REPORT Marta Caffagnini

# PERFORMANCE EVALUATION OF TinyML ON DIFFERENT DEVICES

**1. SCOPE**

The scope of this project is the evaluation of the performance of different TinyML models running on different devices.

TinyML is a field of machine learning that explores the types of models runnable on small, low-powered devices like microcontrollers.

The devices are 3 microcontroller boards (Wemos, ESP32 and ESP8266).

The microcontrollers were programmed using C++ programming language and PlatformIO installed on Visual Studio Code.

4 TinyML models with increasing complexity were analyzed.

The performance evaluation consists in testing the execution time and memory footprint of these models on different devices.

**2. TOOLS**

2.1 Devices

- Wemos D1 R32 is ESP32 based. The ESP32 chip on this board has 320 KB RAM and 1 MB Flash. ESP32 has a dual core processor. ESP32 has a clock speed of 240MHz. ESP32 has Wi-fi and Bluetooth.
- ESP32 with 320 KB RAM, 1 MB Flash and a dual core processor.
- ESP8266 with 80 KB RAM, 4 MB Flash and a single core processor. ESP8266 has a clock speed of 80 MHz and has only Wi-fi.

2.3 TensorFlow Lite

TensorFlow Lite is a mobile library for deploying models on mobile, microcontrollers and other edge devices. How it works:

- Pick a model
- Convert a TensorFlow model into a compressed flat buffer with the TensorFlow Lite Converter.
- Take the compressed .tflite file and load it into a mobile or embedded device
- In this project the .tflite file is converted into a C++ array in order to be used in the C++ main sketch.

2.4 EloquentTinyML

EloquentTinyML is an Arduino library to make TensorFlow Lite for Microcontrollers neural networks more accessible and easy to use.

### 2.5 everywhereml

A Python package to train Machine Learning models that run (almost) everywhere. This package was used to port the classifier from Python to C++ with a single line of code.

### 2.6 tinymlgen

A Python package to export TensorFlow Lite neural networks to C++ for microcontrollers.

### 2.7 PlatformIO

PlatformIO is a cross-platform, cross-architecture, multi-framework IDE tool for embedded systems and embedded applications. In this platform is available a tool to inspect memory which was used for the analysis of the models running on different devices.

### 2.8 Inspect tool

A tool available on Platform IO.
The tab displays general information about used hardware and a graphical representation of the percentage of busy memory in RAM and in Flash memory.
Its report differs from the regular information reported after each build step. Project Inspector takes into account the memory section which was allocated for the stack and heap to check if there is enough free memory left for the stack and heap to fit into the RAM.

The microcontroller has 2 types of memory:

- RAM (Random-Access-Memory) is ‘volatile’. RAM memory is used to store the variables during program execution and offers fast read access times.
- Flash memory is ‘non-volatile’. Flash memory is used to store the program.

## 3. TinyML MODELS

The selected models for this evaluation have different degrees of complexity in terms of machine learning algorithm and input dimension.

- The first model is **Iris Flower Classification** and its classifier is Random Forest. The model is trained using Iris dataset which describes 3 species of the Iris flower in terms of petal width, petal height, sepal length, sepal height. Given these 4 features, the model is able to predict the name of the flower.
- The second TinyML model is a neural network which, given a value, computes its **sine and cosine**. Neural network is more complex than Random Forest, but in this case the input complexity is low. This neural network has 2 layers of 16 neurons.
- The third TinyML model is a neural network which, given an image, **recognizes hand-written digits** (from 0 to 9). The input consists of an 8x8 pixel image. The model is a convolutional neural network with 3 layers.
- The fourth model was selected in order to test the limit of the devices. The model’s task is the same as the previous one but in this case the input image is bigger.
  The neural network recognizes hand-written digits and is trained with **MNIST** dataset.
  The input is an image of 28x28 pixels.
  The model is a convolutional neural network with 4 layers.

