

EdgeAI Made Easy

Hands-On with the Raspberry Pi

Marcelo Rovai

2024-11-29

Table of contents

Preface	7
Acknowledgments	9
Introduction	10
About this Book	12
Setup	16
Introduction	17
Key Features	17
Raspberry Pi Models (covered in this book)	18
Engineering Applications	18
Hardware Overview	19
Raspberry Pi Zero 2W	19
Raspberry Pi 5	20
Installing the Operating System	20
The Operating System (OS)	20
Installation	21
Initial Configuration	24
Remote Access	24
SSH Access	24
To shut down the Raspi via terminal:	26
Transfer Files between the Raspi and a computer	26
Increasing SWAP Memory	30
Installing a Camera	32
Installing a USB WebCam	33
Installing a Camera Module on the CSI port	38
Running the Raspi Desktop remotely	42
Updating and Installing Software	48
Model-Specific Considerations	48
Raspberry Pi Zero (Raspi-Zero)	48
Raspberry Pi 4 or 5 (Raspi-4 or Raspi-5)	49

Image Classification	51
Introduction	52
Applications in Real-World Scenarios	52
Advantages of Running Classification on Edge Devices like Raspberry Pi	52
Setting Up the Environment	53
Updating the Raspberry Pi	53
Installing Required Libraries	53
Setting up a Virtual Environment (Optional but Recommended)	53
Installing TensorFlow Lite	54
Installing Additional Python Libraries	54
Creating a working directory:	54
Setting up Jupyter Notebook (Optional)	56
Verifying the Setup	57
Making inferences with Mobilenet V2	59
Define a general Image Classification function	66
Testing with a model trained from scratch	68
Installing Picamera2	69
Image Classification Project	72
The Goal	73
Data Collection	73
Training the model with Edge Impulse Studio	83
Dataset	83
The Impulse Design	85
Image Pre-Processing	87
Model Design	89
Model Training	89
Trading off: Accuracy versus speed	91
Model Testing	92
Deploying the model	92
Live Image Classification	99
Conclusion:	106
Resources	107
Object Detection	109
Introduction	110
Object Detection Fundamentals	111
Pre-Trained Object Detection Models Overview	113
Setting Up the TFLite Environment	114
Creating a Working Directory:	114
Inference and Post-Processing	115
EfficientDet	121
Object Detection Project	122
The Goal	122

Raw Data Collection	123
Labeling Data	126
Training an SSD MobileNet Model on Edge Impulse Studio	134
Uploading the annotated data	134
The Impulse Design	135
Preprocessing all dataset	137
Model Design, Training, and Test	139
Deploying the model	140
Inference and Post-Processing	141
Training a FOMO Model at Edge Impulse Studio	151
How FOMO works?	152
Impulse Design, new Training and Testing	154
Deploying the model	157
Inference and Post-Processing	159
Exploring a YOLO Model using Ultralytics	164
Talking about the YOLO Model	165
Installation	167
Testing the YOLO	168
Export Model to NCNN format	170
Exploring YOLO with Python	170
Training YOLOv8 on a Customized Dataset	174
Inference with the trained model, using the Raspi	178
Object Detection on a live stream	180
Conclusion	185
Resources	186
Counting objects with YOLO	187
Introduction	188
Installing and using Ultralytics YOLOv8	189
Testing the YOLO	190
Export Model to NCNN format	192
Exploring YOLO with Python	192
Estimating the number of Bees	197
Dataset	198
Pre-Processing	201
Training YOLOv8 on a Customized Dataset	204
Inference with the trained model, using the Rasp-Zero	208
Considerations about the Post-Processing	212
Script For Reading the SQLite Database	216
Adding Environment data	217
Conclusion	218
Resources	219

Small Language Models (SLM)	221
Introduction	222
Setup	222
Raspberry Pi Active Cooler	223
Generative AI (GenAI)	225
Large Language Models (LLMs)	226
Closed vs Open Models:	227
Small Language Models (SLMs)	228
Ollama	229
Installing Ollama	230
Meta Llama 3.2 1B/3B	232
Google Gemma 2 2B	236
Microsoft Phi3.5 3.8B	238
Multimodal Models	240
Inspecting local resources	243
Ollama Python Library	245
Function Calling	251
1. Importing Libraries	253
2. Defining Input and Model	254
3. Defining the Response Data Structure	254
4. Setting Up the OpenAI Client	254
5. Generating the Response	255
6. Calculating the Distance	255
Adding images	257
SLMs: Optimization Techniques	262
RAG Implementation	263
A simple RAG project	264
Going Further	270
Conclusion	271
Resources	273
Vision-Language Models at the Edge	274
Why Florence-2 at the Edge?	275
Florence-2 Model Architecture	275
Technical Overview	277
Architecture	277
Training Dataset (FLD-5B)	278
Key Capabilities	279
Practical Applications	279
Comparing Florence-2 with other VLMs	280
Setup and Installation	280
Environment configuration	281
Testing the installation	284

4. Defining the Prompt	288
7. Generating the Output	289
Florence-2 Tasks	293
Exploring computer vision and vision-language tasks	295
Caption	297
DETAILED_CAPTION	297
MORE_DETAILED_CAPTION	298
OD - Object Detection	299
DENSE_REGION_CAPTION	301
CAPTION_TO_PHRASE_GROUNDING	302
Cascade Tasks	303
OPEN_VOCABULARY_DETECTION	304
Referring expression segmentation	305
Region to Segmentation	308
Region to Texts	309
OCR	310
Latency Summary	314
Fine-Tuning	315
Conclusion	316
Key Advantages of Florence-2	316
Trade-offs	317
Best Use Cases	317
Future Implications	318
Resources	318
References	319
To learn more:	319
Online Courses	319
Books	319
Projects Repository	319
TinyML4D	320
About the author	321

Preface

In the rapidly evolving landscape of technology, the convergence of artificial intelligence and edge computing stands as one of the most exciting frontiers. This intersection promises to revolutionize how we interact with the world around us, bringing intelligence and decision-making capabilities directly to the devices we use every day. At the heart of this revolution lies the Raspberry Pi, a powerful yet accessible single-board computer that has democratized computing and now stands poised to do the same for edge AI.

The journey to this book began with a simple question: How can we make advanced machine learning accessible to everyone, not just those with access to powerful cloud resources or specialized hardware? The answer we found was the Raspberry Pi, which is the size of a credit card.

“EdgeML Made Easy: Hands-On with the Raspberry Pi” is a product of our passion for technology and belief in its power to solve real-world problems. It represents countless hours of experimentation, learning, and teaching distilled into a format that we hope will inspire and empower you to explore the fascinating world of edge AI.

This book is not just about theory or abstract concepts. It’s about getting your hands dirty, writing code, training models, and seeing your creations come to life. We’ve designed each chapter to blend foundational knowledge and practical application, always with an eye toward what’s possible on the Raspberry Pi platform.

From the compact Raspberry Pi Zero to the more powerful Pi 5, we explore how these incredible devices can become the brains of intelligent systems—recognizing images, understanding speech, detecting objects, and even running small language models. Each project in this book is a stepping stone, building your skills and confidence as you progress.

But beyond the technical skills, we hope this book instills something more valuable – a sense of curiosity and possibility. The field of edge AI is still in its infancy, with new applications and techniques emerging daily. By mastering the fundamentals presented here, you’ll be well-equipped to explore these frontiers, perhaps even pushing the boundaries of what’s possible on edge devices.

Whether you’re a student seeking to understand AI’s practical applications, a professional seeking to expand your skill set, or an enthusiast eager to add intelligence to your projects, we hope this book serves as both a guide and an inspiration.

As you embark on this journey, remember that every expert was once a beginner. The learning path is filled with challenges and moments of joy and discovery. Embrace both, and let your creativity guide you.

Thank you for joining us on this exciting adventure into edge machine learning. Let's begin exploring what's possible when we bring AI to the edge, one Raspberry Pi at a time.

Happy coding, and may your models always converge!

Prof. Marcelo Rovai November, 2024

Acknowledgments

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

Special commendation is reserved for Professor Vijay Janapa Reddi of Harvard University. His steadfast belief in the transformative potential of open-source communities, coupled with his invaluable guidance and teachings, has not just served as a beacon but as a cornerstone for our efforts from the beginning.

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

OpenAI's DALL-E generated illustrations of some of the images on the e-book and chapter covers via ChatGPT. Code and text reviews were done with the help of Claude 3.5 Sonnet and ChatGPT 4o.

Introduction

In the rapidly evolving landscape of technology, the convergence of artificial intelligence and edge computing is revolutionizing how we interact with and understand the world around us. At the forefront of this transformation is the Raspberry Pi, a powerful yet accessible single-board computer that has become a cornerstone for innovators, educators, and hobbyists alike.

“EdgeML Made Easy: Hands-On with the Raspberry Pi” is designed to bridge the gap between complex machine learning concepts and practical, real-world applications using the Raspberry Pi platform. This book is your guide to harnessing the power of edge AI, bringing sophisticated machine learning capabilities directly to where data is generated and actions are taken.

Why Raspberry Pi for Edge ML?

The Raspberry Pi, with its compact form factor, robust processing capabilities, and vibrant community support, offers an ideal platform for exploring and implementing edge machine learning solutions. Unlike more constrained microcontrollers, the Raspberry Pi provides:

1. Sufficient computational power to run complex ML models
2. A full-fledged operating system, enabling easier development and deployment
3. Extensive connectivity options, facilitating integration with various sensors and actuators
4. A rich ecosystem of libraries and tools optimized for machine learning tasks

This book will explore how to leverage these advantages to implement cutting-edge ML applications, from image classification and object detection to pose estimation and natural language processing.

What You’ll Learn

This hands-on guide will take you through the entire process of developing ML applications on the Raspberry Pi:

1. Setting up your Raspberry Pi for ML development
2. Collecting and preparing data for various ML tasks
3. Training and optimizing models for edge deployment
4. Implementing real-time inference on the Raspberry Pi
5. Building practical projects that combine multiple ML techniques

Whether you're a student, educator, maker, or professional looking to expand your skills, this book provides the knowledge and practical experience needed to bring your ML ideas to life on the Raspberry Pi platform.

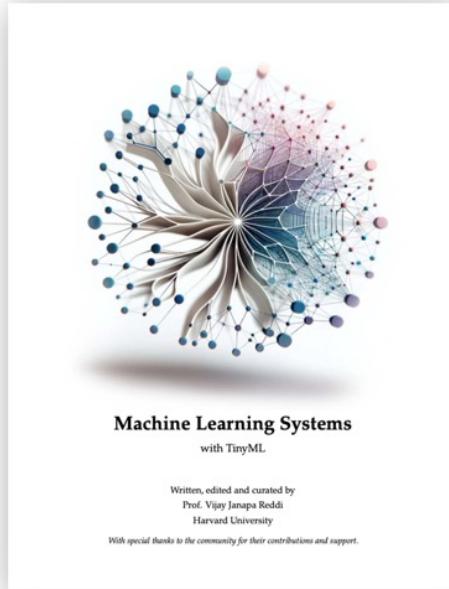
Empowering Innovation at the Edge

By the end of this journey, you'll be equipped with the skills to create intelligent, responsive systems that can see, understand, and interact with their environment. From smart cameras and voice assistants to industrial automation and IoT solutions, the possibilities are limited only by your imagination.

Join us as we explore the exciting intersection of machine learning and edge computing and discover how the Raspberry Pi can become your gateway to innovating at the edge. Let's embark on this journey to make edge ML accessible, practical, and impactful in solving real-world challenges.

About this Book

This book is part of the open book [Machine Learning Systems](#), which we invite you to



read.

“EdgeAI Made Easy: Hands-On with the Raspberry Pi” is an accessible, practical guide designed to empower readers with the knowledge and skills needed to implement artificial intelligence (DL and GenAI) at the edge using the Raspberry Pi platform. This book is part of the open-source [Machine Learning Systems](#) initiative, which aims to democratize AI education and application.

Key Features:

1. Practical Approach: Each chapter is built around hands-on projects demonstrating real-world applications of edge ML on Raspberry Pi.
2. Progressive Learning: The book starts with fundamental concepts and progresses to more advanced topics, ensuring a smooth learning curve for readers of various skill levels.

3. Raspberry Pi Focus: All examples and projects are optimized for various Raspberry Pi models, including the Pi Zero 2W, Pi 4, and Pi 5, highlighting each model's unique capabilities.
4. Comprehensive Coverage: From image processing and computer vision to natural language processing and sensor data analysis, this book covers various ML (and GenAI) applications relevant to edge computing.
5. Open-Source Tools: We emphasize using open-source frameworks and libraries, such as Edge Impulse Studio, TensorFlow Lite, OpenCV, PyTorch, and Ollama, ensuring accessibility and continuity in your learning journey.
6. Resource Optimization: Learn techniques to optimize ML models for the constrained resources of edge devices, balancing performance with efficiency.
7. Deployment Ready: Gain insights into best practices for deploying and maintaining ML models on Raspberry Pi in production environments.

Who This Book Is For:

- Students and educators in computer science, engineering, and related fields
- Hobbyists and makers interested in adding AI capabilities to their projects
- Professionals looking to implement edge AI solutions in various industries.
- Researchers exploring the intersection of IoT, edge computing, and machine learning

Prerequisites:

While this book is designed to be accessible to a broad audience, readers will benefit from:

- Basic familiarity with Python programming
- Fundamental understanding of machine learning concepts
- Experience with Raspberry Pi or similar single-board computers (helpful but not required)

Structure of the Book:

The book is divided into chapters, each focusing on a specific aspect of edge ML on Raspberry Pi. Every chapter includes:

- Theoretical background to understand the concepts
- Step-by-step tutorials for implementing ML models
- Practical projects that apply the learned techniques
- Tips for troubleshooting and optimizing performance
- Suggestions for further exploration and experimentation

What's Inside

- **Introduction to EdgeAI and TinyML:** A foundational look at embedded machine learning, including the differences between traditional AI, cloud AI, and AI at the edge.
- **Getting Started with the Raspberry Pi:** Learn to set up your Raspberry Pi for EdgeAI projects, including installation, system setup, and required libraries.
- **Hands-On Projects:** Step-by-step guides on implementing popular machine learning applications, such as image classification, object detection, anomaly detection, and more, directly on Raspberry Pi.
- **Large Language Models at the Edge:** Explores running large language models (LLMs) on edge devices like the Raspberry Pi. It covers setting up Ollama and Python to leverage these models for tasks such as text generation, summarization, and conversational AI, making powerful language AI accessible at the edge.
- **Vision-Language Models:** Focusing on deploying Florence-2, Microsoft's state-of-the-art Vision-Language Model, for various computer vision tasks such as captioning, object detection, segmentation, and visual grounding.

By the end of this book, you'll have a solid foundation in implementing various ML applications on Raspberry Pi and the confidence to tackle your edge AI projects. Whether you're looking to create smart home devices, develop intelligent monitoring systems, or explore the frontiers of edge computing, "EdgeAI Made Easy" will be your trusted companion on this exciting journey.

Setup

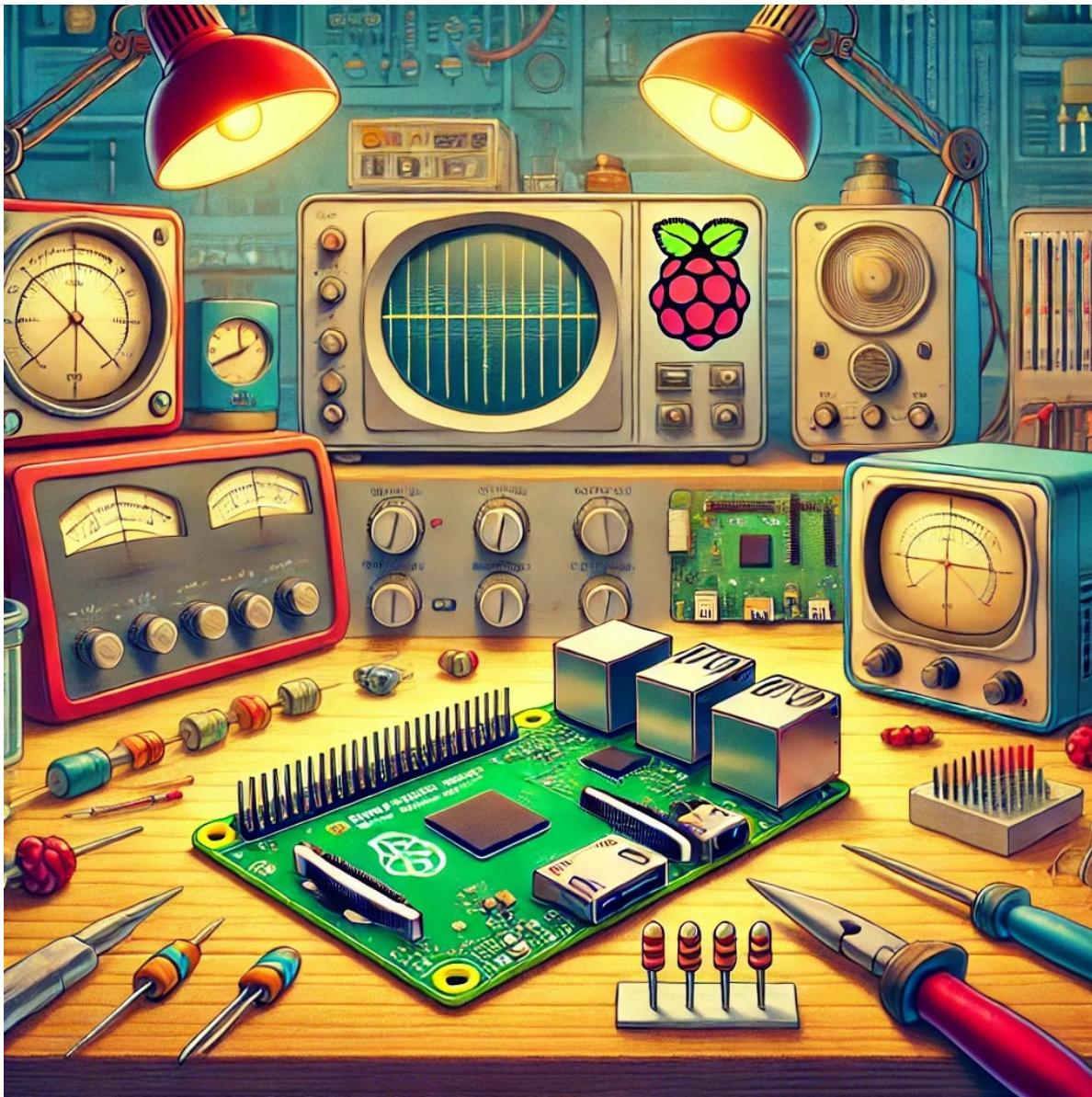


Figure 1: *DALL·E prompt - An electronics laboratory environment inspired by the 1950s, with a cartoon style. The lab should have vintage equipment, large oscilloscopes, old-fashioned tube radios, and large, boxy computers. The Raspberry Pi 5 board is prominently displayed, accurately shown in its real 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.

Introduction

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 8GB 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 provide 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: 1GHz single-core CPU (ARM Cortex-A53), 512MB 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: 2.4GHz quad-core CPU (ARM Cortex A-76), up to 8GB RAM, PCIe interface for expansions

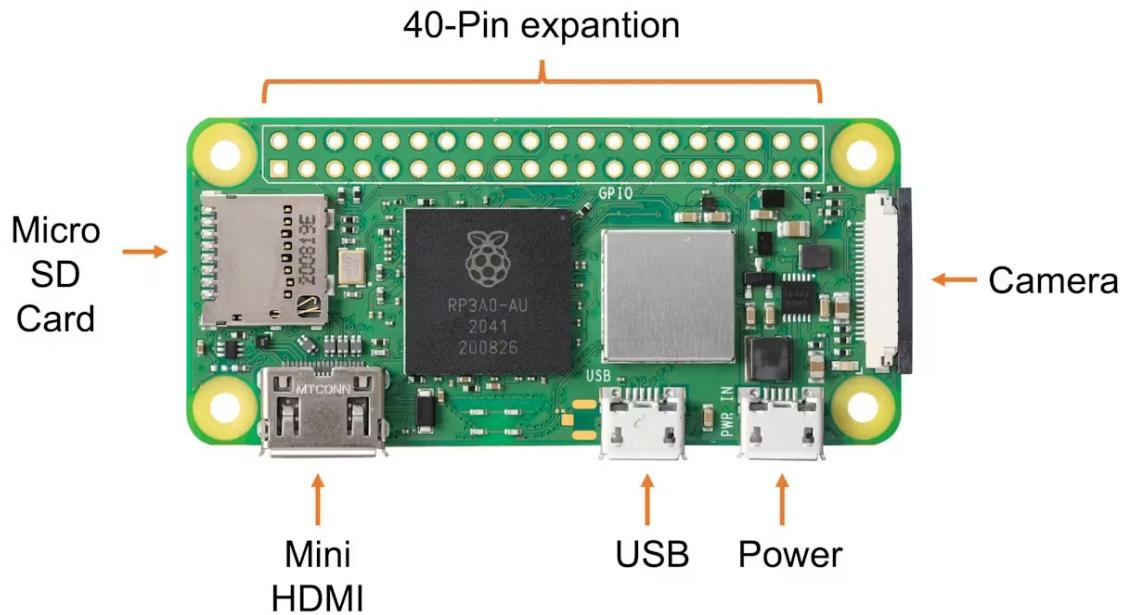
Engineering Applications

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.

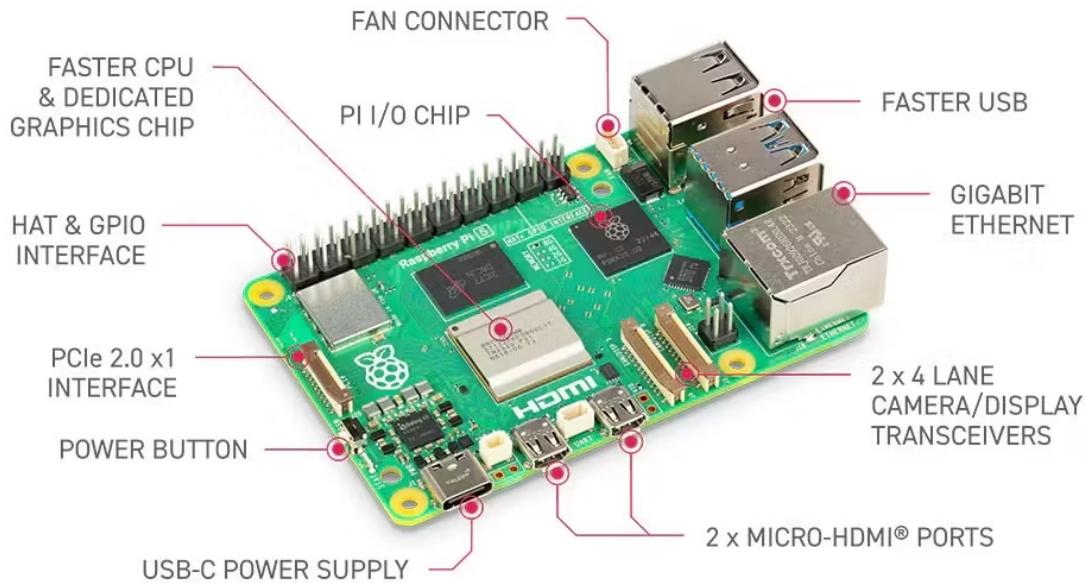
Hardware Overview

Raspberry Pi Zero 2W



- **Processor:** 1GHz quad-core 64-bit Arm Cortex-A53 CPU
- **RAM:** 512MB SDRAM
- **Wireless:** 2.4GHz 802.11 b/g/n wireless LAN, Bluetooth 4.2, BLE
- **Ports:** Mini HDMI, micro USB OTG, CSI-2 camera connector
- **Power:** 5V via micro USB port

Raspberry Pi 5



- **Processor:**
 - Pi 5: Quad-core 64-bit Arm Cortex-A76 CPU @ 2.4GHz
 - Pi 4: Quad-core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz
- **RAM:** 2GB, 4GB, or 8GB options (8GB 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:** 5V 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.

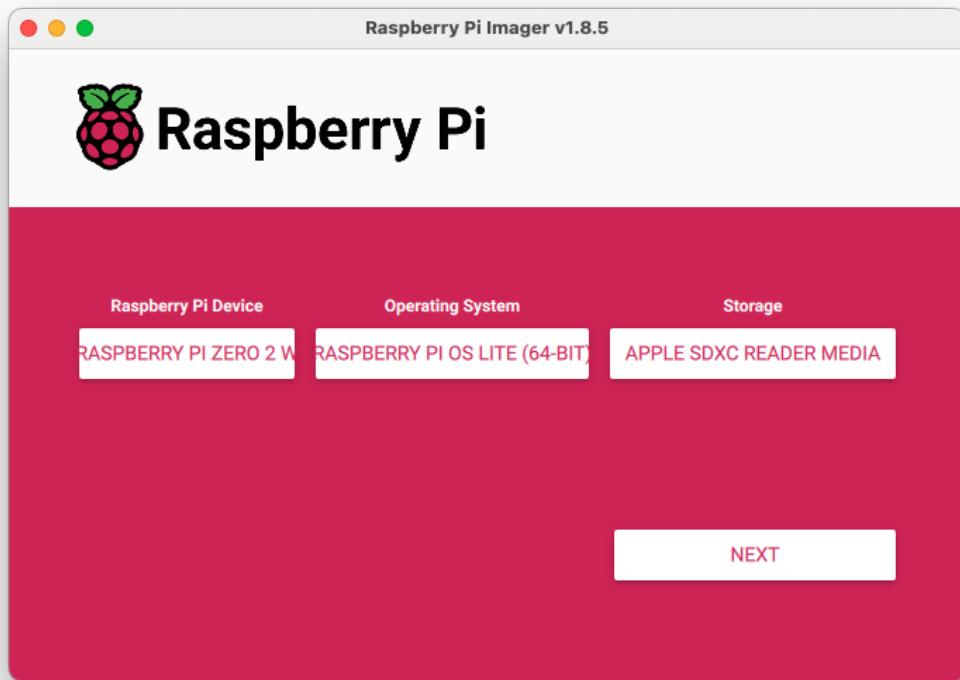
Installation

To use the Raspberry Pi, we will need an operating system. By default, Raspberry Pis 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.

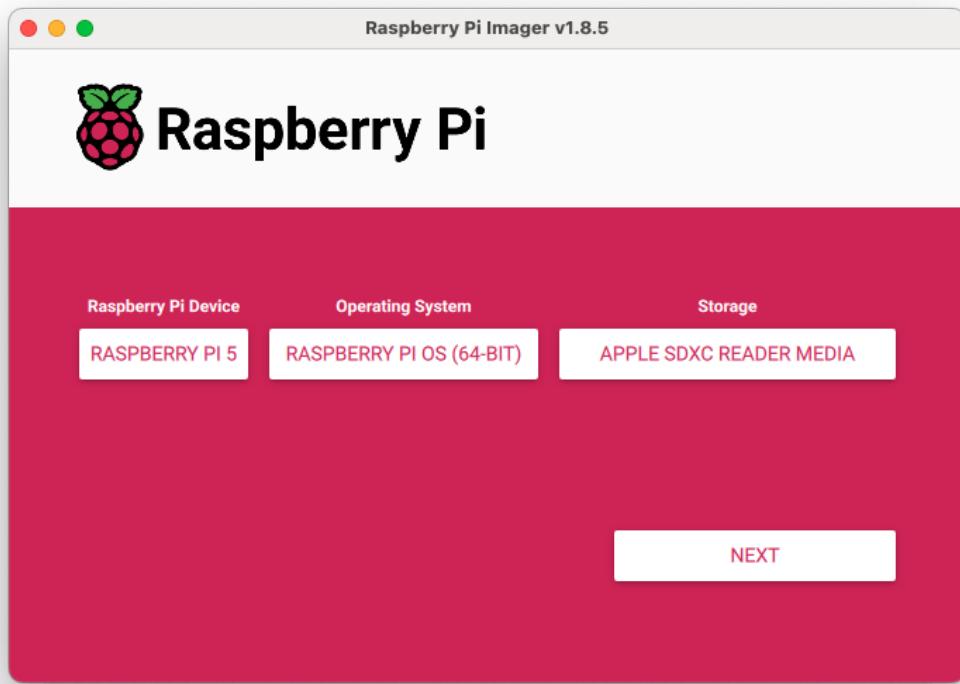
Follow the steps to install the OS in your Raspi.

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)**.

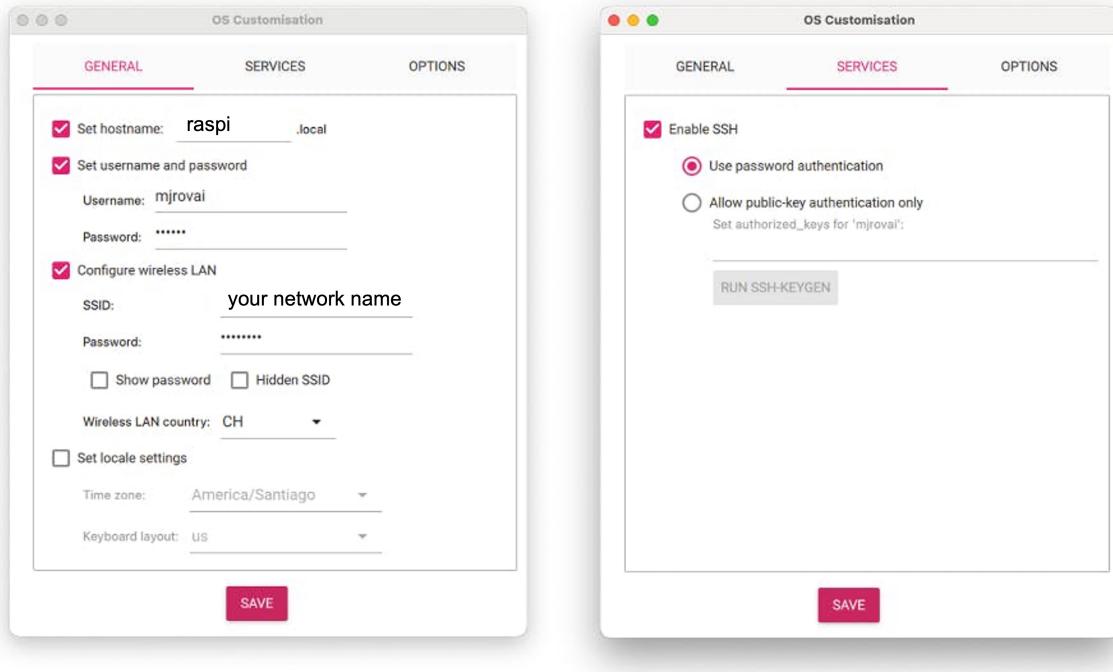


Due to its reduced SDRAM (512MB), the recommended OS for the Raspi-Zero is the 32-bit version. However, to run some machine learning models, such as the YOLOv8 from Ultralitics, we should use the 64-bit version. Although Raspi-Zero can run a *desktop*, we will choose the LITE version (no Desktop) to 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

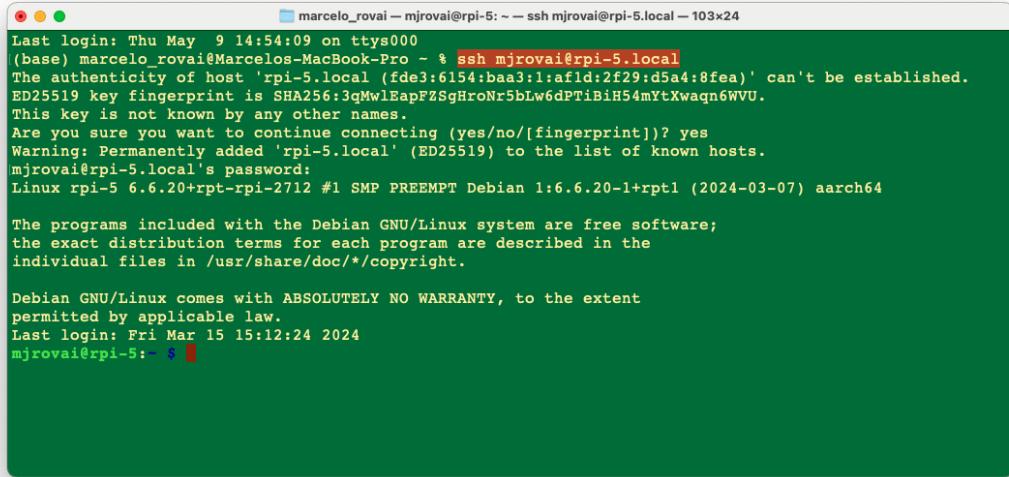
SSH Access

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.



The screenshot shows a terminal window titled "marcelo_rovai — mjrovai@rpi-5: ~ — ssh mjrovai@rpi-5.local — 103x24". The session starts with a warning about host key fingerprint authentication, followed by a password prompt. The user enters the password and is then presented with the standard Debian 6.6.20 system prompt. The terminal has a dark green background and white text.

```
Last login: Thu May  9 14:54:09 on ttys000
(base) marcelo_rovai@Marcelos-MacBook-Pro ~ % ssh mjrovai@rpi-5.local
The authenticity of host 'rpi-5.local (fde3:6154:baa3:1:afld:2f29:d5a4:8fea)' can't be established.
ED25519 key fingerprint is SHA256:3qMwlEapFZSgHroNr5blw6dPTiBiH54mYtXwagn6WVU.
This key is not known by any other names.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added 'rpi-5.local' (ED25519) to the list of known hosts.
mjrovai@rpi-5.local's password:
Linux rpi-5 6.6.20+rpt-rpi-2712 #1 SMP PREEMPT Debian 1:6.6.20-1+rpt1 (2024-03-07) 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 Mar 15 15:12:24 2024
mjrovai@rpi-5:~ $
```

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:af1d: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.

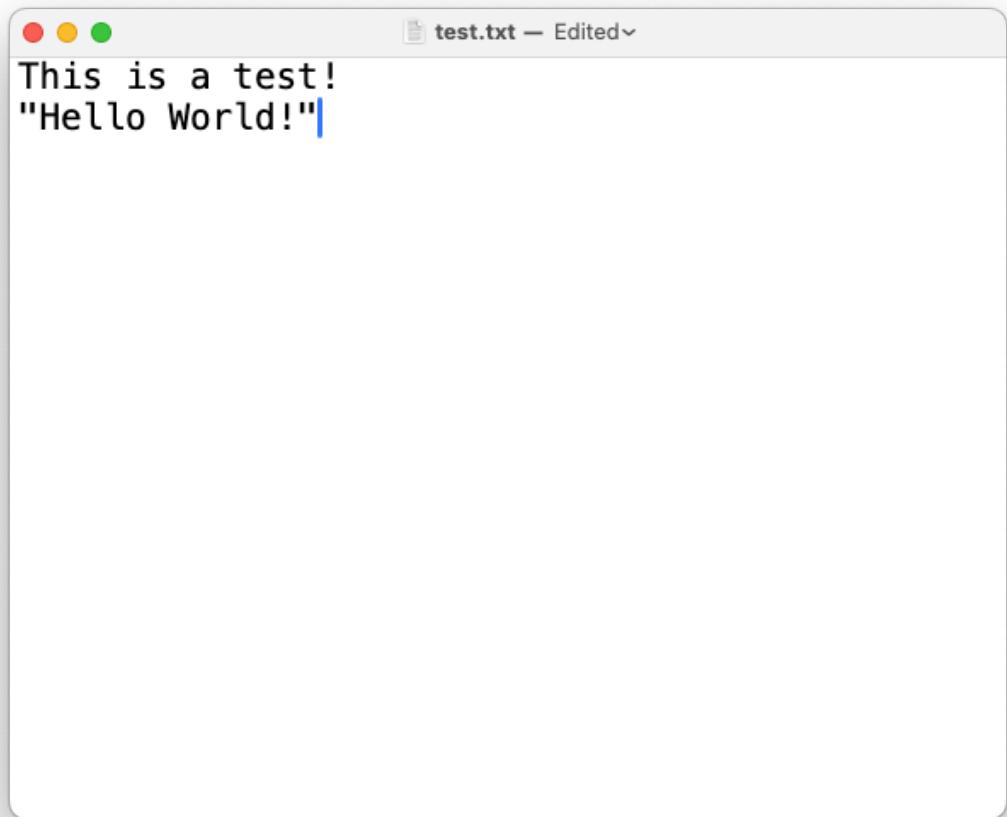
Transfer Files between the Raspi and a computer

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):

0.0.0.1 * 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:

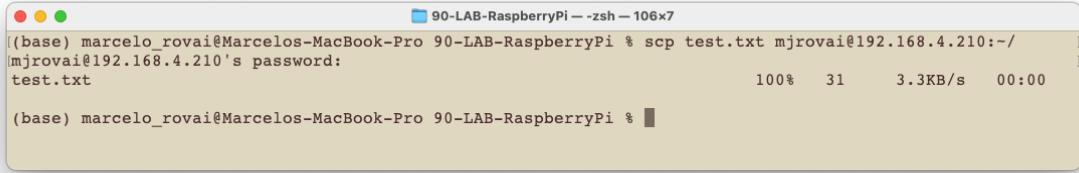
```
$ scp test.txt <username>@<pi_ip_address>:~/
```

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

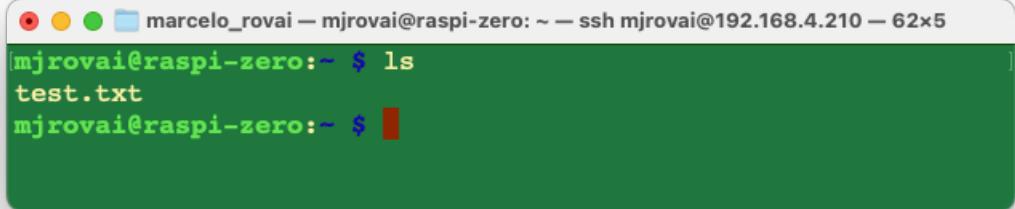
of 192.168.4.210:

```
scp test.txt mjrovai@192.168.4.210:~/
```



```
(base) marcelo_rovai@Marcelos-MacBook-Pro 90-LAB-RaspberryPi % scp test.txt mjrovai@192.168.4.210:~/  
mjrovai@192.168.4.210's password:  
test.txt  
      100%   31      3.3KB/s  00:00  
(base) marcelo_rovai@Marcelos-MacBook-Pro 90-LAB-RaspberryPi %
```

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:



```
marcelo_rovai — mjrovai@raspi-zero: ~ — ssh mjrovai@192.168.4.210 — 62x5  
[mjrovai@raspi-zero:~ $ ls  
test.txt  
mjrovai@raspi-zero:~ $ ]
```

0.0.0.0.2 * 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 .
```

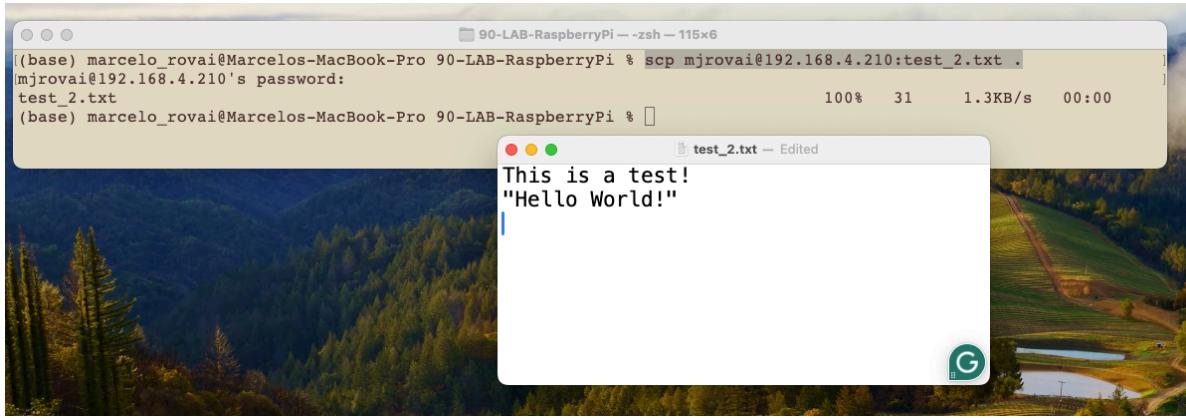
For example:

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 .
```

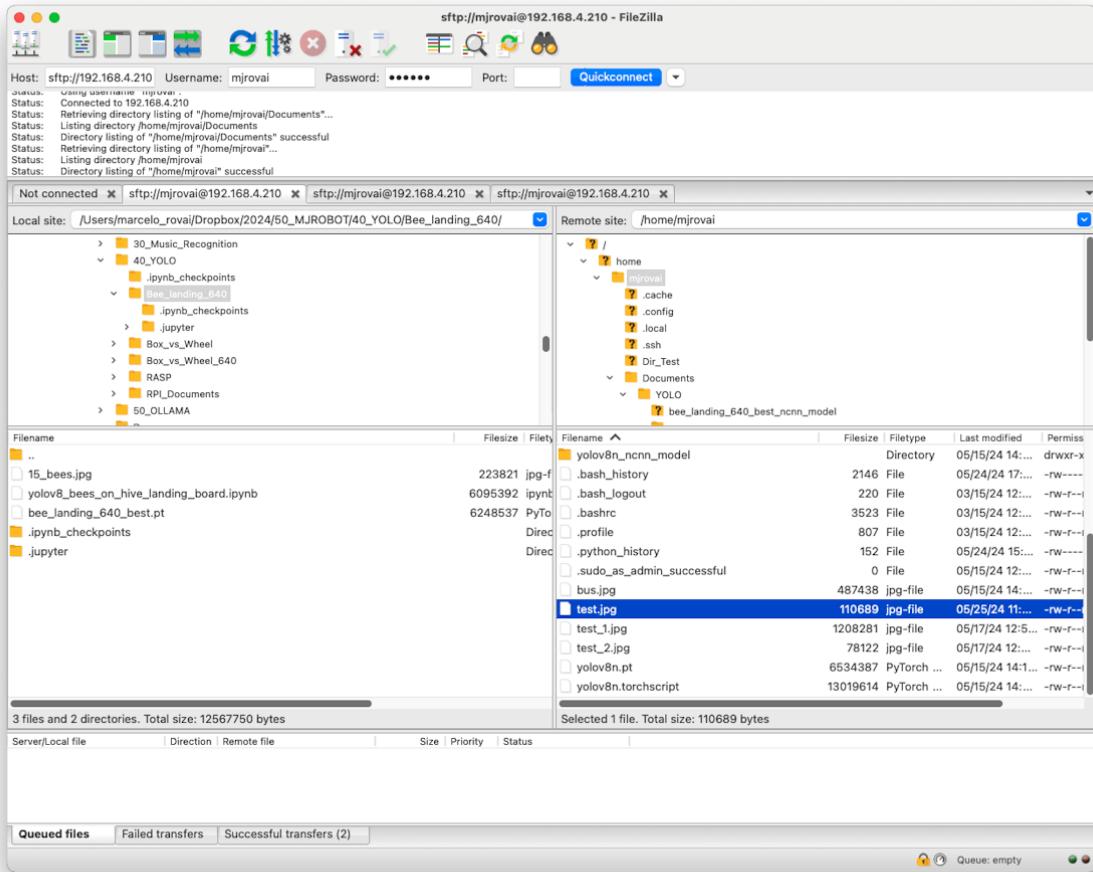


Transferring files using FTP

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
```

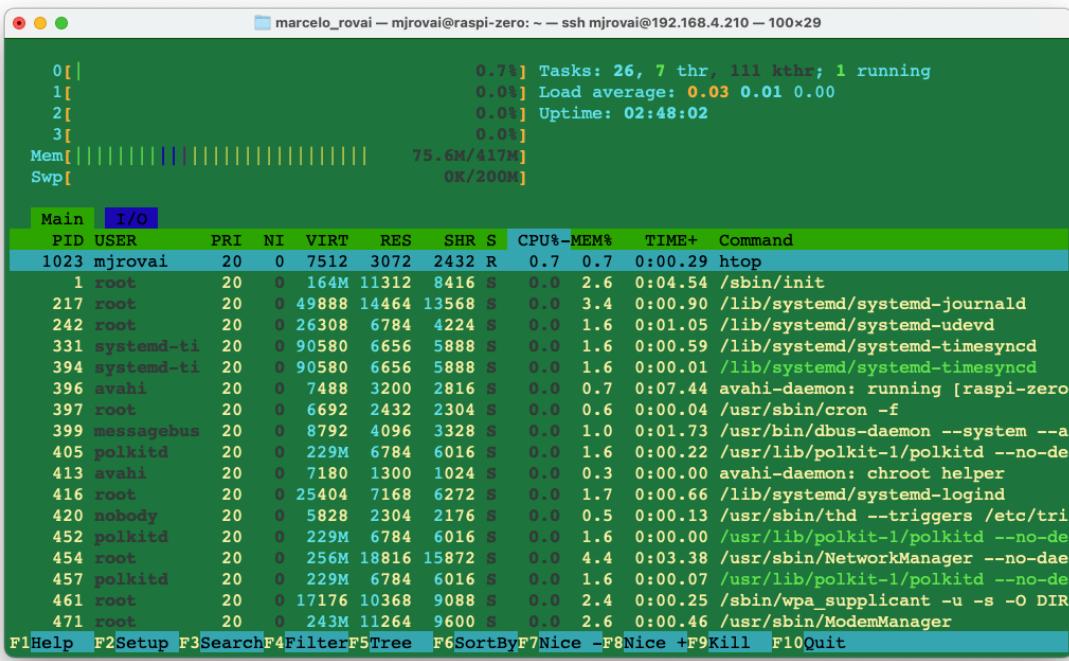
and enter your Raspi **username** and **password**. Pressing **Quickconnect** will open two windows, one for your host computer desktop (right) and another for the Raspi (left).



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 (500MB), compared to a selection of 2GB to 8GB 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 Raspi-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 100MB, which is very small for running some more complex and demanding Machine Learning applications (for example, YOLO). Let’s increase it to 2MB:

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 2GB (in my case, 1.95GB).

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.

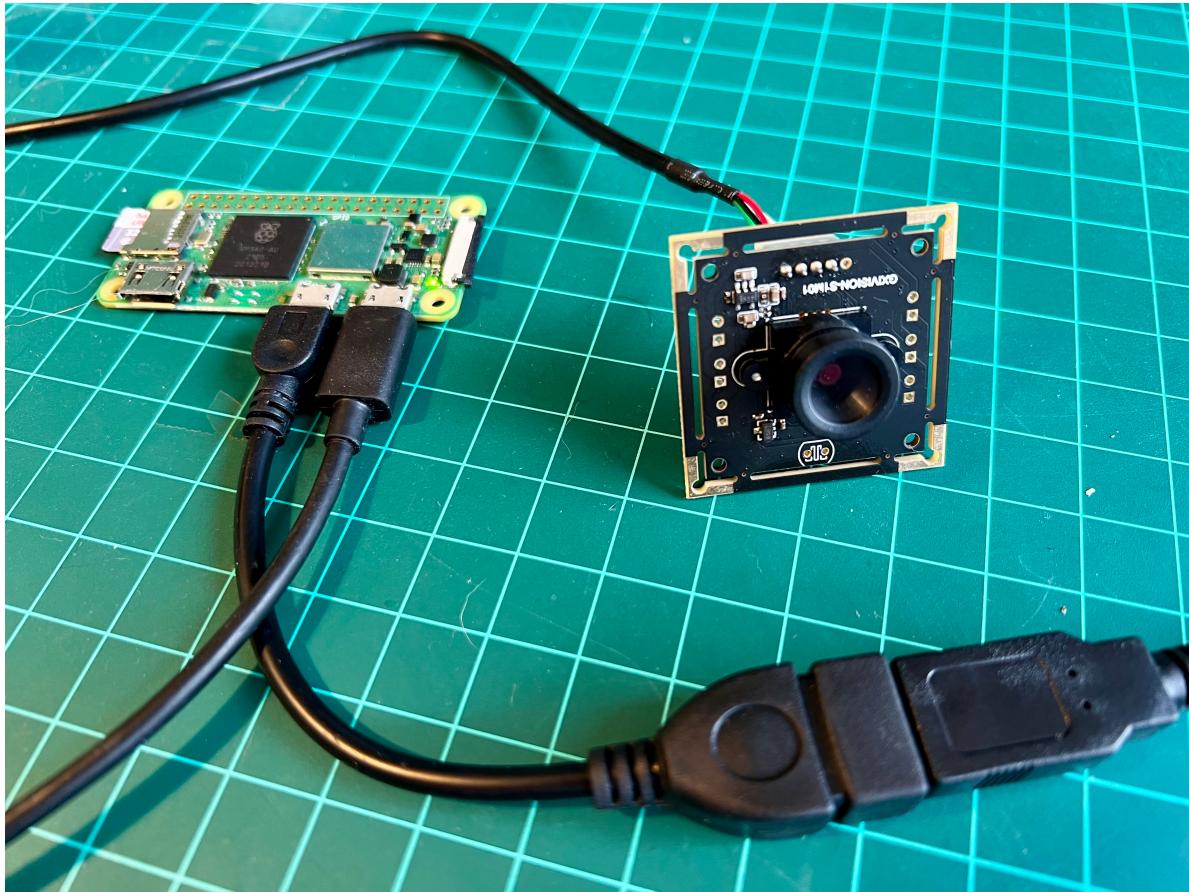
USB Webcams generally have inferior quality to the camera modules that connect to the CSI port. They can also not be controlled using the `raspistill` and `rasivid` commands in the terminal or the `picamera` recording package in Python. Nevertheless, there may be reasons why you want to connect a USB 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 now
```

2. Connect the USB Webcam (USB Camera Module 30fps,1280x720) 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_rovai — mjrovai@raspi-zero: ~ — ssh mjrovai@192.168.4.210 — 66x5
mjrovai@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
mjrovai@raspi-zero:~ $
```

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.

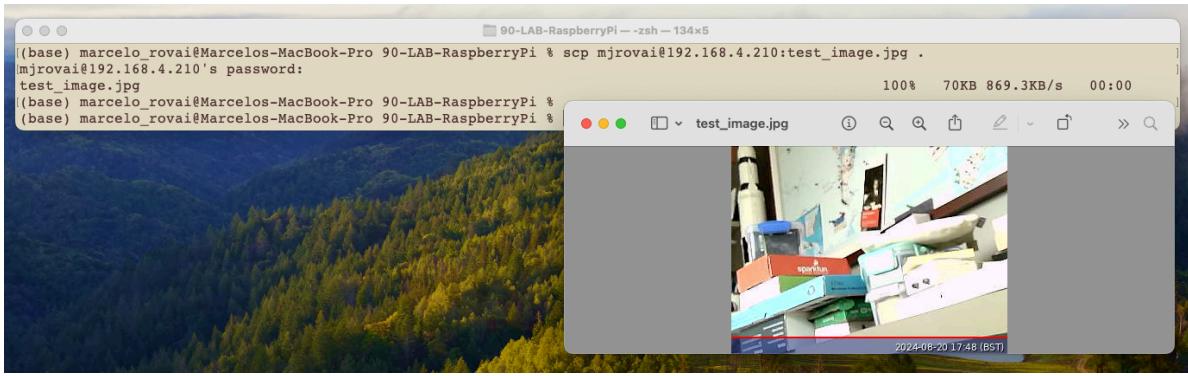
```
marcelo_rovai — mjrovai@raspi-zero: ~ — ssh mjrovai@192.168.4.210 — 85x14
mjrovai@raspi-zero:~ $ fswebcam test_image.jpg
--- Opening /dev/video0...
Trying source module v4l2...
/dev/video0 opened.
No input was specified, using the first.
Adjusting resolution from 384x288 to 320x240.
--- Capturing 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 .
```

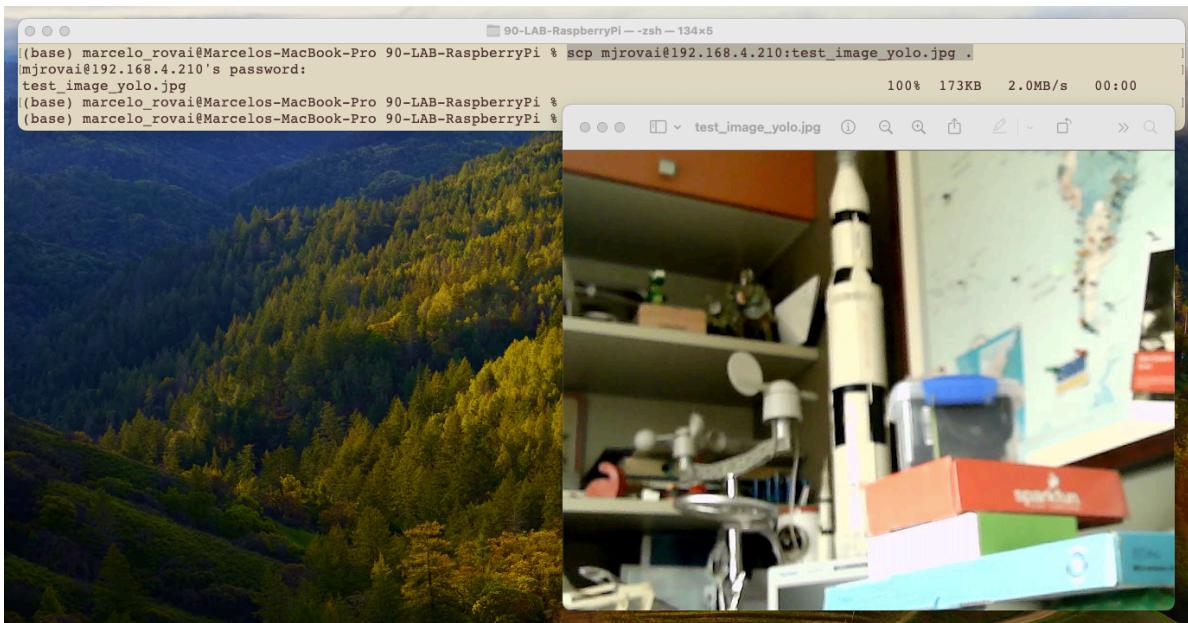
Replace “mjrovai” with your username and “raspi-zero” with Pi’s hostname.



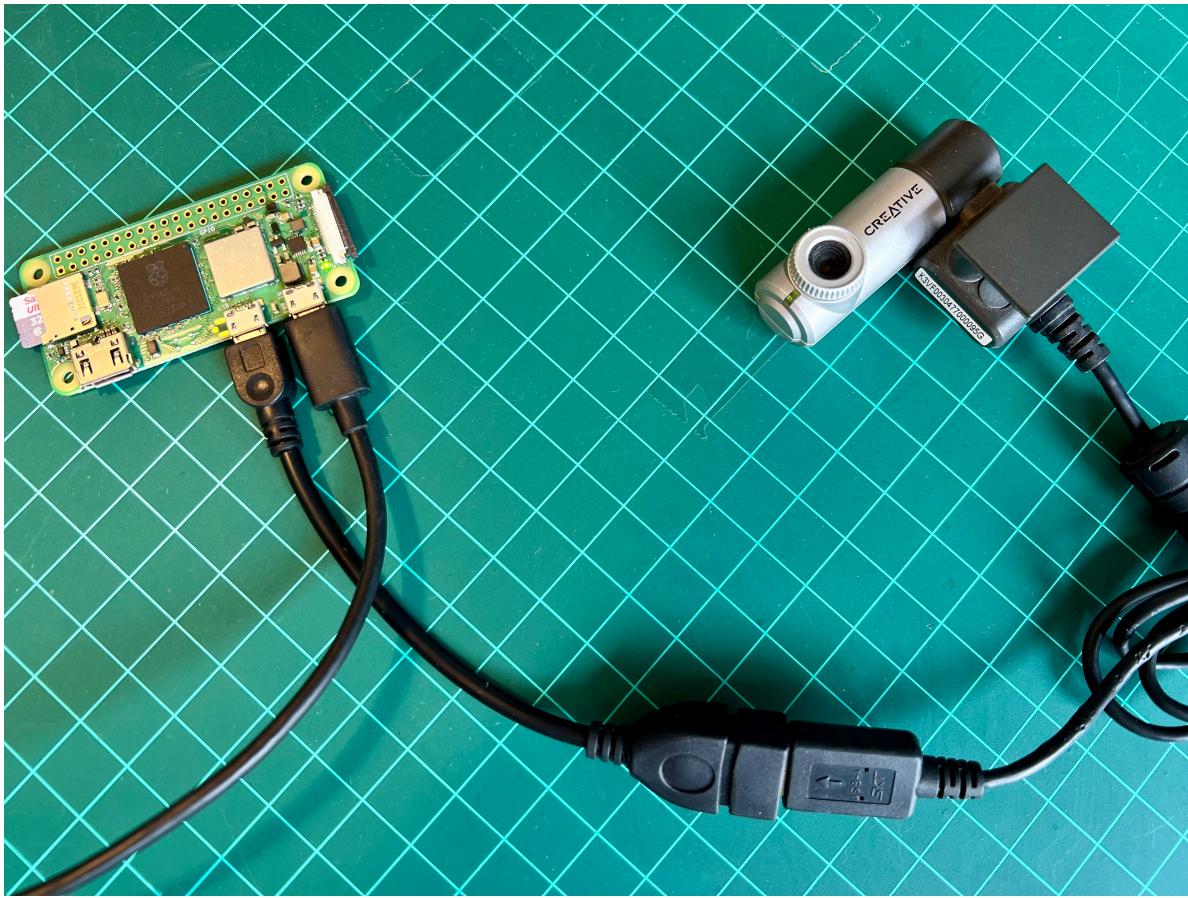
- If the image quality isn't satisfactory, you can adjust various settings; for example, define a resolution that is suitable for YOLO (640x640):

