



HARDWARE KITS

Marcelo Rovai
with Vijay Janapa Reddi
Harvard University

A Hands-On Companion to the
Machine Learning Systems textbook
mlsysbook.ai

January 6, 2026

Table of contents

Chapter 1 Hardware Kits	1
1.1 Hardware Platforms	2
1.2 What You Will Build	2
1.3 Getting Started	3
1.4 Part of the MLSysBook Ecosystem	3
Getting Started	5
Step 1: Select Your Platform	5
Step 2: Set Up Your Environment	6
Step 3: Choose Your First Lab	6
Step 4: Start Your First Lab	6
Prerequisites	7
Connection to ML Systems Textbook	7
Hardware Platforms	9
Featured Platform	9
Platform Overview	10
Platform Comparison	10
Platform Selection Guidelines	11
Hardware Platform Specifications	11
Getting Started	17
IDE Setup	19
System Requirements	19
Platform-Specific Software Installation	20
Development Tool Configuration	23
Environment Verification	24
Common Setup Issues and Solutions	25
Troubleshooting and Support	26
Ready for Laboratory Exercises	27

I key:arduino	29
II Arduino Nicla Vision	31
Overview	33
Where to Buy	33
Pre-requisites	34
Setup	34
Exercises	34
Setup	35
Overview	36
Hardware	36
Arduino IDE Installation	37
Installing the OpenMV IDE	42
Connecting the Nicla Vision to Edge Impulse Studio	48
Expanding the Nicla Vision Board (optional)	51
Summary	55
Resources	55
Image Classification	57
Overview	58
Computer Vision	59
Image Classification Project Goal	59
Data Collection	60
Training the model with Edge Impulse Studio	63
Dataset	63
The Impulse Design	67
Model Training	72
Model Testing	73
Deploying the model	74
Image Classification (non-official) Benchmark	85
Summary	86
Resources	86
Object Detection	87
Overview	88
The Object Detection Project Goal	90
Data Collection	92
Edge Impulse Studio	93
The Impulse Design	97
Model Design, Training, and Test	99
Deploying the Model	104

Summary	109
Resources	109
Keyword Spotting (KWS)	112
Overview	112
How does a voice assistant work?	113
The KWS Hands-On Project	114
Dataset	116
Creating Impulse (Pre-Process / Model definition)	121
Model Design and Training	124
Testing	127
Deploy and Inference	127
Post-processing	130
Summary	133
Resources	133
Motion Classification and Anomaly Detection	136
Overview	136
IMU Installation and testing	137
The Case Study: Simulated Container Transportation	141
Data Collection	142
Impulse Design	148
Models Training	153
Testing	154
Deploy	155
Summary	159
Resources	161
III key:xiao	163
IV Seeed XIAO ESP32S3	165
Overview	167
Where to Buy	167
Pre-requisites	168
Setup	168
Exercises	168
Setup	169
Overview	170
Installing the XIAO ESP32S3 Sense on Arduino IDE	174
Testing the board with BLINK	176

Microphone Test	177
Testing the Camera	182
Testing WiFi	187
Testing the IMU Sensor (LSM6DS3TR-C)	192
Testing the OLED Display (SSD1306)	197
Summary	202
Resources	203
Appendix	205
Heat Sink Considerations	205
Installing the Heat Sink	206
Image Classification	209
Overview	209
Image Classification	210
An Image Classification Project	215
Image Classification Project from a Dataset	225
Training the model with Edge Impulse Studio	226
Model Deployment	233
Summary	242
Resources	243
Object Detection	245
Overview	245
The Object Detection Project Goal	248
Data Collection	249
Edge Impulse Studio	251
The Impulse Design	257
Model Design, Training, and Test	259
Deploying the Model (Arduino IDE)	263
Deploying the Model (SenseCraft-Web-Toolkit)	267
Summary	271
Resources	271
Keyword Spotting (KWS)	273
Overview	274
The KWS Project	275
Dataset	278
Training model with Edge Impulse Studio	291
Testing	298
Deploy and Inference	299
Postprocessing	303
Summary	306
Resources	308

Motion Classification and Anomaly Detection	310
Overview	310
Installing the IMU	311
The TinyML Motion Classification Project	314
Data Collection	315
Data Collection at the Studio	319
Data Pre-Processing	322
Model Design	324
Impulse Design	325
Generating features	326
Training	328
Testing	330
Deploy	331
Inference	332
Post-Processing	339
Summary	339
Resources	340
 V key:grove	 341
 VI Grove Vision AI V2	 343
 Overview	 345
Where to Buy	346
Pre-requisites	346
Setup and No-Code Applications	347
Exercises	347
 Setup and No-Code Applications	 349
Introduction	349
The SenseCraft AI Studio	353
Exploring CV AI models	356
An Image Classification Project	364
Summary	370
Resources	372
 Image Classification	 373
Introduction	374
Summary	399
Resources	400
 Object Detection	 401

VII key:raspberry	403
VIII Raspberry Pi	405
Overview	407
Where to Buy	408
Pre-requisites	408
Setup	408
Exercises	408
Setup	411
Overview	412
Hardware Overview	414
Installing the Operating System	415
Remote Access	419
Increasing SWAP Memory	423
Installing a Camera	425
Running the Raspi Desktop remotely	433
Updating and Installing Software	437
Model-Specific Considerations	437
Image Classification	439
Overview	440
Setting Up the Environment	441
Making inferences with Mobilenet V2	447
Image Classification Project	457
Training the model with Edge Impulse Studio	466
The Impulse Design	468
Live Image Classification	481
Summary:	489
Resources	490
Object Detection	492
Overview	492
Pre-Trained Object Detection Models Overview	496
Object Detection Project	503
Training an SSD MobileNet Model on Edge Impulse Studio	511
Training a FOMO Model at Edge Impulse Studio	527
Exploring a YOLO Model using Ultralytics	538
Object Detection on a live stream	553
Summary	558
Resources	559

Small Language Models (SLM)	561
Overview	562
Setup	562
Generative AI (GenAI)	565
Ollama	569
Ollama Python Library	582
SLMs: Optimization Techniques	601
RAG Implementation	602
Summary	609
Resources	611
Vision-Language Models (VLM)	613
Introduction	613
Technical Overview	616
Setup and Installation	619
Florence-2 Tasks	632
Exploring computer vision and vision-language tasks	634
Latency Summary	652
Fine-Tuning	654
Summary	654
Future Implications	657
Resources	657
IX key:shared	659
X Shared Resources	661
Overview	663
KWS Feature Engineering	665
Overview	666
The KWS	666
Overview to Audio Signals	668
Overview to MFCCs	670
Hands-On using Python	674
Summary	674
Resources	676
DSP Spectral Features	677
Overview	678
Extracting Features Review	678
A TinyML Motion Classification project	679

Data Pre-Processing	681
Time Domain Statistical features	688
Spectral features	691
Time-frequency domain	693
Summary	702

XI key:backmatter	705
--------------------------	------------

Chapter 1

Hardware Kits

Hands-On Embedded ML Labs for Real-World Deployment



Figure 1.1: Hardware platforms for embedded ML labs

These hands-on laboratories accompany the Machine Learning Systems textbook, bringing theory to life on real hardware. Deploy machine learning on embedded devices you can hold in your hand, from image classification to voice recognition to motion detection. Professional

development boards costing \$25-100 provide immediate, tangible feedback: LEDs light up, motors spin, and buzzers sound when your model runs successfully.

Working within the resource constraints of embedded devices (typically 2MB of RAM and 1MB of flash) forces you to confront the same engineering trade-offs that define large-scale ML systems, but in a tangible environment where every optimization decision has immediate, observable consequences.

Laboratory Development

These hands-on laboratories were co-designed by Prof. Vijay Janapa Reddi and Marcelo Rovai, with Marcelo leading their development. His decades of embedded systems expertise shaped accessible, practical learning experiences that bridge theory with real-world implementation.

1.1 Hardware Platforms

Table 1.1: Hardware platform comparison

Platform	Price	Best For	Capabilities
Grove Vision AI V2	~\$25	Beginners	Vision, Plug & Play
XIAO ESP32S3	~\$40	Best Value	Vision, Audio, Motion
Raspberry Pi	~\$60-80	Advanced	Vision, LLM, VLM
Arduino Nicla Vision	~\$95	Professional	Vision, Audio, Motion

1.2 What You Will Build

Computer Vision: Image classification and object detection on microcontrollers. Train models to recognize objects, detect faces, or classify scenes, then deploy them to devices running on battery power.

Audio Processing: Keyword spotting and voice command recognition. Build wake-word detectors and simple voice interfaces that run entirely on-device without cloud connectivity.

Motion Classification: Activity and gesture recognition from IMU data. Create wearable-style applications that detect walking, running, or custom gestures using accelerometer and gyroscope sensors.

Large Language Models: Run LLMs and VLMs on edge devices using Raspberry Pi. Experience the frontier of on-device AI with models that can understand and generate text.

1.3 Getting Started

1. **Choose Hardware:** Select a platform based on your budget and learning goals. See Platforms for detailed comparisons.
2. **Set Up Environment:** Install Arduino IDE or platform-specific tools. Follow the IDE Setup Guide for step-by-step instructions.
3. **Build & Deploy:** Work through the labs for your chosen platform. Start with Getting Started for an overview of available exercises.

1.4 Part of the MLsysBook Ecosystem

These hardware labs complement the broader ML Systems learning experience:

- **Textbook:** Comprehensive theory and concepts covering the full ML systems stack
- **Hardware Kits:** Hands-on embedded deployment (you are here)
- **TinyTorch:** Build your own ML framework from scratch

Getting Started

This guide walks you through selecting hardware, configuring your development environment, and running your first embedded ML application. Most students complete setup in under an hour.

Step 1: Select Your Platform

Your choice depends on budget, learning objectives, and the types of applications you want to build.

For beginners or budget-conscious learners:

Platform	Cost	Why Choose It
Grove Vision AI V2	~\$25	No-code interface, fastest path to running models
XIAO ESP32S3	~\$40	Best value, supports vision, audio, and motion

For advanced applications:

Platform	Cost	Why Choose It
Raspberry Pi	~\$60-80	Full Linux environment, LLMs and VLMs
Nicla Vision	~\$95	Professional-grade, ultra-low power design

For detailed specifications and technical comparisons, see Platforms.

Step 2: Set Up Your Environment

Development environment configuration is platform-dependent but follows a common pattern: install software tools, configure communication with hardware, and verify the setup works.

Time estimate: 30-60 minutes depending on platform and internet speed.

Follow the IDE Setup Guide for complete procedures covering:

- System requirements for your development computer
- Arduino IDE installation for microcontroller platforms
- Python environment configuration for Raspberry Pi
- SenseCraft AI web interface for Grove Vision AI V2
- Serial communication and hardware verification

Step 3: Choose Your First Lab

Each platform supports different exercise categories. Select labs that match both your hardware and learning goals.

Table 1.4: Exercise availability by platform

Lab Category	Grove Vision	XIAO	Nicla	Raspberry Pi
Image Classification	✓	✓	✓	✓
Object Detection	✓	✓	✓	✓
Keyword Spotting		✓	✓	
Motion Classification		✓	✓	
Large Language Models				✓
Vision Language Models				✓

Step 4: Start Your First Lab

Grove Vision AI V2: Begin with Setup and No-Code Apps. You'll deploy a pre-trained model in minutes using the visual interface.

XIAO ESP32S3: Start with Setup, then proceed to Image Classification to train and deploy your first custom model.

Nicla Vision: Complete Setup to configure your board, then try Image Classification.

Raspberry Pi: Follow Setup, then choose your path: - Image Classification for computer vision fundamentals - LLM Deployment to run language models on edge hardware

Prerequisites

These labs assume:

- **Programming:** Proficiency in Python. Familiarity with C/C++ is helpful for microcontroller platforms but not required.
- **Mathematics:** Working knowledge of linear algebra and basic probability at the undergraduate level.
- **Hardware:** No prior embedded systems experience. Each lab includes complete setup and troubleshooting procedures.

Connection to ML Systems Textbook

These laboratories complement specific chapters in the ML Systems textbook:

- **Image Classification labs** reinforce concepts from the Computer Vision and Model Optimization chapters
- **Keyword Spotting labs** connect to Audio Processing and Real-time Inference
- **Motion Classification labs** demonstrate Sensor Fusion and Time-series Analysis
- **LLM/VLM labs** extend Large Model Deployment to resource-constrained environments

Each lab identifies relevant textbook sections for deeper theoretical understanding.

Hardware Platforms

This chapter provides detailed technical specifications for the four hardware platforms used in these laboratories. Each platform represents a different point along the spectrum of embedded computing capabilities, from ultra-low-power microcontrollers to full-featured edge computers.

These platforms were selected because they illustrate distinct engineering trade-offs in power consumption, computational capability, and development complexity. All are widely used in commercial applications, ensuring that skills developed here transfer directly to professional embedded systems work.

Featured Platform



The XIAOML Kit
A hands-on introduction to machine learning systems using TinyML.
Designed by Professor Vijay Janapa Reddi (Harvard University), author of the Machine Learning Systems textbook.

What's inside: XIAO ESP32-S3 Sense CAM + IMU + Heatsinks + Labs + SD Toolkit

Build: Build keyword detection, image classification, motion detection, object detection, and more

For: For learners, educators, and real-world builders

Learners: mlsysbook.ai Builders: mlsysbook.ai/kits Developers: github.com/mlsysbook

Figure 1.2: Complete XIAOML Kit with all components

The XIAOML Kit is the most recent addition to our educational hardware platforms (released on July 31st, 2025). It offers a comprehensive TinyML development environment for learning about ML systems, featuring integrated wireless connectivity, a camera, multiple sensors, and extensive documentation. This compact board exemplifies how contemporary embedded systems can efficiently provide advanced machine learning capabilities within a cost-effective framework.

Platform Overview

Our curriculum features four carefully selected platforms that span the full spectrum of embedded computing capabilities. Each platform shown in Table 1.5 has been chosen to illustrate specific engineering trade-offs and learning objectives.

Table 1.5: Platform selection strategy table.

Platform	Primary Learning Focus	Cost	Power Profile	Best For
XIAOML Kit	IoT & Wireless ML	~\$40	Low Power	Cost-sensitive deployments
Arduino Nicla	Ultra-low Power Design	~\$95	Ultra-low	Battery-powered devices
Grove Vision AI	Hardware Acceleration	~\$25	Medium	Industrial applications
Raspberry Pi	Full ML Frameworks	\$60-145	High	Advanced edge computing

Platform Comparison

Table 1.6 provides a comprehensive technical comparison of all four platforms.

Table 1.6: Platform comparison matrix.

Characteristic	XIAOML Kit	Raspberry Pi	Arduino Nicla	Grove Vision AI V2
Cost Range (USD)	~\$40	\$60-145	~\$95	~\$25
Power Consumption	Low	High	Ultra-low	Medium
Processing Power	Medium	Very High	Low	High (NPU)
Memory Capacity	8MB	1-16GB	2MB	16MB
Primary Use Case	IoT networks	Edge computing	Battery devices	Industrial AI

Characteristic	XIAOML Kit	Raspberry Pi	Arduino Nicla	Grove Vision AI V2
ML Framework	TF Lite	TensorFlow, PyTorch	TensorFlow Lite	SenseCraft AI
Development Env.	Arduino/PlatformIO	Python/Linux	Arduino IDE	Visual/Code

Platform Selection Guidelines

Selecting the appropriate platform depends on specific learning objectives and project requirements. Table 1.7 provides a systematic mapping to guide these decisions.

Table 1.7: Platform capabilities matrix.

Learning Objective/Application	XIAOML Kit	Ras Pi	Arduino Nicla	Grove Vision AI V2
Embedded Systems Basics	✓	Limited	✓	✓
Wireless Connectivity	✓	✓		✓
Ultra-Low Power Design			✓	
Full ML Frameworks		✓		
Hardware Acceleration				✓
Real-time Vision	Limited	✓	✓	✓
Edge-Cloud Integration	✓	✓		✓
Production Deployment	✓		✓	✓

Hardware Platform Specifications

This section provides detailed technical specifications for each platform, including processor architecture, memory hierarchy, sensor capabilities, and development toolchain requirements.

XIAOML Kit (Seeed Studio)

💡 Best For: IoT & Wireless ML

The XIAOML Kit excels at wireless connectivity and cost-sensitive deployments. It's perfect for learning IoT sensor networks, remote monitoring systems, and wireless ML inference where you need reliable connectivity in a compact, affordable package.

The XIAO ESP32S3 represents the category of ultra-compact, wireless-enabled microcontrollers optimized for IoT applications. The name “XIAO” () translates to “tiny” in Chinese, reflecting the board’s 21×17.5mm form factor.



Figure 1.3: XIAO ESP32S3 development board

Processor Architecture: ESP32-S3 dual-core Xtensa LX7 running at 240MHz

Memory Hierarchy: 8MB PSRAM and 8MB Flash storage

Connectivity: WiFi 802.11 b/g/n and Bluetooth 5.0

Integrated Sensors: OV2640 camera sensor, digital microphone, 6-axis inertial measurement unit

Power Characteristics: 3.3V operation with multiple low-power modes

Development Environment: Arduino IDE and PlatformIO support with extensive library ecosystem. Supports C/C++ programming with Arduino-style abstractions and direct ESP-IDF for advanced users.

Application Focus: IoT sensor networks, remote monitoring systems, wireless ML inference, cost-sensitive deployments

Arduino Nicla Vision

💡 Best For: Ultra-Low Power Design

The Arduino Nicla Vision is optimized for battery-powered devices and always-on sensing applications. It's ideal for learning ultra-low power design, image classification systems, and object detection applications where battery life is measured in months, not hours.

The Nicla Vision exemplifies professional-grade embedded vision systems built around the STM32H7 microcontroller. This platform demonstrates how specialized hardware design enables sophisticated ML inference within severe resource constraints.

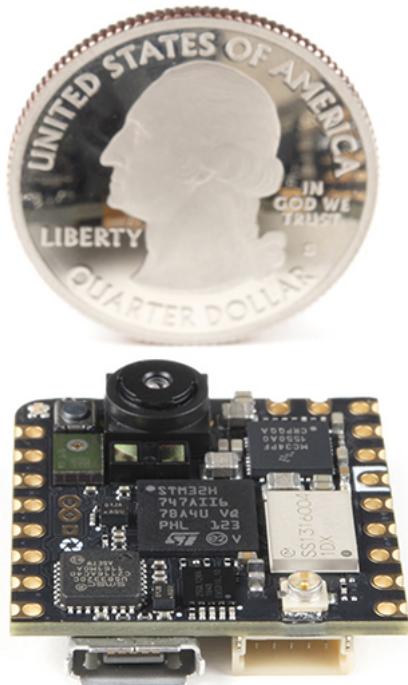


Figure 1.4: Arduino Nicla Vision with camera module

Processor Architecture: STM32H747 dual-core ARM Cortex-M7/M4 running at 480MHz

Memory Hierarchy: 2MB integrated RAM and 16MB Flash storage

Integrated Sensors: GC2145 camera sensor, MP34DT05 digital microphone, 6-axis IMU

Power Characteristics: 3.3V operation optimized for battery-powered deployment

Development Environment: Arduino IDE and OpenMV IDE support with specialized computer vision libraries. MicroPython support for rapid prototyping alongside C/C++ for production deployments.

Application Focus: Battery-powered devices, image classification systems, object detection applications, always-on sensing

Grove Vision AI V2

💡 Best For: Hardware Acceleration

The Grove Vision AI V2 features dedicated neural processing hardware for orders-of-magnitude performance improvements. It's perfect for learning industrial inspection systems, real-time video analytics, and advanced object detection where you need NPU-accelerated inference capabilities.

The Grove Vision AI V2 incorporates dedicated neural processing hardware (NPU) to demonstrate hardware-accelerated ML inference. This platform illustrates how specialized AI processors achieve orders-of-magnitude performance improvements over software-only implementations.

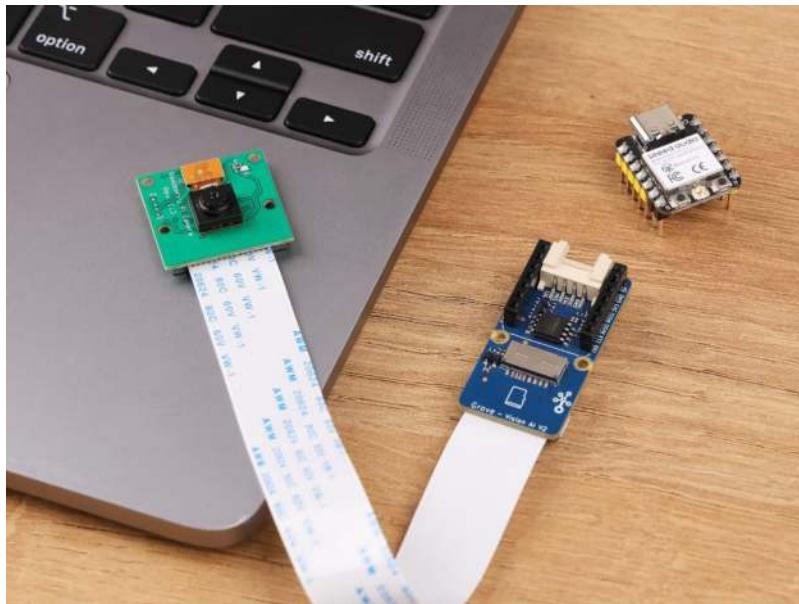


Figure 1.5: Grove Vision AI V2 with NPU

Processor Architecture: ARM Cortex-M55 with integrated Ethos-U55 NPU

Memory Hierarchy: 16MB external memory for model and data storage

Neural Processing Unit: Dedicated hardware accelerator for ML inference

Camera Interface: Standard CSI connector supporting various camera modules

Audio Input: Onboard digital microphone

Development Environment: SenseCraft AI visual programming platform for no-code development, with Arduino IDE support for custom applications. Supports both graphical programming and traditional C/C++ development workflows.

Application Focus: Industrial inspection systems, real-time video analytics, advanced object detection, NPU-accelerated inference

Raspberry Pi (Models 4/5 and Zero 2W)

Best For: Full ML Frameworks

The Raspberry Pi bridges embedded systems and traditional computing, providing a complete Linux environment for advanced ML applications. It's ideal for learning edge AI gateways, advanced computer vision systems, language model deployment, and multi-modal AI applications where you need full computing capabilities.

The Raspberry Pi family bridges embedded systems and traditional computing, providing a full Linux environment while maintaining educational accessibility. This platform demonstrates how increased computational resources enable sophisticated ML applications.

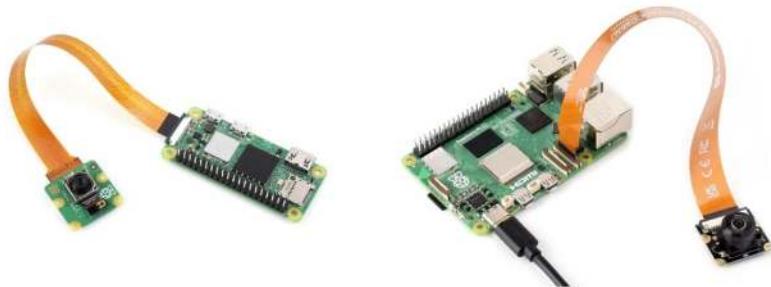


Figure 1.6: Raspberry Pi 5 and Pi Zero 2W comparison

Processor Architecture: ARM Cortex-A76 (Pi 5) or Cortex-A53 (Zero 2W)

Memory Hierarchy: 1-16GB DDR4 RAM depending on model

Storage: MicroSD card primary storage with USB 3.0 expansion

Connectivity: Gigabit Ethernet, WiFi, Bluetooth, multiple USB ports

Camera Interface: Dedicated CSI connector plus USB camera support

Operating System: Debian-based Raspberry Pi OS (full Linux distribution)

Development Environment: Full Linux development environment with native Python, C/C++, and JavaScript support. Package managers (apt, pip) provide access to extensive ML libraries including TensorFlow, PyTorch, and OpenCV.

Application Focus: Edge AI gateways, advanced computer vision systems, language model deployment, multi-modal AI applications

Getting Started

To get started with the hardware kits used in this course, you can purchase them directly from the following official sources:

- Seeed Studio – XIAOML Kit and Grove Vision AI V2 Module
- Arduino Store – Nicla Vision
- Raspberry Pi Foundation – Boards and Kits
- DigiKey, Mouser, SparkFun — Alternative distributors for a variety of components and kits

Check each site for educational discounts, bundles, and regional availability. Most kits are available as starter packages that include the board and basic accessories.

IDE Setup

Setting up your development environment is a critical first step that determines your success throughout the laboratory sequence. Unlike cloud-based ML development where infrastructure is abstracted away, embedded systems require understanding the complete toolchain from code compilation to hardware deployment.

Environment setup typically takes 30-60 minutes, depending on platform choice and internet speed. These procedures are designed for students with no prior embedded systems experience.

System Requirements

Before beginning installation, verify your development computer meets these requirements:

Development Computer:

- **Operating System:** Windows 10/11, macOS 10.15+, or Linux (Ubuntu 18.04+)
- **Memory:** 8GB RAM minimum (16GB recommended for Raspberry Pi development)
- **Storage:** 10GB free space for development tools and libraries
- **USB Ports:** At least one USB 2.0/3.0 port for device connection
- **Internet Connection:** Required for software installation and library downloads

Software Prerequisites:

- **Arduino IDE 2.0+** for Arduino-based platforms (XIAO, Nicla Vision)
- **Python 3.8+** for Raspberry Pi development

- **Git** for version control and example code access
- **Text Editor/IDE** such as VS Code or PyCharm

Hardware Accessories:

- **USB cables:** USB-C or Micro-USB (must support data transfer, not power-only)
- **SD Card:** 32GB+ Class 10 for Raspberry Pi
- **Power adapters:** Appropriate for each platform
- **Camera modules:** Included with most kits or available separately

Platform-Specific Software Installation

Each hardware platform demands different development approaches that mirror real-world embedded engineering practices. Arduino-based systems focus on resource efficiency and real-time constraints, Raspberry Pi demonstrates comprehensive edge computing capabilities, while specialized AI hardware highlights dedicated acceleration techniques.

Select the installation procedures appropriate for your chosen hardware platform.

Arduino-Based Platforms (Nicla Vision, XIAOMI Kit)

Arduino-based embedded systems provide direct hardware control with minimal abstraction layers, making them ideal for understanding resource constraints and optimization techniques. The development environment emphasizes immediate feedback between code changes and system behavior.

Arduino IDE Installation:

1. Download Arduino IDE 2.0 from arduino.cc/software
2. Install following the platform-specific setup wizard
3. Launch Arduino IDE and navigate to File → Preferences
4. Add board support URLs:
 - For XIAOMI Kit: https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_index.json

- For Nicla Vision: URL provided in Arduino IDE Board Manager

Board Package Installation:

1. Open Tools → Board → Boards Manager
2. Search for your platform:
 - XIAOMI Kit: Search “ESP32” and install “esp32 by Espressif Systems”
 - Nicla Vision: Search “Arduino Mbed OS Nicla Boards” and install
3. Select your board from Tools → Board menu
4. Install required libraries via Library Manager

Essential Libraries:

- TensorFlow Lite Micro
- Platform-specific camera drivers
- Sensor interface libraries (I2C, SPI)

Grove Vision AI V2 Platform

This platform introduces hardware-accelerated AI through dedicated neural processing units, demonstrating how specialized silicon achieves performance improvements impossible with general-purpose processors. The visual programming interface showcases rapid prototyping capabilities, while traditional development environments offer more extensive customization options.

SenseCraft AI Setup:

1. Create an account at sensecraft.seeed.cc
2. Connect Grove Vision AI V2 via USB
3. Access the device through the SenseCraft AI web interface
4. No local software installation required for the visual programming workflow

Arduino IDE Setup (for custom development):

Follow Arduino-based platform instructions above, using the Seeed Studio board package URL in the board manager.

Raspberry Pi Platform

The Raspberry Pi environment bridges embedded constraints with full computing capabilities, enabling students to experience both resource optimization and advanced ML frameworks. This dual perspective illustrates how computational resources impact algorithmic choices and system architecture decisions.

Operating System Installation:

1. Download Raspberry Pi Imager
2. Flash Raspberry Pi OS (64-bit recommended) to a microSD card (32GB minimum)
3. Configure SSH access and WiFi credentials during the imaging process
4. Insert the flashed SD card and boot the Raspberry Pi

Software Environment Setup:

The following commands establish a complete Python-based ML development environment with proper dependency management:

```
# Update system packages
sudo apt update && sudo apt upgrade -y

# Install Python development tools
# python3-pip: Python package installer
# python3-venv: Virtual environment creation
# python3-dev: Python development headers
sudo apt install python3-pip \
    python3-venv \
    python3-dev -y

# Install ML framework dependencies
# libatlas-base-dev: Linear algebra library (BLAS/LAPACK)
# libhdf5-dev: HDF5 data format library
# libhdf5-serial-dev: HDF5 serial version
sudo apt install libatlas-base-dev \
    libhdf5-dev \
    libhdf5-serial-dev -y

# Install computer vision dependencies
# libcamera-dev: Camera interface library
# python3-libcamera: Python bindings for libcamera
# python3-kms++: Kernel mode setting library
```

```
sudo apt install libcamera-dev \
    python3-libcamera \
    python3-kms++ -y

# Create virtual environment for projects
python3 -m venv ~/ml_projects
source ~/ml_projects/bin/activate

# Install core ML packages
# tensorflow: Main ML framework
# tensorflow-lite: Optimized for edge/mobile devices
# opencv-python: Computer vision library
# numpy: Numerical computing foundation
pip install tensorflow \
    tensorflow-lite \
    opencv-python \
    numpy
```

Development Tool Configuration

Proper tool configuration ensures reliable communication between your development workstation and embedded hardware. These settings establish the foundation for code deployment, debugging, and performance monitoring throughout the laboratory exercises.

Serial Communication Setup

Serial communication provides the primary interface for debugging and data monitoring in embedded systems, offering insights into system behavior that are essential for understanding performance constraints and optimization opportunities.

Windows:

- Install appropriate USB-to-serial drivers (CH340, FTDI, or platform-specific)
- Configure Device Manager to recognize the development board

macOS/Linux:

- Most USB-to-serial adapters work without additional drivers
- Verify device detection: `ls /dev/tty*` (macOS/Linux)

- Add user to dialout group: `sudo usermod -a -G dialout $USER` (Linux)

IDE Configuration

Development environment settings directly impact the efficiency of the code-test-deploy cycle that characterizes embedded development. Proper configuration reduces debugging time and provides clear feedback about system performance.

Arduino IDE Settings:

- Configure the appropriate COM port under Tools → Port
- Set the correct board and processor selection
- Verify upload speed (typically 115200 baud)
- Enable verbose output during compilation for debugging

Raspberry Pi Development:

- Configure SSH keys for remote development
- Install VS Code with Python and Remote SSH extensions
- Set up Jupyter notebook access for interactive development

Environment Verification

Verification procedures confirm that your development environment can successfully communicate with hardware and execute basic operations. These tests establish baseline functionality before proceeding to more complex laboratory exercises.

Hardware Detection Tests

The following verification procedures test core functionality required for laboratory exercises, ensuring that both hardware communication and software libraries operate correctly.

Arduino Platforms:

```
void setup() {  
    Serial.begin(115200);  
    Serial.println("Development environment test");  
    Serial.print("Board: ");  
    Serial.println(ARDUINO_BOARD);  
}  
  
void loop() {  
    Serial.println("Environment operational");  
    delay(1000);  
}
```

Raspberry Pi:

```
# Test camera interface  
libcamera-hello --timeout 5000  
  
# Test Python ML environment  
python3 -c \  
    "import tensorflow as tf; print('TensorFlow:', tf.__version__)"  
python3 -c \  
    "import cv2; print('OpenCV:', cv2.__version__)"
```

Grove Vision AI V2:

- Verify device detection in the SenseCraft AI web interface
- Test basic model deployment through visual programming interface

Common Setup Issues and Solutions

Setup challenges are common and offer valuable learning opportunities regarding embedded system constraints and debugging techniques. The following solutions address the most frequently encountered issues during environment configuration.

Device Connection Problems:

- Verify the USB cable supports data transfer (not power-only)
- Install platform-specific USB drivers if the device is not recognized
- Try different USB ports or USB hubs if the connection is unstable

Compilation Errors:

- Confirm the correct board and processor selection in the IDE

- Verify all required libraries are installed with compatible versions
- Check for sufficient disk space for the compilation process

Runtime Issues:

- Ensure adequate power supply (especially for camera operations)
- Verify SD card compatibility and formatting (Raspberry Pi)
- Check memory allocation for ML models within platform constraints

Network Connectivity (WiFi-enabled platforms):

- Confirm network credentials and security protocols
- Check firewall settings for development tool access
- Verify that the network allows device-to-development machine communication

Troubleshooting and Support

Common Hardware Issues:

- **Device not recognized:** Ensure the USB cable supports data transfer, try different ports
- **Upload failures:** Check board selection and port configuration in the IDE
- **Power issues:** Verify adequate power supply, especially for camera operations
- **Memory errors:** Confirm model size fits within platform constraints

Software Setup Issues:

- **Library conflicts:** Use compatible versions specified in the setup guides
- **Compilation errors:** Verify all dependencies are installed correctly
- **Network connectivity:** Check firewall settings and network permissions

Platform-Specific Resources:

- **XIAOML Kit:** Seeed Studio Documentation

- XIAO ESP32S3 Series documentation
- **Arduino Nicla Vision:** Arduino Documentation
- **Grove Vision AI V2:** SenseCraft AI Platform
- **Raspberry Pi:** Official Documentation

Community Support:

- **GitHub Issues:** Report bugs and request features through the project repository
- **Discussion Forums:** Platform-specific communities on Arduino, Raspberry Pi, and Seeed Studio websites
- **Stack Overflow:** Tag questions with appropriate platform tags for community assistance

Ready for Laboratory Exercises

With your development environment configured and verified, you have established the foundational tools needed for embedded ML programming. The skills developed during environment setup—understanding toolchains, managing dependencies, and verifying system functionality—apply throughout all subsequent laboratory work.

Your configured environment now supports the entire development workflow, from algorithm implementation to hardware deployment and performance optimization. The Laboratory Overview offers exercise categories organized by complexity and learning objectives, designed to systematically build on these foundational capabilities.

Recommended starting sequence:

1. Begin with basic sensor exercises to verify hardware functionality
2. Progress to single-modality ML applications (image or audio)
3. Advance to multi-modal and optimization exercises

Each laboratory exercise includes detailed implementation procedures, expected performance benchmarks, and troubleshooting guidance specific to the project requirements. The development environment you have established provides the foundation for exploring the complete spectrum of embedded ML applications and optimization techniques.

I

KEY:ARDUINO

Part I

II

ARDUINO NICLA VISION

Part II

Overview

These labs provide a unique opportunity to gain practical experience with machine learning (ML) systems. Unlike working with large models requiring data center-scale resources, these exercises allow you to directly interact with hardware and software using TinyML. This hands-on approach gives you a tangible understanding of the challenges and opportunities in deploying AI, albeit at a tiny scale. However, the principles are largely the same as what you would encounter when working with larger systems.

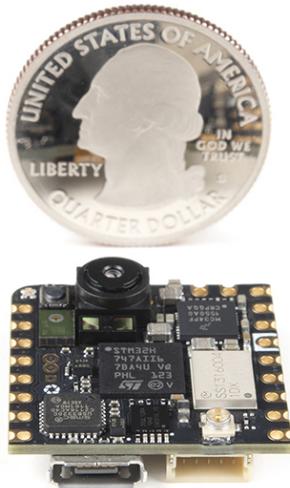


Figure 1.7: Nicla Vision. Source: Arduino.

Where to Buy

The Arduino Nicla Vision is available from the official Arduino Store:

- Arduino Store (~\$95)

Pre-requisites

- **Nicla Vision Board:** Ensure you have the Nicla Vision board.
- **USB Cable:** For connecting the board to your computer.
- **Network:** With internet access for downloading necessary software.

Setup

- Setup Nicla Vision

Exercises

Modality	Task	Description	Link
Vision	Image Classification	Learn to classify images	Link
Vision	Object Detection	Implement object detection	Link
Sound	Keyword Spotting	Explore voice recognition systems	Link
IMU	Motion Classification and Anomaly Detection	Classify motion data and detect anomalies	Link

Setup

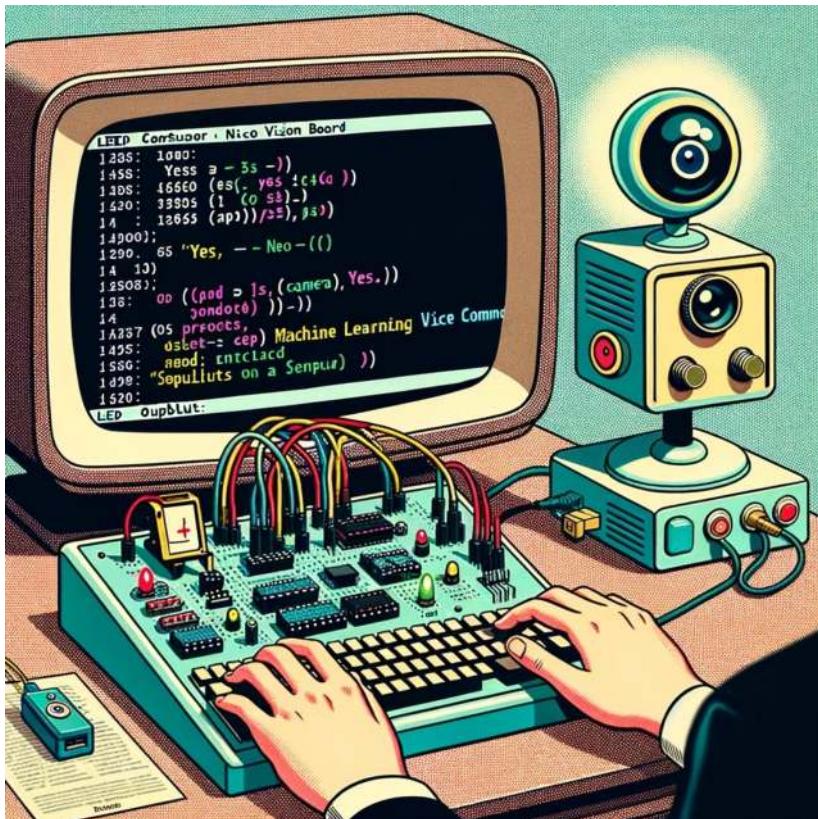
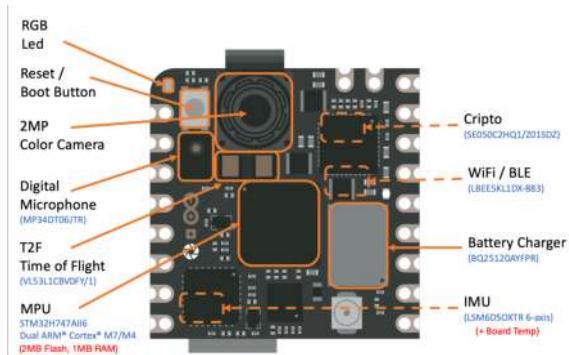


Figure 1.8: DALL-E 3 Prompt: Illustration reminiscent of a 1950s cartoon where the Arduino NICLA VISION board, equipped with various sensors including a camera, is the focal point on an old-fashioned desk. In the background, a computer screen with rounded edges displays the Arduino IDE. The code is related 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.

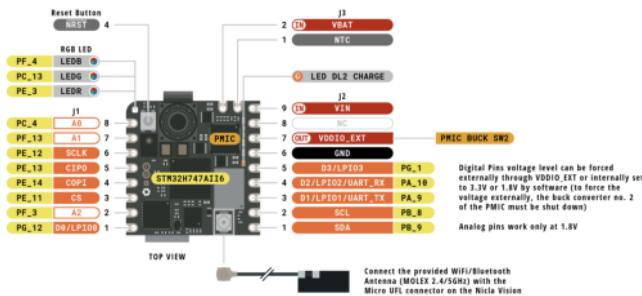


Hardware

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.



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):

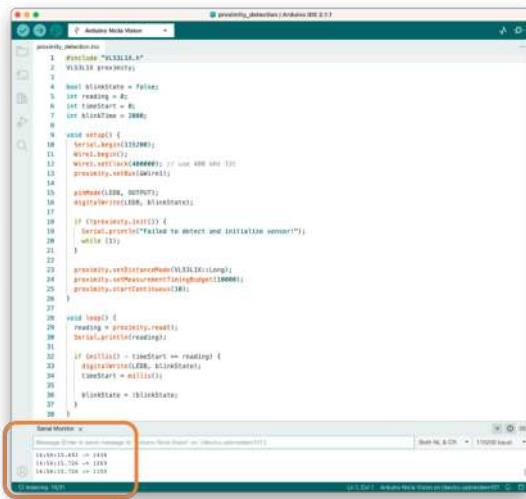


Testing the ToF (Time of Flight) Sensor

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`:



The screenshot shows the Arduino IDE interface. The main window displays the code for 'proximity_detections' in an Arduino sketch. The code initializes pins, sets up serial communication at 115200 bps, and initializes proximity sensor pins. It includes a setup() function to initialize the serial port and proximity sensor, and a loop() function that reads the proximity sensor value every 100ms, converts it to centimeters, and prints the result to the Serial Monitor. If the distance is less than 10 cm, it prints a warning message. The Serial Monitor window below shows the output of the code, with a red box highlighting the first few lines of data.

```
proximity_detections
1 #include "HC-SR04.h"
2
3 #define trigPin 7
4 #define echoPin 6
5 bool blinkState = false;
6 int timeStart = 0;
7 int silenceTime = 2000;
8
9 void setup() {
10   Serial.begin(115200);
11   Wire.begin();
12   Wire.setClock(400000); // use 400 kHz (2)
13   proximity.setDutyCycle(100);
14
15   pinMode(trigPin, OUTPUT);
16   digitalWrite(trigPin, HIGH);
17
18   if (proximity.init() == 0) {
19     Serial.println("Failed to detect and initialize sensor!");
20     while (1);
21   }
22
23   proximity.setDistanceMin(100);
24   proximity.setDistanceMax(1000);
25   proximity.setEchoTimeout(1000);
26 }
27
28 void loop() {
29   reading = proximity.read();
30   Serial.println(reading);
31
32   if (reading <= 10) {
33     digitalWrite(trigPin, LOW);
34     timeStart = millis();
35
36     blinkState = !blinkState;
37   }
38 }
```

Serial Monitor

13:49:12.492 => 1404
13:49:12.503 => 1403
13:49:12.708 => 1393

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

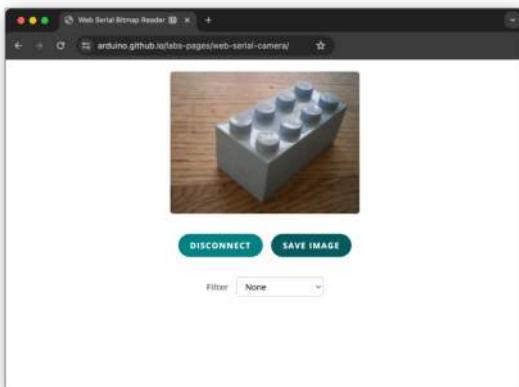
We can also test the camera using, for example, the code provided on Examples > Camera > CameraCaptureRawBytes. We cannot see the image directly, but we can get the raw image data generated by the camera.

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.

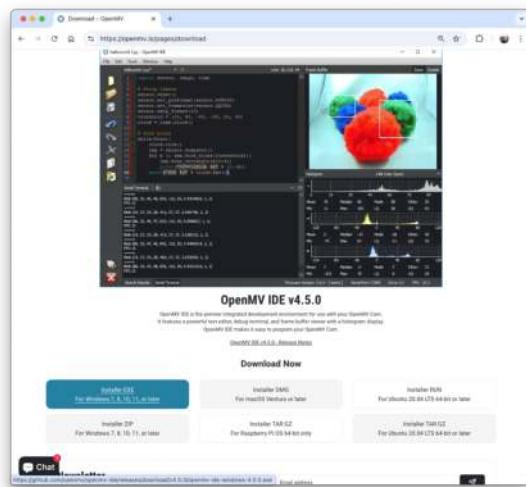
The **web application** for displaying the camera image can be accessed here. We may also look at [this tutorial], which explains the setup in more detail.



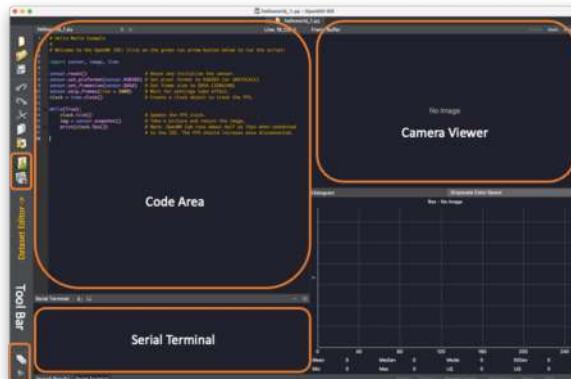
Installing the OpenMV IDE

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 > helloworld.py**



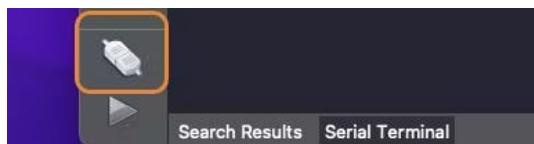
Any messages sent through a serial connection (using `print()` or error messages) will be displayed on the **Serial Terminal** during run time. The image captured by a camera will be displayed in the **Camera Viewer** Area (or Frame Buffer) and in the Histogram area, immediately below the Camera Viewer.

Updating the Bootloader

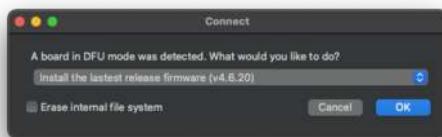
Before connecting the Nicla to the OpenMV IDE, ensure you have the latest bootloader version. Go to your Arduino IDE, select the Nicla board, and open the sketch on `Examples > STM_32H747_System STM32H747_manageBootloader`. Upload the code to your board. The Serial Monitor will guide you.

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].

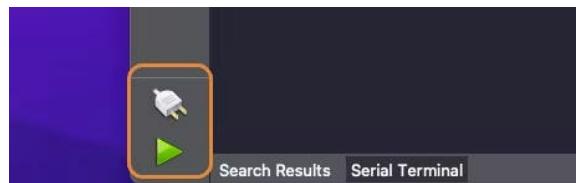
Nicla's green LED will start flashing while the OpenMV firmware is uploaded to the board, and a terminal window will then open, showing the flashing progress.



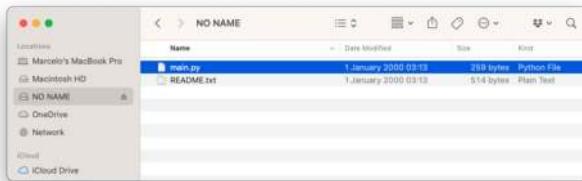
Wait until the green LED stops flashing and fading. When the process ends, you will see a message saying, “DFU firmware update complete!”. Press [OK].



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...).

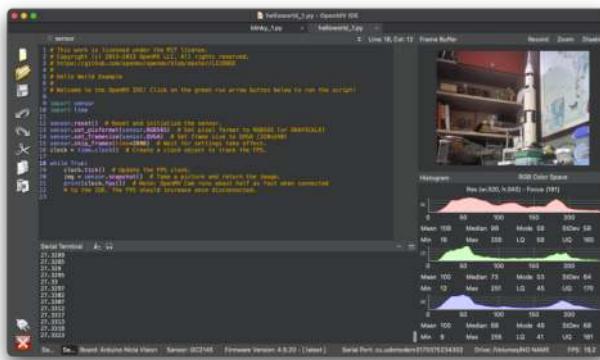
```
main.py
1 # main.py -- put your code here!
2 import pyb, time
3 led = pyb.LED(3) <-- Blue LED *
4 usb = pyb.USB_VCP()
5 while (usb.isconnected() == False):
6     led.on()
7     time.sleep_ms(150)
8     led.off()
9     time.sleep_ms(100)
10    led.on()
11    time.sleep_ms(150)
12    led.off()
13    time.sleep_ms(600)
14
* LED(1) : Red
  LED(2) : Green
  LED(3) : Blue
```

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

Testing the Camera

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

When clicking the green play button, the MicroPython script (`helloworld.py`) on the Code Area will be uploaded and run on the Nicla Vision. On-Camera Viewer, you will start to see the video streaming. The Serial Monitor will show us the FPS (Frames per second), which should be around 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.

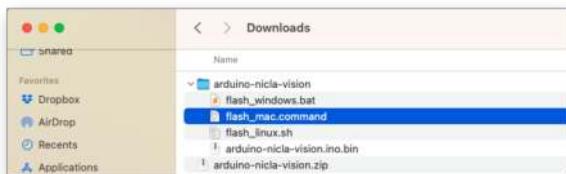
In the GitHub, You can find other Python scripts. Try to test the onboard sensors.

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:

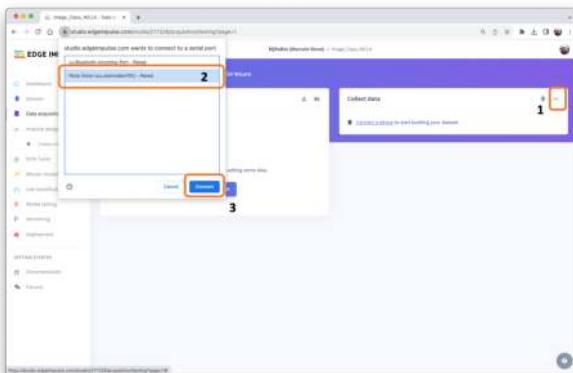
- Download the Arduino CLI for your specific computer architecture (OS)
- Download the most updated EI Firmware.
- Unzip both files and place all the files in the same folder.
- Put the Nicla-Vision on Boot Mode, pressing the reset button twice.
- Run the uploader (EI FW) corresponding to your OS:



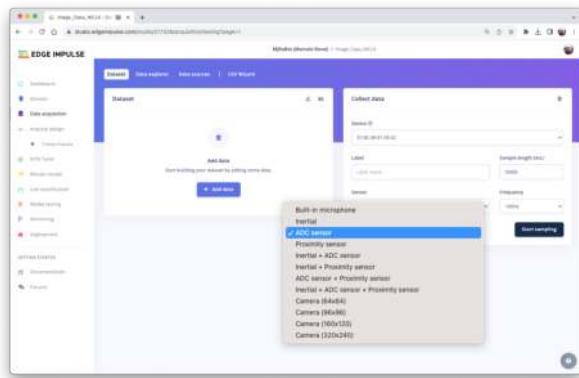
- Executing the specific batch code for your OS will upload the binary *arduino-nicla-vision.bin* to your board.

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

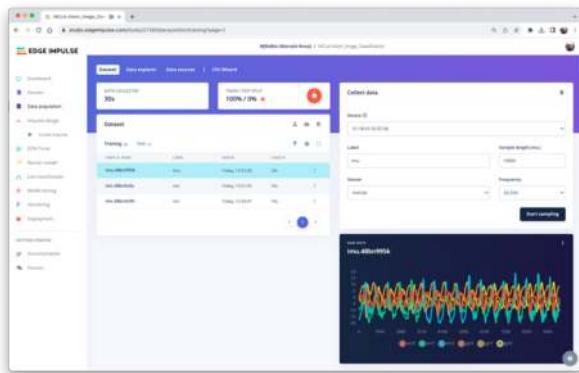
Go to your project on the Studio, and on the Data Acquisition tab, select WebUSB (1). A window will pop up; choose the option that shows that the Nicla is paired (2) and press [Connect] (3).



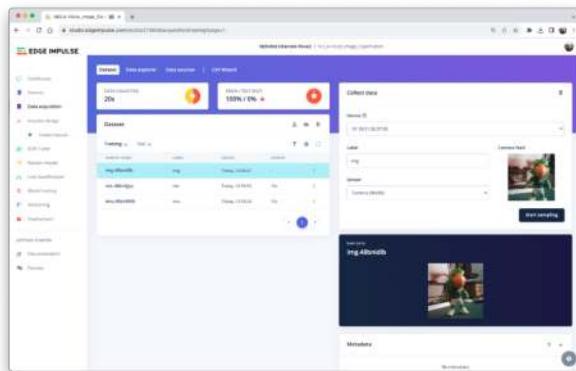
You can choose which sensor data to pick in the Collect Data section on the Data Acquisition tab.



For example. IMU data (inertial):



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)

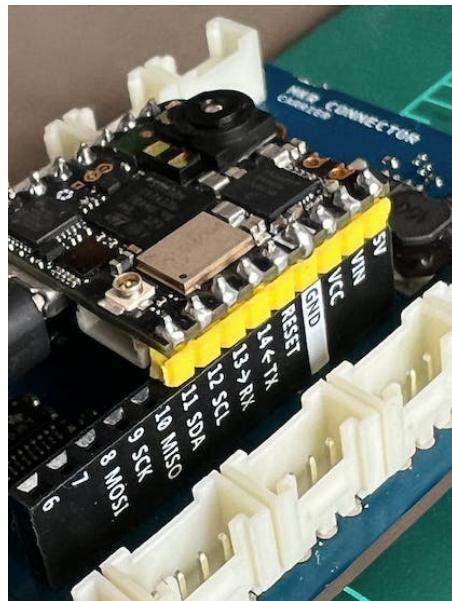
A last item to explore is that sometimes, during prototyping, it is essential to experiment with external sensors and devices. An excellent expansion to the Nicla is the Arduino MKR Connector Carrier (Grove compatible).

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.

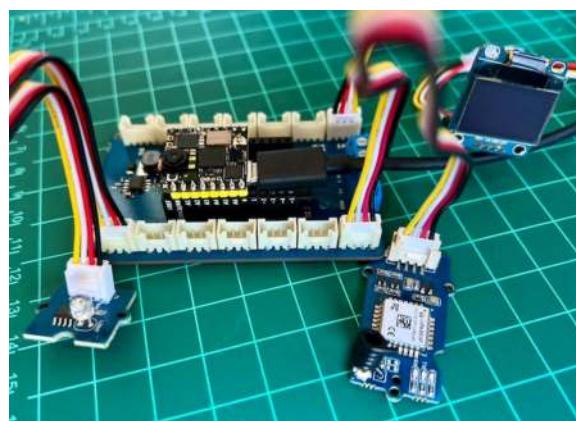
Note that all 17 Nicla Vision pins will be connected to the Shield Groves, but some Grove connections remain disconnected.



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

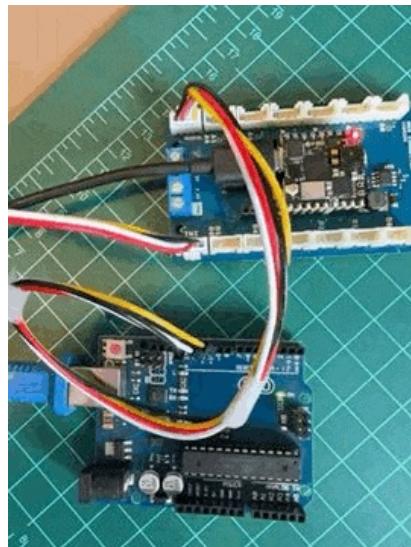
import time
from pyb import UART
from pyb import LED

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.

Summary

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

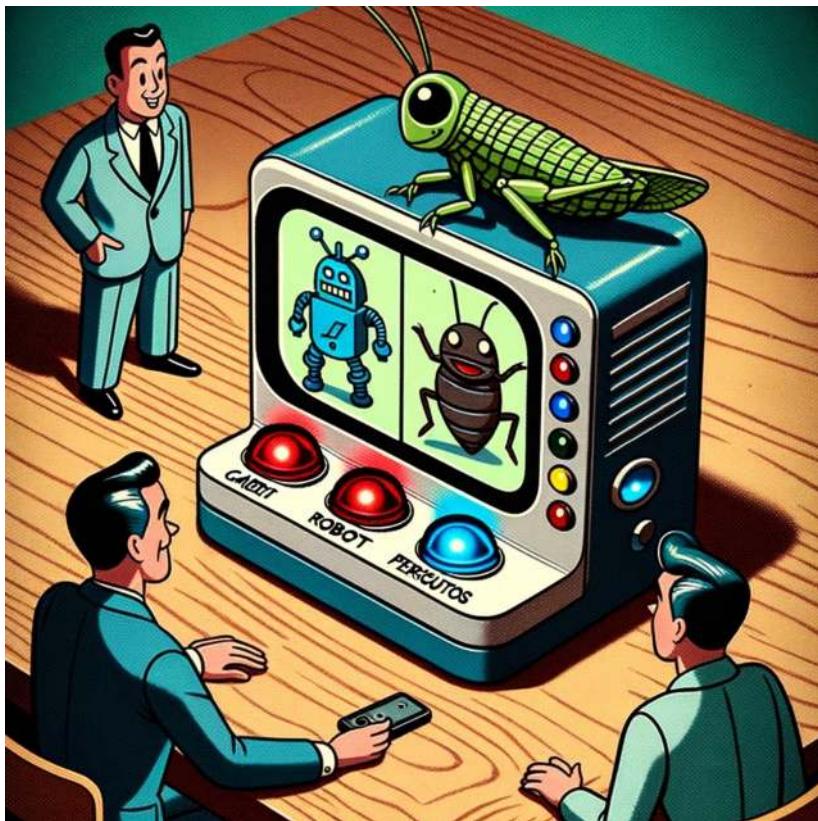


Figure 1.9: DALL-E 3 Prompt: Cartoon in a 1950s style featuring a compact electronic device with a camera module placed on a wooden table. The screen displays blue robots on one side and green periquitos on the other. LED lights on the device indicate classifications, while characters in retro clothing 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.

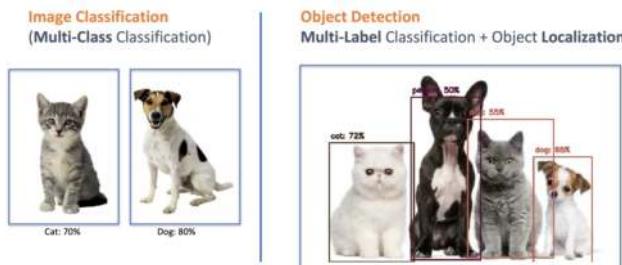
We’ll employ techniques like quantization and pruning to reduce the computational load. By the end of this tutorial, you’ll have a working prototype capable of classifying images in real-time, all running on

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

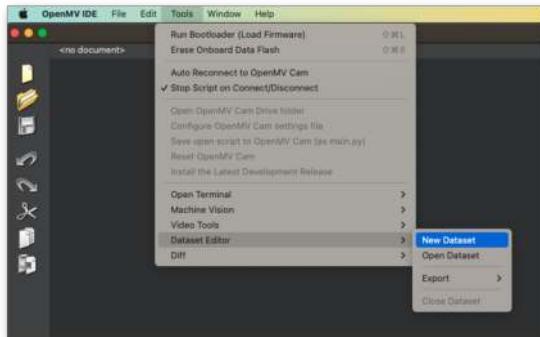
Once we have defined our Machine Learning project goal, the next and most crucial step is collecting the dataset. For image capturing, we can use:

- 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.



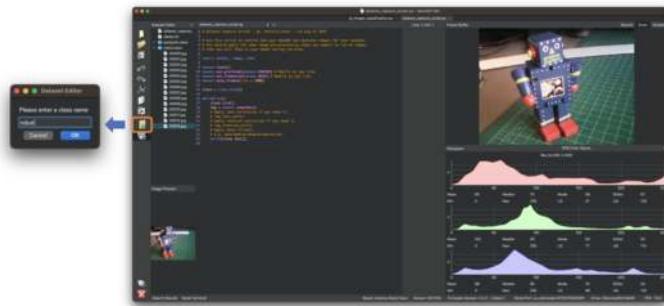
Using the upper icon (1), enter with the first class name, for example, “periquito”:



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.

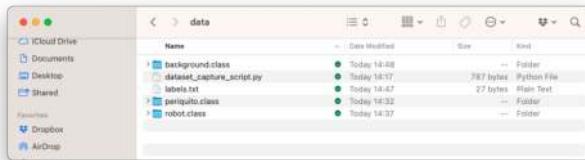


We suggest around 50 to 60 images from each category. Try to capture different angles, backgrounds, and light conditions.

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.

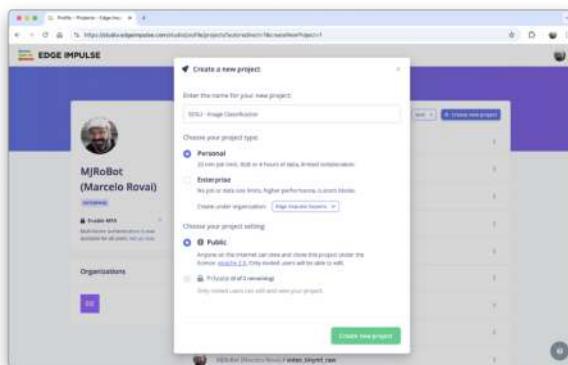
We will end up with a dataset on our computer that contains three classes: *periquito*, *robot*, and *background*.



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

Training the model with Edge Impulse Studio

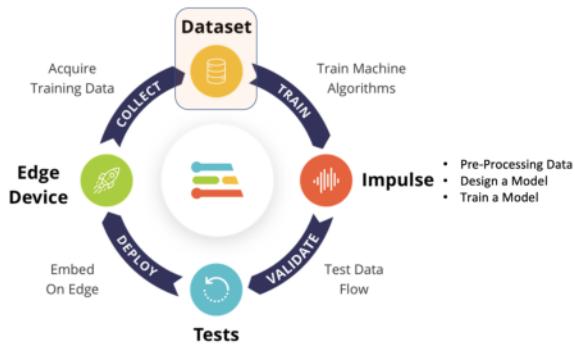
We will use the Edge Impulse Studio to train our model. Enter the account credentials and create a new project:



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

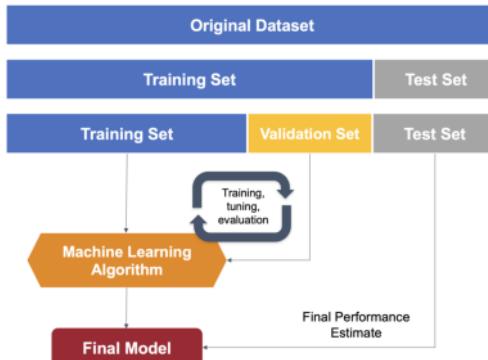
Dataset

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).

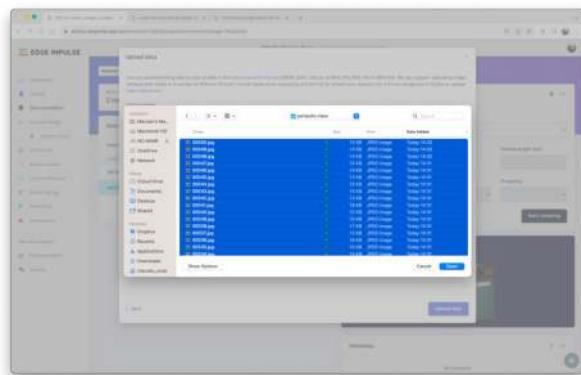


Regarding the Dataset, it is essential to point out that our Original Dataset, captured with the OpenMV IDE, will be split into *Training*, *Validation*, and *Test*. The Test Set will be spared from the beginning and reserved for use only in the Test phase after training. The Validation Set will be used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

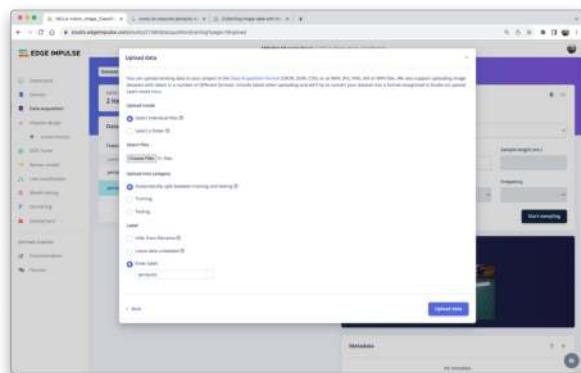
The EI Studio will take a percentage of training data to be used for validation



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



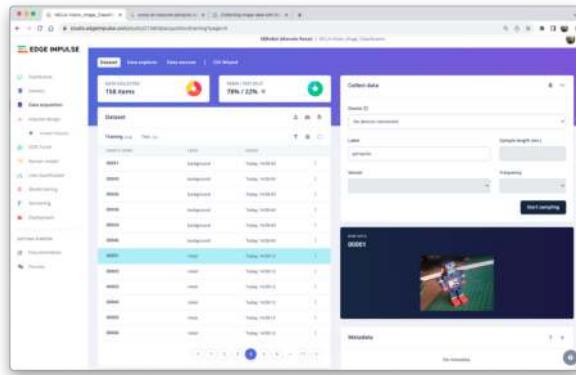
Leave to the Studio the splitting of the original dataset into *train* and *test* and choose the label about that specific data:



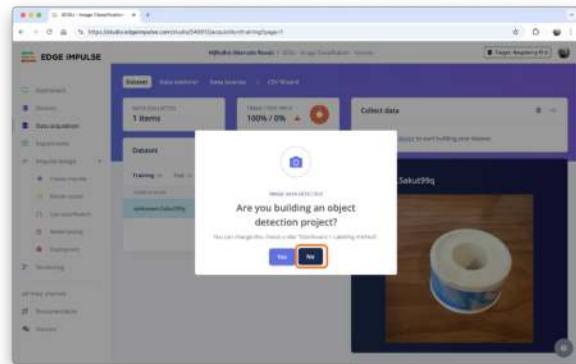
Repeat the procedure for all three classes.

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

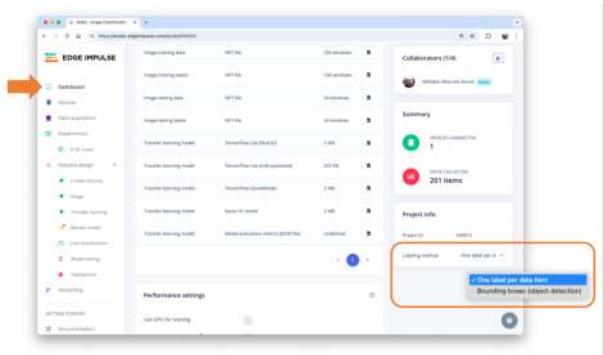
At the end, you should see your “raw data” in the Studio:



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):



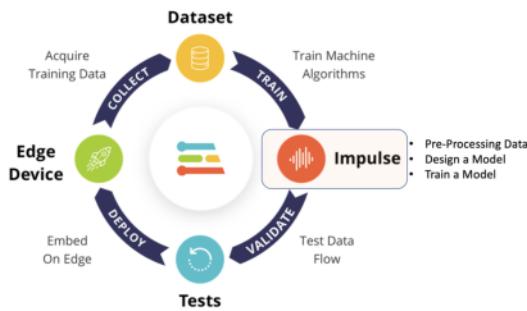
Optionally, the Studio allows us to explore the data, showing a complete view of all the data in the project. We can clear, inspect, or change labels by clicking on individual data items. In our case, the data seems OK.



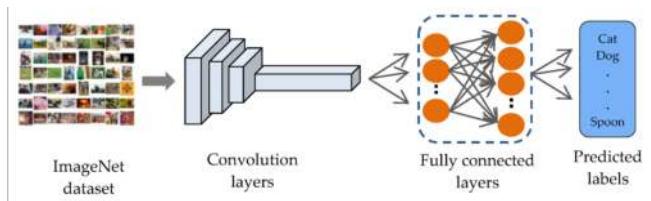
The Impulse Design

In this phase, we should define how to:

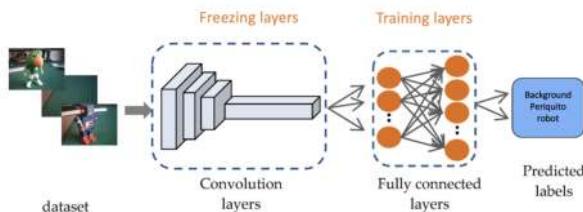
- Pre-process our data, which consists of resizing the individual images and determining the color depth to use (be it RGB or Grayscale) and
- Specify a Model, in this case, it will be the Transfer Learning (Images) to fine-tune a pre-trained MobileNet V2 image classification model on our data. This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

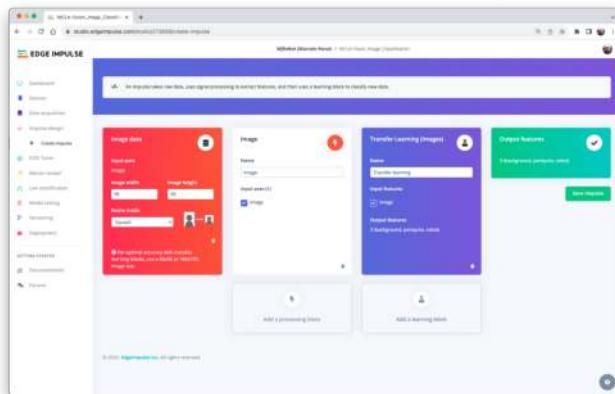
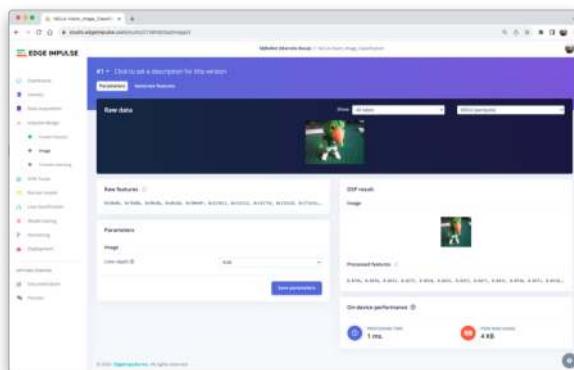
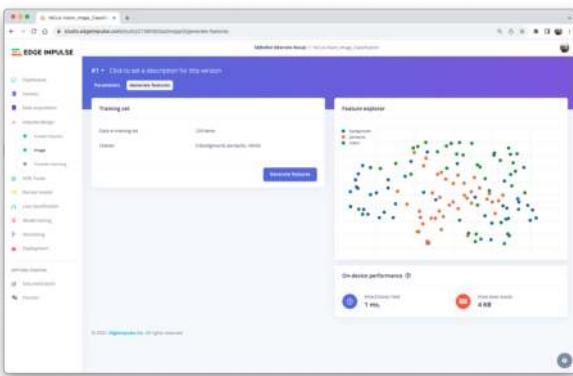


Image Pre-Processing

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 introduce 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).

MobileNetV1 96x96 0.1
Uses around 53.2K RAM and 101K ROM with default settings and optimizations. Works best with 96x96 input size. Supports both RGB and grayscale.

Model

MobileNetV2 96x96 0.35
Uses around 296.8K RAM and 575.2K ROM with default settings and optimizations. Works best with 96x96 input size. Supports both RGB and grayscale.

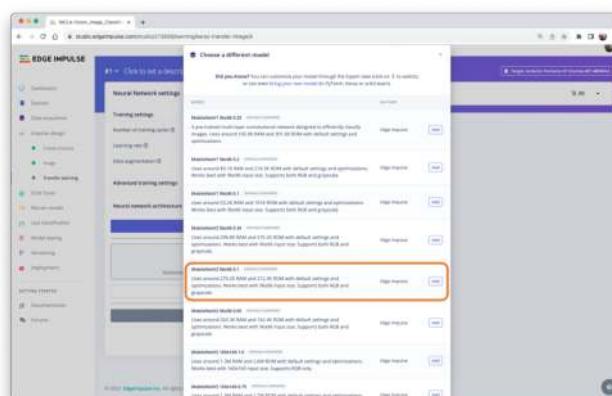
Image Size

MobileNetV2 96x96 0.1
Uses around 270.2K RAM and 212.3K ROM with default settings and optimizations. Works best with 96x96 input size. Supports both RGB and grayscale.

Alpha

MobileNetV2 96x96 0.05
Uses around 265.3K RAM and 162.4K ROM with default settings and optimizations. Works best with 96x96 input size. Supports both RGB and grayscale.

We will use **MobileNetV2 96x96 0.1 (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)

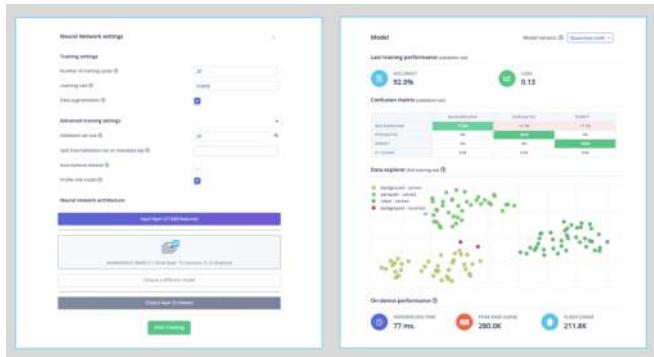
    # Increase the image size, then randomly crop it down to
    # the original dimensions
    resize_factor = random.uniform(1, 1.2)
    new_height = math.floor(resize_factor * INPUT_SHAPE[0])
    new_width = math.floor(resize_factor * INPUT_SHAPE[1])
    image = tf.image.resize_with_crop_or_pad(image, 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
```

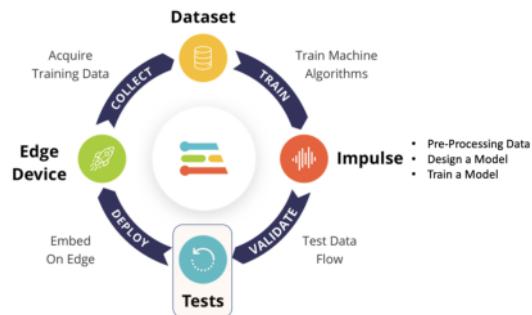
Exposure to these variations during training can help prevent your model from taking shortcuts by “memorizing” superficial clues in your training data, meaning it may better reflect the deep underlying patterns in your dataset.

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

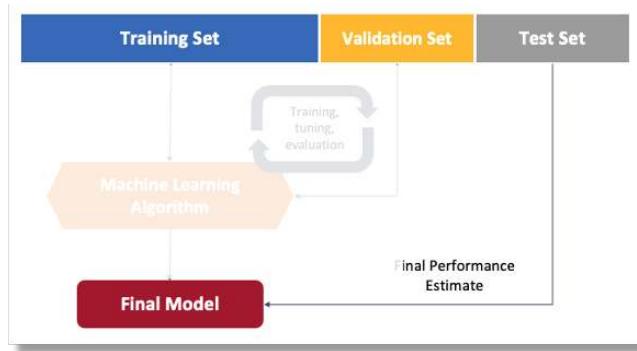


The result is excellent, with 77 ms of latency (estimated), which should result in around 13 fps (frames per second) during inference.

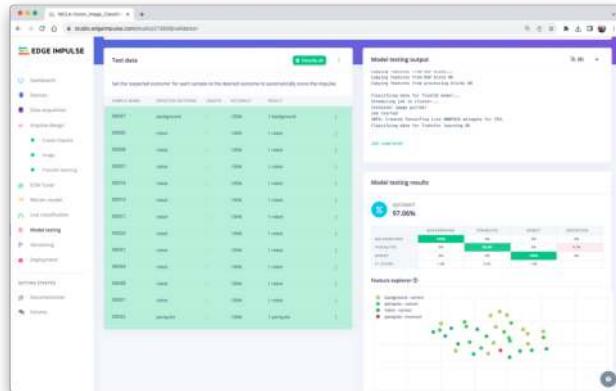
Model Testing



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

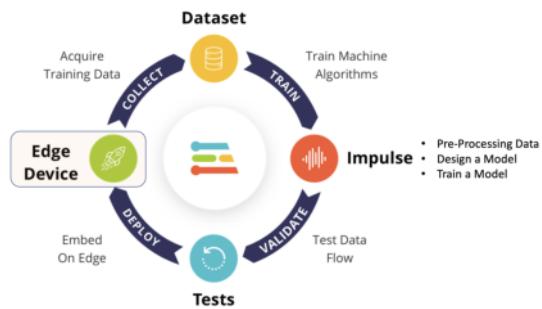


The result is, again, excellent.



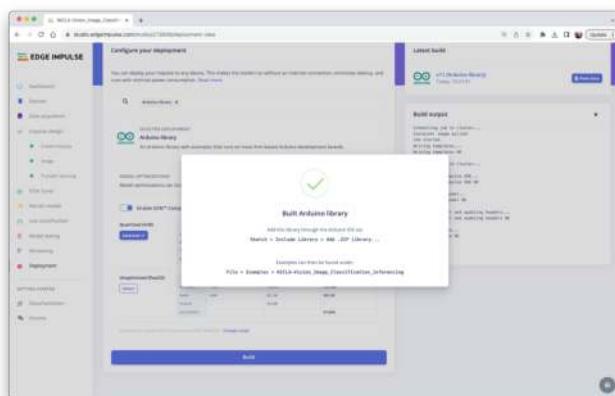
Deploying the model

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

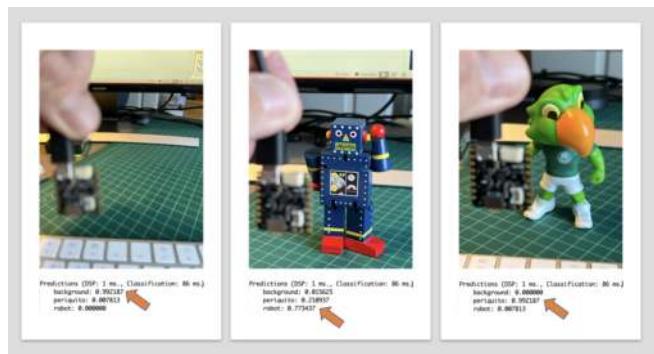
First, Let's deploy it as an Arduino Library:



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.

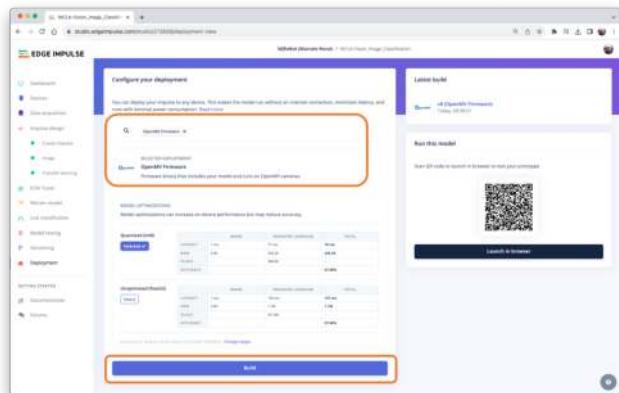


Here is a short video showing the inference results: <https://youtu.be/bZPZZJblU-o>

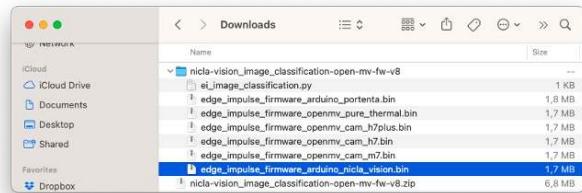
OpenMV

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.

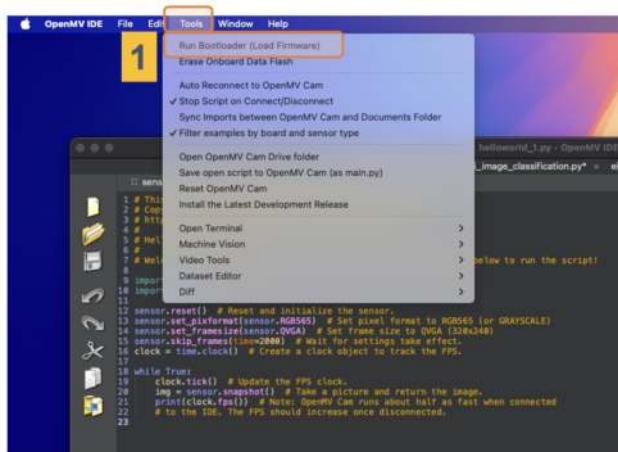
Select OpenMV Firmware on the Deploy Tab and press [Build].



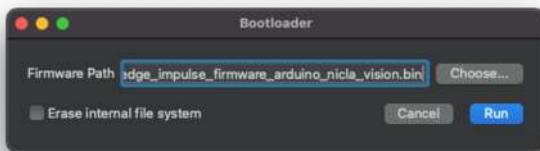
On the computer, we will find a ZIP file. Open it:



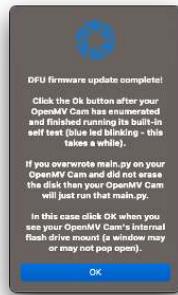
Use the Bootloader tool on the OpenMV IDE to load the FW on your board (1):



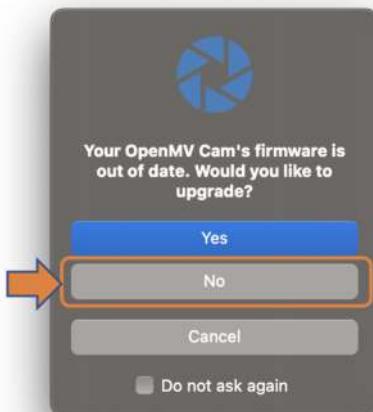
Select the appropriate file (.bin for Nicla-Vision):



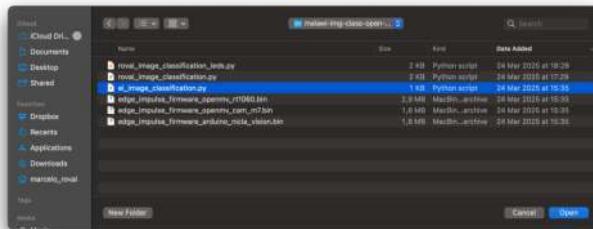
After the download is finished, press OK:



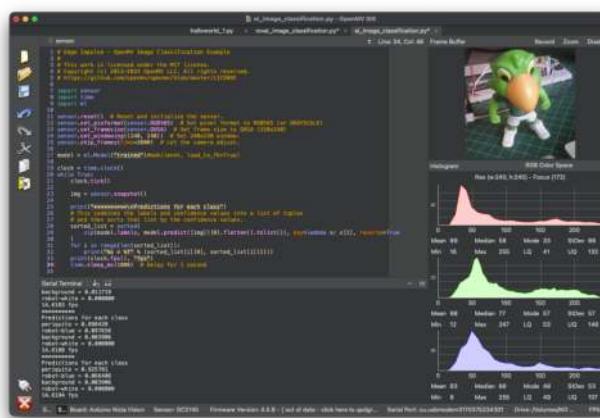
If a message says that the FW is outdated, **DO NOT UPGRADE**. Select [NO].



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 difficult 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.
# Set pixel format to RGB565 (or GRayscale)
sensor.set_pixformat(sensor.RGB565)
# Set frame size to QVGA (320x240)
sensor.set_framesize(sensor.QVGA)
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]

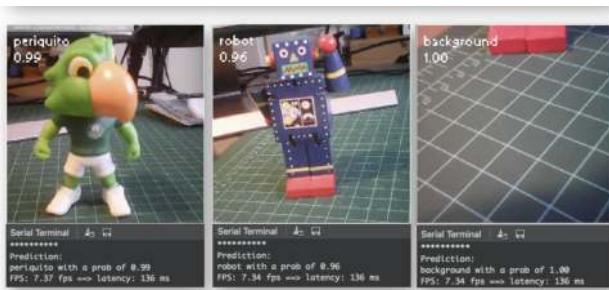
    if max_val < 0.5:
        max_lbl = 'uncertain'

    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:

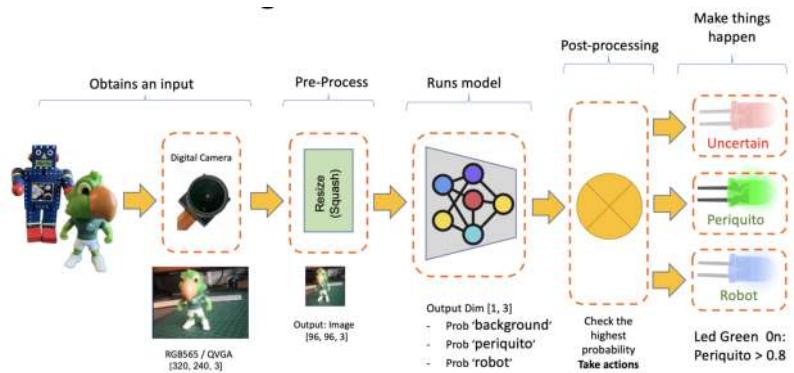


Note that the latency (136 ms) is almost double of what we got directly with the Arduino IDE. This is because we are using the IDE as an interface and also the time to wait for the camera to be ready. If we start the clock just before the inference, the latency should drop to around 70 ms.

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.
# Set pixel format to RGB565 (or GRayscale)
sensor.set_pixformat(sensor.RGB565)
# Set frame size to QVGA (320x240)
sensor.set_framesize(sensor.QVGA)
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
    )

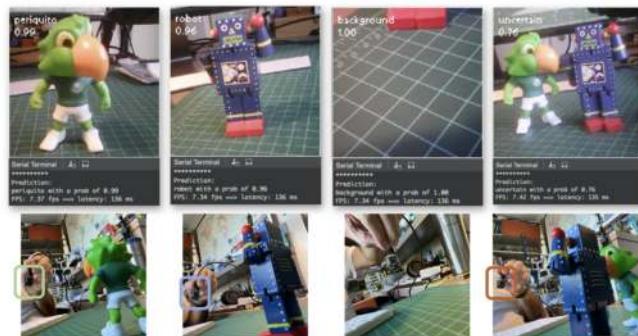
    setLEDs(max_lbl)
    time.sleep_ms(200) # Delay for .2 second

```

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

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

Here is the result:



In more detail



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:



Summary

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

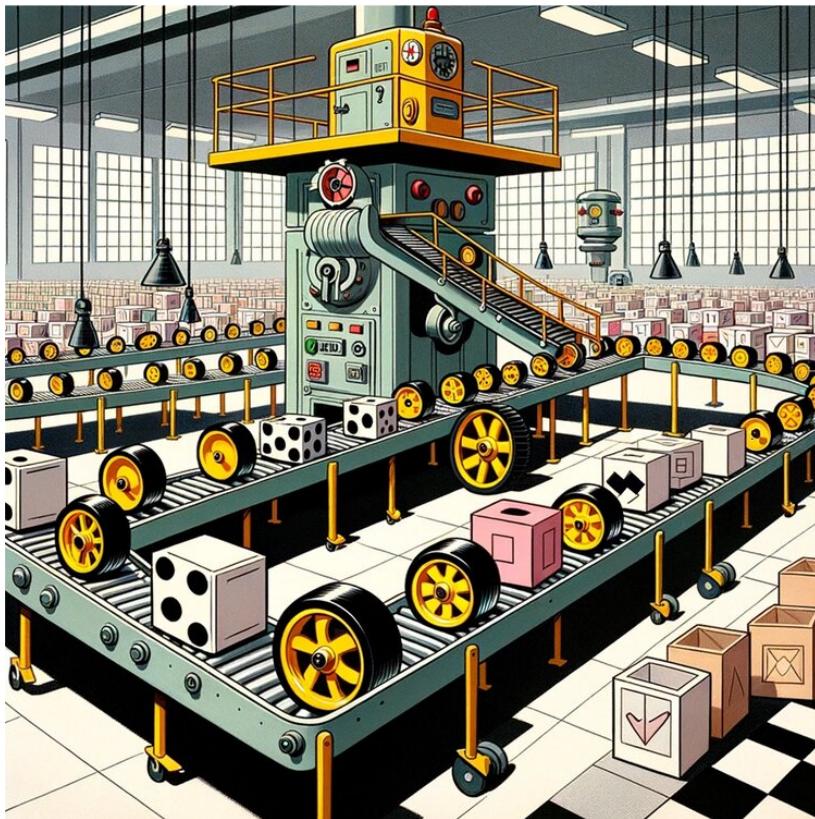


Figure 1.10: DALL-E 3 Prompt: Cartoon in the style of the 1940s or 1950s showcasing a spacious industrial warehouse interior. A conveyor belt is prominently featured, carrying a mixture of toy wheels and boxes. The wheels are distinguishable with their bright yellow centers and black tires. The boxes 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.

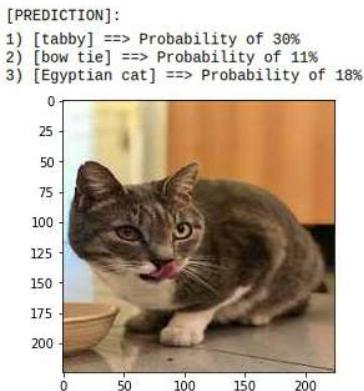
Overview

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

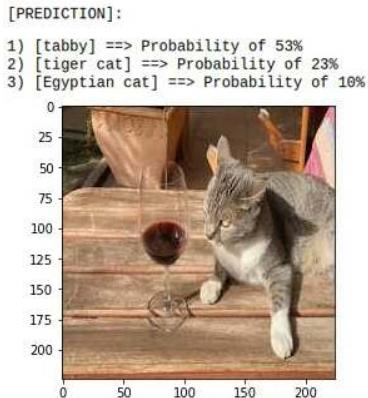


Object Detection versus Image Classification

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:



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:



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



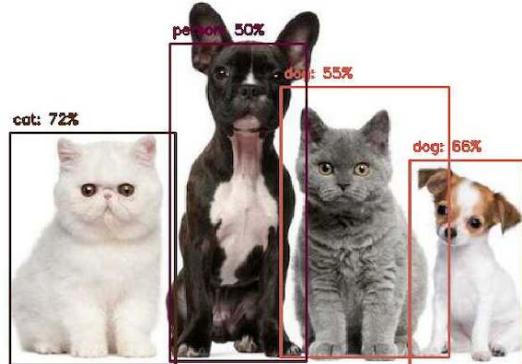
The model identifies the above image utterly wrong as an “ashcan,” possibly due to the color tonalities.

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

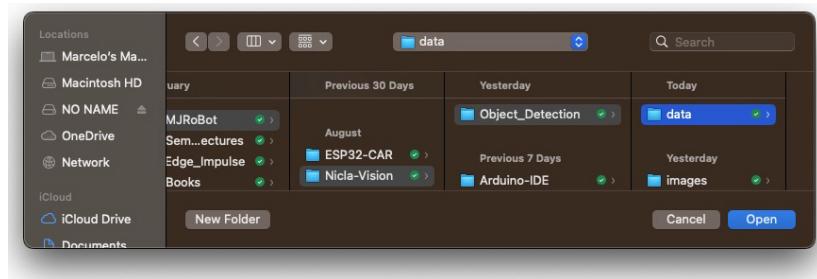
For image capturing, we can use:

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

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

Collecting Dataset with OpenMV IDE

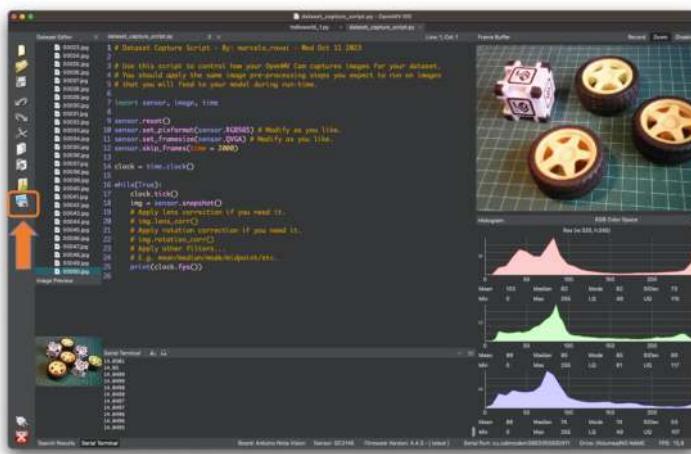
First, we create a folder on the computer where the data will be saved, for example, “data.” Next, on the OpenMV IDE, we go to Tools > Dataset Editor and select New Dataset to start the dataset collection:



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.

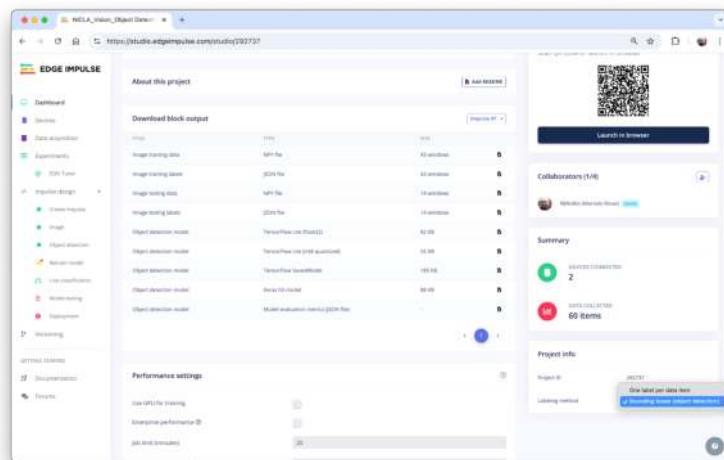
Edge Impulse Studio

Setup the project

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

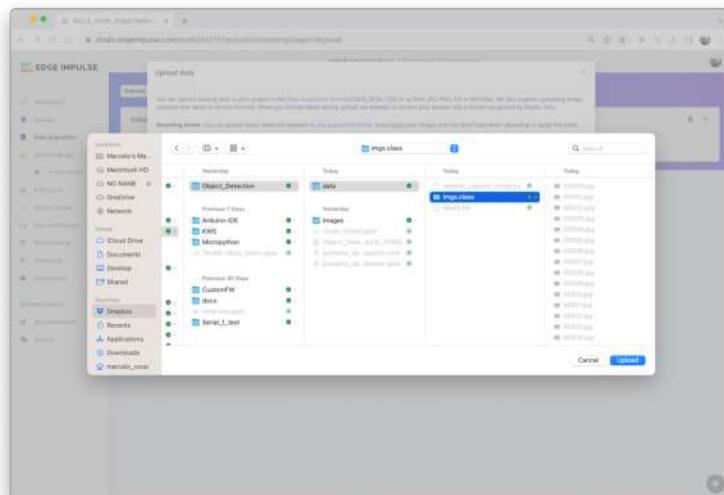
Here, you can clone the project developed for this hands-on: **NICLA_Vision_Object_Detection**.

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)**.

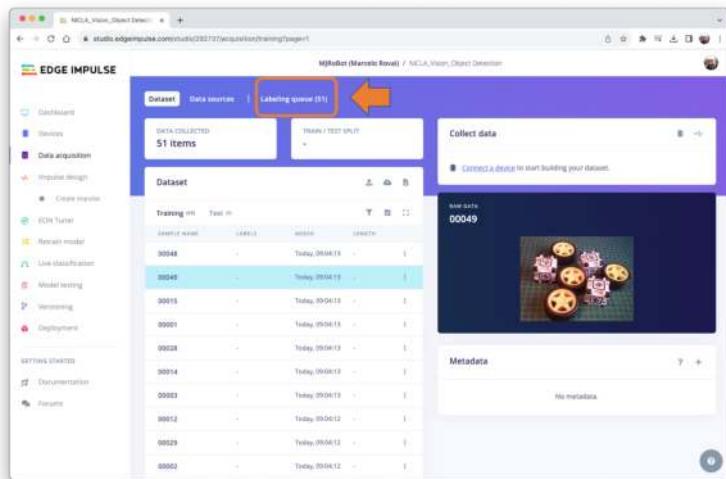


Uploading the unlabeled data

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

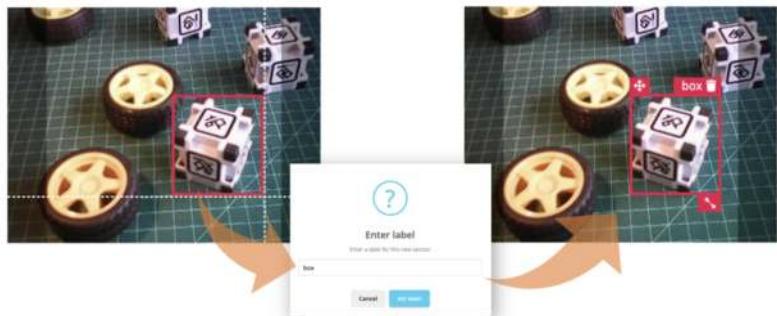
Edge Impulse launched an auto-labeling feature for Enterprise customers, easing labeling tasks in object detection projects.

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).

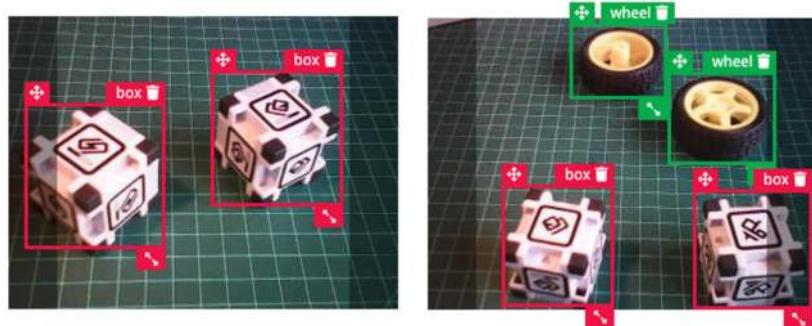
If you already have a labeled dataset containing bounding boxes, import your data using the EI uploader.

Labeling the Dataset

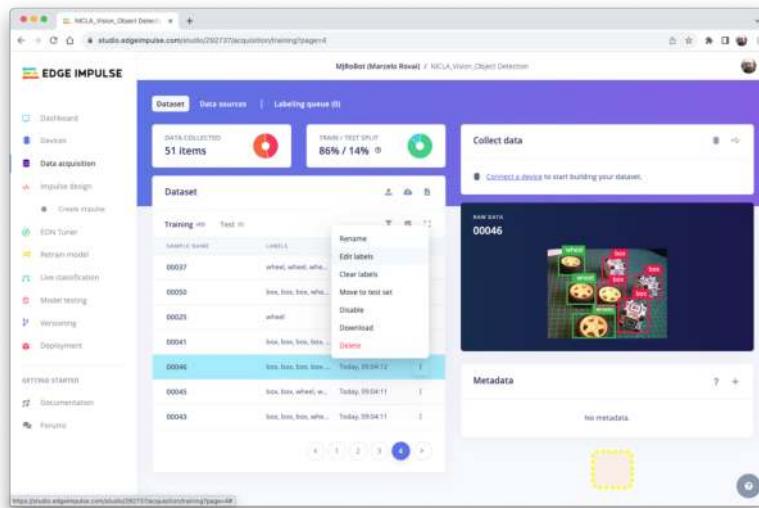
Starting with the first image of your unlabeled data, use your mouse to drag a box around an object to add a label. Then click **Save labels** to advance to the next item.



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:



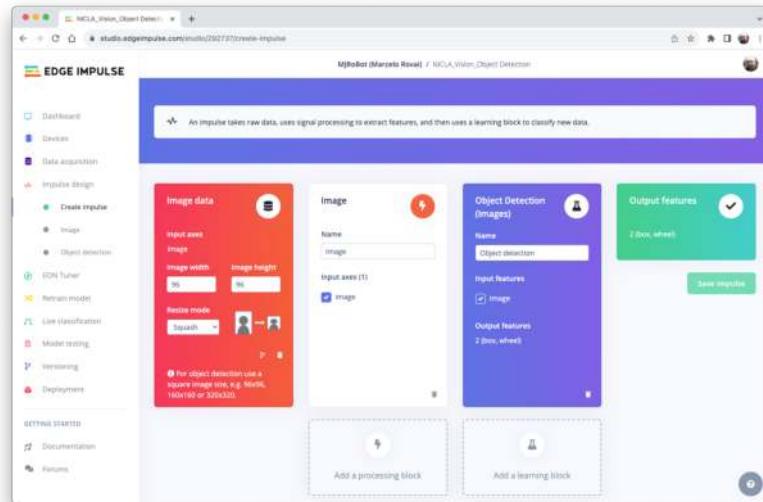
We will be guided to replace the wrong label and correct the dataset.



The Impulse Design

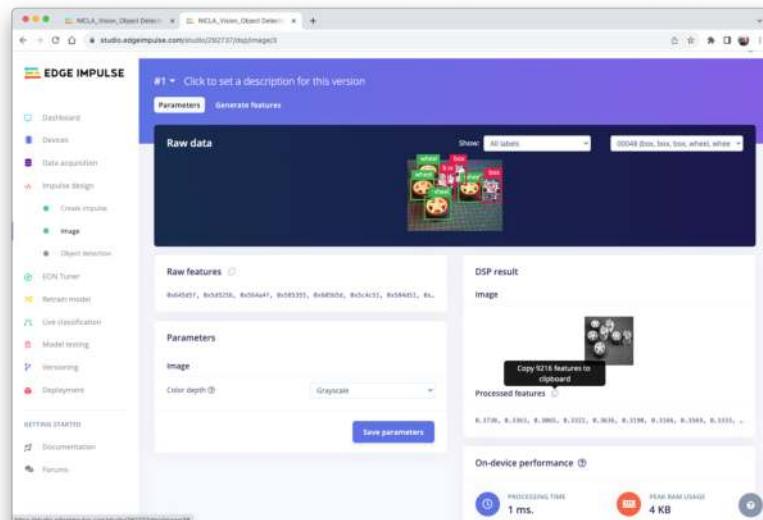
In this phase, we should define how to:

- **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.”

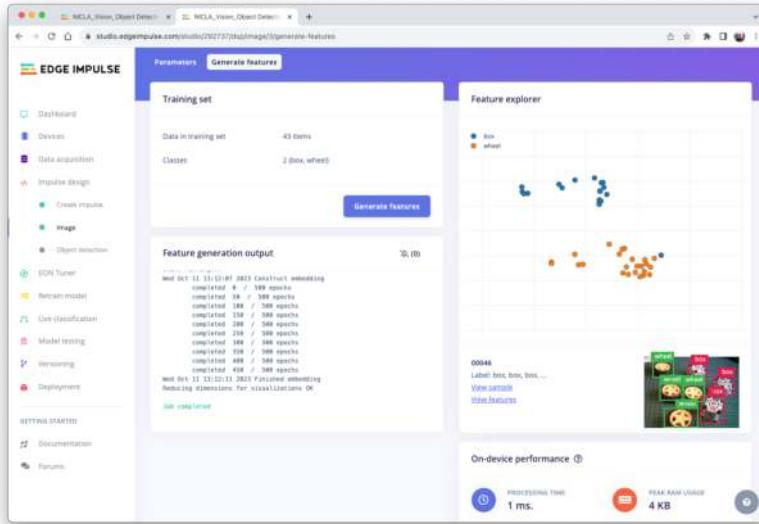


Preprocessing all dataset

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.

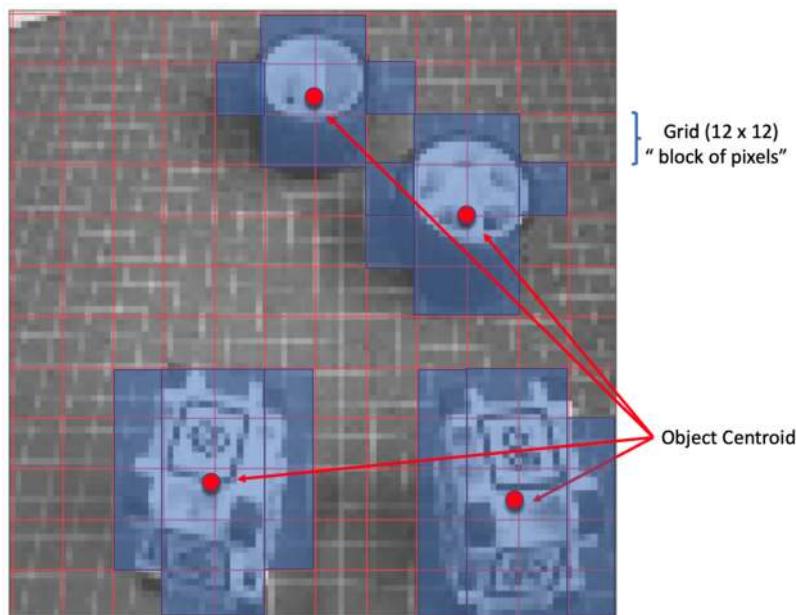
Model Design, Training, and Test

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*).

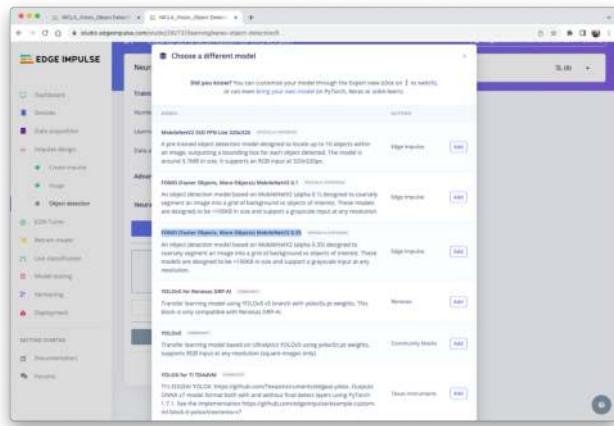
FOMO is an innovative machine learning model for object detection, which can use up to 30 times less energy and memory than traditional models like Mobilenet SSD and YOLOv5. FOMO can operate on microcontrollers with less than 200 KB of RAM. The main reason this is possible is that while other models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



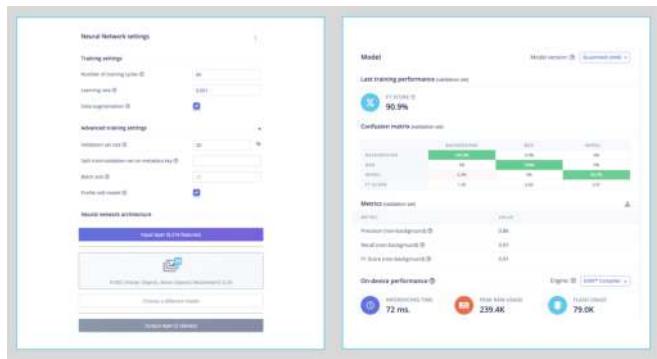
Regarding the training hyper-parameters, the model will be trained with:

- 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*).



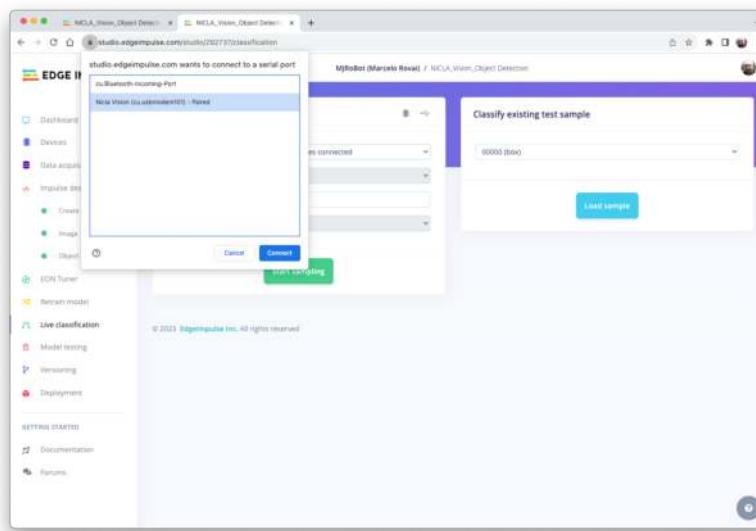
In object detection tasks, accuracy is generally not the primary evaluation metric. Object detection involves classifying objects and providing bounding boxes around them, making it a more complex problem than simple classification. The issue is that we do not have the bounding box, only the centroids.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

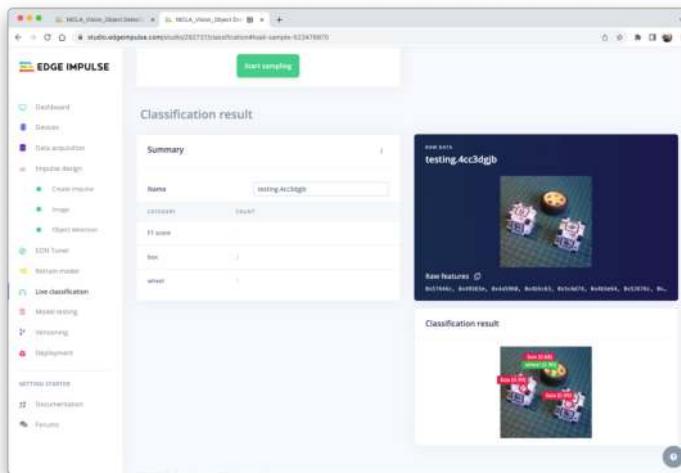
Since Edge Impulse officially supports the Nicla Vision, let's connect it to the Studio. For that, follow the steps:

- 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:



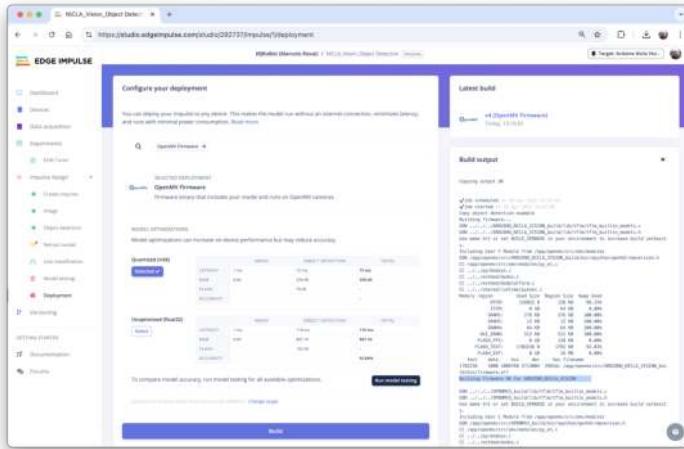
Once connected, you can use the Nicla to capture actual images to be tested by the trained model on Edge Impulse Studio.



