TinyML Performance Evaluation on Heterogeneous Devices

Gabriele Savoia

Abstract

TinyML nowaday represents an important and active research area in the field of machine learning applied to IoT systems as it cover the so called *on-device inference* thanks to which is possibile to deploy an ML/DL model on a low-power and low-footprint microcontroller.

This class-project aims to first describe the classic workflow in order to create and deploy a Tensorflow Lite model on a specific device. Then, three main model architectures (fully-connected, CNN, LSTM) each with three level of complexities (small, medium, large) are created and deployed in different devices (ESP-32, Arduino Nano 33 BLE Sense, ESP-8266) in order to analyze and compare them in terms of inference time and memory footprint.

1 Tensorflow Lite model creation and deploy

In this project is considered the *Tensorflow Lite Micro* (TLM) framework as it is one of the most popular software for TinyML.

It is based on Tensorflow [TF], an open-source software library originally written by Google for simplify variuos tasks in AI field like neural network models creation, train and inference.

Like TLM's official website [TLMa] report, Tensorflow Lite for Microcontroller (written in C++11) allows to run Tensorflow model on microcontrollers with a very low memory impact (only a few kilobytes). In fact, the core runtime fits in 16 KB and does not require operating system support, standard C or C++ library or dynamic memory allocation.

The following sections describe the two main steps in order to use this framework in real devices: the *creation phase* and *deployment phase*.

1.1 Model creation

The creation phase consists in the model structure definition and train followed by its conversion to a device-readable and more compact version. In particular, the main steps are:

1. create a standard Tensorflow model and train it on a specific dataset: for this project Google Colab python notebook is used. Note that there are some constraints in terms of model size (due to device RAM and storage limitation) but also in terms of operations used (as reported in [TLMc] not all Tensorflow Operations are supported in TLM so not all model architectures are available). Model size can be directly controlled by the choise of layers dimensions: fewer and smaller layers lead to a lighter model but with a higher probability of underfitting;

2. use TensorFlow Lite Converter [TLC] to convert the standard Tensorflow model (created in the step above) to a (smaller) Tensorflow Lite one. The latter is represented as a compact and efficient Flatbuffer [1], and is usually saved in a *.tflite* file. Optionally it is also possible to apply a *Post-training quantization* [Ful] which can allow to obtain the smallest possible model size;

3. convert Tensorflow Lite model to a C array of bytes which will then be stored in a C++ header file (with *.h* extension) through the use of standard tools such as Unix *xxd*.

1.2 Model deploy

In order to deploy the model (saved as C++ header file), Tensorflow Lite Micro makes available a C++ library [TLMb] that can be used to handle Tensorflow lite model directly in the C++ device code and, as reported in chapter 5 in [Tin], the main steps can be summarized in the following:

1. import library and model: through Arduino IDE [Arda] or PlatformIO [Pla], import the C++ Tensorflow Lite Micro library and the C++ header file containing the Tensorflow Lite model array of byte we want to use;

[1]Flatbuffer: `https://flatbuffers.dev`

2. create the model: pass model data into a specific function and obtain the pointer to the model structure which hold data and informations about it;
3. define operation resolver: this is a structure that knows all the operations available and provide them to the interpreter that runs the model (see point 5). In order to reduce memory usage, it is possible to not consider all the possible operations but only those needed by the model;
4. define tensor arena: create an area of memory with a certain size dedicated for store input, output and intermediate tensors required by the model. Its dimension depends on the model structure and must be setted large enough but paying attention to not overcome device RAM limitation. Usually this value is chosen through trial and error;
5. interpreter creation: through a class instantiation, is possible to create an interpreter that can run the specified model on custom data (ex. cominig from sensors) and make inference;
6. run inference: through a specific interpreter function, run inference and see the result in the output vector.

In this project are used both PlatformIO (for ESP-32 and ESP-8266) and Arduino IDE (for Arduino Nano 33 BLE Sense). Through the use of the EloquentTinyML library [Elo], there is no need to code all the steps listed above and with only its two functions (*begin()* and *predict()*) is possible to make inference on custom input data on both Arduino and ESP. If this library is very usefull for fully-connected and CNN models, it is not for LSTM model. This is because it is not up to date, so the newly-introduced operations for LSTM layer support are absent. For this reason, in order to deploy LSTM model, are coded all the steps listed above with the help of ArduinoTensorFlowLite [Ardd] (for Arduino) and TensorFlowLiteESP32 [Ten] (for ESP) libraries that refers to a newly version of Tensorflow Lite Micro.

