

TinyML Made Easy

Hands-On with the Nicla Vision



Marcelo Rovai
November, 2023

TinyML Made Easy

Hands-On with the Nicla Vision

Marcelo Rovai

2023-10-28

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 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 doing the exercises will enable the readers to undertake other projects that might interest them.

Ermanno Pietrosemoli, President Fundación Escuela Latinoamericana de Redes

October 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, 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.

We also thank Ermanno Petrosemoli for his meticulous review of the content in this material.

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 cheap electronic components, usually with just a few kilobytes of RAM, and 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. But, curiously, these devices receive little attention because, many times, they are used just to replace functionalities that older electromechanical systems face in cars, washing machines, or remote controls.

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).

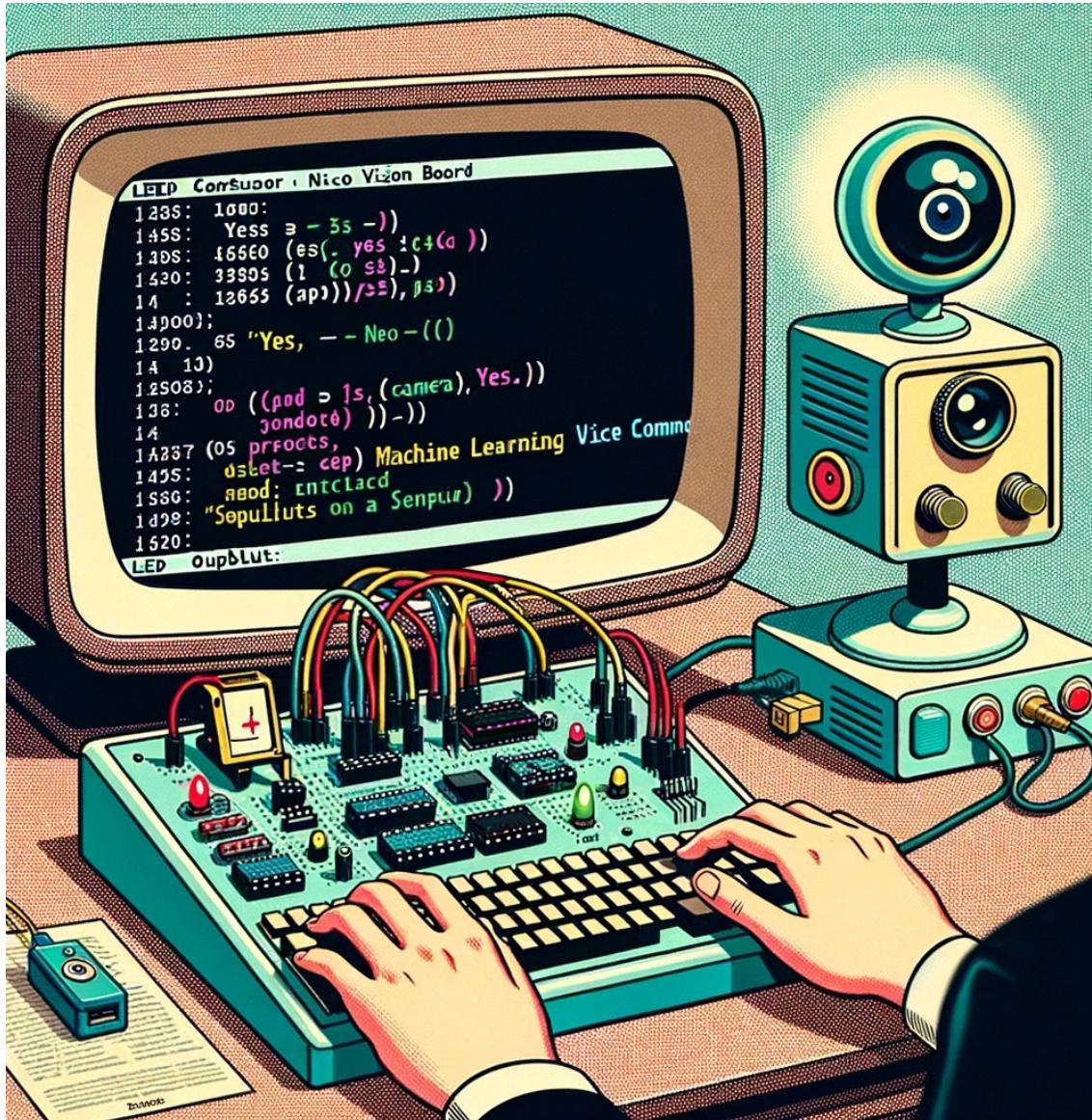
On the other hand, in the last decades, we have witnessed the development of Machine Learning models (sub-area of Artificial Intelligence) trained with “tons” of data and powerful mainframes. But now, it suddenly becomes possible for “noisy” and complex signals, such as images, audio, or accelerometers, 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 extracting “machine intelligence” from the physical world (where the data is generated).

The **WALC 2023 Applied AI Track** is an introductory course on the intersection between Machine Learning and Embedded Devices. The spread of embedded devices with ultra-low power consumption

(on the order of milliwatts), together with the introduction of machine learning frameworks dedicated to embedded devices, such as TensorFlow Lite for Microcontrollers (TF Lite Micro), allow the mass proliferation of IoT devices empowered by AI (“AioT”).

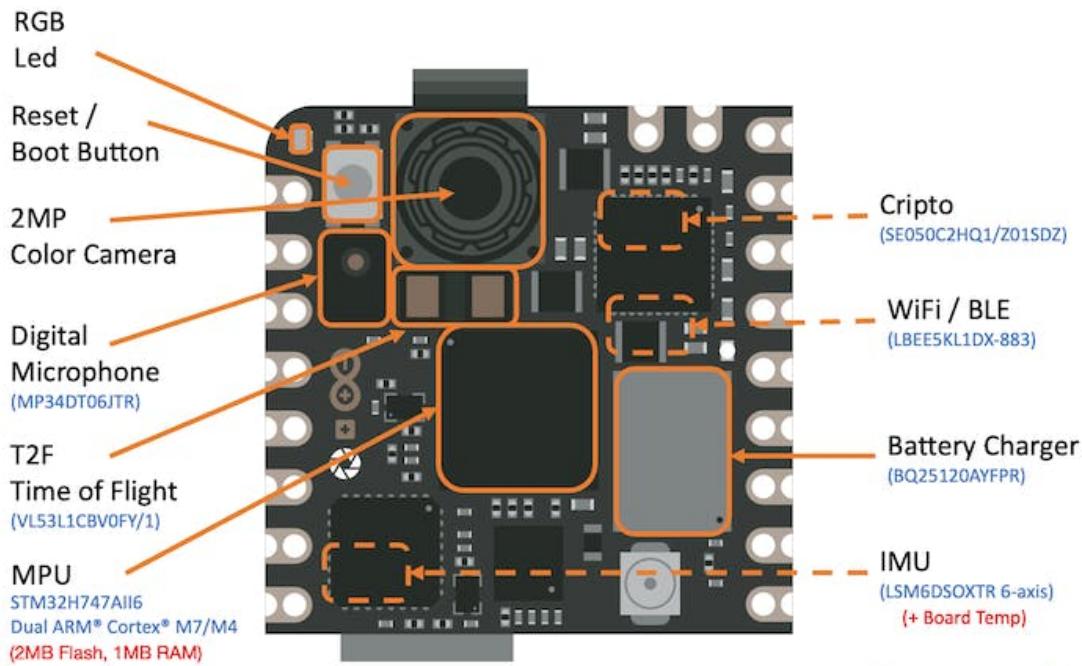
1 Setup Nicla Vision



1.1 Introduction

The [Arduino Nicla Vision](#) (sometimes called *NiclaV*) is a development board that includes two processors that can run tasks in parallel. It is part of a family of development boards with the same form factor but designed for specific tasks, such as the [Nicla Sense ME](#) and the [Nicla Voice](#). The

Niclas can efficiently run processes created with TensorFlow™ Lite. For example, one of the cores of the NiclaV runs a computer vision algorithm on the fly (inference), while the other executes low-level operations like controlling a motor and communicating or acting as a user interface. The onboard wireless module allows the management of WiFi and Bluetooth Low Energy (BLE) connectivity simultaneously.



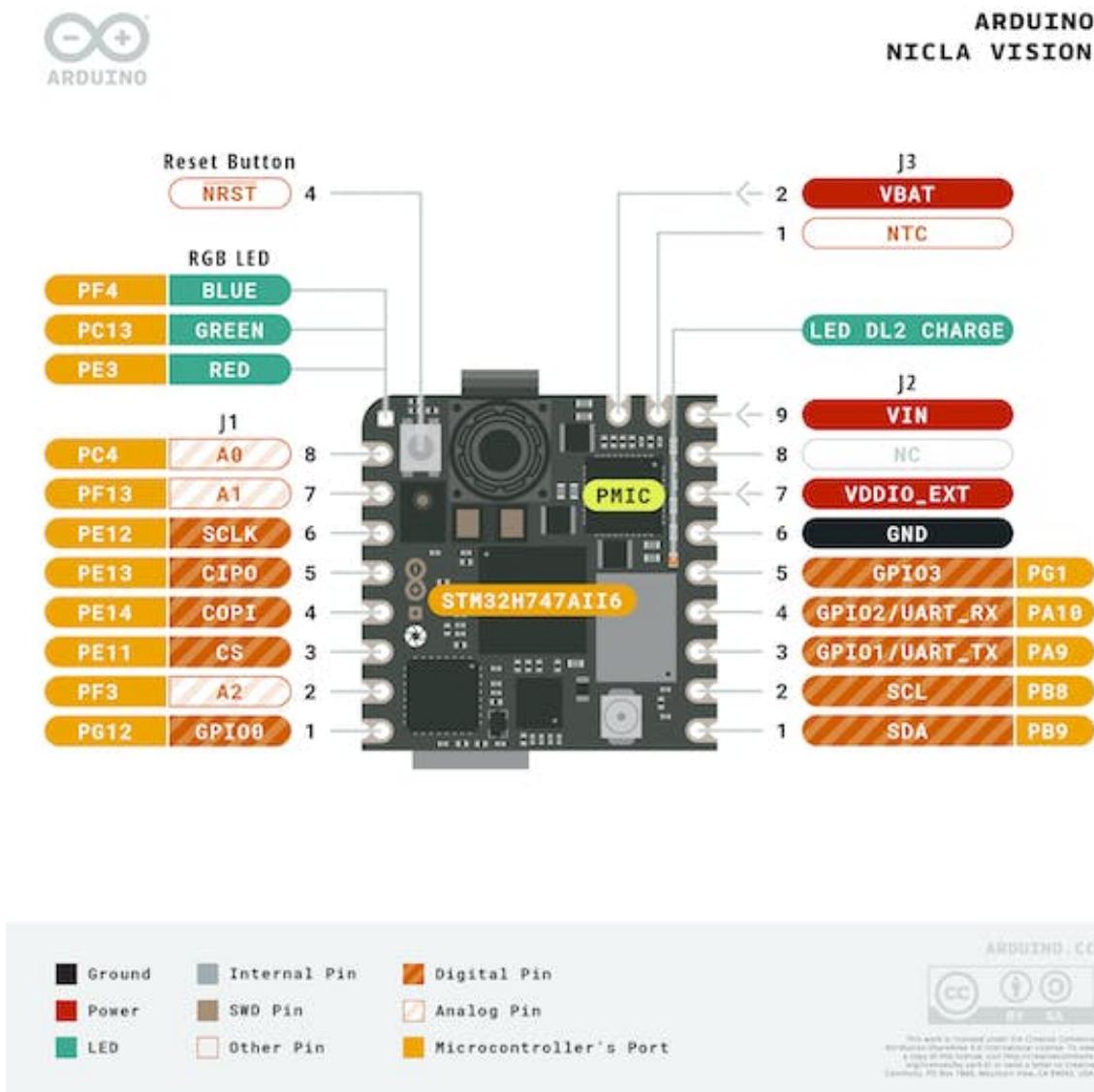
1.2 Hardware

1.2.1 Two Parallel Cores

The central processor is the dual-core [STM32H747](#), including a Cortex® M7 at 480 MHz and a Cortex® M4 at 240 MHz. The two cores communicate via a Remote Procedure Call mechanism that seamlessly allows calling functions on the other processor. Both processors share all the on-chip peripherals and can run:

- Arduino sketches on top of the Arm® Mbed™ OS
- Native Mbed™ applications

- MicroPython / JavaScript via an interpreter
- TensorFlow™ Lite



1.2.2 Memory

Memory is crucial for embedded machine learning projects. The NiclaV board can host up to 16 MB of QSPI Flash for storage. However, it is essential to consider that the MCU SRAM is the one to be used with machine learning inferences; the STM32H747 is only 1MB, shared by

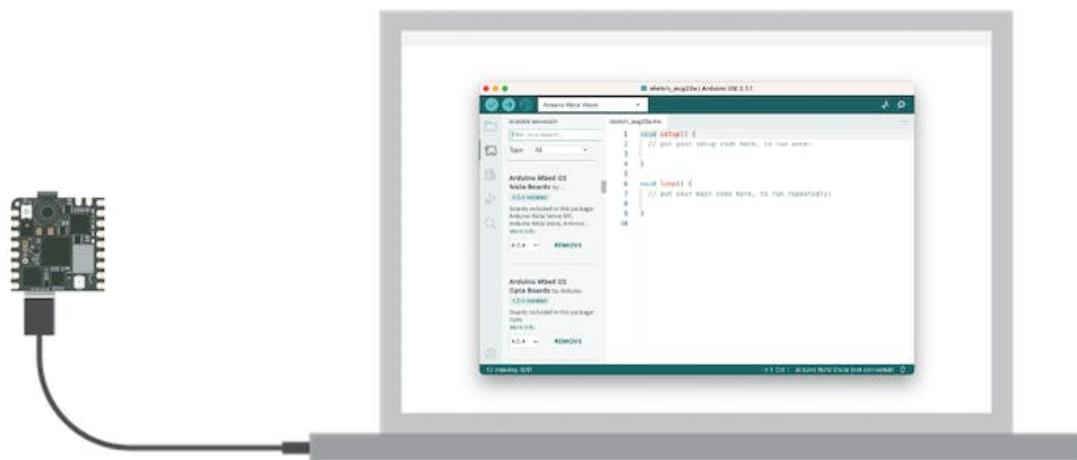
both processors. This MCU also has incorporated 2MB of FLASH, mainly for code storage.

1.2.3 Sensors

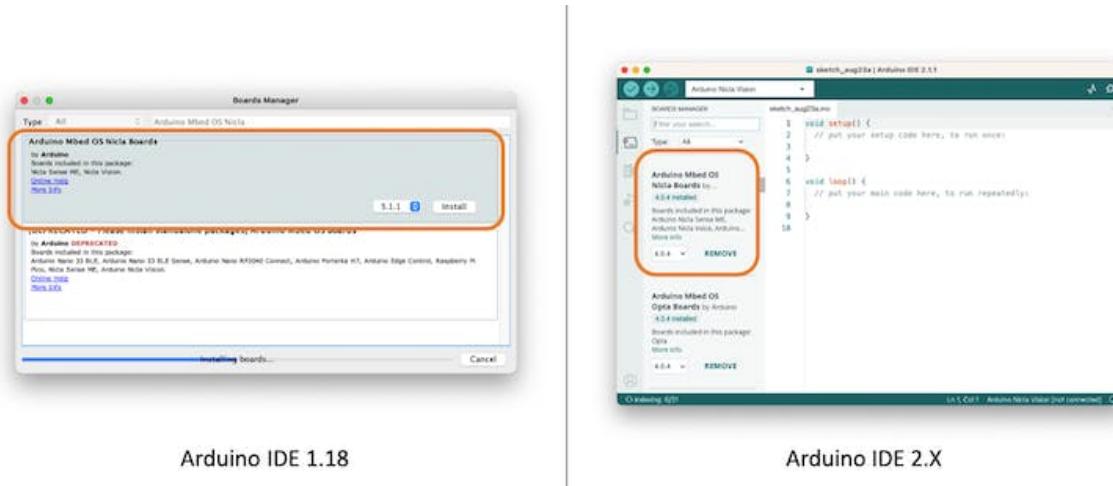
- **Camera:** A GC2145 2 MP Color CMOS Camera.
- **Microphone:** The MP34DT05 is an ultra-compact, low-power, omnidirectional, digital MEMS microphone built with a capacitive sensing element and the IC interface.
- **6-Axis IMU:** 3D gyroscope and 3D accelerometer data from the LSM6DSOX 6-axis IMU.
- **Time of Flight Sensor:** The VL53L1CBV0FY Time-of-Flight sensor adds accurate and low power-ranging capabilities to the Nicla Vision. The invisible near-infrared VCSEL laser (including the analog driver) is encapsulated with receiving optics in an all-in-one small module below the camera.

1.3 Arduino IDE Installation

Start connecting the board (*microUSB*) to your computer:



Install the Mbed OS core for Nicla boards in the Arduino IDE. Having the IDE open, navigate to Tools > Board > Board Manager, look for Arduino Nicla Vision on the search window, and install the board.



Next, go to Tools > Board > Arduino Mbed OS Nicla Boards and select Arduino Nicla Vision. Having your board connected to the USB, you should see the Nicla on Port and select it.

Open the Blink sketch on Examples/Basic and run it using the IDE Upload button. You should see the Built-in LED (green RGB) blinking, which means the Nicla board is correctly installed and functional!

1.3.1 Testing the Microphone

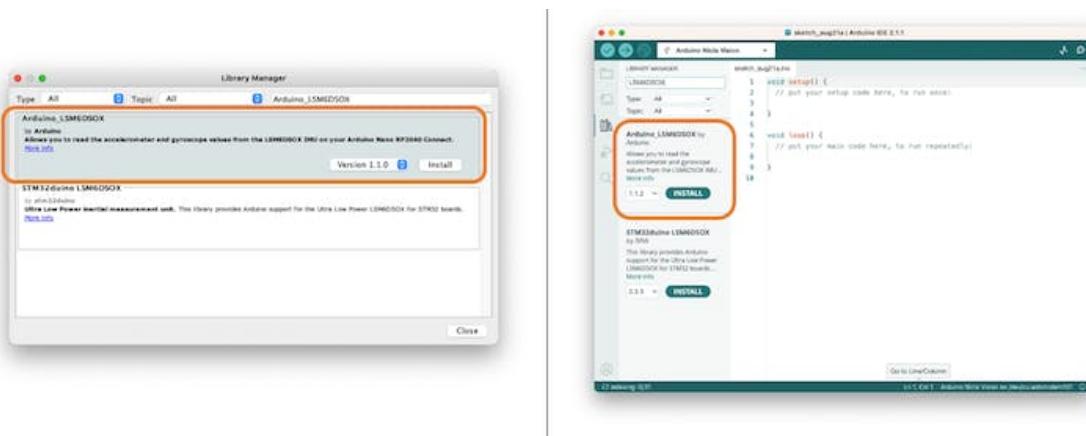
On Arduino IDE, go to Examples > PDM > PDMSerialPlotter, open and run the sketch. Open the Plotter and see the audio representation from the microphone:



Vary the frequency of the sound you generate and confirm that the mic is working correctly.

1.3.2 Testing the IMU

Before testing the IMU, it will be necessary to install the LSM6DSOX library. For that, go to Library Manager and look for LSM6DSOX. Install the library provided by Arduino:

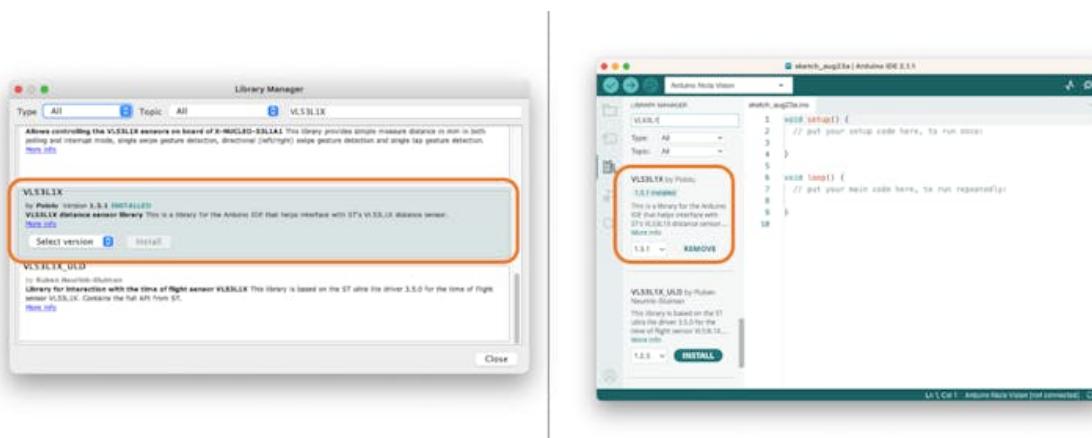


Next, go to Examples > Arduino_LSM6DSOX > SimpleAccelerometer and run the accelerometer test (you can also run Gyro and board temperature):

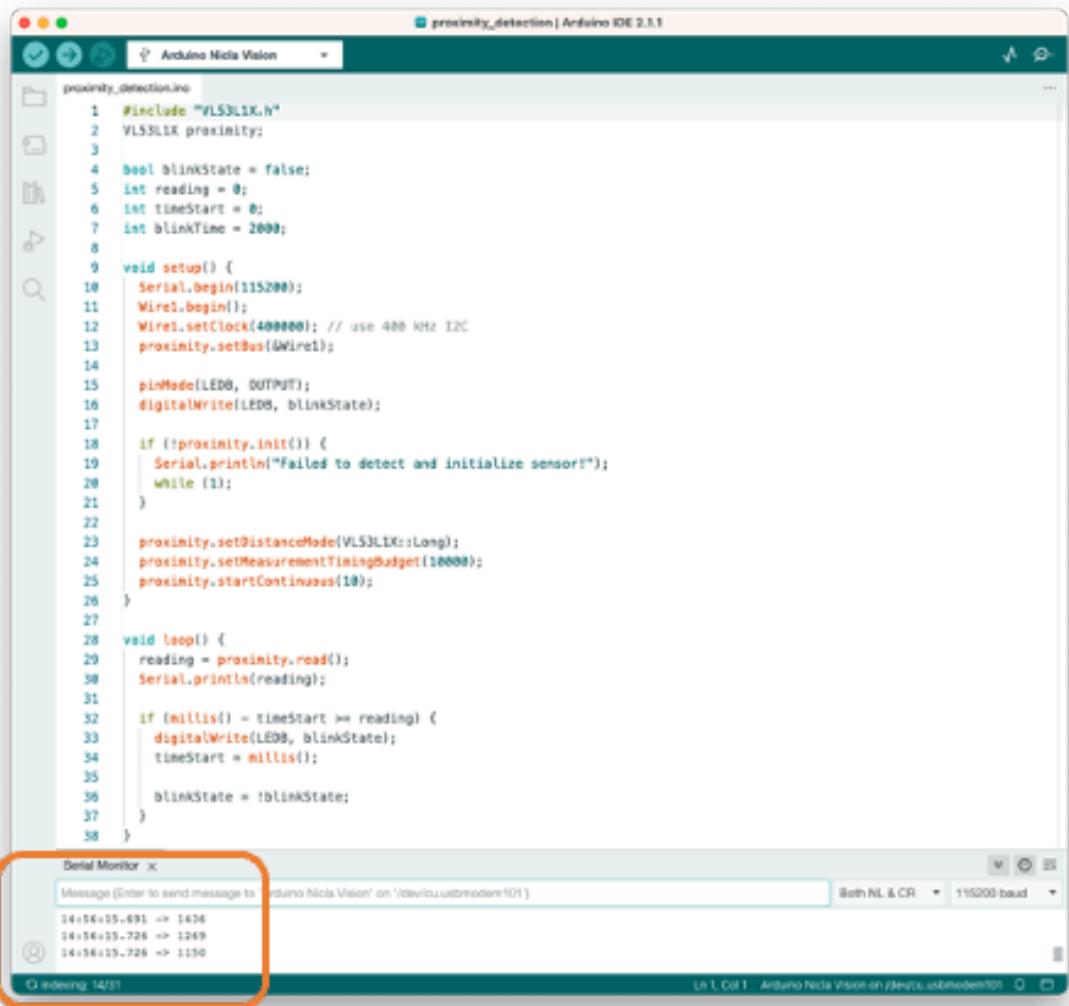


1.3.3 Testing the ToF (Time of Flight) Sensor

As we did with IMU, it is necessary to install the VL53L1X ToF library. For that, go to Library Manager and look for VL53L1X. Install the library provided by Pololu:



Next, run the sketch `proximity_detection.ino`:



The screenshot shows the Arduino IDE interface with the following details:

- Title Bar:** proximity_detection | Arduino IDE 2.1.1
- Sketch Area:** The code for `proximity_detection.ino` is displayed. It includes the `VL53L1X.h` library, initializes the sensor, sets up I2C communication, and implements a loop to read distance measurements and update a LED.
- Serial Monitor:** A window titled "Serial Monitor" is open at the bottom. It shows the message "Message (Enter to send message to 'Arduino Nicta Vision' on 'Ydevicu.usbmodem101')". Below this, it displays three lines of data:
 - 14:56:25-691 => 1496
 - 14:56:25-726 => 1269
 - 14:56:25-726 => 1150The last line is highlighted with a red oval.
- Status Bar:** Shows "Line 1, Col 1" and "Arduino Nicta Vision on Ydevicu.usbmodem101".

On the Serial Monitor, you will see the distance from the camera to an object in front of it (max of 4m).



1.3.4 Testing the Camera

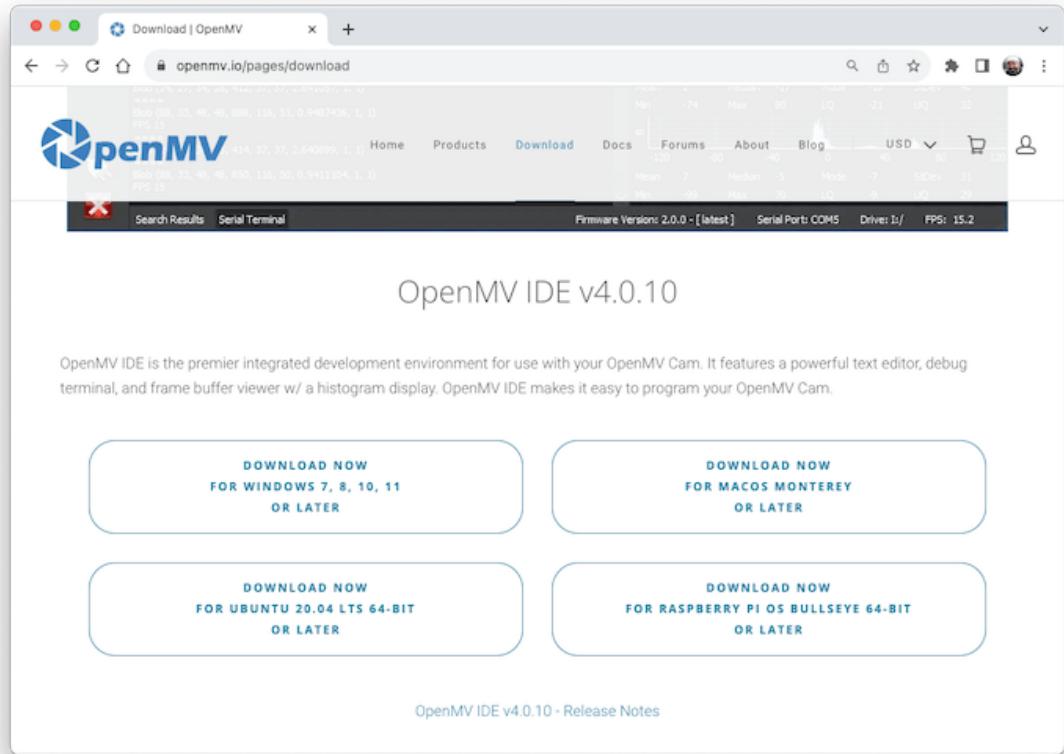
We can also test the camera using, for example, the code provided on Examples > Camera > CameraCaptureRawBytes. We cannot see the image directly, but it is possible to get the raw image data generated by the camera.

Anyway, the best test with the camera is to see a live image. For that, we will use another IDE, the OpenMV.

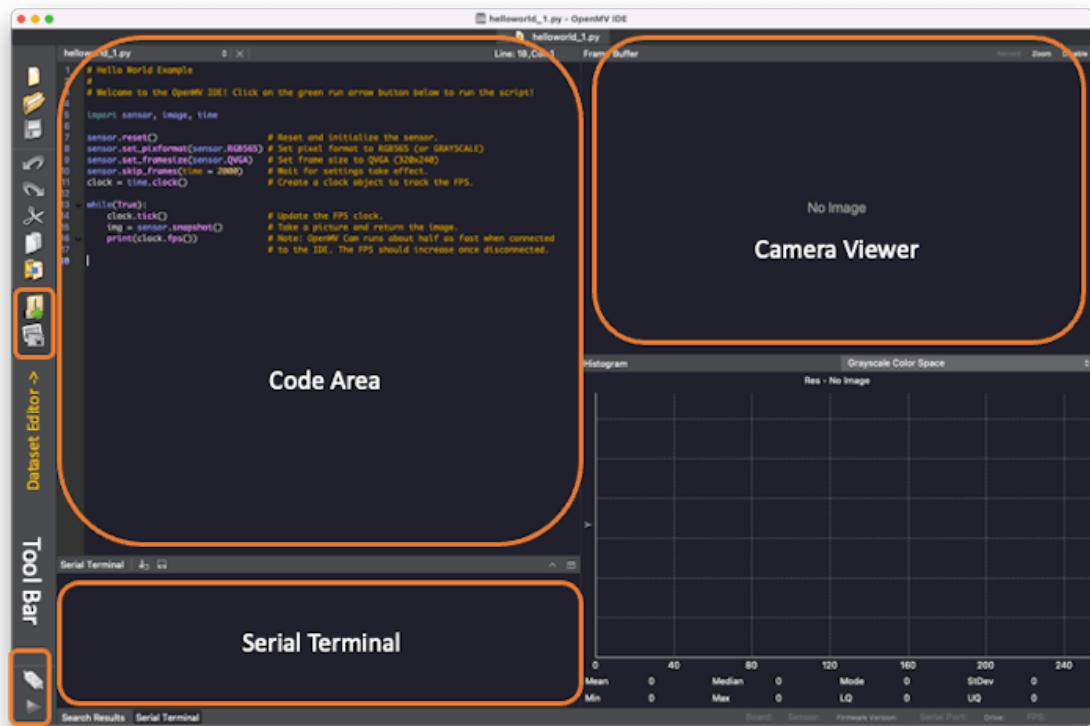
1.4 Installing the OpenMV IDE

OpenMV IDE is the premier integrated development environment with OpenMV Cameras like the one on the Nicla Vision. It features a powerful text editor, debug terminal, and frame buffer viewer with a histogram display. We will use MicroPython to program the camera.

Go to the [OpenMV IDE page](#), download the correct version for your Operating System, and follow the instructions for its installation on your computer.



The IDE should open, defaulting to the `helloworld_1.py` code on its Code Area. If not, you can open it from `Files > Examples > HelloWord > helloworld.py`

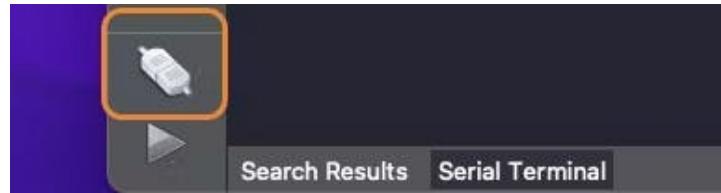


Any messages sent through a serial connection (using `print()` or error messages) will be displayed on the **Serial Terminal** during run time. The image captured by a camera will be displayed in the **Camera Viewer Area** (or Frame Buffer) and in the Histogram area, immediately below the Camera Viewer.

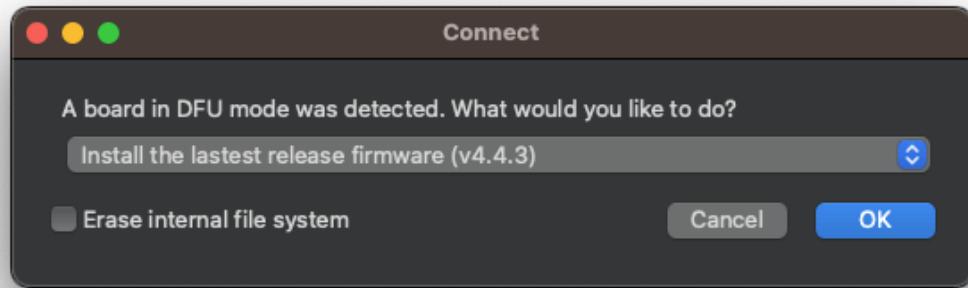
OpenMV IDE is the premier integrated development environment with OpenMV Cameras and the Arduino Pro boards. It features a powerful text editor, debug terminal, and frame buffer viewer with a histogram display. We will use MicroPython to program the Nicla Vision.

Before connecting the Nicla to the OpenMV IDE, ensure you have the latest bootloader version. Go to your Arduino IDE, select the Nicla board, and open the sketch on `Examples > STM_32H747_System STM_32H747_updateBootloader`. Upload the code to your board. The Serial Monitor will guide you.

After updating the bootloader, put the Nicla Vision in bootloader mode by double-pressing the reset button on the board. The built-in green LED will start fading in and out. Now return to the OpenMV IDE and click on the connect icon (Left ToolBar):

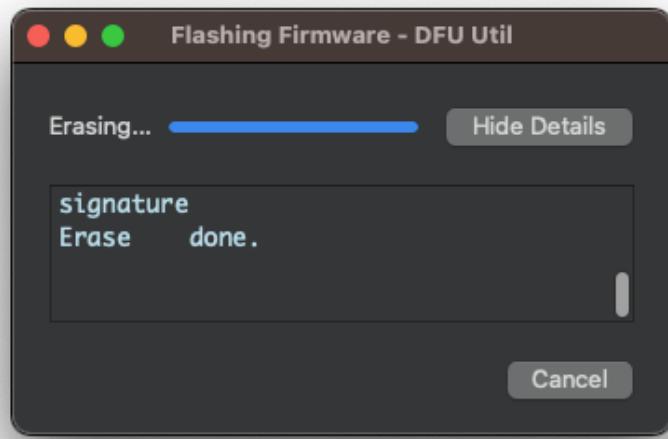


A pop-up will tell you that a board in DFU mode was detected and ask how you would like to proceed. First, select `Install the latest release firmware (vX.Y.z)`. This action will install the latest OpenMV firmware on the Nicla Vision.



You can leave the option `Erase internal file system` unselected and click [OK].

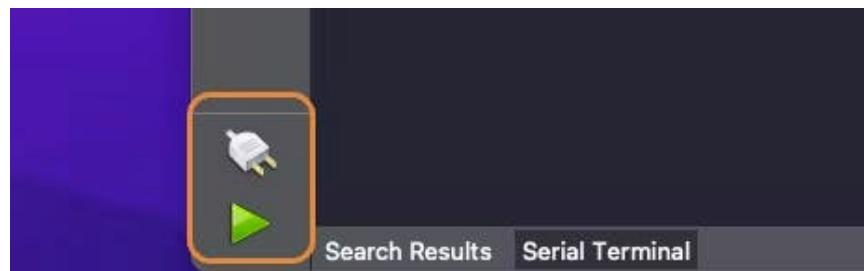
Nicla's green LED will start flashing while the OpenMV firmware is uploaded to the board, and a terminal window will then open, showing the flashing progress.



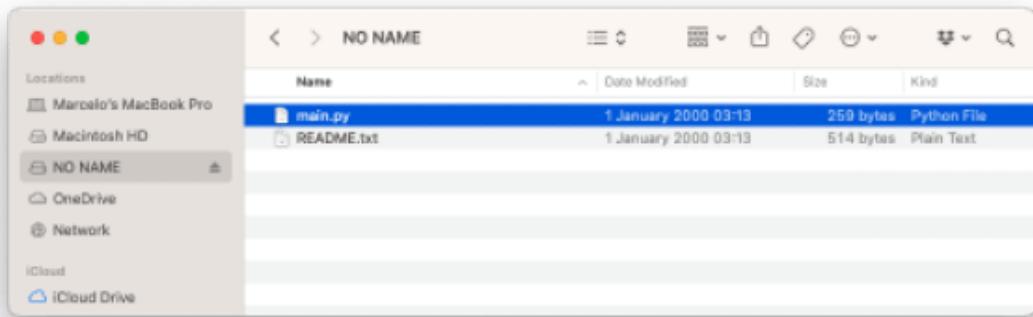
Wait until the green LED stops flashing and fading. When the process ends, you will see a message saying, “DFU firmware update complete!”. Press [OK].