```
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

```
marcelo_rovai — mjrovai@raspi-zero: ~ — ssh mjrovai@192.168.4.210 — 85x6
mjrovai@raspi-zero:~$ lsusb
Bus 001 Device 002: ID 041e:401f Creative Technology, Ltd Webcam Notebook [PD1171]
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
mjrovai@raspi-zero:~$
```

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:

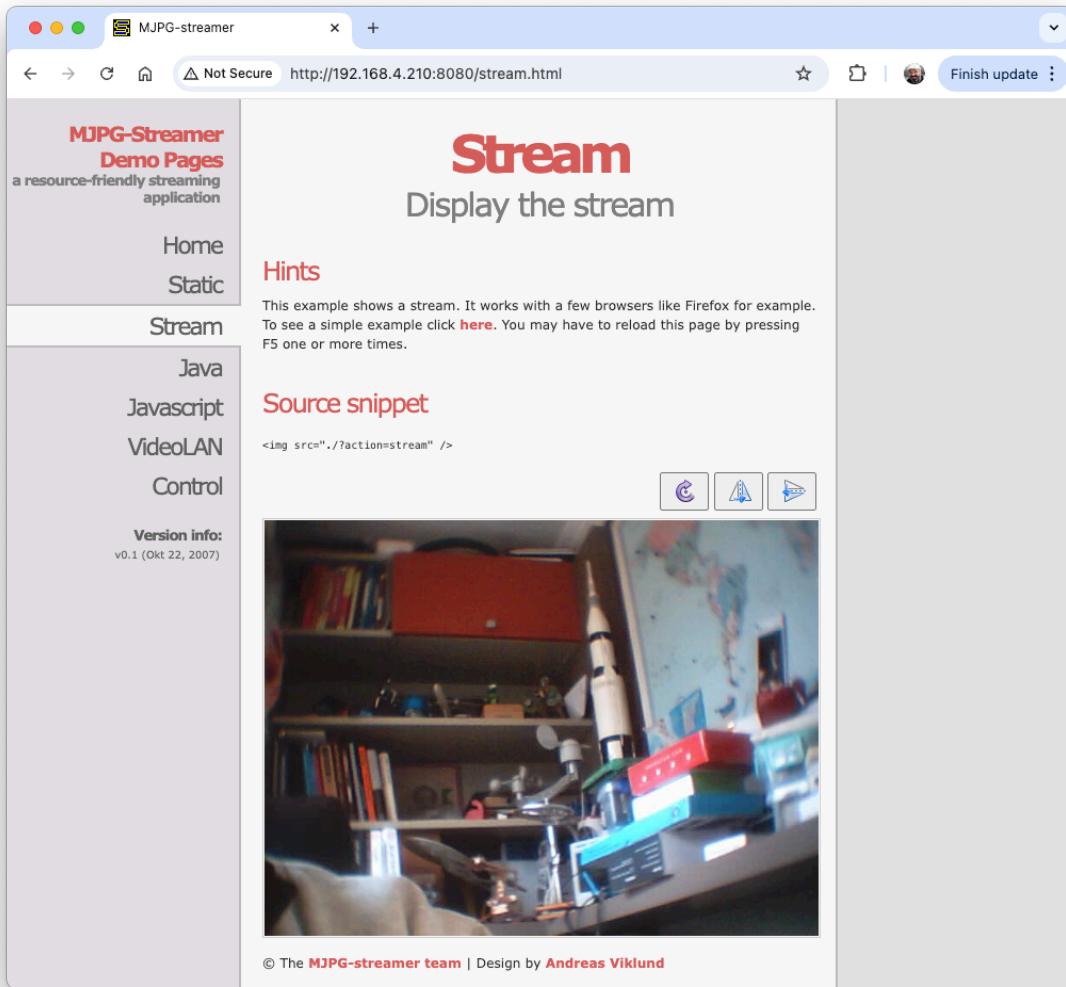
```
mjpg_streamer -i "input_uvc.so" -o "output_http.so -w ./www"
```

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
```



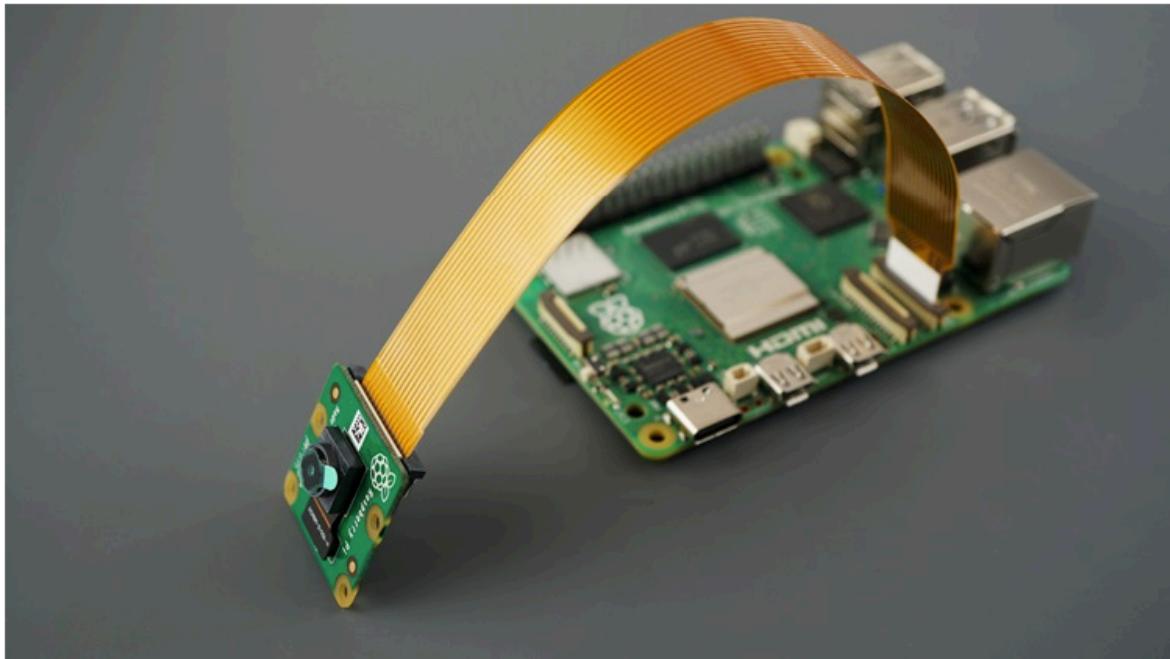
Installing a Camera Module on the CSI port

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](#), released in 2016. The latest camera model is the [12-megapixel Camera Module 3](#), released in 2023.

The original 5MP 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 Raspis, 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 5MP Arducam OV5647 camera, add the line:

```
dtoverlay=ov5647,cam0
```

Or for the v2 module, which has the 8MP Sony IMX219 camera:

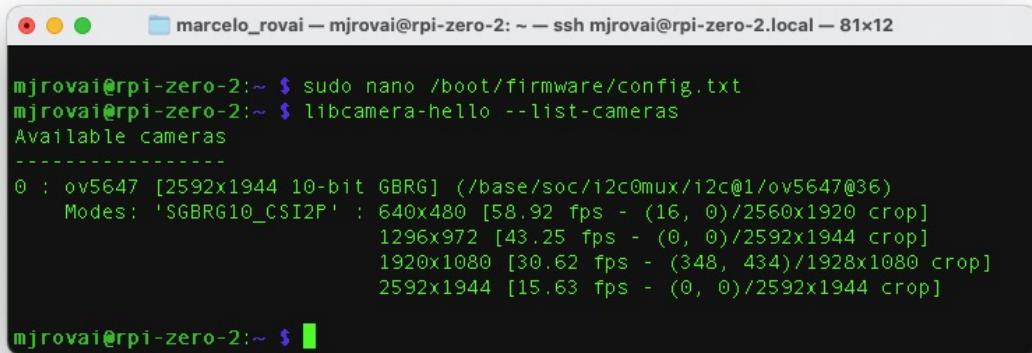
```
dtoverlay=imx219,cam0
```

Save the file (CTRL+O [ENTER] CRTL+X) and reboot the Raspi:

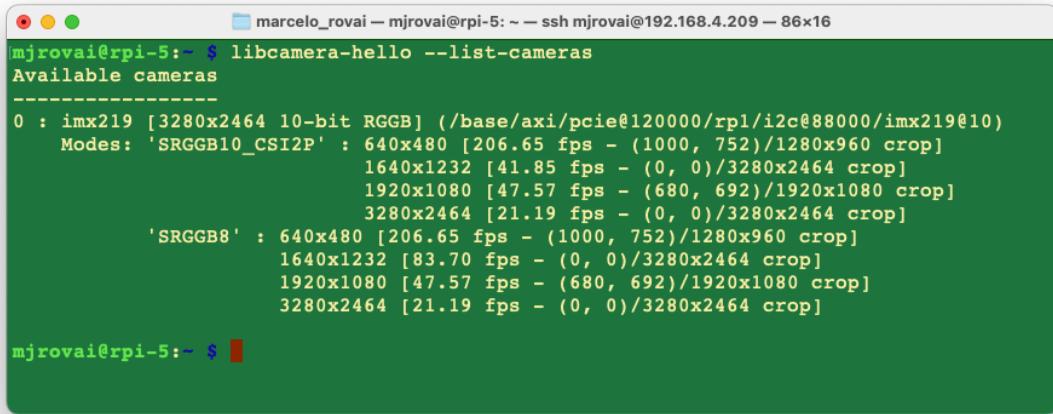
```
Sudo reboot
```

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]
```



```
marcelo_rovai ~ mjrovai@rpi-5: ~ ssh mjrovai@192.168.4.209 ~ 86x16
mjrovai@rpi-5: ~ $ libcamera-hello --list-cameras
Available cameras
-----
0 : imx219 [3280x2464 10-bit RGGB] (/base/axi/pcie@120000/rp1/i2c@88000/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]

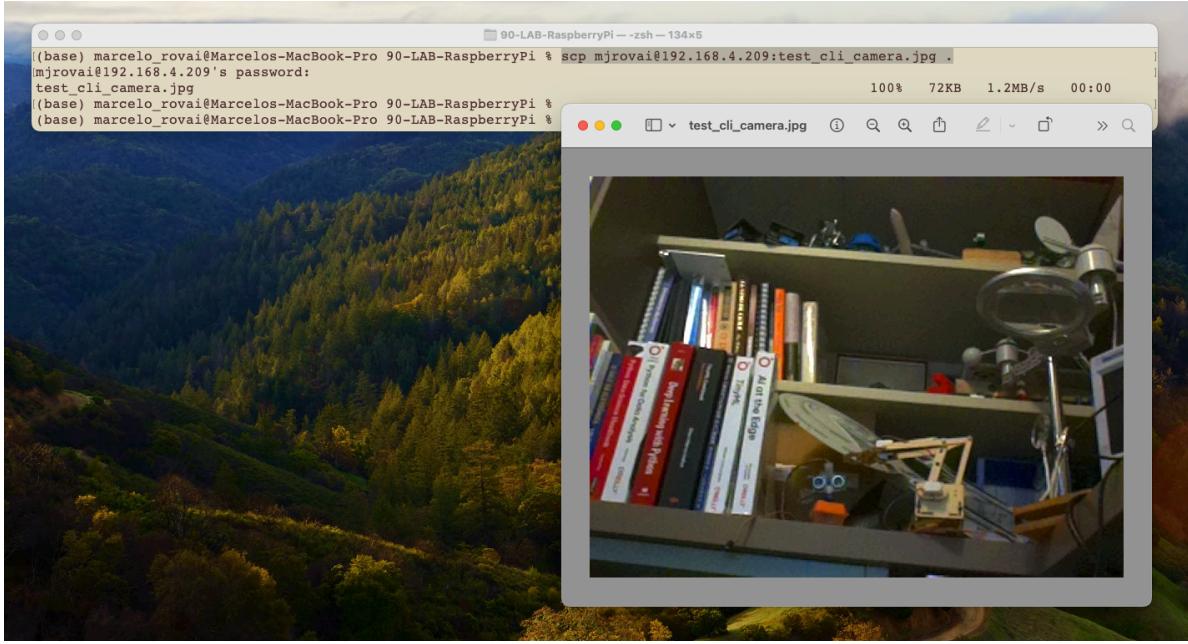
mjrovai@rpi-5: ~ $
```

[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 x 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

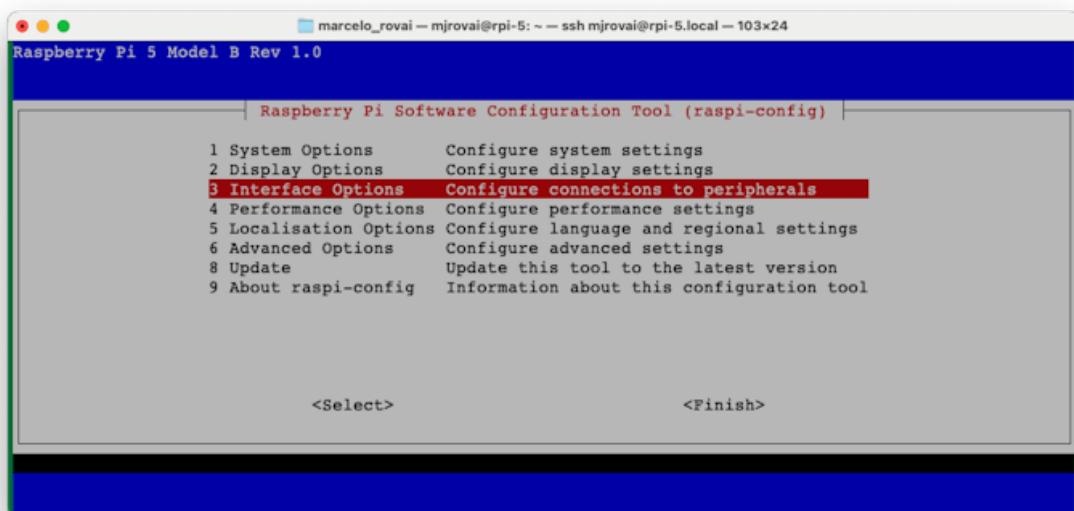
While we've primarily interacted with the Raspberry Pi using terminal commands via SSH, we can access the whole graphical desktop environment remotely if we have installed the complete Raspberry Pi OS (for example, [Raspberry Pi OS \(64-bit\)](#)). This can be particularly useful for tasks that benefit 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:

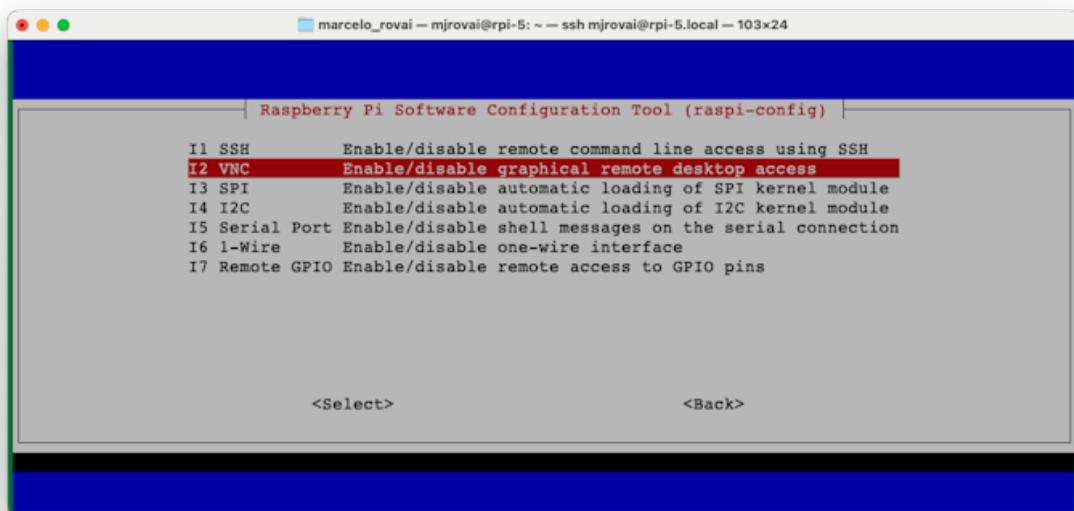
- Connect to your Raspberry Pi via SSH.
- Run the Raspberry Pi configuration tool by entering:

```
sudo raspi-config
```

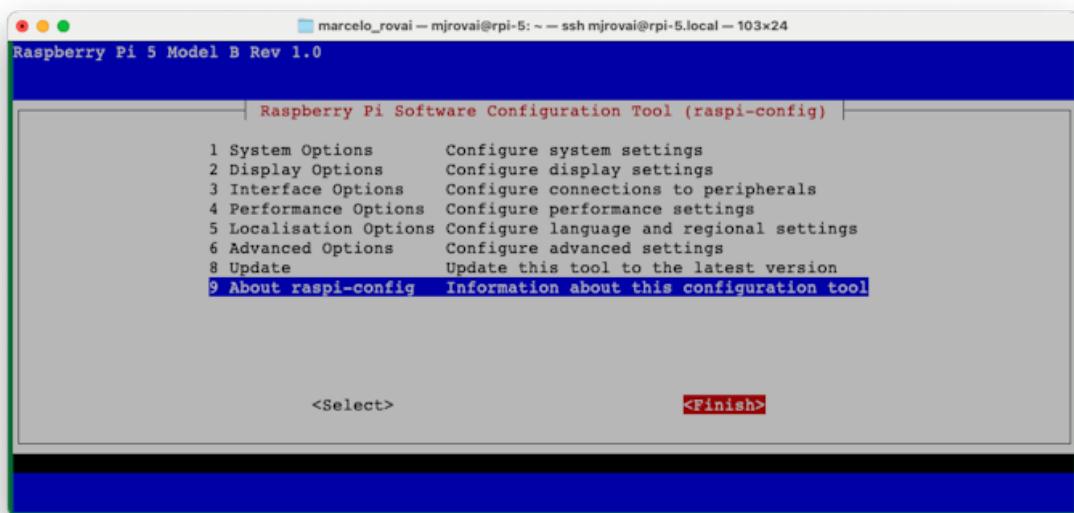
- Navigate to **Interface Options** using the arrow keys.



- 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:

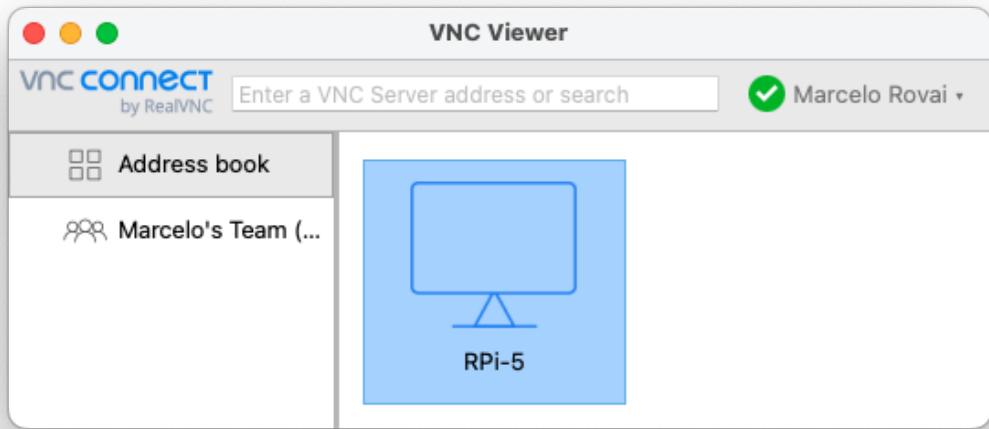
```
hostname -I
```

A screenshot of a terminal window titled "marcelo_rovai — mjrovai@rpi-5: ~ — ssh mjrovai@rpi-5.local — 57x5". The window displays the command "hostname -I" followed by its output:

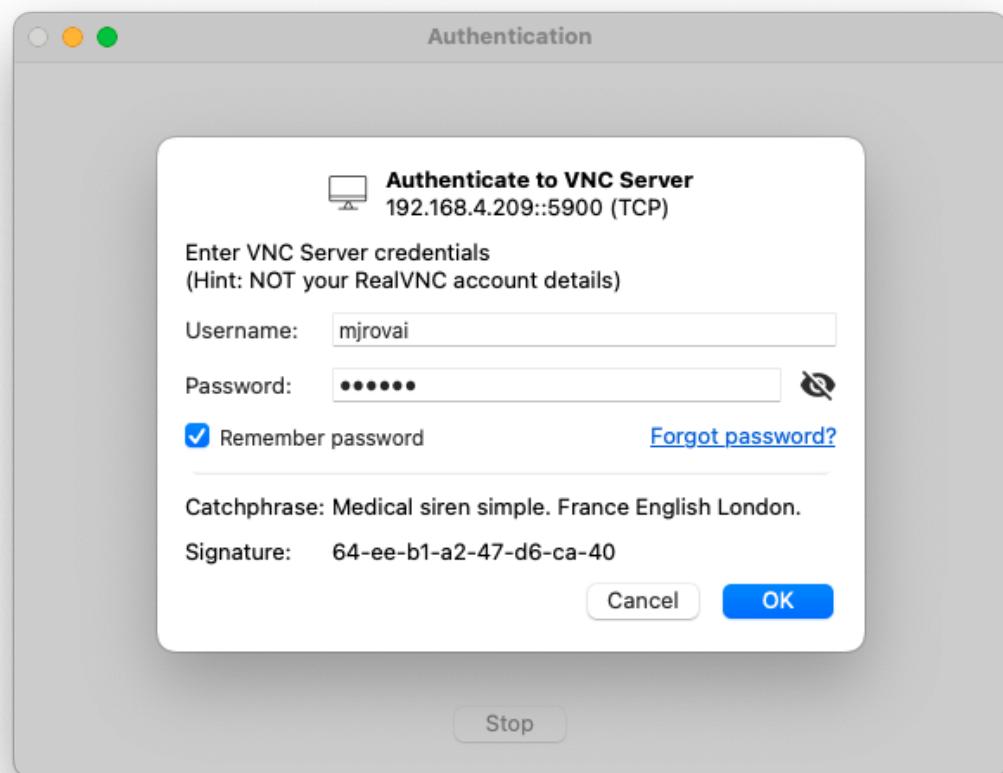
```
[mjrovai@rpi-5:~] $ hostname -I  
192.168.4.209 fde3:6154:baa3:1:af1d:2f29:d5a4:8fea  
mjrovai@rpi-5:~] $
```

4. Connect to Your Raspberry Pi:

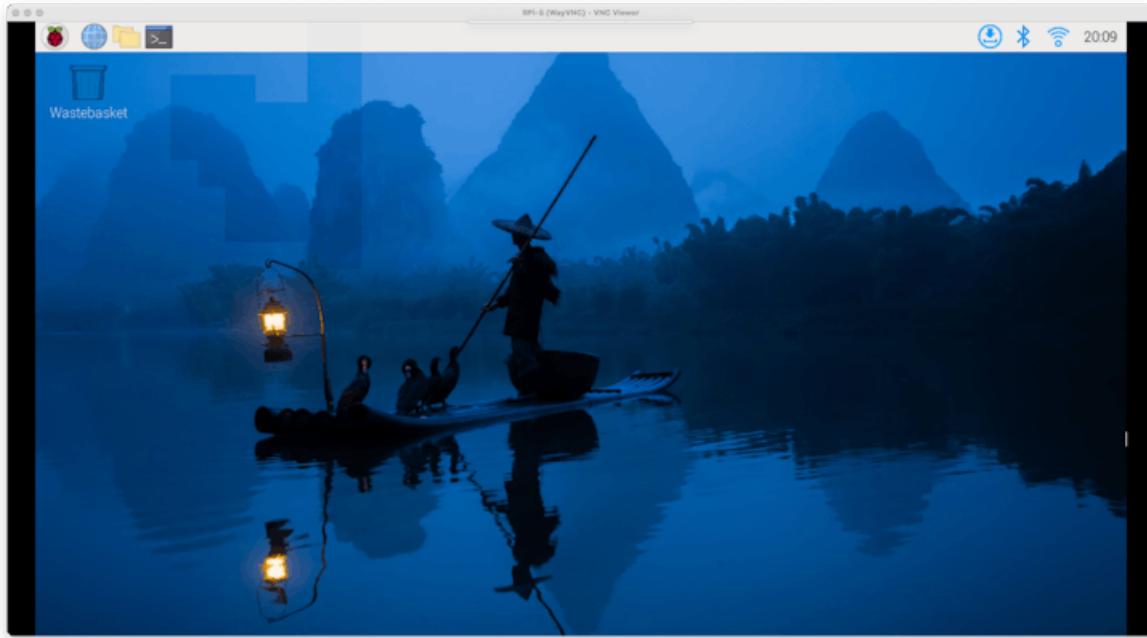
- Open your VNC viewer application.



- Enter your Raspberry Pi's IP address and hostname.
- When prompted, enter your Raspberry Pi's username and password.

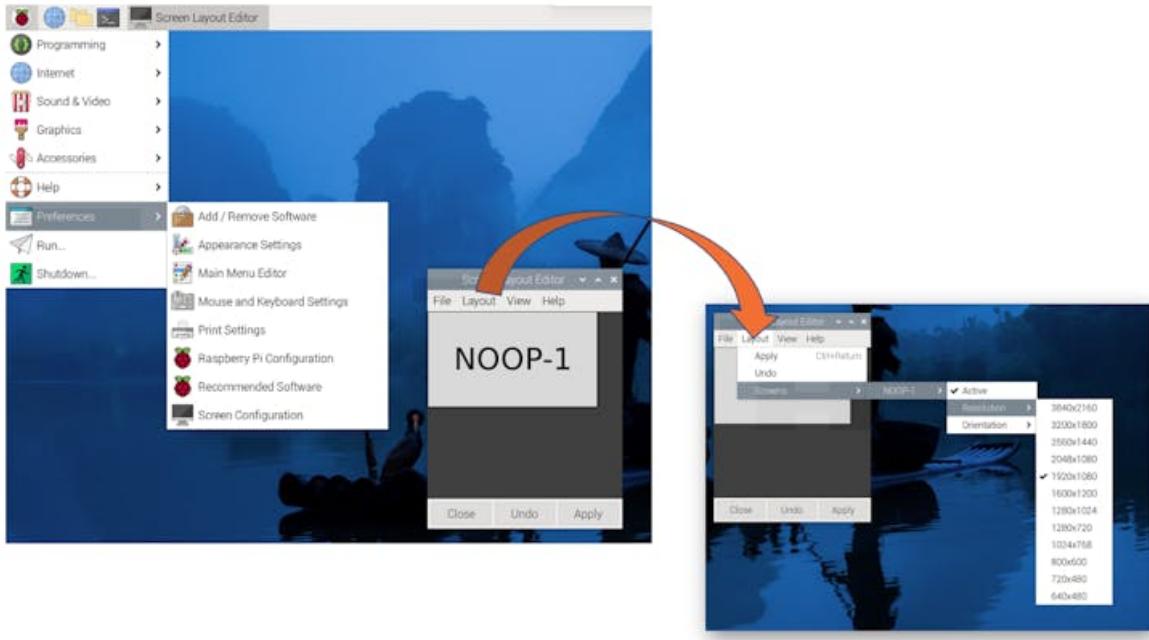


5. The Raspberry Pi 5 Desktop should appear on your computer monitor.



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:



Updating and Installing Software

1. Update your system:

```
sudo apt update && sudo apt upgrade -y
```

2. Install essential software:

```
sudo apt install python3-pip -y
```

3. Enable pip for Python projects:

```
sudo rm /usr/lib/python3.11/EXTERNALLY-MANAGED
```

Model-Specific Considerations

Raspberry Pi Zero (Raspi-Zero)

- Limited processing power, best for lightweight projects
- It is better to use a headless setup (SSH) to conserve resources.

- Consider increasing swap space for memory-intensive tasks.
- It can be used for Image Classification and Object Detection Labs but not for the LLM (SLM).

Raspberry Pi 4 or 5 (Raspi-4 or Raspi-5)

- Suitable for more demanding projects, including AI and machine learning.
- It can run the whole desktop environment smoothly.
- 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

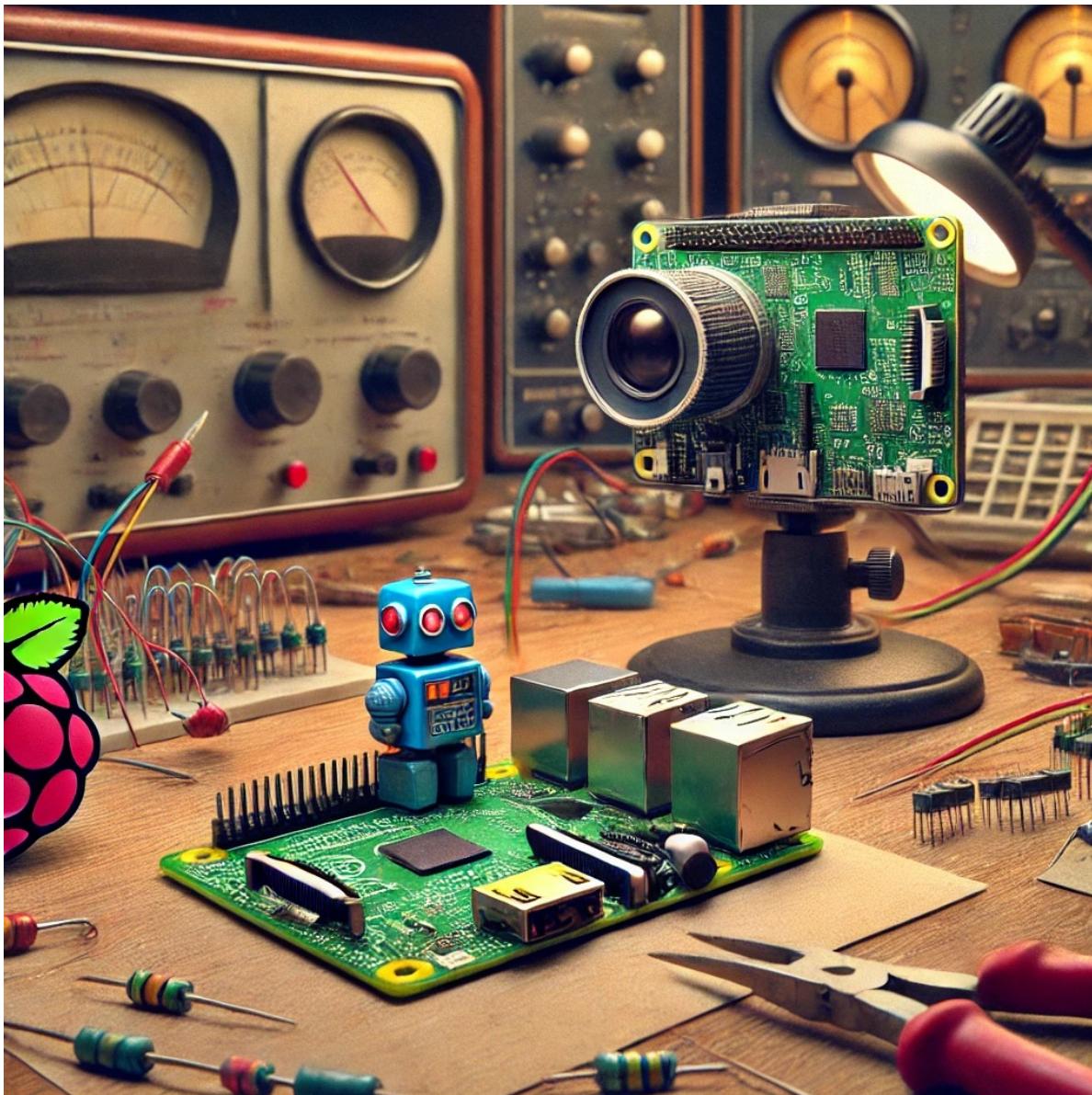


Figure 2: *DALL · E prompt - A cover image for an ‘Image Classification’ chapter in a Raspberry Pi tutorial, designed in the same vintage 1950s electronics lab style as previous covers. The scene should feature a Raspberry Pi connected to a camera module, with the camera capturing a photo of the small 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.*

Introduction

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:

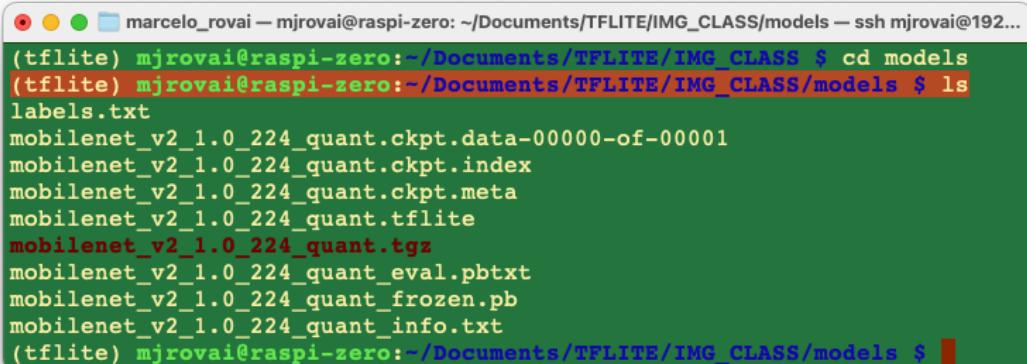
```
wget https://storage.googleapis.com/download.tensorflow.org/models/
tflite_11_05_08/mobilenet_v2_1.0_224_quant.tgz

tar xzf mobilenet_v2_1.0_224_quant.tgz
```

Get its [labels](#):

```
wget https://github.com/Mjrovai/EdgeML-with-Raspberry-Pi/blob/main/IMG_CLASS/models/labels
```

In the end, you should have the models in its directory:



The screenshot shows a terminal window with a dark background and light-colored text. It displays the command 'ls' being run in a directory named 'models'. The output of the command is a list of files and directories related to the MobileNet V2 model, including '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', and 'mobilenet_v2_1.0_224_quant_info.txt'.

```
(tflite) mjrovai@raspi-zero:~/Documents/TFLITE/IMG_CLASS/models$ cd models
(tflite) mjrovai@raspi-zero:~/Documents/TFLITE/IMG_CLASS/models$ 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)

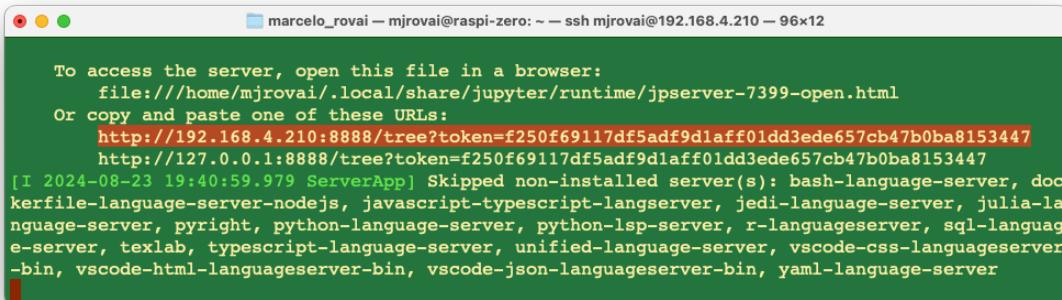
If you prefer using Jupyter Notebook for development:

```
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:

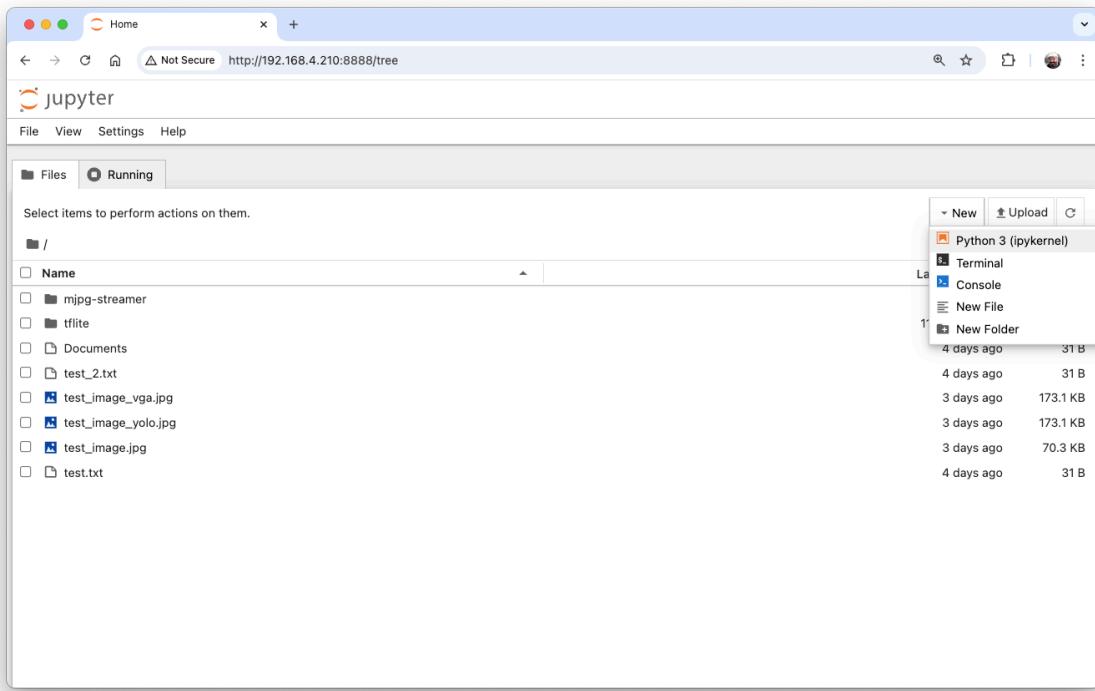


The terminal window shows the following output:

```
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-languageserver, 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.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**

```
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__)

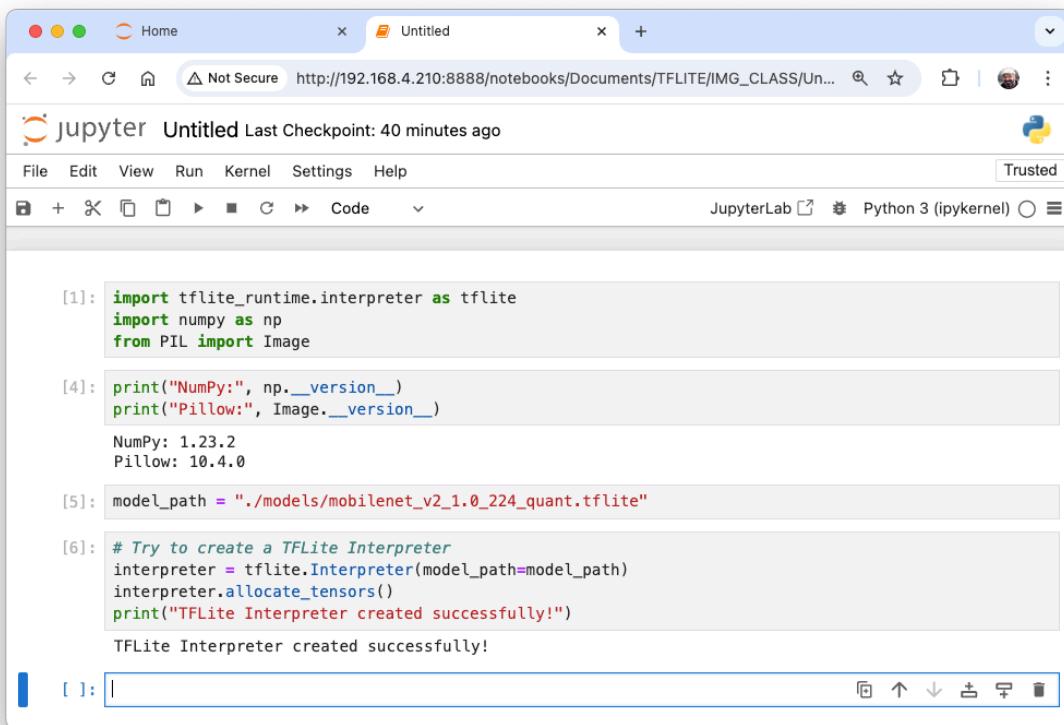
# Try to create a TFLite Interpreter
model_path = "./models/mobilenet_v2_1.0_224_quant.tflite"
interpreter = tflite.Interpreter(model_path=model_path)
interpreter.allocate_tensors()
print("TFLite Interpreter created successfully!")

^G Help      ^O Write Out ^W Where Is ^K Cut      ^T Execute
^X Exit      ^R Read File ^\ Replace   ^U Paste    ^J Justify
```

And run it with the command:

```
marcelo_rovai — mjrovai@raspi-zero: ~/Documents/TFLITE/IMG_CLASS — ssh mjrovai@192.168.4.210 — 87x6
(tflite) mjrovai@raspi-zero:~ $ cd Documents/TFLITE/IMG_CLASS/
(tflite) mjrovai@raspi-zero:~/Documents/TFLITE/IMG_CLASS $ python3 setup_test.py
NumPy: 1.23.2
Pillow: 10.4.0
TFLite Interpreter created successfully!
(tflite) mjrovai@raspi-zero:~/Documents/TFLITE/IMG_CLASS $
```

Or you can run it directly on the [Notebook](#):



The screenshot shows a Jupyter Notebook interface running on a Raspberry Pi. The browser title is "jupyter Untitled Last Checkpoint: 40 minutes ago". The notebook has one cell containing the following Python code:

```
[1]: import tensorflow as tf
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

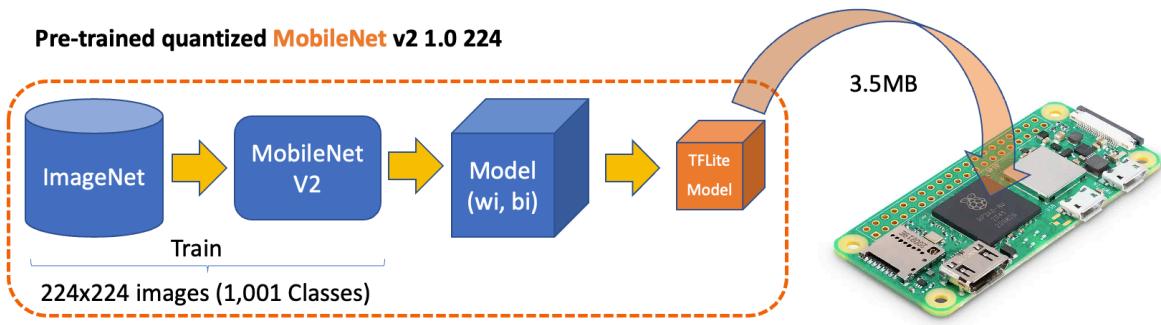
[5]: model_path = "./models/mobilenet_v2_1.0_224_quant.tflite"

[6]: # Try to create a TFLite Interpreter
interpreter = tf.lite.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 224x224 images (1.2 million) for 1,001 classes (1,000 object categories plus 1 background). The model was converted to a compact 3.5MB 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.lite_runtime.interpreter as tflite
```

Load the TFLite model and allocate tensors:

```
model_path = "./models/mobilenet_v2_1.0_224_quant.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 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 (224x224x3) 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).

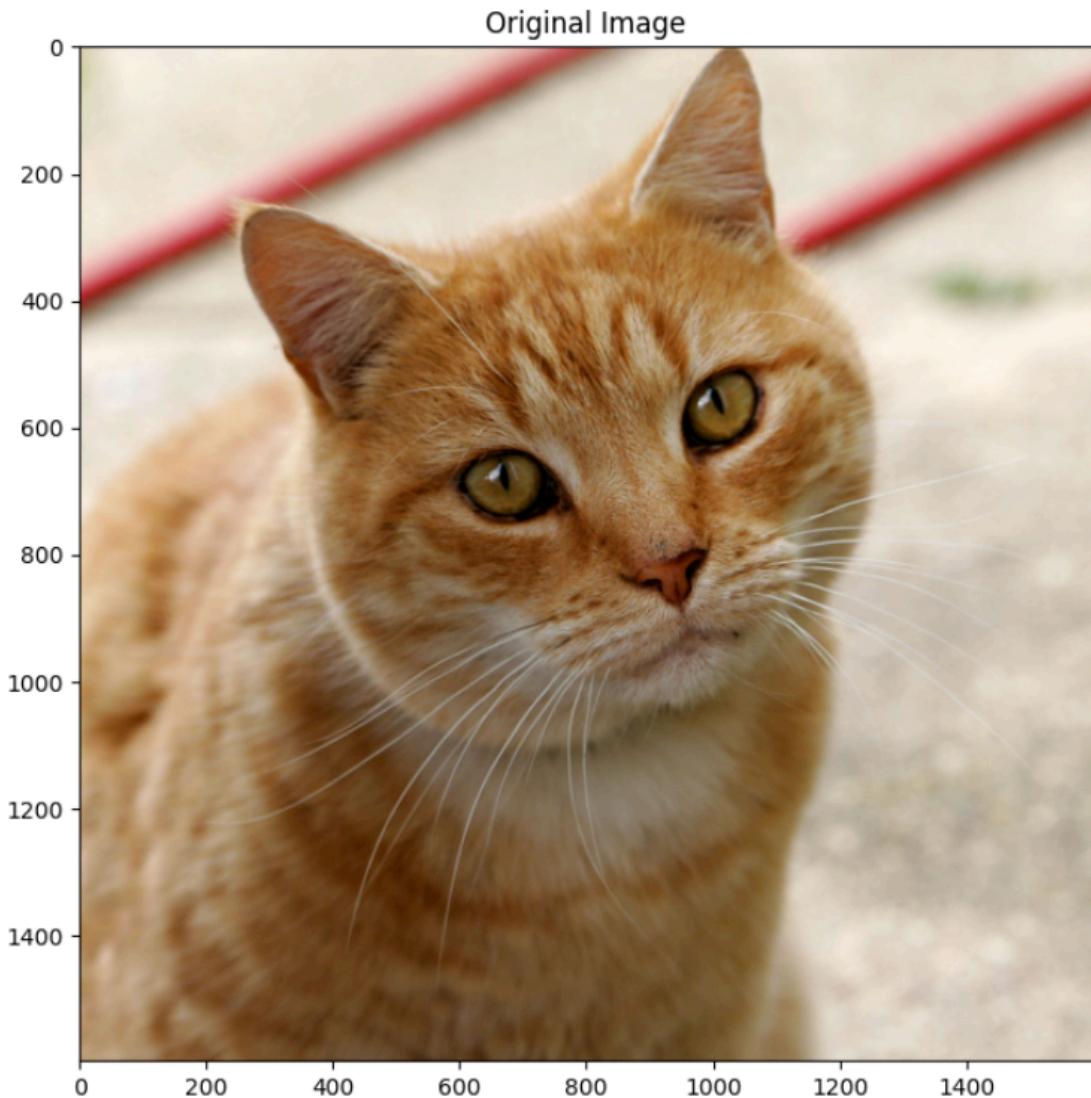
Let's get a test image. You can transfer it from your computer or download one for testing. Let's first create a folder under our working directory:

```
mkdir images
cd images
wget https://upload.wikimedia.org/wikipedia/commons/3/3a/Cat03.jpg
```

Let's load and display the image:

```
# Load he image
img_path = "./images/Cat03.jpg"
img = Image.open(img_path)

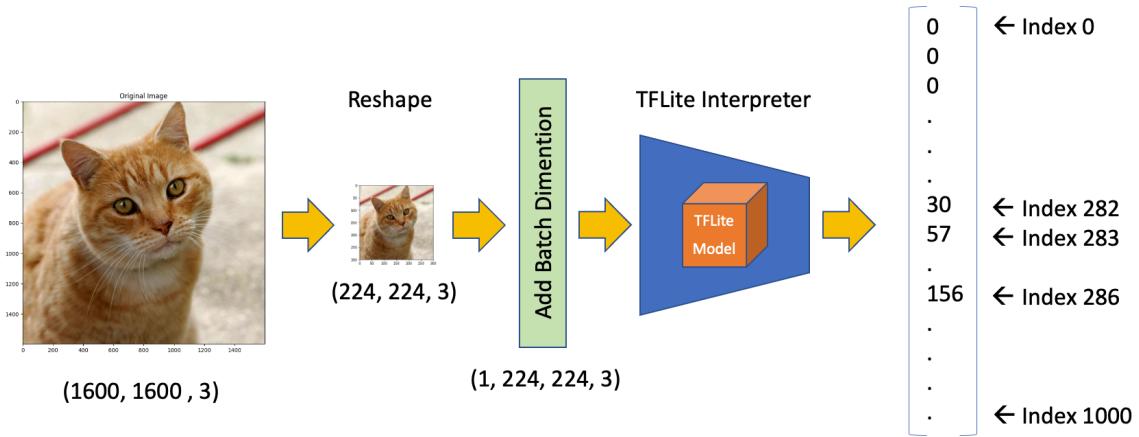
# Display the image
plt.figure(figsize=(8, 8))
plt.imshow(img)
plt.title("Original Image")
plt.show()
```



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]

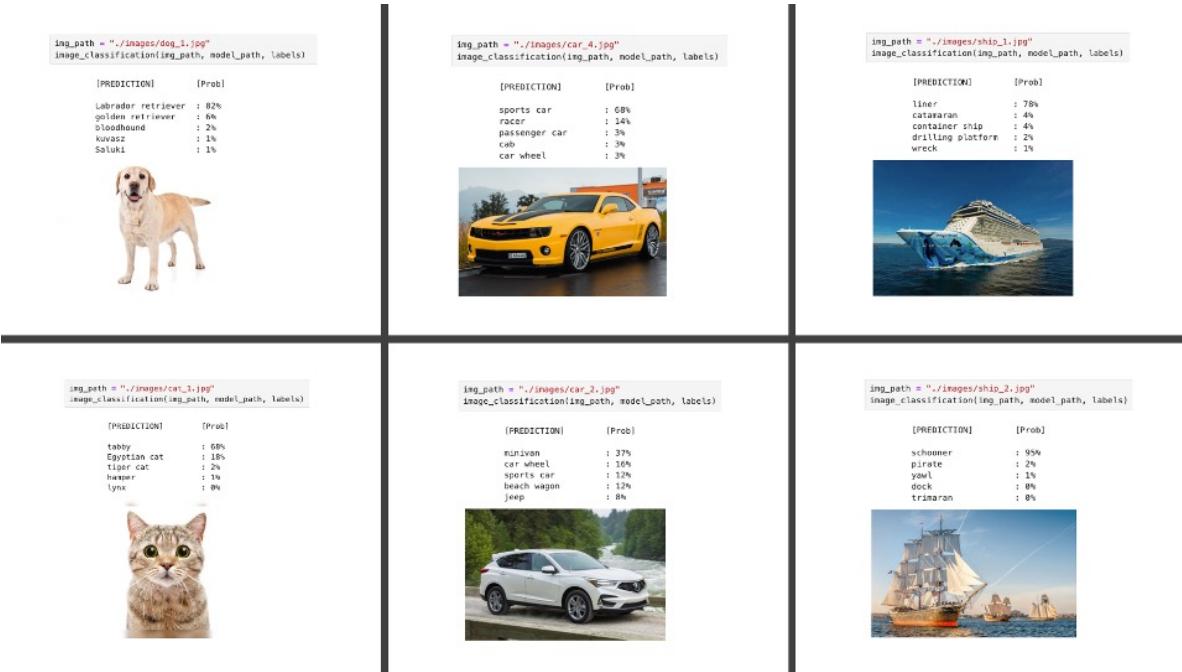
# Get quantization parameters
scale, zero_point = output_details[0]['quantization']

# 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:



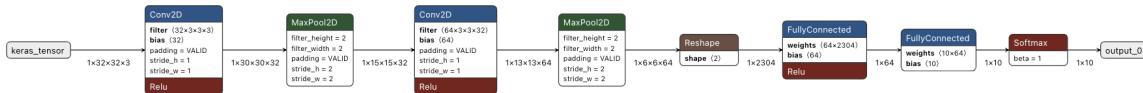
Testing with a model trained from scratch

Let's get a TFLite model trained from scratch. For that, you can follow the Notebook:

CNN to classify Cifar-10 dataset

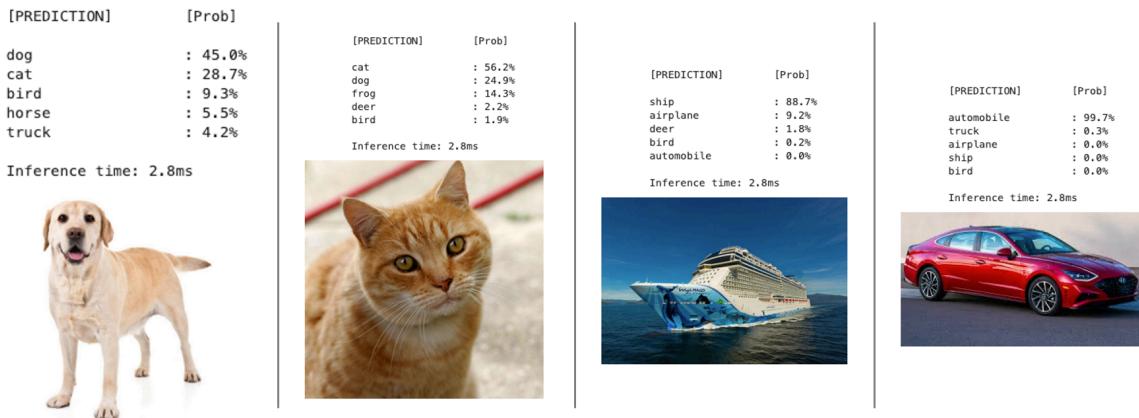
In the notebook, we trained a model using the CIFAR10 dataset, which contains 60,000 images from 10 classes of CIFAR (*airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck*). CIFAR has 32x32 color images (3 color channels) where the objects are not centered and can have the object 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__)
```

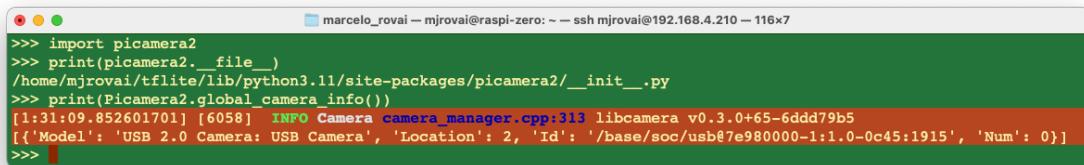
The above code will show the file location of the `picamera2` module itself, proving that the library can be accessed from the environment.