2 Devices

In this project are considered three different devices: ESP-32, Arduino Nano 33 BLE Sense and ESP-8266. In the following are reported their main characteristics in terms of CPU specifications and memory capacity:

- *ESP-32*: [ESPa] is characterized by a 32 bit Xtensa dual core CPU with 240 Mhz clock speed. It has 1 MB of Flash memory and 320 kB of RAM. It also integrates a 2.4Ghz WiFi and Bluetooth chip making this device versatile and adapt for Iot applications;
- *Arduino Nano 33 BLE Sense*: like official documentation report ([Ardb]), Arduino Nano 33 BLE Sense is based on nRF52840 microcontroller [Ardc] with ARM Cortex-M4 32 bit as CPU with 64 Mhz of clock speed. It has 1 MB of Flash memory and 256 kB of RAM. It has a set of integrated sensors of different type from humidity or temperature to proximity and motion detection;
- *ESP-8266*: [ESPb] is the predecessor of ESP-32 and consists on a 32 bit Xtensa single core CPU with 80 Mhz clock speed. It has 1 MB of Flash memory and 80 kB of RAM. Unlike ESP-32, it has only WiFi.

3 Model Evaluation

In this section are reported and compared the results in terms of time inference and memory footprint for each deep learning model with different degree of complexity and in relation to each devices.

In particular, the three different model architectures (fully-connected, CNN, LSTM) are contextualised in three different use cases respectively: *room-occupancy detection*, *drowsiness detection* and *time series forecast*. For each of these are reported three variants based on model complexity: small, medium and large.

3.1 Room Occupancy Detection

The first model type considered is a fully-connected neural network that can understand the presence or absence of persons in a room based on 5 specific values coming from sensors (temperature, humidity, light, CO2, humidity ratio) using a specific dataset [Roo] for the train part. It is an interesting use case since there is no need of a camera and can be implemented using also simple and light predicitve models.

For this binary classification problem are created three different fully-connceted neural network with increasing complexity:

- small: composed by 1 dense layer with 20 units;

- medium: composed by 2 dense layers with 20 and 50 units each;
- large: composed by 3 dense layers with 20, 50, 80 units each.

Since this dataset is quite simple, all these three networks gain a 98% of accuracy and, as table 1 reports, they differ significantly in terms of model size.

Model (fully-connected)	Evaluation (accuracy)	Size
Small (20 units)	98%	2,2 kB
Medium (20, 50 units)	98%	7,0 kB
Large (20, 50, 80 units)	98%	23,8 kB

Table 1: Models evaluation and size for room-occupnacy detection.

Given the three types of model described above (small, medium, large), the goal is to compare performance (in terms of inference time and memory footprint) on three different devices: ESP-32, Arduino Nano 33 BLE Sense and ESP-8266.

3.1.1 Inference time comparison

For each model and for each device are collected the times inference (in microseconds) of 200 predictions with the goal to understand the impact of model complexity and the differences between the devices. In particular the inference time measurement takes into account the three foundamental inference phases: input model definition, inference phase and output model retrieval phase.

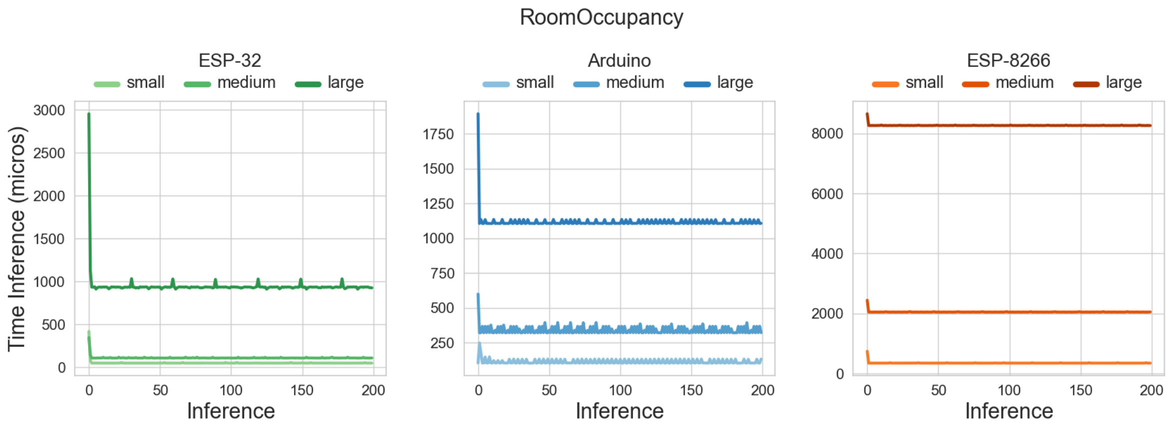

Figure 1: Time (in micros) for each of the 200 inferences (for room-occupancy problem) by each device in relation to different model size.