After the selection, the trained models (all written in Python code) had to be converted in C++.

For this step everywhereml (for Random Forest) or tinymlgen (for neural networks) packages were used to port the Python code to C++ code.

The models were saved in header files, then included in PlatformIO projects.

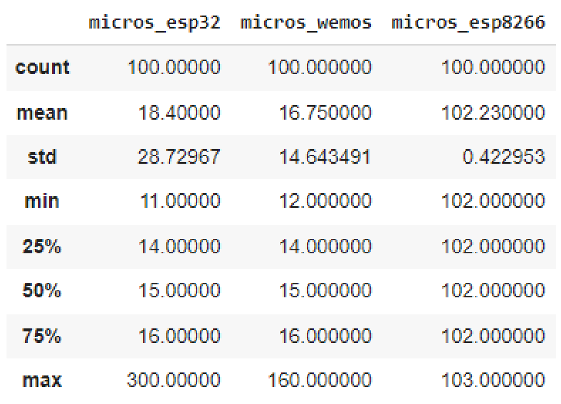
PlatformIO projects were created to test every model on every device.

Every exported code was called from the main sketch of the matching project. In the main file the library EloquentTinyML was used to make TensorFlow Lite more easy to use.

## 4. TIME ANALYSIS

In every main sketch the loop() function was designed to create a random instance in every execution and feed it as input to the model. In every project the function was runned 100 times to collect all execution times. It wasn't the time to execute one iteration of the loop() function but only the time to classify one instance.

- **Iris Flower Classification**

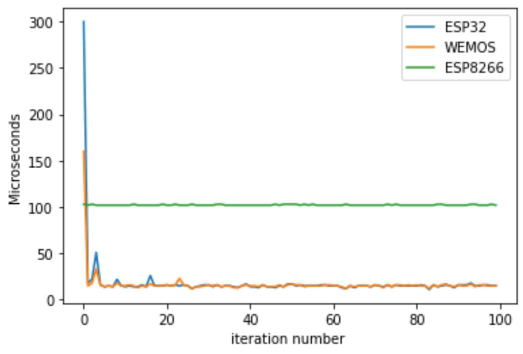


| | micros_esp32 | micros_wemos | micros_esp8266 |
|---|---|---|---|
| **count** | 100.00000 | 100.000000 | 100.000000 |
| **mean** | 18.40000 | 16.750000 | 102.230000 |
| **std** | 28.72967 | 14.643491 | 0.422953 |
| **min** | 11.00000 | 12.000000 | 102.000000 |
| **25%** | 14.00000 | 14.000000 | 102.000000 |
| **50%** | 15.00000 | 15.000000 | 102.000000 |
| **75%** | 16.00000 | 16.000000 | 102.000000 |
| **max** | 300.00000 | 160.000000 | 103.000000 |

This table contains data related to the performance of the microcontrollers classifying a random instance. The data was collected through 100 iterations and each iteration provided the execution time in microseconds for each microcontroller.

Based on the statistics, Wemos generally performs better than the ESP32 and ESP8266, as it has the lowest mean execution time. ESP8266 has the smallest standard deviation because the function to measure execution time was micros() which is less precise than esp_timer (used with the other boards). To collect reliable data with micros(), It was collected the time it takes to classify 500 instances and then the time obtained was divided by the number of instances classified.

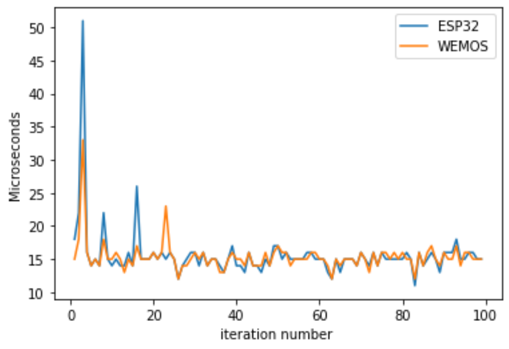
The ESP8266 board has the smallest standard deviation in execution time because its execution times are average values. The function used to measure time was micros() which is less precise than the esp_timer function used with other boards. Reliable data was collected by measuring the time it took to classify 500 instances using micros() and then dividing the total time by the number of instances classified.



