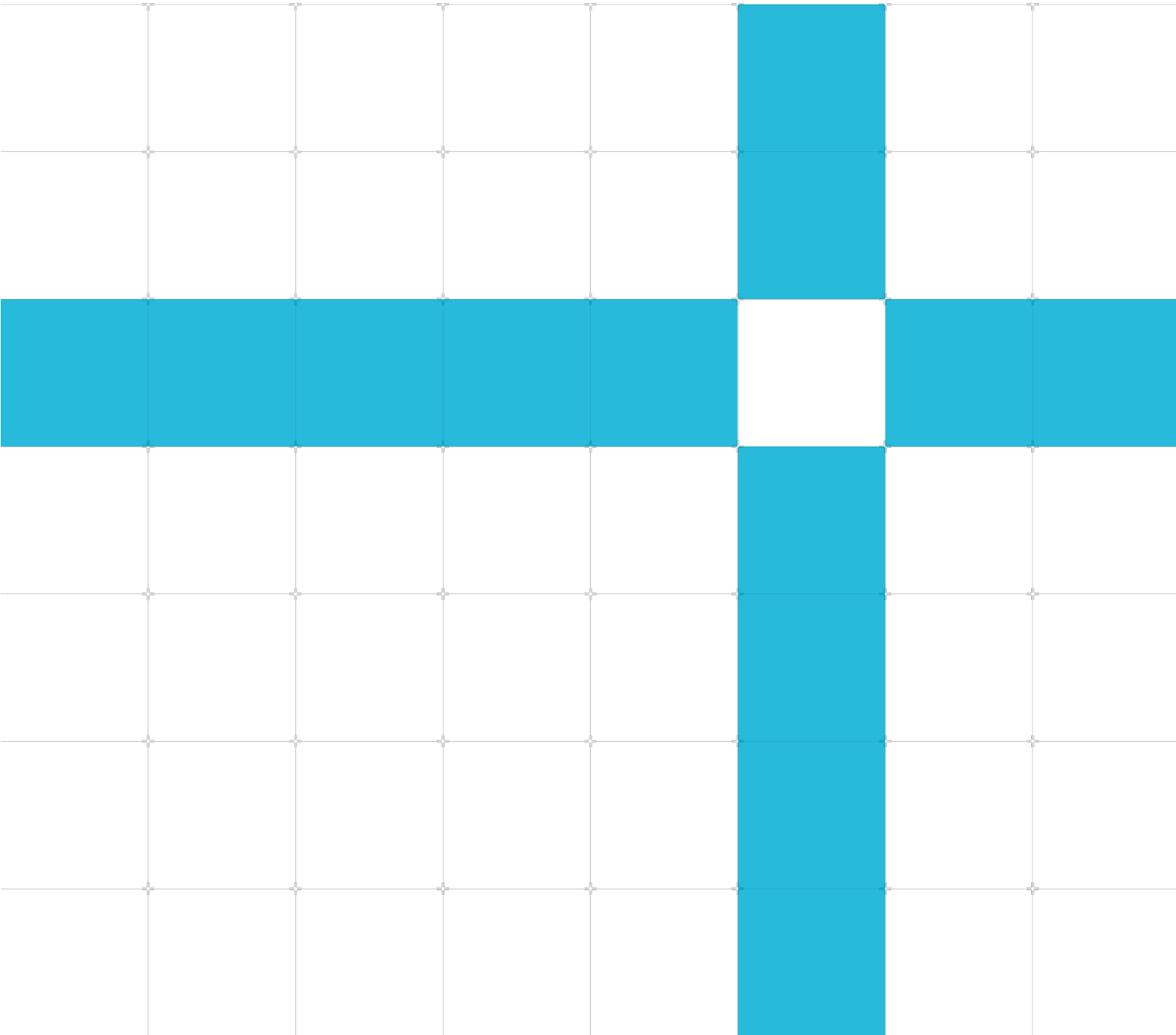
arm

Explore an MCU-friendly face recognition model

Non-Confidential

Issue 0100
102295_0100_00



Explore an MCU-friendly face recognition model

Release information

Document history

Issue	Date	Confidentiality	Change
01	19th November 2020	Non-confidential	First release

Non-Confidential Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Web Address

www.arm.com

Contents

1 Overview

This guide introduces some of the design considerations and best practices to consider when training and quantizing an MCU-friendly face recognition model.

In recent years, facial recognition technology has become ubiquitous in daily life. There are numerous ways that it has made our lives easier, including:

- Electronic passport control gates
- Securely unlocking your smartphone
- Automatically tagging your friends in photographs

Great advances have been made in the accuracy and reliability of these systems. These advances have mainly come about due to the adoption and use of neural network models.