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

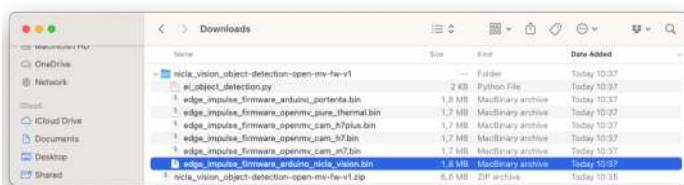
Select OpenMV Firmware on the Deploy Tab and press [Build].



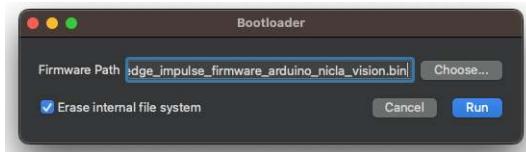
When you try to connect the Nicla with the OpenMV IDE again, it will try to update its FW. Choose the option **Load a specific firmware** instead. Or go to 'Tools > Runs Bootloader (Load Firmware)'.

✓ Install the lastest release firmware (v4.4.3)
Load a specific firmware
 Just erase the interal file system

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.
# Set pixel format (RGB565 or GRayscale)
sensor.set_pixformat(sensor.RGB565)
# Set frame size to QVGA (320x240)
sensor.set_framesize(sensor.QVGA)
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 trained object. So, we will
# get those output images and then run find_blobs() on them
# to extract the centroids. We will also run get_stats() on
# the detected blobs to determine their score.
# The Non-Max-Suppression (NMS) object then filters out
# overlapping detections and maps their position in the
# output image back to the original input image. The
# function then returns a list per class which each contain
# a list of (rect, score) tuples representing the detected
# 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()

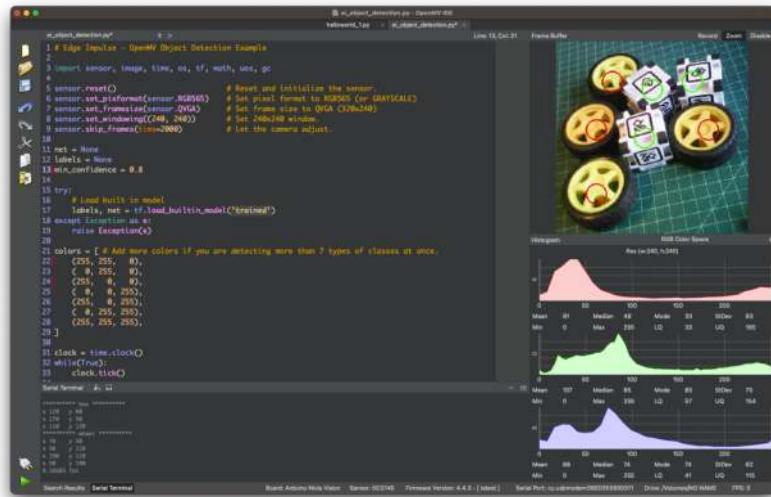
    img = sensor.snapshot()

    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>

Summary

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

Keyword Spotting (KWS)



Figure 1.11: 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.

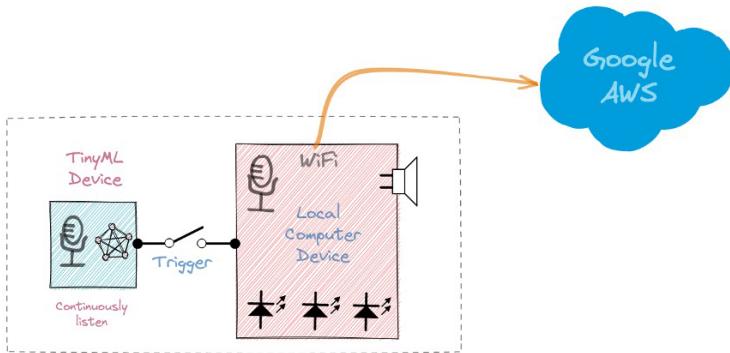
Our model will be designed to recognize keywords that can trigger device wake-up or specific actions, bringing them to life with voice-activated commands.

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).

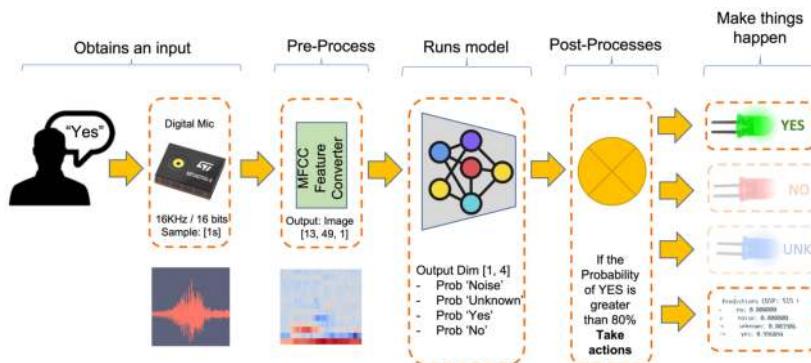
https://youtu.be/e_OPgcnsyvM

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):



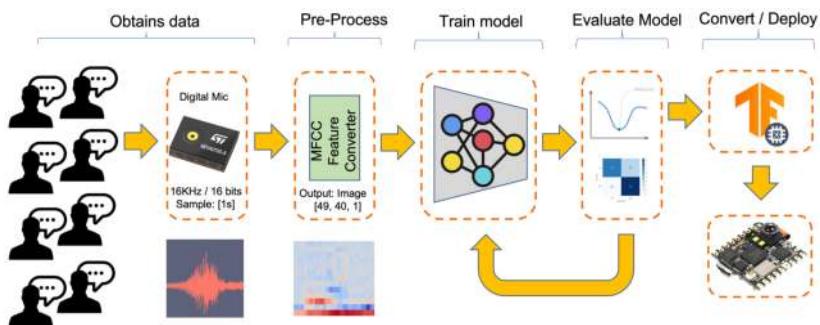
Our KWS application will recognize four classes of sound:

- **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

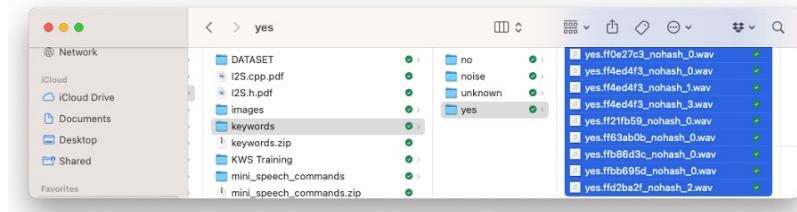
The critical component of any Machine Learning Workflow is the **dataset**. Once we have decided on specific keywords, in our case (*YES* and *NO*), we can take advantage of the dataset developed by Pete Warden, “Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition.” This dataset has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

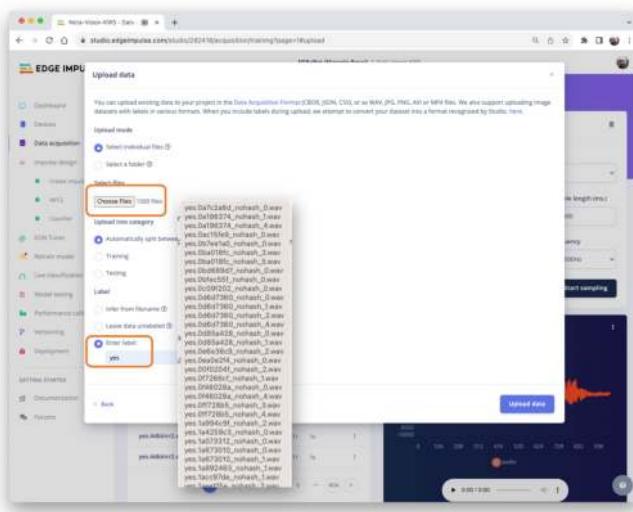
- Download the keywords dataset.
- Unzip the file to a location of your choice.

Uploading the dataset to the Edge Impulse Studio

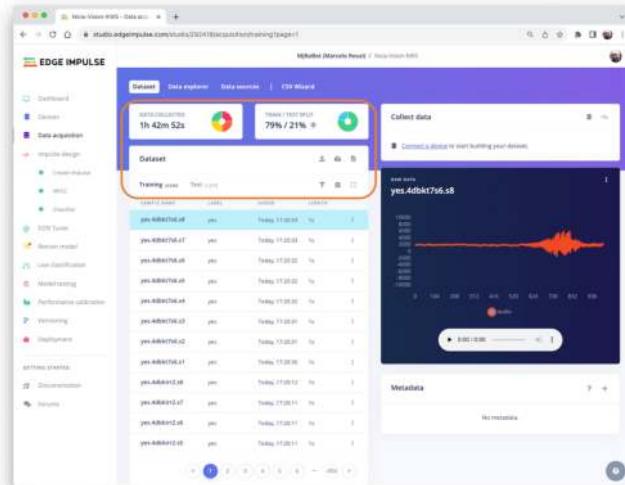
Initiate a new project at Edge Impulse Studio (EIS) and select the Upload Existing Data tool in the Data Acquisition section. Choose the files to be uploaded:



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

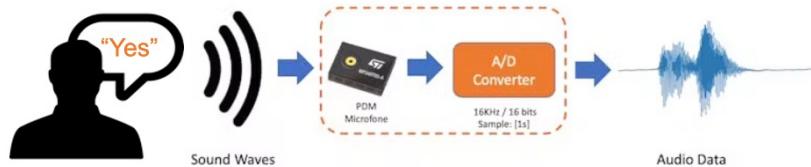
Although we have a lot of data from Pete's dataset, collecting some words spoken by us is advised. When working with accelerometers,

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.

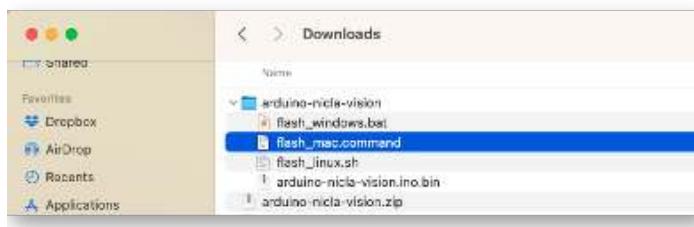
So, any device that can generate audio data with this basic specification (16 KHz/16 bits) will work fine. As a *device*, we can use the NiclaV, a computer, or even your mobile phone.



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:

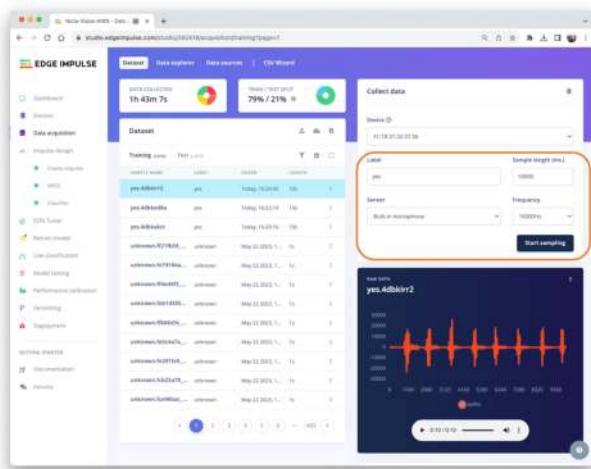
- Download the last updated EIS Firmware and unzip it.
- Open the zip file on your computer and select the uploader corresponding to your OS:



- 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.

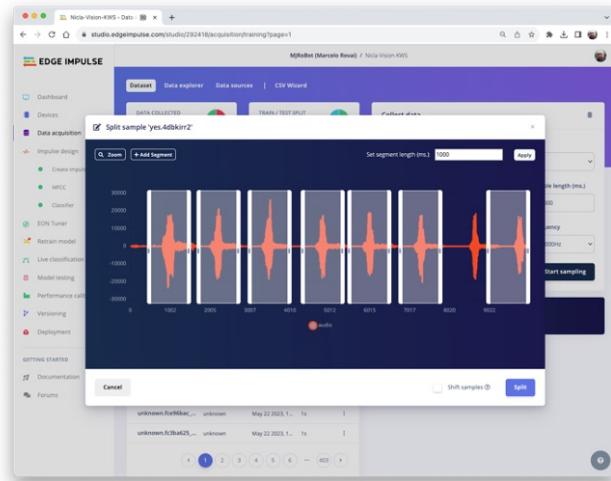
Go to your project on EIS, and on the Data Acquisition tab, select WebUSB. A window will pop up; choose the option that shows that the Nicla is paired and press [Connect].

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.

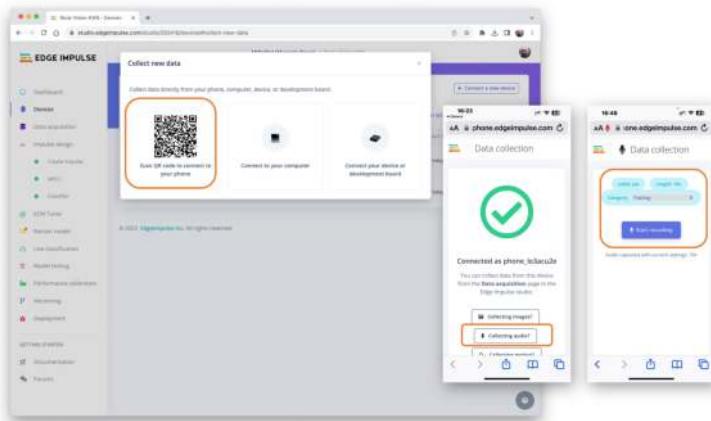


Once inside the tool, split the data into 1-second (1000 ms) records. If necessary, add or remove segments. This procedure should be repeated for all new samples.

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**.



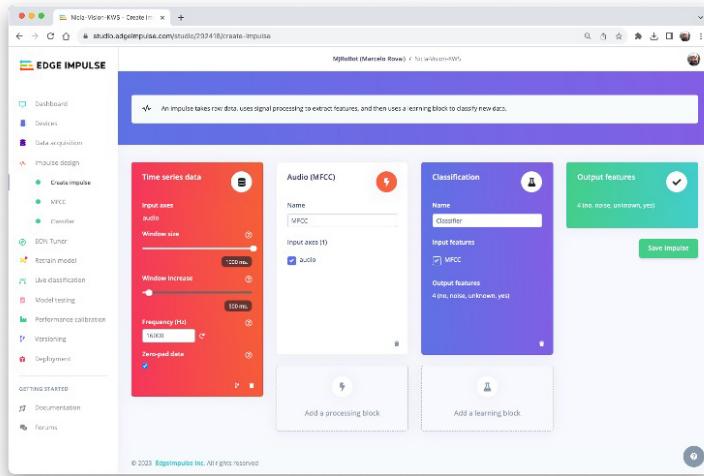
Repeat the same procedure used with the NiclaV.

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.

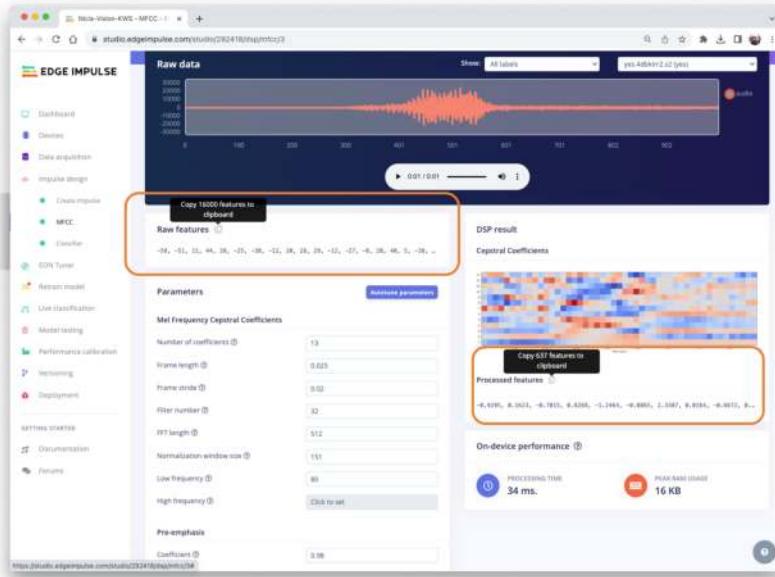
Next, we select the Classification block to build our model from scratch using a Convolution Neural Network (CNN).

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.

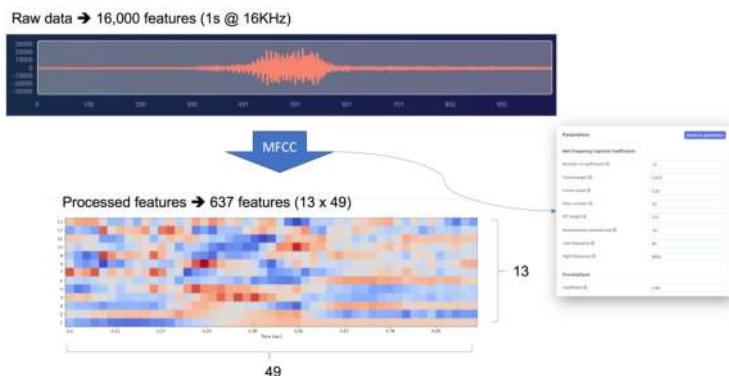
Pre-Processing (MFCC)

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.



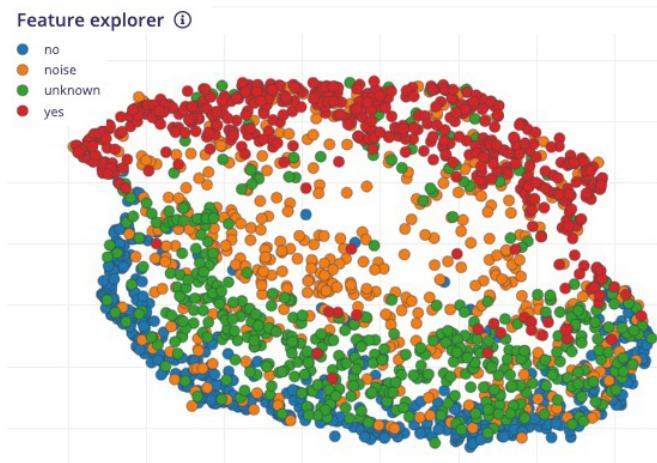
We will take the Raw features (our 1-second, 16 KHz sampled audio data) and use the MFCC processing block to calculate the Processed features. For every 16,000 raw features ($16,000 \times 1$ second), we will get 637 processed features (13×49).



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.

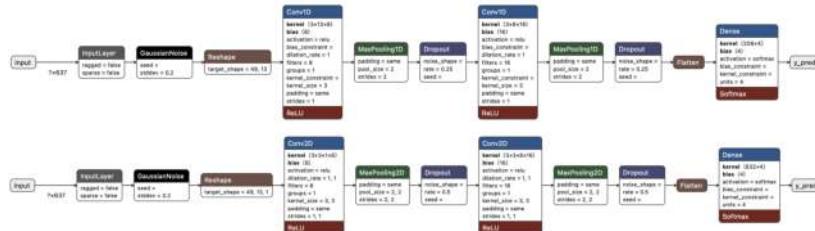
Going under the hood

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:

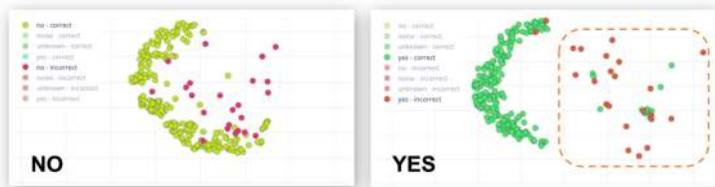


As hyper-parameters, we will have a Learning Rate of [0.005] and a model trained by [100] epochs. We will also include a data augmentation method based on SpecAugment. We trained the 1D and the 2D models with the same hyperparameters. The 1D architecture had a better overall result (90.5% accuracy when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

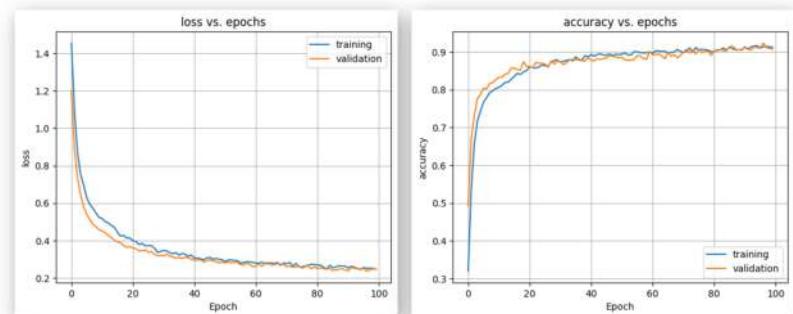
It is also interesting to pay attention to the 1D Confusion Matrix. The F1 Score for yes is 95%, and for no, 91%. That was expected by what we saw with the Feature Explorer (no and unknown at close distance). In trying to improve the result, you can inspect closely the results of the samples with an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

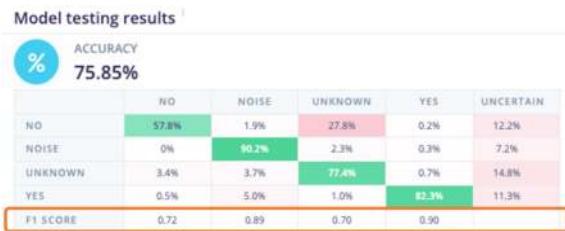
Going under the hood

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.

Configure your deployment

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

Q Arduino library X

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

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

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

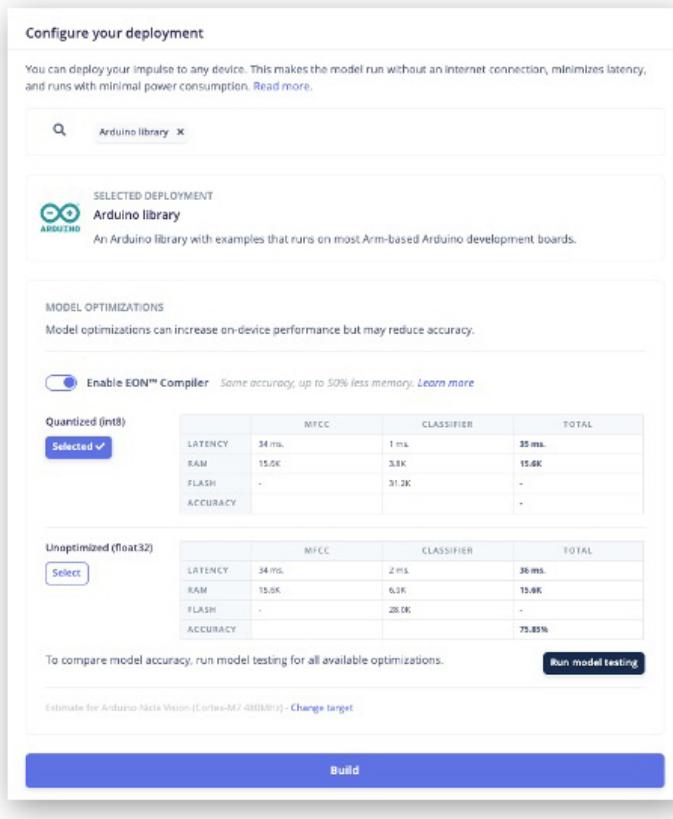
Quantized (int8)	MFCC	CLASSIFIER	TOTAL
Selected			
LATENCY	34 ms.	1 ms.	35 ms.
RAM	15.6K	3.3K	15.6K
FLASH	-	31.3K	-
ACCURACY			-

Unoptimized (float32)	MFCC	CLASSIFIER	TOTAL
Select			
LATENCY	34 ms.	2 ms.	36 ms.
RAM	15.6K	6.3K	15.6K
FLASH	-	28.0K	-
ACCURACY			75.85%

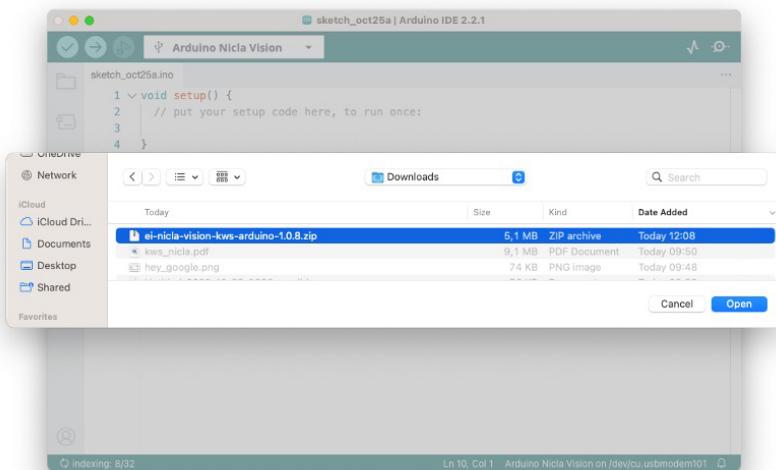
To compare model accuracy, run model testing for all available optimizations. [Run model testing](#)

Estimate for Arduino Nucleo Vision (Cortex-M7 400MHz) - [Change target](#)

Build



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



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

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



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



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(LED_R, OUTPUT);
    pinMode(LED_G, OUTPUT);
    pinMode(LED_B, OUTPUT);

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

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

```
/*
 * @brief      turn_off_leds function - turn-off all RGB LEDs
 */
```

```
void turn_off_leds(){
    digitalWrite(LED_R, HIGH);
    digitalWrite(LED_G, HIGH);
    digitalWrite(LED_B, HIGH);
}
```

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

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

        case 1:
            turn_off_leds();
            break;

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

        case 3:
            turn_off_leds();
            digitalWrite(LED_G, LOW);
            break;
    }
}
```

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

```
...
```

```
if (++print_results >= (EI_CLASSIFIER_SLICES_PER_MODEL_WINDOW)) {
    // print the predictions
    ei_printf("Predictions ");
    ei_printf("(DSP: %d ms., Classification: %d ms.,
        Anomaly: %d ms.)",
        result.timing.dsp, result.timing.classification,
        result.timing.anomaly);
    ei_printf(": \n");
    int pred_index = 0;      // Initialize pred_index
    float pred_value = 0;    // Initialize pred_value
    for (size_t ix = 0; ix < EI_CLASSIFIER_LABEL_COUNT; ix++) {
        if (result.classification[ix].value > pred_value){
            pred_index = ix;
            pred_value = result.classification[ix].value;
        }
        // ei_printf("    %s: ",
        // result.classification[ix].label);
        // ei_printf_float(result.classification[ix].value);
        // ei_printf("\n");
    }
    ei_printf(" PREDICTION: ==> %s with probability %.2f\n",
        result.classification[pred_index].label,
        pred_value);
    turn_on_leds (pred_index);

#ifndef 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>

Summary

You will find the notebooks and code used in this hands-on tutorial on the GitHub repository.

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

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

Motion Classification and Anomaly Detection



Figure 1.12: 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.

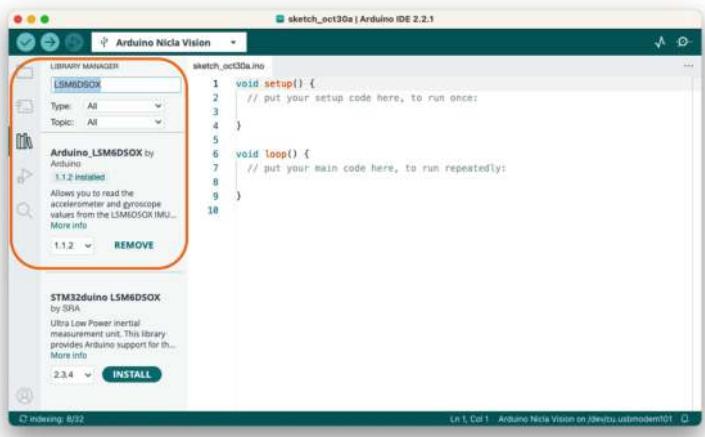
💡 Learning Objectives

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

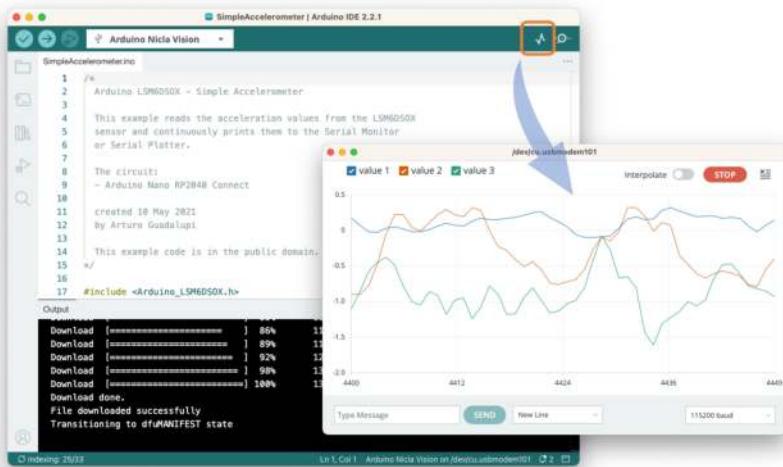
By the end of this tutorial, you'll have a working prototype that can classify different types of motion and detect anomalies during the transportation of containers. This knowledge can be a stepping stone to more advanced projects in the burgeoning field of TinyML involving vibration.

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 on-board **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.

```
Output Serial Monitor X

Message (Enter to send message to 'Arduino Nicla Vision')

17:58:59.986 -> 0.62 -0.08 9.85
17:59:00.021 -> 0.63 -0.08 9.85
17:59:00.054 -> 0.62 -0.08 9.85
17:59:00.054 -> 0.62 -0.08 9.86
17:59:00.087 -> 0.62 -0.08 9.85
17:59:00.087 -> 0.63 -0.07 9.86
17:59:00.120 -> 0.63 -0.08 9.86
17:59:00.120 -> 0.62 -0.08 9.85
17:59:00.153 -> 0.62 -0.08 9.86
.
.
.
.
17:59:00.888 -> 0.63 -0.08 9.85
17:59:00.920 -> 0.64 -0.08 9.86
17:59:00.920 -> 0.62 -0.08 9.85
17:59:00.952 -> 0.62 -0.08 9.86
17:59:00.986 -> 0.63 -0.07 9.85
17:59:00.986 -> 0.63 -0.08 9.86
17:59:01.017 -> 0.62 -0.07 9.85
```

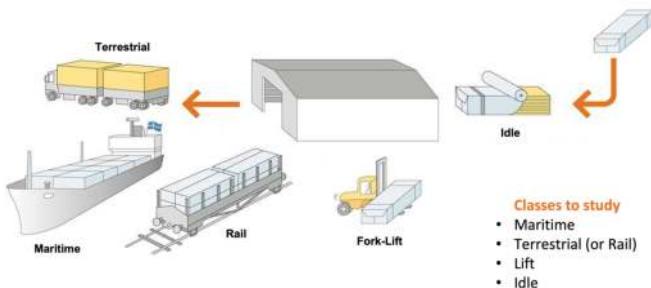
50 samples / second

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.

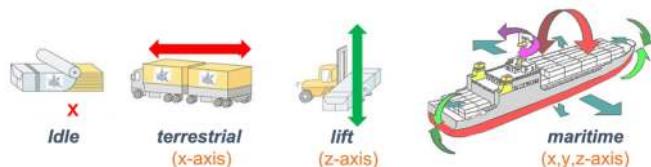
The Case Study: Simulated Container Transportation

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

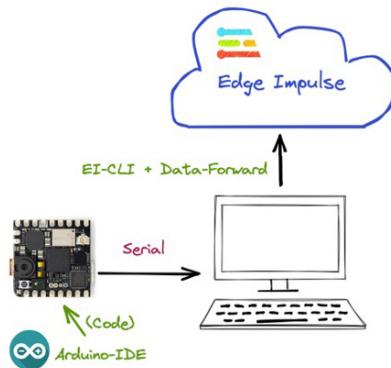
We will connect the Nicla directly to the Edge Impulse Studio, which will also be used for data pre-processing, model training, testing, and deployment. For that, you have two options:

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:

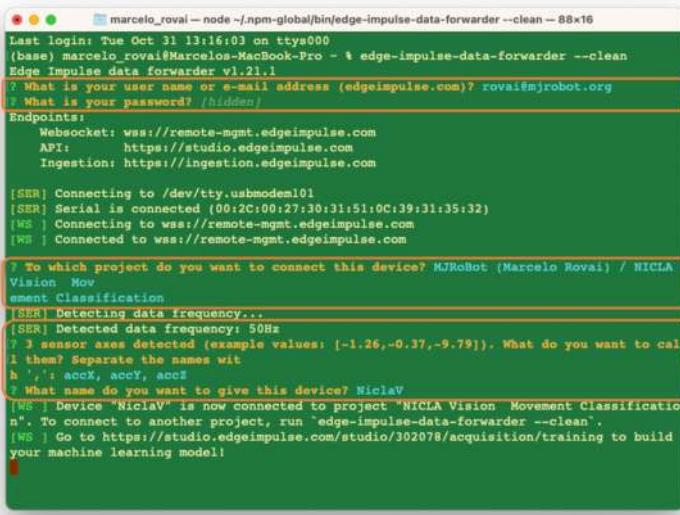
1. Install the Edge Impulse CLI and the Node.js into your computer.
2. Upload a sketch for data capture (the one discussed previously in this tutorial).
3. Use the CLI Data Forwarder to capture data from the Nicla's accelerometer and send it to the Studio, as shown in this diagram:



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*):



```

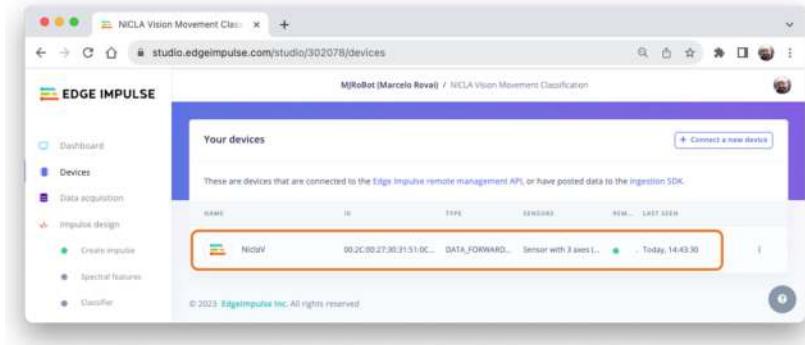
marcelo_roval — node ~/npm-global/bin/edge-impulse-data-forwarder --clean --88x16
Last login: Tue Oct 31 13:16:03 on ttys000
(base) marcelo_roval@Marcelos-MacBook-Pro ~ % edge-impulse-data-forwarder --clean
Edge Impulse data forwarder v1.21.1
? What is your user name or e-mail address (edgeimpulse.com)? roval@mjrobot.org
? What is your password? (hidden)
Endpoints:
  Websocket: ws://remote-mgmt.edgeimpulse.com
  API: https://studio.edgeimpulse.com
  Ingestion: https://ingestion.edgeimpulse.com

[SER] Connecting to /dev/tty.usbmodem101
[SER] Serial is connected [00:2C:00:27:30:31:51:0C:39:31:35:32]
[WB] Connecting to ws://remote-mgmt.edgeimpulse.com
[WB] Connected to ws://remote-mgmt.edgeimpulse.com

? To which project do you want to connect this device? MJRoBot (Marcelo Rovai) / NicLA Vision Movement Classification
[SER] Detecting data frequency...
[SER] Detected data frequency: 50Hz
? 3 sensor axes detected (example values: [-1.26,-0.37,-9.79]). What do you want to call them? Separate the names with a ',': accX, accY, accZ
? What name do you want to give this device? NiclaV
[WB] Device "NiclaV" is now connected to project "NicLA Vision Movement Classification". To connect to another project, run 'edge-impulse-data-forwarder --clean'.
[WB] Go to https://studio.edgeimpulse.com/studio/302078/acquisition/training to build your machine learning model!

```

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]:

Collect data

Device ②

NiclaV

Label

terrestrial

Sample length (ms.)

10000

Sensor

Sensor with 3 axes (accX, accY, accZ)

Frequency

50Hz

Start sampling

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:



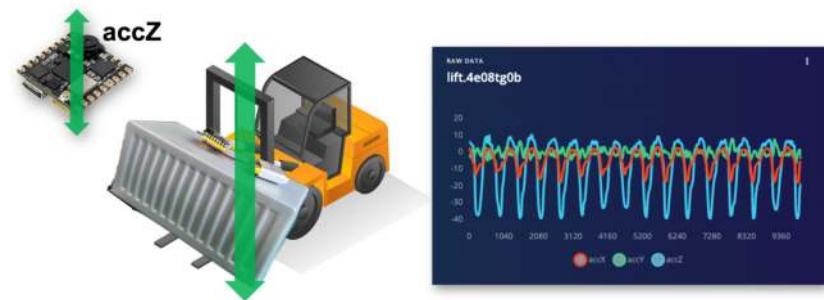
As expected, the movement was captured mainly in the *Y*-axis (green). In the blue, we see the *Z* axis, around -10 m/s^2 (the Nicla has the camera facing up).

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

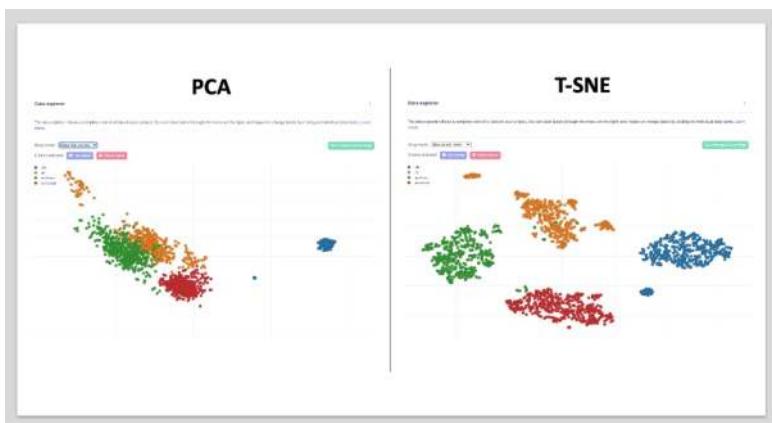


You can capture, for example, 2 minutes (twelve samples of 10 seconds) for each of the four classes (a total of 8 minutes of data). Using the three dots menu after each one of the samples, select 2 of them, reserving them for the Test set. Alternatively, you can use the automatic Train/Test Split tool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

The screenshot shows the Edge Impulse Studio interface with the following details:

- Left Sidebar:** Includes sections for Dashboard, Devices, Data acquisition, Impulse design (Create impulse, Spectral feature, Classifier), EDN Tuner, Retrain model, Use classification, Model testing, Versioning, and Deployments.
- Top Bar:** Shows "NINA Vision Movement Class" and the URL "studio.edgeimpulse.com/studio/302078/acquisition/training?page=3".
- Header:** "MjModel (Marine) Rev1 / NINA Vision Movement Classification".
- Dataset Summary:** "DATA COLLECTED: 8m 0s" and "TRAIN / TEST SPLIT: 83% / 17%".
- Dataset Table:** Shows a list of samples with columns: SAMPLE NAME, LABEL, ACCELERATION, and LENGTH. One entry is highlighted: "maritime.4e0dd14c" (Label: maritime, Today, 18:00:33, 10s).
- Collect Data Panel:** Includes fields for Device (NetW), Label (site), Sample length (ms.) (100000), Sensor (Sensor with 3 axes (accX, accY, accZ)), Frequency (50Hz), and a "Start sampling" button.
- Plot View:** A line graph titled "new data maritime.4e0dd14c" showing three data series (red, green, blue) over time, with a legend indicating "site" (green), "maritime" (blue), and "terrestrial" (red).

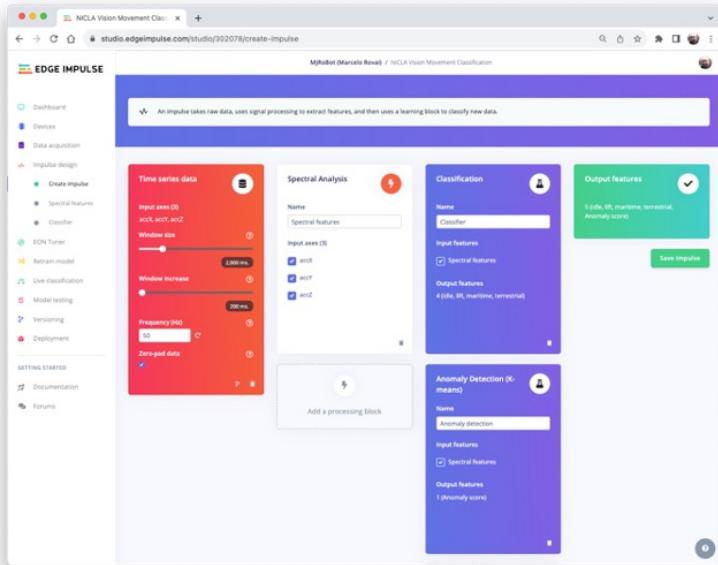
Once you have captured your dataset, you can explore it in more detail using the Data Explorer, a visual tool to find outliers or mislabeled data (helping to correct them). The data explorer first tries to extract meaningful features from your data (by applying signal processing and neural network embeddings) 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.



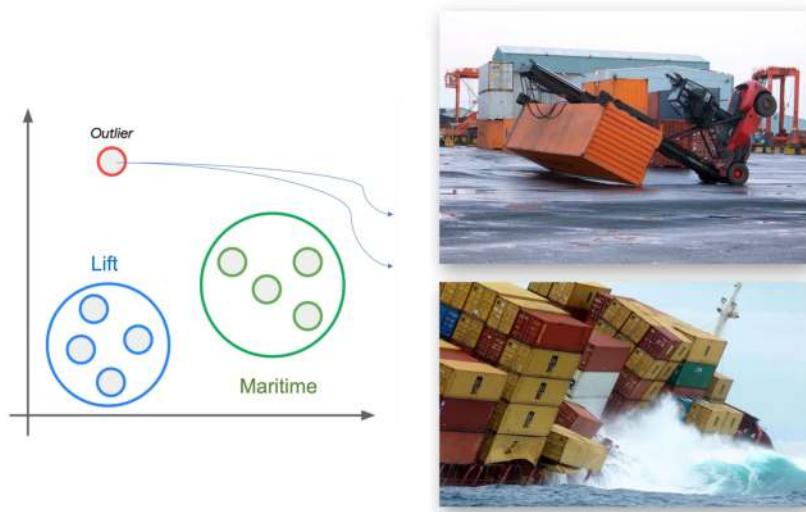
In our case, the dataset seems OK (good separation). But the PCA shows we can have issues between maritime (green) and lift (orange). This is expected, once on a boat, sometimes the movement can be only “vertical”.

Impulse Design

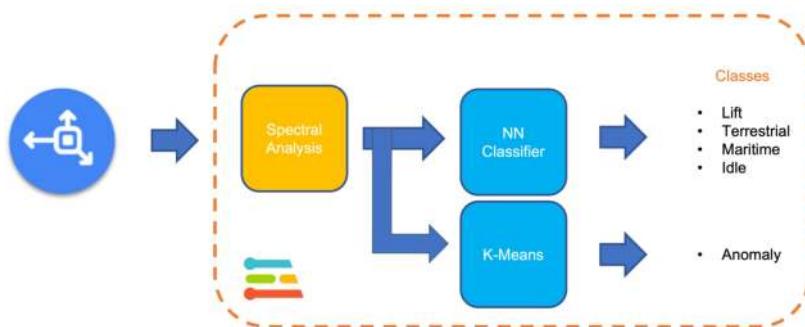
The next step is the definition of our Impulse, which takes the raw data and uses signal processing to extract features, passing them as the input tensor of a *learning block* to classify new data. Go to **Impulse Design** and **Create Impulse**. The Studio will suggest the basic design. Let's also add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



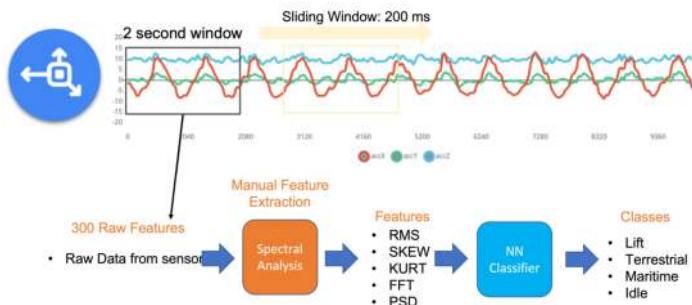
Let's dig into those steps and parameters to understand better what we are doing here.

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 \times 2 seconds \times 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

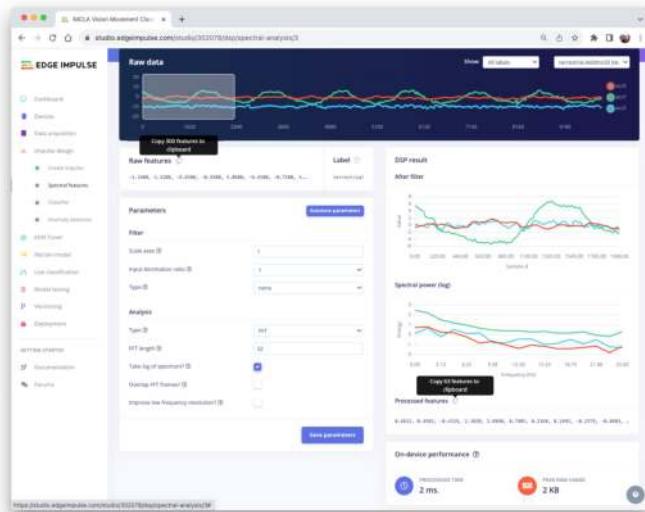
- Kurtosis: 1 feature

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

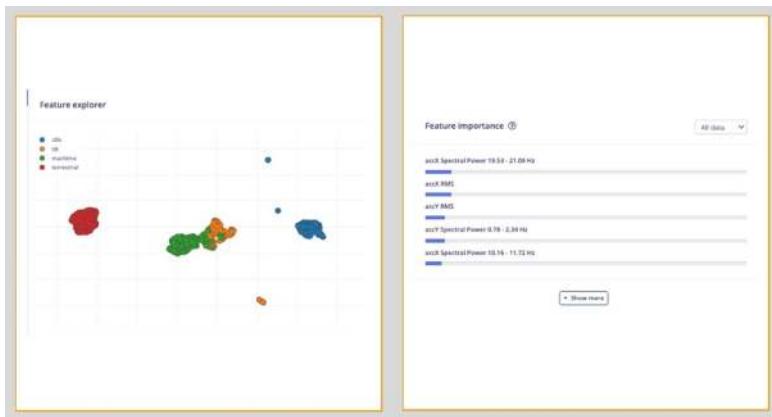
Once we understand what the pre-processing does, it is time to finish the job. So, let's take the raw data (time-series type) and convert it to tabular data. For that, go to the Spectral Features section on the Parameters tab, define the main parameters as discussed in the previous section ([FFT] with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

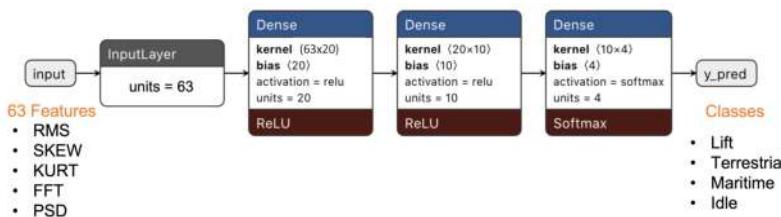
The Feature Explorer will show those data in 2D using UMAP. Uniform Manifold Approximation and Projection (UMAP) is a dimension reduction technique that can be used for visualization similarly to t-SNE but is also applicable for general non-linear dimension reduction.

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.



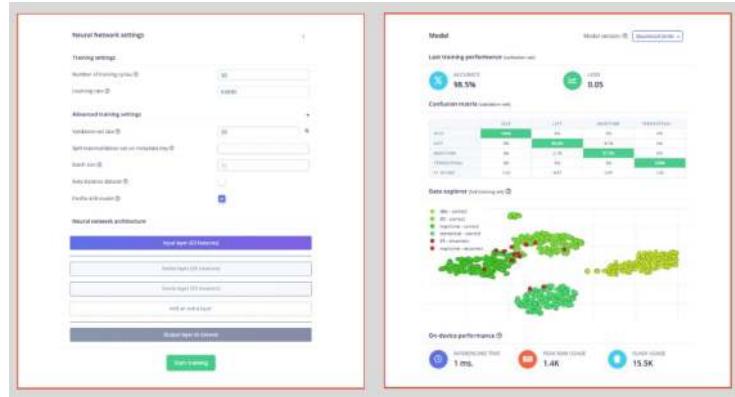
Models Training

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:

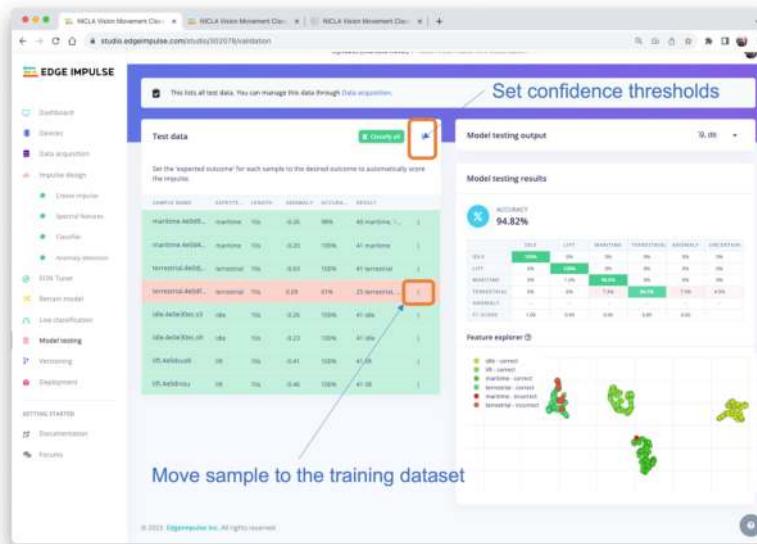


Testing

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.



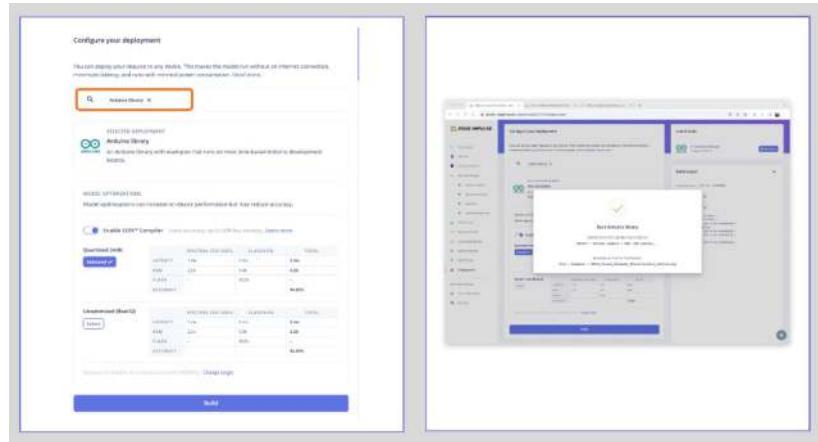
You can also perform Live Classification with your device (which should still be connected to the Studio).

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.

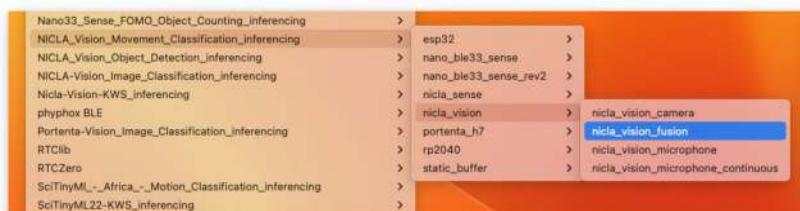


On your Arduino IDE, go to the Sketch tab, select Add.ZIP Library, and Choose the.zip file downloaded by the Studio. A message will appear in the IDE Terminal: Library installed.

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
```

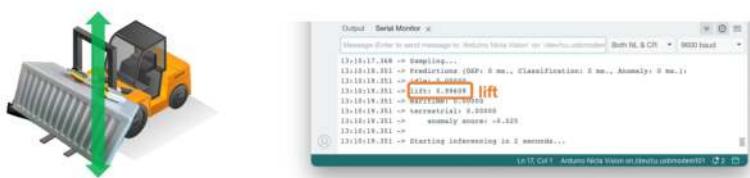
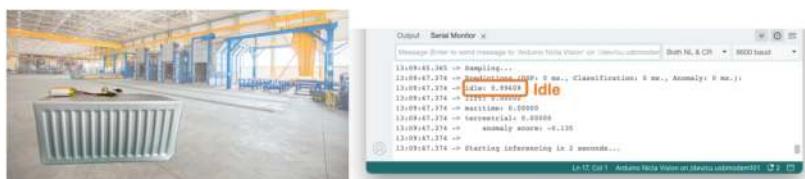
You can keep the code this way for testing because the trained model will use only features pre-processed from the accelerometer. But consider that you will write your code only with the needed libraries for a real project.

And that is it!

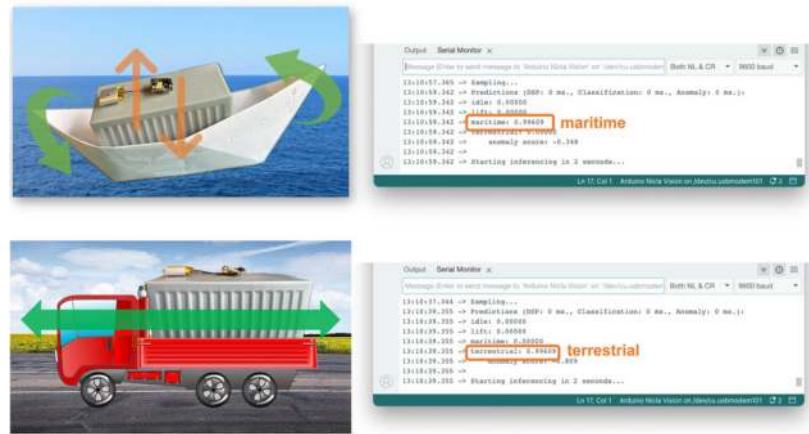
You can now upload the code to your device and proceed with the inferences. Press the Nicla [RESET] button twice to put it on boot mode (disconnect from the Studio if it is still connected), and upload the sketch to your board.

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).

Summary

The notebooks and code used 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

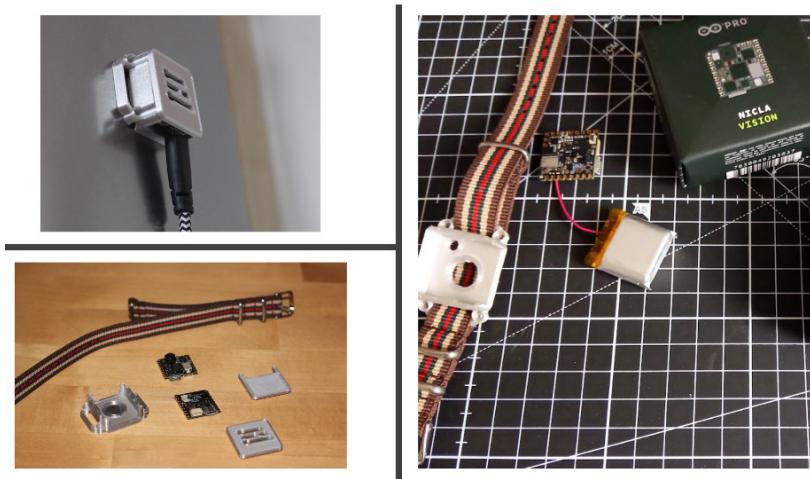
- **Equipment Monitoring:** Tracking the performance and usage of agricultural machinery.
- **Animal Behavior Analysis:** Monitoring livestock movements to detect behaviors indicating health issues or stress.

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

III

KEY:XIAO

Part III

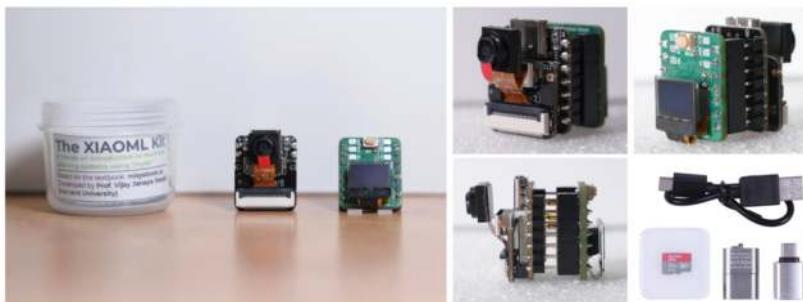
IV

SEEED XIAO ESP32S3

Part IV

Overview

These labs provide a unique opportunity to gain practical experience with machine learning (ML) systems. Unlike working with large models that require data center-scale resources, these exercises enable you to directly interact with hardware and software using TinyML. This hands-on approach provides a tangible understanding of the challenges and opportunities in deploying AI, albeit on a small scale. However, the principles are largely the same as what you would encounter when working with larger systems.



Where to Buy

The XIAOML Kit bundles the XIAO ESP32S3 Sense with an expansion board, IMU, OLED display, and SD card toolkit:

- XIAOML Kit (Seeed Studio) (~\$40)

Individual components are also available separately from Seeed Studio.

Pre-requisites

- **The XIAOML Kit:**
 - XIAO ESP32S3 Sense board
 - Expansion board with a 6-axis IMU and 0.42" OLED display.
 - SD card toolkit:
 - * SD Card and USB adapter for data storage
 - * USB-C Cable for connecting the board to your computer.
- **Network:** With internet access for downloading the necessary software.

Setup

- Setup the XIAOML Kit

Exercises

Modal- ity	Task	Description	Link
Vision	Image Classification	Learn to classify images	Link
Vision	Object Detection	Implement object detection	Link
Sound	Keyword Spotting	Explore voice recognition systems	Link
IMU	Motion Classification and Anomaly Detection	Classify motion data and detect anomalies	Link

Setup

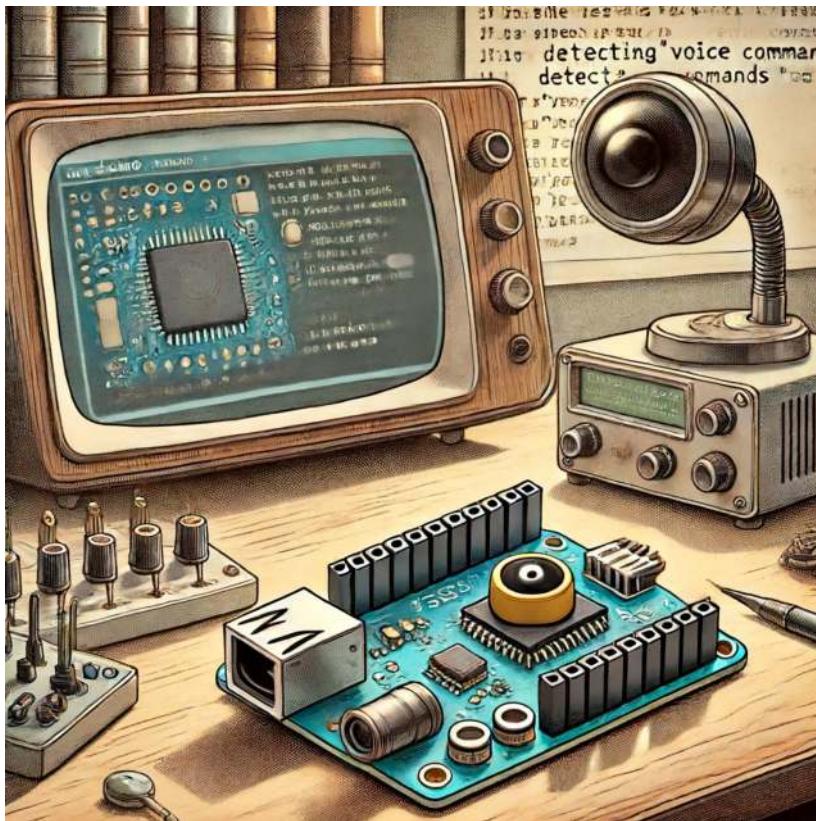


Figure 1.13: DALL-E prompt - 1950s cartoon-style drawing of a XIAO ESP32S3 board with a distinctive camera module, as shown in the image provided. The board is placed on a classic lab table with various sensors, including a microphone. Behind the board, a vintage computer screen displays the Arduino IDE in muted colors, with code focusing on LED pin setups and machine learning inference for voice commands. The Serial Monitor on the IDE showcases outputs detecting voice commands like 'yes' and 'no'. The scene merges the retro charm of mid-century labs with modern electronics.

Overview

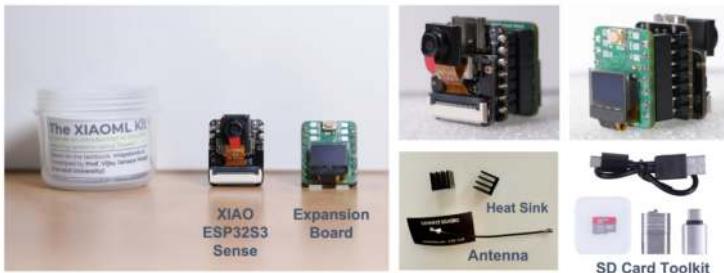
The XIAOML Kit is designed to provide hands-on experience with TinyML applications. The kit includes the powerful XIAO ESP32S3 Sense development board and an expansion board that adds essential sensors for machine learning projects.

Complete XIAOML Kit Components:

- **XIAO ESP32S3 Sense:** Main development board with integrated camera sensor, digital microphone, and SD card support
- **Expansion Board:** Features a 6-axis IMU (LSM6DS3TR-C) and 0.42" OLED display for motion sensing and data visualization
- **SD Card Toolkit:** Includes SD card and USB adapter for data storage and model deployment
- **USB-C Cable:** For connecting the board to your computer
- **Antenna and Heat Sinks**

Attention

Do not install the heat sinks (or carefully, remove them) on/from the XIAO ESP32S3 if you want to use the XIAO ML Kit Expansion Board. See Appendix for more information.



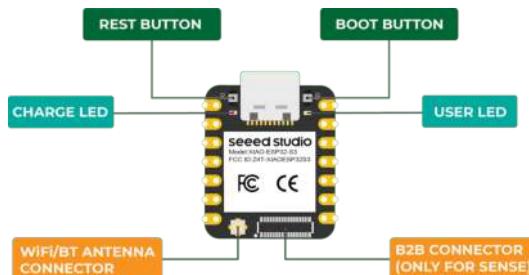
XIAO ESP32S3 Sense - Core Board Features

The XIAO ESP32S3 Sense serves as the heart of the XIAOML Kit, integrating embedded ML computing power with photography and audio capabilities, making it an ideal platform for TinyML applications in intelligent voice and vision AI.

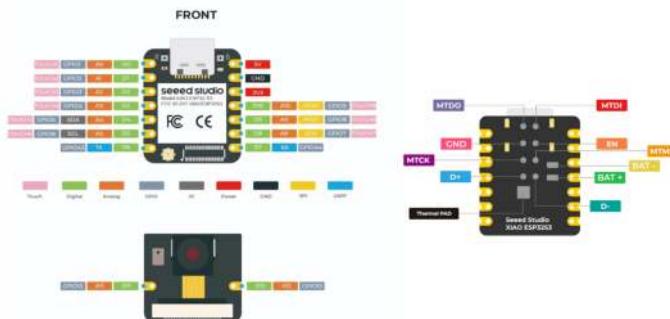


Key Features

- **Powerful MCU:** ESP32S3 32-bit, dual-core, Xtensa processor operating up to 240 MHz, with Arduino / MicroPython support
- **Advanced Functionality:** Detachable OV2640 camera sensor for 1600 × 1200 resolution, compatible with OV5640 camera sensor, plus integrated digital microphone
- **Elaborate Power Design:** Lithium battery charge management with four power consumption models, deep sleep mode with power consumption as low as 14 µA
- **Great Memory:** 8 MB PSRAM and 8 MB FLASH, supporting SD card slot for external 32 GB FAT memory
- **Outstanding RF Performance:** 2.4 GHz Wi-Fi and BLE dual wireless communication, supports 100m+ remote communication with U.FL antenna
- **Compact Design:** 21 × 17.5 mm, adopting the classic XIAO form factor, suitable for space-limited projects



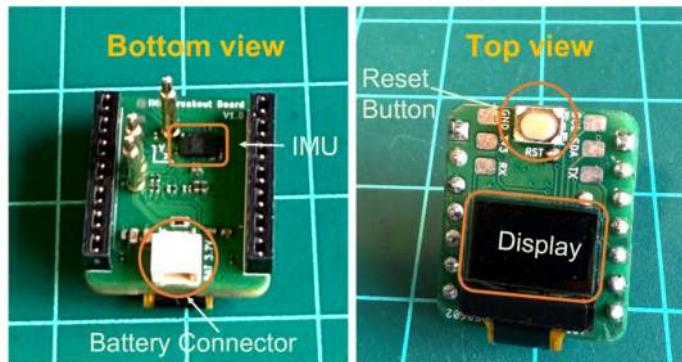
Below is the general board pinout:



For more details, please refer to the Seeed Studio Wiki page

Expansion Board Features

The expansion board extends the XIAOML Kit's capabilities for motion-based machine learning applications:



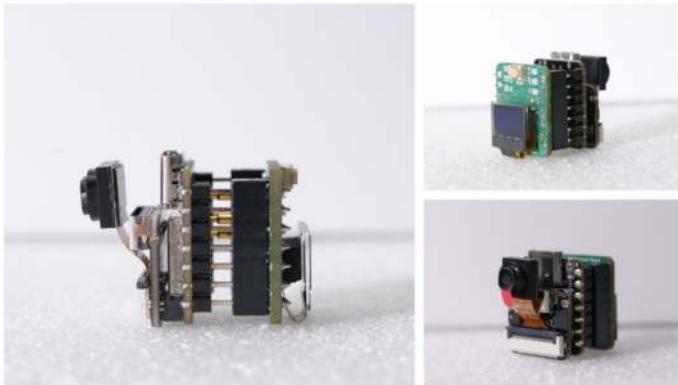
Components:

- 6-axis IMU (LSM6DS3TR-C):
 - 3-axis accelerometer and 3-axis gyroscope for motion detection and classification
 - * Accelerometer range: $\pm 2/\pm 4/\pm 8/\pm 16$ g
 - * Gyroscope range: $\pm 125/\pm 250/\pm 500/\pm 1000/\pm 2000$ dps

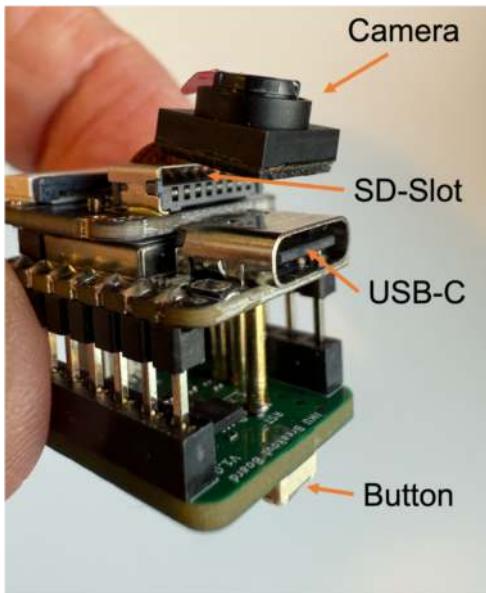
- * I2C interface (address: 0x6A)
- 0.42" OLED Display
 - Monochrome display (72×40 resolution) for real-time data visualization
 - * Controller: SSD1306
 - * I2C interface (address: 0x3C)
- Restart Button (EN)
- Battery Connector (BAT+, BAT-)

Complete Kit Assembly

The expansion board connects seamlessly to the XIAO ESP32S3 Sense, creating a comprehensive platform for multimodal machine learning experiments covering vision, audio, and motion sensing.



Please pay attention to the mounting orientation of the module:



Note that

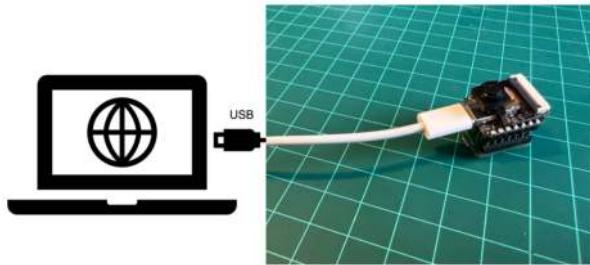
- The EN connection, shown at the bottom of the ESP32S3 Sense, is available on the expansion board via the RST button.
- The BAT+ and BAT- connections are also available through the BAT3.7V white connector.

XIAOML Kit Applications:

- **Vision:** Image classification and object detection using the integrated camera
- **Audio:** Keyword spotting and voice recognition with the built-in microphone
- **Motion:** Activity recognition and anomaly detection using the IMU sensors
- **Multi-modal:** Combined sensor fusion for complex ML applications

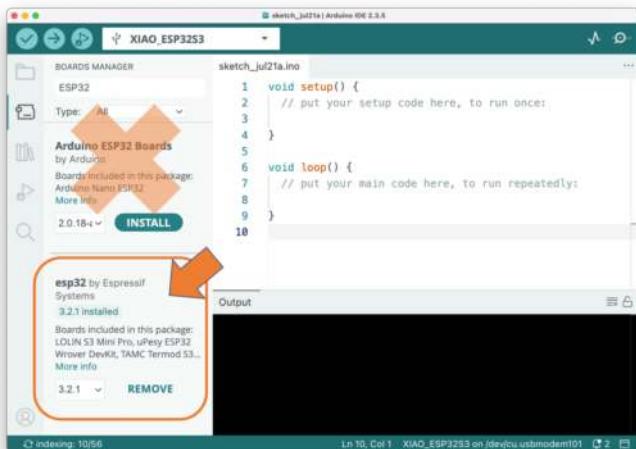
Installing the XIAO ESP32S3 Sense on Arduino IDE

1. Connect the XIAOML Kit to your computer via the USB-C port.



2. Download and Install the stable version of Arduino IDE according to your operating system.
[\[Download Arduino IDE\]](#)
3. Open the **Arduino IDE** and select the Boards Manager (represented by the UNO Icon).
4. Enter “**ESP32**”, and select “**esp32 by Espressif Systems**.“ You can install or update the board support packages.

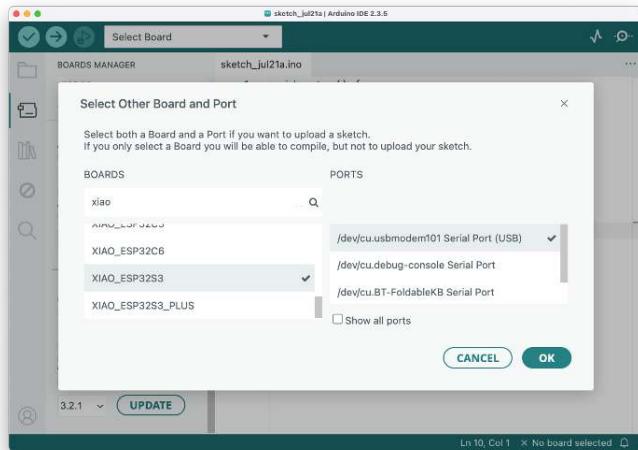
Do not select “**Arduino ESP32 Boards** by Arduino”, which are the support package for the Arduino Nano ESP32 and not our board.



Attention

Versions 3.x may experience issues when using the XIAO ESP32S3 Sense with Edge Impulse deploy codes. If this is the case, use the last 2.0.x stable version (for example, 2.0.17) instead.

5. Click Select Board, enter with *xiao* or *esp32s3*, and select the XIAO_ESP32S3 in the boards manager and the corresponding PORT where the ESP32S3 is connected.



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

Testing the board with BLINK

The XIAO ESP32S3 Sense features a built-in LED connected to GPIO21. So, you can run the blink sketch (which can be found under *Files/Examples/Basics/Blink*). The sketch uses the `LED_BUILTIN` Arduino constant, which internally corresponds to the LED connected to pin 21. Alternatively, you can change the Blink sketch accordingly.

```
#define LED_BUILTIN 21 // This line is optional

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

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

```
    digitalWrite(LED_BUILT_IN, HIGH); //Turn off
    delay (1000); //Wait 1 sec
}
```

Note that the pins operate with inverted logic: LOW turns on and HIGH turns off.



Microphone Test

Let's start with sound detection. Enter with the code below or go to the GitHub project and download the sketch: XIAOML_Kit_Mic_Test and run it on the Arduino IDE:

```
/*
  XIAO ESP32S3 Simple Mic Test
  (for ESP32 Library version 3.0.x and later)
*/

#include <ESP_I2S.h>
I2SClass I2S;

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

  // setup 42 PDM clock and 41 PDM data pins
```

```
I2S.setPinsPdmRx(42, 41);

// start I2S at 16 kHz with 16-bits per sample
if (!I2S.begin(I2S_MODE_PDM_RX,
                16000,
                I2S_DATA_BIT_WIDTH_16BIT,
                I2S_SLOT_MODE_MONO)) {
    Serial.println("Failed to initialize I2S!");
    while (1); // do nothing
}

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

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

Open the **Serial Plotter**, and you will see the loudness change curve of the sound.



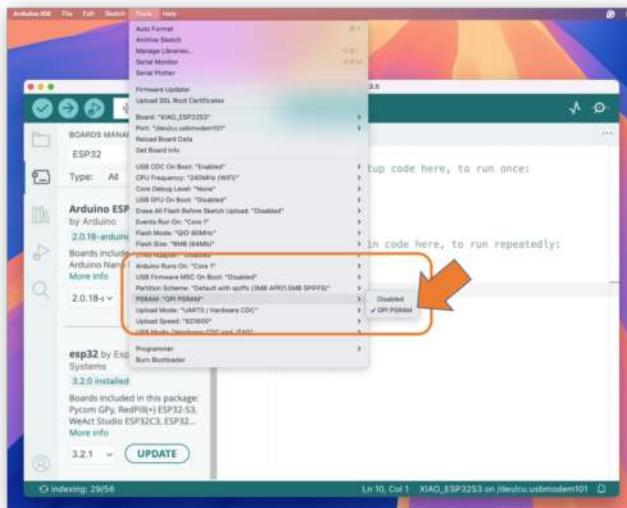
When producing sound, you can verify it on the Serial Plotter.

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

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

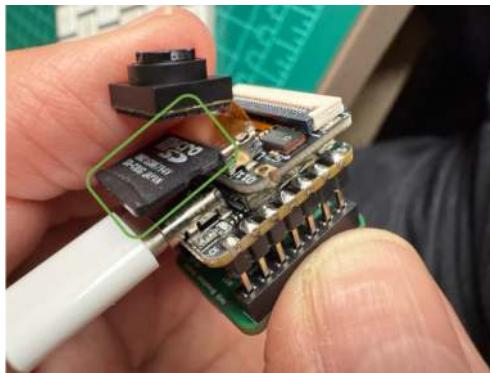
ESP32-S3 has only a few hundred kilobytes of internal RAM on the MCU chip. This can be insufficient for some purposes, so up to 16 MB of external PSRAM (pseudo-static RAM) can be connected with the SPI flash chip (The XIAO has 8 MB of PSRAM). The external memory is incorporated in the memory map and, with certain restrictions, is usable in the same way as internal data RAM.

- To turn it on, go to Tools->PSRAM: "OPI PSRAM"->OPI PSRAM



XIAO ESP32S3 Sense supports microSD cards up to **32GB**. If you are ready to purchase a microSD card for XIAO, please refer to the specifications below. Format the microSD card to **FAT32 format** before using it.

Now, insert the FAT32 formatted SD card into the XIAO as shown in the photo below



```
/*
 * WAV Recorder for Seeed XIAO ESP32S3 Sense
 * (for ESP32 Library version 3.0.x and later)
 */

#include "ESP_I2S.h"
#include "FS.h"
#include "SD.h"

void setup() {
    // Create an instance of the I2SClass
    I2SClass i2s;

    // Create variables to store the audio data
    uint8_t *wav_buffer;
    size_t wav_size;

    // Initialize the serial port
    Serial.begin(115200);
    while (!Serial) {
        delay(10);
    }

    Serial.println("Initializing I2S bus...");

    // Set up the pins used for audio input
    i2s.setPinsPdmRx(42, 41);

    // start I2S at 16 kHz with 16-bits per sample
    if (!i2s.begin(I2S_MODE_PDM_RX,
                    16000,
```

```
I2S_DATA_BIT_WIDTH_16BIT,
I2S_SLOT_MODE_MONO)) {
Serial.println("Failed to initialize I2S!");
while (1); // do nothing
}

Serial.println("I2S bus initialized.");
Serial.println("Initializing SD card...");

// Set up the pins used for SD card access
if(!SD.begin(21)){
    Serial.println("Failed to mount SD Card!");
    while (1) ;
}
Serial.println("SD card initialized.");
Serial.println("Recording 20 seconds of audio data...");

// Record 20 seconds of audio data
wav_buffer = i2s.recordWAV(20, &wav_size);

// Create a file on the SD card
File file = SD.open("/arduinor_rec.wav", FILE_WRITE);
if (!file) {
    Serial.println("Failed to open file for writing!");
    return;
}
Serial.println("Writing audio data to file...");

// Write the audio data to the file
if (file.write(wav_buffer, wav_size) != wav_size) {
    Serial.println("Failed to write audio data to file!");
    return;
}

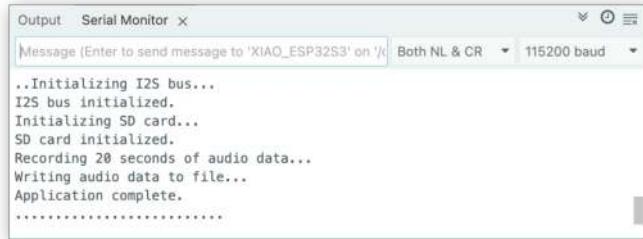
// Close the file
file.close();

Serial.println("Application complete.");
}

void loop() {
delay(1000);
```

```
    Serial.printf(".");
}
```

- Save the code, for example, as `Wav_Record.ino`, and run it in the Arduino IDE.
- This program is executed only once after the user turns on the serial monitor (or when the RESET button is pressed). It records for 20 seconds and saves the recording file to a microSD card as “`arduino_rec.wav`.”
- When the “.” is output every second in the serial monitor, the program execution is complete, and you can play the recorded sound file using a card reader.



The sound quality is excellent!

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

To know more about the File System on the XIAO ESP32S3 Sense, please refer to this link.

Testing the Camera

For testing (and using the camera, we can use several methods:

- The SenseCraft AI Studio
- The CameraWebServer app on Arduino IDE (See the next section)
- Capturing images and saving them on an SD card (similar to what we did with audio)

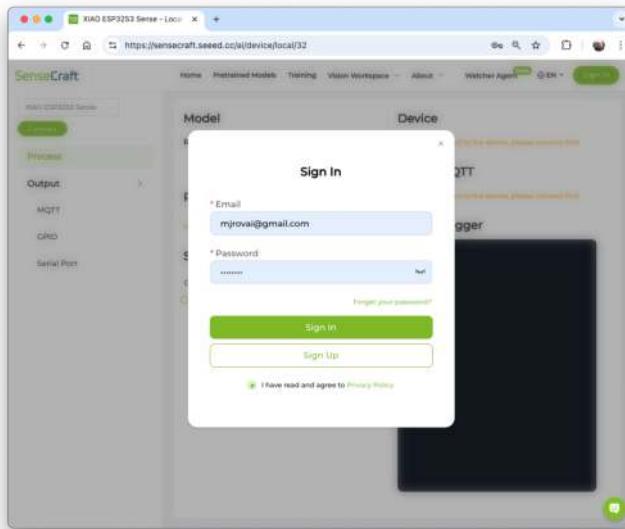
Testing the camera with the SenseCraft AI Studio

The easiest way to see the camera working is to use the SenseCraft AI Studio, a robust platform that offers a wide range of AI models compatible with various devices, including the **XIAO ESP32S3 Sense** and the Grove Vision AI V2.

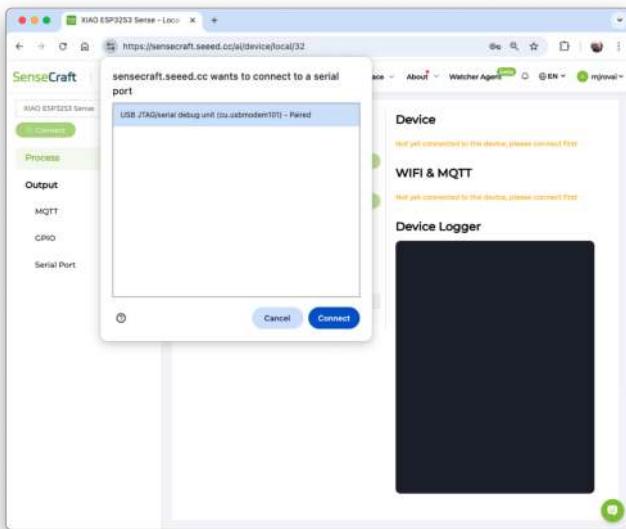
We can also use the **SenseCraft Web Toolkit**, a simplified version of the SenseCraft AI Studio.

Let's follow the steps below to start the **SenseCraft AI**:

- Open the SenseCraft AI Vision Workspace in a web browser, such as **Chrome**, and sign in (or create an account).

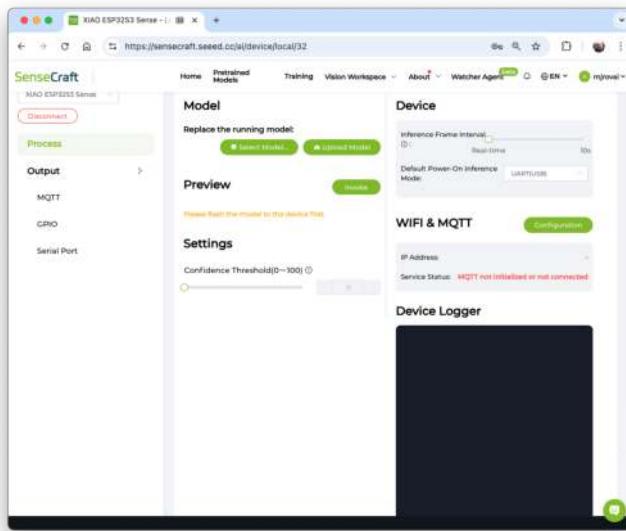


- Having the XIAOML Kit physically connected to the notebook, select it as below:

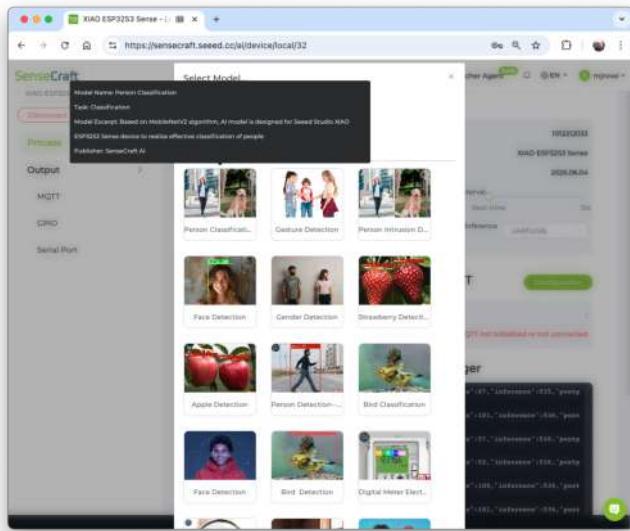


Note: The **WebUSB tool** may not function correctly in certain browsers, such as Safari. Use Chrome instead. Also, confirm that the Arduino IDE or any other serial device is not connected to the XIAO.

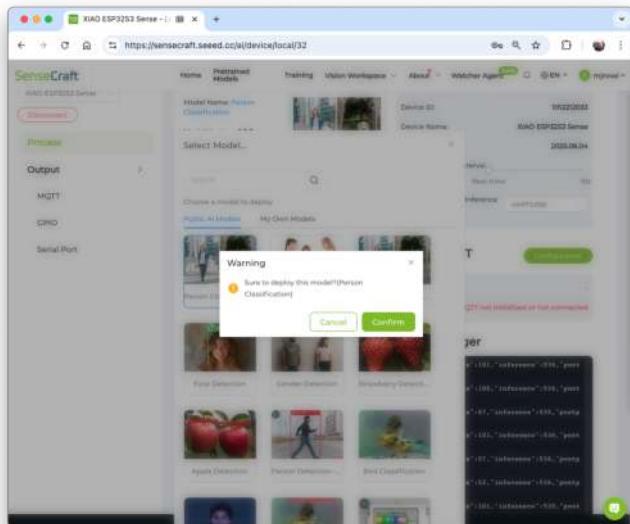
To see the camera working, we should upload a model. We can try several Computer Vision models previously uploaded by Seeed Studio. Use the button [Select Model] and choose among the available models.



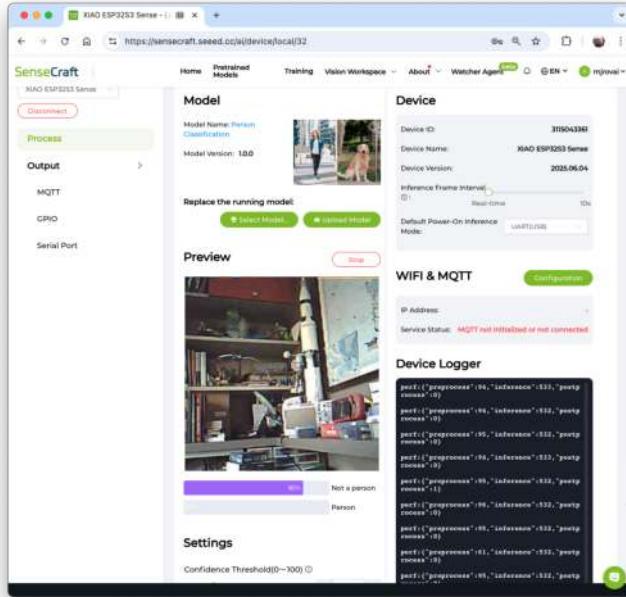
Passing the cursor over the AI models, we can have some information about them, such as name, description, **category** or **task** (Image Classification, Object Detection, or Pose/Keypoint Detection), the **algorithm** (like YOLO V5 or V8, FOMO, MobileNet V2, etc.) and in some cases, **metrics** (Accuracy or mAP).



We can choose one of the ready-to-use AI models, such as “Person Classification”, by clicking on it and pressing the [Confirm] button, or upload our own model.



In the **Preview Area**, we can see the streaming generated by the camera.



We will return to the SenseCraft AI Studio in more detail during the Vision AI labs.

Testing WiFi

Installation of the antenna

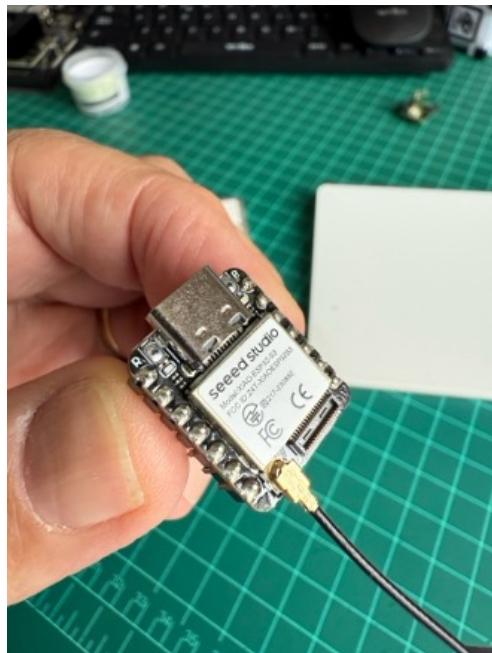
The XIAO ML Kit arrived fully assembled. First, remove the Sense Expansion Board (which contains the Camera, Mic, and SD Card Reader) from the XIAO.

On the bottom left of the front of XIAO ESP32S3, there is a separate "WiFi/BT Antenna Connector". To improve your WiFi/Bluetooth signal, remove the antenna from the package and attach it to the connector.

There is a small trick to installing the antenna. If you press down hard on it directly, you will find it very difficult to press and your fingers

will hurt! The correct way to install the antenna is to insert one side of the antenna connector into the connector block first, then gently press down on the other side to ensure the antenna is securely installed.

Removing the antenna is also the case. Do not use brute force to pull the antenna directly; instead, apply force to one side to lift, making the antenna easy to remove.



Reinstalling the expansion board is very simple; you just need to align the connector on the expansion board with the B2B connector on the XIAO ESP32S3, press it hard, and hear a “click.” The installation is complete.

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

Open the Arduino IDE and select our board and port. Go to Examples and look for **WiFi ==> WiFiScan** under the “Examples for the XIAO ESP32S3”. Upload the sketch to the board.

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



Simple WiFi Server (Turning LED ON/OFF)

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

Go to Examples and look for WiFi ==> SimpleWiFiServer under the "Examples for the XIAO ESP32S3".

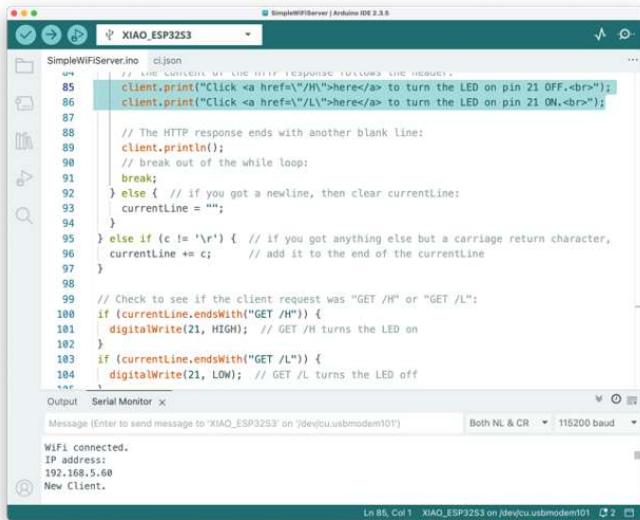
Before running the sketch, you should enter your network credentials:

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

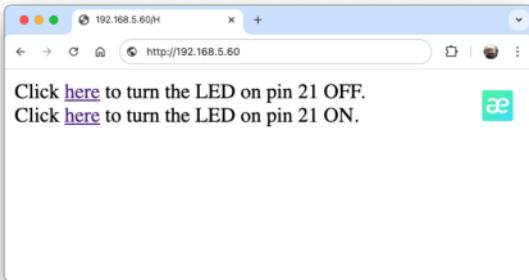
And modify pin 5 to pin 21, where the built-in LED is installed. Also, let's modify the webpage (lines 85 and 86) to reflect the correct LED Pin and that it is active with LOW:

```
client.print("Click <a href=\"/H\">here</a> to turn the LED on pin 21 OFF.<br>");
client.print("Click <a href=\"/L\">here</a> to turn the LED on pin 21 ON.<br>");
```

You can monitor your server's performance using the Serial Monitor.



Take the IP address shown in the Serial Monitor and enter it in your browser. You will see a page with links that can turn the built-in LED of your XIAO ON and OFF.



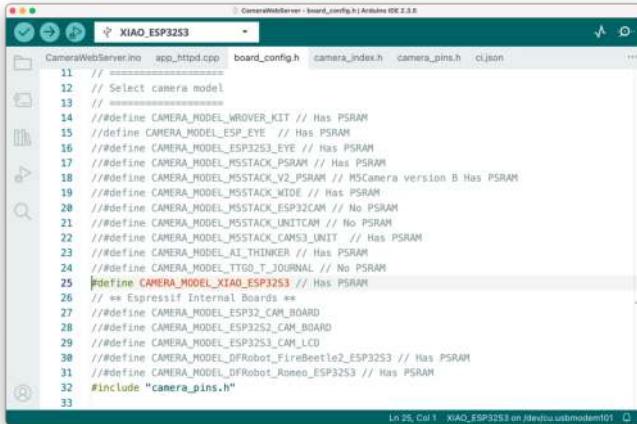
Using the CameraWebServer

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

On the board_config.h tab, comment on all cameras' models, except the XIAO model pins:

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

Do not forget to check the Tools to see if PSRAM is enabled.

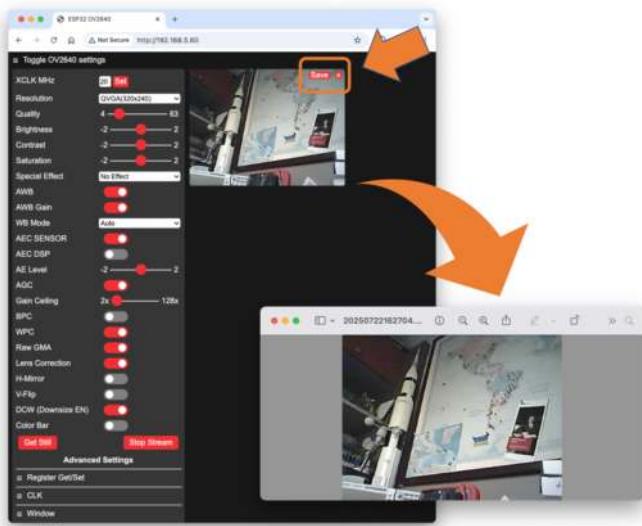


As done before, in the CameraWebServer.ino tab, enter your wifi credentials and upload the code to the device.

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

```
WiFi connecting....
WiFi connected
Camera Ready! Use 'http://192.168.5.60' to connect
```

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



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

Testing the IMU Sensor (LSM6DS3TR-C)

An **Inertial Measurement Unit (IMU)** is a sensor that measures motion and orientation. The LSM6DS3TR-C on your XIAOMI kit is a **6-axis IMU**, meaning it combines:

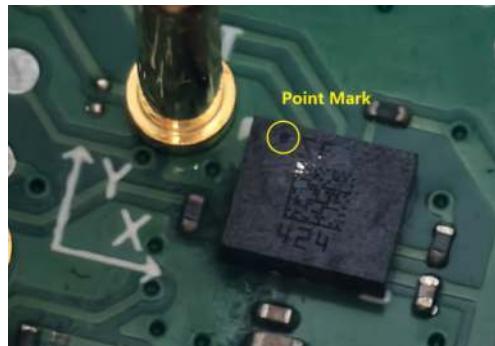
- **3-axis Accelerometer:** Measures linear acceleration (including gravity) along X, Y, and Z axes
- **3-axis Gyroscope:** Measures angular velocity (rotation rate) around X, Y, and Z axes

Technical Specifications:

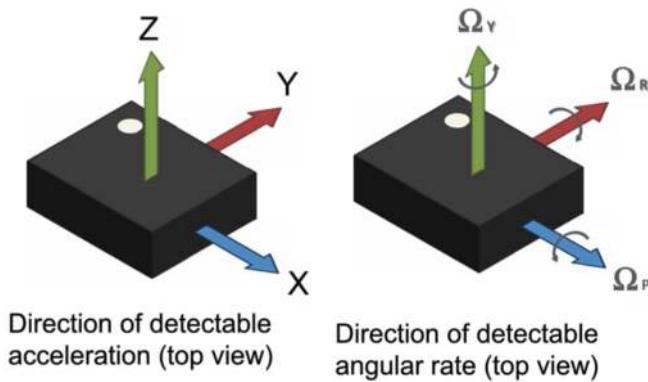
- **Communication:** I2C interface at address 0x6A
- **Accelerometer Range:** $\pm 2/\pm 4/\pm 8/\pm 16$ g (we use ± 2 g by default)
- **Gyroscope Range:** $\pm 125/\pm 250/\pm 500/\pm 1000/\pm 2000$ dps (we use ± 250 dps by default)
- **Resolution:** 16-bit ADC
- **Power Consumption:** Ultra-low power design

Coordinate System:

The sensor follows a right-hand coordinate system. When looking at the IMU sensor with the point mark visible (Expansion Board bottom view):



- **X-axis:** Points to the right
- **Y-axis:** Points forward (away from you)
- **Z-axis:** Points upward (out of the board)

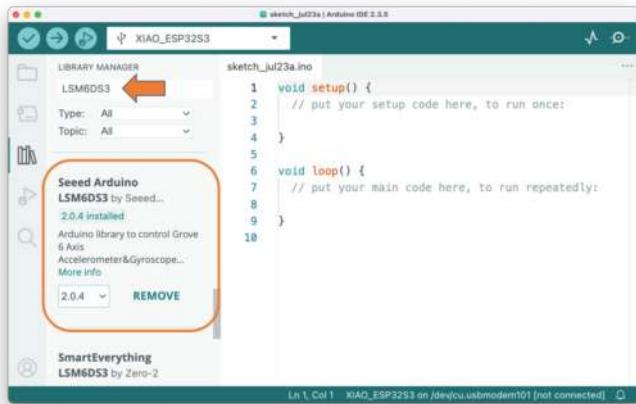


Required Libraries

Before uploading the code, install the required library:

1. Open the **Arduino IDE** and select **Manage Libraries** (represented by the Books Icon).

2. For the IMU library, enter “*LSM6DS3*”, and select “**Seeed Arduino LSM6DS3 by Seeed**”. You can INSTALL or UPDATE the board support packages.



Important: Do NOT install “*Arduino_LSM6DS3 by Arduino*” - that’s for different boards!

Test Code

Enter with the code below at the Arduino IDE and uploaded it to Kit:

```
#include <LSM6DS3.h>  
#include <Wire.h>  
  
// Create IMU object using I2C interface  
// LSM6DS3TR-C sensor is located at I2C address 0x6A  
LSM6DS3 myIMU(I2C_MODE, 0x6A);  
  
// Variables to store sensor readings  
float accelX, accelY, accelZ; // Accelerometer values (g-force)  
float gyroX, gyroY, gyroZ; // Gyroscope values (degrees per second)  
  
void setup() {  
    // Initialize serial communication at 115200 baud rate  
    Serial.begin(115200);  
  
    // Wait for serial port to connect (useful for debugging)
```

```
while (!Serial) {
    delay(10);
}

Serial.println("XIAOMI Kit IMU Test");
Serial.println("LSM6DS3TR-C 6-Axis IMU Sensor");
Serial.println("=====");

// Initialize the IMU sensor
if (myIMU.begin() != 0) {
    Serial.println("ERROR: IMU initialization failed!");
    Serial.println("Check connections and I2C address");
    while(1) {
        delay(1000); // Halt execution if IMU fails to initialize
    }
} else {
    Serial.println(" IMU initialized successfully");
    Serial.println();

    // Print sensor information
    Serial.println("Sensor Information:");
    Serial.println("- Accelerometer range: ±2g");
    Serial.println("- Gyroscope range: ±250 dps");
    Serial.println("- Communication: I2C at address 0x6A");
    Serial.println();

    // Print data format explanation
    Serial.println("Data Format:");
    Serial.println("AccelX,AccelY,AccelZ,GyroX,GyroY,GyroZ");
    Serial.println("Units: g-force (m/s²), degrees/second");
    Serial.println();

    delay(2000); // Brief pause before starting measurements
}
}

void loop() {
    // Read accelerometer data (in g-force units)
    accelX = myIMU.readFloatAccelX();
    accelY = myIMU.readFloatAccelY();
    accelZ = myIMU.readFloatAccelZ();

    // Read gyroscope data (in degrees per second)
```

```
gyroX = myIMU.readFloatGyroX();
gyroY = myIMU.readFloatGyroY();
gyroZ = myIMU.readFloatGyroZ();

// Print readable format to Serial Monitor
Serial.print("Accelerometer (g): ");
Serial.print("X="); Serial.print(accelX, 3);
Serial.print(" Y="); Serial.print(accely, 3);
Serial.print(" Z="); Serial.print(accelZ, 3);

Serial.print(" | Gyroscope (°/s): ");
Serial.print("X="); Serial.print(gyroX, 2);
Serial.print(" Y="); Serial.print(gyroY, 2);
Serial.print(" Z="); Serial.print(gyroZ, 2);
Serial.println();

// Print CSV format for Serial Plotter
Serial.println(String(accelX) + "," + String(accely) + "," +
String(accelZ) + "," + String(gyroX) + "," +
String(gyroY) + "," + String(gyroZ));

// Update rate: 10 Hz (100ms delay)
delay(100);
}
```

The Serial monitor will show the values, and the plotter will show their variation over time. For example, by moving the Kit over the **y-axis**, we will see that value 2 (red line) changes accordingly. Note that **z-axis** is represented by value 3 (green line), which is near 1.0g. The blue line (value 1) is related to the **x-axis**.



You can select the values 4 to 6 to see the Gyroscope behavior.

Testing the OLED Display (SSD1306)

OLED (Organic Light-Emitting Diode) displays are self-illuminating screens where each pixel produces its own light. The XIAO ML kit features a compact 0.42-inch monochrome OLED display, ideal for displaying sensor data, status information, and simple graphics.

Technical Specifications:

- **Size:** 0.42 inches diagonal
- **Resolution:** 72 × 40 pixels
- **Controller:** SSD1306
- **Interface:** I2C at address 0x3C
- **Colors:** Monochrome (black pixels on white background, or vice versa)
- **Viewing:** High contrast, visible in bright light
- **Power:** Low power consumption, no backlight needed

Display Characteristics:

- **Pixel-perfect:** Each of the 2,880 pixels (72×40) can be individually controlled

- **Fast refresh:** Suitable for animations and real-time data
- **No ghosting:** Instant pixel response
- **Wide viewing angle:** Clear from multiple viewing positions

Required Libraries

Before uploading the code, install the required library:

1. Open the **Arduino IDE** and select the “Manage Libraries” (represented by the Books Icon).
2. Enter **u8g2** and select **U8g2 by oliver**. You can install or update the board support packages.

Note: U8g2 is a powerful graphics library supporting many display types



The **U8g2** library is a monochrome graphics library with these features:

- Support for many display controllers (including SSD1306)
- Text rendering with various fonts
- Drawing primitives (lines, rectangles, circles)
- Memory-efficient page-based rendering
- Hardware and software I2C support

Test Code

Enter with the code below at the Arduino IDE and uploaded it to Kit:

```
#include <U8g2lib.h>
#include <Wire.h>

// Initialize the OLED display
// SSD1306 controller, 72x40 resolution, I2C interface
U8G2_SSD1306_72X40_ER_1_HW_I2C u8g2(U8G2_R2, U8X8_PIN_NONE);

void setup() {
    Serial.begin(115200);

    Serial.println("XIAOML Kit - Hello World");
    Serial.println("=====");

    // Initialize the display
    u8g2.begin();

    Serial.println(" Display initialized");
    Serial.println("Showing Hello World message...");

    // Clear the display
    u8g2.clearDisplay();
}

void loop() {
    // Start drawing sequence
    u8g2.firstPage();
    do {
        // Set font
        u8g2.setFont(u8g2_font_ncenB08_tr);

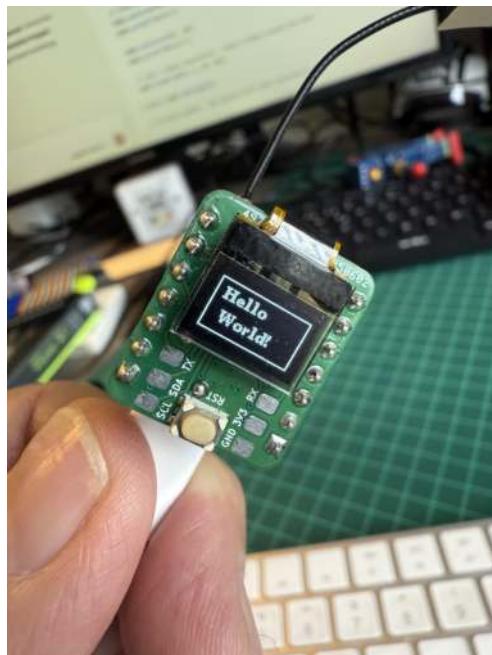
        // Display "Hello World" centered
        u8g2.setCursor(8, 15);
        u8g2.print("Hello");

        u8g2.setCursor(12, 30);
        u8g2.print("World!");

        // Add a simple decoration - draw a frame around the text
        u8g2.drawFrame(2, 2, 68, 36);
    }
}
```

```
    } while (u8g2.nextPage());  
  
    // No delay needed - the display will show continuously  
}
```

If everything works fine, you should see at the display, “Hello World” inside a rectangle.



OLED - Text Sizes and Positioning

- Note that the text is positioned with `setCursor(x, y)`, in this case centered:

```
u8g2.setCursor(8, 15);
```

- The font used in the code was medium.

