POLITECNICO
MILANO 1863

TinyML techniques for Fruit Classification

Authors:
Vicente Castro & Daniele Sinigaglia

Professor: Manuel Roveri

Teaching Assistant: Massimo Pavan

Course: Hardware Architectures for Embedded and Edge AI

January 23, 2024

1 Introduction

The proposed solution aims at automatising the ***fruit categorisation*** process done by the user. This task is initialised when the customer wants to weight their desired fruits or vegetables at the scale and finishes with the right identification and posterior labelling.

From a technical perspective, the system leverages the *TinyML* infrastructure to ***classify images*** in real time. Internally, the main hardware and software components are:

- The camera module *OV7675* with 0.3MP precision.
- Arduino Nano 33 BLE with *1MB ROM* and *256KB RAM*.
- TensorflowLite[1] library for micro-controllers.
- Camera libraries, a mix from *HarvardTiny-MLX* and *OV767X*.

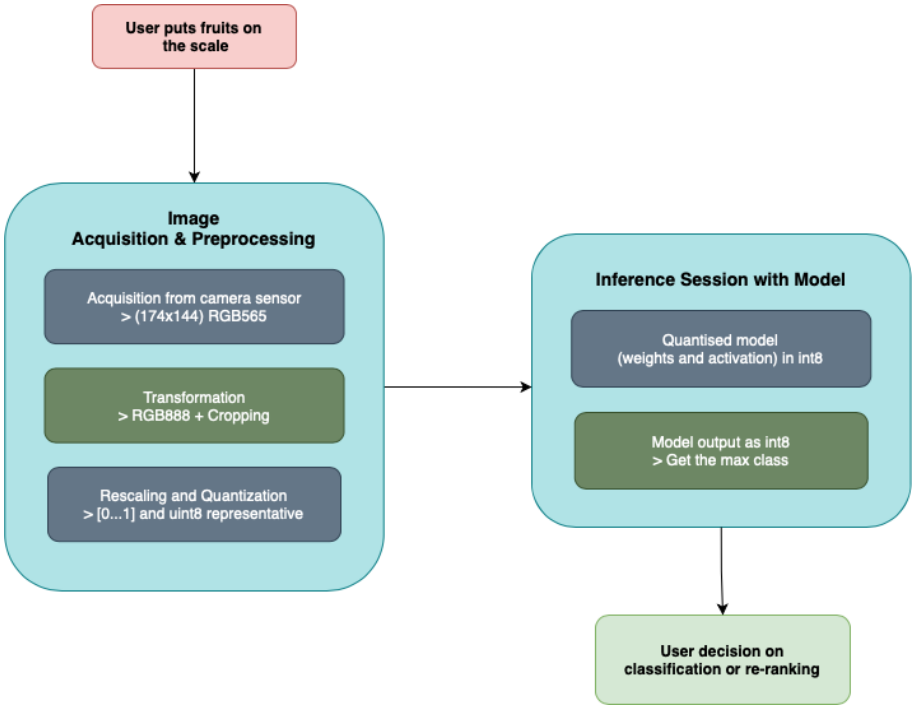

Figure 1: Solution diagram

2 Methods

This section presents the main developing phases, starting from how and why the solution followed an specific preprocessing, to the model design decisions. Furthermore, the main datasets are presented, including their "real-world" dynamics and relevancy for the problem.

2.1 Hardware Capabilities

Before acquiring dataset and selecting models, it is relevant to understand the state and the capabilities of the presented hardware. As a first step on the design, an *exploration* on the capabilities of the camera and Arduino memory was done.

2.1.1 Frame Size and Colour Representation

The original *camera module* can take pictures in different formats and image sizes. Most notably, for coloured images two types of schemes are available: `RGB444` and `RGB565`, the numbers represent how many *bits* per colour channel are written. In the first case, 12*bits* (1.5Bytes) are used to store a single pixel, with the second, 2Bytes are needed. Given that the **second case offered a bigger precision**, this colour representation was chosen.