This plot shows the variation of the execution time (in microseconds) of the boards over 100 iterations. The execution time is only the amount of time it takes to classify one instance. The x-axis represents the iteration number, while the y-axis represents the execution time in microseconds.

The first iteration is the one that took more time than the others. This could happen because maybe the microcontroller is in a low-power state during the first iteration, and may require some time to go to full power.

ESP8266 is slower than the other 2 boards. Wemos and ESP32 look the same so ESP8266 was removed from the plot to better analyze the difference between Wemos and ESP32. Also the first iteration was removed because it's an outlier that skews the scale of the y-axis, making it difficult to discern the differences.



This plot shows two lines, one for ESP32 and the other for Wemos, both lines show similar trends, but the line for ESP32 is generally above the line for Wemos, indicating that ESP32 is slower than Wemos in classifying one instance.

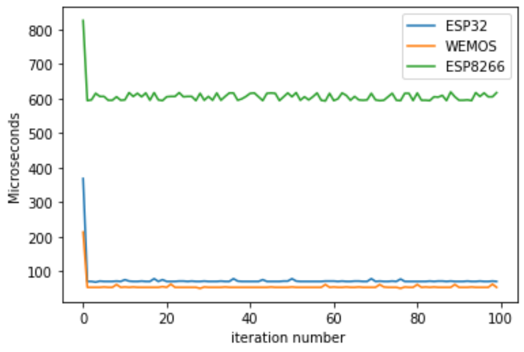
- **Sine Cosine model**

| | micros_esp32 | micros_wemos | micros_esp8266 |
|---|---|---|---|
| **count** | 100.000000 | 100.000000 | 100.000000 |
| **mean** | 73.780000 | 55.290000 | 606.490000 |
| **std** | 29.778163 | 16.078906 | 23.841012 |
| **min** | 68.000000 | 50.000000 | 593.000000 |
| **25%** | 70.000000 | 53.000000 | 595.000000 |
| **50%** | 70.000000 | 53.000000 | 605.000000 |
| **75%** | 71.000000 | 53.000000 | 615.000000 |
| **max** | 368.000000 | 213.000000 | 826.000000 |

This table contains data related to the performance of the microcontrollers predicting sine and cosine of a random instance. The data was collected through 100 iterations and each iteration provided the execution time in microseconds for each microcontroller.

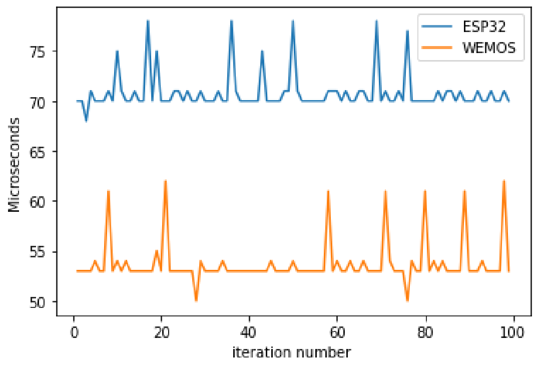
These statistics include the count, mean, standard deviation, minimum, 25th percentile, median (50th percentile), 75th percentile, and maximum values.

From the statistics, the mean execution time is highest for the ESP8266 device, followed by the ESP32 and Wemos devices. The standard deviation is highest for the ESP32 device, this means that there is more variability in the measurements for this device. The minimum execution time is lowest for the Wemos device, while the maximum execution time is highest for the ESP8266 device. There are some outliers, especially for the maximum values. Overall, these statistics suggest that the ESP8266 device takes the longest time to compute, while the Wemos device is the fastest.