```
/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_rovai — mjrovai@raspi-zero: ~ — ssh mjrovai@192.168.4.210 — 116x7
>>> import picamera2
>>> print(picamera2.__file__)
/home/mjrovai/tflite/lib/python3.11/site-packages/picamera2/__init__.py
>>> print(Picamera2.global_camera_info())
[1:31:09.852601701] [6058] INFO Camera camera_manager.cpp:313 libcamera v0.3.0+65-6ddd79b5
[{'Model': 'USB 2.0 Camera: USB Camera', 'Location': 2, 'Id': '/base/soc/usb@7e980000-1:1.0-0c45:1915', 'Num': 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)

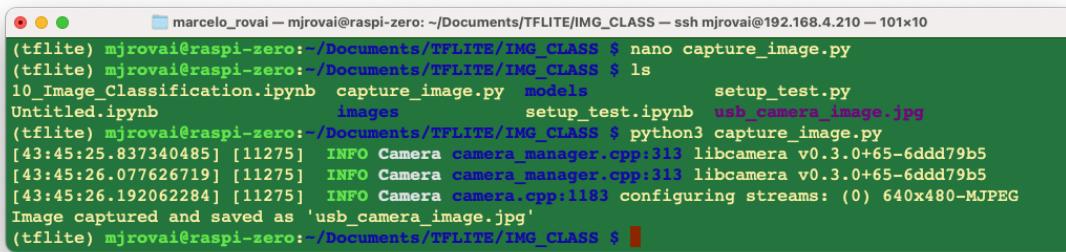
# Start the camera
picam2.start()

# 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'")
```

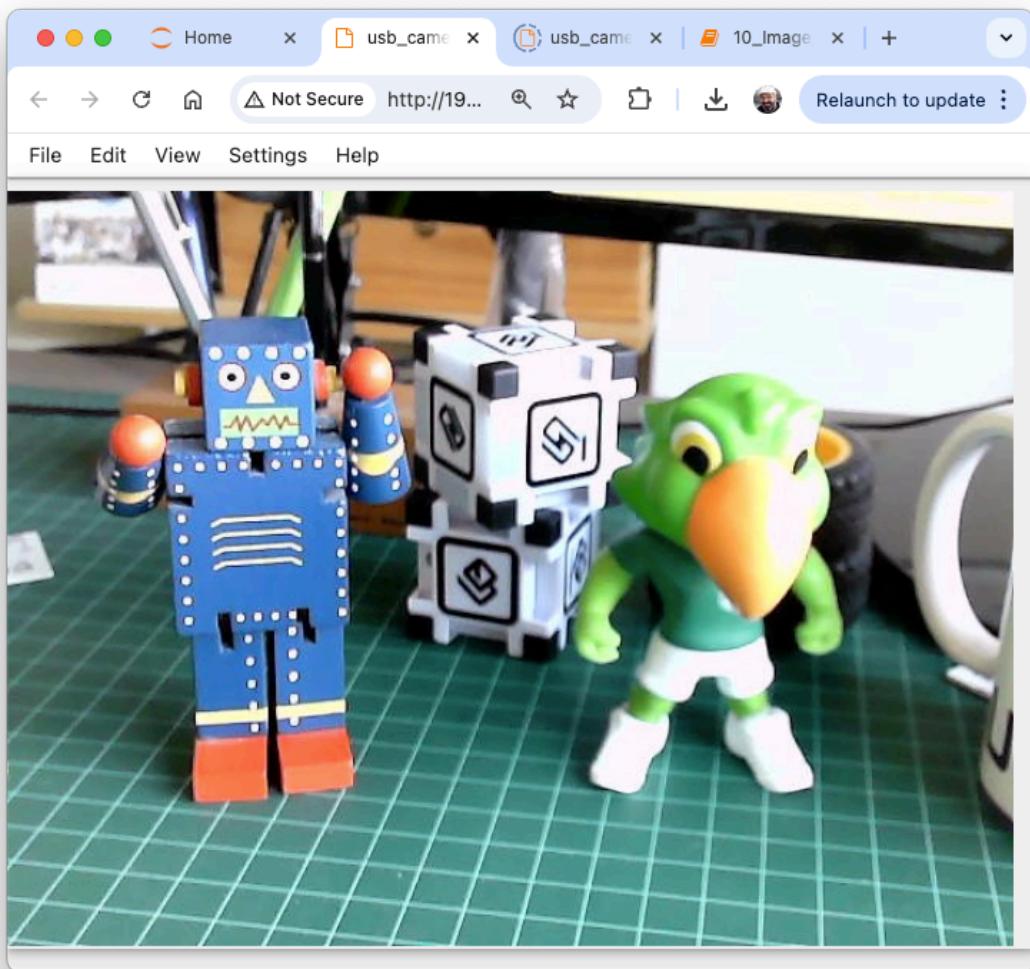
```
# Stop the camera  
picam2.stop()
```

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.



```
marcelo_rovai — mjrovai@raspi-zero: ~/Documents/TFLITE/IMG_CLASS — ssh mjrovai@192.168.4.210 — 101x10  
(tflite) mjrovai@raspi-zero:~/Documents/TFLITE/IMG_CLASS $ nano capture_image.py  
(tflite) mjrovai@raspi-zero:~/Documents/TFLITE/IMG_CLASS $ ls  
10_Image_Classification.ipynb  capture_image.py  models      setup_test.py  
Untitled.ipynb                 images          setup_test.ipynb  usb_camera_image.jpg  
(tflite) mjrovai@raspi-zero:~/Documents/TFLITE/IMG_CLASS $ python3 capture_image.py  
[43:45:25.837340485] [11275] INFO Camera camera_manager.cpp:313 libcamera v0.3.0+65-6ddd79b5  
[43:45:26.077626719] [11275] INFO Camera camera_manager.cpp:313 libcamera v0.3.0+65-6ddd79b5  
[43:45:26.192062284] [11275] INFO Camera camera.cpp:1183 configuring streams: (0) 640x480-MJPEG  
Image captured and saved as 'usb_camera_image.jpg'  
(tflite) mjrovai@raspi-zero:~/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.



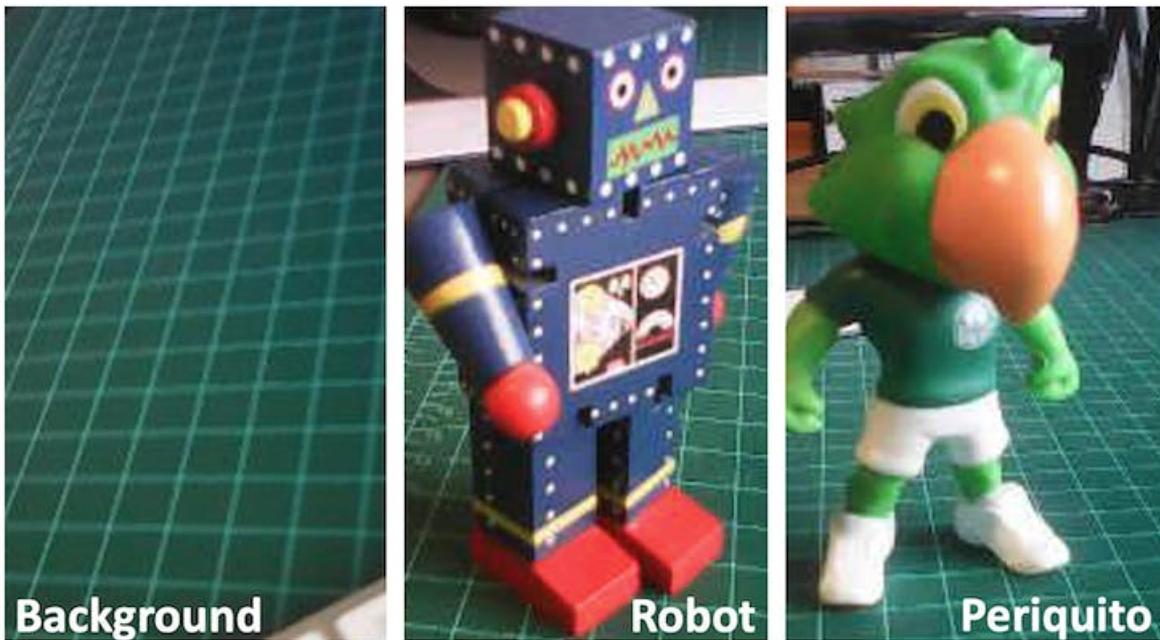
If you are working with a Raspi-5 with a whole desktop, you can open the file directly on the device.

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>

```

```

        </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>
    ''')

```

```

        </html>
    '', 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)

```

4. Run this script:

```
python3 get_img_data.py
```

5. 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:

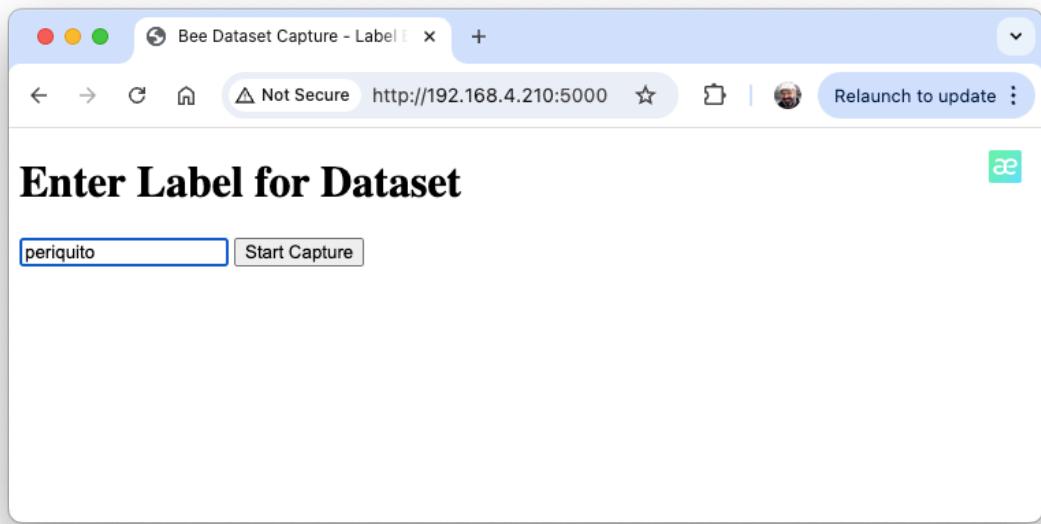
1. **Flask Web Application:** Handles routing and serves the web interface.
2. **Picamera2 Integration:** Controls the Raspberry Pi camera.
3. **Threaded Frame Capture:** Ensures smooth live preview.
4. **File Management:** Organizes captured images into labeled directories.

Key Functions:

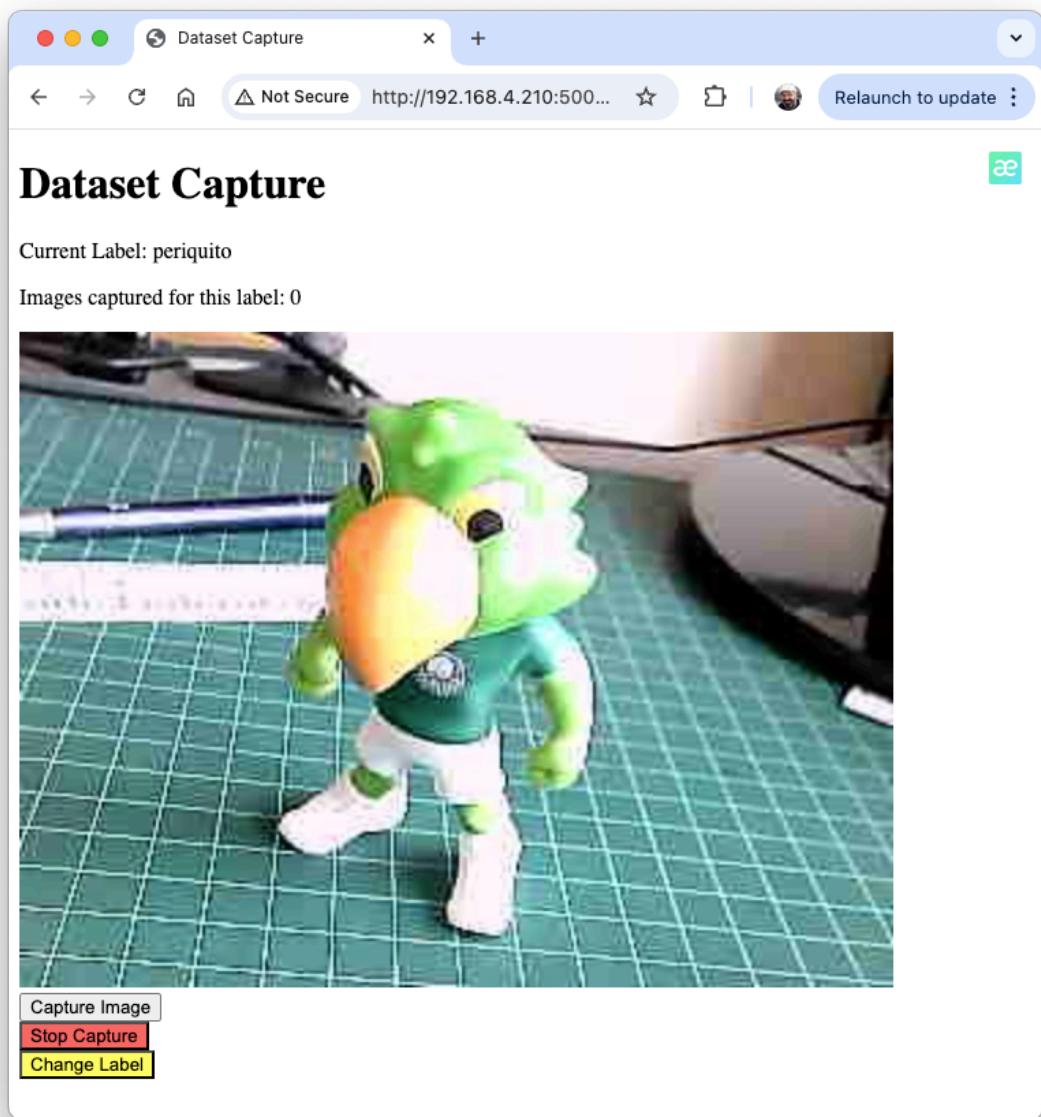
- `initialize_camera()`: Sets up the Picamera2 instance.
- `get_frame()`: Continuously captures frames for the live preview.
- `generate_frames()`: Yields frames for the live video feed.
- `shutdown_server()`: Sets the shutdown event, stops the camera, and shuts down the Flask server
- `index()`: Handles 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:

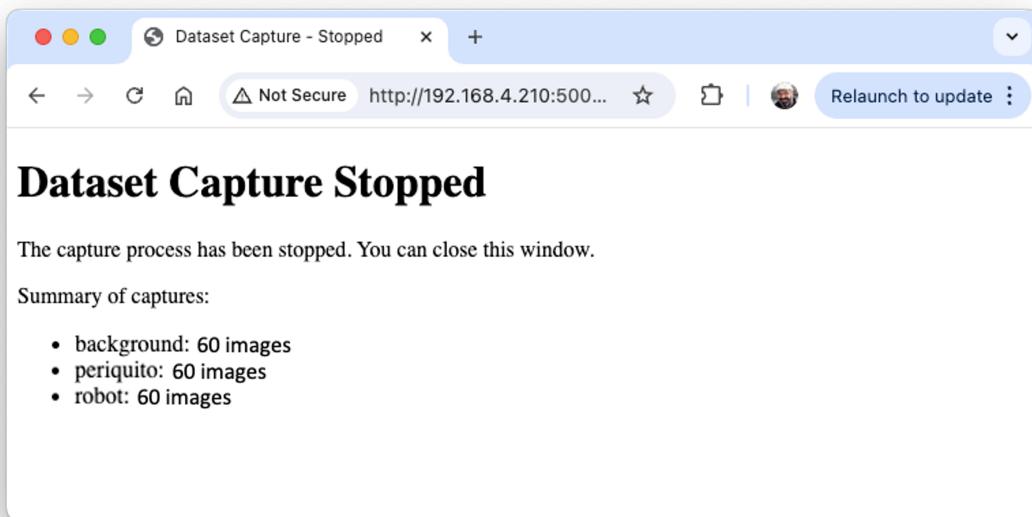
1. Start the script on your Raspberry Pi.
2. Access the web interface from a browser.
3. Enter a label for the images you want to capture and press **Start Capture**.



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 (320X240).
- Modify the HTML templates for a different look and feel.
- Add additional image processing or analysis steps in the `capture_image()` function.

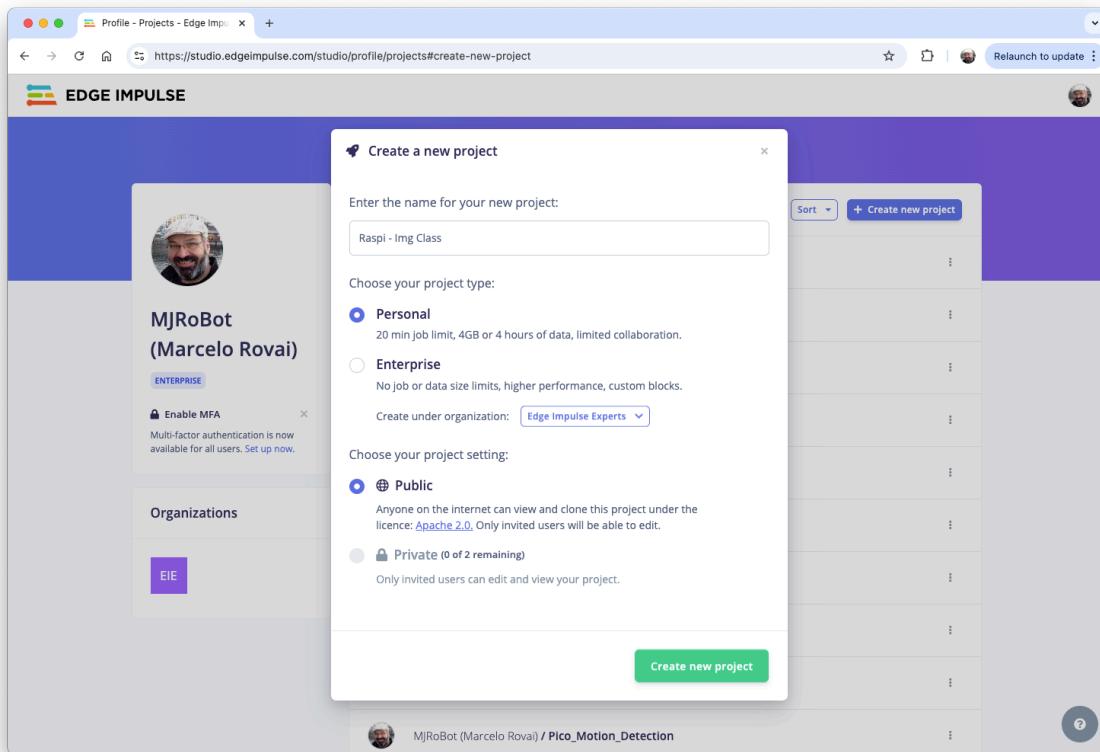
Number of samples on Dataset:

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](#).

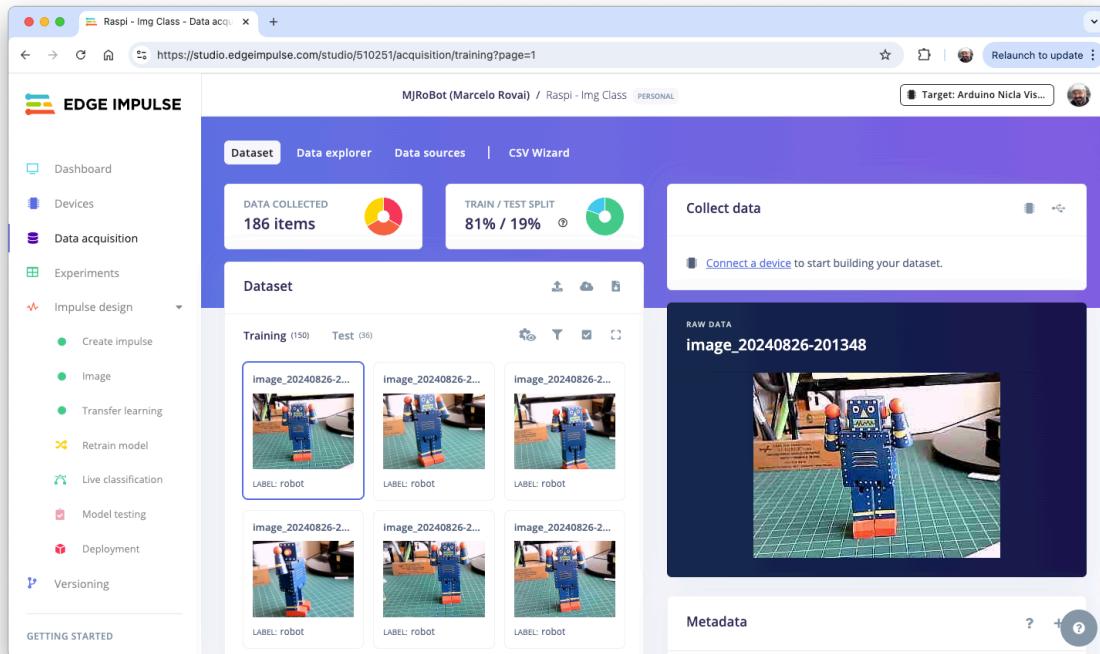
Dataset

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).

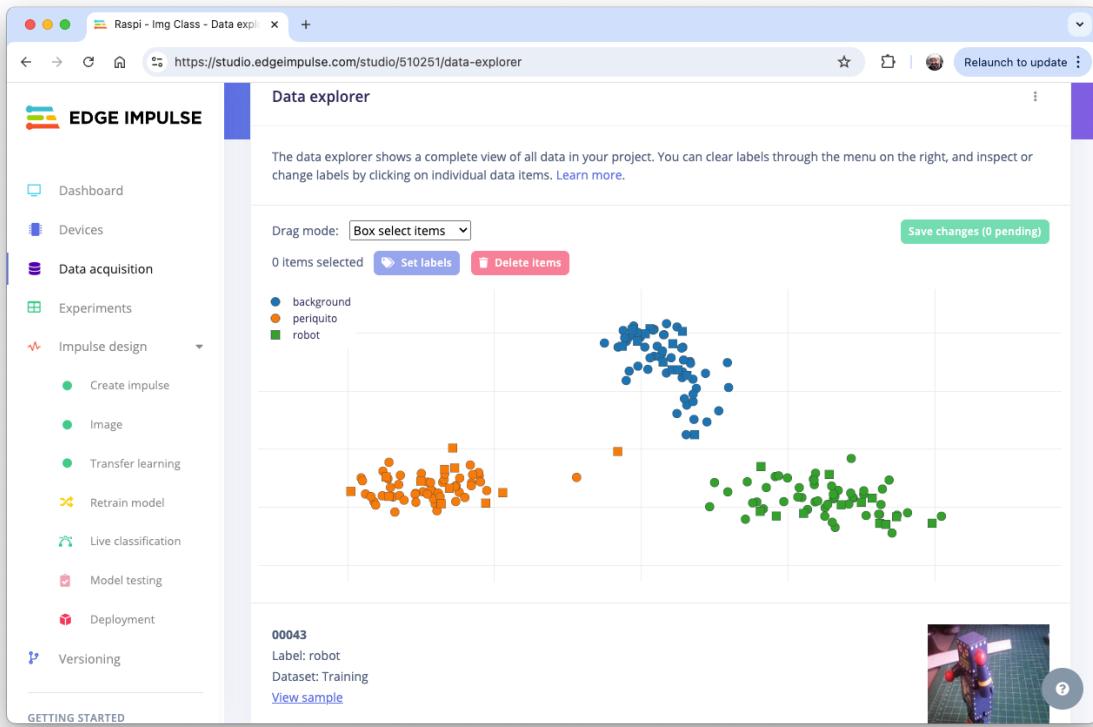
Regarding the Dataset, it is essential to point out that our Original Dataset, captured with the Raspi, will be split into *Training*, *Validation*, and *Test*. The Test Set will be separated from the beginning and reserved for use only in the Test phase after training. The Validation Set will be used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

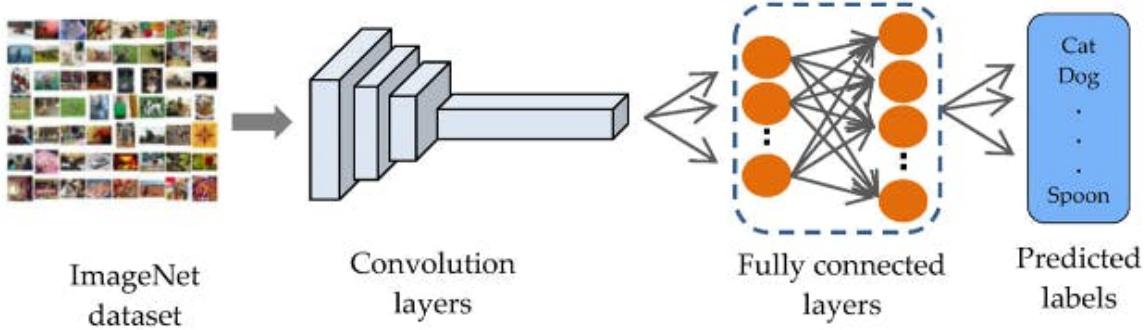


The Impulse Design

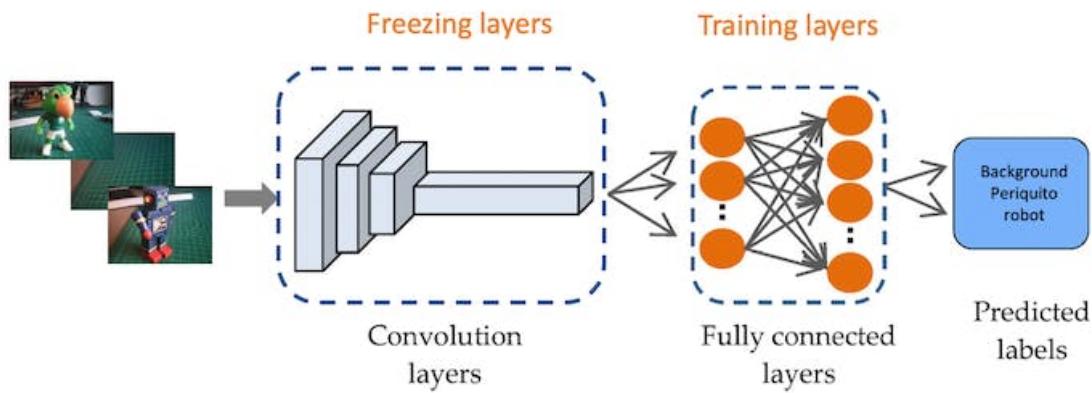
In this phase, we should define how to:

- 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 160x160 and squashing them (squared form, without cropping). Select Image and Transfer Learning blocks. Save the Impulse.

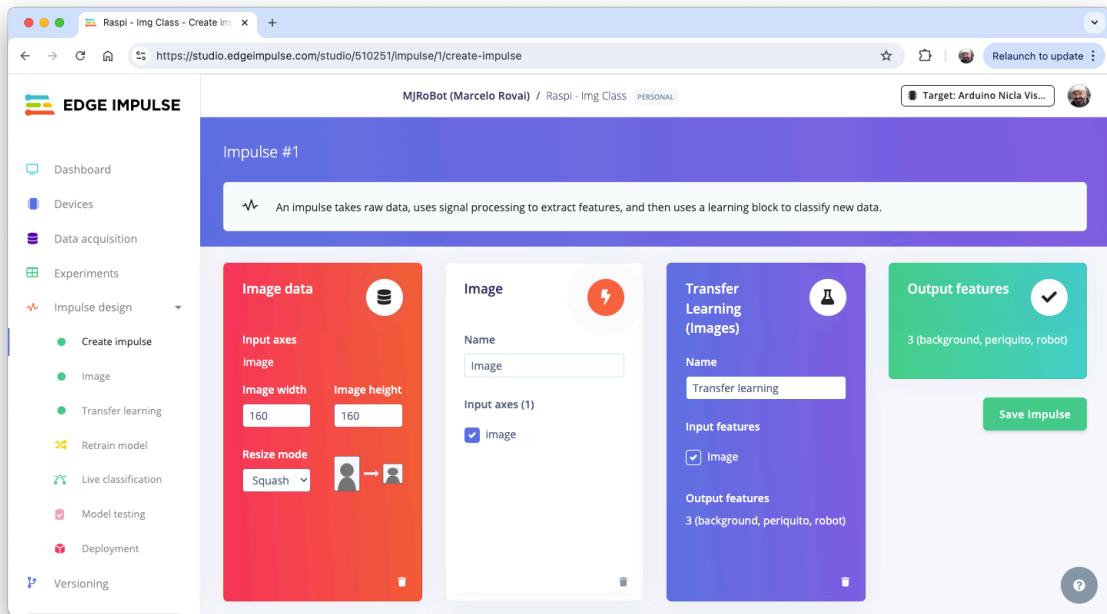


Image Pre-Processing

All the input QVGA/RGB565 images will be converted to 76,800 features (160x160x3).

DSP result

Image



Copy 76800 features to clipboard

Processed features

Copy to clipboard

0.1333, 0.1020, 0.1098, 0.1373, 0.0980, 0.1098, 0.1608, 0.1020, 0.125...

On-device performance



PROCESSING TIME

1 ms.



PEAK RAM USAGE

4 KB



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 160x160 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)

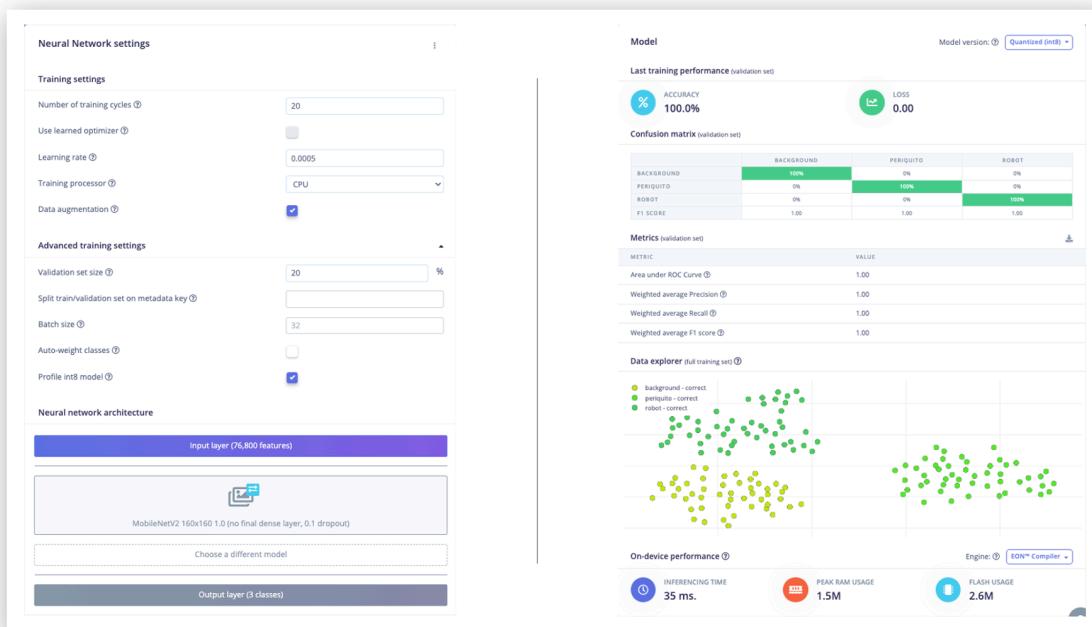
# Vary the brightness of the image
image = tf.image.random_brightness(image, max_delta=0.2)

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 35ms 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 160x160 or 92x92
 - 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	
Image training labels	NPY file	150 windows	
Image testing data	NPY file	36 windows	
Image testing labels	NPY file	36 windows	
Transfer learning model	TensorFlow Lite (float32)	9 MB	
Transfer learning model	TensorFlow Lite (int8 quantized)	3 MB	
Transfer learning model	Model evaluation metrics (JSON file)	5 KB	
Transfer learning model	TensorFlow SavedModel	8 MB	
Transfer learning model	Keras h5 model	8 MB	

Let's also download the float32 version for comparasion

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_runtime.interpreter as tflite
```

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 = 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 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 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:

```
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]

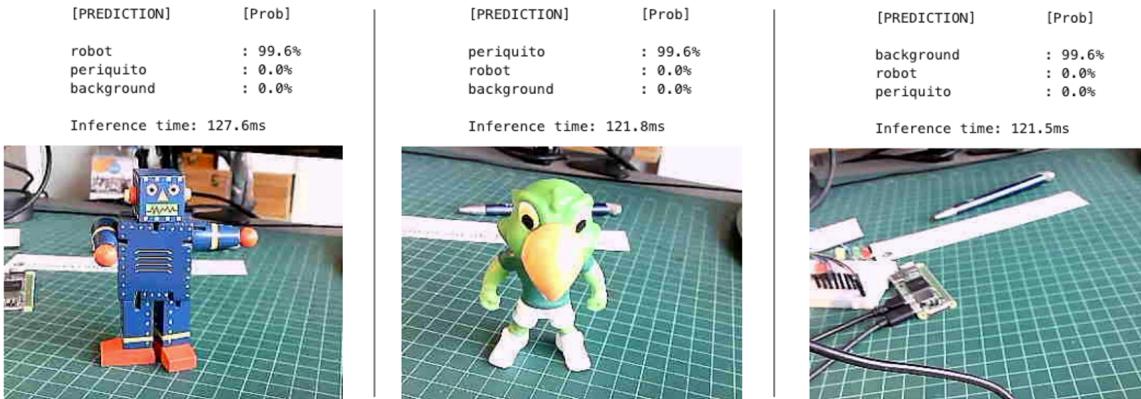
# Handle output based on type
output_dtype = output_details[0]['dtype']
if output_dtype in [np.int8, np.uint8]:
    # Dequantize the output
    scale, zero_point = output_details[0]['quantization']
    predictions = (predictions.astype(np.float32) - zero_point) * scale

if apply_softmax:
    # Apply softmax
    exp_preds = np.exp(predictions - np.max(predictions))
    probabilities = exp_preds / np.sum(exp_preds)
else:
    probabilities = predictions

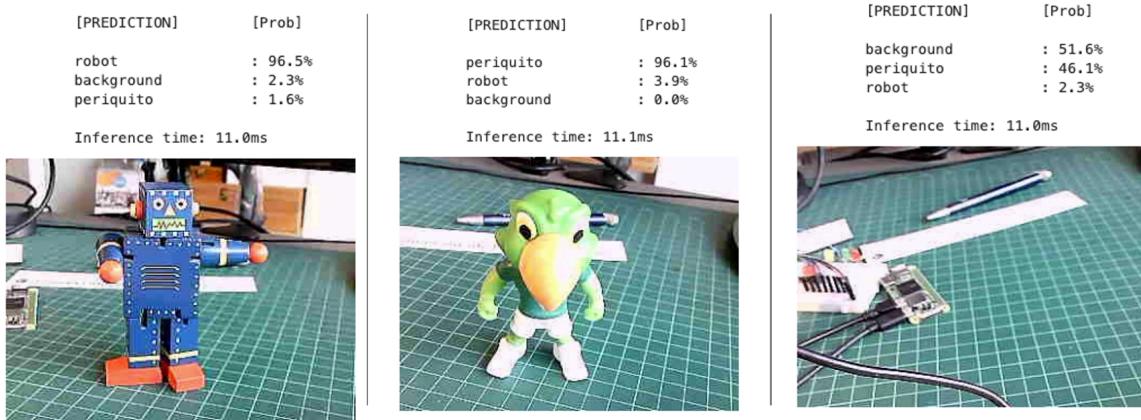
print("\n\t[PREDICTION] [Prob]\n")
for i in range(top_k_results):
    print("\t{:20}: {:.1f}%".format(
        labels[top_k_indices[i]],
        probabilities[top_k_indices[i]] * 100))
print ("\n\tInference time: {:.1f}ms".format(inference_time))

```

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 tflite_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)

```

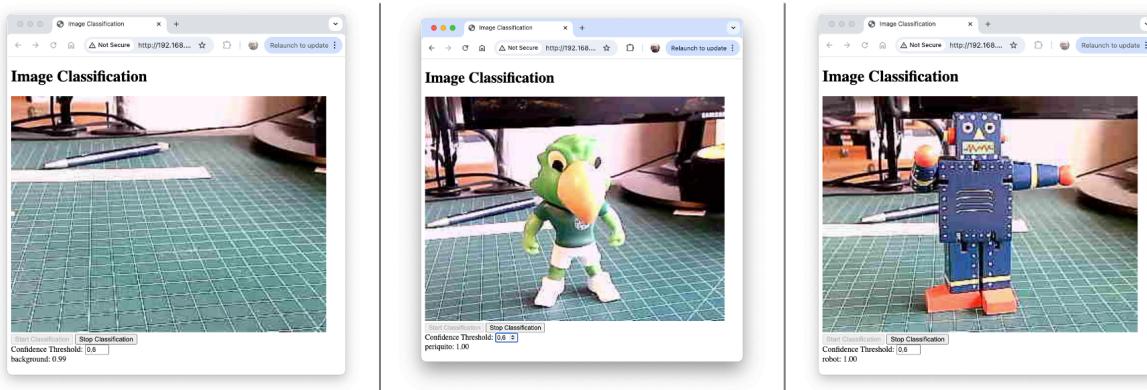
On the terminal, run:

```
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.

Conclusion:

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

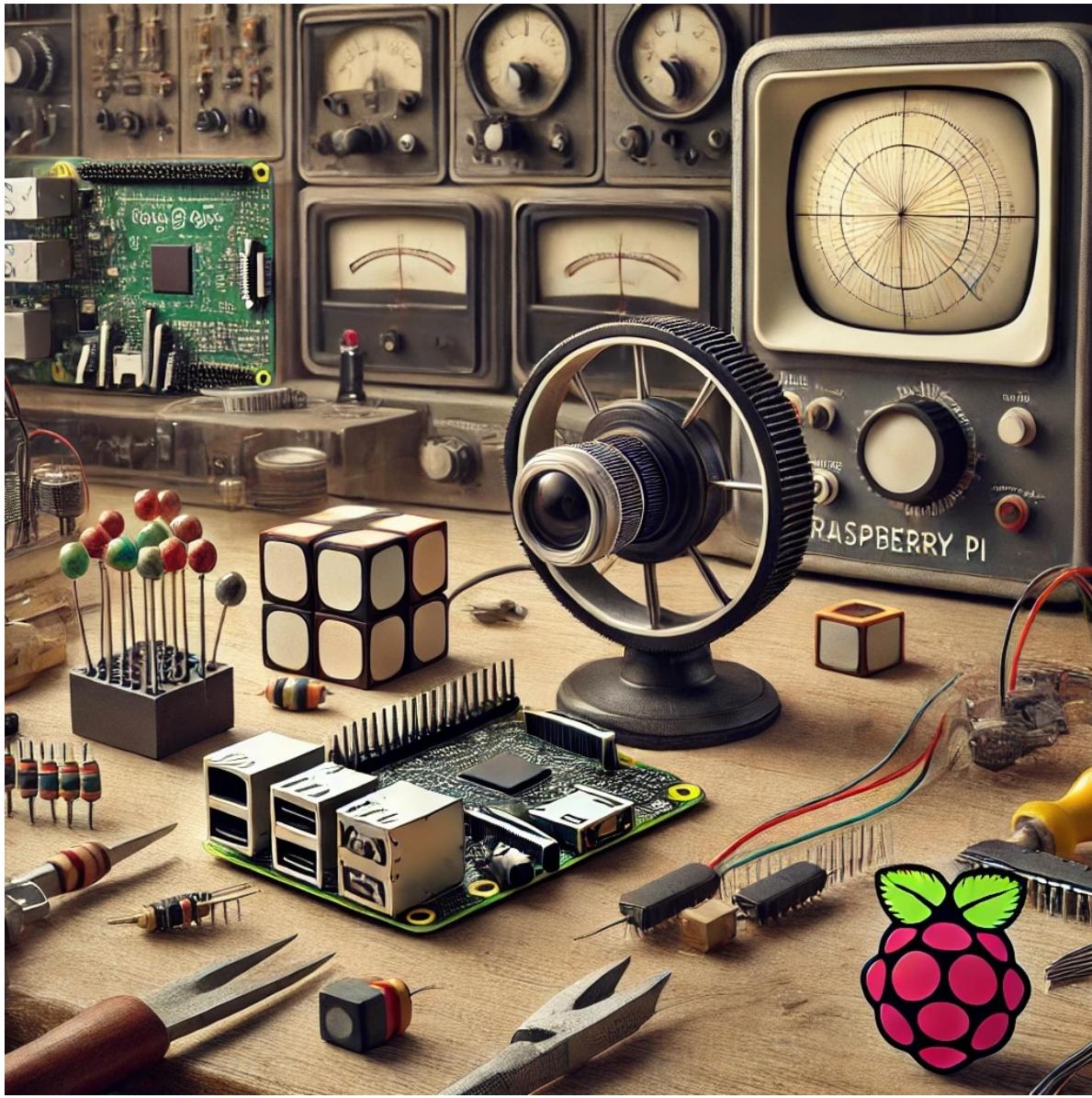


Figure 3: *DALL·E prompt - A cover image for an ‘Object Detection’ chapter in a Raspberry Pi tutorial, designed in the same vintage 1950s electronics lab style as previous covers. The scene should prominently feature wheels and cubes, similar to those provided by the user, placed on a workbench in the 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.*

Introduction

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

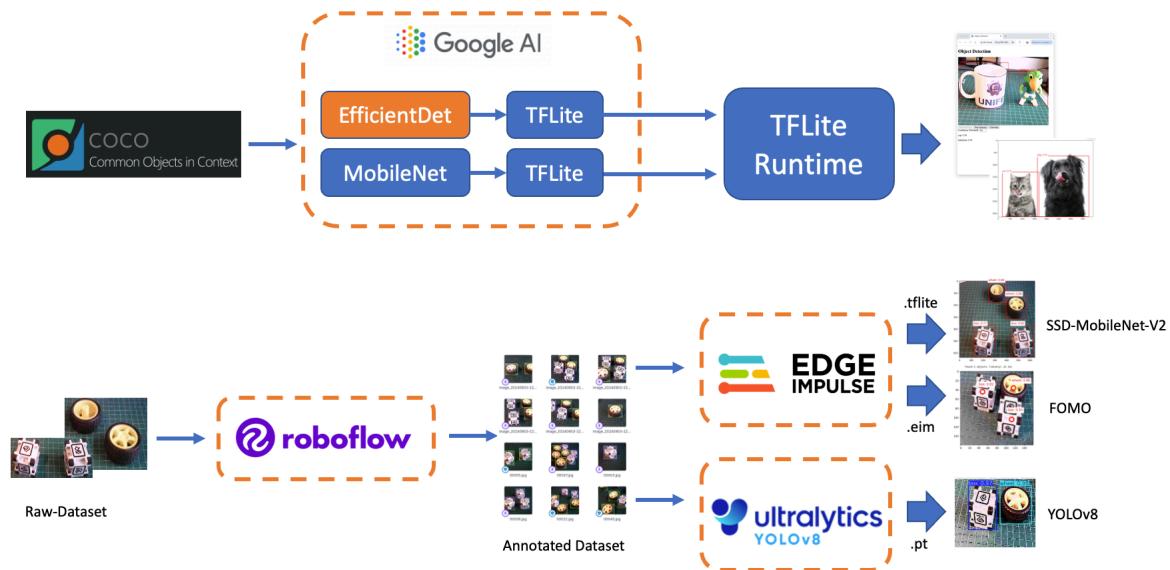
As we put our hands into object detection, we'll build upon the concepts and techniques we explored in image classification. We'll examine popular object detection architectures designed for efficiency, such as:

- 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
- Ultralitics



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

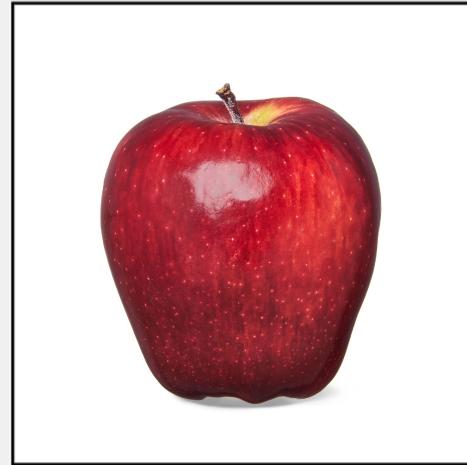
Image Classification:

- 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

Image Classification



Output: "Apple"

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.

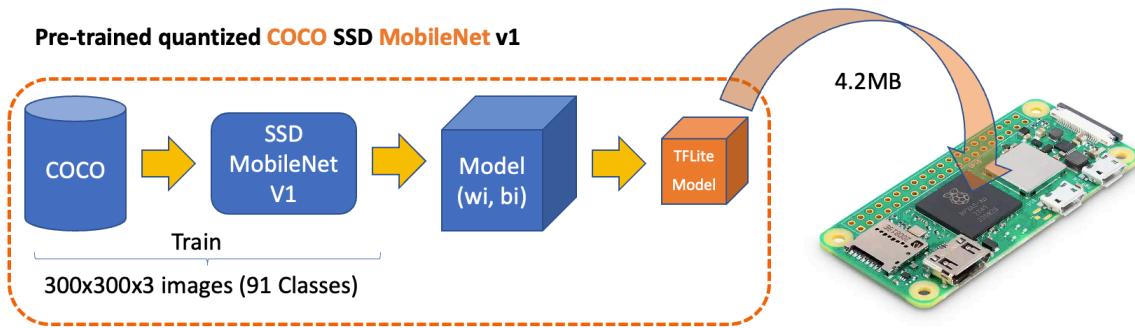
You can test some common models online by visiting [Object Detection - MediaPipe Studio](#)

On [Kaggle](#), we can find the most common pre-trained tflite models to use with the Raspi, [ssd_mobilenet_v1](#), and [efficiendet](#). 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 300x300 SSD-Mobilenet V1 model and compare it with the 320x320 EfficientDet-lite0, also trained using the COCO 2017 dataset. Both models were converted to a TensorFlow Lite format (4.2MB for the SSD Mobilenet and 4.6MB 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:

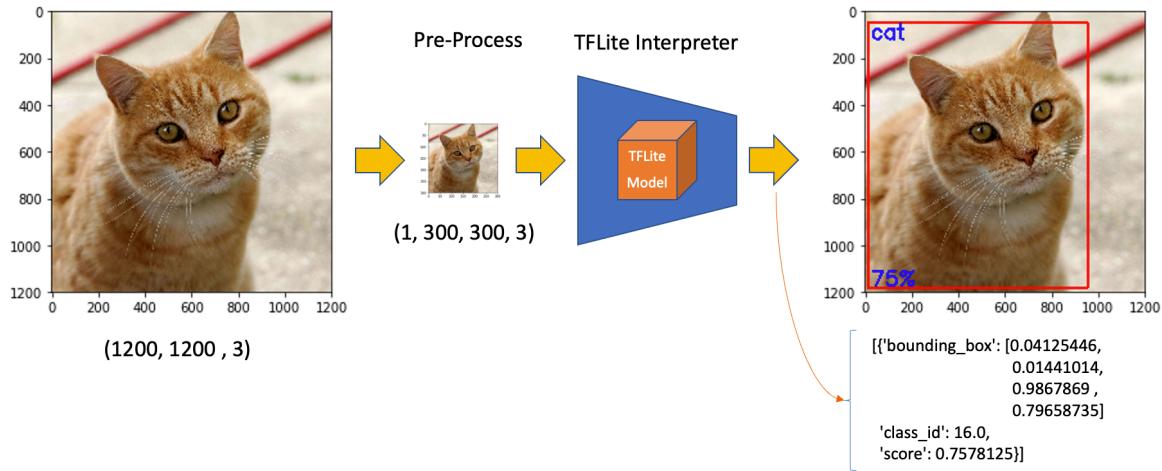
```
model_path = "./models/ssd-mobilenet-v1-tflite-default-v1.tflite"
interpreter = tflite.Interpreter(model_path=model_path)
interpreter.allocate_tensors()
```

Get input and output tensors.

```
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()
```

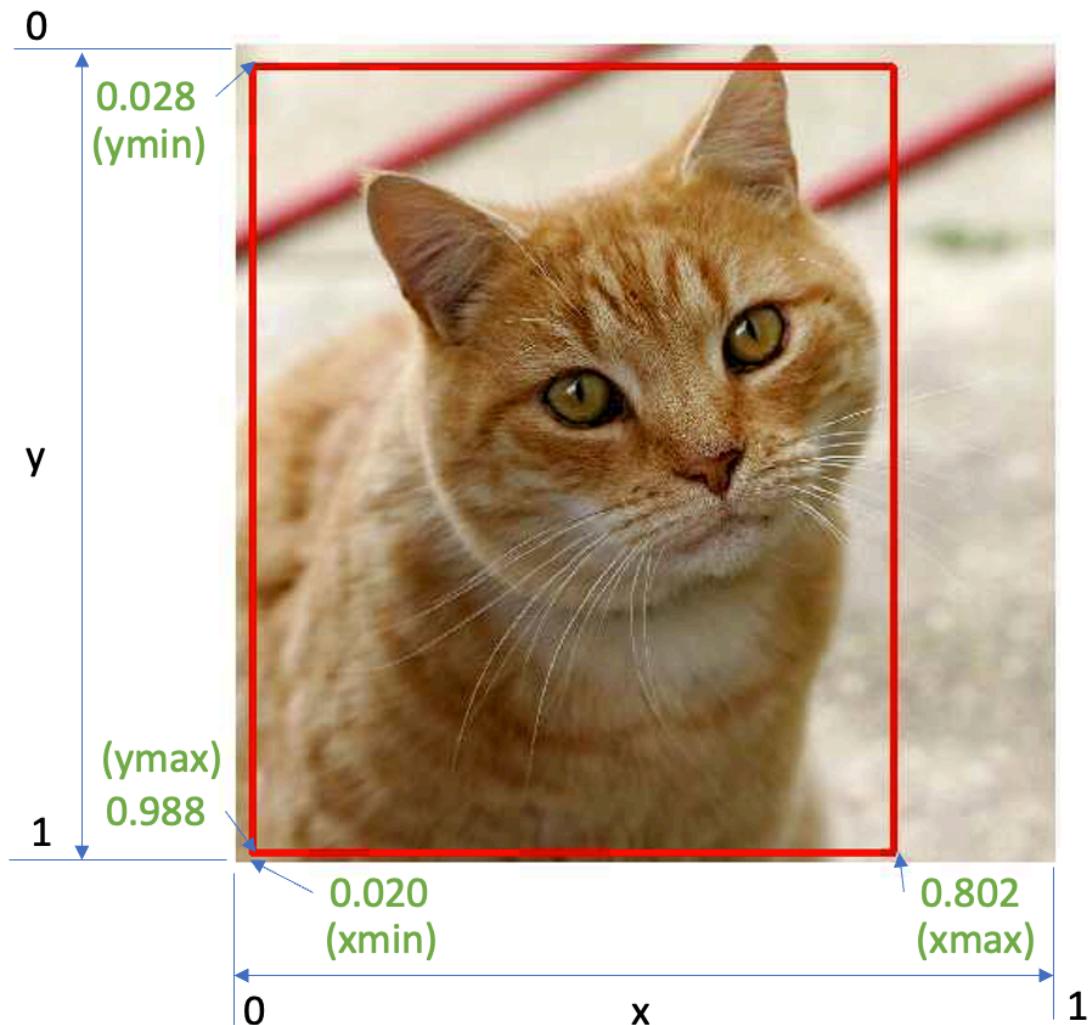
Input details will inform us how the model should be fed with an image. The shape of (1, 300, 300, 3) with a dtype of `uint8` tells us that a non-normalized (pixel value range from 0 to 255) image with dimensions (300x300x3) should be input one by one (Batch Dimension: 1).

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:



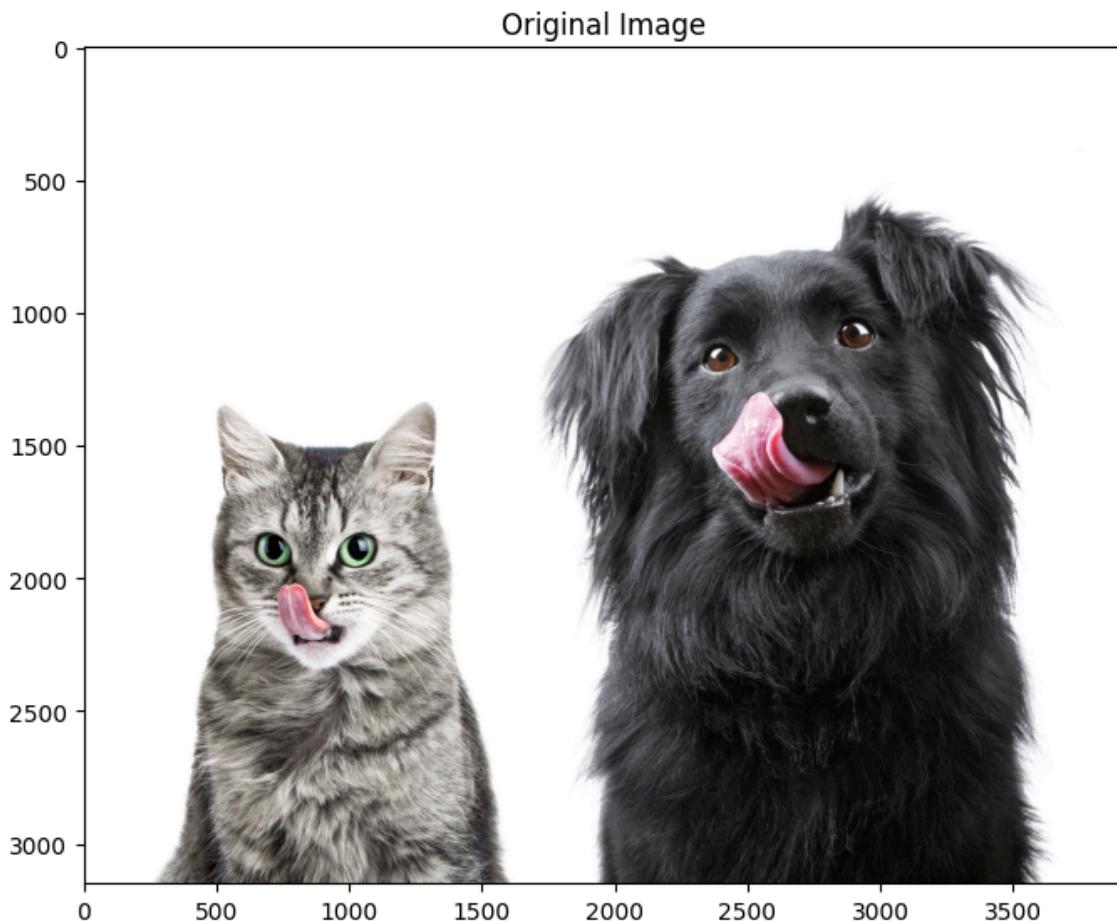
Next, we should find what class ID equal to 16 means. Opening the file `coco_labels.txt`, as a list, each element has an associated index, and inspecting index 16, we get, as expected, `cat`. The probability is the value returning from the score.

