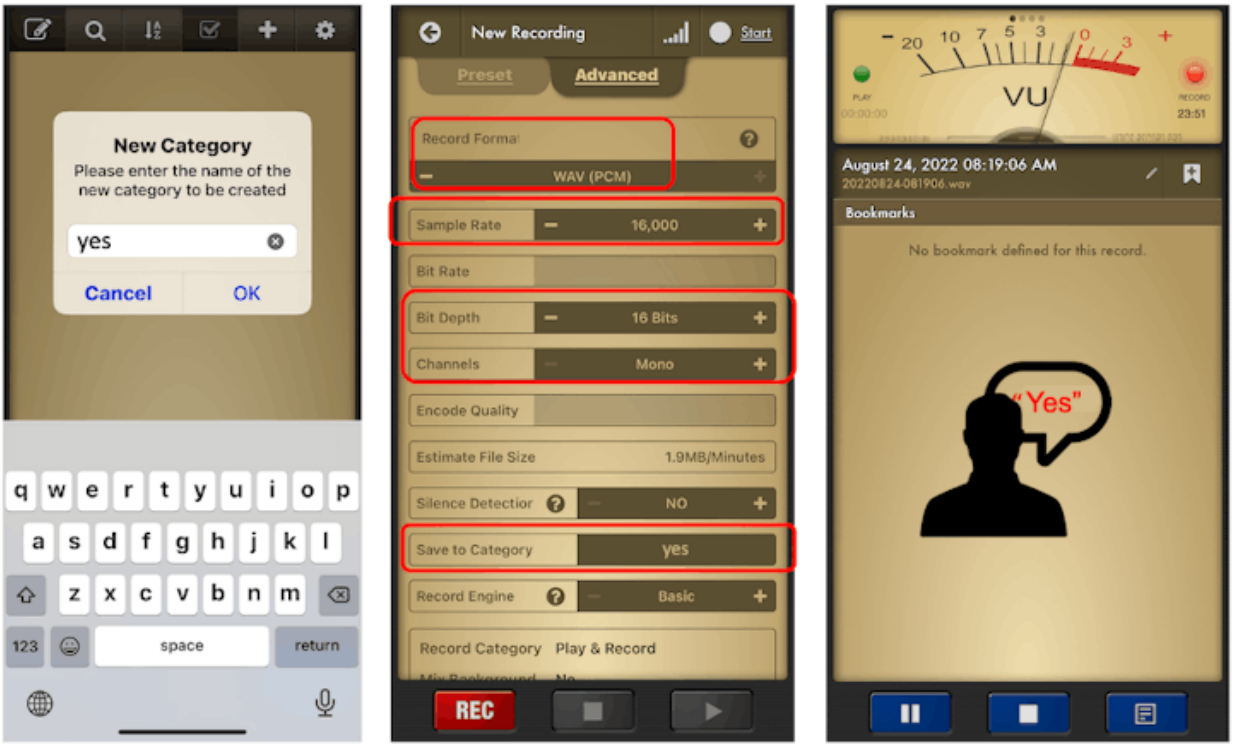
Ultimately, we will get the saved files on the SD card.



The files are ready to be uploaded to Edge Impulse Studio

5.2.3 Capturing (offline) Audio Data Apps

Alternatively, you can also use your PC or smartphone to capture audio data with a sampling frequency 16KHz and a bit depth of 16 Bits. A good app for that is *Voice Recorder Pro* (IOS). You should save your records as .wav files and send them to your computer.

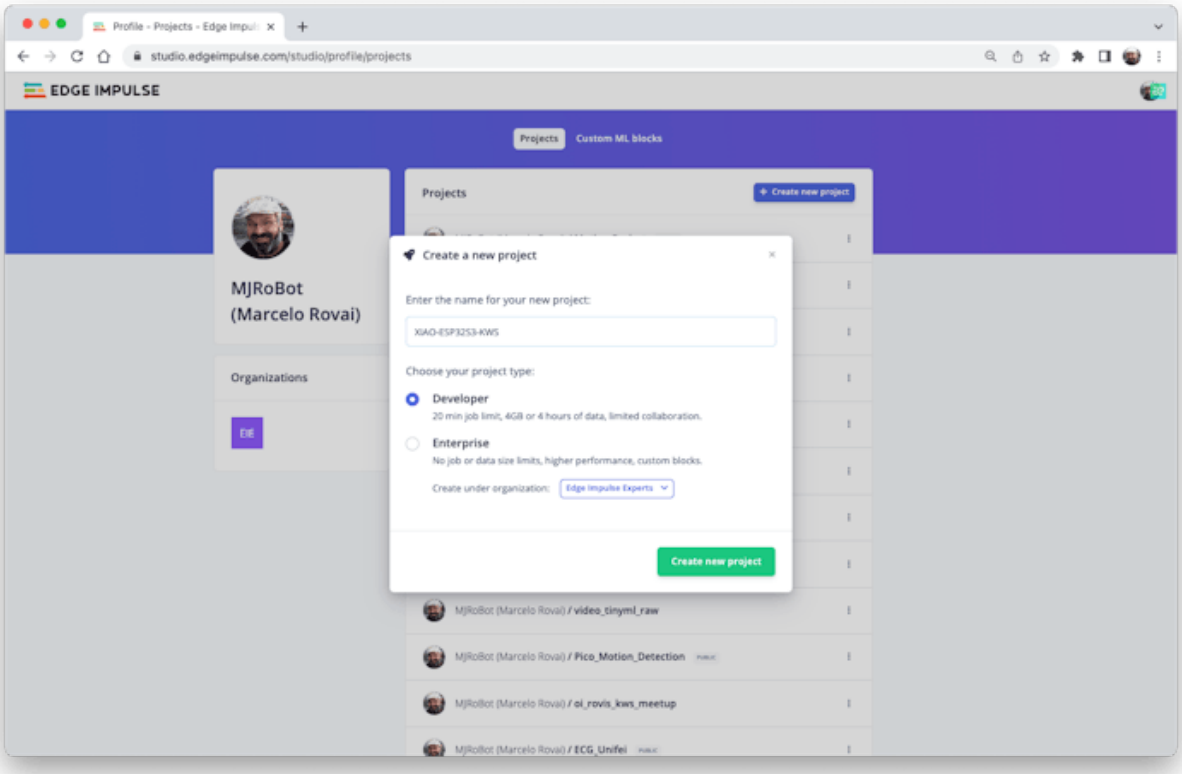


Note that any app, such as Audacity, can be used for audio recording or even your computer.

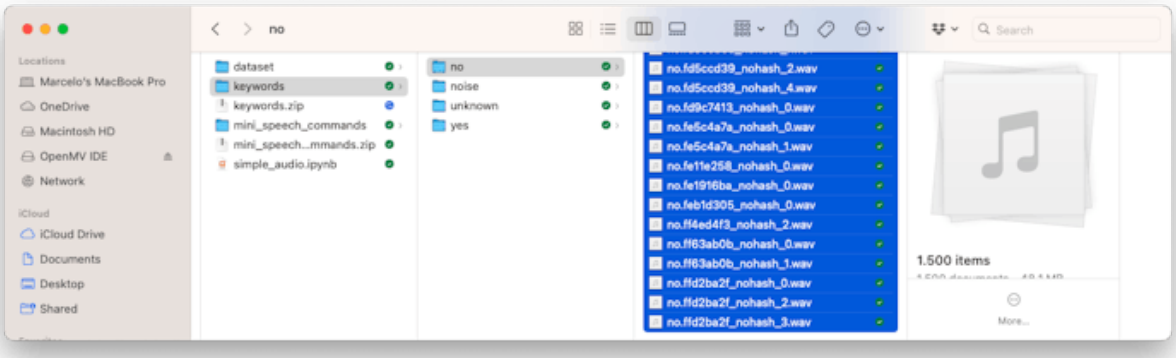
5.3 Training model with Edge Impulse Studio

5.3.1 Uploading the Data

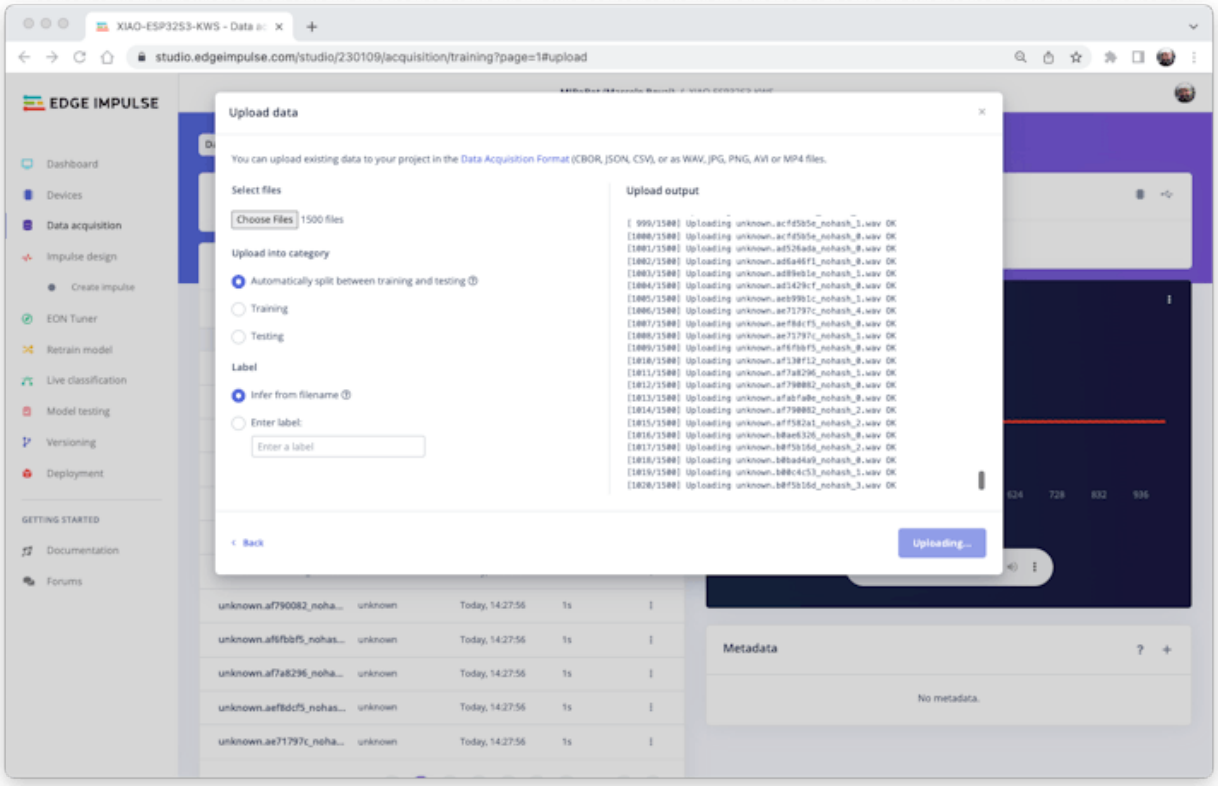
When the raw dataset is defined and collected (Pete's dataset + recorded keywords), we should initiate a new project at Edge Impulse Studio:



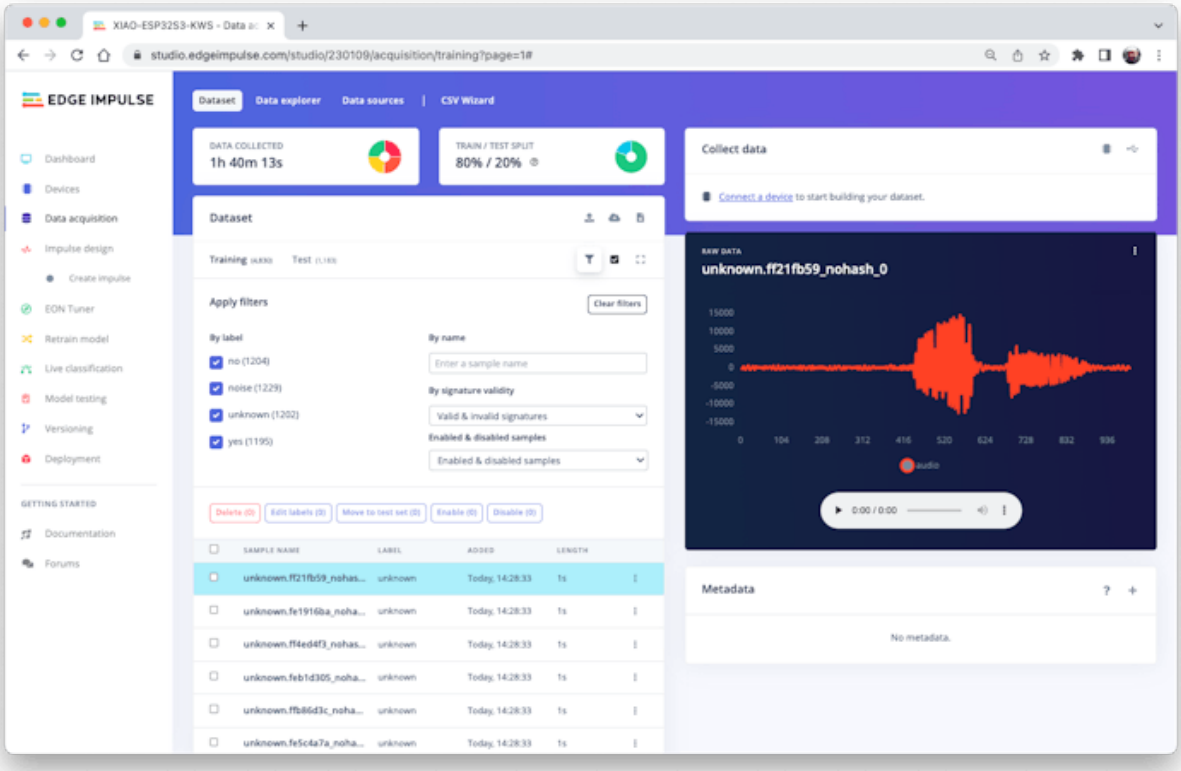
Once the project is created, select the Upload Existing Data tool in the Data Acquisition section. Choose the files to be uploaded:



And upload them to the Studio (You can automatically split data in train/test). Repete to all classes and all raw data.

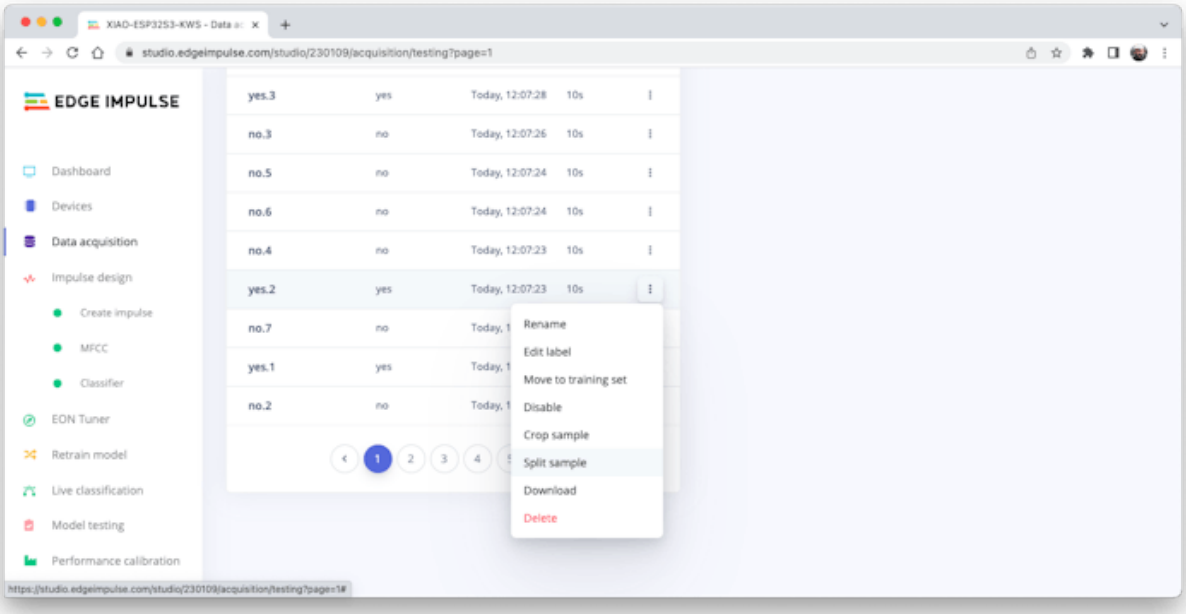


The samples will now appear in the Data acquisition section.

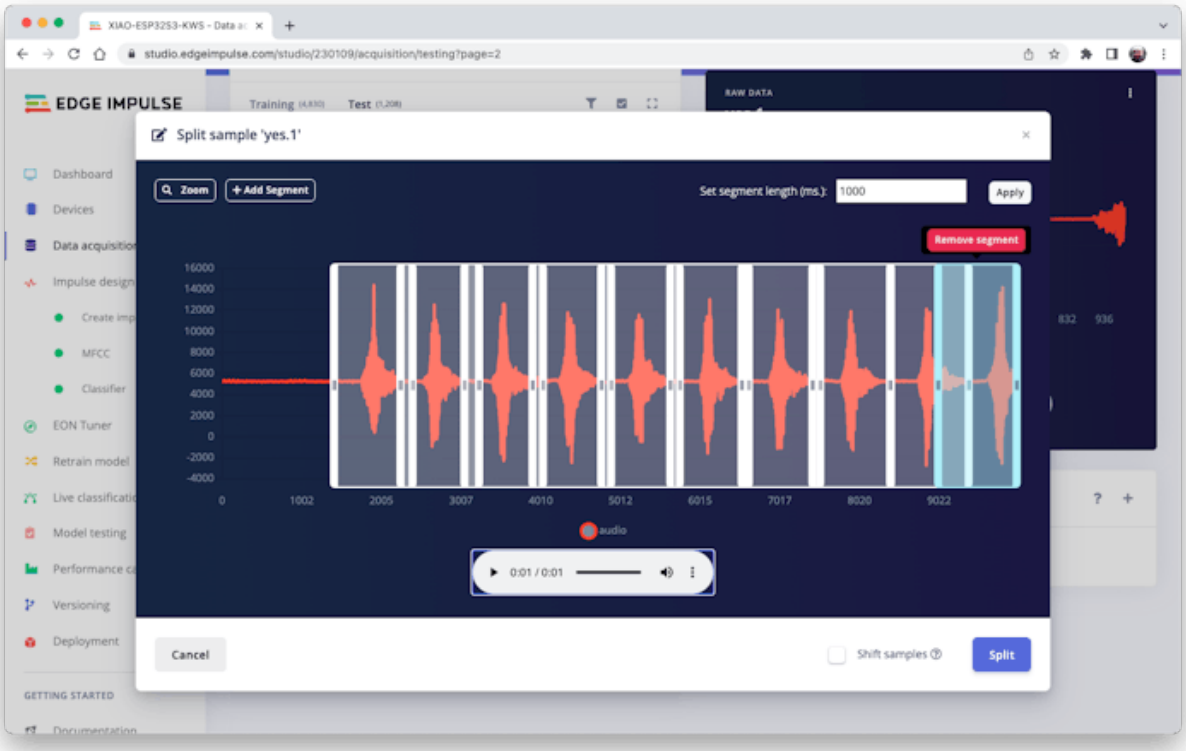


All data on Pete's dataset have a 1s length, but the samples recorded in the previous section have 10s and must be split into 1s samples to be compatible.

Click on three dots after the sample name and select Split sample.



Once inside the tool, split the data into 1-second records. If necessary, add or remove segments:



This procedure should be repeated for all samples.

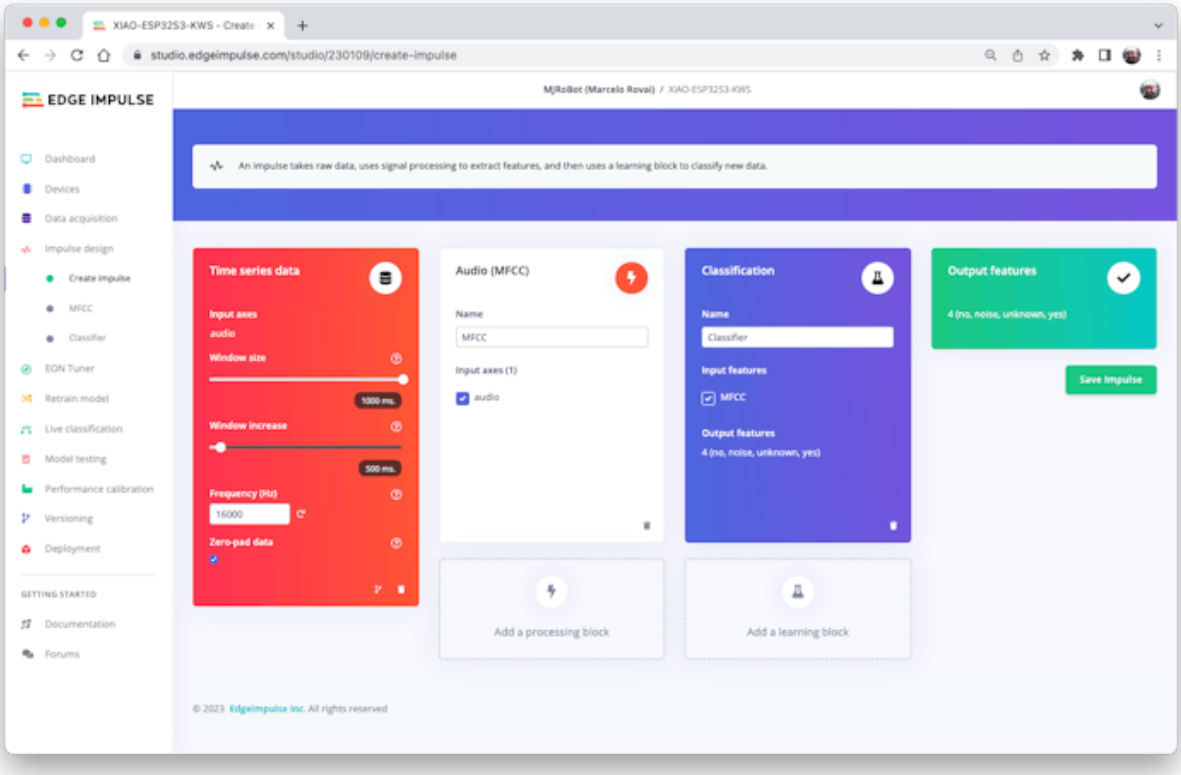
> Note: For longer audio files (minutes), first, split into 10-second segments and after that, use the tool again to get the final 1-second splits.

Suppose we do not split data automatically in train/test during upload. In that case, we can do it manually (using the three dots menu, moving samples individually) or using Perform Train / Test Split on Dashboard - Danger Zone.

> We can optionally check all datasets using the tab Data Explorer.

5.3.2 Creating Impulse (Pre-Process / Model definition)

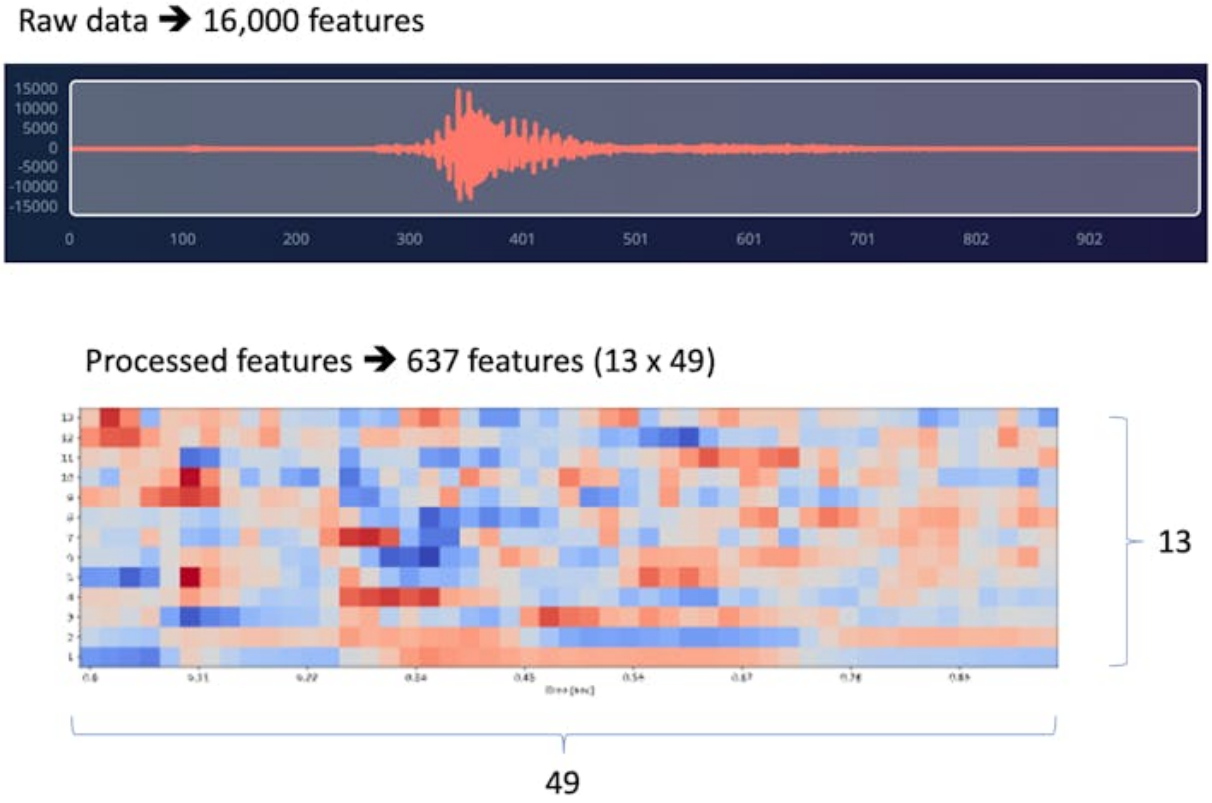
An **impulse** *takes raw data, uses signal processing to extract features, and then uses a learning block to classify new data.*



First, we will take the data points with a 1-second window, augmenting the data, sliding that window each 500ms. Note that the option zero-pad data is

set. It is essential to fill with zeros samples smaller than 1 second (in some cases, I reduced the 1000 ms window on the split tool to avoid noises and spikes).

Each 1-second audio sample should be pre-processed and converted to an image (for example, 13 x 49 x 1). We will use MFCC, which extracts features from audio signals using Mel Frequency Cepstral Coefficients, which are great for the human voice.

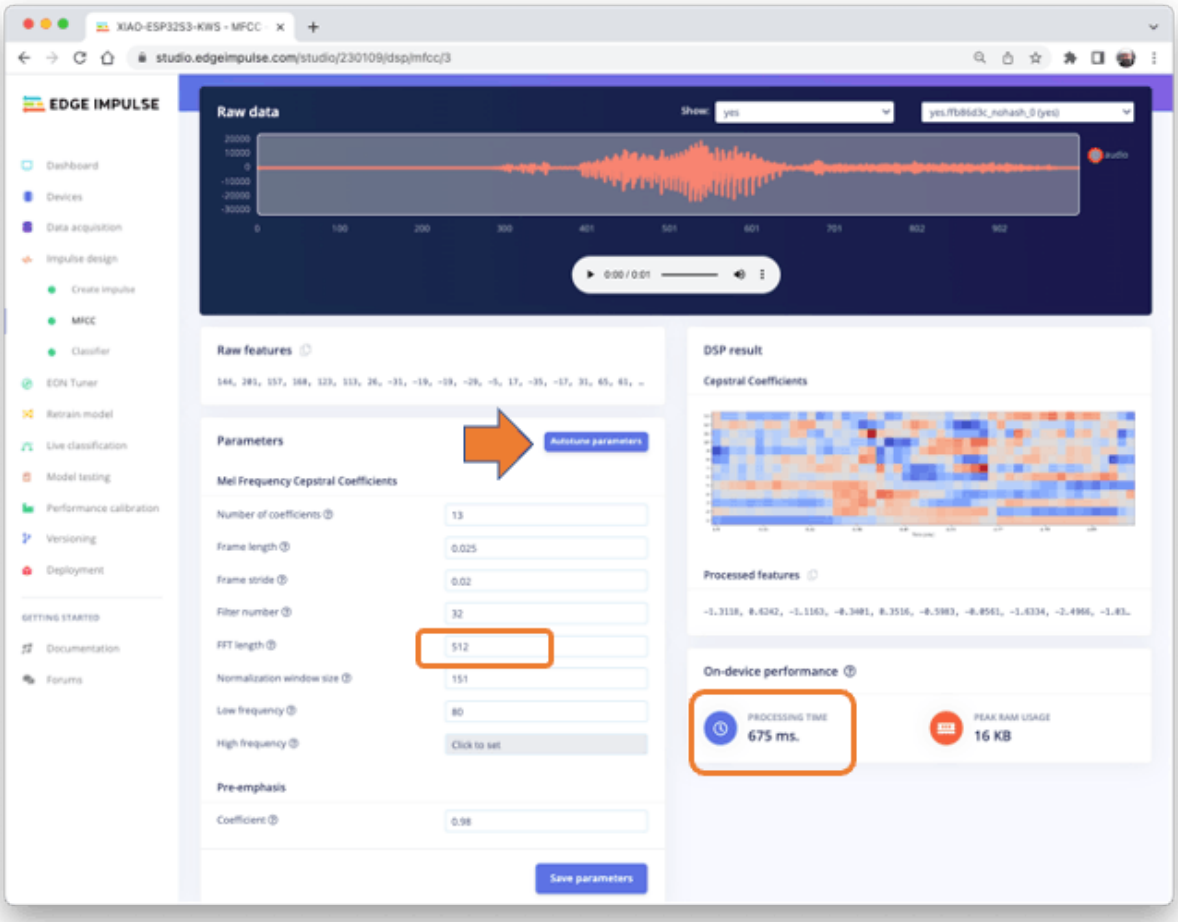


Next, we select KERAS for classification and build our model from scratch by doing Image Classification using Convolution Neural Network).

