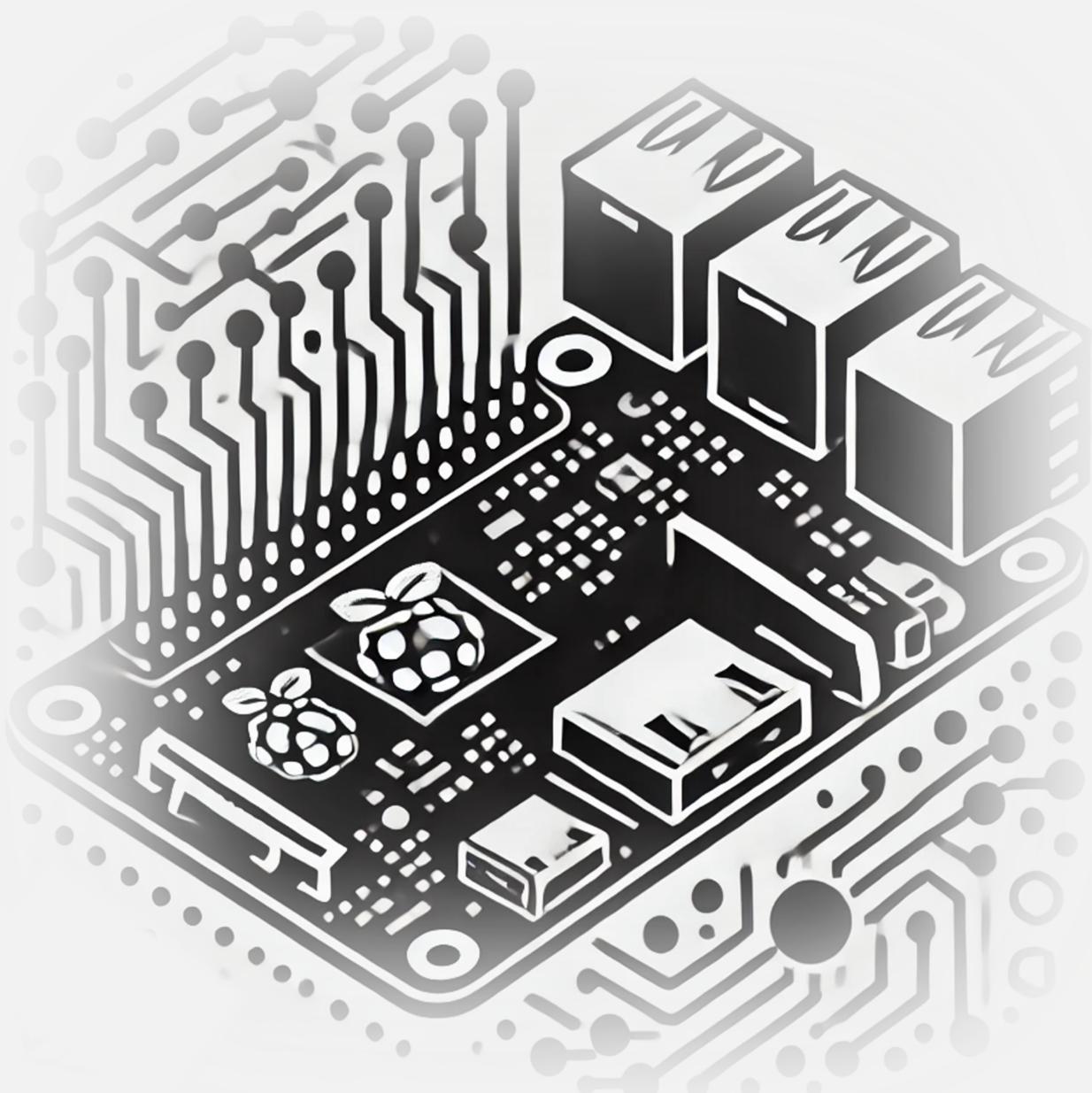


Edge AI Engineering

Hands-On with the Raspberry Pi



UNIFEI

Marcelo Rovai

Professor Honoris Causa

Edge AI Engineering

Hands-on with the Raspberry Pi

Marcelo Rovai

2025-10-23

Table of contents

Preface	3
Acknowledgments	5
Introduction	6
Edge AI Engineering	6
Why Edge AI Matters	6
The Raspberry Pi Advantage	6
What You'll Learn	7
Who This Book Is For	7
About this Book	8
Key Features	8
Structure and Organization	9
Prerequisites	9
Classification of AI Applications	11
Fixed Function AI vs. Generative AI	11
Fixed Function AI (Reactive)	11
Generative AI (Proactive)	12
Summary Table	12
Conclusion	12
The Edge AI Advantage	13
Setup	15
Introduction	16
Key Features	16
Raspberry Pi Models (covered in this book)	17
Engineering Applications	17
Hardware Overview	18
Raspberry Pi Zero 2W	18
Raspberry Pi 5	19
Installing the Operating System	20
The Operating System (OS)	20
Installation	21
Initial Configuration	23

Remote Access	24
SSH Access	24
To shut down the Raspi via terminal:	25
Transfer Files between the Raspberry Pi and a computer	25
Increasing SWAP Memory	29
Installing a Camera	31
Installing a Camera Module on the CSI port	31
Installing a USB WebCam	36
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 Fundamentals	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 System Level Libraries	53
Setting up a Virtual Environment	54
Install Python Packages (inside Virtual Environment)	54
Setting up Jupyter Notebook	56
Installing TensorFlow Lite	61
Creating a working directory:	61
Verifying the Setup	62
Making inferences with Mobilenet V2	64
Define a general Image Classification function	71
Testing the classification with the Camera	73
Exploring a Model Trained from Zero	75
Conclusion:	76
Resources	78
Custom Image Classification Project	80
Image Classification Project	81
The Goal	81
Data Collection	81
Training the model with Edge Impulse Studio	89
Dataset	90
The Impulse Design	91
Image Pre-Processing	93
Model Design	94

Model Training	95
Trading off: Accuracy versus speed	96
Model Testing	98
Deploying the model	98
Live Image Classification	106
Summary:	114
Resources	115
Object Detection: Fundamentals	117
Introduction	118
Object Detection Fundamentals	119
Pre-Trained Object Detection Models Overview	123
Setting Up the TFLite Environment	124
Creating a Working Directory:	124
Inference and Post-Processing	125
EfficientDet	132
Object Detection on a live stream	133
Conclusion	138
Resources	138
Custom Object Detection Project	140
Object Detection Project	141
The Goal	141
Raw Data Collection	142
Labeling Data	145
Training an SSD MobileNet Model on Edge Impulse Studio	155
Uploading the annotated data	155
The Impulse Design	156
Preprocessing all dataset	158
Model Design, Training, and Test	160
Deploying the model	162
Inference and Post-Processing	164
Training a FOMO Model at Edge Impulse Studio	174
How FOMO works?	175
Impulse Design, new Training and Testing	177
Deploying the model	180
Inference and Post-Processing	182
Conclusion	187
Resources	188
Computer Vision Applications with YOLO	189
Exploring a YOLO Models using Ultralitics	189
Talking about the YOLO Model	190

Available Model Sizes	192
Installation	192
Testing the YOLO	193
Export Models to NCNN format	196
Exploring YOLO with Python	197
Inference Arguments	201
Exploring other Computer Vision Applications	204
Environment Setup and Dependencies	205
Model Configuration and Loading	205
Performance Characteristics	205
Results Object Structure	206
Bounding Box Analysis	206
8. Visualization and Customization	207
Exploring Other Computer Vision Tasks	209
Image Classification	209
Instance Segmentation	211
Training YOLO on a Customized Dataset	214
Object Detection Project	214
The Dataset	215
Inference with the trained model, using the Raspi	220
Conclusion	222
Resources	223
Counting objects with YOLO	224
Introduction	225
Estimating the number of Bees	226
Dataset	227
Pre-Processing	231
Training YOLOv8 on a Customized Dataset	234
Inference with the trained model, using the Rasp-Zero	238
Considerations about the Post-Processing	242
Script For Reading the SQLite Database	247
Adding Environment data	247
Conclusion	249
Resources	250
Text Generation with RNNs	251
Introduction	251
What Are We Actually Building?	252
Neural Network Architectures Background	252
The Human Brain Analogy	252
Recurrent Neural Networks (RNN)	252
The Memory Problem and GRU Solution	254

Dataset Preparation	256
Data Preprocessing Steps	257
Tokenization and Vocabulary	257
Character-Level Tokenization	257
Vocabulary Building Process	258
Creating the Character Dictionary	258
Training Sequences	259
The Sliding Window Approach	259
Training Configuration	260
Creating Training Data	261
Character Embeddings	261
From Sparse to Dense Representation	261
Learning Character Relationships	261
Visualization and Understanding	262
Model Architecture	264
RNN Architecture Components	264
Model Summary	264
Memory and Processing Flow	265
Why GRU over Basic RNN?	265
Training Process: Teaching the Model to Write	266
The Learning Objective	266
Hardware and Time Requirements	266
Monitoring Progress	266
Preventing Overfitting	267
Training Configuration	267
Training Implementation	268
Text Generation	268
The Generation Process	268
Temperature Control	269
Implementation	269
Generation Example (Temperature = 0.5)	270
Example Output Analysis	270
Challenges and Limitations	271
Context Window Constraints	271
Character vs. Word Level Trade-offs	271
Coherence Challenges	271
Summary	272
Potential Improvements	272
Connecting to Modern Language Models	272
Scale Comparison	272
Architectural Evolution	273
Training Efficiency	273
Summary: Why Modern Models Perform Better?	273

Conclusion:	274
Resources	275
Knowledge Distillation in Practice	276
Introduction to Knowledge Distillation	277
Why Knowledge Distillation Matters	278
The Teacher-Student Paradigm	278
Theoretical Foundations	279
Soft Targets vs. Hard Targets	279
The Temperature Parameter	279
Distillation Loss Function	280
Implementation with TensorFlow and MNIST	280
Dataset Overview	280
Teacher Model Architecture	281
Student Model Architecture	283
Knowledge Distillation Implementation	284
Training Process	284
Results and Analysis	285
Visualizing Distillation Benefits	285
Advanced Techniques	286
Feature-Based Distillation	286
Attention-Based Distillation	287
Practical Tips for Advanced Distillation	287
Scaling to Large Language Models	287
Challenges in LLM Distillation	287
LLM Distillation Techniques	288
From MNIST to Real-World LLMs: Llama 3.2 Case Study	288
Applications in Modern LLM Development	290
Practical Guidelines and Best Practices	291
Choosing the Right Architecture	291
Hyperparameter Selection	291
Common Pitfalls and Solutions	292
Performance Evaluation	292
Conclusion	292
Key Takeaways	293
From MNIST to LLMs	293
Final Thoughts	293
Resources	293
Small Language Models (SLM)	295
Introduction	296
Setup	296
Raspberry Pi Active Cooler	297

Generative AI (GenAI)	298
Large Language Models (LLMs)	299
Closed vs Open Models:	300
Small Language Models (SLMs)	301
Ollama	303
Installing Ollama	304
Meta Llama 3.2 1B/3B	306
Google Gemma	310
Microsoft Phi3.5 3.8B	315
MoonDream	318
Lava-Phi-3	320
Inspecting local resources	323
Ollama Python Library	325
Ollama Generate	328
Ollama.chat()	333
Image Description:	335
Calling Direct API	340
Error Handling	341
Connection Management	341
Features & Functionality	342
Dependencies	342
Advanced Features	342
When to Use Each?	343
Performance Difference	343
Going Further	344
Conclusion	345
Resources	346
SLM: Basic Optimization Techniques	347
Introduction	347
Function Calling Introduction	349
But what exactly is “function calling”?	349
Using the SLM for calculations	349
The Solution: Function Calling / Tool Use	351
Best Practices	351
Function Calling Solution for Calculations	352
Define the Tool (Function Schema)	352
Implement the Function	352
Project: Calculating Distances	353
Running with other models	360
Adding images	360
Retrieval Augmentation Generation (RAG)	365
A simple RAG project	366

Conclusion	373
Resources	373
Vision-Language Models at the Edge	375
Why Florence-2 at the Edge?	376
Florence-2 Model Architecture	376
Technical Overview	378
Architecture	378
Training Dataset (FLD-5B)	379
Key Capabilities	380
Practical Applications	380
Comparing Florence-2 with other VLMs	381
Setup and Installation	381
Environment configuration	382
Testing the installation	385
4. Defining the Prompt	389
7. Generating the Output	390
Florence-2 Tasks	394
Exploring computer vision and vision-language tasks	396
Caption	398
DETAILED_CAPTION	398
MORE_DETAILED_CAPTION	399
OD - Object Detection	400
DENSE_REGION_CAPTION	402
CAPTION_TO_PHRASE_GROUNDING	403
Cascade Tasks	404
OPEN_VOCABULARY_DETECTION	405
Referring expression segmentation	406
Region to Segmentation	409
Region to Texts	410
OCR	411
Latency Summary	415
Fine-Tuning	416
Conclusion	417
Key Advantages of Florence-2	417
Trade-offs	418
Best Use Cases	418
Future Implications	419
Resources	419
Audio and Vision AI Pipeline	420
Introduction	420

The Audio to Audio AI Pipeline Architecture	421
Understanding Multimodal AI Systems	421
System Architecture Overview	421
Edge AI Considerations	422
Hardware Setup and Audio Capture	422
Audio Hardware Detection	422
Testing Basic Audio Functionality	423
Python Audio Integration	424
Test Audio Python Script	425
Speech Recognition (SST) with Moonshine	427
Why Moonshine for Edge Deployment	427
Model Selection Strategy	427
Implementation and Preprocessing	428
SLM Integration and Response Generation	428
Connecting STT Output to SLMs	428
Optimizing Prompts for Voice Interaction	429
Text-to-Speech (TTS) with PIPER	430
Voice Model Selection and Installation	431
Handling Long Text and Special Cases	433
Pipeline Integration and Optimization	433
Building the Complete System	433
Performance Optimization Strategies	434
Memory Management Considerations	434
Designing Resilient Systems	434
Debugging Complex Pipelines	434
The full Python Script	435
The Image to Audio AI Pipeline Architecture	437
Troubleshooting	442
Conclusion	442
Resources	443
Physical Computing with Raspberry Pi	445
From Sensors to Smart Analysis with Small Language Models	445
Introduction	446
Prerequisites	447
Accessing the GPIOs	447
Pin Numbering Systems	448
Safety First	448
GPIO Zero Library	449
“Hello World”: Blinking an LED	450
Understanding the LED Circuit	450
Testing with GPIO Zero	451
Installing all LEDs (the “actuators”)	453

Sensors Installation and setup	455
Button	456
Installing Adafruit CircuitPython	459
DHT22 - Temperature & Humidity Sensor	461
Installing the BMP280: Barometric Pressure & Altitude Sensor	465
Measuring Weather and Altitude With BMP280	470
Playing with Sensors and Actuators	473
Testing the Notebook setup	475
Initialization	476
GPIO Input and Output	477
Getting and displaying Sensor Data	481
Widgets	483
Advanced GPIO Zero Features	485
PWM LED Brightness Control	485
Composite Devices	485
Other Useful GPIO Zero Devices	486
Interacting an SLM with the Physical world	486
Other Models	494
Conclusion	494
Key Achievements	494
Technical Insights	494
Practical Applications	495
Challenges and Solutions	495
Future Enhancements	496
Final Thoughts	496
Resources	496
Experimenting with SLMs for IoT Control	497
Introduction	497
Setup	499
Hardware Setup	499
Software Prerequisites	501
Basic Sensor Integration	501
SLM Basic Analysis	503
Act on Output (Actuators)	506
Creating new functions for the Actuation:	508
Prompting Engineering	511
Key Changes in the code:	511
System Overview: Enhanced IoT Environmental Monitoring with SLM Control	512
The new code	516
Code Flow Diagram	519
TEST1: Temp above the threshold	519
TEST2: Temp below the threshold	521

TEST3: Alarm Button pressed	522
Interacting with IoT Systems, using Natural Language Commands	522
What will change?	523
How It will Work	524
Key Functions	526
Running the System	527
Tests	527
Special Commands	529
Languages	529
Advantages of This Approach	529
Error Handling	529
Tips for Best Results	530
Flow Diagram	530
Adding Data Logging	531
Key Features	531
Running the System	532
Example Queries	532
Data Files	533
sensor_readings.csv	533
command_history.csv	533
Tips	533
Flow Diagram	534
Prompt Optimization and Efficiency	534
Quick Solution:	535
New Code	537
Key Changes Under the Hood:	537
Using Pydantic	538
2. Implementing Pydantic in the Code	539
Next Steps	542
Conclusion	544
Resources	545
Advancing EdgeAI: Beyond Basic SLMs	546
Understanding SLM Limitations	547
1. Knowledge Constraints	547
2. Reasoning Limitations	548
3. Inconsistent Outputs	548
4. Domain Specialization	548
Techniques for Enhancing SLM at the Edge	549
Optimizing Prompting Strategies	550
Chain-of-Thought Prompting	550
Few-Shot Learning	550

Task Decomposition	551
Building Agents with SLMs	552
General Knowledge Router	563
Improving Agent Reliability	566
1. Function Calling with Pydantic	566
2. Response Validation	569
Retrieval-Augmented Generation (RAG)	570
Understanding RAG	570
Implementing a Naive RAG System	571
Instalation	572
Key Components of the Naive RAG System	572
Advantages of RAG for Edge AI	574
Optimizing RAG for Edge Devices	575
Application: Enhanced Weather Station with RAG	575
Using the RAG System for Edge AI Engineering	577
Testing Different Models and Chunk Sizes	581
Advanced Agentic RAG System	584
System Architecture	585
Key Workflow	586
Important Code Sections	587
Detailed Workflow Diagram	589
Examples	591
Fine-Tuning SLMs for Edge Deployment	593
Preparing for Fine-Tuning	594
Setting Up a Fine-Tuning Process	594
Real implementation: Supervised Fine-Tuning (SFT)	595
Conclusion	597
Resources	598
Edge AI Engineering - Weekly Labs	599
Week 1: Introduction and Setup	599
Lab 1: Raspberry Pi Configuration	599
Lab 2: Development Environment Setup	599
Week 2: Image Classification Fundamentals	600
Lab 3: Working with Pre-trained Models	600
Lab 4: Custom Dataset Creation	601
Week 3: Custom Image Classification	601
Lab 5: Edge Impulse Model Training	601
Lab 6: Model Deployment to Raspberry Pi	602
Week 4: Object Detection Fundamentals	602
Lab 7: Pre-trained Object Detection	602
Lab 8: EfficientDet and FOMO Models	603

Week 5: Custom Object Detection	604
Lab 9: Dataset Creation and Annotation	604
Lab 10: Training Models in Edge Impulse	604
Week 6: Advanced Object Detection	605
Lab 11: FOMO Model Training	605
Lab 12: YOLO Implementation	605
Week 7: Object Counting Project	606
Lab 13: Custom YOLO Training	606
Lab 14: Fixed-Function AI Integration (Optional)	606
Week 8: Introduction to Generative AI	607
Lab 15: Raspberry Pi Configuration for SLMs	607
Lab 16: Ollama Installation and Testing	607
Week 9: SLM Python Integration	608
Lab 17: Ollama Python Library	608
Lab 18: Function Calling and Structured Outputs	609
Week 10: Retrieval-Augmented Generation	610
Lab 19: RAG Fundamentals	610
Lab 20: Advanced RAG	610
Week 11: Vision-Language Models	611
Lab 21: Florence-2 Setup	611
Lab 22: Vision Tasks with Florence-2	611
Week 12: Physical Computing Basics	612
Lab 23: Sensor and Actuator Integration	612
Lab 24: Jupyter Notebook Integration	613
Week 13: SLM-Physical Computing Integration	613
Lab 25: Basic SLM Analysis	613
Lab 26: SLM-IoT Control System	614
Week 14: Advanced Edge AI Techniques	615
Lab 27: Building Agents	615
Lab 28: Advanced Prompting and Validation	615
Week 15: Final Project Integration	616
Lab 29: Agentic RAG System	616
Lab 30: Final Project	616
Hardware Requirements	617
Basic Setup (Weeks 1-7)	617
Generative AI (Weeks 8-15)	617
Physical Computing (Weeks 12-15)	617
Software Requirements	618
Development Environment	618
Computer Vision and DL	618
Generative AI	618
Physical Computing	618
Assessment Criteria	619

Tips for Success	619
References	620
To learn more:	620
Online Courses	620
Books	620
Projects Repository	620
TinyML4D	621
About the author	622

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 (SBC) that has democratized computing and now stands poised to do the same for edge AI.

This book, which serves as the official textbook for [IESTI05 Edge AI Engineering](#) at the Federal University of Itajubá ([UNIFEI](#)) in Brazil, embodies both a passion for technology and a conviction in its capacity to address real-world problems. While developed to support UNIFEI's engineering curriculum, the content is designed to be valuable for all learners, whether in academic settings or pursuing independent study.

"Edge AI Engineering: Hands-on with the Raspberry Pi" 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. Each chapter blends foundational knowledge with practical application, focusing on what's possible with 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 serves as a stepping stone, building your skills and confidence as you progress.

Beyond the technical skills, this book aims to instill 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 looking 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

Brazil, September 2025

Acknowledgments

I extend my 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 served as a beacon and 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.

Google ImageFX and OpenAI's DALL-E generated illustrations of some of the images on the book and chapter covers. Claude Sonnet helped with code and text reviews.

Introduction

Edge AI Engineering

In today's rapidly evolving technological landscape, the convergence of artificial intelligence and edge computing represents one of the most promising frontiers of innovation. Edge AI—the practice of running AI algorithms locally on hardware devices rather than in the cloud—transforms how we interact with technology daily, enabling more responsive, private, and efficient intelligent systems.

This book, “Edge AI Engineering: Hands-on with the Raspberry Pi,” is your practical guide to this exciting field. We'll explore fixed-function AI (reactive systems that process specific inputs) and generative AI (proactive systems that create new content) through hands-on projects using the versatile and accessible Raspberry Pi platform.

Why Edge AI Matters

Traditional AI deployment often relies on cloud infrastructure, requiring constant connectivity and introducing latency. Edge AI addresses these limitations by bringing intelligence directly to where data is generated and actions occur. This approach offers several compelling advantages:

- **Reduced latency:** Process data locally for near-instantaneous responses
- **Enhanced privacy:** Keep sensitive information on your device rather than sending it to remote servers
- **Network independence:** Maintain functionality even without internet connectivity
- **Lower bandwidth usage:** Process data locally, sending only relevant results when needed
- **Energy efficiency:** Optimize processing for resource-constrained environments

The Raspberry Pi Advantage

The Raspberry Pi, with its combination of affordability, processing capability, and extensive GPIO options, provides an ideal platform for exploring Edge AI concepts. From the compact Raspberry Pi Zero 2W to the more powerful Pi 5, these devices offer:

- Sufficient computational power for running optimized AI models
- A complete Linux-based operating system for straightforward development
- Extensive connectivity options for integrating with sensors and actuators
- A vibrant community and ecosystem of libraries and tools
- An accessible entry point for students, hobbyists, and professionals alike

What You'll Learn

This book takes a progressive approach to Edge AI engineering, starting with foundational concepts and building toward more advanced applications:

1. **Essential setup and configuration:** Prepare your Raspberry Pi for Edge AI development
2. **Computer vision applications:** Implement image classification and object detection systems
3. **Small Language Models (SLMs):** Run and optimize language models directly on your Raspberry Pi
4. **Vision-Language Models:** Explore multimodal AI with Florence-2
5. **Physical computing integration:** Connect AI systems with sensors and actuators
6. **Advanced optimization techniques:** Enhance model performance through methods like RAG, agents, and function calling

Each chapter includes detailed explanations, step-by-step instructions, and practical projects demonstrating real-world applications of Edge AI concepts.

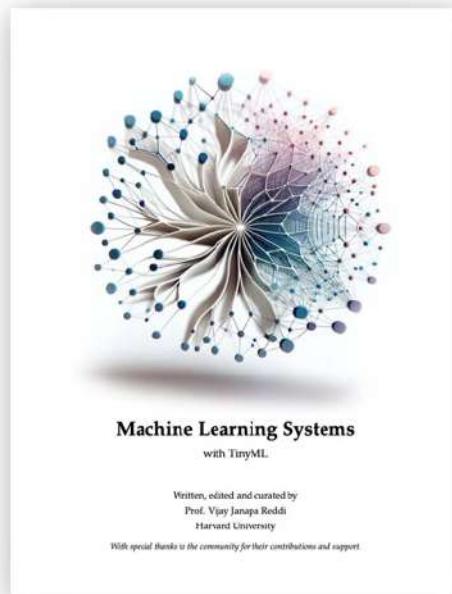
Who This Book Is For

Whether you're a student exploring AI for the first time, an educator developing a curriculum, a maker building innovative projects, or a professional seeking to expand your skills, this book provides the knowledge and hands-on experience needed to implement Edge AI solutions on the Raspberry Pi platform successfully.

Join us on this journey to the edge of AI innovation, where we'll bridge theory and practice through engaging, accessible projects that demonstrate the transformative potential of intelligent edge computing.

About this Book

Several chapters in this book are also part of the open book [Machine Learning Systems](#), which we invite you to read.



“Edge AI Engineering: Hands-on with the Raspberry Pi” is designed as a practical, project-based learning resource that bridges theoretical AI concepts with tangible implementations. This book is part of the open-source Machine Learning Systems initiative, democratizing access to AI education and applications.

Key Features

- 1. Progressive Learning Path:** The book structure follows a natural progression from basic to advanced concepts, beginning with foundational computer vision applications and advancing to generative AI techniques.

2. **Model-Specific Optimizations:** Each chapter provides targeted guidance for different Raspberry Pi models, helping you maximize performance whether using a Pi Zero 2W or Pi 5.
3. **Open-Source Foundation:** We emphasize accessible tools and frameworks, including Edge Impulse Studio, TensorFlow Lite, PyTorch, Transformers, and Ollama, ensuring you can continue your learning journey with widely available resources.
4. **Practical Problem-Solving:** Rather than abstract exercises, each project addresses real-world challenges that demonstrate the practical value of Edge AI.
5. **Resource Optimization Techniques:** Learn essential strategies for deploying AI on resource-constrained devices, balancing performance needs with hardware limitations.
6. **Cross-Domain Applications:** Explore implementations spanning computer vision, natural language processing, and physical computing, showcasing the versatility of Edge AI.

Structure and Organization

The book is organized into two main sections:

1. **Fixed Function AI (Computer Vision):** Chapters covering image classification, object detection, and specialized applications like object counting.
2. **Generative AI (Language and Vision Models):** Chapters exploring Small Language Models, Vision-Language Models, physical computing integration, and advanced optimization techniques.

Each chapter follows a consistent format that includes:

- Conceptual background and theory
- Step-by-step implementation guides
- Practical projects with complete code
- Performance optimization strategies
- Ideas for further exploration

Prerequisites

While designed to be accessible, readers will benefit from:

- Basic Python programming knowledge
- Familiarity with Linux command-line basics
- Elementary understanding of machine learning concepts
- Previous experience with Raspberry Pi (helpful but not required)

By completing this book, you'll possess the skills to design, implement, and optimize Edge AI applications across a wide range of use cases, leveraging the unique capabilities of the Raspberry Pi platform to bring intelligence to the edge.

Classification of AI Applications

As we embark on our journey through Edge AI Engineering with the Raspberry Pi, it's essential to understand the fundamental classification of AI applications that form the structure of this book. Our exploration is divided into two parts, each representing a different paradigm in artificial intelligence implementation.

Fixed Function AI vs. Generative AI

AI applications can be broadly categorized into two approaches that represent different capabilities, interaction models, and implementation strategies:

Fixed Function AI (Reactive)

Fixed Function AI, or Reactive AI, operates by analyzing specific inputs according to predetermined patterns and rules and then producing consistent outputs for given scenarios. These systems:

- **Respond to specific triggers:** They activate only when presented with particular inputs.
- **Follow defined patterns:** Their behavior is predictable and consistent.
- **Excel at structured tasks:** They perform exceptionally well at classification, detection, and pattern recognition
- **Operate within boundaries:** Their capabilities are limited to their specific programming.

In the first part of this book (Chapters 2-4), we explore fixed-function AI through computer vision applications:

- Image classification for identifying objects in images
- Object detection for locating and labeling multiple objects
- Specialized detection applications like counting objects

These applications demonstrate how edge devices can deliver reliable, efficient AI in constrained environments, focusing on specific, well-defined tasks.

Generative AI (Proactive)

Generative AI, also known as Proactive AI, represents a fundamental shift in capability. These systems can:

- **Create new content:** They generate novel text, images, or solutions.
- **Understand context:** They interpret and respond to nuanced situations.
- **Engage in dialogue:** They maintain contextual awareness across interactions.
- **Adapt to novel scenarios:** They apply knowledge to situations beyond their explicit training.

The second part of this book (Chapters 5-9) explores Generative AI at the edge:

- Small Language Models that bring conversational AI to edge devices
- Vision-Language Models that combine visual and textual understanding
- Physical computing integration that connects AI to the real world
- Advanced techniques to enhance edge AI capabilities

This progression from Fixed Function to Generative AI mirrors the evolution of artificial intelligence itself—from specialized systems designed for specific tasks to more flexible, creative systems capable of addressing a broader range of challenges.

Summary Table

AI Type	Core Focus	Example Applications	Typical Use Cases
Fixed Function (Reactive)	Data analysis, assessment, automation	Fraud detection, spam filters, image recognition	Banking, healthcare diagnostics, security systems
Generative (Proactive)	Content creation, anticipation, dialogue	ChatGPT, DALL·E, predictive maintenance, smart assistants	Content creation, customer support, design, automation

Conclusion

- **Fixed Function (Reactive) AI** is ideal for applications requiring efficiency, predictability, and low resource use, where tasks are well-defined and do not require creative output or adaptation.
- **Generative (Proactive) AI** is suited for scenarios demanding creativity, personalization, and anticipatory actions. It enables richer, more human-like interactions and innovative solutions across industries.

The Edge AI Advantage

Both Fixed Function and Generative AI gain unique benefits when deployed at the edge:

- **Reduced latency:** Processing happens locally, eliminating network delays
- **Enhanced privacy:** Sensitive data remains on-device
- **Operational reliability:** Systems function regardless of network connectivity
- **Resource efficiency:** Optimized models utilize limited hardware effectively

By understanding these fundamental classifications, you'll realize how different AI approaches serve distinct purposes and how each can be effectively implemented on edge devices like the Raspberry Pi.

As we progress through this book, this classification framework will help contextualize each project and technique, connecting individual implementations to broader AI concepts and applications.

#

Fixed Function AI (Reactive)

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 the Raspberry Pi Zero 2 W (Raspi-Zero) and the 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 provides access to interfaces like I2C, SPI, and UART, allowing for detailed control and communication with external devices.
5. **Real-Time Capabilities:** With proper configuration, Raspberry Pis can be used for soft real-time applications, making them suitable for control systems and signal processing tasks.
6. **Power Efficiency:** Low power consumption enables battery-powered and energy-efficient designs, especially in models like the Pi Zero.

Raspberry Pi Models (covered in this book)

1. **Raspberry Pi Zero 2 W** (*Raspi-Zero*):
 - Ideal for: Compact embedded systems
 - Key specs: 1GHz single-core CPU (ARM Cortex-A53), 512MB RAM, minimal power consumption (usually, around 600mW in Idle. Also, the power-off (“zombie current”) is very low, around 45mA (225mW), which is significantly less than full-size Raspberry Pi models.
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, higher power consumption (usually, around 3-3.5W in Idle)

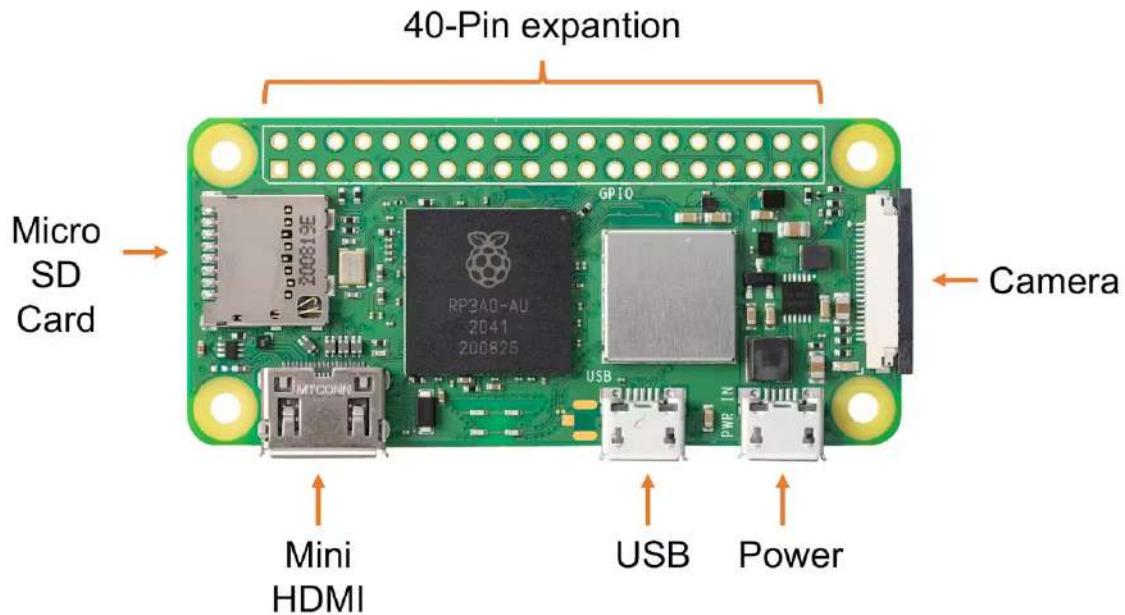
Engineering Applications

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.

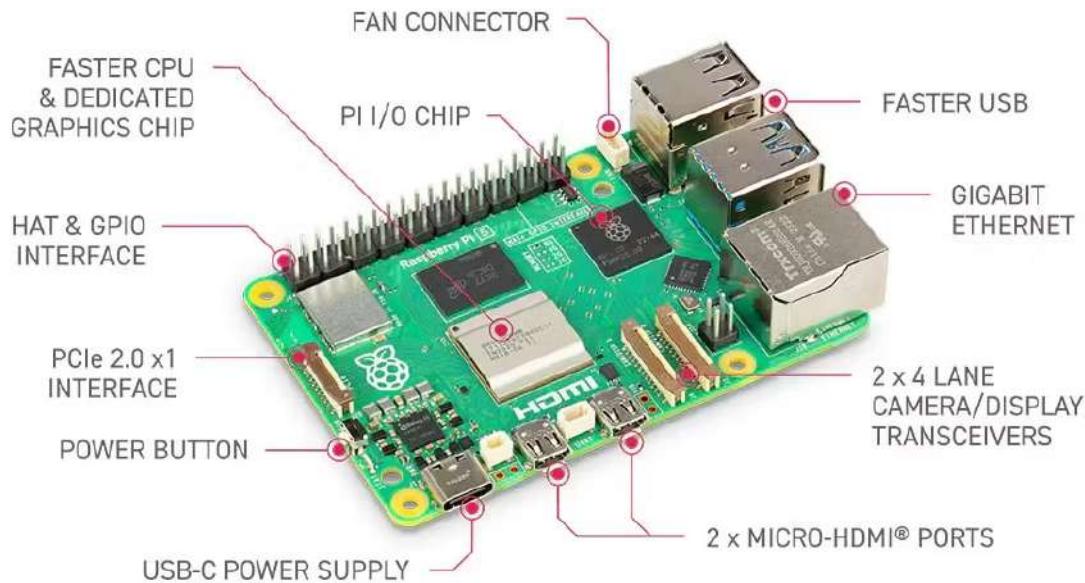
Hardware Overview

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/2.5A via micro USB port [12.5W Micro USB power supply](#)

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 (2.4 GHz and 5 GHz), 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/5A, 5V/3A limits peripherals to 600mA, via USB-C connector [27W USB-C power supply](#)

In the labs, we will use different names to address the Raspberry Pi: **Raspi**, **Raspi-5**, **Raspi-Zero**, etc. Usually, **Raspi** or **Raspberry Pi** is used when the instructions or comments apply to every model.

Installing the Operating System

The Operating System (OS)

An operating system (OS) is essential software that manages computer hardware and software resources, providing standard services for computer programs. It is the core software that runs on a computer, serving as an intermediary between hardware and application software. The OS oversees 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.

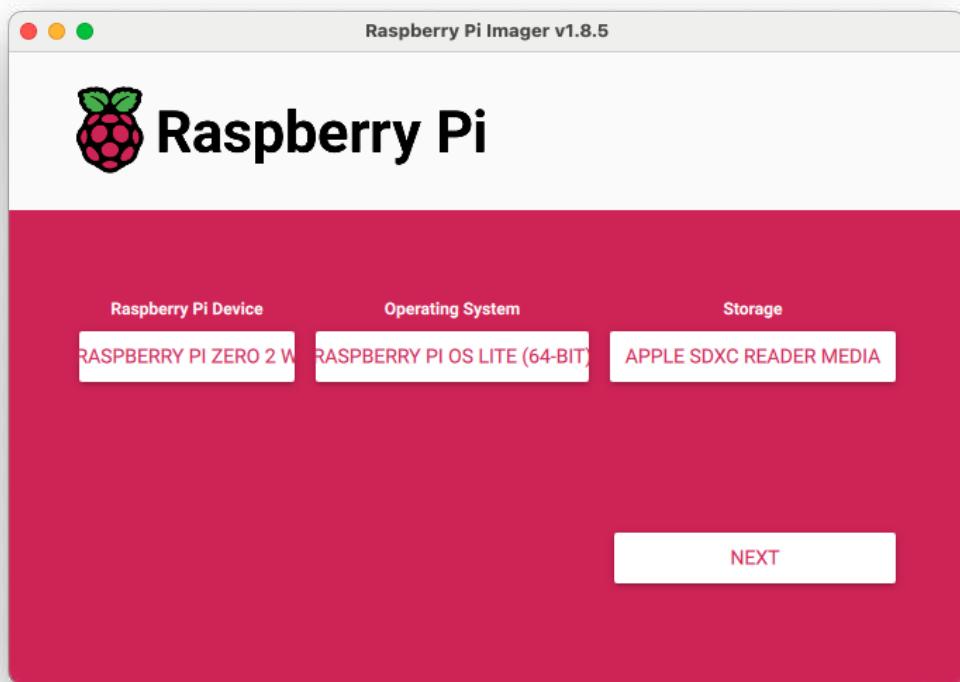
Installation

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.

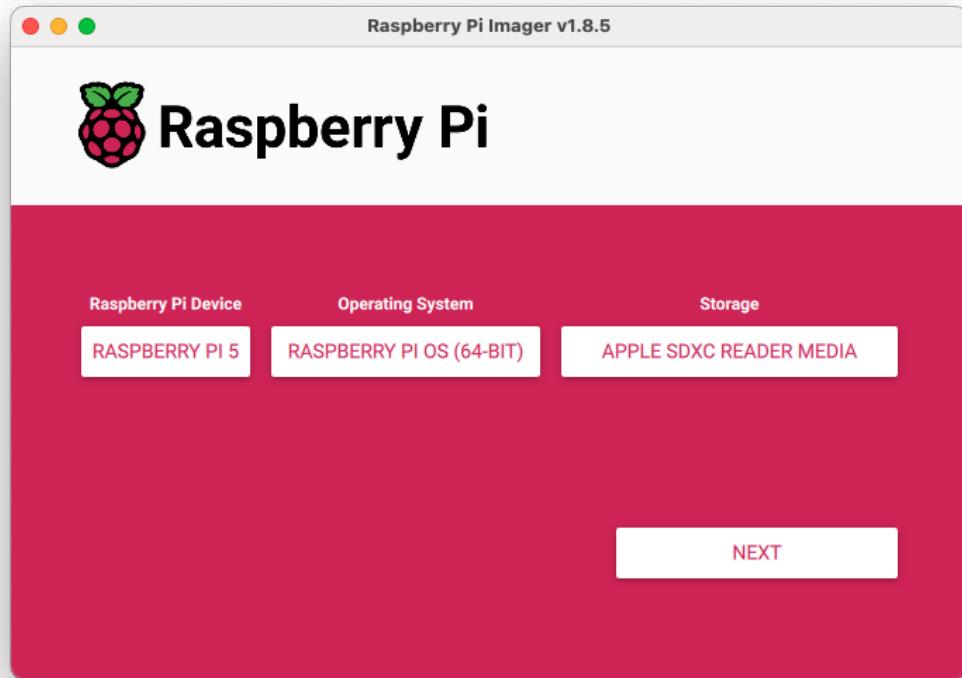
Follow the steps to install the OS on your Raspberry Pi.

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)**.

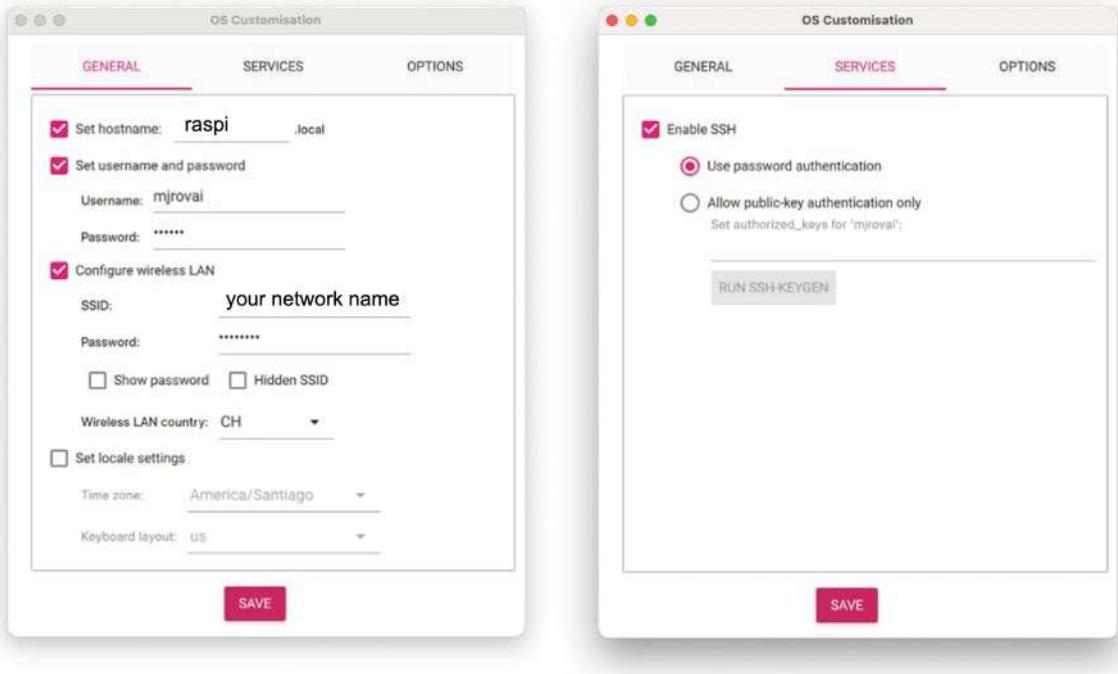


Due to its reduced SDRAM (512MB), the recommended OS for the Raspberry Pi Zero is the 32-bit version. However, to run some machine learning models, such as the YOLOv8 from Ultralitics, we should use the 64-bit version. Although Raspi-Zero can run a *desktop*, we will choose the LITE version (no Desktop) 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 Raspberry Pi *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. Please replace it with the one you're currently using.

Initial Configuration

1. Insert the microSD card into your Raspberry Pi.
2. Connect power to boot up the Raspberry Pi. For the Rasp-5, use a 5V/3A (or 5A) external Power supply. For the Rasp-Zero, use a 5V/2.5A external Power supply (Optionally, for light uses, it is possible to use the computer's USB to power on the Rasp-Zero.)
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 Raspberry Pi [here](#) or [here](#).

Remote Access

SSH Access

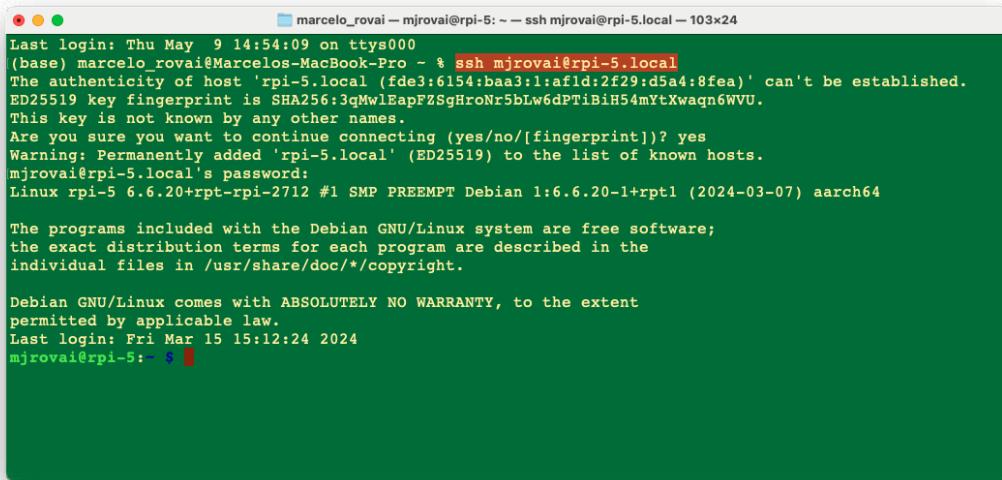
The easiest way to interact with the Raspi-Zero is via SSH (“Headless”). You can use a Terminal (MAC/Linux), [PuTTy](#) (Windows), or any other.

The Raspberry Pi and the notebook should be on the same WiFi network. Note that the Raspberry Pi 5 is dual-band 802.11ac Wi-Fi (2.4 GHz and 5 GHz), but the Raspberry Pi Zero 2W is only 2.4GHz.

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: `ssh username@hostname.local` for example, `ssh mjrovai@rpi-5.local` , `ssh mjrovai@raspi.local` , etc.



```
marcelo_rovai — mjrovai@rpi-5: ~ — ssh mjrovai@rpi-5.local — 103x24
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:a1d:2f29:d5a4:8fea)' can't be established.
ED25519 key fingerprint is SHA256:3qMw1EapFZSgHroNr5bLw6dPTiBiH54mYtXwaqn6WVU.
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 Raspberry Pi. 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 Raspberry Pi IP address. On the terminal, you can use:

```
hostname -I
```



```
[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 a safety shutdown, 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 Raspberry Pi and a computer

Transferring files between the Raspberry Pi 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):

0.0.0.0.1 * 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`.

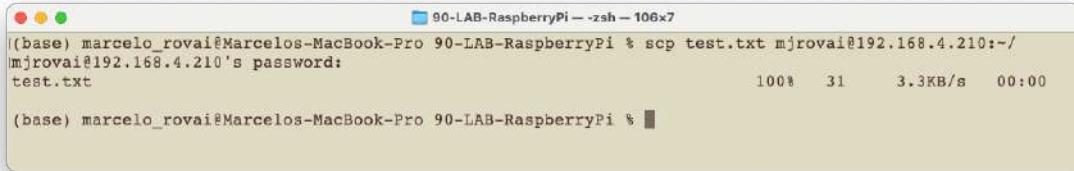
To copy the file named `test.txt` from your personal computer to a user's home folder on your Raspberry Pi, run the following command from the directory containing `test.txt`, replacing the `<username>` placeholder with the username you use to log in to your Raspberry Pi and the `<pi_ip_address>` placeholder 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.

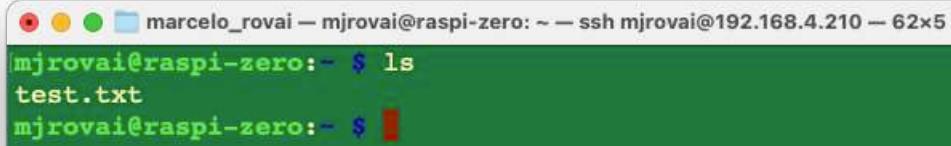
For example, let's transfer the file `test.txt` to the ROOT of my Raspi-zero, which has an IP of `192.168.4.210`:

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



The screenshot shows a terminal window titled "90-LAB-RaspberryPi - zsh - 106x7". The command `scp test.txt mjrovai@192.168.4.210:~/` is being typed. A password prompt "mjrovai@192.168.4.210's password:" appears. The progress bar shows 100% completion with a speed of 3.3KB/s and a time of 00:00. The command is completed with a success message "(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:



The screenshot shows a terminal window titled "marcelo_rovai — mjrovai@raspi-zero: ~ — ssh mjrovai@192.168.4.210 — 62x5". The command `ls` is run, showing a single file named `test.txt`. The command is completed with a success message "mjrovai@raspi-zero: ~ \$".

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 .
```

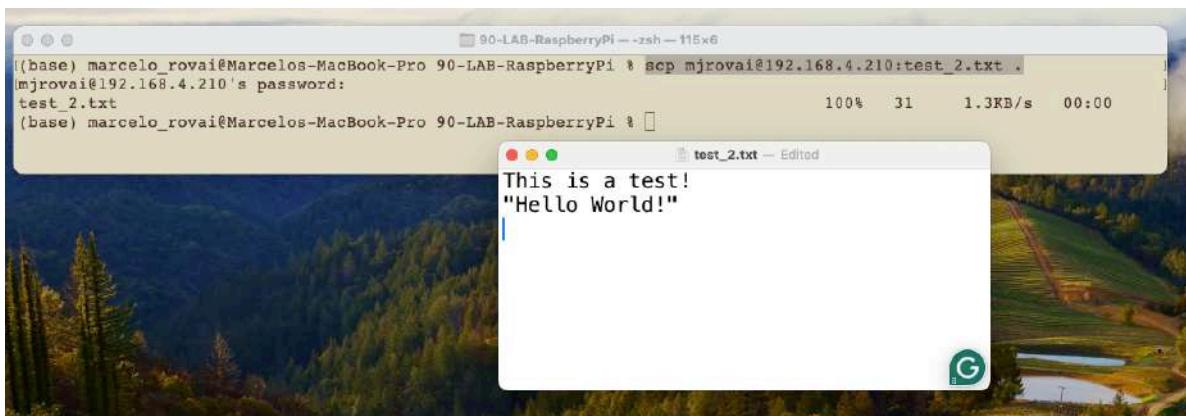
For example:

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 .
```

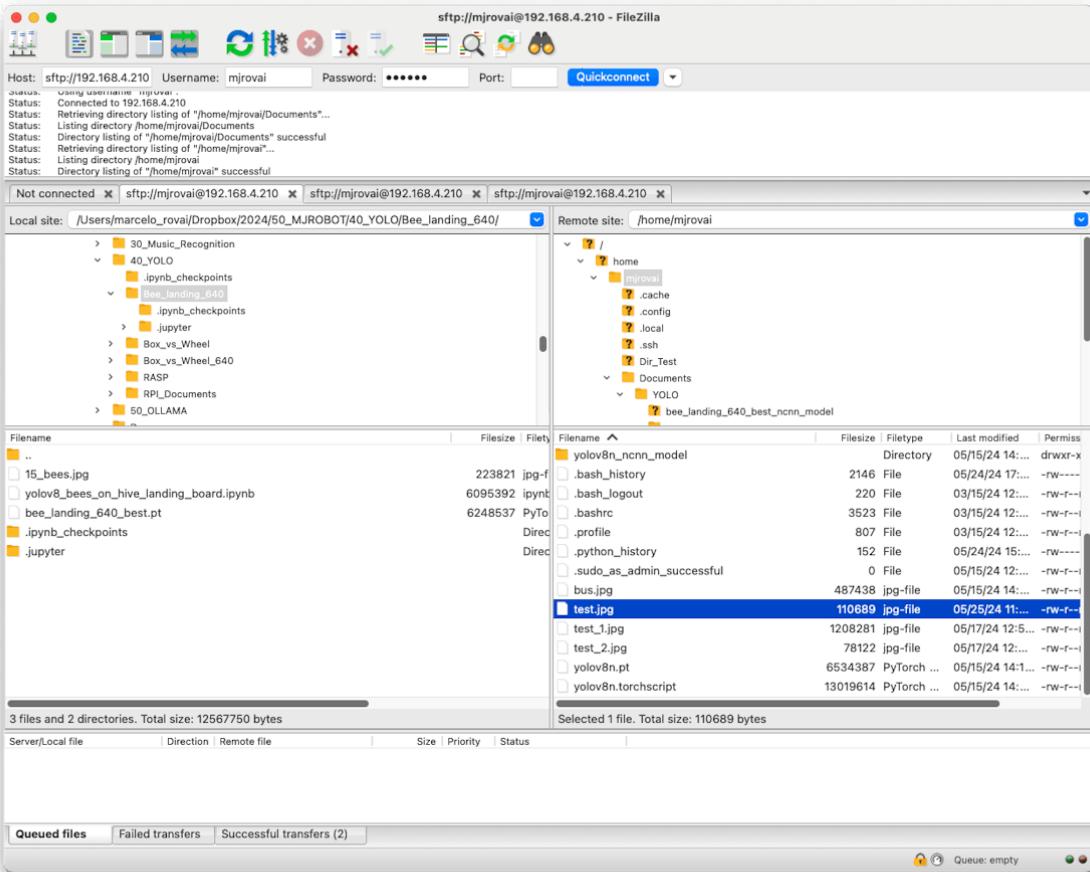


Transferring files using FTP

Transferring files using FTP, such as [FileZilla FTP Client](#), is also possible. Follow the instructions to install the program on your Desktop, then use the Raspberry Pi's IP address as the Host. For example:

```
sftp://192.168.4.210
```

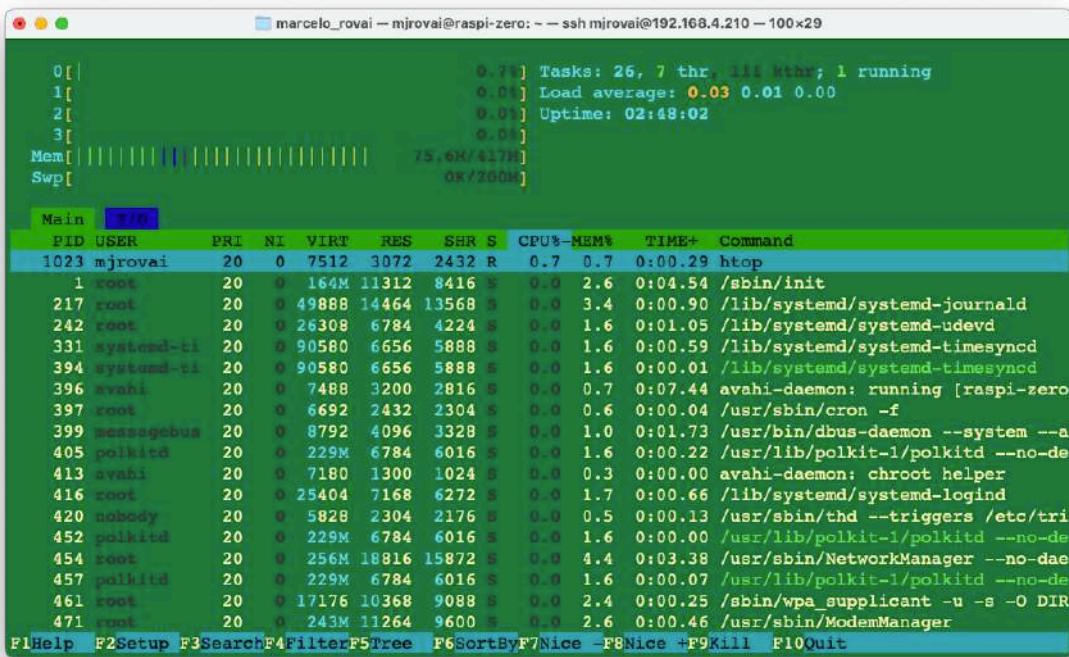
Enter your Raspberry Pi username and password. Pressing Quickconnect will open two windows, one for your host computer desktop (right) and another for the Raspberry Pi (left).



Increasing SWAP Memory

Using `htop`, a cross-platform interactive process viewer, you can easily monitor the resources running on your Raspberry Pi, such as the list of processes, the running CPUs, and the memory used in real-time. To lunch `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 Raspberry Pi 4 or 5. For any Raspberry Pi, it is possible to increase the memory available to the system with “Swap.” Swap memory, also known as swap space, is a technique used in computer operating systems to temporarily store data from RAM (Random Access Memory) on the SD card when the physical RAM is fully utilized. This allows the operating system (OS) to continue running even when RAM is full, which can prevent system crashes or slowdowns.

Swap memory benefits devices with limited RAM, such as the Raspberry Pi Zero. Increasing swap can help run more demanding applications or processes, but it’s essential to balance this with the potential performance impact of frequent disk access.

By default, the Rapi-Zero’s SWAP (Swp) memory is only 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, open and modify 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, requiring a camera. 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.

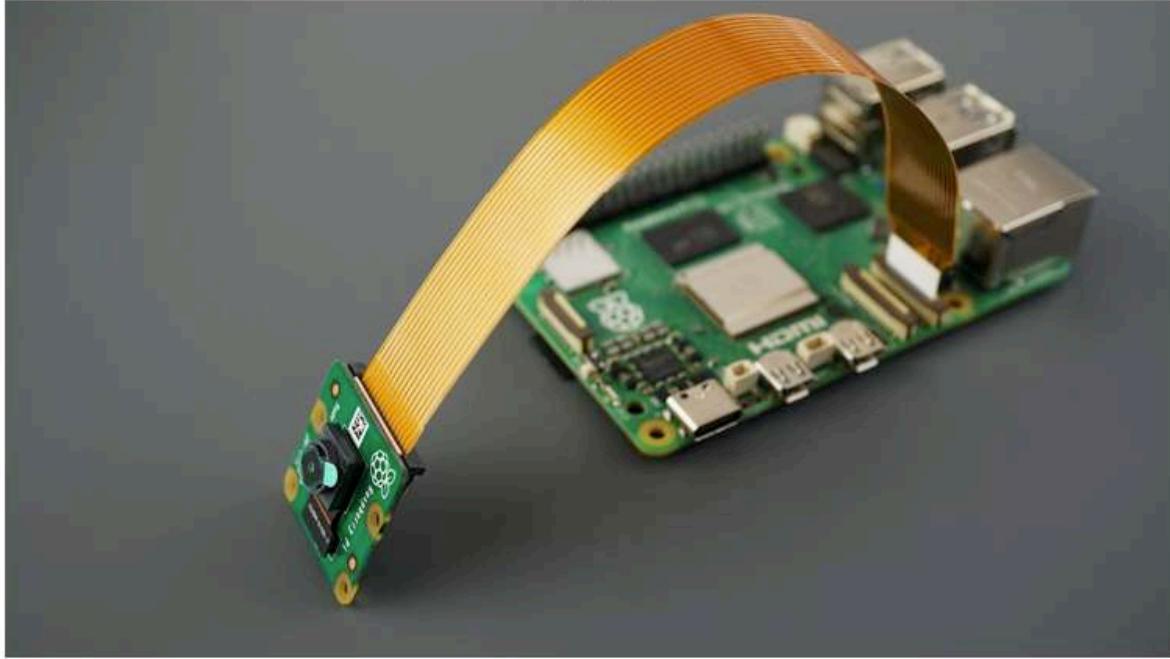
Installing a Camera Module on the CSI port

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:



First, try to list the installed cameras:

```
rpicam-hello --list-cameras
```

If the command is not recognized, install

```
sudo apt install libcamera-apps
```

Try to list the installed camera again. If Ok, you should see something like:

```
marcelo_rovai@mjrovai@raspi-zero: ~ ssh mjrovai@192.168.4.210 - 80x11
[mjrovai@raspi-zero: ~ $ rpicam-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 [46.34 fps - (0, 0)/2592x1944 crop]
            1920x1080 [32.81 fps - (348, 434)/1928x1080 crop]
            2592x1944 [15.63 fps - (0, 0)/2592x1944 crop]

mjrovai@raspi-zero: ~ $
```

Any camera module will work on the Raspberry Pi, but for that, it is possible that the `config.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:

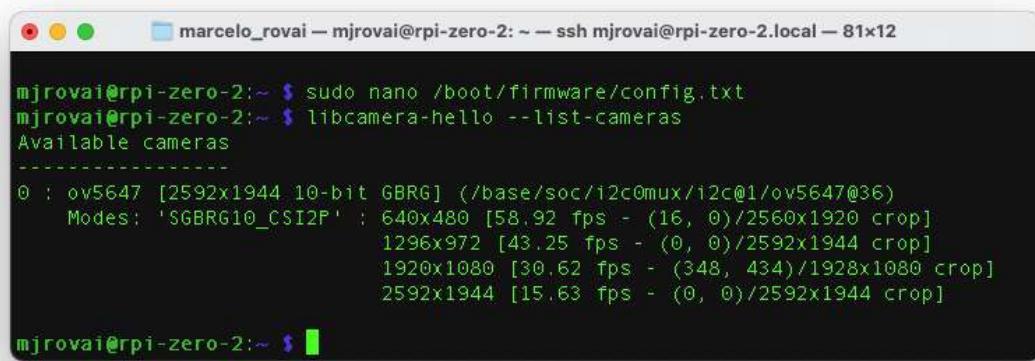
```
dtoverlay=imx219,cam0
```

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:

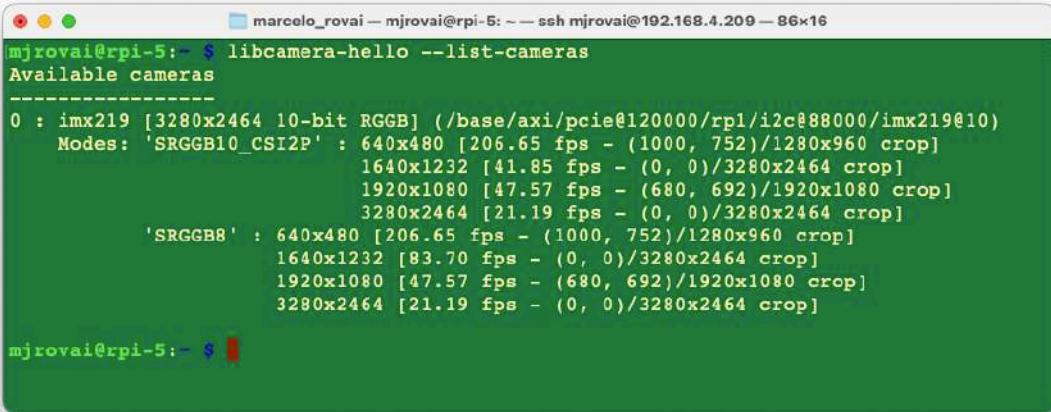
```
rpicam-hello --list-cameras
```



```
marcelo_rovai — mjrovai@rpi-zero-2: ~ — ssh mjrovai@rpi-zero-2.local — 81x12

mjrovai@rpi-zero-2:~ $ sudo nano /boot/firmware/config.txt
mjrovai@rpi-zero-2:~ $ libcamera-hello --list-cameras
Available cameras
-----
0 : ov5647 [2592x1944 10-bit GBRG] (/base/soc/i2c0mux/i2c@1/ov5647@36)
    Modes: 'SGBRG10_CSI2P' : 640x480 [58.92 fps - (16, 0)/2560x1920 crop]
            1296x972 [43.25 fps - (0, 0)/2592x1944 crop]
            1920x1080 [30.62 fps - (348, 434)/1928x1080 crop]
            2592x1944 [15.63 fps - (0, 0)/2592x1944 crop]

mjrovai@rpi-zero-2:~ $
```



```
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)
  Nodes: '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]

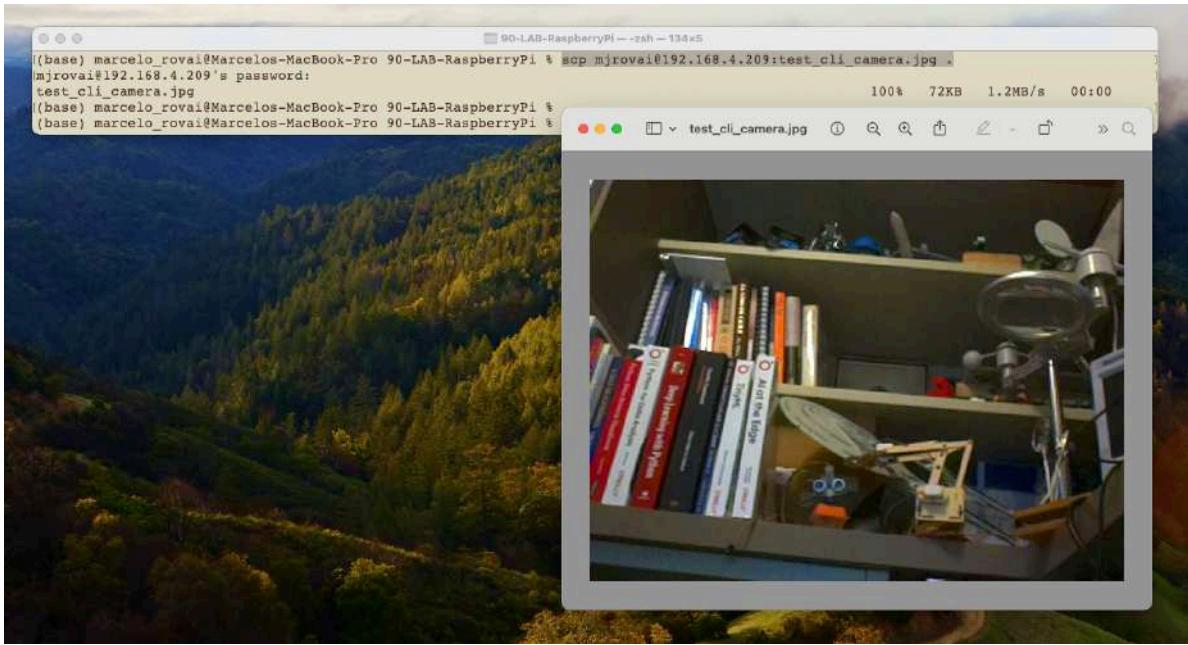
mjrovai@rpi-5: ~ $
```

[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
```

To view the saved file, we should use `ls -f`, which lists all the current directory content in long format. As before, we can use `scp` to view the image:



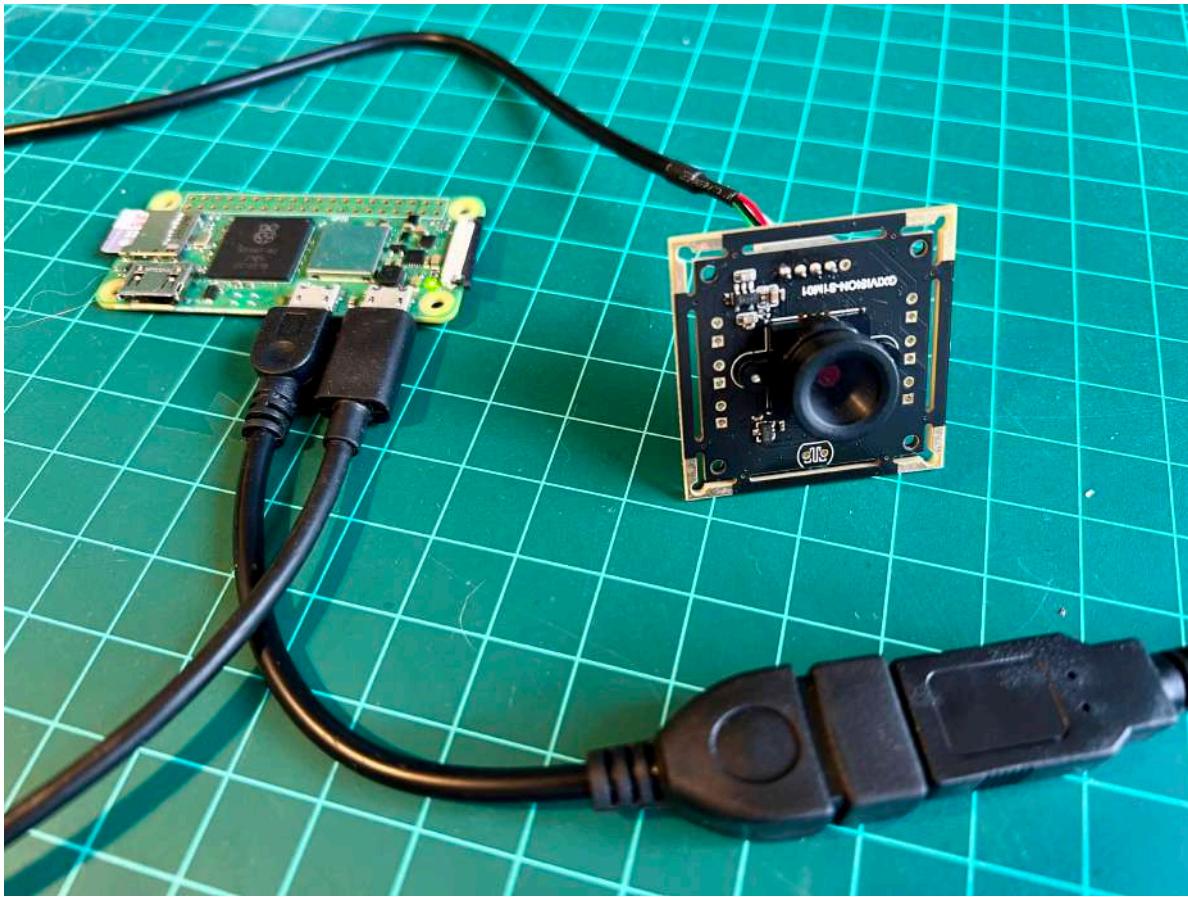
Alternatively, you can transfer it to your desktop using **FileZilla**.

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 Raspberry Pi (in this example, I am using the Raspberry Pi Zero, but the instructions work for all Raspberry Pis).



3. Power on again and run the SSH
4. To check if your USB camera is recognized, run:

```
lsusb
```

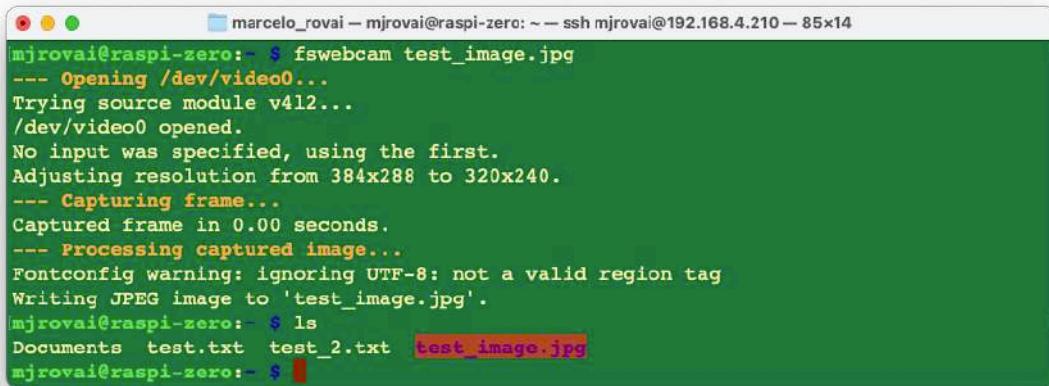
You should see your camera listed in the output.

```
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.



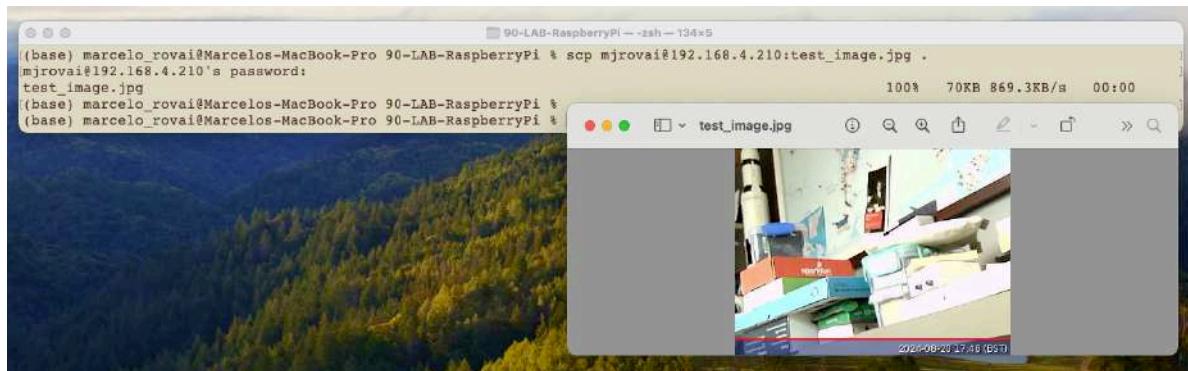
```
mjrovai@raspi-zero:~$ fswebcam test_image.jpg
--- Opening /dev/video0...
Trying source module v4l2...
/dev/video0 opened.
No input was specified, using the first.
Adjusting resolution from 384x288 to 320x240.
--- Capturing frame...
Captured frame in 0.00 seconds.
--- Processing captured image...
Fontconfig warning: ignoring UTF-8: not a valid region tag
Writing JPEG image to 'test_image.jpg'.
mjrovai@raspi-zero:~$ ls
Documents test.txt test_2.txt test_image.jpg
mjrovai@raspi-zero:~$
```

6. Since we are using SSH to connect to our Rapsi, we must transfer the image to our main computer so we can view it. We can use FileZilla or SCP for this:

Open a terminal **on your host computer** and run:

```
scp mjrovai@raspi-zero.local:~/test_image.jpg .
```

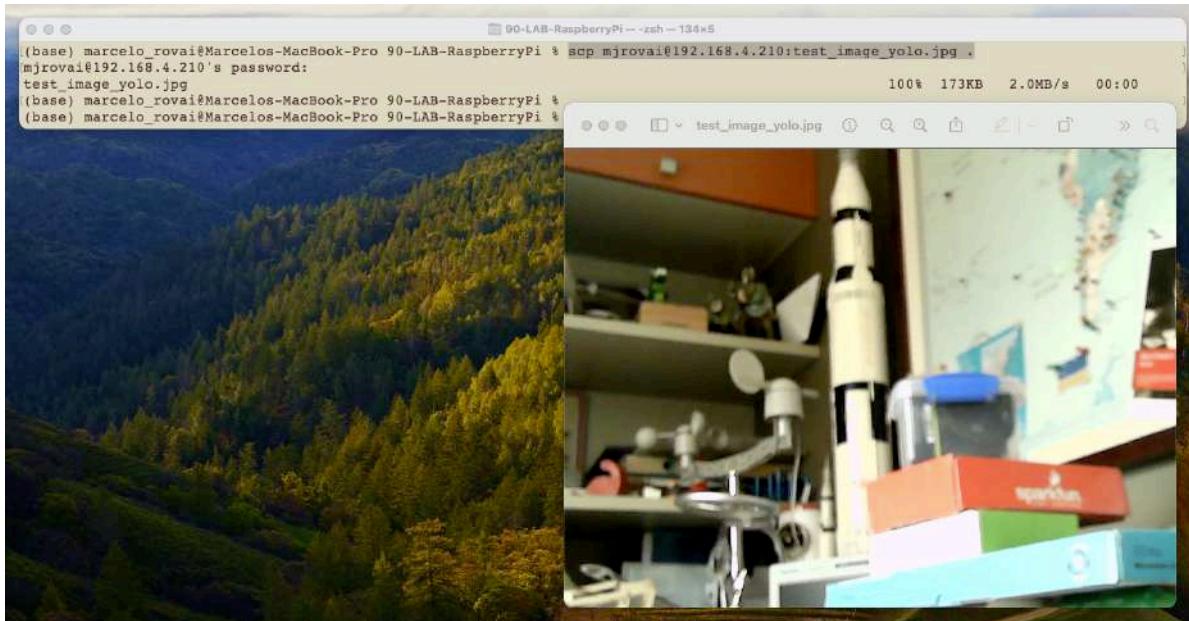
Replace “mjrovai” with your username and “raspi-zero” with Pi’s hostname.



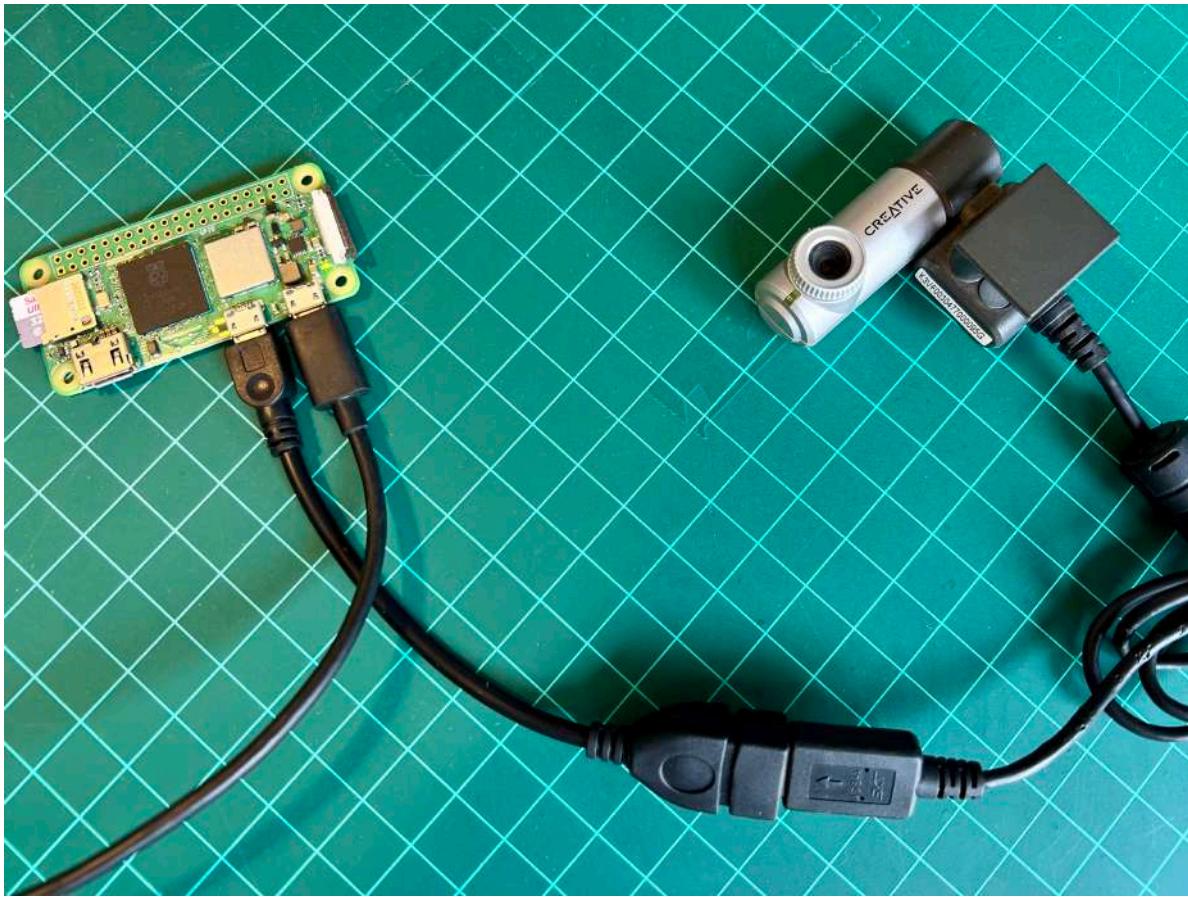
7. If the image quality isn’t satisfactory, you can adjust various settings; for example, define a resolution that is suitable for YOLO (640x640):

```
fswebcam -r 640x640 --no-banner test_image_yolo.jpg
```

This captures a higher-resolution image without the default banner.



An ordinary USB Webcam can also be used:



And verified using lsusb

```
marcelo_rovai — mirovai@raspi-zero: ~ — ssh mirovai@192.168.4.210 — 85x6
mjr0vai@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
mjr0vai@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:

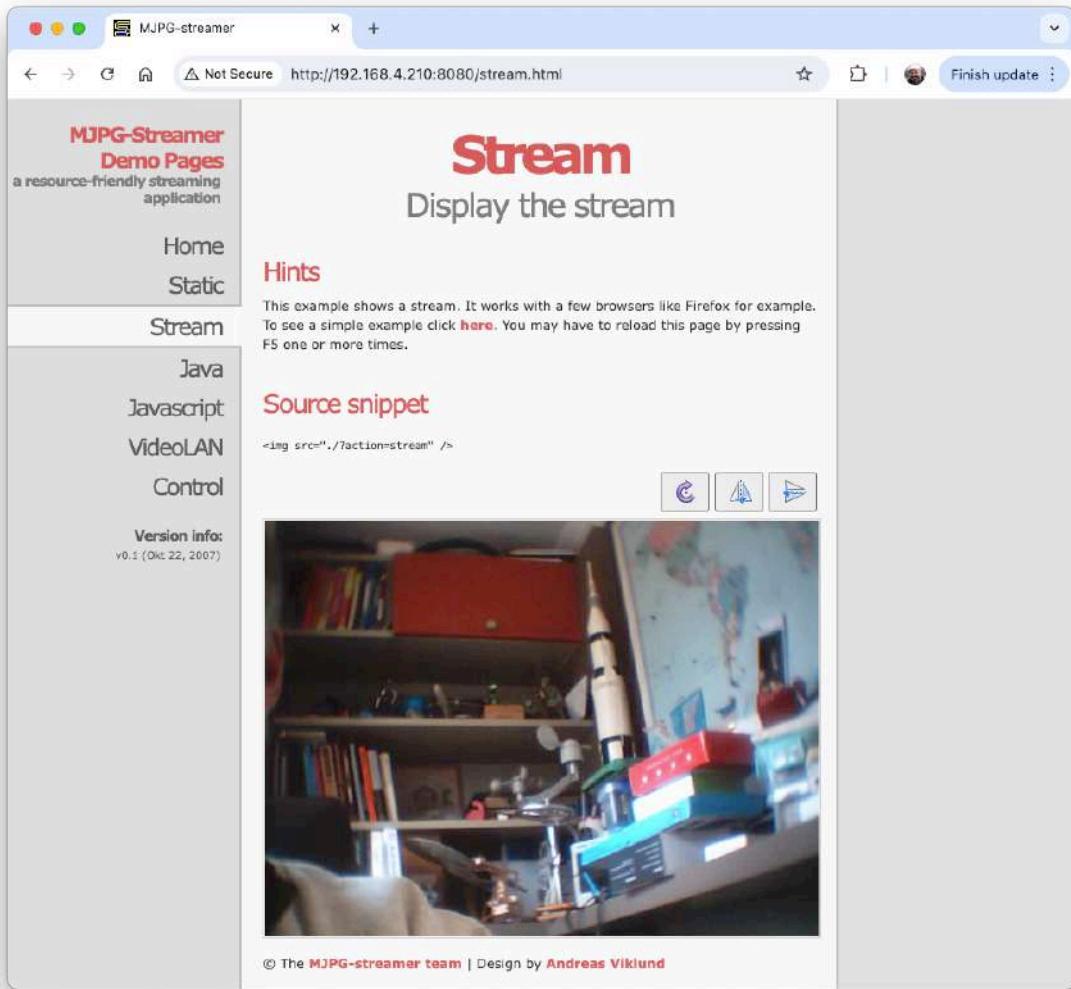
```
mjpg_streamer -i "input_uvc.so" -o "output_http.so -w ./www"
```

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
```



Running the Raspi Desktop remotely

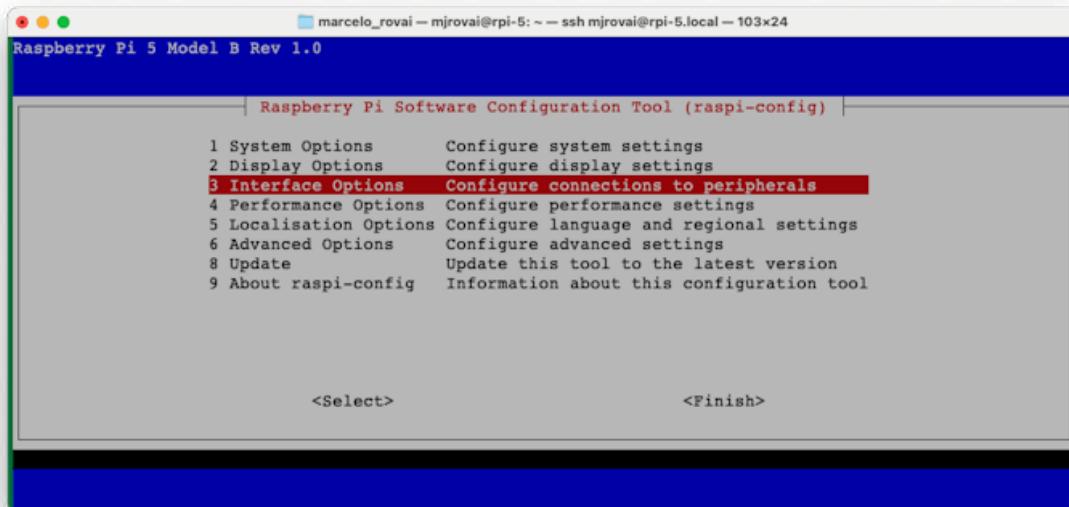
While we've primarily interacted with the Raspberry Pi using terminal commands via SSH, we can access the whole graphical desktop environment remotely if we have installed the complete Raspberry Pi OS (for example, [Raspberry Pi OS \(64-bit\)](#)). This can be particularly useful for tasks that benefit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

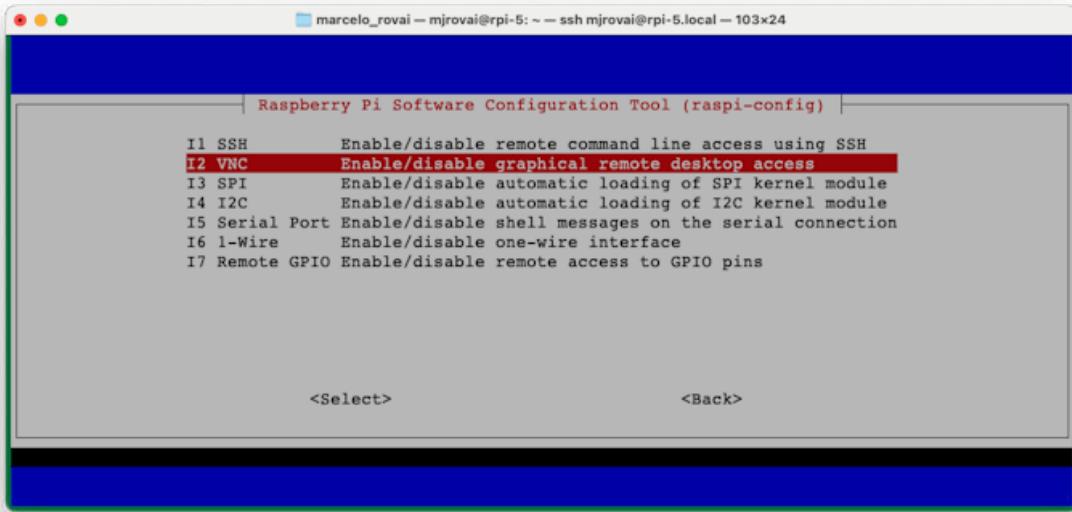
- Connect to your Raspberry Pi via SSH.
- Run the Raspberry Pi configuration tool by entering:

```
sudo raspi-config
```

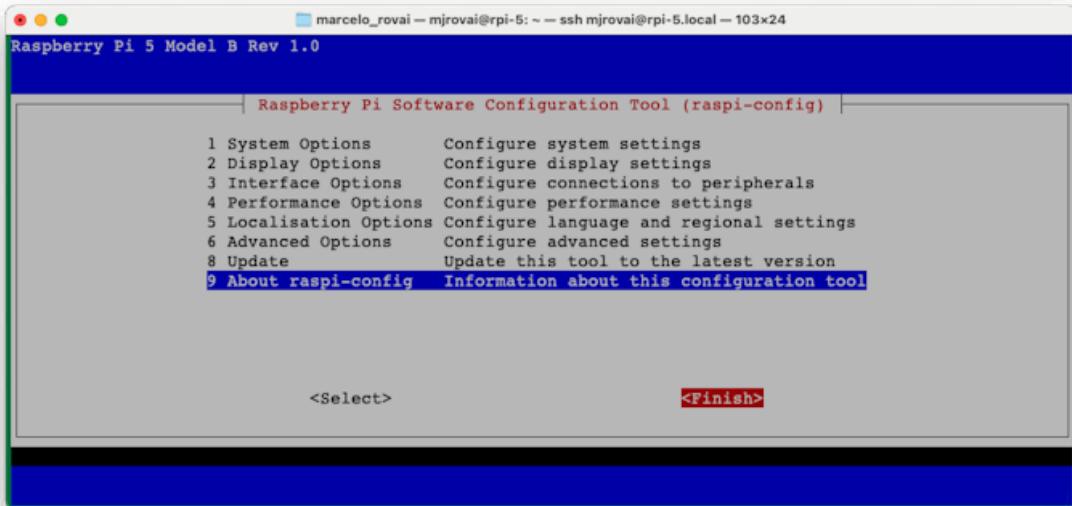
- Navigate to **Interface Options** using the arrow keys.



- Select VNC and Yes to enable the VNC server.



- Exit the configuration tool, saving changes when prompted.



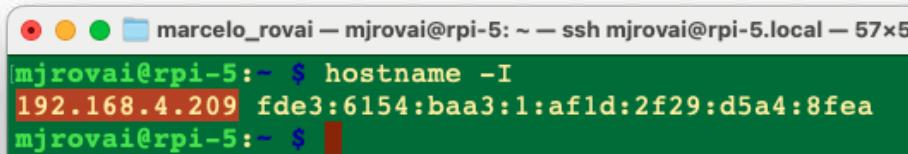
2. Install a VNC Viewer on Your Computer:

- 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.

- Once installed, you should confirm the Raspi IP address. For example, on the terminal, you can use:

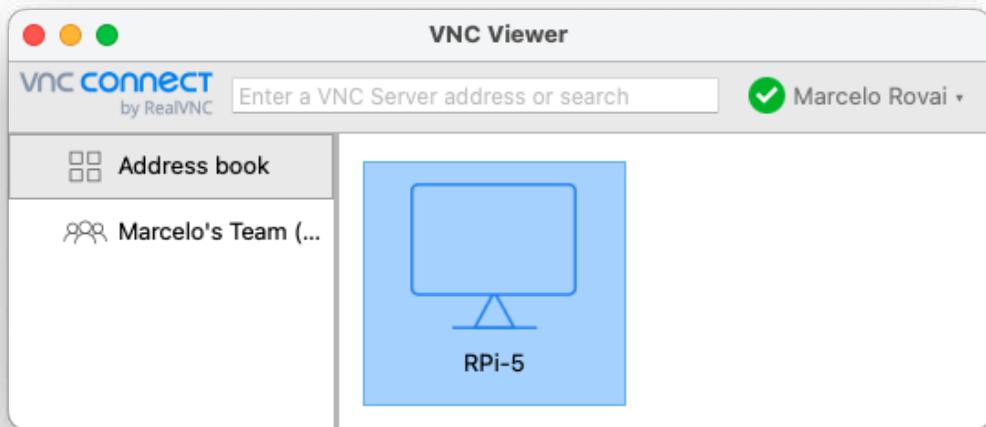
```
hostname -I
```



```
[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:~ $ ]
```

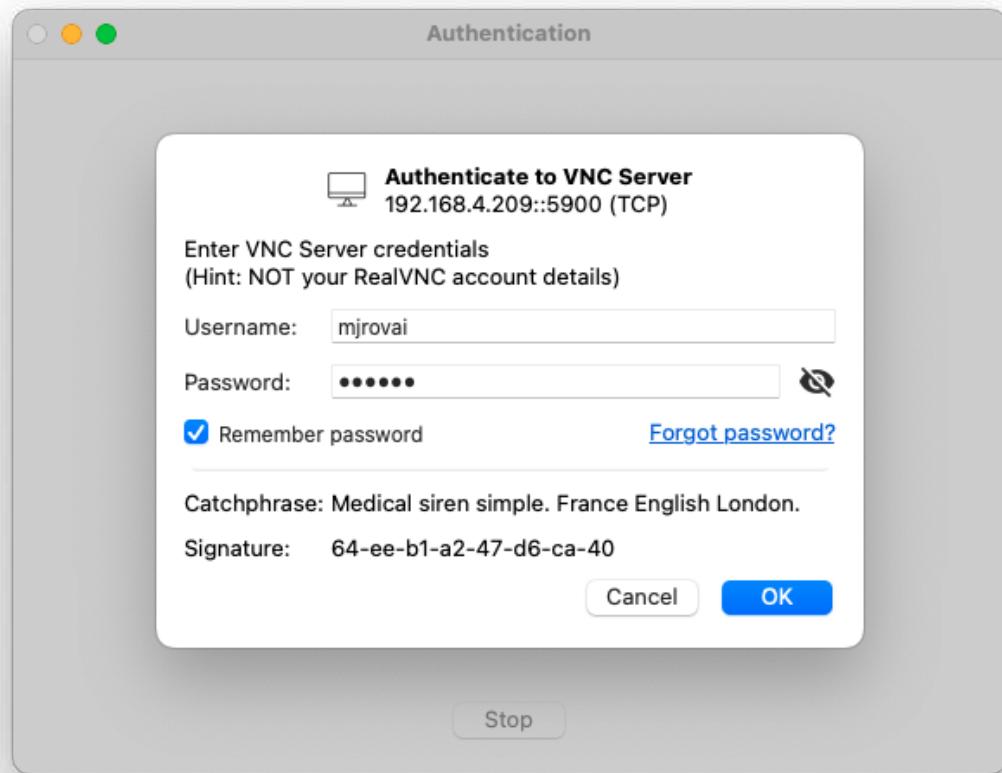
- Connect to Your Raspberry Pi:

- Open your VNC viewer application.

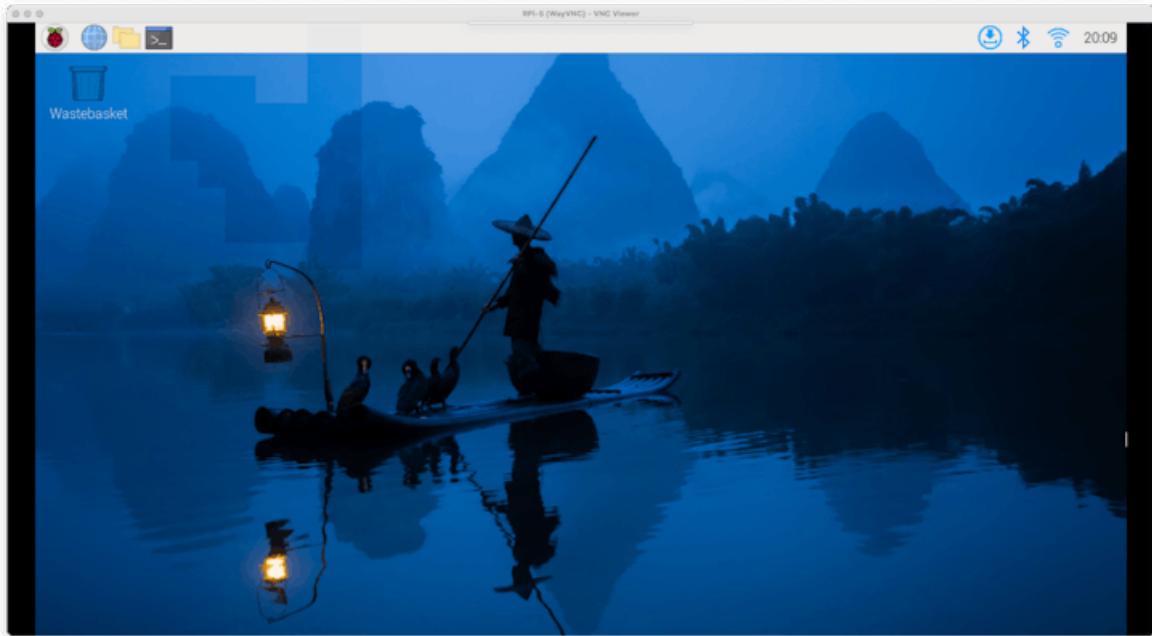


- Enter your Raspberry Pi's IP address and hostname.

- When prompted, enter your Raspberry Pi's username and password.

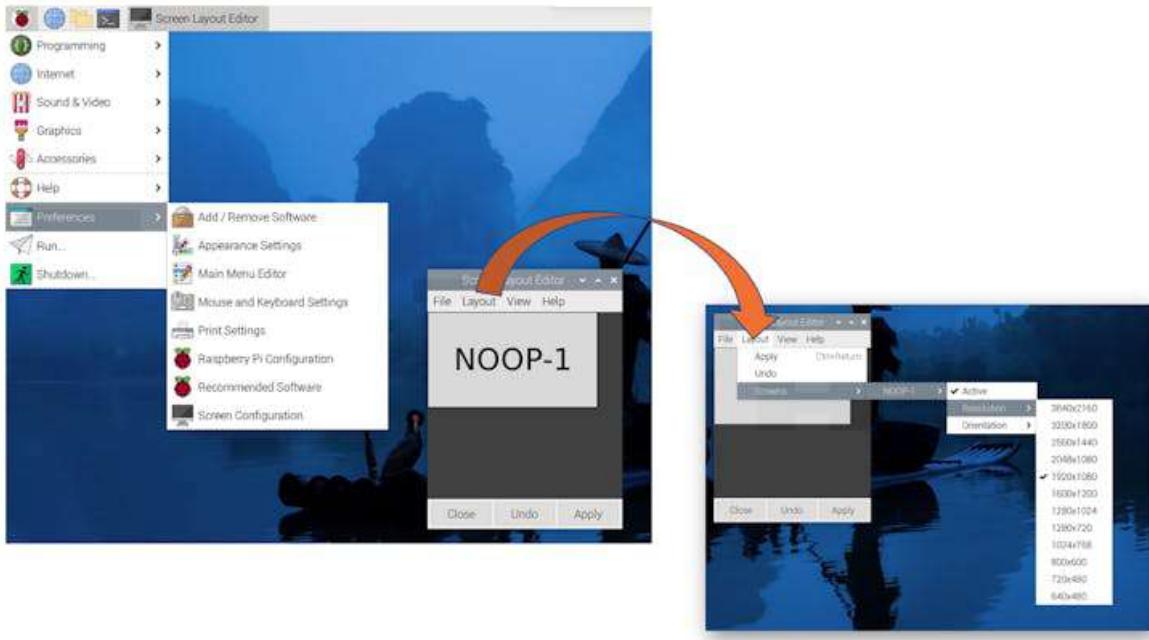


5. The Raspberry Pi 5 Desktop should appear on your computer monitor.



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 Fundamentals

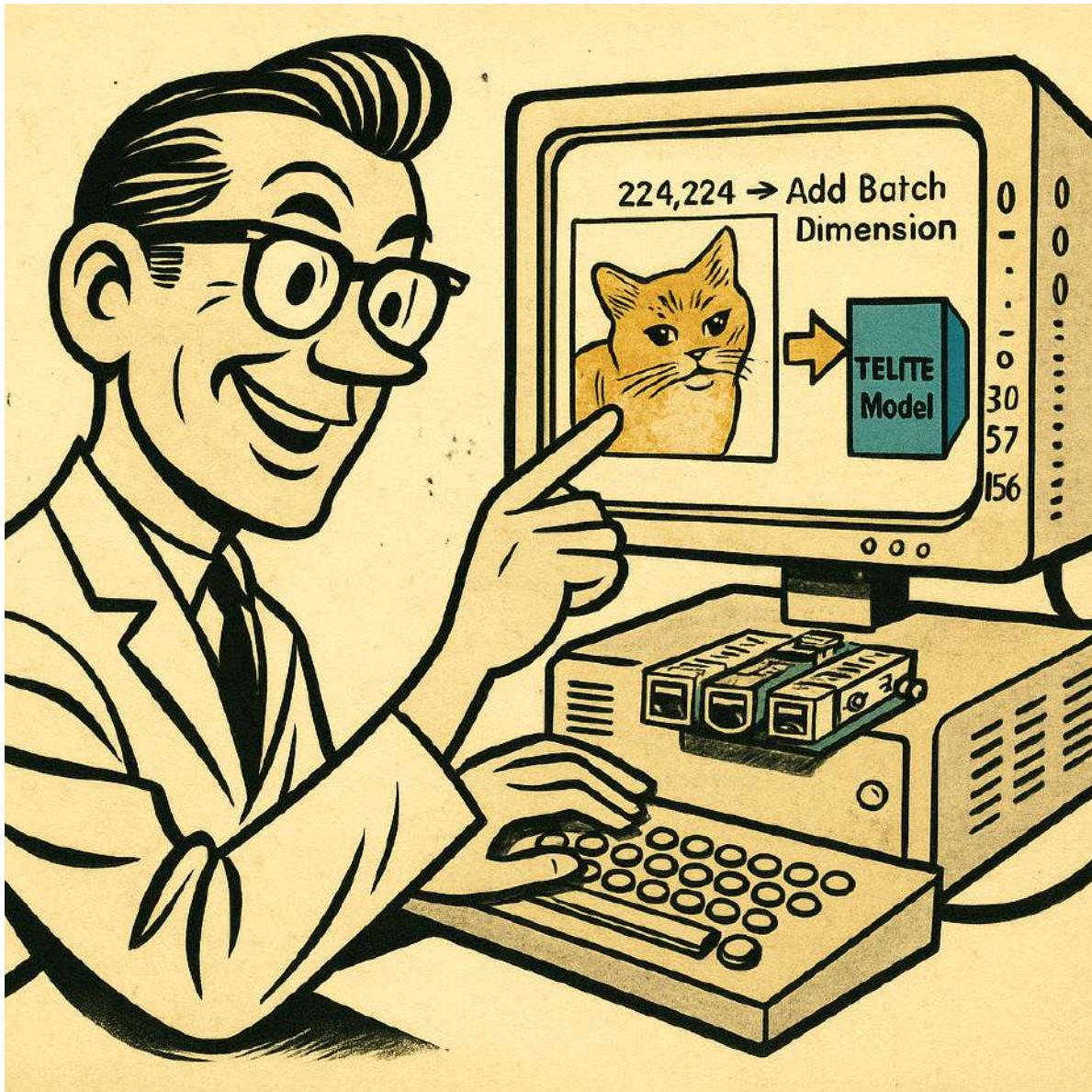


Figure 2: *DALL·E prompt - “Create a Cartoon with style from the 50’s doing Image Classification on a Raspberry Pi - based on the image uploaded.”*

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
sudo reboot # Reboot to ensure all updates take effect
```

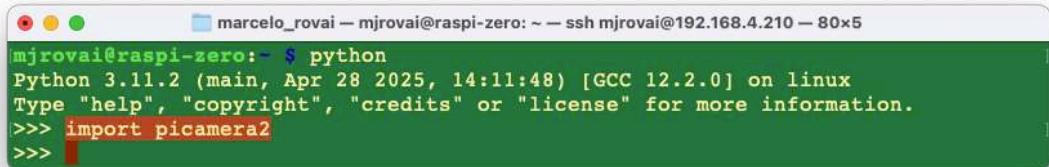
Installing Required System Level Libraries

Install Python tools and camera libraries

```
sudo apt install -y python3-pip python3-venv python3-picamera2
sudo apt install -y libcamera-dev libcamera-tools libcamera-apps
```

[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 Pi 5.

Testing picamera2 Installation



```
marcelo_rovai@raspi-zero: ~ ssh mjrovai@192.168.4.210 - 80x5
[mjrovai@raspi-zero: ~ $ python
Python 3.11.2 (main, Apr 28 2025, 14:11:48) [GCC 12.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import picamera2
>>>
```

Setting up a Virtual Environment

Create a virtual environment with access to system packages to manage dependencies:

```
python3 -m venv ~/tfslite_env --system-site-packages
```

Activate the environment:

```
source ~/tfslite_env/bin/activate
```

To exit the virtual environment, use:

```
deactivate
```

Install Python Packages (inside Virtual Environment)

Ensure you're in the virtual environment (venv)

```
pip install numpy pillow # Image processing
pip install matplotlib # For displaying images
pip install opencv-python # Computer vision
```

Verify installation

```
pip list | grep -E "(numpy|pillow|opencv|picamera)"
```

```

marcelo_roval - mjrovai@raspi-zero: ~ ssh mjrovai@192.168.4.210 - 95x7
(tfelite_env) mjrovai@raspi-zero:~$ pip list | grep -E "(numpy|pillow|opencv|picamera)"
numpy                1.24.2
opencv-python         4.12.0.88
picamera2             0.3.30
pillow               11.3.0
(tfelite_env) mjrovai@raspi-zero:~$ 

```

The proven Numpy version should be 1.24.2. If you have another version installed, first uninstall it.

```
pip uninstall numpy
```

Install `version 1.24.2`, which is compatible with the `tfelite_runtime` and `picamera`.

```
pip3 install numpy==1.24.2
```

System vs pip Package Installation Rule

Use `sudo apt install` (outside of venv) for:

- System-level dependencies and libraries
- Hardware interface libraries (as a camera)
- Development headers and build tools
- Anything that needs to interface directly with hardware

Use `pip install` (inside venv) for:

- Pure Python packages
- Application-specific libraries
- Packages that don't need system-level access

Rule of thumb: Use `sudo apt install` only for system dependencies and hardware interfaces. Use `pip install` (without sudo) inside an activated virtual environment for everything else. Inside the vent, PIP or PIP3 are the same.

The virtual environment will automatically see both the system packages and the pip-installed packages thanks to the `--system-site-packages` flag.

Setting up Jupyter Notebook

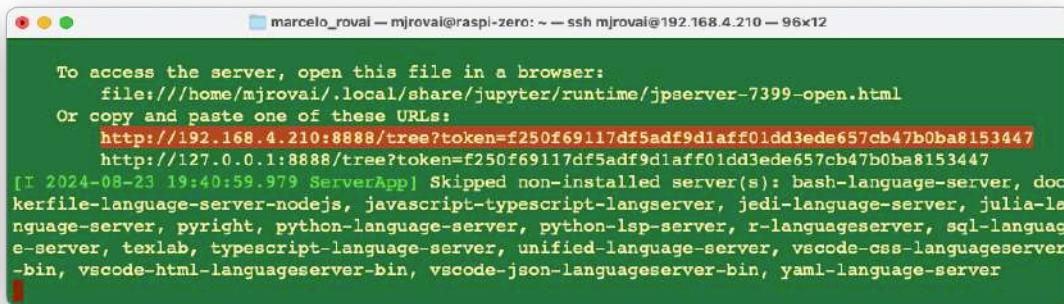
Let's set up Jupyter Notebook optimized for headless Raspberry Pi camera work and development:

```
pip install jupyter jupyterlab notebook  
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:

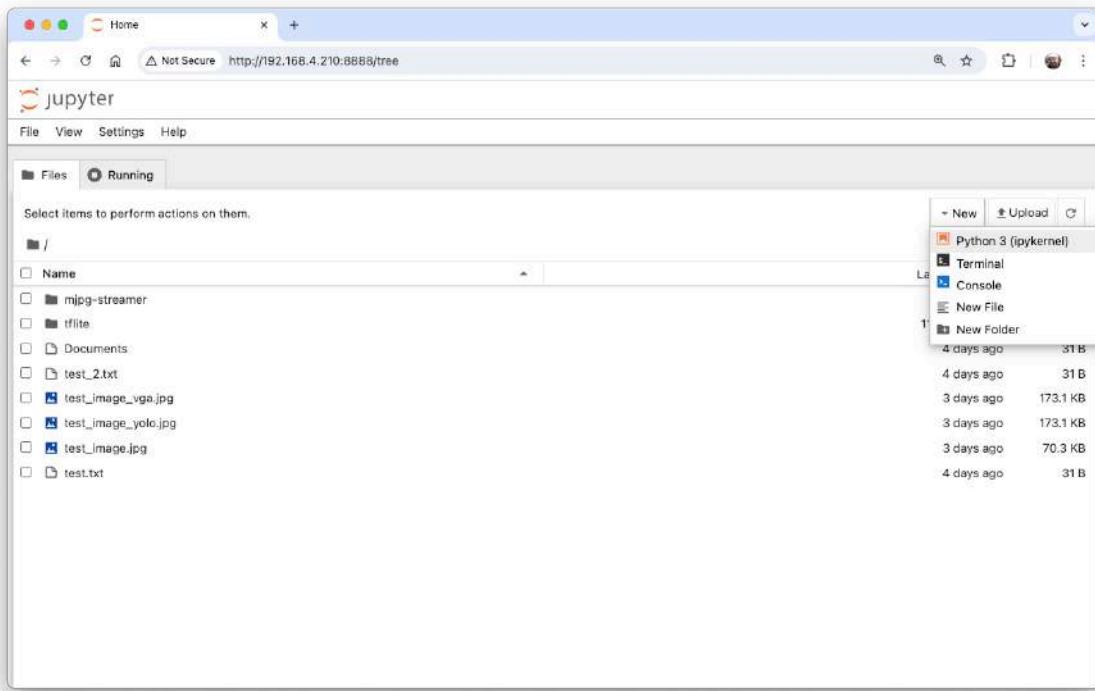


The screenshot shows a terminal window with the following text output:

```
marcelo_royai — 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
[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 the working directory in the Raspi and create a new Python 3 notebook. For example:

```
cd Documents  
mkdir Python
```

It is possible to create folders directly in the Jupyter Notebook

Create a new Notebook and test the code below:

Import Libraries

```
import time  
import numpy as np  
from PIL import Image  
import matplotlib.pyplot as plt  
from picamera2 import Picamera2
```

Load an image from the internet, for example (note that it is possible to run a command line from the Notebook, using ! before the command):

```
!wget https://upload.wikimedia.org/wikipedia/commons/3/3a/Cat03.jpg
```

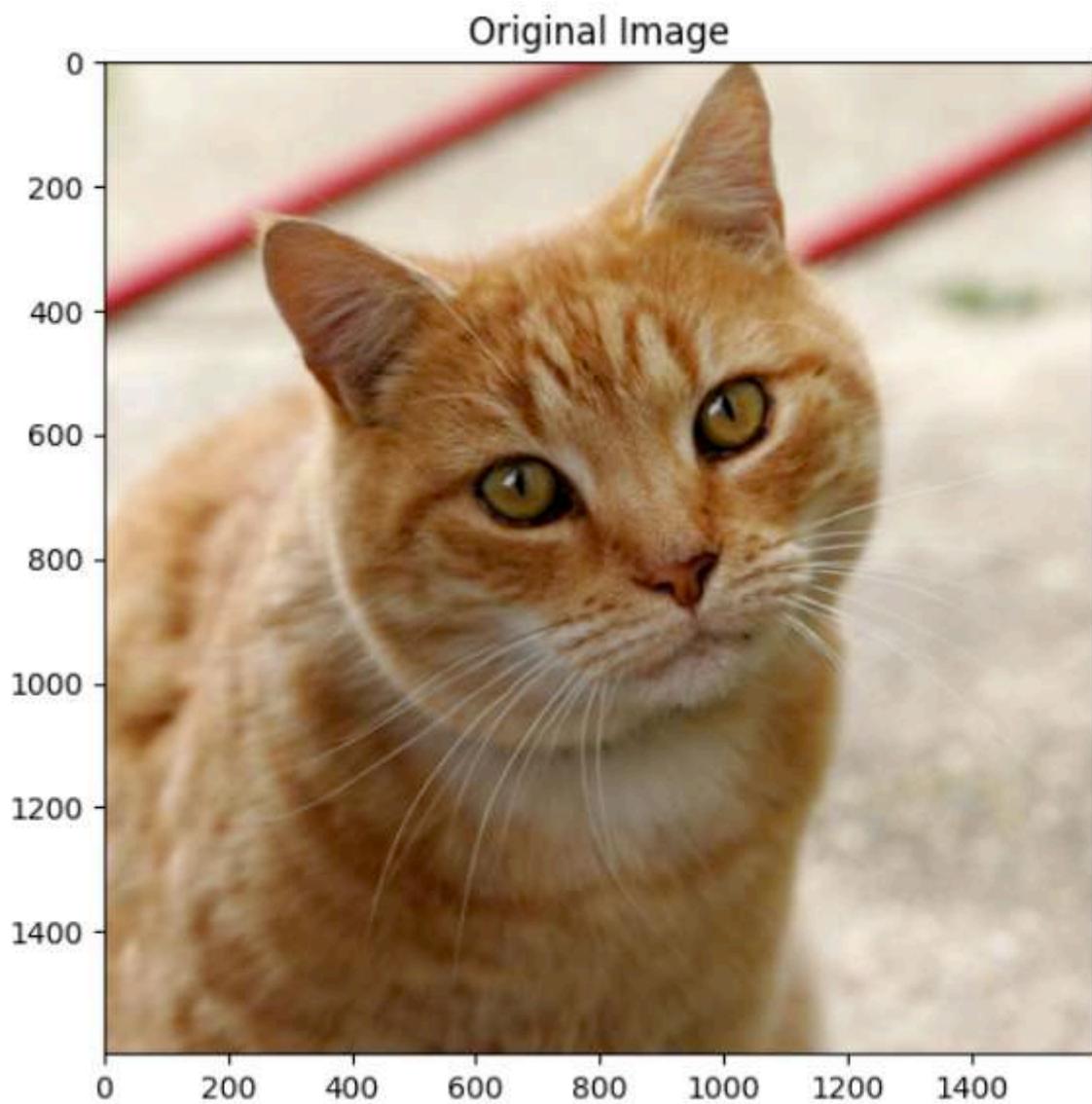
An image (`Cat03.jpg`) will be downloaded to the current directory.

Load and show the image:

```
img_path = "Cat03.jpg"
img = Image.open(img_path)

# Display the image
plt.figure(figsize=(6, 6))
plt.imshow(img)
plt.title("Original Image")
plt.show()
```

We can see the image displayed on the Notebook:



Now, let's use the camera to capture a local image:

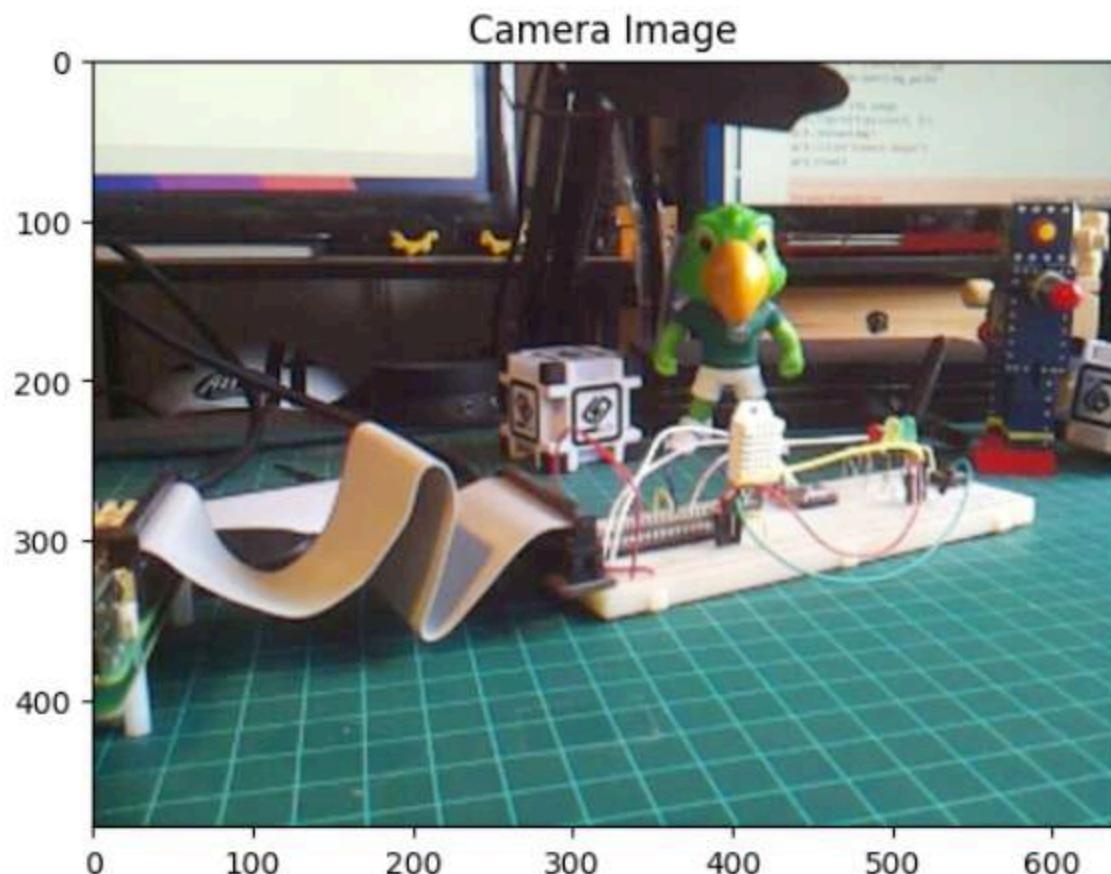
```
from picamera2 import Picamera2  
import time  
  
# Initialize camera  
picam2 = Picamera2()  
picam2.start()
```

```
# Wait for camera to warm up
time.sleep(2)

# Capture image
picam2.capture_file("class3_test.jpg")
print("Image captured: class3_test.jpg")

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

And use a similar code as before to show it (adapting the `img_path` and `title`):



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
```

The wheel installed: `tflite_runtime-2.14.0-cp311-cp311-manylinux_2_34_aarch64.whl`

Reboot the system:

```
sudo reboot
```

Verify instalation:

```
pip list | grep -E "(tflite-runtime)"
```

We should see as a response: `tflite-runtime 2.14.0`.

Creating a working directory:

If you are working on the Raspi-Zero with the minimum OS (No Desktop), you do 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:

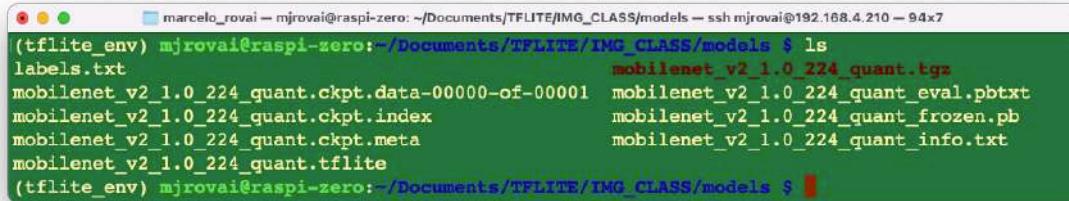
An appropriate pre-trained model is crucial for successful image classification on resource-constrained devices like the Raspberry Pi. **MobileNet** is designed for mobile and embedded vision applications with a good balance between accuracy and speed. Versions: MobileNetV1, MobileNetV2, MobileNetV3. 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://raw.githubusercontent.com/Mjrovai/EdgeML-with-Raspberry-Pi/refs/heads/main/IM...
```

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



```
marcelo_rovai@mjrovai@raspi-zero:~/Documents/TFLITE/IMG_CLASS/models$ ls  
(tflite_env) marcelo_rovai@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_eval.pbtxt  
mobilenet_v2_1.0_224_quant_frozen.pb  
mobilenet_v2_1.0_224_quant_info.txt  
(tflite_env) marcelo_rovai@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`. We can delete the other files.

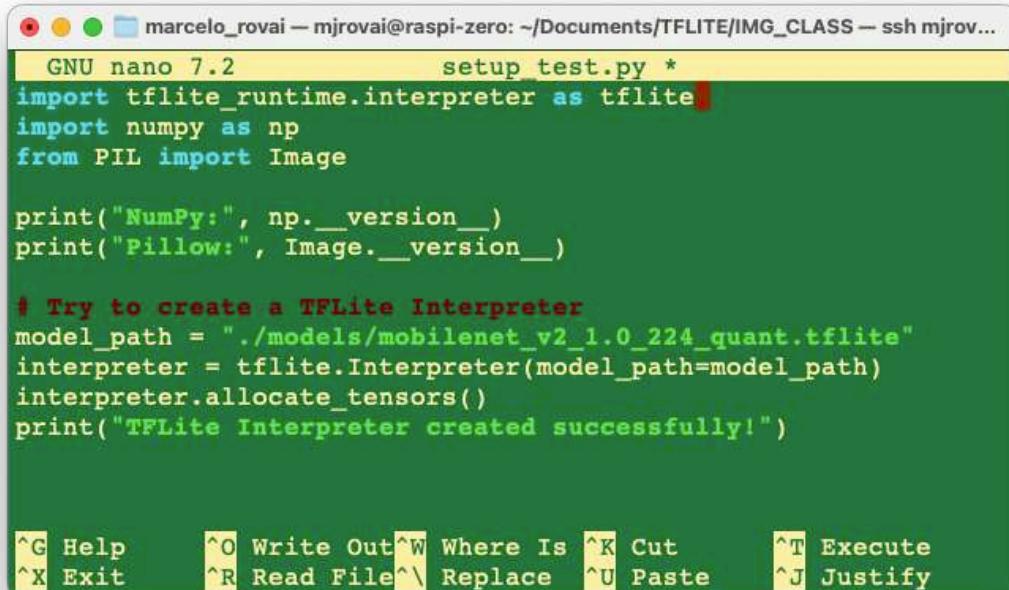
Verifying the Setup

Let's test our 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 = tflite.Interpreter(model_path=model_path)
interpreter.allocate_tensors()
print("TFLite Interpreter created successfully!")
```

We 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 tflite_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!")

