TinyML Made Easy

Hands-On with the Nicla Vision

Marcelo Rovai

2025-04-10

Table of contents

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

Ermanno Pietrosemoli, President Fundación Escuela Latinoamericana de Redes

November 2023.

Acknowledgments

We extend our deepest gratitude to the entire TinyML4D Academic Network, comprised of distinguished professors, researchers, and professionals. Notable contributions from Marco Zennaro, 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 the extracting of "machine intelligence" from the physical world (where the data is generated).

TinyML Made Ease is a foundational text designed to facilitate the understanding and application of Embedded Machine Learning, or TinyML. In an era where embedded devices boast ultra-low power consumption—measured in mere milliwatts—and machine learning frameworks such as TensorFlow Lite for Microcontrollers (TF Lite Micro) become increasingly tailored for embedded applications, the intersection of AI and IoT is rapidly expanding. This book demystifies the integration of AI capabilities into these devices, paving the way for a wide-reaching adoption of AI-enabled IoT, or "AioT."

About this Book

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

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

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

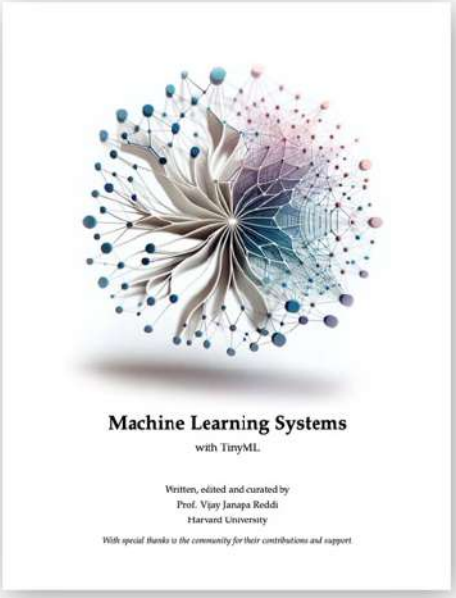
- Microcontroller setup and sensor tests,
- Image classification,
- Dataset labeling and Object detection,
- Audio processing and keyword spotting,
- Time-series data and Digital Signal Processing,
- Motion classification and Anomaly detection.

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

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

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

The content of this book is also part of the open book Machine Learning Systems, which we invite you to read.



Machine Learning Systems

with TinyML

Written, edited and curated by
Prof. Vijay Janapa Reddi
Harvard University

With special thanks to the community for their contributions and support

Supplementary Online Resources

Setup Nicla Vision

- Micropython codes
- Arduino Codes

Image Classification

- Micropython codes
- Dataset
- Edge Impulse Project

Object Detection

- Edge Impulse Project

Audio Feature Engineering

- Audio_Data_Analysis Colab Notebook

Keyword Spotting (KWS)

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

DSP Spectral Features

- DSP - Spectral Features Colab Notebook

Motion Classification and Anomaly Detection)

- Arduino Code
- Edge_Impulse_Spectral_Features_Block Colab Notebook
- Edge Impulse Project

Setup

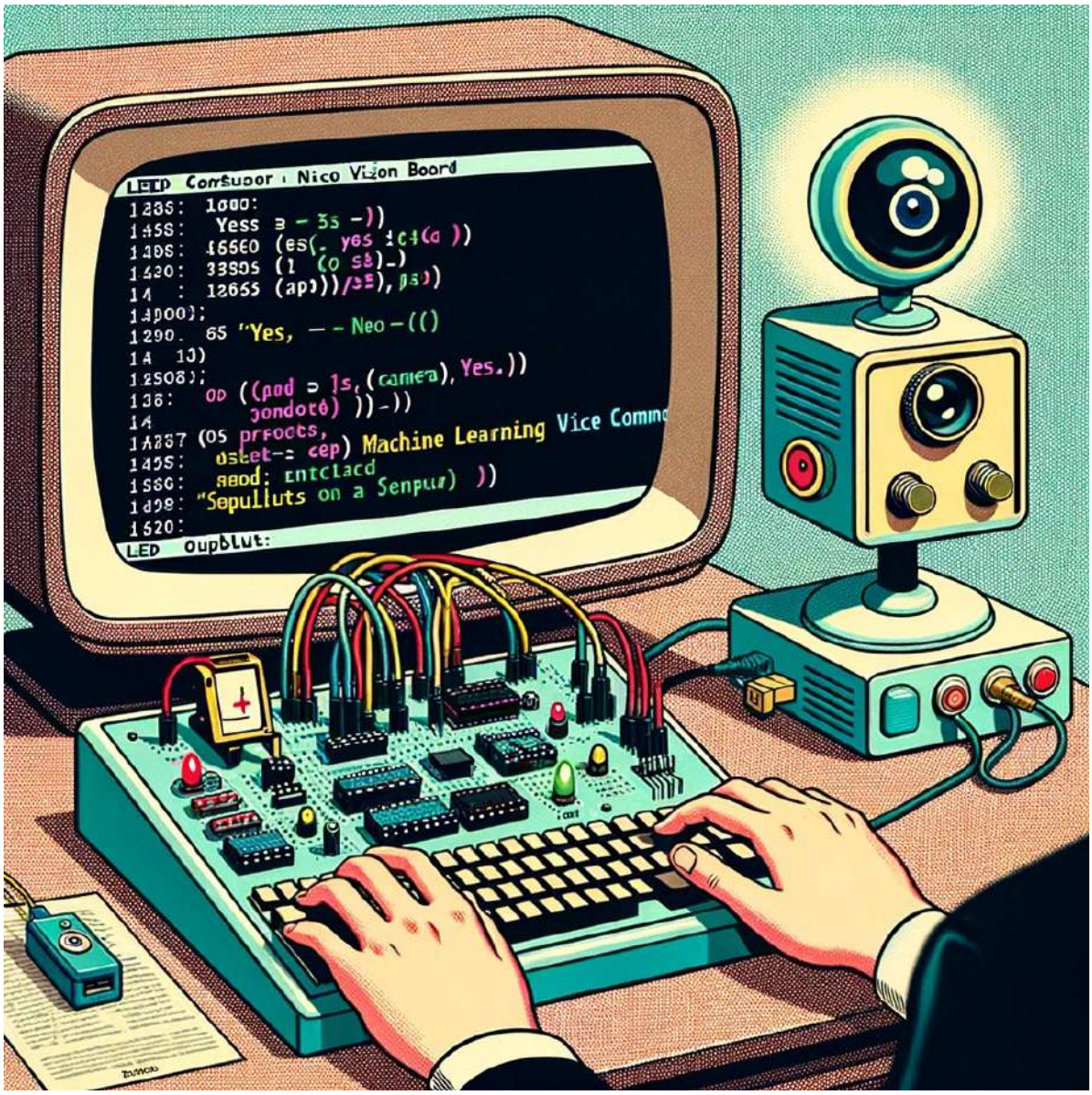
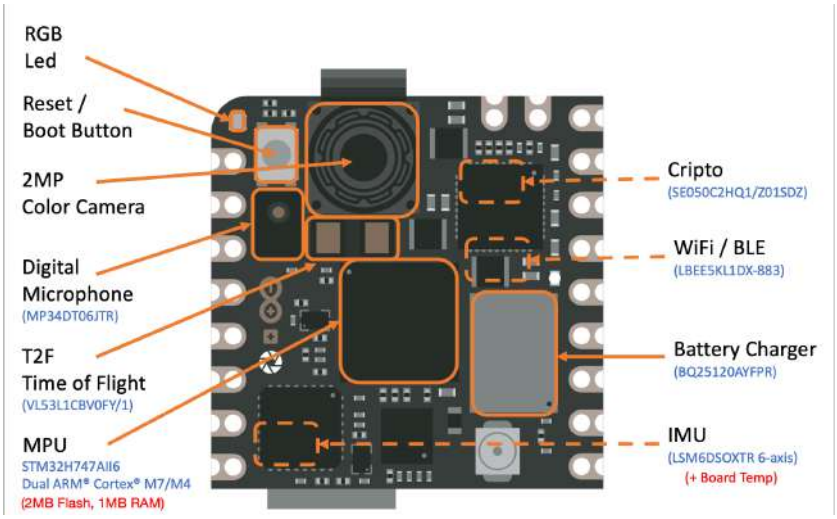


Figure 1: *DALL · E 3 Prompt: Illustration reminiscent of a 1950s cartoon where the Arduino NICLA VISION board, equipped with various sensors including a camera, is the focal point on an old-fashioned desk. In the background, a computer screen with rounded edges displays the Arduino IDE. The code is related to LED configurations and machine learning voice command detection. Outputs on the Serial Monitor explicitly display the words 'yes' and 'no'.*

Overview

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). At the same time, the other executes low-level operations like controlling a motor and communicating or acting as a user interface. The onboard wireless module allows the simultaneous management of WiFi and Bluetooth Low Energy (BLE) connectivity.

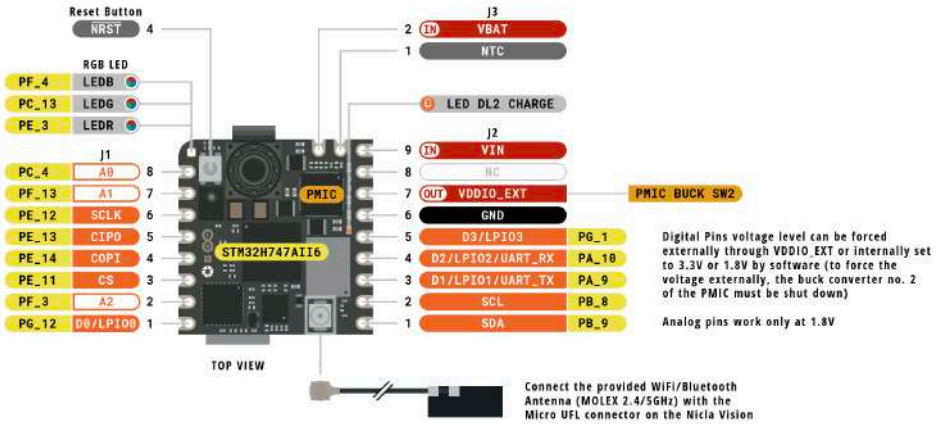


Hardware

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



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

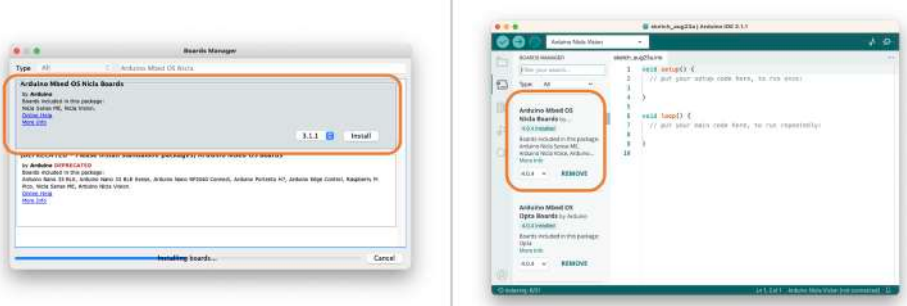
Sensors

- **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 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.

Arduino IDE Installation

Start connecting the board (*micro USB*) 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.



Arduino IDE 1.18 Arduino IDE 2.X

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!

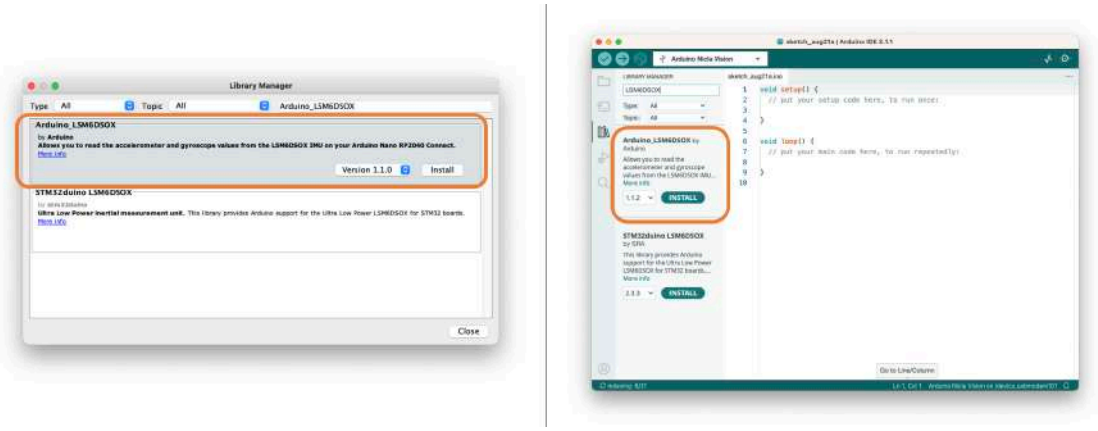
Testing the Microphone

On Arduino IDE, go to `Examples > PDM > PDMSerialPlotter`, open it, 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.

Testing the IMU

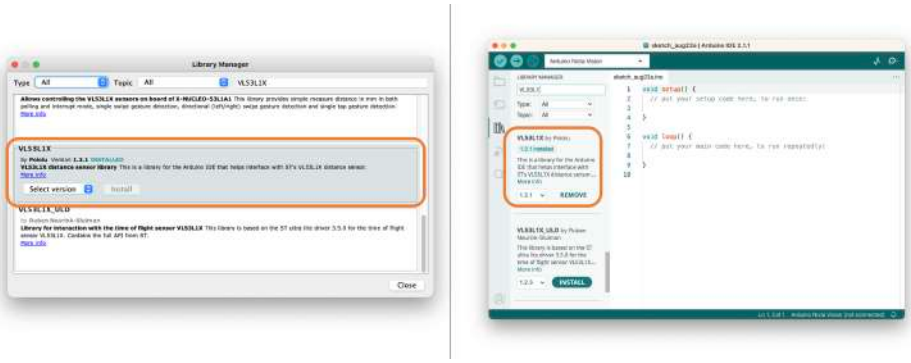
Before testing the IMU, it will be necessary to install the LSM6DSOX library. To do so, go to Library Manager and look for LSM6DSOX. Install the library provided by Arduino:



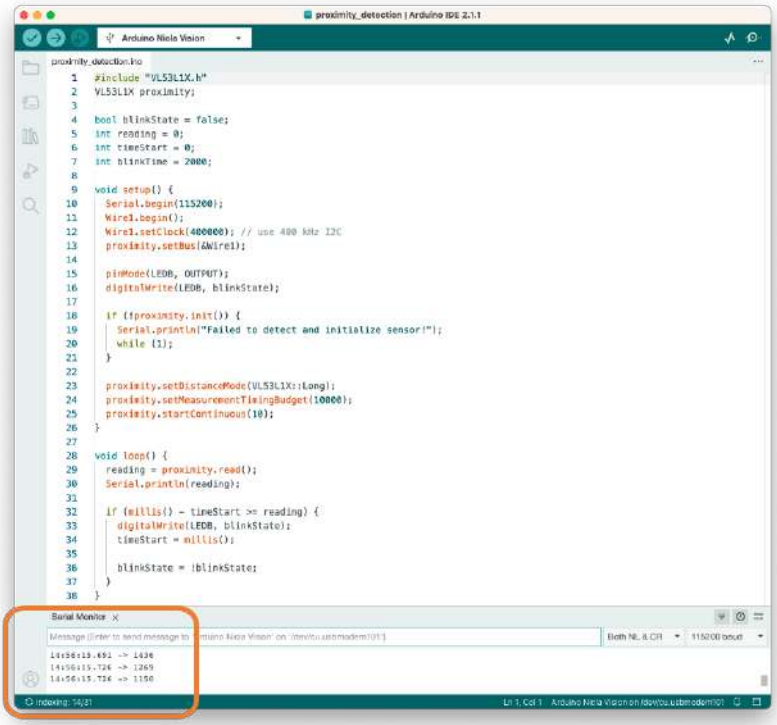
Next, go to `Examples > Arduino_LSM6DSOX > SimpleAccelerometer` and run the accelerometer test (you can also run Gyro and board temperature):

Testing the ToF (Time of Flight) Sensor

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



Next, run the sketch proximity_detection.ino:



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

Testing the Camera

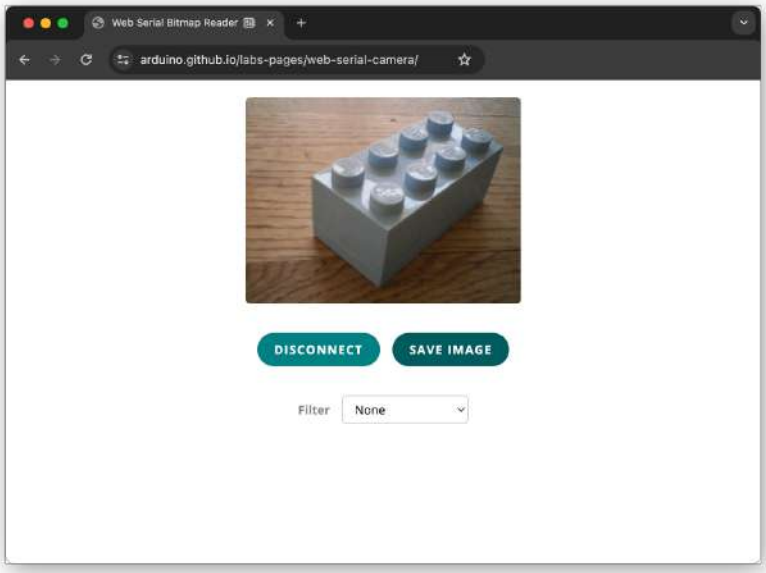
We can also test the camera using, for example, the code provided on `Examples > Camera > CameraCaptureRawBytes`. We cannot see the image directly, but we can get the raw image data generated by the camera.

We can use the `Web Serial Camera` (API) to see the image generated by the camera. This web application streams the camera image over Web Serial from camera-equipped Arduino boards.

The Web Serial Camera example shows you how to send image data over the wire from your Arduino board and how to unpack the data in JavaScript for rendering. In addition, in the source code of the web application, we can find some example image filters that show us how to manipulate pixel data to achieve visual effects.

The **Arduino sketch** (CameraCaptureWebSerial) for sending the camera image data can be found here and is also directly available from the "`Examples→Camera`" menu in the Arduino IDE when selecting the Nicla board.

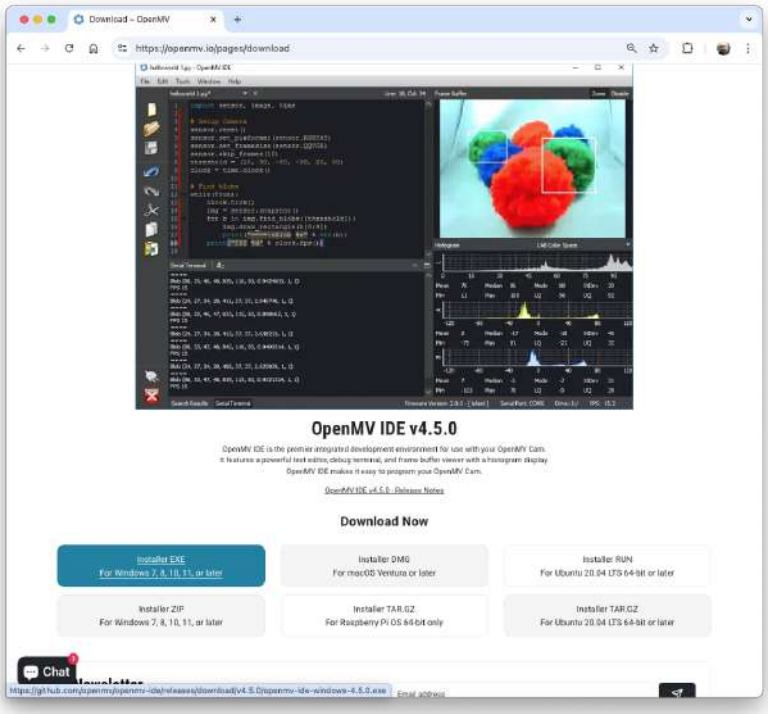
The **web application** for displaying the camera image can be accessed here. We may also look at [this tutorial, which explains the setup in more detail.



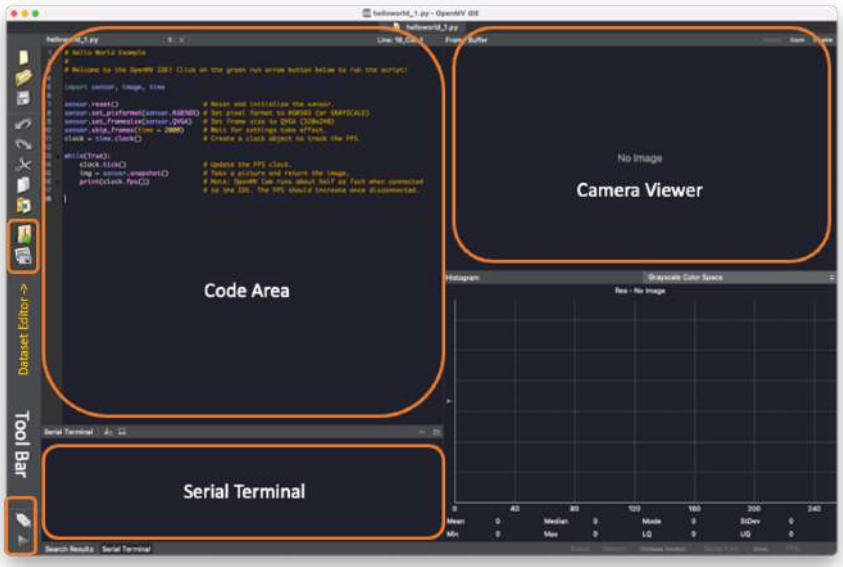
Installing the OpenMV IDE

OpenMV IDE is the premier integrated development environment with OpenMV cameras, similar to 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 > helloword.py`



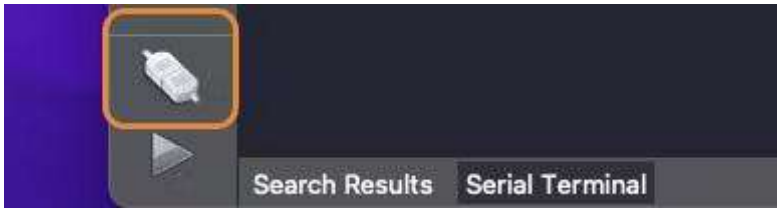
Any messages sent through a serial connection (using print() or error messages) will be displayed on the **Serial Terminal** during run time. The image captured by a camera will be displayed in the **Camera Viewer** Area (or Frame Buffer) and in the Histogram area, immediately below the Camera Viewer.

Updating the Bootloader

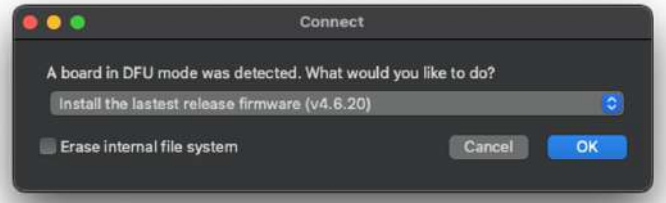
Before connecting the Nicla to the OpenMV IDE, ensure you have the latest bootloader version. Go to your Arduino IDE, select the Nicla board, and open the sketch on `Examples > STM_32H747_System STM32H747_manageBootloader`. Upload the code to your board. The Serial Monitor will guide you.

Installing the Firmware

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.



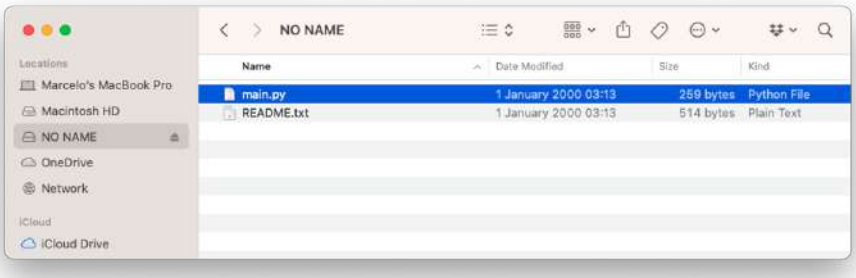
You can leave the option `Erase internal file system` unselected and click `[OK]`.

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.

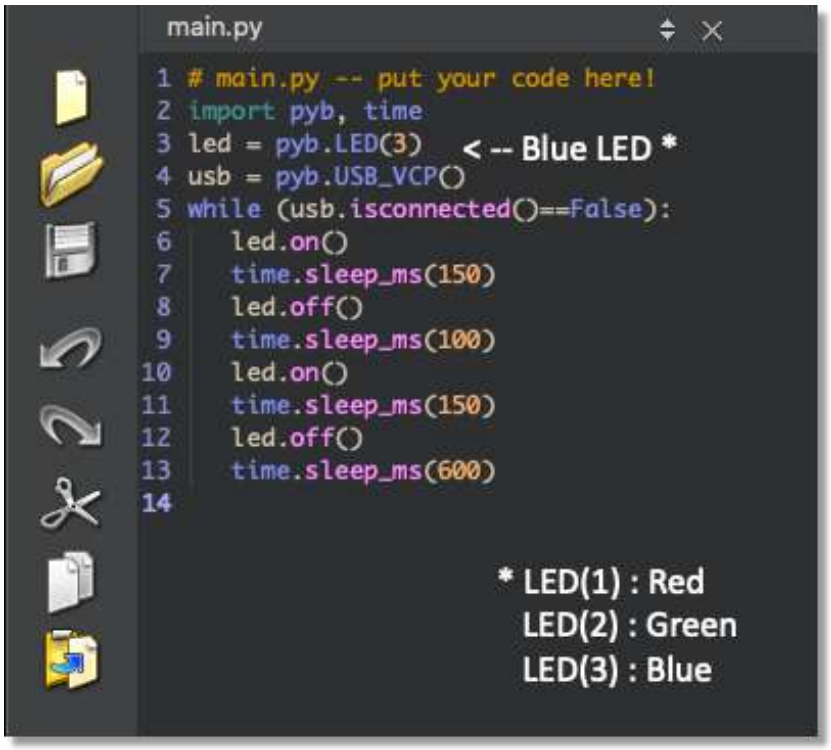
Wait until the green LED stops flashing and fading. When the process ends, you will see a message saying, "DFU firmware update complete!". Press `[OK]`.

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, the main.py script stored on it automatically executes. You can load the main.py code on the IDE (`File > Open File...`).

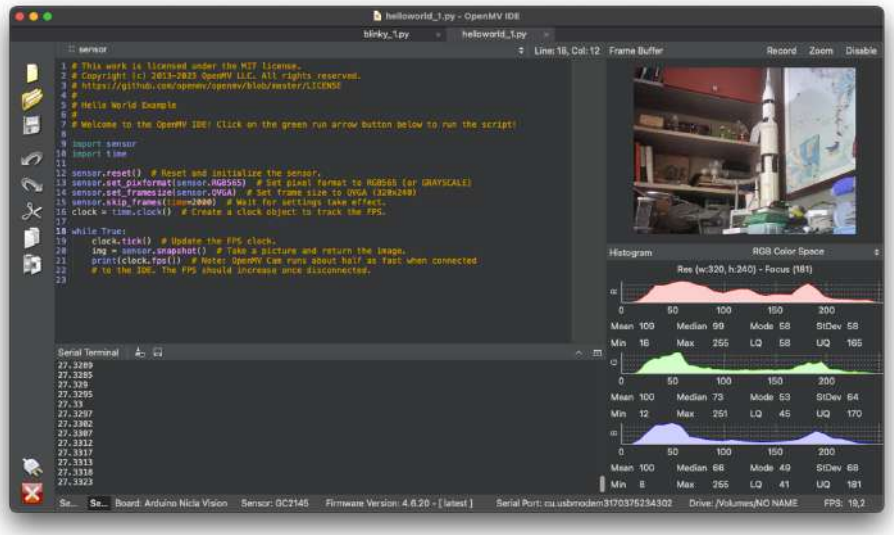


This code is the "Blink" code, confirming that the HW is OK.

Testing the Camera

To test the camera, let's run *helloword_1.py*. For that, select the script on `File > Examples > HelloWorld > helloword.py`,

When clicking the green play button, the MicroPython script (*hellowolrd.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 27fps.