In *Computer Vision* applications, scaling the input resolution usually improves the overall performance of the model. Following this notion, scaling to the biggest possible *image* size for the frame is relevant. The *OV7675* offers 5 different sizes:

- ***QQVGA***, with a dimension of (160px, 120px) or 19200 pixels. A total of 38.400Bytes ($\sim$ 38.4KB) in the chosen representation. About 15% of the total available *RAM*.
- ***QCIF***, with size (176px, 144px) or 25344 pixels. Nearly $\sim 50.6KB$ or 20% of the *RAM*.
- ***QVGA***, with dimension (320px, 240px), using $\sim 153.6KB$. Nearly 60% of the *RAM*.
- ***CIF***, with dimension (352px, 240px), using $\sim 168.9KB$. Close to 66% of the *RAM*.
- ***VGA***, with dimension (640px, 480px), using $\sim 614.4KB$. More than double the board *RAM*.

Considering the memory required for the model inference (allocation weights and activation layers) and input quality, **the *QCIF* size was selected**. At an early stage, the *QQVGA* format was tested, but the quality of the image taken seemed to suffer distortions that could potentially decrease any model performance. Furthermore, the chosen resolution allowed the biggest frame, while allowing having a decently small *convolutional network*.

2.1.2 OV7675 Receptive Field

Another camera constraint was the *receptive field* of the module or how big is the visual space the camera can see. The sensor presented a narrowed view, to illustrate the reader, figure _ presents the obtained image at $1mt$ from the target.

Many of the application for these memory constrained devices use a small image, with a predefined size as input. Normally, from the captured image, a preprocessing layer would either *crop* of the centre frame or *resize* the whole image to the desired shape. As the second operation is more expensive in both memory and computations, **the application uses *cropping* of the central area** of the image to obtain the model *field-of-view (FOV)*.

This design choice imposed a problem given the small receptive field, this processed narrowed even more the FOV. Two software solutions were used to overcome the issue. First, the **model input size** was maximised for the memory, from the typical (96, 96) shape, it was extended to (112, 112), allowing a bigger FOV. Additionally, the model was trained, via augmentations, to incorporate these reduced views.

2.2 Datasets

Two datasets were used in order to train the models:

- The *Fruit-Recognition Dataset*[3], presents a collection of 44406 images acquired using a low-resolution $320x258$ pixels camera. This dataset was conceived for producing images with a context like supermarkets and fruit-shops, presenting different shadows, lightning, and positions. Furthermore, the background presented is metallic like a common scale.
- *Arduino Dataset*, collected manually with the available Arduino and camera module. It was taken in a *simulated* environment, also **including different lightning conditions and direct interference with bags and human hands**. Consists of 300 images of $112x112$pixels (50 per class), stored after applying the preprocessing function explained in 2.2.1.

As the first dataset contained many types of fruits, a cleaning of the classes was performed. Filtering by popularity and consumption, the **dataset was reduced to** 6 **classes:** *Green Apples*, *Red Apples*, *Bananas*, *Mangoes*, *Oranges* and *Tomatoes*. As a first approach to the problem, the reduction in the number of classes is to understand how good a model can perform under these constrained conditions to later scale it up. Some results on this are discussed in later sections

Finally, it is relevant to mention a business **assumption made for the collection of the second dataset**. The proposed solution must be easy to train. In the current state, as each market and food store is different, a small re-training will have to be done, specially, if both the hardware devices and acquisition conditions cannot be fully controlled. By scale necessities, these new images will need to be collected by the owner, effectively **constraining the number of samples available for re-training**. This idea carried to collection of this *"big-enough"* second dataset.