However, these accuracy gains come at the cost of increasing compute and power requirements. This increased need for compute power means that performing inferences on these models is traditionally offloaded to servers in the cloud. Offloading the task to the cloud comes with its own problems though, mainly in latency and data security.

Therefore, there is now a significant push to develop hardware and software that enables execution of these networks quickly and securely on edge devices. Security and latency are improved because user data never has to leave the device. This is because there is no need to rely on patchy internet connections.

1.1 Before you begin

Before you read this guide, you should be familiar with MCU-friendly face recognition models and with the concept of quantization.

2 What happens in an MCU face recognition model?

In this section of the guide, we provide an explanation of what is happening in our facial recognition model. At the core of our model is a powerful convolutional neural network. After training, this model will calculate a facial fingerprint for every face that it is shown.

An Arm microcontroller-based device like a Cortex-M4 or a Cortex-M7, stores a database of known facial fingerprints belonging to people who we would like to recognize. When the model tries to identify a new face, the model calculates the facial fingerprint and compares it to the facial fingerprints in the database. If the facial fingerprint is similar enough to a facial fingerprint in the database, the model outputs who it thinks the face belongs too. it is. If the model does not recognize the facial fingerprint, then the model outputs that it does not recognize this new face.

To get our model to produce good facial fingerprints, we train it on a classification task. We train the network to correctly classify the images of different celebrities. If you are working on your own model you can use a public dataset that contains images of celebrities, for example the **CelebA dataset**.

Note: To make sure that we had an unbiased and robust facial recognition model, we used training data that contained images of celebrities from a variety of different ethnicities and races.

After training to a suitable accuracy, we removed the final classification layer of the model. What is left is a model that produces a feature vector as our facial fingerprint. By learning to classify many different people, we hope that the model has also learned to embed knowledge about different face characteristics within this feature vector. Some characteristics of the face that the model may use, include the distance between eyes or size of mouth which could be used to distinguish people's faces.

3 Model choice

In this section of the guide, we explain the reasoning behind our model choice.

In this guide, we are working in the embedded space. Therefore, it is important to be aware of the limitations our target device may have, for example, the amount of available flash memory or RAM. These limitations should be considered when deciding what neural network architecture to use. For example, a model with a large memory footprint may be too big to fit in the available memory on your device.

In this guide, we use the **MobilenetV2 model** architecture from Google. This family of models provides accuracy, a small memory footprint and is provides efficient execution on performance-constrained devices. The MobilenetV2 model architecture is a family of models, which means that it is possible to select a model that fits the restrictions a device may have. This makes it an ideal model choice for an embedded platform.

The trade-off between accuracy, speed, and size requirements is something that must always be considered when choosing what model to use. To give an example of this, our requirements are:

- Inference time under 500ms
- A model size of under 1MB
- To be as accurate as possible

To meet those requirements, we first focused on the model size. The model size is determined by the number of parameters in the model. As mentioned in What happens in an MCU face recognition model? we remove the final classification Conv2D layer of the model after training. This is because we only need the tensors that go into this layer as the fingerprint for our face recognition.

The following image shows the structure of the last few layers of Mobilenet v2.

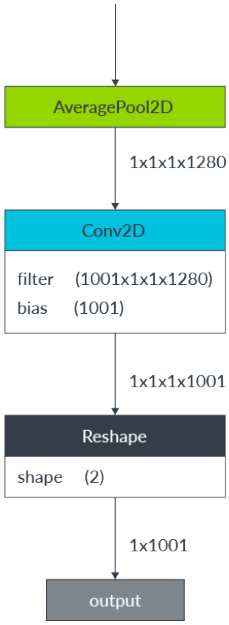


Image 1, Last layers of Mobilenet v2

This last Conv2D layer accounts for (1001x1x1x1280) + 1001 = 1.28M parameters. Therefore, the memory size can be calculated by subtracting 1.28 from the parameters. In our quantized model, each parameter uses 1 byte of memory and our size requirements are for the model to use under 1MB of memory. Therefore, by removing 1.28 from the parameters, we can see which model fits our size requirements. In the memory size column in the table below, this shows that all the models under float_v2_0.75_96 fit our size requirements.

We then must decide on an input resolution. Working with higher resolutions can give us better accuracy but comes at the expense of slower inference times. Inference times for these different input resolutions were measured on an Arm microcontroller-based device like a Cortex-M4 or a Cortex-M7. The highest resolution that satisfied our inference time requirement was selected, which was128 input size.

Note: Both model size and input resolution affect inference time, so it can be worth checking different combinations to find the sweet spot. A smaller model with higher resolution might give us faster inference times with similar accuracy.