A green play button appears when the Nicla Vison connects to the Tool Bar.



Also, note that a drive named “NO NAME” will appear on your computer.:)



Every time you press the [RESET] button on the board, it automatically executes the *main.py* script stored on it. You can load the *main.py* code on the IDE (File > Open File...).

```
1 # main.py -- put your code here!
2 import pyb, time
3 led = pyb.LED(3)  <-- Blue LED *
4 usb = pyb.USB_VCP()
5 while (usb.isconnected() == False):
6     led.on()
7     time.sleep_ms(150)
8     led.off()
9     time.sleep_ms(100)
10    led.on()
11    time.sleep_ms(150)
12    led.off()
13    time.sleep_ms(600)

* LED(1) : Red
  LED(2) : Green
  LED(3) : Blue
```

This code is the “Blink” code, confirming that the HW is OK.

For testing the camera, let's run *helloworld_1.py*. For that, select the script on File > Examples > HelloWorld > helloworld.py,

When clicking the green play button, the MicroPython script (*helloworld.py*) on the Code Area will be uploaded and run on the Nicla Vision. On-Camera Viewer, you will start to see the video streaming. The

Serial Monitor will show us the FPS (Frames per second), which should be around 14fps.



Here is the [helloworld.py](#) script:

```
# Hello World Example 2
#
# Welcome to the OpenMV IDE! Click on the green run arrow button below to run

import sensor, image, time

sensor.reset()                      # Reset and initialize the sensor.
sensor.set_pixformat(sensor.RGB565)  # Set pixel format to RGB565 (or GRayscale)
sensor.set_framesize(sensor.QVGA)     # Set frame size to QVGA (320x240)
sensor.skip_frames(time = 2000)       # Wait for settings take effect.
clock = time.clock()                # Create a clock object to track the FPS.

while(True):
    clock.tick()                    # Update the FPS clock.
    img = sensor.snapshot()         # Take a picture and return the image.
    print(clock.fps())
```

In [GitHub](#), you can find the Python scripts used here.

The code can be split into two parts:

- **Setup:** Where the libraries are imported, initialized and the variables are defined and initiated.
- **Loop:** (while loop) part of the code that runs continually. The image (*img* variable) is captured (one frame). Each of those frames can be used for inference in Machine Learning Applications.

To interrupt the program execution, press the red [X] button.

Note: OpenMV Cam runs about half as fast when connected to the IDE. The FPS should increase once disconnected.

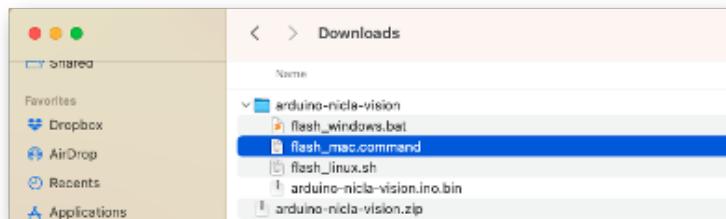
In the [GitHub](#), You can find other Python scripts. Try to test the onboard sensors.

1.5 Connecting the Nicla Vision to Edge Impulse Studio

We will need the Edge Impulse Studio later in other exercises. [Edge Impulse](#) is a leading development platform for machine learning on edge devices.

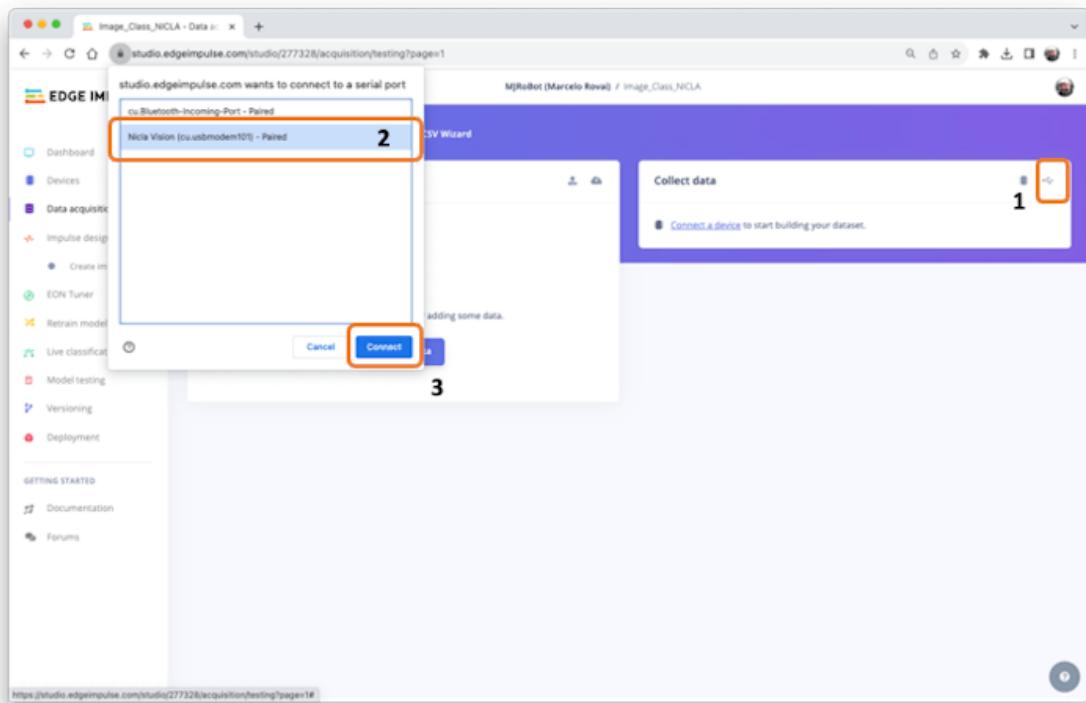
Edge Impulse officially supports the Nicla Vision. So, for starting, please create a new project on the Studio and connect the Nicla to it. For that, follow the steps:

- Download the most updated [EI Firmware](#) and unzip it.
- Open the zip file on your computer and select the uploader corresponding to your OS:

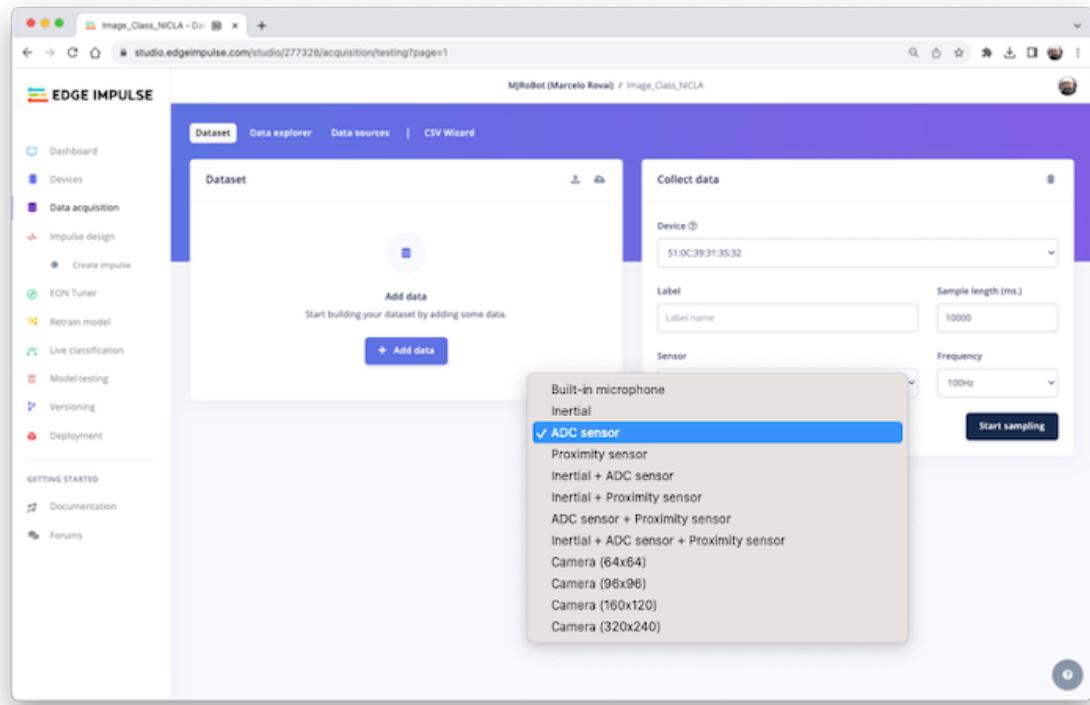


- Put the Nicla-Vision on Boot Mode, pressing the reset button twice.
- Execute the specific batch code for your OS for uploading the binary *arduino-nicla-vision.bin* to your board.

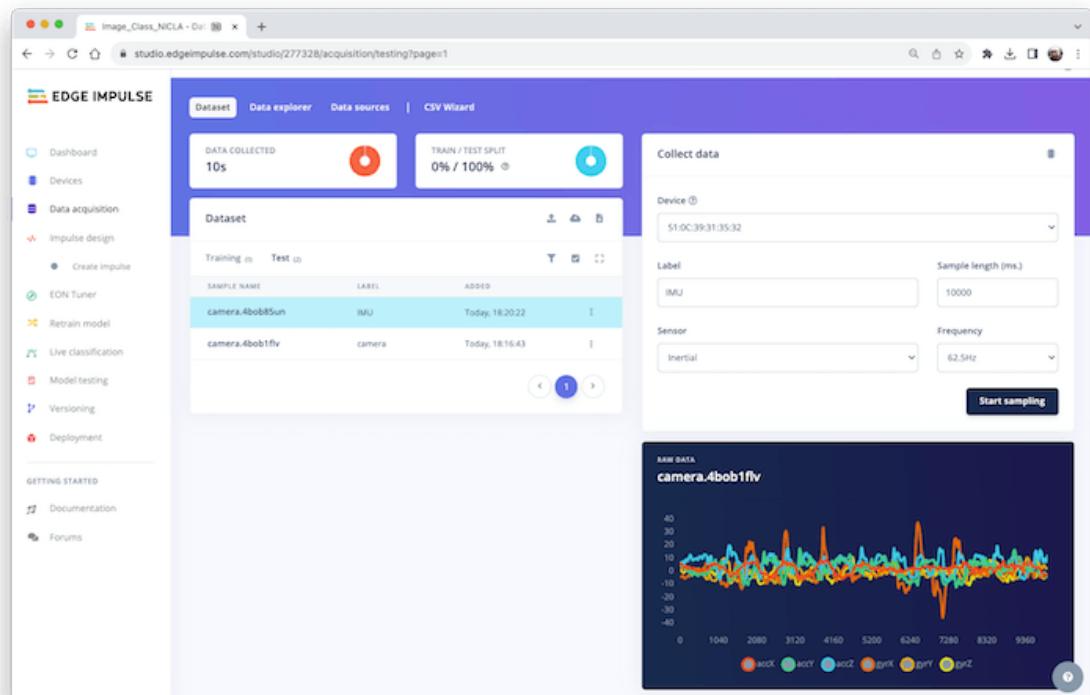
Go to your project on the Studio, and on the Data Acquisition tab, select WebUSB (1). A window will pop up; choose the option that shows that the Nicla is paired (2) and press [Connect] (3).



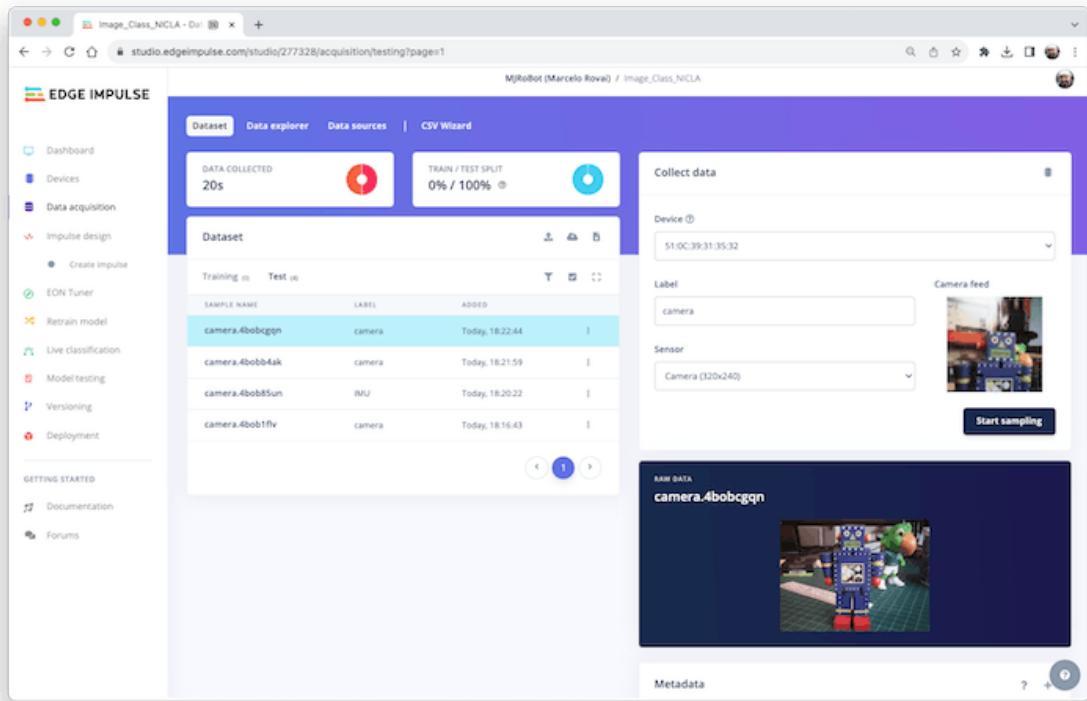
In the *Collect Data* section on the Data Acquisition tab, you can choose which sensor data to pick.



For example. IMU data:



Or Image (Camera):



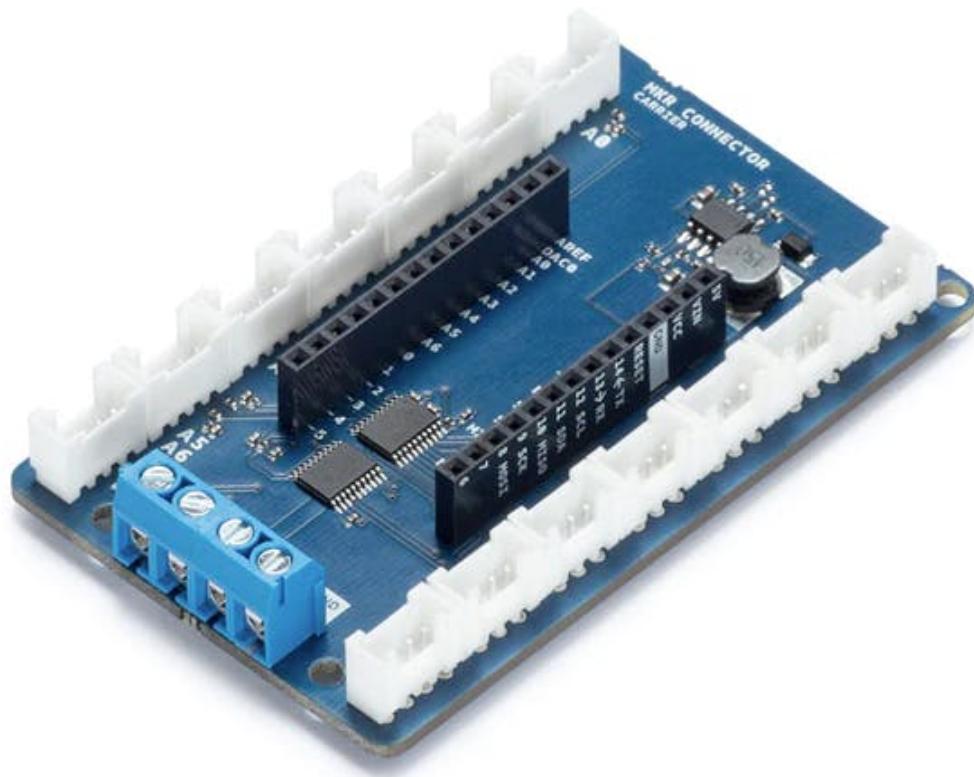
And so on. You can also test an external sensor connected to the ADC (Nicla pin 0) and the other onboard sensors, such as the microphone and the ToF.

1.6 Expanding the Nicla Vision Board (optional)

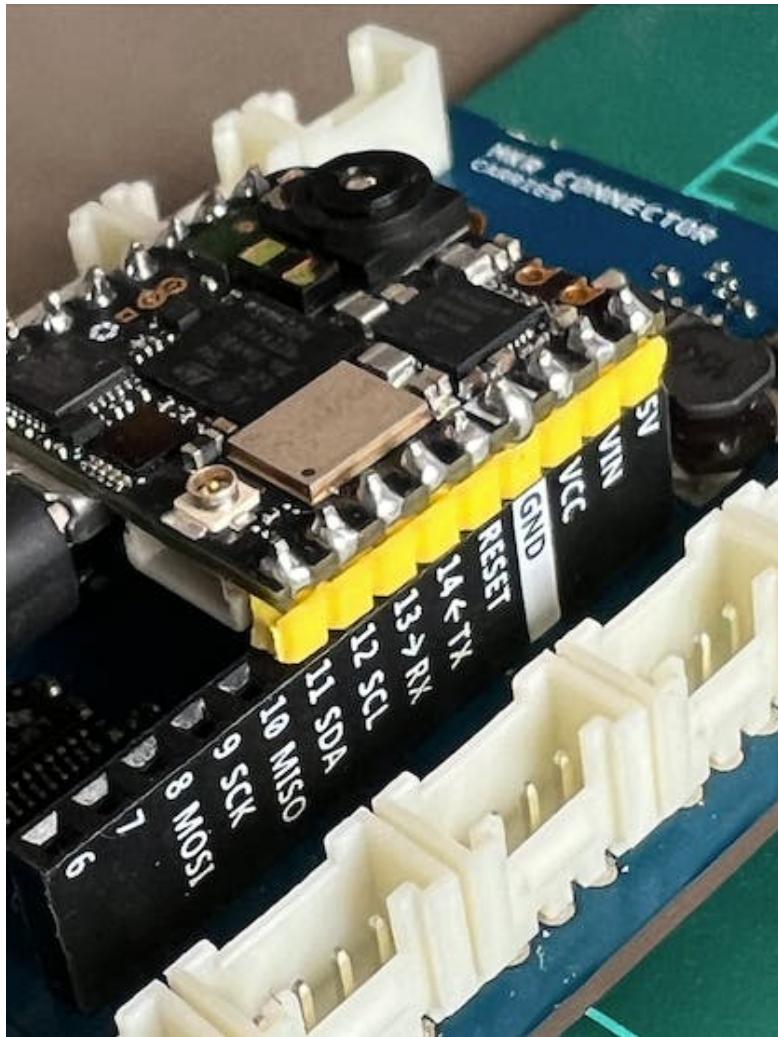
A last item to be explored is that sometimes, during prototyping, it is essential to experiment with external sensors and devices, and an excellent expansion to the Nicla is the [Arduino MKR Connector Carrier \(Grove compatible\)](#).

The shield has 14 Grove connectors: five single analog inputs (A0-A5), one double analog input (A5/A6), five single digital I/Os (D0-D4), one double digital I/O (D5/D6), one I2C (TWI), and one UART (Serial). All connectors are 5V compatible.

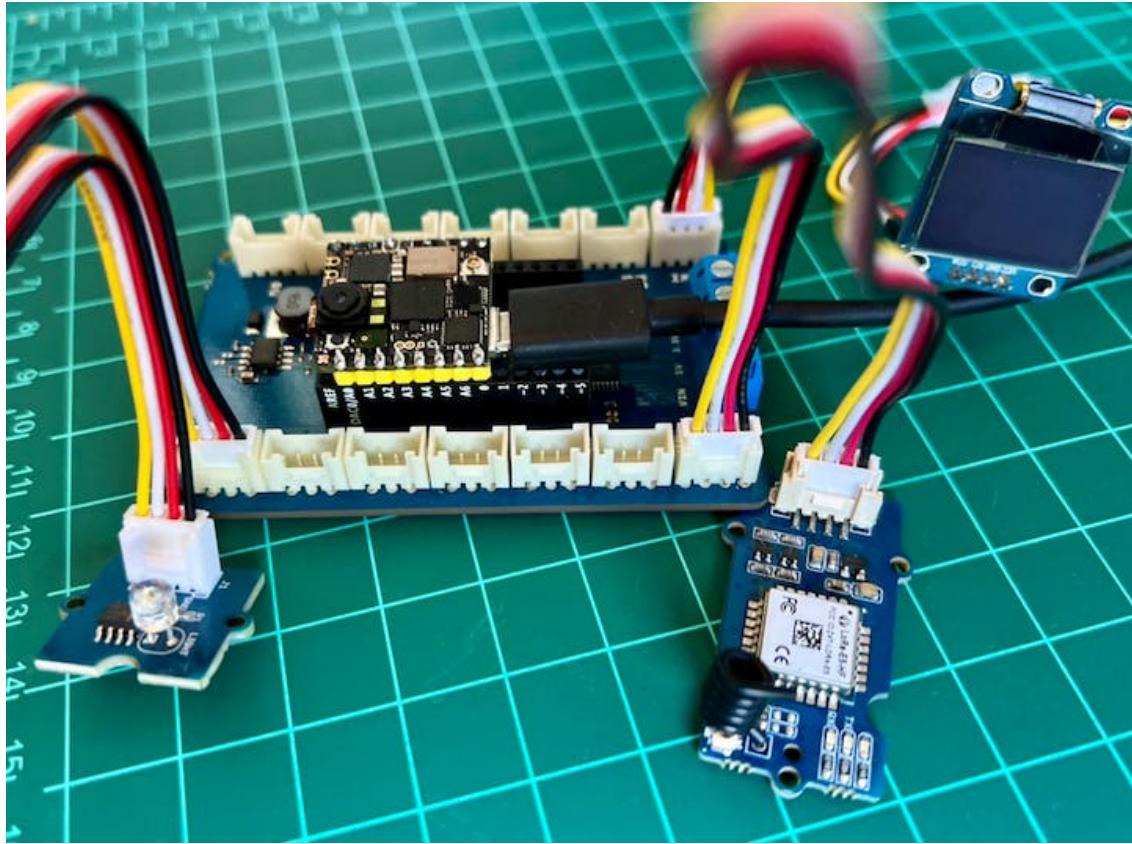
Note that all 17 Nicla Vision pins will be connected to the Shield Groves, but some Grove connections remain disconnected.



This shield is MKR compatible and can be used with the Nicla Vision and Portenta.



For example, suppose that on a TinyML project, you want to send inference results using a LoRaWAN device and add information about local luminosity. Often, with offline operations, a local low-power display such as an OLED is advised. This setup can be seen here:



The [Grove Light Sensor](#) would be connected to one of the single Analog pins (A0/PC4), the [LoRaWAN device](#) to the UART, and the [OLED](#) to the I2C connector.

The Nicla Pins 3 (Tx) and 4 (Rx) are connected with the Serial Shield connector. The UART communication is used with the LoRaWan device. Here is a simple code to use the UART:

```
# UART Test - By: marcelo_rovai - Sat Sep 23 2023

import time
from pyb import UART
from pyb import LED

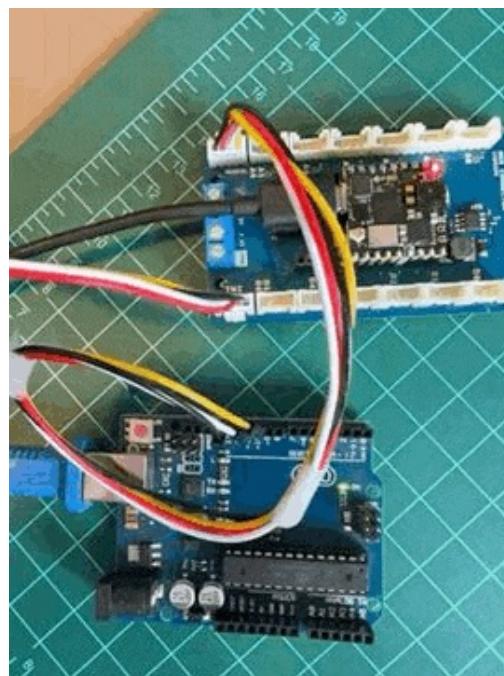
redLED = LED(1) # built-in red LED

# Init UART object.
# Nicla Vision's UART (TX/RX pins) is on "LP1"
uart = UART("LP1", 9600)

while(True):
    uart.write("Hello World!\r\n")
```

```
redLED.toggle()  
time.sleep_ms(1000)
```

To verify that the UART is working, you should, for example, connect another device as the Arduino UNO, displaying “Hello Word” on the Serial Monitor. Here is the [code](#).



Below is the *Hello World* code to be used with the I2C OLED. The MicroPython SSD1306 OLED driver (ssd1306.py), created by Adafruit, should also be uploaded to the Nicla (the ssd1306.py script can be found in [GitHub](#)).

```
# Nicla_OLED_Hello_World - By: marcelo_rovai - Sat Sep 30 2023  
  
#Save on device: MicroPython SSD1306 OLED driver, I2C and SPI interfaces creat  
import ssd1306  
  
from machine import I2C  
i2c = I2C(1)  
  
oled_width = 128  
oled_height = 64  
oled = ssd1306.SSD1306_I2C(oled_width, oled_height, i2c)  
  
oled.text('Hello, World', 10, 10)  
oled.show()
```

Finally, here is a simple script to read the ADC value on pin “PC4” (Nicla pin A0):

```
# Light Sensor (A0) - By: marcelo_rovai - Wed Oct 4 2023

import pyb
from time import sleep

adc = pyb.ADC(pyb.Pin("PC4"))      # create an analog object from a pin
val = adc.read()                  # read an analog value

while (True):

    val = adc.read()
    print ("Light={}".format (val))
    sleep (1)
```

The ADC can be used for other sensor variables, such as [Temperature](#).

Note that the above scripts ([downloaded from Github](#)) introduce only how to connect external devices with the Nicla Vision board using MicroPython.

1.7 Conclusion

The Arduino Nicla Vision is an excellent *tiny device* for industrial and professional uses! However, it is powerful, trustworthy, low power, and has suitable sensors for the most common embedded machine learning applications such as vision, movement, sensor fusion, and sound.

On the [GitHub repository](#), you will find the last version of all the codes used or commented on in this hands-on exercise.

2 CV on Nicla Vision



2.1 Introduction

As we initiate our studies into embedded machine learning or tinyML, it's impossible to overlook the transformative impact of Computer Vision (CV) and Artificial Intelligence (AI) in our lives. These two intertwined

disciplines redefine what machines can perceive and accomplish, from autonomous vehicles and robotics to healthcare and surveillance.

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



In the “bullseye” of the Radar is the *Edge Computer Vision*, and when we talk about Machine Learning (ML) applied to vision, the first thing that comes to mind is **Image Classification**, a kind of ML “Hello World”!

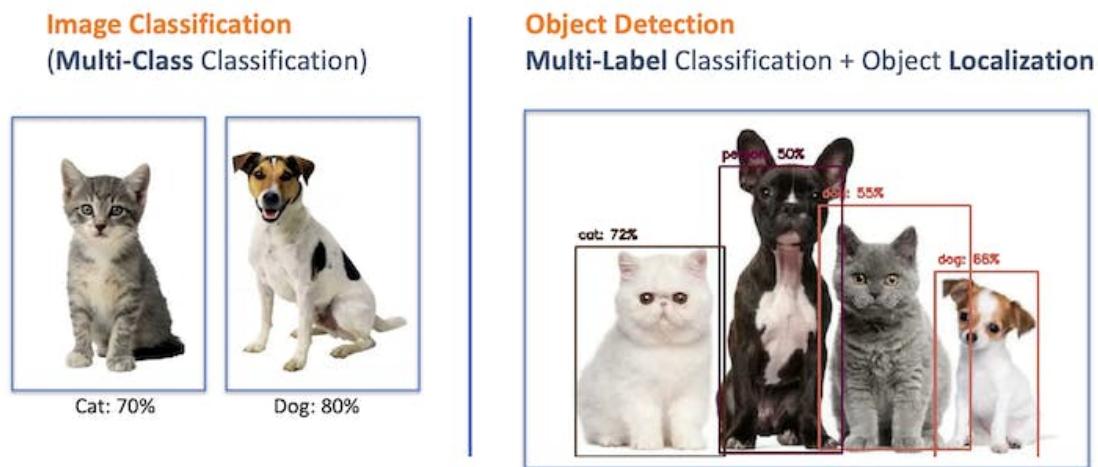
This exercise will explore a computer vision project utilizing Convolutional Neural Networks (CNNs) for real-time image classification. Leveraging TensorFlow’s robust ecosystem, we’ll implement a pre-trained MobileNet model and adapt it for edge deployment. The focus will be on optimizing the model to run efficiently on resource-constrained hardware without sacrificing accuracy.

We'll employ techniques like quantization and pruning to reduce the computational load. By the end of this tutorial, you'll have a working prototype capable of classifying images in real-time, all running on a low-power embedded system based on the Arduino Nicla Vision board.

2.2 Computer Vision

At its core, computer vision aims to enable machines to interpret and make decisions based on visual data from the world, essentially mimicking the capability of the human optical system. Conversely, AI is a broader field encompassing machine learning, natural language processing, and robotics, among other technologies. When you bring AI algorithms into computer vision projects, you supercharge the system's ability to understand, interpret, and react to visual stimuli.

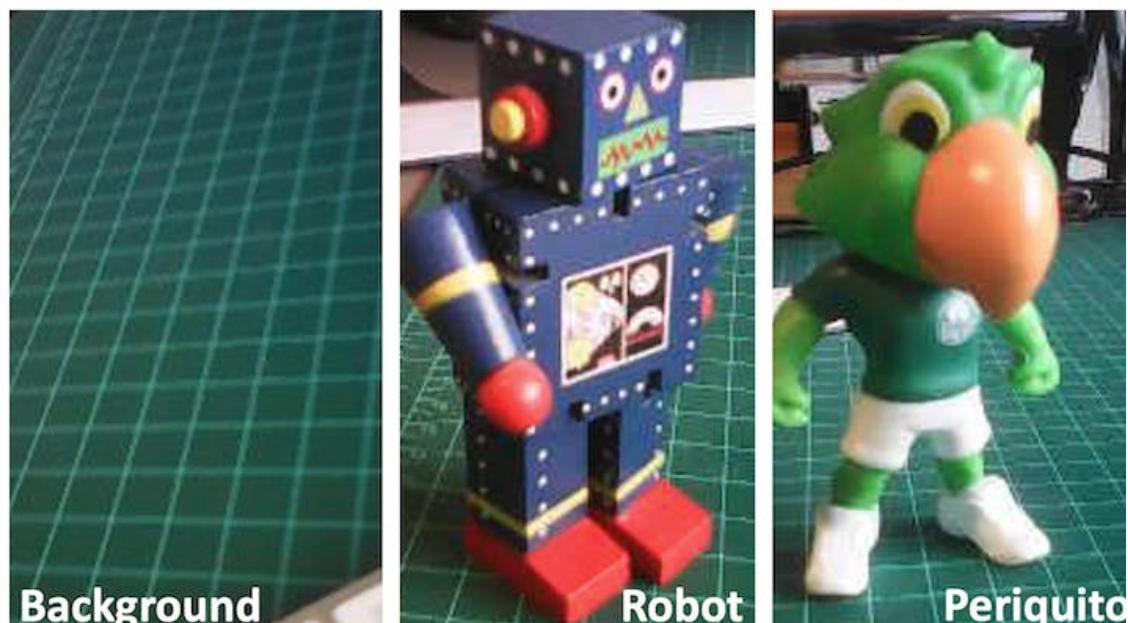
When discussing Computer Vision projects applied to embedded devices, the most common applications that come to mind are *Image Classification* and *Object Detection*.



Both models can be implemented on tiny devices like the Arduino Nicla Vision and used on real projects. In this chapter, we will cover Image Classification.

2.3 Image Classification Project Goal

The first step in any ML project is to define the goal. In this case, it is to detect and classify two specific objects present in one image. For this project, we will use two small toys: a *robot* and a small Brazilian parrot (named *Periquito*). Also, we will collect images of a *background* where those two objects are absent.

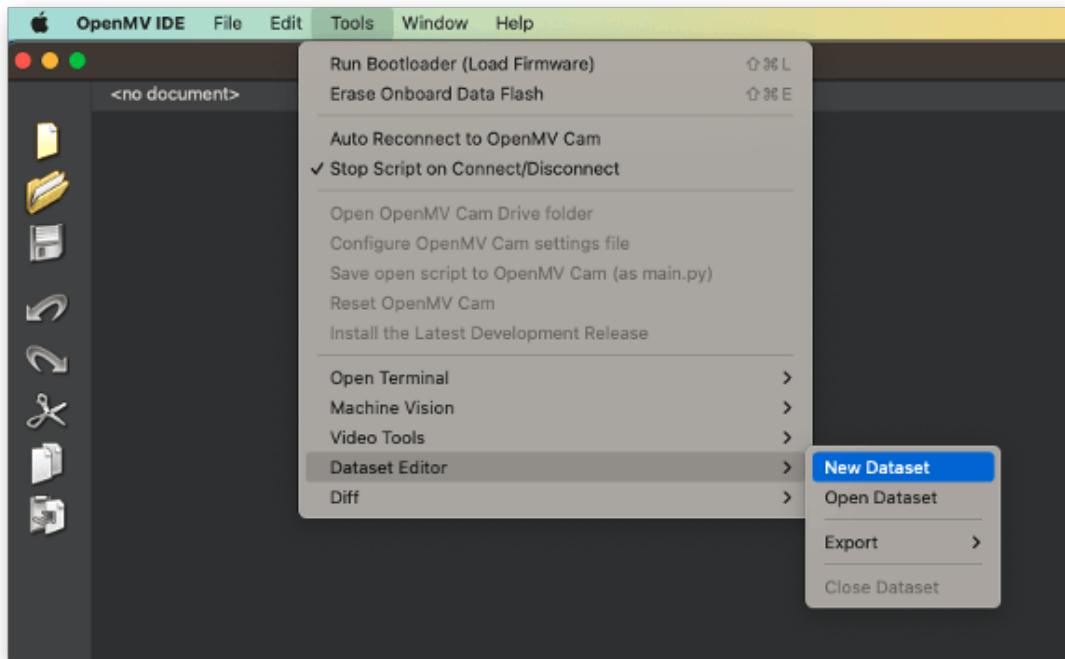


2.4 Data Collection

Once you have defined your Machine Learning project goal, the next and most crucial step is the dataset collection. You can use the Edge Impulse Studio, the OpenMV IDE we installed, or even your phone for the image capture. Here, we will use the OpenMV IDE for that.