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 800ms, 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
```

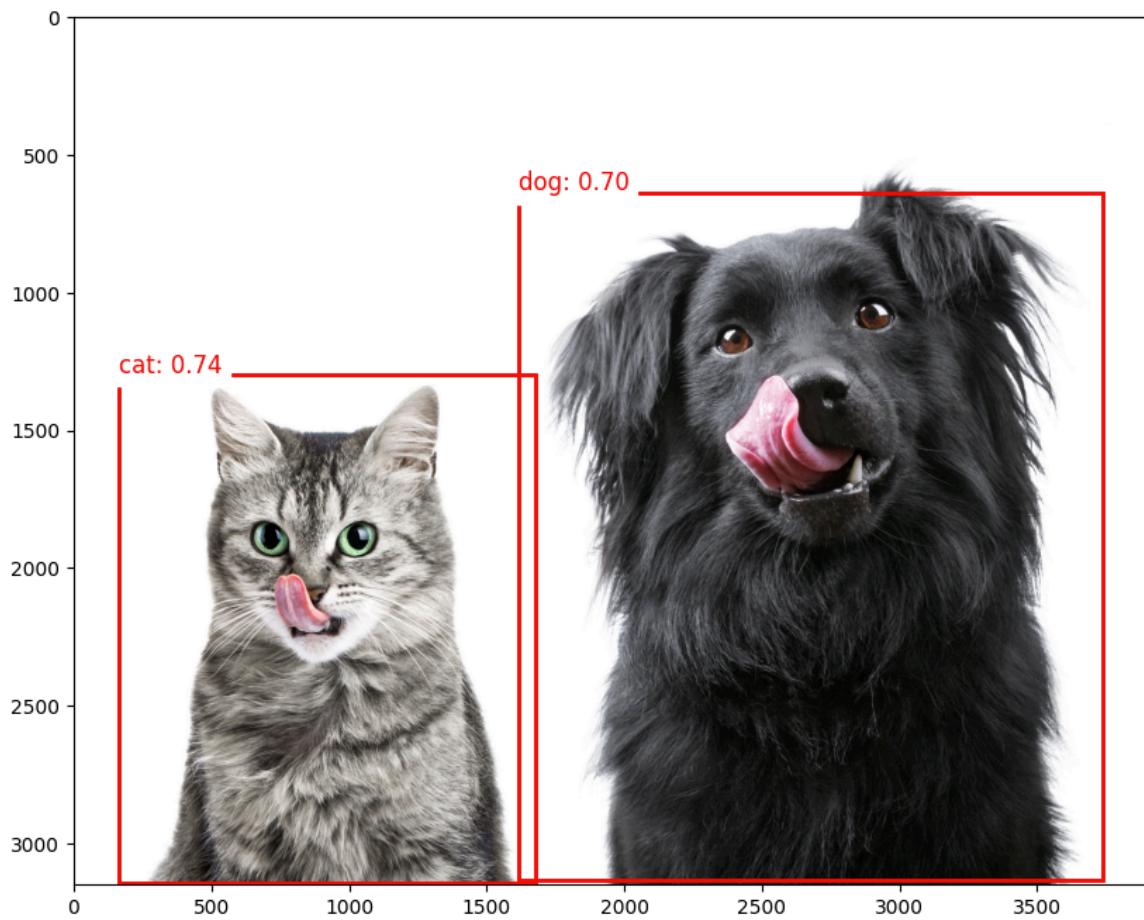
And we can also visualize the results:

```
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

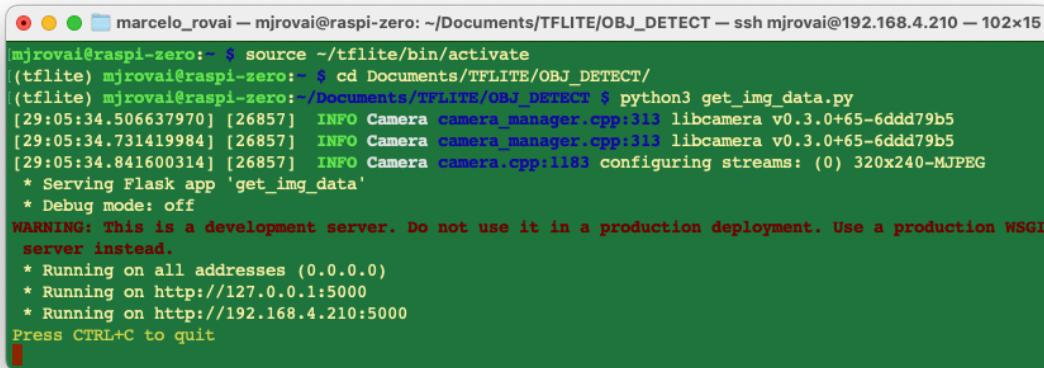
Once we have defined our Machine Learning project goal, the next and most crucial step is collecting the dataset. We can use a phone, the Raspi, or a mix to create the raw dataset (with no labels). Let's use the simple web app on our Raspberry Pi to view the QVGA (320 x 240) captured images in a browser.

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:

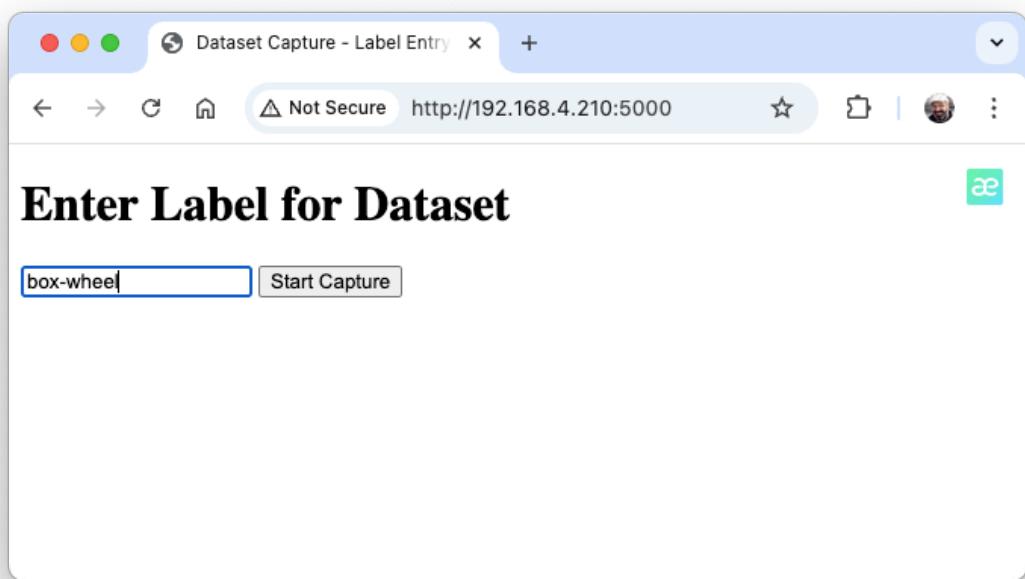
- On the Raspberry Pi itself (if you have a GUI): Open a web browser and go to `http://localhost:5000`
- From another device on the same network: Open a web browser and go to `http://<raspberry_pi_ip>:5000` (Replace `<raspberry_pi_ip>` with your Raspberry Pi's IP address). For example: `http://192.168.4.210:5000/`



```
mjrovai@raspi-zero:~$ source ~/tfelite/bin/activate
(melite) mjrovai@raspi-zero:~$ cd Documents/TFLITE/OBJ_DETECT/
(melite) mjrovai@raspi-zero:~/Documents/TFLITE/OBJ_DETECT$ python3 get_img_data.py
[29:05:34.506637970] [26857] INFO Camera camera_manager.cpp:313 libcamera v0.3.0+65-6ddd79b5
[29:05:34.731419984] [26857] INFO Camera camera_manager.cpp:313 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
```

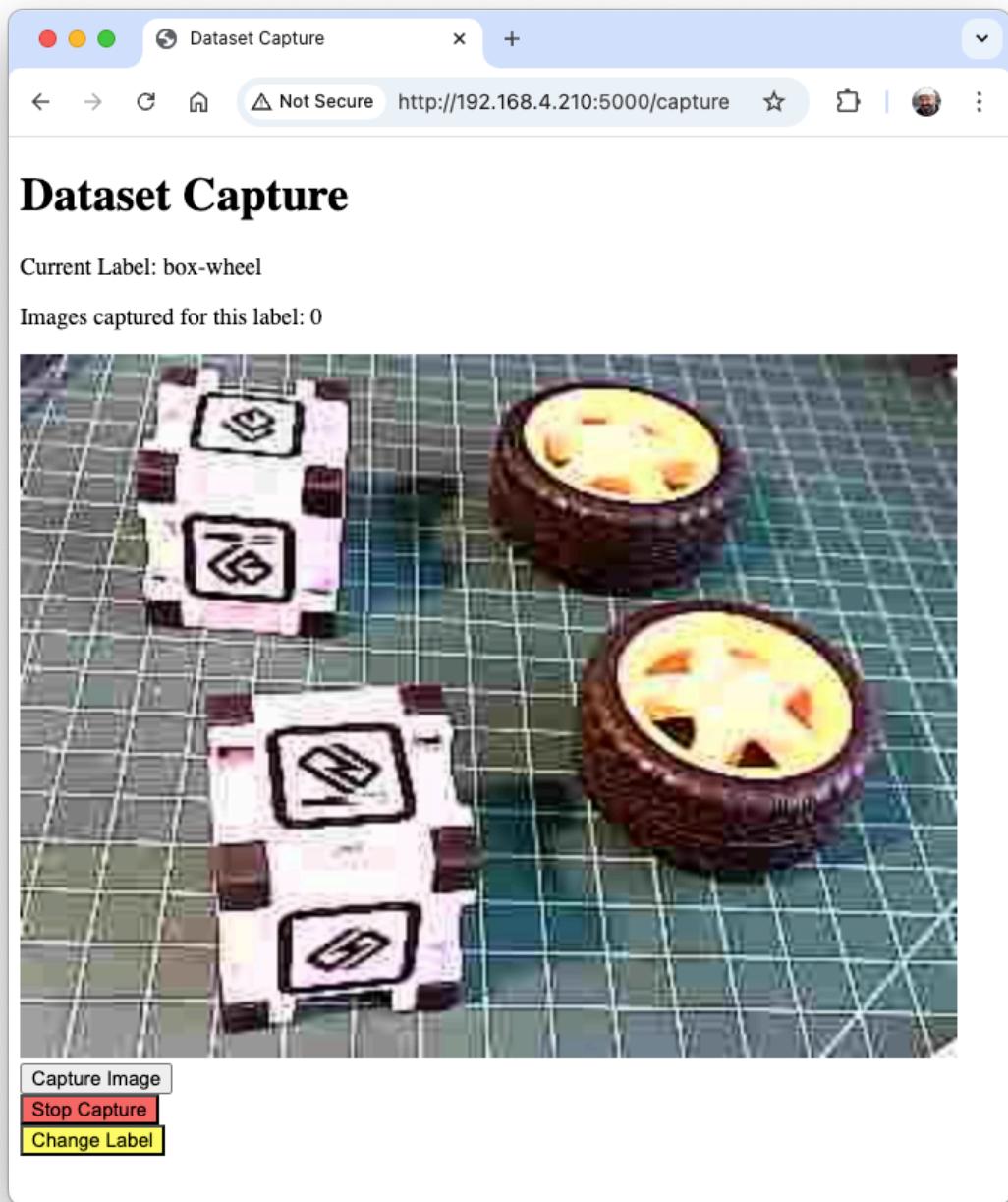
The Python script creates a web-based interface for capturing and organizing image datasets using a Raspberry Pi and its camera. It's handy for machine learning projects that require labeled image data or not, as in our case here.

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:

```
(tflite) mjrovai@raspi-zero:~/Documents/TFLITE/OBJ_DETECT/dataset $ ls
Untitled.ipynb  dataset  get_img_data.py  images  models
(tflite) mjrovai@raspi-zero:~/Documents/TFLITE/OBJ_DETECT $ cd dataset
(tflite) mjrovai@raspi-zero:~/Documents/TFLITE/OBJ_DETECT/dataset $ ls
box-wheel
(tflite) mjrovai@raspi-zero:~/Documents/TFLITE/OBJ_DETECT/dataset $ ls box-wheel
image_20240903-224450.jpg  image_20240903-224513.jpg  image_20240903-224530.jpg
image_20240903-224452.jpg  image_20240903-224516.jpg  image_20240903-224533.jpg
image_20240903-224458.jpg  image_20240903-224520.jpg  image_20240903-224535.jpg
image_20240903-224504.jpg  image_20240903-224524.jpg
(tflite) mjrovai@raspi-zero:~/Documents/TFLITE/OBJ_DETECT/dataset $
```

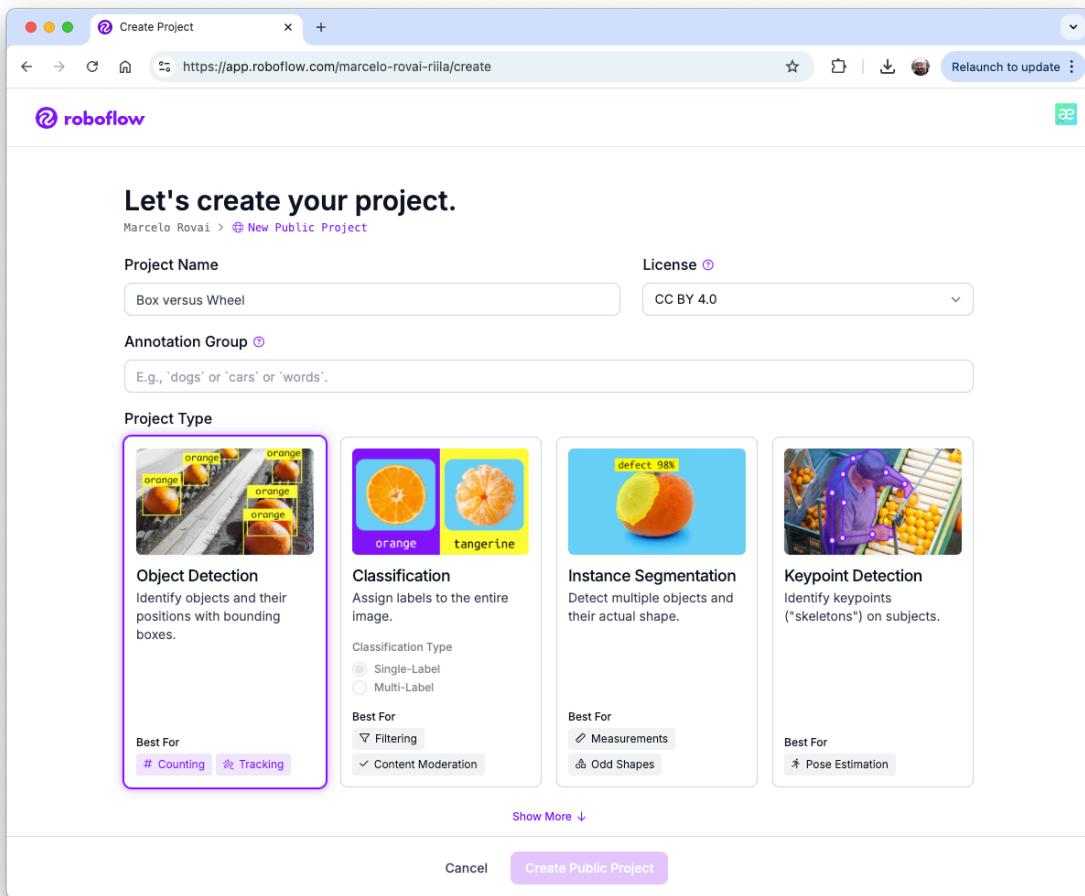
Get around 60 images. Try to capture different angles, backgrounds, and light conditions. Filezilla can transfer the created raw dataset to your main computer.

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.

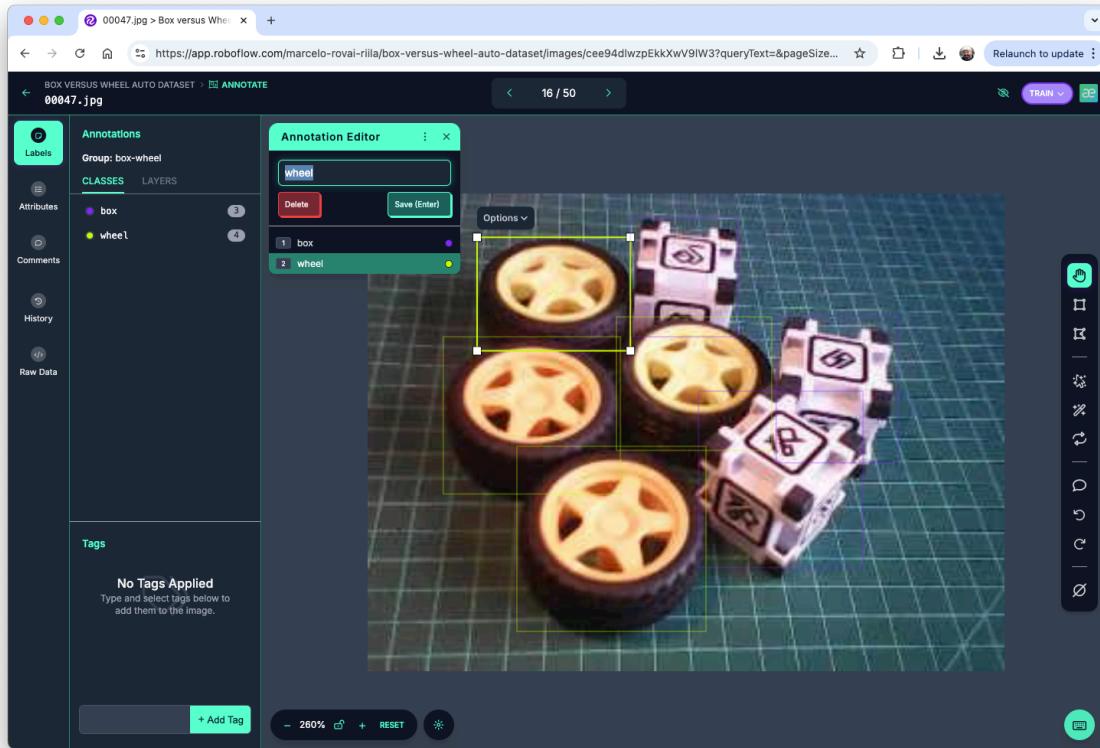
We should upload the raw dataset to [Roboflow](#). Create a free account there and start a new project, for example, ("box-versus-wheel").



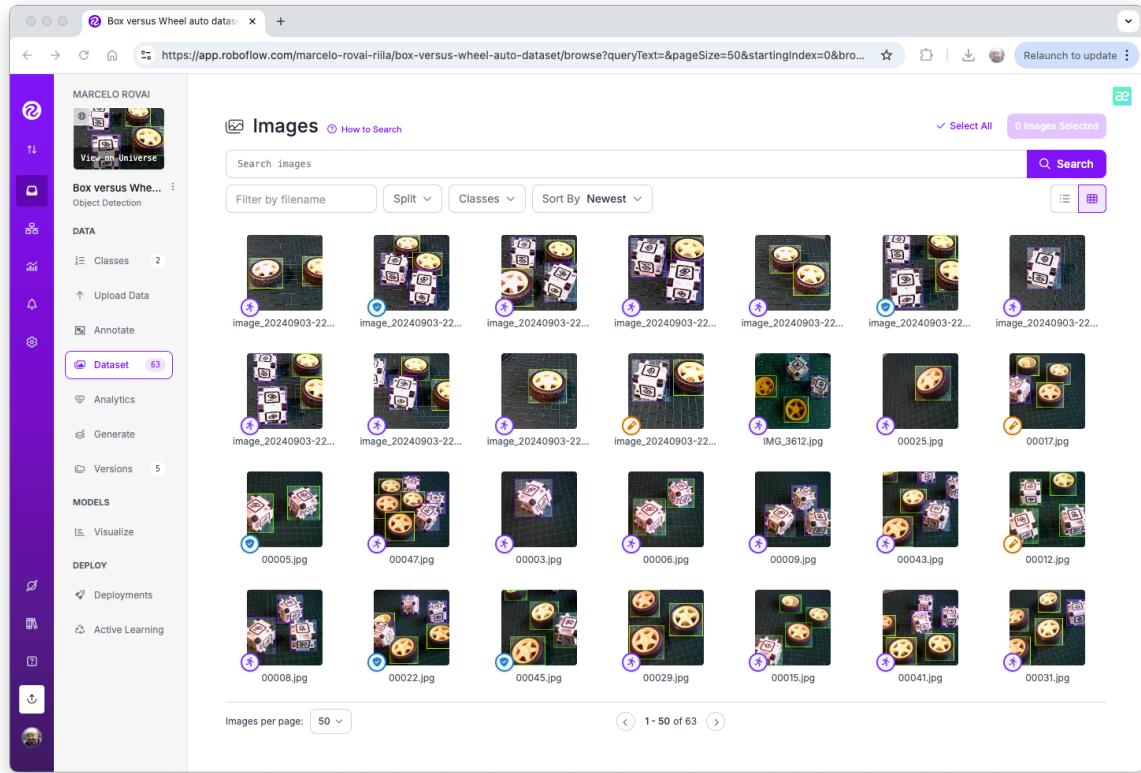
We will not enter in deep details about the Roboflow process once many tutorials are available.

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.



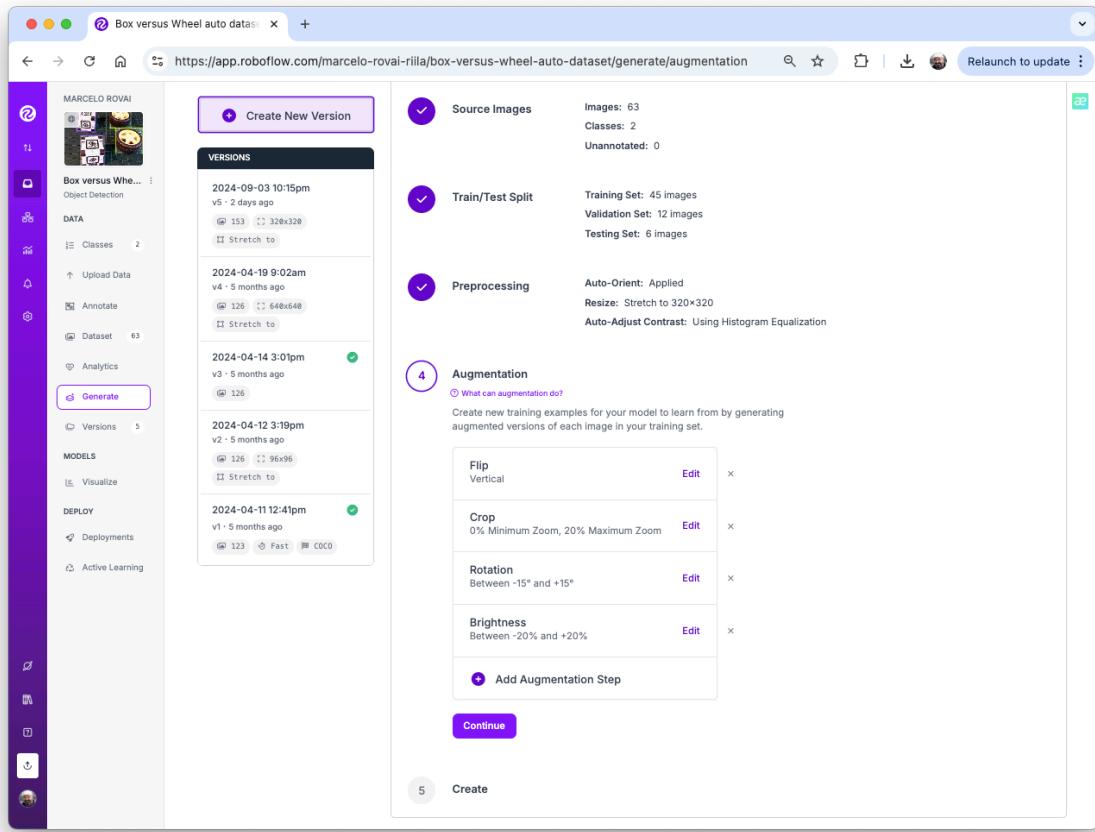
Once all images are annotated, you should split them into training, validation, and testing.



Data Pre-Processing

The last step with the dataset is preprocessing to generate a final version for training. Let's resize all images to 320x320 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°), crop, and vary the brightness and exposure.

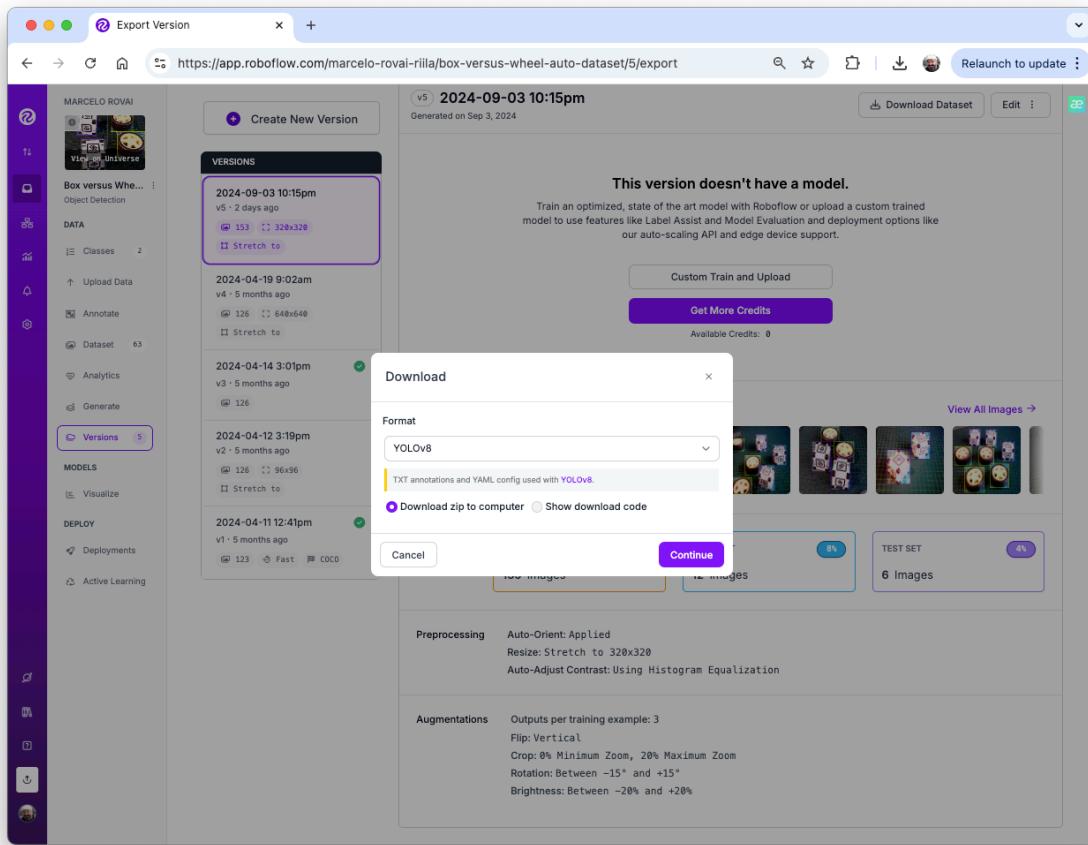


At the end of the process, we will have 153 images.

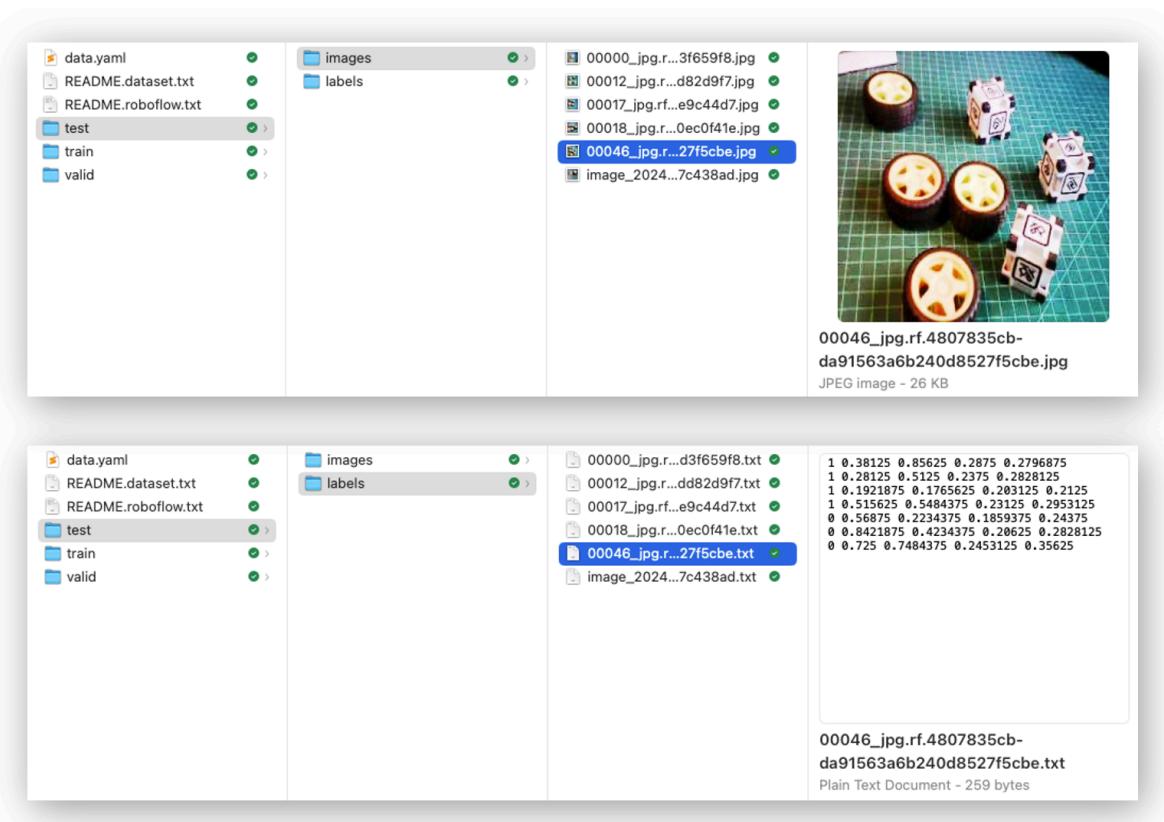
The screenshot shows the Roboflow web interface for a dataset titled "Box versus Wheel auto dataset".

- Left Sidebar:** Contains project navigation, including "MARCELO ROVALI", "Box versus Whe...", "Object Detection", "DATA", "Classes 2", "Upload Data", "Annotate", "Dataset 63", "Analytics", "Generate", "Versions 5", "Models", "Visualize", "Deploy", "Deployments", and "Active Learning".
- Central Dashboard:**
 - Version History:** Shows four versions:
 - v5 - 2 days ago: 153 images, 320x320 pixels, Stretch to
 - v4 - 5 months ago: 126 images, 640x640 pixels, Stretch to
 - v3 - 5 months ago: 126 images
 - v1 - 5 months ago: 123 images, Fast, COCO
 - Current Version (v5):** Generated on Sep 3, 2024 at 2024-09-03 10:15pm.
 - Model Status:** "This version doesn't have a model." with a "Custom Train and Upload" button.
 - Image Preview:** Shows 153 total images with a "View All Images" link.
 - Dataset Split:** TRAIN SET (135 Images), VALID SET (12 Images), TEST SET (6 Images).
 - Preprocessing:** Auto-Orient: Applied, Resize: Stretch to 320x320, Auto-Adjust Contrast: Using Histogram Equalization.
 - Augmentations:** Outputs per training example: 3, with details: Flip: Vertical, Crop: 0% Minimum Zoom, 20% Maximum Zoom, Rotation: Between -15° and +15°, Brightness: Between -20% and +20%.

Now, you should export the annotated dataset in a format that Edge Impulse, Ultralitics, 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](#), or here:

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.

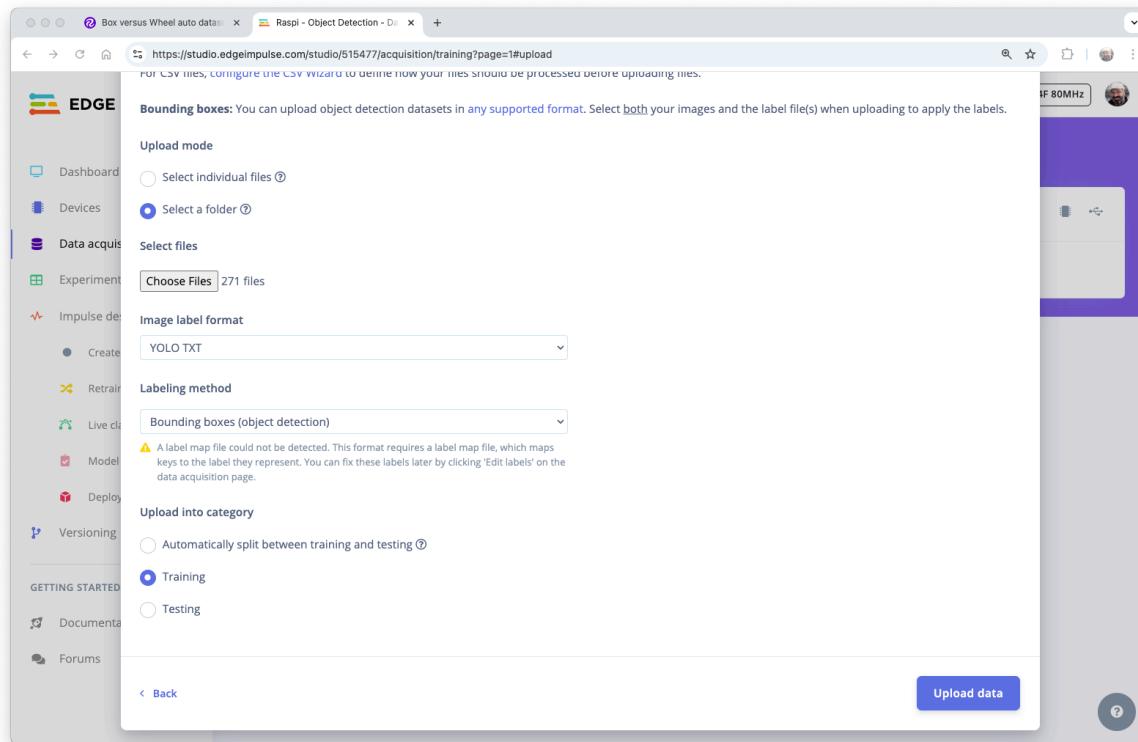
Here, you can clone the project developed for this hands-on lab: [Raspi - Object Detection](#).

On the Project Dashboard tab, go down and on **Project info**, and for Labeling method select **Bounding boxes (object detection)**

Uploading the annotated data

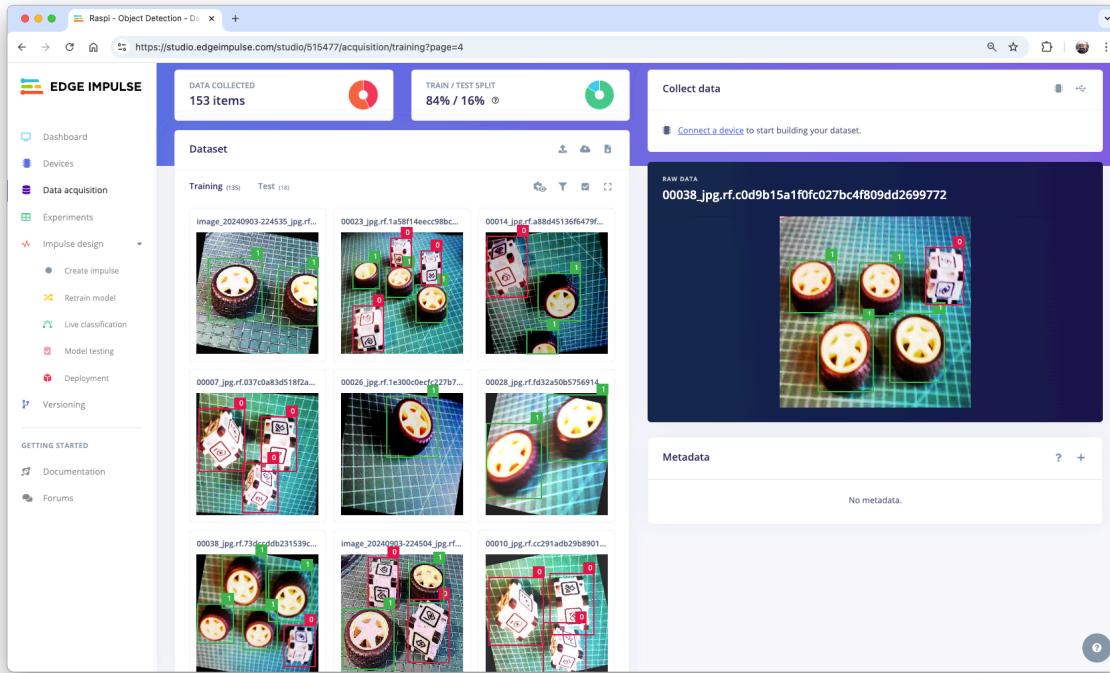
On Studio, go to the **Data acquisition** tab, and on the **UPLOAD DATA** section, upload from your computer the raw dataset.

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%).

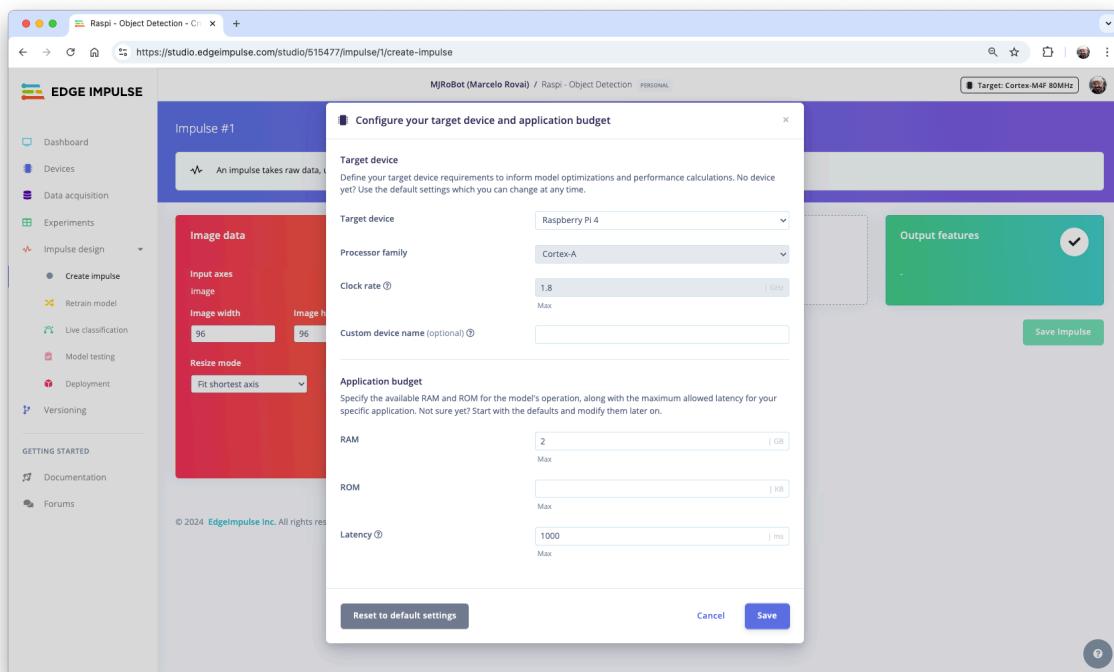
Note that labels will be stored at the labels files 0 and 1 , which are equivalent to **box** and **wheel**.



The Impulse Design

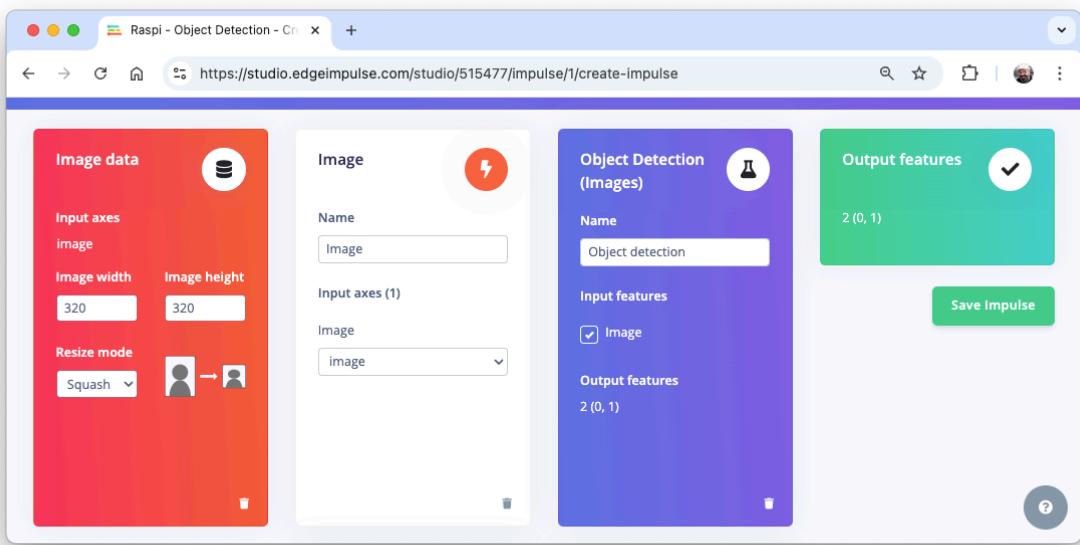
The first thing to define when we enter the **Create impulse** step is to describe the target device for deployment. A pop-up window will appear. We will select Raspberry 4, an intermediary device between the Raspi-Zero and the Raspi-5.

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.



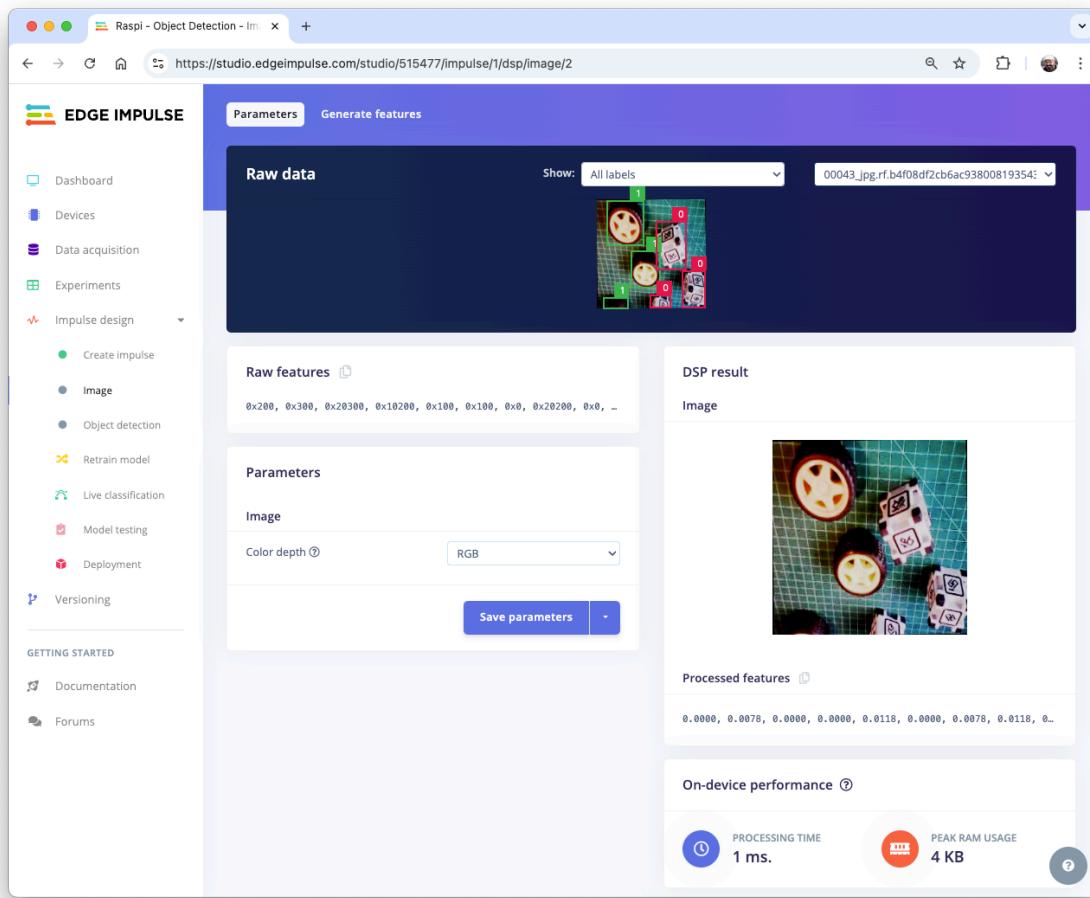
In this phase, you should define how to:

- **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.”

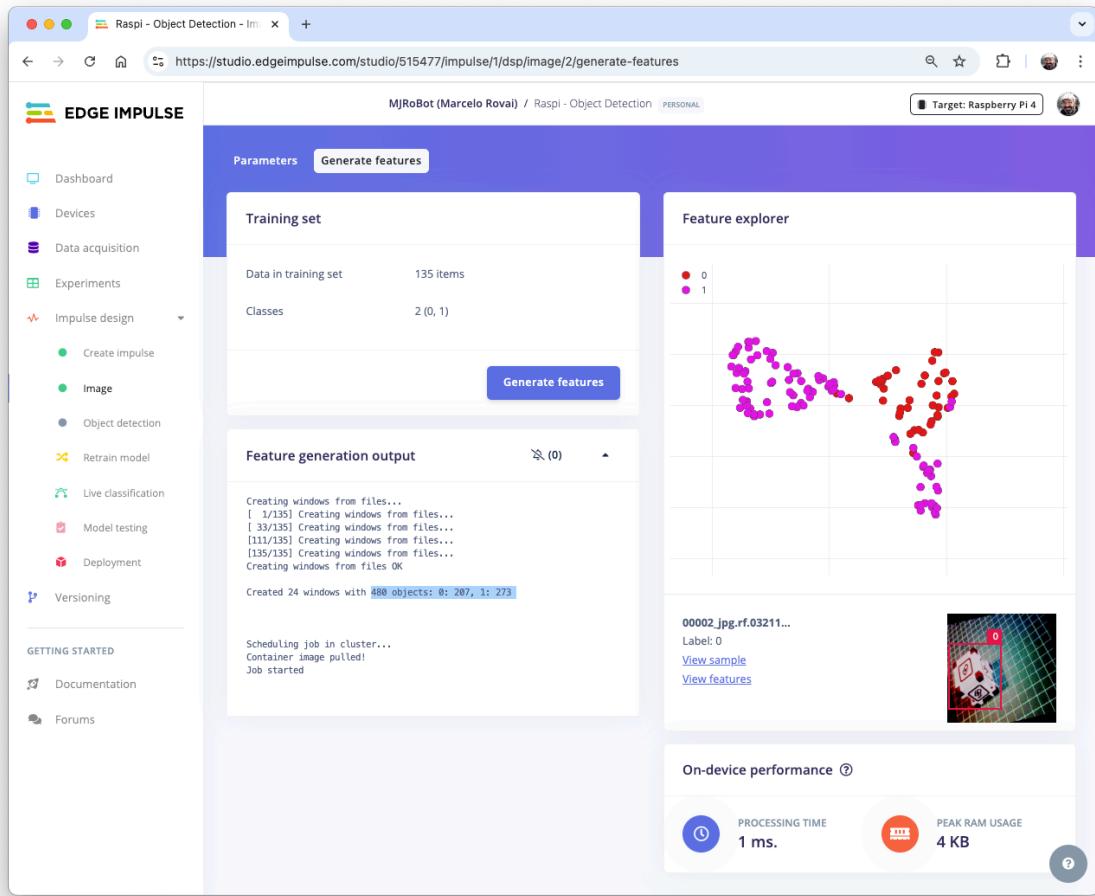


Preprocessing all dataset

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 3.7MB in size. It supports an RGB input at 320x320px.

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.

The screenshot shows two main sections of the Edge Impulse Dashboard:

Neural Network settings (Left):

- Training settings:**
 - Number of training cycles: 25
 - Use learned optimizer:
 - Learning rate: 0.15
 - Training processor: CPU
- Advanced training settings:**
 - Validation set size: 20 %
 - Split train/validation set on metadata key:
 - Batch size: 32
 - Profile int8 model:
- Neural network architecture:**
 - Input layer (307,200 features)
 - MobileNetV2 SSD FPN-Lite 320x320
 - Choose a different model
 - Output layer (2 classes)

Model (Right):

- Last training performance (validation set):**
 - Precision Score: 88.8%
- Metrics (validation set):**

Metric	Value
mAP	0.59
mAP@([IoU=50])	0.94
mAP@([IoU=75])	0.72
mAP@[area=small]	-1.00
mAP@[area=medium]	0.61
mAP@[area=large]	0.58
Recall@[max_detections=1]	0.25
Recall@[max_detections=10]	0.70
Recall@[max_detections=100]	0.70
Recall@[area=small]	-1.00
Recall@[area=medium]	0.70
Recall@[area=large]	0.70
- On-device performance:**
 - Engine: TensorFlow Lite
 - Inferencing Time: 463 ms.
 - Flash Usage: 11.0M

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:

Download block output

TITLE	TYPE	SIZE
Image training data	NPY file	135 windows
Image training labels	JSON file	135 windows
Image testing data	NPY file	18 windows
Image testing labels	JSON file	18 windows
Object detection model	TensorFlow Lite (float32)	11 MB
Object detection model	TensorFlow Lite (int8 quantized)	3 MB
Object detection model	Model evaluation metrics (JSON file)	2 KB
Object detection model	TensorFlow SavedModel	10 MB

<https://studio.edgeimpulse.com/v1/api/515477/learn-data/3/model/tflite-int8>

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._runtime.interpreter as tflite
```

Define the model path and labels:

```
model_path = "./models/ei-raspi-object-detection-SSD-MobileNetv2-320x0320-\\
int8.tflite"
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
```

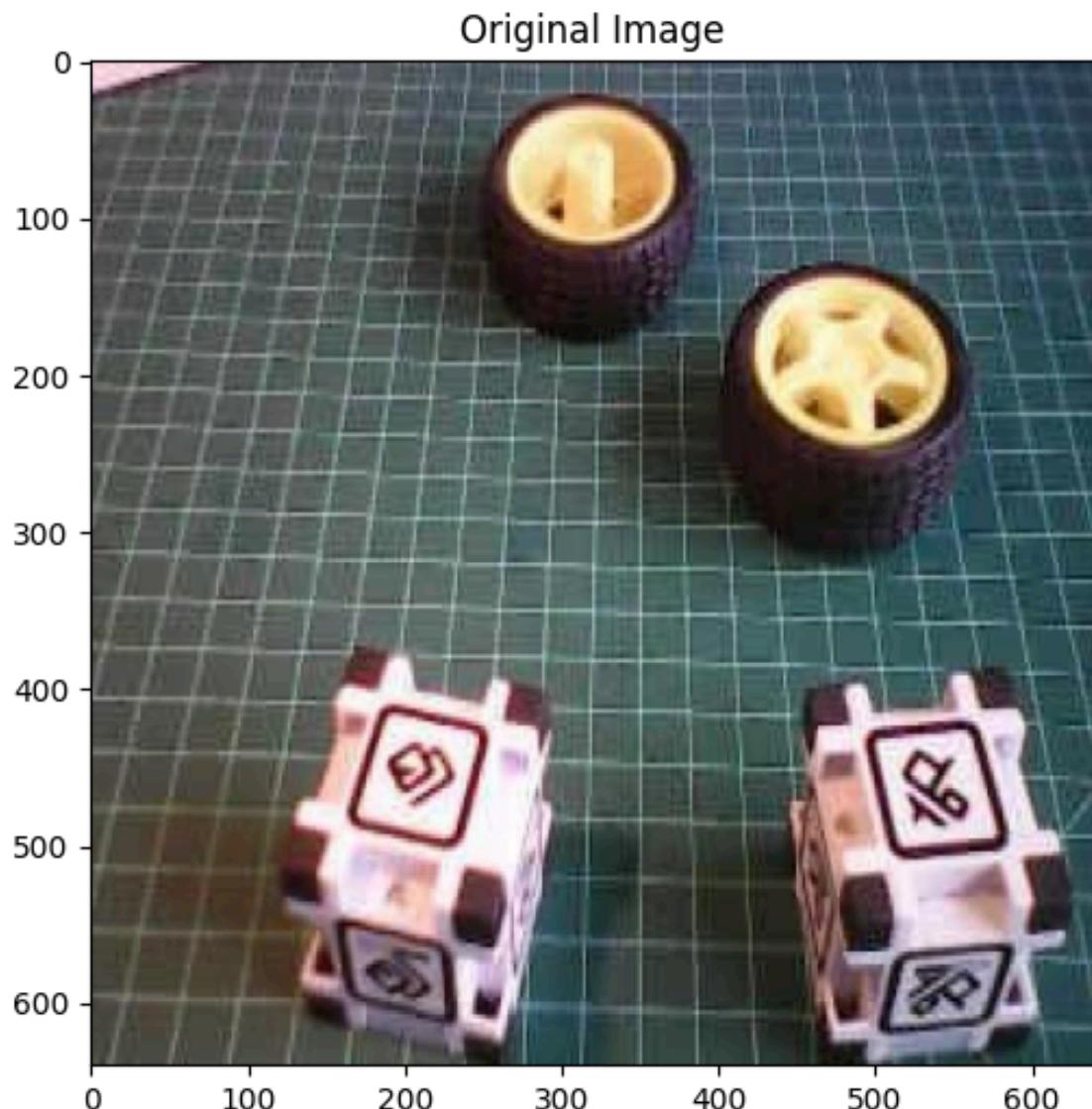
```
numpy.int8
```

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

```

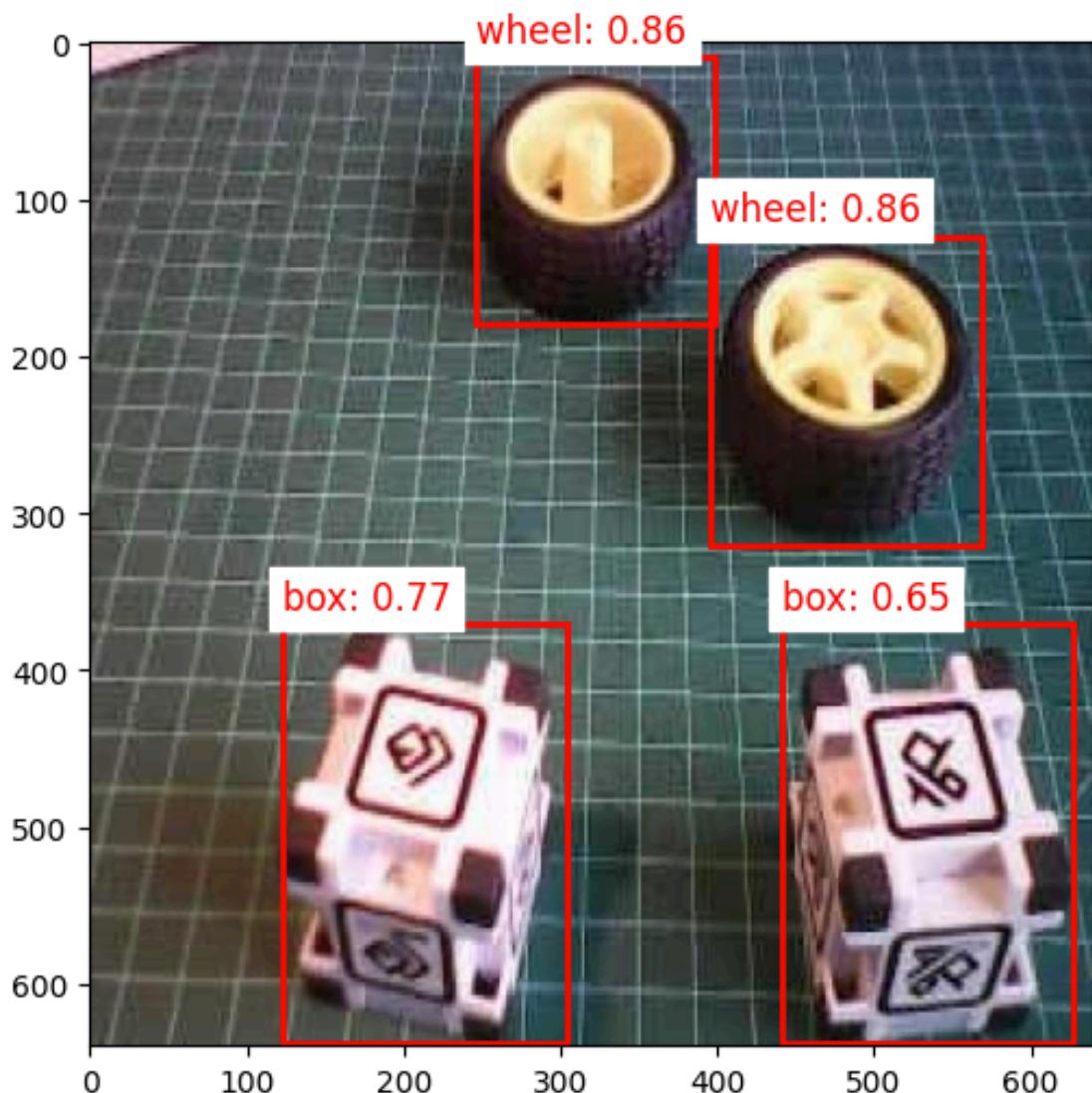
From the results, we can see that 4 objects were detected: two with class ID 0 (**box**) and two with class ID 1 (**wheel**), what is correct!

