.

IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

Anatomy of Deep Learning Image Classification and Object Detection on Commercial Edge Devices: A Case Study on Face Mask Detection

DIMITRIOS KOLOSOV [1], VASILIOS KELEFOURAS[2], PANDELIS KOURTESSIS[1], AND IOSIF MPORAS[1]

[1]School of Physics, Engineering and Computer Science, University of Hertfordshire (e-mail: d.kolosov2@herts.ac.uk, p.kourtessis@herts.ac.uk, i.mporas@herts.ac.uk)
[2]School of Engineering, Computing and Mathematics, University of Plymouth (e-mail: vasilios.kelefouras@plymouth.ac.uk)

Corresponding author: Dimitrios Kolosov (e-mail: d.kolosov2@herts.ac.uk).

ABSTRACT Developing efficient on-the-edge Deep Learning (DL) applications is a challenging and non-trivial task, as first different DL models need to be explored with different trade-offs between accuracy and complexity, second, various optimization options, frameworks and libraries are available that need to be explored, third, a wide range of edge devices are available with different computation and memory constraints. As such, trade-offs arise among inference time, energy consumption, efficiency (throughput/watt) and value (throughput/dollar). To shed some light in this problem, a case study is delivered where seven Image Classification (IC) and six Object Detection (OD) State-of-The-Art (SOTA) DL models were used to detect face masks on the following commercial off-the-shelf edge devices: Raspberry PI 4, Intel Neural Compute Stick 2, Jetson Nano, Jetson Xavier NX, and i.MX 8M Plus. First, a full end-to-end video pipeline face mask wearing detection architecture is developed. Then, the thirteen DL models were optimized, evaluated and compared on the edge devices, in terms of accuracy and inference time. To leverage the computational power of the edge devices, the models have been optimized, first, by using the SOTA optimization frameworks (TensorFlow Lite, OpenVINO, TensorRT, eIQ) and, second, by evaluating/comparing different optimization options, e.g., different levels of quantization. Note that the five edge devices are evaluated and compared too, in terms of inference time, value and efficiency. Last, we obtain insightful observations on which optimization frameworks, libraries and options to use and on how to select the right device depending on the target metric (inference time, efficiency and value). For example, we show that Jetson Xavier NX platform is the best in terms of latency and efficiency (FPS/Watt), while Jetson Nano is the best in terms of value (FPS/$).

INDEX TERMS image classification, object detection, edge computing, computer vision, performance evaluation

I. INTRODUCTION

ALTHOUGH Deep Neural Networks (DNNs) are nowadays extensively used in a wide range of computer vision applications and hardware platforms [1], their deployment on resource-limited edge devices is not a trivial process, as they are normally both compute and memory intensive [2]. The training phase of the Deep Learning (DL) models is normally held on powerful cloud/remote servers, but the inference phase may be required to run on the edge to address latency and privacy requirements. Running the inference part on an edge device in an efficient way is of critical importance as the trained model is normally run thousands, perhaps even millions, of times [3].

Developing efficient on-the-edge vision AI applications is a challenging task as many different solutions must be explored and evaluated. First, a wide range of IC and OD models are available, providing different trade-offs between accuracy and complexity. We showcase that the most lightweight model is not necessarily the fastest as current compilers and optimization frameworks fail to generate efficient machine code for coprocessors and vector processing units (INT8/INT16 SIMD instructions). Second, different optimization frameworks (e.g., TensorFlow-TensorRT and TensorRT), optimization options (e.g., quantization, multithreading), and libraries (e.g., NVIDIA AI inference libraries, libraries for efficiently reading and processing images) are available and need to be investigated . Third, a wide range of edge devices exist, with diverse hardware architectures, providing trade-offs among latency time, development time, energy consumption, financial cost, efficiency (throughput/watt), and value (throughput/dollar). Fourth, extra engineering effort might be required to optimize the DL models when using vendor-specific tools, e.g., TensorRT supports a specific type of layers only, requiring custom plugins for any custom and/or non-supported layer.

In this paper, we optimize, evaluate and compare seven IC and six OD on-the-edge SOTA models on five commercial off-the-shelf hardware platforms. Note that more models have been investigated and tested here in order to find and select the most suitable for the edge solution, i.e., fast models with adequate accuracy; 11 lightweight and 2 complex models have been selected. The IC models selected are MobileNetV1, MobileNetV2, four different variants of MobileNetV3 and InceptionV3. The OD models selected are SSD-MobileNetV1, SSD-MobileNetV2, SSD-InceptionV2, SSDLITE-MobileNetV2, SSDLITE-MobileNetV3Large, and SSDLITE-MobileNetV3Small. We have selected a wide range of different MobileNet models because they are tailored for edge devices, and thus they present the best solution in terms of latency, by only slightly sacrificing accuracy. The hardware platforms used are the following: Raspberry Pi 4 (RP4), Intel Neural Compute Stick 2 attached to Raspberry Pi 4 (NCS2), NVIDIA Jetson Nano (JNANO), NVIDIA Jetson Xavier NX (JXAVIER), and i.MX 8M Plus (IMX8P).

The SOTA models were first fine-tuned in the TensorFlow framework and then optimized by using the following SOTA frameworks for each corresponding hardware platform: TensorFlow Lite, Intel OpenVINO, TensorFlow-TensorRT, NVIDIA TensorRT, and NXP eIQ. All different possible quantization levels are evaluated for each model and hardware platform. We show that performance does not always align with the quantization level (in this paper latency and performance are used interchangeably). Furthermore, we have enabled other optimizations too, where possible, e.g., multithreading. In Section IX, we provide insightful observations on which optimization options and frameworks to use in each case.

Furthermore, we compare the diverse hardware platforms in terms of inference time, efficiency, and value. We provide important insight about which models, frameworks, and optimization options to use for each hardware platform as well as which platform to use depending on the target metric. We show that JXAVIER is the best option in terms of latency and efficiency, while JNANO is the best option in terms of value.

Our use case consists of an IC and OD face mask wearing detection application. We have chosen this application as COVID-19 still negatively impacts our lives and vision-based AI technology can mitigate the problem with such a use case with video analytics and monitoring. Note that the performance of the entire image data path is evaluated, including the pre/post-processing steps and loading of the DL model.

The experimental results show that for IC, MobileNetV3-Small Minimalistic / MobileNetV3-Small models are the most efficient in terms of latency while for ID, MobileNetV3-Large/SSD-InceptionV2 models in terms of accuracy. The optimization tools can provide up to 6.8x/16.4x/15.9x/56.4x/36.0x times faster inference on IC models and up to 2x/9x/4x/9.3x/80.5x times faster inference on OD models, for RP4/NCS2/JNANO/JXAVIER/IMX8P, respectively.

The contributions of this paper are as follows:

(1) Optimization, evaluation, and comparison of seven IC and six OD on-the-edge SOTA models, in terms of accuracy and latency, on five commercial off-the-shelf edge devices.
(2) An evaluation of TensorFlow Lite, OpenVINO, TensorFlow TensorRT, TensorRT, and eIQ optimization frameworks and their main optimization options on five edge devices.
(3) A comparison between Raspberry Pi 4, Intel Neural Compute Stick 2, NVIDIA Jetson Nano, NVIDIA Jetson Xavier NX, and NXP i.MX 8M Plus hardware platforms in terms of value (FPS/price), and efficiency (FPS/power) metrics when running IC and OD applications.
(4) A face mask detection machine learning architecture is developed.
(5) Easily reproducible open-source benchmarking templates are delivered that only use publicly available vision libraries.

It is important to note that for the first time such a high number of hardware platforms, frameworks, and IC/OD models have been benchmarked and compared, not only on model latency performance but the full video pipeline.

The remainder of this paper is organized as follows. Firstly Section II reviews related work. In Section III, the proposed face mask architecture is presented. In Sections IV, V, and VI, the DL models, optimization frameworks, and edge devices studied and discussed, respectively. In Sections VII and VIII, the experimental setup and experimental results are presented, respectively. Section IX is dedicated to discussion, and finally, Section X to conclusions and future work.

II. RELATED WORK

Deploying efficiently AI applications on edge devices poses various challenges like discussed in [4], specifically constraints around compute, memory, and power consumption. To tackle these, quantization and weight pruning [5] are two popular techniques that normally trade a slight reduction in accuracy for performance gains. In quantization, the neural network weights and/or the feature maps are expressed by using shorter data types, such as FP16, INT16, or INT8 instead of FP32 [6]; this leads to a lower memory footprint as well as to a lower latency as the computation cost is reduced and the SIMD instructions can be used to calculate more operations per instruction. In weight pruning [7], neurons with small saliency (sensitivity) are removed, resulting in a sparse computational graph [8]; neurons with small saliency are those whose removal minimally affects the model output/loss function.

There are various deep convolution neural network models that vary in terms of accuracy and number of parameters. For deep learning IC applications, if the target edge device is compute and memory limited and frames per second (FPS) is a more important metric than accuracy, then a lightweight model is preferred, such as EfficientNet [9], MobileNets, SqueezeNet, ShuffleNet, PeleeNet, MnasNet, and OFA [10]. These models adopt various innovative techniques to reduce the number of parameters and operations per second while maintaining satisfactory accuracy. In general, MobileNets proved to be tailored for edge devices with limited computation and memory resources with improvements across versions v2 and v3 seen only on ARM-based hardware as we confirmed in our work. While more complex models such as InceptionV3 are more appropriate for applications needing high accuracy but require a dedicated AI co-processor for reaching high FPS performance. As far as the on-the-edge deep learning OD models are concerned, there are one-stage (e.g., SSD [11], YOLO [12]) and two-stage detectors (e.g., FPN [13], Mask R-CNN [14], Faster R-CNN [15]). Two-stage detectors focus on achieving high localization and object recognition accuracy at the expense of requiring high compute capabilities, while the one-stage detectors focus on achieving high inference speeds with lightweight architectures. In this case study, one-stage SSD (single shot detection) type models were used. SSD tends to be more resource efficient and outperforms other types (such as R-CNN, Fast R-CNN, Faster R-CNN) because the tasks of object localization and classification are done in a single forward pass of the network [11].

To allow for the computation-intensive DL models to efficiently run on the edge, various hardware platforms (accelerators) have been introduced such as NVIDIA Jetsons (CUDA cores), Intel NCS2 (Vision Processing Unit), Google Edge TPU (ASIC), and Neural Processing Unit (NPU) of i.MX 8M Plus. Accelerators offer various benefits such as energy efficiency, ultra-low latency, and lower costs, that enable new applications for building sensory systems in the real world that were not possible previously [16]. FPGAs are also present and are an excellent choice for custom DL implementations because of their power efficiency, latency, throughput, flexibility in interfaces and reconfigurability [17]–[19]. This diverse and ever-growing complexity of modern on-the-edge hardware architectures has introduced optimization frameworks to keep pace with hardware advancements and effectively use dedicated resources. NVIDIA provides TRT, Intel provides OpenVINO, while TFLITE is well optimized for ARM microcontrollers and microprocessors. The disadvantage with using these types of accelerators is that you may be limited either in software or hardware in deploying specific data types like FP16/INT8 or specific layers.

A large number of studies have been published evaluating DL IC and/or OD models on edge devices. Reference [20] investigates the on-the-edge inference of DNNs in terms of latency, energy consumption, and temperature, on five different hardware platforms; unlike the proposed method, this work does not take advantage of the optimization frameworks we have investigated. In [21], an in-depth benchmark analysis of three embedded platforms is performed for image vision applications including MobileNet and InceptionV2; in [22], EDLAB is delivered, an end-to-end benchmark to evaluate the overall performance of three image classification and one object detection models across Intel NCS2, Edge TPU and Jetson Xavier NX. In [23], a performance analysis of the edge TPU board is provided for object classification. In [24], NVIDIA Jetson Nano and Google Coral Dev Board are evaluated. In [25], a survey on DL object detection methods is presented. In [26], a survey of DL methods and software tools for IC and OD is presented. In [10], a review of SOTA object detectors and lightweight classification architectures is delivered, without exploring performance on edge hardware. None of the above provide this number of models, edge devices, and optimization options for end-to-end analysis.