In figure 1 are reported all the predictions and is possibile to see how the increase of model complexity (in this case the number of hidden dense layers and their units) leads to a growth of inference time in each device. In the graphs are also visible significant peaks regard the first predictions (probably due to no initial model warmup[2] and because devices require some micros after the start for reach full-performance) but after that the values stabilize.

To better compare the differences between devices, in figure 2 are reported the mean time inference (and the respective standard deviation (the vertical black line)) for each model complexity in relation to the three devices. If is possible to see that ESP-32 has the lowest mean inference time while ESP-8266 the highest, can be also noted that ESP-32 has the highest standard deviation compared the others. Although Arduino is slower than ESP-32 it still turns out to be more performant than ESP-8266 (in large model is almost 7,5x).

3.1.2 Memory footprint comparison

When using Tensorflow Lite on microcontroller, the tensor arena size parameter must be specified. As explained in 1.2, its value is crucial as it represents the memory area dimension in which input, output and intermediate model's tensors are

[2]Initial model warmup: https://stackoverflow.com/questions/60841318/tensorflow-js-first-prediction-delay

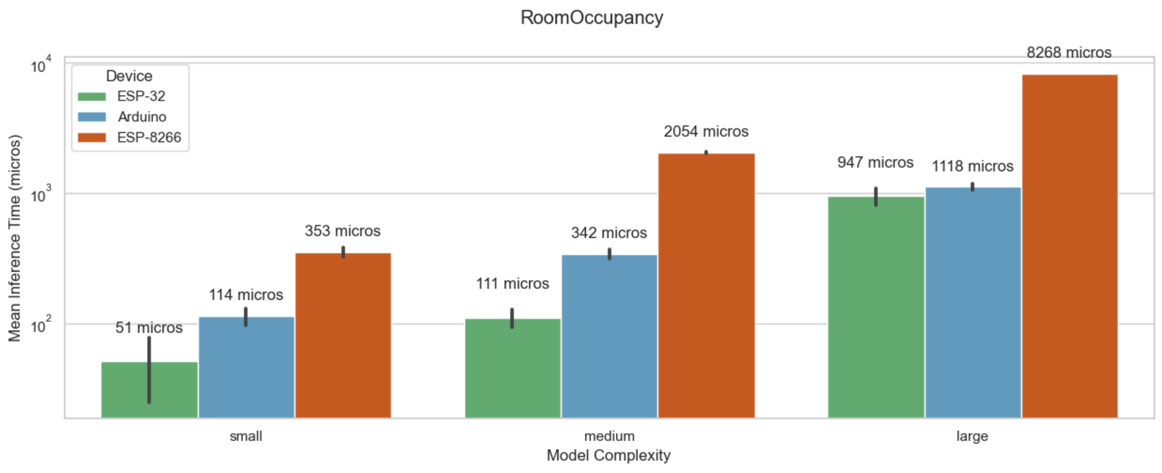

Figure 2: Average time (in micros) and standard deviation (dark line) for each device in relation to different model size.

stored and usually is chosen through trial and error. For this model, after some attempts, a value of 3*1024 is considered good.

In this section the goal is to understand how different model complexities affect RAM and FLASH memory and then compare devices in relation to their memory usage. The values obtained correspond to those shown by compiler after the building phase (not through PlatformIO Inspector because it has some problems (always 100% RAM) with Arduino). With this approach, the RAM values correspond to static memory allocations.

In table 6 are then summarized the values of RAM and FLASH in terms of percentage relative to the device's maximum capacity and can be seen that:

- ESP-32 has the lowest RAM usage (both in percentage and in KB);
- Arduino has the lowest FLASH usage (both in percentage and in KB);
- ESP-8266 has the highest RAM and FLASH usage compared other models;
- for all devices can be seen that the increase of model complexity leads to an increase of FLASH memory;

	(%) RAM required per model			(%) FLASH required per model		
	Small	Medium	Large	Small	Medium	Large
ESP-32 RAM=320KB FLASH=1MB	7,8% (25KB)	7,8% (25KB)	7,8% (25KB)	29,1% (380KB)	29,4% (385KB)	30,7% (402KB)
ARDUINO RAM=256KB FLASH=1MB	19,8% (52KB)	19,8% (52KB)	19,8% (52KB)	25,0% (246KB)	25,5% (250KB)	27,2% (276KB)
ESP-8266 RAM=80KB FLASH=1MB	59,6% (48,8KB)	65,4% (53,6KB)	86,0% (70,4KB)	37,2% (388KB)	37,6% (393KB)	39,3% (410KB)

Table 2: % of RAM and FLASH required by the devices for room-occupancy detection in relation to each model complexity.