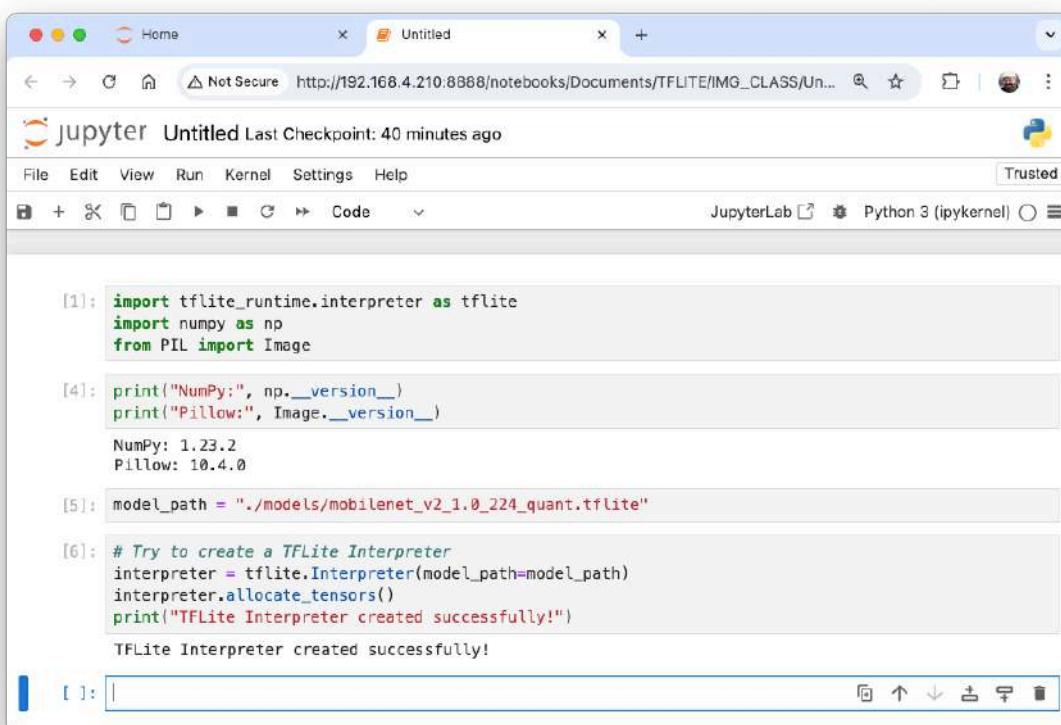
^G Help      ^O Write Out ^W Where Is ^K Cut      ^T Execute
^X Exit     ^R Read File ^\ Replace   ^U Paste    ^J Justify
```

And run it with the command:

```
python setup_test.py
```

```
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 window titled "Untitled". The URL in the browser bar is "http://192.168.4.210:8888/notebooks/Documents/TFLITE/IMG_CLASS/Untitled.ipynb". The notebook contains the following 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"

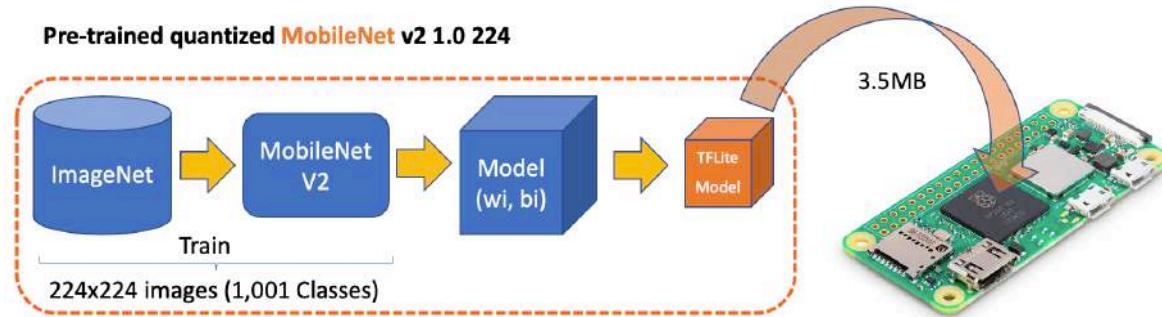
[6]: # Try to create a TFLite Interpreter
interpreter = tf.lite.Interpreter(model_path=model_path)
interpreter.allocate_tensors()
print("TFLite Interpreter created successfully!")

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.



In the `IMG_CLASS` working directory, let's start a new [notebook](#) to follow all the steps to classify one image:

Import the needed libraries:

```
import time
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import tensorflow as tf
```

Load the TFLite model and allocate tensors:

```
model_path = "./models/mobilenet_v2_1.0_224_quant.tflite"
interpreter = tf.lite.Interpreter(model_path=model_path)
interpreter.allocate_tensors()
```

Get input and output tensors.

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

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).

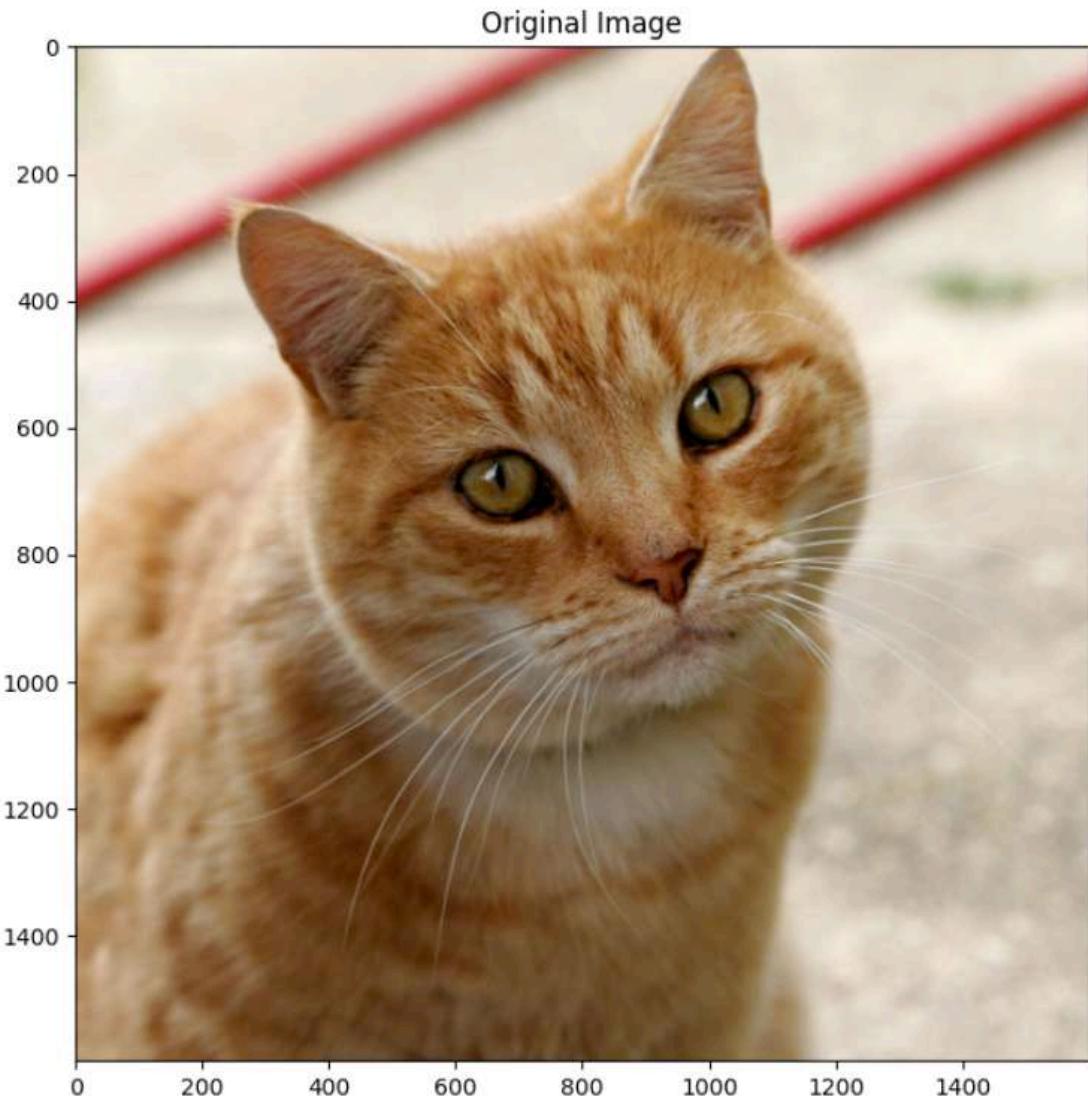
Let's get a test image. We can either transfer it from our computer or download one for testing, as we did before. Let's first create a folder under our working directory:

```
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)

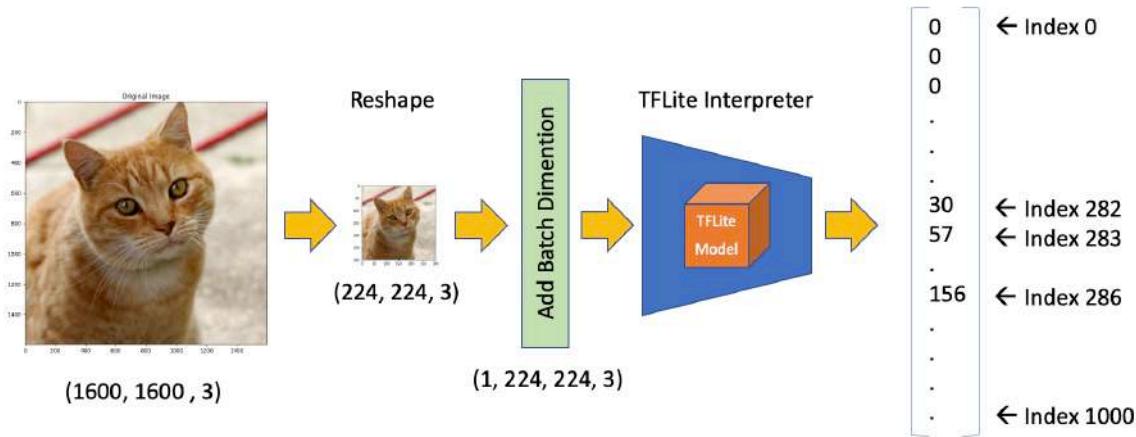
# Display the image
plt.figure(figsize=(8, 8))
plt.imshow(img)
plt.title("Original Image")
plt.show()
```



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. To use our model, we should reshape it to (224, 224, 3) and add a batch dimension of 1, as defined in the input details: (1, 224, 224, 3). The inference result, as shown in the output details, will be an array of size 1001, 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 four of the top indices are related to felines. The **prediction** content is the probability associated with each one of the labels. As we saw in the **output details**, those values are quantized and should be dequantized:

```
scale, zero_point = output_details[0]['quantization']
dequantized_output = (predictions.astype(np.float32) - zero_point) * scale
dequantized_output

array([-0.8900329, -0.4944627,  1.285603, ..., -1.9778508,  1.5822806,
       4.549057], dtype=float32)
```

The output (positive and negative numbers) shows that the output probably does not have a Softmax. Checking the model documentation (<https://arxiv.org/abs/1801.04381v4>): MobileNet V2 typically doesn't include a softmax layer at the output. It usually ends with a

1x1 convolution followed by average pooling and a fully connected layer. So, for getting the probabilities (0 to 1), we should apply Softmax:

```
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 to 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]

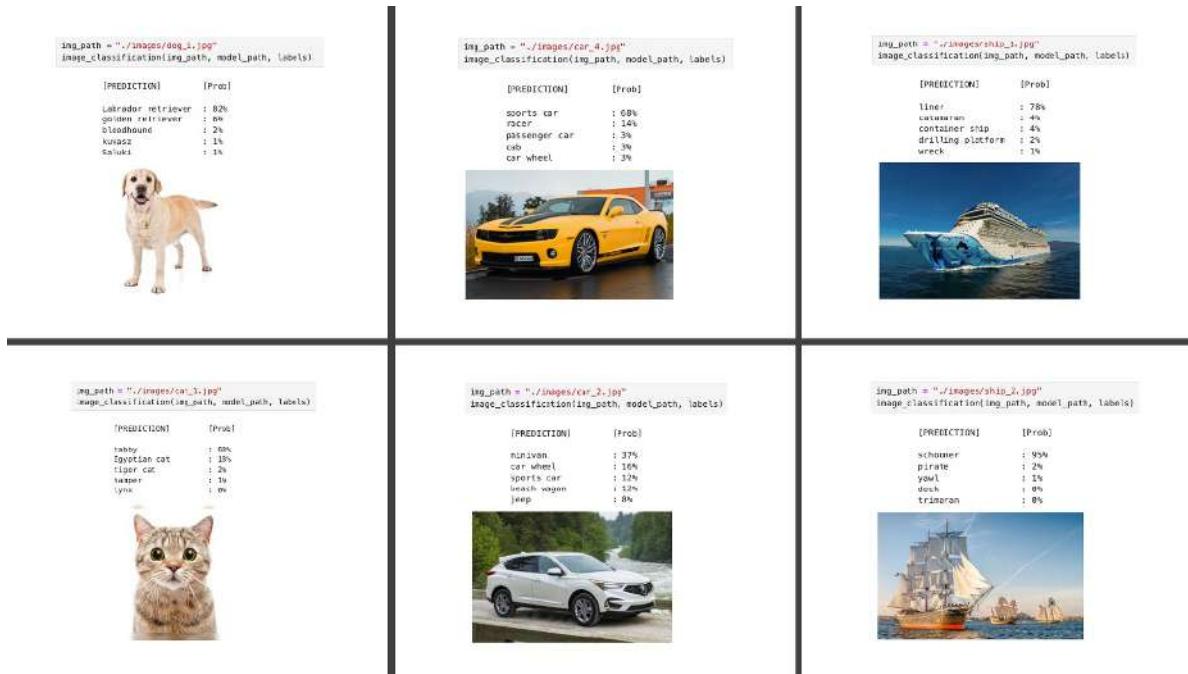
# Get quantization parameters
scale, zero_point = output_details[0]['quantization']

# 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 the classification with the Camera

Let's modify the Python script used before to capture an image from the camera (size: 224x224), saving it in the images folder:

```
from picamera2 import Picamera2
import time

def capture_image(image_path):

    # Initialize camera
    picam2 = Picamera2() # default is index 0

    # Configure the camera
    config = picam2.create_still_configuration(main={"size": (224, 224)})
    picam2.configure(config)
    picam2.start()

    # Wait for camera to warm up
    time.sleep(2)
```

```
# Capture image
picam2.capture_file(image_path)
print("Image captured: "+"image_path")

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

Now, let's capture an image and sent it for inference:

```
img_path = './images/cam_img_test.jpg'
model_path = "./models/mobilenet_v2_1.0_224_quant.tflite"
labels = load_labels("./models/labels.txt")
capture_image(img_path)
image_classification(img_path, model_path, labels, top_k_results=5)
```

[PREDICTION]	[Prob]
coffee mug	: 99%
cup	: 0%
whiskey jug	: 0%
teapot	: 0%
water jug	: 0%



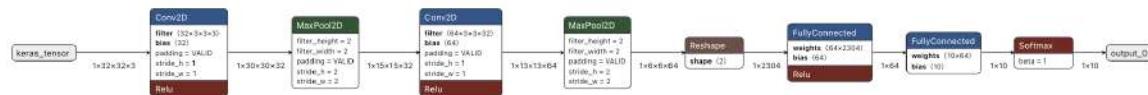
Exploring a Model Trained from Zero

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

CNN to classify Cifar-10 dataset

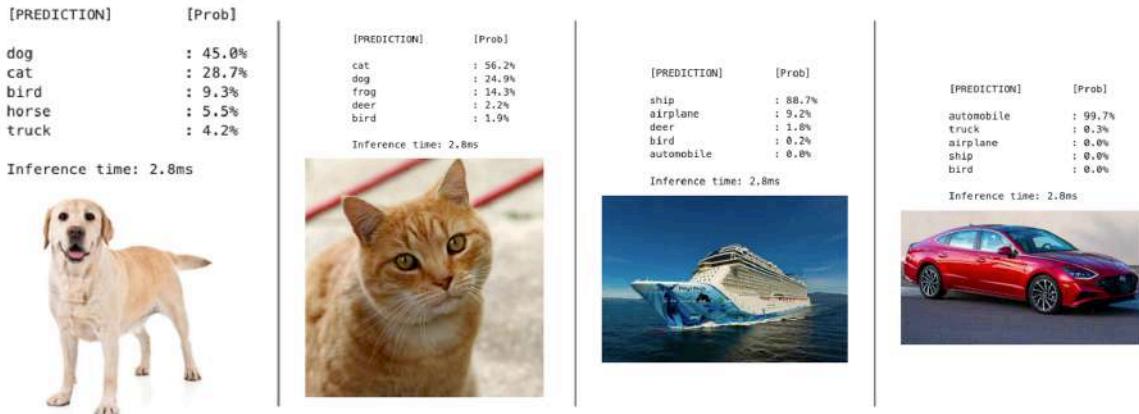
In the notebook, we trained a model using the CIFAR10 dataset, which contains 60,000 images from 10 classes of CIFAR (*airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck*). CIFAR has 32x32 color images (3 color channels) where the objects are not centered and can have the object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



Runing the notebook [20_Cifar_10_Image_Classification.ipynb](#) with the trained model: *cifar10.tflite*, and following the same steps as we did with the MobileNet model, we can see that the inference result is inferior in terms of probability when compared with the MobileNetV2.

Below are examples of images using the *General Function for Image Classification* on a Raspberry Pi Zero, as shown in the last section.



Conclusion:

This chapter has established a solid foundation for understanding and implementing image classification on Raspberry Pi devices using Python and TensorFlow Lite. Throughout this

journey, we have explored the essential components that make edge-based computer vision both practical and powerful.

We began by understanding the theoretical foundations of image classification and its real-world applications across diverse sectors, from healthcare to environmental monitoring. The advantages of running classification on edge devices like the Raspberry Pi—including low latency, offline functionality, enhanced privacy, and cost-effectiveness—make it an attractive solution for many practical applications.

The hands-on experience of setting up the development environment provided crucial insights into the requirements and constraints of embedded systems. We successfully configured TensorFlow Lite runtime, installed essential Python libraries, and established a working directory structure that serves as the foundation for computer vision projects.

Working with the pre-trained MobileNet V2 model demonstrated several key concepts:

- **Model Architecture Understanding:** We explored how pre-trained models like MobileNet V2 are optimized for mobile and embedded applications, achieving an excellent balance between accuracy and computational efficiency.
- **Quantization Benefits:** The 3.5MB quantized model showed how compression techniques make sophisticated neural networks feasible on resource-constrained devices without significant accuracy loss.
- **Inference Pipeline:** We implemented the complete inference workflow, from image pre-processing (resizing to 224×224 , handling data types) to post-processing (dequantization, softmax application, and top-k prediction extraction).
- **Performance Considerations:** The chapter highlighted the importance of understanding model input/output specifications, memory management, and the trade-offs between accuracy and speed on edge devices.

The practical implementation using cameras connected to a Raspberry Pi demonstrated the seamless integration between hardware and software components. The ability to capture images directly from the device and perform real-time classification showcases the potential for autonomous and IoT applications.

Key technical achievements include:

- Successfully setting up TensorFlow Lite runtime and dependencies
- Implementing proper image preprocessing pipelines
- Understanding quantized model operations and dequantization processes
- Creating reusable functions for image classification tasks
- Integrating camera capture with inference workflows

This foundational knowledge prepares us for more advanced topics, including custom model training and deployment. The skills developed here—understanding model architectures, implementing inference pipelines, and working with embedded Python environments—are transferable to a wide range of computer vision applications.

The chapter serves as a stepping stone toward building more sophisticated AI systems on edge devices, demonstrating that powerful computer vision capabilities are accessible even on modest hardware platforms when properly optimized and implemented.

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](#)

Custom Image Classification Project

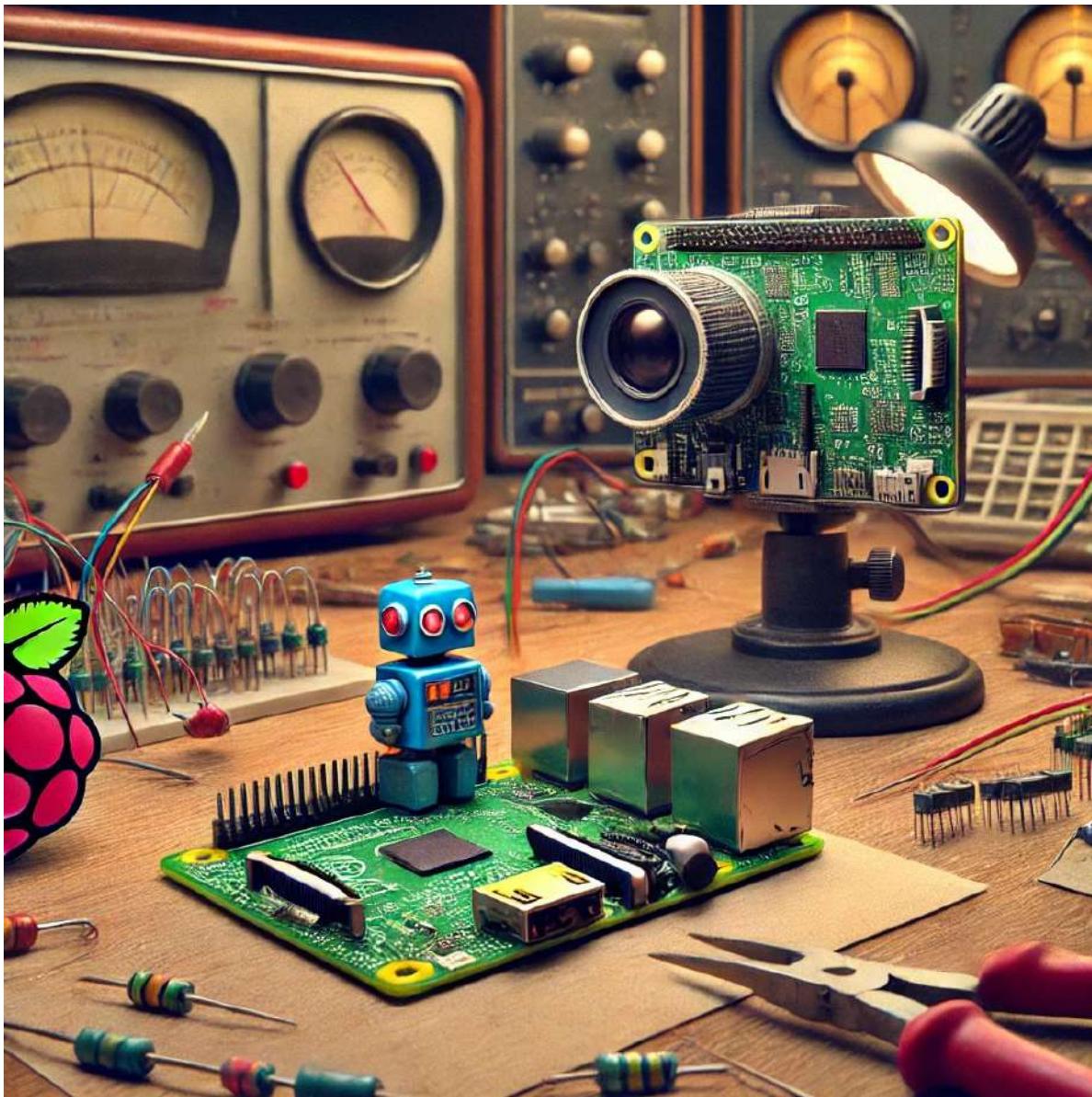
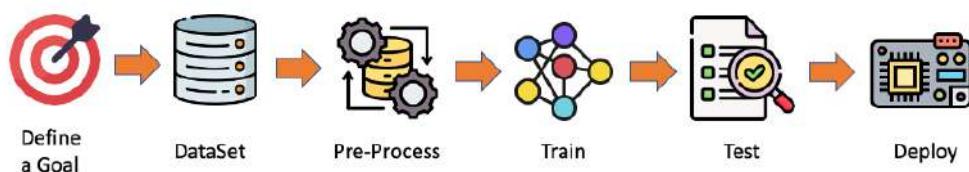


Figure 3: *DALL · E prompt - A cover image for an ‘Image Classification’ chapter in a Raspberry Pi tutorial, designed in the same vintage 1950s electronics lab style as previous covers. The scene should feature a Raspberry Pi connected to a camera module, with the camera capturing a photo of the small 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.*

Image Classification Project

In this chapter, we will develop a complete Image Classification project using the Edge Impulse Studio. As we did with the MobiliNet V2, the trained and converted TFLite model will be used for inference using a Python script.

Here a typical ML workflow that we will use in our project:



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. Go to the working folder (IMG_CLASS) and 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>
        </body>
    ''')

```

```


<div id="shutdown-message" style="display: none;
color: red;">
    Capture process has been stopped.
    You can close this window.
</div>
<form action="/capture_image" method="post">
    <input type="submit" value="Capture Image">
</form>
<form action="/stop" method="post">
    <input type="submit" value="Stop Capture"
style="background-color: #ff6666;">
</form>
<form action="/" method="get">
    <input type="submit" value="Change Label"
style="background-color: #ffff66;">
</form>
</body>
</html>
'', label=current_label, capture_count=capture_counts.get(
current_label, 0))

@app.route('/video_feed')
def video_feed():
    return Response(generate_frames(),
                    mimetype='multipart/x-mixed-replace;
boundary=frame')

@app.route('/capture_image', methods=['POST'])
def capture_image():
    global capture_counts
    if current_label and not shutdown_event.is_set():
        capture_counts[current_label] += 1
        timestamp = time.strftime("%Y%m%d-%H%M%S")
        filename = f"image_{timestamp}.jpg"
        full_path = os.path.join(base_dir, current_label,
                               filename)

        picam2.capture_file(full_path)

```

```

        return redirect(url_for('capture_page'))


@app.route('/stop', methods=['POST'])
def stop():
    summary = render_template_string('''
        <!DOCTYPE html>
        <html>
        <head>
            <title>Dataset Capture - Stopped</title>
        </head>
        <body>
            <h1>Dataset Capture Stopped</h1>
            <p>The capture process has been stopped.
                You can close this window.</p>
            <p>Summary of captures:</p>
            <ul>
                {% for label, count in capture_counts.items() %}
                    <li>{{ label }}: {{ count }} images</li>
                {% endfor %}
            </ul>
        </body>
        </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)

```

3. Run this script:

```
python3 get_img_data.py
```

4. Access the web interface:

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

This Python script creates a web-based interface for capturing and organizing image datasets using a Raspberry Pi and its camera. It's handy for machine learning projects that require labeled image data.

Key Features:

1. **Web Interface:** Accessible from any device on the same network as the Raspberry Pi.
2. **Live Camera Preview:** This shows a real-time feed from the camera.
3. **Labeling System:** Allows users to input labels for different categories of images.
4. **Organized Storage:** Automatically saves images in label-specific subdirectories.
5. **Per-Label Counters:** Keeps track of how many images are captured for each label.
6. **Summary Statistics:** Provides a summary of captured images when stopping the capture process.

Main Components:

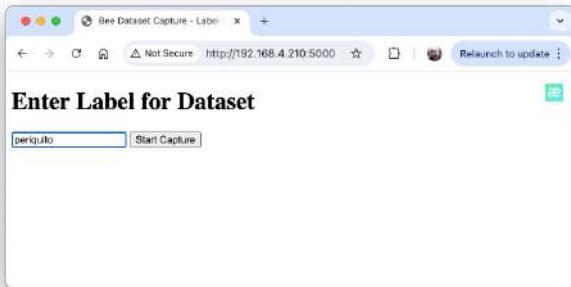
1. **Flask Web Application:** Handles routing and serves the web interface.
2. **Picamera2 Integration:** Controls the Raspberry Pi camera.
3. **Threaded Frame Capture:** Ensures smooth live preview.
4. **File Management:** Organizes captured images into labeled directories.

Key Functions:

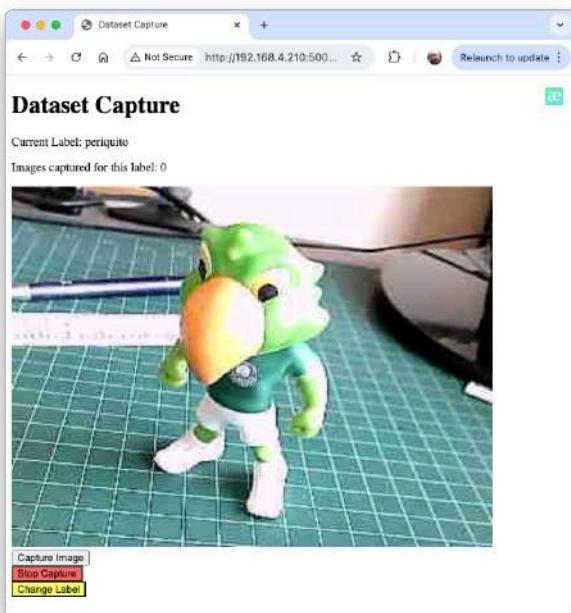
- `initialize_camera()`: Sets up the Picamera2 instance.
- `get_frame()`: Continuously captures frames for the live preview.
- `generate_frames()`: Yields frames for the live video feed.
- `shutdown_server()`: Sets the shutdown event, stops the camera, and shuts down the Flask server
- `index()`: Handles the label input page.
- `capture_page()`: Displays the main capture interface.
- `video_feed()`: Shows a live preview to position the camera
- `capture_image()`: Saves an image with the current label.
- `stop()`: Stops the capture process and displays a summary.

Usage Flow:

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



4. Use the live preview to position the camera.
5. Click **Capture Image** to save images under the current label.



6. Change labels as needed for different categories, selecting **Change Label**.
7. Click **Stop Capture** when finished to see a summary.



Technical Notes:

- The script uses threading to handle concurrent frame capture and web serving.
- Images are saved with timestamps in their filenames for uniqueness.
- The web interface is responsive and can be accessed from mobile devices.

Customization Possibilities:

- Adjust image resolution in the `initialize_camera()` function. Here we used QVGA (320×240).
- Modify the HTML templates for a different look and feel.
- Add additional image processing or analysis steps in the `capture_image()` function.

Number of samples on Dataset:

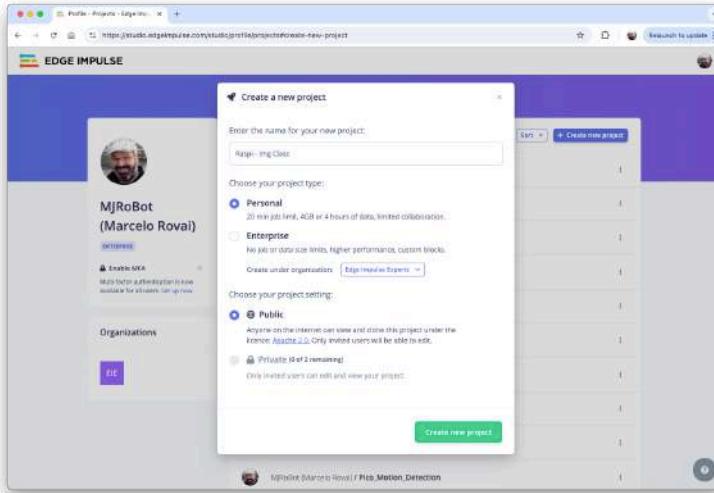
Get around 60 images from each category (`periquito`, `robot` and `background`). Try to capture different angles, backgrounds, and light conditions.

On the Raspi, we will end with a folder named `dataset`, which contains three sub-folders: `periquito`, `robot`, and `background`, one for each class of images.

You can use `Filezilla` to transfer the created dataset to your main computer.

Training the model with Edge Impulse Studio

We will use the Edge Impulse Studio to train our model. Go to the [Edge Impulse Page](#), enter your account credentials, and create a new project:



Here, you can clone a similar project: [Raspi - Img Class](#).

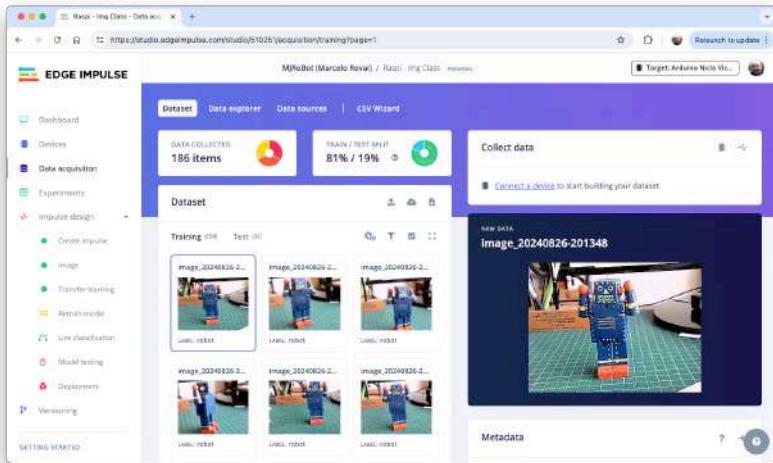
Dataset

We will walk through four main steps using the EI Studio (or Studio). These steps are crucial in preparing our model for use on the Raspi: Dataset, Impulse, Tests, and Deploy (on the Edge Device, in this case, the Raspi).

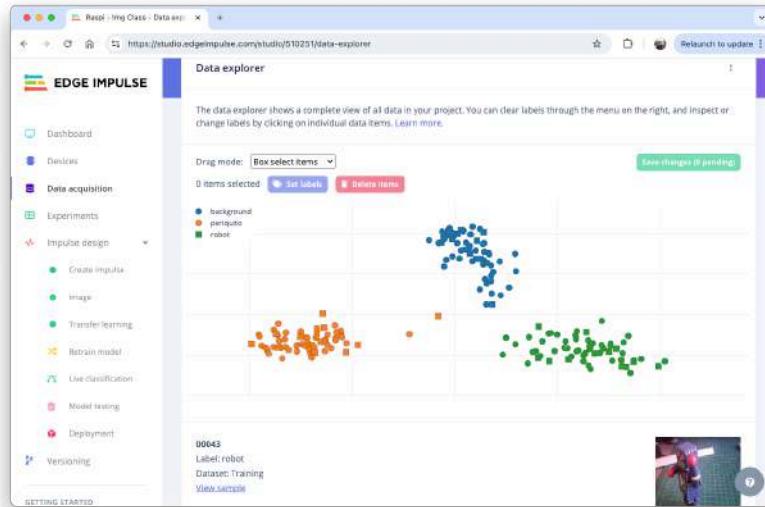
Regarding the Dataset, it is essential to point out that our Original Dataset, captured with the Raspi, will be split into *Training*, *Validation*, and *Test*. The Test Set will be separated from the beginning and reserved for use only in the Test phase after training. The Validation Set will be used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

On Studio, follow the steps to upload the captured data:

1. Go to the **Data acquisition** tab, and in the **UPLOAD DATA** section, upload the files from your computer in the chosen categories.
2. Leave to the Studio the splitting of the original dataset into *train* and *test* and choose the label about
3. Repeat the procedure for all three classes. At the end, you should see your “raw data” in the Studio:



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



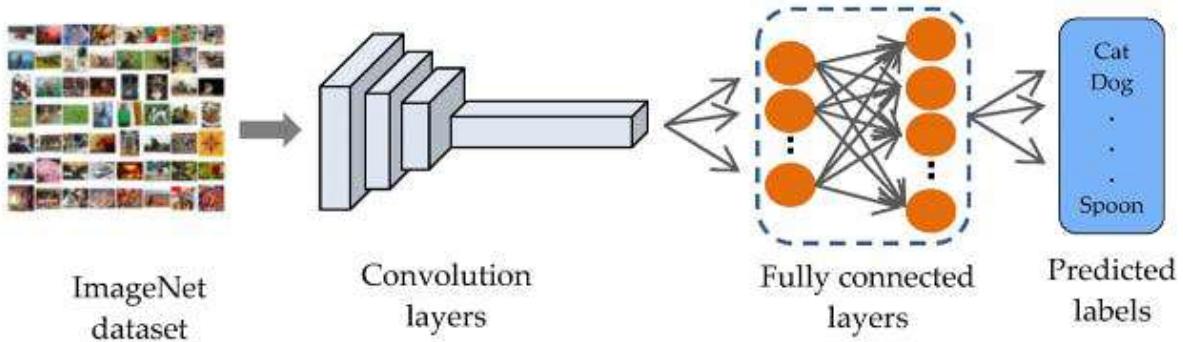
The Impulse Design

In this phase, we should define how to:

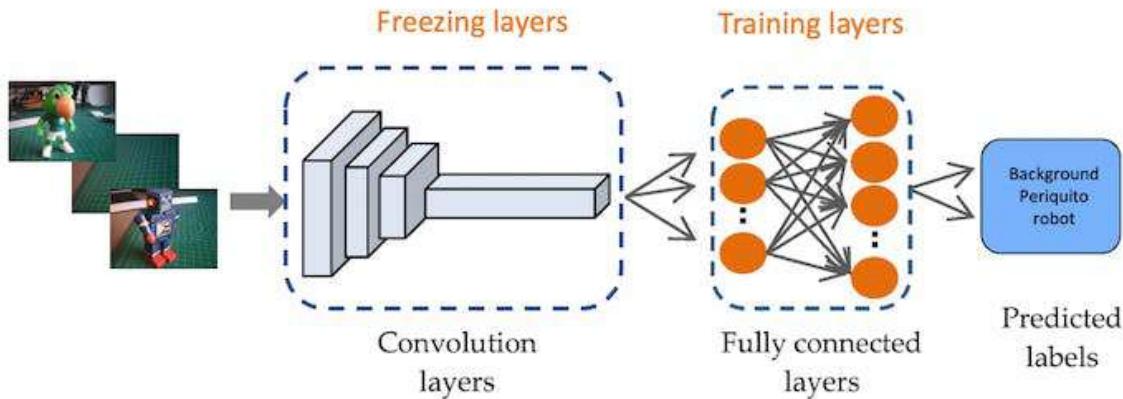
- Pre-process our data, which consists of resizing the individual images and determining the color depth to use (be it RGB or Grayscale) and

- Specify a Model. In this case, it will be the **Transfer Learning (Images)** to fine-tune a pre-trained MobileNet V2 image classification model on our data. This method performs well even with relatively small image datasets (around 180 images in our case).

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



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



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

Go to the Impulse Design Tab and create the *impulse*, defining an image size of 160×160 and squashing them (squared form, without cropping). Select Image and Transfer Learning blocks. Save the Impulse.

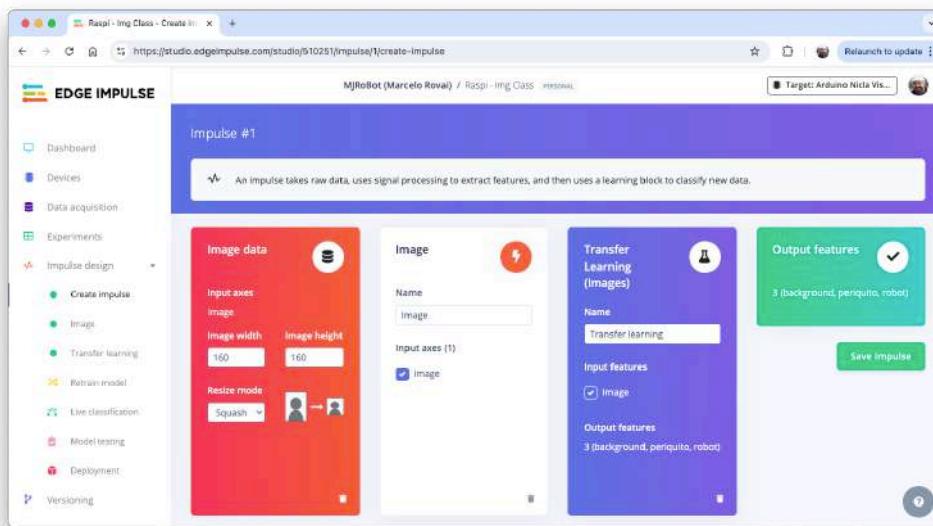


Image Pre-Processing

All the input QVGA/RGB565 images will be converted to 76,800 features ($160 \times 160 \times 3$).

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 160×160 pixels and a *Width Multiplier* of 1.0 (full width, not reduced). This configuration balances between model size, speed, and accuracy.

Model Training

Another valuable deep learning technique is **Data Augmentation**. Data augmentation improves the accuracy of machine learning models by creating additional artificial data. A data augmentation system makes small, random changes to the training data during the training process (such as flipping, cropping, or rotating the images).

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

```
# Implements the data augmentation policy
def augment_image(image, label):
    # Flips the image randomly
    image = tf.image.random_flip_left_right(image)

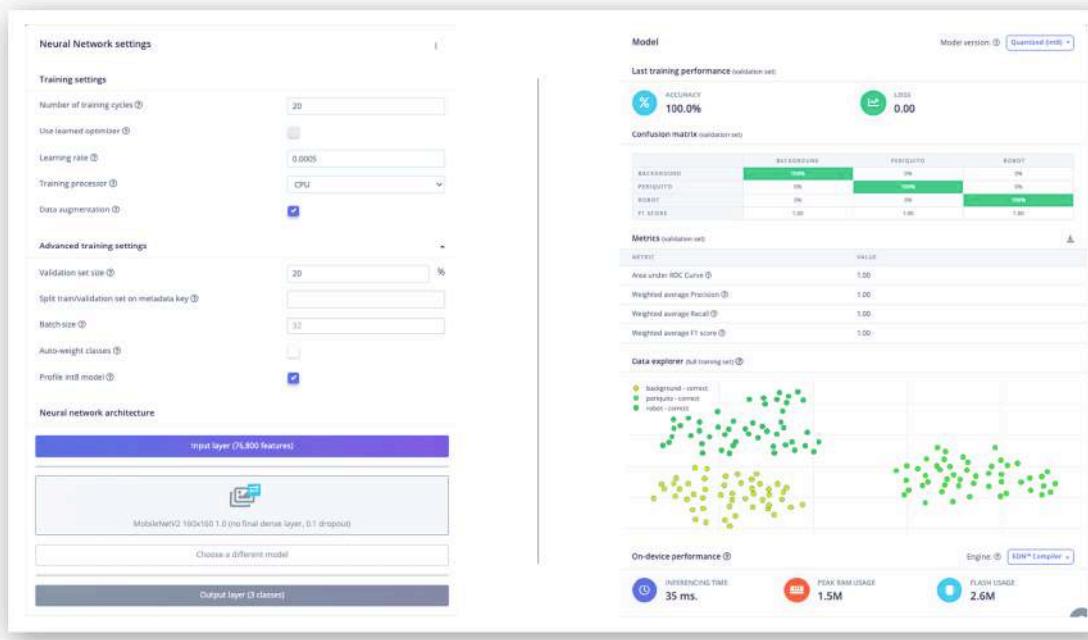
    # Increase the image size, then randomly crop it down to
    # the original dimensions
    resize_factor = random.uniform(1, 1.2)
    new_height = math.floor(resize_factor * INPUT_SHAPE[0])
    new_width = math.floor(resize_factor * INPUT_SHAPE[1])
    image = tf.image.resize_with_crop_or_pad(image,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

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

```
return image, label
```

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

The final dense layer of our model will have 0 neurons with a 10% dropout for overfitting prevention. Here is the Training result:



The result is excellent, with a reasonable 35 ms of latency (for a Raspi-4), which should result in around 30 fps (frames per second) during inference. A Raspi-Zero should be slower, and the Raspi-5, faster.

Trading off: Accuracy versus speed

If faster inference is needed, we should train the model using smaller alphas (0.35, 0.5, and 0.75) or even reduce the image input size, trading with accuracy. However, reducing the input image size and decreasing the alpha (width multiplier) can speed up inference for MobileNet V2, but they have different trade-offs. Let's compare:

1. Reducing Image Input Size:

Pros:

- Significantly reduces the computational cost across all layers.
- Decreases memory usage.
- It often provides a substantial speed boost.

Cons:

- It may reduce the model's ability to detect small features or fine details.
- It can significantly impact accuracy, especially for tasks requiring fine-grained recognition.

2. Reducing Alpha (Width Multiplier):

Pros:

- Reduces the number of parameters and computations in the model.
- Maintains the original input resolution, potentially preserving more detail.
- It can provide a good balance between speed and accuracy.

Cons:

- It may not speed up inference as dramatically as reducing input size.
- It can reduce the model's capacity to learn complex features.

Comparison:

1. Speed Impact:

- Reducing input size often provides a more substantial speed boost because it reduces computations quadratically (halving both width and height reduces computations by about 75%).
- Reducing alpha provides a more linear reduction in computations.

2. Accuracy Impact:

- Reducing input size can severely impact accuracy, especially when detecting small objects or fine details.
- Reducing alpha tends to have a more gradual impact on accuracy.

3. Model Architecture:

- Changing input size doesn't alter the model's architecture.
- Changing alpha modifies the model's structure by reducing the number of channels in each layer.

Recommendation:

1. If our application doesn't require detecting tiny details and can tolerate some loss in accuracy, reducing the input size is often the most effective way to speed up inference.

2. Reducing alpha might be preferable if maintaining the ability to detect fine details is crucial or if you need a more balanced trade-off between speed and accuracy.
3. For best results, you might want to experiment with both:
 - Try MobileNet V2 with input sizes like 160×160 or 92×92
 - Experiment with alpha values like 1.0, 0.75, 0.5 or 0.35.
4. Always benchmark the different configurations on your specific hardware and with your particular dataset to find the optimal balance for your use case.

Remember, the best choice depends on your specific requirements for accuracy, speed, and the nature of the images you're working with. It's often worth experimenting with combinations to find the optimal configuration for your particular use case.

Model Testing

Now, you should take the data set aside at the start of the project and run the trained model using it as input. Again, the result is excellent (92.22%).

Deploying the model

As we did in the previous section, we can deploy the trained model as .tflite and use Raspi to run it using Python.

On the Dashboard tab, go to Transfer learning model (int8 quantized) and click on the download icon:

Download block output			
TITLE	TYPE	SIZE	
Image training data	NPY file	150 windows	
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 comparison

Transfer the models from your computer to the Raspi (`./models`), for example, using FileZilla. Also, capture some images for inference and save them in (`./images`), or use the images in the `./dataset` folder.

Let's remember what we did in the last chapter:

Activate the environment:

```
source ~/tflite_env/bin/activate
```

Run a Jupyter Notebook, using the command:

```
jupyter notebook --ip=192.168.4.210 --no-browser
```

Change the IP address for yours

Open a new notebook and enter with the code below:

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
```

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 `255`. This means that we should pre-process the image to match it. We can check here:

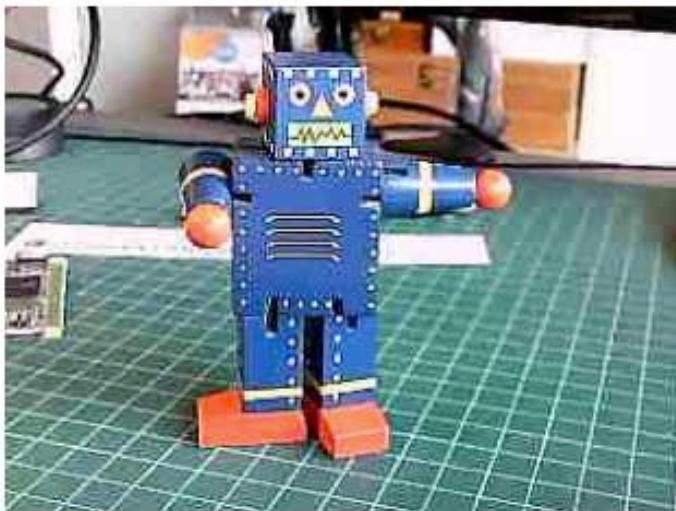
```
input_dtype = input_details[0]['dtype']
input_dtype

numpy.int8
```

So, let's open the image and show it:

```
img = Image.open(img_path)
plt.figure(figsize=(4, 4))
plt.imshow(img)
```

```
plt.axis('off')
plt.show()
```



And perform the pre-processing:

```
scale, zero_point = input_details[0]['quantization']
img = img.resize((input_details[0]['shape'][1],
                  input_details[0]['shape'][2]))
img_array = np.array(img, dtype=np.float32) / 255.0
img_array = (
    (img_array / scale + zero_point)
    .clip(-128, 127)
    .astype(np.int8)
)
input_data = np.expand_dims(img_array, axis=0)
```

Checking the input data, we can verify that the input tensor is compatible with what is expected by the model:

```
input_data.shape, input_data.dtype
```

```
((1, 160, 160, 3), dtype('int8'))
```

Now, it is time to perform the inference. Let's also calculate the latency of the model:

```

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

```

The model will take around 125ms to perform the inference in the Raspi-Zero, which is 3 to 4 times longer than a Raspi-5.

Now, we can get the output labels and probabilities. It is also important to note that the model trained on the Edge Impulse Studio has a softmax activation function in its output (different from the original Movilenet V2), and we can 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(

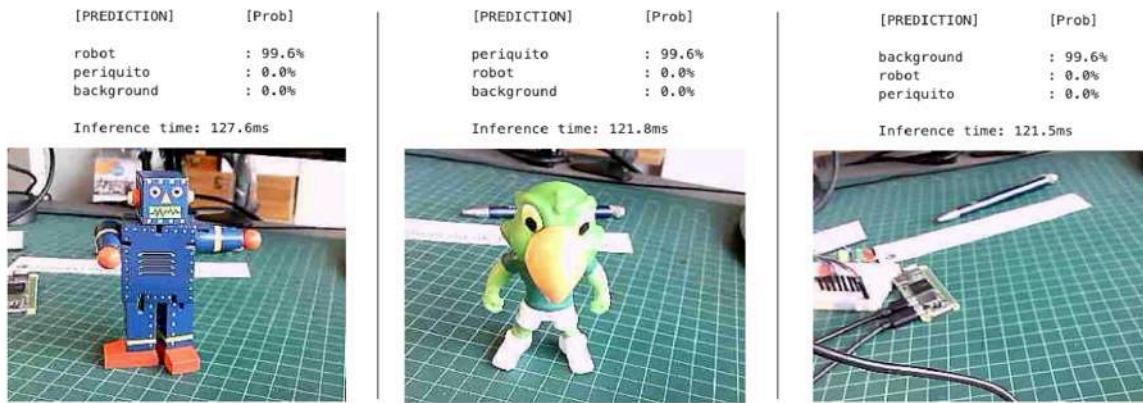
```

```

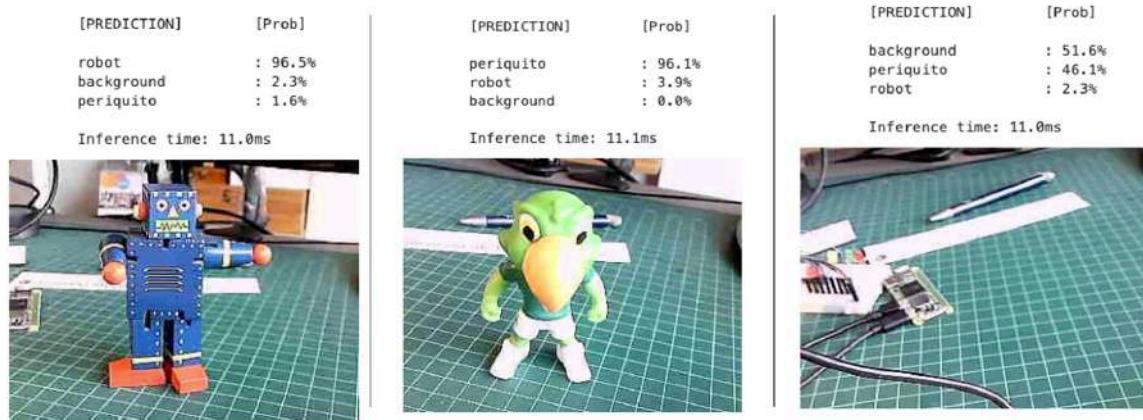
    labels[top_k_indices[i]],
    probabilities[top_k_indices[i]] * 100))
print ("\n\tInference time: {:.1f}ms".format(inference_time))

```

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 that captures images with the camera in real-time and displays their 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>

<button id="startBtn" onclick="startClassification()">

Start Classification

</button>

<button id="stopBtn" onclick="stopClassification()" disabled>

```

        Stop Classification
    </button>

    <br>
    <label for="confidence">Confidence Threshold:</label>
    <input type="number"
        id="confidence"
        name="confidence"
        min="0" max="1"
        step="0.1"
        value="0.8"
        onchange="updateConfidence()" />

    <br>
    <div id="classification">
        Waiting for classification...
    </div>

</body>
</html>
''')

@app.route('/video_feed')
def video_feed():
    return Response(
        generate_frames(),
        mimetype='multipart/x-mixed-replace; boundary=frame'
    )

@app.route('/start', methods=['POST'])
def start_classification():
    global is_classifying
    is_classifying = True
    return '', 204

@app.route('/stop', methods=['POST'])
def stop_classification():
    global is_classifying
    is_classifying = False
    return '', 204

```

```

@app.route('/update_confidence', methods=['POST'])
def update_confidence():
    global confidence_threshold
    confidence_threshold = float(request.form['confidence'])
    return '', 204

@app.route('/get_classification')
def get_classification():
    if not is_classifying:
        return jsonify({'label': 'Not classifying',
                       'probability': 0})
    try:
        result = classification_queue.get_nowait()
    except Queue.Empty:
        result = {'label': 'Processing', 'probability': 0}
    return jsonify(result)

if __name__ == '__main__':
    initialize_camera()
    threading.Thread(target=get_frame, daemon=True).start()
    threading.Thread(target=classification_worker,
                     daemon=True).start()
    app.run(host='0.0.0.0', port=5000, threaded=True)

```

On the terminal, run:

```
python3 img_class_live_infer.py
```

And access the web interface:

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

Here are some screenshots of the app running on an external desktop



Here, you can see the app running on the YouTube:

<https://www.youtube.com/watch?v=o1QsQrpCMw4>

The code creates a web application for real-time image classification using a Raspberry Pi, its camera module, and a TensorFlow Lite model. The application uses Flask to serve a web interface where it is possible to view the camera feed and see live classification results.

Key Components:

1. **Flask Web Application:** Serves the user interface and handles requests.
2. **PiCamera2:** Captures images from the Raspberry Pi camera module.
3. **TensorFlow Lite:** Runs the image classification model.
4. **Threading:** Manages concurrent operations for smooth performance.

Main Features:

- Live camera feed display
- Real-time image classification
- Adjustable confidence threshold
- Start/Stop classification on demand

Code Structure:

1. Imports and Setup:

- Flask for web application
- PiCamera2 for camera control
- TensorFlow Lite for inference
- Threading and Queue for concurrent operations

2. Global Variables:

- Camera and frame management
- Classification control
- Model and label information

3. Camera Functions:

- `initialize_camera()`: Sets up the PiCamera2
- `get_frame()`: Continuously captures frames
- `generate_frames()`: Yields frames for the web feed

4. Model Functions:

- `load_model()`: Loads the TFLite model
- `classify_image()`: Performs inference on a single image

5. Classification Worker:

- Runs in a separate thread
- Continuously classifies frames when active
- Updates a queue with the latest results

6. Flask Routes:

- `/`: Serves the main HTML page
- `/video_feed`: Streams the camera feed
- `/start` and `/stop`: Controls classification
- `/update_confidence`: Adjusts the confidence threshold
- `/get_classification`: Returns the latest classification result

7. HTML Template:

- Displays camera feed and classification results
- Provides controls for starting/stopping and adjusting settings

8. Main Execution:

- Initializes camera and starts necessary threads
- Runs the Flask application

Key Concepts:

1. **Concurrent Operations**: Using threads to handle camera capture and classification separately from the web server.
2. **Real-time Updates**: Frequent updates to the classification results without page reloads.
3. **Model Reuse**: Loading the TFLite model once and reusing it for efficiency.
4. **Flexible Configuration**: Allowing users to adjust the confidence threshold on the fly.

Usage:

1. Ensure all dependencies are installed.
2. Run the script on a Raspberry Pi with a camera module.
3. Access the web interface from a browser using the Raspberry Pi's IP address.
4. Start classification and adjust settings as needed.

Summary:

Image classification has emerged as a powerful and versatile application of machine learning, with significant implications for various fields, from healthcare to environmental monitoring. This chapter has demonstrated how to implement a robust image classification system on edge devices like the Raspi-Zero and Raspi-5, showcasing the potential for real-time, on-device intelligence.

We've explored the entire pipeline of an image classification project, from data collection and model training using Edge Impulse Studio to deploying and running inferences on a Raspi. The process highlighted several key points:

1. The importance of proper data collection and preprocessing for training effective models.
2. The power of transfer learning, allowing us to leverage pre-trained models like MobileNet V2 for efficient training with limited data.
3. The trade-offs between model accuracy and inference speed, especially crucial for edge devices.
4. The implementation of real-time classification using a web-based interface, demonstrating practical applications.

The ability to run these models on edge devices like the Raspi opens up numerous possibilities for IoT applications, autonomous systems, and real-time monitoring solutions. It allows for reduced latency, improved privacy, and operation in environments with limited connectivity.

As we've seen, even with the computational constraints of edge devices, it's possible to achieve impressive results in terms of both accuracy and speed. The flexibility to adjust model parameters, such as input size and alpha values, allows for fine-tuning to meet specific project requirements.

Looking forward, the field of edge AI and image classification continues to evolve rapidly. Advances in model compression techniques, hardware acceleration, and more efficient neural network architectures promise to further expand the capabilities of edge devices in computer vision tasks.

This project serves as a foundation for more complex computer vision applications and encourages further exploration into the exciting world of edge AI and IoT. Whether it's for industrial

automation, smart home applications, or environmental monitoring, the skills and concepts covered here provide a solid starting point for a wide range of innovative projects.

Resources

- [Dataset Example](#)
- [Python Scripts](#)
- [Edge Impulse Project](#)
- [Image Classification Project - Edge Impulse Notebook](#)

Object Detection: Fundamentals

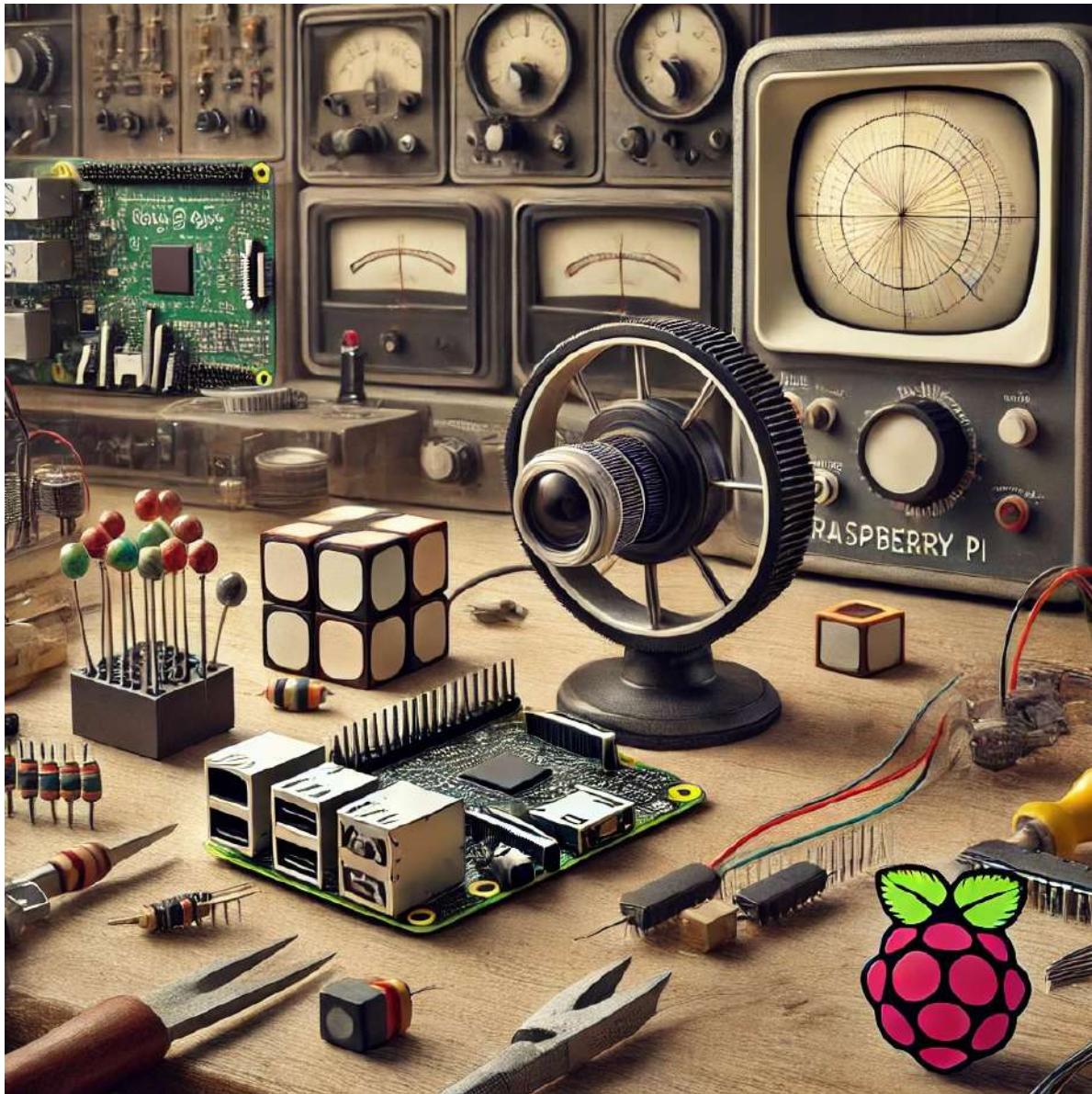


Figure 4: *DALL·E prompt - A cover image for an ‘Object Detection’ chapter in a Raspberry Pi tutorial, designed in the same vintage 1950s electronics lab style as previous covers. The scene should prominently feature wheels and cubes, similar to those provided by the user, placed on a workbench in the 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 even more pronounced in object detection. However, the rewards are also more significant, as object detection enables more nuanced and detailed analysis of visual data.

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

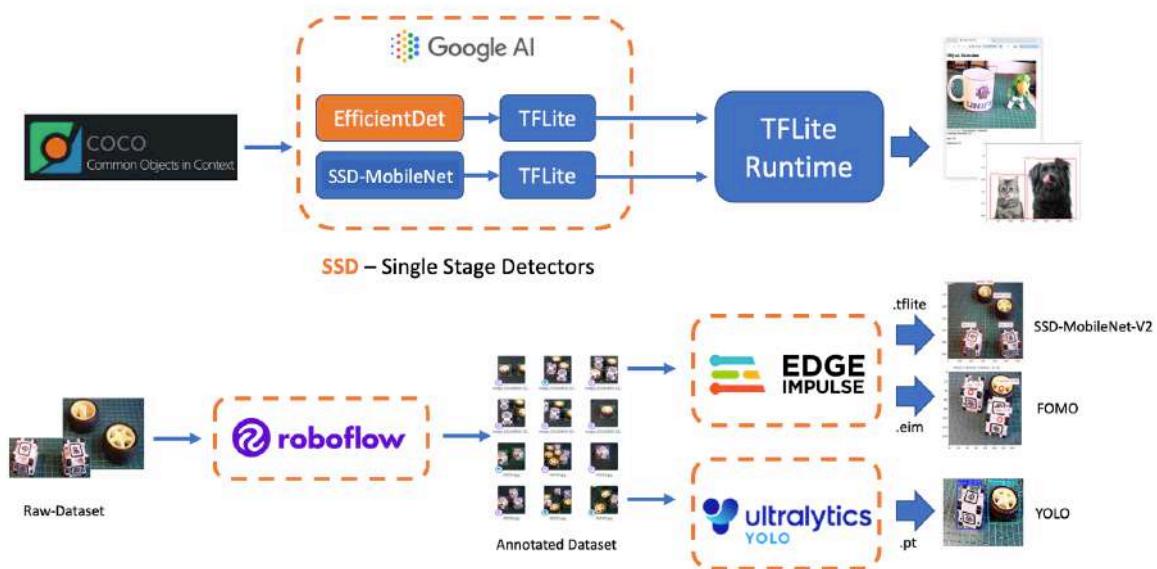
As we put our hands into object detection, we'll build upon the concepts and techniques we explored in image classification. We'll examine popular object detection architectures designed for efficiency, such as:

- 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

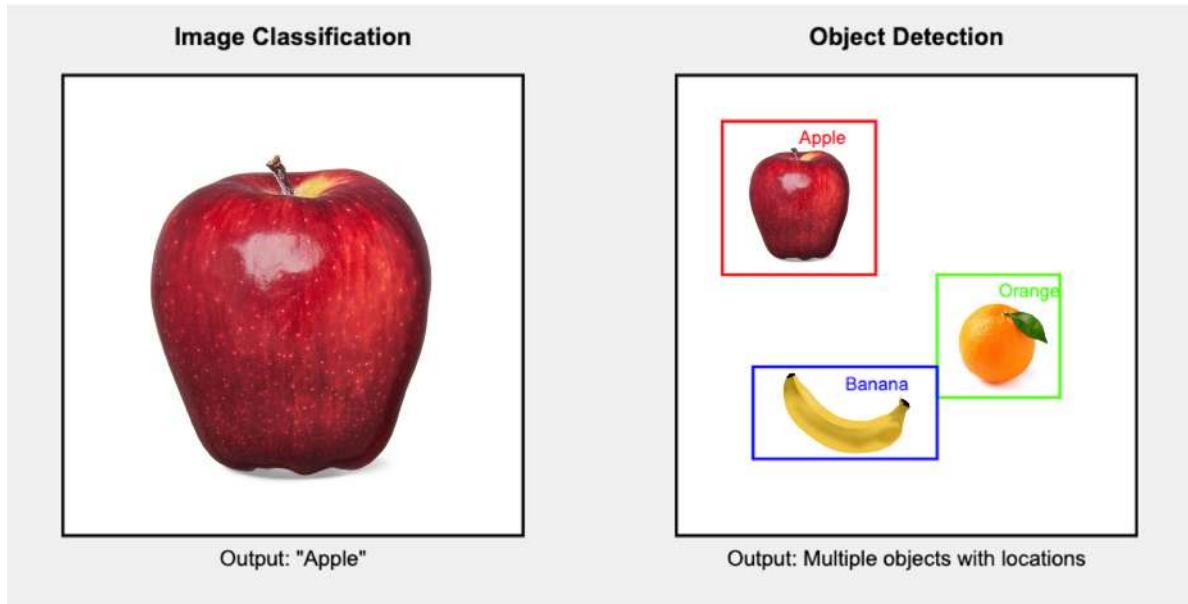
Image Classification:

- 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

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 the location of objects within 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

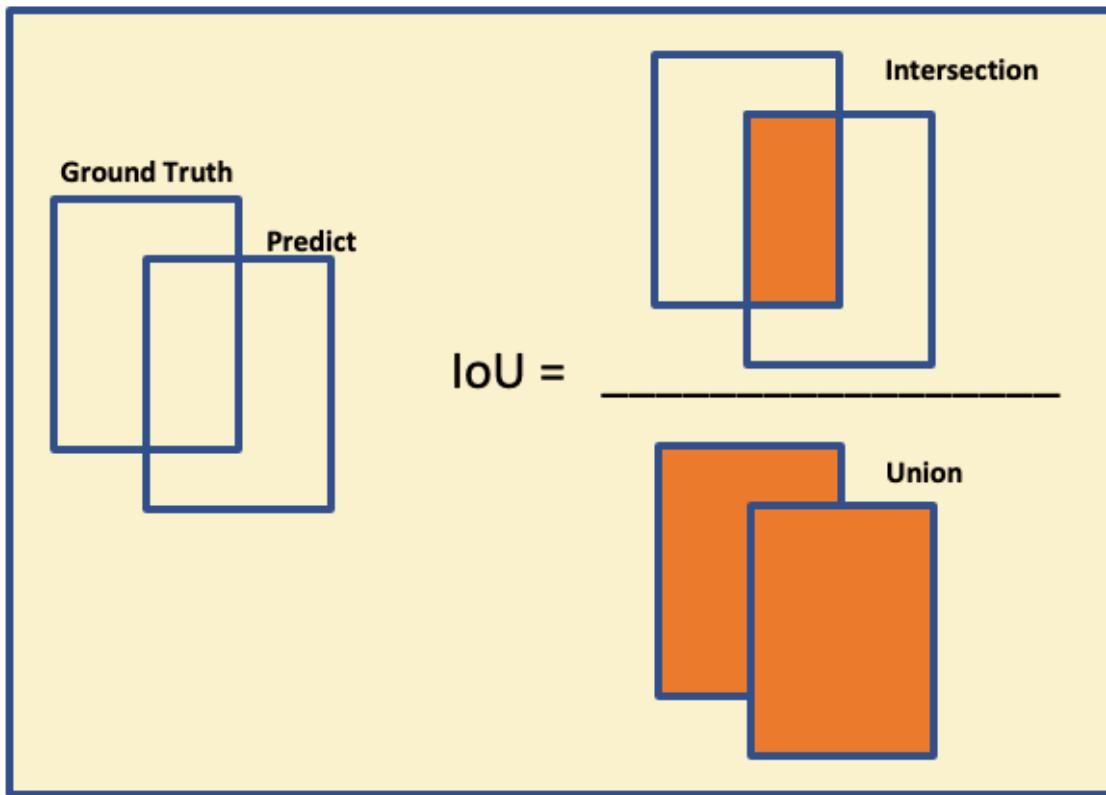
There are two main approaches to object detection:

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, such as the Raspberry Pi.

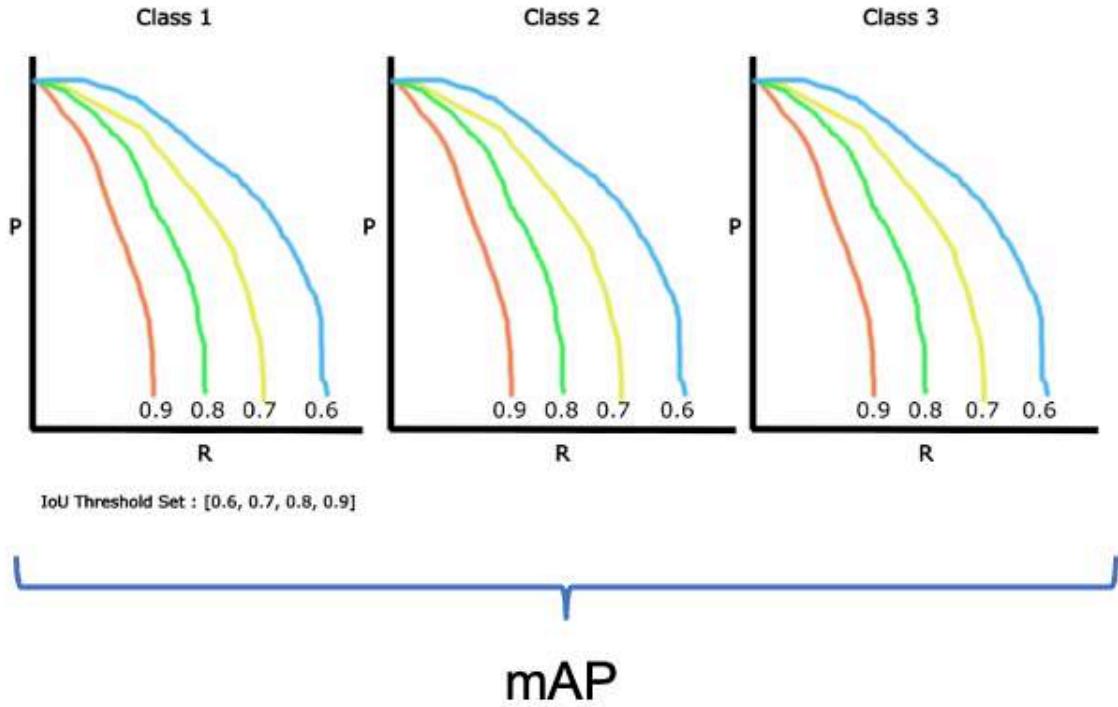
Evaluation Metrics

Object detection uses different metrics compared to image classification:

- **Intersection over Union (IoU)** is a metric used to evaluate the **accuracy** of an object detector. It measures the overlap between two bounding boxes: the **Ground Truth** box (the manually labeled correct box) and the **Predicted** box (the box generated by the object detection model). The IoU value is calculated by dividing the area of the **Intersection** (the overlapping area) by the area of the **Union** (the total area covered by both boxes). A higher IoU value indicates a better prediction.



- **Mean Average Precision (mAP)** is a widely used metric for evaluating the **performance** of object detection models. It provides a single number that reflects a model's ability to accurately both **classify** and **localize** objects. The “mean” in mAP refers to the average taken over all object classes in the dataset. The “average precision” (AP) is calculated for each class, and then these AP values are averaged to get the final mAP score. A high mAP score indicates that the model is excellent at identifying all objects and placing a tight-fitting, accurate bounding box around them.



- **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](#)

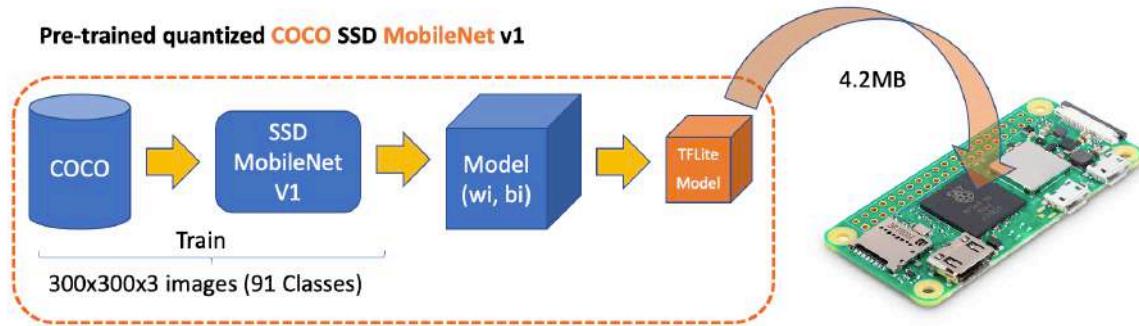
On [Kaggle](#), we can find the most common pre-trained TFLite models to use with the Raspberry Pi, [ssd_mobilenet_v1](#), and [efficiendet](#). Those models were trained on the COCO (Common Objects in Context) dataset, with over 200,000 labeled images in 91 categories.

Download the models and upload them to the `./models` folder on the Raspberry Pi.

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.



The model outputs up to ten detections per image, including bounding boxes, class IDs, and confidence scores.

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
```

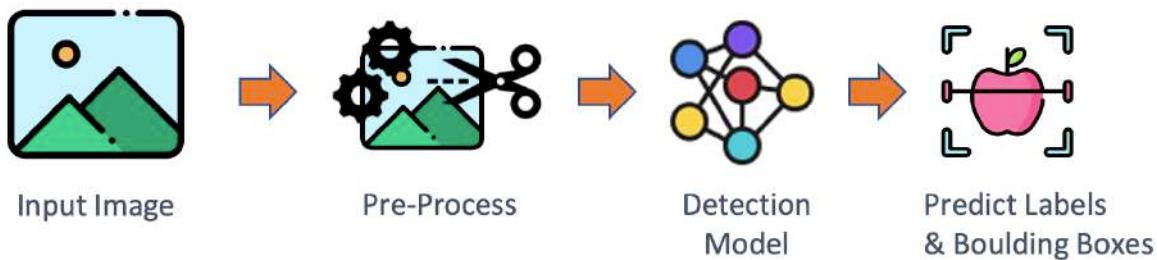
- Installing TensorFlow Lite Runtime
- Installing Additional Python Libraries (inside the environment)

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 in 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).

```
input_details
```

```
[{'name': 'normalized_input_image_tensor',
 'index': 175,
 'shape': array([ 1, 300, 300, 3], dtype=int32),
 'shape_signature': array([ 1, 300, 300, 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': {}}]
```

TFLite Interpreter

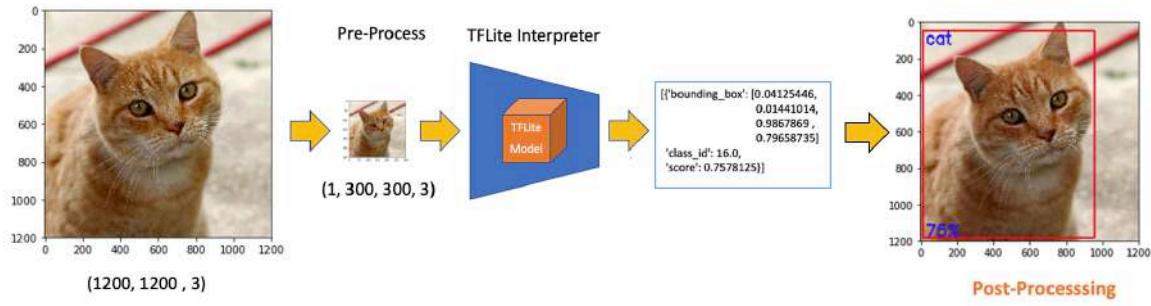
The **output details** include not only the labels (“classes”) and probabilities (“scores”) but also the relative window position of the bounding boxes (“boxes”) about where the object is located on the image and the number of detected objects (“num_detections”). The output details also tell us that the model can detect a **maximum of 10 objects** in the image.

TFLite Interpreter

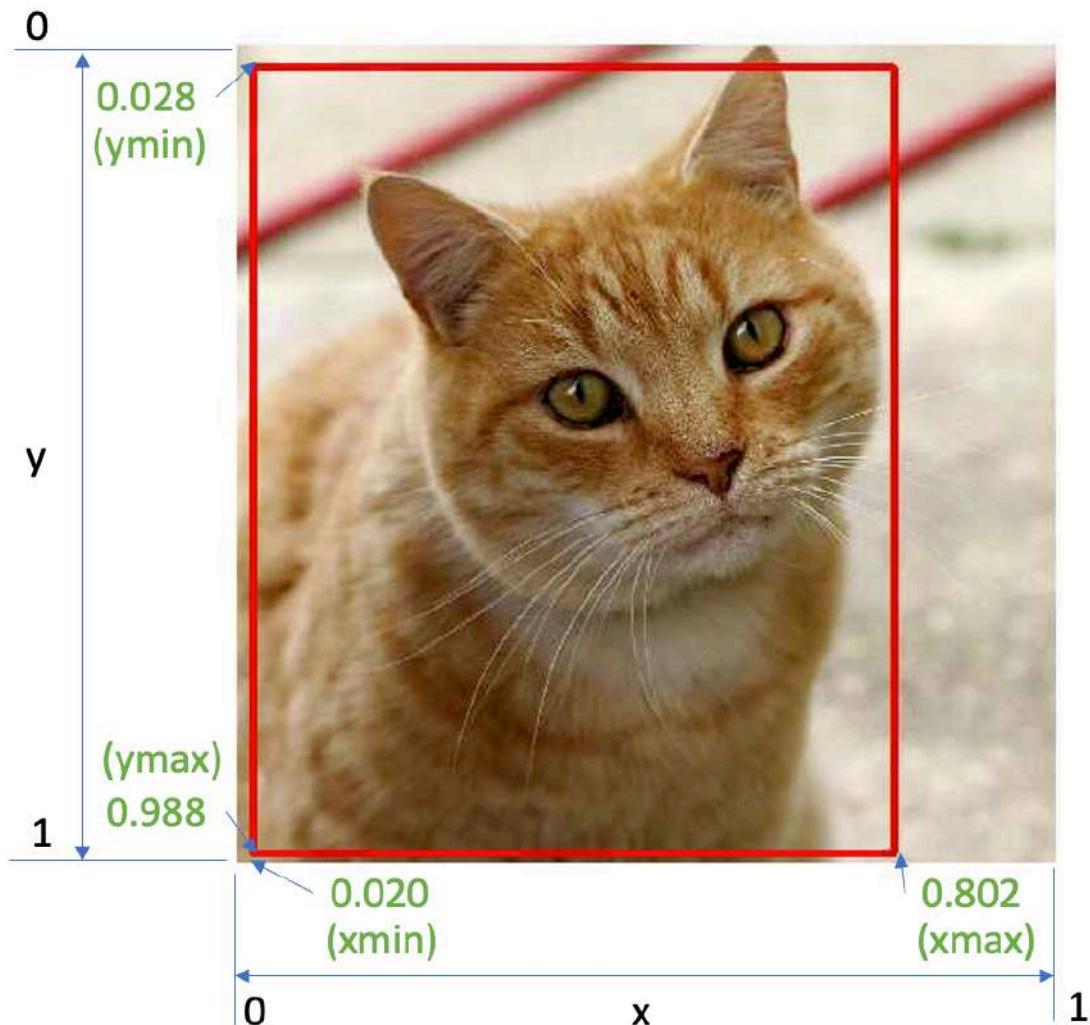
```
output_details
```

```
[{'name': 'TFLite_Detection_PostProcess',
 'index': 167,
 'shape': array([ 1, 10, 4], dtype=int32),
 'shape_signature': array([ 1, 10, 4], dtype=int32),
 'dtype': numpy.float32,
 'quantization': (0.0, 0),
 'quantization_parameters': {'scales': array([], dtype=float32),
 'zero_points': array([], dtype=int32),
 'quantized_dimension': 0},
 'sparsity_parameters': {}}, 1. Boxes
 {'name': 'TFLite_Detection_PostProcess:1',
 'index': 168,
 'shape': array([ 1, 10], dtype=int32),
 'shape_signature': array([ 1, 10], dtype=int32),
 'dtype': numpy.float32,
 'quantization': (0.0, 0),
 'quantization_parameters': {'scales': array([], dtype=float32),
 'zero_points': array([], dtype=int32),
 'quantized_dimension': 0},
 'sparsity_parameters': {}}, 2. Scores
 {'name': 'TFLite_Detection_PostProcess:2',
 'index': 169,
 'shape': array([ 1, 10], dtype=int32),
 'shape_signature': array([ 1, 10], dtype=int32),
 'dtype': numpy.float32,
 'quantization': (0.0, 0),
 'quantization_parameters': {'scales': array([], dtype=float32),
 'zero_points': array([], dtype=int32),
 'quantized_dimension': 0},
 'sparsity_parameters': {}}, 3. Classes
 {'name': 'TFLite_Detection_PostProcess:3',
 'index': 170,
 'shape': array([1], dtype=int32),
 'shape_signature': array([1], dtype=int32),
 'dtype': numpy.float32,
 'quantization': (0.0, 0),
 'quantization_parameters': {'scales': array([], dtype=float32),
 'zero_points': array([], dtype=int32),
 'quantized_dimension': 0},
 'sparsity_parameters': {}}, 4. Count
```

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.



Considering that y ranges from the top (y_{min}) to the bottom (y_{max}) and x ranges from left (x_{min}) to right (x_{max}), 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:



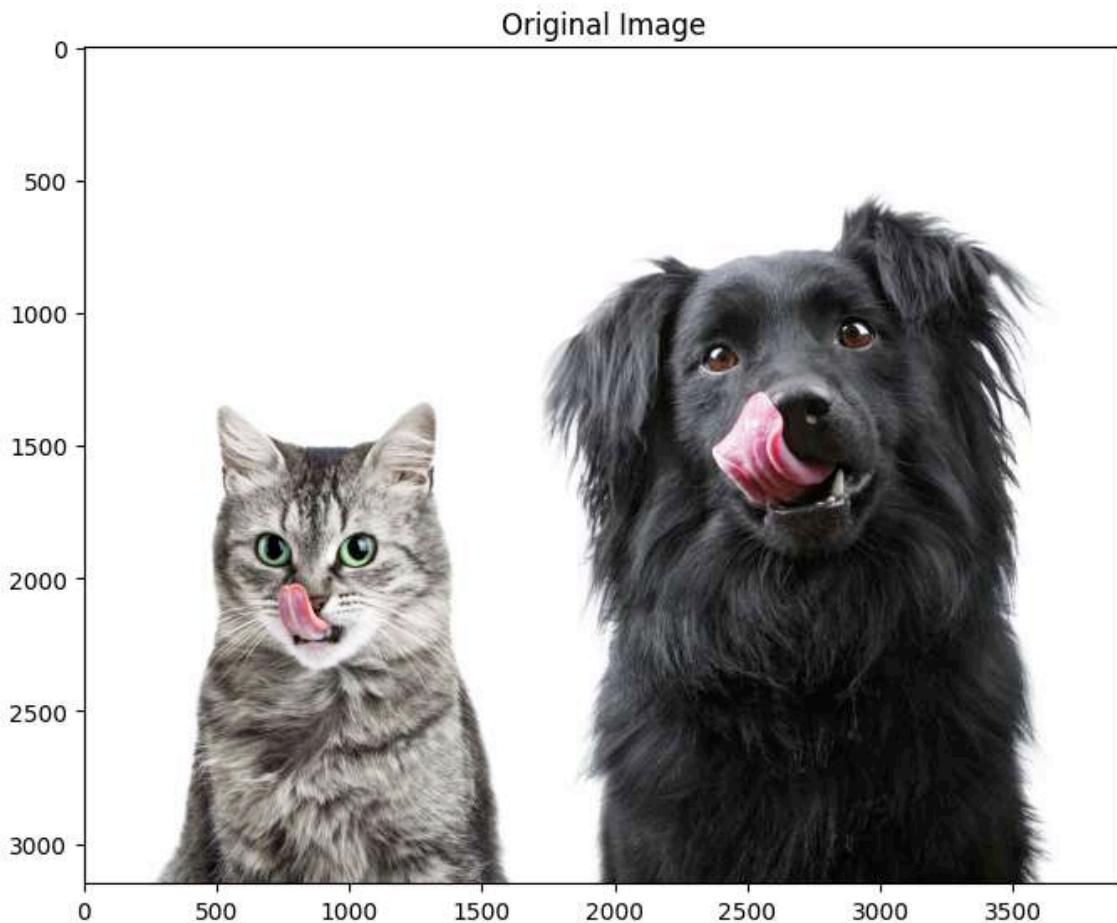
Next, we should find what class ID 16 means. Opening the file `coco_labels.txt`, as a list, each element has an associated index, and inspecting index 16, we get, as expected, `cat`. The probability is the value returned from the score.

Let's now upload some images with multiple objects on them 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 dimensions:

```
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 four 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 two 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
```

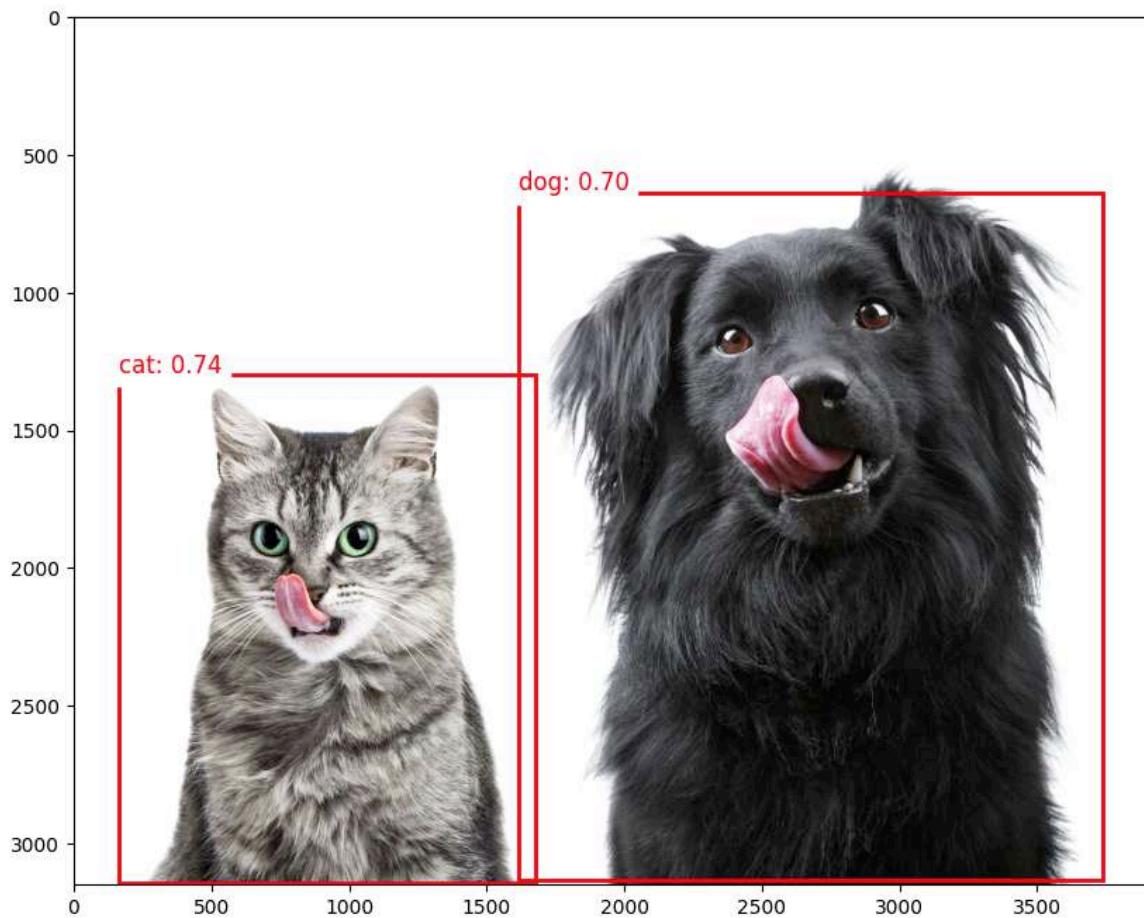
And we can also visualize the results:

```
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')

```



The choice of the `confidence threshold` is crucial. For example, changing it to 0.2 will show false positives. A proper code should handle it.

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 enables a family of models with varying 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 on a live stream

The object detection models can also detect objects in real-time using a camera. The captured image should be the input for the trained and converted model. For the Raspberry Pi 4 or 5 with a desktop, OpenCV can capture the frames and display the inference result.

However, even without a desktop, 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*.

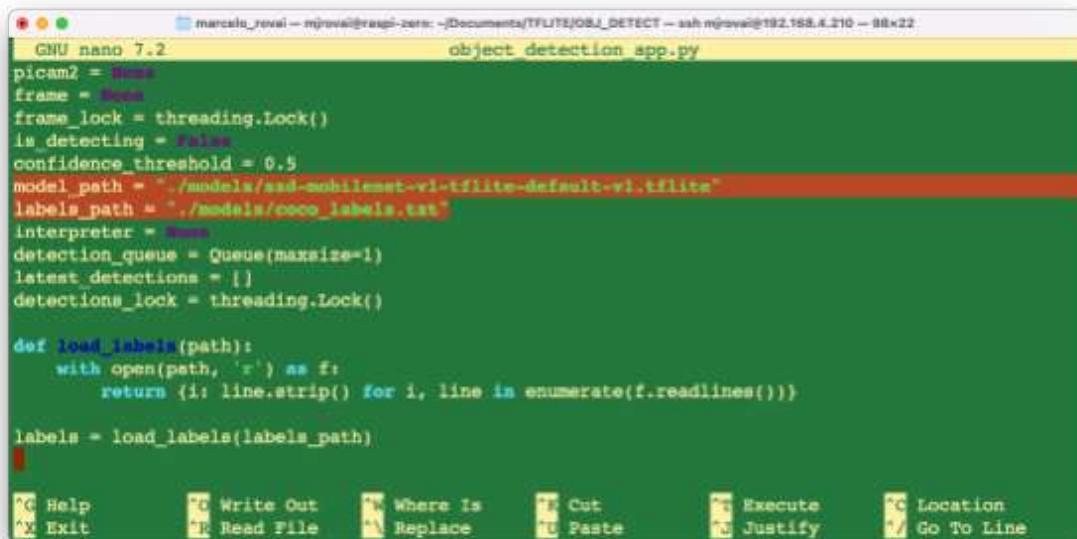
This app version should work for any TFLite models.

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](#).

Check the model and labels:



```
marcelo_reval - mjreval@raspi-zero: ~/Documents/TFLITE/OBJ_DETECT - ash mjreval@192.168.4.210 - 98x22
GNU nano 7.2                                     object_detection_app.py
picam2 = None
frame = None
frame_lock = threading.Lock()
is_detecting = False
confidence_threshold = 0.5
model_path = "./models/ssd-mobilenet-v1-tflite-default-v1.tflite"
labels_path = "./models/coco_labels.txt"
interpreter = None
detection_queue = Queue(maxsize=1)
latest_detections = {}
detections_lock = threading.Lock()

def load_labels(path):
    with open(path, 'r') as f:
        return {i: line.strip() for i, line in enumerate(f.readlines())}

labels = load_labels(labels_path)
```

And on the terminal, run:

```
python object_detection_app.py
```

After starting, you should receive the message on the terminal (the IP is from my Raspberry):

* Running on `http://192.168.4.210:5000`

Press CTRL+C to quit

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 is a screenshot 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 building backend servers.
- 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 enables simultaneous frame capture, object detection, and web server operation, which is 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 enables efficient, real-time object detection while maintaining a responsive web interface on a resource-constrained edge device, such as a Raspberry Pi. Threading and efficient libraries, such as 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, by 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 in 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 examined the object detection models SSD-MobileNet and EfficientDet, comparing their performance and trade-offs on edge devices.

The lab exemplified a real-time object detection web application, demonstrating how these models can be integrated into practical, interactive systems.

The ability to perform object detection on edge devices opens up numerous possibilities across various domains, including precision agriculture, industrial automation, quality control, 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:

- Using a custom dataset (labeled on Roboflow), walking through the process of training models using Edge Impulse Studio and Ultralytics, and deploying them on Raspberry Pi.
- To improve inference speed on edge devices, explore various optimization methods, such as model quantization (TFLite int8) and format conversion (e.g., to NCNN).
- 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

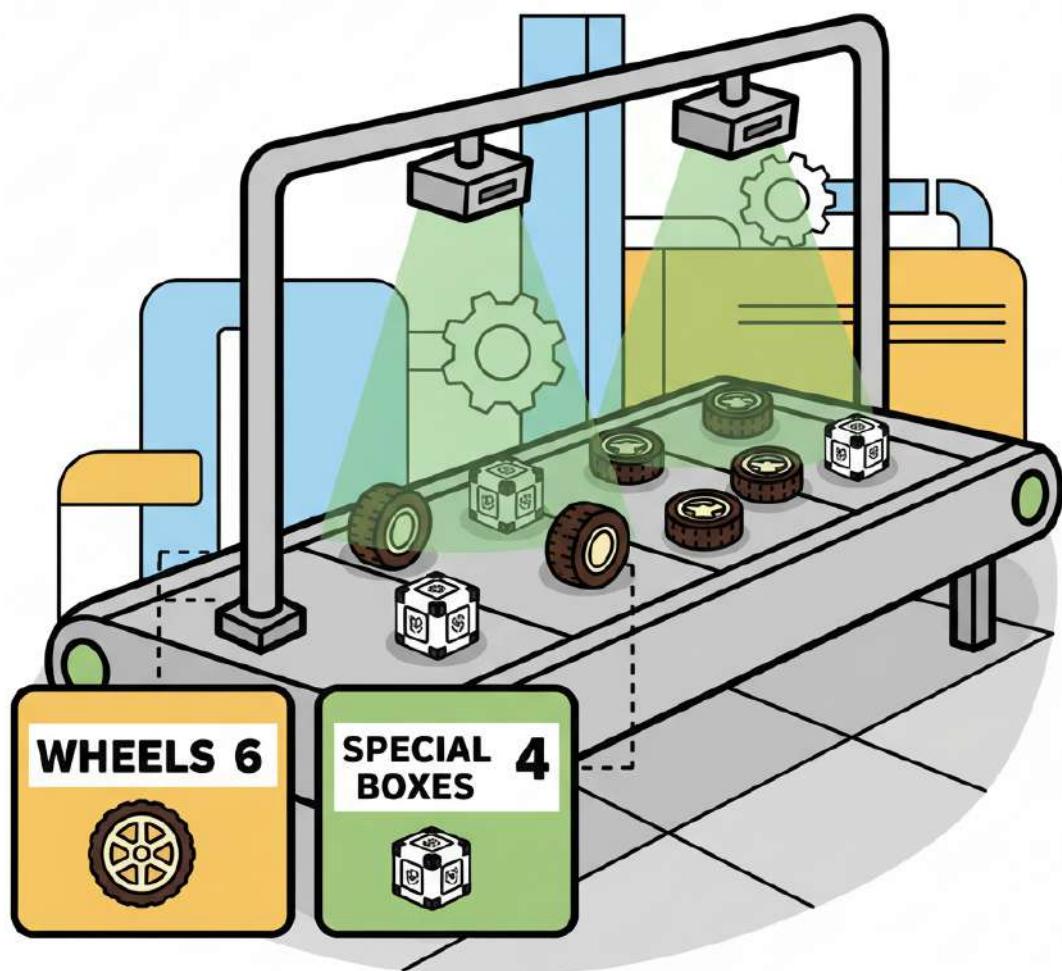
Object detection on edge devices can create intelligent, responsive systems that bring the power of AI directly into the physical world, opening up new frontiers in how we interact with and understand our environment.

Resources

- [SSD-MobileNet Notebook on a Raspi](#)

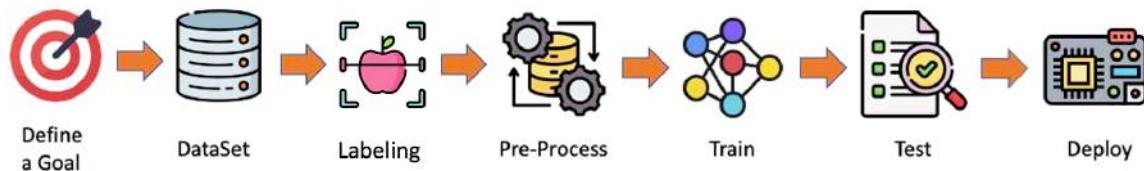
- EfficientDet Notebook on a Raspi
- Python Scripts
- Models

Custom Object Detection Project



Object Detection Project

In this chapter, we will develop a complete Object Detection project from data collection, labelling, 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

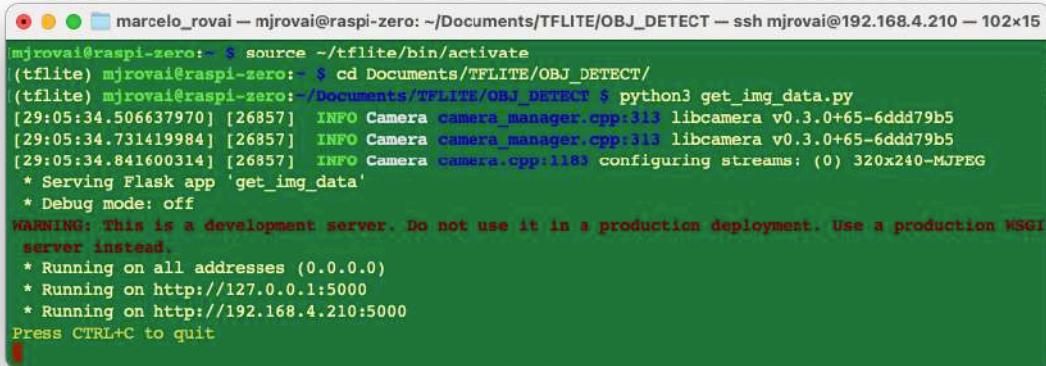
Once we have defined our Machine Learning project goal, the next and most crucial step is collecting the dataset. We can use a phone, the Raspi, or a mix to create the raw dataset (with no labels). Let's use the simple web app on our Raspberry Pi to view the QVGA (320 x 240) captured images in a browser.

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:

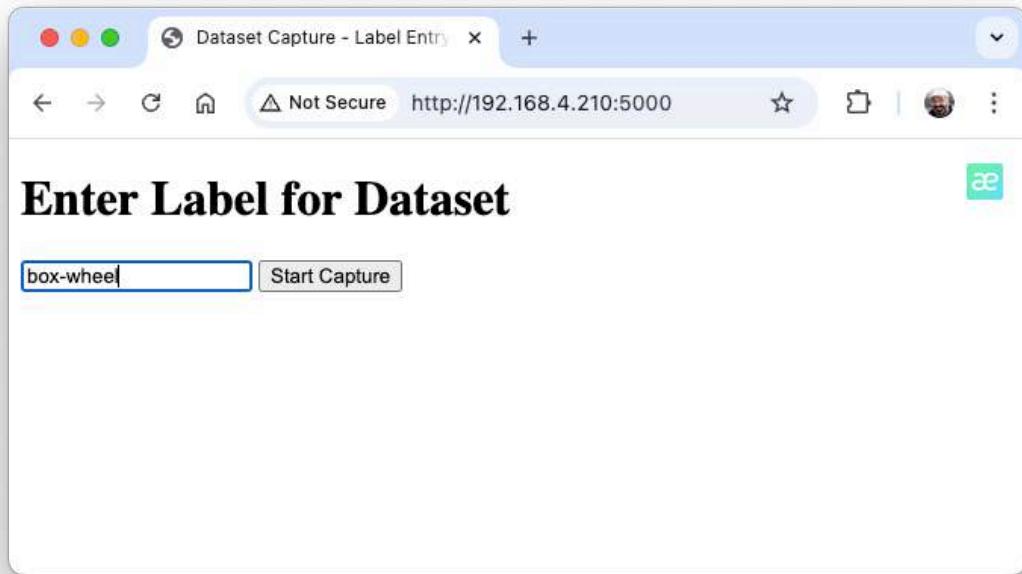
- On the Raspberry Pi itself (if you have a GUI): Open a web browser and go to `http://localhost:5000`
- From another device on the same network: Open a web browser and go to `http://<raspberry_pi_ip>:5000` (Replace `<raspberry_pi_ip>` with your Raspberry Pi's IP address). For example: `http://192.168.4.210:5000/`



```
mjrovai@raspi-zero:~$ source ~/tfelite/bin/activate
(tfelite) mjrovai@raspi-zero:~$ cd Documents/TFLITE/OBJ_DETECT/
(tfelite) 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 in the folder `dataset/box-wheel`:

```

marcelo_rovai@raspi-zero: ~/Documents/TFLITE/OBJ_DETECT/dataset - ssh mjrovai@192.168.4.210 - 102x11
(tflite) mjrovai@raspi-zero:~/Documents/TFLITE/OBJ_DETECT $ 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 $ 

```

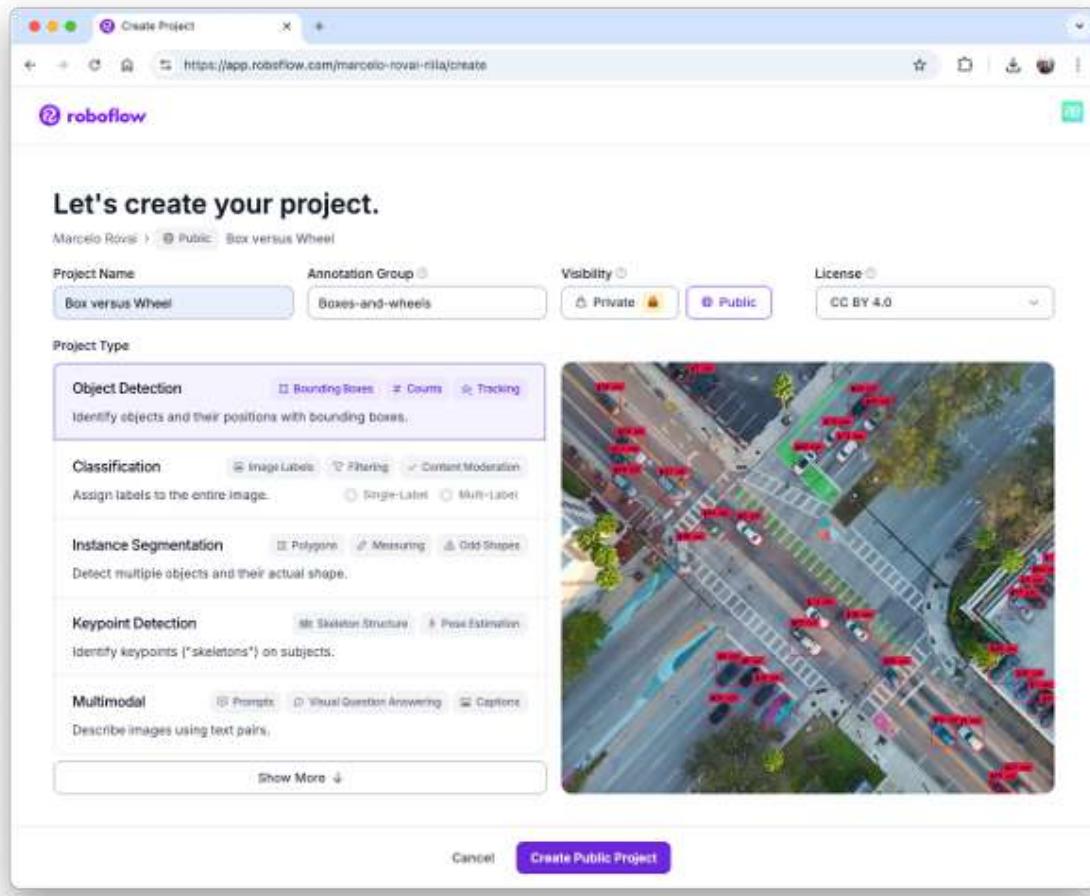
Get around 60 images. Try to capture different angles, backgrounds, and light conditions. Filezilla can transfer the created raw dataset to your main computer.

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.

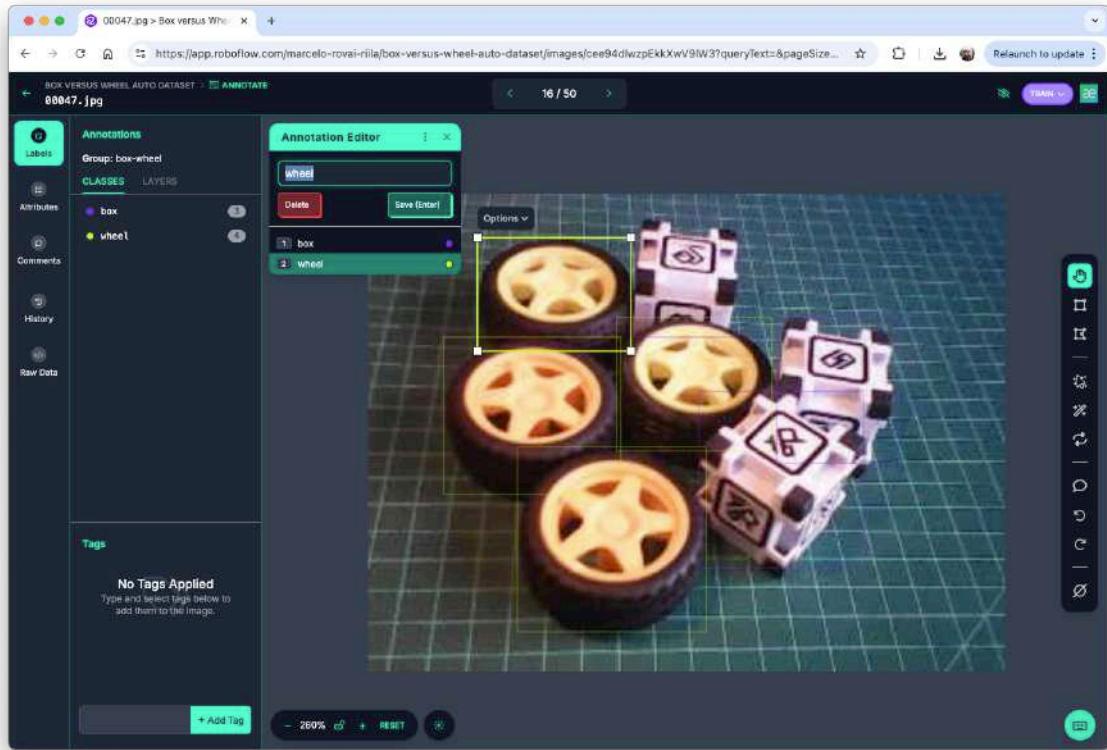
We should upload the raw dataset to [Roboflow](#). Create a free account there and start a new project, for example, ("box-versus-wheel").



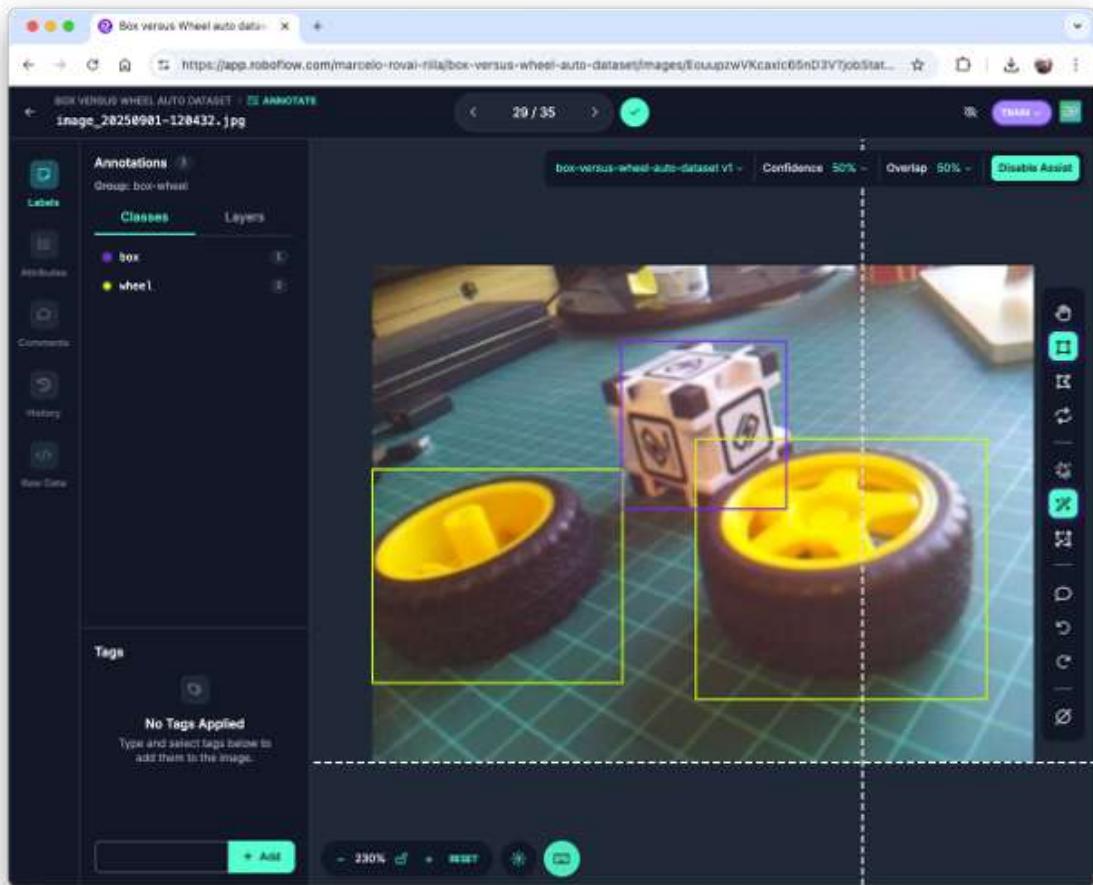
We will not enter in deep details about the Roboflow process once many tutorials are available.

Annotate

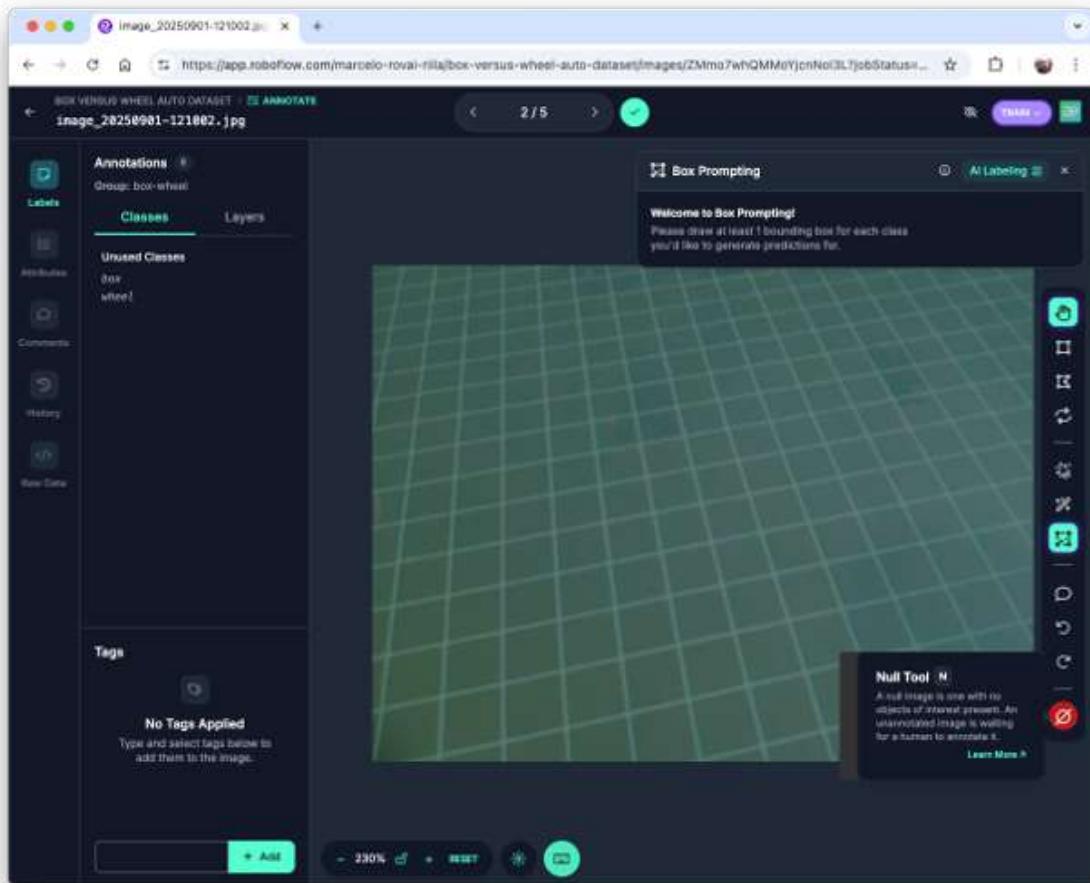
Once the project is created and the dataset is uploaded, you can use the “Auto-Label” tool to make the annotations or do it manually.



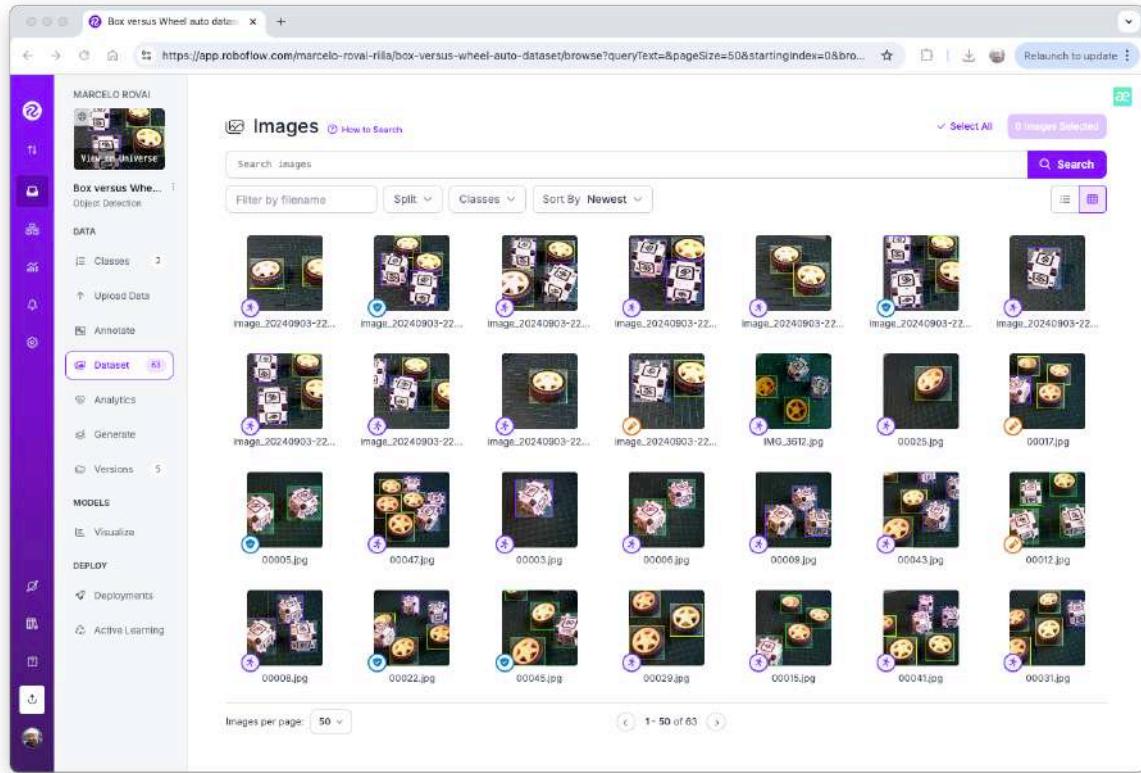
The Label Assist tool can be handy to help on the labeling process.



Note that you should also upload images with only a background, which should be saved w/o any annotations using the Null Tool option.



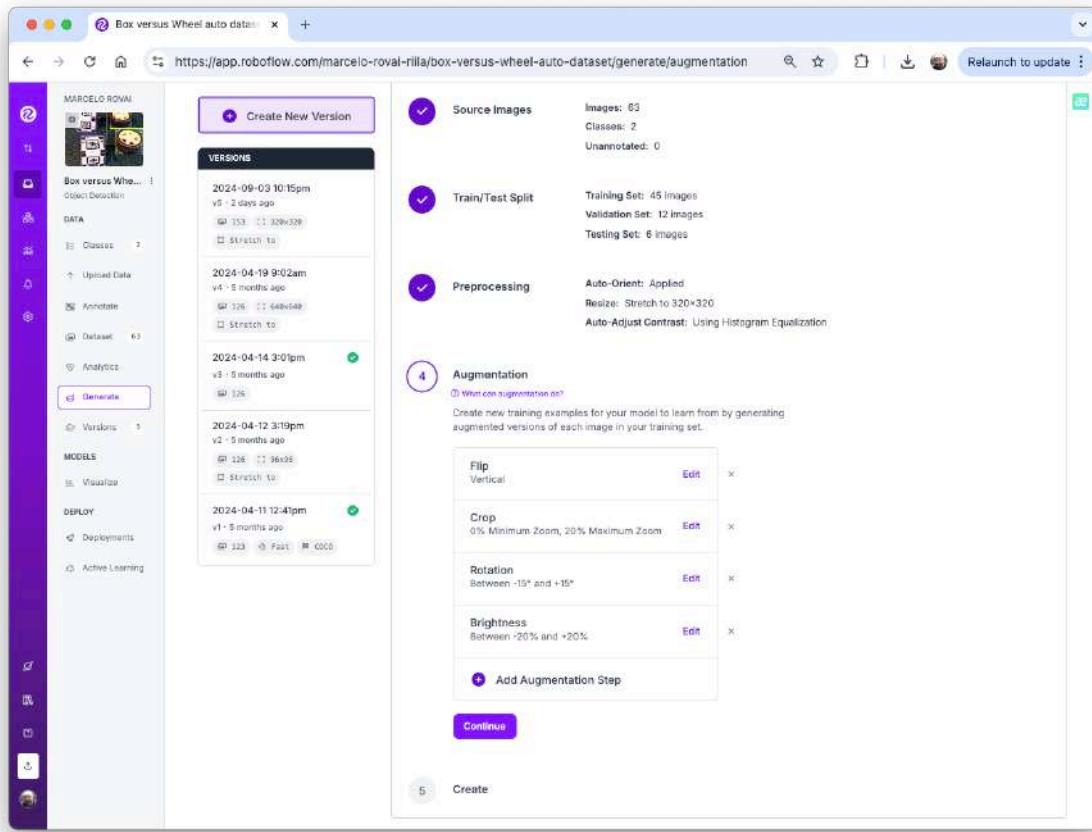
Once all images are annotated, you should split them into training, validation, and testing.



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.

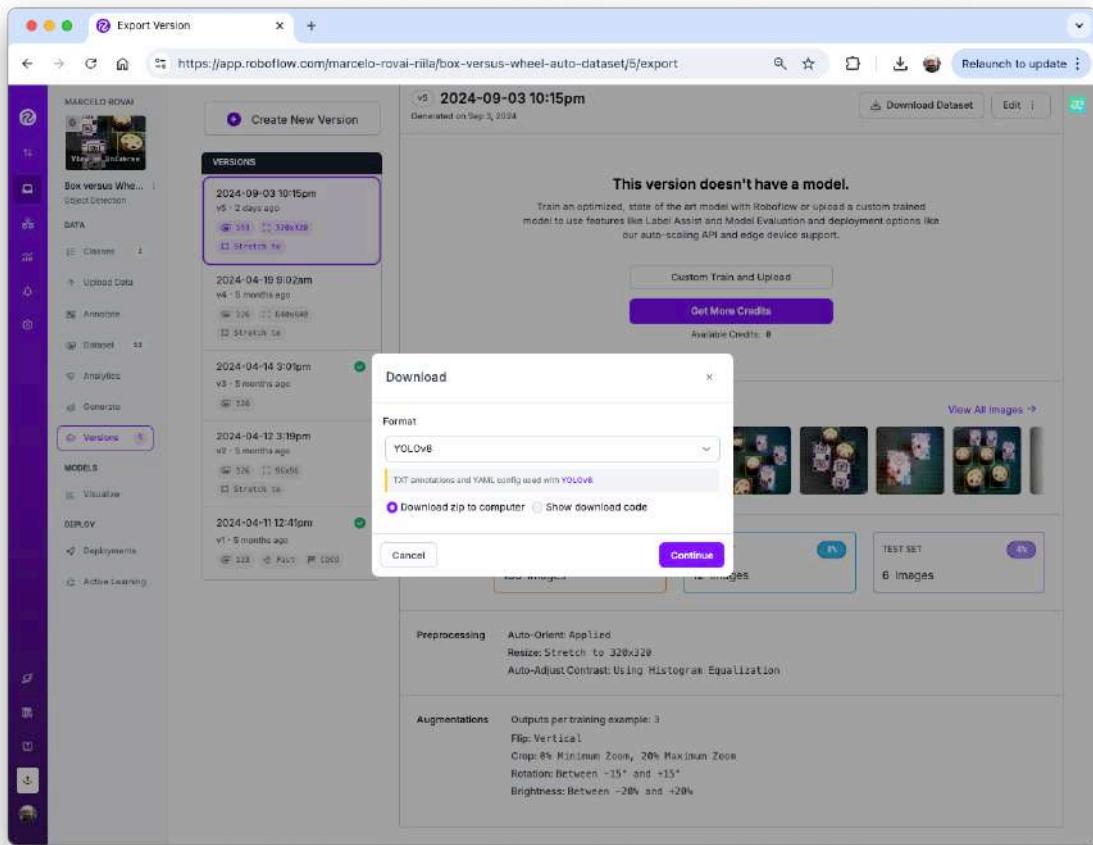


At the end of the process, we will have 153 images.