```
u8g2.setFont(u8g2_font_ncenB08_tr);
```

But other font sizes are available:

- `u8g2_font_4x6_tr`: Tiny font (4×6 pixels)
- `u8g2_font_6x10_tr`: Small font (6×10 pixels)
- `u8g2_font_ncenB08_tr`: Medium bold font
- `u8g2_font_ncenB14_tr`: Large bold font

Shapes

The code added a simple decoration, drawing a frame around the text

```
u8g2.drawFrame(2, 2, 68, 36);
```

But other shapes are available:

- **Rectangle outline:** `drawFrame(x, y, width, height)`
- **Filled rectangle:** `drawBox(x, y, width, height)`
- **Circle:** `drawCircle(x, y, radius)`
- **Line:** `drawLine(x1, y1, x2, y2)`
- **Individual pixels:** `drawPixel(x, y)`

Coordinates

The display uses a coordinate system where:

- **Origin (0,0):** Top-left corner
- **X-axis:** Increases from left to right (0 to 71)
- **Y-axis:** Increases from top to bottom (0 to 39)
- **Text positioning:** `setCursor(x, y)` where y is the baseline of text

Display Rotation

- You can change the rotation parameter by using:
 - `U8G2_R0`: Normal orientation
 - `U8G2_R1`: 90° clockwise
 - `U8G2_R2`: 180° (upside down)
 - `U8G2_R3`: 270° clockwise

Custom Characters:

```
// Draw custom bitmap
static const unsigned char myBitmap[] = {0x00, 0x3c, 0x42, 0x42, 0x3c, 0x00};
u8g2.drawBitmap(x, y, 1, 6, myBitmap);
```

Text Measurements:

```
int width = u8g2.getStrWidth("Hello"); // Get text width  
int height = u8g2.getAscent(); // Get font height
```

The OLED display is now ready to show your sensor data, system status, or any custom graphics you design for your ML projects!

Summary

The XIAOML Kit with ESP32S3 Sense represents a powerful, yet accessible entry point into the world of TinyML and embedded machine learning. Through this setup process, we have systematically tested every component of the XIAOML Kit, confirming that all sensors and peripherals are functioning correctly. The ESP32S3's dual-core processor and 8MB of PSRAM provide sufficient computational power for real-time ML inference, while the OV2640 camera, digital microphone, LSM6DS3TR-C IMU, and 0.42" OLED display create a complete multimodal sensing platform. WiFi connectivity opens possibilities for edge-to-cloud ML workflows, and our Arduino IDE development environment is now properly configured with all necessary libraries.

Beyond mere functionality tests, we've gained practical insights into coordinate systems, data formats, and operational characteristics of each sensor—knowledge that will prove invaluable when designing ML data collection and preprocessing pipelines for the upcoming projects.

This setup process demonstrates key principles that extend far beyond this specific kit. Working with the ESP32S3's memory limitations and processing capabilities provides an authentic experience with the resource constraints inherent in edge AI—the same considerations that apply when deploying models on smartphones, IoT devices, or autonomous systems. Having multiple modalities (vision, audio, motion) on a single platform enables exploration of multimodal ML approaches, which are increasingly important in real-world AI applications.

Most importantly, from raw sensor data to model inference to user feedback via the OLED display, the kit provides a complete ML deployment cycle in miniature, mirroring the challenges faced in production AI systems.

With this foundation in place, you're now equipped to tackle the core TinyML applications in the following chapters:

- **Vision Projects:** Leveraging the camera for image classification and object detection
- **Audio Projects:** Processing audio streams for keyword spotting and voice recognition
- **Motion Projects:** Using IMU data for activity recognition and anomaly detection

Each application will build upon the hardware understanding and software infrastructure we've established, demonstrating how artificial intelligence can be deployed not just in data centers, but in resource-constrained devices that directly interact with the physical world.

The principles encountered with this kit—real-time processing, sensor fusion, and edge inference—are the same ones driving the future of AI deployment in autonomous vehicles, smart cities, medical devices, and industrial automation. By completing this setup successfully, you're now prepared to explore this exciting frontier of embedded machine learning.

Resources

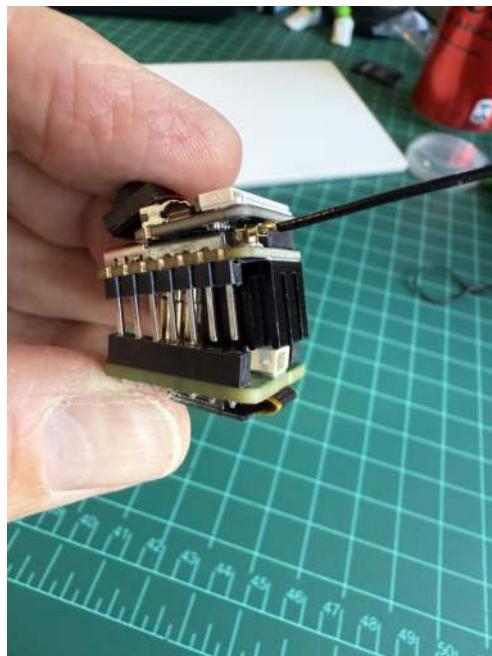
- XIAOML Kit Code
- XIAO ESP32S3 Sense manual & example code
- Usage of Seeed Studio XIAO ESP32S3 microphone
- File System and XIAO ESP32S3 Sense
- Camera Usage in Seeed Studio XIAO ESP32S3 Sense

Appendix

Heat Sink Considerations

If you need to use the XIAO ESP32S3 Sense for camera applications WITHOUT the Expansion Board, you may install the heat sink.

Note that having the heat sink installed, it is not possible to connect the XIAO ESP32S3 Sense with the Expansion Board.



Installing the Heat Sink

To ensure optimal cooling for your XIAO ESP32S3 Sense, you should install the provided heat sink during camera applications. Its design is specifically tailored to address cooling needs, particularly during intensive operations such as camera usage.

Two heat sinks are included in the kit, but you can use only one to guarantee access to the Battery pins.

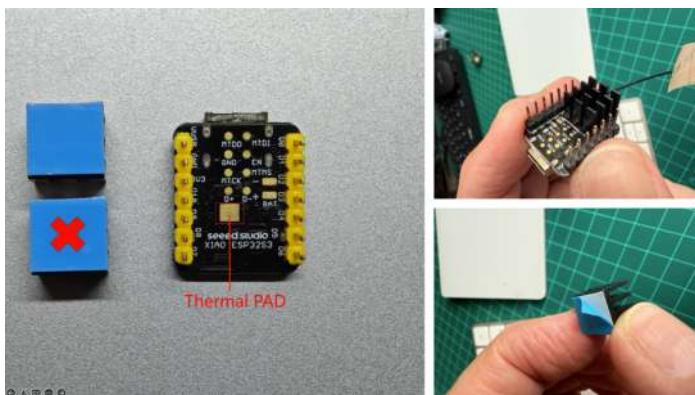
Installation:

- Ensure your device is powered off and unplugged from any power source before you start.
- Prioritize covering the Thermal PAD with the heat sink, as it is directly above the ESP32S3 chip, the primary source of heat. Proper alignment ensures optimal heat dissipation, and **it is essential to keep the BAT pins as unobstructed as possible**.

Now, let's begin the installation process:

Step 1. Prepare the Heat Sink: Start by removing the protective cover from the heat sink to expose the thermal adhesive. This will prepare the heat sink for a secure attachment to the ESP32S3 chip.

Step 2. Assemble the Heat Sink:



After installation, ensure everything is properly secured with no risk of short circuits. Verify that the heat sink is properly aligned and securely attached.

If one heat synk is not enough, a second one can be installed, sharing both the thermal pad, but in this situation, be aware that all pins became unavailable.

Attention

Remove carefully the heat sinks before using the IMU expansion board again

Image Classification

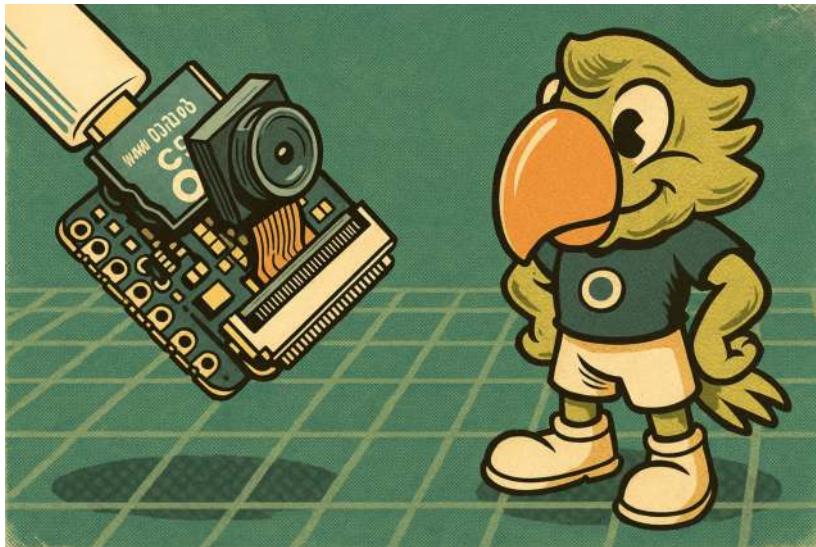


Figure 1.14: *DALL-E prompt - 1950s style cartoon illustration based on a real image by Marcelo Rovai*

Overview

We are increasingly facing an artificial intelligence (AI) revolution, where, as Gartner states, **Edge AI and Computer Vision** have a very high impact potential, and **it is for now!**

When we look into Machine Learning (ML) applied to vision, the first concept that greets us is **Image Classification**, a kind of ML's *Hello World* that is both simple and profound!

The Seeed Studio XIAO ML Kit provides a comprehensive hardware solution centered around the XIAO ESP32-S3 Sense, featuring an integrated **OV3660** camera and SD card support. Those features make the XIAO ESP32S3 Sense an excellent starting point for exploring TinyML vision AI.

In this Lab, we will explore Image Classification using the non-code tool **SenseCraft AI** and explore a more detailed development with **Edge Impulse Studio** and **Arduino IDE**.

Learning Objectives

- **Deploy Pre-trained Models** using SenseCraft AI Studio for immediate computer vision applications
- **Collect and Manage Image Datasets** for custom classification tasks with proper data organization
- **Train Custom Image Classification Models** using transfer learning with MobileNet V2 architecture
- **Optimize Models for Edge Deployment** through quantization and memory-efficient preprocessing
- **Implement Post-processing Pipelines**, including GPIO control and real-time inference integration
- **Compare Development Approaches** between no-code and advanced ML platforms for embedded applications

Image Classification

Image classification is a fundamental task in computer vision that involves categorizing entire images into one of several predefined classes. This process entails analyzing the visual content of an image and assigning it a label from a fixed set of categories based on the dominant object or scene it depicts.

Image classification is crucial in various applications, ranging from organizing and searching through large databases of images in digital libraries and social media platforms to enabling autonomous systems to comprehend their surroundings. Common architectures that have

significantly advanced the field of image classification include Convolutional Neural Networks (CNNs), such as AlexNet, VGGNet, and ResNet. These models have demonstrated remarkable accuracy on challenging datasets, such as ImageNet, by learning hierarchical representations of visual data.

As the cornerstone of many computer vision systems, image classification drives innovation, laying the groundwork for more complex tasks like object detection and image segmentation, and facilitating a deeper understanding of visual data across various industries. So, let's start exploring the Person Classification model ("Person - No Person"), a ready-to-use computer vision application on the SenseCraft AI.

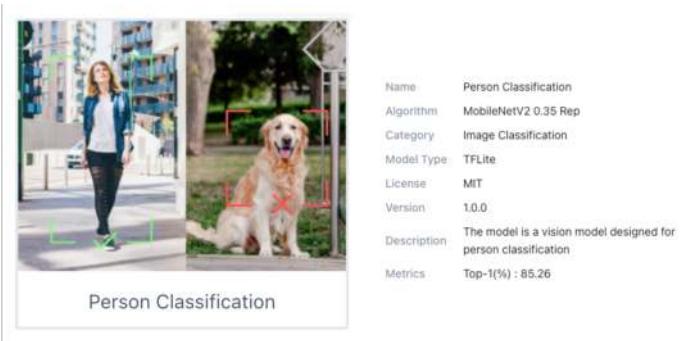
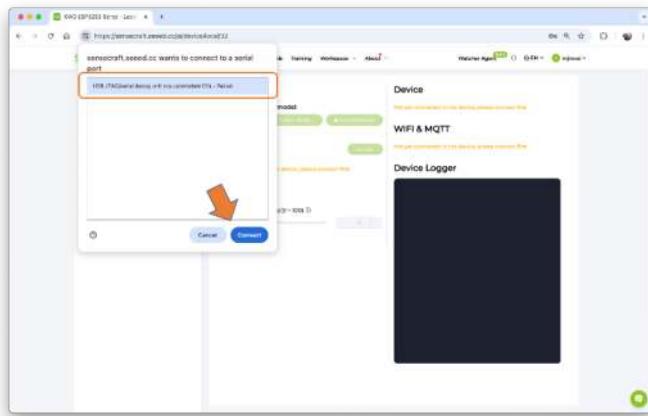
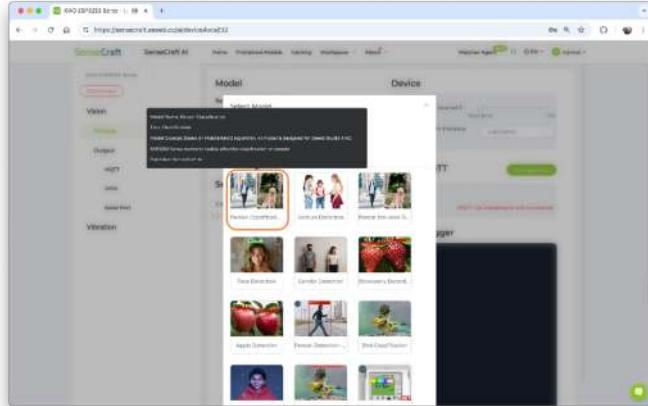


Image Classification on the SenseCraft AI Workspace

Start by connecting the XIAOML Kit (or just the XIAO ESP32S3 Sense, disconnected from the Expansion Board) to the computer via USB-C, and then open the SenseCraft AI Workspace to connect it.



Once connected, select the option [Select Model...] and enter in the search window: “Person Classification”. From the options available, select the one trained over the MobileNet V2 (passing the mouse over the models will open a pop-up window with its main characteristics).

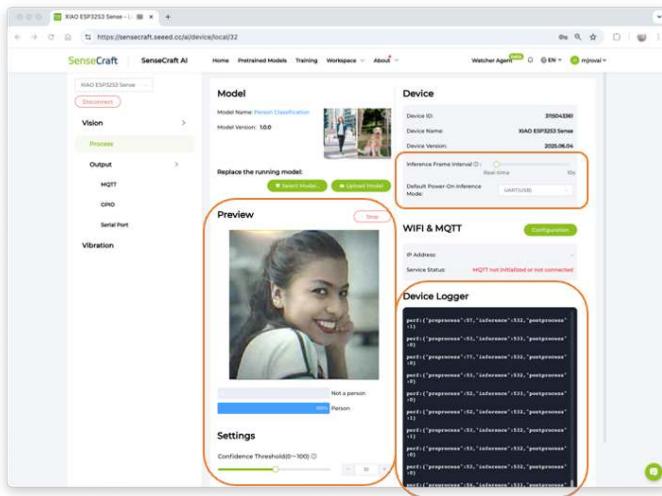


Click on the chosen model and confirm the deployment. A new firmware for the model should start uploading to our device.

Note that the percentage of models downloaded and firmware uploaded will be displayed. If not, try disconnecting the device, then reconnect it and press the boot button.

After the model is uploaded successfully, we can view the live feed from the XIAO camera and the classification result (Person or Not a Person) in the **Preview** area, along with the inference details displayed in the **Device Logger**.

Note that we can also select our **Inference Frame Interval**, from “Real-Time” (Default) to 10 seconds, and the **Mode** (UART, I2C, etc) as the data is shared by the device (the default is UART via USB).



At the Device Logger, we can see that the latency of the model is from 52 to 78 ms for pre-processing and around 532ms for inference, which will give us a total time of a little less than 600ms, or about **1.7 Frames per second (FPS)**.

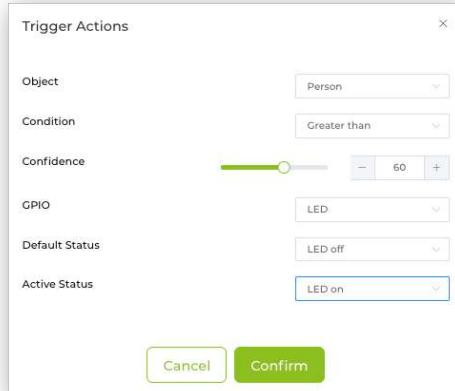
To run the Mobilenet V2 0.35, the XIAO had a peak current of 160mA at 5.23V, resulting in a **power consumption of 830mW**.

Post-Processing

An essential step in an Image Classification project pipeline is to define what we want to do with the inference result. So, imagine that we will use the XIAO to automatically turn on the room lights if a person is detected.



With the SebseCraft AI, we can do it on the `Output -> GPIO` section. Click on the Add icon to trigger the action when event conditions are met. A pop-up window will open, where you can define the action to be taken. For example, if a person is detected with a confidence of more than 60% the internal LED should be ON. In a real scenario, a GPIO, for example, D0, D1, D2, D11, or D12, would be used to trigger a relay to turn on a light.



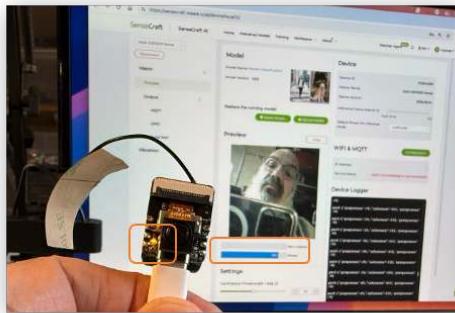
Once confirmed, the created **Trigger Action** will be shown. Press **Send** to upload the command to the XIAO.

Output data through GPIO						
Trigger Actions						
Object	GPIO	Condition	Confidence	Default Status	Active Status	Operation
Person	LED	Greater than	62	LED off	LED on	

Trigger action when event conditions are met

Delete Send

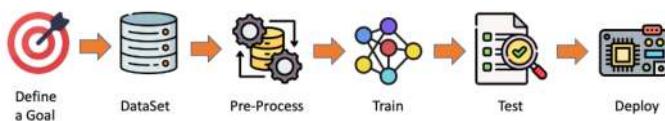
Now, pointing the XIAO at a person will make the internal LED go ON.



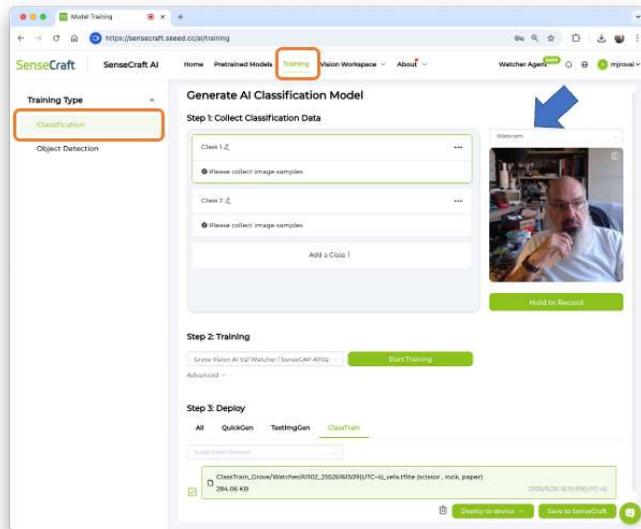
We will explore more trigger actions and post-processing techniques further in this lab.

An Image Classification Project

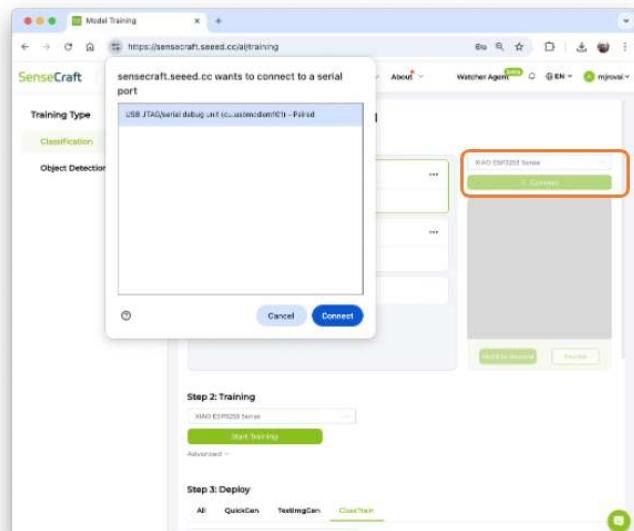
Let's create a simple Image Classification project using SenseCraft AI Studio. Below, we can see a typical machine learning pipeline that will be used in our project.



On SenseCraft AI Studio: Let's open the tab Training:



The default is to train a Classification model with a WebCam if it is available. Let's select the XIAOESP32S3 Sense instead. Pressing the green button [Connect] will cause a Pop-Up window to appear. Select the corresponding Port and press the blue button [Connect].



The image streamed from the Grove Vision AI V2 will be displayed.

The Goal

The first step, as we can see in the ML pipeline, is to define a goal. Let's imagine that we have an industrial installation that should automatically sort wheels and boxes.



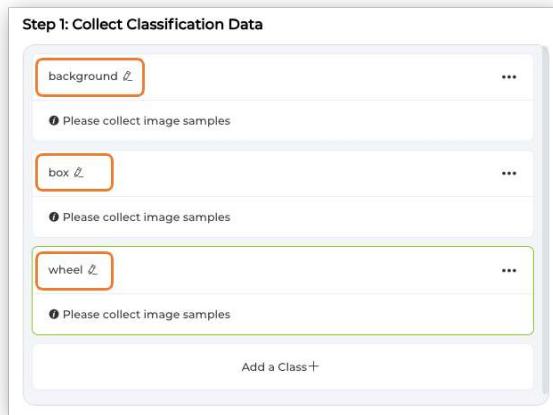
So, let's simulate it, classifying, for example, a toy box and a toy wheel. We should also include a 3rd class of images, background, where there are no objects in the scene.



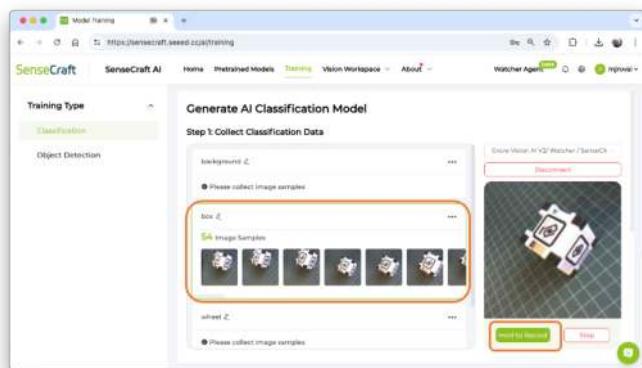
Data Collection

Let's create the classes, following, for example, an alphabetical order:

- Class1: background
- Class 2: box
- Class 3: wheel



Select one of the classes and keep pressing the green button (Hold to Record) under the preview area. The collected images (and their counting) will appear on the Image Samples Screen. Carefully and slowly, move the camera to capture different angles of the object. To modify the position or interfere with the image, release the green button, rearrange the object, and then hold it again to resume the capture.



After collecting the images, review them and delete any incorrect ones.

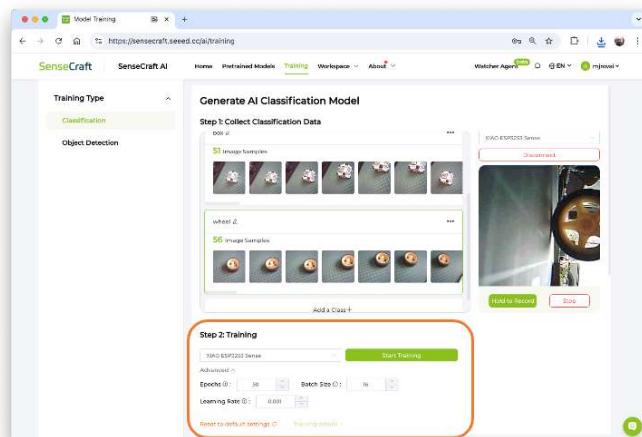


Collect around **50 images** from each class and go to Training Step.

Note that it is possible to download the collected images to be used in another application, for example, with the Edge Impulse Studio.

Training

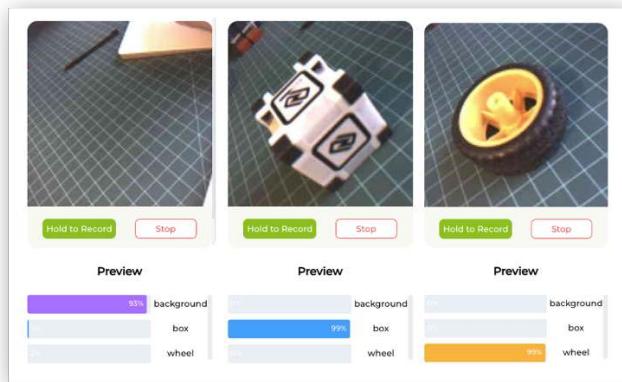
Confirm if the correct device is selected (XIAO ESP32S3 Sense) and press [Start Training]



Test

After training, the inference result can be previewed.

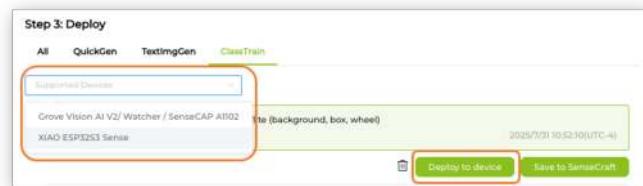
Note that the model is not running on the device. We are, in fact, only capturing the images with the device and performing a **live preview** using the training model, which is running in the Studio.



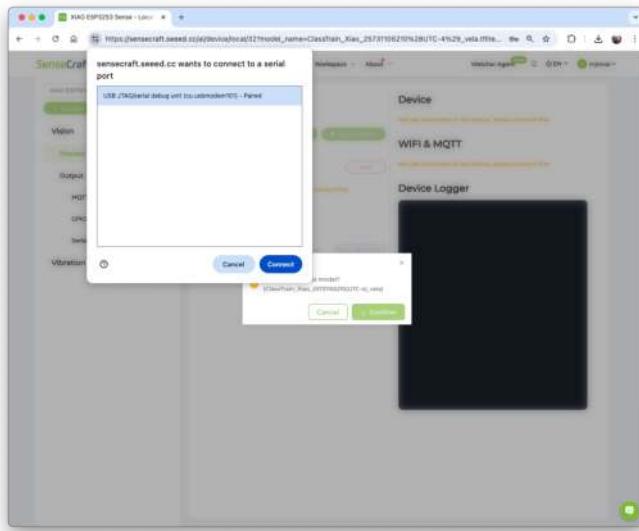
Now is the time to really deploy the model in the device.

Deployment

Select the trained model and XIAO ESP32S3 Sense at the Supported Devices window. And press [Deploy to device].

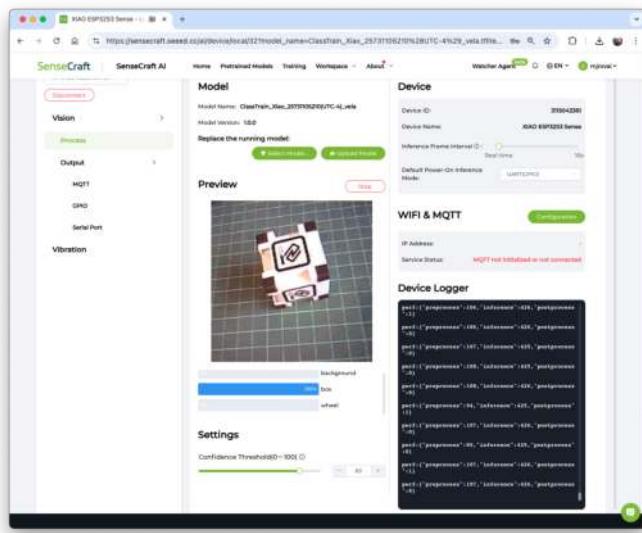


The SeneCraft AI will redirect us to the **Vision Workplace** tab. Confirm the deployment, select the Port, and Connect it.



The model will be flashed into the device. After an automatic reset, the model will start running on the device. On the Device Logger, we can see that the inference has a **latency of approximately 426 ms**, plus a **pre-processing of around 110ms**, corresponding to a **frame rate of 1.8 frames per second (FPS)**.

Also, note that in **Settings**, it is possible to adjust the model's confidence.



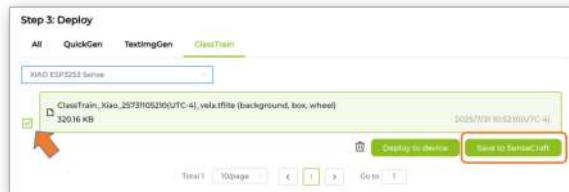
To run the Image Classification Model, the XIAO ESP32S3 had a peak current of 14mA at 5.23V, resulting in a **power consumption of 730mW**.

As before, in the **Output → GPIO**, we can turn the GPIOs or the Internal LED ON based on the detected class. For example, the LED will be turned ON when the wheel is detected.

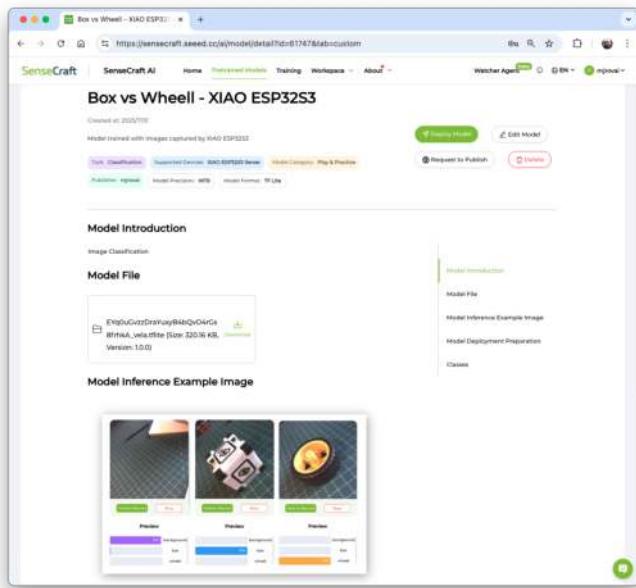


Saving the Model

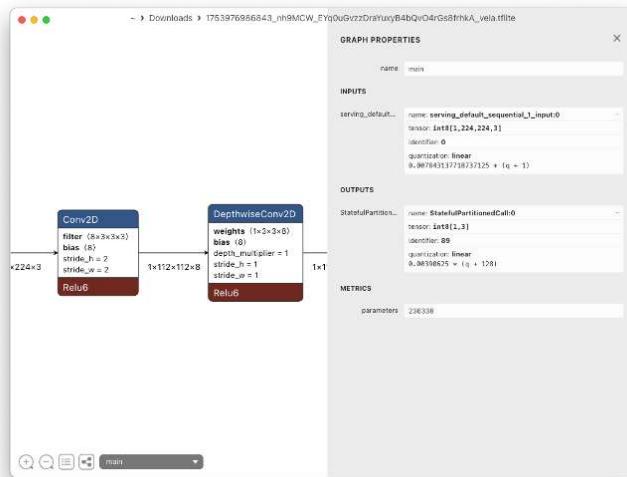
It is possible to save the model in the SenseCraft AI Studio. The Studio will retain all our models for later deployment. For that, return to the Training tab and select the button [Save to SenseCraft]:



Follow the instructions to enter the model's name, description, image, and other details.



Note that the trained model (an Int8 MobileNet V2 with a size of 320KB) can be downloaded for further use or even analysis, for example, using Netron. Note that the model uses images of size 224x224x3 as its Input Tensor. In the next step, we will use different hyperparameters on the Edge Impulse Studio.



Also, the model can be deployed again to the device at any time. Automatically, the **Workspace** will be open on the SenseCraft AI.

Image Classification Project from a Dataset

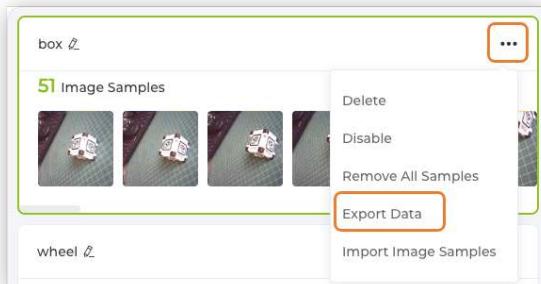
The primary objective of our project is to train a model and perform inference on the XIAO ESP32S3 Sense. For training, we should find some data (**in fact, tons of data!**).

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

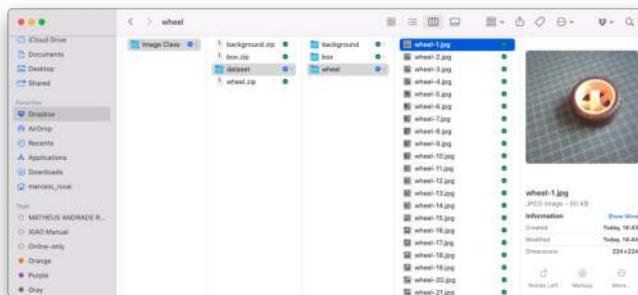
With TinyML, a set of techniques associated with machine learning inference on embedded devices, we should limit the classification to three or four categories due to limitations (mainly memory). We can, for example, train the images captured for the Box versus Wheel, which can be downloaded from the SenseCraft AI Studio.

Alternatively, we can use a completely new dataset, such as one that differentiates apples from bananas and potatoes, or other categories. If possible, try finding a specific dataset that includes images from those categories. Kaggle fruit-and-vegetable-image-recognition is a good start.

Let's download the dataset captured in the previous section. Open the menu (3 dots) on each of the captured classes and select Export Data.



The dataset will be downloaded to the computer as a .ZIP file, with one file for each class. Save them in your working folder and unzip them. You should have three folders, one for each class.



Optionally, you can add some fresh images, using, for example, the code discussed in the setup lab.

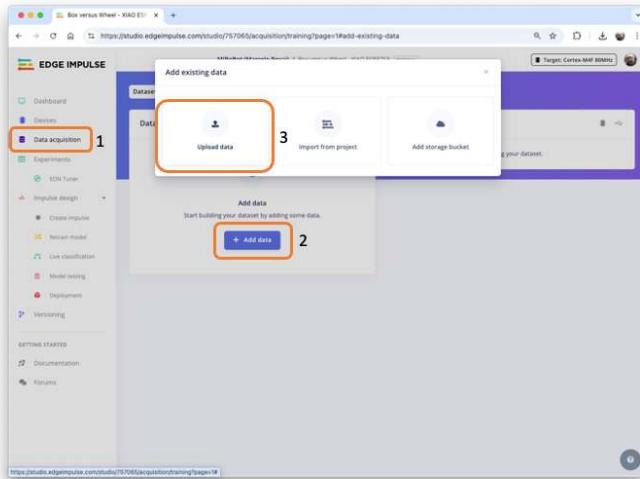
Training the model with Edge Impulse Studio

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

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

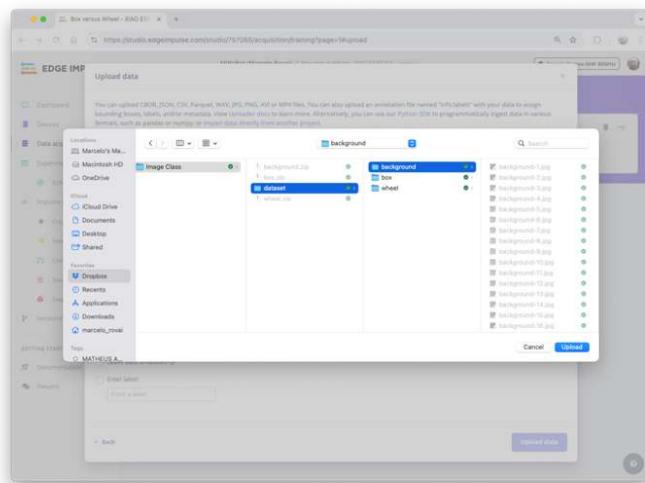
Data Acquisition

Next, go to the **Data acquisition** section and there, select + Add data. A pop-up window will appear. Select UPLOAD DATA.

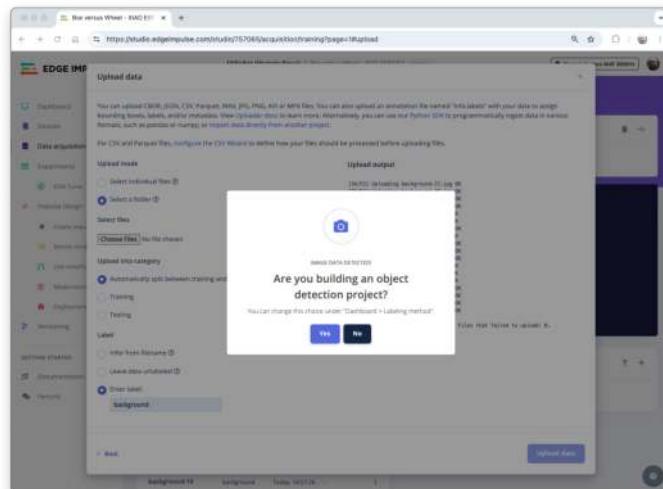


After selection, a new Pop-Up window will appear, asking to update the data.

- In Upload mode: select a folder and press [Choose Files].
- Go to the folder that contains one of the classes and press [Upload]

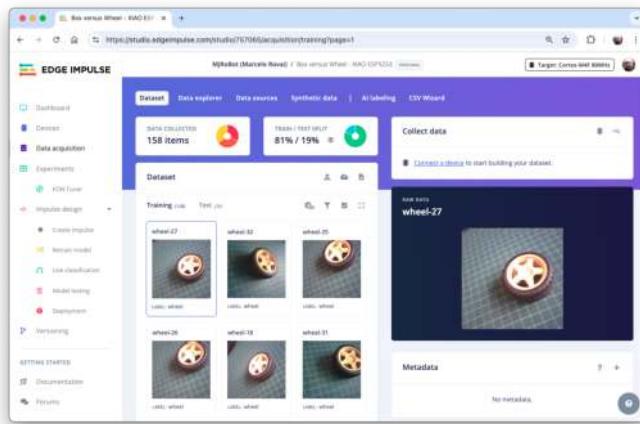


- You will return automatically to the Upload data window.
 - Select Automatically split between training and testing
 - And enter the label of the images that are in the folder.
 - Select [Upload data]
 - At this point, the files will start to be uploaded, and after that, another Pop-Up window will appear asking if you are building an object detection project. Select [no]



Repeat the procedure for all classes. **Do not forget to change the label's name.** If you forget and the images are uploaded, please note that they will be mixed in the Studio. Do not worry, you can manually move the data between classes further.

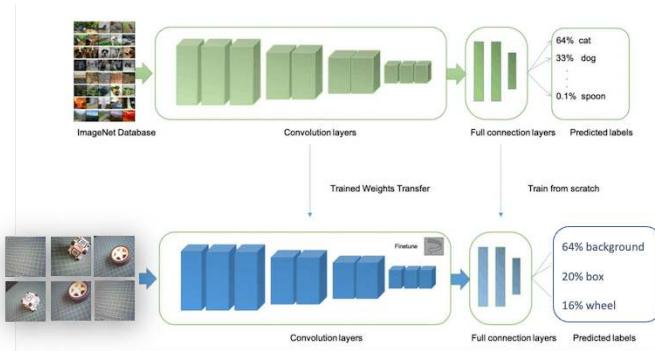
Close the Upload Data window and return to the **Data acquisition** page. We can see that all dataset was uploaded. Note that on the upper panel, we can see that we have 158 items, all of which are balanced. Also, 19% of the images were left for testing.



Impulse Design

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

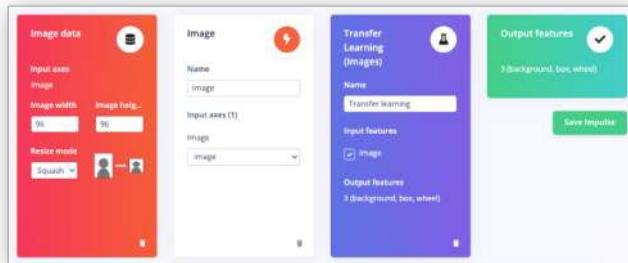
Classifying images is the most common application of deep learning, but a substantial amount of data is required to accomplish this task. We have around 50 images for each category. Is this number enough? Not at all! We will need thousands of images to "teach" or "model" each class, allowing us to differentiate them. However, we can resolve this issue by retraining a previously trained model using thousands of images. We refer to this technique as "**Transfer Learning**" (TL). With TL, we can fine-tune a pre-trained image classification model on our data, achieving good performance even with relatively small image datasets, as in our case.



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

So, starting from the raw images, we will resize them (96×96) Pixels are fed to our Transfer Learning block. Let's create an Impulse.

At this point, we can also define our target device to monitor our "budget" (memory and latency). The XIAO ESP32S3 is not officially supported by Edge Impulse, so let's consider the Espressif ESP-EYE, which is similar but slower.



Save the Impulse, as shown above, and go to the **Image** section.

Pre-processing (Feature Generation)

Besides resizing the images, we can convert them to grayscale or retain their original RGB color depth. Let's select [RGB] in the **Image** section. Doing that, each data sample will have a dimension of 27,648 features

(96x96x3). Pressing [Save Parameters] will open a new tab, Generate Features. Press the button [Generate Features] to generate the features.

Model Design, Training, and Test

In 2007, Google introduced MobileNetV1. In 2018, MobileNetV2: Inverted Residuals and Linear Bottlenecks, was launched, and, in 2019, the V3. The Mobilinet is a family of general-purpose computer vision neural networks explicitly designed for mobile devices to support classification, detection, and other applications. MobileNets are small, low-latency, low-power models parameterized to meet the resource constraints of various use cases.

Although the base MobileNet architecture is already compact and has low latency, a specific use case or application may often require the model to be even smaller and faster. MobileNets introduce a straightforward parameter, α (alpha), called the width multiplier to construct these smaller, less computationally expensive models. The role of the width multiplier α is to thin a network uniformly at each layer.

Edge Impulse Studio has available MobileNet V1 (96x96 images) and V2 (96x96 and 160x160 images), with several different α values (from 0.05 to 1.0). For example, you will get the highest accuracy with V2, 160x160 images, and $\alpha=1.0$. Of course, there is a trade-off. The higher the accuracy, the more memory (around 1.3M RAM and 2.6M ROM) will be needed to run the model, implying more latency. The smaller footprint will be obtained at another extreme with MobileNet V1 and $\alpha=0.10$ (around 53.2K RAM and 101K ROM).

We will use the **MobileNet V2 0.35** as our base model (but a model with a greater alpha can be used here). The final layer of our model, preceding the output layer, will have 16 neurons with a 10% dropout rate for preventing overfitting.

Another necessary technique to use with deep learning is **data augmentation**. Data augmentation is a method that can help improve the accuracy of machine learning models 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).

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)

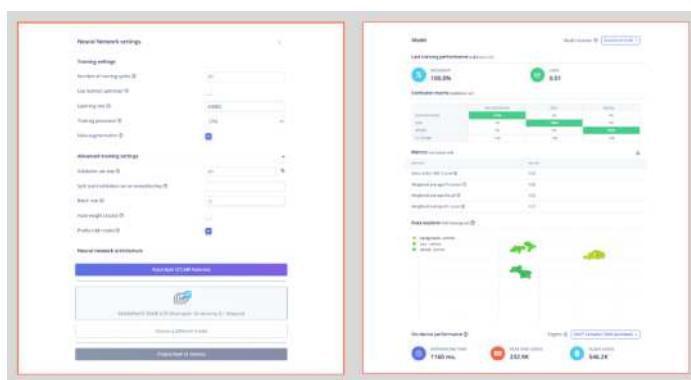
    return image, label

```

Now, let's us define the hyperparameters:

- Epochs: 20,
- Bach Size: 32
- Learning Rate: 0.0005
- Validation size: 20%

And, so, we have as a training result:



The model profile predicts **233 KB of RAM** and **546 KB of Flash**, indicating no problem with the Xiao ESP32S3, which has 8 MB of PSRAM.

Additionally, the Studio indicates a **latency of around 1160 ms**, which is very high. However, this is to be expected, given that we are using the ESP-EYE, whose CPU is an Extensa LX6, and the ESP32S3 uses a newer and more powerful Xtensa LX7.

With the test data, we also achieved 100% accuracy, even with a quantized INT8 model. This result is not typical in real projects, but our project here is relatively simple, with two objects that are very distinctive from each other.

Model Deployment

We can deploy the trained model:

- As .TFLITE to be used on the **SenseCraft AI**
- As an Arduino Library in the **Edge Impulse Studio**.

Let's start with the SenseCraft, which is more straightforward and more intuitive.

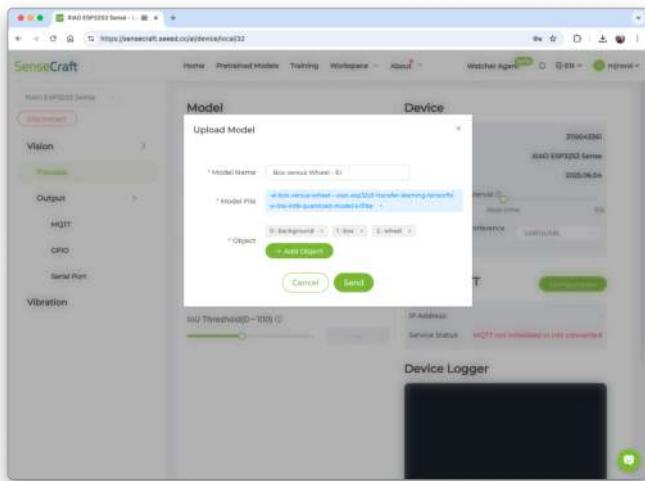
Model Deployment on the SenseCraft AI

On the **Dashboard**, it is possible to download the trained model in several different formats. Let's download TensorFlow Lite (int8 quantized), which has a size of 623KB.



On **SenseCraft AI Studio**, go to the Workspace tab, select XIAO ESP32S3, the corresponding Port, and connect the device.

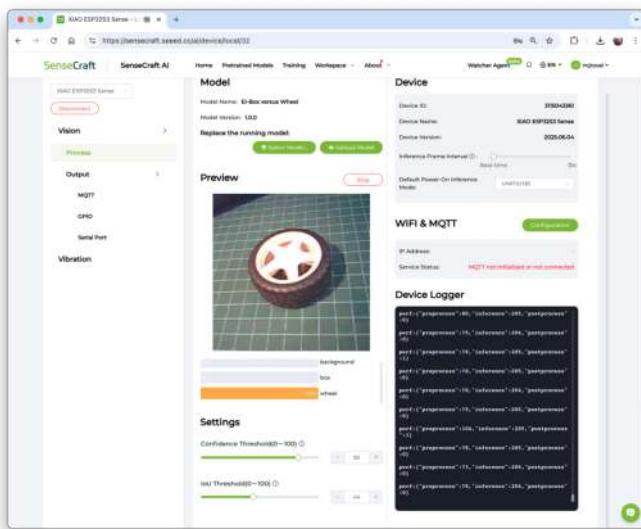
You should see the last model that was uploaded to the device. Select the green button [Upload Model]. A pop-up window will prompt you to enter the model name, the model file, and the class names (**objects**). We should use labels in alphabetical order: 0: background, 1: box, and 2: wheel, and then press [Send].



After a few seconds, the model will be uploaded (“flashed”) to our device, and the camera image will appear in real-time on the **Preview Sector**. The Classification result will be displayed under the image preview. It is also possible to select the **Confidence Threshold** of your inference using the cursor on **Settings**.

On the **Device Logger**, we can view the Serial Monitor, where we can observe the latency, which is approximately 81 ms for pre-processing and 205 ms for inference, **corresponding to a frame rate of 3.4 frames per second (FPS)**, what is double of we got, training the model on SenseCraft, because we are working with smaller images (96x96 versus 224x224).

The total latency is around **4 times faster** than the estimation made in Edge Impulse Studio on an Xtensa LX6 CPU; now we are performing the inference on an Xtensa LX7 CPU.



Post-Processing

It is possible to obtain the output of a model inference, including Latency, Class ID, and Confidence, as shown on the Device Logger in SenseCraft AI. This allows us to utilize the **XIAO ESP32S3 Sense as an AI sensor**. In other words, we can retrieve the model data using different communication protocols such as MQTT, UART, I2C, or SPI, depending on our project requirements.

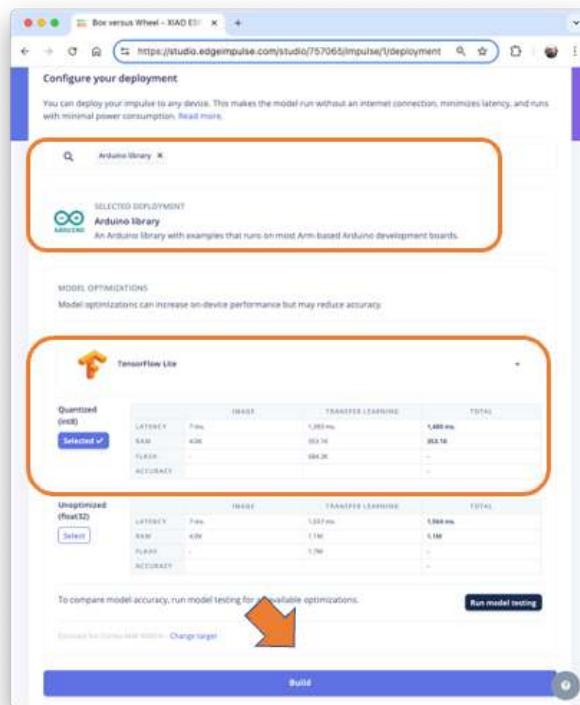
The idea is similar to what we have done on the Seeed Grove Vision AI V2 Image Classification Post-Processing Lab.

Below is an example of a connection using the I2C bus.

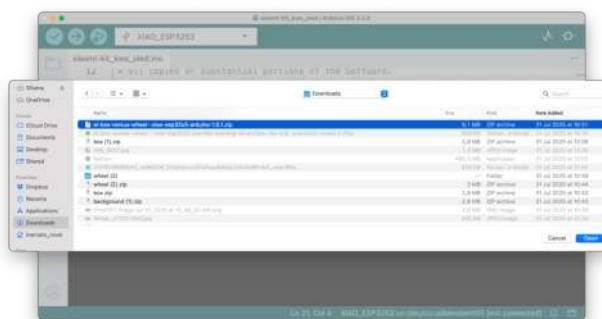
Please refer to the Seeed Studio Wiki for more information.

Model Deployment as an Arduino Library at EI Studio

On the Deploy section at Edge Impulse Studio, Select Arduino library, TensorFlow Lite, Quantized(int8), and press [Build]. The trained model will be downloaded as a .zip Arduino library:



Open your Arduino IDE, and under **Sketch**, go to **Include Library** and **add .ZIP Library**. Next, select the file downloaded from Edge Impulse Studio and press **[Open]**.



Go to the Arduino IDE Examples and look for the project by its name (in this case: "Box_versus_Whell_...Interfering". Open esp32 -> esp32 - camera. The sketch esp32_camera.ino will be downloaded to the IDE.

This sketch was developed for the standard ESP32 and will not work with the XIAO ESP32S3 Sense. It should be modified. Let's download the modified one from the project GitHub: Image_class_XIAOML-Kit.ino.

XIAO ESP32S3 Image Classification Code Explained

The code captures images from the onboard camera, processes them, and classifies them (in this case, "Box", "Wheel", or "Background") using the trained model on EI Studio. It runs continuously, performing real-time inference on the edge device.

In short,:

Camera → JPEG Image → RGB888 Conversion → Resize to 96x96 → Neural Network → Classification Results → Serial Output

Key Components.

1. Library Includes and Dependencies

```
#include <Box_versus_Wheel_-_XIAO_ESP32S3_inferencing.h>
#include "edge-impulse-sdk/dsp/image/image.hpp"
#include "esp_camera.h"
```

- **Edge Impulse Inference Library:** Contains our trained model and inference engine
- **Image Processing:** Provides functions for image manipulation
- **ESP Camera:** Hardware interface for the camera module

2. Camera Pin Configurations

The XIAO ESP32S3 Sense can work with different camera sensors (OV2640 or OV3660), which may have different pin configurations. The code defines three possible configurations:

```
// Configuration 1: Most common OV2640 configuration
#define CONFIG_1_XCLK_GPIO_NUM    10
#define CONFIG_1_SIOD_GPIO_NUM    40
#define CONFIG_1_SIOC_GPIO_NUM    39
// ... more pins
```

This flexibility allows the code to automatically try different pin mappings if the first one doesn't work, making it more robust across different hardware revisions.

3. Memory Management Settings

```
#define EI_CAMERA_RAW_FRAME_BUFFER_COLS 320
#define EI_CAMERA_RAW_FRAME_BUFFER_ROWS 240
#define EI_CLASSIFIER_ALLOCATION_HEAP 1
```

- **Frame Buffer Size:** Defines the raw image size (320x240 pixels)
 - **Heap Allocation:** Uses dynamic memory allocation for flexibility
 - **PSRAM Support:** The ESP32S3 has 8MB of PSRAM for storing large data like images

setup() - Initialization.

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

    if (ei_camera_init() == false) {
        ei_printf("Failed to initialize Camera!\r\n");
    } else {
        ei_printf("Camera initialized\r\n");
    }

    ei_sleep(2000); // Wait 2 seconds before starting
}
```

This function:

1. Initializes serial communication for debugging output
 2. Initializes the camera with automatic configuration detection
 3. Waits 2 seconds before starting continuous inference

loop() - Main Processing Loop. The loop performs these steps continuously:

Step 1: Memory Allocation

Allocates memory for the image buffer, preferring PSRAM (faster external RAM) but falling back to regular heap if needed.

Step 2: Image Capture

```
if (ei_camera_capture((size_t)EI_CLASSIFIER_INPUT_WIDTH,
                      (size_t)EI_CLASSIFIER_INPUT_HEIGHT,
                      snapshot_buf) == false) {
    ei_printf("Failed to capture image\r\n");
    free(snapshot_buf);
    return;
}
```

Captures an image from the camera and stores it in the buffer.

Step 3: Run Inference

```
ei_impulse_result_t result = { 0 };
EI_IMPULSE_ERROR err = run_classifier(&signal, &result, false);
```

Runs the machine learning model on the captured image.

Step 4: Output Results

```
for (uint16_t i = 0; i < EI_CLASSIFIER_LABEL_COUNT; i++) {
    ei_printf(" %s: %.5f\r\n",
              ei_classifier_inferencing_categories[i],
              result.classification[i].value);
}
```

Prints the classification results showing confidence scores for each category.

ei_camera_init() - Smart Camera Initialization. This function implements an intelligent initialization sequence:

```
bool ei_camera_init(void) {
    // Try Configuration 1 (OV2640 common)
    update_camera_config(1);
    esp_err_t err = esp_camera_init(&camera_config);
    if (err == ESP_OK) goto camera_init_success;

    // Try Configuration 2 (OV3660)
    esp_camera_deinit();
    update_camera_config(2);
    err = esp_camera_init(&camera_config);
    if (err == ESP_OK) goto camera_init_success;
```

```
// Continue trying other configurations...
}
```

The function:

1. Tries multiple pin configurations
2. Tests different clock frequencies (10MHz or 16MHz)
3. Attempts PSRAM first, then falls back to DRAM
4. Applies sensor-specific settings based on detected hardware

ei_camera_capture() - Image Processing Pipeline.

```
bool ei_camera_capture(uint32_t img_width, uint32_t img_height, uint8_t ...
    // 1. Get frame from camera
    camera_fb_t *fb = esp_camera_fb_get();

    // 2. Convert JPEG to RGB888 format
    bool converted = fmt2rgb888(fb->buf, fb->len, PIXFORMAT_JPEG, snapshot);

    // 3. Return frame buffer to camera driver
    esp_camera_fb_return(fb);

    // 4. Resize if needed
    if (do_resize) {
        ei::image::processing::crop_and_interpolate_rgb888(...);
    }
}
```

This function:

1. Captures a JPEG image from the camera
2. Converts it to RGB888 format (required by the ML model)
3. Resizes the image to match the model's input size (96x96 pixels)

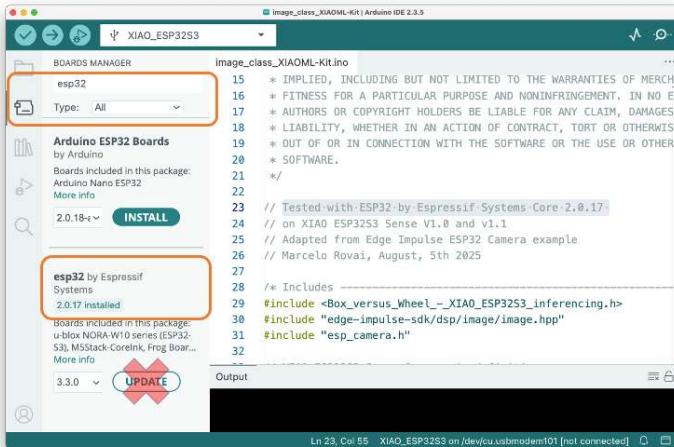
Inference

- Upload the code to the XIAO ESP32S3 Sense.

Attention

- The Xiao ESP32S3 **MUST** have the PSRAM enabled.
You can check it on the Arduino IDE upper menu:
Tools->PSRAM:OPI PSRAM

- The Arduino Boards package (esp32 by Espressif Systems) should be version 2.017. Do not update it



- Open the Serial Monitor
- Point the camera at the objects, and check the result on the Serial Monitor.



Post-Processing

In edge AI applications, the inference result is only as valuable as our ability to act upon it. While serial output provides detailed information for debugging and development, real-world deployments require

immediate, human-readable feedback that doesn't depend on external monitors or connections.

The XIAOML Kit tiny 0.42" OLED display (72×40 pixels) serves as a crucial post-processing component that transforms raw ML inference results into immediate, human-readable feedback—displaying detected class names and confidence levels directly on the device, eliminating the need for external monitors and enabling truly standalone edge AI deployment in industrial, agricultural, or retail environments where instant visual confirmation of AI predictions is essential.

So, let's modify the sketch to automatically adapt to the model trained on Edge Impulse by reading the class names and count directly from the model. The display will show abbreviated class names (3 letters) with larger fonts for better visibility on the tiny 72x40 pixel display. Download the code from the GitHub: [XIAOML-Kit-Img_Class_OLED_Gen](#).

Running the code, we can see the result:



Summary

The XIAO ESP32S3 Sense is a remarkably capable and flexible platform for image classification applications. Through this lab, we've explored two distinct development approaches that cater to different skill levels and project requirements.

- The **SenseCraft AI Studio** provides an accessible entry point with its **no-code interface**, enabling rapid prototyping and deployment of pre-trained models like person detection. With real-time inference and integrated post-processing capabilities, it demonstrates how AI can be deployed without extensive programming or ML knowledge.

- For more advanced applications, **Edge Impulse Studio** offers comprehensive machine learning pipeline tools, including custom dataset management, transfer learning with several pre-trained models, such as MobileNet, and model optimization.

Key insights from this lab include the importance of image resolution trade-offs, the effectiveness of transfer learning for small datasets, and the practical considerations of edge AI deployment, such as power consumption and memory constraints.

The Lab demonstrates fundamental TinyML principles that extend beyond this specific hardware: resource-constrained inference, real-time processing requirements, and the complete pipeline from data collection through model deployment to practical applications. With built-in post-processing capabilities including GPIO control and communication protocols, the XIAO serves as more than just an inference engine—it becomes a complete AI sensor platform.

This foundation in image classification prepares you for more complex computer vision tasks while showcasing how modern edge AI makes sophisticated computer vision accessible, cost-effective, and deployable in real-world embedded applications ranging from industrial automation to smart home systems.

Resources

- Getting Started with the XIAO ESP32S3
- SenseCraft AI Studio Home
- SenseCraft Vision Workspace
- Dataset example
- Edge Impulse Project
- XIAO as an AI Sensor
- Seeed Arduino SSCMA Library
- XIAOML Kit Code

Object Detection

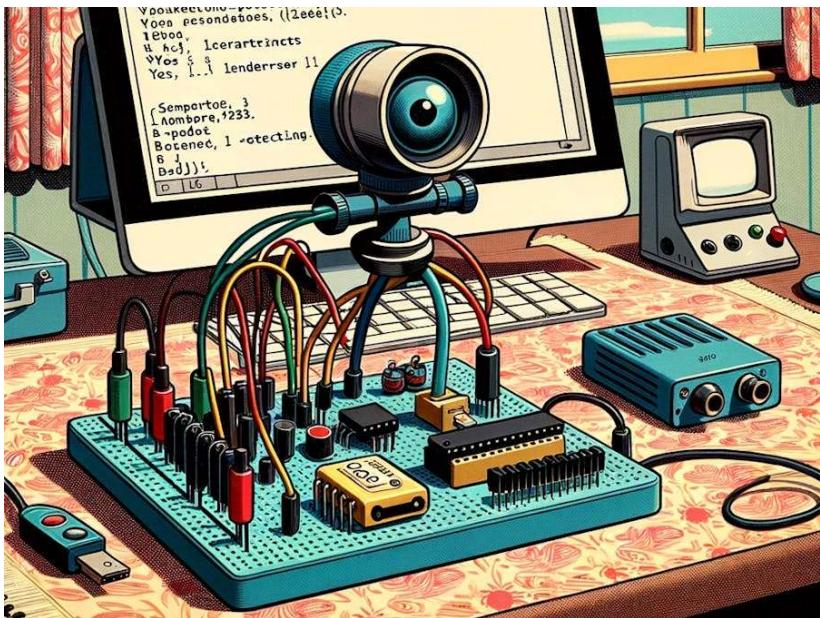


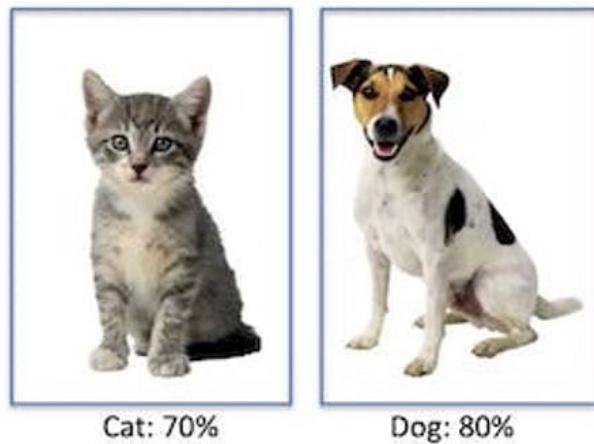
Figure 1.15: DALL-E prompt - Cartoon styled after 1950s animations, showing a detailed board with sensors, particularly a camera, on a table with patterned cloth. Behind the board, a computer with a large back showcases the Arduino IDE. The IDE's content hints at LED pin assignments and machine learning inference for detecting spoken commands. The Serial Monitor, in a distinct window, reveals outputs for the commands 'yes' and 'no'.

Overview

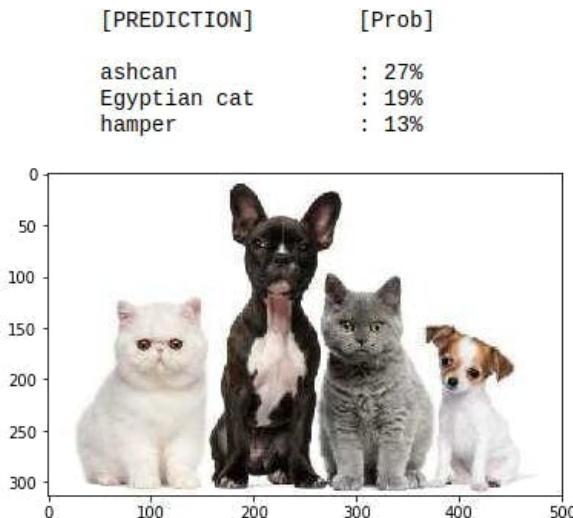
In the last section regarding Computer Vision (CV) and the XIAO ESP32S3, *Image Classification*, we learned how to set up and classify images with this remarkable development board. Continuing our CV journey, we will explore **Object Detection** on microcontrollers.

Object Detection versus Image Classification

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



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

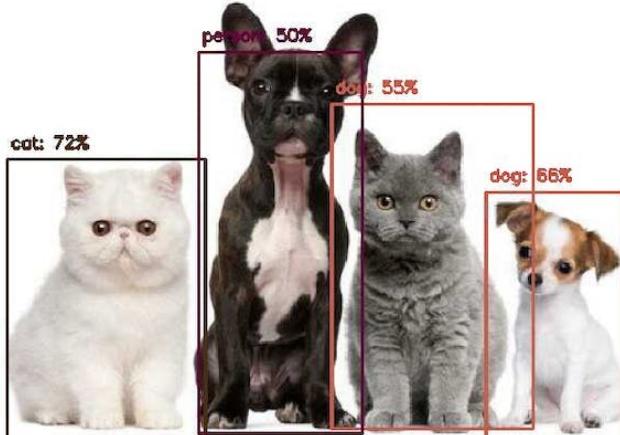


An image classification model identifies the above image utterly wrong as an “ashcan,” possibly due to the color tonalities.

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

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

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



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

An Innovative Solution for Object Detection: FOMO

Edge Impulse launched in 2022, **FOMO** (Faster Objects, More Objects), a novel solution to perform object detection on embedded devices, such

as the Nicla Vision and Portenta (Cortex M7), on Cortex M4F CPUs (Arduino Nano33 and OpenMV M4 series) as well the Espressif ESP32 devices (ESP-CAM, ESP-EYE and XIAO ESP32S3 Sense).

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

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

The Object Detection Project Goal

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



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

- Background (No objects)
- Fruit
- Bug

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



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

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

Data Collection

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

Collecting Dataset with the XIAO ESP32S3

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

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

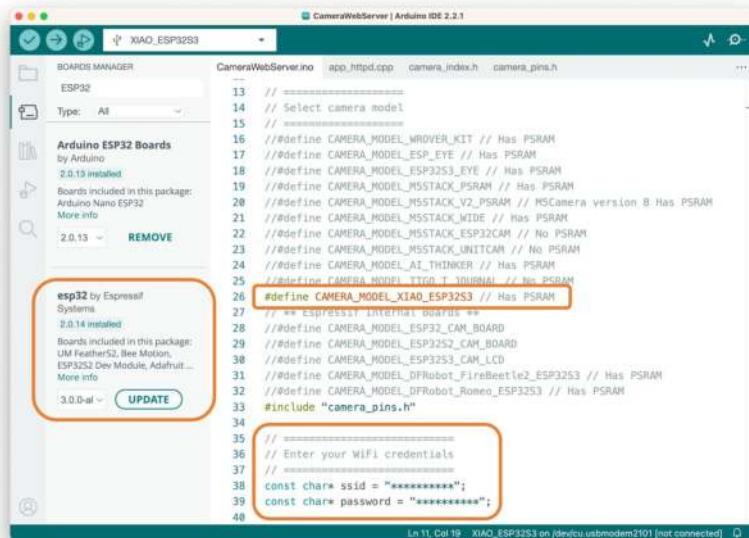
Attention

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

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

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

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

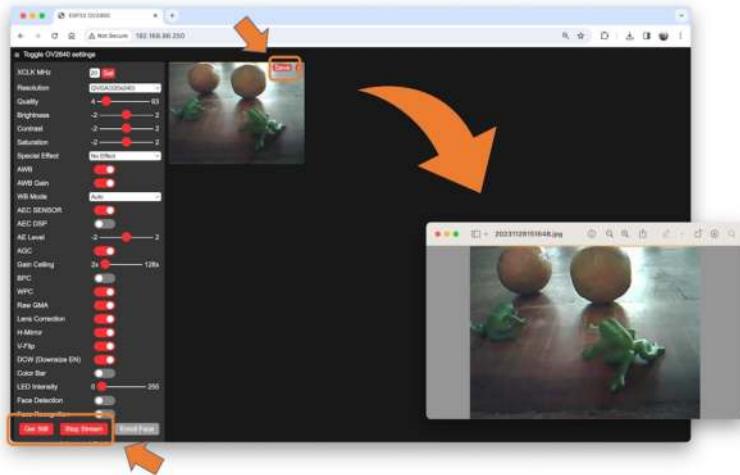


If the code is executed correctly, you should see the address on the Serial Monitor:



Copy the address on your browser and wait for the page to be uploaded. Select the camera resolution (for example, QVGA) and select [START]

STREAM]. Wait for a few seconds/minutes, depending on your connection. You can save an image on your computer download area using the [Save] button.



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

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

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

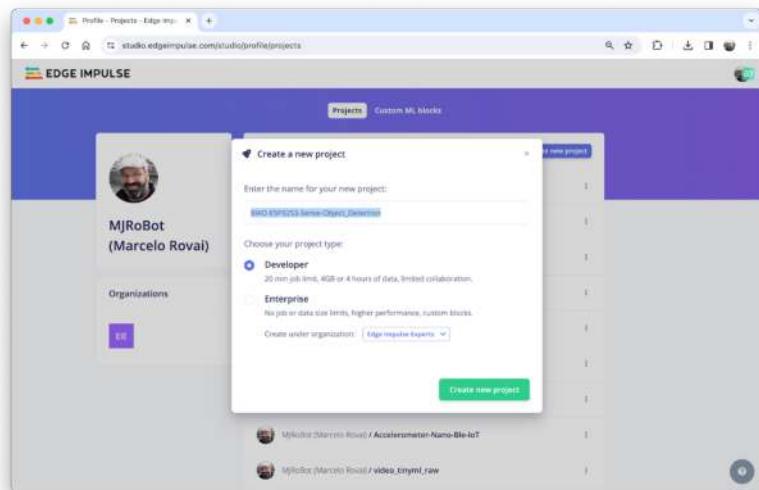
The stored images use a QVGA frame size of 320×240 and RGB565 (color pixel format).

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

Edge Impulse Studio

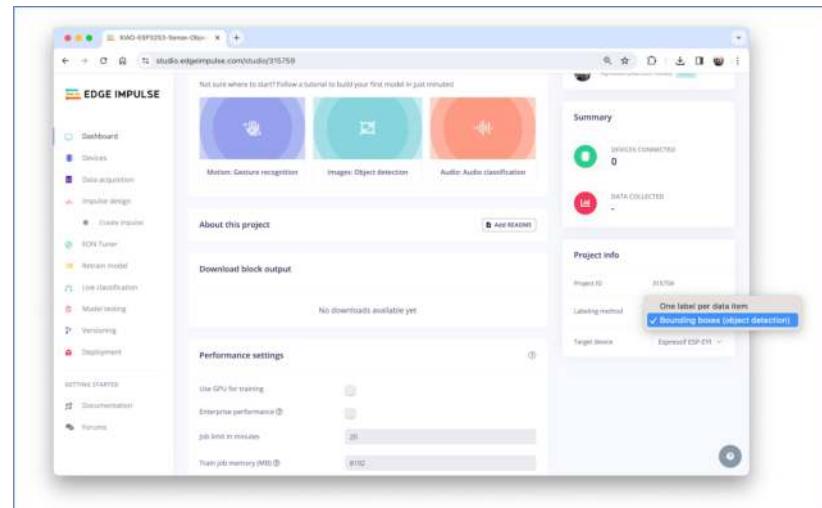
Setup the project

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



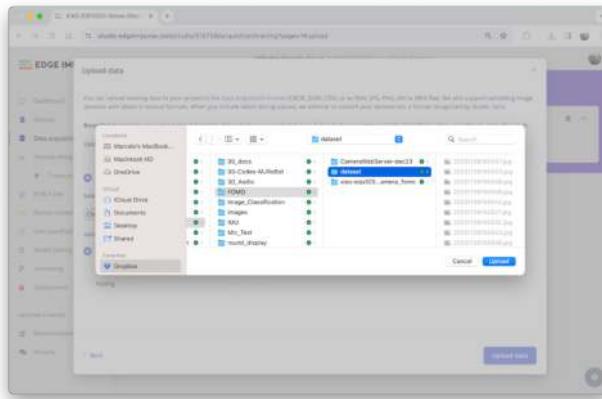
Here, you can clone the project developed for this hands-on:
XIAO-ESP32S3-Sense-Object_Detection

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

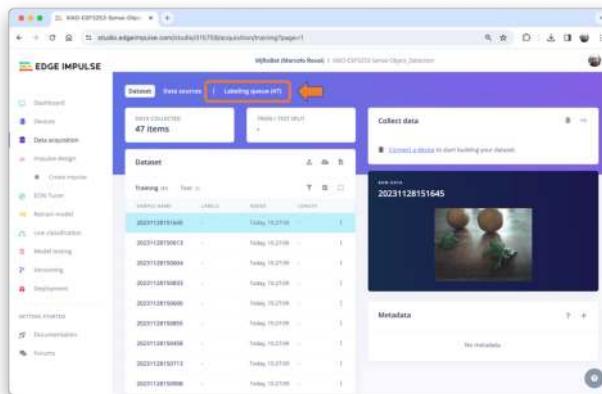


Uploading the unlabeled data

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



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



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

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

- Using yolov5
- Tracking objects between frames

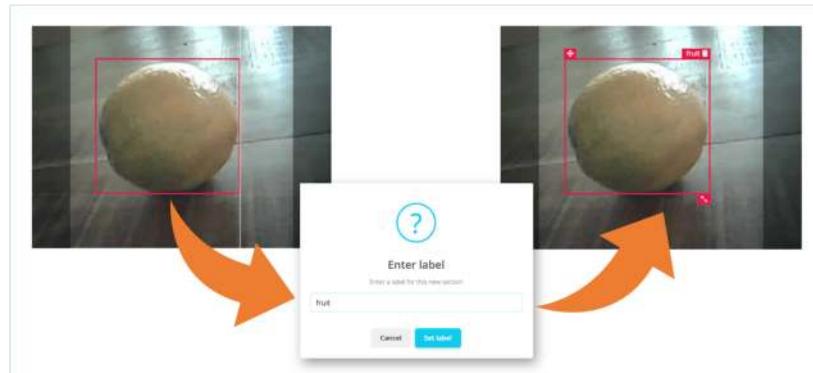
Edge Impulse launched an auto-labeling feature for Enterprise customers, easing labeling tasks in object detection projects.

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

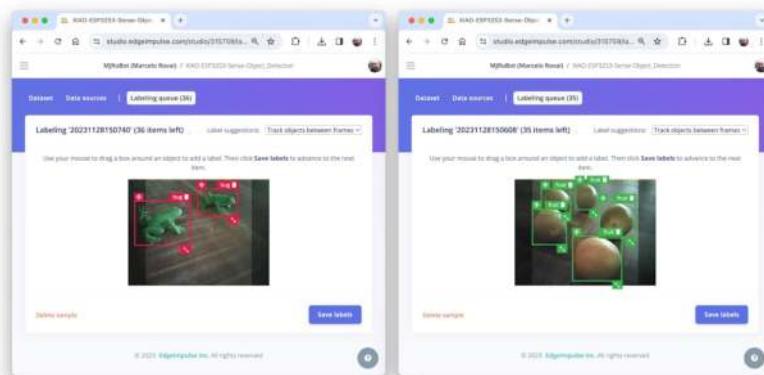
You can use the EI uploader to import your data if you already have a labeled dataset containing bounding boxes.

Labeling the Dataset

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



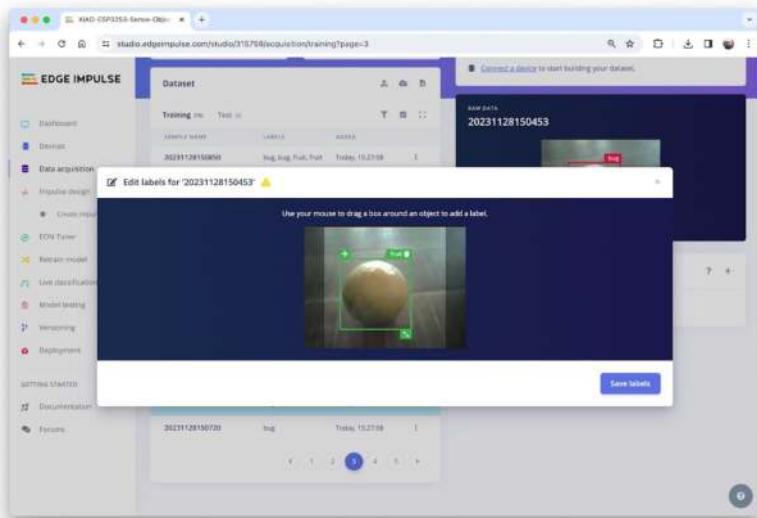
Continue with this process until the queue is empty. At the end, all images should have the objects labeled as those samples below:



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

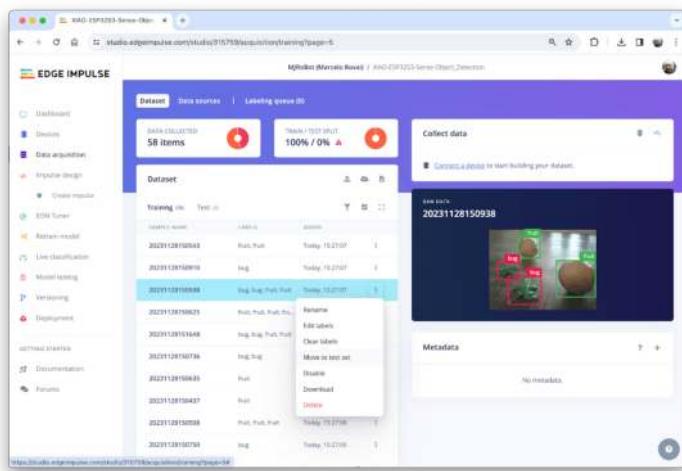
Training set	Test set	Last modified
Labels & Assets	Labels & Assets	Today, 15:27:08
20231128150500	Bug, Bug, Fruit, Fruit	Today, 15:27:08
20231128150716	Bug	Today, 15:27:08
20231128150514	Fruit, Fruit, Fruit	Today, 15:27:08
20231128150732	Bug, Bug	Today, 15:27:08
20231128150733	Bug	Today, 15:27:08
20231128150532	Fruit, Fruit, Fruit, Fruit	Today, 15:27:08
20231128150503	Bug, Bug	Today, 15:27:08
20231128150730	Bug, Bug	Today, 15:27:08
20231128150718	Bug	Today, 15:27:08
20231128150527	Bug, Fruit, Fruit, Fruit	Today, 15:27:08
20231128150453	Bug	Today, 15:27:08
20231128150720	Bug	Today, 15:27:08

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



Balancing the dataset and split Train/Test

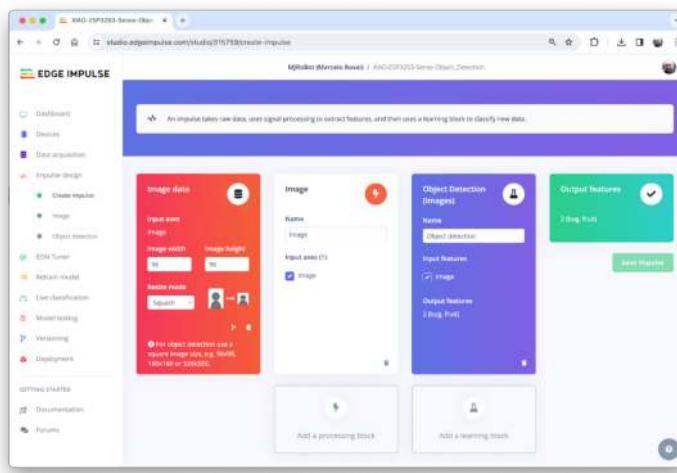
After labeling all data, it was realized that the class fruit had many more samples than the bug. So, 11 new and additional bug images were collected (ending with 58 images). After labeling them, it is time to select some images and move them to the test dataset. You can do it using the three-dot menu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



The Impulse Design

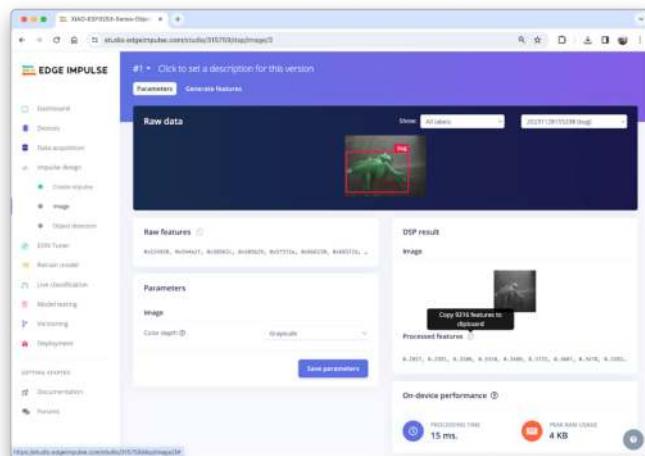
In this phase, you should define how to:

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

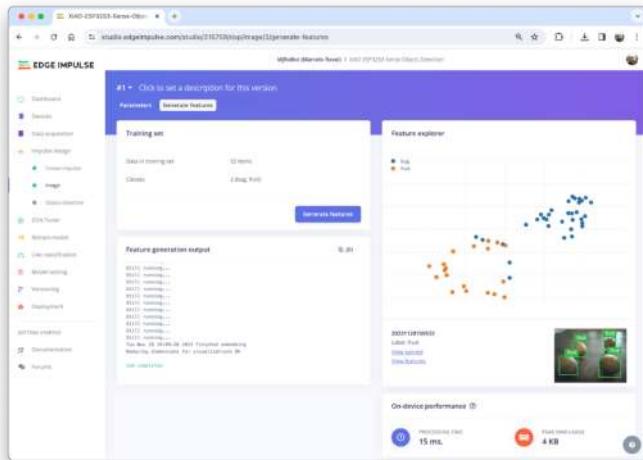


Preprocessing all dataset

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.

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

Model Design, Training, and Test

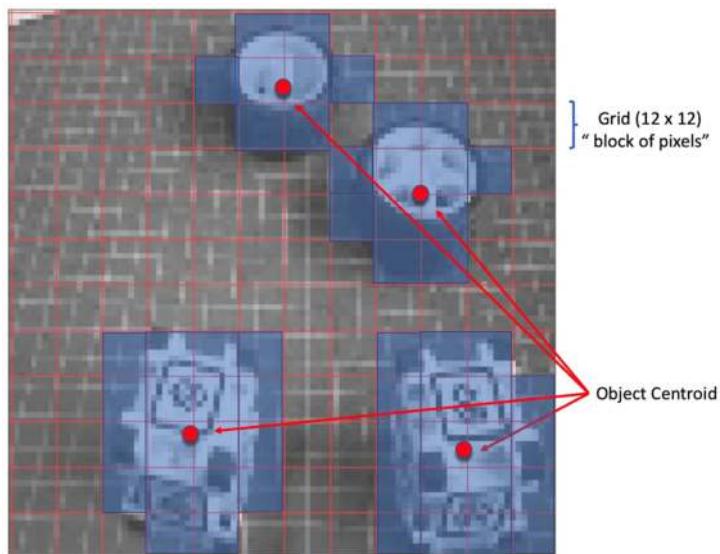
We will use FOMO, an object detection model based on MobileNetV2 (alpha 0.35) designed to coarsely segment an image into a grid of **background vs objects of interest** (here, *boxes* and *wheels*).

FOMO is an innovative machine learning model for object detection, which can use up to 30 times less energy and memory than traditional models like Mobilenet SSD and YOLOv5. FOMO can operate on microcontrollers with less than 200 KB of RAM. The main reason this is possible is that while other models calculate the object's size by drawing a square around it (bounding box), FOMO ignores the size of the image, providing only the information about where the object is located in the image through its centroid coordinates.

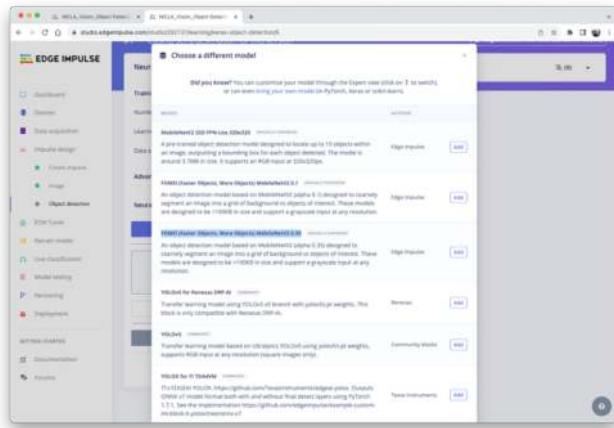
How FOMO works?

FOMO takes the image in grayscale and divides it into blocks of pixels using a factor of 8. For the input of 96×96 , 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.



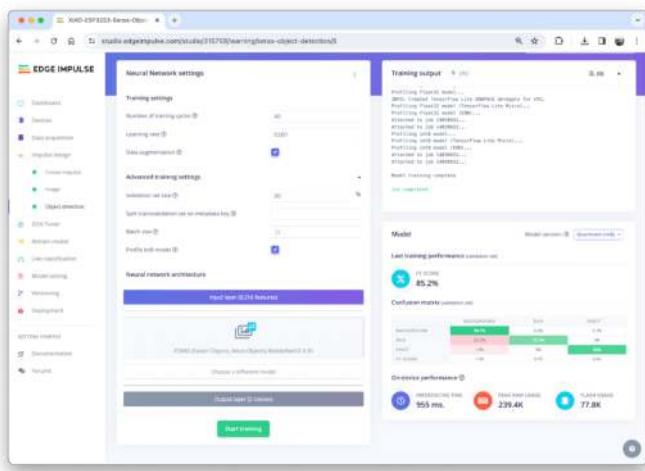
Regarding the training hyper-parameters, the model will be trained with:

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

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

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

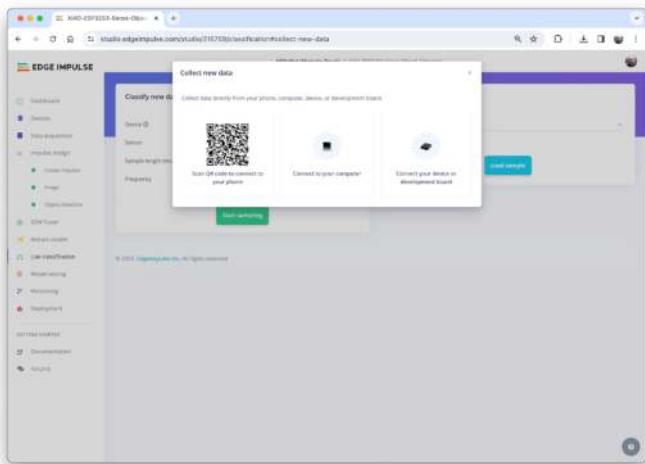
Note that FOMO automatically added a 3rd label background to the two previously defined (*box* and *wheel*).



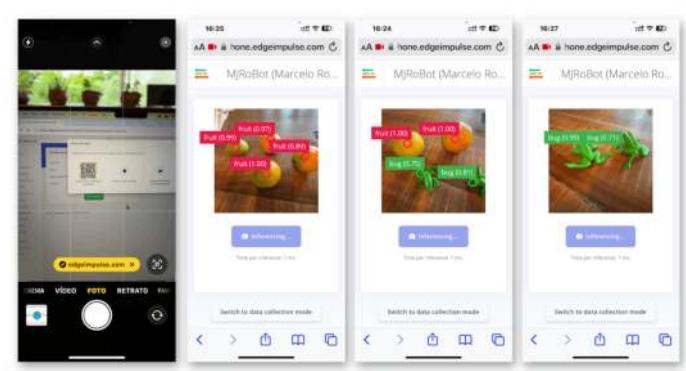
In object detection tasks, accuracy is generally not the primary evaluation metric. Object detection involves classifying objects and providing bounding boxes around them, making it a more complex problem than simple classification. The issue is that we do not have the bounding box, only the centroids.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

Test model with “Live Classification”

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



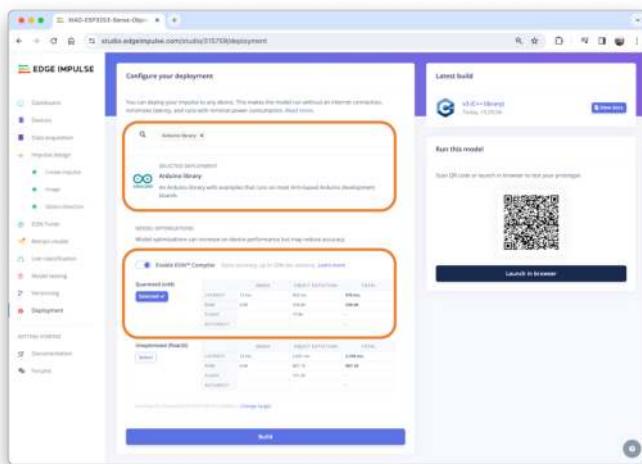
Once connected, you can use the smartphone to capture actual images to be tested by the trained model on Edge Impulse Studio.



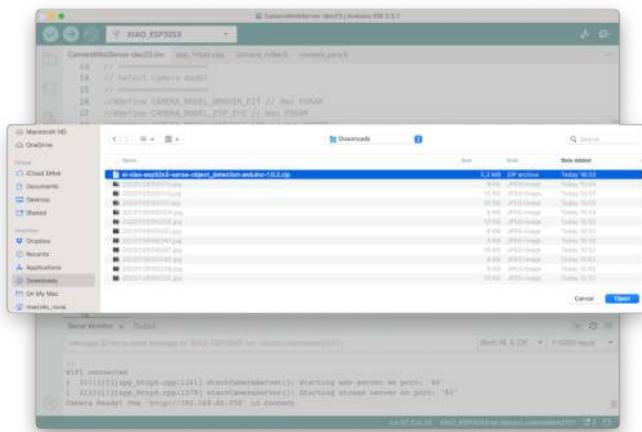
One thing to be noted is that the model can produce false positives and negatives. This can be minimized by defining a proper Confidence Threshold (use the Three dots menu for the setup). Try with 0.8 or more.

Deploying the Model (Arduino IDE)

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



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



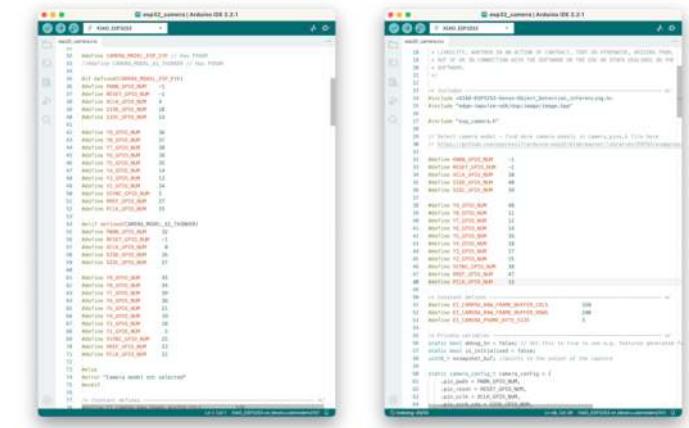
Under the Examples tab on Arduino IDE, you should find a sketch code (`esp32 > esp32_camera`) under your project name.



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

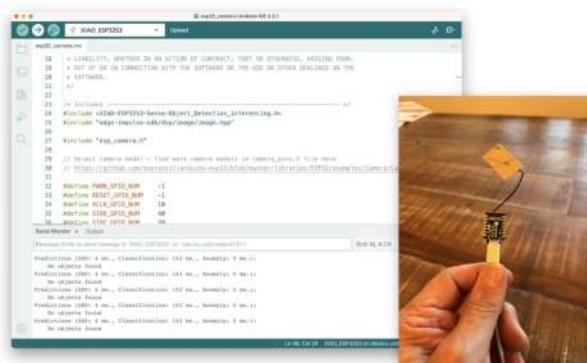
```
#define PWDN_GPIO_NUM      -1
#define RESET_GPIO_NUM     -1
#define XCLK_GPIO_NUM       10
#define SIOD_GPIO_NUM       40
#define SIOC_GPIO_NUM       39
#define Y9_GPIO_NUM          48
#define Y8_GPIO_NUM          11
#define Y7_GPIO_NUM          12
#define Y6_GPIO_NUM          14
#define Y5_GPIO_NUM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

Here you can see the resulting code:

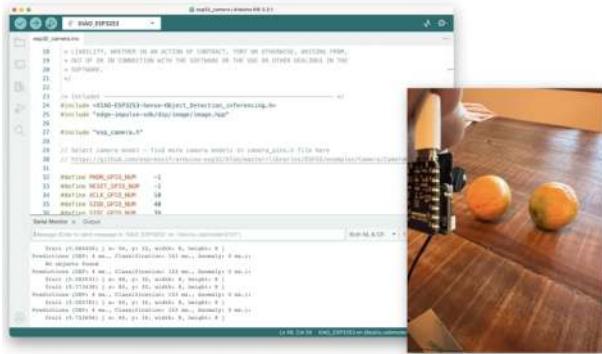


Upload the code to your XIAO ESP32S3 Sense, and you should be OK to start detecting fruits and bugs. You can check the result on Serial Monitor.

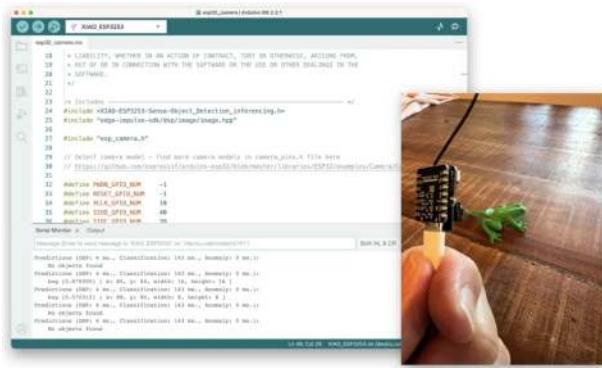
Background



Fruits



Bugs



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

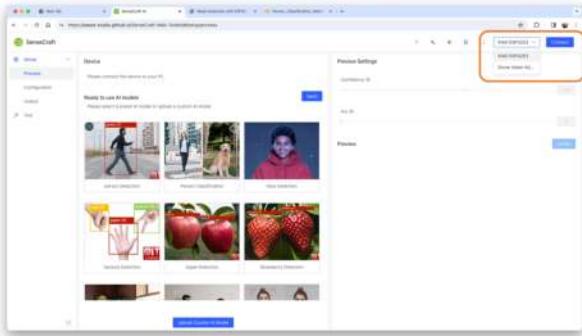
Deploying the Model (SenseCraft-Web-Toolkit)

As discussed in the Image Classification chapter, verifying inference with Image models on Arduino IDE is very challenging because we can

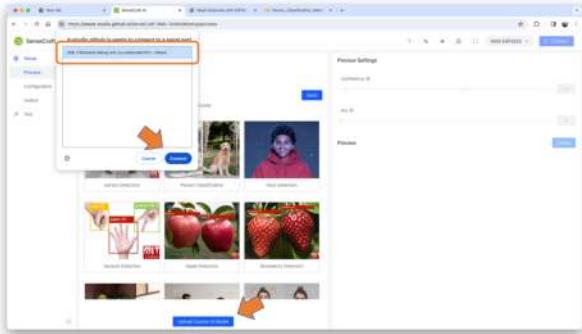
not see what the camera focuses on. Again, let's use the **SenseCraft-Web Toolkit**.

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

1. Open the SenseCraft-Web-Toolkit website.
2. Connect the XIAO to your computer:
 - Having the XIAO connected, select it as below:



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

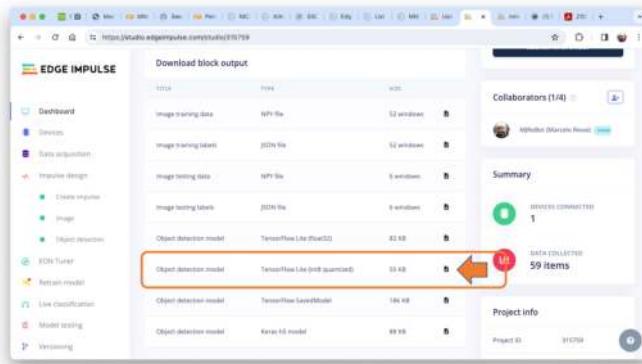


You can try several Computer Vision models previously uploaded by Seeed Studio. Try them and have fun!

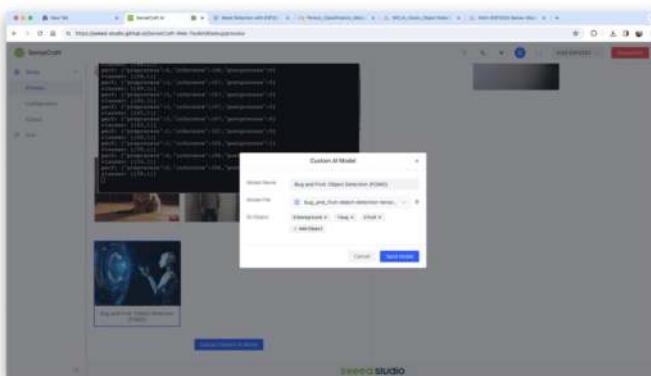
In our case, we will use the blue button at the bottom of the page: [Upload Custom AI Model].

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

3. Go to your project at Edge Impulse Studio, or clone this one:
 - XIAO-ESP32S3-CAM-Fruits-vs-Veggies-v1-ESP-NN
4. On Dashboard, download the model (“block output”): Object Detection model - TensorFlow Lite (int8 quantized)

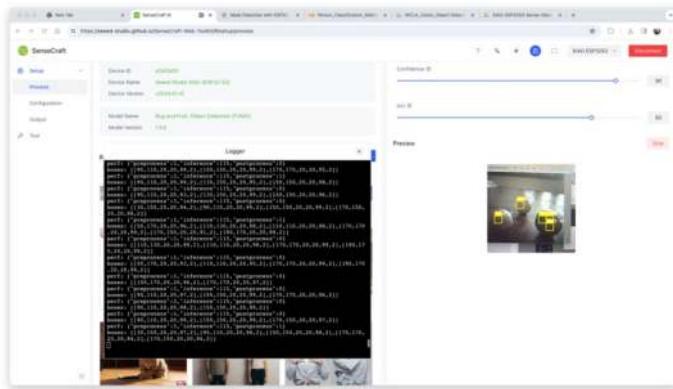


5. On SenseCraft-Web-Toolkit, use the blue button at the bottom of the page: [Upload Custom AI Model]. A window will pop up. Enter the Model file that you downloaded to your computer from Edge Impulse Studio, choose a Model Name, and enter with labels (ID: Object):



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

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



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

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

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

On Device Log, you will get information as:

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

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

Here are other screenshots:



Summary

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.

Resources

- Edge Impulse Project

Keyword Spotting (KWS)

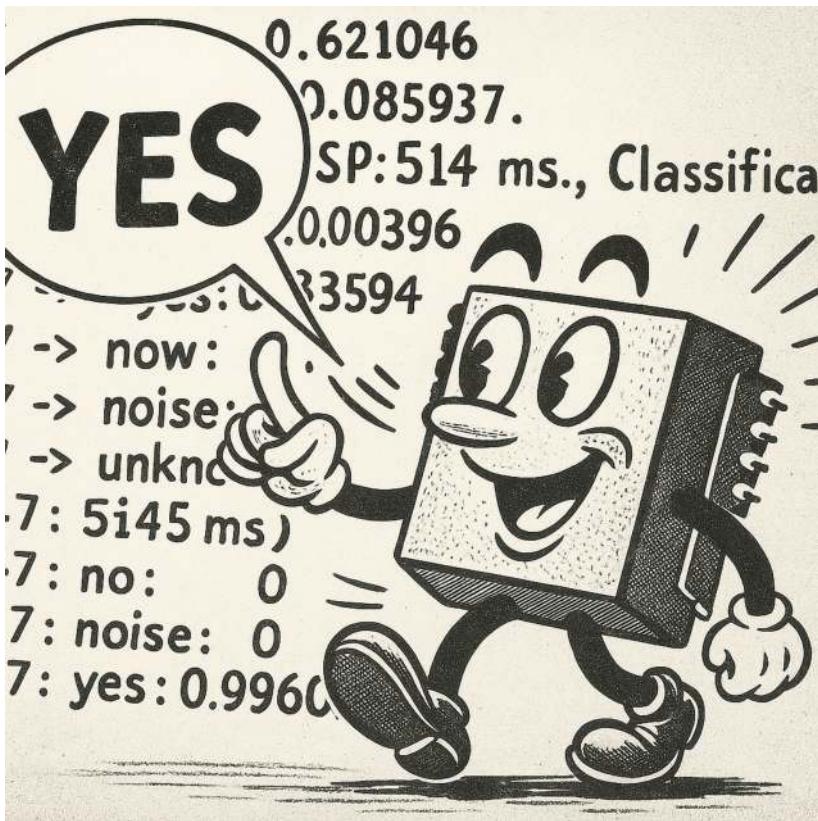


Figure 1.16: DALL-E prompt - 1950s style cartoon illustration based on a real image by Marcelo Rovai

Overview

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

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

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

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

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

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

Learning Objectives

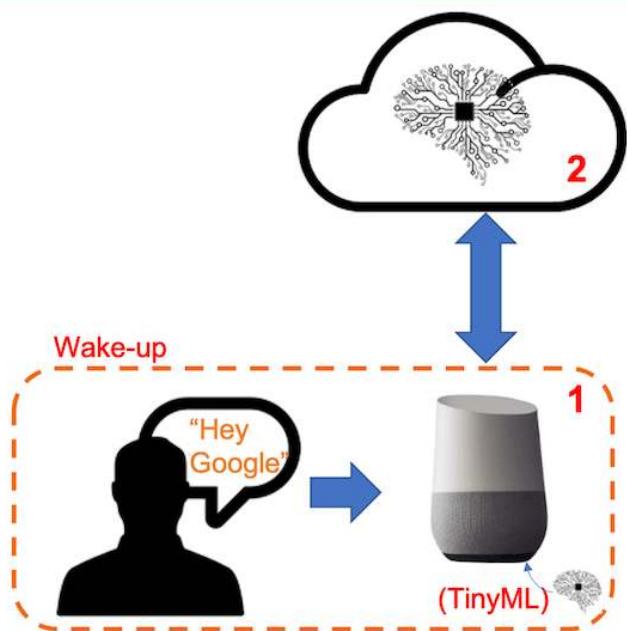
- **Understand Voice Assistant Architecture** including cascaded detection systems and the role of edge-based keyword spotting as the first stage of voice processing pipelines
- **Master Audio Data Collection Techniques** using both offline methods (XIAO ESP32S3 microphone with SD card

- storage) and online methods (smartphone integration with Edge Impulse Studio)
- **Implement Digital Signal Processing for Audio** including I2S protocol fundamentals, audio sampling at 16kHz/16-bit, and conversion between time-domain audio signals and frequency-domain features using MFCC
 - **Train Convolutional Neural Networks for Audio Classification** using transfer learning techniques, data augmentation strategies, and model optimization for four-class classification (YES, NO, NOISE, UNKNOWN)
 - **Deploy Optimized Models on Microcontrollers** through quantization (INT8), memory management with PSRAM, and real-time inference optimization for embedded systems
 - **Develop Complete Post-Processing Pipelines** including confidence thresholding, GPIO control for external devices, OLED display integration, and creating standalone AI sensor systems
 - **Compare Development Workflows** between no-code platforms (Edge Impulse Studio) and traditional embedded programming (Arduino IDE) for TinyML applications

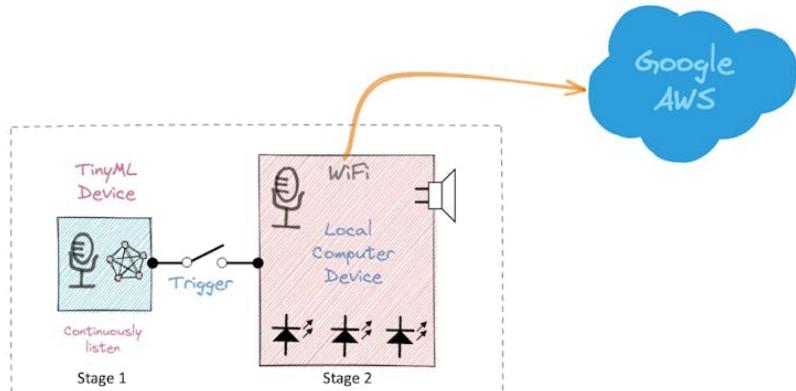
The KWS Project

How does a voice assistant work?

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



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



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

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

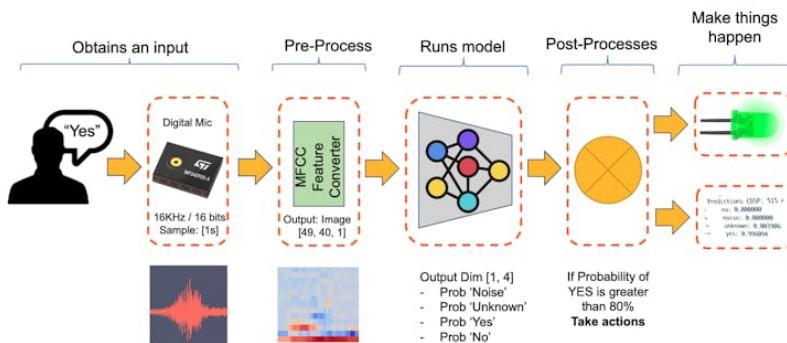
The video below shows an example where I emulate a Google Assistant on a Raspberry Pi (Stage 2), having an Arduino Nano 33 BLE as the tinyML device (Stage 1).

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

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

The Inference Pipeline

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



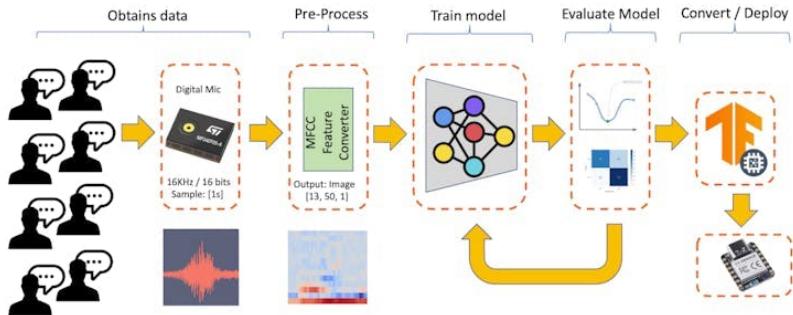
Our KWS application will recognize four classes of sound:

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

Optionally for real-world projects, it is always advised to include different words than keywords, such as "Noise" (or Background) and "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 Machine Learning Workflow is the **dataset**. Once we have decided on specific keywords (YES and NO), we can take advantage of the dataset developed by Pete Warden, “Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition.” This dataset has 35 keywords (with +1,000 samples each), such as yes, no, stop, and go. In other words, we can get 1,500 samples of *yes* and *no*.

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

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

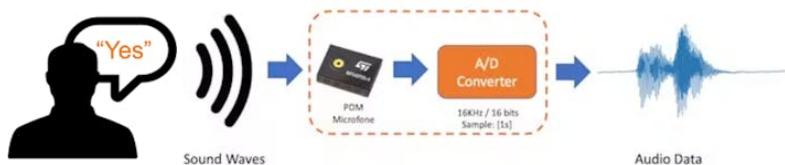
Although we have a lot of data from Pete’s dataset, collecting some words spoken by us is advised. When working with accelerometers, creating a dataset with data captured by the same type of sensor was essential. In the case of *sound*, the classification differs because it involves, in reality, *audio*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

The key difference between sound and audio is their form of energy. Sound is mechanical wave energy (longitudinal sound waves) that propagate through a medium causing

vibrations in pressure within the medium. Audio is made of electrical energy (analog or digital signals) that represent sound electrically.

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

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 proper XIAO ESP32S3 Sense, a computer, or even your mobile phone.



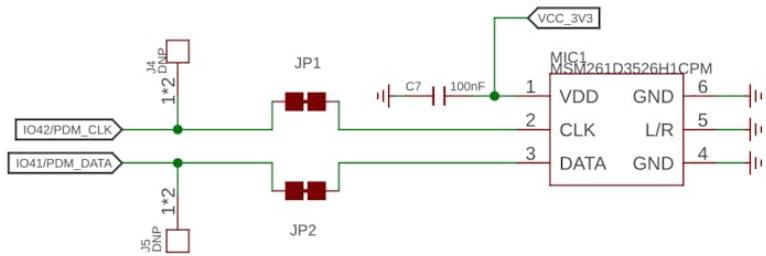
Capturing online Audio Data with Edge Impulse and a smartphone

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

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

Capturing (offline) Audio Data with the XIAO ESP32S3 Sense

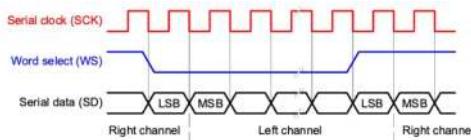
The built-in microphone is the MSM261D3526H1CPM, a PDM digital output MEMS microphone with Multi-modes. Internally, it is connected to the ESP32S3 via an I2S bus using pins IO41 (Clock) and IO41 (Data).



What is I2S?

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

The I2S protocol consists of at least three lines:



1. **Bit (or Serial) clock line (BCLK or CLK):** This line toggles to indicate the start of a new bit of data (pin IO42).
2. **Word select line (WS):** This line toggles to indicate the start of a new word (left channel or right channel). The Word select clock (WS) frequency defines the sample rate. In our case, L/R on the microphone is set to ground, meaning that we will use only the left channel (mono).
3. **Data line (SD):** This line carries the audio data (pin IO41)

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

Let's start understanding how to capture raw data using the microphone. Go to the GitHub project and download the sketch: XIAOEsp2s3_Mic-Test:

Attention

- The Xiao ESP32S3 **MUST** have the PSRAM enabled.
You can check it on the Arduino IDE upper menu:
Tools-> PSRAM:OPI PSRAM
- The Arduino Library (`esp32` by Espressif Systems) should be **version 2.017**. Please do not update it.

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

#include <I2S.h>

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

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

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

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

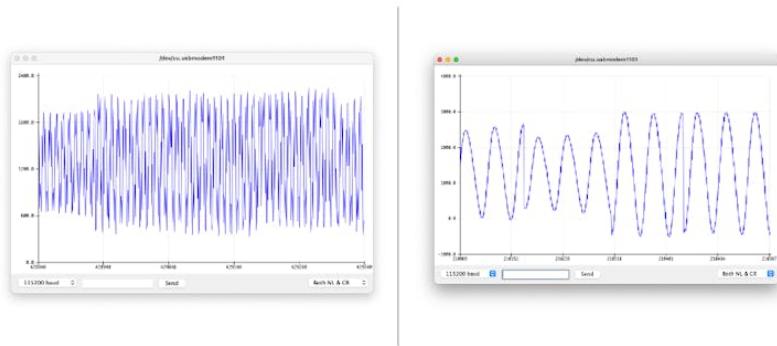
This code is a simple microphone test for the XIAO ESP32S3 using the I2S (Inter-IC Sound) interface. It sets up the I2S interface to capture audio data at a sample rate of 16 kHz with 16 bits per sample and then continuously reads samples from the microphone and prints them to the serial monitor.

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

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

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

Below is a test “whispering” in two different tones.



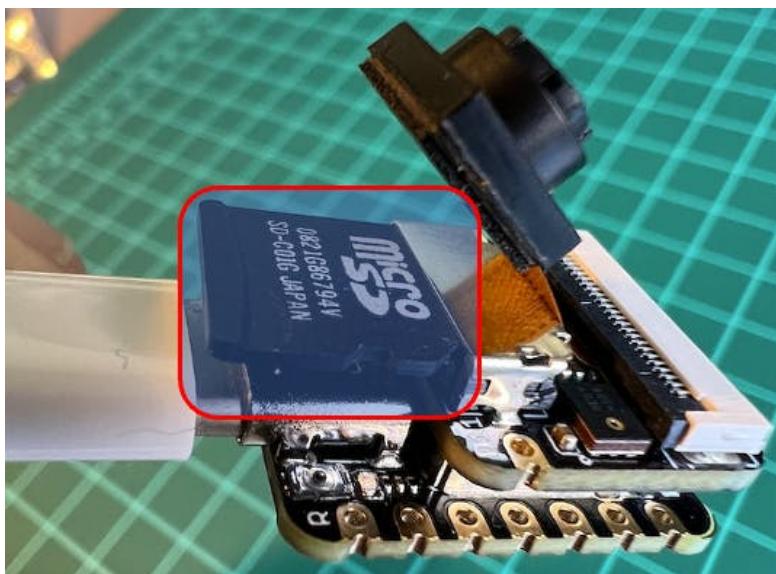
Save Recorded Sound Samples

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

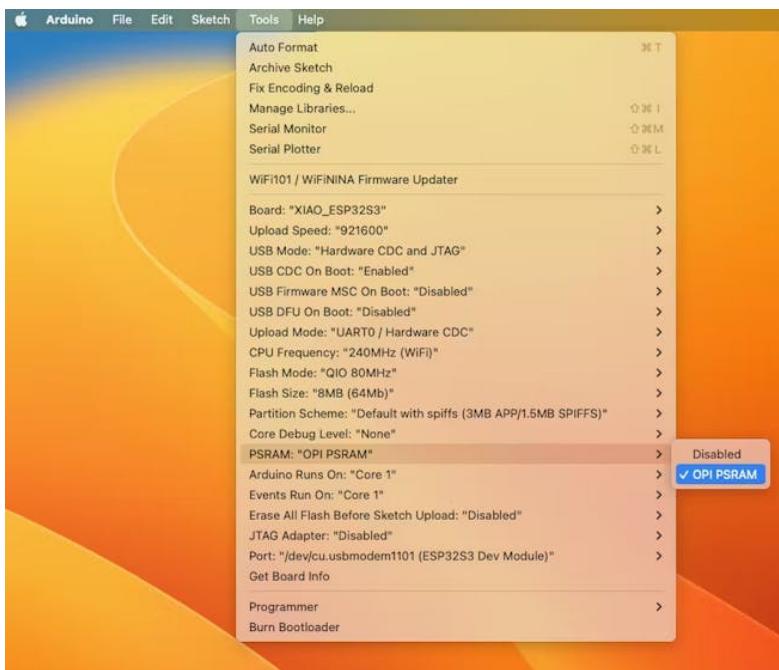
ESP32-S3 has only a few hundred kilobytes of internal RAM on the MCU chip. It can be insufficient for some purposes so that ESP32-S3 can use up to 16 MB of external PSRAM (Pseudo-static RAM) connected in parallel with the SPI flash chip. The external memory is incorporated in the memory

map and, with certain restrictions, is usable in the same way as internal data RAM.

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



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



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

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

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

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

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

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

Here, various constants are defined for the program.