5.3.3 Pre-Processing (MFCC)

The next step is to create the images to be trained in the next phase:

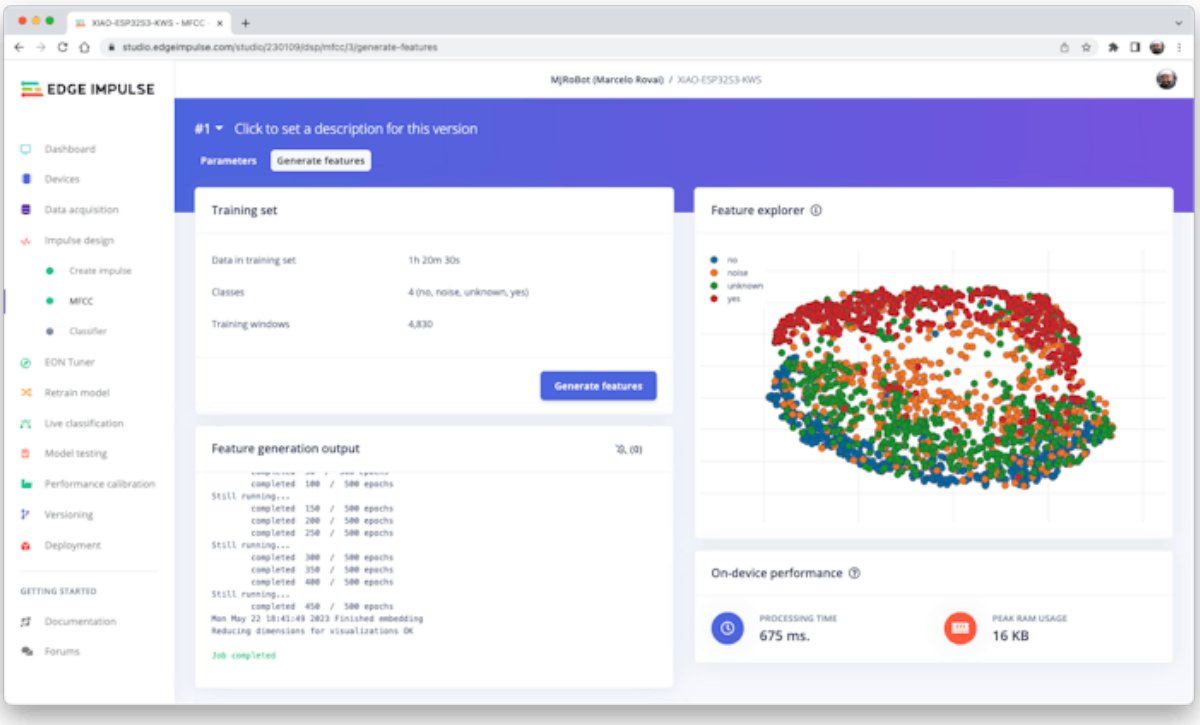
We can keep the default parameter values or take advantage of the DSP Autotuneparameters option, which we will do.



The result will not spend much memory to pre-process data (only 16KB). Still, the estimated processing time is high, 675 ms for an Espressif ESP-EYE (the closest reference available), with a 240KHz clock (same as our device), but with a smaller CPU (XTensa LX6, versus the LX7 on the ESP32S). The real inference time should be smaller.

Suppose we need to reduce the inference time later. In that case, we should return to the pre-processing stage and, for example, reduce the FFT length to 256, change the Number of coefficients, or another parameter.

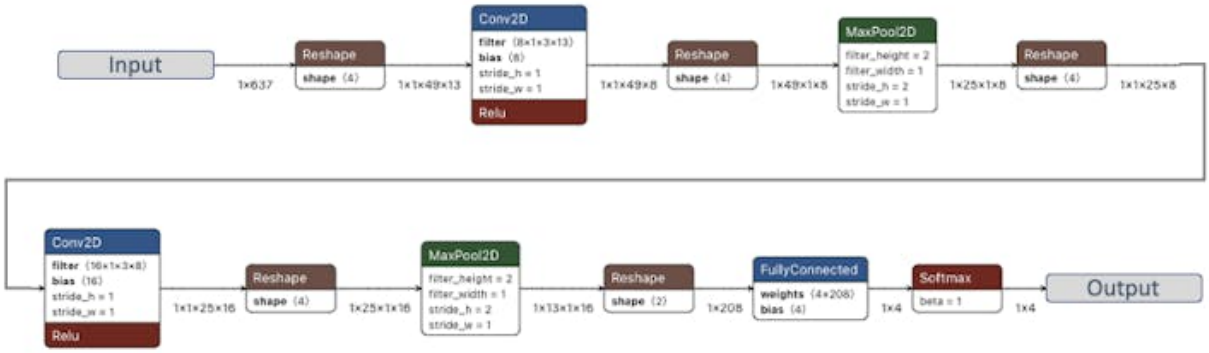
For now, let's keep the parameters defined by the Autotuning tool. Save parameters and generate the features.



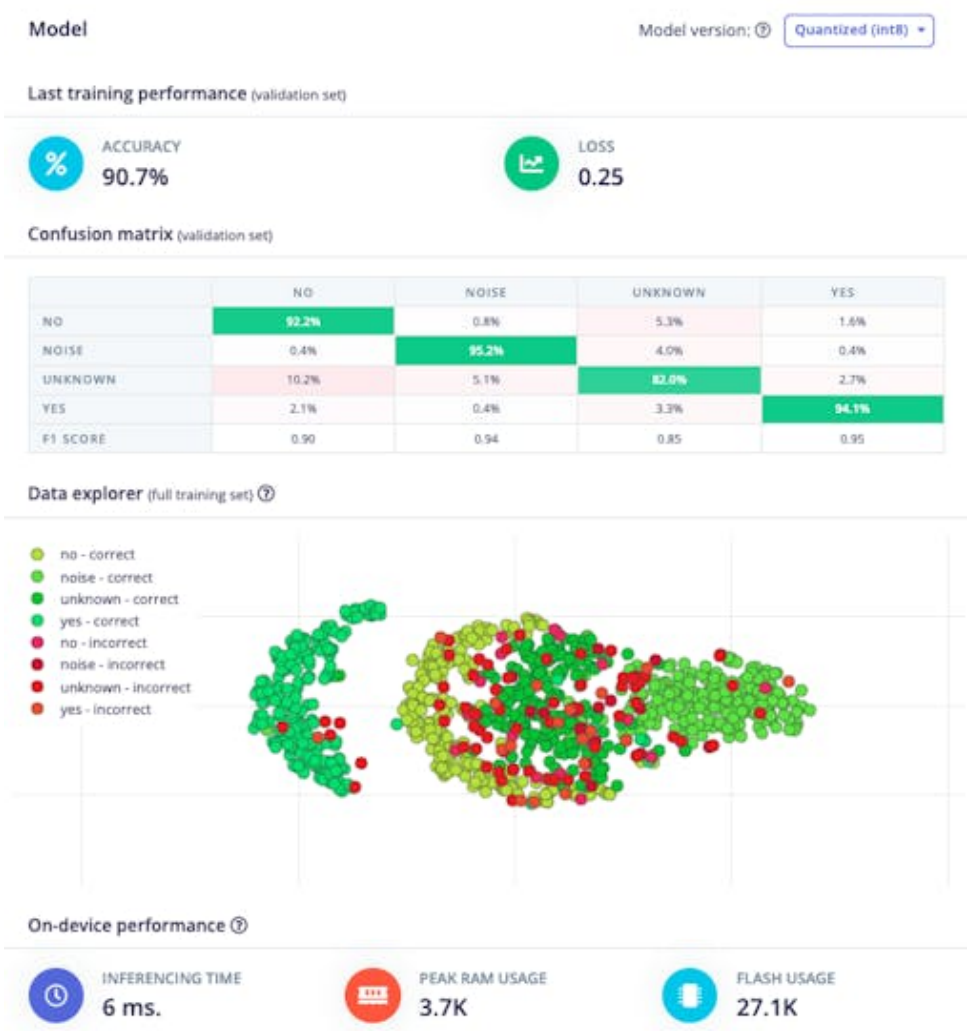
If you want to go further with converting temporal serial data into images using FFT, Spectrogram, etc., you can play with this CoLab: Audio Raw Data Analysis.

5.3.4 Model Design and Training

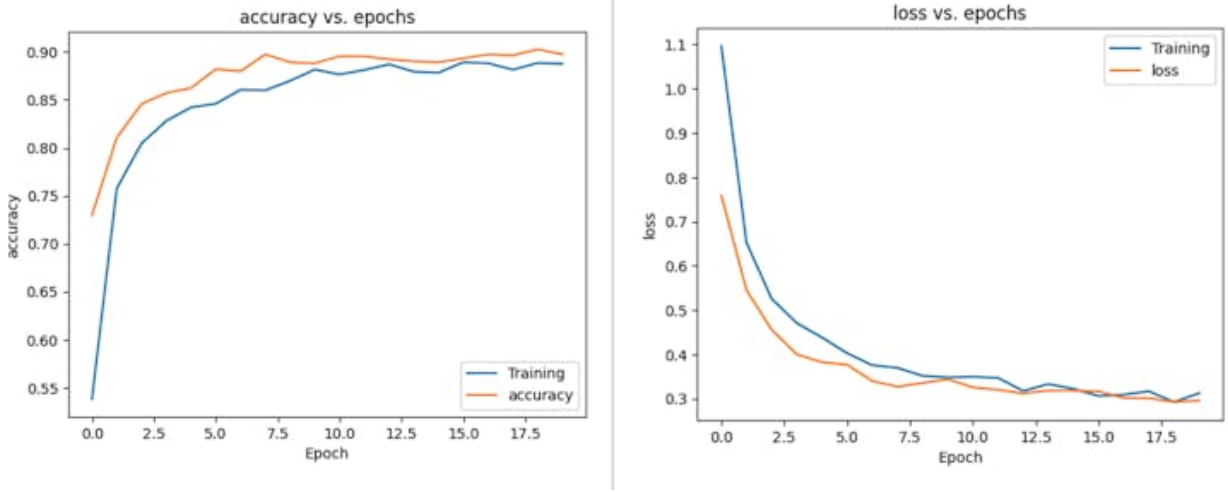
We will use a Convolution Neural Network (CNN) model. The basic architecture is defined with two blocks of Conv1D + MaxPooling (with 8 and 16 neurons, respectively) and a 0.25 Dropout. And on the last layer, after Flattening four neurons, one for each class:



As hyper-parameters, we will have a Learning Rate of 0.005 and a model that will be trained by 100 epochs. We will also include data augmentation, as some noise. The result seems OK:



If you want to understand what is happening “under the hood,” you can download the dataset and run a Jupyter Notebook playing with the code. For example, you can analyze the accuracy by each epoch:



This CoLab Notebook can explain how you can go further: KWS Classifier Project - Looking “Under the hood.”

5.4 Testing

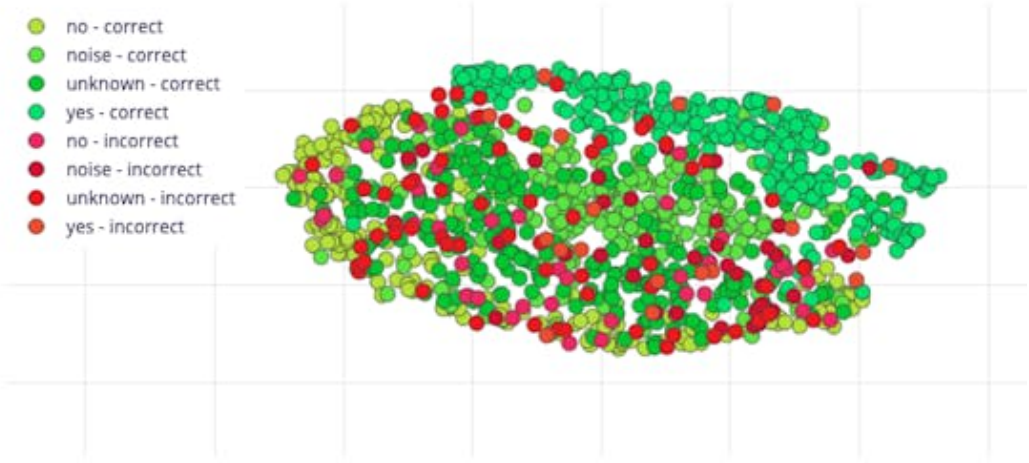
Testing the model with the data put apart before training (Test Data), we got an accuracy of approximately 87%.

Model testing results

ACCURACY
86.73%

	NO	NOISE	UNKNOWN	YES	UNCERTAIN
NO	86.3%	0.7%	3.9%	1.4%	7.7%
NOISE	0%	88.6%	3.3%	0.7%	7.5%
UNKNOWN	4.4%	2.7%	78.1%	1.7%	13.1%
YES	0.3%	0%	0.7%	93.9%	5.1%
F1 SCORE	0.90	0.92	0.84	0.95	

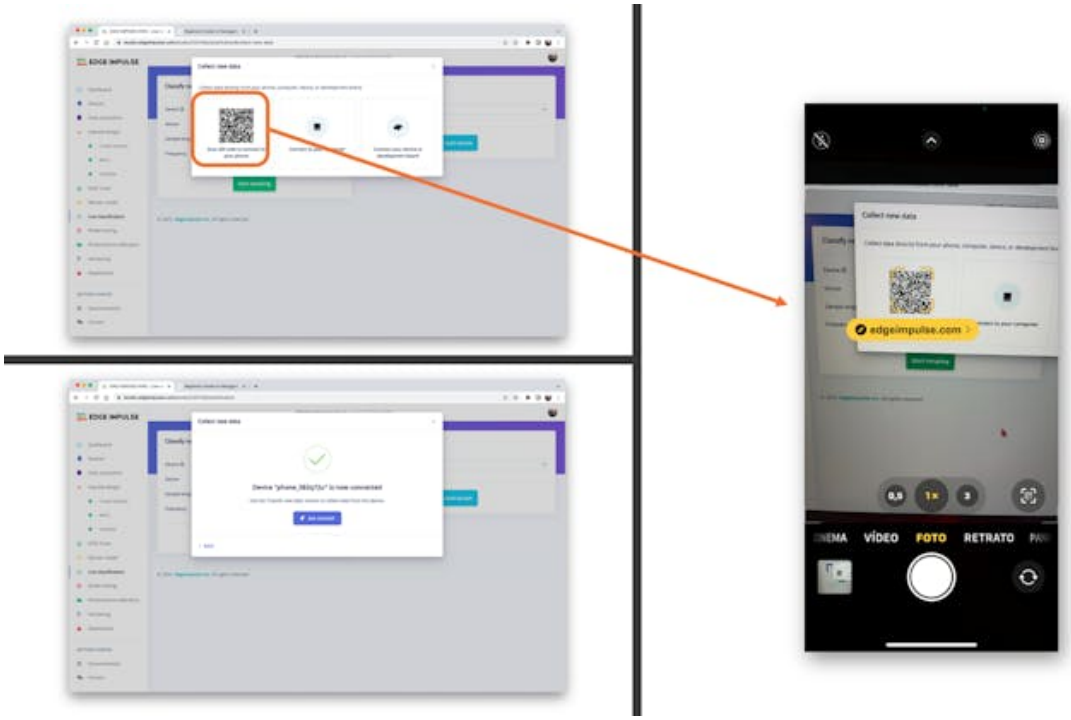
Feature explorer



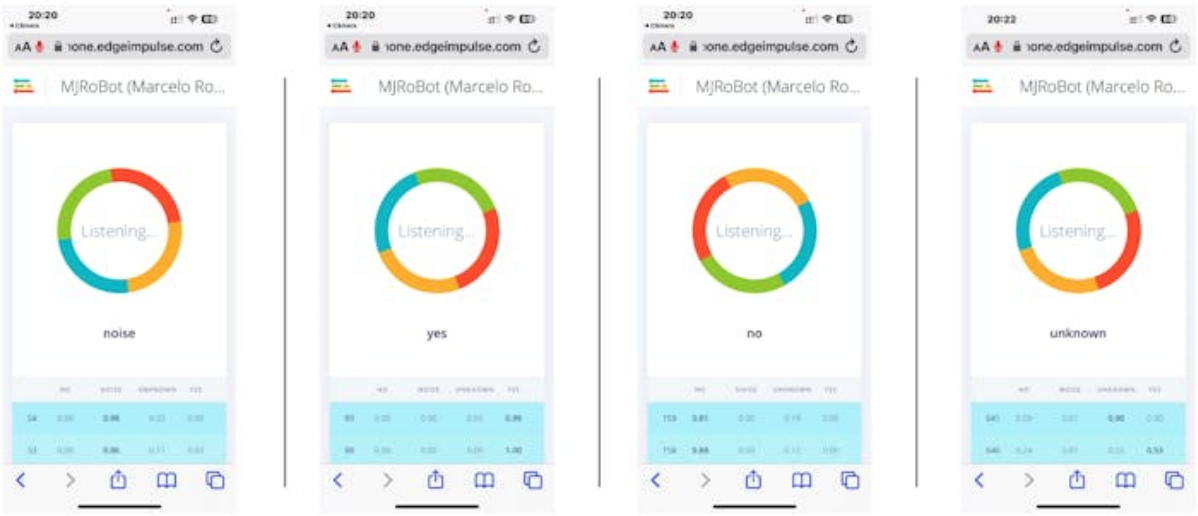
Inspecting the F1 score, we can see that for YES. We got 0.95, an excellent result once we used this keyword to “trigger” our postprocessing stage (turn on the built-in LED). Even for NO, we got 0.90. The worst result is for unknown, what is OK.

We can proceed with the project, but it is possible to perform Live Classification using a smartphone before deployment on our device. Go to the Live Classification section and click on Connect a Development board:

Point your phone to the barcode and select the link.

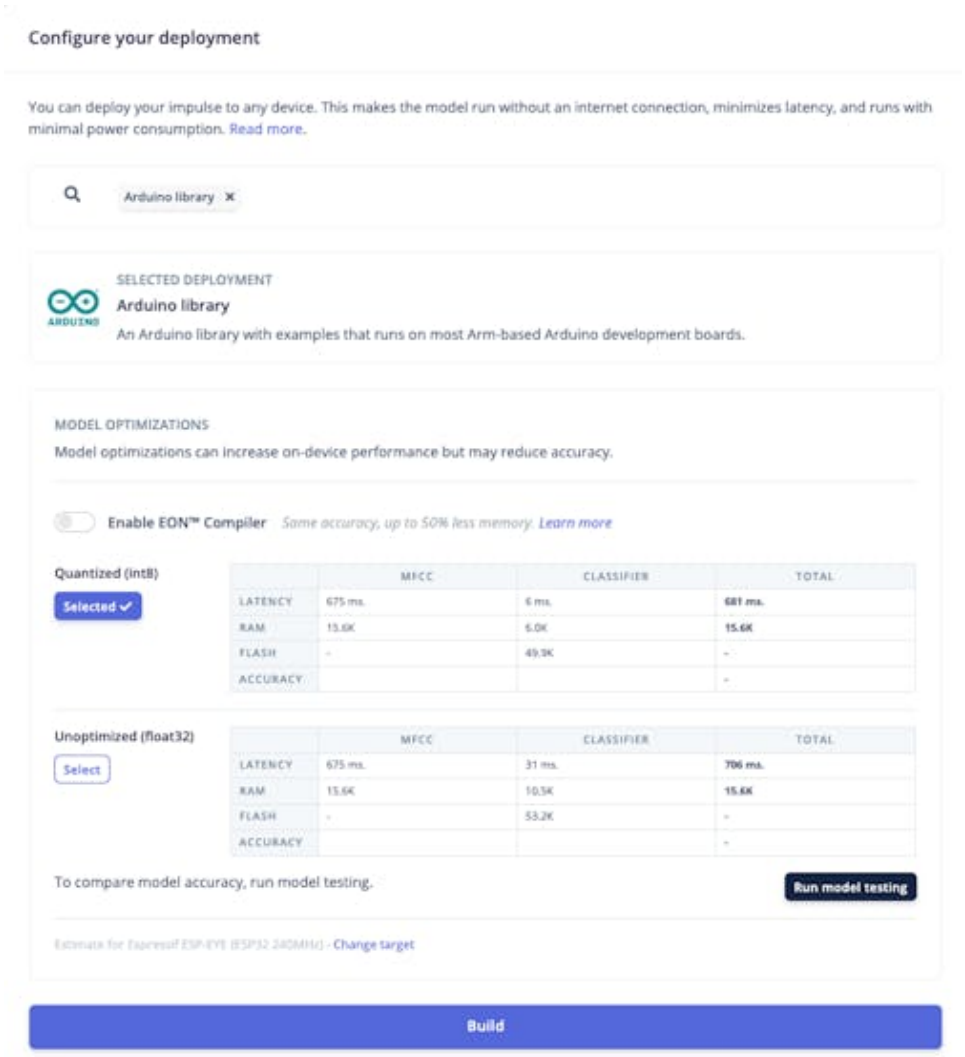


Your phone will be connected to the Studio. Select the option Classification on the app, and when it is running, start testing your keywords, confirming that the model is working with live and real data:



5.5 Deploy and Inference

The Studio will package all the needed libraries, preprocessing functions, and trained models, downloading them to your computer. You should select the option Arduino Library, and at the bottom, choose Quantized (Int8) and press the button Build.



Now it is time for a real test. We will make inferences wholly disconnected from the Studio. Let's change one of the ESP32 code examples created when you deploy the Arduino Library.

In your Arduino IDE, go to the File/Examples tab look for your project, and select esp32/esp32_microphone:

This code was created for the ESP-EYE built-in microphone, which should be adapted for our device.

Start changing the libraries to handle the I2S bus:

```
41 /* Includes ---------------------------------
42 #include <XIAO-ESP32S3-KWS_inferencing.h>
43
44 #include "freertos/FreeRTOS.h"
45 #include "freertos/task.h"
46
47 #include "driver/i2s.h"
48
```

By:

```
#include <I2S.h>
#define SAMPLE_RATE 16000U
#define SAMPLE_BITS 16
```

Initialize the IS2 microphone at setup(), including the lines:

```
void setup()
{
...
    I2S.setAllPins(-1, 42, 41, -1, -1);
    if (!I2S.begin(PDM_MONO_MODE, SAMPLE_RATE, SAMPLE_BITS)) {
      Serial.println("Failed to initialize I2S!");
    while (1) ;
...
}
```

On the static void capture_samples(void* arg) function, replace the line 153 that reads data from I2S mic:

```
145 static void capture_samples(void* arg) {
146
147   const int32_t i2s_bytes_to_read = (uint32_t)arg;
148   size_t bytes_read = i2s_bytes_to_read;
149
150   while (record_status) {
151
152     /* read data at once from i2s */
153     i2s_read((i2s_port_t)1, (void*)sampleBuffer, i2s_bytes_to_read, &bytes_read, 100);
154
```

By:

```
/* read data at once from i2s */
esp_i2s::i2s_read(esp_i2s::I2S_NUM_0,
                  (void*)sampleBuffer,
                  i2s_bytes_to_read,
                  &bytes_read, 100);
```

On function static bool microphone_inference_start(uint32_t n_samples), we should comment or delete lines 198 to 200, where the microphone initialization function is called. This is unnecessary because the I2S microphone was already initialized during the setup().

```
186 static bool microphone_inference_start(uint32_t n_samples)
187 {
188     inference.buffer = (int16_t *)malloc(n_samples * sizeof(int16_t));
189
190     if(inference.buffer == NULL) {
191         return false;
192     }
193
194     inference.buf_count  = 0;
195     inference.n_samples  = n_samples;
196     inference.buf_ready  = 0;
197
198 //    if (i2s_init(EI_CLASSIFIER_FREQUENCY)) {
199 //        ei_printf("Failed to start I2S!");
200 //    }
201
```

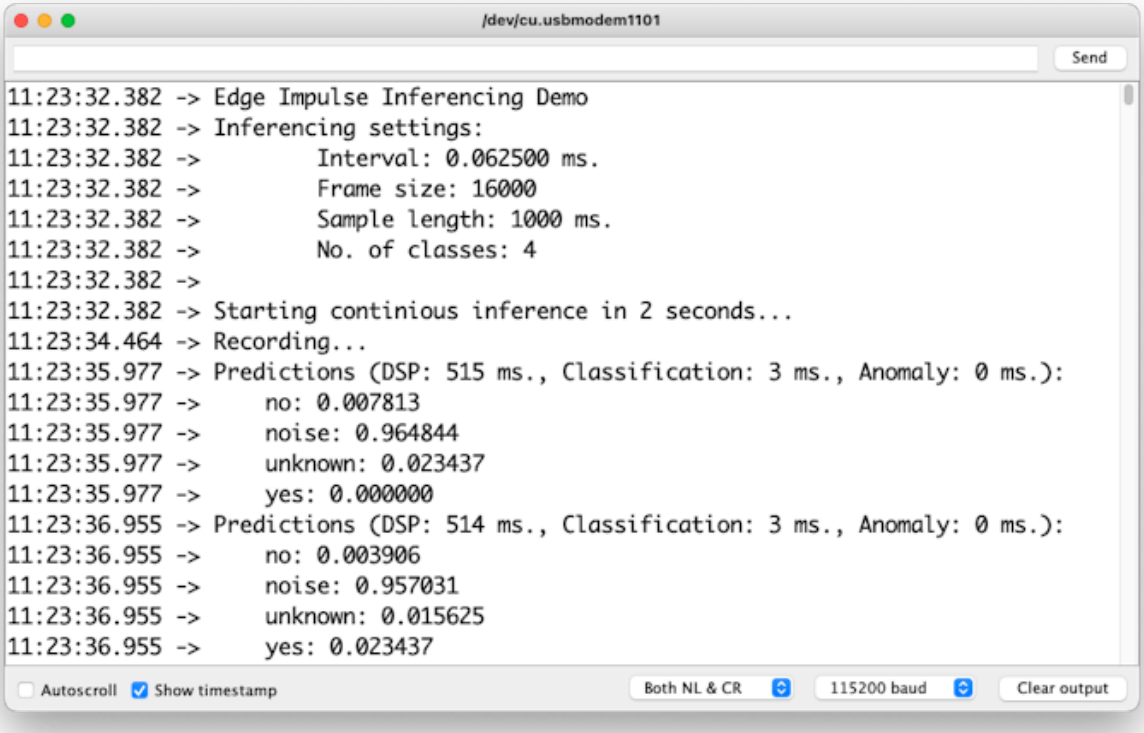
Finally, on static void microphone_inference_end(void) function, replace line 243:

```
241 static void microphone_inference_end(void)
242 {
243     i2s_deinit();
244     ei_free(inference.buffer);
245 }
```

By:

```
static void microphone_inference_end(void)
{
    free(sampleBuffer);
    ei_free(inference.buffer);
}
```

You can find the complete code on the project's GitHub. Upload the sketch to your board and test some real inferences:



5.6 Postprocessing

Now that we know the model is working by detecting our keywords, let's modify the code to see the internal LED going on every time a YES is detected.

You should initialize the LED:

```
#define LED_BUILT_IN 21
...
void setup()
{
...
  pinMode(LED_BUILT_IN, OUTPUT); // Set the pin as output
  digitalWrite(LED_BUILT_IN, HIGH); //Turn off
...
}
```

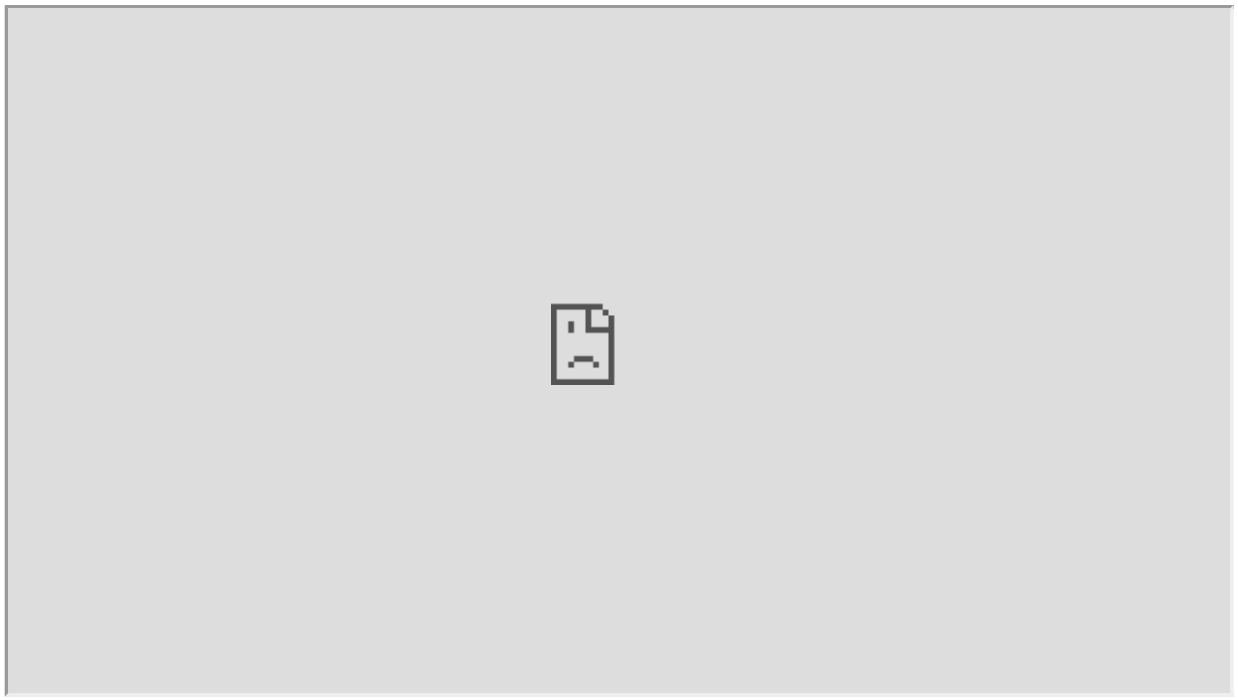
And change the // print the predictions portion of the previous code (on loop():

```
int pred_index = 0;     // Initialize pred_index
float pred_value = 0;   // Initialize pred_value

// print the predictions
ei_printf("Predictions ");
ei_printf("(DSP: %d ms., Classification: %d ms., Anomaly: %d ms.)",
     result.timing.dsp, result.timing.classification,
result.timing.anomaly);
ei_printf(": \n");
for (size_t ix = 0; ix < EI_CLASSIFIER_LABEL_COUNT; ix++) {
      ei_printf("    %s: ", result.classification[ix].label);
      ei_printf_float(result.classification[ix].value);
      ei_printf("\n");

      if (result.classification[ix].value > pred_value){
         pred_index = ix;
         pred_value = result.classification[ix].value;
      }
}

// show the inference result on LED
if (pred_index == 3){
    digitalWrite(LED_BUILT_IN, LOW); //Turn on
```

```
}
else{
   digitalWrite(LED_BUILT_IN, HIGH); //Turn off
}
```

You can find the complete code on the project's GitHub. Upload the sketch to your board and test some real inferences:

The idea is that the LED will be ON whenever the keyword YES is detected. In the same way, instead of turning on an LED, this could be a "trigger" for an external device, as we saw in the introduction.