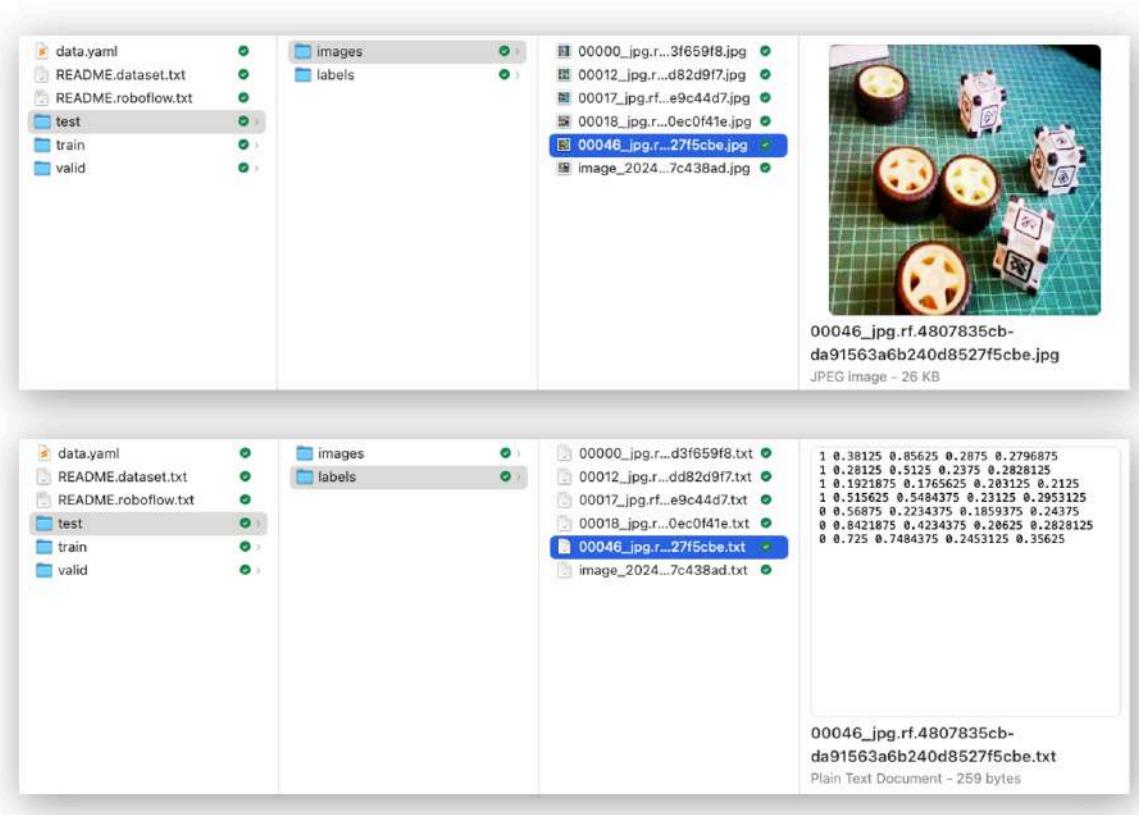
The screenshot shows the Roboflow web interface for a project titled "Box versus Wheel auto dataset".

- Left Sidebar:**
 - Project Overview: Box versus Wheel auto dataset
 - Object Detection
 - DATA:
 - Classes: 2
 - Upload Data
 - Annotate
 - Dataset: 10
 - Analytics
 - Generate
 - VERSIONS: 5
 - MODELS
 - DEPLOY
 - Active Learning
- Central Dashboard:**
 - VERSIONS:**
 - v0 / 2024-09-03 10:15pm
 - v1 - 2 days ago
 - v2 - 5 months ago
 - v3 - 6 months ago
 - v4 - 5 months ago
 - v5 / 2024-04-14 3:01pm
 - v6 - 5 months ago
 - v7 - 5 months ago
 - This version doesn't have a model.**
 - Train an optimized, state of the art model with Roboflow or upload a custom trained model to use features like Label Assist and Model Evaluation and deployment options like our auto-scaling API and edge device support.**
 - Custom Train and Upload**
 - Get More Credits**
 - Available Credits: 0**
 - 153 Total Images**
 - 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
 - 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 (or v11). 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.

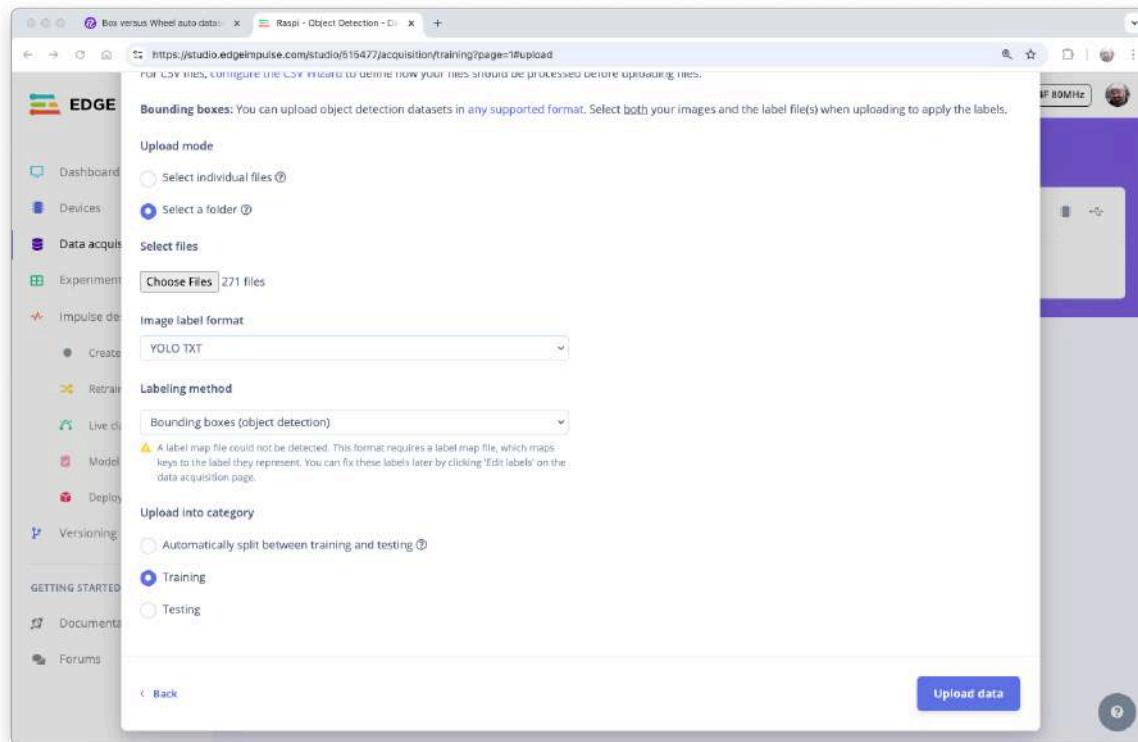
Here, you can clone the project developed for this hands-on lab: [Raspi - Object Detection](#).

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

Uploading the annotated data

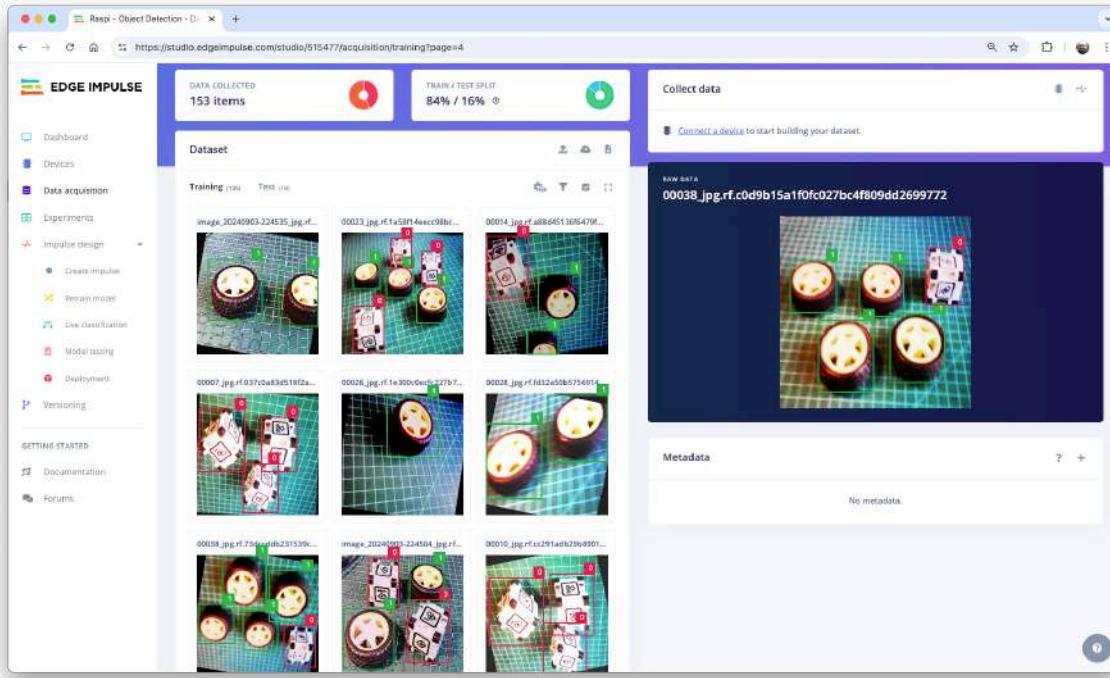
On Studio, go to the **Data acquisition** tab, and on the **UPLOAD DATA** section, upload from your computer the raw dataset.

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%).

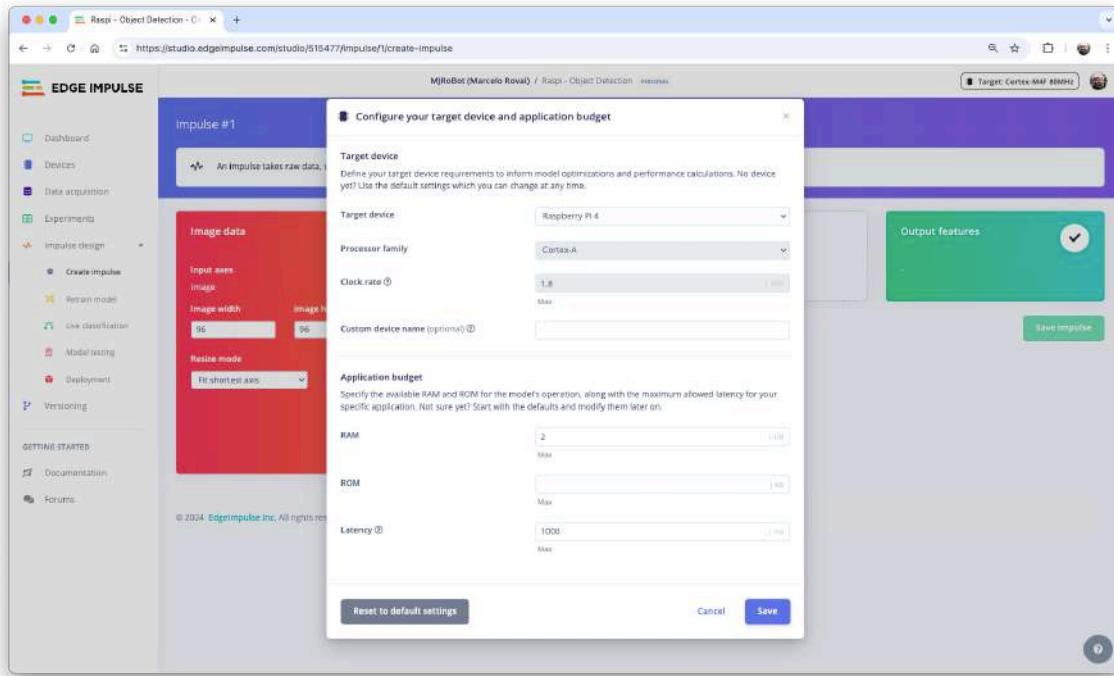
Note that labels will be stored at the labels files 0 and 1 , which are equivalent to **box** and **wheel**.



The Impulse Design

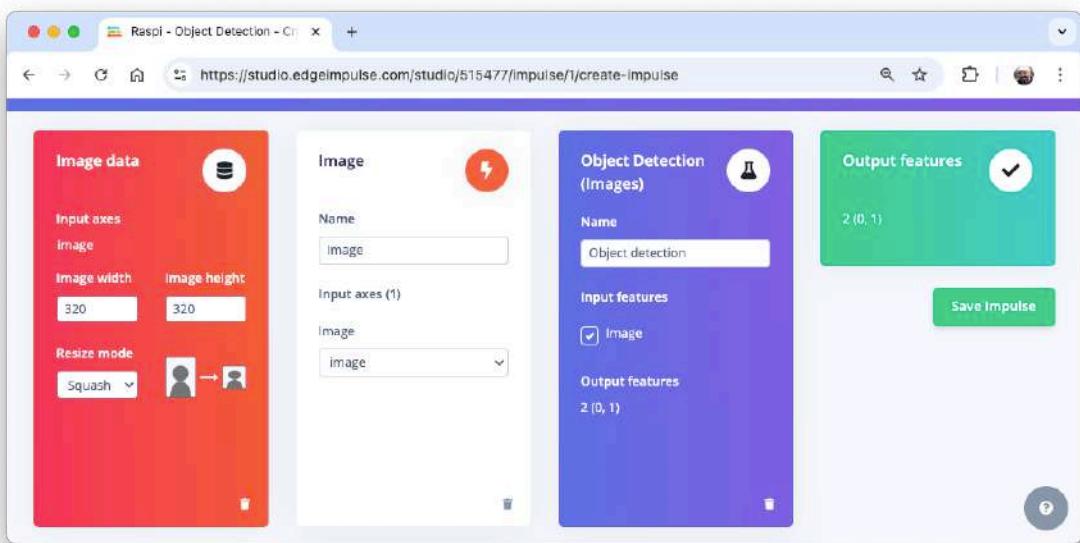
The first thing to define when we enter the **Create impulse** step is to describe the target device for deployment. A pop-up window will appear. We will select Raspberry 4, an intermediary device between the Raspi-Zero and the Raspi-5.

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.



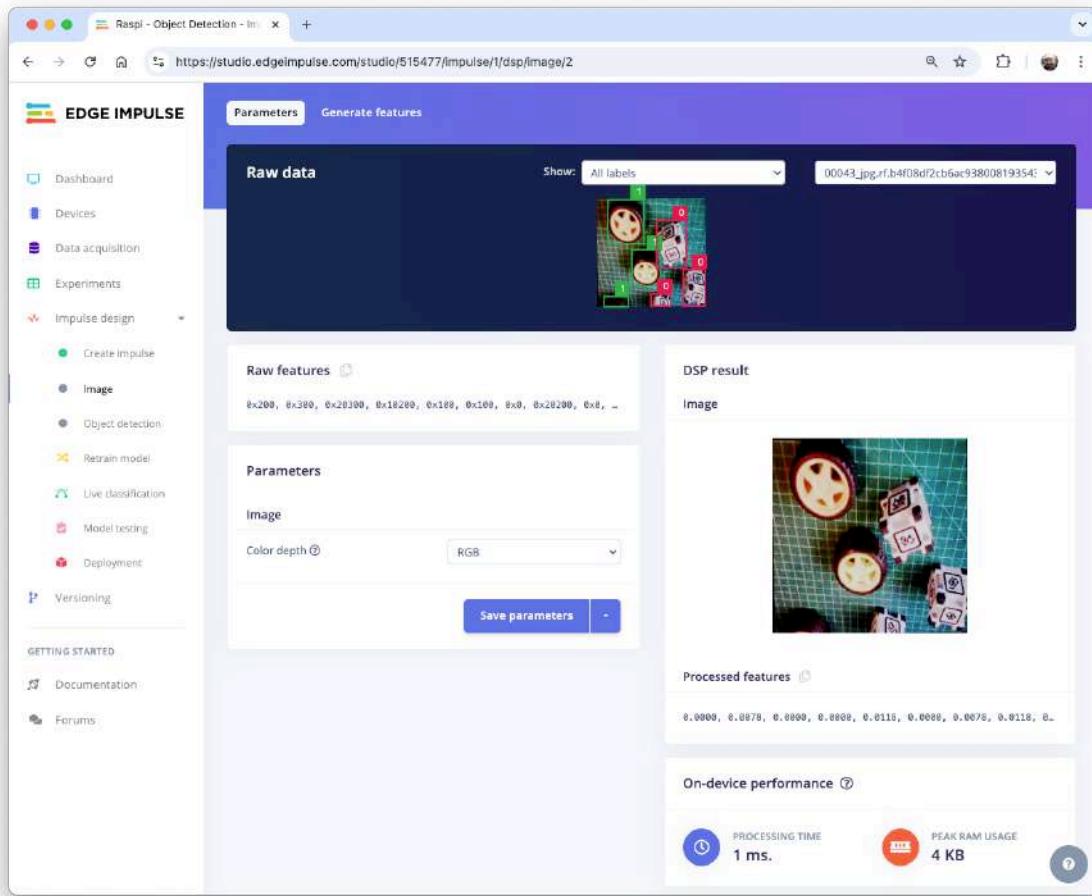
In this phase, you should define how to:

- **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.”

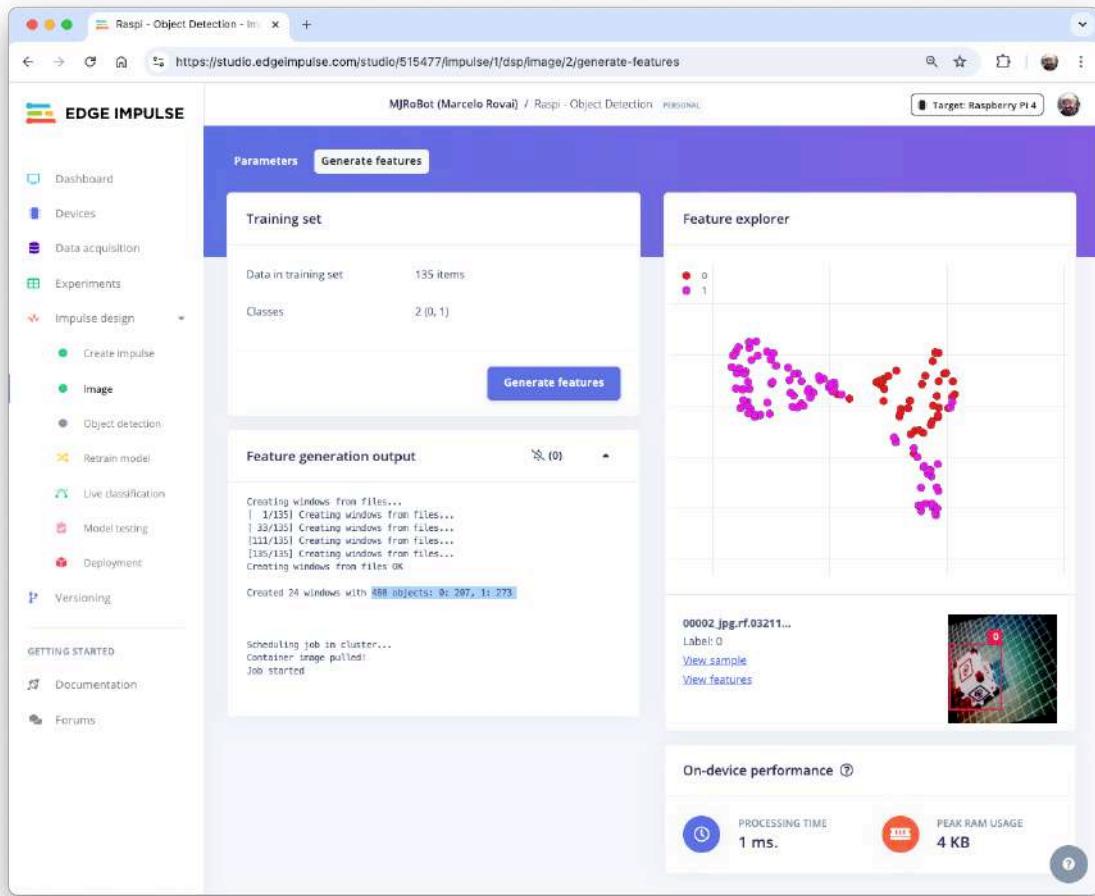


Preprocessing all dataset

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.



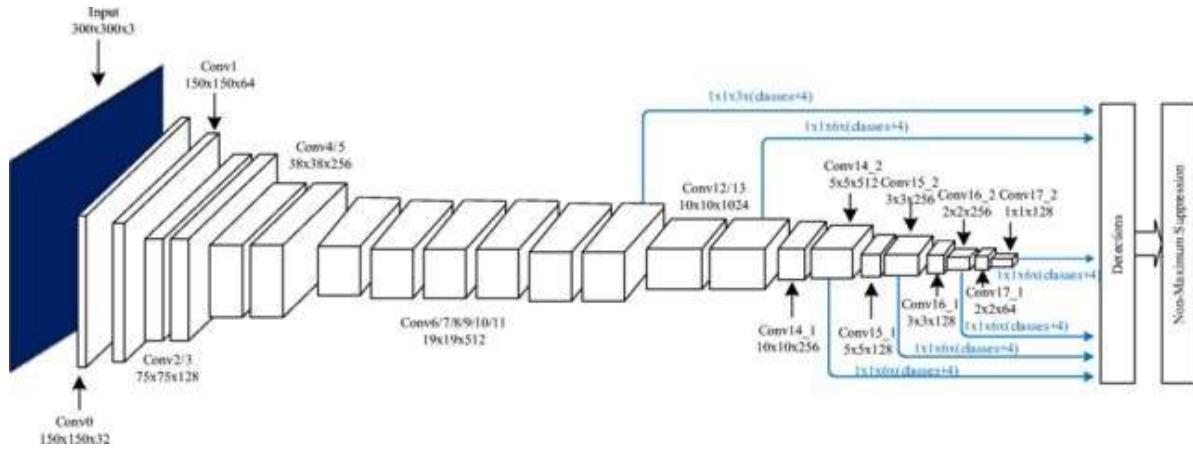
The feature explorer shows that all samples evidence a good separation after the feature generation.

Model Design, Training, and Test

For training, we should select a pre-trained model. Let's use the **MobileNetV2 SSD FPN-Lite (320x320 only)**.

- **Base Network** (MobileNetV2)
- **Detection Network** (Single Shot Detector or SSD)
- **Feature Extractor** (FPN-Lite)

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 approximately 3.7 MB 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.

Neural Network settings

Training settings

Number of training cycles	25
Use learned optimizer	<input type="checkbox"/>
Learning rate	0.15
Training processor	CPU

Advanced training settings

Validation set size	20	9%
Split train/validation set on metadata key	<input type="text"/>	
Batch size	32	
Profile init model	<input checked="" type="checkbox"/>	

Neural network architecture

Input layer (307,396 features)

MobileNetV2 SSD-FPN-Lite 320x320

Choose a different model

Output layer (2 classes)

Model Model version: Unoptimized (float32) ▾

Last training performance (validation set)

PRECISION SCORE 88.8%

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 ▾

INFERRING 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:

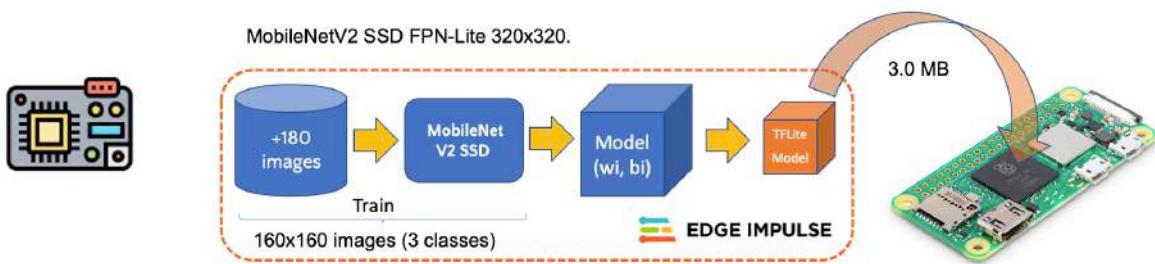
- **TFLite model**, which lets deploy the trained model as `.tflite` for the Raspi to run it using Python.
- **Linux (AARCH64)**, a binary for Linux (AARCH64), implements the Edge Impulse Linux protocol, which lets us run our models on any Linux-based development board, with SDKs for Python, for example. 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 tflite_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:

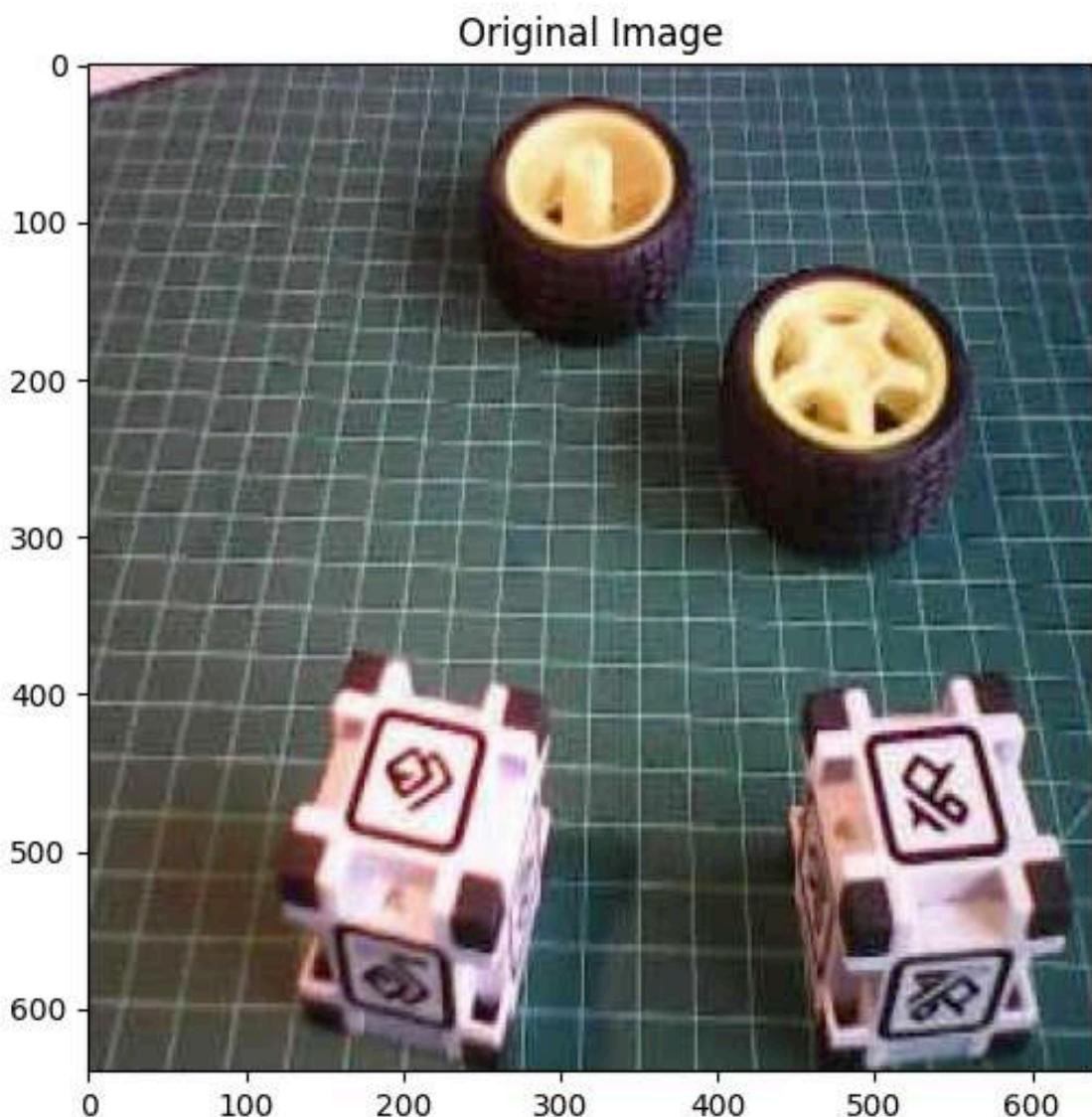
```
input_dtype = input_details[0]['dtype']
input_dtype

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

```

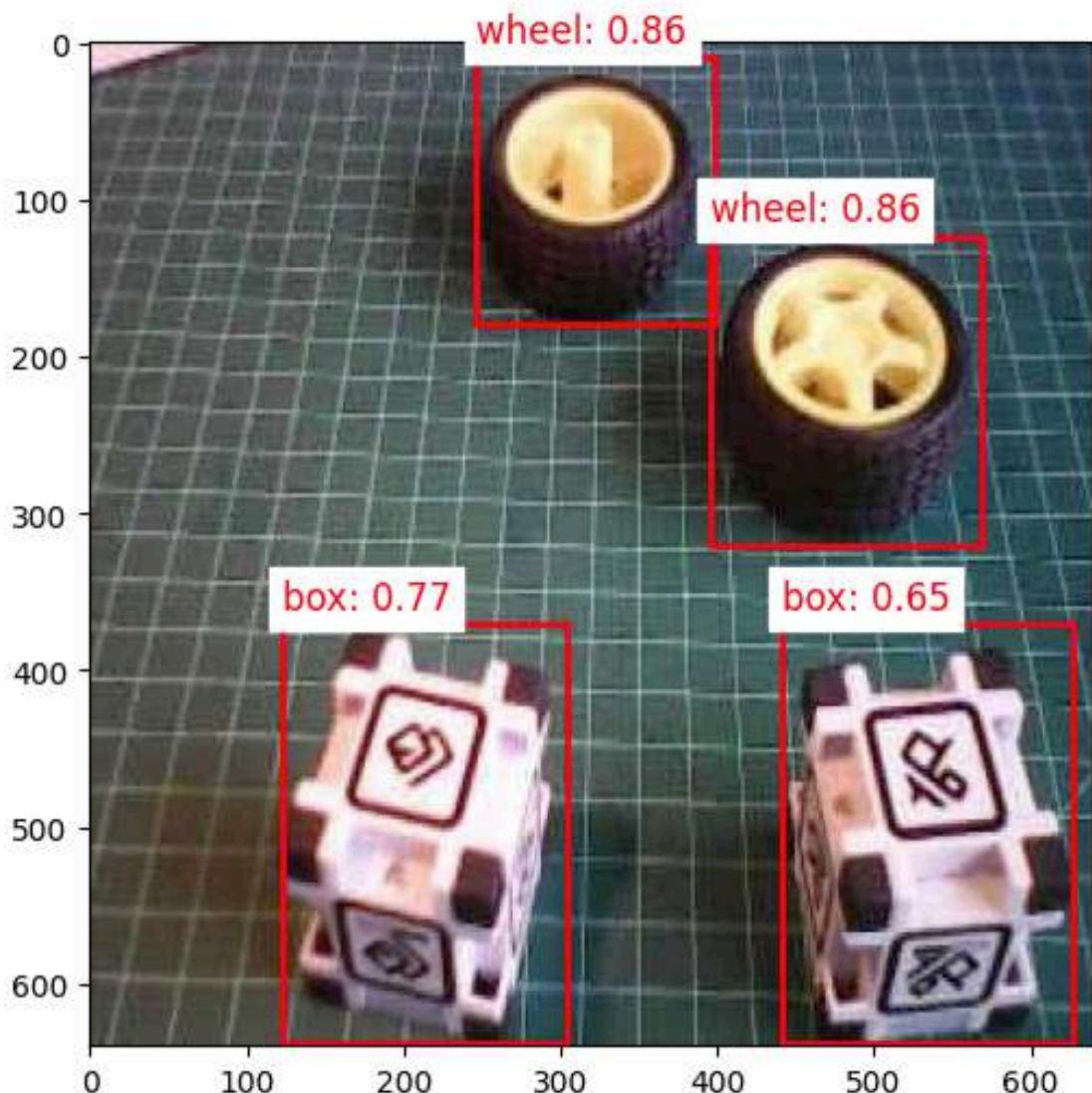
From the results, we can see that 4 objects were detected: two with class ID 0 (**box**) and two with class ID 1 (**wheel**), what is correct!

Let's visualize the result for a threshold of 0.5

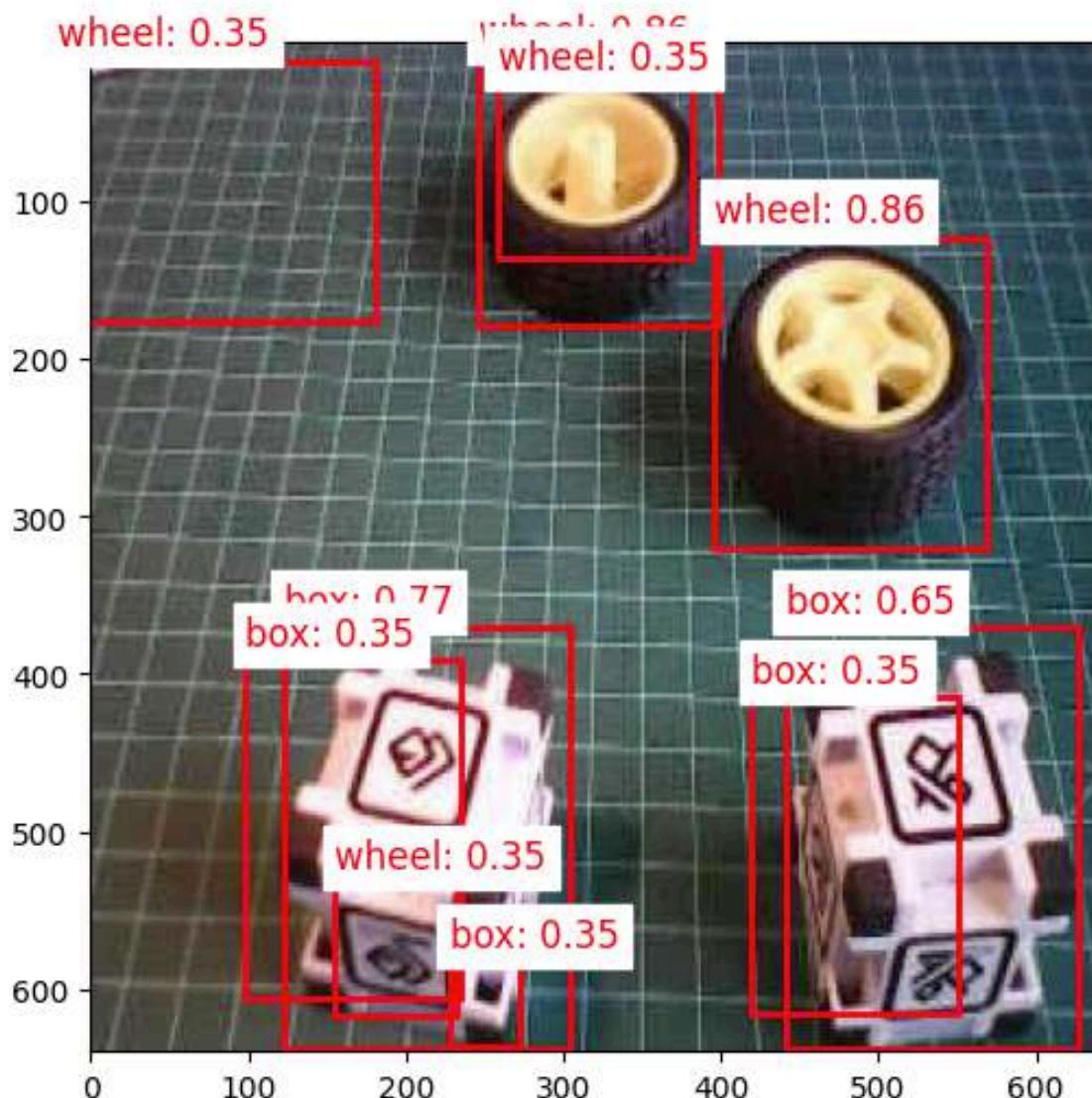
```

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)

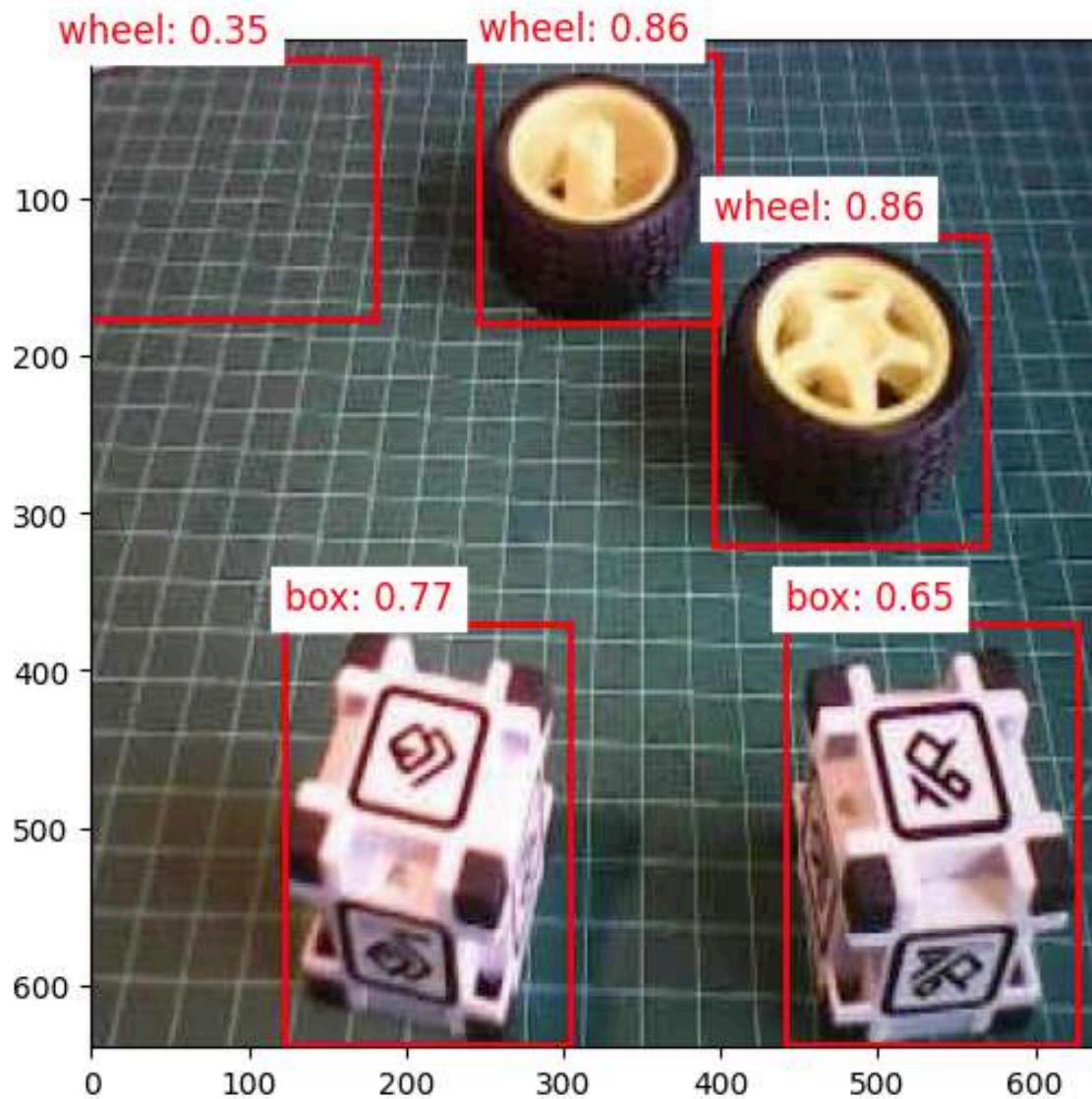
    # Inference on Raspi-Zero
    start_time = time.time()
    interpreter.set_tensor(input_details[0]['index'], input_data)
    interpreter.invoke()
    end_time = time.time()
    inference_time = (end_time - start_time) * 1000 # Convert to ms
    print ("Inference time: {:.1f}ms".format(inference_time))

    # 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

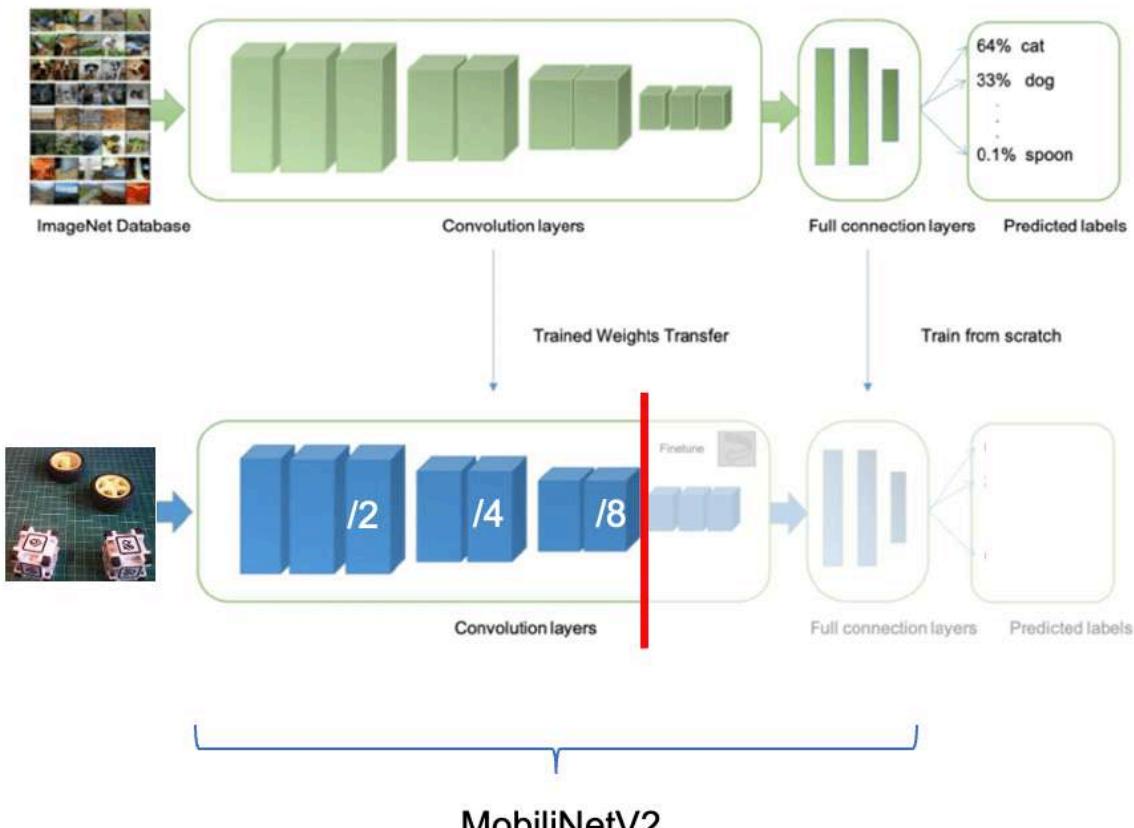
The inference with the SSD MobileNet model worked well, but the latency was significantly high. The inference varied from 0.5 to 1.3 seconds on a Raspi-Zero, which means around or less than 1 FPS (1 frame per second). One alternative to speed up the process is to use FOMO (Faster Objects, More Objects).

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.

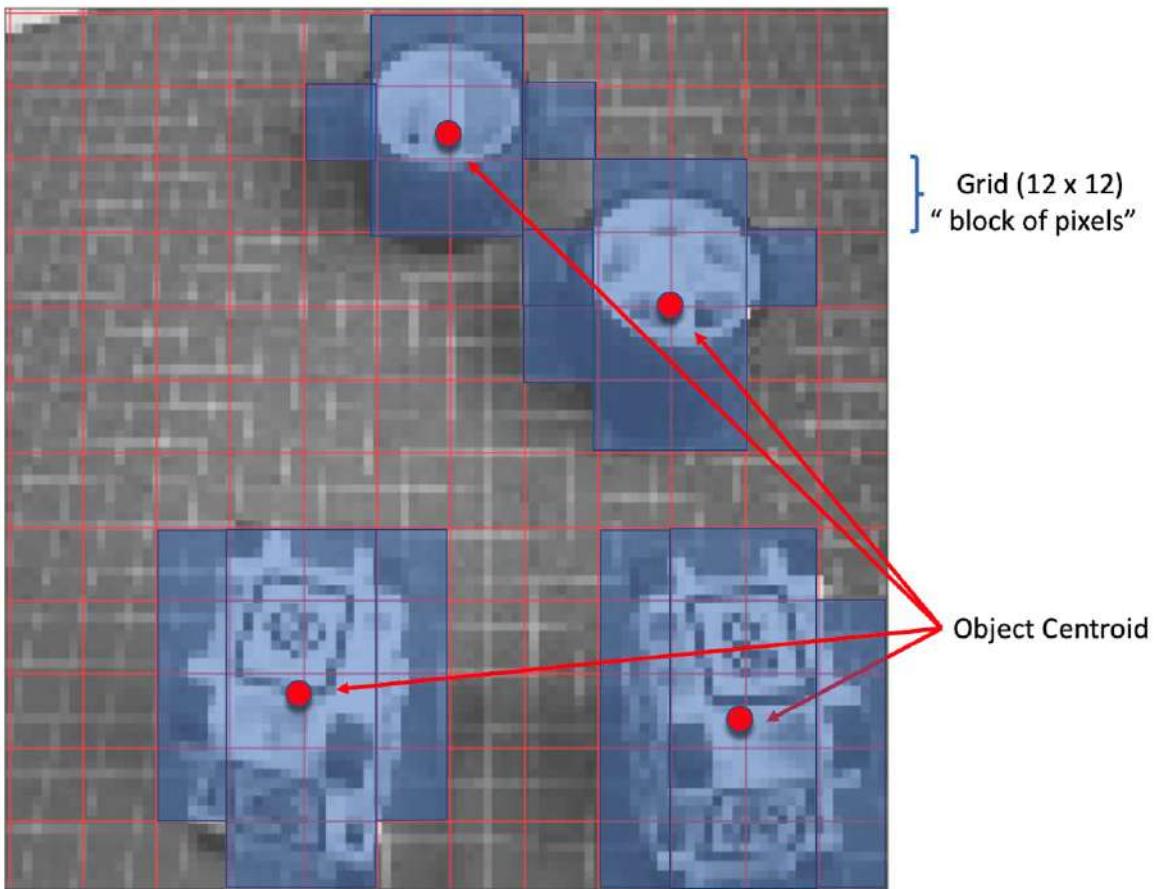


Once these features are extracted, FOMO's simpler architecture, focused on center-point de-

detection, interprets the feature map to determine where objects are located in the image. The output is a grid of cells, where each cell represents whether or not an object center is detected. The model outputs one or more confidence scores for each cell, indicating the likelihood of an object being present.

Let's see how it works on an image.

FOMO divides the image into blocks of pixels using a factor of 8. For the input of 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.

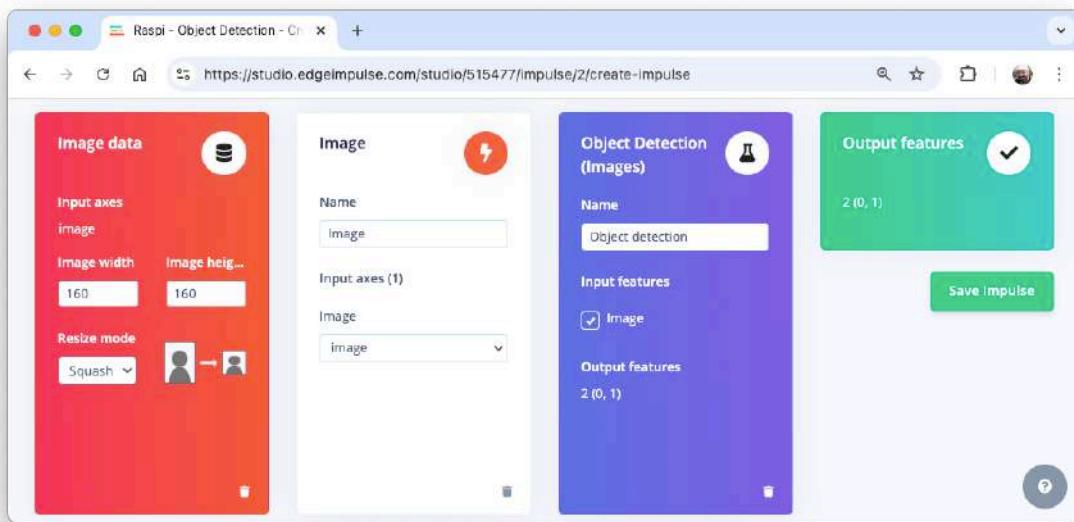


Trade-off Between Speed and Precision:

- **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.

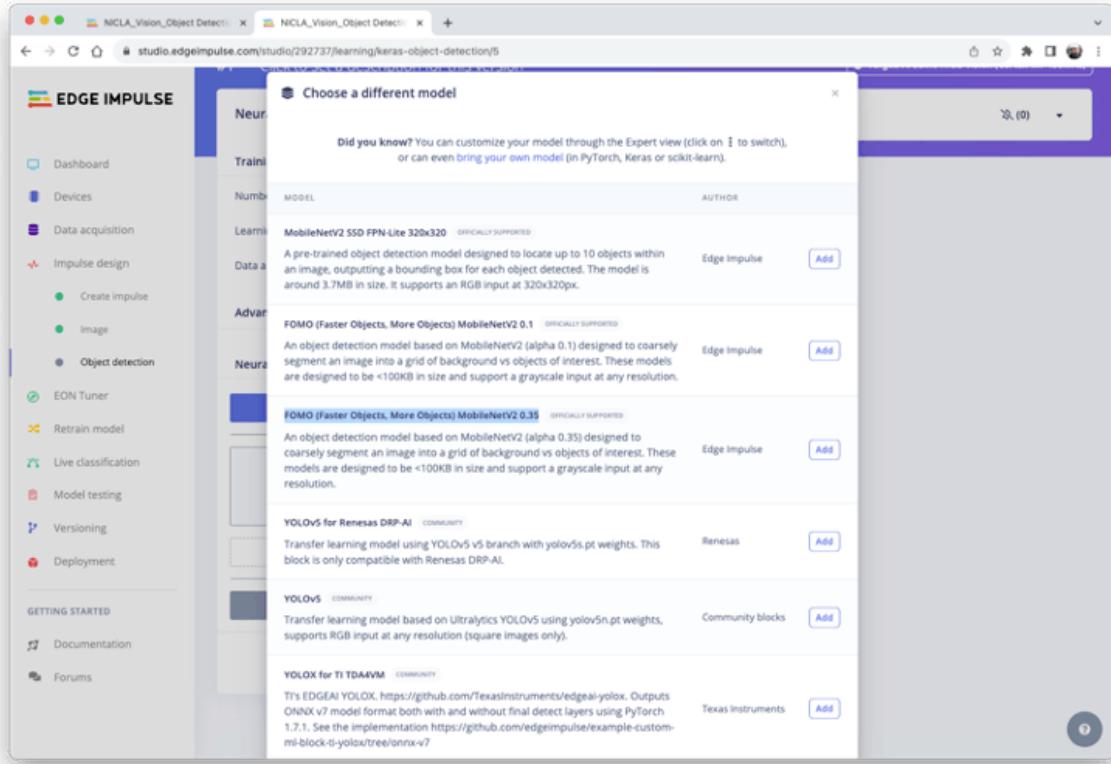
Impulse Design, new Training and Testing

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.

Model Model version: Quantized (int8) ▾

Last training performance (validation set)

F1 SCORE 93.3%

	BACKGROUND	0	1
BACKGROUND	100.0%	0.0%	0%
0	9.1%	10.9%	0%
1	10.8%	0%	89.2%
F1 SCORE	1.00	0.92	0.94

Metrics (validation set)

METRIC	VALUE
Precision (non-background)	0.97
Recall (non-background)	0.90
F1 Score (non-background)	0.93

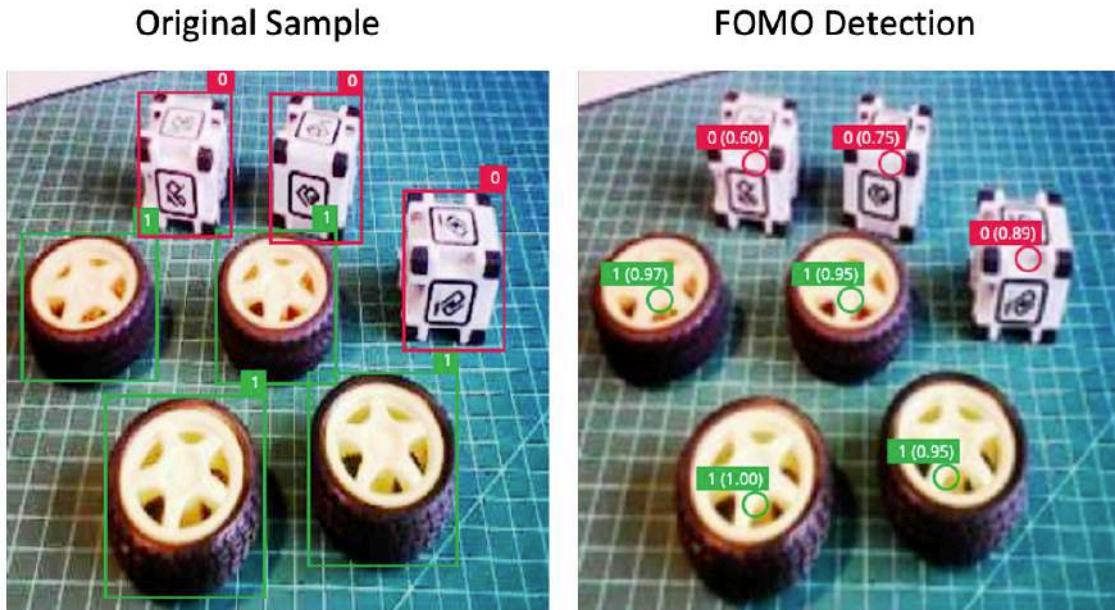
On-device performance

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 "Rasp - Object Detection". The main content area displays deployment options for a "Linux (AARCH64)" target. It includes sections for "DEFAULT DEPLOYMENT" (with a yellow Linux icon), "DEPLOY TO ANY LINUX-BASED DEVELOPMENT BOARD" (with a Node.js icon), and "MODEL OPTIMIZATIONS". The "Quantized (int8)" section is selected, showing latency and RAM usage for IMAGE and OBJECT DETECTION across CPU, RAM, and FLASH. The "Unoptimized (float32)" section is also shown for comparison. A "Build" button is at the bottom.

DEFAULT DEPLOYMENT

Linux (AARCH64)

A binary for Linux (AARCH64) that implements the Edge Impulse Linux protocol.

DEPLOY TO ANY LINUX-BASED DEVELOPMENT BOARD

Edge Impulse for Linux lets you run your models on any Linux-based development board, with SDKs for Node.js, Python, Go and C++ to integrate your models quickly into your application.

1. Install the Edge Impulse Linux CLI
2. Run `edge-impulse-linux-client` (run with `--clear` to switch projects)

See the documentation for more information and setup instructions. Alternatively, you can download your model for Linux (AARCH64) below:

MODEL OPTIMIZATIONS

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

	IMAGE	OBJECT DETECTION	TOTAL
LATENCY	1 ms.	8 ms.	9 ms.
RAM	4.3M	424.1K	438.3K
FLASH	-	78.0K	-
ACCURACY			94.6%

	IMAGE	OBJECT DETECTION	TOTAL
LATENCY	1 ms.	12 ms.	11 ms.
RAM	4.3M	1.4M	3.4M
FLASH	-	103.4K	-
ACCURACY			94.6%

Estimate for Raspberry Pi 4 Change target

Build

The model will be automatically downloaded to your computer.

On our Raspi, let's create a new working area:

```
cd ~  
cd Documents  
mkdir EI_Linux  
cd EI_Linux  
mkdir models  
mkdir images
```

Rename the model for easy identification:

For example, `raspi-object-detection-linux-aarch64-FOMO-int8.eim` and transfer it to the new Raspi `./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)

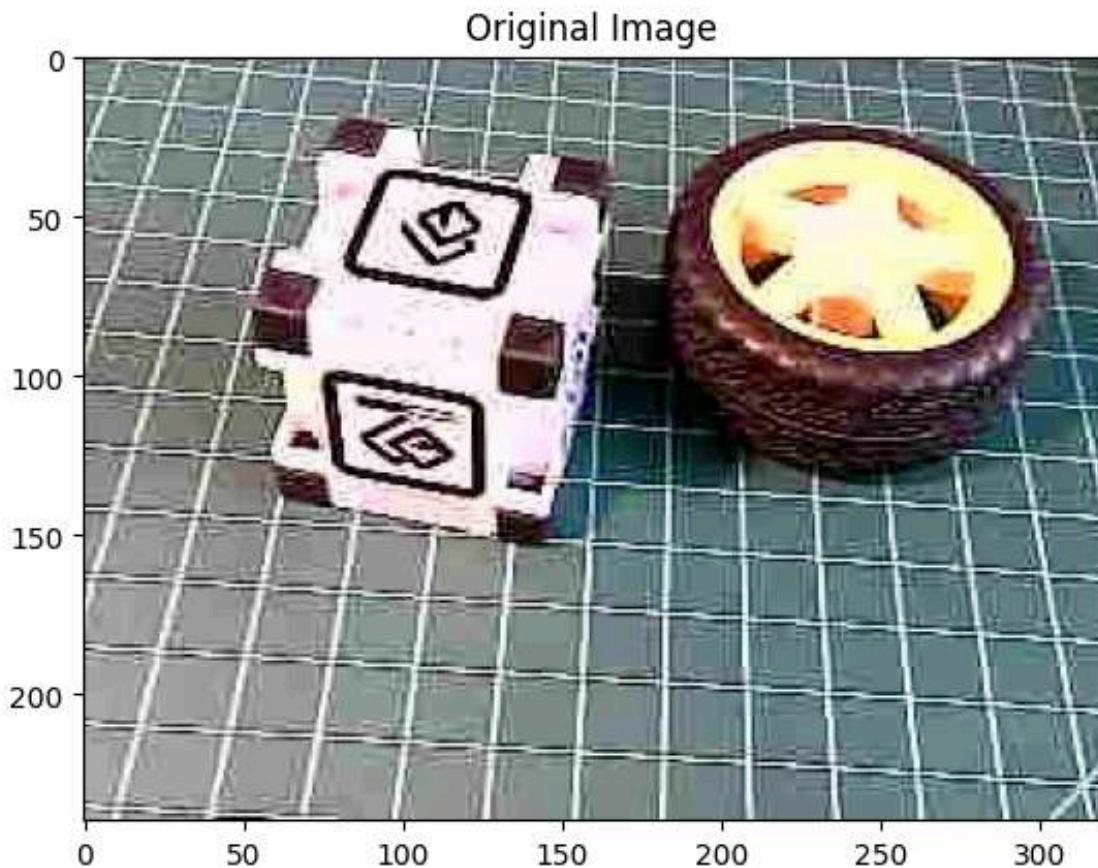
# Initialize model
model_info = runner.init()
```

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).

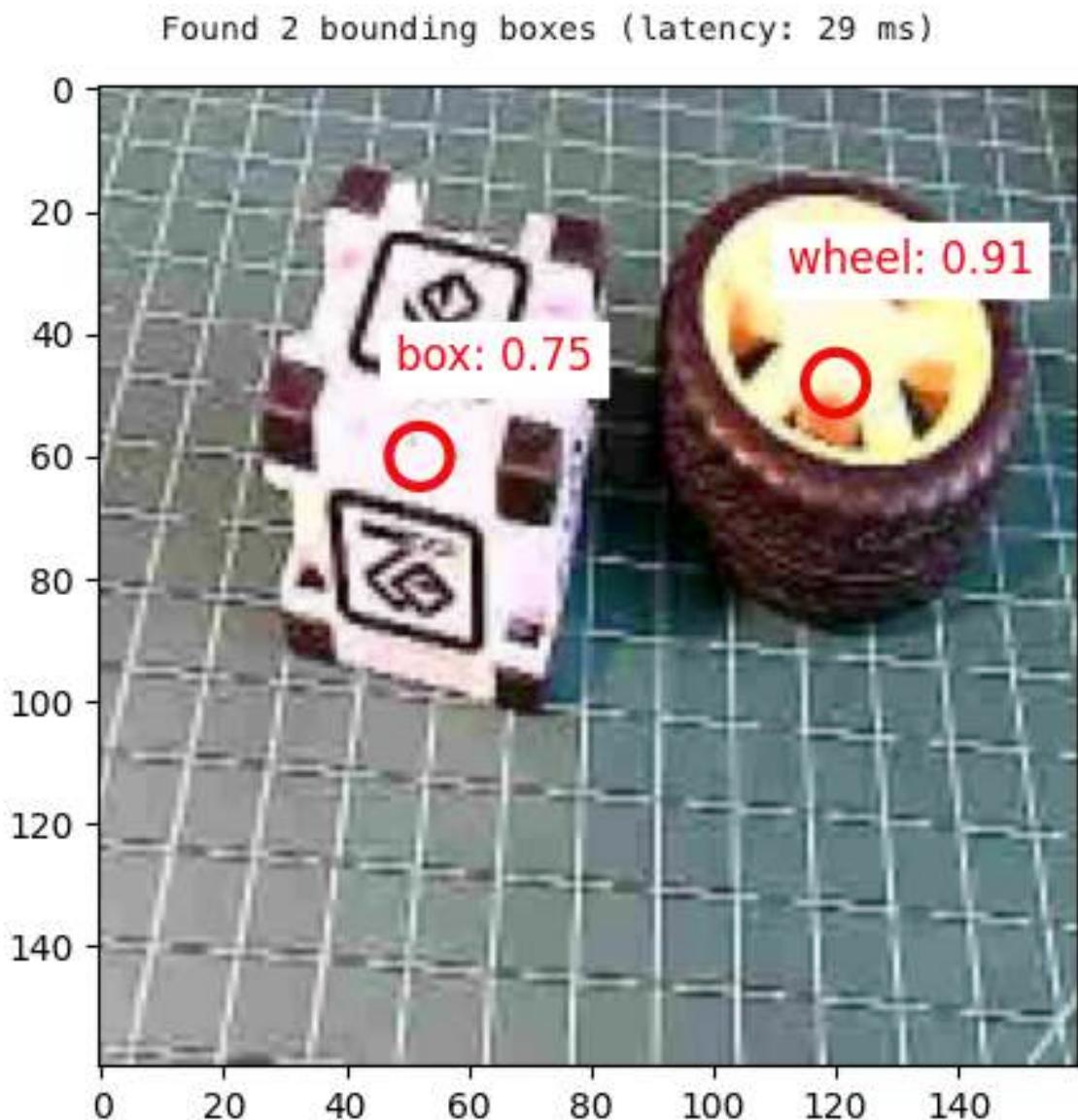
```
print('\tFound %d bounding boxes (latency: %d ms)' % (
    len(res["result"]["bounding_boxes"]),
    res['timing']['dsp'] + res['timing']['classification']))
plt.figure(figsize=(5,5))
plt.imshow(cropped)

# 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()
```



Conclusion

This chapter has explored the implementation of a custom object detector on edge devices, such as 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, as SSD-

MobileNet and FOMO, 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 with Edge Impulse Studio and Ultralytics and deploying them on a Raspberry Pi.
3. **Optimization Techniques:** To improve inference speed on edge devices, we explored various optimization methods, such as model quantization (TFLite int8).
4. **Performance Considerations:** Throughout the lab, we discussed the balance between model accuracy and inference speed, a critical consideration for edge AI applications.

As discussed before, the ability to perform object detection on edge devices opens up numerous possibilities across various domains, including precision agriculture, industrial automation, quality control, 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:

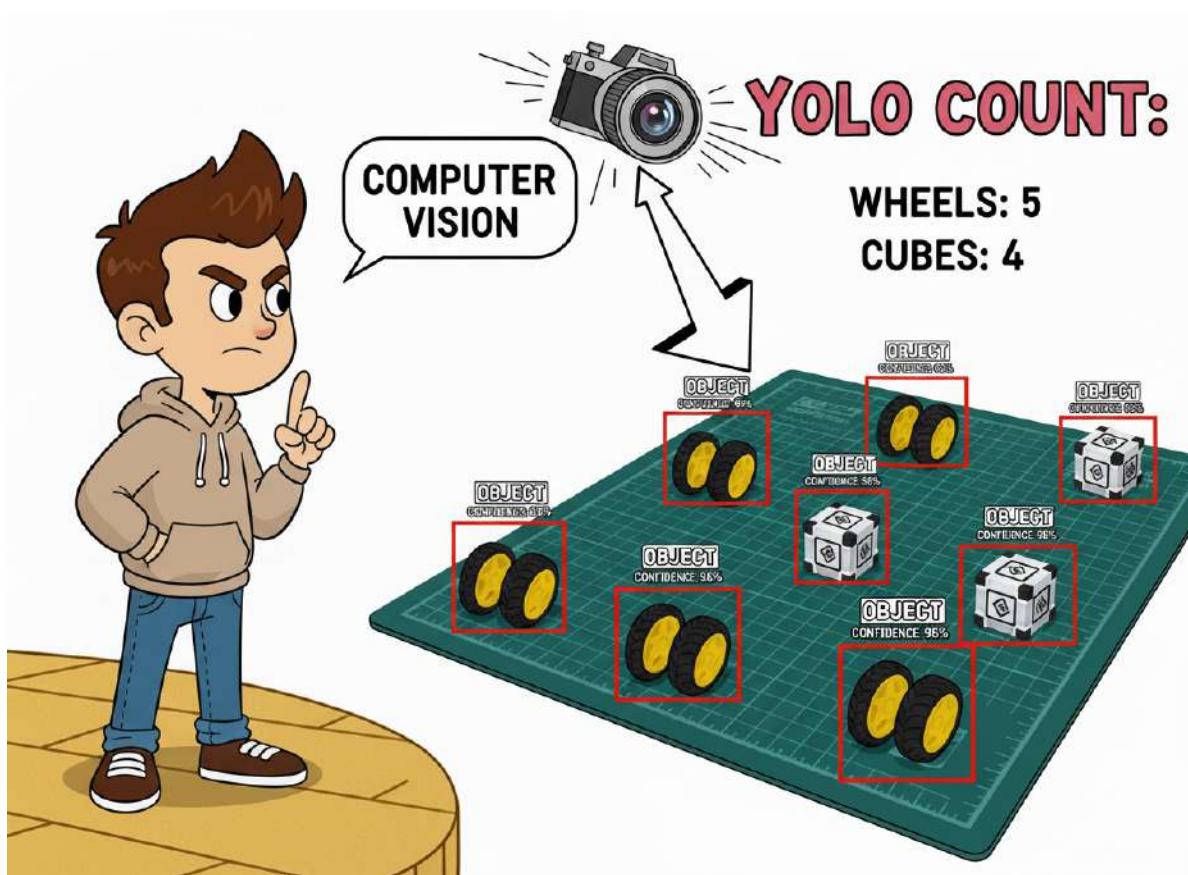
- Implementing multi-model pipelines for more complex tasks
- Exploring hardware acceleration options for Raspberry Pi
- Integrating object detection with other sensors for more comprehensive edge AI systems
- Developing edge-to-cloud solutions that leverage both local processing and cloud resources

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 \(“Box versus Wheel”\)](#)
- [FOMO - EI Linux Notebook on a Raspi](#)
- [Edge Impulse Project - SSD MobileNet and FOMO](#)
- [Python Scripts](#)
- [Models](#)

Computer Vision Applications with YOLO



Exploring a YOLO Models using Ultralytics

In this chapter, we will explore YOLOv8 and v11. [Ultralytics YOLO](#) (v8 and v11) are versions of the acclaimed real-time object detection and image segmentation model, YOLO. YOLOv8 and v11 are 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:

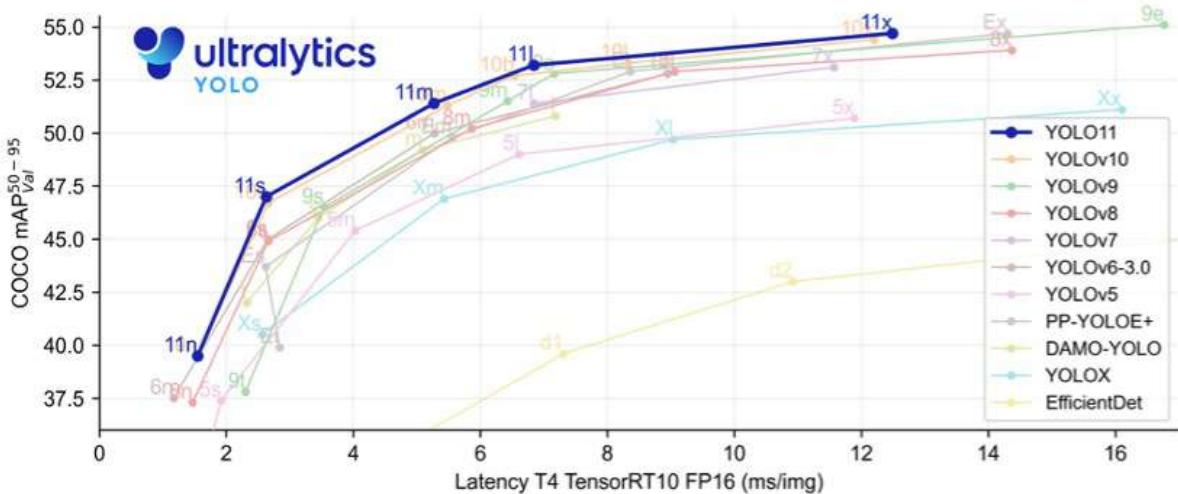
- YOLO employs a single neural network to process the entire image. This network divides the image into a grid and, for each grid cell, directly predicts bounding boxes and associated class probabilities. This end-to-end training improves speed and simplifies the model architecture.

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 YOLOv12. 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.
- YOLOv11 offers substantial improvements in accuracy, speed, and parameter efficiency compared to prior versions such as YOLOv8 and YOLOv10, making it one of the most versatile and powerful real-time object detection models available as of 2025



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.

7. Model Capabilities

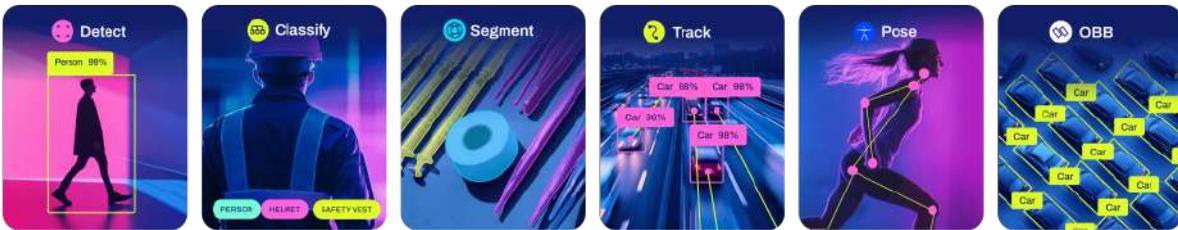
YOLO models support multiple computer vision tasks:

- Object Detection:** Identifying and localizing objects with bounding boxes
- Instance Segmentation:** Pixel-level object segmentation
- Pose Estimation:** Human pose keypoint detection

- **Classification:** Image classification tasks

Ultralitics YOLO Detect, Segment, and Pose models pre-trained on the [COCO](#) dataset, and [Classify](#) on the [ImageNet](#) dataset.

[Track](#) mode is available for all Detect, Segment, and Pose models. The latest versions of YOLO can also perform OBB, which stands for Oriented Bounding Box, a rectangular box in computer vision that can rotate to match the orientation of an object within an image, providing a much tighter and more precise fit than traditional axis-aligned bounding boxes.



Available Model Sizes

YOLO offers several model variants optimized for different use cases, for example. The YOLOv8:

- **YOLOv8n (Nano):** Smallest model, fastest inference, lowest accuracy
- **YOLOv8s (Small):** Balanced performance for edge devices
- **YOLOv8m (Medium):** Higher accuracy, moderate computational requirements
- **YOLOv8l (Large):** High accuracy, requires more computational resources
- **YOLOv8x (Extra Large):** Highest accuracy, most computational intensive

For Raspberry Pi applications, YOLOv8n or YOLO11n are typically the best choice due to their optimized size and speed.

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 YOLO

```
python3 -m venv ~/yolo --system-site-packages  
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 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, for example, the YOLOV8n model (the smallest in the family) at the Terminal (CLI):

```
yolo predict model='yolov8n' source='https://ultralytics.com/images/bus.jpg'
```

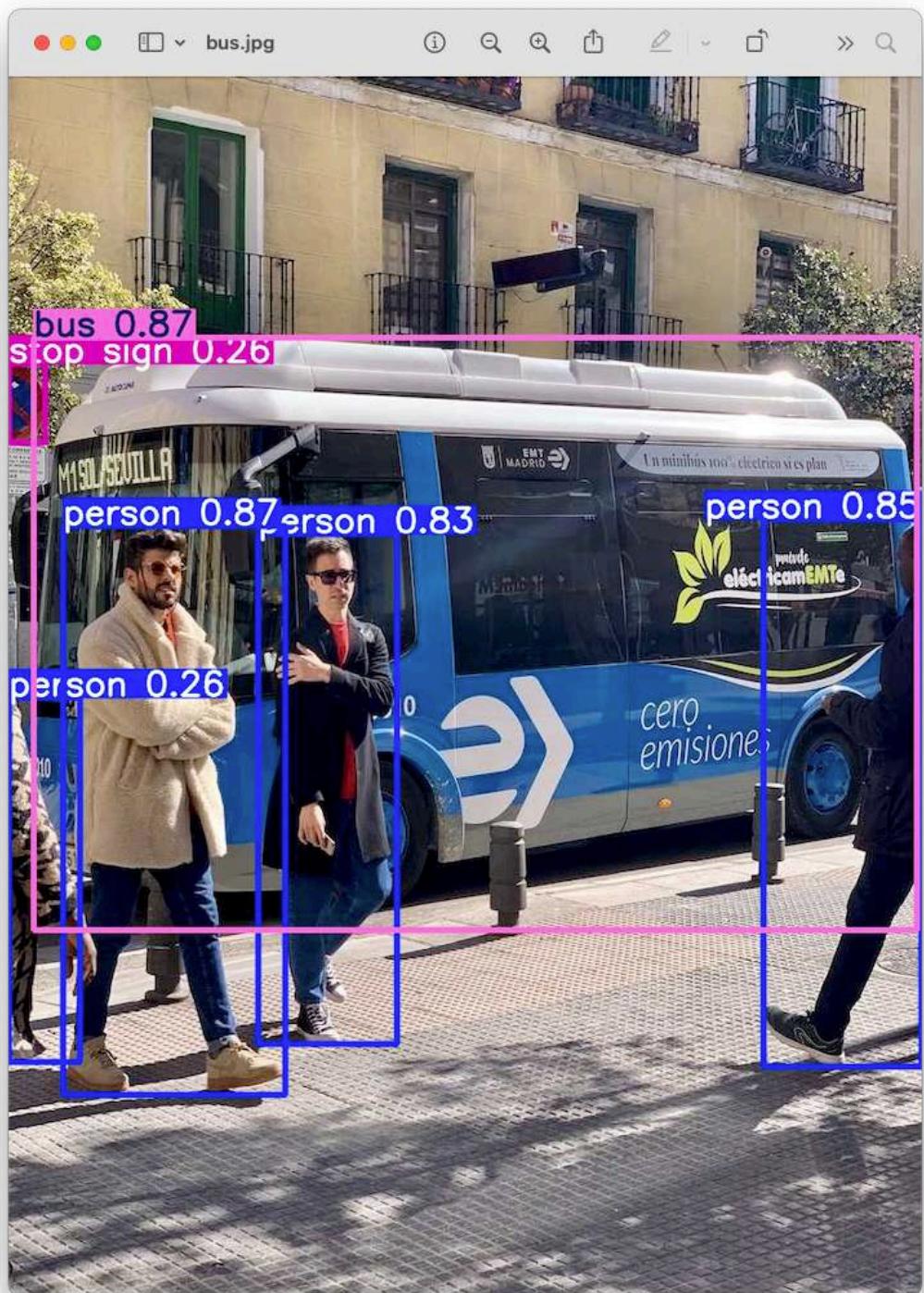
Note that the first time we invoke a model, it will automatically be downloaded to the current directory.

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

```
mjrovai@raspi-zero:~/Documents/YOLO$ cd ~
mjrovai@raspi-zero:~$ source ~/yolo/bin/activate
(yolo) mjrovai@raspi-zero:~$ cd Documents/YOLO/
(yolo) mjrovai@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]
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
```

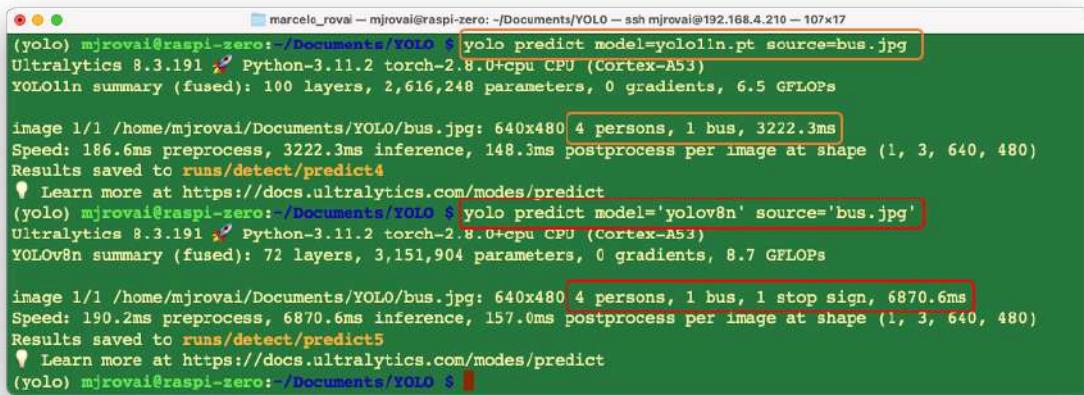
Also, we got a message that `Results saved to runs/detect/predict`. Inspecting that directory, we can see a new image saved (bus.jpg). Let's download it from the Raspi to our desktop for inspection:



So, the Ultralytics YOLO is correctly installed on our Raspberry Pi. Note that on the Raspberry Pi Zero, an issue is the high latency for this inference, which takes several seconds, even with the most compact model in the family (YOLOv8n).

Testing with the YOLOv11

The procedure is the same as we did with version v8. As a comparison, we can see that the YOLOv11 is faster than the v8, but seems a little less precise, as it does not detect the “stop sign” as the v8.



```
marcelo_royal@raspi-zero: ~/Documents/YOLO -- ssh mjrovai@192.168.4.210 -- 107x17
(yolo) mjrovai@raspi-zero:~/Documents/YOLO$ yolo predict model=yolol1n.pt source=bus.jpg
Ultralytics 8.3.191 Python-3.11.2 torch-2.8.0+cpu CPU (Cortex-A53)
YOLOl1n summary (fused): 100 layers, 2,616,248 parameters, 0 gradients, 6.5 GFLOPs

image 1/1 /home/mjrovai/Documents/YOLO/bus.jpg: 640x480 4 persons, 1 bus, 3222.3ms
Speed: 186.6ms preprocess, 3222.3ms inference, 148.3ms postprocess per image at shape (1, 3, 640, 480)
Results saved to runs/detect/predict4
💡 Learn more at https://docs.ultralytics.com/modes/predict
(yolo) mjrovai@raspi-zero:~/Documents/YOLO$ yolo predict model='yolov8n' source='bus.jpg'
Ultralytics 8.3.191 Python-3.11.2 torch-2.8.0+cpu CPU (Cortex-A53)
YOLOv8n summary (fused): 72 layers, 3,151,904 parameters, 0 gradients, 8.7 GFLOPs

image 1/1 /home/mjrovai/Documents/YOLO/bus.jpg: 640x480 4 persons, 1 bus, 1 stop sign, 6870.6ms
Speed: 190.2ms preprocess, 6870.6ms inference, 157.0ms postprocess per image at shape (1, 3, 640, 480)
Results saved to runs/detect/predict5
💡 Learn more at https://docs.ultralytics.com/modes/predict
(yolo) mjrovai@raspi-zero:~/Documents/YOLO$
```

Export Models to NCNN format

Deploying computer vision models on edge devices with limited computational power, such as the Raspberry Pi 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 of deployment and use on mobile phones, and it 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).

Let's move the downloaded YOLO models to the `./models` folder and the `bus.jpg` to `./images`.

And convert our models and rerun the inferences:

1. Export the YOLO PyTorch models to NCNN format, creating: `yolov8n_ncnn_model` and `yolo11n_ncnn_model`

```
yolo export model=./models/yolov8n.pt format=ncnn  
yolo export model=./models/yolo11n.pt format=ncnn
```

2. Run inference with the exported models:

```
yolo predict task=detect model='./models/yolov8n_ncnn_model' source='./images/bus.jpg'  
yolo predict task=detect model='./models/yolo11n_ncnn_model' source='./images/bus.jpg'
```

The first inference, when the model is loaded, typically has a high latency; however, from the second inference, it is possible to note that the inference time decreases.

We can now realize that neither model detects the “Stop Signal”, with YOLOv11 being the fastest. The optimized models are more rapid but also less accurate.

```
marcosio_royal ~ mirovai@raspi-zero:~/Documents/YOLO - ssh mirovai@192.168.4.210 - 135x9  
(yolo) mirovai@raspi-zero:~/Documents/YOLO $ yolo predict task=detect model='./models/yolov8n_ncnn_model' source='./images/bus.jpg'  
Ultralytics 8.3.191 Python-3.11.2 torch-2.8.0+cpu CPU (Cortex-A53)  
Loading ./models/yolov8n_ncnn_model for NCNN inference...  
  
image 1/1 /home/mirovai/Documents/YOLO/images/bus.jpg: 640x640 4 persons, 1 bus, 3566.8ms  
Speed: 484.4ms preprocess, 3566.8ms inference, 1519.0ms postprocess per image at shape (1, 3, 640, 640)  
Results saved to runs/detect/predict10  
💡 Learn more at https://docs.ultralytics.com/modes/predict  
(yolo) mirovai@raspi-zero:~/Documents/YOLO $  
  
marcosio_royal ~ mirovai@raspi-zero:~/Documents/YOLO - ssh mirovai@192.168.4.210 - 138x9  
(yolo) mirovai@raspi-zero:~/Documents/YOLO $ yolo predict task=detect model=models/yolo11n_ncnn_model imgsz=640 source='./images/bus.jpg'  
Ultralytics 8.3.191 Python-3.11.2 torch-2.8.0+cpu CPU (Cortex-A53)  
Loading models/yolo11n_ncnn_model for NCNN inference...  
  
image 1/1 /home/mirovai/Documents/YOLO/images/bus.jpg: 640x640 4 persons, 1 bus, 1034.6ms  
Speed: 1764.9ms preprocess, 1034.6ms inference, 3406.3ms postprocess per image at shape (1, 3, 640, 640)  
Results saved to runs/detect/predict11  
💡 Learn more at https://docs.ultralytics.com/modes/predict  
(yolo) mirovai@raspi-zero:~/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:

```
python
```

Now, we should call the YOLO library from Ultralitics and load the model:

```
from ultralytics import YOLO
model = YOLO('./models/yolov8n_ncnn_model')
```

Run inference over an image (let's use again bus.jpg):

```
img = './images/bus.jpg'
result = model.predict(img, save=True, imgsz=640, conf=0.5, iou=0.3)
```

```
marcelo_roval — miroval@raspi-zero: ~/Documents/YOLO — ssh miroval@192.168.4.210 — 105x16
(yolo) miroval@raspi-zero:~/Documents/YOLO$ python
Python 3.11.2 (main, Apr 28 2025, 14:11:48) [GCC 12.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from ultralytics import YOLO
>>> model = YOLO('./models/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 = './images/bus.jpg'
>>> result = model.predict(img, save=True, imgsz=640, conf=0.5, iou=0.3)
Loading ./models/yolov8n_ncnn_model for NCNN inference...

WARNING ━ NMS time limit 2.050s exceeded
image 1/1 /home/miroval/Documents/YOLO/images/bus.jpg: 640x640 3 persons, 1 bus, 1477.9ms
Speed: 428.0ms preprocess, 1477.9ms inference, 6843.7ms postprocess per image at shape (1, 3, 640, 640)
Results saved to runs/detect/predict18
>>> █
```

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 with a shape of (4, 6). Each line is one of the objects detected, being the first four columns, the bounding boxes coordinates, the 5th, the confidence, and the 6th, the class (in this case, 0: person and 5: bus):

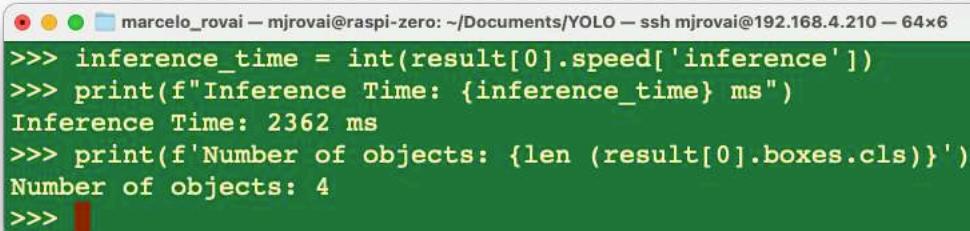
```
marcelo_roval — miroval@raspi-zero: ~/Documents/YOLO — ssh miroval@192.168.4.210 — 85x6
>>> result[0].boxes.data
tensor([[6.7045e+02, 3.8056e+02, 8.0992e+02, 8.7965e+02, 8.9021e-01, 0.0000e+00],
       [2.2168e+02, 4.0742e+02, 3.4378e+02, 8.5619e+02, 8.8328e-01, 0.0000e+00],
       [5.0682e+01, 3.9730e+02,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

```
from ultralytics import YOLO

# Load the YOLOv8 model
model = YOLO('./models/yolov8n_ncnn_model')

# Run inference
img = './images/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)}')

```

```

marcelo_revai — mjrovai@raspi-zero: ~/Documents/YOLO — ssh mjrovai@192.168.4.210 — 70x19
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.

Inference Arguments

`model.predict()` accepts multiple arguments that can be passed at inference time to override defaults:

Inference arguments:

Argument	Type	Default	Description
source	str	'ultralytics/assets'	Specifies the data source for inference. Can be an image path, video file, directory, URL, or device ID for live feeds. Supports a wide range of formats and sources, enabling flexible application across different types of input .
conf	float	0.25	Sets the minimum confidence threshold for detections. Objects detected with confidence below this threshold will be disregarded. Adjusting this value can help reduce false positives.
iou	float	0.7	Intersection Over Union (IoU) threshold for Non-Maximum Suppression (NMS). Lower values result in fewer detections by eliminating overlapping boxes, useful for reducing duplicates.
imgsz	int or tuple	640	Defines the image size for inference. Can be a single integer 640 for square resizing or a (height, width) tuple. Proper sizing can improve detection accuracy and processing speed.
rect	bool	True	If enabled, minimally pads the shorter side of the image until it's divisible by stride to improve inference speed. If disabled, pads the image to a square during inference.
half	bool	False	Enables half-precision (FP16) inference, which can speed up model inference on supported GPUs with minimal impact on accuracy.
device	str	None	Specifies the device for inference (e.g., <code>cpu</code> , <code>cuda:0</code> or <code>0</code>). Allows users to select between CPU, a specific GPU, or other compute devices for model execution.

Argument	Type	Default	Description
<code>batch</code>	<code>int</code>	1	Specifies the batch size for inference (only works when the source is a directory , video file or .txt file). A larger batch size can provide higher throughput, shortening the total amount of time required for inference.
<code>max_det</code>	<code>int</code>	300	Maximum number of detections allowed per image. Limits the total number of objects the model can detect in a single inference, preventing excessive outputs in dense scenes.
<code>vid_stride</code>	<code>int</code>	1	Frame stride for video inputs. Allows skipping frames in videos to speed up processing at the cost of temporal resolution. A value of 1 processes every frame, higher values skip frames.
<code>stream_buffer</code>	<code>bool</code>	<code>False</code>	Determines whether to queue incoming frames for video streams. If <code>False</code> , old frames get dropped to accommodate new frames (optimized for real-time applications). If <code>True</code> , queues new frames in a buffer, ensuring no frames get skipped, but will cause latency if inference FPS is lower than stream FPS.
<code>visualize</code>	<code>bool</code>	<code>False</code>	Activates visualization of model features during inference, providing insights into what the model is “seeing”. Useful for debugging and model interpretation.
<code>augment</code>	<code>bool</code>	<code>False</code>	Enables test-time augmentation (TTA) for predictions, potentially improving detection robustness at the cost of inference speed.
<code>agnostic_nms</code>	<code>bool</code>	<code>False</code>	Enables class-agnostic Non-Maximum Suppression (NMS), which merges overlapping boxes of different classes. Useful in multi-class detection scenarios where class overlap is common.
<code>classes</code>	<code>list[int]</code>	<code>None</code>	Filters predictions to a set of class IDs. Only detections belonging to the specified classes will be returned. Useful for focusing on relevant objects in multi-class detection tasks.

Argument	Type	Default	Description
<code>retina_masks</code>	<code>bool</code>	<code>False</code>	Returns high-resolution segmentation masks. The returned masks (<code>masks.data</code>) will match the original image size if enabled. If disabled, they have the image size used during inference.
<code>embed</code>	<code>list[int]</code>	<code>None</code>	Specifies the layers from which to extract feature vectors or <code>embeddings</code> . Useful for downstream tasks like clustering or similarity search.
<code>project</code>	<code>str</code>	<code>None</code>	Name of the project directory where prediction outputs are saved if <code>save</code> is enabled.
<code>name</code>	<code>str</code>	<code>None</code>	Name of the prediction run. Used for creating a subdirectory within the project folder, where prediction outputs are stored if <code>save</code> is enabled.
<code>stream</code>	<code>bool</code>	<code>False</code>	Enables memory-efficient processing for long videos or numerous images by returning a generator of <code>Results</code> objects instead of loading all frames into memory at once.
<code>verbose</code>	<code>bool</code>	<code>True</code>	Controls whether to display detailed inference logs in the terminal, providing real-time feedback on the prediction process.

Visualization arguments:

Argument	Type	Default	Description
<code>show</code>	<code>bool</code>	<code>False</code>	If <code>True</code> , displays the annotated images or videos in a window. Useful for immediate visual feedback during development or testing.
<code>save</code>	<code>bool</code>	<code>False</code> or <code>True</code>	Enables saving of the annotated images or videos to file. Useful for documentation, further analysis, or sharing results. Defaults to <code>True</code> when using CLI & <code>False</code> when used in Python.
<code>save_frames</code>	<code>bool</code>	<code>False</code>	When processing videos, saves individual frames as images. Useful for extracting specific frames or for detailed frame-by-frame analysis.

Argument	Type	Default	Description
save_txt	bool	False	Saves detection results in a text file, following the format [class] [x_center] [y_center] [width] [height] [confidence]. Useful for integration with other analysis tools.
save_conf	bool	False	Includes confidence scores in the saved text files. Enhances the detail available for post-processing and analysis.
save_crop	bool	False	Saves cropped images of detections. Useful for dataset augmentation, analysis, or creating focused datasets for specific objects.
show_labels	bool	True	Displays labels for each detection in the visual output. Provides immediate understanding of detected objects.
show_conf	bool	True	Displays the confidence score for each detection alongside the label. Gives insight into the model's certainty for each detection.
show_boxes	bool	True	Draws bounding boxes around detected objects. Essential for visual identification and location of objects in images or video frames.
line_width	None or int	None	Specifies the line width of bounding boxes. If None, the line width is automatically adjusted based on the image size. Provides visual customization for clarity.

Exploring other Computer Vision Applications

Let's set up Jupyter Notebook optimized for headless Raspberry Pi camera work and development:

```
pip install jupyter jupyterlab notebook
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 and its **Token** to open the notebook. Copy and paste it into the Browser.

Environment Setup and Dependencies

```
import time
import numpy as np
from PIL import Image
from ultralytics import YOLO
import matplotlib.pyplot as plt
```

Here we have all the necessary libraries, which we installed automatically when we installed Ultralytics.

- **Time:** Performance measurement and benchmarking
- **NumPy:** Numerical computations and array operations
- **PIL (Python Imaging Library):** Image loading and manipulation
- **Ultralytics YOLO:** Core YOLO functionality
- **Matplotlib:** Visualization and plotting results

Model Configuration and Loading

```
model_path= "./models/yolo11n.pt"
task = "detect"
verbose = False

model = YOLO(model_path, task, verbose)
```

- **Model Selection:** YOLOv11n (nano) is chosen for its balance of speed and accuracy
- **Task Specification:** We will select `detect`, which in fact is the default for the model. But remember that YOLO supports multiple computer vision tasks, which will be explored later.
- **Verbose Control:** output model information during model initialization

Performance Characteristics

Let's open the previous bus image using PIL

```
source = Image.open("./images/bus.jpg")
```

And run an inference in the source:

```
results = model.predict(source, save=False, imgsz=640, conf=0.5, iou=0.3)
```

From the inference results info, we can see that the first time an inference is run, the latency is greater.

```
# First inference  
0: 640x480 4 persons, 1 bus, 7528.3ms  
  
# Second inference  
0: 640x480 4 persons, 1 bus, 2822.1ms
```

The dramatic difference between the first inference (7.5s) and subsequent inferences (2.8s) illustrates:

- **Model Loading Overhead:** Initial inference includes model loading time
- **Optimization Effects:** Subsequent inferences benefit from cached optimizations

Results Object Structure

Let's explore the YOLO's output structure:

```
result = results[0]  
# - boxes, keypoints, masks, names  
# - orig_img, orig_shape, path  
# - speed metrics
```

Bounding Box Analysis

```
result.boxes.cls # Class IDs: tensor([5., 0., 0., 0., 0.])  
result.boxes.conf # Confidence scores  
result.boxes.xyxy # Bounding box coordinates
```

- **Coordinate Systems:** On `Result.boxes`, we can get different bounding box formats (xyxy, xywh, normalized):
 - xywh: Tensor with bounding box coordinates in center_x, center_y, width, height format, in pixels.
 - xywhn: Normalized center_x, center_y, width, height, scaled to the image dimensions, values in .
 - xyxy: Tensor of boxes as x1, y1, x2, y2 in pixels, representing the top-left and bottom-right corners.
 - xyxyn: Normalized x1, y1, x2, y2, scaled by image width and height, values in .

8. Visualization and Customization

The Ultralytics `plot()` can be customized to show as the detection result, for example, only the bounding boxes:

```
im_bgr = result.plot(boxes=True, labels=False, conf=False)

img = Image.fromarray(im_bgr[..., ::-1])

plt.figure(figsize=(6, 6))
plt.imshow(img)
#plt.axis('off') # This turns off the axis numbers
plt.title("YOLO Result")
plt.show()
```

YOLO Result



Customization Options:

The plot() method in Ultralytics YOLO Results object accepts several arguments to control what is visualized on the image, including boxes, masks, keypoints, confidences, labels, and more. Common Arguments for plot()

- boxes (bool): Show/hide bounding boxes. Default is True.
- conf (bool): Show/hide confidence scores. Default is True.
- labels (bool): Show/hide class labels. Default is True.
- masks (bool): Show/hide segmentation masks (when available, e.g. in segment tasks).
- kpt_line (bool): Draw lines connecting pose keypoints (skeleton diagram). Default is True in pose tasks.
- line_width (int): Set annotation line thickness.
- font_size (int): Set font size for text annotations.
- show (bool): If True, immediately display the image (interactive environments).

Exploring Other Computer Vision Tasks

Image Classification

As explored in previous chapters, the output of an image classifier is a single class label and a confidence score. Image classification is useful when we need to know only what class an image belongs to and don't need to know where objects of that class are located or what their exact shape is.

```
model_path= "./models/yolo11n-cls.pt"
task = "classification"

model = YOLO(model_path, task, verbose)
```

Note that a specific variation of the model, for image classification, will be downloaded. Now, let's do an inference, using the same bus image:

```
results = model.predict(source, save=False)
result = results[0]
```

The model inference will return:

```
0: 224x224 minibus 0.57, police_van 0.34, trolleybus 0.04, recreational_vehicle 0.01, stre
Speed: 5233.9ms preprocess, 3355.1ms inference, 28.2ms postprocess per image at shape (1,
```

We can check the top5 inference results using Python:

```
classes = result.probs.top5
classes
```

And we will get:

```
[654, 734, 874, 757, 829]
```

What is related to:

```
for id in classes:
    print(result.names[id])

minibus
police_van
trolleybus
recreational_vehicle
streetcar
```

And the top 5 probabilities:

```
probs = result.probs.top5conf.tolist()
probs

[0.5710113048553467,
 0.33745330572128296,
 0.04209813103079796,
 0.014150412753224373,
 0.005880324635654688]
```

The Top 1 result can be extract directly from the result:

```
print(result.names[result.probs.top1],
      round(result.probs.top1conf.tolist(), 2))

minibus 0.57
```

Instance Segmentation

```
model_path= "./models/yolo11n-seg.pt"
task = "segment"

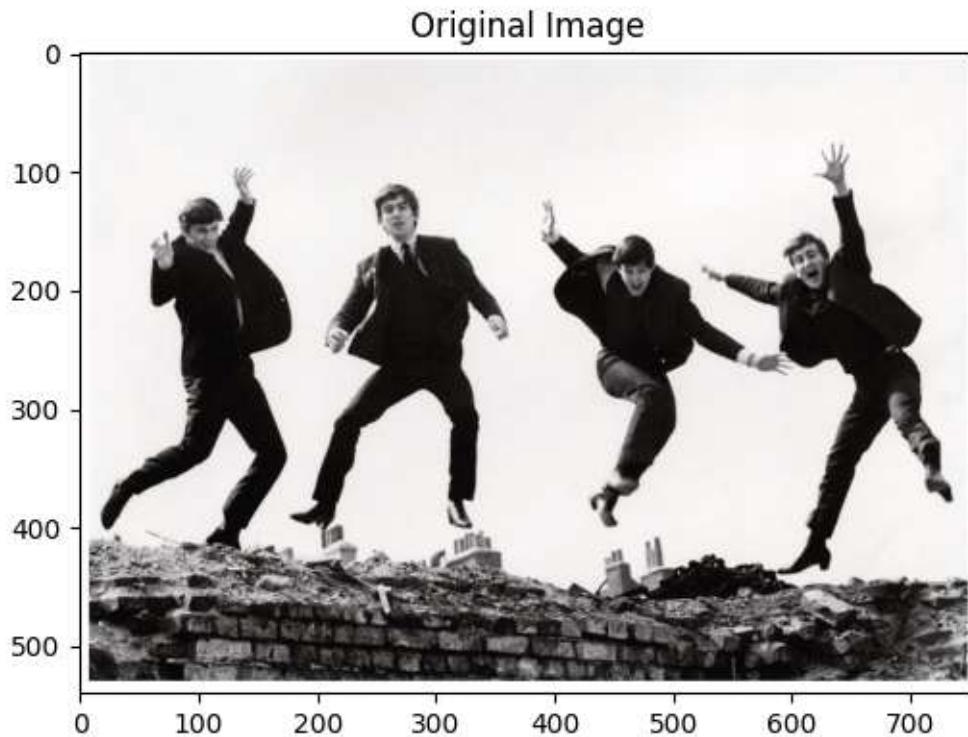
model = YOLO(model_path, task, verbose)
```

Note that a specific variation of the model, for instance segmentation, will be downloaded.
Now, let's use another image for testing:

```
source = Image.open("./images/beatles.jpg")
```

Display the image

```
plt.figure(figsize=(6, 6))
plt.imshow(source)
#plt.axis('off') # This turns off the axis numbers
plt.title("Original Image")
plt.show()
```



And run the inference:

```
results = model.predict(source, save=False)
result = results[0]
```

Display the result:

```
im_bgr = result.plot(boxes=False, conf=False, masks=True)
img = Image.fromarray(im_bgr[..., ::-1])
plt.figure(figsize=(6, 6))
plt.imshow(img)
plt.axis('off') # This turns off the axis numbers
plt.title("YOLO Segmentation Result")
plt.show()
```

YOLO Segmentation Result



Pose Estimation

Download the model:

```
model_path= "./models/yolo11n-pose.pt"
task = "pose"

model = YOLO(model_path, task, verbose)
```

Running the Inference on the beatles image:

```
source = Image.open("./images/beatles.jpg")
results = model.predict(source, save=False)
result = results[0]
```

Showing the human pose keypoint detection and skeleton visualization.

```
im_bgr = result.plot(boxes=False, conf=False, kpt_line=True)
img = Image.fromarray(im_bgr[..., ::-1])
plt.figure(figsize=(6, 6))
plt.imshow(img)
plt.axis('off') # This turns off the axis numbers
plt.title("YOLO Pose Estimation Result")
plt.show()
```

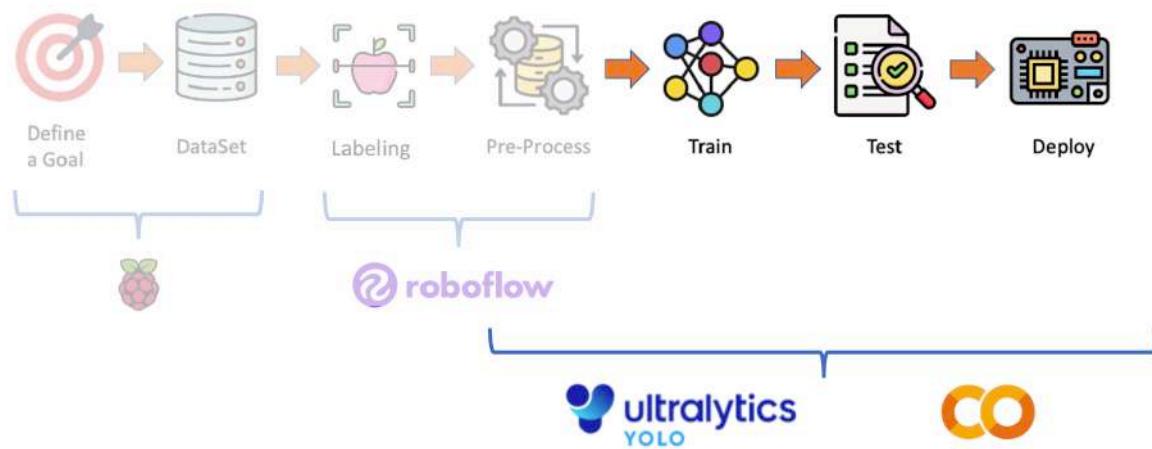
YOLO Pose Estimation Result



Training YOLO on a Customized Dataset

Object Detection Project

We will now develop a customized object detection project from the data collected and labelled with Roboflow. The training and deployment will be done in Python using a CoLab and Ultralytics functions.



We will use with YOLO, the same dataset previously used to train the SSD-MobileNet V2 and FOMO models.

As a reminder, we are assuming we are in an industrial facility that must sort and count **wheels** and special **boxes**.

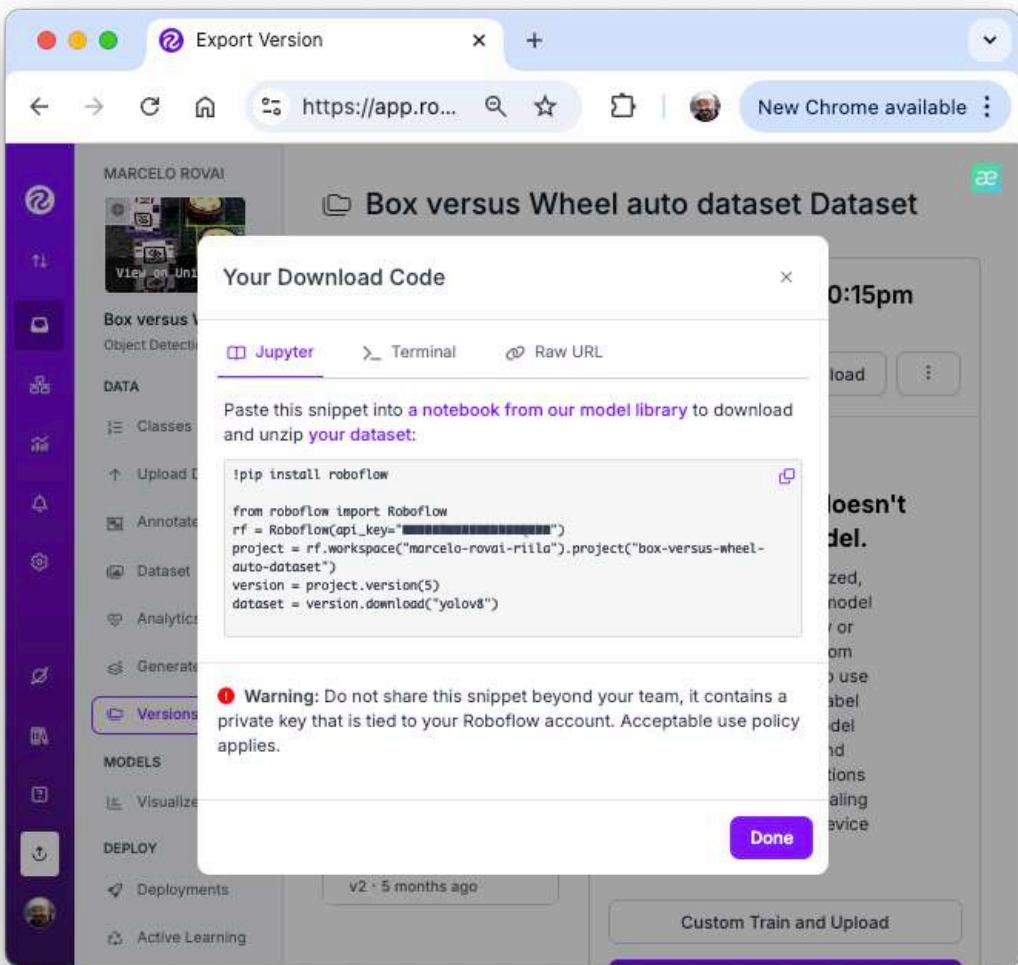


Each image can have three classes:

- Background (no objects)
- Box
- Wheel

The 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 choose one model (let's say YOLOv8) and adapt one of the publicly available examples from Ultralytics, then run it on Google Colab. Below, you can find my adaptation:

- 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.

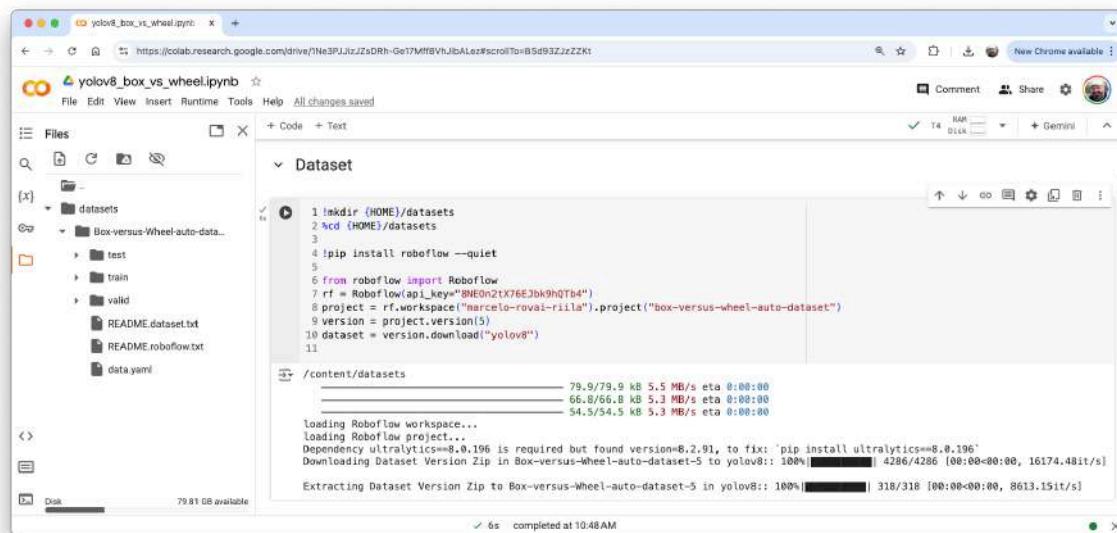
```

[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/`:



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}

```

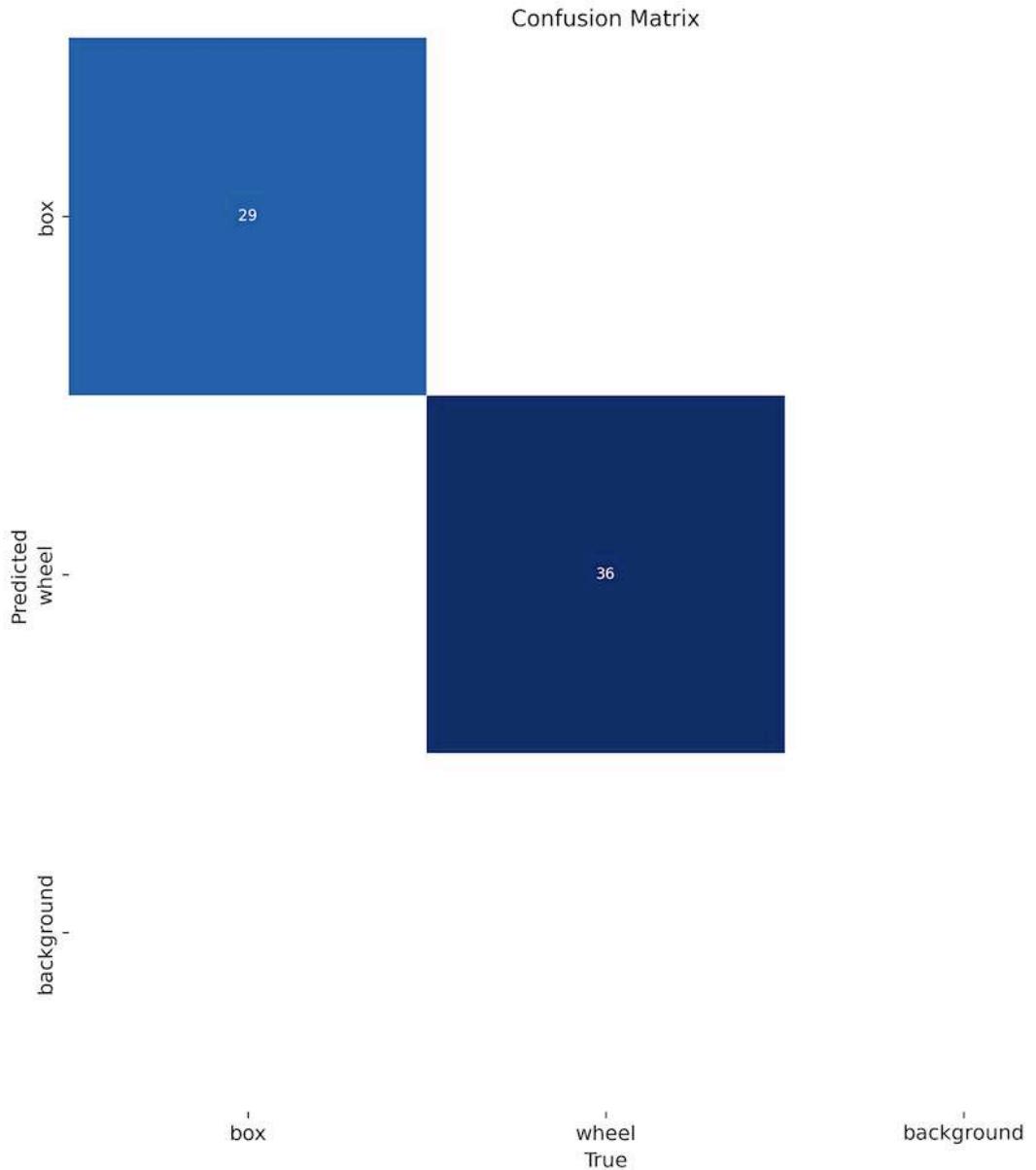
25 epochs completed in 0.026 hours.
Optimizer stripped from runs/detect/train/weights/last.pt, 6.2MB
Optimizer stripped from runs/detect/train/weights/best.pt, 6.2MB

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`.

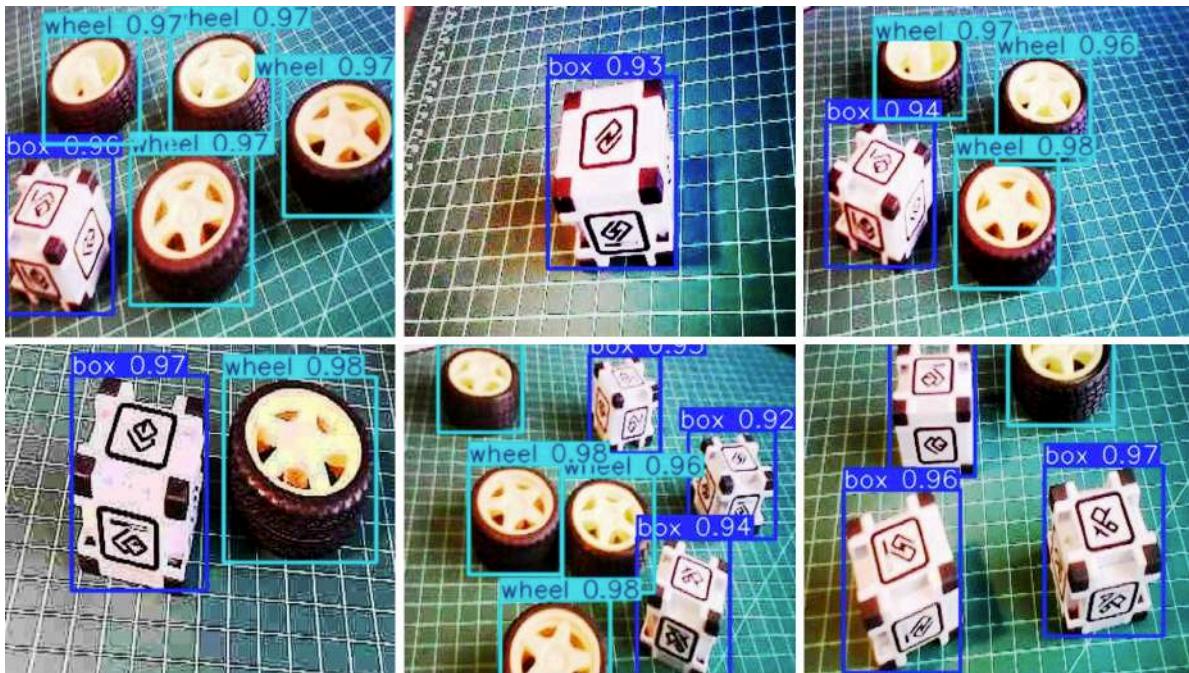
```
!yolo task=detect mode=val model={HOME}/runs/detect/train/weights/best.pt data={dataset.lo
```

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 so
```

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
```

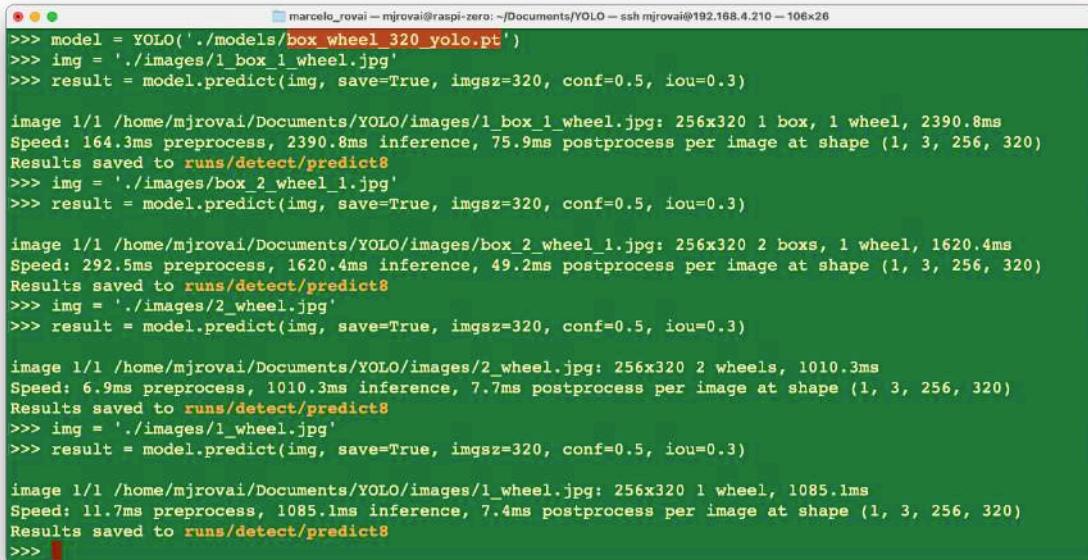
As before, we will import the YOLO library and define our converted model to detect bees:

```
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). We can reduce this latency and convert the model to TFLite or NCNN.

The model trained with YOLOv11 has a latency of around 800 ms, similar to the result of v8 with ncnn.

Conclusion

This chapter has explored the YOLO model and the implementation of a custom object detector on a 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, 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 with Ultralytics and deploying them on a Raspberry Pi.
3. **Optimization Techniques:** To improve inference speed on edge devices, we explored various optimization methods, such as format conversion (e.g., to NCNN).
4. **Performance Considerations:** Throughout the lab, we discussed the balance between model accuracy and inference speed, a critical consideration for edge AI applications.

As discussed before, the ability to perform object detection on edge devices opens up numerous possibilities across various domains, including precision agriculture, industrial automation, quality control, 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”)
- [YOLOv8 Box versus Wheel Dataset Training on CoLab](#)
- [YOLOv11 Box versus Wheel Dataset Training on CoLab](#)
- [Model Predictions with Ultralytics YOLO](#)
- [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 5: 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.

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 Raspberry Pi. 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).