Here is the `helloworld.py` script:

```
import sensor, 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.

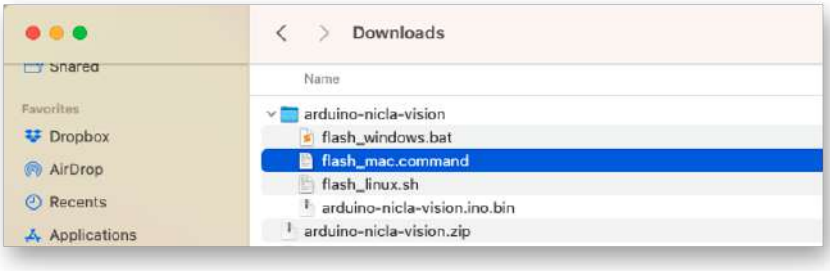
In the GitHub, You can find other Python scripts. Try to test the onboard sensors.

Connecting the Nicla Vision to Edge Impulse Studio

We will need the Edge Impulse Studio later in other labs. Edge Impulse is a leading development platform for machine learning on edge devices.

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

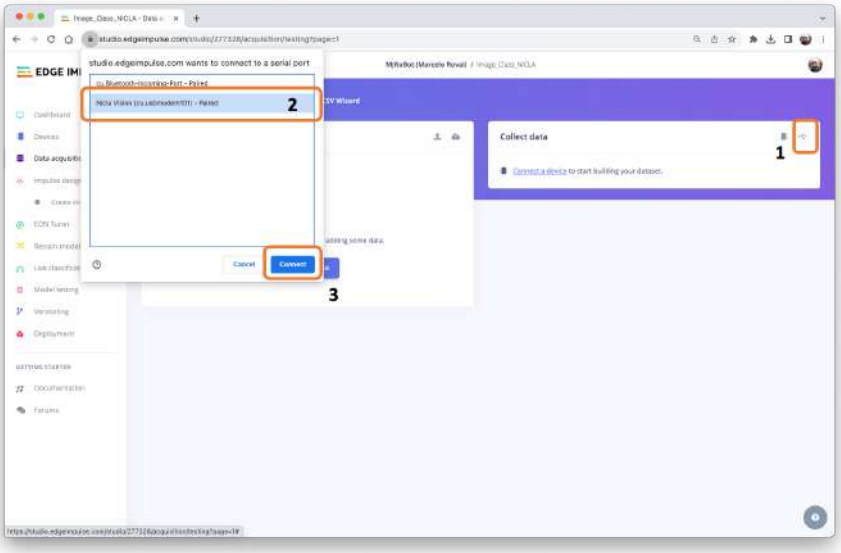
- Download the Arduino CLI for your specific computer architecture (OS)
- Download the most updated EI Firmware.
- Unzip both files and place all the files in the same folder.
- Put the Nicla-Vision on Boot Mode, pressing the reset button twice.
- Run the uploader (EI FW) corresponding to your OS:



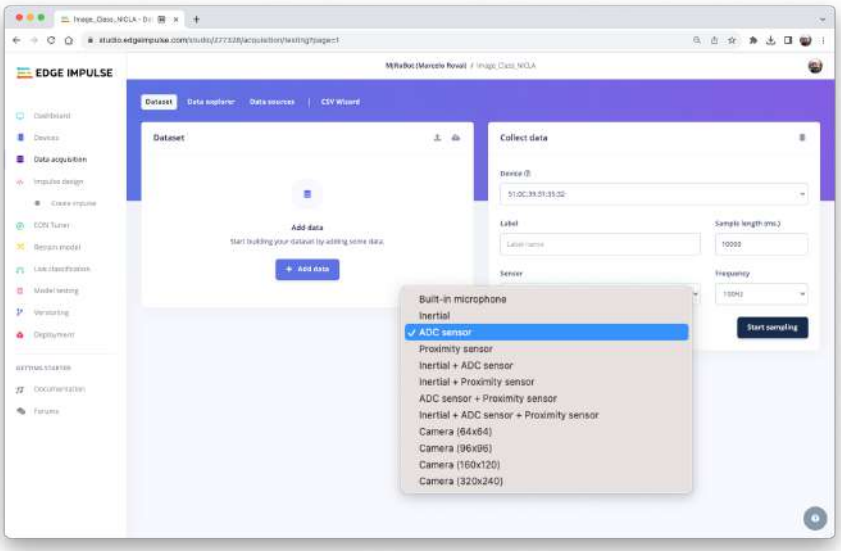
- Executing the specific batch code for your OS will upload the binary *arduino-nicla-vision.bin* to your board.

Using `Chrome`, WebUSB can be used to connect the Nicla to the EI Studio. **The EI CLI is not needed.**

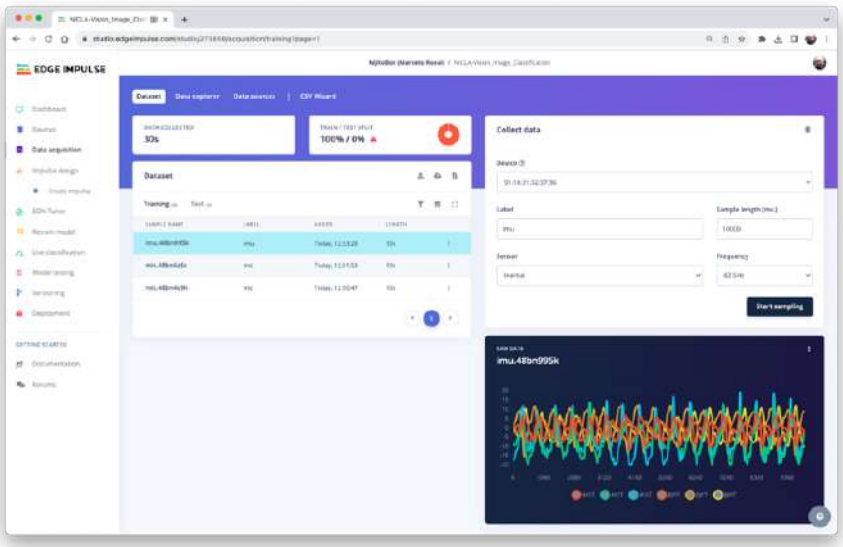
Go to your project on the Studio, and on the `Data Acquisition tab`, select `WebUSB` (1). A window will pop up; choose the option that shows that the `Nicla is paired` (2) and press `[Connect]` (3).



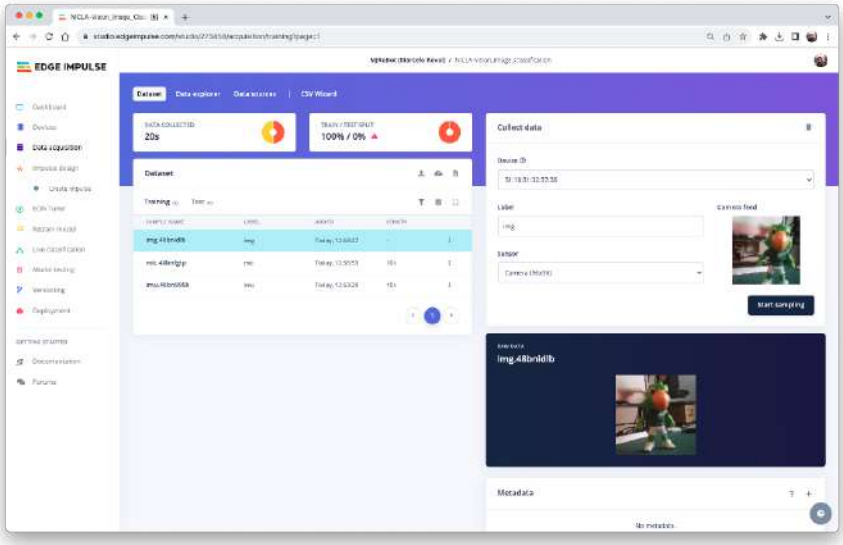
You can choose which sensor data to pick in the Collect Data section on the Data Acquisition tab.



For example. `IMU data (inercial)`:



Or Image (`Camera`):



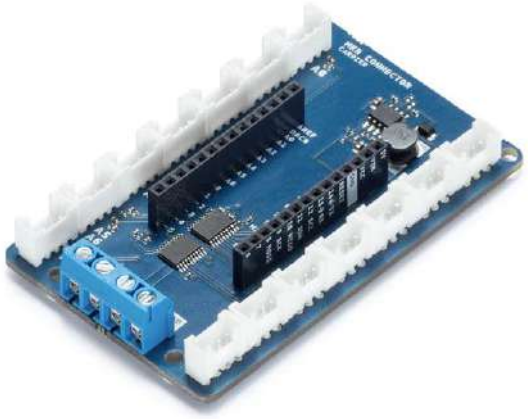
You can also test an external sensor connected to the `ADC` (Nicla pin 0) and the other onboard sensors, such as the `built-in microphone`, the `ToF (Proximity)` or a combination of sensors (`fusion`).

Expanding the Nicla Vision Board (optional)

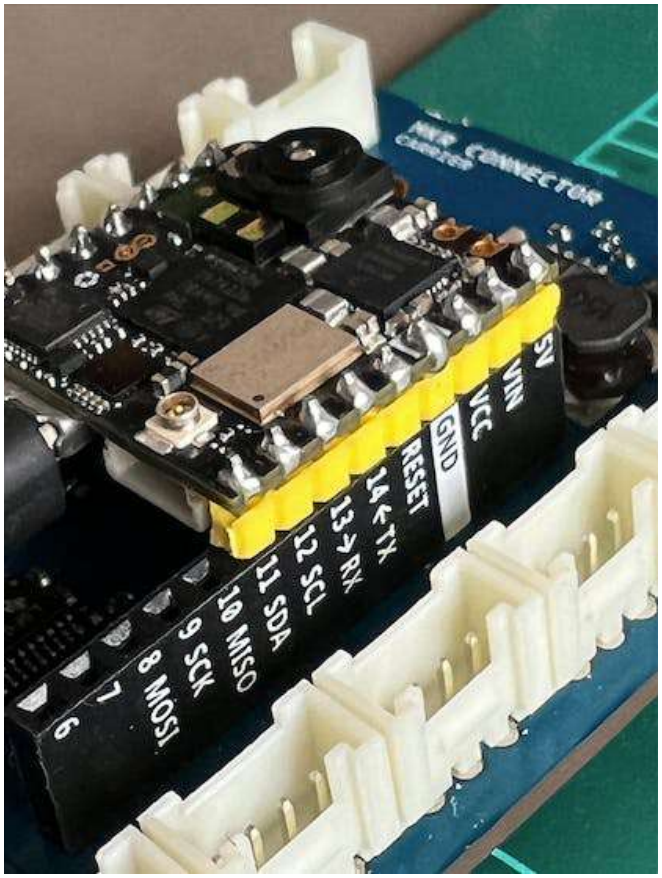
A last item to explore is that sometimes, during prototyping, it is essential to experiment with external sensors and devices. An excellent expansion to the Nicla is the Arduino MKR Connector Carrier (Grove compatible).

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.

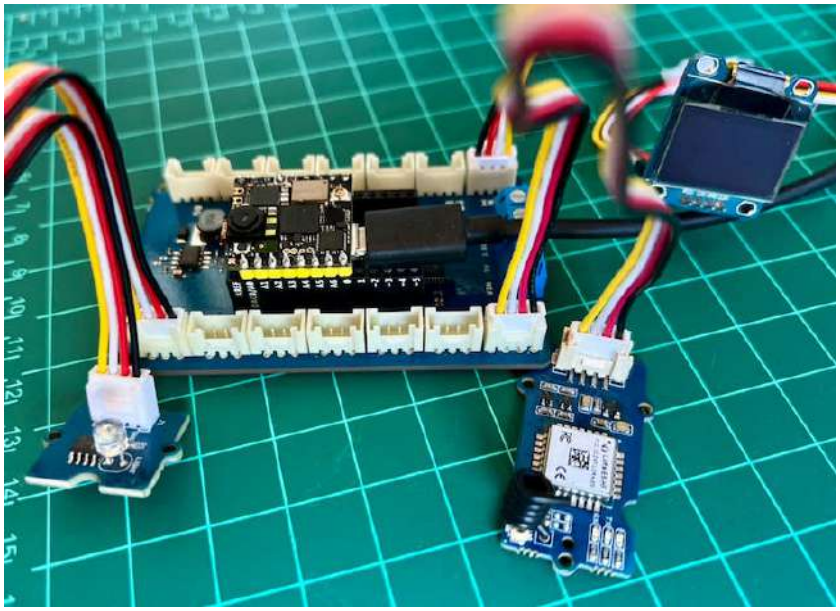
> Note that all 17 Nicla Vision pins will be connected to the Shield Groves, but some Grove connections remain disconnected.



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:

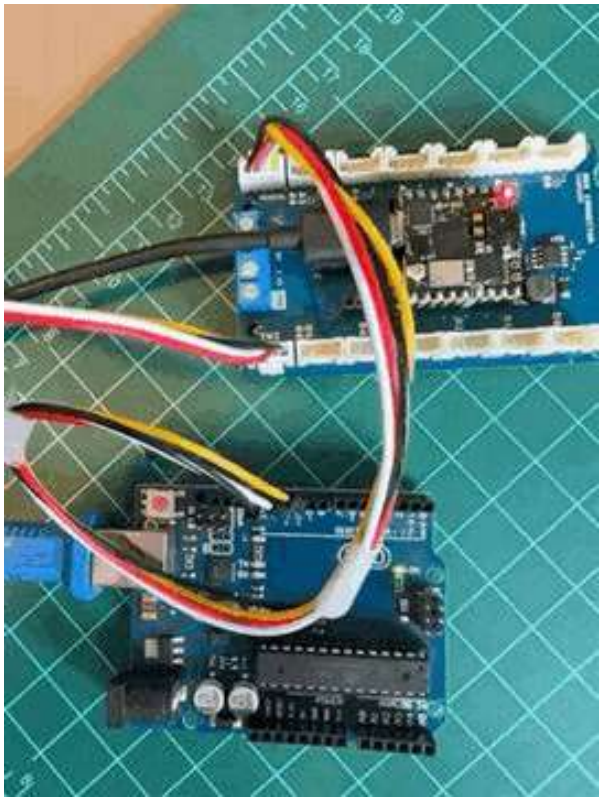


The Grove Light Sensor would be connected to one of the single Analog pins (A0/PC4), the LoRaWAN device to the UART, and the OLED to the I2C connector.

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 created by Adafruit
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.

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 code used or commented on in this hands-on lab.

Resources

- Micropython codes
- Arduino Codes

Image Classification

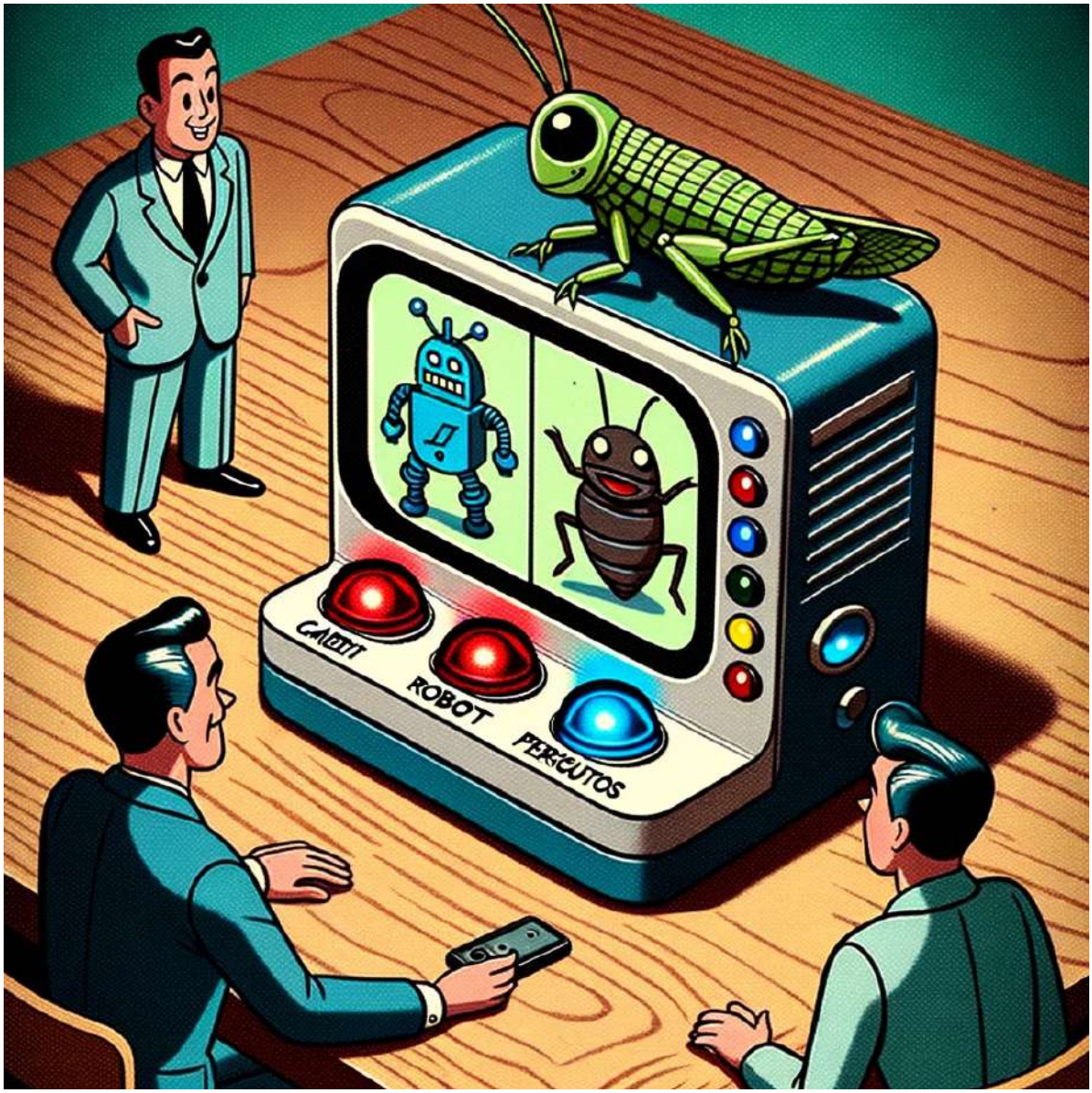
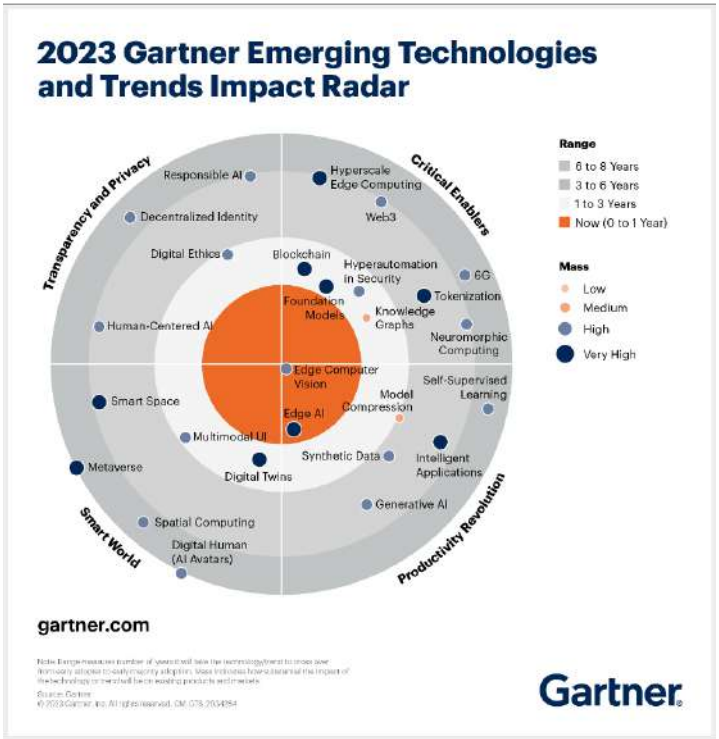


Figure 2: *DALL · E 3 Prompt: Cartoon in a 1950s style featuring a compact electronic device with a camera module placed on a wooden table. The screen displays blue robots on one side and green periquitos on the other. LED lights on the device indicate classifications, while characters in retro clothing observe with interest.*

Overview

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"!

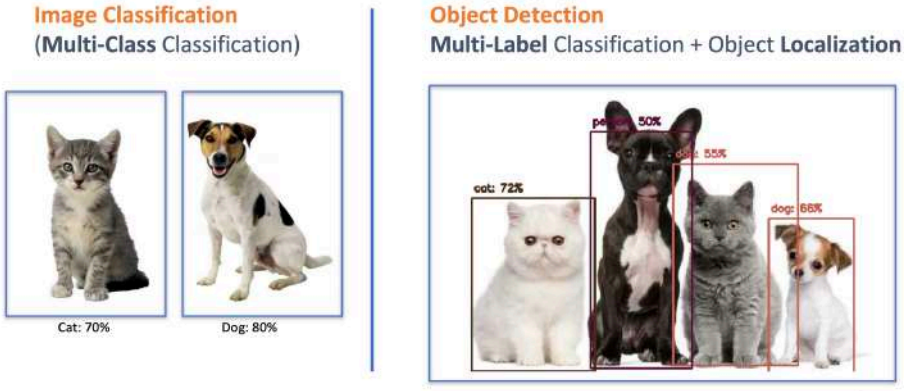
This lab will explore a computer vision project utilizing Convolutional Neural Networks (CNNs) for real-time image classification. Leveraging TensorFlow's robust ecosystem, we'll implement a pre-trained MobileNet model and adapt it for edge deployment. The focus will be optimizing the model to run efficiently 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.

Computer Vision

At its core, computer vision enables 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.

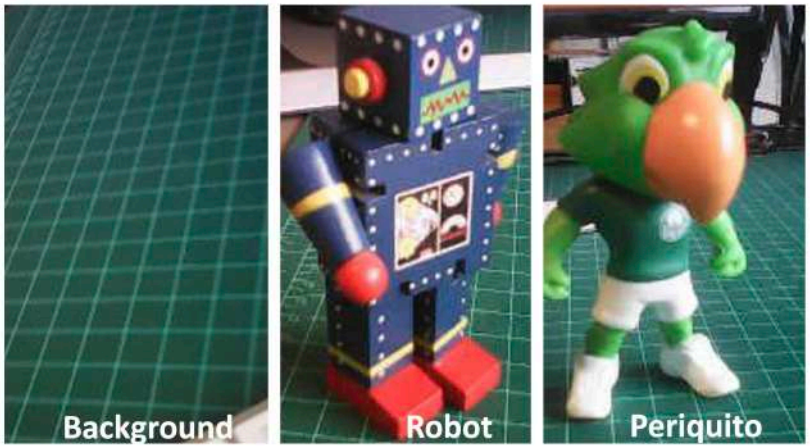
When discussing Computer Vision projects applied to embedded devices, the most common applications that come to mind are *Image Classification* and *Object Detection*.



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.

Image Classification Project Goal

The first step in any ML project is to define the goal. In this case, the goal 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). We will also collect images of a *background* where those two objects are absent.



Data Collection

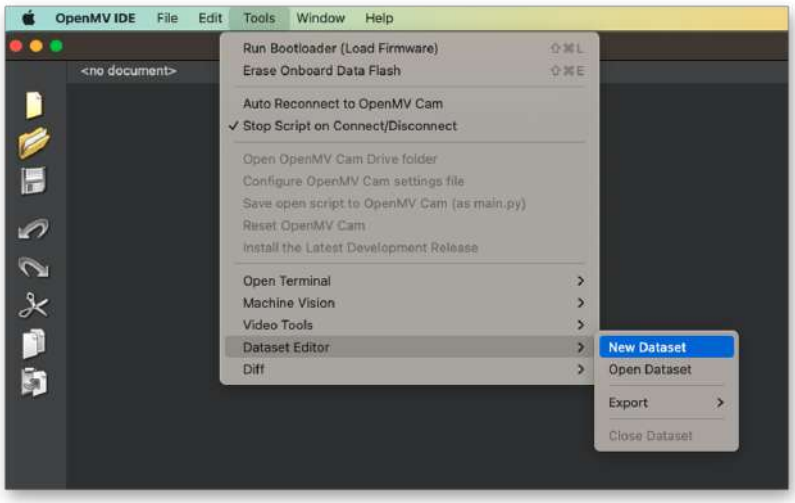
Once we have defined our Machine Learning project goal, the next and most crucial step is collecting the dataset. For image capturing, we can use:

- Web Serial Camera tool,
- Edge Impulse Studio,
- OpenMV IDE,
- A smartphone.

Here, we will use the **OpenMV IDE**.

Collecting Dataset with OpenMV IDE

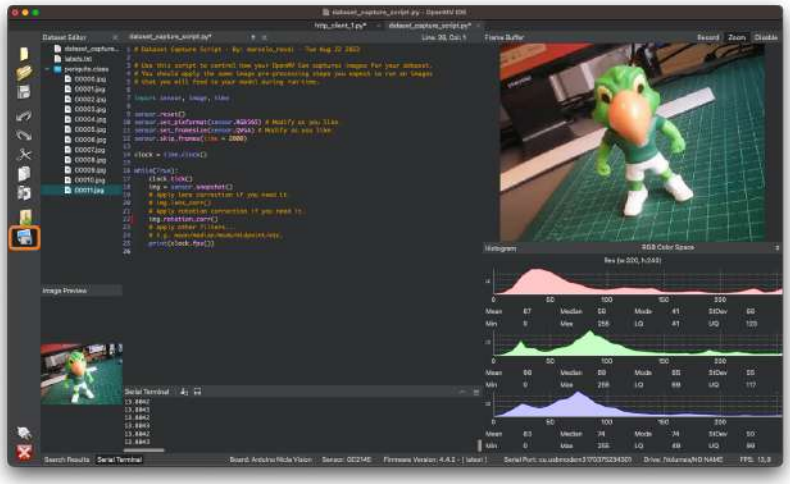
First, we should create a folder on our computer where the data will be saved, for example, "data." Next, on the OpenMV IDE, we go to `Tools > Dataset Editor` and select `New Dataset` to start the dataset collection:



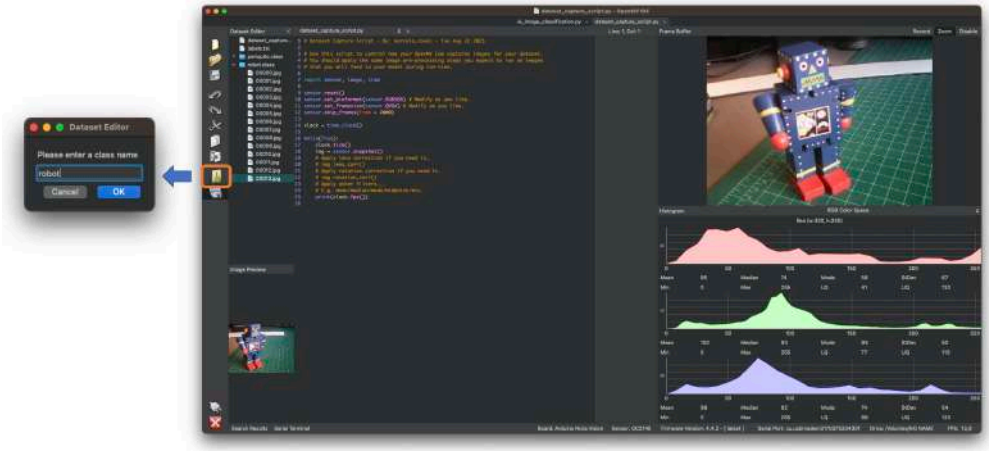
The IDE will ask us to open the file where the data will be saved. Choose the "data" folder that was created. Note that new icons will appear on the Left panel.

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.

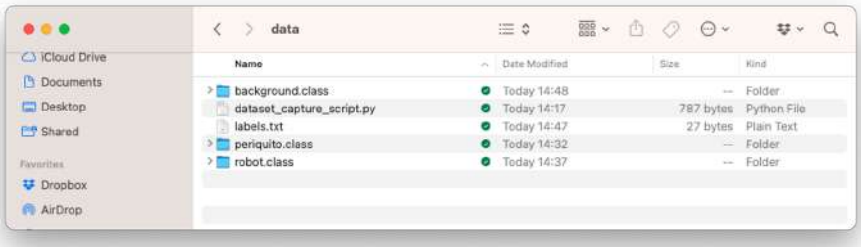


> We suggest around 50 to 60 images from each category. Try to capture different angles, backgrounds, and light conditions.

The stored images use a QVGA frame size of 320×240 and the RGB565 (color pixel format).

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

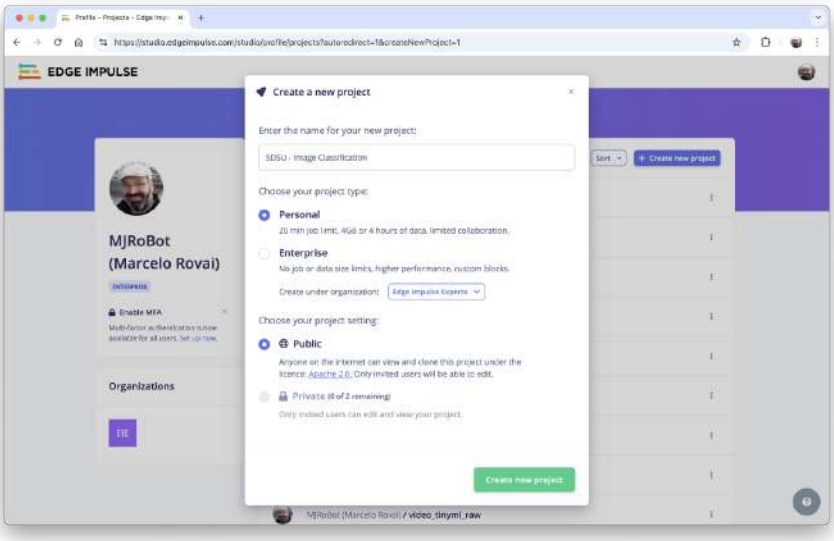
We will end up with a dataset on our computer that contains three classes: *periquito*, *robot*, and *background*.



We should return to *Edge Impulse Studio* and upload the dataset to our created project.

Training the model with Edge Impulse Studio

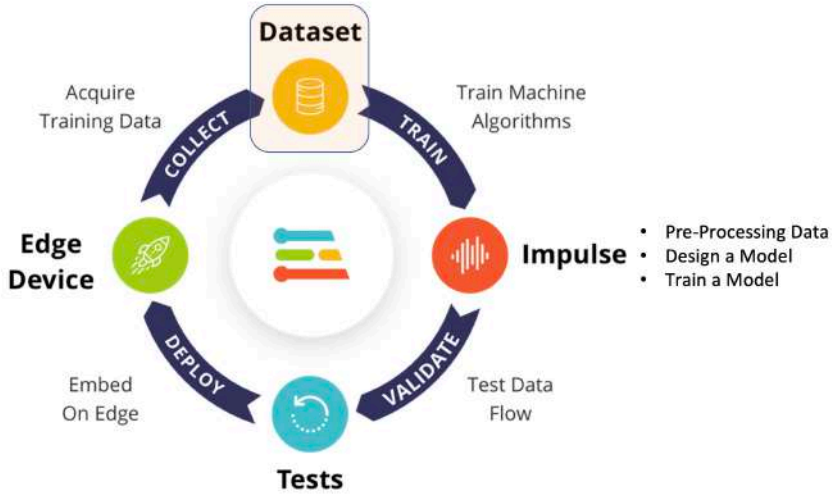
We will use the Edge Impulse Studio to train our model. Enter the account credentials and create a new project:



Here, you can clone a similar project: NICLA-Vision_Image_Classification.

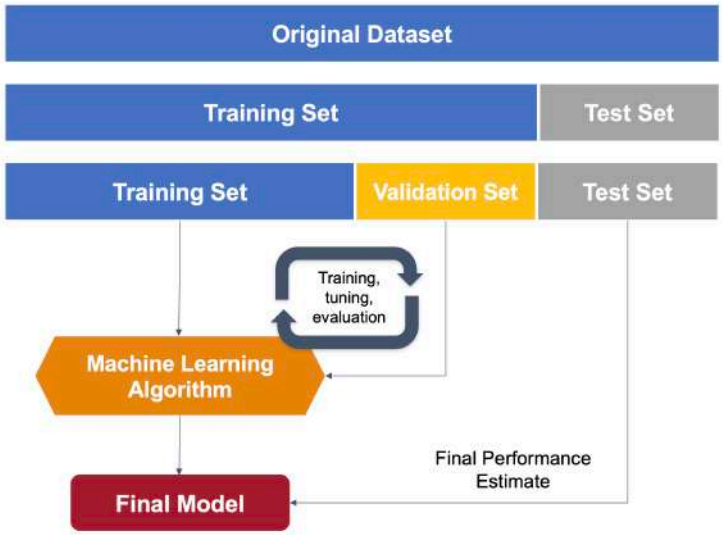
Dataset

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).

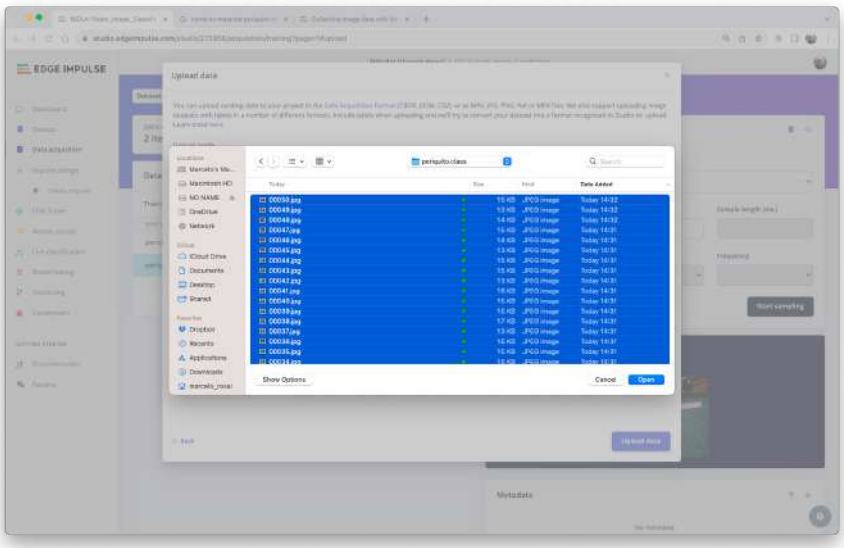


Regarding the Dataset, it is essential to point out that our Original Dataset, captured with the OpenMV IDE, will be split into *Training*, *Validation*, and *Test*. The Test Set will be spared from the beginning and reserved for use only in the Test phase after training. The Validation Set will be used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

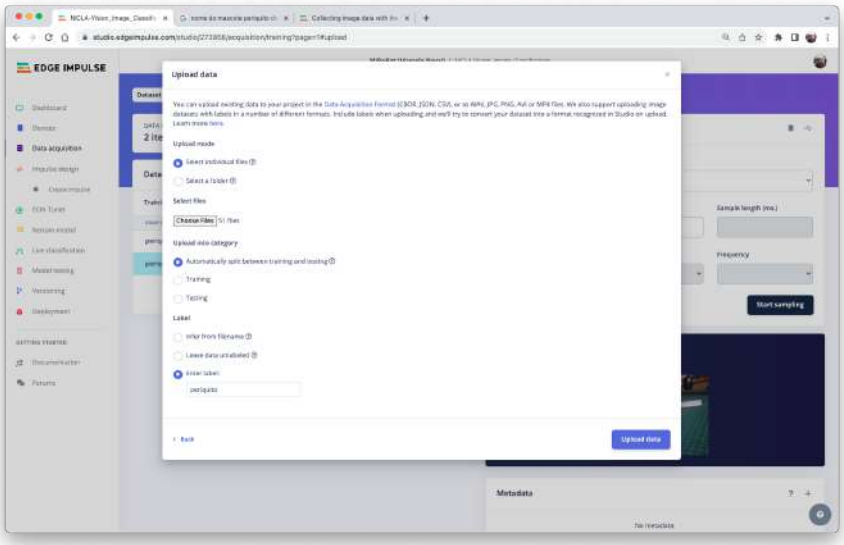
The EI Studio will take a percentage of training data to be used for validation



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



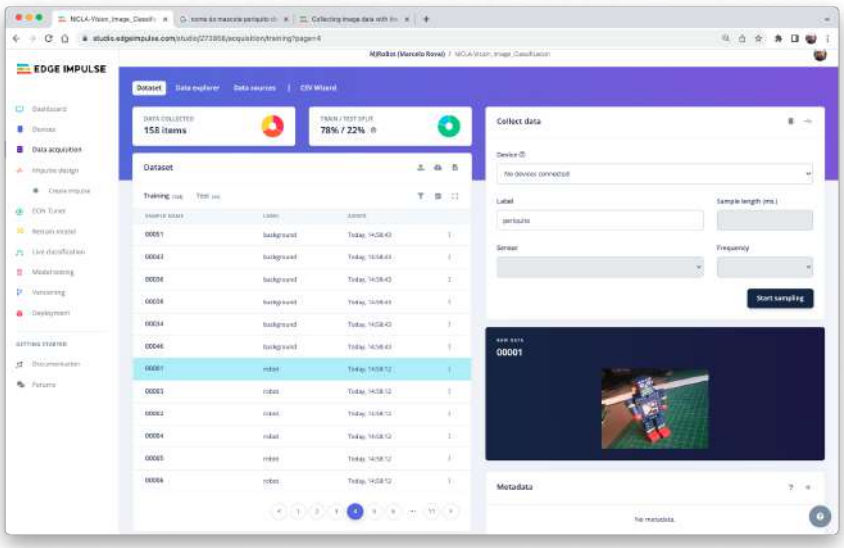
Leave to the Studio the splitting of the original dataset into *train and test* and choose the label about that specific data:



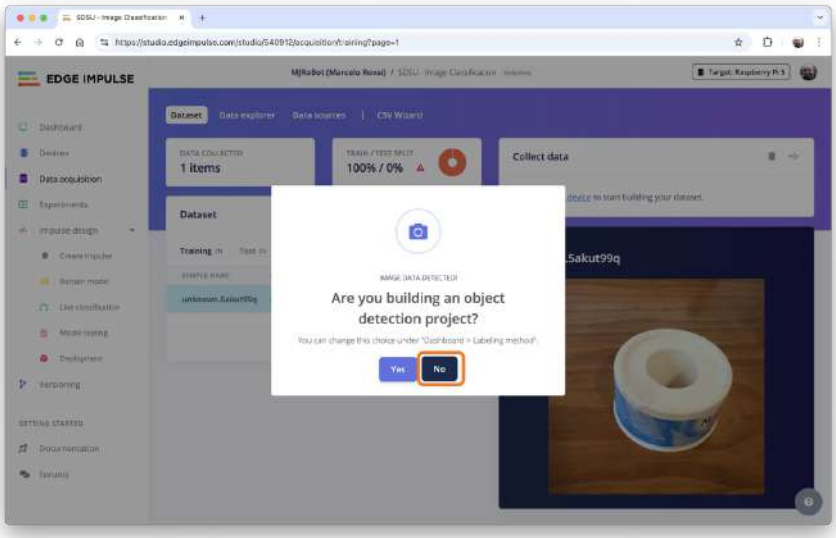
Repeat the procedure for all three classes.

Selecting a folder and upload all the files at once is possible.

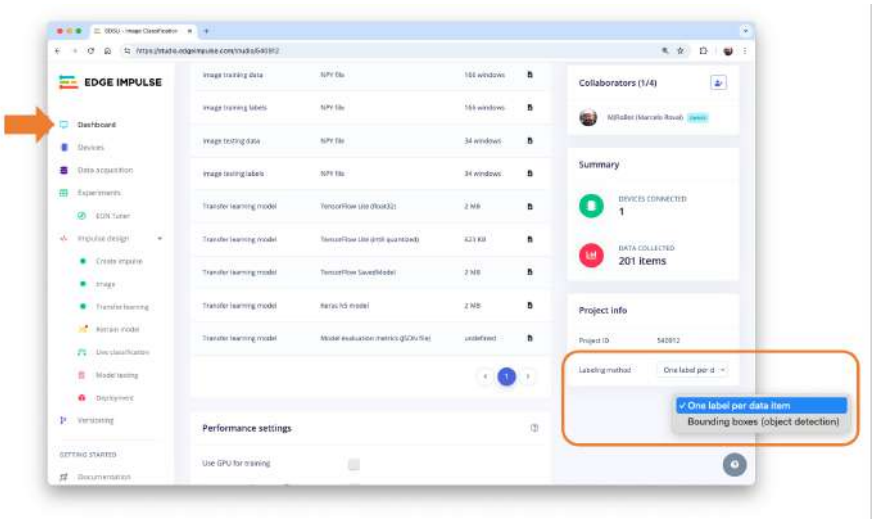
At the end, you should see your "raw data" in the Studio:



Note that when you start to upload the data, a pop-up window can appear, asking if you are building an Object Detection project. Select `[NO]`.



We can always change it in the Dashboard section: `One label per data item` (Image Classification):

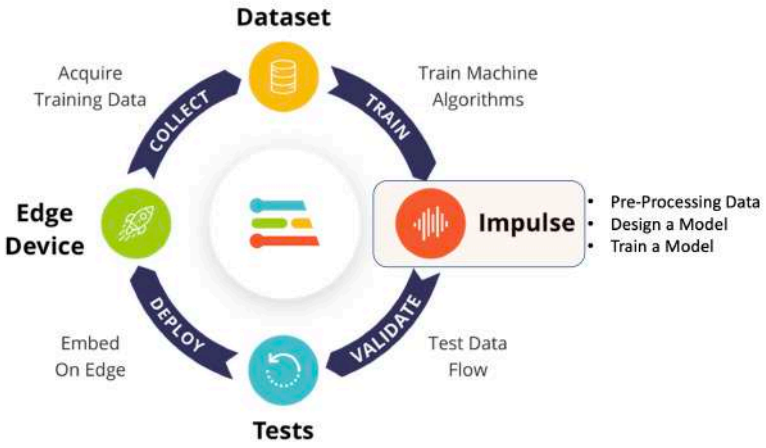


Optionally, the Studio allows us to explore the data, showing a complete view of all the data in the project. We can clear, inspect, or change labels by clicking on individual data items. In our case, the data seems OK.

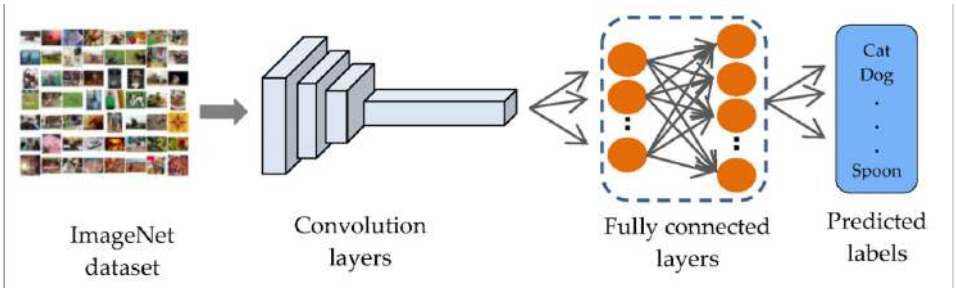
The Impulse Design

In this phase, we should define how to:

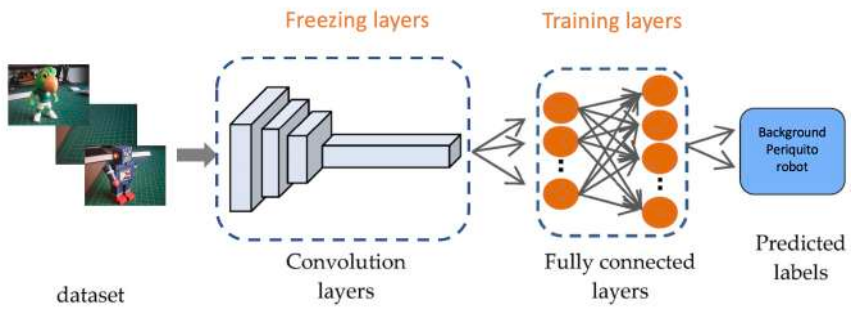
- Pre-process our data, which consists of resizing the individual images and determining the `color depth` to use (be it RGB or Grayscale) and
- Specify a Model, in this case, it will be the `Transfer Learning (Images)` to fine-tune a pre-trained MobileNet V2 image classification model on our data. This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

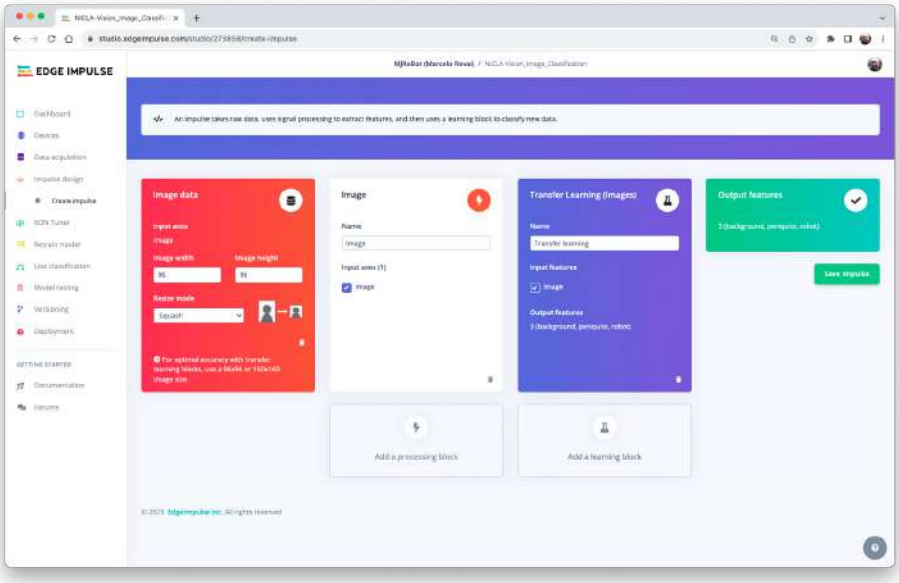
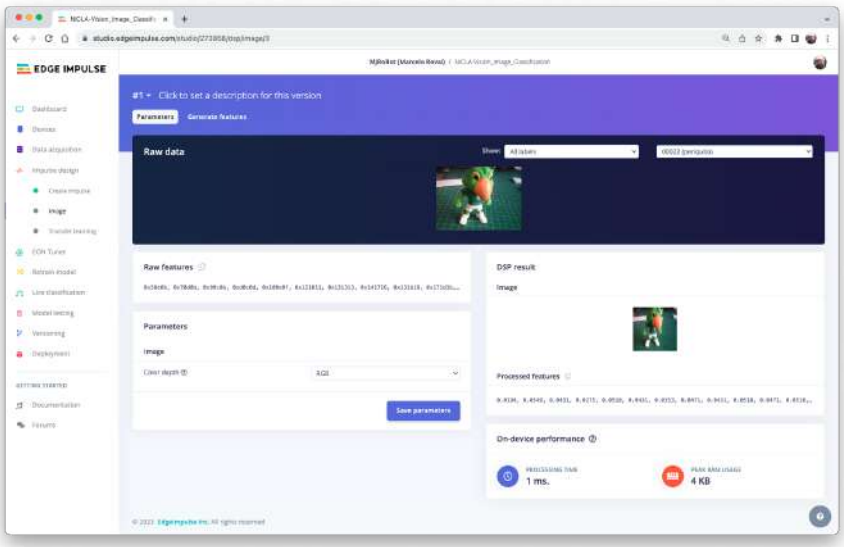
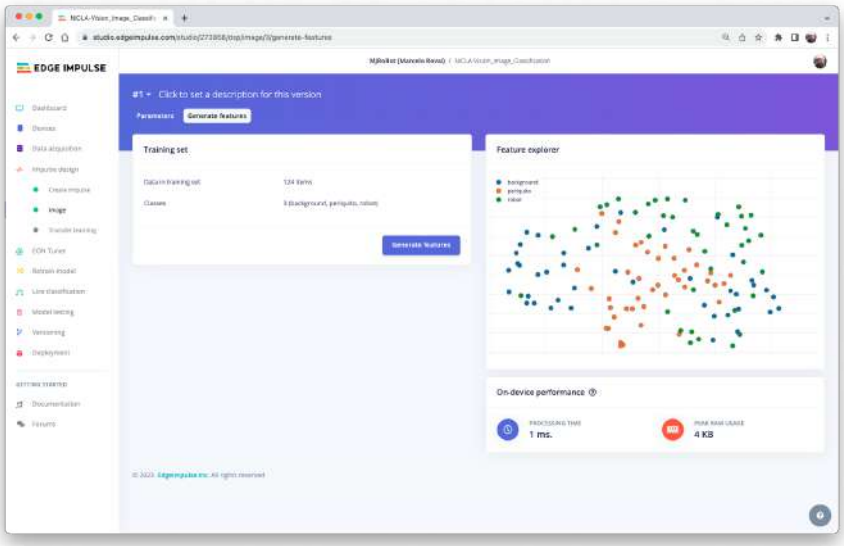


Image Pre-Processing

All the input QVGA/RGB565 images will be converted to 27,640 features ($96 \times 96 \times 3$).



Press `[Save parameters]` and Generate all features:



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.

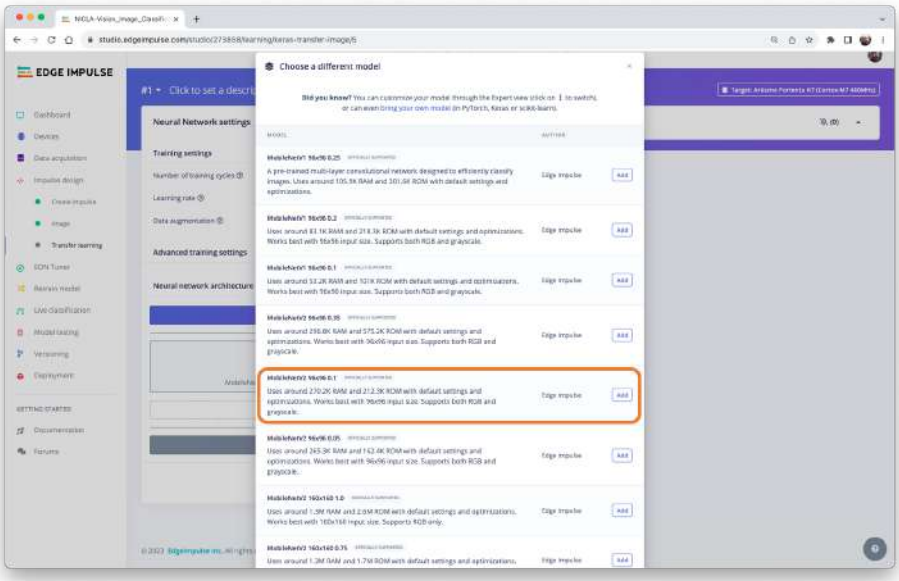
MobileNets are small, low-latency, low-power models parameterized to meet the resource constraints of various use cases. in 2018, Google launched MobileNetV2: Inverted Residuals and Linear Bottlenecks.

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 improve 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 (96×96 images) and V2 (96×96 or 160×160 images), with several different α values (from 0.05 to 1.0). For example, you will get the highest accuracy with V2, 160×160 images, and $\alpha = 1.0$. Of course, there is a trade-off. The higher the accuracy, the more memory (around 1.3 MB RAM and 2.6 MB 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 53.2 K RAM and 101 K ROM).

We will use **MobileNetV2 96x96 0.1** (**or 0.05**) 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:



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).

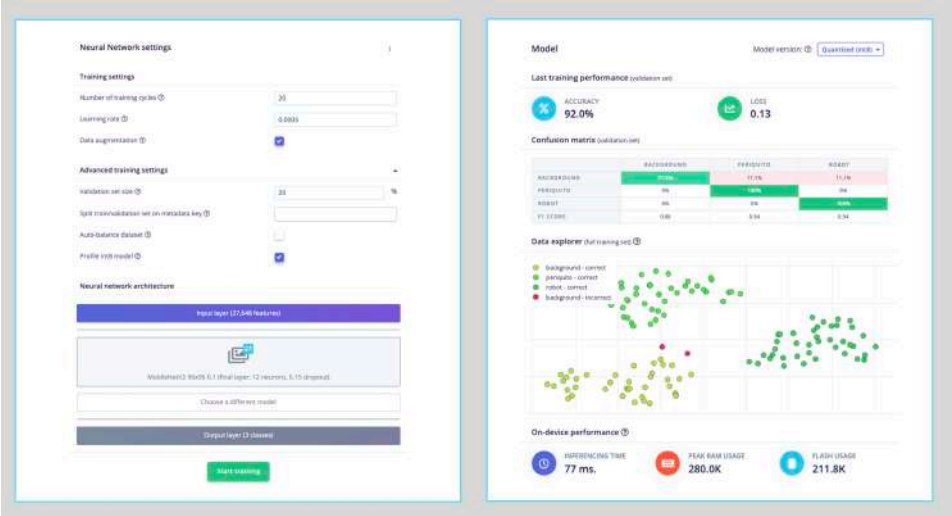
Looking under the hood, here you can see how Edge Impulse implements a data Augmentation policy on your data:

```
# 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
```

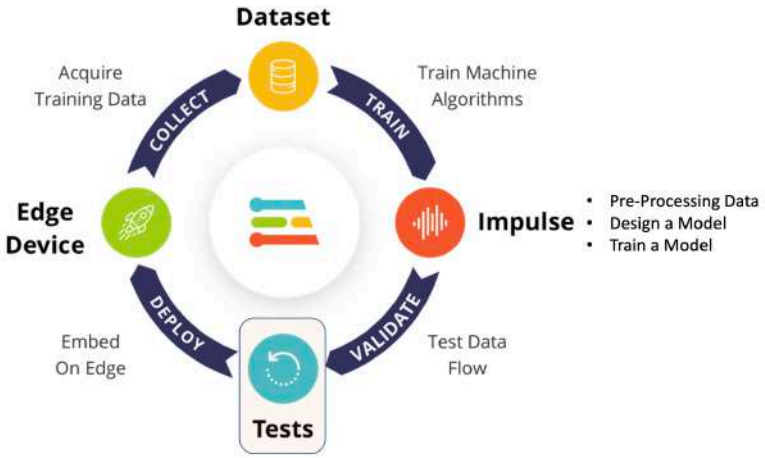
Exposure to these variations during training can help prevent your model from taking shortcuts by "memorizing" superficial clues in your training data, meaning it may better reflect the deep underlying patterns in your dataset.

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

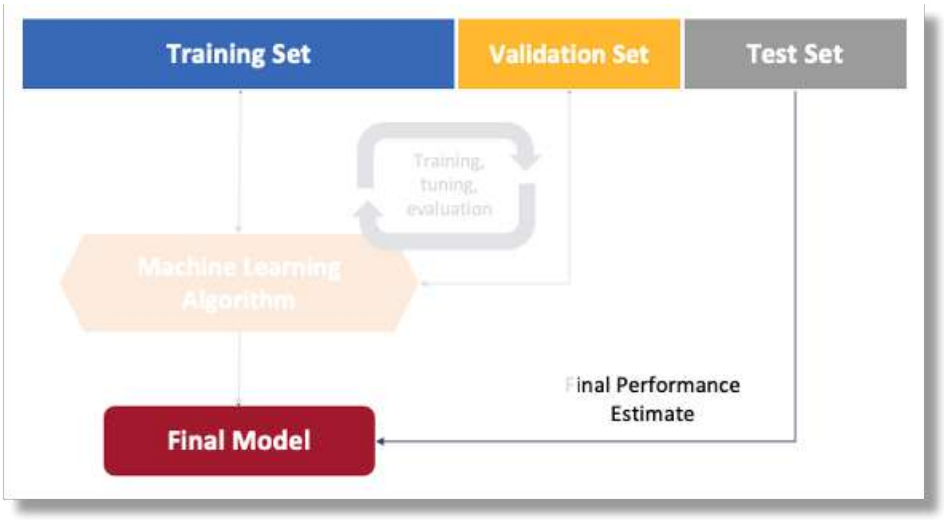


The result is excellent, with 77 ms of latency (estimated), which should result in around 13 fps (frames per second) during inference.

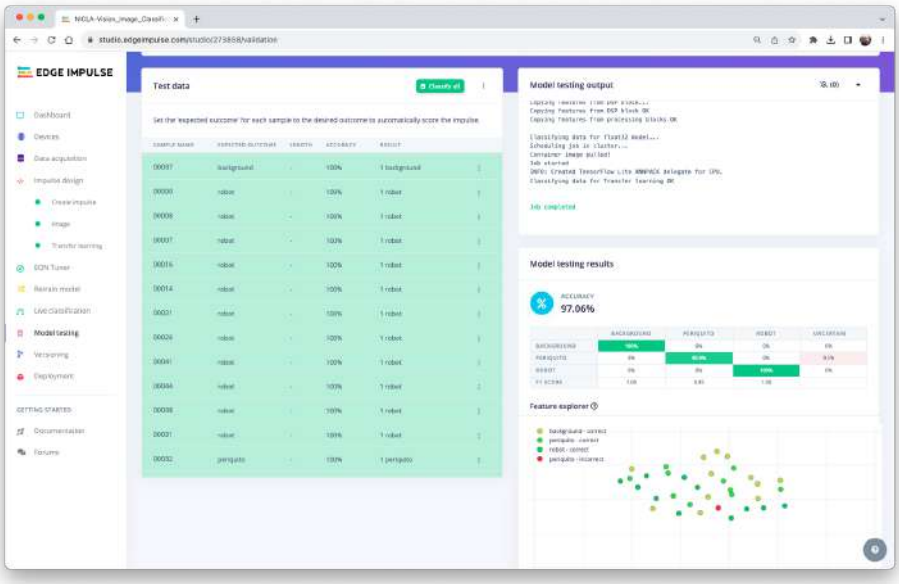
Model Testing



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

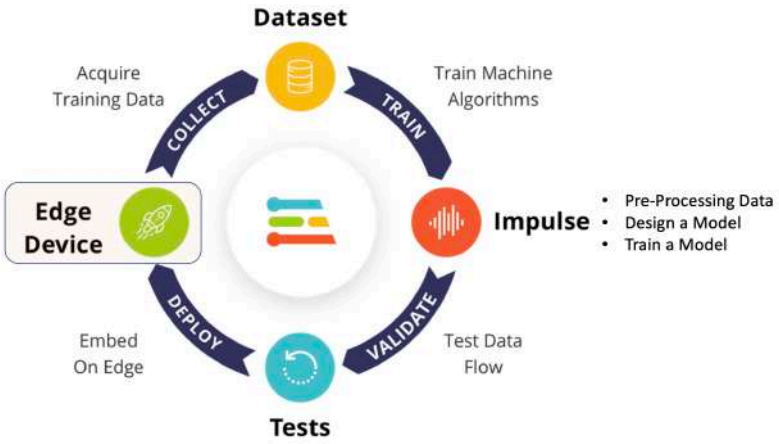


The result is, again, excellent.



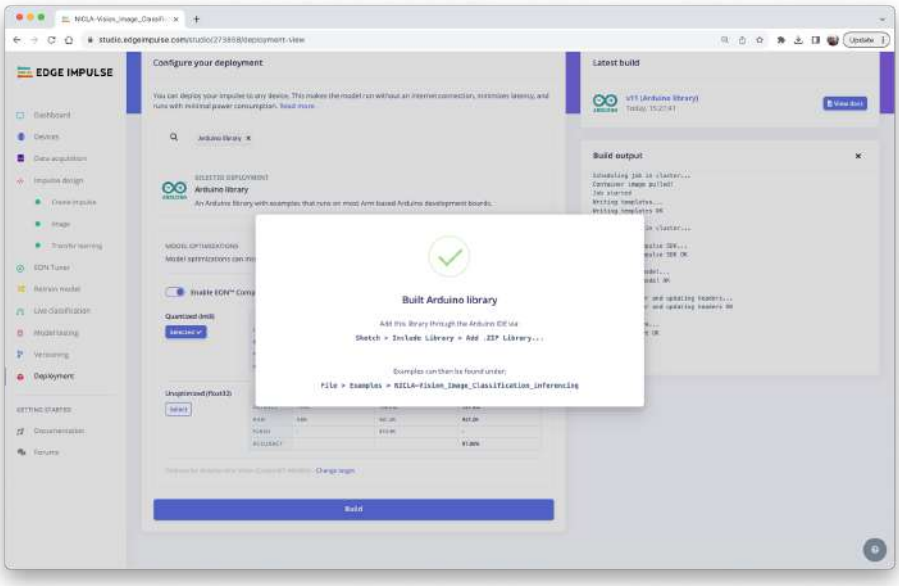
Deploying the model

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



Arduino Library

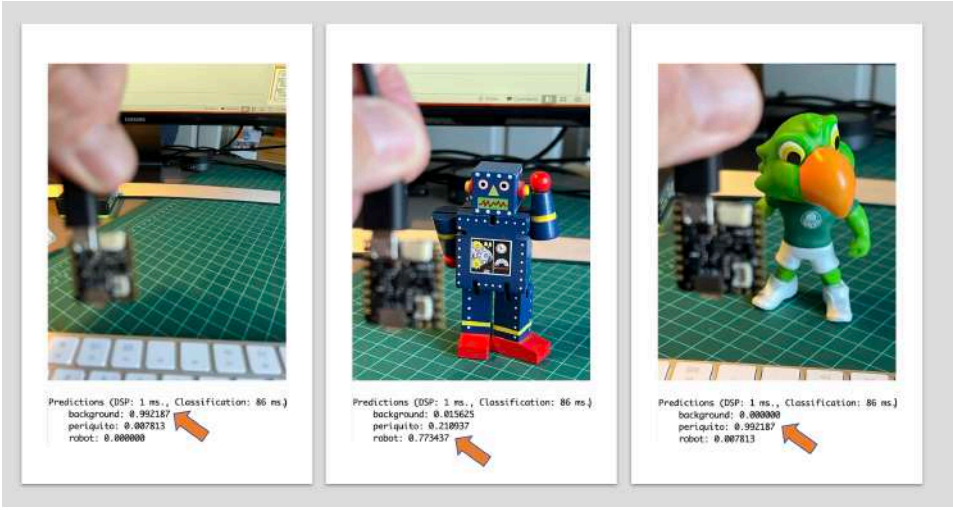
First, Let's deploy it as an Arduino Library:



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

> Note that Arduino Nicla Vision has, by default, 512 KB of RAM allocated for the M7 core and an additional 244 KB 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 86 ms of measured latency.

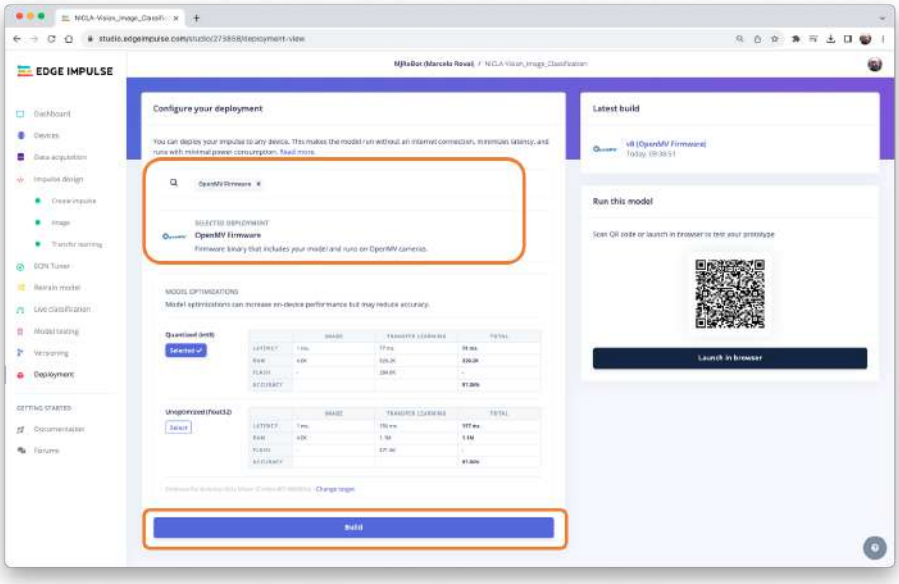


Here is a short video showing the inference results: https://youtu.be/bZPZZJblU-o

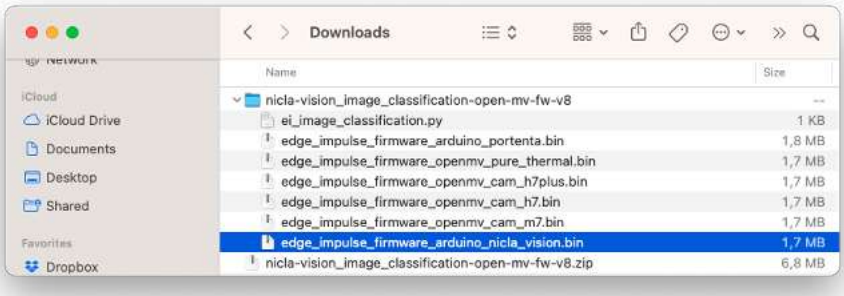
OpenMV

It is possible to deploy the trained model to be used with OpenMV in two ways: as a library and as a firmware (FW). Choosing FW, the Edge Impulse Studio generates optimized models, libraries, and frameworks needed to make the inference. Let's explore this option.

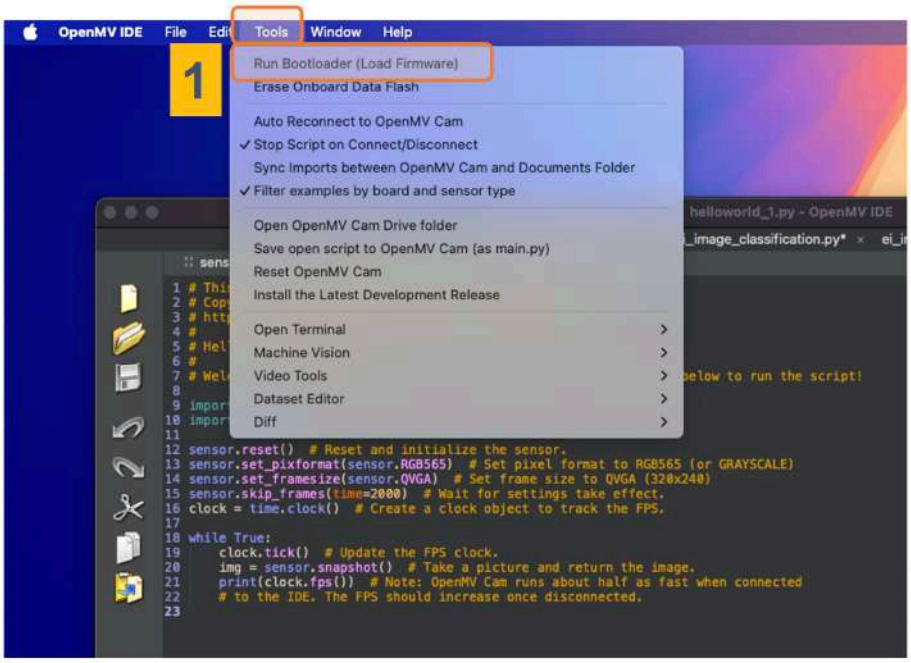
Select `OpenMV Firmware` on the `Deploy Tab` and press `[Build]`.



On the computer, we will find a ZIP file. Open it:



Use the Bootloader tool on the OpenMV IDE to load the FW on your board (1):

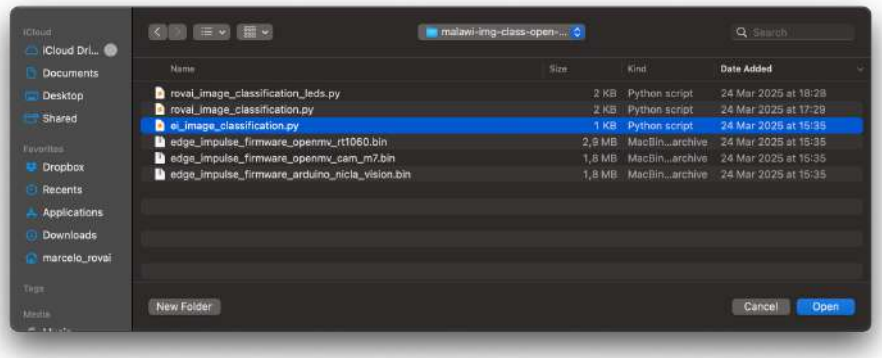


Select the appropriate file (.bin for Nicla-Vision):

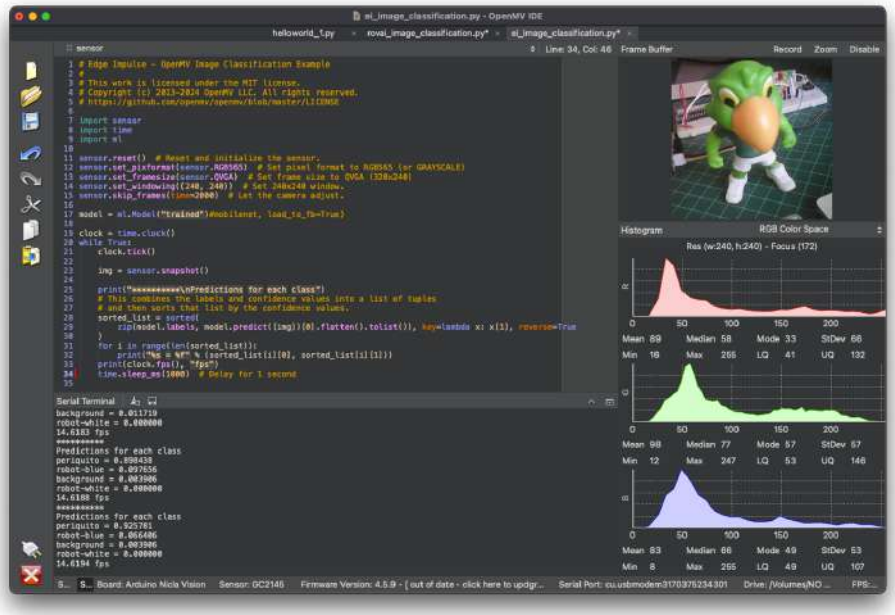
After the download is finished, press OK:

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.



The classification result will appear at the Serial Terminal. If it is dificult to read the result, include a new line in the code to add some delay:

```
import time
While True:
...
    time.sleep_ms(200)  # Delay for .2 second
```

Changing the Code to add labels

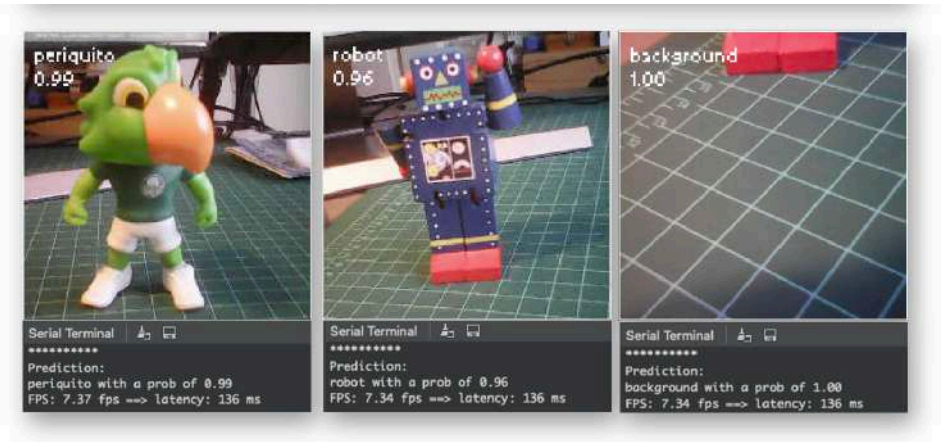
The code provided by Edge Impulse can be modified so that we can see, for test reasons, the inference result directly on the image displayed on the OpenMV IDE.

Upload the code from GitHub, or modify it as below:

```
# Marcelo Rovai - NICLA Vision - Image Classification
# Adapted from Edge Impulse - OpenMV Image Classification Example
# @24March25

import sensor
import time
import ml

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

model = ml.Model("trained")#mobilenet, load_to_fb=True)

clock = time.clock()

while True:
    clock.tick()
    img = sensor.snapshot()

    fps = clock.fps()
    lat = clock.avg()
    print("**********\nPrediction:")
    # Combines labels & confidence into a list of tuples and then
    # sorts that list by the confidence values.
    sorted_list = sorted(
        zip(model.labels, model.predict([img])[0].flatten().tolist()),
      key=lambda x: x[1], reverse=True
    )

    # Print only the class with the highest probability
    max_val = sorted_list[0][1]
    max_lbl = sorted_list[0][0]
```

```
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 the label with the highest probability to the image viewer
img.draw_string(
10, 10,
max_lbl + "\n{:.2f}".format(max_val),
mono_space = False,
scale=3
)

time.sleep_ms(500)  # Delay for .5 second
```

Here you can see the result:

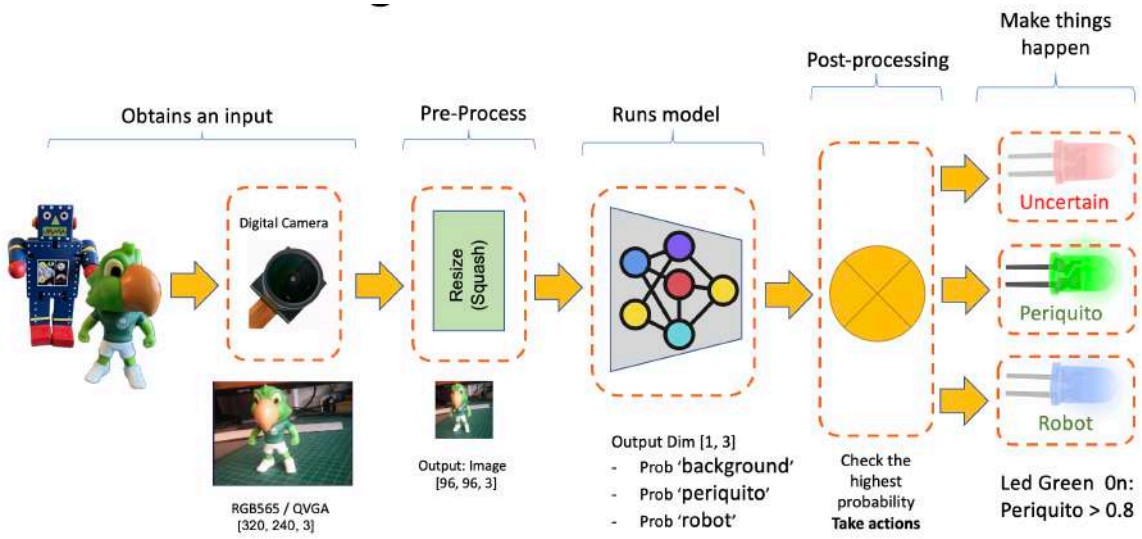


Note that the latency (136 ms) is almost double of what we got directly with the Arduino IDE. This is because we are using the IDE as an interface and also the time to wait for the camera to be ready. If we start the clock just before the inference, the latency should drop to around 70 ms.

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

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, time, ml
from machine import LED

ledRed = LED("LED_RED")
ledGre = LED("LED_GREEN")
ledBlu = LED("LED_BLUE")

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

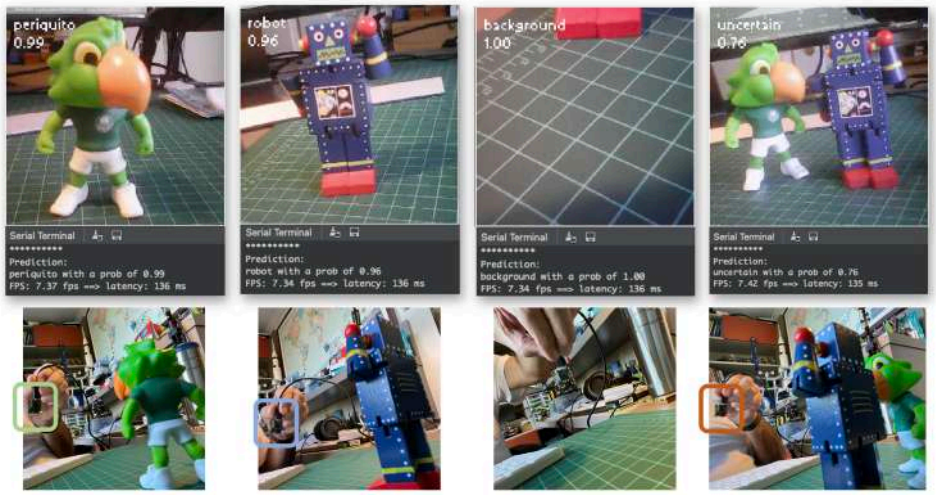
```
model = ml.Model("trained")#mobilenet, load_to_fb=True)

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

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()
    fps = clock.fps()
    lat = clock.avg()
    print("**********\nPrediction:")
    sorted_list = sorted(
        zip(model.labels, model.predict([img])[0].flatten().tolist()),
      key=lambda x: x[1], reverse=True
    )
```

```
    # Print only the class with the highest probability
    max_val = sorted_list[0][1]
    max_lbl = sorted_list[0][0]

    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 the label with the highest probability to the image viewer
    img.draw_string(
    10, 10,
    max_lbl + "\n{:.2f}".format(max_val),
    mono_space = False,
    scale=3
    )

    setLEDs(max_lbl)
    time.sleep_ms(200)  # Delay for .2 second
```

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

- Led Red 0n: Uncertain (no class is over 0.8)
- Led Green 0n: Periquito > 0.8
- Led Blue 0n: Robot > 0.8
- All LEDs Off: Background > 0.8

Here is the result:



In more detail

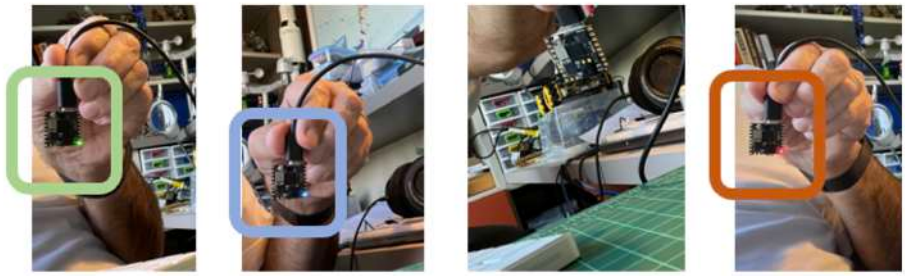
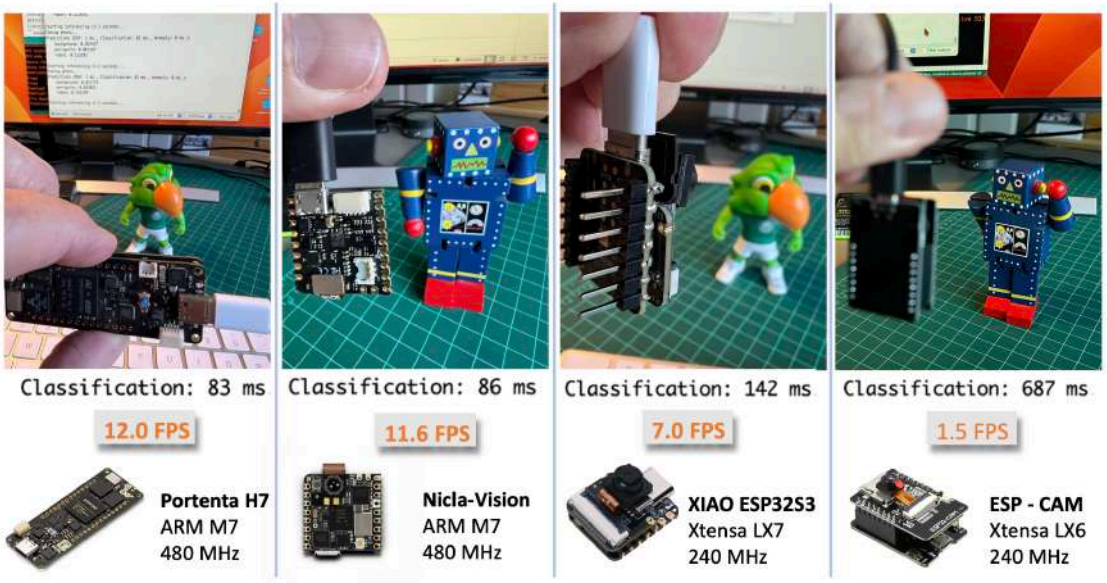


Image Classification (non-official) Benchmark

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:



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.

Resources

- Micropython codes
- Dataset
- Edge Impulse Project

Object Detection

Figure 3: *DALL · E 3 Prompt: Cartoon in the style of the 1940s or 1950s showcasing a spacious industrial warehouse interior. A conveyor belt is prominently featured, carrying a mixture of toy wheels and boxes. The wheels are distinguishable with their bright yellow centers and black tires. The boxes are white cubes painted with alternating black and white patterns. At the end of the moving conveyor stands a retro-styled robot, equipped with tools and sensors, diligently classifying and counting the arriving wheels and boxes. The overall aesthetic is reminiscent of mid-century animation with bold lines and a classic color palette.*

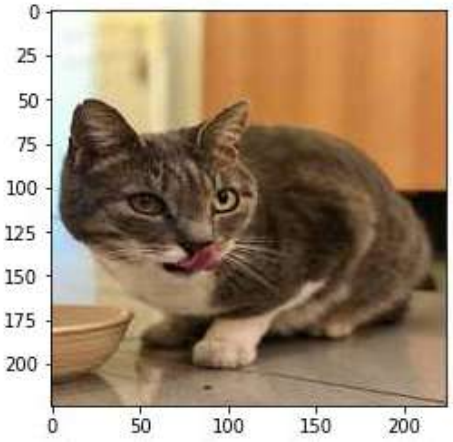
Overview

This continuation of Image Classification on Nicla Vision is now exploring **Object Detection**.

Object Detection versus Image Classification

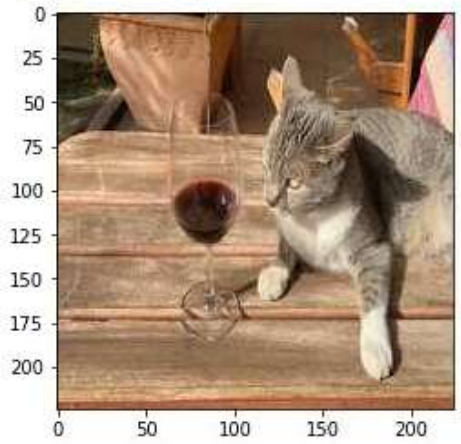
The main task with Image Classification models is to produce a list of the most probable object categories present on an image, for example, to identify a tabby cat just after his dinner:

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



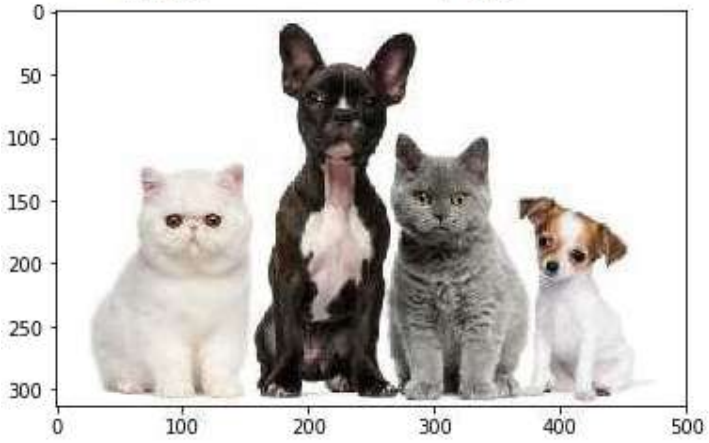
But what happens when the cat jumps near the wine glass? The model still only recognizes the predominant category on the image, the tabby cat:

```
[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%
```

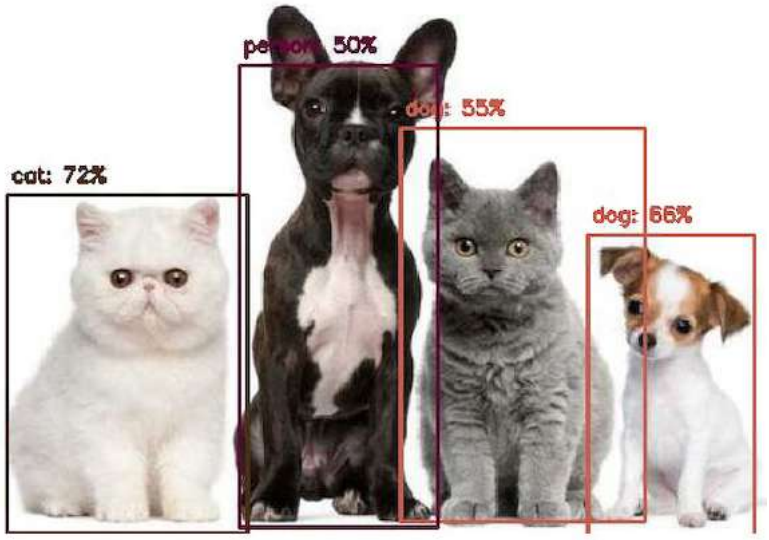


The model identifies the above image utterly wrong as an "ashcan," possibly due to the color tonalities.

> The model used in all previous examples is MobileNet, which was trained with a large dataset, *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 Raspberry Pi but unsuitable for use with embedded devices, where the RAM is usually lower than 1 Mbyte.

An innovative solution for Object Detection: FOMO

Edge Impulse launched in 2022, **FOMO** (Faster Objects, More Objects), a novel solution for performing 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) and the Espressif ESP32 devices (ESP-CAM and XIAO ESP32S3 Sense).

In this Hands-On lab, 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.

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.

Data Collection

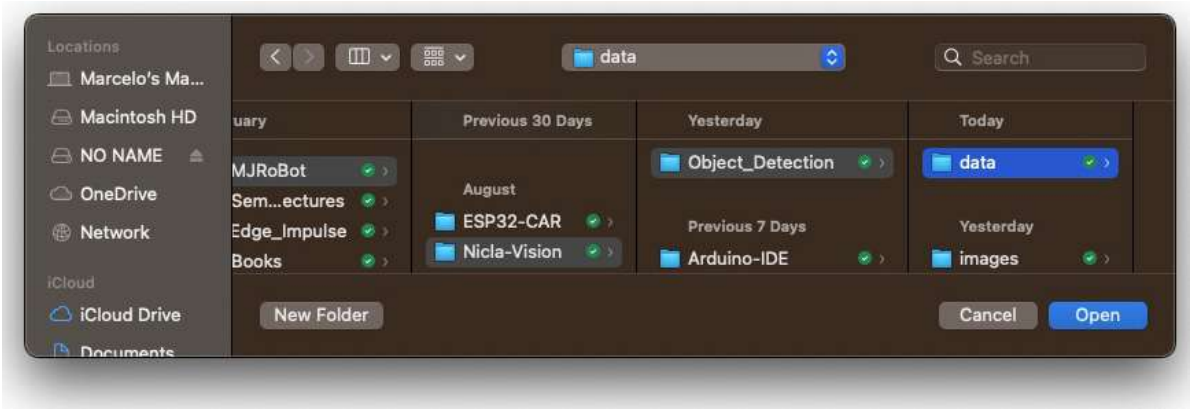
For image capturing, we can use:

- Web Serial Camera tool,
- Edge Impulse Studio,
- OpenMV IDE,
- A smartphone.

Here, we will use the **OpenMV IDE**.

Collecting Dataset with OpenMV IDE

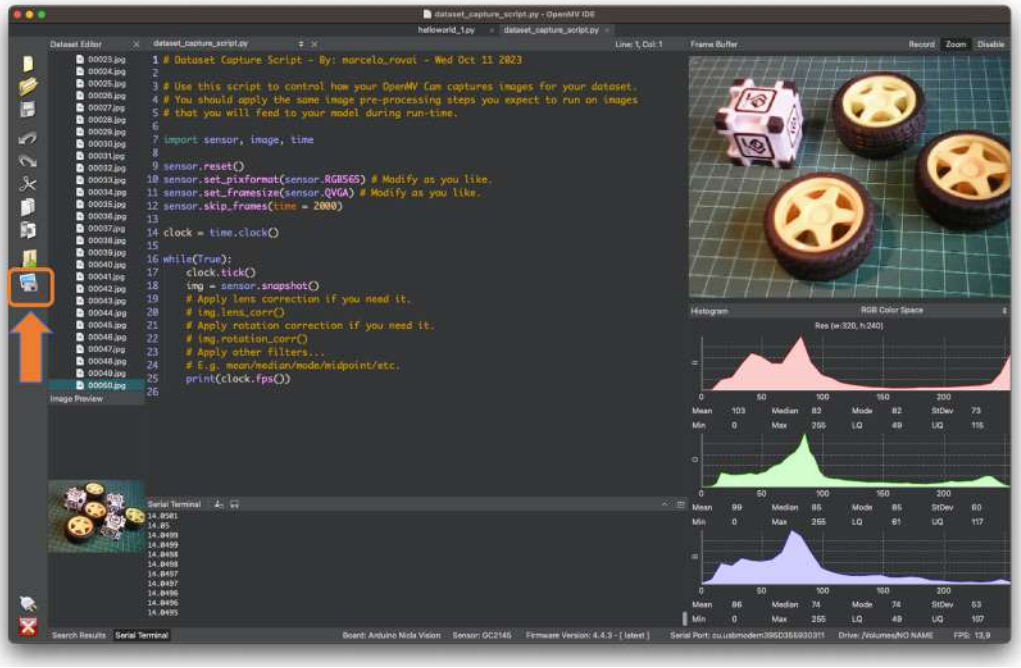
First, we create a folder on the computer where the data will be saved, for example, "data." Next, on the OpenMV IDE, we go to Tools > Dataset Editor and select New Dataset to start the dataset collection:



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

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

> The stored images use a QVGA frame size 320 × 240 and RGB565 (color pixel format).

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

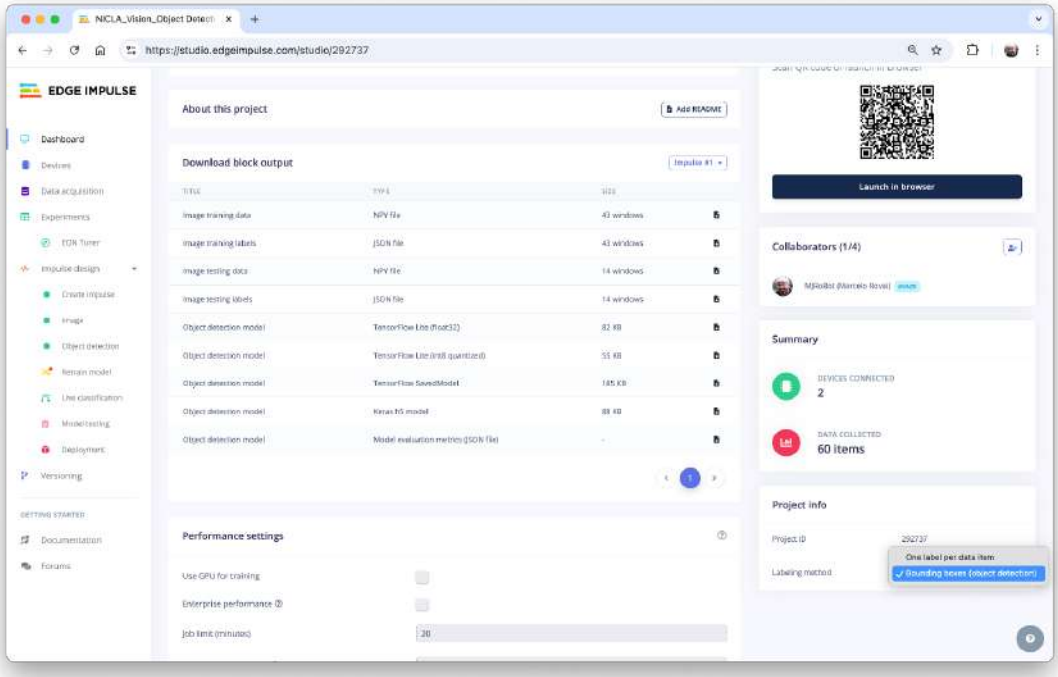
Edge Impulse Studio

Setup the project

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

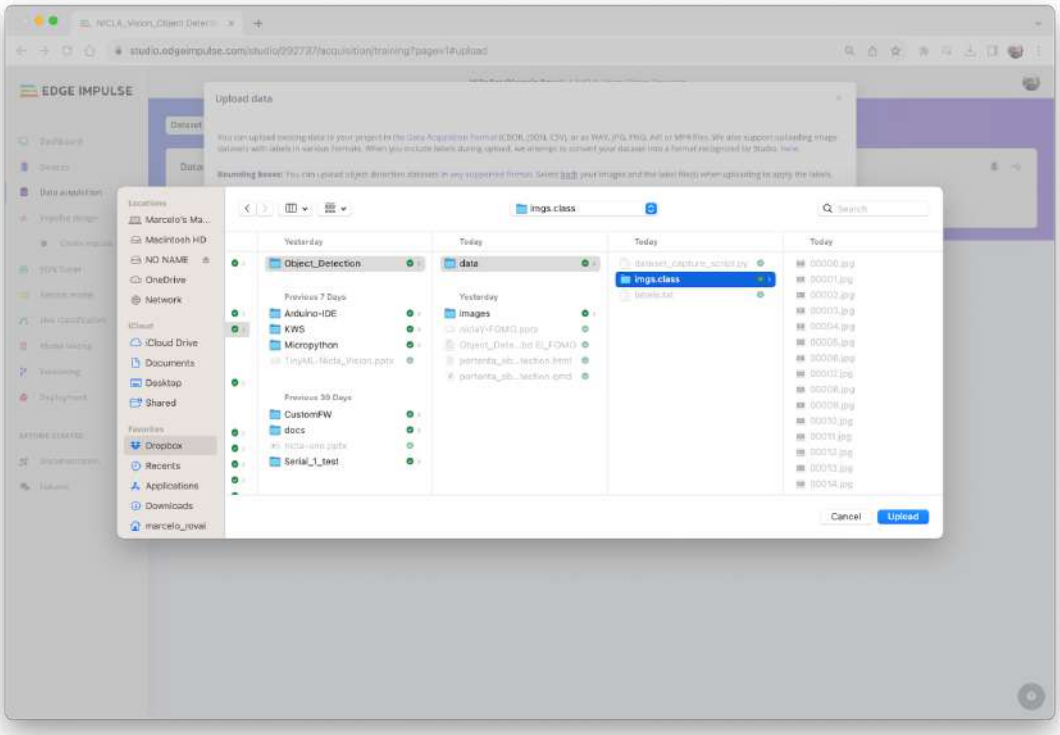
> Here, you can clone the project developed for this hands-on: NICLA_Vision_Object_Detection.

On the Project `Dashboard`, go to **Project info** and select **Bounding boxes (object detection),** and at the right-top of the page, select `Target`, **Arduino Nicla Vision (Cortex-M7)**.

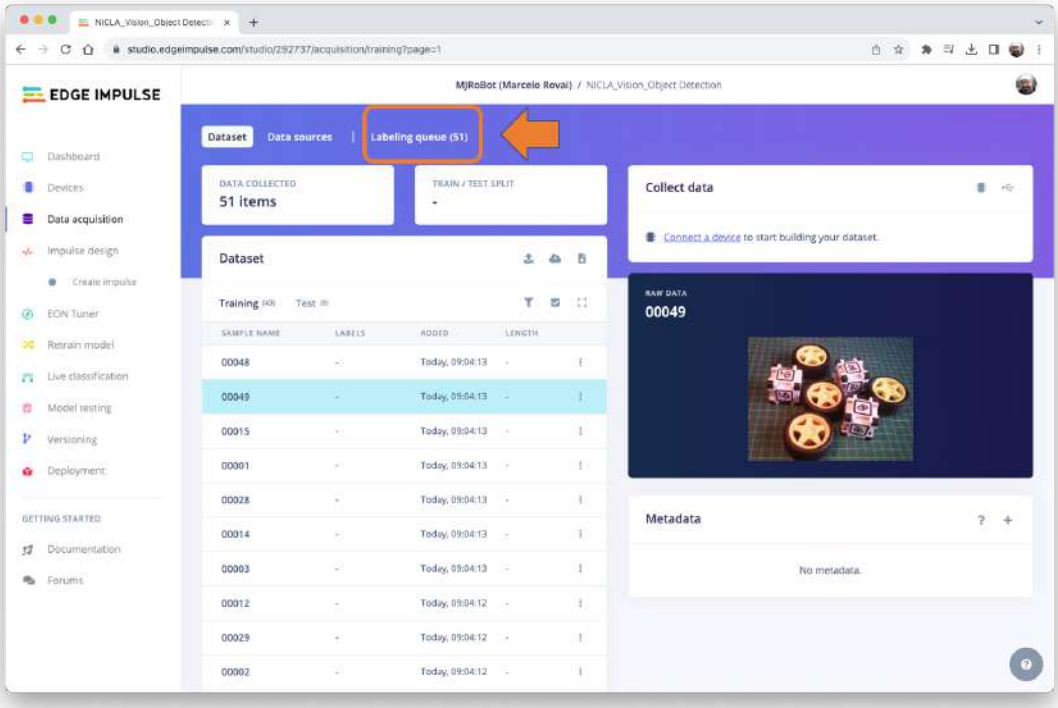


Uploading the unlabeled data

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 unlabeled images (51) were uploaded, but they still need to be labeled appropriately before being used 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

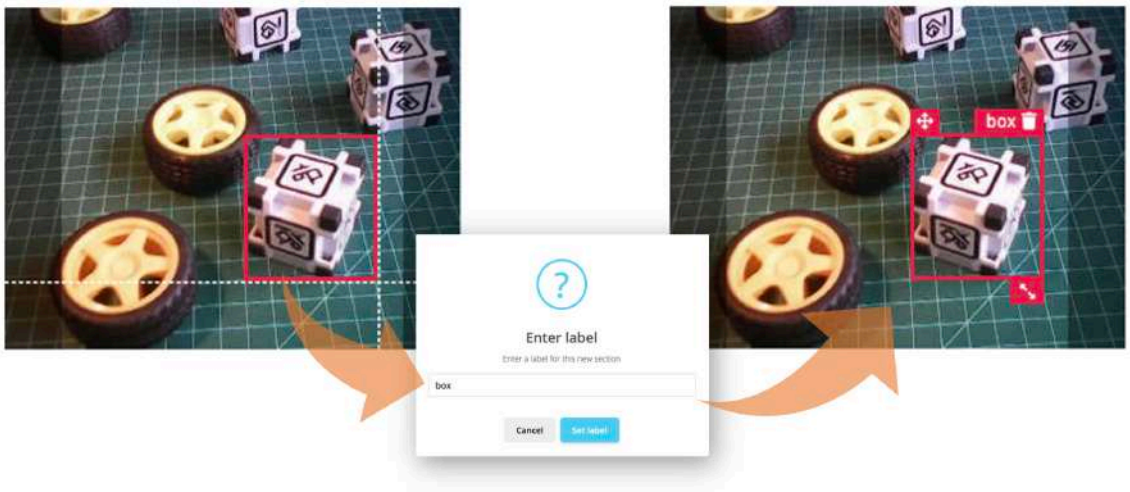
> Edge Impulse launched an auto-labeling feature for Enterprise customers, easing labeling tasks in object detection projects.

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).

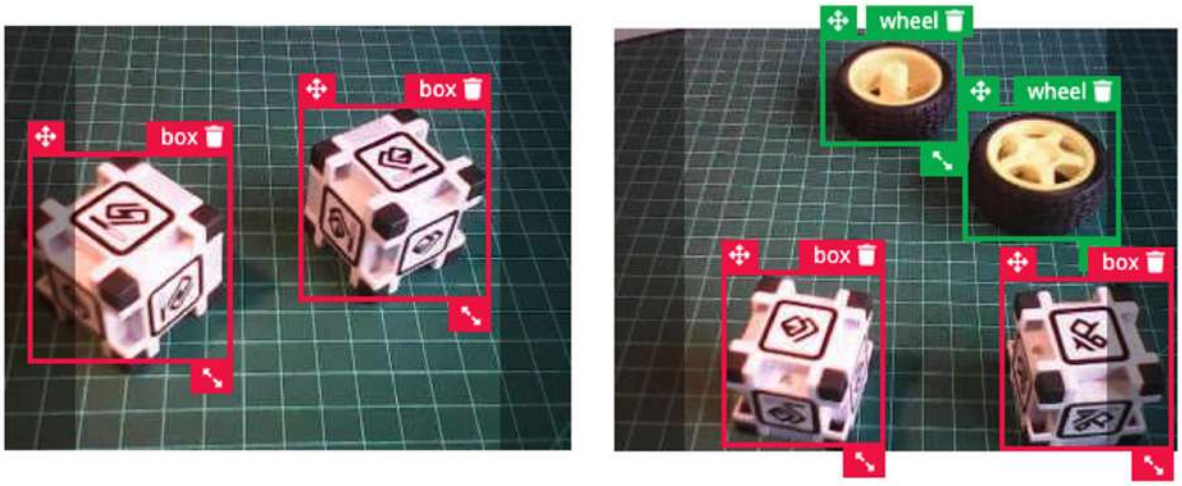
> If you already have a labeled dataset containing bounding boxes, import your data using the EI uploader.

Labeling the Dataset

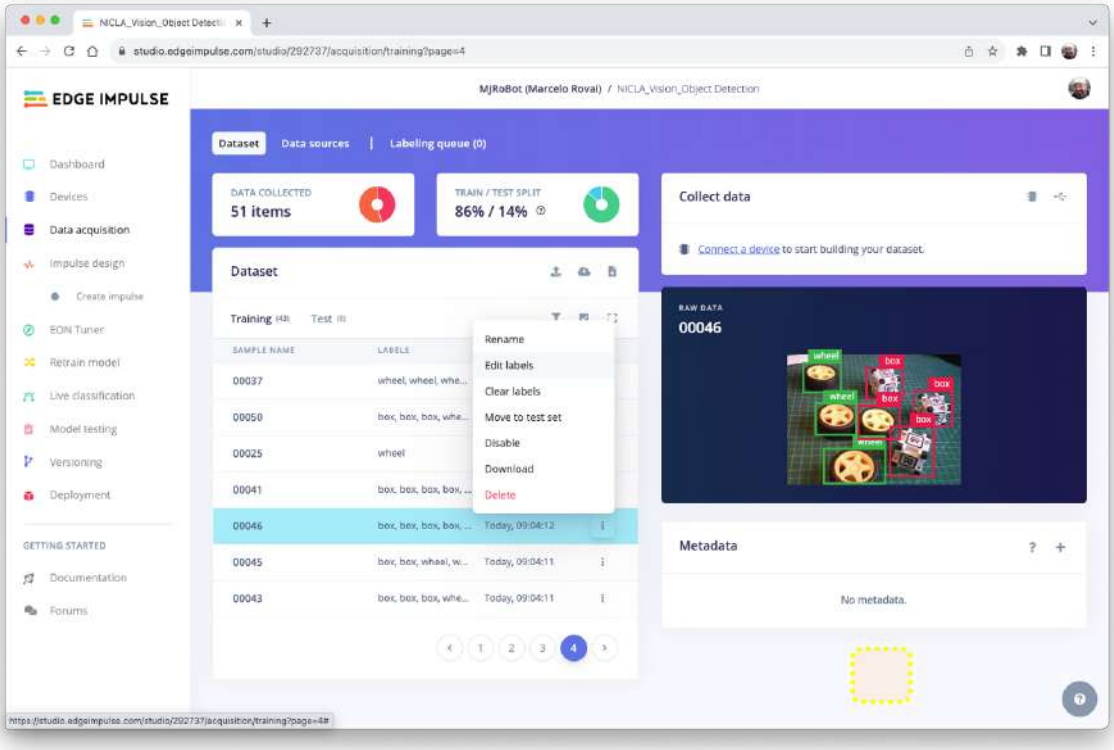
Starting with the first image of your unlabeled data, use your mouse to drag a box around an object to add a label. Then click **Save labels** to advance to the next item.



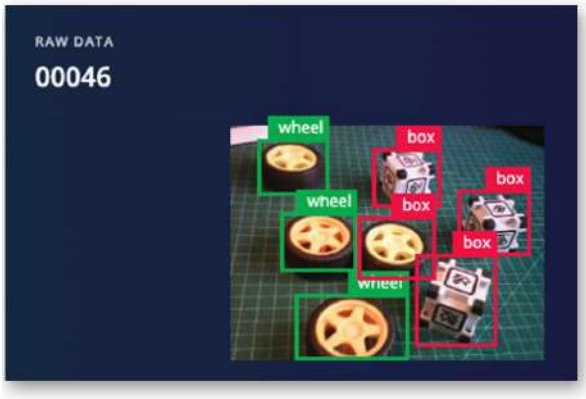
Continue with this process until the queue is empty. At the end, all images should have the objects labeled as those samples below:

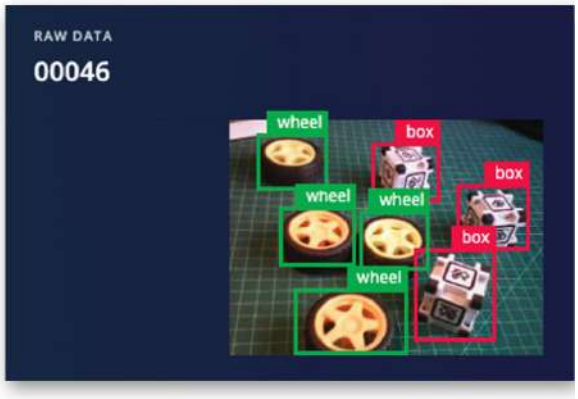


Next, review the labeled samples on the `Data acquisition` tab. If one of the labels is wrong, it can be edited using the *`three dots`* menu after the sample name:



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

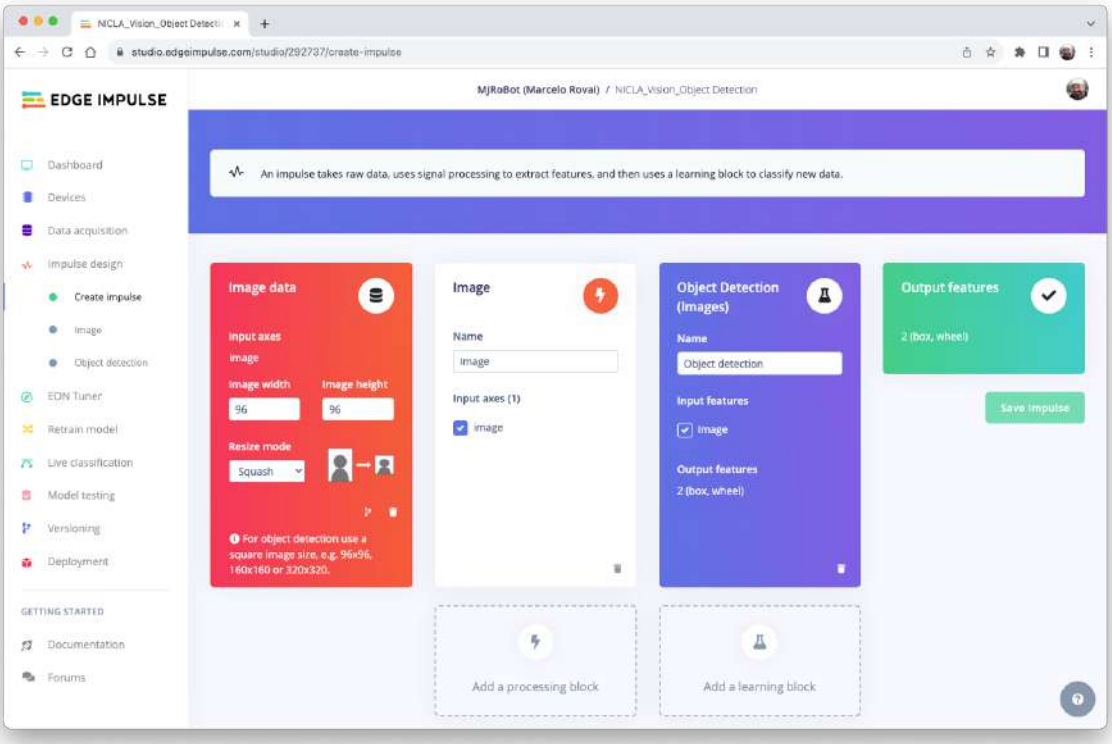




The Impulse Design

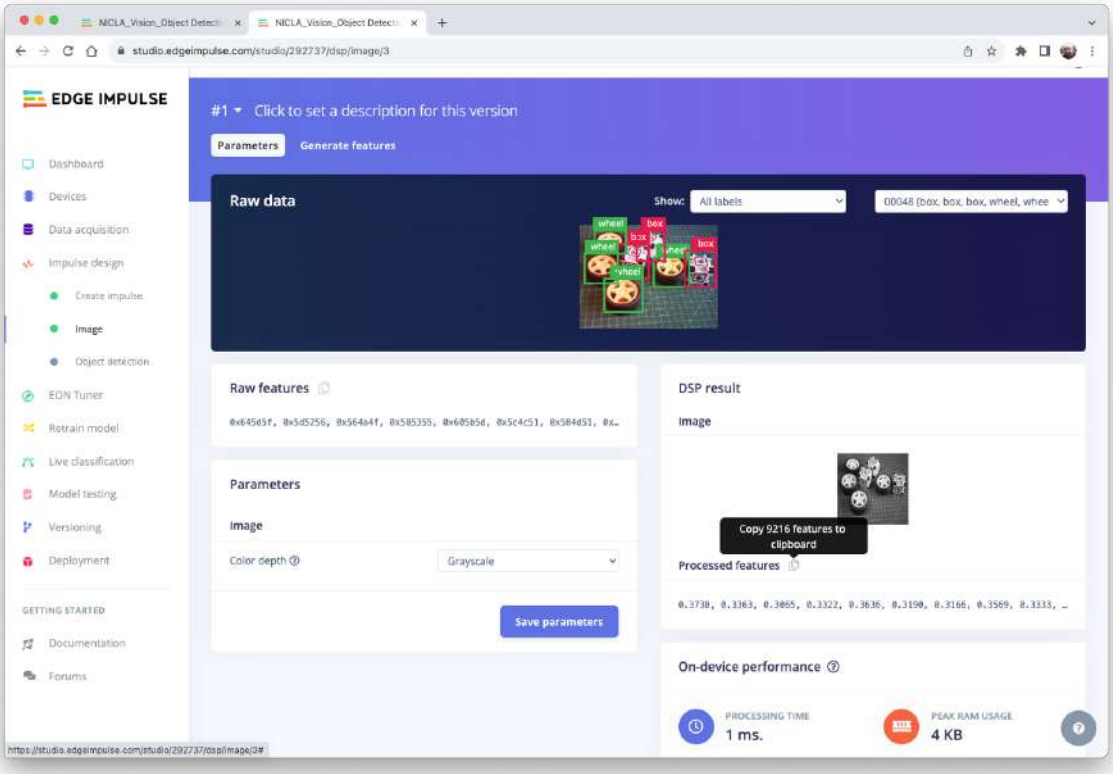
In this phase, we should define how to:

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

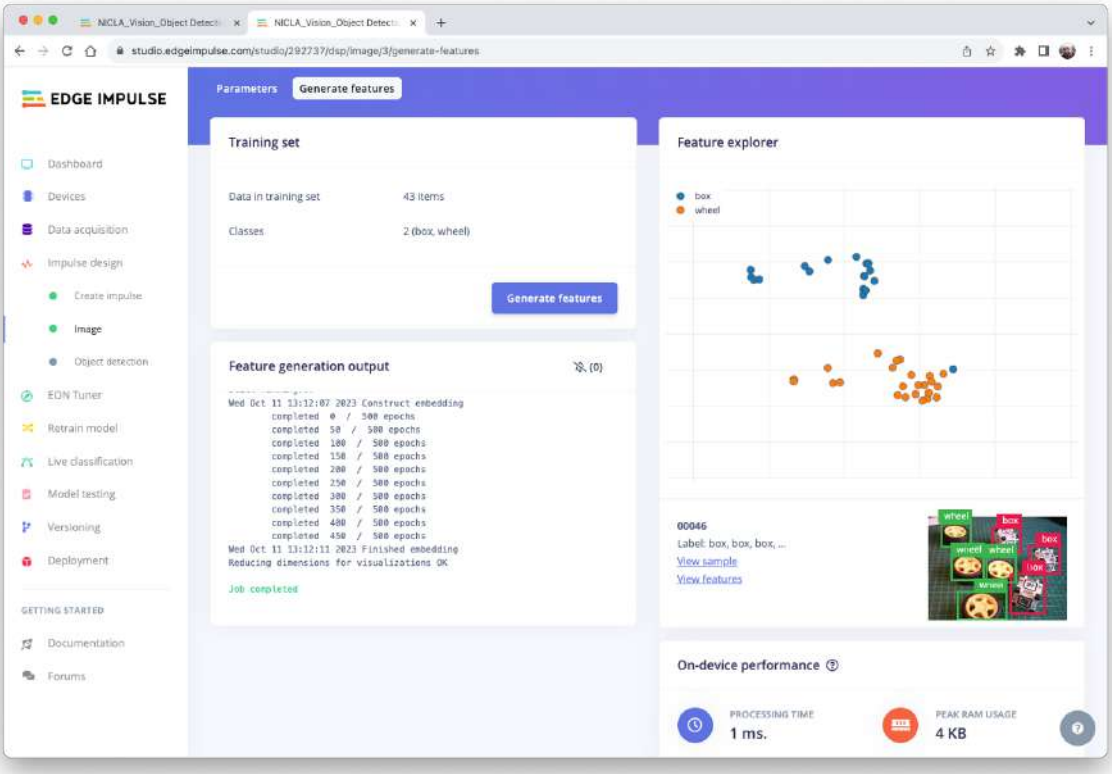


Preprocessing all dataset

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



The Studio moves automatically to the next section, `Generate features`, where all samples will be pre-processed, resulting in a dataset with individual $96 \times 96 \times 1$ 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) is apparently in the wrong space, but clicking on it confirms that the labeling is correct.

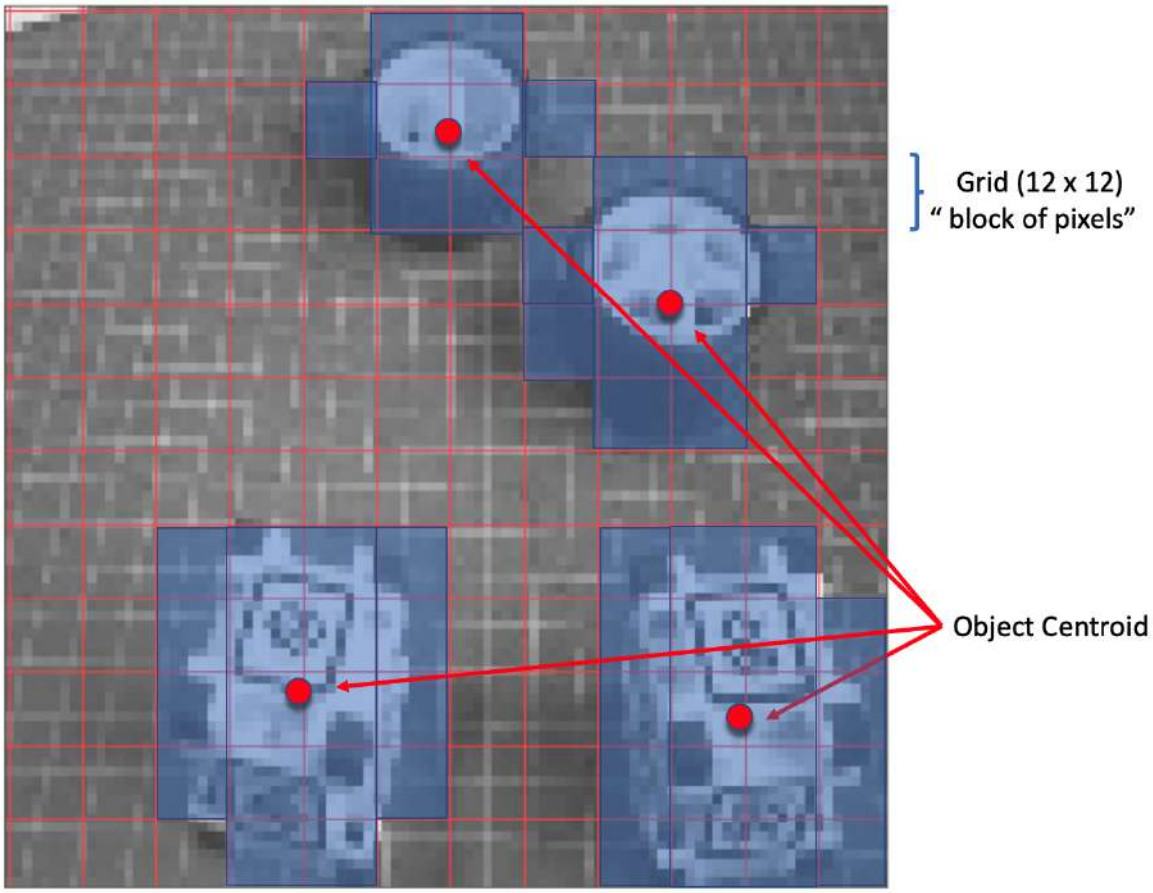
Model Design, Training, and Test

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*).

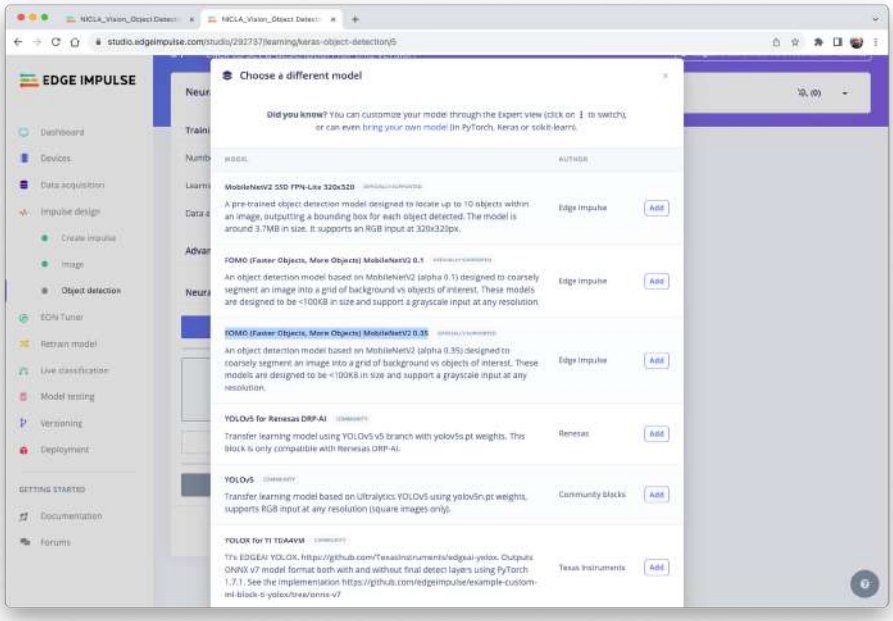
FOMO is an innovative machine learning model for object detection, which can use up to 30 times less energy and memory than traditional models like Mobilenet SSD and YOLOv5. FOMO can operate on microcontrollers with less than 200 KB of RAM. The main reason this is possible is that while other models 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 12×12 ($96/8 = 12$). Next, FOMO will run a classifier through each pixel block to calculate the probability that there is a box or a wheel in each of them and, subsequently, determine the regions that have the highest probability of containing the object (If a pixel block has no objects, it will be classified as *background*). From the overlap of the final region, the FOMO provides the coordinates (related to the image dimensions) of the centroid of this region.



For training, we should select a pre-trained model. Let's use the **`FOMO (Faster Objects, More Objects) MobileNetV2 0.35.`** This model uses around 250 KB of RAM and 80 KB of ROM (Flash), which suits well with our board since it has 1 MB of RAM and ROM.



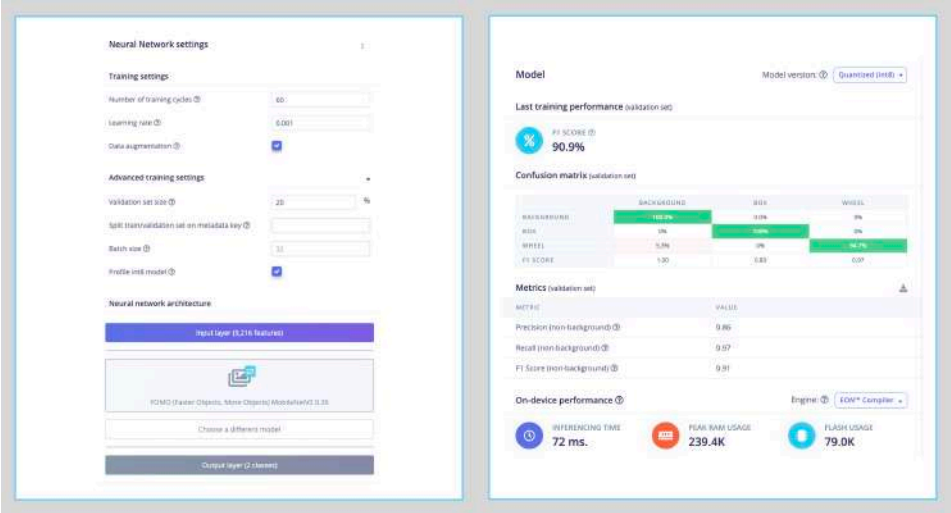
Regarding the training hyper-parameters, the model will be trained with:

- 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 an F1 score of around 91% (validation) and 93% (test data).

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



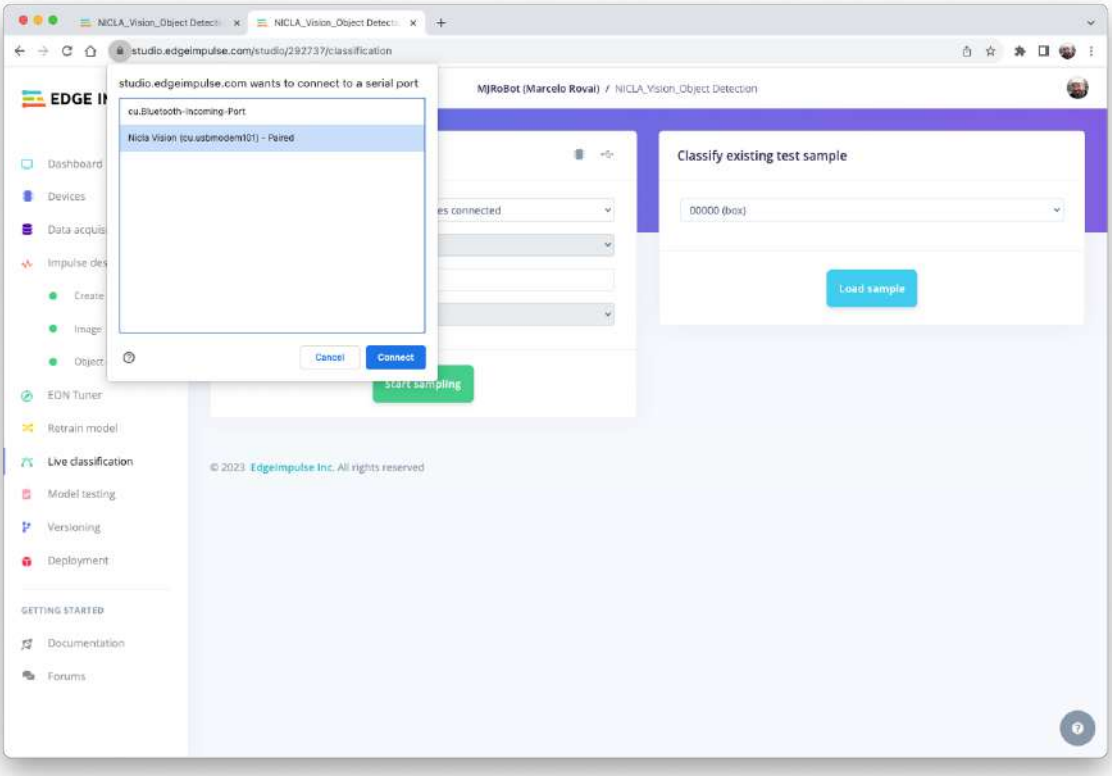
In object detection tasks, accuracy is generally not the primary evaluation metric. Object detection involves classifying objects and providing bounding boxes around them, making it a more complex problem than simple classification. The issue is that we do not have the bounding box, only the centroids. 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.

Test model with "Live Classification"

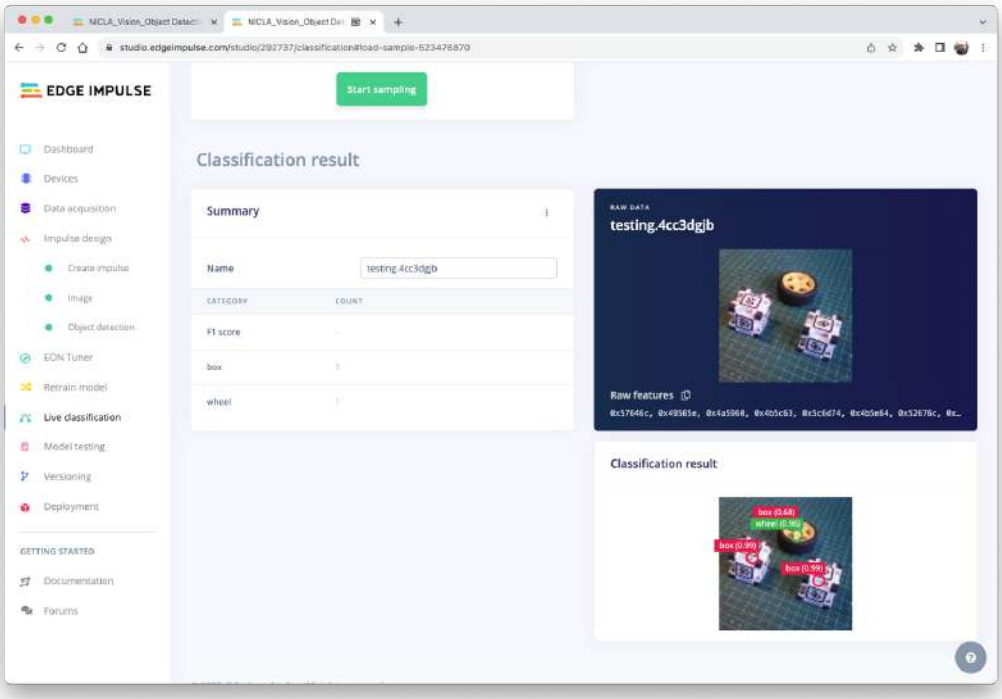
Since Edge Impulse officially supports the Nicla Vision, let's connect it to the Studio. For that, follow the steps:

- 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 to upload the binary (`arduino-nicla-vision.bin`) to your board.

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



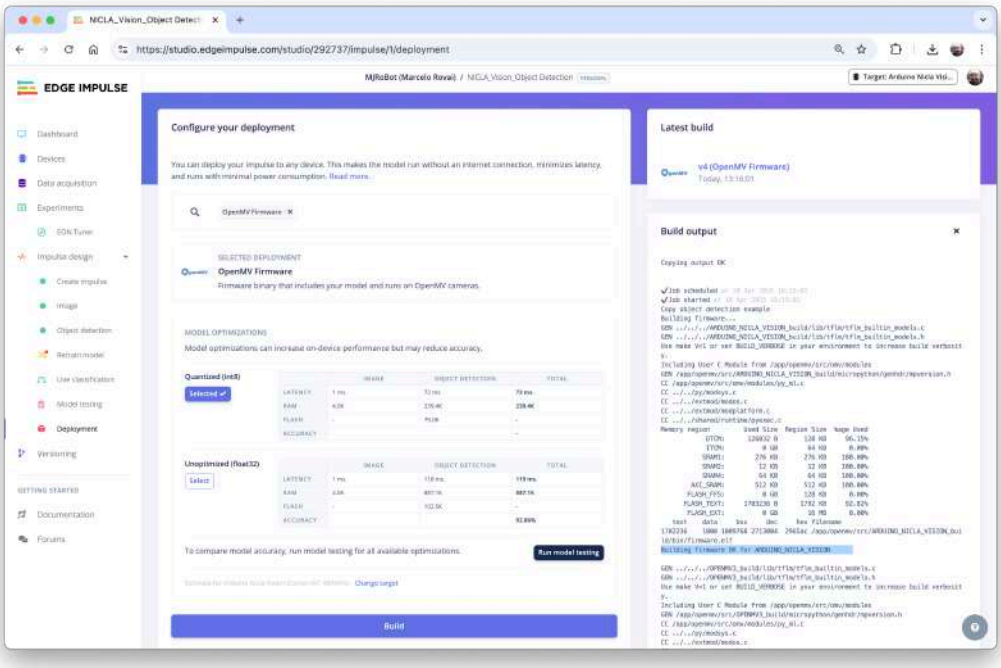
Once connected, you can use the Nicla to capture actual images to be tested by the trained model on Edge Impulse Studio.



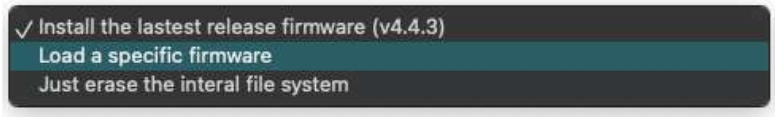
One thing to note is that the model can produce false positives and negatives. This can be minimized by defining a proper `Confidence Threshold` (use the `three dots` menu for the setup). Try with 0.8 or more.

Deploying the Model

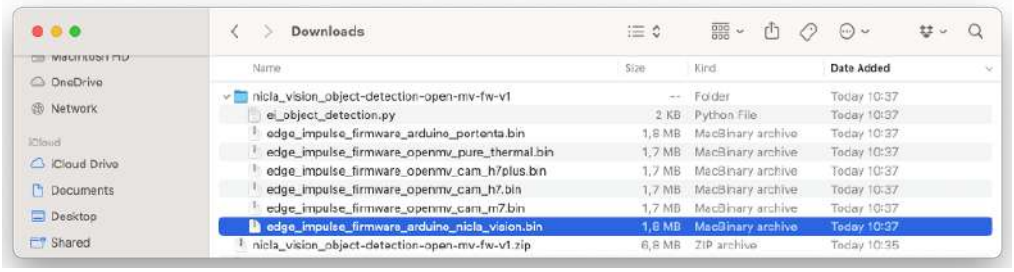
Select `OpenMV Firmware` on the Deploy Tab and press `[Build]`.



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. Or go to 'Tools > Runs Boatloader (Load Firmware).



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.

> Note: If a Pop-up appears saying that the FW is out of date, press `[NO]`, to upgrade it.

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

```
import sensor
import time
import ml
from ml.utils import NMS
import math
import image

sensor.reset()  # Reset and initialize the sensor.
sensor.set_pixformat(sensor.RGB565)  # Set pixel format (RGB565or GRAYSCALE)
sensor.set_framesize(sensor.QVGA)  # Set frame size to QVGA (320x240)
sensor.skip_frames(time=2000)  # Let the camera adjust.
```

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.

```
threshold_list = [(math.ceil(min_confidence * 255), 255)]

# Load built-in model
model = ml.Model("trained")
print(model)
```

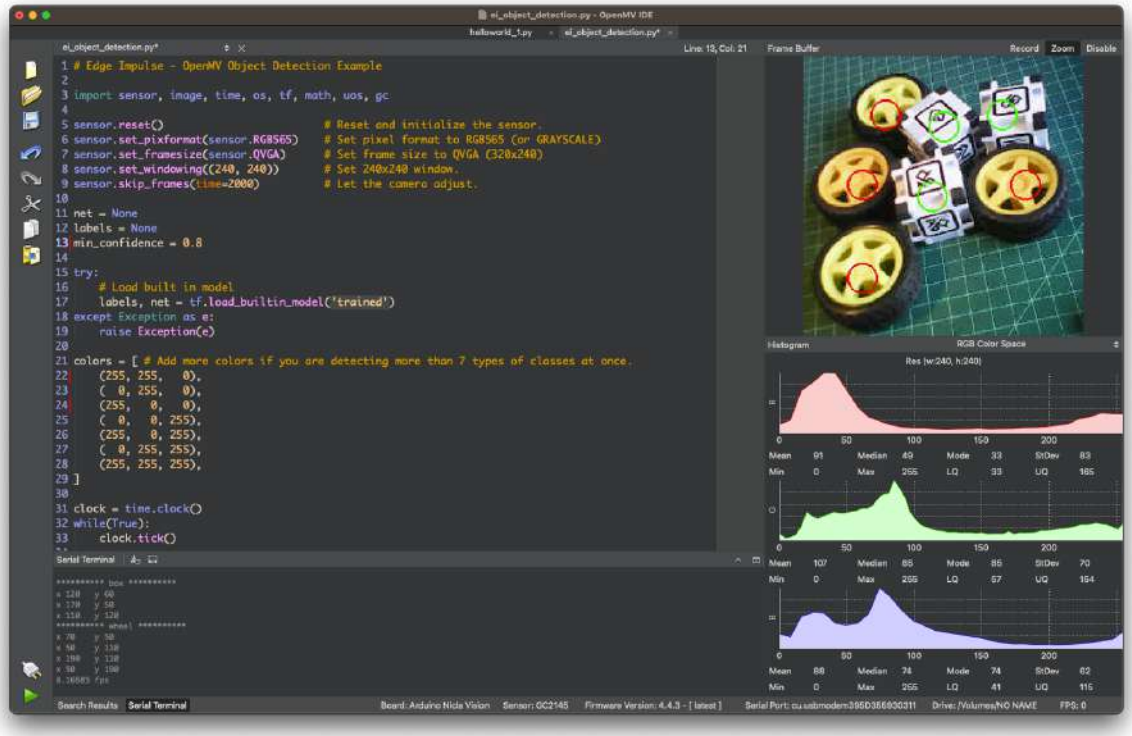
```
# Alternatively, models can be loaded from the filesystem storage.
# model = ml.Model('<object_detection_modelwork>.tflite', load_to_fb=True)
# labels = [line.rstrip('\n') for line in open("labels.txt")]

colors = [ # Add more colors if you are detecting more
           # than 7 types of classes at once.
    (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

```
# FOMO outputs an image per class where each pixel in the image is the centroid of the tra
# object. So, we will get those output images and then run find_blobs() on them to extract
# centroids. We will also run get_stats() on the detected blobs to determine their score.
# The Non-Max-Supression (NMS) object then filters out overlapping detections and maps the
# position in the output image back to the original input image. The function then returns
# list per class which each contain a list of (rect, score) tuples representing the detect
# objects.

def fomo_post_process(model, inputs, outputs):
    n, oh, ow, oc = model.output_shape[0]
    nms = NMS(ow, oh, inputs[0].roi)
    for i in range(oc):
        img = image.Image(outputs[0][0, :, :, i] * 255)
        blobs = img.find_blobs(
            threshold_list, x_stride=1, area_threshold=1, pixels_threshold=1
        )
        for b in blobs:
            rect = b.rect()
            x, y, w, h = rect
            score = (
                img.get_statistics(thresholds=threshold_list, roi=rect).l_mean() / 255.0
            )
            nms.add_bounding_box(x, y, x + w, y + h, score, i)
    return nms.get_bounding_boxes()
```

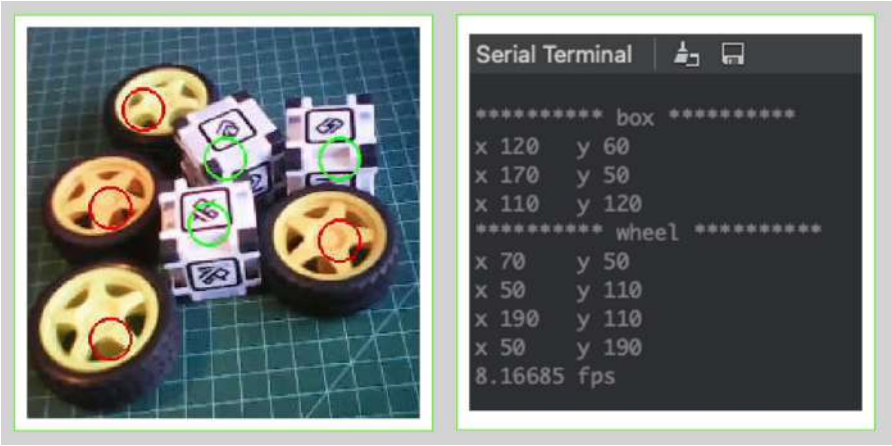
```
clock = time.clock()
while True:
    clock.tick()

    img = sensor.snapshot()

    for i, detection_list in enumerate(model.predict([img], callback=fomo_post_process)):
        if i == 0:
            continue  # background class
        if len(detection_list) == 0:
            continue  # no detections for this class?

        print("********** %s **********" % model.labels[i])
        for (x, y, w, h), score in detection_list:
            center_x = math.floor(x + (w / 2))
            center_y = math.floor(y + (h / 2))
            print(f"x {center_x}\ty {center_y}\tscore {score}")
            img.draw_circle((center_x, center_y, 12), color=colors[i])

    print(clock.fps(), "fps", end="\n")
```

and press the `green Play button` to run the code:



From the camera's 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 (240×240).

Be aware 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 or YOLO. For example, when running a MobileNetV2 SSD FPN-Lite 320 × 320 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

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. This can be very useful on projects counting bees, for example.

Resources

- Edge Impulse Project

KWS Feature Engineering

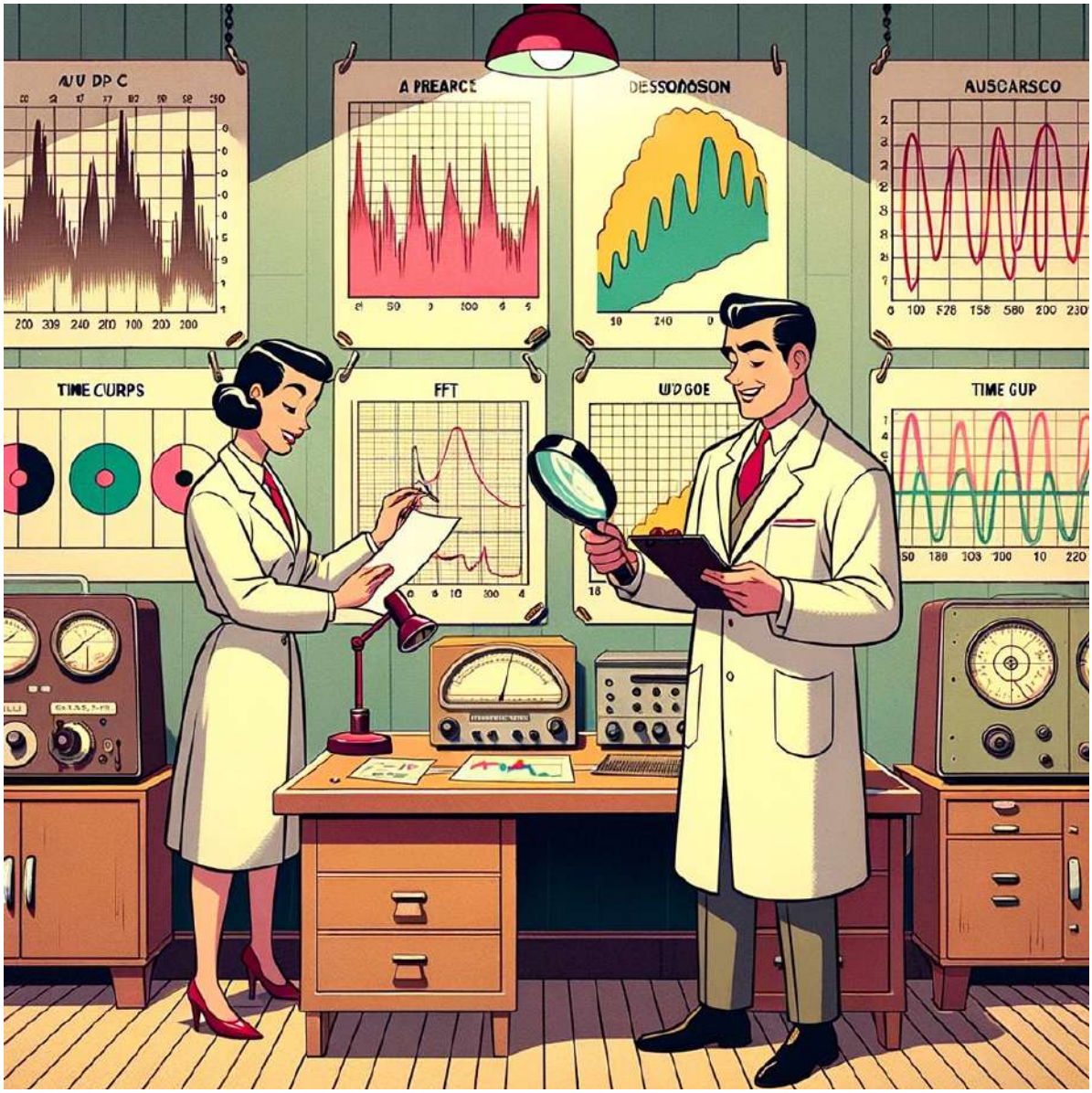


Figure 4: *DALL · E 3 Prompt: 1950s style cartoon scene set in an audio research room. Two scientists, one holding a magnifying glass and the other taking notes, examine large charts pinned to the wall. These charts depict FFT graphs and time curves related to audio data analysis. The room has a retro ambiance, with wooden tables, vintage lamps, and classic audio analysis tools.*

Overview

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.

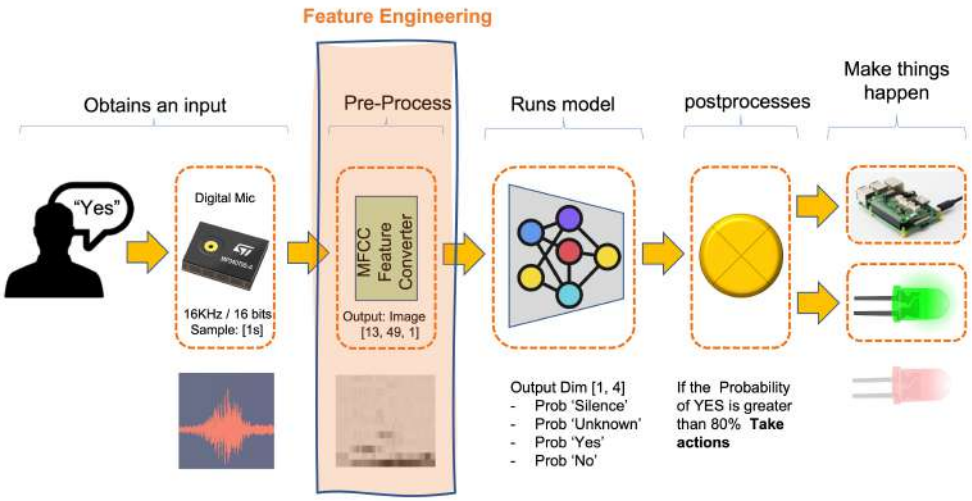
Machine learning models, especially traditional algorithms, don't understand audio waves. They understand numbers arranged in some meaningful way, i.e., features. These features encapsulate the characteristics of the audio signal, making it easier for models to distinguish between different sounds.

> 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

The KWS

The most common TinyML application is Keyword Spotting (KWS), a subset of the broader field of speech recognition. While general speech recognition transcribes 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:



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.

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.

Overview to Audio Signals

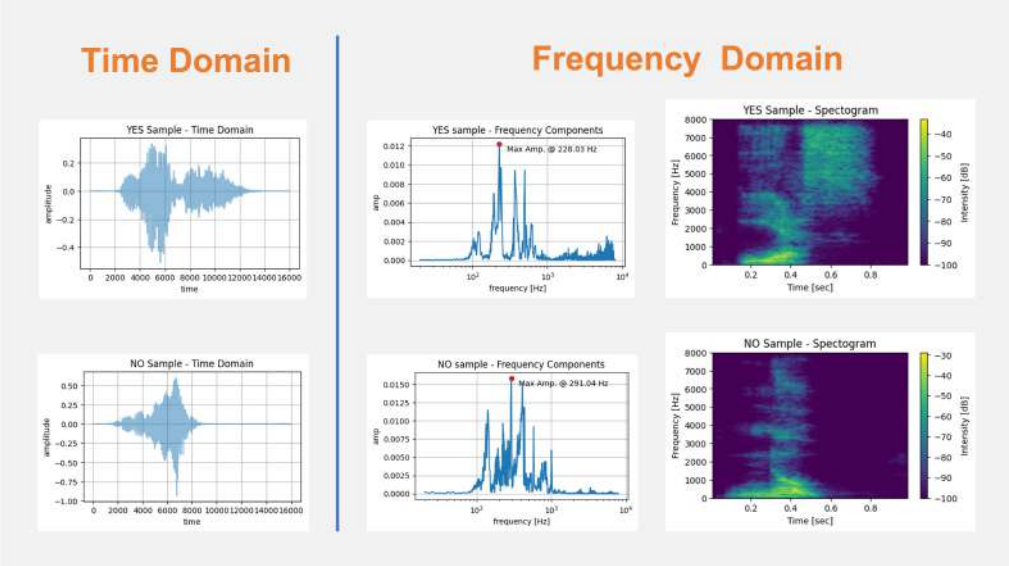
Understanding the basic properties of audio signals is crucial for effective feature extraction and, ultimately, for successfully applying machine learning algorithms in audio classification tasks. Audio signals are complex waveforms that capture fluctuations in air pressure over time. These signals 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 16 kHz. Although 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:



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.

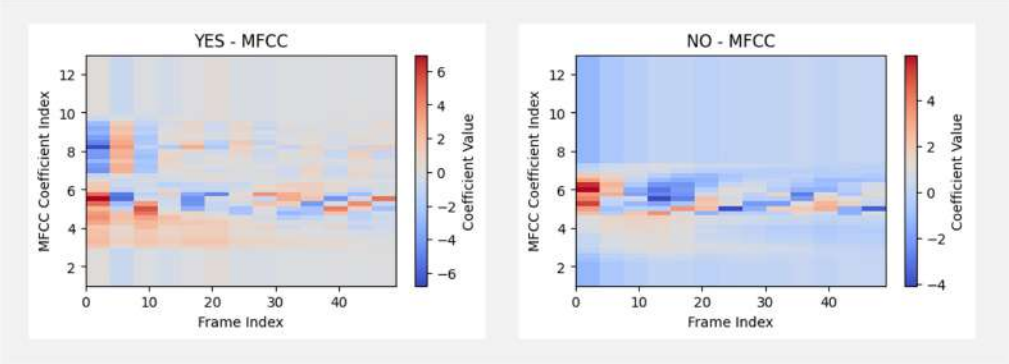
For these reasons, feature extraction techniques such as Mel-frequency Cepstral Coefficients (MFCCs), Mel-Frequency Energies (MFEs), and simple Spectograms are commonly used to transform raw audio data into a more manageable and informative format. These features capture the essential characteristics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

Overview to MFCCs

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.

Why are MFCCs important?

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

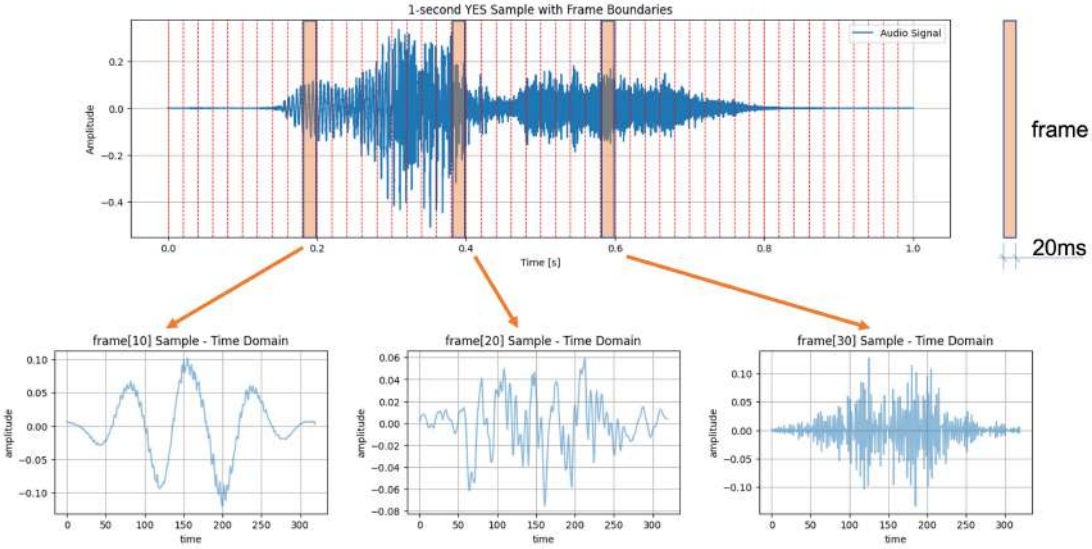
- **Dimensionality Reduction**: MFCCs capture essential spectral characteristics of the audio signal while significantly reducing the dimensionality of the data, making it ideal for resource-constrained TinyML applications.
- **Robustness**: MFCCs are less susceptible to noise and variations in pitch 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.

Computing MFCCs

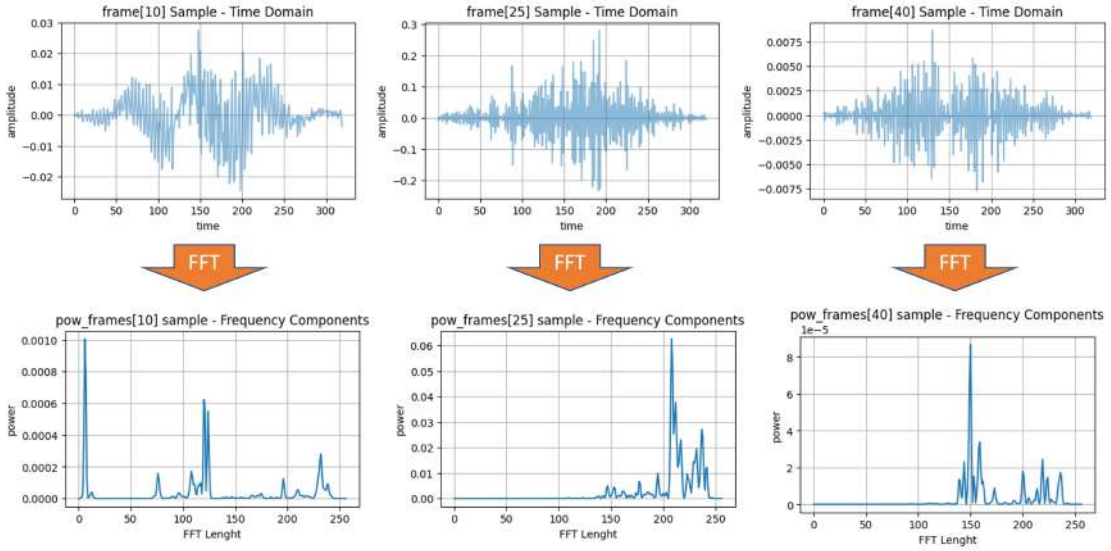
The computation of Mel-frequency Cepstral Coefficients (MFCCs) involves several key steps. Let's walk through these, which are particularly important for Keyword Spotting (KWS) tasks on TinyML devices.