Let's visualize the result for a threshold of 0.5

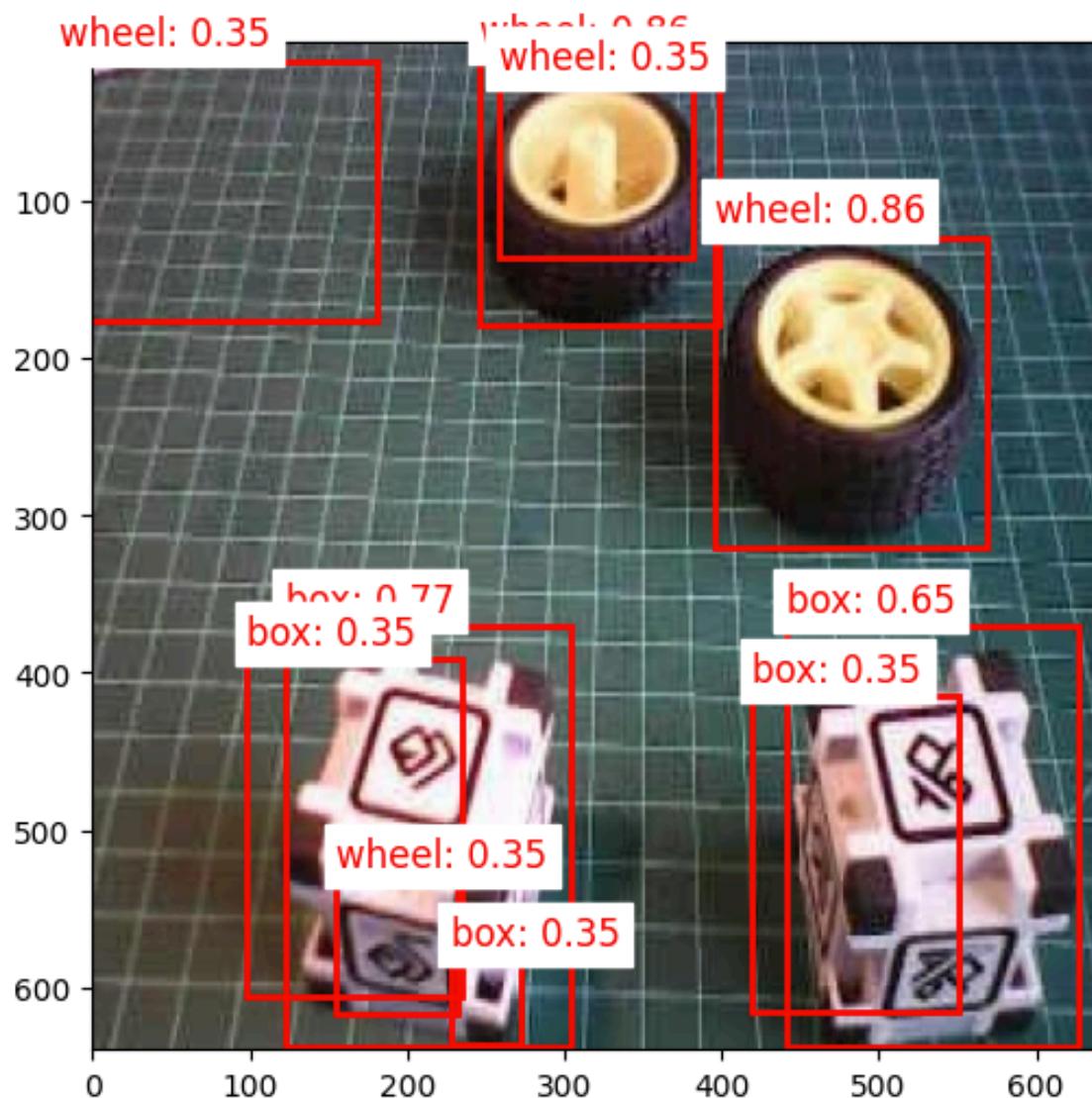
```

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()[:-1:1]

    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)

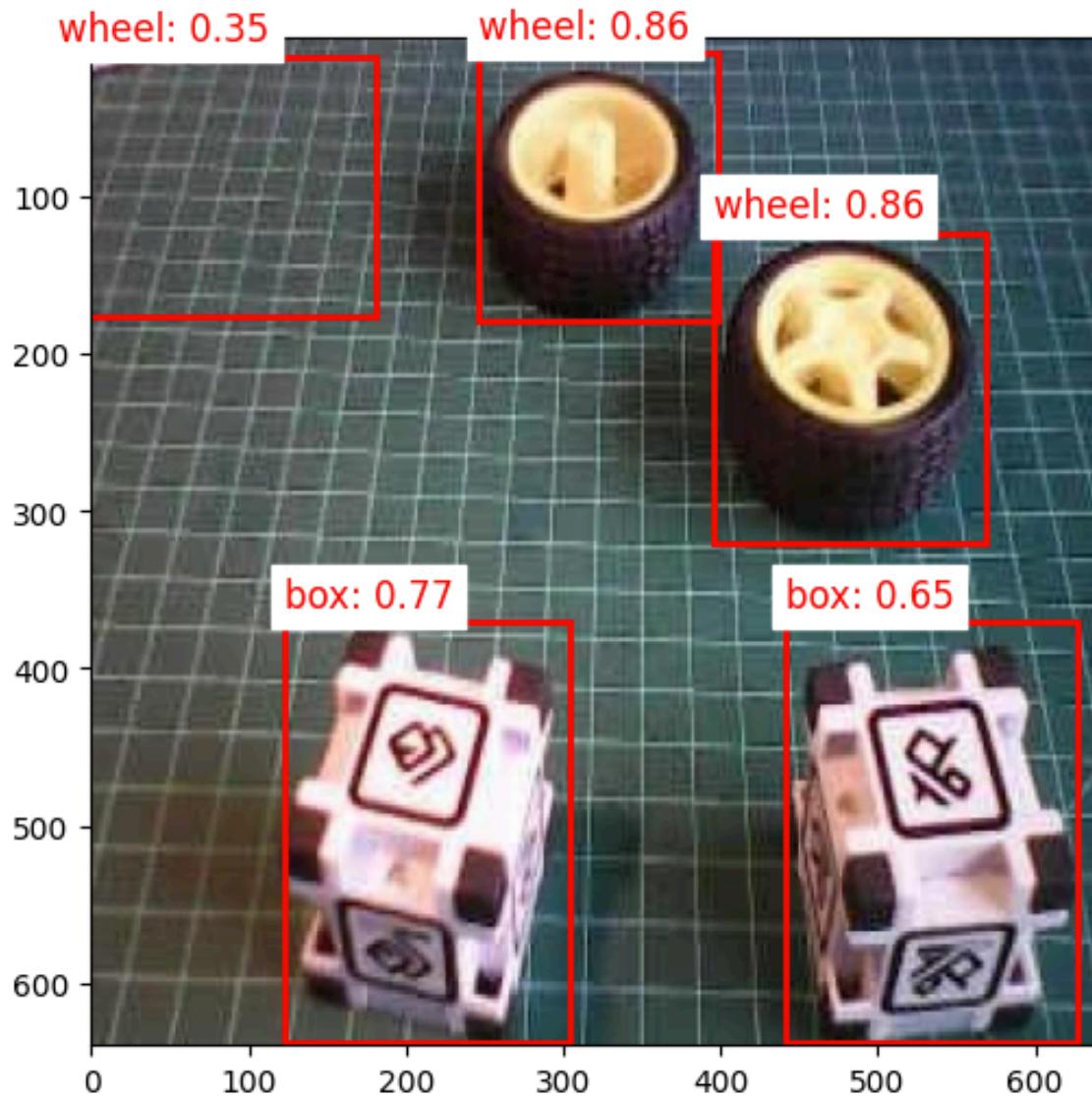
    # Inference on Raspi-Zero
    start_time = time.time()
    interpreter.set_tensor(input_details[0]['index'], input_data)
    interpreter.invoke()
    end_time = time.time()
    inference_time = (end_time - start_time) * 1000 # Convert to ms
    print ("Inference time: {:.1f}ms".format(inference_time))

    # 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

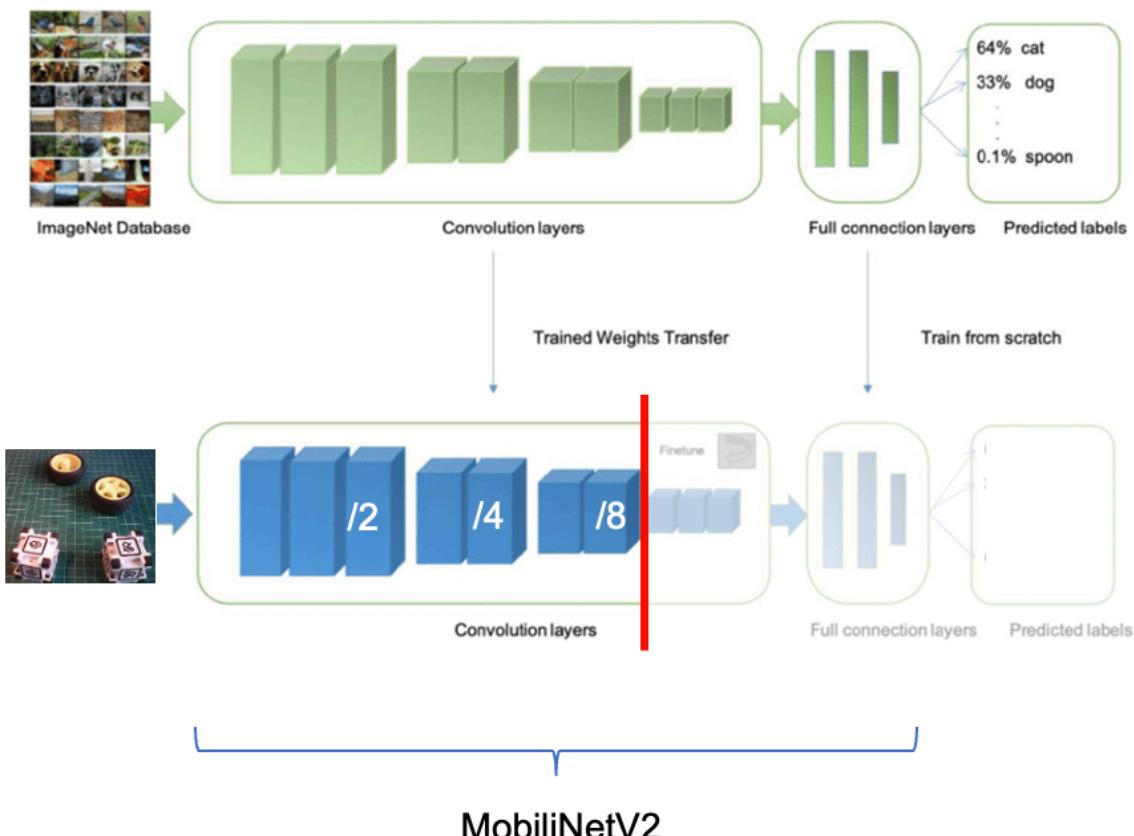
The inference with the SSD MobileNet model worked well, but the latency was significantly high. The inference varied from 0.5 to 1.3 seconds on a Raspi-Zero, which means around or less than 1 FPS (1 frame per second). One alternative to speed up the process is to use FOMO (Faster Objects, More Objects).

This novel machine learning algorithm lets us count multiple objects and find their location in

an image in real-time using up to 30x 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.

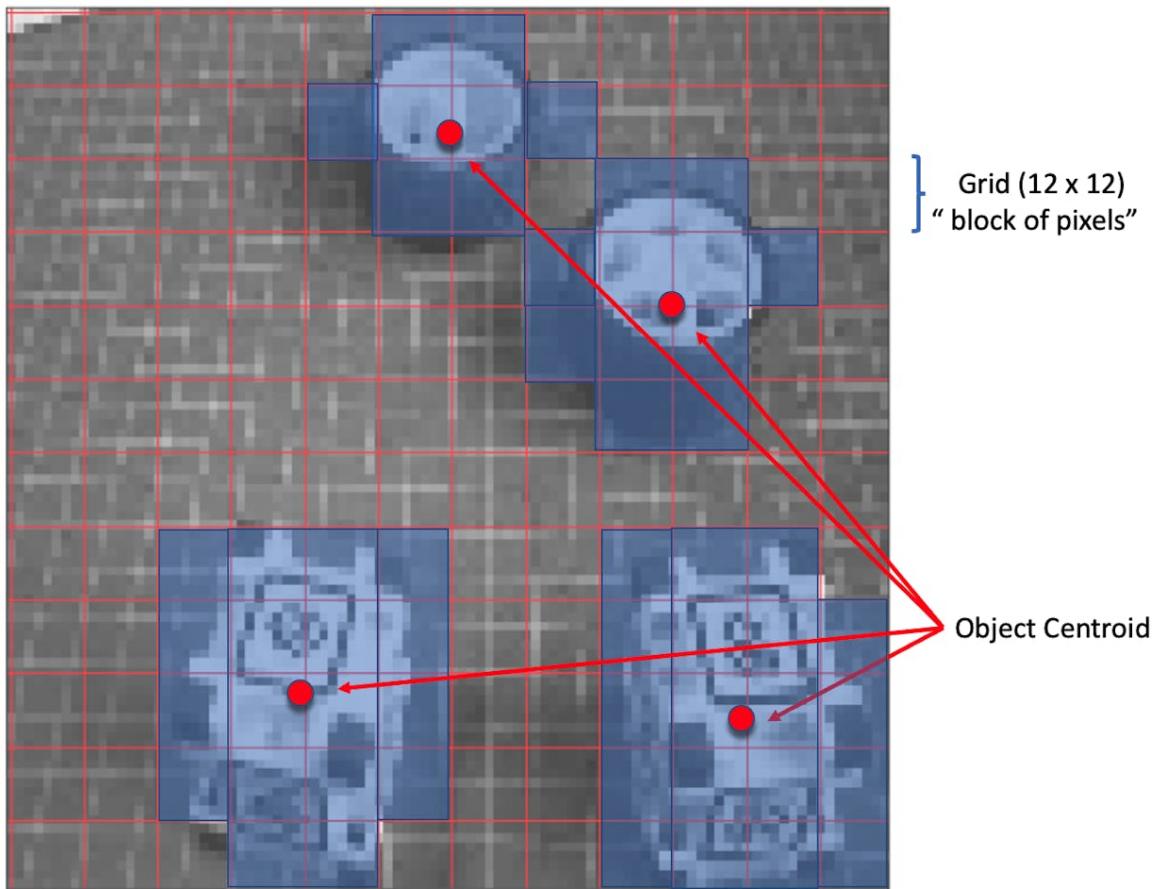


Once these features are extracted, FOMO's simpler architecture, focused on center-point de-

tection, 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 96x96, the grid would be 12x12 ($96/8=12$). For a 160x160, the grid will be 20x20, 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.

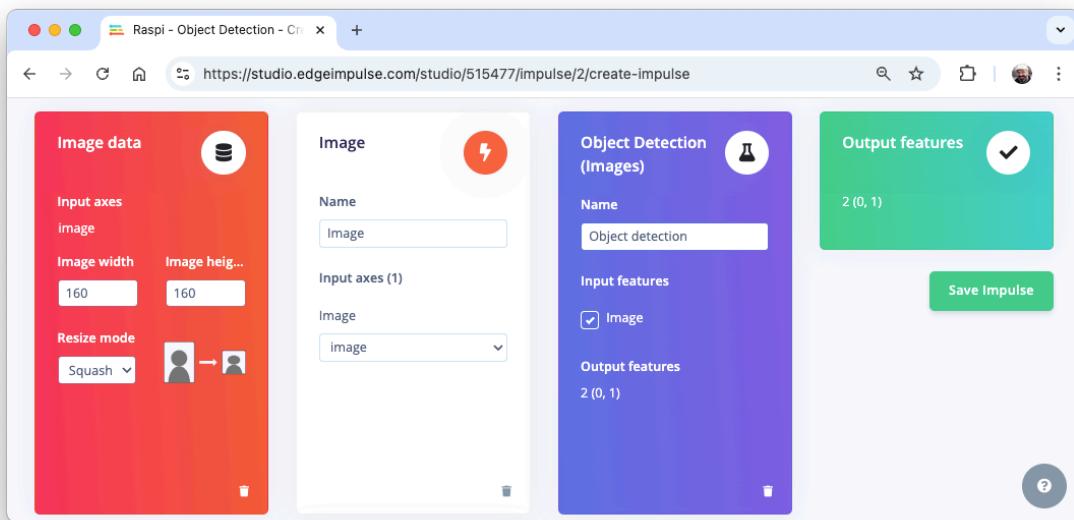


Trade-off Between Speed and Precision:

- **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.

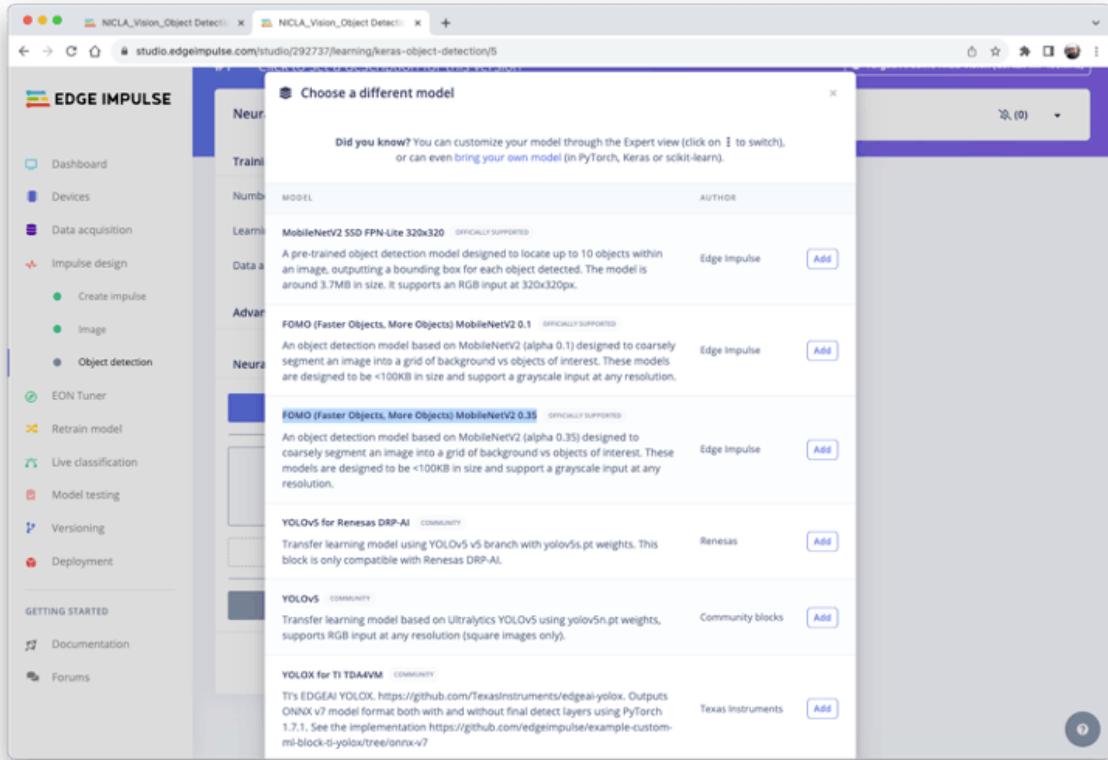
Impulse Design, new Training and Testing

Return to Edge Impulse Studio, and in the **Experiments** tab, create another impulse. Now, the input images should be 160x160 (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 8ms (Raspi-4), around 60X less than we got with the SSD MovieNetV2.

The screenshot shows two main sections of the TensorFlow Model Optimizer interface.

Left Panel (Training settings):

- Neural Network settings:**
 - Training settings:**
 - Number of training cycles: 30
 - Use learned optimizer: checked
 - Learning rate: 0.001
 - Training processor: CPU
 - Data augmentation: unchecked
 - Advanced training settings:**
 - Validation set size: 20 %
 - Split train/validation set on metadata key: empty
 - Batch size: 32
 - Profile int8 model: checked
- Neural network architecture:**
 - Input layer (76,800 features)
 - FOMO (Faster Objects, More Objects) MobileNetV2 0.35
 - Choose a different model
 - Output layer (2 classes)

Right Panel (Model testing):

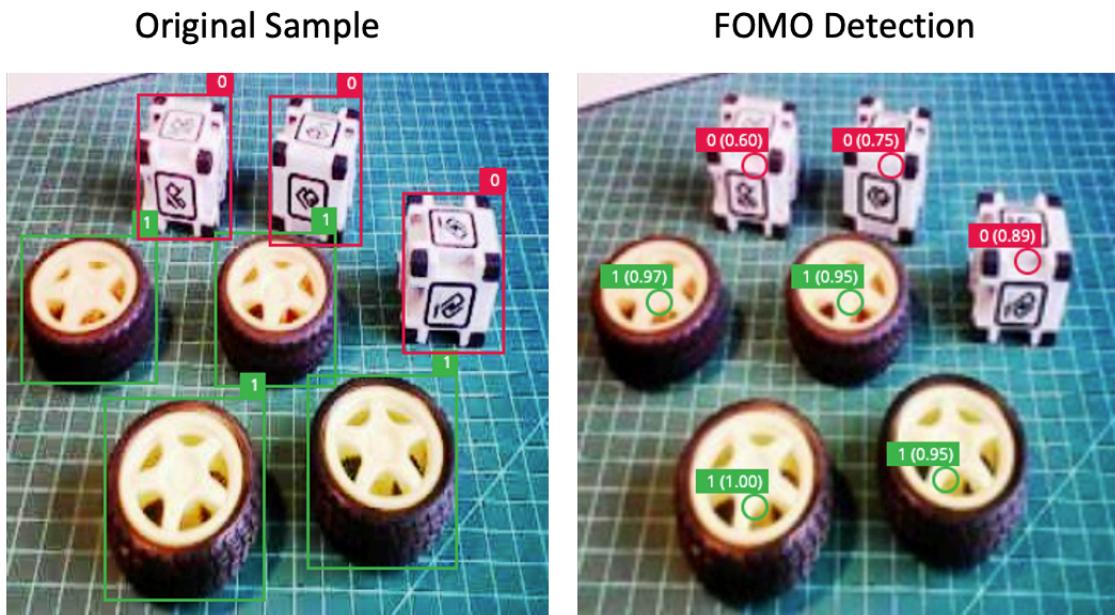
- Model:** Model version: Quantized (int8)
- Last training performance (validation set):**
 - F1 SCORE: 93.3%
- Confusion matrix (validation set):**

Background	0	1
0	9.1%	90.9%
1	10.8%	89.2%
F1 SCORE	1.00	0.92
- Metrics (validation set):**

Metric	Value
Precision (non-background)	0.97
Recall (non-background)	0.90
F1 Score (non-background)	0.93
- On-device performance:** Engine: EON™ Compiler
 - Inferencing time: 8 ms.
 - Peak RAM usage: 626.5K
 - Flash usage: 79.0K

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 screenshot shows the Edge Impulse Studio interface with a project titled "Raspi - Object Detection". The main content area displays deployment options for a "Linux (AARCH64)" target. It includes a section for "DEFAULT DEPLOYMENT" featuring a Linux penguin icon, a brief description, and a note about the Edge Impulse Linux protocol. Below this, there's a "DEPLOY TO ANY LINUX-BASED DEVELOPMENT BOARD" section with instructions for installing the CLI and running the runner. A note says "See the documentation for more information and setup instructions. Alternatively, you can download your model for Linux (AARCH64) below." Two tables compare "Quantized (int8)" and "Unoptimized (float32)" models across metrics: IMAGE, OBJECT DETECTION, and TOTAL. The "Quantized" table shows lower latency (1 ms vs 1 ms), higher RAM usage (4.0K vs 4.0K), and higher accuracy (94.44% vs 94.44%). The "Unoptimized" table shows higher latency (10 ms vs 11 ms), lower RAM usage (2.4M vs 2.4M), and the same accuracy. At the bottom, a note says "Estimate for Raspberry Pi 4 · Change target" and a large blue "Build" button is centered.

	IMAGE	OBJECT DETECTION	TOTAL
LATENCY	1 ms.	8 ms.	9 ms.
RAM	4.0K	626.5K	626.5K
FLASH	-	79.0K	-
ACCURACY			94.44%

	IMAGE	OBJECT DETECTION	TOTAL
LATENCY	1 ms.	10 ms.	11 ms.
RAM	4.0K	2.4M	2.4M
FLASH	-	103.4K	-
ACCURACY			94.44%

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 `./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](#).

Let's set up a Virtual Environment for working with the Linux Python SDK

```
python3 -m venv ~/eilinx  
source ~/eilinx/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)

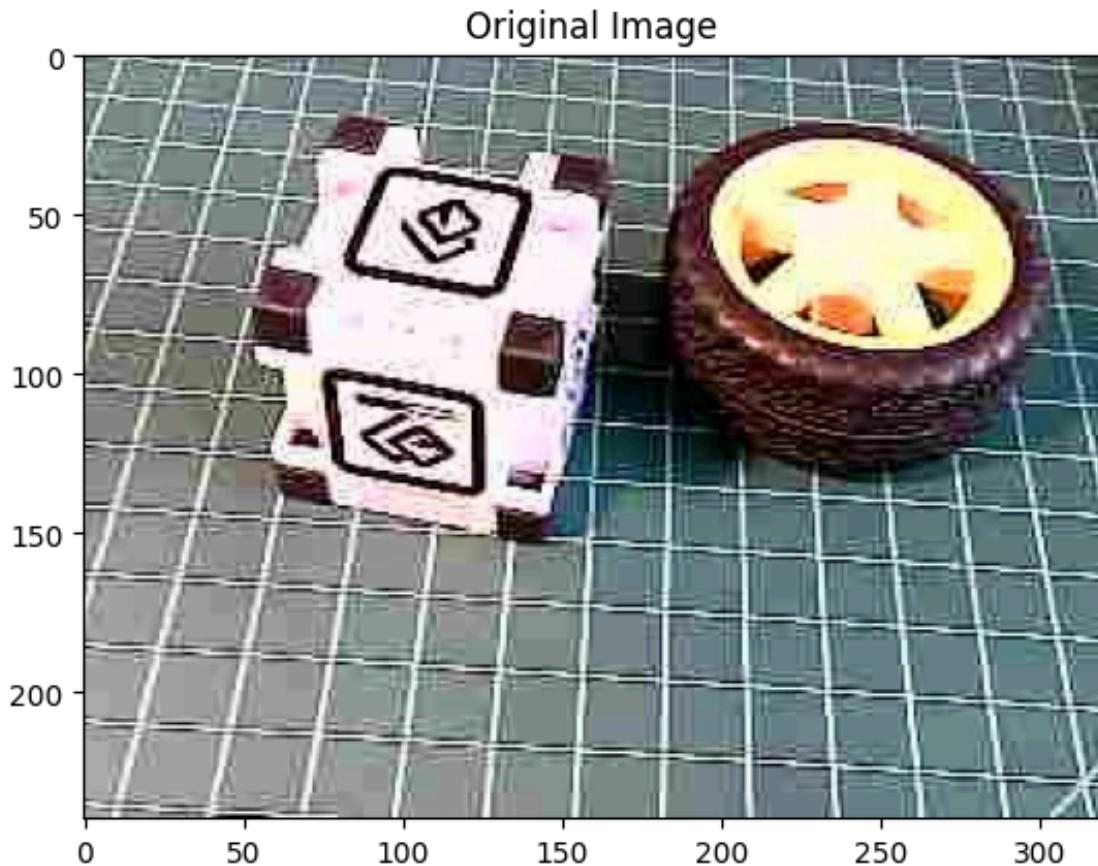
# Initialize model
model_info = runner.init()
```

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_setings(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).

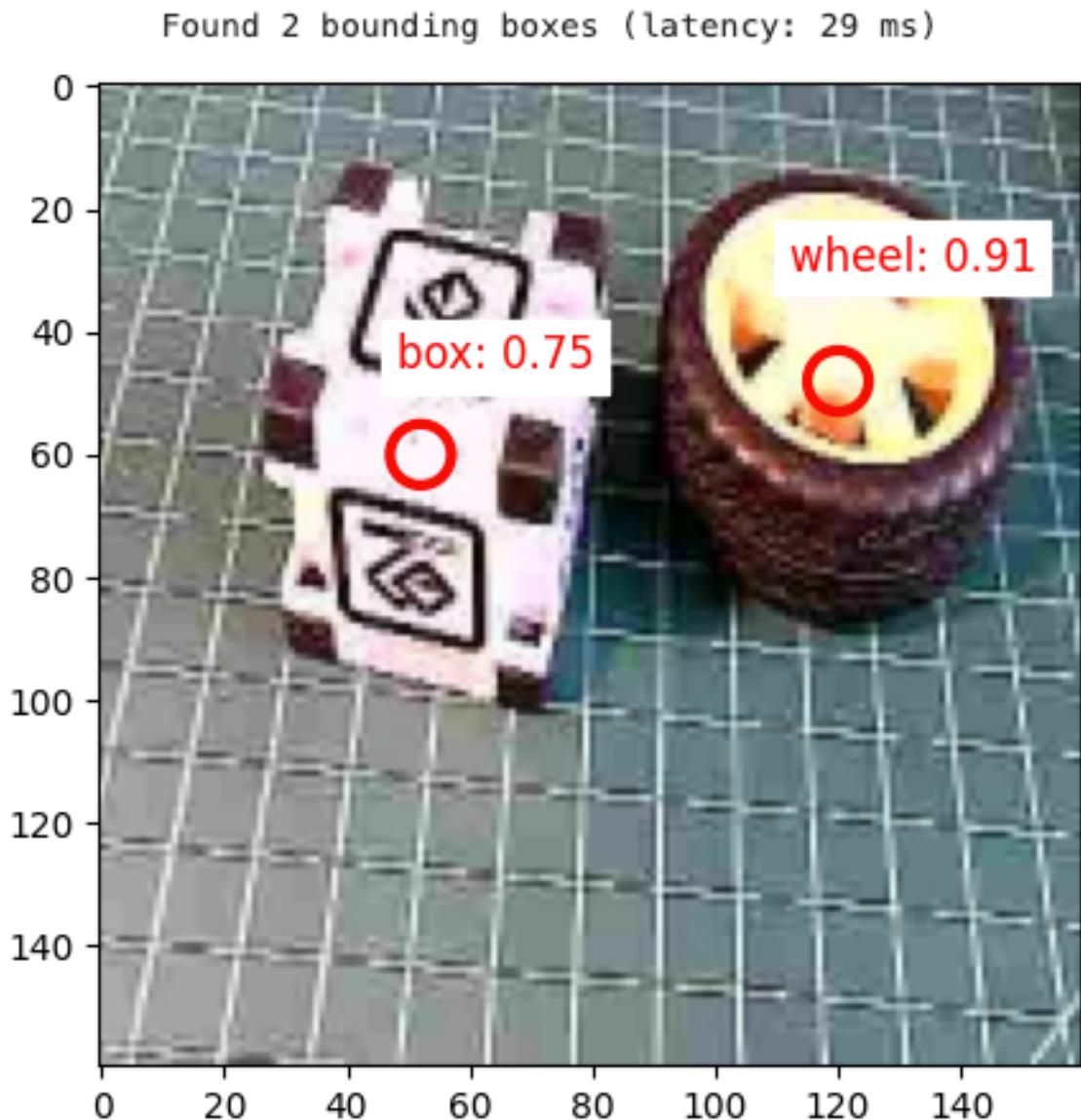
```
print('\tFound %d bounding boxes (latency: %d ms)' % (
    len(res["result"]["bounding_boxes"]),
    res['timing']['dsp'] + res['timing']['classification']))
plt.figure(figsize=(5,5))
plt.imshow(cropped)

# 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:

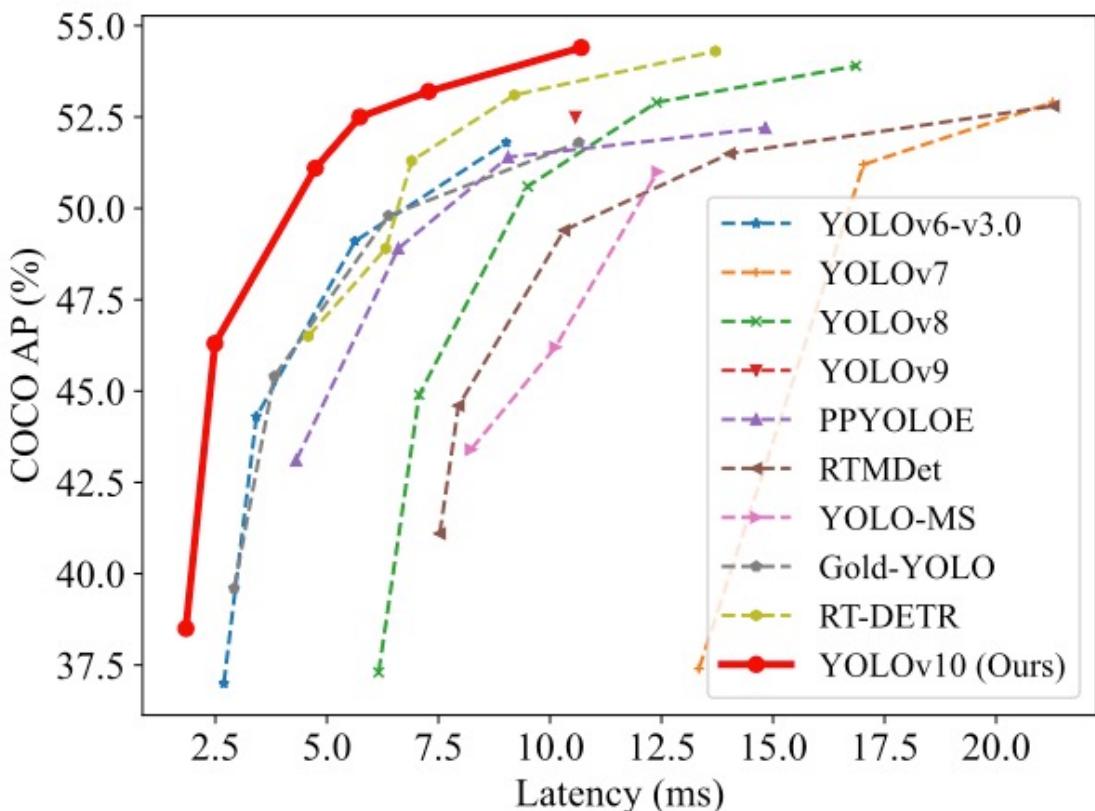
- YOLO employs a single neural network to process the entire image. This network divides the image into a grid and, for each grid cell, directly predicts bounding boxes and associated class probabilities. This end-to-end training improves speed and simplifies the model architecture.

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 Ultralitelycs 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.



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,

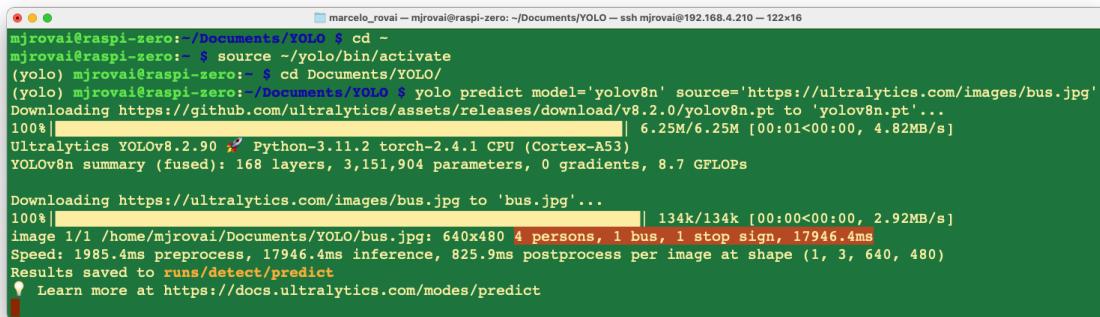
```
source ~/yolo/bin/activate
cd /Documents/YOLO
```

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:



```
marcelo_rovai@raspi-zero:~/Documents/YOLO $ cd -
marcelo_rovai@raspi-zero:~ $ source ~/yolo/bin/activate
(yolo) marcelo_rovai@raspi-zero:~ $ cd Documents/YOLO/
(yolo) marcelo_rovai@raspi-zero:~/Documents/YOLO $ 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]
Ultralytics YOLOv8.2.90 🚀 Python-3.11.2 torch-2.4.1 CPU (Cortex-A53)
YOLOv8n summary (fused): 168 layers, 3,151,904 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/mjrovai/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/modes/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:

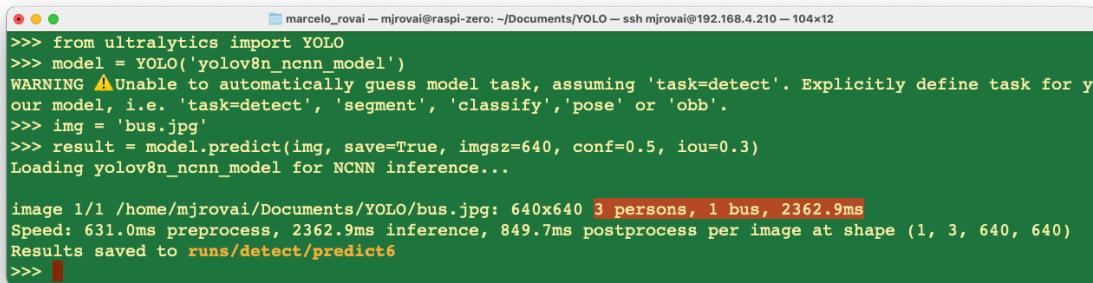
```
python3
```

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)
```

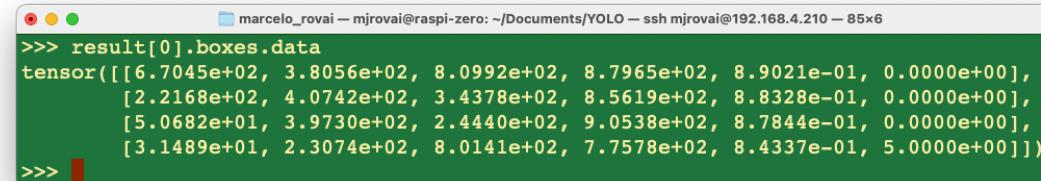


```
marcelo_rovai — mjrovai@raspi-zero: ~/Documents/YOLO — ssh mjrovai@192.168.4.210 — 104x12
>>> from ultralytics import YOLO
>>> model = YOLO('yolov8n_ncnn_model')
WARNING ▲Unable to automatically guess model task, assuming 'task=detect'. Explicitly define task for your model, i.e. 'task=detect', 'segment', 'classify','pose' or 'obb'.
>>> img = 'bus.jpg'
>>> result = model.predict(img, save=True, imgsz=640, conf=0.5, iou=0.3)
Loading yolov8n_ncnn_model for NCNN inference...
image 1/1 /home/mjrovai/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/predict6
>>> █
```

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):



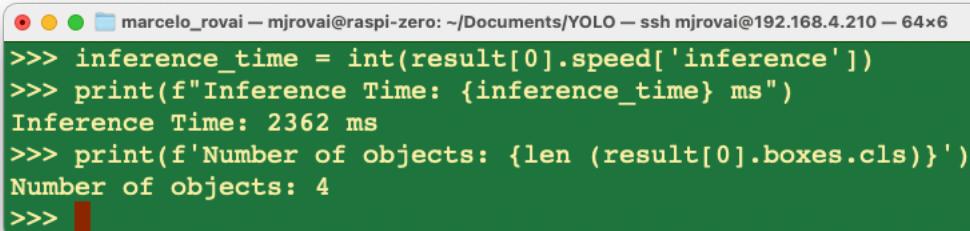
```
marcelo_rovai — mjrovai@raspi-zero: ~/Documents/YOLO — ssh mjrovai@192.168.4.210 — 85x6
>>> result[0].boxes.data
tensor([[6.7045e+02, 3.8056e+02, 8.0992e+02, 8.7965e+02, 8.9021e-01, 0.0000e+00],
       [2.2168e+02, 4.0742e+02, 3.4378e+02, 8.5619e+02, 8.8328e-01, 0.0000e+00],
       [5.0682e+01, 3.9730e+02, 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)}')

```

```

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)}')

^G Help      ^O Write Out   ^W Where Is   ^K Cut        ^T Execute
^X Exit     ^R Read File   ^\ Replace    ^U Paste      ^J Justify

```

And enter with the commands: [CTRL+O] + [ENTER] + [CTRL+X] to save the Python script.

Run the script:

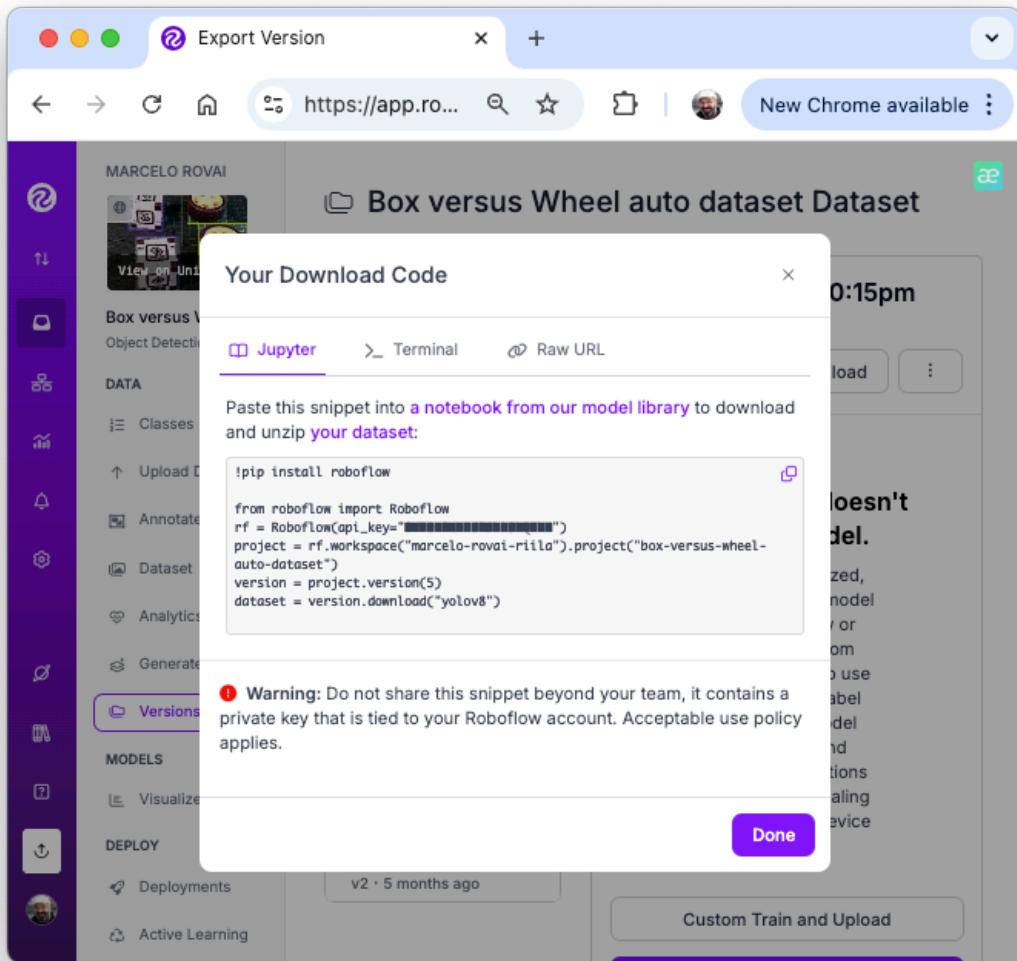
```
python yolov8_tests.py
```

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 “Boxe 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.

```

✓ [3]  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()

→ 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 CoLab, pasting the Download code that we get from Roboflow. Note that our dataset will be mounted under /content/datasets/:

```

File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
Dataset
1 !mkdir {HOME}/datasets
2 %cd {HOME}/datasets
3
4 !pip install roboflow --quiet
5
6 from roboflow import Roboflow
7 rf = Roboflow(api_key="8NEOn2tX76EJbk9hQTb4")
8 project = rf.workspace("marcelo-roval-riila").project("box-versus-wheel-auto-dataset")
9 version = project.version(5)
10 dataset = version.download("yolov8")
11

/content/datasets
Extracting Dataset Version Zip to Box-versus-Wheel-auto-dataset-5 in yolov8: 100%|██████████| 318/318 [00:00<00:00, 8613.15it/s]

```

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
  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}

```

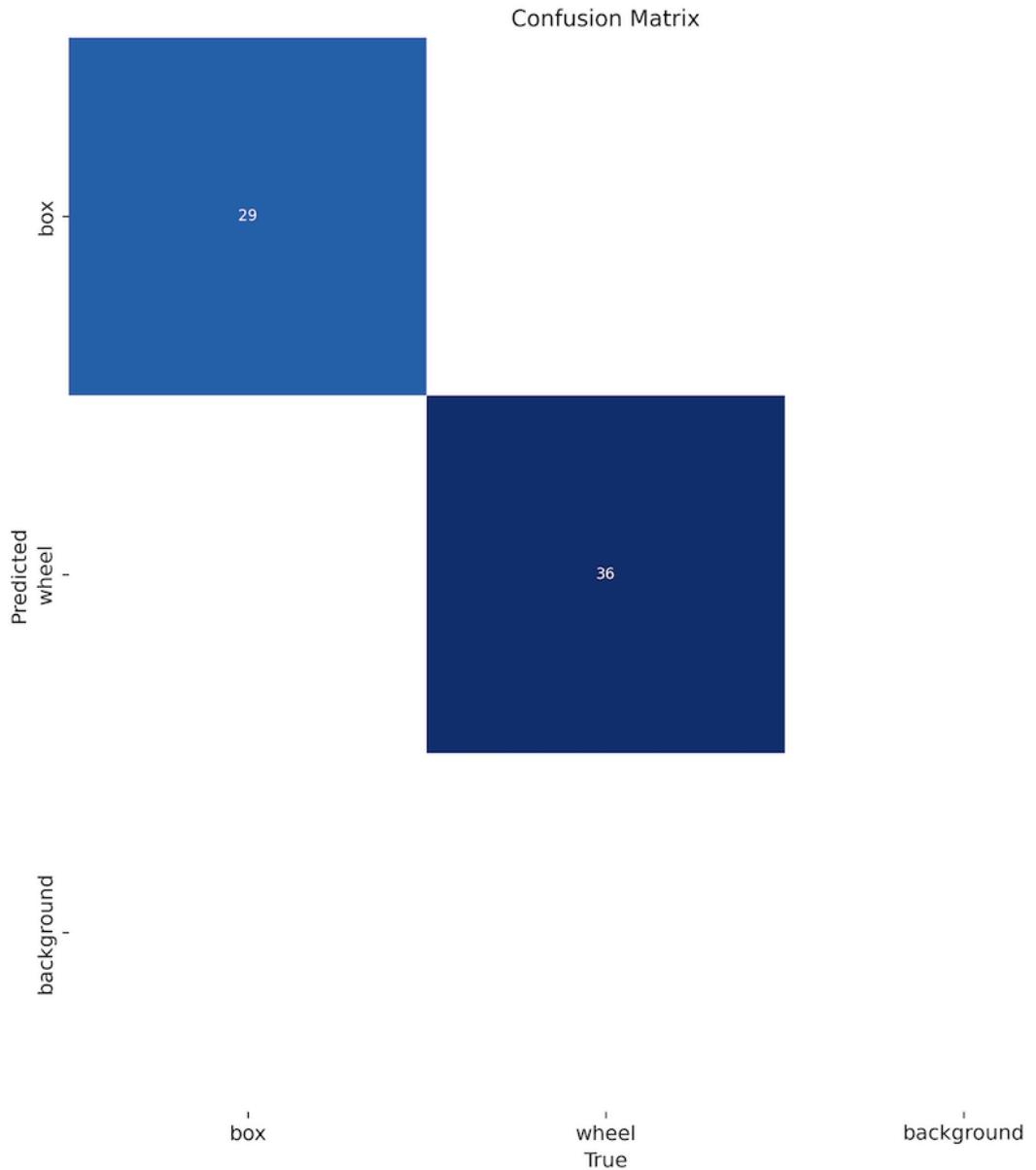
25 epochs completed in 0.026 hours.
Optimizer stripped from runs/detect/train/weights/last.pt, 6.2MB
Optimizer stripped from runs/detect/train/weights/best.pt, 6.2MB

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, 3,006,038 parameters, 0 gradients, 8.1 GFLOPs

Class	Images	Instances	Box(P)	R	mAP50	mAP50-95
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

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.lo
```

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
```

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
```

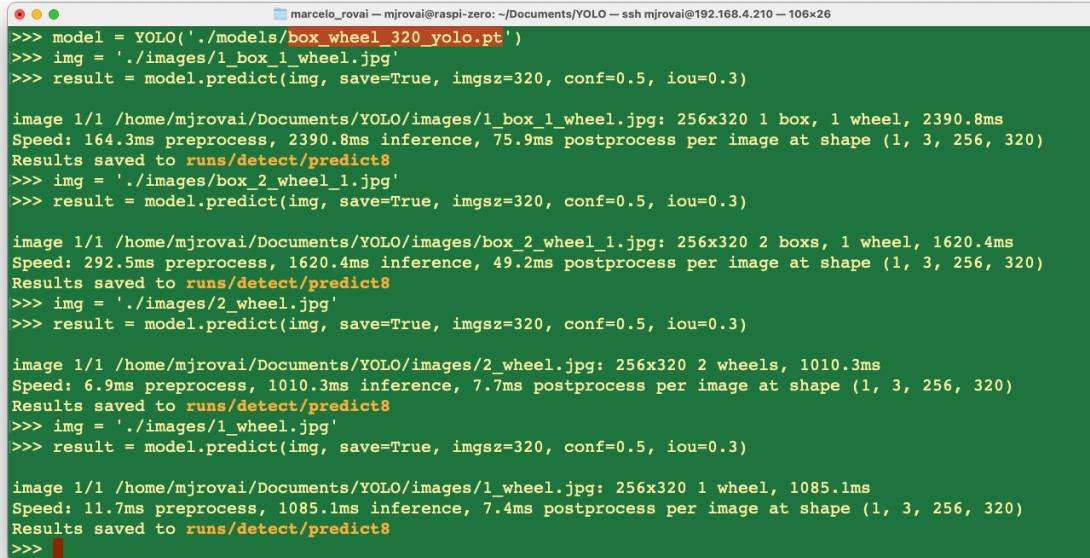
As before, we will import the YOLO library and define our converted model to detect bees:

```
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`



```
marcelo_rovai — mjrovai@raspi-zero: ~/Documents/YOLO — ssh mjrovai@192.168.4.210 — 106x26  
>>> 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/mjrovai/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/mjrovai/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/mjrovai/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/mjrovai/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 backend 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](#).

Conclusion

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](#)

Counting objects with YOLO

Deploying YOLov8 on Raspberry Pi Zero 2W for Real-Time Bee Counting at the Hive Entrance.”



Introduction

At the [Federal University of Itajuba in Brazil](#), with the master's student José Anderson Reis and Professor José Alberto Ferreira Filho, we are exploring a project that delves into the intersection of technology and nature. This tutorial will review our first steps and share our observations on deploying YOLOv8, a cutting-edge machine learning model, on the compact

and efficient Raspberry Pi Zero 2W (*Raspi-Zero*). We aim to estimate the number of bees entering and exiting their hive—a task crucial for beekeeping and ecological studies.

Why is this important? Bee populations are vital indicators of environmental health, and their monitoring can provide essential data for ecological research and conservation efforts. However, manual counting is labor-intensive and prone to errors. By leveraging the power of embedded machine learning, or tinyML, we automate this process, enhancing accuracy and efficiency.



Figure 4: img

This tutorial will cover setting up the Raspberry Pi, integrating a camera module, optimizing and deploying YOLOv8 for real-time image processing, and analyzing the data gathered.

Installing and using Ultralytics 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.

Let's start installing the Ultralytics packages for local inference on the Rasp-Zero:

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

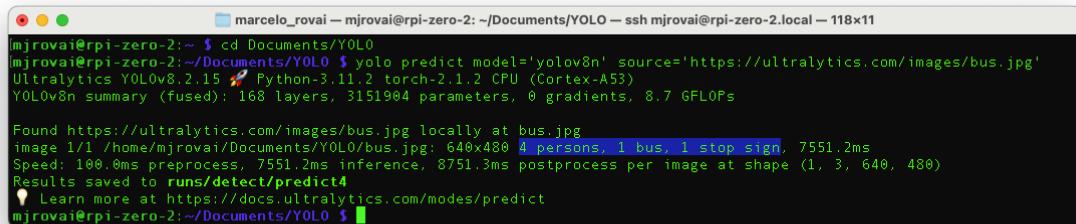
After the Rasp-Zero booting, let's create a directory for working with YOLO and change the current location to it::

```
mkdir Documents/YOLO
cd Documents/YOLO
```

Let's 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'
```

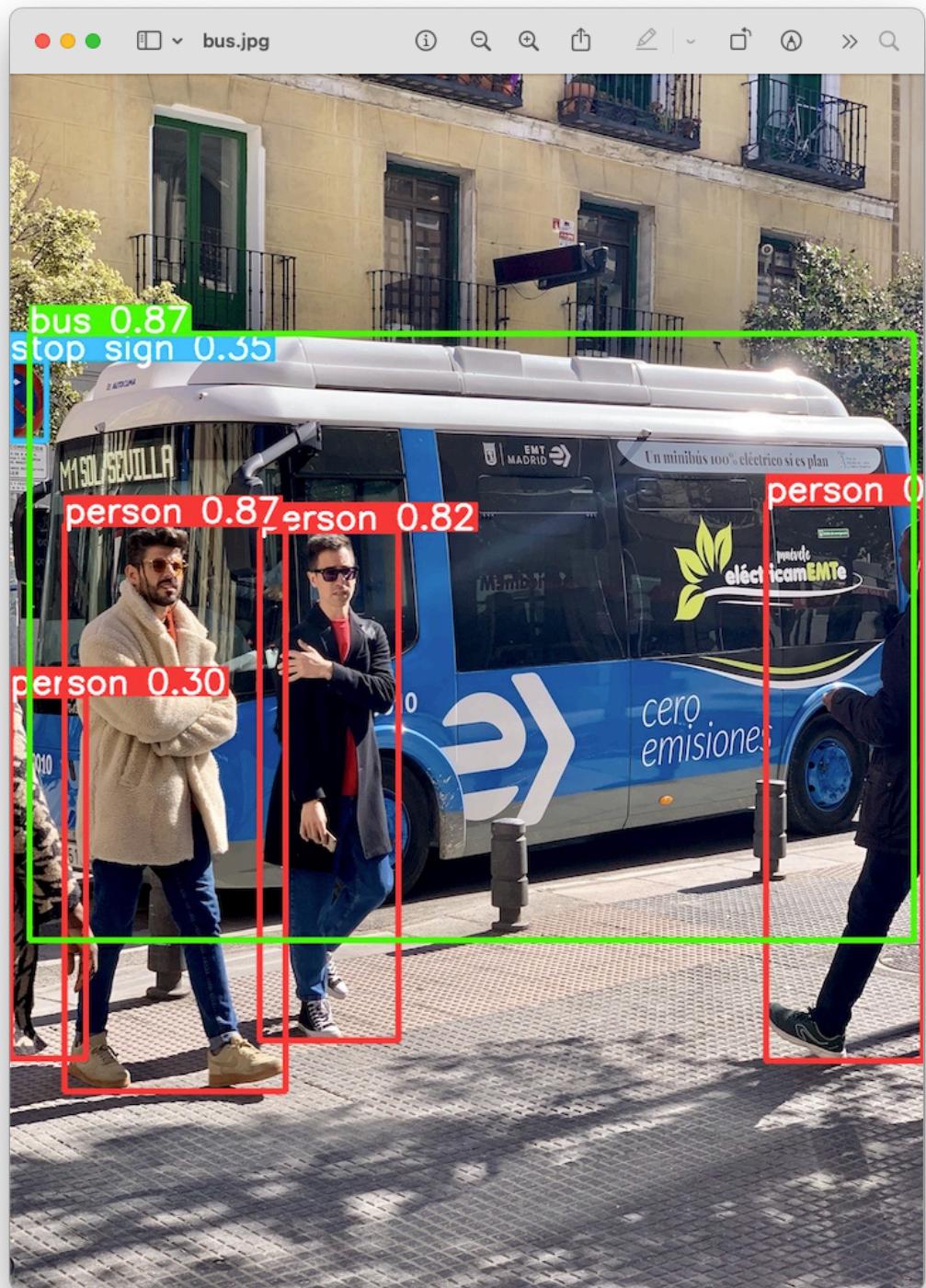
The inference result will appear in the terminal. In the image (bus.jpg), 4 persons, 1 bus, and 1 stop signal were detected:



```
marcelo_rovai@rpi-zero-2: ~/Documents/YOLO — ssh marcelo_rovai@rpi-zero-2.local — 118x11
mjrovai@rpi-zero-2: ~$ cd Documents/YOLO
mjrovai@rpi-zero-2: ~/Documents/YOLO $ yolo predict model='yolov8n' source='https://ultralytics.com/images/bus.jpg'
Ultralytics YOLOv8.2.15 🚀 Python-3.11.2 torch-2.1.2 CPU (Cortex-A53)
YOLOv8n summary (fused): 168 layers, 3151904 parameters, 0 gradients, 8.7 GFLOPs

Found https://ultralytics.com/images/bus.jpg locally at bus.jpg
image 1/1 /home/mjrovai/Documents/YOLO/bus.jpg: 640x480 4 persons, 1 bus, 1 stop sign, 7551.2ms
Speed: 100.0ms preprocess, 7551.2ms inference, 8751.3ms postprocess per image at shape (1, 3, 640, 480)
Results saved to runs/detect/predict4
💡 Learn more at https://docs.ultralytics.com/modes/predict
mjrovai@rpi-zero-2: ~/Documents/YOLO $
```

Also, we got a message that `Results saved to runs/detect/predict4`. Inspecting that directory, we can see a new image saved (bus.jpg). Let's download it from the Rasp-Zero to our desktop for inspection:



So, the Ultralytics YOLO is correctly installed on our Rasp-Zero.