This plot shows the performance of the three boards (ESP32, Wemos, and ESP8266) in terms of the time taken to predict the sine and cosine for 100 iterations. The x-axis represents the iteration number, while the y-axis represents the time in microseconds.

It appears that the ESP32 and Wemos have similar performance, while the ESP8266 takes much longer to predict the sine and cosine.



In this plot the ESP8266 line was removed to better analyze the difference between Wemos and ESP32. Also the first iteration was removed from the plot to better visualize the behavior of the boards after the initial warm-up period.

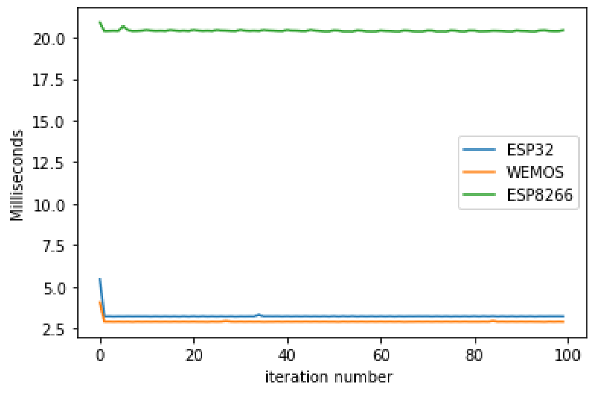
Now It’s clear the difference. It appears that Wemos is faster than ESP32.

- **Hand-written digits Recognition (8x8 pixel)**

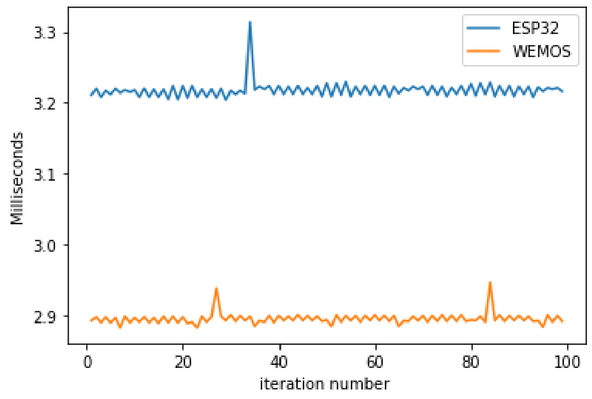
| | millis_esp32 | millis_wemos | millis_esp8266 |
|---|---|---|---|
| **count** | 100.000000 | 100.000000 | 100.000000 |
| **mean** | 3.239930 | 2.906090 | 20.424040 |
| **std** | 0.222641 | 0.114633 | 0.061652 |
| **min** | 3.204000 | 2.882000 | 20.376000 |
| **25%** | 3.211000 | 2.891000 | 20.396000 |
| **50%** | 3.218000 | 2.892000 | 20.412000 |
| **75%** | 3.223000 | 2.899000 | 20.436500 |
| **max** | 5.441000 | 4.038000 | 20.901000 |

The table shows the results of measuring the execution time in milliseconds of image classification using three different microcontrollers: ESP32, Wemos, and ESP8266. The measurements were taken 100 times for each microcontroller.

ESP8266 has the highest mean execution time and the Wemos has the lowest mean execution time, while the ESP32 is in the middle.



The plot shows the execution time of the three different microcontrollers (ESP32, Wemos, and ESP8266). It appears that the execution time of all three microcontrollers is relatively stable over the course of the iterations. The ESP8266 has the highest execution time, followed by the ESP32 and then the Wemos. Overall, the plot suggests that the Wemos is generally the fastest of the three microcontrollers, while the ESP8266 is the slowest.



In this plot ESP8266 was removed to better analyze the difference between Wemos and ESP32. Also the first iteration was removed from the plot to better visualize the behavior of the boards after the initial warm-up period.

Now It’s clear the difference between ESP32 and Wemos.