2.4.1 Collecting Dataset with OpenMV IDE

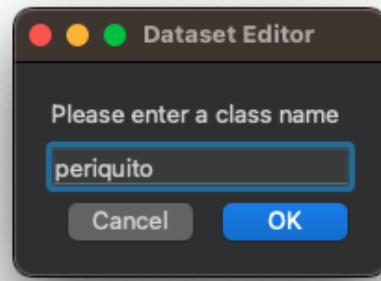
First, create in your computer a folder where your data will be saved, for example, “data.” Next, on the OpenMV IDE, go to Tools > Dataset Editor and select New Dataset to start the dataset collection:



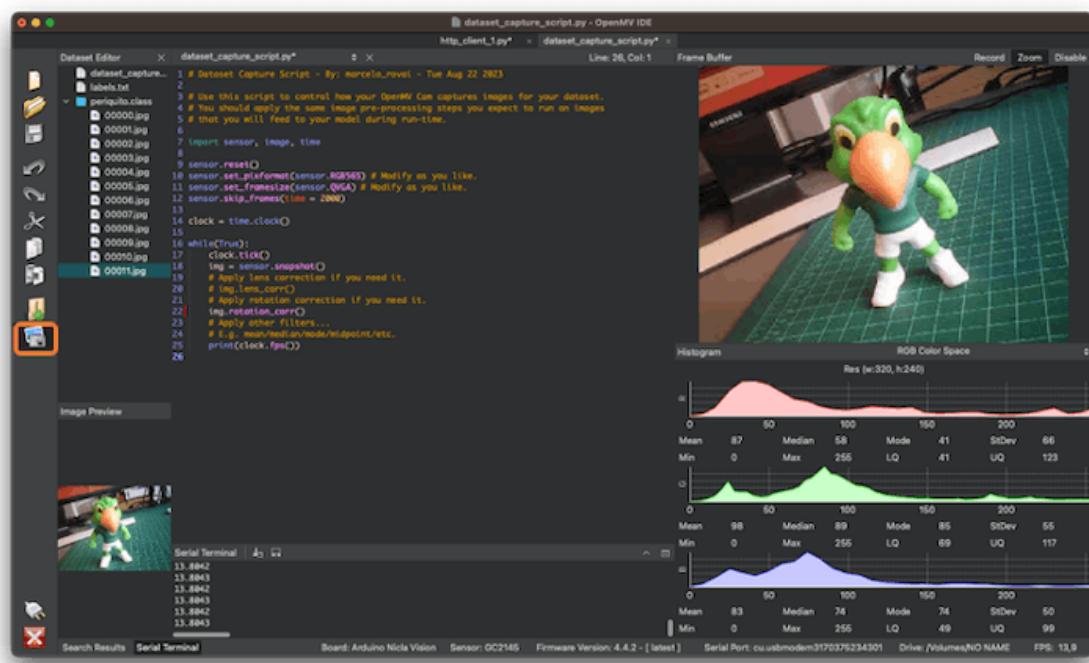
The IDE will ask you to open the file where your data will be saved and choose the “data” folder that was created. Note that new icons will appear on the Left panel.



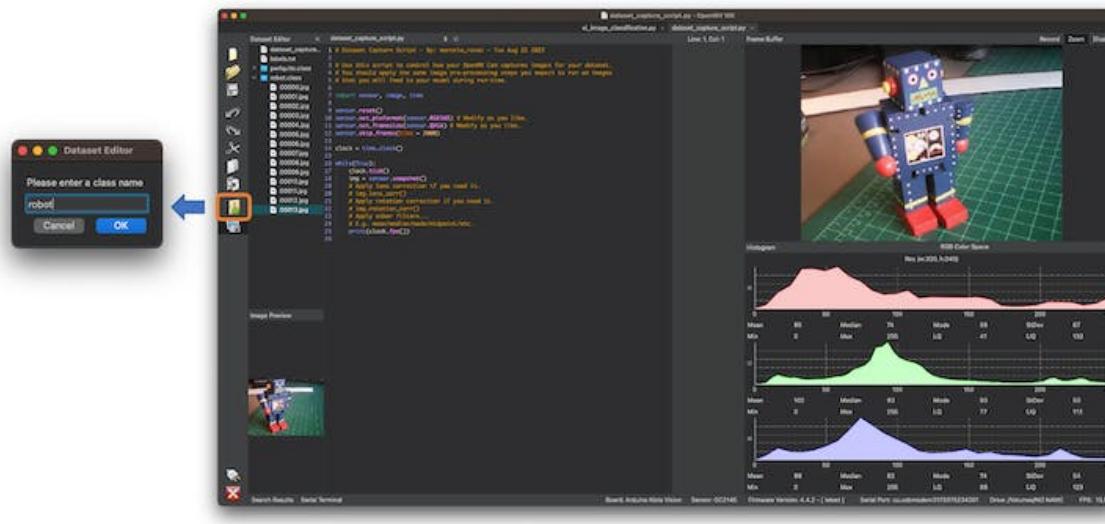
Using the upper icon (1), enter with the first class name, for example, “periquito”:



Running the `dataset_capture_script.py` and clicking on the camera icon (2), will start capturing images:



Repeat the same procedure with the other classes

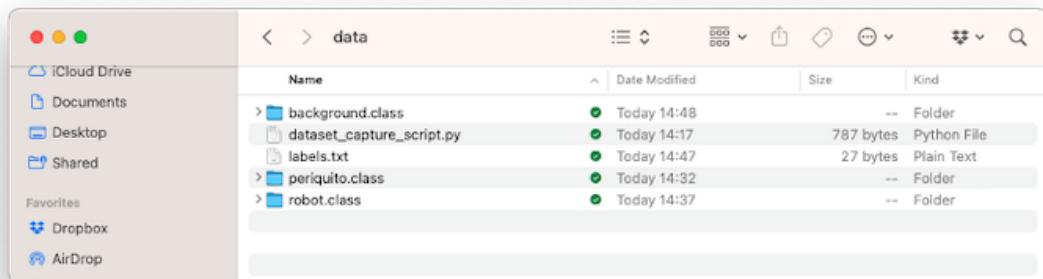


We suggest around 60 images from each category. Try to capture different angles, backgrounds, and light conditions.

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

After capturing your dataset, close the Dataset Editor Tool on the Tools > Dataset Editor.

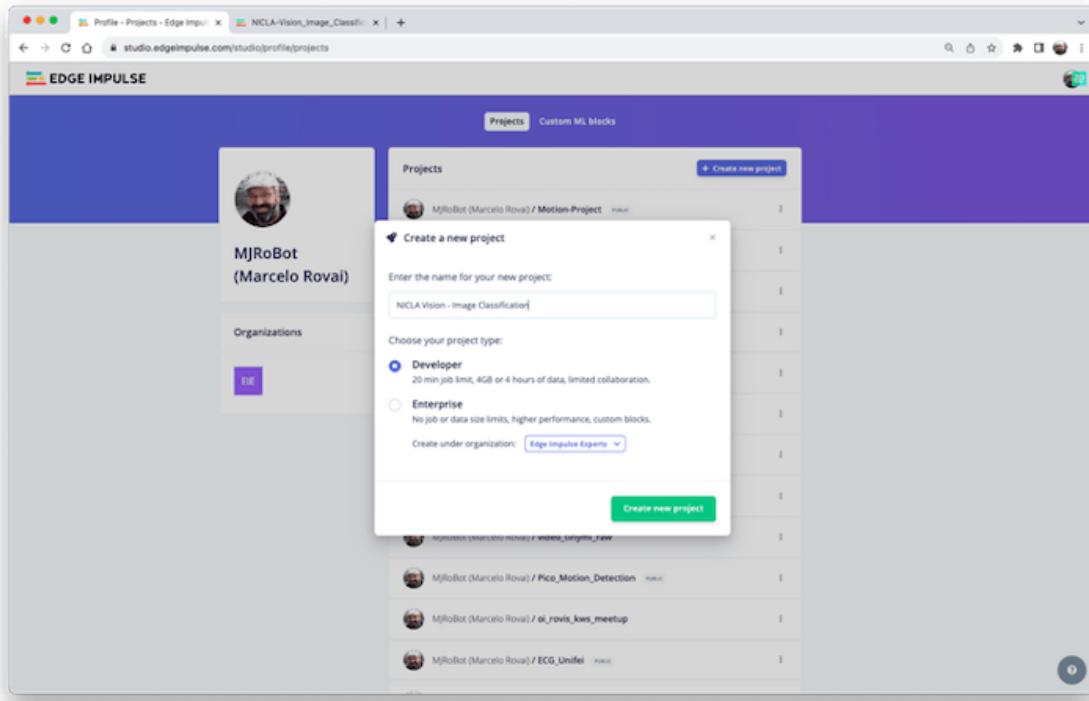
On your computer, you will end with a dataset that contains three classes: *periquito*, *robot*, and *background*.



You should return to *Edge Impulse Studio* and upload the dataset to your project.

2.5 Training the model with Edge Impulse Studio

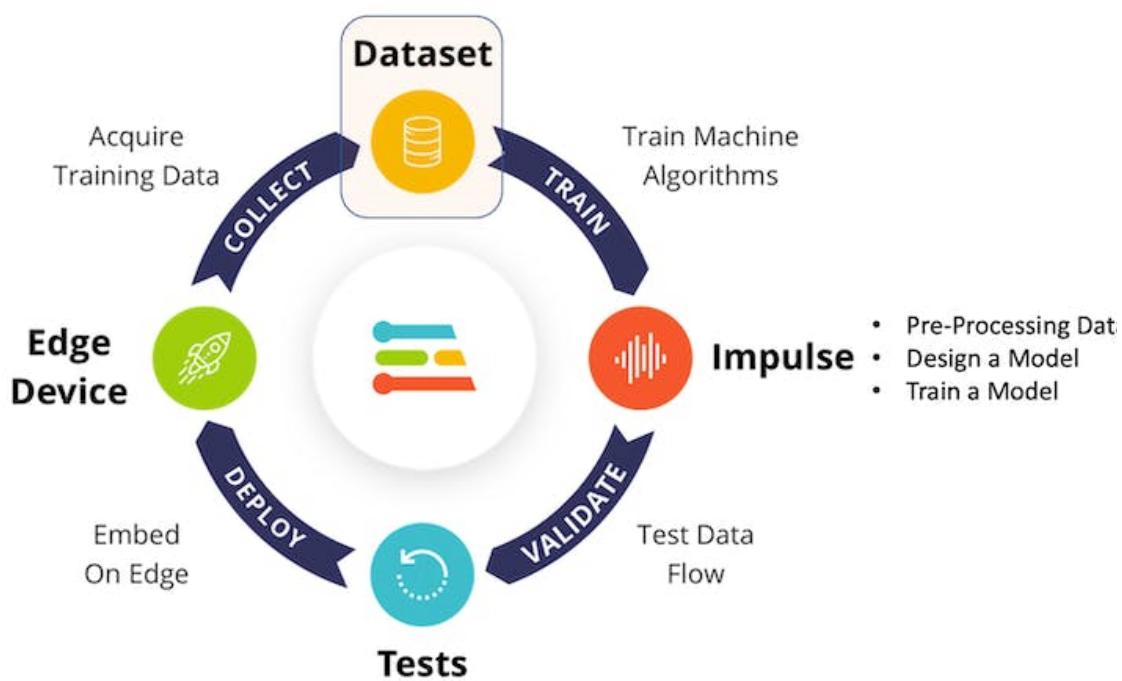
We will use the Edge Impulse Studio for training our model. Enter your account credentials and create a new project:



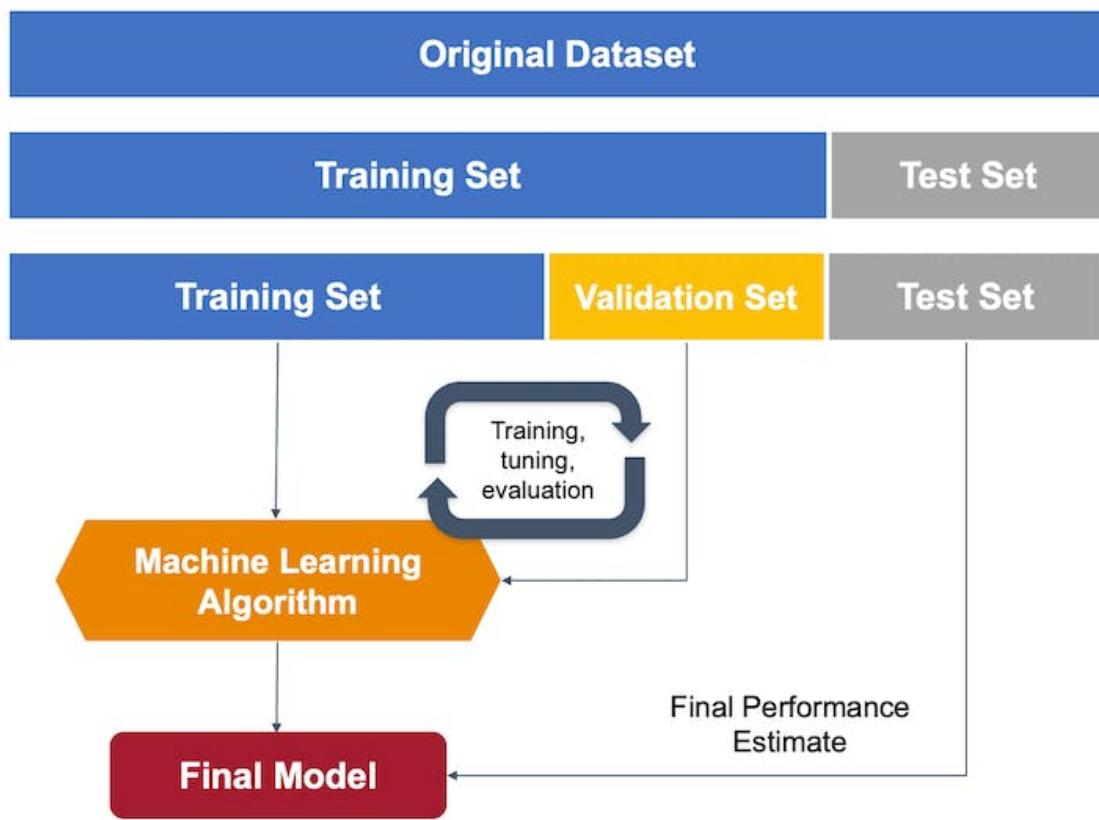
Here, you can clone a similar project: [NICLA-Vision_Image_Classification](#).

2.6 Dataset

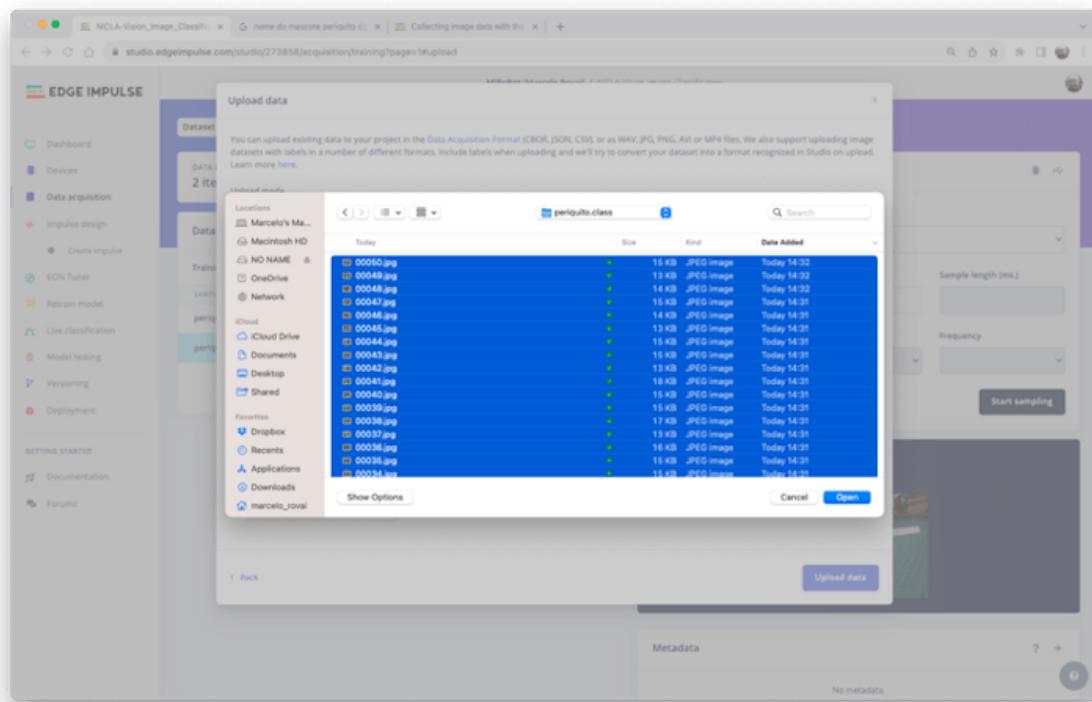
Using the EI Studio (or *Studio*), we will go over four main steps to have our model ready for use on the Nicla Vision board: Dataset, Impulse, Tests, and Deploy (on the Edge Device, in this case, the NiclaV).



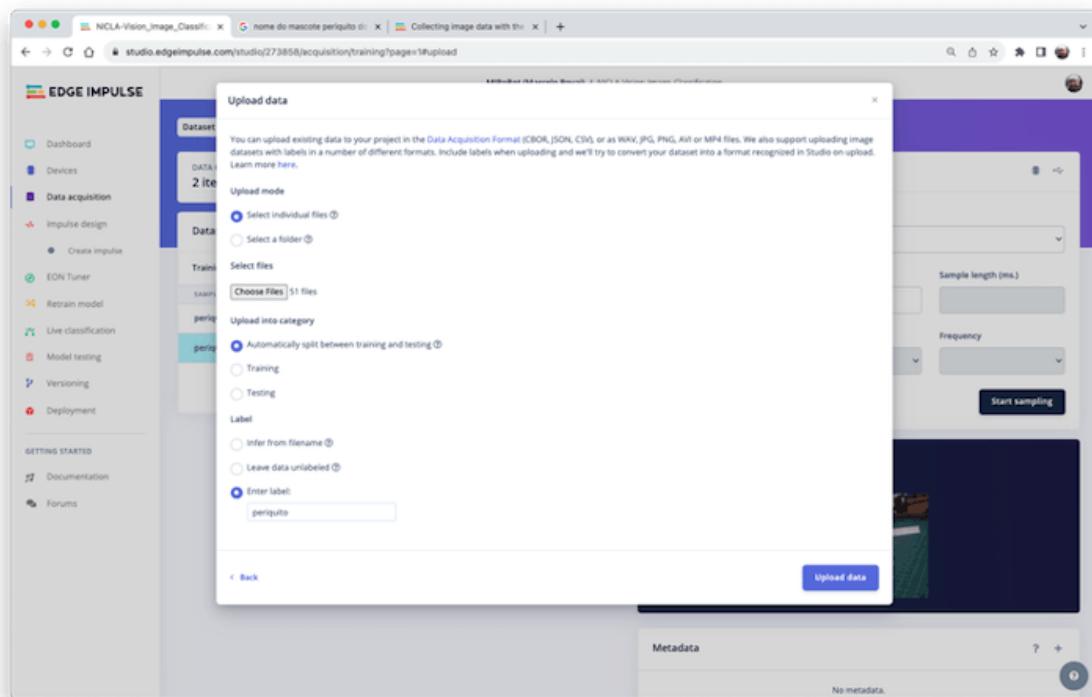
Regarding the Dataset, it is essential to point out that our Original Dataset, captured with the OpenMV IDE, will be split into *Training*, *Validation*, and *Test*. The Test Set will be divided from the beginning, and a part will reserved to be used only in the Test phase after training. The Validation Set will be used during training.



On Studio, go to the Data acquisition tab, and on the UPLOAD DATA section, upload the chosen categories files from your computer:



Leave to the Studio the splitting of the original dataset into *train and test* and choose the label about that specific data:



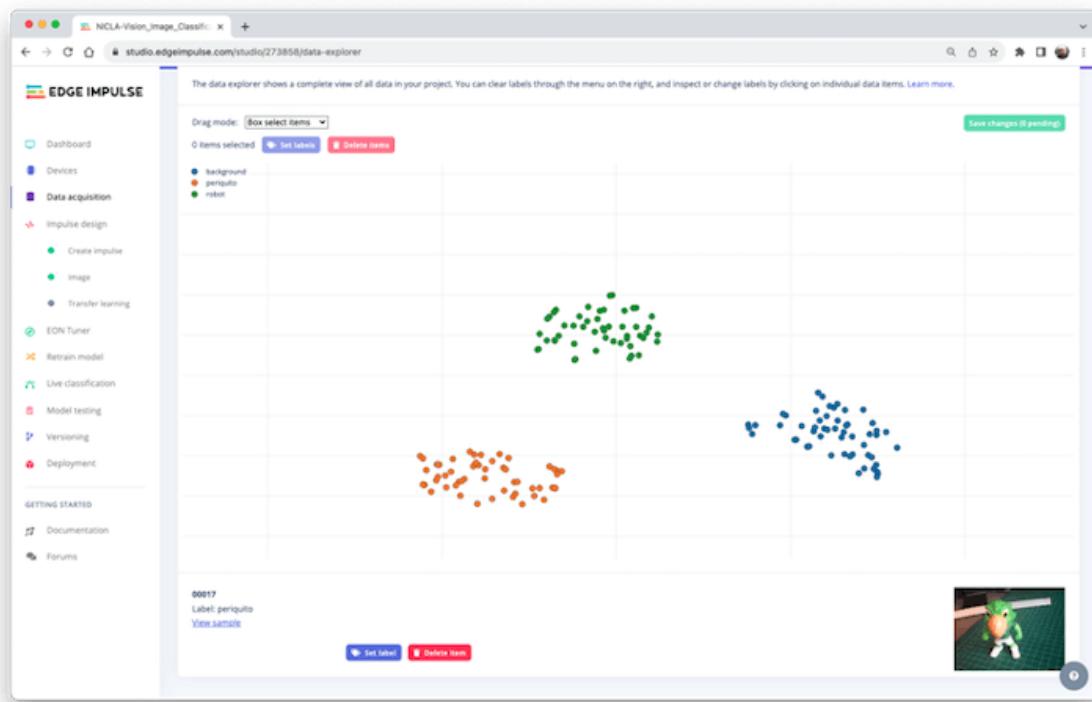
Repeat the procedure for all three classes. At the end, you should see your “raw data” in the Studio:

The screenshot shows the Edge Impulse Studio interface. On the left, a sidebar lists project options like Dashboard, Devices, Data acquisition, and Model testing. The main area is titled 'Dataset' and shows a summary: 'DATA COLLECTED 158 items' and 'TRAIN / TEST SPLIT 78% / 22%'. Below this is a table of data samples:

SAMPLE NAME	LABEL	CREATED
00051	background	Today, 14:58:43
00043	background	Today, 14:58:43
00036	background	Today, 14:58:43
00038	background	Today, 14:58:43
00034	background	Today, 14:58:43
00046	background	Today, 14:58:43
00001	robot	Today, 14:58:12
00003	robot	Today, 14:58:12
00002	robot	Today, 14:58:12
00004	robot	Today, 14:58:12
00005	robot	Today, 14:58:12
00006	robot	Today, 14:58:12

To the right, a 'Collect data' panel includes fields for 'Label' (set to 'periquito') and 'Sensor' (set to 'camera'). A 'Start sampling' button is present. Below this is a preview window showing a small image of a blue robot labeled '00001'. The bottom section is labeled 'Metadata' with a note 'No metadata.'

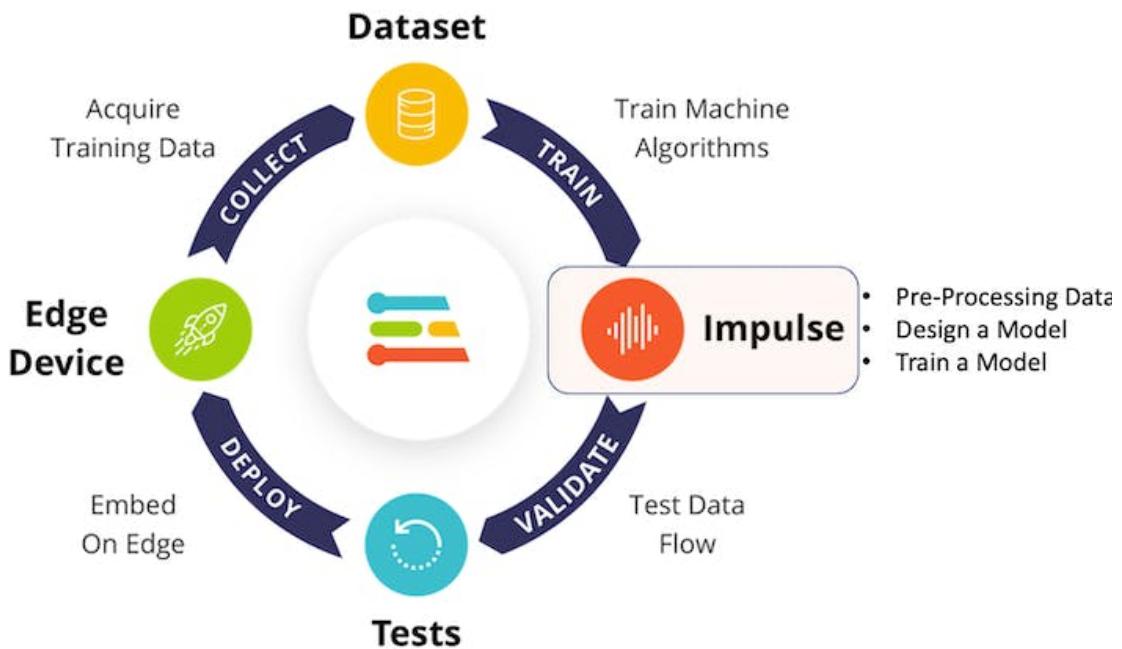
The Studio allows you to explore your data, showing a complete view of all the data in your project. You can clear, inspect, or change labels by clicking on individual data items. In our case, a very simple project, the data seems OK.



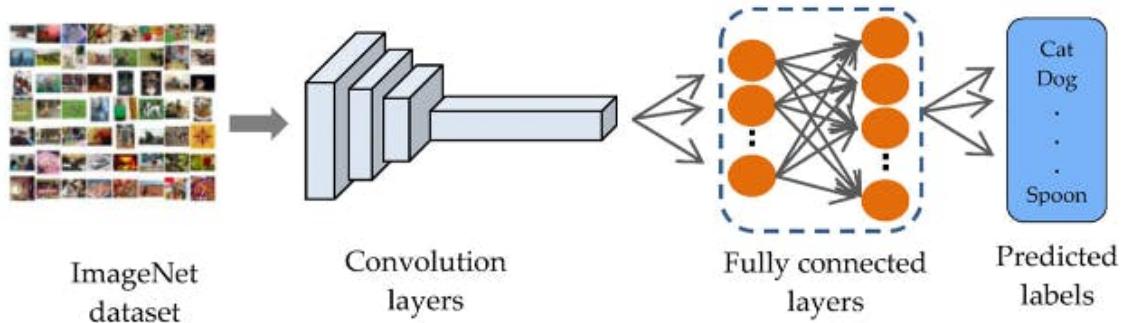
2.7 The Impulse Design

In this phase, we should define how to:

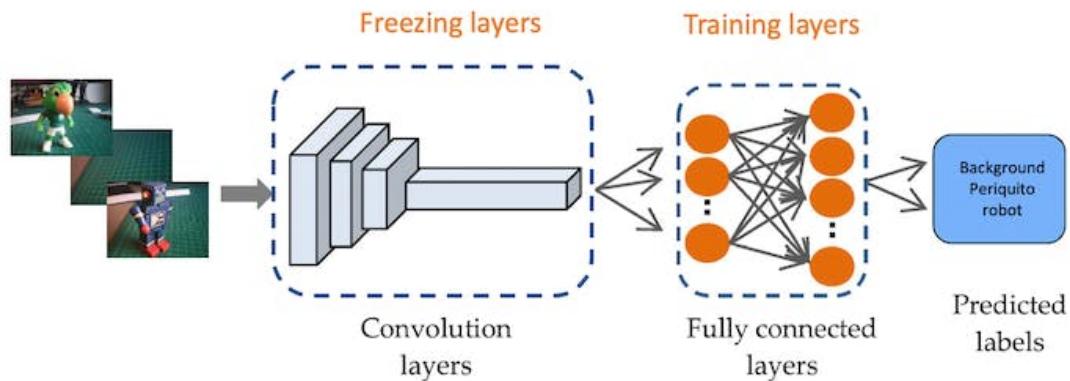
- Pre-process our data, which consists of resizing the individual images and determining the color depth to use (be it RGB or Grayscale) and
- Specify a Model, in this case, it will be the Transfer Learning (Images) to fine-tune a pre-trained MobileNet V2 image classification model on our data. This method performs well even with relatively small image datasets (around 150 images in our case).



Transfer Learning with MobileNet offers a streamlined approach to model training, which is especially beneficial for resource-constrained environments and projects with limited labeled data. MobileNet, known for its lightweight architecture, is a pre-trained model that has already learned valuable features from a large dataset (ImageNet).

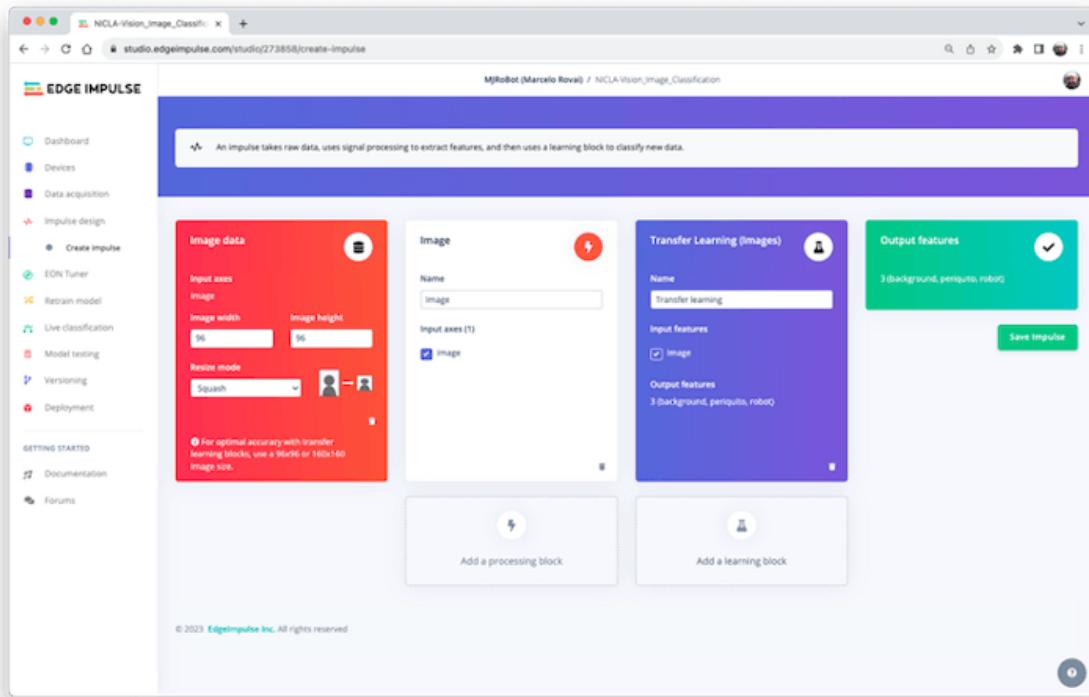


By leveraging these learned features, you can train a new model for your specific task with fewer data and computational resources and yet achieve competitive accuracy.



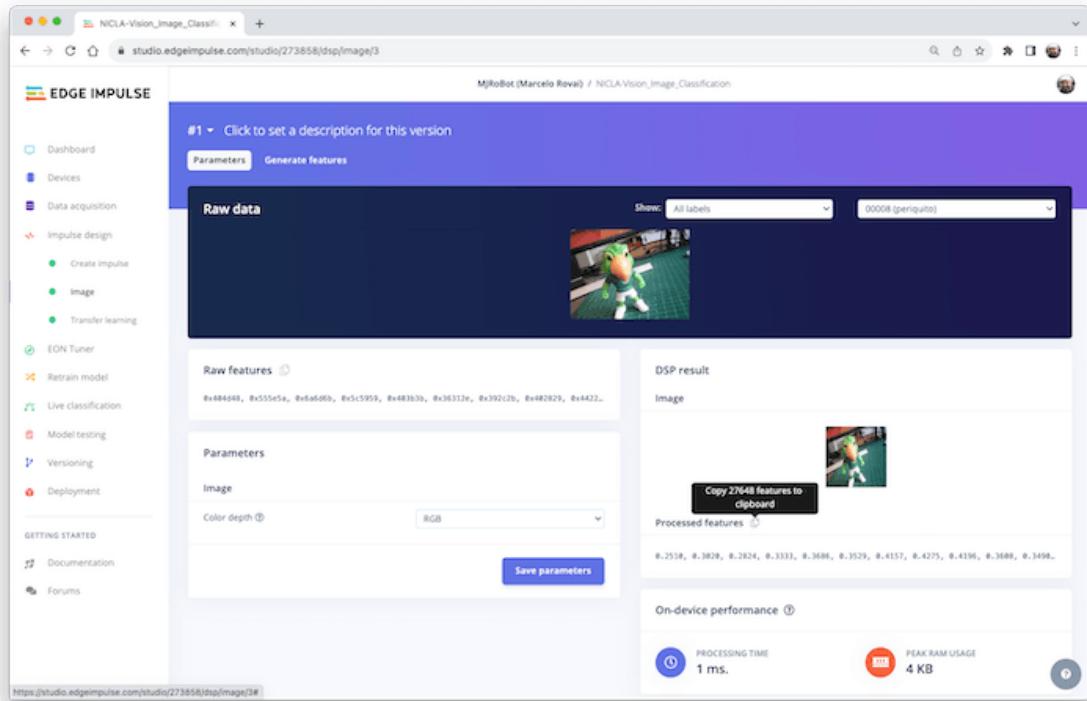
This approach significantly reduces training time and computational cost, making it ideal for quick prototyping and deployment on embedded devices where efficiency is paramount.

Go to the Impulse Design Tab and create the *impulse*, defining an image size of 96x96 and squashing them (squared form, without cropping). Select Image and Transfer Learning blocks. Save the Impulse.

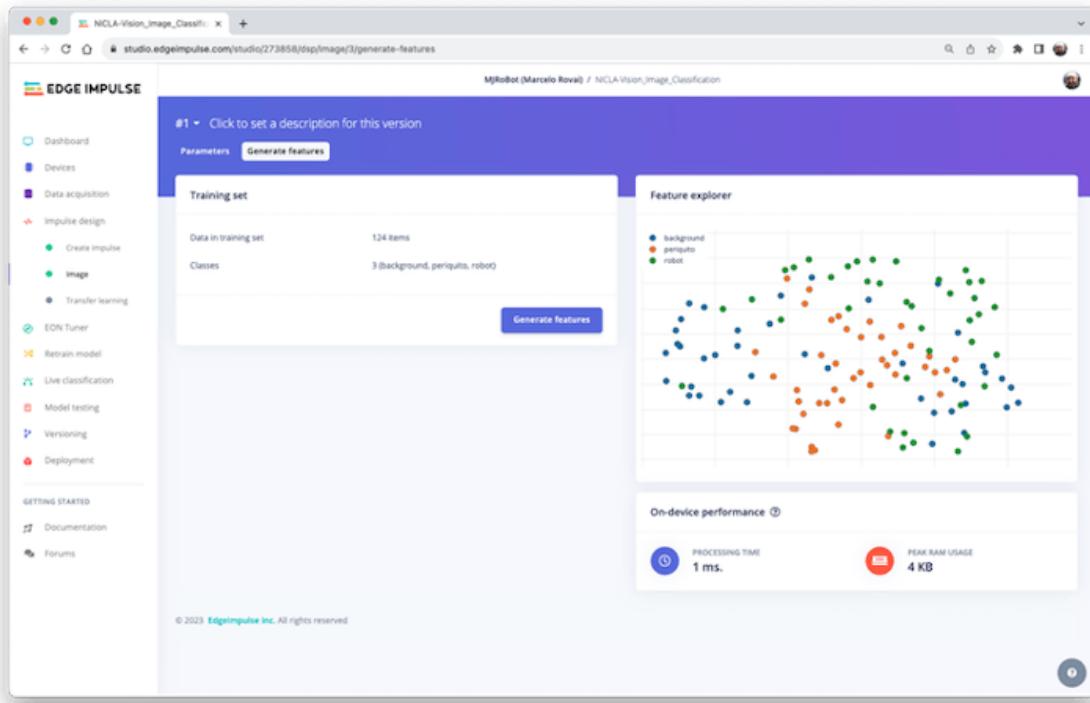


2.7.1 Image Pre-Processing

All the input QVGA/RGB565 images will be converted to 27,640 features (96x96x3).



Press [Save parameters] and Generate all features:



2.7.2 Model Design

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. In 2018, Google launched [MobileNetV2: Inverted Residuals and Linear Bottlenecks](#).