5.7 Conclusion

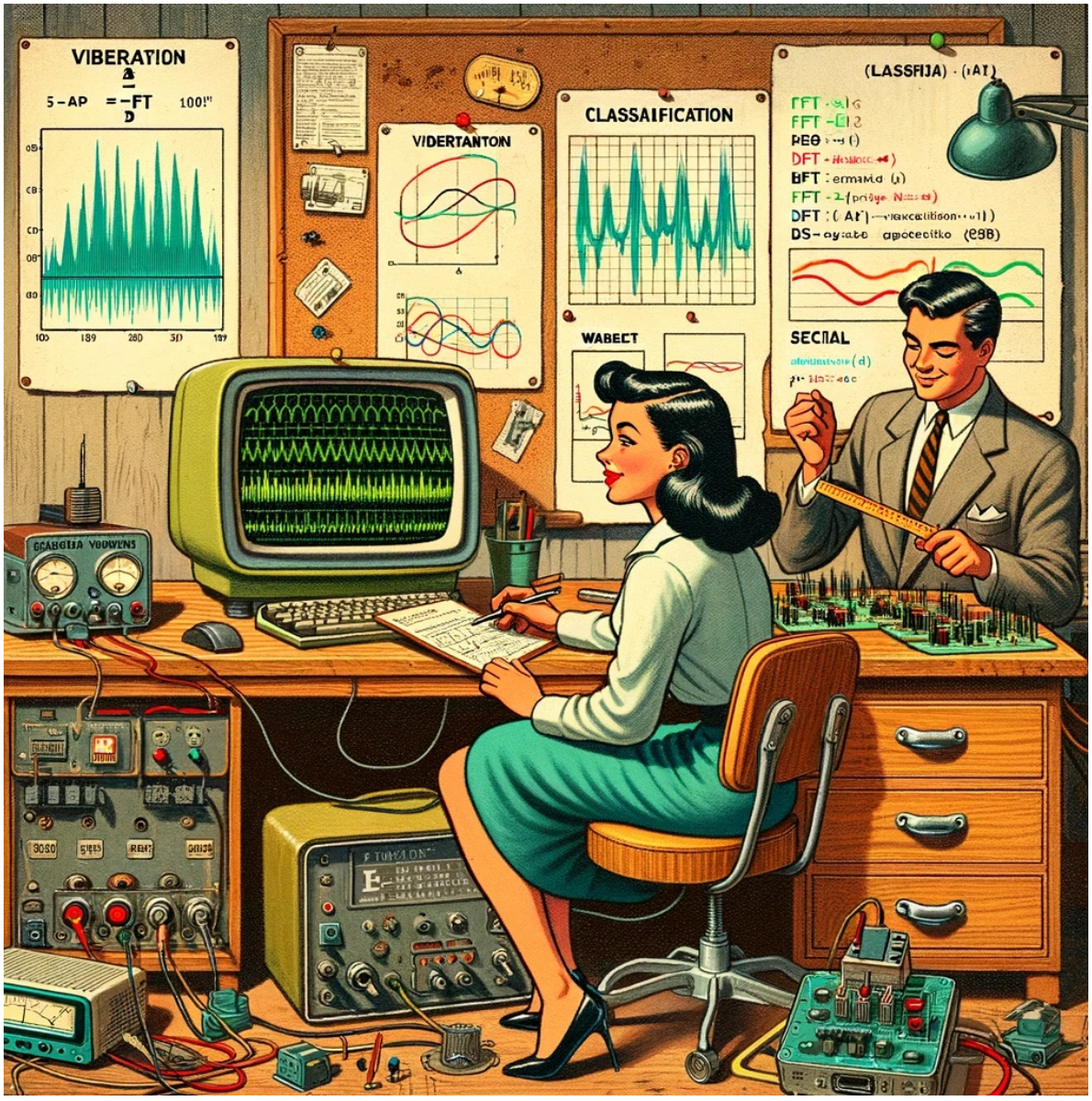
The Seeed XIAO ESP32S3 Sense is a *giant tiny device*! However, it is powerful, trustworthy, not expensive, low power, and has suitable sensors to be used on the most common embedded machine learning applications such as vision and sound. Even though Edge Impulse does not officially support XIAO ESP32S3 Sense (yet!), we realized that using the Studio for training and deployment is straightforward.

> On my GitHub repository, you will find the last version all the codes used on this project and the previous ones of the XIAO ESP32S3 series.

Before we finish, consider that Sound Classification is more than just voice. For example, you can develop TinyML projects around sound in several areas, such as:

- **Security** (Broken Glass detection)
- **Industry** (Anomaly Detection)
- **Medical** (Snore, Toss, Pulmonary diseases)
- **Nature** (Beehive control, insect sound)

6 DSP - Spectral Features



DALL·E 3 Prompt: 1950s style cartoon illustration of a Latin male and female scientist in vibration research room. The man is using a calculus ruler to examine ancient circuitry. T woman is at a computer with complex vibration graphs. The wooden table has boards wi sensors, prominently an accelerometer. A classic, rounded-back computer shows the Arduino IDE with code for LED pin assignments and machine learning algorithms for movement detection. The Serial Monitor displays FFT, classification, wavelets, and DSPs Vintage lamps, tools, and charts with FFT and Wavelets graphs complete the scene.

6.1 Introduction

TinyML projects related to motion (or vibration) involve data from IMUs (usually **accelerometers** and **Gyroscopes**). These time-series type datasets should be preprocessed before inputting them into a Machine Learning model training, which is a challenging area for embedded machine learning. Still, Edge Impulse helps overcome this complexity with its digital signal processing (DSP) preprocessing step and, more specifically, the Spectral Features Block for Inertial sensors.

But how does it work under the hood? Let's dig into it.

6.2 Extracting Features Review

Extracting features from a dataset captured with inertial sensors, such as accelerometers, involves processing and analyzing the raw data. Accelerometers measure the acceleration of an object along one or more axes (typically three, denoted as X, Y, and Z). These measurements can be used to understand various aspects of the object's motion, such as movement patterns and vibrations. Here's a high-level overview of the process:

Data collection: First, we need to gather data from the accelerometers. Depending on the application, data may be collected at different sampling rates. It's essential to ensure that the sampling rate is high enough to capture the relevant dynamics of the studied motion (the sampling rate should be at least double the maximum relevant frequency present in the signal).

Data preprocessing: Raw accelerometer data can be noisy and contain errors or irrelevant information. Preprocessing steps, such as filtering and normalization, can help clean and standardize the data, making it more suitable for feature extraction.

> The Studio does not perform normalization or standardization, so sometimes, when working with Sensor Fusion, it could be necessary to perform this step before uploading data to the Studio. This is particularly

crucial in sensor fusion projects, as seen in this tutorial, Sensor Data Fusion with Spresense and CommonSense.

Segmentation: Depending on the nature of the data and the application, dividing the data into smaller segments or **windows** may be necessary. This can help focus on specific events or activities within the dataset, making feature extraction more manageable and meaningful. The **window size** and overlap (**window span**) choice depend on the application and the frequency of the events of interest. As a rule of thumb, we should try to capture a couple of "data cycles."

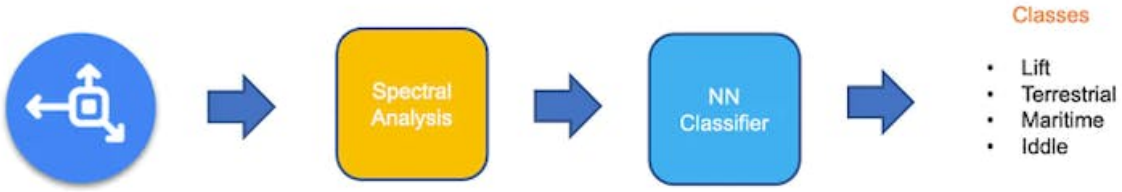
Feature extraction: Once the data is preprocessed and segmented, you can extract features that describe the motion's characteristics. Some typical features extracted from accelerometer data include:

- **Time-domain** features describe the data's statistical properties within each segment, such as mean, median, standard deviation, skewness, kurtosis, and zero-crossing rate.
- **Frequency-domain** features are obtained by transforming the data into the frequency domain using techniques like the Fast Fourier Transform (FFT). Some typical frequency-domain features include the power spectrum, spectral energy, dominant frequencies (amplitude and frequency), and spectral entropy.
- **Time-frequency** domain features combine the time and frequency domain information, such as the Short-Time Fourier Transform (STFT) or the Discrete Wavelet Transform (DWT). They can provide a more detailed understanding of how the signal's frequency content changes over time.

In many cases, the number of extracted features can be large, which may lead to overfitting or increased computational complexity. Feature selection techniques, such as mutual information, correlation-based methods, or principal component analysis (PCA), can help identify the most relevant features for a given application and reduce the dimensionality of the dataset. The Studio can help with such feature-relevant calculations.

Let's explore in more detail a typical TinyML Motion Classification project covered in this series of Hands-Ons.

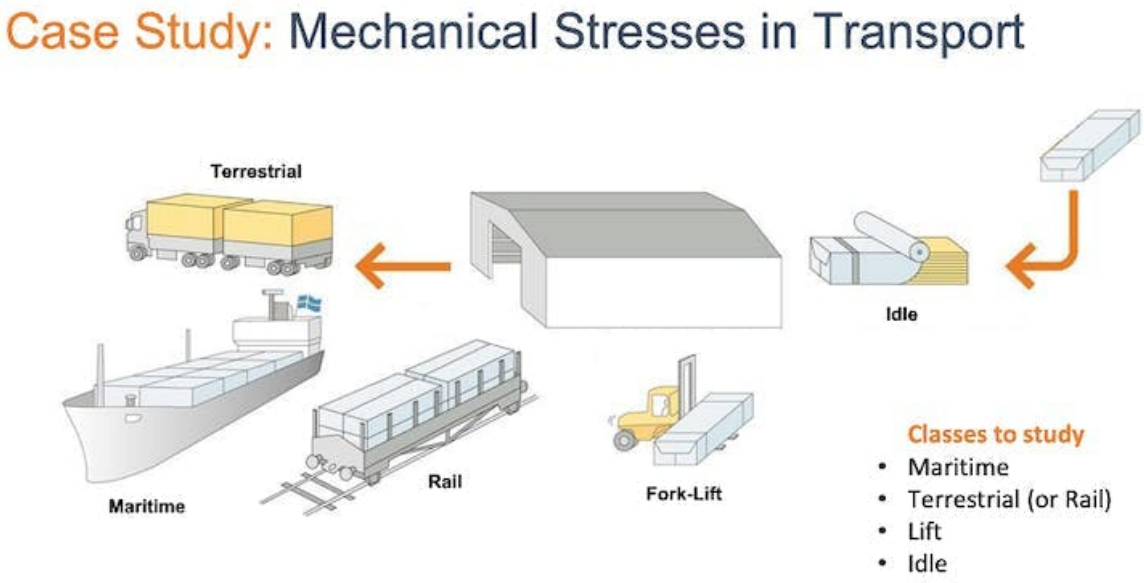
6.3 A TinyML Motion Classification project



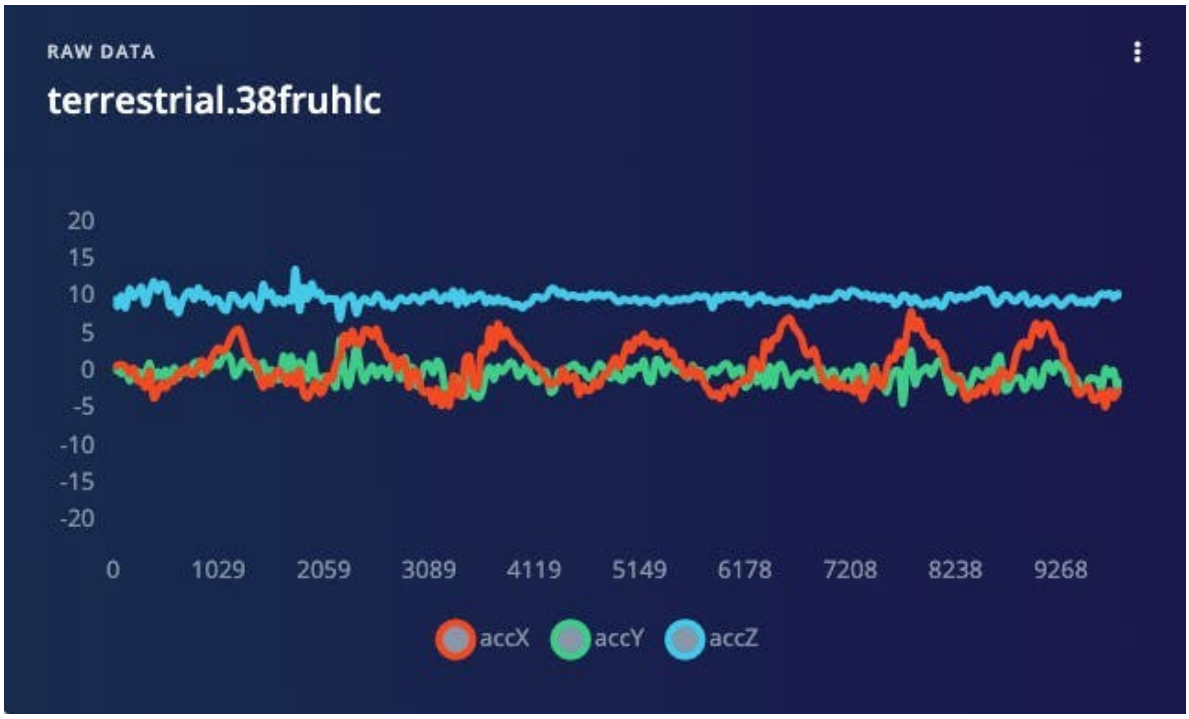
In the hands-on project, *Motion Classification and Anomaly Detection*, we simulated mechanical stresses in transport, where our problem was to classify four classes of movement:

- **Maritime** (pallets in boats)
- **Terrestrial** (pallets in a Truck or Train)
- **Lift** (pallets being handled by Fork-Lift)
- **Idle** (pallets in Storage houses)

The accelerometers provided the data on the pallet (or container).



Below is one sample (raw data) of 10 seconds, captured with a sampling frequency of 50Hz:

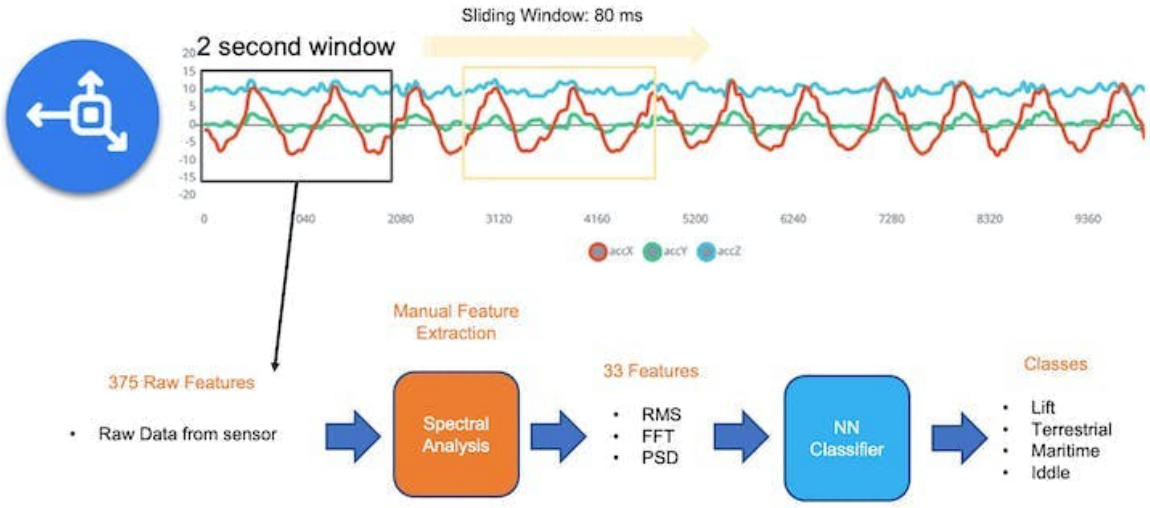


The result is similar when this analysis is done over another dataset with the same principle, using a different sampling frequency, 62.5Hz instead of 50Hz.

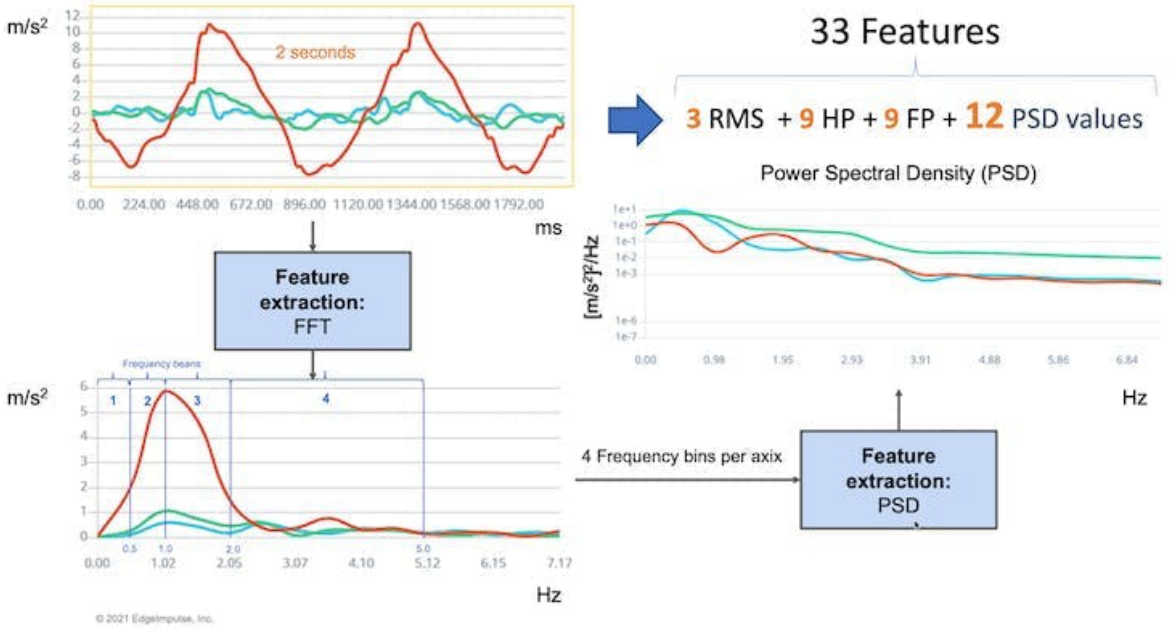
6.4 Data Pre-Processing

The raw data captured by the accelerometer (a “time series” data) should be converted to “tabular data” using one of the typical Feature Extraction methods described in the last section.

We should segment the data using a sliding window over the sample data for feature extraction. The project captured accelerometer data every 10 seconds with a sample rate of 62.5 Hz. A 2-second window captures 375 data points (3 axis x 2 seconds x 62.5 samples). The window is slid every 80ms, creating a larger dataset where each instance has 375 “raw features.”



On the Studio, the previous version (V1) of the **Spectral Analysis Block** extracted as time-domain features only the RMS, and for the frequency-domain, the peaks and frequency (using FFT) and the power characteristics (PSD) of the signal over time resulting in a fixed tabular dataset of 33 features (11 per each axis),



Those 33 features were the Input tensor of a Neural Network Classifier.

In 2022, Edge Impulse released version 2 of the Spectral Analysis block, which we will explore here.

6.4.1 Edge Impulse - Spectral Analysis Block V.2 under the hood

In Version 2, Time Domain Statistical features per axis/channel are:

- RMS
- Skewness
- Kurtosis

And the Frequency Domain Spectral features per axis/channel are:

- Spectral Power
- Skewness (in the next version)
- Kurtosis (in the next version)

In this link, we can have more details about the feature extraction.

> Clone the public project. You can also follow the explanation, playing with the code using my Google CoLab Notebook: Edge Impulse Spectral Analysis Block Notebook.

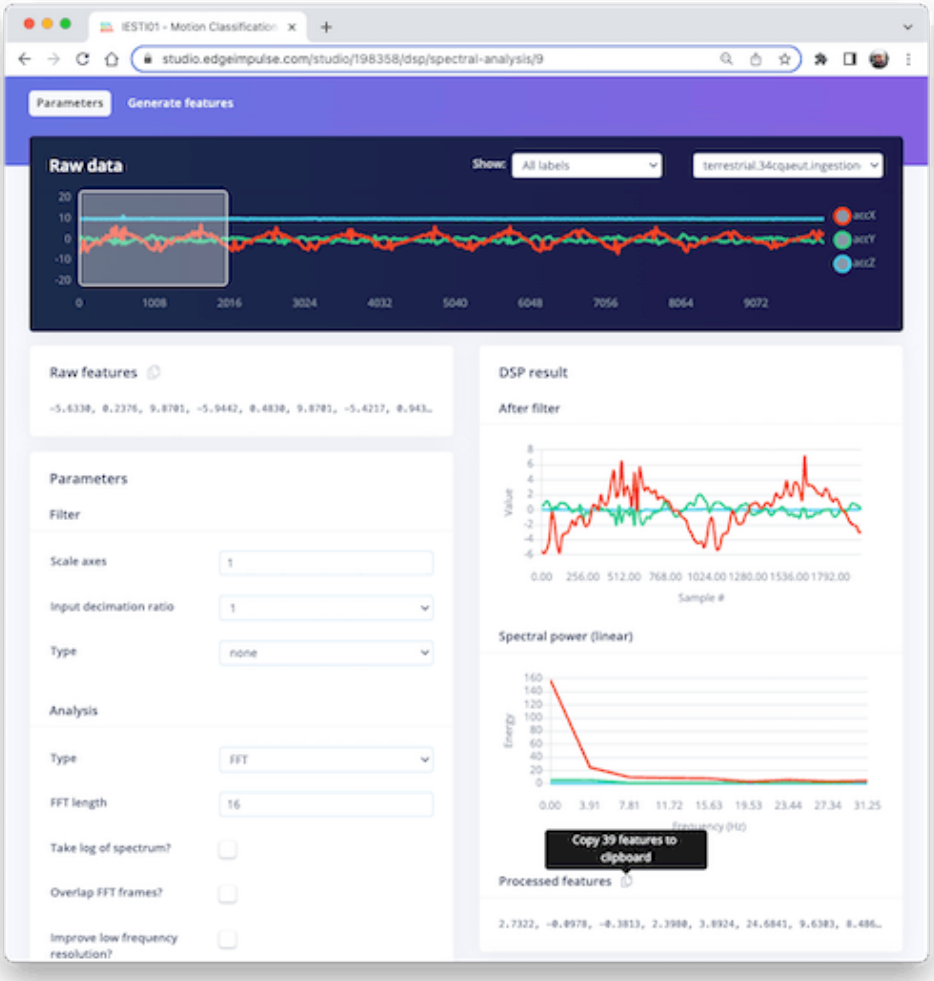
Start importing the libraries:

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import math
from scipy.stats import skew, kurtosis
from scipy import signal
from scipy.signal import welch
from scipy.stats import entropy
from sklearn import preprocessing
import pywt

plt.rcParams['figure.figsize'] = (12, 6)
plt.rcParams['lines.linewidth'] = 3
```

From the studied project, let's choose a data sample from accelerometers as below:

- Window size of 2 seconds: `[2,000]` ms
- Sample frequency: `[62.5]` Hz
- We will choose the `[None]` filter (for simplicity) and a
- FFT length: `[16]`.

```
f =  62.5 # Hertz
wind_sec = 2 # seconds
FFT_Lenght = 16
axis = ['accX', 'accY', 'accZ']
n_sensors = len(axis)
```



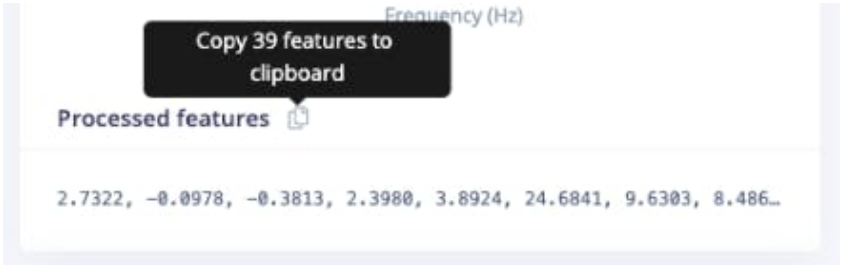
Selecting the *Raw Features* on the Studio Spectral Analysis tab, we can copy all 375 data points of a particular 2-second window to the clipboard.

Paste the data points to a new variable *data*:

```
data=[-5.6330, 0.2376, 9.8701, -5.9442, 0.4830, 9.8701, -5.4217, ...]
No_raw_features = len(data)
N = int(No_raw_features/n_sensors)
```

The total raw features are 375, but we will work with each axis individually, where N= 125 (number of samples per axis).

We aim to understand how Edge Impulse gets the processed features.



So, you should also past the processed features on a variable (to compare the calculated features in Python with the ones provided by the Studio) :

```
features = [2.7322, -0.0978, -0.3813, 2.3980, 3.8924, 24.6841, 9.6303,
N_feat = len(features)
N_feat_axis = int(N_feat/n_sensors)
```

The total number of processed features is 39, which means 13 features/axis.

Looking at those 13 features closely, we will find 3 for the time domain (RMS, Skewness, and Kurtosis):

- `[rms] [skew] [kurtosis]`

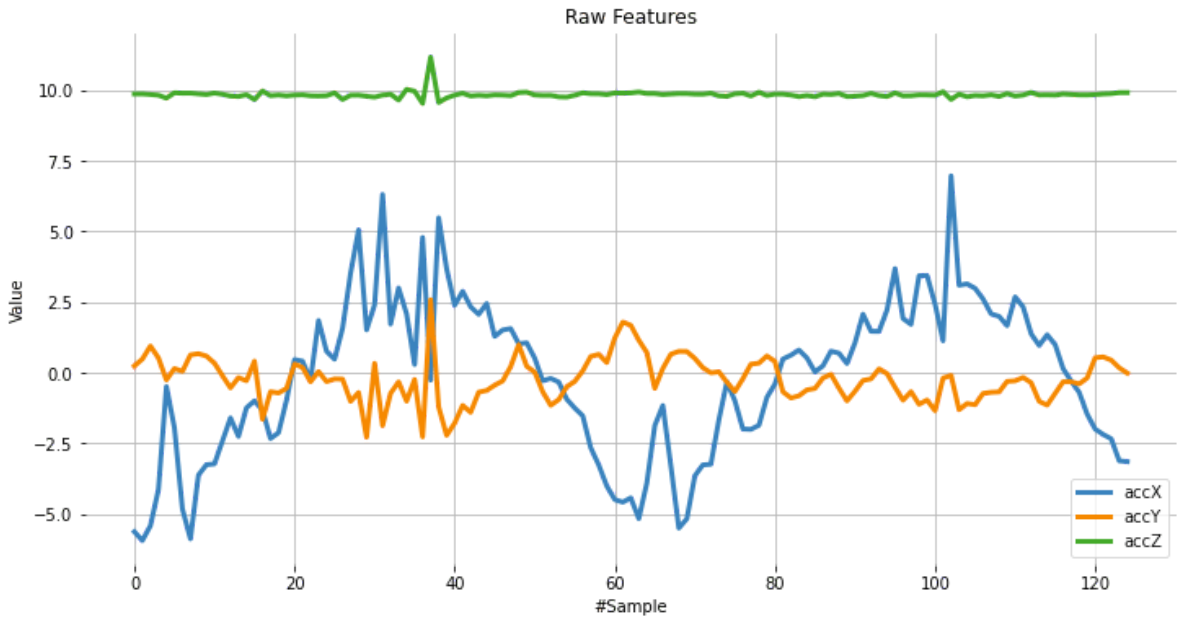
and 10 for the frequency domain (we will return to this later).

- `[spectral skew][spectral kurtosis][Spectral Power 1] ... [Spectral Power 8]`

Splitting raw data per sensor

The data has samples from all axes; let's split and plot them separately:

```
def plot_data(sensors, axis, title):
    [plt.plot(x, label=y) for x,y in zip(sensors, axis)]
    plt.legend(loc='lower right')
    plt.title(title)
    plt.xlabel('#Sample')
    plt.ylabel('Value')
    plt.box(False)
    plt.grid()
    plt.show()

accX = data[0::3]
accY = data[1::3]
accZ = data[2::3]
sensors = [accX, accY, accZ]
plot_data(sensors, axis, 'Raw Features')
```



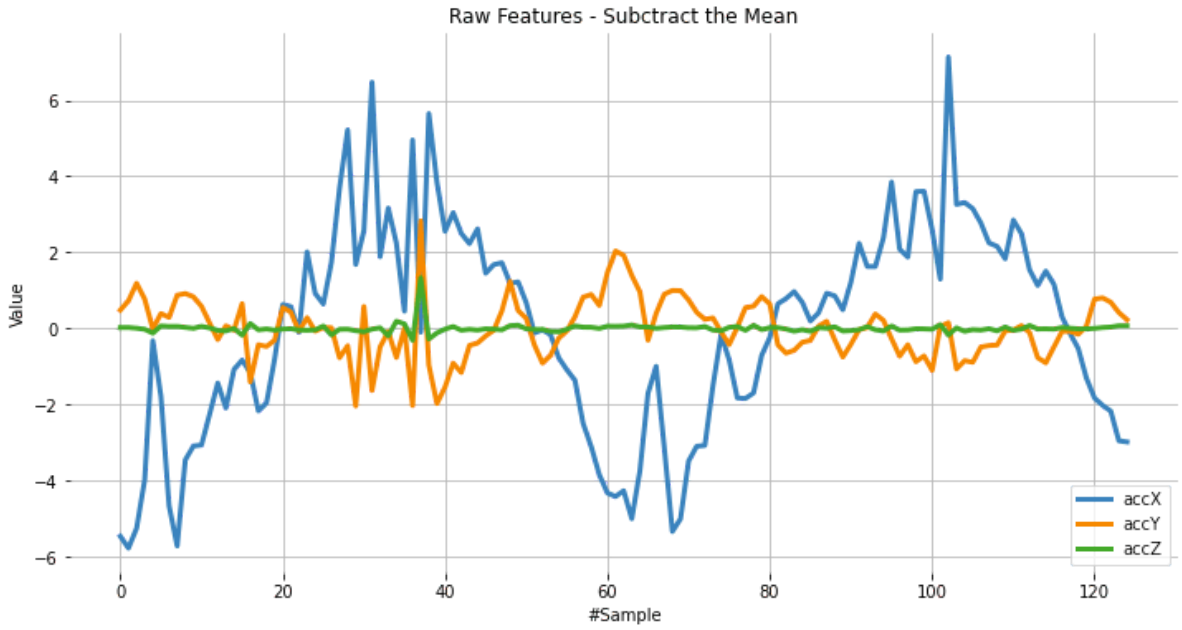
Subtracting the mean

Next, we should subtract the mean from the *data*. Subtracting the mean from a data set is a common data pre-processing step in statistics and machine learning. The purpose of subtracting the mean from the data is to center the data around zero. This is important because it can reveal patterns and relationships that might be hidden if the data is not centered.

Here are some specific reasons why subtracting the mean can be helpful:

- It simplifies analysis: By centering the data, the mean becomes zero, making some calculations simpler and easier to interpret.
- It removes bias: If the data is biased, subtracting the mean can remove it and allow for a more accurate analysis.
- It can reveal patterns: Centering the data can help uncover patterns that might be hidden if the data is not centered. For example, centering the data can help you identify trends over time if you analyze a time series dataset.
- It can improve performance: In some machine learning algorithms, centering the data can improve performance by reducing the influence of outliers and making the data more easily comparable. Overall, subtracting the mean is a simple but powerful technique that can be used to improve the analysis and interpretation of data.