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

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

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

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

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

    if (!SD.begin(21)){
        Serial.println("Failed to mount SD Card!");
        while (1);
    }
    Serial.printf("Enter with the label name\n");
}
```

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

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

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

```
void record_wav(String fileName)
{
    ...
    File file = SD.open(fileName.c_str(), FILE_WRITE);
    ...
    rec_buffer = (uint8_t *)ps_malloc(record_size);
    ...
    esp_i2s::i2s_read(esp_i2s::I2S_NUM_0,
                      rec_buffer,
                      record_size,
                      &sample_size,
```

```
    portMAX_DELAY);  
...  
}
```

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

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

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

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

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

```
// Start recording  
esp_i2s::i2s_read(esp_i2s::I2S_NUM_0,  
                   rec_buffer,  
                   record_size,  
                   &sample_size,  
                   portMAX_DELAY);  
if (sample_size == 0) {  
    Serial.printf("Record Failed!\n");  
} else {  
    Serial.printf("Record %d bytes\n", sample_size);  
}
```

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

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

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

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

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

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

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

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

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

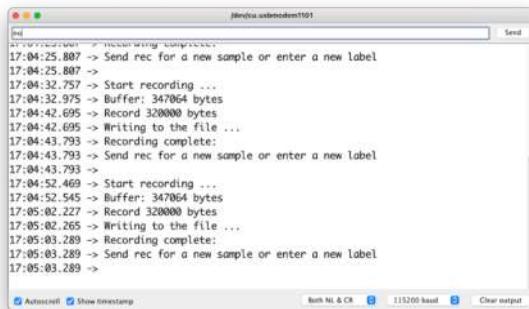
The Serial monitor will prompt you to receive the label to be recorded.



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



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



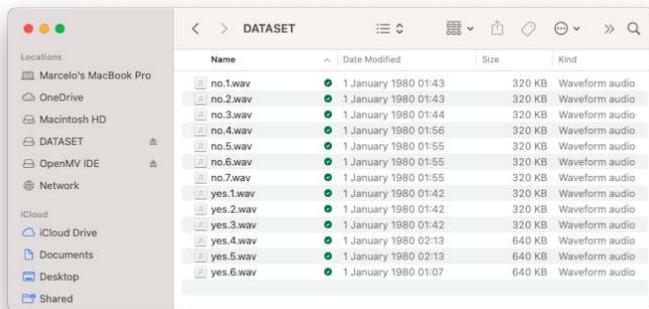
```

juno@juno-OptiPlex-5090:~/Desktop$ ./recording.sh
17:04:25.887 -> Send rec for a new sample or enter a new label
17:04:25.887 ->
17:04:32.757 -> Start recording ...
17:04:32.975 -> Buffer: 347064 bytes
17:04:42.695 -> Record 3200000 bytes
17:04:42.695 -> Writing to the file ...
17:04:43.793 -> Recording complete:
17:04:43.793 -> Send rec for a new sample or enter a new label
17:04:43.793 ->
17:04:52.469 -> Start recording ...
17:04:52.545 -> Buffer: 347064 bytes
17:05:02.227 -> Record 3200000 bytes
17:05:02.265 -> Writing to the file ...
17:05:03.289 -> Recording complete:
17:05:03.289 -> Send rec for a new sample or enter a new label
17:05:03.289 ->

```

Autoscroll Show timestamp Both NL & CR 215260 baud Clear output

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



The files are ready to be uploaded to Edge Impulse Studio

Capturing (offline) Audio Data Apps

There are many ways to capture audio data; the simplest one is to use a mobile phone or a PC as a **connected device** on the Edge Impulse Studio.

The PC or smartphone should capture audio data with a sampling frequency of 16 kHz and a bit depth of 16 Bits.

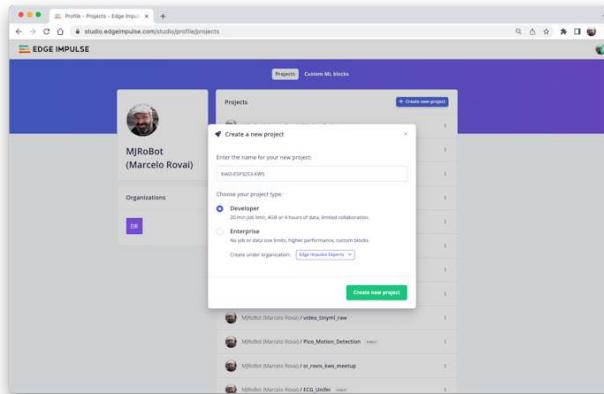
Another alternative is to use dedicated apps. A good app for that is *Voice Recorder Pro* (IOS). You should save your records as .wav files and send them to your computer.



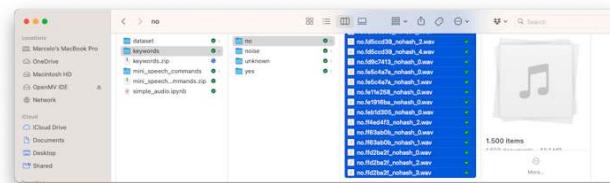
Training model with Edge Impulse Studio

Uploading the Data

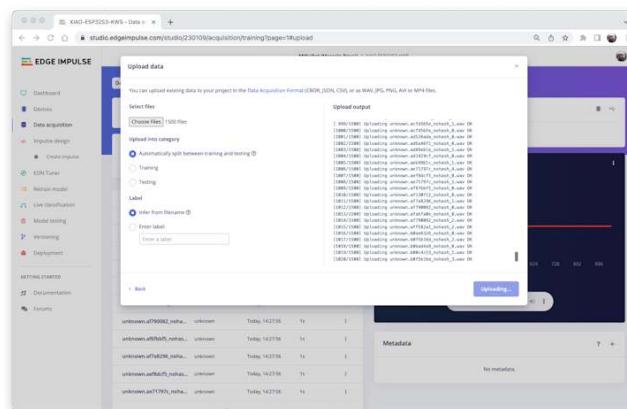
When the raw dataset is defined and collected (Pete's dataset + recorded keywords), we should initiate a new project at Edge Impulse Studio:



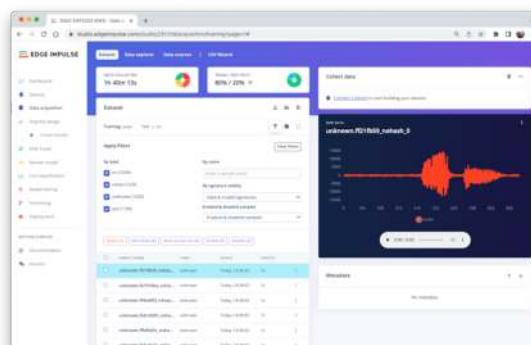
Once the project is created, select the Upload Existing Data tool in the Data Acquisition section. Choose the files to be uploaded:



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

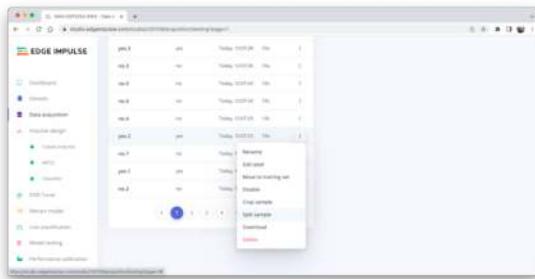


The samples will now appear in the Data acquisition section.

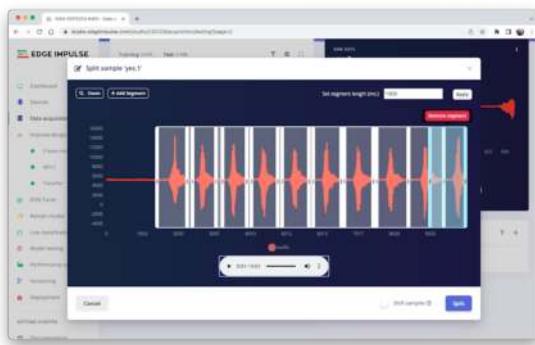


All data on Pete's dataset have a 1 s length, but the samples recorded in the previous section have 10 s and must be split into 1s samples to be compatible.

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



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



This procedure should be repeated for all samples.

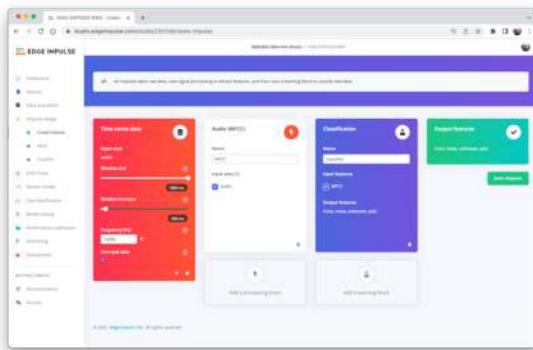
Note: For longer audio files (minutes), first, split into 10-second segments and after that, use the tool again to get the final 1-second splits.

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

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

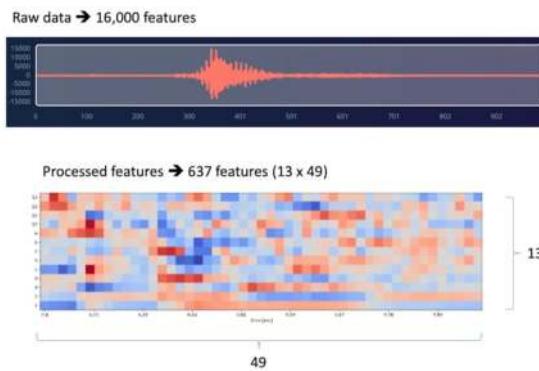
Creating Impulse (Pre-Process / Model definition)

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



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

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

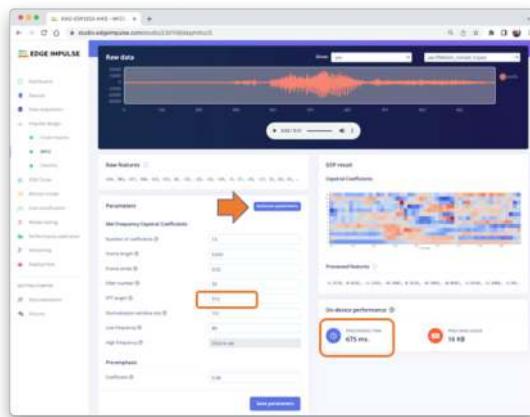


Next, we select KERAS for classification and build our model from scratch by doing Image Classification using Convolution Neural Network).

Pre-Processing (MFCC)

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

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

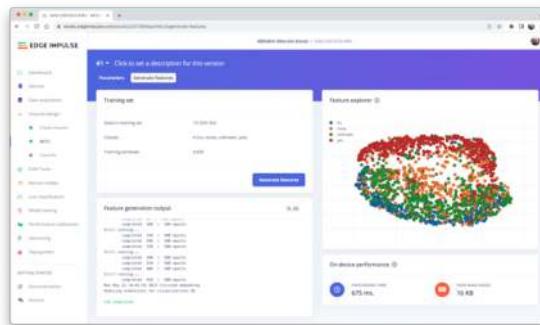


The result will not spend much memory to pre-process data (only 16KB). Still, the estimated processing time is high, 675 ms for an Espressif ESP-EYE (the closest reference available), with a 240 kHz clock (same as our

device), but with a smaller CPU (XTensa LX6, versus the LX7 on the ESP32S). The real inference time should be smaller.

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

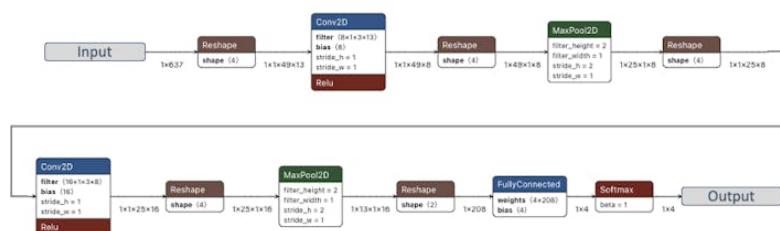
For now, let's keep the parameters defined by the Autotuning tool. Save parameters and generate the features.



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

Model Design and Training

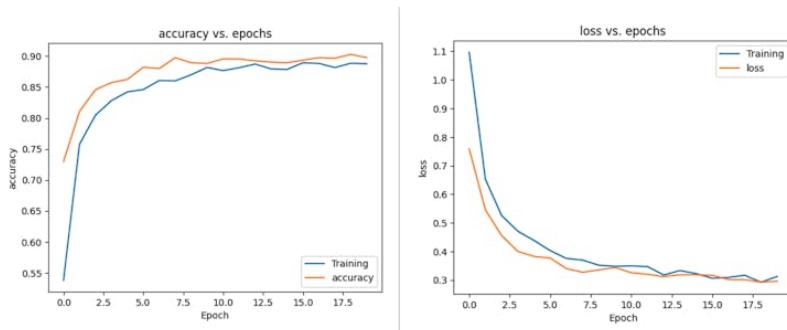
We will use a Convolution Neural Network (CNN) model. The basic architecture is defined with two blocks of Conv1D + MaxPooling (with 8 and 16 neurons, respectively) and a 0.25 Dropout. And on the last layer, after Flattening four neurons, one for each class:



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



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



This CoLab Notebook can explain how you can go further: KWS Classifier Project - Looking “Under the hood Training/xiao_esp32s3_keyword_spotting_project_nn_classifier.ipynb.”

Testing

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

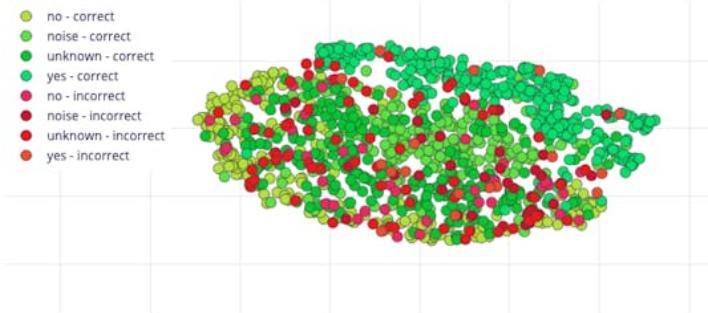
Model testing results

ACCURACY
86.73%

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

Feature explorer ②

- no - correct
- noise - correct
- unknown - correct
- yes - correct
- no - incorrect
- noise - incorrect
- unknown - incorrect
- yes - incorrect

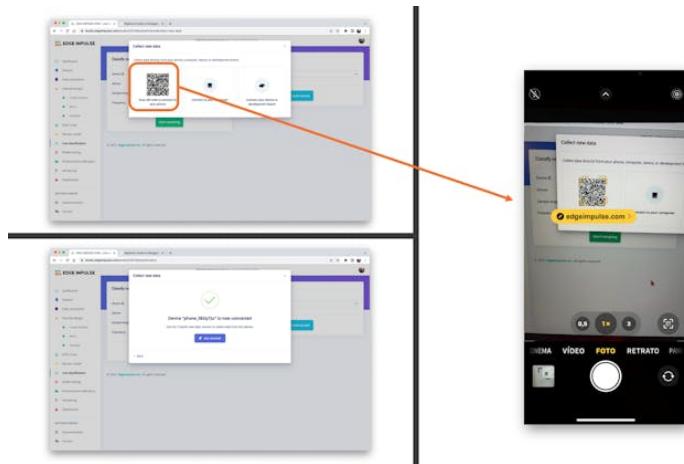


Inspecting the F1 score, we can see that for YES, we got 0.95, an excellent result once we used this keyword to “trigger” our postprocessing stage (turn on the built-in LED). Even for NO, we got 0.90. The worst result is for unknown, what is OK.

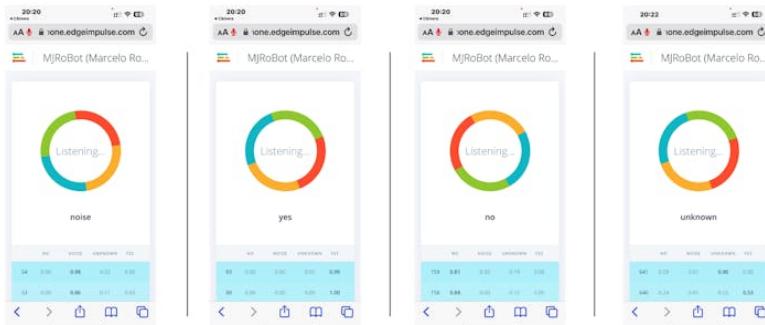
We can proceed with the project, but it is possible to perform Live Classification using a smartphone before deployment on our device. Go to the Live Classification section and click on Connect a Development board:



Point your phone to the barcode and select the link.



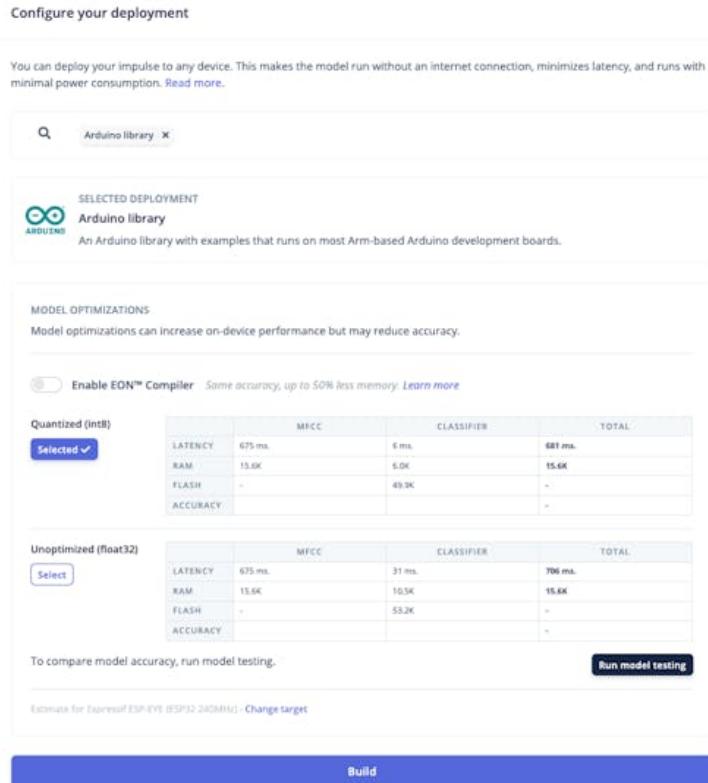
Your phone will be connected to the Studio. Select the option Classification on the app, and when it is running, start testing your keywords, confirming that the model is working with live and real data:



Deploy and Inference

The Studio will package all the needed libraries, preprocessing functions, and trained models, downloading them to your computer. Select

the Arduino Library option, then choose Quantized (Int8) from the bottom menu and press Build.



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

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



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

Start changing the libraries to handle the I2S bus:

```

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

```

By:

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

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

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

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

```

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

By:

```
/* read data at once from i2s */
esp_i2s::i2s_read(esp_i2s::I2S_NUM_0,
                  (void*)sampleBuffer,
```

```
i2s_bytes_to_read,
&bytes_read, 100);
```

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

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

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

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

By:

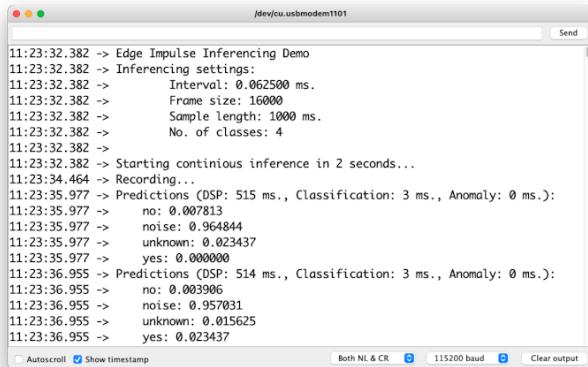
```
static void microphone_inference_end(void)
{
    free(sampleBuffer);
    ei_free(inference.buffer);
}
```

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

Attention

- The Xiao ESP32S3 **MUST** have the PSRAM enabled.
You can check it on the Arduino IDE upper menu:
Tools->PSRAM:OPI PSRAM

- The Arduino Library (esp32 by Espressif Systems) should be **version 2.017**. Please do not update it.



A screenshot of a terminal window titled "/dev/cu.usbmodem1101". The window displays the output of the Edge Impulse Inference Demo. The text shows configuration settings like interval, frame size, sample length, and number of classes. It then starts recording and performing inference. Predictions are shown for three categories: 'no', 'noise', and 'yes'. The 'noise' category has the highest probability (0.964844). The 'yes' category also has a high probability (0.023437). The terminal includes standard controls at the bottom: 'Autoscroll', 'Show timestamp', 'Both NL & CR', '115200 baud', and 'Clear output'.

```
11:23:32.382 -> Edge Impulse Inferencing Demo
11:23:32.382 -> Inferencing settings:
11:23:32.382 ->     Interval: 0.062500 ms.
11:23:32.382 ->     Frame size: 16000
11:23:32.382 ->     Sample length: 1000 ms.
11:23:32.382 ->     No. of classes: 4
11:23:32.382 ->
11:23:32.382 -> Starting continuous inference in 2 seconds...
11:23:34.464 -> Recording...
11:23:35.977 -> Predictions (DSP: 515 ms., Classification: 3 ms., Anomaly: 0 ms.):
11:23:35.977 ->     no: 0.007813
11:23:35.977 ->     noise: 0.964844
11:23:35.977 ->     unknown: 0.023437
11:23:35.977 ->     yes: 0.000000
11:23:36.955 -> Predictions (DSP: 514 ms., Classification: 3 ms., Anomaly: 0 ms.):
11:23:36.955 ->     no: 0.003906
11:23:36.955 ->     noise: 0.957031
11:23:36.955 ->     unknown: 0.015625
11:23:36.955 ->     yes: 0.023437
```

Postprocessing

In edge AI applications, the inference result is only as valuable as our ability to act upon it. While serial output provides detailed information for debugging and development, real-world deployments require immediate, human-readable feedback that doesn't depend on external monitors or connections.

Let's explore two post-processing approaches. Using the internal XIAO's LED and the OLED on the XIAOML Kit.

With LED

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

You should initialize the LED:

```
#define LED_BUILT_IN 21
...
void setup()
{
    ...
}
```

```
pinMode(LED_BUILT_IN, OUTPUT); // Set the pin as output
digitalWrite(LED_BUILT_IN, HIGH); //Turn off
...
}
```

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

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

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

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

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

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



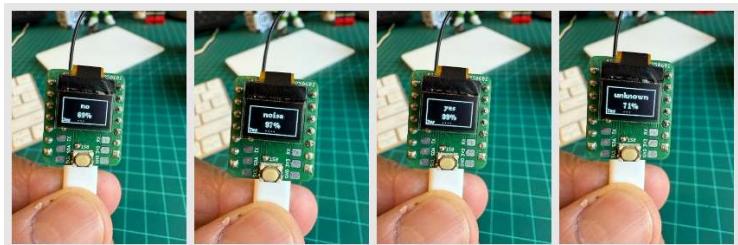
The idea is that the LED will be ON whenever the keyword YES is detected. In the same way, instead of turning on an LED, this could be a “trigger” for an external device, as we saw in the introduction.

With OLED Display

The XIAOMI Kit tiny 0.42" OLED display (72×40 pixels) serves as a crucial post-processing component that transforms raw ML inference results into immediate, human-readable feedback—displaying detected class names and confidence levels directly on the device, eliminating the need for external monitors and enabling truly standalone edge AI deployment in industrial, agricultural, or retail environments where instant visual confirmation of AI predictions is essential.

So, let’s modify the sketch to automatically adapt to the model trained on Edge Impulse by reading the class names and count directly from the model. Download the code from GitHub: [xiaoml-kit_kws_oled](#).

Running the code, we can see the result:



Summary

This lab demonstrated the complete development cycle of a keyword spotting system using the XIAOML Kit, showcasing how modern TinyML platforms make sophisticated audio AI accessible on resource-constrained devices. Through hands-on implementation, we've bridged the gap between theoretical machine learning concepts and practical embedded AI deployment.

Technical Achievements:

The project successfully implemented a complete audio processing pipeline from raw sound capture through real-time inference. Using the XIAO ESP32S3's integrated digital microphone, we captured audio data at professional quality (16kHz/16-bit) and processed it using Mel Frequency Cepstral Coefficients (MFCC) for feature extraction. The deployed CNN model achieved excellent accuracy in distinguishing between our target keywords ("YES", "NO") and background conditions ("NOISE", "UNKNOWN"), with inference times suitable for real-time applications.

Platform Integration:

Edge Impulse Studio proved invaluable as a comprehensive MLOps platform for embedded systems, handling everything from data collection and labeling through model training, optimization, and deployment. The seamless integration between cloud-based training and edge deployment exemplifies modern TinyML workflows, while the Arduino IDE provided the flexibility needed for custom post-processing implementations.

Real-World Applications:

The techniques learned extend far beyond simple keyword detection. Voice-activated control systems, industrial safety monitoring through sound classification, medical applications for respiratory analysis, and

environmental monitoring for wildlife or equipment sounds all leverage similar audio processing approaches. The cascaded detection architecture demonstrated here—using edge-based KWS to trigger more complex cloud processing—is fundamental to modern voice assistant systems.

Embedded AI Principles:

This project highlighted crucial TinyML considerations, including power management, memory optimization through PSRAM utilization, and the trade-offs between model complexity and inference speed. The successful deployment of a neural network performing real-time audio analysis on a microcontroller demonstrates how AI capabilities, once requiring powerful desktop computers, can now operate on battery-powered devices.

Development Methodology:

We explored multiple development pathways, from data collection strategies (offline SD card storage versus online streaming) to deployment options (Edge Impulse's automated library generation versus custom Arduino implementation). This flexibility is crucial for adapting to various project requirements and constraints.

Future Directions:

The foundation established here enables the exploration of more advanced audio AI applications. Multi-keyword recognition, speaker identification, emotion detection from voice, and environmental sound classification all build upon the same core techniques. The integration capabilities demonstrated with OLED displays and GPIO control illustrate how KWS can serve as the intelligent interface for broader IoT systems.

Consider that Sound Classification encompasses much more than just voice recognition. This project's techniques apply across numerous domains:

- **Security Applications:** Broken glass detection, intrusion monitoring, gunshot detection
- **Industrial IoT:** Machinery health monitoring, anomaly detection in manufacturing equipment
- **Healthcare:** Sleep disorder monitoring, respiratory condition assessment, elderly care systems
- **Environmental Monitoring:** Wildlife tracking, urban noise analysis, smart building acoustic management

- **Smart Home Integration:** Multi-room voice control, appliance status monitoring through sound signatures

Key Takeaways:

The XIAOML Kit proves that professional-grade AI development is achievable with accessible tools and modest budgets. The combination of capable hardware (ESP32S3 with PSRAM and integrated sensors), mature development platforms (Edge Impulse Studio), and comprehensive software libraries creates an environment where complex AI concepts become tangible, working systems.

This lab demonstrates that the future of AI isn't just in massive data centers, but in intelligent edge devices that can process, understand, and respond to their environment in real-time—opening possibilities for ubiquitous, privacy-preserving, and responsive artificial intelligence systems.

Resources

- XIAO ESP32S3 Codes
- XIAOML Kit Code
- Subset of Google Speech Commands Dataset
- KWS MFCC Analysis Colab Notebook
- KWS CNN training Colab Notebook
- XIAO ESP32S3 Post-processing Code
- Edge Impulse Project

Motion Classification and Anomaly Detection

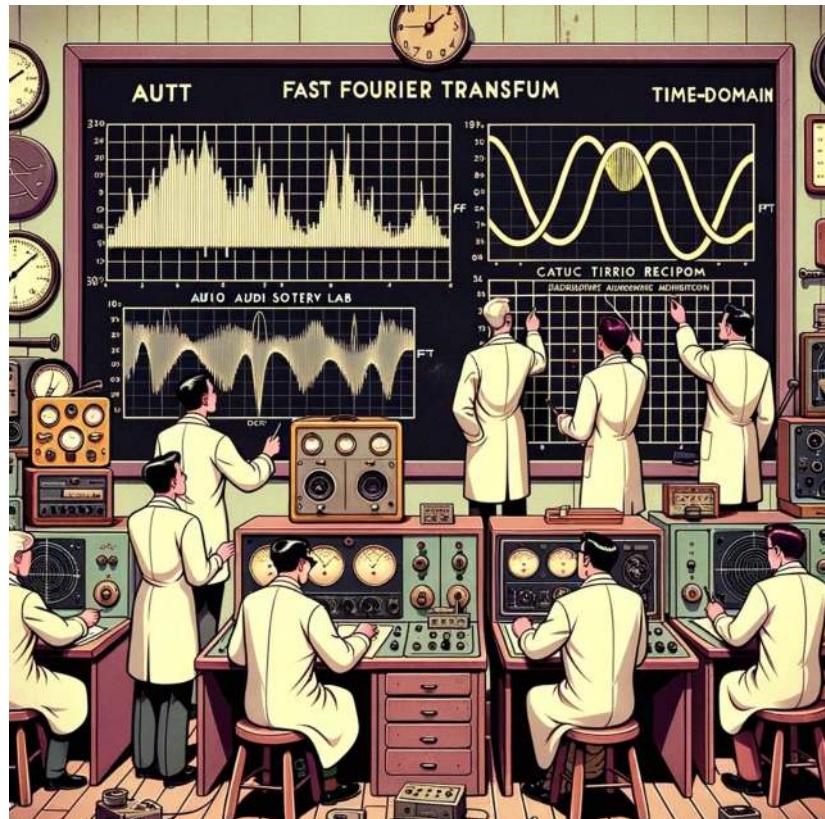


Figure 1.17: DALL-E prompt - 1950s style cartoon illustration set in a vintage audio lab. Scientists, dressed in classic attire with white lab coats, are intently analyzing audio data on large chalkboards. The boards display intricate FFT (Fast Fourier Transform) graphs and time-domain curves. Antique audio equipment is scattered around, but the data representations are clear and detailed, indicating their focus on audio analysis.

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 the safe and efficient transit of these containers is a monumental task that requires leveraging modern technology, and TinyML is undoubtedly one of the key solutions.

In this hands-on lab, we will work to solve real-world problems related to transportation. We will develop a Motion Classification and Anomaly Detection system using the XIAOML Kit, 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, including terrestrial and maritime transit, vertical movement via forklifts, and periods of stationary storage in warehouses.

💡 Learning Objectives

- Setting up the XIAOML Kit
- Data Collection and Preprocessing
- Building the Motion Classification Model
- Implementing Anomaly Detection
- Real-world Testing and Analysis

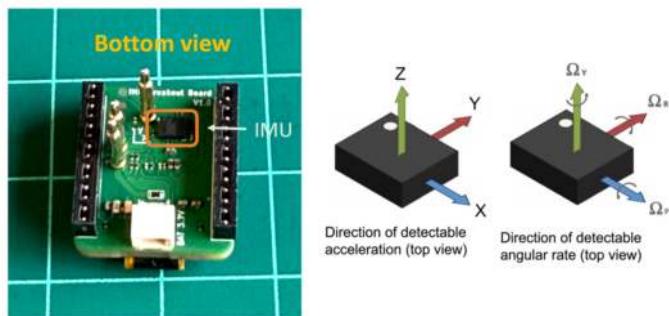
By the end of this lab, you'll have a working prototype that can classify different types of motion and detect anomalies during the transportation of containers. This knowledge can serve as a stepping stone to more advanced projects in the burgeoning field of TinyML, particularly those involving vibration.

Installing the IMU

The XIAOML Kit comes with a built-in LSM6DS3TR-C 6-axis IMU sensor on the expansion board, eliminating the need for external sensor connections. This integrated approach offers a clean and reliable platform for motion-based machine learning applications.

The LSM6DS3TR-C combines a 3-axis accelerometer and 3-axis gyroscope in a single package, connected via I2C to the XIAO ESP32S3 at address 0x6A that provides:

- **Accelerometer ranges:** $\pm 2/\pm 4/\pm 8/\pm 16$ g (we'll use ± 2 g by default)
- **Gyroscope ranges:** $\pm 125/\pm 250/\pm 500/\pm 1000/\pm 2000$ dps (we'll use ± 250 dps by default)
- **Resolution:** 16-bit ADC
- **Communication:** I2C interface at address 0x6A
- **Power:** Ultra-low power design



Coordinate System: The sensor operates within a right-handed coordinate system. When looking at the expansion board from the bottom (where you can see the IMU sensor with the point mark):

- **X-axis:** Points to the right
- **Y-axis:** Points forward (away from you)
- **Z-axis:** Points upward (out of the board)

Setting Up the Hardware

Since the XIAOML Kit comes pre-assembled with the expansion board, no additional hardware connections are required. The LSM6DS3TR-C IMU is already properly connected via I2C.

What's Already Connected:

- LSM6DS3TR-C IMU → I2C (SDA/SCL) → XIAO ESP32S3
- I2C Address: 0x6A
- Power: 3.3V from XIAO ESP32S3

Required Library: You should have the library installed during the Setup. If not, install the Seeed Arduino LSM6DS3 library following the steps:

1. Open Arduino IDE Library Manager
2. Search for “LSM6DS3”
3. Install “**Seeed Arduino LSM6DS3**” by Seeed Studio
4. **Important:** Do NOT install “Arduino_LSM6DS3 by Arduino” - that’s for different boards!

Testing the IMU Sensor

Let's start with a simple test to verify the IMU is working correctly. Upload this code to test the sensor:

```
#include <LSM6DS3.h>
#include <Wire.h>

// Create IMU object using I2C interface
LSM6DS3 myIMU(I2C_MODE, 0x6A);

float accelX, accelY, accelZ;
float gyroX, gyroY, gyroZ;

void setup() {
    Serial.begin(115200);
    while (!Serial) delay(10);

    Serial.println("XIAOMI Kit IMU Test");
    Serial.println("LSM6DS3TR-C 6-Axis IMU");
    Serial.println("====");

    // Initialize the IMU
    if (myIMU.begin() != 0) {
        Serial.println("ERROR: IMU initialization failed!");
        while(1) delay(1000);
    } else {
        Serial.println(" IMU initialized successfully");
        Serial.println("Data Format: AccelX,AccelY,AccelZ,");
        Serial.println("GyroX,GyroY,GyroZ");
        Serial.println("Units: g-force, degrees/second");
        Serial.println();
    }
}

void loop() {
    // Read accelerometer data (in g-force)
    accelX = myIMU.readFloatAccelX();
    accelY = myIMU.readFloatAccelY();
    accelZ = myIMU.readFloatAccelZ();

    // Read gyroscope data (in degrees per second)
    gyroX = myIMU.readFloatGyroX();
    gyroY = myIMU.readFloatGyroY();
```

```

gyroZ = myIMU.readFloatGyroZ();

// Print readable format
Serial.print("Accel (g): X="); Serial.print(accelX, 3);
Serial.print(" Y="); Serial.print(accely, 3);
Serial.print(" Z="); Serial.print(accelz, 3);
Serial.print(" | Gyro (°/s): X="); Serial.print(gyroX, 2);
Serial.print(" Y="); Serial.print(gyroY, 2);
Serial.print(" Z="); Serial.println(gyroZ, 2);

delay(100); // 10 Hz update rate
}

```

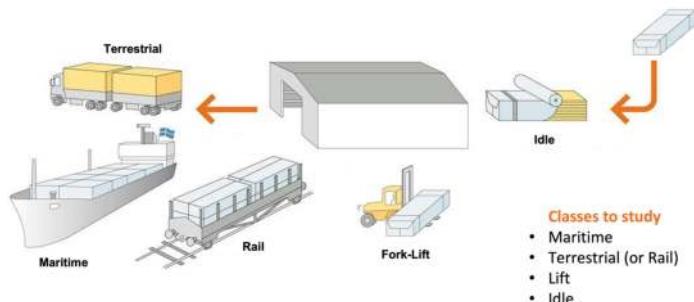
When the kit is resting flat on a table, you should see:

- Z-axis acceleration around +1.0g (gravity)
- X and Y acceleration near 0.0g
- All gyroscope values near 0°/s

Move the kit around to see the values change accordingly.

The TinyML Motion Classification Project

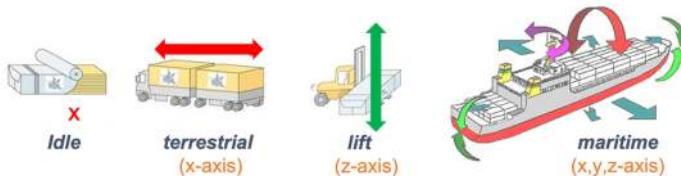
We will simulate container (or, more accurately, package) transportation through various scenarios to make this tutorial more relatable and practical.



Using the accelerometer of the XIAOML Kit, we'll capture motion data by manually simulating the conditions of:

- **Maritime** (pallets on boats) - Movement in all axes with wave-like patterns
- **Terrestrial** (pallets on trucks/trains) - Primarily horizontal movement
- **Lift** (pallets being moved by forklift) - Primarily vertical movement
- **Idle** (pallets in storage) - Minimal movement

From the above image, 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 have several options available. In a real-world scenario, we can have our device, for example, connected directly to one container, and the collected data stored in a file (for example, CSV) on an SD card. Data can also be sent remotely to a nearby repository, such as a mobile phone, using Wi-Fi or Bluetooth (as demonstrated 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.

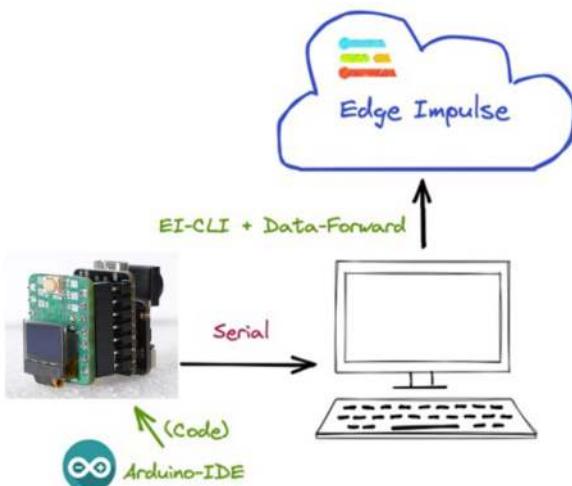
Preparing the Data Collection Code

In this lab, we will connect the Kit directly to the Edge Impulse Studio, which will also be used for data pre-processing, model training, testing, and deployment.

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

Follow the instructions here to install Node.js and Edge Impulse CLI on your computer.

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



We'll modify our test code to output data in a format suitable for Edge Impulse:

```
#include <LSM6DS3.h>
#include <Wire.h>

#define FREQUENCY_HZ      50
#define INTERVAL_MS        (1000 / (FREQUENCY_HZ + 1))

LSM6DS3 myIMU(I2C_MODE, 0x6A);
static unsigned long last_interval_ms = 0;

void setup() {
    Serial.begin(115200);
    while (!Serial) delay(10);
```

```
Serial.println("XIAOMI Kit - Motion Data Collection");
Serial.println("LSM6DS3TR-C IMU Sensor");

// Initialize IMU
if (myIMU.begin() != 0) {
    Serial.println("ERROR: IMU initialization failed!");
    while(1) delay(1000);
}

delay(2000);
Serial.println("Starting data collection in 3 seconds...");
delay(3000);
}

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

        // Read accelerometer data
        float ax = myIMU.readFloatAccelX();
        float ay = myIMU.readFloatAccelY();
        float az = myIMU.readFloatAccelZ();

        // Convert to m/s2 (multiply by 9.81)
        float ax_ms2 = ax * 9.81;
        float ay_ms2 = ay * 9.81;
        float az_ms2 = az * 9.81;

        // Output in Edge Impulse format
        Serial.print(ax_ms2);
        Serial.print("\t");
        Serial.print(ay_ms2);
        Serial.print("\t");
        Serial.println(az_ms2);
    }
}
```

Upload the code to the Arduino IDE. We should see the accelerometer values (converted to m/s²) at the Serial Monitor:

```
sketch_jul23a.ino
40
41
42
43     // Output in Edge Impulse format
44     Serial.print(ax_ms2);
45     Serial.print("\t");
46     Serial.print(ay_ms2);
47     Serial.print("\t");
48     Serial.println(az_ms2);
49 }
50

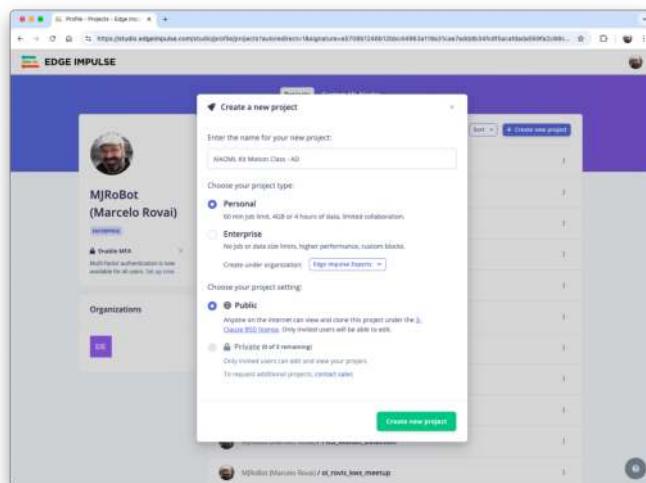
Output  Serial Monitor X
Message (Enter to send message to 'XIAO_ESP32S3' on '/dev/cu.usbmodem101')
Both NL & CR  115200 baud
-1.14  0.48  10.20
-0.66  0.58  9.99
-0.74  0.51  10.12
-0.93  0.45  10.29

Ln 6, Col 1  XIAO_ESP32S3 on /dev/cu.usbmodem101  2  2
```

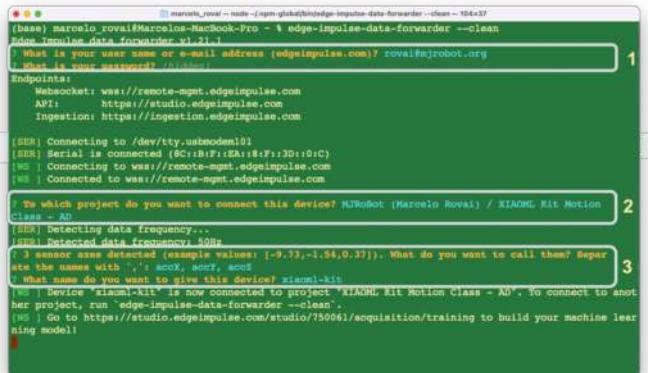
Keep the code running, but **turn off the Serial Monitor**. The data generated by the Kit will be sent to the Edge Impulse Studio via Serial Connection.

Connecting to Edge Impulse for Data Collection

Create an Edge Impulse Project - Go to Edge Impulse Studio and create a new project - Choose a descriptive name (keep under 63 characters for Arduino library compatibility)



- Set up CLI Data Forwarder** - Install Edge Impulse CLI on your computer
 - Confirm that the XIAOML Kit is connected to the computer, **the code is running and the Serial Monitor is OFF**, otherwise we can get an error.
 - On the Computer Terminal, run: `edge-impulse-data-forwarder --clean` - Enter your Edge Impulse credentials - Select your project and configure device settings



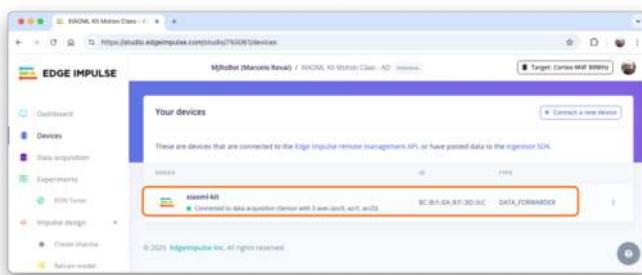
```

(base) marcelo_royai@Marcelos-MacBook-Pro ~ % edge-impulse-data-forwarder --clean
Edge Impulse data forwarder v1.21.1
? What is your user name or e-mail address (edgeimpulse.com)? rovai@irobot.org
? What is your password? [REDACTED]
Endpoints:
Websocket: ws://remote-agent.edgeimpulse.com
API: https://studio.edgeimpulse.com
Ingestion: https://ingestion.edgeimpulse.com

[ER] Connecting to /dev/tty.usbmodem101
[ER] Serial is connected (C:0:F:1E:8:F:1D0:0:C)
[WS] Connecting to ws://remote-agent.edgeimpulse.com
[WS] Connected to ws://remote-agent.edgeimpulse.com

? To which project do you want to connect this device? MJRobot (Marcelo Rovai) / XIAOML Kit Motion Class - AD
[ER] Detecting data frequency...
[ER] Detected data frequency: 50Hz
? 3 sensor axes detected (example values: (-9.73,-1.56,0.37)). What do you want to call them? Negate the names with !, !accX, !accY, !accZ
? What name do you want to give this device? xiaomlkit
? Device xiaomlkit is now connected to project 'XIAOML Kit Motion Class - AD'. To connect to another project, run 'edge-impulse-data-forwarder --clean'.
[WS] Go to https://studio.edgeimpulse.com/studio/750061/acquisition/training to build your machine learning model
  
```

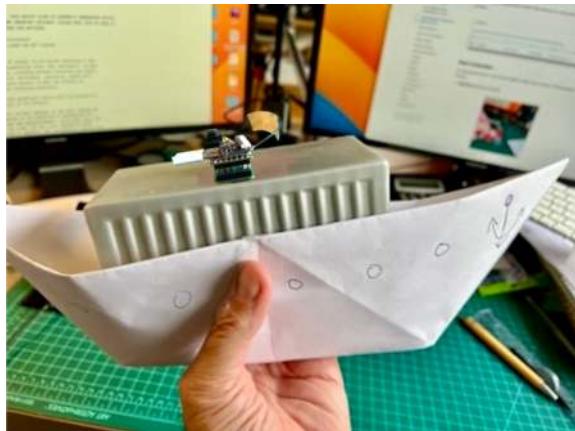
- Go to the Edge Impulse Studio Project. On the Device section is possible to verify if the kit is correctly connected (the dot should be green).



Data Collection at the Studio

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

celerometer (In this case, our XIAOMI Kit). Now imagine your container is on a boat, facing an angry ocean:



Or in a Truck, travelling on a road, or being moved with a forklift, etc.

Movement Simulation

Maritime Class:

- Hold the kit and simulate boat movement
- Move in all three axes with wave-like, undulating motions
- Include gentle rolling and pitching movements

Terrestrial Class:

- Move the kit horizontally in straight lines (left to right and vice versa)
- Simulate truck/train vibrations with small horizontal shakes
- Occasional gentle bumps and turns

Lift Class:

- Move the kit primarily in vertical directions (up and down)
- Simulate forklift operations: up, pause, down
- Include some short horizontal positioning movements

Idle Class:

- Place the kit on a stable surface
- Minimal to no movement
- Capture environmental vibrations and sensor noise

Data Acquisition

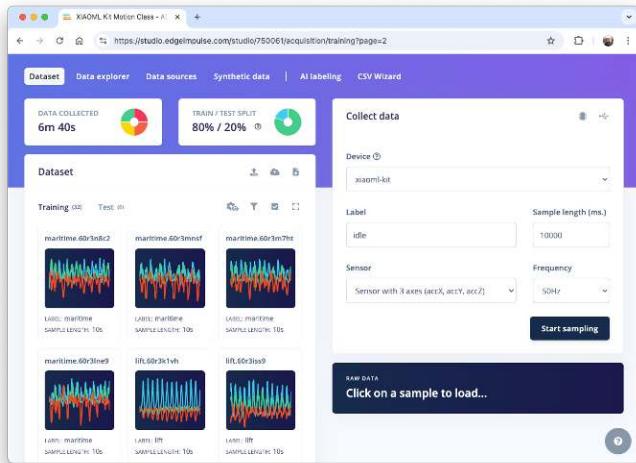
On the Data Acquisition section, you should see that your board [xiaoml-kit] is connected. The sensor is available: [sensor with 3 axes (accX, accY, accZ)] with a sampling frequency of [50 Hz]. The Studio suggests a sample length of [10000] ms (10 s). The last thing left is defining the sample label.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

Press [Start Sample] and move your kit horizontally (left to right), keeping it in one direction. After 10 seconds, our data will be uploaded to the Studio.

Below is one sample (raw data) of 10 seconds of collected data. It is notable that the ondulatory movement predominantly occurs along the Y-axis (left-right). The other axes are almost stationary (the X-axis is centered around zero, and the Z-axis is centered around 9.8 ms^2 due to gravity).

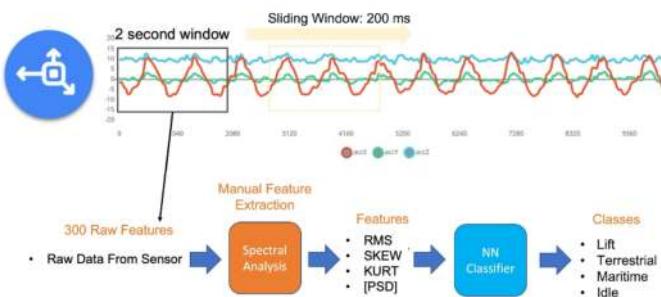


You should capture, for example, around 2 minutes (ten to twelve samples of 10 seconds each) for each of the four classes. Using the 3 dots after each sample, select two and move them to the **Test set**. Alternatively, you can use the Automatic Train/Test Split tool on the **Danger Zone** of the Dashboard tab. Below, it is possible to see the result datasets:



Data Pre-Processing

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



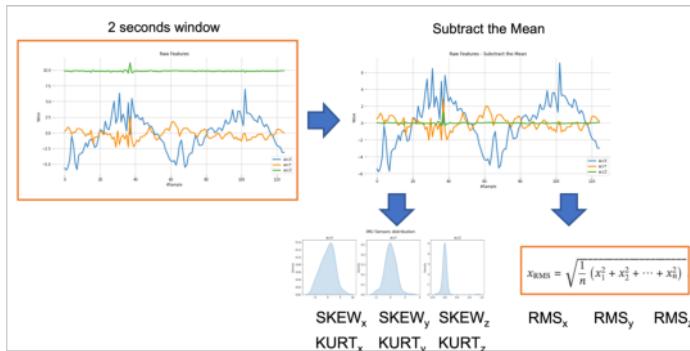
We can see 10 seconds of accelerometer data captured with a sample rate (SR) of 50 Hz. A 2-second window will capture 300 data points (3 axes × 2 seconds × 50 samples). We will slide this window every 200ms, creating a larger dataset where each instance has 300 raw features.

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

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

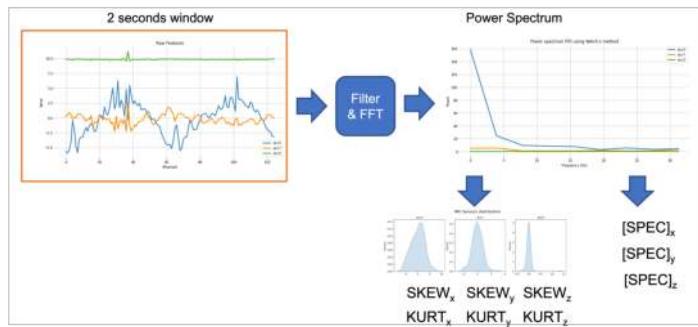
On the Studio, this dataset will be the input of a Spectral Analysis block, which is excellent for analyzing repetitive motion, such as data from accelerometers. This block will perform a DSP (Digital Signal Processing), extracting features such as "FFT" or "Wavelets". In the most common case, FFT, the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

- RMS
- Skewness
- Kurtosis



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

- Spectral Power
- Skewness
- Kurtosis



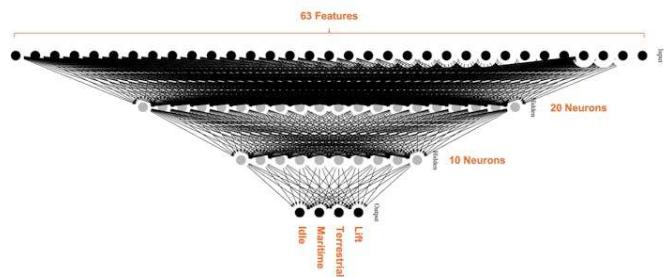
For example, for an FFT length of 32 points, the Spectral Analysis Block's resulting output will be 21 features per axis (a total of 63 features).

Those 63 features will serve as the input tensor for a Neural Network Classifier and the Anomaly Detection model (K-Means).

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

Model Design

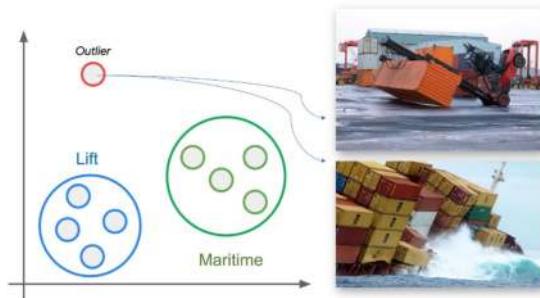
Our classifier will be a Dense Neural Network (DNN) that will have 63 neurons on its input layer, two hidden layers with 20 and 10 neurons, and an output layer with four neurons (one per each class), as shown here:



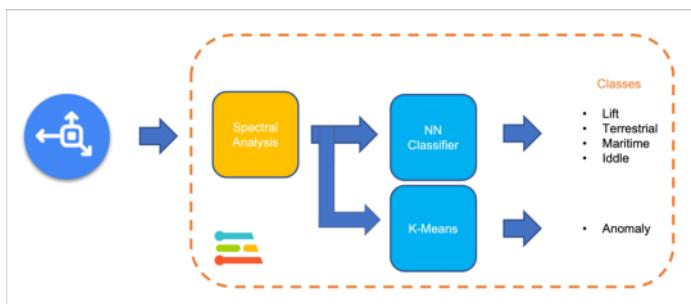
Impulse Design

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

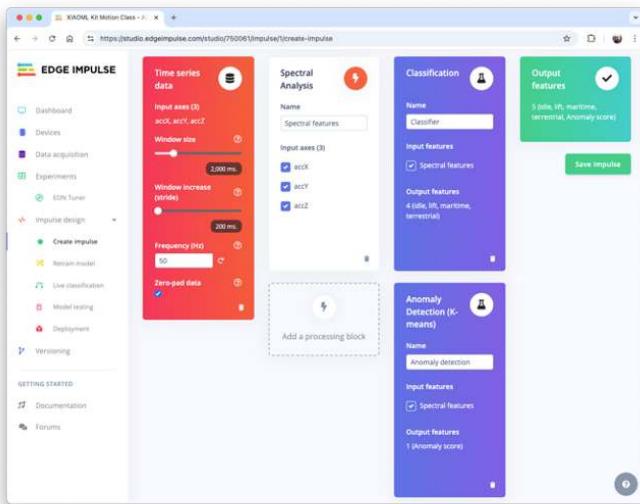
We also utilize a second model, the **K-means**, which can be used for Anomaly Detection. If we imagine that we could have our known classes as clusters, any sample that cannot fit into one of these clusters could be an outlier, an anomaly (for example, a container rolling out of a ship on the ocean or being upside down on the floor).



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

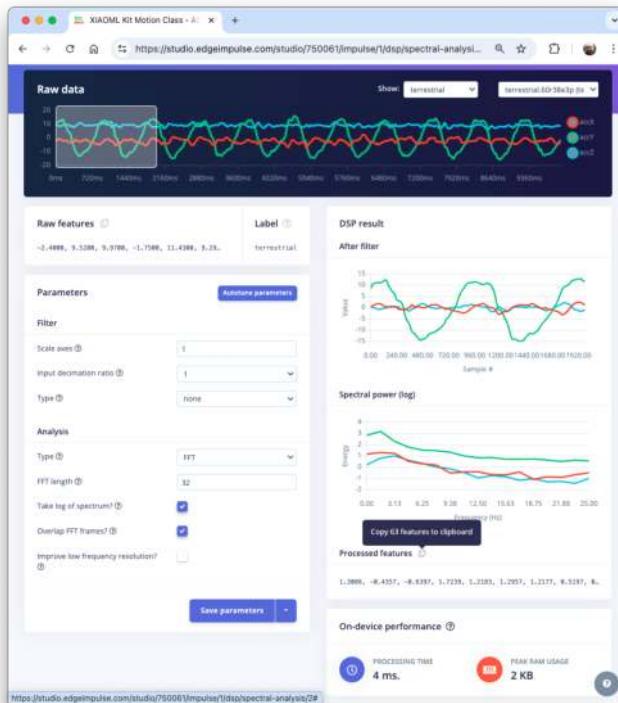


Below the final Impulse design:



Generating features

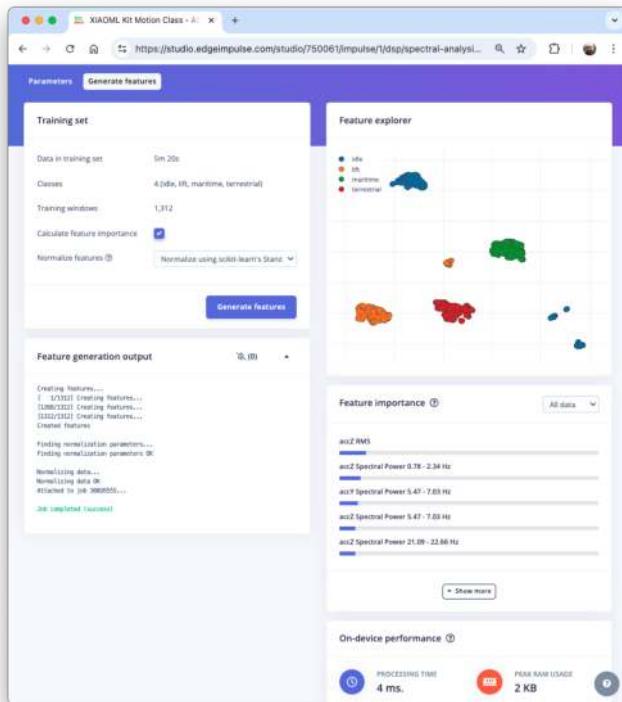
At this point in our project, we have defined the pre-processing method, and the model has been designed. Now, it is time to have the job done. First, let's convert the raw data (time-series type) into tabular data. Go to the Spectral Features tab and select [Save Parameters]. Alternatively, instead of using the default values, we can select the [Autotune parameters] button. In this case, the Studio will define new hyperparameters, as the filter design and FFT length, based on the raw data.



At the top menu, select the `Generate features` tab, and there, select the options, `Calculate feature importance`, `Normalize features`, and press the `[Generate features]` button. Each 2-second window of data (300 datapoints) will be converted into a single tabular data point with 63 features.

The Feature Explorer will display this data in 2D using UMAP. Uniform Manifold Approximation and Projection (UMAP) is a dimensionality reduction technique that can be used for visualization, similar to t-SNE, but also for general non-linear dimensionality reduction.

The visualization enables one to verify that the classes present an excellent separation, indicating that the classifier should perform well.



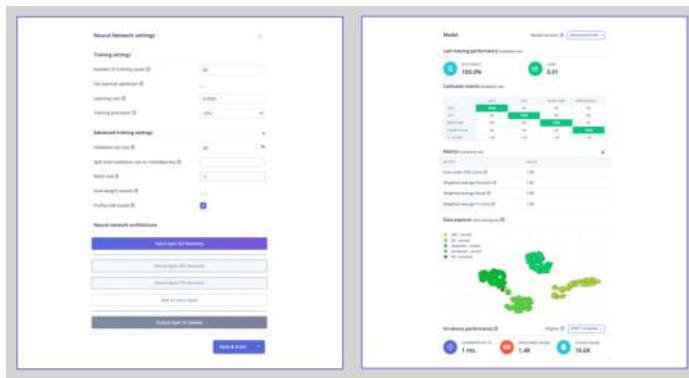
Optionally, you can analyze the relative importance of each feature for one class compared with other classes.

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 and 20% of the data for validation for 30 epochs. After training, we can see that the accuracy is 100%.



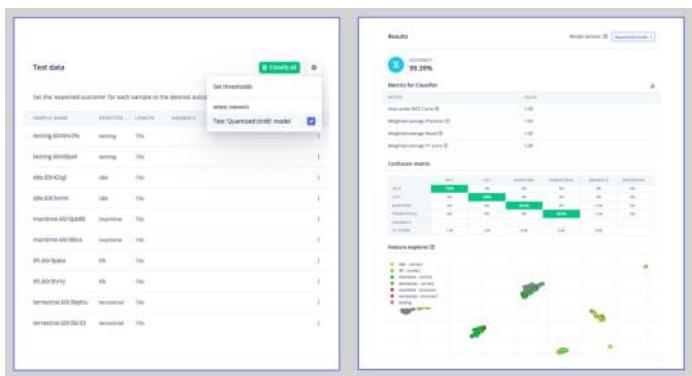
For anomaly detection, we should choose the suggested features that are precisely the most important in feature extraction. The number of clusters will be 32, as suggested by the Studio. After training, we can select some data for testing, such as maritime data. The resulting Anomaly score was min: -0.1642, max: 0.0738, avg: -0.0867.

When changing the data, it is possible to realize that small or negative Anomaly Scores indicate that the data are normal.



Testing

Using 20% of the data left behind during the data capture phase, we can verify how our model will behave with unknown data; if not 100% (what is expected), the result was very good (8%).



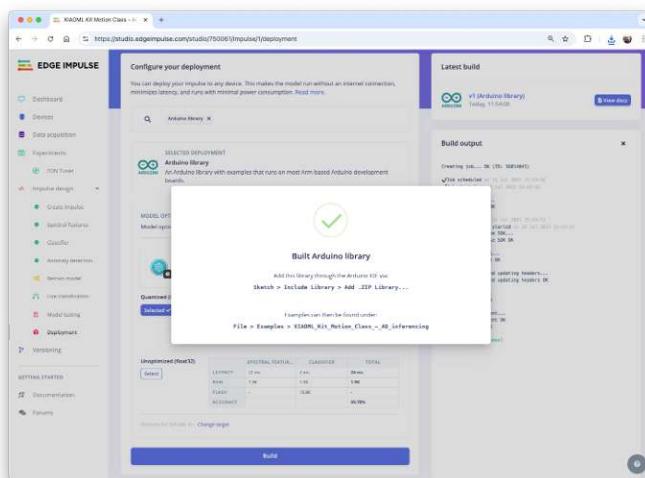
You should also use your kit (which is still connected to the Studio) and perform some Live Classification. For example, let's test some "terrestrial" movement:



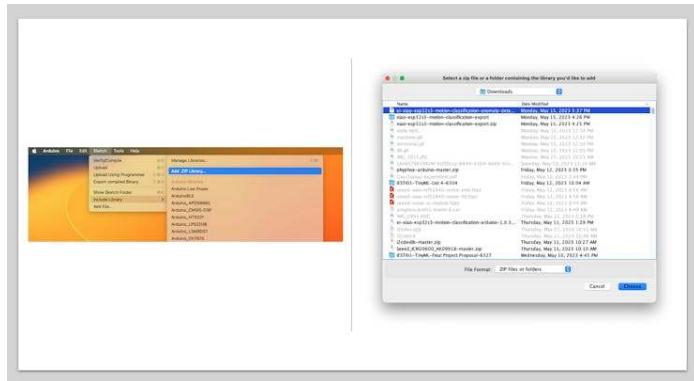
Be aware that here, you will capture real data with your device and upload it to the Studio, where an inference will be made using the trained model (note that the model is not on your device).

Deploy

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



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



Inference

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

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

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

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

```
/* Includes ----- */  
#include <XIAOML_Kit_Motion_Class_-_AD_inferencing.h>  
#include <Arduino_LSM9DS1.h>
```

Change the "includes" portion with the code related to the IMU:

```
#include <XIAOML_Kit_Motion_Class_-_AD_inferencing.h>  
#include <LSM6DS3.h>  
#include <Wire.h>
```

Change the Constant Defines

```
// IMU setup
LSM6DS3 myIMU(I2C_MODE, 0x6A);

// Inference settings
#define CONVERT_G_TO_MS2    9.81f
#define MAX_ACCEPTED_RANGE  2.0f * CONVERT_G_TO_MS2
```

On the setup function, initiate the IMU:

```
// Initialize IMU
if (myIMU.begin() != 0) {
    Serial.println("ERROR: IMU initialization failed!");
    return;
}
```

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

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

Change it with this block of code:

```
// Read IMU data
float x = myIMU.readFloatAccelX();
float y = myIMU.readFloatAccelY();
float z = myIMU.readFloatAccelZ();
```

You should reorder the following two blocks of code. First, you make the conversion to raw data to “Meters per squared second (m/s^2)”, followed by the test regarding the maximum acceptance range (that here is in m/s^2 , but on Arduino, was in Gs):

```
// Convert to m/s2
buffer[i + 0] = x * CONVERT_G_TO_MS2;
buffer[i + 1] = y * CONVERT_G_TO_MS2;
buffer[i + 2] = z * CONVERT_G_TO_MS2;

// Apply range limiting
for (int j = 0; j < 3; j++) {
    if (fabs(buffer[i + j]) > MAX_ACCEPTED_RANGE) {
        buffer[i + j] = copysign(MAX_ACCEPTED_RANGE, buffer[i + j]);
    }
}
```

And this is enough. We can also adjust how the inference is displayed in the Serial Monitor. You can now upload the complete code below to your device and proceed with the inferences.

```
// Motion Classification with LSM6DS3TR-C IMU
#include <XIAOML_Kit_Motion_Class_-_AD_inferencing.h>
#include <LSM6DS3.h>
#include <Wire.h>

// IMU setup
LSM6DS3 myIMU(I2C_MODE, 0x6A);

// Inference settings
#define CONVERT_G_TO_MS2    9.81f
#define MAX_ACCEPTED_RANGE  2.0f * CONVERT_G_TO_MS2

static bool debug_nn = false;
static float buffer[EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE] = { 0 };
static float inference_buffer[EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE];

void setup() {
    Serial.begin(115200);
    while (!Serial) delay(10);

    Serial.println("XIAOML Kit - Motion Classification");
    Serial.println("LSM6DS3TR-C IMU Inference");

    // Initialize IMU
    if (myIMU.begin() != 0) {
        Serial.println("ERROR: IMU initialization failed!");
        return;
    }

    Serial.println(" IMU initialized");

    if (EI_CLASSIFIER_RAW_SAMPLES_PER_FRAME != 3) {
        Serial.println("ERROR: EI_CLASSIFIER_RAW_SAMPLES_PER_FRAME"
                      "should be 3");
        return;
    }

    Serial.println(" Model loaded");
    Serial.println("Starting motion classification...");
}

void loop() {
    ei_printf("\nStarting inferencing in 2 seconds...\n");
}
```

```
delay(2000);

ei_printf("Sampling...\n");

// Clear buffer
for (size_t i = 0; i < EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE; i++) {
    buffer[i] = 0.0f;
}

// Collect accelerometer data
for (int i = 0; i < EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE; i += 3) {
    uint64_t next_tick = micros() +
        (EI_CLASSIFIER_INTERVAL_MS * 1000);

    // Read IMU data
    float x = myIMU.readFloatAccelX();
    float y = myIMU.readFloatAccelY();
    float z = myIMU.readFloatAccelZ();

    // Convert to m/s2
    buffer[i + 0] = x * CONVERT_G_TO_MS2;
    buffer[i + 1] = y * CONVERT_G_TO_MS2;
    buffer[i + 2] = z * CONVERT_G_TO_MS2;

    // Apply range limiting
    for (int j = 0; j < 3; j++) {
        if (fabs(buffer[i + j]) > MAX_ACCEPTED_RANGE) {
            buffer[i + j] = copysign(MAX_ACCEPTED_RANGE,
                                      buffer[i + j]);
        }
    }
}

delayMicroseconds(next_tick - micros());
}

// Copy to inference buffer
for (int i = 0; i < EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE; i++) {
    inference_buffer[i] = buffer[i];
}

// Create signal from buffer
signal_t signal;
int err = numpy::signal_from_buffer(inference_buffer,
```

```
    EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE, &signal);
if (err != 0) {
    ei_printf("ERROR: Failed to create signal from buffer (%d)\n",
              err);
    return;
}

// Run the classifier
ei_impulse_result_t result = { 0 };
err = run_classifier(&signal, &result, debug_nn);
if (err != EI_IMPULSE_OK) {
    ei_printf("ERROR: Failed to run classifier (%d)\n", err);
    return;
}

// Print predictions
ei_printf("Predictions (DSP: %d ms, Classification: %d ms, "
          "Anomaly: %d ms):\n",
          result.timing.dsp, result.timing.classification, result.timing.anomaly);

for (size_t ix = 0; ix < EI_CLASSIFIER_LABEL_COUNT; ix++) {
    ei_printf("    %s: %.5f\n", result.classification[ix].label,
              result.classification[ix].value);
}

// Print anomaly score
#if EI_CLASSIFIER_HAS_ANOMALY == 1
    ei_printf("Anomaly score: %.3f\n", result.anomaly);
#endif

// Determine prediction
float max_confidence = 0.0;
String predicted_class = "unknown";

for (size_t ix = 0; ix < EI_CLASSIFIER_LABEL_COUNT; ix++) {
    if (result.classification[ix].value > max_confidence) {
        max_confidence = result.classification[ix].value;
        predicted_class = String(result.classification[ix].label);
    }
}

// Display result with confidence threshold
if (max_confidence > 0.6) {
```

```

        ei_printf("\n PREDICTION: %s (%.1f%% confidence)\n",
                  predicted_class.c_str(), max_confidence * 100);
    } else {
        ei_printf("\n UNCERTAIN: Highest confidence is %s (%.1f%%)\n",
                  predicted_class.c_str(), max_confidence * 100);
    }

    // Check for anomaly
#ifndef EI_CLASSIFIER_HAS_ANOMALY == 1
    if (result.anomaly > 0.5) {
        ei_printf(" ANOMALY DETECTED! Score: %.3f\n", result.anomaly);
    }
#endif

    delay(1000);
}

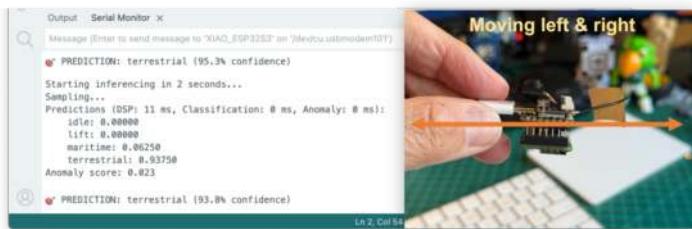
void ei_printf(const char *format, ...) {
    static char print_buf[1024] = { 0 };
    va_list args;
    va_start(args, format);
    int r = vsnprintf(print_buf, sizeof(print_buf), format, args);
    va_end(args);
    if (r > 0) {
        Serial.write(print_buf);
    }
}

```

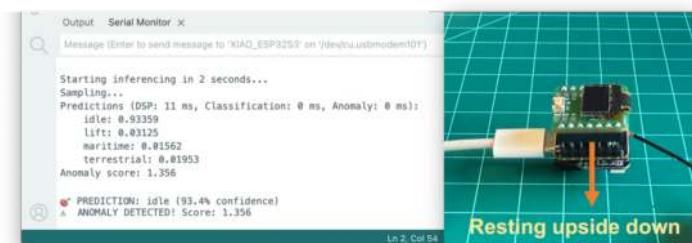
The complete code is available on the Lab's GitHub.

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





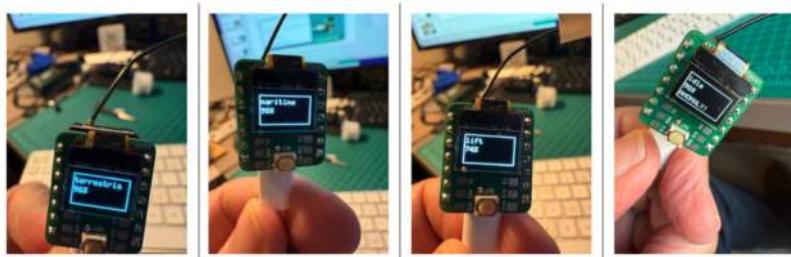
And, of course, some “anomaly”, for example, putting the XIAO upside-down. The anomaly score will be over 0.5:



Post-Processing

Now that we know the model is working, we suggest modifying the code to see the result with the Kit completely offline (disconnected from the PC and powered by a battery, a power bank, or an independent 5V power supply).

The idea is that if a specific movement is detected, a corresponding message will appear on the OLED display.



The modified inference code to have the OLED display is available on the Lab's GitHub.

Summary

This lab demonstrated how to build a complete motion classification system using the XIAOMI Kit's built-in LSM6DS3TR-C IMU sensor. Key achievements include:

Technical Implementation:

- Utilized the integrated 6-axis IMU for motion sensing
- Collected labeled training data for four transportation scenarios
- Implemented spectral feature extraction for time-series analysis
- Deployed a neural network classifier optimized for microcontroller inference
- Added anomaly detection for identifying unusual movements

Machine Learning Pipeline:

- Data collection directly from embedded sensors
- Feature engineering using frequency domain analysis
- Model training and optimization in Edge Impulse

- Real-time inference on resource-constrained hardware
- Performance monitoring and validation

Practical Applications: The techniques learned apply directly to real-world scenarios, including:

- Asset tracking and logistics monitoring
- Predictive maintenance for machinery
- Human activity recognition
- Vehicle and equipment monitoring
- IoT sensor networks for smart cities

Key Learnings:

- Working with IMU coordinate systems and sensor fusion
- Balancing model accuracy with inference speed on edge devices
- Implementing robust data collection and preprocessing pipelines
- Deploying machine learning models to embedded systems
- Integrating multiple sensors (IMU + display) for complete solutions

The integration of motion classification with the XIAOML Kit demonstrates how modern embedded systems can perform sophisticated AI tasks locally, enabling real-time decision-making without reliance on the cloud. This approach is fundamental to the future of edge AI in industrial IoT, autonomous systems, and smart device applications.

Resources

- XIAOML KIT Code
- DSP Spectral Features
- Edge Impulse Project
- Edge Impulse Spectral Features Block Colab Notebook
- Edge Impulse Documentation
- Edge Impulse Spectral Features
- Seeed Studio LSM6DS3 Library

V

KEY:GROVE

Part V

VI

GROVE VISION AI V2

Part VI

Overview

These labs offer an opportunity to gain practical experience with machine learning (ML) systems on a high-end, yet compact, embedded device, the Seeed Studio Grove Vision AI V2. Unlike working with large models requiring data center-scale resources, these labs allow you to interact with hardware and software using TinyML directly. This hands-on approach provides a tangible understanding of the challenges and opportunities in deploying AI, albeit on a small scale. However, the principles are essentially the same as what you would encounter when working with larger or even smaller systems.

The Grove Vision AI V2 occupies a unique position in the embedded AI landscape, bridging the gap between basic microcontroller solutions, such as the Seeed XIAO ESP32S3 Sense or Arduino Nicla Vision, and more powerful single-board computers, like the Raspberry Pi. At its heart lies the Himax WiseEye2 HX6538 processor, featuring a **dual-core Arm Cortex-M55 and an integrated ARM Ethos-U55 neural network unit**.

The Arm Ethos-U55 represents a specialized machine learning processor class, specifically designed as a microNPU to accelerate ML inference in area-constrained embedded and IoT devices. This powerful combination of the Ethos-U55 with the AI-capable Cortex-M55 processor delivers a remarkable 480x uplift in ML performance over existing Cortex-M-based systems. Operating at 400 MHz with configurable internal system memory (SRAM) up to 2.4 MB, the Grove Vision AI V2 offers professional-grade computer vision capabilities while maintaining the power efficiency and compact form factor essential for edge applications.

This positioning makes it an ideal platform for learning advanced TinyML concepts, offering the simplicity and reduced power require-

ments of smaller systems while providing capabilities that far exceed those of traditional microcontroller-based solutions.

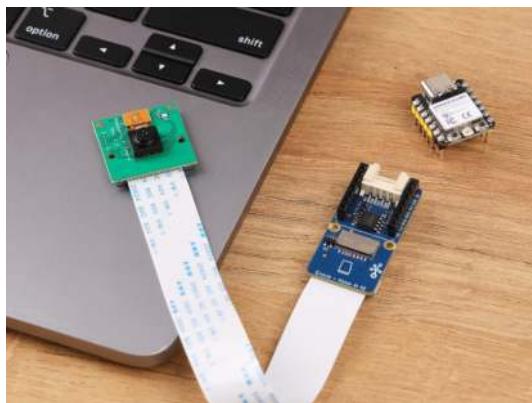


Figure 1.18: Grove - Vision AI Module V2. Source: SEEED Studio.

Where to Buy

The Grove Vision AI V2 is available from Seeed Studio:

- Grove Vision AI V2 (Seeed Studio) (~\$25)

You will also need a compatible camera module (Raspberry Pi OV5647) and optionally a master controller like the XIAO ESP32S3.

Pre-requisites

- **Grove Vision AI V2 Board:** Ensure you have the Grove Vision AI V2 Board.
- **Raspberry Pi OV5647 Camera Module:** The camera should be connected to the Grove Vision AI V2 Board for image capture.
- **Master Controller:** Can be a Seeed XIAO ESP32S3, a XIAO ESP32C6, or other devices.
- **USB-C Cable:** This is for connecting the board to your computer.
- **Network:** With internet access for downloading the necessary software.
- **XIAO Expansion Board Base:** This helps connect the Master Device to the Physical World (optional).

Setup and No-Code Applications

- Setup and No-Code Apps

Exercises

Modality	Task	Description	Link
Vision	Image Classification	Learn to classify images	Link
Vision	Object Detection	Implement object detection	TBD

Setup and No-Code Applications



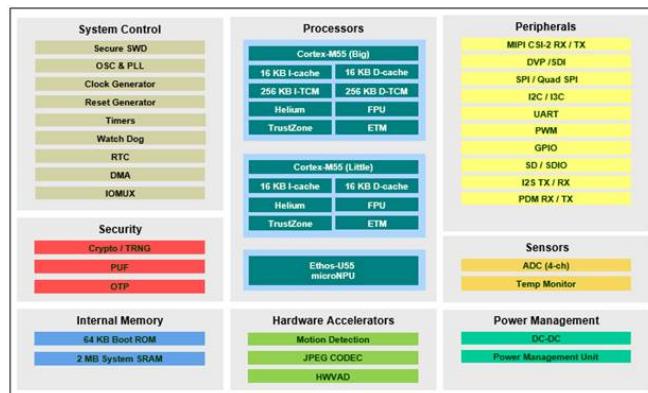
In this Lab, we will explore computer vision (CV) applications using the Seeed Studio *Grove Vision AI Module V2*, a powerful yet compact device specifically designed for embedded machine learning applications. Based on the **Himax WiseEye2** chip, this module is designed to enable AI capabilities on edge devices, making it an ideal tool for Edge Machine Learning (ML) applications.

Introduction

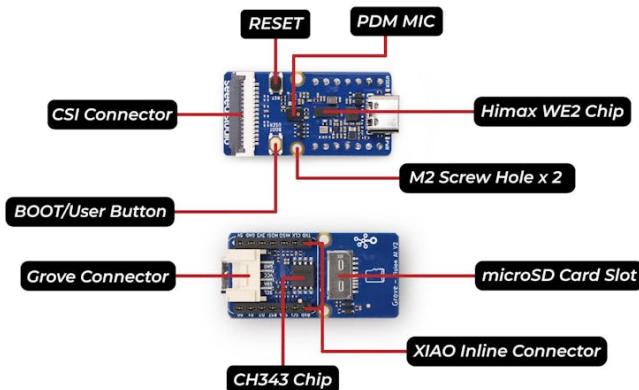
Grove Vision AI Module (V2) Overview



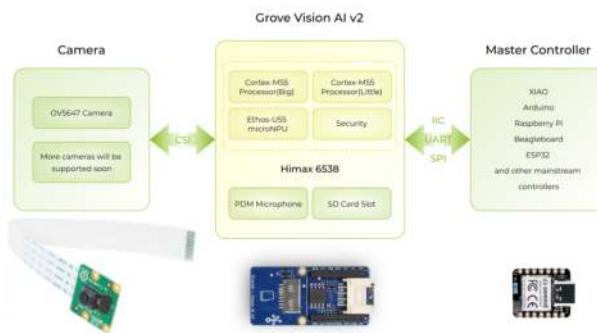
The Grove Vision AI (V2) is an MCU-based vision AI module that utilizes a Himax WiseEye2 HX6538 processor featuring a **dual-core Arm Cortex-M55 and an integrated ARM Ethos-U55 neural network unit**. The Arm Ethos-U55 is a machine learning (ML) processor class, specifically designed as a microNPU, to accelerate ML inference in area-constrained embedded and IoT devices. The Ethos-U55, combined with the AI-capable Cortex-M55 processor, provides a 480x uplift in ML performance over existing Cortex-M-based systems. Its clock frequency is 400 MHz, and its internal system memory (SRAM) is configurable, with a maximum capacity of 2.4 MB.



Note: Based on Seeed Studio documentation, besides the Himax internal memory of 2.5MB (2.4MB SRAM + 64KB ROM), the Grove Vision AI (V2) is also equipped with a 16MB/133 MHz external flash.

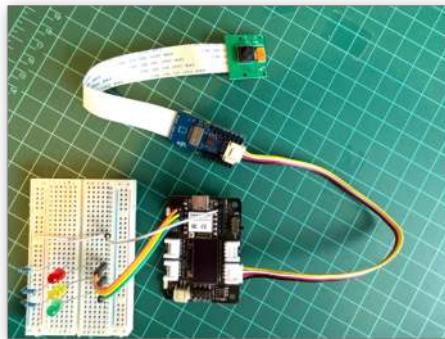


Below is a block Diagram of the Grove Vision AI (V2) system, including a camera and a master controller.

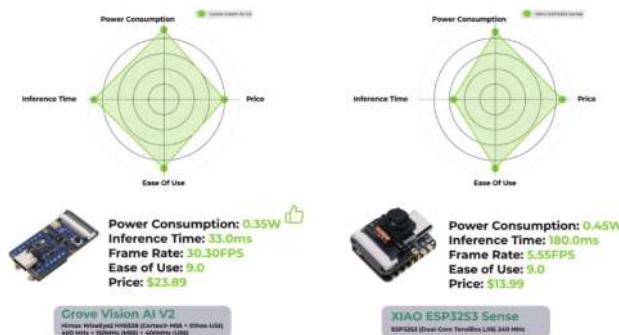


With interfaces like **IIC**, **UART**, **SPI**, and **Type-C**, the Grove Vision AI (V2) can be easily connected to devices such as **XIAO**, **Raspberry Pi**, **BeagleBoard**, and **ESP-based products** for further development. For instance, integrating Grove Vision AI V2 with one of the devices from the XIAO family makes it easy to access the data resulting from inference on the device through the Arduino IDE or MicroPython, and conveniently connect to the cloud or dedicated servers, such as Home Assistance.

Using the **I2C Grove connector**, the Grove Vision AI V2 can be easily connected with any Master Device.



Besides performance, another area to comment on is **Power Consumption**. For example, in a comparative test against the XIAO ESP32S3 Sense, running Swift-YOLO Tiny 96x96, despite achieving higher performance (30 FPS vs. 5.5 FPS), the Grove Vision AI V2 exhibited lower power consumption (0.35 W vs. 0.45 W) when compared with the XIAO ESP32S3 Sense.



The above comparison (and with other devices) can be found in the article 2024 MCU AI Vision Boards: Performance Comparison, which confirms the power of Grove Vision AI (V2).

Camera Installation

Having the Grove Vision AI (V2) and camera ready, you can connect, for example, a **Raspberry Pi OV5647 Camera Module** via the CSI cable.

When connecting, please pay attention to the direction of the row of pins and ensure they are plugged in correctly, not in the opposite direction.



The SenseCraft AI Studio

The SenseCraft AI Studio is a robust platform that offers a wide range of AI models compatible with various devices, including the XIAO ESP32S3 Sense and the **Grove Vision AI V2**. In this lab, we will walk through the process of using an AI model with the Grove Vision AI V2 and preview the model's output. We will also explore some key concepts, settings, and how to optimize the model's performance.



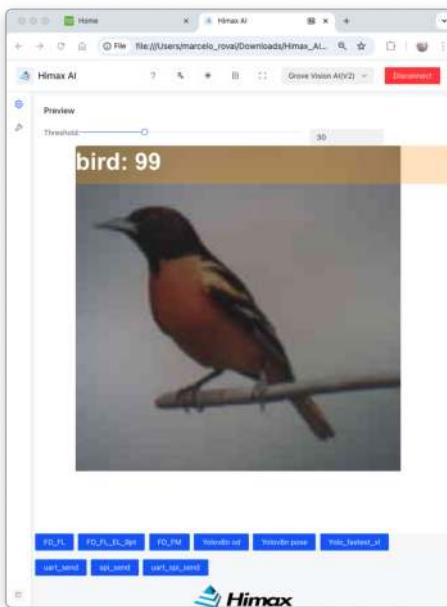
Models can also be deployed using the **SenseCraft Web Toolkit**, a simplified version of the SenseCraft AI Studio.

We can start using the SenseCraft Web Toolkit for simplicity, or go directly to the SenseCraft AI Studio, which has more resources.

The SenseCraft Web-Toolkit

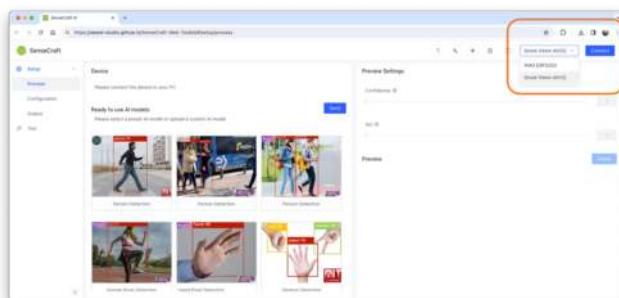
The SenseCraft Web Toolkit is a visual model deployment tool included in the SSCMA(Seeed SenseCraft Model Assistant). This tool enables us to deploy models to various platforms with ease through simple operations. The tool offers a user-friendly interface and does not require any coding.

The SenseCraft Web Toolkit is based on the Himax AI Web Toolkit, which can (**optionally**) be downloaded from [here](#). Once downloaded and unzipped to the local PC, double-click `index.html` to run it locally.

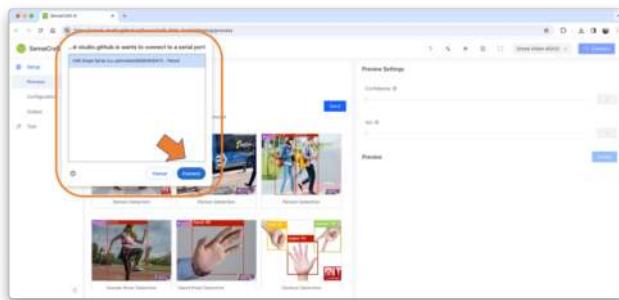


But in our case, let's follow the steps below to start the **SenseCraft-Web-Toolkit**:

- Open the SenseCraft-Web-Toolkit website on a web browser as **Chrome**.
 - Connect Grove Vision AI (V2) to your computer using a Type-C cable.
 - Having the XIAO connected, select it as below:



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



Note: The **WebUSB tool** may not function correctly in certain browsers, such as Safari. Use Chrome instead.

We can try several Basic Computer Vision models previously uploaded by Seeed Studio. Passing the cursor over the AI models, we can have some information about them, such as name, description, **category** (Image Classification, Object Detection, or Pose/Keypoint Detection), the **algorithm** (like YOLO V5 or V8, FOMO, MobileNet V2, etc.) and **metrics** (Accuracy or mAP).



We can choose one of those ready-to-use AI models by clicking on it and pressing the [Send] button, or upload our model.

For the **SenseCraft AI** platform, follow the instructions here.

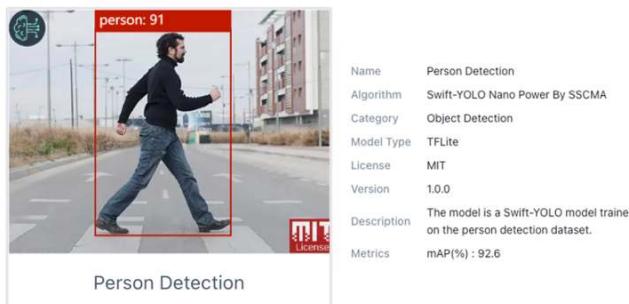
Exploring CV AI models

Object Detection

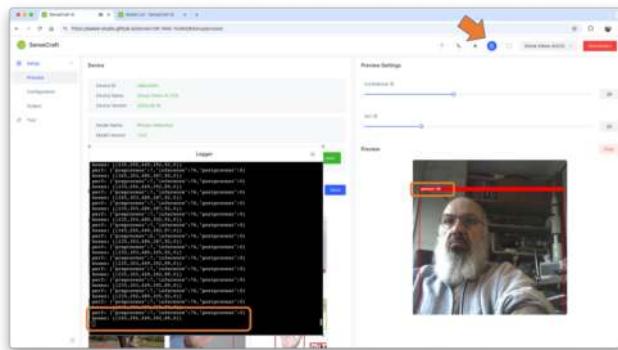
Object detection is a pivotal technology in computer vision that focuses on identifying and locating objects within digital images or video frames. Unlike image classification, which categorizes an entire image into a single label, object detection recognizes multiple objects within the image and determines their precise locations, typically represented by bounding boxes. This capability is crucial for a wide range of applications, including autonomous vehicles, security, surveillance systems, and augmented reality, where understanding the context and content of the visual environment is essential.

Common architectures that have set the benchmark in object detection include the YOLO (You Only Look Once), SSD (Single Shot MultiBox Detector), FOMO (Faster Objects, More Objects), and Faster R-CNN (Region-based Convolutional Neural Networks) models.

Let's choose one of the ready-to-use AI models, such as **Person Detection**, which was trained using the Swift-YOLO algorithm.



Once the model is uploaded successfully, you can see the live feed from the Grove Vision AI (V2) camera in the Preview area on the right. Also, the inference details can be shown on the Serial Monitor by clicking on the [Device Log] button at the top.



In the SenseCraft AI Studio, the Device Logger is always on the screen.

Pointing the camera at me, only one person was detected, so that the model output will be a single “box”. Looking in detail, the module sends continuously two lines of information:

```
perf: {"preprocess":7,"inference":76,"postprocess":0}
boxes: [[245,292,449,392,0]]
```

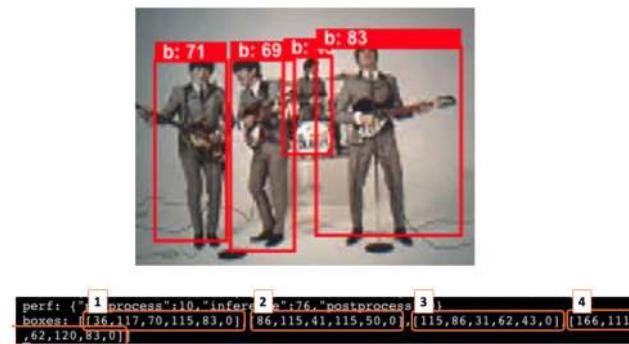
perf (Performance), displays latency in milliseconds.

- Preprocess time (image capture and Crop): **7ms**;
- Inference time (model latency): **76ms (13 fps)**
- Postprocess time (display of the image and inclusion of data): less than 0ms.

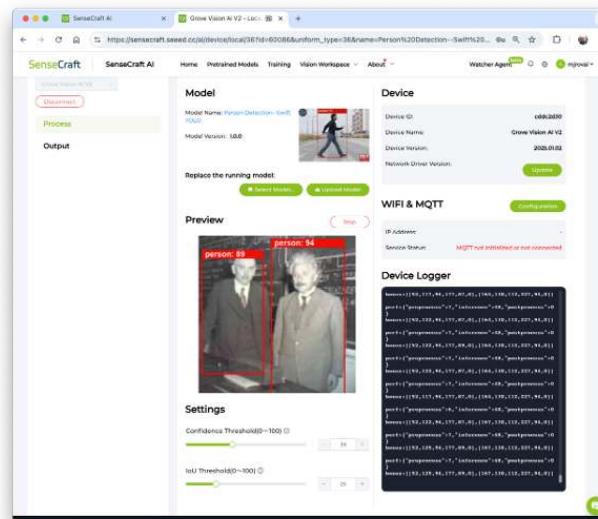
boxes: Show the objects detected in the image. In this case, only one.

- The box has the x, y, w, and h coordinates of **(245, 292, 449, 392)**, and the object (person, label **0**) was captured with a value of **.89**.

If we point the camera at an image with several people, we will get one box for each person (object):



On the SenseCraft AI Studio, the inference latency (48ms) is lower than on the SenseCraft ToolKit (76ms), due to a distinct deployment implementation.



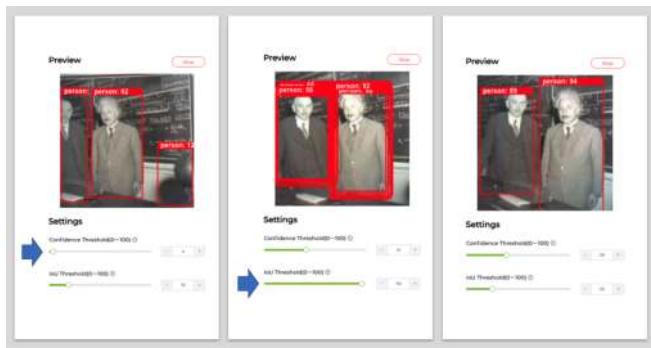
Power Consumption

The peak power consumption running this Swift-YOLO model was 410 milliwatts.

Preview Settings

We can see that in the Settings, two settings options can be adjusted to optimize the model's recognition accuracy.

- **Confidence:** Refers to the level of certainty or probability assigned to its predictions by a model. This value determines the minimum confidence level required for the model to consider a detection as valid. A higher confidence threshold will result in fewer detections but with higher certainty, while a lower threshold will allow more detections but may include some false positives.
- **IoU:** Used to assess the accuracy of predicted bounding boxes compared to truth bounding boxes. IoU is a metric that measures the overlap between the predicted bounding box and the ground truth bounding box. It is used to determine the accuracy of the object detection. The IoU threshold sets the minimum IoU value required for a detection to be considered a true positive. Adjusting this threshold can help in fine-tuning the model's precision and recall.



Experiment with different values for the Confidence Threshold and IoU Threshold to find the optimal balance between detecting persons accurately and minimizing false positives. The best settings may vary depending on our specific application and the characteristics of the images or video feed.

Pose/Keypoint Detection

Pose or keypoint detection is a sophisticated area within computer vision that focuses on identifying specific points of interest within an image or video frame, often related to human bodies, faces, or other objects of interest. This technology can detect and map out the various keypoints of a subject, such as the **joints on a human body** or the features of a face, enabling the analysis of postures, movements, and gestures. This has profound implications for various applications,

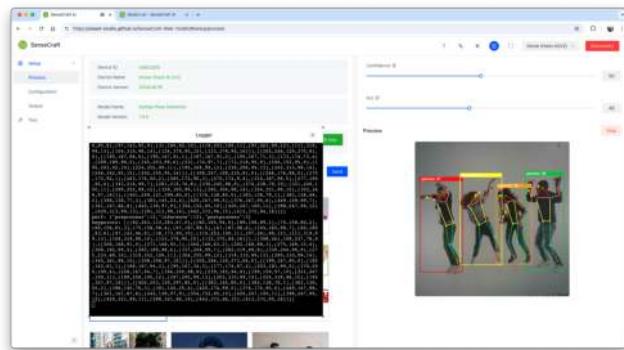
including augmented reality, human-computer interaction, sports analytics, and healthcare monitoring, where understanding human motion and activity is crucial.

Unlike general object detection, which identifies and locates objects, pose detection drills down to a finer level of detail, capturing the nuanced positions and orientations of specific parts. Leading architectures in this field include OpenPose, AlphaPose, and PoseNet, each designed to tackle the challenges of pose estimation with varying degrees of complexity and precision. Through advancements in deep learning and neural networks, pose detection has become increasingly accurate and efficient, offering real-time insights into the intricate dynamics of subjects captured in visual data.

So, let's explore this popular CV application, *Pose/Keypoint Detection*.



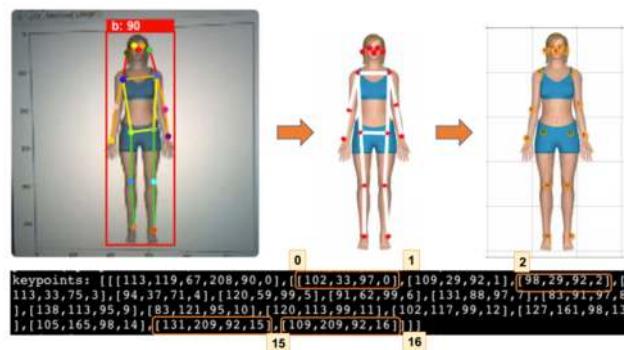
Stop the current model inference by pressing [Stop] in the Preview area. Select the model and press [Send]. Once the model is uploaded successfully, you can view the live feed from the Grove Vision AI (V2) camera in the Preview area on the right, along with the inference details displayed in the Serial Monitor (accessible by clicking the [Device Log] button at the top).



The YOLOV8 Pose model was trained using the COCO-Pose Dataset, which contains 200K images labeled with 17 keypoints for pose estimation tasks.

Let's look at a single screenshot of the inference (to simplify, let's analyse an image with a single person in it). We can note that we have two lines, one with the inference **performance** in milliseconds (121 ms) and a second line with the **keypoints** as below:

- 1 box of info, the same as we got with the object detection example (box coordinates (113, 119, 67, 208), inference result (90), label (0)).
 - 17 groups of 4 numbers represent the 17 “joints” of the body, where ‘0’ is the nose, ‘1’ and ‘2’ are the eyes, ‘15’ and ‘16’ are the feet, and so on.



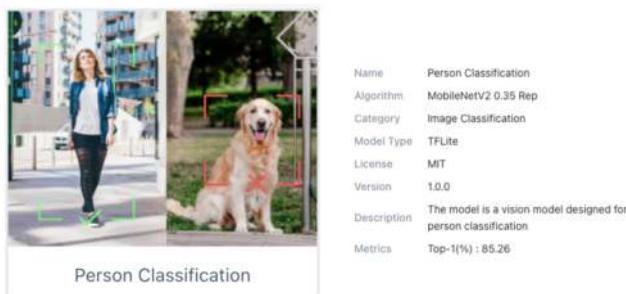
To understand a pose estimation project more deeply, please refer to the tutorial: [Exploring AI at the Edge! - Pose Estimation](#).

Image Classification

Image classification is a foundational task within computer vision aimed at categorizing **entire images** into one of several predefined classes. This process involves analyzing the visual content of an image and assigning it a label from a fixed set of categories based on the predominant object or scene it contains.

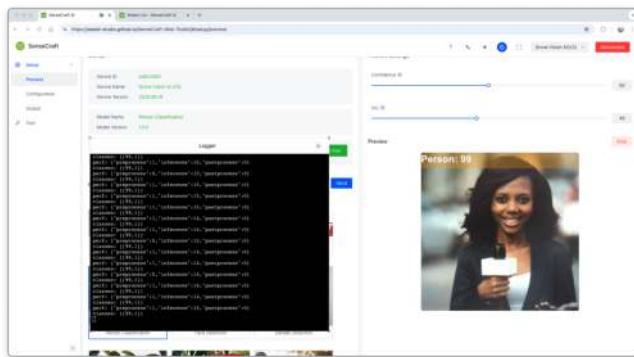
Image classification is crucial in various applications, ranging from organizing and searching through large databases of images in digital libraries and social media platforms to enabling autonomous systems to comprehend their surroundings. Common architectures that have significantly advanced the field of image classification include Convolutional Neural Networks (CNNs), such as AlexNet, VGGNet, and ResNet. These models have demonstrated remarkable accuracy on challenging datasets, such as **ImageNet**, by learning hierarchical representations of visual data.

As the cornerstone of many computer vision systems, image classification drives innovation, laying the groundwork for more complex tasks like object detection and image segmentation, and facilitating a deeper understanding of visual data across various industries. So, let's also explore this computer vision application.



This example is available on the SenseCraft ToolKit, but not in the SenseCraft AI Studio. In the last one, it is possible to find other examples of Image Classification.

After the model is uploaded successfully, we can view the live feed from the Grove Vision AI (V2) camera in the Preview area on the right, along with the inference details displayed in the Serial Monitor (by clicking the [Device Log] button at the top).



As a result, we will receive a score and the class as output.

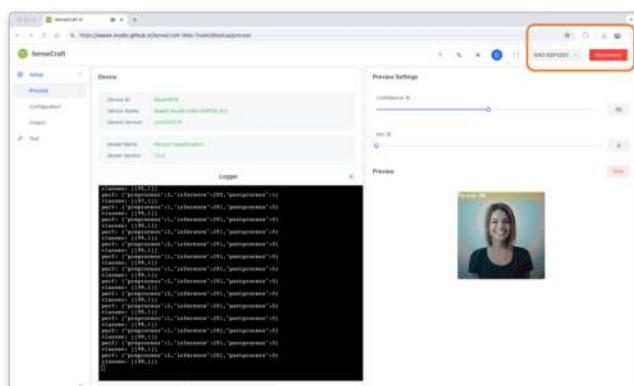
```
perf: {"preprocess":1,"inference":15,"postprocess":0}
classes: [[99,1]]
```

For example, [99, 1] means class: 1 (Person) with a score of 0.99. Once this model is a binary classification, class 0 will be “No Person” (or Background). The Inference latency is **15ms** or around 70fps.

Power Consumption

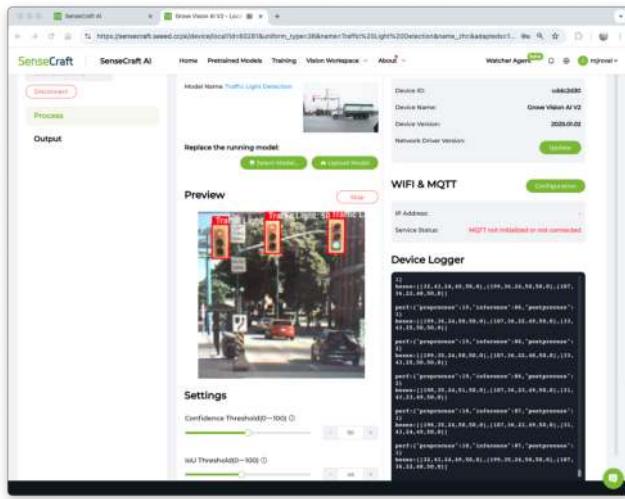
To run the Mobilenet V2 0.35, the Grove Vision AI V2 had a peak current of 80mA at 5.24V, resulting in a **power consumption of 420mW**.

Running the same model on XIAO ESP32S3 Sense, the **power consumption was 523mW** with a latency of 291ms.



Exploring Other Models on SenseCraft AI Studio

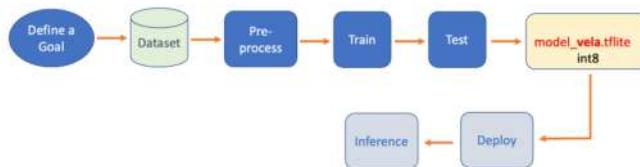
Several public AI models can also be downloaded from the SenseCraft AI WebPage. For example, you can run a Swift-YOLO model, detecting traffic lights as shown here:



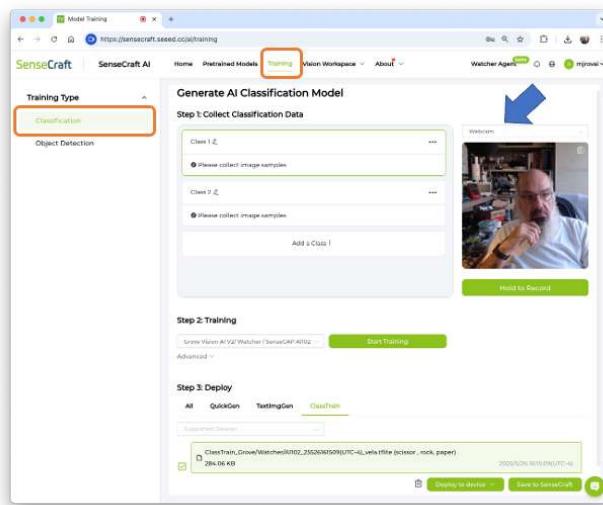
The latency of this model is approximately 86 ms, with an average power consumption of 420 mW.

An Image Classification Project

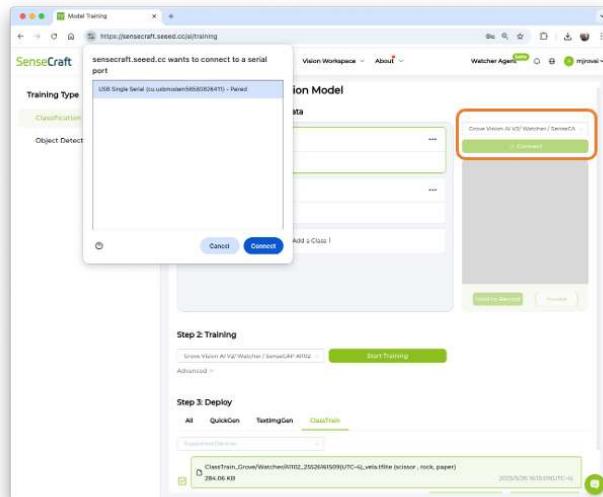
Let's create a complete Image Classification project, using the SenseCraft AI Studio.



On SenseCraft AI Studio: Let's open the tab Training:



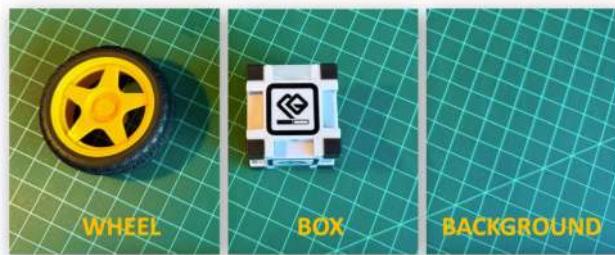
The default is to train a Classification model with a WebCam if it is available. Let's select the Grove Vision AI V2 instead. Pressing the green button [Connect], a Pop-Up window will appear. Select the corresponding Port and press the blue button [Connect].



The image streamed from the Grove Vision AI V2 will be displayed.

The Goal

The first step is always to define a goal. Let's classify, for example, two simple objects—for instance, a toy box and a toy wheel. We should also include a 3rd class of images, background, where no object is in the scene.



Data Collection

Let's create the classes, following, for example, an alphabetical order:

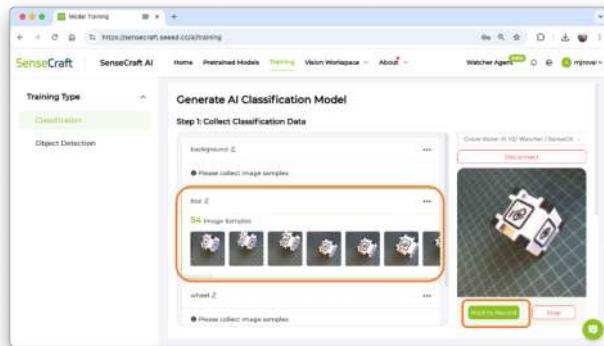
- Class 1: background
- Class 2: box
- Class 3: wheel

Step 1: Collect Classification Data

background ↗	...
Please collect image samples	
box ↗	...
Please collect image samples	
wheel ↗	...
Please collect image samples	

Add a Class +

Select one of the classes and keep pressing the green button under the preview area. The collected images will appear on the Image Samples Screen.



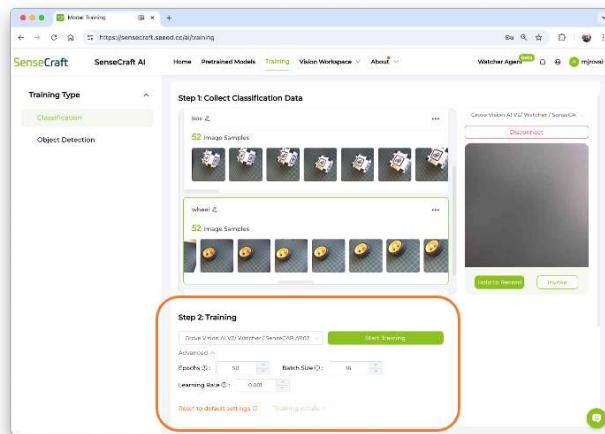
After collecting the images, review them and delete any incorrect ones.



Collect around 50 images from each class and go to Training Step:

Training

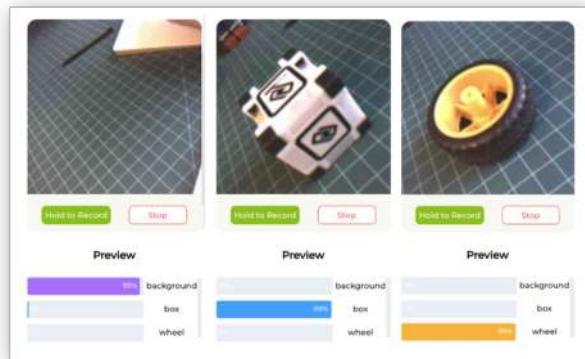
Confirm if the correct device is selected (Grove Vision AI V2) and press [Start Training]



Test

After training, the inference result can be previewed.

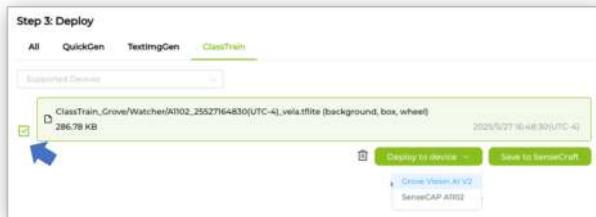
Note that the model is not running on the device. We are, in fact, only capturing the images with the device and performing a live preview using the training model, which is running in the Studio.



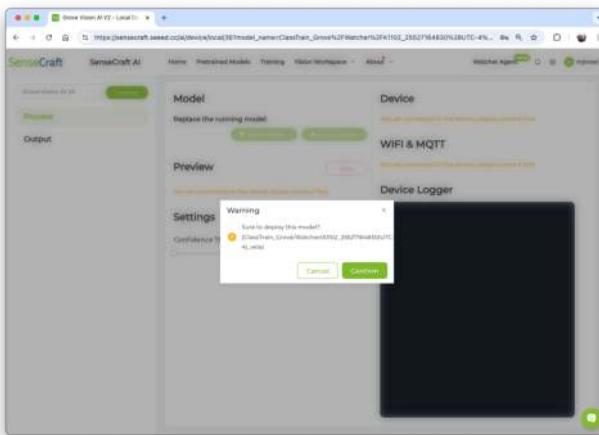
Now is time to really deploy the model in the device:

Deployment

Select the trained model on [Deploy to device], select the Grove Vision AI V2:

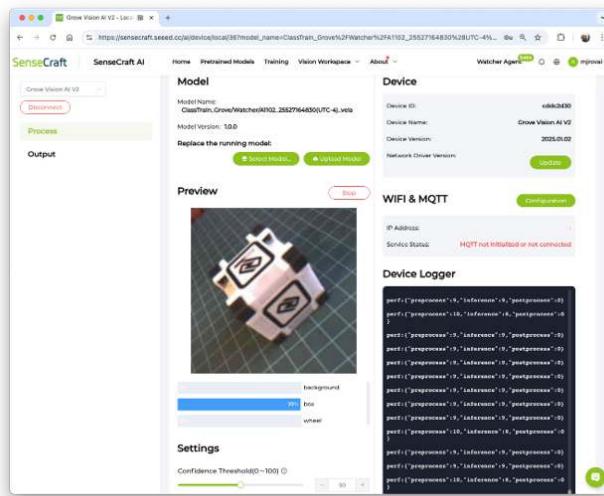


The Studio will redirect us to the Vision Workplace tab. Confirm the deployment, select the appropriate Port, and connect it:



The model will be flashed into the device. After an automatic reset, the model will start running on the device. On the Device Logger, we can see that the inference has a **latency of approximately 8 ms**, corresponding to a **frame rate of 125 frames per second (FPS)**.