In [27], the inference time of 14 IC DL models is evaluated by using the OpenVINO toolkit but using a workstation utilised Intel Xeon CPU and integrated Iris Pro GPU. In [28], a framework to deploy DL-based applications in fog-cloud environments is presented. In [29], the performance and energy consumption of three commercial devices is evaluated for DL inferencing. Reference [30] implements and evaluates real-time target detection and tracking on Intel NCS2 and NVIDIA Jetson TX /AGX via a drone. Reference [31] explores problems in computer vision applications and presents the OpenVINO toolkit as a solution for bringing AI to the edge but does not apply it and explore it on edge hardware. The TFLITE and TF-TRT optimization frameworks are analysed in terms of throughput, latency, and power consumption in [32]. In [33], TensorFlow Lite Micro is presented to address the deployment of DL on MCUs. Lastly, [34] compares edge deployment of lightweight models on Google Coral, Intel NCS2, and NVIDIA Jetson Nano for a specific use case, classification of waste.

Last, a group of studies has been published evaluating DL face mask detection applications. In [35]–[37], three face mask detection architectures are developed and evaluated solely on PCs. In [38], one-stage and two-stage approaches are presented for face mask detection, with only accuracy being evaluated and not their performance for edge devices. Finally, NVIDIA published a GitHub repo [39] on how to train, optimize and deploy a face mask detection application on their Jetson hardware using their Transfer Learning Toolkit (TLT) and DeepStream SDK, but it is not applicable for other types of edge devices.

Compared to all the previously mentioned related work, we have explored 7x image classification and 6x object detection models on 5x edge devices with a specific use case in mind, far greater than any of the other literature. Additionally, unlikely most of the related work, we have explored the frameworks/compilers of each target hardware and how the quantization/optimization affects the performance of the whole end-to-end pipeline, and not just the inference times of the model. We have not only evaluated the latency of each stage of the pipeline but included other metrics such as efficiency (FPS/Max Power) and value(FPS/Cost) that provide a different perspective on what each type of technology has to offer.

III. FACE MASK DETECTION ARCHITECTURE

A generic software block diagram of the proposed face mask detection architecture is shown in Fig. 1; two different methods are explored, an IC-based (Method1) and an OD-based (Method2). The process starts with reading locally stored images and finishes with overlaying the results of the face mask detections on the original frame. Table 1 shows the inputs/outputs of the two methods. Note that the input data needs to be in the same format that the DL model was trained on, along with the correct interpretation of the results.

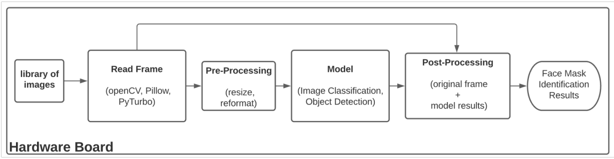

FIGURE 1. Face Mask Detection Block Diagram

A. METHOD1 (IC)

In order to detect if there is a face mask or not present in the current frame, we used image classification. The data pipeline consists of reading the frame, pre-processing the frame into the right resolution, applying appropriate pixel normalisation (based on the model and the data it was trained on), and then the execution of the model. Lastly, a preprocessing step overlays the label and corresponding confidence results onto the original image at its original resolution.

B. METHOD2 (OD)

To detect all faces with and without face masks, along with bounded boxes showcasing their location in the current frame, we used object detection. This was similar to Method1, but the models are more complex and further post-processing is required to overlay detection boxes onto the original frame.

The pre-processing phase consists of two main steps. First, the input image is resized to the resolution that the model has been trained on. For example, the input images must be resized to 224x224 for Method1 and to 300x300 for Method2. Although all the images are resized to the same resolution (for a specific architecture), the bigger the input image is, the higher the model's accuracy is, as the images are resized by using interpolation. In the second step, the input image is converted into the right colour format (e.g., RGB), data format (e.g., Batch x Width x Height x Channels (BWHC)), and applied with the corresponding pixel normalisation which depends on the type of model and the data it was trained on.

TABLE 1. Inputs/Output of Method1 and Method2

Method1: IC	Input	Batch x Width x Height x Channel (1x224x224x3)
	Output	Confidence % of each class (2)
Method2: OD	Input	Batch x Width x Height x Channel (1x300x300x3)
	Output	Classes, Confidence %, box coordinates [x1,x2,y1,y2], number of detections

The post-processing step aims to show the label of the detected class for Method1 and add rectangles/labels around the detections on top of the original image for Method2. It is important to note that the execution time of the post-processing step depends on the number of faces identified (in this case study, we always assume just one face).

Note that we have studied and evaluated all the video pipeline stages in Fig. 1, and not just the execution time of the model, by using three different input image resolutions, 13 DL models, and various libraries/frameworks. This allows for a better evaluation of the selected hardware platforms. Furthermore, this application tackles a real-life problem and showcases various on-the-edge solutions, depending on the technical requirements and performance needs.

IV. DEEP LEARNING MODELS

A. METHOD1 (IC)

Seven SOTA pre-trained models from TensorFlow 2 have been used for Method1 as shown in Table 2. Each model was pre-loaded with weights based on ImageNet, a large dataset consisting of 1.4M images and 1000 classes [40]. The base of each model was frozen, while fine-tunable GlobalAveragePooling2D, Dropout, and a SoftMax activation were added as the last layers to predict the two target classes. All MobileNets were trained with the alpha value set to 1; alpha values control the width of the network, which proportionally reduces or increases the number of filters of each layer. This allows for further customizing the MobileNet models, offering different trade-offs between latency and accuracy.

The models that we used for Method1 are shown hereafter:

- M1: MobileNetV1 [41]. M1 was introduced by Google in 2017 as a lightweight and efficient architecture for generic embedded vision applications aimed at the mobile industry. M1 uses depthwise separable convolutions instead of standard convolutions which offer improvements in terms of latency and model size. Its ImageNet accuracy is 70.6%.

- M2: MobileNetV2 [42]. M2 is the successor of M1. It achieves fewer arithmetical instructions and lower memory size than M1. Its ImageNet accuracy is 72.0%.
- M3: MobileNetV3 Large [43]. M3 is the successor of M2. It achieves fewer arithmetical instructions and higher accuracy than M2. The "large" variant is aimed for high resource / high accuracy use cases, with ImageNet accuracy of 75.6% [44].
- M4: MobileNetV3 Large Minimalistic. M4 is the "minimalistic" version of M3, which has the same per-layer dimensions characteristic as MobileNetV3 however, they don't utilize any of the advanced blocks [45], with ImageNet accuracy of 72.3% [44].
- M5: MobileNetV3 Small. M5 is the "small" variant of MobileNetV3 that is aimed at low resource use cases, with ImageNet accuracy of 68.1%. [44].
- M6: MobileNetV3 Small Minimalistic. M6 is the "minimalistic" version of M5, with ImageNet accuracy of 61.9% [44].
- M7: InceptionV3 [46]. InceptionV3 is a widely used image recognition model that has been shown to attain greater than 78.1% accuracy on the ImageNet dataset. Compared to the MobileNets, it is of higher complexity / trainable parameters, which make it more accurate but also computationally more demanding.

B. METHOD2 (OD)

For Method2, six SOTA pre-trained COCO models have been used from TensorFlow 1 Detection Model Zoo as shown in Table 2. The last feature layers that generate bounding boxes or locations of the target classes are based on the Single Shot Detection (SSD) [11] architecture. This single-stage approach offers competitive accuracy and is faster than methods such as the multi-stage R-CNN, Fast R-CNN and Faster R-CNN [15], which are based on regional proposal networks and are computationally intense. This makes the SSD-type object detectors better suited for edge deployment.

The models we used for Method2 did not have their architecture modified in any way and are shown hereafter:

- O1: SSD-MobileNetV1 [41]. O5 is an M1 variant for object detection. Its mean COCO Average Precision (mAP) is 21% [47].
- O2: SSD-MobileNetV2 [42]. O2 is an M2 variant for object detection. O2 has a higher COCO mAP value than O1 (22% [47]), but also fewer parameters than O1.
- O3: SSD-InceptionV2 [46]. O3 is a more accurate (COCO mAP 24% [47]) object detection model than O1, O2, and O4-O6, but with larger memory size, and higher computational complexity. We have used this model to evaluate the performance of more complex models on edge devices.
- O4: SSDLITE-MobileNetV2 [42]. O4 is an optimized version of O2. All the regular convolutions of O2 have been replaced by separable convolutions and therefore O4 achieves the lowest computational complexity amongst O1-O4 with a COCO mAP of 22% [47].
- O5: SSDLITE-MobileNetV3 Large [43]. O5 is an M3 variant for object detection with COCO mAP of 22.6% [47].
- O6: SSDLITE-MobileNetV3 Small [43]. O6 is an M5 variant for object detection with COCO mAP of 15.4% [47]. O6 is less accurate than O5 but it uses a smaller model size.

TABLE 2. IC and OD Models used

	Model	Parameters	Size	Version
M1	MobileNetV1	3.23M	13.2MB	TF2.5.0
M2	MobileNetV2	2.26M	9.5MB	
M3	MobileNetV3 Large	4.23M	17.8MB	
M4	MobileNetV3 Large Minimalistic	2.67M	11.3MB	
M5	MobileNetV3 Small	1.53M	6.8MB	
M6	MobileNetV3 Small Minimalistic	1.03M	4.6MB	
M7	InceptionV3	21.81M	88.1MB	
O1	SSD-MobileNetV1	5.51M	22.7MB	TF1.15.3
O2	SSD-MobileNetV2	3.87M	16.4MB	
O3	SSD-InceptionV2	13.3M	54.0MB	
O4	SSDLITE-MobileNetV2	3.01M	13.1MB	
O5	SSDLITE-MobileNetV3 Large	2.17M	9.6MB	
O6	SSDLITE-MobileNetV3 Small	0.93M	4.4MB	

V. EDGE DEVICES

Method1 and Method2 have been trained on a powerful desktop PC and evaluated on five commercial off-the-shelf hardware platforms in terms of inference time, efficiency and value. Although it is meaningless to run the face mask detection application on a PC, it is used as a point of reference to better evaluate the performance of the edge hardware platforms.

The hardware platforms are listed below from the least powerful to the most powerful:

(A) **Raspberry PI 4**
The main computing element of Raspberry Pi 4 (RP4) is a quad-core ARM Cortex-A72 64-bit CPU that supports NEON 128-bit wide vector instructions, running at a maximum clock speed of 1.5GHz. The CPU is connected to a 4GB LPDDR4 memory. RP4 costs about $62 and its maximum power consumption is 9 Watts. It included Raspbian 10.7 OS, TensorFlow 2.5.0 (cp37-linux_armv7l) and tflite-runtime 2.5.0.

(B) **Intel Neural Compute Stick 2 attached to RP4**
The Intel Neural Compute Stick 2 (NCS2) is a deep learning inference development kit; NCS2 takes advantage of Intel Movidius Myriad X Vision Processing Unit (VPU)). The Myriad X includes 16 low-power vector processing units 128-bit wide (a.k.a. SHAVE), running at 700MHz. NCS2 costs about $70 and its maximum power consumption is 2 Watts. NCS2 is not a standalone platform as it is a USB stick. NCS2 USB stick has

TABLE 3. Edge Devices Specifications

HW	*CPU*	*Memory*	*GPU*	*Max Power Consumption*	*Price*
RP4	Cortex-A72	4GB LPDDR4	VideoCore VI	9W	75 USD
NCS2	N/A	500MB Internal	N/A	2W	99 USD
JNANO	Cortex-A57	4GB LPDDR4	128-core NVIDIA Maxwell	10W	99 USD
IMX8P	Cortex-A53	6GB LPDDR	HiFi4 DSP + NPU	15W	449 USD
JXAVIER	Carmel v8.2	8GB LPDDR	384-core NVIDIA Volta + 48 Tensor cores	20W	399 USD
PC	i9-9900K	48GB DDR4	NVIDIA GTX1060 6GB	600W	2000 USD

been used as an accelerator and it has been attached to a Raspberry PI 4 via USB 3.0.