```
dtmean = [(sum(x)/len(x)) for x in sensors]
[print('mean_'+x+'= ', round(y, 4)) for x,y in zip(axis, dtmean)][0]

accX = [(x - dtmean[0]) for x in accX]
accY = [(x - dtmean[1]) for x in accY]
accZ = [(x - dtmean[2]) for x in accZ]
sensors = [accX, accY, accZ]

plot_data(sensors, axis, 'Raw Features - Subctract the Mean')
```



6.5 Time Domain Statistical features

RMS Calculation

The RMS value of a set of values (or a continuous-time waveform) is the square root of the arithmetic mean of the squares of the values or the square of the function that defines the continuous waveform. In physics, the RMS value of an electrical current is defined as the “value of the direct current that dissipates the same power in a resistor.”

In the case of a set of n values $\{x1, x2, \ldots, xn\}$, the RMS is:

$$x_{\mathrm{RMS}} = \sqrt{\frac{1}{n}\left(x_1^2 + x_2^2 + \cdots + x_n^2\right)} .$$

NOTE that the RMS value is different for the original raw data, and after subtracting the mean

```
# Using numpy and standartized data (subtracting mean)
rms = [np.sqrt(np.mean(np.square(x))) for x in sensors]
```

We can compare the calculated RMS values here with the ones presented by Edge Impulse:

```
[print('rms_'+x+'= ', round(y, 4)) for x,y in zip(axis, rms)][0]
print("\nCompare with Edge Impulse result features")
print(features[0:N_feat:N_feat_axis])
```

```
rms_accX= 2.7322

rms_accY= 0.7833

rms_accZ= 0.1383
```

Compared with Edge Impulse result features:

```
[2.7322, 0.7833, 0.1383]
```

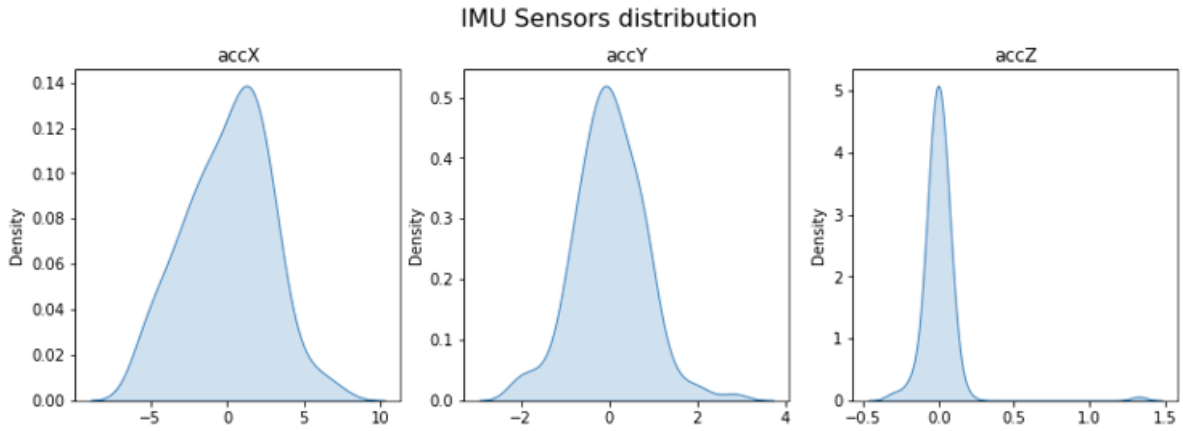
Skewness and kurtosis calculation

In statistics, skewness and kurtosis are two ways to measure the **shape of a distribution**.

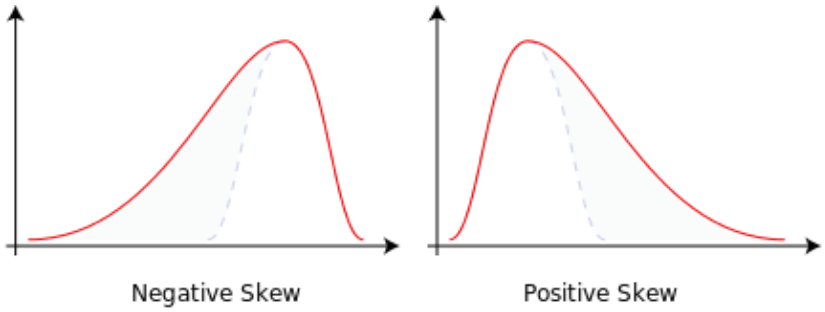
Here, we can see the sensor values distribution:

```
fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(13, 4))
sns.kdeplot(accX, fill=True, ax=axes[0])
sns.kdeplot(accY, fill=True, ax=axes[1])
sns.kdeplot(accZ, fill=True, ax=axes[2])
axes[0].set_title('accX')
axes[1].set_title('accY')
axes[2].set_title('accZ')
```

```
plt.suptitle('IMU Sensors distribution', fontsize=16, y=1.02)
plt.show()
```



Skewness is a measure of the asymmetry of a distribution. This value can be positive or negative.



- A negative skew indicates that the tail is on the left side of the distribution, which extends towards more negative values.
- A positive skew indicates that the tail is on the right side of the distribution, which extends towards more positive values.
- A zero value indicates no skewness in the distribution at all, meaning the distribution is perfectly symmetrical.

```
skew = [skew(x, bias=False) for x in sensors]
[print('skew_'+x+'= ', round(y, 4)) for x,y in zip(axis, skew)][0]
print("\nCompare with Edge Impulse result features")
features[1:N_feat:N_feat_axis]
```

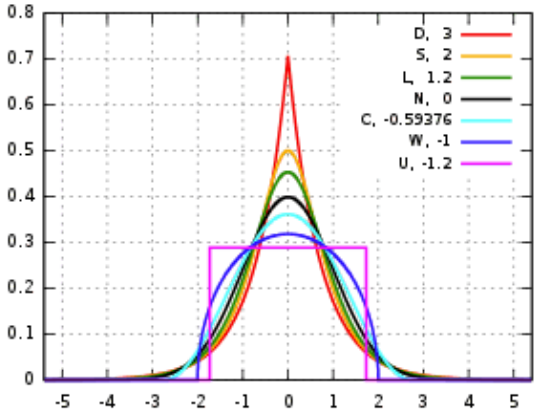
```
skew_accX= -0.099

skew_accY= 0.1756
```

```
skew_accZ= 6.9463
```

Compared with Edge Impulse result features:

```
[-0.0978, 0.1735, 6.8629]
```

Kurtosis is a measure of whether or not a distribution is heavy-tailed or light-tailed relative to a normal distribution.



- The kurtosis of a normal distribution is zero.
- If a given distribution has a negative kurtosis, it is said to be playkurtic, which means it tends to produce fewer and less extreme outliers than the normal distribution.
- If a given distribution has a positive kurtosis , it is said to be leptokurtic, which means it tends to produce more outliers than the normal distribution.

```
kurt = [kurtosis(x, bias=False) for x in sensors]
[print('kurt_'+x+'= ', round(y, 4)) for x,y in zip(axis, kurt)][0]
print("\nCompare with Edge Impulse result features")
features[2:N_feat:N_feat_axis]
```

```
kurt_accX= -0.3475

kurt_accY= 1.2673

kurt_accZ= 68.1123
```

Compared with Edge Impulse result features:

```
[-0.3813, 1.1696, 65.3726]
```

6.6 Spectral features

The filtered signal is passed to the Spectral power section, which computes the **FFT** to generate the spectral features.

Since the sampled window is usually larger than the FFT size, the window will be broken into frames (or “sub-windows”), and the FFT is calculated over each frame.

FFT length - The FFT size. This determines the number of FFT bins and the resolution of frequency peaks that can be separated. A low number means more signals will average together in the same FFT bin, but it also reduces the number of features and model size. A high number will separate more signals into separate bins, generating a larger model.

- The total number of Spectral Power features will vary depending on how you set the filter and FFT parameters. With No filtering, the number of features is 1/2 of the FFT Length.

Spectral Power - Welch’s method

We should use Welch’s method to split the signal on the frequency domain in bins and calculate the power spectrum for each bin. This method divides the signal into overlapping segments, applies a window function to each segment, computes the periodogram of each segment using DFT, and averages them to obtain a smoother estimate of the power spectrum.

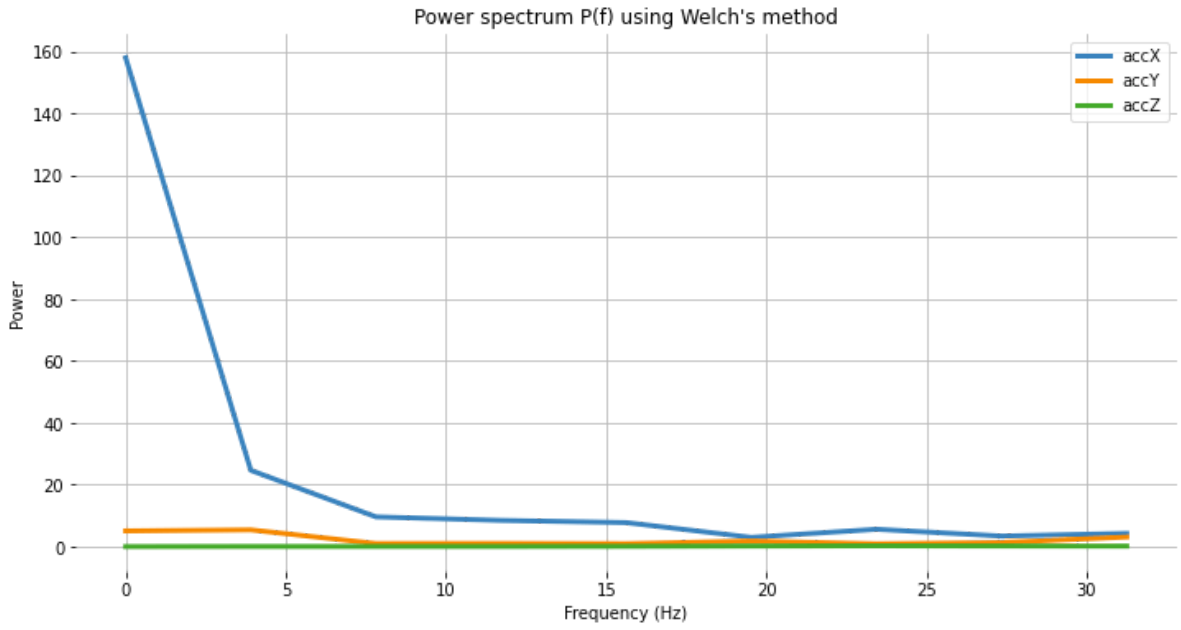
```
# Function used by Edge Impulse instead of scipy.signal.welch().
def welch_max_hold(fx, sampling_freq, nfft, n_overlap):
    n_overlap = int(n_overlap)
    spec_powers = [0 for _ in range(nfft//2+1)]
    ix = 0
    while ix <= len(fx):
        # Slicing truncates if end_idx > len, and rfft will auto-zero p
        fft_out = np.abs(np.fft.rfft(fx[ix:ix+nfft], nfft))
        spec_powers = np.maximum(spec_powers, fft_out**2/nfft)
        ix = ix + (nfft-n_overlap)
    return np.fft.rfftfreq(nfft, 1/sampling_freq), spec_powers
```

Applying the above function to 3 signals:

```
fax,Pax = welch_max_hold(accX, fs, FFT_Lenght, 0)
fay,Pay = welch_max_hold(accY, fs, FFT_Lenght, 0)
faz,Paz = welch_max_hold(accZ, fs, FFT_Lenght, 0)
specs = [Pax, Pay, Paz ]
```

We can plot the Power Spectrum P(f):

```
plt.plot(fax,Pax, label='accX')
plt.plot(fay,Pay, label='accY')
plt.plot(faz,Paz, label='accZ')
plt.legend(loc='upper right')
plt.xlabel('Frequency (Hz)')
#plt.ylabel('PSD [V**2/Hz]')
plt.ylabel('Power')
plt.title('Power spectrum P(f) using Welch's method')
plt.grid()
plt.box(False)
plt.show()
```



Besides the Power Spectrum, we can also include the skewness and kurtosis of the features in the frequency domain (should be available on a new version):

```
spec_skew = [skew(x, bias=False) for x in specs]
spec_kurtosis = [kurtosis(x, bias=False) for x in specs]
```

Let's now list all Spectral features per axis and compare them with EI:

```
print("EI Processed Spectral features (accX): ")
print(features[3:N_feat_axis][0:])
print("\nCalculated features:")
print (round(spec_skew[0],4))
print (round(spec_kurtosis[0],4))
[print(round(x, 4)) for x in Pax[1:]][0]
```

EI Processed Spectral features (accX):

2.398, 3.8924, 24.6841, 9.6303, 8.4867, 7.7793, 2.9963, 5.6242, 3.4198, 4.2735

Calculated features:

2.9069 8.5569 24.6844 9.6304 8.4865 7.7794 2.9964 5.6242 3.4198 4.2736

```
print("EI Processed Spectral features (accY): ")
print(features[16:26][0:]) #13: 3+N_feat_axis;  26 = 2x N_feat_axis
print("\nCalculated features:")
print (round(spec_skew[1],4))
print (round(spec_kurtosis[1],4))
[print(round(x, 4)) for x in Pay[1:]][0]
```

EI Processed Spectral features (accY):

0.9426, -0.8039, 5.429, 0.999, 1.0315, 0.9459, 1.8117, 0.9088, 1.3302, 3.112

Calculated features:

1.1426 -0.3886 5.4289 0.999 1.0315 0.9458 1.8116 0.9088 1.3301 3.1121

```
print("EI Processed Spectral features (accZ): ")
print(features[29:][0:]) #29: 3+(2*N_feat_axis);
print("\nCalculated features:")
print (round(spec_skew[2],4))
print (round(spec_kurtosis[2],4))
[print(round(x, 4)) for x in Paz[1:]][0]
```

EI Processed Spectral features (accZ):

0.3117, -1.3812, 0.0606, 0.057, 0.0567, 0.0976, 0.194, 0.2574, 0.2083, 0.166

Calculated features:

0.3781 -1.4874 0.0606 0.057 0.0567 0.0976 0.194 0.2574 0.2083 0.166

6.7 Time-frequency domain

6.7.1 Wavelets

Wavelet is a powerful technique for analyzing signals with transient features or abrupt changes, such as spikes or edges, which are difficult to interpret with traditional Fourier-based methods.

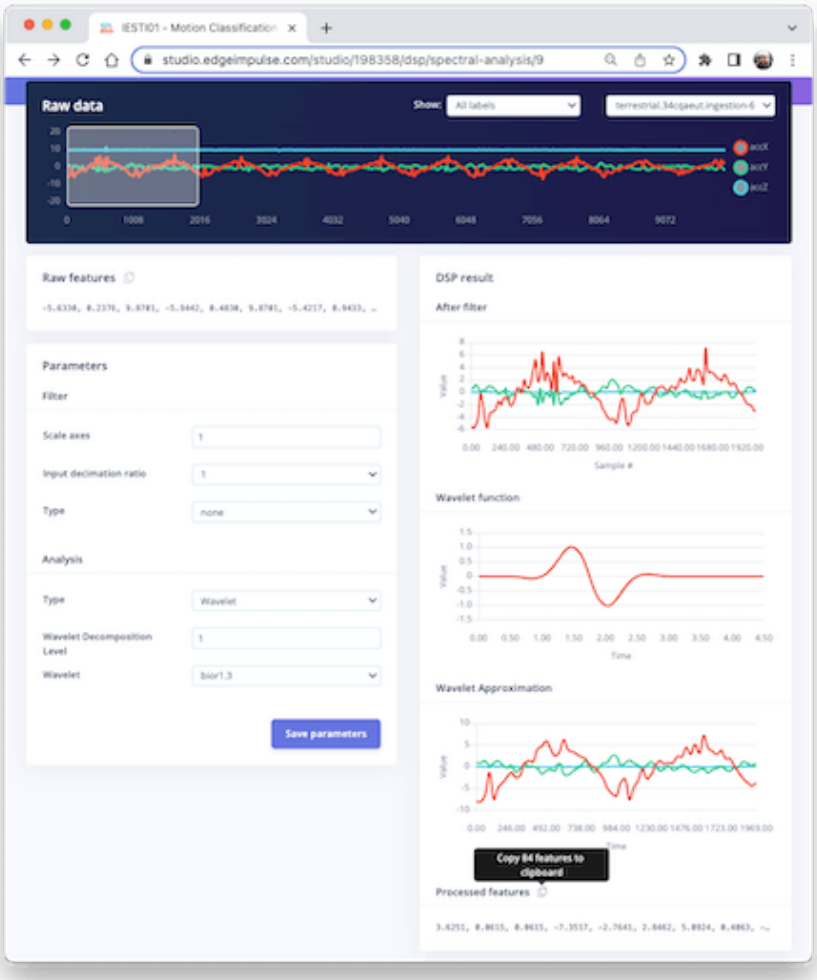
Wavelet transforms work by breaking down a signal into different frequency components and analyzing them individually. The transformation is achieved by convolving the signal with a **wavelet function**, a small waveform centered at a specific time and frequency. This process effectively decomposes the signal into different frequency bands, each of which can be analyzed separately.

One of the critical benefits of wavelet transforms is that they allow for time-frequency analysis, which means that they can reveal the frequency content of a signal as it changes over time. This makes them particularly useful for analyzing non-stationary signals, which vary over time.

Wavelets have many practical applications, including signal and image compression, denoising, feature extraction, and image processing.

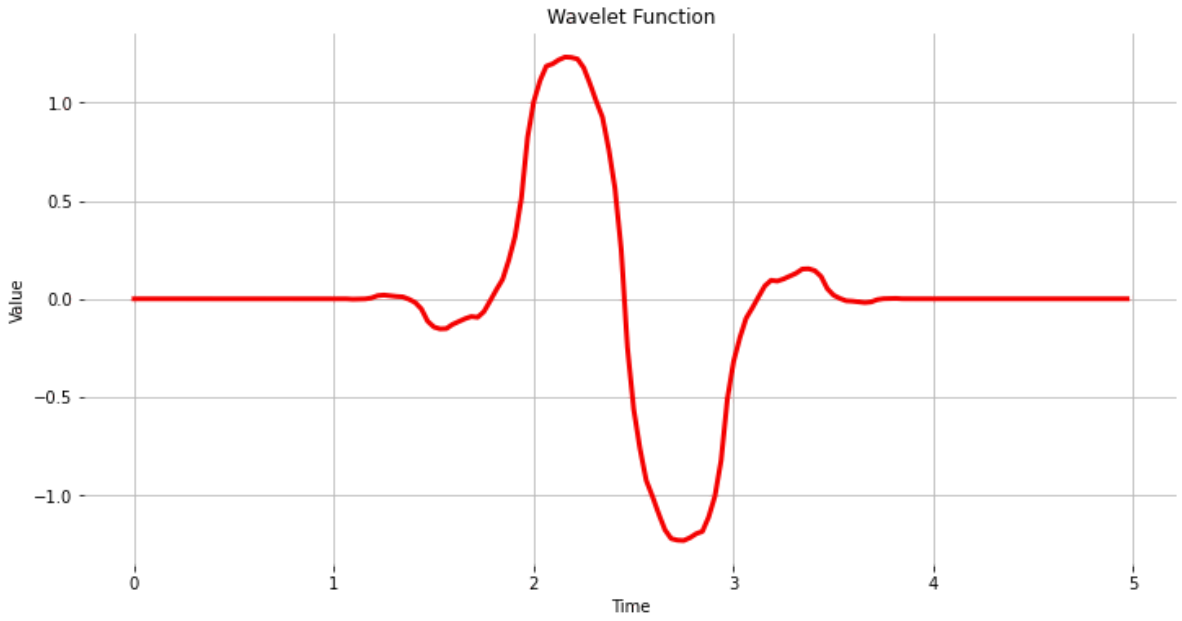
Let's select Wavelet on the Spectral Features block in the same project:

- Type: Wavelet
- Wavelet Decomposition Level: 1
- Wavelet: bior1.3



The Wavelet Function

```
wavelet_name='bior1.3'
num_layer = 1

wavelet = pywt.Wavelet(wavelet_name)
[phi_d,psi_d,phi_r,psi_r,x] = wavelet.wavefun(level=5)
plt.plot(x, psi_d, color='red')
plt.title('Wavelet Function')
plt.ylabel('Value')
plt.xlabel('Time')
plt.grid()
plt.box(False)
plt.show()
```



As we did before, let's copy and past the Processed Features:


```
features = [3.6251, 0.0615, 0.0615, -7.3517, -2.7641, 2.8462, 5.0924, .
N_feat = len(features)
N_feat_axis = int(N_feat/n_sensors)
```

Edge Impulse computes the Discrete Wavelet Transform (DWT) for each one of the Wavelet Decomposition levels selected. After that, the features will be extracted.

In the case of **Wavelets**, the extracted features are *basic statistical values*, *crossing values*, and *entropy.* There are, in total, 14 features per layer as below:

- [11] Statiscal Features: **n5, n25, n75, n95, mean, median,** standard deviation **(std)**, variance **(var)** root mean square **(rms), kurtosis**, and

skewness **(skew)**.
- [2] Crossing Features: Zero crossing rate **(zcross)** and mean crossing rate **(mcross)** are the times that the signal passes through the baseline (y = 0) and the average level (y = u) per unit of time, respectively
- [1] Complexity Feature: **Entropy** is a characteristic measure of the complexity of the signal

All the above 14 values are calculated for each Layer (including L0, the original signal)

- The total number of features varies depending on how you set the filter and the number of layers. For example, with [None] filtering and Level[1], the number of features per axis will be 14 x 2 (L0 and L1) = 28. For the three axes, we will have a total of 84 features.

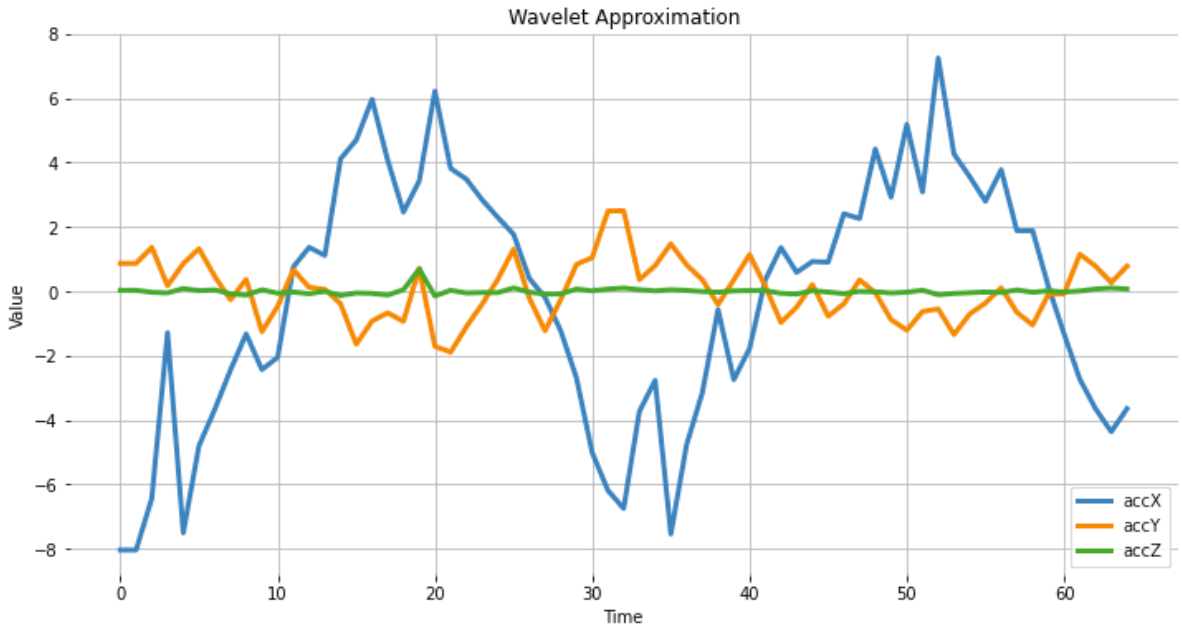
6.7.2 Wavelet Analysis

Wavelet analysis decomposes the signal (**accX, accY**, **and accZ**) into different frequency components using a set of filters, which separate these components into low-frequency (slowly varying parts of the signal containing long-term patterns), such as **accX_l1, accY_l1, accZ_l1** and, high-frequency (rapidly varying parts of the signal containing short-term patterns) components, such as **accX_d1, accY_d1, accZ_d1**, permitting the extraction of features for further analysis or classification.

Only the low-frequency components (approximation coefficients, or cA) will be used. In this example, we assume only one level (Single-level Discrete Wavelet Transform), where the function will return a tuple. With a multilevel decomposition, the “Multilevel 1D Discrete Wavelet Transform”, the result will be a list (for detail, please see: Discrete Wavelet Transform (DWT))

```
(accX_l1, accX_d1) = pywt.dwt(accX, wavelet_name)
(accY_l1, accY_d1) = pywt.dwt(accY, wavelet_name)
(accZ_l1, accZ_d1) = pywt.dwt(accZ, wavelet_name)
sensors_l1 = [accX_l1, accY_l1, accZ_l1]

# Plot power spectrum versus frequency
plt.plot(accX_l1, label='accX')
plt.plot(accY_l1, label='accY')
plt.plot(accZ_l1, label='accZ')
```

```
plt.legend(loc='lower right')
plt.xlabel('Time')
plt.ylabel('Value')
plt.title('Wavelet Approximation')
plt.grid()
plt.box(False)
plt.show()
```



6.7.3 Feature Extraction

Let's start with the basic statistical features. Note that we apply the function for both the original signals and the resultant cAs from the DWT:

```
def calculate_statistics(signal):
    n5 = np.percentile(signal, 5)
    n25 = np.percentile(signal, 25)
    n75 = np.percentile(signal, 75)
    n95 = np.percentile(signal, 95)
    median = np.percentile(signal, 50)
    mean = np.mean(signal)
    std = np.std(signal)
    var = np.var(signal)
    rms = np.sqrt(np.mean(np.square(signal)))
    return [n5, n25, n75, n95, median, mean, std, var, rms]
```

```
stat_feat_l0 = [calculate_statistics(x) for x in sensors]
stat_feat_l1 = [calculate_statistics(x) for x in sensors_l1]
```

The Skelness and Kurtosis:

```
skew_l0 = [skew(x, bias=False) for x in sensors]
skew_l1 = [skew(x, bias=False) for x in sensors_l1]
kurtosis_l0 = [kurtosis(x, bias=False) for x in sensors]
kurtosis_l1 = [kurtosis(x, bias=False) for x in sensors_l1]
```

Zero crossing (zcross) is the number of times the wavelet coefficient crosses the zero axis. It can be used to measure the signal's frequency content since high-frequency signals tend to have more zero crossings than low-frequency signals.

Mean crossing (mcross), on the other hand, is the number of times the wavelet coefficient crosses the mean of the signal. It can be used to measure the amplitude since high-amplitude signals tend to have more mean crossings than low-amplitude signals.

```
def getZeroCrossingRate(arr):
    my_array = np.array(arr)
    zcross = float("{0:.2f}".format((((my_array[:-1] * \
                                       my_array[1:]) < 0)\
                                       .sum())/len(arr)))
    return zcross

def getMeanCrossingRate(arr):
    mcross = getZeroCrossingRate(np.array(arr) - np.mean(arr))
    return mcross

def calculate_crossings(list):
    zcross=[]
    mcross=[]
    for i in range(len(list)):
        zcross_i = getZeroCrossingRate(list[i])
        zcross.append(zcross_i)
        mcross_i = getMeanCrossingRate(list[i])
        mcross.append(mcross_i)
    return zcross, mcross

cross_l0 = calculate_crossings(sensors)
cross_l1 = calculate_crossings(sensors_l1)
```

In wavelet analysis, **entropy** refers to the degree of disorder or randomness in the distribution of wavelet coefficients. Here, we used Shannon entropy, which measures a signal's uncertainty or randomness. It is calculated as the negative sum of the probabilities of the different possible outcomes of the signal multiplied by their base 2 logarithm. In the context of wavelet analysis, Shannon entropy can be used to measure the complexity of the signal, with higher values indicating greater complexity.

```
def calculate_entropy(signal, base=None):
    value, counts = np.unique(signal, return_counts=True)
    return entropy(counts, base=base)

entropy_l0 = [calculate_entropy(x) for x in sensors]
entropy_l1 = [calculate_entropy(x) for x in sensors_l1]
```

Let's now list all the wavelet features and create a list by layers.