Also, note that it is possible to adjust the model's confidence.



To run the Image Classification Model, the Grove Vision AI V2 had a peak current of 80mA at 5.24V, resulting in a **power consumption of 420mW**.

Saving the Model

It is possible to save the model in the SenseCraft AI Studio. The Studio will keep all our models, which can be deployed later. For that, return to the Training tab and select the button [Save to SenseCraft]:



Summary

In this lab, we explored several computer vision (CV) applications using the Seeed Studio Grove Vision AI Module V2, demonstrating its exceptional capabilities as a powerful yet compact device specifically designed for embedded machine learning applications.

Performance Excellence: The Grove Vision AI V2 demonstrated remarkable performance across multiple computer vision tasks. With its **Himax WiseEye2 chip featuring a dual-core Arm Cortex-M55 and integrated ARM Ethos-U55 neural network unit**, the device delivered:

- **Image Classification:** 15 ms inference time (67 FPS)
- **Object Detection (Person):** 48 ms to 76 ms inference time (21 FPS to 13 FPS)
- **Pose Detection:** 121 ms real-time keypoint detection with 17-joint tracking (8 FPS)

Power Efficiency Leadership: One of the most compelling advantages of the Grove Vision AI V2 is its superior power efficiency. Comparative testing revealed significant improvements over traditional embedded platforms:

- **Grove Vision AI V2:** 80 mA (410 mW) peak consumption (60+ FPS)
- **XIAO ESP32S3:** Performing similar CV tasks (Image Classification) 523 mW (3+ FPS)

Practical Implementation: The device's versatility was demonstrated through a comprehensive end-to-end project, encompassing dataset creation, model training, deployment, and offline inference.

Developer-Friendly Ecosystem: The SenseCraft AI Studio, with its no-code deployment and integration capabilities for custom applications, makes the Grove Vision AI V2 accessible to both beginners and advanced developers. The extensive library of pre-trained models and support for custom model deployment provide flexibility for diverse applications.

The Grove Vision AI V2 represents a significant advancement in edge AI hardware, offering professional-grade computer vision capabilities in a compact, energy-efficient package that democratizes AI deployment for embedded applications across industrial, IoT, and educational domains.

Key Takeaways

This Lab demonstrates that sophisticated computer vision applications are not limited to cloud-based solutions or power-hungry hardware, as the Raspberry Pi or Jetson Nanos – they can now be deployed effectively at the edge with remarkable efficiency and performance.

Optionally, we can have the XIAO Vision AI Camera. This innovative vision solution seamlessly combines the Grove Vision AI V2 module, XIAO ESP32-C3 controller, and an OV5647 camera, all housed in a custom 3D-printed enclosure:



Resources

[SenseCraft AI Studio Instructions.](#)

[SenseCraft-Web-Toolkit website.](#)

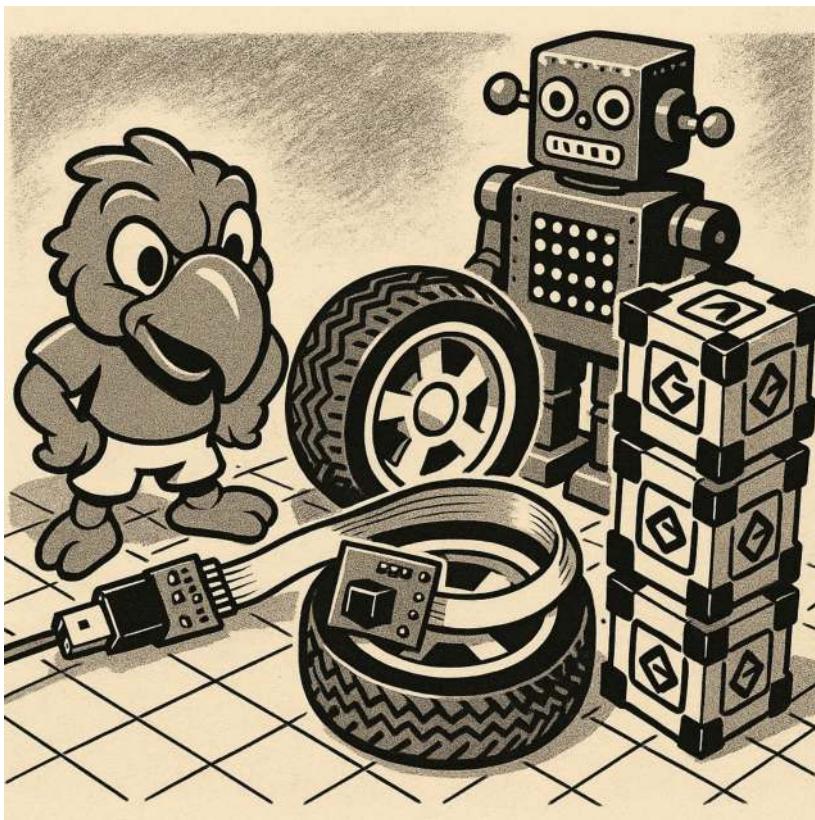
[SenseCraft AI Studio](#)

[Himax AI Web Toolkit](#)

[Himax examples](#)

Image Classification

Using Seeed Studio Grove Vision AI Module V2 (Himax WiseEye2)



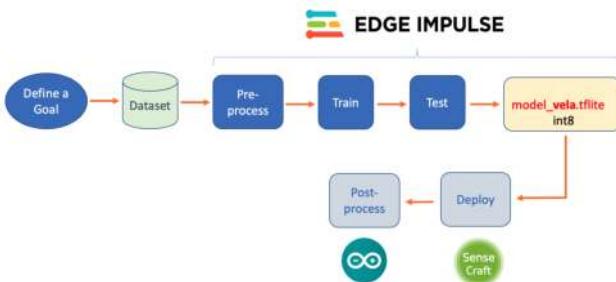
In this Lab, we will explore Image Classification using the Seeed Studio *Grove Vision AI Module V2*, a powerful yet compact device specifically designed for embedded machine learning applications. Based on the

Himax WiseEye2 chip, this module is designed to enable AI capabilities on edge devices, making it an ideal tool for Edge Machine Learning (ML) applications.

Introduction

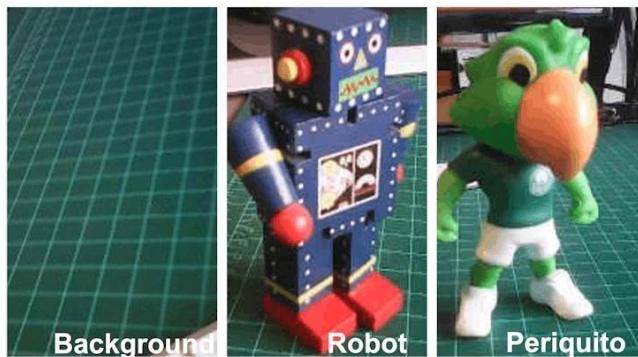
So far, we have explored several computer vision models previously uploaded by Seeed Studio or used the SenseCraft AI Studio for Image Classification, without choosing a specific model. Let's now develop our Image Classification project from scratch, where we will select our data and model.

Below, we can see the project's main steps and where we will work with them:



Project Goal

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

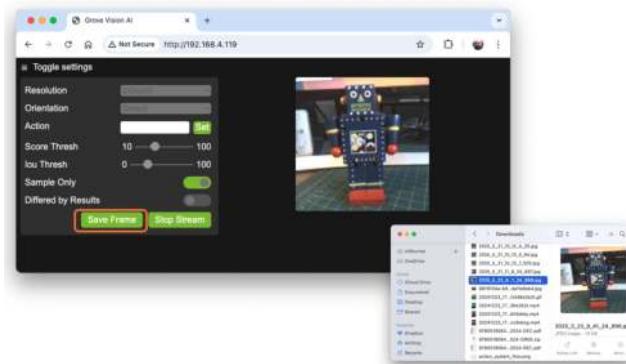


Data Collection

With the Machine Learning project goal defined, dataset collection is the next and most crucial step. Suppose your project utilizes images that are publicly available on datasets, for example, to be used on a **Person Detection** project. In that case, you can download the Wake Vision dataset for use in the project.

But, in our case, we define a project where the images do not exist publicly, so we need to generate them. We can use a phone, computer camera, or other devices to capture the photos, offline or connected to the Edge Impulse Studio.

If you want to use the Grove Vision AI V2 to capture your dataset, you can use the SenseCraft AI Studio as we did in the previous Lab, or the `camera_web_server` sketch as we will describe later in the [Postprocessing / Getting the Video Stream](#) section of this Lab.

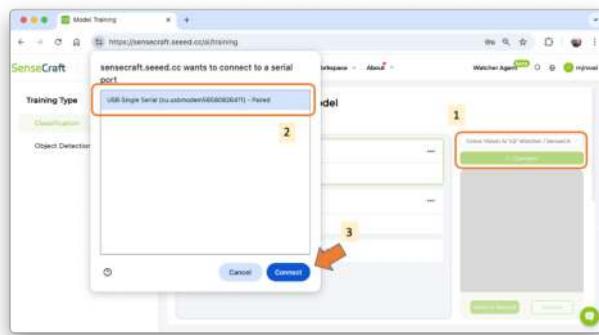


In this Lab, we will use the SenseCraft AI Studio to collect the dataset.

Collecting Data with the SenseCraft AI Studio

On SenseCraft AI Studio: Let's open the tab Training.

The default is to train a Classification model with a WebCam if it is available. Let's select the Grove Vision AI V2 instead. Pressing the green button [Connect] (1), a Pop-Up window will appear. Select the corresponding Port (2) and press the blue button [Connect] (3).

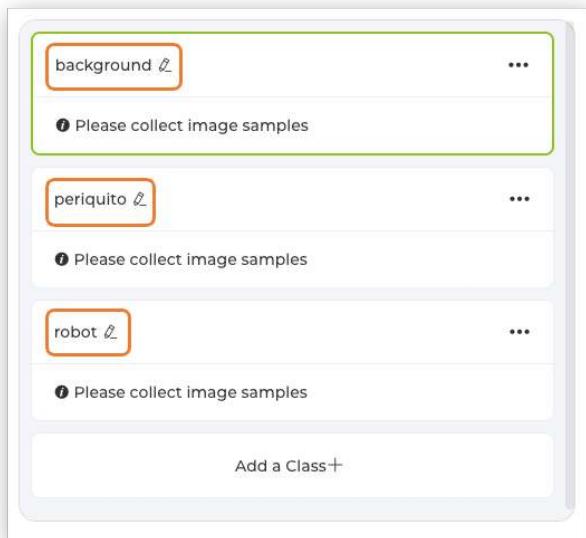


The image streamed from the Grove Vision AI V2 will be displayed.

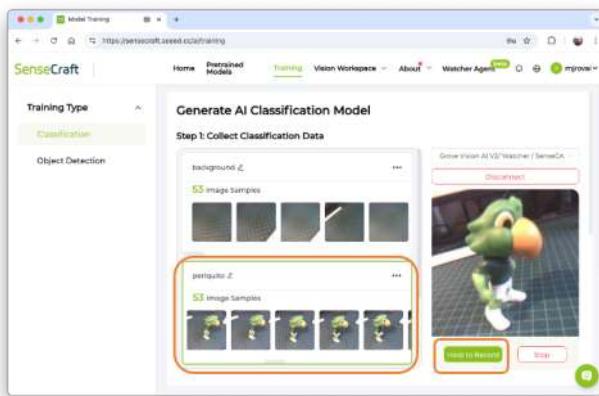
Image Collection

Let's create the classes, following, for example, an alphabetical order:

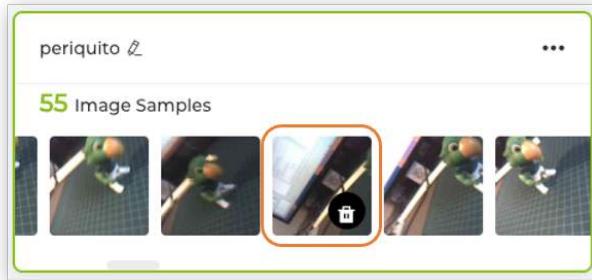
- Class1: background
- Class 2: periquito
- Class 3: robot



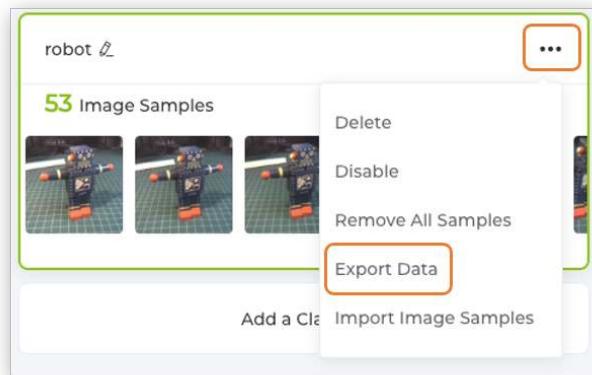
Select one of the classes (note that a green line will be around the window) and keep pressing the green button under the preview area. The collected images will appear on the Image Samples Screen.



After collecting the images, review them and, if necessary, delete any incorrect ones.



Collect around 50 images from each class. After you collect the three classes, open the menu on each of them and select Export Data.

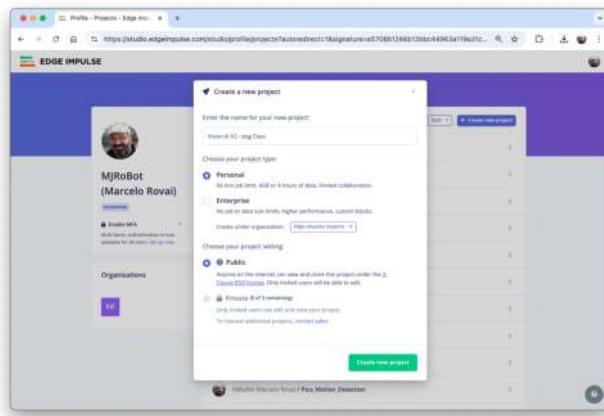


In the Download area of the Computer, we will get three zip files, each one with its corresponding class name. Each Zip file contains a folder with the images.

Uploading the dataset to the Edge Impulse Studio

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

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



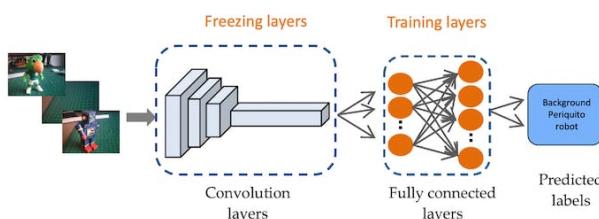
The dataset comprises approximately 50 images per label, with 40 for training and 10 for testing.

Impulse Design and Pre-Processing

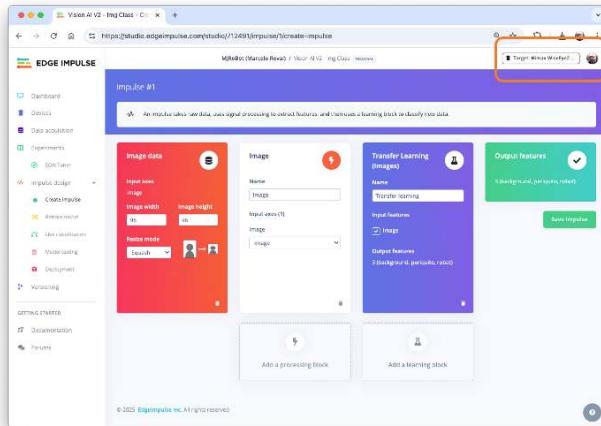
Impulse Design

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

Classifying images is the most common application of deep learning, but a substantial amount of data is required to accomplish this task. We have around 50 images for each category. Is this number enough? Not at all! We will need thousands of images to “teach” or “model” each class, allowing us to differentiate them. However, we can resolve this issue by retraining a previously trained model using thousands of images. We refer to this technique as “Transfer Learning” (TL). With TL, we can fine-tune a pre-trained image classification model on our data, achieving good performance even with relatively small image datasets, as in our case.



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



For comparison, we will keep the image size as 96 x 96. However, keep in mind that with the Grove Vision AI Module V2 and its internal SRAM of 2.4 MB, larger images can be utilized (for example, 160 x 160).

Also select the Target device (Himax WiseEye2 (M55 400 MHz + U55)) on the up-right corner.

Pre-processing (Feature generation)

Besides resizing the images, we can convert them to grayscale or retain their original RGB color depth. Let's select [RGB] in the Image section. Doing that, each data sample will have a dimension of 27,648 features (96x96x3). Pressing [Save Parameters] will open a new tab, Generate Features. Press the button [Generate Features] to generate the features.

Model Design, Training, and Test

In 2007, Google introduced MobileNetV1. In 2018, MobileNetV2: Inverted Residuals and Linear Bottlenecks, was launched, and, in 2019, the V3. The Mobilinet is a family of general-purpose computer vision neural networks explicitly designed for mobile devices to support

classification, detection, and other applications. MobileNets are small, low-latency, low-power models parameterized to meet the resource constraints of various use cases.

Although the base MobileNet architecture is already compact and has low latency, a specific use case or application may often require the model to be even smaller and faster. MobileNets introduce a straightforward parameter, α (alpha), called the width multiplier to construct these smaller, less computationally expensive models. The role of the width multiplier α is to thin a network uniformly at each layer.

Edge Impulse Studio has available MobileNet V1 (96x96 images) and V2 (96x96 and 160x160 images), with several different α values (from 0.05 to 1.0). For example, you will get the highest accuracy with V2, 160x160 images, and $\alpha=1.0$. Of course, there is a trade-off. The higher the accuracy, the more memory (around 1.3M RAM and 2.6M ROM) will be needed to run the model, implying more latency. The smaller footprint will be obtained at another extreme with MobileNet V1 and $\alpha=0.10$ (around 53.2K RAM and 101K ROM).

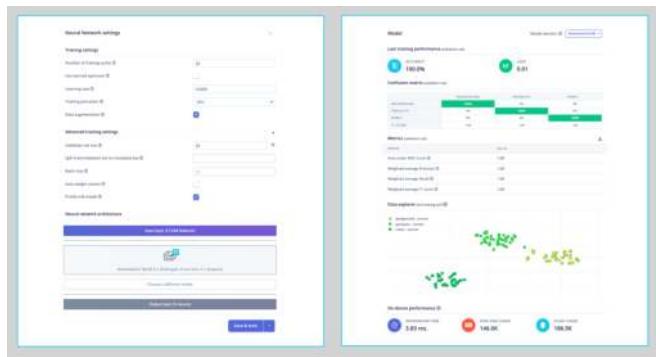
For comparison, we will use the **MobileNet V2 0.1** as our base model (but a model with a greater alpha can be used here). The final layer of our model, preceding the output layer, will have 8 neurons with a 10% dropout rate for preventing overfitting.

Another necessary technique to use with deep learning is **data augmentation**. Data augmentation is a method that can help improve the accuracy of machine learning models 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).

Set the Hyperparameters:

- Epochs: 20,
- Batch Size: 32
- Learning Rate: 0.0005
- Validation size: 20%

Training result:



The model profile predicts **146 KB of RAM** and **187 KB of Flash**, indicating no problem with the Grove AI Vision (V2), which has almost 2.5 MB of internal SRAM. Additionally, the Studio indicates a **latency of around 4 ms**.

Despite this, with a 100% accuracy on the Validation set when using the spare data for testing, we confirmed an Accuracy of 81%, using the Quantized (Int8) trained model. However, it is sufficient for our purposes in this lab.

Model Deployment

On the Deployment tab, we should select: Seeed Grove Vision AI Module V2 (Himax WiseEye2) and press [Build]. A ZIP file will be downloaded to our computer.

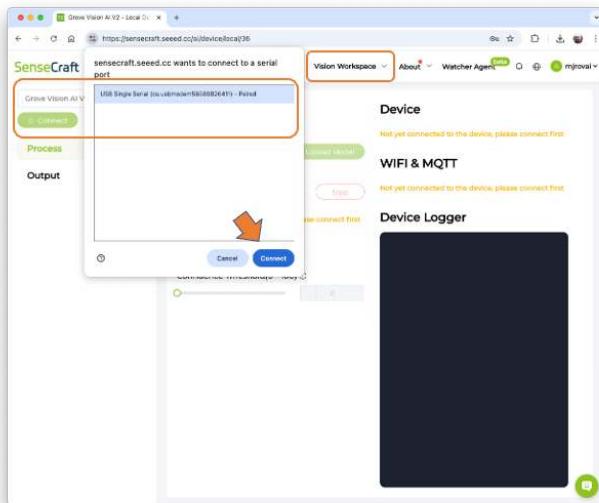
The Zip file contains the `model_vela.tflite`, which is a TensorFlow Lite (TFLite) model optimized for neural processing units (NPUs) using the Vela compiler, a tool developed by Arm to adapt TFLite models for Ethos-U NPUs.

Name	Size	Kind
vision-ai-v2-img-class-seed-grove-vision-ai-module-v2-v1	--	Folder
README.txt	779 bytes	Plain Text Document
model_vela.tflite	193 KB	TensorFlow Lite Model
firmware.img	406 KB	Disk Image
xmodem	--	Folder
xmodem_send.py	9 KB	Python script
xmodem_recv.py	5 KB	Python script
serReadLoop.py	3 KB	Python script
requirements.txt	28 bytes	Plain Text Document
vision-ai-v2-img-class-seed-grove-vision-ai-module-v2-v1.zip	383 bytes	ZIP archive

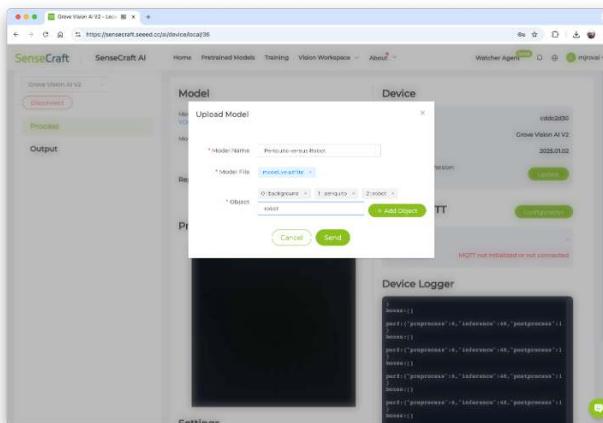
We can flash the model following the instructions in the `README.txt` or use the SenseCraft AI Studio. We will use the latter.

Deploy the model on the SenseCraft AI Studio

On SenseCraft AI Studio, go to the **Vision Workspace** tab, and connect the device:

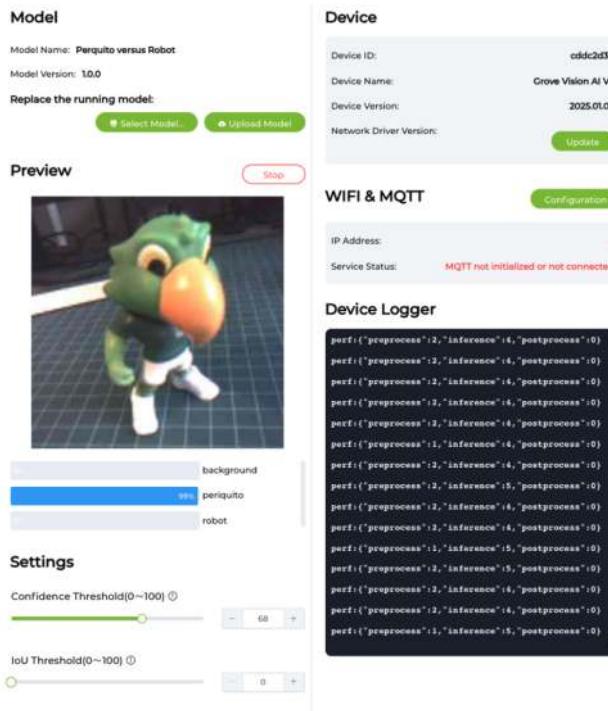


You should see the last model that was uploaded to the device. Select the green button [**Upload Model**]. A pop-up window will ask for the **model name**, the **model file**, and to enter the class names (**objects**). We should use labels following alphabetical order: 0: background, 1: periquito, and 2: robot, and then press [**Send**].

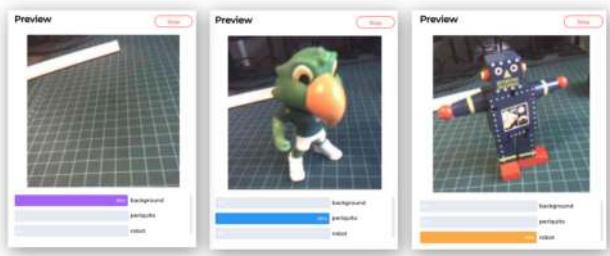


After a few seconds, the model will be uploaded (“flashed”) to our device, and the camera image will appear in real-time on the **Preview** Sector. The Classification result will be displayed under the image preview. It is also possible to select the **Confidence Threshold** of your inference using the cursor on **Settings**.

On the **Device Logger**, we can view the Serial Monitor, where we can observe the latency, which is approximately 1 to 2 ms for pre-processing and 4 to 5 ms for inference, aligning with the estimates made in Edge Impulse Studio.



Here are other screenshots:



The power consumption of this model is approximately 70 mA, equivalent to 0.4 W.

Image Classification (non-official) Benchmark

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

Taking advantage of this opportunity, a similarly trained model, MobileNetV2 96x96, with an alpha of 0.1, was also deployed on the ESP-CAM, the XIAO, and a Raspberry Pi Zero W2. Here is the result:



The Grove Vision AI V2 with an **ARM Ethos-U55** was approximately 14 times faster than devices with an ARM-M7, and more than 100 times faster than an Xtensa LX6 (ESP-CAM). Even when compared to a Raspberry Pi, with a much more powerful CPU, the U55 reduces latency by almost half. Additionally, the power consumption is lower than that of other devices (see the full article here for power consumption comparison).

Postprocessing

Now that we have the model uploaded to the board and working correctly, classifying our images, let's connect a Master Device to export the inference result to it and see the result completely offline (disconnected from the PC and, for example, powered by a battery).

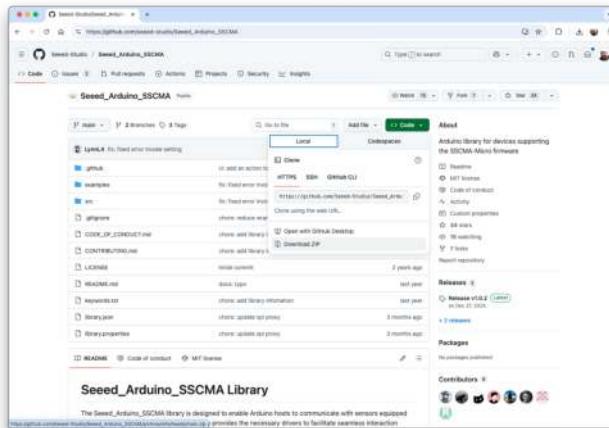
Note that we can use any microcontroller as a Master Controller, such as the XIAO, Arduino, or Raspberry Pi.

Getting the Video Stream

The image processing and model inference are processed locally in Grove Vision AI (V2), and we want the result to be output to the XIAO (Master Controller) via IIC. For that, we will use the **Arduino SSMA library**. This library's primary purpose is to process Grove Vision AI's data stream, which does not involve model inference.

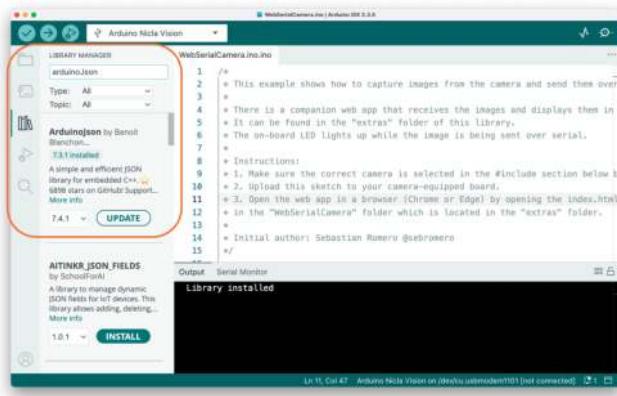
The Grove Vision AI (V2) communicates (Inference result) with the XIAO via the IIC; the device's IIC address is 0x62. Image information transfer is via the USB serial port.

Step 1: Download the Arduino SSMA library as a zip file from its GitHub:

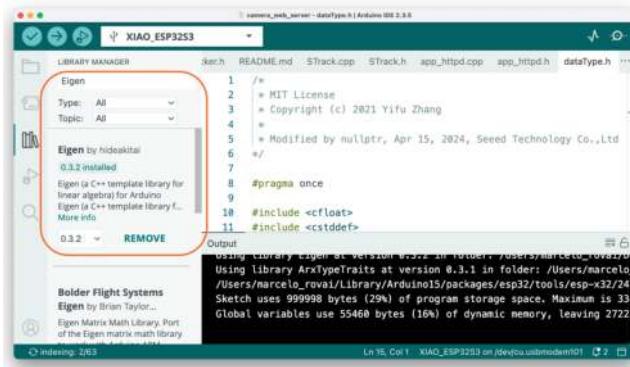


Step 2: Install it in the Arduino IDE (`sketch > Include Library > Add .Zip Library`).

Step 3: Install the **ArduinoJSON** library.



Step 4: Install the Eigen Library



Step 3: Now, connect the XIAO and Grove Vision AI (V2) via the socket (a row of pins) located at the back of the device.



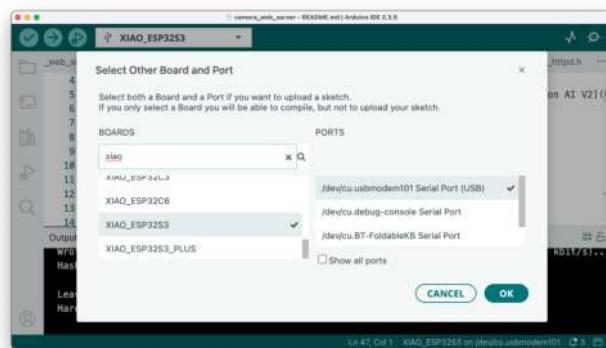
CAUTION: Please note the direction of the connection, Grove Vision AI's Type-C connector should be in the same direction as XIAO's Type-C connector.

Step 5: Connect the **XIAO USB-C** port to your computer

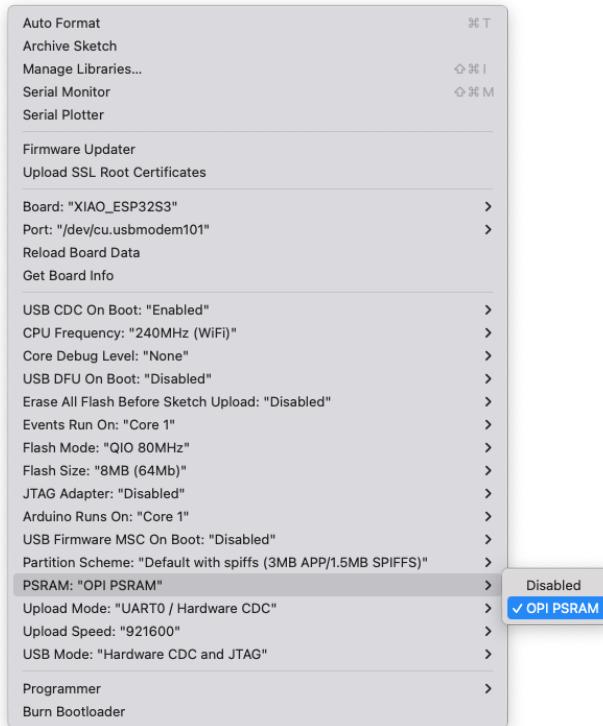


Step 6: In the Arduino IDE, select the Xiao board and the corresponding USB port.

Once we want to stream the video to a webpage, we will use the **XIAO ESP32S3**, which has wifi and enough memory to handle images. Select **XIAO_ESP32S3** and the appropriate USB Port:



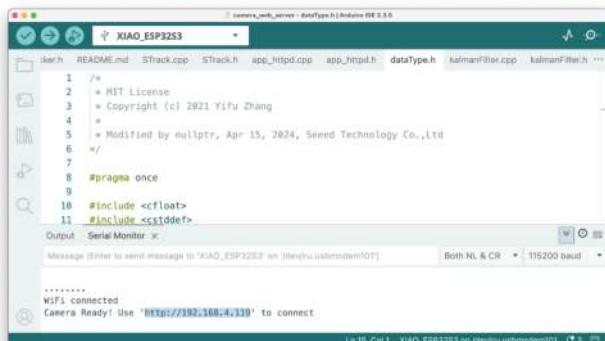
By default, the PSRAM is disabled. Open the Tools menu and on PSRAM: "OPI PSRAM"select OPI PSRAM.

**Step 7:** Open the example in Arduino IDE:

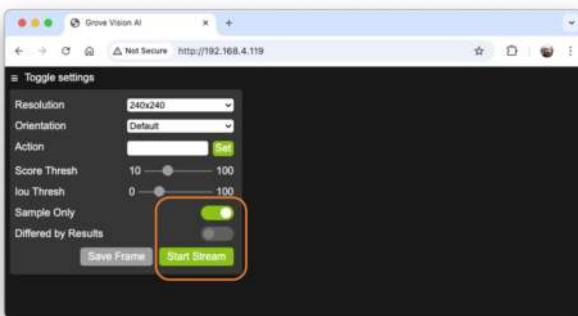
File -> Examples -> Seeed_Arduino_SSCMA -> camera_web_server.

And edit the ssid and password in the camera_web_server.ino sketch to match the Wi-Fi network.

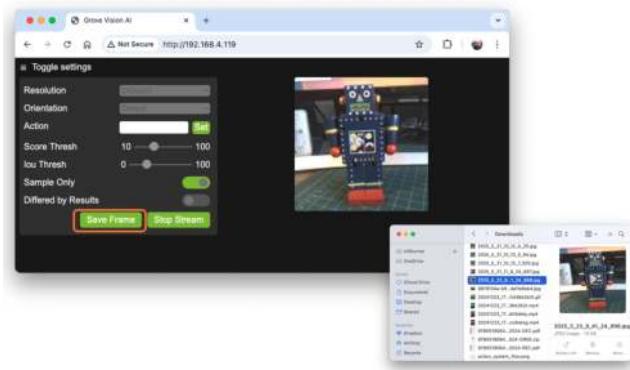
Step 8: Upload the sketch to the board and open the Serial Monitor. When connected to the Wi-Fi network, the board's IP address will be displayed.



Open the address using a web browser. A Video App will be available. To see **only** the video stream from the Grove Vision AI V2, press [Sample Only] and [Start Stream].

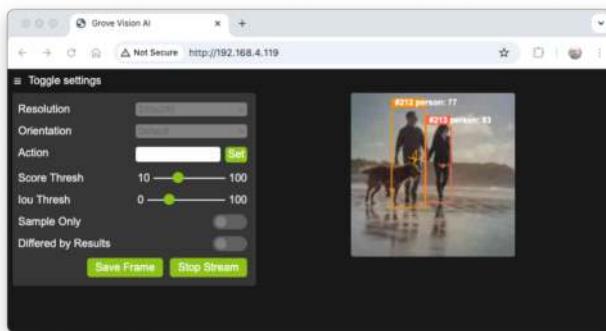


If you want to create an image dataset, you can use this app, saving frames of the video generated by the device. Pressing [Save Frame], the image will be saved in the download area of our desktop.



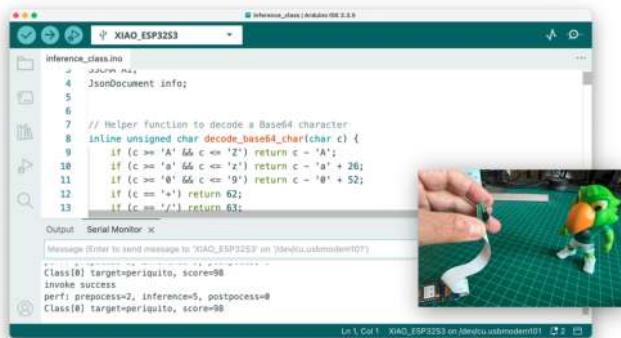
Opening the App **without** selecting [Sample Only], the inference result should appear on the video screen, but this does not happen for Image Classification. For Object Detection or Pose Estimation, the result is embedded with the video stream.

For example, if the model is a Person Detection using YoloV8:



Getting the Inference Result

- Go to File -> Examples -> Seeed_Arduino_SSCMA -> inference_class.
- Upload the sketch to the board, and open the Serial Monitor.
- Pointing the camera at one of our objects, we can see the inference result on the Serial Terminal.



The inference running on the Arduino IDE had an average consumption of 160 mA or 800 mW and a peak of 330 mA 1.65 W when transmitting the image to the App.

Postprocessing with LED

The idea behind our postprocessing is that whenever a specific image is detected (for example, the Periquito - Label:1), the User LED is turned on. If the Robot or a background is detected, the LED will be off.

Copy the below code and past it to your IDE:

```
#include <Seeed_Arduino_SSCMA.h>
SSCMA AI;

void setup()
{
    AI.begin();

    Serial.begin(115200);
    while (!Serial);
    Serial.println("Inferencing - Grove AI V2 / XIAO ESP32S3");

    // Pins for the built-in LED
    pinMode(LED_BUILTIN, OUTPUT);
    // Ensure the LED is OFF by default.
    // Note: The LED is ON when the pin is LOW, OFF when HIGH.
    digitalWrite(LED_BUILTIN, HIGH);
}

void loop()
{
```

```
if (!AI.invoke()){
    Serial.println("\nInvoke Success");
    Serial.print("Latency [ms]: prepocess=");
    Serial.print(AI.perf().prepocess);
    Serial.print(", inference=");
    Serial.print(AI.perf().inference);
    Serial.print(", postpocess=");
    Serial.println(AI.perf().postprocess);
    int pred_index = AI.classes()[0].target;
    Serial.print("Result= Label: ");
    Serial.print(pred_index);
    Serial.print(", score=");
    Serial.println(AI.classes()[0].score);
    turn_on_led(pred_index);
}

/***
 * @brief      turn_off_led function - turn-off the User LED
 */
void turn_off_led(){
    digitalWrite(LED_BUILTIN, HIGH);
}

/***
 * @brief      turn_on_led function used to turn on the User LED
 * @param[in]  pred_index
 *             label 0: [0] ==> ALL OFF
 *             label 1: [1] ==> LED ON
 *             label 2: [2] ==> ALL OFF
 *             label 3: [3] ==> ALL OFF
 */
void turn_on_led(int pred_index) {
    switch (pred_index)
    {
        case 0:
            turn_off_led();
            break;
        case 1:
            turn_off_led();
            digitalWrite(LED_BUILTIN, LOW);
            break;
        case 2:
```

```

        turn_off_led();
        break;
    case 3:
        turn_off_led();
        break;
    }
}

```

This sketch uses the Seeed_Arduino_SSCMA.h library to interface with the Grove Vision AI Module V2. The AI module and the LED are initialized in the `setup()` function, and serial communication is started.

The `loop()` function repeatedly calls the `invoke()` method to perform inference using the built-in algorithms of the Grove Vision AI Module V2. Upon a successful inference, the sketch prints out performance metrics to the serial monitor, including preprocessing, inference, and postprocessing times.

The sketch processes and prints out detailed information about the results of the inference:

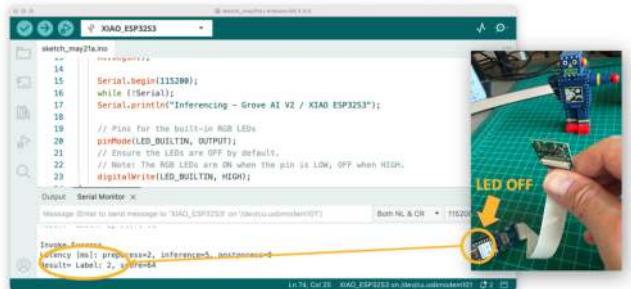
- (`AI.classes()[0]`) that identifies the class of image (`.target`) and its confidence score (`.score`).
- The inference result (class) is stored in the integer variable `pred_index`, which will be used as an input to the function `turn_on_led()`. As a result, the LED will turn ON, depending on the classification result.

Here is the result:

If the Periquito is detected (Label:1), the LED is ON:



If the Robot is detected (Label:2) the LED is OFF (Same for Background (Label:0):



Therefore, we can now power the Grove Vision AI V2 + Xiao ESP32S3 with an external battery, and the inference result will be displayed by the LED completely offline. The consumption is approximately 165 mA or 825 mW.

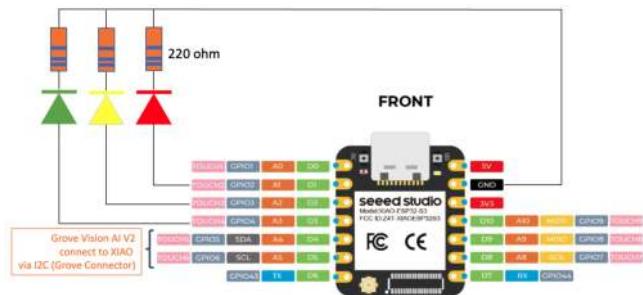
It is also possible to send the result using Wifi, BLE, or other communication protocols available on the used Master Device.

Optional: Post-processing on external devices

Of course, one of the significant advantages of working with EdgeAI is that devices can run entirely disconnected from the cloud, allowing for seamless **interactions with the real world**. We did it in the last section, but using the internal Xiao LED. Now, we will connect external LEDs (which could be any actuator).



The LEDS should be connected to the XIAO ground via a 220-ohm resistor.



The idea is to modify the previous sketch to handle the three external LEDs.

GOAL: Whenever the image of a **Periquito** is detected, the LED **Green** will be ON; if it is a **Robot**, the LED **Yellow** will be ON; if it is a **Background**, the LED **Red** will be ON.

The image processing and model inference are processed locally in Grove Vision AI (V2), and we want the result to be output to the XIAO via IIC. For that, we will use the Arduino SSMA library again.

Here the sketch to be used:

```
#include <Seeed_Arduino_SSCMA.h>
SSCMA AI;

// Define the LED pin according to the pin diagram
// The LEDS negative lead should be connected to the XIAO ground
// via a 220-ohm resistor.
int LEDR = D1; # XIAO ESP32S3 Pin 1
int LEDY = D2; # XIAO ESP32S3 Pin 2
int LEDG = D3; # XIAO ESP32S3 Pin 3

void setup()
{
    AI.begin();

    Serial.begin(115200);
    while (!Serial);
    Serial.println("Inferencing - Grove AI V2 / XIAO ESP32S3");

// Initialize the external LEDs
    pinMode(LEDR, OUTPUT);
    pinMode(LEDY, OUTPUT);
```

```
pinMode(LEDG, OUTPUT);
// Ensure the LEDs are OFF by default.
// Note: The LEDs are ON when the pin is HIGH, OFF when LOW.
digitalWrite(LEDR, LOW);
digitalWrite(LEDY, LOW);
digitalWrite(LEDG, LOW);
}

void loop()
{
    if (!AI.invoke()){
        Serial.println("\nInvoke Success");
        Serial.print("Latency [ms]: prepocess=");
        Serial.print(AI.perf().prepocess);
        Serial.print(", inference=");
        Serial.print(AI.perf().inference);
        Serial.print(", postpocess=");
        Serial.println(AI.perf().postprocess);
        int pred_index = AI.classes()[0].target;
        Serial.print("Result= Label: ");
        Serial.print(pred_index);
        Serial.print(", score=");
        Serial.println(AI.classes()[0].score);
        turn_on_leds(pred_index);
    }
}

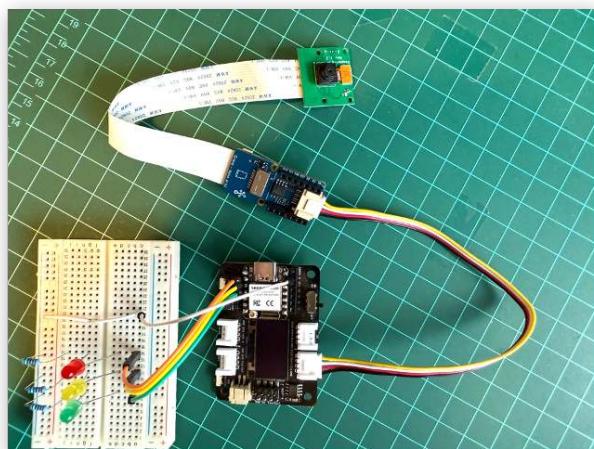
/**
 * @brief turn_off_leds function - turn-off all LEDs
 */
void turn_off_leds(){
    digitalWrite(LEDR, LOW);
    digitalWrite(LEDY, LOW);
    digitalWrite(LEDG, LOW);
}

/**
 * @brief turn_on_leds function used to turn on a specific LED
 * @param[in] pred_index
 *           label 0: [0] ==> Red ON
 *           label 1: [1] ==> Green ON
 *           label 2: [2] ==> Yellow ON
 */

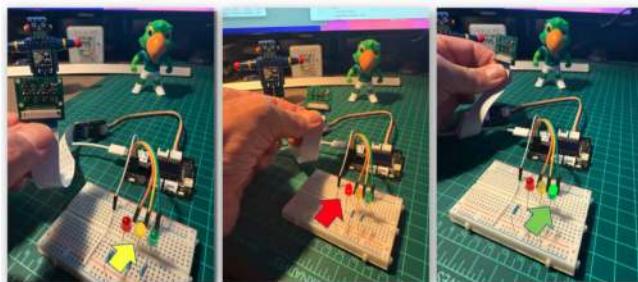
```

```
void turn_on_leds(int pred_index) {
    switch (pred_index)
    {
        case 0:
            turn_off_leds();
            digitalWrite(LED_R, HIGH);
            break;
        case 1:
            turn_off_leds();
            digitalWrite(LED_G, HIGH);
            break;
        case 2:
            turn_off_leds();
            digitalWrite(LED_Y, HIGH);
            break;
        case 3:
            turn_off_leds();
            break;
    }
}
```

We should connect the Grove Vision AI V2 with the XIAO using its I2C Grove connector. For the XIAO, we will use an Expansion Board for the facility (although it is possible to connect the I2C directly to the XIAO's pins). We will power the boards using the USB-C connector, but a battery can also be used.



Here is the result:



The power consumption reached a peak of 240 mA (Green LED), equivalent to 1.2 W. Driving the Yellow and Red LEDs consumes 14 mA, equivalent to 0.7 W. Sending information to the terminal via serial has no impact on power consumption.

Summary

In this lab, we've explored the complete process of developing an image classification system using the Seeed Studio Grove Vision AI Module V2 powered by the Himax WiseEye2 chip. We've walked through every stage of the machine learning workflow, from defining our project goals to deploying a working model with real-world interactions.

The Grove Vision AI V2 has demonstrated impressive performance, with inference times of just 4-5ms, dramatically outperforming other common tinyML platforms. Our benchmark comparison showed it to be approximately 14 times faster than ARM-M7 devices and over 100 times faster than an Xtensa LX6 (ESP-CAM). Even when compared to a Raspberry Pi Zero W2, the Edge TPU architecture delivered nearly twice the speed while consuming less power.

Through this project, we've seen how transfer learning enables us to achieve good classification results with a relatively small dataset of custom images. The MobileNetV2 model with an alpha of 0.1 provided an excellent balance of accuracy and efficiency for our three-class problem, requiring only 146 KB of RAM and 187 KB of Flash memory, well within the capabilities of the Grove Vision AI Module V2's 2.4 MB internal SRAM.

We also explored several deployment options, from viewing inference results through the SenseCraft AI Studio to creating a standalone system

with visual feedback using LEDs. The ability to stream video to a web browser and process inference results locally demonstrates the versatility of edge AI systems for real-world applications.

The power consumption of our final system remained impressively low, ranging from approximately 70mA (0.4W) for basic inference to 240mA (1.2W) when driving external components. This efficiency makes the Grove Vision AI Module V2 an excellent choice for battery-powered applications where power consumption is critical.

This lab has demonstrated that sophisticated computer vision tasks can now be performed entirely at the edge, without reliance on cloud services or powerful computers. With tools like Edge Impulse Studio and SenseCraft AI Studio, the development process has become accessible even to those without extensive machine learning expertise.

As edge AI technology continues to evolve, we can expect even more powerful capabilities from compact, energy-efficient devices like the Grove Vision AI Module V2, opening up new possibilities for smart sensors, IoT applications, and embedded intelligence in everyday objects.

Resources

[Collecting Images with SenseCraft AI Studio.](#)

[Edge Impulse Studio Project](#)

[SenseCraft AI Studio - Vision Workplace \(Deploy Models\)](#)

[Other Himax examples](#)

[Arduino Sketches](#)

Object Detection

This Lab is under Development

VII

KEY:RASPBERRY

Part VII

VIII

RASPBERRY PI

Part VIII

Overview

These labs offer invaluable hands-on experience with machine learning systems, leveraging the versatility and accessibility of the Raspberry Pi platform. Unlike working with large-scale models that demand extensive cloud resources, these exercises allow you to directly interact with hardware and software in a compact yet powerful edge computing environment. You'll gain practical insights into deploying AI at the edge by utilizing Raspberry Pi's capabilities, from the efficient Pi Zero to the more robust Pi 4 or Pi 5 models. This approach provides a tangible understanding of the challenges and opportunities in implementing machine learning solutions in resource-constrained settings. While we're working at a smaller scale, the principles and techniques you'll learn are fundamentally similar to those used in larger systems. The Raspberry Pi's ability to run a whole operating system and its extensive GPIO capabilities allow for a rich learning experience that bridges the gap between theoretical knowledge and real-world application. Through these labs, you'll grasp the intricacies of EdgeML and develop skills applicable to a wide range of AI deployment scenarios.



Figure 1.19: Raspberry Pi Zero 2-W and Raspberry Pi 5 with Camera

Where to Buy

Raspberry Pi boards are available from authorized resellers worldwide:

- Raspberry Pi Products (Official site with reseller locator)
- Raspberry Pi Zero 2 W: ~\$15
- Raspberry Pi 4 (4GB): ~\$55
- Raspberry Pi 5 (4GB): ~\$60
- Raspberry Pi 5 (8GB): ~\$80

Camera modules, power adapters, and SD cards are available from the same resellers.

Pre-requisites

- **Raspberry Pi:** Ensure you have at least one of the boards: the Raspberry Pi Zero 2 W, Raspberry Pi 4 or 5 for the Vision Labs, and the Raspberry 5 for the GenAi labs.
- **Power Adapter:** To Power on the boards.
 - Raspberry Pi Zero 2-W: 2.5 W with a Micro-USB adapter
 - Raspberry Pi 4 or 5: 3.5 W with a USB-C adapter
- **Network:** With internet access for downloading the necessary software and controlling the boards remotely.
- **SD Card (32 GB minimum) and an SD card Adapter:** For the Raspberry Pi OS.

Setup

- Setup Raspberry Pi

Exercises

Modality	Task	Description	Link
Vision	Image Classification	Learn to classify images	Link

Modality	Task	Description	Link
Vision	Object Detection	Implement object detection	Link
GenAI	Small Language Models	Deploy SLMs at the Edge	Link
GenAI	Visual-Language Models	Deploy VLMs at the Edge	Link

Setup

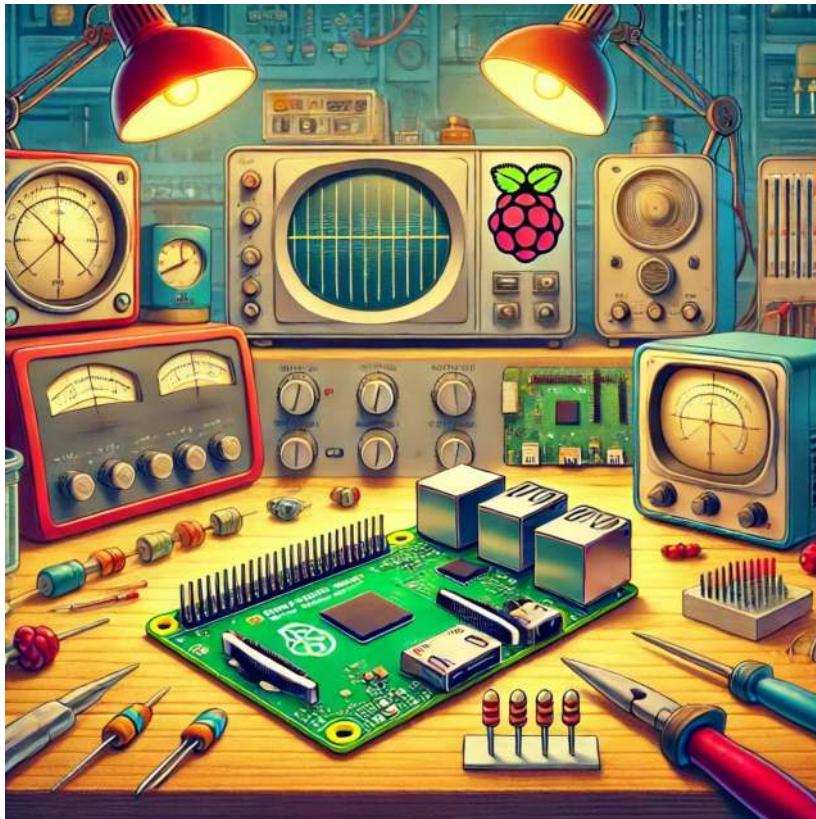


Figure 1.20: DALL-E prompt - An electronics laboratory environment inspired by the 1950s, with a cartoon style. The lab should have vintage equipment, large oscilloscopes, old-fashioned tube radios, and large, boxy computers. The Raspberry Pi 5 board is prominently displayed, accurately shown in its real size, similar to a credit card, on a workbench. The Pi board is surrounded by classic lab tools like a soldering iron, resistors, and wires. The overall scene should be vibrant, with exaggerated colors and playful details characteristic of a cartoon. No logos or text should be included.

This chapter will guide you through setting up Raspberry Pi Zero 2 W (*Raspi-Zero*) and Raspberry Pi 5 (*Raspi-5*) models. We'll cover hardware setup, operating system installation, initial configuration, and tests.

The general instructions for the *Raspi-5* also apply to the older Raspberry Pi versions, such as the Raspi-3 and Raspi-4.

Overview

The Raspberry Pi is a powerful and versatile single-board computer that has become an essential tool for engineers across various disciplines. Developed by the Raspberry Pi Foundation, these compact devices offer a unique combination of affordability, computational power, and extensive GPIO (General Purpose Input/Output) capabilities, making them ideal for prototyping, embedded systems development, and advanced engineering projects.

Key Features

1. **Computational Power:** Despite their small size, Raspberry Pis offer significant processing capabilities, with the latest models featuring multi-core ARM processors and up to 8 GB of RAM.
2. **GPIO Interface:** The 40-pin GPIO header allows direct interaction with sensors, actuators, and other electronic components, facilitating hardware-software integration projects.
3. **Extensive Connectivity:** Built-in Wi-Fi, Bluetooth, Ethernet, and multiple USB ports enable diverse communication and networking projects.
4. **Low-Level Hardware Access:** Raspberry Pis provides access to interfaces like I2C, SPI, and UART, allowing for detailed control and communication with external devices.
5. **Real-Time Capabilities:** With proper configuration, Raspberry Pis can be used for soft real-time applications, making them suitable for control systems and signal processing tasks.
6. **Power Efficiency:** Low power consumption enables battery-powered and energy-efficient designs, especially in models like the Pi Zero.

Raspberry Pi Models (covered in this book)

1. Raspberry Pi Zero 2 W (*Raspi-Zero*):

- Ideal for: Compact embedded systems
- Key specs: 1 GHz single-core CPU (ARM Cortex-A53), 512 MB RAM, minimal power consumption

2. Raspberry Pi 5 (*Raspi-5*):

- Ideal for: More demanding applications such as edge computing, computer vision, and edgeAI applications, including LLMs.
- Key specs: 2.4 GHz quad-core CPU (ARM Cortex A-76), up to 8 GB RAM, PCIe interface for expansions

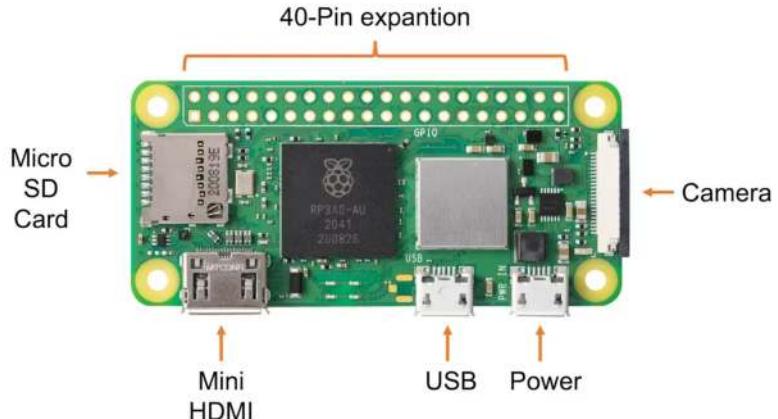
Engineering Applications

1. **Embedded Systems Design:** Develop and prototype embedded systems for real-world applications.
2. **IoT and Networked Devices:** Create interconnected devices and explore protocols like MQTT, CoAP, and HTTP/HTTPS.
3. **Control Systems:** Implement feedback control loops, PID controllers, and interface with actuators.
4. **Computer Vision and AI:** Utilize libraries like OpenCV and TensorFlow Lite for image processing and machine learning at the edge.
5. **Data Acquisition and Analysis:** Collect sensor data, perform real-time analysis, and create data logging systems.
6. **Robotics:** Build robot controllers, implement motion planning algorithms, and interface with motor drivers.
7. **Signal Processing:** Perform real-time signal analysis, filtering, and DSP applications.
8. **Network Security:** Set up VPNs, firewalls, and explore network penetration testing.

This tutorial will guide you through setting up the most common Raspberry Pi models, enabling you to start on your machine learning project quickly. We'll cover hardware setup, operating system installation, and initial configuration, focusing on preparing your Pi for Machine Learning applications.

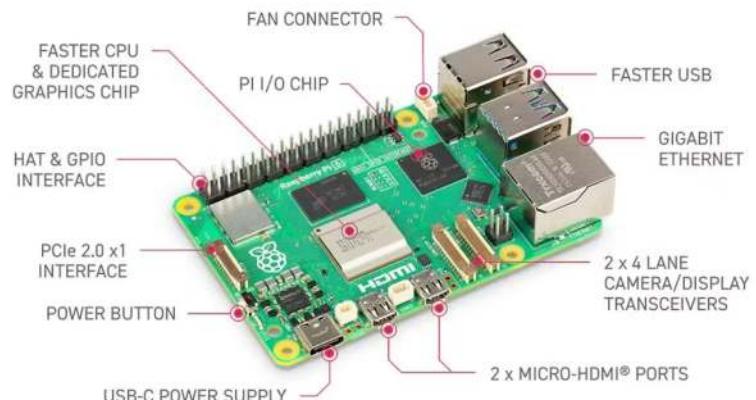
Hardware Overview

Raspberry Pi Zero 2W



- **Processor:** 1 GHz quad-core 64-bit Arm Cortex-A53 CPU
- **RAM:** 512 MB SDRAM
- **Wireless:** 2.4 GHz 802.11 b/g/n wireless LAN, Bluetooth 4.2, BLE
- **Ports:** Mini HDMI, micro USB OTG, CSI-2 camera connector
- **Power:** 5 V via micro USB port

Raspberry Pi 5



- **Processor:**
 - Pi 5: Quad-core 64-bit Arm Cortex-A76 CPU @ 2.4 GHz
 - Pi 4: Quad-core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5 GHz
- **RAM:** 2 GB, 4 GB, or 8 GB options (8 GB recommended for AI tasks)
- **Wireless:** Dual-band 802.11ac wireless, Bluetooth 5.0
- **Ports:** 2 × micro HDMI ports, 2 × USB 3.0 ports, 2 × USB 2.0 ports, CSI camera port, DSI display port
- **Power:** 5 V DC via USB-C connector (3A)

In the labs, we will use different names to address the Raspberry: Raspi, Raspi-5, Raspi-Zero, etc. Usually, Raspi is used when the instructions or comments apply to every model.

Installing the Operating System

The Operating System (OS)

An operating system (OS) is fundamental software that manages computer hardware and software resources, providing standard services for computer programs. It is the core software that runs on a computer, acting as an intermediary between hardware and application software. The OS manages the computer's memory, processes, device drivers, files, and security protocols.

1. Key functions:

- Process management: Allocating CPU time to different programs
- Memory management: Allocating and freeing up memory as needed
- File system management: Organizing and keeping track of files and directories
- Device management: Communicating with connected hardware devices
- User interface: Providing a way for users to interact with the computer

2. Components:

- Kernel: The core of the OS that manages hardware resources

- Shell: The user interface for interacting with the OS
- File system: Organizes and manages data storage
- Device drivers: Software that allows the OS to communicate with hardware

The Raspberry Pi runs a specialized version of Linux designed for embedded systems. This operating system, typically a variant of Debian called Raspberry Pi OS (formerly Raspbian), is optimized for the Pi's ARM-based architecture and limited resources.

The latest version of Raspberry Pi OS is based on Debian Bookworm.

Key features:

1. Lightweight: Tailored to run efficiently on the Pi's hardware.
2. Versatile: Supports a wide range of applications and programming languages.
3. Open-Source: Allows for customization and community-driven improvements.
4. GPIO support: Enables interaction with sensors and other hardware through the Pi's pins.
5. Regular updates: Continuously improved for performance and security.

Embedded Linux on the Raspberry Pi provides a full-featured operating system in a compact package, making it ideal for projects ranging from simple IoT devices to more complex edge machine-learning applications. Its compatibility with standard Linux tools and libraries makes it a powerful platform for development and experimentation.

Installation

To use the Raspberry Pi, we will need an operating system. By default, Raspberry Pi checks for an operating system on any SD card inserted in the slot, so we should install an operating system using Raspberry Pi Imager.

Raspberry Pi Imager is a tool for downloading and writing images on *macOS*, *Windows*, and *Linux*. It includes many popular operating system images for Raspberry Pi. We will also use the Imager to preconfigure credentials and remote access settings.

Follow the steps to install the OS in your Raspi.

1. Download and install the Raspberry Pi Imager on your computer.
2. Insert a microSD card into your computer (a 32GB SD card is recommended).
3. Open Raspberry Pi Imager and select your Raspberry Pi model.
4. Choose the appropriate operating system:
 - **For Raspi-Zero:** For example, you can select: Raspberry Pi OS Lite (64-bit).

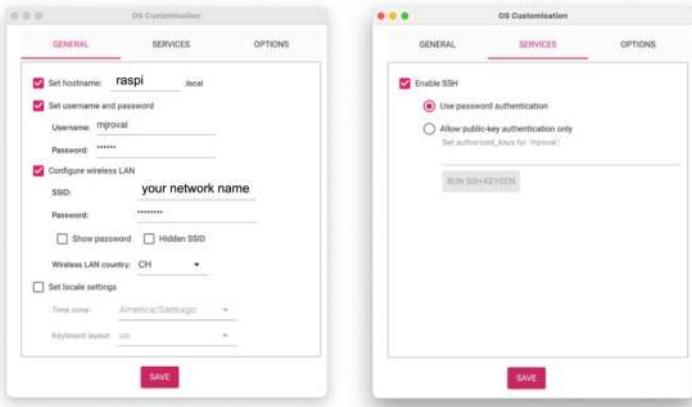


Due to its reduced SDRAM (512 MB), the recommended OS for the Raspi-Zero is the 32-bit version. However, to run some machine learning models, such as the YOLOv8 from Ultralytics, we should use the 64-bit version. Although Raspi-Zero can run a *desktop*, we will choose the LITE version (no Desktop) to reduce the RAM needed for regular operation.

- **For Raspi-5:** We can select the full 64-bit version, which includes a desktop: Raspberry Pi OS (64-bit)



5. Select your microSD card as the storage device.
6. Click on Next and then the gear icon to access advanced options.
7. Set the *hostname*, the Raspi *username and password*, configure WiFi and *enable SSH* (Very important!)



8. Write the image to the microSD card.

In the examples here, we will use different hostnames depending on the device used: raspi, raspi-5, raspi-Zero, etc. It would help if you replaced it with the one you are using.

Initial Configuration

1. Insert the microSD card into your Raspberry Pi.

2. Connect power to boot up the Raspberry Pi.
3. Please wait for the initial boot process to complete (it may take a few minutes).

You can find the most common Linux commands to be used with the Raspi here or here.

Remote Access

SSH Access

The easiest way to interact with the Raspi-Zero is via SSH ("Headless"). You can use a Terminal (MAC/Linux), PuTTy (Windows), or any other.

1. Find your Raspberry Pi's IP address (for example, check your router).
2. On your computer, open a terminal and connect via SSH:

```
ssh username@[raspberry_pi_ip_address]
```

Alternatively, if you do not have the IP address, you can try the following: bash ssh username@hostname.local for example, ssh mjrovai@rpi-5.local , ssh mjrovai@raspi.local , etc.

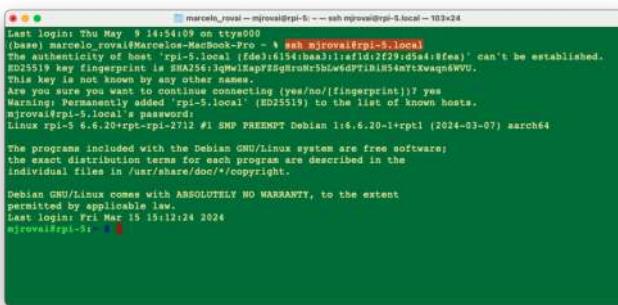


Figure 1.21: img

When you see the prompt:

```
mjrovai@rpi-5: ~ $
```

It means that you are interacting remotely with your Raspi. It is a good practice to update/upgrade the system regularly. For that, you should run:

```
sudo apt-get update  
sudo apt upgrade
```

You should confirm the Raspi IP address. On the terminal, you can use:

```
hostname -I
```



```
marcelo_rovai — mjrovai@rpi-5: ~ — ssh mjrovai@rpi-5.local — 57x5  
mjrovai@rpi-5:~$ hostname -I  
192.168.4.209 fde3:6154:baa3:1:afld:2f29:d5a4:8fea  
mjrovai@rpi-5:~$
```

To shut down the Raspi via terminal:

When you want to turn off your Raspberry Pi, there are better ideas than just pulling the power cord. This is because the Raspi may still be writing data to the SD card, in which case merely powering down may result in data loss or, even worse, a corrupted SD card.

For safety shut down, use the command line:

```
sudo shutdown -h now
```

To avoid possible data loss and SD card corruption, before removing the power, you should wait a few seconds after shutdown for the Raspberry Pi's LED to stop blinking and go dark. Once the LED goes out, it's safe to power down.

Transfer Files between the Raspi and a computer

Transferring files between the Raspi and our main computer can be done using a pen drive, directly on the terminal (with scp), or an FTP program over the network.

Using Secure Copy Protocol (scp):

Copy files to your Raspberry Pi. Let's create a text file on our computer, for example, `test.txt`.



You can use any text editor. In the same terminal, an option is the nano.

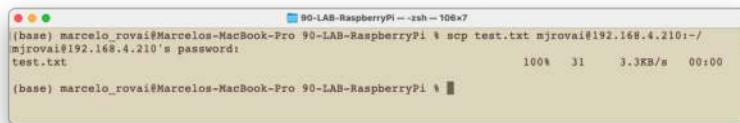
To copy the file named `test.txt` from your personal computer to a user's home folder on your Raspberry Pi, run the following command from the directory containing `test.txt`, replacing the `<username>` placeholder with the username you use to log in to your Raspberry Pi and the `<pi_ip_address>` placeholder with your Raspberry Pi's IP address:

```
$ scp test.txt <username>@<pi_ip_address>:~/
```

Note that `~/` means that we will move the file to the ROOT of our Raspi. You can choose any folder in your Raspi. But you should create the folder before you run `scp`, since `scp` won't create folders automatically.

For example, let's transfer the file `test.txt` to the ROOT of my Raspi-zero, which has an IP of 192.168.4.210:

```
scp test.txt mjrovai@192.168.4.210:~/
```



I use a different profile to differentiate the terminals. The above action happens **on your computer**. Now, let's go to our Raspi (using the SSH) and check if the file is there:



Copy files from your Raspberry Pi. To copy a file named `test.txt` from a user's home directory on a Raspberry Pi to the current directory on another computer, run the following command **on your Host Computer**:

```
$ scp <username>@<pi_ip_address>:myfile.txt .
```

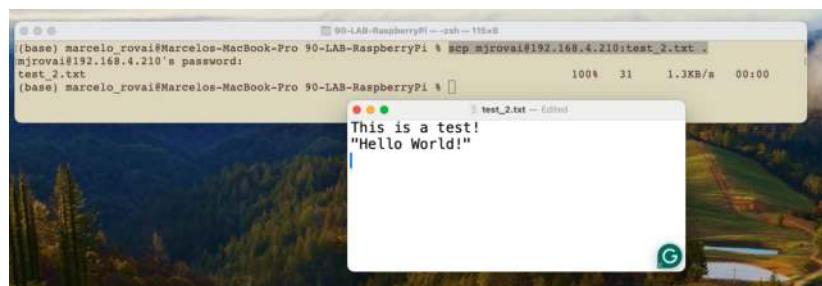
For example:

On the Raspi, let's create a copy of the file with another name:

```
cp test.txt test_2.txt
```

And on the Host Computer (in my case, a Mac)

```
scp mjrovai@192.168.4.210:test_2.txt .
```

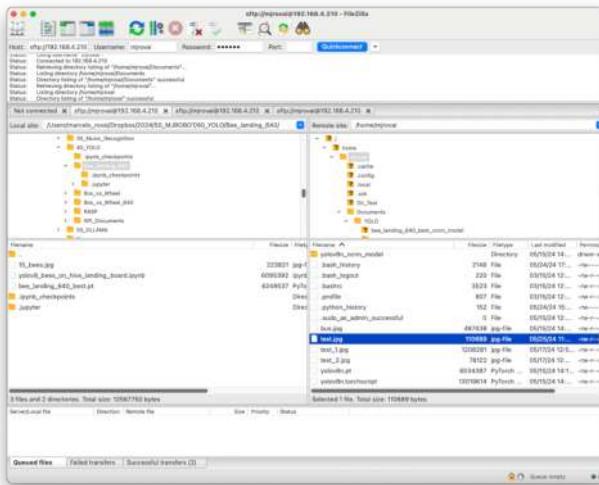


Transferring files using FTP

Transferring files using FTP, such as FileZilla FTP Client, is also possible. Follow the instructions, install the program for your Desktop OS, and use the Raspi IP address as the Host. For example:

`sftp://192.168.4.210`

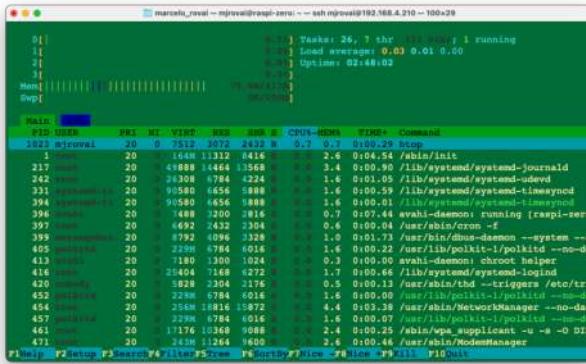
and enter your Raspi username and password. Pressing Quickconnect will open two windows, one for your host computer desktop (right) and another for the Raspi (left).



Increasing SWAP Memory

Using `htop`, a cross-platform interactive process viewer, you can easily monitor the resources running on your Raspi, such as the list of processes, the running CPUs, and the memory used in real-time. To launch `htop`, enter with the command on the terminal:

`htop`



Regarding memory, among the devices in the Raspberry Pi family, the Raspi-Zero has the smallest amount of SRAM (500 MB), compared to a selection of 2 GB to 8 GB on the Raspis 4 or 5. For any Raspi, it is possible to increase the memory available to the system with “Swap.” Swap memory, also known as swap space, is a technique used in computer operating systems to temporarily store data from RAM (Random Access Memory) on the SD card when the physical RAM is fully utilized. This allows the operating system (OS) to continue running even when RAM is full, which can prevent system crashes or slowdowns.

Swap memory benefits devices with limited RAM, such as the Raspberry Pi Zero. Increasing swap can help run more demanding applications or processes, but it's essential to balance this with the potential performance impact of frequent disk access.

By default, the Rapi-Zero's SWAP (Swp) memory is only 100 MB, which is very small for running some more complex and demanding Machine Learning applications (for example, YOLO). Let's increase it to 2 MB:

First, turn off swap-file:

```
sudo dphys-swapfile swapoff
```

Next, you should open and change the file `/etc/dphys-swapfile`. For that, we will use the nano:

```
sudo nano /etc/dphys-swapfile
```

Search for the **CONF_SWAPSIZE** variable (default is 200) and update it to **2000**:

CONF SWAPSIZE=2000

And save the file.

Next, turn on the swapfile again and reboot the Raspi-zero:

```
sudo dphys-swapfile setup  
sudo dphys-swapfile swapon  
sudo reboot
```

When your device is rebooted (you should enter with the SSH again), you will realize that the maximum swap memory value shown on top is now something near 2 GB (in my case, 1.95 GB).

To keep the *htop* running, you should open another terminal window to interact continuously with your Raspi.

Installing a Camera

The Raspi is an excellent device for computer vision applications; a camera is needed for it. We can install a standard USB webcam on the micro-USB port using a USB OTG adapter (Raspi-Zero and Raspi-5) or a camera module connected to the Raspi CSI (Camera Serial Interface) port.

USB Webcams generally have inferior quality to the camera modules that connect to the CSI port. They can also not be controlled using the `raspistill` and `raspivid` commands in the terminal or the `picamera` recording package in Python. Nevertheless, there may be reasons why you want to connect a USB camera to your Raspberry Pi, such as because of the benefit that it is much easier to set up multiple cameras with a single Raspberry Pi, long cables, or simply because you have such a camera on hand.

Installing a USB WebCam

1. Power off the Raspi:

```
sudo shutdown -h no
```

2. Connect the USB Webcam (USB Camera Module 30 fps, 1280 × 720) to your Raspi (In this example, I am using the Raspi-Zero, but the instructions work for all Raspis).



3. Power on again and run the SSH
4. To check if your USB camera is recognized, run:

```
lsusb
```

You should see your camera listed in the output.

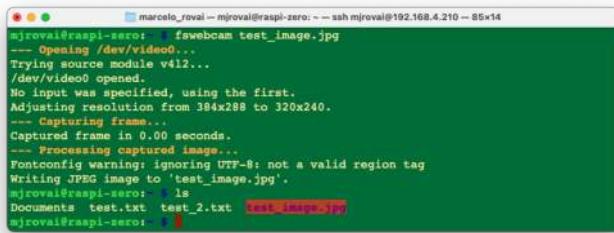
```
marcelo_royal@mjroval@raspi-zero: ~ ssh mjroval@192.168.4.210 -- 66x
mjroval@raspi-zero: ~ lsusb
Bus 001 Device 003: ID 0c45:1915 Microdia USB 2.0 Camera
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
mjroval@raspi-zero: ~
mjroval@raspi-zero: ~
```

A screenshot of a terminal window titled "Terminal". The window shows the command "lsusb" being run and its output. The output lists two devices: a Microdia USB 2.0 Camera and a Linux Foundation 2.0 root hub. The terminal window has a dark background with white text and a red border.

5. To take a test picture with your USB camera, use:

```
fswebcam test_image.jpg
```

This will save an image named “test_image.jpg” in your current directory.



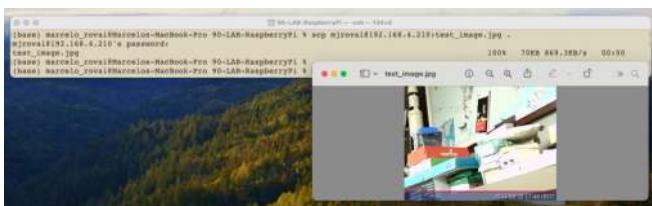
```
mjrovai@raspi-zero: ~ $ fswebcam test_image.jpg
--- Opening /dev/video0...
Trying source module v4l2...
/dev/video0 opened.
No input was specified, using the first.
Adjusting resolution from 384x288 to 320x240.
--- Capturing frame...
Captured frame in 0.00 seconds.
--- Processing captured image...
Fontconfig warning: ignoring UTF-8: not a valid region tag
Writing JPEG image to 'test_image.jpg'.
mjrovai@raspi-zero: ~ $ ls
Documents test.txt test_2.txt test_image.jpg
mjrovai@raspi-zero: ~ $
```

6. Since we are using SSH to connect to our Rapsi, we must transfer the image to our main computer so we can view it. We can use FileZilla or SCP for this:

Open a terminal **on your host computer** and run:

```
scp mjrovai@raspi-zero.local:~/test_image.jpg .
```

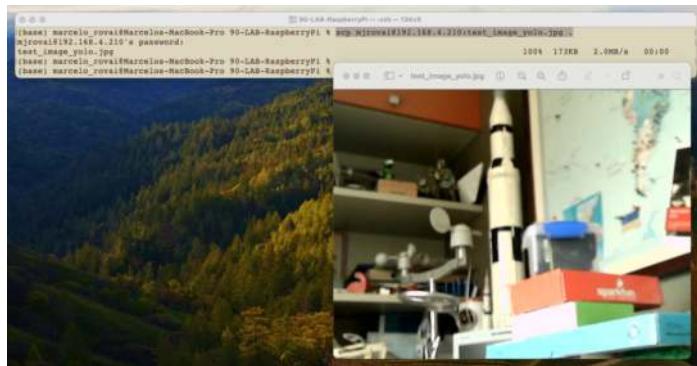
Replace “mjrovai” with your username and “raspi-zero” with Pi’s hostname.



7. If the image quality isn’t satisfactory, you can adjust various settings; for example, define a resolution that is suitable for YOLO (640x640):

```
fswebcam -r 640x640 --no-banner test_image_yolo.jpg
```

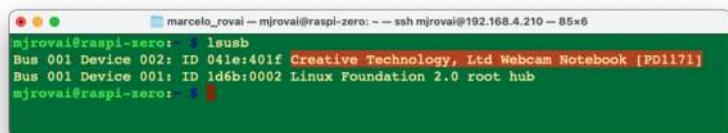
This captures a higher-resolution image without the default banner.



An ordinary USB Webcam can also be used:



And verified using lsusb



Video Streaming

For stream video (which is more resource-intensive), we can install and use mjpg-streamer:

First, install Git:

```
sudo apt install git
```

Now, we should install the necessary dependencies for mjpg-streamer, clone the repository, and proceed with the installation:

```
sudo apt install cmake libjpeg62-turbo-dev
git clone https://github.com/jacksonliam/mjpg-streamer.git
cd mjpg-streamer/mjpg-streamer-experimental
make
sudo make install
```

Then start the stream with:

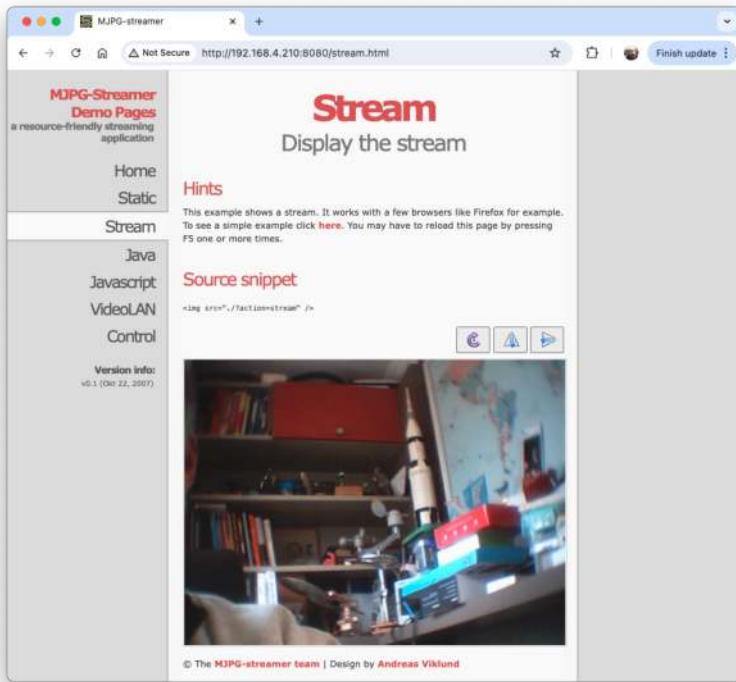
```
mjpg_streamer -i "input_uvc.so" -o "output_http.so -w ./www"
```

We can then access the stream by opening a web browser and navigating to:

`http://<your_pi_ip_address>:8080`. In my case: `http://192.168.4.210:8080`

We should see a webpage with options to view the stream. Click on the link that says “Stream” or try accessing:

```
http://<raspberry_pi_ip_address>:8080/?action=stream
```



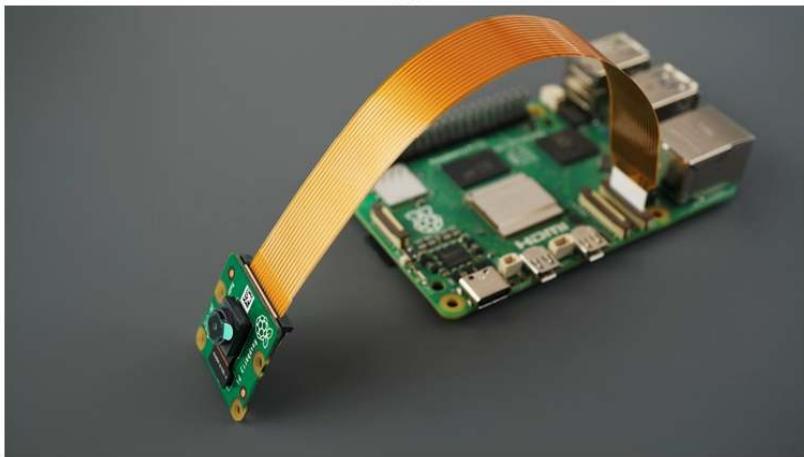
Installing a Camera Module on the CSI port

There are now several Raspberry Pi camera modules. The original 5-megapixel model was released in 2013, followed by an 8-megapixel Camera Module 2 that was later released in 2016. The latest camera model is the 12-megapixel Camera Module 3, released in 2023.

The original 5 MP camera (**Arducam OV5647**) is no longer available from Raspberry Pi but can be found from several alternative suppliers. Below is an example of such a camera on a Raspi-Zero.



Here is another example of a v2 Camera Module, which has a **Sony IMX219** 8-megapixel sensor:



Any camera module will work on the Raspberry Pis, but for that, the `configuration.txt` file must be updated:

```
sudo nano /boot/firmware/config.txt
```

At the bottom of the file, for example, to use the 5 MP Arducam OV5647 camera, add the line:

```
dtoverlay=ov5647,cam0
```

Or for the v2 module, which has the 8MP Sony IMX219 camera:

```
dtoverlay=imx219,cam0
```

Save the file (CTRL+O [ENTER] CRTL+X) and reboot the Raspi:

Sudo reboot

After the boot, you can see if the camera is listed:

```
libcamera-hello --list-cameras
```

```
mjrovai@rpi-zero-2:~$ sudo nano /boot/firmware/config.txt
mjrovai@rpi-zero-2:~$ libcamera-hello --list-cameras
Available cameras
-----
0 : ov5647 [2592x1944 10-bit GBRG] (/base/soc/i2c0mux/i2c@1/ov5647@36)
    Modes: 'SGBRG10_CSI2P' : 640x480 [58.92 fps - (16, 0)/2560x1920 crop]
            1296x972 [43.25 fps - (0, 0)/2592x1944 crop]
            1920x1080 [30.62 fps - (348, 434)/1928x1080 crop]
            2592x1944 [15.63 fps - (0, 0)/2592x1944 crop]
```

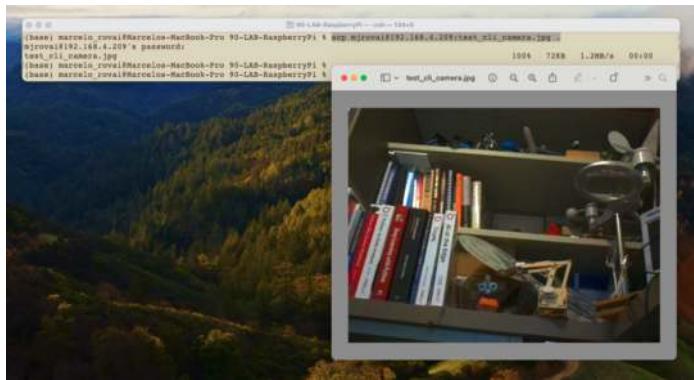
```
mjrovai@rpi-5:~$ libcamera-hello --list-cameras
Available cameras
-----
0 : imx219 [3280x2464 10-bit RGGB] (/base/axi/pcie@120000/rpi/i2c@880000/imx219@10)
    Modes: 'SRGGB10_CSI2P' : 640x480 [206.65 fps - (1000, 752)/1280x960 crop]
            1640x1232 [41.85 fps - (0, 0)/3280x2464 crop]
            1920x1080 [47.57 fps - (680, 692)/1920x1080 crop]
            3280x2464 [21.19 fps - (0, 0)/3280x2464 crop]
    'SRGGB8' : 640x480 [206.65 fps - (1000, 752)/1280x960 crop]
            1640x1232 [83.70 fps - (0, 0)/3280x2464 crop]
            1920x1080 [47.57 fps - (680, 692)/1920x1080 crop]
            3280x2464 [21.19 fps - (0, 0)/3280x2464 crop]
```

libcamera is an open-source software library that supports camera systems directly from the Linux operating system on Arm processors. It minimizes proprietary code running on the Broadcom GPU.

Let's capture a jpeg image with a resolution of 640×480 for testing and save it to a file named `test_cli_camera.jpg`

```
rpicam-jpeg --output test_cli_camera.jpg --width 640 --height 480
```

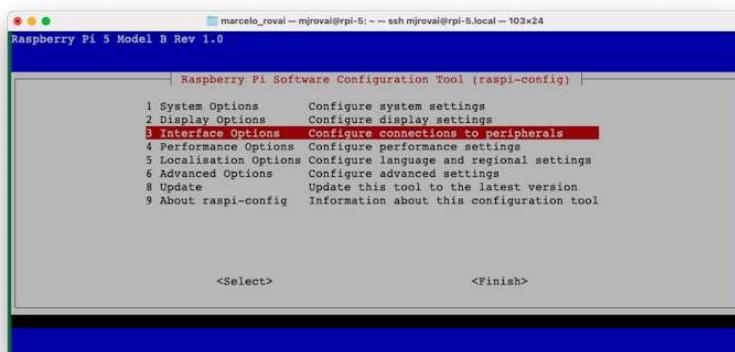
if we want to see the file saved, we should use `ls -f`, which lists all current directory content in long format. As before, we can use `scp` to view the image:



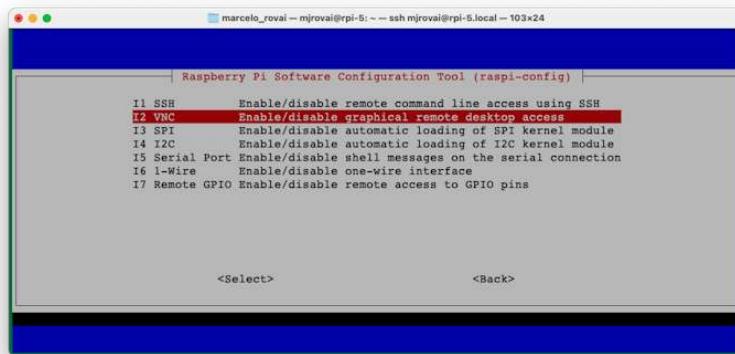
Running the Raspi Desktop remotely

While we've primarily interacted with the Raspberry Pi using terminal commands via SSH, we can access the whole graphical desktop environment remotely if we have installed the complete Raspberry Pi OS (for example, Raspberry Pi OS (64-bit)). This can be particularly useful for tasks that benefit from a visual interface. To enable this functionality, we must set up a VNC (Virtual Network Computing) server on the Raspberry Pi. Here's how to do it:

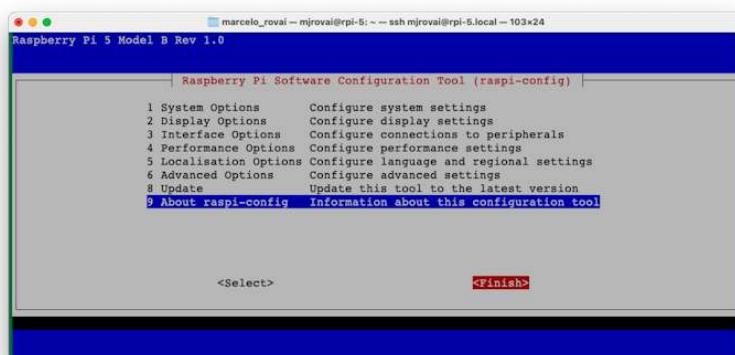
1. Enable the VNC Server:
 - Connect to your Raspberry Pi via SSH.
 - Run the Raspberry Pi configuration tool by entering:
`sudo raspi-config`
 - Navigate to `Interface Options` using the arrow keys.



- Select VNC and Yes to enable the VNC server.



- Exit the configuration tool, saving changes when prompted.



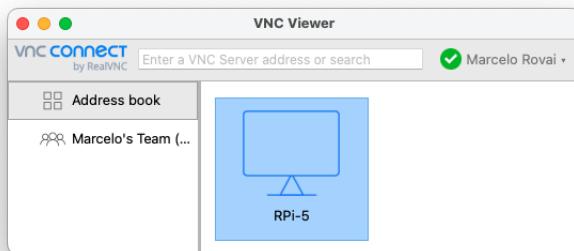
2. Install a VNC Viewer on Your Computer:
 - Download and install a VNC viewer application on your main computer. Popular options include RealVNC Viewer, TightVNC, or VNC Viewer by RealVNC. We will install VNC Viewer by RealVNC.
3. Once installed, you should confirm the Raspi IP address. For example, on the terminal, you can use:
`hostname -I`



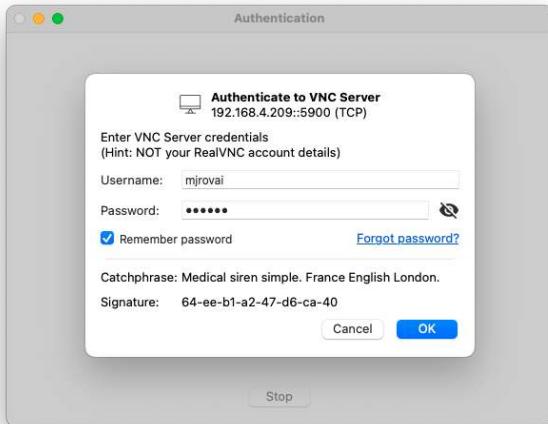
```
marcelo_rovai@rpi-5: ~ ssh mjrovai@rpi-5.local - 57x5
mjrovai@rpi-5:~$ hostname -I
192.168.4.209 fde3:6154:baa3:1:afid:2f29:d5a4:8fea
mjrovai@rpi-5:~$
```

4. Connect to Your Raspberry Pi:

- Open your VNC viewer application.



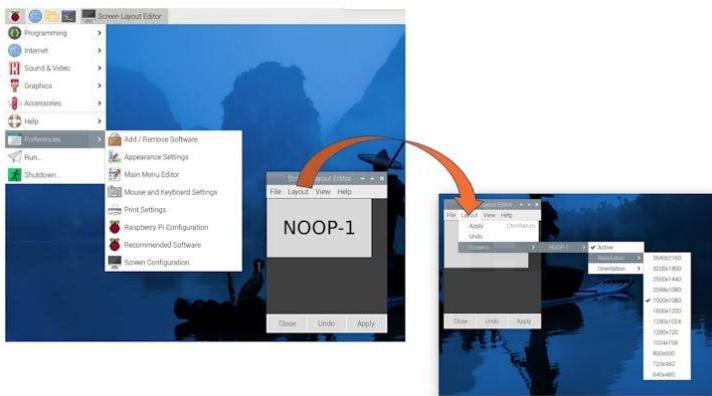
- Enter your Raspberry Pi's IP address and hostname.
- When prompted, enter your Raspberry Pi's username and password.



5. The Raspberry Pi 5 Desktop should appear on your computer monitor.



6. Adjust Display Settings (if needed):
 - Once connected, adjust the display resolution for optimal viewing. This can be done through the Raspberry Pi's desktop settings or by modifying the config.txt file.
 - Let's do it using the desktop settings. Reach the menu (the Raspberry Icon at the left upper corner) and select the best screen definition for your monitor:



- Raspi-4 can be used for Image Classification and Object Detection Labs but will not work well with LLMs (SLM).
- For Raspi-5, consider using an active cooler for temperature management during intensive tasks, as in the LLMs (SLMs) lab.

Remember to adjust your project requirements based on the specific Raspberry Pi model you're using. The Raspi-Zero is great for low-power, space-constrained projects, while the Raspi-4 or 5 models are better suited for more computationally intensive tasks.

Image Classification

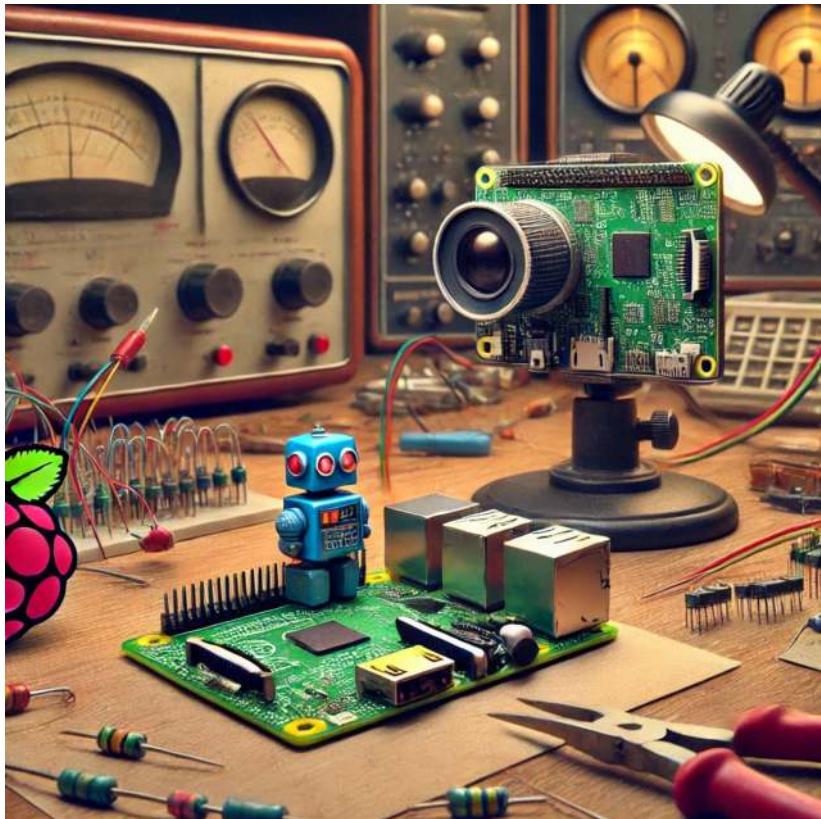


Figure 1.22: DALL-E prompt - A cover image for an 'Image Classification' chapter in a Raspberry Pi tutorial, designed in the same vintage 1950s electronics lab style as previous covers. The scene should feature a Raspberry Pi connected to a camera module, with the camera capturing a photo of the small blue robot provided by the user. The robot should be placed on a workbench, surrounded by classic lab tools like soldering irons, resistors, and wires. The lab background should include vintage equipment like oscilloscopes and tube radios, maintaining the detailed and nostalgic feel of the era. No text or logos should be included.

Overview

Image classification is a fundamental task in computer vision that involves categorizing an image into one of several predefined classes. It's a cornerstone of artificial intelligence, enabling machines to interpret and understand visual information in a way that mimics human perception.

Image classification refers to assigning a label or category to an entire image based on its visual content. This task is crucial in computer vision and has numerous applications across various industries. Image classification's importance lies in its ability to automate visual understanding tasks that would otherwise require human intervention.

Applications in Real-World Scenarios

Image classification has found its way into numerous real-world applications, revolutionizing various sectors:

- Healthcare: Assisting in medical image analysis, such as identifying abnormalities in X-rays or MRIs.
- Agriculture: Monitoring crop health and detecting plant diseases through aerial imagery.
- Automotive: Enabling advanced driver assistance systems and autonomous vehicles to recognize road signs, pedestrians, and other vehicles.
- Retail: Powering visual search capabilities and automated inventory management systems.
- Security and Surveillance: Enhancing threat detection and facial recognition systems.
- Environmental Monitoring: Analyzing satellite imagery for deforestation, urban planning, and climate change studies.

Advantages of Running Classification on Edge Devices like Raspberry Pi

Implementing image classification on edge devices such as the Raspberry Pi offers several compelling advantages:

1. Low Latency: Processing images locally eliminates the need to send data to cloud servers, significantly reducing response times.

2. Offline Functionality: Classification can be performed without an internet connection, making it suitable for remote or connectivity-challenged environments.
3. Privacy and Security: Sensitive image data remains on the local device, addressing data privacy concerns and compliance requirements.
4. Cost-Effectiveness: Eliminates the need for expensive cloud computing resources, especially for continuous or high-volume classification tasks.
5. Scalability: Enables distributed computing architectures where multiple devices can work independently or in a network.
6. Energy Efficiency: Optimized models on dedicated hardware can be more energy-efficient than cloud-based solutions, which is crucial for battery-powered or remote applications.
7. Customization: Deploying specialized or frequently updated models tailored to specific use cases is more manageable.

We can create more responsive, secure, and efficient computer vision solutions by leveraging the power of edge devices like Raspberry Pi for image classification. This approach opens up new possibilities for integrating intelligent visual processing into various applications and environments.

In the following sections, we'll explore how to implement and optimize image classification on the Raspberry Pi, harnessing these advantages to create powerful and efficient computer vision systems.

Setting Up the Environment

Updating the Raspberry Pi

First, ensure your Raspberry Pi is up to date:

```
sudo apt update  
sudo apt upgrade -y
```

Installing Required Libraries

Install the necessary libraries for image processing and machine learning:

```
sudo apt install python3-pip  
sudo rm /usr/lib/python3.11/EXTERNALLY-MANAGED  
pip3 install --upgrade pip
```

Setting up a Virtual Environment (Optional but Recommended)

Create a virtual environment to manage dependencies:

```
python3 -m venv ~/tflite  
source ~/tflite/bin/activate
```

Installing TensorFlow Lite

We are interested in performing **inference**, which refers to executing a TensorFlow Lite model on a device to make predictions based on input data. To perform an inference with a TensorFlow Lite model, we must run it through an **interpreter**. The TensorFlow Lite interpreter is designed to be lean and fast. The interpreter uses a static graph ordering and a custom (less-dynamic) memory allocator to ensure minimal load, initialization, and execution latency.

We'll use the TensorFlow Lite runtime for Raspberry Pi, a simplified library for running machine learning models on mobile and embedded devices, without including all TensorFlow packages.

```
pip install tflite_runtime --no-deps
```

The wheel installed: tflite_runtime-2.14.0-cp311-cp311-manylinux_2_34_aarch64.whl

Installing Additional Python Libraries

Install required Python libraries for use with Image Classification:

If you have another version of Numpy installed, first uninstall it.

```
pip3 uninstall numpy
```

Install version 1.23.2, which is compatible with the tflite_runtime.

```
pip3 install numpy==1.23.2
```

```
pip3 install Pillow matplotlib
```

Creating a working directory:

If you are working on the Raspi-Zero with the minimum OS (No Desktop), you may not have a user-pre-defined directory tree (you can check it with `ls`). So, let's create one:

```
mkdir Documents  
cd Documents/  
mkdir TFLITE  
cd TFLITE/  
mkdir IMG_CLASS  
cd IMG_CLASS  
mkdir models  
cd models
```

On the Raspi-5, the /Documents should be there.

Get a pre-trained Image Classification model:

An appropriate pre-trained model is crucial for successful image classification on resource-constrained devices like the Raspberry Pi. **MobileNet** is designed for mobile and embedded vision applications with a good balance between accuracy and speed. Versions: MobileNetV1, MobileNetV2, MobileNetV3. Let's download the V2:

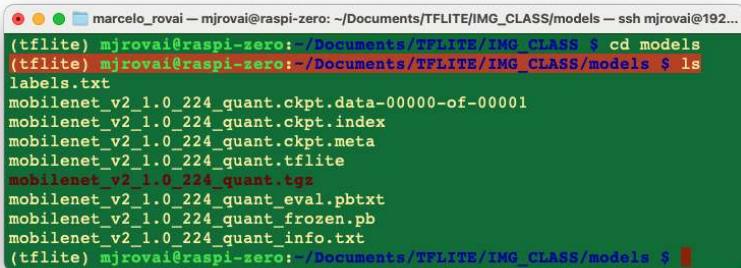
```
# One long line, split with backslash \  
wget https://storage.googleapis.com/download.tensorflow.org/\  
models/tflite_11_05_08/mobilenet_v2_1.0_224_quant.tgz
```

```
tar xzf mobilenet_v2_1.0_224_quant.tgz
```

Get its labels:

```
wget https://raw.githubusercontent.com/Mjrovai/EdgeML-with-Raspberry-Pi/refs/heads/\  
main/IMG_CLASS/models/labels.txt
```

In the end, you should have the models in its directory:



```
marcelo_rovai - mjrovai@raspi-zero: ~/Documents/TFLITE/IMG_CLASS/models -- ssh mjrovai@192.168.4.210 cd models
(tflite) mjrovai@raspi-zero: ~/Documents/TFLITE/IMG_CLASS $ ls
labels.txt
mobilenet_v2_1.0_224_quant.ckpt.data-00000-of-00001
mobilenet_v2_1.0_224_quant.ckpt.index
mobilenet_v2_1.0_224_quant.ckpt.meta
mobilenet_v2_1.0_224_quant.tflite
mobilenet_v2_1.0_224_quant.tgz
mobilenet_v2_1.0_224_quant_eval.pbtxt
mobilenet_v2_1.0_224_quant_frozen.pb
mobilenet_v2_1.0_224_quant_info.txt
(tflite) mjrovai@raspi-zero: ~/Documents/TFLITE/IMG_CLASS/models $
```

We will only need the `mobilenet_v2_1.0_224_quant.tflite` model and the `labels.txt`. You can delete the other files.

Setting up Jupyter Notebook (Optional)

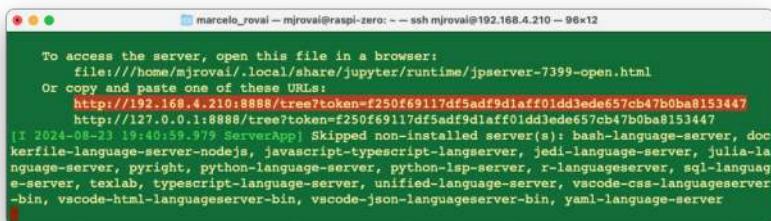
If you prefer using Jupyter Notebook for development:

```
pip3 install jupyter
jupyter notebook --generate-config
```

To run Jupyter Notebook, run the command (change the IP address for yours):

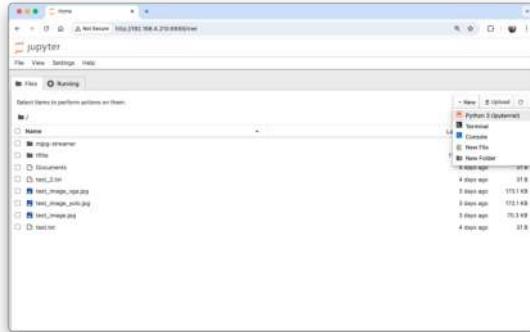
```
jupyter notebook --ip=192.168.4.210 --no-browser
```

On the terminal, you can see the local URL address to open the notebook:



```
marcelo_rovai - mjrovai@raspi-zero: ~ -- ssh mjrovai@192.168.4.210 - 96x12
To access the server, open this file in a browser:
  file:///home/mjrovai/.local/share/jupyter/runtime/jpserver-7399-open.html
Or copy and paste one of these URLs:
  http://192.168.4.210:8888/tree?token=f250f69117df5adf9d1aff01dd3ede657cb47b0ba8153447
  http://127.0.0.1:8888/tree?token=f250f69117df5adf9d1aff01dd3ede657cb47b0ba8153447
[I 2024-08-23 19:40:59.979 ServerApp] Skipped non-installed server(s): bash-language-server, dockerfile-language-server-nodejs, javascript-typescript-langserver, jedi-language-server, julia-language-server, pyright, python-language-server, python-lsp-server, r-languageserver, sql-language-server, texlab, typescript-language-server, unified-language-server, vscode-css-languageserver-bin, vscode-html-languageserver-bin, vscode-json-languageserver-bin, yaml-language-server
```

You can access it from another device by entering the Raspberry Pi's IP address and the provided token in a web browser (you can copy the token from the terminal).



Define your working directory in the Raspi and create a new Python 3 notebook.

Verifying the Setup

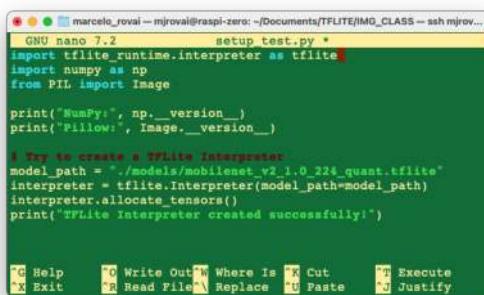
Test your setup by running a simple Python script:

```
import tensorflow as tf
import numpy as np
from PIL import Image

print("NumPy:", np.__version__)
print("Pillow:", Image.__version__)

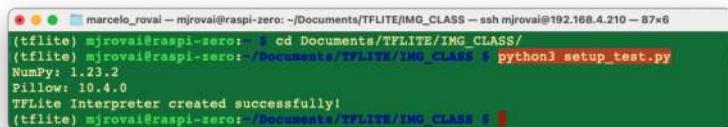
# Try to create a TFLite Interpreter
model_path = "./models/mobilenet_v2_1.0_224_quant.tflite"
interpreter = tf.lite.Interpreter(model_path=model_path)
interpreter.allocate_tensors()
print("TFLite Interpreter created successfully!")
```

You can create the Python script using nano on the terminal, saving it with **CTRL+O + ENTER + CTRL+X**



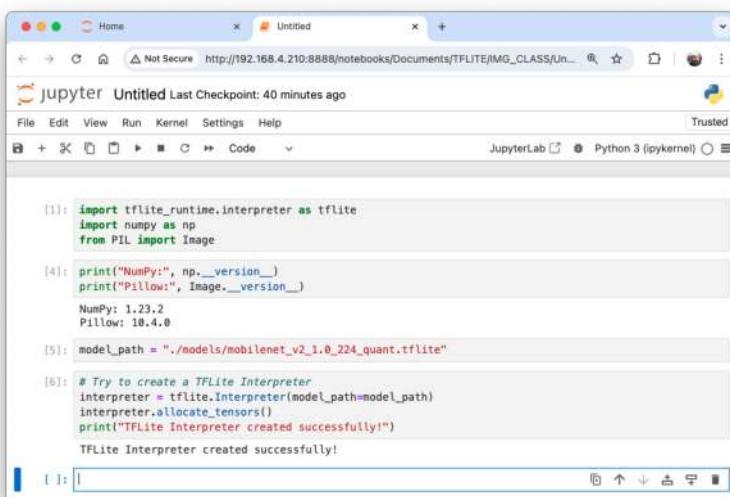
```
marcelo_roval -- mjroval@raspi-zero: ~/Documents/TFLITE/IMG_CLASS -- ssh mjroval...  
GNU nano 7.2      setup_test.py *  
import tensorflow.runtime.interpreter as tflite  
import numpy as np  
from PIL import Image  
  
print("NumPy:", np.__version__)  
print("Pillow:", Image.__version__)  
  
# Try to create a TFLite Interpreter  
model_path = "./models/mobilenet_v2_1.0_224_quant.tflite"  
interpreter = tflite.Interpreter(model_path=model_path)  
interpreter.allocate_tensors()  
print("TFLite Interpreter created successfully!")
```

And run it with the command:



```
marcelo_roval -- mjroval@raspi-zero: ~/Documents/TFLITE/IMG_CLASS -- ssh mjroval@192.168.4.210 - 87x6  
(tflite) mjroval@raspi-zero: ~ cd Documents/TFLITE/IMG_CLASS/  
(tflite) mjroval@raspi-zero: ~/Documents/TFLITE/IMG_CLASS $ python3 setup_test.py  
NumPy: 1.23.2  
Pillow: 10.4.0  
TFLite Interpreter created successfully!  
(tflite) mjroval@raspi-zero: ~/Documents/TFLITE/IMG_CLASS $
```

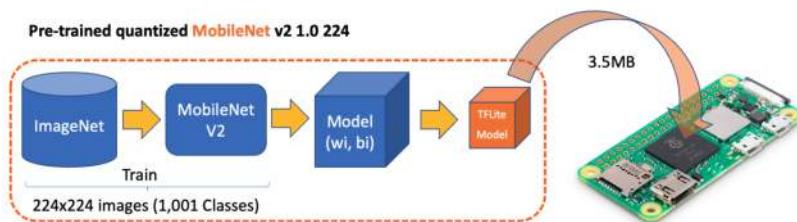
Or you can run it directly on the Notebook:



```
[1]: import tensorflow.runtime.interpreter as tflite  
import numpy as np  
from PIL import Image  
  
[4]: print("NumPy:", np.__version__)  
print("Pillow:", Image.__version__)  
NumPy: 1.23.2  
Pillow: 10.4.0  
  
[5]: model_path = "./models/mobilenet_v2_1.0_224_quant.tflite"  
  
[6]: # Try to create a TFLite Interpreter  
interpreter = tflite.Interpreter(model_path=model_path)  
interpreter.allocate_tensors()  
print("TFLite Interpreter created successfully!")  
TFLite Interpreter created successfully!
```

Making inferences with Mobilenet V2

In the last section, we set up the environment, including downloading a popular pre-trained model, Mobilenet V2, trained on ImageNet's 224×224 images (1.2 million) for 1,001 classes (1,000 object categories plus 1 background). The model was converted to a compact 3.5 MB TensorFlow Lite format, making it suitable for the limited storage and memory of a Raspberry Pi.