Export Model to NCNN format

So, let's convert our model and rerun the inference:

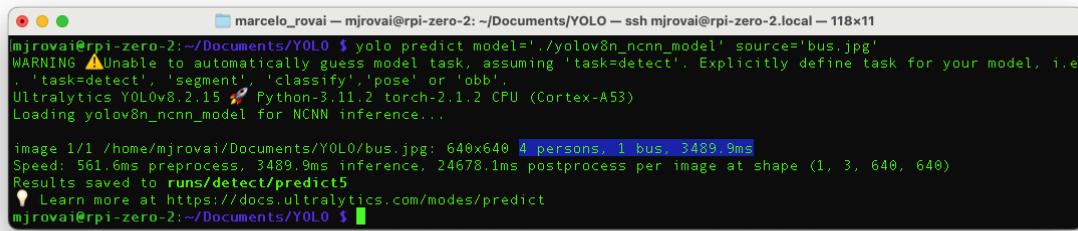
1. Export a YOLOv8n PyTorch model to NCNN format, creating: '/yolov8n_ncnn_model'

```
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'
```

Now, we can see that the latency was reduced by half.

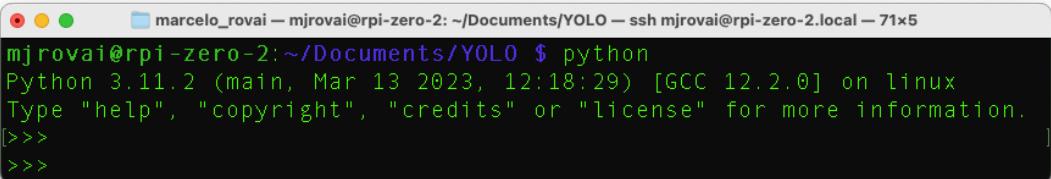


```
marcelo_rovai@rpzi-zero-2:~/Documents/YOLO$ yolo predict model='./yolov8n_ncnn_model' source='bus.jpg'
WARNING ┈ Unable to automatically guess model task, assuming 'task=detect'. Explicitly define task for your model, i.e.
. 'task=detect', 'segment', 'classify', 'pose' or 'obb'
Ultralytics YOLOv8.2.15 Python-3.11.2 torch-2.1.2 CPU (Cortex-A53)
Loading yolov8n_ncnn_model for NCNN inference...
image 1/1 /home/mjrovai/Documents/YOLO/bus.jpg: 640x640 4 persons, 1 bus, 3489.9ms
Speed: 561.6ms preprocess, 3489.9ms inference, 24678.1ms postprocess per image at shape (1, 3, 640, 640)
Results saved to runs/detect/predict
💡 Learn more at https://docs.ultralytics.com/modes/predict
mjrovai@rpzi-zero-2:~/Documents/YOLO$
```

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
```



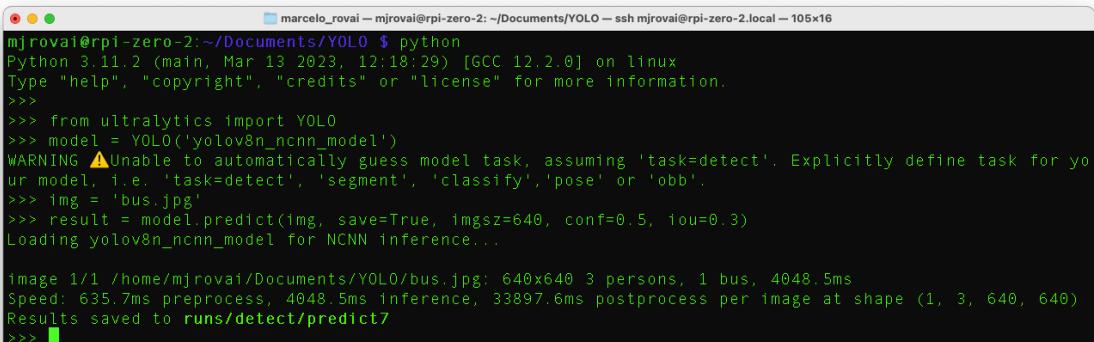
```
mjrovai@rpi-zero-2:~/Documents/YOLO$ python
Python 3.11.2 (main, Mar 13 2023, 12:18:29) [GCC 12.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
[>>>
>>>
```

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, imgs=640, conf=0.5, iou=0.3)
```



```
mjrovai@rpi-zero-2:~/Documents/YOLO$ python
Python 3.11.2 (main, Mar 13 2023, 12:18:29) [GCC 12.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
[>>>
>>> from ultralytics import YOLO
>>> model = YOLO('yolov8n_ncnn_model')
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, imgs=640, conf=0.5, iou=0.3)
Loading yolov8n_ncnn_model for NCNN inference...
image 1/1 /home/mjrovai/Documents/YOLO/bus.jpg: 640x640 3 persons, 1 bus, 4048.5ms
Speed: 635.7ms preprocess, 4048.5ms inference, 33897.6ms postprocess per image at shape (1, 3, 640, 640)
Results saved to runs/detect/predict7
>>> █
```

We can verify that the result is the same as the one we get running the inference at the terminal level (CLI).

```
image 1/1 /home/mjrovai/Documents/YOLO/bus.jpg: 640x640 3 persons, 1 bus, 4048.5ms
Speed: 635.7ms preprocess, 4048.5ms inference, 33897.6ms postprocess per image at shape (1, 3, 640, 640)
Results saved to runs/detect/predict7
```

But, we are interested in analyzing 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):

```
marcelo_rovai — mjrovai@rpi-zero-2: ~/Documents/YOLO — ssh mjrovai@rpi-zero-2.local — 83x6
>>> result[0].boxes.data
tensor([[6.7102e+02, 3.7803e+02, 8.1000e+02, 8.7980e+02, 9.0090e-01, 0.0000e+00],
       [2.2142e+02, 4.0759e+02, 3.4348e+02, 8.5584e+02, 8.8382e-01, 0.0000e+00],
       [5.0357e+01, 3.9810e+02, 2.4408e+02, 9.0484e+02, 8.7554e-01, 0.0000e+00],
       [3.4337e+01, 2.2932e+02, 7.9558e+02, 7.6855e+02, 8.4215e-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@rpi-zero-2: ~/Documents/YOLO — ssh mjrovai@rpi-zero-2.local — 62x6
>>> inference_time = int(result[0].speed['inference'])
>>> print(f"Inference Time: {inference_time} ms")
Inference Time: 4048 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. In our final project, we will run a Python script at once rather than manually entering it line by line in the interpreter.

For that, 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
```

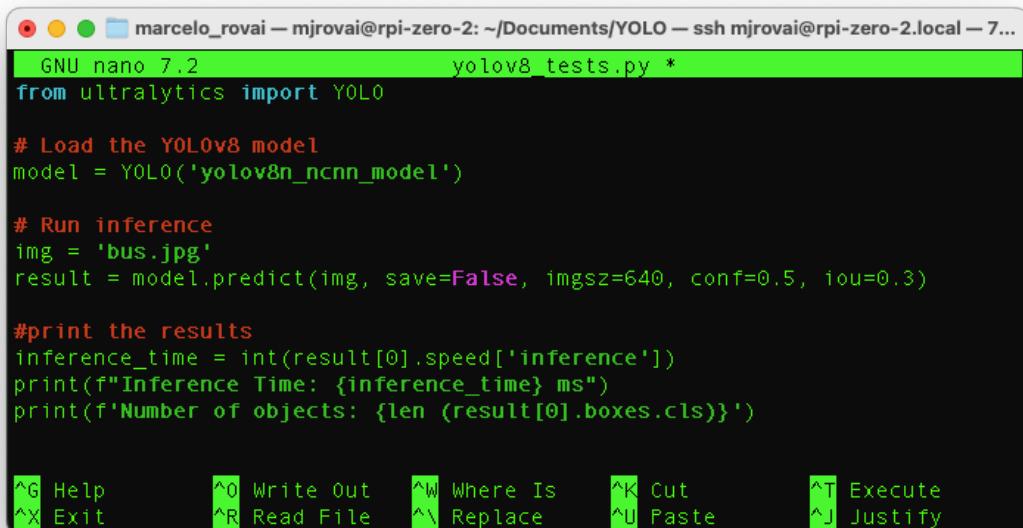
Enter with the code lines:

```
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)}')
```



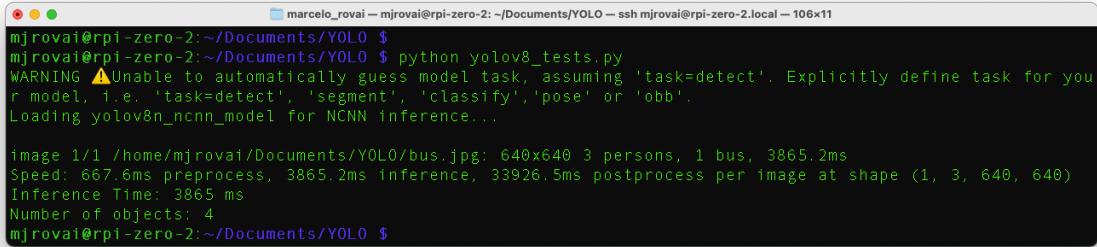
The screenshot shows a terminal window titled "marcelo_rovai — mjrovai@rpi-zero-2: ~/Documents/YOLO — ssh mjrovai@rpi-zero-2.local — 7...". The window displays the content of the file "yolov8_tests.py". The code is identical to the one shown in the previous code block. At the bottom of the terminal, there is a menu bar with various keyboard shortcuts for navigating the text editor.

^G Help	^O Write Out	^W Where Is	^K Cut	^T Execute
^X Exit	^R Read File	^V Replace	^U Paste	^J Justify

And enter with the commands: [CTRL+O] + [ENTER] + [CTRL+X] to save the Python script.

Run the script:

```
python yolov8_tests.py
```



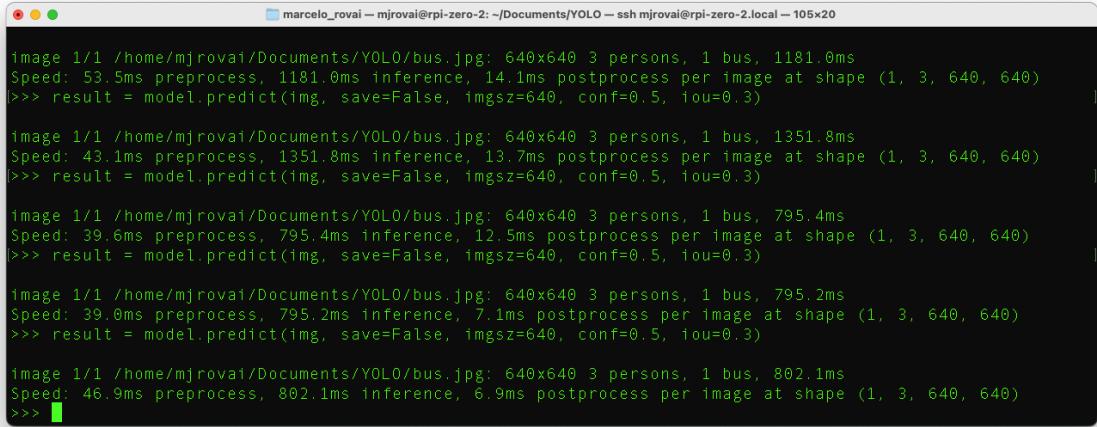
```
marcelo_rovai@rpi-zero-2:~/Documents/YOLO$ python yolov8_tests.py
WARNING ▲Unable to automatically guess model task, assuming 'task=detect'. Explicitly define task for your model, i.e. 'task=detect', 'segment', 'classify', 'pose' or 'obb'.
Loading yolov8n_ncnn_model for NCNN inference...

image 1/1 /home/mjrovai/Documents/YOLO/bus.jpg: 640x640 3 persons, 1 bus, 3865.2ms
Speed: 667.6ms preprocess, 3865.2ms inference, 33926.5ms postprocess per image at shape (1, 3, 640, 640)
Inference Time: 3865 ms
Number of objects: 4
mjrovai@rpi-zero-2:~/Documents/YOLO $
```

We can verify again that the result is precisely the same as when we run the inference at the terminal level (CLI) and with the built-in Python interpreter.

Note about the Latency:

The process of 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 took 3 to 4 seconds, but after that, the inference time is reduced to less than 1 second.



```
marcelo_rovai@rpi-zero-2:~/Documents/YOLO$ python yolov8_tests.py
image 1/1 /home/mjrovai/Documents/YOLO/bus.jpg: 640x640 3 persons, 1 bus, 1181.0ms
Speed: 53.5ms preprocess, 1181.0ms inference, 14.1ms postprocess per image at shape (1, 3, 640, 640)
>>> result = model.predict(img, save=False, imgsz=640, conf=0.5, iou=0.3)

image 1/1 /home/mjrovai/Documents/YOLO/bus.jpg: 640x640 3 persons, 1 bus, 1351.8ms
Speed: 43.1ms preprocess, 1351.8ms inference, 13.7ms postprocess per image at shape (1, 3, 640, 640)
>>> result = model.predict(img, save=False, imgsz=640, conf=0.5, iou=0.3)

image 1/1 /home/mjrovai/Documents/YOLO/bus.jpg: 640x640 3 persons, 1 bus, 795.4ms
Speed: 39.6ms preprocess, 795.4ms inference, 12.5ms postprocess per image at shape (1, 3, 640, 640)
>>> result = model.predict(img, save=False, imgsz=640, conf=0.5, iou=0.3)

image 1/1 /home/mjrovai/Documents/YOLO/bus.jpg: 640x640 3 persons, 1 bus, 795.2ms
Speed: 39.0ms preprocess, 795.2ms inference, 7.1ms postprocess per image at shape (1, 3, 640, 640)
>>> result = model.predict(img, save=False, imgsz=640, conf=0.5, iou=0.3)

image 1/1 /home/mjrovai/Documents/YOLO/bus.jpg: 640x640 3 persons, 1 bus, 802.1ms
Speed: 46.9ms preprocess, 802.1ms inference, 6.9ms postprocess per image at shape (1, 3, 640, 640)
>>> █
```

Estimating the number of Bees

For our project at the university, we are preparing to collect a dataset of bees at the entrance of a beehive using the same camera connected to the Rasp-Zero. The images should be collected every 10 seconds. With the Arducam OV5647, the horizontal Field of View (FoV) is 53.5° , which means that a camera positioned at the top of a standard Hive (46 cm) will capture all of its entrance (about 47 cm).

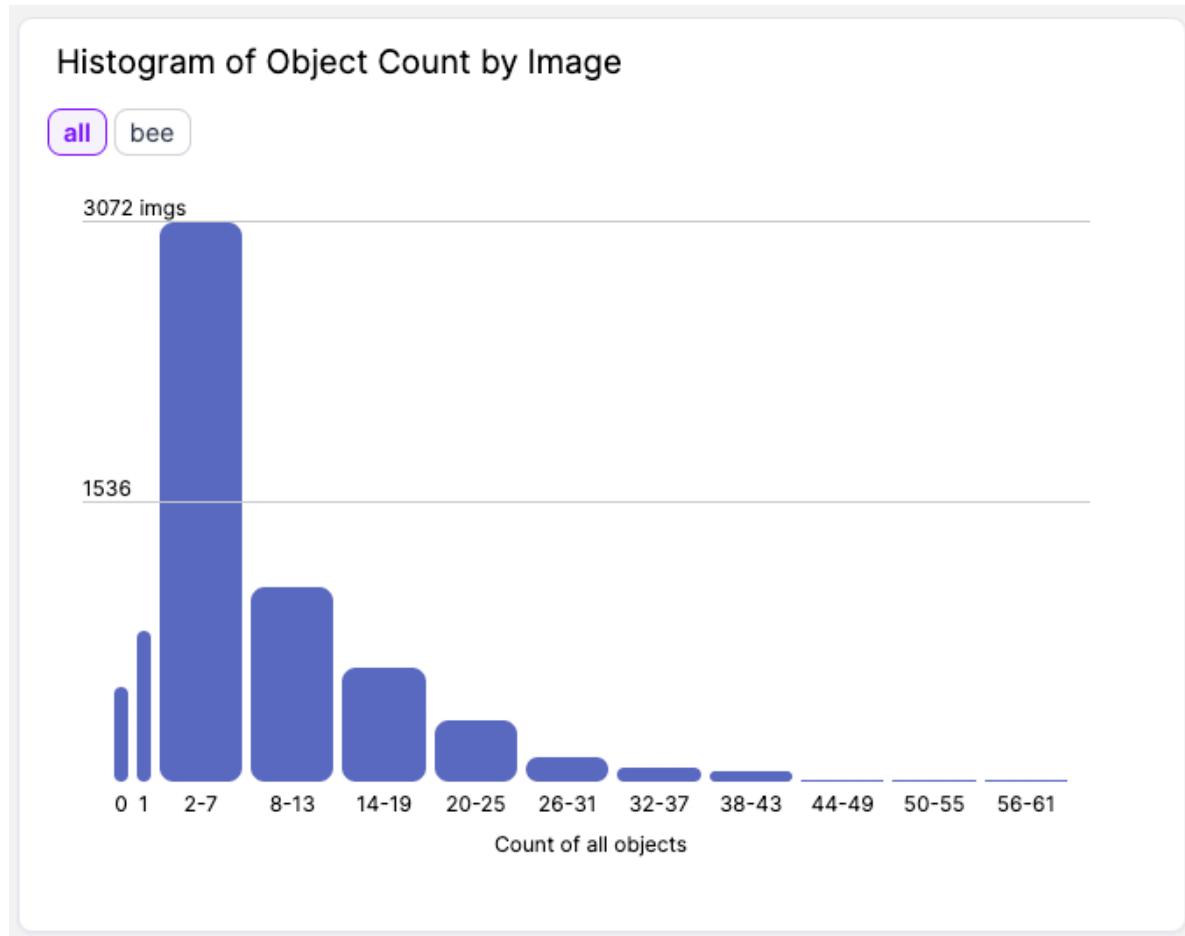


Dataset

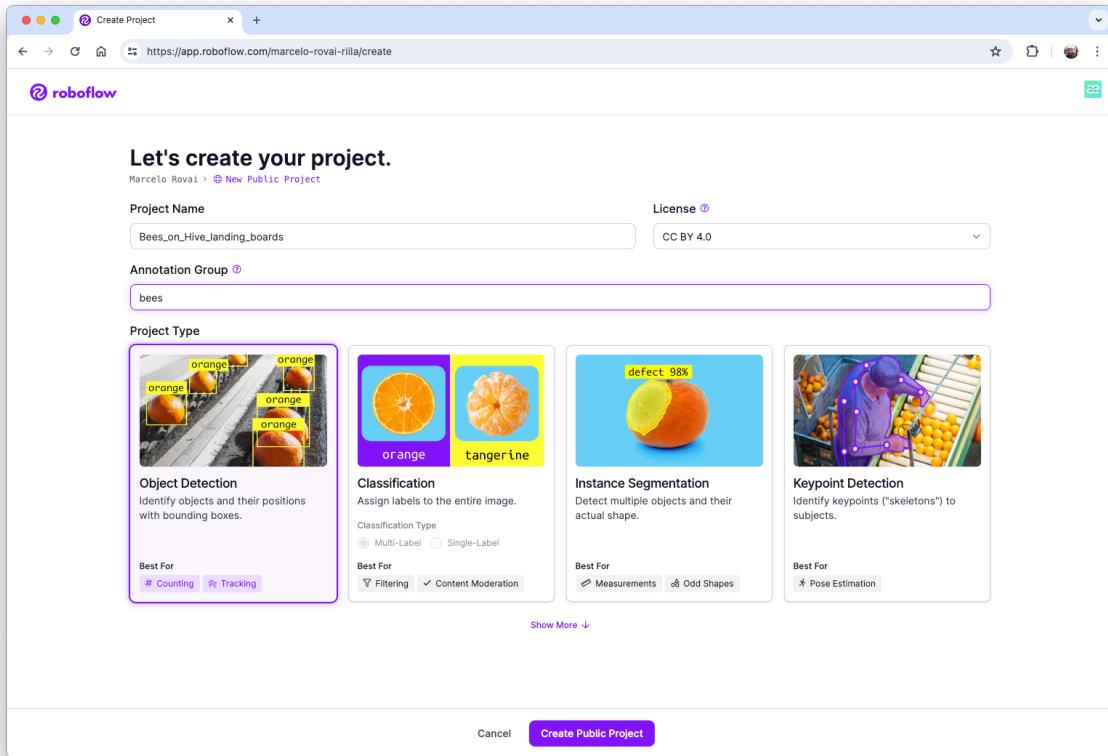
The dataset collection is the most critical phase of the project and should take several weeks or months. For this tutorial, we will use a public dataset: “Sledevic, Tomyslav (2023), “[Labeled dataset for bee detection and direction estimation on beehive landing boards,” Mendeley Data, V5, doi: 10.17632/8gb9r2yhfc.5”

The original dataset has 6,762 images (1920 x 1080), and around 8% of them (518) have no bees (only background). This is very important with Object Detection, where we should keep around 10% of the dataset with only background (without any objects to be detected).

The images contain from zero to up to 61 bees:

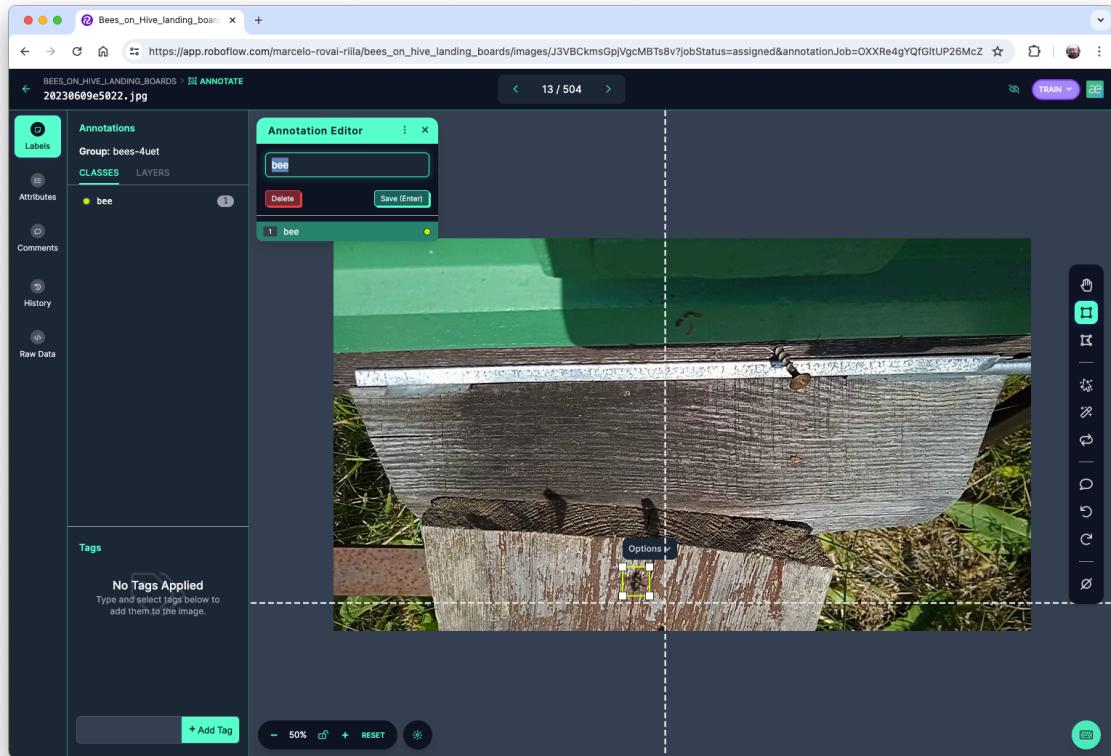


We downloaded the dataset (images and annotations) and uploaded it to [Roboflow](#). There, you should create a free account and start a new project, for example, (“Bees_on_Hive_landing_boards”):

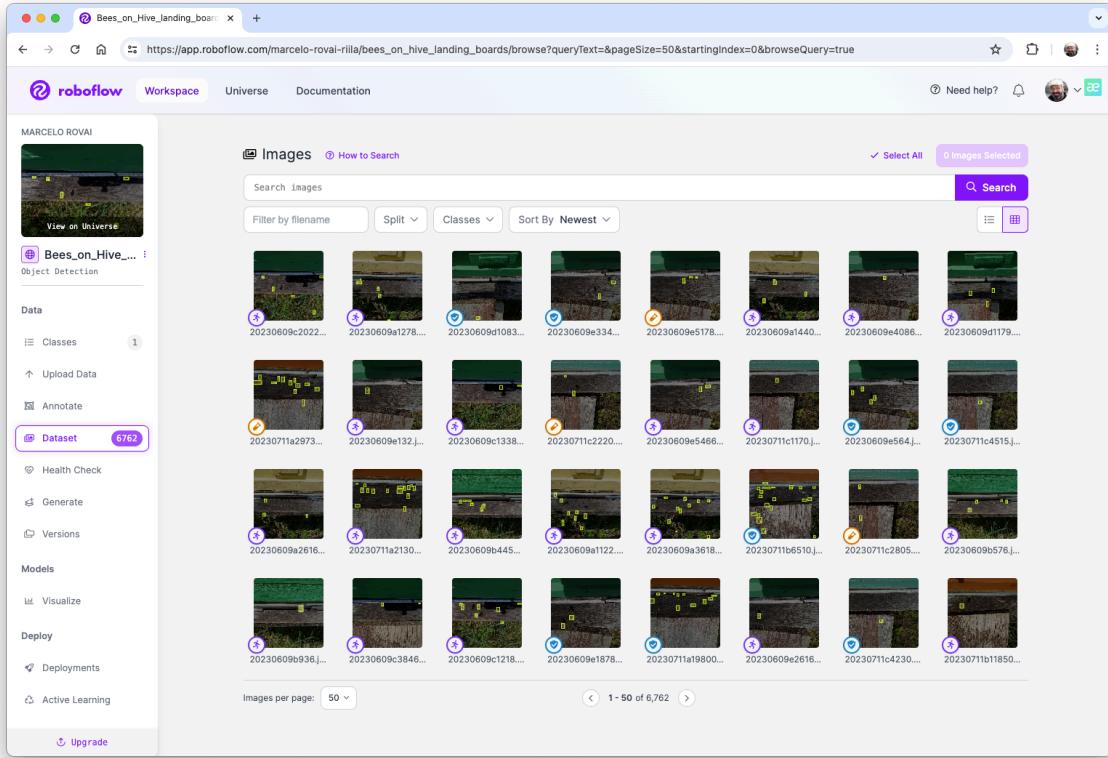


We will not enter details about the Roboflow process once many tutorials are available.

Once the project is created and the dataset is uploaded, you should review the annotations using the “Auto-Label” Tool. Note that all images with only a background should be saved w/o any annotations. At this step, you can also add additional images.



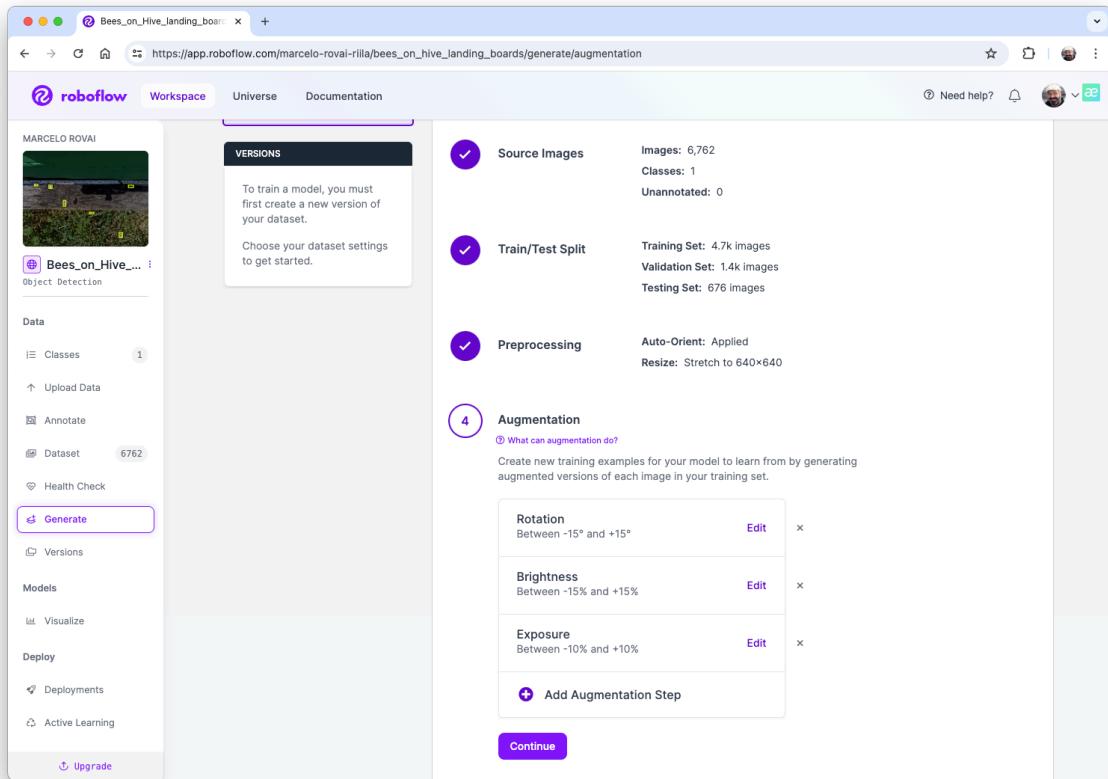
Once all images are annotated, you should split them into training, validation, and testing.



Pre-Processing

The last step with the dataset is preprocessing to generate a final version for training. The Yolov8 model can be trained with 640 x 640 pixels (RGB) images. Let's resize all images 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°) and vary the brightness and exposure.



This will create a final dataset of 16,228 images.

16228 Total Images [View All Images →](#)



Dataset Split

TRAIN SET

87%

14199 Images

VALID SET

8%

1353 Images

TEST SET

4%

676 Images

Preprocessing Auto-Orient: Applied
 Resize: Stretch to 640x640

Augmentations Outputs per training example: 3
 Rotation: Between -15° and +15°
 Brightness: Between -15% and +15%
 Exposure: Between -10% and +10%

Now, you should export the annotated dataset in a YOLOv8 format. You can download a zipped version of the dataset to your desktop or get a downloaded code to be used with a Jupyter Notebook:

Your Download Code



Jupyter

Terminal

Raw URL

Paste this snippet into [a notebook from our model library](#) to download and unzip [your dataset](#):

```
!pip install roboflow

from roboflow import Roboflow
rf = Roboflow(api_key="REDACTED")
project = rf.workspace("marcelo-roval-riila").project("bees_on_hive_landing_boards")
version = project.version(1)
dataset = version.download("yolov8")
```



⚠ Warning: Do not share this snippet beyond your team, it contains a private key that is tied to your Roboflow account. Acceptable use policy applies.

Done

And that is it! We are prepared to start our training using Google Colab.

The pre-processed dataset can be found at the [Roboflow site](#).

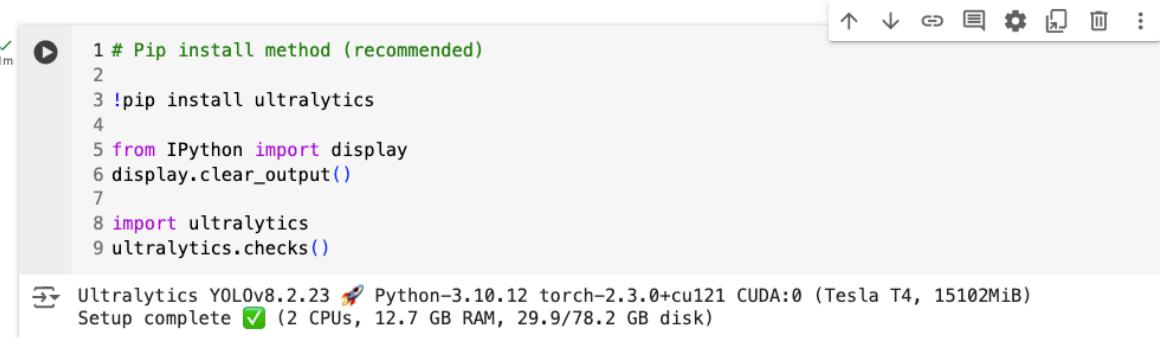
Training YOLOv8 on a Customized Dataset

For training, let's adapt one of the public examples available from Ultralitytics and run it on Google Colab:

- yolov8_bees_on_hive_landing_board.ipynb [\[Open In Colab\]](#)

Critical points on the Notebook:

1. Run it with GPU (the NVidia T4 is free)
2. Install Ultralytics using PIP.



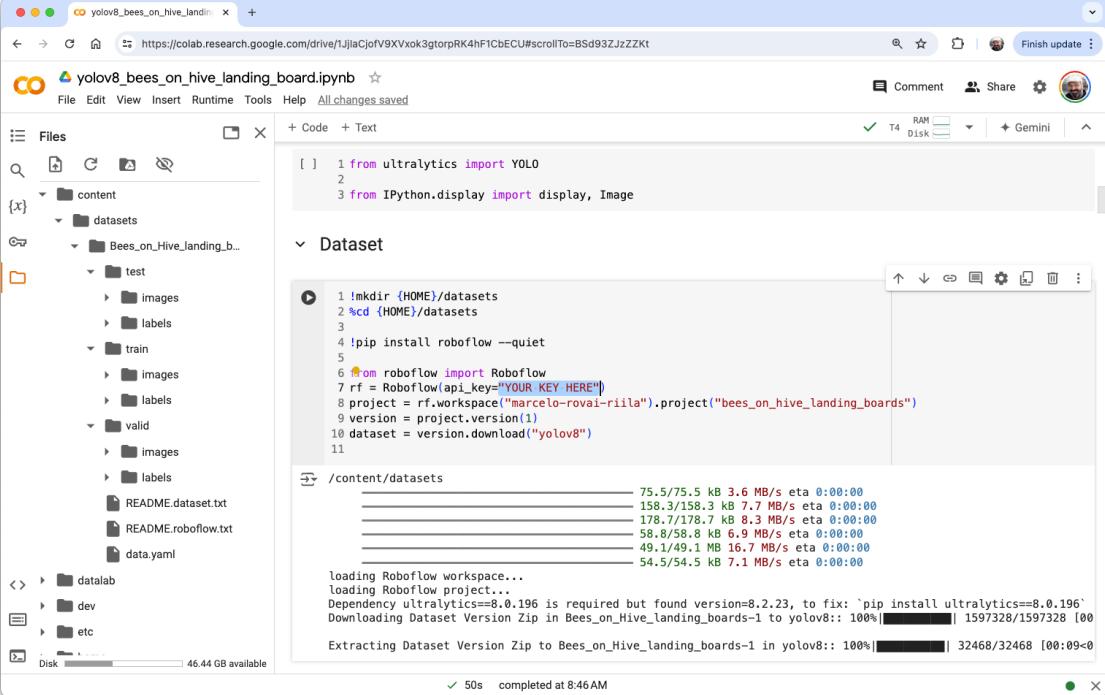
```

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()

```

Setup complete ✓ (2 CPUs, 12.7 GB RAM, 29.9/78.2 GB disk)

3. Now, you can import the YOLO and upload your dataset to the CoLab, pasting the Download code that you get from Roboflow. Note that your dataset will be mounted under /content/datasets/:



```

[ ] 1 from ultralytics import YOLO
2
3 from IPython.display import display, Image

[ ] 1 !mkdir {HOME}/datasets
2 %cd {HOME}/datasets
3
4 !pip install roboflow --quiet
5
6 from roboflow import Roboflow
7 rf = Roboflow(api_key="YOUR KEY HERE")
8 project = rf.workspace("marcelo-rovali-riila").project("beans_on_hive_landing")
9 version = project.version(1)
10 dataset = version.download("yolov8")
11

```

```

[ ] /content/datasets
      75.5/75.5 kB 3.6 MB/s eta 0:00:00
      158.3/158.3 kB 7.7 MB/s eta 0:00:00
      178.7/178.7 kB 8.3 MB/s eta 0:00:00
      58.8/58.8 kB 6.9 MB/s eta 0:00:00
      49.1/49.1 kB 16.7 MB/s eta 0:00:00
      54.5/54.5 kB 7.1 MB/s eta 0:00:00
loading Roboflow workspace...
loading Roboflow project...
Dependency ultralytics==8.0.196 is required but found version=8.2.23, to fix: 'pip install ultralytics==8.0.196'
Downloading Dataset Version Zip in Beans_on_Hive_landing-1 to yolov8:: 100%|██████████| 1597328/1597328 [00:00<00:00]
Extracting Dataset Version Zip to Beans_on_Hive_landing-1 in yolov8:: 100%|██████████| 32468/32468 [00:09<00:00]

```

4. It is important to verify and change, if needed, the file `data.yaml` with the correct path for the images:

```

names:
- bee
nc: 1
roboflow:
  license: CC BY 4.0
  project: bees_on_hive_landing_boards
  url: https://universe.roboflow.com/marcelo-rovai-riila/bees_on_hive_landing_boards/datas
  version: 1
  workspace: marcelo-rovai-riila
test: /content/datasets/Bees_on_Hive_landing_boards-1/test/images
train: /content/datasets/Bees_on_Hive_landing_boards-1/train/images
val: /content/datasets/Bees_on_Hive_landing_boards-1/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):

```

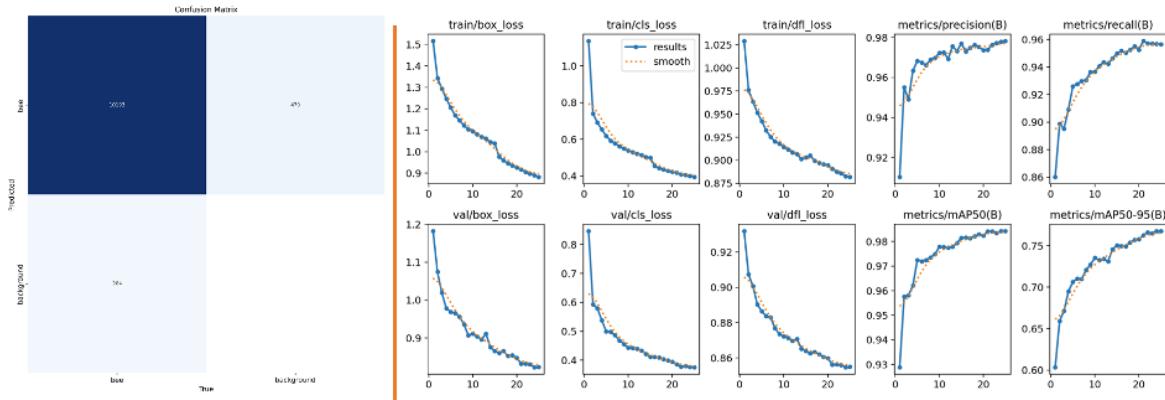
!yolo task=detect mode=train model={MODEL} data={dataset.location}/data.yaml epochs={EPOCHS}

25 epochs completed in 2.679 hours.
Optimizer stripped from runs/detect/train3/weights/last.pt, 6.2MB
Optimizer stripped from runs/detect/train3/weights/best.pt, 6.2MB

Validating runs/detect/train3/weights/best.pt...
Ultralytics YOLOv8.2.15 🚀 Python-3.10.12 torch-2.2.1+cu121 CUDA:0 (Tesla T4, 15102MiB)
Model summary (fused): 168 layers, 3005843 parameters, 0 gradients, 8.1 GFLOPs
    Class   Images   Instances      Box(P)      R      mAP50      mAP50-95: 100% 43/43 [00:33<00:00,  1.27it/s]
        all     1353     10477     0.978     0.957     0.984     0.768
Speed: 0.3ms preprocess, 2.5ms inference, 0.0ms loss, 5.6ms postprocess per image
Results saved to runs/detect/train3

```

The model took 2.7 hours to train and has an excellent result (mAP50 of 0.984). At the end of the training, all results are saved in the folder listed, for example: `/runs/detect/train3/`. There, you can find, for example, the confusion matrix and the metrics curves per epoch.



7. Note that the trained model (`best.pt`) is saved in the folder `/runs/detect/train3/weights/`. Now, you should validate the trained model with the `valid/images`.

```
!yolo task=detect mode=val model={HOME}/runs/detect/train3/weights/best.pt data={dataset}.l
```

The results were similar to training.

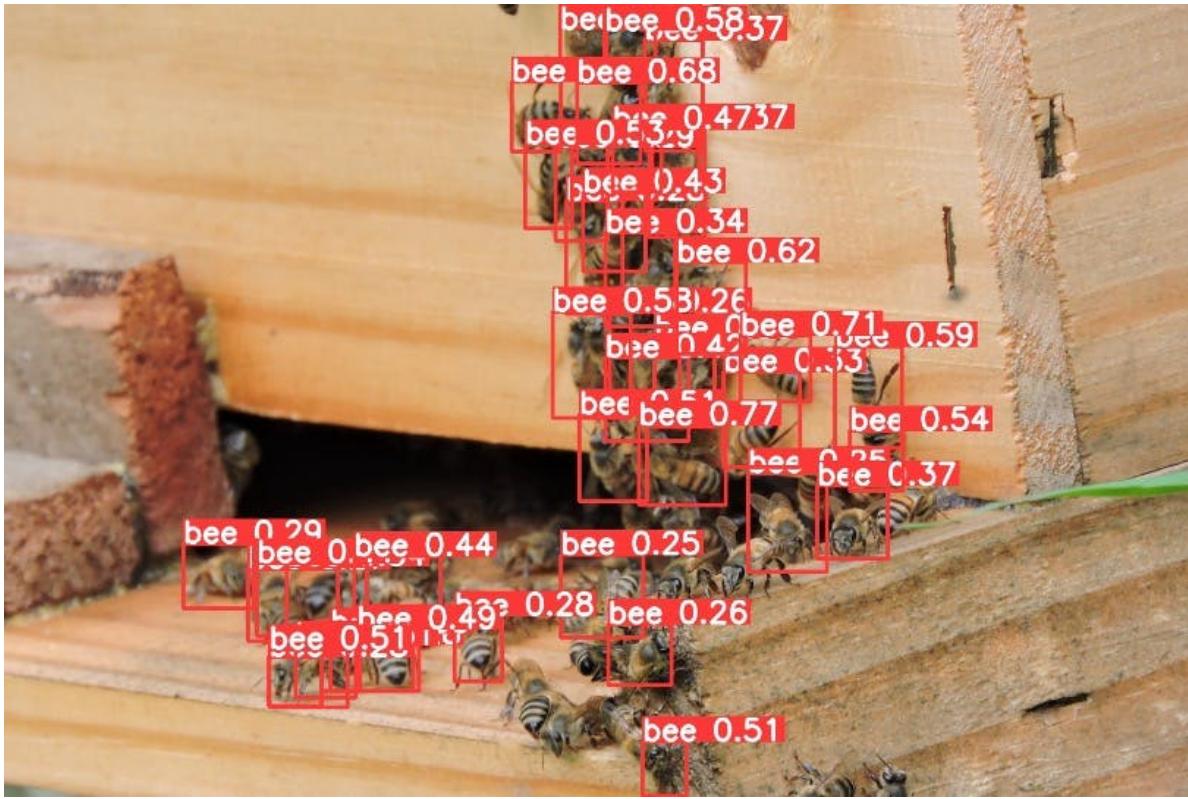
8. Now, we should perform inference on the images left aside for testing

```
!yolo task=detect mode=predict model={HOME}/runs/detect/train3/weights/best.pt conf=0.25 s
```

The inference results are saved in the folder `runs/detect/predict`. Let's see some of them:



We can also perform inference with a completely new and complex image from another beehive with a different background (the beehive of Professor Maurilio of our University). The results were great (but not perfect and with a lower confidence score). The model found 41 bees.



9. The last thing to do is export the train, validation, and test results for your Drive at Google. To do so, you should mount your 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/Bee_Project/YOLO/bees_on_hive'
```

Inference with the trained model, using the Rasp-Zero

Using the FileZilla FTP, let's transfer the `best.pt` to our Rasp-Zero (before the transfer, you may change the model name, for example, `bee_landing_640_best.pt`).

The first thing to do is convert the model to an NCNN format:

```
yolo export model=bee_landing_640_best.pt format=ncnn
```

As a result, a new converted model, `bee_landing_640_best_ncnn_model` is created in the same directory.

Let's create a folder to receive some test images (under `Documents/YOLO/`):

```
mkdir test_images
```

Using the FileZilla FTP, let's transfer a few images from the test dataset to our Rasp-Zero:



Let's use the Python Interpreter:

```
python
```

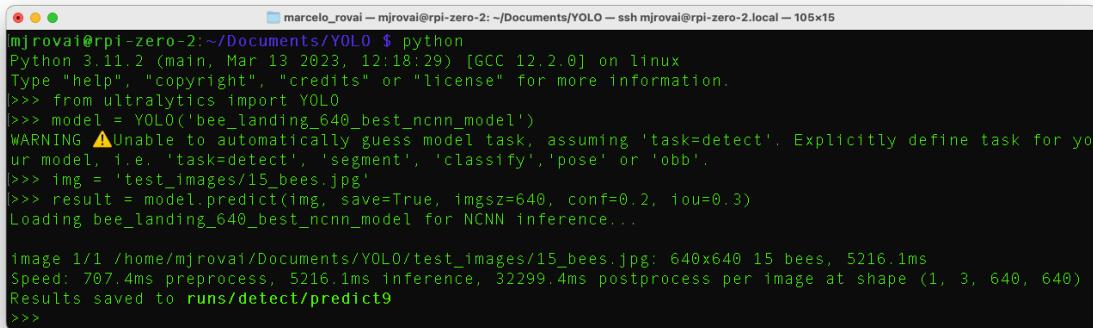
As before, we will import the YOLO library and define our converted model to detect bees:

```
from ultralytics import YOLO
model = YOLO('bee_landing_640_best_ncnn_model')
```

Now, let's define an image and call the inference (we will save the image result this time to external verification):

```
img = 'test_images/15_bees.jpg'  
result = model.predict(img, save=True, imgsz=640, conf=0.2, iou=0.3)
```

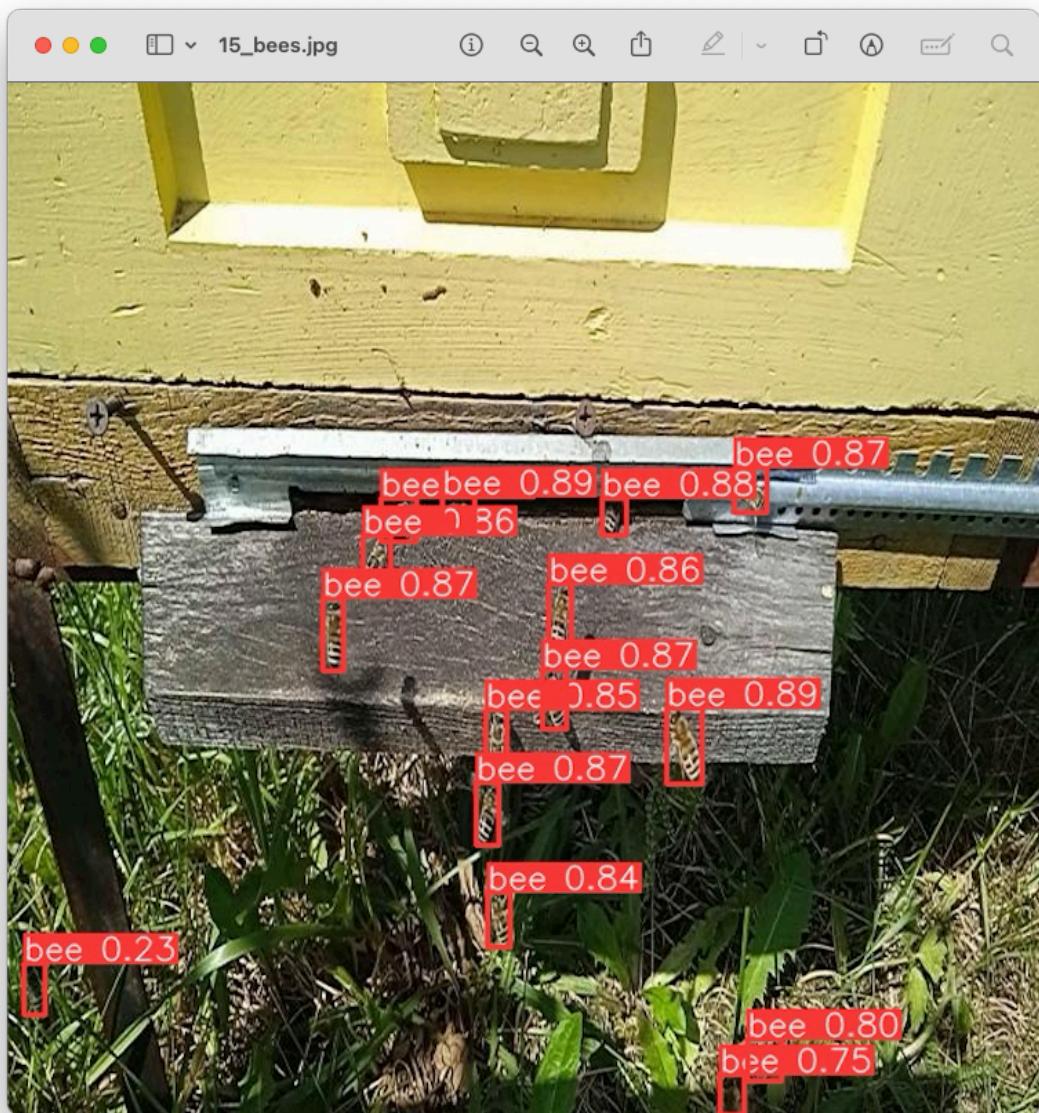
The inference result is saved on the variable `result`, and the processed image on `runs/detect/predict9`



A terminal window titled "marcelo_roversi@rpi-zero-2: ~/Documents/YOLO \$ python" displays the execution of a Python script. The script imports YOLO, loads a model, and performs inference on an image. It includes a warning about task detection and saves the results to a specified directory. The terminal shows the command line, the Python interpreter version, and the output of the script's execution.

```
marcelo_roversi@rpi-zero-2: ~/Documents/YOLO $ python  
Python 3.11.2 (main, Mar 13 2023, 12:18:29) [GCC 12.2.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> from ultralytics import YOLO  
>>> model = YOLO('bee_landing_640_best_ncnn_model')  
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 = 'test_images/15_bees.jpg'  
>>> result = model.predict(img, save=True, imgsz=640, conf=0.2, iou=0.3)  
Loading bee_landing_640_best_ncnn_model for NCNN inference...  
  
image 1/1 /home/mjroval/Documents/YOLO/test_images/15_bees.jpg: 640x640 15 bees, 5216.1ms  
Speed: 707.4ms preprocess, 5216.1ms inference, 32299.4ms postprocess per image at shape (1, 3, 640, 640)  
Results saved to runs/detect/predict9  
>>>
```

Using FileZilla FTP, we can send the inference result to our Desktop for verification:



let's go over the other images, analyzing the number of objects (bees) found:

```

marcelo_rovai — mjrovai@rpi-zero-2: ~/Documents/YOLO — ssh mjrovai@rpi-zero-2.local — 103x21
>>>
>>>
>>> img = 'test_images/6_beans.jpg'
>>> result = model.predict(img, save=False, imgs=640, conf=0.3, iou=0.3)
|
image 1/1 /home/mjrovai/Documents/YOLO/test_images/6_beans.jpg: 640x640 9 beans, 732.5ms
Speed: 19.9ms preprocess, 732.5ms inference, 13.5ms postprocess per image at shape (1, 3, 640, 640)
>>> result = model.predict(img, save=False, imgs=640, conf=0.5, iou=0.3)
|
image 1/1 /home/mjrovai/Documents/YOLO/test_images/6_beans.jpg: 640x640 7 beans, 747.8ms
Speed: 16.5ms preprocess, 747.8ms inference, 32.9ms postprocess per image at shape (1, 3, 640, 640)
>>> img = 'test_images/8_beans.jpg'
>>> result = model.predict(img, save=False, imgs=640, conf=0.5, iou=0.3)
|
image 1/1 /home/mjrovai/Documents/YOLO/test_images/8_beans.jpg: 640x640 7 beans, 728.4ms
Speed: 15.5ms preprocess, 728.4ms inference, 7.8ms postprocess per image at shape (1, 3, 640, 640)
>>> img = 'test_images/14_beans.jpg'
>>> result = model.predict(img, save=False, imgs=640, conf=0.5, iou=0.3)
|
image 1/1 /home/mjrovai/Documents/YOLO/test_images/14_beans.jpg: 640x640 13 beans, 734.4ms
Speed: 19.8ms preprocess, 734.4ms inference, 9.3ms postprocess per image at shape (1, 3, 640, 640)

```

Depending on the confidence, we can have some false positives or negatives. But in general, with a model trained based on the smaller base model of the YOLOv8 family (YOLOv8n) and also converted to NCNN, the result is pretty good, running on an Edge device such as the Rasp-Zero. Also, note that the inference latency is around 730ms.