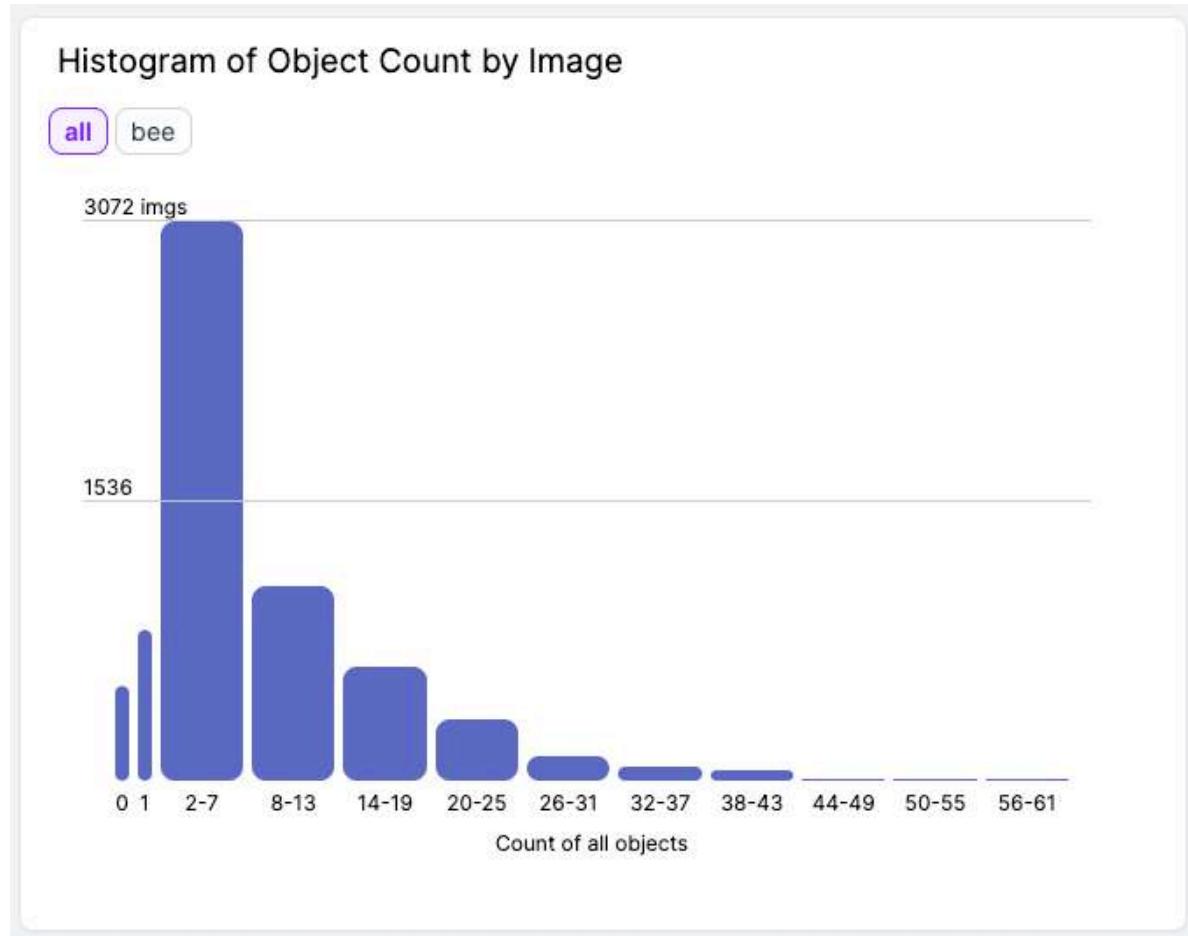


Dataset

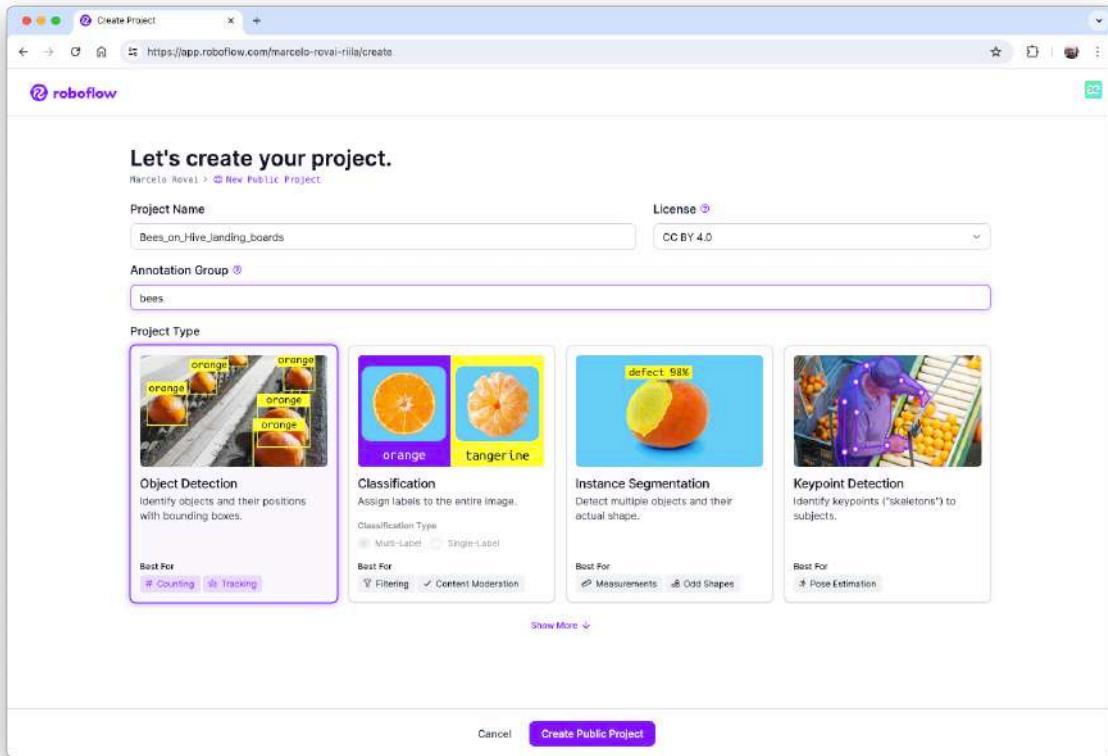
The dataset collection is the most critical phase of the project and should take several weeks or months. For this tutorial, we will use a public dataset: “Sledevic, Tomyslav (2023), “[Labeled dataset for bee detection and direction estimation on beehive landing boards,” Mendeley Data, V5, doi: 10.17632/8gb9r2yhfc.5”

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:

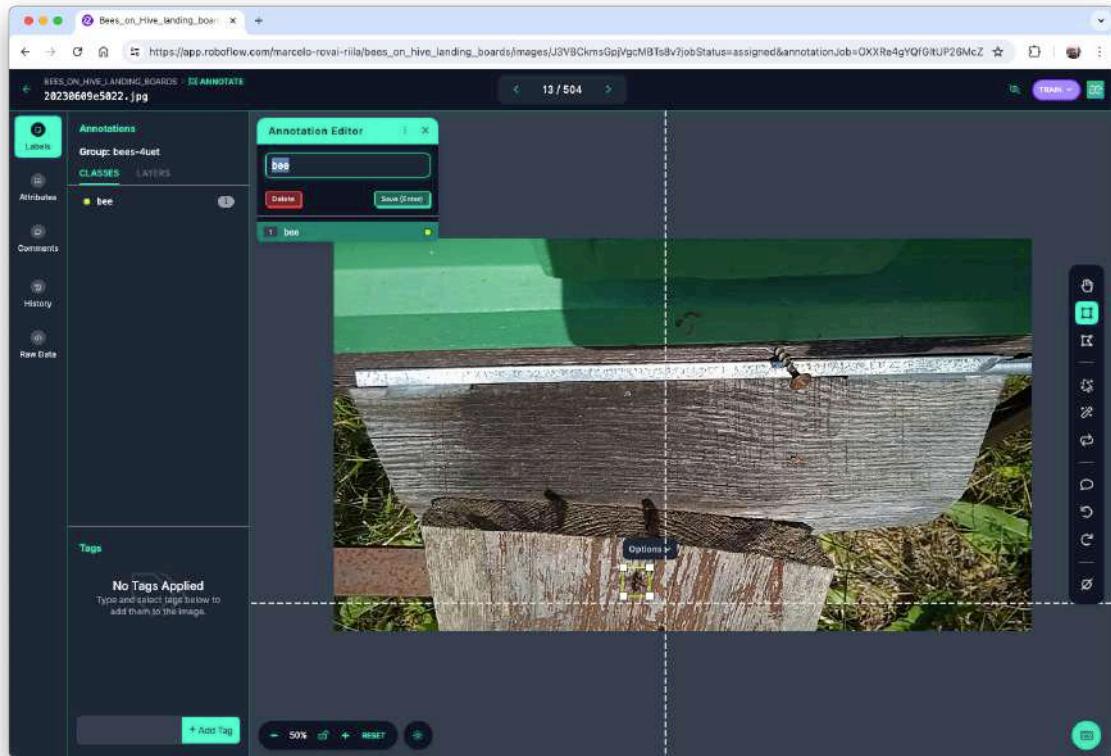


We downloaded the dataset (images and annotations) and uploaded it to [Roboflow](#). There, you should create a free account and start a new project, for example, (“Bees_on_Hive_landing_boards”):

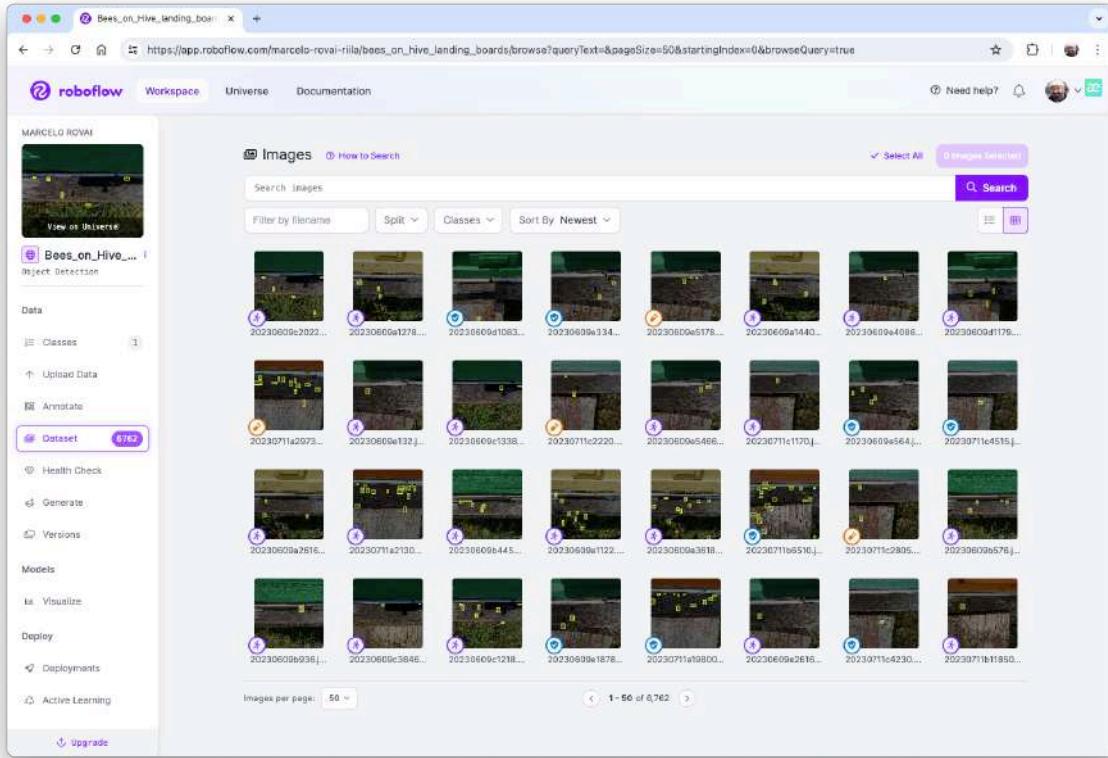


We will not enter details about the Roboflow process once many tutorials are available.

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.



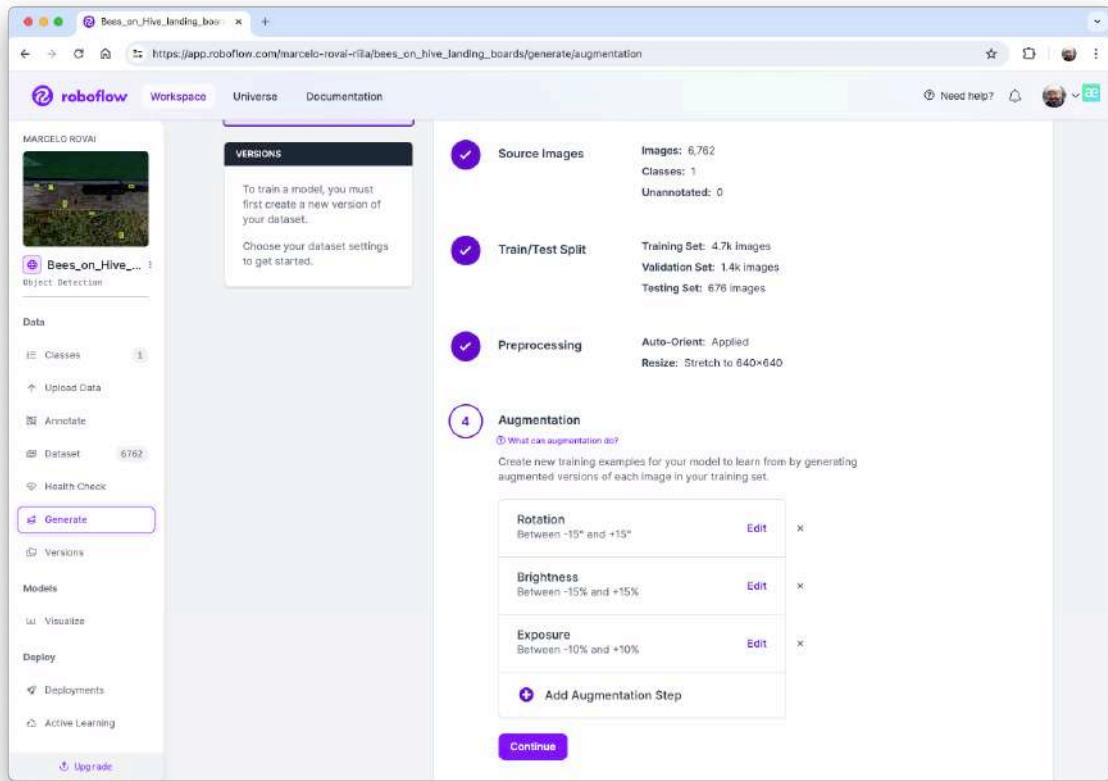
Once all images are annotated, you should split them into training, validation, and testing.



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-rovai-riilo").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](#).

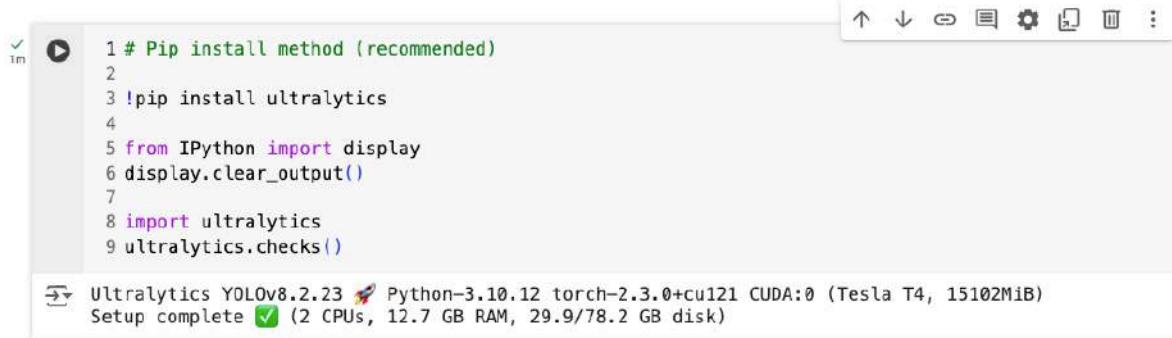
Training YOLOv8 on a Customized Dataset

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.



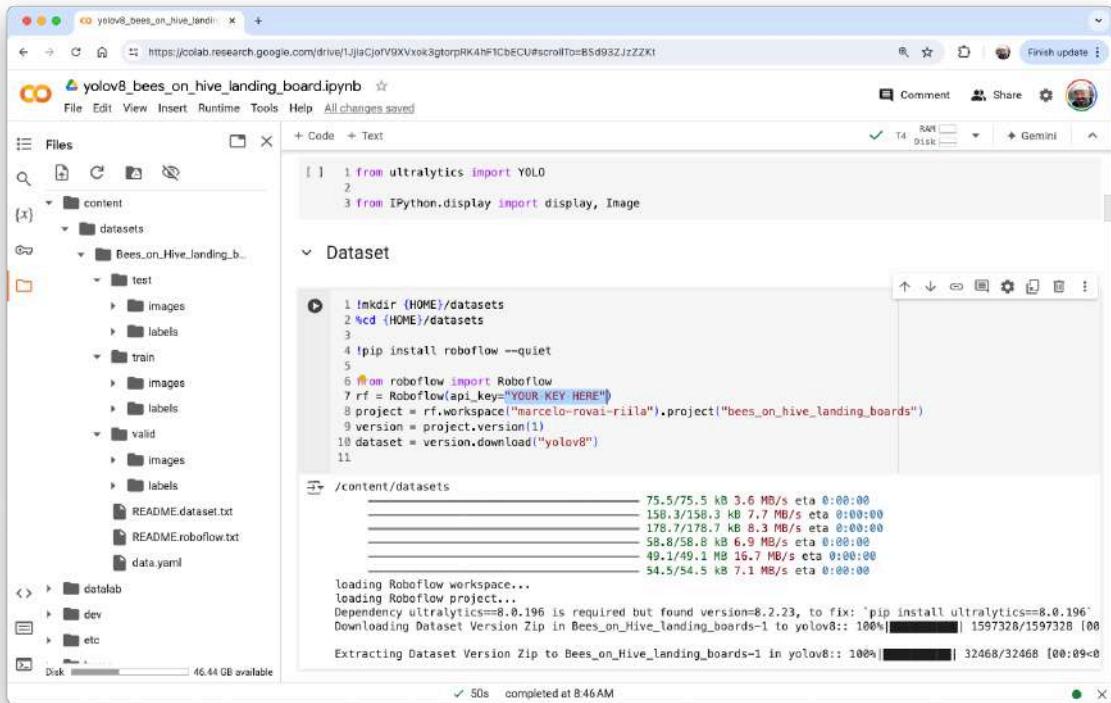
```

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()

```

→ Ultralytics YOLOv8.2.23 🚀 Python-3.10.12 torch-2.3.0+cu121 CUDA:0 (Tesla T4, 15102MiB)
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

```

Dataset

```

1 !mkdir -p {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_boards")
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
Extracting Dataset Version Zip to Beans_on_Hive_landing_boards-1 to yolov8: 100% | 1597328/1597328 [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}

```

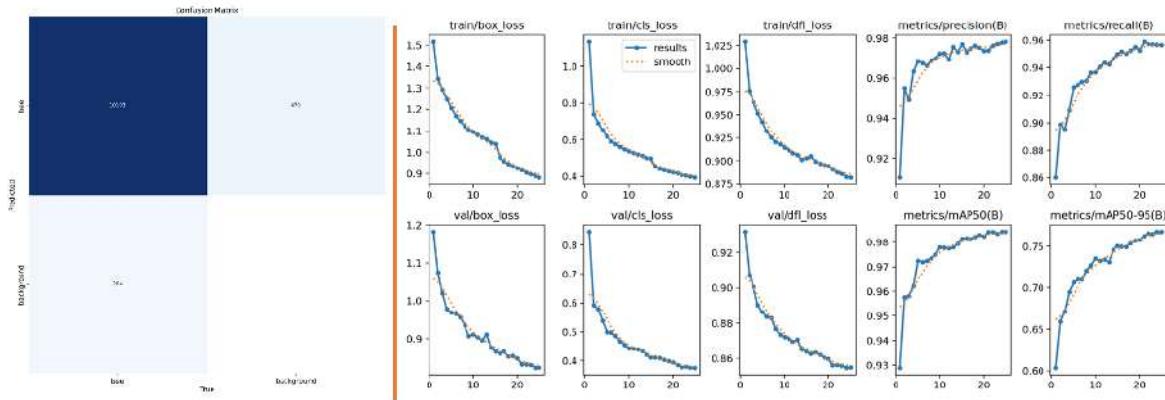
25 epochs completed in 2.679 hours.
Optimizer stripped from runs/detect/train3/weights/last.pt, 6.2MB
Optimizer stripped from runs/detect/train3/weights/best.pt, 6.2MB

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)
all	1353	10477	0.978	0.957	0.984	0.768

: 100% 43/43 [00:33<00:00, 1.27it/s]
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.

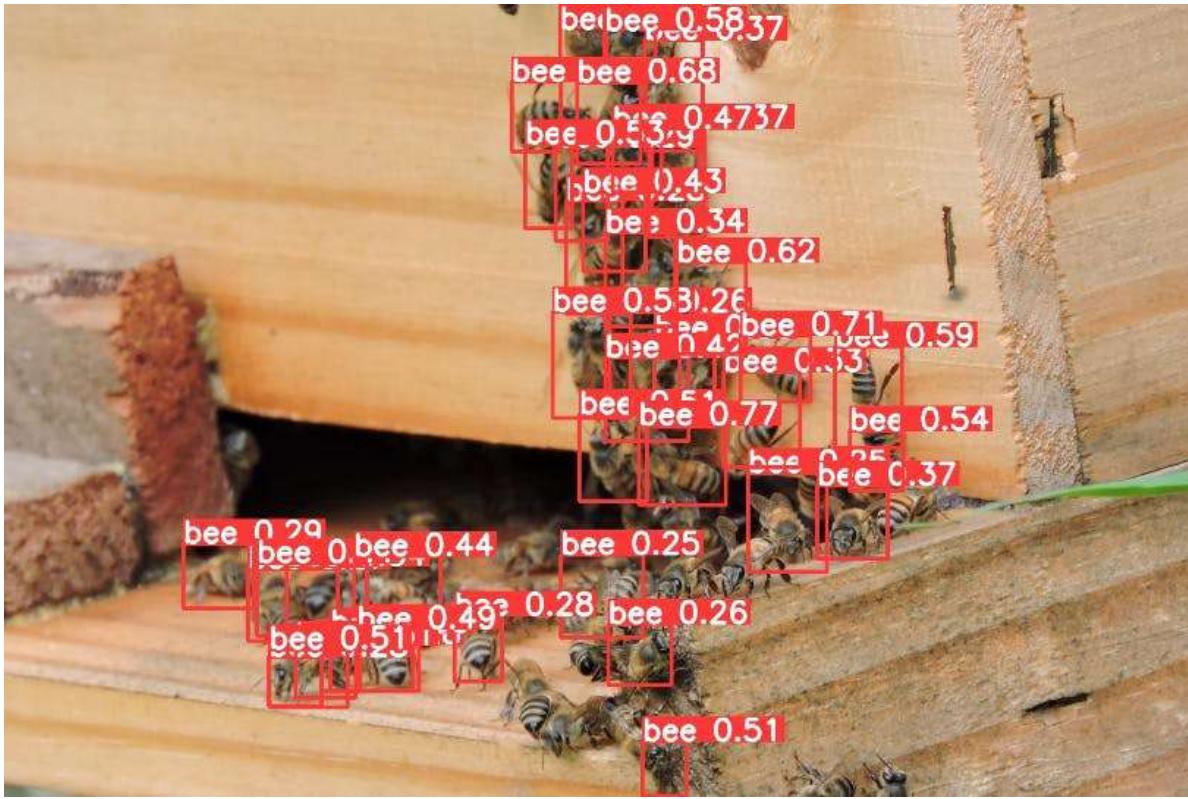
8. Now, we should perform inference on the images left aside for testing

```
!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
```

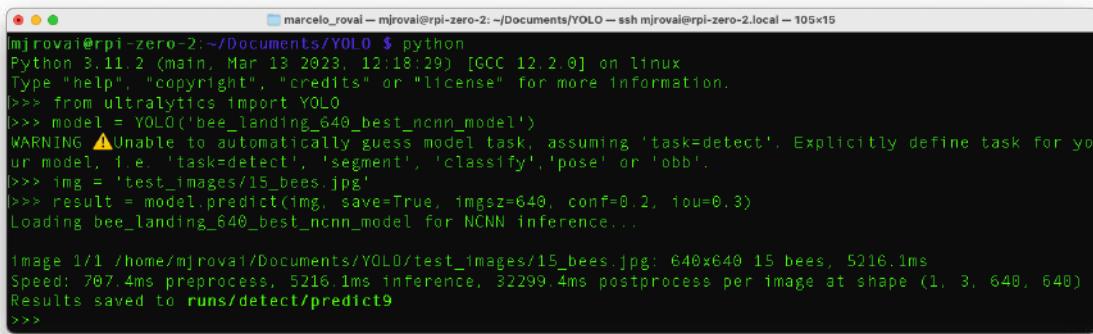
As before, we will import the YOLO library and define our converted model to detect bees:

```
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)
```

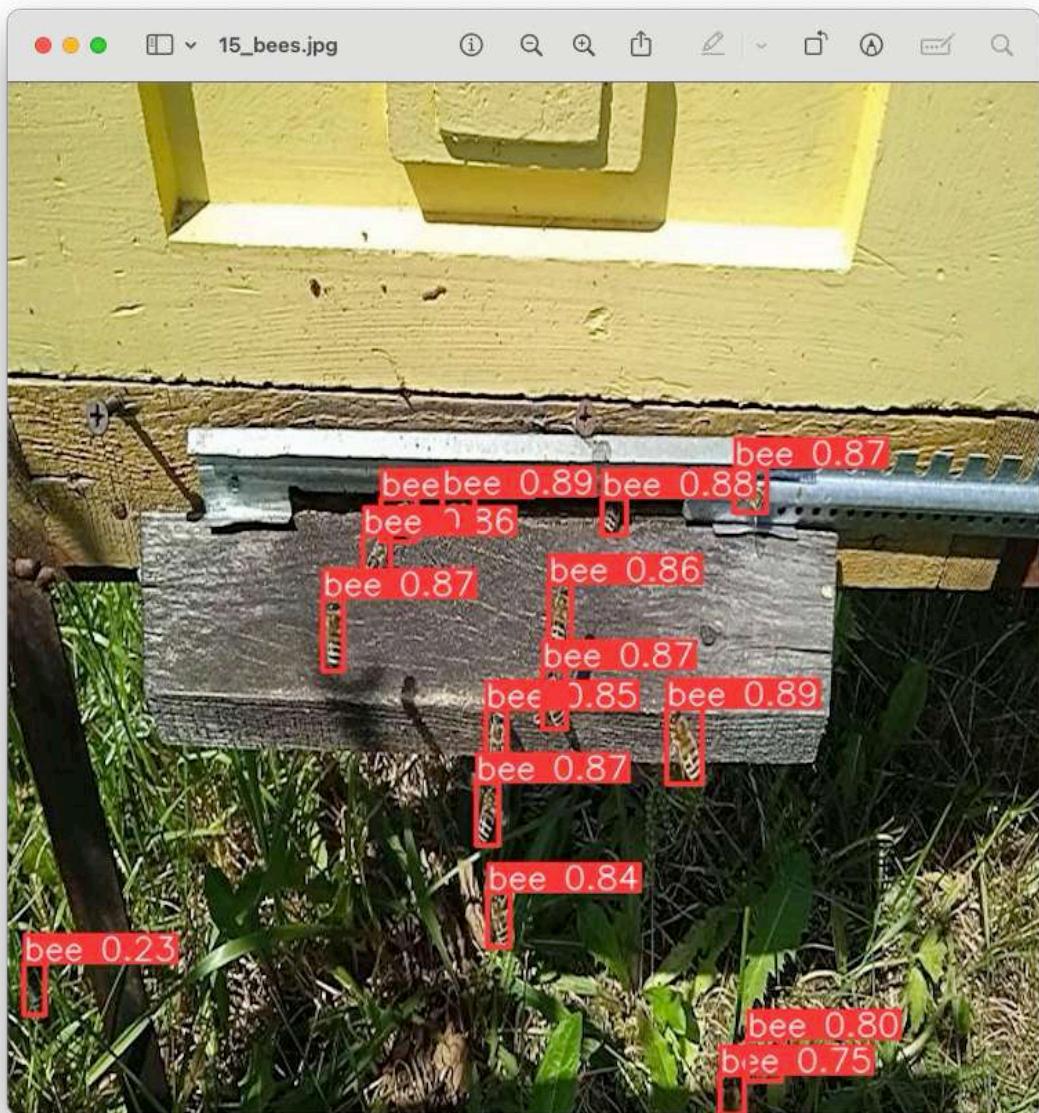
The inference result is saved on the variable `result`, and the processed image on `runs/detect/predict9`



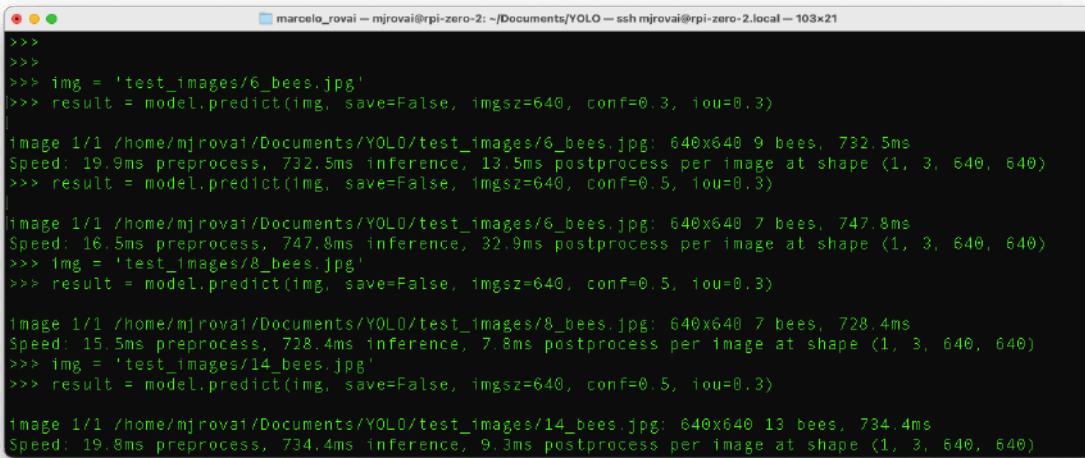
A screenshot of a terminal window titled "marcelo_royal - mjroval@rpi-zero-2: ~/Documents/YOLO". The window shows Python code running on a RPi Zero 2. The code imports YOLO, loads a model, and performs inference on an image. It includes a warning about task detection and prints results to the terminal.

```
marcelo_royal@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_bees.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_bees.jpg: 640x640 9 bees, 732.5ms
Speed: 19.8ms 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_bees.jpg: 640x640 7 bees, 747.8ms
Speed: 16.5ms preprocess, 747.8ms inference, 32.9ms postprocess per image at shape (1, 3, 640, 640)
>>> img = 'test_images/8_bees.jpg'
>>> result = model.predict(img, save=False, imgs=640, conf=0.5, iou=0.3)

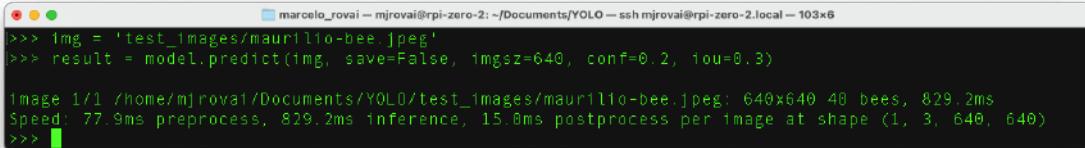
image 1/1 /home/mjrovai/Documents/YOLO/test_images/8_bees.jpg: 640x640 7 bees, 728.4ms
Speed: 15.5ms preprocess, 728.4ms inference, 7.8ms postprocess per image at shape (1, 3, 640, 640)
>>> img = 'test_images/14_bees.jpg'
>>> result = model.predict(img, save=False, imgs=640, conf=0.5, iou=0.3)

image 1/1 /home/mjrovai/Documents/YOLO/test_images/14_bees.jpg: 640x640 13 bees, 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=F
num_bees = 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, num_bees)
               )
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:

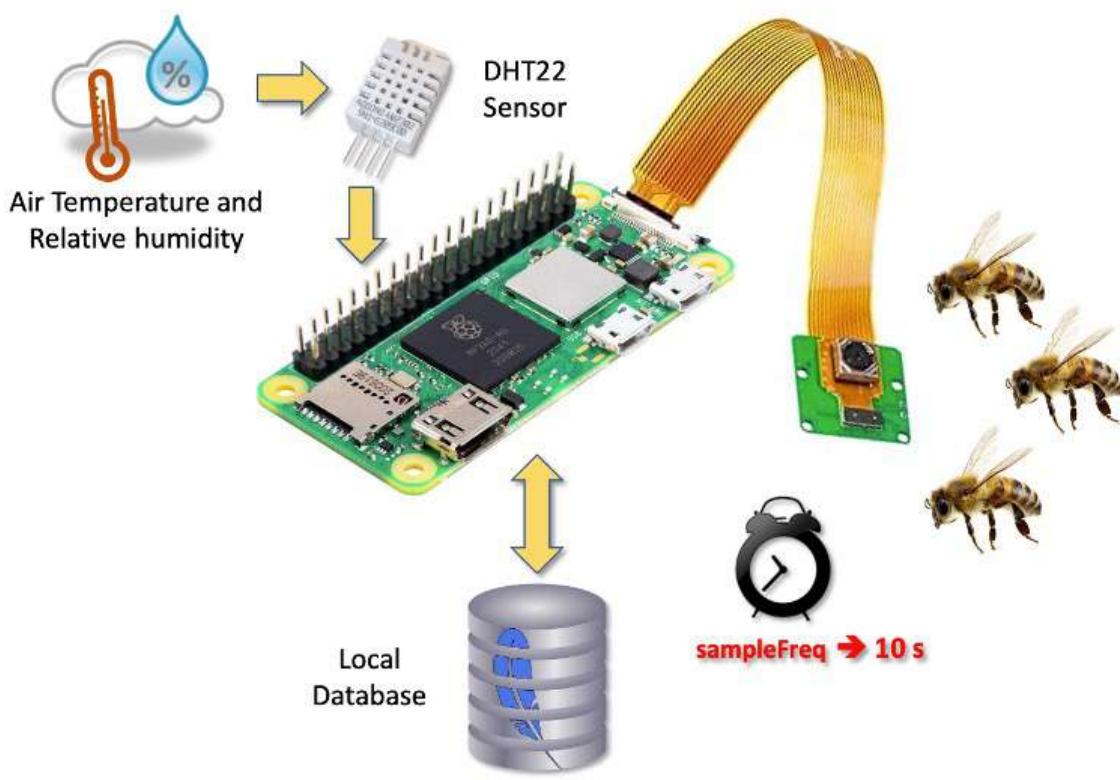
```
#!/usr/bin/env python3
import sqlite3

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()

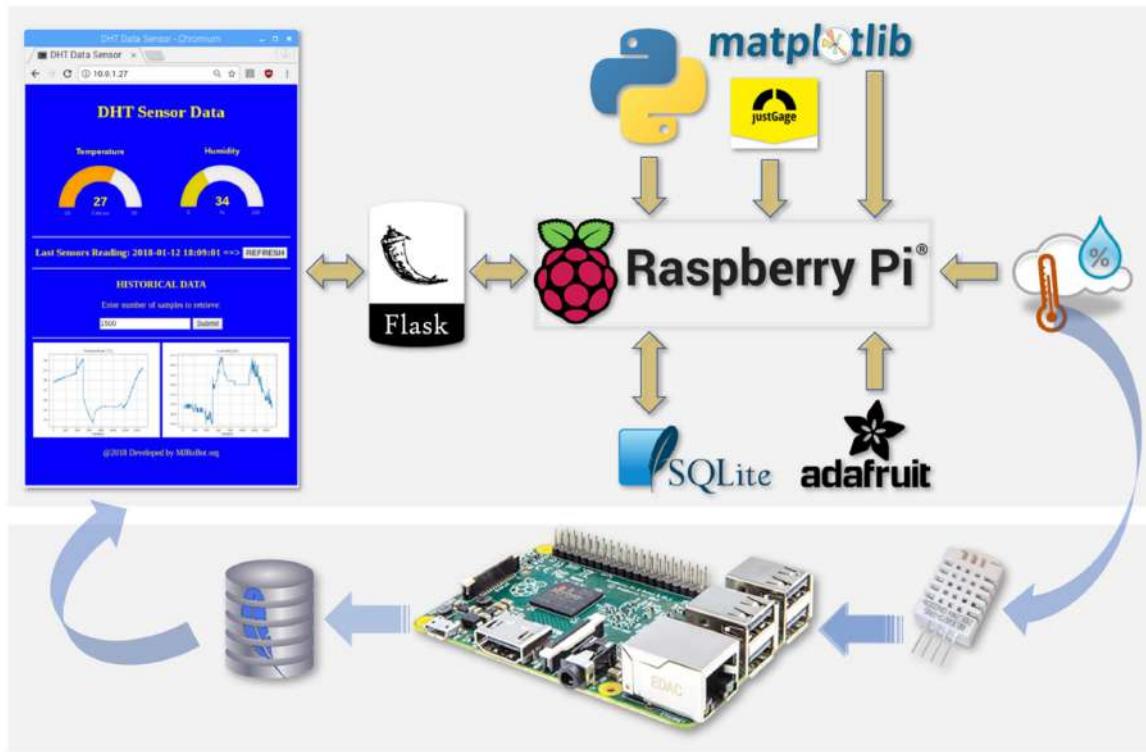
if __name__ == "__main__":
    main()
```

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](#).

#

Generative AI (Proactive)

Text Generation with RNNs

The Jules Verne Bot



Introduction

In this chapter, we will explore how to build a character-level text generation model using Recurrent Neural Networks (RNNs), specifically inspired by the works of Jules Verne.

The “Jules Verne Bot” project will help to show us the fundamental concepts of sequence modeling and text generation using deep learning techniques, as a preview of how the modern LLMs work.

Project Overview:

- **Goal:** Create an AI model that generates text in the style of Jules Verne
- **Architecture:** RNN with GRU (Gated Recurrent Unit) layers
- **Approach:** Character-level text prediction
- **Framework:** TensorFlow/Keras
- **Platform:** Google Colab with Tesla T4 GPU

What Are We Actually Building?

Imagine we could teach a computer to write like Jules Verne, the famous author of “Twenty Thousand Leagues Under the Sea” and “Around the World in Eighty Days.” That’s precisely what we’re doing with the Jules Verne Bot. This project creates an artificial intelligence system that learns the patterns, style, and vocabulary from Jules Verne’s novels, then generates new text that sounds like it could have come from his pen.

Think of it like this: if we read enough of someone’s writing, we start to recognize their style. We notice they use certain phrases, prefer specific sentence structures, or have favorite topics. Our neural network does something similar, but with mathematical precision. It analyzes millions of characters from Verne’s works and learns to predict what character should come next in any given sequence.

Neural Network Architectures Background

Before we dive into the technical details, let’s understand why we use neural networks for this task and why we chose the specific type we did.

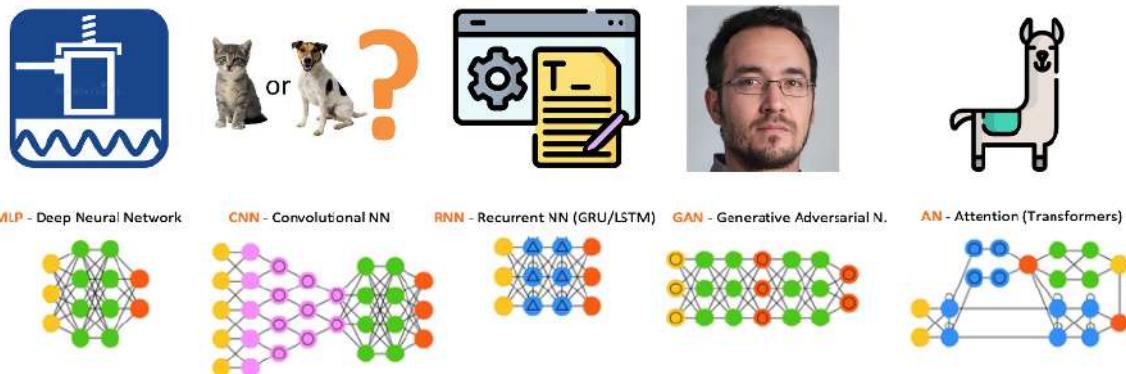
The Human Brain Analogy

When you read a sentence like “The submarine descended into the dark...” your brain automatically starts predicting what might come next. Maybe “depths” or “ocean” or “waters.” Your brain does this because it has learned patterns from all the text you’ve ever read. Neural networks work similarly, but they learn these patterns through mathematical calculations rather than biological processes.

Recurrent Neural Networks (RNN)

Before diving into our RNN implementation, let’s understand where RNNs fit in the neural network ecosystem:

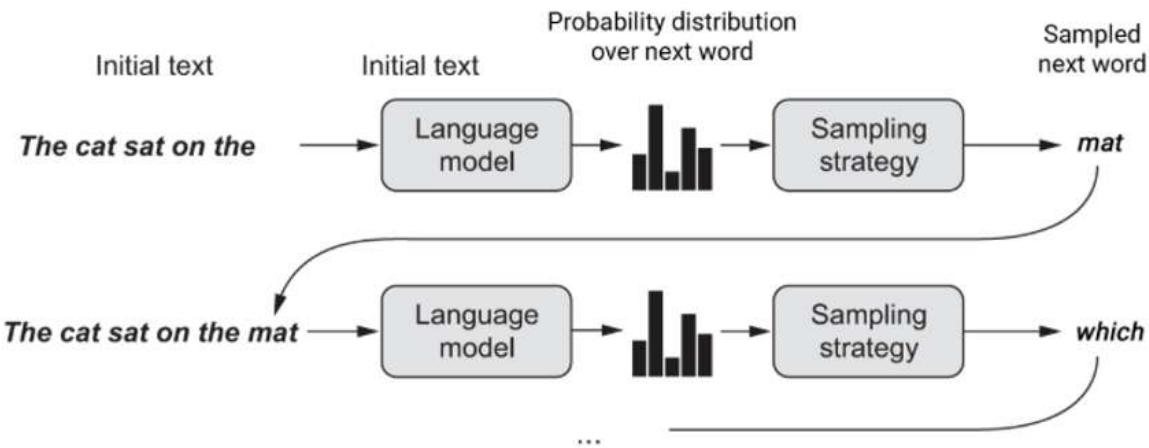
Key Neural Network Architectures:



- **MLP (Multi-Layer Perceptron)**: Basic feedforward networks for general tasks, for example, vibration analysis
- **CNN (Convolutional Neural Networks)**: Specialized for image processing as Image Classification tasks and spatial data
- **RNN (Recurrent Neural Networks)**: Designed for sequential data like text and time series
- **GAN (Generative Adversarial Networks)**: Two networks competing for realistic data generation, as images
- **Transformers (Attention Networks)**: Modern architecture using attention mechanisms, as in LLMs (Large Language Models, such as GPT)

We chose a Recurrent Neural Network (RNN) for this project because text has a crucial property: order matters tremendously. The sequence “The cat sat on the mat” means something completely different from “Mat the on sat cat the.” Regular neural networks process all inputs simultaneously, like looking at a photograph. But for text, we need a network that processes information sequentially, remembering what came before to understand what should go next.

In text generation, we aim to predict the most probable word to follow a sentence.



Think of reading a book. You don't just look at all the words on a page simultaneously. You read word by word, sentence by sentence, and your understanding builds as you progress. Each new word is interpreted in the context of everything you've read before in that chapter. RNNs work the same way.

The Memory Problem and GRU Solution

Early RNNs had a significant problem: they couldn't remember information for very long. Imagine trying to understand a story where you could only remember the last few words you read. You'd lose track of characters, plot points, and context very quickly.

This is where the Gated Recurrent Unit (GRU) comes in. Think of GRU as an improved memory system with two special abilities:

Reset Gate: This decides when to "forget" old information. If the story switches to a new scene or character, the reset gate helps the network forget irrelevant details from the previous context.

Update Gate: This decides how much new information to incorporate. When encountering important plot points or character names, the update gate helps the network remember these crucial details for longer.

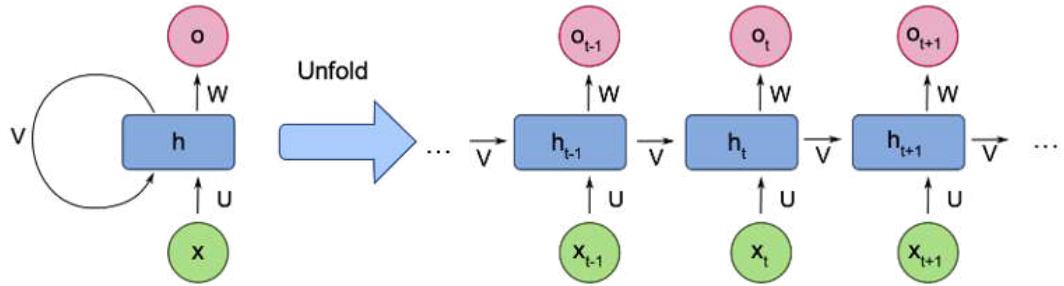
It's like having a smart note-taking system that automatically decides what's worth remembering and what can be forgotten.

Why RNNs for Text Generation?

Recurrent Neural Networks are designed explicitly for sequential data processing. Key characteristics:

- **Memory:** RNNs maintain an internal state (memory) to remember previous inputs
- **Sequential Processing:** Process data one element at a time, making them ideal for text
- **Variable Length Input:** Can handle sequences of different lengths
- **Parameter Sharing:** Same weights applied across different time steps

RNN Architecture Flow:



Input Sequence: $x(t-1) \rightarrow x(t) \rightarrow x(t+1) \rightarrow \dots$

Hidden State: $h(t-1) \rightarrow h(t) \rightarrow h(t+1) \rightarrow \dots$

Output: $o(t-1) \rightarrow o(t) \rightarrow o(t+1) \rightarrow \dots$

Dataset Preparation



Our model is trained on a curated collection of 10 classic Jules Verne novels, downloaded from public domain texts of the [Gutenberg Project](#):

1. “**A Journey to the Centre of the Earth**”
2. “**In Search of the Castaways**”
3. “**An Antarctic Mystery**”
4. “**In the year 2889**”
5. “**Around the World in Eighty Days**”
6. “**Michael Strogoff**”
7. “**Five Weeks in a Balloon**”
8. “**The Mysterious Island**”
9. “**From the Earth to the Moon**”
10. “**Twenty Thousand Leagues under the Sea**”

“A Journey to the Centre of the Earth” teaches the model about geological descriptions and underground adventures. “Twenty Thousand Leagues Under the Sea” provides vocabulary about marine life and submarine technology. “Around the World in Eighty Days” offers geographical references and travel descriptions. Each book contributes unique vocabulary and stylistic elements while maintaining Verne’s consistent voice.

The complete dataset contains **5,768,791 characters**, with **123 unique characters**. To put this in perspective, that’s roughly equivalent to **1,200 pages of double-spaced text**.

This gives our neural network plenty of material to learn from, ensuring it can capture both common patterns and unique expressions in Verne’s writing.

Data Preprocessing Steps

```
# Example preprocessing workflow
def preprocess_text(text):
    # Convert to lowercase for consistency
    text = text.lower()

    # Remove unwanted characters (optional)
    # Keep punctuation for realistic text generation

    return text

# Load and combine all books
all_text = ""
for book in book_list:
    with open(book, 'r') as f:
        all_text += preprocess_text(f.read())
```

Tokenization and Vocabulary

Character-Level Tokenization

Here’s where our approach differs from how humans typically think about text. While we naturally think in words and sentences, our model processes text character by character. This means it learns that certain letters frequently follow others, that spaces separate words, and that punctuation marks signal sentence boundaries.

Why choose character-level processing? Consider the word “extraordinary,” which appears frequently in Verne’s work. A word-level model would need to have seen this exact word during training to use it. But a character-level model can generate this word by learning that ‘e’ often starts words, ‘x’ can follow ‘e’, ‘t’ often follows ‘x’, and so on. This allows our model to create new words or handle misspellings gracefully.

The downside is that character-level processing requires more computational steps to generate the same amount of text. Generating “Hello world” requires 11 prediction steps instead of just 2. However, for our educational purposes, this trade-off provides valuable insights into how language generation works at its most fundamental level.

Unlike word-level tokenization, character-level tokenization treats each character as a token.

Advantages of Character-Level Tokenization:

- **No Out-of-Vocabulary Issues:** Every possible character sequence can be generated
- **Smaller Vocabulary:** Only 123 unique characters vs thousands of words
- **Language Agnostic:** Works with any language or symbol system
- **Handles Rare Words:** Can generate new words character by character

Please see the following site for a great general visual explanation, from Andrej Karpathy, [The Unreasonable Effectiveness of Recurrent Neural Networks](#).

Vocabulary Building Process

Computers work with numbers, not letters, so we need to convert our text into a numerical representation. We start by finding every unique character in our dataset. This includes not just letters A-Z and a-z, but also numbers, punctuation marks, spaces, and even special characters that might appear in the original texts.

Our Jules Verne collection contains 123 unique characters. These include obvious ones like letters and common punctuation, but also less common characters like accented letters from French names or special typography marks from the original publications.

Creating the Character Dictionary

We create two dictionaries: one that converts characters to numbers (encoding) and another that converts numbers back to characters (decoding). For example:

‘a’ might become 47, ‘b’ becomes 48, ‘c’ becomes 49, and so on. The space character might be 1, and the period might be 72. These assignments are arbitrary but consistent throughout our project.

When we want to process the phrase “The sea”, we convert it to something like [84, 72, 69, 1, 83, 69, 47]. When the model generates numbers like [84, 72, 69, 1, 87, 47, 83], we convert them back to “The was” (as an example).

```
# Create character-to-index mapping
text = "Your complete dataset text here..."
vocab = sorted(set(text))
char_to_idx = {char: idx for idx, char in enumerate(vocab)}
idx_to_char = {idx: char for idx, char in enumerate(vocab)}
```

```
print(f"Vocabulary size: {len(vocab)}")
print(f"Unique characters: {vocab}")
```

```
['\n', ' ', '!', '"', '#', '$', '%', '&', "", '(', ')', '*', '+',
',', '-', '.', '/', '0', '1', '2', '3', '4', '5', '6', '7', '8',
'9', ':', ';', '<', '=', '>', '?', 'A', 'B', 'C', 'D', 'E', 'F',
'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S',
'T', 'U', 'V', 'W', 'X', 'Y', 'Z', '[', ']', '_', 'a', 'b', 'c',
'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p',
'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '{', '}', '~',
'f', '°', '½', 'À', 'Ã', 'È', 'à', 'â', 'æ', 'ç', 'è', 'é', 'ê',
'ë', 'ï', 'ñ', 'ô', 'û', 'ü', 'Œ', 'œ', 'ѓ', '—', '„', '„',
'', '•', '„', '„', '™', '\ufe0f'], dtype='<U1')
```

```
[31, 42, 40, 1, 30, 28, 46, 40, 48, 45, 45, 42, 0, 0, 36, 0, 0,
31, 71, 1, 76, 65, 76, 77, 68, 71, 13, 0, 0, 48, 69, 57, 1, 70,
71, 65, 76, 61, 1, 60, 61, 75, 76, 57, 75, 11, 1, 78, 65, 70, 60,
71, 1, 60, 57, 1, 59, 65, 60, 57, 60, 61, 1, 72, 57, 74, 57, 1,
71, 1, 32, 70, 63, 61, 70, 64, 71, 1, 41, 71, 78, 71, 11, 1, 61,
70, 59, 71, 70, 76, 74, 61, 65, 1, 70, 71, 0, 76, 74, 61, 69, 1,
60, 57, 1, 30, 61, 70, 76, 74, 57, 68, 1, 77, 69, 1, 74, 57, 72,
57, 82, 1, 57, 73, 77, 65, 1, 60, 71, 1, 58, 57, 65, 74, 74, 71,
11, 1, 73, 77, 61, 1, 61, 77, 1, 59, 71, 70, 64, 61])
```

You can experiment with tokenization using OpenAI's tokenizer tool at:
<https://platform.openai.com/tokenizer>

Training Sequences

The Sliding Window Approach

Our model learns by playing a sophisticated prediction game. We show it sequences of 120 characters and ask it to predict what the 121st character should be. Think of it like a fill-in-the-blank exercise, but instead of missing words, we're missing the next character.

For example, if our text contains “The submarine descended into the dark depths of the ocean”, we might show the model “The submarine descended into the dark depths of the ocea” and ask it to predict “n”. Then we slide our window forward by one character and show it “he submarine descended into the dark depths of the ocean” and ask it to predict the next character.

Training Configuration

- **Sequence Length:** 120 characters (approximately one paragraph)
- **Input-Output Relationship:** Predict the next character given the previous characters

Why 120 Characters?

We chose 120 characters as our context window because it represents roughly one paragraph of text. This gives the model enough context to understand local patterns (like completing words and phrases) while remaining computationally manageable. In practical terms, 120 characters might look like:

“The Nautilus had been cruising in these waters for some time. Captain Nemo stood on the bridge, observing the vast exp”

From this context, the model might predict “a” to complete “expanse” or “l” to form “explore”.

The longer the context window, the better the model can maintain coherence, but the more computer memory and processing time it requires.

Training Example

Input Sequence: “Hello my nam” **Target Sequence:** “ello my name”

The model learns:

- Given “H”, predict “e”
- Given “He”, predict “l”
- Given “Hel”, predict “l”
- And so on...

This means our dataset of 5.8 million characters becomes millions of individual training examples, each teaching the model about character sequence patterns.

Creating Training Data

```
def create_training_sequences(text, seq_length):
    sequences = []
    targets = []

    for i in range(len(text) - seq_length):
        # Input sequence
        seq = text[i:i + seq_length]
        # Target (next character)
        target = text[i + 1:i + seq_length + 1]

        sequences.append([char_to_idx[char] for char in seq])
        targets.append([char_to_idx[char] for char in target])

    return sequences, targets
```

Character Embeddings

From Sparse to Dense Representation

Initially, each character is represented as a one-hot vector, which is mostly zeros with a single one indicating which character it is. For 123 characters, this means each character is represented by a vector with 123 elements, where 122 are zero and 1 is one. This is wasteful and doesn't capture any relationships between characters.

Character embeddings solve this problem by representing each character as a dense vector of real numbers. Instead of 123 mostly-zero values, each character becomes 256 meaningful numbers. These numbers are learned during training and end up encoding relationships between characters.

Learning Character Relationships

Something fascinating happens during training: characters that behave similarly end up with similar embedding vectors. Vowels tend to cluster together because they can often substitute for each other in similar contexts. Consonants that frequently appear together (like 'th' or 'ch') develop related embeddings.

The model learns that uppercase and lowercase versions of the same letter are related but distinct. It discovers that digits form their own cluster since they appear in similar contexts

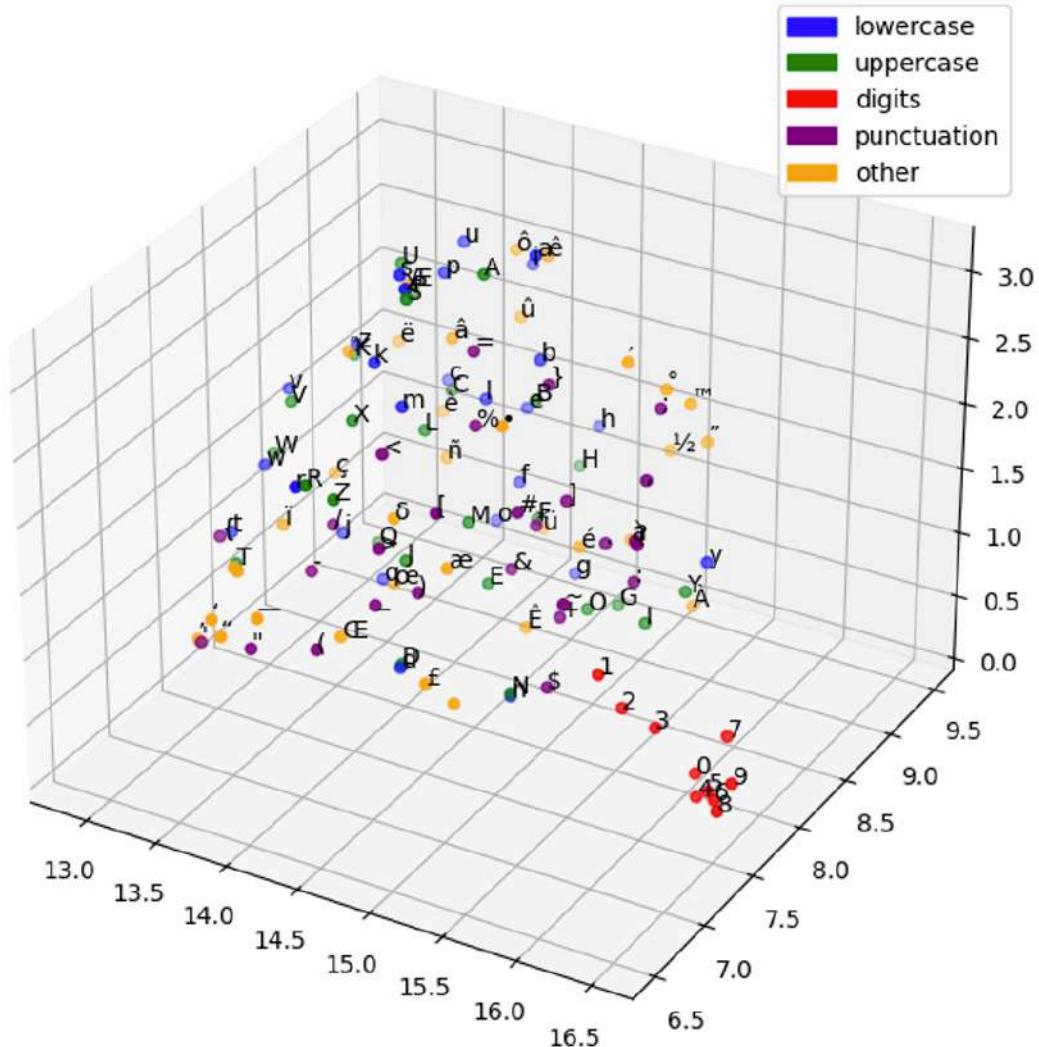
(dates, measurements, chapter numbers). Punctuation marks develop embeddings based on their grammatical functions.

Visualization and Understanding

When we project these 256-dimensional embeddings down to 3D space for visualization, we can see these learned relationships. The embedding space becomes a map where distance represents similarity. Characters that can substitute for each other in many contexts end up close together, while characters with completely different roles end up far apart.

This learned representation becomes the foundation for everything else the model does. The quality of these embeddings directly affects the model's ability to generate coherent text.

3D Projection of Character Embeddings Using UMAP



You can play with [Word2Vec - Embedding Projector](#)

Model Architecture

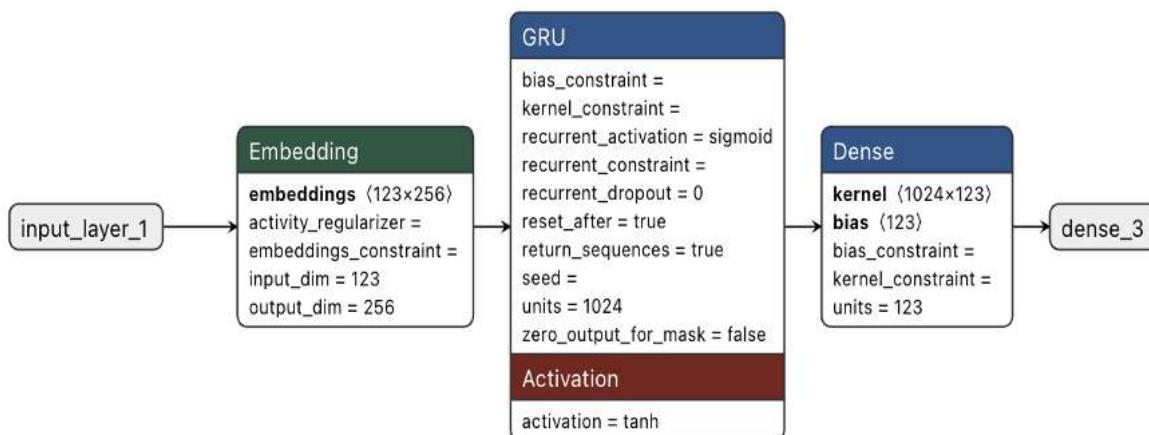
RNN Architecture Components

Our Jules Verne Bot consists of three main components, each serving a specific purpose in the text generation pipeline.

Embedding Layer: This is our translation layer. It takes character indices (numbers like 47, 83, 72) and converts them into dense 256-dimensional vectors that capture character relationships. Think of this as converting raw symbols into a format that captures meaning and relationships.

GRU Layer: This is the brain of our operation. With 1024 hidden units, this layer processes sequences and maintains memory about what it has seen. When processing the sequence “The submarine descended”, the GRU maintains a hidden state that encodes information about the submarine, the action of descending, and the overall maritime context.

Dense Output Layer: This is our decision-making layer. It takes the GRU’s 1024-dimensional hidden state and converts it into 123 probabilities, one for each character in our vocabulary. These probabilities represent the model’s confidence about what character should come next.



Model Summary

Model: "sequential_4"

Layer (type)	Output Shape	Param #
<hr/>		
embedding_4 (Embedding)	(1, 120, 256)	31,488

gru_3 (GRU)	(1, 120, 1024)	3,938,304
dense_3 (Dense)	(1, 120, 123)	126,075
=====		
Total params: 4,095,867 (15.62 MB)		
Trainable params: 4,095,867 (15.62 MB)		
Non-trainable params: 0 (0.00 B)		

Our model has 4,095,867 parameters. These are the individual numbers that the model adjusts during training to improve its predictions. To put this in perspective, each parameter is like a tiny dial that affects how the model processes information.

Training involves adjusting all 4 million dials to work together harmoniously.

The GRU layer contains most of these parameters (about 3.9 million) because it needs to learn complex patterns about how characters relate to each other across different time steps. The embedding layer has about 31,000 parameters (123 characters \times 256 dimensions), and the output layer has about 126,000 parameters.

Memory and Processing Flow

When processing text, information flows through the model like this:

A character index enters the embedding layer and becomes a 256-dimensional vector. This vector enters the GRU, which combines it with its current memory state to produce a new 1024-dimensional hidden state. This hidden state captures everything the model “knows” at this point in the sequence.

The hidden state goes to the dense layer, which produces probability scores for each of the 123 possible next characters. The character with the highest probability becomes the model’s prediction.

Crucially, the GRU’s hidden state becomes its memory for the next character prediction. This creates a chain of memory that allows the model to maintain context across the entire sequence.

Why GRU over Basic RNN?

GRU Advantages:

- **Solves Vanishing Gradient:** Better information flow through long sequences
- **Selective Memory:** Can choose what to remember and forget

- **Computational Efficiency:** Fewer parameters than LSTM
- **Better Performance:** More stable training than basic RNNs

Training Process: Teaching the Model to Write

The Learning Objective

Training a neural network means adjusting its millions of parameters so it makes better predictions. We use a loss function called sparse categorical crossentropy, which measures how far off the model's predictions are from the correct answers.

Think of it like teaching someone to play darts. Each throw (prediction) has a target (the correct next character). The loss function measures how far each dart lands from the bullseye. Training adjusts the player's technique (the model's parameters) to improve accuracy over time.

Hardware and Time Requirements

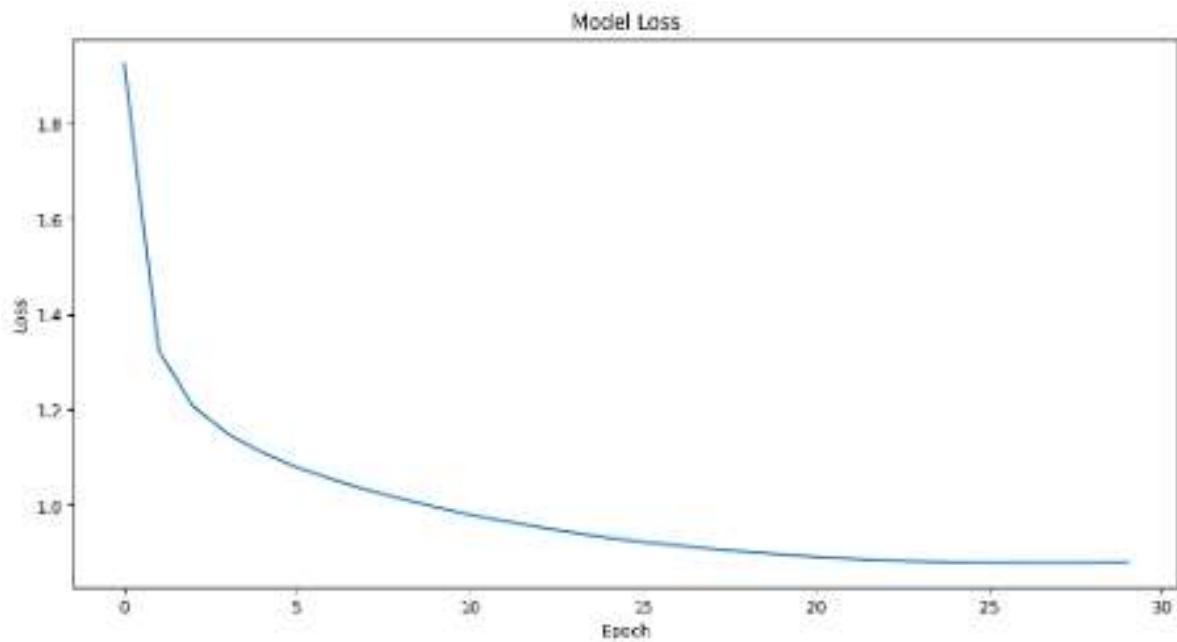
We trained our model on a **Tesla T4 GPU**, which can perform thousands of calculations simultaneously. This parallelization is crucial because each training step involves matrix multiplications with millions of numbers. The training took 33 minutes for 30 complete passes through the entire dataset.

To understand why we need a GPU, consider that training involves calculating gradients for all 4 million parameters, potentially thousands of times per second. A regular CPU would take days or weeks to complete the same training that a GPU accomplishes in minutes.

Monitoring Progress

During training, we watch the loss decrease from about 1.9 to 0.9. This represents the model's improving ability to predict the next character. Early in training, the model makes essentially random predictions. By the end, it has learned sophisticated patterns about English spelling, grammar, and Jules Verne's writing style.

The learning curve typically shows rapid improvement in the first few epochs as the model learns basic patterns like common letter combinations. Later epochs show slower but steady improvement as the model refines its understanding of more complex patterns like narrative structure and thematic elements.



Preventing Overfitting

One challenge in training is overfitting, where the model memorizes the training data instead of learning generalizable patterns. We use techniques like monitoring validation loss and potentially stopping training early if the model stops improving on unseen text.

Training Configuration

Hardware Setup:

- **GPU:** Tesla T4 (Google Colab)
- **GPU RAM:** 15.0 GB
- **Training Time:** 33 minutes for 30 epochs

Training Parameters:

- **Loss Function:** Categorical Sparse Crossentropy
- **Optimizer:** Adam (adaptive learning rate)
- **Epochs:** 30
- **Batch Size:** 128
- **Buffer Size:** 10,000 (for dataset shuffling)

Training Implementation

```
# Model compilation
model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

# Training with callbacks
history = model.fit(
    dataset,
    epochs=30,
    batch_size=128,
    validation_split=0.1,
    callbacks=[
        tf.keras.callbacks.ReduceLROnPlateau(patience=3),
        tf.keras.callbacks.EarlyStopping(patience=5)
    ]
)
```

Text Generation

The Generation Process

Once trained, our model becomes a text generation engine. We start with a seed phrase like:

“THE FLYING SUBMARINE”

and ask the model to continue the story. The process works character by character:

The model receives “THE FLYING SUBMARINE” and predicts the most likely next character based on everything it learned from Jules Verne’s works. Maybe it predicts a space, starting a new word. Then we feed “THE FLYING SUBMARINE” (with the space) back to the model and ask for the next character.

This process continues indefinitely, with each new character becoming part of the context for predicting the next one. The model might generate “THE FLYING SUBMARINE descended into the mysterious depths...” as it draws upon patterns learned from Verne’s nautical adventures.

Temperature Control

Here's where we can control the model's creativity through a parameter called **temperature**. Temperature affects how the model chooses between different possible next characters.

With temperature set to 0.1, the model almost always picks the most probable next character. This produces very predictable, conservative text that closely mimics the training data but might be repetitive or boring.

With temperature set to 1.0, the model considers all possible next characters according to their learned probabilities. This produces more varied and creative text, but sometimes makes unusual choices that lead to interesting narrative directions.

With temperature above 1.5, the model becomes quite random, often producing text that starts coherently but gradually becomes nonsensical as unlikely character combinations accumulate.

In short:

- **Temperature = 0.5**: More predictable, conservative text
- **Temperature = 1.0**: More creative, diverse text
- **Temperature = 1.5**: Very random, potentially nonsensical text

Implementation

```
def generate_text(model, start_string, num_generate=1000, temperature=1.0):
    # Convert start string to numbers
    input_eval = [char_to_idx[s] for s in start_string]
    input_eval = tf.expand_dims(input_eval, 0)

    text_generated = []
    model.reset_states()

    for i in range(num_generate):
        predictions = model(input_eval)
        predictions = tf.squeeze(predictions, 0)

        # Apply temperature
        predictions = predictions / temperature
        predicted_id = tf.random.categorical(predictions, num_samples=1)[-1,0].numpy()

        # Add predicted character to input
        input_eval = tf.expand_dims([predicted_id], 0)
```

```
    text_generated.append(idx_to_char[predicted_id])  
  
    return start_string + ''.join(text_generated)
```

Generation Example (Temperature = 0.5)

Seed: "THE FLYING SUBMARINE"

Generated Text:

THE FLYING SUBMARINE
CHAPTER 100 VENTANTILE

This eBook is for the use of anyone anywhere in the United States and most other parts of the earth and miserable eruptions. The solar rays should be entirely under the shock of the intensity of the sea. We were all sorts. Are we to prepare for our feelings?"

"I can never see them a good geographer," said Mary.

"Well, then, John, for I get to the Pampas, that we ought to obey the same time. In the country of this latitude changed my brother, and the Nautilus floated in a sea which contained the rudder and lower colour visibly. The loiter was a fatalint region the two scientific discoverers. Several times turning toward the river, the cry of doors and over an inclined plains of the Angara, with a threatening water and disappeared in the midst of the solar rays.

The weather was spread and strewn with closed bottoms which soon appeared that the unexpected sheets of wind was soon and linen, and the whole seas were again landed on the subject of the natives, and the prisoners were successively assuming the sides of this agreement for fifteen days with a threatening voice.

...

Example Output Analysis

Let's examine some generated text: "The weather was spread and strewn with closed bottoms which soon appeared that the unexpected sheets of wind was soon and linen, and the whole seas were again landed on the subject of the natives..."

This excerpt shows both the model’s strengths and limitations. It successfully captures Verne’s descriptive style and maritime vocabulary (“seas,” “wind,” “natives”). The sentence structure feels appropriately Victorian and elaborate. However, the meaning becomes confused with phrases like “closed bottoms” and “sheets of wind was soon and linen.”

This illustrates the fundamental challenge of character-level generation: the model learns local patterns (how words are spelled, common phrases) much better than global coherence (logical narrative flow, consistent meaning).

Challenges and Limitations

Context Window Constraints

Our 120-character context window creates a fundamental limitation. The model can only “see” about one paragraph of previous text when making predictions. This means it might introduce a character named Captain Smith, then 200 characters later introduce another character with the same name, having “forgotten” the first introduction.

Humans writing stories maintain mental models of characters, plot lines, and world-building details across entire novels. Our model’s memory effectively resets every 120 characters, making long-term narrative consistency nearly impossible.

Character vs. Word Level Trade-offs

Character-level generation requires many more prediction steps than word-level generation. Generating the phrase “extraordinary adventure” requires 22 character predictions instead of just 2 word predictions. This makes character-level generation much slower and more computationally expensive.

However, character-level generation offers unique advantages. The model can generate new words it has never seen before by combining character patterns. It can handle misspellings, made-up words, or technical terms more gracefully than word-level models that have fixed vocabularies.

Coherence Challenges

Perhaps the biggest limitation is maintaining semantic coherence. The model might generate grammatically correct text that makes no logical sense. It can describe “The submarine floating in the air above the mountain peaks” because it has learned that submarines float and that Verne often described mountains, but it hasn’t learned the physical constraint that submarines float in water, not air.

This happens because the model learns statistical patterns without understanding meaning. It knows that certain word combinations are common without understanding why they make sense.

Summary

1. Limited Context Window

- **Issue:** Only 120 characters of context
- **Impact:** Cannot maintain coherence over long passages
- **Example:** May forget characters or plot points mentioned earlier

2. Character vs Word Level

- **Issue:** Character-level generation is slower and less efficient
- **Impact:** Requires more computation for equivalent output
- **Trade-off:** Better handling of rare words vs efficiency

3. Coherence Problems

- **Issue:** May generate grammatically correct but semantically inconsistent text
- **Cause:** Limited understanding of story structure and plot consistency

4. Repetitive Patterns

- **Issue:** May fall into repetitive loops
- **Cause:** Model overfitting to common patterns in training data

Potential Improvements

1. **Longer Context Windows:** Increase sequence length for better coherence
2. **Hierarchical Models:** Separate models for different text levels (word, sentence, paragraph)
3. **Fine-tuning:** Additional training on specific styles or topics
4. **Beam Search:** Better text generation algorithms instead of greedy sampling

What will lead us to modern **Language Models** based on Transformers arquiteture .

Connecting to Modern Language Models

Scale Comparison

To appreciate how far language modeling has advanced, consider the scale differences between our Jules Verne Bot and modern language models:

Our model has 4 million parameters and was trained on 5.8 million characters (about 10 books). GPT-3 has 175 billion parameters and was trained on 45 terabytes of text (roughly equivalent to millions of books). That's a difference of over 40,000 times more parameters and millions of times more training data.

Modern small language models (SLMs) like Phi-3-mini still dwarf our model with 3.8 billion parameters, but they represent more efficient designs that achieve impressive performance with “only” 1,000 times more parameters than our model.

Aspect	Jules Verne Bot	GPT-3 (2020)	Phi-3-mini (2024)
Architecture	RNN (GRU)	Transformer	Transformer
Parameters	4 million	175 billion	3.8 billion
Training Data	5.8M characters	45TB text	3.3T tokens
Context Length	120 characters	2,048 tokens	128,000 tokens
Tokenization	Character-level	Subword (BPE)	Subword
Training Time	33 minutes	Months	7 days
GPU Requirements	1 Tesla T4	Thousands of GPUs	512 H100 GPUs

Architectural Evolution

The biggest advancement since RNNs is the **Transformer architecture**, which uses attention mechanisms instead of recurrent processing. While RNNs process text sequentially (like reading word by word), Transformers can examine all parts of a text simultaneously and learn relationships between any two words, regardless of how far apart they are.

This solves the long-term memory problem that limits our RNN model. A Transformer can maintain awareness of a character introduced in a hypothetical “chapter 1” while writing “chapter 10”, something our 120-character context window makes impossible.

Training Efficiency

Modern models also benefit from more sophisticated training techniques. They’re pre-trained on massive, diverse datasets to learn general language patterns, then fine-tuned on specific tasks. They use techniques like instruction tuning, where they learn to follow human commands, and reinforcement learning from human feedback, where they learn to generate text that humans find helpful and appropriate.

Summary: Why Modern Models Perform Better?

1. Transformer Architecture

- **Attention Mechanism:** Can look at any part of the input sequence
- **Parallel Processing:** Much faster training and inference
- **Better Long-range Dependencies:** Maintains context over thousands of tokens

2. Scale

- **More Data:** Trained on vastly more diverse text
- **More Parameters:** Can memorize and generalize better
- **More Compute:** Allows for more sophisticated training techniques

3. Advanced Techniques

- **Pre-training + Fine-tuning:** Learn general language then specialize
- **Instruction Tuning:** Trained to follow human instructions
- **RLHF:** Reinforcement Learning from Human Feedback

Conclusion:

Building the Jules Verne Bot teaches us that creating artificial intelligence systems capable of generating human-like text requires careful consideration of multiple components working together. The embedding layer learns to represent characters meaningfully, the RNN layer processes sequences and maintains memory, and the output layer makes predictions based on learned patterns.

The project also illustrates the fundamental trade-offs in machine learning: between model complexity and training speed, between creativity and coherence, between local accuracy and global consistency. These trade-offs appear in every AI system, from simple character-level generators to the most sophisticated language models.

Most importantly, this project demonstrates that impressive AI capabilities emerge from relatively simple components combined thoughtfully. Our 4-million parameter model, while limited compared to modern systems, genuinely learns to write in Jules Verne's style through nothing more than statistical pattern recognition and mathematical optimization.

The techniques we've explored, sequence processing, embedding learning, and generation strategies, form the foundation for understanding any language model. Whether you encounter RNNs, Transformers, or future architectures yet to be invented, the core concepts remain consistent: learn patterns from data, encode meaning in mathematical representations, and generate new content by predicting what should come next.

Understanding these fundamentals provides the foundation for working with, improving, or creating the next generation of language models that will shape how humans and computers communicate in the future.

Resources

- Gutenberg Project
- The Unreasonable Effectiveness of Recurrent Neural Networks
- Word2Vec - Embedding Projector
- OpenAI's tokenizer tool
- Generating Text with RNNs: The Jules Verne Bot - CoLab

Knowledge Distillation in Practice

From MNIST to LLMs

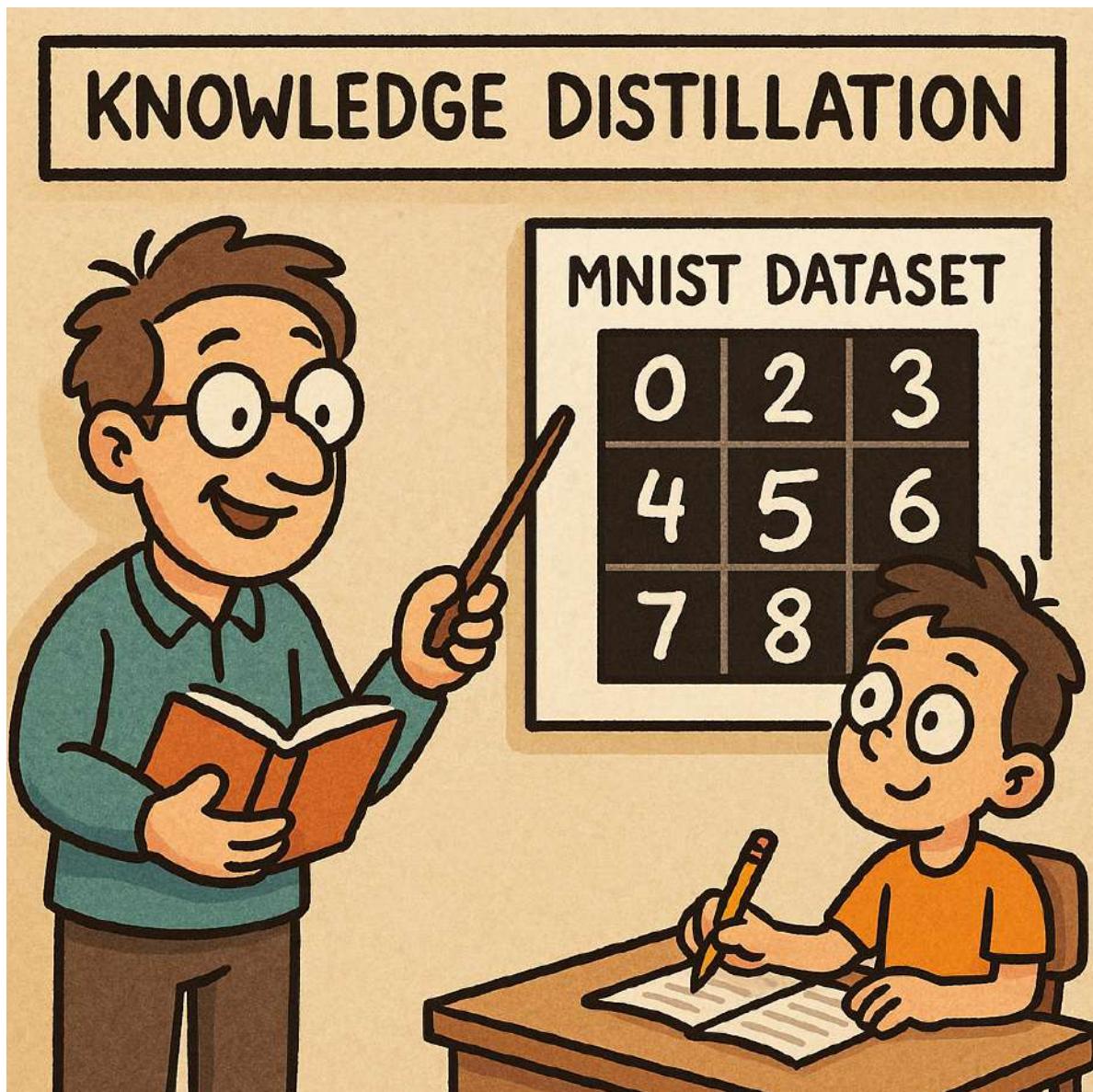


Figure 6: Image created by DALLE-3

Introduction to Knowledge Distillation

Knowledge distillation is a powerful technique in machine learning that enables the transfer of knowledge from a large, complex model (the “teacher”) to a smaller, more efficient model (the “student”). This process allows us to create compact models that maintain much of the

performance of their larger counterparts while being significantly faster and requiring fewer computational resources.

Why Knowledge Distillation Matters

In today's AI landscape, models are becoming increasingly large and complex. While these models achieve remarkable performance, they often require substantial computational resources, making deployment challenging in resource-constrained environments such as mobile devices, edge computing systems, or real-time applications. Knowledge distillation addresses this challenge by:

1. **Reducing Model Size:** Creating smaller models with fewer parameters
2. **Improving Inference Speed:** Enabling faster predictions in production
3. **Lowering Computational Costs:** Reducing memory and processing requirements
4. **Maintaining Performance:** Preserving much of the original model's accuracy
5. **Enabling Edge Deployment:** Making AI accessible on resource-limited devices

The Teacher-Student Paradigm

The core concept of knowledge distillation revolves around the teacher-student relationship:

- **Teacher Model:** A large, well-trained model with high capacity and performance
- **Student Model:** A smaller, more efficient model trained to mimic the teacher's behavior
- **Knowledge Transfer:** The process of transferring the teacher's "dark knowledge" to the student

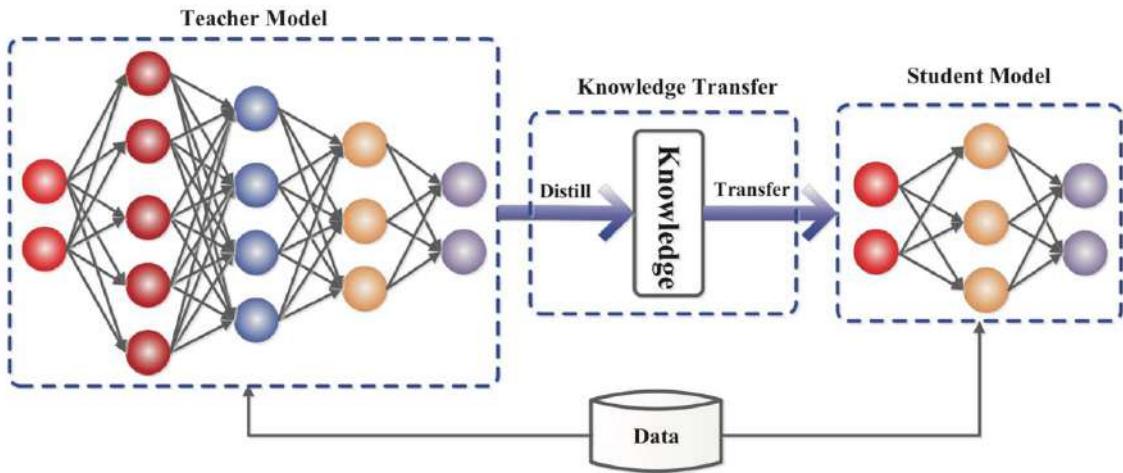


Figure 7: Figure from “Knowledge Distillation: A Survey, Jianping Gou Baosheng Yu Stephen J. Maybank Dacheng Tao, 2021

See paper: “Knowledge Distillation: A Survey, Jianping Gou, Baosheng Yu, Stephen J. Maybank, Dacheng Tao, 2021

Theoretical Foundations

Soft Targets vs. Hard Targets

Traditional supervised learning uses “hard targets” - one-hot encoded labels that provide limited information. For example, in MNIST digit classification, the label for the digit “5” would be represented as [0, 0, 0, 0, 0, 1, 0, 0, 0, 0].

Knowledge distillation leverages “soft targets” - the probability distributions produced by the teacher model. These soft targets contain richer information about the relationships between classes. For instance, the teacher might output [0.01, 0.02, 0.01, 0.05, 0.1, 0.78, 0.02, 0.01, 0.0, 0.0] for a “5”, indicating that it’s most confident about “5” but also considers “4” somewhat similar.

The Temperature Parameter

The temperature parameter () is crucial in knowledge distillation. It controls the “softness” of the probability distribution by modifying the softmax function:

$$P(\text{class}_i) = \exp(z_i / \tau) / \sum_j \exp(z_j / \tau)$$

Where:

- z_i are the logits (pre-softmax outputs)
- τ is the temperature parameter

Effects of Temperature:

- $\tau = 1$: Standard softmax (normal sharpness)
- $\tau > 1$: Softer distribution (more uniform, reveals class relationships)
- $\tau < 1$: Sharper distribution (more peaked, less informative)

In our implementation, we use a temperature of 5.0, which creates softer distributions that effectively transfer the teacher's "dark knowledge" to the student.

Distillation Loss Function

The distillation loss combines two components:

1. **Distillation Loss (L_{KD})**: Measures how well the student mimics the teacher's soft targets
2. **Student Loss (L_{CE})**: Traditional cross-entropy loss with hard targets

$$L_{total} = \alpha * L_{CE} + (1 - \alpha) * L_{KD}$$

Where α is a weighting parameter that balances the two objectives, in our implementation, we use $\alpha = 0.3$, giving more weight to the soft targets from the teacher.

The distillation loss is typically computed using the KL divergence:

$$L_{KD} = \frac{1}{n} * KL_divergence(Teacher_soft_targets, Student_soft_targets)$$

The $\frac{1}{n}$ factor compensates for the gradient scaling effect of temperature. This is crucial for stable training.

Implementation with TensorFlow and MNIST

Dataset Overview

MNIST is an ideal dataset for learning knowledge distillation:

- **60,000 training images** of handwritten digits (0-9)
- **10,000 test images**

- 28x28 grayscale images
- 10 classes (digits 0-9)
- Well-established baseline performances

Teacher Model Architecture

Our teacher model has substantial capacity with multiple convolutional layers, batch normalization, and dropout for regularization:

```
def build_teacher_model():
    model = models.Sequential([
        layers.Conv2D(64, 3,
                     activation='relu',
                     padding='same',
                     input_shape=(28, 28, 1)),
        layers.BatchNormalization(),
        layers.Conv2D(64, 3, activation='relu', padding='same'),
        layers.MaxPooling2D(2, 2),
        layers.BatchNormalization(),

        layers.Conv2D(128, 3, activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.Conv2D(128, 3, activation='relu', padding='same'),
        layers.MaxPooling2D(2, 2),
        layers.BatchNormalization(),

        layers.Conv2D(256, 3, activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.GlobalAveragePooling2D(),

        layers.Dense(256, activation='relu'),
        layers.BatchNormalization(),
        layers.Dropout(0.5),
        layers.Dense(128, activation='relu'),
        layers.BatchNormalization(),
        layers.Dropout(0.5),
        layers.Dense(10, activation='softmax')
    ])
    return model
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 64)	640
batch_normalization (BatchNormalization)	(None, 28, 28, 64)	256
conv2d_1 (Conv2D)	(None, 28, 28, 64)	36,928
max_pooling2d (MaxPooling2D)	(None, 14, 14, 64)	0
batch_normalization_1 (BatchNormalization)	(None, 14, 14, 64)	256
conv2d_2 (Conv2D)	(None, 14, 14, 128)	73,856
batch_normalization_2 (BatchNormalization)	(None, 14, 14, 128)	512
conv2d_3 (Conv2D)	(None, 14, 14, 128)	147,584
max_pooling2d_1 (MaxPooling2D)	(None, 7, 7, 128)	0
batch_normalization_3 (BatchNormalization)	(None, 7, 7, 128)	512
conv2d_4 (Conv2D)	(None, 7, 7, 256)	295,168
batch_normalization_4 (BatchNormalization)	(None, 7, 7, 256)	1,024
global_average_pooling2d (GlobalAveragePooling2D)	(None, 256)	0
dense (Dense)	(None, 256)	65,792
batch_normalization_5 (BatchNormalization)	(None, 256)	1,024
dropout (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 128)	32,896
batch_normalization_6 (BatchNormalization)	(None, 128)	512
dropout_1 (Dropout)	(None, 128)	0
dense_2 (Dense)	(None, 10)	1,290

Total params: 658,250 (2.51 MB)

Trainable params: 656,202 (2.50 MB)

Non-trainable params: 2,048 (8.00 KB)

This architecture achieved **99.44 % accuracy** on MNIST.

Student Model Architecture

Our student model is intentionally much simpler, with fewer layers and significantly fewer parameters:

```
def build_student_model():
    model = models.Sequential([
        layers.Conv2D(16, 3,
                     activation='relu',
                     padding='same',
                     input_shape=(28, 28, 1)),
        layers.MaxPooling2D(2, 2),
        layers.Conv2D(32, 3, activation='relu', padding='same'),
        layers.MaxPooling2D(2, 2),
        layers.Flatten(),
        layers.Dense(64, activation='relu'),
        layers.Dense(10, activation='softmax')
    ])
    return model
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d_5 (Conv2D)	(None, 28, 28, 16)	160
max_pooling2d_2 (MaxPooling2D)	(None, 14, 14, 16)	0
conv2d_6 (Conv2D)	(None, 14, 14, 32)	4,640
max_pooling2d_3 (MaxPooling2D)	(None, 7, 7, 32)	0
flatten (Flatten)	(None, 1568)	0
dense_3 (Dense)	(None, 64)	100,416
dense_4 (Dense)	(None, 10)	650

Total params: 105,866 (413.54 KB)

Trainable params: 105,866 (413.54 KB)

Non-trainable params: 0 (0.00 B)

The student model has fewer parameters than the teacher (105K versus 658K), or 6.2 times smaller.

Knowledge Distillation Implementation

In our implementation, we use a custom training loop that explicitly calculates both hard and soft losses:

```
# Forward pass
with tf.GradientTape() as tape:
    predictions = kd_student(x_batch, training=True)

    # Hard target loss (cross-entropy with true labels)
    hard_loss = tf.keras.losses.categorical_crossentropy(y_hard_batch, predictions)

    # Soft target loss (KL divergence with teacher predictions)
    soft_loss = tf.keras.losses.kullback_leibler_divergence(y_soft_batch, predictions)

    # Combined loss with temperature scaling factor for soft loss
    loss = alpha * hard_loss + (1 - alpha) * soft_loss * (temperature ** 2)
```

Key components of our implementation:

1. **Temperature Scaling:** We apply a temperature of 5.0 to soften the teacher's outputs
2. **Alpha Weighting:** We use $\alpha = 0.3$ to prioritize learning from the teacher's soft targets
3. **KL Divergence:** This loss function helps the student match the teacher's probability distributions
4. **Temperature Correction:** We multiply the soft loss by 5^2 to correct for gradient scaling

Training Process

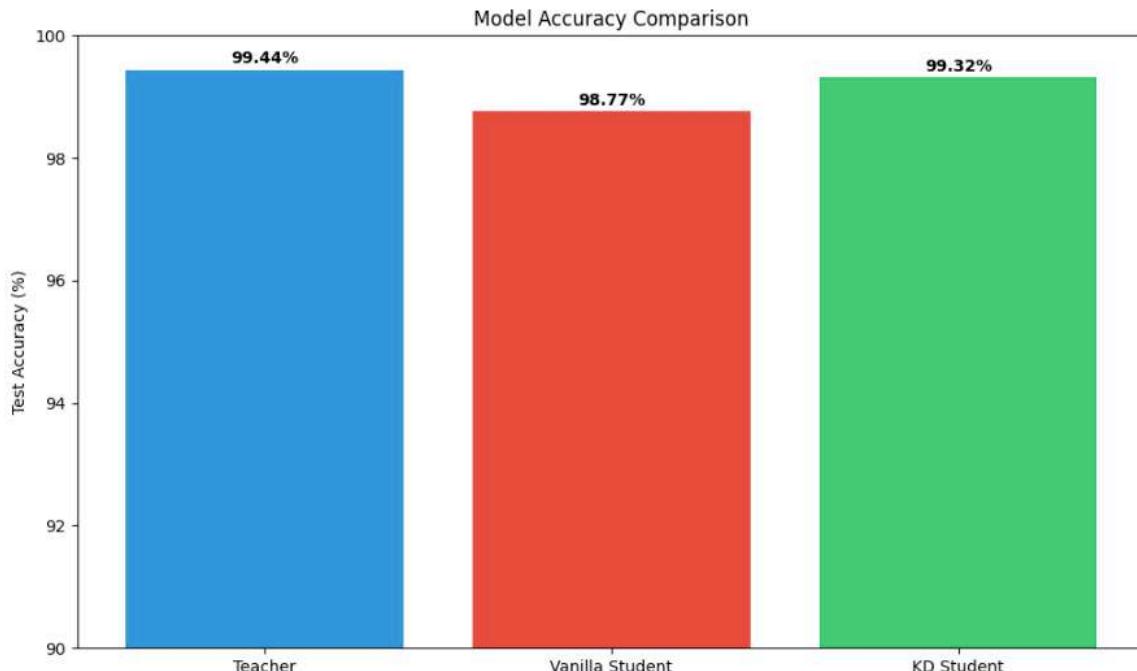
Our implementation follows these steps:

1. **Train the Teacher:** First, we train the complex teacher model using early stopping and a reduced learning rate.
2. **Train a Vanilla Student:** We train a student model normally on the hard labels for comparison.
3. **Generate Soft Targets:** We use the teacher to create softened probability distributions.
4. **Train the Distilled Student:** We train another student using our custom distillation training loop.
5. **Evaluate and Compare:** We compare the performance, size, and speed of all three models.

Results and Analysis

Our implementation typically shows these results:

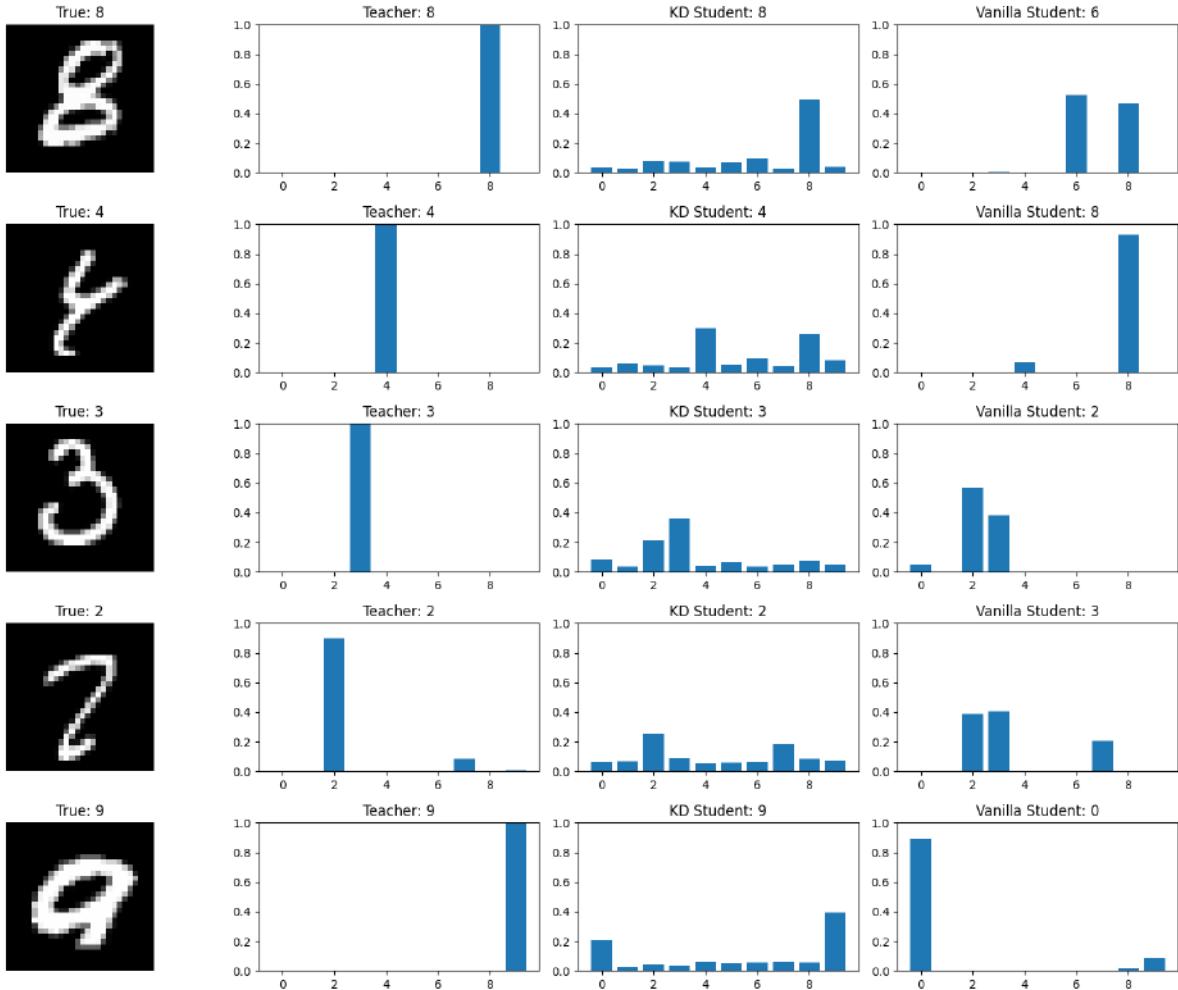
- **Teacher Model:** 99.44% accuracy, largest size, slowest inference (0.8632 seconds)
- **Vanilla Student:** 98.77% accuracy, smaller size, faster inference (0.3579 seconds)
- **Distilled Student:** 99.32% accuracy, same size as vanilla student, but better performance (+0.55%) and faster inference than the Teacher (0.5467 seconds)



These results demonstrate the key benefit of knowledge distillation: the distilled student achieves performance closer to the teacher while maintaining the efficiency benefits of the smaller architecture.

Visualizing Distillation Benefits

0.0.0.0.1 * We analyze the models on challenging examples where the teacher succeeds but the vanilla student fails. This reveals how knowledge distillation enables students to handle complex cases by learning the teacher’s “dark knowledge.”



Advanced Techniques

Feature-Based Distillation

Beyond output-level distillation, we can transfer knowledge from intermediate layers:

```
def feature_distillation_loss(teacher_features, student_features):
    """Distill knowledge from intermediate feature maps"""
    loss = 0
    for t_feat, s_feat in zip(teacher_features, student_features):
        # Align dimensions if necessary
```

```
s_feat_aligned = align_feature_dimensions(s_feat, t_feat.shape)
loss += tf.keras.losses.MSE(t_feat, s_feat_aligned)
return loss
```

Attention-Based Distillation

Transfer attention patterns from teacher to student:

```
def attention_distillation_loss(teacher_attention, student_attention):
    """Distill attention mechanisms"""
    return tf.keras.losses.MSE(teacher_attention, student_attention)
```

Practical Tips for Advanced Distillation

1. **Layer Alignment:** When using feature distillation, carefully align the feature dimensions
2. **Feature Selection:** Not all features are equally important; focus on the most informative ones
3. **Multi-Teacher Distillation:** Combine knowledge from multiple teachers for better results
4. **Online Distillation:** Train teacher and student simultaneously for mutual improvement

Scaling to Large Language Models

Challenges in LLM Distillation

Scaling knowledge distillation to Large Language Models presents unique challenges:

1. **Model Size:** LLMs have billions of parameters, making distillation computationally expensive
2. **Sequence Generation:** Unlike classification, LLMs generate sequences, requiring sequence-level distillation
3. **Vocabulary Differences:** Teacher and student may have different vocabularies
4. **Context Length:** Handling variable-length sequences and attention patterns
5. **Multi-task Learning:** LLMs perform multiple tasks simultaneously

LLM Distillation Techniques

1. Sequence-Level Distillation

Instead of token-level predictions, match entire sequence probabilities:

```
def sequence_level_distillation(teacher_logits, student_logits, sequence_lengths):
    """Distill at the sequence level for better coherence"""
    teacher_probs = tf.nn.softmax(teacher_logits / temperature)
    student_probs = tf.nn.softmax(student_logits / temperature)

    # Mask out padding tokens
    mask = create_sequence_mask(sequence_lengths)

    return masked_kl_divergence(teacher_probs, student_probs, mask)
```

2. Recent LLM Distillation Examples

- **DistilBERT**: 40% smaller than BERT, retains 97% of performance.
- **TinyBERT**: 7.5x smaller and 9.4x faster than BERT-base
- **MiniLM**: Uses deep self-attention distillation for efficient transfer.
- **Distil-GPT2**: Compressed version of GPT-2 with minimal performance loss
- **Llama 3.2 3B and 1B**: Distilled from Llama 3.2 models (70B/8B parameters)

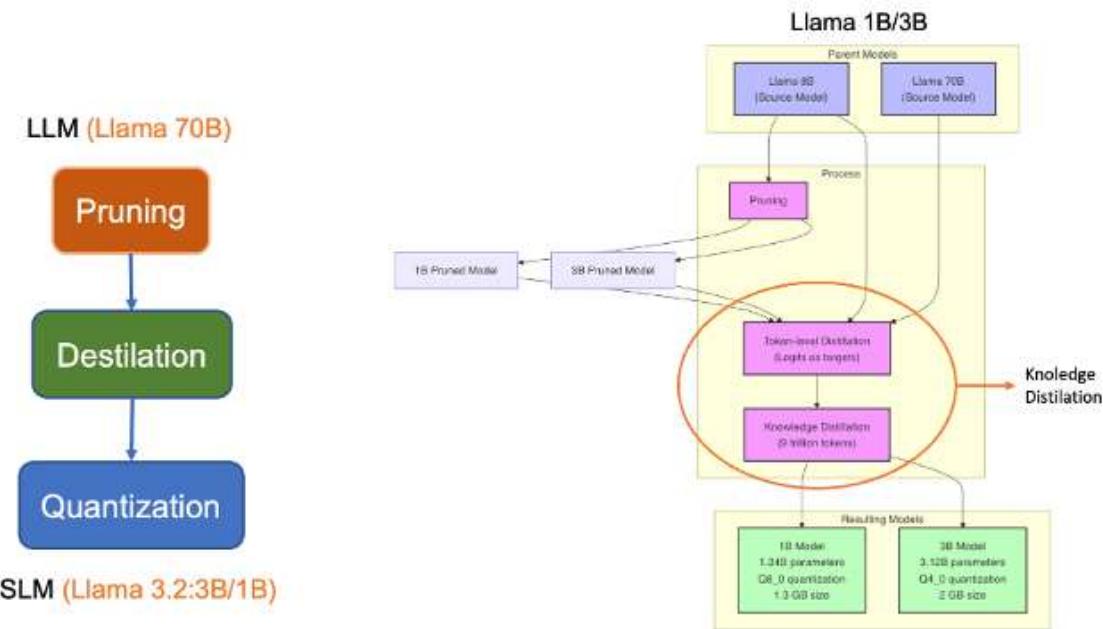
3. Distilling Reasoning Abilities

Modern approaches focus on transferring reasoning abilities, not just predictions:

- **Chain-of-Thought Distillation**: Transfer step-by-step reasoning process
- **Explanation-Based Distillation**: Use teacher's explanations to guide student learning
- **Rationale Extraction**: Identify key reasoning patterns for targeted transfer

From MNIST to Real-World LLMs: Llama 3.2 Case Study

1. **Teacher Model**: The larger Llama 3.2 models (70B/8B parameters) serve as the teachers
2. **Student Model**: Llama 3.2 3B and 1B are the distilled student models. They were created using pruning techniques, which systematically remove less meaningful connections (weights) in the neural network.



Key Points

- Scale Difference:** The parameter reduction from 70B to 3B (~23x) or 1B (~70x) demonstrates industrial-scale distillation
- Performance Preservation:** Despite massive size reduction, the smaller models maintain impressive capabilities:
 - The 3B model preserves most of the reasoning abilities of larger models.
 - The 1B model remains highly functional for many tasks.
- Practical Benefits:**
 - The 1B model can run on consumer laptops and even some mobile devices.
 - The 3B model offers a balance of performance and accessibility.
- Distillation Techniques Used:**
 - Meta likely used a combination of response-based and feature-based distillation.
 - They may have employed temperature scaling and specialized loss functions similar to what we demonstrated in our MNIST example.
 - The principles we covered (soft targets, temperature, loss weighting) all apply at this larger scale.

While our MNIST example demonstrates the application of knowledge distillation principles using a simple dataset, these same principles can be applied directly to state-of-the-art language models.

Meta's Llama 3.2 family provides a perfect real-world example:

Model	Parameters	Size		Use Case
		Reduction	(Teacher)	
Llama 3.2 70B	70 billion	~9x	(Teacher)	Data centers, high-performance applications
Llama 3.2 8B	8 billion	~23x		Server deployment, high-end workstations
Llama 3.2 3B	3 billion	~70x		Consumer laptops, desktop applications
Llama 3.2 1B	1 billion			Edge devices, mobile applications, embedded systems

The 3B and 1B models represent successful distillations of the larger models, preserving core capabilities while dramatically reducing computational requirements. This demonstrates the industrial importance of knowledge distillation techniques we've explored.

It is possible to note that while the underlying principles remain the same as our MNIST example, industrial LLM distillation includes additional techniques:

1. **Distillation-Specific Data:** Carefully curated datasets designed to transfer specific capabilities
2. **Multi-Stage Distillation:** Gradual compression through intermediate models ($70B \rightarrow 8B \rightarrow 3B \rightarrow 1B$)
3. **Task-Specific Fine-Tuning:** Optimizing smaller models for specific use cases after distillation
4. **Custom Loss Functions:** Specialized loss terms to preserve reasoning patterns and generation quality

Despite these additional complexities, the core concept remains the same: using a larger, more capable model to guide the training of a smaller, more efficient one.

Applications in Modern LLM Development

1. From ChatGPT to Mobile Assistants

Large models like ChatGPT (175B+ parameters) can be distilled to create mobile-friendly assistants (1-2B parameters) that maintain core capabilities while running locally on smartphones.

2. Domain-Specific Distillation

Instead of distilling general capabilities, focus on specific domains:

- **Medical LLMs:** Distill medical knowledge from large models to smaller, specialized ones
- **Legal Assistants:** Create compact models focused on legal reasoning and terminology
- **Educational Tools:** Develop small models optimized for teaching specific subjects

3. From Research to Production

The journey from research models to production deployment often involves distillation:

1. **Research Phase:** Develop large, state-of-the-art models (100B+ parameters)
2. **Distillation Phase:** Compress knowledge into deployment-ready models (1-10B parameters)
3. **Deployment Phase:** Further optimize for specific hardware and latency requirements

Practical Guidelines and Best Practices

Choosing the Right Architecture

Based on our experiments, we recommend:

1. **Teacher-Student Size Ratio:** Aim for a 10-20x parameter reduction for significant efficiency gains
2. **Architectural Similarity:** Maintain similar architectural patterns between teacher and student
3. **Bottleneck Identification:** Ensure the student has adequate capacity at critical layers

Hyperparameter Selection

Our experiments suggest these optimal settings:

1. **Temperature ():**
 - For MNIST: 3-5 works well
 - For complex tasks: 5-10 may be better
 - If outputs are already soft: Lower temperatures (2-3) may suffice
2. **Alpha Weighting ():**

- For simpler tasks: 0.3-0.5 (balanced approach)
- For complex reasoning: 0.1-0.3 (more emphasis on teacher's knowledge)
- When teacher is extremely accurate: Lower values work better

3. Training Duration:

- Distilled students often benefit from longer training (1.5-2x the epochs)
- Use early stopping with patience to avoid overfitting

Common Pitfalls and Solutions

1. **Teacher Performance:** Ensure the teacher actually outperforms the student (we fixed this in our implementation)
2. **Temperature Selection:** If knowledge transfer is poor, experiment with different temperatures
3. **Loss Weighting:** If the student ignores soft targets, reduce α to emphasize distillation loss
4. **Gradient Scaling:** Always apply the β^2 correction factor to the soft loss

Performance Evaluation

Always measure multiple dimensions of performance:

1. **Accuracy:** Primary performance metric (should be closer to teacher than vanilla student)
2. **Model Size:** Parameter count and memory footprint (should match vanilla student)
3. **Inference Speed:** Time per prediction (should be significantly faster than teacher)
4. **Challenging Cases:** Performance on difficult examples (should be better than vanilla student)

Conclusion

Knowledge distillation provides a powerful approach to create efficient, deployable models without sacrificing performance. Our implementation demonstrates that even with a 15-20x reduction in model size, we can maintain performance close to the larger teacher model.

Key Takeaways

1. **Efficiency Without Sacrifice:** Knowledge distillation enables smaller models to achieve performance similar to larger ones
2. **Dark Knowledge Matters:** The soft probability distributions contain valuable information beyond just the predicted class
3. **Temperature and Alpha:** These hyperparameters are crucial for effective knowledge transfer
4. **Practical Benefits:** Smaller size, faster inference, and lower resource requirements make AI more accessible

From MNIST to LLMs

The principles we've demonstrated with MNIST directly scale to Large Language Models:

1. **Same Core Concept:** Transfer knowledge from larger to smaller models
2. **Same Hyperparameters:** Temperature and alpha weighting remain critical
3. **Same Benefits:** Size reduction, speed improvement, and accessibility
4. **Same Challenges:** Ensuring adequate student capacity and effective knowledge transfer

Final Thoughts

Knowledge distillation isn't just an academic technique—it's essential for practical AI deployment. The same principles that helped us compress our MNIST classifier can be scaled to compress models with hundreds of billions of parameters. This universality makes knowledge distillation an indispensable skill for engineering students entering the field of AI.

Resources

[Notebook](#)

Small Language Models (SLM)

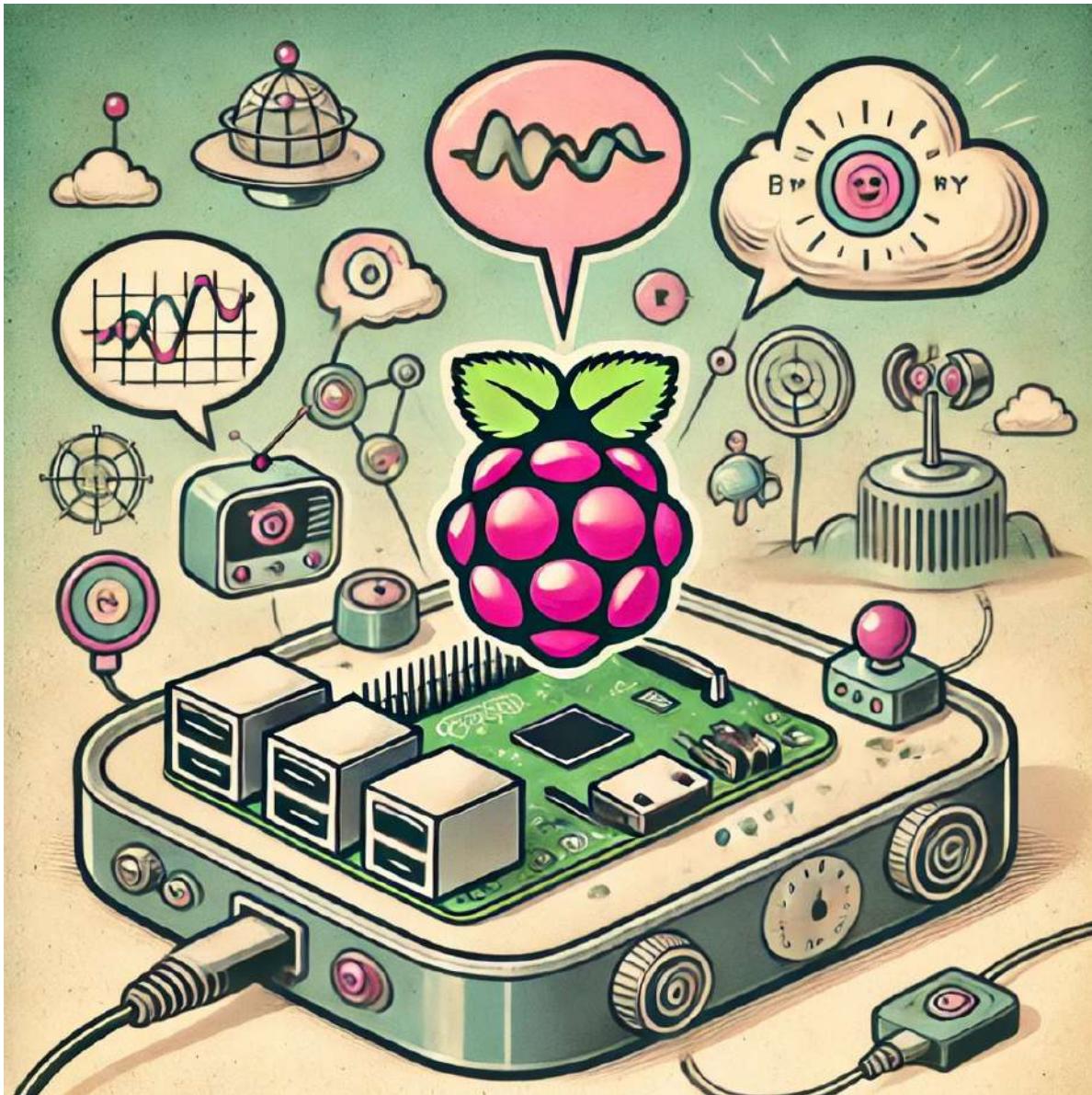


Figure 8: *DALL·E prompt* - A 1950s-style cartoon illustration showing a Raspberry Pi running a small language model at the edge. The Raspberry Pi is stylized in a retro-futuristic way with rounded edges and chrome accents, connected to playful cartoonish sensors and devices. Speech bubbles are floating 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 chapter 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 chapter guides us 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 (desktops or edge devices such as NVIDIA Jetson or Raspberry Pis).

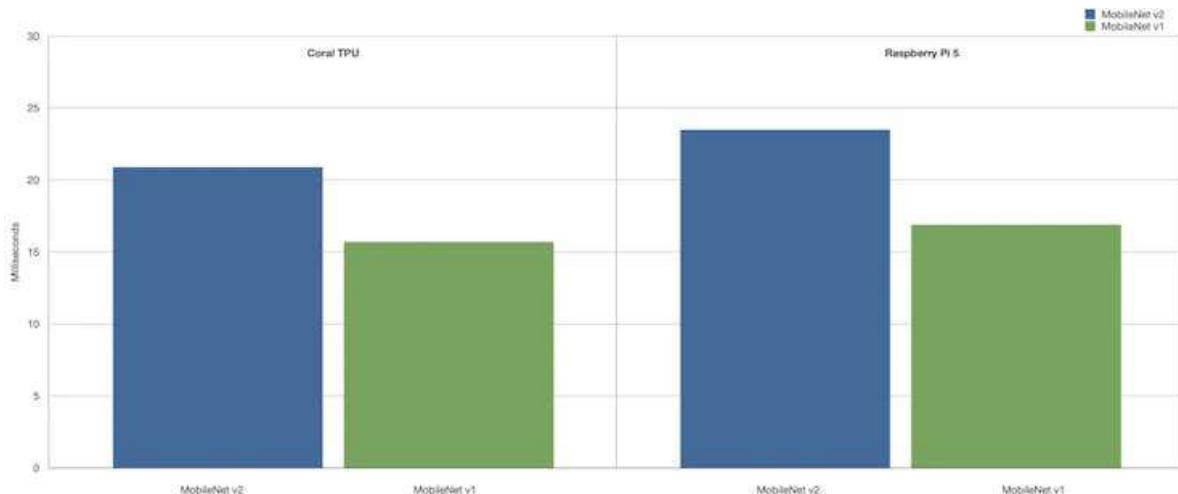
Ollama is designed to be efficient, scalable, and easy to use, making it a good option for deploying AI models such as Microsoft Phi, Google Gemma, Meta Llama, and more. We will integrate some of those models into projects using Python's ecosystem, exploring their potential in real-world scenarios (or at least point in this direction).

Setup

We could use any Raspberry Pi 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, 8GB, and 16GB of high-speed LPDDR4X SDRAM, with 8GB as our choice for running 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.

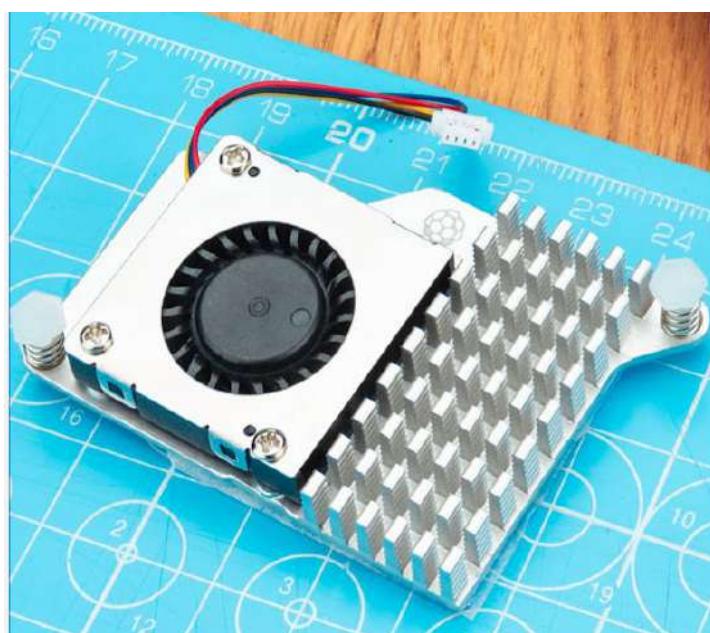
By the way, as [Alasdair Allan](#) discussed, inferencing directly on the Raspberry Pi 5 CPU—with no GPU acceleration—is now on par with the performance of the Coral TPU.



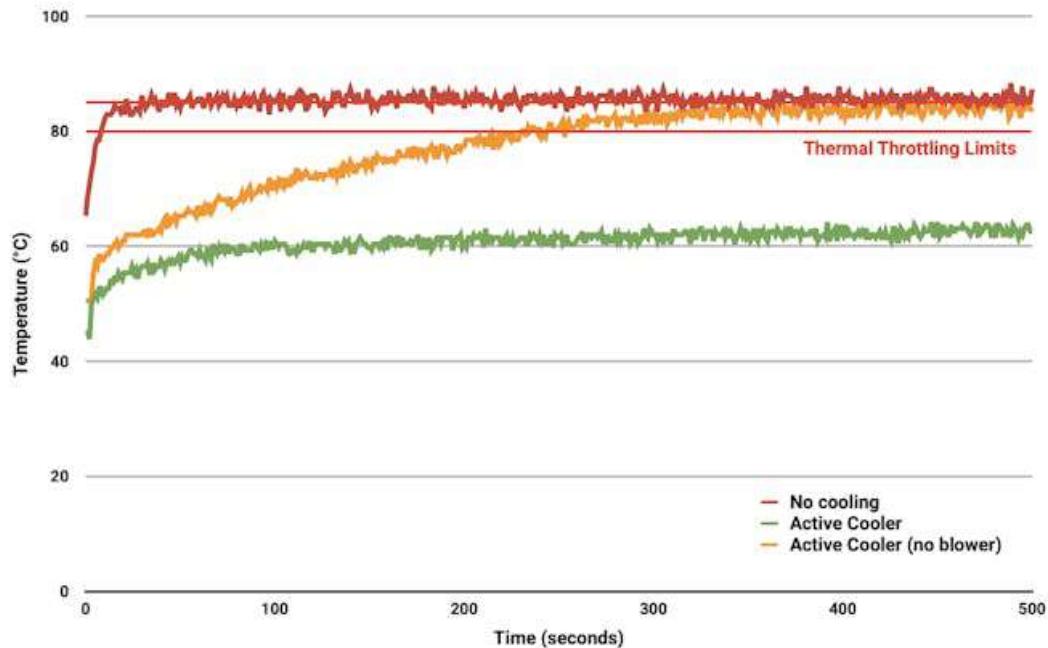
For more info, please see the complete article: [Benchmarking TensorFlow and TensorFlow Lite on Raspberry Pi 5](#).

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 is turned on; at 67.5°C, the fan speed is 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 media 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, which we call *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 data types, 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-3 and beyond (by OpenAI), Claude (by Anthropic), Gemini (by Google).

Open models, also known as open-source models, are AI models whose code, architecture, and, often, training data are publicly available. 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.

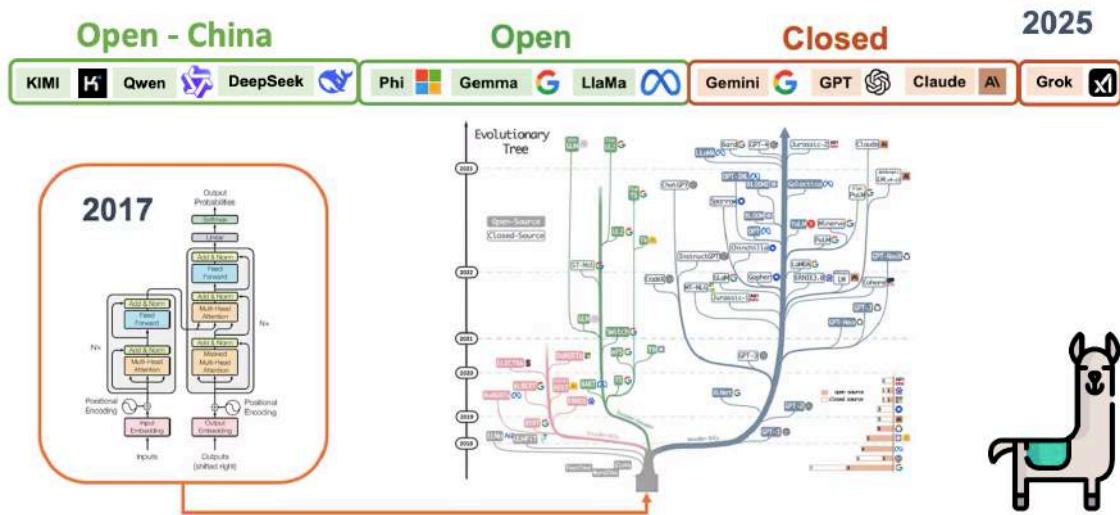


Figure 9: Adapted from <https://arxiv.org/pdf/2304.13712>

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).