The following table shows the results that were obtained when we altered the model version and image resolution:

Different model version and image resolution	Quantized	MACs(M)	Parameters (M)	Memory size	Top 1 accuracy	Top 5 accuracy
float_v2_1.4_224	uint8	582	6.06	4.78	75.0	92.5
float_v2_1.3_224	uint8	509	5.34	4.06	74.4	92.1
float_v2_1.0_224	uint8	300	3.47	2.19	71.8	91.0
float_v2_1.0_192	uint8	221	3.47	2.19	70.7	90.1
float_v2_1.0_160	uint8	154	3.47	2.19	68.8	89.0
float_v2_1.0_128	uint8	99	3.47	2.19	65.3	86.9
float_v2_1.0_96	uint8	56	3.47	2.19	60.3	83.2
float_v2_0.75_224	uint8	209	2.61	1.33	69.8	89.6
float_v2_0.75_192	uint8	153	2.61	1.33	68.7	88.9
float_v2_0.75_160	uint8	107	2.61	1.33	66.4	87.3
float_v2_0.75_128	uint8	69	2.61	1.33	63.2	85.3
float_v2_0.75_96	uint8	39	2.61	1.33	58.8	81.6
float_v2_0.5_224	uint8	97	1.95	0.67	65.4	86.4
float_v2_0.5_192	uint8	71	1.95	0.67	63.9	85.4
float_v2_0.5_160	uint8	50	1.95	0.67	61.0	83.2
float_v2_0.5_128	uint8	32	1.95	0.67	57.7	80.8
float_v2_0.5_96	uint8	18	1.95	0.67	51.2	75.8
float_v2_0.35_224	uint8	59	1.66	0.38	60.3	82.9

Different model version and image resolution	Quantized	MACs(M)	Parameters (M)	Memory size	Top 1 accuracy	Top 5 accuracy
float_v2_0.35_192	uint8	43	1.66	0.38	58.2	81.2
float_v2_0.35_160	uint8	30	1.66	0.38	55.7	79.1
float_v2_0.35_128	uint8	20	1.66	0.38	50.8	75.0
float_v2_0.35_96	uint8	11	1.66	0.38	45.5	70.4

Table 1 Mobilenet v2

4 Quantization

This section of the guide describes how we use quantization in our model. Quantization is the process of approximating a neural network that uses floating-point numbers to one of fixed-point integers. Quantization is applied to our model as it dramatically reduces both the memory requirement and computational cost of running the network.

Deep neural network consists of many parameters which are known as weights, for example, the famous VGG network has over 100 million parameters. In most cases, the bottleneck of running deep neural network is in transferring the weights and data between main memory and compute cores. With quantization, rather than using 32 bits for each weight value, we use just 8 bits. Therefore, the model becomes a quarter of its original size, and we instantly speed up the memory transfer by four times. This also bring other benefits including faster inference time.

We investigated two methods of quantization offered by TensorFlow: **post-training quantization** and **Quantization Aware Training** (QAT). Both methods will produce a fully quantized model where weights and calculations are in fixed-point.

5 Post-training quantization

This section shows how we use post-training quantization in the model.

During post-training quantization, trained model weights are quantized to the required bit width. Also, a small calibration dataset is used to model expected inputs and outputs for the different layers, so that network activations can be successfully quantized. The calibration dataset are samples of what your model expects to see when you deploy it. You can use some data from your training set for this job. This is shown in the following code:

```
import tensorflow as tf
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
def representative_dataset_gen():
    for _ in range(num_calibration_steps):
       # Get sample input data as a numpy array in a method of your choosing
       yield [input]
converter.representative_dataset = representative_dataset_gen
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
converter.inference_input_type = tf.int8
converter.inference_output_type = tf.int8
tflite_quant_model = converter.convert()
```

Post-training quantization was easy to do, and we were able to get a working model from it quickly. However, the accuracy drop was not in the ideal range. On reflection, we believe that the large drop we encountered is linked to our choice of a small model architecture. Therefore, if we had used a bigger network, we expect this drop would have been smaller.

6 Quantization aware training

This section shows how we use quantization aware training in the model.

When using quantization aware training, the quantization of weights and activations is simulated during training. The simulation of weights and activation allows the model to adjust its weights and learn to adapt as well as it can to the quantization that we enforce.