For example, by running the inference on `Maurilio-bee.jpeg`, we can find 40 bees. During the test phase on Colab, 41 bees were found (we only missed one here.)

```

marcelo_rovai — mjrovai@rpi-zero-2: ~/Documents/YOLO — ssh mjrovai@rpi-zero-2.local — 103x6
>>> img = 'test_images/maurilio-bee.jpeg'
>>> result = model.predict(img, save=False, imgs=640, conf=0.2, iou=0.3)
|
image 1/1 /home/mjrovai/Documents/YOLO/test_images/maurilio-bee.jpeg: 640x640 40 bees, 829.2ms
Speed: 77.9ms preprocess, 829.2ms inference, 15.0ms postprocess per image at shape (1, 3, 640, 640)
>>> █

```

Considerations about the Post-Processing

Our final project should be very simple in terms of code. We will use the camera to capture an image every 10 seconds. As we did in the previous section, the captured image should be the input for the trained and converted model. We should get the number of bees for each image and save it in a database (for example, timestamp: number of bees).

We can do it with a single Python script or use a Linux system timer, like `cron`, to periodically capture images every 10 seconds and have a separate Python script to process these images as

they are saved. This method can be particularly efficient in managing system resources and can be more robust against potential delays in image processing.

Setting Up the Image Capture with cron

First, we should set up a `cron` job to use the `rpicam-jpeg` command to capture an image every 10 seconds.

1. Edit the crontab:

- Open the terminal and type `crontab -e` to edit the cron jobs.
- `cron` normally doesn't support sub-minute intervals directly, so we should use a workaround like a loop or watch for file changes.

2. Create a Bash Script (`capture.sh`):

- **Image Capture:** This bash script captures images every 10 seconds using `rpicam-jpeg`, a command that is part of the `raspijpeg` tool. This command lets us control the camera and capture JPEG images directly from the command line. This is especially useful because we are looking for a lightweight and straightforward method to capture images without the need for additional libraries like `Picamera` or external software. The script also saves the captured image with a timestamp.

```
#!/bin/bash
# Script to capture an image every 10 seconds

while true
do
    DATE=$(date +"%Y-%m-%d_%H%M%S")
    rpicam-jpeg --output test_images/$DATE.jpg --width 640 --height 640
    sleep 10
done
```

- We should make the script executable with `chmod +x capture.sh`.
- The script must start at boot or use a `@reboot` entry in `cron` to start it automatically.

Setting Up the Python Script for Inference

Image Processing: The Python script continuously monitors the designated directory for new images, processes each new image using the YOLOv8 model, updates the database with the count of detected bees, and optionally deletes the image to conserve disk space.

Database Updates: The results, along with the timestamps, are saved in an SQLite database. For that, a simple option is to use [sqlite3](#).

In short, we need to write a script that continuously monitors the directory for new images, processes them using a YOLO model, and then saves the results to a SQLite database. Here's how we can create and make the script executable:

```
#!/usr/bin/env python3
import os
import time
import sqlite3
from datetime import datetime
from ultralytics import YOLO

# Constants and paths
IMAGES_DIR = 'test_images/'
MODEL_PATH = 'bee_landing_640_best_ncnn_model'
DB_PATH = 'bee_count.db'

def setup_database():
    """ Establishes a database connection and creates the table if it doesn't exist. """
    conn = sqlite3.connect(DB_PATH)
    cursor = conn.cursor()
    cursor.execute('''
        CREATE TABLE IF NOT EXISTS bee_counts
        (timestamp TEXT, count INTEGER)
    ''')
    conn.commit()
    return conn

def process_image(image_path, model, conn):
    """ Processes an image to detect objects and logs the count to the database. """
    result = model.predict(image_path, save=False, imgsz=640, conf=0.2, iou=0.3, verbose=False)
    num_beans = len(result[0].boxes.cls)
    timestamp = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
    cursor = conn.cursor()
    cursor.execute("INSERT INTO bee_counts (timestamp, count) VALUES (?, ?)", (timestamp,
```

```

    conn.commit()
    print(f'Processed {image_path}: Number of bees detected = {num_bees}')

def monitor_directory(model, conn):
    """ Monitors the directory for new images and processes them as they appear. """
    processed_files = set()
    while True:
        try:
            files = set(os.listdir(IMAGES_DIR))
            new_files = files - processed_files
            for file in new_files:
                if file.endswith('.jpg'):
                    full_path = os.path.join(IMAGES_DIR, file)
                    process_image(full_path, model, conn)
                    processed_files.add(file)
            time.sleep(1) # Check every second
        except KeyboardInterrupt:
            print("Stopping...")
            break

def main():
    conn = setup_database()
    model = YOLO(MODEL_PATH)
    monitor_directory(model, conn)
    conn.close()

if __name__ == "__main__":
    main()

```

The python script must be executable, for that:

1. **Save the script:** For example, as `process_images.py`.
2. **Change file permissions** to make it executable:

```
chmod +x process_images.py
```

3. **Run the script** directly from the command line:

```
./process_images.py
```

We should consider keeping the script running even after closing the terminal; for that, we can use `nohup` or `screen`:

```
nohup ./process_images.py &
```

or

```
screen -S bee_monitor  
./process_images.py
```

Note that we are capturing images with their own timestamp and then log a separate timestamp for when the inference results are saved to the database. This approach can be beneficial for the following reasons:

1. Accuracy in Data Logging:

- **Capture Timestamp:** The timestamp associated with each image capture represents the exact moment the image was taken. This is crucial for applications where precise timing of events (like bee activity) is important for analysis.
- **Inference Timestamp:** This timestamp indicates when the image was processed and the results were recorded in the database. This can differ from the capture time due to processing delays or if the image processing is batched or queued.

2. Performance Monitoring:

- Having separate timestamps allows us to monitor the performance and efficiency of your image processing pipeline. We can measure the delay between image capture and result logging, which helps optimize the system for real-time processing needs.

3. Troubleshooting and Audit:

- Separate timestamps provide a better audit trail and troubleshooting data. If there are issues with the image processing or data recording, having distinct timestamps can help isolate whether delays or problems occurred during capture, processing, or logging.

Script For Reading the SQLite Database

Here is an example of a code to retrieve the data from the database:

```
#!/usr/bin/env python3  
import sqlite3  
  
def main():  
    db_path = 'bee_count.db'  
    conn = sqlite3.connect(db_path)  
    cursor = conn.cursor()
```

```

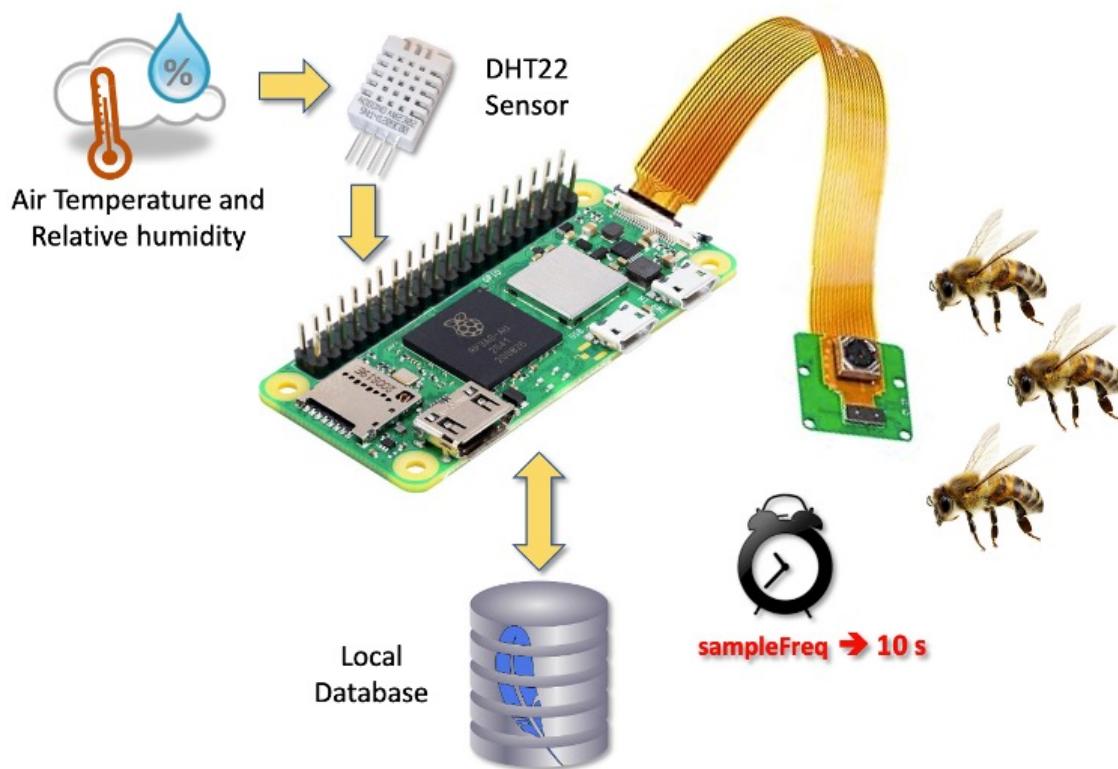
query = "SELECT * FROM bee_counts"
cursor.execute(query)
data = cursor.fetchall()
for row in data:
    print(f"Timestamp: {row[0]}, Number of bees: {row[1]}")
conn.close()

if __name__ == "__main__":
    main()

```

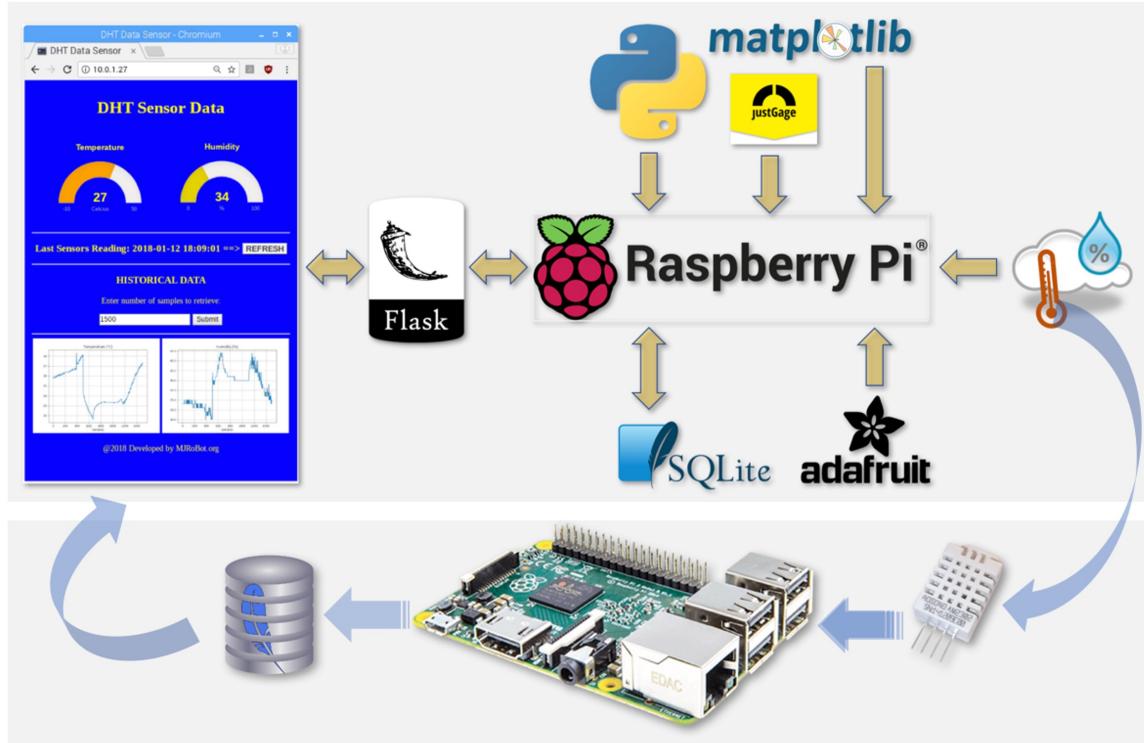
Adding Environment data

Besides bee counting, environmental data, such as temperature and humidity, are essential for monitoring the bee-hive health. Using a Rasp-Zero, it is straightforward to add a digital sensor such as the DHT-22 to get this data.



Environmental data will be part of our final project. If you want to know more about connecting sensors to a Raspberry Pi and, even more, how to save the data to a local database and

send it to the web, follow this tutorial: [From Data to Graph: A Web Journey With Flask and SQLite](#).



Conclusion

In this tutorial, we have thoroughly explored integrating the YOLOv8 model with a Raspberry Pi Zero 2W to address the practical and pressing task of counting (or better, “estimating”) bees at a beehive entrance. Our project underscores the robust capability of embedding advanced machine learning technologies within compact edge computing devices, highlighting their potential impact on environmental monitoring and ecological studies.

This tutorial provides a step-by-step guide to the practical deployment of the YOLOv8 model. We demonstrate a tangible example of a real-world application by optimizing it for edge computing in terms of efficiency and processing speed (using NCNN format). This not only serves as a functional solution but also as an instructional tool for similar projects.

The technical insights and methodologies shared in this tutorial are the basis for the complete work to be developed at our university in the future. We envision further development, such as integrating additional environmental sensing capabilities and refining the model's accuracy and processing efficiency. Implementing alternative energy solutions like the proposed solar power setup will expand the project's sustainability and applicability in remote or underserved locations.

Resources

The Dataset paper, Notebooks, and PDF version are in the [Project repository](#).

Small Language Models (SLM)

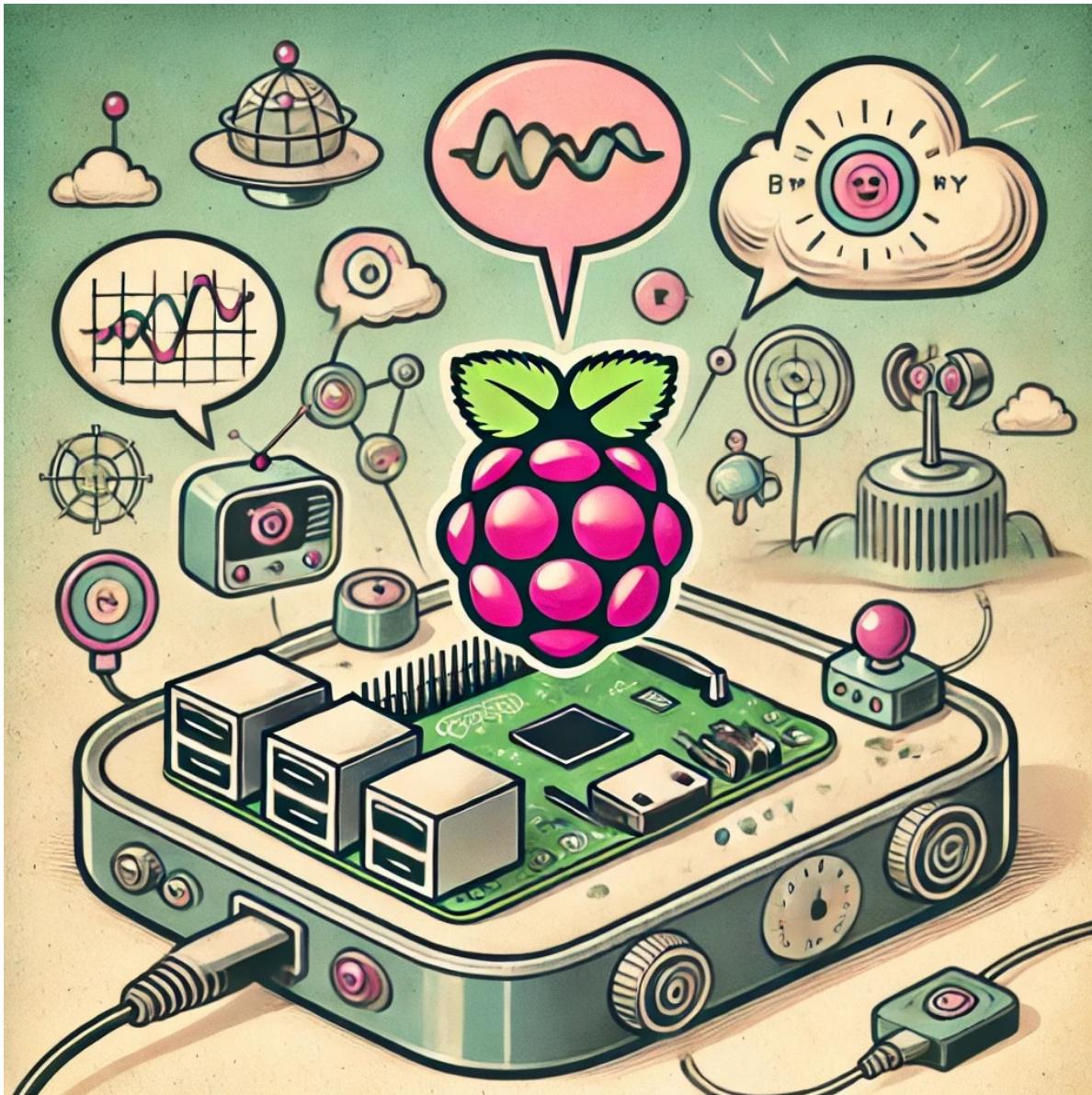


Figure 5: *DALL·E prompt* - A 1950s-style cartoon illustration showing a Raspberry Pi running a small language model at the edge. The Raspberry Pi is stylized in a retro-futuristic way with rounded edges and chrome accents, connected to playful cartoonish sensors and devices. Speech bubbles are floating 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.

Introduction

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).



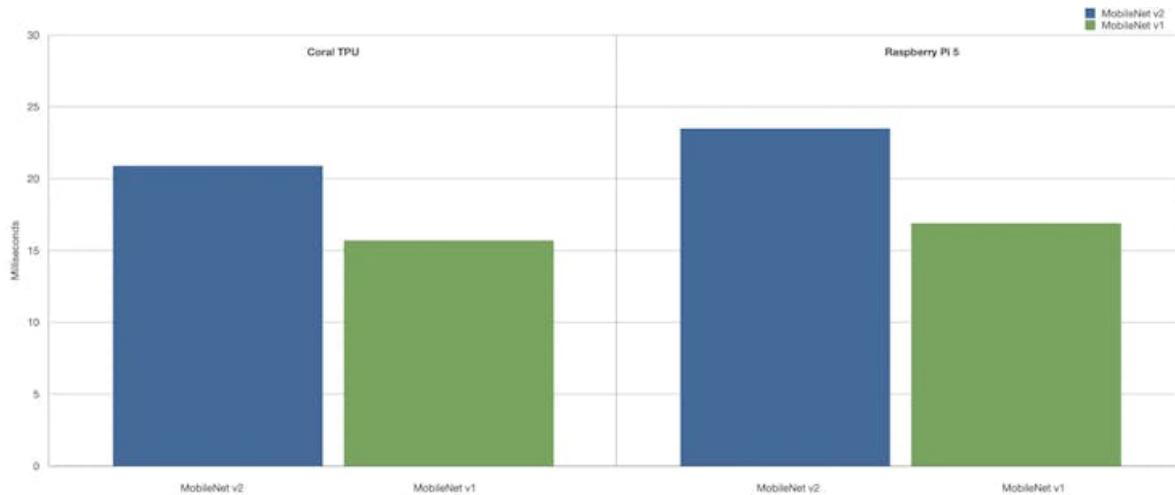
Setup

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 2.4GHz 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 4GB and 8GB 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.

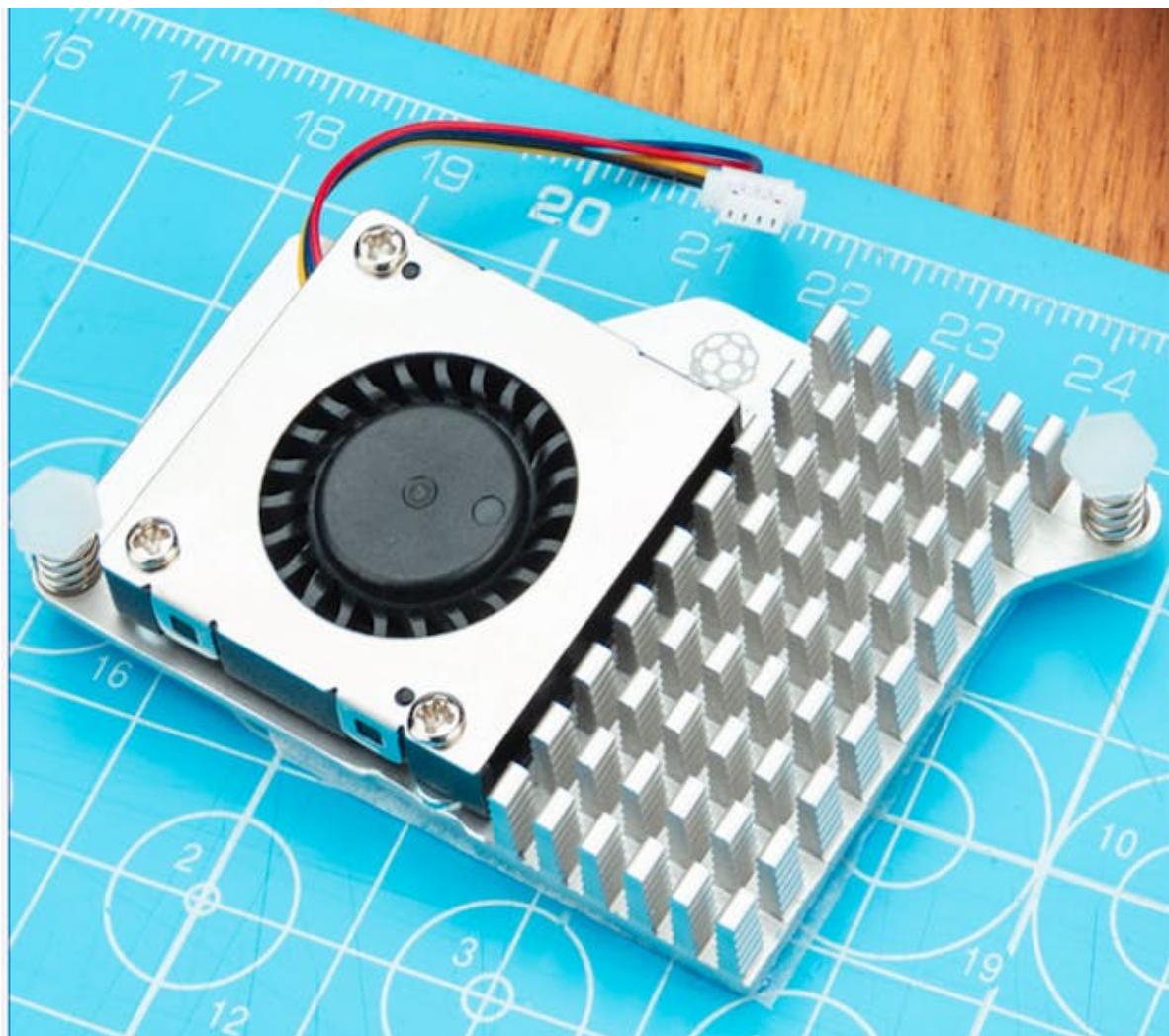
By the way, as [Alasdair Allan](#) discussed, inferencing directly on the Raspberry Pi 5 CPU—with no GPU acceleration—is now on par with the performance of the Coral TPU.



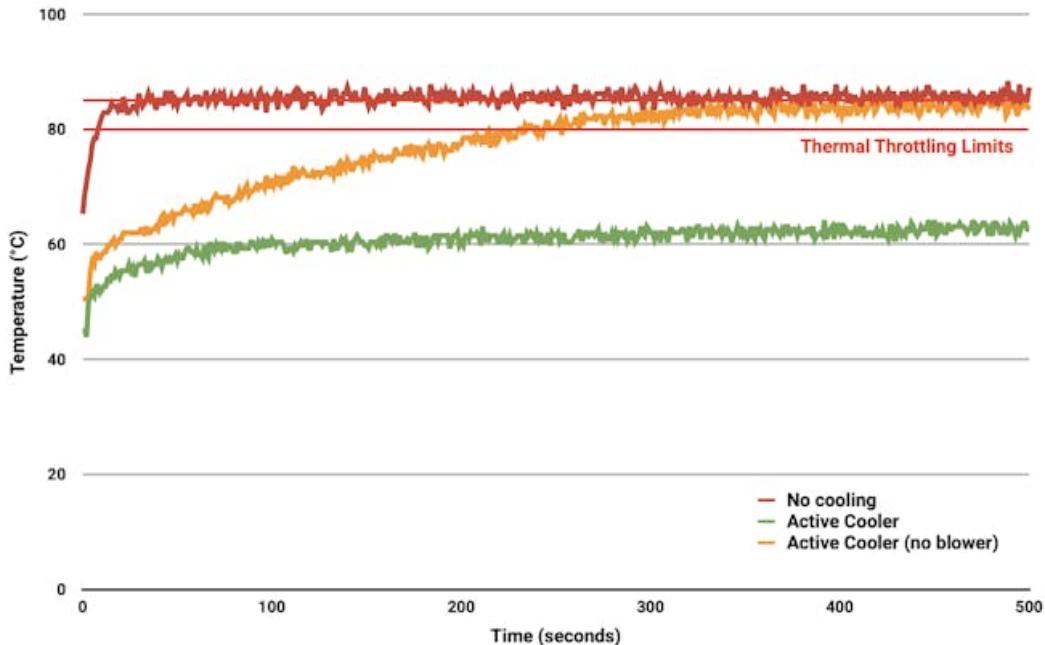
For more info, please see the complete article: [Benchmarking TensorFlow and TensorFlow Lite on Raspberry Pi 5](#).

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 heatsink 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 *multi-modal 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.

Certainly. Let's define open and closed models in the context of AI and language models:

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.

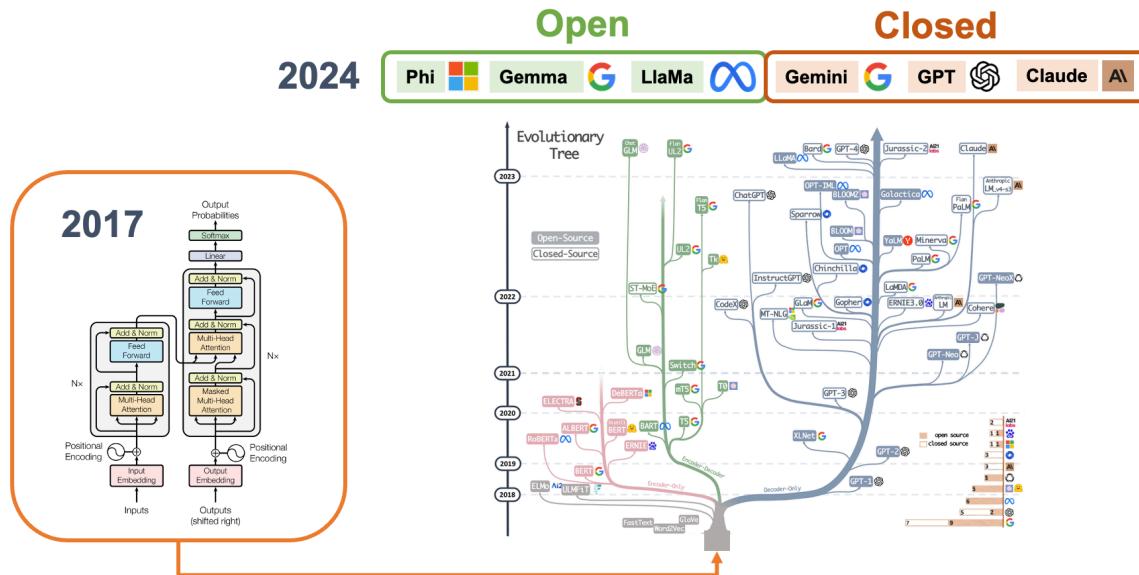


Figure 6: Adapted from <https://arxiv.org/pdf/2304.13712>

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 7: 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 — mjrovai@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.

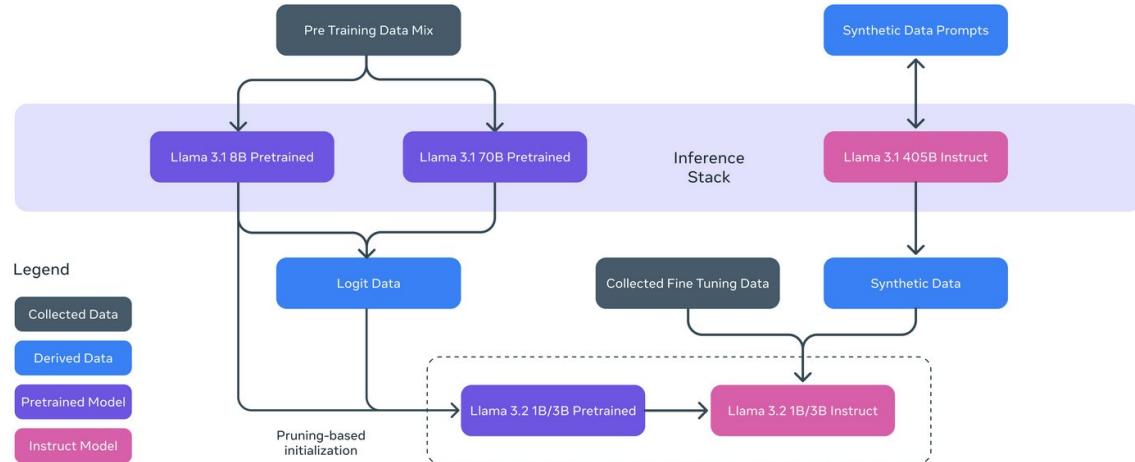
Meta Llama 3.2 1B/3B



Let's install and run our first small language model, [Llama 3.2 1B](#) (and 3B). The Meta Llama, 3.2 collections of multilingual large language models (LLMs), is a collection of pre-trained and instruction-tuned generative models in 1B and 3B sizes (text in/text out). The Llama 3.2 instruction-tuned text-only models are optimized for multilingual dialogue use cases, including agentic retrieval and summarization tasks.

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 2GB, respectively. Its context window is 131,072 tokens.

1B & 3B Pruning & Distillation



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_rovai - mjrovai@raspi-5: ~ ssh mjrovai@192.168.4.209 - 79x26
(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 4f659a1e86d7... 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:

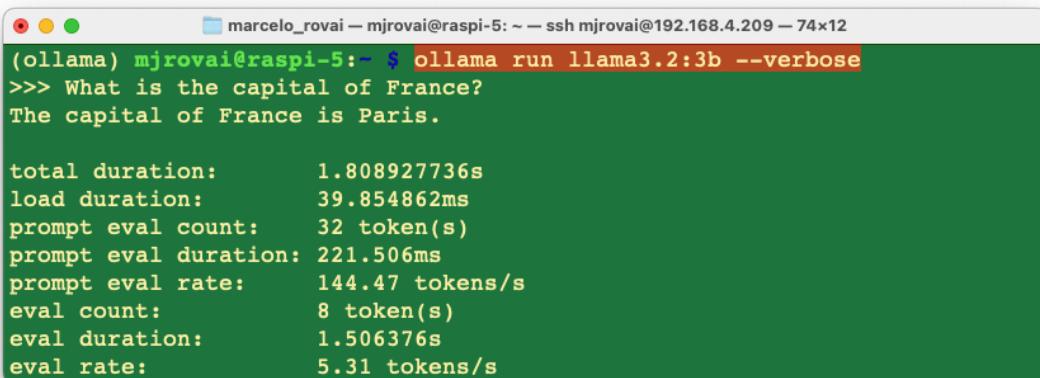
- **Total Duration (2.620170326s)**: This is the complete time taken from the start of the command to the completion of the response. It encompasses loading the model, processing the input prompt, and generating the response.
- **Load Duration (39.947908ms)**: This duration indicates the time to load 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 (1.644773s)**: 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 (889.941ms):** 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 — 74x12
(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

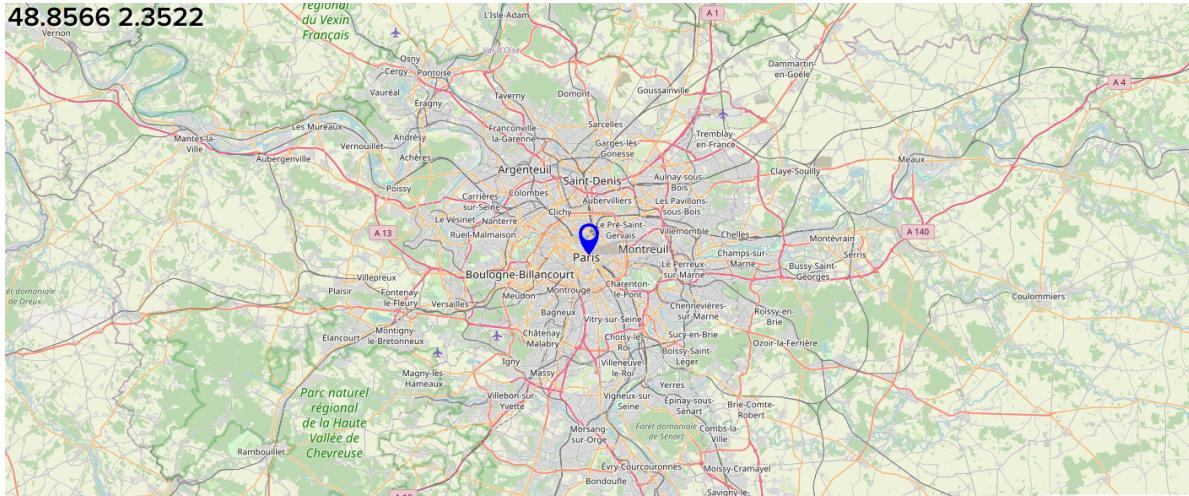
```
>>> What is the distance between Paris and Santiago, Chile?
```

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
```

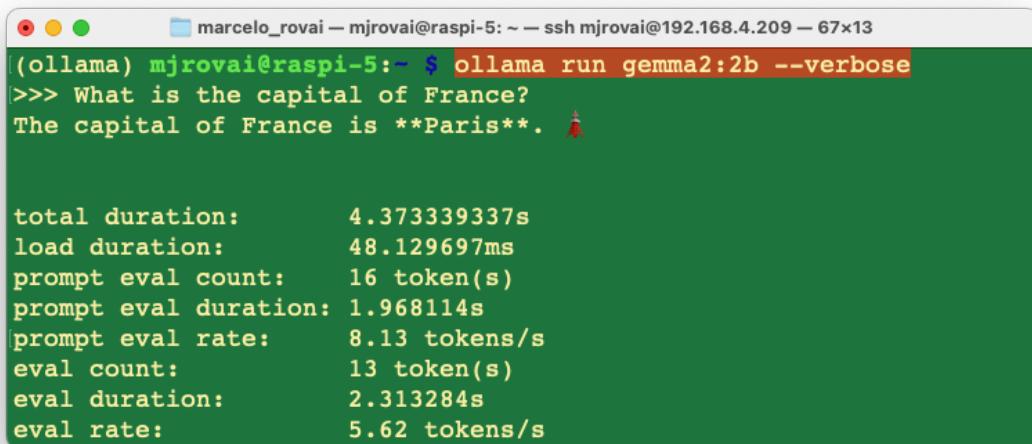
Running the model with the command before, we should have the Ollama prompt available for us to input a question and start chatting with the LLM model; for example,

```
>>> What is the capital of France?
```

Almost immediately, we get the correct answer:

```
The capital of France is **Paris**.
```

And it's statistics.



```
marcelo_rovai — mjrovai@raspi-5: ~ — ssh mjrovai@192.168.4.209 — 67x13
(ollama) mjrovai@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 Lama 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 1.4GB (higher performance/lower quality) to 16-bit quantization (`fp-16`) of 7.6GB (lower performance/higher quality).

Let's run the 4-bit quantization (`Q4_0`), which will need 2.2GB 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:

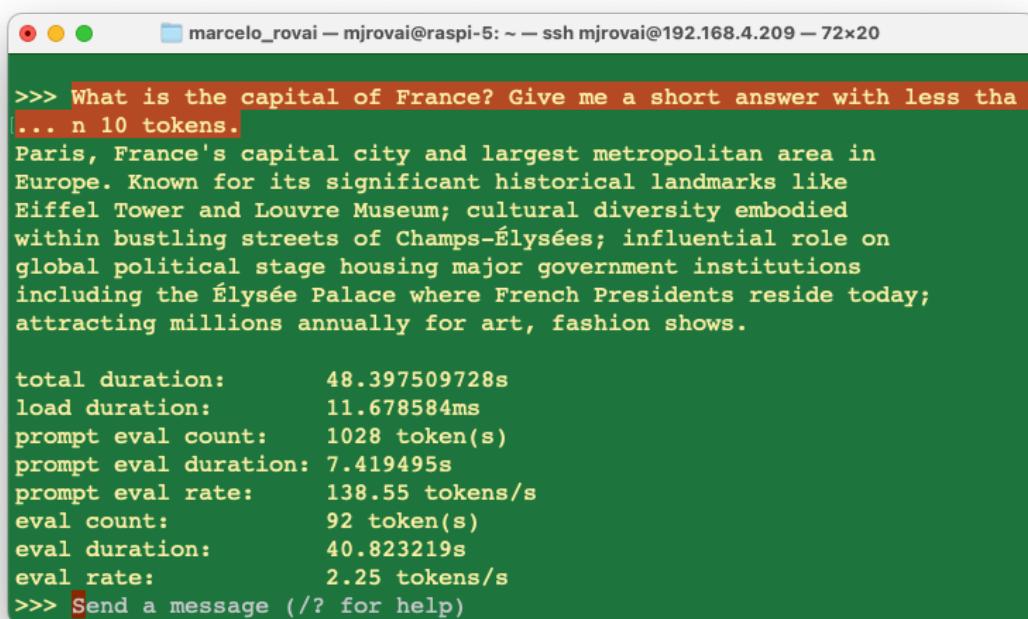
```
>>> What is the capital of France?
```

The capital of France is Paris. It's extricates 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, 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 every day,

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 30s, 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
```

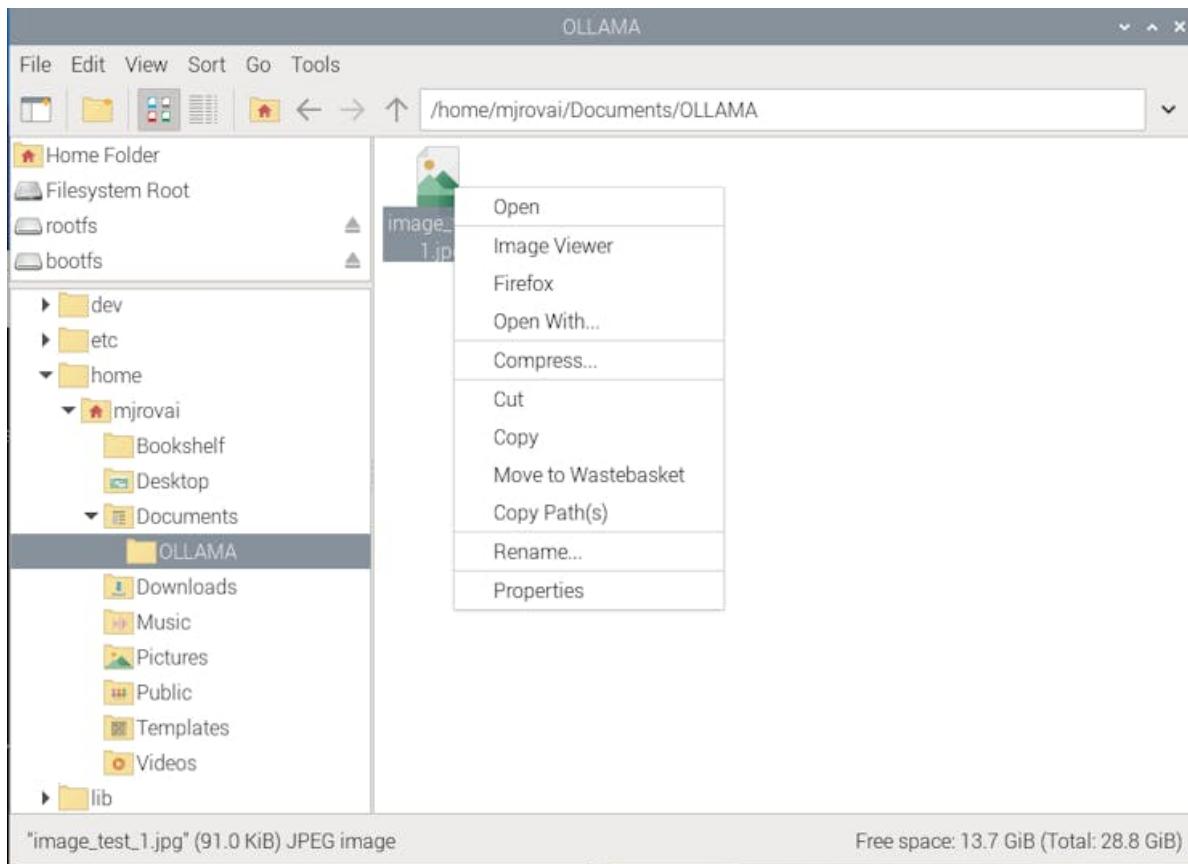
Let's download a 640x320 image from the internet, for example (Wikipedia: [Paris, France](#)):



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.

```

marcelo_rovai — mjrovai@raspi-5: ~/Documents/OLLAMA — ssh mjrovai@192.168.4.209 — 84x36
(ollama) mjrovai@raspi-5:~/Documents/OLLAMA $ pwd
/home/mjrovai/Documents/OLLAMA
(ollama) mjrovai@raspi-5:~/Documents/OLLAMA $ ollama run llava-phi3:3.8b --verbose
>>> Describe the image /home/mjrovai/Documents/OLLAMA/image_test_1.jpg
Added image '/home/mjrovai/Documents/OLLAMA/image_test_1.jpg'
The image captures a breathtaking view of Paris, France. The cityscape is
dotted with buildings in various shades of white and gray, interspersed with
lush green trees that add a touch of nature to the urban setting.

In the heart of the scene stands the Eiffel Tower, an iconic symbol of Paris,
its iron lattice structure reaching up into the clear blue sky. The tower's
distinctive silhouette is unmistakable against the backdrop of the sky, which
is a vibrant shade of blue with just a few clouds scattered across it.

The Seine River gracefully winds its way through the city, bordered by an
array of buildings on both sides. The river is lined with several bridges that
connect different parts of the city and facilitate movement for pedestrians
and vehicles alike.

Above all these elements, a few birds can be seen soaring freely in the sky,
their presence adding life to the scene. Their flight paths crisscross over
the river and the buildings, creating dynamic patterns that draw the eye.

Overall, this image presents a beautiful daytime snapshot of Paris - its
architectural marvels, natural beauty, and bustling city life coexisting in
harmony.

total duration:      3m55.972199346s
load duration:     16.198011ms
prompt eval count:   1 token(s)
prompt eval duration: 2m19.561783s
prompt eval rate:    0.01 tokens/s
eval count:         276 token(s)
eval duration:      1m36.330959s
eval rate:          2.87 tokens/s
>>> Send a message (/? for help)

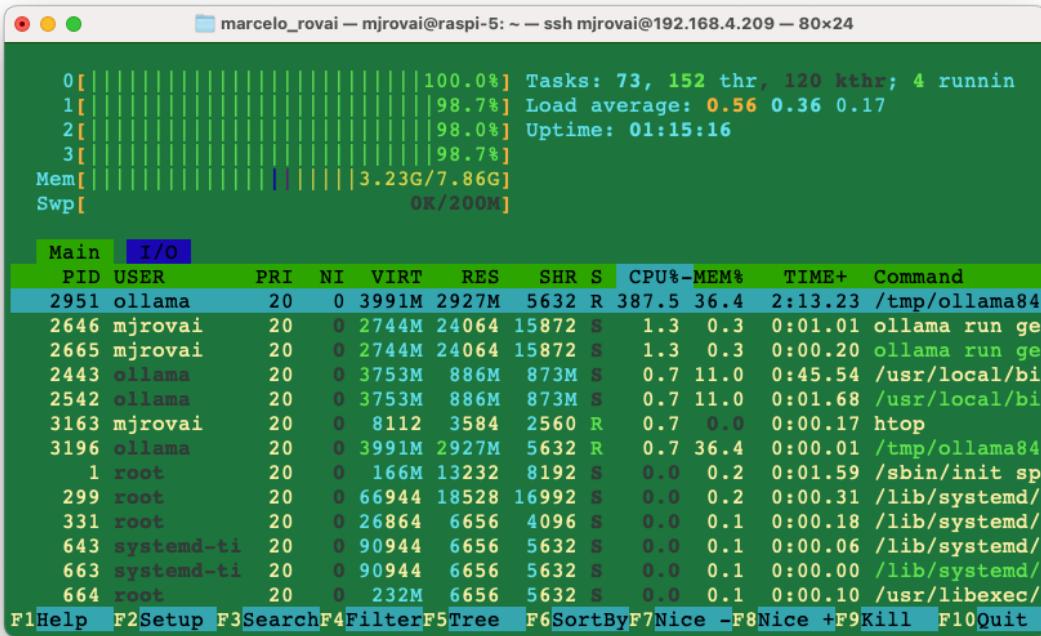
```

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 3.24GB. Exiting Ollama, the memory goes down to around 377MB (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:

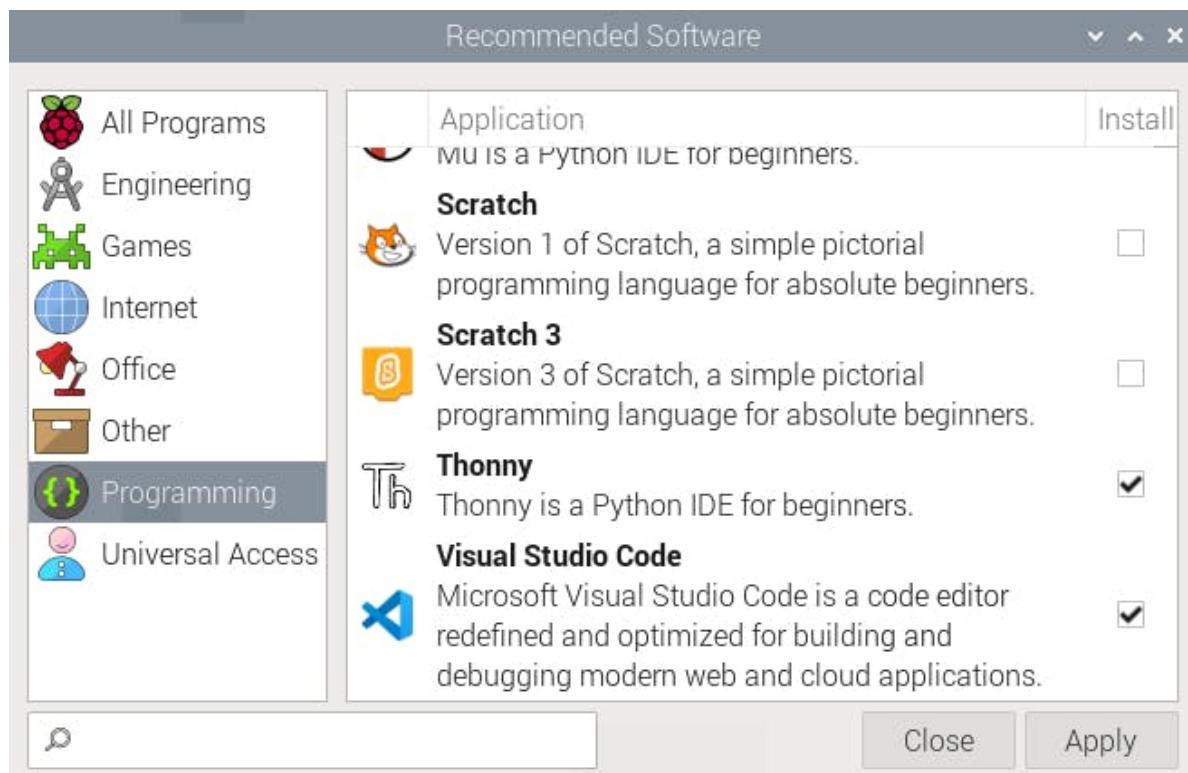
https://www.youtube.com/embed/_4K20tOsXK8

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:

```

marcelo_rovai — mjrovai@raspi-5: ~ ssh mjrovai@192.168.4.209 — 130x31
(oollama) mjrovai@raspi-5:~$ jupyter notebook --ip=192.168.4.209 --no-browser
[I 2024-09-25 15:25:03.768 ServerApp] jupyter_lsp | extension was successfully linked.
[I 2024-09-25 15:25:03.772 ServerApp] jupyter_server_terminals | extension was successfully linked.
[I 2024-09-25 15:25:03.776 ServerApp] jupyterlab | extension was successfully linked.
[I 2024-09-25 15:25:03.780 ServerApp] notebook | extension was successfully linked.
[I 2024-09-25 15:25:04.022 ServerApp] notebook_shim | extension was successfully linked.
[I 2024-09-25 15:25:04.034 ServerApp] notebook_shim | extension was successfully loaded.
[I 2024-09-25 15:25:04.036 ServerApp] jupyter_lsp | extension was successfully loaded.
[I 2024-09-25 15:25:04.037 ServerApp] jupyter_server_terminals | extension was successfully loaded.
[I 2024-09-25 15:25:04.038 LabApp] JupyterLab extension loaded from /home/mjrovai/oollama/lib/python3.11/site-packages/jupyterlab
[I 2024-09-25 15:25:04.038 LabApp] JupyterLab application directory is /home/mjrovai/oollama/share/jupyter/lab
[I 2024-09-25 15:25:04.039 LabApp] Extension Manager is 'pypi'.
[I 2024-09-25 15:25:04.082 ServerApp] jupyterlab | extension was successfully loaded.
[I 2024-09-25 15:25:04.085 ServerApp] notebook | extension was successfully loaded.
[I 2024-09-25 15:25:04.085 ServerApp] Serving notebooks from local directory: /home/mjrovai
[I 2024-09-25 15:25:04.085 ServerApp] Jupyter Server 2.14.2 is running at:
[I 2024-09-25 15:25:04.085 ServerApp] http://192.168.4.209:8888/tree?token=79a989d699951f61d357cd5aa1146d350eaf3ed1471a422
[I 2024-09-25 15:25:04.085 ServerApp] http://127.0.0.1:8888/tree?token=79a989d699951f61d357cd5aa1146d350eaf3ed1471a422
[I 2024-09-25 15:25:04.085 ServerApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 2024-09-25 15:25:04.087 ServerApp]

To access the server, open this file in a browser:
file:///home/mjrovai/.local/share/jupyter/runtime/jpserver-10313-open.html
Or copy and paste one of these URLs:
http://192.168.4.209:8888/tree?token=79a989d699951f61d357cd5aa1146d350eaf3ed1471a422
http://127.0.0.1:8888/tree?token=79a989d699951f61d357cd5aa1146d350eaf3ed1471a422
[I 2024-09-25 15:25:04.098 ServerApp] Skipped non-installed server(s): bash-language-server, dockerfile-language-server-nodejs, javascript-typescript-langsServer, jedi-language-server, julia-language-server, pyright, python-language-server, python-lsp-server, r-langsServer, sql-language-server, texlab, typescript-language-server, unified-language-server, vscode-css-langsServer-bin, vscode-html-langsServer-bin, vscode-json-langsServer-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': '8ccf136fdd5298f3ffe2d69862750ea7fb56555fa4d5b18c04e3fa4d82ee09d7',
'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 19.38 seconds.
- **prompt_eval_duration:** The time taken to evaluate the prompt in nanoseconds. Around 1.9.0 seconds.
- **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: {(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'][0]['content']  
print(f"\n{resp_1}")  
print(f"\n [INFO] Total Duration: {(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: {(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: {(response['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: {(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 see that, with the model's ("response") answer to a variable, we can efficiently work with it, integrating it into real-world projects. However, a big problem is that the model can respond differently to the same prompt. Let's say that what we want, as the model's response in the last examples, is only the name of a given country's capital and its coordinates,

nothing more, even with very verbose models such as the Microsoft Phi. We can use the `Ollama` function's calling to guarantee the same answers, which is perfectly compatible with 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.