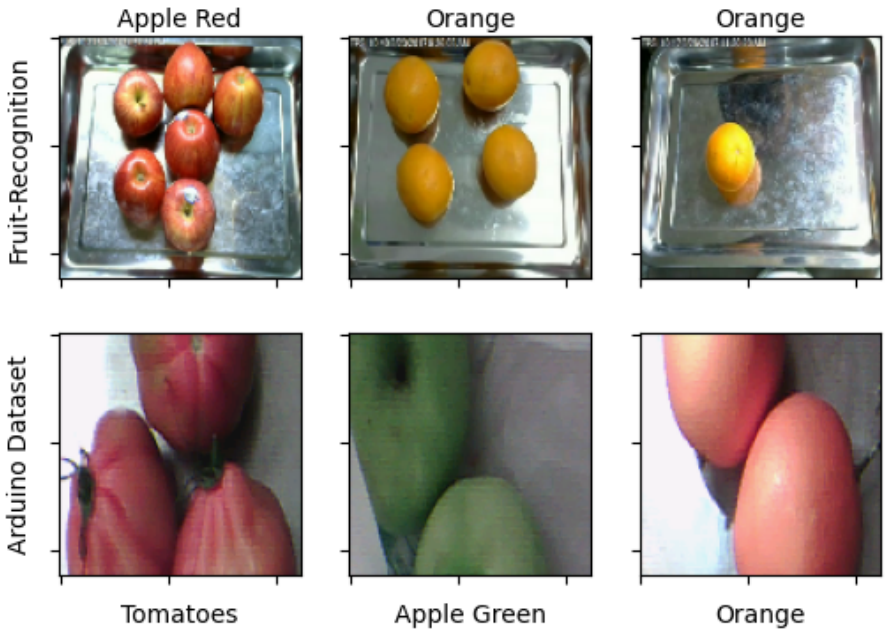

Figure 2: Samples from both datasets

2.2.1 Acquisition and Pre-processing

As the camera-acquired images are represented in a `RGB565` format a preprocessing step was necessary to get the typical `RGB888` input. Additionally, the cropping of the frame is performed at the same time for memory efficiency.

An image stored in *5-6-5* format will have a different pixel values limit. In this case, the red and blue channels will have a valid range in the $[0, 31]$ interval, while the green (a much more distinguishable colour for humans) will be in the $[0, 63]$ range. Another added problem is the fact that these three values are "inside" only 2Bytes, so bit operations are needed to separate each channel value. These computations are depicted in the preprocessing function of Figure 3.

```
// Convert RGB565 (16-bits) to RGB888 (24-bits)
uint8_t baseRed = (pixelCombined & 0xF800) >> 11;
uint8_t baseGreen = (pixelCombined & 0x07E0) >> 5;
uint8_t baseBlue = (pixelCombined & 0x001F);

uint8_t red = (baseRed << 3) | (baseRed >> 2);
uint8_t green = (baseGreen << 2) | (baseGreen >> 4);
uint8_t blue = (baseBlue << 3) | (baseBlue >> 2);
```

Figure 3: Pre-processing function (`pixelCombined` is a short representing the pixel)

As an observation, a common input range for *CNN* models is [0..1], an operation that could be performed by dividing each channel by its max value. Nevertheless, doing this direct step changes the results, slightly, compared to doing a remapping to 255 (8bit representation) before the division. The latter approach was taken, as it facilitated the dataset acquisition and it only needed 3 extra bytes of local memory per pixel.

2.2.2 Data Augmentation

There are strong reasons to augment both datasets, but specially for the first one, a key performance point is how close the dataset is to the on-production data. As previously illustrated, the images obtained with the camera module have a short receptive field, and thus, most of the times, the fruit will not be contained in the image. Furthermore, on inspection, the images included in the *Fruit Recognition Dataset* are of much more quality that the manually acquired, probably due to using `RGB888` and better sensors. A visual comparison is provided in Figure 2.

To tackle these issues, two augmentations were heavily used while pre-training the models on this

dataset:

- `RandomZoom`, which randomly zooms into the image. It was performed with a high zooming factor.
- `RandomColorDegeneration`, which randomly transforms the image to grayscale and back, diminishing the represented colour in each channel.

Additionally, commonly used augmentation like `RandomFlip`, `RandomRotation`, `RandomContrast` and `RandomBrightness` were also included. All these operations are computed on the fly in the training phase, providing countless transformations for one image and no extra storage needs. An overview of these is gshown in Figure 4.

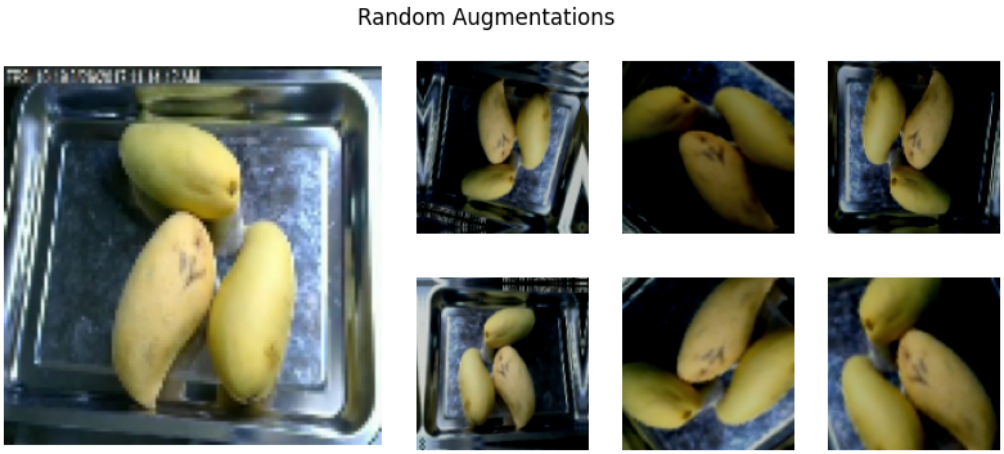

Figure 4: Random augmentation on *Fruit-Recognition Dataset*

When fine-tuning the models with the acquired Arduino dataset, only the latter augmentations were kept. Regardless, heavy augmentation was still used to regularise the model and avoid overfitting under few examples.