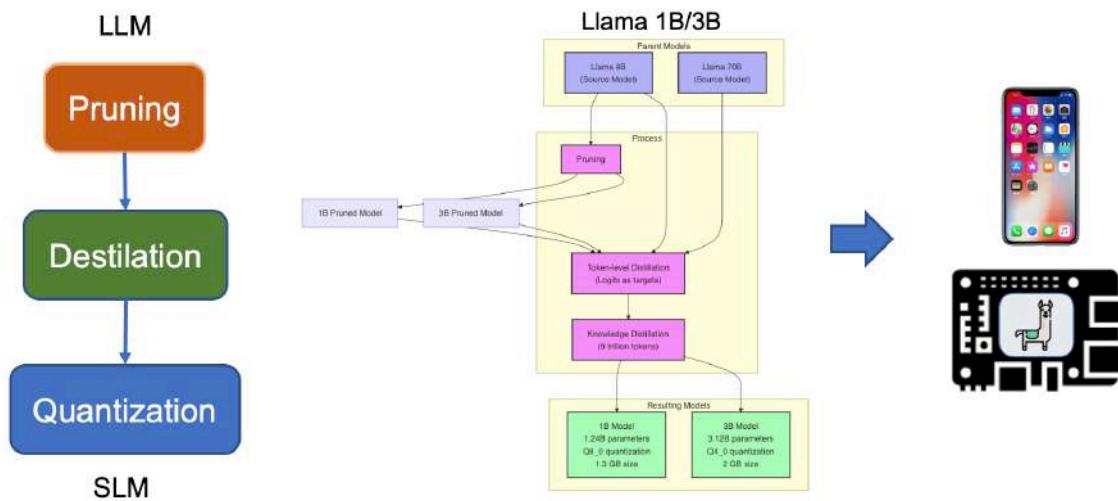
SLMs are compact versions of LLMs designed to run efficiently on resource-constrained devices such as smartphones, IoT devices, and single-board computers like the Raspberry Pi. These models are significantly smaller in size and computational requirements than their larger counterparts while still retaining 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 fewer than 5 to 8 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 10: ollama logo

The primary and most user-friendly tool for running Small Language Models (SLMs) directly on a Raspberry Pi, especially the Pi 5, is [Ollama](#). It is an open-source framework that allows us to install, manage, and run various SLMs (such as TinyLlama, smollm, Microsoft Phi, Google Gemma, Meta Llama, MoonDream, LLaVa, among others) locally on our Raspberry Pi for tasks such as text generation, image captioning, and translation.

Alternatives options:

[llama.cpp](#) and the Hugging Face [Transformers](#) library are well-supported for running SLMs on Raspberry Pi. [llama.cpp](#) is particularly efficient for running quantized models natively. At the same time, Hugging Face Transformers offers a broader range of models and tasks, which are best suited to smaller architectures due to hardware limitations.

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:

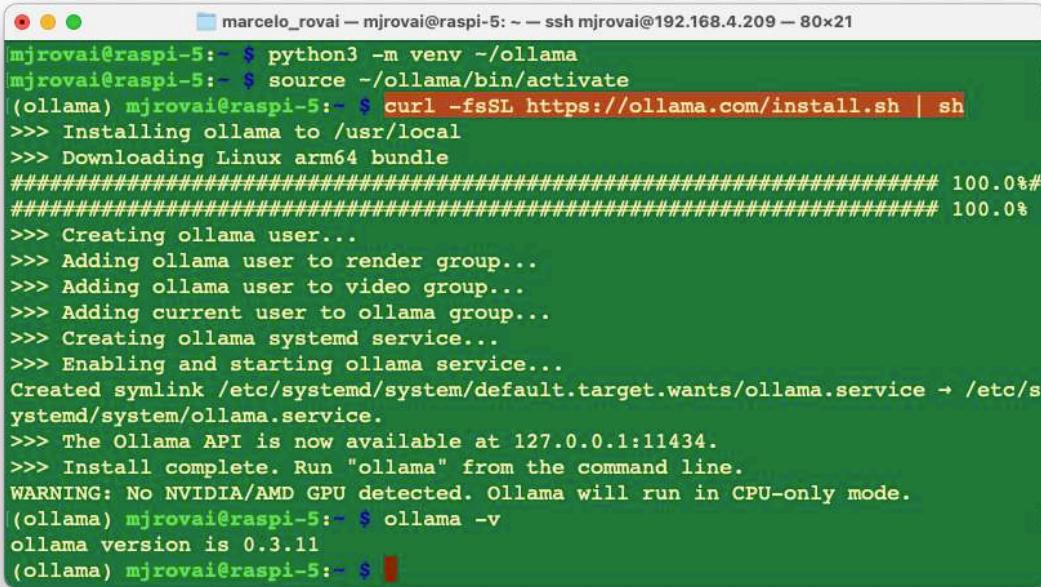
```
python3 -m venv ~/ollama
source ~/ollama/bin/activate
```

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.

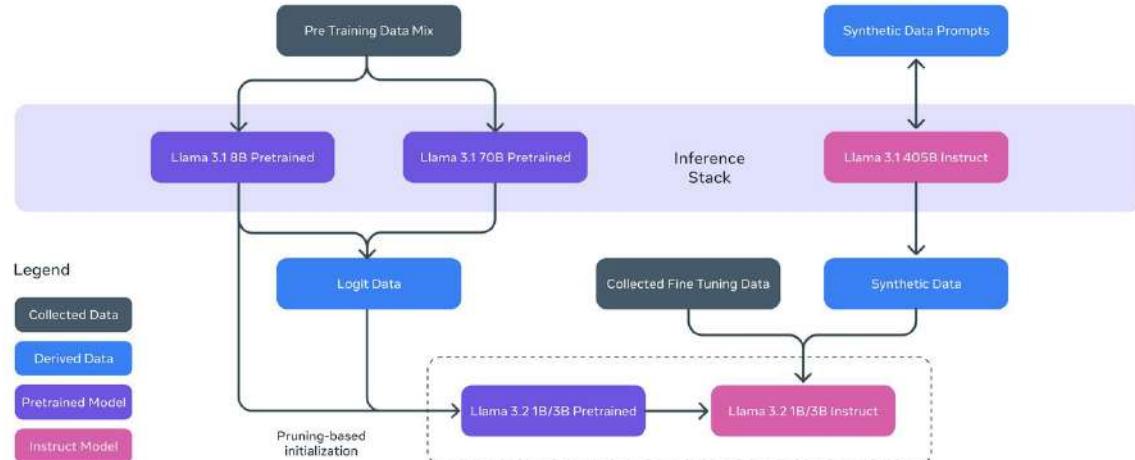
Meta Llama 3.2 1B/3B



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 4f659ale86d7... 100% [██████████] 485 B

verifying sha256 digest
writing manifest
success
>>> What is the capital of France?
The capital of France is Paris.

total duration:      2.620170326s
load duration:      39.947908ms
prompt eval count:   32 token(s)
prompt eval duration: 1.644773s
prompt eval rate:    19.46 tokens/s
eval count:          8 token(s)
eval duration:       889.941ms
eval rate:           8.99 tokens/s

```

Each metric gives insights into how the model processes inputs and generates outputs. Here's a breakdown of what each metric means:

- **Total Duration (2.620170326s):** 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

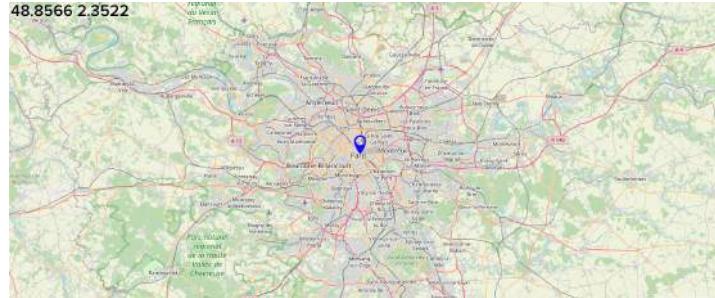
```
>>> What is the distance between Paris and Santiago, Chile?
```

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



Gemma

[Google Gemma](#), is a collection of lightweight, state-of-the-art open models built from the same technology that powers our Gemini models. Today, the Gemma family has the `Gemma3` and `Gemma3n` models.

Gemma3

We can, for example, install `gemma3:latest`, using `ollama run gemma3:latest`. This model has 4.3B parameters, with a context length of 8,192 and an embedding length of 2,560. A typical quantization schema is the Q4_K_M. This model has vision capabilities. Besides the 4B, we can also install the 1B parameter model.

Install and run the Model

```
ollama run gemma3:4b --verbose
```

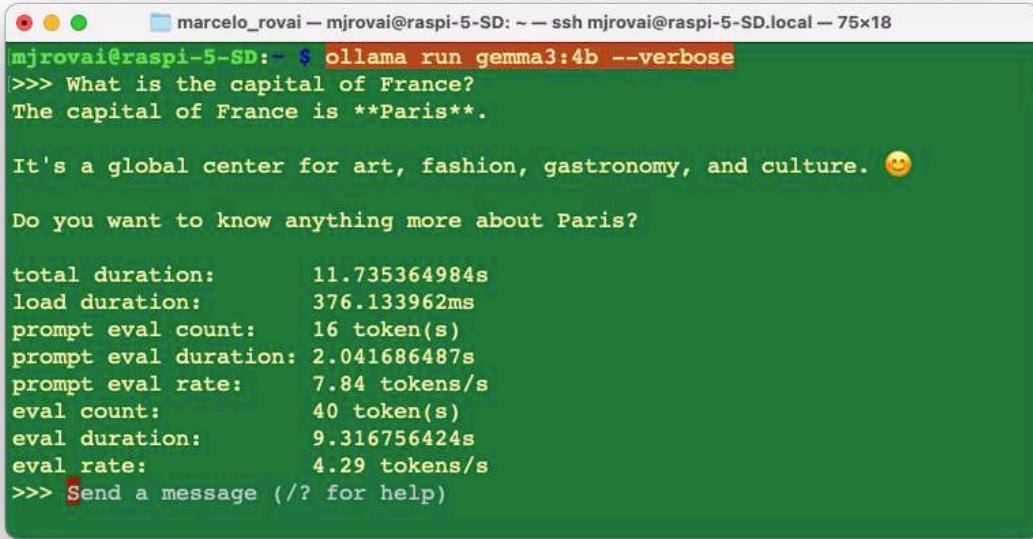
Running the model with the command before, we should have the Ollama prompt available for us to input a question and start chatting with the LLM model; for example,

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

Almost immediately, we get the correct answer:

The capital of France is **Paris**. It's a global center for art, fashion, gastronomy, and culture. Do you want to know anything more about Paris?

And its statistics.



A screenshot of a terminal window titled "marcelo_rovai — mjrovai@raspi-5-SD: ~ — ssh mjrovai@raspi-5-SD.local — 75x18". The terminal shows the following interaction:

```
mjrovai@raspi-5-SD:~$ ollama run gemma3:4b --verbose
>>> What is the capital of France?
The capital of France is **Paris**.

It's a global center for art, fashion, gastronomy, and culture. 😊

Do you want to know anything more about Paris?

total duration:      11.735364984s
load duration:      376.133962ms
prompt eval count:   16 token(s)
prompt eval duration: 2.041686487s
prompt eval rate:    7.84 tokens/s
eval count:          40 token(s)
eval duration:       9.316756424s
eval rate:           4.29 tokens/s
>>> Send a message (/? for help)
```

We can see that Gemma 3:4B has roughly the same performance as Lama 3.2:3B, despite having more parameters.

Other examples:

```
marcelo_rovai — mjrovai@raspi-5-SD: ~ — ssh mjrovai@raspi-5-SD.local — 70x11
>>> What is the distance between Paris and Santiago, Chile?
Okay, let's calculate the distance between Paris, France and
Santiago, Chile.

The approximate distance between Paris and Santiago, Chile is
**10,660 kilometers (6,629 miles)**.

That's a *long* trip!

Would you like me to break that down in terms of flight time or
suggest a route?
```

Also, a good response but less accurate than Llama3.2:3B.

```
marcelo_rovai — mjrovai@raspi-5-SD: ~ — ssh mjrovai@raspi-5-SD.local — 70x10
>>> what is the latitude and longitude of Paris?
Okay, here's the latitude and longitude for Paris, France:

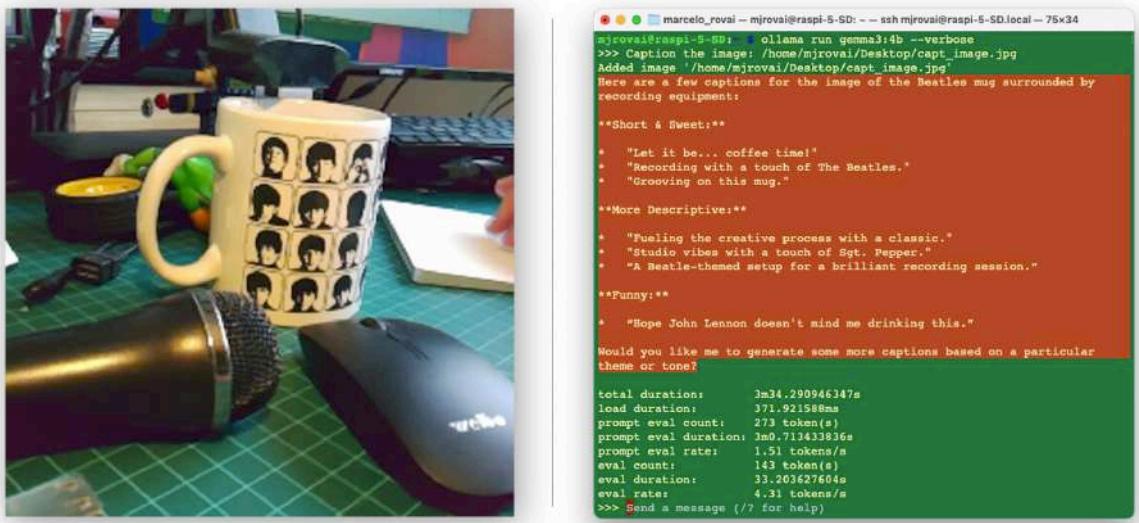
* **Latitude:** 48.8566° N (North)
* **Longitude:** 2.3522° E (East)

You can also find this information easily by searching "Paris
latitude longitude" on Google Maps or any other mapping service.

Do you want to know anything else about Paris's location?
```

A good and accurate answer (a little more verbose than the Llama answers).

An advantage of the Gemma3 models are their vision capability, for example, we can ask it to caption an image:



This is a very accurate model, but it still has high latency, as we can see in the example above (more than 3 minutes to caption the image).

The Gemma 3 1B size models are text only and don't support image input.

Gemma 3n

Gemma3 is a powerful, efficient open-source model that runs locally on phones, tablets, and laptops. The models are listed with parameter counts, such as **E2B** and **E4B**, that are *lower* than the total number of parameters contained in the models. The **E** prefix indicates these models can operate with a reduced set of Effective parameters. This reduced-parameter operation can be achieved using the flexible parameter technology built into Gemma 3n models, which helps them run efficiently on lower-resource devices.

The parameters in Gemma 3n models are divided into four main groups: text, visual, audio, and per-layer embedding (PLE) parameters. In the standard execution of the E2B model, over 5 billion parameters are loaded. However, by using parameter skipping and PLE caching, this model can be operated with an effective memory load of just under 2 billion (1.91B) parameters, as illustrated below:

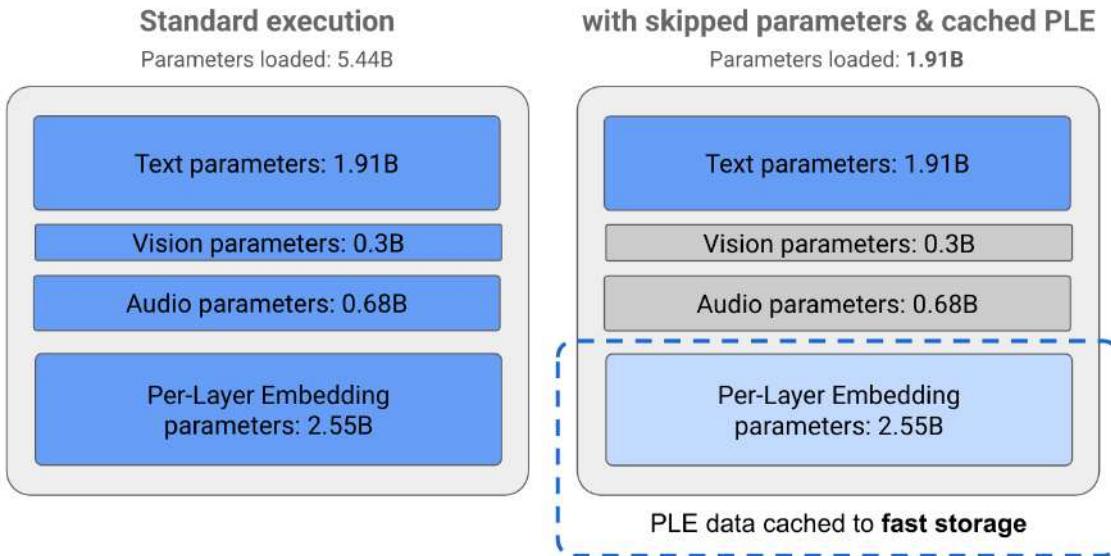


Figure 11: Immage from Google: Gemma 3n diagram of parameter usage

Once installed, (using `ollama run gemma3n:e2b`), inspecting the model we get:

- Architecture: gemma3n
- Parameters: 4.5B
- Context length 32768
- Embedding length 2048
- Quantization Q4_K_M
- Capabilities: completion only

When we run it, we can see that, despite having 4.5B parameters, it is faster than Llama3.3:3B.

```

marcelo_rovai — mjrovai@raspi-5-SD: ~ ssh mjrovai@raspi-5-SD.local — 77x13
mjrovai@raspi-5-SD: $ ollama run gemma3n:e2b --verbose
>>> What is the capital of France?
The capital of France is **Paris**.

total duration:      3.51189399s
load duration:       660.644566ms
prompt eval count:   16 token(s)
prompt eval duration: 1.327778409s
prompt eval rate:    12.05 tokens/s
eval count:          11 token(s)
eval duration:       1.521399316s
eval rate:           7.23 tokens/s

```

Microsoft Phi3.5 3.8B



Let's pull now the [Phi3.5](#), a 3.8B lightweight state-of-the-art open model by Microsoft. The model belongs to the Phi-3 model family and supports 128K token context length and the languages: Arabic, Chinese, Czech, Danish, Dutch, English, Finnish, French, German, Hebrew, Hungarian, Italian, Japanese, Korean, Norwegian, Polish, Portuguese, Russian, Spanish, Swedish, Thai, Turkish, and Ukrainian.

The model size, in terms of bytes, will depend on the specific quantization format used. The size can go from 2-bit quantization (`q2_k`) of 1.4 GB (higher performance/lower quality) to 16-bit quantization (`fp-16`) of 7.6 GB (lower performance/higher quality).

Let's run the 4-bit quantization (`Q4_0`), which will need 2.2 GB of RAM, with an intermediary trade-off regarding output quality and performance.

```
ollama run phi3.5:3.8b --verbose
```

You can use `run` or `pull` to download the model. What happens is that Ollama keeps note of the pulled models, and once the PHI3 does not exist, before running it, Ollama pulls it.

Let's enter with the same prompt used before:

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

The capital of France is Paris. It's extradites significant historical, cultural, and political importance to the country as well as being a major European city known for its art, fashion, gastronomy, and culture. Its influence extends beyond national borders, with millions of tourists visiting each year from around the globe. The Seine River flows through Paris before it reaches the broader English Channel at Le Havre. Moreover, France is one of Europe's leading economies with its capital playing a key role

...

The answer was very “verbose”, let's specify a better prompt:

The screenshot shows a terminal window with the following text:

```
>>> 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 in October 2024, the Llama2:3B or Gemma 3 are better choices. Try other models, depending on your needs.

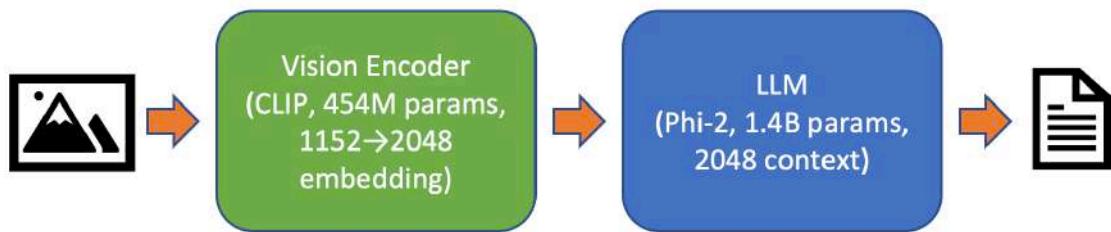
[Open LLM Leaderboard](#) can give you an idea about the best models in size, benchmark, license, etc.

The best model to use is the one fit for your specific necessity. Also, take into consideration that this field evolves with new models everyday.

MoonDream

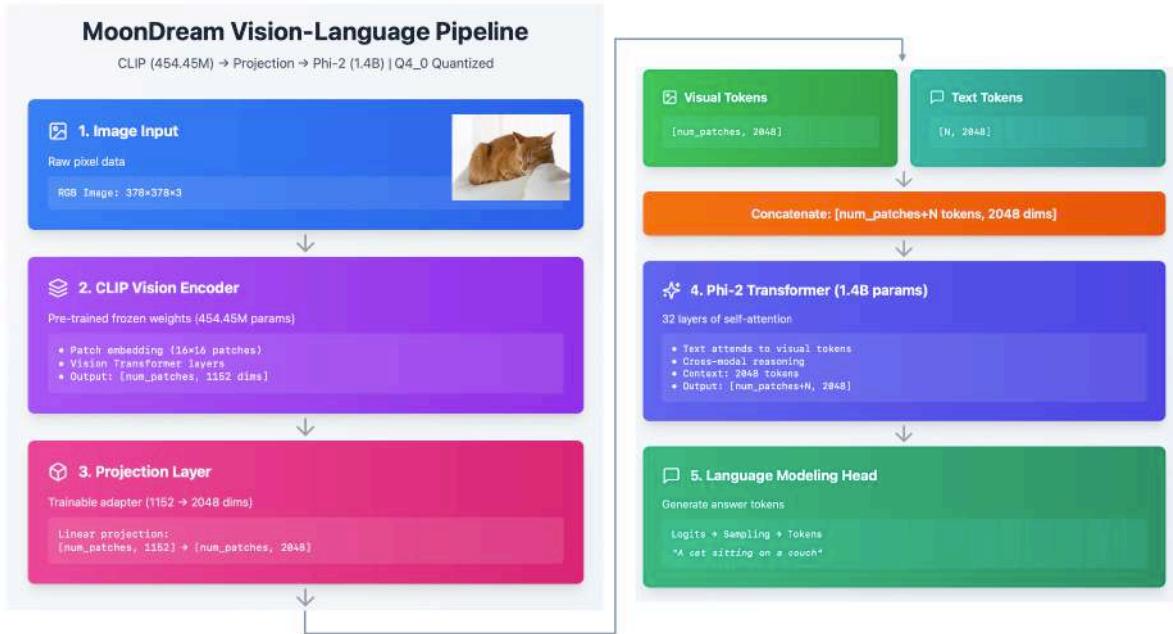


[Moondream](#) is an open-source visual language model that understands images using simple text prompts. It's fast and wildly capable. It is 3 to 4 times faster than the Gemma3, for example.



Moondream is a compact and efficient open-source vision-language model (VLM) designed to analyze images and generate descriptive text, answer questions, detect objects, count, point, and perform OCR—all through natural-language instructions, even on resource-limited devices like the Raspberry Pi or edge hardware.

Below is a diagram showing the flux from an image and its description:



Moondream can interpret images much like a human would, enabling tasks like:

- Captioning and describing images in detail
- Visual question answering (VQA) such as “What is the color of the shirt?”
- Object detection and pointing out locations
- Counting and contextual reasoning about visual scenes

```

marcelo_rovai — mjrovai@raspi-5-SD: ~ ssh mjrovai@raspi-5-SD.local — 87x18
(ollama) mjrovai@raspi-5-SD: ~ $ ollama run moondream --verbose
>>> describe the image: /home/mjrovai/Documents/OLLAMA/image_test.jpg
Added image '/home/mjrovai/Documents/OLLAMA/image_test.jpg'

The image features a man with glasses and a white beard, who is wearing a
collared shirt. He appears to be looking off into the distance or at something
out of frame. The room has several bookshelves filled with numerous books,
indicating that he might have an interest in reading or studying.

total duration:      54.36227837s
load duration:      63.257535ms
prompt eval count:   743 token(s)
prompt eval duration: 48.361595541s
prompt eval rate:    15.36 tokens/s
eval count:          62 token(s)
eval duration:       5.935988132s
eval rate:           10.44 tokens/s
>>> Send a message (/? for help)

marcelo_rovai — mjrovai@raspi-5-SD: ~ ssh mjrovai@raspi-5-SD.loca...
>>> what is the color of the shirt?
The man's shirt is white.

total duration:      1m15.308297242s
  
```

Image captured: image_test.jpg

Lava-Phi-3



Another multimodal model is the [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.

In terms of latency, it is a pair with Gemma3 and much slower than MoonDream

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
```

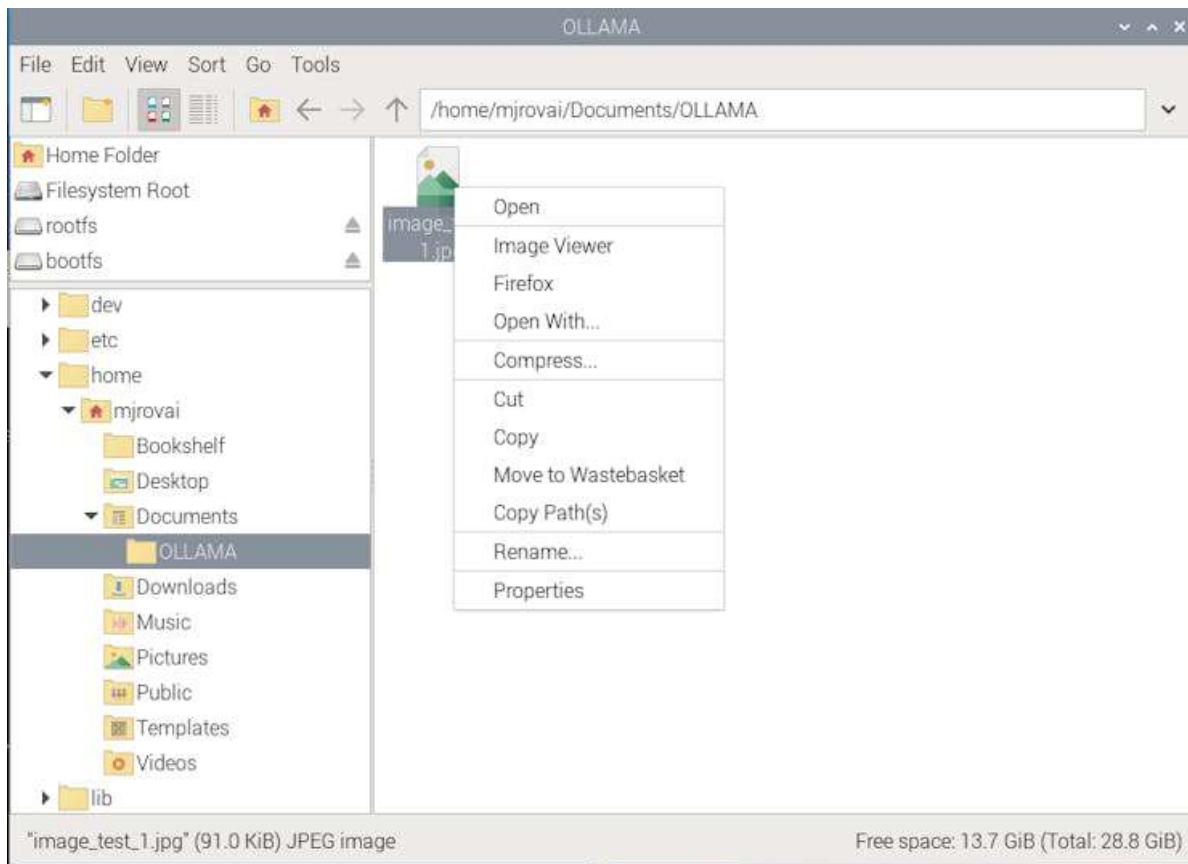
Let's download a 640x320 image from the internet, for example (Wikipedia: [Paris, France](#)):



Using FileZilla, for example, let's upload the image to the OLLAMA folder on 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 right-clicking the image.



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)

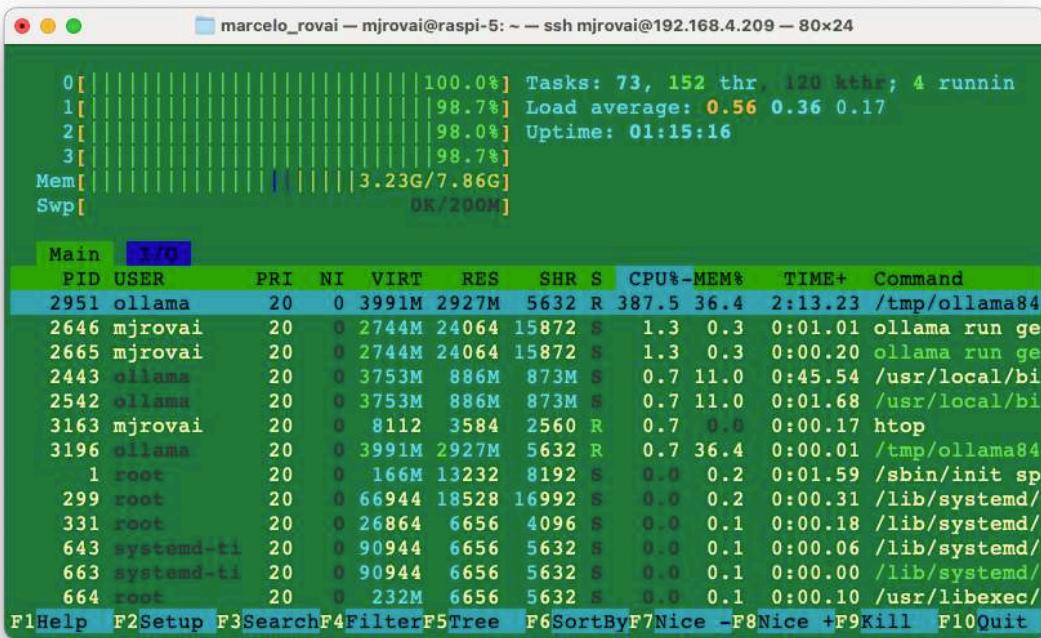
```

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:

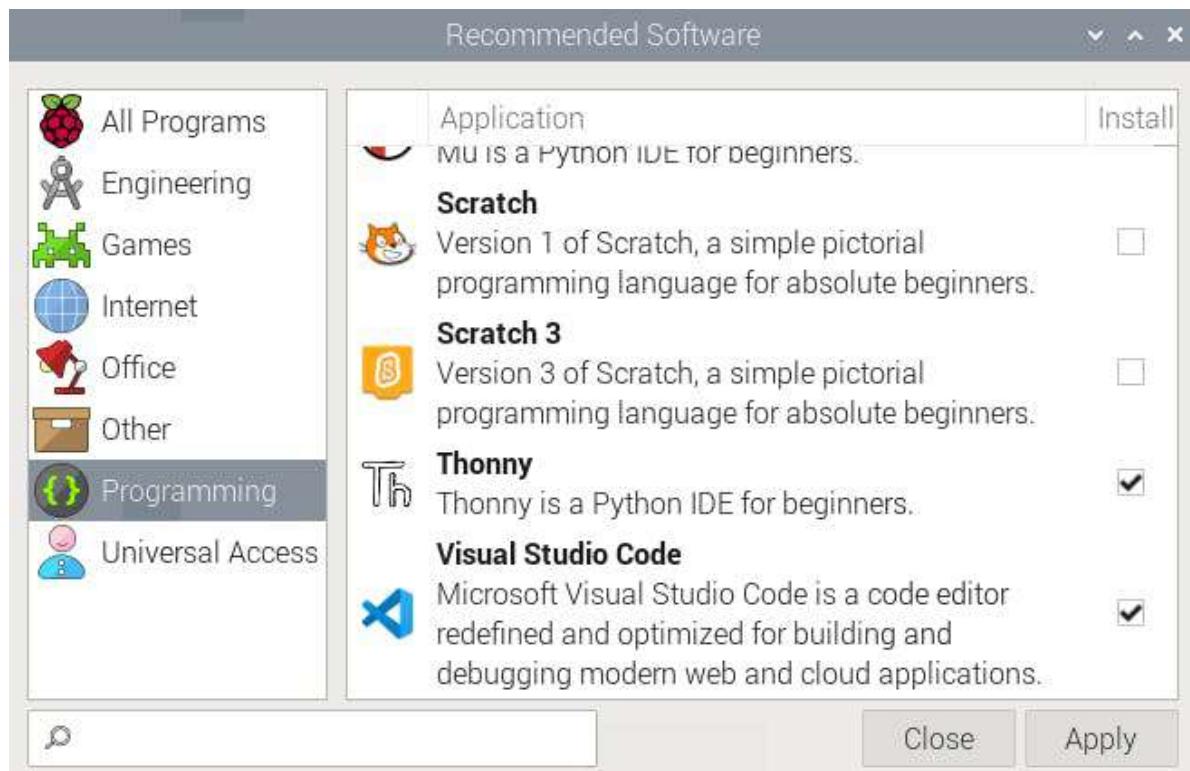
https://www.youtube.com/embed/_4K20tOsXK8

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 we run Raspberry Pi OS on a desktop, several options, such as Thonny and Geany, are already installed by default (accessed via [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 your option, 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
(ollama) 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 15:25:04.036 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/ollama/lib/python3.11/site-packages/jupyterlab
[I 2024-09-25 15:25:04.039 LabApp] JupyterLab application directory is /home/mjrovai/ollama/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=79a989d699951f61d357cdd5aa1146d350eaf3ed1471a422
[I 2024-09-25 15:25:04.085 ServerApp] http://127.0.0.1:8888/tree?token=79a989d699951f61d357cdd5aa1146d350eaf3ed1471a422
[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).
[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=79a989d699951f61d357cdd5aa1146d350eaf3ed1471a422
http://127.0.0.1:8888/tree?token=79a989d699951f61d357cdd5aa1146d350eaf3ed1471a422
[I 2024-09-25 15:25:04.088 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
models = ollama.list()
for model in models['models']:
    print(model['model'])

gemma3:4b
riven/smolvlm:latest
gemma3n:e2b
moondream:latest
llama3.2:3b

```

Same as we had on the terminal using `ollama show llama 3.2: 3b` , we can get model information:

```

info = ollama.show('llama3.2:3b')

print("Model/Format      : ", getattr(info, 'model', None))
print("Parameter Size   : ", getattr(info.details, 'parameter_size', None))
print("Quantization Level: ", getattr(info.details, 'quantization_level', None))
print("Family           : ", getattr(info.details, 'family', None))
print("Supported Capabilities: ", getattr(info, 'capabilities', None))
print("Date Modified (Local) : ", getattr(info, 'modified_at', None))
print("License Short     : ", (str(getattr(info, 'license', None)).split('\n')[0]))

print("Key Architecture Details:")

for key in [
    'general.architecture', 'general.finetune',
    'llama.context_length', 'llama.embedding_length', 'llama.block_count'
]:
    print(f" - {key}: {info.modelinfo.get(key)}")

```

As a result, we have:

```

Model/Format      : None
Parameter Size   : 3.2B
Quantization Level: Q4_K_M
Family           : llama
Supported Capabilities: ['completion', 'tools']
Date Modified (Local) : 2025-09-15 15:39:48.153510-03:00
License Short     : LLAMA 3.2 COMMUNITY LICENSE AGREEMENT
Key Architecture Details :
 - general.architecture: llama
 - general.finetune: Instruct
 - llama.context_length: 131072
 - llama.embedding_length: 3072
 - llama.block_count: 28

```

Ollama Generate

Let's repeat one of the questions that we did before, but now using `ollama.generate()` from the Ollama Python library. This API generates a response for the given prompt using 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.

```

response = ollama.generate(
    model="llama3.2:3b",
    prompt="Qual a capital do Brasil"
)
print(response['response'])

```

We get the response: A capital do Brasil é Brasília.

If you are running the code as a Python script, you should save it as, for example, test_ollama.py. You can run it in the IDE or run it directly in the terminal. Also, remember always to call the model and define it when running a stand-alone script.

```
python test_ollama.py
```

Let's print the full response now. As a result, we will have the model response in a JSON format:

```

GenerateResponse(model='llama3.2:3b', created_at='2025-10-14T18:30:43.877536633Z',
done=True, done_reason='stop', total_duration=28105898130, load_duration=23201685971,
prompt_eval_count=30, prompt_eval_duration=3364148131, eval_count=9,
eval_duration=1539216446, response='A capital do Brasil é Brasília.', thinking=None,
context=[128006, 9125, 128007, 271, 38766, 1303, 33025, 2696, 25, 6790, 220, 2366, 18,
271, 128009, 128006, 882, 128007, 271, 32129, 264, 6864, 656, 43025, 128009, 128006,
78191, 128007, 271, 32, 6864, 656, 43025, 4046, 62224, 76472, 13])

```

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.
 - A capital do Brasil é Brasília.
- **context:** the token IDs representing the input and context used by the model. Tokens are numerical representations of text that the language model uses for processing.

```
[128006, 9125, 128007, 271, 38766, 1303, 33025, 2696, 25, 6790, 220, 2366, 18, 271, 128009, 128006, 882, 128007, 32129, 264, 6864, 656, 43025, 128009, 128006, 78191, 128007, 271, 32, 6864, 656, 43025, 4046, 62224, 76472, 13]
```

The Performance Metrics:

- **total_duration:** The total time taken for the operation in nanoseconds. In this case, approximately 2.81 seconds.
- **load_duration:** The time taken to load the model or components in nanoseconds. About 0.23 seconds.

- **prompt_eval_duration:** The time taken to evaluate the prompt in nanoseconds. Around 0.33 seconds.
- **eval_count:** The number of tokens evaluated during the generation. Here, 9 tokens.
- **eval_duration:** The time taken for the model to generate the response in nanoseconds. Approximately 1.54 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{response['response']}")  
print(f"\n [INFO] Total Duration: {(response['total_duration'])/1e9:.2f} seconds")
```

Now, we got:

```
A capital do Brasil é Brasília.  
[INFO] Total Duration: 2.81 seconds
```

Streaming with ollama.generate()

To stream the output from `ollama.generate()` in Python, set `stream=True` and iterate over the generator to print each chunk as it’s produced. This enables real-time response streaming, similar to chat models.

```
stream = ollama.generate(  
    model='llama3.2:3b',  
    prompt='Tell me an interesting fact about Brazil',  
    stream=True  
)  
  
for chunk in stream:  
    print(chunk['response'], end='', flush=True)
```

Here's one:

Did you know that Brazil is home to the world's largest population of pink dolphins? The pink dolphin, also known as the boto, can be found in the Amazon and Paraná rivers, which run through the country. These majestic creatures are a popular tourist attraction and are considered an iconic symbol of Brazilian culture.

In fact, there are two species of pink dolphins: the freshwater boto (inhabiting the Amazon River) and the saltwater botolphus (found in the Paraná River). Both are endangered due to habitat loss, pollution, and other human activities. Conservation efforts are underway to protect these incredible animals and their habitats.

Brazil's unique biodiversity is truly fascinating!

- Each `chunk['response']` is a part of the generated text, streamed as it's created
- This allows responsive, real-time interaction in the terminal or UI.

This approach is ideal for long or complex generations, making the user experience feel faster and more interactive.

System Prompt

Add a `system` parameter to set overall instructions or behavior for the model (useful for role assignment and tone control):

```
response = ollama.generate(model='llama3.2:3b',
                            prompt='Tell about industry',
                            system='You are an expert on Brazil.',
                            stream=False)
print(response['response'])
```

Brazil is a rapidly developing country with a diverse and vibrant economy, driven by several key industries. Here's an overview:

****Main Industries:****

1. **Agriculture**: Agriculture is the backbone of Brazil's economy, accounting for approximately 10% of GDP. The country is the world's largest producer of soybeans, coffee, sugar, cotton, and cattle.
2. **Mining**: Mining is another significant sector, with Brazil being a major producer of iron ore, copper, gold, diamonds, and other minerals.
3. **Manufacturing**: Brazil has a growing manufacturing sector, particularly in industries such as automobiles, machinery, textiles, and food processing.
4. **Energy**: Energy is an increasingly important sector, with Brazil investing heavily in renewable energy sources like solar, wind, and hydroelectric power.
5. **Services**: The services sector, including finance, logistics, tourism, and IT, accounts for a significant share of the country's GDP.

****Key Sectors:****

1. **Soybean exports**: Brazil is the world's largest exporter of soybeans, accounting for over 40% of global production.
2. **Coffee exports**: Brazil is also a major coffee producer and exporter, with most of its production being Arabica beans.
3. **Iron ore exports**: Brazil is the second-largest iron ore producer in the world, with most of its output being exported to China, Japan, and South Korea.
4. **Automotive industry**: Brazil has a thriving automotive sector, with major manufacturers like Volkswagen, General Motors, and Fiat Chrysler Automobiles (FCA) operating in the country.

****Challenges:****

1. **Dependence on exports**: Brazil's economy is heavily dependent on exports, making it vulnerable to fluctuations in global demand and trade policies.
2. **Infrastructure challenges**: Many of Brazil's major industries face infrastructure challenges, including inadequate transportation networks, inefficient energy systems, and limited telecommunications connectivity.
3. **Environmental concerns**: Brazil's growing industries have raised environmental concerns, particularly regarding deforestation, pollution, and climate change.

****Opportunities:****

1. **Diversification**: Brazil has opportunities to diversify its economy by developing new industries, such as renewable energy, biotechnology, and technology.
2. **Regional integration**: Brazil is part of the Mercosur trading bloc, which offers opportunities for regional trade and investment.
3. **Innovation**: The country has a growing startup ecosystem and a vibrant cultural scene, providing opportunities for innovation and entrepreneurship.

Overall, Brazil's economy is characterized by its large agricultural sector, rich natural resources, and growing industries like manufacturing, energy, and services. However, the country also faces challenges related to dependence on exports, infrastructure, and environmental concerns.

Temperature and Sampling

Control creativity/randomness via `temperature`, and customize output style with extra settings like `top_p` and `num_ctx` (context window size)

```
response = ollama.generate(  
    model='llama3.2:3b',  
    prompt='Why the sky is blue?',  
    options={'temperature':0.1},  
    stream=True)  
for chunk in response:  
    print(chunk['response'], end='', flush=True)
```

The sky appears blue because of a phenomenon called Rayleigh scattering, named after the British physicist Lord Rayleigh. He discovered that when sunlight enters Earth's atmosphere, it encounters tiny molecules of gases such as nitrogen and oxygen.

Here's what happens:

1. **Sunlight enters the atmosphere**: When sunlight enters our atmosphere, it consists of a spectrum of colors, including all the colors of the visible light.
2. **Scattering occurs**: As sunlight travels through the atmosphere, it encounters tiny molecules of gases like nitrogen (N_2) and oxygen (O_2). These molecules scatter the light in all directions.
3. **Shorter wavelengths are scattered more**: The smaller wavelengths of light, such as blue and violet, are scattered more than the longer wavelengths, like red and orange. This is because the smaller molecules are more effective at scattering shorter wavelengths.
4. **Blue light dominates**: As a result of this scattering, the blue light is distributed throughout the atmosphere, reaching our eyes from all directions. This is why the sky appears blue during the daytime, especially in the direction of the sun.

Other factors can influence the color of the sky, such as:

- * **Dust and pollution**: Tiny particles in the air can scatter light in different ways, making the sky appear more hazy or gray.
- * **Water vapor**: High levels of water vapor in the atmosphere can cause the sky to appear more blue or greenish.
- * **Sun position**: The angle of the sun affects the amount of direct sunlight that reaches our eyes, which can change the apparent color of the sky.

So, to summarize, the sky appears blue because of the scattering of sunlight by tiny molecules in the Earth's atmosphere, with shorter wavelengths like blue being scattered more than longer wavelengths.

```
response = ollama.chat(  
    messages=[  
        {"role": "user",  
         "content": "Poetically describe Paris in one short sentence"},  
        ],  
        model='llama3.2:3b',  
        options={"temperature": 1.0} # Set. temp. to 1.0 for more creativity  
    )  
print(response['message']['content'])
```

Like a velvet-draped secret, Paris whispers ancient mysteries to the moonlit Seine, her sighing shadows weaving an eternal waltz of love and dreams.

Ollama.chat()

Another way to get our response is to use `ollama.chat()`, which generates the next message in a chat with a provided model. This is a streaming endpoint, so a series of responses will occur. Streaming can be disabled using `"stream": false`. The final response object will also include statistics and additional data from the request.

```
PROMPT_1 = 'What is the capital of France?'  
  
response = ollama.chat(model=MODEL, messages=[  
    {'role': 'user', 'content': PROMPT_1, }, ],)  
resp_1 = response['message']['content']  
print(f"\n{resp_1}")  
print(f"\n [INFO] Total Duration: {(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
```

The above code works with two prompts. Let’s include a conversation variable to really provide the chat with a memory:

```
# Initialize conversation history
conversation = []

# Function to chat with memory
def chat_with_memory(prompt, model=MODEL):
```

```

global conversation

# Add user message to conversation
conversation.append({"role": "user", "content": prompt})

# Generate response with conversation history
response = ollama.chat(
    model=MODEL,
    messages=conversation
)

# Add assistant's response to conversation history
conversation.append(response["message"])

# Return just the response text
return response["message"]["content"]

# Question
prompt = "Qual a capital do Brasil"
print(chat_with_memory(prompt))

# Ask a follow-up question that relies on memory
follow_up = "E a do Perú?"
print(chat_with_memory(follow_up))

```

A capital do Brasil é Brasília.
A capital do Peru é Lima.

Image Description:

As we did with the visual models and 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.

Let's capture an image from the Raspberry Pi camera and get the description, now using the MoonDream model:

```

import time
import numpy as np

```

```

import matplotlib.pyplot as plt
from picamera2 import Picamera2
from PIL import Image

def capture_image(image_path):
    # Initialize camera
    picam2 = Picamera2() # default is index 0

    # Configure the camera
    config = picam2.create_still_configuration(main={"size": (520, 520)})
    picam2.configure(config)
    picam2.start()

    # Wait for the camera to warm up
    time.sleep(2)

    # Capture image
    picam2.capture_file(image_path)
    print("Image captured: "+image_path)

    # Stop camera
    picam2.stop()
    picam2.close()

```

Using the above code, we can capture an image, which can be displayed with:

```

def show_image(image_path):
    img = Image.open(image_path)

    # Display the image
    plt.figure(figsize=(6, 6))
    plt.imshow(img)
    plt.title("Captured Image")
    plt.show()

```

Let's also create a function to describe the image:

```

def image_description(img_path, model):
    with open(img_path, 'rb') as file:
        response = ollama.chat(
            model=model,

```

```

messages=[
    {
        'role': 'user',
        'content': '''return the description of the image''',
        'images': [file.read()],
    },
],
options = {
    'temperature': 0,
}
)
return response

```

Now, let's put all together and capture an image from the camera:

```

IMG_PATH = "/home/mjrovai/Documents/OLLAMA/SST/capt_image.jpg"
MODEL = "moondream:latest"

capture_image(IMG_PATH)
show_image(IMG_PATH)
response = image_description(IMG_PATH, MODEL)
caption = response['message']['content']
print ("\n==> AI Response:", caption)
print(f"\n[INFO] ==> Total Duration: {response['total_duration']/1e9:.2f} seconds")

```

Pointing the camera at my table:



We got the description:

The image features a green table with various items on it. A white mug adorned with black faces is prominently displayed, and there are several other mugs scattered around the table as well. In addition to the mugs, there's also a microphone placed near them, suggesting that this might be an office or workspace setting where someone could enjoy their coffee while recording podcasts or audio content.

A computer keyboard can be seen in the background, indicating that it is likely connected to a computer for work purposes. A mouse and a cell phone are also present on the table, further emphasizing the technology-oriented nature of this scene.

```
[INFO] ==> Total Duration: 82.57 seconds
```

We can now change the `image_description` function to ask “Who are the faces in the mug?”.

The answer:

```
==> AI Response:  
The mug has a picture of the Beatles on it.
```

In the [Ollama_Python_Library Intro notebook](#), we can find the experiments using the Ollama Python library.

Calling Direct API

One alternative to running an SLM in Python using Ollama is to call the API directly. Let’s explore some advantages and disadvantages of both methods.

Python Library:

```
response = ollama.generate(  
    model=MODEL,  
    prompt=QUERY)  
result = response['response']
```

Direct API Calls:

```
import requests  
import json  
  
# Configuration  
OLLAMA_URL = "http://localhost:11434/api"  
MODEL = MODEL  
  
response = requests.post(  
    f"{OLLAMA_URL}/generate",  
    json={  
        "model": MODEL,  
        "prompt": QUERY,
```

```

        "stream": False
    }
)

response = requests.post(
    f"{OLLAMA_URL}/generate",
    json={"model": MODEL,
          "prompt": query,
          "stream": False}
)
result = response.json().get("response", "")

```

One clear advantage of the Python library is that it handles URL construction, request formatting, and response parsing.

Error Handling

Python Library:

- Raises specific exceptions (e.g., `ollama.ResponseError`, `ollama.RequestError`)
- Better typed error messages
- Automatically handles connection issues

Direct API Calls:

- We should manually check the `response.status_code`
- Generic HTTP errors
- Need to handle connection exceptions yourself

Connection Management

Python Library:

- Automatically detects Ollama instance (checks `OLLAMA_HOST` env variable or defaults to `localhost:11434`)
- Built-in connection pooling and retry logic
- Handles timeouts gracefully

Direct API Calls:

- We should specify the URL manually
- Must implement our own retry logic
- Need to manage connection pooling yourself

Features & Functionality

Python Library:

- Clean access to all Ollama features (generate, chat, embeddings, list models, pull, etc.)
- Streaming is simple: `for chunk in ollama.generate(..., stream=True)`
- Type hints and better IDE support

Direct API Calls:

- Access to any API endpoint, including undocumented ones
- Full control over request headers, timeouts, etc.
- Can use any HTTP library (requests, httpx, urllib, etc.)

Dependencies

Python Library:

```
pip install ollama
```

- Adds a dependency to our project
- Depends on `httpx` under the hood

Direct API Calls:

```
pip install requests
```

- We choose our HTTP library
- More lightweight if we only need basic functionality

Advanced Features

Python Library:

```
# Streaming
for chunk in ollama.generate(model=MODEL, prompt=query, stream=True):
    print(chunk['response'], end='')

# Chat history
response = ollama.chat(
    model=MODEL,
    messages=[
```

```

        {'role': 'user', 'content': 'Hello!'},
        {'role': 'assistant', 'content': 'Hi there!'},
        {'role': 'user', 'content': 'How are you?'}
```

]

)

List models

models = ollama.list()

Pull models

ollama.pull('llama3.2:3b')

Direct API Calls:

- Require us to implement all these patterns manually
- More boilerplate code

When to Use Each?

Use the Python Library when:

- Building standard applications
- Need cleaner, more maintainable code
- Need good error handling out of the box
- Need streaming support
- Okay with adding a dependency

Use Direct API Calls when:

- Need fine-grained control over HTTP requests
- Are working in a constrained environment
- Need to customize timeouts, headers, or proxies
- Need to minimize dependencies
- Are debugging API issues
- Need to access experimental/undocumented endpoints

Performance Difference

Minimal difference in practice! The Python library uses `httpx` which is comparable to `requests`. Both make the same underlying HTTP calls to Ollama.

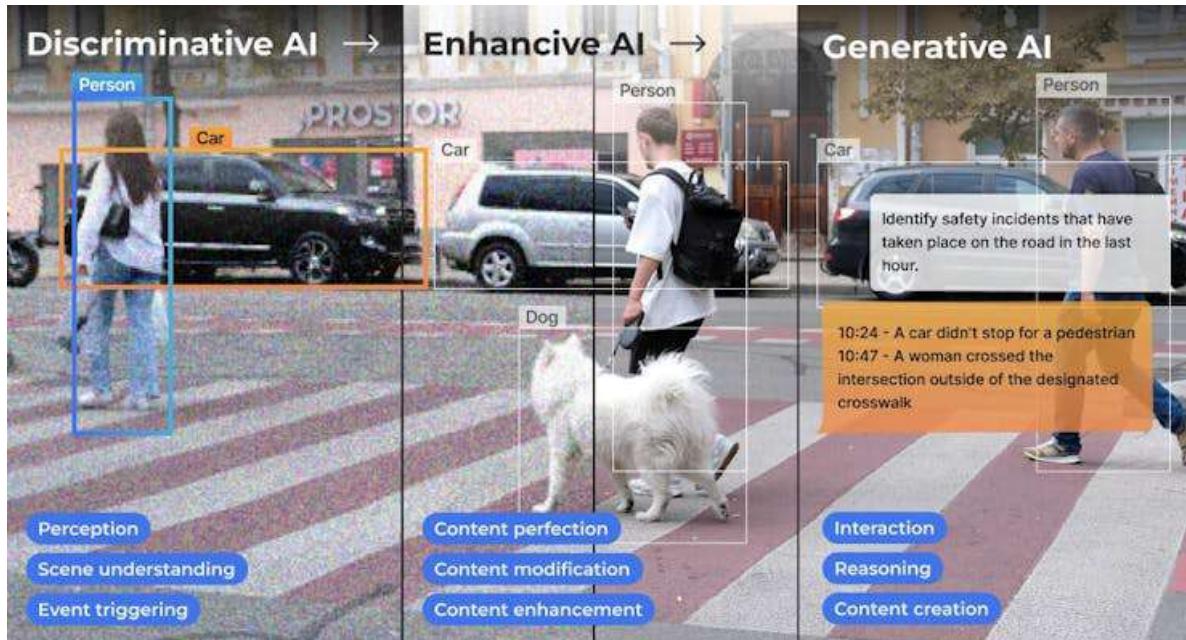
Bottom line: For most use cases, the Python library is the better choice due to its simplicity and built-in features. Use direct API calls only when you need specific control or have constraints that prevent adding the dependency.

Going Further

The small LLM models tested worked well at the edge, both with text and with images, but, of course, the last one had high latency. A combination of specific, dedicated models can lead to better results; for example, in real cases, an Object Detection model (such as YOLO) can provide a general description and count of objects in an image, which, 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 chapter 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 MoonDream, 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

- [Ollama_Python_Library Intro notebook](#)

SLM: Basic Optimization Techniques



Figure 12: *DALL · E prompt - I am writing a tutorial using Raspberry Pi. I am talking about Optimization Techniques, such as Function Calling and RAG, using Ollama and SLMs. I want a landscape-format image for the tutorial cover (without title). Should be a cartoon styled in 50s*

Introduction

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 appears coherent but lacks grounding 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 and more common techniques for enhancing LLM (and SLM) performance and efficiency are Function (or Tool) Calling, Prompt engineering, Fine-tuning, and Retrieval-Augmented Generation (RAG).

- **Function (Tool) calling** allows models to perform actions beyond generating text. By integrating with external functions or APIs, SLMs can access real-time data, automate tasks, and perform precise calculations—addressing the reliability issues that arise from the model’s limitations in mathematical operations.
- **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 a powerful approach that’s ideal for resource-constrained edge devices. This method combines the knowledge embedded in pre-trained models with the ability to access external, up-to-date information without requiring fine-tuning. By retrieving relevant data from a local knowledge base, RAG significantly enhances accuracy and reduces hallucinations—all without the computational overhead of model retraining.
- **Fine-tuning**, while more resource-intensive, offers a way to specialize LLMs for specific domains or tasks. This process involves further training the model on carefully curated datasets, allowing it to adapt its vast general knowledge to particular applications. Fine-tuning can lead to substantial performance improvements, especially in specialized fields or for unique use cases.

In this chapter, we’ll start focusing on two techniques that are particularly well-suited for edge devices like the Raspberry Pi: **Function Calling** and **RAG**.

We will learn more in detail about optimization techniques for SLMs, in the chapter:
[Advancing EdgeAI: Beyond Basic SLMs](#)

Function Calling Introduction

So far, we can see that, with the model’s (“response”) answer to a variable, we can efficiently work with it and integrate it into real-world projects. However, a big problem is that the model can respond differently to the same prompt. Let’s say, as in the last examples, that we want the model’s response to be 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 the OpenAI API.

But what exactly is “function calling”?

In modern artificial intelligence, function calling with Large Language Models (LLMs) allows these models to perform actions beyond generating text. By integrating with external functions or APIs, LLMs can access real-time data, automate tasks, and interact with various systems.

For instance, instead of merely responding to a weather query, 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, making it a powerful tool for driving workflows and automating processes, thereby 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>

And on this link: [HuggingFace Function Calling](#)

Using the SLM for calculations

Let’s do a simple calculation in Python:

123456*123456

The result would be: 15,241,383,936. No issues on it, but let’s ask a SLM to do the same simple task:

```
import ollama

response = ollama.chat(
    model='llama3.2:3B',
    messages=[{
        "role": "user",
```

```
        "content": "What is 123456 multiplied by 123456? Only give me the answer"
    ],
    options={"temperature": 0}
)
```

Examining the response: `response.message.content`, we would get: 51,131,441,376, what is completely wrong!

This is a **fundamental limitation of LLMs** - they're not calculators. Here's why the answer is wrong:

Why LLMs Fail at Math

0.0.0.0.1 * 1. LLMs Predict Text, They Don't Calculate

- LLMs work by predicting the next most likely token based on patterns in training data
- They don't perform actual arithmetic operations
- They're essentially "guessing" what a plausible answer looks like

0.0.0.0.2 * 2. Tokenization Issues

Numbers are broken into tokens in ways that don't align with mathematical operations:

```
"123456" might tokenize as: ["123", "456"] or ["12", "34", "56"]
```

This makes it nearly impossible for the model to "see" the actual numbers properly for computation.

0.0.0.0.3 * 3. Pattern Matching vs. Computation

- The model has seen similar multiplication problems in training
- It tries to recall patterns rather than compute
- For simple problems (2×3), it might seem to work because it memorizes common facts
- For larger numbers (123456×123456), it has no memorized pattern to fall back on

0.0.0.0.4 * 4. The Correct Answer

```
123456 × 123456 = 15,241,383,936
```

The LLM will likely give something that "looks" like a big number but is mathematically incorrect.

The Solution: Function Calling / Tool Use

The pattern is:

1. Use the LLM for understanding intent (classification)
2. Use Python for actual computation (the `multiply()` function)

```
def multiply(a, b):
    """Actual computation - always correct"""
    result = a * b
    return f"The product of {a} and {b} is {result}."
```

Why Even Temperature=0 Doesn't Help

Setting `temperature=0` makes the output **deterministic** (same input → same output), but it doesn't make it **correct**. The model will confidently give the same wrong answer every time.

LLM/SLM Math Performance by Problem Type

Problem Type	LLM Accuracy	Why
$2 + 2$	~99%	Memorized in training
$47 + 89$	~70%	Some pattern recognition
123×456	~30%	Struggles with multi-digit
123456×123456	~0%	No chance without computation
Calculate 15% tip on \$47.83	~40%	Multi-step reasoning fails

Best Practices

```
# DON'T: Ask LLM/SLM to calculate directly
response = ollama.chat(
    model='llama3.2:3b',
    messages=[{"role": "user", "content": "What is 123456 * 123456?"}]
)

# DO: Use LLM to understand, Python to compute
classification = ask_ollama_for_classification("What is 123456 * 123456?")
if classification["type"] == "multiplication":
```

```
result = multiply(123456, 123456) # Python does the math
```

Bottom line: Use LLMs for natural language understanding and intent classification, but delegate actual computations to proper tools/functions. This is the core principle behind **tool use** and **function calling** in modern LLM applications!

Function Calling Solution for Calculations

Define the Tool (Function Schema)

```
multiply_tool = {  
    "type": "function",  
    "function": {  
        "name": "multiply_numbers",  
        "description": "Multiply two numbers together",  
        "parameters": {  
            "type": "object",  
            "required": ["a", "b"],  
            "properties": {  
                "a": {"type": "number", "description": "First number"},  
                "b": {"type": "number", "description": "Second number"}  
            }  
        }  
    }  
}
```

Implement the Function

```
def multiply_numbers(a, b):  
    # Convert to int or float as needed  
    a = float(a)  
    b = float(b)  
    return {"result": a * b}
```

Now, let's create a function to handle the user query and calling for the tool when needed.

```
def answer_query(QUERY):  
    response = ollama.chat(
```

```

        'llama3.2:3B',
        messages=[{"role": "user", "content": QUERY}],
        tools=[multiply_tool]
    )

# Check if the model wants to call the tool
if response.message.tool_calls:
    for tool in response.message.tool_calls:
        if tool.function.name == "multiply_numbers":
            # Ensure arguments are passed as numbers
            result = multiply_numbers(**tool.function.arguments)
            print(f"Result: {result['result']:.2f}")
        else:
            print(f"It is not a Multiplication")

```

Note the line `tools=[multiply_tool]`, now as a part of the ollama's calling.

And run the function, we will get the correct answer.

```
answer_query("What is 123456 multiplied by 123456?")
```

Result: 15,241,383,936.00

Great! And now, can I use the same code to answer general questions? Let's test it:

```
answer_query("What is the capital of Brazil?")
```

Result: 1,000,000.00

The result is wrong. So, the above approach works fine for using the tool, but to answer it correctly (even without a tool), we should implement an “agentic approach”, which is a subject for later (See the Chapter: [Advancing EdgeAI: Beyond Basic SLMs](#))

Project: Calculating Distances

Suppose we want an SLM to return the distance in km from the capital city of the country specified by the user to the user's current location. We can see that the first is not so simple:

it is not always enough to enter only the country's name; the SLM can also give us a different (and incorrect) answer every time.

```
marcelo_rovai — mjrovai@raspi-5-SD: ~ ssh mjrovai@raspi-5-SD.local — 80x22
(ollama) mjrovai@raspi-5-SD: ~ $ ollama run llama3.2:3B
>>> How far in kilometers, Santiago de Chile is from Paris?
The distance between Santiago de Chile and Paris is approximately 15,141
km.

>>> And from Bogotá?
The distance between Santiago de Chile and Bogotá is approximately 5,144
km.

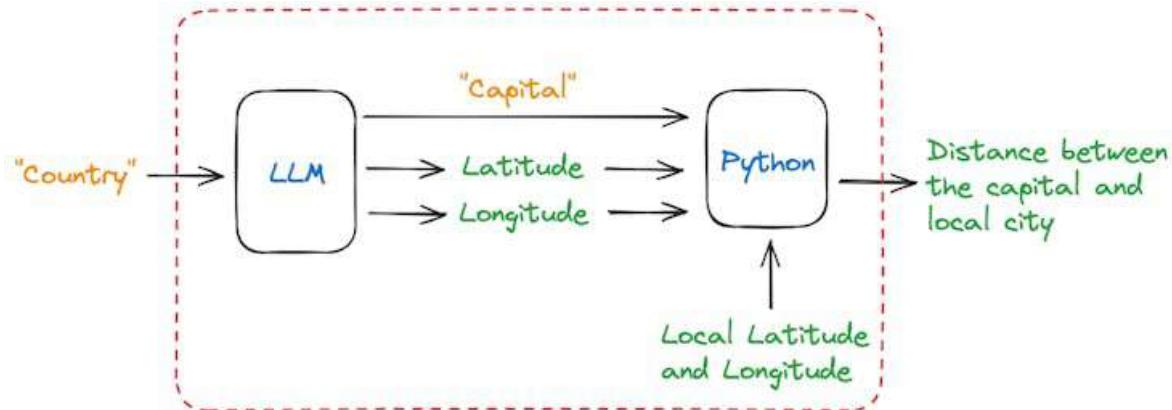
>>> And from the capital of US?
I think you meant "from Washington D.C." The distance between Santiago de
Chile and Washington D.C. is approximately 12,461 km.

>>> And from the capital of Colombia?
Since Bogotá is already in Colombia, I'll assume you're asking for the
distance between Santiago de Chile and Bogotá again.

The distance between Santiago de Chile and Bogotá is approximately 5,144
km.

>>> Send a message (/? for help)
```

OK, for trying to mitigate it, let's 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 should return the capital city's name (as

a string) and its latitude and longitude (as floats). Using those coordinates, we can use a simple Python library ([haversine](#)) to compute the great-circle distance between the two latitude/longitude points.

The idea of this project is to demonstrate a combination of language model interaction (IA) and geospatial calculations using the Haversine formula (traditional computing).

First, let us install the *Haversine* library:

```
pip install haversine
```

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:

Importing Libraries

```
import time
from haversine import haversine
from ollama import chat
```

Basic Variables and Model

```
MODEL = 'llama3.2:3B'      # The name of the model to be used
mylat = -33.33              # Latitude of Santiago de Chile
mylon = -70.51              # Longitude of Santiago de Chile
```

- **MODEL**: Specifies the model being used, which is, in this example, the Llama3.2.
- **mylat and mylon**: Coordinates of Santiago de Chile, used as the starting point for the distance calculation.

Defining a Python Function That Acts as a Tool

```
def calc_distance(lat, lon, city):
    """Compute distance and print a descriptive message."""
    distance = haversine((mylat, mylon), (lat, lon), unit="km")
    msg = f"\nSantiago de Chile is about {int(round(distance, -1))} I am running a \
few minutes late; my previous meeting is running over.kilometers away from {city}."
    return {"city": city, "distance_km": int(round(distance, -1)), "message": msg}
```

This is the real Python function that Ollama will be allowed to call. It performs the following steps:

- Takes latitude, longitude, and city name as input arguments.
- Uses the `haversine`

library to calculate the distance from Santiago to the target city.

- Returns a JSON-like dictionary containing the computed distance and a human-readable text summary.

In Ollama's terminology, this is a tool — a callable external function that the LLM may invoke automatically

Declaring the tool descriptor (schema)

```
tools = [
    {
        "type": "function",
        "function": {
            "name": "calc_distance",
            "description": "Calculates the distance from Santiago, Chile to a \
given city's coordinates.",
            "parameters": {
                "type": "object",
                "properties": {
                    "lat": {"type": "number", "description": "Latitude of the city"},
                    "lon": {"type": "number", "description": "Longitude of the city"},
                    "city": {"type": "string", "description": "Name of the city"}
                },
                "required": ["lat", "lon", "city"]
            }
        }
    }
]
```

This JSON object describes the metadata and input schema of the tool so that the LLM knows:

- Name: which function to call.
- Description: what purpose it serves.
- Parameters: input argument types and their descriptions. This schema mirrors the OpenAI function-calling format and is fully supported in Ollama 0.4 .

Defining tools this way allows Ollama to validate arguments before sending a call request back.

Asking the Model to Use the Tool

```
response = chat(
    model=MODEL,
    messages=[{
        "role": "user",
        "content": f"Find the decimal latitude and longitude of the capital of \

```

```

I am running a few minutes late; my previous meeting is running over. running a few\
minutes late; my previous meeting is running over.

    {country},"

        " then use the calc_distance tool to determine how far it is from \
Santiago de Chile."
    ],
    tools=tools
)

```

- The `chat()` function is called with the chosen model, a message, and the `tools` list.
- The prompt instructs the model first to identify the capital and its coordinates, and then invoke the tool (`calc_distance`) with those values.
- Ollama returns a structured response that may include a `tool_calls` section, indicating which tool to execute.

Executing the Model's Tool Call

For example, if `country = "Colombia"`, the model will return as the result:

```

ChatResponse(model='llama3.2:3B', created_at='2025-10-17T18:27:49.842521004Z', done=True, done_reason='stop', total_duration=9196115725, load_duration=207209859, prompt_eval_count=223, prompt_eval_duration=2668307535, eval_count=32, eval_duration=6319895710, message=Message(role='assistant', content='', thinking=None, images=None, tool_name=None, tool_calls=[ToolCall(function=Function(name='calc_distance', arguments={'city': 'Bogota', 'lat': 4.6583, 'lon': -74.0961}))]))

```

Where the `arguments` are the expected response in JSON format.

To get and display the result to the user, we should iterate over all `tool_calls`, extract the JSON arguments (lat, lon, city), and execute the local Python function using them. To finish it, we should print a human-readable `message` (e.g., “Santiago de Chile is about X kilometers away from ”CITY”).

So, let's first get the name of the city and the coordinates with:

```

for call in response.message.tool_calls:
    raw_args = call["function"]["arguments"]

    city = raw_args['city']
    lat = float(raw_args['lat'])
    lon = float(raw_args['lon'])

```

Now, we can calculate and print the distance, using `haversine()`:

```

distance = haversine((mylat, mylon), (lat, lon), unit='km')
print(f"Santiago de Chile is about {int(round(distance, -1))}:
      kilometers away from {city}.")

```

In this case:

```
Santiago de Chile is about 4,240 kilometers away from Bogota.
```

NOTE: Sometimes the model returns parameter names that differ from what your function expects. Specifically, Ollama occasionally returns argument objects like:

```
{"lat1": -33.33, "lon1": -70.51, "lat2": 48.8566, "lon2": 2.3522, "city":
"Paris"}
```

Instead of the schema-defined keys (`lat`, `lon`, `city`).

This happens because some LLMs (such as Llama 3.2 and Qwen 3) attempt to be “helpful” by naming coordinates explicitly—`lat1`/`lon1` for origin and `lat2`/`lon2` for destination—even when the schema only defines `lat`/`lon`.

To handle this, optionally a mapping-correction step can be added after decoding the `tocall` arguments.

```

if hasattr(response.message, "tool_calls") and response.message.tool_calls:
    for call in response.message.tool_calls:
        if call["function"]["name"] == "calc_distance":
            raw_args = call["function"]["arguments"]

            # Decode JSON if necessary
            args = json.loads(raw_args) if isinstance(raw_args, str) else raw_args

            # Normalize key names
            if "lat1" in args or "lat2" in args:
                args["lat"] = args.get("lat2") or args.get("lat1")
                args["lon"] = args.get("lon2") or args.get("lon1")
            if "latitude" in args:
                args["lat"] = args["latitude"]
            if "longitude" in args:
                args["lon"] = args["longitude"]
            args = {k: v for k, v in args.items() if k in ("lat", "lon", "city")}

            # Convert numbers
            args["lat"] = float(args["lat"])

```

```

args["lon"] = float(args["lon"])

result = calc_distance(**args)
print(result["message"])

```

Timing and Diagnostic Output

```

elapsed = time.perf_counter() - start
print(f"[INFO] ==> Model {MODEL} : {elapsed:.1f} s")

```

This records how long the operation took from prompt submission to tool execution, useful for benchmarking response performance.

Example Usage

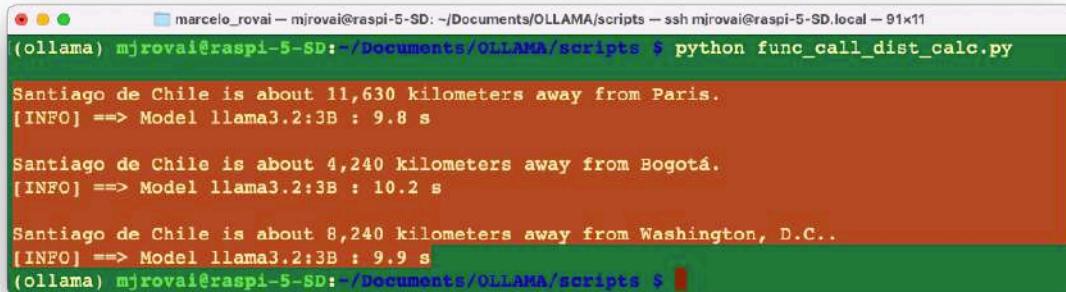
If we enter different countries, for example, France, Colombia, and the United States, We can note that we always receive the same structured information:

```

ask_and_measure("France")
ask_and_measure("Colombia")
ask_and_measure("United States")

```

If you run the code as a script, the result will be printed on the terminal:



```

marcelo_rovai@raspi-5-SD:~/Documents/OLLAMA/scripts$ ssh mirovai@raspi-5-SD.local -t python func_call_dist_calc.py
(ollama) mirovai@raspi-5-SD:~/Documents/OLLAMA/scripts$ python func_call_dist_calc.py
Santiago de Chile is about 11,630 kilometers away from Paris.
[INFO] ==> Model llama3.2:3B : 9.8 s

Santiago de Chile is about 4,240 kilometers away from Bogotá.
[INFO] ==> Model llama3.2:3B : 10.2 s

Santiago de Chile is about 8,240 kilometers away from Washington, D.C..
[INFO] ==> Model llama3.2:3B : 9.9 s
(ollama) mirovai@raspi-5-SD:~/Documents/OLLAMA/scripts$ 

```

And the calculations are pretty good!



The complete script can be found at: [func_call_dist_calc.py](#) and on the [10-Ollama_Function_Calling_notebook](#).

Running with other models

The models that will run with the described approach are the ones that can handle `tools`. For example, Gemma 3 and 3n will not work.

An alternative is to use the Pydantic library to serialize the schema using `model_json_schema()`. Using the Pydantic library, models as Gemma can also be used, as explored in the:

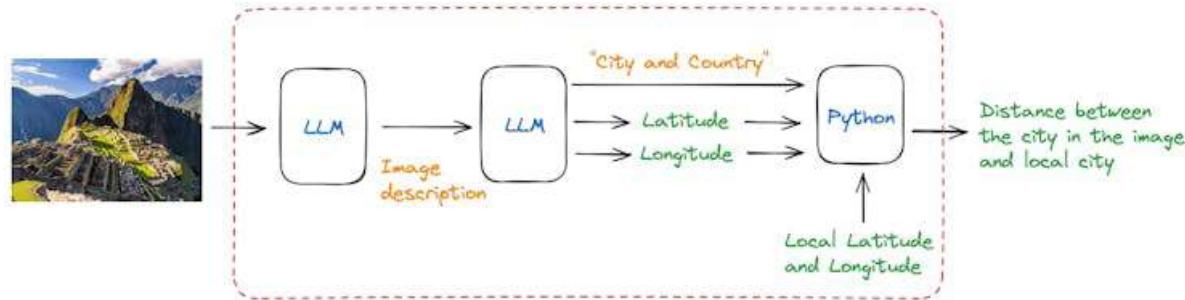
[20-Ollama_Function_Calling_Pydantic notebook](#)

Adding images

Now it is time to wrap up everything so far! Let's modify the script using Pydantic 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 that 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 time
from haversine import haversine
from ollama import chat
from pydantic import BaseModel, Field
  
```

We can see the image if you run the code on the Jupyter Notebook. For that, we also need to 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 = 'gemma3:4b'
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 = 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. In this case, we can get something as:

```
'{\n    "city": "Machu Picchu",\n    "country": "Peru",\n    "lat":\n-13.1631,\n    "lon": -72.5450\n}'
```

Let's define a Pydantic model (CityCoord) that describes the expected structure of the SLM's response. It expects four fields: country, city (city name), lat (latitude), and lon (longitude).

```
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")
    lat: float = Field(..., description="Decimal Latitude of the city in the image")
    lon: float = Field(..., description="Decimal Longitude of the city in the image")
```

The image description generated for the function will be passed as a prompt for the model again.

```
response = chat(
    model=MODEL,
    messages=[{
        "role": "user",
        "content": image_description # image_description from previous model's run
    }],
    format=CityCoord.model_json_schema(), # Structured JSON format
    options={"temperature": 0}
)
```

Now we can get the required data using:

```
resp = CityCoord.model_validate_json(response.message.content)
```

And so, we can calculate and print the distance, using `haversine()`:

```
distance = haversine((mylat, mylon), (resp.lat, resp.lon), unit='km')
print(f"\nThe 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")
```

And we will get:

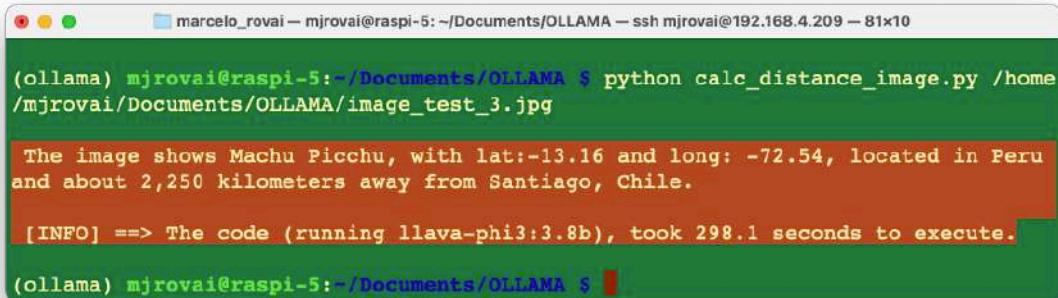
```
The image shows Machu Picchu, with lat:-13.16 and long: -72.55, located in Peru and about
```

In the [30-Function_Calling_with_images](#) notebook, you can find 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.



A screenshot of a terminal window titled "marcelo_rovai — mjrovai@raspi-5: ~/Documents/OLLAMA — ssh mjrovai@192.168.4.209 — 81x10". The command entered is "python calc_distance_image.py /home/mjrovai/Documents/OLLAMA/image_test_3.jpg". The output text is highlighted in orange and reads: "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." Below this, a green bar displays the message "[INFO] ==> The code (running llava-phi3:3.8b), took 298.1 seconds to execute." The prompt "(ollama) mjrovai@raspi-5:~/Documents/OLLAMA \$" is visible at the bottom.

Let's change the model for the gemma3:4b:



A screenshot of a terminal window titled "marcelo_rovai — mjrovai@raspi-5-SD: ~/Documents/OLLAMA — ssh mjrovai@raspi-5-SD.local — 80x9". The command entered is "python calc_distance_image.py /home/mjrovai/Documents/OLLAMA/image_test_3.jpg". The output text is highlighted in orange and reads: "The image shows Machu Picchu, with lat:-13.16 and long: -72.55, located in Peru and about 2,250 kilometers away from Santiago, Chile." Below this, a green bar displays the message "[INFO] ==> The code (running gemma3:4b), took 226.5 seconds to execute." The prompt "(ollama) mjrovai@raspi-5-SD:~/Documents/OLLAMA \$" is visible at the bottom.

The app is working fine with both models, with the Gemma being faster.

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 $
```

```
marcelo_rovai — mjrovai@raspi-5-SD: ~/Documents/OLLAMA — ssh mjrovai@raspi-5-SD.local — 81x9

(ollama) mjrovai@raspi-5-SD:~/Documents/OLLAMA $ python calc_distance_image.py /h
ome/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.

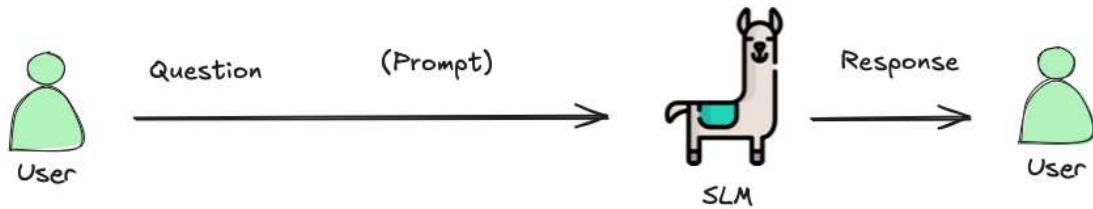
[INFO] ==> The code (running gemma3:4b), took 224.7 seconds to execute.

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

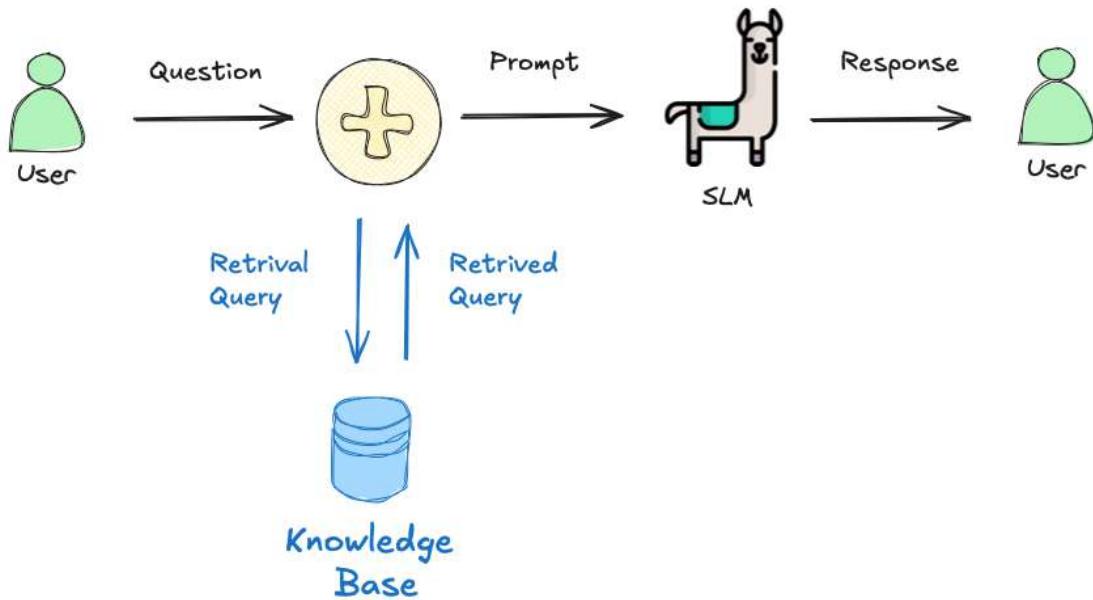
Of course, there are many ways to optimize the code used here. Still, the idea is to explore the considerable potential of *function calling* with SLMs at the edge, allowing those models to integrate with external functions or APIs. Going beyond text generation, SLMs can access real-time data, automate tasks, and interact with various systems.

Retrievel Augmentation Generation (RAG)

In a basic interaction between a user and a language model, the user asks a question, which is sent to the model as a prompt. 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. Chunking 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: mandaçaia (Melipona \
    quadrifasciata), mandaguari (Scaptotrigona postica) and jataí-amarela \
    (Tetragonisca angustula)."
]
```

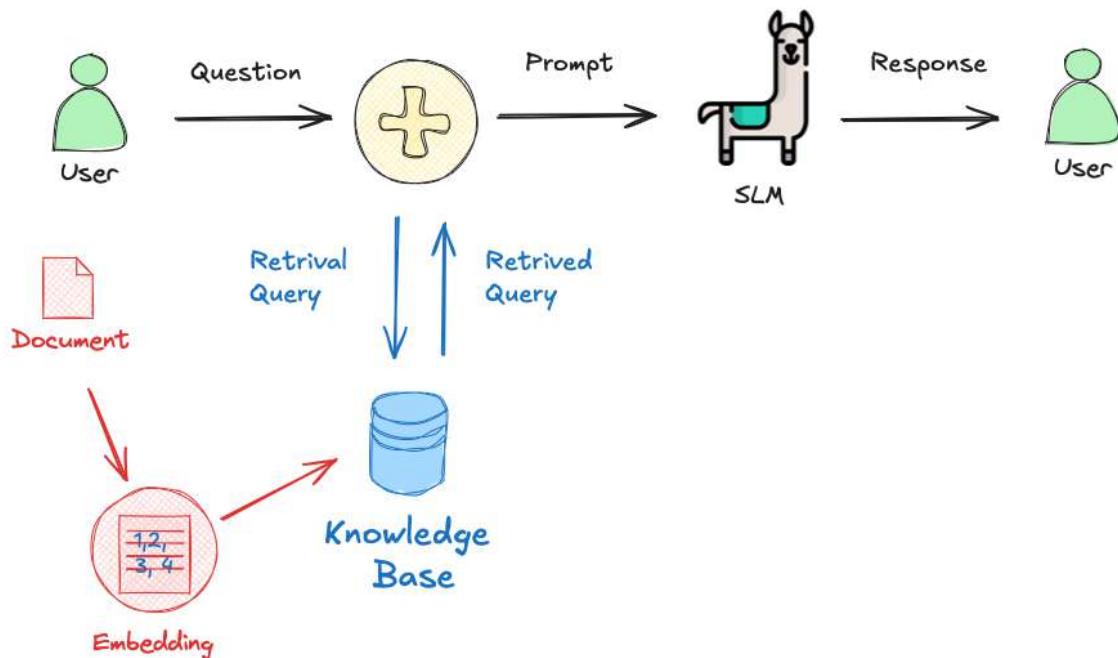
We do not need to “chunk” the document here because we will use each element of the list as a chunk.

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_bees(prompt, n_results=5, temp=0.0, top_k=10, top_p=0.5):
    start_time = time.perf_counter() # Start timing

    # generate an embedding for the prompt and retrieve the data
    response = ollama.embeddings(
        prompt=prompt,
        model=EMB_MODEL
    )

    results = collection.query(
        query_embeddings=[response["embedding"]],
        n_results=n_results
    )
    data = results['documents']

    # generate a response combining the prompt and data retrieved
    output = ollama.generate(
        model=MODEL,
        prompt=f"Using this data: {data}. 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 Brazil?"
rag_beans(prompt)

```

Sim, existem abelhas no Brasil! De acordo com o relato de Hans Staden, há três espécies de abelhas nativas do Brasil que foram mencionadas: 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 abelhas, a maioria das quais não é agressiva e não põe veneno.

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

In the Chapter [Advancing EdgeAI: Beyond Basic SLMs](#), we will learn how to implement a [Naive RAG System](#)

Conclusion

Throughout this chapter, we've explored two fundamental optimization techniques that significantly enhance the capabilities of Small Language Models (SLMs) running on edge devices like the Raspberry Pi: **Function Calling** and **Retrieval-Augmented Generation (RAG)**.

We began by addressing a critical limitation of language models—their inability to perform accurate calculations. By implementing function calling, we demonstrated how to transform SLMs from text generators into actionable agents that can interact with external tools and APIs. Whether extracting structured data like geographic coordinates, fetching real-time weather information, or performing precise mathematical operations, function calling bridges the gap between natural language understanding and deterministic computation. The key principle remains:

Use SLMs for intent classification and understanding, while delegating specific tasks to specialized functions that guarantee accuracy.

The RAG implementation showcased an elegant solution to another fundamental challenge—the static nature of model knowledge and the tendency toward hallucinations. By integrating ChromaDB with vector embeddings, we created a system that augments the model's responses with relevant, factual information retrieved from a knowledge base. This approach not only grounds the model's answers in verifiable data but also enables the system to stay current without the resource-intensive process of retraining or fine-tuning.

These techniques are particularly valuable for edge AI applications where computational resources are limited, and offline operation is often required. Function calling ensures reliability and precision, while RAG provides flexibility and accuracy without demanding continuous internet connectivity once the knowledge base is established.

As we move forward with our edge AI projects, remember that the power of SLMs lies not just in their standalone capabilities but in how effectively we orchestrate them with complementary tools and techniques. The patterns demonstrated here—structured tool use and knowledge augmentation—form the foundation for building robust, production-ready AI applications on resource-constrained devices.

In the chapter [Advancing EdgeAI: Beyond Basic SLMs](#), we'll go deeper into these concepts and explore more advanced implementations.

Resources

- [10-Ollama_Function_Calling notebook](#)
- [20-Ollama_Function_Calling_Pydantic](#)
- [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.

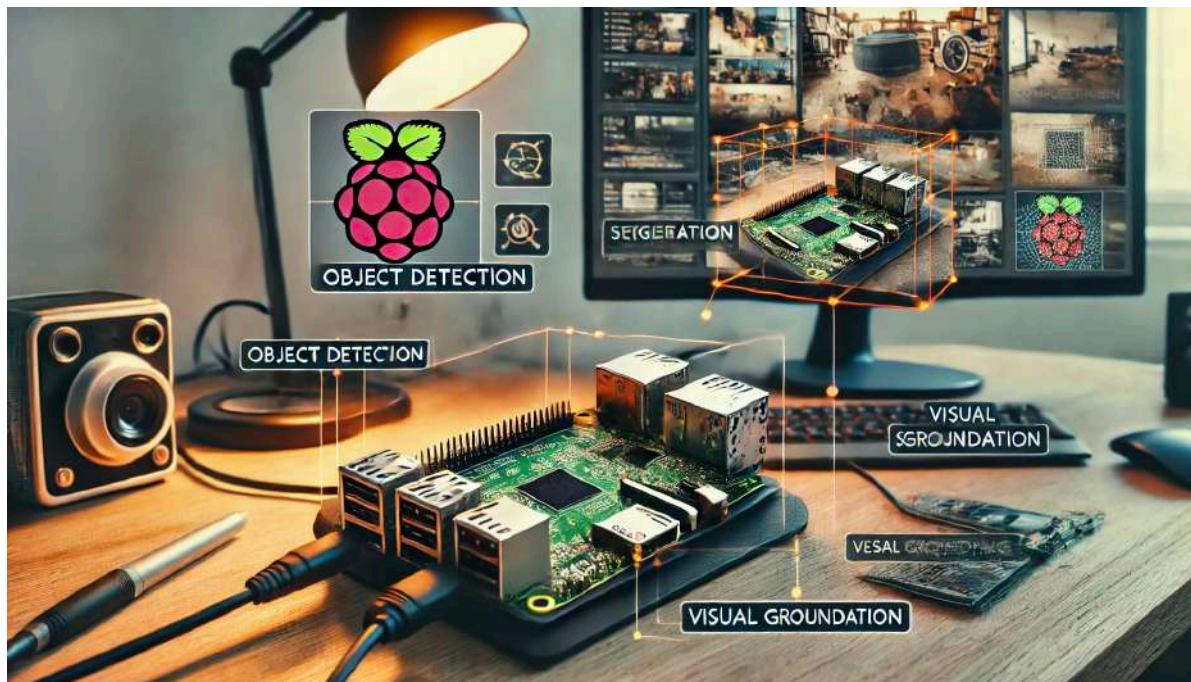


Figure 13: *DALL·E prompt - A Raspberry Pi setup featuring vision tasks.* The image shows a Raspberry Pi connected to a camera, with various computer vision tasks displayed visually around it, including object detection, image captioning, segmentation, and visual grounding. The Raspberry Pi is placed 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

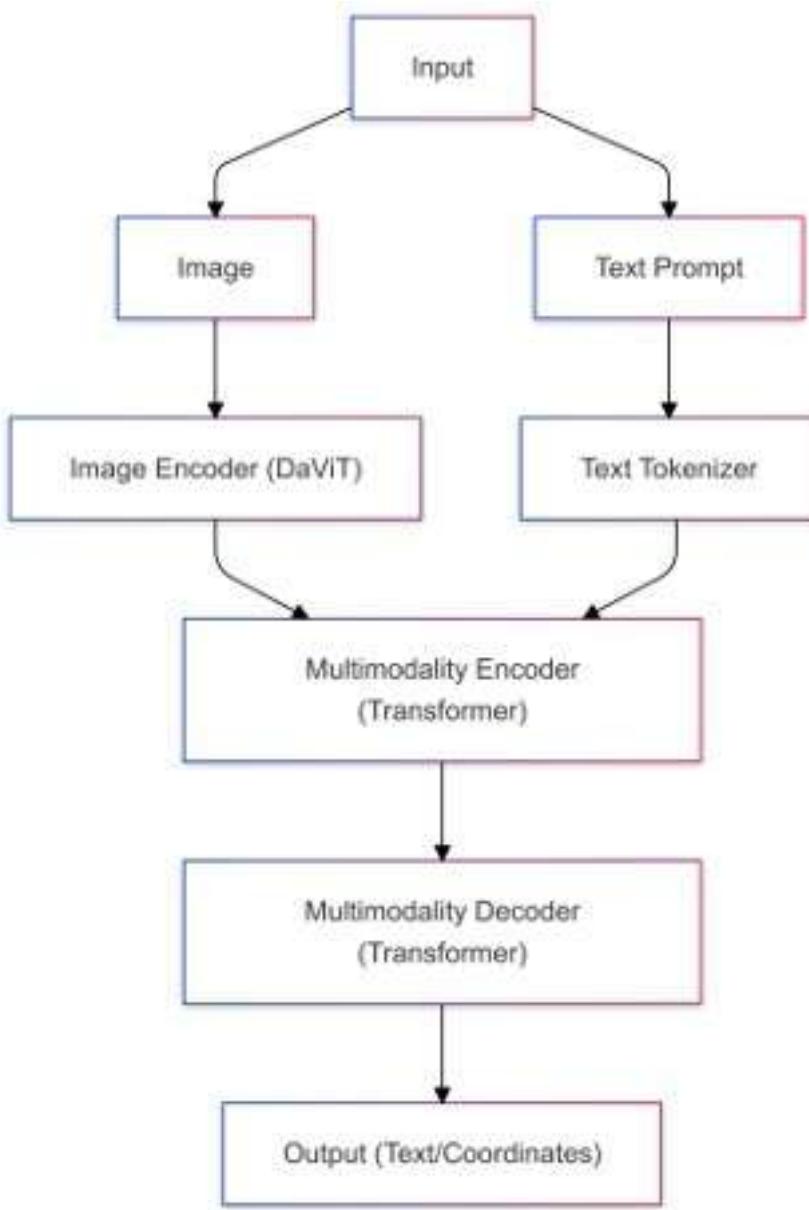
Visual grounding involves linking textual descriptions to specific regions within an image. This enables the model to understand where particular objects or entities described in a prompt are in the image. For example, if the prompt is “a red car,” the model will identify and highlight the region where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

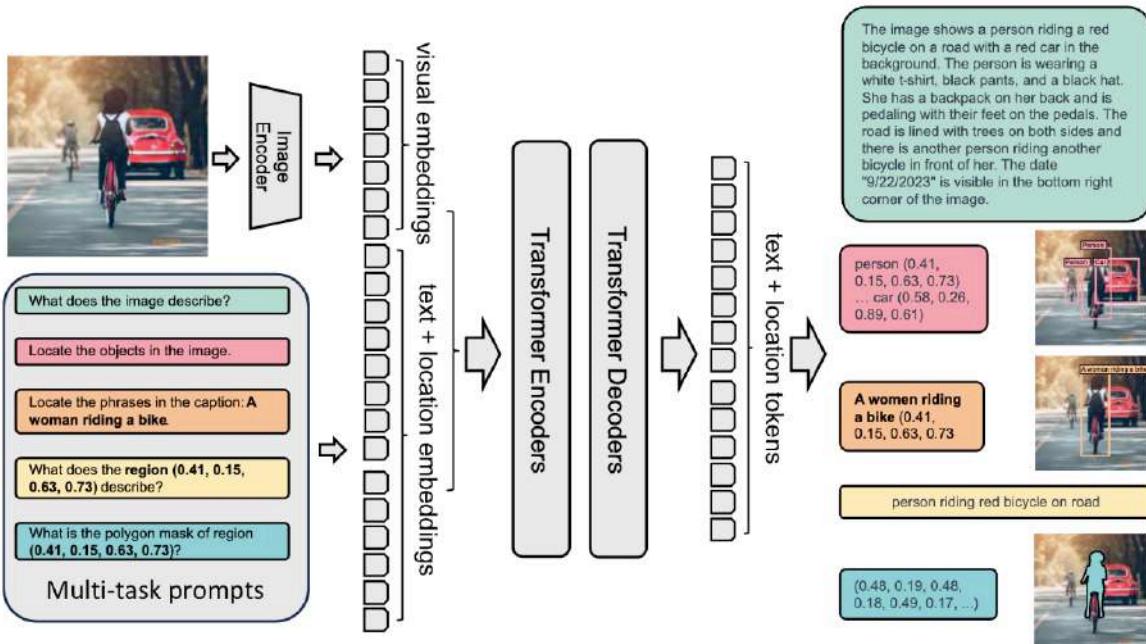
The model's training on the extensive FLD-5B dataset ensures it can effectively handle diverse vision tasks without requiring task-specific modifications. Florence-2 uses textual prompts to activate specific tasks, making it highly flexible and capable of zero-shot generalization. For tasks like object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

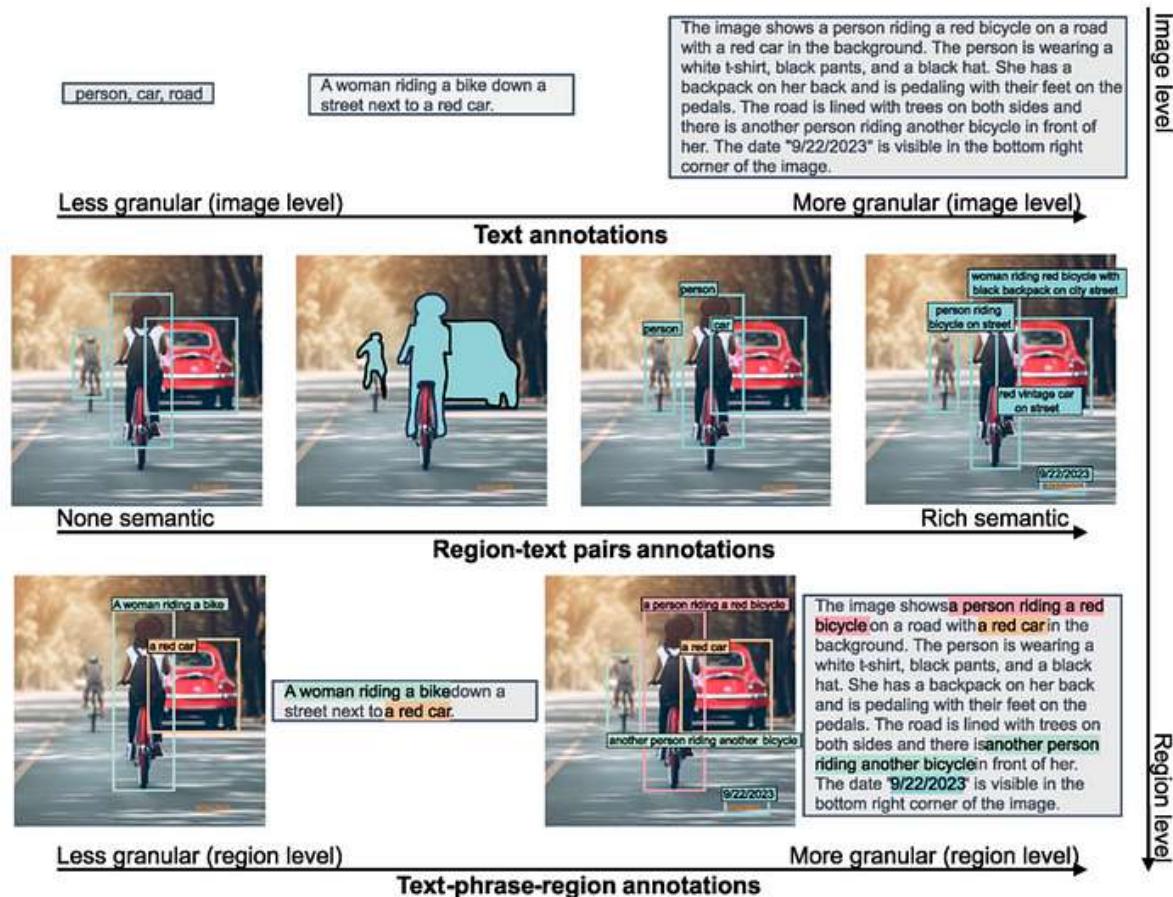
Architecture



- **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.

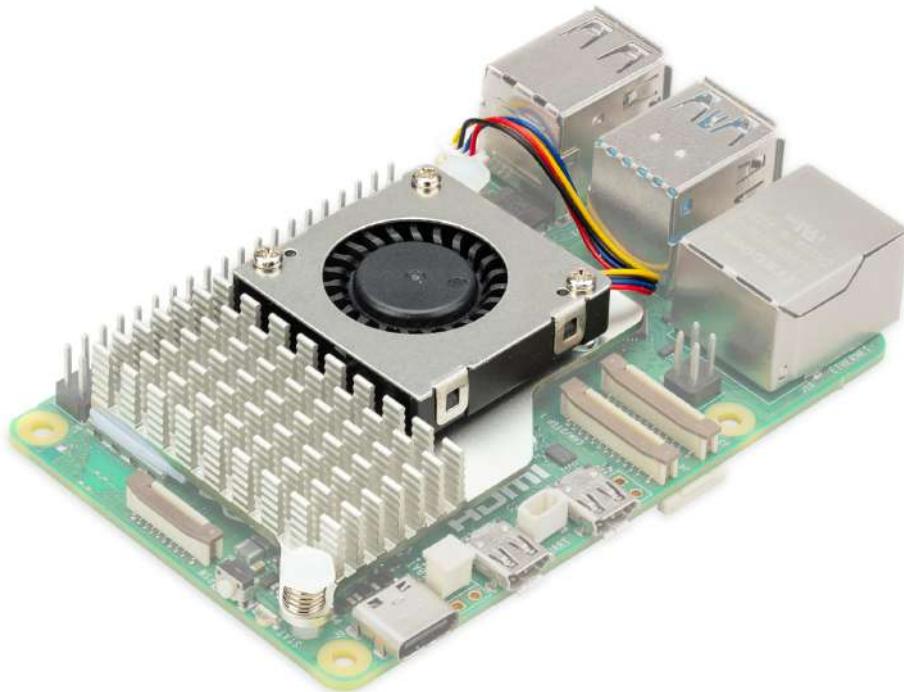
In benchmark tests, Florence-2 has shown remarkable performance in tasks like COCO captioning and referring expression comprehension. It outperformed models like PolyFormer and UNINEXT in object detection and segmentation tasks on the [COCO dataset](#). It is a highly competitive choice for real-world 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
```

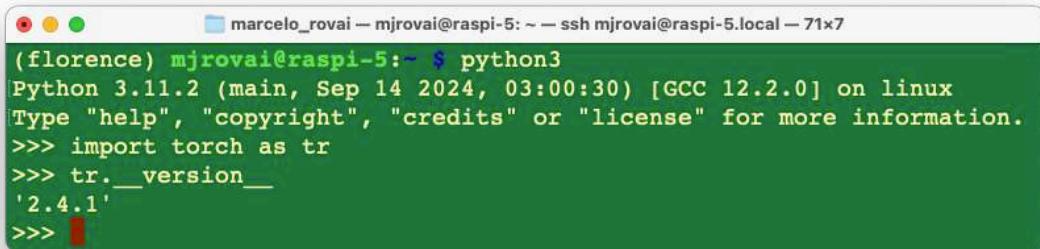
Let's set up and activate a **Virtual Environment** for working with Florence-2:

```
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.

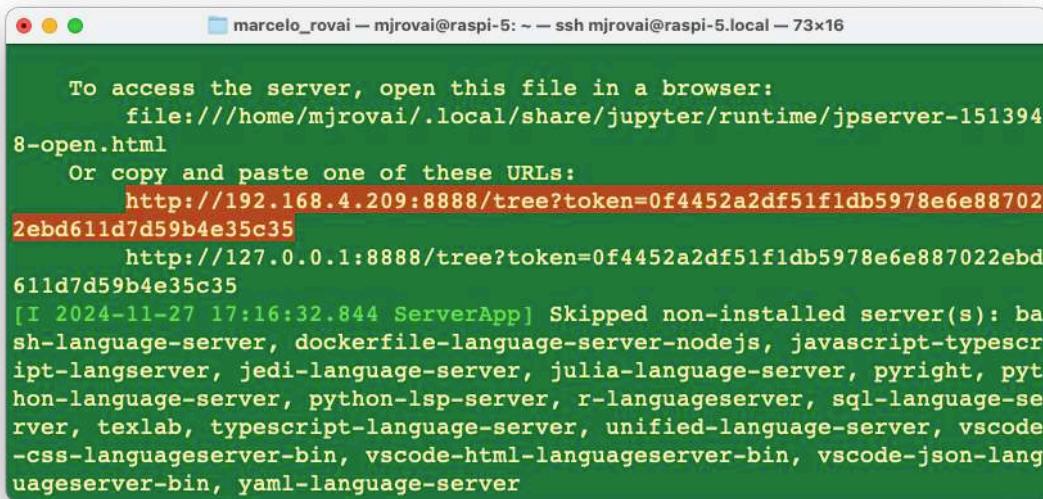
```
pip3 install jupyter
pip3 install numpy Pillow matplotlib
jupyter notebook --generate-config
```

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 terminal displays the following text:

```
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=0f4452a2df51f1db5978e6e88702  
zebd611d7d59b4e35c35  
http://127.0.0.1:8888/tree?token=0f4452a2df51f1db5978e6e887022ebd  
611d7d59b4e35c35  
[I 2024-11-27 17:16:32.844 ServerApp] Skipped non-installed server(s): ba  
sh-language-server, dockerfile-language-server-nodejs, javascript-typescr  
ipt-langserver, jedi-language-server, julia-language-server, pyright, pyt  
hon-language-server, python-lsp-server, r-languageserver, sql-languag  
e-server, texlab, typescript-language-server, unified-language-server, vscode  
-css-languageserver-bin, vscode-html-languageserver-bin, vscode-json-lang  
uageserver-bin, yaml-language-server
```

The notebook with the code used on this initial test can be found on the Lab GitHub:

- [10-florence2_test.ipynb](#)

We can access it on the remote computer by entering the Raspberry Pi's IP address and the provided token in a web browser (copy the entire URL from the terminal).

From the Home page, create a new notebook [Python 3 (ipykernel)] and copy and paste the [example code](#) from Hugging Face Hub.

The code is designed to run Florence-2 on a given image to perform **object detection**. It loads the model, processes an image and a prompt, and then generates a response to identify and describe the objects in the image.

- The **processor** helps prepare text and image inputs.
- The **model** takes the processed inputs to generate a meaningful response.
- The **post-processing** step refines the generated output into a more interpretable form, like bounding boxes for detected objects.

This workflow leverages the versatility of Florence-2 to handle **vision-language tasks** and is implemented efficiently using PyTorch, Transformers, and related image-processing tools.

```

import requests
from PIL import Image
import torch
from transformers import AutoProcessor, AutoModelForCausalLM

device = "cuda:0" if torch.cuda.is_available() else "cpu"
torch_dtype = torch.float16 if torch.cuda.is_available() else torch.float32

model = AutoModelForCausalLM.from_pretrained("microsoft/Florence-2-base",
                                              torch_dtype=torch_dtype,
                                              trust_remote_code=True).to(device)
processor = AutoProcessor.from_pretrained("microsoft/Florence-2-base",
                                           trust_remote_code=True)

prompt = "<OD>"

url = "https://huggingface.co/datasets/huggingface/documentation-
images/resolve/main/transformers/tasks/car.jpg?download=true"
image = Image.open(requests.get(url, stream=True).raw)

inputs = processor(text=prompt, images=image, return_tensors="pt").to(
    device, torch_dtype)

generated_ids = model.generate(
    input_ids=inputs["input_ids"],
    pixel_values=inputs["pixel_values"],
    max_new_tokens=1024,
    do_sample=False,
    num_beams=3,
)
generated_text = processor.batch_decode(generated_ids, skip_special_tokens=False)[0]

parsed_answer = processor.post_process_generation(generated_text, task="<OD>",
                                                 image_size=(image.width,
                                                             image.height))

print(parsed_answer)

```

Let's break down the provided code step by step:

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()

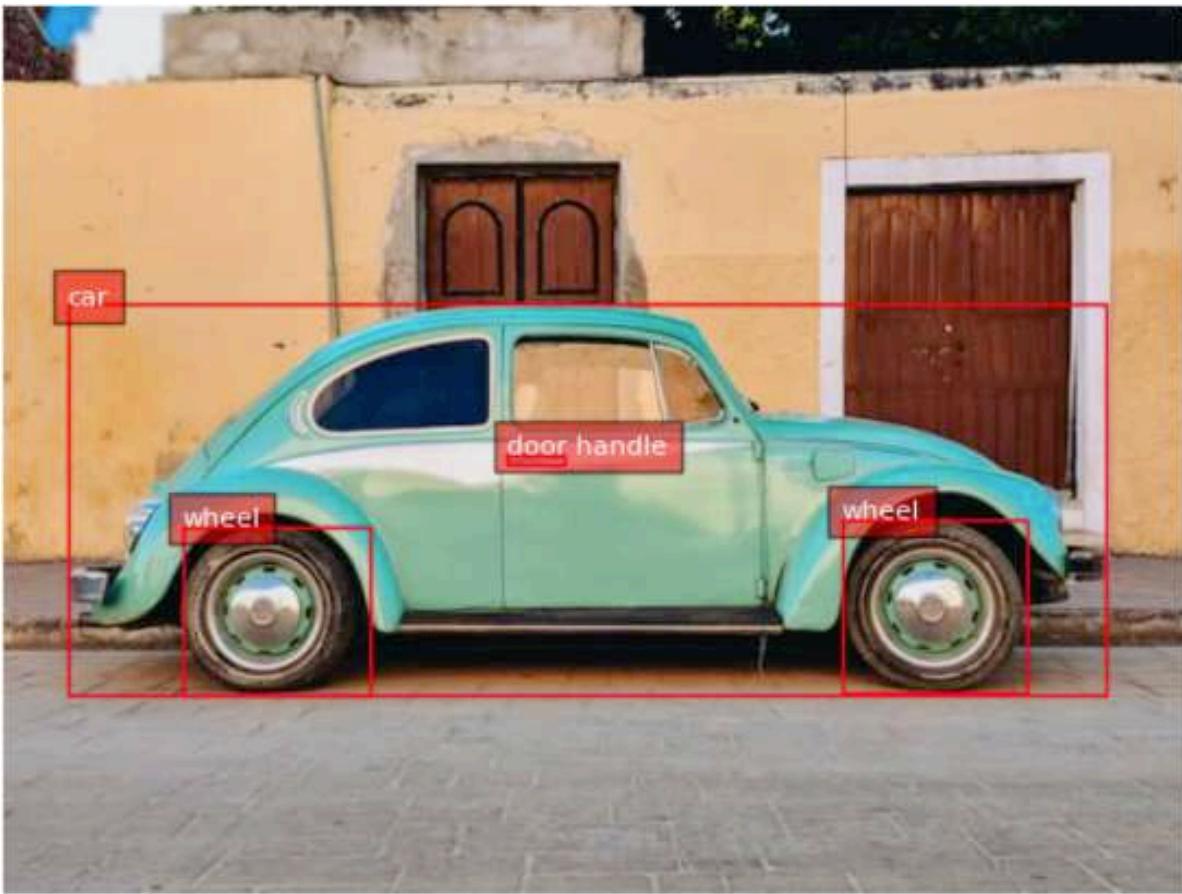
```

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['<OD>'])
```

We get:



Florence-2 Tasks

Florence-2 is designed to perform a variety of computer vision and vision-language tasks through **prompts**. These tasks can be activated by providing a specific textual prompt to the model, as we saw with <OD> (Object Detection).

Florence-2's versatility comes from combining these prompts, allowing us to guide the model's behavior to perform specific vision tasks. Changing the prompt allows us to adapt Florence-2 to different tasks without needing task-specific modifications in the architecture. This capability directly results from Florence-2's unified model architecture and large-scale multi-task training on the FLD-5B dataset.

Here are some of the key tasks that Florence-2 can perform, along with example prompts:

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.

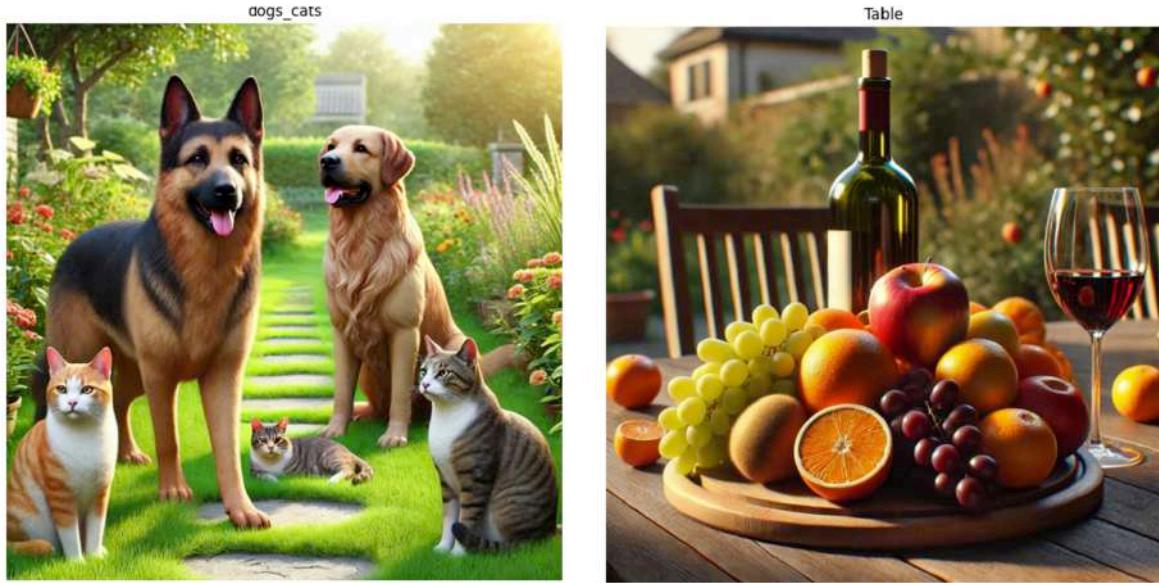
Exploring computer vision and vision-language tasks

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>.

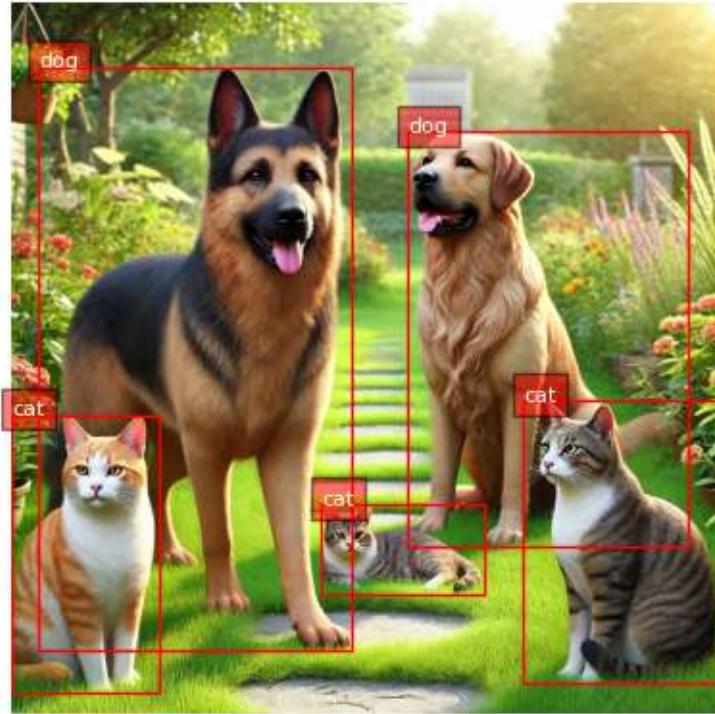
```
task_prompt = '<OD>'  
results = run_example(task_prompt,image=dogs_cats)  
print(results)
```

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>'])
```



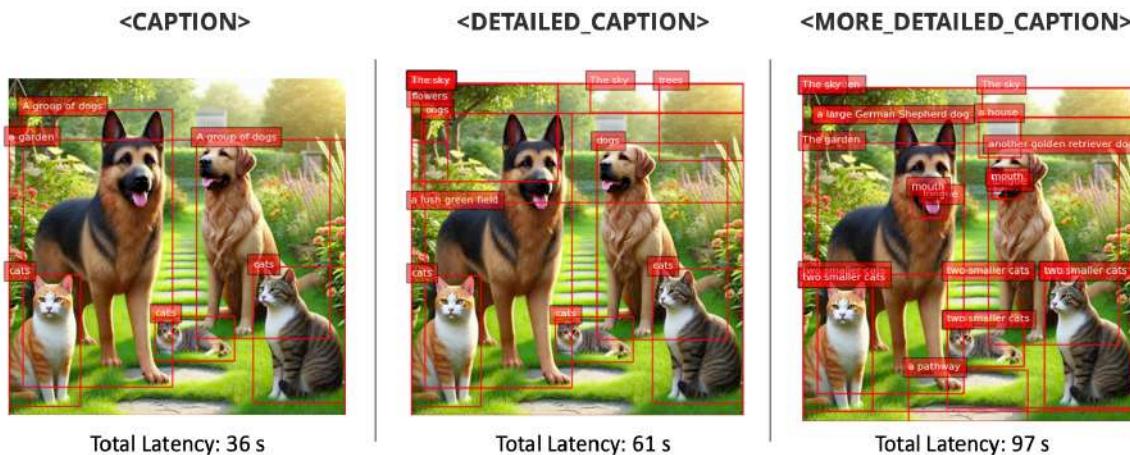
[INFO] ==> Florence-2-base (<CAPTION_TO_PHRASE_GROUNDING>), took 15.7 seconds to execute each task.

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.

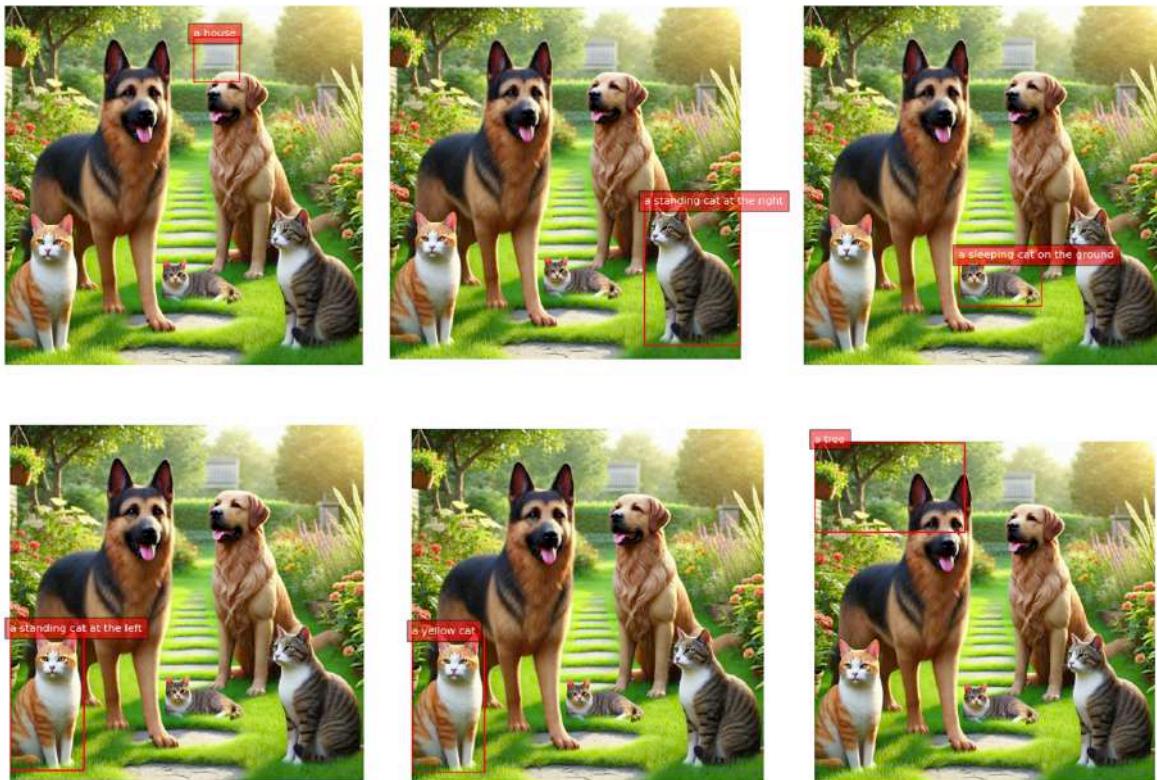


OPEN_VOCABULARY_DETECTION

<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



Marcelo Rovai

Professor na UNIFEI e
Co-Diretor do TinyML4D

Let's examine the image with '<MORE_DETAILED_CAPTION>' :

```
[INFO] ==> Florence-2-base (<MORE_DETAILED_CAPTION>), took 85.2 seconds to execute.  
  
{'<MORE_DETAILED_CAPTION>': 'The image is a promotional poster for an event called "Machine Learning Embarcados" hosted by Marcelo Roval. The poster has a black background with white text. On the left side of the poster, there is a logo of a coffee cup with the text "Café Com Embarcados" above it. Below the logo, it says "25 de Setembro as 17th" 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 Paises em Desenvolvimento" and the second text box says "Toda quarta-feira".  
In 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']

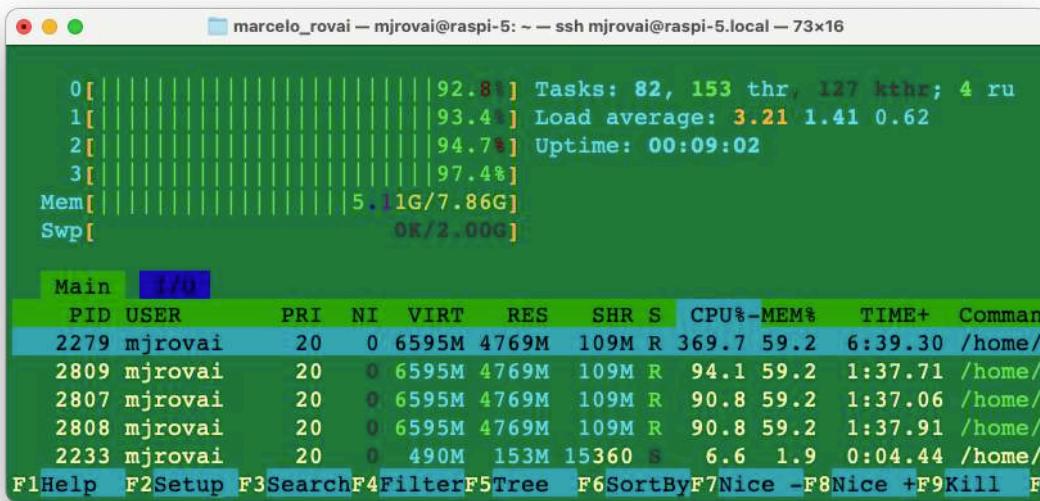
'</s>Machine Learning',
'Café',
'com',
'Embarcado',
'Embarcados',
'Democratizando a Inteligência',
'Artificial para Paises em',
'25 de Setembro ás 17h',
'Desenvolvimento',
'Toda quarta-feira',
'Marcelo Roval',
'Professor na UNIFIEI e',
'Transmissão via',
'in',
'Co-Director do TinyML4D']
```

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 on a Raspberry Pi. The top section displays system statistics: CPU usage (Tasks: 82, 153 thr, 127 kthr; 4 ru), Load average (3.21 1.41 0.62), and Uptime (00:09:02). Below this, memory usage is shown as 5.11G/7.86G. The main part of the screen is a table of processes, with 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 at priority 20. The bottom of the screen shows various keyboard shortcuts for navigation and filtering.