MobileNet V1 and MobileNet V2 aim at mobile efficiency and embedded vision applications but differ in architectural complexity and performance. While both use depthwise separable convolutions to reduce the computational cost, MobileNet V2 introduces Inverted Residual Blocks and Linear Bottlenecks to enhance performance. These new features allow V2 to capture more complex features using fewer parameters, making it computationally more efficient and generally more accurate than its predecessor. Additionally, V2 employs a non-linear activation in the intermediate expansion layer. It still uses a linear activation for the bottleneck layer, a design choice found to preserve important information.

through the network. MobileNet V2 offers an optimized architecture for higher accuracy and efficiency and will be used in this project.

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 even smaller and faster. MobileNets 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 that of thinning a network uniformly at each layer.

Edge Impulse Studio can use both MobileNetV1 (96x96 images) and V2 (96x96 or 160x160 images), with several different α values (from 0.05 to 1.0). For example, you will get the highest accuracy with V2, 160x160 images, and $\alpha=1.0$. Of course, there is a trade-off. The higher the accuracy, the more memory (around 1.3MB RAM and 2.6MB ROM) will be needed to run the model, implying more latency. The smaller footprint will be obtained at the other extreme with MobileNetV1 and $\alpha=0.10$ (around 53.2K RAM and 101K ROM).

MobileNetV1 96x96 0.1

Uses around 53.2K RAM and 101K ROM with default settings and optimizations. Works best with 96x96 input size. Supports both RGB and grayscale.

Model

MobileNetV2 96x96 0.35

Uses around 296.8K RAM and 575.2K ROM with default settings and optimizations. Works best with 96x96 input size. Supports both RGB and grayscale.

Image Size

MobileNetV2 96x96 0.1

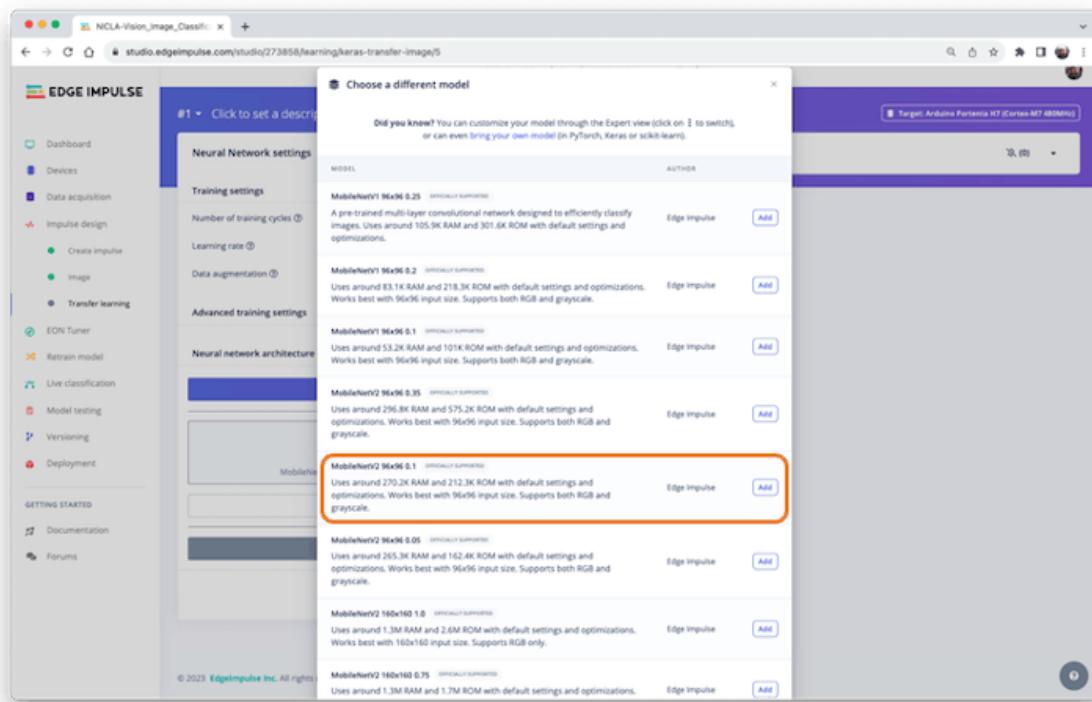
Uses around 270.2K RAM and 212.3K ROM with default settings and optimizations. Works best with 96x96 input size. Supports both RGB and grayscale.

Alpha

MobileNetV2 96x96 0.05

Uses around 265.3K RAM and 162.4K ROM with default settings and optimizations. Works best with 96x96 input size. Supports both RGB and grayscale.

We will use **MobileNetV2 96x96 0.1** for this project, with an estimated memory cost of 265.3 KB in RAM. This model should be OK for the Nicla Vision with 1MB of SRAM. On the Transfer Learning Tab, select this model:



2.8 Model Training

Another valuable technique to be used with Deep Learning is **Data Augmentation**. Data augmentation is a method to improve the accuracy of machine learning models by 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).

Looking under the hood, 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)

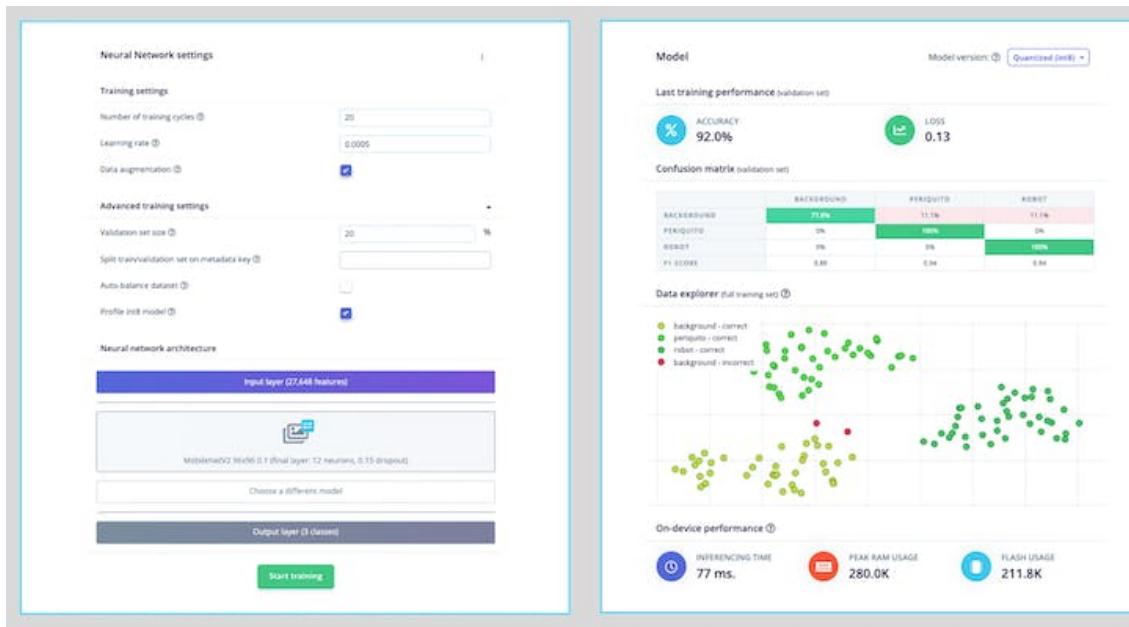
# Vary the brightness of the image
image = tf.image.random_brightness(image, max_delta=0.2)

return image, label

```

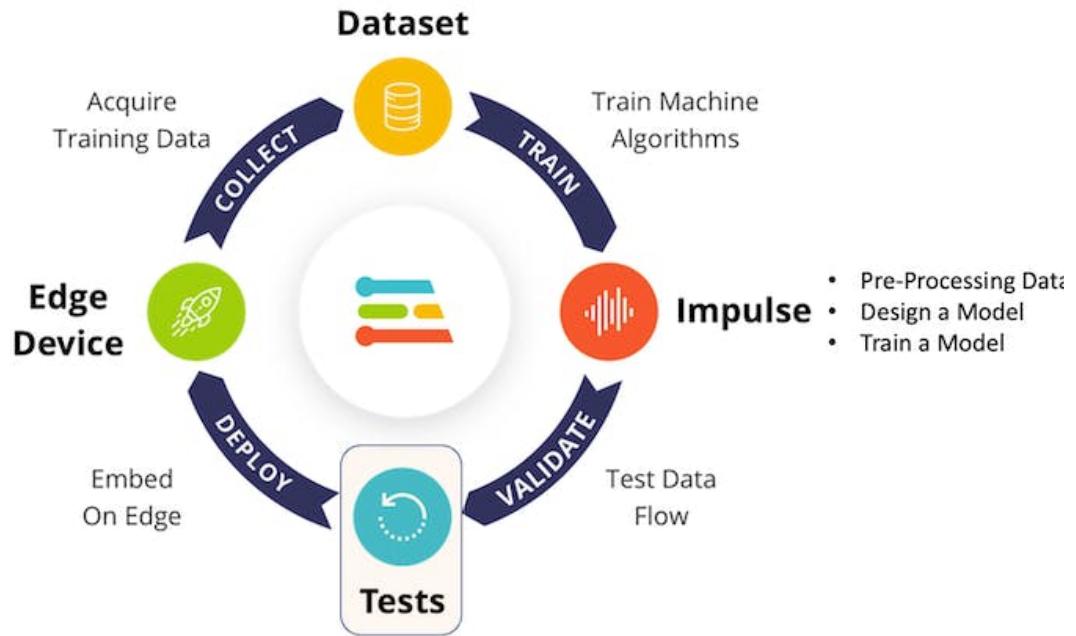
Exposure to these variations during training can help prevent your model from taking shortcuts by “memorizing” superficial clues in your training data, meaning it may better reflect the deep underlying patterns in your dataset.

The final layer of our model will have 12 neurons with a 15% dropout for overfitting prevention. Here is the Training result:

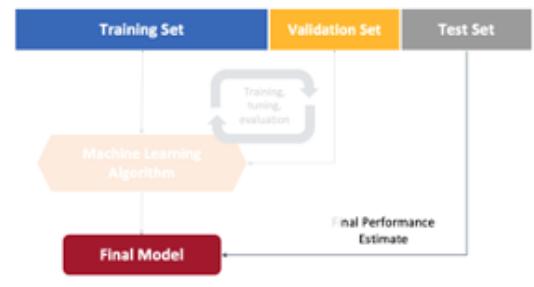


The result is excellent, with 77ms of latency, which should result in 13fps (frames per second) during inference.

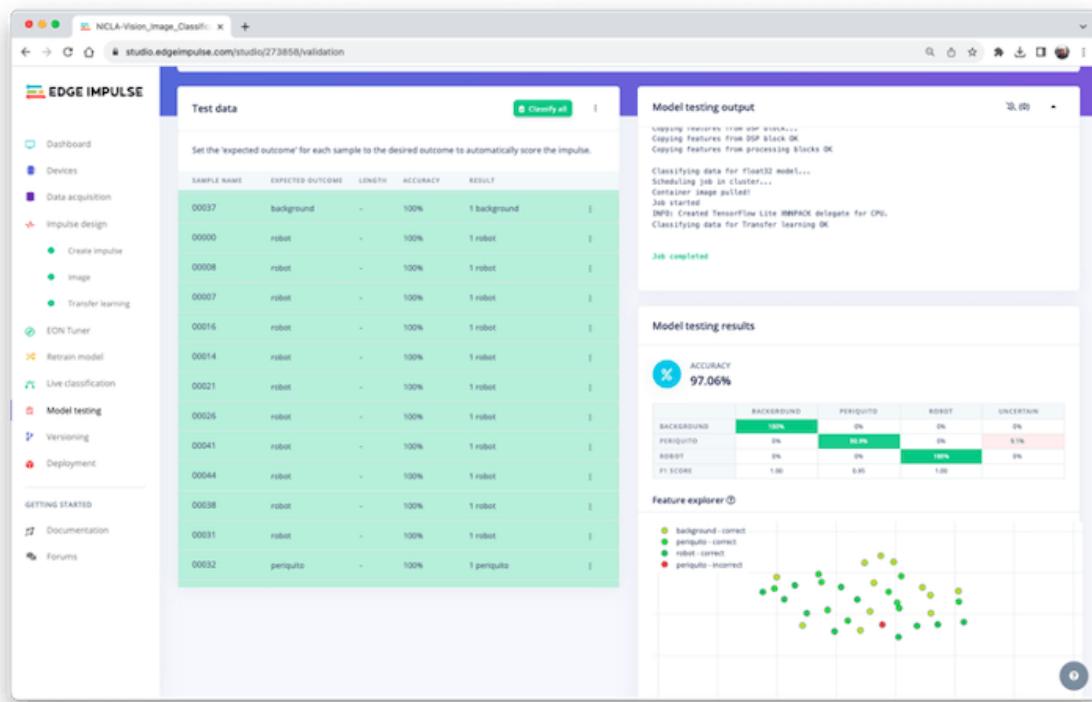
2.9 Model Testing



Now, you should take the data set aside at the start of the project and run the trained model using it as input:

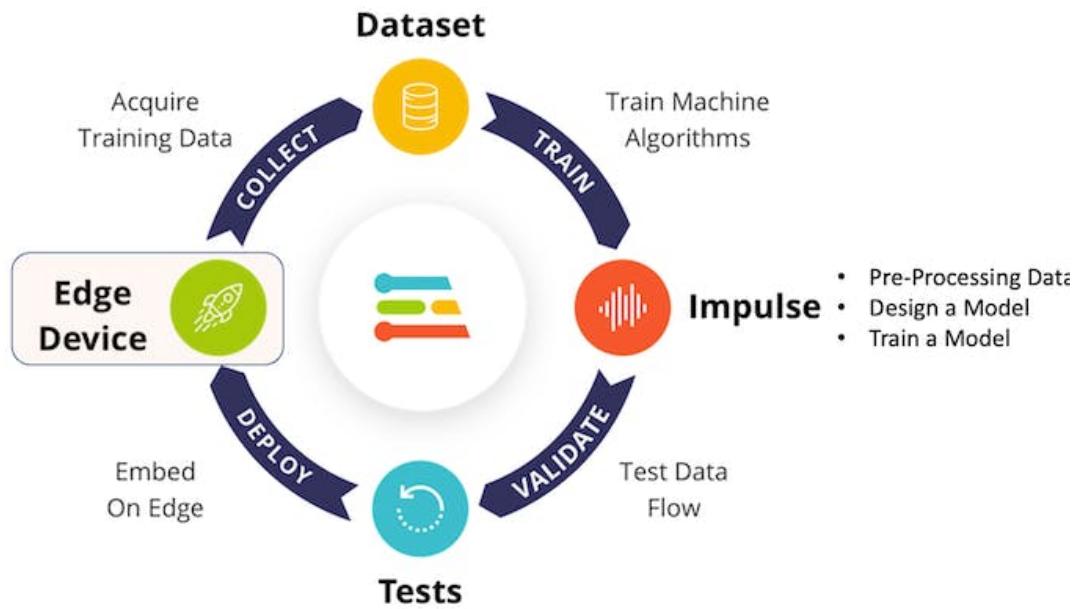


The result is, again, excellent.



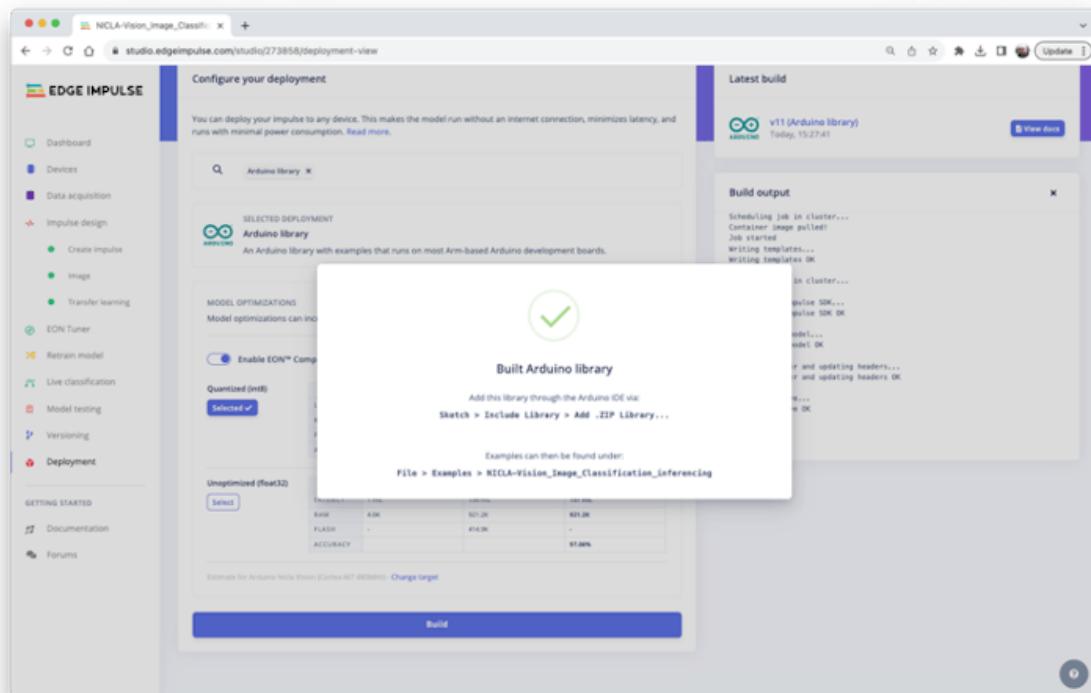
2.10 Deploying the model

At this point, we can deploy the trained model as.tflite and use the OpenMV IDE to run it using MicroPython, or we can deploy it as a C/C++ or an Arduino library.



2.10.1 Arduino Library

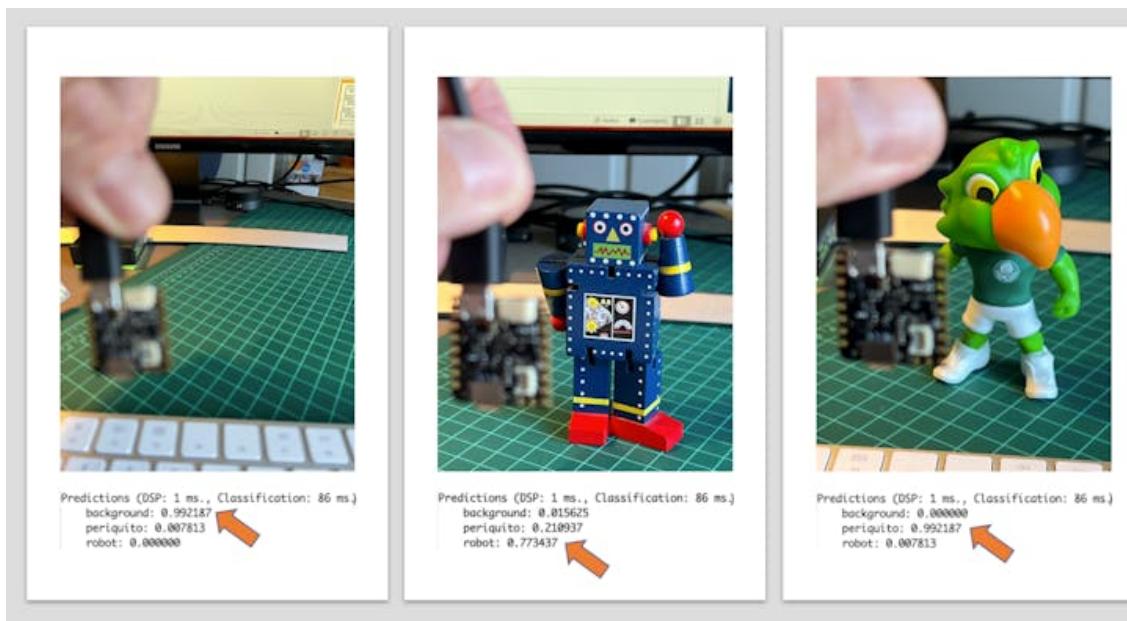
First, Let's deploy it as an Arduino Library:



You should install the library as.zip on the Arduino IDE and run the sketch *nicla_vision_camera.ino* available in Examples under your library name.

Note that Arduino Nicla Vision has, by default, 512KB of RAM allocated for the M7 core and an additional 244KB on the M4 address space. In the code, this allocation was changed to 288 kB to guarantee that the model will run on the device (`malloc_addblock((void*)0x30000000, 288 * 1024);`).

The result is good, with 86ms of measured latency.



Here is a short video showing the inference results:

<https://youtu.be/bZPZZJblU-o>

2.10.2 OpenMV

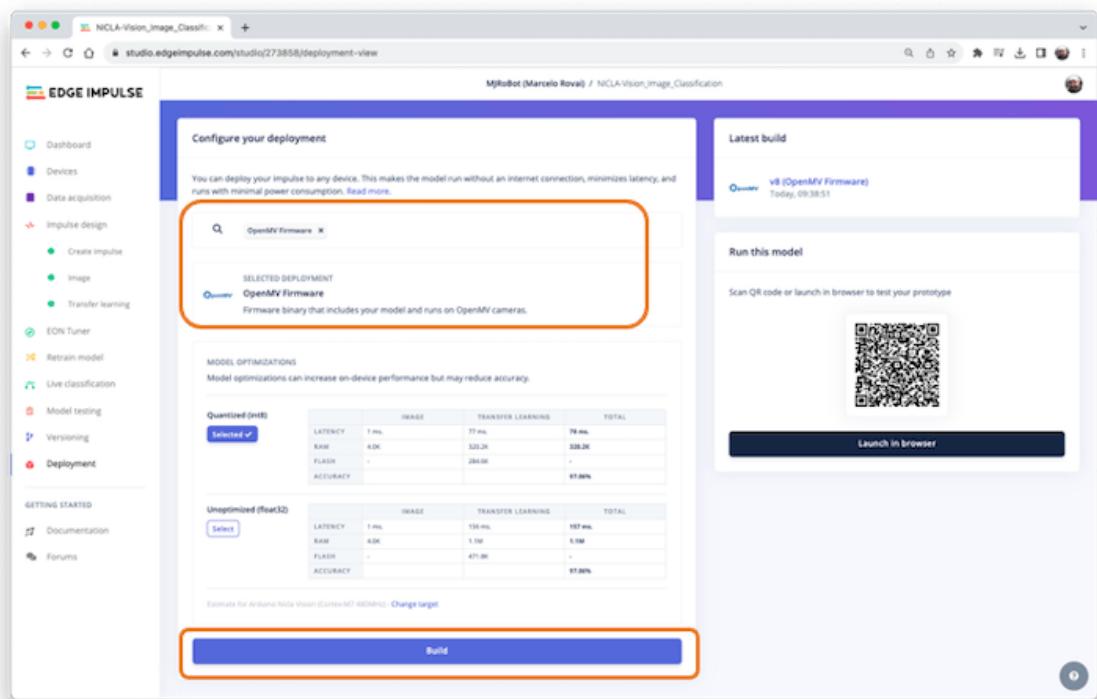
It is possible to deploy the trained model to be used with OpenMV in two ways: as a library and as a firmware.

Three files are generated as a library: the trained.tflite model, a list with labels, and a simple MicroPython script that can make inferences using the model.

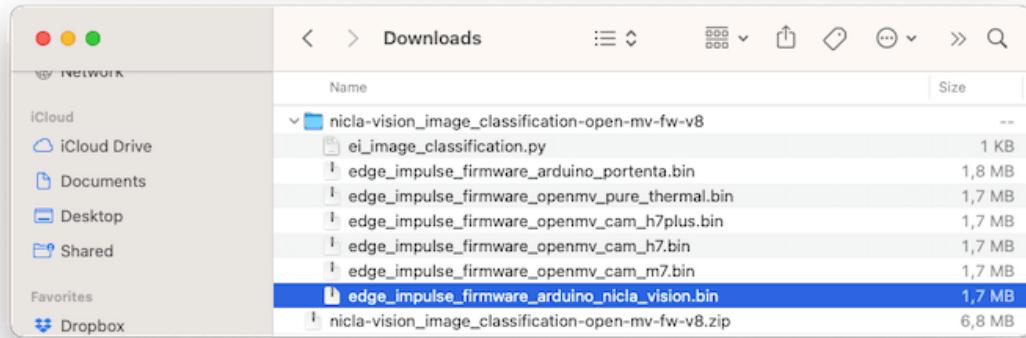
Name	Size	Kind	Date Added
ei-nicla-vision_image_classification-openmv-v17	--	Folder	Today 14:59
trained.tflite	234 KB	TensorFlow...Lite Model	Today 14:59
labels.txt	26 bytes	Plain Text Document	Today 14:59
ei_image_classification.py	2 KB	Python File	Today 14:59
ei-nicla-vision_image_classification-openmv-v17.zip	140 KB	ZIP archive	Today 14:59

Running this model as a `.tflite` directly in the Nicla was impossible. So, we can sacrifice the accuracy using a smaller model or deploy the model as an OpenMV Firmware (FW). Choosing FW, the Edge Impulse Studio generates optimized models, libraries, and frameworks needed to make the inference. Let's explore this option.

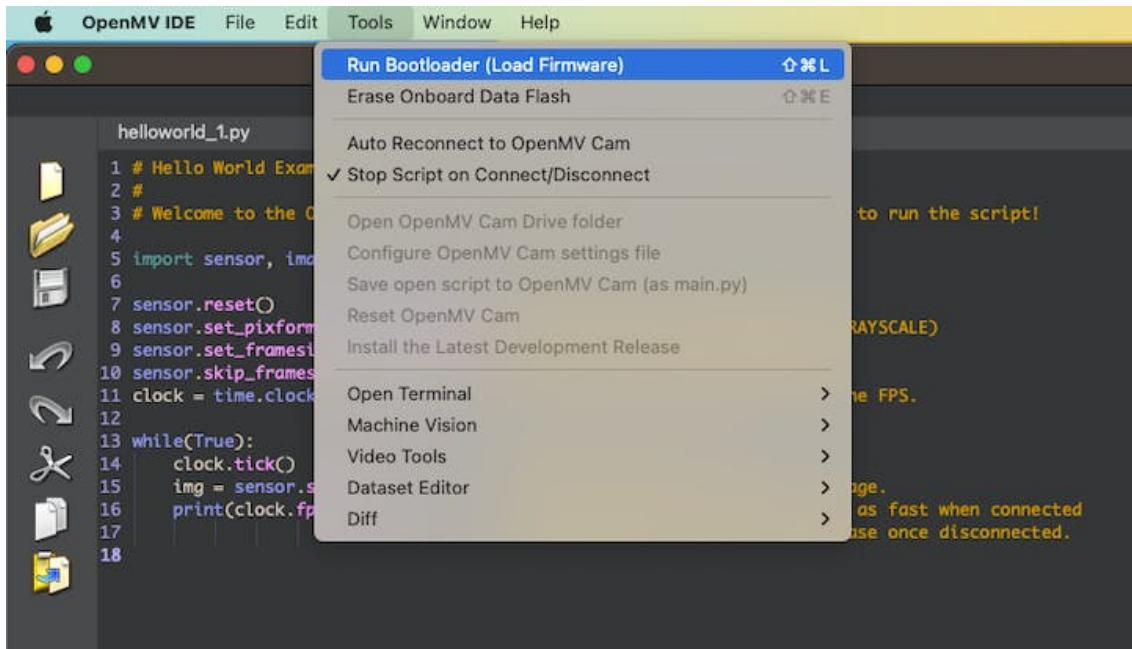
Select OpenMV Firmware on the Deploy Tab and press [Build].



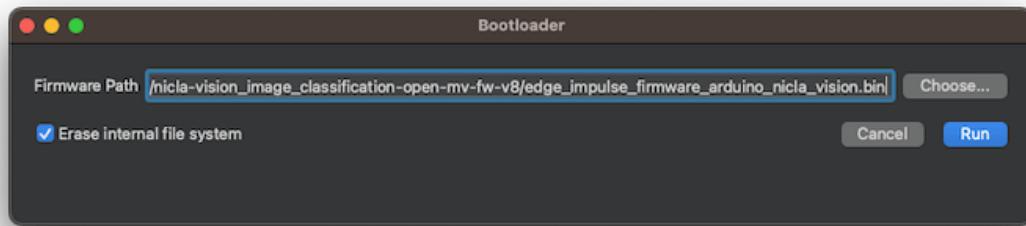
On your computer, you will find a ZIP file. Open it:



Use the Bootloader tool on the OpenMV IDE to load the FW on your board:



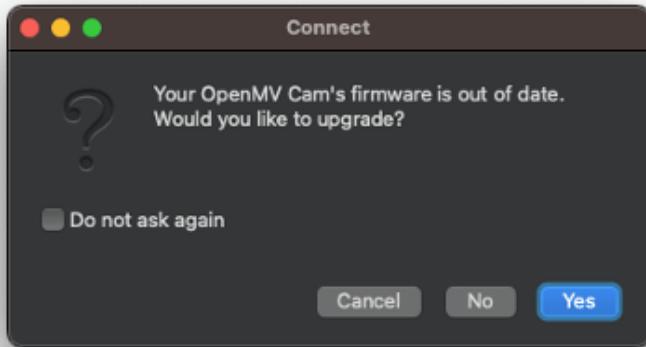
Select the appropriate file (.bin for Nicla-Vision):



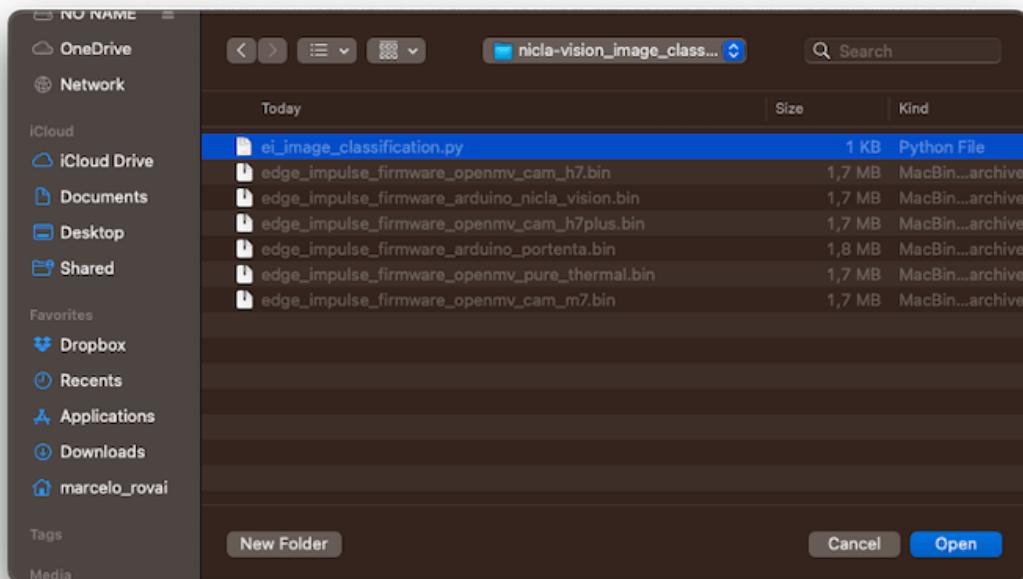
After the download is finished, press OK:



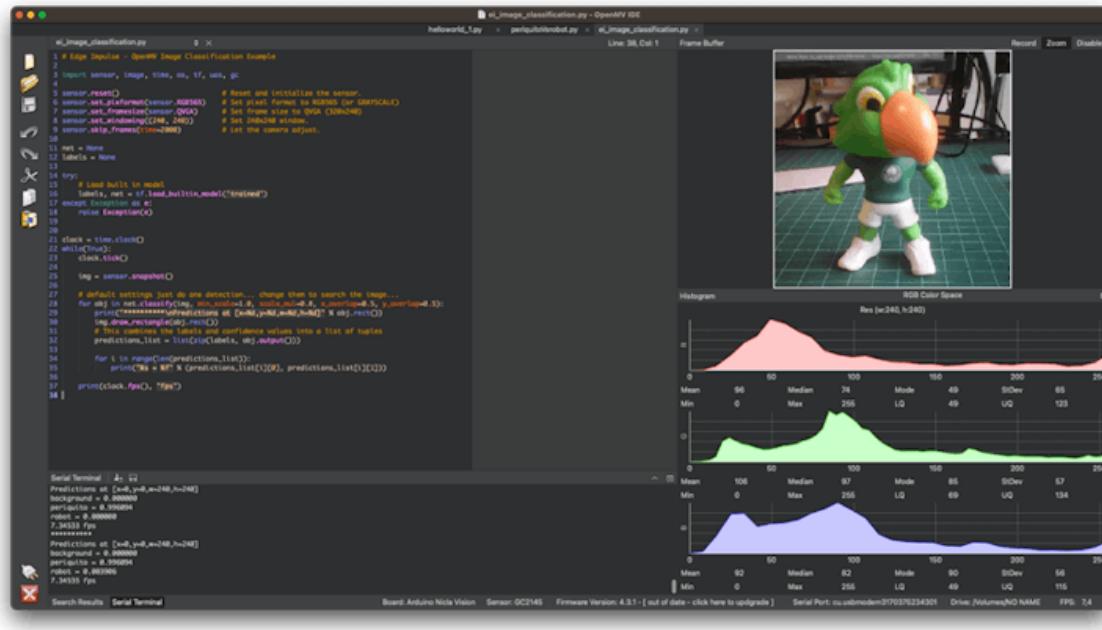
If a message says that the FW is outdated, DO NOT UPGRADE. Select [NO].



Now, open the script **ei_image_classification.py** that was downloaded from the Studio and the .bin file for the Nicla.



Run it. Pointing the camera to the objects we want to classify, the inference result will be displayed on the Serial Terminal.



2.10.2.1 Changing the Code to add labels

The code provided by Edge Impulse can be modified so that we can see, for test reasons, the inference result directly on the image displayed on the OpenMV IDE.

[Upload the code from GitHub](#), or modify it as below:

```
# Marcelo Rovai - NICLA Vision - Image Classification
# Adapted from Edge Impulse - OpenMV Image Classification Example
# @24Aug23

import sensor, image, time, os, tf, uos, gc

sensor.reset()                                # Reset and initialize the sensor.
sensor.set_pixformat(sensor.RGB565)            # Set ppx fpx to RGB565 (or GRayscale)
sensor.set_framesize(sensor.QVGA)                # Set frame size to QVGA (320x240)

sensor.set_windowing((240, 240))                # Set 240x240 window.
sensor.skip_frames(time=2000)                   # Let the camera adjust.

net = None
labels = None

try:
    # Load built in model
    labels, net = tf.load_builtin_model('trained')
except Exception as e:
```

```

raise Exception(e)

clock = time.clock()
while(True):
    clock.tick() # Starts tracking elapsed time.

    img = sensor.snapshot()

    # default settings just do one detection
    for obj in net.classify(img,
                             min_scale=1.0,
                             scale_mul=0.8,
                             x_overlap=0.5,
                             y_overlap=0.5):
        fps = clock.fps()
        lat = clock.avg()

        print("*****\nPrediction:")
        img.draw_rectangle(obj.rect())
        # This combines the labels and confidence values into a list of tuples
        predictions_list = list(zip(labels, obj.output()))

        max_val = predictions_list[0][1]
        max_lbl = 'background'
        for i in range(len(predictions_list)):
            val = predictions_list[i][1]
            lbl = predictions_list[i][0]

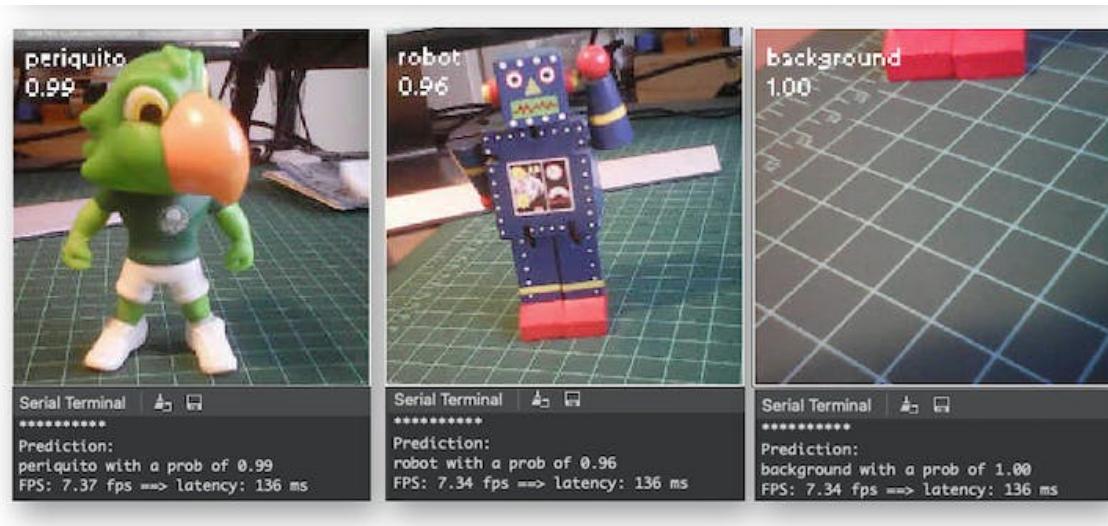
            if val > max_val:
                max_val = val
                max_lbl = lbl