Let's start a new notebook to follow all the steps to classify one image:

Import the needed libraries:

```
import time
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import tensorflow as tf
```

Load the TFLite model and allocate tensors:

```
model_path = "./models/mobilenet_v2_1.0_224_quant.tflite"
interpreter = tf.Interpreter(model_path=model_path)
interpreter.allocate_tensors()
```

Get input and output tensors.

```
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()
```

Input details will give us information about how the model should be fed with an image. The shape of $(1, 224, 224, 3)$ informs us that an image with dimensions $(224 \times 224 \times 3)$ should be input one by one (Batch Dimension: 1).

```
input_details
[{'name': 'input',
 'index': 171,
 'shape': array([ 1, 224, 224,   3], dtype=int32), ← Input Image Shape
 'shape_signature': array([ 1, 224, 224,   3], dtype=int32),
 'dtype': numpy.uint8,
 'quantization': (0.0078125, 128),
 'quantization_parameters': {'scales': array([0.0078125], dtype=float32),
 'zero_points': array([128], dtype=int32),
 'quantized_dimension': 0},
 'sparsity_parameters': {}}]
```

The **output details** show that the inference will result in an array of 1,001 integer values. Those values result from the image classification, where each value is the probability of that specific label being related to the image.

```
output_details
[{'name': 'output',
 'index': 172,
 'shape': array([ 1, 1001], dtype=int32), ← Output model
 'shape_signature': array([ 1, 1001], dtype=int32),
 'dtype': numpy.uint8,
 'quantization': (0.09889253973960876, 58),
 'quantization_parameters': {'scales': array([0.09889254], dtype=float32),
 'zero_points': array([58], dtype=int32),
 'quantized_dimension': 0},
 'sparsity_parameters': {}}]
```

Let's also inspect the dtype of input details of the model

```
input_dtype = input_details[0] ["dtype"]
input_dtype

dtype('uint8')
```

This shows that the input image should be raw pixels (0 - 255).

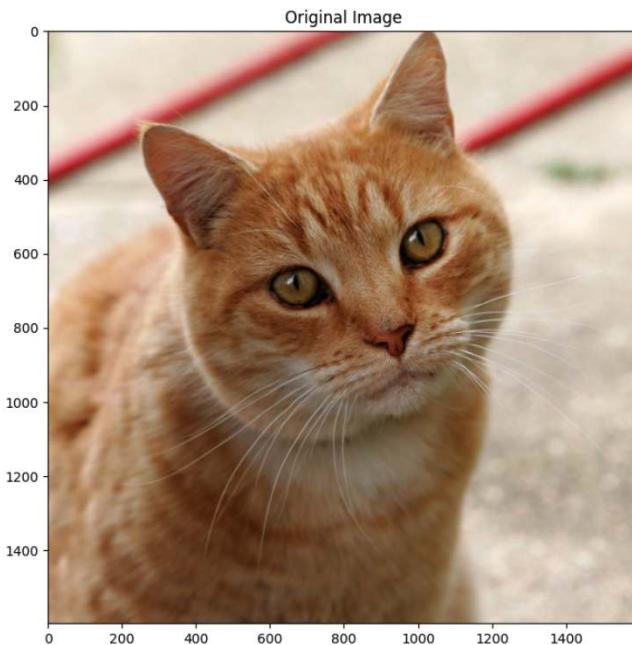
Let's get a test image. You can transfer it from your computer or download one for testing. Let's first create a folder under our working directory:

```
mkdir images
cd images
wget https://upload.wikimedia.org/wikipedia/commons/3/3a/Cat03.jpg
```

Let's load and display the image:

```
# Load the image
img_path = "./images/Cat03.jpg"
img = Image.open(img_path)

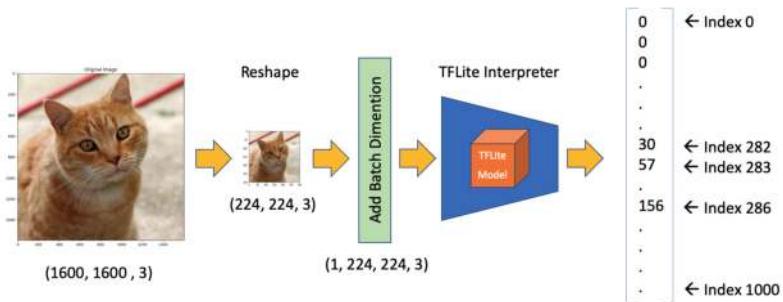
# Display the image
plt.figure(figsize=(8, 8))
plt.imshow(img)
plt.title("Original Image")
plt.show()
```



We can see the image size running the command:

```
width, height = img.size
```

That shows us that the image is an RGB image with a width of 1600 and a height of 1600 pixels. So, to use our model, we should reshape it to (224, 224, 3) and add a batch dimension of 1, as defined in input details: (1, 224, 224, 3). The inference result, as shown in output details, will be an array with a 1001 size, as shown below:



So, let's reshape the image, add the batch dimension, and see the result:

```
img = img.resize(
    (input_details[0]["shape"][1], input_details[0]["shape"][2])
)
input_data = np.expand_dims(img, axis=0)
input_data.shape
```

The input_data shape is as expected: (1, 224, 224, 3)

Let's confirm the dtype of the input data:

```
input_data.dtype
```

```
dtype('uint8')
```

The input data dtype is 'uint8', which is compatible with the dtype expected for the model.

Using the input_data, let's run the interpreter and get the predictions (output):

```
interpreter.set_tensor(input_details[0]["index"], input_data)
interpreter.invoke()
predictions = interpreter.get_tensor(output_details[0]["index"])[0]
```

The prediction is an array with 1001 elements. Let's get the Top-5 indices where their elements have high values:

```
top_k_results = 5
top_k_indices = np.argsort(predictions)[-1:][:-top_k_results]
top_k_indices
```

The top_k_indices is an array with 5 elements: `array([283, 286, 282])`

So, 283, 286, 282, 288, and 479 are the image's most probable classes. Having the index, we must find to what class it appoints (such as car, cat, or dog). The text file downloaded with the model has a label associated with each index from 0 to 1,000. Let's use a function to load the .txt file as a list:

```
def load_labels(filename):
    with open(filename, "r") as f:
        return [line.strip() for line in f.readlines()]
```

And get the list, printing the labels associated with the indexes:

```
labels_path = "./models/labels.txt"
labels = load_labels(labels_path)

print(labels[286])
print(labels[283])
print(labels[282])
print(labels[288])
print(labels[479])
```

As a result, we have:

```
Egyptian cat
tiger cat
tabby
lynx
carton
```

At least the four top indices are related to felines. The **prediction** content is the probability associated with each one of the labels. As we saw on output details, those values are quantized and should be dequantized and apply softmax.

```
scale, zero_point = output_details[0] ["quantization"]
dequantized_output = (
    predictions.astype(np.float32) - zero_point
) * scale
exp_output = np.exp(dequantized_output - np.max(dequantized_output))
probabilities = exp_output / np.sum(exp_output)
```

Let's print the top-5 probabilities:

```

print(probabilities[286])
print(probabilities[283])
print(probabilities[282])
print(probabilities[288])
print(probabilities[479])

0.27741462
0.3732285
0.16919471
0.10319158
0.023410844

```

For clarity, let's create a function to relate the labels with the probabilities:

```

for i in range(top_k_results):
    print(
        "\t{:20}: {:.%}" .format(
            labels[top_k_indices[i]],
            (int(probabilities[top_k_indices[i]] * 100)),
        )
    )

    tiger cat      : 37%
Egyptian cat    : 27%
tabby           : 16%
lynx            : 10%
carton          : 2%

```

Define a general Image Classification function

Let's create a general function to give an image as input, and we get the Top-5 possible classes:

```

def image_classification(
    img_path, model_path, labels, top_k_results=5
):
    # load the image
    img = Image.open(img_path)
    plt.figure(figsize=(4, 4))
    plt.imshow(img)
    plt.axis("off")

    # Load the TFLite model
    interpreter = tflite.Interpreter(model_path=model_path)

```

```
interpreter.allocate_tensors()

# Get input and output tensors
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

# Preprocess
img = img.resize(
    (input_details[0]["shape"][1], input_details[0]["shape"][2])
)
input_data = np.expand_dims(img, axis=0)

# Inference on Raspi-Zero
interpreter.set_tensor(input_details[0]["index"], input_data)
interpreter.invoke()

# Obtain results and map them to the classes
predictions = interpreter.get_tensor(output_details[0]["index"])[
    0
]

# Get indices of the top k results
top_k_indices = np.argsort(predictions)[::-1][:top_k_results]

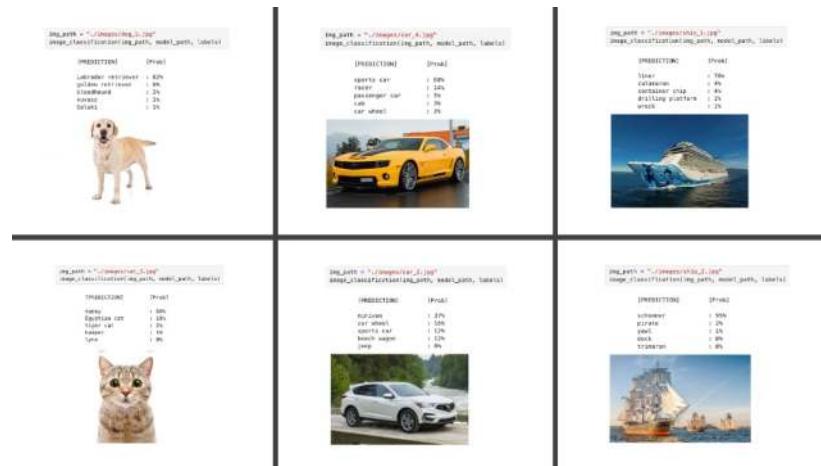
# Get quantization parameters
scale, zero_point = output_details[0]["quantization"]

# Dequantize the output and apply softmax
dequantized_output = (
    predictions.astype(np.float32) - zero_point
) * scale
exp_output = np.exp(
    dequantized_output - np.max(dequantized_output)
)
probabilities = exp_output / np.sum(exp_output)

print("\n\t[PREDICTION] [Prob]\n")
for i in range(top_k_results):
    print(
        "\t{:20}: {}%".format(
            labels[top_k_indices[i]],
            (int(probabilities[top_k_indices[i]] * 100)),
        )
    )
```

)

And loading some images for testing, we have:



Testing with a model trained from scratch

Let's get a TFLite model trained from scratch. For that, you can follow the Notebook:

CNN to classify Cifar-10 dataset

In the notebook, we trained a model using the CIFAR10 dataset, which contains 60,000 images from 10 classes of CIFAR (*airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck*). CIFAR has 32×32 color images (3 color channels) where the objects are not centered and can have the object with a background, such as airplanes that might have a cloudy sky behind them! In short, small but real images.

The CNN trained model (*cifar10_model.keras*) had a size of 2.0MB. Using the *TFLite Converter*, the model *cifar10.tflite* became with 674MB (around 1/3 of the original size).



On the notebook Cifar 10 - Image Classification on a Raspi with TFLite (which can be run over the Raspi), we can follow the same steps we did with the *mobilenet_v2_1.0_224_quant.tflite*. Below are examples

of images using the *General Function for Image Classification* on a Raspi-Zero, as shown in the last section.



Installing Picamera2

Picamera2, a Python library for interacting with Raspberry Pi's camera, is based on the *libcamera* camera stack, and the Raspberry Pi foundation maintains it. The Picamera2 library is supported on all Raspberry Pi models, from the Pi Zero to the RPi 5. It is already installed system-wide on the Raspi, but we should make it accessible within the virtual environment.

1. First, activate the virtual environment if it's not already activated:

```
source ~/tf-lite/bin/activate
```

2. Now, let's create a .pth file in your virtual environment to add the system site-packages path:

```
echo "/usr/lib/python3/dist-packages" > \
$VIRTUAL_ENV/lib/python3.11/
site-packages/system_site_packages.pth
```

Note: If your Python version differs, replace `python3.11` with the appropriate version.

3. After creating this file, try importing picamera2 in Python:

```
python3
>>> import picamera2
>>> print(picamera2.__file__)
```

The above code will show the file location of the `picamera2` module itself, proving that the library can be accessed from the environment.

```
/home/mjrovai/tflite/lib/python3.11/site-packages/
picamera2/__init__.py
```

You can also list the available cameras in the system:

```
>>> print(Picamera2.global_camera_info())
```

In my case, with a USB installed, I got:



```
marcelo_roval - mjroval@raspi-zero: ~ sah mjroval@192.168.4.210 ~ 118x7
>>> import picamera2
>>> print(picamera2._File__file)
/home/mjroval/.flite/lib/python3.11/site-packages/picamera2/_init__.py
>>> print(picamera2.global_camera_info())
[{:id:0, :model:'USB 3.0 Camera' (USB Camera), :location:'/base/soc/usb#7@980000-1:1.0-0c45:1919', :id:0}]
>>>
```

Now that we've confirmed picamera2 is working in the environment with an index 0, let's try a simple Python script to capture an image from your USB camera:

```
from picamera2 import Picamera2
import time

# Initialize the camera
picam2 = Picamera2() # default is index 0

# Configure the camera
config = picam2.create_still_configuration(main={"size": (640, 480)})
picam2.configure(config)

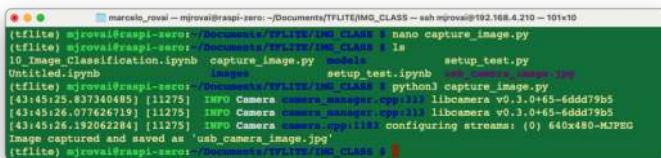
# Start the camera
picam2.start()

# Wait for the camera to warm up
time.sleep(2)

# Capture an image
picam2.capture_file("usb_camera_image.jpg")
print("Image captured and saved as 'usb_camera_image.jpg'")

# Stop the camera
picam2.stop()
```

Use the Nano text editor, the Jupyter Notebook, or any other editor. Save this as a Python script (e.g., `capture_image.py`) and run it. This should capture an image from your camera and save it as "usb_camera_image.jpg" in the same directory as your script.



```
macaile_royal@mjroval@raspi-vero: ~/Documents/TFLITE/IMG_CLASS -- ssh mjroval@192.168.4.210 -- 101x10
(edfile) mjroval@raspi-vero:~/Documents/TFLITE/IMG_CLASS$ nano capture_image.py
(edfile) mjroval@raspi-vero:~/Documents/TFLITE/IMG_CLASS$ ls
10_image_Classification.ipynb  capture_image.py  images  setup.py
Untitled.ipynb  images  setup_test.ipynb  vlc_mjpeg.py
(edfile) mjroval@raspi-vero:~/Documents/TFLITE/IMG_CLASS$ python3 capture_image.py
[43:45:25.837340485] [11275] INFO Camera cameras_manager.cpp:[113] libcamera v0.3.0+65-6dd79b5
[43:45:26.077626719] [11275] INFO Camera cameras_manager.cpp:[113] libcamera v0.3.0+65-6dd79b5
[43:45:26.192062284] [11275] INFO Camera cameras.cpp:[113] configuring streams: (0) 640x480-MJPEG
Image captured and saved as 'usb_camera_image.jpg'
(edfile) mjroval@raspi-vero:~/Documents/TFLITE/IMG_CLASS$
```

If the Jupyter is open, you can see the captured image on your computer. Otherwise, transfer the file from the Raspi to your computer.



If you are working with a Raspi-5 with a whole desktop, you can open the file directly on the device.

Image Classification Project

Now, we will develop a complete Image Classification project using the Edge Impulse Studio. As we did with the Movilinet V2, the trained and converted TFLite model will be used for inference.

The Goal

The first step in any ML project is to define its goal. In this case, it is to detect and classify two specific objects present in one image. For this

project, we will use two small toys: a robot and a small Brazilian parrot (named Periquito). 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. We can use a phone for the image capture, but we will use the Raspi here. Let's set up a simple web server on our Raspberry Pi to view the QVGA (320 x 240) captured images in a browser.

1. First, let's install Flask, a lightweight web framework for Python:
`pip3 install flask`
2. Let's create a new Python script combining image capture with a web server. We'll call it `get_img_data.py`:

```
from flask import Flask, Response, render_template_string,
                 request, redirect, url_for
from picamera2 import Picamera2
import io
import threading
import time
import os
import signal

app = Flask(__name__)

# Global variables
base_dir = "dataset"
picam2 = None
frame = None
```

```
frame_lock = threading.Lock()
capture_counts = {}
current_label = None
shutdown_event = threading.Event()

def initialize_camera():
    global picam2
    picam2 = Picamera2()
    config = picam2.create_preview_configuration(
        main={"size": (320, 240)})
    )
    picam2.configure(config)
    picam2.start()
    time.sleep(2) # Wait for camera to warm up

def get_frame():
    global frame
    while not shutdown_event.is_set():
        stream = io.BytesIO()
        picam2.capture_file(stream, format='jpeg')
        with frame_lock:
            frame = stream.getvalue()
        time.sleep(0.1) # Adjust as needed for smooth preview

def generate_frames():
    while not shutdown_event.is_set():
        with frame_lock:
            if frame is not None:
                yield (b"--frame\r\n"
                       b'Content-Type: image/jpeg\r\n\r\n' +
                       frame + b'\r\n')
        time.sleep(0.1) # Adjust as needed for smooth streaming

def shutdown_server():
    shutdown_event.set()
    if picam2:
        picam2.stop()
    # Give some time for other threads to finish
    time.sleep(2)
    # Send SIGINT to the main process
    os.kill(os.getpid(), signal.SIGINT)

@app.route('/', methods=['GET', 'POST'])
```

```
def index():
    global current_label
    if request.method == 'POST':
        current_label = request.form['label']
        if current_label not in capture_counts:
            capture_counts[current_label] = 0
        os.makedirs(os.path.join(base_dir, current_label),
                    exist_ok=True)
    return redirect(url_for('capture_page'))
return render_template_string('''
<!DOCTYPE html>
<html>
<head>
    <title>Dataset Capture - Label Entry</title>
</head>
<body>
    <h1>Enter Label for Dataset</h1>
    <form method="post">
        <input type="text" name="label" required>
        <input type="submit" value="Start Capture">
    </form>
</body>
</html>
''')
@app.route('/capture')
def capture_page():
    return render_template_string('''
<!DOCTYPE html>
<html>
<head>
    <title>Dataset Capture</title>
    <script>
        var shutdownInitiated = false;
        function checkShutdown() {
            if (!shutdownInitiated) {
                fetch('/check_shutdown')
                    .then(response => response.json())
                    .then(data => {
                        if (data.shutdown) {
                            shutdownInitiated = true;
                            document.getElementById(
                                'video-feed').src =
                            '';
```

```
        document.getElementById(
            'shutdown-message')
        .style.display = 'block';
    }
})
);
}
setInterval(checkShutdown, 1000); // Check
                                every second
</script>
</head>
<body>
<h1>Dataset Capture</h1>
<p>Current Label: {{ label }}</p>
<p>Images captured for this label: {{ capture_count
}}</p>

<div id="shutdown-message" style="display: none;
color: red;">
    Capture process has been stopped.
    You can close this window.
</div>
<form action="/capture_image" method="post">
    <input type="submit" value="Capture Image">
</form>
<form action="/stop" method="post">
    <input type="submit" value="Stop Capture"
style="background-color: #ff6666;">
</form>
<form action="/" method="get">
    <input type="submit" value="Change Label"
style="background-color: #ffff66;">
</form>
</body>
</html>
''' , label=current_label, capture_count=capture_counts.get(
current_label, 0))

@app.route('/video_feed')
def video_feed():
    return Response(generate_frames(),
```

```
mimetype='multipart/x-mixed-replace;
boundary=frame')

@app.route('/capture_image', methods=['POST'])
def capture_image():
    global capture_counts
    if current_label and not shutdown_event.is_set():
        capture_counts[current_label] += 1
        timestamp = time.strftime("%Y%m%d-%H%M%S")
        filename = f"image_{timestamp}.jpg"
        full_path = os.path.join(base_dir, current_label,
                               filename)

        picam2.capture_file(full_path)

    return redirect(url_for('capture_page'))

@app.route('/stop', methods=['POST'])
def stop():
    summary = render_template_string('''
        <!DOCTYPE html>
        <html>
        <head>
            <title>Dataset Capture - Stopped</title>
        </head>
        <body>
            <h1>Dataset Capture Stopped</h1>
            <p>The capture process has been stopped.
                You can close this window.</p>
            <p>Summary of captures:</p>
            <ul>
                {% for label, count in capture_counts.items() %}
                    <li>{{ label }}: {{ count }} images</li>
                {% endfor %}
            </ul>
        </body>
    ''', capture_counts=capture_counts)

    # Start a new thread to shutdown the server
    threading.Thread(target=shutdown_server).start()

    return summary
```

```
@app.route('/check_shutdown')
def check_shutdown():
    return {'shutdown': shutdown_event.is_set()}

if __name__ == '__main__':
    initialize_camera()
    threading.Thread(target=get_frame, daemon=True).start()
    app.run(host='0.0.0.0', port=5000, threaded=True)
```

3. Run this script:

```
python3 get_img_data.py
```

4. Access the web interface:

- On the Raspberry Pi itself (if you have a GUI): Open a web browser and go to <http://localhost:5000>
- From another device on the same network: Open a web browser and go to http://<raspberry_pi_ip>:5000 (Replace <raspberry_pi_ip> with your Raspberry Pi's IP address). For example: <http://192.168.4.210:5000/>

This Python script creates a web-based interface for capturing and organizing image datasets using a Raspberry Pi and its camera. It's handy for machine learning projects that require labeled image data.

Key Features:

1. **Web Interface:** Accessible from any device on the same network as the Raspberry Pi.
2. **Live Camera Preview:** This shows a real-time feed from the camera.
3. **Labeling System:** Allows users to input labels for different categories of images.
4. **Organized Storage:** Automatically saves images in label-specific subdirectories.
5. **Per-Label Counters:** Keeps track of how many images are captured for each label.
6. **Summary Statistics:** Provides a summary of captured images when stopping the capture process.

Main Components:

1. **Flask Web Application:** Handles routing and serves the web interface.
2. **Picamera2 Integration:** Controls the Raspberry Pi camera.
3. **Threaded Frame Capture:** Ensures smooth live preview.
4. **File Management:** Organizes captured images into labeled directories.

Key Functions:

- `initialize_camera()`: Sets up the Picamera2 instance.
- `get_frame()`: Continuously captures frames for the live preview.
- `generate_frames()`: Yields frames for the live video feed.
- `shutdown_server()`: Sets the shutdown event, stops the camera, and shuts down the Flask server
- `index()`: Handles the label input page.
- `capture_page()`: Displays the main capture interface.
- `video_feed()`: Shows a live preview to position the camera
- `capture_image()`: Saves an image with the current label.
- `stop()`: Stops the capture process and displays a summary.

Usage Flow:

1. Start the script on your Raspberry Pi.
2. Access the web interface from a browser.
3. Enter a label for the images you want to capture and press **Start Capture**.

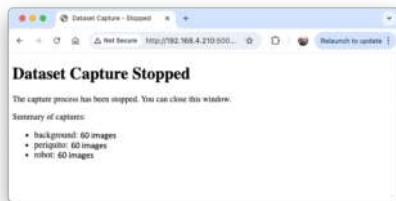


4. Use the live preview to position the camera.

5. Click Capture Image to save images under the current label.



6. Change labels as needed for different categories, selecting Change Label.
7. Click Stop Capture when finished to see a summary.



Technical Notes:

- The script uses threading to handle concurrent frame capture and web serving.
- Images are saved with timestamps in their filenames for uniqueness.
- The web interface is responsive and can be accessed from mobile devices.

Customization Possibilities:

- Adjust image resolution in the `initialize_camera()` function.
Here we used QVGA (320×240).
- Modify the HTML templates for a different look and feel.
- Add additional image processing or analysis steps in the `capture_image()` function.

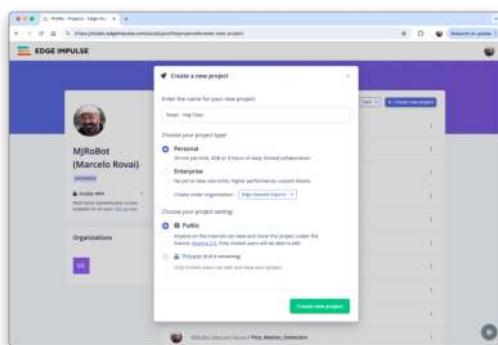
Number of samples on Dataset:

Get around 60 images from each category (*periquito*, *robot* and *background*). Try to capture different angles, backgrounds, and light conditions. On the Raspi, we will end with a folder named `dataset`, which contains 3 sub-folders *periquito*, *robot*, and *background*. one for each class of images.

You can use `Filezilla` to transfer the created dataset to your main computer.

Training the model with Edge Impulse Studio

We will use the Edge Impulse Studio to train our model. Go to the Edge Impulse Page, enter your account credentials, and create a new project:



Here, you can clone a similar project: Raspi - Img Class.

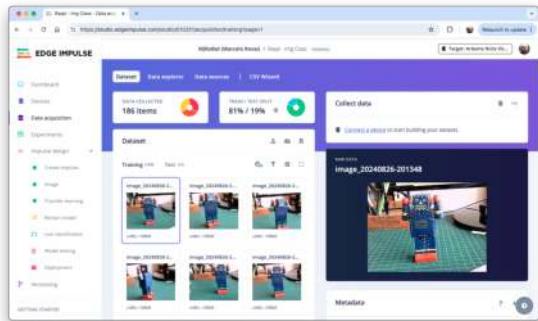
Dataset

We will walk through four main steps using the EI Studio (or Studio). These steps are crucial in preparing our model for use on the Raspi: Dataset, Impulse, Tests, and Deploy (on the Edge Device, in this case, the Raspi).

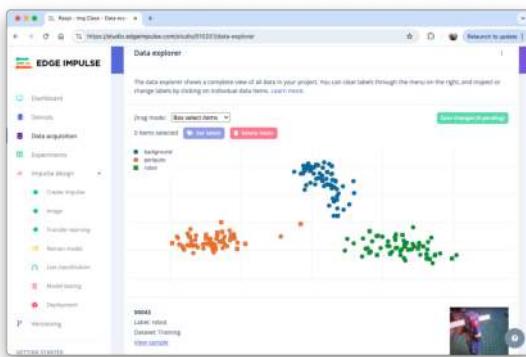
Regarding the Dataset, it is essential to point out that our Original Dataset, captured with the Raspi, will be split into *Training*, *Validation*, and *Test*. The Test Set will be separated from the beginning and reserved for use only in the Test phase after training. The Validation Set will be used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

On Studio, follow the steps to upload the captured data:

1. Go to the Data acquisition tab, and in the UPLOAD DATA section, upload the files from your computer in the chosen categories.
2. Leave to the Studio the splitting of the original dataset into *train* and *test* and choose the label about
3. Repeat the procedure for all three classes. At the end, you should see your “raw data” in the Studio:



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

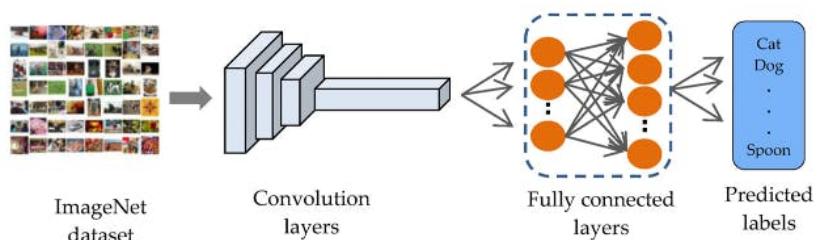


The Impulse Design

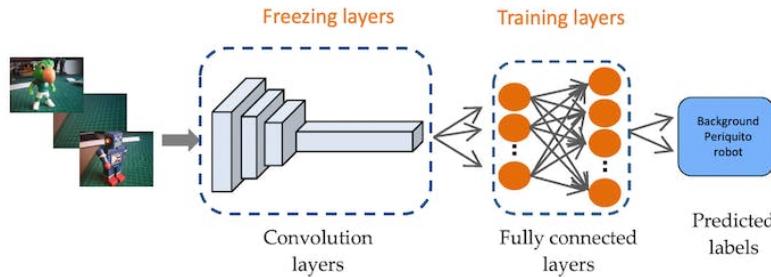
In this phase, we should define how to:

- 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 180 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, we can train a new model for your specific task with fewer data and computational resources and 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 160×160 and squashing them (squared form, without cropping). Select Image and Transfer Learning blocks. Save the Impulse.

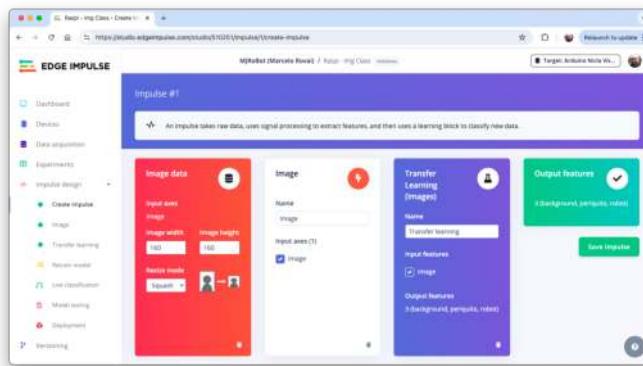
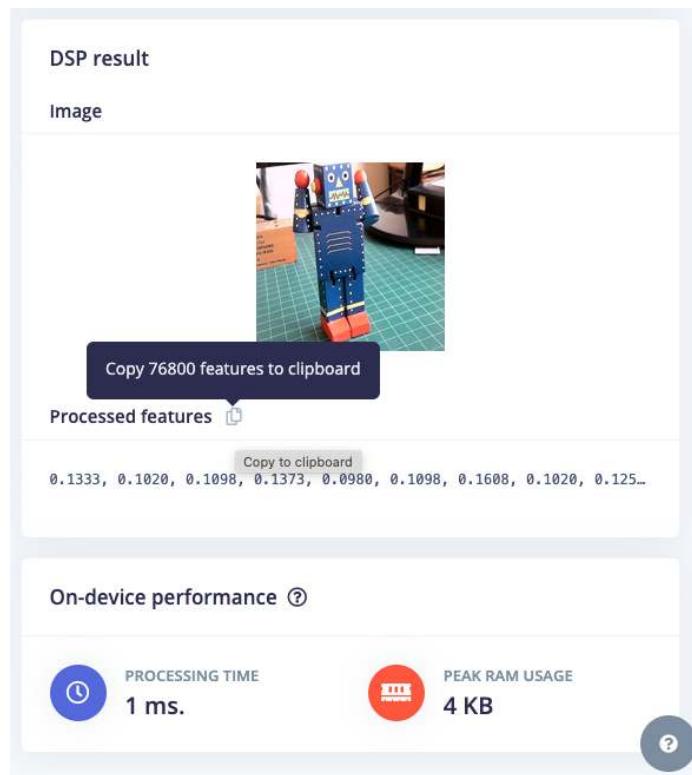


Image Pre-Processing

All the input QVGA/RGB565 images will be converted to 76,800 features ($160 \times 160 \times 3$).



Press Save parameters and select Generate features in the next tab.

Model Design

MobileNet is a family of efficient convolutional neural networks designed for mobile and embedded vision applications. The key features of MobileNet are:

1. Lightweight: Optimized for mobile devices and embedded systems with limited computational resources.
2. Speed: Fast inference times, suitable for real-time applications.
3. Accuracy: Maintains good accuracy despite its compact size.

MobileNetV2, introduced in 2018, improves the original MobileNet architecture. Key features include:

1. Inverted Residuals: Inverted residual structures are used where shortcut connections are made between thin bottleneck layers.
2. Linear Bottlenecks: Removes non-linearities in the narrow layers to prevent the destruction of information.
3. Depth-wise Separable Convolutions: Continues to use this efficient operation from MobileNetV1.

In our project, we will do a Transfer Learning with the MobileNetV2 160x160 1.0, which means that the images used for training (and future inference) should have an *input Size* of 160×160 pixels and a *Width Multiplier* of 1.0 (full width, not reduced). This configuration balances between model size, speed, and accuracy.

Model Training

Another valuable deep learning technique is **Data Augmentation**. Data augmentation improves the accuracy of machine learning models by creating additional artificial data. A data augmentation system makes small, random changes to the 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)

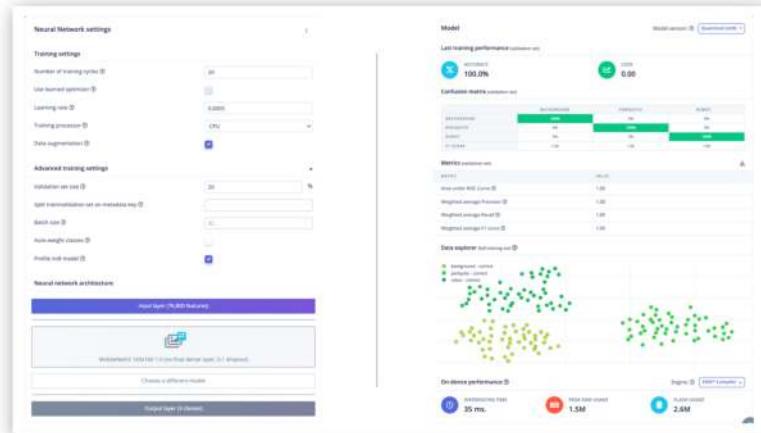
    # Increase the image size, then randomly crop it down to
    # the original dimensions
    resize_factor = random.uniform(1, 1.2)
    new_height = math.floor(resize_factor * INPUT_SHAPE[0])
    new_width = math.floor(resize_factor * INPUT_SHAPE[1])
    image = tf.image.resize_with_crop_or_pad(
        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 dense layer of our model will have 0 neurons with a 10% dropout for overfitting prevention. Here is the Training result:



The result is excellent, with a reasonable 35 ms of latency (for a Raspi-4), which should result in around 30 fps (frames per second) during inference. A Raspi-Zero should be slower, and the Raspi-5, faster.

Trading off: Accuracy versus speed

If faster inference is needed, we should train the model using smaller alphas (0.35, 0.5, and 0.75) or even reduce the image input size, trading with accuracy. However, reducing the input image size and decreasing the alpha (width multiplier) can speed up inference for MobileNet V2, but they have different trade-offs. Let's compare:

1. Reducing Image Input Size:

Pros:

- Significantly reduces the computational cost across all layers.
- Decreases memory usage.
- It often provides a substantial speed boost.

Cons:

- It may reduce the model's ability to detect small features or fine details.
- It can significantly impact accuracy, especially for tasks requiring fine-grained recognition.

2. Reducing Alpha (Width Multiplier):

Pros:

- Reduces the number of parameters and computations in the model.
- Maintains the original input resolution, potentially preserving more detail.
- It can provide a good balance between speed and accuracy.

Cons:

- It may not speed up inference as dramatically as reducing input size.
- It can reduce the model's capacity to learn complex features.

Comparison:

1. Speed Impact:

- Reducing input size often provides a more substantial speed boost because it reduces computations quadratically (halving both width and height reduces computations by about 75%).
- Reducing alpha provides a more linear reduction in computations.

2. Accuracy Impact:

- Reducing input size can severely impact accuracy, especially when detecting small objects or fine details.
- Reducing alpha tends to have a more gradual impact on accuracy.

3. Model Architecture:

- Changing input size doesn't alter the model's architecture.
- Changing alpha modifies the model's structure by reducing the number of channels in each layer.

Recommendation:

1. If our application doesn't require detecting tiny details and can tolerate some loss in accuracy, reducing the input size is often the most effective way to speed up inference.
2. Reducing alpha might be preferable if maintaining the ability to detect fine details is crucial or if you need a more balanced trade-off between speed and accuracy.
3. For best results, you might want to experiment with both:
 - Try MobileNet V2 with input sizes like 160×160 or 92×92
 - Experiment with alpha values like 1.0, 0.75, 0.5 or 0.35.
4. Always benchmark the different configurations on your specific hardware and with your particular dataset to find the optimal balance for your use case.

Remember, the best choice depends on your specific requirements for accuracy, speed, and the nature of the images you're working with. It's often worth experimenting with combinations to find the optimal configuration for your particular use case.

Model Testing

Now, you should take the data set aside at the start of the project and run the trained model using it as input. Again, the result is excellent (92.22%).

Deploying the model

As we did in the previous section, we can deploy the trained model as .tflite and use Raspi to run it using Python.

On the Dashboard tab, go to Transfer learning model (int8 quantized) and click on the download icon:

Download block output			
TITLE	TYPE	SIZE	
image training data	NPY file	150 windows	B
image training labels	NPY file	150 windows	B
image testing data	NPY file	36 windows	B
image testing labels	NPY file	36 windows	B
Transfer learning model	TensorFlow Lite (float32)	9 MB	B
Transfer learning model	TensorFlow Lite (int8 quantized)	3 MB	B
Transfer learning model	Model evaluation metrics (JSON file)	5 KB	B
Transfer learning model	TensorFlow SavedModel	8 MB	B
Transfer learning model	Keras h5 model	8 MB	B

Let's also download the float32 version for comparison

Transfer the model from your computer to the Raspi (./models), for example, using FileZilla. Also, capture some images for inference (./images).

Import the needed libraries:

```
import time
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import tensorflow as tf
```

Define the paths and labels:

```
img_path = "./images/robot.jpg"
model_path = "./models/ei-raspi-img-class-int8-quantized-\
            model.tflite"
labels = ["background", "periquito", "robot"]
```

Note that the models trained on the Edge Impulse Studio will output values with index 0, 1, 2, etc., where the actual labels will follow an alphabetic order.

Load the model, allocate the tensors, and get the input and output tensor details:

```
# Load the TFLite model
interpreter = tf.lite.Interpreter(model_path=model_path)
interpreter.allocate_tensors()
```

```
# Get input and output tensors
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()
```

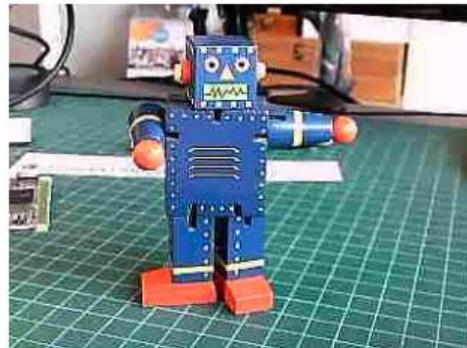
One important difference to note is that the dtype of the input details of the model is now int8, which means that the input values go from -128 to +127, while each pixel of our image goes from 0 to 255. This means that we should pre-process the image to match it. We can check here:

```
input_dtype = input_details[0] ["dtype"]
input_dtype

numpy.int8
```

So, let's open the image and show it:

```
img = Image.open(img_path)
plt.figure(figsize=(4, 4))
plt.imshow(img)
plt.axis("off")
plt.show()
```



And perform the pre-processing:

```
scale, zero_point = input_details[0] ["quantization"]
img = img.resize(
    (input_details[0] ["shape"] [1], input_details[0] ["shape"] [2])
)
img_array = np.array(img, dtype=np.float32) / 255.0
```

```
img_array = (
    (img_array / scale + zero_point).clip(-128, 127).astype(np.int8)
)
input_data = np.expand_dims(img_array, axis=0)
```

Checking the input data, we can verify that the input tensor is compatible with what is expected by the model:

```
input_data.shape, input_data.dtype
```

```
((1, 160, 160, 3), dtype('int8'))
```

Now, it is time to perform the inference. Let's also calculate the latency of the model:

```
# Inference on Raspi-Zero
start_time = time.time()
interpreter.set_tensor(input_details[0]["index"], input_data)
interpreter.invoke()
end_time = time.time()
inference_time = (end_time - start_time) * 1000 # Convert
# to milliseconds
print("Inference time: {:.1f}ms".format(inference_time))
```

The model will take around 125ms to perform the inference in the Raspi-Zero, which is 3 to 4 times longer than a Raspi-5.

Now, we can get the output labels and probabilities. It is also important to note that the model trained on the Edge Impulse Studio has a softmax in its output (different from the original Movilenet V2), and we should use the model's raw output as the "probabilities."

```
# Obtain results and map them to the classes
predictions = interpreter.get_tensor(output_details[0]["index"])[0]

# Get indices of the top k results
top_k_results = 3
top_k_indices = np.argsort(predictions)[-1:][:-top_k_results]

# Get quantization parameters
scale, zero_point = output_details[0]["quantization"]

# Dequantize the output
dequantized_output = (
    predictions.astype(np.float32) - zero_point
) * scale
```

```
probabilities = dequantized_output

print("\n\t[PREDICTION] [Prob]\n")
for i in range(top_k_results):
    print(
        "\t{:20}: {:.2f}%".format(
            labels[top_k_indices[i]],
            probabilities[top_k_indices[i]] * 100,
        )
    )
```

[PREDICTION]	[Prob]
robot	: 99.61%
periquito	: 0.00%
background	: 0.00%

Let's modify the function created before so that we can handle different type of models:

```
def image_classification(
    img_path, model_path, labels, top_k_results=3, apply_softmax=False
):
    # Load the image
    img = Image.open(img_path)
    plt.figure(figsize=(4, 4))
    plt.imshow(img)
    plt.axis("off")

    # Load the TFLite model
    interpreter = tflite.Interpreter(model_path=model_path)
    interpreter.allocate_tensors()

    # Get input and output tensors
    input_details = interpreter.get_input_details()
    output_details = interpreter.get_output_details()

    # Preprocess
    img = img.resize(
        (input_details[0]["shape"][1], input_details[0]["shape"][2])
    )
```

```
input_dtype = input_details[0]["dtype"]

if input_dtype == np.uint8:
    input_data = np.expand_dims(np.array(img), axis=0)
elif input_dtype == np.int8:
    scale, zero_point = input_details[0]["quantization"]
    img_array = np.array(img, dtype=np.float32) / 255.0
    img_array = (
        (img_array / scale + zero_point)
        .clip(-128, 127)
        .astype(np.int8)
    )
    input_data = np.expand_dims(img_array, axis=0)
else: # float32
    input_data = (
        np.expand_dims(np.array(img, dtype=np.float32), axis=0)
        / 255.0
    )

# Inference on Raspi-Zero
start_time = time.time()
interpreter.set_tensor(input_details[0]["index"], input_data)
interpreter.invoke()
end_time = time.time()
inference_time = (
    end_time - start_time
) * 1000 # Convert to milliseconds

# Obtain results
predictions = interpreter.get_tensor(output_details[0]["index"])[
    0
]

# Get indices of the top k results
top_k_indices = np.argsort(predictions)[::-1][:top_k_results]

# Handle output based on type
output_dtype = output_details[0]["dtype"]
if output_dtype in [np.int8, np.uint8]:
    # Dequantize the output
    scale, zero_point = output_details[0]["quantization"]
    predictions = (
        predictions.astype(np.float32) - zero_point
```

```

) * scale

if apply_softmax:
    # Apply softmax
    exp_preds = np.exp(predictions - np.max(predictions))
    probabilities = exp_preds / np.sum(exp_preds)
else:
    probabilities = predictions

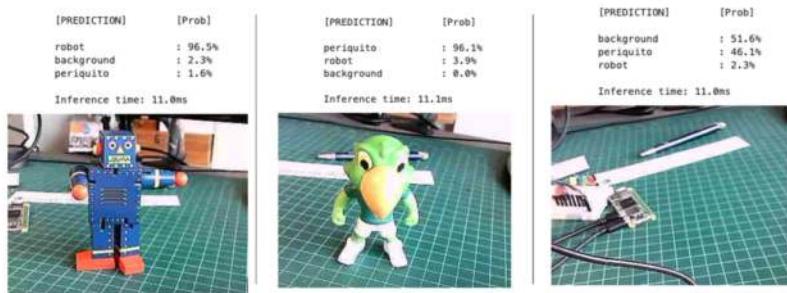
print("\n\t[PREDICTION] [Prob]\n")
for i in range(top_k_results):
    print(
        "\t{:20}: {:.1f}%".format(
            labels[top_k_indices[i]],
            probabilities[top_k_indices[i]] * 100,
        )
    )
print("\n\tInference time: {:.1f}ms".format(inference_time))

```

And test it with different images and the int8 quantized model (**160x160 alpha =1.0**).



Let's download a smaller model, such as the one trained for the Nicla Vision Lab (int8 quantized model, 96x96, alpha = 0.1), as a test. We can use the same function:



The model lost some accuracy, but it is still OK once our model does not look for many details. Regarding latency, we are around **ten times faster** on the Raspi-Zero.

Live Image Classification

Let's develop an app to capture images with the USB camera in real time, showing its classification.

Using the nano on the terminal, save the code below, such as `img_class_live_infer.py`.

```
from flask import Flask, Response, render_template_string,
                 request, jsonify
from picamera2 import Picamera2
import io
import threading
import time
import numpy as np
from PIL import Image
import tensorflow.lite_runtime.interpreter as tflite
from queue import Queue

app = Flask(__name__)

# Global variables
picam2 = None
frame = None
frame_lock = threading.Lock()
is_classifying = False
confidence_threshold = 0.8
model_path = "./models/ei-raspi-img-class-int8-quantized-\
            model.tflite"
```

```
labels = ['background', 'periquito', 'robot']
interpreter = None
classification_queue = Queue(maxsize=1)

def initialize_camera():
    global picam2
    picam2 = Picamera2()
    config = picam2.create_preview_configuration(
        main={"size": (320, 240)})
    )
    picam2.configure(config)
    picam2.start()
    time.sleep(2) # Wait for camera to warm up

def get_frame():
    global frame
    while True:
        stream = io.BytesIO()
        picam2.capture_file(stream, format='jpeg')
        with frame_lock:
            frame = stream.getvalue()
        time.sleep(0.1) # Capture frames more frequently

def generate_frames():
    while True:
        with frame_lock:
            if frame is not None:
                yield (
                    b"--frame\r\n"
                    b"Content-Type: image/jpeg\r\n\r\n"
                    + frame + b'\r\n'
                )
        time.sleep(0.1)

def load_model():
    global interpreter
    if interpreter is None:
        interpreter = tflite.Interpreter(model_path=model_path)
        interpreter.allocate_tensors()
    return interpreter

def classify_image(img, interpreter):
    input_details = interpreter.get_input_details()
```

```
output_details = interpreter.get_output_details()

img = img.resize((input_details[0]['shape'][1],
                  input_details[0]['shape'][2]))
input_data = np.expand_dims(np.array(img), axis=0) \
    .astype(input_details[0]['dtype'])

interpreter.set_tensor(input_details[0]['index'], input_data)
interpreter.invoke()

predictions = interpreter.get_tensor(output_details[0]
                                      ['index'])[0]

# Handle output based on type
output_dtype = output_details[0]['dtype']
if output_dtype in [np.int8, np.uint8]:
    # Dequantize the output
    scale, zero_point = output_details[0]['quantization']
    predictions = (predictions.astype(np.float32) -
                   zero_point) * scale

return predictions

def classification_worker():
    interpreter = load_model()
    while True:
        if is_classifying:
            with frame_lock:
                if frame is not None:
                    img = Image.open(io.BytesIO(frame))
                    predictions = classify_image(img, interpreter)
                    max_prob = np.max(predictions)
                    if max_prob >= confidence_threshold:
                        label = labels[np.argmax(predictions)]
                    else:
                        label = 'Uncertain'
                    classification_queue.put({
                        'label': label,
                        'probability': float(max_prob)
                    })
            time.sleep(0.1) # Adjust based on your needs

@app.route('/')
def index():
    return render_template_string('''
```

```
<!DOCTYPE html>
<html>
<head>
    <title>Image Classification</title>
    <script>
        src="https://code.jquery.com/jquery-3.6.0.min.js">
    </script>
    <script>
        function startClassification() {
            $.post('/start');
            $('#startBtn').prop('disabled', true);
            $('#stopBtn').prop('disabled', false);
        }
        function stopClassification() {
            $.post('/stop');
            $('#startBtn').prop('disabled', false);
            $('#stopBtn').prop('disabled', true);
        }
        function updateConfidence() {
            var confidence = $('#confidence').val();
            $.post('/update_confidence',
                {confidence: confidence}
            );
        }
        function updateClassification() {
            $.get('/get_classification', function(data) {
                $('#classification').text(data.label + ': ' +
                    data.probability.toFixed(2));
            });
        }
        $(document).ready(function() {
            setInterval(updateClassification, 100);
            // Update every 100ms
        });
    </script>
</head>
<body>
    <h1>Image Classification</h1>
    

    <br>
```

```
<button id="startBtn"
        onclick="startClassification()"
        Start Classification
</button>

<button id="stopBtn"
        onclick="stopClassification()"
        disabled>
        Stop Classification
</button>

<br>
<label for="confidence">Confidence Threshold:</label>
<input type="number"
        id="confidence"
        name="confidence"
        min="0" max="1"
        step="0.1"
        value="0.8"
        onchange="updateConfidence()" />

<br>
<div id="classification">
    Waiting for classification...
</div>

</body>
</html>
''')

@app.route('/video_feed')
def video_feed():
    return Response(
        generate_frames(),
        mimetype='multipart/x-mixed-replace; boundary=frame'
    )

@app.route('/start', methods=['POST'])
def start_classification():
    global is_classifying
    is_classifying = True
    return '', 204
```

```
@app.route('/stop', methods=['POST'])
def stop_classification():
    global is_classifying
    is_classifying = False
    return '', 204

@app.route('/update_confidence', methods=['POST'])
def update_confidence():
    global confidence_threshold
    confidence_threshold = float(request.form['confidence'])
    return '', 204

@app.route('/get_classification')
def get_classification():
    if not is_classifying:
        return jsonify({'label': 'Not classifying',
                       'probability': 0})
    try:
        result = classification_queue.get_nowait()
    except Queue.Empty:
        result = {'label': 'Processing', 'probability': 0}
    return jsonify(result)

if __name__ == '__main__':
    initialize_camera()
    threading.Thread(target=get_frame, daemon=True).start()
    threading.Thread(target=classification_worker,
                     daemon=True).start()
    app.run(host='0.0.0.0', port=5000, threaded=True)
```

On the terminal, run:

```
python3 img_class_live_infer.py
```

And access the web interface:

- On the Raspberry Pi itself (if you have a GUI): Open a web browser and go to `http://localhost:5000`
- From another device on the same network: Open a web browser and go to `http://<raspberry_pi_ip>:5000` (Replace `<raspberry_pi_ip>` with your Raspberry Pi's IP address). For example: `http://192.168.4.210:5000`

Here are some screenshots of the app running on an external desktop



Here, you can see the app running on the YouTube:

<https://www.youtube.com/watch?v=o1QsQrpCMw4>

The code creates a web application for real-time image classification using a Raspberry Pi, its camera module, and a TensorFlow Lite model. The application uses Flask to serve a web interface where it is possible to view the camera feed and see live classification results.

Key Components:

1. **Flask Web Application:** Serves the user interface and handles requests.
2. **PiCamera2:** Captures images from the Raspberry Pi camera module.
3. **TensorFlow Lite:** Runs the image classification model.
4. **Threading:** Manages concurrent operations for smooth performance.

Main Features:

- Live camera feed display
- Real-time image classification
- Adjustable confidence threshold
- Start/Stop classification on demand

Code Structure:

1. Imports and Setup:

- Flask for web application
- PiCamera2 for camera control

- TensorFlow Lite for inference
- Threading and Queue for concurrent operations

2. Global Variables:

- Camera and frame management
- Classification control
- Model and label information

3. Camera Functions:

- `initialize_camera()`: Sets up the PiCamera2
- `get_frame()`: Continuously captures frames
- `generate_frames()`: Yields frames for the web feed

4. Model Functions:

- `load_model()`: Loads the TFLite model
- `classify_image()`: Performs inference on a single image

5. Classification Worker:

- Runs in a separate thread
- Continuously classifies frames when active
- Updates a queue with the latest results

6. Flask Routes:

- `/`: Serves the main HTML page
- `/video_feed`: Streams the camera feed
- `/start` and `/stop`: Controls classification
- `/update_confidence`: Adjusts the confidence threshold
- `/get_classification`: Returns the latest classification result

7. HTML Template:

- Displays camera feed and classification results
- Provides controls for starting/stopping and adjusting settings

8. Main Execution:

- Initializes camera and starts necessary threads
- Runs the Flask application

Key Concepts:

1. **Concurrent Operations:** Using threads to handle camera capture and classification separately from the web server.

2. **Real-time Updates:** Frequent updates to the classification results without page reloads.
3. **Model Reuse:** Loading the TFLite model once and reusing it for efficiency.
4. **Flexible Configuration:** Allowing users to adjust the confidence threshold on the fly.

Usage:

1. Ensure all dependencies are installed.
2. Run the script on a Raspberry Pi with a camera module.
3. Access the web interface from a browser using the Raspberry Pi's IP address.
4. Start classification and adjust settings as needed.

Summary:

Image classification has emerged as a powerful and versatile application of machine learning, with significant implications for various fields, from healthcare to environmental monitoring. This chapter has demonstrated how to implement a robust image classification system on edge devices like the Raspi-Zero and Raspi-5, showcasing the potential for real-time, on-device intelligence.

We've explored the entire pipeline of an image classification project, from data collection and model training using Edge Impulse Studio to deploying and running inferences on a Raspi. The process highlighted several key points:

1. The importance of proper data collection and preprocessing for training effective models.
2. The power of transfer learning, allowing us to leverage pre-trained models like MobileNet V2 for efficient training with limited data.
3. The trade-offs between model accuracy and inference speed, especially crucial for edge devices.
4. The implementation of real-time classification using a web-based interface, demonstrating practical applications.

The ability to run these models on edge devices like the Raspi opens up numerous possibilities for IoT applications, autonomous systems, and

real-time monitoring solutions. It allows for reduced latency, improved privacy, and operation in environments with limited connectivity.

As we've seen, even with the computational constraints of edge devices, it's possible to achieve impressive results in terms of both accuracy and speed. The flexibility to adjust model parameters, such as input size and alpha values, allows for fine-tuning to meet specific project requirements.

Looking forward, the field of edge AI and image classification continues to evolve rapidly. Advances in model compression techniques, hardware acceleration, and more efficient neural network architectures promise to further expand the capabilities of edge devices in computer vision tasks.

This project serves as a foundation for more complex computer vision applications and encourages further exploration into the exciting world of edge AI and IoT. Whether it's for industrial automation, smart home applications, or environmental monitoring, the skills and concepts covered here provide a solid starting point for a wide range of innovative projects.

Resources

- Dataset Example
- Setup Test Notebook on a Raspi
- Image Classification Notebook on a Raspi
- CNN to classify Cifar-10 dataset at CoLab
- Cifar 10 - Image Classification on a Raspi
- Python Scripts
- Edge Impulse Project

Object Detection

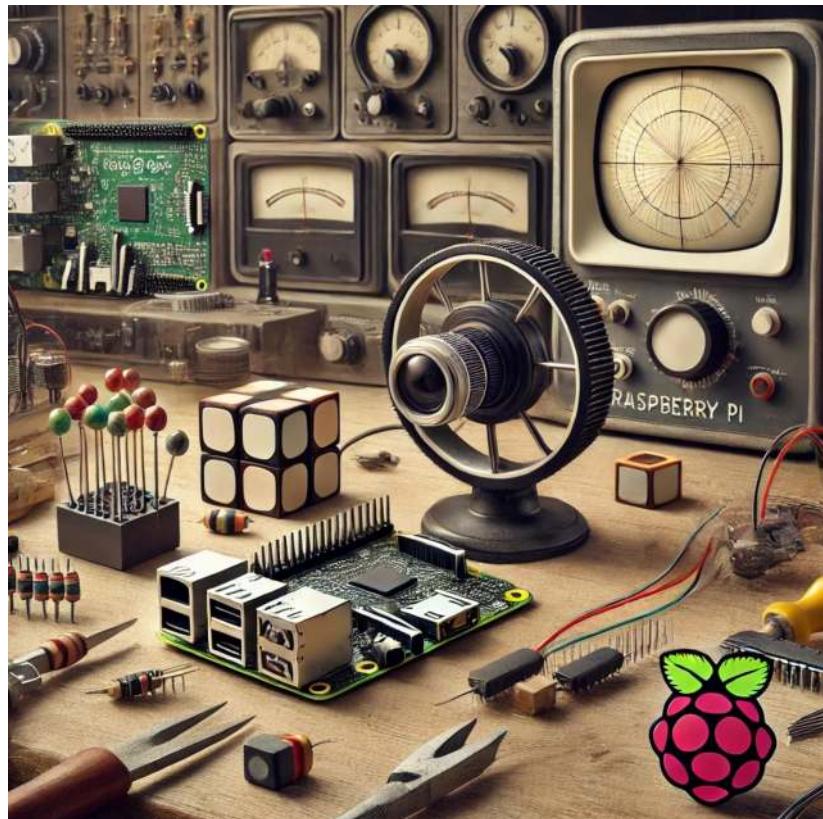


Figure 1.23: DALL-E prompt - A cover image for an ‘Object Detection’ chapter in a Raspberry Pi tutorial, designed in the same vintage 1950s electronics lab style as previous covers. The scene should prominently feature wheels and cubes, similar to those provided by the user, placed on a workbench in the foreground. A Raspberry Pi with a connected camera module should be capturing an image of these objects. Surround the scene with classic lab tools like soldering irons, resistors, and wires. The lab background should include vintage equipment like oscilloscopes and tube radios, maintaining the detailed and nostalgic feel of the era. No text or logos should be included.

Overview

Building upon our exploration of image classification, we now turn our attention to a more advanced computer vision task: object detection. While image classification assigns a single label to an entire image, object detection goes further by identifying and locating multiple objects within a single image. This capability opens up many new applications and challenges, particularly in edge computing and IoT devices like the Raspberry Pi.

Object detection combines the tasks of classification and localization. It not only determines what objects are present in an image but also pinpoints their locations by, for example, drawing bounding boxes around them. This added complexity makes object detection a more powerful tool for understanding visual scenes, but it also requires more sophisticated models and training techniques.

In edge AI, where we work with constrained computational resources, implementing efficient object detection models becomes crucial. The challenges we faced with image classification—balancing model size, inference speed, and accuracy—are amplified in object detection. However, the rewards are also more significant, as object detection enables more nuanced and detailed visual data analysis.

Some applications of object detection on edge devices include:

1. Surveillance and security systems
2. Autonomous vehicles and drones
3. Industrial quality control
4. Wildlife monitoring
5. Augmented reality applications

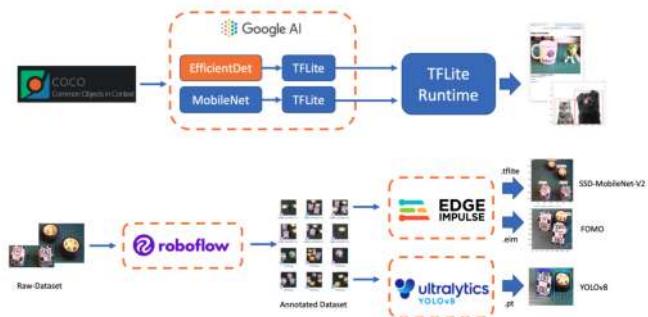
As we put our hands into object detection, we'll build upon the concepts and techniques we explored in image classification. We'll examine popular object detection architectures designed for efficiency, such as:

- Single Stage Detectors, such as MobileNet and EfficientDet,
- FOMO (Faster Objects, More Objects), and
- YOLO (You Only Look Once).

To learn more about object detection models, follow the tutorial [A Gentle Introduction to Object Recognition With Deep Learning](#).

We will explore those object detection models using

- TensorFlow Lite Runtime (now changed to LiteRT),
- Edge Impulse Linux Python SDK and
- Ultralytics



Throughout this lab, we'll cover the fundamentals of object detection and how it differs from image classification. We'll also learn how to train, fine-tune, test, optimize, and deploy popular object detection architectures using a dataset created from scratch.

Object Detection Fundamentals

Object detection builds upon the foundations of image classification but extends its capabilities significantly. To understand object detection, it's crucial first to recognize its key differences from image classification:

Image Classification vs. Object Detection

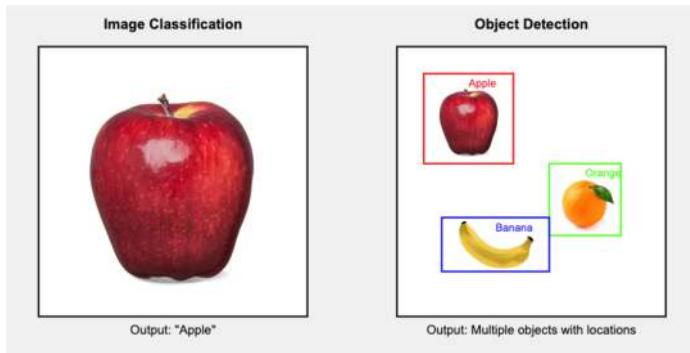
Image Classification:

- Assigns a single label to an entire image
- Answers the question: "What is this image's primary object or scene?"
- Outputs a single class prediction for the whole image

Object Detection:

- Identifies and locates multiple objects within an image
- Answers the questions: "What objects are in this image, and where are they located?"
- Outputs multiple predictions, each consisting of a class label and a bounding box

To visualize this difference, let's consider an example:



This diagram illustrates the critical difference: image classification provides a single label for the entire image, while object detection identifies multiple objects, their classes, and their locations within the image.

Key Components of Object Detection

Object detection systems typically consist of two main components:

1. Object Localization: This component identifies where objects are located in the image. It typically outputs bounding boxes, rectangular regions encompassing each detected object.
2. Object Classification: This component determines the class or category of each detected object, similar to image classification but applied to each localized region.

Challenges in Object Detection

Object detection presents several challenges beyond those of image classification:

- Multiple objects: An image may contain multiple objects of various classes, sizes, and positions.
- Varying scales: Objects can appear at different sizes within the image.
- Occlusion: Objects may be partially hidden or overlapping.
- Background clutter: Distinguishing objects from complex backgrounds can be challenging.
- Real-time performance: Many applications require fast inference times, especially on edge devices.

Approaches to Object Detection

There are two main approaches to object detection:

1. Two-stage detectors: These first propose regions of interest and then classify each region. Examples include R-CNN and its variants (Fast R-CNN, Faster R-CNN).
2. Single-stage detectors: These predict bounding boxes (or centroids) and class probabilities in one forward pass of the network. Examples include YOLO (You Only Look Once), EfficientDet, SSD (Single Shot Detector), and FOMO (Faster Objects, More Objects). These are often faster and more suitable for edge devices like Raspberry Pi.

Evaluation Metrics

Object detection uses different metrics compared to image classification:

- **Intersection over Union (IoU):** Measures the overlap between predicted and ground truth bounding boxes.
- **Mean Average Precision (mAP):** Combines precision and recall across all classes and IoU thresholds.
- **Frames Per Second (FPS):** Measures detection speed, crucial for real-time applications on edge devices.

Pre-Trained Object Detection Models Overview

As we saw in the introduction, given an image or a video stream, an object detection model can identify which of a known set of objects might be present and provide information about their positions within the image.

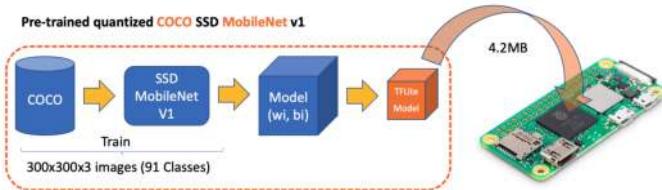
You can test some common models online by visiting Object Detection - MediaPipe Studio

On Kaggle, we can find the most common pre-trained tflite models to use with the Raspi, ssd_mobilenet_v1, and EfficientDet. Those models were trained on the COCO (Common Objects in Context) dataset, with over 200,000 labeled images in 91 categories. Go, download the models, and upload them to the ./models folder in the Raspi.

Alternatively, you can find the models and the COCO labels on GitHub.

For the first part of this lab, we will focus on a pre-trained 300×300 SSD-Mobilenet V1 model and compare it with the 320×320 EfficientDet-lite0, also trained using the COCO 2017 dataset. Both models were converted to a TensorFlow Lite format (4.2 MB for the SSD Mobilenet and 4.6 MB for the EfficientDet).

SSD-Mobilenet V2 or V3 is recommended for transfer learning projects, but once the V1 TFLite model is publicly available, we will use it for this overview.



Setting Up the TFLite Environment

We should confirm the steps done on the last Hands-On Lab, Image Classification, as follows:

- Updating the Raspberry Pi
- Installing Required Libraries
- Setting up a Virtual Environment (Optional but Recommended)

```
source ~/tflite/bin/activate
```

- Installing TensorFlow Lite Runtime
- Installing Additional Python Libraries (inside the environment)

Creating a Working Directory:

Considering that we have created the Documents/TFLITE folder in the last Lab, let's now create the specific folders for this object detection lab:

```
cd Documents/TFLITE/
mkdir OBJ_DETECT
cd OBJ_DETECT
mkdir images
mkdir models
cd models
```

Inference and Post-Processing

Let's start a new notebook to follow all the steps to detect objects on an image:

Import the needed libraries:

```
import time
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import tensorflow.lite_runtime.interpreter as tflite
```

Load the TFLite model and allocate tensors:

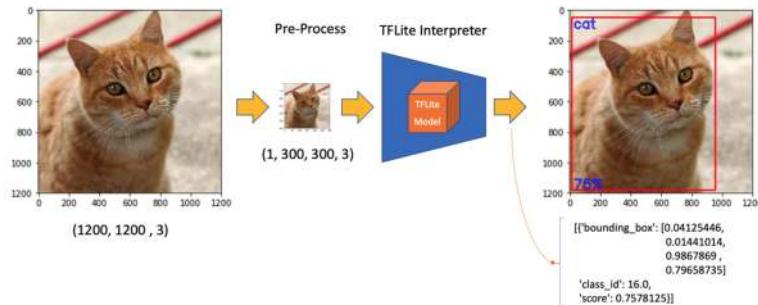
```
model_path = "./models/ssd-mobilenet-v1-tflite-default-v1.tflite"
interpreter = tflite.Interpreter(model_path=model_path)
interpreter.allocate_tensors()
```

Get input and output tensors.

```
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()
```

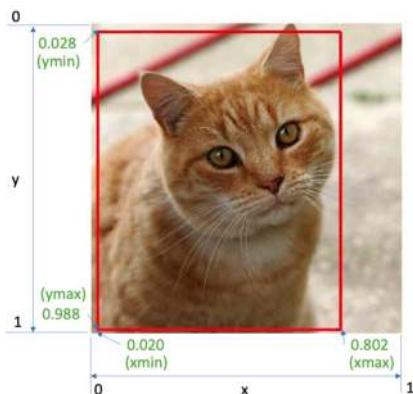
Input details will inform us how the model should be fed with an image. The shape of (1, 300, 300, 3) with a dtype of uint8 tells us that a non-normalized (pixel value range from 0 to 255) image with dimensions $(300 \times 300 \times 3)$ should be input one by one (Batch Dimension: 1).

The **output details** include not only the labels ("classes") and probabilities ("scores") but also the relative window position of the bounding boxes ("boxes") about where the object is located on the image and the number of detected objects ("num_detections"). The output details also tell us that the model can detect a **maximum of 10 objects** in the image.



So, for the above example, using the same cat image used with the *Image Classification Lab* looking for the output, we have a **76% probability** of having found an object with a **class ID of 16** on an area delimited by a **bounding box of [0.028011084, 0.020121813, 0.9886069, 0.802299]**. Those four numbers are related to ymin, xmin, ymax and xmax, the box coordinates.

Taking into consideration that y goes from the top (ymin) to the bottom (ymax) and x goes from left (xmin) to the right (xmax), we have, in fact, the coordinates of the top/left corner and the bottom/right one. With both edges and knowing the shape of the picture, it is possible to draw a rectangle around the object, as shown in the figure below:

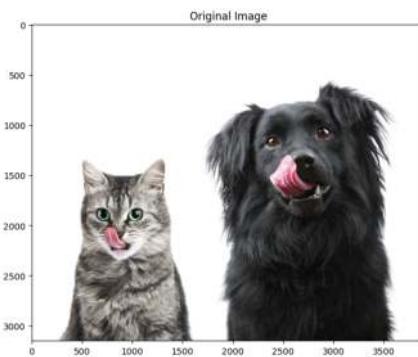


Next, we should find what class ID equal to 16 means. Opening the file `coco_labels.txt`, as a list, each element has an associated index, and inspecting index 16, we get, as expected, cat. The probability is the value returning from the score.

Let's now upload some images with multiple objects on it for testing.

```
img_path = "./images/cat_dog.jpeg"
orig_img = Image.open(img_path)

# Display the image
plt.figure(figsize=(8, 8))
plt.imshow(orig_img)
plt.title("Original Image")
plt.show()
```



Based on the input details, let's pre-process the image, changing its shape and expanding its dimension:

```
img = orig_img.resize(
    (input_details[0]["shape"][1], input_details[0]["shape"][2])
)
input_data = np.expand_dims(img, axis=0)
input_data.shape, input_data.dtype
```

The new input_data shape is (1, 300, 300, 3) with a dtype of uint8, which is compatible with what the model expects.

Using the input_data, let's run the interpreter, measure the latency, and get the output:

```
start_time = time.time()
interpreter.set_tensor(input_details[0]["index"], input_data)
interpreter.invoke()
end_time = time.time()
inference_time = (
    end_time - start_time
```

```
) * 1000 # Convert to milliseconds
print("Inference time: {:.1f}ms".format(inference_time))
```

With a latency of around 800 ms, we can get 4 distinct outputs:

```
boxes = interpreter.get_tensor(output_details[0]["index"])[0]
classes = interpreter.get_tensor(output_details[1]["index"])[0]
scores = interpreter.get_tensor(output_details[2]["index"])[0]
num_detections = int(
    interpreter.get_tensor(output_details[3]["index"])[0]
)
```

On a quick inspection, we can see that the model detected 2 objects with a score over 0.5:

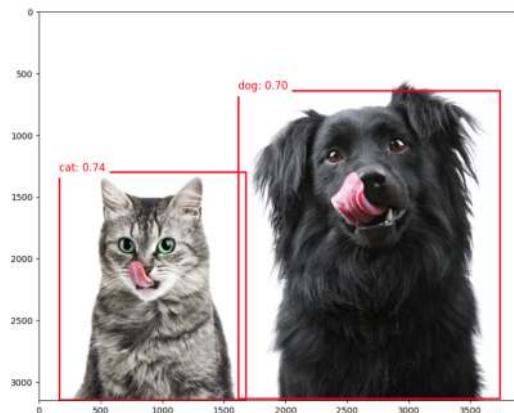
```
for i in range(num_detections):
    if scores[i] > 0.5: # Confidence threshold
        print(f"Object {i}:")
        print(f"  Bounding Box: {boxes[i]}")
        print(f"  Confidence: {scores[i]}")
        print(f"  Class: {classes[i]}")

Object 0:
  Bounding Box: [0.4125163  0.04130688  0.997076   0.42888364]
  Confidence: 0.73828125
  Class: 16.0
Object 1:
  Bounding Box: [0.20249811 0.41268167 0.99390197 0.95425284]
  Confidence: 0.69921875
  Class: 17.0
```

And we can also visualize the results:

```
plt.figure(figsize=(12, 8))
plt.imshow(orig_img)
for i in range(num_detections):
    if scores[i] > 0.5: # Adjust threshold as needed
        ymin, xmin, ymax, xmax = boxes[i]
        (left, right, top, bottom) = (
            xmin * orig_img.width,
            xmax * orig_img.width,
            ymin * orig_img.height,
            ymax * orig_img.height,
        )
        rect = plt.Rectangle(
            (left, top),
            right - left,
```

```
        bottom - top,
        fill=False,
        color="red",
        linewidth=2,
    )
    plt.gca().add_patch(rect)
    class_id = int(classes[i])
    class_name = labels[class_id]
    plt.text(
        left,
        top - 10,
        f"{class_name}: {scores[i]:.2f}",
        color="red",
        fontsize=12,
        backgroundcolor="white",
    )
)
```



EfficientDet

EfficientDet is not technically an SSD (Single Shot Detector) model, but it shares some similarities and builds upon ideas from SSD and other object detection architectures:

1. EfficientDet:

- Developed by Google researchers in 2019
- Uses EfficientNet as the backbone network
- Employs a novel bi-directional feature pyramid network (BiFPN)

- It uses compound scaling to scale the backbone network and the object detection components efficiently.
2. Similarities to SSD:
 - Both are single-stage detectors, meaning they perform object localization and classification in a single forward pass.
 - Both use multi-scale feature maps to detect objects at different scales.
 3. Key differences:
 - Backbone: SSD typically uses VGG or MobileNet, while EfficientDet uses EfficientNet.
 - Feature fusion: SSD uses a simple feature pyramid, while EfficientDet uses the more advanced BiFPN.
 - Scaling method: EfficientDet introduces compound scaling for all components of the network
 4. Advantages of EfficientDet:
 - Generally achieves better accuracy-efficiency trade-offs than SSD and many other object detection models.
 - More flexible scaling allows for a family of models with different size-performance trade-offs.

While EfficientDet is not an SSD model, it can be seen as an evolution of single-stage detection architectures, incorporating more advanced techniques to improve efficiency and accuracy. When using EfficientDet, we can expect similar output structures to SSD (e.g., bounding boxes and class scores).

On GitHub, you can find another notebook exploring the EfficientDet model that we did with SSD MobileNet.

Object Detection Project

Now, we will develop a complete Image Classification project from data collection to training and deployment. As we did with the Image Classification project, the trained and converted model will be used for inference.

We will use the same dataset to train 3 models: SSD-MobileNet V2, FOMO, and YOLO.

The Goal

All Machine Learning projects need to start with a 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

Raw Data Collection

Once we have defined our Machine Learning project goal, the next and most crucial step is collecting the dataset. We can use a phone, the Raspi, or a mix to create the raw dataset (with no labels). Let's use the simple web app on our Raspberry Pi to view the QVGA (320 x 240) captured images in a browser.

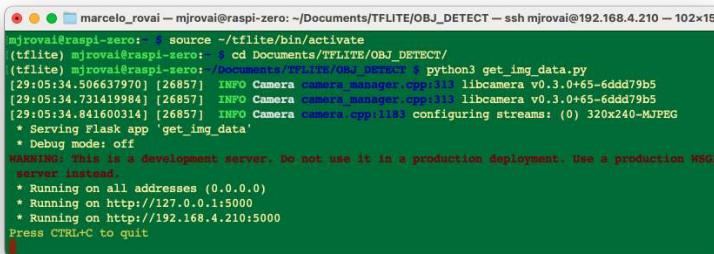
From GitHub, get the Python script `get_img_data.py` and open it in the terminal:

```
python3 get_img_data.py
```

Access the web interface:

- On the Raspberry Pi itself (if you have a GUI): Open a web browser and go to `http://localhost:5000`

- From another device on the same network: Open a web browser and go to `http://<raspberry_pi_ip>:5000` (Replace `<raspberry_pi_ip>` with your Raspberry Pi's IP address). For example:
`http://192.168.4.210:5000/`



```
mjrovai@raspi-zero: ~$ source ~/tf-lite/bin/activate
(mjrovai@raspi-zero: ~$ cd Documents/TFLITE/OBJ_DETECT/
(mjrovai@raspi-zero: ~/Documents/TFLITE/OBJ_DETECT$ python3 get_img_data.py
[29:05:34.506637970] [26857] INFO Camera camera_manager.cpp:113 libcamera v0.3.0+65-6ddd79b5
[29:05:34.731419984] [26857] INFO Camera camera_manager.cpp:113 libcamera v0.3.0+65-6ddd79b5
[29:05:34.841600314] [26857] INFO Camera camera.cpp:1183 configuring streams: (0) 320x240-MJPEG
* Serving Flask app 'get_img_data'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI
server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://192.168.4.210:5000
Press CTRL+C to quit
```

The Python script creates a web-based interface for capturing and organizing image datasets using a Raspberry Pi and its camera. It's handy for machine learning projects that require labeled image data or not, as in our case here.

Access the web interface from a browser, enter a generic label for the images you want to capture, and press Start Capture.



Note that the images to be captured will have multiple labels that should be defined later.

Use the live preview to position the camera and click Capture Image to save images under the current label (in this case, `box-wheel`).



When we have enough images, we can press Stop Capture. The captured images are saved on the folder dataset/box-wheel:

```
(tfile) miroval@raspi-server:~/Documents/TFLITE/OBJ_DETECT/dataset -- ssh miroval@192.168.4.210 -t 102x11
ls
ls: cannot access 'image_20240903-224511.jpg': No such file or directory
ls: cannot access 'image_20240903-224512.jpg': No such file or directory
ls: cannot access 'image_20240903-224513.jpg': No such file or directory
ls: cannot access 'image_20240903-224514.jpg': No such file or directory
ls: cannot access 'image_20240903-224515.jpg': No such file or directory
ls: cannot access 'image_20240903-224516.jpg': No such file or directory
ls: cannot access 'image_20240903-224517.jpg': No such file or directory
ls: cannot access 'image_20240903-224518.jpg': No such file or directory
ls: cannot access 'image_20240903-224519.jpg': No such file or directory
ls: cannot access 'image_20240903-224520.jpg': No such file or directory
ls: cannot access 'image_20240903-224521.jpg': No such file or directory
ls: cannot access 'image_20240903-224522.jpg': No such file or directory
ls: cannot access 'image_20240903-224523.jpg': No such file or directory
ls: cannot access 'image_20240903-224524.jpg': No such file or directory
ls: cannot access 'image_20240903-224525.jpg': No such file or directory
ls: cannot access 'image_20240903-224526.jpg': No such file or directory
ls: cannot access 'image_20240903-224527.jpg': No such file or directory
ls: cannot access 'image_20240903-224528.jpg': No such file or directory
ls: cannot access 'image_20240903-224529.jpg': No such file or directory
ls: cannot access 'image_20240903-224530.jpg': No such file or directory
ls: cannot access 'image_20240903-224531.jpg': No such file or directory
ls: cannot access 'image_20240903-224532.jpg': No such file or directory
(tfile) miroval@raspi-server:~/Documents/TFLITE/OBJ_DETECT/dataset -
```

Get around 60 images. Try to capture different angles, backgrounds, and light conditions. Filezilla can transfer the created raw dataset to your main computer.

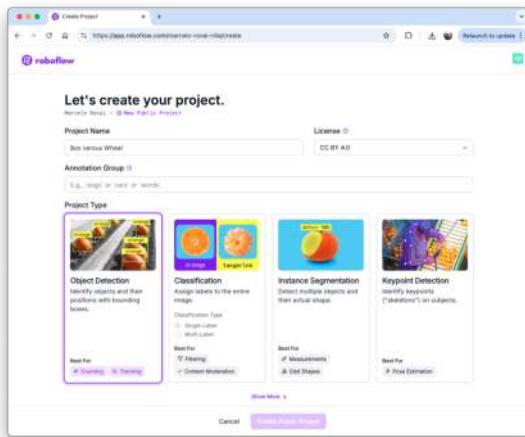
Labeling Data

The next step in an Object Detect project is to create a labeled dataset. We should label the raw dataset images, creating bounding boxes around

each picture's objects (box and wheel). We can use labeling tools like LabelImg, CVAT, Roboflow, or even the Edge Impulse Studio. Once we have explored the Edge Impulse tool in other labs, let's use Roboflow here.

We are using Roboflow (free version) here for two main reasons. 1) We can have auto-labeler, and 2) The annotated dataset is available in several formats and can be used both on Edge Impulse Studio (we will use it for MobileNet V2 and FOMO train) and on CoLab (YOLOv8 train), for example. Having the annotated dataset on Edge Impulse (Free account), it is not possible to use it for training on other platforms.

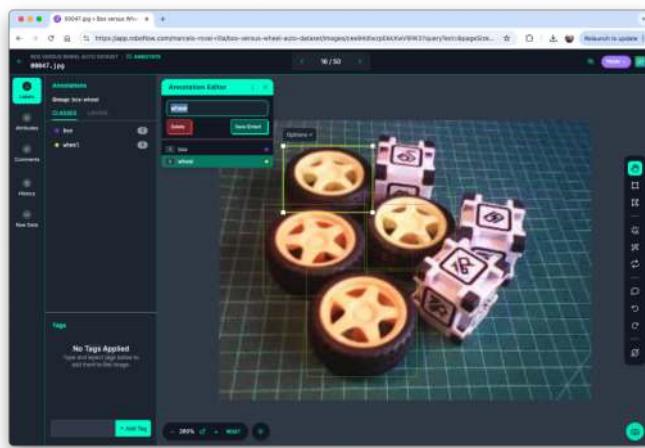
We should upload the raw dataset to Roboflow. Create a free account there and start a new project, for example, ("box-versus-wheel").



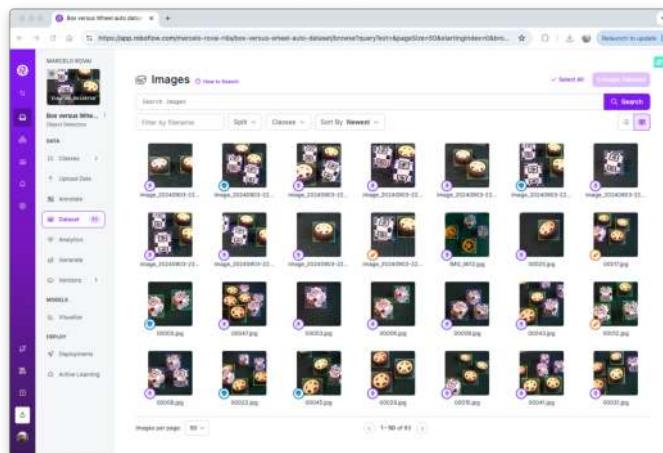
We will not enter in deep details about the Roboflow process once many tutorials are available.

Annotate

Once the project is created and the dataset is uploaded, you should make the annotations using the "Auto-Label" Tool. Note that you can also upload images with only a background, which should be saved w/o any annotations.



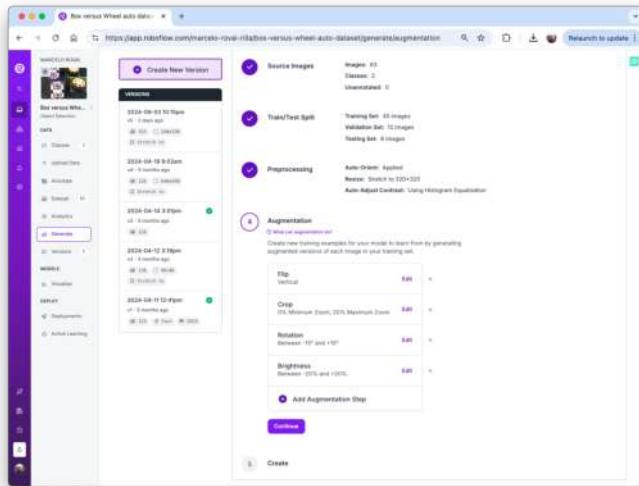
Once all images are annotated, you should split them into training, validation, and testing.



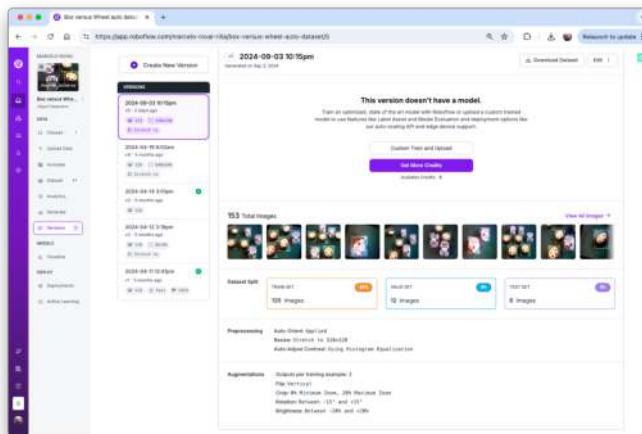
Data Pre-Processing

The last step with the dataset is preprocessing to generate a final version for training. Let's resize all images to 320×320 and generate augmented versions of each image (augmentation) to create new training examples from which our model can learn.

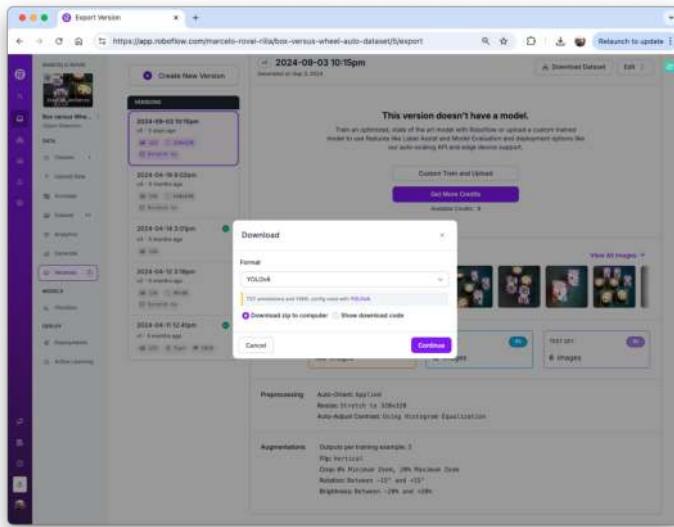
For augmentation, we will rotate the images ($+/-15^\circ$), crop, and vary the brightness and exposure.



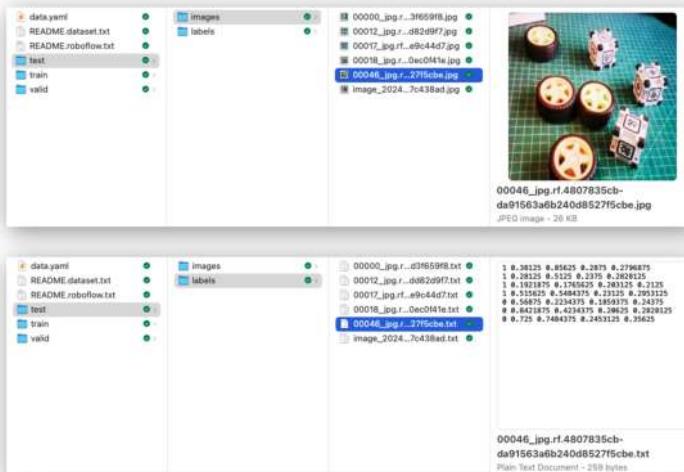
At the end of the process, we will have 153 images.



Now, you should export the annotated dataset in a format that Edge Impulse, Ultralytics, and other frameworks/tools understand, for example, YOLOv8. Let's download a zipped version of the dataset to our desktop.



Here, it is possible to review how the dataset was structured



There are 3 separate folders, one for each split (train/test/valid). For each of them, there are 2 subfolders, images, and labels. The pictures are stored as **image_id.jpg** and **image_id.txt**, where "image_id" is unique for every picture.

The labels file format will be `class_id bounding box coordinates`, where in our case, `class_id` will be 0 for `box` and 1 for `wheel`. The numerical id (0, 1, 2...) will follow the alphabetical order of the class name.

The `data.yaml` file has info about the dataset as the classes' names (`names: ['box', 'wheel']`) following the YOLO format.

And that's it! We are ready to start training using the Edge Impulse Studio (as we will do in the following step), Ultralytics (as we will when discussing YOLO), or even training from scratch on CoLab (as we did with the Cifar-10 dataset on the Image Classification lab).

The pre-processed dataset can be found at the Roboflow site.

Training an SSD MobileNet Model on Edge Impulse Studio

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

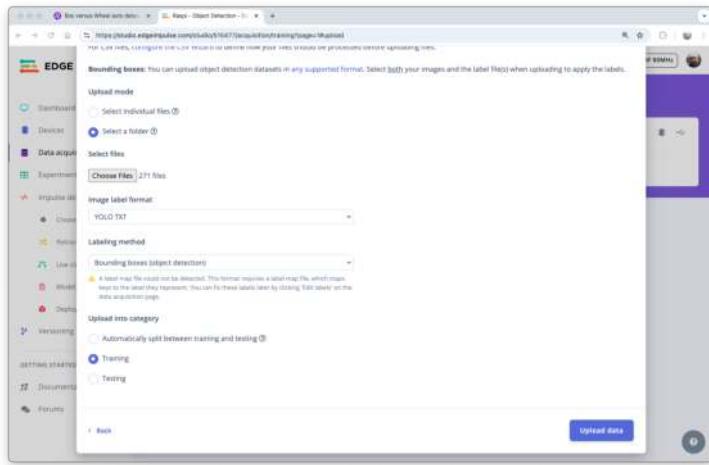
Here, you can clone the project developed for this hands-on lab: Raspi - Object Detection.

On the Project Dashboard tab, go down and on **Project info**, and for Labeling method select **Bounding boxes (object detection)**

Uploading the annotated data

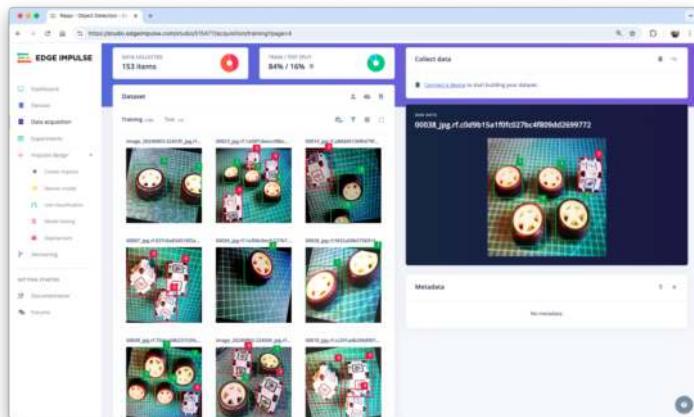
On Studio, go to the Data acquisition tab, and on the UPLOAD DATA section, upload from your computer the raw dataset.

We can use the option **Select a folder**, choosing, for example, the folder `train` in your computer, which contains two sub-folders, `images`, and `labels`. Select the Image label format, "YOLO TXT", upload into the category **Training**, and press **Upload data**.



Repeat the process for the test data (upload both folders, test, and validation). At the end of the upload process, you should end with the annotated dataset of 153 images split in the train/test (84%/16%).

Note that labels will be stored at the labels files 0 and 1 , which are equivalent to box and wheel.

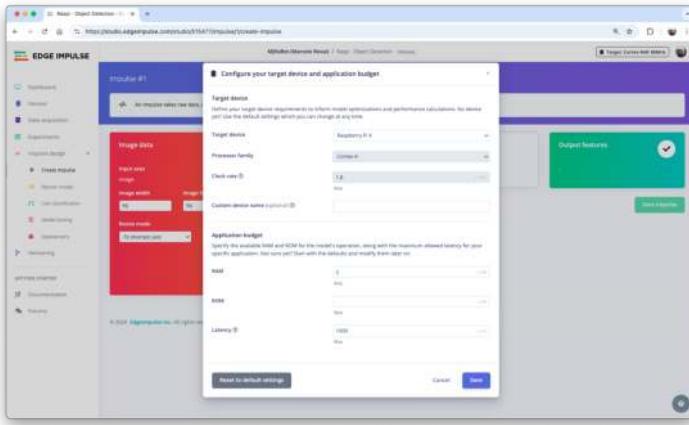


The Impulse Design

The first thing to define when we enter the Create impulse step is to describe the target device for deployment. A pop-up window will

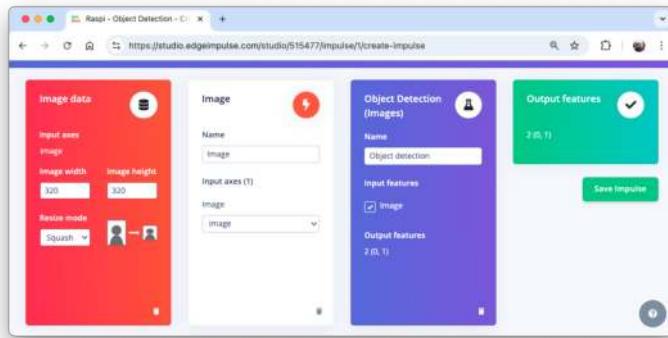
appear. We will select Raspberry 4, an intermediary device between the Raspi-Zero and the Raspi-5.

This choice will not interfere with the training; it will only give us an idea about the latency of the model on that specific target.



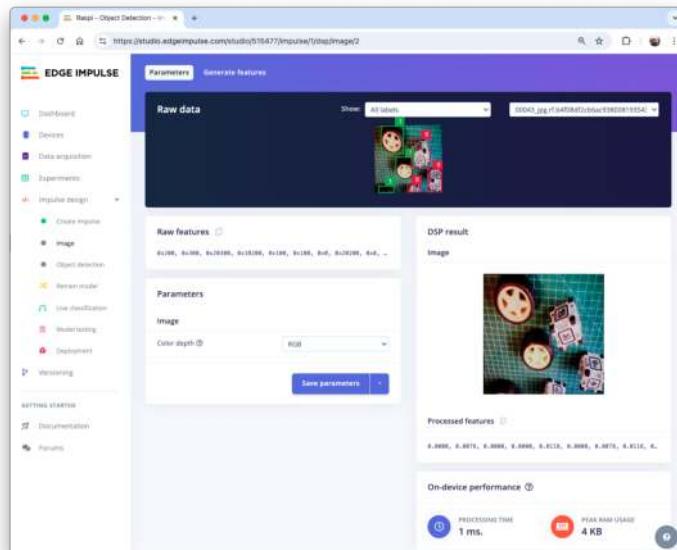
In this phase, you should define how to:

- **Pre-processing** consists of resizing the individual images. In our case, the images were pre-processed on Roboflow, to 320x320 , so let's keep it. The resize will not matter here because the images are already squared. If you upload a rectangular image, squash it (squared form, without cropping). Afterward, you could define if the images are converted from RGB to Grayscale or not.
- **Design a Model**, in this case, “Object Detection.”

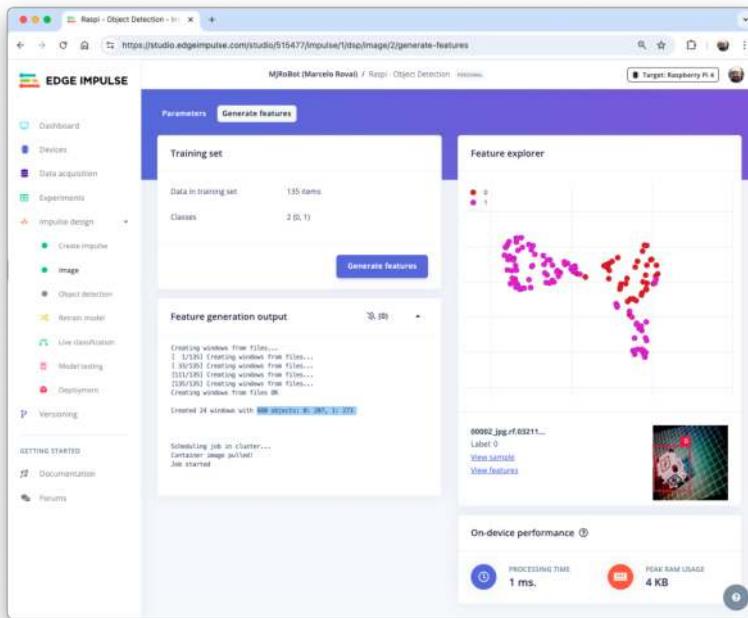


Preprocessing all dataset

In the section **Image**, select **Color depth** as **RGB**, and press **Save parameters**.



The Studio moves automatically to the next section, **Generate features**, where all samples will be pre-processed, resulting in 480 objects: 207 boxes and 273 wheels.



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

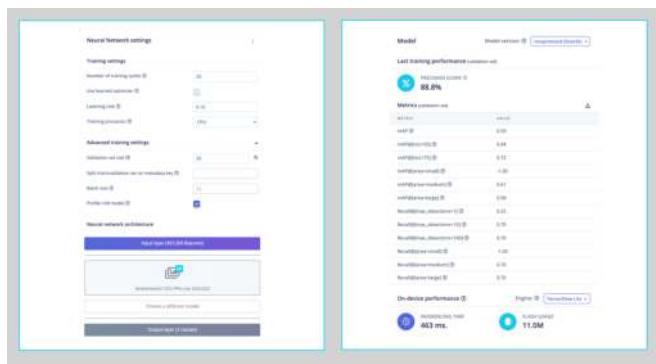
Model Design, Training, and Test

For training, we should select a pre-trained model. Let's use the **MobileNetV2 SSD FPN-Lite (320x320 only)**. It is a pre-trained object detection model designed to locate up to 10 objects within an image, outputting a bounding box for each object detected. The model is around 3.7 MB in size. It supports an RGB input at 320×320 px.

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

- Epochs: 25
- Batch size: 32
- Learning Rate: 0.15.

For validation during training, 20% of the dataset (*validation_dataset*) will be spared.



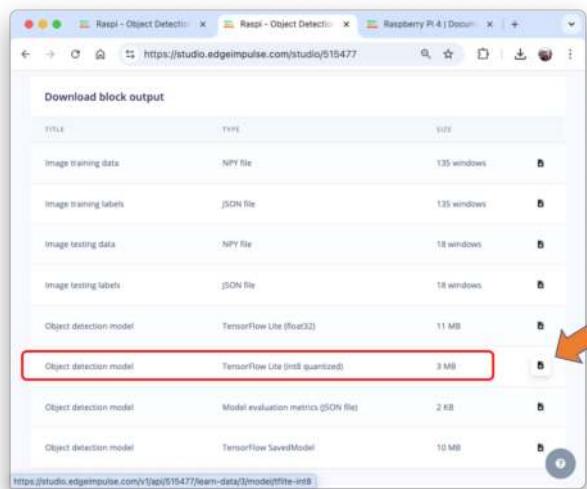
As a result, the model ends with an overall precision score (based on COCO mAP) of 88.8%, higher than the result when using the test data (83.3%).

Deploying the model

We have two ways to deploy our model:

- **TFLite model**, which lets deploy the trained model as `.tflite` for the Raspi to run it using Python.
- **Linux (AARCH64)**, a binary for Linux (AARCH64), implements the Edge Impulse Linux protocol, which lets us run our models on any Linux-based development board, with SDKs for Python, for example. See the documentation for more information and setup instructions.

Let's deploy the **TFLite model**. On the Dashboard tab, go to Transfer learning model (int8 quantized) and click on the download icon:



Transfer the model from your computer to the Raspi folder `./models` and capture or get some images for inference and save them in the folder `./images`.

Inference and Post-Processing

The inference can be made as discussed in the *Pre-Trained Object Detection Models Overview*. Let's start a new notebook to follow all the steps to detect cubes and wheels on an image.

Import the needed libraries:

```
import time
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from PIL import Image
import tensorflow as tf
```

Define the model path and labels:

```
model_path = "./models/ei-raspi-object-detection-SSD-\n        MobileNetv2-320x320-int8.lite"
labels = ["box", "wheel"]
```

Remember that the model will output the class ID as values (0 and 1), following an alphabetic order regarding the class names.

Load the model, allocate the tensors, and get the input and output tensor details:

```
# Load the TFLite model
interpreter = tflite.Interpreter(model_path=model_path)
interpreter.allocate_tensors()

# Get input and output tensors
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()
```

One crucial difference to note is that the `dtype` of the input details of the model is now `int8`, which means that the input values go from -128 to +127, while each pixel of our raw image goes from 0 to 256. This means that we should pre-process the image to match it. We can check here:

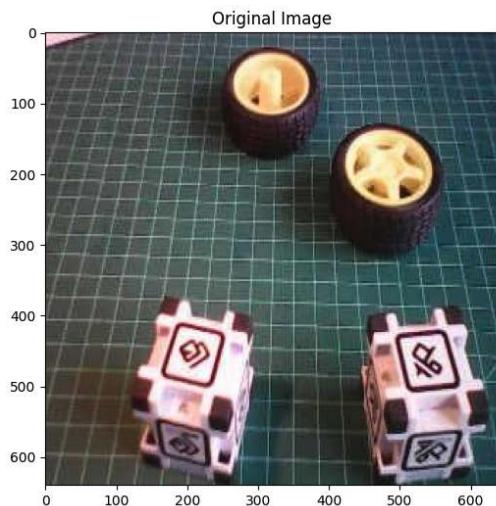
```
input_dtype = input_details[0] ["dtype"]
input_dtype

numpy.int8
```

So, let's open the image and show it:

```
# Load the image
img_path = "./images/box_2_wheel_2.jpg"
orig_img = Image.open(img_path)

# Display the image
plt.figure(figsize=(6, 6))
plt.imshow(orig_img)
plt.title("Original Image")
plt.show()
```



And perform the pre-processing:

```
scale, zero_point = input_details[0]["quantization"]
img = orig_img.resize(
    (input_details[0]["shape"][1], input_details[0]["shape"][2]))
img_array = np.array(img, dtype=np.float32) / 255.0
img_array = (
    (img_array / scale + zero_point).clip(-128, 127).astype(np.int8))
input_data = np.expand_dims(img_array, axis=0)
```

Checking the input data, we can verify that the input tensor is compatible with what is expected by the model:

```
input_data.shape, input_data.dtype
```

```
((1, 320, 320, 3), dtype('int8'))
```

Now, it is time to perform the inference. Let's also calculate the latency of the model:

```
# Inference on Raspi-Zero
start_time = time.time()
interpreter.set_tensor(input_details[0]["index"], input_data)
interpreter.invoke()
end_time = time.time()
```

```
inference_time = (
    end_time - start_time
) * 1000 # Convert to milliseconds
print("Inference time: {:.1f}ms".format(inference_time))
```

The model will take around 600ms to perform the inference in the Raspi-Zero, which is around 5 times longer than a Raspi-5.

Now, we can get the output classes of objects detected, its bounding boxes coordinates, and probabilities.

```
boxes = interpreter.get_tensor(output_details[1]["index"])[0]
classes = interpreter.get_tensor(output_details[3]["index"])[0]
scores = interpreter.get_tensor(output_details[0]["index"])[0]
num_detections = int(
    interpreter.get_tensor(output_details[2]["index"])[0]
)

for i in range(num_detections):
    if scores[i] > 0.5: # Confidence threshold
        print(f"Object {i}:")
        print(f"  Bounding Box: {boxes[i]}")
        print(f"  Confidence: {scores[i]}")
        print(f"  Class: {classes[i]}")

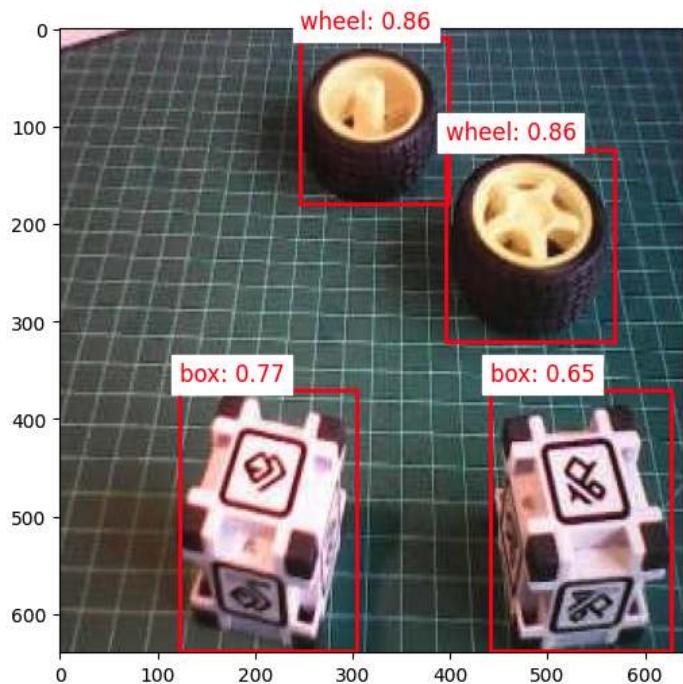
Object 0:
  Bounding Box: [0.01461247 0.38439587 0.2793928 0.62159896]
  Confidence: 0.86328125
  Class: 1.0
Object 1:
  Bounding Box: [0.19234724 0.6176628 0.5012042 0.888332 ]
  Confidence: 0.86328125
  Class: 1.0
Object 2:
  Bounding Box: [0.5792029 0.19102246 0.9971932 0.47538966]
  Confidence: 0.7734375
  Class: 0.0
Object 3:
  Bounding Box: [0.5792029 0.68904555 0.9971932 0.97973716]
  Confidence: 0.6484375
  Class: 0.0
```

From the results, we can see that 4 objects were detected: two with class ID 0 (box) and two with class ID 1 (wheel), what is correct!

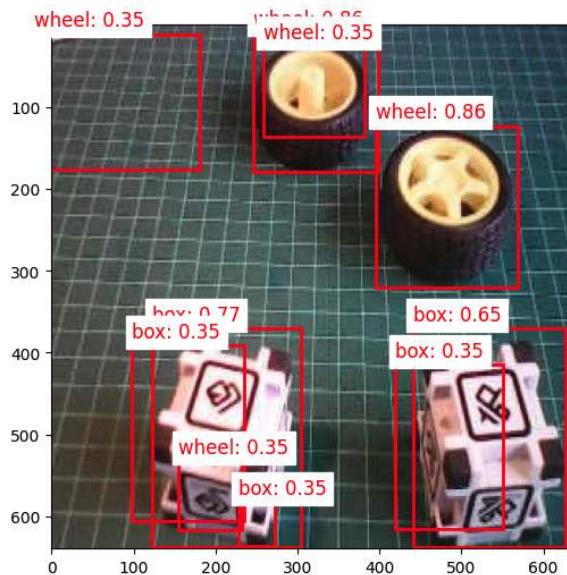
Let's visualize the result for a threshold of 0.5

```
threshold = 0.5
plt.figure(figsize=(6, 6))
plt.imshow(orig_img)
for i in range(num_detections):
    if scores[i] > threshold:
        ymin, xmin, ymax, xmax = boxes[i]
        (left, right, top, bottom) = (
            xmin * orig_img.width,
            xmax * orig_img.width,
            ymin * orig_img.height,
            ymax * orig_img.height,
        )
        rect = plt.Rectangle(
            (left, top),
            right - left,
            bottom - top,
            fill=False,
            color="red",
            linewidth=2,
        )
        plt.gca().add_patch(rect)
        class_id = int(classes[i])
        class_name = labels[class_id]
        plt.text(
            left,
            top - 10,
            f"{class_name}: {scores[i]:.2f}",
            color="red",
            fontsize=12,
            backgroundcolor="white",
        )

```



But what happens if we reduce the threshold to 0.3, for example?



We start to see false positives and **multiple detections**, where the model detects the same object multiple times with different confidence levels and slightly different bounding boxes.

Commonly, sometimes, we need to adjust the threshold to smaller values to capture all objects, avoiding false negatives, which would lead to multiple detections.

To improve the detection results, we should implement **Non-Maximum Suppression (NMS)**, which helps eliminate overlapping bounding boxes and keeps only the most confident detection.

For that, let's create a general function named `non_max_suppression()`, with the role of refining object detection results by eliminating redundant and overlapping bounding boxes. It achieves this by iteratively selecting the detection with the highest confidence score and removing other significantly overlapping detections based on an Intersection over Union (IoU) threshold.

```
def non_max_suppression(boxes, scores, threshold):
    # Convert to corner coordinates
    x1 = boxes[:, 0]
    y1 = boxes[:, 1]
    x2 = boxes[:, 2]
    y2 = boxes[:, 3]
```

```
areas = (x2 - x1 + 1) * (y2 - y1 + 1)
order = scores.argsort() [::-1]

keep = []
while order.size > 0:
    i = order[0]
    keep.append(i)
    xx1 = np.maximum(x1[i], x1[order[1:]])
    yy1 = np.maximum(y1[i], y1[order[1:]])
    xx2 = np.minimum(x2[i], x2[order[1:]])
    yy2 = np.minimum(y2[i], y2[order[1:]])

    w = np.maximum(0.0, xx2 - xx1 + 1)
    h = np.maximum(0.0, yy2 - yy1 + 1)
    inter = w * h
    ovr = inter / (areas[i] + areas[order[1:]] - inter)

    inds = np.where.ovr <= threshold)[0]
    order = order[inds + 1]

return keep
```

How it works:

1. Sorting: It starts by sorting all detections by their confidence scores, highest to lowest.
2. Selection: It selects the highest-scoring box and adds it to the final list of detections.
3. Comparison: This selected box is compared with all remaining lower-scoring boxes.
4. Elimination: Any box that overlaps significantly (above the IoU threshold) with the selected box is eliminated.
5. Iteration: This process repeats with the next highest-scoring box until all boxes are processed.