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%-MEM%	TIME+	Command
2279	mjrovai	20	0	6595M	4769M	109M	R	369.7 59.2	6:39.30	/home/
2809	mjrovai	20	0	6595M	4769M	109M	R	94.1 59.2	1:37.71	/home/
2807	mjrovai	20	0	6595M	4769M	109M	R	90.8 59.2	1:37.06	/home/
2808	mjrovai	20	0	6595M	4769M	109M	R	90.8 59.2	1:37.91	/home/
2233	mjrovai	20	0	490M	153M	15360	S	6.6 1.9	0:04.44	/home/

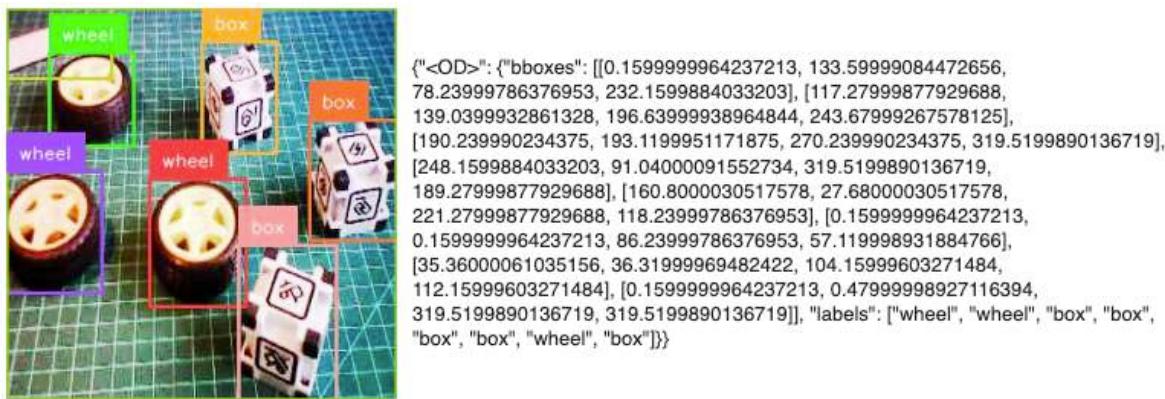
Running complex tasks can use all 8GB of the Raspi-5's memory. For example, the above screenshot during the Florence OD task shows 4 CPUs at full speed and over 5GB of memory in use. Consider increasing the SWAP memory to 2 GB.

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.

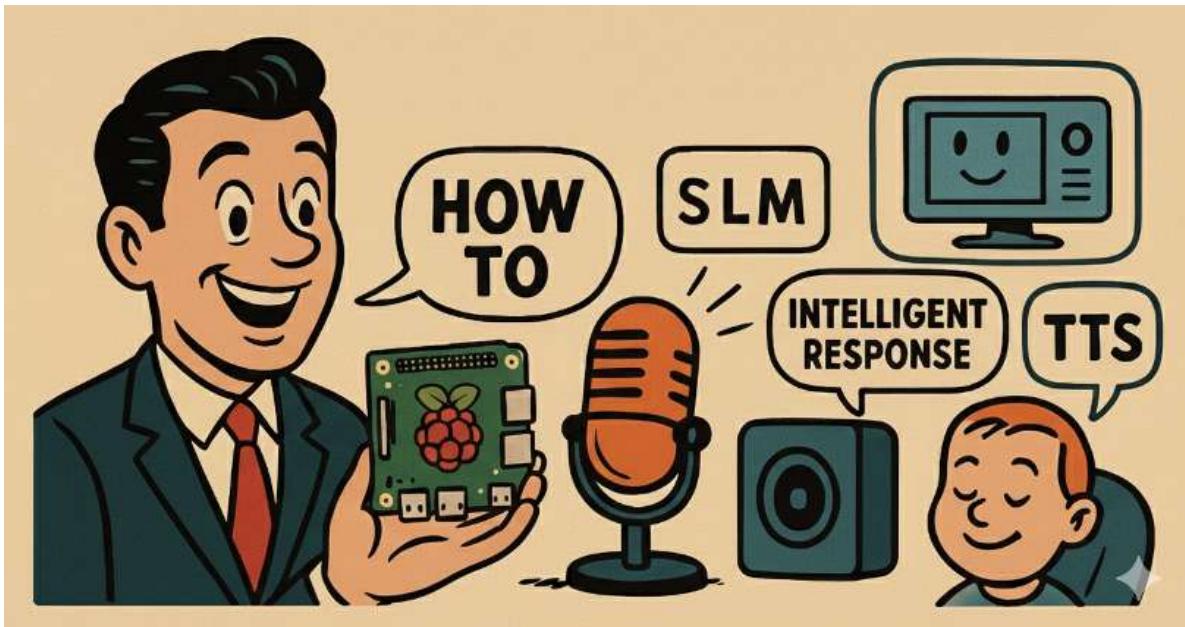
The lab demonstrates Florence-2's viability on edge devices, suggesting future IoT, mobile computing, and embedded systems applications where deploying multiple specialized models would be impractical.

Resources

- [10-florence2_test.ipynb](#)
- [20-florence_2.ipynb](#)
- [30-Finetune_florence_2_on_detection_dataset_box_vs_wheel.ipynb](#)

Audio and Vision AI Pipeline

Building Voice and Vision Interactive Edge AI Systems

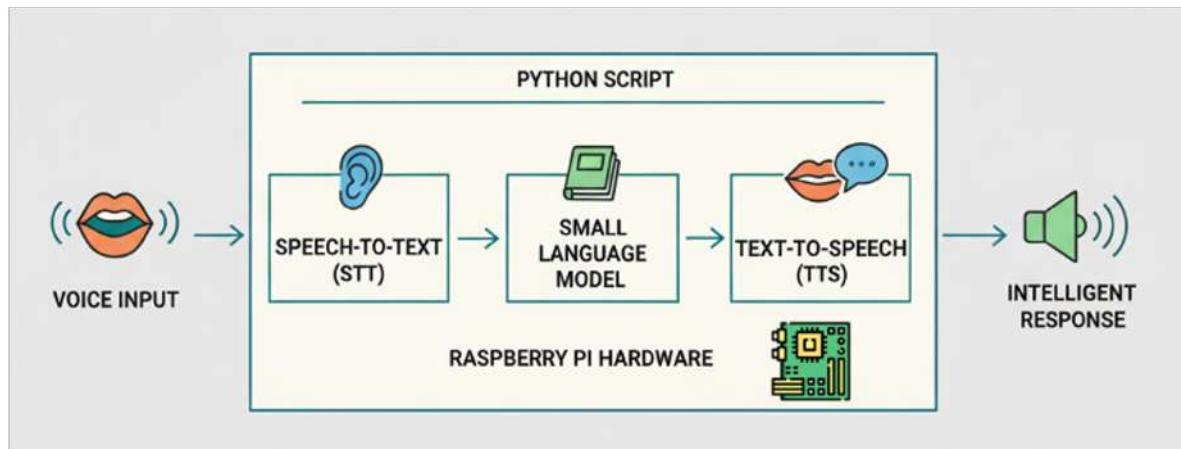


Introduction

In this chapter, we extend our SLM and SVL capabilities by creating a complete audio processing pipeline that transforms voice or image input into intelligent vocal responses. We will learn to integrate Speech-to-Text (STT), Small Language (or Visual) Models, and Text-to-Speech (TTS) technologies to build conversational AI systems that run entirely on Raspberry Pi hardware.

This chapter bridges the gap between our existing computer vision knowledge and multimodal AI applications, demonstrating how different AI components work together in real-world edge deployments.

The Audio to Audio AI Pipeline Architecture



We will understand how to architect and implement multimodal AI systems by building a complete voice interaction pipeline. The goal is to gain practical experience with audio processing on edge devices while learning to efficiently integrate multiple AI models within resource constraints. Additionally, you will develop troubleshooting skills for complex AI pipelines and understand the engineering trade-offs involved in edge audio processing.

Understanding Multimodal AI Systems

When we built computer vision systems earlier in the course, we processed visual data to extract meaningful information. Audio AI systems follow a similar principle but work with temporal audio signals instead of static images. The key insight is that speech processing requires multiple specialized models working together, rather than a single end-to-end system.

Modern small models, such as [Gemma 3n](#), can process audio directly and its prompt. Today (September 2025), Gemma 3n can transcribe text from audio files using Hugging Face Transformers, but it is not available with Ollama

Consider how humans process spoken language. We simultaneously parse the acoustic signal, understand the linguistic content, reason about the meaning, and formulate responses. Our AI pipeline mimics this process by breaking it into distinct, manageable components.

System Architecture Overview

Our comprehensive audio AI pipeline comprises four main components, connected in sequence.



Audio captured by the microphone is processed through a Speech-to-Text model, which converts sound waves into text transcriptions. This text becomes input for our Small Language Model, which generates intelligent responses. Finally, a Text-to-Speech system converts the written response back into spoken audio.

Each component has specific requirements and limitations. The STT model must handle various accents and noise conditions. The SLM needs sufficient context to generate coherent responses. The TTS system must produce speech that sounds natural. Understanding these individual requirements helps us optimize the overall system performance.

Edge AI Considerations

Running this pipeline on a Raspberry Pi presents unique challenges compared to cloud-based solutions. We must carefully manage memory usage, processing time, and model sizes. The benefit is complete local processing with no internet dependency and enhanced privacy protection.

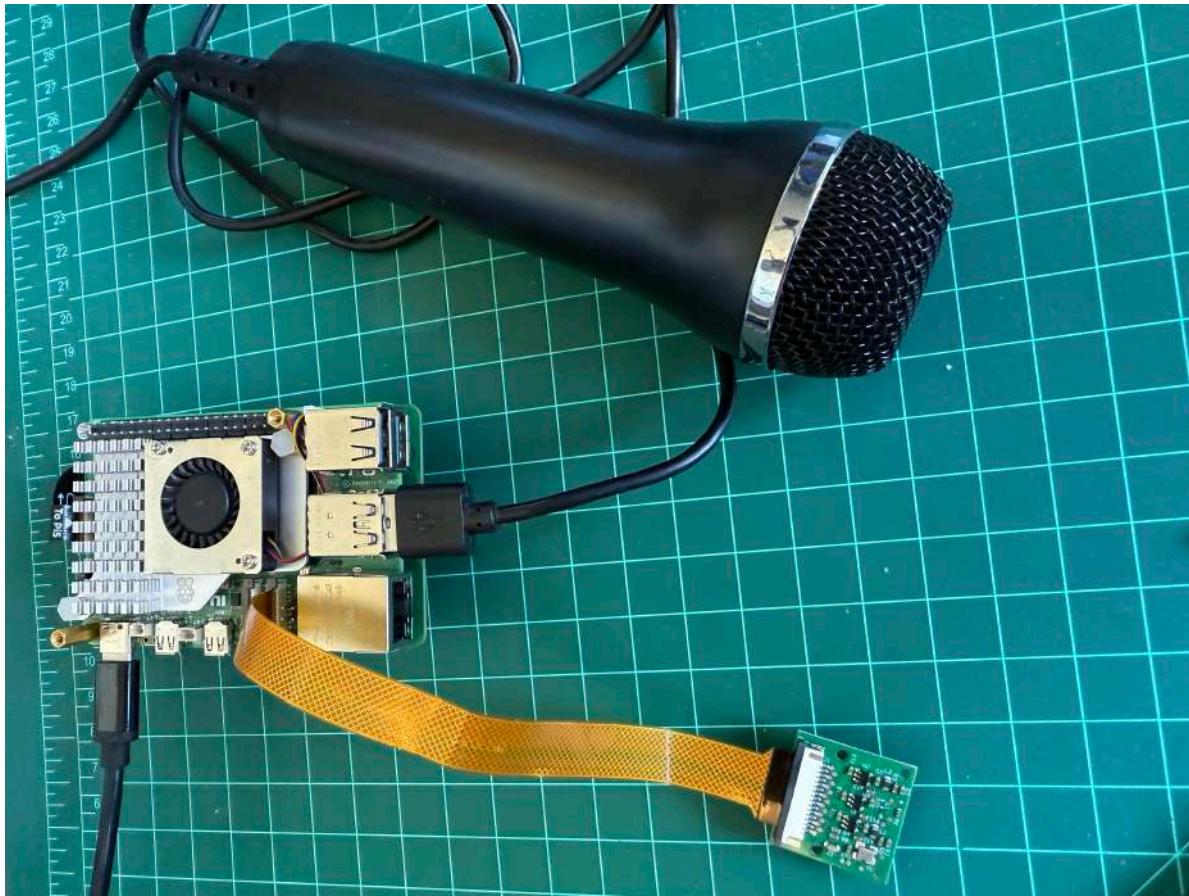
The choice of models becomes critical. We select [Moonshine](#) for STT because it's specifically optimized for edge devices. We utilize small language models, such as `llama3.2:3b`, for reasonable performance on limited hardware. For TTS, we choose [PIPER](#) for its balance between quality and computational efficiency.

Hardware Setup and Audio Capture

Audio Hardware Detection

Begin by identifying the audio capabilities of our system. The Raspberry Pi can work with various audio input and output devices, but proper configuration is essential for reliable operation.

Use the command `arecord -l` to list available recording devices. You should see output showing your microphone's card and device numbers. For USB microphones, this typically appears as something like `card 2: Microphone [USB Condenser Microphone], device 0: USB Audio [USB Audio]`. The critical information is the **card number** and device number, which you'll reference as `hw:2,0` in the code.



```
marcelo_rovai — mjrovai@raspi-5-SD: ~ — ssh mjrovai@raspi-5-SD.local — 81x6
(ollama) mjrovai@raspi-5-SD:~$ arecord -l
**** List of CAPTURE Hardware Devices ****
card 2: Microphone [USB Condenser Microphone], device 0: USB Audio [USB Audio]
  Subdevices: 1/1
  Subdevice #0: subdevice #0
(ollama) mjrovai@raspi-5-SD:~$
```

Testing Basic Audio Functionality

Before writing Python code, we should verify that our audio setup works correctly at the system level. Let's record a short test file using:

```
arecord --device="plughw:2,0" --format=S16_LE --rate=16000 -c2 myaudio.wav
```

Play back the recording with `aplay myaudio.wav` to confirm that both capture and playback work correctly (use [CTRL]+[C] to stop the recording or add a duration in seconds to the command line).



```
marcelo_reval -- mjroval@raspi-5-SD: ~ ssh mjroval@raspi-5-SD.local -t 107x13
(ollama) mjroval@raspi-5-SD: ~ $ arecord --device="plughw:2,0" --format=S16_LE --rate=16000 -c2 myaudio.wav
Recording WAVE 'myaudio.wav' : Signed 16 bit Little Endian, Rate 16000 Hz, Stereo
^CAborted by signal Interrupt...
arecord: pcm_read:2221: read error: Interrupted system call
(ollama) mjroval@raspi-5-SD: ~ $ ls
Bookshelf Pictures Videos output.wav voice.wav
Desktop Public audio-test.ipynb test.wav
Documents Templates gemma-3n-E4B-it test16k.wav
Downloads Untitled.ipynb myaudio.wav test_cli_camera.jpg
Music Untitled2.ipynb ollama transformers_env
(ollama) mjroval@raspi-5-SD: ~ $ aplay myaudio.wav
Playing WAVE 'myaudio.wav' : Signed 16 bit Little Endian, Rate 16000 Hz, Stereo
(ollama) mjroval@raspi-5-SD: ~ $
```

The .WAV file can be played on another device (such as a computer) or on the Raspberry Pi, as a speaker can be connected via USB or Bluetooth.

Python Audio Integration

Now, we should install the necessary audio processing dependencies.

```
source ~/ollama/bin/activate
```

The PyAudio library requires system-level audio libraries; therefore, install them using `sudo apt-get`.

```
sudo apt-get update
sudo apt-get install libasound-dev libportaudio2 libportaudiocpp0 portaudio19-dev
sudo apt-get install python3-pyaudio
```

- `libasound-dev` covers ALSA development headers needed for audio libraries on Pi
- `python3-pyaudio` provides a prebuilt PyAudio package for most use cases

Let's create a working directory: `Documents/OLLAMA/SST` and verify the USB device index, with the below script (`verify_usb_index.py`):

```
import pyaudio
p = pyaudio.PyAudio()
for ii in range(p.get_device_count()):
    print(ii, p.get_device_info_by_index(ii).get('name'))
```

As a result, we should get:

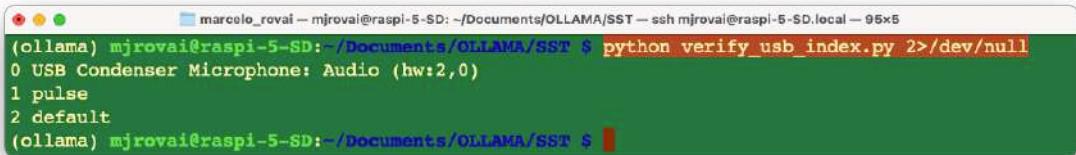
```
0 USB Condenser Microphone: Audio (hw:2,0)
1 pulse
2 default
```

What confirms that the index is 2 (hw:2,0)

A lot of messages should appear. They are mostly ALSA and JACK warnings about missing or undefined virtual/surround sound devices—they are common on Raspberry Pi systems with minimal or headless sound configs and typically do not impact basic USB microphone capture. If our USB Microphone appears as a recording device (as it does: “hw:2,0”), we can safely ignore most of these unless audio capture fails.

To clean the output, we can use:

```
python verify_usb_index.py 2>/dev/null
```



```
marcelo_reval — mjrovai@raspi-5-SD: ~/Documents/OLLAMA/SST — ssh mjrovai@raspi-5-SD.local — 95x5
(mollama) mjrovai@raspi-5-SD:~/Documents/OLLAMA/SST $ python verify_usb_index.py 2>/dev/null
0 USB Condenser Microphone: Audio (hw:2,0)
1 pulse
2 default
(mollama) mjrovai@raspi-5-SD:~/Documents/OLLAMA/SST $
```

Test Audio Python Script

This script (`test_audio_capture.py`) records 10 seconds of mono audio at 16 kHz to `output.wav` using the USB microphone.

```
import pyaudio
import wave

FORMAT = pyaudio.paInt16
CHANNELS = 1
RATE = 16000      # 16 kHz
CHUNK = 1024
RECORD_SECONDS = 10
DEVICE_INDEX = 2  # replace this with your detected USB mic's index
WAVE_OUTPUT_FILENAME = "output.wav"
```

```

audio = pyaudio.PyAudio()

stream = audio.open(format=FORMAT, channels=CHANNELS,
                     rate=RATE, input=True, input_device_index=DEVICE_INDEX,
                     frames_per_buffer=CHUNK)

print("Recording...")
frames = []

for i in range(0, int(RATE / CHUNK * RECORD_SECONDS)):
    data = stream.read(CHUNK, exception_on_overflow=False)
    frames.append(data)

print("Finished recording.")

stream.stop_stream()
stream.close()
audio.terminate()

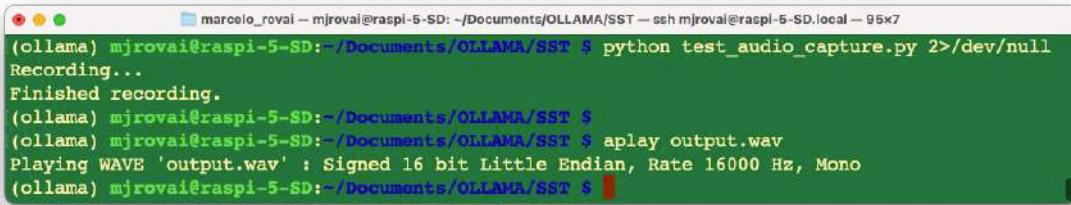
wf = wave.open(WAVE_OUTPUT_FILENAME, 'wb')
wf.setnchannels(CHANNELS)
wf.setsampwidth(audio.get_sample_size(FORMAT))
wf.setframerate(RATE)
wf.writeframes(b''.join(frames))
wf.close()

```

Understanding the audio configuration parameters helps prevent common problems. We use 16-bit PCM format (`pyaudio.paInt16`) because it provides good quality while remaining computationally efficient. The 16kHz sampling rate balances audio quality with processing requirements - most speech recognition models expect this rate.

The buffer size (`CHUNK = 1024`) affects latency and reliability. Smaller buffers reduce latency but may cause audio dropouts on busy systems. Larger buffers increase latency but provide more stable recording.

Let's Playback to verify if we get it correctly:



```
marcelo_roval - miroval@raspi-5-SD: ~/Documents/OLLAMA/SST - ssh miroval@raspi-5-SD.local - 95x7
(ollama) miroval@raspi-5-SD:~/Documents/OLLAMA/SST $ python test_audio_capture.py 2>/dev/null
Recording...
Finished recording.
(ollama) miroval@raspi-5-SD:~/Documents/OLLAMA/SST $ 
(ollama) miroval@raspi-5-SD:~/Documents/OLLAMA/SST $ aplay output.wav
Playing WAVE 'output.wav' : Signed 16 bit Little Endian, Rate 16000 Hz, Mono
(ollama) miroval@raspi-5-SD:~/Documents/OLLAMA/SST $
```

Your browser does not support the audio element.

Speech Recognition (SST) with Moonshine

Why Moonshine for Edge Deployment

Traditional speech recognition systems, such as OpenAI’s Whisper, are highly accurate but require substantial computational resources. [Moonshine](#) is specifically designed for edge devices, using optimized model architectures and quantization techniques to achieve good performance on resource-constrained hardware.

The ONNX (Open Neural Network Exchange) version of Moonshine provides additional optimization benefits. ONNX Runtime includes hardware-specific optimizations that can significantly improve inference speed on ARM processors, such as those found in Raspberry Pi devices.

Model Selection Strategy

Moonshine offers different model sizes with clear trade-offs between accuracy and computational requirements. The “tiny” model processes audio quickly but may struggle with difficult audio conditions. The “base” model provides better accuracy but requires more processing time and memory.

For initial development, we should start with the tiny model to ensure that the pipeline works correctly. Once the complete system is functional, we can experiment with larger models to find the optimal balance for our specific use case and hardware capabilities.

Implementation and Preprocessing

Install Moonshine with

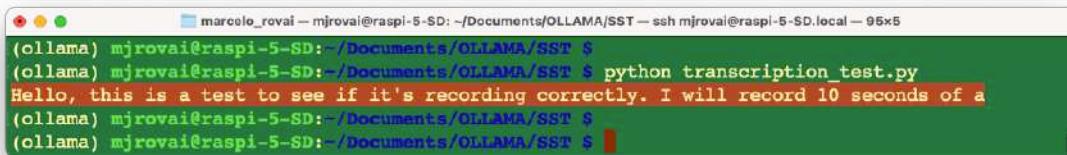
```
pip install useful-moonshine-onnx@git+https://github.com/moonshine  
ai/moonshine.git#subdirectory=moonshine-onnx`
```

This specific installation method ensures compatibility with the ONNX runtime optimizations.

Let's run the test script below (`transcription_test.py`):

```
import moonshine_onnx  
text = moonshine_onnx.transcribe('output.wav', 'moonshine/tiny')  
print(text[0])
```

As a result, we will get the corresponding text, which was recorded before:



```
marcelo_roval - mjrovai@raspi-5-SD: ~/Documents/OLLAMA/SST - ssh mjrovai@raspi-5-SD.local - 96x5  
(ollama) mjrovai@raspi-5-SD:~/Documents/OLLAMA/SST $  
(ollama) mjrovai@raspi-5-SD:~/Documents/OLLAMA/SST $ python transcription_test.py  
Hello, this is a test to see if it's recording correctly. I will record 10 seconds of a  
(ollama) mjrovai@raspi-5-SD:~/Documents/OLLAMA/SST $  
(ollama) mjrovai@raspi-5-SD:~/Documents/OLLAMA/SST $
```

Empty or incorrect transcriptions are common issues in speech recognition systems. These failures can result from background noise, insufficient volume, unclear speech, or mismatches in audio format. Implementing robust error handling prevents these issues from crashing your entire pipeline.

SLM Integration and Response Generation

Connecting STT Output to SLMs

The text output from your speech recognition system becomes input for your Small Language Model. However, the characteristics of spoken language differ significantly from written text, especially when filtered through speech recognition systems.

Spoken language tends to be more informal, may contain false starts and repetitions, and might include transcription errors. Our SLM integration should account for these characteristics. On

a final implementation, we should consider preprocessing the STT output to clean up obvious transcription errors or providing context to the SLM about the voice interaction nature of the input.

Optimizing Prompts for Voice Interaction

Voice-based interactions have different expectations than text-based chats. Responses should be concise since users must listen to the entire output. Avoid complex formatting or long lists that work well in text but become cumbersome when spoken aloud.

We should design our system prompts to encourage responses appropriate for voice interaction. For example, “Provide a brief, conversational response suitable for speaking aloud” can help guide the SLM toward more appropriate output formatting.

Unlike single-query text interactions, voice conversations often involve multiple exchanges. Implementing conversation context memory significantly enhances the user experience. However, context management on edge devices requires careful consideration of memory usage.

Consider implementing a sliding window approach, where you maintain the last few exchanges in memory but discard older context to prevent memory exhaustion, balancing context length with available system resources.

Let's create a function to handle this. For test, run `slm_test.py`:

```
import ollama

def generate_voice_response(user_input, model="llama3.2:3b"):
    """
    Generate a response optimized for voice interaction
    """

    # Context-setting prompt that guides the model's behavior
    system_context = """
        You are a helpful AI assistant designed for voice interactions.
        Your responses will be converted to speech and spoken aloud to the user.

    Guidelines for your responses:
    - Keep responses conversational and concise (ideally under 50 words)
    - Avoid complex formatting, lists, or visual elements
    - Speak naturally, as if having a friendly conversation
    - If the user's input seems unclear, ask for clarification politely
    - Provide direct answers rather than lengthy explanations unless specifically requested
    """

    # Generate the response
    response = ollama.generate(system_context + user_input, model)

    return response
```

```

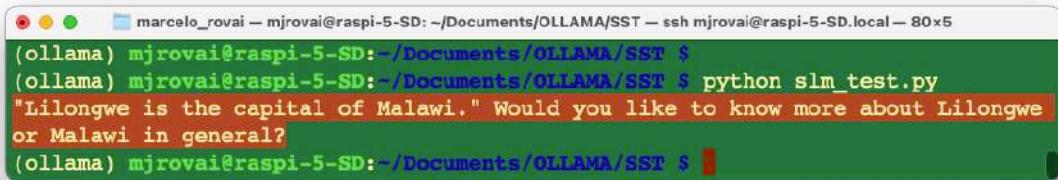
"""
# Combine system context with user input
full_prompt = f"{system_context}\n\nUser said: {user_input}\n\nResponse:"

response = ollama.generate(
    model=model,
    prompt=full_prompt
)

return response['response']

# Answering the user question:
user_input = "What is the capital of Malawi?"
response = generate_voice_response(user_input)
print (response)

```



```

marcelo_rovai — mjrovai@raspi-5-SD: ~/Documents/OLLAMA/SST — ssh mjrovai@raspi-5-SD.local — 80x5
(ollama) mjrovai@raspi-5-SD:~/Documents/OLLAMA/SST $ 
(ollama) mjrovai@raspi-5-SD:~/Documents/OLLAMA/SST $ python slm_test.py
"What is the capital of Malawi?" Would you like to know more about Lilongwe
or Malawi in general?
(ollama) mjrovai@raspi-5-SD:~/Documents/OLLAMA/SST $ 

```

Text-to-Speech (TTS) with PIPER

Text-to-speech systems face different challenges than speech recognition systems. While STT must handle various input conditions, TTS must generate consistent, natural-sounding output across diverse text inputs. The quality of TTS has a significant impact on the user experience in voice interaction systems.

[PIPER](#) provides an excellent balance between voice quality and computational efficiency for edge deployments. Unlike cloud-based TTS services, PIPER runs entirely locally, ensuring privacy and eliminating network dependencies.

Voice Model Selection and Installation

PIPER offers various voice models with different characteristics. The “low”, “medium”, and “high” quality designations primarily refer to model size and computational requirements rather than dramatic quality differences. For most applications, the low-quality models provide acceptable voice output while running efficiently on Raspberry Pi hardware.

Install PIPER with `pip install piper-tts`, create a voices directory:

```
pip install piper-tts
mkdir -p voices
```

Download voice models from the Hugging Face repository. Each voice model requires both the model file (.onnx) and a configuration file (.json). The configuration file contains model-specific parameters essential for generating proper audio.

We should download both files for our chosen voice; for example, the English female “lessac” voice provides clear, natural speech suitable for most applications.

```
wget -O voices/en_US-lessac-low.onnx https://huggingface.co/rhasspy/piper-voices/resolve/v1.0.0/en/en_US/lessac/low/en_US-lessac-low.onnx
```

```
wget -O voices/en_US-lessac-low.onnx.json https://huggingface.co/rhasspy/piper-voices/resolve/v1.0.0/en/en_US/lessac/low/en_US-lessac-low.onnx.json
```

Let’s run the code below (`tts_test.py`) for testing:

```
import subprocess
import os

def text_to_speech_piper(text, output_file="piper_output.wav"):
    """
    Convert text to speech using PIPER and save to WAV file

    Args:
        text (str): Text to convert to speech
        output_file (str): Output WAV file path
    """
    # Path to your voice model
    model_path = "voices/en_US-lessac-low.onnx"

    # Check if model exists
    if not os.path.exists(model_path):
```

```

print(f"Error: Model file not found at {model_path}")
return False

try:
    # Run PIPER command
    process = subprocess.Popen(
        ['piper', '--model', model_path, '--output_file', output_file],
        stdin=subprocess.PIPE,
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE,
        text=True
    )

    # Send text to PIPER
    stdout, stderr = process.communicate(input=text)

    if process.returncode == 0:
        print(f"\nSpeech generated successfully: {output_file}")
        return True
    else:
        print(f"Error: {stderr}")
        return False

except Exception as e:
    print(f"Error running PIPER: {e}")
    return False

# converting text to sound:
txt = "Lilongwe is the capital of Malawi. Would you like to know more about \
      Lilongwe or Malawi in general?"

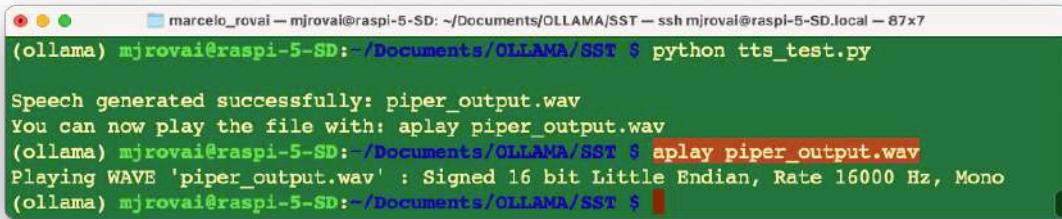
if text_to_speech_piper(txt):
    print("You can now play the file with: aplay piper_output.wav")
else:
    print("Failed to generate speech")

```

Runing the script, a `piper_output.wav` file will be generated, which is the text converted into speech.

Your browser does not support the audio element.

To listen to the sound, we can run: `aplay piper_output.wav`.



```
marcelo_roval — mjroval@raspi-5-SD: ~/Documents/OLLAMA/SST — ssh mjroval@raspi-5-SD.local — 87x7
(ollama) mjroval@raspi-5-SD:~/Documents/OLLAMA/SST $ python tts_test.py

Speech generated successfully: piper_output.wav
You can now play the file with: aplay piper_output.wav
(ollama) mjroval@raspi-5-SD:~/Documents/OLLAMA/SST $ aplay piper_output.wav
Playing WAVE 'piper_output.wav' : Signed 16 bit Little Endian, Rate 16000 Hz, Mono
(ollama) mjroval@raspi-5-SD:~/Documents/OLLAMA/SST $
```

Handling Long Text and Special Cases

TTS systems may struggle with very long input text or special characters. Implement text preprocessing to handle these cases gracefully. Break long responses into shorter segments, handle abbreviations and numbers appropriately, and filter out problematic characters that might cause TTS failures.

Consider implementing text chunking for responses longer than a reasonable speaking length. This prevents both TTS processing issues and user fatigue from overly long audio responses.

Pipeline Integration and Optimization

Building the Complete System

Integrating all components requires careful attention to error handling and resource management. Each stage of the pipeline can fail independently, and robust systems must handle these failures gracefully rather than crashing.

Design our integration with modularity in mind. Test each component independently before combining them. This approach simplifies debugging and allows you to optimize individual components separately.

We should also implement proper logging throughout our pipeline. When complex systems fail, detailed logs help identify whether the issue occurs in audio capture, speech recognition, language model processing, text-to-speech conversion, or audio playback.

Performance Optimization Strategies

Measure the timing of each pipeline component to identify bottlenecks. Typically, the SLM inference takes the longest time, followed by TTS generation. Understanding these timing characteristics helps prioritize optimization efforts.

Consider implementing concurrent processing where possible. For example, you might begin TTS processing for the first part of an extended response while the SLM is still generating the remainder. However, be cautious about memory usage when implementing parallel processing on resource-constrained devices.

Memory Management Considerations

Edge devices have limited RAM, and loading multiple large models simultaneously can cause memory pressure. Implement strategies to manage memory efficiently, such as loading models only when needed or using model swapping for infrequently used components.

Monitor system memory usage during operation and implement safeguards to prevent memory exhaustion. Consider implementing graceful degradation where your system switches to smaller, more efficient models if memory becomes constrained.

Designing Resilient Systems

Production-quality voice interaction systems must handle various failure modes gracefully. Network interruptions, hardware disconnections, model loading failures, and unexpected input conditions should not cause your system to crash.

Implement comprehensive error handling at each pipeline stage. When speech recognition produces empty output, provide the user with meaningful feedback rather than processing empty strings. When TTS fails, consider falling back to text display or simplified audio feedback.

Design user feedback mechanisms that work within your voice interaction paradigm. Audio beeps, LED indicators, or simple voice messages can communicate system status without requiring visual displays.

Debugging Complex Pipelines

Multi-stage systems present unique debugging challenges. When the overall system fails, identifying the specific failure point requires systematic testing approaches.

Implement test modes that allow you to inject known inputs at each pipeline stage. This capability enables you to isolate problems to specific components rather than repeatedly testing the entire system.

Create diagnostic outputs that help understand system behavior. For example, displaying transcription confidence scores, SLM response times, or TTS processing status helps identify performance issues or quality problems.

The full Python Script

Considering the previous points, let's assemble all the essential components that work together: audio capture, transcription, language model processing, and text-to-speech. We should combine these into a complete voice pipeline that flows naturally from one step to the next.

The key insight here is that each of our developed scripts represents a stage in what's called an "audio processing pipeline". Let's walk through how we can connect these pieces.

Understanding the Pipeline Architecture

The pipeline follows a logical sequence: the voice becomes audio data, that audio is converted into text, the text is processed into an AI response, the response is converted into speech audio, and finally, that speech audio is transformed into sound that we can hear.

We should have a `run_voice_pipeline()` function in addition to the previous ones that acts as a coordinator, ensuring each step completes successfully before proceeding to the next. If any step fails, the entire pipeline stops gracefully rather than trying to continue with missing data.

Key Integration Points

We should connect the scripts by ensuring the output of each function becomes the input for the following function. For example, `record_audio()` creates "user_input.wav", which `transcribe_audio()` reads to produce text, which `generate_response()` processes to develop an AI response, and so on.

The error handling at each step ensures that if our microphone isn't working, or if the AI model is busy, or if the voice model files are missing, we get clear feedback about what went wrong rather than mysterious crashes.

Optimizations for Raspberry Pi

The Raspberry Pi has limited resources compared to a desktop computer, so we should include several optimizations. The `cleanup_temp_files()` function prevents our storage from filling up with temporary audio files. The audio configuration uses 16kHz sampling (which matches Moonshine's expectations) rather than CD-quality 44kHz, reducing processing overhead.

The continuous assistant mode includes a manual trigger (pressing Enter) rather than **voice activation** detection, which saves CPU cycles that would otherwise be spent constantly monitoring audio input.

On a final product, we could include, for example, a KWS (Keyword Spotting) function, based on a TinyML device, where only when a trigger word is spoken, the `record_audio()` function starts to work.

Understanding Voice Activity Detection

One of the main improvements from the single scripts stacked and tested separately is what audio engineers call “voice activity detection” (VAD). When we speak to someone face-to-face, we don’t announce, “I’m going to talk for exactly 10 seconds now.” Instead, we say our thoughts, pause naturally, and the listener intuitively knows when you’ve finished.

Our new `record_audio_with_silence_detection()` function mimics this natural process by continuously analyzing the audio signal’s amplitude—essentially measuring how “loud” each tiny slice of audio is. When the amplitude stays below a threshold for 2 seconds, for example, the system intelligently concludes that we have finished speaking.

The Technical Magic Behind Silence Detection

The recording process now works like a sophisticated audio surveillance system. Every fraction of a second, it captures a small chunk of audio data (determined by your CHUNK size of 1024 samples) and immediately analyzes it. Using Python’s struct module, it converts the raw byte data into numerical values representing sound pressure levels.

The crucial calculation happens in this line: `volume = max(chunk_data) / 32768.0`. This finds the loudest moment in that tiny audio slice and normalizes it to a scale from 0 to 1, where 0 represents complete silence and 1 represents the maximum possible volume your microphone can capture.

Calibrating the Sensitivity

The `silence_threshold=0.01` parameter is our sensitivity control knob: **too sensitive** (closer to 0.00) and it might stop recording when you pause to think; **not sensitive enough** (closer to 0.1) and it might keep recording through long periods of quiet background noise.

For a typical indoor Raspberry Pi setup, 0.01 strikes a good balance. It’s sensitive enough to detect when we have stopped talking, but robust enough to ignore minor background sounds, such as air conditioning or distant traffic.

We should experiment with this value based on the specific environment.

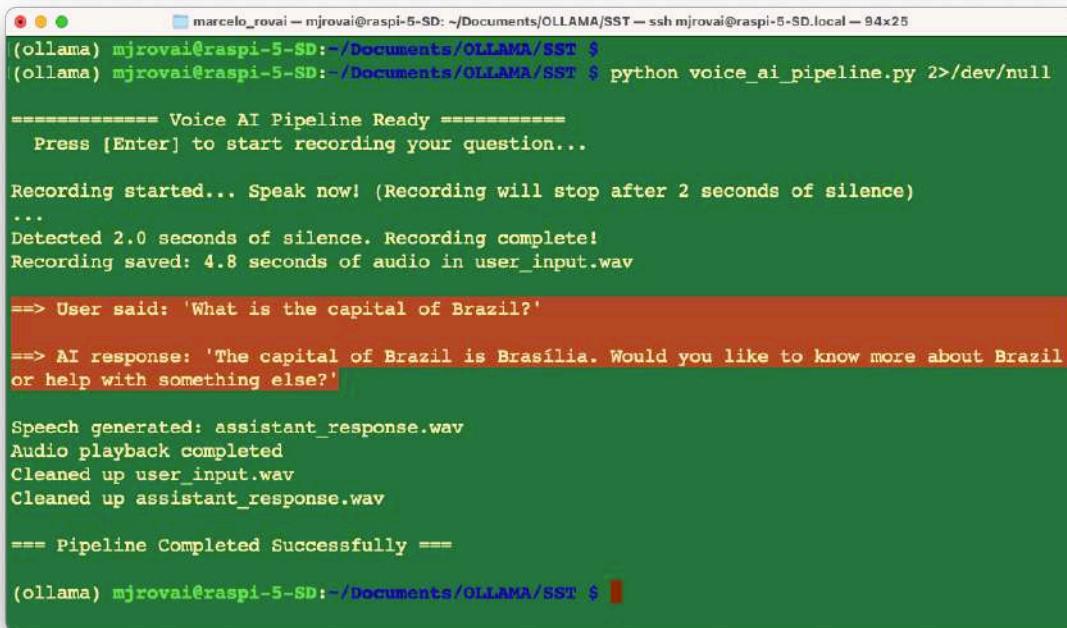
Testing and Deployment Strategy

Download the complete script from GitHub: [voice_ai_pipeline.py](#)

We can start by testing individual components using `detect_audio_devices()` first to confirm our USB microphone is still at index 2. Then, we can run `run_voice_pipeline()` with a simple question to verify the complete flow works.

Once we are confident in single interactions, we can use `continuous_voice_assistant()` for extended conversations. This mode lets us have back-and-forth exchanges with your AI assistant, making it feel more like a natural conversation partner.

If you want to experiment with different SLM models, we need to change the model parameter in `generate_response()`.



The screenshot shows a terminal window titled "marcelo_rovai -> Documents/OLLAMA/SST". The user runs the command "python voice_ai_pipeline.py 2>/dev/null". The pipeline starts recording audio, detects silence, and generates a response. The user's question and the AI's response are highlighted in red boxes. The process ends successfully.

```
marcelo_rovai -> Documents/OLLAMA/SST - ssh mjrovai@raspi-5-SD.local - 94x25
(ollama) mjrovai@raspi-5-SD:~/Documents/OLLAMA/SST $ python voice_ai_pipeline.py 2>/dev/null
=====
Voice AI Pipeline Ready
Press [Enter] to start recording your question...

Recording started... Speak now! (Recording will stop after 2 seconds of silence)
...
Detected 2.0 seconds of silence. Recording complete!
Recording saved: 4.8 seconds of audio in user_input.wav

=> User said: 'What is the capital of Brazil?'
=> AI response: 'The capital of Brazil is Brasília. Would you like to know more about Brazil or help with something else?'
Speech generated: assistant_response.wav
Audio playback completed
Cleaned up user_input.wav
Cleaned up assistant_response.wav

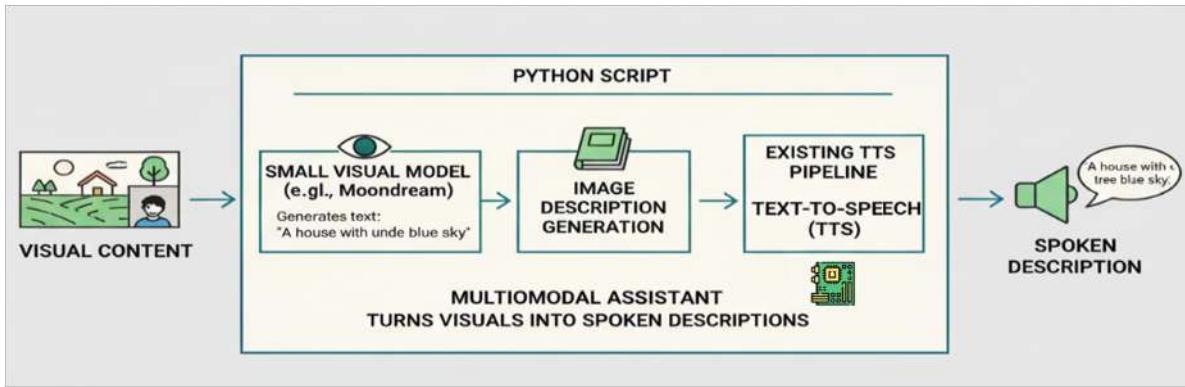
== Pipeline Completed Successfully ==

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

The test with sound can be followed in the video: <https://youtu.be/zgLMYdMWFjg>

The Image to Audio AI Pipeline Architecture

Voice interaction systems have significant potential for educational applications and accessibility improvements by designing interfaces that adapt to different user needs and capabilities. By integrating small visual models, such as Moondream, with existing TTS pipelines, we can create multimodal assistants that describe images for people with visual impairments, converting visual content into detailed spoken descriptions of scenes, objects, and spatial relationships.



To use the camera, we must ensure that the NumPy version is compatible with the Raspberry Pi system (1.24.2). If this version was changed (what it should be due to the Moonshine installation), revert it to the 1.24.2 version.

```
pip uninstall numpy
pip install 'numpy==1.24.2'
```

Now, let's apply what we have explored in previous chapters, along with what we learn in this one, to create a simple code that converts an image captured by a camera into its corresponding spoken caption by the speaker.

The code is straightforward and is intended solely to test the solution's potential.

1. The `capture_image(IMG_PATH)` function captures a photo and saves it to the specified path (IMG_PATH).
2. The `caption = image_description(IMG_PATH, MODEL)` function will describe the image using MoonDream, a powerful yet compact visual language model (VLM). The description in a text format will be saved in the variable `caption`.
3. The `text_to_speech_piper(caption)` function will create a .WAV file from the `caption`.
4. And finally, the `play_audio()` function will play the .WAV file generated by PIPER.

Here is the complete Python script ([img_caption_speech.py](#)):

```
import os
import time
import subprocess
import ollama
from picamera2 import Picamera2
```

```

def capture_image(img_path):
    # Initialize camera
    picam2 = Picamera2()
    picam2.start()

    # Wait for camera to warm up
    time.sleep(2)

    # Capture image
    picam2.capture_file(img_path)
    print("\n==> Image captured: "+img_path)

    # Stop camera
    picam2.stop()
    picam2.close()

def image_description(img_path, model):
    print ("\n==> WAIT, SVL Model working ...")
    with open(img_path, 'rb') as file:
        response = ollama.chat(
            model=model,
            messages=[
                {
                    'role': 'user',
                    'content': '''return the description of the image''',
                    'images': [file.read()],
                },
            ],
            options = {
                'temperature': 0,
            }
        )
    return response['message']['content']

def text_to_speech_piper(text, output_file="assistant_response.wav"):

    # Path to your voice model
    model_path = "voices/en_US-lessac-low.onnx"

```

```

# Check if model exists
if not os.path.exists(model_path):
    print(f"Error: Model file not found at {model_path}")
    return False

try:
    # Run PIPER command
    process = subprocess.Popen(
        ['piper', '--model', model_path, '--output_file', output_file],
        stdin=subprocess.PIPE,
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE,
        text=True
    )

    # Send text to PIPER
    stdout, stderr = process.communicate(input=text)

    if process.returncode == 0:
        print(f"\nSpeech generated successfully: {output_file}")
        return True
    else:
        print(f"Error: {stderr}")
        return False

except Exception as e:
    print(f"Error running PIPER: {e}")
    return False

def play_audio(filename="assistant_response.wav"):
    try:
        # Use aplay to play the audio file
        result = subprocess.run(['aplay', filename],
                               capture_output=True,
                               text=True)

        if result.returncode == 0:
            print("\nAudio playback completed")
            return True
        else:
            print(f"\nPlayback error: {result.stderr}")
            return False
    
```

```

except Exception as e:
    print(f"\nError playing audio: {e}")
    return False

# Example usage and testing functions
if __name__ == "__main__":
    print("\n===== Image to Speech AI Pipeline Ready =====")
    print("Press [Enter] to capture an image and voice caption it ...")

    # Step 1: Wait for user to initiate recording
    input("Press Enter to start ...")

    IMG_PATH = "/home/mjrovai/Documents/OLLAMA/SST/capt_image.jpg"
    MODEL = "moondream:latest"

    capture_image(IMG_PATH)
    caption = image_description(IMG_PATH, MODEL)
    print ("\n==> AI Response:", caption)

    text_to_speech_piper(caption)
    play_audio()

```

And here we can see (and listen) to the result:

```

marcelo_rovai@raspi-5-SD:~/Documents/OLLAMA/SST$ ssh mjrovai@raspi-5-SD.local -l 59x22
(ollama) mjrovai@raspi-5-SD:~/Documents/OLLAMA/SST$ python img_caption_speech.py 2>/dev/null

===== Image to Speech AI Pipeline Ready =====
Press [Enter] to capture an image and voice caption it ...

==> Image captured: /home/mjrovai/Documents/OLLAMA/SST/capt_image.jpg
==> WAIT, SVL Model working ...
==> AI Response:
The image features a bearded man wearing glasses sitting in
an office chair. He is looking up at something, possibly a
map or a picture on the wall behind him. The room has vari
ous objects scattered around, including several books and a
clock on the wall near the center-left side of the scene.

Speech generated successfully: assistant_response.wav
Audio playback completed

```



Your browser does not support the audio element.

Troubleshooting

To work simultaneously with STT and the camera in the same environment, we should ensure that all core scientific packages are version-aligned for the Raspberry Pi environment. To use NumPy 1.24.2 on our Raspberry Pi, we must ensure that both our SciPy and Librosa versions are compatible with that NumPy version.

So, to avoid an eventual `numpy.exceptions` import error, and other possible incompatibilities, we should downgrade scipy (and librosa) to versions that support numpy 1.24.2. According to official compatibility tables, scipy 1.11.x works with numpy 1.24.x. Librosa versions released after 0.9.0 also provide better support for recent NumPy releases. However, some older versions of Librosa are not compatible with NumPy 1.24.2 due to deprecated NumPy attributes. So, to fix it, we should run the lines below:

```
pip uninstall scipy librosa
pip install 'scipy>=1.11,<1.12' 'librosa>=0.10,<0.11'
```

Conclusion

This chapter introduced us to multimodal AI system development through an audio and vision processing pipeline. We learned to integrate speech recognition, language models, and speech synthesis into a cohesive system that runs efficiently on edge hardware.

We explored how to architect systems with multiple AI components, handle complex error conditions, and optimize performance within resource constraints.

These skills are essential for the advanced topics in upcoming chapters, including RAG systems, agent architectures, and the integration of physical computing. The system thinking approach used here will be essential for future AI engineering work.

We should consider how the voice interaction capabilities we built might enhance other AI systems. Many applications benefit from voice interfaces, and the foundation established here can be adapted and extended for various use cases. We can, for example, transform our audio pipeline into a smart home assistant by integrating physical computing elements. Voice commands can trigger LED indicators, read sensor values, or control actuators connected to our Raspberry Pi GPIO pins. Voice queries about environmental conditions can trigger sensor readings, while voice commands can control connected devices.

This chapter extends our SLM and VLM work by adding input and/or output modalities beyond text and images. The same language models we used previously now process voice-derived input and generate responses for speech synthesis.

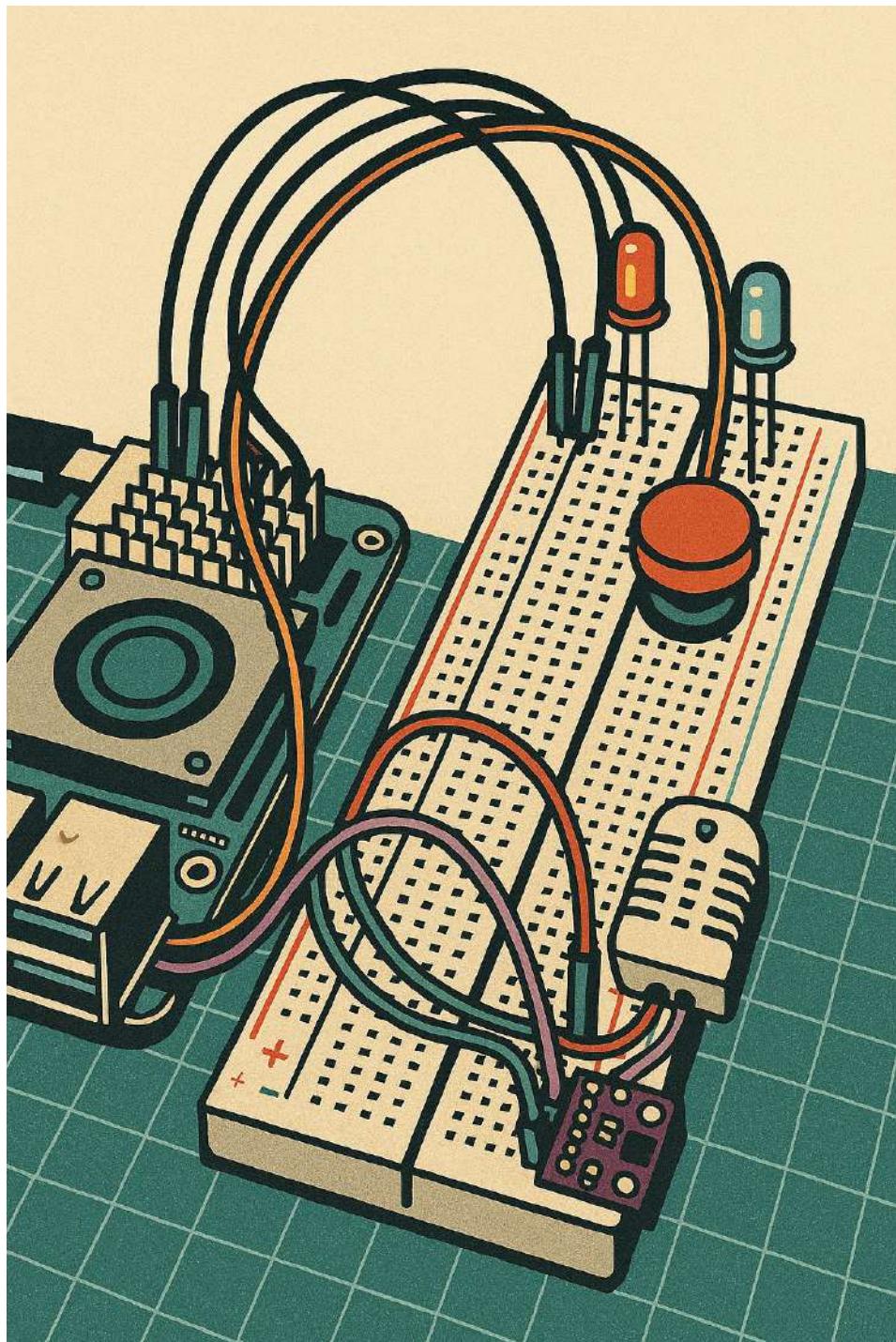
Consider how RAG systems from later chapters might integrate with voice interactions. Voice queries could trigger document retrieval, with synthesized responses incorporating retrieved information.

Resources

[Python Scripts](#)

Physical Computing with Raspberry Pi

From Sensors to Smart Analysis with Small Language Models



Introduction

Physical computing creates interactive systems that sense and respond to the analog world. While this field has traditionally focused on direct sensor readings and programmed responses, we're entering an exciting new era where Large Language Models (LLMs) can add sophisticated decision-making and natural language interaction to physical computing projects.

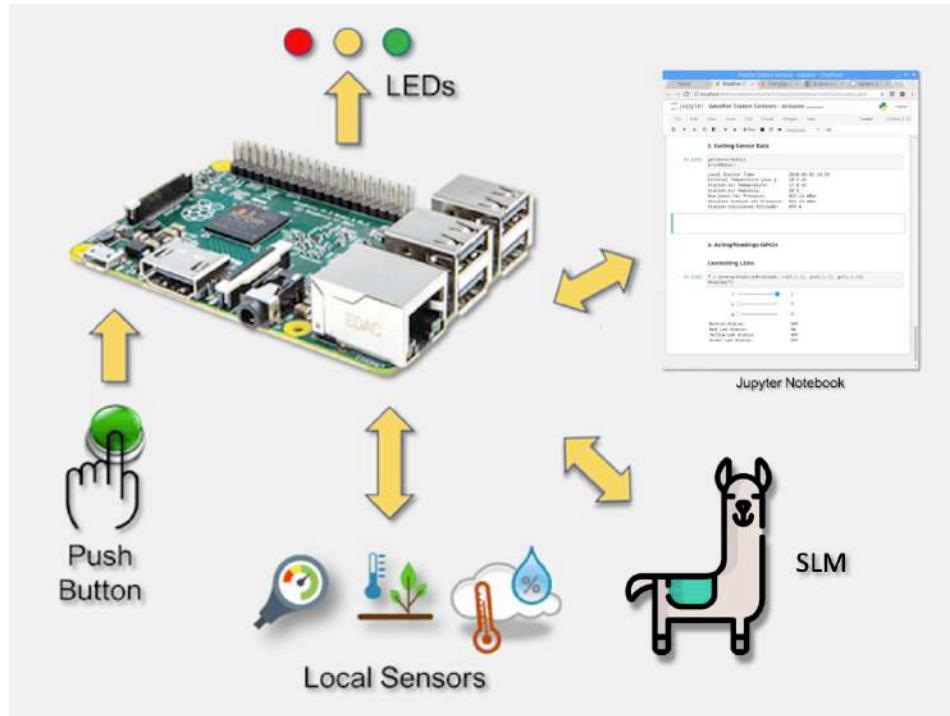
In the *Small Language Models (SLM)* chapter, we learned how to run an LLM (or, more precisely, an SLM) on a Single Board Computer (SBC) such as the Raspberry Pi. In this chapter, we will go through the process of setting up a Raspberry Pi for physical computing, with an eye toward future AI integration. We'll cover:

- Setting up the Raspberry Pi for physical computing
- Working with essential sensors and actuators
- Understanding GPIO (General Purpose Input/Output) programming
- Establishing a foundation for integrating LLMs with physical devices
- Creating interactive systems that can respond to both sensor data and natural language commands

We will also use a Jupyter notebook (programmed in Python) to interact with sensors and actuators—an important and necessary first step toward the goal of integrating the Raspi with an SLM.

The combination of Raspberry Pi's versatility and the power of SLMs opens up exciting possibilities for creating more intelligent and responsive physical computing systems.

The diagram below gives us an overview of the project:



Prerequisites

- Raspberry Pi (model 4 or 5)
- [DHT22 Temperature and Relative Humidity Sensor](#)
- [BMP280 Barometric Pressure, Temperature and Altitude Sensor](#)
- Colored LEDs (3x)
- Push Button (1x)
- Resistor 4K7 ohm (2x)
- Resistor 220 or 330 ohm (3x)

Accessing the GPIOs

The Raspberry Pi's GPIO (General Purpose Input/Output) pins allow us to connect electronic components and control them with Python code. This opens up endless possibilities for creating interactive projects, home automation systems, robotics, and more.

This chapter covers the modern **GPIO Zero library** for interactions with buttons and LEDs.

IMPORTANT: RPi.GPIO does NOT support Raspberry Pi 5!

With the Raspberry Pi 5, we must use GPIO Zero or the newer lgpio library. RPi.GPIO only works on Pi models 1-4, the Pi Zero, Pi Zero 2, and the Pi Zero 2W.

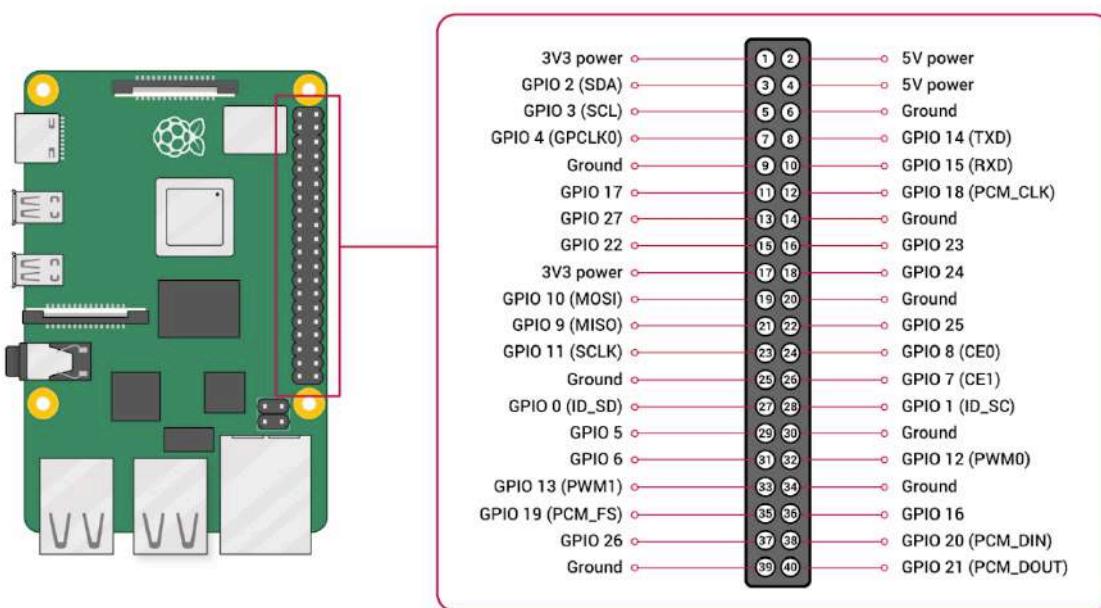
The Raspberry Pi has 40 pins on its header, but not all of them are GPIO pins. Some provide power (3.3V and 5V), others are ground pins, and the rest are programmable GPIO pins.

Pin Numbering Systems

There are two ways to reference GPIO pins:

- **BCM (Broadcom):** Uses the GPIO number (e.g., GPIO17, GPIO27)
- **BOARD:** Uses the physical pin number on the header (e.g., Pin 11, Pin 13)

In this chapter, we'll use BCM numbering as it's more commonly used in Python programming.



Safety First

Before connecting any components, please read these important safety guidelines:

- **Never connect 5V directly to GPIO pins** - GPIO pins are 3.3V tolerant only
- **Always use current-limiting resistors with LEDs** - Without them, you risk damaging the LED or GPIO pin
- **Double-check your connections** before powering on
- **Disconnect power** when making circuit changes
- **Respect polarity** - LEDs for example, have positive (long leg) and negative (short leg) sides

GPIO Zero Library

A modern, high-level library that makes GPIO programming much simpler and more intuitive. It uses object-oriented programming and includes built-in features like automatic pin cleanup, device abstraction, and event detection.

It is essential to note that the [GPIO Zero Library](#) uses Broadcom (BCM) pin numbering for GPIO pins, rather than physical (board) numbering. Any pin marked “GPIO” in the previous diagram can be used as a PIN. For example, if an LED were attached to GPIO13, we would specify the PIN as 13 rather than 33 (the physical one).

It was created by Ben Nuttall of the Raspberry Pi Foundation, Dave Jones, and other contributors ([GitHub](#)).

Advantages of GPIO Zero:

- Simpler, more readable code
- Object-oriented approach (LED, Button, etc.)
- Built-in device patterns and behaviors
- Automatic cleanup on program exit
- Better for beginners and rapid prototyping
- Includes composite devices (robots, traffic lights, etc.)

Installation:

- GPIO Zero comes pre-installed on Raspberry Pi OS!

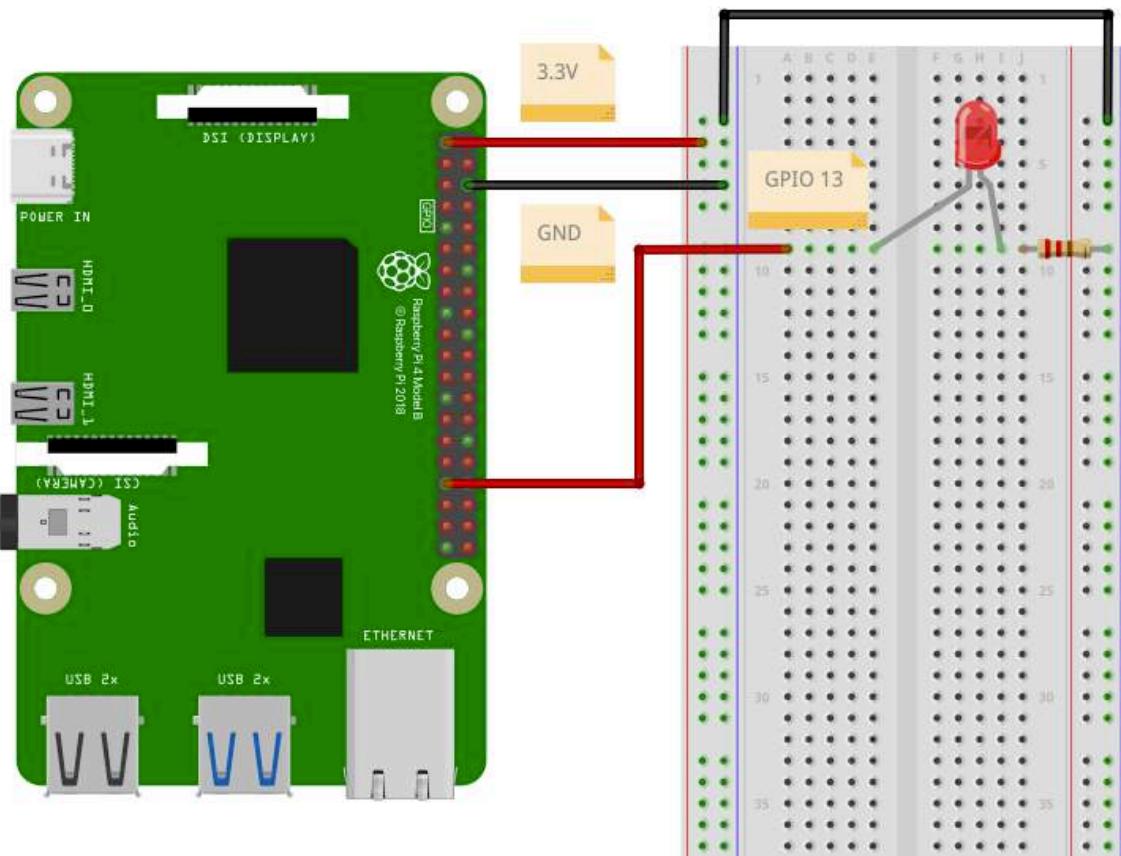
“Hello World”: Blinking an LED

Let's start with the classic ‘Hello World’ of physical computing - making an LED blink!

To connect our RPi to the world, let's first connect:

- Physical Pin 6 (GND) to GND Breadboard Power Grid (Blue -), using a black jumper
- Physical Pin 1 (3.3V) to +VCC Breadboard Power Grid (Red +), using a red jumper

Now, let's connect an LED (red) using the physical pin 33 (GPIO13) connected to the LED cathode (longer LED leg). Connect the LED anode to the breadboard GND using a 220 ohms resistor to reduce the current drawn from the Raspberry Pi, as shown below:



Understanding the LED Circuit

An LED (Light-Emitting Diode) requires current to flow through it to produce light. However, without a resistor, too much current can flow, damaging the LED or your Raspberry Pi. We use a resistor to limit the current to a safe level.

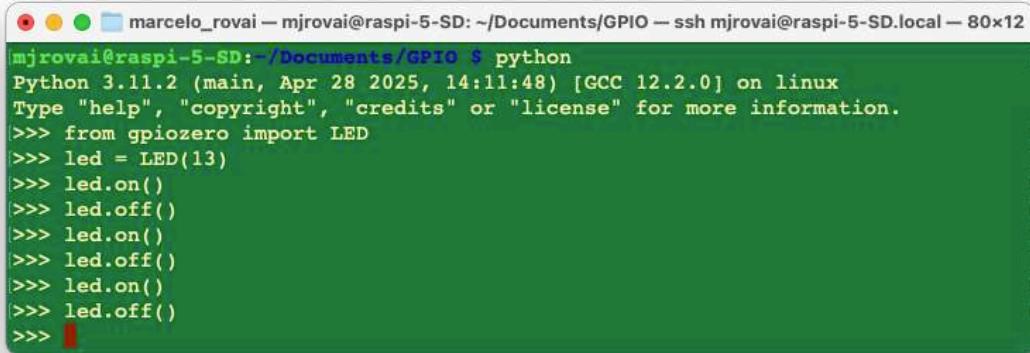
Why 220Ω or 330Ω Resistors?

These values limit the current to approximately 10-15mA, which is safe for most standard LEDs and GPIO pins. The exact value isn't critical—anything from $220\ \Omega$ to $1\ k\Omega$ will work fine.

Testing with GPIO Zero

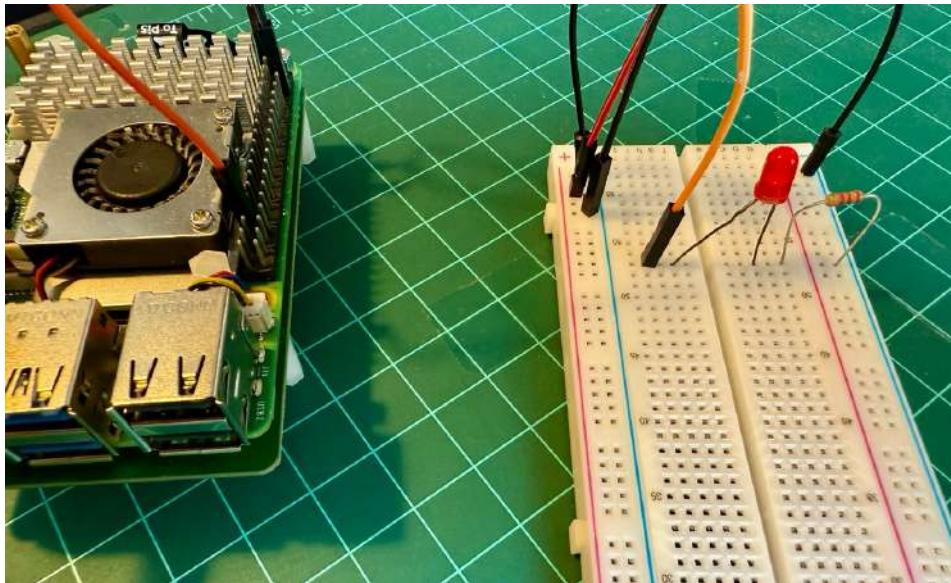
We can use the built-in Python interpreter to test the LED. In the terminal, enter `python`, and once in the interpreter, enter with the commands below:

```
python
>>> from gpiozero import LED
>>> led = LED(13)
>>> led.on()
>>> led.off()
```



A screenshot of a terminal window titled "marcelo_ravai — mjrovai@raspi-5-SD: ~/Documents/GPIO — ssh mjrovai@raspi-5-SD.local — 80x12". The window shows the following Python session:

```
mjrovai@raspi-5-SD:~/Documents/GPIO$ python
Python 3.11.2 (main, Apr 28 2025, 14:11:48) [GCC 12.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from gpiozero import LED
>>> led = LED(13)
>>> led.on()
>>> led.off()
>>> led.on()
>>> led.off()
>>> led.on()
>>> led.off()
>>> 
```



On the Raspberry, start at home and go to Documents.

```
cd Documents
```

Create a directory to save the scripts and install the libraries. Move to there:

```
mkdir GPIO  
cd GPIO
```

We can use any text editor (such as Nano) to create and run the script. Save the file, for example, as `led_test.py`, and then execute it using the terminal:

```
python led_test.py
```

Now, let's blink the LED (the actual "Hello world") when talking about physical computing. To do that, we must also import another library: `time`. We need it to define how long the LED will be ON and OFF. In the case below, the LED will blink every 1 second.

```
from gpiozero import LED  
from time import sleep  
led = LED(13)  
while True:  
    led.on()  
    sleep(1)  
    led.off()  
    sleep(1)
```

We can use any text editor (such as Nano) to create and run the script. Save the file, for example, as `blink.py`, and then execute it using the terminal:

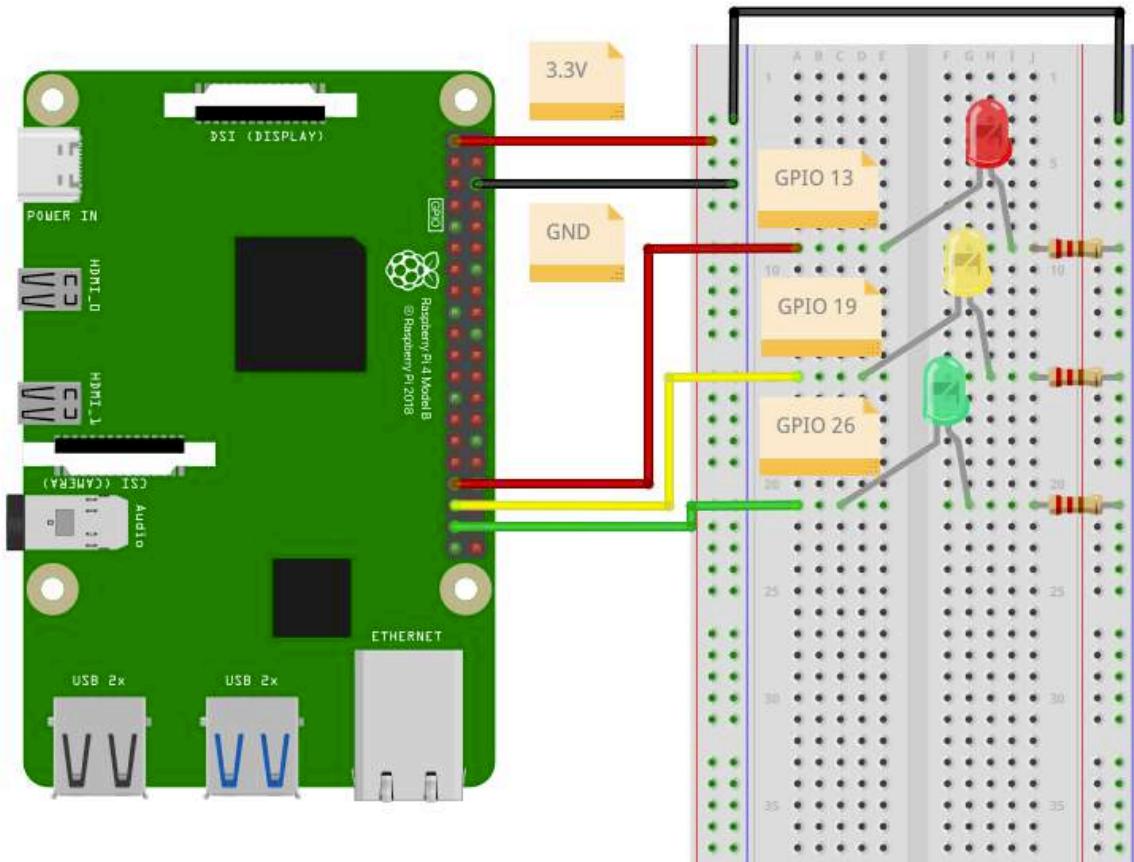
```
python blink.py
```

Alternatively, we can reduce the blink code as below:

```
from gpiozero import LED
from signal import pause
led = LED(13)
led.blink() # 1 second by default
# led.blink(on_time=0.5, off_time=0.5) # Fast blink
pause()
```

Installing all LEDs (the “actuators”)

The LEDs can be used as “actuators”; depending on the condition of a code running on our Pi, we can command one of the LEDs to fire! We will install two more LEDs, in addition to the red one already installed. Follow the diagram and install the yellow (on GPIO 19) and the green (on GPIO 26).



For testing we can run a similar code as the used with the single red led, changing the pin accordantly, for example.

```

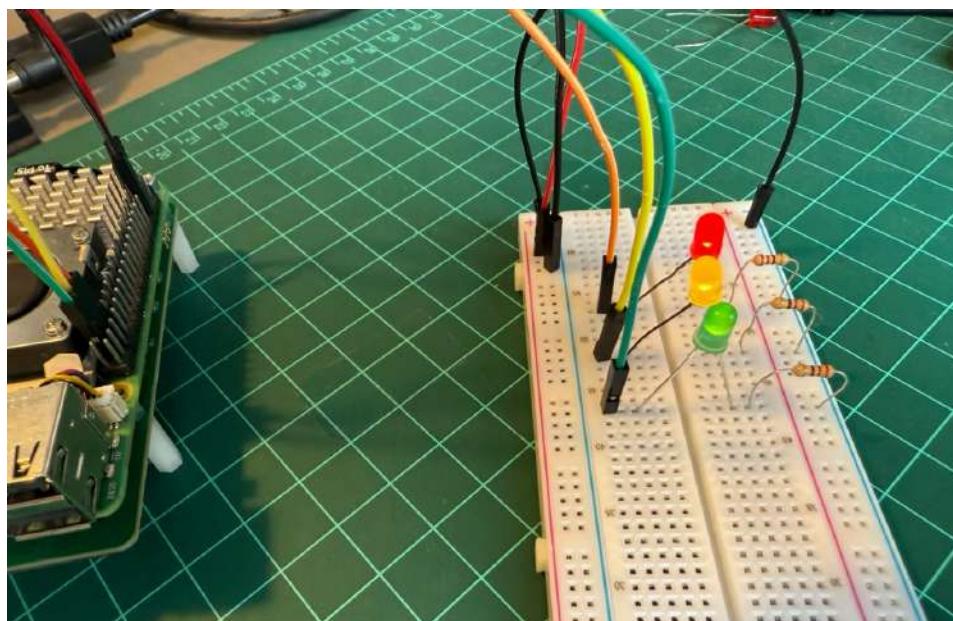
from gpiozero import LED
from time import sleep

ledRed = LED(13)
ledYlw = LED(19)
ledGrn = LED(26)

ledRed.off()
ledYlw.off()
ledGrn.off()

ledRed.on()
ledYlw.on()
    
```

```
ledGrn.on()  
  
sleep(5)  
  
ledRed.off()  
ledYlw.off()  
ledGrn.off()
```



Remember that instead of LEDs, we could have relays, motors, etc.

Sensors Installation and setup

In this section, we will setup the Raspberry Pi to capture data from several different sensors:

Sensors and Communication type:

- **Button** (Command via a Push-Button) ==> Digital direct connection
- **DHT22** (Temperature and Humidity) ==> Digital communication
- **BMP280** (Temperature and Pressure) ==> I2C Protocol

Button

Now, let's learn how to read input from a button. This allows your Raspberry Pi to respond to physical interactions!

Understanding Pull-Up and Pull-Down Resistors

When a button is not pressed, the GPIO pin is 'floating' - it's not connected to anything and can read random values. We use pull-up or pull-down resistors to set the pin's default state.

- **Pull-Down:** Default state is LOW (0V), becomes HIGH when button pressed
- **Pull-Up:** Default state is HIGH (3.3V), becomes LOW when button pressed

The Raspberry Pi has internal pull-up and pull-down resistors!

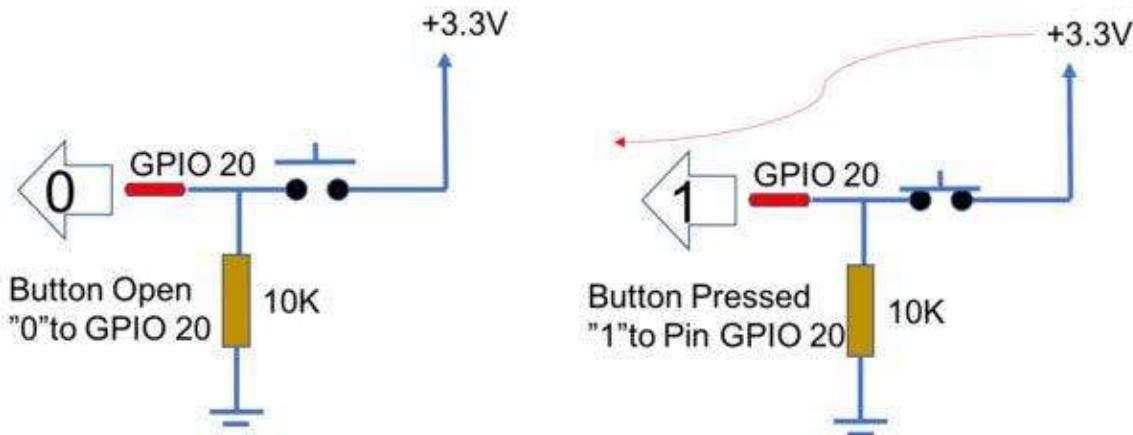
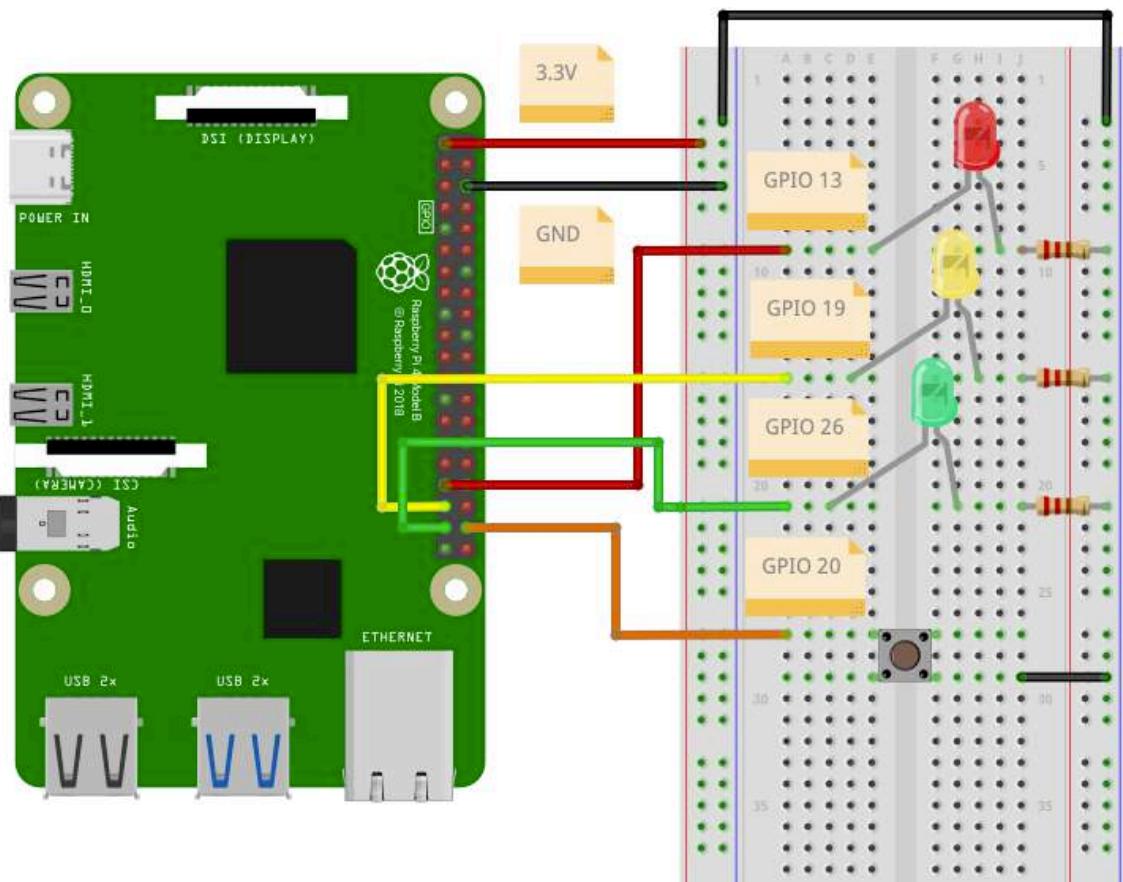


Figure 14: img

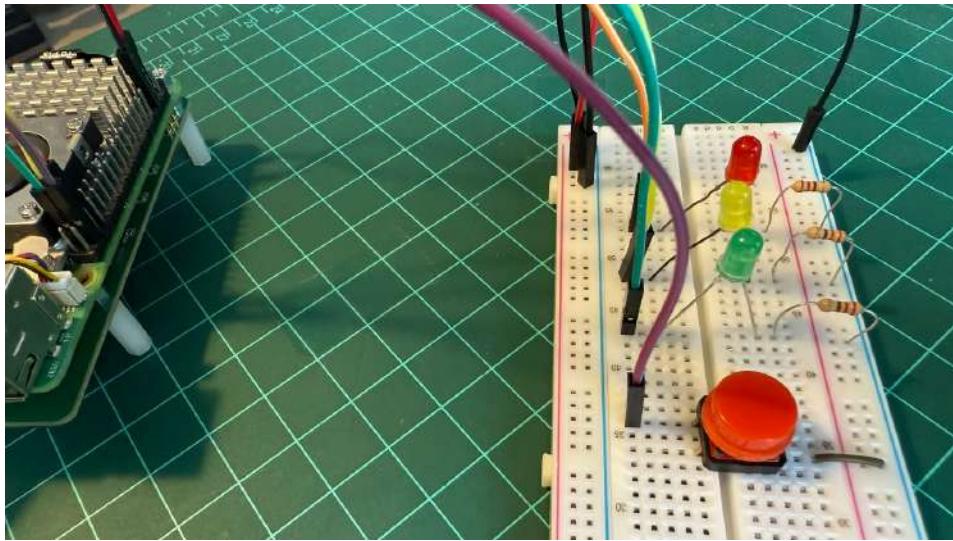
The simplest way to run an external command is with a push button, and the GPIO Zero Library makes it easy to include in the project. We do not need to think about Pull-up or Pull-down resistors, etc. In terms of HW, the only thing to do is to connect one leg of our push-button to any one of the Raspi GPIOs and the other one to GND, as shown in the diagram:



- Push-Button leg1 to GPIO 20
- Push-Button leg2 to GND

A simple code for reading the button can be:

```
from gpiozero import Button
button = Button(20)
while True:
    if button.is_pressed:
        print("Button is pressed")
    else:
        print("Button is not pressed")
```



On a Raspberry Pi Zero 2W, for example, we could use the `RPi.GPIO`, whose code is:

```
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)

BUTTON_PIN = 20

# Set up with internal pull-up
GPIO.setup(BUTTON_PIN, GPIO.IN, pull_up_down=GPIO.PUD_UP)

try:
    while True:
        if GPIO.input(BUTTON_PIN) == GPIO.LOW:
            print('Button pressed!')
        else:
            print('Button not pressed')
        time.sleep(0.1)

except KeyboardInterrupt:
    GPIO.cleanup()
```

Installing Adafruit CircuitPython

The GPIO Zero library is an excellent hardware interfacing library for Raspberry Pi. It's great for digital in/out, analog inputs, servos, basic sensors, etc. However, it doesn't cover SPI/I2C sensors or drivers. By using CircuitPython via [adafruit_blinka](#), we can take advantage of all of the drivers and example code developed by Adafruit!

Note that we will keep using GPIO Zero for pins, buttons, and LEDs.

Enable Interfaces

Run these commands to enable the various interfaces such as I2C and SPI:

```
sudo raspi-config nonint do_i2c 0  
sudo raspi-config nonint do_spi 0  
sudo raspi-config nonint do_serial_hw 0  
sudo raspi-config nonint disable_raspi_config_at_boot 0
```

Install required dependencies

```
sudo apt-get install -y python3-libgpiod i2c-tools libgpiod-dev
```

Install Blinka

Let's enter the Ollama environment, already created to install Blinka:

```
source ~/ollama/bin/activate
```

Install the library

```
pip install adafruit-blinka
```

Check I2C and SPI

The script will automatically enable I2C and SPI. You can run the following command to verify:

```
ls /dev/i2c* /dev/spi*
```

```
marcelo_rovai — mjrovai@raspi-5-SD: ~/Documents/GPIO — ssh mjrovai@raspi-5-SD.local — 74x5
(ollama) mjrovai@raspi-5-SD:~/Documents/GPIO $ ls /dev/i2c* /dev/spi*
/dev/i2c-1  /dev/i2c-13  /dev/i2c-6  /dev/spidev0.1
/dev/i2c-10  /dev/i2c-14  /dev/spidev0.0  /dev/spidev10.0
(ollama) mjrovai@raspi-5-SD:~/Documents/GPIO $
(ollama) mjrovai@raspi-5-SD:~/Documents/GPIO $
```

Verify Blinka Version

```
python3 -m pip show adafruit-blinka
```

```
marcelo_rovai — mjrovai@raspi-5-SD: ~/Documents/GPIO — ssh mjrovai@raspi-5-SD.local — 86x16
(ollama) mjrovai@raspi-5-SD:~/Documents/GPIO $ python3 -m pip show adafruit-blinka
Name: Adafruit-Blinka
Version: 8.66.2
Summary: CircuitPython APIs for non-CircuitPython versions of Python such as CPython on Linux and MicroPython.
Home-page: https://github.com/adafruit/Adafruit_Blinka
Author: Adafruit Industries
Author-email: circuitpython@adafruit.com
License: MIT
Location: /home/mjrovai/ollama/lib/python3.11/site-packages
Requires: Adafruit-Blinka-Raspberry-Pi5-Neopixel, adafruit-circuitpython-typing, Adafruit-PlatformDetect, Adafruit-PureIO, binho-host-adapter, lgpio, pyftdi, RPi.GPIO, rpi_ws281x, sysv_ipc
Required-by: adafruit-circuitpython-busdevice, adafruit-circuitpython-connectionmanager, adafruit-circuitpython-requests, adafruit-circuitpython-typing
(ollama) mjrovai@raspi-5-SD:~/Documents/GPIO $
```

Blinka Installation Test

Create a new file called **blinka_test.py** with **nano** or your favorite text editor and put the following in:

```
import board
import digitalio
import busio

print("Hello, blinka!")
```

```

# Try to create a Digital input
pin = digitalio.DigitalInOut(board.D4)
print("Digital IO ok!")

# Try to create an I2C device
i2c = busio.I2C(board.SCL, board.SDA)
print("I2C ok!")

# Try to create an SPI device
spi = busio.SPI(board.SCLK, board.MOSI, board.MISO)
print("SPI ok!")

print("done!")

```

Save it and run it at the command line:

```
python blinka_test.py
```

```

marcelo_rovai — mjrovai@raspi-5-SD: ~/Documents/GPIO — ssh mjrovai@raspi-5-SD.local — 71x8
(ollama) mjrovai@raspi-5-SD:~/Documents/GPIO $ nano blinka_test.py
(ollama) mjrovai@raspi-5-SD:~/Documents/GPIO $ python blinka_test.py
Hello, blinka!
Digital IO ok!
I2C ok!
SPI ok!
done!
(ollama) mjrovai@raspi-5-SD:~/Documents/GPIO $ 
```

DHT22 - Temperature & Humidity Sensor

The first sensor to be installed will be the DHT22 for capturing air temperature and relative humidity data.

Overview

The low-cost DHT temperature and humidity sensors are elementary and slow, but great for logging basic data. They consist of a capacitive humidity sensor and a thermistor. A bare chip inside performs the analog-to-digital conversion and outputs a digital signal containing

the temperature and humidity. The digital signal is relatively easy to read using any microcontroller.

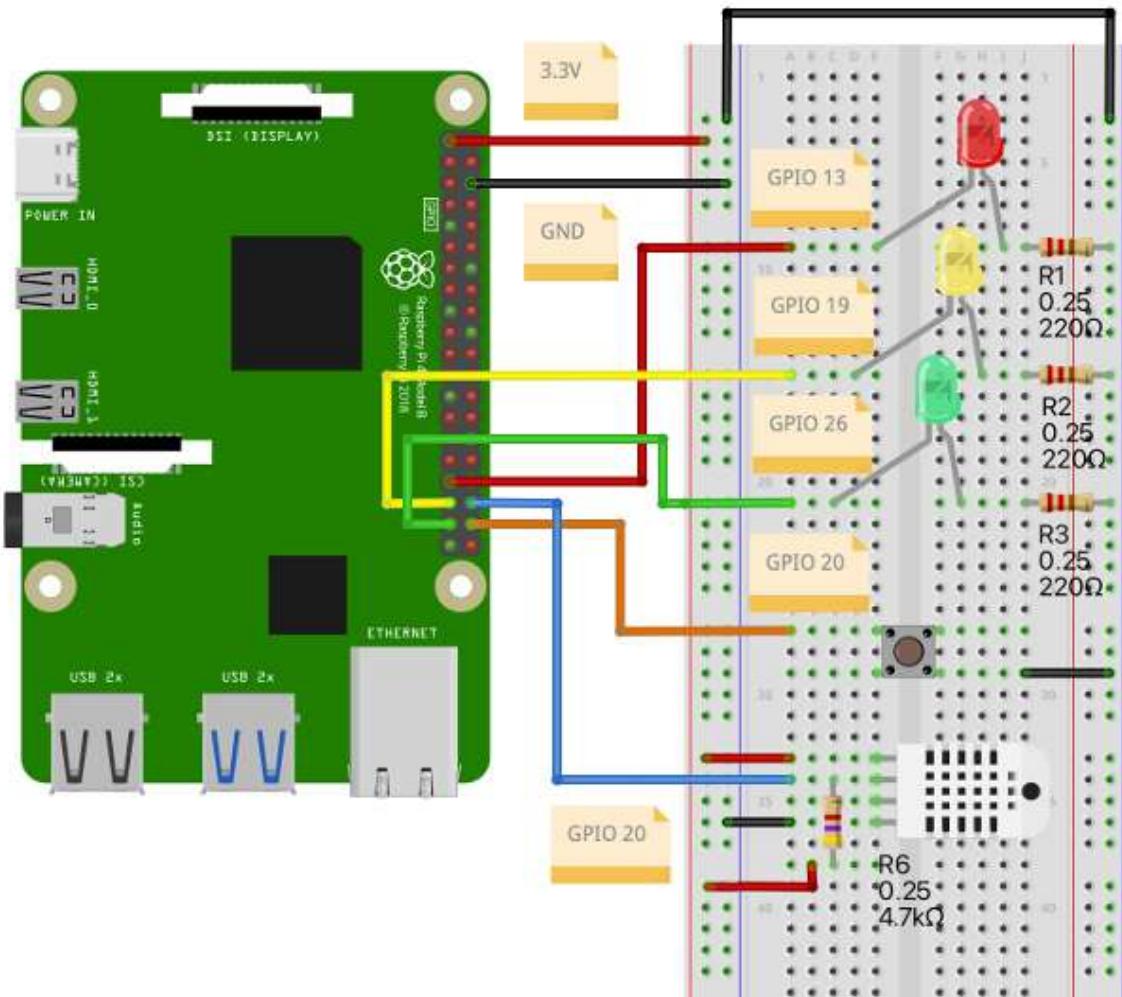
DHT22 Main characteristics:

- Suitable for 0-100% humidity readings with 2-5% accuracy
- Suitable for -40 to 125°C temperature readings $\pm 0.5^\circ\text{C}$ accuracy
- No more than 0.5 Hz sampling rate (once every 2 seconds)
- Low cost
- 3 to 5V power and I/O
- 2.5mA max current use during conversion (while requesting data)
- Body size 15.1mm x 25mm x 7.7mm
- 4 pins with 0.1" spacing

Once we use the sensor at distances less than 20m, a 4K7 ohm resistor should be connected between the **Data** and **VCC** pins. The DHT22 output data pin will be connected to Raspberry GPIO 16. Check the electrical diagram, connecting the sensor to RPi pins as below:

- Pin 1 - Vcc ==> 3.3V
- Pin 2 - Data ==> GPIO 16
- Pin 3 - Not Connect
- Pin 4 - Gnd ==> Gnd

Do not forget to Install the 4K7 ohm resistor between the VCC and Data pins.



Once the sensor is connected, we must install its library on our Raspberry Pi. First, we should install the Adafruit CircuitPython library, which we have already done, and the [Adafruit_CircuitPython_DHT](#).

```
pip install adafruit-circuitpython-dht
```

Create a new Python script as below and name it, for example, `dht_test.py`:

```
import time
import board
import adafruit_dht
dhtDevice = adafruit_dht.DHT22(board.D16)
```

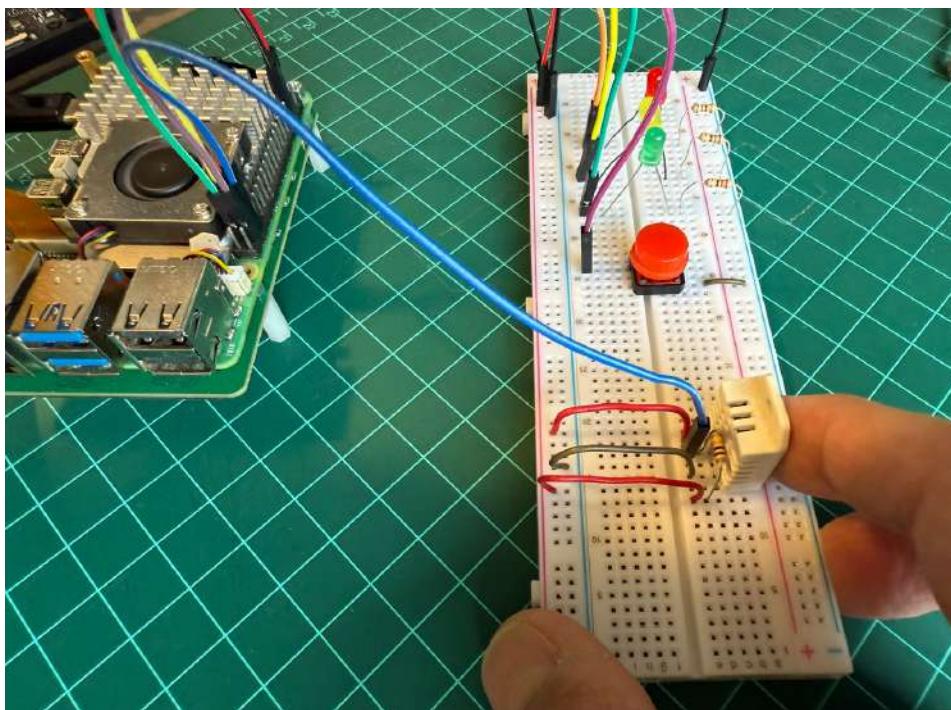
```

while True:
    try:
        # Print the values to the serial port
        temperature_c = dhtDevice.temperature
        temperature_f = temperature_c * (9 / 5) + 32
        humidity = dhtDevice.humidity
        print(
            "Temp: {:.1f} F / {:.1f} C    Humidity: {}% ".format(
                temperature_f, temperature_c, humidity
            )
        )
        time.sleep(2.0)

    except RuntimeError as error:
        # Errors happen fairly often, DHT's are hard to read,
        # just keep going
        print(error.args[0])
        time.sleep(2.0)
        continue
    except Exception as error:
        dhtDevice.exit()
        raise error

```

Placing a finger on the sensor, we can see that both temperature and humidity begin to rise.



```

marcelo_rovai - mjrovai@raspi-5-SD: ~/Documents/GPIO -- ssh mjrovai@raspi-5-SD.local -- 83x11
(olllama) mjrovai@raspi-5-SD:~/Documents/GPIO$ nano dht_test.py
(olllama) mjrovai@raspi-5-SD:~/Documents/GPIO$ python dht_test.py
Temp: 72.0 F / 22.2 C    Humidity: 36.8%
Temp: 70.9 F / 21.6 C    Humidity: 37.2%
Temp: 70.7 F / 21.5 C    Humidity: 37.2%
Temp: 70.7 F / 21.5 C    Humidity: 37.2%
Temp: 70.7 F / 21.5 C    Humidity: 37.2%
Temp: 70.9 F / 21.6 C    Humidity: 47.5%
Temp: 71.8 F / 22.1 C    Humidity: 61.4%
Temp: 73.2 F / 22.9 C    Humidity: 60.3%
Temp: 74.7 F / 23.7 C    Humidity: 56.0%

```

Installing the BMP280: Barometric Pressure & Altitude Sensor

Sensor Overview:

Environmental sensing has become increasingly important in various industries, from weather forecasting to indoor navigation and consumer electronics. At the forefront of this technological advancement are sensors like the BMP280 and BMP180 (deprecated), which excel in measuring temperature and barometric pressure with exceptional precision and reliability.

As its predecessor, the BMP180, the [BMP280](#) is an absolute barometric pressure sensor, which is especially feasible for mobile applications. Its diminutive dimensions and low power consumption allow for its implementation in battery-powered devices such as mobile phones, GPS modules, or watches. The BMP280 is based on Bosch's proven piezo-resistive pressure sensor technology featuring high accuracy and linearity as well as long-term stability and high EMC robustness. Numerous device operation options guarantee the highest flexibility. The device is optimized for power consumption, resolution, and filter performance.

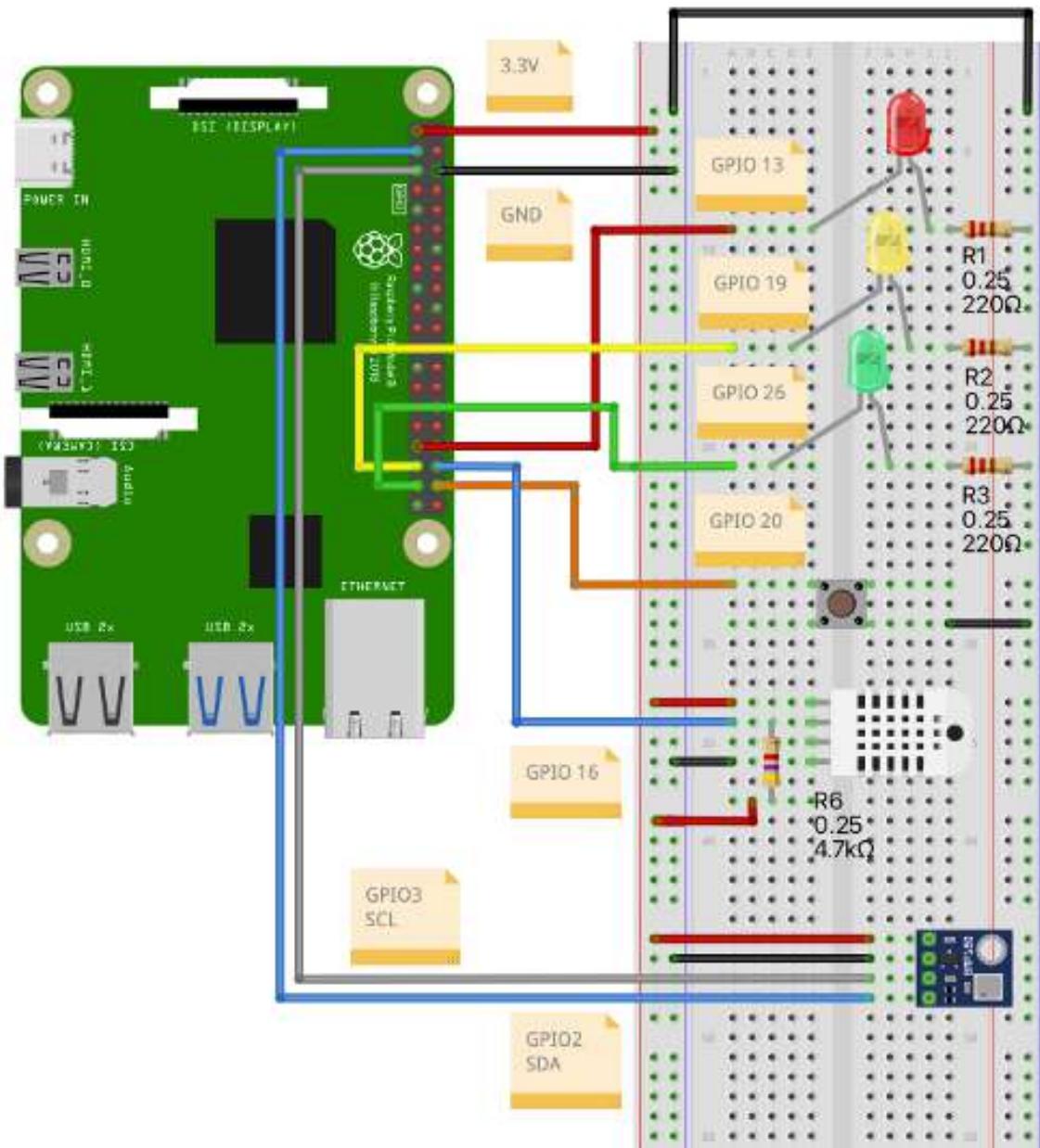
Technical data

Parameter	Technical data
Operation range	Pressure: 300...1100 hPa Temp.: -40...85°C ~ ±1 hPa
Absolute accuracy (950...1050 hPa, 0...+40°C)	± 0.12 hPa (typical) equivalent to ±1 m
Relative accuracy $p = 700\ldots 900\text{hPa}$ (Temp. @ 25°C)	
Average typical current consumption (1 Hz dt/rate)	3.4 A @ 1 Hz
Average current consumption (1 Hz dt refresh rate)	2.74 A, typical (ultra-low power mode)
Average current consumption in sleep mode	0.1 A
Average measurement time	5.5 msec (ultra-low power preset)
Supply voltage VDDIO	1.2 ... 3.6 V
Supply voltage VDD	1.71 ... 3.6 V
Resolution of data	Pressure: 0.01 hPa ($< 10 \text{ cm}$) Temp.: 0.01° C 1.5 Pa/K, equiv. to 12.6 cm/K
Temperature coefficient offset (+25°...+40°C @ 900hPa)	
Interface	I ² C and SPI

BMP280 Sensor Installation

Follow the diagram and make the connections:

- Vin ==> 3.3V
- GND ==> GND
- SCL ==> GPIO 3
- SDA ==> GPIO 2



Enabling I2C Interface

Go to RPi Configuration and confirm that the I2C interface is enabled. If not, enable it.

```
sudo raspi-config nonint do_i2c 0
```

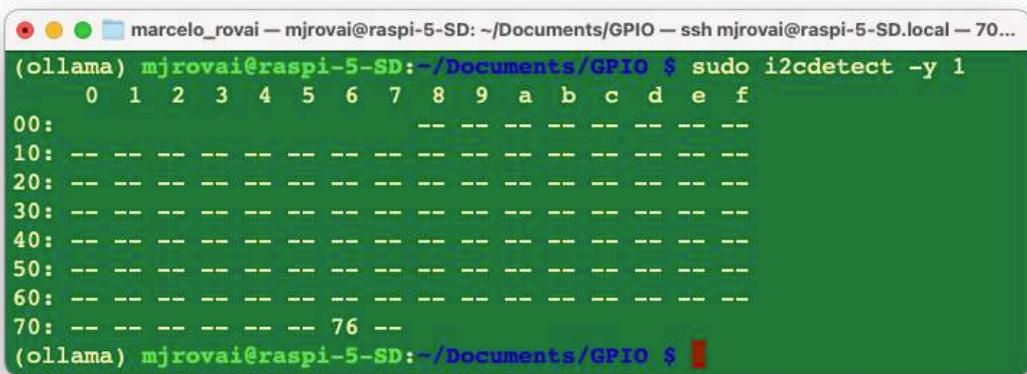
Using the BMP280

If everything has been installed and connected correctly, you can turn on your Rapspi and start interpreting the BMP180's information about the environment.

The first thing to do is to check if the Raspi sees your BMP280. Try the following in a terminal:

```
sudo i2cdetect -y 1
```

We should confirm that the BMP280 is on channel 77 (default) or 76.



```
(ollama) mjrovai@raspi-5-SD:~/Documents/GPIO $ sudo i2cdetect -y 1
      0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00: --
10: --
20: --
30: --
40: --
50: --
60: --
70: -- 76 --
(ollama) mjrovai@raspi-5-SD:~/Documents/GPIO $
```

In my case, the bus address is 0x76, so we should define it in the code.

Installing the BMP 280 Library:

Once the sensor is connected, we must install its library on our Raspi. For that, we should install the [Adafruit_CircuitPython_BMP280](#).

```
pip install adafruit-circuitpython-bmp280
```

Create a new Python script as below and name it, for example, `bmp280_test.py`:

```
import time
import board

import adafruit_bmp280

i2c = board.I2C()
bmp280 = adafruit_bmp280.Adafruit_BMP280_I2C(i2c, address = 0x76)
```

```
bmp280.sea_level_pressure = 1013.25

while True:
    print("\nTemperature: %0.1f C" % bmp280.temperature)
    print("Pressure: %0.1f hPa" % bmp280.pressure)
    print("Altitude = %0.2f meters" % bmp280.altitude)
    time.sleep(2)
```

Execute the script:

```
python bmp280Test.py
```

The Terminal shows the result.

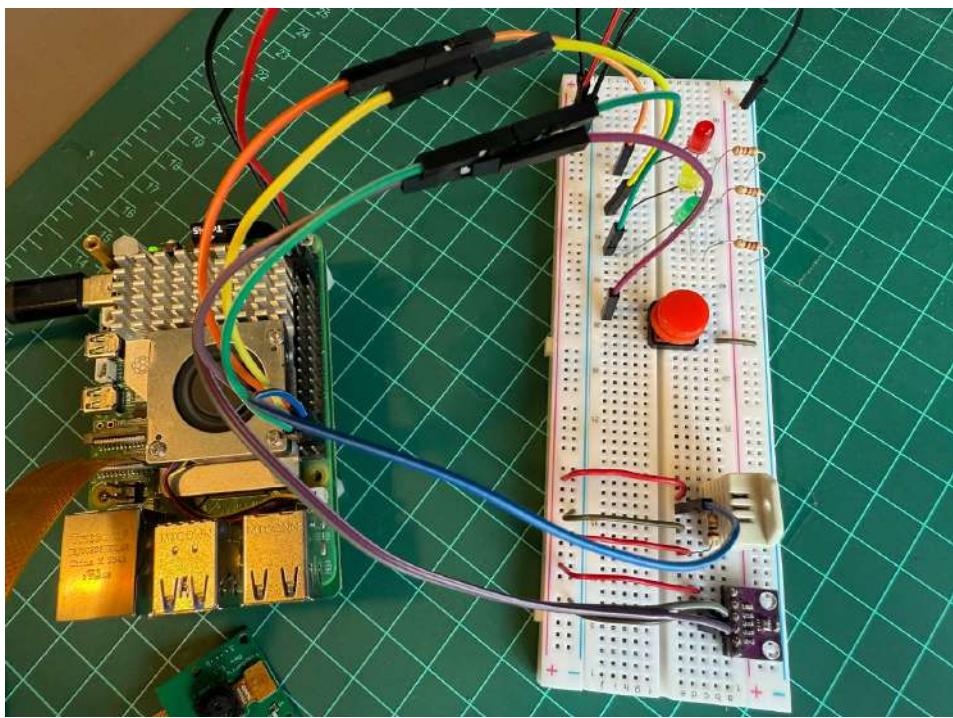


The screenshot shows a terminal window with a dark background and light-colored text. At the top, there are three small colored icons (red, yellow, green) followed by the text "marcelo_rovai — mjrovai@raspi-5-SD: ~/Documents/GPIO — ssh mjrovai@raspi-5-SD.local — 70...". Below this, two command-line entries are shown in blue: "(ollama) mjrovai@raspi-5-SD:~/Documents/GPIO \$ nano bmp280_test.py" and "(ollama) mjrovai@raspi-5-SD:~/Documents/GPIO \$ python bmp280_test.py". The output of the script is displayed in white text:
Temperature: 22.7 C
Pressure: 908.6 hPa
Altitude = 910.07 meters

Temperature: 22.7 C
Pressure: 908.6 hPa
Altitude = 909.57 meters

Note that pressure is presented in hPa. See the next section to better understand this unit.

Below is the complete setup for our SLM tests.



Measuring Weather and Altitude With BMP280



Let's take some time to understand more about what we will get with the BMP readings.

You can skip this part of the tutorial, or return later.

The BMP280 (and its predecessor, the BMP180) was designed to measure atmospheric pressure accurately. Atmospheric pressure varies with both weather and altitude.

What is Atmospheric Pressure?

Atmospheric pressure is a force that the air around you exerts on everything. The weight of the gasses in the atmosphere creates atmospheric pressure. A standard unit of pressure is “pounds per square inch” or psi. We will use the international notation, newtons per square meter, called pascals (Pa).

If you took 1 cm wide column of air would weigh about 1 kg

This weight, pressing down on the footprint of that column, creates the atmospheric pressure that we can measure with sensors like the BMP280. Because that cm-wide column of air weighs about 1 kg, the average sea level pressure is about 101,325 pascals, or better, 1013.25 hPa (1 hPa is also known as milibar - mbar). This will drop about 4% for every 300 meters you ascend. The higher you get, the less pressure you’ll see because the column to the top of the atmosphere is much shorter and weighs less. This is useful because you can determine your altitude by measuring the pressure and doing math.

The air pressure at 3,810 meters is only half that at sea level.

The BMP280 outputs absolute pressure in hPa (mbar). One pascal is a minimal amount of pressure, approximately the amount that a sheet of paper will exert resting on a table. You will often see measurements in hectopascals (1 hPa = 100 Pa). The library here provides outputs of floating-point values in hPa, equaling one millibar (mbar).

Here are some conversions to other pressure units:

- 1 hPa = 100 Pa = 1 mbar = 0.001 bar
- 1 hPa = 0.75006168 Torr
- 1 hPa = 0.01450377 psi (pounds per square inch)
- 1 hPa = 0.02953337 inHg (inches of mercury)
- 1 hPa = 0.00098692 atm (standard atmospheres)

Temperature Effects

Because temperature affects the density of a gas, density affects the mass of a gas, and mass affects the pressure (whew), atmospheric pressure will change dramatically with temperature. Pilots know this as “density altitude”, which makes it easier to take off on a cold day than a hot one because the air is denser and has a more significant aerodynamic effect. To compensate for temperature, the BMP280 includes a rather good temperature sensor and a pressure sensor.

To perform a pressure reading, you first take a temperature reading, then combine that with a raw pressure reading to come up with a final temperature-compensated pressure measurement. (The library makes all of this very easy.)

Measuring Absolute Pressure

If your application requires measuring absolute pressure, all you have to do is get a temperature reading, then perform a pressure reading (see the test script for details). The final pressure reading will be in hPa = mbar. You can convert this to a different unit using the above conversion factors.

Note that the absolute pressure of the atmosphere will vary with both your altitude and the current weather patterns, both of which are useful things to measure.

Weather Observations

The atmospheric pressure at any given location on Earth (or anywhere with an atmosphere) isn't constant. The complex interaction between the earth's spin, axis tilt, and many other factors result in moving areas of higher and lower pressure, which in turn cause the variations in weather we see every day. By watching for changes in pressure, you can predict short-term changes in the weather. For example, dropping pressure usually means wet weather or a storm is approaching (a low-pressure system is moving in). Rising pressure usually means clear weather is coming (a high-pressure system is moving through). But remember that atmospheric pressure also varies with altitude. The absolute pressure in my home, Lo Barnechea, in Chile (altitude 960m), will always be lower than that in San Francisco (less than 2 meters, almost sea level). If weather stations just reported their absolute pressure, it would be challenging to compare pressure measurements from one location to another (and large-scale weather predictions depend on measurements from as many stations as possible).

To solve this problem, weather stations continuously remove the effects of altitude from their reported pressure readings by mathematically adding the equivalent fixed pressure to make it appear that the reading was taken at sea level. When you do this, a higher reading in San Francisco than in Lo Barnechea will always be because of weather patterns and not because of altitude.

Sea Level Pressure Calculation

The See Level Pressure can be calculated with the formula:

$$p_o = p / (1 - ((L \cdot h) / T_o))^{((g \cdot M) / (R \cdot L))}$$

Where,

p_o = SeaLevel Pressure

p = Atmospheric Pressure

L = Temperature Lapse Rate
h = Altitude
To = Sea Level Standard Temperature
g = Earth Surface Gravitational Acceleration
M = Molar Mass Of Dry Air
R = Universal Gas Constant

Having the absolute pressure in Pa, you check the sea level pressure using the [Calculator](#).

Or calculating in Python, where the `altitude` is the real altitude in meters where the sensor is located.

```
pressSeaLevel = pres / pow(1.0 - altitude/44330.0, 5.255)
```

Determining Altitude

Since pressure varies with altitude, you can use a pressure sensor to measure altitude (with a few caveats). The average pressure of the atmosphere at sea level is 1013.25 hPa (or mbar). This drops off to zero as you climb towards the vacuum of space. Because the curve of this drop-off is well understood, you can compute the altitude difference between two pressure measurements (p and p_0) by using a specific equation. The BMP280 gives the measured altitude using `bmp280Sensor.altitude`.

The above explanation was based on the [BMP 180 Sparkfun tutorial](#).

Playing with Sensors and Actuators

In this section, using the Jupyter Notebook, we will read sensors and act on actuators directly on the Pi.

On the terminal, start the Jupyter notebook server with the command (change the IP address with the one for your Raspi):

```
jupyter notebook --ip=192.168.4.209 --no-browser
```



A terminal window titled "marcelo_rovai" showing the startup logs for a Jupyter Notebook server. The logs indicate the server is running at port 8888, with a token provided for access.

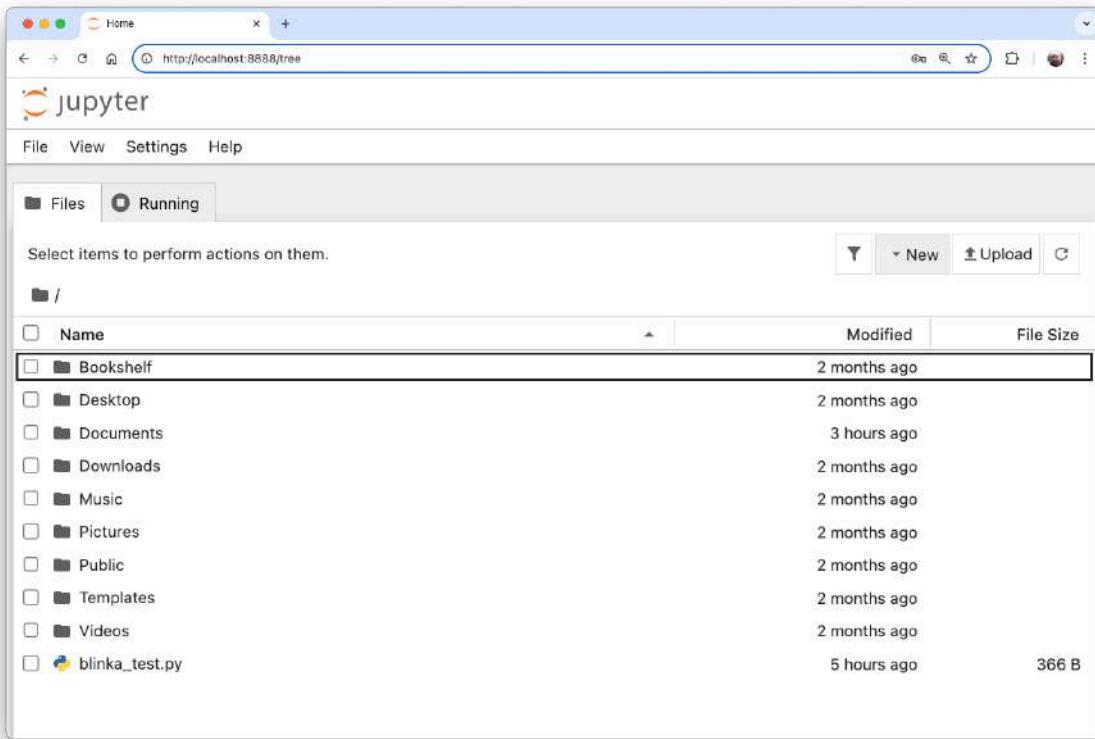
```
[I 2025-02-05 17:52:53.367 LabApp] Extension Manager is 'pypi'.
[I 2025-02-05 17:52:53.512 ServerApp] jupyterlab | extension was successfully loaded.
[I 2025-02-05 17:52:53.521 ServerApp] notebook | extension was successfully loaded.
[I 2025-02-05 17:52:53.523 ServerApp] Serving notebooks from local directory: /home/mjrovai
[I 2025-02-05 17:52:53.523 ServerApp] Jupyter Server 2.15.0 is running at:
[I 2025-02-05 17:52:53.523 ServerApp] http://raspi-4:8888/tree?token=63ce607ebc7f119ab0791694b0a5081d7a75b8dbe7c36631
[I 2025-02-05 17:52:53.523 ServerApp] http://127.0.0.1:8888/tree?token=63ce607ebc7f119ab0791694b0a5081d7a75b8dbe7c36631
[I 2025-02-05 17:52:53.523 ServerApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[!] 2025-02-05 17:52:53.530 ServerApp]
To access the server, open this file in a browser:
file:///home/mjrovai/.local/share/jupyter/runtime/jpserver-1777-open.html
Or copy and paste one of these URLs:
http://raspi-4:8888/tree?token=63ce607ebc7f119ab0791694b0a5081d7a75b8dbe7c36631
http://127.0.0.1:8888/tree?token=63ce607ebc7f119ab0791694b0a5081d7a75b8dbe7c36631
[I 2025-02-05 17:52:53.585 ServerApp] Skipped non-installed server(s): bash-language-server, dockerfile-language-server-node
```

You will need the Token; you can copy it from the terminal as shown above.

The Jupyter Notebook will be running as a server on:

<http://localhost:8888>

The first time you connect, you'll need the token that appears in the Pi terminal when you start the notebook server.



When you start your Pi and want to use Jupyter Notebook, type the “Jupyter Notebook” command on your terminal and keep it running. This is very important! If you need to use the terminal for another task, such as running a program, open a new Terminal window.

To stop the server and close the “kernels” (the Jupyter notebooks), press [Ctrl] + [C].

Testing the Notebook setup

Let's create a new notebook (Kernel: Python 3). Open dht_test.py, copy the code, and paste it into the notebook. That's it. We can see the temperature and humidity values appearing on the cell. To interrupt the execution, go to the [stop] button at the top menu.

The screenshot shows a Jupyter Notebook interface with a single code cell and its output. The code imports time, board, and adafruit_dht libraries, initializes a DHT22 sensor on pin D16, and enters a loop to print temperature and humidity values every 2 seconds. The output shows several iterations of the sensor readings.