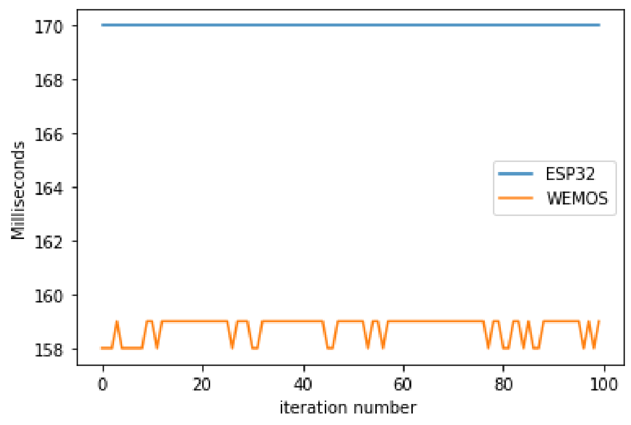
The execution time of both microcontrollers fluctuates around a certain range of values, with occasional spikes. Overall, it appears that the Wemos microcontroller takes less time to execute the operation compared to the ESP32 microcontroller, as the Wemos line is lower than the ESP32 line.

- **Hand-written digits Recognition (28x28 pixel) MNIST dataset**

| | milli_esp32 | milli_wemos |
|---|---|---|
| **count** | 100.0 | 100.000000 |
| **mean** | 170.0 | 158.760000 |
| **std** | 0.0 | 0.429235 |
| **min** | 170.0 | 158.000000 |
| **25%** | 170.0 | 159.000000 |
| **50%** | 170.0 | 159.000000 |
| **75%** | 170.0 | 159.000000 |
| **max** | 170.0 | 159.000000 |

The ESP8266 is not included in the table because it was unable to run the program due to insufficient RAM capacity.

ESP32 device takes around 170 milliseconds per iteration, while the Wemos device takes around 158-159 milliseconds per iteration. Wemos has the best performance as before.



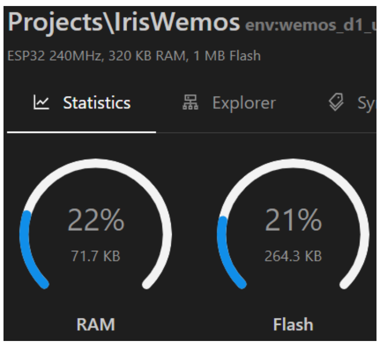
The plot shows the execution time in milliseconds for ESP32 and Wemos running the same model. The x-axis represents the iteration number, while the y-axis represents the time in milliseconds.

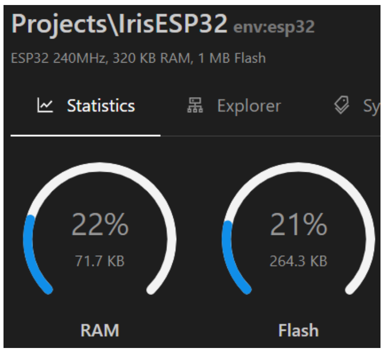
From the plot, we can see that Wemos has a lower execution time compared to ESP32. The execution time for Wemos seems to vary over the iterations in the range from 158 to 159 milliseconds.

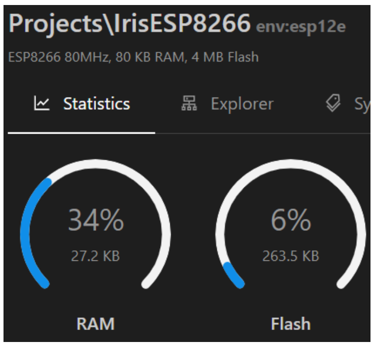
## 5. MEMORY FOOTPRINT ANALYSIS

To analyze the utilization of the available memory on different devices running the same program, the tool Inspect available on Platform IO was used.