2.3 Models and post-processing

The base architecture for all the tested models is a ***MobileNetV1*** [4] **with depth factor of** 0.25. This model was used as a features extractor and, on top of it, a *classification head* with an output layer of 6 neurons was attached.

After some iterations to search for the minimal model memory-wise, the architecture in Figure 5 was chosen:

- `MobileNetV1` backbone, a checkpoint of the weights trained on *ImageNet*[2] dataset were used. These weights were trained on images with shape 96x96 and contain 218544 parameters.
- `BatchNormalization` played a crucial role for the model performance, without this intermediate layer the model failed to learn on any task.
- `ClassificationHead` consists of a single *Dense* layer with 6 output neurons. Due to numerical stability, no activation function was used on this layer.

2.3.1 Quantisation

A crucial step to be able to run these models in the constrained environment of Arduino, is the post-processing phase. The final model has 221.110 parameters and uses a total memory of $863.7KB$ (most of the values are `floats`, but there are `ints` too), as such, it cannot be loaded directly in the *RAM* of the board.

The main **post-processing step for all the model is the *quantisation***, a process in which, the weights, activations and possibly the inputs/outputs are changed to be represented as `int8` instead of *floats*. This reduces the total size of the weights and activations but can harm the performance of the model.

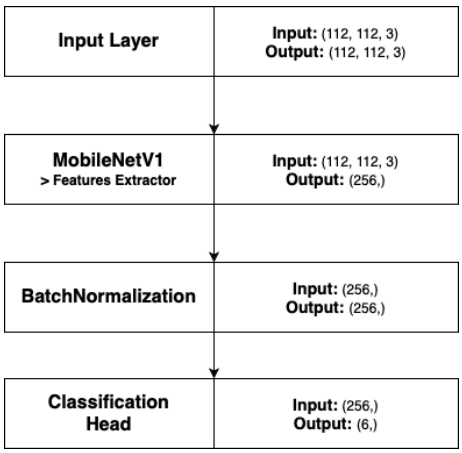

Figure 5: Base model

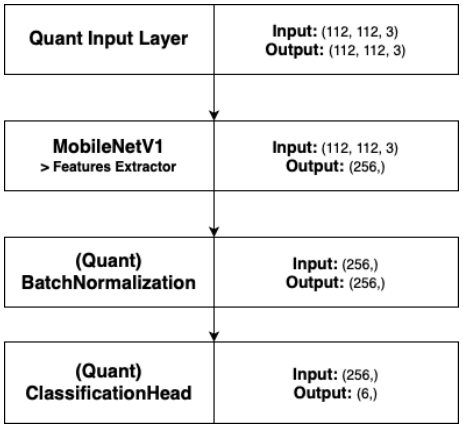

Figure 6: Quant-aware Model

This process works well for the `Classification Head` and the `BatchNormalization` layers, but it is only able to directly transform around 60% of the *MobileNetV1* architecture. This is a problem given that some fallback and will occur and some optimisation will be necessarily lost. Three quantisation schemes were studied:

- `Dynamic range quantisation (DRQ)`, quantifies the weights to an integer representation. Then, at inference time, the activations are quantised on the fly to operate with the *integer* weights.
- `Integer with Float-fallback (Mixed)`, it a similar scheme as the previous one, but tries to convert all the models' maths to an integer representation. When this is not accomplished, the available float operation is used.
- `Full integer only (FIQ)`, this is the full quantisation of the full to their integer values. Additionally, *input* and *output* results are also quantised to integers. This method resulted in the biggest reduction in *on-device* memory and is the only one that can be ran on the Arduino board.

For the last two methods, a `RepresentativeDataset` needs to be provided. During the conversion, a profiling of the inputs and activations values is computed, the range of the quantisation is adjusted based on these results. The dataset used for this purpose was the training dataset.