3.2 Drowsiness Detection

The second model considered aims to detect, starting from an image (grayscale), if the eye figured in it is open or not. This type of model can be expanded and used in a real use case in order to understand (for example) if a person is starting to sleep and can be very usefull in the context of drive assistance.

In particular is used a specific dataset [Dro] and are created three different Convolutional Neural Network with increasing complexity:

- small: composed by single 2D convolution with 1 filter, 3x3 kernel, followed by a 2x2 max-pooling and a 16 unit dense layer;
- medium: composed by single 2D convolution with 8 filter, 3x3 kernel, followed by a 2x2 max-pooling and a 16 unit dense layer;
- large: composed by two 2D convolution separated by a 2x2 max-pooling. The first convolution has 8 filters and 3x3 kernel while the second has 16 filter and a 2x2 kernel size. At the end there is a 16 unit dense layer.

The image in input are resized in 10x10 pixel due to memory constraints on ESP-8266.

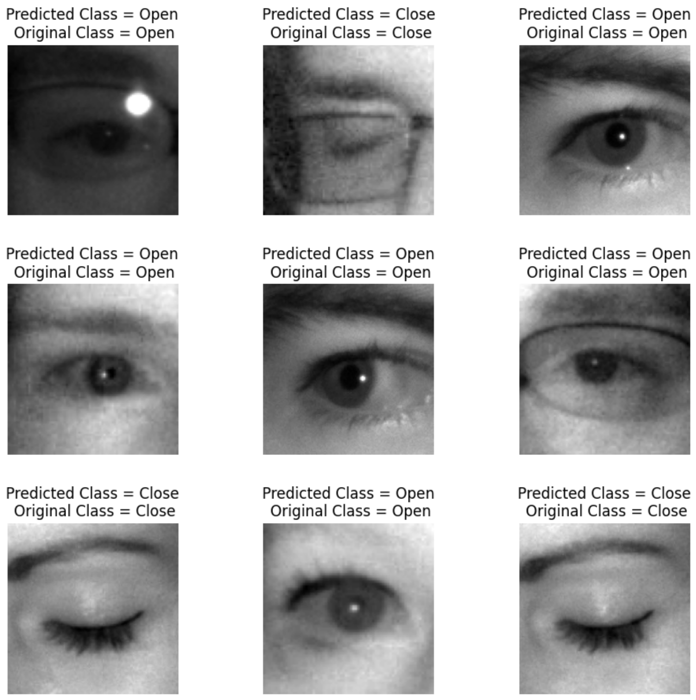

Figure 3: Example of prediction of the 'large' CNN model.

In figure 3 are reported some results from the 'large' CNN model in order to clarify the goal of this binary classification problem and table 5 summarizes the main characteristics of the created model in terms of accuracy and size.

Model (CNN)	Evaluation (accuracy)	Size
Small: (conv(1)+dense(16))	93,1%	3,9 kB
Medium: (conv(8)+dense(16))	93,6%	11,4 kB
Large: (conv(8)+conv(16)+dense(16))	96,3%	15,0 kB

Table 3: Models evaluation and size for drowsiness detection.

3.2.1 Inference time comparison

Here is adpoted the same experiment setup described for room occupancy detection problem 3.1.1 and the results obtained are almost the same. In figure 4 infact, can be seen that the increase of model complexity (in this case the number of convolution layers and their filters) leads to a growth of inference time for each device. In the same way in figure 5 is possibile to note that ESP-32 performs better than the other two devices and Arduino has a lower inference time compared ESP-8266 for each level of model complexity.

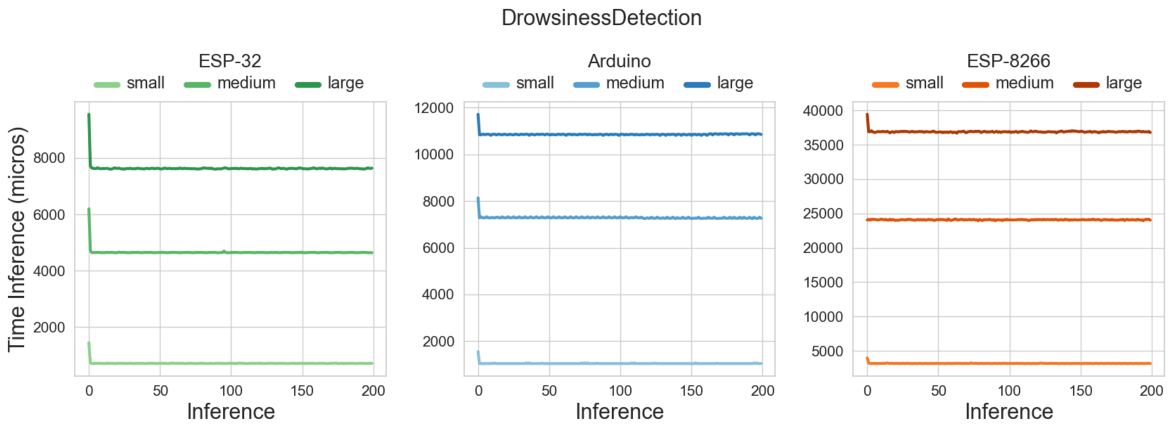

Figure 4: Time (in micros) for each of the 200 inferences (for drowsiness-detection problem) by each device in relation to different model size.