- **Iris Flower Classification**

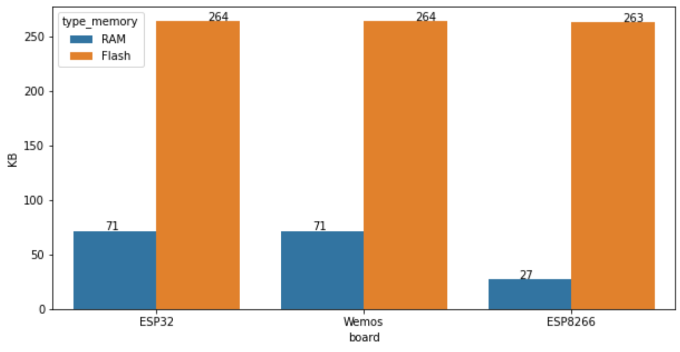

Projects\IrisWemos env:wemos_d1_
ESP32 240MHz, 320 KB RAM, 1 MB Flash
Statistics
Explorer
Sy
22%
71.7 KB
RAM
21%
264.3 KB
Flash


Projects\IrisESP32 env:esp32
ESP32 240MHz, 320 KB RAM, 1 MB Flash
Statistics
Explorer
Sy
22%
71.7 KB
RAM
21%
264.3 KB
Flash


Projects\IrisESP8266 env:esp12e
ESP8266 80MHz, 80 KB RAM, 4 MB Flash
Statistics
Explorer
Sy
34%
27.2 KB
RAM
6%
263.5 KB
Flash

These are the dashboards showing the percentage of busy memory (RAM and Flash) in the 3 boards. ESP32 and Wemos use the same amount of memory and have the same percentages because the available memory is the same. The ESP8266 board has a higher percentage of busy RAM compared to the ESP32 and Wemos, and less percentage of busy Flash.

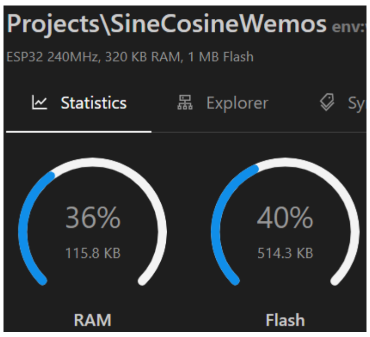
The Flash memory type is generally less busy than the RAM type.

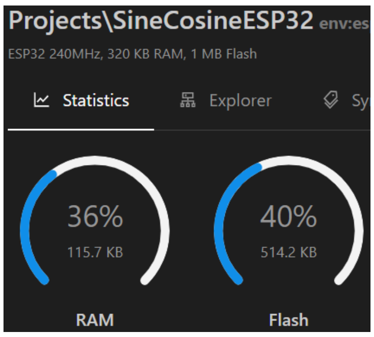


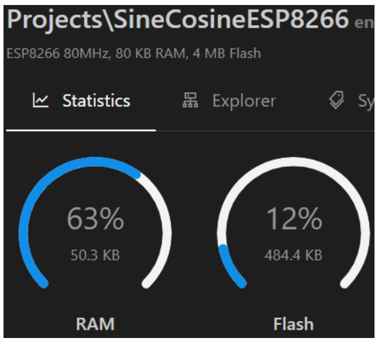
This plot shows a comparison of the amount of memory in kilobytes (KB) for different types of memory (RAM and Flash) on the microcontroller boards (ESP32, Wemos, and ESP8266). The x-axis represents the board type, and the y-axis represents the amount of memory in KB.

The plot shows that the Wemos board has the same amount of memory as the ESP32 for both RAM and Flash. The ESP8266 has the lowest amount of memory for both types, probably because of the architectural differences between ESP32 and ESP8266.