Method	Memory usage	Ralative Speedup
Base Model	2.1MB	x1
Dynamic Range	293KB	x-4.8
Mixed	323KB	x2.4
Full-integer	323KB	x2.1

Table 1: Summary statistics on the quantisation methods

Table 1 presents the relevant statistics on the quantisation process. It is relevant to notice how the *DRQ*, even when utilising less memory, takes a negative speedup in computing time. This may be due to the *dynamic* quantisation at the activations at inference time.

The final model size is 323.160B or $\sim 323KB$. The model is compiled and passed into the Arduino memory as a `constant`, utilising the *ROM* and the *RAM*. Nevertheless, all the activations and intermediate results do need to be on *RAM* and, with this, the model total memory usage is **around** $\sim 70\%$ **of the total *RAM*.** To whole script, including the camera buffer utilises a **total of** 93% **of all the available memory**.

2.4 Training

The training procedure was divided in three phases:

2.4.1 Fruits pretraining

The initial training of the model was performed on the *Fruit-Recognition* dataset. During this phase the model learns the general 6 classes and utilises all the heavy augmentations discussed in prior sections. This phase should produce a robust model capable of recognising the targeted fruits.

For validation and testing purposes, the dataset was divided into a *train* and *test* splits. After, the *train* set was again divided into *training* and *validation* datasets, the latter was used for the model selection procedures.

The initial training is done via **transfer-learning**. As the weights in the classification head are initialised from random, early iterations could have unstable gradients and potentially harm the learned representations on early layers (if trained jointly). Once the latter layers are trained, the rest of the model is unfreeze allowing the gradients to flow and **fine-tune the early layers** to the specific data input distribution.

2.4.2 Fine-tuning on the Arduino Dataset

As this dataset is much smaller and faster to train on, the model selection was done by leveraging cross-validation. The original dataset was divided into *train* and *test*, then 10 *stratified folds* were computed from the *train* split. This method was implemented to search for the best set of hyper-parameters to train the final model on.

This section phase presented two main difficulties. First, as the dataset was rather small, the model was prone to overfitting. To overcome this problem, three heavy regularisation techniques were used:

- **Data Augmentation**, as discussed before, but with stronger random factors.
- `Weight Decay`, add the notion of constraining the norm of the weights (it is a L_2 regularisation). This prior, effectively affects how much the weights can adapt during training, reducing overfitting.
- `Dropout Layer`[5], was add before the output layer. This layer works, during training, by masking out random values of the input tensor. The layer was removed when the model is exported.

During the *CV* search for the hyper-parameters, the optimal *Learning Rate* and *Weight Decay* values were looked up. Additionally, these results were used to guide the total number of epoch to train on.

2.4.3 Quantisation-Aware Training

As a last step, in order to train the model in a more similar task to the one it will perform, **quantization-aware training was studied**. The basic idea is to add a *wrapper* on each layer, so the inputs and outputs are quantised accordingly, in this way, the model can learn to perform the same task on precision-constrained inputs. The added operation needs to be differentiable so the backward pass works, fortunately, its gradient can be approximated to 1.

The main problem faced during this stage was the conversion from the original base model to the quantized version, several tricks needed to be used. As mentioned before, the *MobileNetV1* backbones cannot be fully quantised and, as such, no trivial wrapper can be used on this "layer". For this reason, this layer was not quantised during this step, but a **compositional trick was utilised to quantise the input of this layer**.

Furthermore, the `BatchNormalization` layer does not accept a standard quantisation if the previous layer is not a convolution. As this is not our case, a custom quantisation technique was needed. After its definition, the final *quant-aware model* can be seen in Figure 6.

3 Results

As previously mentioned, each dataset was divided in two splits: *train* and *test*, then *train* was again divided into *training* and *validation* (either statically or with *cross validation*). In the following results, all hyper-parameters were computed using the *validation* metrics, then, the model was

trained with these values in the entire *train* set. The metrics are always computed on the unseen *test* set.

3.1 Pre-training phase

The main model selection problem was to understand how good this "basic" model could behave against the fruits dataset. The first use of the *validation* split was to determine the best number *transfer-learning* epoch to use. During the experiments the best results were obtained when training for 18 epochs. These values are shown in Figure 7.