- **Pre-emphasis**: The first step is pre-emphasis, which is applied to accentuate the high-frequency components of the audio signal and balance the frequency spectrum. This is achieved by applying a filter that amplifies the difference between consecutive samples. The formula for pre-emphasis is: $y(t) = x(t) - \alpha x(t-1)$, where α is the pre-emphasis factor, typically around 0.97.
- **Framing**: Audio signals are divided into short frames (the *frame length*), usually 20 to 40 milliseconds. This is based on the assumption that frequencies in a signal are stationary over a short period. Framing helps in analyzing the signal in such small time slots. The *frame stride* (or step) will displace one frame and the adjacent. Those steps could be sequential or overlapped.
- **Windowing**: Each frame is then windowed to minimize the discontinuities at the frame boundaries. A commonly used window function is the Hamming window. Windowing prepares the signal for a Fourier transform by minimizing the edge effects. The image below shows three frames (10, 20, and 30) and the time samples after windowing (note that the frame length and frame stride are 20 ms):

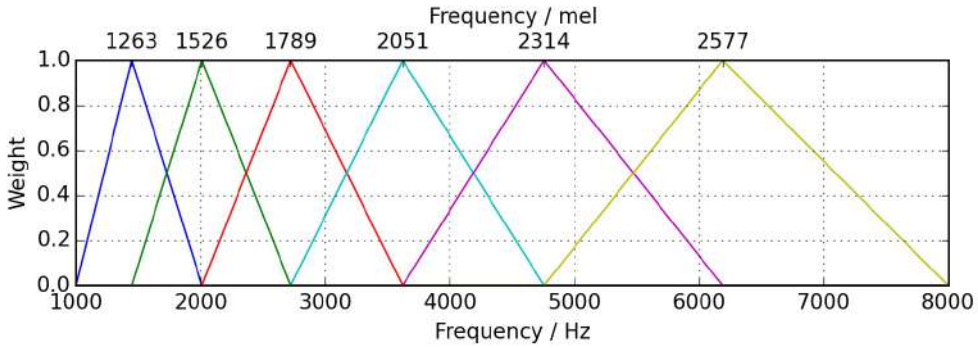


- **Fast Fourier Transform (FFT)** The Fast Fourier Transform (FFT) is applied to each windowed frame to convert it from the time domain to the frequency domain. The FFT gives us a complex-valued representation that includes both magnitude and phase information. However, for MFCCs, only the magnitude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

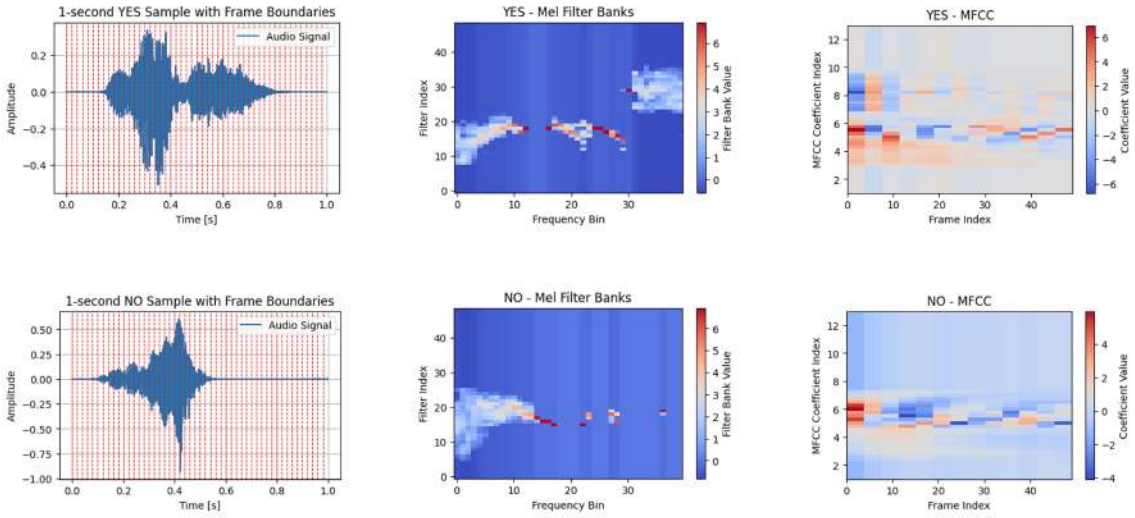
The power spectrum $P(f)$ of a signal $x(t)$ is defined as $P(f) = |X(f)|^2$, where $X(f)$ is the Fourier Transform of $x(t)$. By squaring the magnitude of the Fourier Transform, we emphasize *stronger* frequencies over *weaker* ones, thereby capturing more relevant spectral characteristics of the audio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



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



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



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]

Conclusion

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.

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.

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.

Resources

- Audio_Data_Analysis Colab Notebook

Keyword Spotting (KWS)

Figure 5: *DALL · E 3 Prompt: 1950s style cartoon scene set in a vintage audio research room. Two Afro-American female scientists are at the center. One holds a magnifying glass, closely examining ancient circuitry, while the other takes notes. On their wooden table, there are multiple boards with sensors, notably featuring a microphone. Behind these boards, a computer with a large, rounded back displays the Arduino IDE. The IDE showcases code for LED pin assignments and machine learning inference for voice command detection. A distinct window in the IDE, the Serial Monitor, reveals outputs indicating the spoken commands 'yes' and 'no'. The room ambiance is nostalgic with vintage lamps, classic audio analysis tools, and charts depicting FFT graphs and time-domain curves.*

Overview

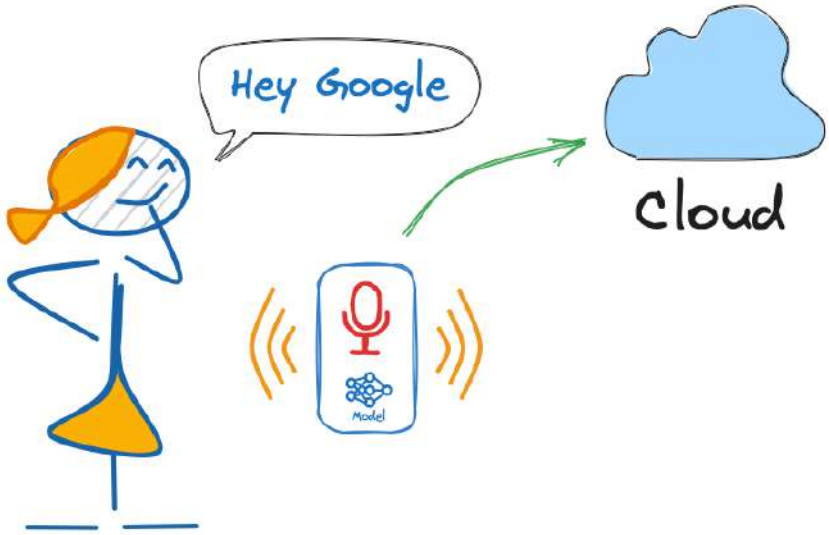
Having already explored the Nicla Vision board in the *Image Classification* and *Object Detection* applications, we are now shifting our focus to voice-activated applications with a project on Keyword Spotting (KWS).

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.

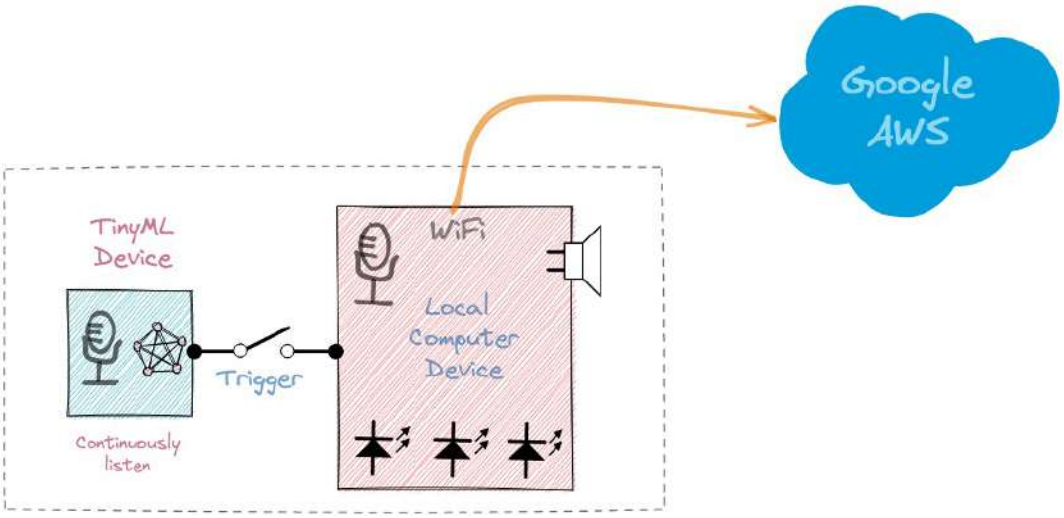
Our model will be designed to recognize keywords that can trigger device wake-up or specific actions, bringing them to life with voice-activated commands.

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).

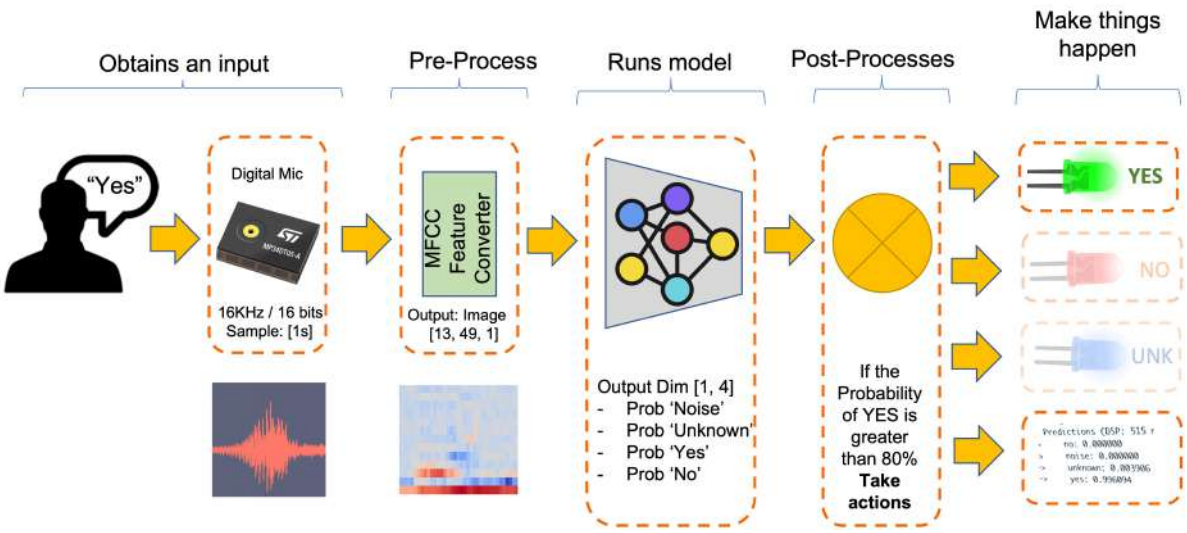
https://youtu.be/e_OPgcnsyvM

> 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.

The KWS Hands-On Project

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



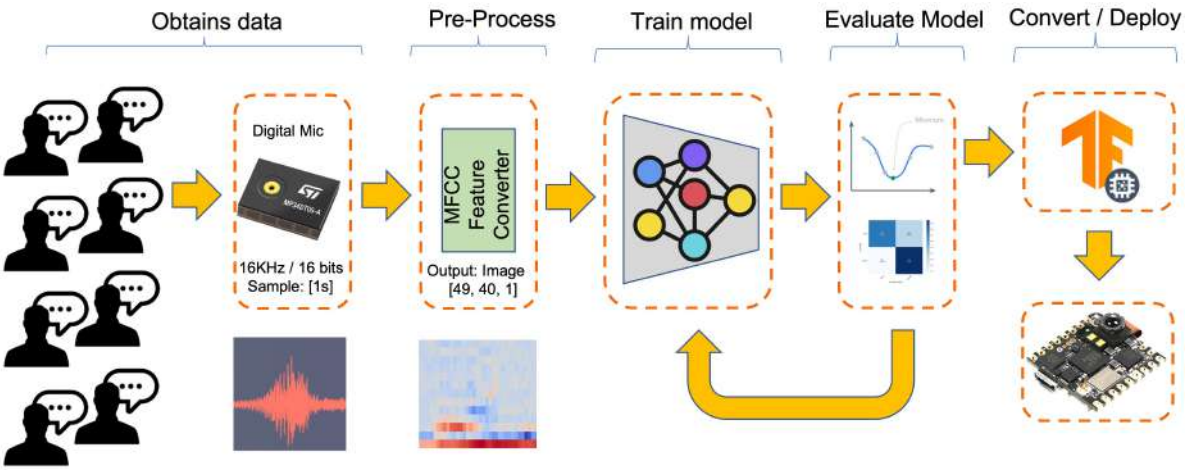
Our KWS application will recognize four classes of sound:

- **YES** (Keyword 1)
- **NO** (Keyword 2)
- **NOISE** (no words spoken; only background noise is present)
- **UNKNOWN** (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."

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"):



Dataset

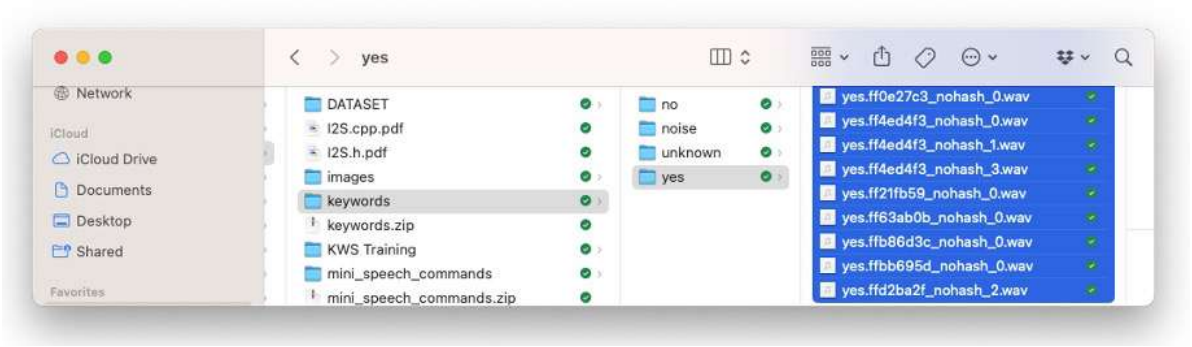
The critical component of any Machine Learning Workflow is the **dataset**. Once we have decided on specific keywords, in our case (*YES* and NO), we can take advantage of the dataset developed by Pete Warden, "Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition." This dataset has 35 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:

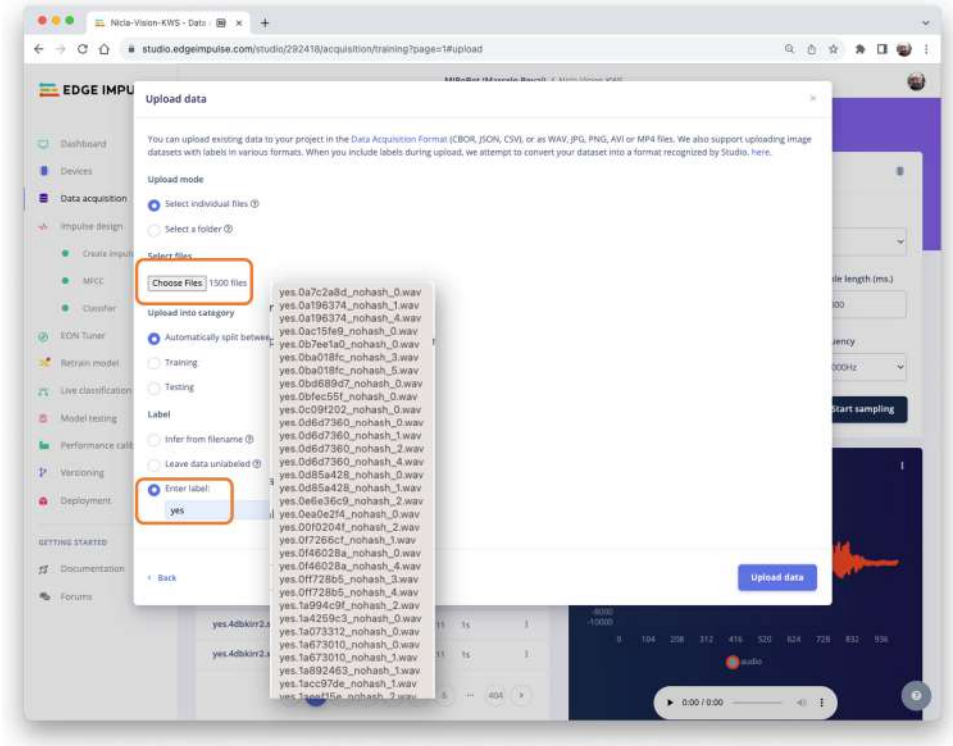
- Download the keywords dataset.
- Unzip the file to a location of your choice.

Uploading the dataset to the Edge Impulse Studio

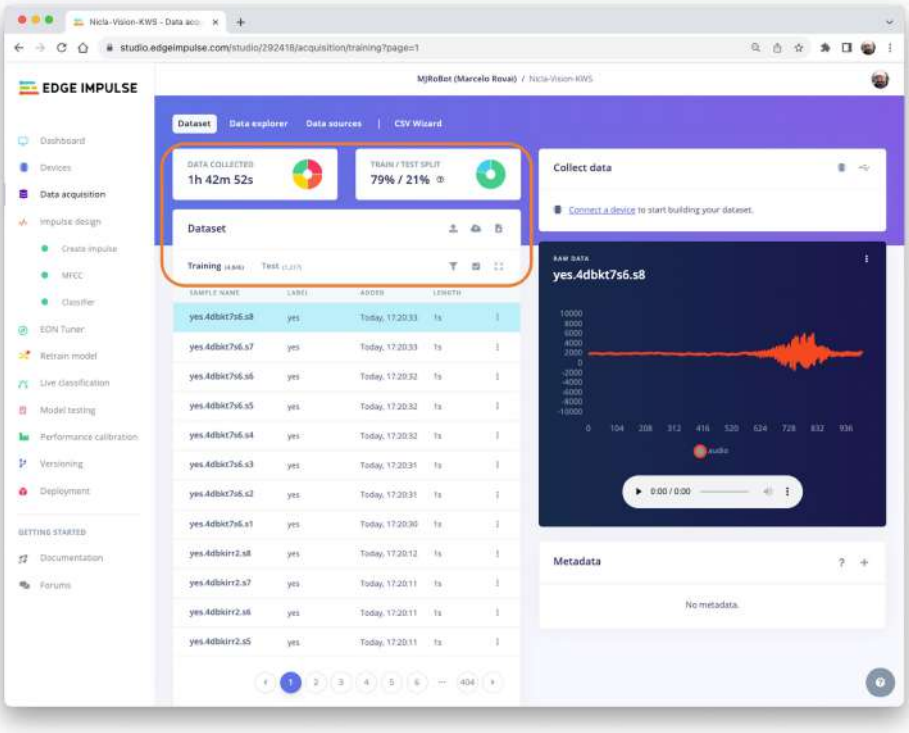
Initiate a new project at Edge Impulse Studio (EIS) and select the `Upload Existing Data` tool in the `Data Acquisition` section. Choose the files to be uploaded:



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



The 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.



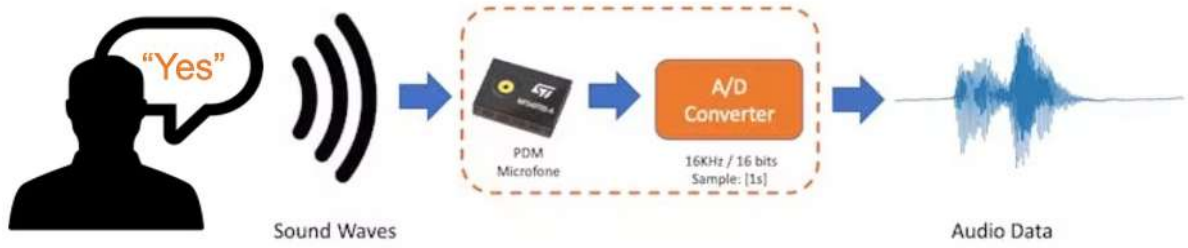
Capturing additional Audio Data

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 16 KHz frequency with 16-bit per sample amplitude.

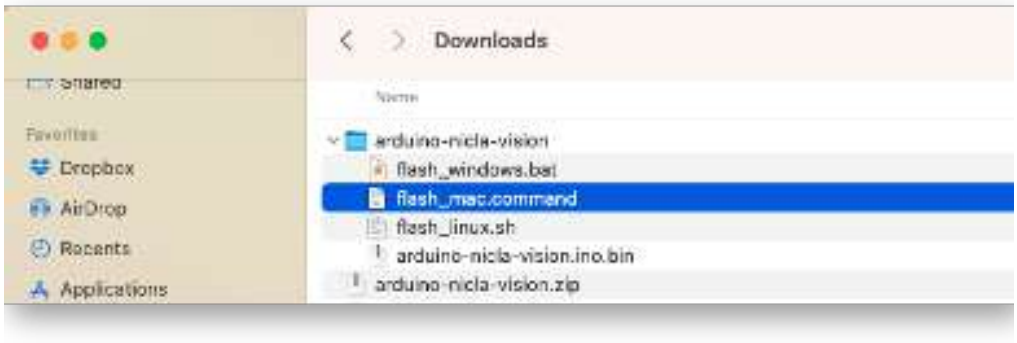
So, any device that can generate audio data with this basic specification (16 KHz/16 bits) will work fine. As a *device*, we can use the NiclaV, a computer, or even your mobile phone.



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:

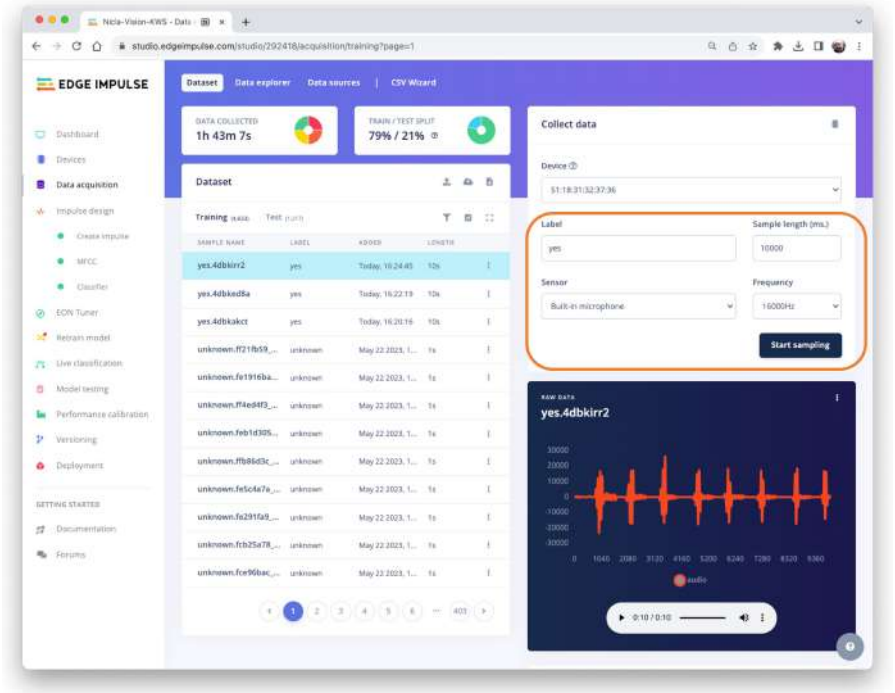
- Download the last updated EIS Firmware and unzip it.
- Open the zip file on your computer and select the uploader corresponding to your OS:



- 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.

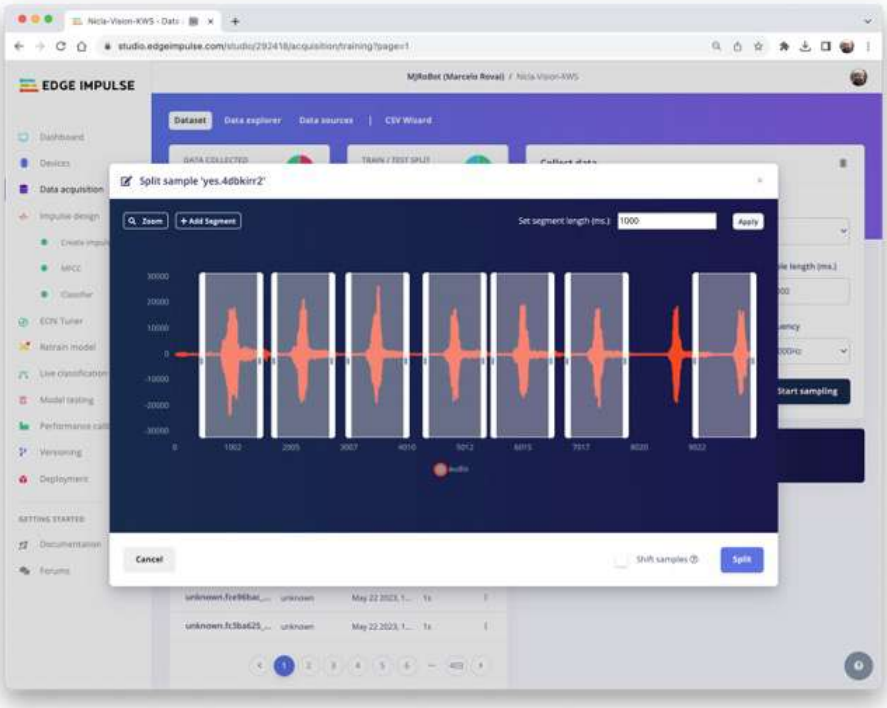
Go to your project on EIS, and on the `Data Acquisition tab`, select `WebUSB`. A window will pop up; choose the option that shows that the `Nicla is paired` and press `[Connect]`.

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`.



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.

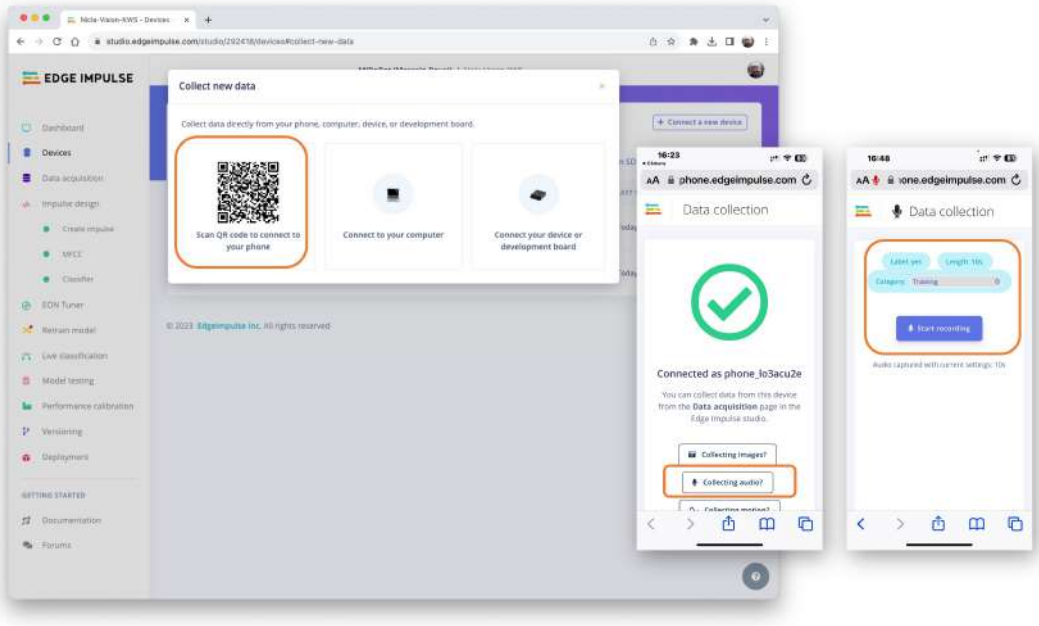


Once inside the tool, split the data into 1-second (1000 ms) records. If necessary, add or remove segments. This procedure should be repeated for all new samples.

Using a smartphone and the EI Studio

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



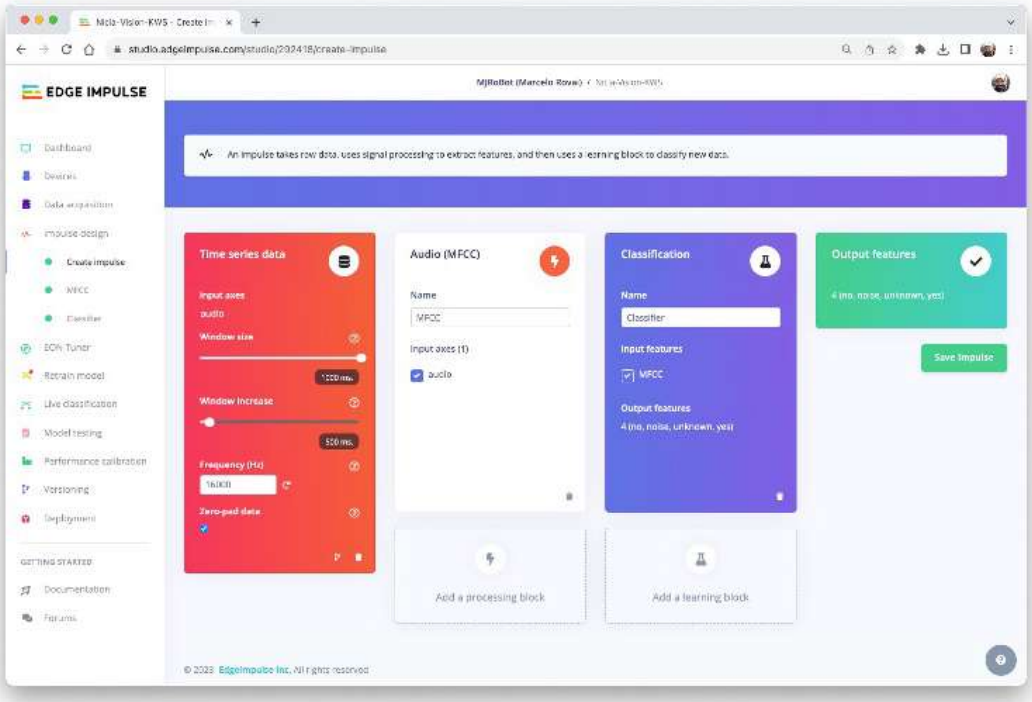
Repeat the same procedure used with the NiclaV.

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

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.*

Impulse Design



First, we will take the data points with a 1-second window, augmenting the data and sliding that window in 500 ms 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, $13 \times 49 \times 1$). 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.

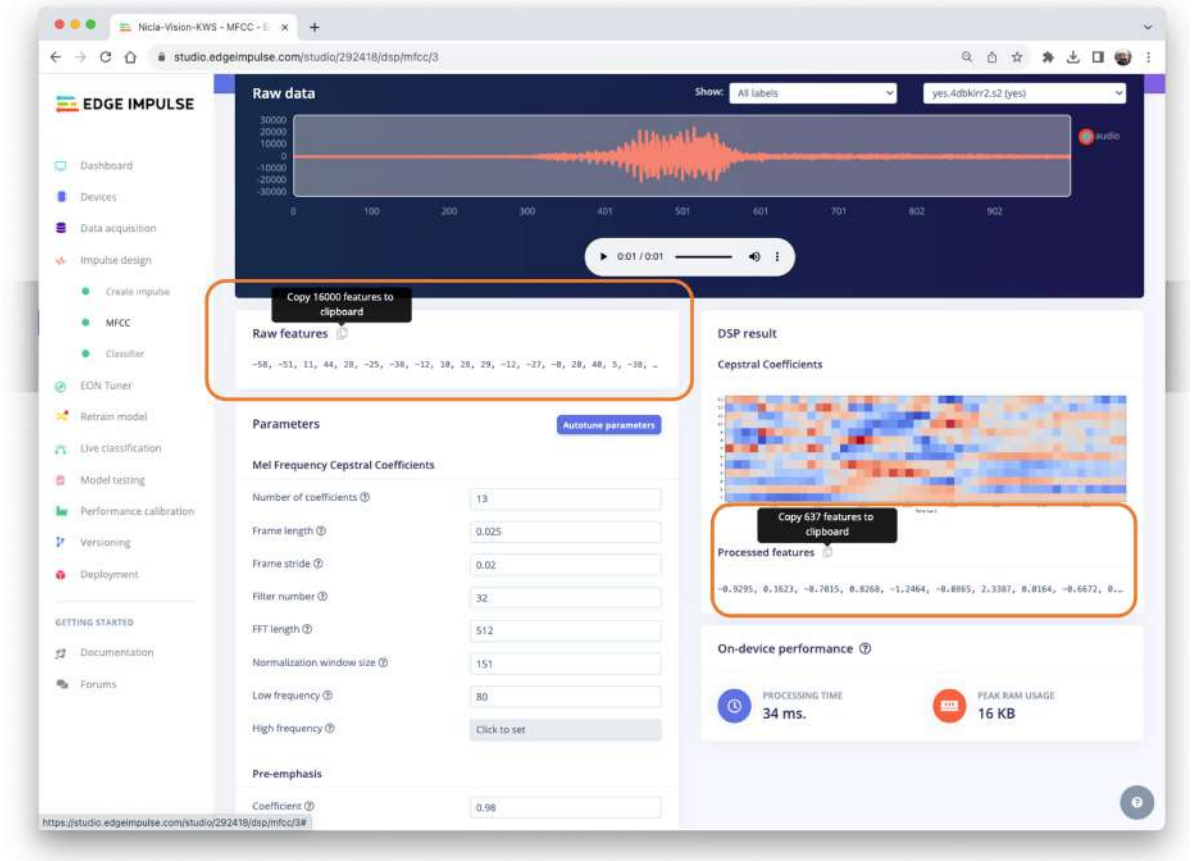
Next, we select the `Classification` block to build our model from scratch using a Convolution Neural Network (CNN).

> 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.

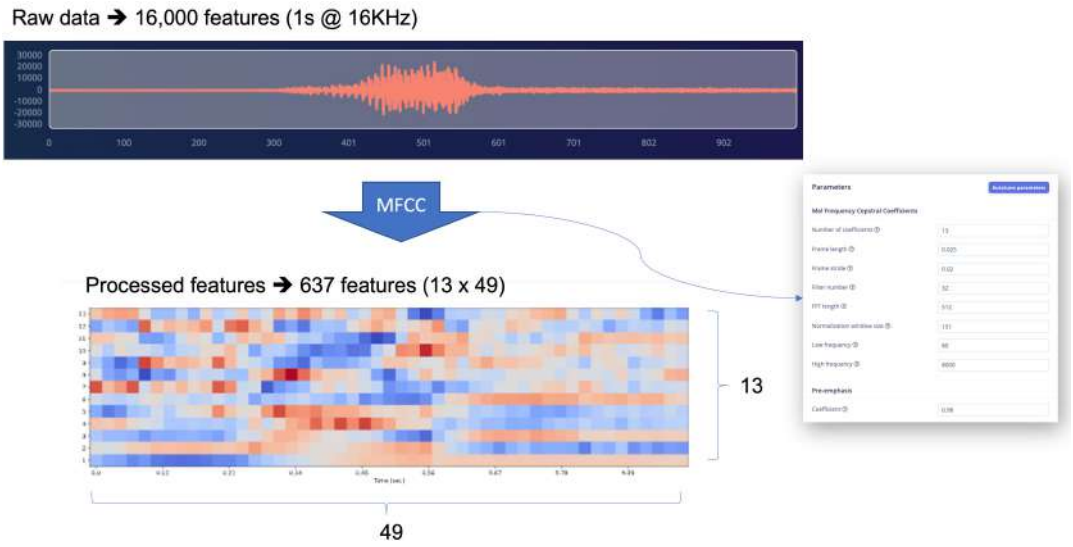
Pre-Processing (MFCC)

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.

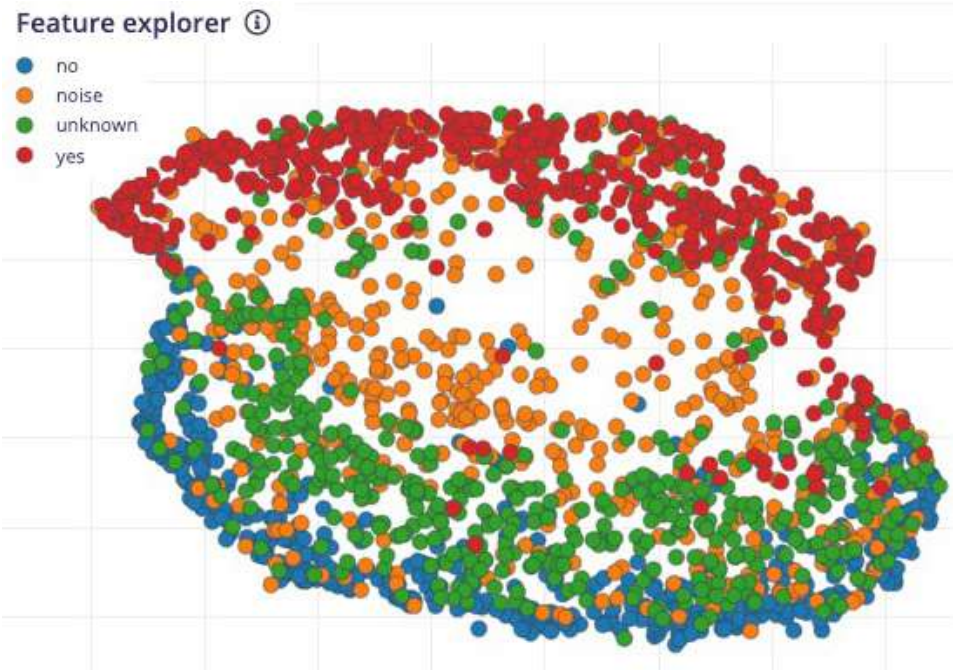


We will take the `Raw features` (our 1-second, 16 KHz sampled audio data) and use the MFCC processing block to calculate the `Processed features`. For every 16,000 raw features (16,000 × 1 second), we will get 637 processed features (13 × 49).



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



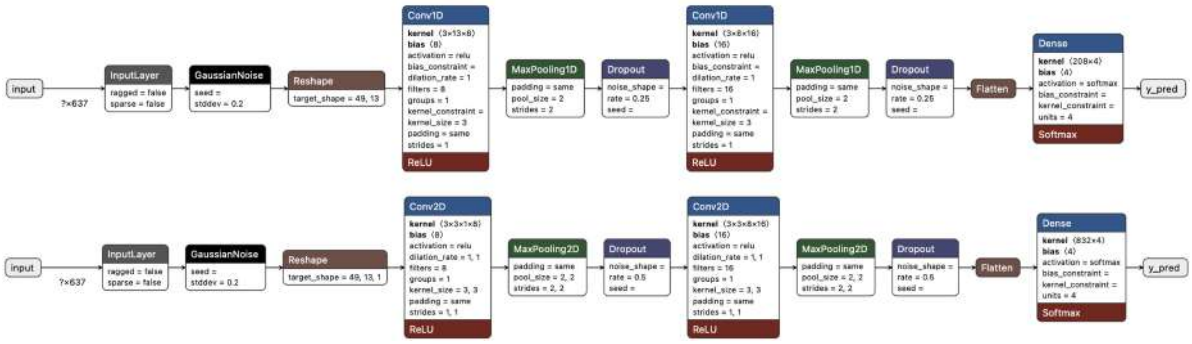
The result seems OK, with a visually clear separation between *yes* features (in red) and *no* features (in blue). The *unknown* features seem nearer to the *no space* than the *yes*. This suggests that the keyword *no* has more propensity to false positives.

Going under the hood

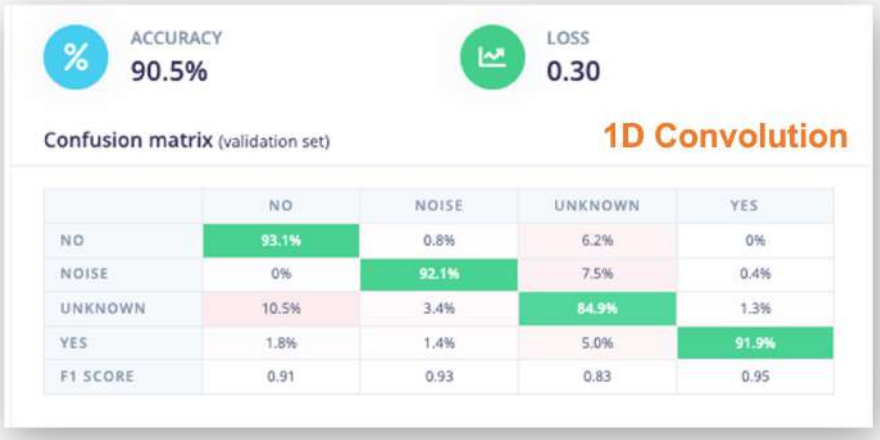
To understand better how the raw sound is preprocessed, look at the *Feature Engineering for Audio Classification* chapter. You can play with the MFCC features generation by downloading this notebook from GitHub or [Opening it In Colab]

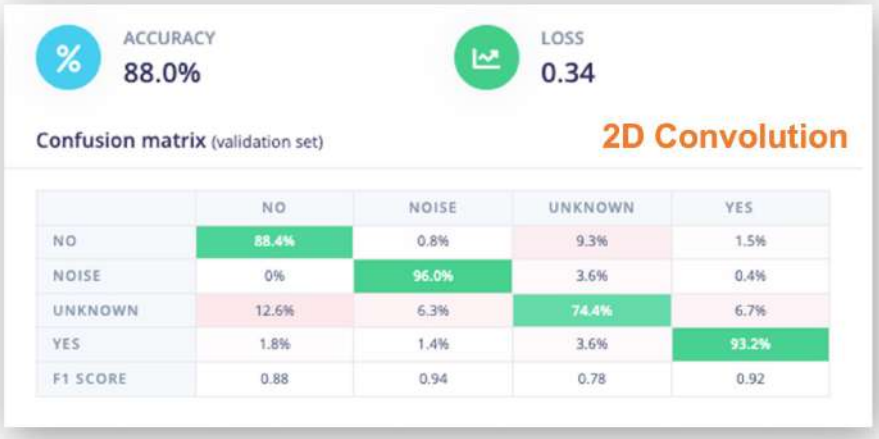
Model Design and Training

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.





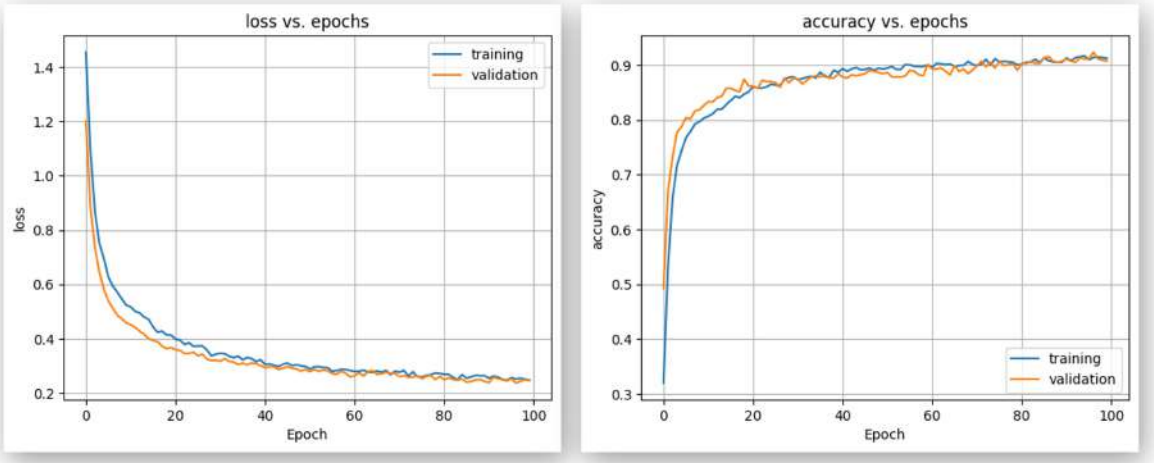
> Using 1D convolutions is more efficient because it requires fewer parameters than 2D convolutions, making them more suitable for resource-constrained environments.

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.

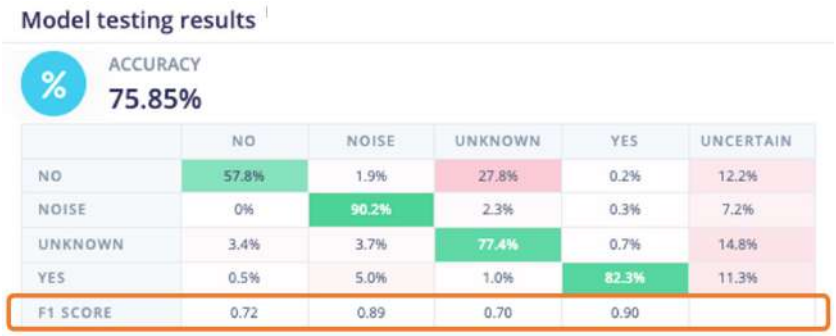
Going under the hood

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:



Testing

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



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.