(C) **NVIDIA Jetson Nano**
Jetson Nano (JNANO) includes an embedded GPU with 128 CUDA cores, a quad-core ARM Cortex-A57 64-bit CPU and 4GB LPDDR4. JNANO costs about $99 and its maximum power consumption is 10 Watts. It runs Ubuntu 18.04.5 LTS and uses Python 3.6.9, CUDA 10.2, TensorRT 7.1.3.0 and Jetpack 4.5.1. Multiple power modes are supported including trade-offs between the number of CPU cores being used and their operational frequency. We used the power mode MAXN (10 Watts) where the 4 CPU cores run at 1.48GHz and the GPU at 921.6MHz.

(D) **NXP i.MX 8M Plus**
i.MX 8M Plus (IMX8P) board has been released in 2021. It includes a quad-core ARM Cortex-A53 running at 1.8GHz, an ARM Cortex M7, a HiFi4 DSP running at 800Mhz, LPDDR4, and most importantly a Neural Processing Unit (NPU). The NPU includes several hardware features such as 128-bit vector engines and tensor processing cores. IMX8P costs about $449 and its maximum power consumption is 15 Watts. The OS is Yocto 5.10.52-lts-5.10.y+ga11753a89ec6, using Python 3.9.5 and tflite-runtime 2.5.0.

(E) **NVIDIA Jetson Xavier NX**
Jetson Xavier NX (JXAVIER) is more powerful than JNANO, as it includes more GPU cores, a more powerful CPU, 8GB LPDDR4, and two low-power Deep Learning Accelerators (DLAs). In particular, its GPU includes 384 cores and 48 Tensor Cores, while its CPU is a 64-bit 6-core NVIDIA Carmel ARMv8.2. The DLA comprises several IP-core models which are configurable and achieve 4.5TOPS, each. Note that the DLA has not been designed to provide better inference time, but lower power consumption instead. JXAVIER costs about $399 and its maximum power consumption is 15 Watts (the latest Jetpack 4.6 pushes this to 20Watts maximum). It runs Ubuntu 18.04.5 LTS and uses Python 3.6.9, CUDA 10.2, TensorRT 7.1.3.0 and Jetpack 4.5.1. Power mode 2 is used (15 Watts), where the 6 CPU cores run at 1.42GHz and the GPU runs at 1.11GHz GPU.

(F) **Intel i9-9900K CPU (PC)**
The PC supports an 8-core Intel i9-9900K CPU, an NVIDIA GTX1060 6GB GPU, 48GB DDR4-2666, 1TB SSD hard drive, and Ubuntu 18.04 LTS. We have also used Python 3.6.13 and OpenCV-Python 4.5.3.56. The PC costs about $2000 and its maximum power is 600 Watt.

NEON, SHAVE AND IMX8P VECTORIZATION ENGINES

To better understand how the DL models run on the hardware platforms and better understand Section VIII, we provide a brief explanation of vectorization, a key processor feature that boosts performance. Modern processors support extra hardware units to realize vector / Single Instruction Multiple Data (SIMD) instructions; this feature allows for the processing of multiple image pixels. In our case, by using a single instruction; a single CPU core executes multiple operations in a single instruction (a.k.a. SIMD).

RP4 supports NEON 128-bit wide instructions, NCS2 supports SHAVE 128-bit wide instructions, while the PC supports AVX 256-wide instructions. All processors support a rich instruction set including 8-bit, 16-bit, 32-bit, and 64-bit operations, e.g., 128-bit instructions can process either 16x 8-bit values, 8x 16-bit values, 4x 32-bit values, or 2x 64-bit values, in a single instruction, boosting performance. This is the main reason that quantization improves performance.

However, nowadays compilers are not smart enough to convert DL applications to machine code that efficiently uses the right vector instructions in all cases, and therefore manually vectorized code versions or optimized libraries, are needed. This is because first, data dependencies in the code make the vectorization process less efficient and therefore manual changes are needed to fully exploit the wide instructions, and second, different vector instructions include different latency/throughput values. As a result, different implementations of the same model give significant variations in performance.

VI. OPTIMIZATION FRAMEWORKS

The hardware architectures of the edge devices are diverse and heterogeneous, including more than one type of coprocessors, such as GPUs, SIMD units, and DL accelerators. As it was explained in Section V, to take advantage of these powerful coprocessors, hardware-specific optimization frameworks are needed.

The SOTA optimization frameworks used (Table 4) are the following:

(A) **TensorFlow Lite (TFLITE) for RP4 and IMX8P**
TensorFlow Lite [48] is TensorFlow's lightweight solution for mobile and embedded devices. For ARM-based hardware, TFLITE has integrated XNNPACK [49] which takes advantage of ARM NEON vector processing unit but also supports several HW accelerators. It enables low-latency inference of on-device machine learning models with a small binary size and fast performance. TFLITE supports quantization with FP16, DINT8, and INT8 data formats and the latest version of TFLITE runtime engine is able to perform multi-threaded execution.
TFLITE tools were used to optimize M1-M7 and O1-O6 on RP4 and IMX8P post-training. The new optimized models are quantized from 32-bit Floating Point (FP32) numbers to FP16, dynamic INT8 (DINT8), and 8-bit integers (INT8). This results in a smaller memory footprint (less memory is required for the model) and faster computations. 8-bit computations can be executed faster than 32-bit computations if the appropriate vector instructions are used (see Section V above).
In FP16 quantization, 5bits are used for the exponent, and 10bits are used for the mantissa, while in FP32 8bits are used for the exponent and 23bits for the mantissa. The other two supported types of quantization are full 8-bit quantization (INT8) and dynamic range 8-bit quantization (DINT8). In INT8, quantization is applied to both the activations and the tensor weights. In DINT8, the weights are quantized post-training to INT8, and the activations are quantized dynamically at the inference phase. Thus, DINT8 comes with an extra computation overhead, but more efficient than FP16 for the appropriate hardware.

(B) **eIQ (TFLITE) for IMX8P**
eIQ is a software development environment with various tools that help with the development of AI applications targeted for NXP MCUs or CPUs [50]. It is incorporated with DeepView ML Tool suite [51] that allows developers to use a graphical interface to label datasets and train and deploy AI solutions for NXP silicon. It includes a model optimizer utility, inference engines, NN compilers, libraries, and hardware abstraction layers that support TensorFlow Lite, Glow, Arm NN, and Arm CMSIS-NN. eIQ has been used to optimize and deploy M1-M7 and O1-O6 models to FP16 and INT8 data types to be compared versus the models derived from TFLITE tools.

(C) **OpenVINO for NCS2**
OpenVINO [52] is an optimization framework that focuses on optimising and deploying DL models on Intel hardware platforms, ranging from the edge to the cloud. OpenVINO can be used to optimize pre-trained models derived from TensorFlow, PyTorch, or other popular frameworks. OpenVINO v2021.4 has been used to optimize M1-M7 and O1-O6 for NCS2. FP16 quantization is used as that is the only data type supported by NCS2.

(D) **TensorFlow-TensorRT (TF-TRT) for JXAVIER and JNANO**
TF-TRT [53] is an optimization framework dedicated to GPUs. It is the integration of the TensorFlow framework with NVIDIA's TensorRT. TF-TRT performs several optimizations to the compatible Neural Network (NN) graphs such as eliminating layers with unused outputs and fusing, where possible, convolution, bias, and ReLU layers to form a single layer. The incompatible graphs and unsupported layers do not take advantage of TRT and are left in their original FP32 implementation. TF-TRT supports FP16 and INT8 quantization (JXAVIER only). TF-TRT has been used to optimize M1-M7 and O1-O6 on JNANO and JXAVIER with FP16 quantization. Note that TF-TRT requires a significant amount of extra storage memory but the hardware device might not have this amount of free memory. TF-TRT has been used to optimize M1-M7 and O1-O6 on JNANO and JXAVIER with FP16 quantization.

(E) **TensorRT (TRT) for JXAVIER and JNANO**
TensorRT (TRT) is the NVIDIA software development kit for delivering high-performance deep learning inference on GPUs [54] and does not require the TensorFlow library. It is used to optimize already trained models and run them efficiently on NVIDIA devices. TRT has been used to further optimize M1-M7 and O1-O6 on JNANO and JXAVIER. It provides better latency times than TF-TRT, as the entire CNN graph is optimized as a single component (not layer by layer); this means that it is not possible to optimize unsupported layers. Note that for the non-supported layers, custom plug-ins are required, which makes its usage less user-friendly. It supports FP16, INT16, and INT8 quantization (JXAVIER only). TRT has been used to further optimize M1-M7 and O1-O6 on JNANO and JXAVIER. It supports FP16 and INT8 quantization (JXAVIER only).

TABLE 4. Hardware/Framework used datatypes

HW	***TF***	***TFLITE***	***eIQ***	***OpenVINO***	***TF-TRT***	**TRT**
PC	FP32	-	-	-	-	-
RP4	FP32	FP16, DINT8, INT8	-	-	-	-
IMX8P	-	FP16, DINT8, INT8	FP16, INT8	-	-	-
NCS2	-	-	-	FP16	-	-
JNANO	FP32	-	-	-	FP16	FP16
JXAVIER	FP32	-	-	-	FP16	FP16, INT8

VII. EVALUATED DATASETS

Two datasets have been used for IC and two for the OD method. The datasets consist of 2 classes; images of people wearing and not wearing a face mask. Note that the aim of this research work is not to find the datasets that maximize the models' accuracy, but do have enough samples to provide adequate detection results.

The datasets used for the IC models are shown below:

- Dataset1: d1 [55]. Total of 1376 images have been used; 690 with mask and 686 images without mask.
- Dataset2: d2 [56]. Total of 4095 images have been used; 2165 with mask and 1930 images without mask.

The datasets used for the OD models are shown below:

- Dataset3: d3 [57]. Total of 853 images have been used; 3232 labels with mask and 717 labels without mask. Labels with incorrectly wearing masks were removed due to low count of samples.
- Dataset4: d4. Total of 1619 images have been used; 3232 labels with mask and 2014 labels without mask. Dataset3 provided poor results for people without wearing a mask and therefore FDDB [58] was added to improve its accuracy.

VIII. EXPERIMENTAL RESULTS

The experimental results section is partitioned into three subsections. In Subsection A, the metrics used for comparison between models and edge device performance are described. In Subsection B, the models' accuracy is evaluated. In Subsection C, the inference time of all the face mask detection application steps (Fig. 1) is evaluated for both methods. Furthermore the edge devices are benchmarked and compared in terms of inference time, value and efficiency.

