IoT Systems Project | Year 2022/2023

Federico Bertoli | 157896

Github Repository: https://github.com/thebertozz/IoT-TinyML

Introduction

The goals of this project are two:

The **first** goal is to evaluate the performance of a set on ML models, specifically TinyML models, running on 3 low power IoT devices:

- ESP32 Dev Module
- ESP32 Wemos D1 R32
- ESP8266

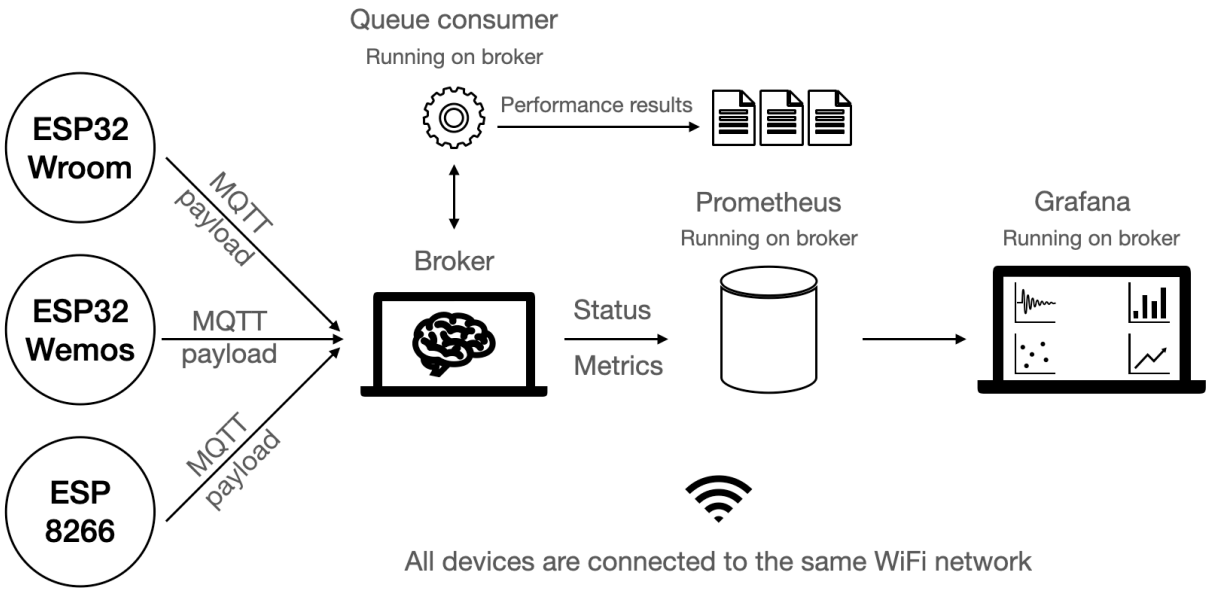
The boards have the following hardware specs:

Board	Platform	CPU	RAM	Flash
ESP 32 Dev	Espressif 32 (6.1.0)	ESP32 240Mhz	320KB	4 MB
ESP 32 Wemos	Espressif 32 (6.1.0)	ESP32 240Mhz	320KB	4 MB
8266 ESP	Espressif 8266	ESP8266 80Mhz	80KB	4 MB

All of these boards have WiFi capability and expandability for connecting external devices.

The **second** goal is to have an infrastructure that mimics a real scenario as realistically as possibile, with a common project template to be able to deploy the models on the devices, connect to a WiFi network and send MQTT payloads to a RabbitMQ broker; also, since monitoring is very important, metrics collection has been implemented, with the status of the queue monitored via Prometheus and then visualized on a Grafana dashboard.

The complete project architecture is the following:



The software components

TinyML

TinyML stands for running Machine Learning algorithms on constrained hardware, typically with strong memory and power constraints.

For this project I used a TensorFlow Lite for Microcontrollers wrapper (https://www.tensorflow.org/lite/microcontrollers) which is called EloquentTinyML (https://eloquentarduino.com/); it simplifies the deploy of the models and exposes simpler APIs for running TensorFlow.