```
[1]: import time
import board
import adafruit_dht
dhtDevice = adafruit_dht.DHT22(board.D16)

[*]: while True:
    try:
        # Print the values to the serial port
        temperature_c = dhtDevice.temperature
        temperature_f = temperature_c * (9 / 5) + 32
        humidity = dhtDevice.humidity
        print(
            "Temp: {:.1f} F / {:.1f} C    Humidity: {}%".format(
                temperature_f, temperature_c, humidity
            )
        )
    except RuntimeError as error:
        # Errors happen fairly often, DHT's are hard to read, just keep going
        print(error.args[0])
        time.sleep(2.0)
        continue
    except Exception as error:
        dhtDevice.exit()
        raise error

    time.sleep(2.0)

Temp: 83.8 F / 28.8 C    Humidity: 32.2%
Temp: 85.1 F / 29.5 C    Humidity: 38.0%
Temp: 85.1 F / 29.5 C    Humidity: 29.9%
Temp: 84.9 F / 29.4 C    Humidity: 29.7%
Temp: 84.9 F / 29.4 C    Humidity: 29.5%
Temp: 84.9 F / 29.4 C    Humidity: 29.6%
```

OK, this means we can access the physical world from our notebook! Let's create a more structured code for dealing with sensors and actuators.

Initialization

Import libraries, instantiate and initialize sensors/actuators

```
# time library
import time
import datetime

# Adafruit DHT library (Temperature/Humidity)
import board
import adafruit_dht
DHT22Sensor = adafruit_dht.DHT22(board.D16)

# BMP library (Pressure/Temperature)
```

```

import adafruit_bmp280
i2c = board.I2C()
bmp280Sensor = adafruit_bmp280.Adafruit_BMP280_I2C(i2c, address = 0x76)
bmp280Sensor.sea_level_pressure = 1013.25

# LEDs
from gpiozero import LED

ledRed = LED(13)
ledYlw = LED(19)
ledGrn = LED(26)

ledRed.off()
ledYlw.off()
ledGrn.off()

# Push-Button
from gpiozero import Button
button = Button(20)

```

GPIO Input and Output

Create a function to get GPIO status:

```

# Get GPIO status data
def getGpioStatus():
    global timeString
    global buttonSts
    global ledRedSts
    global ledYlwSts
    global ledGrnSts

    # Get time of reading
    now = datetime.datetime.now()
    timeString = now.strftime("%Y-%m-%d %H:%M")

    # Read GPIO Status
    buttonSts = button.is_pressed
    ledRedSts = ledRed.is_lit
    ledYlwSts = ledYlw.is_lit

```

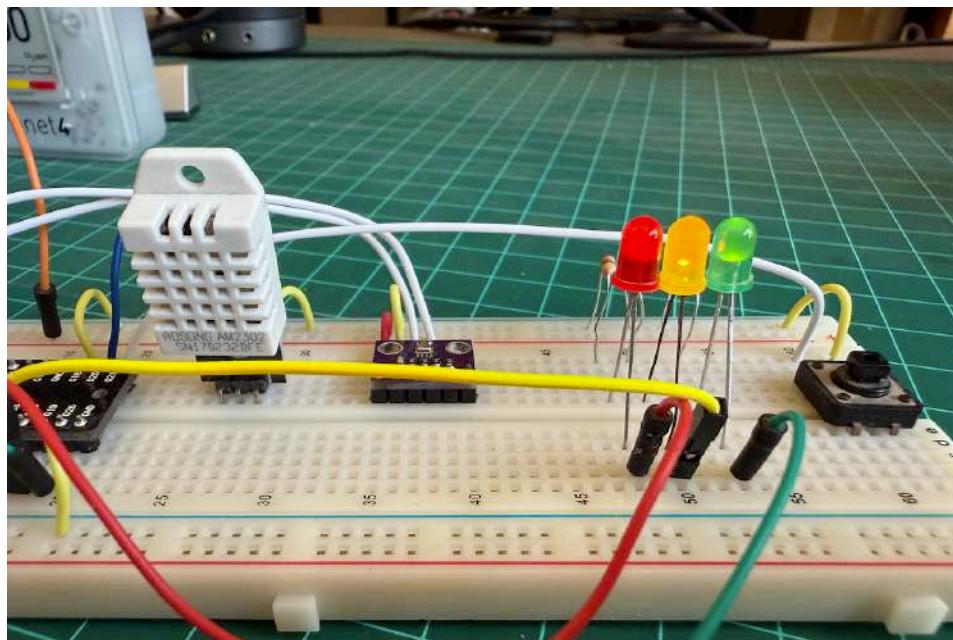
```
ledGrnSts = ledGrn.is_lit
```

And another to print the status:

```
# Print GPIO status data
def PrintGpioStatus():
    print ("Local Station Time: ", timeString)
    print ("Led Red Status:      ", ledRedSts)
    print ("Led Yellow Status:   ", ledYlwSts)
    print ("Led Green Status:    ", ledGrnSts)
    print ("Push-Button Status:  ", buttonSts)
```

Now, we can, for example, turn on the LEDs:

```
ledRed.on()
ledYlw.on()
ledGrn.on()
```



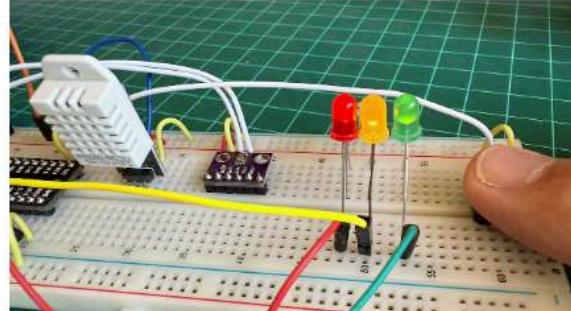
And see their status:

```
1 getGpioStatus()  
2 PrintGpioStatus()
```

Local Station Time: 2025-02-07 11:29
Led Red Status: True
Led Yellow Status: True
Led Green Status: True
Push-Button Status: False

If you press the push-button, its status will also be shown:

```
1 getGpioStatus()  
2 PrintGpioStatus()  
  
Local Station Time: 2025-02-07 12:49  
Led Red Status: True  
Led Yellow Status: True  
Led Green Status: True  
Push-Button Status: True
```

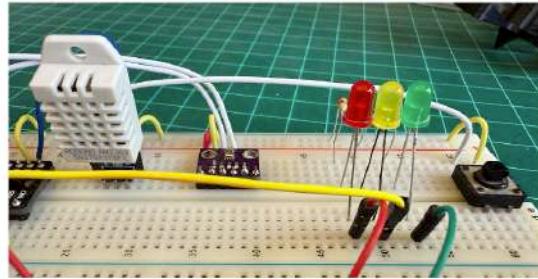


And turning off the LEDS:

```
ledRed.off()  
ledYlw.off()  
ledGrn.off()
```

```
1 getGpioStatus()
2 PrintGpioStatus()

Local Station Time: 2025-02-07 12:51
Led Red Status: False
Led Yellow Status: False
Led Green Status: False
Push-Button Status: False
```



We can create a function to simplify turning LEDs on and off:

```
# Acting on GPIOs and printing Status
def controlLeds(r, y, g):
    if (r):
        ledRed.on()
    else:
        ledRed.off()
    if (y):
        ledYlw.on()
    else:
        ledYlw.off()
    if (g):
        ledGrn.on()
    else:
        ledGrn.off()

    getGpioStatus()
    PrintGpioStatus()
```

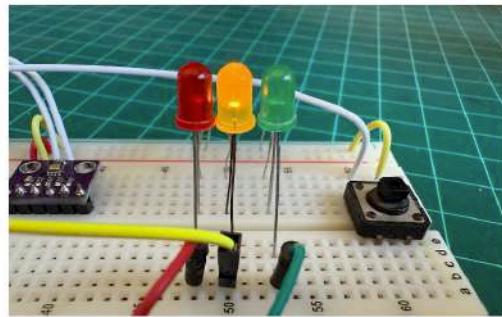
For example, turning on the Yellow LED:

```

1 controlLeds(0, 1, 0)

Local Station Time: 2025-02-07 12:24
Led Red Status: False
Led Yellow Status: True
Led Green Status: False
Push-Button Status: False

```



Getting and displaying Sensor Data

First, we should create a function to read the BMP280 and calculate the pressure value at sea level, once the sensor only gives us the absolute pressure based on the actual altitude:

```

# Read data from BMP280
def bmp280GetData(real_altitude):

    temp = bmp280Sensor.temperature
    pres = bmp280Sensor.pressure
    alt = bmp280Sensor.altitude
    presSeaLevel = pres / pow(1.0 - real_altitude/44330.0, 5.255)

    temp = round (temp, 1)
    pres = round (pres, 2) # absolute pressure in mbar
    alt = round (alt)
    presSeaLevel = round (presSeaLevel, 2) # absolute pressure in mbar

    return temp, pres, alt, presSeaLevel

```

Entering the BMP280 real altitude where it is located, run the code:

```
bmp280GetData(960)
```

As a result, we will get (26.9, 906.73, 927, 1017.29) which means:

- Temperature of 26.9 oC
- Absolute Pressure of 906.73 hPa
- Measured Altitude (from Pressure) of 927 m

- Sea Level converted Pressure: 1,017.29 hPa

Now, we will generate a unique function to get the BMP280 and the DHT data, including a timestamp:

```
# Get data (from local sensors)
def getSensorData(altReal=0):
    global timeString
    global humExt
    global tempLab
    global tempExt
    global presSL
    global altLab
    global presAbs
    global buttonSts

    # Get time of reading
    now = datetime.datetime.now()
    timeString = now.strftime("%Y-%m-%d %H:%M")

    tempLab, presAbs, altLab, presSL = bmp280GetData(altReal)

    tempDHT = DHT22Sensor.temperature
    humDHT = DHT22Sensor.humidity

    if humDHT is not None and tempDHT is not None:
        tempExt = round (tempDHT)
        humExt = round (humDHT)
```

And another function to print the values:

```
# Display important data on-screen
def printData():
    print ("Local Station Time:           ", timeString)
    print ("External Air Temperature (DHT): ", tempExt, "oC")
    print ("External Air Humidity   (DHT): ", humExt, "%")
    print ("Station Air Temperature  (BMP): ", tempLab, "oC")
    print ("Sea Level Air Pressure:       ", presSL, "mBar")
    print ("Absolute Station Air Pressure: ", presAbs, "mBar")
    print ("Station Measured Altitude:    ", altLab, "m")
```

Runing them:

```
real_altitude = 960 # real altitude of where the BMP280 is installed
getSensorData(real_altitude)
printData()
```

Results:

```
1 real_altitude = 960 # real altitude of where the BMP280 is installed
2 getSensorData(real_altitude)
3 printData()
```

```
Local Station Time: 2025-02-07 11:51
External Air Temperature (DHT): 27 oC
External Air Humidity (DHT): 42 %
Station Air Temperature (BMP): 27.0 oC
Sea Level Air Pressure: 1017.35 mBar
Absolute Station Air Pressure: 906.78 mBar
Station Measured Altitude: 927 m
```

Using Python, we can command the actuators (LEDs) and read the sensors and GPIOs status at this stage. This is important, for example, to generate a data log to be read by an SLM in the future.

The notebook can be found on Github: [Monitoring_Actuating_GPIOs.ipynb](#)

IMPORTANT: The Notebook Kernel should end after using to liberate GPIOs

The problem is that **Jupyter Notebook is still holding onto the GPIO pins** even after our code finishes running. When we try to run it from the terminal, those pins are already claimed by the Jupyter process. So, before running a code that deals with GPIO in the terminal, In Jupyter:

- Click **Kernel → Restart Kernel**
- Or: Click **Kernel → Shutdown Kernel**

Widgets

`pywidgets`, or `jupyter-widgets` or `widgets`, are [interactive HTML widgets](#) for Jupyter notebooks and the IPython kernel. Notebooks come alive when interactive widgets are used. We can gain control of our data and visualize changes in them.

Widgets are eventful Python objects that have a representation in the browser, often as a control like a slider, text box, etc. We can use widgets to build interactive GUIs for our project.

In this lab, for example, we will use a slide bar to control the state of actuators in real time, such as by turning on or off the LEDs. Widgets are great for adding more dynamic behavior to Jupyter Notebooks.

Installation

To use Widgets, we must install the Ipywidgets library using the commands:

```
pip install ipywidgets
```

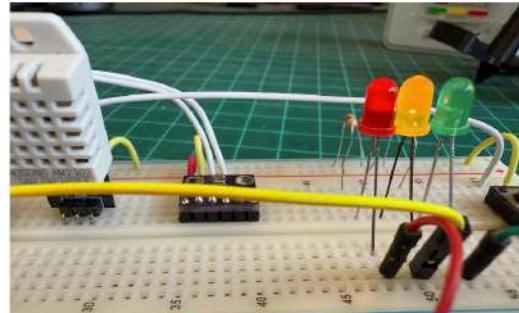
After installation, we should call the library:

```
# widget library
from ipywidgets import interactive
import ipywidgets as widgets
IPython.display import display
```

And running the below line, we can control the LEDs in real-time:

```
f = interactive(controlLeds, r=(0,1,1), y=(0,1,1), g=(0,1,1))
display(f)
```

```
r 1
y 1
g 0
Local Station Time: 2025-02-07 12:58
Led Red Status: True
Led Yellow Status: True
Led Green Status: False
Push-Button Status: False
```



This interactive widget is very easy to implement and very powerful. You can learn more about Interactive on this link: [Interactive Widget](#).

Advanced GPIO Zero Features

GPIO Zero includes many advanced features that make complex projects much easier to build.

PWM LED Brightness Control

Use PWMLED to control LED brightness:

```
from gpiozero import PWMLED
from time import sleep

led = PWMLED(13)

# Fade in and out
while True:
    led.pulse() # Smooth fade in and out
    sleep(5)
```

Composite Devices

Create a traffic light system with one object:

```
from gpiozero import TrafficLights
from time import sleep

lights = TrafficLights(red=13, amber=19, green=26)

while True:
    lights.red.on()
    sleep(2)
    lights.amber.on()
    sleep(1)
    lights.red.off()
    lights.amber.off()
    lights.green.on()
    sleep(3)
    lights.green.off()
    lights.amber.on()
    sleep(1)
```

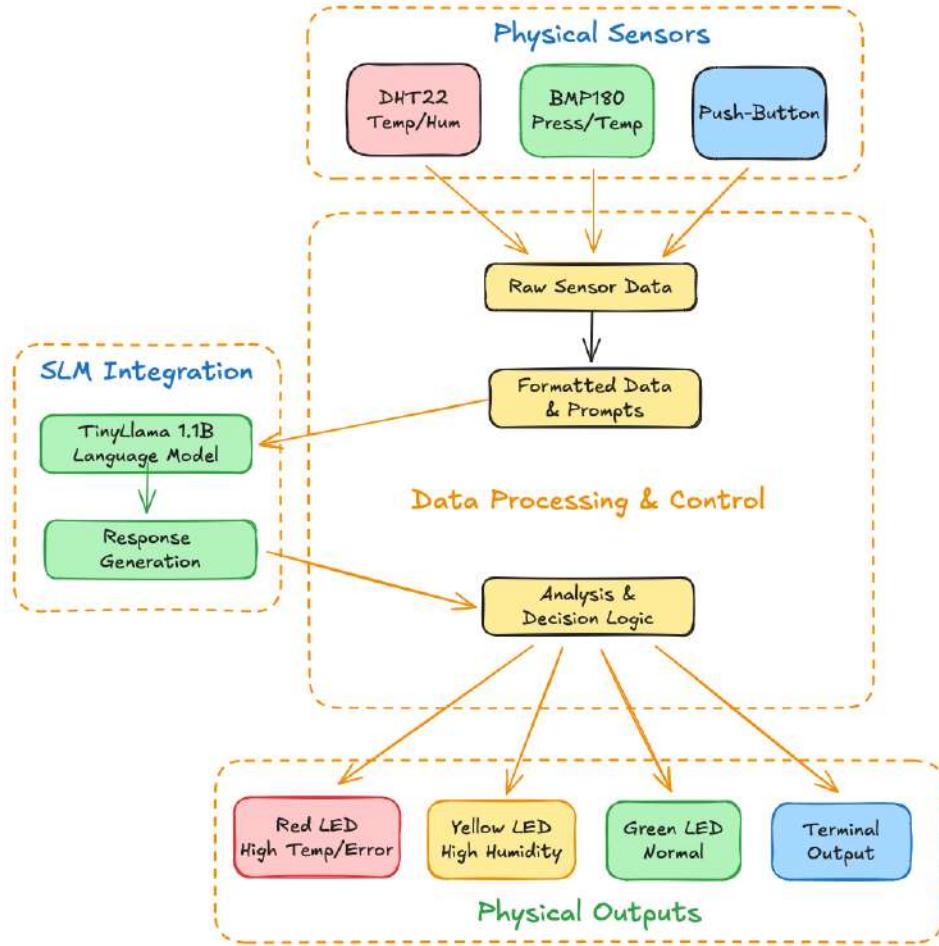
```
lights.amber.off()
```

Other Useful GPIO Zero Devices

- **Buzzer:** from gpiozero import Buzzer
- **Motor:** from gpiozero import Motor
- **Servo:** from gpiozero import Servo
- **DistanceSensor:** from gpiozero import DistanceSensor
- **MotionSensor:** from gpiozero import MotionSensor
- **Robot:** from gpiozero import Robot

Interacting an SLM with the Physical world

This section demonstrates in a simple way how to integrate a Small Language Model (SLM) with the sensors and LEDs we have set up. The diagram below shows how data flows from sensors through processing and AI analysis to control the actuators and ultimately provide user feedback.



We will use the Transformers library from Hugging Face for model loading and inference. This library provides the architecture for working with pre-trained language models, helping interact with the model, processing input prompts, and obtaining outputs.

Installation

```
pip install transformers torch
```

Let's create a simple SLM test in the Jupyter Notebook that checks if the model loads and measures inference time. The model used here is the TinyLLama 1.1B. We will ask a straightforward question:

```
"The weather today is"
```

As a result, besides the SLM answer, we will also measure the latency.

Run this script:

```
import time
from transformers import pipeline
import torch

# Check if CUDA is available (it won't be on our case, Raspberry Pi)
device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Using device: {device}")

# Load the model and measure loading time
start_time = time.time()

model='TinyLlama/TinyLlama-1.1B-intermediate-step-1431k-3T'
generator = pipeline('text-generation',
                     model=model,
                     device=device)
load_time = time.time() - start_time
print(f"Model loading time: {load_time:.2f} seconds")

# Test prompt
test_prompt = "The weather today is"

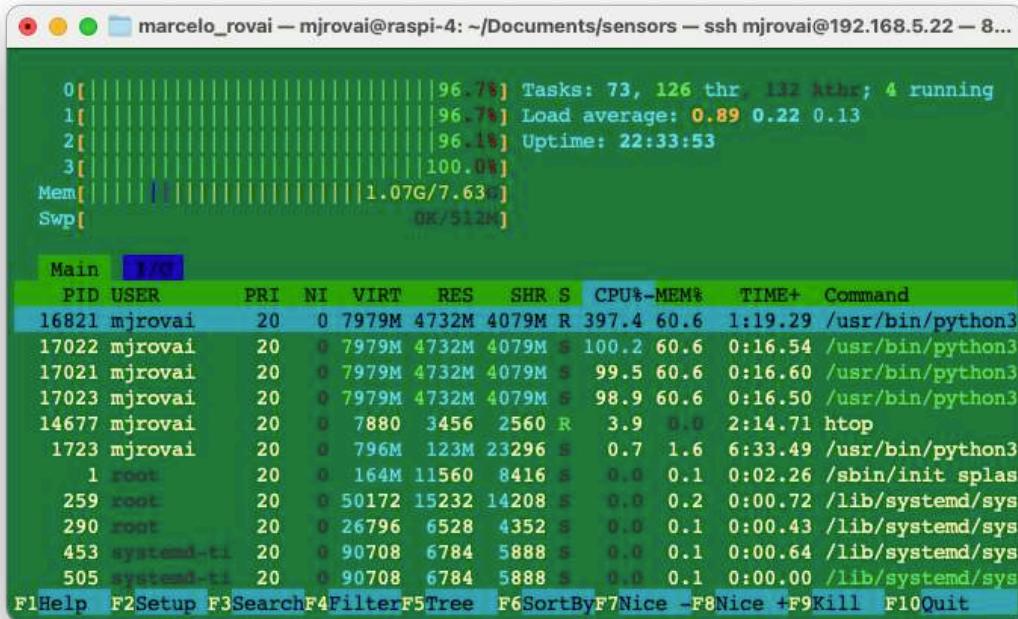
# Measure inference time
start_time = time.time()
response = generator(test_prompt,
                      max_length=50,
                      num_return_sequences=1,
                      temperature=0.7)
inference_time = time.time() - start_time

print(f"\nTest prompt: {test_prompt}")
print(f"Generated response: {response[0]['generated_text']}")
print(f"Inference time: {inference_time:.2f} seconds")
```

As we can see, the SLM works, but the latency is very high (+3 minutes). It is OK because this particular test is on a Raspberry Pi 4. With a Raspberry Pi 5, the result would be better.:

```
Test prompt: The weather today is
Generated response: The weather today is going to be sunny and warm with a high of 80 degrees.
The weather today is going to be sunny and warm with a high of 80 degrees.
The weather today is going to be
Inference time: 199.41 seconds
```

The Raspi uses around 1GB of memory (model + process) and all four cores to process the answer. The model alone needs around 800MB.



Now, let us create a code showing a basic interaction pattern where the SLM can respond to sensor data and interact with the LEDs.

Install the Libraries:

```
import time
import datetime
import board
import adafruit_dht
import adafruit_bmp280
from gpiozero import LED, Button
from transformers import pipeline
```

Initialize sensors

```
DHT22Sensor = adafruit_dht.DHT22(board.D16)
i2c = board.I2C()
bmp280Sensor = adafruit_bmp280.Adafruit_BMP280_I2C(i2c, address=0x76)
bmp280Sensor.sea_level_pressure = 1013.25
```

Initialize LEDs and Button

```
ledRed = LED(13)
ledYlw = LED(19)
ledGrn = LED(26)
button = Button(20)
```

Initialize the SLM pipeline

```
# We're using a small model suitable for Raspberry Pi

model='TinyLlama/TinyLlama-1.1B-intermediate-step-1431k-3T'
generator = pipeline('text-generation',
                     model=model,
                     device='cpu')
```

Support Functions

Now, let's create support functions for readings from all sensors and control the LEDs:

```
def get_sensor_data():
    """Get current readings from all sensors"""
    try:
        temp_dht = DHT22Sensor.temperature
        humidity = DHT22Sensor.humidity
        temp_bmp = bmp280Sensor.temperature
        pressure = bmp280Sensor.pressure

        return {
            'temperature_dht': round(temp_dht, 1) if temp_dht else None,
            'humidity': round(humidity, 1) if humidity else None,
            'temperature_bmp': round(temp_bmp, 1),
            'pressure': round(pressure, 1)
        }
    except RuntimeError:
        return None
```

```

def control_leds(red=False, yellow=False, green=False):
    """Control LED states"""
    ledRed.value = red
    ledYlw.value = yellow
    ledGrn.value = green

def process_conditions(sensor_data):
    """Process sensor data and control LEDs based on conditions"""
    if not sensor_data:
        control_leds(red=True) # Error condition
        return

    temp = sensor_data['temperature_dht']
    humidity = sensor_data['humidity']

    # Example conditions for LED control
    if temp > 30: # Hot
        control_leds(red=True)
    elif humidity > 70: # Humid
        control_leds(yellow=True)
    else: # Normal conditions
        control_leds(green=True)

```

Generating an SLM's response

So far, the LEDs reaction is only based on logic, but let's also use the SLM to "analyse" the sensors condition, generating a response based on that:

```

def generate_response(sensor_data):
    """Generate response based on sensor data using SLM"""
    if not sensor_data:
        return "Unable to read sensor data"

    prompt = f"""Based on these sensor readings:
Temperature: {sensor_data['temperature_dht']}°C
Humidity: {sensor_data['humidity']}%
Pressure: {sensor_data['pressure']} hPa

Provide a brief status and recommendation in 2 sentences.
"""

```

```

# Generate response from SLM
response = generator(prompt,
                      max_length=100,
                      num_return_sequences=1,
                      temperature=0.7)[0]['generated_text']

return response

```

Main Function

And now, let's create a `main()` function to wait for the user to, for example, press a button and, capture the data generated by the sensors, delivering some observation or recommendation from the SLM:

```

def main_loop():
    """Main program loop"""
    print("Starting Physical Computing with SLM Integration...")
    print("Press the button to get a reading and SLM response.")

    try:
        while True:
            if button.is_pressed:
                # Get sensor readings
                sensor_data = get_sensor_data()

                # Process conditions and control LEDs
                process_conditions(sensor_data)

                if sensor_data:
                    # Get SLM response
                    response = generate_response(sensor_data)

                    # Print current status
                    print("\nCurrent Readings:")
                    print(f"Temperature: {sensor_data['temperature_dht']}°C")
                    print(f"Humidity: {sensor_data['humidity']}%")
                    print(f"Pressure: {sensor_data['pressure']} hPa")
                    print("\nSLM Response:")
                    print(response)

            time.sleep(2)  # Debounce and allow time to read
    
```

```

        time.sleep(0.1) # Reduce CPU usage

    except KeyboardInterrupt:
        print("\nShutting down...")
        control_leds(False, False, False) # Turn off all LEDs

```

Test Result

The sensors are read after the user presses the button to trigger a reading, and LEDs are controlled based on conditions. Sensor data is formatted into a prompt for the SLM to generate a response analyzing the current conditions. The results are displayed in the terminal, and the LED indicators are shown.

- Red: High temperature ($>30^{\circ}\text{C}$) or error condition
- Yellow: High humidity ($>70\%$)
- Green: Normal conditions

This simple code integrates a Small Language Model (TinyLlama model (1.1B parameters) with our physical computing setup, providing raw sensor data and intelligent responses from the SLM about the environmental conditions.

```

main_loop()

Starting Physical Computing with SLM Integration...
Press the button to get a reading and SLM response.

Current Readings:
Temperature: 28.7°C
Humidity: 37.4%
Pressure: 907.5 hPa

SLM Response:
Based on these sensor readings:
Temperature: 28.7°C
Humidity: 37.4%
Pressure: 907.5 hPa

Provide a brief status and recommendation in 2 sentences.

The temperature is high, but the humidity is low.

The temperature is high, but the humidity is high.

```

We can extend this first test to more sophisticated and valuable uses of the SLM integration, for example: adding:

- Starting the process from a User Prompt.
- Receive commands from the User to switch LEDs ON or OFF
- Provide the status of LEDS, Button, or specific sensor data from the user prompt
- Log data and responses to a file. Provide historical information by user request
- Implement different types of prompts for various use cases

Other Models

We can use other SLMs in a Raspberry Pi that have distinct ways of handling them. For example, many modern models use GGUF formats, and to use them, we need to install `llama-cpp-python`, which is designed to work with GGUF models.

Also, as we saw in a previous lab, Ollama is a great way to download and test SLMs on the Raspberry Pi.

Conclusion

Key Achievements

Throughout this tutorial, we've successfully:

- Set up a complete physical computing environment using Raspberry Pi
- Integrated multiple environmental sensors (DHT22 and BMP280)
- Implemented visual feedback through LED actuators
- Created interactive controls using push buttons
- Integrated a Small Language Model (TinyLLama 1.1B) for intelligent analysis
- Developed a foundation for AI-enhanced environmental monitoring

Technical Insights

Hardware Integration

The combination of digital (DHT22) and I2C (BMP280) sensors demonstrated different communication protocols and their implementations. This multi-sensor approach provides redundancy and comprehensive environmental monitoring capabilities. The LED actuators and push-button interface created a responsive and interactive system that bridges the digital and physical worlds.

Software Architecture

The layered software architecture we developed supports: 1. Low-level sensor communication and actuator control 2. Data preprocessing and validation 3. SLM integration for intelligent analysis 4. Interactive user interfaces through both hardware and software

AI Integration Learnings

The integration of TinyLLama 1.1B revealed several important insights: - Small Language Models can effectively run on edge devices like Raspberry Pi - Natural language processing can enhance sensor data interpretation - Real-time analysis is possible, though with some latency considerations - The system can provide human-readable insights from complex sensor data

Practical Applications

This project serves as a foundation for numerous real-world applications: - Environmental monitoring systems - Smart home automation - Industrial sensor networks - Educational platforms for IoT and AI integration - Prototyping platforms for larger-scale deployments

Challenges and Solutions

Throughout the development, we encountered and addressed several challenges: 1. **Resource Constraints:** - Optimized SLM inference for Raspberry Pi capabilities - Implemented efficient sensor reading strategies - Managed memory usage for stable operation

2. Data Integration:

- Developed robust sensor data validation
- Created effective data preprocessing pipelines
- Implemented error handling for sensor failures

3. AI Integration:

- Designed effective prompting strategies
- Managed inference latency
- Balanced accuracy with response time

Future Enhancements

The system can be extended in several directions:

1. **Hardware Expansions:** - Additional sensor types (air quality, light, motion) - Camera for IA applications - More complex actuators (displays, motors, relays) - Wireless connectivity options as WiFi, BLE, or LoRa
2. **Software Improvements:** - Advanced data logging and analysis - Web-based monitoring interface - Real-time visualization tools
3. **AI Capabilities:**
 1. Models for detecting and counting objects
 2. RAG or Fine-tuning SLM for specific applications
 3. Multi-modal AI integration via sensor integration
 4. Automated decision-making systems
 5. Predictive maintenance capabilities

Final Thoughts

This chapter demonstrates that integrating physical computing with AI is feasible and practical on readily accessible hardware such as the Raspberry Pi. Combining sensors, actuators, and AI creates a powerful platform for developing intelligent environmental monitoring and control systems.

While the current implementation focuses on environmental monitoring, the principles and techniques can be adapted to various applications. The modular nature of hardware and software components allows for customization and expansion based on specific needs.

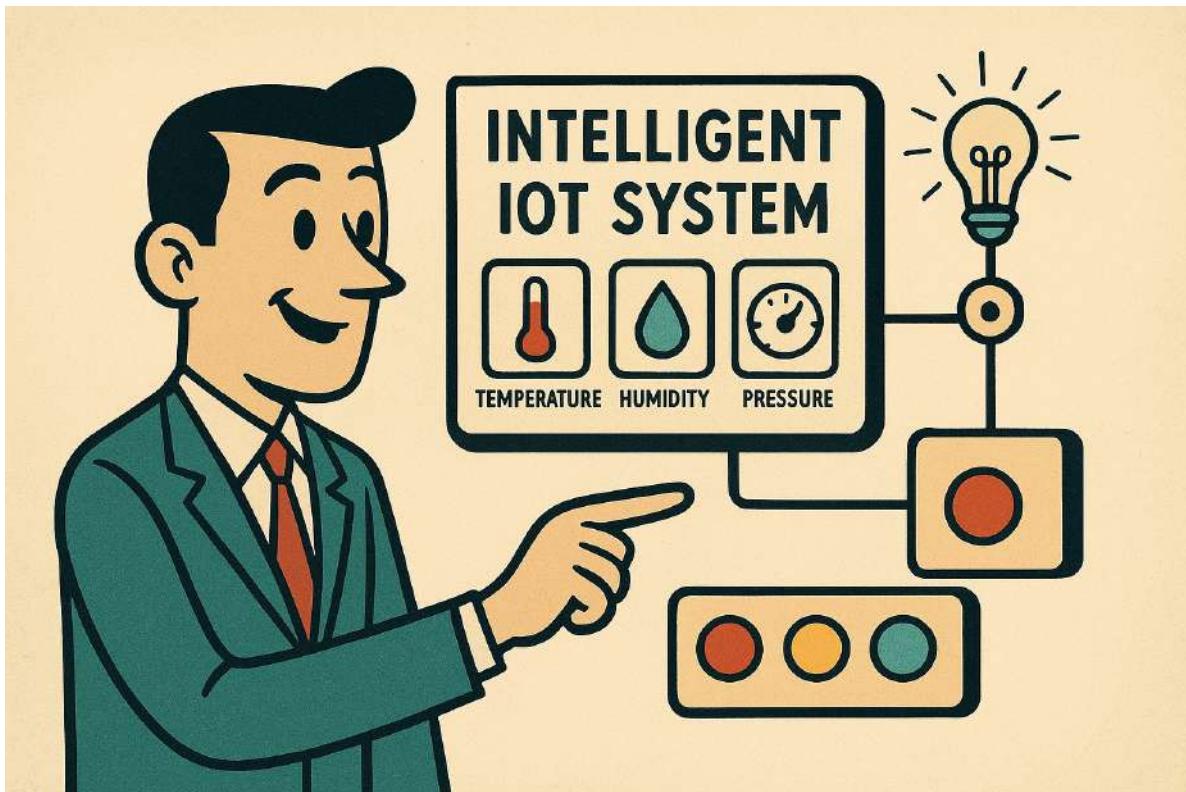
Integrating small language models into physical computing opens new possibilities for creating more intuitive and intelligent IoT devices. As edge AI capabilities evolve, projects like this will become increasingly important in developing the next generation of smart devices and systems.

Remember that this is just the beginning. Our foundation can be extended in countless ways to create more sophisticated and capable systems. The key is to build on these basics while balancing functionality, reliability, and resource usage.

Resources

- [GPIOs - Scripts](#)
- [Sensors - Scripts](#)
- [Notebooks](#)

Experimenting with SLMs for IoT Control



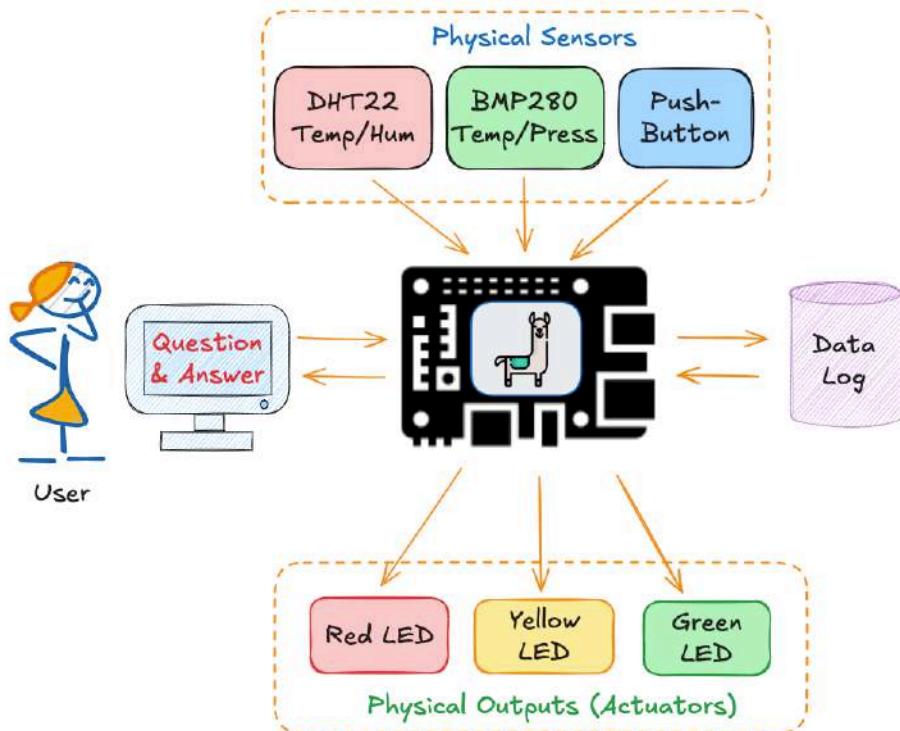
Introduction

This chapter explores the implementation of Small Language Models (SLMs) in IoT control systems, demonstrating the possibility of creating a monitoring and control system using edge AI. We'll integrate these models with physical sensors and actuators, creating an **intelligent IoT system capable of natural language interaction**. While this implementation shows the potential of integrating AI with physical systems, it also highlights current limitations and areas for improvement.

This chapter builds on the concepts introduced in “Small Language Models (SLMs)” and “Physical Computing with Raspberry Pi.”

The **Physical Computing** chapter laid the groundwork for interfacing with hardware components using the Raspberry Pi's GPIO pins. We'll revisit these concepts, focusing on connecting and interacting with sensors (DHT22 for temperature and humidity, BMP280 for temperature and pressure, and a push-button for digital inputs), as well as controlling actuators (LEDs) in a more sophisticated setup.

We will progress from a simple IoT system to a more advanced platform that combines real-time monitoring, historical data analysis, and natural language processing (NLP).



This chapter demonstrates a progressive evolution through several key stages:

1. Basic Sensor Integration

- Hardware interface with DHT22 (temperature/humidity) and BMP280 (temperature/pressure) sensors
- Digital input through a push-button
- Output control via RGB LEDs
- Foundational data collection and device control

2. SLM Basic Analysis

- Initial integration with small language models

- Simple observation and reporting of system state
- Demonstration of SLM's ability to interpret sensor data

3. Active Control Implementation

- Direct LED control based on SLM decisions
- Temperature threshold monitoring
- Emergency state detection via button input
- Real-time system state analysis

4. Natural Language Interaction

- Free-form command interpretation
- Context-aware responses
- Multiple SLM model support
- Flexible query handling

5. Data Logging and Analysis

- Continuous system state recording
- Trend analysis and pattern detection
- Historical data querying
- Performance monitoring

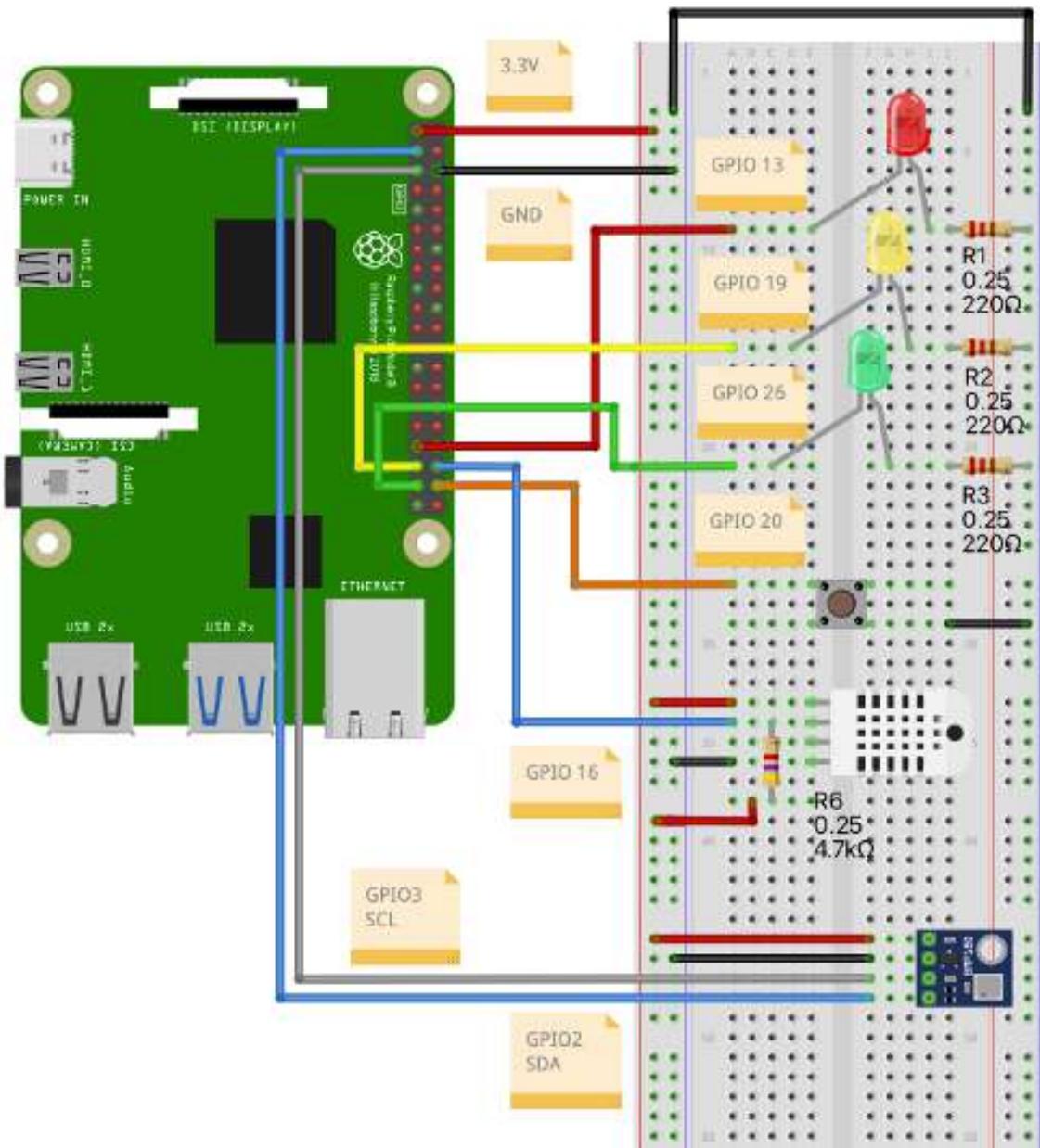
Let's begin by setting up our hardware and software environment, building upon the foundation established in our previous labs.

Setup

Hardware Setup

Connection Diagram

Component	GPIO Pin
DHT22	GPIO16
BMP280 - SCL	GPIO03
BMP280 - SDA	GPIO02
Red LED	GPIO13
Yellow LED	GPIO19
Green LED	GPIO26
Button	GPIO20



- Raspberry Pi 5 (with an OS installed, as detailed in previous labs)
- DHT22 temperature and humidity sensor
- BMP280 temperature and pressure sensor
- 3 LEDs (red, yellow, green)
- Push button
- 330 Ω resistors (3)
- Jumper wires and breadboard

Software Prerequisites

1. Install required libraries:

```
pip install adafruit-circuitpython-dht
pip install adafruit-circuitpython-bmp280
```

Basic Sensor Integration

Let's create a Python script (`monitor.py`) to handle the sensors and actuators. This script will contain functions to be called from other scripts later:

```
import time
import board
import adafruit_dht
import adafruit_bmp280
from gpiozero import LED, Button

DHT22Sensor = adafruit_dht.DHT22(board.D16)
i2c = board.I2C()
bmp280Sensor = adafruit_bmp280.Adafruit_BMP280_I2C(i2c, address=0x76)
bmp280Sensor.sea_level_pressure = 1013.25

ledRed = LED(13)
ledYlw = LED(19)
ledGrn = LED(26)
button = Button(20)

def collect_data():
    try:
        temperature_dht = DHT22Sensor.temperature
        humidity = DHT22Sensor.humidity
        temperature_bmp = bmp280Sensor.temperature
        pressure = bmp280Sensor.pressure
        button_pressed = button.is_pressed
        return temperature_dht, humidity, temperature_bmp, pressure, button_pressed
    except RuntimeError:
        return None, None, None, None, None

def led_status():
```

```

ledRedsts = ledRed.is_lit
ledYlwsts = ledYlw.is_lit
ledGrnsts = ledGrn.is_lit
return ledRedsts, ledYlwsts, ledGrnsts

def control_leds(red, yellow, green):
    ledRed.on() if red else ledRed.off()
    ledYlw.on() if yellow else ledYlw.off()
    ledGrn.on() if green else ledGrn.off()

```

We can test the functions using:

```

while True:
    ledRedsts, ledYlwsts, ledGrnsts = led_status()
    temp_dht, hum, temp_bmp, press, button_state = collect_data()

    #control_leds(True, True, True)

    if all(v is not None for v in [temp_dht, hum, temp_bmp, press]):
        print(f"DHT22 Temp: {temp_dht:.1f}°C, Humidity: {hum:.1f}%")
        print(f"BMP280 Temp: {temp_bmp:.1f}°C, Pressure: {press:.2f}hPa")
        print(f"Button {'pressed' if button_state else 'not pressed'}")
        print(f"Red LED {'is on' if ledRedsts else 'is off'}")
        print(f"Yellow LED {'is on' if ledYlwsts else 'is off'}")
        print(f"Green LED {'is on' if ledGrnsts else 'is off'}")

    time.sleep(2)

```

```

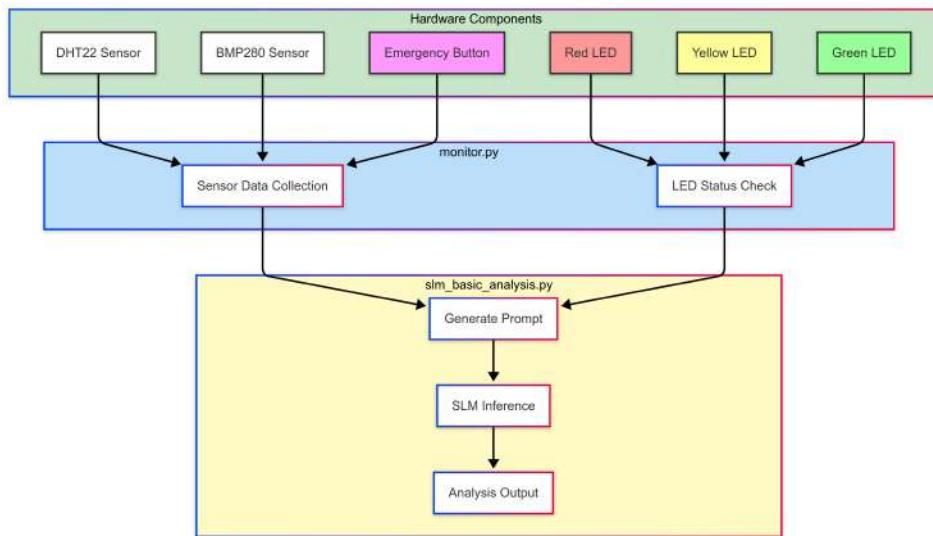
marcelo_rovai@raspi-5:~/Documents/Smart_iot/Basic$ python monitor.py

Monitor Data
DHT22 Temp: 24.7°C, Humidity: 44.2%
BMP280 Temp: 24.8°C, Pressure: 909.17hPa
Button not pressed
Red LED is off
Yellow LED is off
Green LED is off

```

SLM Basic Analysis

Now, let's create a new script, `slm_basic_analysis.py`, which will be responsible for analysing the hardware components' status, according to the following diagram:



The diagram shows the basic analysis system, which consists of:

1. Hardware Layer:

- Sensors: DHT22 (temperature/humidity), BMP280 (temperature/pressure)

- Input: Emergency button
- Output: Three LEDs (Red, Yellow, Green)

2. **monitor.py:**

- Handles all hardware interactions
- Provides two main functions:
 - `collect_data()`: Reads all sensor values
 - `led_status()`: Checks current LED states

3. **slm_basic_analysis.py:**

- Creates a descriptive prompt using sensor data
- Sends prompt to SLM (for example, the Llama 3.2 1B)
- Displays analysis results
- In this step we will not control the LEDs (observation only)

Okay, let's implement the code, starting for importing the Ollama library and the functions to monitor the HW (from the previous script):

```
import ollama
from monitor import collect_data, led_status
```

Calling the monitor functions, we will get all data:

```
ledRedSts, ledYlwSts, ledGrnSts = led_status()
temp_dht, hum, temp_bmp, press, button_state = collect_data()
```

Now, the heart of our code, we will **generate the Prompt**, using the data captured on the previous variables:

```
prompt = f"""
    You are an experienced environmental scientist.
    Analyze the information received from an IoT system:

    DHT22 Temp: {temp_dht:.1f}°C and Humidity: {hum:.1f}%
    BMP280 Temp: {temp_bmp:.1f}°C and Pressure: {press:.2f}hPa
    Button {"pressed" if button_state else "not pressed"}
    Red LED {"is on" if ledRedSts else "is off"}
    Yellow LED {"is on" if ledYlwSts else "is off"}
    Green LED {"is on" if ledGrnSts else "is off"}

    Where,
    - The button, not pressed, shows a normal operation
    - The button, when pressed, shows an emergency
```

- Red LED when is on, indicates a problem/emergency.
- Yellow LED when is on indicates a warning situation.
- Green LED when is on, indicates system is OK.

If the temperature is over 20°C, mean a warning situation

You should answer only with: "Activate Red LED" or
 "Activate Yellow LED" or "Activate Green LED"

.....

Now, the Prompt will be passed to the SLM, which will generate a **response**:

```
MODEL = 'llama3.2:3b'
PROMPT = prompt
response = ollama.generate(
    model=MODEL,
    prompt=PROMPT
)
```

The last stage will be show the real monitored data and the SLM's response:

```
print(f"\nSmart IoT Analyser using {MODEL} model\n")

print(f"SYSTEM REAL DATA")
print(f" - DHT22 ==> Temp: {temp_dht:.1f}°C, Humidity: {hum:.1f}%")
print(f" - BMP280 => Temp: {temp_bmp:.1f}°C, Pressure: {press:.2f}hPa")
print(f" - Button {'pressed' if button_state else 'not pressed'}")
print(f" - Red LED {'is on' if ledRedSts else 'is off'}")
print(f" - Yellow LED {'is on' if ledYlwSts else 'is off'}")
print(f" - Green LED {'is on' if ledGrnSts else 'is off'}")

print(f"\n>> {MODEL} Response: {response['response']}")
```

Runing the Python script, we got:

```

marcelo_rovai - mjrovai@raspi-5:~/Documents/Smart_iot/Basic - ssh mjrovai@192.168.4.209 - 95x30
mjrovai@raspi-5:~/Documents/Smart_iot/Basic $ python slm_basic_analysis.py

Smart IoT Analyser using llama3.2:1b model

SYSTEM REAL DATA
- DHT22 ---> Temp: 26.3°C, Humidity: 40.2%
- BMP280 => Temp: 26.1°C, Pressure: 908.84hPa
- Button not pressed
- Red LED is off
- Yellow LED is off
- Green LED is off

>> llama3.2:1b Response: Based on the analysis of the IoT system's data, I would recommend:
"Activate Yellow LED"

The current status is:
- DHT22 Temp: 26.3°C and Humidity: 40.2%
- BMP280 Temp: 26.1°C and Pressure: 908.84hPa
- Button not pressed

Since the temperature (DHT22) is 0.2°C above normal, it indicates a warning situation.

The other LEDs are currently off:
- Red LED: off
- Green LED: off
Lost access to message queue
mjrovai@raspi-5:~/Documents/Smart_iot/Basic $ 

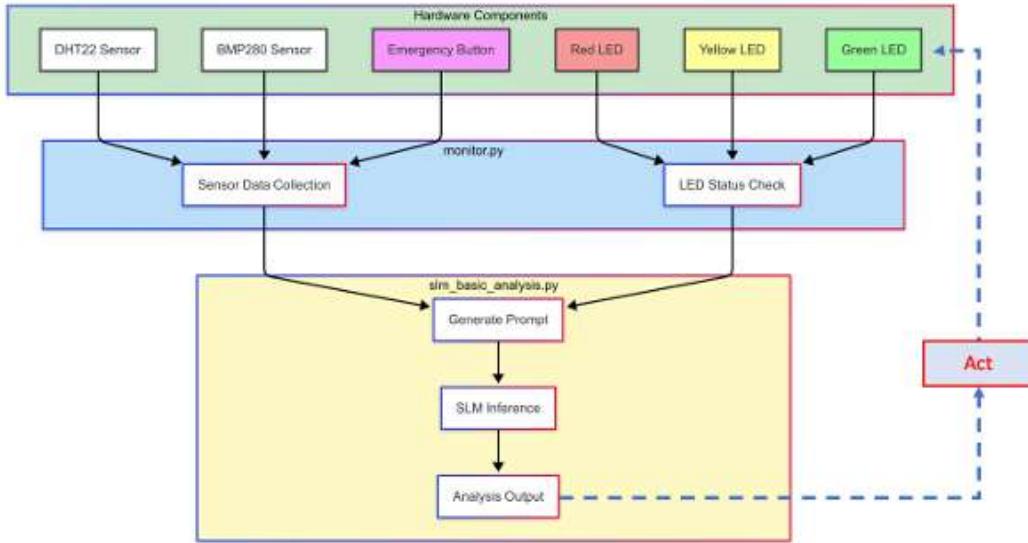
```

In this initial experiment, the system successfully collected sensor data (temperatures of 26.3°C and 26.1°C from DHT22 and BMP280, respectively, 40.2% humidity, and 908.84hPa pressure) and processed this information through the SLM, which produced a coherent response recommending the activation of the yellow LED due to elevated temperature conditions.

The model's ability to interpret sensor data and provide logical, rule-based decisions shows promise. Still, the simplistic nature of the current implementation (using basic thresholds and binary LED outputs) suggests significant room for improvement through more sophisticated prompting strategies, historical data integration, and the implementation of safety mechanisms. Also, the result is probabilistic, meaning it should change after execution.

Act on Output (Actuators)

Let's use the output generated and use it to actuate on the LEDs, our "actuators":



We can add a new function, `parse_llm_response()`, to return a command to the the LEDs based on the SLM's response:

```

def parse_llm_response(response_text):
    """Parse the LLM response to extract LED control instructions."""
    response_lower = response_text.lower()
    red_led = 'activate red led' in response_lower
    yellow_led = 'activate yellow led' in response_lower
    green_led = 'activate green led' in response_lower
    return (red_led, yellow_led, green_led)

```

The return to this function:

```
red, yellow, green = parse_llm_response(response['response'])
```

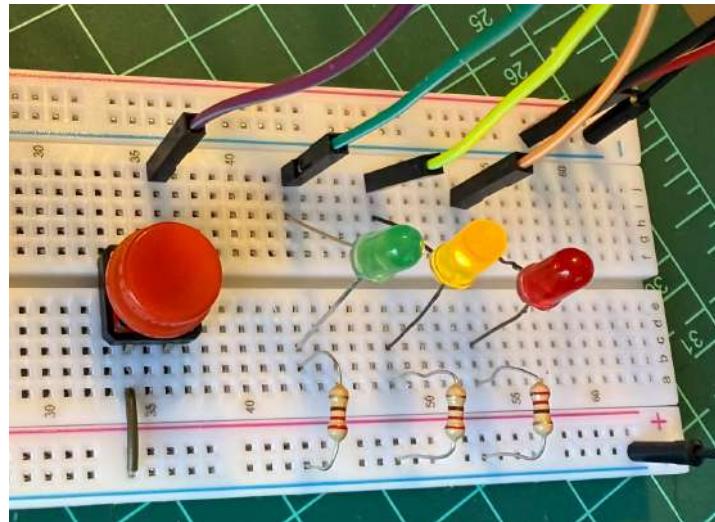
Would be:(`False, True, False`), which can be the input to the function `control_leds(red, yellow, green)`'

```

control_leds(red, yellow, green)
led_status()

```

```
(False, True, False)
```



Creating new functions for the Actuation:

One for the inference:

```
def slm_inference(PROMPT, MODEL):
    response = ollama.generate(
        model=MODEL,
        prompt=PROMPT
    )
    return response
```

And another for output analysis and actuation:

```
def output_actuator(response, MODEL):
    print(f"\nSmart IoT Actuator using {MODEL} model\n")

    print(f"SYSTEM REAL DATA")
    print(f" - DHT22 ==> Temp: {temp_dht:.1f}°C, Humidity: {hum:.1f}%")
    print(f" - BMP280 => Temp: {temp_bmp:.1f}°C, Pressure: {press:.2f}hPa")
    print(f" - Button {'pressed' if button_state else 'not pressed'}")

    print(f"\n>> {MODEL} Response: {response['response']}")

    # Control LEDs based on response
    red, yellow, green = parse_llm_response(response['response'])
    control_leds(red, yellow, green)
```

```

print(f"\nSYSTEM ACTUATOR STATUS")
ledRedSts, ledYlwSts, ledGrnSts = led_status()
print(f" - Red LED {'is on' if ledRedSts else 'is off'}")
print(f" - Yellow LED {'is on' if ledYlwSts else 'is off'}")
print(f" - Green LED {'is on' if ledGrnSts else 'is off'}")

```

By updating the system and calling the two functions in sequence, we can have the full cycle of analysis by the SLM and the actuation on the output LEDs:

```

ledRedSts, ledYlwSts, ledGrnSts = led_status()
temp_dht, hum, temp_bmp, press, button_state = collect_data()

response = slm_inference(PROMPT, MODEL)
output_actuator(response, MODEL)

```

The new script can be found on GitHub: [slm_basic_act_leds](#).

Let's press the button, call the functions, and see what happens:

```

ledRedSts, ledYlwSts, ledGrnSts = led_status()
temp_dht, hum, temp_bmp, press, button_state = collect_data()
response = slm_inference(PROMPT, MODEL)
output_actuator(response, MODEL)

```

Smart IoT Actuator using llama3.2:3b model

SYSTEM REAL DATA

- DHT22 ==> Temp: 20.6°C, Humidity: 31.3%
- BMP280 => Temp: 21.5°C, Pressure: 911.71hPa
- Button pressed

>> llama3.2:3b Response: Given the current temperature (21.7°C) which is below the threshold of 20°C and there are no other alarming values, I would not trigger any specific color change.

However, considering the high humidity level (39.8%), it might be a cause for concern if other parameters were also off the normal range.

Since there's only one parameter that's slightly above the warning zone and no other critical alarms, I will choose the green LED as an indicator of system status being OK.

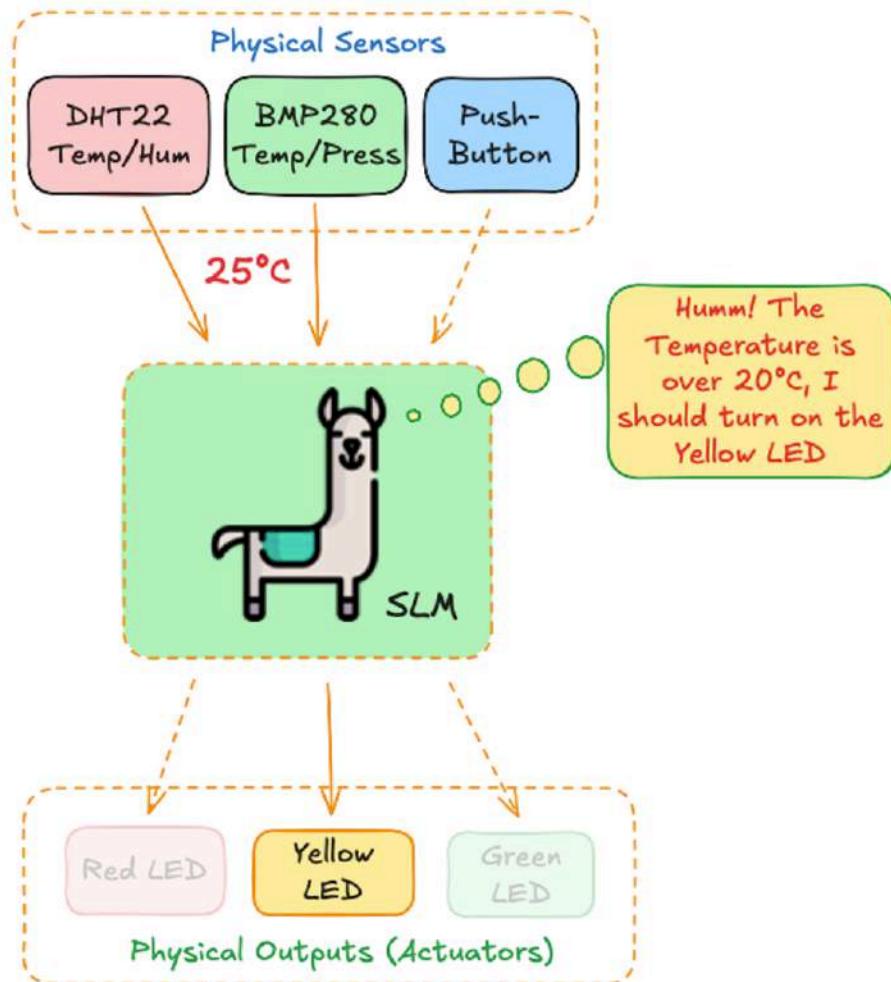
SYSTEM ACTUATOR STATUS

- Red LED is off
- Yellow LED is off
- Green LED is off

We can see that, despite the button being pressed, the SLM did not consider it and misinterpreted the temperature value as ABOVE the threshold, not below. Also, despite the fact that we asked for a straight answer about which LED to turn on, the model lost time with analysis,

This issue relates to the prompt we wrote. Let's cover it in the next section.

Prompting Engineering



Looking at the answers we got with the previous implementation, which are not always correct, we can see that the main issue is **unreliable text parsing**. Using JSON to parse the answer should be much better!

Key Changes in the code:

1. **Added JSON import**
2. **Fixed variable scope bug** - The previous code tried to use `temp_dht`, `hum`, etc. in the prompt *before* they were defined. The prompt is now created in a function after receiving

sensor data.

3. **Changed to JSON format** - The prompt now asks for a structured JSON response:

```
{"red_led": true, "yellow_led": false, "green_led": false}
```

4. **Updated parser** - Now parses JSON instead of searching for text strings. Includes error handling and fallback to safe state (all LEDs off) if parsing fails.

Why JSON is Better:

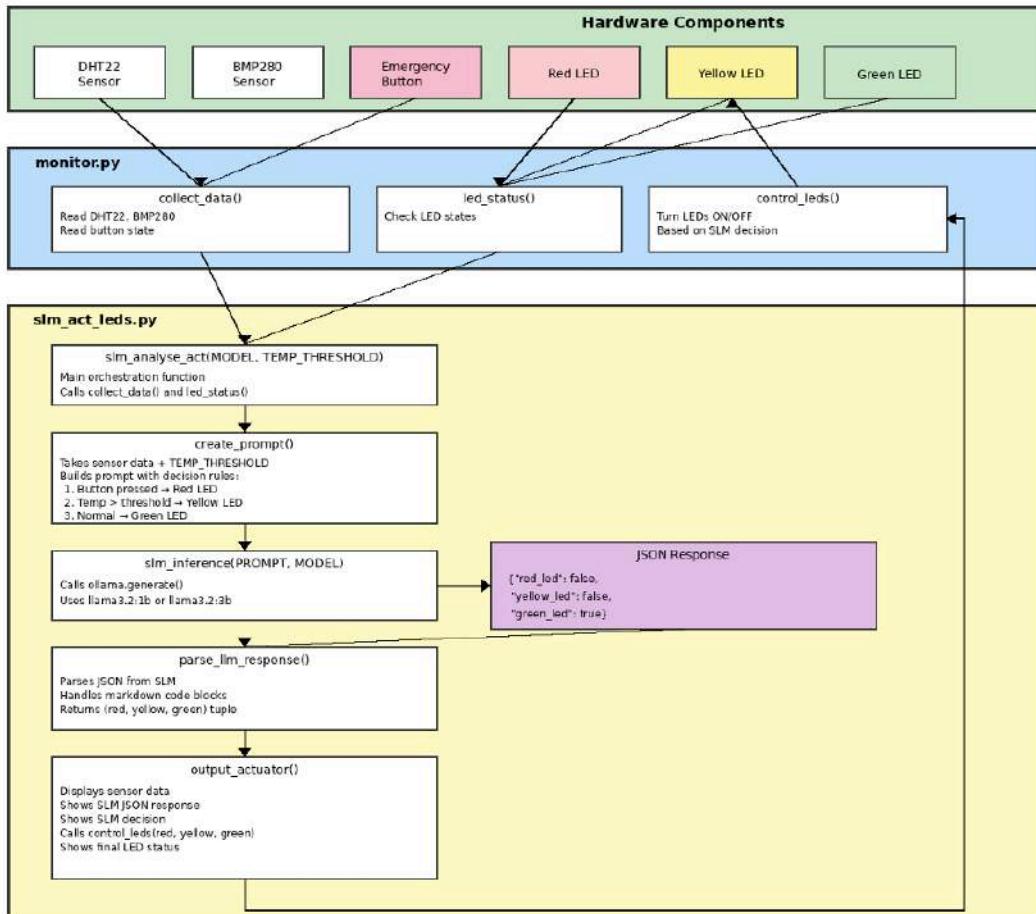
- **More reliable:** No ambiguity about which LEDs to activate
- **Structured:** Clear true/false values instead of parsing text
- **Error-resistant:** The parser handles markdown code blocks (some models wrap JSON in “`”) and provides safe fallback
- **Flexible:** Easy to add more fields later if needed

Let's revise the previous code, with a better prompt and parcing funcion, now based on JSON.

System Overview: Enhanced IoT Environmental Monitoring with SLM Control

This project demonstrates how a Small Language Model (SLM) can make intelligent decisions in an IoT system. The system monitors environmental conditions using sensors and controls LED indicators based on the SLM's analysis.

New System Architecture



The system is organized into three main layers:

1. Hardware Components Layer

The physical layer consists of:

- **DHT22 Sensor:** Measures temperature and humidity
- **BMP280 Sensor:** Measures temperature and atmospheric pressure
- **Emergency Button:** Manual override for emergency situations
- **Three LEDs:** Visual indicators
 - **Red LED:** Emergency/Problem state

- **Yellow LED:** Warning state
- **Green LED:** Normal operation

2. Hardware Interface Layer ([monitor.py](#))

This layer provides the bridge between hardware and software with three key functions:

- `collect_data()`: Reads all sensor values (temperature, humidity, pressure) and button state
- `led_status()`: Checks the current state of all LEDs
- `control_leds(red, yellow, green)`: Controls which LED is turned on based on boolean values

3. Intelligence Layer ([slm_act_leds.py](#))

This is where the SLM makes decisions. The workflow follows these steps:

a) Data Collection & Preparation

- `slm_analyse_act(MODEL, TEMP_THRESHOLD)`: Main function that orchestrates the entire process
 - Calls `collect_data()` to get current sensor readings
 - Calls `led_status()` to check LED states

b) Prompt Generation

- `create_prompt()`: Builds a detailed prompt for the SLM, including:
 - Current sensor readings
 - Temperature threshold value
 - Decision rules with clear priorities:
 1. Button pressed → Red LED (Emergency - highest priority)
 2. Temperature exceeds threshold → Yellow LED (Warning)
 3. Normal conditions → Green LED (All OK)
 - Pre-analyzed data to help the SLM understand the current state

c) SLM Inference

- `slm_inference(PROMPT, MODEL)`: Sends the prompt to the Ollama SLM (llama3.2:3b or any other SLM, as Gemma or Phi)
- The SLM analyzes the situation and returns a **JSON response**:

```

{
  "red_led": false,
  "yellow_led": false,
  "green_led": true
}

```

d) Response Processing

- `parse_llm_response()`: Parses the JSON response from the SLM
 - Handles edge cases like markdown code blocks
 - Extracts boolean values for each LED
 - Returns a tuple: (`red`, `yellow`, `green`)

e) Actuation

- `output_actuator()`: Takes the SLM's decision and executes it
 - Displays all sensor data
 - Shows the SLM's JSON response
 - Shows the final decision
 - Calls `control_leds()` to physically turn LEDs on/off
 - Displays final LED status for verification

How It Works: A Complete Cycle

1. **Sensors continuously monitor** the environment (temperature, humidity, pressure, button state)
2. **Data flows** from hardware through `monitor.py` to `slm_act_leds.py`
3. **The SLM receives** a structured prompt with current conditions and decision rules
4. **The SLM analyzes** the data and decides which LED should be activated
5. **The decision returns** as a JSON object specifying exactly which LED to turn on
6. **The system executes** the decision by controlling the LEDs through `monitor.py`
7. **Visual feedback** is provided via the LED, and the system logs all data and decisions

Key Design Principles

JSON Communication: Using JSON format ensures reliable, structured communication between the SLM and the actuation system, reducing parsing errors.

SLM-Driven Decisions: The SLM has complete control over LED decisions, demonstrating how AI can autonomously manage IoT systems.

Configurable Threshold: The temperature threshold is a parameter that makes the system adaptable to different environments without code changes. Can be updated by the user.

Clear Priority Rules: The system follows explicit priority rules that the SLM understands:

- Safety first (button press = emergency)
- Then environmental warnings (temperature)
- Then normal operation

Now, this architecture showcases how Small Language Models can be integrated into IoT systems to provide intelligent, context-aware decision-making while maintaining simplicity and reliability.

The new code

```
import ollama
import json
from monitor import collect_data, led_status, control_leds

def create_prompt(temp_dht, hum, temp_bmp, press, button_state,
                  ledRedSts, ledYlwSts, ledGrnSts, TEMP_THRESHOLD):
    """Create a prompt for the LLM with current sensor data."""
    return f"""
        You are controlling an IoT LED system. Analyze the sensor data and decide
        which ONE LED to activate.

        SENSOR DATA:
        - DHT22 Temperature: {temp_dht:.1f}°C
        - BMP280 Temperature: {temp_bmp:.1f}°C
        - Humidity: {hum:.1f}%
        - Pressure: {press:.2f}hPa
        - Button: {"PRESSED" if button_state else "NOT PRESSED"}

        TEMPERATURE THRESHOLD: {TEMP_THRESHOLD}°C

        DECISION RULES (apply in this priority order):
        1. IF button is PRESSED → Activate Red LED (EMERGENCY - highest priority)
        2. IF button is NOT PRESSED AND (DHT22 temp > {TEMP_THRESHOLD}°C OR
           BMP280 temp > {TEMP_THRESHOLD}°C) → Activate Yellow LED (WARNING)
        3. IF button is NOT PRESSED AND (DHT22 temp < {TEMP_THRESHOLD}°C AND
           BMP280 temp < {TEMP_THRESHOLD}°C) → Activate Green LED (NORMAL)
```

```

CURRENT ANALYSIS:
- Button status: {"PRESSED" if button_state else "NOT PRESSED"}
- DHT22 temp ({temp_dht:.1f}°C) is {"OVER" if temp_dht > TEMP_THRESHOLD
  else "AT OR BELOW"} threshold ({TEMP_THRESHOLD}°C)
- BMP280 temp ({temp_bmp:.1f}°C) is {"OVER" if temp_bmp > TEMP_THRESHOLD
  else "AT OR BELOW"} threshold ({TEMP_THRESHOLD}°C)

```

Based on these rules, respond with ONLY a JSON object (no other text):
 {{"red_led": true, "yellow_led": false, "green_led": false}}

Only ONE LED should be true, the other two must be false.

"""

```

def parse_llm_response(response_text):
    """Parse the LLM JSON response to extract LED control instructions."""
    try:
        # Clean the response - remove any markdown code blocks if present
        response_text = response_text.strip()
        if response_text.startswith('```'):
            # Extract JSON from markdown code block
            lines = response_text.split('\n')
            response_text = '\n'.join(lines[1:-1])
            if len(lines) > 2
                else response_text

        # Parse JSON
        data = json.loads(response_text)
        red_led = data.get('red_led', False)
        yellow_led = data.get('yellow_led', False)
        green_led = data.get('green_led', False)
        return (red_led, yellow_led, green_led)
    except (json.JSONDecodeError, KeyError) as e:
        print(f"Error parsing JSON response: {e}")
        print(f"Response was: {response_text}")
        # Fallback to safe state (all LEDs off)
        return (False, False, False)

def output_actuator(response, MODEL, temp_dht, hum, temp_bmp,
                    press, button_state):
    print(f"\nSmart IoT Actuator using {MODEL} model\n")

```

```

print(f"SYSTEM REAL DATA")
print(f" - DHT22 ==> Temp: {temp_dht:.1f}°C, Humidity: {hum:.1f}%")
print(f" - BMP280 => Temp: {temp_bmp:.1f}°C, Pressure: {press:.2f}hPa")
print(f" - Button {'pressed' if button_state else 'not pressed'}")

print(f"\n>> {MODEL} Response: {response['response']}")

# Parse LLM response and use it directly (no validation)
red, yellow, green = parse_llm_response(response['response'])
print(f">> SLM decision: Red={red}, Yellow={yellow}, Green={green}")

# Control LEDs based on SLM decision
control_leds(red, yellow, green)

print(f"\nSYSTEM ACTUATOR STATUS")
ledRedSts, ledYlwSts, ledGrnSts = led_status()
print(f" - Red LED {'is on' if ledRedSts else 'is off'}")
print(f" - Yellow LED {'is on' if ledYlwSts else 'is off'}")
print(f" - Green LED {'is on' if ledGrnSts else 'is off'}")

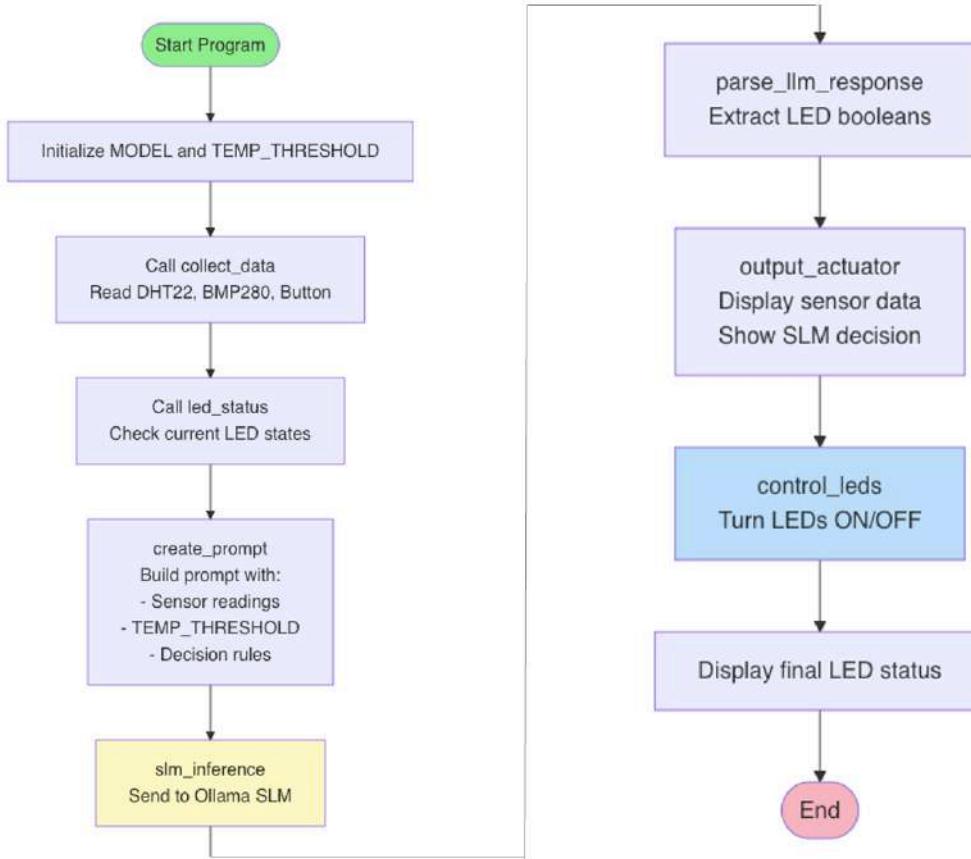
def slm_analyse_act(MODEL, TEMP_THRESHOLD):
    """Main function to get sensor data, run SLM inference, and actuate LEDs."""
    # Get system info
    ledRedSts, ledYlwSts, ledGrnSts = led_status()
    temp_dht, hum, temp_bmp, press, button_state = collect_data()

    # Create prompt with current sensor data
    PROMPT = create_prompt(temp_dht,
                           hum,
                           temp_bmp,
                           press,
                           button_state,
                           ledRedSts,
                           ledYlwSts,
                           ledGrnSts,
                           TEMP_THRESHOLD)

    # Analyse and actuate on LEDs
    response = slm_inference(PROMPT, MODEL)
    output_actuator(response, MODEL, temp_dht, hum, temp_bmp, press, button_state)

```

Code Flow Diagram



TEST1: Temp above the threshold

Definitions

```

# Model to be used
MODEL = 'llama3.2:3b'

# Temperature threshold for warning
TEMP_THRESHOLD = 20.0
  
```

Calling the program

```
slm_analyse_act(MODEL, TEMP_THRESHOLD)
```

Result

Smart IoT Actuator using llama3.2:3b model

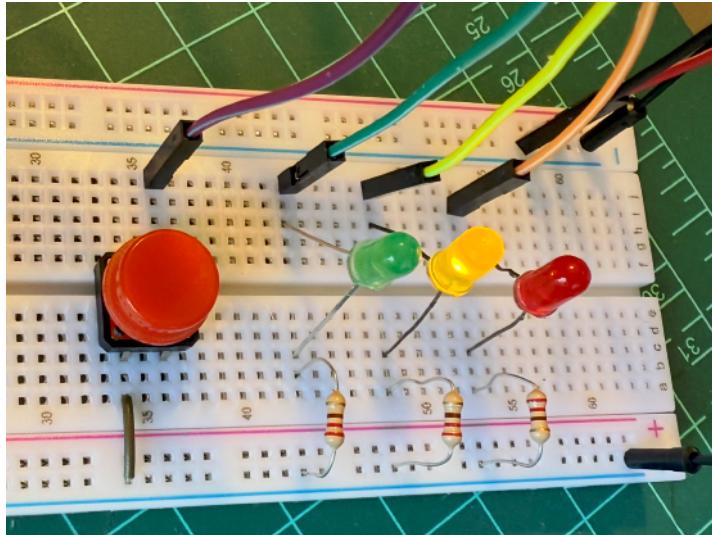
SYSTEM REAL DATA

- DHT22 ==> Temp: 21.5°C, Humidity: 28.8%
- BMP280 => Temp: 22.4°C, Pressure: 910.04hPa
- Button not pressed

```
>> llama3.2:3b Response: {"red_led": false, "yellow_led": true, "green_led": false}  
>> SLM decision: Red=False, Yellow=True, Green=False
```

SYSTEM ACTUATOR STATUS

- Red LED is off
- Yellow LED is on
- Green LED is off



TEST2: Temp below the threshold

```
# Temperature threshold for warning  
TEMP_THRESHOLD = 25.0
```

```
slm_analyse_act(MODEL, TEMP_THRESHOLD)
```

Smart IoT Actuator using llama3.2:3b model

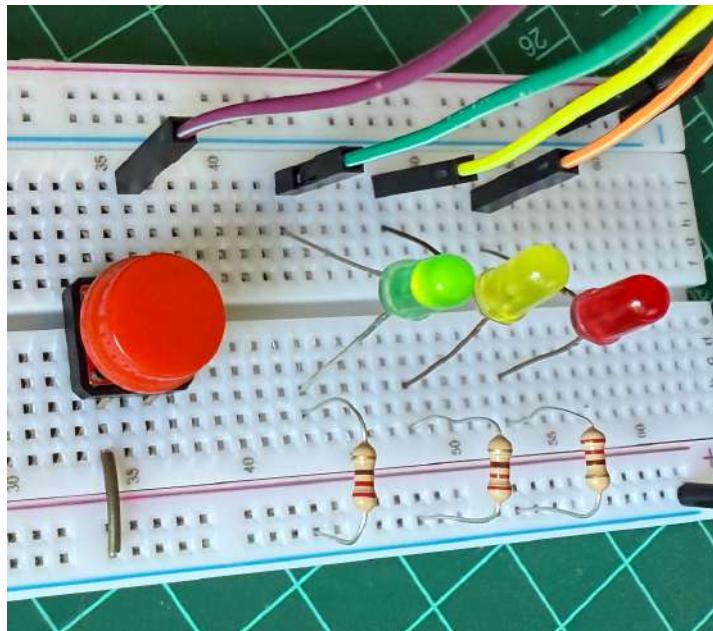
SYSTEM REAL DATA

- DHT22 ==> Temp: 21.6°C, Humidity: 29.1%
- BMP280 => Temp: 22.4°C, Pressure: 910.07hPa
- Button not pressed

```
>> llama3.2:3b Response: {"red_led": false, "yellow_led": false, "green_led": true}  
>> SLM decision: Red=False, Yellow=False, Green=True
```

SYSTEM ACTUATOR STATUS

- Red LED is off
- Yellow LED is off
- Green LED is on



TEST3: Alarm Button pressed

```
slm_analyse_act(MODEL, TEMP_THRESHOLD)
```

```
Smart IoT Actuator using llama3.2:3b model
```

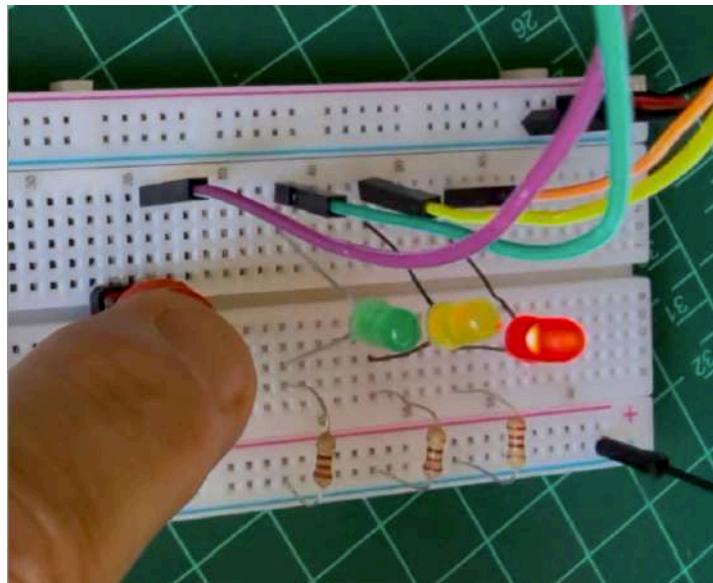
```
SYSTEM REAL DATA
```

- DHT22 ==> Temp: 21.6°C, Humidity: 29.0%
- BMP280 => Temp: 22.5°C, Pressure: 909.99hPa
- Button pressed

```
>> llama3.2:3b Response: {"red_led": true, "yellow_led": false, "green_led": false}  
>> SLM decision: Red=True, Yellow=False, Green=False
```

```
SYSTEM ACTUATOR STATUS
```

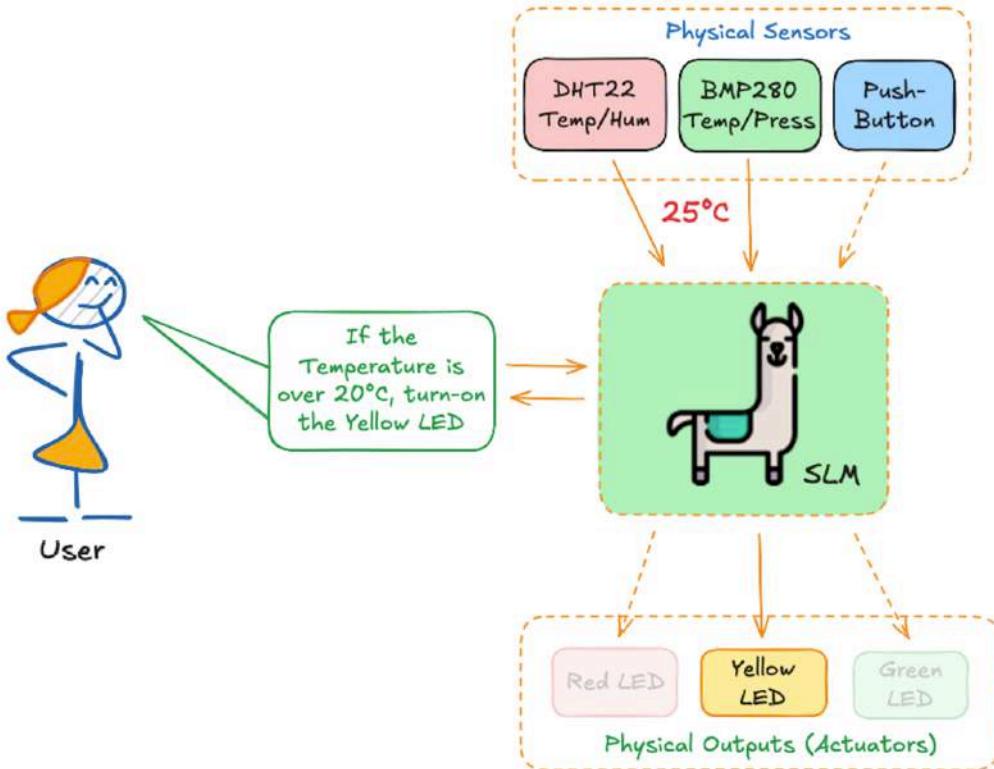
- Red LED is on
- Yellow LED is off
- Green LED is off



Interacting with IoT Systems, using Natural Language Commands

Now, let's transform our IoT monitoring setup into an **interactive assistant** that accepts natural language commands and queries. Instead of autonomously monitoring conditions, the

system will now respond to our requests in real-time.



What will change?

Original System (Autonomous)

- Continuously monitored sensors
- Automatically decided LED states based on pre-defined rules
- No user interaction

New System (Interactive)

- Waits for user commands
- Accepts natural language queries and commands
- Provides conversational responses
- Executes actions based on user requests
- Displays comprehensive system status

How It will Work

1. Interactive Loop

The system runs in a continuous loop:

User Input → Sensor Reading → SLM Analysis → LED Control → Status Display → Wait for Next Input

2. Dual-Purpose Response

The SLM now returns a JSON object with TWO components:

```
{  
  "message": "Helpful text response to the user",  
  "leds": {  
    "red_led": false,  
    "yellow_led": true,  
    "green_led": false  
  }  
}
```

- **message**: What the assistant tells you (conversational response)
- **leds**: What the assistant does (LED control)

3. Command Types

A. Information Queries

Ask questions about sensor readings - LEDs remain unchanged.

Examples:

- “What’s the current temperature?”
- “What are the actual conditions?”
- “What’s the humidity level?”
- “Is the button pressed?”

Response format:

```
{  
  "message": "The current temperature is 21.5°C from DHT22 and 22.3°C from BMP280.",  
  "leds": {"red_led": false, "yellow_led": true, "green_led": false} // keeps current sta
```

```
}
```

B. Direct LED Commands

Tell the system which LEDs to turn on/off.

Examples:

- “Turn on the yellow LED”
- “Turn on all LEDs”
- “Turn off all LEDs”
- “Turn on the red and green LEDs”

Response format:

```
{
  "message": "Yellow LED turned on.",
  "leds": {"red_led": false, "yellow_led": true, "green_led": false}
}
```

C. Conditional Commands

Commands that depend on sensor readings or button state.

Examples:

- “If the temperature is above 20°C, turn on the yellow LED”
- “If the button is pressed, turn on the red LED”
- “If the button is not pressed, turn on the green LED”

Response format:

```
{
  "message": "Temperature is 21.5°C, which is above 20°C. Yellow LED turned on.",
  "leds": {"red_led": false, "yellow_led": true, "green_led": false}
}
```

D. Toggle/Switch Commands

Commands that change LED states based on current conditions.

Examples:

- “If the button is pressed, switch the LED conditions”
- “Toggle all LEDs”

Response format:

```
{  
  "message": "Button is pressed. LED states switched.",  
  "leds": {"red_led": true, "yellow_led": false, "green_led": false} // inverted from cur  
}
```

E. Analysis Queries

Ask the SLM to analyze sensor data.

Examples:

- “Based on the conditions, would we have rain?”
- “Is the weather getting warmer?”
- “What do the sensor readings indicate?”

Response format:

```
{  
  "message": "Based on pressure of 910.18hPa and humidity of 28.8%, conditions are dry. Ra  
  "leds": {"red_led": false, "yellow_led": false, "green_led": true} // keeps current sta  
}
```

Key Functions

0.0.0.0.1 * create_interactive_prompt()

- Creates a comprehensive prompt with:
 - Current sensor readings
 - Current LED states
 - User’s request
 - Examples of how to respond
 - Rules for the SLM to follow

0.0.0.0.2 * parse_interactive_response()

- Extracts the message from the SLM
- Extracts the LED control commands
- Handles JSON parsing errors gracefully

0.0.0.0.3 * display_system_status()

- Shows comprehensive system status:
 - All sensor readings
 - Button state
 - LED states (with visual indicators)

0.0.0.0.4 * interactive_mode()

- Main loop that:
 - Accepts user input
 - Reads current sensor data
 - Sends request to SLM
 - Parses response
 - Controls LEDs
 - Displays results

Here is the complete code: [slm_act_leds_interactive.py](#)

Running the System

```
python slm_act_leds_interactive.py
```

Tests

```
=====
IoT Environmental Monitoring System - Interactive Mode
Using Model: llama3.2:3b
=====
```

```
Commands you can try:
```

- What's the current temperature?
- What are the actual conditions?
- Turn on the yellow LED
- If temperature is above 20°C, turn on yellow LED
- If button is pressed, turn on red LED
- Turn on all LEDs
- Turn off all LEDs
- Will it rain based on current conditions?

```
- Type 'status' to see system status  
- Type 'exit' or 'quit' to stop
```

You: what's the current temperature?

Assistant: [Thinking...]

Assistant: The current temperature is 21.5°C from DHT22 and 22.3°C from BMP280.

LED Update: Red=OFF, Yellow=OFF, Green=ON

You: if temperature is above 20°C, turn on yellow LED

Assistant: [Thinking...]

Assistant: Temperature is 21.5°C, which is above 20°C. Yellow LED turned on.

LED Update: Red=OFF, Yellow=ON, Green=OFF

You: will it rain?

Assistant: [Thinking...]

Assistant: Based on pressure of 910.18hPa and humidity of 28.8%, the air is relatively dry. Rain is unlikely in the immediate future. LEDs unchanged.

LED Update: Red=OFF, Yellow=ON, Green=OFF

You: status

SYSTEM STATUS

DHT22 Sensor: Temp = 21.5°C, Humidity = 28.8%

BMP280 Sensor: Temp = 22.3°C, Pressure = 910.18hPa

Button: NOT PRESSED

LED Status:

Red LED: OFF

Yellow LED: ON

Green LED: OFF

You: exit

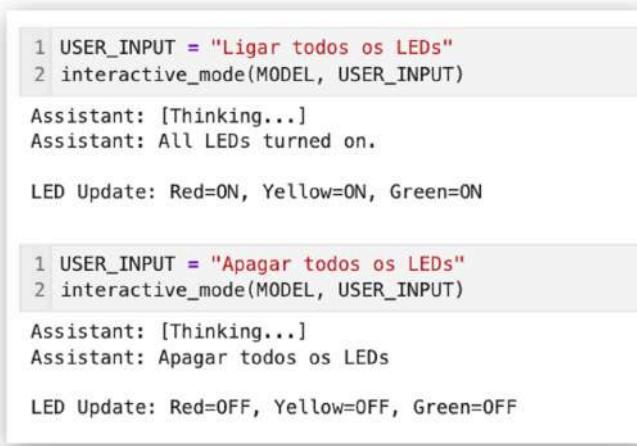
Exiting interactive mode. Goodbye!

Special Commands

- **status:** Display comprehensive system status without calling the SLM
- **exit or quit:** Exit the interactive mode

Languages

One advantage of using SLMs is that, once multilingual models are used (as Llama 3,2 or Gemma), the user can choose the better language for them, independent of the language used during coding.



```
1 USER_INPUT = "Ligar todos os LEDs"
2 interactive_mode(MODEL, USER_INPUT)
Assistant: [Thinking...]
Assistant: All LEDs turned on.

LED Update: Red=ON, Yellow=ON, Green=ON

1 USER_INPUT = "Apagar todos os LEDs"
2 interactive_mode(MODEL, USER_INPUT)
Assistant: [Thinking...]
Assistant: Apagar todos os LEDs

LED Update: Red=OFF, Yellow=OFF, Green=OFF
```

Advantages of This Approach

1. **Natural Language Interface:** Users don't need to know programming or specific commands
2. **Flexible Control:** Can handle complex conditional logic based on sensor data
3. **Informational:** Can answer questions about the system without changing states
4. **Context-Aware:** SLM understands current conditions and makes appropriate decisions
5. **Conversational:** Provides helpful feedback about what it's doing

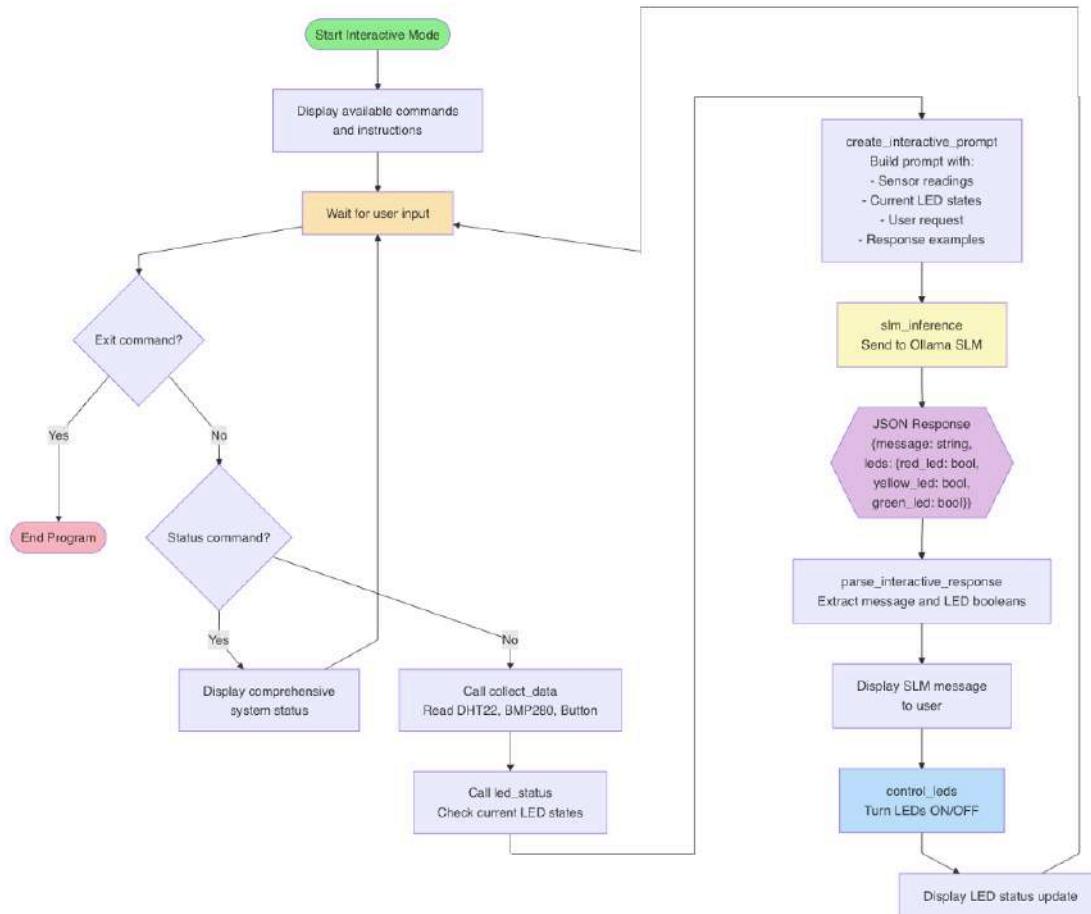
Error Handling

- If sensor data cannot be read, the system will notify you and wait for the next command
- If the SLM response cannot be parsed, the system keeps LEDs in their current state

Tips for Best Results

1. **Be specific:** “Turn on the yellow LED” works better than “turn on the light”
2. **Use conditions clearly:** “If temperature is above 20°C” is clearer than “when it’s hot”
3. **Ask for status:** Use the `status` command frequently to verify system state
4. **One LED at a time:** Unless you specifically say “all LEDs”, the system defaults to one LED on

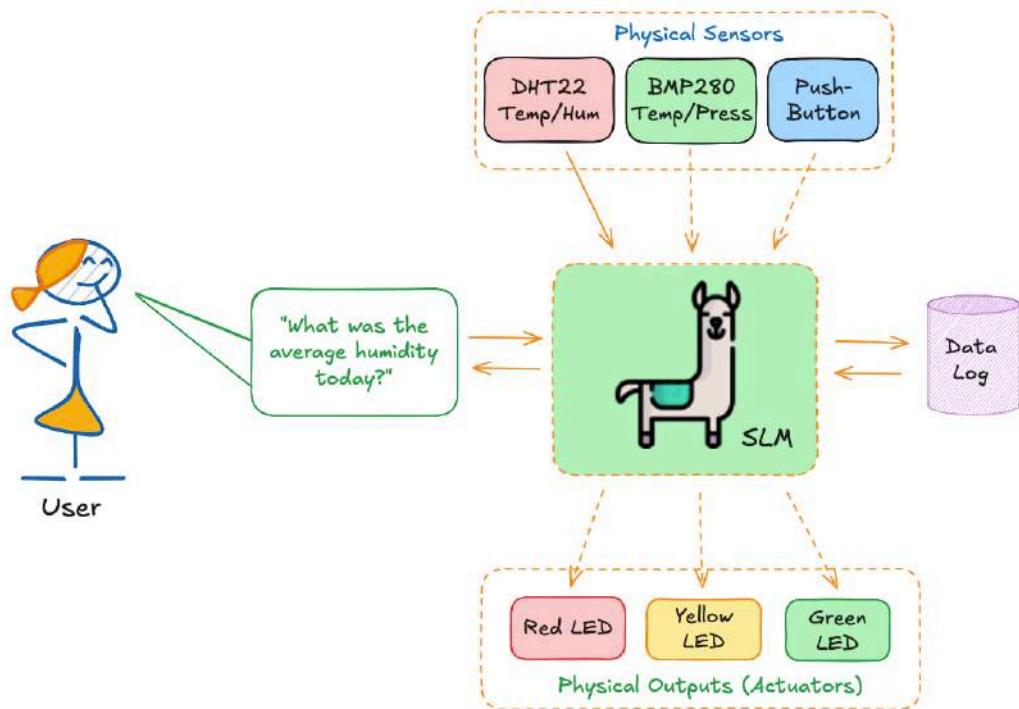
Flow Diagram



The development of our Interactive IoT-SLM system can also be followed using the Jupyter Notebook: [SLM_IoT.ipynb](#).

Adding Data Logging

Now, we will develop an enhanced version that adds data logging, analysis, and historical query capabilities. The system automatically logs all sensor readings and commands, allowing it to analyze trends and query historical data using natural language.



Key Features

1. Automatic Background Logging

- Logs sensor readings every 60 seconds
- Runs in background thread
- Stores data in CSV files

2. CSV Data Storage

- sensor_readings.csv: All sensor data + LED states
- command_history.csv: All commands and responses

3. Statistical Analysis

- Min/max/average calculations
- LED state change tracking
- Button press counting
- Trend analysis

4. Natural Language Queries

Ask questions like:

- “Show me temperature trends for the last 24 hours”
- “What was the average humidity today?”
- “How many times was the button pressed?”

Running the System

```
python slm_act_leds_with_logging.py
```

Example Queries

Real-time:

- “What’s the current temperature?”
- “Turn on the yellow LED”

Historical:

- “Show me temperature trends”
- “What was the average humidity today?”
- “How many times was the button pressed?”

Built-in commands:

- **status** - Show current system status
- **stats** - Show 24-hour statistics
- **exit** - Stop system

Data Files

sensor_readings.csv

```
timestamp,temp_dht,humidity,temp_bmp,pressure,  
button_pressed,red_led,yellow_led,green_led
```

command_history.csv

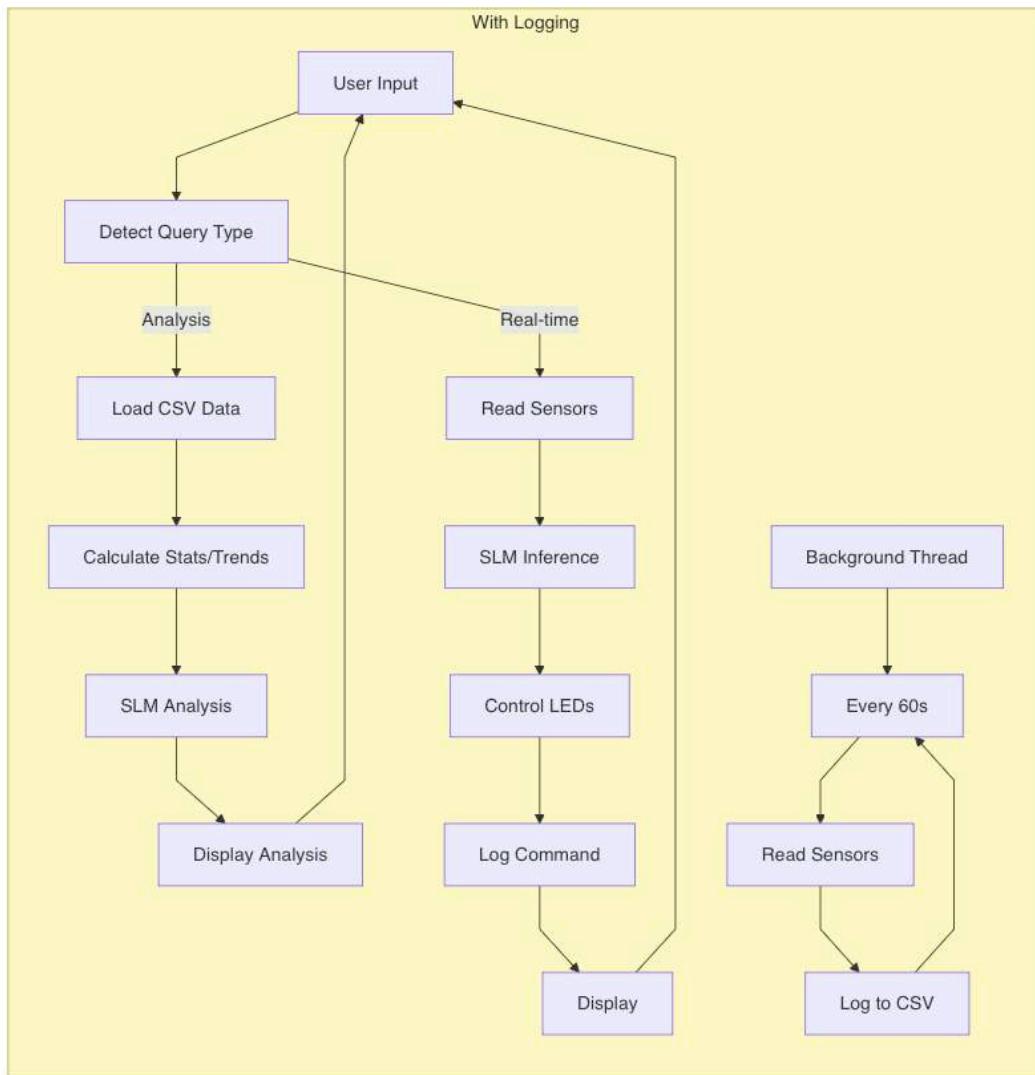
```
timestamp,user_command,slm_response,red_led,yellow_led,green_led
```

Tips

1. Let system run 1+ hour for meaningful trends
2. Use specific time frames: “last 6 hours”
3. Use **stats** command for quick overview
4. CSV files can be opened in Excel

The complete scripts for the datalogger version are here: [data_logger.py](#) and [slm_act_leds_with_logging.py](#)

Flow Diagram



Prompt Optimization and Efficiency

We can modify for example the `interactive_mode(MODEL, USER_INPUT)` function, to include at its end, some statistics about the model's performance:

```

...
# Display Latency
    print(f"\nTotal Duration: {(response['total_duration']/1e9):.2f} seconds")
    print(f"prompt_eval_duration: {(response['prompt_eval_duration']/1e9):.2f} s")
    print(f"load_duration: {(response['load_duration']/1e9):.2f} s")
    print(f"eval_count: {response['eval_count']}")
    print(f"eval_duration: {(response['eval_duration']/1e9):.2f} s")
    print(f"eval_rate: {response['eval_count']/(response['eval_duration']/1e9):.2f} \
tokens/s")

```

For example, running

```

USER_INPUT = "Turn off all LEDs"
interactive_mode(MODEL, USER_INPUT)

```

We will get:

```

Total Duration: 88.81 seconds
prompt_eval_duration: 80.41 s
load_duration: 1.98 s
eval_count: 32
eval_duration: 6.41 s
eval_rate: 4.99 tokens/s

```

Based on the `prompt_eval_duration` value, the PROMPT is our main bottleneck, so we must make it as concise as possible. The SLM must process all of the context *before* it can generate the first output token.

Quick Solution:

1. Drastically Shorten the Prompt

- **Condense System Status:** Remove unnecessary descriptive text and present the status information in a compact, structured format.

- **Example (Before):**

```

CURRENT SYSTEM STATUS:
- DHT22: Temperature 25.5°C, Humidity 45.1%
- BMP280: Temperature 25.6°C, Pressure 1012.34hPa
- Button: NOT PRESSED
- Red LED: OFF

```

- Yellow LED: OFF
- Green LED: ON

– **Example (After):**

```
STATUS: DHT22=22.5°C/65.0% BMP280=22.3°C/1013.25hPa Button=OFF LEDs:R=OFF/Y=ON/G=
```

- **Reduce Examples:** While examples are crucial for instruction-following, **eliminate redundancy**. Keep only the most diverse and representative examples. The current prompt is very long due to verbose examples and instructions. **Focus on the single-shot example** that shows the *required JSON output format*.
- **Simplify Instructions:** Make the instructions as direct and short as possible. Use keywords instead of full sentences where clarity is maintained.
- **System message:** It defines the assistant's behavior and should sent once at initialization, not at PROMPT

```
SYSTEM_MESSAGE = """You are an IoT assistant controlling an environmental monitoring system with LEDs.
```

Respond with JSON only:

```
{"message": "your helpful response", "leds": {"red_led": bool, "yellow_led": bool, "green_led": bool}}
```

RULES:

- Information queries: keep current LED states unchanged
- LED commands: update LEDs as requested
- Conditional commands (if/when): evaluate condition from sensor data first
- Only ONE LED should be on at a time UNLESS user explicitly says "all"
- Be concise and conversational

```
Always respond with valid JSON containing both "message" and "leds" fields."""
```

2. chat API Instead of generate

- Uses `ollama.chat()` to maintain conversation context
- System message sent only once at initialization
- Subsequent messages are much shorter (only status + user query)

Smart Context Management

- Keeps last four conversation exchanges (8 messages) + system message
- Prevents context from growing too large over time

Model Pre-loading

- Loads model into memory before first query
- Eliminates the 2-second load delay on first run

Let's re-create the functions or add new as below:

New Code

Original Functions:

- `create_interactive_prompt()` - Now creates compact user messages (was ~800 tokens, now ~100 tokens)
- `slm_inference()` - Now uses `ollama.chat()` API instead of `ollama.generate()`
- `parse_interactive_response()` - Unchanged
- `display_system_status()` - Unchanged
- `interactive_mode()` - Updated with all optimizations

New functions:

- `SYSTEM_MESSAGE` - Constant for system prompt (sent only once)
- `preload_model()` - Helper to pre-load model into memory

Key Changes Under the Hood:

1. `create_interactive_prompt()` now returns a compact format:
 - Before: Long prompt with 8 examples (~800 tokens)
 - After: Compact status line + user input (~100 tokens)
2. `slm_inference()` now uses chat API:
 - Before: `ollama.generate(model, prompt)` - regenerates context each time
 - After: `ollama.chat(model, messages)` - maintains conversation context

3. System prompt is sent only once at initialization, not with every query

Using the optimized code: `slm_act_leds_interactive_optimized.py`, we get as a response:

```
=====
Pre-loading model llama3.2:3b...
Model llama3.2:3b loaded successfully!

You: Turn on all LEDs
Assistant: [Thinking...]
Assistant: All LEDs are now ON

LED Update: Red=ON, Yellow=ON, Green=ON

Total Duration: 26.46 seconds
prompt_eval_duration: 20.26 s
load_duration: 0.16 s
eval_count: 33
eval_duration: 6.03 s
eval_rate: 5.47 tokens/s

You: Turn off all LEDS
Assistant: [Thinking...]
Assistant: All LEDs are now OFF

LED Update: Red=OFF, Yellow=OFF, Green=OFF

Total Duration: 13.01 seconds
prompt_eval_duration: 6.54 s
load_duration: 0.16 s
eval_count: 33
eval_duration: 6.31 s
eval_rate: 5.23 tokens/s

You: quit

Exiting interactive mode. Goodbye!
```

The Prompt evaluation time was reduced drastically!

Using Pydantic

Using **Pydantic** is a robust way to improve the reliability, efficiency, and maintainability of our system, mainly since we rely on the LLM to output precise JSON.

Here's how Pydantic can help and what you would need to do:

1. How Pydantic Reduces Latency and Improves Reliability

Pydantic doesn't directly speed up the model's token generation, but it can **indirectly reduce latency** and **eliminate error-handling overhead** by enabling cleaner, faster parsing and more robust communication.

0.0.0.0.1 * A. Reliable and Fast JSON Parsing

The most critical benefit is moving from the general, potentially brittle `json.loads(response_text)` in our `parse_interactive_response` function to Pydantic's fast validation engine.

- **Pydantic's JSON Parsing is C-Accelerated:** Pydantic is built on top of the fast `pydantic-core` library written in Rust, which is significantly faster than Python's standard `json` module, especially for large or complex data structures.
- **Structured Output:** Our current code manually handles potential JSON formatting issues (such as the `jsoncode block delimiters`) and then uses a `try/except` block to extract fields. Pydantic handles all of this automatically and throws a clean error if the structure is wrong.

0.0.0.0.2 * B. Stronger Instruction for the SLM

The model is more likely to generate the *exact* JSON required when you give it a formal schema to follow.

- **JSON Schema:** Pydantic can generate a **JSON Schema** from your Python classes. We can include this schema directly in our prompt, which acts as an unambiguous, machine-readable instruction for the SLM. This often leads to fewer errors in the model's output, reducing the need for costly retries or complex string manipulation.

0.0.0.0.3 * C. Cleaner Python Code

It simplifies our `parse_interactive_response` function immensely, making your code easier to read and debug.

2. Implementing Pydantic in the Code

Let's test it on the Optimized Interactive version: `slm_act_leds_optimized.py`

We should modify the workflow in three main steps:

Step 1: Define the Pydantic Models

We must replace the implicit structure with explicit Pydantic models:

```
from pydantic import BaseModel, Field

class LEDControl(BaseModel):
    """LED control configuration."""
    red_led: bool = Field(description="Red LED state (on/off)")
    yellow_led: bool = Field(description="Yellow LED state (on/off)")
    green_led: bool = Field(description="Green LED state (on/off)")

class AssistantResponse(BaseModel):
    """Complete assistant response with message and LED control."""
    message: str = Field(description="Helpful response to the user")
    leds: LEDControl = Field(description="LED control configuration")
```

Step 2: Update the SLM inference function

We would update the `create_interactive_prompt` to include the schema:

```
def slm_inference(messages, MODEL):
    """Send chat request to Ollama using chat API with structured output
    (Pydantic)."""
    response = ollama.chat(
        model=MODEL,
        messages=messages,
        format=AssistantResponse.model_json_schema() # Using Pydantic schema
    )
    return response
```

Step 3: Update the Parsing Function

Our `parse_interactive_response` becomes much cleaner and more reliable:

```
def parse_interactive_response(response_text):
    """Parse the interactive SLM response using Pydantic (guaranteed valid)."""
    try:
        # Parse directly into Pydantic model - guaranteed valid JSON structure
```

```

data = AssistantResponse.model_validate_json(response_text)

# Extract values from Pydantic model
message = data.message
red_led = data.leds.red_led
yellow_led = data.leds.yellow_led
green_led = data.leds.green_led

return message, (red_led, yellow_led, green_led)

except Exception as e:
    print(f"Error parsing response: {e}")
    print(f"Response was: {response_text}")
    return "Error: Could not parse SLM response.", (False, False, False)

```

With such modifications, the final latency was reduced from 90 to around 60 seconds

```

You: If the temperature is over 20oC, turn on the red LED
Assistant: [Thinking...]
Assistant: Red LED is now ON because the temperature (21.4°C) is above the threshold.

LED Update: Red=ON, Yellow=OFF, Green=OFF

Total Duration: 16.65 seconds
prompt_eval_duration: 7.76 s
load_duration: 0.16 s
eval_count: 46
eval_duration: 8.73 s
eval_rate: 5.27 tokens/s

You: If the button is pressed, turn on the led yellow
Assistant: [Thinking...]
Assistant: Yellow LED is now ON because the button was pressed.

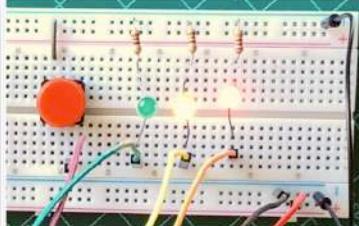
LED Update: Red=ON, Yellow=ON, Green=OFF

Total Duration: 14.93 seconds
prompt_eval_duration: 7.39 s
load_duration: 0.16 s
eval_count: 38
eval_duration: 7.37 s
eval_rate: 5.15 tokens/s

You: quit

Exiting interactive mode. Goodbye!

```



Note that when we sent two different commands to turn on the LEDs, the new one did not turn off the previous one. This is due to the change in the rules. If we want, we can modify it to match whatever we wish to.

The final code can be found on GitHub: [slm_act_leds_interactive_pydantic.py](#)
and in notebook [SLM_IoT.ipynb](#)

In short, using Pydantic over tradicional approach with JSON, we have as benefits:

- **No more parsing errors** - Guaranteed valid JSON
- **Cleaner output** - No markdown code blocks
- **Type safety** - IDE autocomplete and type checking
- **Faster generation** - Constrained decoding is more efficient
- **Better validation** - Pydantic catches invalid values

Next Steps

This chapter involved experimenting with simple applications and verifying the feasibility of using an SLM to control IoT devices. The final result is far from usable in the real world, but it can serve as a starting point for more interesting applications. Below are some observations and suggestions for improvement:

- SLM responses can be probabilistic and inconsistent. To increase reliability, consider implementing a confidence threshold or voting system using multiple prompts/responses.
- Try to add data validation and sanity checks for sensor readings before passing them to the SLM.
- Apply **Structured Response Parsing** as discussed early. Future improvements in this approach could include:
 - Add more sophisticated validation rules
 - Implement command history tracking
 - Add support for compound commands
 - Integrate with the logging system
 - Add user permission levels
 - Implement command templates for common operations
- Consider implementing a fallback mechanism when SLM responses are ambiguous or inconsistent.
- Study using RAG and fine-tuning to increase the system's reliability when using very small models.
- Consider adding input validation for user commands to prevent potential issues.
- The current implementation queries the SLM for every command. We did it to study how SLMs would behave. We should consider implementing a caching mechanism for common queries.

- Some simple commands could be handled without SLM intervention. We can do it programmatically.
- Consider implementing a proper state machine for LED control to ensure consistent behavior.
- Implement more sophisticated trend analysis using statistical methods.
- Add support for more complex queries combining multiple data points.

Conclusion

This chapter has demonstrated the progressive evolution of an IoT system from basic sensor integration to an intelligent, interactive platform powered by Small Language Models. Through our journey, we've explored several key aspects of combining edge AI with physical computing:

Key Achievements

1. Progressive System Development

- Started with basic sensor integration and LED control
- Advanced to SLM-based analysis and decision making
- Implemented natural language interaction
- Added historical data logging and analysis
- Created a complete interactive system

2. SLM Integration Insights

- Demonstrated the feasibility of using SLMs for IoT control
- Explored different models and their capabilities
- Implemented various prompting strategies
- Handled both real-time and historical data analysis

3. Practical Learning Outcomes

- Hardware-software integration techniques
- Real-time sensor data processing
- Natural language command interpretation
- Data logging and trend analysis
- Error handling and system reliability

Challenges and Limitations

Our implementation revealed several important challenges:

1. SLM Reliability

- Probabilistic nature of responses

- Consistency issues in decision making
- Need for better validation and verification

2. System Performance

- Response time considerations
- Resource usage on edge devices
- Efficiency of data logging and analysis

3. Architectural Constraints

- Simple state management
- Basic error handling
- Limited data validation

Final Thoughts

While this implementation demonstrates the potential of combining SLMs with IoT systems, it also highlights the exciting possibilities and challenges ahead. Though experimental, the system we've built provides a solid foundation for understanding how edge AI can enhance IoT applications. As SLMs evolve and improve, their integration with physical computing systems will likely become more robust and practical for real-world applications.

This chapter has shown that, despite current limitations, SLMs can provide intelligent, natural-language interfaces to IoT systems, opening new possibilities for human-machine interaction in the physical world.

The future of IoT systems is shaped by intelligent, edge-based solutions that combine AI's power with the practicality of physical computing.

Resources

- [Python Scripts and notebook](#)

Advancing EdgeAI: Beyond Basic SLMs

Exploring CoT Prompting, Agents, Function Calling, RAG, and more.

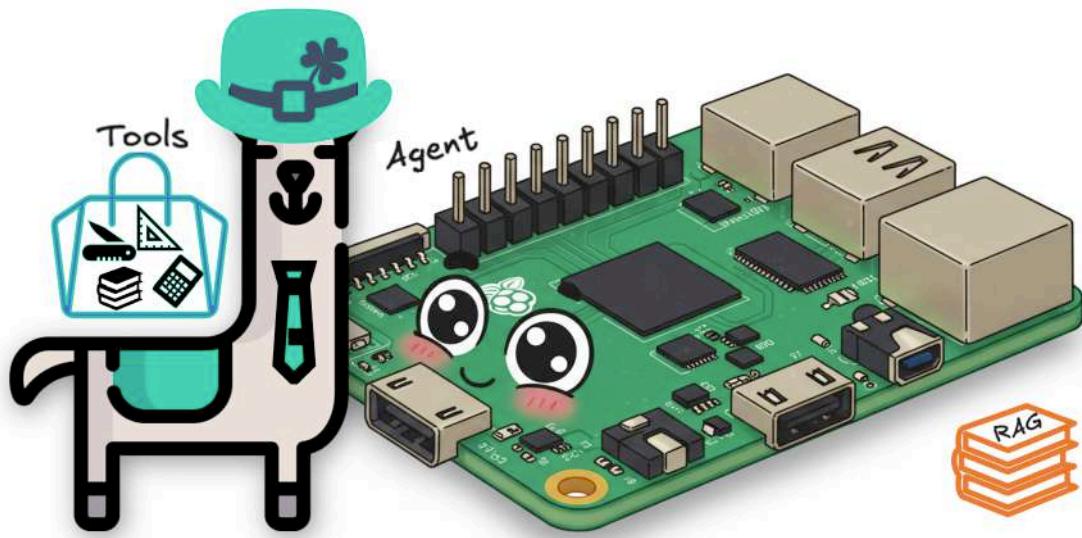


Figure 15: *Image from author, with a Raspberry Pi from ImageFX - prompt - Create a cartoon image with a single Raspberry Pi with a white background*

Building on the foundation established in previous chapters of “Edge AI Engineering,” this chapter explores the limitations of Small Language Models (SLMs) and advanced techniques to enhance their capabilities at the edge. While we’ve demonstrated the feasibility of running SLMs on devices like the Raspberry Pi, practical applications often require addressing inherent limitations in these models.

As we explore techniques like chain-of-thought prompting, agent architectures, function calling, response validation, and Retrieval-Augmented Generation (RAG), we’ll see how clever engineering and system design can mitigate SLMs’ limitations.

Understanding SLM Limitations

Small Language Models, while impressive in their ability to run on edge devices, face several key limitations:

1. Knowledge Constraints

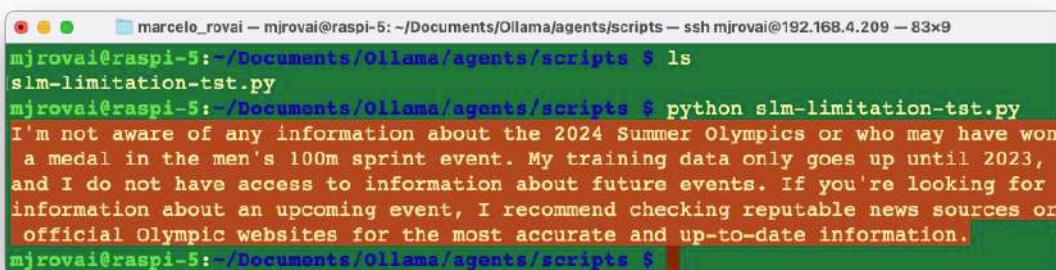
SLMs have limited knowledge based on their training data, often outdated and incomplete. Unlike their larger counterparts, they cannot store the vast information needed for comprehensive expertise across all domains.

Let's run the below example to verify this limitation.