- **Sine Cosine model**

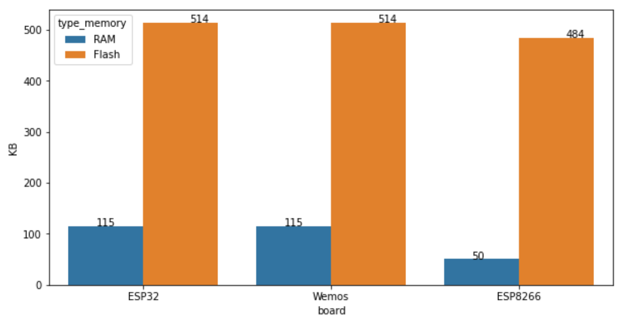

Projects\SineCosineWemos env:
ESP32 240MHz, 320 KB RAM, 1 MB Flash
Statistics
Explorer
Sy
36%
115.8 KB
RAM
40%
514.3 KB
Flash


Projects\SineCosineESP32 env:es
ESP32 240MHz, 320 KB RAM, 1 MB Flash
Statistics
Explorer
Sy
36%
115.7 KB
RAM
40%
514.2 KB
Flash


Projects\SineCosineESP8266 en
ESP8266 80MHz, 80 KB RAM, 4 MB Flash
Statistics
Explorer
Sy
63%
50.3 KB
RAM
12%
484.4 KB
Flash

These are the dashboards provided by the Inspect tool showing the percentage of busy memory (RAM and Flash) on the 3 boards. ESP32 and Wemos used the same amount of memory and have the same percentages because also the memory available is the same. ESP8266 has less KB of RAM available and more MB of Flash memory.

ESP8266 has the highest percentage of busy memory for RAM and the lowest percentage for Flash.

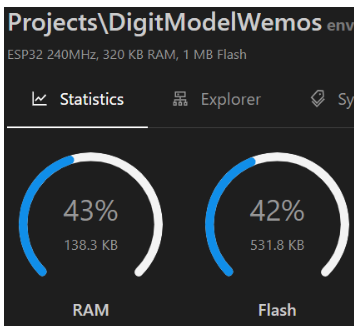


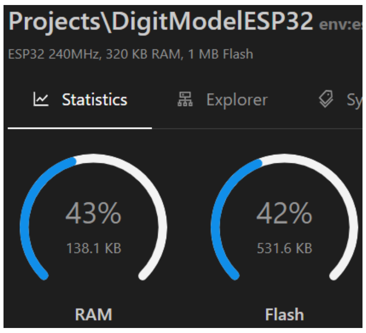
This plot shows a bar chart that compares the memory footprint on the microcontroller boards. Each bar represents a specific board-type memory combination, and the height of the bar represents the corresponding memory footprint in KB.

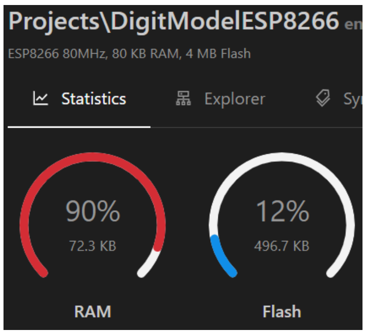
For both types of memory, the ESP32 and Wemos boards have the highest memory footprint.

ESP8266 used less RAM and Flash memory than the others.

- **Hand-written digits Recognition (8x8 pixel)**

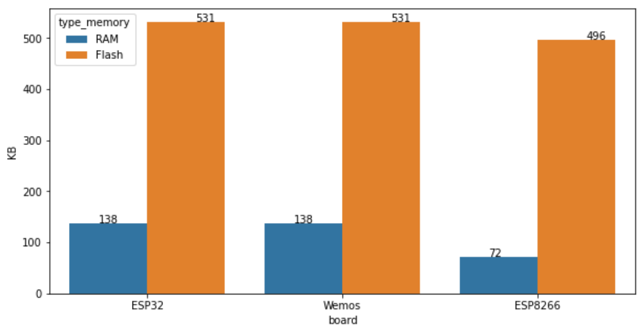

Projects\DigitModelWemos env
ESP32 240MHz, 320 KB RAM, 1 MB Flash
Statistics
Explorer
Sy
43%
138.3 KB
RAM
42%
531.8 KB
Flash