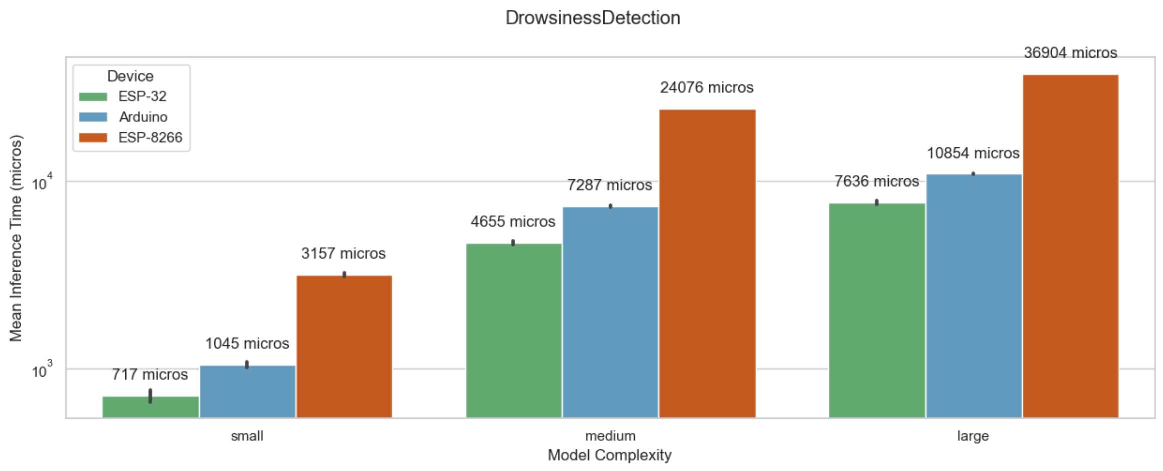

Figure 5: Average time (in micros) and standard deviation (dark line) for each device in relation to different model size.

3.2.2 Memory footprint comparison

For the comparison of memory footprint is chosen a tensor arena size of 20*1024. From the results obtained is possibile to do the same considerations done in section 3.1.2: ESP-32 has the lowest RAM usage while Arduino has the lowest FLASH consuption. All devices are subject to a higher FLASH consuption when model complexity increases and ESP-8266 results the worst in terms of RAM and FLASH usage.

	(%) RAM required per model			(%) FLASH required per model		
	Small	Medium	Large	Small	Medium	Large
ESP-32 RAM=320KB FLASH=1MB	13,2% (43KB)	13,2% (43KB)	13,2% (43KB)	29,2% (382KB)	29,8% (390KB)	30,1% (393KB)
ARDUINO RAM=256KB FLASH=1MB	26,6% (69KB)	26,6% (69KB)	26,6% (69KB)	25,2% (248KB)	26,0% (255KB)	26,3% (259KB)
ESP-8266 RAM=80KB FLASH=1MB	83,4% (68KB)	92,5% (75KB)	97,0% (79KB)	37,4% (390KB)	38,1% (398KB)	38,5% (401KB)

Table 4: % of RAM and FLASH required by the devices for drowsiness detection in relation to each model complexity.

3.3 Timeseries Forecast

The last model considered aims to resolve a generic time series forecast problem that can be usefull in various Iot applications.

Is chosen a generic dataset [Tim] (regarding the product daily sales history in a shop) and are created three different LSTM Neural Network with increasing complexity:

- small: composed by single LSTM layer with 10 units;
- medium: composed by single LSTM layer with 15 units;
- large: composed by two LSTM layers. The first with 15 units and the second with 10 units.

The model is trained using a set of 5 past (lagged) values as feature and the goal is to make on-step prediction ahead (as reported in figure 6).