```
import ollama

response = ollama.generate(
    model="llama3.2:1b",
    prompt="Who won the 2024 Summer Olympics men's 100m sprint final?"
)
print(response['response'])
```

The output of the previous code will likely show hallucination or admission of not knowing, as in the case below:



A screenshot of a terminal window titled "marcelo_rovai — mirovai@raspi-5: ~/Documents/Ollama/agents/scripts — ssh mirovai@192.168.4.209 — 83x9". The terminal shows the following command and its output:

```
mjrovai@raspi-5:~/Documents/Ollama/agents/scripts $ ls
slm-limitation-tst.py
mjrovai@raspi-5:~/Documents/Ollama/agents/scripts $ python slm-limitation-tst.py
I'm not aware of any information about the 2024 Summer Olympics or who may have won
a medal in the men's 100m sprint event. My training data only goes up until 2023,
and I do not have access to information about future events. If you're looking for
information about an upcoming event, I recommend checking reputable news sources or
official Olympic websites for the most accurate and up-to-date information.
mjrovai@raspi-5:~/Documents/Ollama/agents/scripts $
```

This constraint could be solved simply by having an **Agent** search the Internet for the answer or using **Retrieval-Augmented Generation (RAG)**, as we will see later.

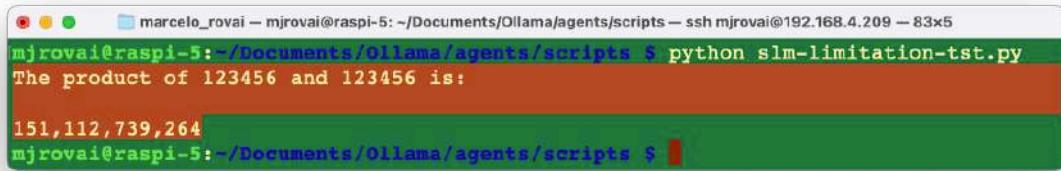
2. Reasoning Limitations

Complex reasoning tasks often exceed the capabilities of SLMs, which struggle with multi-step logical deductions, mathematical computations, and a nuanced understanding of context. **Agents** can be used to mitigate such limitations.

For example, let's reuse the previous code and ask to the SLM to multiply two numbers :

```
import ollama

response = ollama.generate(
    model="llama3.2:3b",
    prompt="Multiply 123456 by 123456"
)
print(response['response'])
```



```
marcelo_rovai - mjrovai@raspi-5: ~/Documents/Ollama/agents/scripts - ssh mjrovai@192.168.4.209 - 83x5
mjrovai@raspi-5:~/Documents/Ollama/agents/scripts$ python slm-limitation-tst.py
The product of 123456 and 123456 is:
151,112,739,264
mjrovai@raspi-5:~/Documents/Ollama/agents/scripts$
```

The response is wrong; once the multiplication result should be 15,241,383,936. This is expected once the language models are not suitable for mathematical computations. Still, we can use an “**agent**” to determine whether a user asks for multiplication or a general question. We will learn how to create an agent later.

3. Inconsistent Outputs

SLMs may produce inconsistent responses to the same query, making them unreliable for critical applications requiring deterministic outputs. Several enhancements, such as **Function Calling** and **Response Validation**, can improve reliability.

4. Domain Specialization

SLMs perform worse than specialized models in domain-specific tasks like visual recognition or time-series analysis. **Fine-tuning** can adapt models to specific domains or tasks, improving performance for targeted applications.

Techniques for Enhancing SLM at the Edge

Small Language Models (SLMs) offer remarkable capabilities for edge devices, but various techniques can significantly enhance their effectiveness. Here, we present a comprehensive framework for optimizing SLMs on resource-constrained devices like the Raspberry Pi, organized from fundamental to advanced approaches.

We will divide those technics into 3 segments:

- **Fundamentals:** Optimizing Prompting Strategies
 - Chain-of-Thought Prompting
 - Few-Shot Learning
 - Task Decomposition
- **Intermediate:** Building Intelligence Systems
 - Building Agents with SLMs
 - General Knowledge Router
 - Function Calling
 - Response Validation
- **Advanced:** Extending Knowledge and Specialization
 - Retrieval-Augmented Generation (RAG)
 - Fine-Tuning for Domain Specialization
- **Integration:** Combining Techniques for Optimal Performance

The true power of these techniques emerges when they're strategically combined:

1. **Agent Architecture with RAG:** Create agents that can access both tools and knowledge bases
2. **Validation-Enhanced RAG:** Apply response validation to ensure RAG outputs are accurate
3. **Fine-Tuned Routers:** Use specialized fine-tuned models to handle routing decisions
4. **Chain-of-Thought with Function Calling:** Combine reasoning traces with structured outputs

For example, a comprehensive weather monitoring system, as we introduced in the chapter “Experimenting with SLMs for IoT Control,” might use the following:

- RAG to access historical weather patterns and interpretation guides

- Function calling to structure sensor data analysis
- Response validation to verify recommendations
- Task decomposition to handle complex multi-part weather analysis

Optimizing Prompting Strategies

Chain-of-Thought Prompting

Chain-of-thought prompting encourages SLMs to break down complex problems into step-by-step reasoning, leading to more accurate results:

```
def solve_math_problem(problem):
    prompt = f"""
        Problem: {problem}
        Let's think about this step by step:
        1. First, I'll identify what we're looking for
        2. Then, I'll identify the relevant information
        3. Next, I'll set up the appropriate equations
        4. Finally, I'll solve the problem carefully
        Solving:
        """
    response = ollama.generate(model="llama3.2:3b", prompt=prompt)
    return response['response']
```

This technique significantly improves performance on reasoning tasks by emulating human problem-solving approaches.

Few-Shot Learning

Few-shot learning provides examples within the prompt, helping SLMs understand the expected response format and reasoning pattern:

```
def classify_sentiment(text):
    prompt = f"""
        Task: Classify the sentiment of the text as positive, negative, or neutral.
        Examples:
        Text: "I love this product, it works perfectly!"
        Sentiment: positive
        Text: "This is the worst experience I've ever had."
        Sentiment: negative
    """
    response = ollama.generate(model="llama3.2:3b", prompt=prompt)
    return response['response']
```

```

Text: "The package arrived on time."
Sentiment: neutral
Text: "{text}"
Sentiment:
"""
response = ollama.generate(model="llama3.2:1b", prompt=prompt)
return response['response'].strip()

```

This approach is particularly effective for classification tasks and standardized outputs.

Task Decomposition

For complex tasks, breaking them into smaller subtasks helps SLMs manage complexity:

```

def analyze_product_review(review):
    # Step 1: Extract main points
    points_prompt = f"Extract the main points from this product review: {review}"
    points_response = ollama.generate(model="llama3.2:1b", prompt=points_prompt)
    main_points = points_response['response']

    # Step 2: Determine sentiment
    sentiment_prompt = f"Determine the overall sentiment of this review: {review}"
    sentiment_response = ollama.generate(model="llama3.2:1b",
                                          prompt=sentiment_prompt)
    sentiment = sentiment_response['response']

    # Step 3: Identify improvement suggestions
    improvements_prompt = f"What suggestions for improvement can be found in \
this review? {review}"
    improvements_response = ollama.generate(model="llama3.2:1b",
                                              prompt=improvements_prompt)
    improvements = improvements_response['response']

    # Final synthesis
    final_prompt = f"""
Create a concise analysis of this product review based on:
Main points: {main_points}
Overall sentiment: {sentiment}
Improvement suggestions: {improvements}
"""

    final_response = ollama.generate(model="llama3.2:1b",

```

```
        prompt=final_prompt)
    return final_response['response']
```

This technique distributes cognitive load across multiple simpler prompts, enabling SLMs to handle tasks that might otherwise exceed their capabilities.

Building Agents with SLMs

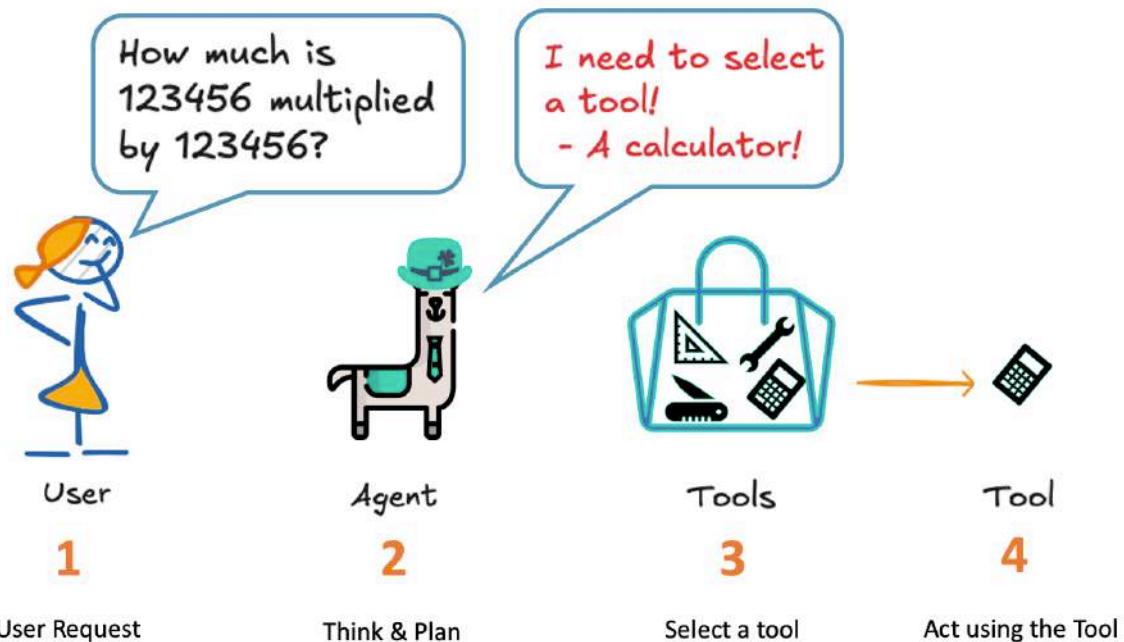
To address some of these limitations, we can develop agents that leverage SLMs as part of a more extensive system with additional capabilities.

What is an Agent? It is an **AI model capable of reasoning, planning, and interacting with its environment**. It can be called an *Agent* because it has an *agency* that can interact with the environment.

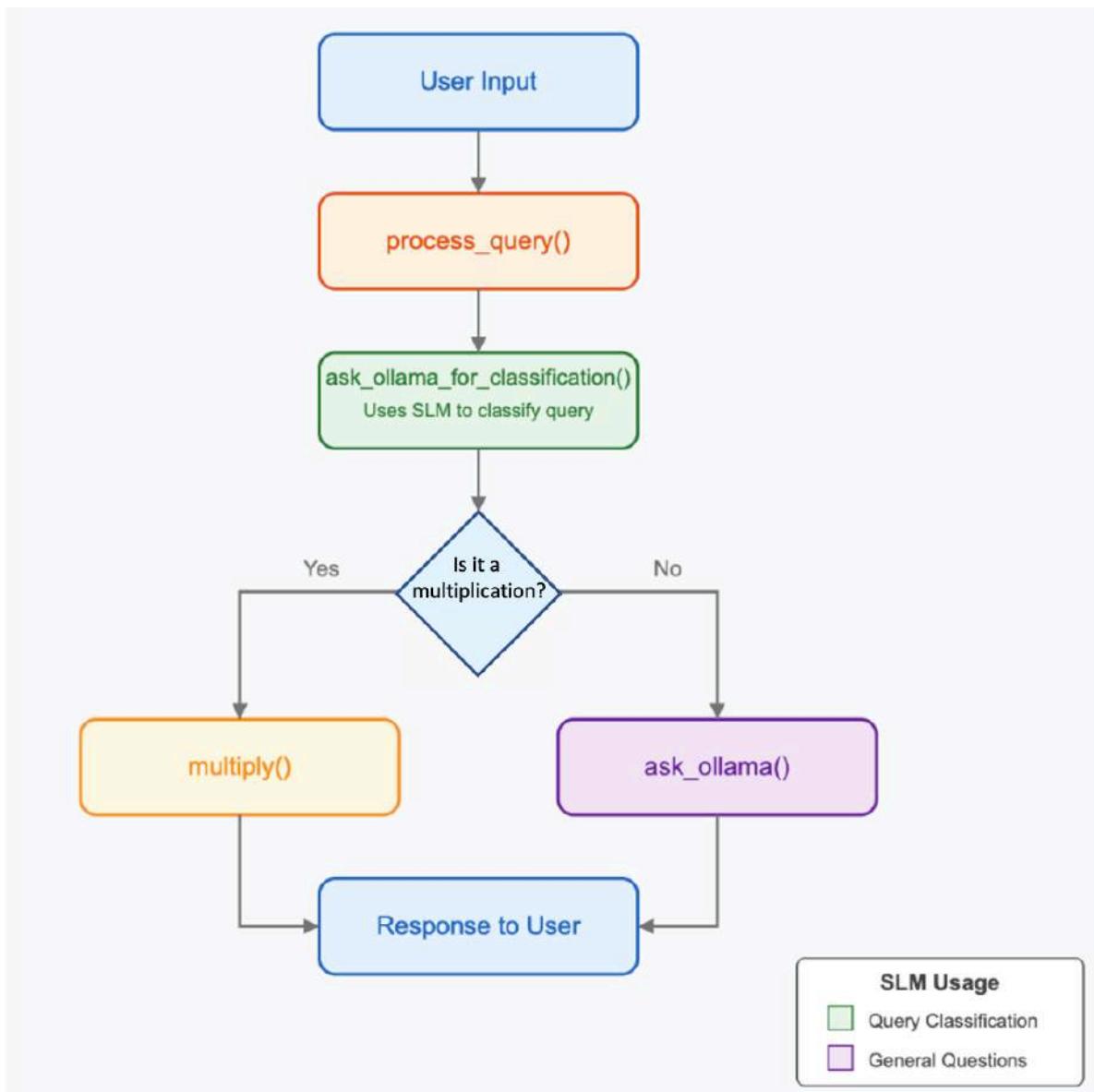
Let's think about the multiplication problem that we faced before. An Agent can be used for that.

An agent is a system that uses an AI Model as its core reasoning engine to:

- **Understand natural language:** (1) Interpret and respond to human instructions meaningfully.
- **Reason and plan:** (2) Analyze information, make decisions, and devise problem-solving strategies.
- **Interact with its environment:** (3 and 4) Gather information, take actions, and observe the results of those actions.



For example, if it is a multiplication, we can use a Python function as a “tool” to calculate it, as shown in the diagram:



Our code works through the following steps:

1. **User Input:** The user types a query like “What is 7 times 8?” or “What is the capital of France?”
2. **Process Query:** The `process_query()` function handles the input and decides what to do with it.
3. **Classification:** The `ask_ollama_for_classification()` function sends the user’s query to the SLM (using Ollama) with a prompt asking it to classify whether the query

is requesting multiplication or asking a general question.

4. **Decision:** Based on the SLM's classification:

- If it's a multiplication request, the SLM also extracts the numbers, and we use our `multiply()` function.
- If it's a general question, we send the original query to the SLM for a direct answer.

5. **Response:** The system returns either the multiplication result or the SLM's answer to the general question.

Here's a Python script that creates a simple agent (or router) between multiplication operations and general questions as described:

```
import requests
import json

# Configuration
OLLAMA_URL = "http://localhost:11434/api"
MODEL = "llama3.2:3b" # You can change this to any model you have installed
VERBOSE = True

def ask_ollama_for_classification(user_input):
    """
    Ask Ollama to classify whether the query is a multiplication request or a \
    general question.
    """
    classification_prompt = f"""
    Analyze the following query and determine if it's asking for multiplication \
    or if it's a general question.

    Query: "{user_input}"

    If it's asking for multiplication, respond with a JSON object in this format:
    {{
        "type": "multiplication",
        "numbers": [number1, number2]
    }}

    If it's a general question, respond with a JSON object in this format:
    {{
        "type": "general_question"
    }}
    """

    response = requests.post(f"{OLLAMA_URL}/classification", json={"query": classification_prompt})
    response.raise_for_status()
    return response.json()
```

```

Respond ONLY with the JSON object, nothing else.
"""

try:
    if VERBOSE:
        print(f"Sending classification request to Ollama")

    response = requests.post(
        f"{{OLLAMA_URL}}/generate",
        json={
            "model": MODEL,
            "prompt": classification_prompt,
            "stream": False
        }
    )

    if response.status_code == 200:
        response_text = response.json().get("response", "").strip()
        if VERBOSE:
            print(f"Classification response: {response_text}")

        # Try to parse the JSON response
        try:
            # Find JSON content if there's any surrounding text
            start_index = response_text.find('{')
            end_index = response_text.rfind('}') + 1
            if start_index >= 0 and end_index > start_index:
                json_str = response_text[start_index:end_index]
                return json.loads(json_str)
            return {"type": "general_question"}
        except json.JSONDecodeError:
            if VERBOSE:
                print(f"Failed to parse JSON: {response_text}")
            return {"type": "general_question"}
    else:
        if VERBOSE:
            print(f"Error: Received status code {response.status_code} \
from Ollama.")
        return {"type": "general_question"}

except Exception as e:
    if VERBOSE:

```

```

        print(f"Error connecting to Ollama: {str(e)}")
        return {"type": "general_question"}

def multiply(a, b):
    """
    Perform multiplication and return a formatted response.
    """
    result = a * b
    return f"The product of {a} and {b} is {result}."

def ask_ollama(query):
    """
    Send a query to Ollama for general question answering.
    """
    try:
        if VERBOSE:
            print(f"Sending query to Ollama")

        response = requests.post(
            f"{OLLAMA_URL}/generate",
            json={
                "model": MODEL,
                "prompt": query,
                "stream": False
            }
        )

        if response.status_code == 200:
            return response.json().get("response", "")
        else:
            return f"Error: Received status code {response.status_code} \
from Ollama."
    except Exception as e:
        return f"Error connecting to Ollama: {str(e)}"

def process_query(user_input):
    """
    Process the user input by first asking Ollama to classify it,
    then either performing multiplication or sending it back as a
    general question.

```

```

"""
# Let Ollama classify the query
classification = ask_ollama_for_classification(user_input)

if VERBOSE:
    print("Ollama classification:", classification)

if classification.get("type") == "multiplication":
    numbers = classification.get("numbers", [0, 0])
    if len(numbers) >= 2:
        return multiply(numbers[0], numbers[1])
    else:
        return "I understood you wanted multiplication, but couldn't \
extract the numbers properly."
else:
    return ask_ollama(user_input)

def main():
    """
    Main function to run the agent interactively.
    """
    print("Ollama Agent (Type 'exit' to quit)")
    print("-----")

    while True:
        user_input = input("\nYou: ")

        if user_input.lower() in ["exit", "quit", "bye"]:
            print("Goodbye!")
            break

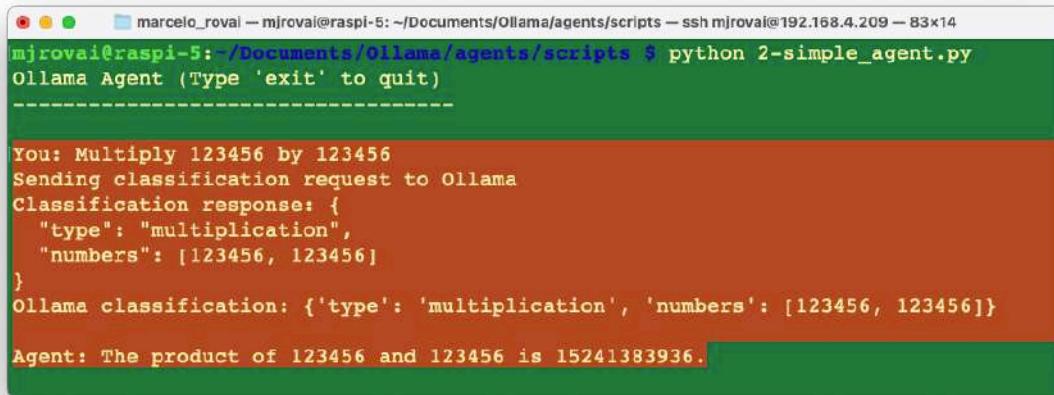
        response = process_query(user_input)
        print(f"\nAgent: {response}")

# Example usage
if __name__ == "__main__":
    # Set to True to see detailed logging
    VERBOSE = True
    main()

```

When we run the script, we can see that, first, the SLM chooses `multiplication`, passing the numbers entered by the user to the “tool,” which, in this case, is the `multiply()` function. As

a result, we got 15,241,383,936, which it is correct.

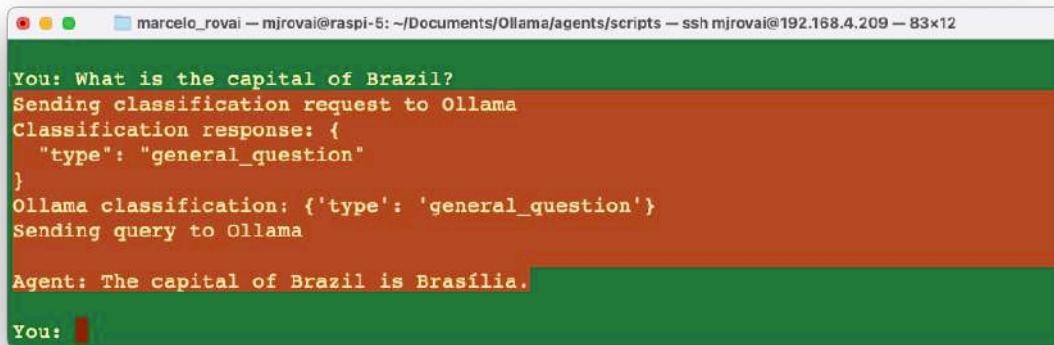


```
mjrovai@raspi-5:~/Documents/Ollama/agents/scripts$ python 2-simple_agent.py
Ollama Agent (Type 'exit' to quit)

You: Multiply 123456 by 123456
Sending classification request to Ollama
Classification response: {
    "type": "multiplication",
    "numbers": [123456, 123456]
}
Ollama classification: {'type': 'multiplication', 'numbers': [123456, 123456]}

Agent: The product of 123456 and 123456 is 15241383936.
```

Let's now enter with another question that has no relation with arithmetic, for example: **What is the capital of Brazil?** In this case, the SLM will decide that the query is a **general** question and pass it on to the SLM to answer it.



```
mjrovai@raspi-5:~/Documents/Ollama/agents/scripts$ python 2-simple_agent.py
Ollama Agent (Type 'exit' to quit)

You: What is the capital of Brazil?
Sending classification request to Ollama
Classification response: {
    "type": "general_question"
}
Ollama classification: {'type': 'general_question'}
Sending query to Ollama

Agent: The capital of Brazil is Brasília.

You:
```

This simple agent (or router) demonstrates the fundamental concept of using an SLM to make decisions about processing different types of user inputs. It shows both the power of SLMs for natural language understanding and their limitations in structured tasks.

Limitations and Considerations

This agent seems to resolve our problem, but it has several limitations that are common when working with SLMs:

1. **JSON Parsing Issues:** SLMs don't always perfectly format JSON responses as requested. The code includes error handling for this.
2. **Classification Reliability:** The SLM might not always correctly classify the query, especially with ambiguous questions.
3. **Number Extraction:** The SLM might extract numbers incorrectly or miss them entirely.
4. **Error Handling:** Robust error handling is essential when working with SLMs because their outputs can be unpredictable.
5. **Latency:** Significant latency is involved in making multiple calls to the SLM. For example, for the above simple agent, the latency was about 50s when using the `llama3.2:3B` on a `Raspberry Pi 5`.

Here, you can see the SLM latency (simple query) per device (in tokens/s):

Model	Raspi 5 (Cortex A-76)	PC (i7)	Mac (M1 Pro)
Gemma3:4b	3.8	8.7	39
Llama3.2:3b	5.5	12	63
Llama3.2:1b	7.5	19.5	111
Gemma3:1b	12	22.45	91

In my simple tests, the 1B models struggled to classify the tasks correctly. The the 3B and 4B models worked fine

Improvements

To create a more robust agent, we can, for example:

1. **Expand Capabilities:** Add support for more operations (addition, subtraction, division).
2. **Better Error Handling:** Improve fallback mechanisms when the SLM fails to extract numbers or classify correctly.
3. **Model Preloading:** Initialize the model at startup to reduce latency.
4. **Adding Regex Fallbacks:** Use regular expressions as a fallback to extract numbers when the SLM fails.
5. **Context Preservation:** Maintain conversation context for multi-turn interactions.

A more robust script can be used with the above improvements. The diagram shows how it would work:

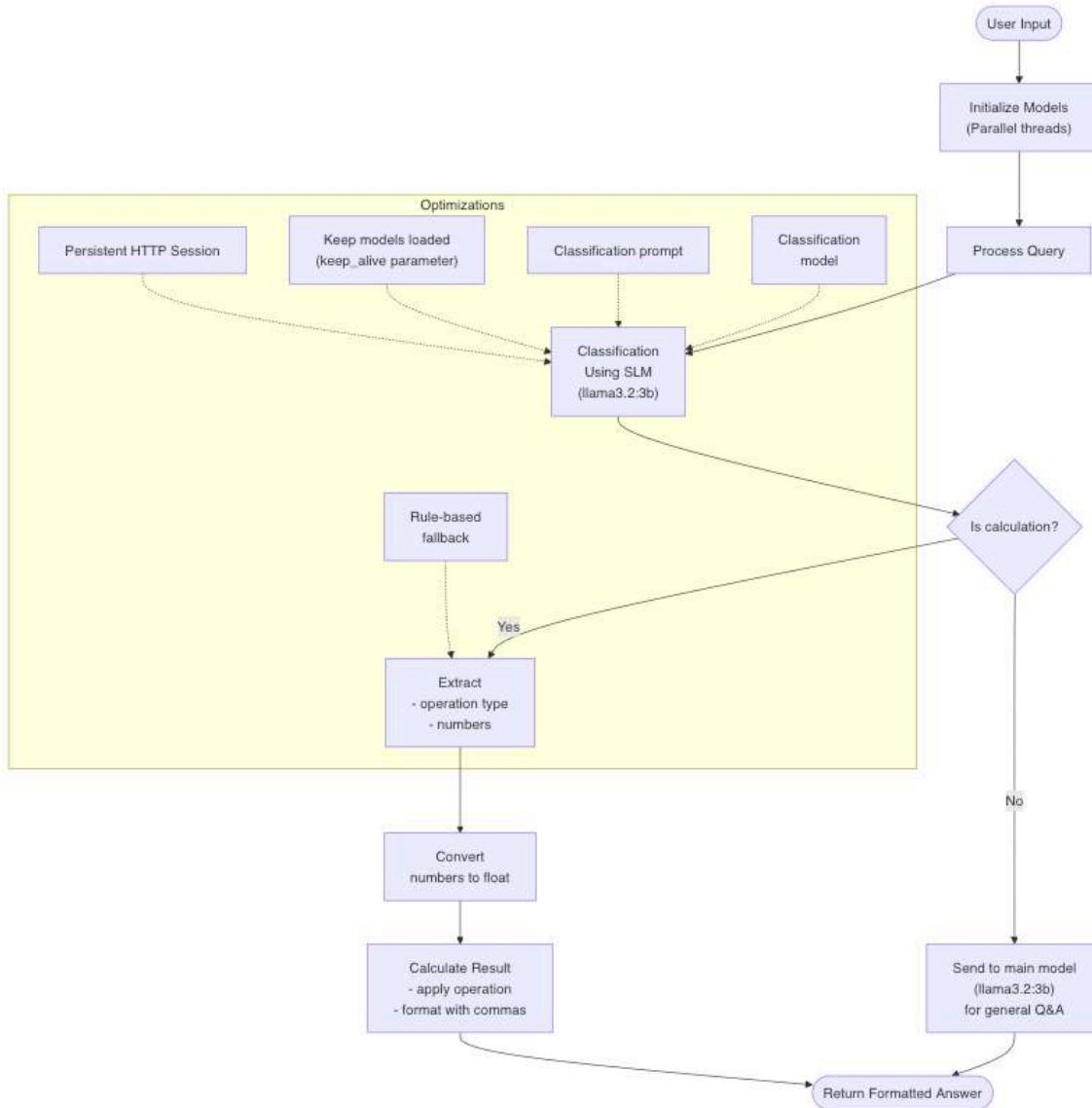


Figure 16: image-20250322113135882

The diagram illustrates the key components of the system:

1. Initialization:

- The system starts by initializing both models in parallel threads
- This prevents cold starts and reduces latency

2. Query Processing Flow:

- User input is first sent to a classification step
- A model (llama3.2:3B) determines if it's a calculation or a general question (we can choose a different model here).
- If it's a calculation:
 - The system extracts the operation type and numbers
 - Numbers are converted from strings to floats
 - The appropriate calculation is performed
 - Results are formatted with comma separators (e.g., 1,234,567.89)
- If it's a general question:
 - The query is sent to the main model (llama3.2:3b) for answering (we can choose a different model here)

3. Optimizations (highlighted in the subgraph):

- Persistent HTTP session for connection reuse
- Keep-alive parameter to prevent model unloading
- Simplified classification prompt for faster processing
- Using a smaller model for the classification task
- Rule-based fallback logic if the model classification fails

The main performance improvements come from:

1. Keeping models loaded in memory
2. Using connection pooling
3. Simplifying the classification task
4. Using a smaller model for classification
5. Initializing models in parallel

This approach maintains the intelligent classification capability while significantly reducing execution time compared to the original implementation.

Running the script `3-ollama-calculator-agent.py`, we get correct results with reduced latency of about 60%.

```
marcelo_rovai@raspi-5:~/Documents/Ollama/agents/scripts$ python ollama-calculator-agent.py
Optimized Ollama Calculator Agent
Main model: llama3.2:3b, Classification model: llama3.2:3b
Type 'exit' to quit, 'model <name>' to change main model, or 'classmodel <name>' to change
classification model
-----
Initializing models: llama3.2:3b and llama3.2:3b
Models initialized and ready

You: Calculate 12345.45 divide per 123.24
Sending classification request using llama3.2:3b
Classification: {'type': 'calculation', 'operation': 'divide', 'numbers': [12345.45, 123.24]}
}

Agent: The result of dividing 12,345.45 by 123.24 is 100.1741.

Time elapsed: 15.06 seconds
-----
You: 
```

```
marcelo_rovai@raspi-5:~/Documents/Ollama/agents/scripts$ ssh mirovai@192.168.4.209 -t
You: what are the capital of Malawi?
Sending classification request using llama3.2:3b
Classification: {'type': 'general_question', 'capital': 'Lilongwe'}
Sending query to Ollama using llama3.2:3b

Agent: The capital of Malawi is Lilongwe.

Time elapsed: 15.51 seconds
-----
You: 
```

General Knowledge Router

Remember when we asked our SLM: Who won the 2024 Summer Olympics men's 100m sprint final? We could not receive an answer because the modes were trained with information previously in late 2023.

To solve this issue, let's build a more advanced agent to classify whether it should use its knowledge to answer a question or fetch updated information from the Internet. This addresses a key limitation of Small Language Models: their knowledge cutoff date.

The general architecture of our agent will be similar to the calculator, but now, we will use a web search API as a tool.

This agent addresses a critical limitation of SLMs - their knowledge cutoff date - by determining when to use the model's built-in knowledge versus when to search for up-to-date information from the web.

How it works:

Uses SLM for Classification: Relies entirely on the SLM to determine whether a query needs web search or can be answered from the model's knowledge.

Provides Date Context: This section supplies the current date to help the SLM make informed decisions about whether information is outdated.

Integrates Tavily Search: Uses [Tavily's powerful search API](#) to find relevant information for queries that need external data.

Handles Timeouts: Includes fallback mechanisms when the model takes too long to respond.

Maintains Source Attribution: Clearly indicates to the user whether the answer comes from the model's knowledge or web search.

Let's run the script: `4-ollama-search-agent.py`

But first, we should install the required libraries:

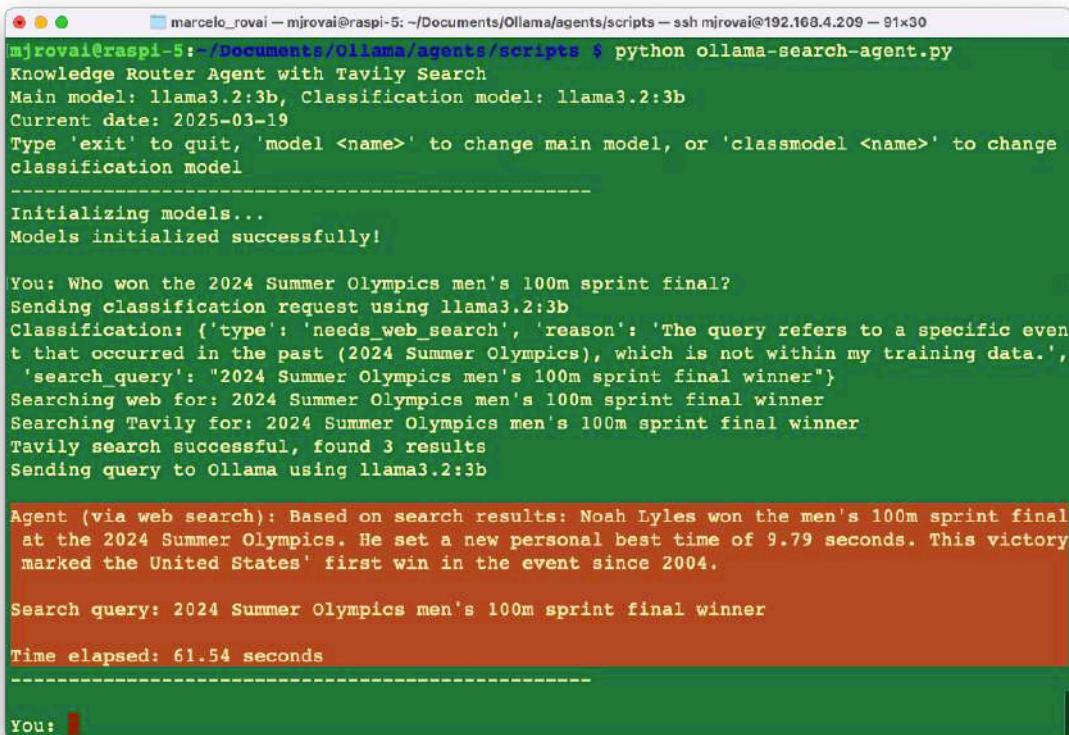
```
pip install requests
pip install tavily-python
```

Replace "tvly-YOUR_API_KEY" with your actual Tavily API key.

Why Tavily is Superior for This Use Case

1. **Built for RAG:** Tavily is specifically designed for retrieval-augmented generation, making it perfect for our knowledge router.
2. **High-Quality Results:** It prioritizes reputable sources and provides context-relevant results.
3. **Built-in Summarization:** The API can provide an AI-generated summary of search results, giving an additional layer of processing before your SLM.
4. **Simple Integration:** Clean API with straightforward responses that are easy to parse.
5. **Generous Free Tier:** 1,000 free searches is plenty for testing and personal use.

Runing the script and entering with the same questions that could not be answered before, we now have: Noah Lyles won the men's 100m sprint final at the 2024 Summer Olympics. He set a new personal best time of 9.79 seconds. This victory marked the United States' first win in the event since 2004.



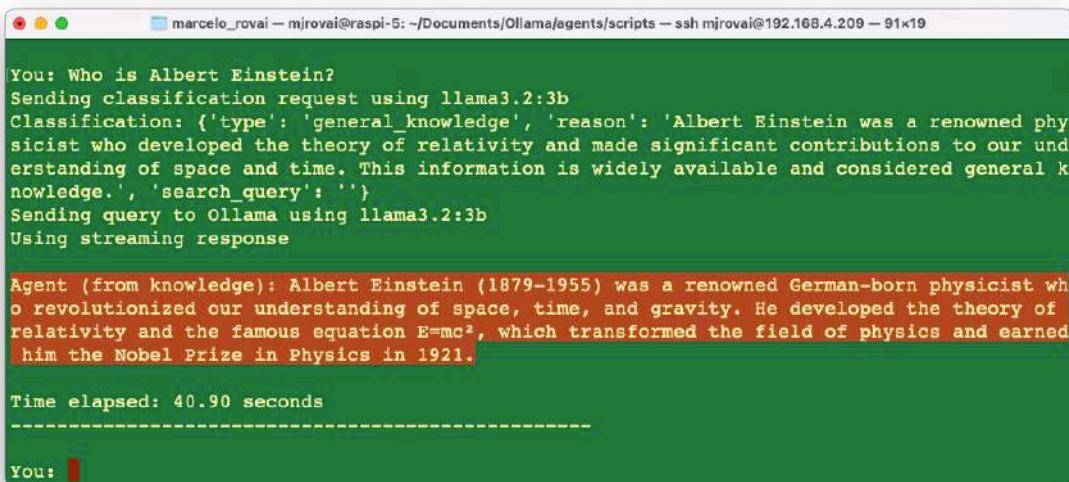
```
mjrovai@raspi-5:~/Documents/Ollama/agents/scripts$ python ollama-search-agent.py
Knowledge Router Agent with Tavily Search
Main model: llama3.2:3b, Classification model: llama3.2:3b
Current date: 2025-03-19
Type 'exit' to quit, 'model <name>' to change main model, or 'classmodel <name>' to change classification model
-----
Initializing models...
Models initialized successfully!

You: Who won the 2024 Summer Olympics men's 100m sprint final?
Sending classification request using llama3.2:3b
Classification: {'type': 'needs_web_search', 'reason': 'The query refers to a specific event that occurred in the past (2024 Summer Olympics), which is not within my training data.', 'search_query': "2024 Summer Olympics men's 100m sprint final winner"}
Searching web for: 2024 Summer Olympics men's 100m sprint final winner
Searching Tavily for: 2024 Summer Olympics men's 100m sprint final winner
Tavily search successful, found 3 results
Sending query to Ollama using llama3.2:3b

Agent (via web search): Based on search results: Noah Lyles won the men's 100m sprint final at the 2024 Summer Olympics. He set a new personal best time of 9.79 seconds. This victory marked the United States' first win in the event since 2004.

Search query: 2024 Summer Olympics men's 100m sprint final winner
Time elapsed: 61.54 seconds
-----
You:
```

When the user enters a common-knowledge question, the agent will send it directly to the SLM. For example, if the user asks, "Who is Albert Einstein?", we get:



You: Who is Albert Einstein?
Sending classification request using llama3.2:3b
Classification: {'type': 'general_knowledge', 'reason': 'Albert Einstein was a renowned physicist who developed the theory of relativity and made significant contributions to our understanding of space and time. This information is widely available and considered general knowledge.', 'search_query': ''}
Sending query to Ollama using llama3.2:3b
Using streaming response

Agent (from knowledge): Albert Einstein (1879–1955) was a renowned German-born physicist who revolutionized our understanding of space, time, and gravity. He developed the theory of relativity and the famous equation E=mc², which transformed the field of physics and earned him the Nobel Prize in Physics in 1921.

Time elapsed: 40.90 seconds

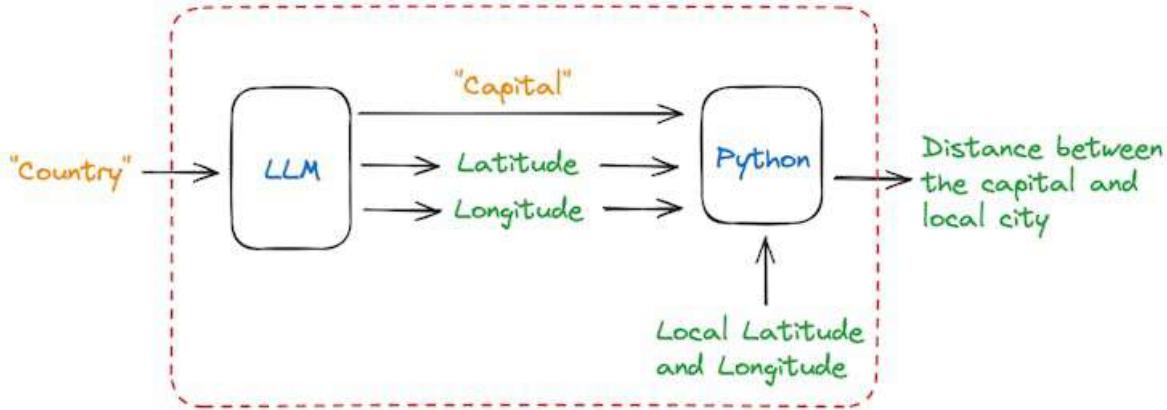
You:

Improving Agent Reliability

There are several ways to enhance an agent's reliability. One is to implement effective, approved, structured function calling, which makes agents' responses more consistent and predictable.

1. Function Calling with Pydantic

In the SLM chapter, we explored function calling when we created 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.



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, the app used a simple Python library ([haversine](#)) to calculate the distance between those 2 points.

The critical library used was [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* helps ensure that the model's response will always be consistent.

Function calling can improve an agent's reliability by ensuring structured outputs and clear tool selection logic. Here's a generic template about how we can implement it :

```

import time
from haversine import haversine
from pydantic import BaseModel, Field
from ollama import chat

MODEL = 'llama3.2:3B'    # The name of the model to be used
mylat = -33.33            # Latitude of Santiago de Chile
mylon = -70.51            # Longitude of Santiago de Chile

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")

def calc_dist(country, model=MODEL):

    start_time = time.perf_counter()  # Start timing

```

```

# Ask Ollama for structured data
response = chat(
    model=model,
    messages=[{
        "role": "user",
        "content": f"Return the capital city of {country}, \
with its decimal latitude and longitude."
    }],
    format=CityCoord.model_json_schema(), # Structured JSON format
    options={"temperature": 0}
)

resp = CityCoord.model_validate_json(response.message.content)

distance = haversine((mylat, mylon), (resp.lat, resp.lon), unit='km')

end_time = time.perf_counter() # End timing
elapsed_time = end_time - start_time # Calculate elapsed time

print(f"\nSantiago de Chile is about {int(round(distance, -1))}:,\") \
kilometers away from {resp.city}.")
print(f"[INFO] ==> {MODEL}: {elapsed_time:.1f} seconds")

# Test
calc_dist('france')
calc_dist('colombia')
calc_dist('united states')

```

Santiago de Chile is about 11,630 kilometers away from Paris.
[INFO] ==> llama3.2:1B: 7.1 seconds

Santiago de Chile is about 4,240 kilometers away from Bogotá.
[INFO] ==> llama3.2:1B: 7.5 seconds

Santiago de Chile is about 8,060 kilometers away from Washington D.C..
[INFO] ==> llama3.2:1B: 8.3 seconds

2. Response Validation

Response validation is crucial to developing and deploying AI agents powered by language models. Here are key points regarding LLM validation for agents:

- Types of Validation

- Response Relevancy: Determines if the LLM output addresses the input informatively and concisely.
- Prompt Alignment: Check if the LLM output follows instructions from the prompt template.
- Correctness: Assesses factual accuracy based on ground truth.
- Hallucination Detection: Identifies fake or made-up information in LLM outputs.

Adding validation prevents incorrect or harmful responses, and here, we can test it with a simple script:

```
import ollama
import json

def validate_response(query, response):
    """Validate that the response is appropriate for the query"""
    validation_prompt = f"""
        User query: {query}
        Generated response: {response}

        Evaluate if this response:
        1. Directly addresses the user's query
        2. Is factually accurate to the best of your knowledge
        3. Is helpful and complete

        Respond in the following JSON format:
        {{
            "valid": true/false,
            "reason": "Explanation if invalid",
            "score": 0-10
        }}
    """

    try:
        validation = ollama.generate(
            model="llama3.2:3b",
            prompt=validation_prompt
        )
```

```

        result = json.loads(validation['response'])
        return result
    except Exception as e:
        print(f"Error during validation: {e}")
        return {"valid": False, "reason": "Validation error", "score": 0}

# Test
query = "What is the Raspberry Pi 5?"
response = "It is a pie created with raspberry and cooked in an oven"
validation = validate_response(query, response)
print(validation)

```

```

marcelo_rovai — mjrovai@raspi-5: ~/Documents/Ollama/agents/scripts — ssh mjrovai@192.168.4.209...
mjrovai@raspi-5:~/Documents/Ollama/agents/scripts$ python 5-validate_response.py
{'valid': False, 'reason': "The response is not relevant to the user's query about the Raspberry Pi 5. It appears to be a humorous or nonsensical answer.", 'score': 2}
mjrovai@raspi-5:~/Documents/Ollama/agents/scripts $

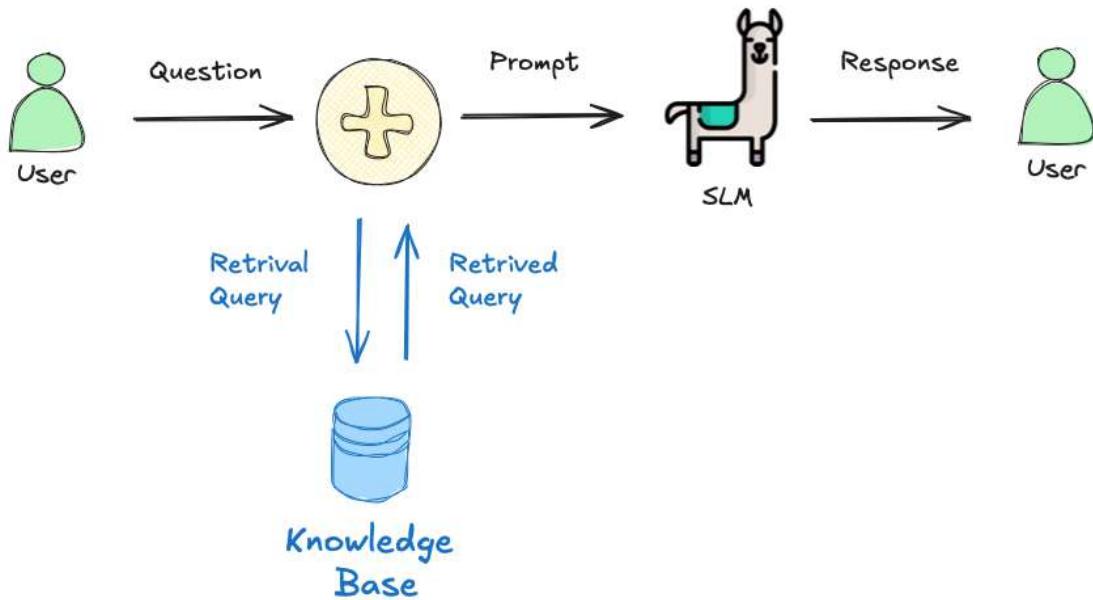
```

Retrieval-Augmented Generation (RAG)

RAG systems enhance Small Language Models (SLMs) by providing relevant information from external sources before generation. This is particularly valuable for edge devices with limited model sizes, as it allows them to access knowledge beyond their training data without increasing the model size.

Understanding RAG

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.



The RAG process consists of these key steps:

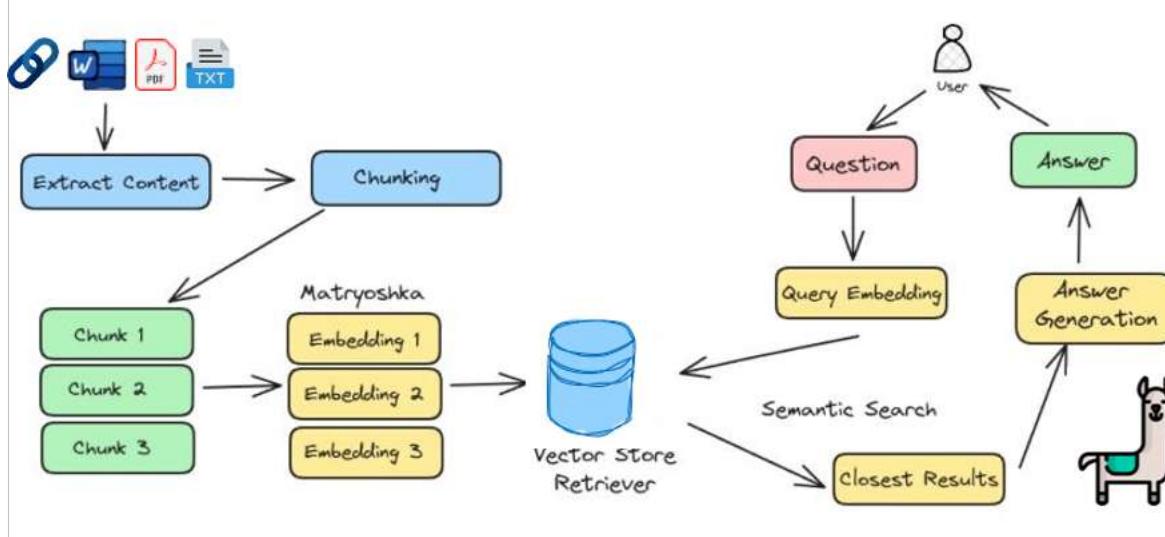
1. **Query Processing:** When a user asks a question, the system converts it into an embedding (a numerical representation).
2. **Document Retrieval:** The system searches a knowledge base for documents with similar embeddings.
3. **Context Enhancement:** Relevant documents are retrieved and combined with the original query.
4. **Generation:** The SLM generates a response using both the query and the retrieved context.

Implementing a Naive RAG System

We will develop two crucial components (scripts) of an RAG system:

1. Creating the Vector Database ([10-Create-Persistent-Vector-Database.py](#)). This script builds a knowledge base by:
 - Loading documents from PDFs and URLs
 - Splitting them into manageable chunks
 - Creating embeddings for each chunk
 - Storing these embeddings in a vector database (Chroma)
2. Querying the Database ([20-Query-the-Persistent-RAG-Database.py](#)). This script:

- Loads the saved vector database
- Accepts user queries
- Retrieves relevant documents based on query similarity
- Combines documents with the query in a prompt
- Generates a response using the SLM



Instalation

Once inside an environment, install the required libraries:

```

pip install -U 'langchain-chroma'
pip install -U langchain
pip install -U langchain-community
pip install -U langchain-ollama
pip install -U langchain-text-splitter
pip install -U langchain-community pypdf
pip install tiktoken
pip install -U langsmith
  
```

Let's examine how these components work together to implement a RAG system on edge devices.

Key Components of the Naive RAG System

1. Document Processing

```

def create_vectorstore():
    # Load documents from PDFs and URLs
    docs_list = []
    # [Document loading code]

    # Split documents into chunks
    text_splitter = RecursiveCharacterTextSplitter.from_tiktoken_encoder(
        chunk_size=300, chunk_overlap=30
    )
    doc_splits = text_splitter.split_documents(docs_list)

    # Create embeddings and store in vector database
    embedding_function = OllamaEmbeddings(model="nomic-embed-text")
    vectorstore = Chroma.from_documents(
        documents=doc_splits,
        collection_name="rag-edgeai-eng-chroma",
        embedding=embedding_function,
        persist_directory=PERSIST_DIRECTORY
    )

    # Persist to disk
    vectorstore.persist()

```

This function processes our documents (chunk size of 300 with an overlap of 30), creating a searchable knowledge base. Notice we're using `OllamaEmbeddings` with the `nomic-embed-text` model, which can run efficiently on edge devices like the Raspberry Pi.

1. Query Processing and Retrieval

```

def answer_question(question, retriever):
    """Generate an answer using the RAG system"""
    start_time = time.time()

    print(f"Question: {question}")
    print("Retrieving documents...")
    docs = retriever.invoke(question)
    docs_content = "\n\n".join(doc.page_content for doc in docs)
    print(f"Retrieved {len(docs)} document chunks")

    print("Generating answer...")

    # Using new LangSmith client and pull_prompt

```

```

client = Client()
rag_prompt = client.pull_prompt("rlm/rag-prompt")

# Compose the RAG chain
if isinstance(rag_prompt, str):
    rag_prompt = ChatPromptTemplate.from_template(rag_prompt)

rag_chain = rag_prompt | llm | StrOutputParser()
answer = rag_chain.invoke({"context": docs_content, "question": question})

end_time = time.time()
latency = end_time - start_time
print(f"Response latency: {latency:.2f} seconds using model: {local_llm}")

return answer

```

This function retrieves relevant documents based on the query and combines them with a specialized RAG prompt to generate a response. The RAG prompt is particularly important as it tells the model how to use the context documents to answer the question.

1. SLM Integration

```

# Initialize the LLM
local_llm = "llama3.2:3b"
llm = ChatOllama(model=local_llm, temperature=0)

```

We're using Ollama to run the SLM locally on our edge device, in this case using the 3B parameter version of Llama 3.2.

Advantages of RAG for Edge AI

Using RAG on edge devices offers several significant advantages:

- 1. Knowledge Extension:** RAG allows small models to access knowledge beyond their training data, effectively extending their capabilities without increasing model size.
- 2. Reduced Hallucination:** By providing factual context, RAG significantly reduces the likelihood of SLMs generating incorrect information.
- 3. Up-to-date Information:** Unlike the fixed knowledge in a model's weights, RAG knowledge bases can be updated regularly with new information.
- 4. Domain Specialization:** RAG can make general SLMs perform like domain specialists by providing domain-specific knowledge bases.

5. **Resource Efficiency:** RAG allows smaller models (which require less memory and computation) to achieve performance comparable to much larger models.

Optimizing RAG for Edge Devices

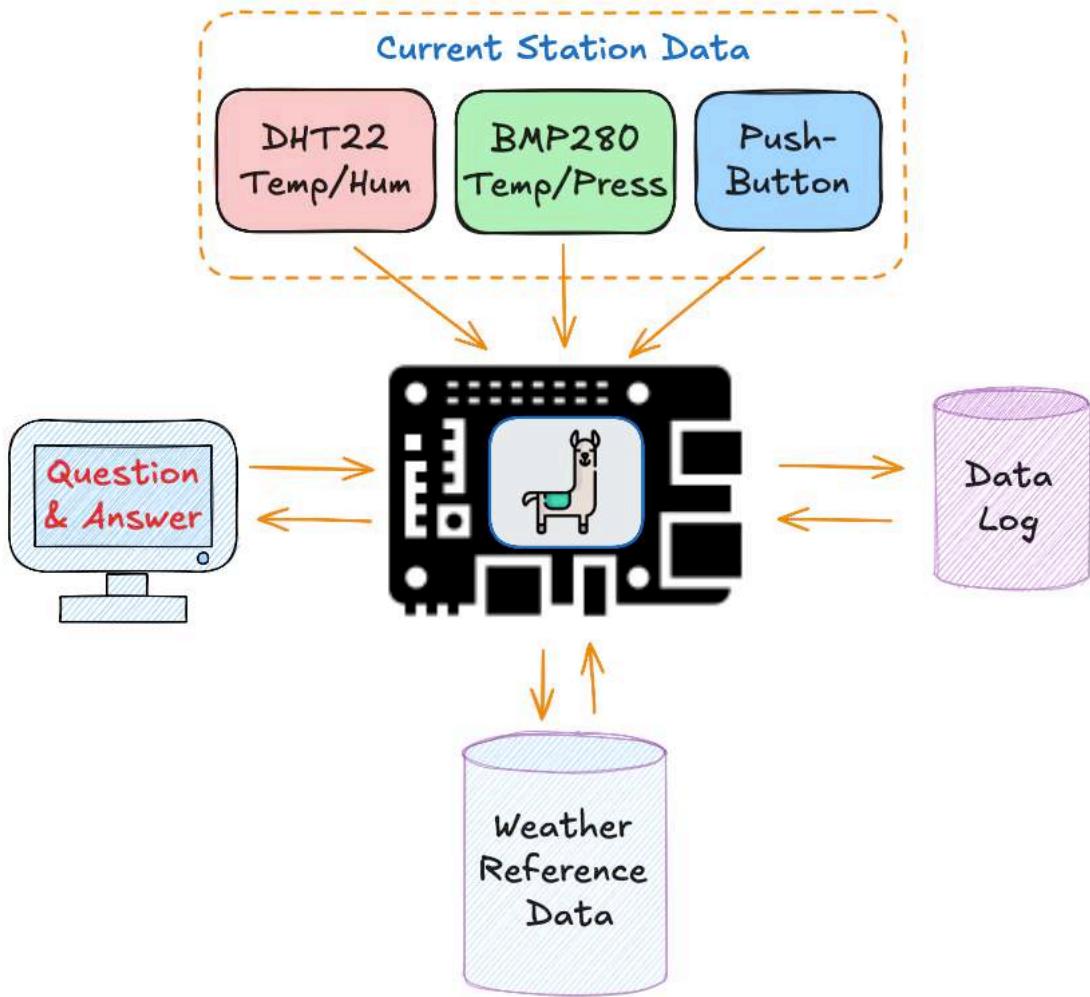
When implementing RAG on resource-constrained edge devices like the Raspberry Pi, consider these optimizations:

1. **Chunk Size:** Smaller chunks (300-500 tokens) reduce memory usage during retrieval and generation.
2. **Retrieval Limits:** Limit the number of retrieved documents ($k=3$ to 5) to reduce context size.
3. **Embedding Model Selection:** Choose lightweight embedding models like `nomic-embed-text` (137M parameters) or `all-minilm` (23M parameters).
4. **Persistent Storage:** As shown in our examples, using persistent storage prevents re-computing embeddings every time that the RAG system is initiated.
5. **Query Optimization:** Implement query preprocessing to improve retrieval accuracy while reducing computational load.

```
def optimize_query(query):
    """Optimize the query for better retrieval results"""
    # Remove filler words, focus on key terms
    stop_words = {"and", "or", "the", "a", "an", "in", "on", "at", "to",
                  "for", "with"}
    terms = [term for term in query.lower().split() if term not in stop_words]
    return " ".join(terms)
```

Application: Enhanced Weather Station with RAG

Building on our advanced weather station (see the chapter “Experimenting with SLMs for IoT Control”), we can, for example, integrate RAG to provide more contextual responses about weather conditions and historical patterns:



```

def weather_station_with_rag(retriever, model="llama3.2:3b"):
    # Get current sensor readings
    temp_dht, humidity, temp_bmp, pressure, button_state = collect_data()

    # Formulate a query for the RAG system based on current readings
    query = f"Analysis of temperature {temp_dht}°C, humidity {humidity}%, \
and pressure {pressure}hPa"

    # Retrieve relevant context
    docs = retriever.invoke(query)
    context = "\n\n".join(doc.page_content for doc in docs)

```

```

# Create a prompt that combines current readings with retrieved context
prompt = f"""
Current Weather Station Data:
- Temperature (DHT22): {temp_dht:.1f}°C
- Humidity: {humidity:.1f}%
- Pressure: {pressure:.2f}hPa

Reference Information:
{context}

Based on current readings and the reference information, provide:
1. An analysis of current weather conditions
2. What these conditions typically indicate
3. Recommendations for any actions needed
"""

# Generate response using SLM
llm = ChatOllama(model=model, temperature=0)
response = llm.invoke(prompt)

return response.content

```

This function enhances our weather station by providing context-aware responses incorporating current sensor readings and relevant information from our knowledge base. This is only an example. To use it, we should have “Weather Reference Data,” which we do not currently have. Instead, let’s create a general RAG system specializing in Edge AI Engineering.

Using the RAG System for Edge AI Engineering

For our RAG system, we will create a database with all chapters already written for the [EdgeAI Engineering book](#) (chapters as URLs) and a PDF [Wevolver 2025 Edge AI Technology Report](#).

```

# PDF documents to include
pdf_paths = ["./data/2025_Edge_AI_Technology_Report.pdf"]

# Define URLs for document sources
urls = [
    "https://mjrovai.github.io/EdgeML_Made_Easy_ebook/raspi\
/object_detection/object_detection.html",
    "https://mjrovai.github.io/EdgeML_Made_Easy_ebook/raspi/image_classification\

```

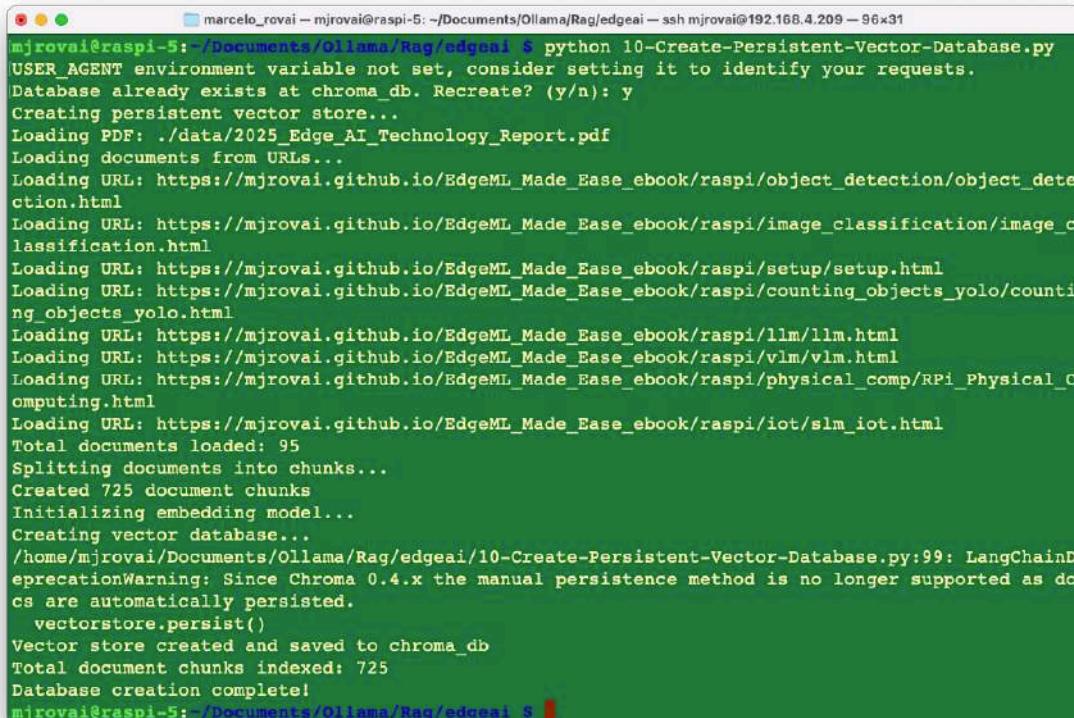
```

    "/image_classification.html",
    "https://mjrovai.github.io/EdgeML_Made_Ease_ebook/raspi/setup/setup.html",
    "https://mjrovai.github.io/EdgeML_Made_Ease_ebook/raspi/counting_objects_yolo\
/counting_objects_yolo.html",
    "https://mjrovai.github.io/EdgeML_Made_Ease_ebook/raspi/l1m/l1m.html",
    "https://mjrovai.github.io/EdgeML_Made_Ease_ebook/raspi/vlm/vlm.html",
    "https://mjrovai.github.io/EdgeML_Made_Ease_ebook/raspi/physical_comp\
/RPi_Physical_Computing.html",
    "https://mjrovai.github.io/EdgeML_Made_Ease_ebook/raspi/iot/slm_iot.html",
]

```

Using the RAG system is straightforward. First, ensure you've created the vector database:

```
# Run once to create the database
python 10-Create-Persistent-Vector-Database.py
```



```

marcelo_rovai@raspi-5:~/Documents/Ollama/Rag/edgeai$ python 10-Create-Persistent-Vector-Database.py
USER_AGENT environment variable not set, consider setting it to identify your requests.
Database already exists at chroma_db. Recreate? (y/n): y
Creating persistent vector store...
Loading PDF: ./data/2025_Edge_AI_Technology_Report.pdf
Loading documents from URLs...
Loading URL: https://mjrovai.github.io/EdgeML_Made_Ease_ebook/raspi/object_detection/object_dete
ction.html
Loading URL: https://mjrovai.github.io/EdgeML_Made_Ease_ebook/raspi/image_classification/image_c
lassification.html
Loading URL: https://mjrovai.github.io/EdgeML_Made_Ease_ebook/raspi/setup/setup.html
Loading URL: https://mjrovai.github.io/EdgeML_Made_Ease_ebook/raspi/counting_objects_yolo/counti
ng_objects_yolo.html
Loading URL: https://mjrovai.github.io/EdgeML_Made_Ease_ebook/raspi/l1m/l1m.html
Loading URL: https://mjrovai.github.io/EdgeML_Made_Ease_ebook/raspi/vlm/vlm.html
Loading URL: https://mjrovai.github.io/EdgeML_Made_Ease_ebook/raspi/physical_comp/RPi_Physical_C
omputing.html
Loading URL: https://mjrovai.github.io/EdgeML_Made_Ease_ebook/raspi/iot/slm_iot.html
Total documents loaded: 95
Splitting documents into chunks...
Created 725 document chunks
Initializing embedding model...
Creating vector database...
/home/mjrovai/Documents/Ollama/Rag/edgeai/10-Create-Persistent-Vector-Database.py:99: LangChainD
eprecationWarning: Since Chroma 0.4.x the manual persistence method is no longer supported as do
cs are automatically persisted.
    vectorstore.persist()
Vector store created and saved to chroma_db
Total document chunks indexed: 725
Database creation complete!
mjrovai@raspi-5:~/Documents/Ollama/Rag/edgeai$ 
```

Then, interact with the system through queries:

```
# Start the interactive query interface
python 20-Query-the-Persistent-RAG-Database.py
```

Example interactions:

```
Your question: What is edge AI?
```

```
Generating answer...
```

```
Question: what is EdgeAI?
```

```
Retrieving documents...
```

```
Retrieved 4 document chunks
```

```
Generating answer...
```

```
Response latency: 165.72 seconds using model: llama3.2:3b
```

ANSWER:

```
=====
EdgeAI refers to the application of artificial intelligence (AI) at the edge
of a network, typically in real-time applications such as IoT sensors, industrial
robots, and smart cameras. The Edge AI ecosystem includes edge devices, edge
servers, and cloud platforms that work together to enable low-latency AI
inferencing and processing of data on-site without relying on continuous cloud connectivit
in AI by leveraging energy-efficient, affordable, and scalable solutions for machine
learning and advanced edge computing.
```

```
=====
```

The screenshot shows a terminal window with the following text:

```

marcelo_rovai@raspi-5: ~/Documents/Ollama/Rag/edgeai - ssh mjr0vai@192.168.4.209 - 96x22

Your question: How to setup a Raspberry Pi?

Generating answer...

Question: How to setup a Raspberry Pi?
Retrieving documents...
Retrieved 4 document chunks
Generating answer...
Response latency: 137.56 seconds using model: llama3.2:3b

ANSWER:
=====
To set up a Raspberry Pi, download and install the Raspberry Pi Imager on your computer, select the operating system (e.g., Raspbian OS 32-bit or 64-bit), configure settings such as hostname, username, password, and SSH enablement, and write the image to an SD card. Insert the SD card in to the Raspberry Pi, connect power, and wait for the initial boot process to complete. You can then access the Raspberry Pi remotely using SSH or start a Jupyter Notebook server to control GPIOs.
=====

Your question: 

```

Those responses demonstrate how RAG enhances the SLM's response with specific information from our knowledge base about Edge AI applications on Raspberry Pi. One issue that should be addressed is the latency.

To reduce latency, we can use for embedding, the `all-minilm` model which is much smaller (23M parameters vs. 137M for `nomic-embed-text`) and creates 384-dimensional embeddings instead of 768, significantly reducing computation time.

Also, smaller chunks can be helpful but have some disadvantages. For example, let's say that we can use a small chunk size (100 tokens with 50 overlap). Here are some considerations:

Advantages

- Memory Efficiency:** Smaller chunks require less memory during retrieval and processing, which is beneficial for resource-constrained devices like the Raspberry Pi.
- More Granular Retrieval:** Smaller chunks can potentially provide more precise matches to specific questions, especially for targeted queries about very specific details.
- Reduced Context Window Usage:** SLMs have limited context windows; smaller chunks allow you to include more distinct pieces of information while staying within these limits.

Disadvantages

1. **Loss of Context:** 100 tokens is approximately 75-80 words, which is often insufficient to capture complete concepts or explanations. Many paragraphs and technical descriptions require more space to convey their full meaning.
2. **Increased Vector Store Size:** More chunks mean more embeddings to store, potentially increasing the overall size of your vector database.
3. **Fragmented Information:** With such small chunks, related information will be split across multiple chunks, making it harder for the model to synthesize coherent answers.

Testing Different Models and Chunk Sizes

A good practice would be to experiment with different chunk sizes and embedding models and measure:

1. **Retrieval Quality:** Are the retrieved chunks relevant to our queries?
2. **Answer Accuracy:** Does the SLM generate correct and comprehensive answers?
3. **Memory Usage:** Is the system staying within the memory constraints of our device?
4. **Response Time:** How does chunk size affect latency?

We can create a simple benchmarking function to have one embedding model defined test the best chunk size:

```
def benchmark_chunk_sizes(document_list,
                           query_list,
                           sizes=[(100, 50), (300, 30), (500, 50), (1000, 100)]):
    """Test different chunk sizes and measure performance"""
    results = {}

    for chunk_size, overlap in sizes:
        print(f"Testing chunk_size={chunk_size}, overlap={overlap}")

        # Create splitter with current settings
        text_splitter = RecursiveCharacterTextSplitter.from_tiktoken_encoder(
            chunk_size=chunk_size, chunk_overlap=overlap
        )

        # Split documents
        start_time = time.time()
        doc_splits = text_splitter.split_documents(document_list)
        split_time = time.time() - start_time
```

```

# Create embeddings and store
embedding_function = OllamaEmbeddings(model="nomic-embed-text")
temp_db_path = f"temp_db_{chunk_size}_{overlap}"

start_time = time.time()
vectorstore = Chroma.from_documents(
    documents=doc_splits,
    collection_name="benchmark",
    embedding=embedding_function,
    persist_directory=temp_db_path
)
db_time = time.time() - start_time

# Create retriever
retriever = vectorstore.as_retriever(k=3)

# Test queries
query_times = []
for query in query_list:
    start_time = time.time()
    docs = retriever.invoke(query)
    query_time = time.time() - start_time
    query_times.append(query_time)

# Store results
results[(chunk_size, overlap)] = {
    "num_chunks": len(doc_splits),
    "splitting_time": split_time,
    "db_creation_time": db_time,
    "avg_query_time": sum(query_times) / len(query_times),
    "max_query_time": max(query_times),
    "min_query_time": min(query_times)
}

# Clean up temporary DB
shutil.rmtree(temp_db_path)

return results

```

Regarding the query side, some optimizations can also reduce the latency at the edge. Let's modify the previous script, with:

1. **Direct Ollama API Calls:** Bypasses the LangChain abstraction layer for embedding

and LLM generation to reduce overhead.

2. **Embedding Caching:** Uses `lru_cache` to prevent recalculating embeddings for repeated queries.
3. **Preloading Models:** Initializes models at startup to avoid cold-start latency.
4. **Optimized Retriever Settings:** Uses minimal k-value (2) and adds a score threshold to filter out irrelevant matches.
5. **Reduced Dependency Usage:** Removes unnecessary imports and simplifies the pipeline.
6. **Concurrent Processing:** Uses ThreadPoolExecutor for batch document embedding (when needed).
7. **Early Termination:** Checks for empty document results before running the LLM.
8. **Simplified Prompt:** Uses a more concise prompt template focused on getting direct answers.
9. **Fixed Seed:** Uses a consistent seed for the LLM to reduce variability in response times.

This optimized version (`25-optimized_RAG_query.py`) significantly reduces the latency compared to our original implementation while maintaining compatibility with our existing `nomic-embed-text` vector database and chunk size (300/30).

The direct Ollama API approach removes several layers of abstraction in the LangChain implementations.

We can see latency improvements from 2 minutes down to approximately 50-110 seconds, depending on the complexity of the queries.

```

marcelo_roval - miroval@raspi-5: ~/Documents/Ollama/Rag/edgeai - ssh miroval@192.168.4.209 - 100x36
Question: What is FOMO?
Retrieving documents...
Retrieved 2 document chunks
Generating answer...
Response latency: 107.41 seconds using model: llama3.2:3b

ANSWER:
=====
FOMO stands for Faster Objects, More Objects.
=====

Your question: How to setup a Raspi5?

Generating answer...

Question: How to setup a Raspi5?
Retrieving documents...
Retrieved 2 document chunks
Generating answer...
Response latency: 85.09 seconds using model: llama3.2:3b

ANSWER:
=====
To set up a Raspberry Pi 5, follow these steps:

1. Download and install the Raspberry Pi Imager on your computer.
2. Insert a 32GB microSD card into your computer.
3. Open Raspberry Pi Imager and select "Raspberry Pi OS (64-bit)" as the operating system.
4. Select the Raspberry Pi 5 model.
5. Set the hostname, username, password, configure WiFi, and enable SSH in the advanced options.
6. Write the image to the microSD card.

Note: The full 64-bit version of Raspberry Pi OS is recommended for the Raspberry Pi 5.
=====

Your question: 

```

In the next section, we'll explore how RAG can be combined with our agent architecture to create even more powerful edge AI systems.

Advanced Agentic RAG System

We can significantly enhance traditional RAG implementations by incorporating some of the modules discussed earlier, such as intelligent routing, validation feedback loops, and explicit knowledge gap identification. This will provide more reliable and transparent answers for users querying document-based knowledge bases.

For example, let's enhance the last RAG system created on the Edge AI Engineering dataset so that the agent can use tools, such as a calculator for arithmetic calculations.

Note that any tool could be used here; the calculator is only a simple example to demonstrate the concept.

When the user asks a question, the system first determines if it needs to use a tool or the RAG approach. For knowledge queries, the RAG system enhances the response with information from the database. The system then validates the answer quality, and if it's not sufficient, tries again with an improved prompt. In cases where questions fall outside the database's scope, the system will clearly inform the user rather than attempting to generate potentially misleading answers.

System Architecture

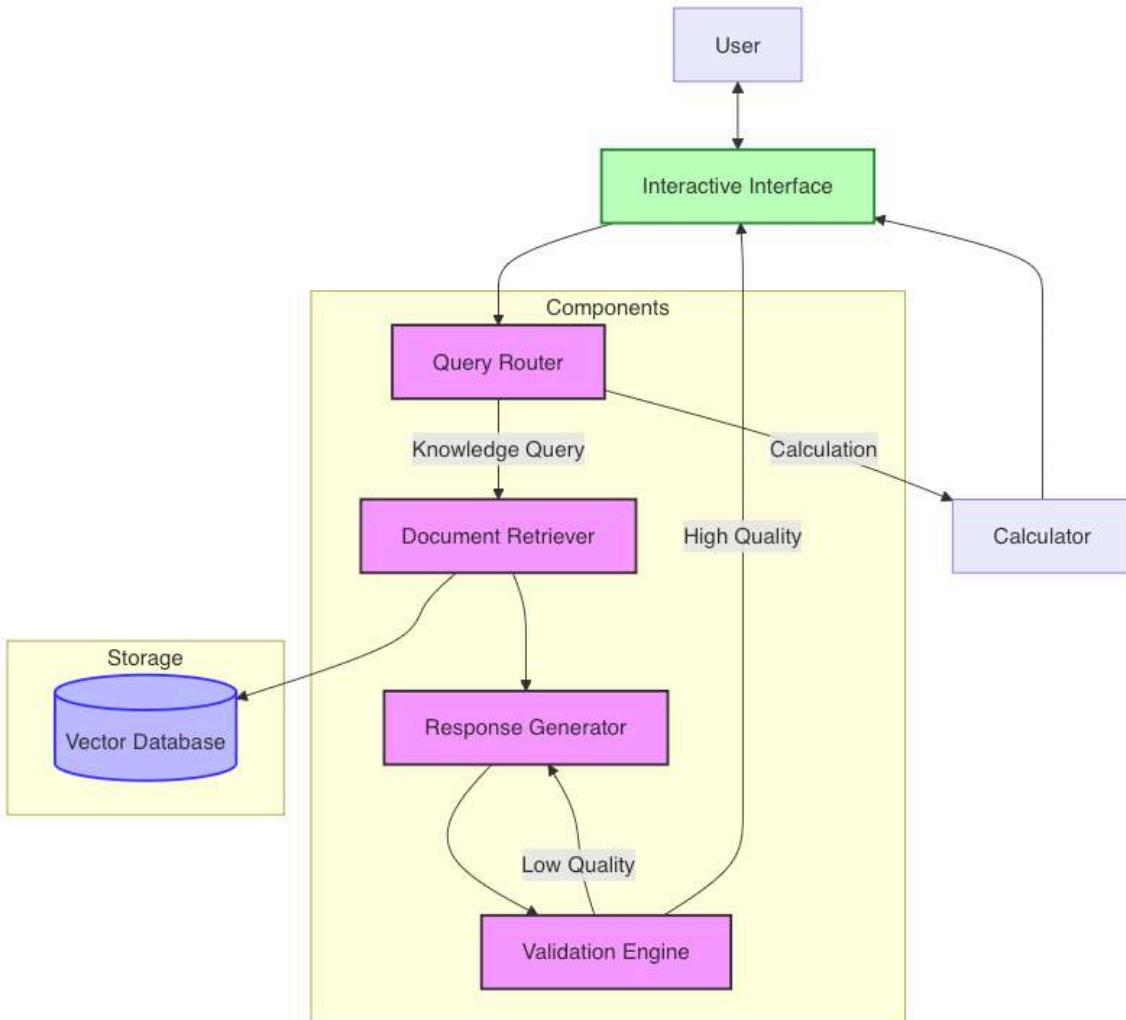


Figure 17: image-20250322113245945

The system functions through several key components:

1. Query Router

- Analyzes incoming queries to determine if they're calculations or knowledge queries
- Can use the same model for the response generator or a lightweight model to reduce overhead
- Implements rule-based fallbacks for robust classification

2. Document Retriever

- Connects to a persistent vector database (Chroma)
- Uses semantic embeddings to find relevant documents
- Returns contextually similar content for knowledge generation

3. Response Generator

- Creates answers based on retrieved documents
- Implements a two-stage approach with validation and improvement
- Adds appropriate disclaimers when information is insufficient

4. Validation Engine

- Evaluates answer quality using structured criteria
- Assigns a numerical score to each generated response
- Triggers enhancement processes when quality is insufficient

5. Interactive Interface

- Provides user-friendly interaction with clear quality indicators
- Supports model switching and verbosity control
- Offers guidance for improving query outcomes

Key Workflow

The system follows this high-level workflow:

1. User submits a query
2. Router determines the query type (calculation vs. knowledge)
3. For calculations:
 - Extract operation and numbers
 - Compute and return result
4. For knowledge queries:
 - Retrieve relevant documents
 - Generate initial answer with RAG

- Validate response quality
 - If quality is low, attempt enhancement with improved prompt
 - If still insufficient, add disclaimer about knowledge gaps
5. Return final answer with quality metrics

Important Code Sections

Query Routing

```
def route_query(query: str) -> Dict[str, Any]:
    """Determine if the query is a calculation, otherwise use RAG"""
    if VERBOSE:
        print(f"Routing query: {query}")

    # Check for calculation keywords
    calc_terms = ["+", "add", "plus", "sum", "-", "subtract", "minus",
                  "difference", "*", "x", "multiply", "times", "product",
                  "/", "÷", "divide",
                  "division", "quotient"]

    # Simple rule-based detection for calculations
    is_calc = any(term in query.lower() for term in calc_terms) and \
              re.search(r'\d+', query)

    if is_calc:
        # Use smaller, faster model for operation and number extraction
        # ...extraction logic here...
        return route_info

    # For everything else, use RAG
    return {"type": "rag", "reasoning": "Non-calculation query, using RAG"}
```

Enhanced RAG with Feedback Loop

```
# First RAG attempt with standard prompt
answer = get_answer_with_rag(query, documents, llm)
processing_type = "rag_standard"

# Validate the response quality
```

```

validation = validate_response(llm, query, answer)
validation_score = validation.get("score", 5)

# If validation score is low, try again with enhanced prompt
if validation_score < 7:
    if VERBOSE:
        print(f"First RAG attempt validation score: {validation_score}/10. \
              Trying enhanced prompt.")

    # Second RAG attempt with enhanced prompt
    enhanced_context = "\n\n".join(documents)
    enhanced_prompt = f"""
    I need a more detailed and accurate answer to the following question:

    {query}

    The previous answer wasn't satisfactory. Let me provide you with \
    relevant information:

    {enhanced_context}

    Based strictly on this information, provide a comprehensive answer.
    Focus specifically on addressing the user's question with precise \
    information from the provided context.
    If the information doesn't fully answer the question, clearly state \
    what you can determine
    from the available information and what remains unknown.
    """
    # ... process enhanced response ...

```

Knowledge Gap Handling

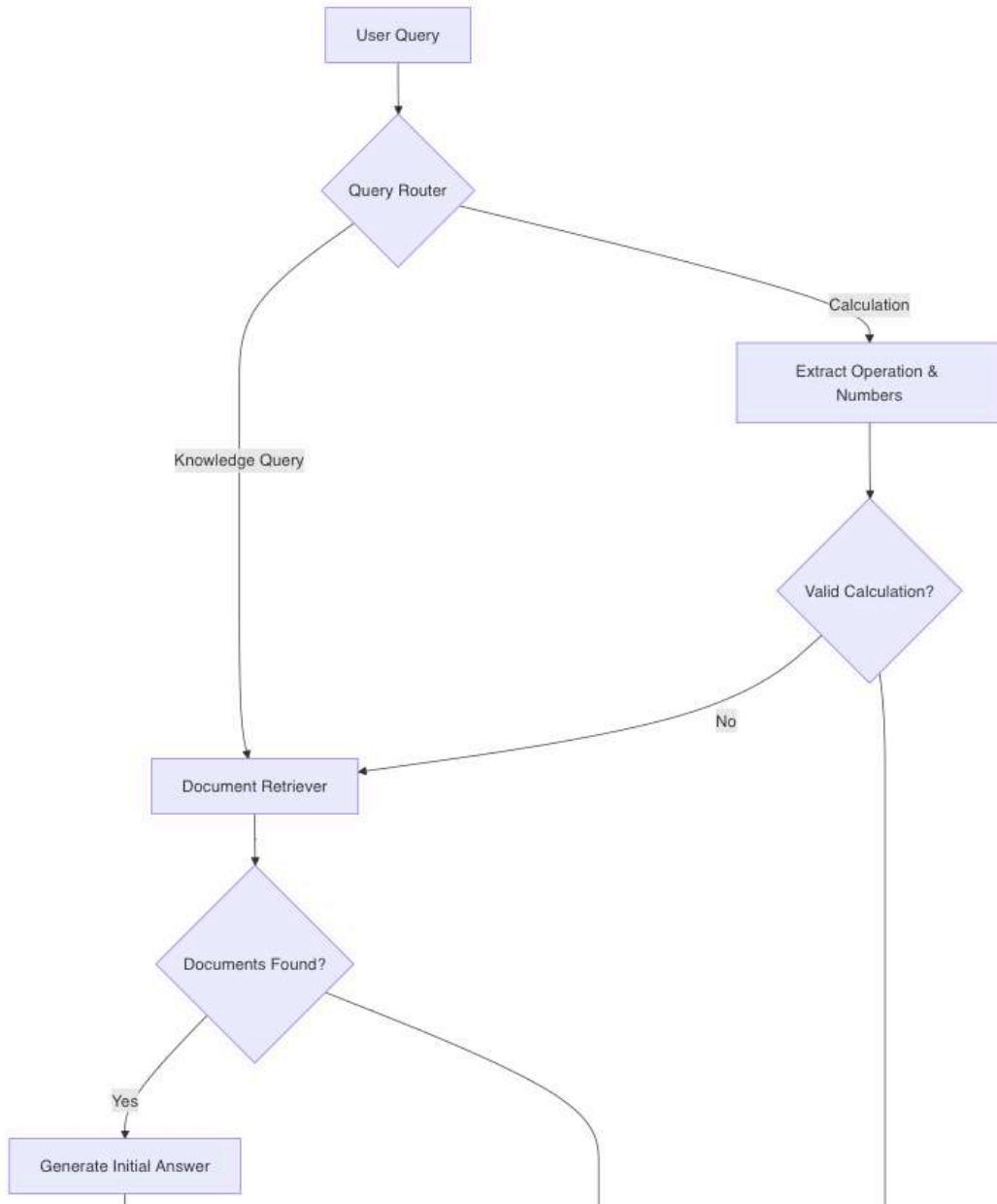
```

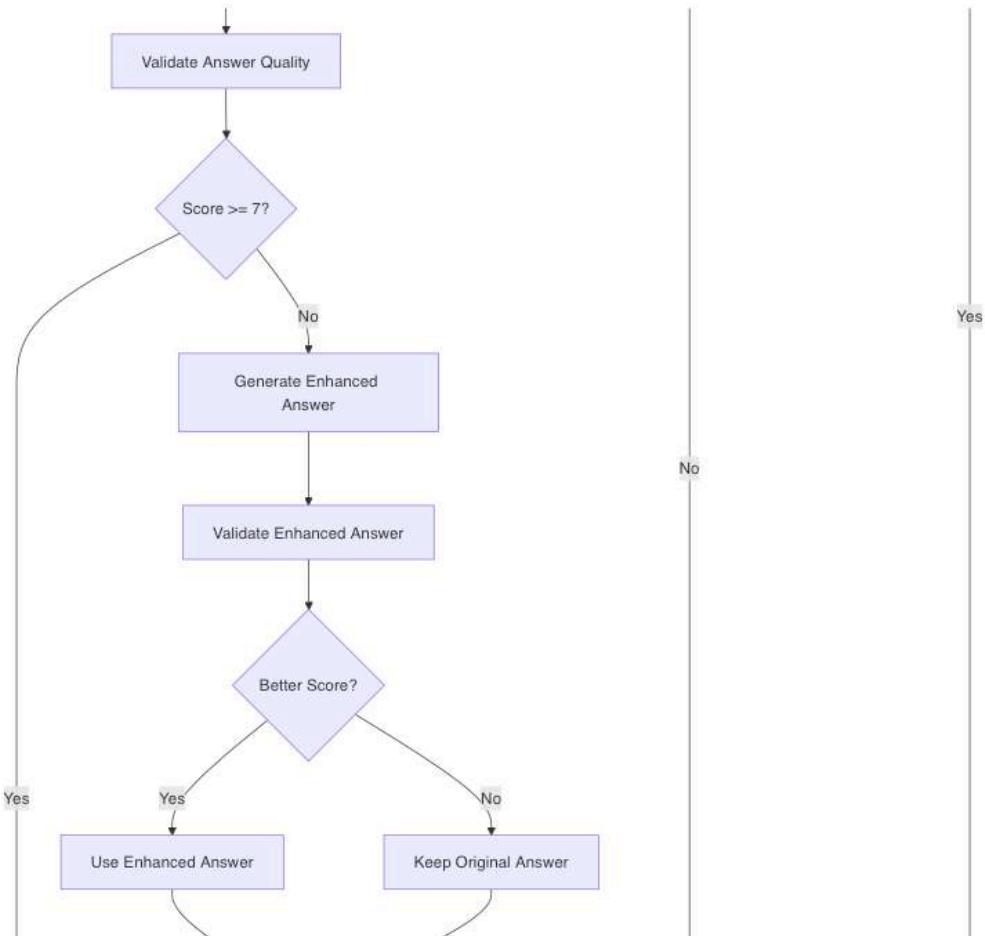
# If still low quality after enhancement, add a note
if improved_score < 6:
    processing_type = "rag_insufficient_info"
    information_gap_note = (
        "\n\nNote: The information in my knowledge base may be incomplete on this"
        "topic. I've provided the best answer based on available information, but"
        "there might be gaps or additional details that would provide a more "
        "complete answer."

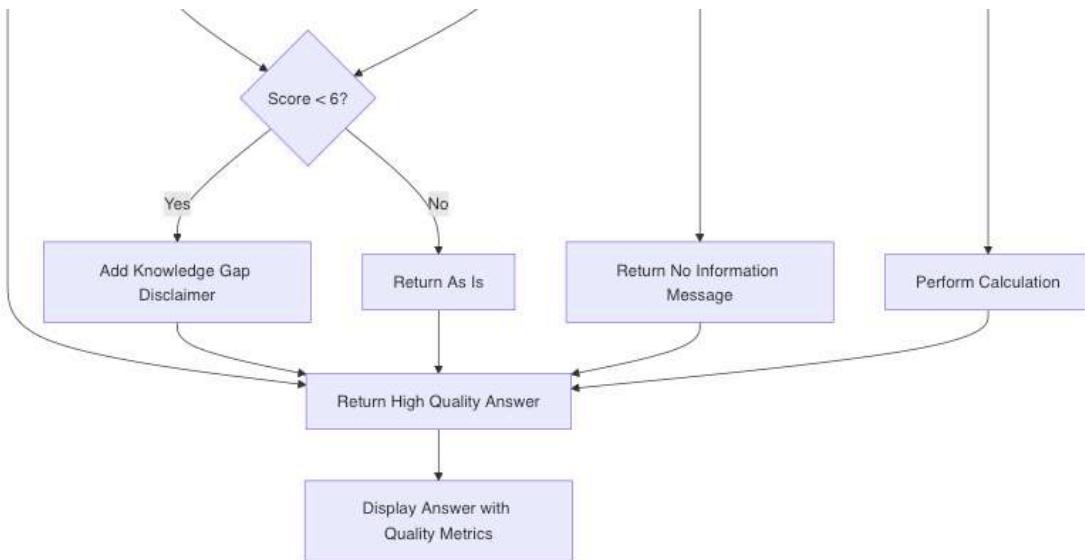
```

```
)  
answer = answer + information_gap_note
```

Detailed Workflow Diagram







Examples

Run the script `30-advanced_agentic_rag.py`:

Simple Calculation

```

marcelo_rovai - mirovai@raspi-5:~/Documents/Ollama/Rag/edgeai - ssh mirovai@192.168.4.209 - 97x15
Your question: Calculate 123456 times 123456.89

Processing...
Routing query: Calculate 123456 times 123456.89
Query type: calculation
Routing reasoning: No reasoning provided

ANSWER:
=====
The product of 123,456 and 123,456.89 is 15,241,493,811.84.
=====
Processing method: Direct calculation (bypassed RAG)
Processing time: 12.52 seconds
Answer quality: ★★★★★ (10/10)
High quality answer ✓
  
```

First Pass Rag

```
marcelo_royai — mjrovai@raspi-5: ~/Documents/Ollama/Rag/edgeai — ssh mjrovai@192.168.4.209 — 106x21
Your question: What is a Raspberry Pi?

Processing...
Routing query: What is a Raspberry Pi?
Query type: rag
Routing reasoning: Non-calculation query, using RAG
Retrieved 4 document chunks in 0.39s

ANSWER:
-----
The Raspberry Pi is a single-board computer that offers a unique combination of affordability, computational power, and extensive GPIO capabilities, making it ideal for prototyping, embedded systems development, and advanced engineering projects. It features a compact design with built-in Wi-Fi, Bluetooth, Ethernet, and multiple USB ports, as well as access to interfaces like I2C, SPI, and UART. The Raspberry Pi is also known for its low power consumption and real-time capabilities, making it suitable for battery-powered and energy-efficient designs.
-----
Processing method: Standard RAG
Processing time: 259.91 seconds
Answer quality: ★★★★☆ (9/10)
High quality answer ✓
```

More Complex Queries

```
marcelo_royai — mjrovai@raspi-5: ~/Documents/Ollama/Rag/edgeai — ssh mjrovai@192.168.4.209 — 106x54
Your question: Describe FOMO for Object Detection

Processing...
Routing query: Describe FOMO for Object Detection
Query type: rag
Routing reasoning: Non-calculation query, using RAG
Retrieved 4 document chunks in 0.51s
JSON decode error: Expecting value: line 3 column 15 (char 35)
First RAG Attempt validation score: 5/10. Trying enhanced prompt.
Enhanced RAG improved score from 5 to 9

ANSWER:
-----
Based on the provided context, I will provide a comprehensive and accurate answer to your question about FOMO for Object Detection.

FOMO (Faster Objects, More Objects) is a novel machine learning algorithm that enables real-time object detection with up to 30x less processing power and memory than traditional object detection models like MobileNet SSD or YOLO. The main difference between FOMO and other object detection models lies in its approach to calculating the location of objects within an image.

In traditional object detection models, a bounding box is drawn around each detected object, which requires calculating the size of the image. In contrast, FOMO ignores the size of the image and provides only the information about where the object is located in the image through its centroid coordinates.

Here's how FOMO works:
1. The input image is divided into blocks of pixels using a factor of 8.
2. A classifier is run through each pixel block to calculate the probability that there is an object (box or wheel) present in each block.
3. The regions with the highest probability of containing an object are determined, and from these overlaps, the FOMO model provides the coordinates (related to the image dimensions) of the centroid of the region.

The advantages of FOMO include:
* Up to 30x less processing power and memory requirements compared to traditional object detection models
* Real-time object detection capabilities

However, it's essential to note that FOMO is a simplified architecture focused on center-point detection, which may compromise on precision compared to more complex object detection models.

In terms of trade-offs between speed and precision, FOMO offers significant speed improvements at the cost of potentially lower precision. The exact balance between speed and precision will depend on the specific use case and requirements.

From the available information, it appears that FOMO is a viable option for edge devices like Raspberry Pi, where latency and processing power are critical factors. However, the trade-offs between speed and precision should be carefully evaluated to ensure that FOMO meets the required performance standards for object detection applications.

What remains unknown from the provided context is:
* The exact implementation details of FOMO, including the specific architecture and hyperparameters used.
* The evaluation metrics used to assess the performance of FOMO in comparison to other object detection models.
* The potential limitations or challenges associated with using FOMO for object detection applications.

Overall, FOMO appears to be a promising approach for real-time object detection on edge devices, offering significant speed improvements at the cost of potentially lower precision. However, further evaluation and testing are necessary to fully understand its capabilities and limitations.
-----
Processing method: Enhanced RAG with feedback loop
Processing time: 550.49 seconds
Answer quality: ★★★★☆ (9/10)
High quality answer ✓
```

Queries outside of the database scope:

```
marcelo_roval — mjroval@raspi-5: ~/Documents/Ollama/Rag/edgeai — ssh mjroval@192.168.4.209 — 103x42

Your question: Tell me about the countries where the next soccer world cup will be played

Processing...
Routing query: Tell me about the countries where the next soccer world cup will be played
Query type: rag
Routing reasoning: Non-calculation query, using RAG
Retrieved 4 document chunks in 0.39s
First RAG attempt validation score: 2/10. Trying enhanced prompt.
Enhanced RAG improved score from 2 to 4

ANSWER:
-----
Based on the provided context, I can provide a comprehensive answer to your question about the countries where the next soccer World Cup will be played.

Unfortunately, the provided articles and sources do not mention anything related to the location of the next Soccer World Cup. The articles appear to be focused on topics such as edge AI, sustainable agriculture, and emerging technologies, but they do not provide any information about the FIFA World Cup or its future locations.

The only relevant information that can be inferred from the provided context is that the Gartner analysts have described supply chains as increasingly dynamic and covering larger networks where data and decisions take place at the edge. However, this information does not provide any clues about the location of the next Soccer World Cup.

Therefore, I must conclude that the available information does not provide a clear answer to your question. The location of the next Soccer World Cup remains unknown based on the provided context.

However, it's worth noting that the FIFA World Cup is typically held in different countries around the world, and the host country for each tournament is usually announced several years in advance. If you're looking for information about the next FIFA World Cup, I recommend checking the official FIFA website or other reliable sources for updates on future tournaments.

Note: The information in my knowledge base may be incomplete on this topic. I've provided the best answer based on available information, but there might be gaps or additional details that would provide a more complete answer.
-----
Processing method: Limited information available in knowledge base
Processing time: 471.05 seconds
Answer quality: ★★ (4/10)
⚠️ The information in the knowledge base may be incomplete or missing for this query
```

Fine-Tuning SLMs for Edge Deployment

Fine-tuning can adapt models to specific domains or tasks, improving performance for targeted applications.

Preparing for Fine-Tuning

```
# Example dataset for fine-tuning a weather response model
training_data = [
    {"input": "What's the weather in London?",
     "output": "I need to check London's current weather. Please use the \
weather tool."},
    {"input": "Is it going to rain tomorrow in Paris?",
     "output": "To answer about tomorrow's weather in Paris, \
I need to use the weather tool."},
    {"input": "Will it be sunny this weekend in Tokyo?",
     "output": "To predict Tokyo's weekend weather, \
I should use the weather tool."}
]

# Format data for fine-tuning
formatted_data = []
for item in training_data:
    formatted_data.append({
        "prompt": item["input"],
        "response": item["output"]
    })

# Save formatted data to a file
import json
with open("weather_finetune_data.json", "w") as f:
    json.dump(formatted_data, f)
```

Setting Up a Fine-Tuning Process

Fine-tuning on edge devices is typically impractical due to resource constraints. Instead, fine-tune on a more powerful machine and deploy the result to the edge:

This is a conceptual example - actual implementation depends on the framework

```
def prepare_for_finetuning(data_path, output_path):
    """
    Prepare a model for fine-tuning
    (run this on a more powerful machine)
    """
```

```

# This is a conceptual example
print(f"Fine-tuning model using data from {data_path}")
print(f"Fine-tuned model will be saved to {output_path}")

# The process would typically involve:
# 1. Loading the base model
# 2. Loading and preprocessing the training data
# 3. Setting up training parameters (learning rate, epochs, etc.)
# 4. Running the fine-tuning process
# 5. Evaluating the fine-tuned model
# 6. Saving and optimizing for edge deployment

# For Ollama, we would create a custom model definition (Modelfile)
modelfile = f"""
FROM llama3.2:1b

# Fine-tuning settings would go here
PARAMETER temperature 0.7
PARAMETER top_p 0.9
PARAMETER top_k 40

# Custom system prompt for the specific domain
SYSTEM You are a specialized assistant for weather-related questions.
"""

with open(output_path, "w") as f:
    f.write(modelfile)

print("Model preparation complete. Next steps:")
print("1. Run fine-tuning on a powerful machine")
print("2. Optimize the resulting model for edge deployment")
print("3. Deploy to your Raspberry Pi")

```

Real implementation: Supervised Fine-Tuning (SFT)

Supervised fine tuning (SFT) is a method to **improve and customize** pre-trained LLMs. It involves retraining base models on a smaller dataset of instructions and answers. The main goal is to transform a basic model that predicts text into an assistant that can follow instructions and answer questions. SFT can also enhance the model's overall performance, add new knowledge, or adapt it to specific tasks and domains.

Before considering SFT, it is recommended to try prompt engineering techniques like few-shot

prompting or retrieval augmented generation (RAG), as discussed previously. In practice, these methods can solve many problems without fine-tuning. If this approach doesn't meet our objectives (regarding quality, cost, latency, etc.), then SFT becomes a viable option when instruction data is available. SFT also offers benefits like additional control and customizability to create personalized LLMs.

However, SFT has limitations. It works best when leveraging knowledge already present in the base model. Learning completely new information, like an unknown language, can be challenging and lead to more frequent hallucinations. For new domains unknown to the base model, it is recommended that it be continuously pre-trained on a raw dataset first.

On the opposite end of the spectrum, instruct models (i.e., already fine-tuned models) can be very close to our requirements. By providing chosen and rejected samples for a small set of instructions (between 100 and 1000 samples), we can force the LLM to behave as we need.

The easiest way to finetune an SLM is by using [Unsloth](#). The three most popular SFT techniques are full fine-tuning, LoRA, and QLoRA.

For details, see: [Fine-tune Llama 3.1 Ultra-Efficiently with Unsloth](#).

For example, using this [link](#), it is possible to find several notebooks with the steps to finetune SLMs—for instance, the [Gemma 3:1B](#).

The fine-tuned model can be saved on HF Hub or locally as **GGUF**, and to run a GGUF model locally, we can use Ollama, as shown below:

1. Download the finetuned GGUF model
2. Create a Modelfile in the home directory:

```
cd ~  
nano Modelfile
```

3. In the Modelfile, specify the path to the GGUF file:

```
FROM ~/Downloads/your-model-name.gguf  
PARAMETER temperature 1.0  
PARAMETER top_p 0.95  
PARAMETER top_k 64
```

4. Save the Modelfile and exit the editor.
5. Create the loadable model in Ollama:

```
ollama create your-model-name -f Modelfile
```

The model name here can be anything.

6. We can now use the model through as we have done with the llama3.2:3B in this chapter..

Conclusion

This chapter has explored comprehensive strategies for overcoming the inherent limitations of Small Language Models in edge computing environments. By implementing techniques ranging from optimized prompting strategies to sophisticated agent architectures and knowledge integration systems, we've demonstrated that it's possible to significantly enhance the capabilities of edge AI systems without requiring more powerful hardware or cloud connectivity.

The techniques presented—chain-of-thought prompting, task decomposition, function calling, response validation, and RAG—form a toolkit that edge AI engineers can apply individually or in combination to address specific challenges. Each approach offers unique advantages: prompting techniques improve reasoning capabilities with minimal overhead, agent architectures enable SLMs to perform actions beyond text generation, and RAG systems dramatically expand an SLM's knowledge without increasing model size.

Our practical implementations on the Raspberry Pi showcase that these enhancements are not merely theoretical but can be deployed in real-world edge scenarios. From the simple calculator agent to the more sophisticated knowledge router and RAG-enabled question answering system, these examples provide templates that developers can adapt to their specific application requirements.

The true power of these techniques emerges when they're strategically combined. An agent architecture with RAG capabilities, enhanced by chain-of-thought reasoning and validated with a feedback loop, creates an edge AI system that approaches the capabilities of much larger models while maintaining the advantages of edge deployment—privacy preservation, reduced latency, and operation without internet connectivity.

As edge AI continues to evolve, these techniques will become increasingly important in bridging the gap between the limited resources available on edge devices and the growing expectations for AI capabilities. By thoughtfully applying these approaches, developers can create intelligent systems that process data locally, respect user privacy, and operate reliably in diverse environments.

The future of edge AI lies not necessarily in deploying ever-larger models but in developing more innovative systems that combine efficient models with intelligent architectures, contextual knowledge integration, and robust validation mechanisms. By mastering these techniques, edge AI practitioners can create solutions that are not just technologically impressive but genuinely useful and trustworthy in addressing real-world challenges.

Resources

The scripts used in this chapter can be found here: [Advancing EdgeAI Scripts](#)

#

Weekly Labs

Edge AI Engineering - Weekly Labs

Week 1: Introduction and Setup

Lab 1: Raspberry Pi Configuration

Objectives:

- Install Raspberry Pi OS using Raspberry Pi Imager
- Configure basic settings (hostname, SSH, WiFi)
- Learn essential Linux commands
- Manage files between your computer and Raspberry Pi

Instructions:

1. Download Raspberry Pi Imager on your computer
2. Configure OS settings (enable SSH, set hostname, WiFi credentials)
3. Boot your Raspberry Pi and confirm connectivity
4. Learn how to use SSH for remote access
5. Transfer files using SCP or FileZilla
6. Update your Raspberry Pi OS (`sudo apt update && sudo apt upgrade`)
7. Practice basic Linux commands (ls, cd, mkdir, cp, mv)

Deliverable: Screenshot showing successful SSH connection to your Raspberry Pi

Lab 2: Development Environment Setup

Objectives:

- Set up Python environment for development
- Configure remote development tools
- Install essential libraries
- Test camera functionality

Instructions:

1. Install Python essentials: `pip install jupyter matplotlib numpy pillow`

2. Configure Jupyter Notebook for remote access:

```
pip install jupyter  
jupyter notebook --generate-config  
jupyter notebook --ip=0.0.0.0 --no-browser
```

3. Connect the camera module (USB or CSI) to your Raspberry Pi
4. Test camera functionality using command-line tools
5. Write a simple Python script to capture an image

Deliverable: A simple Python script that captures and displays an image from your camera and a Screenshot showing a successful image capture

Week 2: Image Classification Fundamentals

Lab 3: Working with Pre-trained Models

Objectives:

- Install TensorFlow Lite runtime
- Download and run MobileNet V2 model
- Process and classify images
- Understand model inputs and outputs

Instructions:

1. Install TensorFlow Lite runtime: `pip install tflite_runtime`
2. Download MobileNet V2 model:

```
wget https://storage.googleapis.com/download.tensorflow.org/models/tflite_11_05_08/mobil
```

3. Download labels file
4. Create a Python script that:
 - Loads the TFLite model
 - Processes input images to 224x224 format
 - Runs inference on test images
 - Displays top-5 predicted classes with confidence scores

Deliverable: Python script that successfully classifies sample images with MobileNet V2 and a Screenshot showing a successful result

Lab 4: Custom Dataset Creation

Objectives:

- Create a simple custom dataset using Raspberry Pi camera
- Organize images into classes
- Prepare dataset for model training

Instructions:

1. Create a web interface for image capture:
 - Use Flask to create a simple web server
 - Set up camera preview and capture functionality
 - Save captured images with appropriate filenames
2. Capture at least 50 images per class for 3 classes
3. Organize the dataset into an appropriate directory structure
4. Document your dataset creation process

Deliverable: Structured dataset with at least 3 classes and 50 images per class

Week 3: Custom Image Classification

Lab 5: Edge Impulse Model Training

Objectives:

- Create an Edge Impulse project
- Upload and process the dataset
- Design and train a transfer learning model
- Evaluate model performance

Instructions:

1. Create an Edge Impulse account and a new project
2. Upload your custom dataset
3. Create an impulse design:

- Set image size to 160x160
 - Use Transfer Learning for feature extraction
4. Generate features for all images
 5. Train model using MobileNet V2
 6. Analyze model performance (accuracy, confusion matrix)
 7. Test model on validation data

Deliverable: Edge Impulse project link and screenshot of model performance metrics

Lab 6: Model Deployment to Raspberry Pi

Objectives:

- Export trained model to TFLite format
- Deploy model to Raspberry Pi
- Create a real-time inference application
- Optimize inference speed

Instructions:

1. Export model as TensorFlow Lite (.tflite)
2. Transfer the model to Raspberry Pi
3. Create a Python application that:
 - Captures live images from the camera
 - Preprocesses images for the model
 - Runs inference and displays results
 - Shows confidence scores
4. Implement a web interface for real-time classification

Deliverable: Python script for real-time image classification with your custom model and a Screenshot showing a successful result

Week 4: Object Detection Fundamentals

Lab 7: Pre-trained Object Detection

Objectives:

- Understand object detection architecture
- Run pre-trained SSD-MobileNet model
- Process detection outputs
- Visualize detected objects

Instructions:

1. Download the pre-trained SSD-MobileNet V1 model
2. Create a Python script that:
 - Loads the model and labels
 - Preprocesses input images
 - Runs inference
 - Extracts bounding boxes, classes, and scores
 - Implements Non-Maximum Suppression (NMS)
 - Visualizes detections with bounding boxes
3. Test on various images with multiple objects

Deliverable: Python script that performs and visualizes object detection on test images and a Screenshot showing a successful result.

Lab 8: EfficientDet and FOMO Models

Objectives:

- Compare different object detection architectures
- Implement EfficientDet and FOMO models
- Analyze performance differences
- Understand trade-offs between models

Instructions:

1. Download the EfficientDet Lite0 model
2. Implement inference with EfficientDet
3. Compare with SSD-MobileNet implementation
4. Learn about FOMO (Faster Objects, More Objects)
5. Analyze trade-offs in accuracy vs. speed
6. Measure inference time on Raspberry Pi

Deliverable: Comparison report of SSD-MobileNet vs. EfficientDet with performance metrics and visualized results

Week 5: Custom Object Detection

Lab 9: Dataset Creation and Annotation

Objectives:

- Create an object detection dataset
- Learn annotation techniques
- Prepare dataset for model training

Instructions:

1. Capture at least 100 images containing objects to detect
2. Upload images to Roboflow or a similar annotation tool
3. Create bounding box annotations for each object
4. Apply data augmentation (rotation, brightness adjustment)
5. Export dataset in YOLO format
6. Document the annotation process

Deliverable: Annotated dataset with at least 2 object classes and 100 total images

Lab 10: Training Models in Edge Impulse

Objectives:

- Upload annotated dataset to Edge Impulse
- Train SSD MobileNet object detection model
- Evaluate model performance
- Export model for deployment

Instructions:

1. Create a new Edge Impulse project for object detection
2. Upload annotated dataset (train/test splits)
3. Create object detection impulse
4. Train SSD MobileNet model
5. Evaluate model performance
6. Export model as TensorFlow Lite

Deliverable: Edge Impulse project link with trained object detection model and performance metrics

Week 6: Advanced Object Detection

Lab 11: FOMO Model Training

Objectives:

- Understanding FOMO architecture benefits
- Train FOMO model on Edge Impulse
- Compare performance with SSD MobileNet
- Deploy optimized model to Raspberry Pi

Instructions:

1. Create a new impulse in Edge Impulse using the same dataset
2. Train FOMO model instead of SSD MobileNet
3. Compare inference speed and accuracy
4. Deploy both models to Raspberry Pi
5. Create an application that can switch between models
6. Measure and document performance differences

Deliverable: Python application that compares SSD MobileNet vs. FOMO performance in real-time

Lab 12: YOLO Implementation

Objectives:

- Install and configure Ultralytics YOLO
- Convert models to optimized NCNN format
- Create real-time detection application
- Implement object counting

Instructions:

1. Install Ultralytics: `pip install ultralytics`
2. Download and test YOLO (n) model
3. Export model to NCNN format for optimization
4. Create Python script for real-time detection
5. Implement object counting algorithm
6. Add visualization of counts over time

Deliverable: Python application for real-time object detection and counting using YOLO

Week 7: Object Counting Project

Lab 13: Custom YOLO Training

Objectives:

- Train YOLO on a custom dataset
- Optimize model for edge deployment
- Create a complete application for object counting

Instructions:

1. Train the YOLO model on your custom dataset
 - Use Google Colab for training if needed
 - Set appropriate hyperparameters
2. Export optimized model for Raspberry Pi
3. Create a Python application that:
 - Captures video feed
 - Detects objects using YOLO
 - Counts objects over time
 - Logs results to a database

Deliverable: Complete object counting application with data logging

Lab 14: Fixed-Function AI Integration (Optional)

Objectives:

- Integrate multiple AI models into a single application
- Create a dashboard for visualization
- Optimize application for long-term deployment

Instructions:

1. Create an integration application that combines:
 - Object detection capabilities
 - Classification for detected objects
 - Counting and tracking over time
2. Implement a simple web dashboard for visualization
3. Add performance monitoring
4. Configure application for startup at boot

Deliverable: Integrated application combining multiple AI capabilities with visualization dashboard

Week 8: Introduction to Generative AI

Lab 15: Raspberry Pi Configuration for SLMs

Objectives:

- Optimize Raspberry Pi for running Small Language Models
- Install an active cooling solution
- Configure memory and swap
- Install essential libraries

Instructions:

1. Install active cooling solution on Raspberry Pi 5

2. Optimize system configuration:

- Increase swap memory: `sudo dphys-swapfile swapoff`, edit `/etc/dphys-swapfile`
- Set CONF_SWAPSIZE to 2048
- `sudo dphys-swapfile setup && sudo dphys-swapfile swapon`

3. Install dependencies:

```
sudo apt update
sudo apt install build-essential python3-dev
```

Deliverable: Screenshot showing system configuration with increased swap and temperature monitor during stress test

Lab 16: Ollama Installation and Testing

Objectives:

- Install Ollama framework
- Pull and test Small Language Models
- Benchmark model performance
- Monitor resource usage

Instructions:

1. Install Ollama:

```
curl -fsSL https://ollama.com/install.sh | sh
```

2. Pull different models (such as):

```
ollama pull llama3.2:1b  
ollama pull gemma:2b  
ollama pull phi3:latest
```

3. Run a basic inference test with each model

4. Measure and compare:

- Load time
- Inference speed (tokens/sec)
- Memory usage
- Temperature

Deliverable: Benchmark report comparing performance metrics of different SLM models on your Raspberry Pi

Week 9: SLM Python Integration

Lab 17: Ollama Python Library

Objectives:

- Use the Ollama Python library
- Create interactive applications
- Process SLM responses programmatically
- Handle multiple conversation turns

Instructions:

1. Install Ollama Python library: `pip install ollama`
2. Create Python script to:

- Connect to Ollama API
- Send prompts to models
- Process and format responses

- Handle conversation context
3. Implement proper error handling
 4. Create a simple interactive CLI application

Deliverable: Python script demonstrating Ollama library usage with conversation handling

Lab 18: Function Calling and Structured Outputs

Objectives:

- Implement function calling with SLMs
- Create applications with structured outputs
- Build validation mechanisms
- Handle image inputs

Instructions:

1. Install required libraries:

```
pip install pydantic instructor openai
```

2. Create Pydantic models for structured outputs
3. Implement function calling with an instructor:

```
client = instructor.patch(    OpenAI(base_url="http://localhost:11434/v1", api_key="o")
```

4. Build distance calculator application using SLM for city/country recognition
5. Add image input processing

Deliverable: Python application that uses function calling for structured interaction with SLMs

Week 10: Retrieval-Augmented Generation

Lab 19: RAG Fundamentals

Objectives:

- Understand RAG architecture
- Create vector database
- Implement embedding generation
- Build simple RAG system

Instructions:

1. Install required libraries:

```
pip install langchain chromadb
```

2. Create a simple dataset with text documents
3. Implement document splitting and chunking
4. Generate embeddings using Ollama
5. Store embeddings in ChromaDB
6. Create query system

Deliverable: Python implementation of a basic RAG system with simple text documents

Lab 20: Advanced RAG

Objectives:

- Optimize RAG for edge devices
- Implement more efficient retrieval
- Create a specialized knowledge base
- Build validation mechanisms

Instructions:

1. Create a specialized knowledge base (e.g., technical documentation)
2. Implement optimized embedding generation
3. Fine-tune retrieval parameters
4. Add response validation
5. Create a persistent vector store
6. Benchmark performance

Deliverable: Optimized RAG implementation with specialized knowledge base and performance analysis

Week 11: Vision-Language Models

Lab 21: Florence-2 Setup

Objectives:

- Install Florence-2 model
- Configure environment
- Run basic inference tests
- Understand model capabilities

Instructions:

1. Install required dependencies:

```
pip install transformers torch torchvision torchaudio  
pip install timm einops  
pip install autodistill-florence-2
```

2. Download model and test basic functionality
3. Run image captioning test
4. Measure performance (memory usage, inference time)

Deliverable: Python script demonstrating basic Florence-2 functionality with performance metrics

Lab 22: Vision Tasks with Florence-2

Objectives:

- Implement various vision tasks
- Create applications for captioning, detection, grounding
- Optimize performance
- Combine tasks

Instructions:

1. Implement image captioning:
 - Basic caption generation
 - Detailed caption generation
2. Implement object detection:
 - Bounding box visualization
 - Multiple object detection
3. Implement visual grounding:
 - Highlight specific objects based on text prompts
4. Create segmentation application
5. Measure the performance of each task

Deliverable: Python application demonstrating multiple vision tasks with Florence-2 and performance analysis

Week 12: Physical Computing Basics

Lab 23: Sensor and Actuator Integration

Objectives:

- Connect digital sensors
- Read environmental data
- Control LEDs and actuators
- Create a data collection system

Instructions:

1. Connect hardware components:
 - DHT22 temperature/humidity sensor
 - BMP280 pressure sensor
 - LEDs (red, yellow, green)
 - Push button
2. Install required libraries:

```
pip install adafruit-circuitpython-dht adafruit-circuitpython-bmp280
```

3. Create a Python script to read sensor data
4. Implement LED control based on conditions
5. Create visualization of sensor data

Deliverable: Python application for reading sensor data and controlling actuators with visualization

Lab 24: Jupyter Notebook Integration

Objectives:

- Use Jupyter Notebook for physical computing
- Create interactive widgets
- Visualize sensor data in real-time
- Control actuators from a notebook

Instructions:

1. Install ipywidgets: `pip install ipywidgets`
2. Create a Jupyter Notebook for sensor data collection
3. Implement interactive widgets for control
4. Create real-time visualization
5. Build a dashboard with multiple data views

Deliverable: Jupyter Notebook with interactive widgets for sensor monitoring and actuator control

Week 13: SLM-Physical Computing Integration

Lab 25: Basic SLM Analysis

Objectives:

- Integrate SLMs with sensor data
- Create analysis application
- Implement decision-making logic
- Control actuators based on SLM responses

Instructions:

1. Create a Python application that:
 - Collects sensor data
 - Formats data for SLM prompt
 - Sends prompt to model
 - Parses response
 - Controls actuators based on response
2. Implement multiple analysis modes
3. Add error handling for SLM responses

Deliverable: Python application integrating SLMs with physical sensors and actuators

Lab 26: SLM-IoT Control System

Objectives:

- Create a complete IoT monitoring system
- Implement natural language interaction
- Add data logging and analysis
- Create web interface

Instructions:

1. Build a complete system with:
 - Sensor data collection
 - SLM-based analysis
 - Natural language command processing
 - Data logging to the database
 - Web interface for interaction
2. Implement multiple SLM models
3. Add historical data analysis
4. Create visualization dashboard

Deliverable: Complete IoT monitoring system with SLM integration and web interface

Week 14: Advanced Edge AI Techniques

Lab 27: Building Agents

Objectives:

- Create agent architecture
- Implement tool usage
- Build decision-making system
- Handle complex tasks

Instructions:

1. Implement calculator agent:
 - Create a query routing system
 - Implement tool functions
 - Build decision-making logic
2. Create knowledge router:
 - Implement web search integration
 - Build classification system
 - Handle time-based queries
3. Measure and optimize performance

Deliverable: Python implementation of agent architecture with tool usage and decision routing

Lab 28: Advanced Prompting and Validation

Objectives:

- Implement chain-of-thought prompting
- Create few-shot learning examples
- Build task decomposition system
- Implement response validation

Instructions:

1. Create examples for different prompting strategies
2. Implement chain-of-thought framework
3. Build few-shot learning templates
4. Create a task decomposition system

5. Implement validation mechanisms
6. Compare the effectiveness of different strategies

Deliverable: Python implementation demonstrating different prompting strategies with performance comparison

Week 15: Final Project Integration

Lab 29: Agentic RAG System

Objectives:

- Combine agent architecture with RAG
- Create a complete knowledge system
- Implement advanced validation
- Build query optimization

Instructions:

1. Create a complete agentic RAG system:
 - Build knowledge database
 - Implement agent architecture
 - Add tool functions
 - Create validation mechanisms
 - Optimize retrieval
2. Test with complex queries
3. Measure performance
4. Create visualization of system components

Deliverable: Complete agentic RAG system with documentation and performance analysis

Lab 30: Final Project

Objectives:

- Design and implement a comprehensive Edge AI system
- Combine multiple techniques
- Create complete documentation
- Present project

Instructions:

1. Design final project combining:
 - Computer vision capabilities
 - SLM integration
 - Physical computing
 - Advanced techniques (RAG, agents, etc.)
2. Implement complete system
3. Create documentation
4. Measure performance
5. Prepare presentation

Deliverable: Complete final project with documentation, code, and presentation

Hardware Requirements

Basic Setup (Weeks 1-7)

- Raspberry Pi Zero 2W or Pi 5
- MicroSD card (32GB+)
- Camera module (USB webcam or Pi camera)
- Power supply

Generative AI (Weeks 8-15)

- Raspberry Pi 5 (8GB RAM recommended)
- Active cooling solution
- MicroSD card (64GB+ recommended)

Physical Computing (Weeks 12-15)

- DHT22 temperature/humidity sensor
- BMP280 pressure/temperature sensor
- LEDs (red, yellow, green)
- Push button
- Resistors (4.7kΩ, 330Ω)
- Jumper wires

- Breadboard
-

Software Requirements

Development Environment

- Raspberry Pi OS (64-bit)
- Python 3.9+
- Jupyter Notebook
- SSH client

Computer Vision and DL

- TensorFlow Lite runtime
- OpenCV
- Edge Impulse
- Ultralytics

Generative AI

- Ollama
- Transformers
- Pytorch
- ChromaDB
- LangChain
- Pydantic
- Instructor

Physical Computing

- GPIO Zero
 - Adafruit CircuitPython libraries
-

Assessment Criteria

Each lab will be evaluated based on:

1. **Functionality (40%)**: Does the implementation work as specified?
2. **Code Quality (20%)**: Is the code well-structured, documented, and efficient?
3. **Documentation (20%)**: Are the process and results documented?
4. **Analysis (20%)**: Is there a thoughtful analysis of results and performance?

The final project will be evaluated based on:

1. **Integration (30%)**: How well different components are integrated
 2. **Innovation (20%)**: Novel approaches or applications
 3. **Implementation (30%)**: Overall quality and functionality
 4. **Presentation (20%)**: Clear explanation and demonstration
-

Tips for Success

1. **Start Early**: These labs build on each other. Falling behind makes later labs more difficult.
2. **Document As You Go**: Take notes, screenshots, and document issues/solutions.
3. **Optimize Resources**: SLMs and VLMs require careful resource management.
4. **Collaborate**: Discuss approaches with classmates while ensuring individual work.
5. **Backup Regularly**: Create backups of your SD card after significant progress.
6. **Measure Performance**: Always benchmark and optimize your implementations.
7. **Ask Questions**: If you're stuck, ask for help early rather than falling behind.

#

References & Author

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” by Vijay Janapa Reddi

Projects Repository

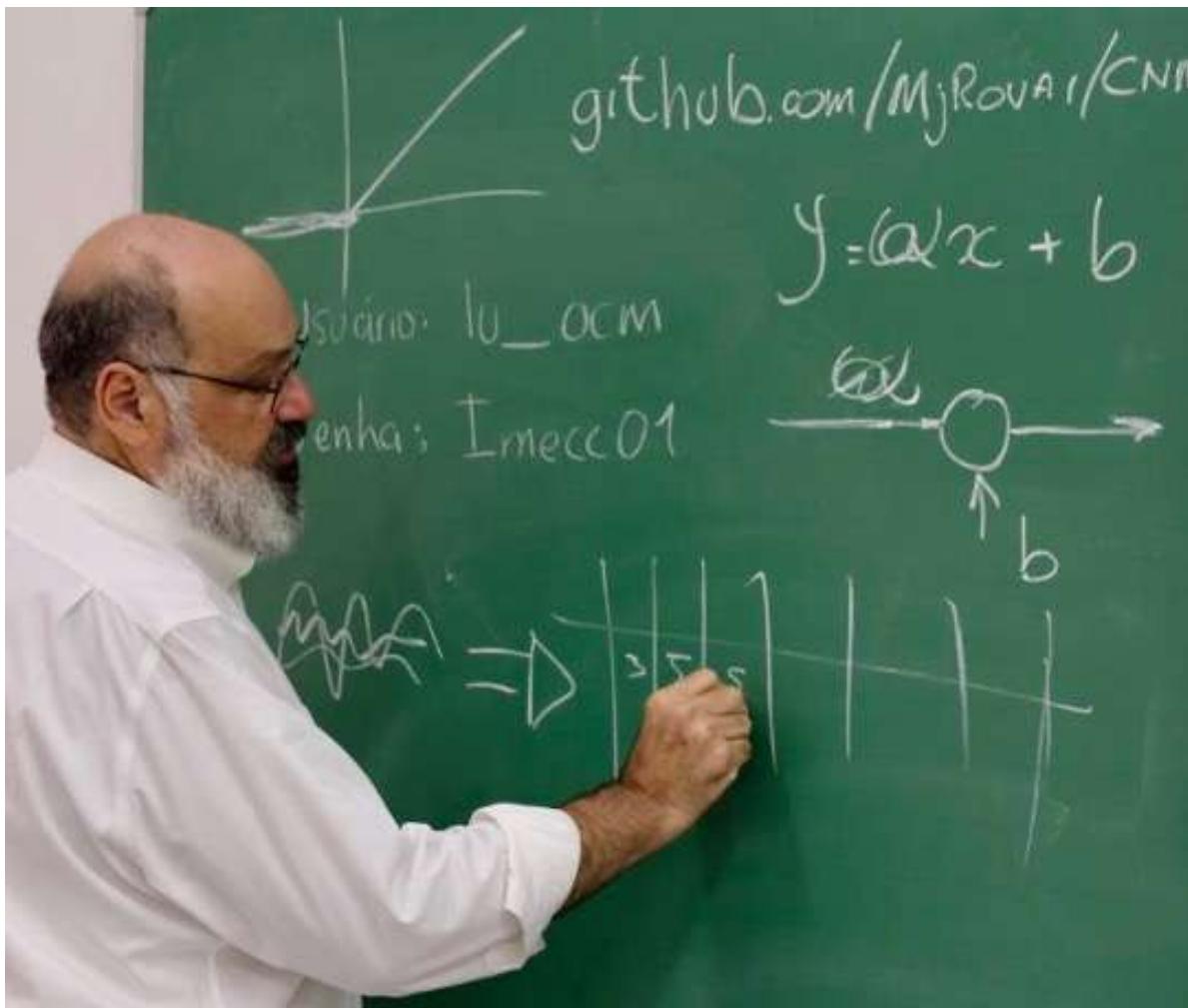
- Edge Impulse Expert Network

TinyML4D

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



About the author



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

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

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

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

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

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