Now, we can define a more precise visualization function that will take into consideration an IoU threshold, detecting only the objects that were selected by the `non_max_suppression` function:

```
def visualize_detections(
    image, boxes, classes, scores, labels, threshold, iou_threshold
):
    if isinstance(image, Image.Image):
```

```
    image_np = np.array(image)
else:
    image_np = image
height, width = image_np.shape[:2]
# Convert normalized coordinates to pixel coordinates
boxes_pixel = boxes * np.array([height, width, height, width])
# Apply NMS
keep = non_max_suppression(boxes_pixel, scores, iou_threshold)
# Set the figure size to 12x8 inches
fig, ax = plt.subplots(1, figsize=(12, 8))
ax.imshow(image_np)
for i in keep:
    if scores[i] > threshold:
        ymin, xmin, ymax, xmax = boxes[i]
        rect = patches.Rectangle(
            (xmin * width, ymin * height),
            (xmax - xmin) * width,
            (ymax - ymin) * height,
            linewidth=2,
            edgecolor="r",
            facecolor="none",
        )
        ax.add_patch(rect)
        class_name = labels[int(classes[i])]
        ax.text(
            xmin * width,
            ymin * height - 10,
            f"{class_name}: {scores[i]:.2f}",
            color="red",
            fontsize=12,
            backgroundcolor="white",
        )
plt.show()
```

Now we can create a function that will call the others, performing inference on any image:

```
def detect_objects(img_path, conf=0.5, iou=0.5):
    orig_img = Image.open(img_path)
    scale, zero_point = input_details[0]["quantization"]
    img = orig_img.resize(
        (input_details[0]["shape"][1], input_details[0]["shape"][2])
    )
```

```
img_array = np.array(img, dtype=np.float32) / 255.0
img_array = (
    (img_array / scale + zero_point)
    .clip(-128, 127)
    .astype(np.int8)
)
input_data = np.expand_dims(img_array, axis=0)

# Inference on Raspi-Zero
start_time = time.time()
interpreter.set_tensor(input_details[0]["index"], input_data)
interpreter.invoke()
end_time = time.time()
inference_time = (
    end_time - start_time
) * 1000 # Convert to milliseconds

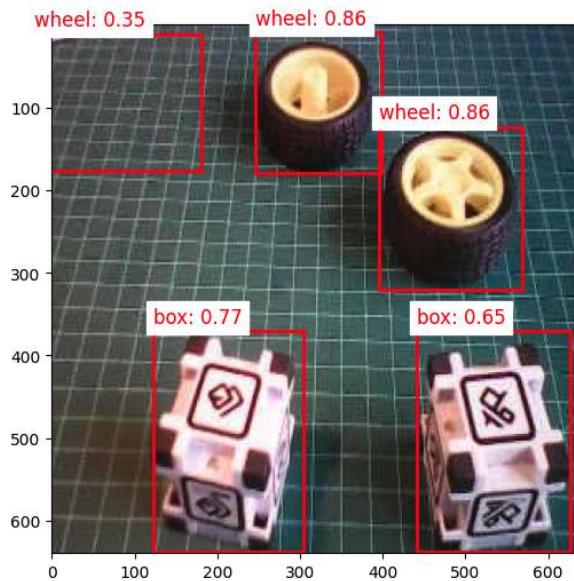
print("Inference time: {:.1f}ms".format(inference_time))

# Extract the outputs
boxes = interpreter.get_tensor(output_details[1]["index"])[0]
classes = interpreter.get_tensor(output_details[3]["index"])[0]
scores = interpreter.get_tensor(output_details[0]["index"])[0]
num_detections = int(
    interpreter.get_tensor(output_details[2]["index"])[0]
)

visualize_detections(
    orig_img,
    boxes,
    classes,
    scores,
    labels,
    threshold=conf,
    iou_threshold=iou,
)
```

Now, running the code, having the same image again with a confidence threshold of 0.3, but with a small IoU:

```
img_path = "./images/box_2_wheel_2.jpg"
detect_objects(img_path, conf=0.3, iou=0.05)
```



Training a FOMO Model at Edge Impulse Studio

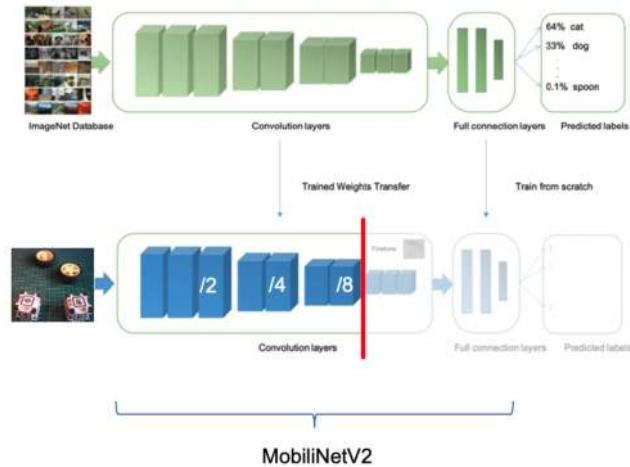
The inference with the SSD MobileNet model worked well, but the latency was significantly high. The inference varied from 0.5 to 1.3 seconds on a Raspi-Zero, which means around or less than 1 FPS (1 frame per second). One alternative to speed up the process is to use FOMO (Faster Objects, More Objects).

This novel machine learning algorithm lets us count multiple objects and find their location in an image in real-time using up to $30\times$ less processing power and memory than MobileNet SSD or YOLO. The main reason this is possible is that while other models calculate the object's size by drawing a square around it (bounding box), FOMO ignores the size of the image, providing only the information about where the object is located in the image through its centroid coordinates.

How FOMO works?

In a typical object detection pipeline, the first stage is extracting features from the input image. **FOMO leverages MobileNetV2 to perform this task.** MobileNetV2 processes the input image to produce a feature map

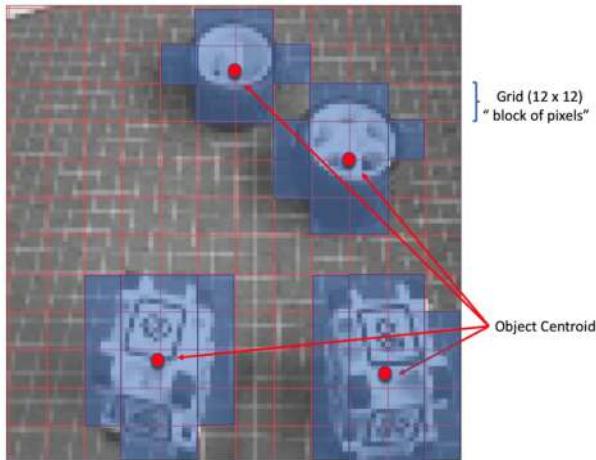
that captures essential characteristics, such as textures, shapes, and object edges, in a computationally efficient way.



Once these features are extracted, FOMO's simpler architecture, focused on center-point detection, interprets the feature map to determine where objects are located in the image. The output is a grid of cells, where each cell represents whether or not an object center is detected. The model outputs one or more confidence scores for each cell, indicating the likelihood of an object being present.

Let's see how it works on an image.

FOMO divides the image into blocks of pixels using a factor of 8. For the input of 96×96 , the grid would be 12×12 ($96/8 = 12$). For a 160×160 , the grid will be 20×20 , and so on. 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.

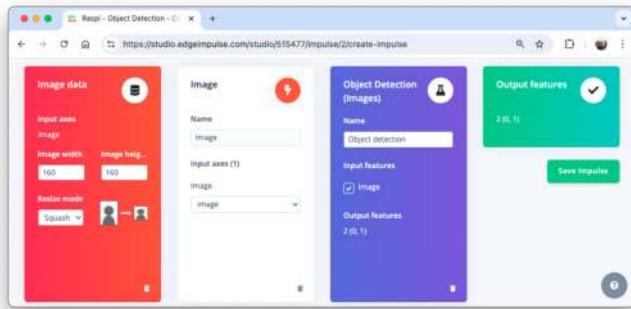


Trade-off Between Speed and Precision:

- **Grid Resolution:** FOMO uses a grid of fixed resolution, meaning each cell can detect if an object is present in that part of the image. While it doesn't provide high localization accuracy, it makes a trade-off by being fast and computationally light, which is crucial for edge devices.
- **Multi-Object Detection:** Since each cell is independent, FOMO can detect multiple objects simultaneously in an image by identifying multiple centers.

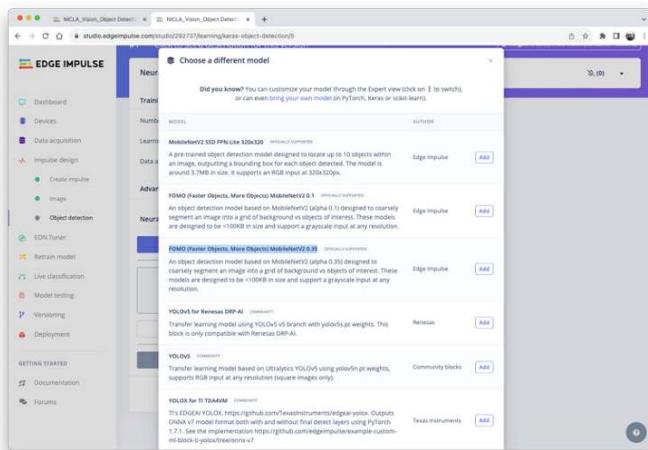
Impulse Design, new Training and Testing

Return to Edge Impulse Studio, and in the Experiments tab, create another impulse. Now, the input images should be 160×160 (this is the expected input size for MobilenetV2).



On the **Image** tab, generate the features and go to the **Object detection** tab.

We should select a pre-trained model for training. Let's use the **FOMO (Faster Objects, More Objects) MobileNetV2 0.35**.

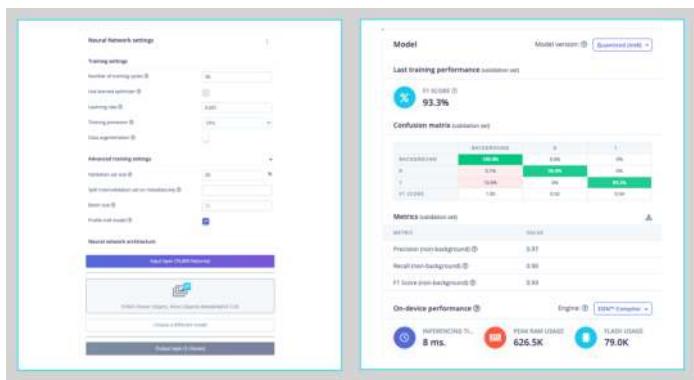


Regarding the training hyper-parameters, the model will be trained with:

- Epochs: 30
- Batch size: 32
- Learning Rate: 0.001.

For validation during training, 20% of the dataset (*validation_dataset*) will be spared. We will not apply Data Augmentation for the remaining 80% (*train_dataset*) because our dataset was already augmented during the labeling phase at Roboflow.

As a result, the model ends with an overall F1 score of 93.3% with an impressive latency of 8 ms (Raspi-4), around 60 \times less than we got with the SSD MobileNetV2.



Note that FOMO automatically added a third label background to the two previously defined *boxes* (0) and *wheels* (1).

On the Model testing tab, we can see that the accuracy was 94%. Here is one of the test sample results:



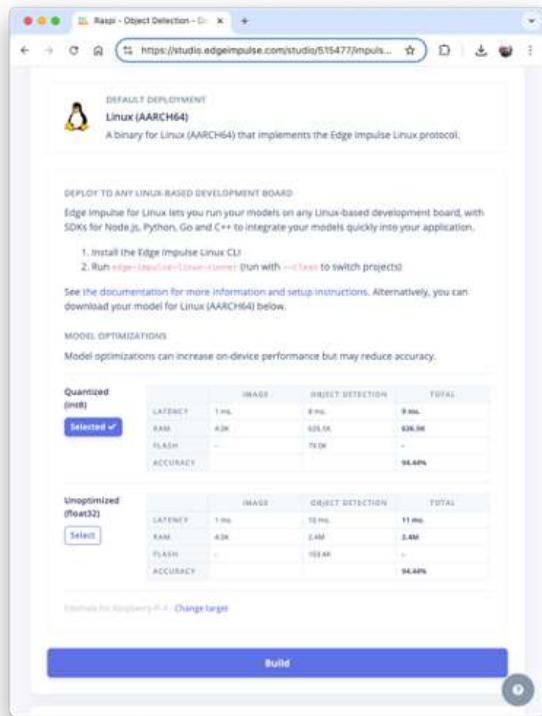
In object detection tasks, accuracy is generally not the primary evaluation metric. Object detection involves classifying objects and providing bounding boxes around them, making it a more complex problem than simple classification. The issue is that we do not have the bounding box, only the centroids.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

Deploying the model

As we did in the previous section, we can deploy the trained model as TFLite or Linux (AARCH64). Let's do it now as **Linux (AARCH64)**, a binary that implements the Edge Impulse Linux protocol.

Edge Impulse for Linux models is delivered in `.eim` format. This executable contains our “full impulse” created in Edge Impulse Studio. The impulse consists of the signal processing block(s) and any learning and anomaly block(s) we added and trained. It is compiled with optimizations for our processor or GPU (e.g., NEON instructions on ARM cores), plus a straightforward IPC layer (over a Unix socket).

At the Deploy tab, select the option **Linux (AARCH64)**, the **int8model** and press **Build**.



The model will be automatically downloaded to your computer.

On our Raspi, let's create a new working area:

```
cd ~  
cd Documents  
mkdir EI_Linux  
cd EI_Linux  
mkdir models  
mkdir images
```

Rename the model for easy identification:

For example, raspi-object-detection-linux-aarch64-FOMO-int8.eim and transfer it to the new Raspi folder ./models and capture or get some images for inference and save them in the folder ./images.

Inference and Post-Processing

The inference will be made using the Linux Python SDK. This library lets us run machine learning models and collect sensor data on Linux machines using Python. The SDK is open source and hosted on GitHub: [edgeimpulse/linux-sdk-python](https://github.com/edgeimpulse/linux-sdk-python).

Let's set up a Virtual Environment for working with the Linux Python SDK

```
python3 -m venv ~/eilinear  
source ~/eilinear/bin/activate
```

And Install the all the libraries needed:

```
sudo apt-get update  
sudo apt-get install libatlas-base-dev\  
libportaudio0 libportaudio2  
sudo apt-get install libportaudiocpp0 portaudio19-dev  
  
pip3 install edge_impulse_linux -i https://pypi.python.org/simple  
pip3 install Pillow matplotlib pyaudio opencv-contrib-python  
  
sudo apt-get install portaudio19-dev  
pip3 install pyaudio  
pip3 install opencv-contrib-python
```

Permit our model to be executable.

```
chmod +x raspi-object-detection-linux-aarch64-FOMO-int8.eim
```

Install the Jupiter Notebook on the new environment

```
pip3 install jupyter
```

Run a notebook locally (on the Raspi-4 or 5 with desktop)

```
jupyter notebook
```

or on the browser on your computer:

```
jupyter notebook --ip=192.168.4.210 --no-browser
```

Let's start a new notebook by following all the steps to detect cubes and wheels on an image using the FOMO model and the Edge Impulse Linux Python SDK.

Import the needed libraries:

```
import sys, time
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from PIL import Image
import cv2
from edge_impulse_linux.image import ImageImpulseRunner
```

Define the model path and labels:

```
model_file = "raspi-object-detection-linux-aarch64-int8.eim"
model_path = "models/" + model_file # Trained ML model from
# Edge Impulse
labels = ["box", "wheel"]
```

Remember that the model will output the class ID as values (0 and 1), following an alphabetic order regarding the class names.

Load and initialize the model:

```
# Load the model file
runner = ImageImpulseRunner(model_path)

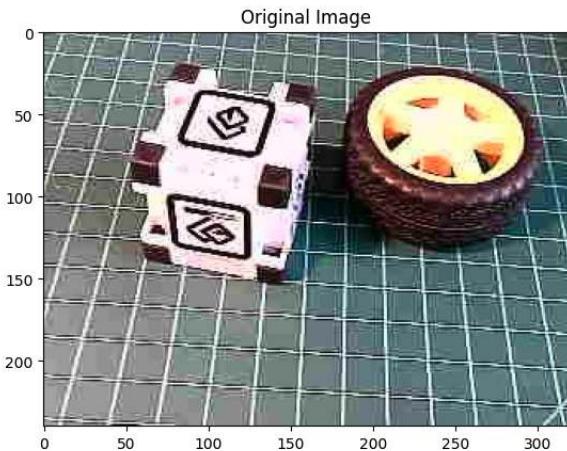
# Initialize model
model_info = runner.init()
```

The `model_info` will contain critical information about our model. However, unlike the TFLite interpreter, the EI Linux Python SDK library will now prepare the model for inference.

So, let's open the image and show it (Now, for compatibility, we will use OpenCV, the CV Library used internally by EI. OpenCV reads the image as BGR, so we will need to convert it to RGB :

```
# Load the image
img_path = "./images/1_box_1_wheel.jpg"
orig_img = cv2.imread(img_path)
img_rgb = cv2.cvtColor(orig_img, cv2.COLOR_BGR2RGB)

# Display the image
plt.imshow(img_rgb)
plt.title("Original Image")
plt.show()
```



Now we will get the features and the preprocessed image (cropped) using the runner:

```
features, cropped = (
    runner.get_features_from_image_auto_studio_settings(img_rgb)
)
```

And perform the inference. Let's also calculate the latency of the model:

```
res = runner.classify(features)
```

Let's get the output classes of objects detected, their bounding boxes centroids, and probabilities.

```
print(
    "Found %d bounding boxes (%d ms.)"
    % (
        len(res["result"]["bounding_boxes"]),
        res["timing"]["dsp"] + res["timing"]["classification"],
    )
)
for bb in res["result"]["bounding_boxes"]:
    print(
        "\t%s (%.2f): x=%d y=%d w=%d h=%d"
        % (
            bb["label"],
            bb["value"],
            bb["x"],
            bb["y"],
```

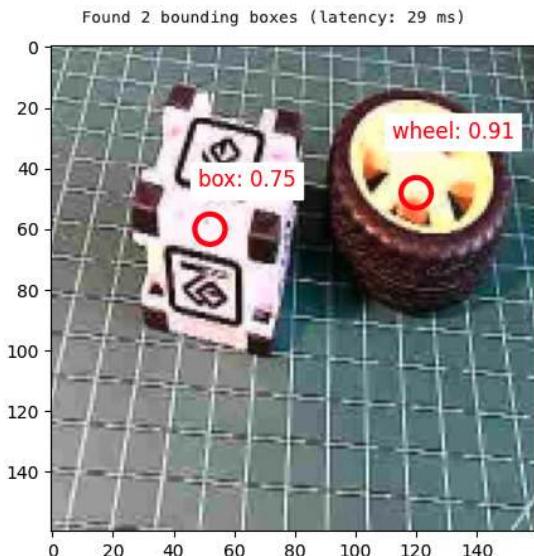
```
        bb["width"] ,  
        bb["height"] ,  
    )  
)  
  
Found 2 bounding boxes (29 ms.)  
1 (0.91): x=112 y=40 w=16 h=16  
0 (0.75): x=48 y=56 w=8 h=8
```

The results show that two objects were detected: one with class ID 0 (box) and one with class ID 1 (wheel), which is correct!

Let's visualize the result (The threshold is 0.5, the default value set during the model testing on the Edge Impulse Studio).

```
print(  
    "\tFound %d bounding boxes (latency: %d ms)"  
    % (  
        len(res["result"]["bounding_boxes"]) ,  
        res["timing"]["dsp"] + res["timing"]["classification"] ,  
    )  
)  
plt.figure(figsize=(5, 5))  
plt.imshow(cropped)  
  
# Go through each of the returned bounding boxes  
bboxes = res["result"]["bounding_boxes"]  
for bbox in bboxes:  
  
    # Get the corners of the bounding box  
    left = bbox["x"]  
    top = bbox["y"]  
    width = bbox["width"]  
    height = bbox["height"]  
  
    # Draw a circle centered on the detection  
    circ = plt.Circle(  
        (left + width // 2, top + height // 2) ,  
        5 ,  
        fill=False ,  
        color="red" ,  
        linewidth=3 ,  
    )  
    plt.gca().add_patch(circ)
```

```
class_id = int(bbox["label"])
class_name = labels[class_id]
plt.text(
    left,
    top - 10,
    f'{class_name}: {bbox["value"]:.2f}',
    color="red",
    fontsize=12,
    backgroundcolor="white",
)
plt.show()
```



Exploring a YOLO Model using Ultralytics

For this lab, we will explore YOLOv8. Ultralytics YOLOv8 is a version of the acclaimed real-time object detection and image segmentation model, YOLO. YOLOv8 is built on cutting-edge advancements in deep learning and computer vision, offering unparalleled performance in terms of speed and accuracy. Its streamlined design makes it suitable for various applications and easily adaptable to different hardware platforms, from edge devices to cloud APIs.

Talking about the YOLO Model

The YOLO (You Only Look Once) model is a highly efficient and widely used object detection algorithm known for its real-time processing capabilities. Unlike traditional object detection systems that repurpose classifiers or localizers to perform detection, YOLO frames the detection problem as a single regression task. This innovative approach enables YOLO to simultaneously predict multiple bounding boxes and their class probabilities from full images in one evaluation, significantly boosting its speed.

Key Features:

1. Single Network Architecture:

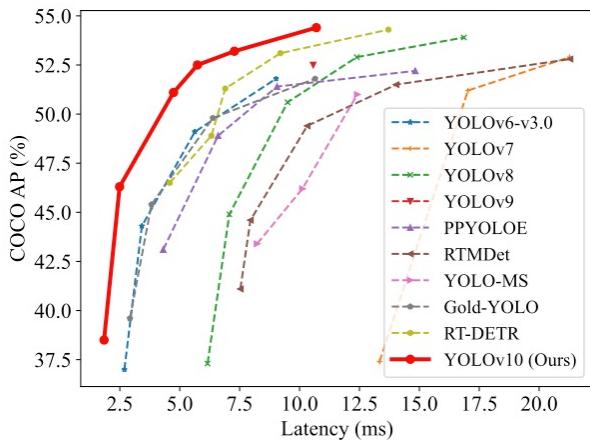
- YOLO employs a single neural network to process the entire image. This network divides the image into a grid and, for each grid cell, directly predicts bounding boxes and associated class probabilities. This end-to-end training improves speed and simplifies the model architecture.

2. Real-Time Processing:

- One of YOLO's standout features is its ability to perform object detection in real-time. Depending on the version and hardware, YOLO can process images at high frames per second (FPS). This makes it ideal for applications requiring quick and accurate object detection, such as video surveillance, autonomous driving, and live sports analysis.

3. Evolution of Versions:

- Over the years, YOLO has undergone significant improvements, from YOLOv1 to the latest YOLOv10. Each iteration has introduced enhancements in accuracy, speed, and efficiency. YOLOv8, for instance, incorporates advancements in network architecture, improved training methodologies, and better support for various hardware, ensuring a more robust performance.
- Although YOLOv10 is the family's newest member with an encouraging performance based on its paper, it was just released (May 2024) and is not fully integrated with the Ultralytics library. Conversely, the precision-recall curve analysis suggests that YOLOv8 generally outperforms YOLOv9, capturing a higher proportion of true positives while minimizing false positives more effectively (for more details, see this article). So, this lab is based on the YOLOv8n.



4. Accuracy and Efficiency:

- While early versions of YOLO traded off some accuracy for speed, recent versions have made substantial strides in balancing both. The newer models are faster and more accurate, detecting small objects (such as bees) and performing well on complex datasets.

5. Wide Range of Applications:

- YOLO's versatility has led to its adoption in numerous fields. It is used in traffic monitoring systems to detect and count vehicles, security applications to identify potential threats and agricultural technology to monitor crops and livestock. Its application extends to any domain requiring efficient and accurate object detection.

6. Community and Development:

- YOLO continues to evolve and is supported by a strong community of developers and researchers (being the YOLOv8 very strong). Open-source implementations and extensive documentation have made it accessible for customization and integration into various projects. Popular deep learning frameworks like Darknet, TensorFlow, and PyTorch support YOLO, further broadening its applicability.
- Ultralytics YOLOv8 can not only Detect (our case here) but also Segment and Pose models pre-trained on the COCO dataset and YOLOv8 Classify models pre-trained on the ImageNet dataset. Track mode is available for all Detect, Segment, and Pose models.



Figure 1.24: Ultralytics YOLO supported tasks

Installation

On our Raspi, let's deactivate the current environment to create a new working area:

```
deactivate
cd ~
cd Documents/
mkdir YOLO
cd YOLO
mkdir models
mkdir images
```

Let's set up a Virtual Environment for working with the Ultralytics YOLOv8

```
python3 -m venv ~/yolo
source ~/yolo/bin/activate
```

And install the Ultralytics packages for local inference on the Raspi

1. Update the packages list, install pip, and upgrade to the latest:

```
sudo apt update
sudo apt install python3-pip -y
pip install -U pip
```

2. Install the ultralytics pip package with optional dependencies:

```
pip install ultralytics [export]
```

3. Reboot the device:

```
sudo reboot
```

Testing the YOLO

After the Raspi-Zero booting, let's activate the yolo env, go to the working directory,

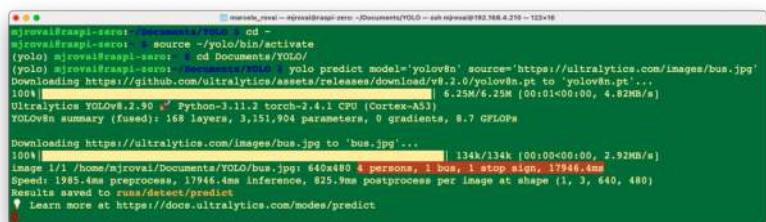
```
source ~/yolo/bin/activate  
cd /Documents/YOLO
```

and run inference on an image that will be downloaded from the Ultralytics website, using the YOLOV8n model (the smallest in the family) at the Terminal (CLI):

```
yolo predict model='yolov8n' \  
source='https://ultralytics.com/images/bus.jpg'
```

The YOLO model family is pre-trained with the COCO dataset.

The inference result will appear in the terminal. In the image (bus.jpg), 4 persons, 1 bus, and 1 stop signal were detected:



```
micro:zero ~ micro:zero ~ micro:zero ~ [Documents/YOLO - ssh micro@192.168.4.210 ~ 128x16  
micro:zero ~ micro:zero ~ source ~/yolo/bin/activate  
(yolo) micro:zero ~ micro:zero ~ cd Documents/YOLO/  
(yolo) micro:zero ~ micro:zero ~ yolo predict model='yolov8n' source='https://ultralytics.com/images/bus.jpg'  
Downloading https://github.com/ultralytics/assets/releases/download/v8.2.0/yolov8n.pt to 'yolov8n.pt'...  
100% [██████████] 6.25M/6.25M (00:01<00:00, 4.82MB/s)  
YOLOv8n YOLOv8n 2.90 Python-3.11.2 torch-2.4.1 CPU (Cortex-A53)  
YOLOv8n summary (fused): 165 layers, 3,151,704 parameters, 0 gradients, 8.7 GFLOPs  
Downloading https://ultralytics.com/images/bus.jpg to 'bus.jpg'...  
100% [██████████] 134k/134k (00:00<00:00, 2.92MB/s)  
Image 1/1 /home/micro:zero/Documents/YOLO/bus.jpg: 640x480 4 persons, 1 bus, 1 stop sign, 17946.4ms  
Speed: 1985.4ms preprocess, 17946.4ms inference, 825.9ms postprocess per image at shape (1, 3, 640, 480)  
Results saved to runs/detect/predict  
Learn more at https://docs.ultralytics.com/models/predict
```

Also, we got a message that Results saved to runs/detect/predict. Inspecting that directory, we can see a new image saved (bus.jpg). Let's download it from the Raspi-Zero to our desktop for inspection:



So, the Ultralytics YOLO is correctly installed on our Raspi. But, on the Raspi-Zero, an issue is the high latency for this inference, around 18 seconds, even with the most miniature model of the family (YOLOv8n).

Export Model to NCNN format

Deploying computer vision models on edge devices with limited computational power, such as the Raspi-Zero, can cause latency issues. One alternative is to use a format optimized for optimal performance. This ensures that even devices with limited processing power can handle advanced computer vision tasks well.

Of all the model export formats supported by Ultralytics, the NCNN is a high-performance neural network inference computing framework optimized for mobile platforms. From the beginning of the design, NCNN

was deeply considerate about deployment and use on mobile phones and did not have third-party dependencies. It is cross-platform and runs faster than all known open-source frameworks (such as TFLite).

NCNN delivers the best inference performance when working with Raspberry Pi devices. NCNN is highly optimized for mobile embedded platforms (such as ARM architecture).

So, let's convert our model and rerun the inference:

1. Export a YOLOv8n PyTorch model to NCNN format, creating: '/yolov8n_ncnn_model'

```
yolo export model=yolov8n.pt format=ncnn
```

2. Run inference with the exported model (now the source could be the bus.jpg image that was downloaded from the website to the current directory on the last inference):

```
yolo predict model='./yolov8n_ncnn_model' source='bus.jpg'
```

The first inference, when the model is loaded, usually has a high latency (around 17s), but from the 2nd, it is possible to note that the inference goes down to around 2s.

Exploring YOLO with Python

To start, let's call the Python Interpreter so we can explore how the YOLO model works, line by line:

```
python3
```

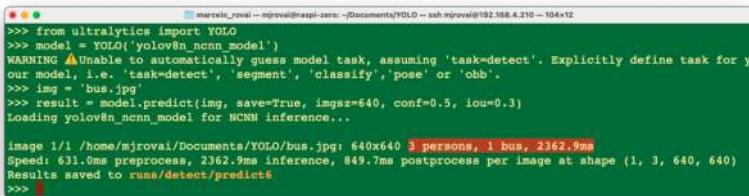
Now, we should call the YOLO library from Ultralytics and load the model:

```
from ultralytics import YOLO
```

```
model = YOLO("yolov8n_ncnn_model")
```

Next, run inference over an image (let's use again bus.jpg):

```
img = "bus.jpg"
result = model.predict(img, save=True, imgsz=640, conf=0.5, iou=0.3)
```



```

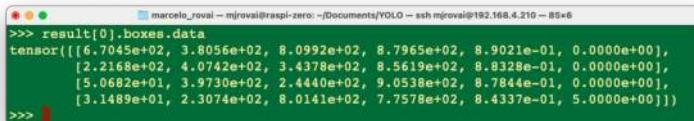
>>> from ultralytics import YOLO
>>> model = YOLO('yolov8n_ncnn.onnx')
WARNING ▲ Unable to automatically guess model task, assuming 'task=detect'. Explicitly define task for your model, i.e. 'task=detect', 'segment', 'classify', 'pose' or 'obb'.
>>> img = 'bus.jpg'
>>> result = model.predict(img, save=True, imgsz=640, conf=0.5, iou=0.3)
Loading yolov8n_ncnn.onnx for NCNN inference...
image 1/1 /home/mjroval/Documents/YOLO/bus.jpg: 640x640 3 persons, 1 bus, 2362.9ms
Speed: 631.0ms preprocess, 2362.9ms inference, 849.7ms postprocess per image at shape (1, 3, 640, 640)
Results saved to runs/detect/predict
>>>

```

We can verify that the result is almost identical to the one we get running the inference at the terminal level (CLI), except that the bus stop was not detected with the reduced NCNN model. Note that the latency was reduced.

Let's analyze the "result" content.

For example, we can see `result[0].boxes.data`, showing us the main inference result, which is a tensor shape (4, 6). Each line is one of the objects detected, being the 4 first columns, the bounding boxes coordinates, the 5th, the confidence, and the 6th, the class (in this case, 0: person and 5: bus):



```

>>> result[0].boxes.data
tensor([[6.7045e+02, 3.8056e+02, 8.0992e+02, 8.7965e+02, 8.9021e-01, 0.0000e+00],
       [2.2168e+02, 4.0742e+02, 3.4378e+02, 8.5619e+02, 8.8328e-01, 0.0000e+00],
       [5.0682e+01, 3.9730e+02, 2.4440e+02, 9.0538e+02, 8.7844e-01, 0.0000e+00],
       [3.1489e+01, 2.3074e+02, 8.0141e+02, 7.7578e+02, 8.4337e-01, 5.0000e+00]])
>>>

```

We can access several inference results separately, as the inference time, and have it printed in a better format:

```

inference_time = int(result[0].speed["inference"])
print(f"Inference Time: {inference_time} ms")

```

Or we can have the total number of objects detected:

```

print(f"Number of objects: {len(result[0].boxes.cls)}")

```

```
marcelo_rovai — mjrovai@raspi-zero: ~/Documents/YOLO — ssh mjrovai@192.168.4.210 — 64x6
>>> inference_time = int(result[0].speed['inference'])
>>> print(f"Inference Time: {inference_time} ms")
Inference Time: 2362 ms
>>> print(f'Number of objects: {len(result[0].boxes.cls)}')
Number of objects: 4
>>>
```

With Python, we can create a detailed output that meets our needs (See Model Prediction with Ultralytics YOLO for more details). Let's run a Python script instead of manually entering it line by line in the interpreter, as shown below. Let's use nano as our text editor. First, we should create an empty Python script named, for example, `yolov8-tests.py`:

```
nano yolov8_tests.py
```

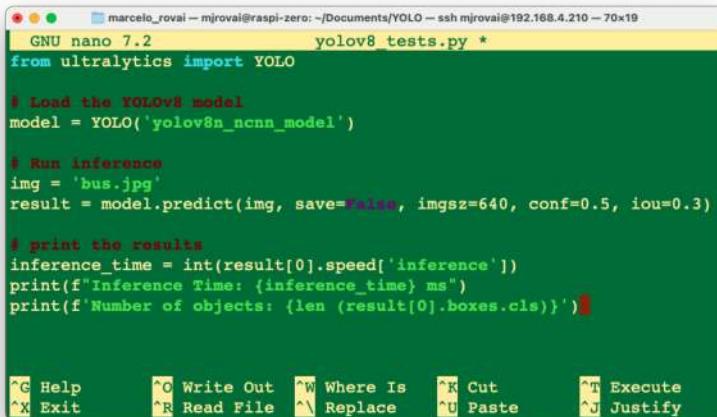
Enter with the code lines:

```
from ultralytics import YOLO

# Load the YOLOv8 model
model = YOLO("yolov8n_ncnn_model")

# Run inference
img = "bus.jpg"
result = model.predict(img, save=False, imgsz=640, conf=0.5, iou=0.3)

# print the results
inference_time = int(result[0].speed["inference"])
print(f"Inference Time: {inference_time} ms")
print(f"Number of objects: {len(result[0].boxes.cls)}")
```



The screenshot shows a terminal window titled "marcelo_rovai — mirovai@raspi-zero: ~/Documents/YOLO — ssh mirovai@192.168.4.210 — 70x19". The window contains the following Python code:

```
GNU nano 7.2              yolov8_tests.py *
from ultralytics import YOLO

# Load the YOLOv8 model
model = YOLO('yolov8n_ncnn_model')

# Run inference
img = 'bus.jpg'
result = model.predict(img, save=False, imgsz=640, conf=0.5, iou=0.3)

# print the results
inference_time = int(result[0].speed['inference'])
print(f"Inference Time: {inference_time} ms")
print(f"Number of objects: {len(result[0].boxes.cls)})"
```

At the bottom of the terminal window, there is a menu bar with the following options: ^G Help, ^O Write Out, ^W Where Is, ^K Cut, ^T Execute, ^X Exit, ^R Read File, ^\ Replace, ^U Paste, and ^J Justify.

And enter with the commands: [CTRL+O] + [ENTER] + [CTRL+X] to save the Python script.

Run the script:

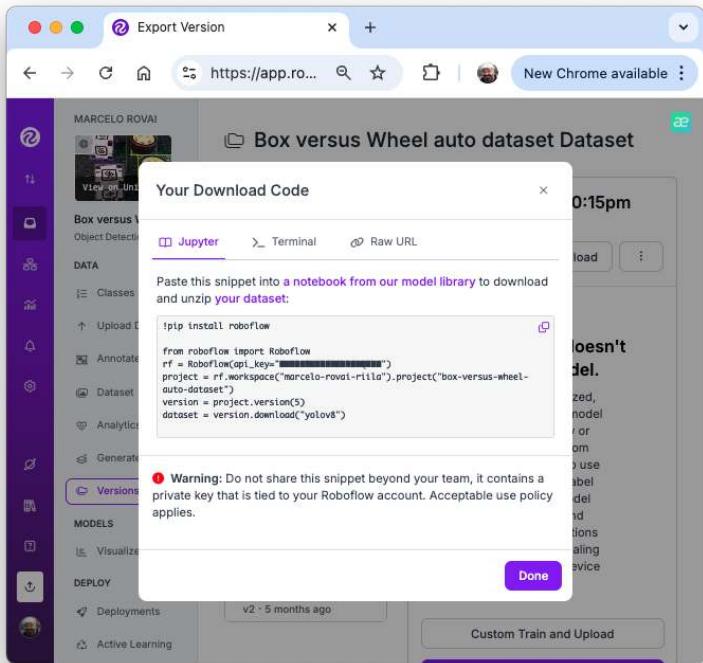
```
python yolov8_tests.py
```

The result is the same as running the inference at the terminal level (CLI) and with the built-in Python interpreter.

Calling the YOLO library and loading the model for inference for the first time takes a long time, but the inferences after that will be much faster. For example, the first single inference can take several seconds, but after that, the inference time should be reduced to less than 1 second.

Training YOLOv8 on a Customized Dataset

Return to our “Box versus Wheel” dataset, labeled on Roboflow. On the Download Dataset, instead of Download a zip to computer option done for training on Edge Impulse Studio, we will opt for Show download code. This option will open a pop-up window with a code snippet that should be pasted into our training notebook.



For training, let's adapt one of the public examples available from Ultralytics and run it on Google Colab. Below, you can find mine to be adapted in your project:

- YOLOv8 Box versus Wheel Dataset Training [Open In Colab]

Critical points on the Notebook:

1. Run it with GPU (the NVidia T4 is free)
2. Install Ultralytics using PIP.

A screenshot of a Jupyter Notebook cell. The cell number is 3. The code is:

```
1 # Pip install method (recommended)
2
3 !pip install ultralytics
4
5 from IPython import display
6 display.clear_output()
7
8 import ultralytics
9 ultralytics.checks()
```

The output of the cell shows the results of the command:

```
Ultralytics YOLOv8.2.91 🚀 Python-3.10.12 torch-2.4.0+cu121 CUDA:0 (Tesla T4, 15102MiB)
Setup complete ✓ (2 CPUs, 12.7 GB RAM, 32.8/112.6 GB disk)
```

3. Now, you can import the YOLO and upload your dataset to the Co-Lab, pasting the Download code that we get from Roboflow. Note that our dataset will be mounted under /content/datasets/:



4. It is essential to verify and change the file `data.yaml` with the correct path for the images (copy the path on each `images` folder).

```
names: 
- box
- wheel
nc: 2
roboflow:
  license: CC BY 4.0
  project: box-versus-wheel-auto-dataset
  url: https://universe.roboflow.com/marcelo-rovai-riila/ \
        box-versus-wheel-auto-dataset/dataset/5
  version: 5
  workspace: marcelo-rovai-riila
test: /content/datasets/Box-versus-Wheel-auto-dataset-5/ \
      test/images
train: /content/datasets/Box-versus-Wheel-auto-dataset-5/ \
      train/images
val: /content/datasets/Box-versus-Wheel-auto-dataset-5/ \
      valid/images
```

5. Define the main hyperparameters that you want to change from default, for example:

```
MODEL = 'yolov8n.pt'  
IMG_SIZE = 640  
EPOCHS = 25 # For a final project, you should consider  
# at least 100 epochs
```

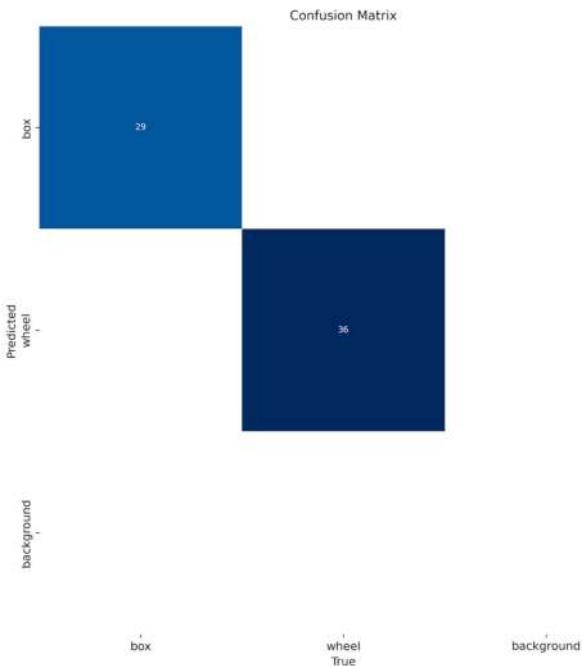
6. Run the training (using CLI):

```
!yolo task=detect mode=train model={MODEL} \  
data={dataset.location}/data.yaml \  
epochs={EPOCHS} \  
imgsz={IMG_SIZE} plots=True
```

```
25 epochs completed in 0.026 hours.  
Optimizer stripped from runs/detect/train/weights/last.pt, 6.2MB  
Optimizer stripped from runs/detect/train/weights/best.pt, 6.2MB  
Validating runs/detect/train/weights/best.pt...  
Ultralytics YOLOv8.2.91 🚀 Python-3.10.12 torch-2.4.0+cu121 CUDA:0 (Tesla T4, 15102MiB)  
Model summary (fused): 168 layers, 3,086,038 parameters, 0 gradients, 8.1 GFLOPs  
Class Images Instances Box(R) R mAP50 mAP50-95: 100% 1/1 [00:00<00:00, 7.61it/s]  
all 12 65 0.997 1 0.995 0.899  
box 11 29 0.999 1 0.995 0.903  
wheel 11 36 0.995 1 0.995 0.896  
Speed: 0.2ms preprocess, 2.6ms inference, 0.0ms loss, 3.2ms postprocess per image
```

Figure 1.25:
image-20240910111319804

The model took a few minutes to be trained and has an excellent result (mAP50 of 0.995). At the end of the training, all results are saved in the folder listed, for example: `/runs/detect/train/`. There, you can find, for example, the confusion matrix.



7. Note that the trained model (`best.pt`) is saved in the folder `/runs/detect/train/weights/`. Now, you should validate the trained model with the `valid/images`.

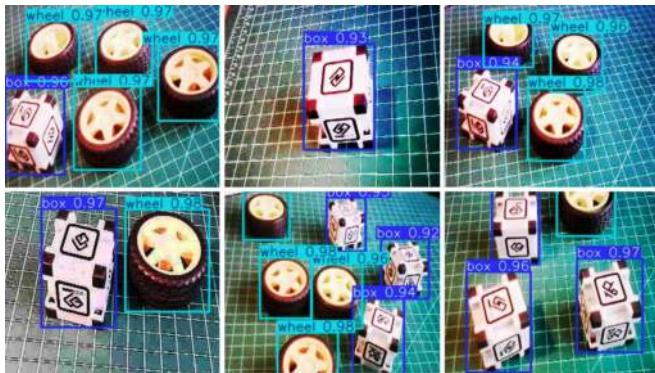
```
!yolo task=detect mode=val model={HOME}/runs/detect/train/\\
weights/best.pt data={dataset.location}/data.yaml
```

The results were similar to training.

8. Now, we should perform inference on the images left aside for testing

```
!yolo task=detect mode=predict model={HOME}/runs/detect/train/\\
weights/best.pt conf=0.25 source={dataset.location}/test/\\
images save=True
```

The inference results are saved in the folder `runs/detect/predict`. Let's see some of them:



9. It is advised to export the train, validation, and test results for a Drive at Google. To do so, we should mount the drive.

```
from google.colab import drive
drive.mount('/content/gdrive')
```

and copy the content of /runs folder to a folder that you should create in your Drive, for example:

```
!scp -r /content/runs '/content/gdrive/MyDrive/\
10_UNIFEI/Box_vs_Wheel_Project'
```

Inference with the trained model, using the Raspi

Download the trained model /runs/detect/train/weights/best.pt to your computer. Using the FileZilla FTP, let's transfer the best.pt to the Raspi models folder (before the transfer, you may change the model name, for example, box_wheel_320_yolo.pt).

Using the FileZilla FTP, let's transfer a few images from the test dataset to .\YOLO\images:

Let's return to the YOLO folder and use the Python Interpreter:

```
cd ..
python
```

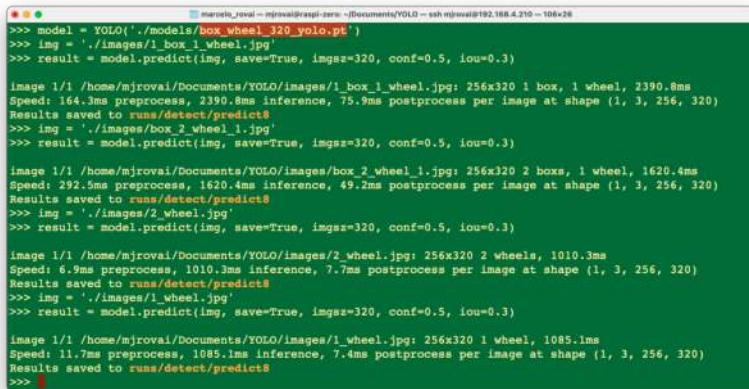
As before, we will import the YOLO library and define our converted model to detect bees:

```
from ultralytics import YOLO
model = YOLO("./models/box_wheel_320_yolo.pt")
```

Now, let's define an image and call the inference (we will save the image result this time to external verification):

```
img = "./images/1_box_1_wheel.jpg"
result = model.predict(img, save=True, imgsz=320, conf=0.5, iou=0.3)
```

Let's repeat for several images. The inference result is saved on the variable `result`, and the processed image on `runs/detect/predict8`



```
marcos_raspi ~ mirovai@raspi-zero:~/Documents/YOLO -- ssh mirovai@192.168.4.210 ~ 106x24
>>> model = YOLO('./models/box_wheel_320_yolo.pt')
>>> img = './images/1_box_1_wheel.jpg'
>>> result = model.predict(img, save=True, imgsz=320, conf=0.5, iou=0.3)

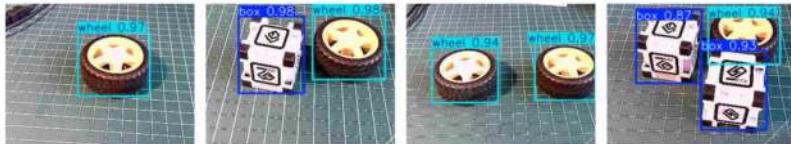
image 1/1 /home/mirovai/Documents/YOLO/images/1_box_1_wheel.jpg: 256x320 1 box, 1 wheel, 2390.8ms
Speed: 164.3ms preprocess, 2390.8ms inference, 75.9ms postprocess per image at shape (1, 3, 256, 320)
Results saved to runs/detect/predict8
>>> img = './images/box_2_wheel_1.jpg'
>>> result = model.predict(img, save=True, imgsz=320, conf=0.5, iou=0.3)

image 1/1 /home/mirovai/Documents/YOLO/images/box_2_wheel_1.jpg: 256x320 2 boxes, 1 wheel, 1620.4ms
Speed: 292.5ms preprocess, 1620.4ms inference, 49.2ms postprocess per image at shape (1, 3, 256, 320)
Results saved to runs/detect/predict8
>>> img = './images/2.wheel.jpg'
>>> result = model.predict(img, save=True, imgsz=320, conf=0.5, iou=0.3)

image 1/1 /home/mirovai/Documents/YOLO/images/2.wheel.jpg: 256x320 2 wheels, 1010.3ms
Speed: 6.9ms preprocess, 1010.3ms inference, 7.7ms postprocess per image at shape (1, 3, 256, 320)
Results saved to runs/detect/predict8
>>> img = './images/1.wheel.jpg'
>>> result = model.predict(img, save=True, imgsz=320, conf=0.5, iou=0.3)

image 1/1 /home/mirovai/Documents/YOLO/images/1.wheel.jpg: 256x320 1 wheel, 1085.1ms
Speed: 11.7ms preprocess, 1085.1ms inference, 7.4ms postprocess per image at shape (1, 3, 256, 320)
Results saved to runs/detect/predict8
>>> 
```

Using FileZilla FTP, we can send the inference result to our Desktop for verification:



We can see that the inference result is excellent! The model was trained based on the smaller base model of the YOLOv8 family (YOLOv8n). The issue is the latency, around 1 second (or 1 FPS on the Raspi-Zero). Of course, we can reduce this latency and convert the model to TFLite or NCNN.

Object Detection on a live stream

All the models explored in this lab can detect objects in real-time using a camera. The captured image should be the input for the trained and converted model. For the Raspi-4 or 5 with a desktop, OpenCV can capture the frames and display the inference result.

However, creating a live stream with a webcam to detect objects in real-time is also possible. For example, let's start with the script developed for the Image Classification app and adapt it for a *Real-Time Object Detection Web Application Using TensorFlow Lite and Flask*.

This app version will work for all TFLite models. Verify if the model is in its correct folder, for example:

```
model_path = "./models/ssd-mobilenet-v1-tflite-default-v1.tflite"
```

Download the Python script `object_detection_app.py` from GitHub.

And on the terminal, run:

```
python3 object_detection_app.py
```

And access the web interface:

- On the Raspberry Pi itself (if you have a GUI): Open a web browser and go to `http://localhost:5000`
- From another device on the same network: Open a web browser and go to `http://<raspberry_pi_ip>:5000` (Replace `<raspberry_pi_ip>` with your Raspberry Pi's IP address). For example:
`http://192.168.4.210: 5000/`

Here are some screenshots of the app running on an external desktop



Let's see a technical description of the key modules used in the object detection application:

1. TensorFlow Lite (tflite_runtime):

- Purpose: Efficient inference of machine learning models on edge devices.
- Why: TFLite offers reduced model size and optimized performance compared to full TensorFlow, which is crucial for resource-constrained devices like Raspberry Pi. It supports hardware acceleration and quantization, further improving efficiency.
- Key functions: Interpreter for loading and running the model, `get_input_details()`, and `get_output_details()` for interfacing with the model.

2. Flask:

- Purpose: Lightweight web framework for creating the back-end server.
- Why: Flask's simplicity and flexibility make it ideal for rapidly developing and deploying web applications. It's less resource-intensive than larger frameworks suitable for edge devices.
- Key components: route decorators for defining API endpoints, Response objects for streaming video, `render_template_string` for serving dynamic HTML.

3. Picamera2:

- Purpose: Interface with the Raspberry Pi camera module.
- Why: Picamera2 is the latest library for controlling Raspberry Pi cameras, offering improved performance and features over the original Picamera library.
- Key functions: `create_preview_configuration()` for setting up the camera, `capture_file()` for capturing frames.

4. PIL (Python Imaging Library):

- Purpose: Image processing and manipulation.
- Why: PIL provides a wide range of image processing capabilities. It's used here to resize images, draw bounding boxes, and convert between image formats.
- Key classes: `Image` for loading and manipulating images, `ImageDraw` for drawing shapes and text on images.

5. NumPy:

- Purpose: Efficient array operations and numerical computing.
- Why: NumPy's array operations are much faster than pure Python lists, which is crucial for efficiently processing image data and model inputs/outputs.
- Key functions: `array()` for creating arrays, `expand_dims()` for adding dimensions to arrays.

6. Threading:

- Purpose: Concurrent execution of tasks.
- Why: Threading allows simultaneous frame capture, object detection, and web server operation, crucial for maintaining real-time performance.

- Key components: Thread class creates separate execution threads, and Lock is used for thread synchronization.

7. **io.BytesIO:**

- Purpose: In-memory binary streams.
- Why: Allows efficient handling of image data in memory without needing temporary files, improving speed and reducing I/O operations.

8. **time:**

- Purpose: Time-related functions.
- Why: Used for adding delays (`time.sleep()`) to control frame rate and for performance measurements.

9. **jQuery (client-side):**

- Purpose: Simplified DOM manipulation and AJAX requests.
- Why: It makes it easy to update the web interface dynamically and communicate with the server without page reloads.
- Key functions: `.get()` and `.post()` for AJAX requests, DOM manipulation methods for updating the UI.

Regarding the main app system architecture:

1. **Main Thread:** Runs the Flask server, handling HTTP requests and serving the web interface.
2. **Camera Thread:** Continuously captures frames from the camera.
3. **Detection Thread:** Processes frames through the TFLite model for object detection.
4. **Frame Buffer:** Shared memory space (protected by locks) storing the latest frame and detection results.

And the app data flow, we can describe in short:

1. Camera captures frame → Frame Buffer
2. Detection thread reads from Frame Buffer → Processes through TFLite model → Updates detection results in Frame Buffer
3. Flask routes access Frame Buffer to serve the latest frame and detection results
4. Web client receives updates via AJAX and updates UI

This architecture allows for efficient, real-time object detection while maintaining a responsive web interface running on a resource-constrained edge device like a Raspberry Pi. Threading and efficient libraries like

TFLite and PIL enable the system to process video frames in real-time, while Flask and jQuery provide a user-friendly way to interact with them.

You can test the app with another pre-processed model, such as the EfficientDet, changing the app line:

```
model_path = "./models/lite-model_efficientdet_lite0_\
    detection_metadata_1.tflite"
```

If we want to use the app for the SSD-MobileNetV2 model, trained on Edge Impulse Studio with the “Box versus Wheel” dataset, the code should also be adapted depending on the input details, as we have explored on its notebook.

Summary

This lab has explored the implementation of object detection on edge devices like the Raspberry Pi, demonstrating the power and potential of running advanced computer vision tasks on resource-constrained hardware. We've covered several vital aspects:

1. **Model Comparison:** We examined different object detection models, including SSD-MobileNet, EfficientDet, FOMO, and YOLO, comparing their performance and trade-offs on edge devices.
2. **Training and Deployment:** Using a custom dataset of boxes and wheels (labeled on Roboflow), we walked through the process of training models using Edge Impulse Studio and Ultralytics and deploying them on Raspberry Pi.
3. **Optimization Techniques:** To improve inference speed on edge devices, we explored various optimization methods, such as model quantization (TFLite int8) and format conversion (e.g., to NCNN).
4. **Real-time Applications:** The lab exemplified a real-time object detection web application, demonstrating how these models can be integrated into practical, interactive systems.
5. **Performance Considerations:** Throughout the lab, we discussed the balance between model accuracy and inference speed, a critical consideration for edge AI applications.

The ability to perform object detection on edge devices opens up numerous possibilities across various domains, from precision agriculture, industrial automation, and quality control to smart home applications

and environmental monitoring. By processing data locally, these systems can offer reduced latency, improved privacy, and operation in environments with limited connectivity.

Looking ahead, potential areas for further exploration include:

- Implementing multi-model pipelines for more complex tasks
- Exploring hardware acceleration options for Raspberry Pi
- Integrating object detection with other sensors for more comprehensive edge AI systems
- Developing edge-to-cloud solutions that leverage both local processing and cloud resources

Object detection on edge devices can create intelligent, responsive systems that bring the power of AI directly into the physical world, opening up new frontiers in how we interact with and understand our environment.

Resources

- Dataset (“Box versus Wheel”)
- SSD-MobileNet Notebook on a Raspi
- EfficientDet Notebook on a Raspi
- FOMO - EI Linux Notebook on a Raspi
- YOLOv8 Box versus Wheel Dataset Training on CoLab
- Edge Impulse Project - SSD MobileNet and FOMO
- Python Scripts
- Models

Small Language Models (SLM)

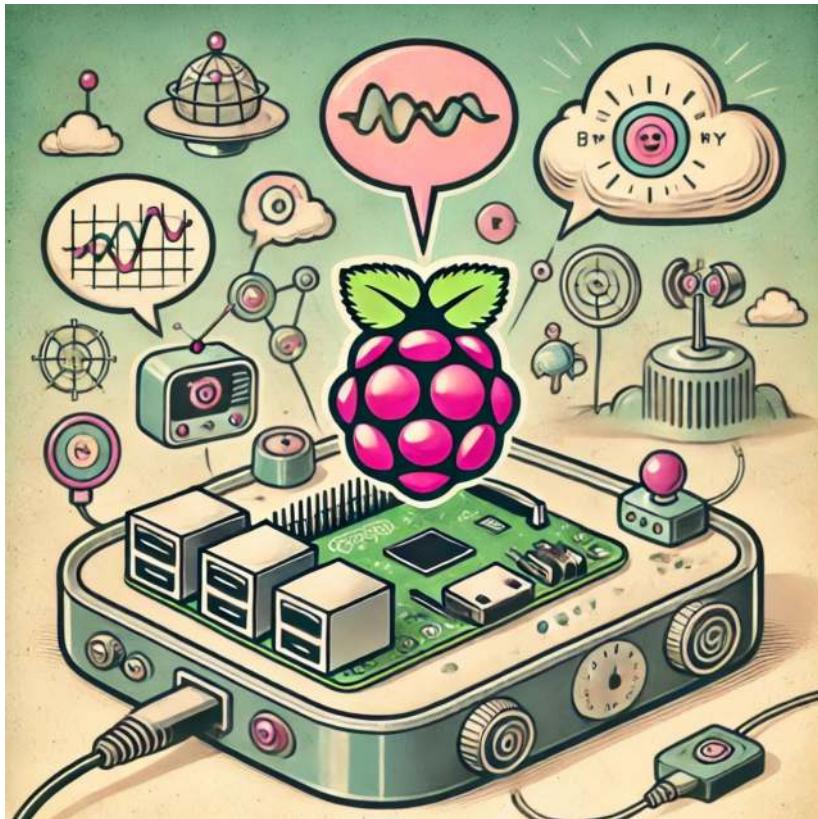


Figure 1.26: DALL-E prompt - A 1950s-style cartoon illustration showing a Raspberry Pi running a small language model at the edge. The Raspberry Pi is stylized in a retro-futuristic way with rounded edges and chrome accents, connected to playful cartoonish sensors and devices. Speech bubbles are floating around, representing language processing, and the background has a whimsical landscape of interconnected devices with wires and small gadgets, all drawn in a vintage cartoon style. The color palette uses soft pastel colors and bold outlines typical of 1950s cartoons, giving a fun and nostalgic vibe to the scene.

Overview

In the fast-growing area of artificial intelligence, edge computing presents an opportunity to decentralize capabilities traditionally reserved for powerful, centralized servers. This lab explores the practical integration of small versions of traditional large language models (LLMs) into a Raspberry Pi 5, transforming this edge device into an AI hub capable of real-time, on-site data processing.

As large language models grow in size and complexity, Small Language Models (SLMs) offer a compelling alternative for edge devices, striking a balance between performance and resource efficiency. By running these models directly on Raspberry Pi, we can create responsive, privacy-preserving applications that operate even in environments with limited or no internet connectivity.

This lab will guide you through setting up, optimizing, and leveraging SLMs on Raspberry Pi. We will explore the installation and utilization of Ollama. This open-source framework allows us to run LLMs locally on our machines (our desktops or edge devices such as the Raspberry Pis or NVidia Jetsons). Ollama is designed to be efficient, scalable, and easy to use, making it a good option for deploying AI models such as Microsoft Phi, Google Gemma, Meta Llama, and LLaVa (Multimodal). We will integrate some of those models into projects using Python's ecosystem, exploring their potential in real-world scenarios (or at least point in this direction).



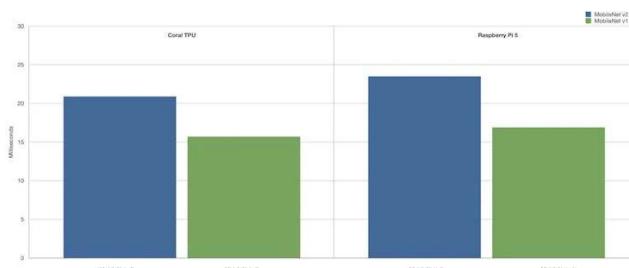
Setup

We could use any Raspi model in the previous labs, but here, the choice must be the Raspberry Pi 5 (Raspi-5). It is a robust platform that substantially upgrades the last version 4, equipped with the Broadcom BCM2712, a 2.4 GHz quad-core 64-bit Arm Cortex-A76 CPU featuring Cryptographic Extension and enhanced caching capabilities. It boasts a VideoCore VII GPU, dual 4Kp60 HDMI® outputs with HDR, and a 4Kp60 HEVC decoder. Memory options include 4 GB and 8 GB of

high-speed LPDDR4X SDRAM, with 8GB being our choice to run SLMs. It also features expandable storage via a microSD card slot and a PCIe 2.0 interface for fast peripherals such as M.2 SSDs (Solid State Drives).

For real SSL applications, SSDs are a better option than SD cards.

By the way, as Alasdair Allan discussed, inferencing directly on the Raspberry Pi 5 CPU—with no GPU acceleration—is now on par with the performance of the Coral TPU.



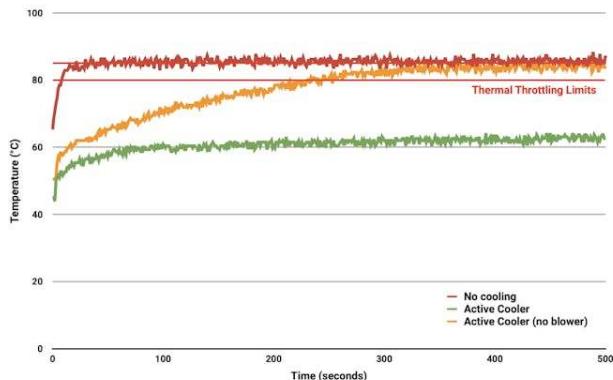
For more info, please see the complete article: [Benchmarking TensorFlow and TensorFlow Lite on Raspberry Pi 5](#).

Raspberry Pi Active Cooler

We suggest installing an Active Cooler, a dedicated clip-on cooling solution for Raspberry Pi 5 (Raspi-5), for this lab. It combines an aluminum heat sink with a temperature-controlled blower fan to keep the Raspi-5 operating comfortably under heavy loads, such as running SLMs.



The Active Cooler has pre-applied thermal pads for heat transfer and is mounted directly to the Raspberry Pi 5 board using spring-loaded push pins. The Raspberry Pi firmware actively manages it: at 60°C, the blower's fan will be turned on; at 67.5°C, the fan speed will be increased; and finally, at 75°C, the fan increases to full speed. The blower's fan will spin down automatically when the temperature drops below these limits.



To prevent overheating, all Raspberry Pi boards begin to throttle the processor when the temperature reaches 80°C and throttle even further when it reaches the maximum temperature of 85°C (more detail [here](#)).

Generative AI (GenAI)

Generative AI is an artificial intelligence system capable of creating new, original content across various mediums such as **text, images, audio, and video**. These systems learn patterns from existing data and use that knowledge to generate novel outputs that didn't previously exist. **Large Language Models (LLMs), Small Language Models (SLMs),** and **multimodal models** can all be considered types of GenAI when used for generative tasks.

GenAI provides the conceptual framework for AI-driven content creation, with LLMs serving as powerful general-purpose text generators. SLMs adapt this technology for edge computing, while multimodal models extend GenAI capabilities across different data types. Together, they represent a spectrum of generative AI technologies, each with its strengths and applications, collectively driving AI-powered content creation and understanding.

Large Language Models (LLMs)

Large Language Models (LLMs) are advanced artificial intelligence systems that understand, process, and generate human-like text. These models are characterized by their massive scale in terms of the amount of data they are trained on and the number of parameters they contain. Critical aspects of LLMs include:

1. **Size:** LLMs typically contain billions of parameters. For example, GPT-3 has 175 billion parameters, while some newer models exceed a trillion parameters.
2. **Training Data:** They are trained on vast amounts of text data, often including books, websites, and other diverse sources, amounting to hundreds of gigabytes or even terabytes of text.
3. **Architecture:** Most LLMs use transformer-based architectures, which allow them to process and generate text by paying attention to different parts of the input simultaneously.
4. **Capabilities:** LLMs can perform a wide range of language tasks without specific fine-tuning, including:
 - Text generation
 - Translation
 - Summarization
 - Question answering
 - Code generation

- Logical reasoning
5. **Few-shot Learning:** They can often understand and perform new tasks with minimal examples or instructions.
 6. **Resource-Intensive:** Due to their size, LLMs typically require significant computational resources to run, often needing powerful GPUs or TPUs.
 7. **Continual Development:** The field of LLMs is rapidly evolving, with new models and techniques constantly emerging.
 8. **Ethical Considerations:** The use of LLMs raises important questions about bias, misinformation, and the environmental impact of training such large models.
 9. **Applications:** LLMs are used in various fields, including content creation, customer service, research assistance, and software development.
 10. **Limitations:** Despite their power, LLMs can produce incorrect or biased information and lack true understanding or reasoning capabilities.

We must note that we use large models beyond text, calling them *multimodal models*. These models integrate and process information from multiple types of input simultaneously. They are designed to understand and generate content across various forms of data, such as text, images, audio, and video.

Closed vs Open Models:

Closed models, also called proprietary models, are AI models whose internal workings, code, and training data are not publicly disclosed. Examples: GPT-4 (by OpenAI), Claude (by Anthropic), Gemini (by Google).

Open models, also known as open-source models, are AI models whose underlying code, architecture, and often training data are publicly available and accessible. Examples: Gemma (by Google), LLaMA (by Meta) and Phi (by Microsoft).

Open models are particularly relevant for running models on edge devices like Raspberry Pi as they can be more easily adapted, optimized, and deployed in resource-constrained environments. Still, it is crucial to verify their Licenses. Open models come with various open-source licenses that may affect their use in commercial applications, while closed models have clear, albeit restrictive, terms of service.

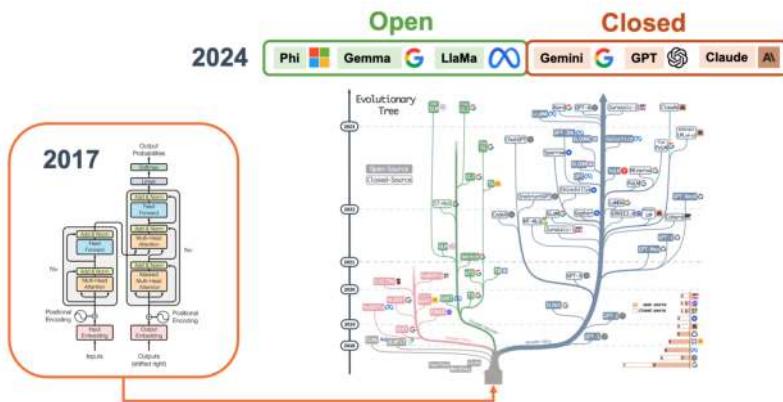


Figure 1.27: Adapted from arXiv

Small Language Models (SLMs)

In the context of edge computing on devices like Raspberry Pi, full-scale LLMs are typically too large and resource-intensive to run directly. This limitation has driven the development of smaller, more efficient models, such as the Small Language Models (SLMs).

SLMs are compact versions of LLMs designed to run efficiently on resource-constrained devices such as smartphones, IoT devices, and single-board computers like the Raspberry Pi. These models are significantly smaller in size and computational requirements than their larger counterparts while still retaining impressive language understanding and generation capabilities.

Key characteristics of SLMs include:

1. **Reduced parameter count:** Typically ranging from a few hundred million to a few billion parameters, compared to two-digit billions in larger models.
2. **Lower memory footprint:** Requiring, at most, a few gigabytes of memory rather than tens or hundreds of gigabytes.
3. **Faster inference time:** Can generate responses in milliseconds to seconds on edge devices.
4. **Energy efficiency:** Consuming less power, making them suitable for battery-powered devices.
5. **Privacy-preserving:** Enabling on-device processing without sending data to cloud servers.

6. Offline functionality: Operating without an internet connection.

SLMs achieve their compact size through various techniques such as knowledge distillation, model pruning, and quantization. While they may not match the broad capabilities of larger models, SLMs excel in specific tasks and domains, making them ideal for targeted applications on edge devices.

We will generally consider SLMs, language models with less than 5 billion parameters quantized to 4 bits.

Examples of SLMs include compressed versions of models like Meta Llama, Microsoft PHI, and Google Gemma. These models enable a wide range of natural language processing tasks directly on edge devices, from text classification and sentiment analysis to question answering and limited text generation.

For more information on SLMs, the paper, LLM Pruning and Distillation in Practice: The Minitron Approach, provides an approach applying pruning and distillation to obtain SLMs from LLMs. And, SMALL LANGUAGE MODELS: SURVEY, MEASUREMENTS, AND INSIGHTS, presents a comprehensive survey and analysis of Small Language Models (SLMs), which are language models with 100 million to 5 billion parameters designed for resource-constrained devices.

Ollama



Figure 1.28: ollama logo

Ollama is an open-source framework that allows us to run language models (LMs), large or small, locally on our machines. Here are some critical points about Ollama:

1. **Local Model Execution:** Ollama enables running LMs on personal computers or edge devices such as the Raspi-5, eliminating the need for cloud-based API calls.
2. **Ease of Use:** It provides a simple command-line interface for downloading, running, and managing different language models.
3. **Model Variety:** Ollama supports various LLMs, including Phi, Gemma, Llama, Mistral, and other open-source models.
4. **Customization:** Users can create and share custom models tailored to specific needs or domains.
5. **Lightweight:** Designed to be efficient and run on consumer-grade hardware.

6. **API Integration:** Offers an API that allows integration with other applications and services.
7. **Privacy-Focused:** By running models locally, it addresses privacy concerns associated with sending data to external servers.
8. **Cross-Platform:** Available for macOS, Windows, and Linux systems (our case, here).
9. **Active Development:** Regularly updated with new features and model support.
10. **Community-Driven:** Benefits from community contributions and model sharing.

To learn more about what Ollama is and how it works under the hood, you should see this short video from Matt Williams, one of the founders of Ollama:

<https://www.youtube.com/embed/90ozfdsQOKo>

Matt has an entirely free course about Ollama that we recommend: <https://youtu.be/9KEUFe4KQAI?si=D-q3CMbHiT-twuy>

Installing Ollama

Let's set up and activate a Virtual Environment for working with Ollama:

```
python3 -m venv ~/ollama  
source ~/ollama/bin/activate
```

And run the command to install Ollama:

```
curl -fsSL https://ollama.com/install.sh | sh
```

As a result, an API will run in the background on 127.0.0.1:11434. From now on, we can run Ollama via the terminal. For starting, let's verify the Ollama version, which will also tell us that it is correctly installed:

```
ollama -v
```

```
marcelo_rovai@raspi-5: ~ ssh mjrovai@192.168.4.209 - 80x21
mjrovai@raspi-5: ~ $ python3 -m venv ~/ollama
mjrovai@raspi-5: ~ $ source ~/ollama/bin/activate
(ollama) mjrovai@raspi-5: ~ $ curl -fsSL https://ollama.com/install.sh | sh
>>> Installing ollama to /usr/local
>>> Downloading Linux arm64 bundle
#####
# 100.0%#
#####
# 100.0%#
>>> Creating ollama user...
>>> Adding ollama user to render group...
>>> Adding ollama user to video group...
>>> Adding current user to ollama group...
>>> Creating ollama systemd service...
>>> Enabling and starting ollama service...
Created symlink /etc/systemd/system/default.target.wants/ollama.service → /etc/
systemd/system/ollama.service.
>>> The Ollama API is now available at 127.0.0.1:11434.
>>> Install complete. Run "ollama" from the command line.
WARNING: No NVIDIA/AMD GPU detected. Ollama will run in CPU-only mode.
(ollama) mjrovai@raspi-5: ~ $ ollama -v
ollama version is 0.3.11
(ollama) mjrovai@raspi-5: ~ $
```

On the Ollama Library page, we can find the models Ollama supports. For example, by filtering by Most popular, we can see Meta Llama, Google Gemma, Microsoft Phi, LLaVa, etc.

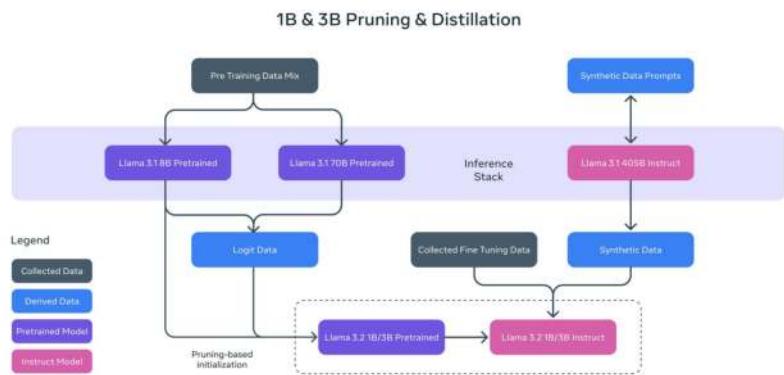
Meta Llama 3.2 1B/3B



Let's install and run our first small language model, Llama 3.2 1B (and 3B). The Meta Llama 3.2 series comprises a set of multilingual generative language models available in 1 billion and 3 billion parameter sizes. These models are designed to process text input and generate text output. The instruction-tuned variants within this collection are specifically optimized for multilingual conversational applications, including tasks involving information retrieval and summarization with an agentic approach. When compared to many existing open-source and proprietary chat models, the Llama 3.2 instruction-tuned models

demonstrate superior performance on widely-used industry benchmarks.

The 1B and 3B models were pruned from the Llama 8B, and then logits from the 8B and 70B models were used as token-level targets (token-level distillation). Knowledge distillation was used to recover performance (they were trained with 9 trillion tokens). The 1B model has 1,24B, quantized to integer (Q8_0), and the 3B, 3.12B parameters, with a Q4_0 quantization, which ends with a size of 1.3 GB and 2 GB, respectively. Its context window is 131,072 tokens.



Install and run the Model

`ollama run llama3.2:1b`

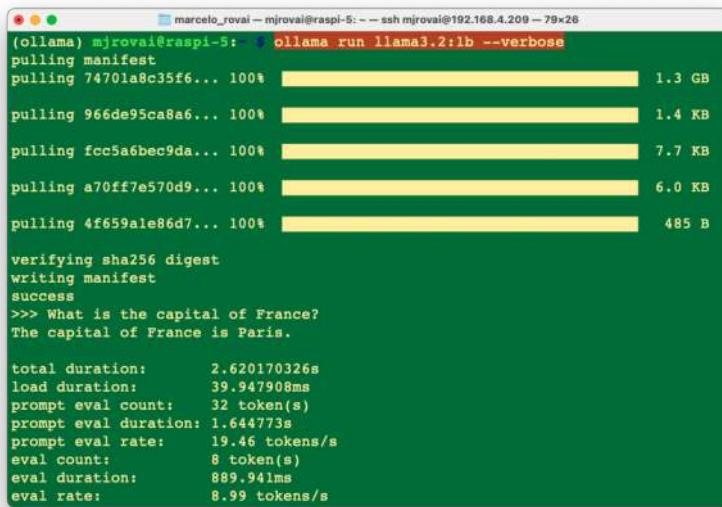
Running the model with the command before, we should have the Ollama prompt available for us to input a question and start chatting with the LLM model; for example,

`>>> What is the capital of France?`

Almost immediately, we get the correct answer:

`The capital of France is Paris.`

Using the option `--verbose` when calling the model will generate several statistics about its performance (The model will be polling only the first time we run the command).



```
marcelo_roval@mjrovai@raspi-5: ~ ssh mjrovai@192.168.4.209 -T 26
(ollama) mjrovai@raspi-5: ~ $ ollama run llama3.2:1b --verbose
pulling manifest
pulling 74701a8c35f6... 100% [██████████] 1.3 GB
pulling 966de95ca8a6... 100% [██████████] 1.4 KB
pulling fcc5a6bec9da... 100% [██████████] 7.7 KB
pulling a70ff7e570d9... 100% [██████████] 6.0 KB
pulling 4f659ale86d7... 100% [██████████] 485 B

verifying sha256 digest
writing manifest
success
>>> What is the capital of France?
The capital of France is Paris.

total duration:      2.620170326s
load duration:      39.947908ms
prompt eval count:   32 token(s)
prompt eval duration: 1.644773s
prompt eval rate:    19.46 tokens/s
eval count:          8 token(s)
eval duration:       889.941ms
eval rate:           8.99 tokens/s
```

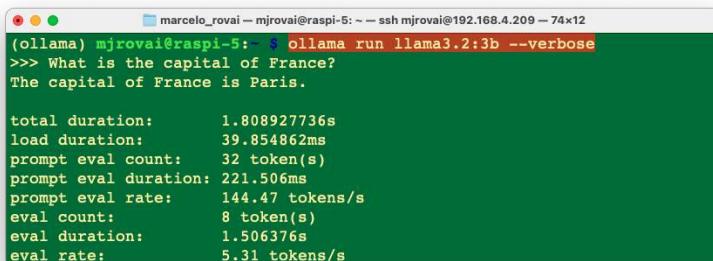
Each metric gives insights into how the model processes inputs and generates outputs. Here's a breakdown of what each metric means:

- **Total Duration (2.620170326 s):** This is the complete time taken from the start of the command to the completion of the response. It encompasses loading the model, processing the input prompt, and generating the response.
- **Load Duration (39.947908 ms):** This duration indicates the time to load the model or necessary components into memory. If this value is minimal, it can suggest that the model was preloaded or that only a minimal setup was required.
- **Prompt Eval Count (32 tokens):** The number of tokens in the input prompt. In NLP, tokens are typically words or subwords, so this count includes all the tokens that the model evaluated to understand and respond to the query.
- **Prompt Eval Duration (1.644773 s):** This measures the model's time to evaluate or process the input prompt. It accounts for the bulk of the total duration, implying that understanding the query and preparing a response is the most time-consuming part of the process.
- **Prompt Eval Rate (19.46 tokens/s):** This rate indicates how quickly the model processes tokens from the input prompt. It reflects the model's speed in terms of natural language comprehension.

- **Eval Count (8 token(s)):** This is the number of tokens in the model's response, which in this case was, "The capital of France is Paris."
- **Eval Duration (889.941 ms):** This is the time taken to generate the output based on the evaluated input. It's much shorter than the prompt evaluation, suggesting that generating the response is less complex or computationally intensive than understanding the prompt.
- **Eval Rate (8.99 tokens/s):** Similar to the prompt eval rate, this indicates the speed at which the model generates output tokens. It's a crucial metric for understanding the model's efficiency in output generation.

This detailed breakdown can help understand the computational demands and performance characteristics of running SLMs like Llama on edge devices like the Raspberry Pi 5. It shows that while prompt evaluation is more time-consuming, the actual generation of responses is relatively quicker. This analysis is crucial for optimizing performance and diagnosing potential bottlenecks in real-time applications.

Loading and running the 3B model, we can see the difference in performance for the same prompt;



```
marcelo_rovai@mjrovai@raspi-5: ~ ssh mjrovai@192.168.4.209 -t 7x12
(ollama) mjrovai@raspi-5:~$ ollama run llama3.2:3b --verbose
>>> What is the capital of France?
The capital of France is Paris.

total duration:      1.808927736s
load duration:      39.854862ms
prompt eval count:  32 token(s)
prompt eval duration: 221.506ms
prompt eval rate:    144.47 tokens/s
eval count:          8 token(s)
eval duration:       1.506376s
eval rate:           5.31 tokens/s
```

The eval rate is lower, 5.3 tokens/s versus 9 tokens/s with the smaller model.

When question about

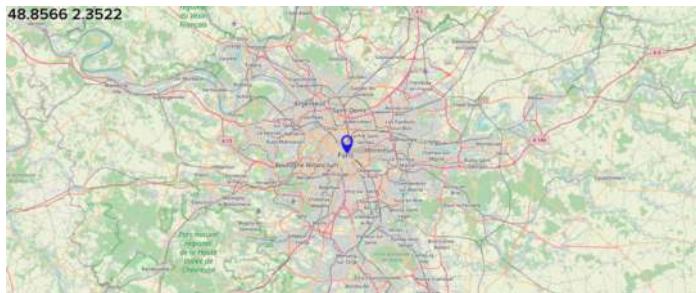
```
>>> What is the distance between Paris and Santiago, Chile?
```

The 1B model answered 9,841 kilometers (6,093 miles), which is inaccurate, and the 3B model answered 7,300 miles (11,700 km), which is close to the correct (11,642 km).

Let's ask for the Paris's coordinates:

```
>>> what is the latitude and longitude of Paris?
```

The latitude and longitude of Paris are 48.8567° N ([48° 55'](#) 42" N) and 2.3510° E (2°22' 8" E), respectively.



Both 1B and 3B models gave correct answers.

Google Gemma 2 2B

Let's install Gemma 2, a high-performing and efficient model available in three sizes: 2B, 9B, and 27B. We will install **Gemma 2 2B**, a lightweight model trained with 2 trillion tokens that produces outsized results by learning from larger models through distillation. The model has 2.6 billion parameters and a Q4_0 quantization, which ends with a size of 1.6 GB. Its context window is 8,192 tokens.



Install and run the Model

```
ollama run gemma2:2b --verbose
```

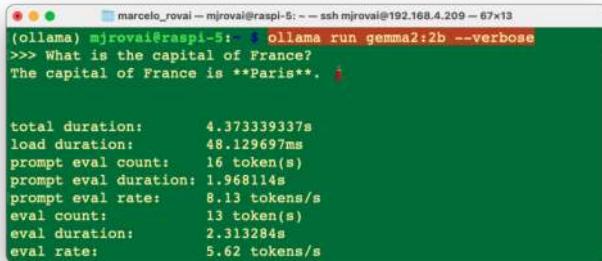
Running the model with the command before, we should have the Ollama prompt available for us to input a question and start chatting with the LLM model; for example,

```
>>> What is the capital of France?
```

Almost immediately, we get the correct answer:

The capital of France is **Paris**.

And it's statistics.



```
marcelo_reval -- miroval@raspi-5: ~ ssh miroval@192.168.4.209 ~ 67x13
(ollama) miroval@raspi-5: ~ $ ollama run gemma2:2b --verbose
>>> What is the capital of France?
The capital of France is **Paris**. ⌁

total duration:      4.373339337s
load duration:       48.129697ms
prompt eval count:   16 token(s)
prompt eval duration: 1.968114s
prompt eval rate:    8.13 tokens/s
eval count:          13 token(s)
eval duration:       2.313284s
eval rate:           5.62 tokens/s
```

We can see that Gemma 2:2B has around the same performance as Llama 3.2:3B, but having less parameters.

Other examples:

>>> What **is** the distance between Paris and Santiago, Chile?

The distance between Paris, France and Santiago, Chile is approximately **7,000 miles (11,267 kilometers)**.

Keep in mind that this is a straight-line distance, and actual travel distance can vary depending on the chosen routes and any stops along the way.

Also, a good response but less accurate than Llama3.2:3B.

>>> what **is** the latitude and longitude of Paris?

You got it! Here are the latitudes and longitudes of Paris, France:

* **Latitude**: 48.8566° N (north)
* **Longitude**: 2.3522° E (east)

Let me know if you'd like to explore more about Paris or its location!

A good and accurate answer (a little more verbose than the Llama answers).

Microsoft Phi3.5 3.8B

Let's pull a bigger (but still tiny) model, the PHI3.5, a 3.8B lightweight state-of-the-art open model by Microsoft. The model belongs to the Phi-3 model family and supports 128K token context length and the languages: Arabic, Chinese, Czech, Danish, Dutch, English, Finnish, French, German, Hebrew, Hungarian, Italian, Japanese, Korean, Norwegian, Polish, Portuguese, Russian, Spanish, Swedish, Thai, Turkish and Ukrainian.

The model size, in terms of bytes, will depend on the specific quantization format used. The size can go from 2-bit quantization (`q2_k`) of 1.4 GB (higher performance/lower quality) to 16-bit quantization (`fp-16`) of 7.6 GB (lower performance/higher quality).

Let's run the 4-bit quantization (`Q4_0`), which will need 2.2 GB of RAM, with an intermediary trade-off regarding output quality and performance.

```
ollama run phi3.5:3.8b --verbose
```

You can use `run` or `pull` to download the model. What happens is that Ollama keeps note of the pulled models, and once the PHI3 does not exist, before running it, Ollama pulls it.

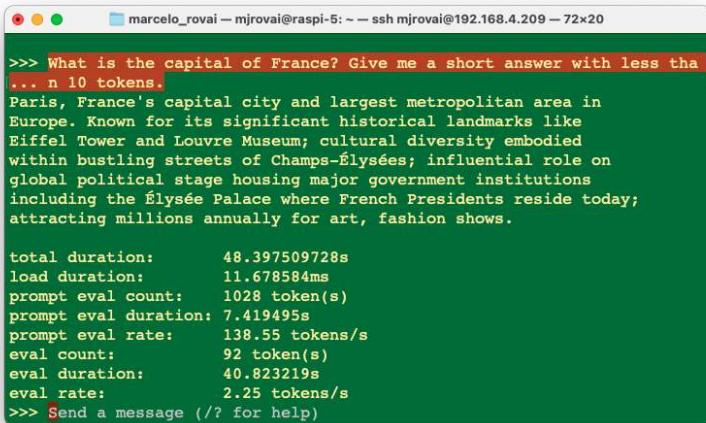
Let's enter with the same prompt used before:

```
>>> What is the capital of France?
```

The capital of France is Paris. It's extradites significant historical, cultural, and political importance to the country as well as being a major European city known for its art, fashion, gastronomy, and culture. Its influence extends beyond national borders, with millions of tourists visiting each year from around the globe. The Seine River flows through Paris before it reaches the broader English Channel at Le Havre. Moreover, France is one of Europe's leading economies with its capital playing a key role

...

The answer was very "verbose", let's specify a better prompt:



```
marcelo_rovai — mjrovai@raspi-5: ~ — ssh mjrovai@192.168.4.209 — 72x20
>>> What is the capital of France? Give me a short answer with less than
... n 10 tokens.
Paris, France's capital city and largest metropolitan area in Europe. Known for its significant historical landmarks like Eiffel Tower and Louvre Museum; cultural diversity embodied within bustling streets of Champs-Élysées; influential role on global political stage housing major government institutions including the Élysée Palace where French Presidents reside today; attracting millions annually for art, fashion shows.

total duration:      48.397509728s
load duration:       11.678584ms
prompt eval count:   1028 token(s)
prompt eval duration: 7.419495s
prompt eval rate:    138.55 tokens/s
eval count:          92 token(s)
eval duration:       40.823219s
eval rate:           2.25 tokens/s
>>> Send a message (/? for help)
```

In this case, the answer was still longer than we expected, with an eval rate of 2.25 tokens/s, more than double that of Gemma and Llama.

Choosing the most appropriate prompt is one of the most important skills to be used with LLMs, no matter its size.

When we asked the same questions about distance and Latitude/Longitude, we did not get a good answer for a distance of 13,507 kilometers (8,429 miles), but it was OK for coordinates. Again, it could have been less verbose (more than 200 tokens for each answer).

We can use any model as an assistant since their speed is relatively decent, but on September 24 (2023), the Llama2:3B is a better choice. You should try other models, depending on your needs. Open LLM Leaderboard can give you an idea about the best models in size, benchmark, license, etc.

The best model to use is the one fit for your specific necessity. Also, take into consideration that this field evolves with new models everyday.

Multimodal Models

Multimodal models are artificial intelligence (AI) systems that can process and understand information from multiple sources, such as images, text, audio, and video. In our context, multimodal LLMs can process

various inputs, including text, images, and audio, as prompts and convert those prompts into various outputs, not just the source type.

We will work here with LLaVA-Phi-3, a fine-tuned LLaVA model from Phi 3 Mini 4k. It has strong performance benchmarks that are on par with the original LLaVA (Large Language and Vision Assistant) model.

The LLaVA-Phi-3 is an end-to-end trained large multimodal model designed to understand and generate content based on visual inputs (images) and textual instructions. It combines the capabilities of a visual encoder and a language model to process and respond to multimodal inputs.

Let's install the model:

```
ollama run llava-phi3:3.8b --verbose
```

Let's start with a text input:

```
>>> You are a helpful AI assistant.  
      What is the capital of France?
```

As an AI language model, I can tell you that the capital of France is Paris. It's not only the largest city in the country but also serves as its political and administrative center. Paris is known for its iconic landmarks such as the Eiffel Tower, Notre-Dame Cathedral, and the Louvre Museum. The city has a rich history, beautiful architecture, and is widely considered to be one of the most romantic cities in the world.

The response took around 30 s, with an eval rate of 3.93 tokens/s! Not bad!

But let us know to enter with an image as input. For that, let's create a directory for working:

```
cd Documents/  
mkdir OLLAMA  
cd OLLAMA
```

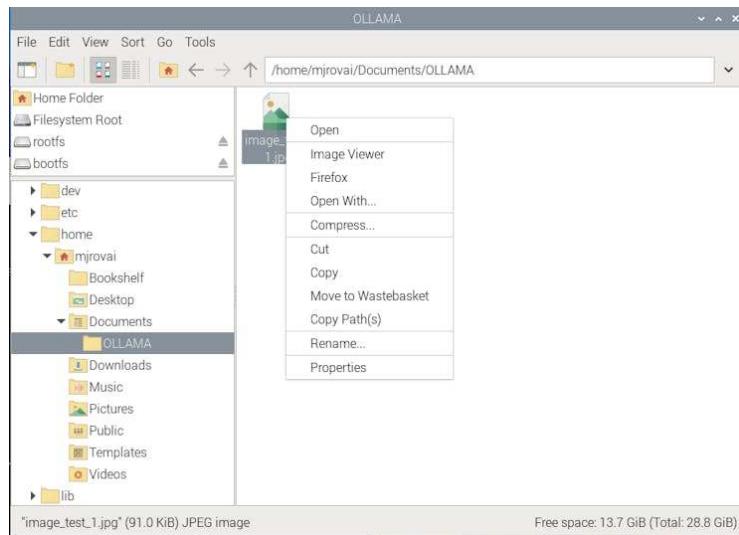
Let's download a 640×320 image from the internet, for example (Wikipedia: Paris, France):



Using FileZilla, for example, let's upload the image to the OLLAMA folder at the Raspi-5 and name it `image_test_1.jpg`. We should have the whole image path (we can use `pwd` to get it).

```
/home/mjrovai/Documents/OLLAMA/image_test_1.jpg
```

If you use a desktop, you can copy the image path by clicking the image with the mouse's right button.



Let's enter with this prompt:

```
>>> Describe the image /home/mjrovai/Documents/OLLAMA\  
          image_test_1.jpg
```

The result was great, but the overall latency was significant; almost 4 minutes to perform the inference.

Inspecting local resources

Using htop, we can monitor the resources running on our device.

htop

During the time that the model is running, we can inspect the resources:

All four CPUs run at almost 100% of their capacity, and the memory used with the model loaded is 3.24 GB. Exiting Ollama, the memory goes down to around 377 MB (with no desktop).

It is also essential to monitor the temperature. When running the Raspberry with a desktop, you can have the temperature shown on the taskbar:



If you are “headless”, the temperature can be monitored with the command:

```
vcgencmd measure_temp
```

If you are doing nothing, the temperature is around 50°C for CPUs running at 1%. During inference, with the CPUs at 100%, the temperature can rise to almost 70°C. This is OK and means the active cooler is working, keeping the temperature below 80°C / 85°C (its limit).

Ollama Python Library

So far, we have explored SLMs’ chat capability using the command line on a terminal. However, we want to integrate those models into our projects, so Python seems to be the right path. The good news is that Ollama has such a library.

The Ollama Python library simplifies interaction with advanced LLM models, enabling more sophisticated responses and capabilities, besides providing the easiest way to integrate Python 3.8+ projects with Ollama.

For a better understanding of how to create apps using Ollama with Python, we can follow Matt Williams’s videos, as the one below:

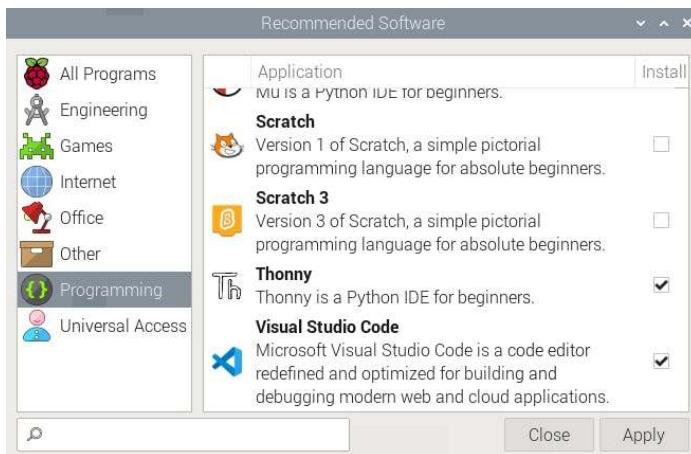
https://www.youtube.com/embed/_4K20tOsXK8

Installation:

In the terminal, run the command:

```
pip install ollama
```

We will need a text editor or an IDE to create a Python script. If you run the Raspberry OS on a desktop, several options, such as Thonny and Geany, have already been installed by default (accessed by [Menu] [Programming]). You can download other IDEs, such as Visual Studio Code, from [Menu] [Recommended Software]. When the window pops up, go to [Programming], select the option of your choice, and press [Apply].



If you prefer using Jupyter Notebook for development:

```
pip install jupyter  
jupyter notebook --generate-config
```

To run Jupyter Notebook, run the command (change the IP address for yours):

```
jupyter notebook --ip=192.168.4.209 --no-browser
```

On the terminal, you can see the local URL address to open the notebook:

```
(olima) mjroval@raspi-5: ~ - ssh mjroval@192.168.4.209 - 130x31
[mjroval@raspi-5 ~]$ jupyter notebook --no-browser
[ServerApp] Jupyter Lab | extension was successfully linked.
[2014-09-25 15:23:03.768] [ServerApp] Jupyter Server/Terminals | extension was successfully linked.
[2014-09-25 15:23:03.772] [ServerApp] JupyterLab | extension was successfully linked.
[2014-09-25 15:23:03.778] [ServerApp] JupyterLab | extension was successfully linked.
[2014-09-25 15:23:04.022] [ServerApp] notebook_shim | extension was successfully linked.
[2014-09-25 15:23:04.034] [ServerApp] notebook_shim | extension was successfully loaded.
[2014-09-25 15:23:04.038] [ServerApp] Jupyter Lab | extension was successfully loaded.
[2014-09-25 15:23:04.037] [ServerApp] Jupyter Server/Terminals | extension was successfully loaded.
[2014-09-25 15:23:04.038] [ServerApp] JupyterLab | extension was successfully loaded.
[2014-09-25 15:23:04.038] [ServerApp] JupyterLab application directory is '/home/mjroval/olima/share/jupyter/lab'.
[2014-09-25 15:23:04.079] [Labapp] Extension Manager is 'pypi'.
[2014-09-25 15:23:04.082] [ServerApp] Jupyter Lab | extension was successfully loaded.
[2014-09-25 15:23:04.085] [ServerApp] notebook | extension was successfully loaded.
[2014-09-25 15:23:04.085] [ServerApp] Serving Notebooks | local storage directory '/home/mjroval/notebooks'.
[2014-09-25 15:23:04.085] [ServerApp] JupyterLab | extension was successfully loaded.
[2014-09-25 15:23:04.085] [ServerApp] Jupyter Server/Terminals | extension was successfully loaded.
[2014-09-25 15:23:04.085] [ServerApp] JupyterLab application directory is '/home/mjroval/olima/share/jupyter/lab'.
[2014-09-25 15:23:04.085] [ServerApp] https://12.198.4.209:8888/tree/token?79a89d699915e1d357cd5da114d3508ed3ed171a422
[2014-09-25 15:23:04.085] [ServerApp] http://172.0.1.1:8888/tree/token?79a89d699915e1d357cd5da114d3508ed3ed171a422
[2014-09-25 15:23:04.085] [ServerApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[2014-09-25 15:23:04.085] [ServerApp] 

To access the server, open this file in a browser:
file:///home/mjroval/.local/share/jupyter/runtime/jpserver/runtime/18031-open.html
Or copy and paste one of these URLs:
http://12.198.4.209:8888/tree/token?79a89d699915e1d357cd5da114d3508ed3ed171a422
http://127.0.1.1:8888/tree/token?79a89d699915e1d357cd5da114d3508ed3ed171a422
13:01:11 [mjroval@raspi-5 ~]$ jupyter labextension install --user --no-build
jupyterlab-language-server, jedi-language-server, julia-language-server, pygments, python-language-server, python-lsp-server, vancscript-typecript-language-server, sgqlc-language-server, texlab, typescript-language-server, unified-language-server, vscode-css-languageserver-bin, vscode-html-languageserver-bin, vscode-javascript-languageserver-bin, yaml-language-server
```

We can access it from another computer by entering the Raspberry Pi's IP address and the provided token in a web browser (we should copy it from the terminal).

In our working directory in the Raspi, we will create a new Python 3 notebook.

Let's enter with a very simple script to verify the installed models:

```
import ollama
```

```
ollama.list()
```

All the models will be printed as a dictionary, for example:

```
{'name': 'gemma2:2b',
'model': 'gemma2:2b',
'modified_at': '2024-09-24T19:30:40.053898094+01:00',
'size': 1629518495,
'digest': (
    '8ccf136fdd5298f3ffe2d69862750ea7fb56555fa4d5b18c0'
    '4e3fa4d82ee09d7'
),
'details': {'parent_model': '',
'format': 'gguf',
'family': 'gemma2',
'families': ['gemma2'],
'parameter_size': '2.6B',
'quantization_level': 'Q4_0'}}}
```

Let's repeat one of the questions that we did before, but now using `ollama.generate()` from Ollama python library. This API will generate a response for the given prompt with the provided model. This is a streaming endpoint, so there will be a series of responses. The final response object will include statistics and additional data from the request.

```
MODEL = "gemma2:2b"
PROMPT = "What is the capital of France?"

res = ollama.generate(model=MODEL, prompt=PROMPT)
print(res)
```

In case you are running the code as a Python script, you should save it, for example, `test_ollama.py`. You can use the IDE to run it or do it directly on the terminal. Also, remember that you should always call the model and define it when running a stand-alone script.

```
python test_ollama.py
```

As a result, we will have the model response in a JSON format:

```
{
  'model': 'gemma2:2b',
  'created_at': '2024-09-25T14:43:31.869633807Z',
  'response': 'The capital of France is **Paris**.\n',
  'done': True,
  'done_reason': 'stop',
  'context': [
    106, 1645, 108, 1841, 603, 573, 6037, 576, 6081, 235336,
    107, 108, 106, 2516, 108, 651, 6037, 576, 6081, 603, 5231,
    29437, 168428, 235248, 244304, 241035, 235248, 108
  ],
  'total_duration': 24259469458,
  'load_duration': 19830013859,
  'prompt_eval_count': 16,
  'prompt_eval_duration': 1908757000,
  'eval_count': 14,
  'eval_duration': 2475410000
}
```

As we can see, several pieces of information are generated, such as:

- **response:** the main output text generated by the model in response to our prompt.
 - The capital of France is **Paris**.

- **context**: the token IDs representing the input and context used by the model. Tokens are numerical representations of text used for processing by the language model.
 - [106, 1645, 108, 1841, 603, 573, 6037, 576, 6081, 235336, 107, 108, 106, 2516, 108, 651, 6037, 576, 6081, 603, 5231, 29437, 168428, 235248, 244304, 241035, 235248, 108]

The Performance Metrics:

- **total_duration**: The total time taken for the operation in nanoseconds. In this case, approximately 24.26 seconds.
- **load_duration**: The time taken to load the model or components in nanoseconds. About 19.83 seconds.
- **prompt_eval_duration**: The time taken to evaluate the prompt in nanoseconds. Around 16 nanoseconds.
- **eval_count**: The number of tokens evaluated during the generation. Here, 14 tokens.
- **eval_duration**: The time taken for the model to generate the response in nanoseconds. Approximately 2.5 seconds.

But, what we want is the plain ‘response’ and, perhaps for analysis, the total duration of the inference, so let’s change the code to extract it from the dictionary:

```
print(f"\n{res['response']}")  
print(  
    f"\n [INFO] Total Duration: "  
    f"{res['total_duration']/1e9:.2f} seconds"  
)
```

Now, we got:

The capital of France is **Paris**.

[INFO] Total Duration: 24.26 seconds

Using Ollama.chat()

Another way to get our response is to use `ollama.chat()`, which generates the next message in a chat with a provided model. This is a streaming endpoint, so a series of responses will occur. Streaming can be disabled using `"stream": false`. The final response object will also include statistics and additional data from the request.

```
PROMPT_1 = "What is the capital of France?"  
  
response = ollama.chat(  
    model=MODEL,  
    messages=[  
        {  
            "role": "user",  
            "content": PROMPT_1,  
        },  
    ],  
)  
resp_1 = response["message"]["content"]  
print(f"\n{resp_1}")  
print(  
    f"\n [INFO] Total Duration: "  
    f"{(res['total_duration']/1e9):.2f} seconds"  
)
```

The answer is the same as before.

An important consideration is that by using `ollama.generate()`, the response is “clear” from the model’s “memory” after the end of inference (only used once), but If we want to keep a conversation, we must use `ollama.chat()`. Let’s see it in action:

```
PROMPT_1 = "What is the capital of France?"  
response = ollama.chat(  
    model=MODEL,  
    messages=[  
        {  
            "role": "user",  
            "content": PROMPT_1,  
        },  
    ],  
)  
resp_1 = response["message"]["content"]  
print(f"\n{resp_1}")  
print(  
    f"\n [INFO] Total Duration: "  
    f"{{(response['total_duration']/1e9):.2f} seconds"  
)  
  
PROMPT_2 = "and of Italy?"  
response = ollama.chat(  
    model=MODEL,
```

```

messages=[
    {
        "role": "user",
        "content": PROMPT_1,
    },
    {
        "role": "assistant",
        "content": resp_1,
    },
    {
        "role": "user",
        "content": PROMPT_2,
    },
],
)
resp_2 = response["message"]["content"]
print(f"\n{resp_2}")
print(
    f"\n [INFO] Total Duration: "
    f"{(response_2['total_duration']/1e9):.2f} seconds"
)

```

In the above code, we are running two queries, and the second prompt considers the result of the first one.

Here is how the model responded:

The capital of France is **Paris**.

[INFO] Total Duration: 2.82 seconds

The capital of Italy is **Rome**.

[INFO] Total Duration: 4.46 seconds

Getting an image description:

In the same way that we have used the LlaVa-PHI-3 model with the command line to analyze an image, the same can be done here with Python. Let's use the same image of Paris, but now with the `ollama.generate()`:

```

MODEL = "llava-phi3:3.8b"
PROMPT = "Describe this picture"

with open("image_test_1.jpg", "rb") as image_file:

```

```
img = image_file.read()

response = ollama.generate(model=MODEL, prompt=PROMPT, images=[img])
print(f"\n{response['response']}")

print(
    f"\n [INFO] Total Duration: "
    f"{(res['total_duration']/1e9):.2f} seconds"
)
```

Here is the result:

This image captures the iconic cityscape of Paris, France. The vantage point is high, providing a panoramic view of the Seine River that meanders through the heart of the city. Several bridges arch gracefully over the river, connecting different parts of the city. The Eiffel Tower, an iron lattice structure with a pointed top and two antennas on its summit, stands tall in the background, piercing the sky. It is painted in a light gray color, contrasting against the blue sky speckled with white clouds.

The buildings that line the river are predominantly white or beige, their uniform color palette broken occasionally by red roofs peeking through. The Seine River itself appears calm and wide, reflecting the city's architectural beauty in its surface. On either side of the river, trees add a touch of green to the urban landscape.

The image is taken from an elevated perspective, looking down on the city. This viewpoint allows for a comprehensive view of Paris's beautiful architecture and layout. The relative positions of the buildings, bridges, and other structures create a harmonious composition that showcases the city's charm.

In summary, this image presents a serene day in Paris, with its architectural marvels - from the Eiffel Tower to the river-side buildings - all bathed in soft colors under a clear sky.

[INFO] Total Duration: 256.45 seconds

The model took about 4 minutes (256.45 s) to return with a detailed image description.

In the 10-Ollama_Python_Library notebook, it is possible to find the experiments with the Ollama Python library.

Function Calling

So far, we can observe that by using the model's response into a variable, we can effectively incorporate it into real-world projects. However, a major issue arises when the model provides varying responses to the same input. For instance, let's assume that we only need the name of a country's capital and its coordinates as the model's response in the previous examples, without any additional information, even when utilizing verbose models like Microsoft Phi. To ensure consistent responses, we can employ the 'Ollama function call,' which is fully compatible with the OpenAI API.

But what exactly is “function calling”?

In modern artificial intelligence, function calling with Large Language Models (LLMs) allows these models to perform actions beyond generating text. By integrating with external functions or APIs, LLMs can access real-time data, automate tasks, and interact with various systems.

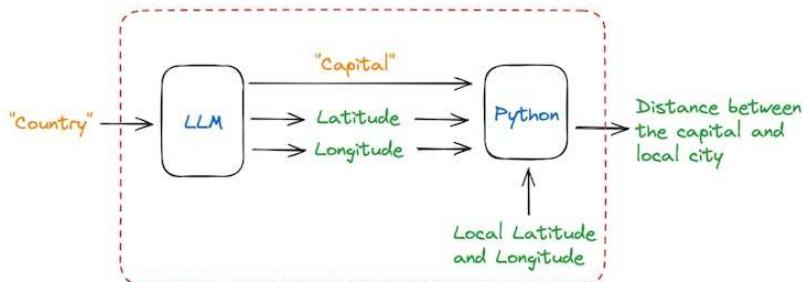
For instance, instead of merely responding to a query about the weather, an LLM can call a weather API to fetch the current conditions and provide accurate, up-to-date information. This capability enhances the relevance and accuracy of the model's responses and makes it a powerful tool for driving workflows and automating processes, transforming it into an active participant in real-world applications.

For more details about Function Calling, please see this video made by Marvin Prison:

<https://www.youtube.com/embed/eHfMCtsb1o>

Let's create a project.

We want to create an *app* where the user enters a country's name and gets, as an output, the distance in km from the capital city of such a country and the app's location (for simplicity, We will use Santiago, Chile, as the app location).



Once the user enters a country name, the model will return the name of its capital city (as a string) and the latitude and longitude of such city (in float). Using those coordinates, we can use a simple Python library (haversine) to calculate the distance between those 2 points.

The idea of this project is to demonstrate a combination of language model interaction, structured data handling with Pydantic, and geospatial calculations using the Haversine formula (traditional computing).

First, let us install some libraries. Besides *Haversine*, the main one is the OpenAI Python library, which provides convenient access to the OpenAI REST API from any Python 3.7+ application. The other one is Pydantic (and instructor), a robust data validation and settings management library engineered by Python to enhance the robustness and reliability of our codebase. In short, *Pydantic* will help ensure that our model's response will always be consistent.

```
pip install haversine
pip install openai
pip install pydantic
pip install instructor
```

Now, we should create a Python script designed to interact with our model (LLM) to determine the coordinates of a country's capital city and calculate the distance from Santiago de Chile to that capital.

Let's go over the code:

1. Importing Libraries

```
import sys
from haversine import haversine
from openai import OpenAI
from pydantic import BaseModel, Field
import instructor
```

- **sys**: Provides access to system-specific parameters and functions. It's used to get command-line arguments.
- **haversine**: A function from the haversine library that calculates the distance between two geographic points using the Haversine formula.
- **openAI**: A module for interacting with the OpenAI API (although it's used in conjunction with a local setup, Ollama). Everything is off-line here.
- **pydantic**: Provides data validation and settings management using Python-type annotations. It's used to define the structure of expected response data.
- **instructor**: A module is used to patch the OpenAI client to work in a specific mode (likely related to structured data handling).

2. Defining Input and Model

```
country = sys.argv[1] # Get the country from
# command-line arguments
MODEL = "phi3.5:3.8b" # The name of the model to be used
mylat = -33.33 # Latitude of Santiago de Chile
mylon = -70.51 # Longitude of Santiago de Chile
```

- **country**: On a Python script, getting the country name from command-line arguments is possible. On a Jupyter notebook, we can enter its name, for example,
 - country = "France"
- **MODEL**: Specifies the model being used, which is, in this example, the phi3.5.
- **mylat and mylon**: Coordinates of Santiago de Chile, used as the starting point for the distance calculation.

3. Defining the Response Data Structure

```
class CityCoord(BaseModel):
    city: str = Field(..., description="Name of the city")
    lat: float = Field(
        ..., description="Decimal Latitude of the city"
    )
    lon: float = Field(
        ..., description="Decimal Longitude of the city"
    )
```

- **CityCoord**: A Pydantic model that defines the expected structure of the response from the LLM. It expects three fields: city (name of the city), lat (latitude), and lon (longitude).

4. Setting Up the OpenAI Client

```
client = instructor.patch(  
    OpenAI(  
        base_url="http://localhost:11434/v1", # Local API base  
        # URL (Ollama)  
        api_key="ollama", # API key  
        # (not used)  
    ),  
    mode=instructor.Mode.JSON, # Mode for  
    # structured  
    # JSON output  
)
```

- **OpenAI**: This setup initializes an OpenAI client with a local base URL and an API key (ollama). It uses a local server.
- **instructor.patch**: Patches the OpenAI client to work in JSON mode, enabling structured output that matches the Pydantic model.

5. Generating the Response

```
resp = client.chat.completions.create(  
    model=MODEL,  
    messages=[  
        {  
            "role": "user",  
            "content": f"return the decimal latitude and \  
decimal longitude of the capital of the {country}.",  
        }  
    ],  
    response_model=CityCoord,  
    max_retries=10,  
)
```

- **client.chat.completions.create**: Calls the LLM to generate a response.
- **model**: Specifies the model to use (llava-phi3).

- **messages**: Contains the prompt for the LLM, asking for the latitude and longitude of the capital city of the specified country.
- **response_model**: Indicates that the response should conform to the CityCoord model.
- **max_retries**: The maximum number of retry attempts if the request fails.

6. Calculating the Distance

```
distance = haversine((mylat, mylon), (resp.lat, resp.lon), unit="km")

print(
    f"Santiago de Chile is about {int(round(distance, -1))} "
    f"kilometers away from {resp.city}."
)
```

- **haversine**: Calculates the distance between Santiago de Chile and the capital city returned by the LLM using their respective coordinates.
- **(mylat, mylon)**: Coordinates of Santiago de Chile.
- **resp.city**: Name of the country's capital
- **(resp.lat, resp.lon)**: Coordinates of the capital city are provided by the LLM response.
- **unit = 'km'**: Specifies that the distance should be calculated in kilometers.
- **print**: Outputs the distance, rounded to the nearest 10 kilometers, with thousands of separators for readability.

Running the code

If we enter different countries, for example, France, Colombia, and the United States, We can note that we always receive the same structured information:

Santiago de Chile is about 8,060 kilometers away from
Washington, D.C..

Santiago de Chile is about 4,250 kilometers away from Bogotá.

Santiago de Chile is about 11,630 kilometers away from Paris.