        # Print label with the highest probability
        if max_val < 0.5:
            max_lbl = 'uncertain'
        print("{} with a prob of {:.2f}{}".format(max_lbl, max_val))
        print("FPS: {:.2f} fps ==> latency: {:.0f} ms".format(fps, lat))

        # Draw label with highest probability to image viewer
        img.draw_string(
            10, 10,
            max_lbl + "\n{:.2f}".format(max_val),
            mono_space = False,
            scale=2
        )

```

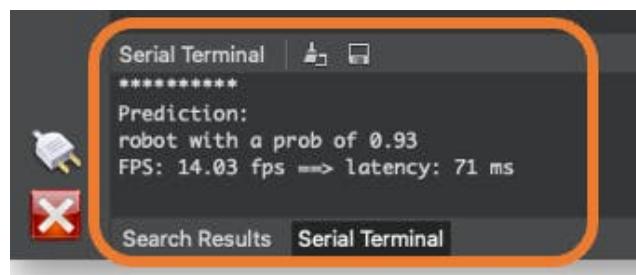
Here you can see the result:



Note that the latency (136 ms) is almost double of what we got directly with the Arduino IDE. This is because we are using the IDE as an interface and also the time to wait for the camera to be ready. If we start the clock just before the inference:

```
56 while(True):
57
58     img = sensor.snapshot()
59
60     clock.tick() # Starts tracking elapsed time.
61
62     # default settings just do one detection... change them to search the image...
63     for obj in net.classify(img, min_scale=1.0, scale_mul=0.8, x_overlap=0.5, y_overlap=0.5):
64         fps = clock.fps()
65         lat = clock.avg()
66
67         print("*****\nPrediction:")
68         img.draw_rectangle(obj.rect())
69         # This combines the labels and confidence values into a list of tuples
70         predictions_list = list(zip(labels, obj.output()))
71
```

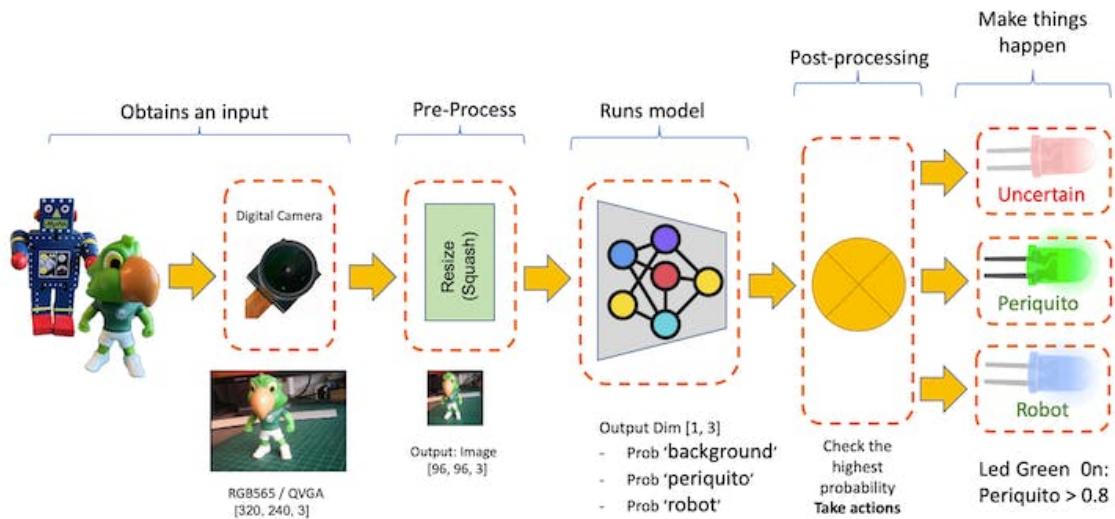
The latency will drop to only 71 ms.



The NiclaV runs about half as fast when connected to the IDE. The FPS should increase once disconnected.

2.10.2.2 Post-Processing with LEDs

When working with embedded machine learning, we are looking for devices that can continually proceed with the inference and result, taking some action directly on the physical world and not displaying the result on a connected computer. To simulate this, we will light up a different LED for each possible inference result.



To accomplish that, we should [upload the code from GitHub](#) or change the last code to include the LEDs:

```

# Marcelo Rovai - NICLA Vision - Image Classification with LEDs
# Adapted from Edge Impulse - OpenMV Image Classification Example
# @24Aug23

import sensor, image, time, os, tf, uos, gc, pyb

ledRed = pyb.LED(1)
ledGre = pyb.LED(2)
ledBlu = pyb.LED(3)

sensor.reset()                                # Reset and initialize the sensor.
sensor.set_pixformat(sensor.RGB565)           # Set pixel fmt to RGB565 (or GRAYSCALE)
sensor.set_framesize(sensor.QVGA)              # Set frame size to QVGA (320x240)
sensor.set_windowing((240, 240))               # Set 240x240 window.
sensor.skip_frames(time=2000)                  # Let the camera adjust.

net = None
labels = None
  
```

```

ledRed.off()
ledGre.off()
ledBlu.off()

try:
    # Load built in model
    labels, net = tf.load_builtin_model('trained')
except Exception as e:
    raise Exception(e)

clock = time.clock()

def setLEDs(max_lbl):

    if max_lbl == 'uncertain':
        ledRed.on()
        ledGre.off()

        ledBlu.off()

    if max_lbl == 'periquito':
        ledRed.off()
        ledGre.on()
        ledBlu.off()

    if max_lbl == 'robot':
        ledRed.off()
        ledGre.off()
        ledBlu.on()

    if max_lbl == 'background':
        ledRed.off()
        ledGre.off()
        ledBlu.off()

while(True):
    img = sensor.snapshot()
    clock.tick() # Starts tracking elapsed time.

    # default settings just do one detection.
    for obj in net.classify(img,
                            min_scale=1.0,
                            scale_mul=0.8,
                            x_overlap=0.5,
                            y_overlap=0.5):
        fps = clock.fps()
        lat = clock.avg()

        print("*****\nPrediction:")
        img.draw_rectangle(obj.rect())

```

```

img.draw_string(...),
# This combines the labels and confidence values into a list of tuples
predictions_list = list(zip(labels, obj.output()))

max_val = predictions_list[0][1]
max_lbl = 'background'
for i in range(len(predictions_list)):
    val = predictions_list[i][1]
    lbl = predictions_list[i][0]

    if val > max_val:
        max_val = val
        max_lbl = lbl

# Print label and turn on LED with the highest probability
if max_val < 0.8:
    max_lbl = 'uncertain'

setLEDs(max_lbl)

print("{} with a prob of {:.2f}".format(max_lbl, max_val))
print("FPS: {:.2f} fps ==> latency: {:.0f} ms".format(fps, lat))