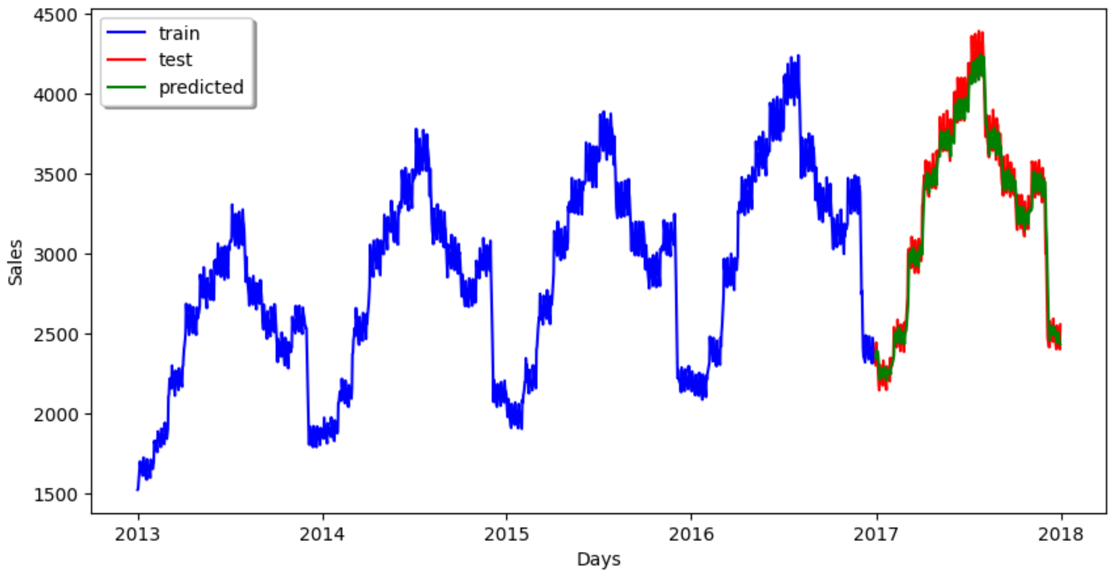

Figure 6: Example of prediction of the 'large' LSTM model.

In table 5 are then summarized the main characteristics of the created models in terms of error (MAPE) and size of the models created.

Model (LSTM)	Evaluation (MAPE)	Size
Small: (LSTM(10))	2,8%	5,2 kB
Medium: (LSTM(15))	2,2%	7,5 kB
Large: (LSTM(10)+LSTM(15))	2,8%	13,4 kB

Table 5: Models evaluation and size for drowsiness detection.

3.3.1 Inference time comparison

The results shown in figure 7 are in line with those obtained for the two precedent use cases in section 3.1.1 and 3.2.1: the increase of model complexity (in this case the number of LSTM layers and their units) leads to a growth of inference time in each device. Accordingly, in figure 8 is still confirmed that ESP-32 outperform the others (especially for 'large' model) and Arduino has a lower inference time than ESP-8266 for each of the possibile model complexities.

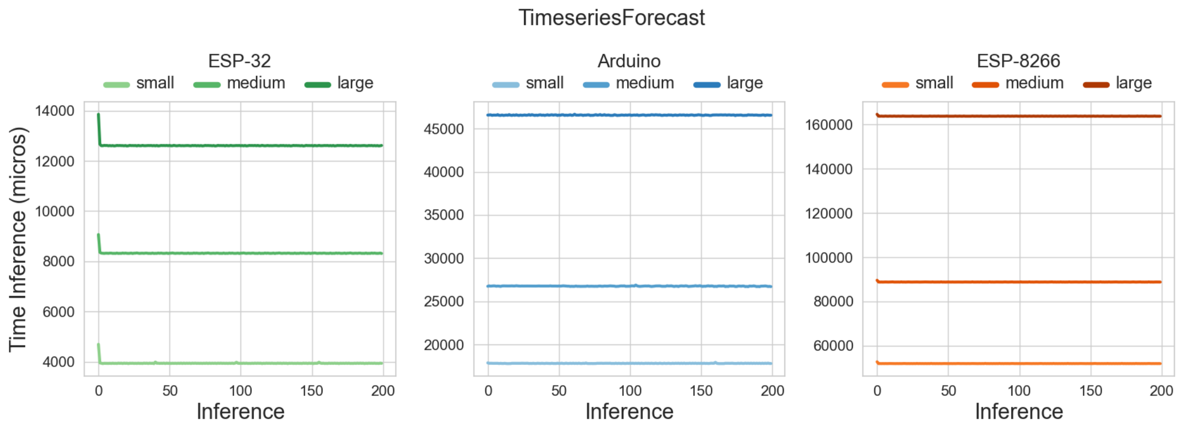

Figure 7: Time (in micros) for each of the 200 inferences (for timeseries forecast problem) by each device in relation to different model size.