ML Models

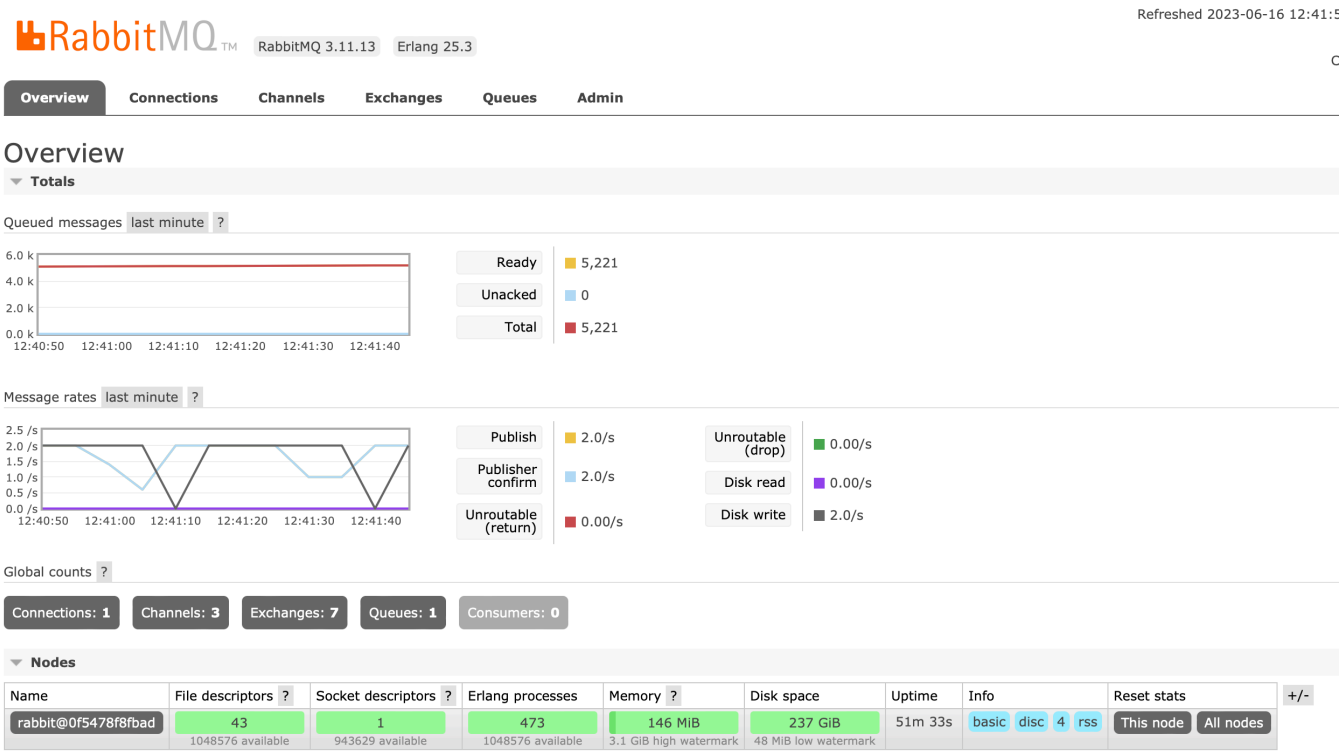
3 stand alone ML models have been used:

- Sine (https://github.com/eloquentarduino/EloquentTinyML/tree/master/examples/SineExample): a simple example for picking a random x and calculating its sine
- Wine (https://www.kaggle.com/code/cristianlapenta/wine-dataset-sklearn-machine-learning-project): a model used for classification between 3 different classes of wine
- Digits (https://scikit-learn.org/stable/auto_examples/datasets/plot_digits_last_image.html): a model with 1797 images, with a resolution of 8x8, to test the number detection

Another model has been used to test the camera recognition feature, specifically a model to find if a person is in front of the camera. This model uses a neural network and is included in the EloquentTinyML library.