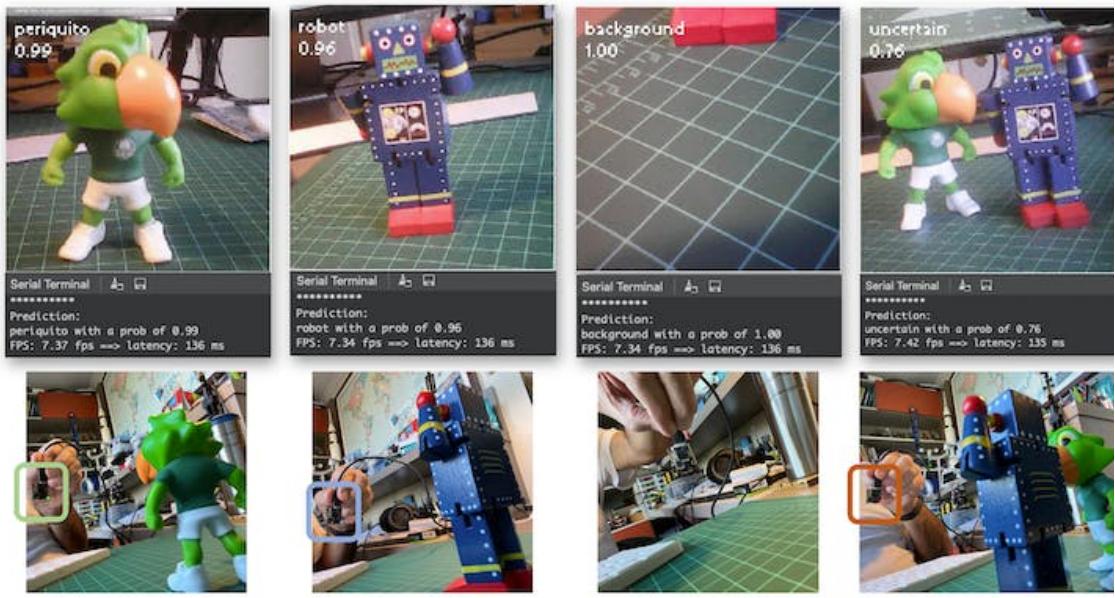
# Draw label with highest probability to image viewer
img.draw_string(
    10, 10,
    max_lbl + "\n{:.2f}".format(max_val),
    mono_space = False,
    scale=2
)

```

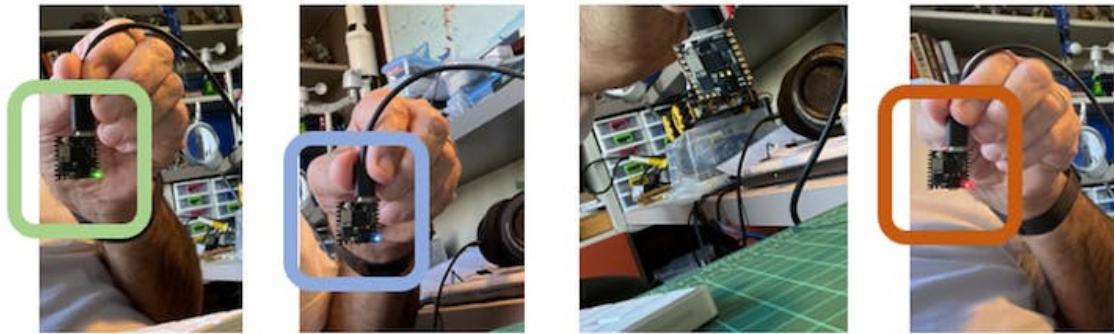
Now, each time that a class scores a result greater than 0.8, the correspondent LED will be lit:

- Led Red On: Uncertain (no class is over 0.8)
- Led Green On: Periquito > 0.8
- Led Blue On: Robot > 0.8
- All LEDs Off: Background > 0.8

Here is the result:



In more detail



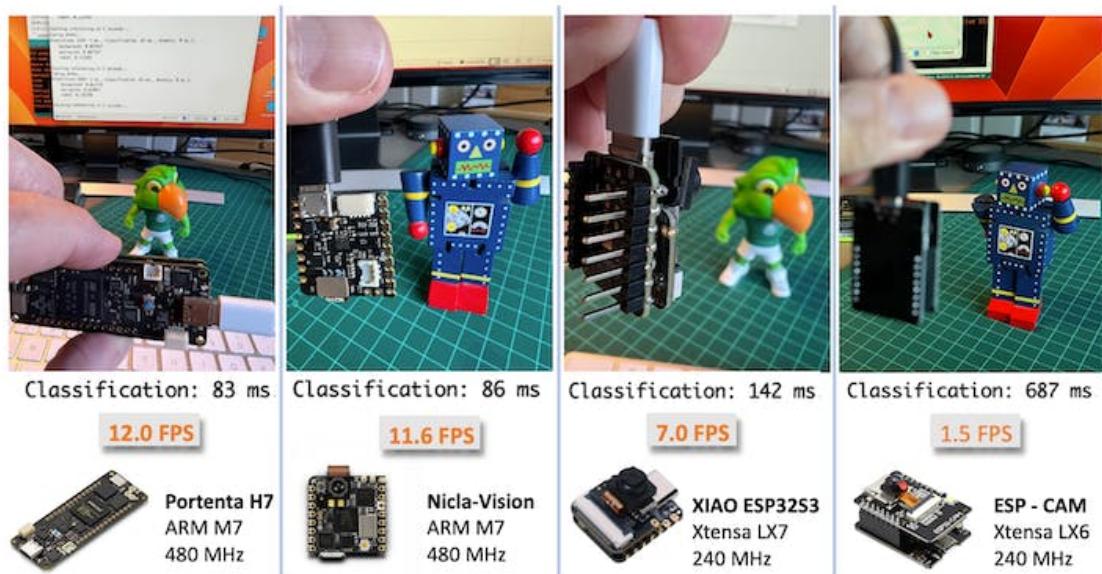
2.11 Image Classification (non-official) Benchmark

Several development boards can be used for embedded machine learning (tinyML), and the most common ones for Computer Vision applications (consuming low energy), are the ESP32 CAM, the Seeed XIAO ESP32S3 Sense, the Arduino Nicla Vison, and the Arduino Portenta.



	ESP 32	Seeed XIAO Sense / ESP32S3	Arduino Pro
32Bits CPU	Xtensa LX6 Dual Core	Arm Cortex-M4F (BLE) Xtensa LX7 Dual Core	Dual Core Arm Cortex M7/M4
CLOCK	240MHz	64 / 240MHz	480/240MHz
RAM	520KB (part available)	256KB / 8MB	1MB
ROM	2MB	2MB / 8MB	2MB
Radio	BLE/WiFi	BLE / WiFi (ESP32S3)	BLE/WiFi
Sensors	Yes (CAM)	Yes (Sense)	Yes (Nicla)
Bat. Power Manag.	No	Yes	Yes
Price	\$	\$\$	\$\$\$\$

Catching the opportunity, the same trained model was deployed on the ESP-CAM, the XIAO, and the Portenta (in this one, the model was trained again, using grayscaled images to be compatible with its camera). Here is the result, deploying the models as Arduino's Library:

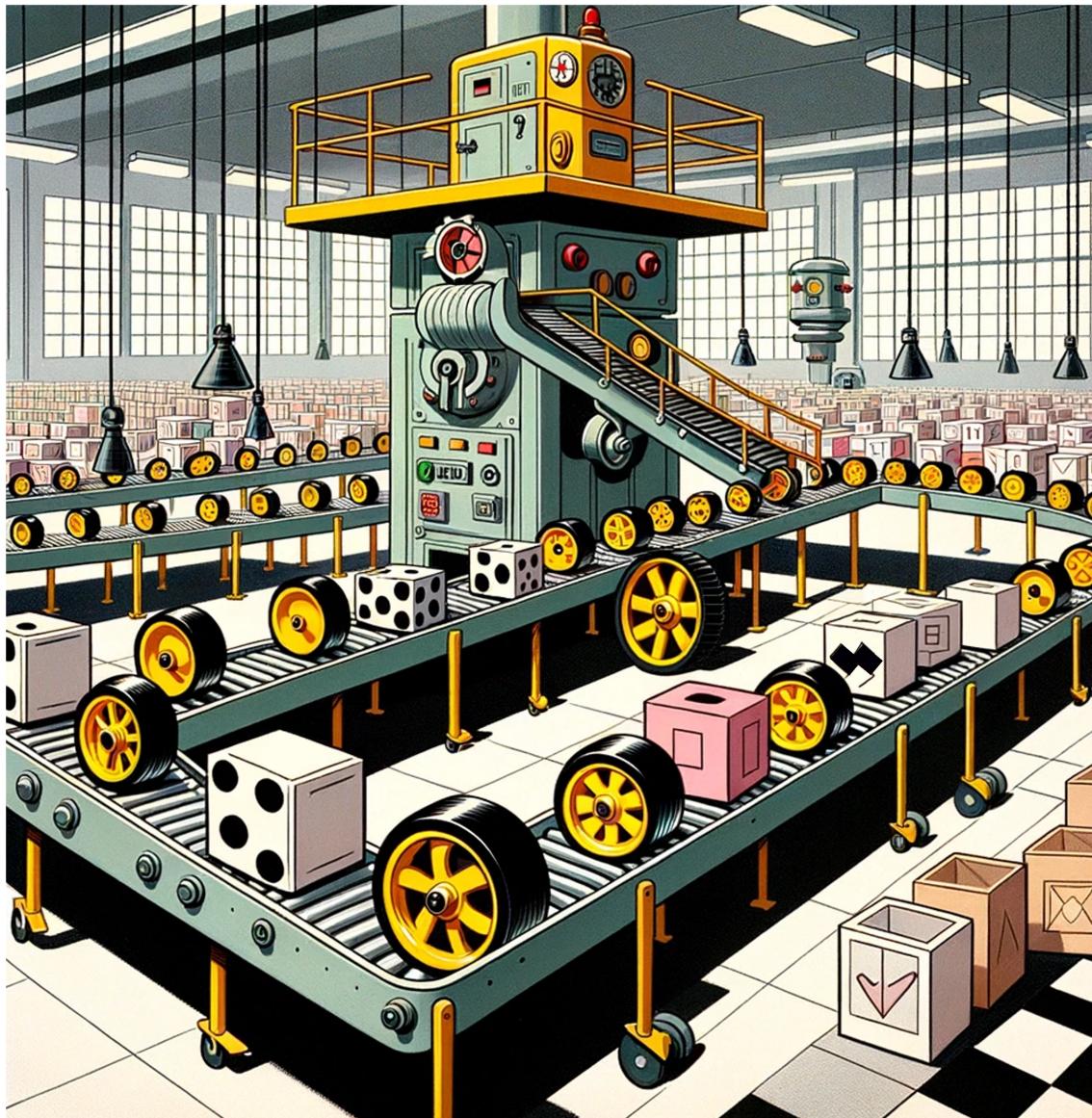


2.12 Conclusion

Before we finish, consider that Computer Vision is more than just image classification. For example, you can develop Edge Machine Learning projects around vision in several areas, such as:

- **Autonomous Vehicles:** Use sensor fusion, lidar data, and computer vision algorithms to navigate and make decisions.
- **Healthcare:** Automated diagnosis of diseases through MRI, X-ray, and CT scan image analysis
- **Retail:** Automated checkout systems that identify products as they pass through a scanner.
- **Security and Surveillance:** Facial recognition, anomaly detection, and object tracking in real-time video feeds.
- **Augmented Reality:** Object detection and classification to overlay digital information in the real world.
- **Industrial Automation:** Visual inspection of products, predictive maintenance, and robot and drone guidance.
- **Agriculture:** Drone-based crop monitoring and automated harvesting.
- **Natural Language Processing:** Image captioning and visual question answering.
- **Gesture Recognition:** For gaming, sign language translation, and human-machine interaction.
- **Content Recommendation:** Image-based recommendation systems in e-commerce.

3 Object Detection



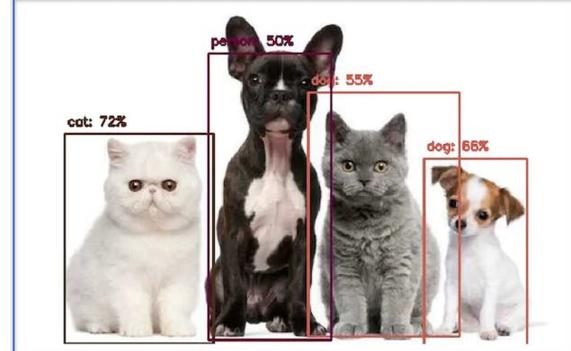
3.1 Introduction

This is a continuation of **CV on Nicla Vision**, now exploring **Object Detection** on microcontrollers.

Image Classification (Multi-Class Classification)



Object Detection Multi-Label Classification + Object Localization

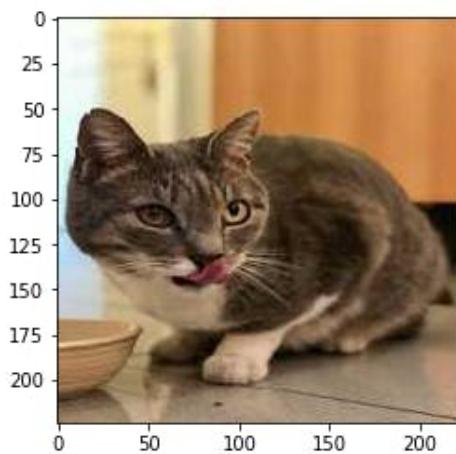


3.1.1 Object Detection versus Image Classification

The main task with Image Classification models is to produce a list of the most probable object categories present on an image, for example, to identify a tabby cat just after his dinner:

[PREDICTION]:

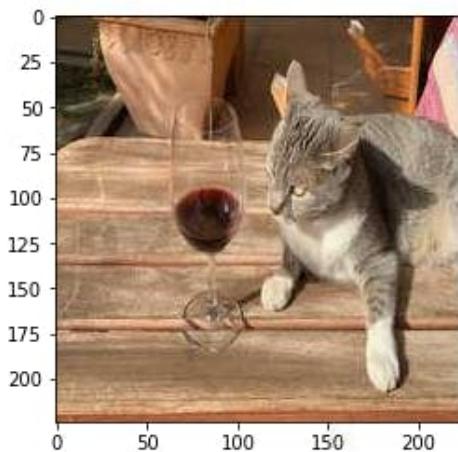
- 1) [tabby] ==> Probability of 30%
- 2) [bow tie] ==> Probability of 11%
- 3) [Egyptian cat] ==> Probability of 18%



But what happens when the cat jumps near the wine glass? The model still only recognizes the predominant category on the image, the tabby cat:

[PREDICTION]:

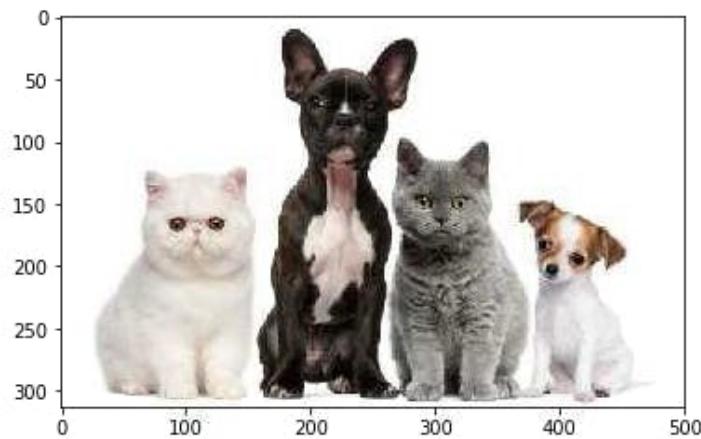
- 1) [tabby] ==> Probability of 53%
- 2) [tiger cat] ==> Probability of 23%
- 3) [Egyptian cat] ==> Probability of 10%



And what happens if there is not a dominant category on the image?

[PREDICTION] [Prob]

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

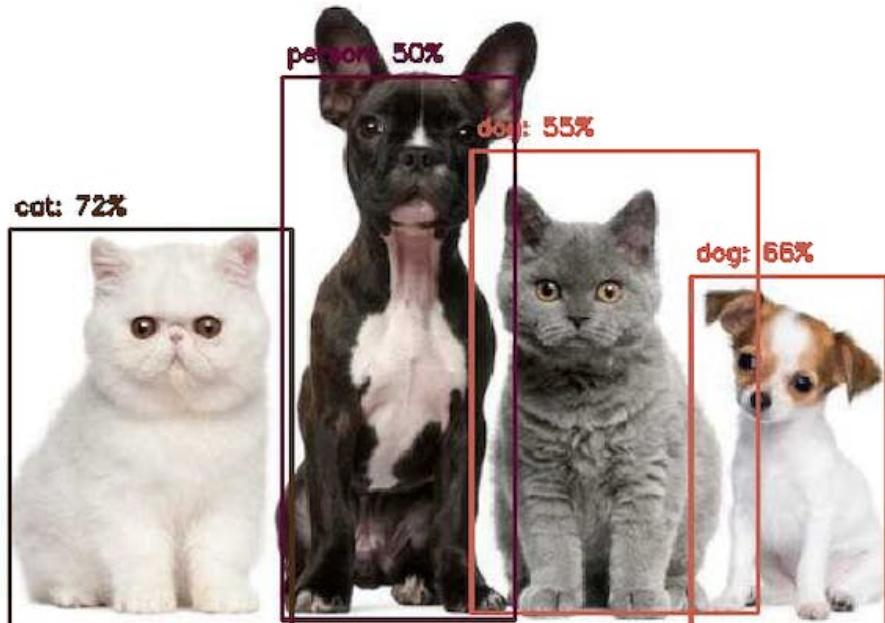


The model identifies the above image completely wrong as an “ashcan,” possibly due to the color tonalities.

The model used in all previous examples is the *MobileNet*, trained with a large dataset, the *ImageNet*.

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 is lower than 1M Bytes.

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, not only on the Nicla Vision (Cortex M7) but also on Cortex M4F CPUs (Arduino Nano33 and OpenMV M4 series) as well the Espressif ESP32 devices (ESP-CAM and XIAO ESP32S3 Sense).

In this Hands-On exercise, we will explore using FOMO with Object Detection, not entering many details about the model itself. To understand more about how the model works, 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 facility and must sort and count **wheels** and special **boxes**.



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

- Background (No objects)

- Box
- Wheel

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



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.

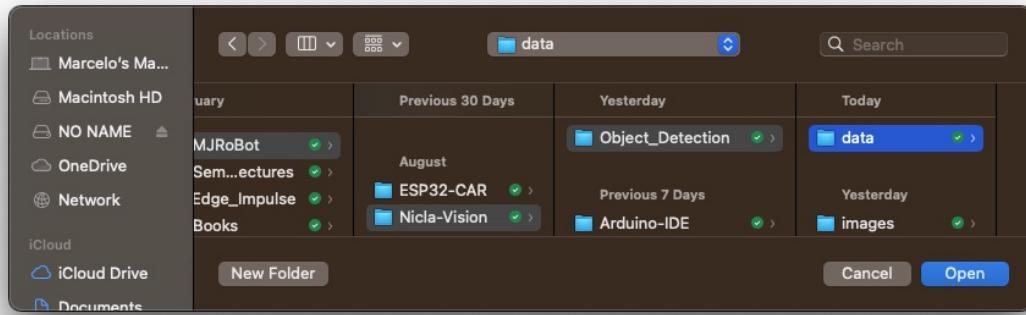
We will develop the project using the Nicla Vision for image capture and model inference. The ML project will be developed using the Edge Impulse Studio. But before starting the object detection project in the Studio, let's create a *raw dataset* (not labeled) with images that contain the objects to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

3.3 Data Collection

We can use the Edge Impulse Studio, the OpenMV IDE, your phone, or other devices for the image capture. Here, we will use again the OpenMV IDE for our purpose.

3.3.1 Collecting Dataset with OpenMV IDE

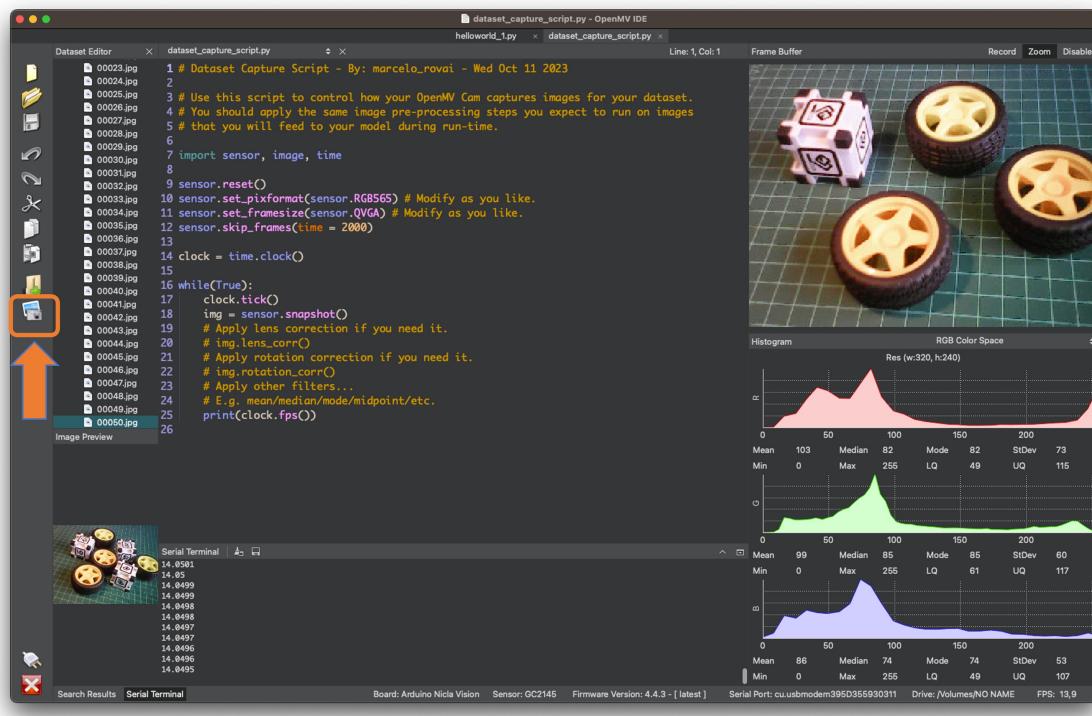
First, create in your computer a folder where your data will be saved, for example, “data.” Next, on the OpenMV IDE, go to Tools > Dataset Editor and select New Dataset to start the dataset collection:



Edge impulse suggests that the objects should be of similar 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 with mixed sizes and positions to see the result.

We will not create separate folders for our images because each contains multiple labels.

Connect the Nicla Vision to the OpenMV IDE and run the `dataset_capture_script.py`. Clicking on the Capture Image button will start capturing images:



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

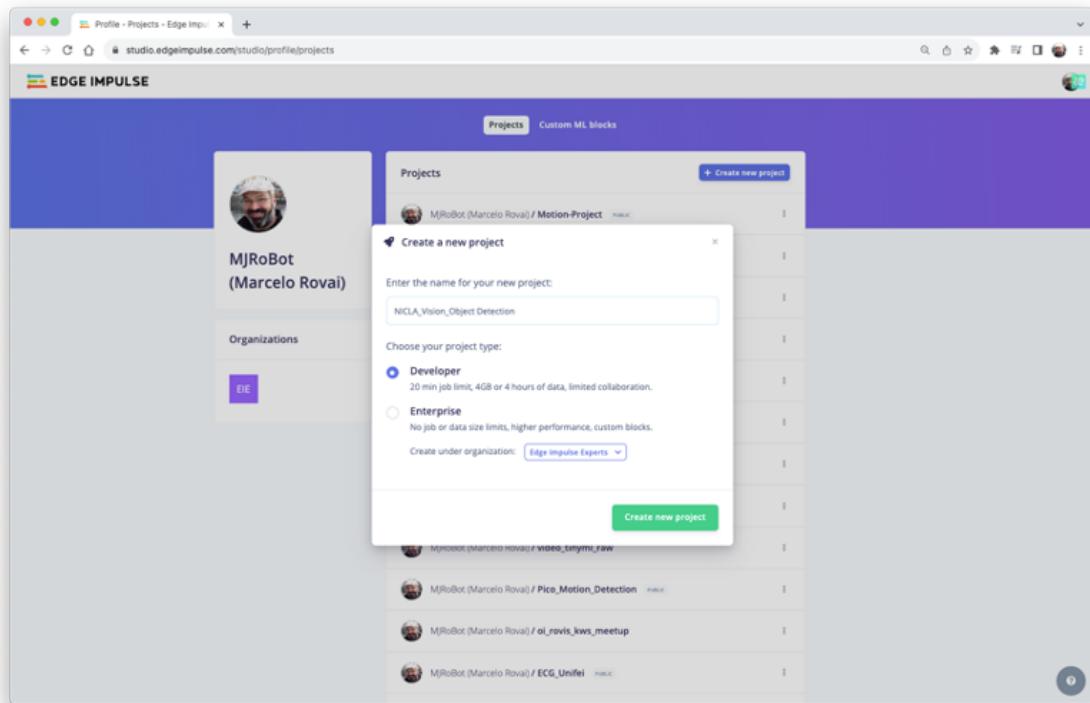
The stored images use a QVGA frame size 320x240 and RGB565 (color pixel format).

After capturing your dataset, close the Dataset Editor Tool on the Tools > Dataset Editor.

3.4 Edge Impulse Studio

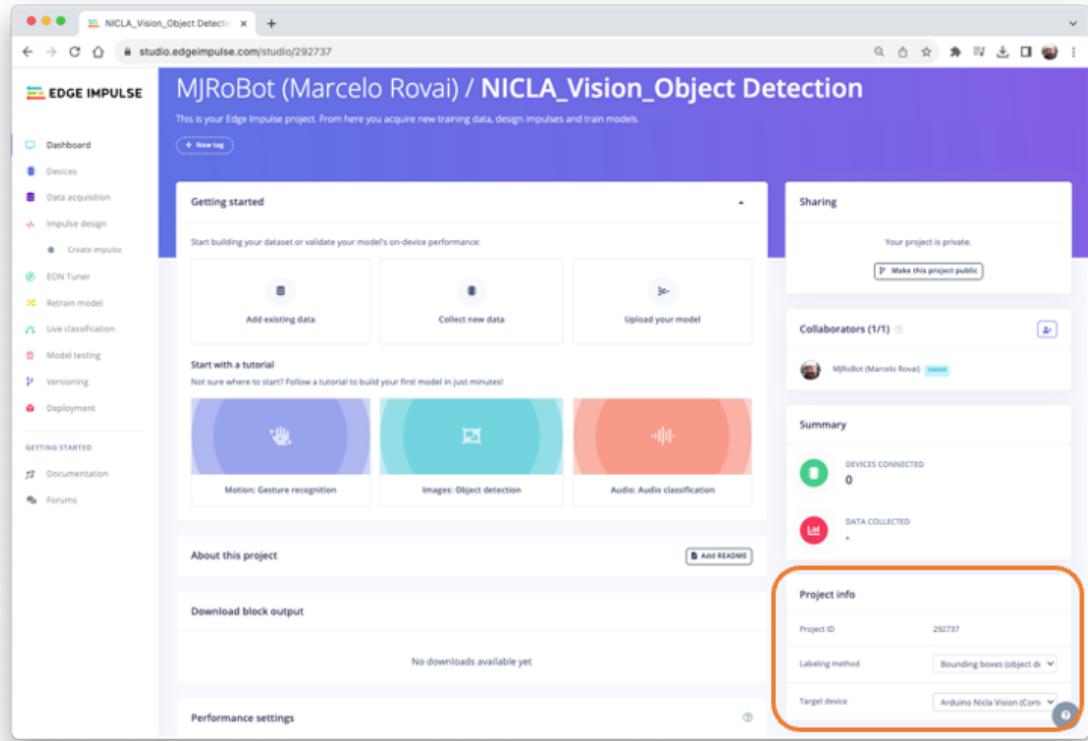
3.4.1 Setup the project

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



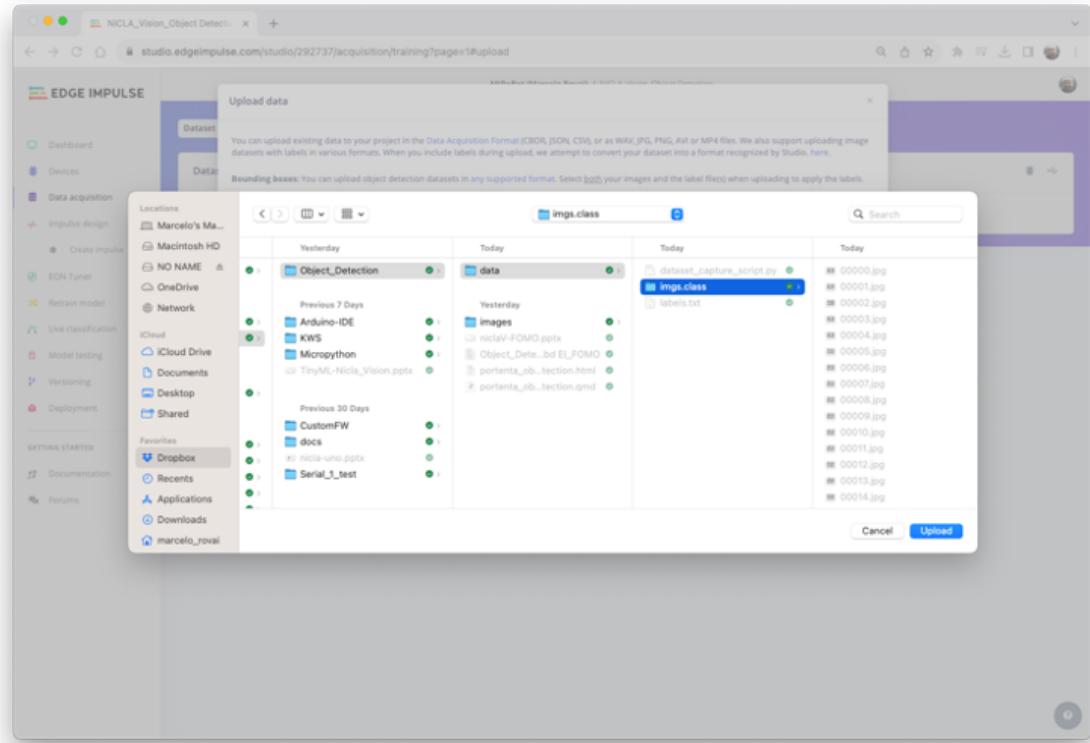
Here, you can clone the project developed for this hands-on:
[NICLA_Vision_Object_Detection](#).

On your Project Dashboard, go down and on **Project info** and select **Bounding boxes (object detection)** and Nicla Vision as your Target Device:

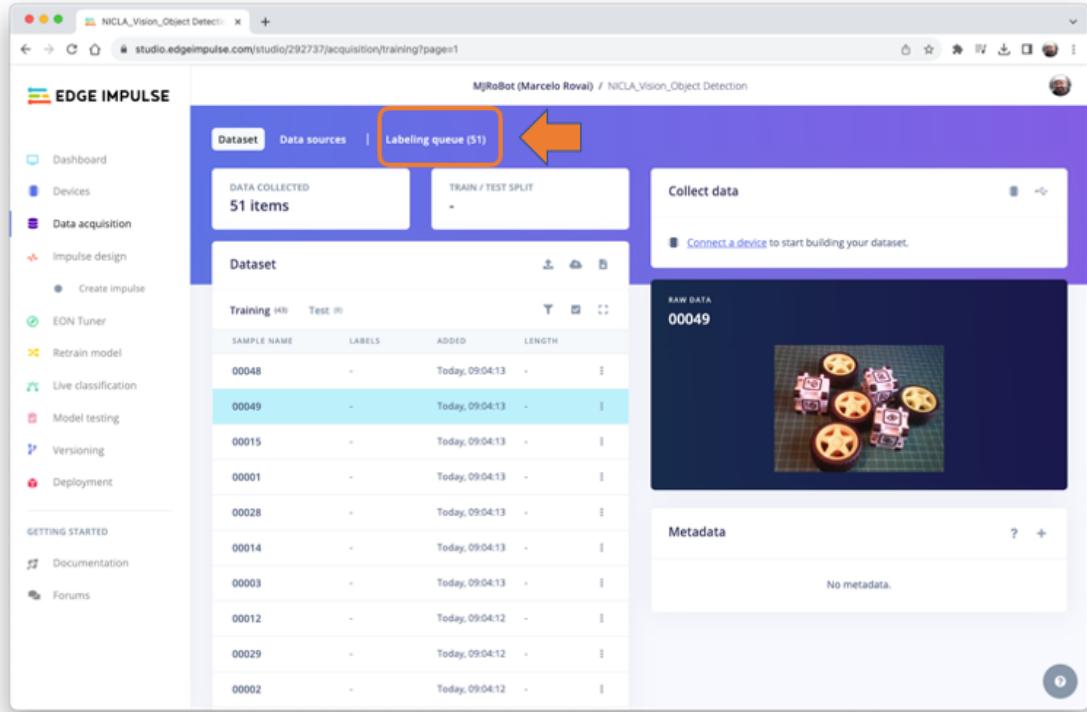


3.4.2 Uploading the unlabeled data

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



You can leave for the Studio to split your data automatically between Train and Test or do it manually.



All the not labeled images (51) were uploaded but they still need to be labeled appropriately before using them as a dataset in the project. The Studio has a tool for that purpose, which you can find in the link [Labeling queue \(51\)](#).

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

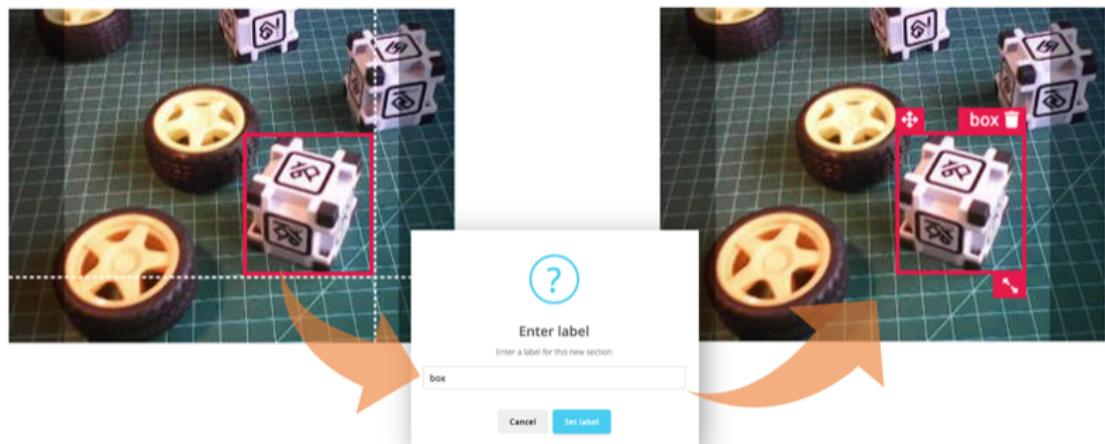
Edge Impulse launched an [auto-labeling feature](#) for Enterprise customers, easing labeling tasks in object detection projects.

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).

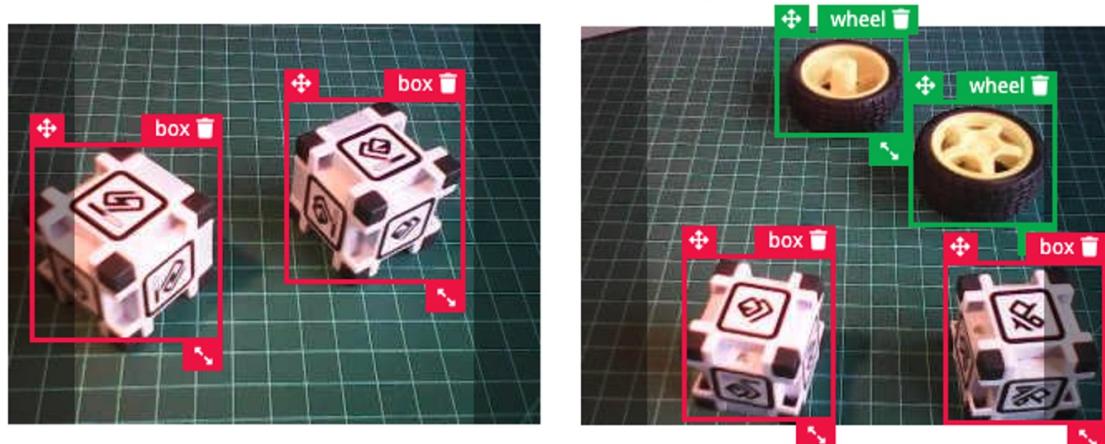
You can use the [El uploader](#) to import your data if you already have a labeled dataset containing bounding boxes.

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:

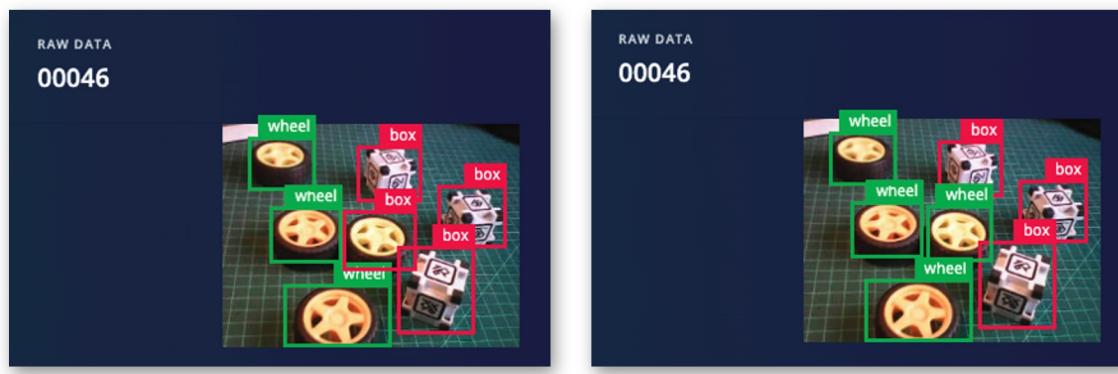


Next, review the labeled samples on the [Data acquisition](#) tab. If one of the labels was wrong, you can edit it using the *three dots* menu after the

sample name:

The screenshot shows the Edge Impulse web interface. On the left, a sidebar menu includes options like Dashboard, Devices, Data acquisition, Impulse design, Create impulse, EON Tuner, Retrain model, Live classification, Model testing, Versioning, Deployment, Documentation, and Forums. The main area is titled 'Dataset' and shows 'DATA COLLECTED 51 items' and 'TRAIN / TEST SPLIT 86% / 14%'. A table lists training samples with labels such as 'wheel', 'box', and 'whe...'. A context menu is open over sample '00046'. To the right, a 'Collect data' section says 'Connect a device to start building your dataset.' Below it, a 'RAW DATA' section shows an image of objects labeled 'wheel' and 'box' with bounding boxes. A 'Metadata' section indicates 'No metadata.'

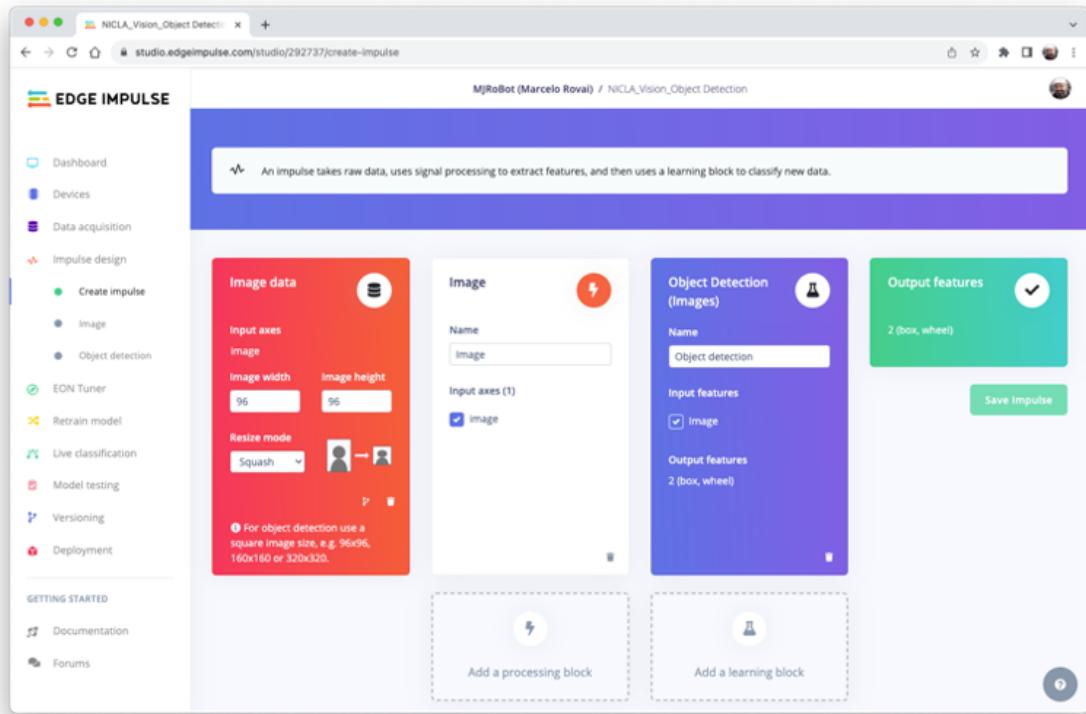
You will be guided to replace the wrong label, correcting the dataset.



3.5 The Impulse Design

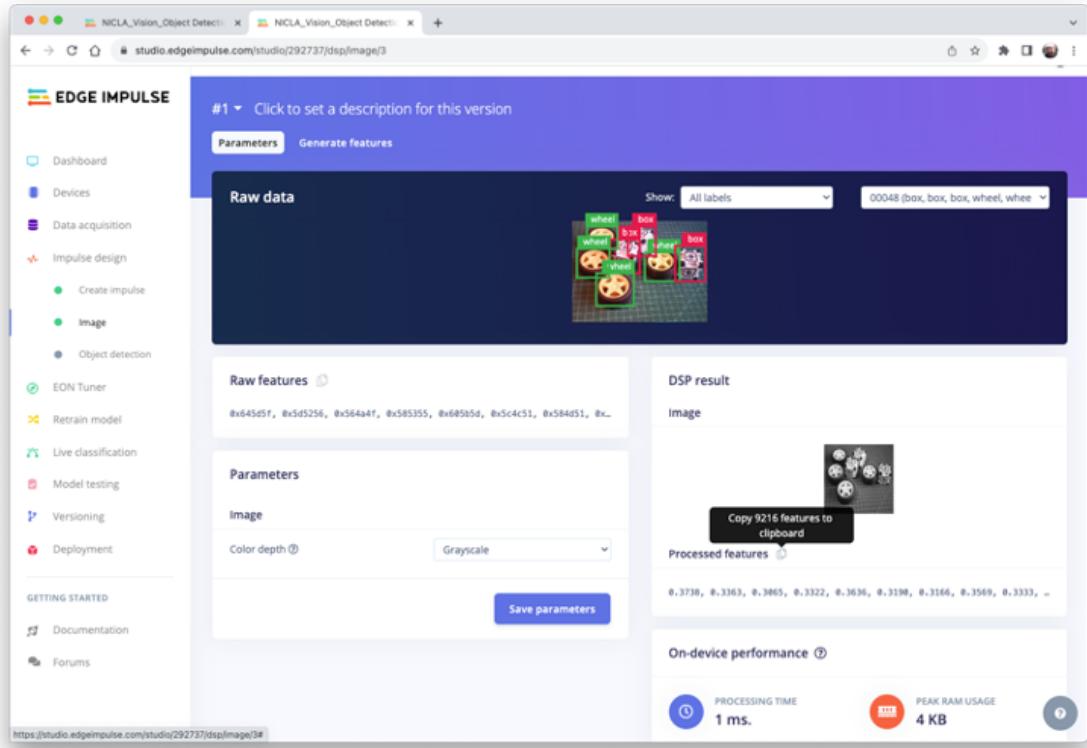
In this phase, you should define how to:

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

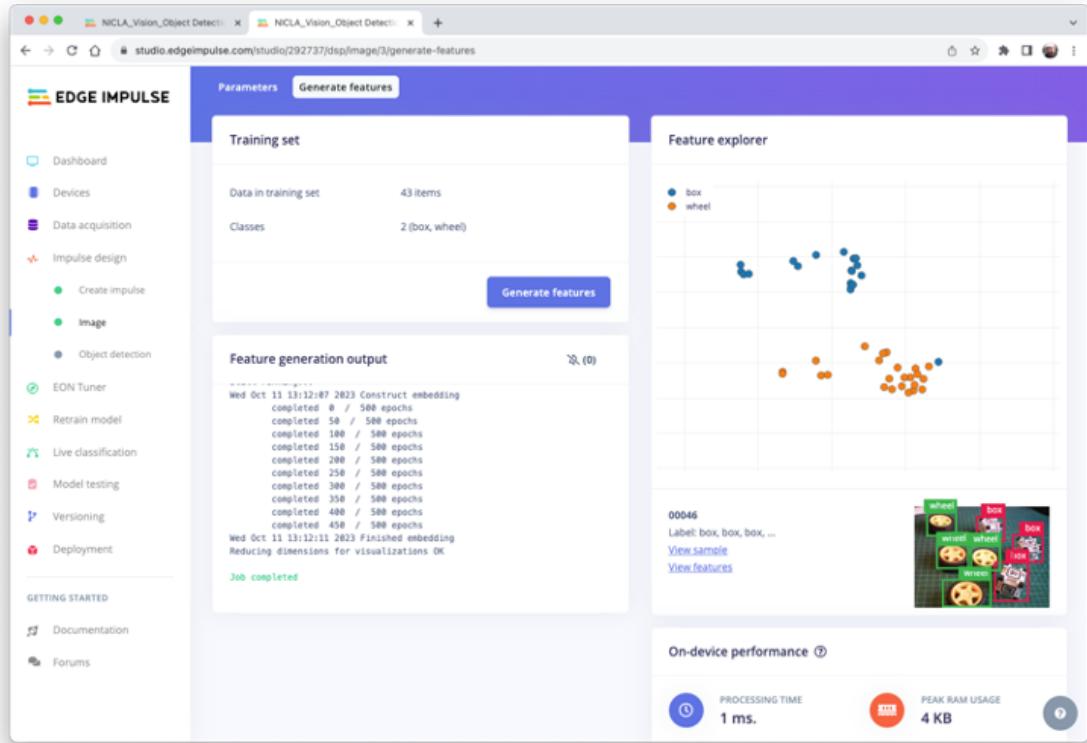


3.5.1 Preprocessing all dataset

In this section, select **Color depth** as Grayscale, which is 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.

One of the samples (46) apparently is in the wrong space, but clicking on it can confirm that the labeling is correct.

3.6 Model Design, Training, and Test

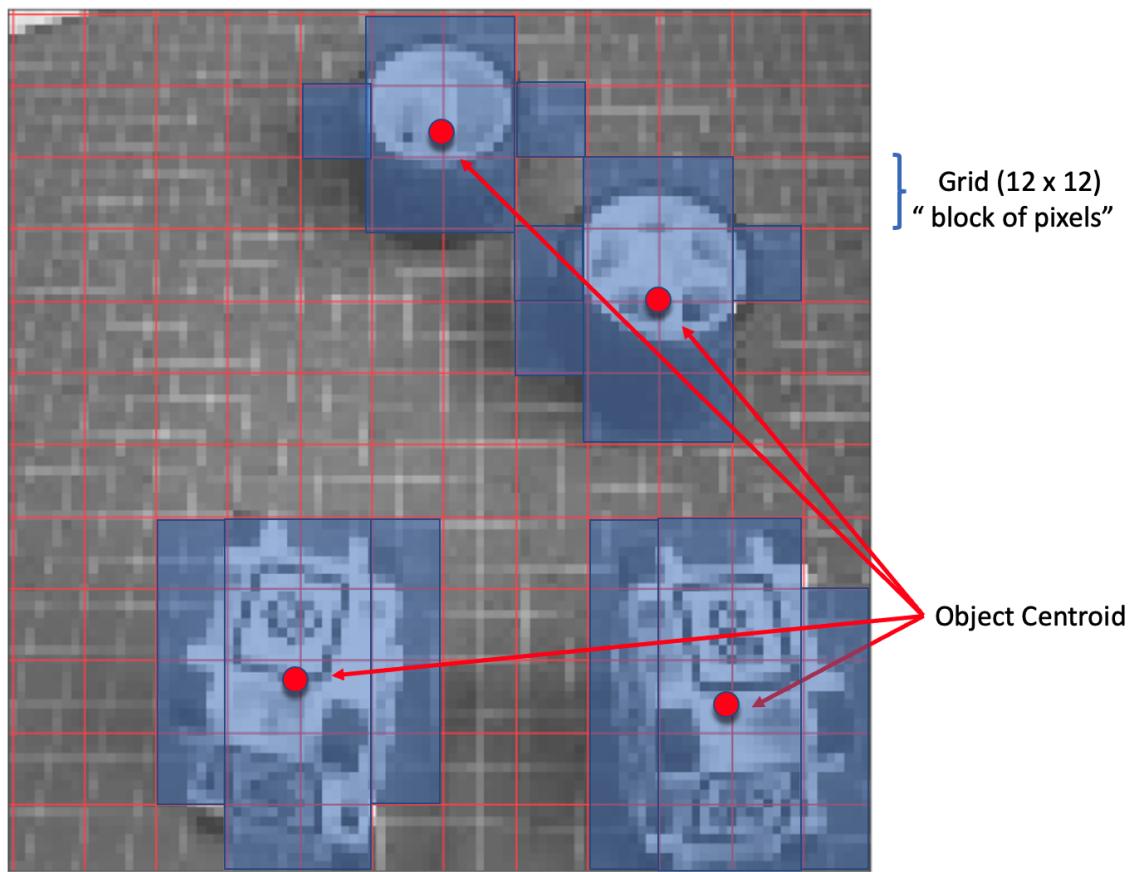
We will use FOMO, an object detection model based on MobileNetV2 (alpha 0.35) designed to coarsely segment an image into a grid of **background vs objects of interest** (here, *boxes* and *wheels*).

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,

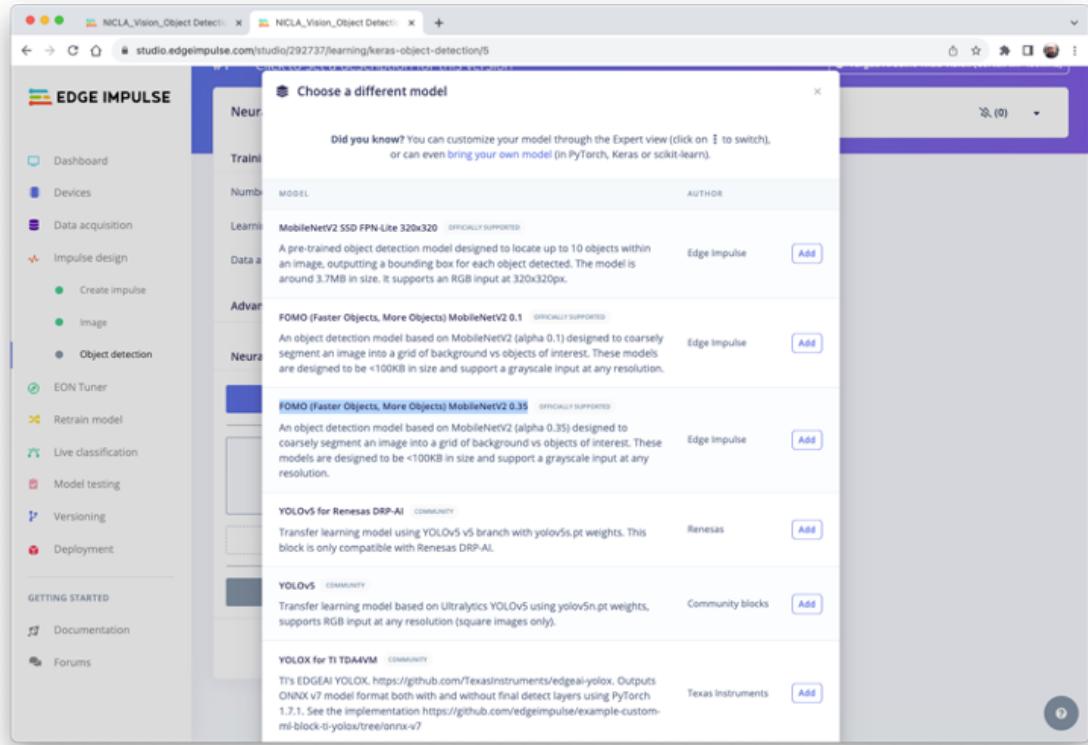
providing only the information about where the object is located in the image, by means of its centroid coordinates.

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 which 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 RAM and 80KB of ROM (Flash), which suits well with our board since it has 1MB of RAM and ROM.



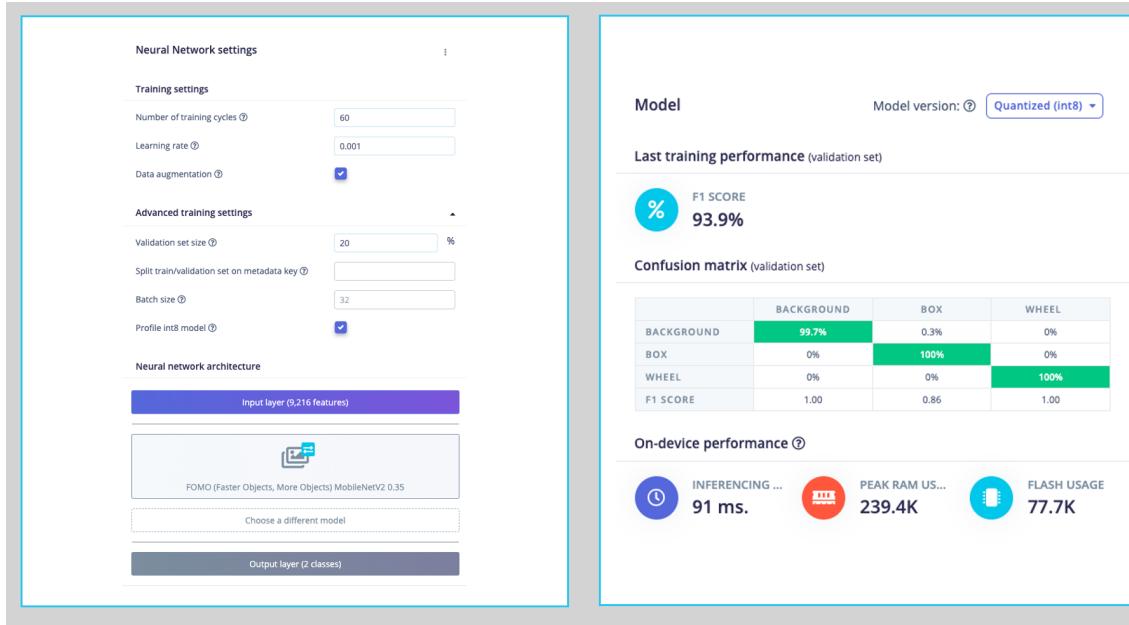
Regarding the training hyper-parameters, the model will be trained with:

- 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 practically 1.00 in the F1 score, with a similar result when using the Test data.

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”

Since Edge Impulse officially supports the Nicla Vision, let's connect it to the Studio. For that, follow the steps:

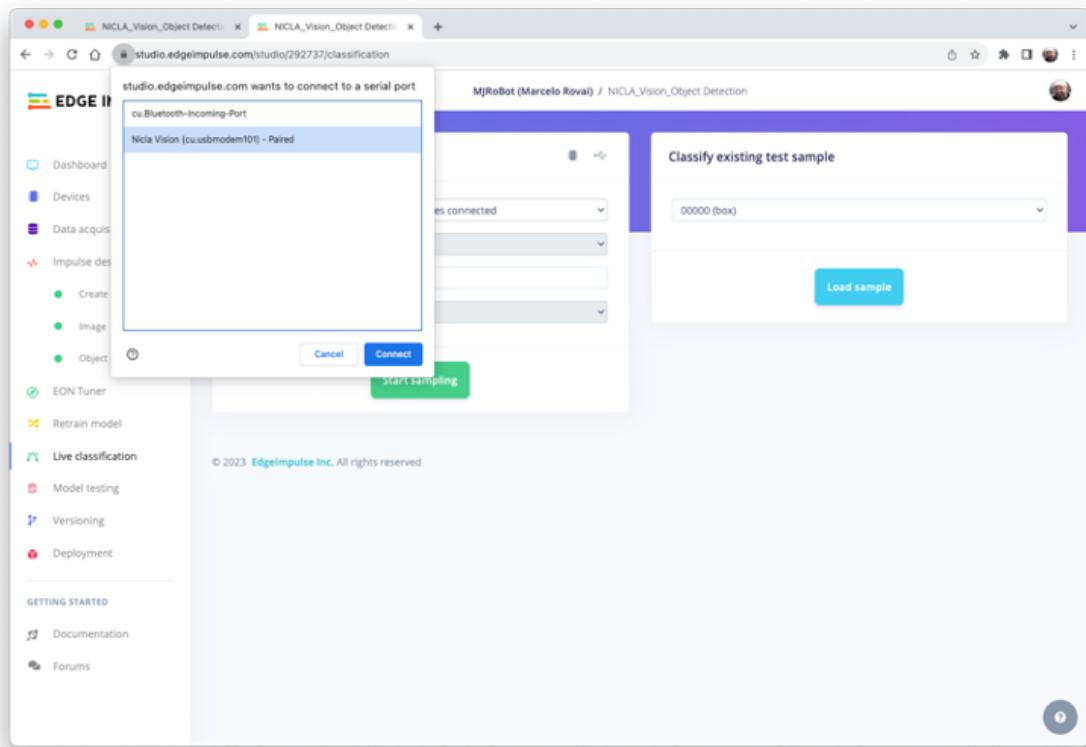
- Download the [last EI Firmware](#) and unzip it.
- Open the zip file on your computer and select the uploader related to your OS:



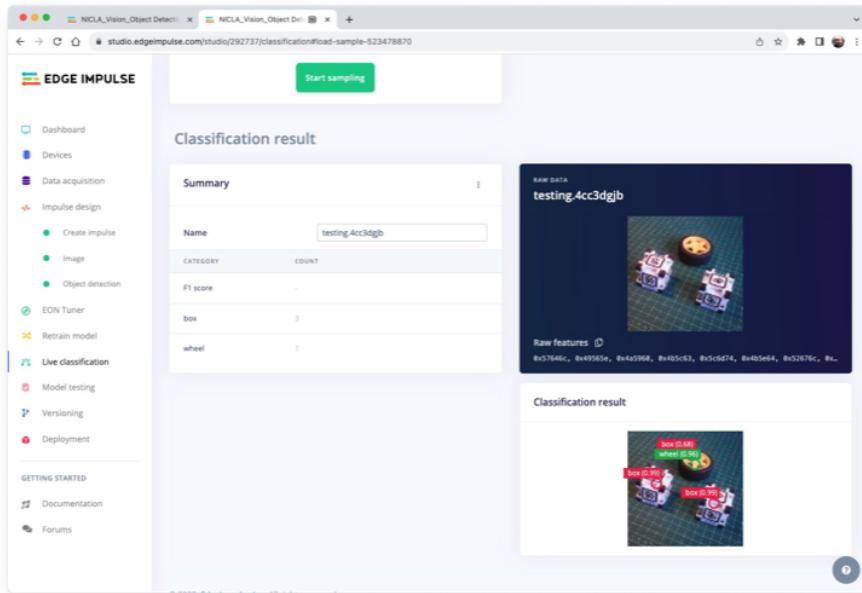
- Put the Nicla-Vision on Boot Mode, pressing the reset button twice.

- Execute the specific batch code for your OS for uploading the binary (`arduino-nicla-vision.bin`) to your board.

Go to **Live classification** section at EI Studio, and using *webUSB*, connect your Nicla Vision:



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

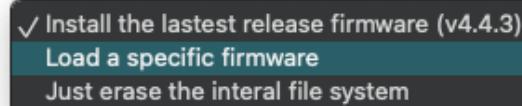


One thing to be noted is that the model can produce false positives and negatives. This can be minimized by defining a proper Confidence Threshold (use the Three dots menu for the set-up). Try with 0.8 or more.

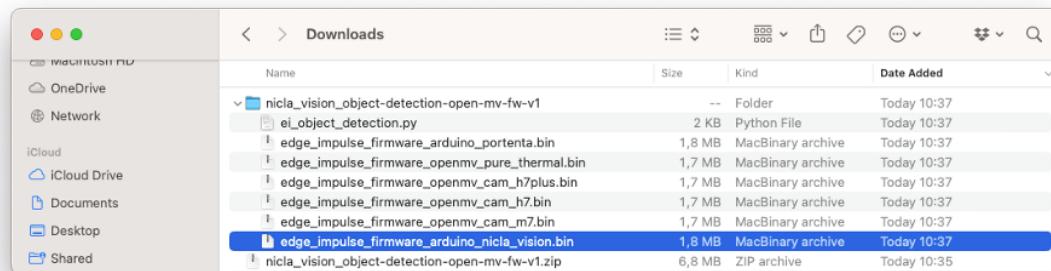
3.7 Deploying the Model

Select OpenMV Firmware on the Deploy Tab and press [Build].

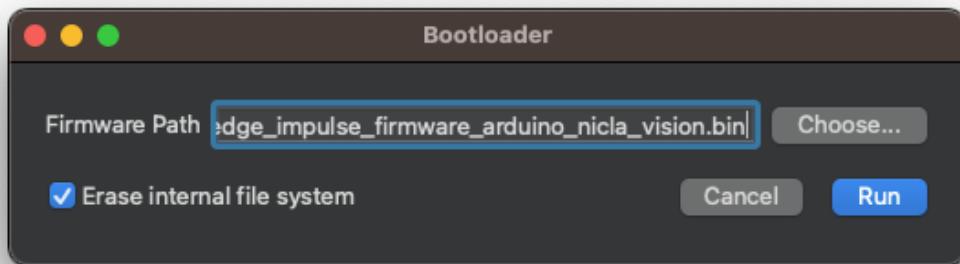
When you try to connect the Nicla with the OpenMV IDE again, it will try to update its FW. Choose the option **Load a specific firmware** instead.



You will find a ZIP file on your computer from the Studio. Open it:



Load the .bin file to your board:



After the download is finished, a pop-up message will be displayed. Press ok, and open the script **ei_object_detection.py** downloaded from the Studio.

Before running the script, let's change a few lines. Note that you can leave the window definition as 240 x 240 and the camera capturing images as QVGA/RGB. The captured image will be pre-processed by the FW deployed from Edge Impulse

```
# Edge Impulse - OpenMV Object Detection Example

import sensor, image, time, os, tf, math, uos, gc

sensor.reset()                      # Reset and initialize the sensor.
sensor.set_pixformat(sensor.RGB565)   # Set pixel format to RGB565 (or GRAYSC
sensor.set_framesize(sensor.QVGA)     # Set frame size to QVGA (320x240)
sensor.set_windowing((240, 240))      # Set 240x240 window.
sensor.skip_frames(time=2000)          # Let the camera adjust.

net = None
labels = None
```

Redefine the minimum confidence, for example, to 0.8 to minimize false positives and negatives.