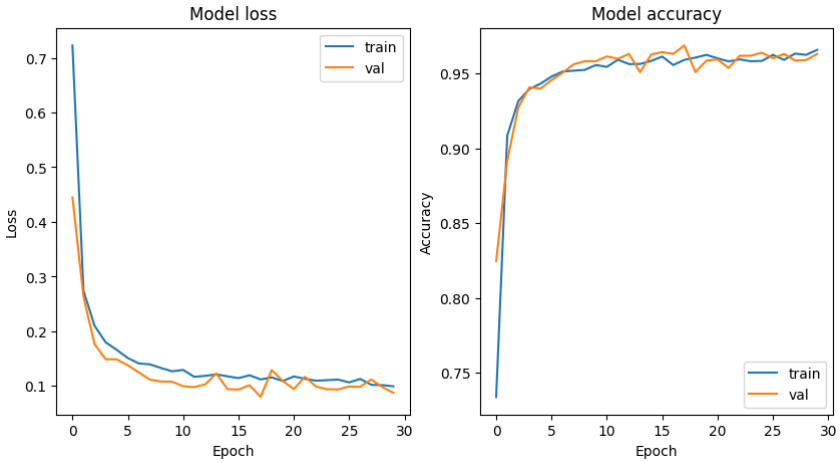

Figure 7: Loss and Accuracy during Transfer Learning

The second step was to understand how much *fine-tuning* of the convolutional backbone was needed to improve performance. As a rule of thumbs, it seems relevant to at least do this process for at least *one* epoch, as the slight differences in data distribution and acquisition could be relevant for the underlying representation. A much more unstable evolution was obtained, with the best results being at *seven* epochs.

During all the experiments a *Scheduler* for the learning rate value was utilised. It uses exponential decay and a fairly small initial value of 2×10^{-3}.

After re-training the model with the whole *train* split and the obtained hyperparameters. The metrics for the base and quantised models were computed. This procedure is also followed in the next sections. A table summarising these results is presented in Table 2, it is relevant to notice that, even when degradation of the performances occurs, this is relatively small.

Model	Accuracy	Macro-F1 Score
Base Model	0.957	0.957
DRQ	0.939	0.939
Mixed	0.931	0.930
FIQ	0.932	0.932

Table 2: Test metrics for the base model

Finally, the confusion matrix for the most relevant models is presented in Figures 8 and 9. Interestingly, the reduction in performance is not homogeneous on all the classes, but affects the *Green Apple* and *Tomatoes* the most, the classes that in the base state had the worst performances.

3.2 Training on the Arduino Dataset

During these experiments, *cross-validation* was used to a more robust model selection. Using ten stratified folds, a search for the best *Learning Rate* and *Weight Decay* was employed. This was an iterative process with manual adjustments on the search values based on the results. On the most redefined search, the values in Table 3 show the metrics aggregated on the all the splits.

A relevant result during this phase was that, contrary to the pre-training methodology, here the *fine-tuning* was done before and it affects only the *CNN* backbone. Some insights on why this was