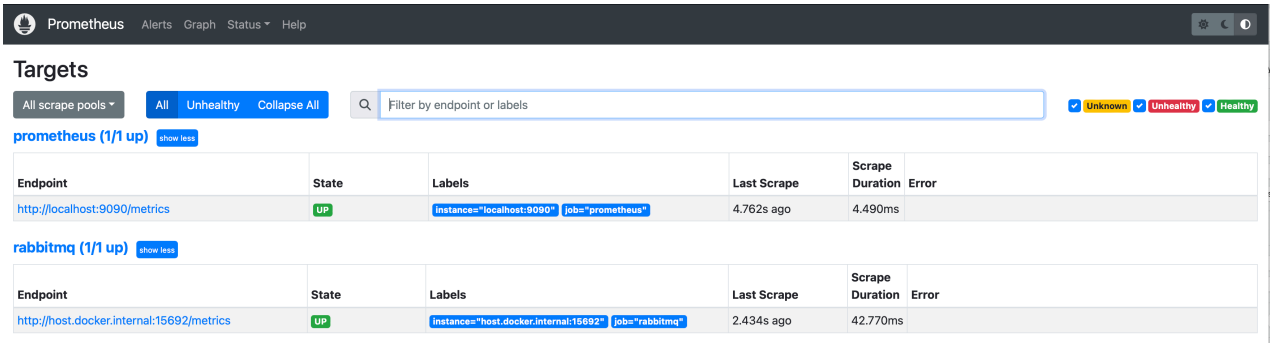
Rabbit MQ (https://www.rabbitmq.com/)

RabbitMQ is a message broker, which lets the devices send custom payloads and then stores them in a queue until some consumer starts the reading process. In this case, it is configured to run as an **MQTT** broker.



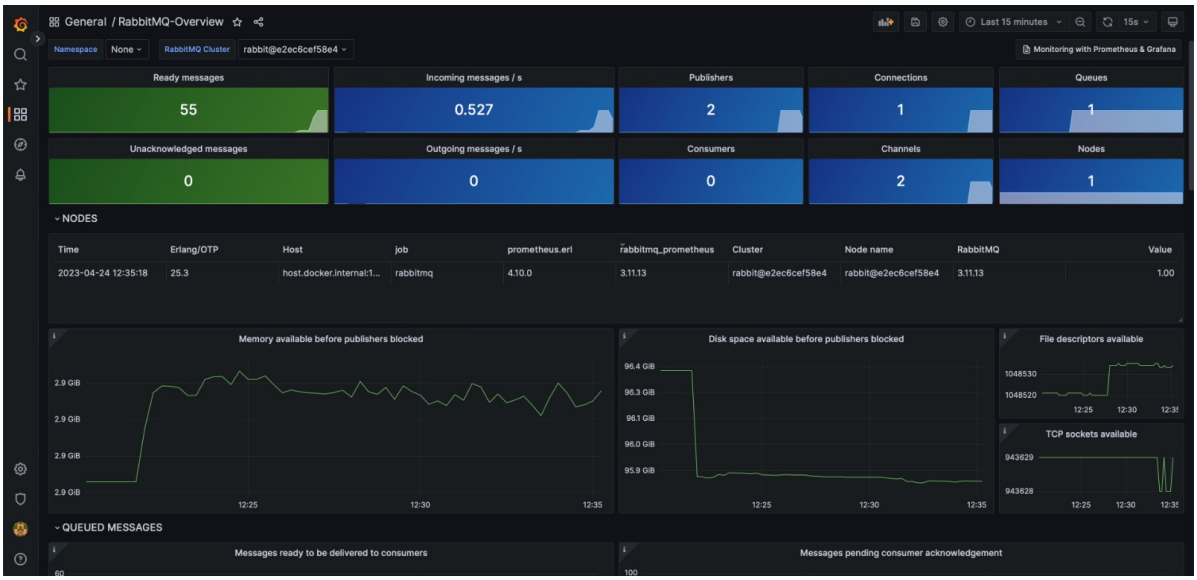
Prometheus (https://prometheus.io)

It is a solution which collects monitoring data from the devices, as for example the amount of occupied memory, the number of active processes, the cpu load and more. Prometheus as is exposes a simple dashboard which shows the current status of the monitoring, but another software layer is needed to show all the collected data.



Grafana (https://grafana.com/)

Grafana is a configurable dashboard to show metrics from a wide variety of services. In this project it is configured to show the monitoring metrics coming from Prometheus and RabbitMQ, showing for example the current connections, the queue messages rate and how many publishers and consumers are active.