A. METRICS

The metrics that were used in this paper to showcase the performance of the edge devices are listed hereafter:

a: Accuracy - IC

To evaluate and understand how the IC models performed in terms of predicting, four performance metrics were used, accuracy, precision, f1-score and recall as per (1), (2), (3), and (4) [59] respectively. These were calculated through the below confusion metrics:

– (TP) True Positives: Correctly predicted positive values
– (TN) True Negatives: Correctly predicted negative values
– (FP) False Positives: Falsely predicted positive values
– (FN) False Negatives: Falsely predicted negative values

$$accuracy = \frac{TP+TN}{TP+FP+FN+TN} \tag{1}$$

$$precision = \frac{TP}{TP+FP} \tag{2}$$

$$recall = \frac{TP}{TP+FN} \tag{3}$$

$$f1-score = \frac{2 \times recall \times precision}{recall + precision} \tag{4}$$

b: Accuracy - OD

The COCO method was used to evaluate the detection performance of the OD models, which has 12 different metrics based on mean Average Precision (mAP) and mean Average Recall (mAR) [60]. Besides low-level metrics in object detection such as the four confusion metrics, another important one used is the Intersection of Union (IoU), which evaluates the overlap between the ground-truth label and the predicted object in terms of area. The main challenge metric was mAP or mAP@[.5:.05:.95], which calculates the average across multiple IoU values.

c: Latency

To accurately extract the execution time of the model, the inference phase is run multiple times and the average time is taken. The overall execution time was at least one minute; this is because apart from this software process, other OS pocesses use the hardware resources too (such as CPU cores, cache memory, etc) and they add 'noise' to our experimental results; by running the target process for about one minute, the 'noise' is minimized.

d: Value

Value is calculated by (5), where FPS is the number of processed frames per second and price is the financial cost of the hardware board in US dollars.

$$Value = \frac{FPS}{Price} \tag{5}$$

e: Efficiency

Efficiency is calculated by (6), where FPS is the number of processed frames per second and power is the maximum power consumption of the board. In our future work, we are planning to measure power consumption by using power meters.

$$Value = \frac{FPS}{Power} \tag{6}$$

B. MODEL ACCURACY EVALUATION

In this subsection, the accuracy of the IC and OD models is evaluated, in Fig. 2 and Fig. 3, respectively. For both methods, the models were fine-tuned with the datasets mentioned in Section VII.

a: IC Training Results

All the IC models had relatively similar training times which were dependent on the size of the dataset and the model, which on average was 2:13 minutes for Dataset1 (1376 images) and 6:06 minutes for Dataset2 (4095 images). Dataset1

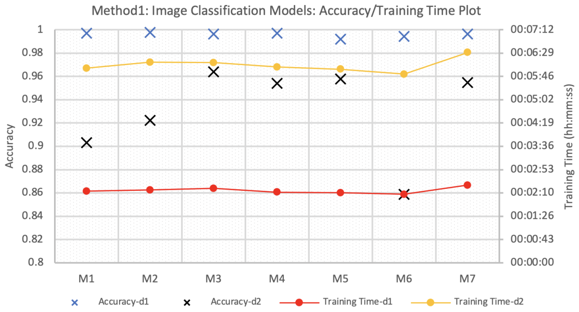

FIGURE 2. Image Classification Models: Accuracy/Training Time Plot (20 epochs)

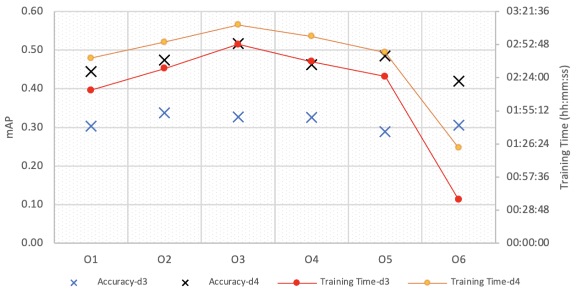

FIGURE 3. Object Detection Models: Accuracy/Training Time Plot (20k epochs)

gave high results in terms of accuracy across all models, but in Dataset2 we can observe better how various models behave with a much larger pool of images. MobileNets showed improvement in accuracy from V1 to V2 to V3, with expected drops of accuracy seen in the minimalistic versions of MobileNetV3, but with a significant drop in M6. The most complex model M7 (21.81M parameters) was not the most accurate, which shows that the most complex model might not be the best choice for a specific use case. Overall, M2 had the highest accuracy (99.78% with 2.26M parameters) for Dataset1 and M3 (96.38% with 4.23M parameters) for Dataset2. Detailed training results with accuracy, precision, f1-score, and recall metrics can be found in Table 5 (Appendix).

b: OD Training Results

The ID models had a much longer training time compared to IC, which was expected as the models were more complex; there could be more than one ground truth label per image, hence more output variables to compute. Dataset3 training times were on average 2:13 hours, while Dataset4 was 2:39 hours. Dataset3 had a low count of faces not wearing a mask, which resulted in low COCO mAP for that class. Dataset4 had the addition of the FDDB dataset, which resulted in considerable improvements across all models. All models across Dataset3 had similar results, which was due to faces not wearing a mask class bringing down the average. Better representation of how models behaved can be seen in the results from Dataset4; the most complex model was the most accurate (O3 with 13.3M parameters), and the least complex was the least accurate (O6 with 0.93M parameters). Overall, the most accurate model in terms of mAP(IoU.50:.05:.95), was O2 (34%) for Dataset3 and O3 (52%) for Dataset4. Detailed training results across various IoU thresholds for mAP and mAR can be found in Table 6 and 7 of the Appendix.

The datasets used for training the models provided sufficient results when tested with test data and/or a live video feed, hence no more effort was put into improving the accuracy, as the focus of this work was intended for metrics around the hardware platforms.

C. EVALUATION OF THE INFERENCE TIME AND EDGE DEVICES

In this subsection, the inference time of all the application steps (Fig. 1) are evaluated on all hardware platforms. The overall runtime of the inference part is given by (8):

$$WT = 5 \times (RF + PreP + L) \tag{7}$$

$$RT = LT + WT + F \times (RF + PrP + L + PoP) \tag{8}$$

where 'LT' (Loading Time) is the time needed to load the DL model and its parameters, 'F' (Frames) is the total number of frames being processed, 'RF' (Read Frames) is the time needed to decode and load the input image to the processor's memory, 'L' (Latency) is the execution time of the DL model and 'PrP/PoP' (Pre-Processing/Post-Processing) is the time taken to apply a mandatory pre-processing/post-processing step. Due to the first inference cycles of the model being longer than usual due to requiring to initialise model and weights, 'WT' (Warmup Time) is run for 5 cycles to remove that "noise" from the following benchmarking steps. Three different input image sizes have been used, i.e., 640x360 (R1), 1280x720 (R2), 1920x1080 (R3). The process of loading the DL model (LT) is applied just once, while the rest of the steps are applied for each input image (frame). Note that the value and efficiency metrics in this paper include the time needed to read and pre/post-process the image.

1) Evaluation and optimization of the time needed to read the input image

The 'ReadFrame' time is the second most computationally expensive routine (the most expensive is running the DL model). This process includes a significant proportion of the overall execution time, especially for large input images, and therefore it needs to be optimized. Note that this process is executed by the CPUs of the edge devices and not by the powerful coprocessors or other dedicated hardware resources.

An evaluation of different Python libraries has been made for three image sizes (Fig. 4). For each size, the image is read multiple times and the average execution time values are taken (about 60 seconds each). By using OpenCV2 library

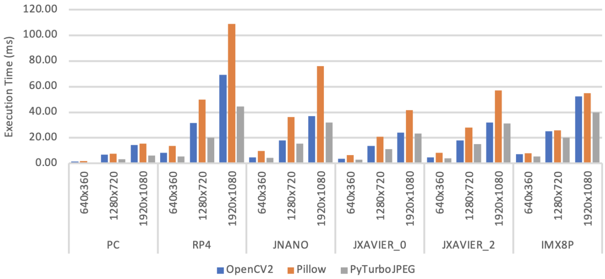

FIGURE 4. Evaluation of different Python libraries that read the input image

[61], the average reading time of an image ranges between 1.55-14.5 milliseconds (ms) for the PC, 8.3-69 ms for the RP4, 4.8-36 ms for the JNANO, 3.8-24 ms for the JXAVIER (power mode 0), 4.8-32 ms for the JXAVIER (power mode 2) and from 7 to 52 ms for IMX8P. JXAVIER supports five different power modes to provide different performance vs power solutions. Power mode 0 uses just 2 out of 6 CPU cores running at 1.9GHz, while power mode 2 uses all the 6 cores but their frequency is lower (1.4GHz). The process of reading the input image is executed on a single core and thus, power mode 0 is more efficient for this task.

The time needed to read the image is lower on the PC and JXAVIER as their DDR memories are faster compared to the DDR memories of the other platforms, e.g., JNANO achieves a memory bandwidth of 25.6GB/sec while JXAVIER 59.7GB/sec. PyTurboJPEG [62], which is an optimized Python library for encoding/decoding JPEG images in x86/x64 and ARM architectures, achieves lower loading times because it uses the CPU SIMD instructions discussed in Subsection V; instead of loading the pixels one by one, multiple pixels are loaded at a time, boosting performance. The highest performance gain is for the PC because it can load 256-bits of data by using a single instruction. On the contrary, the ARM CPUs that the edge devices support can load up to 128-bit. Pillow library provides slightly higher read times than OpenCV for the PC and IMX8P platforms and much higher for the other platforms. Pillow [63] library supports an optimized version leveraging the CPU vector instructions too (a.k.a Pillow-SIMD [64]), but it is not tested here. For the rest of this paper, we have used the PyTurboJPEG library.

As was expected, the bigger the input image, the higher the time to read/store from/to DDR memory. Reading the input image is one of the time-critical parameters, especially for large input images, even when the fast PyTurboJPEG library is used. Although the time needed to run the DL models scales well by providing more processing units (explained next), the read frame time cannot be reduced and therefore it remains a performance bottleneck. This is especially true when the powerful coprocessors are being used, where the time needed to read the image is higher than the time needed to run the DL models.

2) Evaluation of the time needed for pre/post-processing

The time needed to pre-process the image (Fig. 5) is lower than the time needed to read the image. This is because, in the pre-processing step, the image has already been loaded into the CPU's fast cache memory. The time of the pre-processing step is not highly affected by the image size.

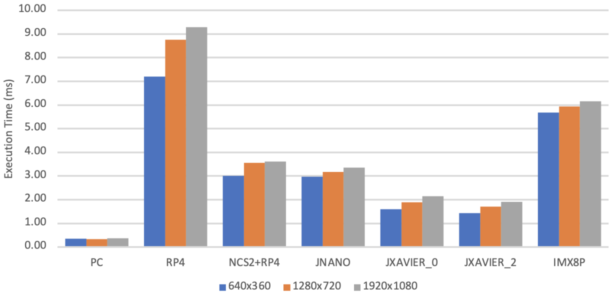

FIGURE 5. Evaluation of the Pre-Processing step

The pre/post-processing steps are insignificant for RP4, as the pre/post-processing time is much lower than the latency time. On the contrary, the pre/post-processing time of the other boards accounts for a significant part of the overall time. This is because the pre/post-processing steps are always executed on the CPU and not on the computationally powerful coprocessors. In NCS2, JNANO, and JXAVIER, the models (latency) scale well by providing more processing elements (GPU cores or vector processing units), while the pre-post-processing step does not scale well as it is executed on the CPU. Therefore, the un-optimized pre/post-processing steps account for a significant portion of time in NCS2, JNANO, and JXAVIER.

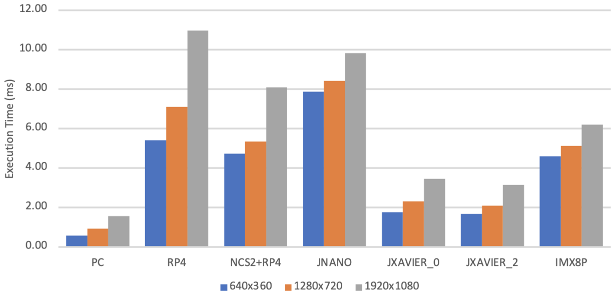

FIGURE 6. Evaluation of the Post-Processing step

3) Evaluation and optimization of the IC models (Method1)

In this subsection, Method1 is evaluated (Fig. 7 - Fig. 13). Note that the loading time is shown in secs, while the latency times are shown in milliseconds (ms).

a: Loading time

As far as the loading time is concerned (Fig. 7-Fig. 12), different models give different loading times as their memory footprint and parameters are different. The larger the memory size of a model, the more time is needed for loading. Note that when a model is loaded for a second time, it is normally loaded faster, because in this case it is already located in the CPU's cache; therefore, the loading time is sometimes higher for the R1 case and lower for the R2/R3 cases. It is important to note that INT8 achieves the least loading time as the memory size of the model is minimized greatly. The loading time of M2 was expected to be lower than that of M1, as the memory size of M2 is smaller, but was not the case.

The loading time ranges between 0.66-1.34secs for the PC (FP32 is used) and between 0.003-10 secs for the RP4. The time needed to load the FP32 model on RP4 is about one order of magnitude higher compared to the PC. However, when INT8 is used, the loading time is highly reduced, and it is even lower than the loading time of the PC. The loading times for the NCS2, JNANO, and JXAVIER are 0.07-1.09 secs, 2.12-65.79 secs, and 1.55-49.62 secs, respectively. NCS2 loading time is higher than that of RP4 because it is done via USB3.0 interface and uses FP16 data type; NCS2 does not support INT8. Last, RP4 with INT8 achieves lower loading times than Jetson platforms because TFLITE uses flat buffers [65].