```
min_confidence = 0.8
```

Change if necessary, the color of the circles that will be used to display the detected object's centroid for a better contrast.

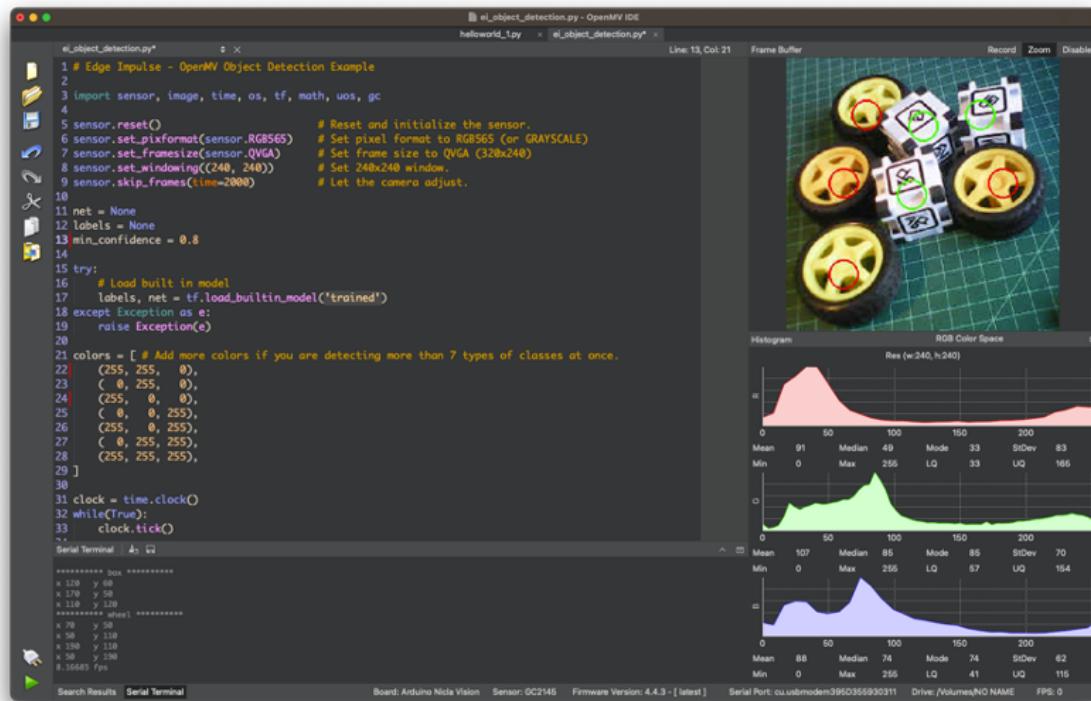
```

try:
    # Load built in model
    labels, net = tf.load_builtin_model('trained')
except Exception as e:
    raise Exception(e)

colors = [ # Add more colors if you are detecting more than 7 types of classes
    (255, 255, 0), # background: yellow (not used)
    (0, 255, 0), # cube: green
    (255, 0, 0), # wheel: red
    (0, 0, 255), # not used
    (255, 0, 255), # not used
    (0, 255, 255), # not used
    (255, 255, 255), # not used
]

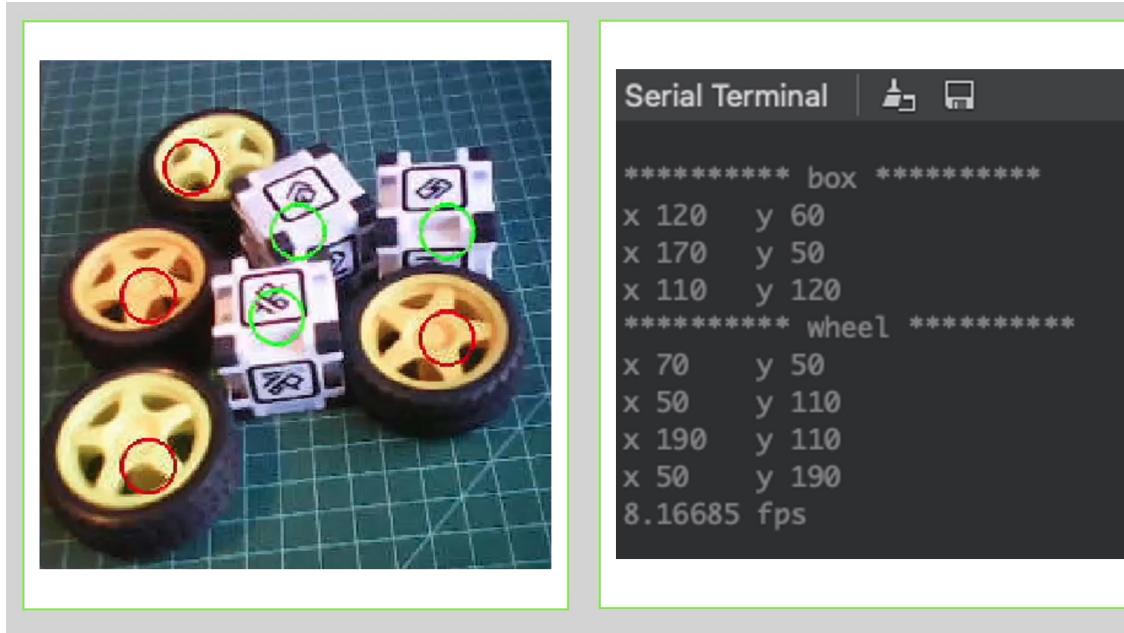
```

Keep the remaining code as it is and press the green Play button to run the code:



On the camera view, we can see the objects with their centroids marked with 12 pixel-fixed circles (each circle has a distinct color, depending on its class). On the Serial Terminal, the model shows the labels detected and their position on the image window (240X240).

Be ware that the coordinate origin is in the upper left corner.



Note that the frames per second rate is around 8 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 5 times higher (around 1.5 fps)

Here is a short video showing the inference results:

<https://youtu.be/JbpoqRp3BbM>

3.8 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, for example, to explore the Nicla doing sensor fusion (camera + microphone) and object detection. This can be very useful on projects involving bees, for example.



4 Audio Feature Engineering



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.

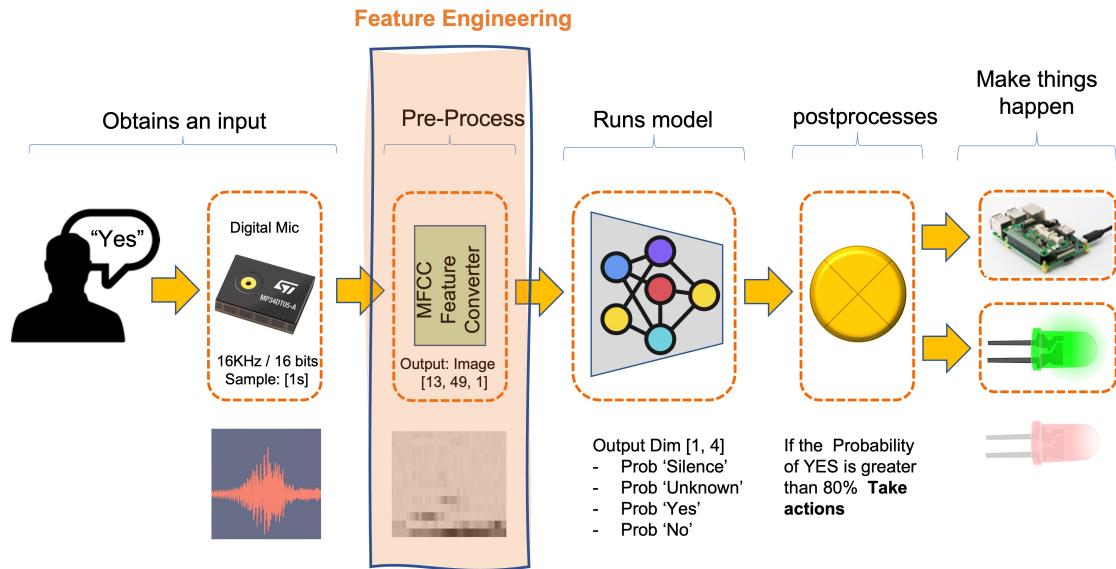
Machine learning models, especially traditional algorithms, don’t understand audio waves. They understand numbers arranged in some meaningful way, i.e., features. These features encapsulate the characteristics of the audio signal, making it easier for models to distinguish between different sounds.

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

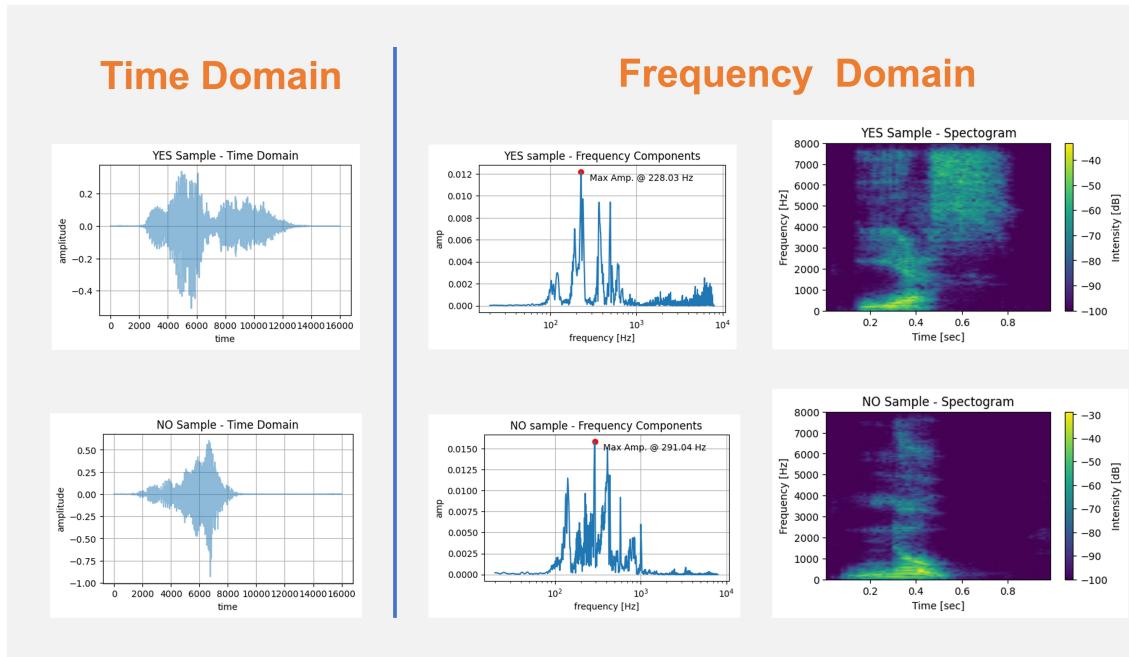
Understanding the basic properties of audio signals is crucial for effective feature extraction and, ultimately, for successfully applying machine learning algorithms in audio classification tasks. Audio signals are complex waveforms that capture fluctuations in air pressure over time. These signals 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.

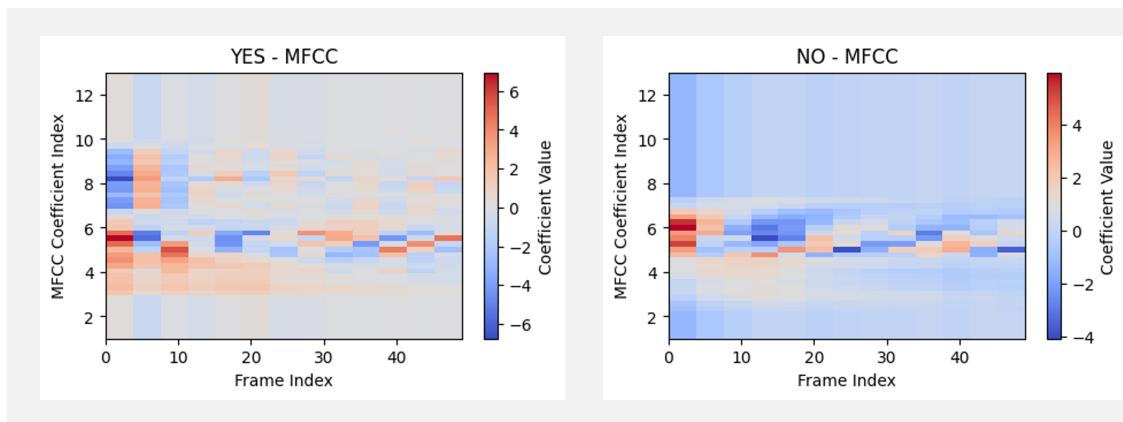
For these reasons, feature extraction techniques such as Mel-frequency Cepstral Coefficients (MFCCs), Mel-Frequency Energies (MFEs), and simple Spectograms are commonly used to transform raw audio data into a more manageable and informative format. These features capture the essential characteristics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

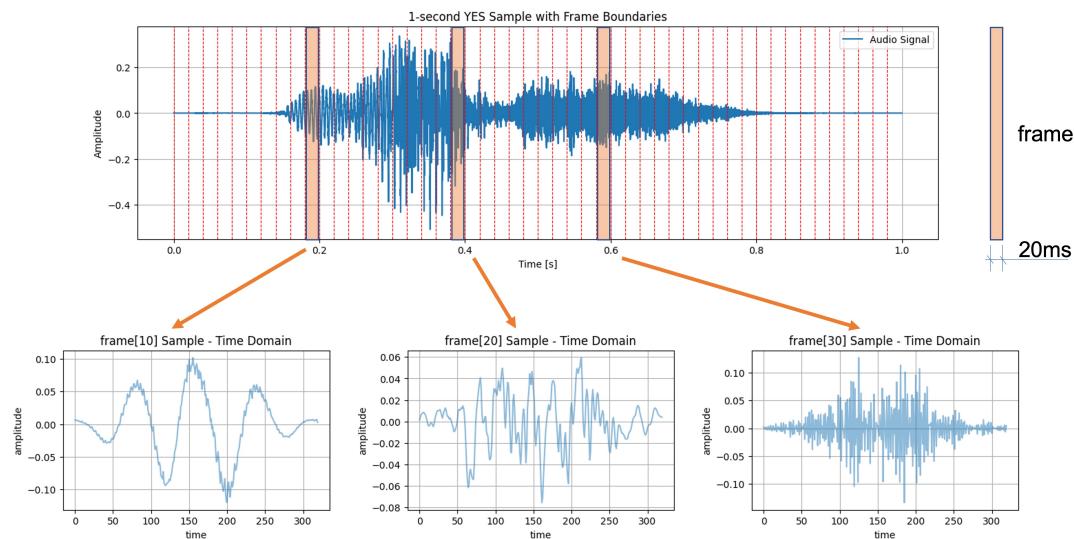
In summary, MFCCs offer a balance of information richness and computational efficiency, making them popular for audio classification tasks, particularly in constrained environments like TinyML.

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):

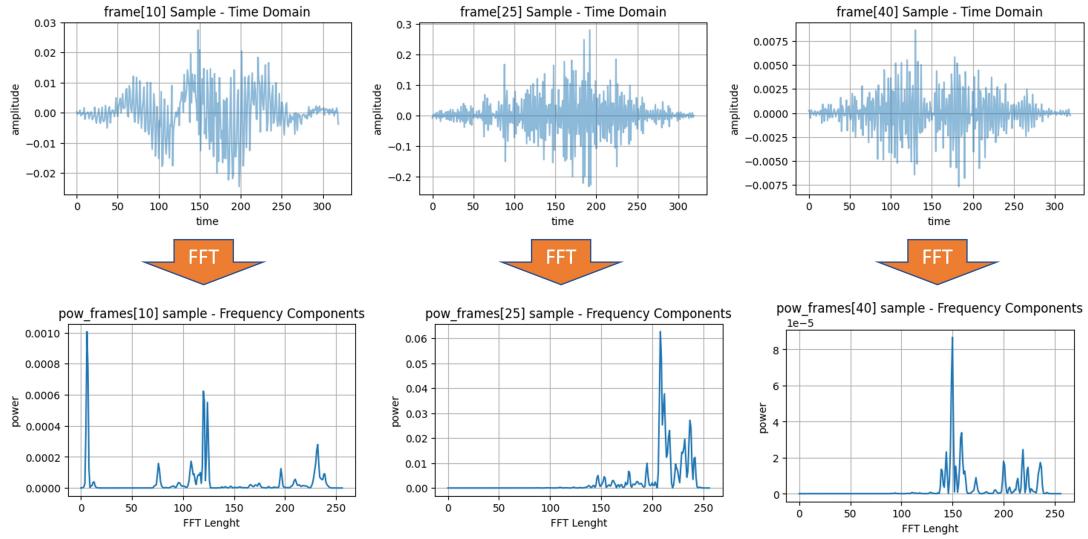


- **Fast Fourier Transform (FFT)** The Fast Fourier Transform (FFT) is applied to each windowed frame to convert it from the time domain to the frequency domain. The FFT gives us a complex-valued representation that includes both magnitude and phase information. However, for MFCCs, only the magnitude 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.

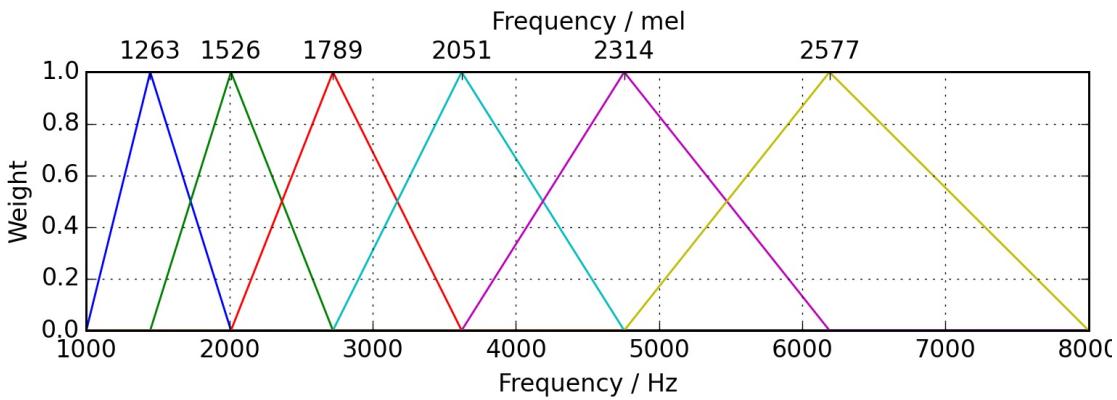
The power spectrum of a signal is defined as

$$P(f) = |F(f)|^2$$
, where $F(f)$ is the Fourier Transform of $x(t)$. By

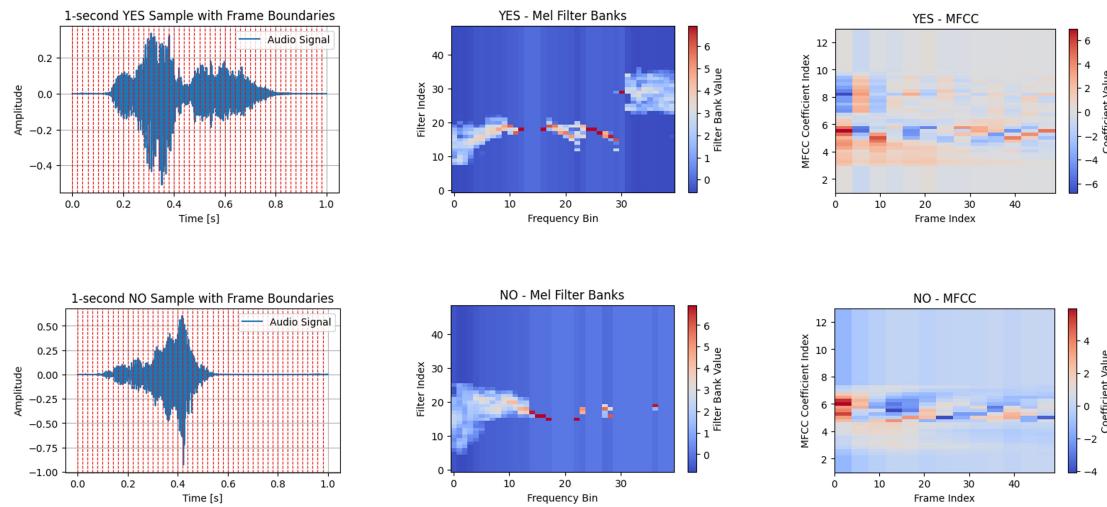
squaring the magnitude of the Fourier Transform, we emphasize *stronger* frequencies over *weaker* ones, thereby capturing more relevant spectral characteristics of the audio signal. This is important in applications like audio classification, speech recognition, and Keyword Spotting (KWS), where the 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)



5.1 Introduction

Having already explored the Nicla Vision board in the *Image Classification* and *Object Detection* applications, we are now shifting our focus to voice-activated applications with a project on Keyword Spotting (KWS).

As introduced in the *Feature Engineering for Audio Classification* Hands-On tutorial, Keyword Spotting (KWS) is integrated into 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 feasible on smaller, low-power devices. This tutorial will guide you through implementing a KWS system using TinyML on the Nicla Vision development board equipped with a digital microphone.

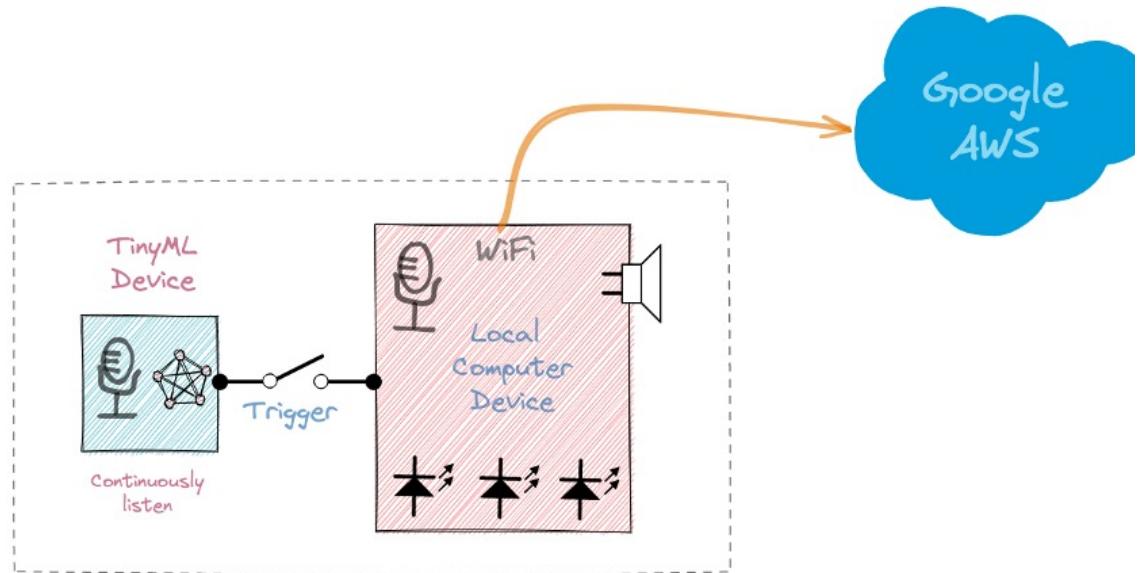
Our model will be designed to recognize keywords that can trigger device wake-up or specific actions, bringing them to life with voice-activated commands.

5.2 How does a voice assistant work?

As said, *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 small microprocessor inside the Echo Dot or Google Home continuously listens, waiting for the keyword to be spotted, using a TinyML model at the edge (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 of a Google Assistant being programmed on a Raspberry Pi (Stage 2), with an Arduino Nano 33 BLE as the tinyML device (Stage 1).

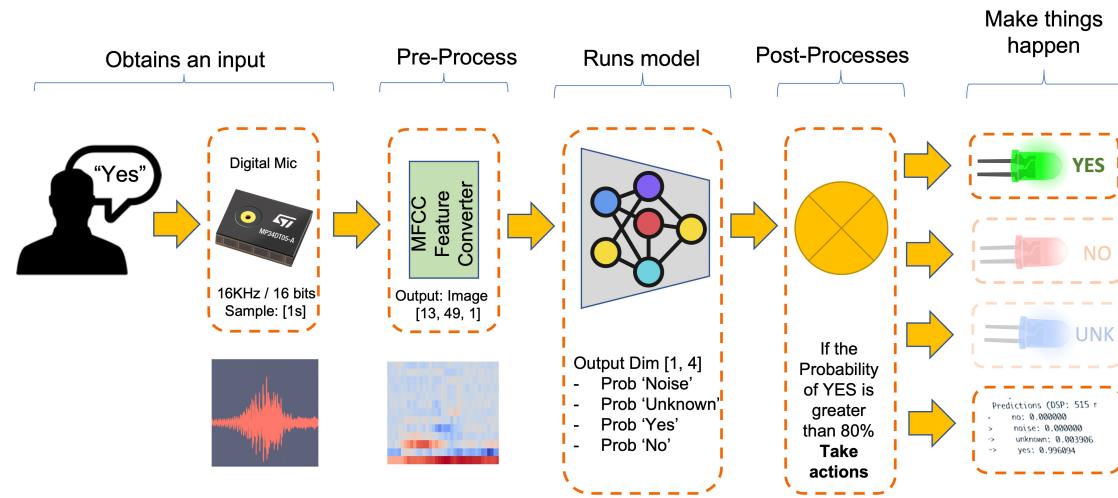
https://youtu.be/e_OPgcnsyvM

To explore the above Google Assistant project, please see the tutorial: [Building an Intelligent Voice Assistant From Scratch](#).

In this KWS project, we will focus on Stage 1 (KWS or Keyword Spotting), where we will use the Nicla Vision, which has a digital microphone that will be used to spot the keyword.

5.3 The KWS Hands-On Project

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



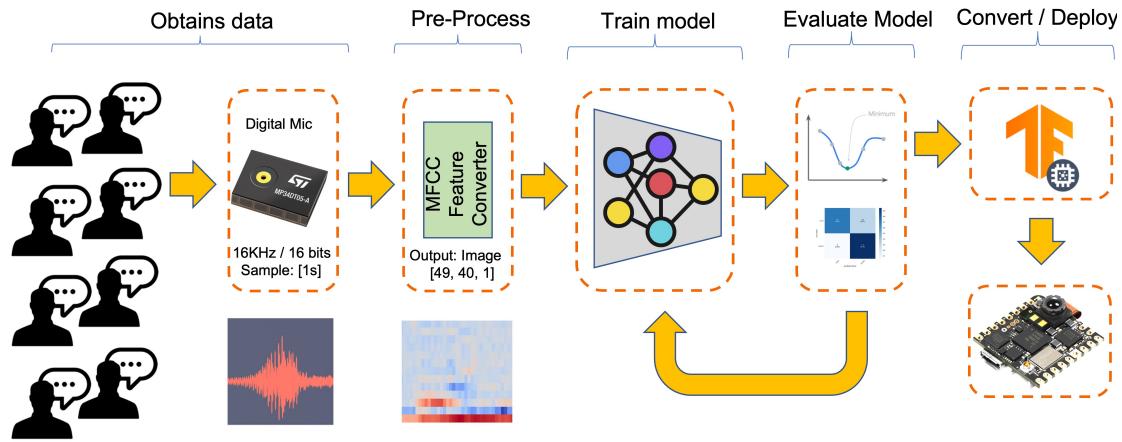
Our KWS application will recognize four classes of sound:

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

For real-world projects, it is always advisable to include other sounds besides the keywords, such as "Noise" (or Background) and "Unknown."

5.3.1 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"):



5.4 Dataset

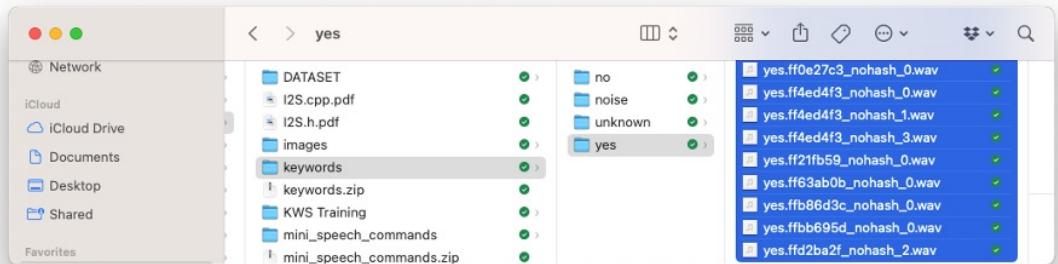
The critical component of any Machine Learning Workflow is the **dataset**. Once we have decided on specific keywords, in our case (*YES* and *NO*), we can take advantage of the dataset developed by Pete Warden, “[Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition](#).” This dataset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

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 to a location of your choice.

5.4.1 Uploading the dataset to the Edge Impulse Studio

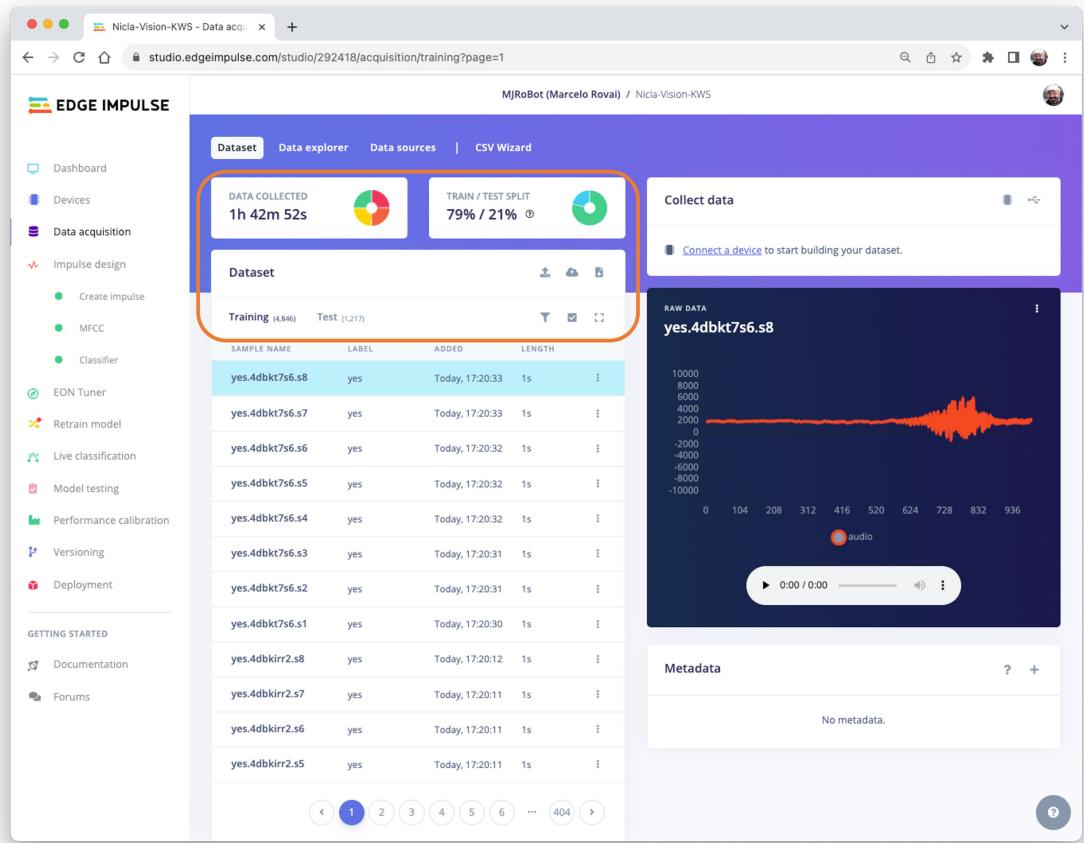
Initiate a new project at Edge Impulse Studio (EIS) and select the [Upload Existing Data](#) tool in the [Data Acquisition](#) section. Choose the files to be uploaded:



Define the Label, select Automatically split between train and test, and Upload data to the EIS. Repeat for all classes.

The screenshot shows the Edge Impulse Studio interface. On the left, there's a sidebar with various project management and data acquisition tools. The main area is titled 'Upload data'. It has sections for 'Upload mode' (set to 'Select individual files'), 'Label' (set to 'Enter label: yes'), and a list of uploaded files. The 'Label' field is highlighted with a red box. The 'Upload data' button at the bottom right is also highlighted with a red box.

The dataset will now appear in the Data acquisition section. Note that the approximately 6,000 samples (1,500 for each class) are split into Train (4,800) and Test (1,200) sets.



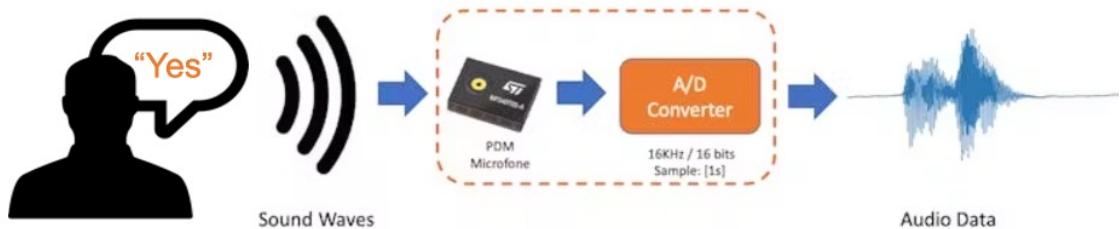
5.4.2 Capturing additional Audio Data

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 is essential. In the case of *sound*, this is optional because what we will classify is, in reality, *audio* data.

The key difference between sound and audio is the type of energy. Sound is mechanical perturbation (longitudinal sound waves) that propagate through a medium, causing variations of pressure in it. Audio is an electrical (analog or digital) signal representing sound.

When we pronounce a keyword, the sound waves should be converted to audio data. The conversion should be done by sampling the signal generated by the microphone at a 16KHz frequency with 16-bit per sample amplitude.

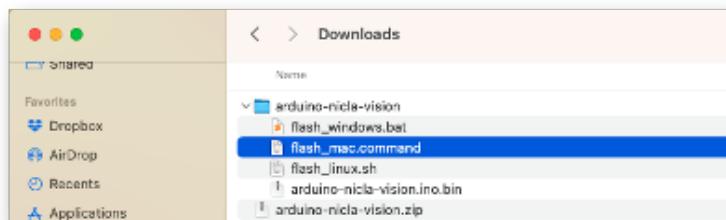
So, any device that can generate audio data with this basic specification (16KHz/16bits) will work fine. As a *device*, we can use the NiclaV, a computer, or even your mobile phone.



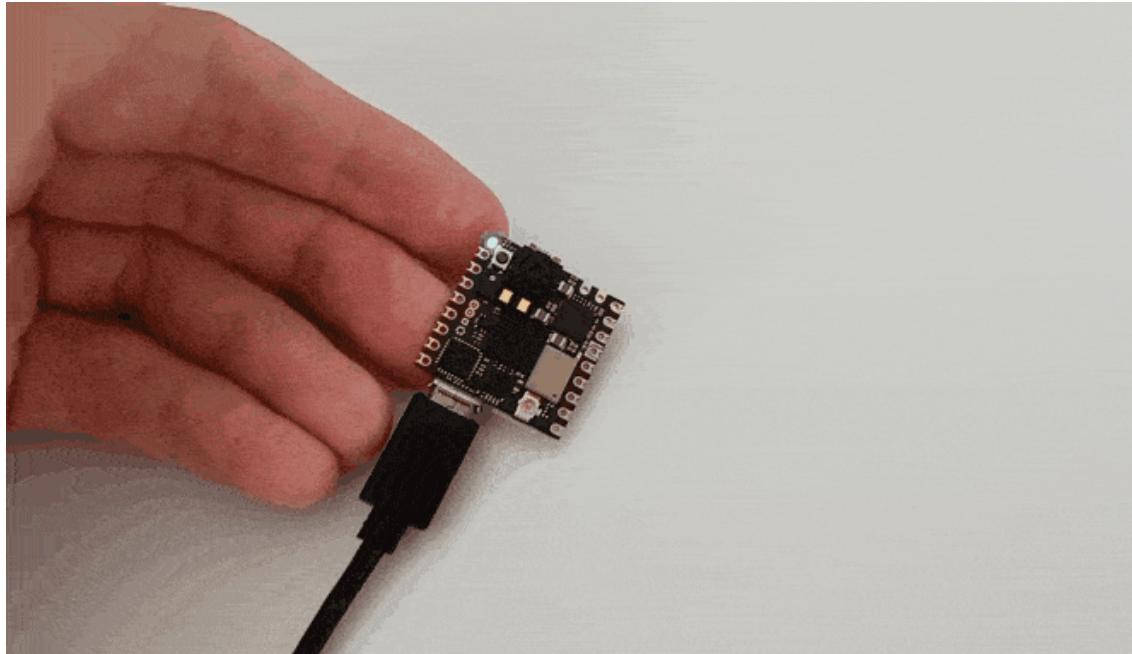
5.4.2.1 Using the NiclaV and the Edge Impulse Studio

As we learned in the chapter *Setup Nicla Vision*, EIS officially supports the Nicla Vision, which simplifies the capture of the data from its sensors, including the microphone. So, please create a new project on EIS and connect the Nicla to it, following these steps:

- Download the last updated [EIS Firmware](#) and unzip it.
- Open the zip file on your computer and select the uploader corresponding to your OS:



- Put the NiclaV in Boot Mode by pressing the reset button twice.



- Upload the binary *arduino-nicla-vision.bin* to your board by running the batch code corresponding to your OS.

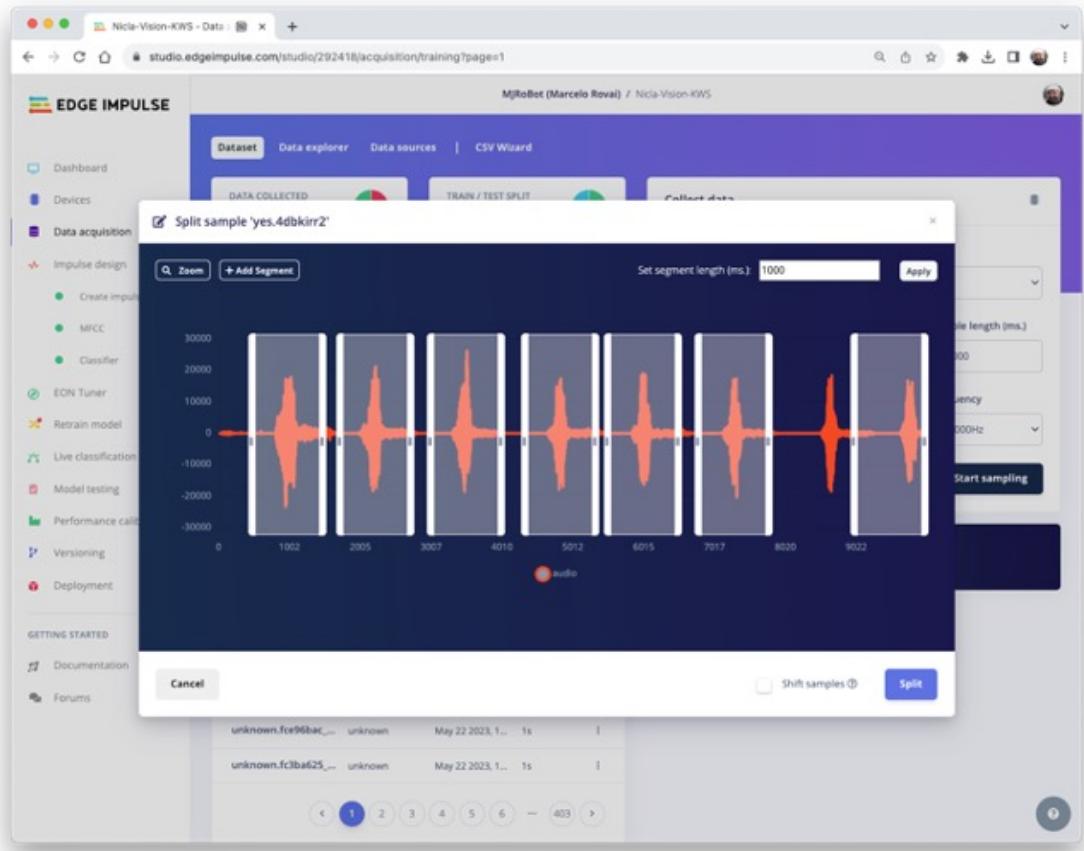
Go to your project on EIS, and on the Data Acquisition tab, select WebUSB. A window will pop up; choose the option that shows that the Nicla is paired and press [Connect].

You can choose which sensor data to pick in the Collect Data section on the Data Acquisition tab. Select: Built-in microphone, define your label (for example, yes), the sampling Frequency[16000Hz], and the Sample length (in milliseconds), for example [10s]. Start sampling.

The screenshot shows the Edge Impulse Studio interface. On the left, there's a sidebar with various options like Dashboard, Devices, Data acquisition, and Performance calibration. The main area is titled 'Dataset' and shows 'DATA COLLECTED 1h 43m 7s' and 'TRAIN / TEST SPLIT 79% / 21%'. Below this is a table of samples with columns for SAMPLE NAME, LABEL, ADDED, and LENGTH. Most samples have a length of 10s, except for one labeled 'unknown'. To the right, there's a 'Collect data' panel with fields for Device (set to '51:18:31:32:37:36'), Label ('yes'), Sample length (ms.) (set to 10000), Sensor ('Built-in microphone'), Frequency (set to 16000Hz), and a 'Start sampling' button. A red box highlights the 'Label' and 'Sample length' fields. Below the collect data panel is a 'RAW DATA' section for a sample named 'yes.4dbkirk2', showing a waveform plot from 0 to 9360 Hz.

Data on Pete's dataset have a length of 1s, but the recorded samples are 10s long and must be split into 1s samples. Click on three dots after the sample name and select Split sample.

A window will pop up with the Split tool.

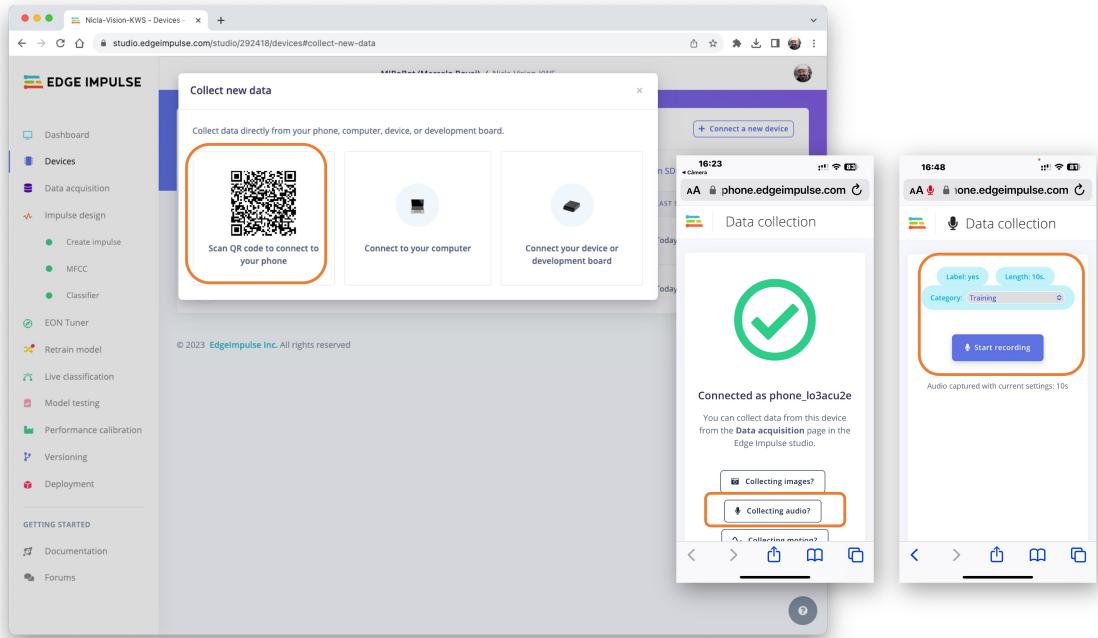


Once inside the tool, split the data into 1-second (1000 ms) records. If necessary, add or remove segments. This procedure should be repeated for all new samples.

5.4.2.2 Using a smartphone and the EI Studio

You can also use your PC or smartphone to capture audio data, using a sampling frequency of 16KHz and a bit depth of 16.

Go to Devices, scan the QR Code using your phone, and click on the link. A data Collection app will appear in your browser. Select Collecting Audio, and define your Label, data capture Length, and Category.



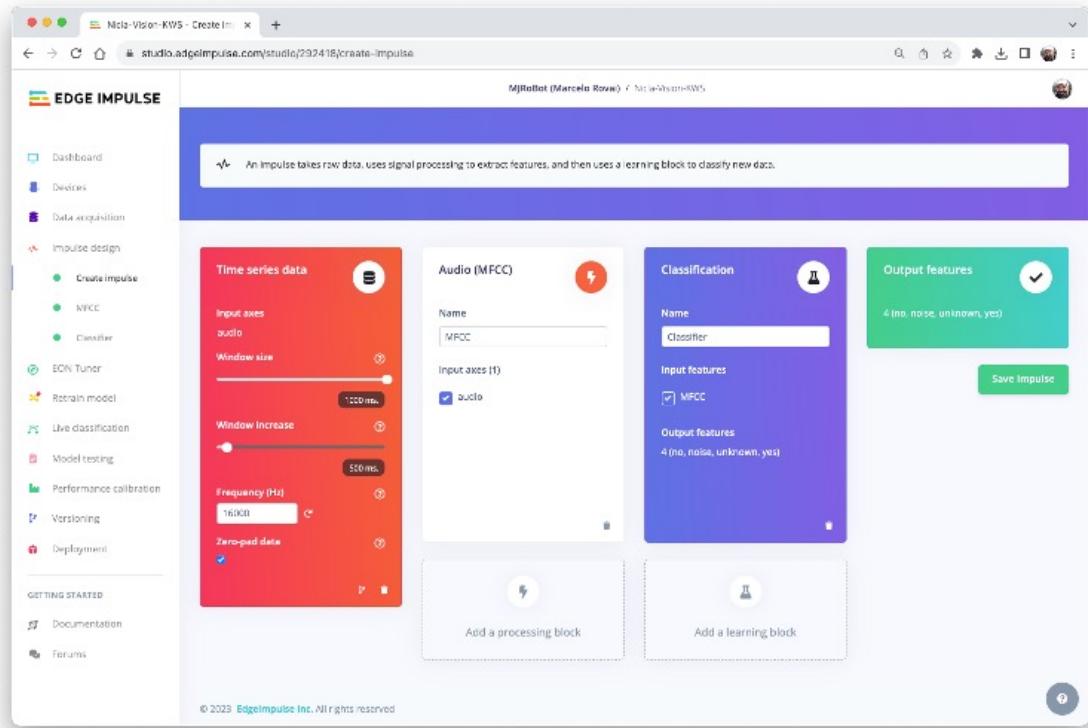
Repeat the same procedure used with the NiclaV.

Note that any app, such as [Audacity](#), can be used for audio recording, provided you use 16KHz/16-bit depth samples.

5.5 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.

5.5.1 Impulse Design



First, we will take the data points with a 1-second window, augmenting the data and sliding that window in 500ms intervals. Note that the option zero-pad data is set. It is essential to fill with ‘zeros’ samples smaller than 1 second (in some cases, some samples can result smaller than the 1000 ms window on the split tool to avoid noise and spikes).

Each 1-second audio sample should be pre-processed and converted to an image (for example, 13 x 49 x 1). As discussed in the *Feature Engineering for Audio Classification* Hands-On tutorial, we will use **Audio (MFCC)**, which extracts features from audio signals using **Mel Frequency Cepstral Coefficients**, which are well suited for the human voice, our case here.

Next, we select the **Classification** block to build our model from scratch using a Convolution Neural Network (CNN).

Alternatively, you can use the **Transfer Learning (Keyword Spotting)** block, which fine-tunes a pre-trained keyword spotting model on your data. This approach has good performance with relatively small keyword datasets.

5.5.2 Pre-Processing (MFCC)

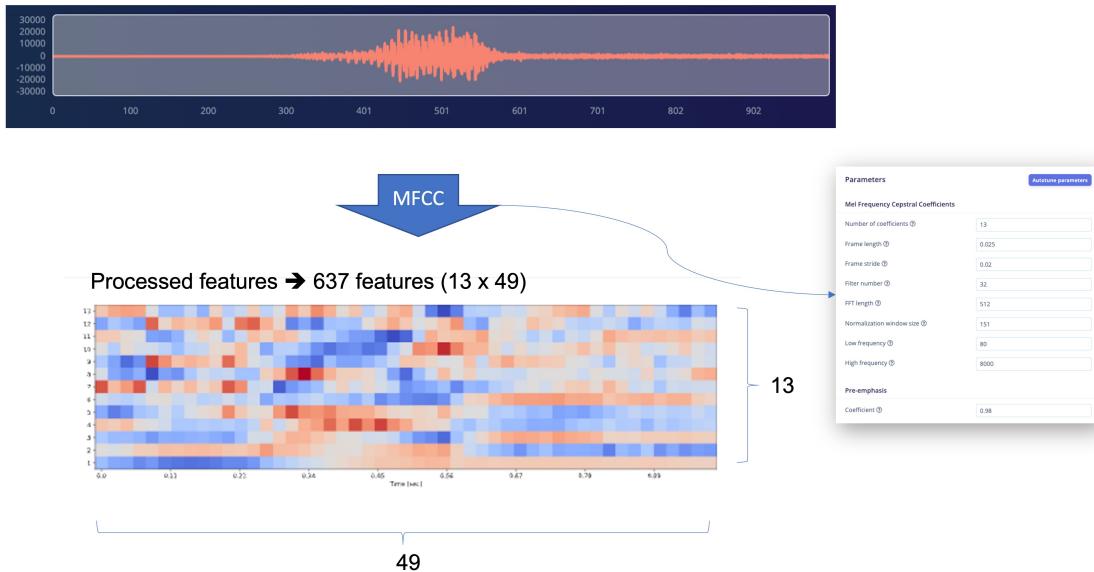
The following step is to create the features to be trained in the next phase:

We could keep the default parameter values, but we will use the DSP Autotune parameters option.

The screenshot shows the Edge Impulse studio interface with the URL <https://studio.edgeimpulse.com/studio/292418/dsp/mfcc/3>. The left sidebar has sections like Dashboard, Devices, Data acquisition, Impulse design, Create impulse, MFCC, Classifier, EON Tuner, Retrain model, Live classification, Model testing, Performance calibration, Versioning, Deployment, Documentation, and Forums. The main area has tabs for Raw data and DSP result. In the Raw data tab, there is a waveform visualization and a button to 'Copy 16000 features to clipboard'. Below it is a 'Raw features' section with a list of numerical values. In the DSP result tab, there is a heatmap titled 'Cepstral Coefficients' and a button to 'Copy 637 features to clipboard'. Below it is a 'Processed features' section with a list of numerical values. The center of the screen contains a 'Parameters' section for the Mel Frequency Cepstral Coefficients block, which includes fields for Number of coefficients (set to 13), Frame length (0.025), Frame stride (0.02), Filter number (32), FFT length (512), Normalization window size (151), Low frequency (80), High frequency (Click to set), and Pre-emphasis (Coefficient 0.98). A blue button labeled 'Autotune parameters' is located in this section. At the bottom left is a link to the page source: <https://studio.edgeimpulse.com/studio/292418/dsp/mfcc/3#>.

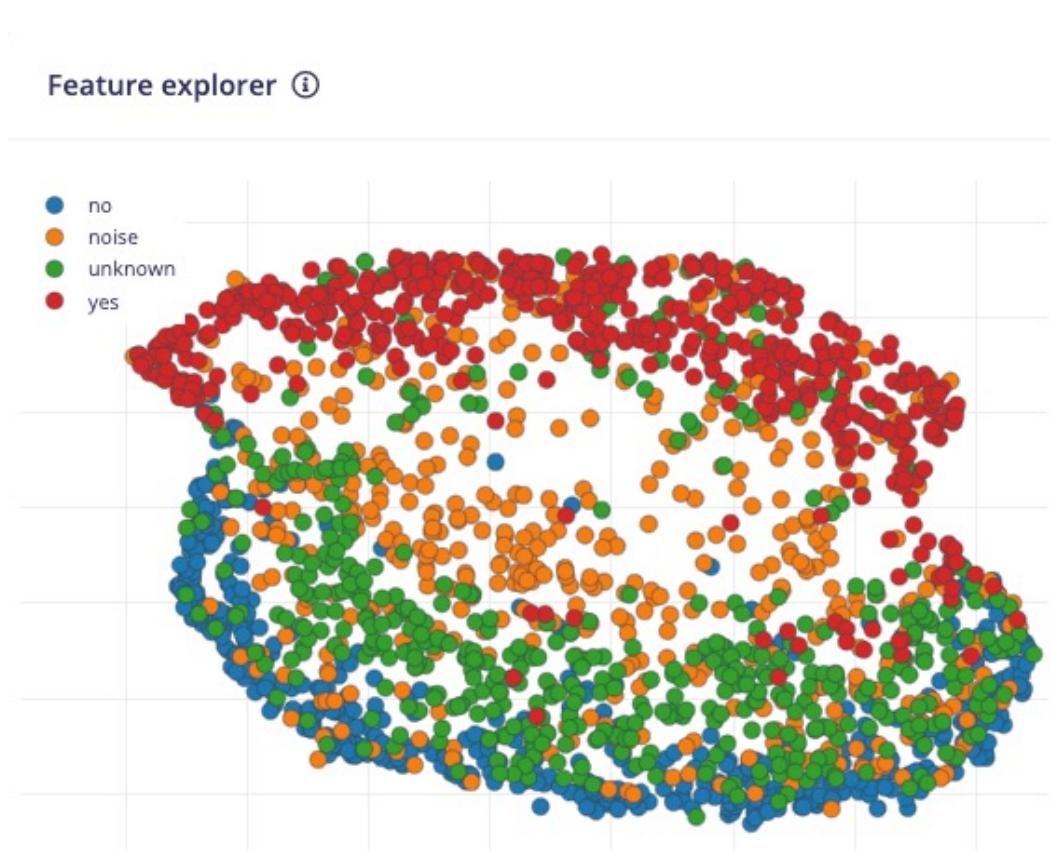
We will take the Raw features (our 1-second, 16KHz sampled audio data) and use the MFCC processing block to calculate the Processed features. For every 16,000 raw features ($16,000 \times 1$ second), we will get 637 processed features (13×49).

Raw data → 16,000 features (1s @ 16KHz)



The result shows that we only used a small amount of memory to pre-process data (16KB) and a latency of 34ms, which is excellent. For example, on an Arduino Nano (Cortex-M4f @ 64MHz), the same pre-process will take around 480ms. The parameters chosen, such as the FFT length [512], will significantly impact the latency.

Now, let's Save parameters and move to the Generated features tab, where the actual features will be generated. Using UMAP, a dimension reduction technique, the Feature explorer shows how the features are distributed on a two-dimensional plot.



The result seems OK, with a visually clear separation between *yes* features (in red) and *no* features (in blue). The *unknown* features seem nearer to the *no space* than the *yes*. This suggests that the keyword *no* has more propensity to false positives.

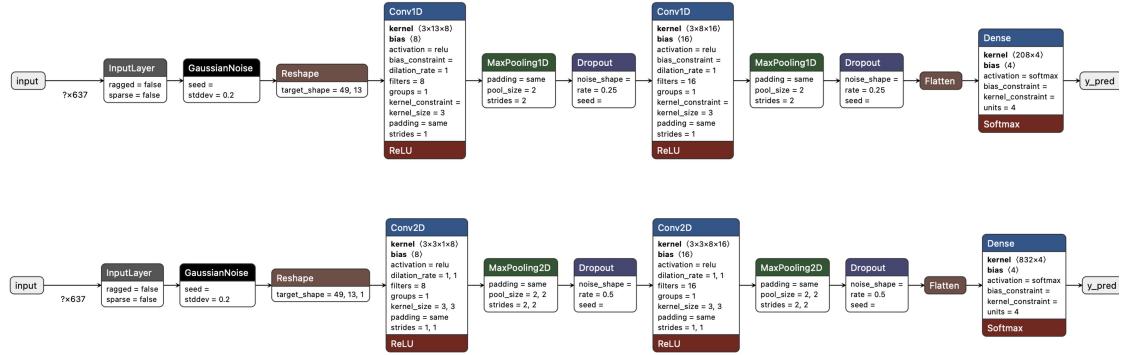
5.5.3 Going under the hood

To understand better how the raw sound is preprocessed, look at the *Feature Engineering for Audio Classification* chapter. You can play with the MFCC features generation by downloading this [notebook](#) from GitHub or [\[Opening it In Colab\]](#)

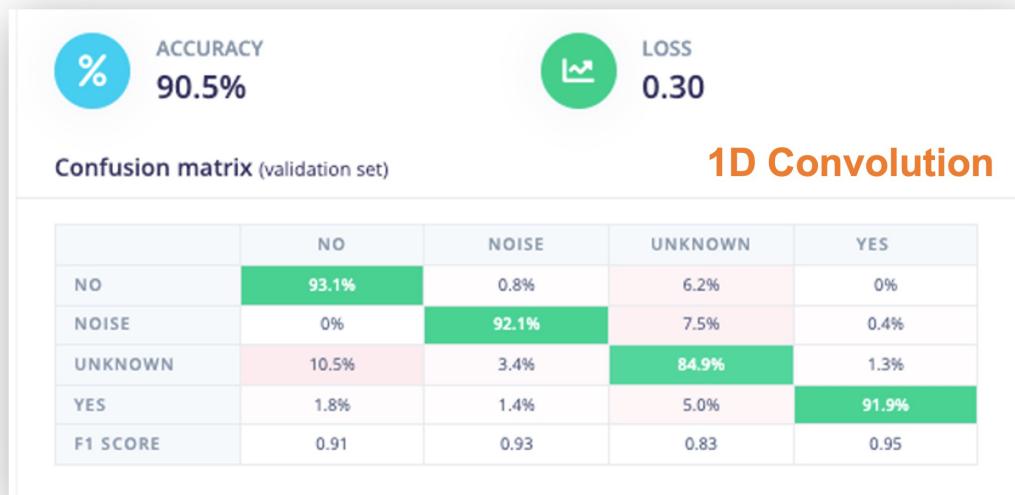
5.6 Model Design and Training

We will use a simple Convolution Neural Network (CNN) model, tested with 1D and 2D convolutions. The basic architecture has two blocks of Convolution + MaxPooling ([8] and [16] filters, respectively) and a Dropout

of [0.25] for the 1D and [0.5] for the 2D. For the last layer, after Flattening, we have [4] neurons, one for each class:



As hyper-parameters, we will have a Learning Rate of [0.005] and a model trained by [100] epochs. We will also include a data augmentation method based on [SpecAugment](#). We trained the 1D and the 2D models with the same hyperparameters. The 1D architecture had a better overall result (90.5% accuracy when compared with 88% of the 2D, so we will use the 1D).



Using 1D convolutions is more efficient because it requires fewer parameters than 2D convolutions, making them more suitable for resource-constrained environments.

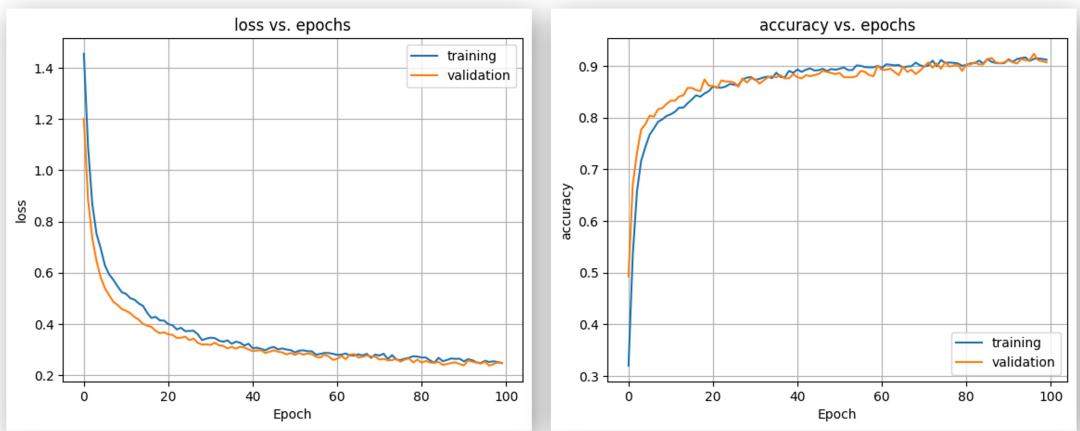
It is also interesting to pay attention to the 1D Confusion Matrix. The F1 Score for yes is 95%, and for no, 91%. That was expected by what we saw with the Feature Explorer (no and unknown at close distance). In trying to improve the result, you can inspect closely the results of the samples with an error.



Listen to the samples that went wrong. For example, for yes, most of the mistakes were related to a yes pronounced as “yeh”. You can acquire additional samples and then retrain your model.

5.6.1 Going under the hood

If you want to understand what is happening “under the hood,” you can download the pre-processed dataset (MFCC training data) from the Dashboard tab and run this [Jupyter Notebook](#), playing with the code or [\[Opening it In Colab\]](#). For example, you can analyze the accuracy by each epoch:



5.7 Testing

Testing the model with the data reserved for training (Test Data), we got an accuracy of approximately 76%.

Model testing results



ACCURACY

75.85%

	NO	NOISE	UNKNOWN	YES	UNCERTAIN
NO	57.8%	1.9%	27.8%	0.2%	12.2%
NOISE	0%	90.2%	2.3%	0.3%	7.2%
UNKNOWN	3.4%	3.7%	77.4%	0.7%	14.8%
YES	0.5%	5.0%	1.0%	82.3%	11.3%
F1 SCORE	0.72	0.89	0.70	0.90	

Inspecting the F1 score, we can see that for YES, we got 0.90, an excellent result since we expect to use this keyword as the primary “trigger” for our KWS project. The worst result (0.70) is for UNKNOWN, which is OK.

For NO, we got 0.72, which was expected, but to improve this result, we can move the samples that were not correctly classified to the training dataset and then repeat the training process.

5.7.1 Live Classification

We can proceed to the project’s next step but also consider that it is possible to perform [Live Classification](#) using the NiclaV or a smartphone to capture live samples, testing the trained model before deployment on our device.

5.8 Deploy and Inference

The EIS will package all the needed libraries, preprocessing functions, and trained models, downloading them to your computer. Go to the Deployment section, select Arduino Library, and at the bottom, choose Quantized (Int8) and press Build.

Configure your deployment

You can deploy your impulse to any device. This makes the model run without an internet connection, minimizes latency, and runs with minimal power consumption. [Read more.](#)

Arduino library X

SELECTED DEPLOYMENT
 **Arduino library**
 An Arduino library with examples that runs on most Arm-based Arduino development boards.

MODEL OPTIMIZATIONS
 Model optimizations can increase on-device performance but may reduce accuracy.

Enable EON™ Compiler Same accuracy, up to 50% less memory. [Learn more](#)

Quantized (int8)		MFCC	CLASSIFIER	TOTAL
Selected ✓		34 ms.	1 ms.	35 ms.
LATENCY		15.6K	3.8K	15.6K
RAM		-	31.2K	-
FLASH				-
ACCURACY				-

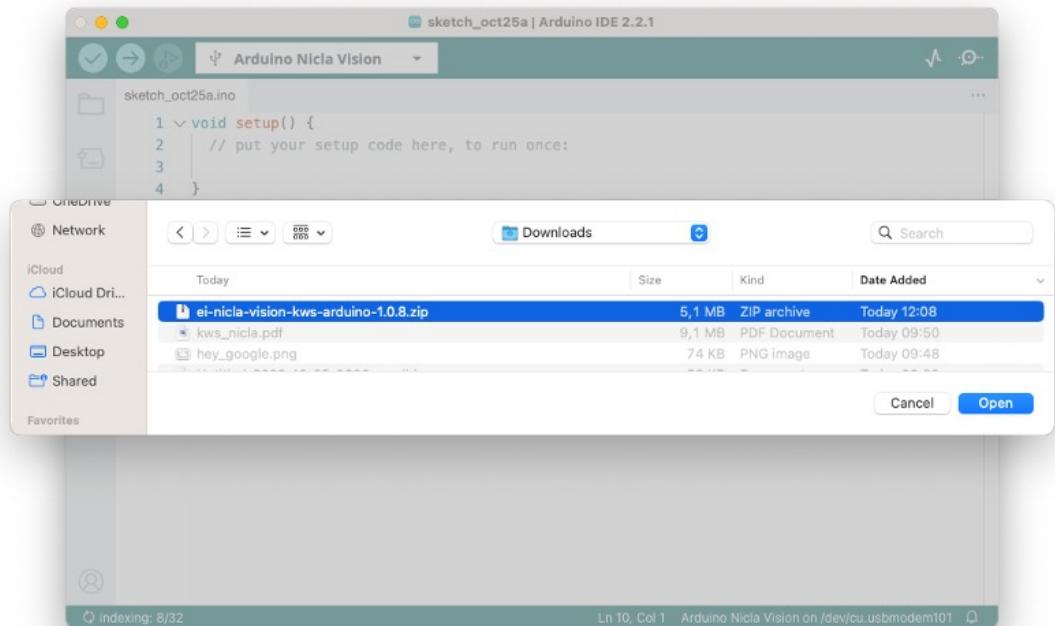
Unoptimized (float32)		MFCC	CLASSIFIER	TOTAL
Select		34 ms.	2 ms.	36 ms.
LATENCY		15.6K	6.2K	15.6K
RAM		-	28.0K	-
FLASH				-
ACCURACY				75.85%

To compare model accuracy, run model testing for all available optimizations. [Run model testing](#)

Estimate for Arduino Nucleo Vision (Cortex-M7 48MHz) - Change target

Build

When the Build button is selected, 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 EIS:



Now, it is time for a real test. We will make inferences while completely disconnected from the EIS. Let's use the NiclaV code example created when we deployed the Arduino Library.

In your Arduino IDE, go to the File/Examples tab, look for your project, and select `nicla-vision/nicla-vision_microphone` (or `nicla-vision_microphone_continuous`)



Press the reset button twice to put the NiclaV in boot mode, upload the sketch to your board, and test some real inferences:



5.9 Post-processing

Now that we know the model is working since it detects our keywords, let's modify the code to see the result with the NiclaV completely offline (disconnected from the PC and powered by a battery, a power bank, or an independent 5V power supply).

The idea is that whenever the keyword YES is detected, the Green LED will light; if a NO is heard, the Red LED will light, if it is a UNKNOW, the Blue LED will light; and in the presence of noise (No Keyword), the LEDs will be OFF.

We should modify one of the code examples. Let's do it now with the `nicla-vision_microphone_continuous`.

Start with initializing the LEDs:

```
...
void setup()
{
    // Once you finish debugging your code, you can comment or delete the
    Serial.begin(115200);
    while (!Serial);
    Serial.println("Inferencing - Nicla Vision KWS with LEDs");

    // Pins for the built-in RGB LEDs on the Arduino NiclaV
    pinMode(LED_R, OUTPUT);
    pinMode(LED_G, OUTPUT);
    pinMode(LED_B, OUTPUT);

    // Ensure the LEDs are OFF by default.
    // Note: The RGB LEDs on the Arduino Nicla Vision
    // are ON when the pin is LOW, OFF when HIGH.
    digitalWrite(LED_R, HIGH);
    digitalWrite(LED_G, HIGH);
    digitalWrite(LED_B, HIGH);

    ...
}
```

Create two functions, turn_off_leds() function , to turn off all RGB LEDs

```
/**  
 * @brief      turn_off_leds function - turn-off all RGB LEDs  
 */  
void turn_off_leds(){  
    digitalWrite(LED_R, HIGH);  
    digitalWrite(LED_G, HIGH);  
    digitalWrite(LED_B, HIGH);  
}
```

Another turn_on_led() function is used to turn on the RGB LEDs according to the most probable result of the classifier.

```
/**  
 * @brief      turn_on_leds function used to turn on the RGB LEDs  
 * @param[in]  pred_index  
 *             no:      [0] ==> Red ON  
 *             noise:   [1] ==> ALL OFF  
 *             unknown: [2] ==> Blue ON  
 *             Yes:     [3] ==> Green ON  
 */  
void turn_on_leds(int pred_index) {  
    switch (pred_index)  
    {  
        case 0:  
            turn_off_leds();  
            digitalWrite(LED_R, LOW);  
            break;  
  
        case 1:  
            turn_off_leds();  
            break;  
  
        case 2:  
            turn_off_leds();  
            digitalWrite(LED_B, LOW);  
            break;  
  
        case 3:  
            turn_off_leds();  
            digitalWrite(LED_G, LOW);  
            break;  
    }  
}
```

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

```
...
```

```

if (++print_results >= (EI_CLASSIFIER_SLICES_PER_MODEL_WINDOW)) {
    // 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");

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

    for (size_t ix = 0; ix < EI_CLASSIFIER_LABEL_COUNT; ix++) {
        if (result.classification[ix].value > pred_value){
            pred_index = ix;
            pred_value = result.classification[ix].value;
        }
        // ei_printf("    %s: ", result.classification[ix].label);
        // ei_printf_float(result.classification[ix].value);
        // ei_printf("\n");
    }
    ei_printf(" PREDICTION: ==> %s with probability %.2f\n",
              result.classification[pred_index].label, pred_value);
    turn_on_leds (pred_index);

#if EI_CLASSIFIER_HAS_ANOMALY == 1
    ei_printf("    anomaly score: ");
    ei_printf_float(result.anomaly);
    ei_printf("\n");
#endif

    print_results = 0;
}
}

...

```

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 Green LED will be ON whenever the keyword YES is detected, the Red will light for a NO, and any other word will turn on the Blue LED. All the LEDs should be off if silence or background noise is present. Remember that the same procedure can “trigger” an external device to perform a desired action instead of turning on an LED, as we saw in the introduction.

<https://youtu.be/25Rd76OTXLY>

5.10 Conclusion

You will find the notebooks and codes used in this hands-on tutorial on the [GitHub](#) repository.

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, Gunshot)
- **Industry** (Anomaly Detection)
- **Medical** (Snore, Cough, Pulmonary diseases)
- **Nature** (Beehive control, insect sound, pouching mitigation)

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-ESTI01 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, Daniel Situnayake
- “TinyML Cookbook” 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, Jenny Plunkett
- “MACHINE LEARNING SYSTEMS for TinyML” e-book extension from CS249r course at Harvard University

Projects Repository

- Edge Impulse Expert Network

TinyML4D

WALC 2023 Applied AI Track 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.



TINYML4D

Table of Contents

Preface	2
Acknowledgments	3
Introduction	4
1 Setup Nicla Vision	6
1.1 Introduction	6
1.2 Hardware	7
1.2.1 Two Parallel Cores	7
1.2.2 Memory	8
1.2.3 Sensors	9
1.3 Arduino IDE Installation	9
1.3.1 Testing the Microphone	10
1.3.2 Testing the IMU	11
1.3.3 Testing the ToF (Time of Flight) Sensor	12
1.3.4 Testing the Camera	14
1.4 Installing the OpenMV IDE	14
1.5 Connecting the Nicla Vision to Edge Impulse Studio	22
1.6 Expanding the Nicla Vision Board (optional)	25
1.7 Conclusion	30
2 CV on Nicla Vision	31
2.1 Introduction	31
2.2 Computer Vision	33
2.3 Image Classification Project Goal	33
2.4 Data Collection	34
2.4.1 Collecting Dataset with OpenMV IDE	34
2.5 Training the model with Edge Impulse Studio	38
2.6 Dataset	38
2.7 The Impulse Design	43

2.7.1 Image Pre-Processing	45
2.7.2 Model Design	47
2.8 Model Training	49
2.9 Model Testing	50
2.10 Deploying the model	52
2.10.1 Arduino Library	53
2.10.2 OpenMV	54
2.11 Image Classification (non-official) Benchmark	65
2.12 Conclusion	67
3 Object Detection	68
3.1 Introduction	68
3.1.1 Object Detection versus Image Classification	69
3.1.2 An innovative solution for Object Detection: FOMO	71
3.2 The Object Detection Project Goal	72
3.3 Data Collection	73
3.3.1 Collecting Dataset with OpenMV IDE	73
3.4 Edge Impulse Studio	75
3.4.1 Setup the project	75
3.4.2 Uploading the unlabeled data	77
3.4.3 Labeling the Dataset	80
3.5 The Impulse Design	81
3.5.1 Preprocessing all dataset	82
3.6 Model Design, Training, and Test	84
3.6.1 Test model with “Live Classification”	87
3.7 Deploying the Model	89
3.8 Conclusion	93
4 Audio Feature Engineering	95
4.1 Introduction	95
4.2 The KWS	96
4.3 Introduction to Audio Signals	98
4.3.1 Why Not Raw Audio?	99

4.4 Introduction to MFCCs	101
4.4.1 What are MFCCs?	101
4.4.2 Why are MFCCs important?	102
4.4.3 Computing MFCCs	102
4.5 Hands-On using Python	105
4.6 Conclusion	106
4.6.1 What Feature Extraction technique should we use?	106
5 Keyword Spotting (KWS)	108
5.1 Introduction	108
5.2 How does a voice assistant work?	109
5.3 The KWS Hands-On Project	110
5.3.1 The Machine Learning workflow	111
5.4 Dataset	112
5.4.1 Uploading the dataset to the Edge Impulse Studio	112
5.4.2 Capturing additional Audio Data	114
5.5 Creating Impulse (Pre-Process / Model definition)	119
5.5.1 Impulse Design	119
5.5.2 Pre-Processing (MFCC)	121
5.5.3 Going under the hood	123
5.6 Model Design and Training	123
5.6.1 Going under the hood	126
5.7 Testing	126
5.7.1 Live Classification	127
5.8 Deploy and Inference	127
5.9 Post-processing	130
5.10 Conclusion	133
References	134
To learn more:	134
Online Courses	134
Books	134
Projects Repository	134