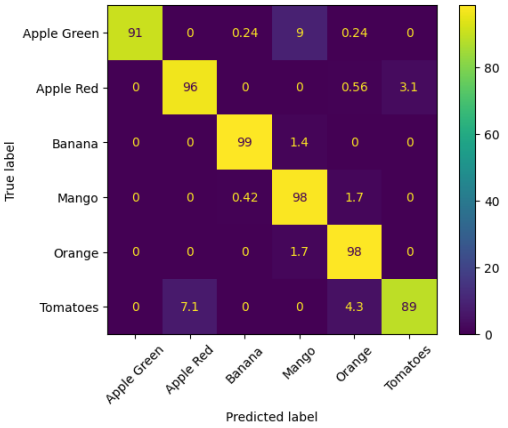

Figure 8: Confusing on the base model

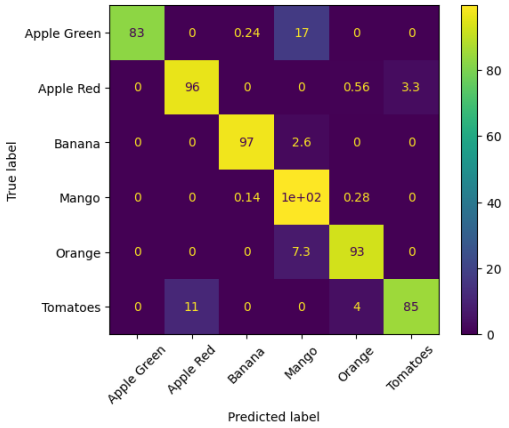

Figure 9: Confusion on the FIQ model

LR	Weight Decay	Fine-Tuning Eps			Transfer-Learning Eps			Accuracy		
		mean, std, max			mean, std, max			mean, std, max		
0.005	0.025	42.4	25.76	81	8.1	4.01	15	0.58	0.08	0.67
0.005	0.075	40.7	20.19	69	7.8	3.49	13	0.57	0.13	0.75
0.015	0.075	19.5	9.24	43	3.7	1.64	6	0.57	0.06	0.67
0.015	0.025	20.9	12.01	43	4.7	3.16	12	0.57	0.057	0.64
0.030	0.025	12.6	4.74	24	3.4	2.07	7	0.53	0.08	0.63
0.030	0.075	12.1	5.13	25	2.9	1.85	7	0.53	0.09	0.72

Table 3: Aggregated statistic over tall 10 splits

significantly better than training both together and only the classification head, may come from the fact that the underlying representation of each fruit didn't change much, but the input image distribution changed significantly. These shifts come from the different image qualities, and the representation power used to acquire them.

For training on the entire dataset, the hyper-parameters in Table 4 were used, guided mostly by the *max* values obtained during the validation. Following the results in Table 5, a bigger degradation of the performance is present and hence, a validation to explore *quant-aware training*. The two relevant confusion matrices are presented in Figure

LR	Weight Decay	Fine-Tuning Epochs	Transfer-Learning Epochs
0.005	0.075	75	15

Table 4: Selected parameters for training

Model	Accuracy	Macro-F1 Score
Arduino Model	0.853	0.849
DRQ	0.820	0.801
Mixed	0.820	0.821
FIQ	0.803	0.810

Table 5: Test metrics for the Arduino-finetuned model

3.3 Quantisation-Aware Training

Finally, the quantisation-aware training results are presented. As previously mentioned, this model can overfit very easily, given the reduced precision of its weight. To overcome this, is was trained under heavy regularisation and for a small number of epochs. Furthermore, the training procedure only involved the training on the whole *train* set as a continuation over the previous model.

The parameters chosen for the training are presented in Table 6. The *small learning rate* comes from the notion of not moving to much in the weights space to avoid disturbing the learned representation.

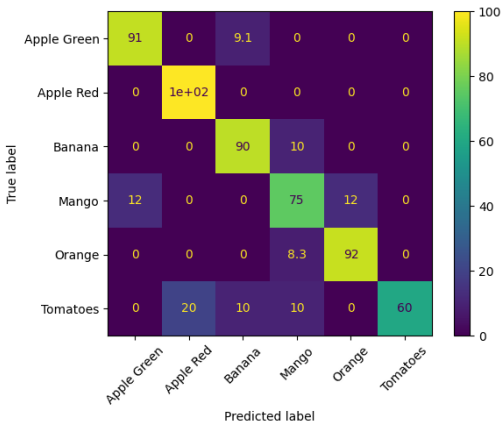

Figure 10: Arduino trained model

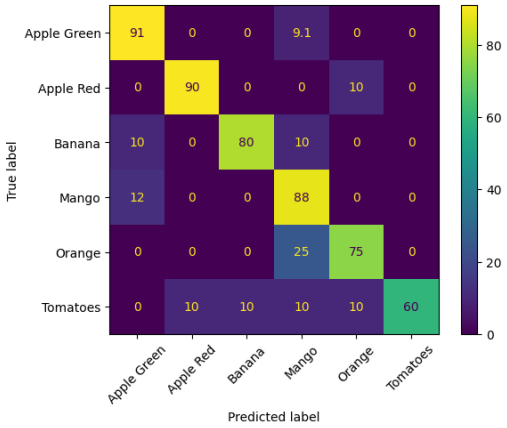

Figure 11: Arduino trained FIQ model

LR	Weight Decay	Training Epochs
1e-4	1.5e-1	16

Table 6: Parameters for QAT

To convert this model to a *micro-controller* friendly model, a direct TFLite transformation can be implemented. Nevertheless, this adds a couple of parameters used during training on the `QuantizeLayer`, removing these layers is optimal for inference and memory usage. This process does not affect performance.

Table 7 shows the results of evaluation the QAT model on the unseen *test* split. Comparing to the previous iteration, there is **vitually none degradation for the first two quantization techniques** and there is even an increase in the F1-Score metric. Even though, for the FIQ model a slight degradation is observed, both metrics remain bigger than the not-QTA model, this indicates the procedure helped in the performance.