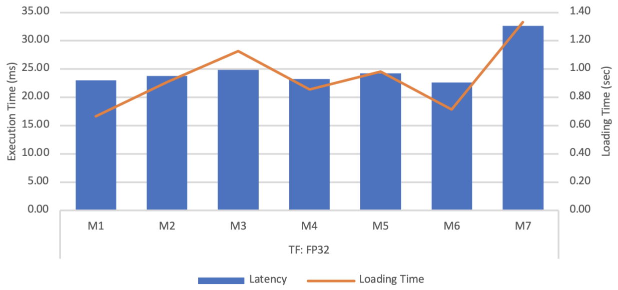

FIGURE 7. Method1: Latency evaluation/comparison of the IC models on the PC

b: Latency

As far as the latency time is concerned, M7 is by far the least efficient model because of its high complexity (Section IV). On the other hand, M6 is the fastest model in most cases as it achieves the lowest computational complexity (Section IV). M6 is from 3.5x to 33x times faster than M7 on the edge devices (Fig. 7-Fig. 12). The only case where M6 is not the fastest model is the TF-TRT FP16 case (both Jetson platforms), where M1 is slightly faster than M6 and the fastest model in this case. According to our analysis, this is because TF-TRT cannot generate efficient machine code for M6 in this case. Performance is very implementation dependent, meaning that different implementations of the same model might give significant variations in performance. Another example that supports this statement is shown in Fig. 10 and Fig. 11, where the TF-TRT FP16 for M6 gives latency values of 18.9 ms and 3.8 ms on JNANO and JXAVIER, respectively, while the NVIDIA optimizer (TRT) gives 5.1 ms and 1.4 ms, respectively; it is obvious that TF-TRT cannot leverage the target's hardware architecture as efficiently as TRT. Furthermore, in eIQ FP16 (Fig. 12), M5 is slightly faster than M6 for the same reason. To conclude, the lightest model is not always the fastest.

As expected, M6 is faster than M5, as M6 is a lightweight variant of M5, which does not include any of the advanced block sets of MobileNetV3. A special case exists in IMX8P, where eIQ cannot convert INT8 quantized M3 and M4 models, and also can't infer the TFLITE INT8 generated ones due to not supporting the advanced block sets of MobileNetV3. Similarly, M4 is faster than M3, as M4 is a lightweight variant of M3. There is just one case where M3 performs better than M4 (eIQ: FP16, Fig. 12); this also typifies the fact that performance is very implementation dependent. Last, M5 performs better than M4 in all cases apart from a) the Jetson platforms, and b) INT8 IMX8P because the powerful NPU can run only the MobileNetV3 minimalistic models.

What was surprising is that M1 is faster than M2 in most cases, which also typifies the fact that performance is very implementation dependent. M2 is faster than M1 when the ARM CPU is used, while M1 is faster than M2 when the coprocessors are used. According to [66], depthwise separable convolutions are not directly supported by NVIDIA GPUs and thus M1 is faster than M2 in this case. This is also reported by [67], where M2 runs faster on ARM, while M1 runs faster on Edge TPU. M2 uses more depthwise separable convolutions compared to M1 (17 compared to 13), to reduce the model's complexity. Although more memory efficient, depthwise 2D convolutions can indeed be slower than regular 2D convolutions due to their poor arithmetic intensity (ratio of compute to memory operations) [68].

Before we provide a detailed analysis for each hardware platform, note that the latency values of the fastest implementations on RP4, NCS2, JNANO, JXAVIER, and IMX8P are 19.2ms, 9.5 ms, 5.09 ms, 1.22 ms, and 4.52 ms, respectively. The un-optimized FP32 TensorFlow model takes 22.38 ms to run on the PC's GPU.

c: Evaluation of the edge devices

(A) **RP4:** On the RP4, TFLITE achieves significant performance improvement over TensorFlow (Fig.8) for all the MobileNet models, but not for the Inception model. Three different quantization levels are used. As expected, FP16 is faster than FP32, DINT8 is faster than FP16 and INT8 is faster than DINT8. According to [7], TFLITE generates more efficient code for the RP4 when INT16 is used, because RP4 CPU does not have hardware support for fast INT8 dot product instructions. In our future work, we are planning to evaluate our models using INT16 too. Furthermore, we have enabled multithreading with one (TFLITE1), two (TFLITE2), and four threads (TFLITE4). Although performance is improved, the scalability is low in all cases.

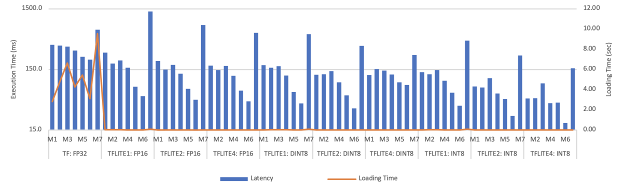

FIGURE 8. Method1: Latency evaluation/comparison of the IC models on the RP4

(B) **NCS2:** On NCS2, all the models run faster than RP4 (Fig. 9). Note that NCS2 cannot run un-optimized models (FP32) and supports just FP16. The smallest latency value being achieved on RP4 is 19.2 ms, while the smallest latency value on NCS2 is 9.5 ms. M6 runs 3.7x/2.5x/1.9x times faster on NCS2 compared to RP4 INT8, when one, two, and four threads are used, respectively (Fig. 9). NCS2 is more efficient because it supports 16 vector engines that can process 128-bits of data in a single instruction, each. OpenVINO can run asynchronously up to four inferences by using multiple threads, but a single inference cannot be parallelized. The former would increase the latency but improve throughput. Asynchronous mode has not been used here.

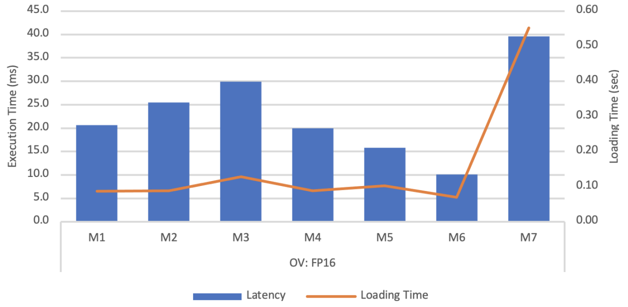

FIGURE 9. Method1: Latency evaluation/comparison of the IC models on the NCS2

(C) **JNANO:** Regarding JNANO, it achieves higher performance gains over NCS2 (Fig. 10). The fastest implementation on JNANO takes 5.09 ms, while on NCS2 takes 19.2 ms. TF-TRT FP16 and TRT FP16 boost performance, providing impressive speed-up values over TensorFlow. TF-TRT FP16 runs from 3.7x to 11.8x times faster than FP32, while NVIDIA's optimizer (TRT FP16) runs from 9.1x to 29.2x times faster than FP32. As was expected, TRT generates higher quality code compared to TF-TRT.

(D) **JXAVIER:** JXAVIER is by far the fastest platform (Fig. 11), e.g., M6 runs about 2.6x times faster (TRT INT8) compared to JNANO (TRT FP16). In JXAVIER, TF-TRT and TRT provide high-performance gains, especially TRT. Note that TRT supports INT8 too, should the hardware be capable. TF-TRT FP16 runs from 15.9x to 39.2x times faster than FP32, while NVIDIA's optimizer (TRT INT8) runs from 32.2x to 76.7x times faster than TF FP32. JXAVIER also supports a low-power accelerator (DLA) through TRT libraries; DLA is less performant but more power efficient, which runs from 13.9x to 19.9x for FP16 and from 16.1x to 23.8x for INT8 compared to TF FP32.

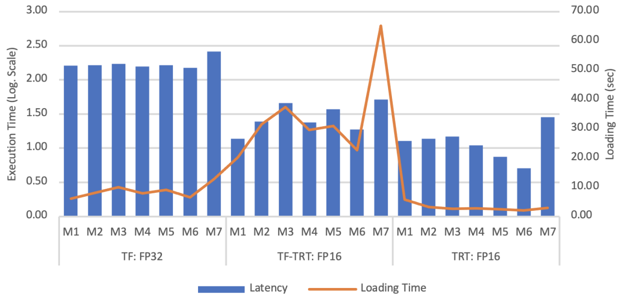

FIGURE 10. Method1: Latency evaluation/comparison of the IC models on the JNANO

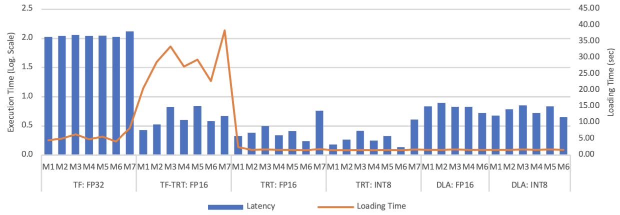

FIGURE 11. Method1: Latency evaluation/comparison of the IC models on the JXAVIER_2

(E) **IMX8P:** For this platform two different optimization tools have been used, TFLITE and eIQ (Section IV). Furthermore, three different quantization levels are used (FP16, DINT8, INT8). Note that IMX8P cannot run un-optimized models. Furthermore, the NPU coprocessor supports only INT8-type models. Therefore, the FP16 and DINT8 quantized models are not supported by the NPU and therefore they run on the CPU. DINT8 achieves performance gains of average 1.8x over FP16 (Fig. 12). INT8 achieves high-performance gains over FP16 when the NPU coprocessor is used; about 36.0x/34.7x when TFLITE/eIQ are used, respectively. TFLITE and eIQ fail to use the NPU for M3, M5, and M7 (eIQ gives errors when running M3 and M5 and therefore they are not shown here) and this is why their performance is poor (they run on ARM). When NPU is used, IMX8P is the second fastest platform. eIQ gives slightly worse performance compared to TFLITE.

(F) **Optimization Frameworks:** The average speed-up values achieved by using the optimization frameworks are shown in Fig. 13. TFLITE achieves high speed-up values for IMX8P when NPU is used (INT8 only), and a significant performance gain on RP4 when INT8 and multithreading are used. Note that NCS2 and IMX8P cannot run un-optimized models and therefore the speed-up shown for NCS2 is over RP4 (host platform), while the speed-up shown for IMX8P is

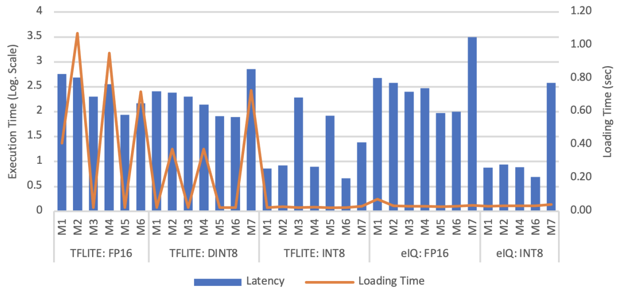

FIGURE 12. Method1: Latency evaluation/comparison of the IC models on the IMX8P

over the FP16 model. TFLITE gives a low speed-up value on IMX8P for DINT8 as the NPU coprocessor supports INT8 models only. To conclude, the optimization tools achieve high-performance gains on all platforms.

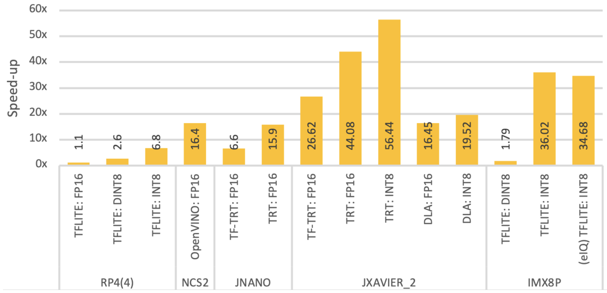

FIGURE 13. Method1: Evaluation/comparison of the optimization frameworks (average speed-up is shown)

(G) **Value:** In Fig. 14, an evaluation in terms of value is applied. JNANO achieves the best solution here, while it provides the third-best model latency results and it is much lower cost than the faster JXAVIER and IMX8P. JXAVIER and RP4 (M6 only) provide the 2nd best solution. IMX8P does not provide a good option here as it is very expensive. As was expected, the PC is by far the worst platform as it is very expensive.