These 3 software components (RabbitMQ, Prometheus and Grafana) run in docker containers, and everything is configured in a custom docker-compose.yml file, for simplifying their deployment.

```
version: '3.8'
services:
  broker:
    image: rabbitmq:3
    restart: always
    environment:
    - RABBITMQ_DEFAULT_USER=federico
    - RABBITMQ_DEFAULT_PASS=iotexamdemo
    command: "/bin/bash -c \"rabbitmq-plugins enable rabbitmq_mqtt; rabbitmq-plugins enable rabbitmq_prometheus; rabbitmq-
server\""
    ports:
      - 1883:1883
      - 15672:15672
      - 15675:15675
      - 15692:15692
      - 5672:5672
    volumes:
      - "./rabbitmq:/etc/rabbitmq"
      - "./data:/var/lib/rabbitmq/mnesia/"
  prometheus:
    image: prom/prometheus:v2.42.0
    restart: always
    ports:
    - 9090:9090
    volumes:
    - /Users/federico/Sviluppatore/IoT/prometheus:/etc/prometheus
  grafana:
    image: grafana/grafana
    restart: always
    ports:
      - 3000:3000
    volumes:
      - grafana-storage:/var/lib/grafana
volumes:
  grafana-storage:
    external: true
```

Having docker installed on the main system, the **docker compose up** command starts the execution of these services.

```
federico@Bertos-MacBook-Pro IoT % docker compose up
[+] Running 3/3
 ⠿ Container iot-grafana-1     Running
 ⠿ Container iot-prometheus-1  Running
 ⠿ Container iot-broker-1      Recreated
Attaching to iot-broker-1, iot-grafana-1, iot-prometheus-1
iot-broker-1      | Enabling plugins on node rabbit@1ec6583a8f76:
iot-broker-1      | rabbitmq_mqtt
iot-broker-1      | The following plugins have been configured:
iot-broker-1      |   rabbitmq_management
iot-broker-1      |   rabbitmq_management_agent
iot-broker-1      |   rabbitmq_mqtt
iot-broker-1      |   rabbitmq_prometheus
iot-broker-1      |   rabbitmq_web_dispatch
```

Queue consumer script

Another software component is the MQTT queue reader (consumer), implemented via a Python script (called **mqttReader.py**) which reads the queue and parses the results, then it prints a .csv file with the last 50 elements for every different board and model (if there is enough data). This will be used for processing the performance results.

Projects template

The projects for running the 3 stand alone models have the same implementation:

- **WiFi** initialization and connection
- **MQTT** queue setup and connection
- **TensorFlow** configuration for the specific model
- **Payload** construction and message publication

Also, every message sent to the queue has the same structure (which is a JSON payload), for example:

{"board": "esp32dev", "model": "wine", "result": 1, "iteration": 1, "microseconds": 120}

This is useful for the consumer script to be able to detect the specific device and model that has sent the message.

Implementation

ML models processing

First of all, for creating usable models to be loaded onto the devices, I used some utility scripts for exporting a .h header file for each one of them; these scripts use TensorFlow, numpy, sklearn and tinymlgen for training, testing and exporting the model. Then the generated file is put inside the **include** folder of each project.

Example

For example, for creating the wine_model.h file, I used Google Colab (https://colab.research.google.com) to have a ready to deploy Python environment, then I installed tinymlgen and ran the script.

```
[1]  pip install tinymlgen

Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting tinymlgen
  Downloading tinymlgen-0.2.tar.gz (1.5 kB)
  Preparing metadata (setup.py) ... done
Requirement already satisfied: tensorflow in /usr/local/lib/python3.10/dist-packages (from tinymlgen) (2.12.0)
Collecting hexdump (from tinymlgen)
  Downloading hexdump-3.3.zip (12 kB)
  Preparing metadata (setup.py) ... done
Requirement already satisfied: absl-py>=1.0.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow->tinymlgen) (1.
```

I then imported the dependencies needed for the training and exporting flow:

```
import numpy as np
from tensorflow.keras import Sequential, layers
from tensorflow.keras.utils import to_categorical
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split
from tinymlgen import port
```

After this setup phase, the remaining part of the script can be executed and the resulting model is printed in the console.

```
# load and split dataset into train, validation, test
X, y = load_wine(return_X_y=True)
y = to_categorical(y)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
X_train, X_valid, y_train, y_valid = train_test_split(X_train, y_train, test_size=0.3)
```

```
# export to file
with open('wine_model.h', 'w', encoding='utf-8') as file:
    print(port(nn, variable_name='wine_model', pretty_print=True, optimize=False))
```

```
input_dim (13,)
output_dim 3
2/2 [==============================] - 0s 10ms/step - loss: 0.5056 - accuracy: 0.8333
Accuracy: 0.8
WARNING:absl:Found untraced functions such as _update_step_xla while saving (showing 1 of 1). These functions will not be directly callable after loading.

#ifdef __has_attribute
#define HAVE_ATTRIBUTE(x) __has_attribute(x)
#else
#define HAVE_ATTRIBUTE(x) 0
#endif
#if HAVE_ATTRIBUTE(aligned) || (defined(__GNUC__) && !defined(__clang__))
#define DATA_ALIGN_ATTRIBUTE __attribute__((aligned(4)))
#else
#define DATA_ALIGN_ATTRIBUTE
#endif

const unsigned char wine_model[] DATA_ALIGN_ATTRIBUTE = {
        0x1c, 0x00, 0x00, 0x00, 0x54, 0x46, 0x4c, 0x33, 0x14, 0x00, 0x20, 0x00,
        0x1c, 0x00, 0x18, 0x00, 0x14, 0x00, 0x10, 0x00, 0x0c, 0x00, 0x00, 0x00,
```