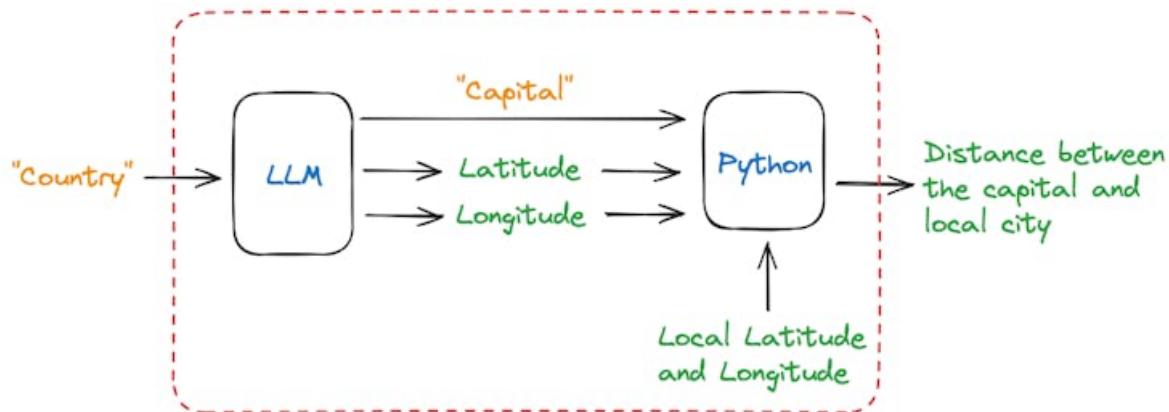
For instance, instead of merely responding to a query about the weather, an LLM can call a weather API to fetch the current conditions and provide accurate, up-to-date information. This capability enhances the relevance and accuracy of the model’s responses and makes it a powerful tool for driving workflows 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/eHfMCtlsb1o>

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 (IA), 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))} :,\n    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
File Edit Tabs Help
mjrovai@rpi-5:~/Documents/OLLAMA $ 
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 $ 
```

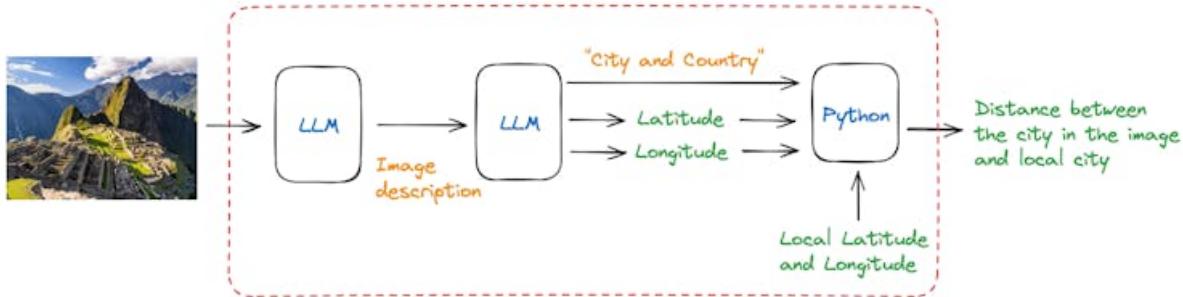
And the calculations are pretty good!



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"\n The image shows {resp.city}, with lat:{round(resp.lat, 2)} and \

```

```

    long: {round(resp.lon, 2)}, located in {resp.country} and about \
        {int(round(distance, -1))}, kilometers 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}), 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.

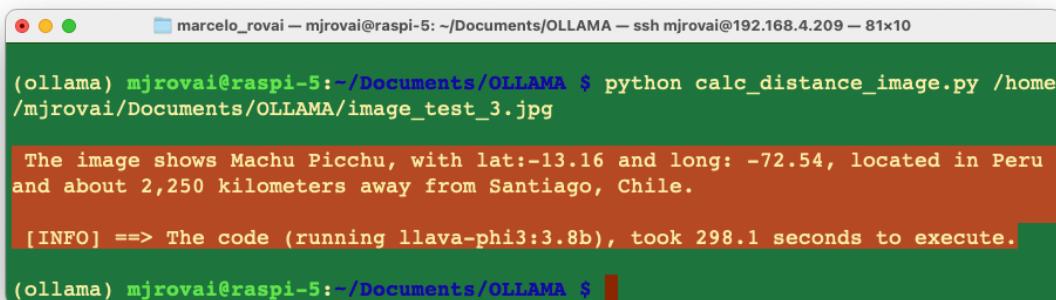
[INFO] ==> The code (running llava-phi3:3.8b), took 491.3 seconds to execute.

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.

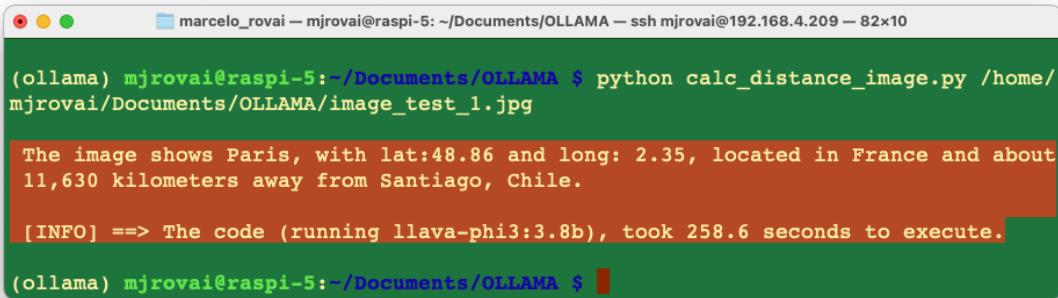


```
(ollama) mjrovai@raspi-5:~/Documents/OLLAMA $ python calc_distance_image.py /home/mjrovai/Documents/OLLAMA/image_test_3.jpg
The image shows Machu Picchu, with lat:-13.16 and long: -72.54, located in Peru
and about 2,250 kilometers away from Santiago, Chile.

[INFO] ==> The code (running llava-phi3:3.8b), took 298.1 seconds to execute.

(ollama) mjrovai@raspi-5:~/Documents/OLLAMA $
```

How about Paris?



```

marcelo_rovai — 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).

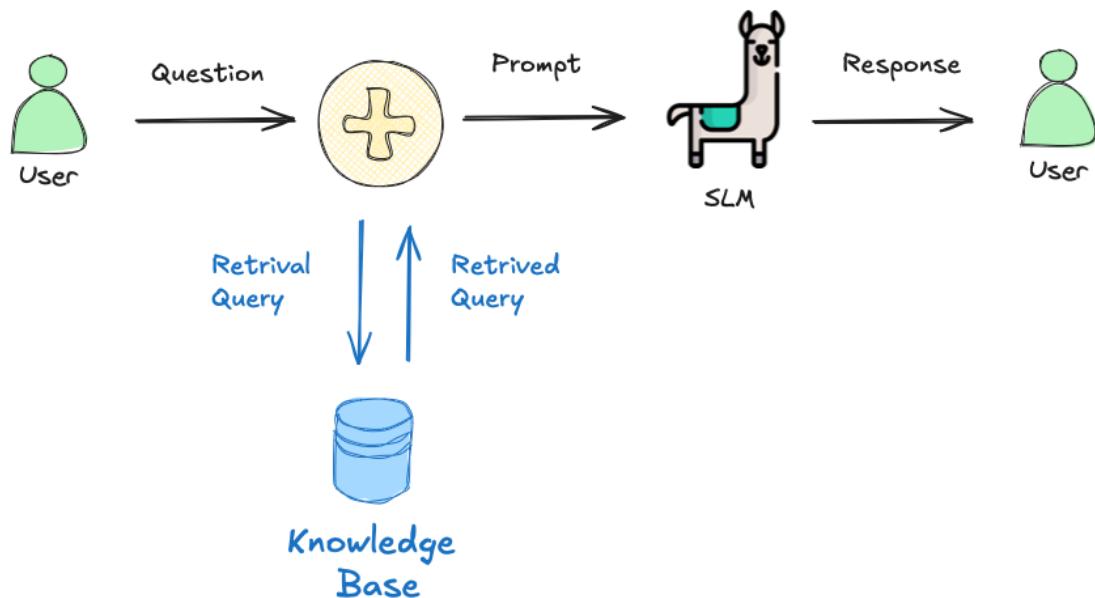
- **Fine-tuning**, while more resource-intensive, offers a way to specialize LLMs for particular domains or tasks. This process involves further training the model on carefully curated datasets, allowing it to adapt its vast general knowledge to specific applications. Fine-tuning can lead to substantial 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 instalation:

```
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 \
quadrispiciata), 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:

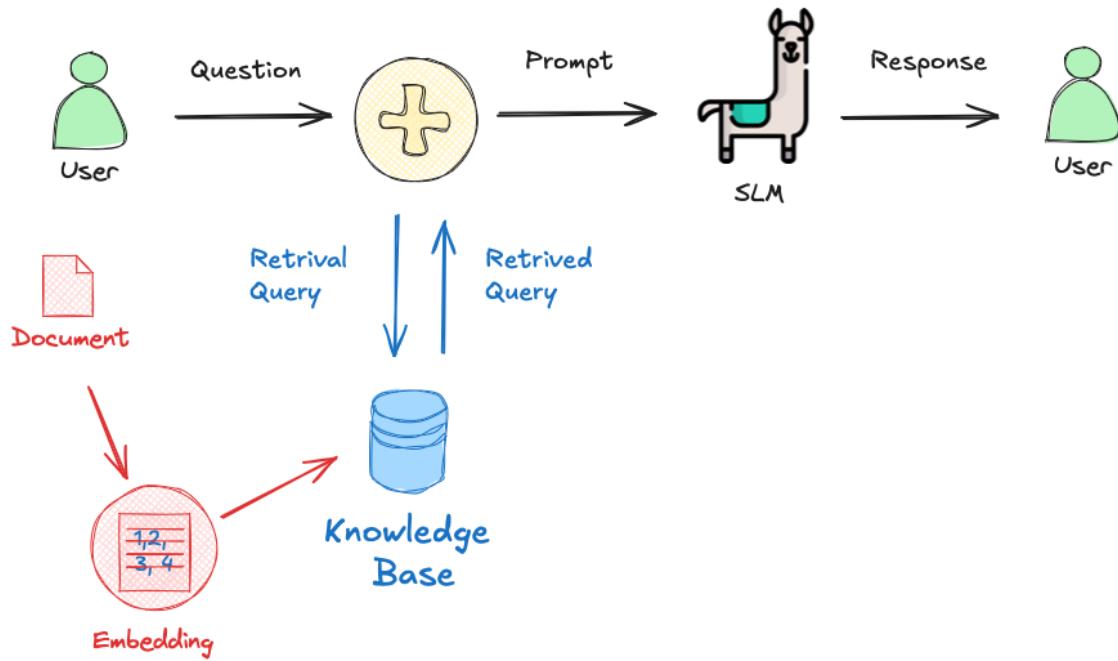
```

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 more 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}. Respond to this prompt: {prompt}",
```

Answer Generation: The augmented prompt is then fed into a language model, in this case, the llama3.2:3b 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}. 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_beans(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}. 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}, 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 Brasil?"  
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: mandaçaia (*Melipona quadrifasciata*), mandaguari (*Scaptotrigona postica*) e jataí-amarela (*Tetragonisca angustula*). Além disso, o Brasil é conhecido por ter mais de 300 espécies diferentes de ab

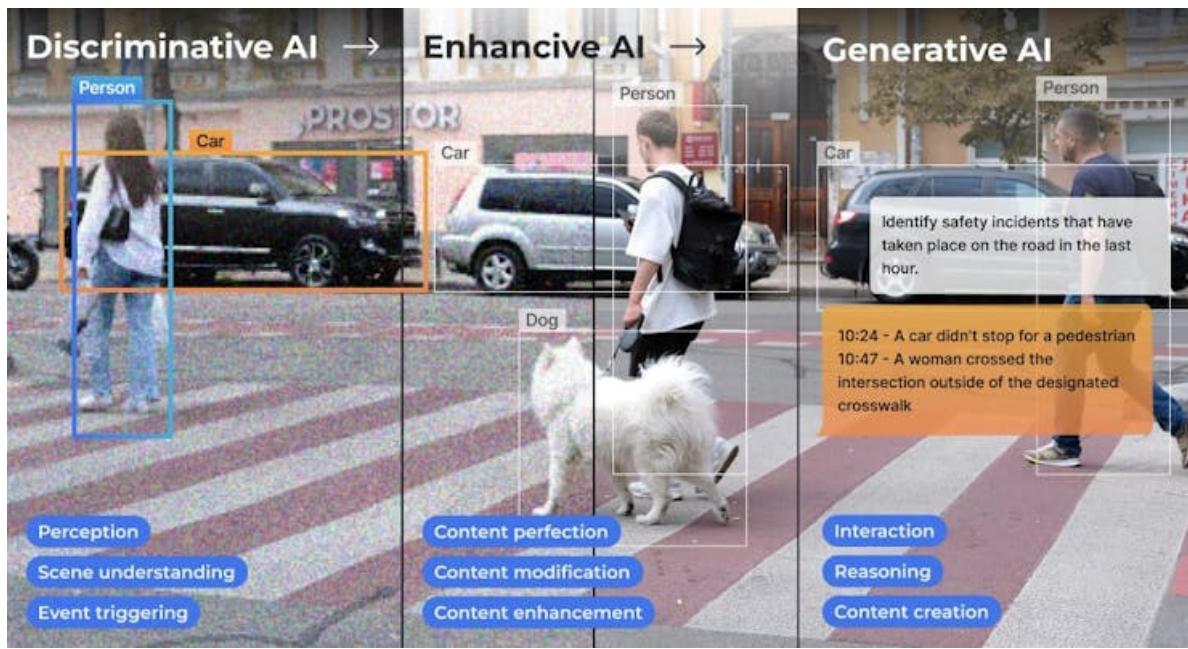
```
[INFO] ==> The code for model: llama3.2:3b, took 54.6s to generate the answer.
```

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.



Conclusion

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 at the Edge

We will learn Vision-Language Models across tasks such as captioning, object detection, grounding, and segmentation on a Raspberry Pi.

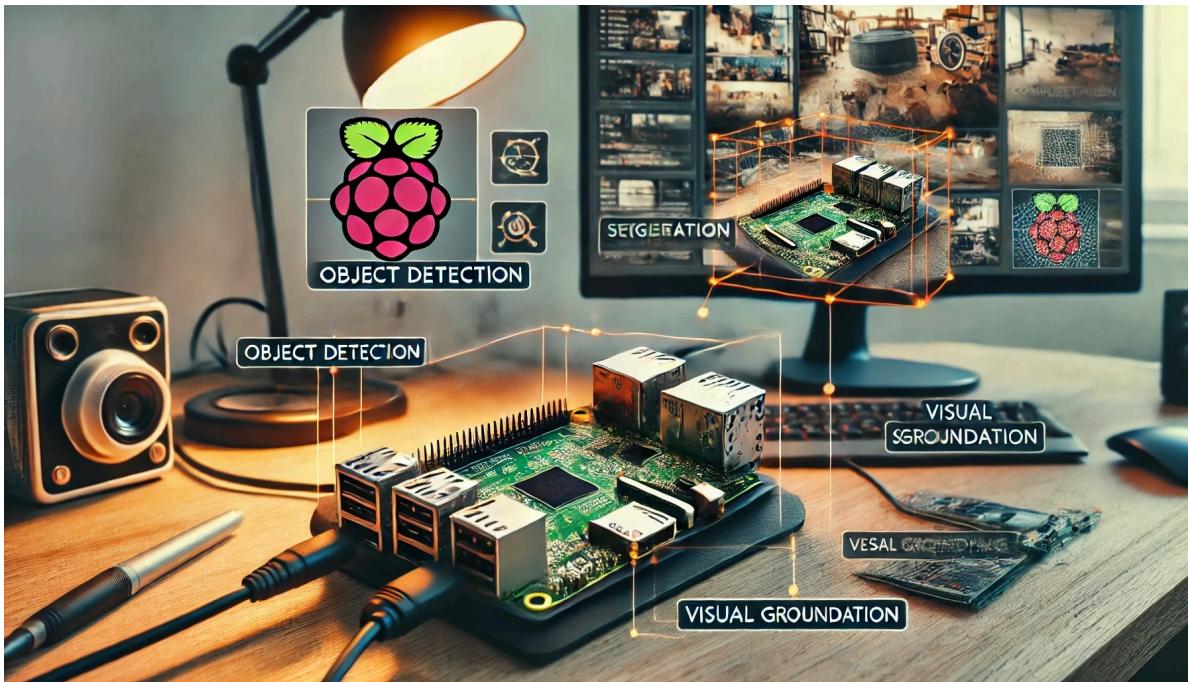


Figure 8: *DALL·E prompt - A Raspberry Pi setup featuring vision tasks.* The image shows a Raspberry Pi connected to a camera, with various computer vision tasks displayed visually around it, including object detection, image captioning, segmentation, and visual grounding. The Raspberry Pi is placed 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.

In this hands-on lab, we will continuously explore AI applications at the Edge, going 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.

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

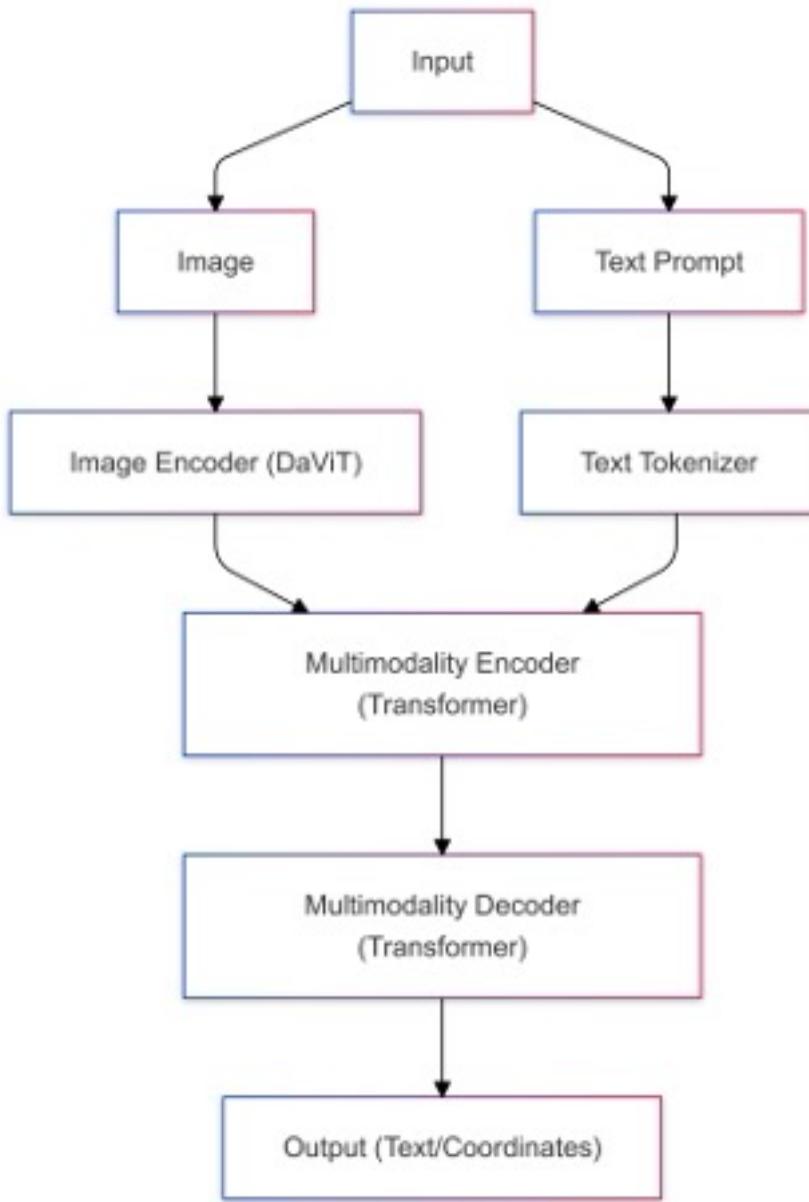
Visual grounding involves linking textual descriptions to specific regions within an image. This enables the model to understand where particular objects or entities described in a prompt are in the image. For example, if the prompt is “a red car,” the model will identify and highlight the region where 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**.



- **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.

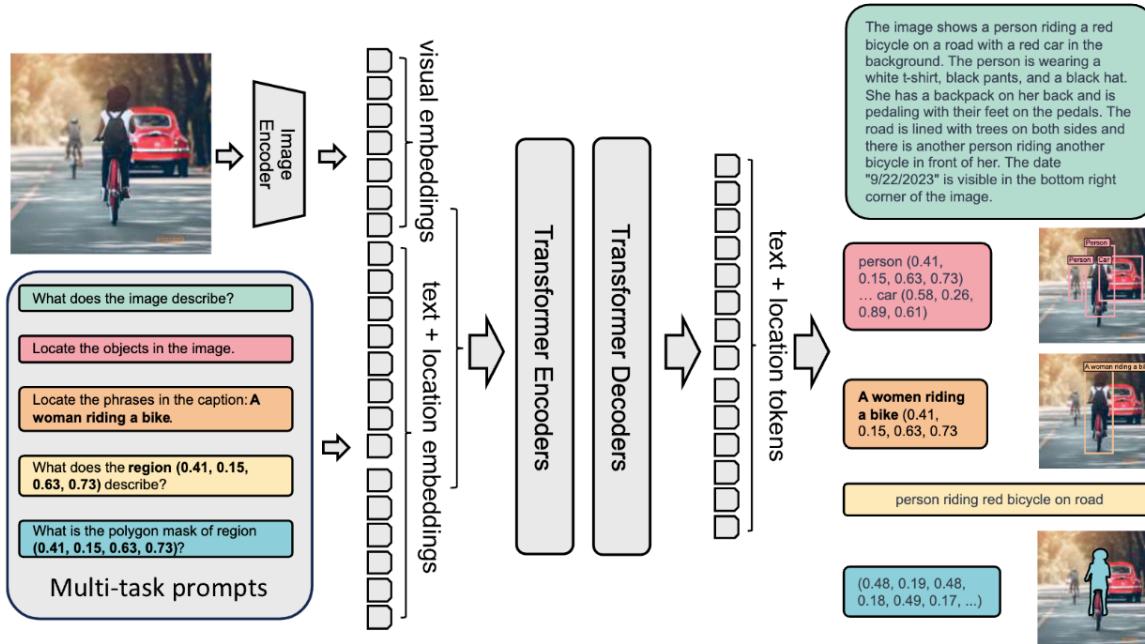
The model's training on the extensive FLD-5B dataset ensures it can effectively handle diverse vision tasks without requiring task-specific modifications. Florence-2 uses textual prompts to activate specific tasks, making it highly flexible and capable of zero-shot generalization. For tasks like object 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:

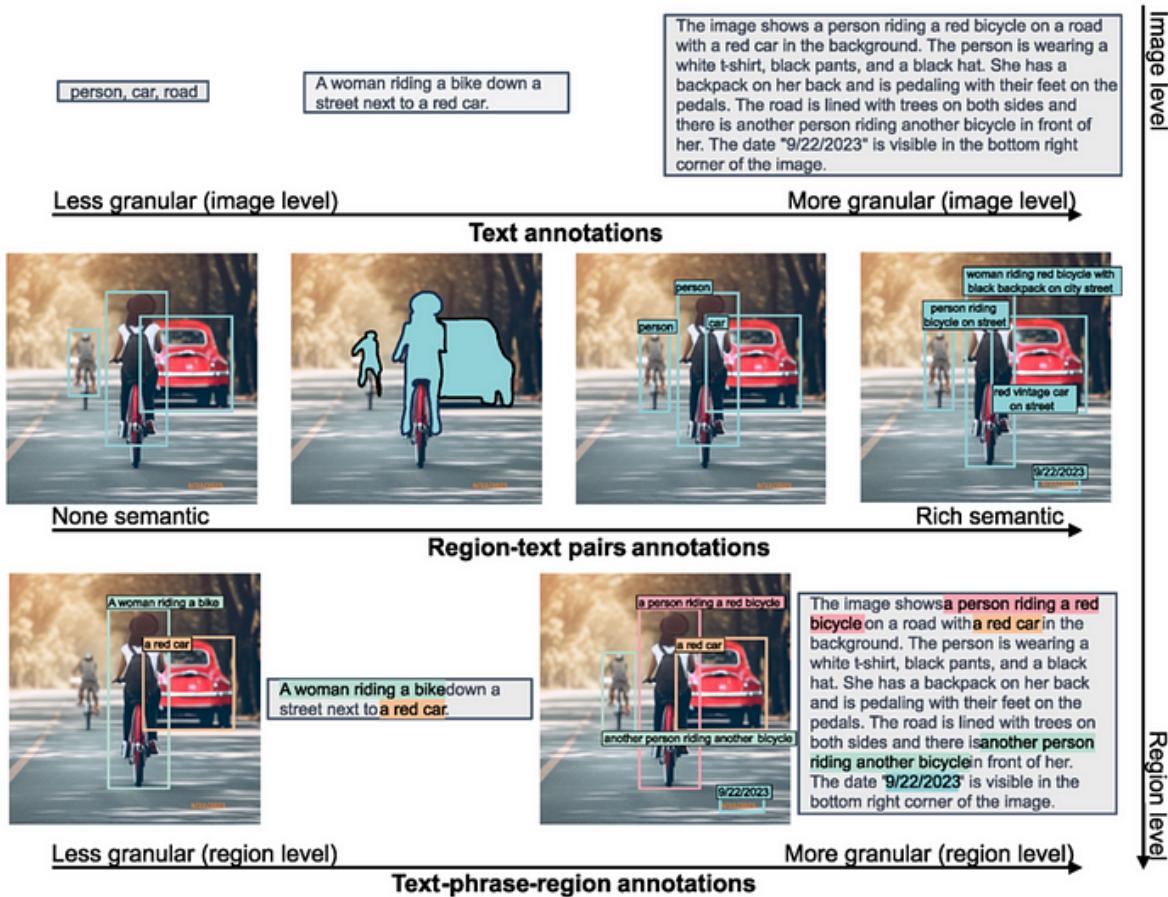
Architecture



- **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.

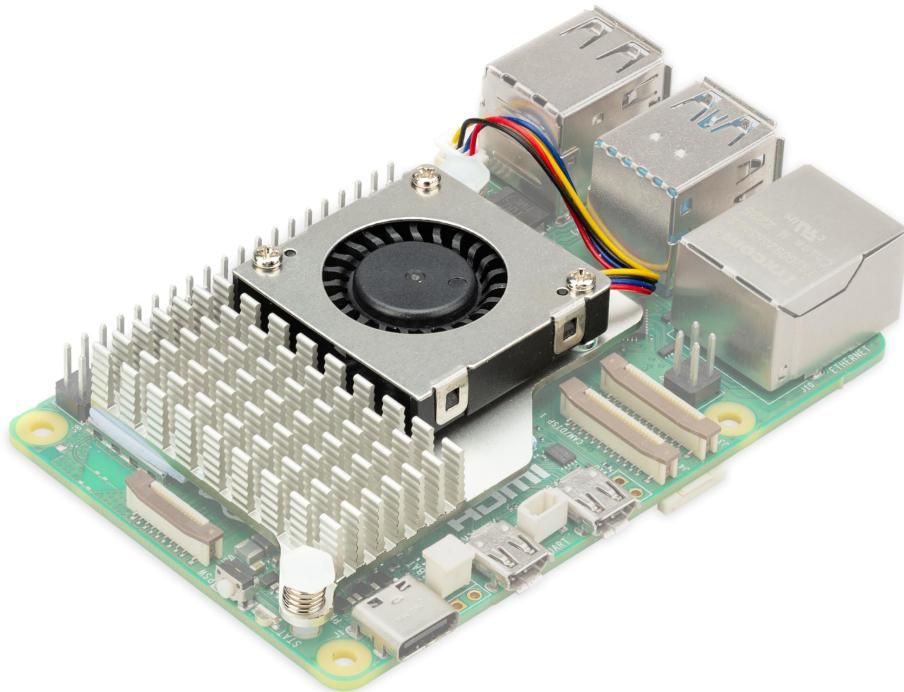
In benchmark tests, Florence-2 has shown remarkable performance in tasks like COCO captioning and referring expression comprehension. It outperformed models like PolyFormer and UNINEXT in object detection and segmentation tasks on the [COCO dataset](#). It is a highly competitive choice for real-world 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 2.4GHz 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 4GB and 8GB of high-speed LPDDR4X SDRAM, with 8GB 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 heatsink 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 Florence-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.

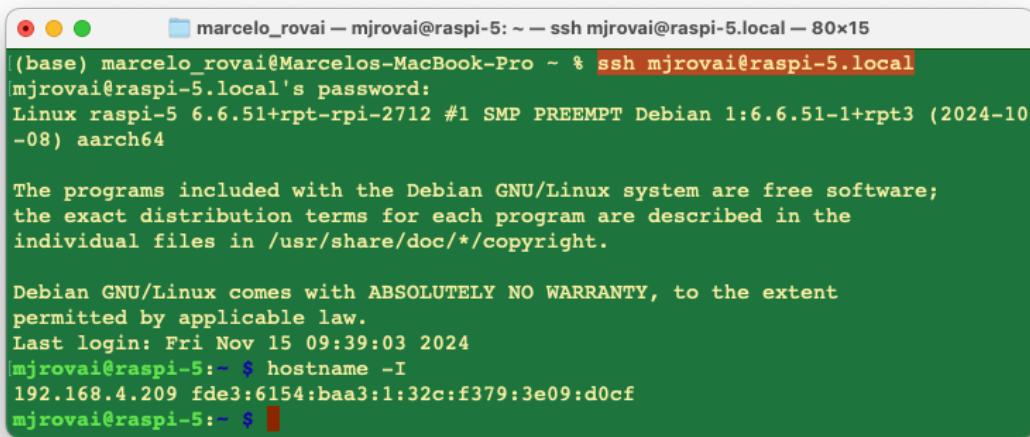
Considering that the Raspberry Pi already has its OS installed, let's use `SSH` to reach it from another computer:

```
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.local's 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
```

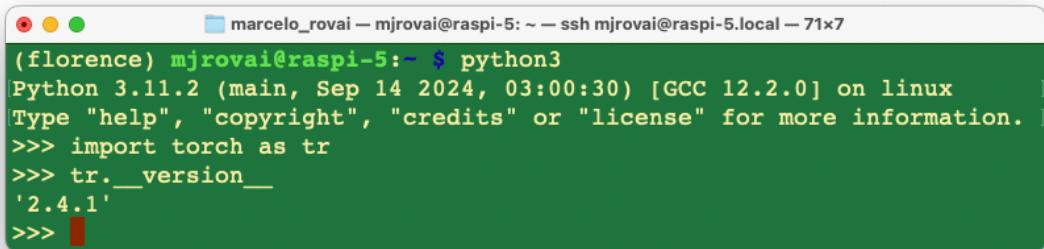
Let's set up and activate a **Virtual Environment** for working with Florence-2:

```
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:



A screenshot of a terminal window titled "marcelo_rovai — mjrovai@raspi-5: ~ — ssh mjrovai@raspi-5.local — 71x7". The window shows the following Python session:

```
(florence) mjrovai@raspi-5:~$ 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.

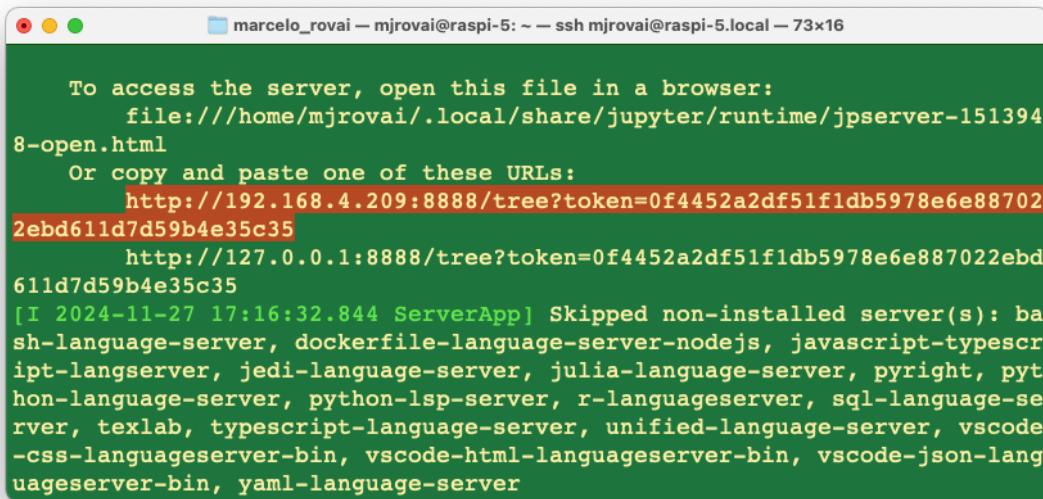
```
pip3 install jupyter
pip3 install numpy Pillow matplotlib
jupyter notebook --generate-config
```

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 screenshot shows an SSH terminal window titled "marcelo_rovai — mjrovai@raspi-5: ~ — ssh mjrovai@raspi-5.local — 73x16". The output of the command shows the Jupyter server starting up. It displays instructions to access the server via a browser or copy/paste URLs. One URL is highlighted in red: <http://192.168.4.209:8888/tree?token=0f4452a2df51f1db5978e6e88702zebd611d7d59b4e35c35>. Below this, there is a log message from the server indicating it skipped non-installed servers.

```
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=0f4452a2df51f1db5978e6e88702zebd611d7d59b4e35c35  
http://127.0.0.1:8888/tree?token=0f4452a2df51f1db5978e6e887022ebd611d7d59b4e35c35  
[I 2024-11-27 17:16:32.844 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
```

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:

1. 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.

2. 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.

3. 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.).

4. 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.

5. 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.

6. 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.

7. 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.

8. 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.

9. Post-processing the Generation

```
parsed_answer = processor.post_process_generation(generated_text, task="",
                                                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.

10. 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()

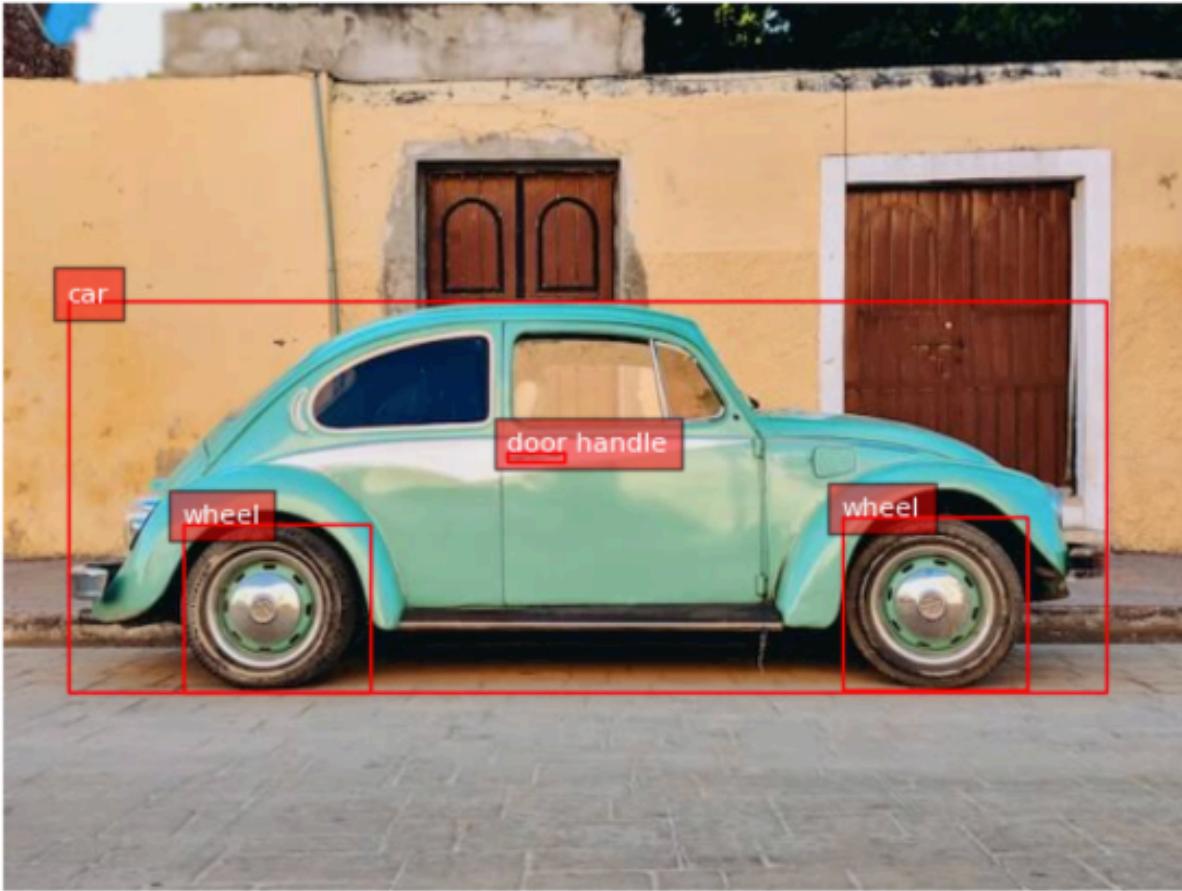
```

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['<0D>'])
```

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:

1. 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.

2. 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.

3. 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.

4. 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 red car is in the image. This is useful for human-computer interaction, where you must find specific objects based on text.

5. 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).

6. 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.

7. 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.

8. 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.

9. 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.

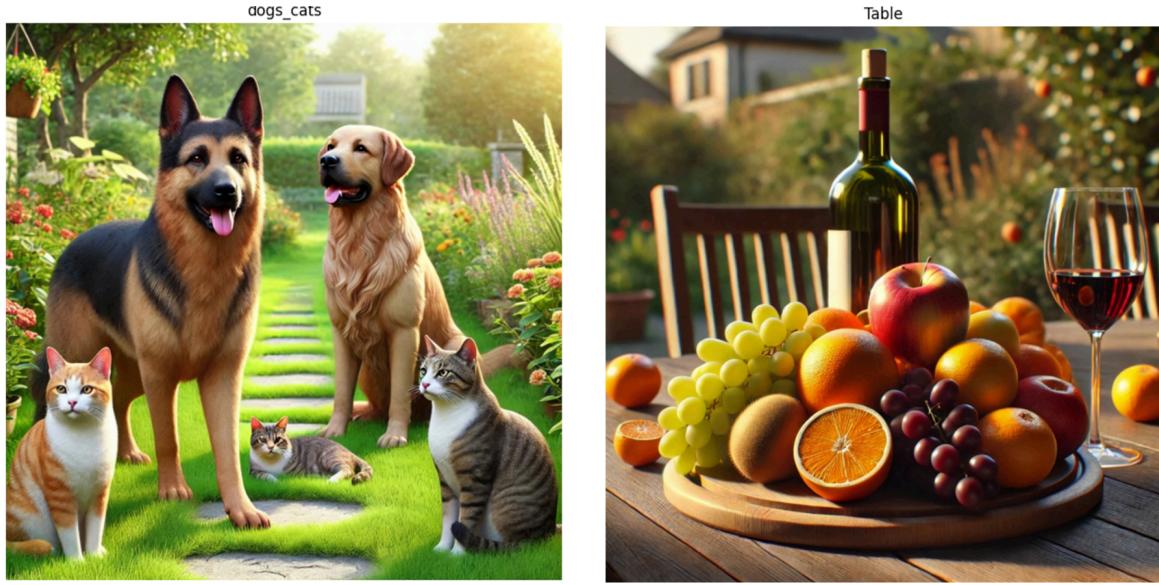
Exploring computer vision and vision-language tasks

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.'}

```

```
background. The sky is visible above them, creating a peaceful atmosphere.'
```

2. Table

```
run_example(task_prompt='<DETAILED_CAPTION>',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='<MORE_DETAILED_CAPTION>',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 or 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).

OD - Object Detection

We can run the same previous function for object detection using the prompt <OD>.

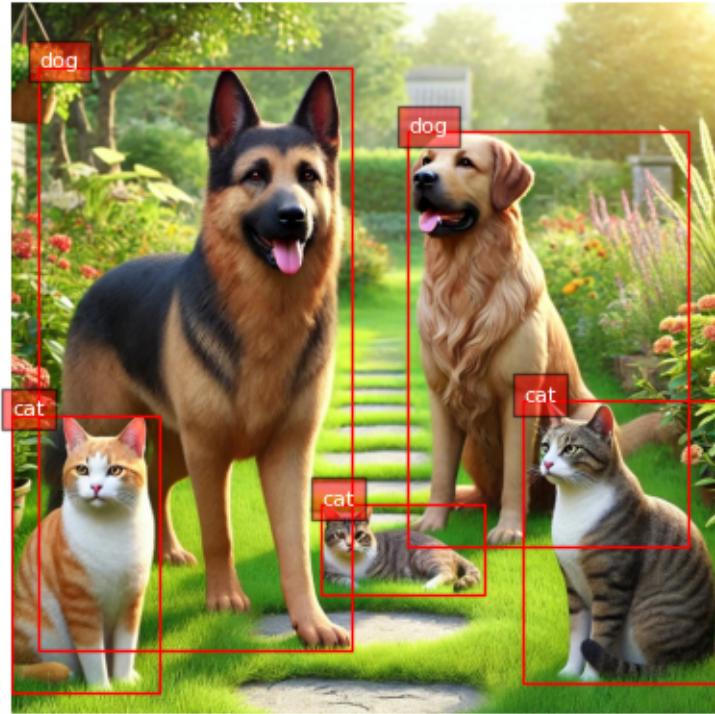
```
task_prompt = '<OD>'  
results = run_example(task_prompt,image=dogs_cats)  
print(results)
```

Let's see the result:

```
[INFO] ==> Florence-2-base (<OD>), took 20.9 seconds to execute.  
  
{'<OD>': {'bboxes': [[[737.7920532226562, 571.904052734375, 1022.4640502929688,  
980.4800415039062], [0.5120000243186951, 593.4080200195312, 211.4560089111328,  
991.7440185546875], [445.9520263671875, 721.4080200195312, 680.4480590820312,  
850.4320678710938], [39.42400360107422, 91.64800262451172, 491.0080261230469,  
933.3760375976562], [570.8800048828125, 184.83201599121094, 974.3360595703125,  
782.8480224609375]], '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 bottle”, “a wine glass,” 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>'])
```



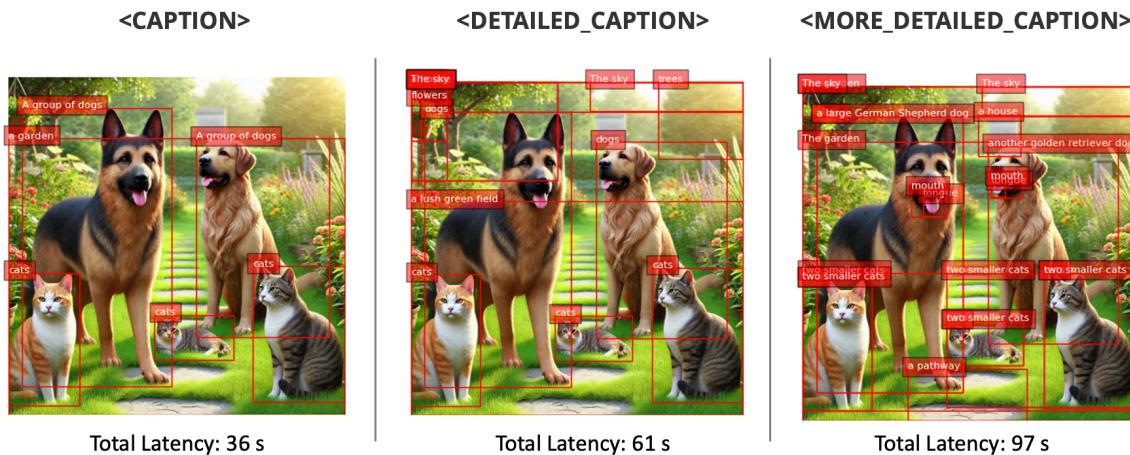
[INFO] ==> Florence-2-base (<CAPTION_TO_PHRASE_GROUNDING>), took 15.7 seconds to execute each task.

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 lead 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, ..., x_n, y_n].

Polygon (x₁, y₁, ..., x_n, y_n): 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:

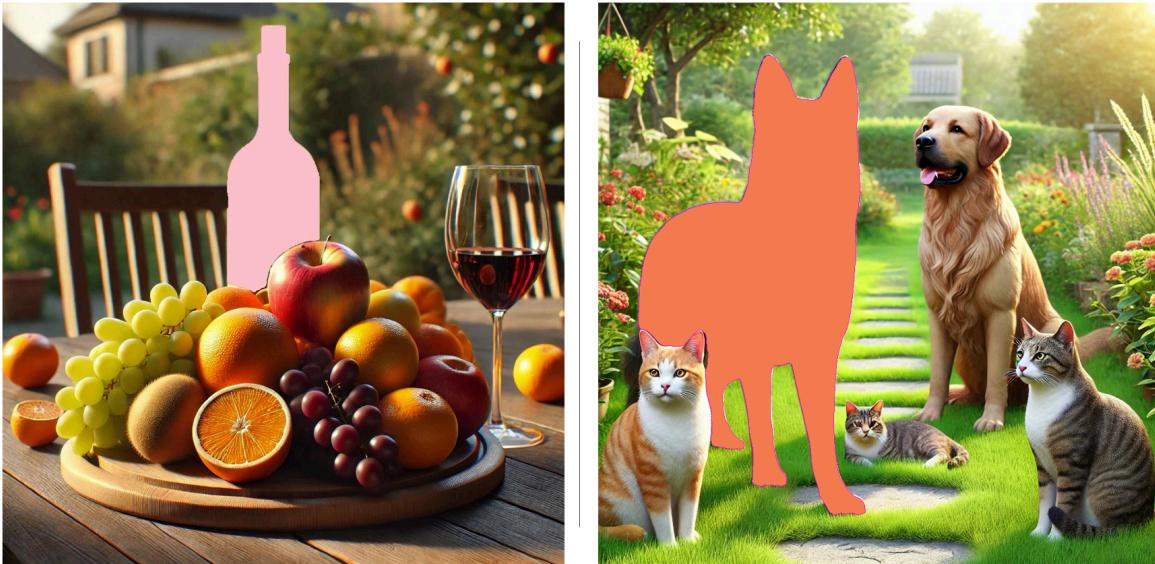
```

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_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_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()

```



Machine Learning Embarcado

Democratizando a Inteligência Artificial para Países em Desenvolvimento



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 às 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 is 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 LearningCafécomEmbarcadoEmbarcadosDemocratizando a  
InteligênciaArtificial para Paises em25 de Setembro ás 17hDesenvolvimentoToda quarta-  
feiraMarcelo RovalProfessor na UNIFIEI eTransmissão viainCo-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, labels = prediction['quad_boxes'], 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']

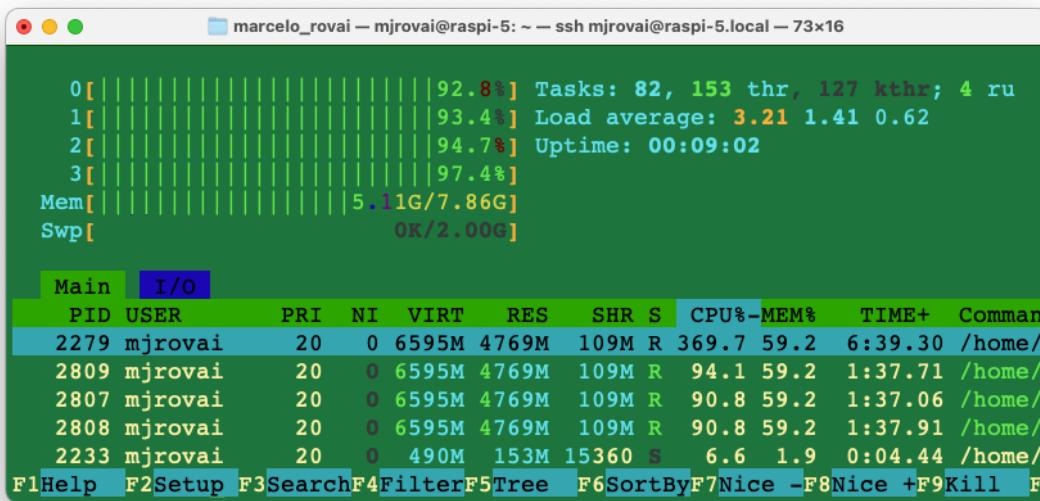
'</s>Machine Learning',
'Café',
'com',
'Embarcado',
'Embarcados',
'Democratizando a Inteligência',
'Artificial para Paises em',
'25 de Setembro ás 17h',
'Desenvolvimento',
'Toda quarta-feira',
'Marcelo Roval',
'Professor na UNIFIEI e',
'Transmissão via',
'in',
'Co-Director do TinyML4D']
```

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.



The screenshot shows the htop command-line interface running in a terminal window. The title bar indicates the user is marcelo_rovai on raspi-5 via ssh. The top section displays system statistics: CPU usage (Tasks: 82, 153 thr, 127 kthr; 4 ru), Load average (3.21 1.41 0.62), Uptime (00:09:02), and memory usage (Mem: 5.11G/7.86G, Swap: 0K/2.00G). Below this is a table of processes, with the I/O tab selected. The table includes columns for PID, USER, PRI, NI, VIRT, RES, SHR, S, CPU%, MEM%, TIME+, and Command. The processes listed are all owned by the user mjrovai and are running in the /home/ directory. The bottom of the screen shows a series of function key labels: F1Help, F2Setup, F3Search, F4Filter, F5Tree, F6SortBy, F7Nice, F8Nice, F9Kill, and F.

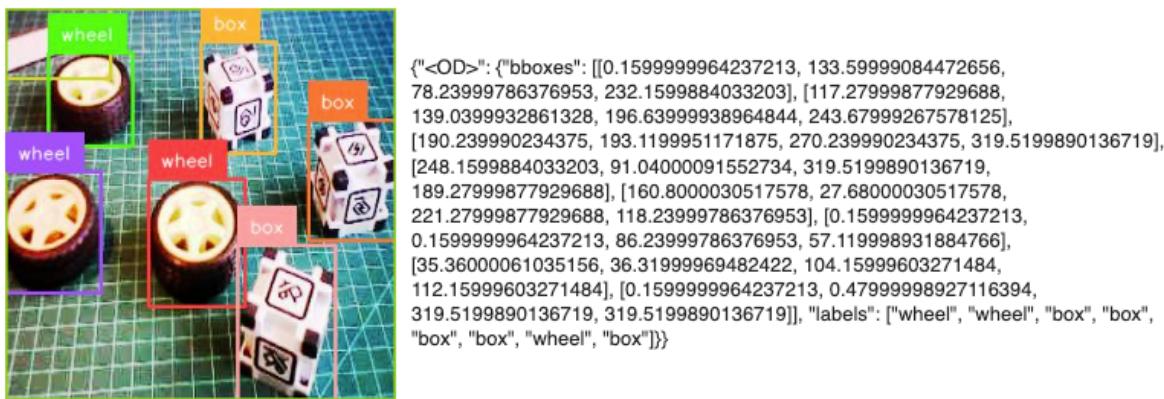
Running complex tasks can use all 8GB of the Raspi-5's memory. For example, the above screenshot during the Florence OD task shows 4 CPUs at full speed and over 5GB of memory in use. Consider increasing the SWAP memory to 2 GB.

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.

Conclusion

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 EasyOCR 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](#)

References

To learn more:

Online Courses

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

Books

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

Projects Repository

- Edge Impulse Expert Network

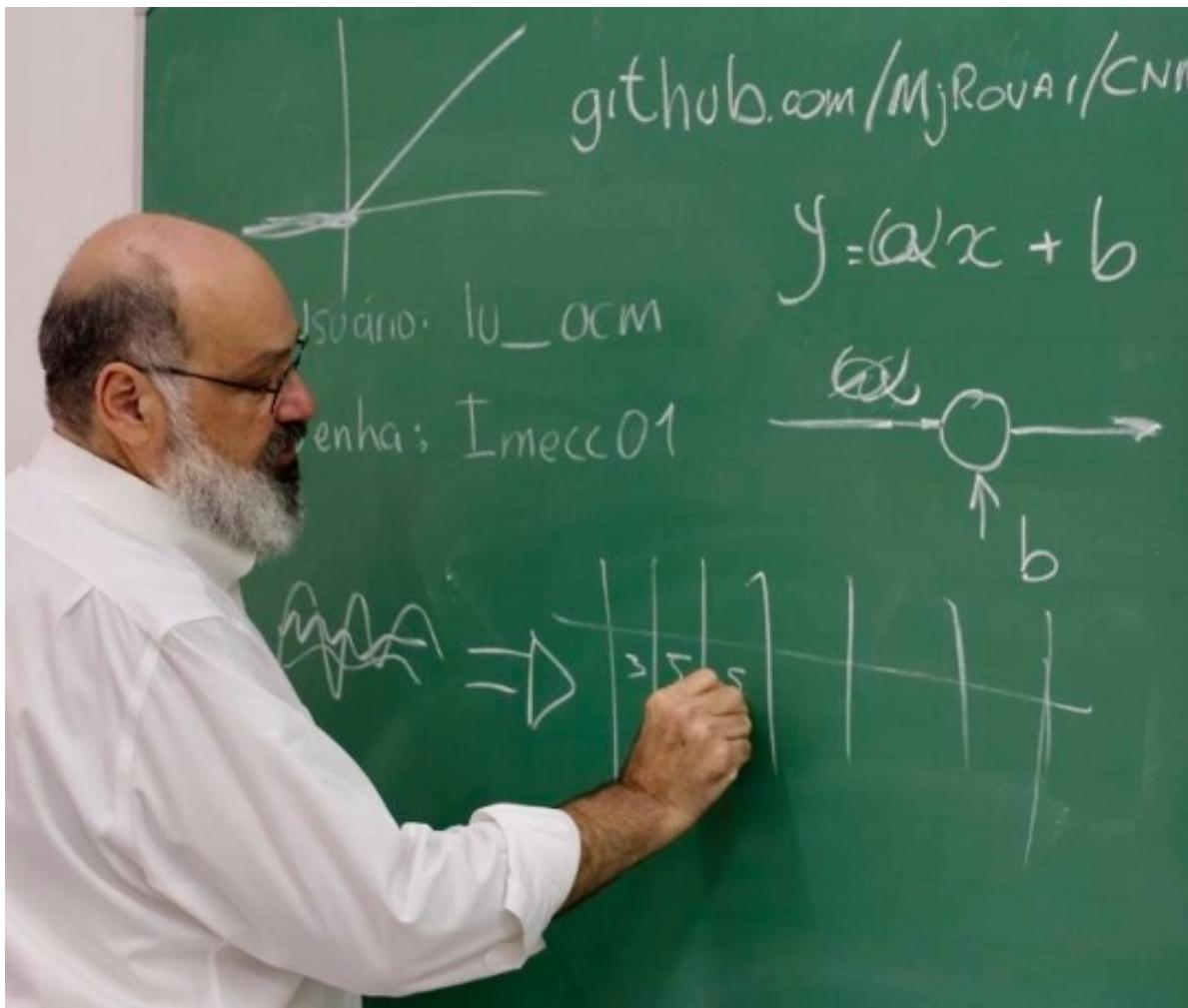
TinyML4D

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



TINYML4D

About the author



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

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

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

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

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

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