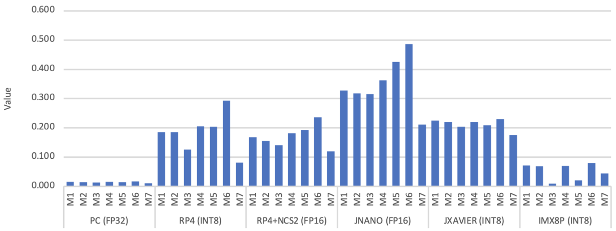

FIGURE 14. Method1: Value evaluation/comparison on different edge devices (the pre/post processing steps are included)

(H) **Efficiency:** In Fig. 15, an evaluation in terms of efficiency is applied. In this case, JXAVIER provides the best solution for all the models but M6. JXAVIER is by far the fastest board and its maximum power is 15 Watts with Jetpack 4.5.1. JNANO comes first for the M6 model and second overall. NCS2 is the third best option.

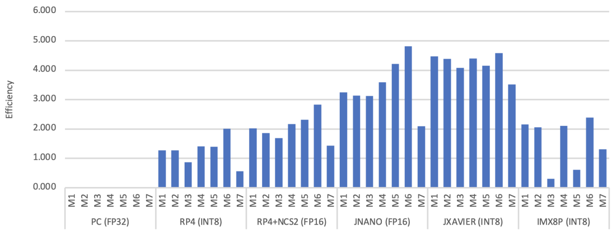

FIGURE 15. Method1: Efficiency evaluation/comparison on different edge devices (the pre/post processing steps are included)

4) Evaluation and optimization of the OD models (Method2)

In this subsection, Method2 is evaluated (Fig. 16-Fig. 24). Note that the loading time is shown in secs, while the latency times are shown in milliseconds (ms).

a: Loading Time

The quantized models achieve a lower loading time compared to the original models (Fig. 16-Fig. 21), as they use less memory. The loading time of the O3 model is higher compared to the other models, as its memory requirements are higher. We were surprised when we found out that the loading time of the quantized models on Jetson platforms can be even higher than the original models. This is a known and reported issue of the Protobuf library [69], [70], which is improved by a big margin by recompiling the Protobuf library with C++ enabled instead of Python implementation.

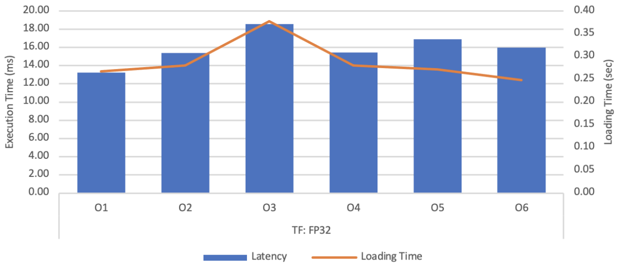

FIGURE 16. Method2: Latency evaluation/comparison of the OD models on the PC

b: Latency

O3 model is the least efficient method for all hardware platforms (Fig. 16-Fig. 21), as it gives the highest number of arithmetical instructions as well as the largest memory footprint. As in Method1, SSD-MobileNetV1 is faster than SSD-MobileNetV2 on all platforms apart from the cases where the ARM CPU is used (RP4, IMX8P FP16 models), therefore O1 is faster than O2 and O2 is faster than O4 only when the coprocessors are used. This is due to depthwise separable convolutions which are not well implemented [66], and as it has been discussed before the models are very implementation dependent. O6 is the least complex model and therefore it achieves the lowest latency values for all platforms apart from the PC. O5 is the second fastest model on RP4 and JNANO, while O1 outperforms O5 on the PC, NCS2, and JXAVIER. O6 and O5 are not supported by all platforms; IMX8P does not support SSD-MobileNetV3 (it supports only the minimalistic MobileNetV3 models for IC). Last, TRT fails to optimize O5 and O6 models because the following layers are not supported: addv2 and fusedbatchnormv3. Replacing these with supported layers resulted in connection issues between adjoining layers, which are to be resolved as future work.

The latency values of the fastest implementations on RP4, NCS2, JNANO, JXAVIER, and IMX8P are 47 ms, 22.4 ms, 17.2 ms, 2.9 ms, and 13.9 ms, respectively. The un-optimized FP32 TensorFlow model takes 13.2 ms to run on the PC's GPU.

c: Evaluation of the edge devices

(A) **RP4:** Three different quantized models are used. Multithreading is not supported by TensorFlow 1.X where we trained the models. TFLITE did not achieve as high performance gains as observed in Method1 with TensorFlow 2.X, with average gains of FP16: 2.0x, DINT8: 1.4x, INT8: 1.8x. TF1.X fails to generate efficient machine-level code here for TFLITE quantized models. FP16 is the fastest quantized level for all the models apart from O1, where INT8 is more efficient in memory footprint than FP16. This also typifies that performance is implementation dependent. DINT8 does not perform that well compared to INT8 and FP16. According to [7], TFLITE generates more efficient code for the RP4 when INT16 is used, because RP4 CPU does not have hardware support for fast INT8 dot product instructions.

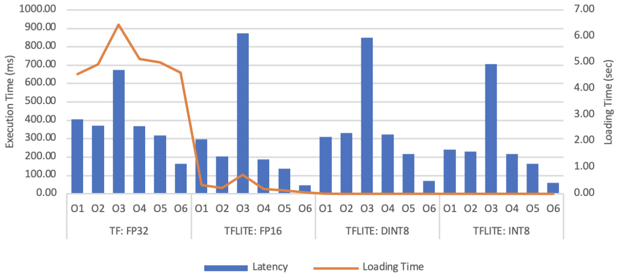

FIGURE 17. Method2: Latency evaluation/comparison of the OD models on the RP4

(B) NCS2: NCS2 achieves better performance than RP4, in all cases (Fig. 18). O6 and O1 are the fastest models. O6 runs 2.1x times faster on NCS2 compared to the fastest solution on RP4 (FP16-O6). On average, NCS2 runs models 9x times faster than RP4. The time needed to read the input image accounts for a significant amount of time here which is performed on RP4, for the reason explained before (Method1).

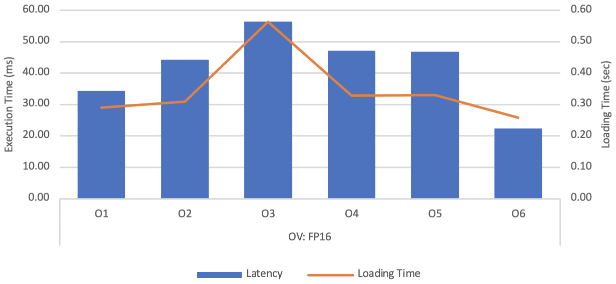

FIGURE 18. Method2: Latency evaluation/comparison of the OD models on the NCS2

(C) JNANO/JXAVIER: On JNANO and JXAVIER, TF-TRT and TRT provide significant speed-up values in all cases and especially TRT (Fig. 19, 20). Regarding TF-TRT, on average JNANO had 3.1x gains and the fastest model being O6 while on JXAVIER an average of 3.8x gains, with O1 being slightly faster. TRT resulted in an average of 4.0x (FP16) for JNANO and 9.3x (INT8) for JXAVIER. TRT does not support O5 and O6 and therefore O1 is the fastest solution on both platforms; TRT cannot convert these models as there are unsupported layers (we even tried using Jetpack 4.5 and 4.6). As it was explained in Method1, the read time, as well as the pre/post-processing time, does not scale well here and as a consequence, the latency time on Jetsons is not the time-critical parameter, especially for JXAVIER.

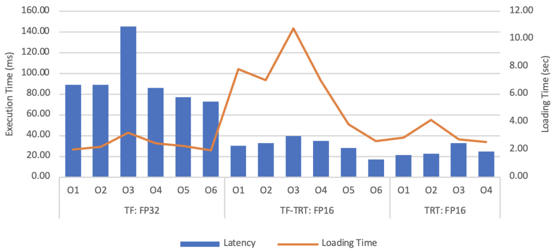

FIGURE 19. Method2: Latency evaluation/comparison of the OD models on the JNANO

(D) IMX8P: TFLITE and eIQ optimizations are used here (Fig. 21). TFLITE provides significant speed-up values when the NPU coprocessor is used (supports only INT8). TFLITE cannot run the FP16 O3 model because the model is too memory-demanding to run on this platform. Furthermore, eIQ does not support conversion of O5 and O6 models and therefore O1 is the fastest model in this case. Unlike the IC case, eIQ provides faster inference here compared to TFLITE.

(E) Optimization Frameworks: The average speed-up values achieved by using the optimization frameworks are shown

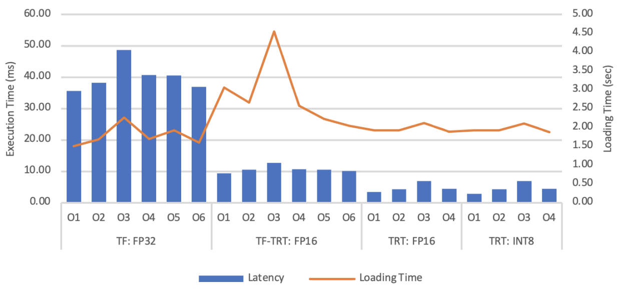

FIGURE 20. Method2: Latency evaluation/comparison of the OD models on the JXAVIER_2

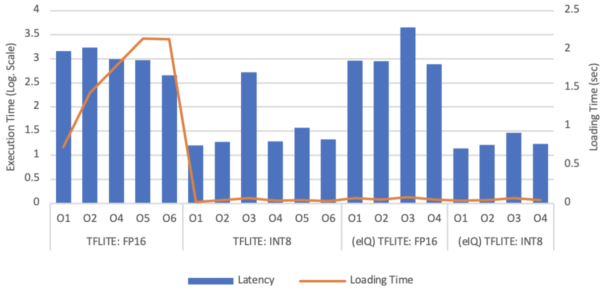

FIGURE 21. Method2: Latency evaluation/comparison of the OD models on the IMX8P

in Fig. 24. The performance gain is lower compared to the IC case, but mainly due to TF1.X not generating efficient machine code for some of the platforms like RP4. Note that NCS2 and IMX8P cannot run un-optimized models and therefore the speed-up shown for NCS2 is over RP4 (host platform), while the speed-up shown for IMX8P is over the FP16 model.

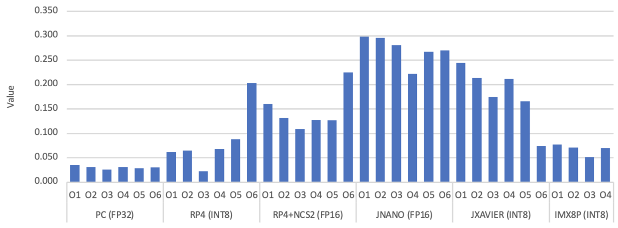

FIGURE 22. Method2: Value evaluation/comparison on different edge devices (the pre/post processing steps are included)

(F) Value: As far as the evaluation in terms of value is concerned (Fig. 22), JNANO and JXAVIER provide the best solutions, depending on the model being used. NCS2 and RP4 are very good solutions for O6 only.

(G) Efficiency: Regarding the evaluation in terms of efficiency (Fig. 23), JXAVIER provides the most efficient solution. JNANO comes second, while NCS2 provides a very good solution for O6. Note that TRT is not supporting yet SSD-MobileNetV3 and therefore we would expect JXAVIER to score even higher in this case. The same holds for JNANO and IMX8P (where O5 and O6 are not supported).