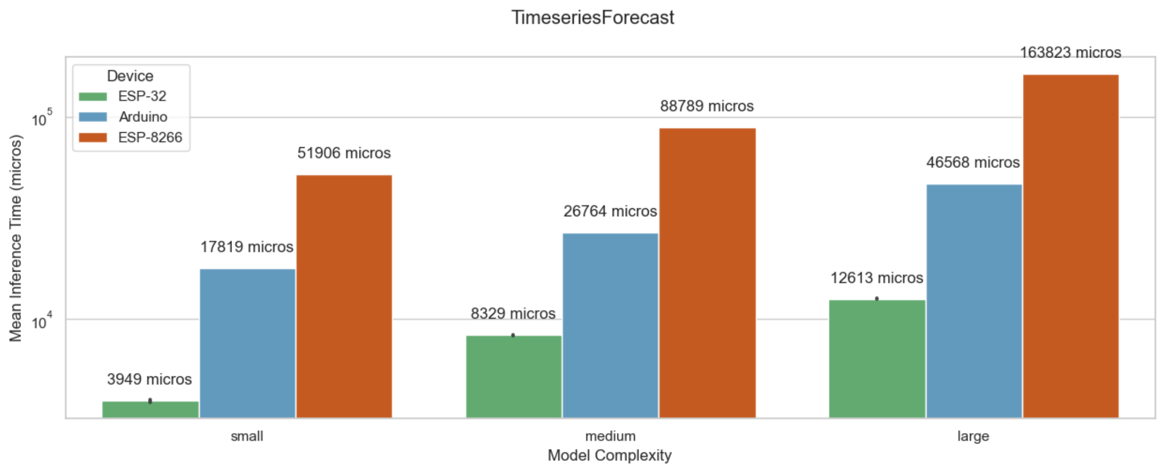

Figure 8: Average time (in micros) and standard deviation (dark line) for each device in relation to different model size.

3.3.2 Memory footprint comparison

For the comparison of memory footprint is chosen a tensor arena size of 6*1024. The results obtained allow to do the same considerations done in 3.1.2 and 3.2.2: ESP-32 is characterized by a low RAM consumption (and Arduino by a low FLASH) while ESP-8266 is the one with the highest RAM and FLASH consumption. Also in this case, when model complexity increases, all devices see a FLASH usage increase.

	(%) RAM required per model			(%) FLASH required per model		
	Small	Medium	Large	Small	Medium	Large
ESP-32 RAM=320KB FLASH=1MB	8,6% (28KB)	8,6% (28KB)	8,6% (28KB)	39,4% (516KB)	39,6% (518KB)	40,0% (524KB)
ARDUINO RAM=256KB FLASH=1MB	22,1% (57KB)	22,1% (57KB)	22,1% (57KB)	35,1% (345KB)	35,3% (347KB)	35,9% (353KB)
ESP-8266 RAM=80KB FLASH=1MB	87,5% (71KB)	90,4% (74KB)	97,5% (79KB)	50,2% (524KB)	50,4% (526KB)	51,0% (532KB)

Table 6: % of RAM and FLASH required by the devices for timeseries forecast in relation to each model complexity.

3.4 Performance improvement (on ESP-8266)

As seen from the results, ESP-8266 has the highest time inference compared ESP-32 and Arduino. In this section is illustrated the effects of quantization on ESP-8266 considering only the 'large' models for each of the three problems already discussed.

In particular is used *full-integer quantization* [Ful], a post-training quantization technique that consists in quantize all model tensors (also input and output) from floating-point to integer (8 bit precision).

The effects of this technique can be noted in table 7 where is possible to see that for each problem, the quantized model results significantly smaller than non-quantized but with a little degradation in terms of accuracy (or MAPE, depends on the problem). In order to compute the accuracy evaluation of quantized model is used a dedicated Tensorflow Lite interpreter (in python side) and input / output are quantized / de-quantized.

	Size (of 'large' model)		Eval (of 'large' model)	
	NON-quantized	Quantized	NON-quantized	Quantized
Room Occupancy	23,8KB	8,8KB	Accuracy: 98%	Accuracy: 95%
Drowsiness Detect.	15,0KB	7,4KB	Accuracy: 96%	Accuracy: 96%
Time Series	13,4KB	9,0KB	MAPE: 2,8%	MAPE: 5,2%

Table 7: Size and accuracy comparison between a non-quantized and full-integer quantized 'large' model for each problem considered.

Once the model is correctly converted and full-integer quantized, it can run on devices as 'normal' models. In relation to ESP-8266, in figure 9 are reported its inference average times (in microseconds) using a non-quantized and a fully-integer quantized ('large') model for each of the already discussed problems.