Note: with the digits model I had to use the already provided file since the script as is is not runnable.

Running the models

Every model has been ran for 50 iterations, with every result sent to the MQTT queue; the consumer script processed the data from the queue, creating the corresponding model file containing the performance results.

This is an example of the script running while the sin model on the ESP8266 is sending data to the queue:

```
sin(0.31) = 0.31          predicted: 0.29
Evaluation time: 472 microseconds
Message sent with packetId: 42
sin(0.63) = 0.59          predicted: 0.57
Evaluation time: 439 microseconds
Message sent with packetId: 43
sin(0.94) = 0.81          predicted: 0.81
```

```
Stored esp8266_sin.csv
Processed 140 elements
Processed 150 elements
Processed 160 elements
Processed 170 elements
Processed 180 elements
Processed 190 elements
Processed 200 elements
Processed 210 elements
```

Performance results

From the generated files, I have calculated an average running time based on the 50 iterations of every model. All these results are in the “Processed results” folder.

ESP8266

Digits

```
RAM:   [========= ]  85.8% (used 70300 bytes from 81920 bytes)
Flash: [====      ]  41.5% (used 433557 bytes from 1044464 bytes)
```

Performance average: **20802,42** microseconds.

Sine

```
RAM:   [======    ]  59.9% (used 49040 bytes from 81920 bytes)
Flash: [====      ]  39.7% (used 414169 bytes from 1044464 bytes)
```

Performance average: **489,64** microseconds.

Wine

```
RAM:   [========  ]  84.7% (used 69368 bytes from 81920 bytes)
Flash: [====      ]  40.9% (used 427681 bytes from 1044464 bytes)
```

Performance average: **6663,76** microseconds.

ESP32 Wemos

Digits

```
RAM:   [==        ]  18.1% (used 59156 bytes from 327680 bytes)
Flash: [=======   ]  69.7% (used 913321 bytes from 1310720 bytes)
```

Performance average: **3242,96** microseconds. This model runs 6 times faster on this board compared to the ESP8266 board.

Sine

```
RAM:   [==        ]  16.1% (used 52716 bytes from 327680 bytes)
Flash: [=======   ]  68.2% (used 894361 bytes from 1310720 bytes)
```

Performance average: **43,96** microseconds. This model runs 11 times faster on this board compared to the ESP8266 board.

Wine

```
RAM:   [==        ]  20.8% (used 68108 bytes from 327680 bytes)
Flash: [=======   ]  69.2% (used 907629 bytes from 1310720 bytes)
```

Performance average: **462,62** microseconds. This model runs 14 times faster on this board compared to the ESP8266 board.

ESP32 Dev

Digits

```
RAM:   [==        ]  18.1% (used 59156 bytes from 327680 bytes)
Flash: [========  ]  69.7% (used 913381 bytes from 1310720 bytes)
```

Performance average: **4051,32** microseconds. This result is 1000 microseconds slower than the Wemos board.

Sine

```
RAM:   [==        ]  16.1% (used 52716 bytes from 327680 bytes)
Flash: [=======   ]  68.2% (used 894397 bytes from 1310720 bytes)
```

Performance average: **89,3** microseconds. Also in this case, the average is almost double than the result from the Wemos board.