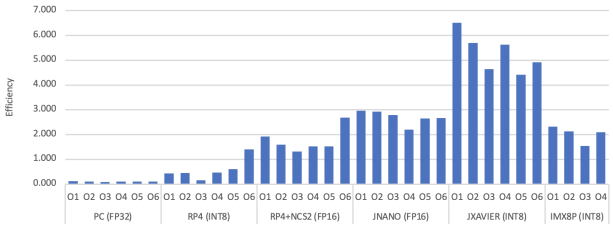

FIGURE 23. Method2: Efficiency evaluation/comparison on different edge devices (the pre/post processing steps are included)

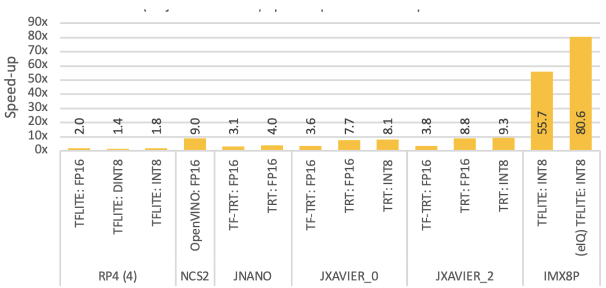

FIGURE 24. Method2: Evaluation/comparison of the optimization frameworks

IX. DISCUSSION

A. OPTIMIZATION FRAMEWORKS

To efficiently run DL IC and OD models on the edge, different optimization frameworks and options need to be investigated and the balance of accuracy vs inference speed must be investigated for the target use case. To this end, we provide our insightful observations:

- The optimization frameworks used in this work provide improved latency values for all the models in most cases (apart from SSD-Inception; see next bullet); they provide up to 6.8x/16.4x/15.9x/56.44x/36x times lower latency on IC models and up to 2x/9x/4x/9.3x/80.6x times lower latency on OD models, for RP4 / NCS2 / JNANO / JXAVIER / IMX8P, respectively. The IC models are better optimized for all the hardware platforms apart from the IMX8P where the OD models achieve higher speed-up compared to the IC ones.
- TFLITE fails to speed up the SSD-Inception model (this is a computationally expensive model) in Method2 and consequently TFLITE gives even slower latency than the un-optimized case.
- MobileNetV2 runs faster than MobileNetV1 only on an ARM processor (RP4 and IMX8P when the coprocessor is not used), while the opposite is true when any type of coprocessor is used (NCS2, JNANO, JXAVIER, IMX8P). One of the main reasons is that depthwise separable convolutions are not well implemented by the

coprocessors [66]–[68], therefore the more they are used, the bigger the bottleneck.

- The models' latency strongly depends on whether the optimization tools generate efficient machine code for the target platform (e.g., use the appropriate SIMD instructions) and whether the optimization tools can take advantage of the available powerful coprocessors' capabilities. We also show that different implementations of the same model provide high variations in latency. For example, the latency of O4 on JXAVIER can be from 4.26 up to 40.72 ms; the latency of O4 is 40.72/10.76/4.42/4.46 ms when TF FP32, TF-TRT FP16, TRT FP16, and TRT INT8, are used, respectively.
- Performance does not always align with the quantization level. Although quantized models with shorter bit-width values should run faster compared to the wider bit-width ones, we found out that this is not always true. This is because the optimization frameworks fail to generate efficient machine code in this case. For example, on RP4, the OD models derived from TF1.X run faster in the FP16 case compared to the INT8 case, while in TF2.X the INT8 models were always the fastest. Furthermore, on JXAVIER, the OD TRT INT8 models run as fast as the FP16 ones, which are also derived from TF1.X. Note that different platforms support different quantization levels.
- The most lightweight model is not always the fastest, but the most complex model is always the slowest. This is because the optimization frameworks fail to generate efficient machine code in many cases. For example, MobileNetV1 is faster than MobileNetV2 when deployed on a non-ARM technology platform, for both IC and OD.
- Depthwise separable convolutions are not well implemented on non-ARM technology hardware.
- TRT (NVIDIA's optimizer) is superior to TF-TRT in all cases (Jetson platforms) due to being able to optimize the model as a whole graph, while TF-TRT optimizes layer by layer and leaves unsupported layers in their original quantization format (FP32).
- Multithreading implementations on RP4 do not scale well. Although the latency is reduced by providing more threads, the speed-up values do not align with the number of CPU cores being used.
- The time needed to read the input image and/or the time needed to pre/post process the image can be comparable (or even higher) to the time needed to run the DL model, especially for large input images. Reading the input frame normally takes more time than running the model, even using the PyTurboJPEG library which improves the reading time. Therefore, using such libraries is of critical importance. The pre/post-processing time is less but still can be comparable to the time needed to run the model. For example, the time needed to read the input frame on JNANO ranges from 3.78-29.6 ms, the time needed to pre-process the image ranges from 2.97-3.35, and the time need to post-process the image 7.9-9.84 ms and the time needed to run the IC/OD TRT models 0.71-1.46 / 21.34-32.7 ms, respectively.
- We believe that there is room for improvement to the optimization frameworks as they fail to generate efficient machine code in many cases. We expect that first, the new versions of the optimization frameworks will further optimize the models, and second, manually optimized code for the target hardware platform would run much faster.
- To efficiently run deep learning IC and/or OD models on the edge devices is a non-trivial and time-consuming task. To ease the model selection phase we deliver Subsections B and C, and to ease the board selection process we provide Subsections D, E, F, G and H.

B. IMAGE CLASSIFICATION (IC) MODELS

Seven SOTA IC models have been used, six lightweight MobileNet models (MobileNetV1, MobileNetV2 and four different versions of MobileNetV3) and a complex one (InceptionV3). MobileNetV3 (M3-M6) is superior as it provides the best solution in terms of both accuracy (Fig. 2) and latency, while InceptionV3 (M7) is the least attractive solution as it is neither the most accurate for our use case nor the fastest. The most accurate model in terms of accuracy is M3, while M3-M5 (MobileNetV3) and M7 (InceptionV3) provide roughly the same accuracy. Note that M3-M5 run several times faster than M7. M6 (MobileNetV3 Small Minimalistic) is the fastest model but it is less accurate than M3-M5. MobileNetV1 is faster than MobileNetV2 in most cases, but MobileNetV2 is more accurate. To sum up, MobileNetV3 is by far the most efficient solution.

C. OBJECT DETECTION (OD) MODELS

Six SOTA IC models have been used, five lightweight MobileNet models (SSD-MobileNetV1, SSD-MobileNetV2, SSDLITE-MobileNetV2, and two different versions of SSD-MobileNetV3) and a complex one (SSD-InceptionV2). As in the IC case, MobileNetV3 is superior in terms of both latency and accuracy (Fig. 3); however, it is not supported by all hardware platforms. SSD-InceptionV2 is by far the slowest model and not the most accurate, and therefore it does not present an efficient solution here. As in the IC case, SSD-MobileNetV1 runs faster than SSD-MobileNetV2 in most cases.

To conclude, if MobileNetV3 is supported to the target hardware platform, it presents the most efficient solution. Otherwise, MobileNetV1 is the most preferable model. MobileNetV3 is expected to further boost performance on the Jetson platforms, if TRT will be supporting this model's architecture. Furthermore, SSD-MobileNetV3 is also expected to boost performance on IMX8P, if its engine and conversion tools are to support its architecture.

D. RP4 PLATFORM

RP4 is the lowest-cost board from all five that were used and it provides a very good value for money; in IC, RP4 presents the 2nd best solution in terms of value just for M6, while in OD, it presents the third best solution just for O6. If higher accuracy is needed M4/M5 are competitive solutions too; for the OD case, the most accurate models run much slower. Although, INT8 is the best quantization level for IC, FP16 performs better for the OD models; it seems that TF1.X cannot efficiently utilise the ARM INT8 SIMD instructions, while this seemed to be improved in TF2.X (based on IC model results). RP4 supports multithreading with TF2.X and although the models do not scale well, lower latency values are always achieved. TFLITE cannot provide high latency gains on OD models compared to the IC ones, and therefore it presents a more competitive solution for IC models. The time needed to read and pre/post process the image is a small percentage of the overall inference time.

E. NCS2 PLATFORM

NCS2 is a low-cost, high-performance accelerator whose maximum power consumption is just 2 Watts but required to be connected to a host, which was RP4 in our setup. For the most lightweight models (M6 and O6), it presents the 3rd best solution in terms of value and efficiency, for both the IC and OD case. NCS2 supports FP16 only, where OpenVINO provides up to 16.4x/9.0x times faster code than the un-optimized FP32 method for the IC/OD, respectively. M6 and O6 models (MobileNetV3) are the superior models in terms of latency. If higher accuracy is needed, M5 and O1 present the best options. The read frame time accounts for a significant amount of the overall execution time even by using the PyTurboJPEG library and thus the usage of such libraries is beneficial. Note that OpenVINO can run asynchronously multiple inferences by using multiple "requests", but a single inference cannot be parallelized. The former would increase the average latency but also increase throughput.

F. JNANO PLATFORM

JNANO is a lost-cost, low-power and powerful hardware platform. This makes JNANO an excellent choice for all the target metrics. JNANO achieves the best solution in terms of value and the second best in terms of efficiency. Just for M6 it presents the best solution in terms of efficiency too. Furthermore, it is the 3rd fastest board after JXAVIER and IMX8P. TRT is by far the optimization tool that needs to be used to leverage the NVIDIA's hardware architecture. JNANO does not support INT8 and thus FP16 quantization is the only choice. MobileNetV3 is the fastest model here. M6 is the fastest model for the IC case, while O6 if the fastest model for the OD case. Note that MobileNetV3 (O5 and O6) is not supported by TRT for the OD method and therefore JNANO will present an even more attractive solution for the OD case if MobileNetV3 can natively be supported by TRT. JNANO achieves low latency values for most of the IC/OD models and thus it provides a competitive solution even when higher accuracy is needed. The time needed to read the input frame accounts for a high part of the overall execution time and thus a library such as PyTurboJPEG is advantageous here, especially for high resolutions input images. The Jetson platforms provide extra GPU utilities for efficiently reading and pre-processing the input frame, but we did not see any improvements over the OpenCV libraries. JNANO achieves lower read frame times compared to RP4 and IMX8P.

G. JXAVIER PLATFORM

JXAVIER is by far the most powerful board and its power consumption is not high, considering the performance gains it can achieve. JXAVIER presents an excellent solution for all the target metrics. Even if it is an high-cost board, its performance gains compensate for its high cost. JXAVIER achieves the best solution in terms of efficiency and latency and the 2nd best solution in terms of value, on both IC and OD. TRT with INT8 is the best option to leverage the NVIDIA's hardware architecture. MobileNetV3 (M6) is the fastest model for IC, but MobileNetV3 is not supported by TRT for the OD case. Thus, MobileNetV1 is the best option for the OD method (O1). It is important to note that apart from the very powerful GPU and CPU, JXAVIER also has two DLA coprocessors to further reduce energy consumption and run multiple models concurrently; although we have evaluated its performance (it runs slower than the GPU) we have not evaluated its power consumption (future work). JXAVIER achieves low latency values for most of the IC/OD models and thus it provides a competitive solution even when higher accuracy is needed.

As in JNANO, a dedicated library for efficiently reading the input frame (such as PyTurboJPEG) is required, as the time needed to run the models is normally lower than reading the input frame and pre/post process the image. It is important to note that JXAVIER supports a very fast LPDDR memory and therefore it reads the input frame faster compared to the other boards. Last, note that JXAVIER supports five different power modes, to provide different trade-offs between latency time and power consumption, e.g., the power mode 0 uses just 2 out of 6 CPU cores which running at 1.9GHz, while power mode 2 uses all the 6 cores but their frequency is lower (1.4GHz).