Projects\DigitModelESP32 env:e
ESP32 240MHz, 320 KB RAM, 1 MB Flash
Statistics
Explorer
Sy
43%
138.1 KB
RAM
42%
531.6 KB
Flash


Projects\DigitModelESP8266 en
ESP8266 80MHz, 80 KB RAM, 4 MB Flash
Statistics
Explorer
Sy
90%
72.3 KB
RAM
12%
496.7 KB
Flash

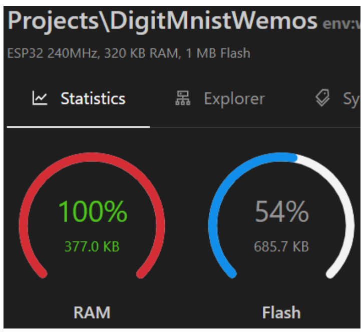
These are the dashboards provided by the Inspect tool showing the percentage of busy memory (RAM and Flash) on the 3 boards.

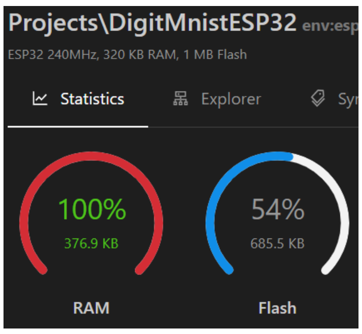
The ESP8266 board has a significantly higher percentage of busy RAM memory compared to its Flash memory. The ESP8266 board has a relatively limited amount of available RAM memory, and this could affect its performance. ESP8266 uses less RAM, but it has less available memory so it’s running out of RAM.



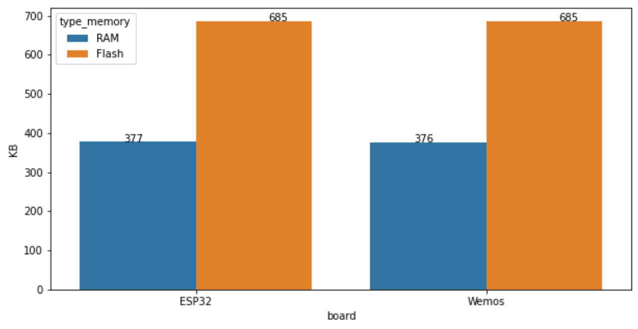
The plot represents the memory footprint (in KB) of different boards and types of memory. The plot shows that the ESP32 and Wemos boards have the same memory footprint for both RAM and Flash memory types. However, the ESP8266 board has a smaller memory footprint than the ESP32 and Wemos boards for both memory types.

- **Hand-written digits Recognition (28x28 pixel) MNIST dataset**


Projects\DigitMnistWemos env:
ESP32 240MHz, 320 KB RAM, 1 MB Flash
Statistics
Explorer
Sy
100%
377.0 KB
RAM
54%
685.7 KB
Flash


Projects\DigitMnistESP32 env:esp
ESP32 240MHz, 320 KB RAM, 1 MB Flash
Statistics
Explorer
Sy
100%
376.9 KB
RAM
54%
685.5 KB
Flash

Both boards have 100% busy memory for RAM, and the same percentage of busy memory for Flash.



This plot shows the memory footprint of two different microcontroller boards (ESP32 and Wemos) for two different types of memory (RAM and Flash). Wemos and ESP32 have quite the same memory footprint as in the previous models. Both RAM are full.

## 6. CONCLUSION

In conclusion, the analyses conducted on different models with varying degrees of complexity have provided information about their performance. It is clear that the ESP8266 board has less RAM memory capacity and is slower compared to the other two boards. On the other hand, the ESP32 and Wemos boards have similar memory capacities and busy memory when a model is running. The Wemos board performs better than the ESP32.