```
L1_features_names = ["L1-n5", "L1-n25", "L1-n75", "L1-n95", "L1-median"
                     "L1-mean", "L1-std", "L1-var", "L1-rms", "L1-skew"
                     "L1-Kurtosis", "L1-zcross", "L1-mcross", "L1-entro

L0_features_names = ["L0-n5", "L0-n25", "L0-n75", "L0-n95", "L0-median"
                     "L0-mean", "L0-std", "L0-var", "L0-rms", "L0-skew"
                     "L0-Kurtosis", "L0-zcross", "L0-mcross", "L0-entro

all_feat_l0 = []
for i in range(len(axis)):
    feat_l0 = stat_feat_l0[i]+[skew_l0[i]]+[kurtosis_l0[i]]+\
    [cross_l0[0][i]]+[cross_l0[1][i]]+[entropy_l0[i]]
    [print(axis[i]+' '+x+'= ', round(y, 4)) for x,y in \
     zip(L0_features_names, feat_l0)][0]
    all_feat_l0.append(feat_l0)
all_feat_l0 = [item for sublist in all_feat_l0 for item in sublist]
print(f"\nAll L0 Features = {len(all_feat_l0)}")

all_feat_l1 = []
for i in range(len(axis)):
feat_l1 = stat_feat_l1[i]+[skew_l1[i]]+[kurtosis_l1[i]]+[cross_l1[0][i]
          [cross_l1[1][i]]+[entropy_l1[i]]
[print(axis[i]+' '+x+'= ', round(y, 4)) for x,y in zip(L1_features_name
                                                       feat_l1)][0]
all_feat_l1.append(feat_l1)
```

```
all_feat_l1 = [item for sublist in all_feat_l1 for item in sublist]
print(f"\nAll L1 Features = {len(all_feat_l1)}")
```

```
accX L0-n5=  -4.9364
accX L0-n25=  -1.8429
accX L0-n75=  1.8842
accX L0-n95=  3.8096
accX L0-median=  0.4058
accX L0-mean=  -0.0
accX L0-std=  2.7322
accX L0-var=  7.4651
accX L0-rms=  2.7322
accX L0-skew=  -0.099
accX L0-Kurtosis=  -0.3475
accX L0-zcross=  0.06
accX L0-mcross=  0.06
accX L0-entropy=  4.8283
accY L0-n5=  -1.149
accY L0-n25=  -0.4475
accY L0-n75=  0.4814
accY L0-n95=  1.1491
accY L0-median=  -0.0315
accY L0-mean=  0.0
accY L0-std=  0.7833
accY L0-var=  0.6136
accY L0-rms=  0.7833
accY L0-skew=  0.1756
accY L0-Kurtosis=  1.2673
accY L0-zcross=  0.29
accY L0-mcross=  0.29
accY L0-entropy=  4.8283
accZ L0-n5=  -0.1242
accZ L0-n25=  -0.0429
accZ L0-n75=  0.0349
accZ L0-n95=  0.0839
accZ L0-median=  -0.0112
accZ L0-mean=  0.0
accZ L0-std=  0.1383
accZ L0-var=  0.0191
accZ L0-rms=  0.1383
accZ L0-skew=  6.9463
accZ L0-Kurtosis=  68.1123
accZ L0-zcross=  0.35
accZ L0-mcross=  0.35
accZ L0-entropy=  4.5649

All L0 Features = 42
```

```
accX L1-n5=  -7.3516
accX L1-n25=  -2.7641
accX L1-n75=  2.8462
accX L1-n95=  5.0924
accX L1-median=  0.4064
accX L1-mean=  -0.2133
accX L1-std=  3.8473
accX L1-var=  14.8015
accX L1-rms=  3.8532
accX L1-skew=  -0.2975
accX L1-Kurtosis=  -0.7631
accX L1-zcross=  0.06
accX L1-mcross=  0.06
accX L1-entropy=  4.1744
accY L1-n5=  -1.3234
accY L1-n25=  -0.6492
accY L1-n75=  0.7844
accY L1-n95=  1.361
accY L1-median=  0.0659
accY L1-mean=  0.0276
accY L1-std=  0.9345
accY L1-var=  0.8732
accY L1-rms=  0.9349
accY L1-skew=  0.2874
accY L1-Kurtosis=  0.0347
accY L1-zcross=  0.31
accY L1-mcross=  0.31
accY L1-entropy=  4.1317
accZ L1-n5=  -0.1126
accZ L1-n25=  -0.0493
accZ L1-n75=  0.0348
accZ L1-n95=  0.1022
accZ L1-median=  -0.0137
accZ L1-mean=  0.0025
accZ L1-std=  0.1053
accZ L1-var=  0.0111
accZ L1-rms=  0.1053
accZ L1-skew=  4.4095
accZ L1-Kurtosis=  28.6586
accZ L1-zcross=  0.4
accZ L1-mcross=  0.37
accZ L1-entropy=  4.1531

All L1 Features = 42
```

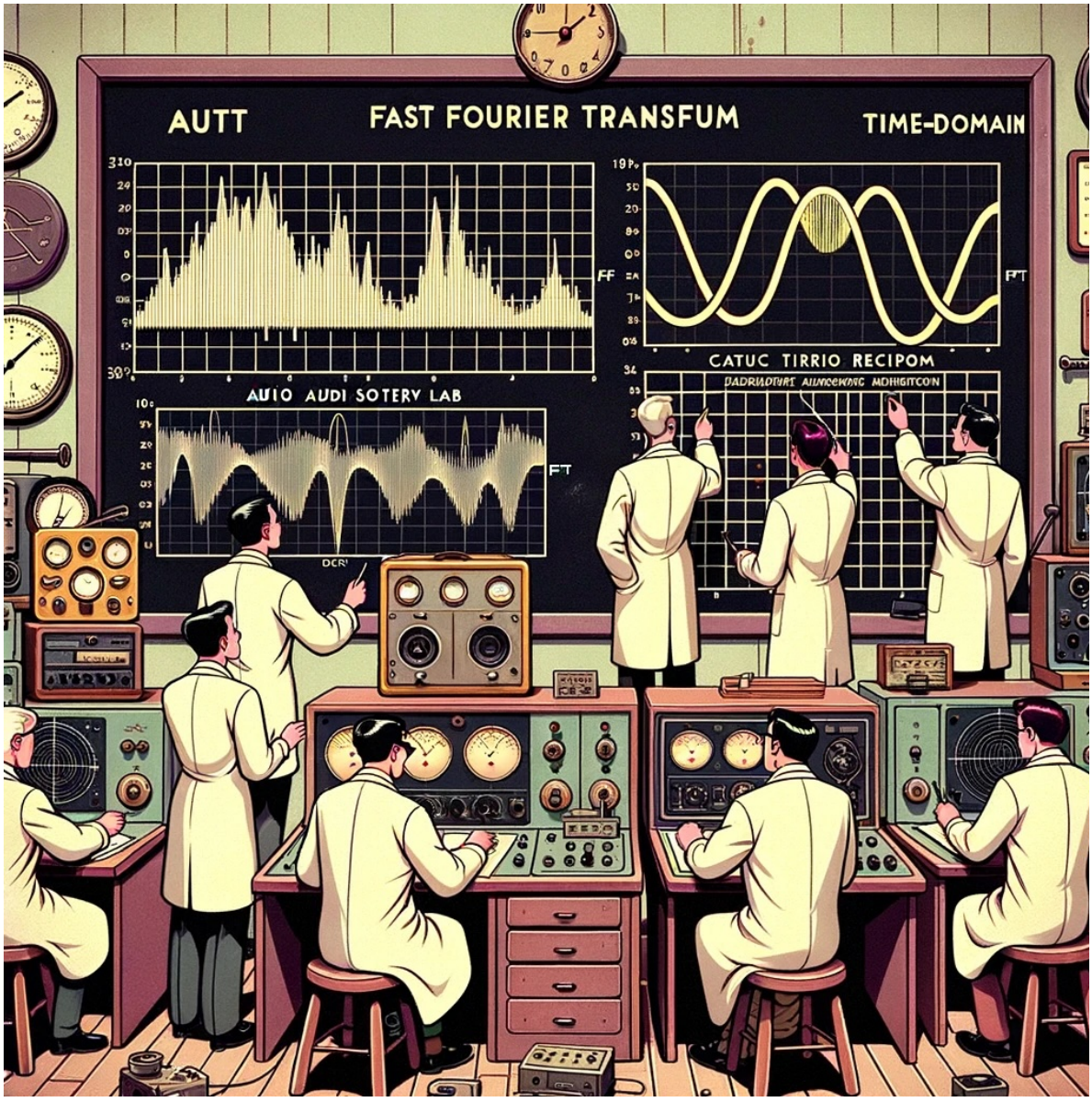
6.8 Conclusion

Edge Impulse Studio is a powerful online platform that can handle the pre-processing task for us. Still, given our engineering perspective, we want to understand what is happening under the hood. This knowledge will help us find the best options and hyper-parameters for tuning our projects.

Daniel Situnayake wrote in his blog: “Raw sensor data is highly dimensional and noisy. Digital signal processing algorithms help us sift the signal from the noise. DSP is an essential part of embedded engineering, and many edge

processors have on-board acceleration for DSP. As an ML engineer, learning basic DSP gives you superpowers for handling high-frequency time series data in your models." I recommend you read Dan's excellent post in its totality: nn to cpp: What you need to know about porting deep learning models to the edge.

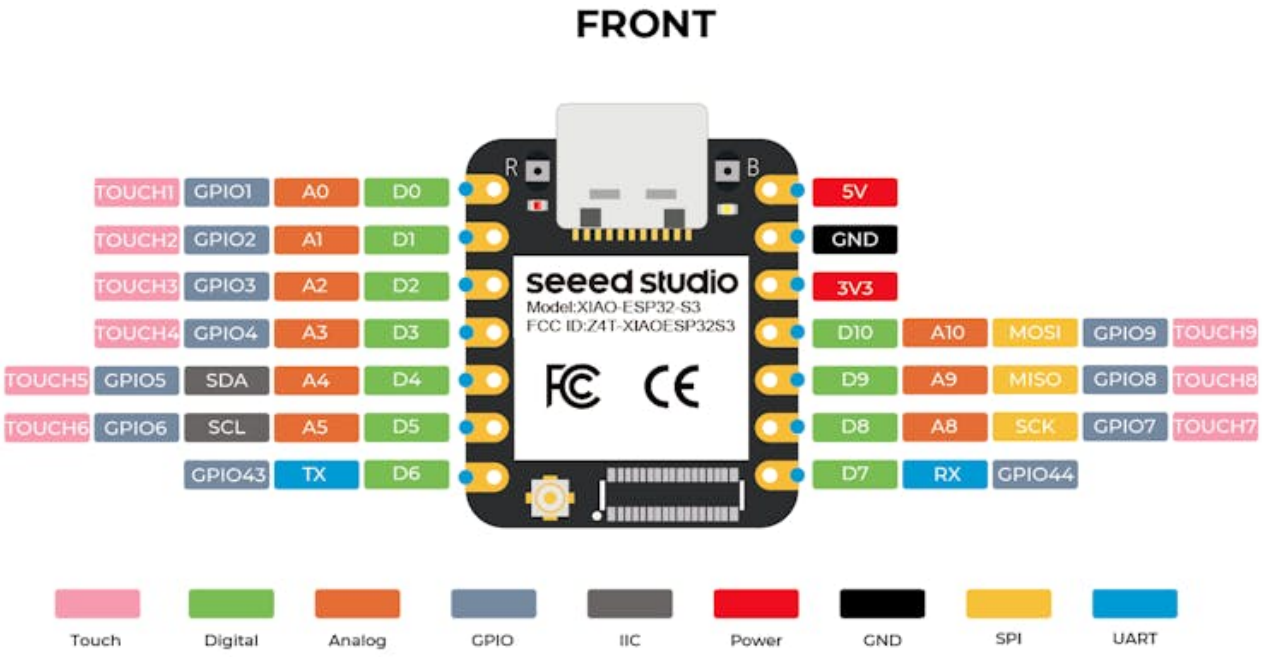
7 Motion Classification and Anomaly Detection



DALL·E prompt - 1950s style cartoon illustration set in a vintage audio lab. Scientists, dressed in classic attire with white lab coats, are intently analyzing audio data on large chalkboards. The boards display intricate FFT (Fast Fourier Transform) graphs and time-domain curves. Antique audio equipment is scattered around, but the data representatio are clear and detailed, indicating their focus on audio analysis.

7.1 Introduction

The XIAO ESP32S3 Sense, with its built-in camera and mic, is a versatile device. But what if you need to add another type of sensor, such as an IMU? No problem! One of the standout features of the XIAO ESP32S3 is its multiple pins that can be used as an I2C bus (SDA/SCL pins), making it a suitable platform for sensor integration.



7.2 Installing the IMU

When selecting your IMU, the market offers a wide range of devices, each with unique features and capabilities. You could choose, for example, the ADXL362 (3-axis), MAX21100 (6-axis), MPU6050 (6-axis), LIS3DHTR (3-axis), or the LCM20600Seeed Grove— (6-axis), which is part of the IMU 9DOF (lcm20600+AK09918). This variety allows you to tailor your choice to your project's specific needs.

For this project, we will use an IMU, the MPU6050 (or 6500), a low-cost (less than 2.00 USD) 6-axis Accelerometer/Gyroscope unit.

At the end of the lab, we will also comment on using the LCM20600.

The MPU-6500 is a 6-axis Motion Tracking device that combines a 3-axis gyroscope, 3-axis accelerometer, and a Digital Motion ProcessorTM (DMP) in a small 3x3x0.9mm package. It also features a 4096-byte FIFO that can lower the traffic on the serial bus interface and reduce power consumption by allowing the system processor to burst read sensor data and then go into a low-power mode.

With its dedicated I2C sensor bus, the MPU-6500 directly accepts inputs from external I2C devices. MPU-6500, with its 6-axis integration, on-chip DMP, and run-time calibration firmware, enables manufacturers to eliminate the costly and complex selection, qualification, and system-level integration of discrete devices, guaranteeing optimal motion performance for consumers. MPU-6500 is also designed to interface with multiple non-inertial digital sensors, such as pressure sensors, on its auxiliary I2C port.

MPU6050

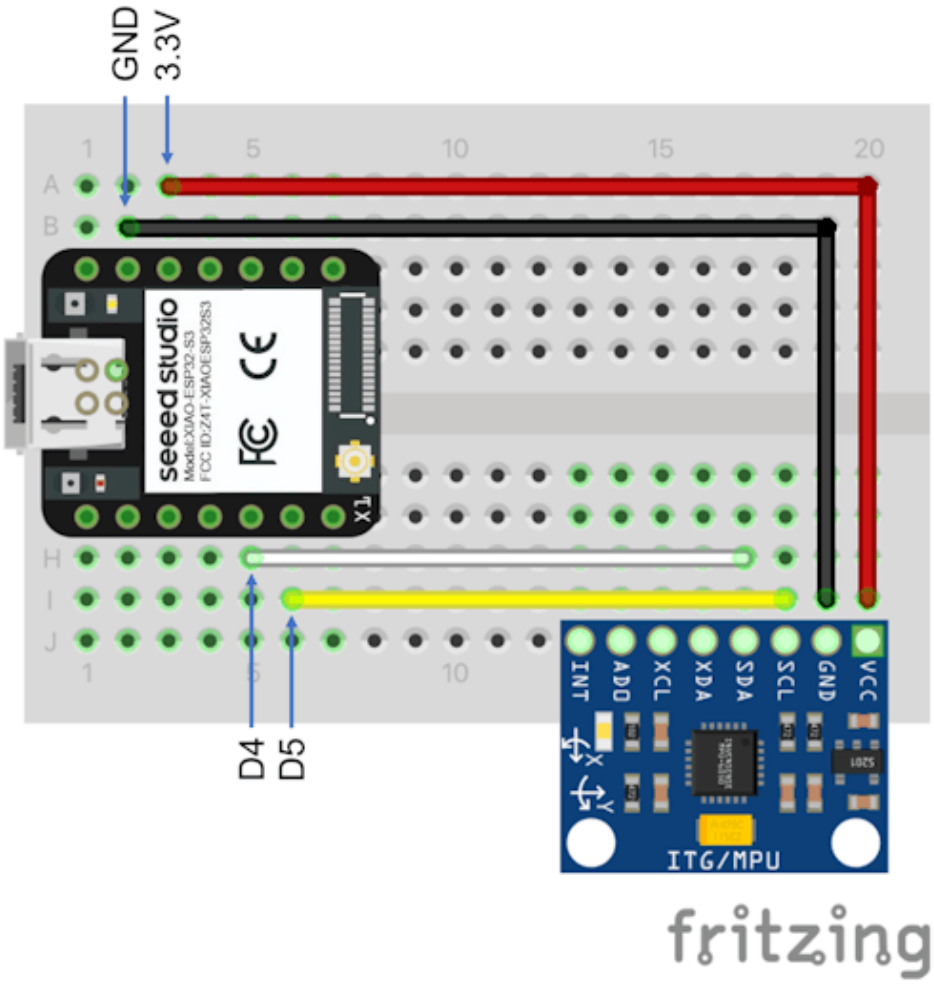
MPU6500

Usually, the libraries available are for MPU6050, but they work for both devices.

Connecting the HW

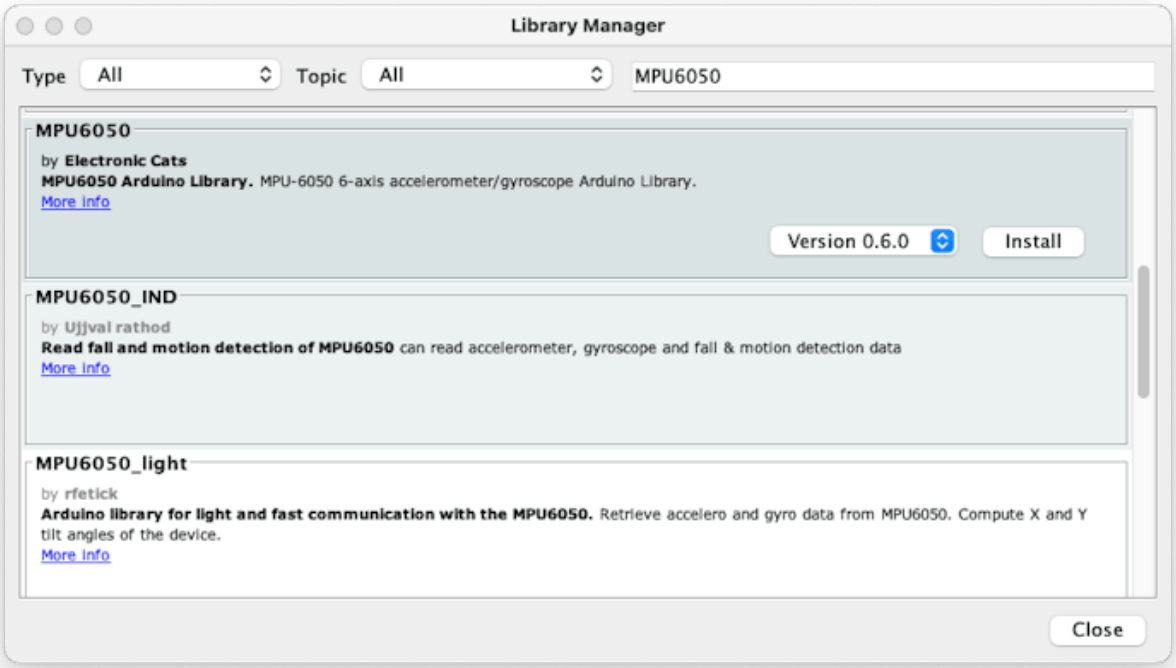
Connect the IMU to the XIAO according to the below diagram:

- MPU6050 **SCL** –> XIAO **D5**
- MPU6050 **SDA** –> XIAO **D4**
- MPU6050 **VCC** –> XIAO **3.3V**
- MPU6050 **GND** –> XIAO **GND**



Install the Library

Go to Arduino Library Manager and type MPU6050. Install the latest version.



Download the sketch MPU6050_Acc_Data_Acquisition.in:

```
/*
 * Based on I2C device class (I2Cdev) Arduino sketch for MPU6050 class
   by Jeff Rowberg <jeff@rowberg.net>
 * and Edge Impulse Data Forwarder Exampe (Arduino)
   - https://docs.edgeimpulse.com/docs/cli-data-forwarder
 *
 * Developed by M.Rovai @11May23
 */

#include "I2Cdev.h"
#include "MPU6050.h"
#include "Wire.h"

#define FREQUENCY_HZ        50
#define INTERVAL_MS         (1000 / (FREQUENCY_HZ + 1))
#define ACC_RANGE           1 // 0: -/+2G; 1: +/-4G

// convert factor g to m/s2 ==> [-32768, +32767] ==> [-2g, +2g]
#define CONVERT_G_TO_MS2    (9.81/(16384.0/(1.+ACC_RANGE)))

static unsigned long last_interval_ms = 0;

MPU6050 imu;
```

```
int16_t ax, ay, az;

void setup() {

    Serial.begin(115200);


    // initialize device
    Serial.println("Initializing I2C devices...");
    Wire.begin();
    imu.initialize();
    delay(10);

//    // verify connection
//    if (imu.testConnection()) {
//      Serial.println("IMU connected");
//    }
//    else {
//      Serial.println("IMU Error");
//    }
    delay(300);

    //Set MCU 6050 OffSet Calibration
    imu.setXAccelOffset(-4732);
    imu.setYAccelOffset(4703);
    imu.setZAccelOffset(8867);
    imu.setXGyroOffset(61);
    imu.setYGyroOffset(-73);
    imu.setZGyroOffset(35);

    /* Set full-scale accelerometer range.
     * 0 = +/- 2g
     * 1 = +/- 4g
     * 2 = +/- 8g
     * 3 = +/- 16g
     */
    imu.setFullScaleAccelRange(ACC_RANGE);
}

void loop() {

      if (millis() > last_interval_ms + INTERVAL_MS) {
        last_interval_ms = millis();

        // read raw accel/gyro measurements from device
        imu.getAcceleration(&ax, &ay, &az);
```

```
        // converting to m/s2
        float ax_m_s2 = ax * CONVERT_G_TO_MS2;
        float ay_m_s2 = ay * CONVERT_G_TO_MS2;
        float az_m_s2 = az * CONVERT_G_TO_MS2;

        Serial.print(ax_m_s2);
        Serial.print("\t");
        Serial.print(ay_m_s2);
        Serial.print("\t");
        Serial.println(az_m_s2);
      }
}
```

Some comments about the code:

Note that the values generated by the accelerometer and gyroscope have a range: [-32768, +32767], so for example, if the default accelerometer range is used, the range in Gs should be: [-2g, +2g]. So, “1G” means 16384.

For conversion to m/s2, for example, you can define the following:

```
#define CONVERT_G_TO_MS2 (9.81/16384.0)
```

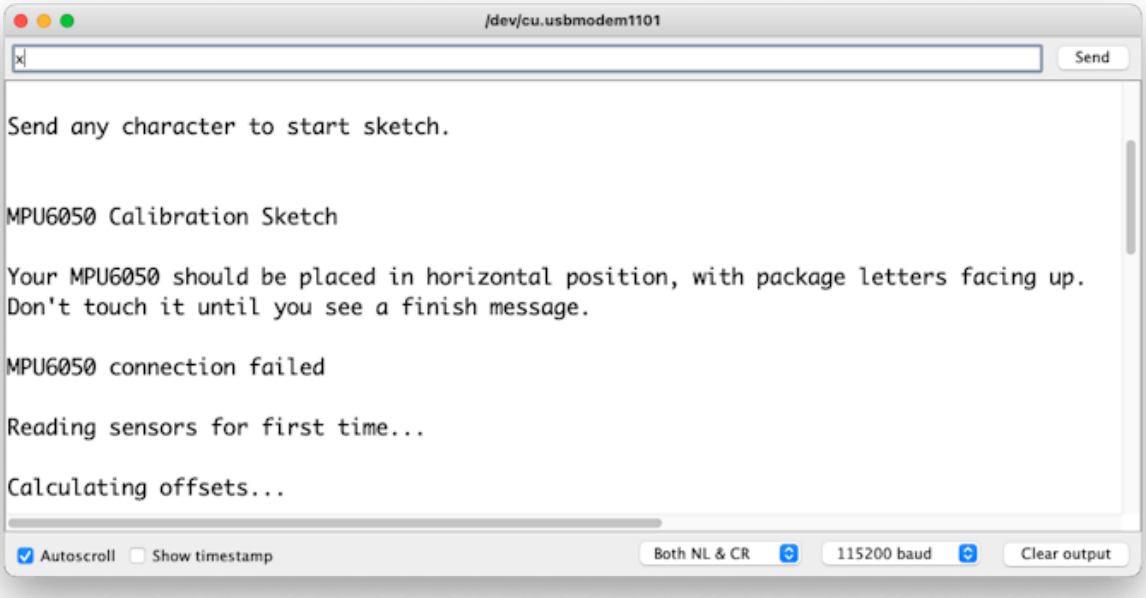
In the code, I left an option (ACC_RANGE) to be set to 0 (+/-2G) or 1 (+/-4G). We will use +/-4G; that should be enough for us. In this case.

We will capture the accelerometer data on a frequency of 50Hz, and the acceleration data will be sent to the Serial Port as meters per squared second (m/s2).

When you ran the code with the IMU resting over your table, the accelerometer data shown on the Serial Monitor should be around 0.00, 0.00, and 9.81. If the values are a lot different, you should calibrate the IMU.

The MCU6050 can be calibrated using the sketch: mcu6050-calibration.ino.

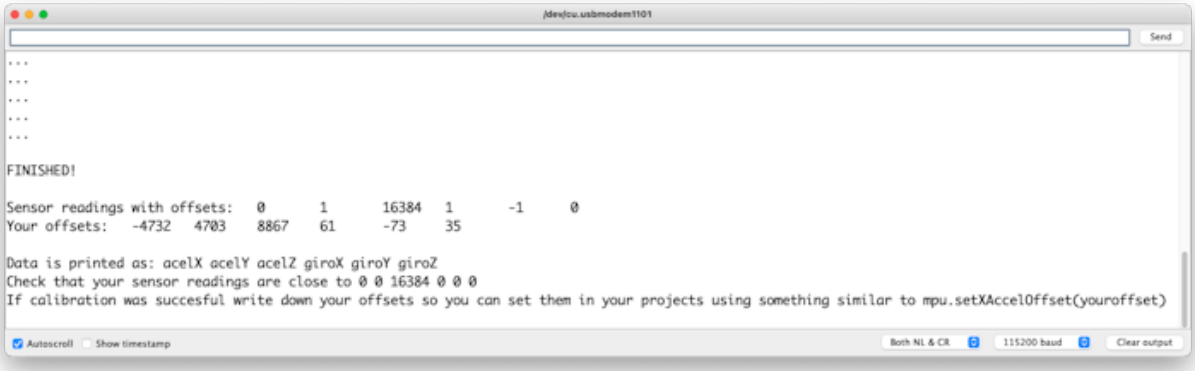
Run the code. The following will be displayed on the Serial Monitor:



Send any character (in the above example, “x”), and the calibration should start.

> Note that A message MPU6050 connection failed. Ignore this message. For some reason, imu.testConnection() is not returning a correct result.

In the end, you will receive the offset values to be used on all your sketches:



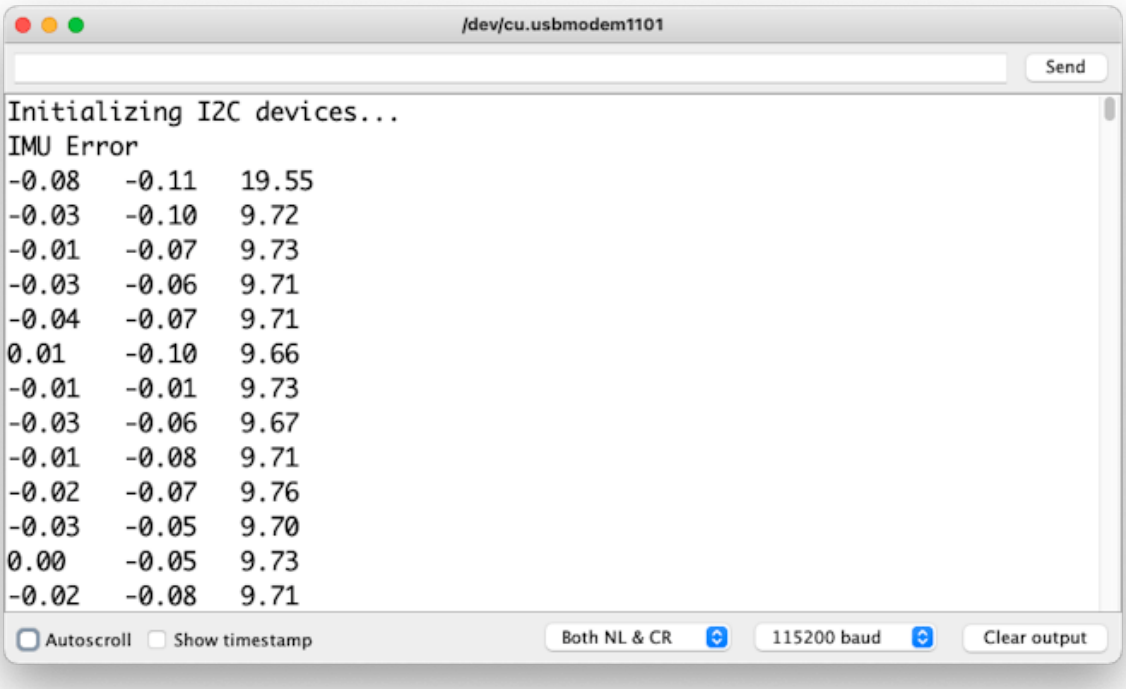
Take the values and use them on the setup:

```
//Set MCU 6050 OffSet Calibration
imu.setXAccelOffset(-4732);
```

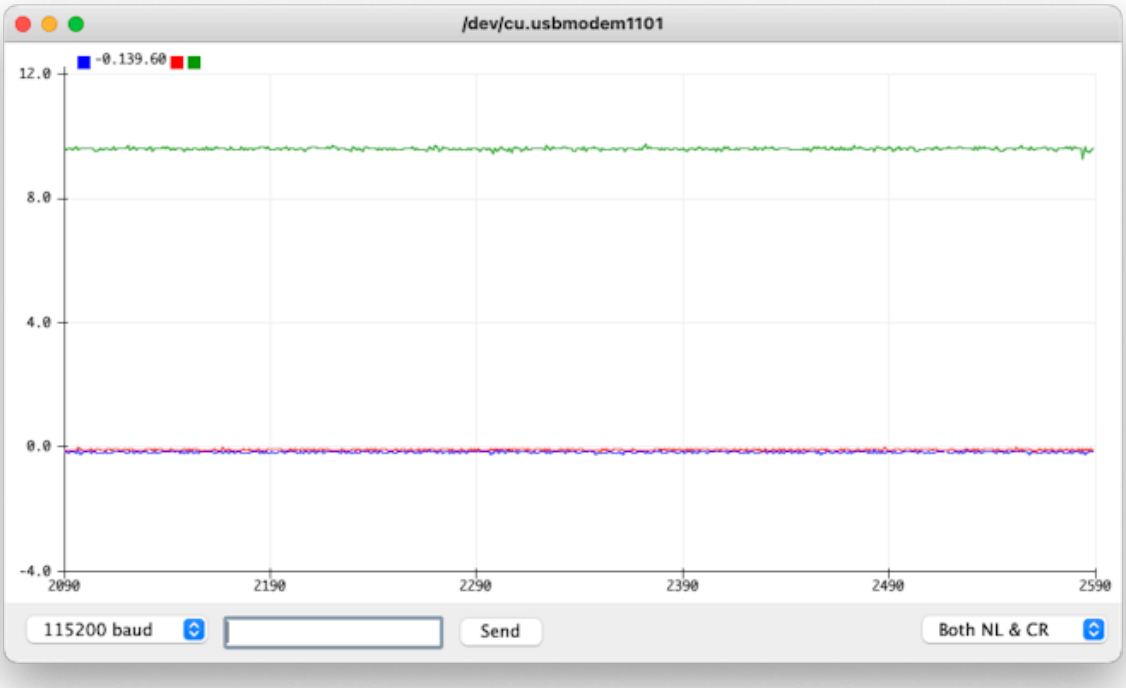
```
imu.setYAccelOffset(4703);
imu.setZAccelOffset(8867);
imu.setXGyroOffset(61);
imu.setYGyroOffset(-73);
imu.setZGyroOffset(35);
```

Now, run the sketch MPU6050_Acc_Data_Acquisition.in:

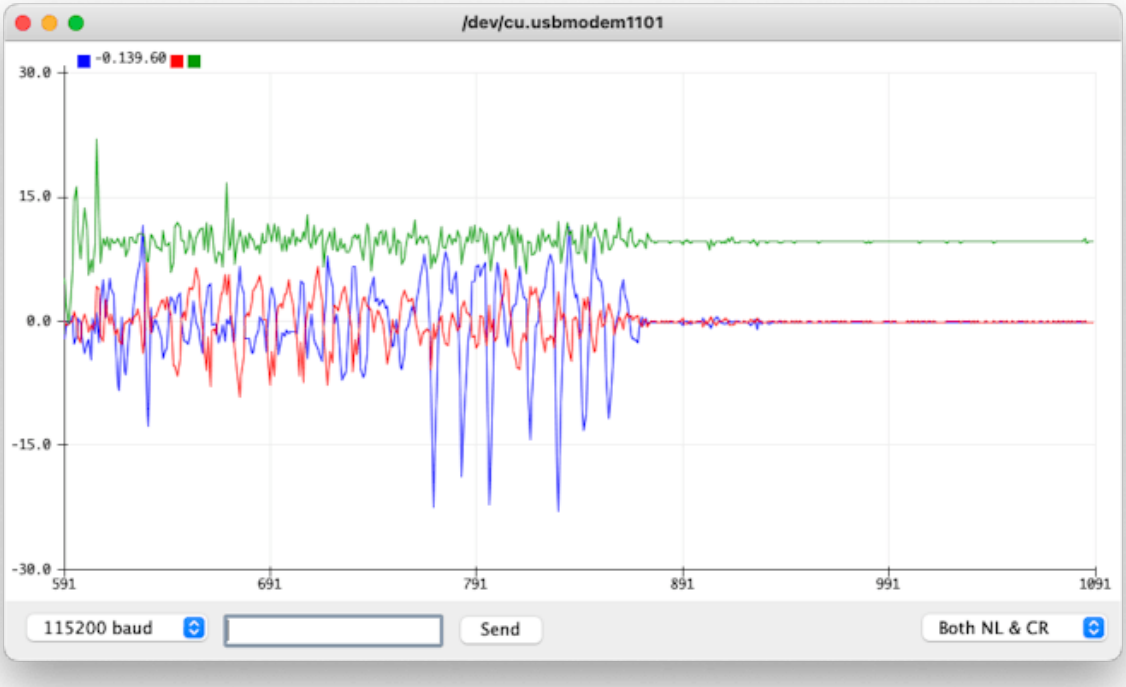
Once you run the above sketch, open the Serial Monitor:



Or check the Plotter:



Move your device in the three axes. You should see the variation on Plotter:

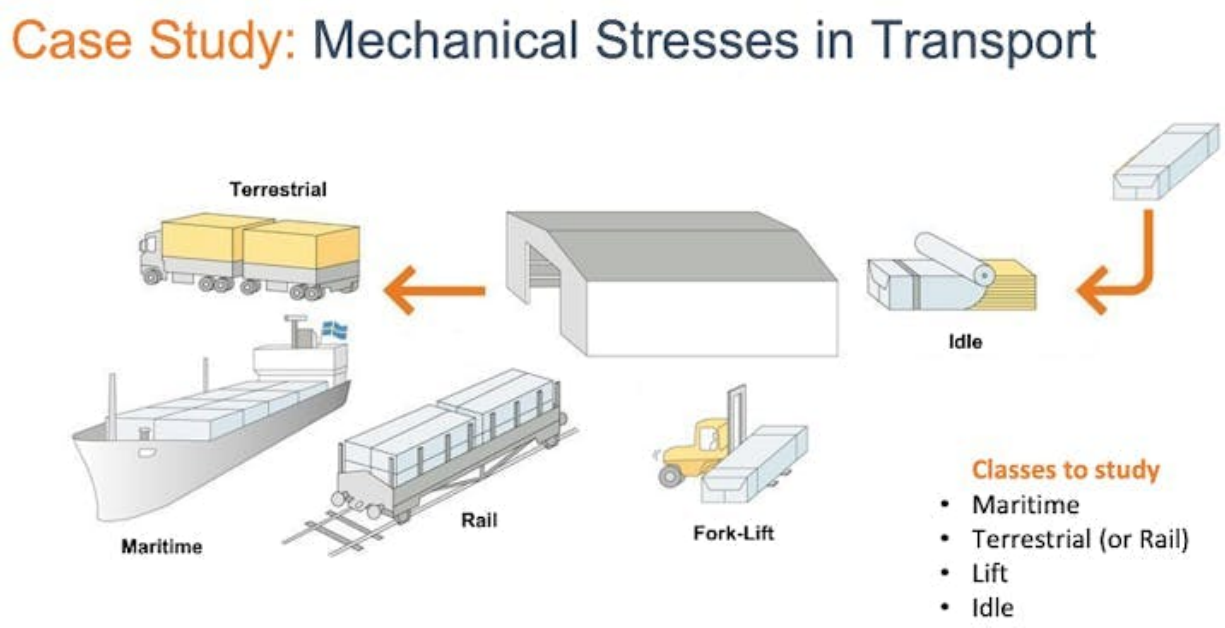


7.3 The TinyML Motion Classification Project

For our lab, we will simulate mechanical stresses in transport. Our problem will be to classify four classes of movement:

- **Maritime** (pallets in boats)
- **Terrestrial** (palettes in a Truck or Train)
- **Lift** (Palettes being handled by Fork-Lift)
- **Idle** (Palettes in Storage houses)

So, to start, we should collect data. Then, accelerometers will provide the data on the palette (or container).



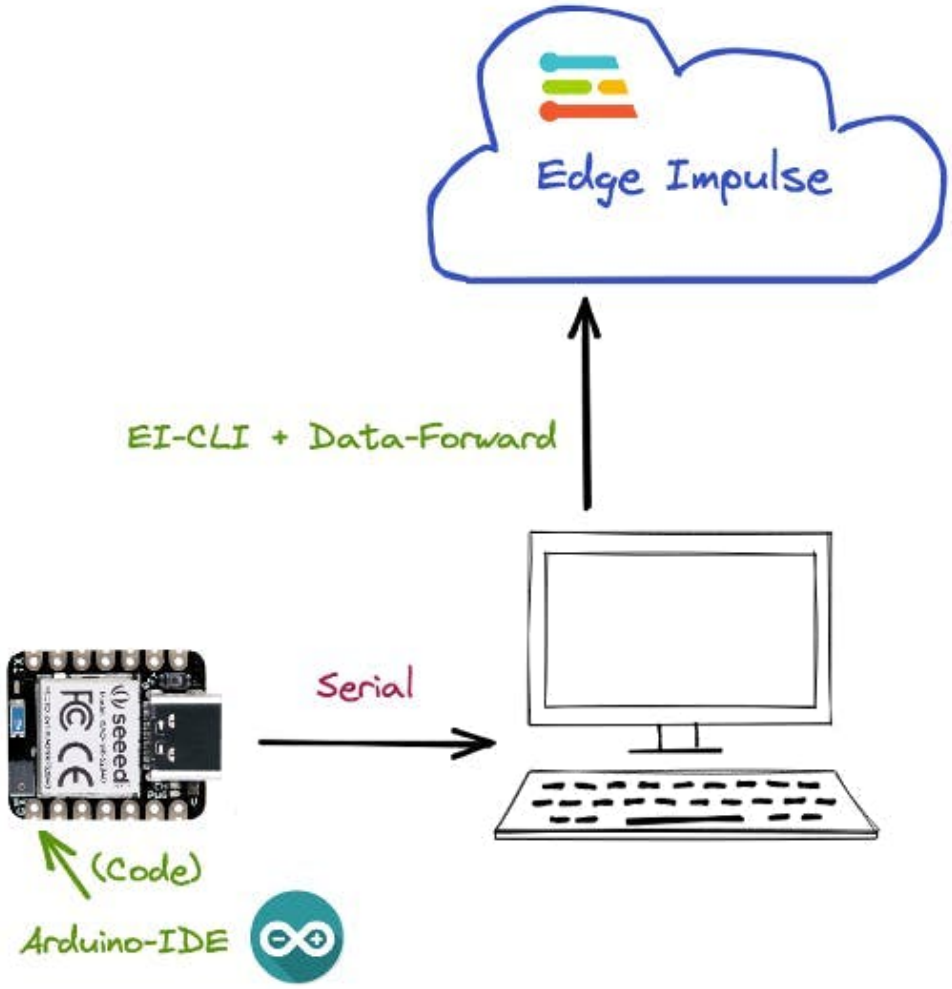
From the above images, we can see that primarily horizontal movements should be associated with the “Terrestrial class,” Vertical movements with the “Lift Class,” no activity with the “Idle class,” and movement on all three axes to Maritime class.

7.4 Connecting the device to Edge Impulse

For data collection, we should first connect our device to the Edge Impulse Studio, which will also be used for data pre-processing, model training, testing, and deployment.

> Follow the instructions hereto install the Node.jsand Edge Impulse CLI on your computer.

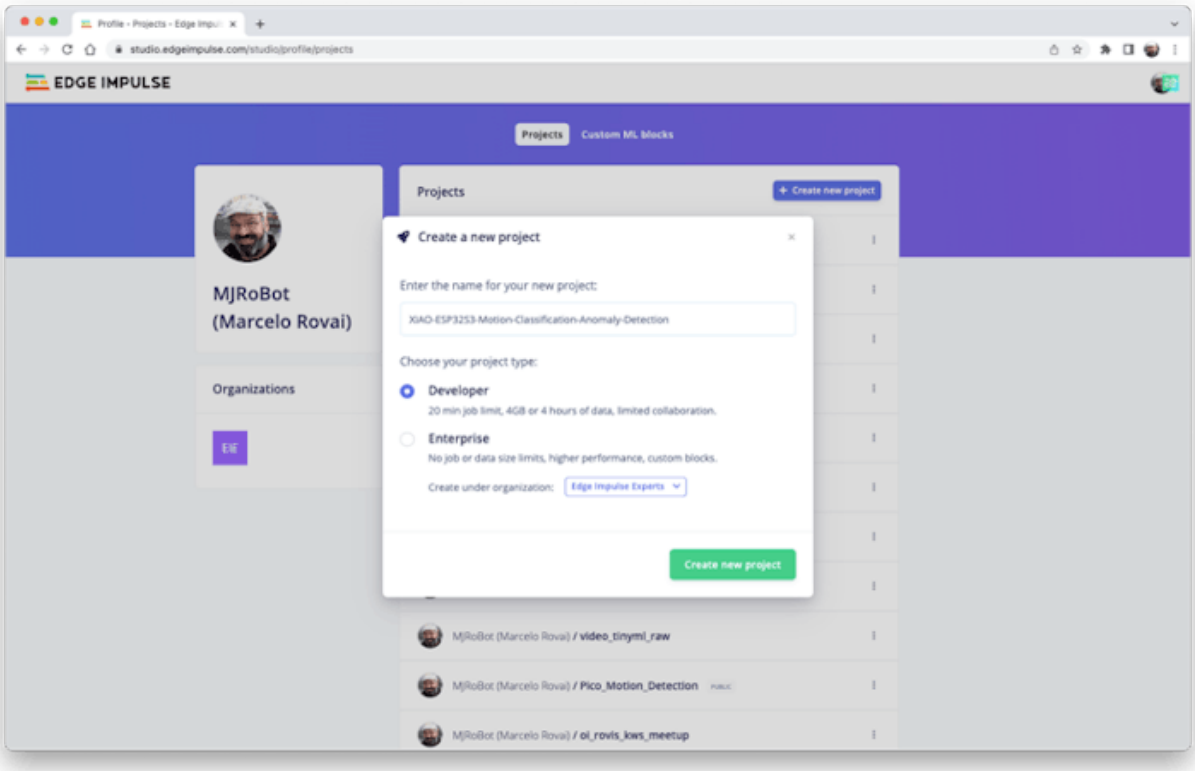
Once the XIAO ESP32S3 is not a fully supported development board by Edge Impulse, we should, for example, use the CLI Data Forwarder to capture data from our sensor and send it to the Studio, as shown in this diagram:



You can alternately capture your data “offline,” store them on an SD card or send them to your computer via Bluetooth or Wi-Fi. In this video, you can learn alternative ways to send data to the Edge Impulse Studio.

Connect your device to the serial port and run the previous code to capture IMU (Accelerometer) data, “printing them” on the serial. This will allow the Edge Impulse Studio to “capture” them.

Go to the Edge Impulse page and create a project.



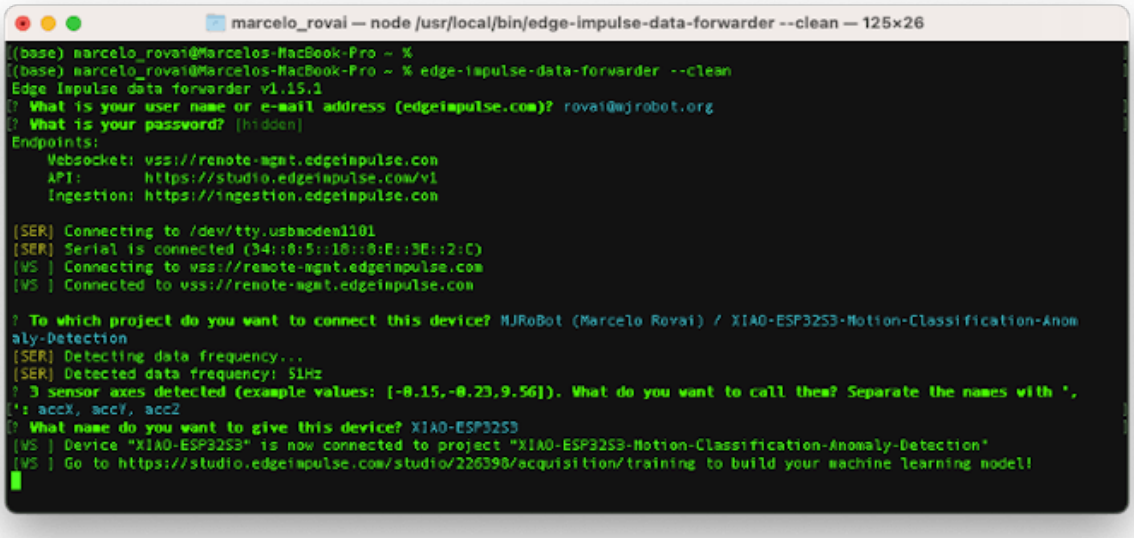
The maximum length for an Arduino library name is **63 characters**. Note that the Studio will name the final library using your project name and include "_inference" to it. The name I chose initially did not work when I tried to deploy the Arduino library because it resulted in 64 characters. So, I need to change it by taking out the "anomaly detection" part.

Start the CLI Data Forwarderon your terminal, entering (if it is the first time) the following command:

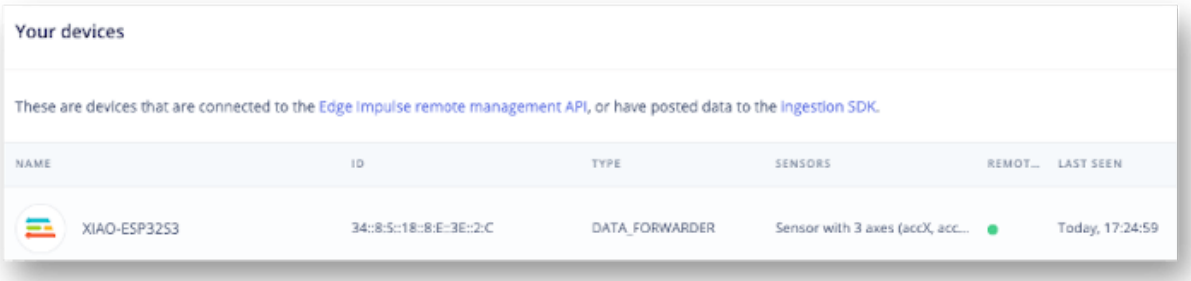
$ edge-impulse-data-forwarder –clean