If you run the code as a script, the result will be printed on the terminal:

```
mjrovai@rpi-5:~/Documents/OLLAMA$ python calc_distance.py "United States"
Santiago de Chile is about 8,060 kilometers away from Washington, D.C..
mjrovai@rpi-5:~/Documents/OLLAMA$ python calc_distance.py "Colombia"
Santiago de Chile is about 4,250 kilometers away from Bogotá.
mjrovai@rpi-5:~/Documents/OLLAMA$ python calc_distance.py "France"
Santiago de Chile is about 11,630 kilometers away from Paris.
mjrovai@rpi-5:~/Documents/OLLAMA$
```

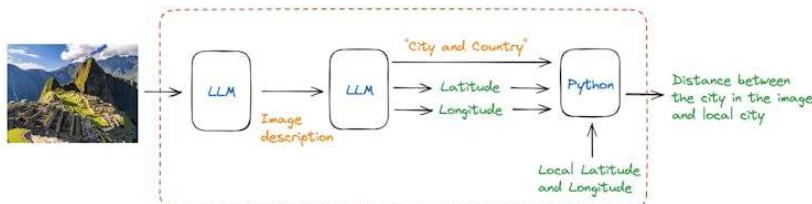
And the calculations are pretty good!



In the 20-Ollama_Function_Calling notebook, it is possible to find experiments with all models installed.

Adding images

Now it is time to wrap up everything so far! Let's modify the script so that instead of entering the country name (as a text), the user enters an image, and the application (based on SLM) returns the city in the image and its geographic location. With those data, we can calculate the distance as before.



For simplicity, we will implement this new code in two steps. First, the LLM will analyze the image and create a description (text). This text will be passed on to another instance, where the model will extract the information needed to pass along.

We will start importing the libraries

```
import sys
import time
from haversine import haversine
import ollama
from openai import OpenAI
from pydantic import BaseModel, Field
import instructor
```

We can see the image if you run the code on the Jupyter Notebook. For that we need also import:

```
import matplotlib.pyplot as plt
from PIL import Image
```

Those libraries are unnecessary if we run the code as a script.

Now, we define the model and the local coordinates:

```
MODEL = "llava-phi3:3.8b"
mylat = -33.33
mylon = -70.51
```

We can download a new image, for example, Machu Picchu from Wikipedia. On the Notebook we can see it:

```
# Load the image
img_path = "image_test_3.jpg"
img = Image.open(img_path)

# Display the image
plt.figure(figsize=(8, 8))
plt.imshow(img)
plt.axis("off")
# plt.title("Image")
plt.show()
```



Now, let's define a function that will receive the image and will return the decimal latitude and decimal longitude of the city in the image, its name, and what country it is located

```
def image_description(img_path):
    with open(img_path, "rb") as file:
        response = ollama.chat(
            model=MODEL,
            messages=[
                {
                    "role": "user",
                    "content": """return the decimal latitude and \
decimal longitude of the city in the image, \
its name, and what country it is located""",
                    "images": [file.read()],
                },
            ],
            options={
                "temperature": 0,
            },
        )
    # print(response['message']['content'])
    return response["message"]["content"]
```

We can print the entire response for debug purposes.

The image description generated for the function will be passed as a prompt for the model again.

```
start_time = time.perf_counter() # Start timing

class CityCoord(BaseModel):
    city: str = Field(
        ...,
        description="Name of the city in the image"
    )
    country: str = Field(
        ...,
        description=(
            "Name of the country where "
            "the city in the image is located"
        ),
    )
    lat: float = Field(
        ...,
        description=("Decimal latitude of the city in " "the image"),
    )
    lon: float = Field(
        ...,
        description=("Decimal longitude of the city in " "the image"),
    )

# enables `response_model` in create call
client = instructor.patch(
    OpenAI(base_url="http://localhost:11434/v1", api_key="ollama"),
    mode=instructor.Mode.JSON,
)

image_description = image_description(img_path)
# Send this description to the model
resp = client.chat.completions.create(
    model=MODEL,
    messages=[
        {
            "role": "user",
            "content": image_description,
        }
    ]
)
```

```
],
response_model=CityCoord,
max_retries=10,
temperature=0,
)
```

If we print the image description , we will get:

The image shows the ancient city of Machu Picchu, located in Peru. The city is perched on a steep hillside and consists of various structures made of stone. It is surrounded by lush greenery and towering mountains. The sky above is blue with scattered clouds.

Machu Picchu's latitude is approximately 13.5086° S, and its longitude is around 72.5494° W.

And the second response from the model (resp) will be:

```
CityCoord(city='Machu Picchu', country='Peru', lat=-13.5086,
          lon=-72.5494)
```

Now, we can do a "Post-Processing", calculating the distance and preparing the final answer:

```
distance = haversine((mylat, mylon), (resp.lat, resp.lon), unit="km")

print(
(
    f"\nThe image shows {resp.city}, with lat: "
    f"{round(resp.lat, 2)} and long: "
    f"{round(resp.lon, 2)}, located in "
    f"{resp.country} and about "
    f"{int(round(distance, -1))}, kilometers "
    f"away from Santiago, Chile.\n"
)
)

end_time = time.perf_counter() # End timing
elapsed_time = end_time - start_time # Calculate elapsed time
print(
    f"[INFO] ==> The code (running {MODEL}), "
    f"took {elapsed_time:.1f} seconds to execute.\n"
)
```

And we will get:

The image shows Machu Picchu, with lat:-13.16 and long: -72.54, located in Peru and about 2,250 kilometers away from Santiago, Chile.

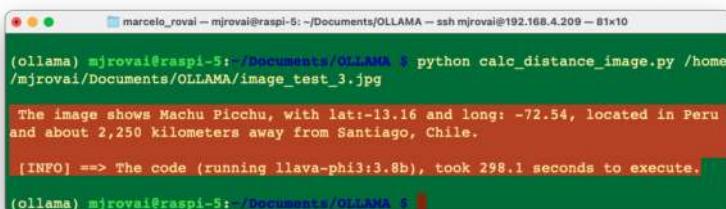
```
print(
    f"[INFO] ==> The code (running {MODEL}), "
    f"took {elapsed_time:.1f} seconds "
    f"to execute.\n"
)
```

In the 30-Function_Calling_with_images notebook, it is possible to find the experiments with multiple images.

Let's now download the script calc_distance_image.py from the GitHub and run it on the terminal with the command:

```
python calc_distance_image.py \
/home/mjrovai/Documents/OLLAMA/image_test_3.jpg
```

Enter with the Machu Picchu image full path as an argument. We will get the same previous result.



```
marcelo_roat ~ mjrovai@raspi-5:~/Documents/OLLAMA ~ ssh mjrovai@192.168.4.209 ~ 81x10
(ollama) mjrovai@raspi-5:~/Documents/OLLAMA $ python calc_distance_image.py /home/mjrovai/Documents/OLLAMA/image_test_3.jpg
The image shows Machu Picchu, with lat:-13.16 and long: -72.54, located in Peru
and about 2,250 kilometers away from Santiago, Chile.
[INFO] ==> The code (running llava-phi3:3.8b), took 298.1 seconds to execute.
(ollama) mjrovai@raspi-5:~/Documents/OLLAMA $
```

How about Paris?



```
marcelo_roat ~ mjrovai@raspi-5:~/Documents/OLLAMA ~ ssh mjrovai@192.168.4.209 ~ 82x10
(ollama) mjrovai@raspi-5:~/Documents/OLLAMA $ python calc_distance_image.py /home/mjrovai/Documents/OLLAMA/image_test_1.jpg
The image shows Paris, with lat:48.86 and long: 2.35, located in France and about
11,630 kilometers away from Santiago, Chile.
[INFO] ==> The code (running llava-phi3:3.8b), took 258.6 seconds to execute.
(ollama) mjrovai@raspi-5:~/Documents/OLLAMA $
```

Of course, there are many ways to optimize the code used here. Still, the idea is to explore the considerable potential of *function calling* with SLMs at the edge, allowing those models to integrate with external

functions or APIs. Going beyond text generation, SLMs can access real-time data, automate tasks, and interact with various systems.

SLMs: Optimization Techniques

Large Language Models (LLMs) have revolutionized natural language processing, but their deployment and optimization come with unique challenges. One significant issue is the tendency for LLMs (and more, the SLMs) to generate plausible-sounding but factually incorrect information, a phenomenon known as **hallucination**. This occurs when models produce content that seems coherent but is not grounded in truth or real-world facts.

Other challenges include the immense computational resources required for training and running these models, the difficulty in maintaining up-to-date knowledge within the model, and the need for domain-specific adaptations. Privacy concerns also arise when handling sensitive data during training or inference. Additionally, ensuring consistent performance across diverse tasks and maintaining ethical use of these powerful tools present ongoing challenges. Addressing these issues is crucial for the effective and responsible deployment of LLMs in real-world applications.

The fundamental techniques for enhancing LLM (and SLM) performance and efficiency are Fine-tuning, Prompt engineering, and Retrieval-Augmented Generation (RAG).

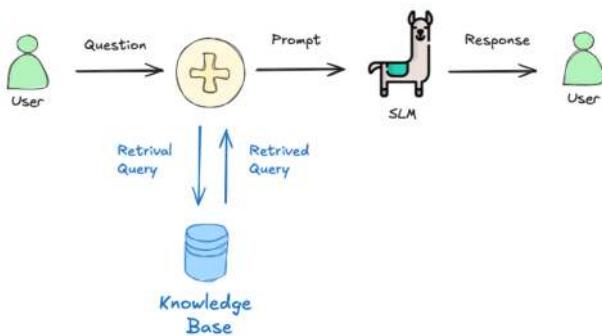
- **Fine-tuning**, while more resource-intensive, offers a way to specialize LLMs for particular domains or tasks. This process involves further training the model on carefully curated datasets, allowing it to adapt its vast general knowledge to specific applications. Fine-tuning can lead to substantial improvements in performance, especially in specialized fields or for unique use cases.
- **Prompt engineering** is at the forefront of LLM optimization. By carefully crafting input prompts, we can guide models to produce more accurate and relevant outputs. This technique involves structuring queries that leverage the model's pre-trained knowledge and capabilities, often incorporating examples or specific instructions to shape the desired response.
- **Retrieval-Augmented Generation (RAG)** represents another powerful approach to improving LLM performance. This method combines the vast knowledge embedded in pre-trained models

with the ability to access and incorporate external, up-to-date information. By retrieving relevant data to supplement the model's decision-making process, RAG can significantly enhance accuracy and reduce the likelihood of generating outdated or false information.

For edge applications, it is more beneficial to focus on techniques like RAG that can enhance model performance without needing on-device fine-tuning. Let's explore it.

RAG Implementation

In a basic interaction between a user and a language model, the user asks a question, which is sent as a prompt to the model. The model generates a response based solely on its pre-trained knowledge. In a RAG process, there's an additional step between the user's question and the model's response. The user's question triggers a retrieval process from a knowledge base.



A simple RAG project

Here are the steps to implement a basic Retrieval Augmented Generation (RAG):

- **Determine the type of documents you'll be using:** The best types are documents from which we can get clean and unobscured text. PDFs can be problematic because they are designed for printing, not for extracting sensible text. To work with PDFs, we should get the source document or use tools to handle it.

- **Chunk the text:** We can't store the text as one long stream because of context size limitations and the potential for confusion. Chunking involves splitting the text into smaller pieces. Chunk text has many ways, such as character count, tokens, words, paragraphs, or sections. It is also possible to overlap chunks.
- **Create embeddings:** Embeddings are numerical representations of text that capture semantic meaning. We create embeddings by passing each chunk of text through a particular embedding model. The model outputs a vector, the length of which depends on the embedding model used. We should pull one (or more) embedding models from Ollama, to perform this task. Here are some examples of embedding models available at Ollama.

Model	Parameter Size	Embedding Size
mxbai-embed-large	334M	1024
nomic-embed-text	137M	768
all-minilm	23M	384

Generally, larger embedding sizes capture more nuanced information about the input. Still, they also require more computational resources to process, and a higher number of parameters should increase the latency (but also the quality of the response).

- **Store the chunks and embeddings in a vector database:** We will need a way to efficiently find the most relevant chunks of text for a given prompt, which is where a vector database comes in. We will use Chromadb, an AI-native open-source vector database, which simplifies building RAGs by creating knowledge, facts, and skills pluggable for LLMs. Both the embedding and the source text for each chunk are stored.
- **Build the prompt:** When we have a question, we create an embedding and query the vector database for the most similar chunks. Then, we select the top few results and include their text in the prompt.

The goal of RAG is to provide the model with the most relevant information from our documents, allowing it to generate more accurate and informative responses. So, let's implement a simple example of an SLM incorporating a particular set of facts about bees ("Bee Facts").

Inside the `ollama` env, enter the command in the terminal for Chromadb installation:

```
pip install ollama chromadb
```

Let's pull an intermediary embedding model, `nomic-embed-text`
`ollama pull nomic-embed-text`

And create a working directory:

```
cd Documents/OLLAMA/  
mkdir RAG-simple-bee  
cd RAG-simple-bee/
```

Let's create a new Jupyter notebook, 40-RAG-simple-bee for some exploration:

Import the needed libraries:

```
import ollama  
import chromadb  
import time
```

And define aor models:

```
EMB_MODEL = "nomic-embed-text"  
MODEL = "llama3.2:3B"
```

Initially, a knowledge base about bee facts should be created. This involves collecting relevant documents and converting them into vector embeddings. These embeddings are then stored in a vector database, allowing for efficient similarity searches later. Enter with the “document,” a base of “bee facts” as a list:

```
documents = [  
    "Bee-keeping, also known as apiculture, involves the \  
    maintenance of bee colonies, typically in hives, by humans.",  
    "The most commonly kept species of bees is the European \  
    honey bee (Apis mellifera).",  
    ...  
    "There are another 20,000 different bee species in \  
    the world.",  
    "Brazil alone has more than 300 different bee species, and \  
    the vast majority, unlike western honey bees, don't sting.",  
    "Reports written in 1577 by Hans Staden, mention three \  
    native bees used by indigenous people in Brazil.", \  
    "The indigenous people in Brazil used bees for medicine \  
    and food purposes",  
    "From Hans Staden report: probable species: mandaçaiá \  
    (Melipona quadrifasciata), mandaguari (Scaptotrigona \  
    ...
```

] postica) and jataí-amarela (*Tetragonisca angustula*)."

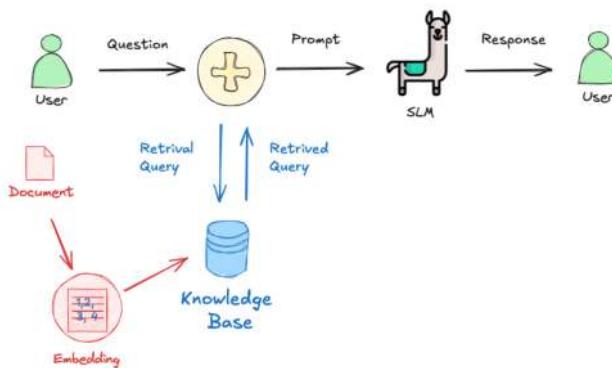
We do not need to "chunk" the document here because we will use each element of the list and a chunk.

Now, we will create our vector embedding database `bee_facts` and store the document in it:

```
client = chromadb.Client()
collection = client.create_collection(name="bee_facts")

# store each document in a vector embedding database
for i, d in enumerate(documents):
    response = ollama.embeddings(model=EMB_MODEL, prompt=d)
    embedding = response["embedding"]
    collection.add(
        ids=[str(i)], embeddings=[embedding], documents=[d]
    )
```

Now that we have our "Knowledge Base" created, we can start making queries, retrieving data from it:



User Query: The process begins when a user asks a question, such as "How many bees are in a colony? Who lays eggs, and how much? How about common pests and diseases?"

```
prompt = "How many bees are in a colony? Who lays eggs and \
how much? How about common pests and diseases?"
```

Query Embedding: The user's question is converted into a vector embedding using **the same embedding model** used for the knowledge base.

```
response = ollama.embeddings(prompt=prompt, model=EMB_MODEL)
```

Relevant Document Retrieval: The system searches the knowledge base using the query embedding to find the most relevant documents (in this case, the 5 most probable). This is done using a similarity search, which compares the query embedding to the document embeddings in the database.

```
results = collection.query(  
    query_embeddings=[response["embedding"]], n_results=5  
)  
data = results["documents"]
```

Prompt Augmentation: The retrieved relevant information is combined with the original user query to create an augmented prompt. This prompt now contains the user's question and pertinent facts from the knowledge base.

```
prompt = (  
    f"Using this data: {data}. " f"Respond to this prompt: {prompt}"  
)
```

Answer Generation: The augmented prompt is then fed into a language model, in this case, the 11lrama3.2:3b model. The model uses this enriched context to generate a comprehensive answer. Parameters like temperature, top_k, and top_p are set to control the randomness and quality of the generated response.

```
output = ollama.generate(  
    model=MODEL,  
    prompt = (  
        f"Using this data: {data}. "  
        f"Respond to this prompt: {prompt}"  
)  
  
    options={  
        "temperature": 0.0,  
        "top_k":10,  
        "top_p":0.5  
    })
```

Response Delivery: Finally, the system returns the generated answer to the user.

```
print(output["response"])
```

Based on the provided data, here are the answers to your \ questions:

1. How many bees are in a colony?

A typical bee colony can contain between 20,000 and 80,000 bees.

2. Who lays eggs and how much?

The queen bee lays up to 2,000 eggs per day during peak seasons.

3. What about common pests and diseases?

Common pests and diseases that affect bees include varroa \ mites, hive beetles, and foulbrood.

Let's create a function to help answer new questions:

```
def rag_bees(prompt, n_results=5, temp=0.0, top_k=10, top_p=0.5):
    start_time = time.perf_counter() # Start timing

    # generate an embedding for the prompt and retrieve the data
    response = ollama.embeddings(
        prompt=prompt,
        model=EMB_MODEL
    )

    results = collection.query(
        query_embeddings=[response["embedding"]],
        n_results=n_results
    )
    data = results['documents']

    # generate a response combining the prompt and data retrieved
    output = ollama.generate(
        model=MODEL,
        prompt = (
            f"Using this data: {data}. "
            f"Respond to this prompt: {prompt}"
        ),
        options={
            "temperature": temp,
            "top_k": top_k,
            "top_p": top_p
        }
    )
```

```
        print(output['response'])

    end_time = time.perf_counter() # End timing
    elapsed_time = round(
        (end_time - start_time), 1
    ) # Calculate elapsed time

    print(
        f"\n[INFO] ==> The code for model: {MODEL}, "
        f"took {elapsed_time}s to generate the answer.\n"
    )

    print(
        f"\n[INFO] ==> The code for model: {MODEL}, "
        f"took {elapsed_time}s to generate the answer.\n"
    )
```

We can now create queries and call the function:

```
prompt = "Are bees in Brazil?"
rag_beans(prompt)

Yes, bees are found in Brazil. According to the data, Brazil \
has more than 300 different bee species, and indigenous people \
in Brazil used bees for medicine and food purposes. \
Additionally, reports from 1577 mention three native bees \
used by indigenous people in Brazil.

[INFO] ==> The code for model: llama3.2:3b, took 22.7s to \
generate the answer.

By the way, if the model used supports multiple languages, we can use it
(for example, Portuguese), even if the dataset was created in English:

prompt = "Existem abelhas no Brazil?"
rag_beans(prompt)

Sim, existem abelhas no Brasil! De acordo com o relato de Hans \
Staden, há três espécies de abelhas nativas do Brasil que \
foram mencionadas: mandaçaiá (Melipona quadrifasciata), \
mandaguari (Scaptotrigona postica) e jataí-amarela \
(Tetragonisca angustula). Além disso, o Brasil é conhecido \
por ter mais de 300 espécies diferentes de abelhas, a \
maioria das quais não é agressiva e não põe veneno.
```

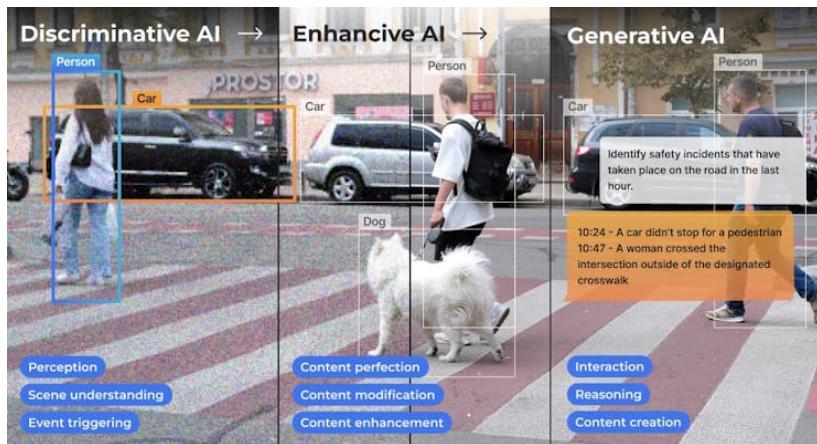
[INFO] ==> The code for model: llama3.2:3b, took 54.6s to \ generate the answer.

Going Further

The small LLM models tested worked well at the edge, both in text and with images, but of course, they had high latency regarding the last one. A combination of specific and dedicated models can lead to better results; for example, in real cases, an Object Detection model (such as YOLO) can get a general description and count of objects on an image that, once passed to an LLM, can help extract essential insights and actions.

According to Avi Baum, CTO at Hailo,

In the vast landscape of artificial intelligence (AI), one of the most intriguing journeys has been the evolution of AI on the edge. This journey has taken us from classic machine vision to the realms of discriminative AI, enhancive AI, and now, the groundbreaking frontier of generative AI. Each step has brought us closer to a future where intelligent systems seamlessly integrate with our daily lives, offering an immersive experience of not just perception but also creation at the palm of our hand.



Summary

This lab has demonstrated how a Raspberry Pi 5 can be transformed into a potent AI hub capable of running large language models (LLMs) for

real-time, on-site data analysis and insights using Ollama and Python. The Raspberry Pi's versatility and power, coupled with the capabilities of lightweight LLMs like Llama 3.2 and LLaVa-Phi-3-mini, make it an excellent platform for edge computing applications.

The potential of running LLMs on the edge extends far beyond simple data processing, as in this lab's examples. Here are some innovative suggestions for using this project:

1. Smart Home Automation:

- Integrate SLMs to interpret voice commands or analyze sensor data for intelligent home automation. This could include real-time monitoring and control of home devices, security systems, and energy management, all processed locally without relying on cloud services.

2. Field Data Collection and Analysis:

- Deploy SLMs on Raspberry Pi in remote or mobile setups for real-time data collection and analysis. This can be used in agriculture to monitor crop health, in environmental studies for wildlife tracking, or in disaster response for situational awareness and resource management.

3. Educational Tools:

- Create interactive educational tools that leverage SLMs to provide instant feedback, language translation, and tutoring. This can be particularly useful in developing regions with limited access to advanced technology and internet connectivity.

4. Healthcare Applications:

- Use SLMs for medical diagnostics and patient monitoring. They can provide real-time analysis of symptoms and suggest potential treatments. This can be integrated into telemedicine platforms or portable health devices.

5. Local Business Intelligence:

- Implement SLMs in retail or small business environments to analyze customer behavior, manage inventory, and optimize operations. The ability to process data locally ensures privacy and reduces dependency on external services.

6. Industrial IoT:

- Integrate SLMs into industrial IoT systems for predictive maintenance, quality control, and process optimization. The Raspberry Pi can serve as a localized data processing unit, reducing latency and improving the reliability of automated systems.

7. Autonomous Vehicles:

- Use SLMs to process sensory data from autonomous vehicles, enabling real-time decision-making and navigation. This can be applied to drones, robots, and self-driving cars for enhanced autonomy and safety.

8. Cultural Heritage and Tourism:

- Implement SLMs to provide interactive and informative cultural heritage sites and museum guides. Visitors can use these systems to get real-time information and insights, enhancing their experience without internet connectivity.

9. Artistic and Creative Projects:

- Use SLMs to analyze and generate creative content, such as music, art, and literature. This can foster innovative projects in the creative industries and allow for unique interactive experiences in exhibitions and performances.

10. Customized Assistive Technologies:

- Develop assistive technologies for individuals with disabilities, providing personalized and adaptive support through real-time text-to-speech, language translation, and other accessible tools.

Resources

- 10-Ollama_Python_Library notebook
- 20-Ollama_Function_Calling notebook
- 30-Function_Calling_with_images notebook
- 40-RAG-simple-bee notebook
- calc_distance_image python script

Vision-Language Models (VLM)

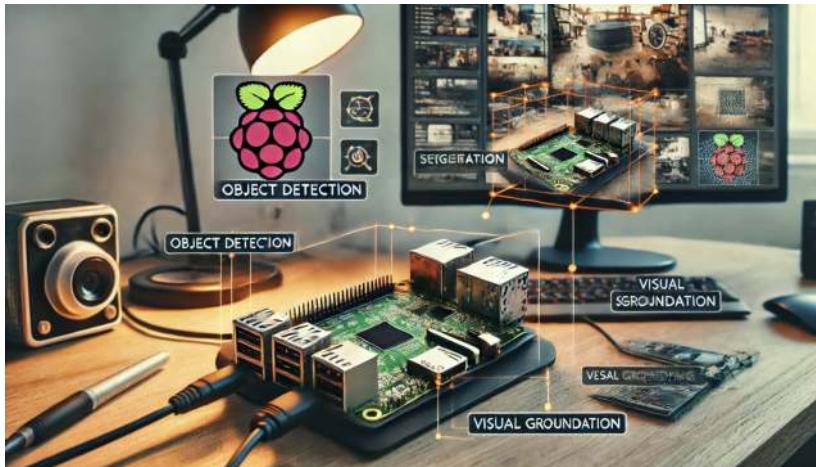


Figure 1.29: DALL-E prompt - A Raspberry Pi setup featuring vision tasks. The image shows a Raspberry Pi connected to a camera, with various computer vision tasks displayed visually around it, including object detection, image captioning, segmentation, and visual grounding. The Raspberry Pi is placed on a desk, with a display showing bounding boxes and annotations related to these tasks. The background should be a home workspace, with tools and devices typically used by developers and hobbyists.

Introduction

In this hands-on lab, we will continuously explore AI applications at the Edge, from the basic setup of the Florence-2, Microsoft's state-of-the-art vision foundation model, to advanced implementations on devices like the Raspberry Pi. We will learn to use Vision Language Models (VLMs) for tasks such as captioning, object detection, grounding, segmentation, and OCR on a Raspberry Pi.

Why Florence-2 at the Edge?

Florence-2 is a vision-language model open-sourced by Microsoft under the MIT license, which significantly advances vision-language models by combining a lightweight architecture with robust capabilities. Thanks to its training on the massive FLD-5B dataset, which contains 126 million images and 5.4 billion visual annotations, it achieves performance comparable to larger models. This makes Florence-2 ideal for deployment at the edge, where power and computational resources are limited.

In this tutorial, we will explore how to use Florence-2 for real-time computer vision applications, such as:

- Image captioning
- Object detection
- Segmentation
- Visual grounding

Visual grounding involves linking textual descriptions to specific regions within an image. This enables the model to understand where particular objects or entities described in a prompt are in the image. For example, if the prompt is “a red car,” the model will identify and highlight the region where the red car is found in the image. Visual grounding is helpful for applications where precise alignment between text and visual content is needed, such as human-computer interaction, image annotation, and interactive AI systems.

In the tutorial, we will walk through:

- Setting up Florence-2 on the Raspberry Pi
- Running inference tasks such as object detection and captioning
- Optimizing the model to get the best performance from the edge device
- Exploring practical, real-world applications with fine-tuning.

Florence-2 Model Architecture

Florence-2 utilizes a unified, prompt-based representation to handle various vision-language tasks. The model architecture consists of two main components: an **image encoder** and a **multi-modal transformer encoder-decoder**.

```

\noindent
\begin{minipage}{\linewidth}
\centering
\scalebox{0.8}{%
\begin{tikzpicture}[font=\small\usefont{T1}{phv}{m}{n}]
\tikzset{Line/.style={line width=1.0pt,black!50},
Box/.style={inner xsep=2pt,
draw=BlueLine,
node distance=0.6,
line width=0.75pt,
fill=BlueL,
anchor=west,
text width=36mm,
align=flush center,
minimum width=36mm,
minimum height=8mm
},
}

\node[Box,draw=RedLine, fill=RedL] (B1){Input};
\node[Box,below left=of B1] (B2){Image};
\node[Box,below right =of B1] (B3){Text Prompt};
\node[Box,below=of B2] (B4){Image Encoder (DaViT)};
\node[Box,below=of B3] (B5){Text Tokenize};
\node[Box,below=of $(B4)!0.5!(B5)$] (B6){Multimodality Encoder (Transformer)};
\node[Box,below=of B6] (B7){Multimodality Decoder (Transformer)};
\node[Box,below=of B7,draw=RedLine, fill=RedL] (B8){Output (Text/Coordinates)};
%
\draw[Line,-latex] (B1)-|(B2);
\draw[Line,-latex] (B1)-|(B3);
\draw[Line,-latex] (B2)--(B4);
\draw[Line,-latex] (B3)--(B5);
\draw[Line,-latex] (B4)|-(B6);
\draw[Line,-latex] (B5)|-(B6);
\draw[Line,-latex] (B6)--(B7);
\draw[Line,-latex] (B7)--(B8);
\end{tikzpicture}}
\end{minipage}

```

- **Image Encoder:** The image encoder is based on the DaViT (Dual Attention Vision Transformers) architecture. It converts input images into a series of visual token embeddings. These embeddings serve as the foundational representations of the visual content,

capturing both spatial and contextual information about the image.

- **Multi-Modal Transformer Encoder-Decoder:** Florence-2's core is the multi-modal transformer encoder-decoder, which combines visual token embeddings from the image encoder with textual embeddings generated by a BERT-like model. This combination allows the model to simultaneously process visual and textual inputs, enabling a unified approach to tasks such as image captioning, object detection, and segmentation.

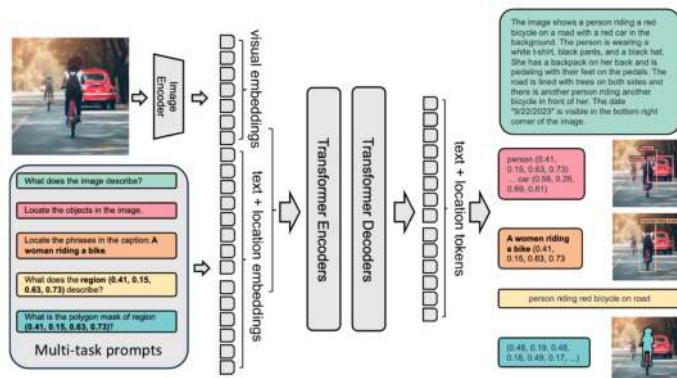
The model's training on the extensive FLD-5B dataset ensures it can effectively handle diverse vision tasks without requiring task-specific modifications. Florence-2 uses textual prompts to activate specific tasks, making it highly flexible and capable of zero-shot generalization. For tasks like object detection or visual grounding, the model incorporates additional location tokens to represent regions within the image, ensuring a precise understanding of spatial relationships.

Florence-2's compact architecture and innovative training approach allow it to perform computer vision tasks accurately, even on resource-constrained devices like the Raspberry Pi.

Technical Overview

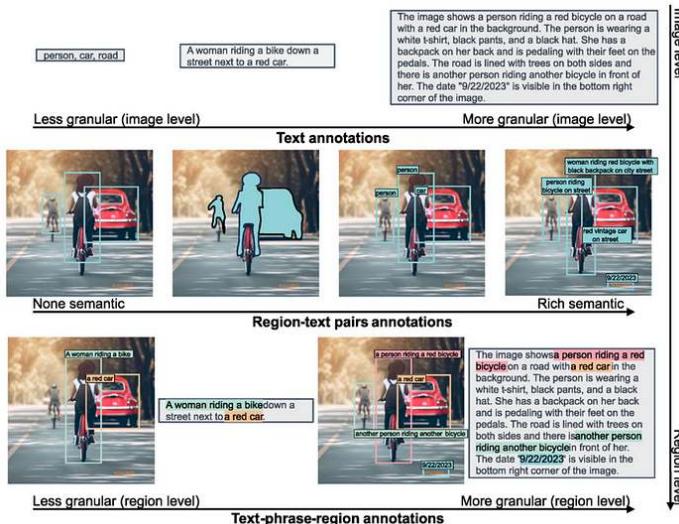
Florence-2 introduces several innovative features that set it apart:

Architecture



- **Lightweight Design:** Two variants available
 - Florence-2-Base: 232 million parameters
 - Florence-2-Large: 771 million parameters
- **Unified Representation:** Handles multiple vision tasks through a single architecture
- **DaViT Vision Encoder:** Converts images into visual token embeddings
- **Transformer-based Multi-modal Encoder-Decoder:** Processes combined visual and text embeddings

Training Dataset (FLD-5B)



- 126 million unique images
- 5.4 billion comprehensive annotations, including:
 - 500M text annotations
 - 1.3B region-text annotations
 - 3.6B text-phrase-region annotations
- Automated annotation pipeline using specialist models
- Iterative refinement process for high-quality labels

Key Capabilities

Florence-2 excels in multiple vision tasks:

Zero-shot Performance

- Image Captioning: Achieves 135.6 CIDEr score on COCO
- Visual Grounding: 84.4% recall@1 on Flickr30k
- Object Detection: 37.5 mAP on COCO val2017
- Referring Expression: 67.0% accuracy on RefCOCO

Fine-tuned Performance

- Competitive with specialist models despite the smaller size
- Outperforms larger models in specific benchmarks
- Efficient adaptation to new tasks

Practical Applications

Florence-2 can be applied across various domains:

1. Content Understanding

- Automated image captioning for accessibility
- Visual content moderation
- Media asset management

2. E-commerce

- Product image analysis
- Visual search
- Automated product tagging

3. Healthcare

- Medical image analysis
- Diagnostic assistance
- Research data processing

4. Security & Surveillance

- Object detection and tracking
- Anomaly detection
- Scene understanding

Comparing Florence-2 with other VLMs

Florence-2 stands out from other visual language models due to its impressive zero-shot capabilities. Unlike models like Google PaliGemma, which rely on extensive fine-tuning to adapt to various tasks, Florence-2 works right out of the box, as we will see in this lab. It can also compete with larger models like GPT-4V and Flamingo, which often have many more parameters but only sometimes match Florence-2's performance. For example, Florence-2 achieves better zero-shot results than Kosmos-2 despite having over twice the parameters.

In benchmark tests, Florence-2 has shown remarkable performance in tasks like COCO captioning and referring expression comprehension. It outperformed models like PolyFormer and UNINEXT in object detection and segmentation tasks on the COCO dataset. It is a highly competitive choice for real-world applications where both performance and resource efficiency are crucial.

Setup and Installation

Our choice of edge device is the Raspberry Pi 5 (Raspi-5). Its robust platform is equipped with the Broadcom BCM2712, a 2.4 GHz quad-core 64-bit Arm Cortex-A76 CPU featuring Cryptographic Extension and enhanced caching capabilities. It boasts a VideoCore VII GPU, dual 4Kp60 HDMI® outputs with HDR, and a 4Kp60 HEVC decoder. Memory options include 4 GB and 8 GB of high-speed LPDDR4X SDRAM, with 8 GB being our choice to run Florence-2. It also features expandable storage via a microSD card slot and a PCIe 2.0 interface for fast peripherals such as M.2 SSDs (Solid State Drives).

For real applications, SSDs are a better option than SD cards.

We suggest installing an Active Cooler, a dedicated clip-on cooling solution for Raspberry Pi 5 (Raspi-5), for this lab. It combines an aluminum heat sink with a temperature-controlled blower fan to keep the Raspi-5 operating comfortably under heavy loads, such as running Florence-2.



Environment configuration

To run Microsoft Florense-2 on the Raspberry Pi 5, we'll need a few libraries:

1. Transformers:

- Florence-2 uses the `transformers` library from Hugging Face for model loading and inference. This library provides the architecture for working with pre-trained vision-language models, making it easy to perform tasks like image captioning, object detection, and more. Essentially, `transformers` helps in interacting with the model, processing input prompts, and obtaining outputs.

2. PyTorch:

- PyTorch is a deep learning framework that provides the infrastructure needed to run the Florence-2 model, which includes tensor operations, GPU acceleration (if a GPU is available), and model training/inference functionalities. The Florence-2 model is trained in PyTorch, and we need it to leverage its functions, layers, and computation capabilities to perform inferences on the Raspberry Pi.

3. Timm (PyTorch Image Models):

- Florence-2 uses `timm` to access efficient implementations of vision models and pre-trained weights. Specifically, the `timm` library is utilized for the **image encoder** part of Florence-2, particularly for managing the DaViT architecture. It provides model definitions and optimized code for common vision

tasks and allows the easy integration of different backbones that are lightweight and suitable for edge devices.

4. Einops:

- **Einops** is a library for flexible and powerful tensor operations. It makes it easy to reshape and manipulate tensor dimensions, which is especially important for the multi-modal processing done in Florence-2. Vision-language models like Florence-2 often need to rearrange image data, text embeddings, and visual embeddings to align correctly for the transformer blocks, and `einops` simplifies these complex operations, making the code more readable and concise.

In short, these libraries enable different essential components of Florence-2:

- **Transformers** and **PyTorch** are needed to load the model and run the inference.
- **Timm** is used to access and efficiently implement the vision encoder.
- **Einops** helps reshape data, facilitating the integration of visual and text features.

All these components work together to help Florence-2 run seamlessly on our Raspberry Pi, allowing it to perform complex vision-language tasks relatively quickly.

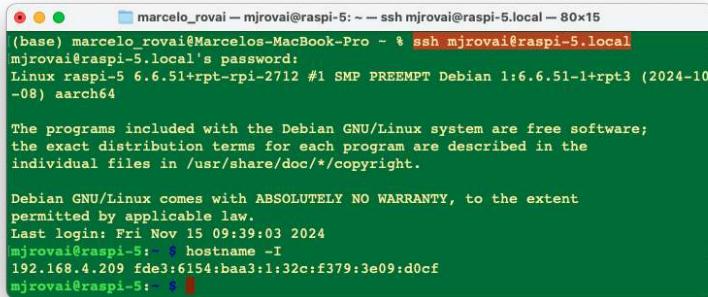
Considering that the Raspberry Pi already has its OS installed, let's use SSH to reach it from another computer:

```
ssh mjrovai@raspi-5.local
```

And check the IP allocated to it:

```
hostname -I
```

```
192.168.4.209
```



```
marcelo_rovai — mjrovai@raspi-5: ~ — ssh mjrovai@raspi-5.local — 80x15
(base) marcelo_rovai@Marcelos-MacBook-Pro ~ % ssh mjrovai@raspi-5.local
mjrovai@raspi-5: password:
Linux raspi-5 6.6.51+rpt-rpi-2712 #1 SMP PREEMPT Debian 1:6.6.51-1+rpt3 (2024-10-08) aarch64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/*copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Fri Nov 15 09:39:03 2024
mjrovai@raspi-5: ~ % hostname -I
192.168.4.209 fde3:6154:baa3:1:32c:f379:3e09:d0cf
mjrovai@raspi-5: ~ %
```

Updating the Raspberry Pi

First, ensure your Raspberry Pi is up to date:

```
sudo apt update
sudo apt upgrade -y
```

Initial setup for using PIP:

```
sudo apt install python3-pip
sudo rm /usr/lib/python3.11/EXTERNALLY-MANAGED
pip3 install --upgrade pip
```

Install Dependencies

```
sudo apt-get install libjpeg-dev libopenblas-dev libopenmpi-dev \
libomp-dev
```

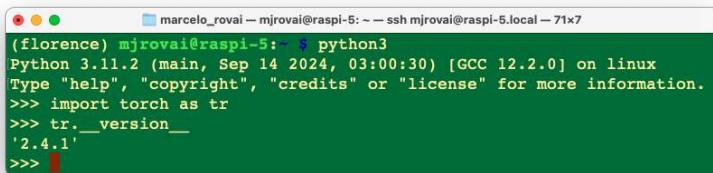
Let's set up and activate a **Virtual Environment** for working with Florence-2:

```
python3 -m venv ~/florence
source ~/florence/bin/activate
```

Install PyTorch

```
pip3 install setuptools numpy Cython
pip3 install requests
pip3 install torch torchvision \
--index-url https://download.pytorch.org/whl/cpu
pip3 install torchaudio \
--index-url https://download.pytorch.org/whl/cpu
```

Let's verify that PyTorch is correctly installed:



```
(florence) mjrovai@raspi-5: ~ ssh mjrovai@raspi-5.local ~ 71x7
marcelo_rovai@raspi-5: ~ ssh mjrovai@raspi-5.local ~ 71x7
(florence) mjrovai@raspi-5: ~ ssh mjrovai@raspi-5.local ~ 71x7
$ python3
Python 3.11.2 (main, Sep 14 2024, 03:00:30) [GCC 12.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch as tr
>>> tr.__version__
'2.4.1'
>>> 
```

Install Transformers, Timm and Einops:

```
pip3 install transformers
pip3 install timm einops
```

Install the model:

```
pip3 install autodistill-florence-2
```

Jupyter Notebook and Python libraries

Installing a Jupyter Notebook to run and test our Python scripts is possible.

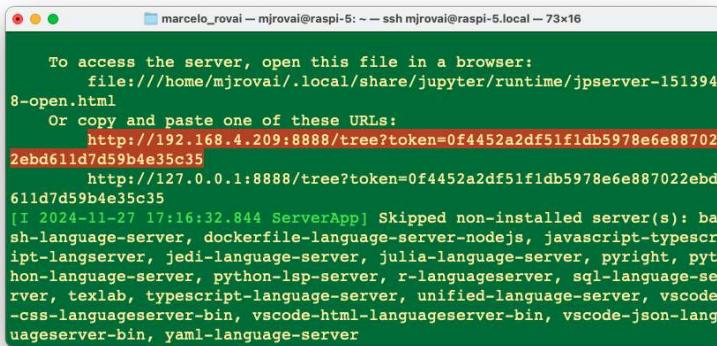
```
pip3 install jupyter
pip3 install numpy Pillow matplotlib
jupyter notebook --generate-config
```

Testing the installation

Running the Jupyter Notebook on the remote computer

```
jupyter notebook --ip=192.168.4.209 --no-browser
```

Running the above command on the SSH terminal, we can see the local URL address to open the notebook:



The terminal window shows the following text:

```
marcelo_rovai — mjrovai@raspi-5: ~ — ssh mjrovai@raspi-5.local — 73x16

To access the server, open this file in a browser:
  file:///home/mjrovai/.local/share/jupyter/runtime/jpserver-151394
8-open.html
Or copy and paste one of these URLs:
  http://192.168.4.209:8888/tree?token=0f4452a2df51fdb5978e6e88702
2ebd611d7d59b4e35c35
  http://127.0.0.1:8888/tree?token=0f4452a2df51fdb5978e6e887022ebd
611d7d59b4e35c35
[I 2024-11-27 17:16:32.844 ServerApp] Skipped non-installed server(s): ba
sh-language-server, dockerfile-language-server-nodejs, javascript-typescr
ipt-langsServer, jedi-language-server, julia-language-server, pyright, pyt
hon-language-server, python-lsp-server, r-languageserver, sql-language-se
rver, texlab, typescript-language-server, unified-language-server, vscode
-css-languageserver-bin, vscode-html-languageserver-bin, vscode-json-lang
uageserver-bin, yaml-language-server
```

The notebook with the code used on this initial test can be found on the Lab GitHub:

- 10-florence2_test.ipynb

We can access it on the remote computer by entering the Raspberry Pi's IP address and the provided token in a web browser (copy the entire URL from the terminal).

From the Home page, create a new notebook [Python 3 (ipykernel)] and copy and paste the example code from Hugging Face Hub.

The code is designed to run Florence-2 on a given image to perform **object detection**. It loads the model, processes an image and a prompt, and then generates a response to identify and describe the objects in the image.

- The **processor** helps prepare text and image inputs.
- The **model** takes the processed inputs to generate a meaningful response.
- The **post-processing** step refines the generated output into a more interpretable form, like bounding boxes for detected objects.

This workflow leverages the versatility of Florence-2 to handle **vision-language tasks** and is implemented efficiently using PyTorch, Transformers, and related image-processing tools.

```
import requests
from PIL import Image
import torch
```

```
from transformers import AutoProcessor, AutoModelForCausalLM

device = "cuda:0" if torch.cuda.is_available() else "cpu"
torch_dtype = (
    torch.float16 if torch.cuda.is_available() else torch.float32
)

model = AutoModelForCausalLM.from_pretrained(
    "microsoft/Florence-2-base",
    torch_dtype=torch_dtype,
    trust_remote_code=True,
).to(device)
processor = AutoProcessor.from_pretrained(
    "microsoft/Florence-2-base", trust_remote_code=True
)

prompt = "<OD>"

url = (
    "https://huggingface.co/datasets/huggingface/"
    "documentation-images/resolve/main/transformers/"
    "tasks/car.jpg?download=true"
)
image = Image.open(requests.get(url, stream=True).raw)

inputs = processor(text=prompt, images=image, return_tensors="pt").to(
    device, torch_dtype
)

generated_ids = model.generate(
    input_ids=inputs["input_ids"],
    pixel_values=inputs["pixel_values"],
    max_new_tokens=1024,
    do_sample=False,
    num_beams=3,
)
generated_text = processor.batch_decode(
    generated_ids, skip_special_tokens=False
)[0]

parsed_answer = processor.post_process_generation(
    generated_text,
    task="<OD>",
)
```

```
    image_size=(image.width, image.height),  
)
```

```
print(parsed_answer)
```

Let's break down the provided code step by step:

Importing Required Libraries

```
import requests  
from PIL import Image  
import torch  
from transformers import AutoProcessor, AutoModelForCausalLM
```

- **requests**: Used to make HTTP requests. In this case, it downloads an image from a URL.
- **PIL (Pillow)**: Provides tools for manipulating images. Here, it's used to open the downloaded image.
- **torch**: PyTorch is imported to handle tensor operations and determine the hardware availability (CPU or GPU).
- **transformers**: This module provides easy access to Florence-2 by using AutoProcessor and AutoModelForCausalLM to load pre-trained models and process inputs.

Determining the Device and Data Type

```
device = "cuda:0" if torch.cuda.is_available() else "cpu"  
  
torch_dtype = (  
    torch.float16 if torch.cuda.is_available() else torch.float32  
)
```

- **Device Setup**: The code checks if a CUDA-enabled GPU is available (`torch.cuda.is_available()`). The device is set to "cuda:0" if a GPU is available. Otherwise, it defaults to "cpu" (our case here).
- **Data Type Setup**: If a GPU is available, `torch.float16` is chosen, which uses half-precision floats to speed up processing and reduce memory usage. On the CPU, it defaults to `torch.float32` to maintain compatibility.

Loading the Model and Processor

```
model = AutoModelForCausalLM.from_pretrained(  
    "microsoft/Florence-2-base",  
    torch_dtype=torch_dtype,  
    trust_remote_code=True,  
) .to(device)  
  
processor = AutoProcessor.from_pretrained(  
    "microsoft/Florence-2-base", trust_remote_code=True  
)
```

- **Model Initialization:**

- `AutoModelForCausalLM.from_pretrained()` loads the pre-trained Florence-2 model from Microsoft's repository on Hugging Face. The `torch_dtype` is set according to the available hardware (GPU/CPU), and `trust_remote_code=True` allows the use of any custom code that might be provided with the model.
- `.to(device)` moves the model to the appropriate device (either CPU or GPU). In our case, it will be set to CPU.

- **Processor Initialization:**

- `AutoProcessor.from_pretrained()` loads the processor for Florence-2. The processor is responsible for transforming text and image inputs into a format the model can work with (e.g., encoding text, normalizing images, etc.).

Defining the Prompt

```
prompt = "<OD>"
```

- **Prompt Definition:** The string "`<OD>`" is used as a prompt. This refers to "Object Detection", instructing the model to detect objects on the image.

Downloading and Loading the Image

```
url = "https://huggingface.co/datasets/huggingface/"  
      "documentation-images/resolve/main/transformers/"  
      "tasks/car.jpg?download=true"  
image = Image.open(requests.get(url, stream=True).raw)
```

- **Downloading the Image:** The `requests.get()` function fetches the image from the specified URL. The `stream=True` parameter ensures the image is streamed rather than downloaded completely at once.
- **Opening the Image:** `Image.open()` opens the image so the model can process it.

Processing Inputs

```
inputs = processor(text=prompt, images=image, return_tensors="pt").to(  
    device, torch_dtype  
)
```

- **Processing Input Data:** The `processor()` function processes the text (`prompt`) and the image (`image`). The `return_tensors="pt"` argument converts the processed data into PyTorch tensors, which are necessary for inputting data into the model.
- **Moving Inputs to Device:** `.to(device, torch_dtype)` moves the inputs to the correct device (CPU or GPU) and assigns the appropriate data type.

Generating the Output

```
generated_ids = model.generate(  
    input_ids=inputs["input_ids"],  
    pixel_values=inputs["pixel_values"],  
    max_new_tokens=1024,  
    do_sample=False,  
    num_beams=3,  
)
```

- **Model Generation:** `model.generate()` is used to generate the output based on the input data.
 - `input_ids`: Represents the tokenized form of the prompt.
 - `pixel_values`: Contains the processed image data.
 - `max_new_tokens=1024`: Specifies the maximum number of new tokens to be generated in the response. This limits the response length.
 - `do_sample=False`: Disables sampling; instead, the generation uses deterministic methods (beam search).

- `num_beams=3`: Enables beam search with three beams, which improves output quality by considering multiple possibilities during generation.

Decoding the Generated Text

```
generated_text = processor.batch_decode(  
    generated_ids, skip_special_tokens=False  
) [0]
```

- **Batch Decode:** `processor.batch_decode()` decodes the generated IDs (tokens) into readable text. The `skip_special_tokens=False` parameter means that the output will include any special tokens that may be part of the response.

Post-processing the Generation

```
parsed_answer = processor.post_process_generation(  
    generated_text,  
    task="<OD>",  
    image_size=(image.width, image.height),  
)
```

- **Post-Processing:** `processor.post_process_generation()` is called to process the generated text further, interpreting it based on the task ("`<OD>`" for object detection) and the size of the image.
- This function extracts specific information from the generated text, such as bounding boxes for detected objects, making the output more useful for visual tasks.

Printing the Output

```
print(parsed_answer)
```

- Finally, `print(parsed_answer)` displays the output, which could include object detection results, such as bounding box coordinates and labels for the detected objects in the image.

Result

Running the code, we get as the Parsed Answer:

```
[{'<OD>' : {
    'bboxes' : [
        [34.23999786376953, 160.0800018310547, 597.4400024414062],
        [371.7599792480469, 272.32000732421875, 241.67999267578125],
        [303.67999267578125, 247.4399871826172, 454.0799865722656],
        [276.7200012207031, 553.9199829101562, 370.79998779296875],
        [96.31999969482422, 280.55999755859375, 198.0800018310547],
        [371.2799987792969]
    ],
    'labels' : ['car', 'door handle', 'wheel', 'wheel']
}}
```

First, let's inspect the image:

```
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 8))
plt.imshow(image)
plt.axis("off")
plt.show()
```



By the Object Detection result, we can see that:

```
'labels' : ['car', 'door handle', 'wheel', 'wheel']
```

It seems that at least a few objects were detected. We can also implement a code to draw the bounding boxes in the find objects:

```
def plot_bbox(image, data):
    # Create a figure and axes
    fig, ax = plt.subplots()

    # Display the image
    ax.imshow(image)

    # Plot each bounding box
    for bbox, label in zip(data["bboxes"], data["labels"]):
        # Unpack the bounding box coordinates
        x1, y1, x2, y2 = bbox
        # Create a Rectangle patch
        rect = patches.Rectangle(
            (x1, y1),
            x2 - x1,
            y2 - y1,
            linewidth=1,
            edgecolor="r",
            facecolor="none",
        )
        # Add the rectangle to the Axes
        ax.add_patch(rect)
        # Annotate the label
        plt.text(
            x1,
            y1,
            label,
            color="white",
            fontsize=8,
            bbox=dict(facecolor="red", alpha=0.5),
        )

    # Remove the axis ticks and labels
    ax.axis("off")

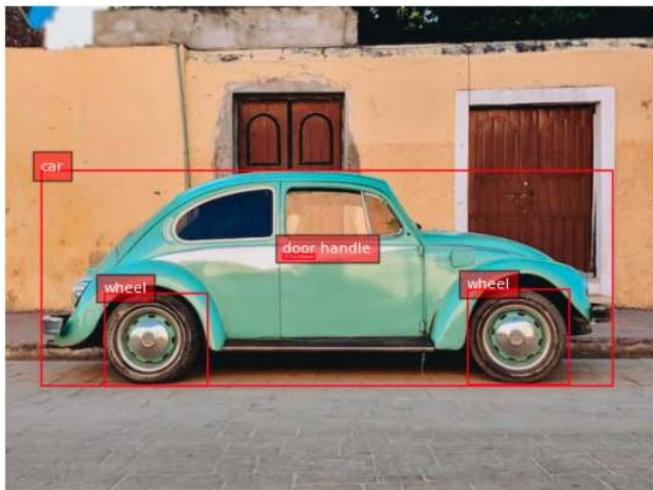
    # Show the plot
    plt.show()
```

Box (x0, y0, x1, y1): Location tokens correspond to the top-left and bottom-right corners of a box.

And running

```
plot_bbox(image, parsed_answer['<OD>'])
```

We get:



Florence-2 Tasks

Florence-2 is designed to perform a variety of computer vision and vision-language tasks through prompts. These tasks can be activated by providing a specific textual prompt to the model, as we saw with <OD> (Object Detection).

Florence-2's versatility comes from combining these prompts, allowing us to guide the model's behavior to perform specific vision tasks. Changing the prompt allows us to adapt Florence-2 to different tasks without needing task-specific modifications in the architecture. This capability directly results from Florence-2's unified model architecture and large-scale multi-task training on the FLD-5B dataset.

Here are some of the key tasks that Florence-2 can perform, along with example prompts:

Object Detection (OD)

- **Prompt:** "<OD>"
- **Description:** Identifies objects in an image and provides bounding boxes for each detected object. This task is helpful for applications like visual inspection, surveillance, and general object recognition.

Image Captioning

- **Prompt:** "<CAPTION>"
- **Description:** Generates a textual description for an input image. This task helps the model describe what is happening in the image, providing a human-readable caption for content understanding.

Detailed Captioning

- **Prompt:** "<DETAILED_CAPTION>"
- **Description:** Generates a more detailed caption with more nuanced information about the scene, such as the objects present and their relationships.

Visual Grounding

- **Prompt:** "<CAPTION_TO_PHRASE_GROUNDING>"
- **Description:** Links a textual description to specific regions in an image. For example, given a prompt like "a green car," the model highlights where the green car is in the image. This is useful for human-computer interaction, where you must find specific objects based on text.

Segmentation

- **Prompt:** "<REFERRING_EXPRESSION_SEGMENTATION>"
- **Description:** Performs segmentation based on a referring expression, such as "the blue cup." The model identifies and segments the specific region containing the object mentioned in the prompt (all related pixels).

Dense Region Captioning

- **Prompt:** "<DENSE_REGION_CAPTION>"
- **Description:** Provides captions for multiple regions within an image, offering a detailed breakdown of all visible areas, including different objects and their relationships.

OCR with Region

- **Prompt:** "<OCR_WITH_REGION>"
- **Description:** Performs Optical Character Recognition (OCR) on an image and provides bounding boxes for the detected text. This is useful for extracting and locating textual information in images, such as reading signs, labels, or other forms of text in images.

Phrase Grounding for Specific Expressions

- **Prompt:** "<CAPTION_TO_PHRASE_GROUNDING>" along with a specific expression, such as "a wine glass".
- **Description:** Locates the area in the image that corresponds to a specific textual phrase. This task allows for identifying particular objects or elements when prompted with a word or keyword.

Open Vocabulary Object Detection

- **Prompt:** "<OPEN_VOCABULARY_OD>"
- **Description:** The model can detect objects without being restricted to a predefined list of classes, making it helpful in recognizing a broader range of items based on general visual understanding.

Exploring computer vision and vision-language tasks

For exploration, all codes can be found on the GitHub:

- 20-florence_2.ipynb

Let's use a couple of images created by Dall-E and upload them to the Rasp-5 (FileZilla can be used for that). The images will be saved on a sub-folder named images :

```
dogs_cats = Image.open("./images/dogs-cats.jpg")
table = Image.open("./images/table.jpg")
```



Let's create a function to facilitate our exploration and to keep track of the latency of the model for different tasks:

```
def run_example(task_prompt, text_input=None, image=None):
    start_time = time.perf_counter() # Start timing
    if text_input is None:
        prompt = task_prompt
    else:
        prompt = task_prompt + text_input
    inputs = processor(
        text=prompt, images=image, return_tensors="pt"
    ).to(device)
    generated_ids = model.generate(
        input_ids=inputs["input_ids"],
        pixel_values=inputs["pixel_values"],
        max_new_tokens=1024,
        early_stopping=False,
        do_sample=False,
        num_beams=3,
    )
    generated_text = processor.batch_decode(
        generated_ids, skip_special_tokens=False
    )[0]
    parsed_answer = processor.post_process_generation(
```

```

        generated_text,
        task=task_prompt,
        image_size=(image.width, image.height),
    )

    end_time = time.perf_counter() # End timing
    elapsed_time = end_time - start_time # Calculate elapsed time
    print(
        f"\n[INFO] ==> Florence-2-base ({task_prompt}), \
            took {elapsed_time:.1f} seconds to execute.\n"
    )

    return parsed_answer

```

Caption

1. Dogs and Cats

```

run_example(task_prompt("<CAPTION>"), image=dogs_cats)
[INFO] ==> Florence-2-base (<CAPTION>), \
took 16.1 seconds to execute.

{'<CAPTION>': 'A group of dogs and cats sitting in a garden.'}

```

2. Table

```

run_example(task_prompt("<CAPTION>"), image=table)
[INFO] ==> Florence-2-base (<CAPTION>), \
took 16.5 seconds to execute.

{'<CAPTION>': 'A wooden table topped with a plate of fruit \
and a glass of wine.'}

```

Detailed Caption

1. Dogs and Cats

```

run_example(task_prompt("<DETAILED_CAPTION>"), image=dogs_cats)
[INFO] ==> Florence-2-base (<DETAILED_CAPTION>), \
took 25.5 seconds to execute.

{'<DETAILED_CAPTION>': 'The image shows a group of cats and \
dogs sitting on top of a lush green field, surrounded by plants \
'
}

```

```
with flowers, trees, and a house in the background. The sky is \
visible above them, creating a peaceful atmosphere.'}
```

2. Table

```
run_example(task_prompt="", image=table)
```

```
[INFO] ==> Florence-2-base (<DETAILED_CAPTION>), \
took 26.8 seconds to execute.
```

```
{'<DETAILED_CAPTION>': 'The image shows a wooden table with \
a bottle of wine and a glass of wine on it, surrounded by \
a variety of fruits such as apples, oranges, and grapes. \
In the background, there are chairs, plants, trees, and \
a house, all slightly blurred.'}
```

More Detailed Caption

1. Dogs and Cats

```
run_example(task_prompt="", image=dogs_cats)
```

```
[INFO] ==> Florence-2-base (<MORE_DETAILED_CAPTION>), \
took 49.8 seconds to execute.
```

```
{'<MORE_DETAILED_CAPTION>': 'The image shows a group of four \
cats and a dog in a garden. The garden is filled with colorful \
flowers and plants, and there is a pathway leading up to \
a house in the background. The main focus of the image is \
a large German Shepherd dog standing on the left side of \
the garden, with its tongue hanging out and its mouth open, \
as if it is panting. On the right side, there are \
two smaller cats, one orange and one gray, sitting on the \
grass. In the background, there is another golden retriever \
dog sitting and looking at the camera. The sky is blue and \
the sun is shining, creating a warm and inviting atmosphere.'}
```

2. Table

```
run_example(task_prompt="< MORE_DETAILED_CAPTION>", image=table)
```

```
[INFO] ==> Florence-2-base (<MORE_DETAILED_CAPTION>), \
took 32.4 seconds to execute.
```

```
{'<MORE_DETAILED_CAPTION>': 'The image shows a wooden table \
with a wooden tray on it. On the tray, there are various \
fruits such as grapes, oranges, apples, and grapes. There \

```

```
is also a bottle of red wine on the table. The background \
shows a garden with trees and a house. The overall mood \
of the image is peaceful and serene.'}
```

We can note that the more detailed the caption task, the longer the latency and the possibility of mistakes (like “The image shows a group of four cats and a dog in a garden”, instead of two dogs and three cats).

Object Detection

We can run the same previous function for object detection using the prompt <OD>.

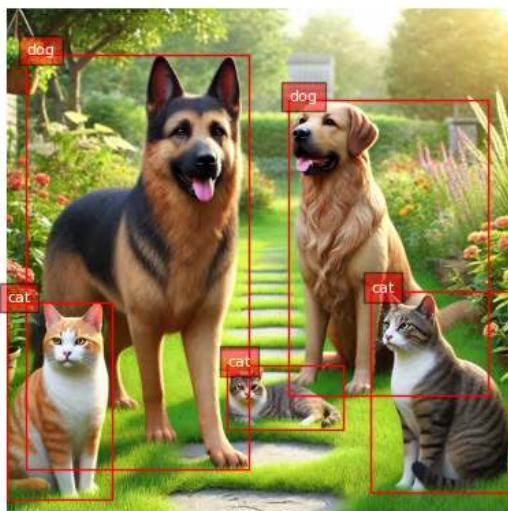
```
task_prompt = "<OD>"  
results = run_example(task_prompt, image=dogs_cats)  
print(results)
```

Let's see the result:

```
[INFO] ==> Florence-2-base (<OD>), took 20.9 seconds to execute.  
  
{'<OD>': {'bboxes': [  
    [737.79, 571.90, 1022.46, 980.48],  
    [0.51, 593.40, 211.45, 991.74],  
    [445.95, 721.40, 680.44, 850.43],  
    [39.42, 91.64, 491.00, 933.37],  
    [570.88, 184.83, 974.33, 782.84]  
],  
    'labels': ['cat', 'cat', 'cat', 'dog', 'dog']  
}}
```

Only by the labels ['cat', 'cat', 'cat', 'dog', 'dog'] is it possible to see that the main objects in the image were captured. Let's apply the function used before to draw the bounding boxes:

```
plot_bbox(dogs_cats, results["<OD>"])
```



Let's also do it with the Table image:

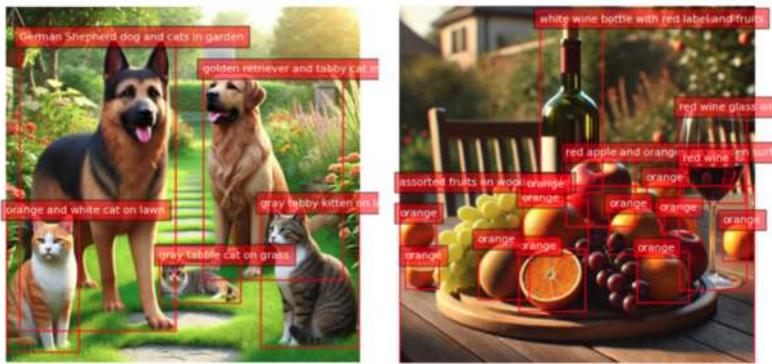
```
task_prompt = "<OD>"  
results = run_example(task_prompt, image=table)  
plot_bbox(table, results["<OD>"])  
  
[INFO] ==> Florence-2-base (<OD>), took 40.8 seconds to execute.
```



Dense Region Caption

It is possible to mix the classic Object Detection with the Caption task in specific sub-regions of the image:

```
task_prompt = "<DENSE_REGION_CAPTION>"  
  
results = run_example(task_prompt, image=dogs_cats)  
plot_bbox(dogs_cats, results["<DENSE_REGION_CAPTION>"])  
  
results = run_example(task_prompt, image=table)  
plot_bbox(table, results["<DENSE_REGION_CAPTION>"])
```



Caption to Phrase Grounding

With this task, we can enter with a caption, such as “a wine glass”, “a wine bottle,” or “a half orange,” and Florence-2 will localize the object in the image:

```
task_prompt = "<CAPTION_TO_PHRASE_GROUNDING>"  
  
results = run_example(  
    task_prompt, text_input="a wine bottle", image=table  
)  
plot_bbox(table, results["<CAPTION_TO_PHRASE_GROUNDING>"])  
  
results = run_example(  
    task_prompt, text_input="a wine glass", image=table  
)  
plot_bbox(table, results["<CAPTION_TO_PHRASE_GROUNDING>"])  
  
results = run_example(  
    task_prompt, text_input="a half orange", image=table  
)  
plot_bbox(table, results["<CAPTION_TO_PHRASE_GROUNDING>"])
```



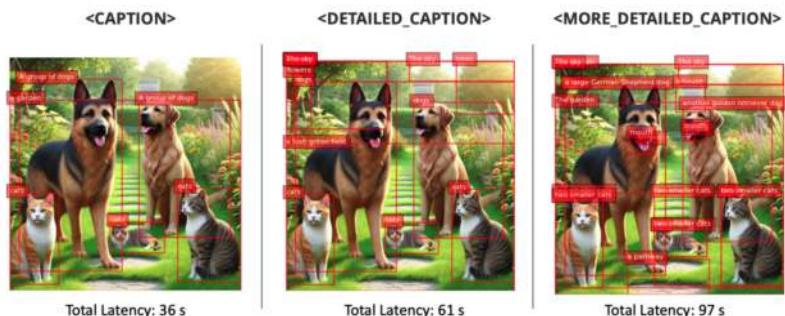
[INFO] ==> Florence-2-base (<CAPTION_TO_PHRASE_GROUNDING>), \
took 15.7 seconds to execute
each task.

Cascade Tasks

We can also enter the image caption as the input text to push Florence-2 to find more objects:

```
task_prompt = "<CAPTION>"  
results = run_example(task_prompt, image=dogs_cats)  
text_input = results[task_prompt]  
task_prompt = "<CAPTION_TO_PHRASE_GROUNDING>"  
results = run_example(task_prompt, text_input, image=dogs_cats)  
plot_bbox(dogs_cats, results["<CAPTION_TO_PHRASE_GROUNDING>"])
```

Changing the task_prompt among <CAPTION>, <DETAILED_CAPTION> and <MORE_DETAILED_CAPTION>, we will get more objects in the image.



Open Vocabulary Detection

<OPEN_VOCABULARY_DETECTION> allows Florence-2 to detect recognizable objects in an image without relying on a predefined list of categories, making it a versatile tool for identifying various items that may not have been explicitly labeled during training. Unlike <CAPTION_TO_PHRASE_GROUNDING>, which requires a specific text phrase to locate and highlight a particular object in an image, <OPEN_VOCABULARY_DETECTION> performs a broad scan to find and classify all objects present.

This makes <OPEN_VOCABULARY_DETECTION> particularly useful for applications where you need a comprehensive overview of everything in an image without prior knowledge of what to expect. Enter with a text describing specific objects not previously detected, resulting in their detection. For example:

```
task_prompt = "<OPEN_VOCABULARY_DETECTION>"  
  
text = [  
    "a house",  
    "a tree",  
    "a standing cat at the left",  
    "a sleeping cat on the ground",  
    "a standing cat at the right",  
    "a yellow cat",  
]  
  
for txt in text:  
    results = run_example(  
        task_prompt, text_input=txt, image=dogs_cats  
    )  
  
    bbox_results = convert_to_od_format(  
        results["<OPEN_VOCABULARY_DETECTION>"]  
    )  
  
    plot_bbox(dogs_cats, bbox_results)
```



[INFO] ==> Florence-2-base (<OPEN_VOCABULARY_DETECTION>), \
took 15.1 seconds to execute
each task.

Note: Trying to use Florence-2 to find objects that were not found can leads to mistakes (see examples on the Notebook).

Referring expression segmentation

We can also segment a specific object in the image and give its description (caption), such as “a wine bottle” on the table image or “a German Sheppard” on the dogs_cats.

Referring expression segmentation results format: {'<REFERRING_EXPRESSION_SEGMENTATION>': {'Polygons': [[[polygon]], ...], 'labels': ['', '', ...]}}, one object is represented by a list of polygons. each polygon is [x1, y1, x2, y2, ..., xn, yn].

Polygon (x1, y1, ..., xn, yn): Location tokens represent the vertices of a polygon in clockwise order.

So, let's first create a function to plot the segmentation:

```
from PIL import Image, ImageDraw, ImageFont
import copy
import random
import numpy as np

colormap = [
```

```
"blue",
"orange",
"green",
"purple",
"brown",
"pink",
"gray",
"olive",
"cyan",
"red",
"lime",
"indigo",
"violet",
"aqua",
"magenta",
"coral",
"gold",
"tan",
"skyblue",
]
]

def draw_polygons(image, prediction, fill_mask=False):
    """
    Draws segmentation masks with polygons on an image.

    Parameters:
        - image_path: Path to the image file.
        - prediction: Dictionary containing 'polygons' and 'labels'
                      keys. 'polygons' is a list of lists, each
                      containing vertices of a polygon. 'labels' is
                      a list of labels corresponding to each polygon.
        - fill_mask: Boolean indicating whether to fill the polygons
                     with color.
    """
    # Load the image

    draw = ImageDraw.Draw(image)

    # Set up scale factor if needed (use 1 if not scaling)
    scale = 1

    # Iterate over polygons and labels
```

```
for polygons, label in zip(
    prediction["polygons"], prediction["labels"]
):
    color = random.choice(colormap)
    fill_color = random.choice(colormap) if fill_mask else None

    for _polygon in polygons:
        _polygon = np.array(_polygon).reshape(-1, 2)
        if len(_polygon) < 3:
            print("Invalid polygon:", _polygon)
            continue

        _polygon = (_polygon * scale).reshape(-1).tolist()

        # Draw the polygon
        if fill_mask:
            draw.polygon(_polygon, outline=color, fill=fill_color)
        else:
            draw.polygon(_polygon, outline=color)

        # Draw the label text
        draw.text(
            (_polygon[0] + 8, _polygon[1] + 2), label, fill=color
        )

    # Save or display the image
    # image.show() # Display the image
    display(image)
```

Now we can run the functions:

```
task_prompt = "<REFERRING_EXPRESSION_SEGMENTATION>"

results = run_example(
    task_prompt, text_input="a wine bottle", image=table
)
output_image = copy.deepcopy(table)
draw_polygons(
    output_image,
    results["<REFERRING_EXPRESSION_SEGMENTATION>"],
    fill_mask=True,
)

results = run_example(
```

```

    task_prompt, text_input="a german sheppard", image=dogs_cats
)
output_image = copy.deepcopy(dogs_cats)
draw_polygons(
    output_image,
    results["<REFERRING_EXPRESSION_SEGMENTATION>"],
    fill_mask=True,
)

```



[INFO] ==> Florence-2-base
(<REFERRING_EXPRESSION_SEGMENTATION>), took 207.0 seconds
to execute each task.

Region to Segmentation

With this task, it is also possible to give the object coordinates in the image to segment it. The input format is '<loc_x1><loc_y1><loc_x2><loc_y2>' , [x1, y1, x2, y2] , which is the quantized coordinates in [0, 999].

For example, when running the code:

```

task_prompt = "<CAPTION_TO_PHRASE_GROUNDING>"
results = run_example(
    task_prompt, text_input="a half orange", image=table
)
results

```

The results were:

```
{'<CAPTION_TO_PHRASE_GROUNDING>': {'bboxes': [[343.552001953125,
689.6640625,
530.9440307617188,
```

```
873.9840698242188] ,  
'labels': ['a half']}
```

Using the bboxes rounded coordinates:

```
task_prompt = "<REGION_TO_SEGMENTATION>"  
results = run_example(  
    task_prompt,  
    text_input=("<loc_343><loc_690>" "<loc_531><loc_874>"),  
    image=table,  
)  
output_image = copy.deepcopy(table)  
draw_polygons(  
    output_image, results["<REGION_TO_SEGMENTATION>"], fill_mask=True  
)
```

We got the segmentation of the object on those coordinates (Latency: 83 seconds):



Region to Texts

We can also give the region (coordinates and ask for a caption):

```
task_prompt = "<REGION_TO_CATEGORY>"  
results = run_example(  
    task_prompt,
```

```
    text_input=("<loc_343><loc_690>" "<loc_531><loc_874>"),
    image=table,
)
results

[INFO] ==> Florence-2-base (<REGION_TO_CATEGORY>), \
took 14.3 seconds to execute.

{{{
'<REGION_TO_CATEGORY>':
'orange<loc_343><loc_690>'
'<loc_531><loc_874>'
}}}
```

The model identified an orange in that region. Let's ask for a description:

```
task_prompt = "<REGION_TO_DESCRIPTION>"
results = run_example(
    task_prompt,
    text_input=("<loc_343><loc_690>" "<loc_531><loc_874>"),
    image=table,
)
results

[INFO] ==> Florence-2-base (<REGION_TO_CATEGORY>), \
took 14.6 seconds to execute.

{{
'<REGION_TO_CATEGORY>':
'orange<loc_343><loc_690>'
'<loc_531><loc_874>'
}}
```

In this case, the description did not provide more details, but it could. Try another example.

OCR

With Florence-2, we can perform Optical Character Recognition (OCR) on an image, getting what is written on it (`task_prompt = '<OCR>'`) and also get the bounding boxes (location) for the detected text (`ask_prompt = '<OCR_WITH_REGION>'`). Those tasks can help extract and locate textual information in images, such as reading signs, labels, or other forms of text in images.

Let's upload a flyer from a talk in Brazil to Raspi. Let's test works in another language, here Portuguese):

```
flayer = Image.open("./images/embarcados.jpg")
# Display the image
plt.figure(figsize=(8, 8))
plt.imshow(flayer)
plt.axis("off")
# plt.title("Image")
plt.show()
```



Let's examine the image with '<MORE_DETAILED_CAPTION>' :

[INFO] ==> Florence-2-base (<MORE_DETAILED_CAPTION>), \
 took 85.2 seconds to execute.

```
{'<MORE_DETAILED_CAPTION>': 'The image is a promotional poster \
for an event called "Machine Learning Embarcados" hosted by \
Marcelo Roval. The poster has a black background with white \
text. On the left side of the poster, there is a logo of a \
coffee cup with the text "Café Com Embarcados" above it. \
Below the logo, it says "25 de Setembro as 17h" which \
translates to "25th of September as 17" in English. \
\n\nOn \
the right side, there are two smaller text boxes with the names \
of the participants and their names. The first text box reads \
"Democratizando a Inteligência Artificial para Países em \
Desenvolvimento" and the second text box says "Toda \
quarta-feira" which is Portuguese for "Transmissão via in \
Portuguese".\n\nIn the center of the image, there has a photo \
of Marcelo, a man with a beard and glasses, smiling at the \
'}
```

```
camera. He is wearing a white hard hat and a white shirt. \
The text boxes are in orange and yellow colors.'}
```

The description is very accurate. Let's get to the more important words with the task OCR:

```
task_prompt = "<OCR>"  
run_example(task_prompt, image=flayer)  
  
[INFO] ==> Florence-2-base <OCR>, took 37.7 seconds to execute.  
  
{'<OCR>':  
    'Machine Learning Café com Embarcado Embarcados '  
    'Democratizando a Inteligência Artificial para Paises em '  
    '25 de Setembro às 17h Desenvolvimento Toda quarta-feira '  
    'Marcelo Roval Professor na UNIFIEI e Transmissão via in '  
    'Co-Director do TinyML4D'}
```

Let's locate the words in the flyer:

```
task_prompt = "<OCR_WITH_REGION>"  
results = run_example(task_prompt, image=flayer)
```

Let's also create a function to draw bounding boxes around the detected words:

```
def draw_ocr_bboxes(image, prediction):  
    scale = 1  
    draw = ImageDraw.Draw(image)  
    bboxes = prediction["quad_boxes"]  
    labels = prediction["labels"]  
    for box, label in zip(bboxes, labels):  
        color = random.choice(colormap)  
        new_box = (np.array(box) * scale).tolist()  
        draw.polygon(new_box, width=3, outline=color)  
        draw.text(  
            (new_box[0] + 8, new_box[1] + 2),  
            "{}".format(label),  
            align="right",  
            fill=color,  
        )  
    display(image)  
  
output_image = copy.deepcopy(flayer)  
draw_ocr_bboxes(output_image, results["<OCR_WITH_REGION>"])
```



We can inspect the detected words:

```
results["<OCR_WITH_REGION>"]["labels"]

'</s>Machine Learning',
'Café',
'com',
'Embarcado',
'Embarcados',
'Democratizando a Inteligência',
'Artificial para Paises em',
'25 de Setembro ás 17h',
'Desenvolvimento',
'Toda quarta-feira',
'Marcelo Roval',
'Professor na UNIFIEI e',
'Transmissão via',
'in',
'Co-Director do TinyML4D']
```

Latency Summary

The latency observed for different tasks using Florence-2 on the Raspberry Pi (Raspi-5) varied depending on the complexity of the task:

- **Image Captioning:** It took approximately 16-17 seconds to generate a caption for an image.
- **Detailed Captioning:** Increased latency to around 25-27 seconds, requiring generating more nuanced scene descriptions.

- **More Detailed Captioning:** It took about 32-50 seconds, and the latency increased as the description grew more complex.
 - **Object Detection:** It took approximately 20-41 seconds, depending on the image's complexity and the number of detected objects.
 - **Visual Grounding:** Approximately 15-16 seconds to localize specific objects based on textual prompts.
 - **OCR (Optical Character Recognition):** Extracting text from an image took around 37-38 seconds.
 - **Segmentation and Region to Segmentation:** Segmentation tasks took considerably longer, with a latency of around 83-207 seconds, depending on the complexity and the number of regions to be segmented.

These latency times highlight the resource constraints of edge devices like the Raspberry Pi and emphasize the need to optimize the model and the environment to achieve real-time performance.

```
marcelo_rrovai ~ ssh mjroval@raspi-5: ~ Tasks: 82, 153 thr 127 kths; 4 ru
[|         |] 92.8% Tasks: 82, 153 thr 127 kths; 4 ru
[|         |] 93.4% Load average: 3.21 1.41 0.62
[|         |] 94.7% Uptime: 00:09:02
[|         |] 97.4% Mem: 5.1G/7.86G
[|         |] OK/2.00e] Swap:
```

Main	1/0									
PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%-MEM%	TIME+	Command
2279	mjroval	20	0	6595M	4769M	109M	R	369.7 59.2	6:39.30	/home/mj...
2809	mjroval	20	0	6595M	4769M	109M	R	94.1 59.2	1:37.71	/home/mj...
2807	mjroval	20	0	6595M	4769M	109M	R	90.8 59.2	1:37.06	/home/mj...
2808	mjroval	20	0	6595M	4769M	109M	R	90.8 59.2	1:37.91	/home/mj...
2233	mjroval	20	0	490M	153M	15360	S	6.6 1.9	0:04.44	/home/mj...

F1Help F2Setup F3Search F4Filter F5Tree F6SortByF7Nice -F8Nice +F9Kill F

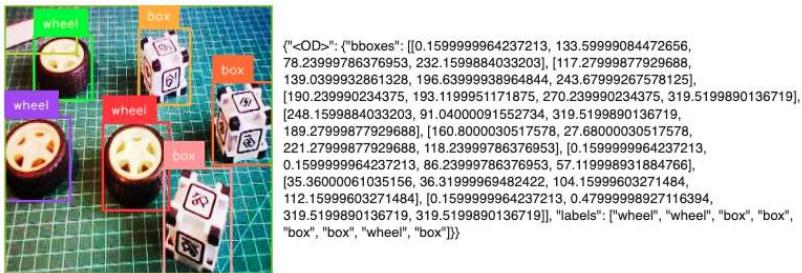
Running complex tasks can use all 8 GB of the Raspi-5's memory. For example, the above screenshot during the Florence OD task shows 4 CPUs at full speed and over 5 GB of memory in use. Consider increasing the SWAP memory to 2 GB.

Checking the CPU temperature with `vcgencmd measure_temp`, showed that temperature can go up to +80°C.

Fine-Tuning

As explored in this lab, Florence supports many tasks out of the box, including captioning, object detection, OCR, and more. However, like other pre-trained foundational models, Florence-2 may need domain-specific knowledge. For example, it may need to improve with medical or satellite imagery. In such cases, **fine-tuning** with a custom dataset is necessary. The Roboflow tutorial, How to Fine-tune Florence-2 for Object Detection Tasks, shows how to fine-tune Florence-2 on object detection datasets to improve model performance for our specific use case.

Based on the above tutorial, it is possible to fine-tune the Florence-2 model to detect boxes and wheels used in previous labs:



It is important to note that after fine-tuning, the model can still detect classes that don't belong to our custom dataset, like cats, dogs, grapes, etc, as seen before).

The complete fine-tuning project using a previously annotated dataset in Roboflow and executed on CoLab can be found in the notebook:

- 30-Finetune_florence_2_on_detection_dataset_box_vs_wheel.ipynb

In another example, in the post, Fine-tuning Florence-2 - Microsoft's Cutting-edge Vision Language Models, the authors show an example of fine-tuning Florence on DocVQA. The authors report that Florence 2 can perform visual question answering (VQA), but the released models don't include VQA capability.

Summary

Florence-2 offers a versatile and powerful approach to vision-language tasks at the edge, providing performance that rivals larger, task-specific

models, such as YOLO for object detection, BERT/RoBERTa for text analysis, and specialized OCR models.

Thanks to its multi-modal transformer architecture, Florence-2 is more flexible than YOLO in terms of the tasks it can handle. These include object detection, image captioning, and visual grounding.

Unlike **BERT**, which focuses purely on language, Florence-2 integrates vision and language, allowing it to excel in applications that require both modalities, such as image captioning and visual grounding.

Moreover, while traditional **OCR models** such as Tesseract and Easy-OCR are designed solely for recognizing and extracting text from images, Florence-2's OCR capabilities are part of a broader framework that includes contextual understanding and visual-text alignment. This makes it particularly useful for scenarios that require both reading text and interpreting its context within images.

Overall, Florence-2 stands out for its ability to seamlessly integrate various vision-language tasks into a unified model that is efficient enough to run on edge devices like the Raspberry Pi. This makes it a compelling choice for developers and researchers exploring AI applications at the edge.

Key Advantages of Florence-2

1. Unified Architecture

- Single model handles multiple vision tasks vs. specialized models (YOLO, BERT, Tesseract)
- Eliminates the need for multiple model deployments and integrations
- Consistent API and interface across tasks

2. Performance Comparison

- Object Detection: Comparable to YOLOv8 (~37.5 mAP on COCO vs. YOLOv8's ~39.7 mAP) despite being general-purpose
- Text Recognition: Handles multiple languages effectively like specialized OCR models (Tesseract, EasyOCR)
- Language Understanding: Integrates BERT-like capabilities for text processing while adding visual context

3. Resource Efficiency

- The Base model (232M parameters) achieves strong results despite smaller size

- Runs effectively on edge devices (Raspberry Pi)
- Single model deployment vs. multiple specialized models

Trade-offs

1. Performance vs. Specialized Models

- YOLO series may offer faster inference for pure object detection
- Specialized OCR models might handle complex document layouts better
- BERT/RoBERTa provide deeper language understanding for text-only tasks

2. Resource Requirements

- Higher latency on edge devices (15-200s depending on task)
- Requires careful memory management on Raspberry Pi
- It may need optimization for real-time applications

3. Deployment Considerations

- Initial setup is more complex than single-purpose models
- Requires understanding of multiple task types and prompts
- The learning curve for optimal prompt engineering

Best Use Cases

1. Resource-Constrained Environments

- Edge devices requiring multiple vision capabilities
- Systems with limited storage/deployment capacity
- Applications needing flexible vision processing

2. Multi-modal Applications

- Content moderation systems
- Accessibility tools
- Document analysis workflows

3. Rapid Prototyping

- Quick deployment of vision capabilities
- Testing multiple vision tasks without separate models
- Proof-of-concept development

Future Implications

Florence-2 represents a shift toward unified vision models that could eventually replace task-specific architectures in many applications. While specialized models maintain advantages in specific scenarios, the convenience and efficiency of unified models like Florence-2 make them increasingly attractive for real-world deployments.

The lab demonstrates Florence-2's viability on edge devices, suggesting future IoT, mobile computing, and embedded systems applications where deploying multiple specialized models would be impractical.

Resources

- 10-florence2_test.ipynb
- 20-florence_2.ipynb
- 30-Finetune_florence_2_on_detection_dataset_box_vs_wheel.ipynb

IX

KEY:SHARED

Part IX

X

SHARED RESOURCES

Part X

Overview

The labs in this section cover topics and techniques that are applicable across different hardware platforms. These labs are designed to be independent of specific boards, allowing you to focus on the fundamental concepts and algorithms used in (tiny) ML applications.

By exploring these shared labs, you'll gain a deeper understanding of the common challenges and solutions in embedded machine learning. The knowledge and skills acquired here will be valuable regardless of the specific hardware you work with in the future.

Exercise	Nicla Vision	XIAO ESP32S3
KWS Feature Engineering	Link	Link
DSP Spectral Features Block	Link	Link

KWS Feature Engineering



Figure 1.30: 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.

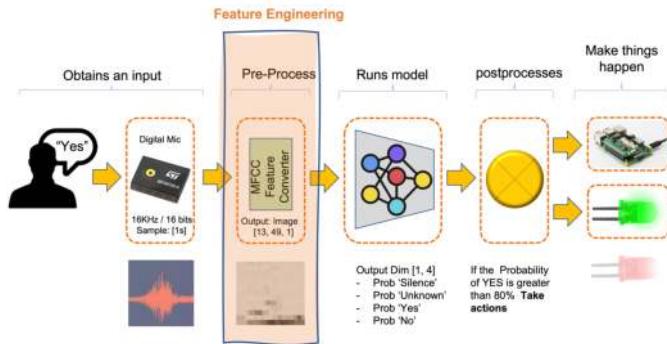
Machine learning models, especially traditional algorithms, don’t understand audio waves. They understand numbers arranged in some meaningful way, i.e., features. These features encapsulate the characteristics of the audio signal, making it easier for models to distinguish between different sounds.

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.

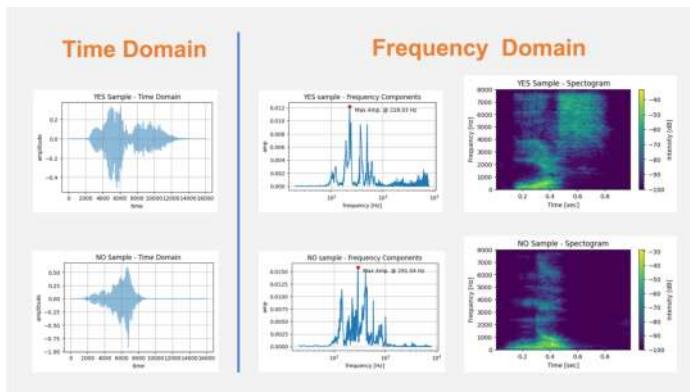
- **Frequency and Amplitude:** Frequency refers to the number of oscillations a waveform undergoes per unit time and is also measured in Hz. In the context of audio signals, different frequencies correspond to different pitches. Amplitude, on the other hand, measures the magnitude of the oscillations 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 Spectrograms 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.

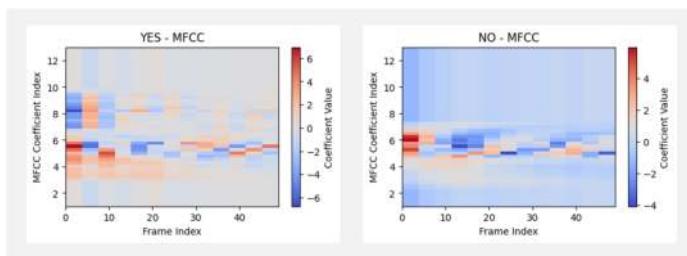
Overview to MFCCs

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:

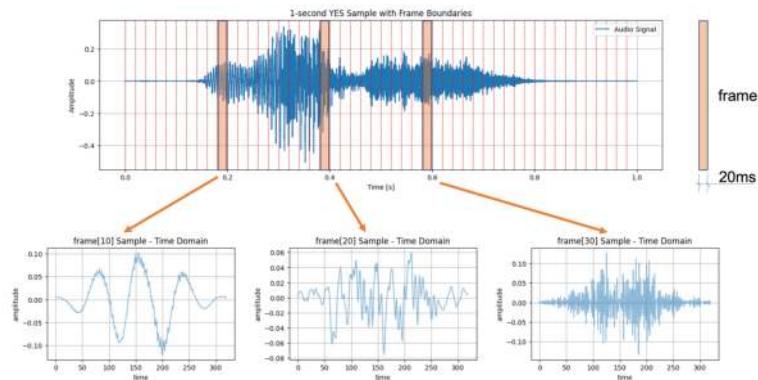
- **Dimensionality Reduction:** MFCCs capture essential spectral characteristics of the audio signal while significantly reducing the dimensionality of the data, making it ideal for resource-constrained TinyML applications.
- **Robustness:** MFCCs are less susceptible to noise and variations in pitch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

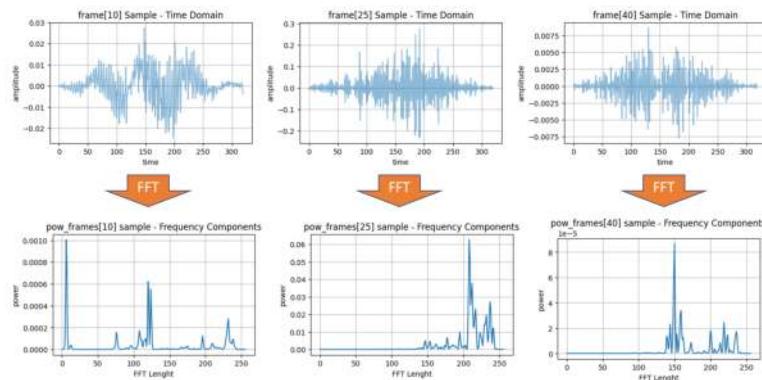
- **Pre-emphasis:** The first step is pre-emphasis, which is applied to accentuate the high-frequency components of the audio signal and balance the frequency spectrum. This is achieved by applying a filter that amplifies the difference between consecutive samples. The formula for pre-emphasis is: $y(t)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



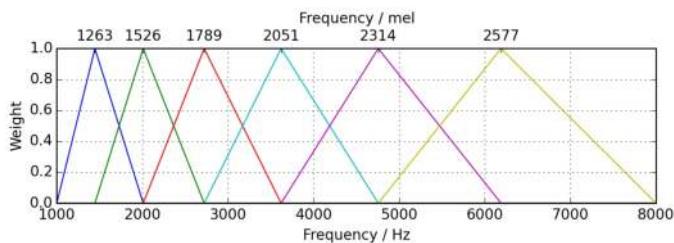
- **Fast Fourier Transform (FFT)** The Fast Fourier Transform (FFT) is applied to each windowed frame to convert it from the time domain to the frequency domain. The FFT gives us a complex-valued representation that includes both magnitude and phase

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.

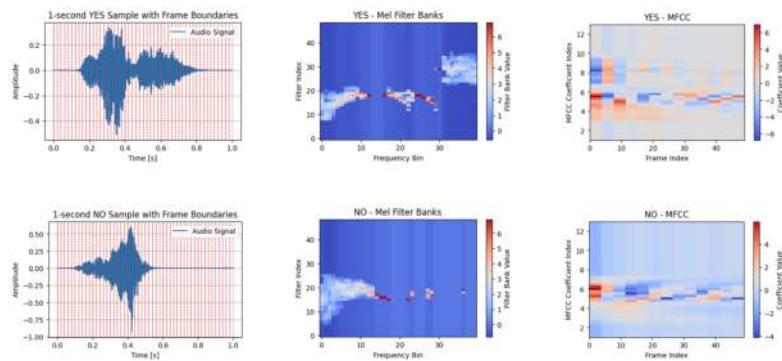
The power spectrum $P(f)$ of a signal $x(t)$ is defined as $P(f) = |X(f)|^2$, where $X(f)$ is the Fourier Transform of $x(t)$. By squaring the magnitude of the Fourier Transform, we emphasize *stronger* frequencies over *weaker* ones, thereby capturing more relevant spectral characteristics of the audio 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]

Summary

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](#)

DSP Spectral Features



Figure 1.31: 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 Arduino 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 **Gyrosopes**). 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.”

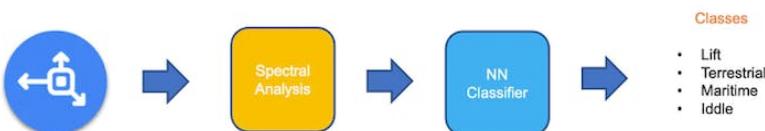
Feature extraction: Once the data is preprocessed and segmented, you can extract features that describe the motion’s characteristics. Some typical features extracted from accelerometer data include:

- **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

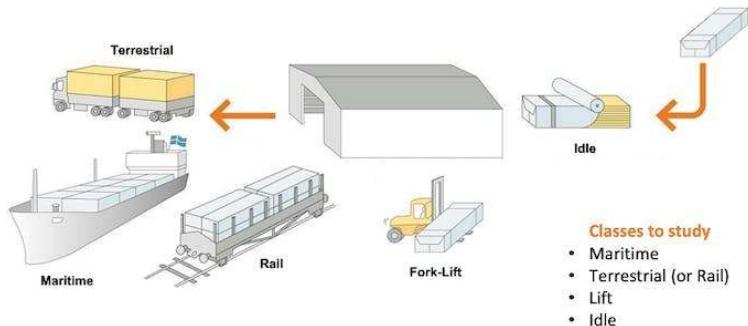


In the hands-on project, *Motion Classification and Anomaly Detection*, we simulated mechanical stresses in transport, where our problem was to classify four classes of movement:

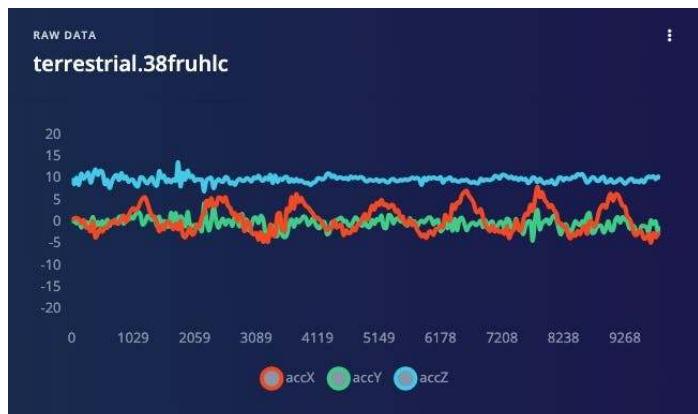
- **Maritime** (pallets in boats)
- **Terrestrial** (pallets in a Truck or Train)
- **Lift** (pallets being handled by Fork-Lift)
- **Idle** (pallets in Storage houses)

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

Case Study: Mechanical Stresses in Transport



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

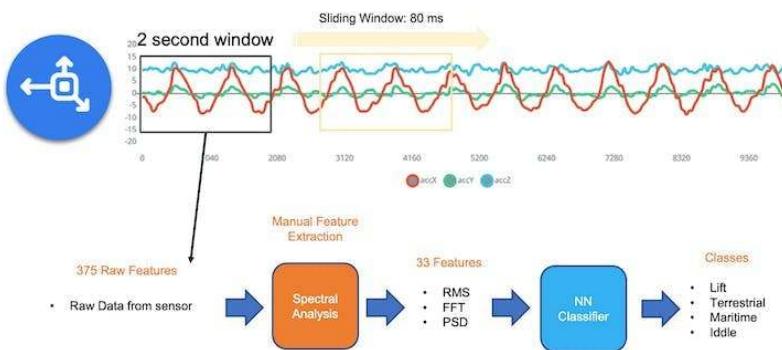


The result is similar when this analysis is done over another dataset with the same principle, using a different sampling frequency, 62.5 Hz instead of 50 Hz.

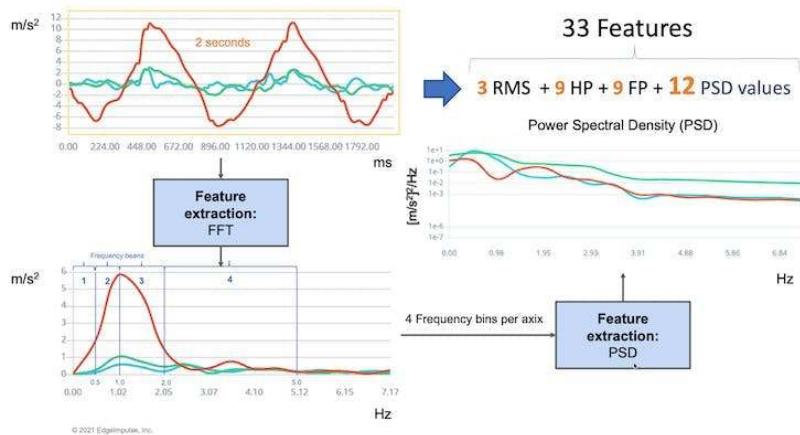
Data Pre-Processing

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 \times 2 seconds \times 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:

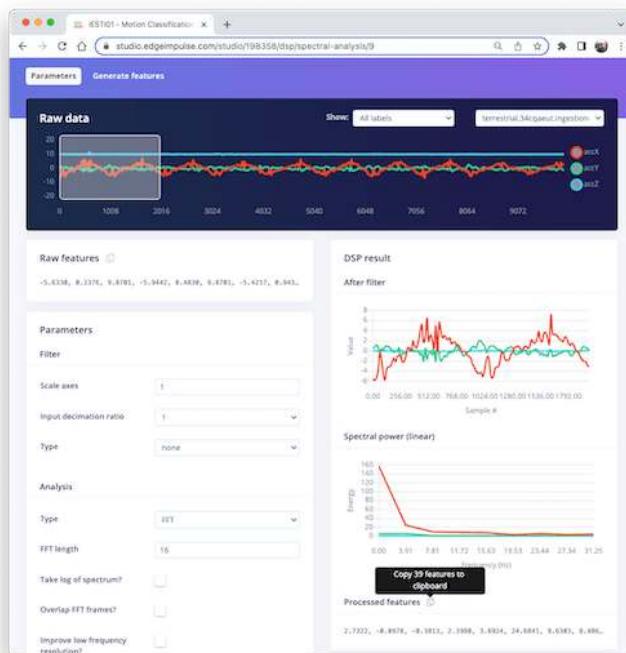
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import math
from scipy.stats import skew, kurtosis
from scipy import signal
from scipy.signal import welch
from scipy.stats import entropy
from sklearn import preprocessing
import pywt

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_Length = 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

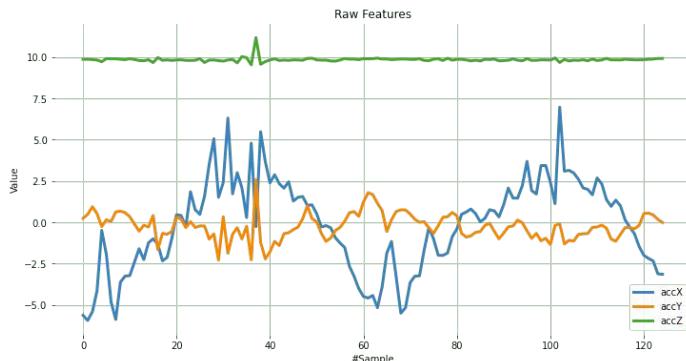
The data has samples from all axes; let's split and plot them separately:

```

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.

```

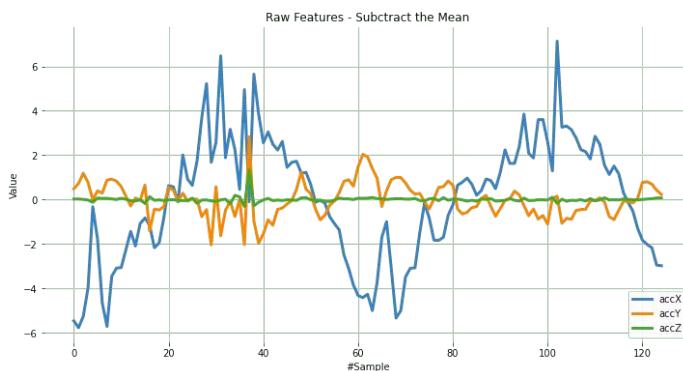
dtmean = [
    (sum(x) / len(x))
    for x in sensors
]

[
    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 - Subtract 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, \dots, x_n , the RMS is:

$$x_{\text{RMS}} = \sqrt{\frac{1}{n} (x_1^2 + x_2^2 + \dots + x_n^2)}$$

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**.

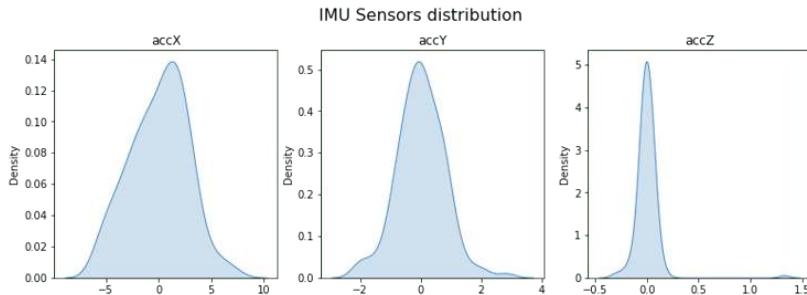
Here, we can see the sensor values distribution:

```
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')
```

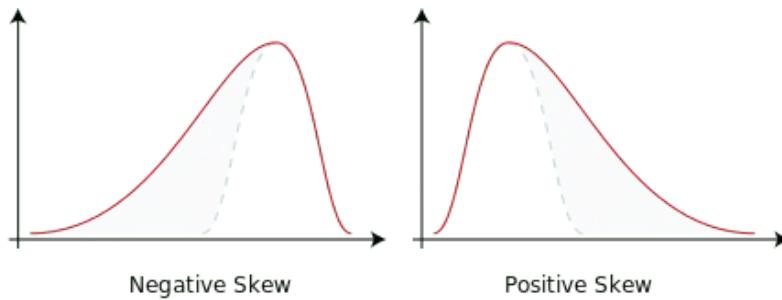
```

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

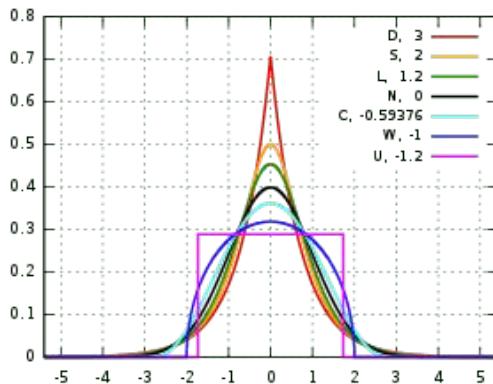
```

```
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 platykurtic, 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.

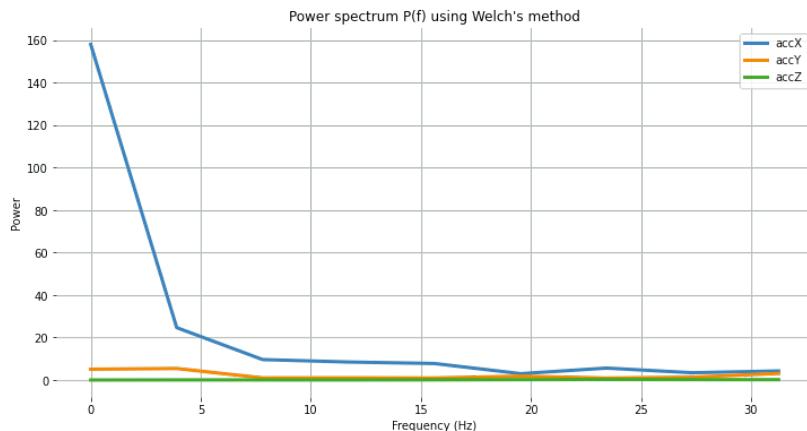
```
# Function used by Edge Impulse instead of scipy.signal.welch().
def welch_max_hold(fx, sampling_freq, nfft, n_overlap):
    n_overlap = int(n_overlap)
    spec_powers = [0 for _ in range(nfft//2+1)]
    ix = 0
    while ix <= len(fx):
        # Slicing truncates if end_idx > len,
        # and rfft 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_Length, 0)
fay,Pay = welch_max_hold(accY, fs, FFT_Length, 0)
faz,Paz = welch_max_hold(accZ, fs, FFT_Length, 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

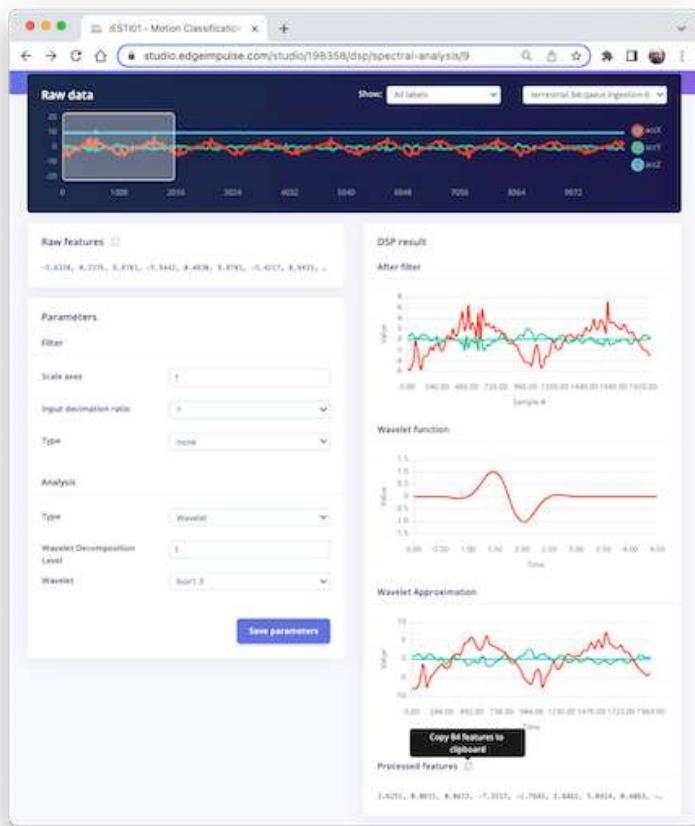
small waveform centered at a specific time and frequency. This process effectively decomposes the signal into different frequency bands, each of which can be analyzed separately.

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.

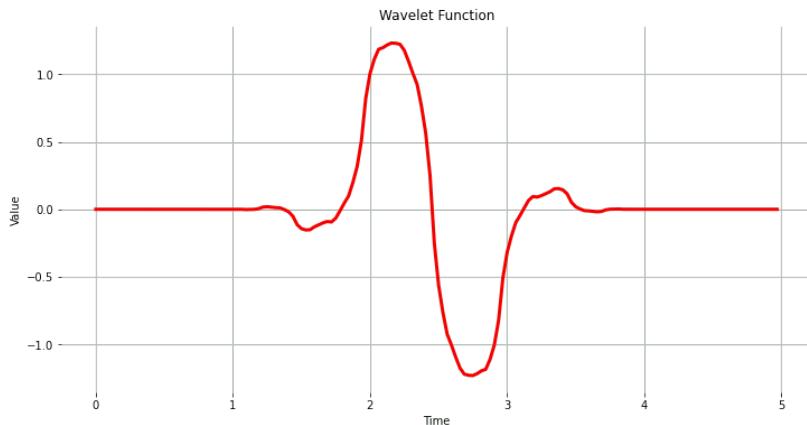
Let's select Wavelet on the Spectral Features block in the same project:

- 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:

- [1] Statistical 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

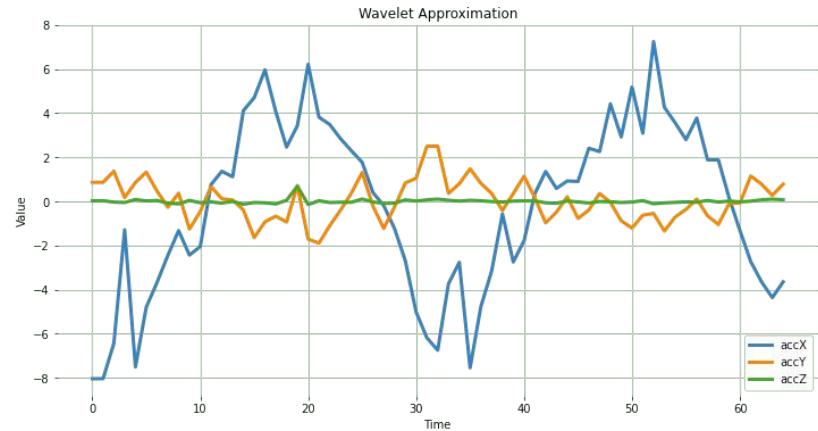
Wavelet analysis decomposes the signal (**accX**, **accY**, and **accZ**) into different frequency components using a set of filters, which separate these components into low-frequency (slowly varying parts of the signal containing long-term patterns), such as **accX_l1**, **accY_l1**, **accZ_l1** and, 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_10 = [calculate_statistics(x) for x in sensors]
stat_feat_11 = [calculate_statistics(x) for x in sensors_11]

The Skewness and Kurtosis:

skew_10 = [skew(x, bias=False) for x in sensors]
skew_11 = [skew(x, bias=False) for x in sensors_11]
```

```
kurtosis_10 = [kurtosis(x, bias=False) for x in sensors]
kurtosis_11 = [kurtosis(x, bias=False) for x in sensors_11]
```

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_10 = calculate_crossings(sensors)
cross_11 = calculate_crossings(sensors_11)
```

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

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_10 = [calculate_entropy(x) for x in sensors]
entropy_11 = [calculate_entropy(x) for x in sensors_11]
```

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_10 = []
for i in range(len(axis)):
    feat_10 = (
        stat_feat_10[i]
        + [skew_10[i]]
        + [kurtosis_10[i]]
        + [cross_10[0][i]]
        + [cross_10[1][i]]
        + [entropy_10[i]])
    )
    print(axis[i] + ' +x+= ', round(y, 4))
    for x, y in zip(L0_features_names, feat_10)][0]
    all_feat_10.append(feat_10)

all_feat_10 = [
    item
    for sublist in all_feat_10
    for item in sublist
]
print(f"\nAll L0 Features = {len(all_feat_10)}")
```

```
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

Summary

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.

XI

KEY:BACKMATTER

Part XI