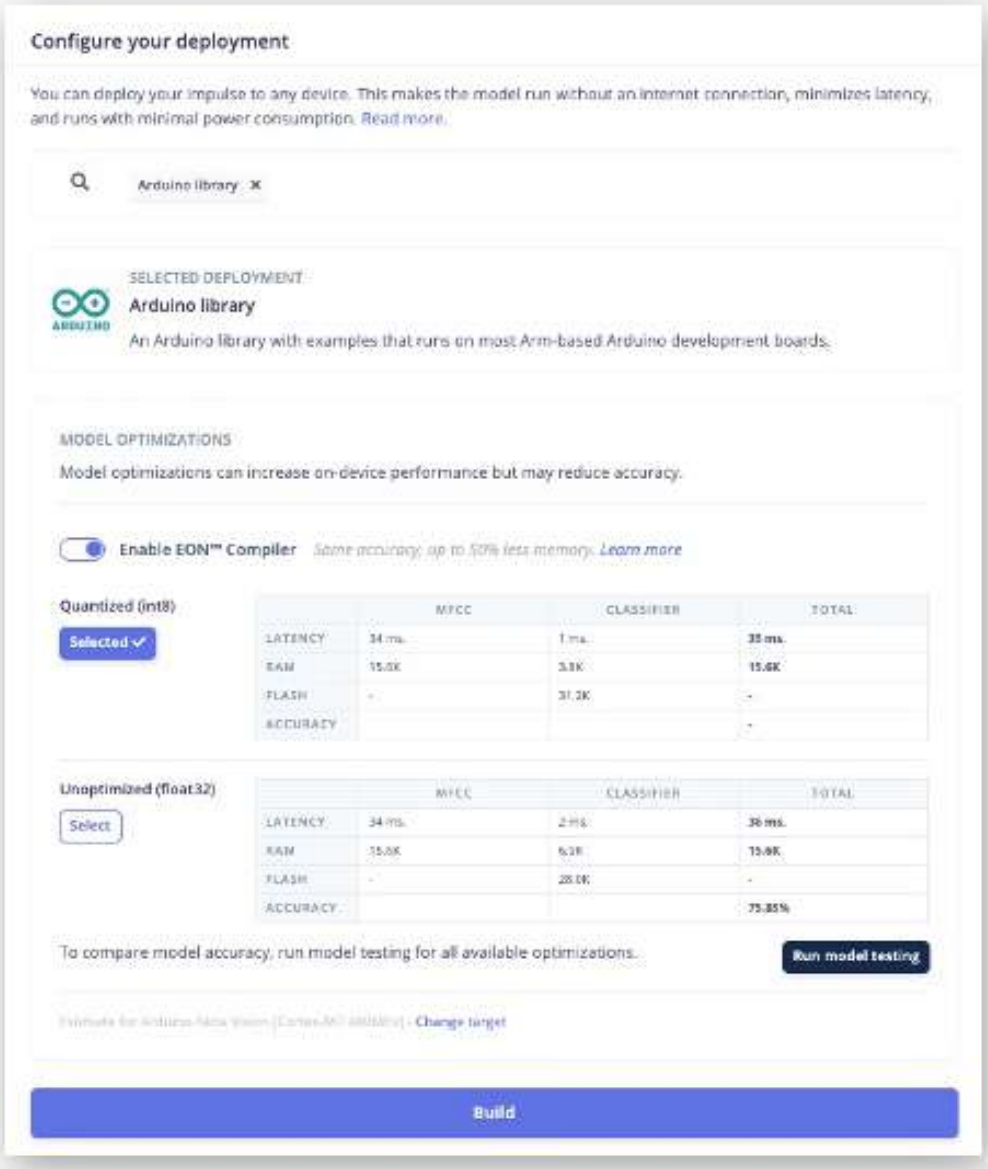
For NO, we got 0.72, which was expected, but to improve this result, we can move the samples that were not correctly classified to the training dataset and then repeat the training process.

Live Classification

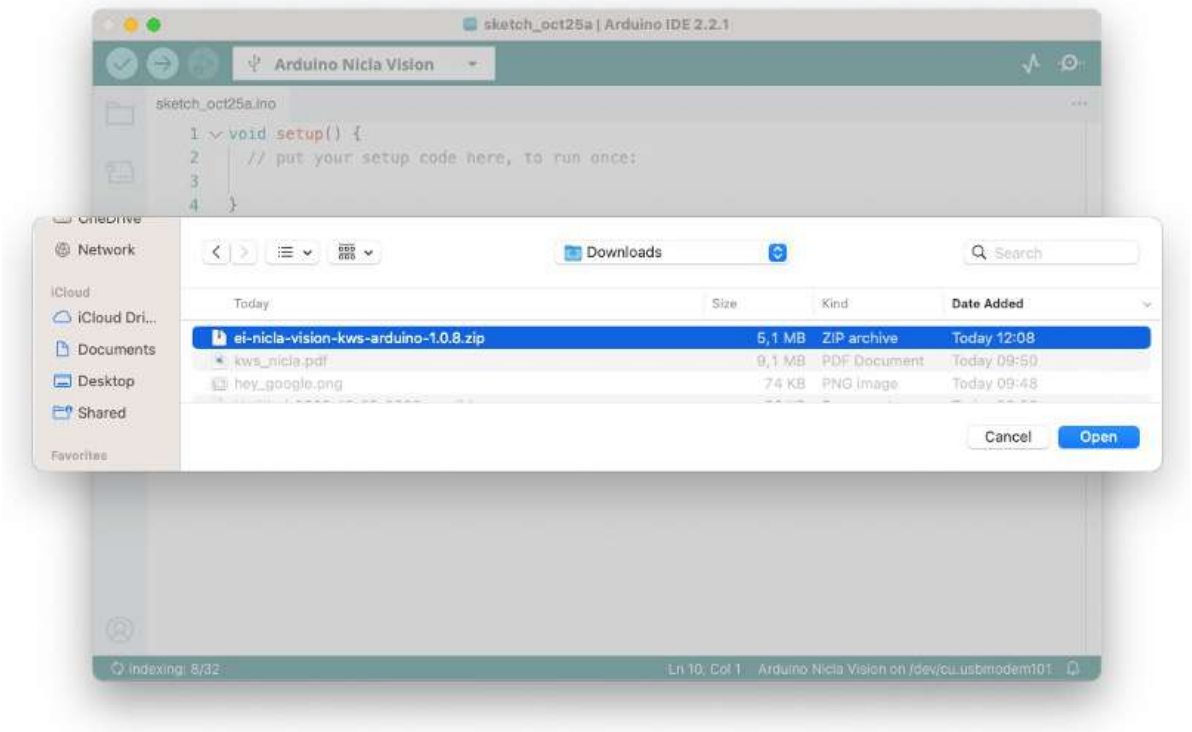
We can proceed to the project's next step but also consider that it is possible to perform `Live Classification` using the NiclaV or a smartphone to capture live samples, testing the trained model before deployment on our device.

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`.

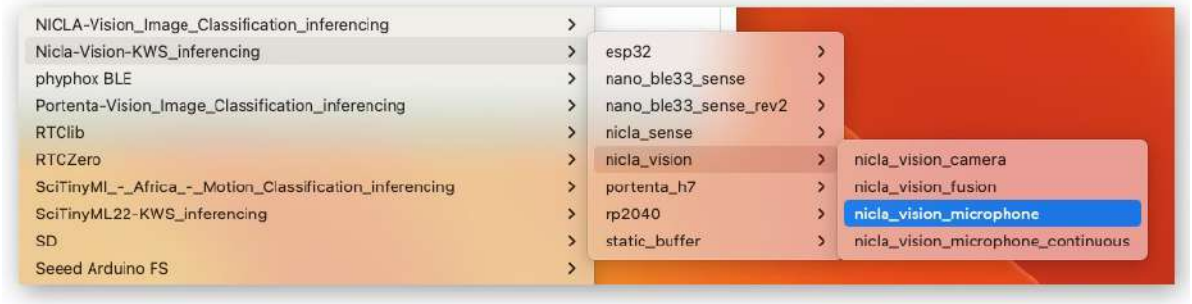


When the `Build` button is selected, a zip file will be created and downloaded to your computer. On your Arduino IDE, go to the `Sketch` tab, select the option `Add .ZIP Library`, and Choose the .zip file downloaded by EIS:



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:

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 UNKNOWN, 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 part of the code
    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(LEDR, OUTPUT);
    pinMode(LEDG, OUTPUT);
    pinMode(LEDB, 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(LEDR, HIGH);
    digitalWrite(LEDG, HIGH);
    digitalWrite(LEDB, 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(LEDR, HIGH);
    digitalWrite(LEDG, HIGH);
    digitalWrite(LEDB, 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(LEDR, LOW);
      break;

    case 1:
      turn_off_leds();
      break;

    case 2:
      turn_off_leds();
      digitalWrite(LEDB, LOW);
      break;

    case 3:
      turn_off_leds();
      digitalWrite(LEDG, 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 lit 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

Conclusion

> You will find the notebooks and codeused 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)

Resources

- Subset of Google Speech Commands Dataset
- KWS MFCC Analysis Colab Notebook
- KWS_CNN_training Colab Notebook
- Arduino Post-processing Code
- Edge Impulse Project

DSP Spectral Features

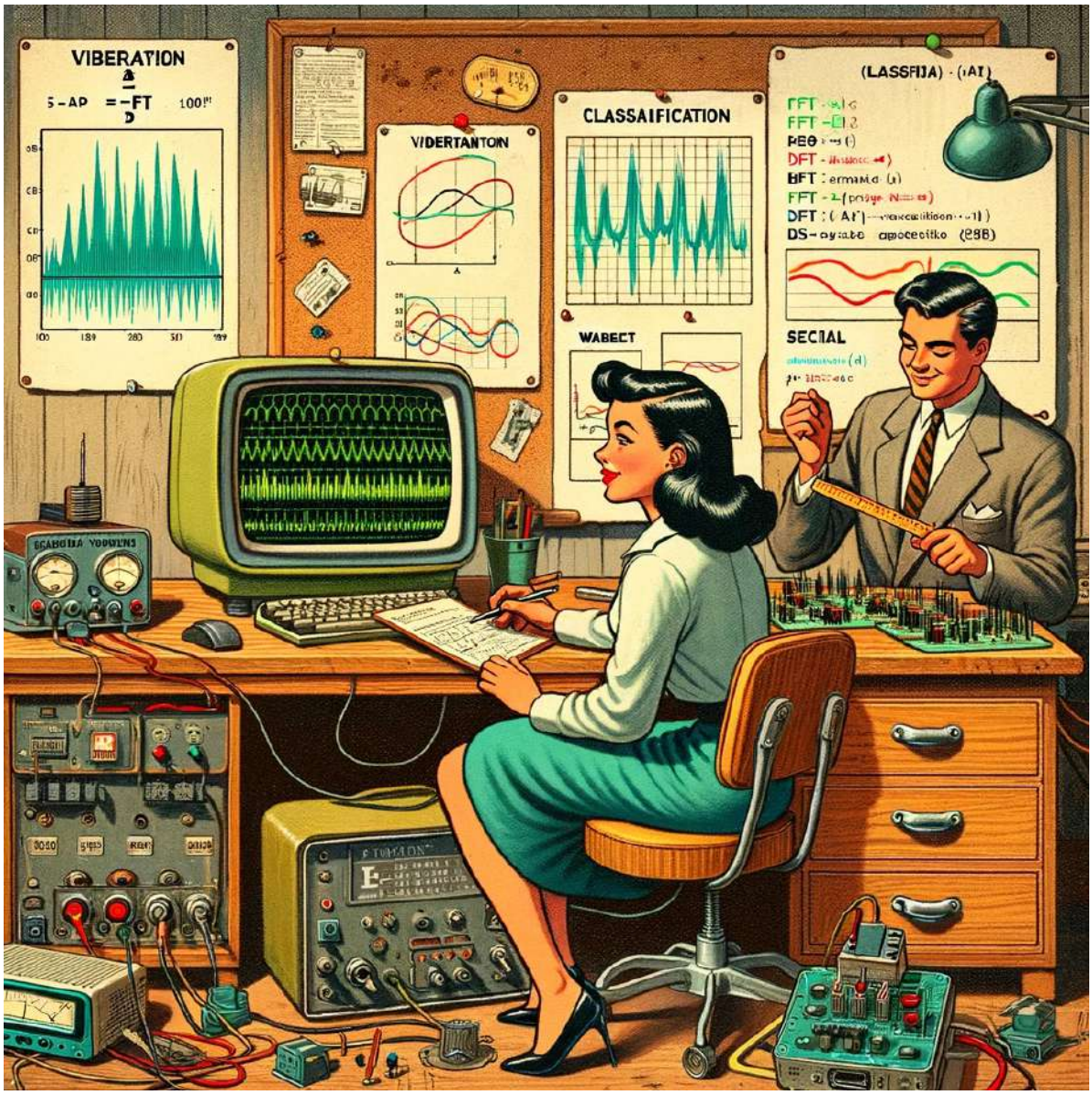


Figure 6: *DALL · E 3 Prompt: 1950s style cartoon illustration of a Latin male and female scientist in a vibration research room. The man is using a calculus ruler to examine ancient circuitry. The woman is at a computer with complex vibration graphs. The wooden table has boards with sensors, prominently an accelerometer. A classic, rounded-back computer shows the Arduino IDE with code for LED pin assignments and machine learning algorithms for movement detection. The Serial Monitor displays FFT, classification, wavelets, and DSPs. Vintage lamps, tools, and charts with FFT and Wavelets graphs complete the scene.*

Overview

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

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

Extracting Features Review

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

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

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

> The Studio does not perform normalization or standardization, so sometimes, when working with Sensor Fusion, it could be necessary to perform this step before uploading data to the Studio. This is particularly crucial in sensor fusion projects, as seen in this tutorial, Sensor Data Fusion with Spresense and CommonSense.

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

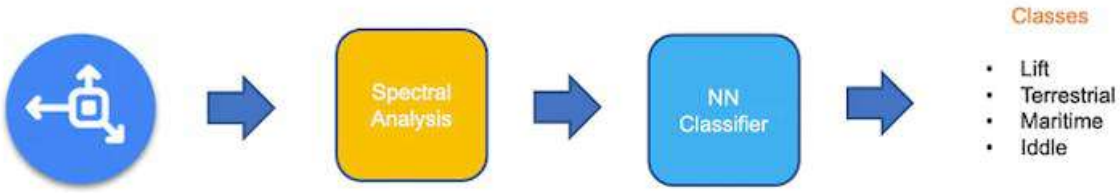
Feature extraction: Once the data is preprocessed and segmented, you can extract features that describe the motion's characteristics. Some typical features extracted from accelerometer data include:

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

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

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

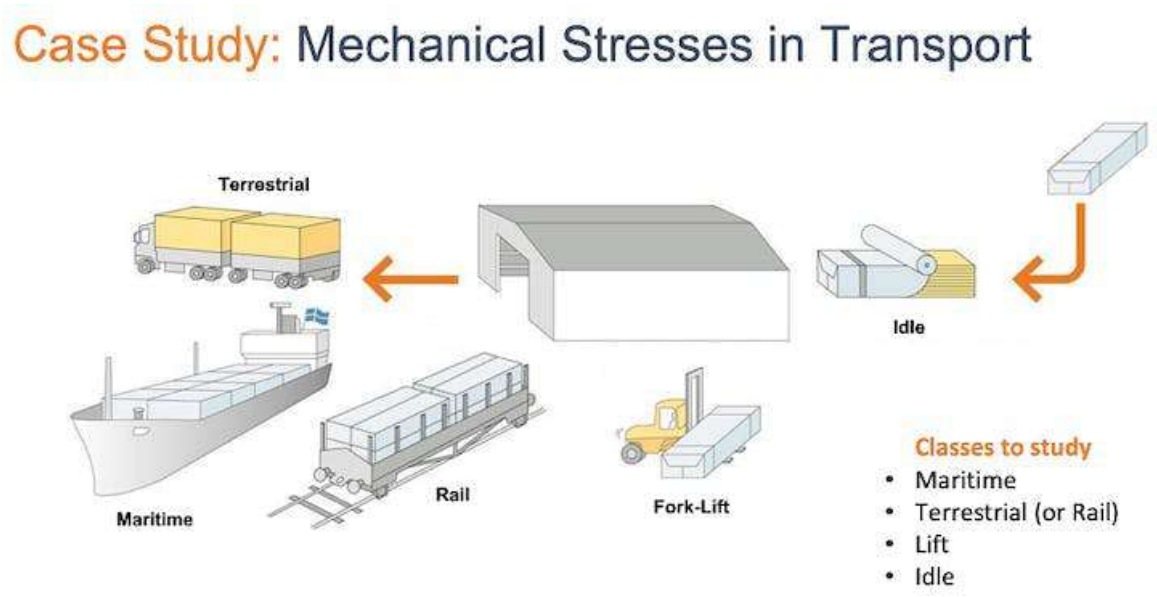
A TinyML Motion Classification project



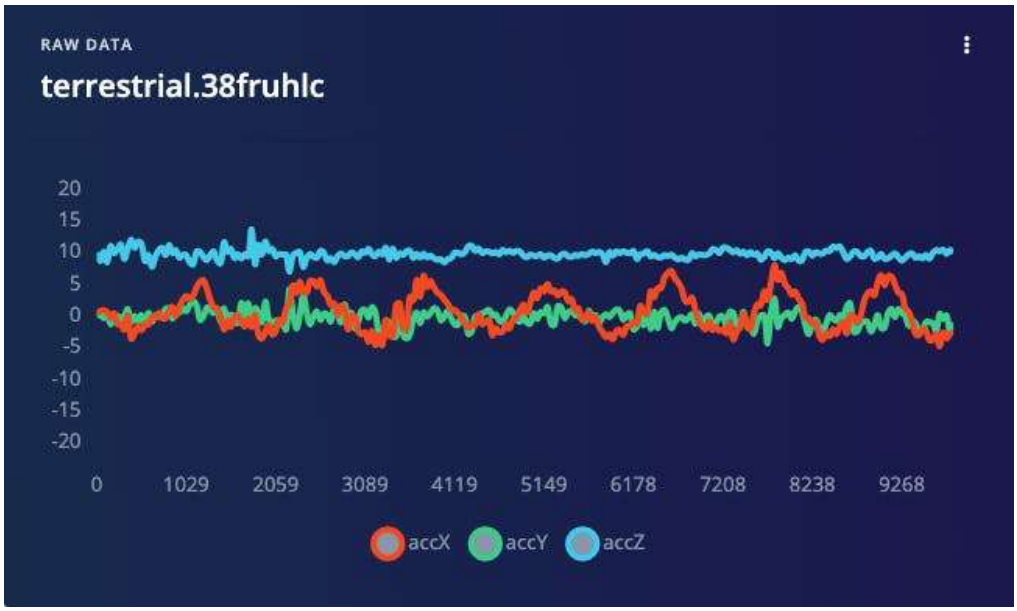
In the hands-on project, *Motion Classification and Anomaly Detection*, we simulated mechanical stresses in transport, where our problem was to classify four classes of movement:

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

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



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

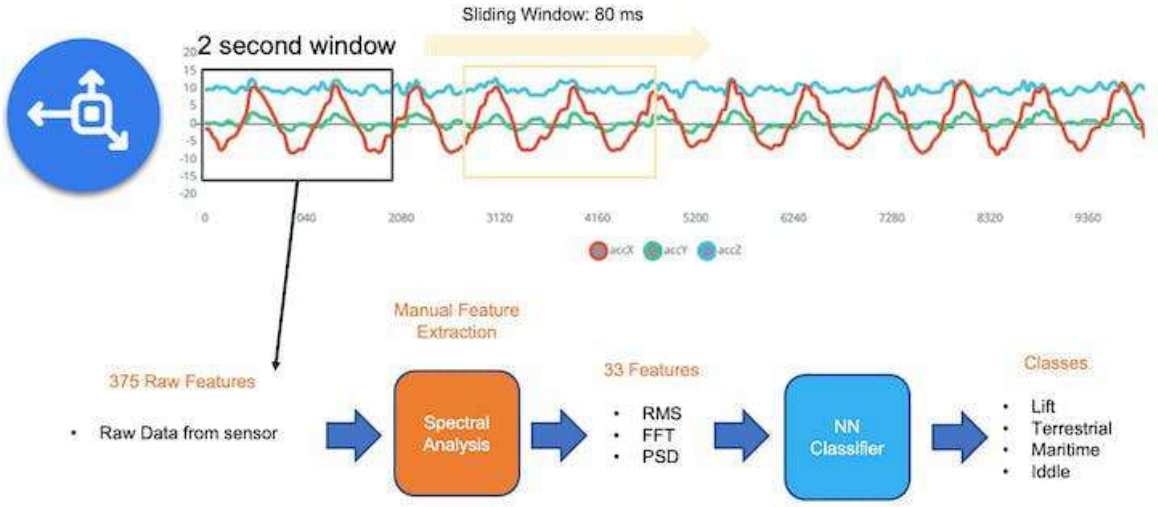


The result is similar when this analysis is done over another dataset with the same principle, using a different sampling frequency, 62.5 Hz instead of 50 Hz.

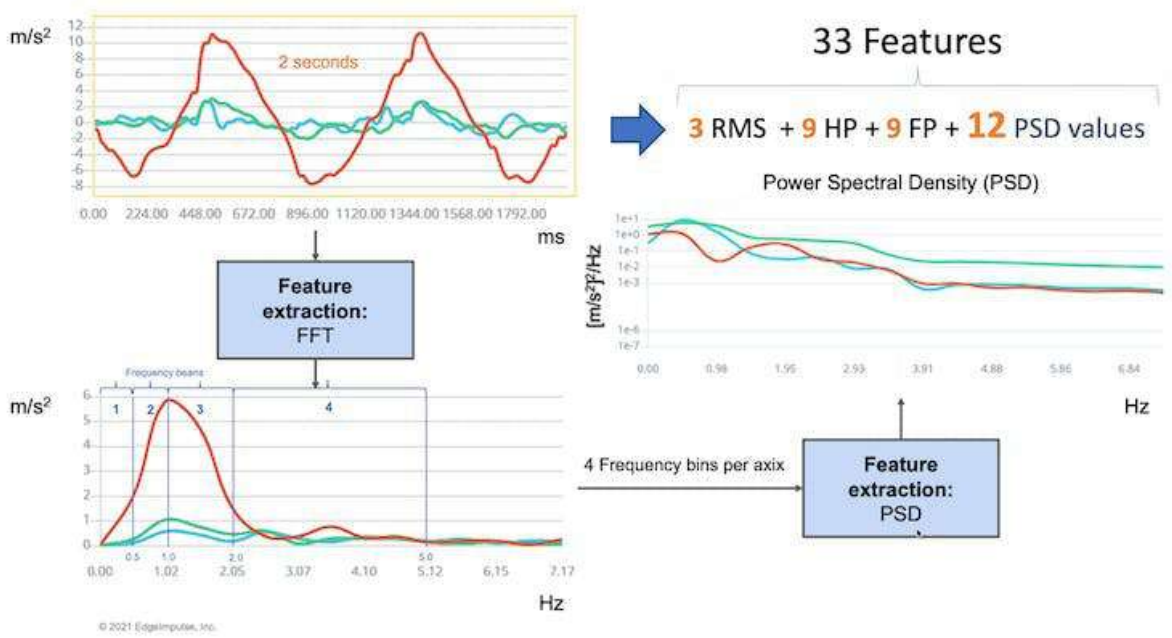
Data Pre-Processing

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

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



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



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

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

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

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

- RMS
- Skewness
- Kurtosis

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

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

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

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

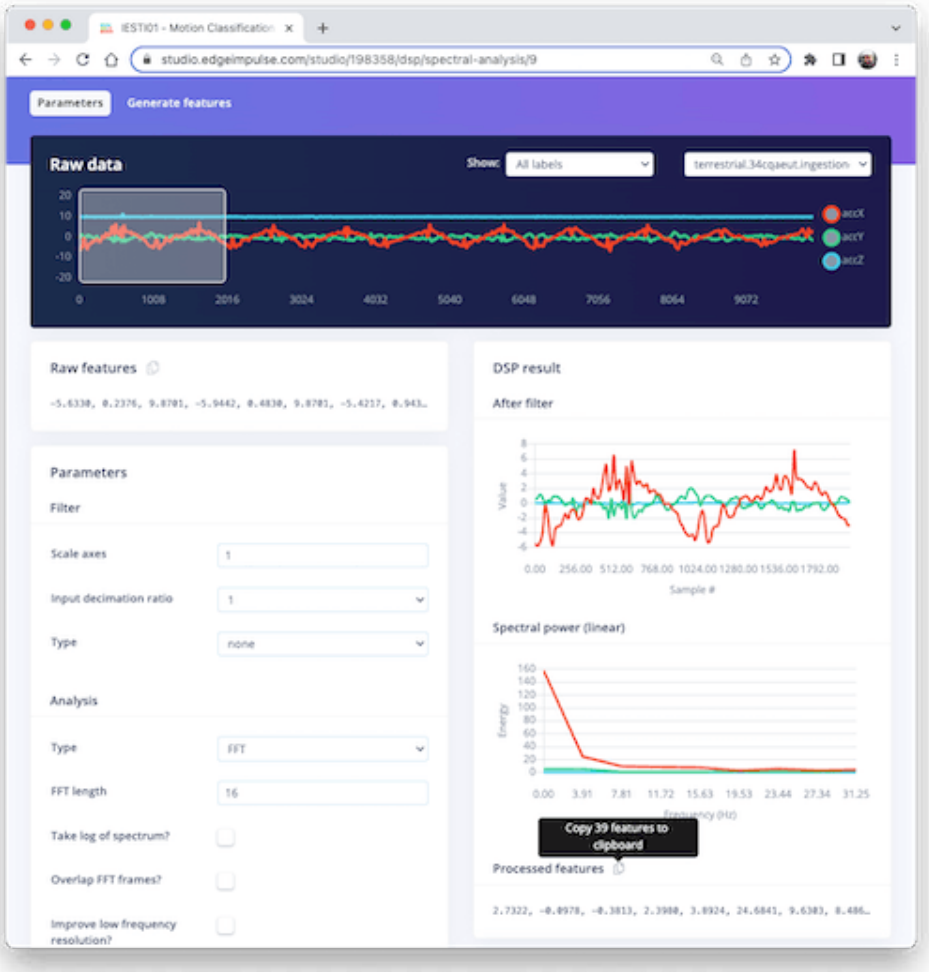
Start importing the libraries:

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

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

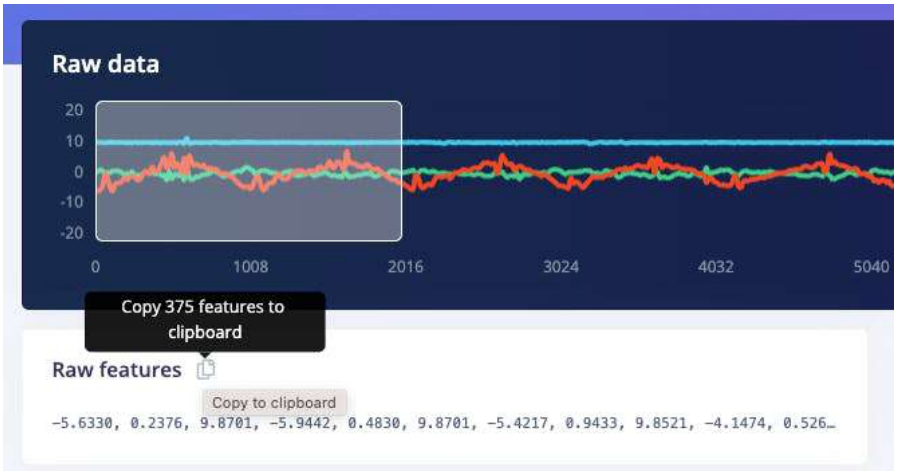
From the studied project, let's choose a data sample from accelerometers as below:

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

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



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



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

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

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

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

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

```
features = [
    2.7322, -0.0978, -0.3813,
```

```
    2.3980, 3.8924, 24.6841,
    9.6303, ...
]
N_feat = len(features)
N_feat_axis = int(N_feat/n_sensors)
```

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

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

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

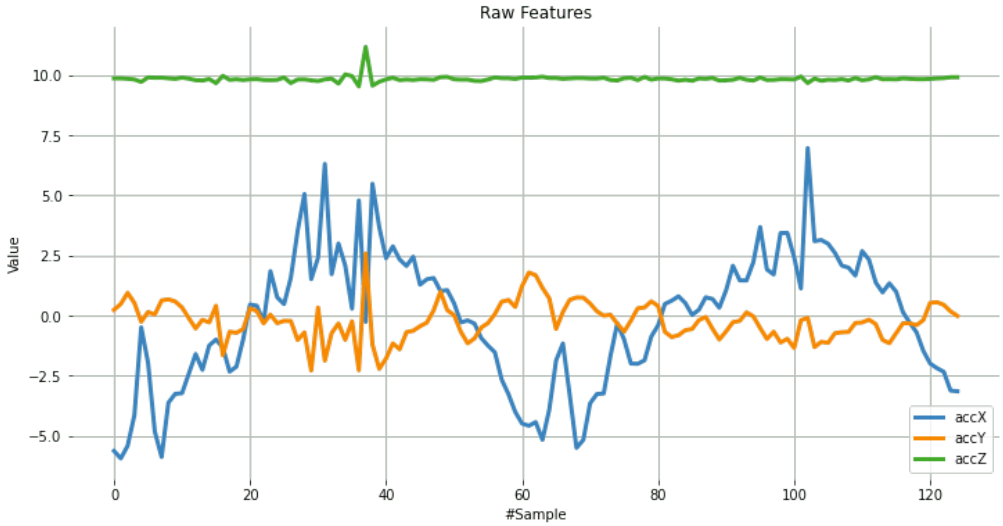
and 10 for the frequency domain (we will return to this later).

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

Splitting raw data per sensor

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

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

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



Subtracting the mean

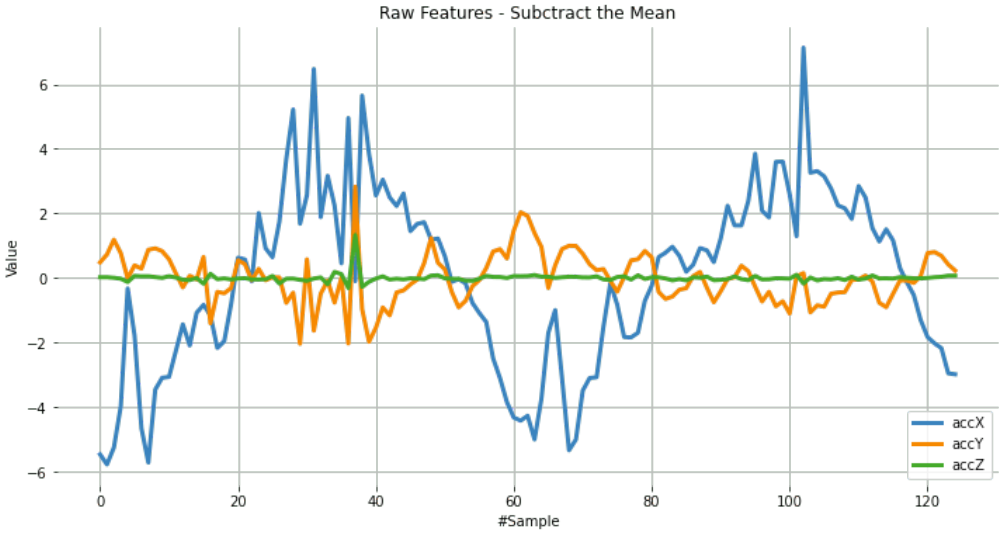
Next, we should subtract the mean from the *data*. Subtracting the mean from a data set is a common data pre-processing step in statistics and machine learning. The purpose of subtracting the mean from the data is to center the data around zero. This is important because it can reveal patterns and relationships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

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

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

```
dtmean = [
    (sum(x) / len(x))
    for x in sensors
]

[
    print('mean_' + x + ' =', round(y, 4))
```

```
    for x, y in zip(axis, dtmean)
][0]

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

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



Time Domain Statistical features

RMS Calculation

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

In the case of a set of n values $x_1, x_2, \ldots, x_n$, the RMS is:

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

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

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

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

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

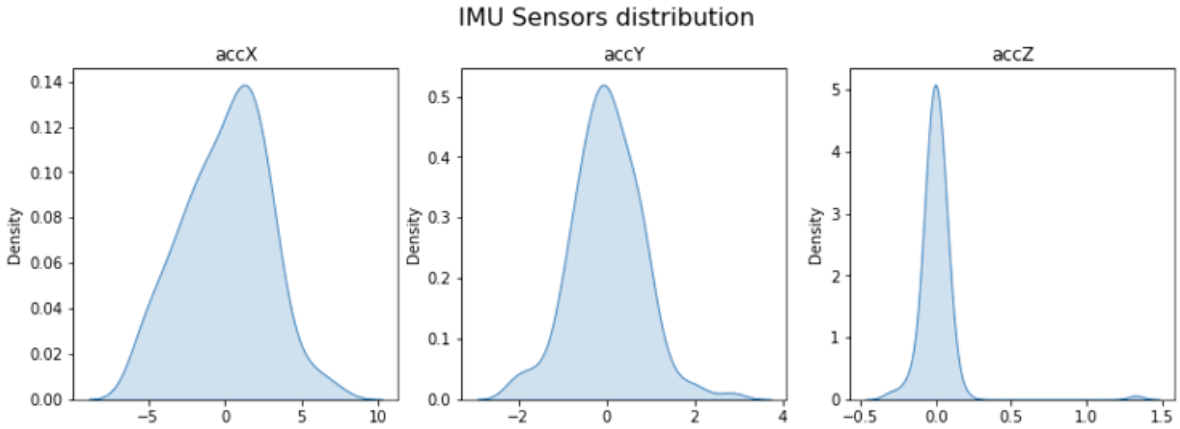
```
rms_accX=  2.7322

rms_accY=  0.7833

rms_accZ=  0.1383

Compared with Edge Impulse result features:

[2.7322, 0.7833, 0.1383]
```

Skewness and kurtosis calculation

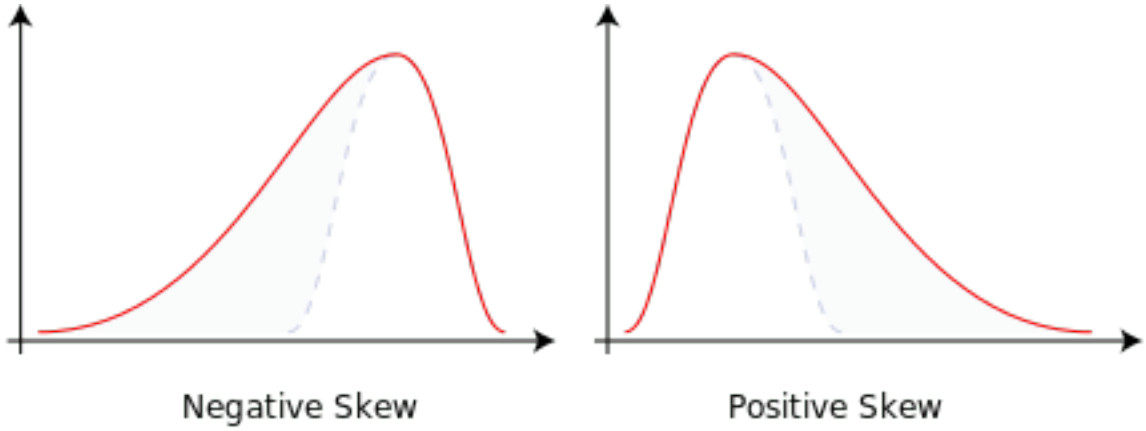
In statistics, skewness and kurtosis are two ways to measure the **shape of a distribution**.

Here, we can see the sensor values distribution:

```
fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(13, 4))
sns.kdeplot(accX, fill=True, ax=axes[0])
sns.kdeplot(accY, fill=True, ax=axes[1])
sns.kdeplot(accZ, fill=True, ax=axes[2])
axes[0].set_title('accX')
axes[1].set_title('accY')
axes[2].set_title('accZ')
plt.suptitle('IMU Sensors distribution', fontsize=16, y=1.02)
plt.show()
```



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



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

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

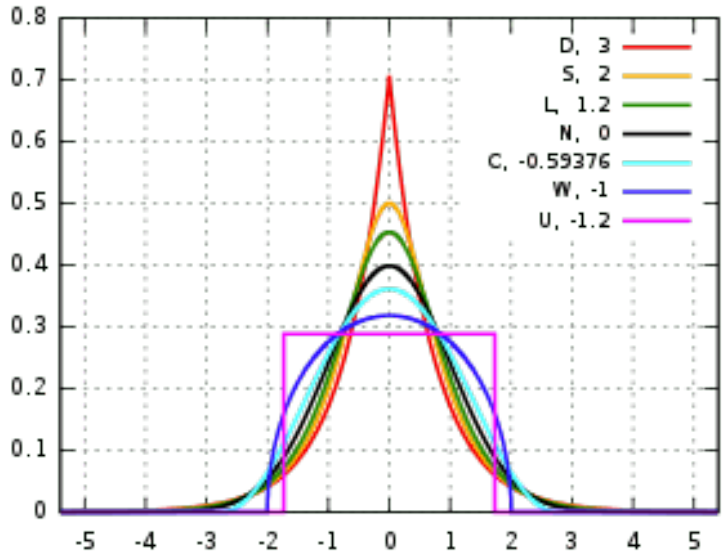
```
skew_accX=  -0.099
```

```
skew_accY=  0.1756

skew_accZ=  6.9463
```

Compared with Edge Impulse result features:

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

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



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

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

```
kurt_accX=  -0.3475

kurt_accY=  1.2673

kurt_accZ=  68.1123
```

Compared with Edge Impulse result features:

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

Spectral features

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

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

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

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

Spectral Power - Welch's method

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

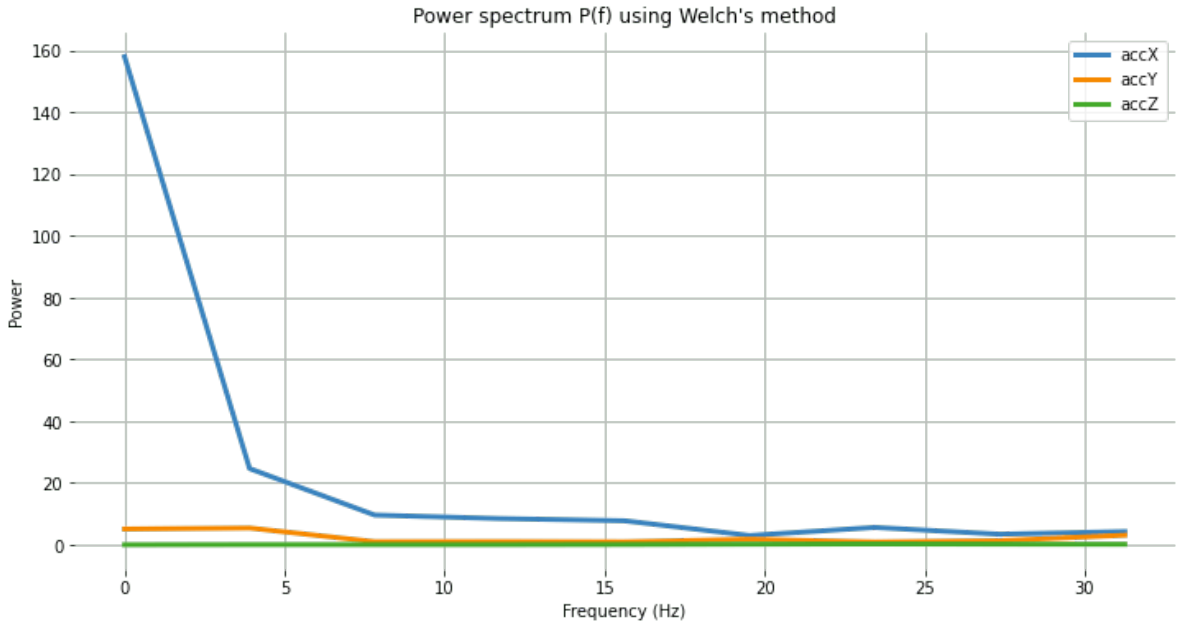
```
# Function used by Edge Impulse instead of scipy.signal.welch().
def welch_max_hold(fx, sampling_freq, nfft, n_overlap):
    n_overlap = int(n_overlap)
    spec_powers = [0 for _ in range(nfft//2+1)]
    ix = 0
    while ix <= len(fx):
        # Slicing truncates if end_idx > len,
        # and rfft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

Applying the above function to 3 signals:

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

We can plot the Power Spectrum P(f):

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



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

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

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

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

EI Processed Spectral features (accX):

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

Calculated features:

2.9069 8.5569 24.6844 9.6304 8.4865 7.7794 2.9964 5.6242 3.4198 4.2736

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

EI Processed Spectral features (accY):

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

Calculated features:

1.1426 -0.3886 5.4289 0.999 1.0315 0.9458 1.8116 0.9088 1.3301 3.1121

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

EI Processed Spectral features (accZ):

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

Calculated features:

0.3781 -1.4874 0.0606 0.057 0.0567 0.0976 0.194 0.2574 0.2083 0.166

Time-frequency domain

Wavelets

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

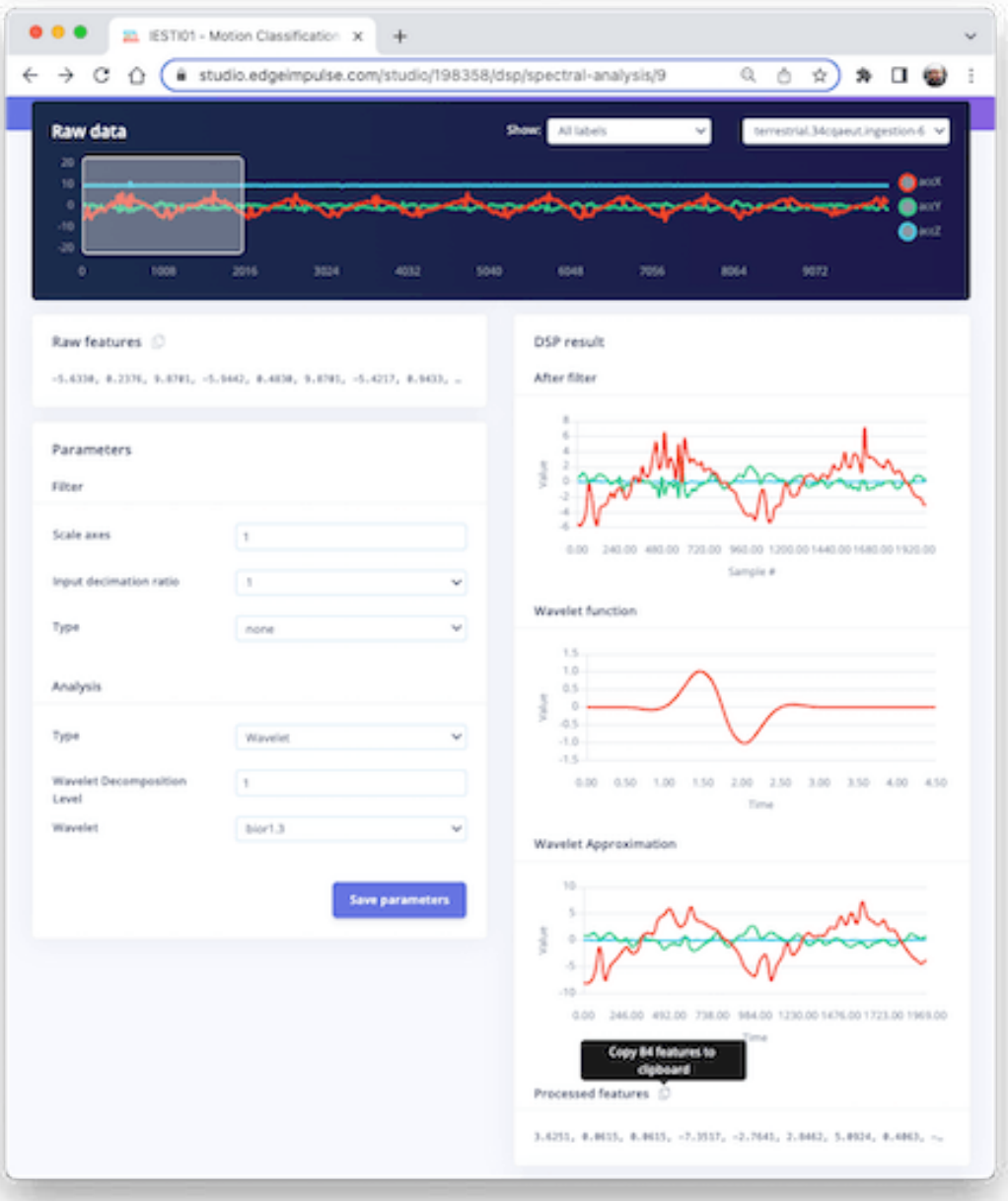
Wavelet transforms work by breaking down a signal into different frequency components and analyzing them individually. The transformation is achieved by convolving the signal with a **wavelet function**, a small waveform centered at a specific time and frequency. This process effectively decomposes the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