As can be seen, full-integer quantization brings a substantial improve in terms of inference time and its effectiveness depends on which model type is considered. The results obtained infact, show a 3,8x time inference speedup for the fully-connected (room-occupancy detection), a 1,6x speedup for CNN (drowsiness-detection) and a 5x speedup for LSTM (time series).

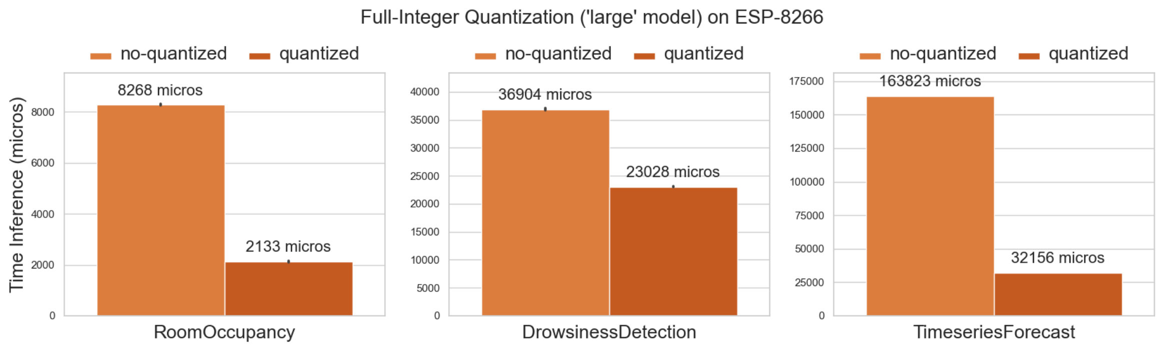

Figure 9: ESP-8266 average times (microsec) for inference with non-quantized and quantized models for each of problems.

4 Conclusions

In this project several neural network (fully-connected, CNN, LSTM) with different complexities ('small', 'medium', 'large') are created and deployed on different devices (ESP-32, Arduino Nano 33 BLE and ESP-8266). In terms of time inference and RAM consumption ESP-32 is the most effcient. Arduino has a higher time inference then ESP-32 while ESP-8266 is the worst in terms of time inference and memory consumption. It is then shown that the increase of model complexity increases time inference and memory consumption in each device. Finally is demonstrated the effects of full-integer quantization technique on ESP-8266: up to 5x speedup in time inference and a consistent model size reduction.

References

[Arda] Arduino ide. https://www.arduino.cc/en/software.

[Ardb] Arduino nano 33 ble sense. https://docs.arduino.cc/hardware/nano-33-ble-sense-rev2.

[Ardc] Arduino nano 33 ble specification. https://content.arduino.cc/assets/Nano_BLE_MCU-nRF52840_PS_v1.1.pdf?_gl=1*d22iam*_ga*OTE5NjU4OTg0LjE2ODQ0Nzk0NjQ.*_ga_NEXN8H46L5*MTY4NDQ5OTU1Ni4yLjEuMTY4NDQ5OTc4Mi4wLjAuMA..

[Ardd] Arduino tensorflowlite library. https://github.com/tensorflow/tflite-micro-arduino-examples.

[Dro] Drowsiness detection dataset. https://www.kaggle.com/code/tmleyncodes/drowsiness-detection.

[Elo] Eloquenttinyml library. https://github.com/eloquentarduino/EloquentTinyML.

[ESPa] Esp-32. https://docs.espressif.com/projects/esp-idf/en/latest/esp32/hw-reference/esp32/get-started-devkitc.html.

[ESPb] Esp-8266. https://www.espressif.com/en/products/socs/esp8266.

[Ful] Tensorflow full-integer quantization. https://www.tensorflow.org/lite/performance/post_training_quantization?hl=en#full_integer_quantization.

[Pla] Platformio. https://platformio.org.

[Roo] Room occupancy dataset. https://www.kaggle.com/code/ashrafsharifi/exploratorydataanalysisproject.

[Ten] Tensorflowliteesp3 library. https://github.com/tanakamasayuki/Arduino_TensorFlowLite_ESP32.

[TF] Tensorflow. https://www.tensorflow.org.

[Tim] Timeseries forecast dataset. https://www.kaggle.com/code/dimitreoliveira/deep-learning-for-time-series-forecasting/notebook.

[Tin] Tinyml book. https://tinymlbook.com.

[TLC] Tensorflow lite converter. https://www.tensorflow.org/lite/models/convert/.

[TLMa] Tensorflow lite micro. https://www.tensorflow.org/lite/microcontrollers.

[TLMb] Tensorflow lite micro c++ library. https://www.tensorflow.org/lite/microcontrollers/library?hl=en.

[TLMc] Tensorflow lite operation supported. https://www.tensorflow.org/lite/microcontrollers/build_convert?hl=it#operation_support.