To do this we used TensorFlow's model optimization toolkit, this toolkit includes tools to prepare a `tf.keras` model for Quantization Aware Training (QAT). We used the toolkit to make our trained floating-point model quantization aware and then trained it for several iterations. Training it for several iterations, allowed us to claw back most of the accuracy we had lost when using just post-training quantization. We then used ROC curve to evaluate the performance of our classification problem at various thresholds settings. The curve in the following graph shows this difference, especially at low false acceptance rates:

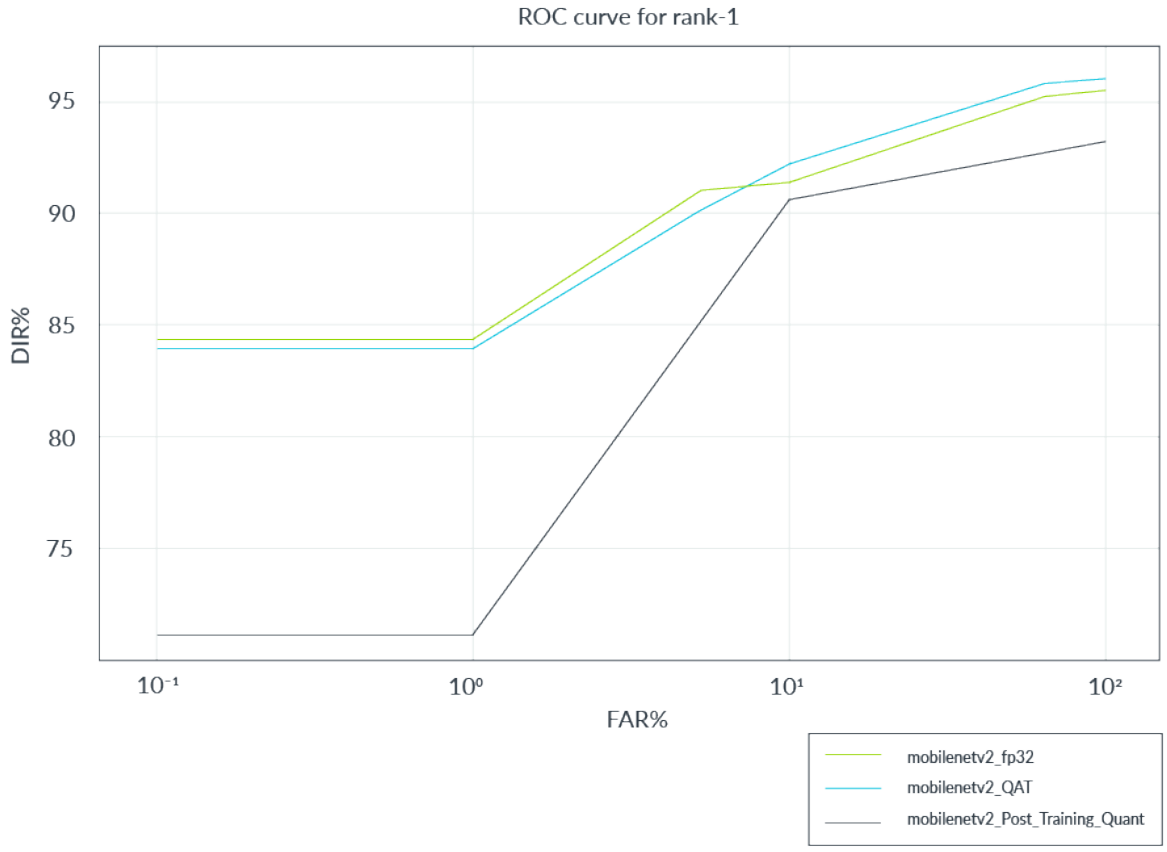


Figure 2: Comparison between fp32 and different quantization methods

Here are some tips for better accuracy when applying quantization aware training to the model*:*

- It is generally better to fine-tune with quantization aware training than to train from scratch.

- Try quantizing the later layers instead of the first layers: For quantization aware training, we found the best results were obtained when we introduced quantization at the end of normal training. We then used this result as a final fine-tuning. When we tried training from scratch, we found it was difficult for the training to even begin to converge.

After training to a suitable accuracy, we removed the final classification layer of the model and fine-tuned it, reserving the already-trained weights. This is because we used the model MobileNet v2 with a 0.5 width parameter in our training, the new last layer of the model produces an output of 640 numbers. This is the feature vector that represents symbolic characteristics of different faces.

7 Related information

Here are some resources related to the material in this guide:

- **Arm architecture and reference manuals**
- **Arm Community** - Ask development questions and find articles and blogs on specific topics from Arm experts.
- **Arm M-profile architecture**
- **Getting started with Arm Microcontroller Resources**
- **TensorFlow Model Optimization Guide**

8 Next steps

This guide shows you the best practices for building an MCU-friendly face recognition model.

You can use this guide to help you build an MCU-friendly face recognition model.

Arm Developer provides **How-to guides** to help you get started with Machine Learning on Arm-based devices. When you have your model ready, you can use **Google TensorFlow Lite for microcontrollers** to run the model on Arm MCUs.