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

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

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

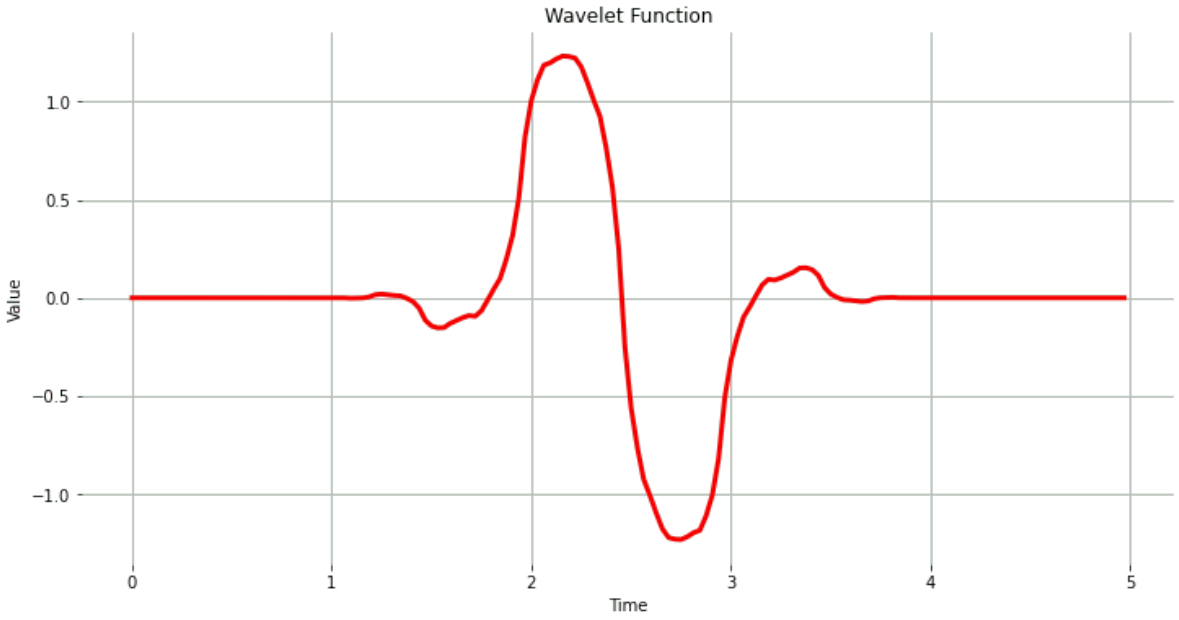
- Type: Wavelet
- Wavelet Decomposition Level: 1
- Wavelet: bior1.3



The Wavelet Function

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

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



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


```
features = [
    3.6251, 0.0615, 0.0615,
```

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

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

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

- [11] Statiscal Features: **n5, n25, n75, n95, mean, median,** standard deviation **(std)**, variance **(var)** root mean square **(rms), kurtosis**, and skewness **(skew)**.
- [2] Crossing Features: Zero crossing rate **(zcross)** and mean crossing rate **(mcross)** are the times that the signal passes through the baseline ($y = 0$) and the average level (y = u) per unit of time, respectively
- [1] Complexity Feature: **Entropy** is a characteristic measure of the complexity of the signal

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

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

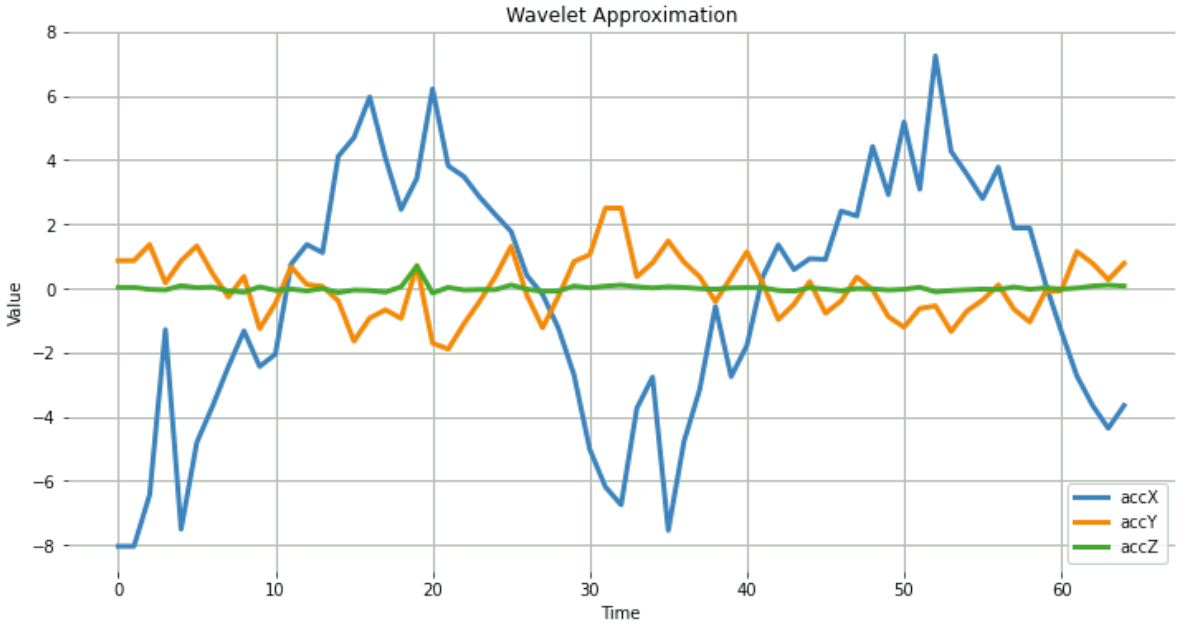
Wavelet Analysis

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

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

```
(accX_l1, accX_d1) = pywt.dwt(accX, wavelet_name)
(accY_l1, accY_d1) = pywt.dwt(accY, wavelet_name)
```

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

# Plot power spectrum versus frequency
plt.plot(accX_l1, label='accX')
plt.plot(accY_l1, label='accY')
plt.plot(accZ_l1, label='accZ')
plt.legend(loc='lower right')
plt.xlabel('Time')
plt.ylabel('Value')
plt.title('Wavelet Approximation')
plt.grid()
plt.box(False)
plt.show()
```



Feature Extraction

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

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

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

The Skelness and Kurtosis:

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

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

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

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


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

```
    return mcross

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

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

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

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

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

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

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

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

```
all_feat_l0 = []
for i in range(len(axis)):
    feat_l0 = (
        stat_feat_l0[i]
        + [skew_l0[i]]
        + [kurtosis_l0[i]]
        + [cross_l0[0][i]]
        + [cross_l0[1][i]]
        + [entropy_l0[i]]
    )
    [print(axis[i] + ' +x+= ', round(y, 4))
       for x, y in zip(L0_features_names, feat_l0)][0]
    all_feat_l0.append(feat_l0)

all_feat_l0 = [
    item
    for sublist in all_feat_l0
    for item in sublist
]
print(f"\nAll L0 Features = {len(all_feat_l0)}")

all_feat_l1 = []
for i in range(len(axis)):
    feat_l1 = (
        stat_feat_l1[i]
        + [skew_l1[i]]
        + [kurtosis_l1[i]]
        + [cross_l1[0][i]]
        + [cross_l1[1][i]]
        + [entropy_l1[i]]
    )
    [print(axis[i]+' '+x+'= ', round(y, 4))
       for x,y in zip(L1_features_names, feat_l1)][0]
    all_feat_l1.append(feat_l1)

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

```
accX L0-n5=  -4.9364          accX L1-n5=  -7.3516
accX L0-n25=  -1.8429         accX L1-n25=  -2.7641
accX L0-n75=  1.8842          accX L1-n75=  2.8462
accX L0-n95=  3.8096          accX L1-n95=  5.0924
accX L0-median=  0.4058       accX L1-median=  0.4064
accX L0-mean=  -0.0           accX L1-mean=  -0.2133
accX L0-std=  2.7322          accX L1-std=  3.8473
accX L0-var=  7.4651          accX L1-var=  14.8015
accX L0-rms=  2.7322          accX L1-rms=  3.8532
accX L0-skew=  -0.099         accX L1-skew=  -0.2975
accX L0-Kurtosis=  -0.3475    accX L1-Kurtosis=  -0.7631
accX L0-zcross=  0.06         accX L1-zcross=  0.06
accX L0-mcross=  0.06         accX L1-mcross=  0.06
accX L0-entropy=  4.8283      accX L1-entropy=  4.1744
accY L0-n5=  -1.149           accY L1-n5=  -1.3234
accY L0-n25=  -0.4475         accY L1-n25=  -0.6492
accY L0-n75=  0.4814          accY L1-n75=  0.7844
accY L0-n95=  1.1491          accY L1-n95=  1.361
accY L0-median=  -0.0315      accY L1-median=  0.0659
accY L0-mean=  0.0            accY L1-mean=  0.0276
accY L0-std=  0.7833          accY L1-std=  0.9345
accY L0-var=  0.6136          accY L1-var=  0.8732
accY L0-rms=  0.7833          accY L1-rms=  0.9349
accY L0-skew=  0.1756         accY L1-skew=  0.2874
accY L0-Kurtosis=  1.2673     accY L1-Kurtosis=  0.0347
accY L0-zcross=  0.29         accY L1-zcross=  0.31
accY L0-mcross=  0.29         accY L1-mcross=  0.31
accY L0-entropy=  4.8283      accY L1-entropy=  4.1317
accZ L0-n5=  -0.1242          accZ L1-n5=  -0.1126
accZ L0-n25=  -0.0429         accZ L1-n25=  -0.0493
accZ L0-n75=  0.0349          accZ L1-n75=  0.0348
accZ L0-n95=  0.0839          accZ L1-n95=  0.1022
accZ L0-median=  -0.0112      accZ L1-median=  -0.0137
accZ L0-mean=  0.0            accZ L1-mean=  0.0025
accZ L0-std=  0.1383          accZ L1-std=  0.1053
accZ L0-var=  0.0191          accZ L1-var=  0.0111
accZ L0-rms=  0.1383          accZ L1-rms=  0.1053
accZ L0-skew=  6.9463         accZ L1-skew=  4.4095
accZ L0-Kurtosis=  68.1123    accZ L1-Kurtosis=  28.6586
accZ L0-zcross=  0.35         accZ L1-zcross=  0.4
accZ L0-mcross=  0.35         accZ L1-mcross=  0.37
accZ L0-entropy=  4.5649      accZ L1-entropy=  4.1531

All L0 Features = 42          All L1 Features = 42
```

Conclusion

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

Daniel Situnayake wrote in his blog: "Raw sensor data is highly dimensional and noisy. Digital signal processing algorithms help us sift the signal from the noise. DSP is an essential part of embedded engineering, and many edge processors have on-board acceleration for DSP. As an ML engineer, learning basic DSP gives you superpowers for handling high-frequency time series data in your models." I recommend you read Dan's excellent post in its totality: nn to cpp: What you need to know about porting deep learning models to the edge.

Motion Classification and Anomaly Detection

Figure 7: *DALL · E 3 Prompt: 1950s style cartoon illustration depicting a movement research room. In the center of the room, there's a simulated container used for transporting goods on trucks, boats, and forklifts. The container is detailed with rivets and markings typical of industrial cargo boxes. Around the container, the room is filled with vintage equipment, including an oscilloscope, various sensor arrays, and large paper rolls of recorded data. The walls are adorned with educational posters about transportation safety and logistics. The overall ambiance of the room is nostalgic and scientific, with a hint of industrial flair.*

Overview

Transportation is the backbone of global commerce. Millions of containers are transported daily via various means, such as ships, trucks, and trains, to destinations worldwide. Ensuring these containers' safe and efficient transit is a monumental task that requires leveraging modern technology, and TinyML is undoubtedly one of them.

In this hands-on tutorial, we will work to solve real-world problems related to transportation. We will develop a Motion Classification and Anomaly Detection system using the Arduino Nicla Vision board, the Arduino IDE, and the Edge Impulse Studio. This project will help us understand how containers experience different forces and motions during various phases of transportation, such as terrestrial and maritime transit, vertical movement via forklifts, and stationary periods in warehouses.

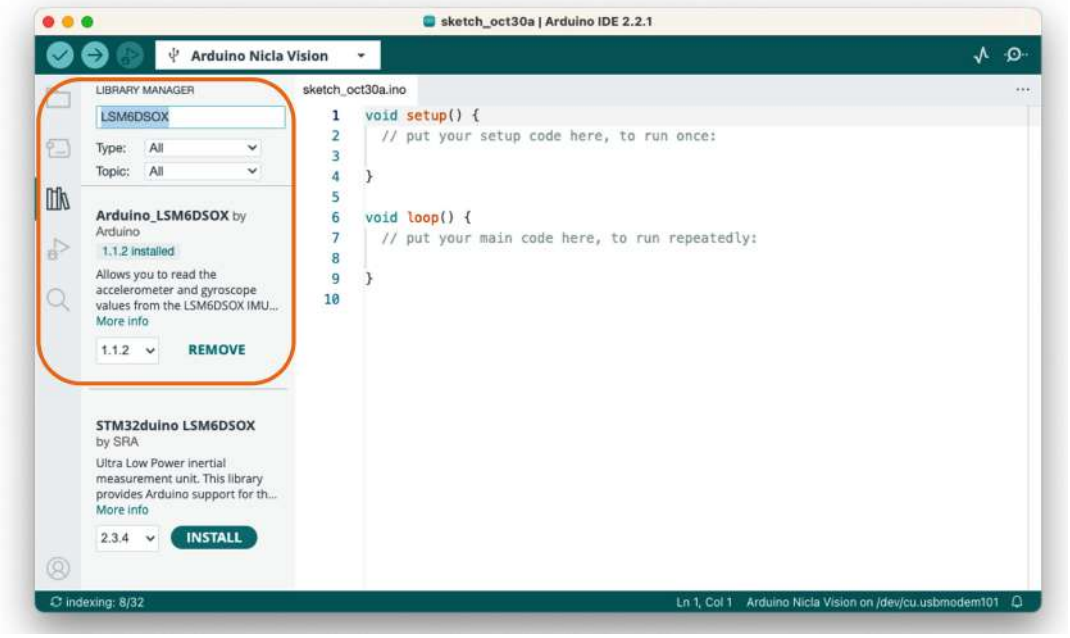
Learning Objectives

- Setting up the Arduino Nicla Vision Board
- Data Collection and Preprocessing
- Building the Motion Classification Model
- Implementing Anomaly Detection
- Real-world Testing and Analysis

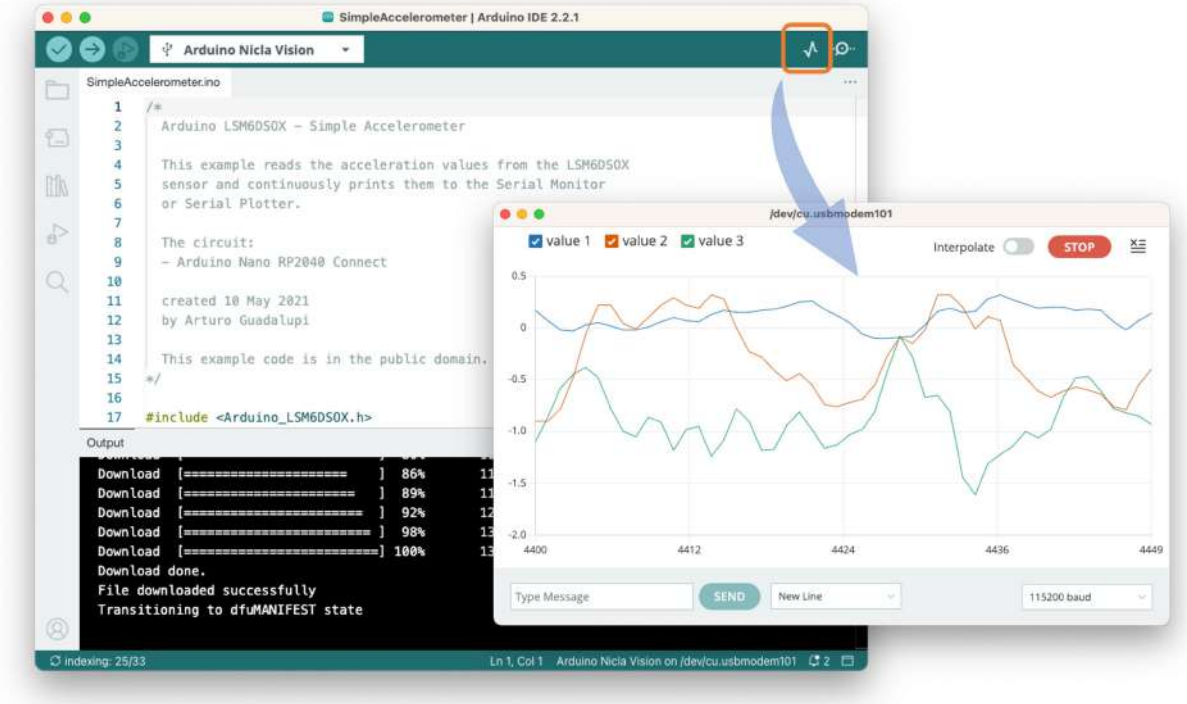
By the end of this tutorial, you'll have a working prototype that can classify different types of motion and detect anomalies during the transportation of containers. This knowledge can be a stepping stone to more advanced projects in the burgeoning field of TinyML involving vibration.

IMU Installation and testing

For this project, we will use an accelerometer. As discussed in the Hands-On Tutorial, *Setup Nicla Vision*, the Nicla Vision Board has an onboard **6-axis IMU**: 3D gyroscope and 3D accelerometer, the LSM6DSOX. Let's verify if the LSM6DSOX IMU library is installed. If not, install it.



Next, go to `Examples > Arduino_LSM6DSOX > SimpleAccelerometer` and run the accelerometer test. You can check if it works by opening the IDE Serial Monitor or Plotter. The values are in g (earth gravity), with a default range of +/- 4g:



Defining the Sampling frequency:

Choosing an appropriate sampling frequency is crucial for capturing the motion characteristics you're interested in studying. The Nyquist-Shannon sampling theorem states that the sampling rate should be at least twice the highest frequency component in the signal to reconstruct it properly. In the context of motion classification and anomaly detection for transportation, the choice of sampling frequency would depend on several factors:

1. **Nature of the Motion**: Different types of transportation (terrestrial, maritime, etc.) may involve different ranges of motion frequencies. Faster movements may require higher sampling frequencies.
2. **Hardware Limitations**: The Arduino Nicla Vision board and any associated sensors may have limitations on how fast they can sample data.
3. **Computational Resources**: Higher sampling rates will generate more data, which might be computationally intensive, especially critical in a TinyML environment.
4. **Battery Life**: A higher sampling rate will consume more power. If the system is battery-operated, this is an important consideration.
5. **Data Storage**: More frequent sampling will require more storage space, another crucial consideration for embedded systems with limited memory.

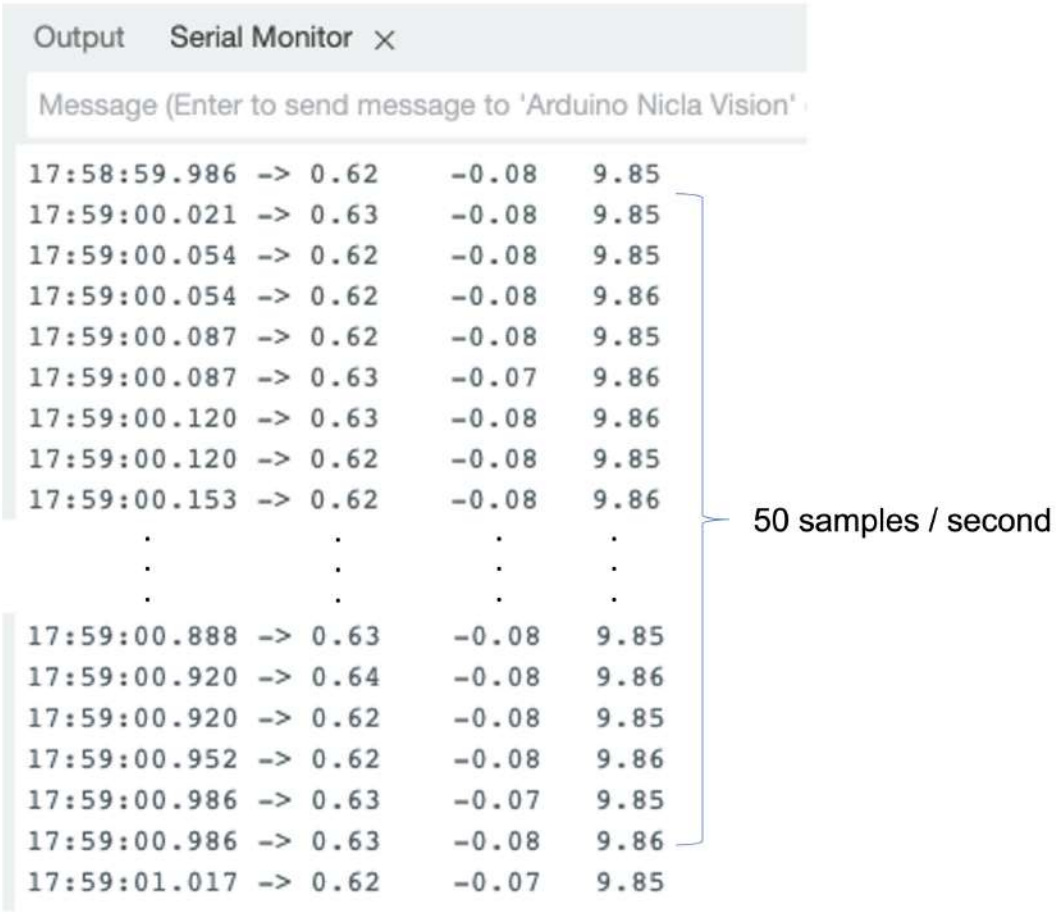
In many human activity recognition tasks, **sampling rates of around 50 Hz to 100 Hz** are commonly used. Given that we are simulating transportation scenarios, which are generally not high-frequency events, a sampling rate in that range (50-100 Hz) might be a reasonable starting point.

Let's define a sketch that will allow us to capture our data with a defined sampling frequency (for example, 50 Hz):

```
/*
 * Based on Edge Impulse Data Forwarder Example (Arduino)
  - https://docs.edgeimpulse.com/docs/cli-data-forwarder
 * Developed by M.Rovai @11May23
 */

/* Include ----------------------------------------- */
#include <Arduino_LSM6DSOX.h>

/* Constant defines -------------------------------- */
#define CONVERT_G_TO_MS2 9.80665f
#define FREQUENCY_HZ        50
#define INTERVAL_MS         (1000 / (FREQUENCY_HZ + 1))
```

```
static unsigned long last_interval_ms = 0;
float x, y, z;

void setup() {
  Serial.begin(9600);
  while (!Serial);

  if (!IMU.begin()) {
    Serial.println("Failed to initialize IMU!");
    while (1);
  }
}

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

    if (IMU.accelerationAvailable()) {
      // Read raw acceleration measurements from the device
      IMU.readAcceleration(x, y, z);

      // converting to m/s2
      float ax_m_s2 = x * CONVERT_G_TO_MS2;
      float ay_m_s2 = y * CONVERT_G_TO_MS2;
      float az_m_s2 = z * CONVERT_G_TO_MS2;

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

Uploading the sketch and inspecting the Serial Monitor, we can see that we are capturing 50 samples per second.

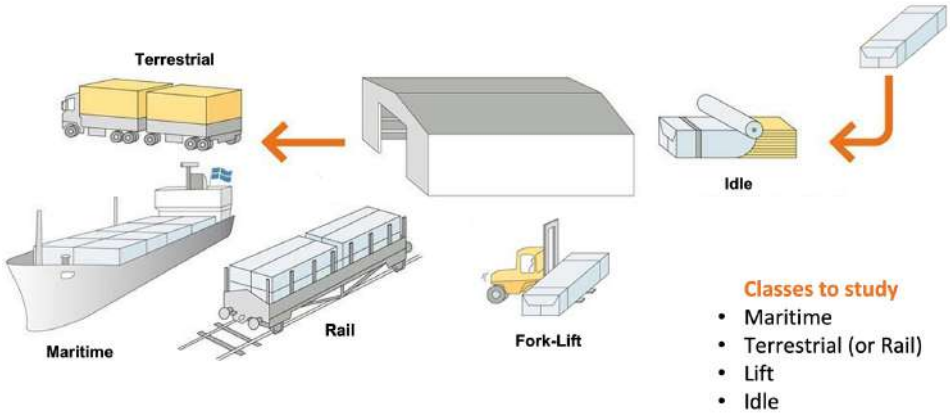


Note that with the Nicla board resting on a table (with the camera facing down), the z-axis measures around 9.8 m/s^2, the expected earth acceleration.

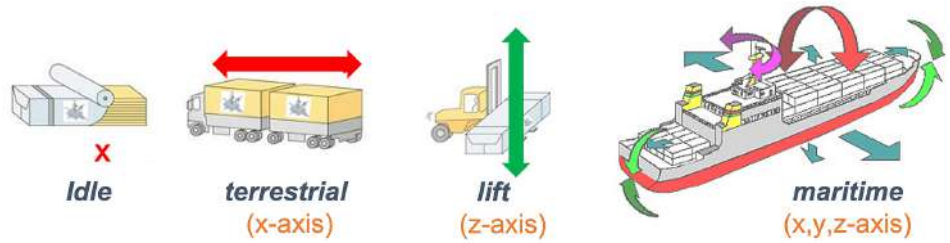
The Case Study: Simulated Container Transportation

We will simulate container (or better package) transportation through different scenarios to make this tutorial more relatable and practical. Using the built-in accelerometer of the Arduino Nicla Vision board, we'll capture motion data by manually simulating the conditions of:

1. **Terrestrial** Transportation (by road or train)
2. **Maritime**-associated Transportation
3. Vertical Movement via Fork-**Lift**
4. Stationary **(Idle**) period in a Warehouse



From the above images, we can define for our simulation that primarily horizontal movements (x or y axis) should be associated with the "Terrestrial class," Vertical movements (z-axis) with the "Lift Class," no activity with the "Idle class," and movement on all three axes to Maritime class.



Data Collection

For data collection, we can have several options. In a real case, we can have our device, for example, connected directly to one container, and the data collected on a file (for example .CSV) and stored on an SD card (Via SPI connection) or an offline repo in your computer. Data can also be sent remotely to a nearby repository, such as a mobile phone, using Bluetooth (as done in this project: Sensor DataLogger). Once your dataset is collected and stored as a .CSV file, it can be uploaded to the Studio using the CSV Wizard tool.

> In this video, you can learn alternative ways to send data to the Edge Impulse Studio.

Connecting the device to Edge Impulse

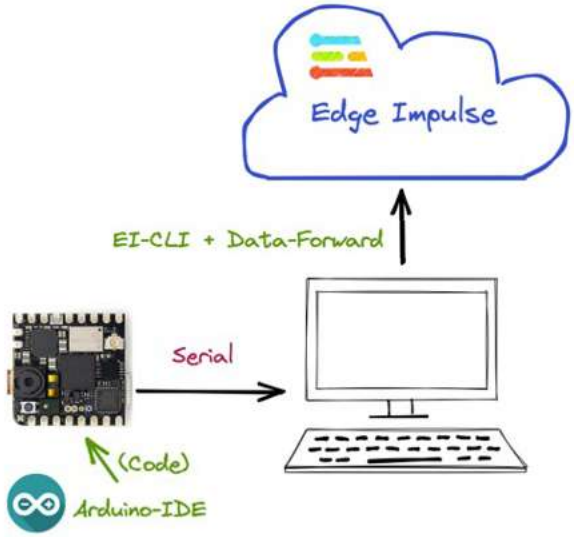
We will connect the Nicla directly to the Edge Impulse Studio, which will also be used for data pre-processing, model training, testing, and deployment. For that, you have two options:

1. Download the latest firmware and connect it directly to the `Data Collection` section.
2. Use the CLI Data Forwarder tool to capture sensor data from the sensor and send it to the Studio.

Option 1 is more straightforward, as we saw in the *Setup Nicla Vision* hands-on, but option 2 will give you more flexibility regarding capturing your data, such as sampling frequency definition. Let's do it with the last one.

Please create a new project on the Edge Impulse Studio (EIS) and connect the Nicla to it, following these steps:

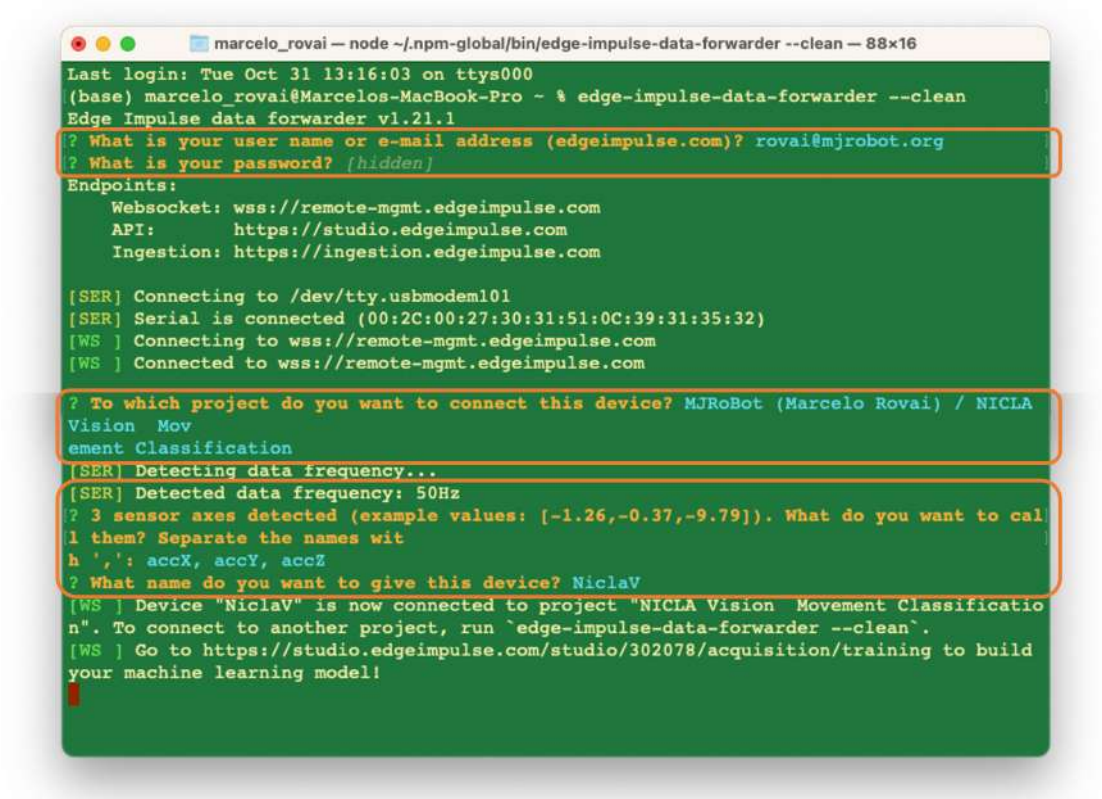
1. Install the Edge Impulse CLI and the Node.js into your computer.
2. Upload a sketch for data capture (the one discussed previously in this tutorial).
3. Use the CLI Data Forwarder to capture data from the Nicla's accelerometer and send it to the Studio, as shown in this diagram:



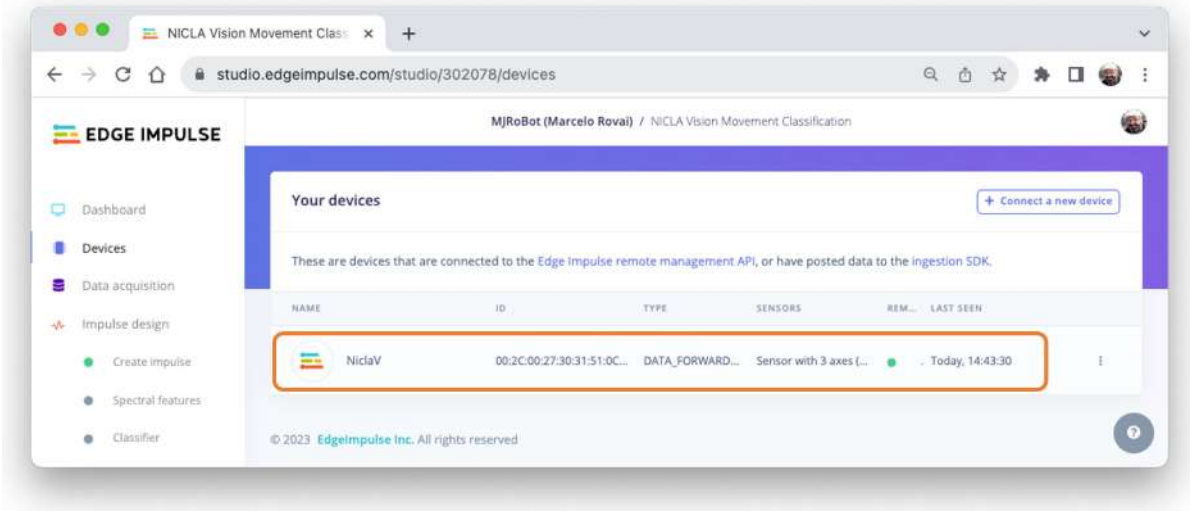
Start the CLI Data Forwarder on your terminal, entering (if it is the first time) the following command:

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

Next, enter your EI credentials and choose your project, variables (for example, *accX*, *accY*, and *accZ*), and device name (for example, *NiclaV*:



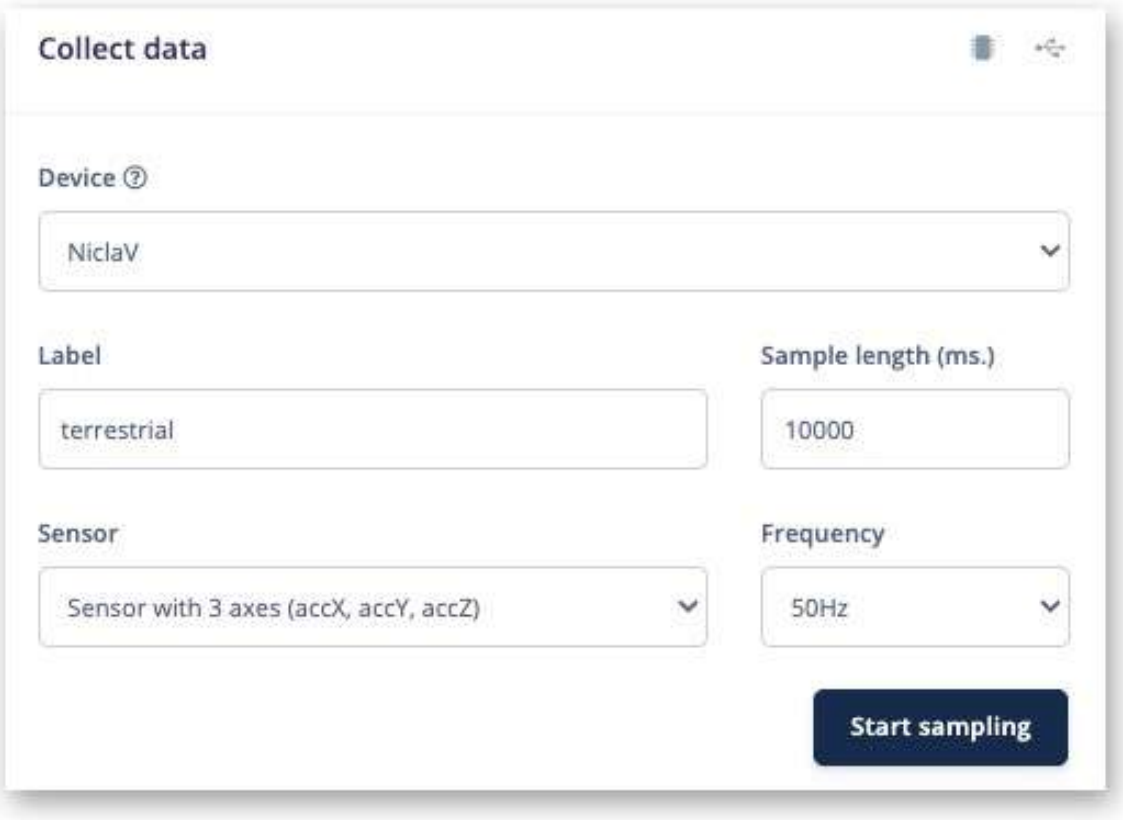
Go to the `Devices` section on your EI Project and verify if the device is connected (the dot should be green):



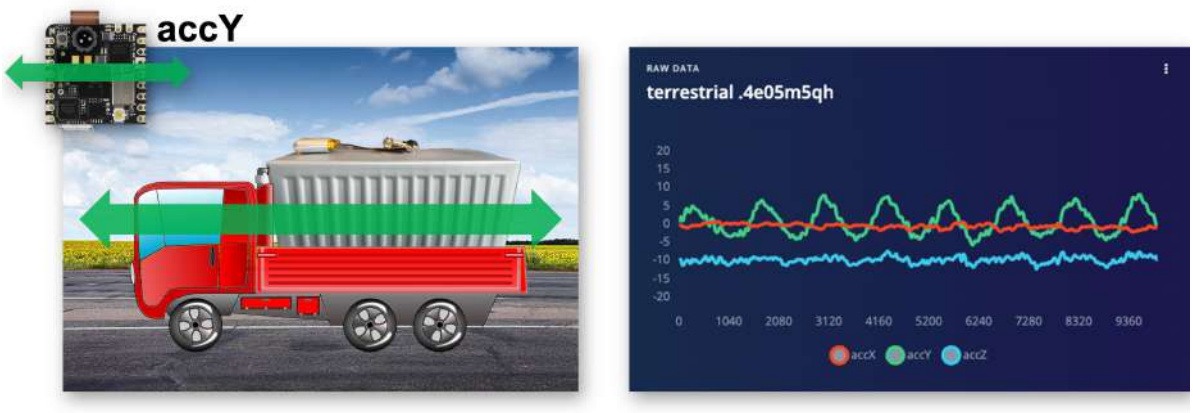
You can clone the project developed for this hands-on: NICLA Vision Movement Classification.

Data Collection

On the `Data Acquisition` section, you should see that your board `[NiclaV]` is connected. The sensor is available: `[sensor with 3 axes (accX, accY, accZ)]` with a sampling frequency of `[50 Hz]`. The Studio suggests a sample length of `[10000]` ms (10 s). The last thing left is defining the sample label. Let's start with `[terrestrial]`:



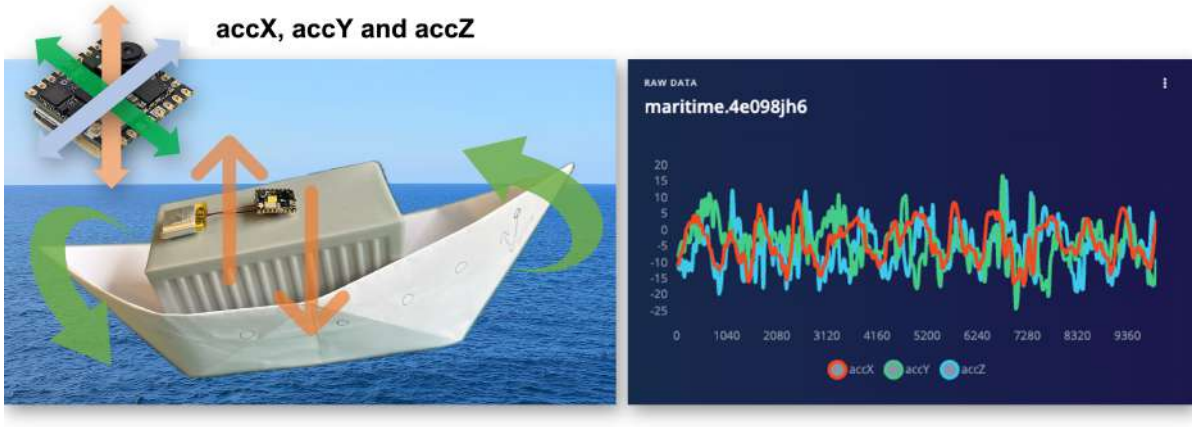
Terrestrial (palettes in a Truck or Train), moving horizontally. Press `[Start Sample]` and move your device horizontally, keeping one direction over your table. After 10 s, your data will be uploaded to the studio. Here is how the sample was collected:



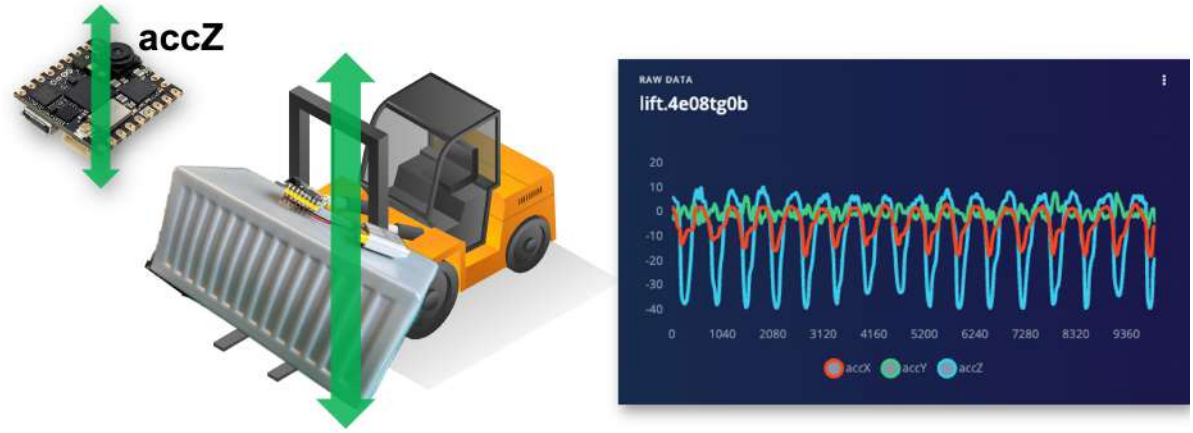
As expected, the movement was captured mainly in the Y-axis (green). In the blue, we see the Z axis, around -10 m/s^2 (the Nicla has the camera facing up).