Wine

```
RAM:   [==        ]  20.8% (used 68108 bytes from 327680 bytes)
Flash: [========  ]  69.2% (used 907473 bytes from 1310720 bytes)
```

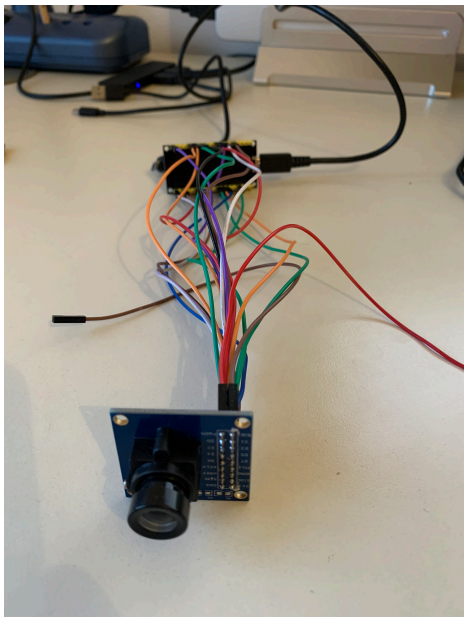
Performance average: **764,58** microseconds. Compared to the Wemos board, the average time is almost double.

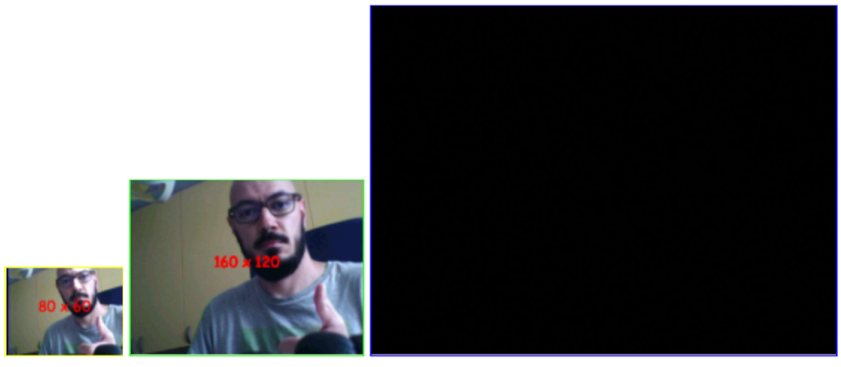
Person detection

Using an **OV7670** camera, I implemented a person detection model, running on the ESP32Dev board. This model is included in the EloquentTinyML library.

For this implementation, I cannot use my project template (WiFi + MQTT) because the board runs out of RAM, so for simplifying things I ran the model 50 times and collected the data manually.

First of all, I wanted to test the camera connection, using the OV7670 drivers and a demo project, called “CameraDemo", this is the result:





QQQ-VGA QQ-VGA Q-VGA

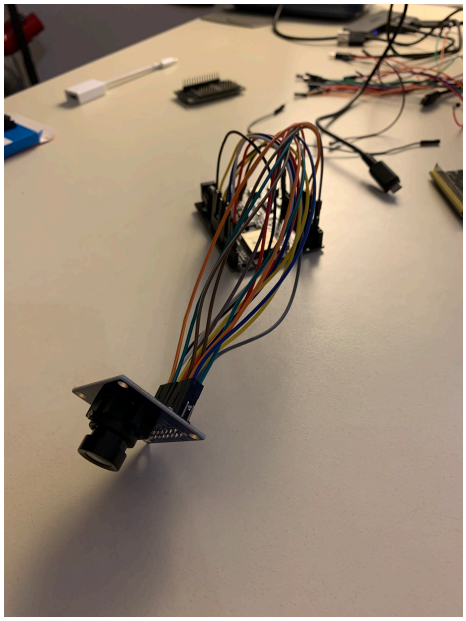
Connected to ws://192.168.4.1:81/	
AP IP	WiFi IP
192.168.4.1	0.0.0.0

Then I started implementing the Person detection model in another project called “Person”, with code logic to calculate the average time at every iteration:

```
No person detected
        > It took 4624ms to detect
        > Person score: 139
        > Not person score: 192
Elapsed time: 4624
Average time on 50 iterations: 4623
```

The result is that basically every iteration has the same running time, resulting in a average of **4623** milliseconds (or 4 seconds and a half). This is a very different runtime compared to the other models, for which we used microseconds.

I also tried connecting the other ESP32 board (Wemos) to the camera, but the schematics I have are not useful for a correct connection, and moreover the board itself seems to have some missing pins or they are labeled differently; trying to connect to other pins and reconfiguring them via code has not been successful.



Conclusions

It’s been really interesting to see how capable these small and cheap devices are, and some have even multiple CPUs (ESP32 is a dual core); in a real world example it would be possible to run some lightweight ML models with reasonable performance.