```
$ edge-impulse-data-forwarder --clean
```

Next, enter your EI credentials and choose your project, variables, and device names:

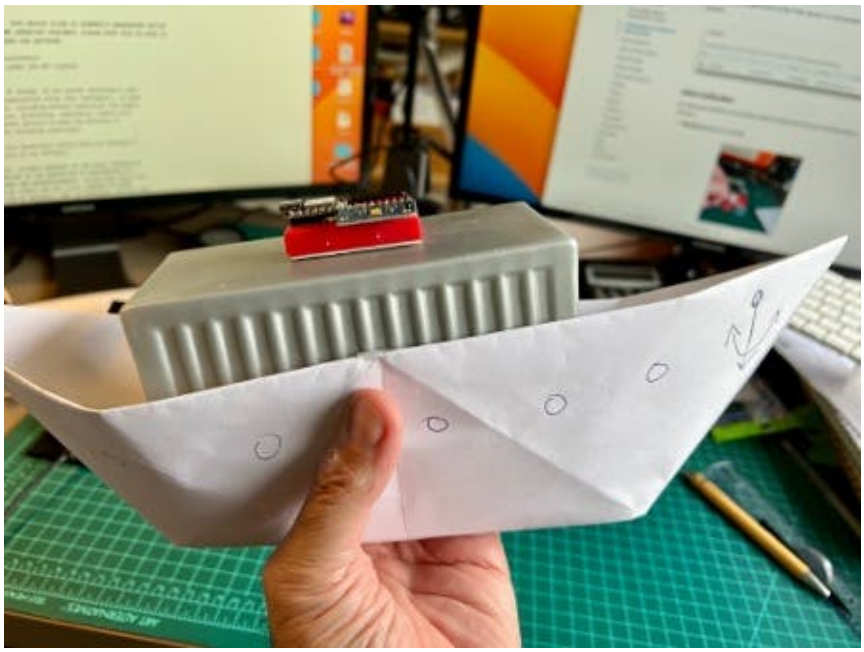


Go to your EI Project and verify if the device is connected (the dot should be green):



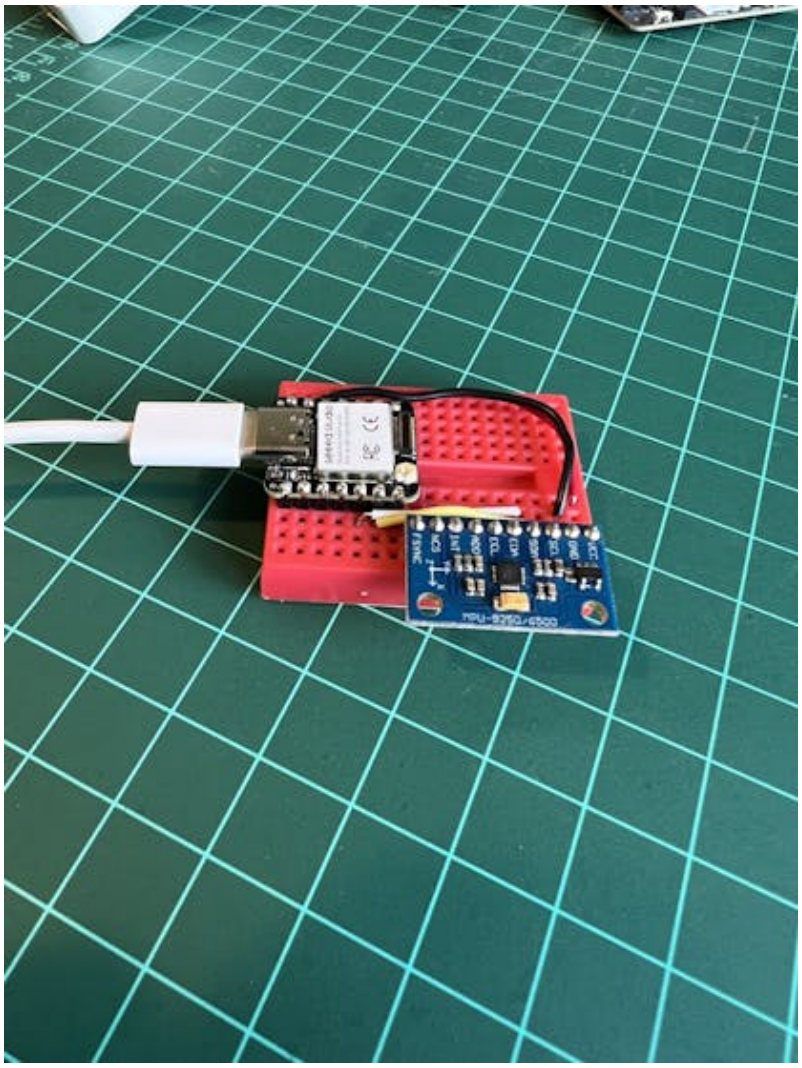
7.5 Data Collection

As discussed before, we should capture data from all four Transportation Classes. Imagine that you have a container with a built-in accelerometer:

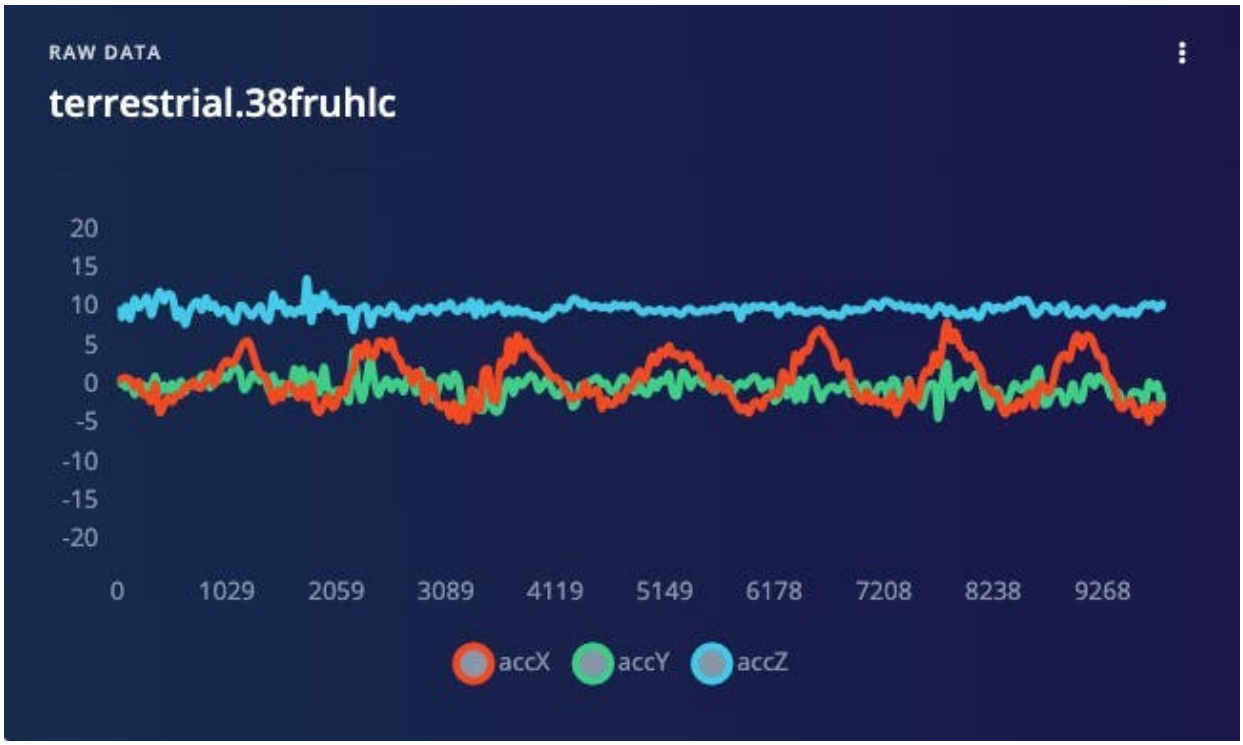


Now imagine your container is on a boat, facing an angry ocean, on a truck, etc.:

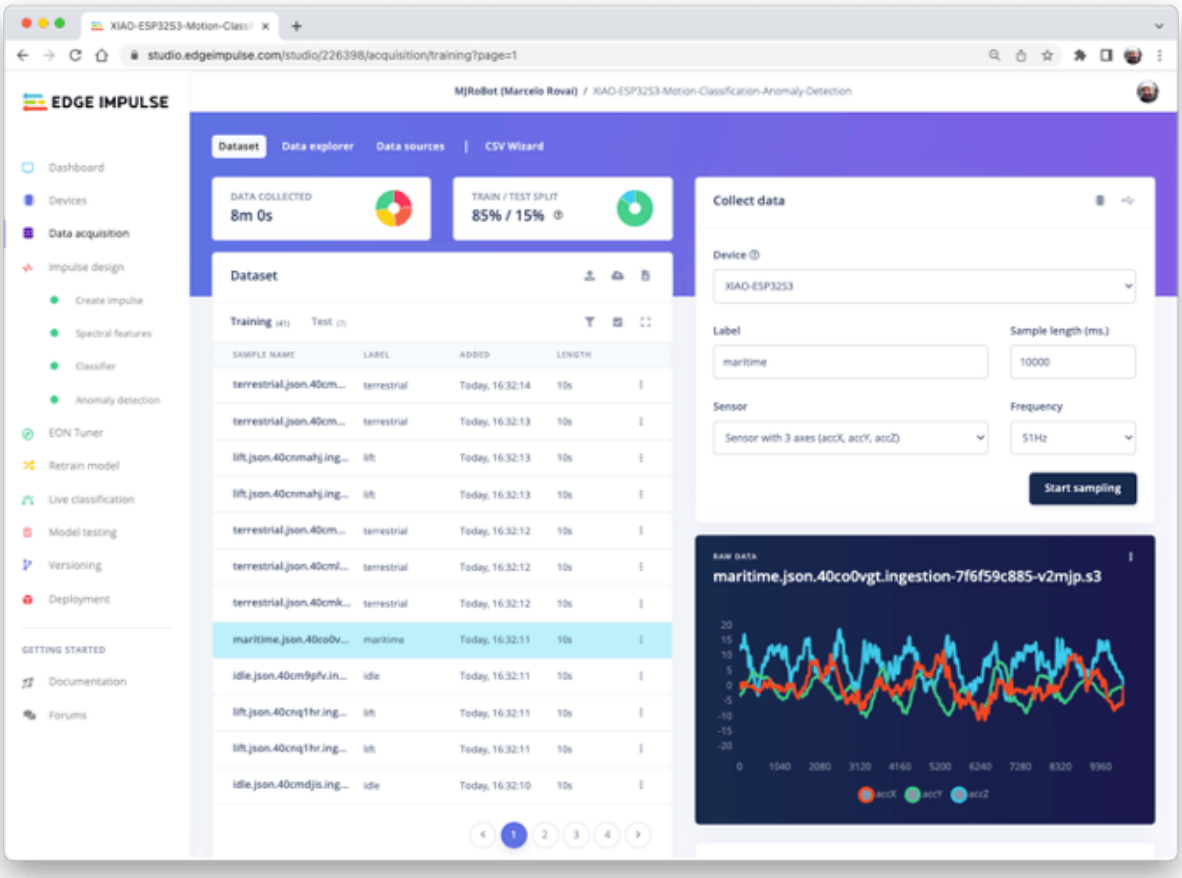
- **Maritime** (pallets in boats)
  - Move the XIAO in all directions, simulating an undulatory boat movement.
- **Terrestrial** (palettes in a Truck or Train)
  - Move the XIAO over a horizontal line.
- **Lift** (Palettes being handled by
  - Move the XIAO over a vertical line.
- **Idle** (Palettes in Storage houses)
  - Leave the XIAO over the table.



Below is one sample (raw data) of 10 seconds:

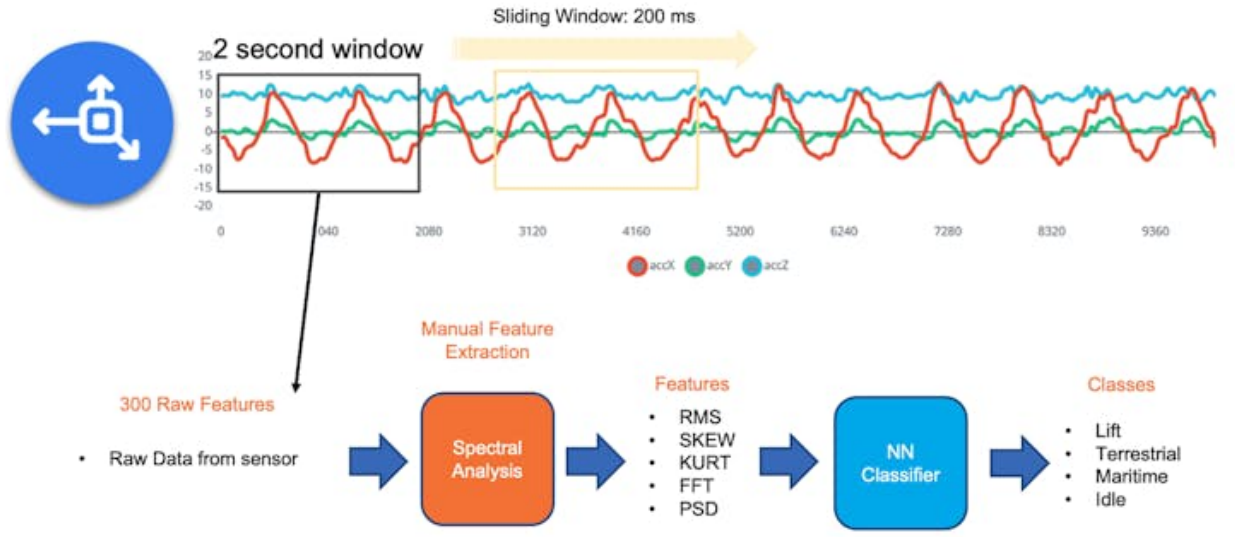


You can capture, for example, 2 minutes (twelve samples of 10 seconds each) for the four classes. Using the “3 dots” after each one of the samples, select 2, moving them for the Test set (or use the automatic Train/Test Split tool on the Danger Zone of Dashboard tab). Below, you can see the result datasets:



7.6 Data Pre-Processing

The raw data type captured by the accelerometer is a “time series” and should be converted to “tabular data”. We can do this conversion using a sliding window over the sample data. For example, in the below figure,



We can see 10 seconds of accelerometer data captured with a sample rate (SR) of 50Hz. A 2-second window will capture 300 data points (3 axis x 2 seconds x 50 samples). We will slide this window each 200ms, creating a larger dataset where each instance has 300 raw features.

> You should use the best SR for your case, considering Nyquist's theorem, which states that a periodic signal must be sampled at more than twice the signal's highest frequency component.

Data preprocessing is a challenging area for embedded machine learning. Still, Edge Impulse helps overcome this with its digital signal processing (DSP) preprocessing step and, more specifically, the Spectral Features.

On the Studio, this dataset will be the input of a Spectral Analysis block, which is excellent for analyzing repetitive motion, such as data from accelerometers. This block will perform a DSP (Digital Signal Processing), extracting features such as "FFT" or "Wavelets". In the most common case, FFT, the **Time Domain Statistical features** per axis/channel are:

- RMS
- Skewness
- Kurtosis

And the **Frequency Domain Spectral features** per axis/channel are:

- Spectral Power
- Skewness
- Kurtosis

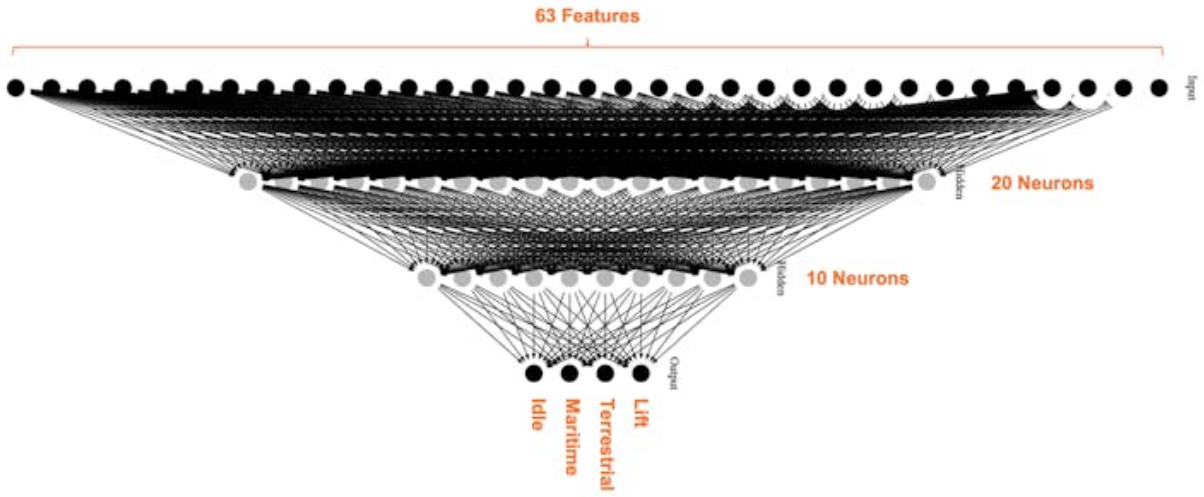
For example, for an FFT length of 32 points, the Spectral Analysis Block's resulting output will be 21 features per axis (a total of 63 features).

Those 63 features will be the Input Tensor of a Neural Network Classifier and the Anomaly Detection model (K-Means).

You can learn more by digging into the lab *DSP - Spectral Features*

7.7 Model Design

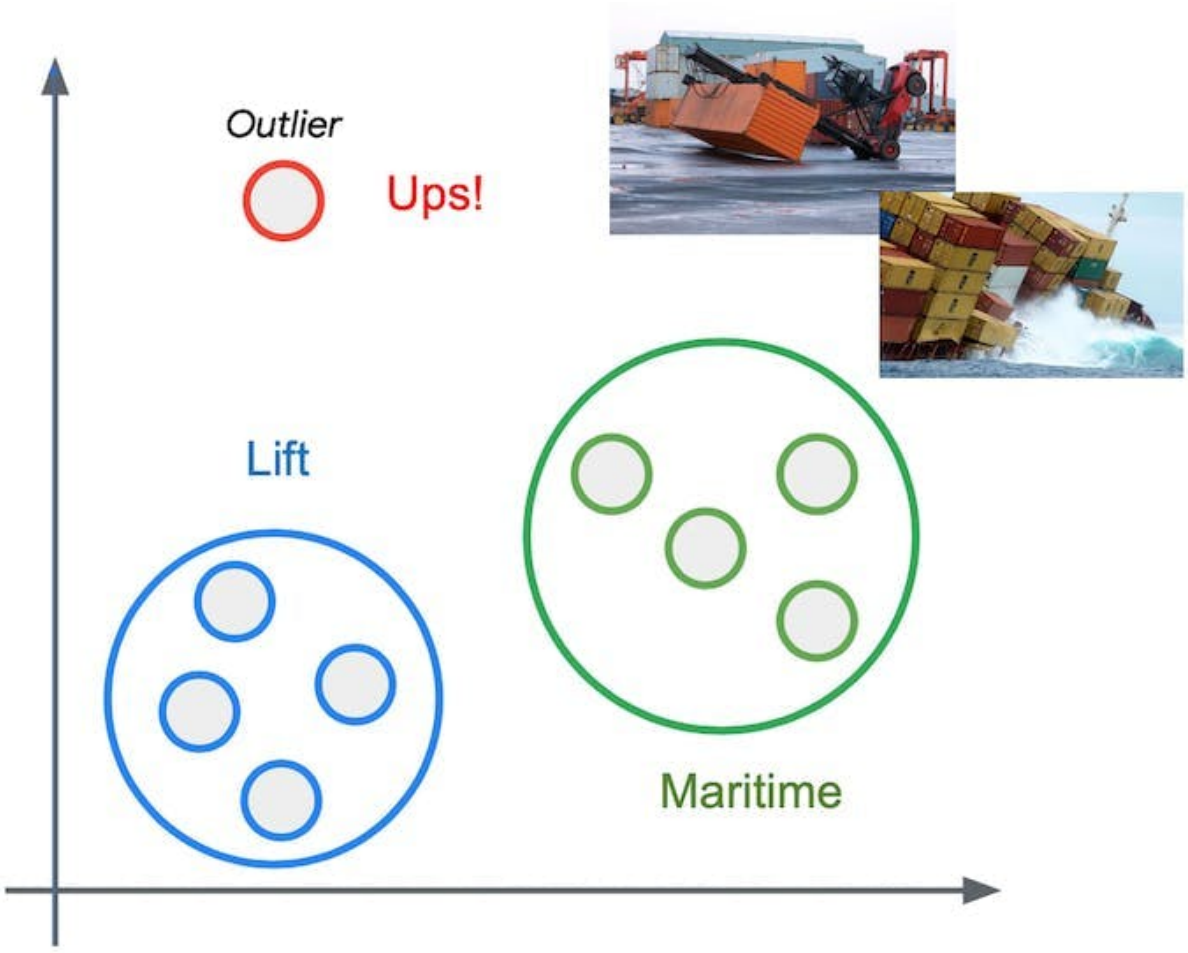
Our classifier will be a Dense Neural Network (DNN) that will have 63 neurons on its input layer, two hidden layers with 20 and 10 neurons, and an output layer with four neurons (one per each class), as shown here:



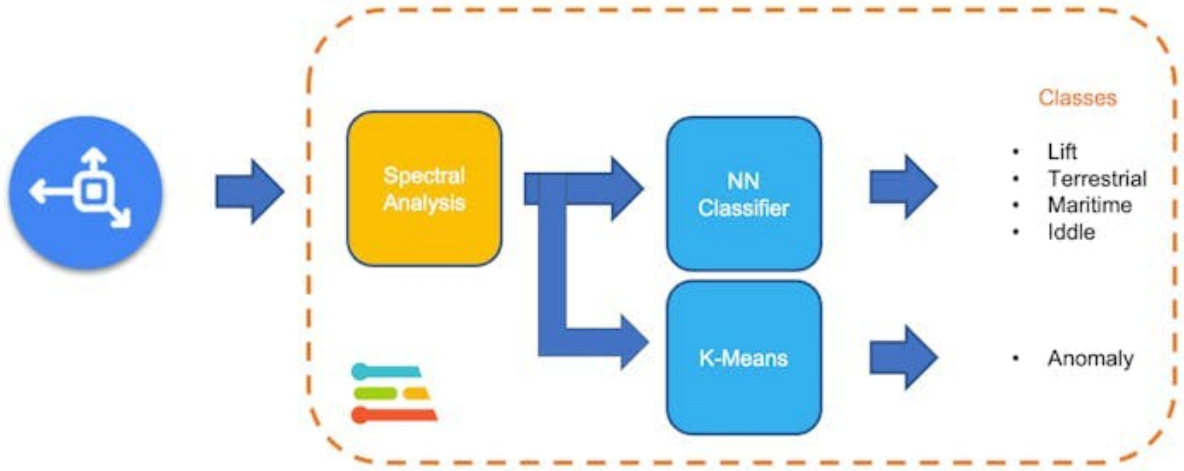
7.8 Impulse Design

An impulse takes raw data, uses signal processing to extract features, and then uses a learning block to classify new data.

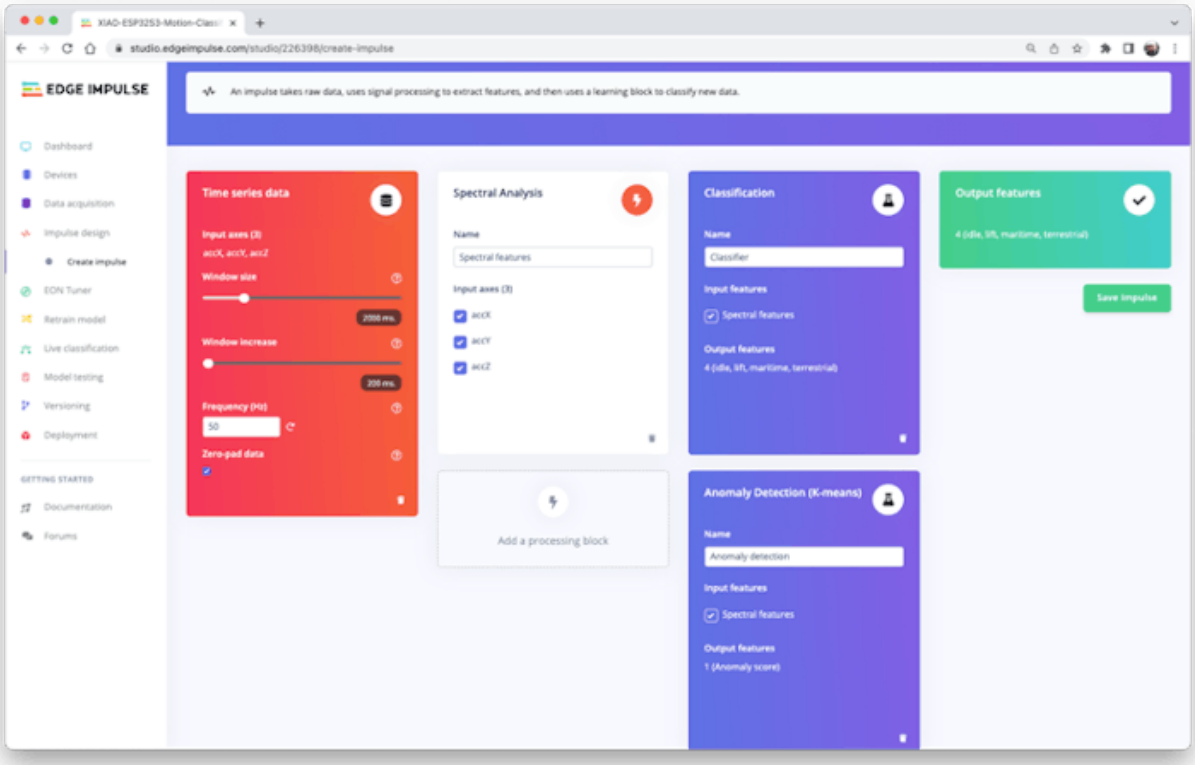
We also take advantage of a second model, the K-means, that can be used for Anomaly Detection. If we imagine that we could have our known classes as clusters, any sample that could not fit on that could be an outlier, an anomaly (for example, a container rolling out of a ship on the ocean).



Imagine our XIAO rolling or moving upside-down, on a movement complement different from the one trained

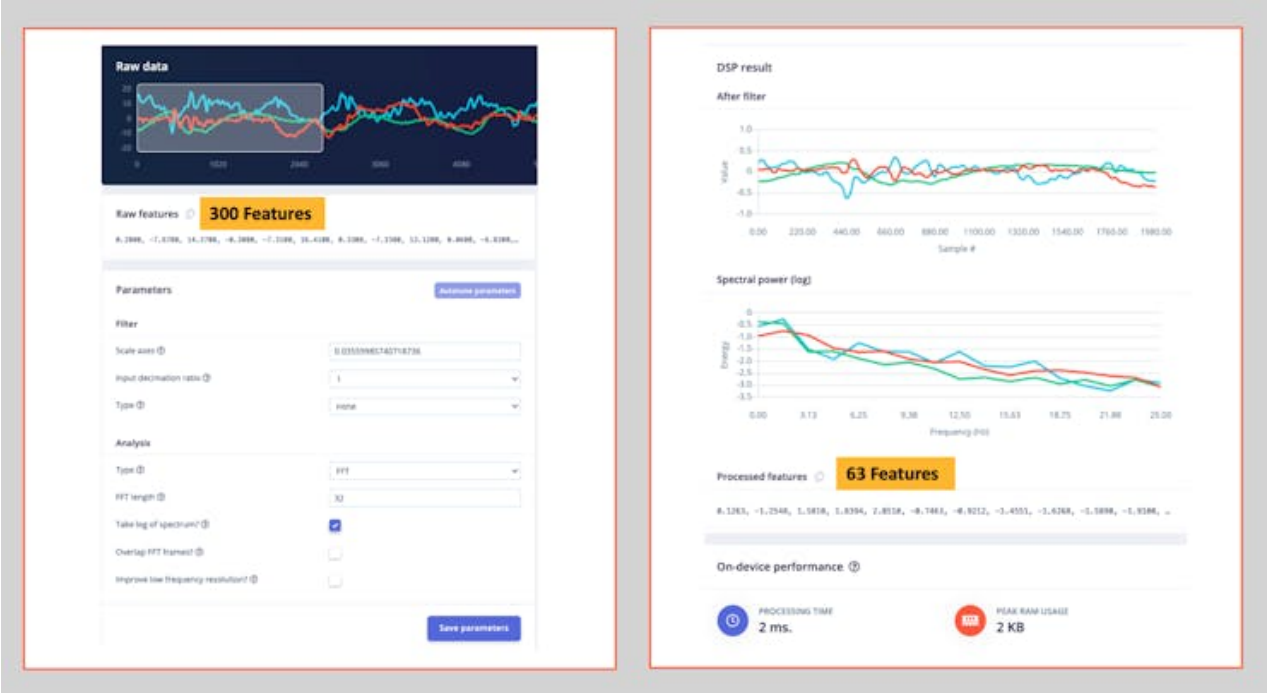


Below is our final Impulse design:



7.9 Generating features

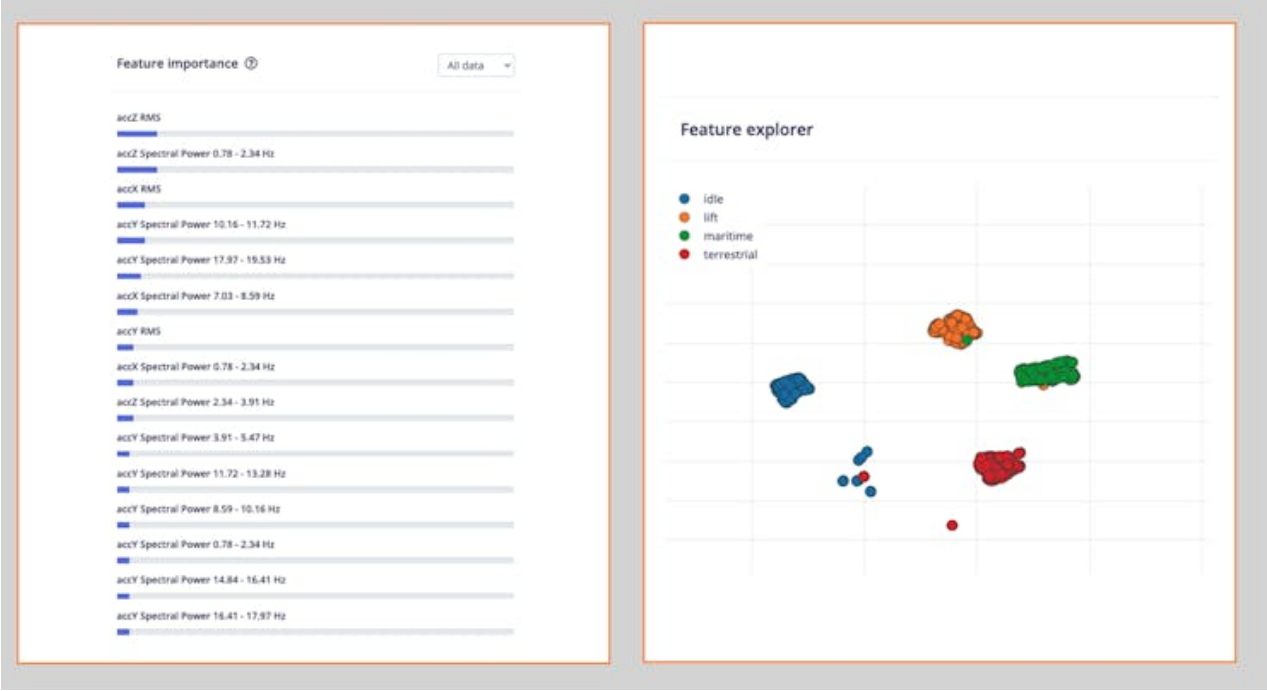
At this point in our project, we have defined the pre-processing method and the model designed. Now, it is time to have the job done. First, let's take the raw data (time-series type) and convert it to tabular data. Go to the Spectral Features tab and select Save Parameters:



At the top menu, select the Generate Features option and the Generate Features button. Each 2-second window data will be converted into one data point of 63 features.

> The Feature Explorer will show those data in 2D using UMAP. Uniform Manifold Approximation and Projection (UMAP) is a dimension reduction technique that can be used for visualization similarly to t-SNE but also for general non-linear dimension reduction.

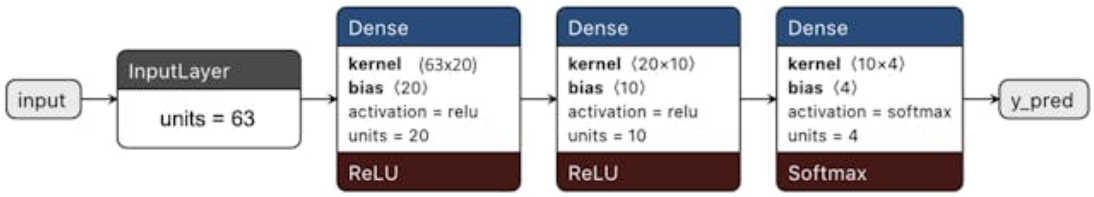
The visualization allows one to verify that the classes present an excellent separation, which indicates that the classifier should work well.

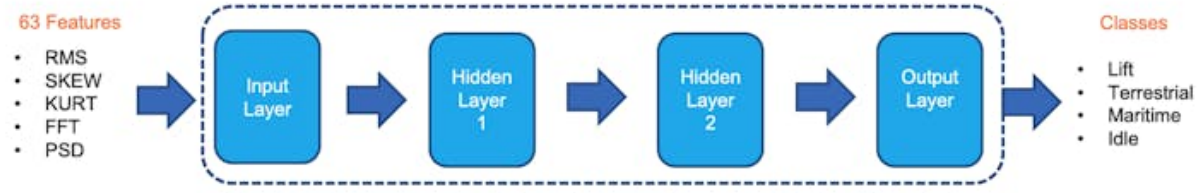


Optionally, you can analyze the relative importance of each feature for one class compared with other classes.

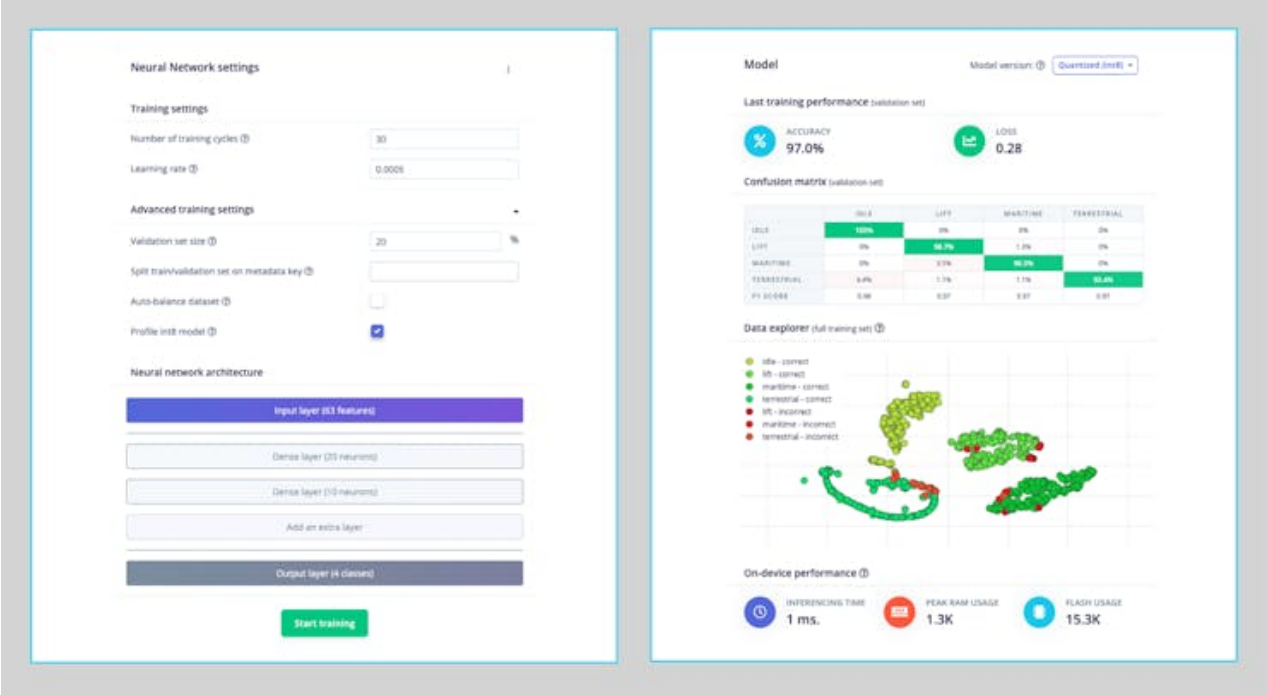
7.10 Training

Our model has four layers, as shown below:

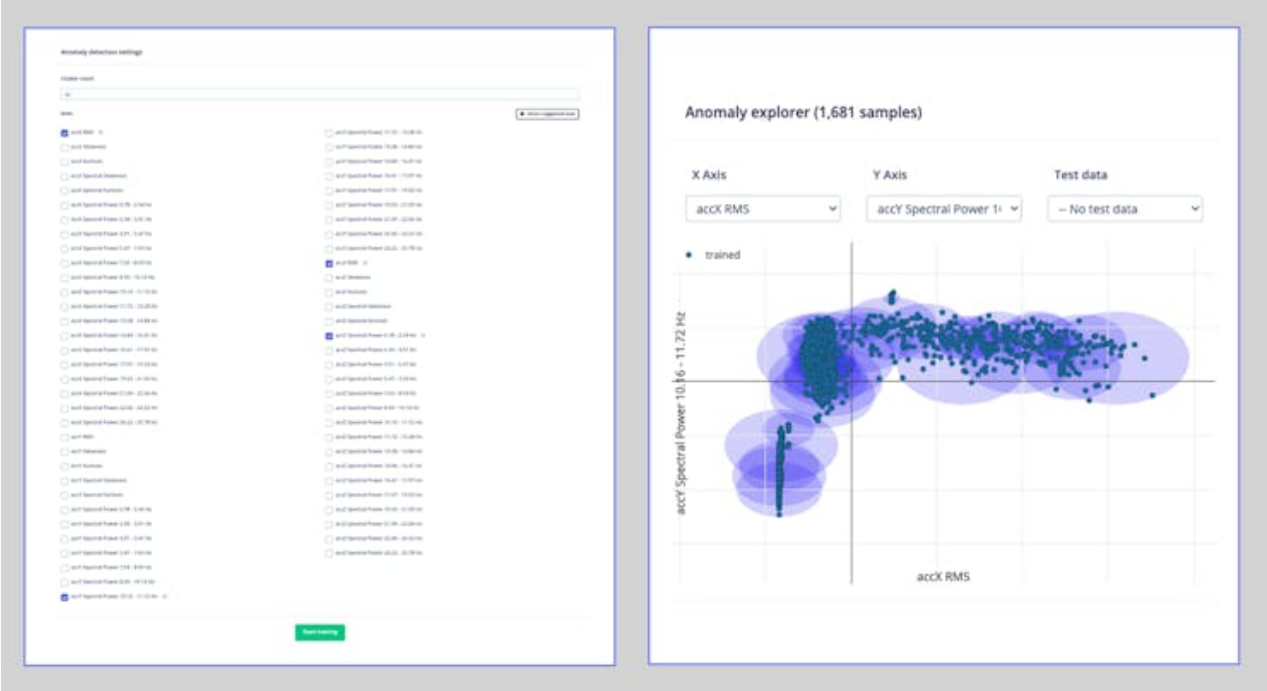




As hyperparameters, we will use a Learning Rate of 0.005 and 20% of data for validation for 30 epochs. After training, we can see that the accuracy is 97%.

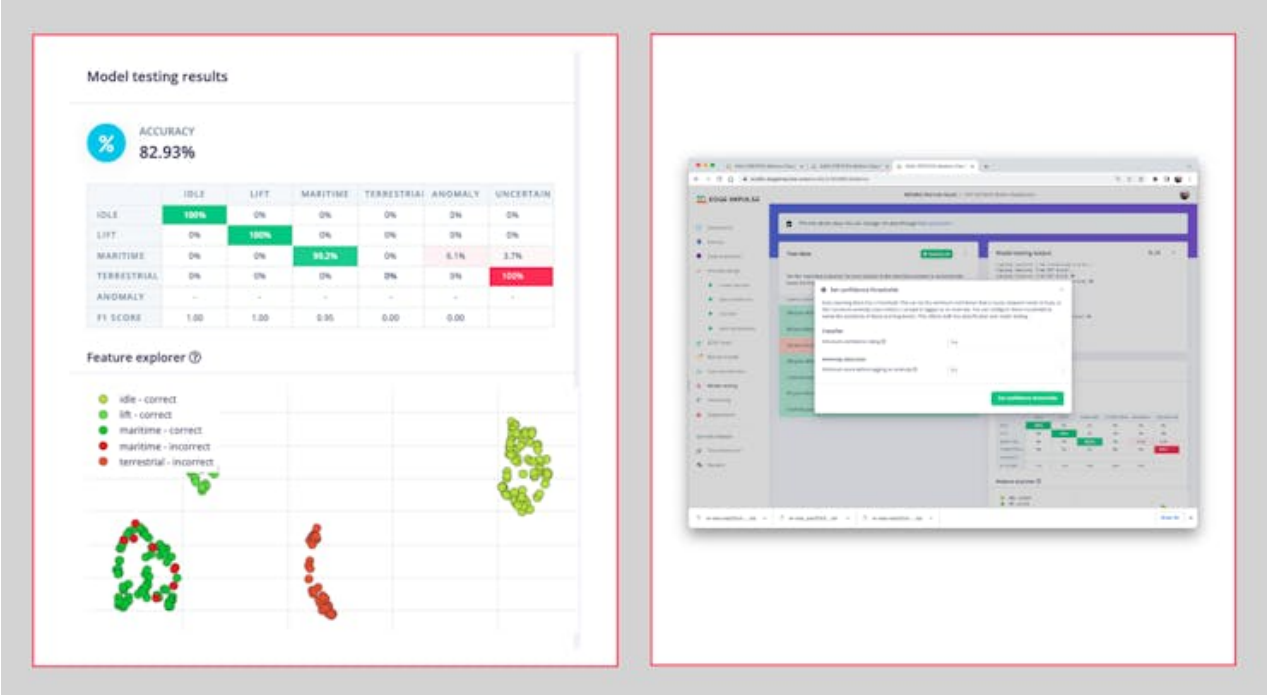


For anomaly detection, we should choose the suggested features that are precisely the most important in feature extraction. The number of clusters will be 32, as suggested by the Studio:

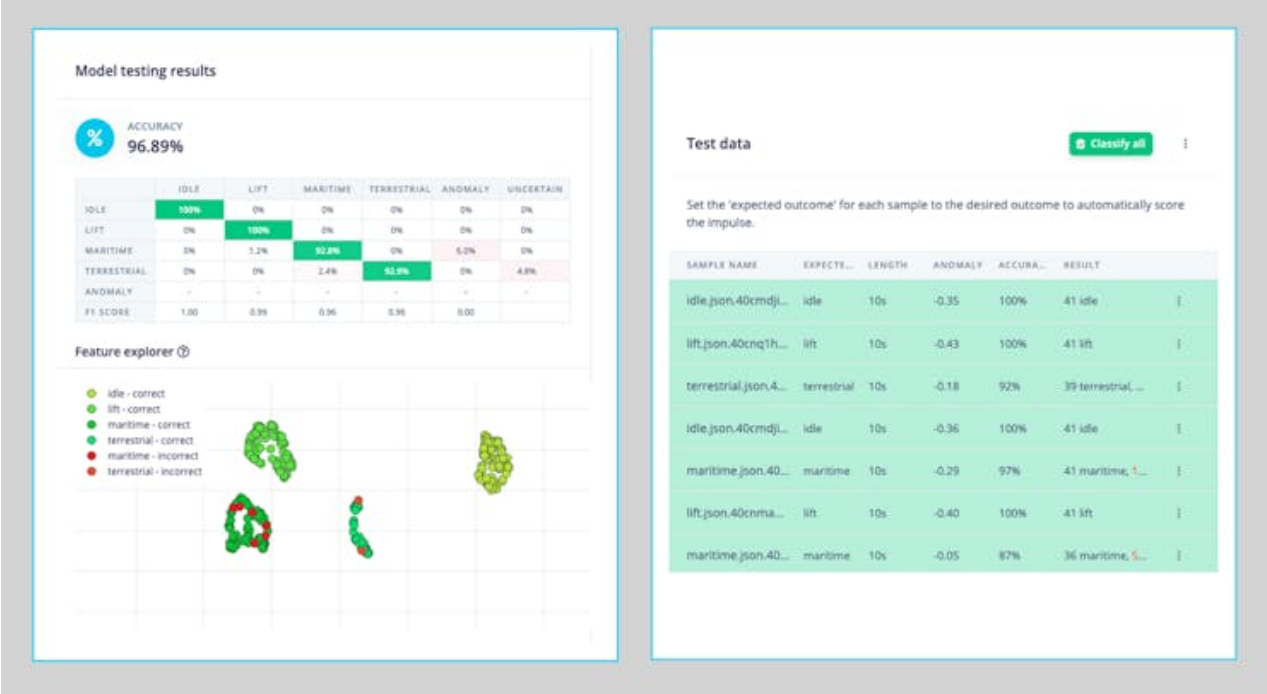


7.11 Testing

Using 20% of the data left behind during the data capture phase, we can verify how our model will behave with unknown data; if not 100% (what is expected), the result was not that good (8%), mainly due to the terrestrial class. Once we have four classes (which output should add 1.0), we can set up a lower threshold for a class to be considered valid (for example, 0.4):



Now, the Test accuracy will go up to 97%.

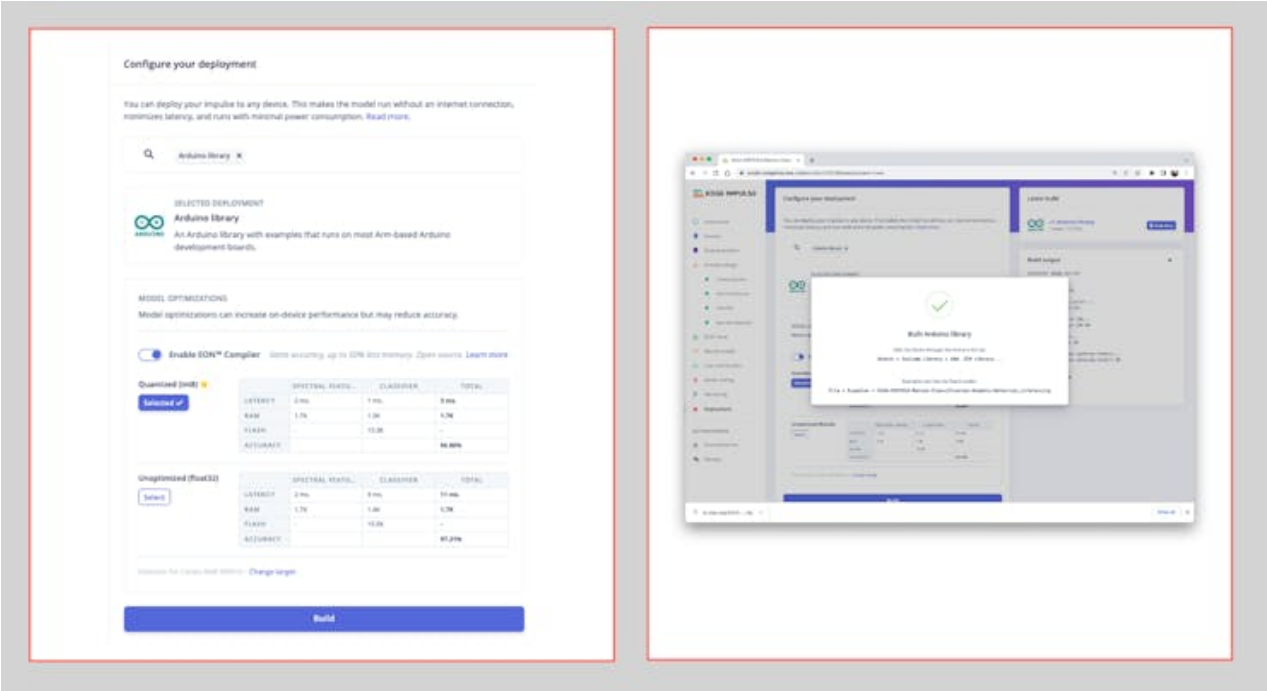


You should also use your device (which is still connected to the Studio) and perform some Live Classification.

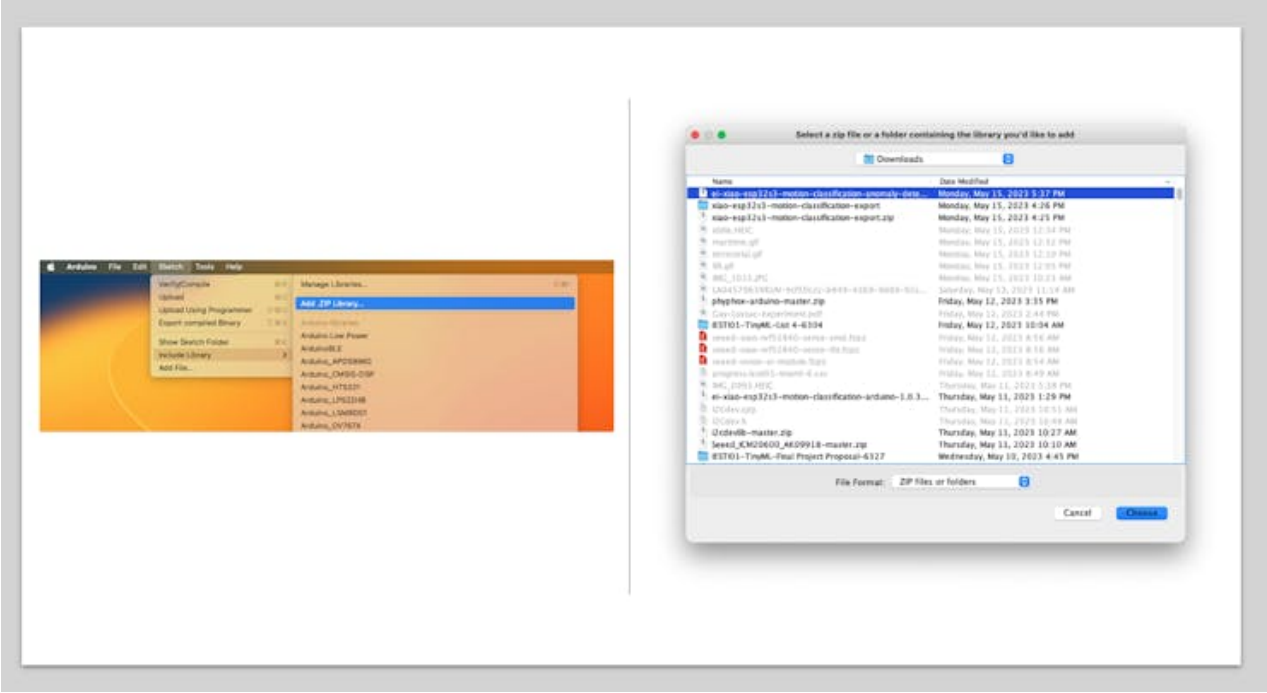
Be aware that here you will capture real data with your device and upload it to the Studio, where an inference will be taken using the trained model (But the model is NOT in your device).

7.12 Deploy

Now it is time for magic˜! The Studio will package all the needed libraries, preprocessing functions, and trained models, downloading them to your computer. You should select the option Arduino Library, and at the bottom, choose Quantized (Int8) and Build. A Zip file will be created and downloaded to your computer.



On your Arduino IDE, go to the Sketch tab, select the option Add.ZIP Library, and Choose the.zip file downloaded by the Studio:



7.13 Inference

Now, it is time for a real test. We will make inferences that are wholly disconnected from the Studio. Let's change one of the code examples created when you deploy the Arduino Library.

In your Arduino IDE, go to the File/Examples tab and look for your project, and on examples, select nano_ble_sense_accelerometer:

Of course, this is not your board, but we can have the code working with only a few changes.

For example, at the beginning of the code, you have the library related to Arduino Sense IMU:

```
/* Includes ---------------------------------------------------------
----- */
#include <XIAO-ESP32S3-Motion-Classification_inferencing.h>
#include <Arduino_LSM9DS1.h>
```

Change the “includes” portion with the code related to the IMU:

```
#include <XIAO-ESP32S3-Motion-Classification_inferencing.h>
#include "I2Cdev.h"
#include "MPU6050.h"
#include "Wire.h"
```

Change the Constant Defines

```
/* Constant defines -------------------------------------------------
----- */
MPU6050 imu;
int16_t ax, ay, az;

#define ACC_RANGE           1 // 0: -/+2G; 1: +/-4G
#define CONVERT_G_TO_MS2    (9.81/(16384/(1.+ACC_RANGE)))
#define MAX_ACCEPTED_RANGE  (2*9.81)+(2*9.81)*ACC_RANGE
```

On the setup function, initiate the IMU set the off-set values and range:

```
// initialize device
Serial.println("Initializing I2C devices...");
Wire.begin();
imu.initialize();
delay(10);

//Set MCU 6050 OffSet Calibration
imu.setXAccelOffset(-4732);
imu.setYAccelOffset(4703);
imu.setZAccelOffset(8867);
imu.setXGyroOffset(61);
imu.setYGyroOffset(-73);
imu.setZGyroOffset(35);

imu.setFullScaleAccelRange(ACC_RANGE);
```

At the loop function, the buffers buffer[ix], buffer[ix + 1], and buffer[ix + 2] will receive the 3-axis data captured by the accelerometer. On the original code, you have the line:

```
IMU.readAcceleration(buffer[ix], buffer[ix + 1], buffer[ix + 2]);
```

Change it with this block of code:

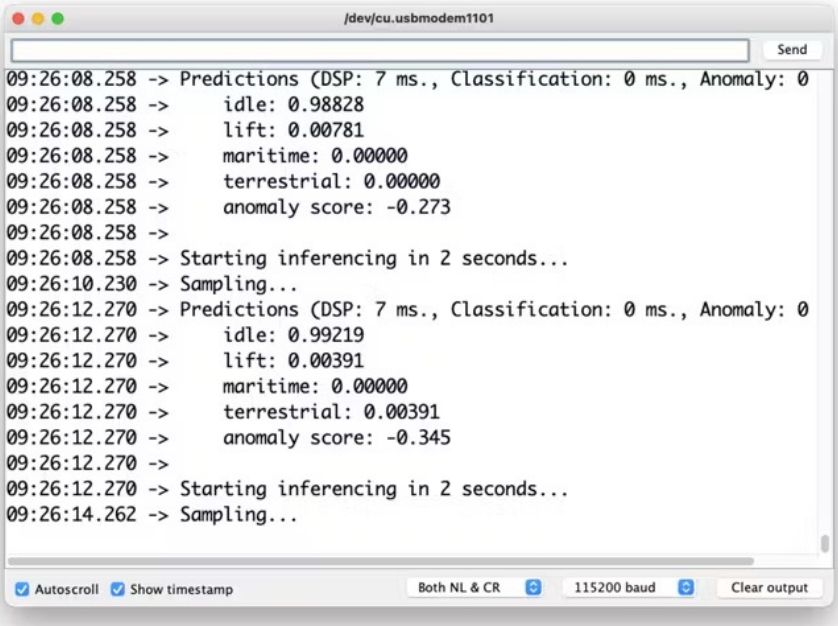
```
imu.getAcceleration(&ax, &ay, &az);
buffer[ix + 0] = ax;
buffer[ix + 1] = ay;
buffer[ix + 2] = az;
```

You should change the order of the following two blocks of code. First, you make the conversion to raw data to “Meters per squared second (ms2)”, followed by the test regarding the maximum acceptance range (that here is in ms2, but on Arduino, was in Gs):

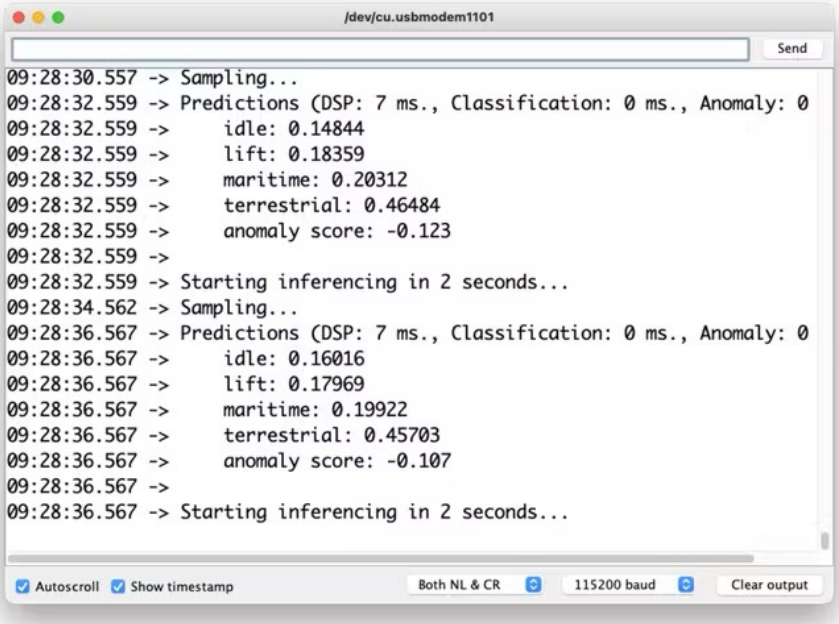
```
buffer[ix + 0] *= CONVERT_G_TO_MS2;
buffer[ix + 1] *= CONVERT_G_TO_MS2;
buffer[ix + 2] *= CONVERT_G_TO_MS2;

for (int i = 0; i < 3; i++) {
     if (fabs(buffer[ix + i]) > MAX_ACCEPTED_RANGE) {
        buffer[ix + i] = ei_get_sign(buffer[ix + i]) *
MAX_ACCEPTED_RANGE;
     }
}
```

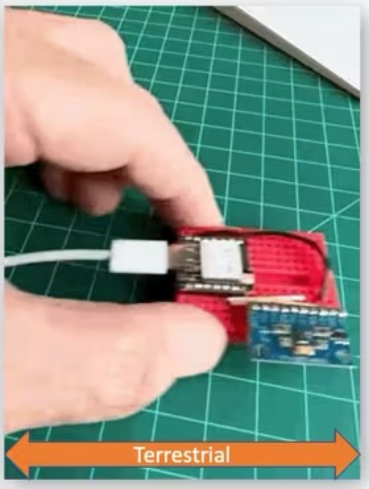
And that is it! You can now upload the code to your device and proceed with the inferences. The complete code is available on the project’s GitHub.

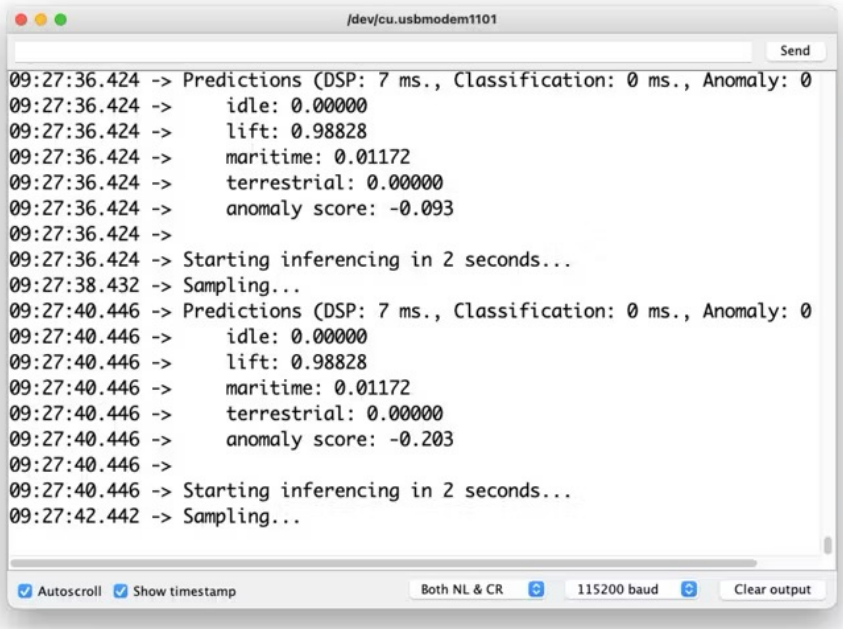
Now you should try your movements, seeing the result of the inference of each class on the images:

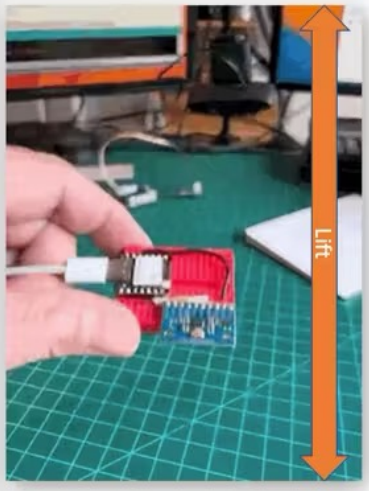

/dev/cu.usbmodem1101
Send
09:26:08.258 -> Predictions (DSP: 7 ms., Classification: 0 ms., Anomaly: 0
09:26:08.258 ->   idle: 0.98828
09:26:08.258 ->   lift: 0.00781
09:26:08.258 ->   maritime: 0.00000
09:26:08.258 ->   terrestrial: 0.00000
09:26:08.258 ->   anomaly score: -0.273
09:26:08.258 ->
09:26:08.258 -> Starting inferencing in 2 seconds...
09:26:10.230 -> Sampling...
09:26:12.270 -> Predictions (DSP: 7 ms., Classification: 0 ms., Anomaly: 0
09:26:12.270 ->   idle: 0.99219
09:26:12.270 ->   lift: 0.00391
09:26:12.270 ->   maritime: 0.00000
09:26:12.270 ->   terrestrial: 0.00391
09:26:12.270 ->   anomaly score: -0.345
09:26:12.270 ->
09:26:12.270 -> Starting inferencing in 2 seconds...
09:26:14.262 -> Sampling...
Autoscroll
Show timestamp
Both NL & CR
115200 baud
Clear output

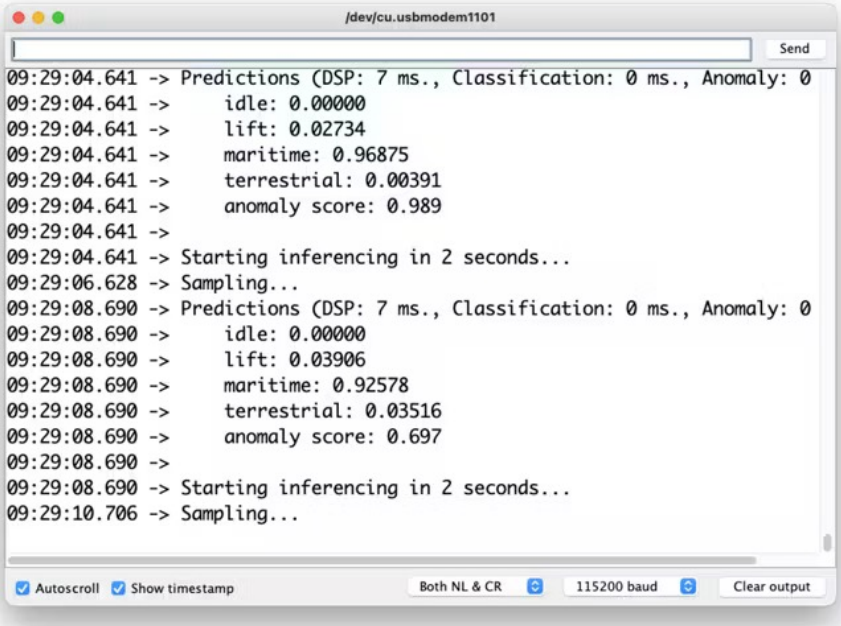

Idle

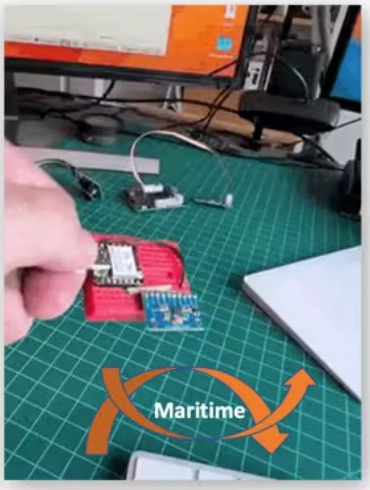

/dev/cu.usbmodem1101
Send
09:28:30.557 -> Sampling...
09:28:32.559 -> Predictions (DSP: 7 ms., Classification: 0 ms., Anomaly: 0
09:28:32.559 ->   idle: 0.14844
09:28:32.559 ->   lift: 0.18359
09:28:32.559 ->   maritime: 0.20312
09:28:32.559 ->   terrestrial: 0.46484
09:28:32.559 ->   anomaly score: -0.123
09:28:32.559 ->
09:28:32.559 -> Starting inferencing in 2 seconds...
09:28:34.562 -> Sampling...
09:28:36.567 -> Predictions (DSP: 7 ms., Classification: 0 ms., Anomaly: 0
09:28:36.567 ->   idle: 0.16016
09:28:36.567 ->   lift: 0.17969
09:28:36.567 ->   maritime: 0.19922
09:28:36.567 ->   terrestrial: 0.45703
09:28:36.567 ->   anomaly score: -0.107
09:28:36.567 ->
09:28:36.567 -> Starting inferencing in 2 seconds...
Autoscroll
Show timestamp
Both NL & CR
115200 baud
Clear output


Terrestrial

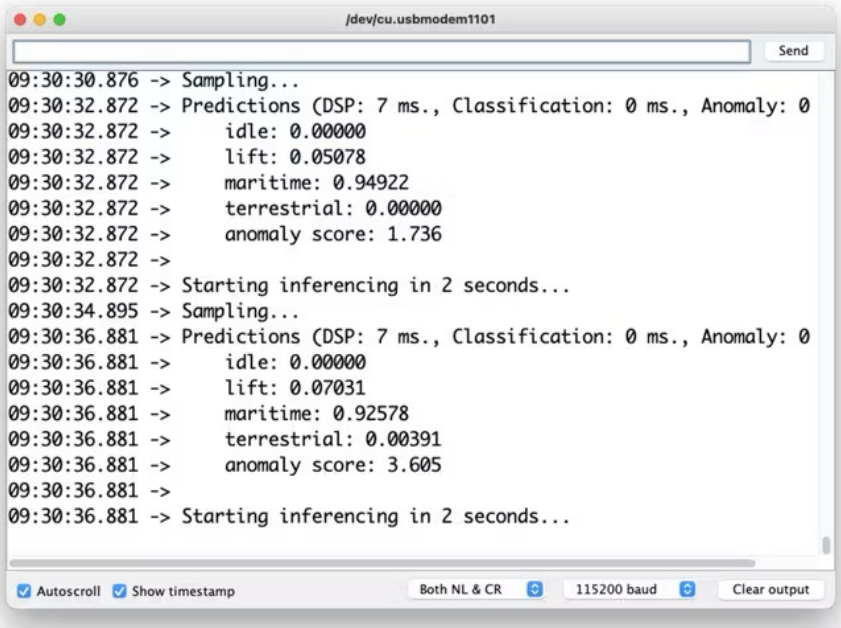


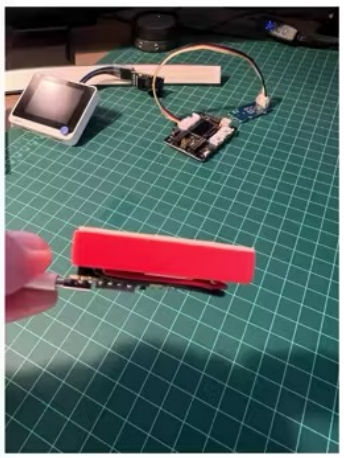






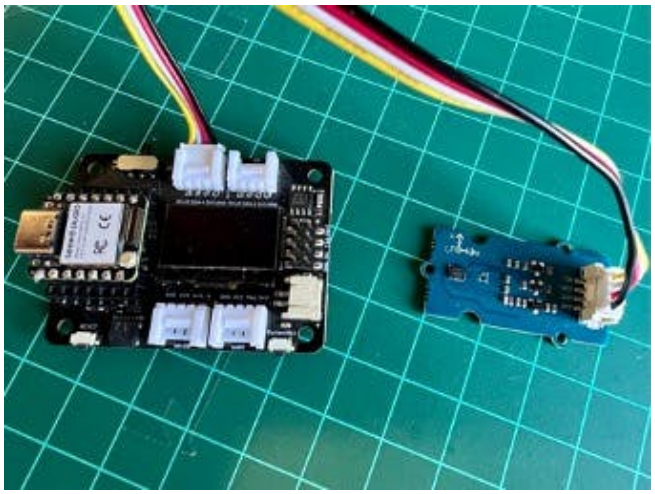
And, of course, some “anomaly”, for example, putting the XIAO upside-down. The anomaly score will be over 1:





7.14 Conclusion.

Regarding the IMU, this project used the low-cost MPU6050 but could also use other IMUs, for example, the LCM20600 (6-axis), which is part of the Seeed Grove - IMU 9DOF (lcm20600+AK09918). You can take advantage of this sensor, which has integrated a Grove connector, which can be helpful in the case you use the XIAO with an extension board, as shown below:



You can follow the instructions here to connect the IMU with the MCU. Only note that for using the Grove ICM20600 Accelerometer, it is essential to update the files **I2Cdev.cpp** and **I2Cdev.h** that you will download from the library provided by Seeed Studio. For that, replace both files from this link. You can find a sketch for testing the IMU on the GitHub project: accelerometer_test.ino.

On the projet's GitHub repository, you will find the last version of all codes and other docs: XIAO-ESP32S3 - IMU.

References

To learn more:

Online Courses

- Harvard School of Engineering and Applied Sciences - CS249r: Tiny Machine Learning
- 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
- UNIFEI-IESTI01 TinyML: “Machine Learning for Embedding Devices”

Books

- “Python for Data Analysis” by Wes McKinney
- “Deep Learning with Python” by François Chollet - GitHub Notebooks
- “TinyML” by Pete Warden and Daniel Situnayake
- “TinyML Cookbook 2nd Edition” by Gian Marco Iodice
- “Technical Strategy for AI Engineers, In the Era of Deep Learning” by Andrew Ng
- “AI at the Edge” book by Daniel Situnayake and Jenny Plunkett
- “XIAO: Big Power, Small Board” by Lei Feng and Marcelo Rovai
- “MACHINE LEARNING SYSTEMS for TinyML” by a collaborative effort

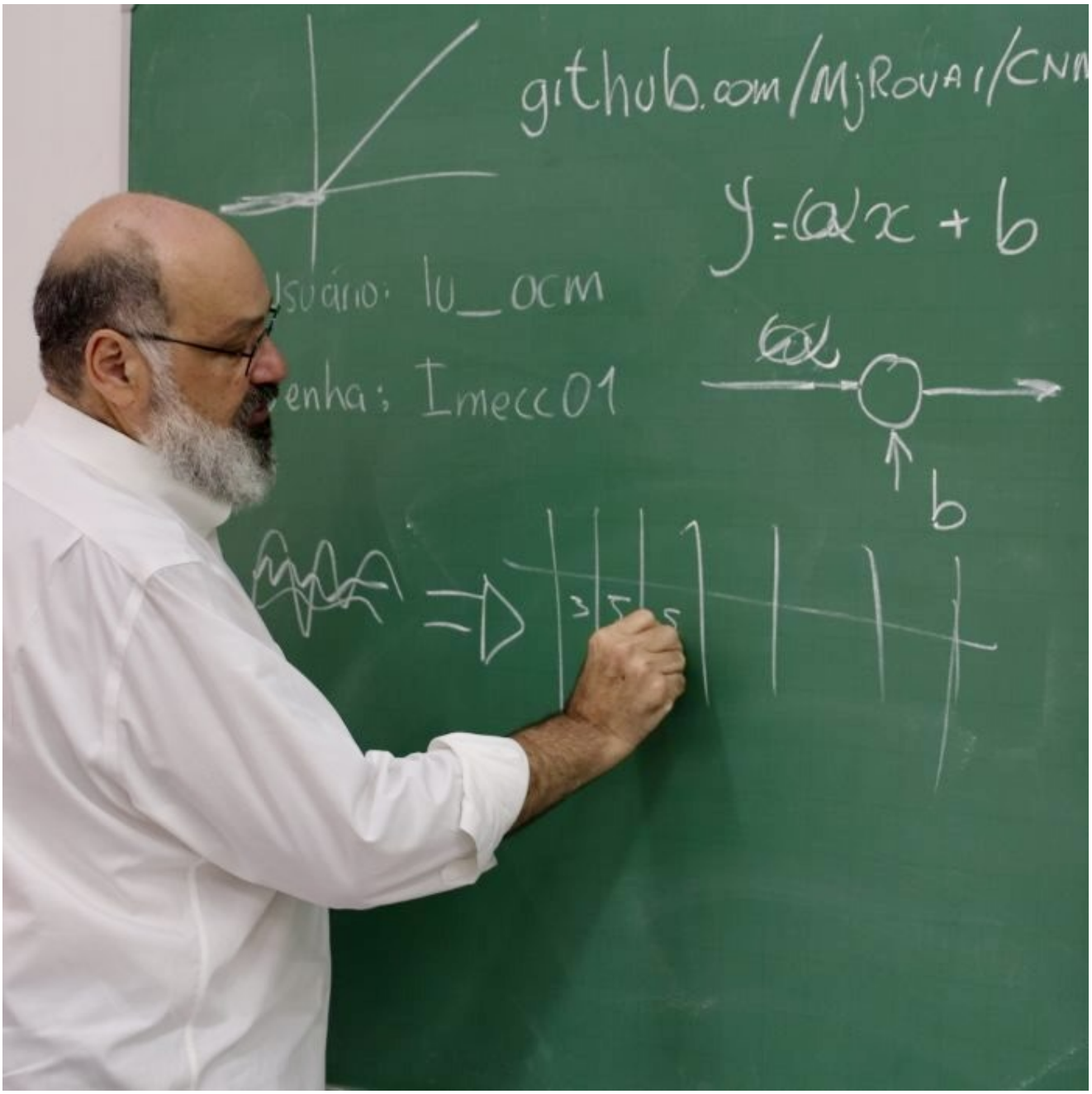
Projects Repository

- Edge Impulse Expert Network

TinyML4D

TinyML Made Easy, an eBook collection of a series of Hands-On tutorials, is part of the TinyML4D, an initiative to make Embedded Machine Learning (TinyML) education available to everyone, explicitly enabling innovative solutions for the unique challenges Developing Countries face.

About the author



Marcelo Rovai, a Brazilian living in Chile, is a recognized figure in engineering and technology education. He holds the title of Professor Honoris Causa from the Federal University of Itajubá (UNIFEI), Brazil. His

educational background includes an Engineering degree from UNIFEI and a specialization from the Polytechnic School of São Paulo University (USP). Further enhancing his expertise, he earned an MBA from IBMEC (INSPER) and a Master's in Data Science from the Universidad del Desarrollo (UDD) in Chile.

With a career spanning several high-profile technology companies such as AVIBRAS Airspace, ATT, NCR, and IGT, where he served as Vice President for Latin America, he brings a wealth of industry experience to his academic endeavors. He is a prolific writer on electronics-related topics and shares his knowledge through open platforms like Hackster.io.

In addition to his professional pursuits, he is dedicated to educational outreach, serving as a volunteer professor at UNIFEI and engaging with the TinyML4D group as a Co-Chair, promoting TinyML education in developing countries. His work underscores a commitment to leveraging technology for societal advancement.

LinkedIn profile: https://www.linkedin.com/in/marcelo-jose-rovai-brazil-chile/

Lectures, books, papers, and tutorials:
https://github.com/Mjrovai/TinyML4D

Table of Contents