As discussed before, we should capture data from all four Transportation Classes. So, imagine that you have a container with a built-in accelerometer facing the following situations:

Maritime (pallets in boats into an angry ocean). The movement is captured on all three axes:

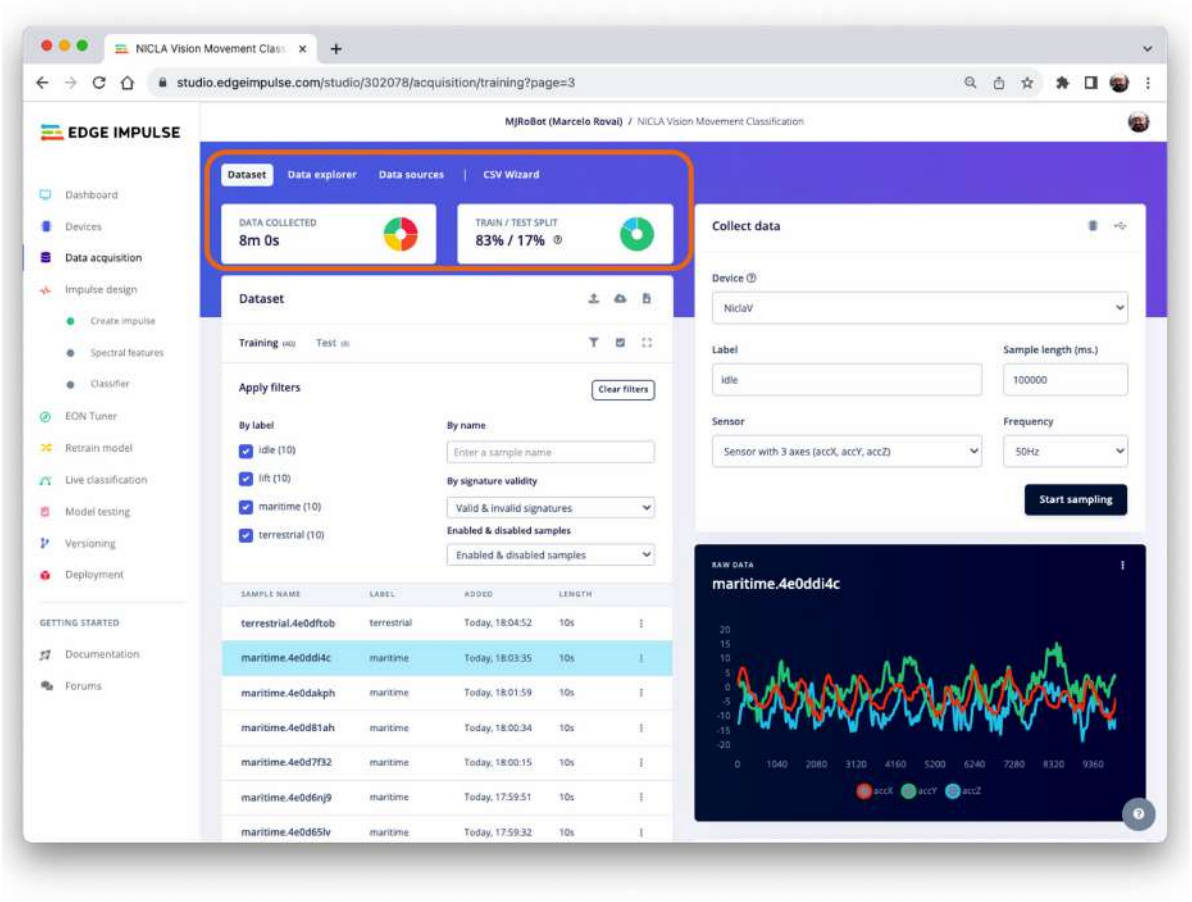


Lift (Palettes being handled vertically by a Forklift). Movement captured only in the Z-axis:

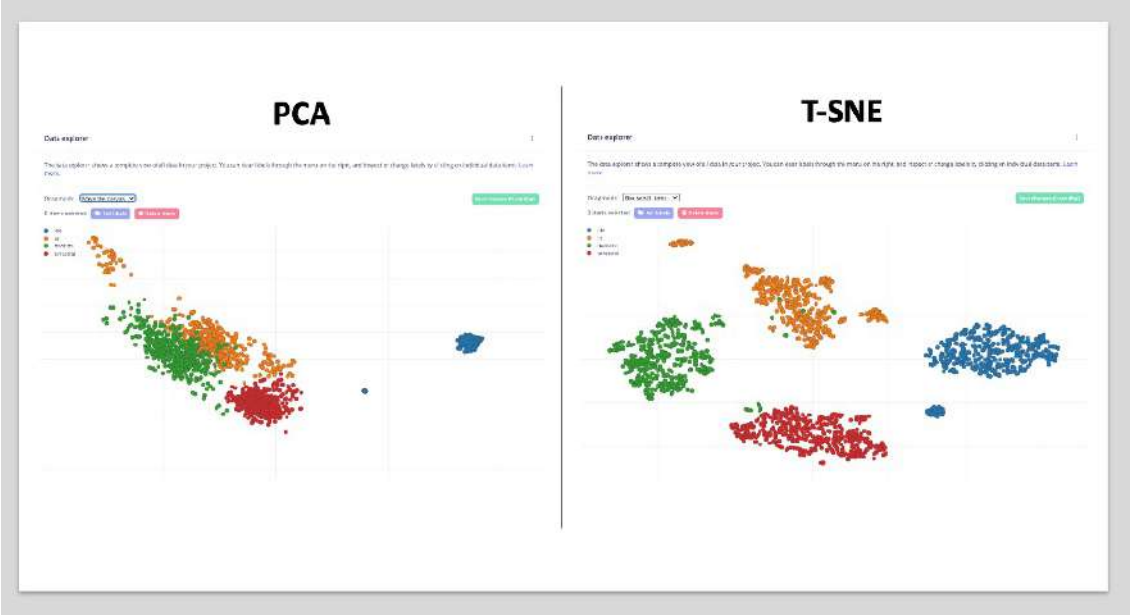


Idle (Paletts in a warehouse). No movement detected by the accelerometer:

You can capture, for example, 2 minutes (twelve samples of 10 seconds) for each of the four classes (a total of 8 minutes of data). Using the `three dots` menu after each one of the samples, select 2 of them, reserving them for the Test set. Alternatively, you can use the automatic `Train/Test Split tool` on the `Danger Zone` of `Dashboard` tab. Below, you can see the resulting dataset:



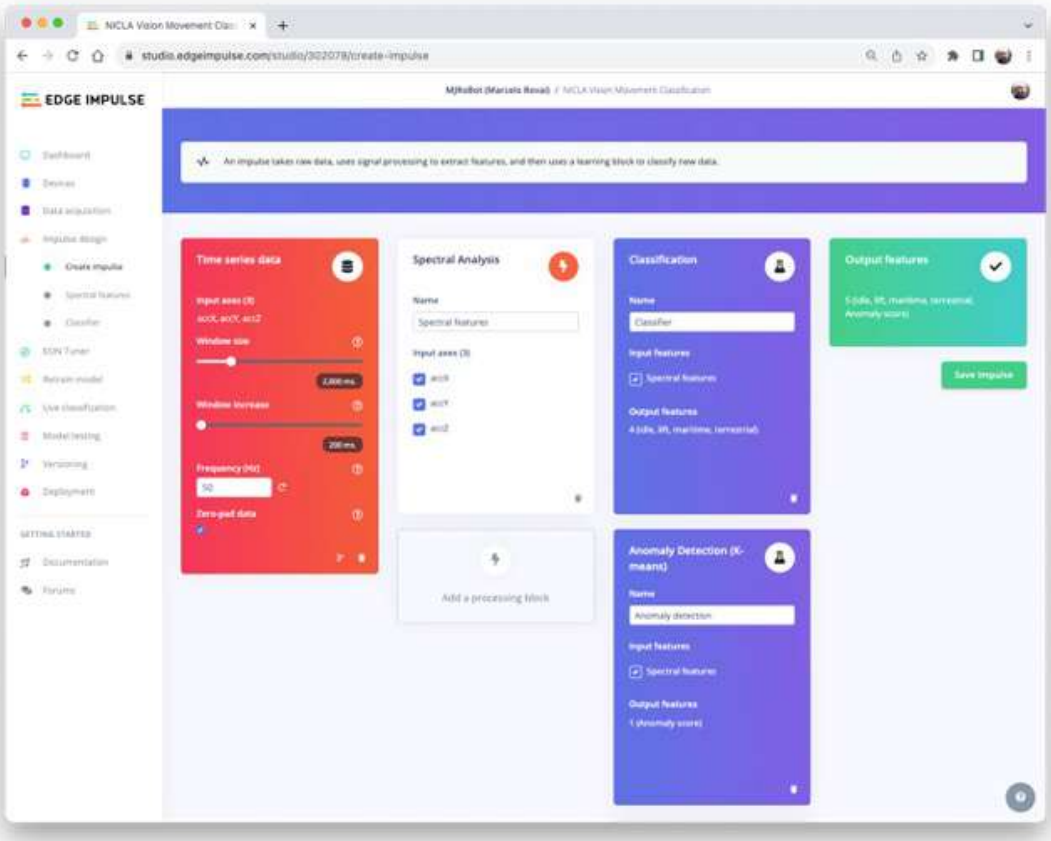
Once you have captured your dataset, you can explore it in more detail using the Data Explorer, a visual tool to find outliers or mislabeled data (helping to correct them). The data explorer first tries to extract meaningful features from your data (by applying signal processing and neural network embeddings) and then uses a dimensionality reduction algorithm such as PCA or t-SNE to map these features to a 2D space. This gives you a one-look overview of your complete dataset.



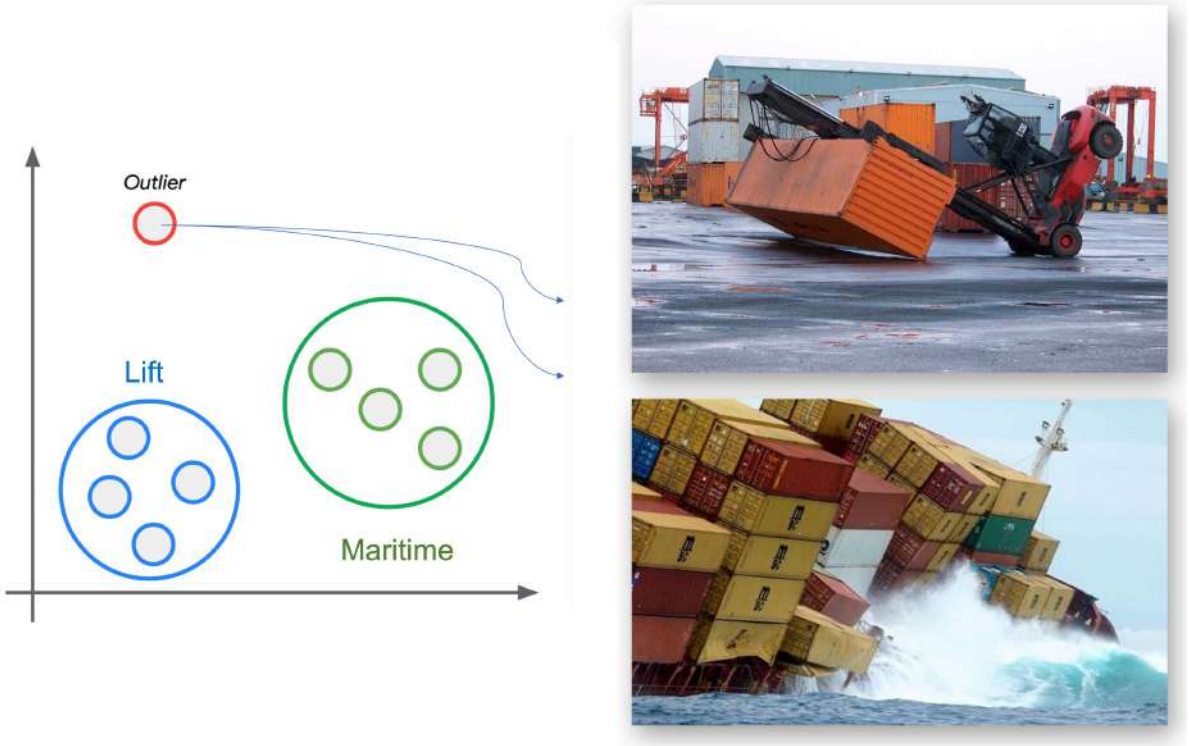
In our case, the dataset seems OK (good separation). But the PCA shows we can have issues between maritime (green) and lift (orange). This is expected, once on a boat, sometimes the movement can be only "vertical".

Impulse Design

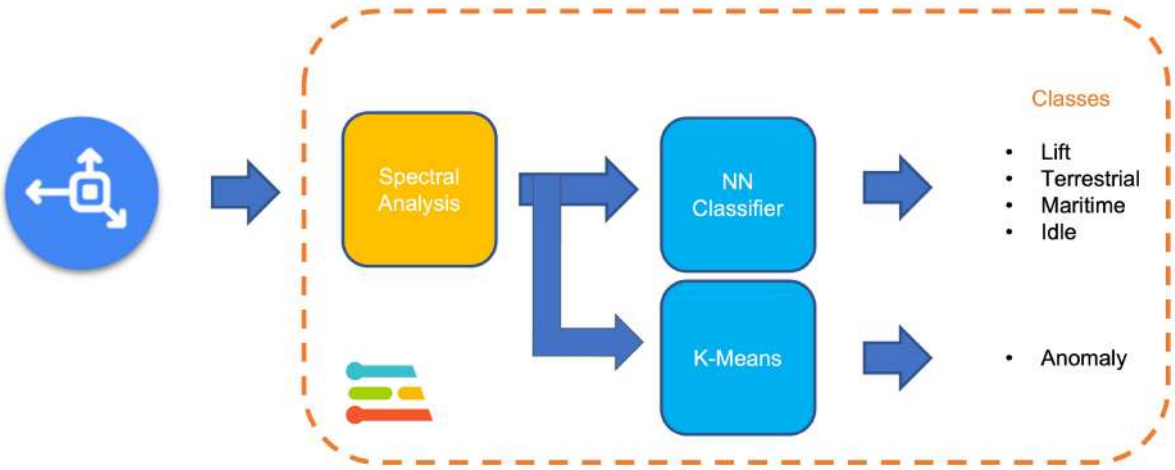
The next step is the definition of our Impulse, which takes the raw data and uses signal processing to extract features, passing them as the input tensor of a *learning block* to classify new data. Go to `Impulse Design` and `Create Impulse`. The Studio will suggest the basic design. Let's also add a second *Learning Block* for `Anomaly Detection`.



This second model uses a K-means model. If we imagine that we could have our known classes as clusters, any sample that could not fit on that could be an outlier, an anomaly such as a container rolling out of a ship on the ocean or falling from a Forklift.



The sampling frequency should be automatically captured, if not, enter it: `[50]` Hz. The Studio suggests a *Window Size* of 2 seconds (`[2000]` ms) with a *sliding window* of `[20]` ms. What we are defining in this step is that we will pre-process the captured data (Time-Seres data), creating a tabular dataset features) that will be the input for a Neural Networks Classifier (DNN) and an Anomaly Detection model (K-Means), as shown below:



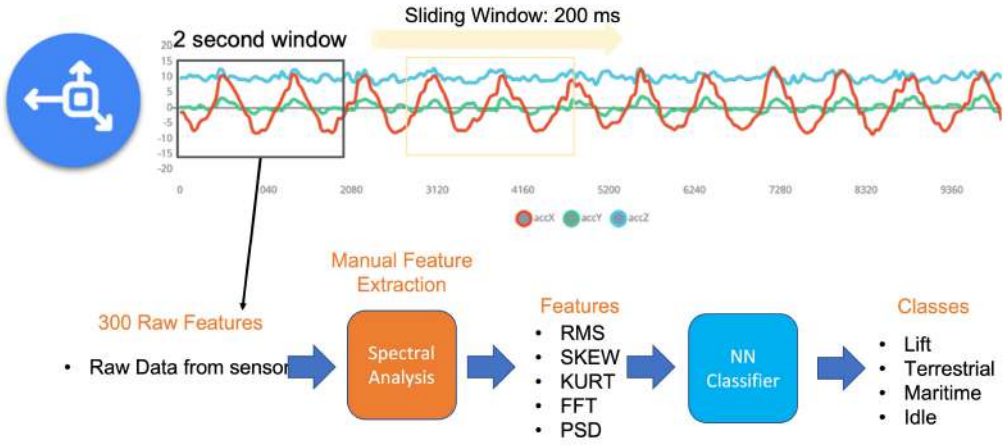
Let's dig into those steps and parameters to understand better what we are doing here.

Data Pre-Processing Overview

Data pre-processing is extracting features from the dataset captured with the accelerometer, which involves processing and analyzing the raw data. Accelerometers measure the acceleration of an object along one or more axes (typically three, denoted as X, Y, and Z). These measurements can be used to understand various aspects of the object's motion, such as movement patterns and vibrations.

Raw accelerometer data can be noisy and contain errors or irrelevant information. Preprocessing steps, such as filtering and normalization, can clean and standardize the data, making it more suitable for feature extraction. In our case, we should divide the data into smaller segments or **windows**. This can help focus on specific events or activities within the dataset, making feature extraction more manageable and meaningful. The **window size** and overlap (**window increase**) choice depend on the application and the frequency of the events of interest. As a thumb rule, we should try to capture a couple of "cycles of data".

> With a sampling rate (SR) of 50 Hz and a window size of 2 seconds, we will get 100 samples per axis, or 300 in total (3 axis × 2 seconds × 50 samples). We will slide this window every 200 ms, creating a larger dataset where each instance has 300 raw features.



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

- **Time-domain** features describe the data's statistical properties within each segment, such as mean, median, standard deviation, skewness, kurtosis, and zero-crossing rate.

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

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

EI Studio Spectral Features

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

On the Studio, the collected raw dataset will be the input of a Spectral Analysis block, which is excellent for analyzing repetitive motion, such as data from accelerometers. This block will perform a DSP (Digital Signal Processing), extracting features such as FFT or Wavelets.

For our project, once the time signal is continuous, we should use FFT with, for example, a length of `[32]`.

The per axis/channel **Time Domain Statistical features** are:

- RMS: 1 feature
- Skewness: 1 feature
- Kurtosis: 1 feature

The per axis/channel **Frequency Domain Spectral features** are:

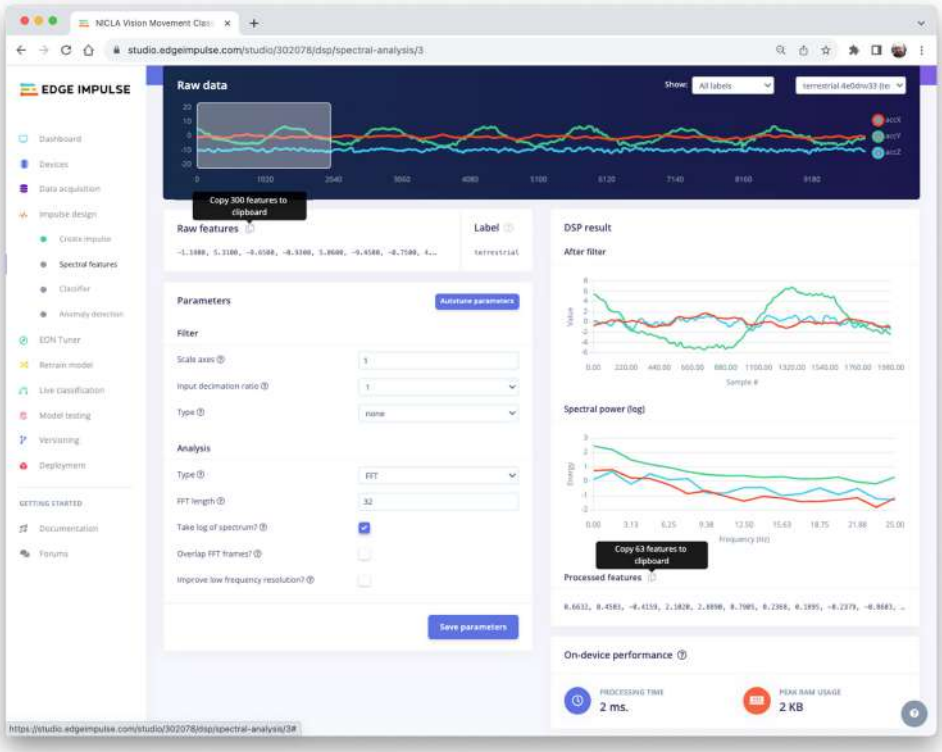
- Spectral Power: 16 features (FFT Length/2)
- Skewness: 1 feature
- Kurtosis: 1 feature

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

> You can learn more about how each feature is calculated by downloading the notebook Edge Impulse - Spectral Features Block Analysis TinyML under the hood: Spectral Analysis or opening it directly on Google CoLab.

Generating features

Once we understand what the pre-processing does, it is time to finish the job. So, let's take the raw data (time-series type) and convert it to tabular data. For that, go to the `Spectral Features` section on the `Parameters` tab, define the main parameters as discussed in the previous section (`[FFT]` with `[32]` points), and select `[Save Parameters]`:



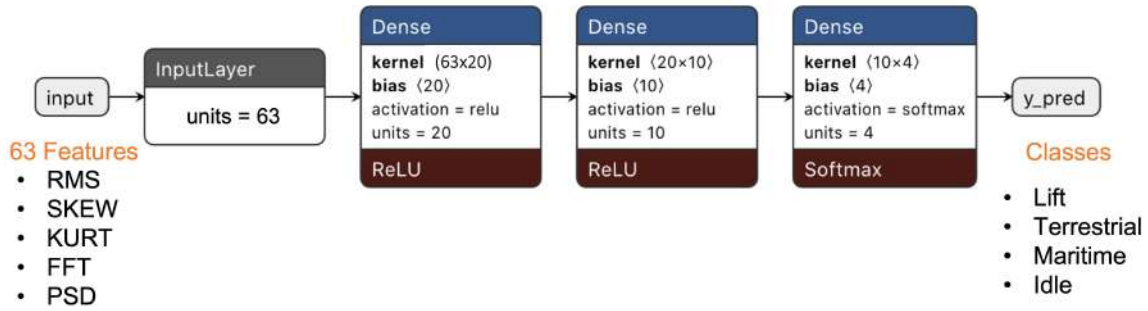
At the top menu, select the `Generate Features` option and the `Generate Features` button. Each 2-second window data will be converted into one data point of 63 features.

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

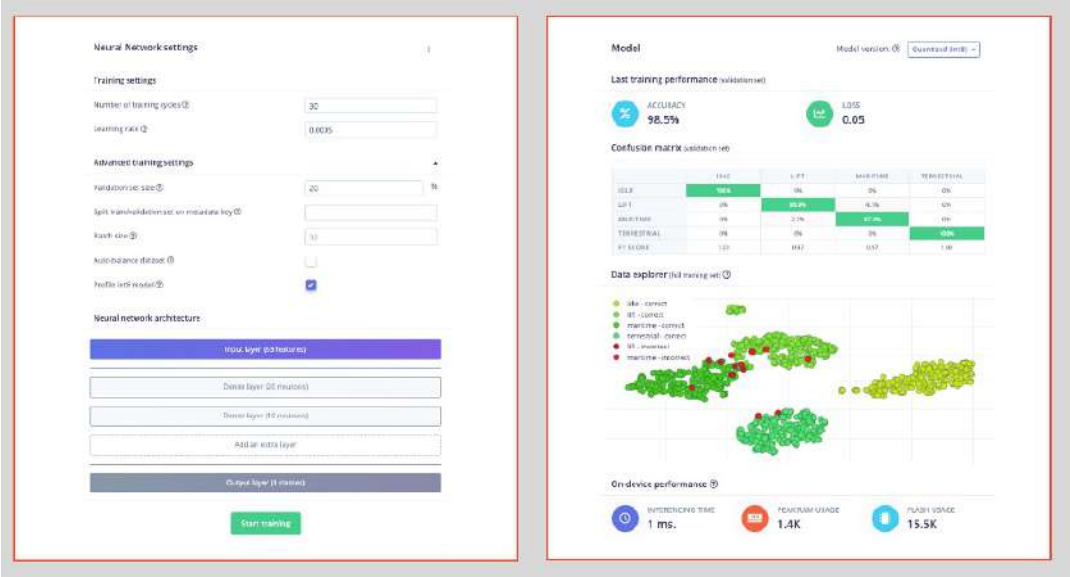
The visualization makes it possible to verify that after the feature generation, the classes present keep their excellent separation, which indicates that the classifier should work well. Optionally, you can analyze how important each one of the features is for one class compared with others.

Models Training

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



As hyperparameters, we will use a Learning Rate of `[0.005]`, a Batch size of `[32]`, and `[20]`% of data for validation for `[30]` epochs. After training, we can see that the accuracy is 98.5%. The cost of memory and latency is meager.



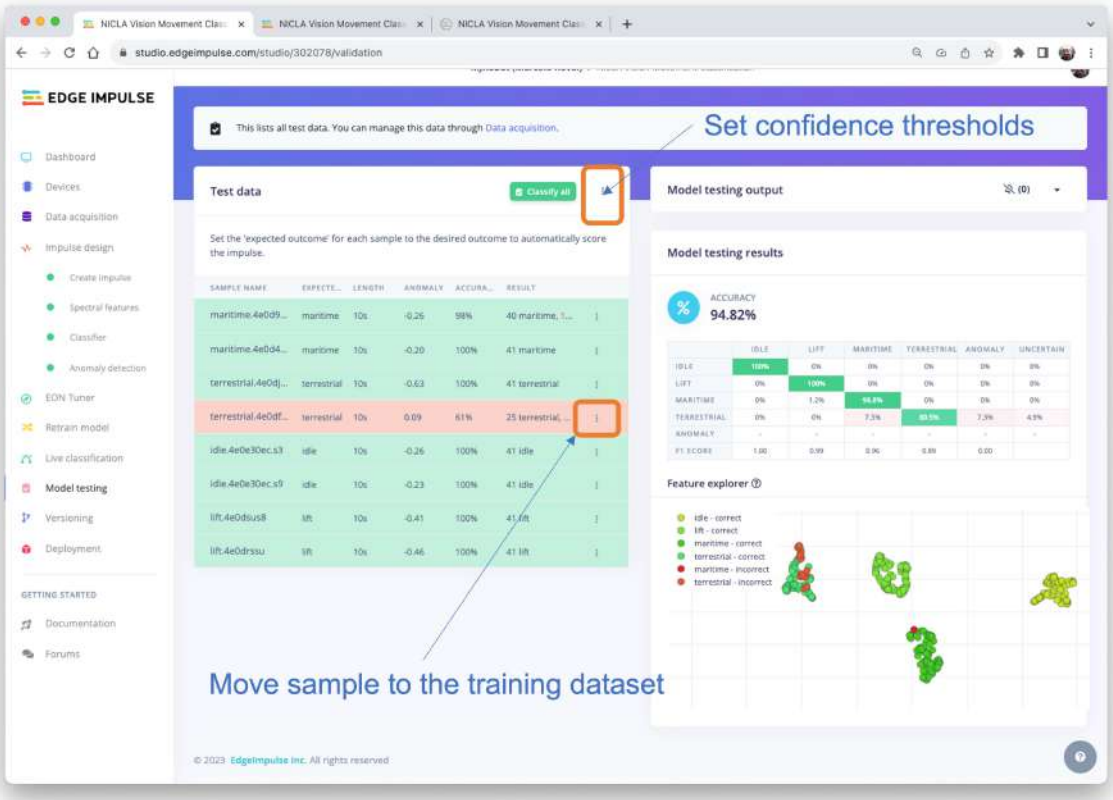
For Anomaly Detection, we will choose the suggested features that are precisely the most important ones in the Feature Extraction, plus the accZ RMS. The number of clusters will be `[32]`, as suggested by the Studio:

Testing

We can verify how our model will behave with unknown data using 20% of the data left behind during the data capture phase. The result was almost 95%, which is good. You can always

work to improve the results, for example, to understand what went wrong with one of the wrong results. If it is a unique situation, you can add it to the training dataset and then repeat it.

The default minimum threshold for a considered uncertain result is `[0.6]` for classification and `[0.3]` for anomaly. Once we have four classes (their output sum should be 1.0), you can also set up a lower threshold for a class to be considered valid (for example, 0.4). You can `Set confidence thresholds` on the `three dots` menu, besides the `Classify all` button.

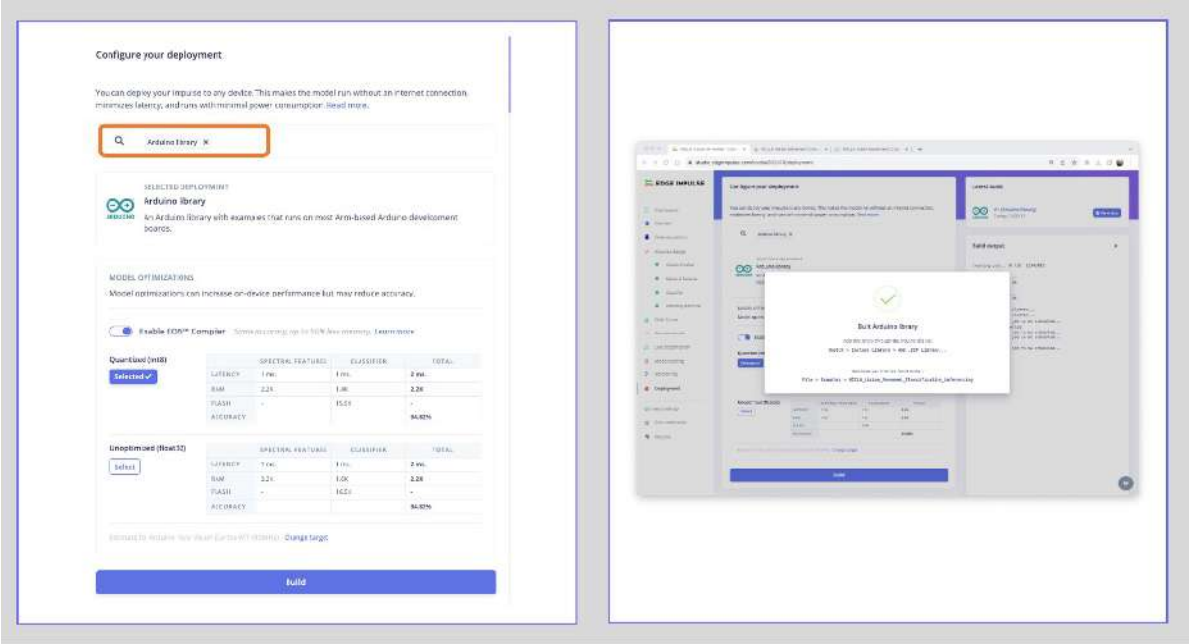


You can also perform Live Classification with your device (which should still be connected to the Studio).

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

Deploy

It is time to deploy the preprocessing block and the trained model to the Nicla. The Studio will package all the needed libraries, preprocessing functions, and trained models, downloading them to your computer. You should select the option `Arduino Library`, and at the bottom, you can choose `Quantized (Int8)` or `Unoptimized (float32)` and `[Build]`. A Zip file will be created and downloaded to your computer.



On your Arduino IDE, go to the `Sketch` tab, select `Add.ZIP Library`, and Choose the.zip file downloaded by the Studio. A message will appear in the IDE Terminal: `Library installed`.

Inference

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

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

Note that the code created by Edge Impulse considers a *sensor fusion* approach where the IMU (Accelerometer and Gyroscope) and the ToF are used. At the beginning of the code, you have the libraries related to our project, IMU and ToF:

```
/* Includes ---------------------------------------------- */
#include <NICLA_Vision_Movement_Classification_inferencing.h>
#include <Arduino_LSM6DSOX.h> //IMU
#include "VL53L1X.h" // ToF
```

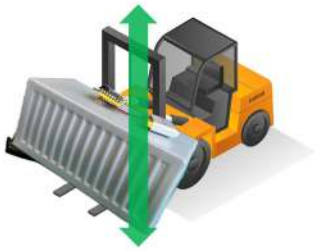
> You can keep the code this way for testing because the trained model will use only features pre-processed from the accelerometer. But consider that you will write your code only with the needed libraries for a real project.

And that is it!

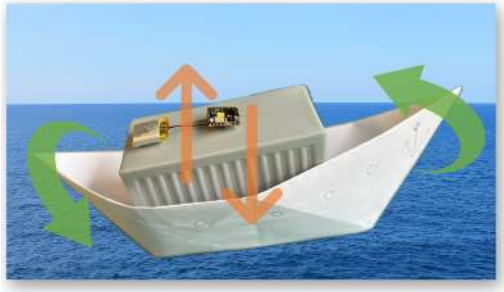
You can now upload the code to your device and proceed with the inferences. Press the Nicla `[RESET]` button twice to put it on boot mode (disconnect from the Studio if it is still connected), and upload the sketch to your board.

Now you should try different movements with your board (similar to those done during data capture), observing the inference result of each class on the Serial Monitor:

- **Idle and lift classes**:

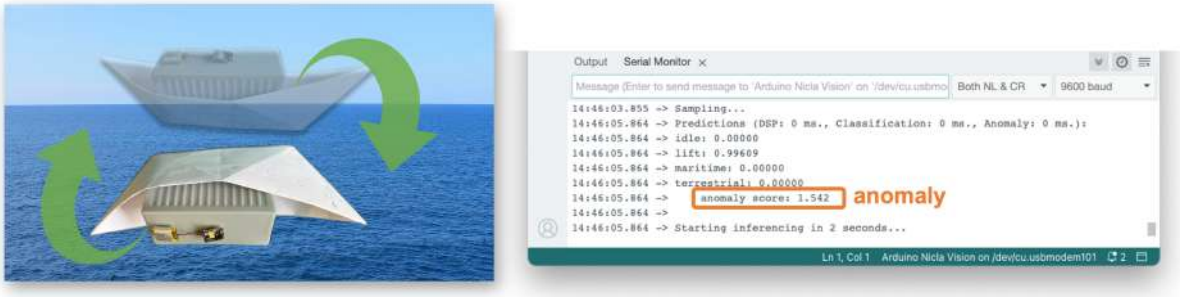


- **Maritime and terrestrial**:



Note that in all situations above, the value of the `anomaly score` was smaller than 0.0. Try a new movement that was not part of the original dataset, for example, "rolling" the Nicla, facing the camera upside-down, as a container falling from a boat or even a boat accident:

- **Anomaly detection**:



In this case, the anomaly is much bigger, over 1.00

Post-processing

Now that we know the model is working since it detects the movements, we suggest that you 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 5 V power supply).

The idea is to do the same as with the KWS project: if one specific movement is detected, a specific LED could be lit. For example, if *terrestrial* is detected, the Green LED will light; if *maritime*, the Red LED will light, if it is a *lift,* the Blue LED will light; and if no movement is detected *(idle*), the LEDs will be OFF. You can also add a condition when an anomaly is detected, in this case, for example, a white color can be used (all e LEDs light simultaneously).

Conclusion

> The notebooks and codeused in this hands-on tutorial will be found on the GitHub repository.

Before we finish, consider that Movement Classification and Object Detection can be utilized in many applications across various domains. Here are some of the potential applications:

Case Applications

Industrial and Manufacturing

- **Predictive Maintenance**: Detecting anomalies in machinery motion to predict failures before they occur.

- **Quality Control**: Monitoring the motion of assembly lines or robotic arms for precision assessment and deviation detection from the standard motion pattern.
- **Warehouse Logistics**: Managing and tracking the movement of goods with automated systems that classify different types of motion and detect anomalies in handling.

Healthcare

- **Patient Monitoring**: Detecting falls or abnormal movements in the elderly or those with mobility issues.
- **Rehabilitation**: Monitoring the progress of patients recovering from injuries by classifying motion patterns during physical therapy sessions.
- **Activity Recognition**: Classifying types of physical activity for fitness applications or patient monitoring.

Consumer Electronics

- **Gesture Control**: Interpreting specific motions to control devices, such as turning on lights with a hand wave.
- **Gaming**: Enhancing gaming experiences with motion-controlled inputs.

Transportation and Logistics

- **Vehicle Telematics**: Monitoring vehicle motion for unusual behavior such as hard braking, sharp turns, or accidents.
- **Cargo Monitoring**: Ensuring the integrity of goods during transport by detecting unusual movements that could indicate tampering or mishandling.

Smart Cities and Infrastructure

- **Structural Health Monitoring**: Detecting vibrations or movements within structures that could indicate potential failures or maintenance needs.
- **Traffic Management**: Analyzing the flow of pedestrians or vehicles to improve urban mobility and safety.

Security and Surveillance

- **Intruder Detection**: Detecting motion patterns typical of unauthorized access or other security breaches.
- **Wildlife Monitoring**: Detecting poachers or abnormal animal movements in protected areas.

Agriculture

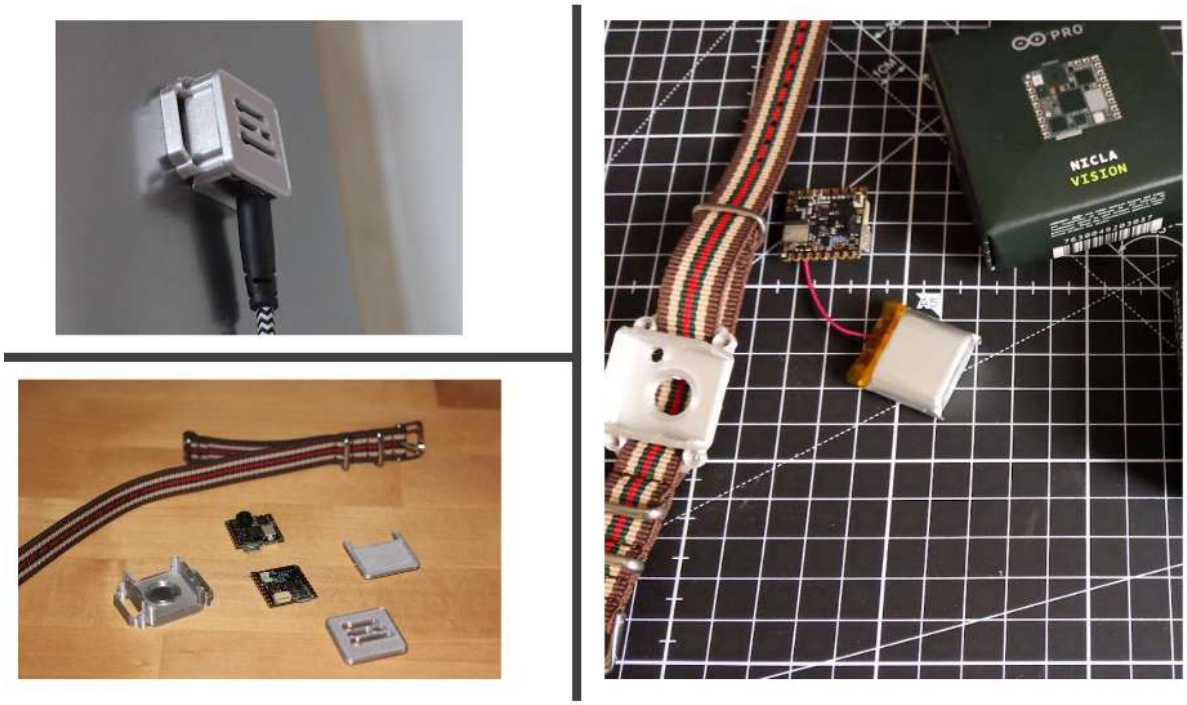
- **Equipment Monitoring**: Tracking the performance and usage of agricultural machinery.
- **Animal Behavior Analysis**: Monitoring livestock movements to detect behaviors indicating health issues or stress.

Environmental Monitoring

- **Seismic Activity**: Detecting irregular motion patterns that precede earthquakes or other geologically relevant events.
- **Oceanography**: Studying wave patterns or marine movements for research and safety purposes.

Nicla 3D case

For real applications, as some described before, we can add a case to our device, and Eoin Jordan, from Edge Impulse, developed a great wearable and machine health case for the Nicla range of boards. It works with a 10mm magnet, 2M screws, and a 16mm strap for human and machine health use case scenarios. Here is the link: Arduino Nicla Voice and Vision Wearable Case.



The applications for motion classification and anomaly detection are extensive, and the Arduino Nicla Vision is well-suited for scenarios where low power consumption and edge processing are advantageous. Its small form factor and efficiency in processing make it an ideal choice for deploying portable and remote applications where real-time processing is crucial and connectivity may be limited.

Resources

- Arduino Code
- Edge Impulse Spectral Features Block Colab Notebook
- Edge Impulse Project

References

To learn more:

Online Courses

- Harvard School of Engineering and Applied Sciences - CS249r: Tiny Machine Learning
- Professional Certificate in Tiny Machine Learning (TinyML) – edX/Harvard
- Introduction to Embedded Machine Learning - Coursera/Edge Impulse
- Computer Vision with Embedded Machine Learning - Coursera/Edge Impulse
- UNIFEI-IESTI01 TinyML: "Machine Learning for Embedding Devices"

Books

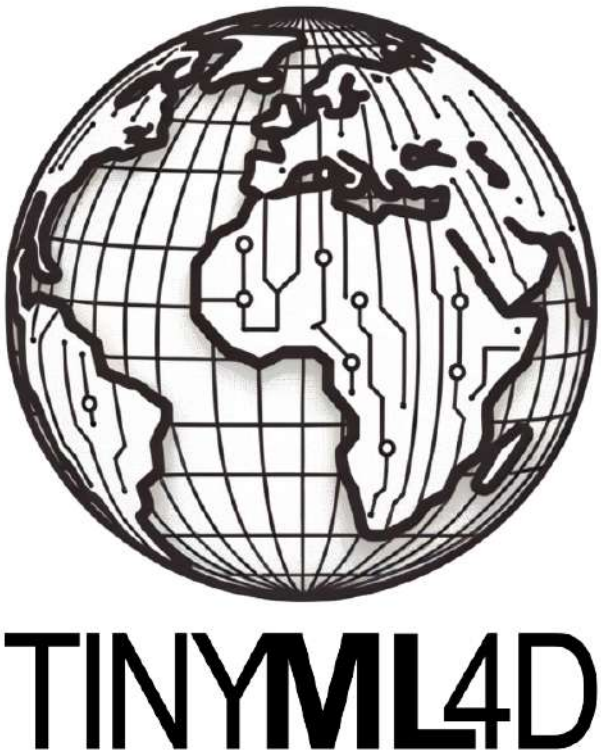
- "Python for Data Analysis" by Wes McKinney
- "Deep Learning with Python" by François Chollet - GitHub Notebooks
- "TinyML" by Pete Warden and Daniel Situnayake
- "TinyML Cookbook 2nd Edition" by Gian Marco Iodice
- "Technical Strategy for AI Engineers, In the Era of Deep Learning" by Andrew Ng
- "AI at the Edge" book by Daniel Situnayake and Jenny Plunkett
- "XIAO: Big Power, Small Board" by Lei Feng and Marcelo Rovai
- "Machine Learning Systems" by Vijay Janapa Reddi

Projects Repository

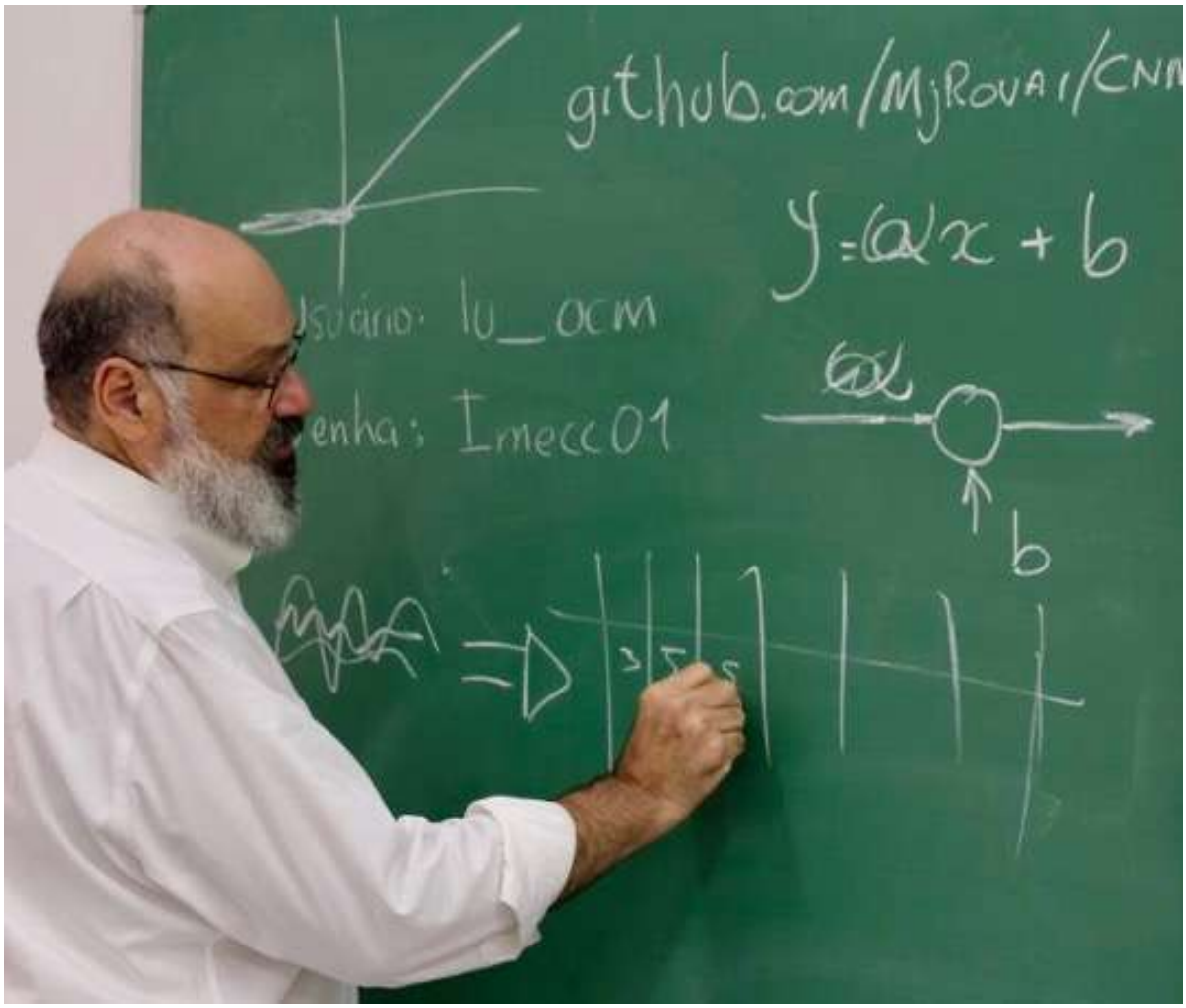
- Edge Impulse Expert Network

TinyML4D

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



About the author



Marcelo Rovai, a Brazilian living in Chile, is a recognized engineering and technology education figure. He holds the title of Professor Honoris Causa from the Federal University of Itajubá (UNIFEI), Brazil. His educational background includes an Engineering degree from UNIFEI and a specialization from the Polytechnic School of São Paulo University (POLI/USP).

Further enhancing his expertise, he earned an MBA from IBMEC (INSPER) and a Master's in Data Science from the Universidad del Desarrollo (UDD) in Chile.

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

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

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

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