Model	Accuracy	Macro-F1 Score
QAT Model	0.836	0.814
DRQ	0.836	0.814
Mixed	0.836	0.839
FIQ	0.820	0.829

Table 7: Test metrics for the QAT model

An explanation for the performance degradation is on fact that the *MobileNetV1* backbone is not completely quantizesable. In fact, as many of the inner operations could not be quantized, the working version of the QTA model is the same as the *Mixed* model and thus, the good relative performance of this method.

The results are presented in Figures 12 and 13.

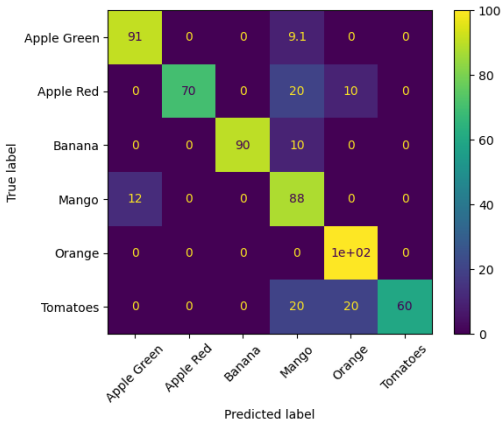

Figure 12: QAT with Mixed procedure

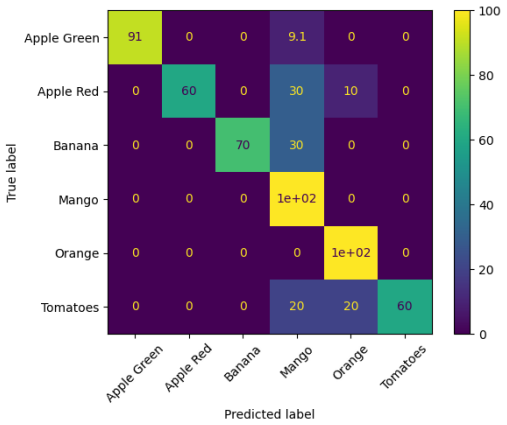

Figure 13: QAT with FIQ procedure

4 Conclusion

This report serves as a first technical guide for the implementation of live-time fruit classification on micro-controllers. Many of the topics discussed are currently being researched and are going to improve significantly in the next couple of years. Nevertheless, ideas like *Quantisation* and *Quantisation-Aware Training* are already changing how we run ML models.

This project was developed using two datasets crafted specifically to have market-like conditions at acquisition times. The relevance of this kind of data, trying to be as close as possible to the real-world scenario, is key for the development of application that are efficient and yet versatile to the real world. Furthermore, it was shown how *software-level* preprocessing can enhance these initial conditions to extract the best of these kind of applications.

There are many things left to try and there is a clear **road map towards future iterations of this project**. Putting aside HW condition, the presented model works well but some improvements need to be done to have a production-ready algorithm.

- **Decoupling from *MobileNetV1* architecture**. This model worked well, and good results were obtained. Nevertheless, the constraints on the level of quantisation available are a key aspect, specially to increase model performance on the *QAT* procedure.
- Additional pre-training. As mentioned before, the environment conditions will be fully controlled by the developers, so a robust model is a key aspect to the success of this application. As a future work, before fine-tuning on the *Arduino Dataset*, a great methodology could be to **meta-train model weights on several manually acquired datasets** (under different machines). In this way, more robustness and easier fine-tuning at the end-user can be obtained. This process is comparable to a *Few-Shot Learning* scheme.

Finally, as an additional business critical layer, a proposal for the study of *how many classes can this model comfortably classify*. The base model showed an *accuracy* of 95% on the pre-train data, even under heavy regularisation. This is a sign that the model is capable enough to handle additional classes. Nevertheless, each new class will add 257B of required memory, so there is a limit on the number that can be included in the HW.

References

[1] Robert David et al. *TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems*. 2021. arXiv: 2010.08678 [cs.LG].

[2] Jia Deng et al. "ImageNet: A large-scale hierarchical image database". In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.

[3] *Fruit Recognition Dataset by Chris Gorgolewki*. 2020. URL: https://www.kaggle.com/datasets/chrisfilo/fruit-recognition.

[4] Andrew G. Howard et al. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. 2017. arXiv: 1704.04861 [cs.CV].

[5] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.