H. IMX8P PLATFORM

IMX8P presents an excellent solution in terms of latency time (it is the second fastest board). However, it is the most expensive board and therefore the worst solution in terms of value, on both IC and OD. Regarding efficiency, it provides an efficient solution for most of the models. Fast inference is achieved only in the INT8 case, where the NPU coprocessor is used. Note that the non-INT8 models run on the ARM processor and in this case the latency is much higher. Although eIQ provides higher performance than TFLITE for the OD case, TFLITE gives slightly better latency times for the IC case. Furthermore, NPU supports only the minimalistic models of MobileNetV3 and thus it cannot run M3 and M5 models; additionally, eIQ fails to convert SSD-MobileNetV3 models in the OD case. Therefore, M6 (MobileNetV3) is the fastest model for the IC case, while O1 (SSD-MobileNetV1) is the fastest model for OD. Last, an optimized library such as PyTurboJPEG to efficiently read the input frames is required here.

X. CONCLUSIONS AND FUTURE WORK

Developing efficient computer vision AI applications is a non-trivial and challenging task as different hardware platforms, models, libraries, optimisation techniques and tools need to be investigated. In this paper, seven image classification and six object detection on-the-edge SOTA models were optimized, evaluated, and compared in terms of accuracy, value and efficiency, on five commercial off-the-shelf edge devices. To this end, an IC and OD face mask wearing detection architecture is developed as a case study to explore and analyse. The models have been optimized by using the SOTA optimization frameworks and different quantization levels for deployment on five edge devices of various types of technology. The five edge devices are also evaluated and compared in terms of inference time, value and efficiency. Each one offered a different flavour of capabilities vs cost/power though the results derived from the mentioned metrics.

In Section IX, we provide insightful observations with regards to the optimization frameworks, models and edge devices. We show that comparing just the execution time of the DL models might be misleading, as the time needed to read the input frame and/or pre/post-process the input/output image is comparable to or even higher than the time needed to run the deep learning models. Therefore, optimizing the process of reading and pre/post processing the image, through the use of appropriate libraries and hardware resources, can be of critical importance too. Furthermore, by not solely depending on latency results and deriving metrics such as efficiency and value gives a perspective that assists in choosing the right solution based on the available power and cost budget for an edge application.

Another important insight is that even by using the SOTA optimization tools the inference time of the complex IC and OD models cannot be reduced in most cases. On the contrary, the inference time of the lightweight MobileNetV1-V3 models can be highly reduced by using the appropriate optimization options and being most efficient on ARM CPU platforms. Another insightful observation is that inference time does not always align with the quantization level as the optimization tools fail to generate efficient machine code in some cases. For the same reason, we show that the most lightweight model is not always the fastest.

Last, we demonstrate that each target hardware has its own set of capabilities/features and offers various levels of performance; the right data type must be considered based on what is supported by the architecture. As mentioned previously, the performance of a given model should be explored in more depth than just the execution time of the model (latency), but for the whole video pipeline, including metrics that include throughput per cost and/or power consumption. From the hardware boards evaluated and the results we obtained, JXAVIER is the best edge device in terms of latency and efficiency, while JNANO is the best in terms of value.

As far as our future work is concerned, we are planning to measure the energy consumption of the edge devices by using power meters, instead of using the maximum power values. In the longer term, we are planning to expand this work to the PyTorch framework and derive a bigger SOTA model pool for both IC and OD models, with further optimization techniques applied such as pruning/weight clustering, optimized pre/post-processing and lastly adding further hardware platforms such as FPGA solutions.

APPENDIX

TABLE 5. Detailed Method1 (IC) Training Results

	Model	Accuracy	Precision	Recall	F1-Score	Training Time (hh:mm:ss)
Dataset1	M1	99.71%	**99.86%**	99.57%	99.71%	00:02:13
	M2	**99.78%**	99.57%	**100.00%**	**99.78%**	00:02:15
	M3	99.64%	99.71%	99.57%	99.64%	00:02:18
	M4	99.71%	99.71%	99.71%	99.71%	00:02:11
	M5	99.20%	98.99%	99.42%	99.20%	00:02:10
	M6	99.42%	99.71%	99.14%	99.42%	00:02:07
	M7	99.64%	99.42%	99.85%	99.64%	00:02:24
Dataset2	M1	90.31%	80.98%	**99.42%**	89.26%	00:06:01
	M2	92.24%	84.97%	99.33%	91.59%	00:06:12
	M3	**96.38%**	**94.39%**	98.26%	**96.29%**	00:06:11
	M4	95.39%	91.41%	99.26%	95.17%	00:06:03
	M5	95.78%	92.30%	99.16%	95.60%	00:05:59
	M6	85.86%	72.71%	98.44%	83.64%	00:05:50
	M7	95.49%	91.62%	99.26%	95.29%	00:06:30

TABLE 6. Detailed Method2 (OD) Training Results - mAP

	Model	mAP IoU	mAP IoU	mAP IoU	mAP	mAP	mAP	Training Time
		.50:.05:.95	.50	.75	small	medium	large	(hh:mm:ss)
Dataset3	O1	30.30%	57.90%	26.80%	19.30%	45.70%	82.90%	02:13:00
	O2	**33.80%**	64.10%	31.40%	**22.90%**	49.50%	81.40%	02:32:00
	O3	32.70%	**66.70%**	28.60%	22.70%	**54.80%**	68.40%	02:53:00
	O4	32.60%	61.40%	**31.70%**	22.00%	50.60%	67.90%	02:38:00
	O5	28.90%	52.90%	27.70%	16.90%	44.20%	84.00%	02:25:00
	O6	30.60%	57.90%	26.60%	19.20%	46.20%	**84.70%**	00:38:00
Dataset4	O1	44.40%	67.20%	49.20%	14.80%	36.10%	72.80%	02:41:00
	O2	47.40%	74.80%	49.30%	20.30%	39.00%	73.90%	02:55:00
	O3	**51.70%**	**82.70%**	**55.40%**	24.00%	46.20%	**75.80%**	03:10:00
	O4	46.20%	69.80%	31.70%	**25.90%**	**50.60%**	71.80%	03:00:00
	O5	48.50%	76.00%	53.40%	18.20%	43.80%	73.30%	02:46:00
	O6	41.90%	69.30%	43.10%	13.20%	34.40%	71.70%	01:23:00

TABLE 7. Detailed Method2 (OD) Training Results - mAR

	Model	mAR	mAR	mAR	mAR	mAR	mAR	Training Time
		max=1	max=10	max=100	small	medium	large	(hh:mm:ss)
Dataset3	O1	17.60%	34.70%	36.20%	26.00%	49.40%	86.00%	02:13:00
	O2	**18.70%**	39.10%	40.80%	30.90%	54.90%	83.90%	02:32:00
	O3	16.70%	**40.40%**	**42.90%**	**32.30%**	**61.70%**	76.10%	02:53:00
	O4	17.70%	37.70%	39.10%	28.80%	57.10%	71.30%	02:38:00
	O5	16.40%	35.70%	39.50%	27.30%	58.50%	86.70%	02:25:00
	O6	16.70%	36.10%	38.80%	27.20%	55.50%	**88.20%**	00:38:00
Dataset4	O1	29.70%	48.10%	48.90%	20.30%	40.20%	76.50%	02:41:00
	O2	30.30%	51.40%	52.70%	26.60%	45.40%	76.80%	02:55:00
	O3	**31.10%**	**55.50%**	**58.60%**	32.90%	55.90%	**80.50%**	03:10:00
	O4	17.80%	41.10%	43.00%	**33.90%**	**58.20%**	74.40%	03:00:00
	O5	29.80%	52.60%	55.50%	26.20%	55.70%	77.10%	02:46:00
	O6	27.50%	46.40%	48.70%	19.90%	43.80%	75.50%	01:23:00

REFERENCES

[1] A. Krizhevsky, I.Sutskever, and G. Hinton, "Imagenet classification with deep convolutional neural networks," *Neural Information Processing Systems*, vol.25, 01 2012.

[2] X. Zhang, Y. Wang, S. Lu, L. Liu, L. Xu, and W. Shi, "Openei: an open framework for edge intelligence," 2019.

[3] A. Allan, "The big benchmarking roundup," [Online]. Available: https://www.hackster.io/news/the-big-benchmarking-roundup-a561fbfe8719

[4] S. Voghoei, N. H. Tonekaboni, J. G. Wallace, and H. R. Arabnia, "Deep learning at the edge," 2019.

[5] H.-H. Phan and N.-S. Vu, "Information theory based pruning for cnn compression and its application to image classification and action recognition," in *2019 16th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, 2019, pp. 1–8.

[6] A. Jain, S. Bhattacharya, M. Masuda, V. Sharma, and Y. Wang, "Efficient execution of quantized deep learning models: a compiler approach," 2020.

[7] M. Pietron and M. Wielgosz, "Retrain or not retrain? - efficient pruning methods of deep cnn networks," in *Computational Science – ICCS 2020*, V. V. Krzhizhanovskaya, G. Závodszky, M. H. Lees, J. J. Dongarra, P. M. A. Sloot, S. Brissos, and J. Teixeira, Eds. Cham: Springer International Publishing, 2020, pp. 452–463.

[8] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference," 2021.

[9] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," *International Conference on Machine Learning*, 2019. [Online]. Available: https://arxiv.org/abs/1905.11946

[10] S. S. A. Zaidi, M. S. Ansari, A. Aslam, N. Kanwal, M. Asghar, and B. Lee, "A survey of modern deep learning based object detection models," *Digital Signal Processing*, vol. 126, p. 103514, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1051200422001312

[11] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: single shot multibox detector," *Lecture Notes in Computer Science*, p. 21–37, 2016. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-46448-0_2

[12] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," 2015. [Online]. Available: https://arxiv.org/abs/1506.02640

[13] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie, "Feature pyramid networks for object detection," 2016. [Online]. Available: https://arxiv.org/abs/1612.03144

[14] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask r-cnn," in *2017 IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 2980–2988.

[15] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: towards real-time object detection with region proposal networks," 2016.

[16] M. Antonini, T. H. Vu, C. Min, A. Montanari, A. Mathur, and F. Kawsar, "Resource characterisation of personal-scale sensing models on edge accelerators," in *Proceedings of the First International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things*, ser. AIChallengeIoT'19. New York, NY, USA: Association for Computing Machinery, 2019, p. 49–55. [Online]. Available: https://doi.org/10.1145/3363347.3363363

[17] Y. Xing, S. Liang, L. Sui, X. Jia, J. Qiu, X. Liu, Y. Wang, Y. Wang, and Y. Shan, "Dnnvm : end-to-end compiler leveraging heterogeneous optimizations on fpga-based cnn accelerators," 2019.

[18] J. Wang and S. Gu, "Fpga implementation of object detection accelerator based on vitis-ai," in *2021 11th International Conference on Information Science and Technology (ICIST)*, 2021, pp. 571–577.

[19] T. Belabed, M. G. F. Coutinho, M. A. C. Fernandes, C. V. Sakuyama, and C. Souani, "user driven fpga-based design automated framework of deep neural networks for low-power low-cost edge computing," *IEEE Access*, vol. 9, pp. 89 162–89 180, 2021.

[20] R. Hadidi, J. Cao, Y. Xie, B. Asgari, T. Krishna, and H. Kim, "Characterizing the deployment of deep neural networks on commercial edge devices," in *2019 IEEE International Symposium on Workload Characterization (IISWC)*, 2019, pp. 35–48.

[21] M. Qasaimeh, K. Denolf, A. Khodamoradi, M. Blott, J. Lo, L. Halder, K. Vissers, J. Zambreno, and P. H. Jones, "Benchmarking vision kernels and neural network inference accelerators on embedded platforms," *Journal of Systems Architecture*, vol. 113, p. 101896, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762120301697

[22] H. Kong, S. Huai, D. Liu, L. Zhang, H. Chen, S. Zhu, S. Li, W. Liu, M. Rastogi, R. Subramaniam, M. Athreya, and M. Anthony Lewis, "Edlab: a benchmark for edge deep learning accelerators," *IEEE Design Test*, pp. 1–1, 2021.

[23] A. A. Asyraaf Jainuddin, Y. C. Hou, M. Z. Baharuddin, and S. Yussof, "Performance analysis of deep neural networks for object classification with edge tpu," in *2020 8th International Conference on Information Technology and Multimedia (ICIMU)*, 2020, pp. 323–328.

[24] P. KANG and J. JO, "Benchmarking modern edge devices for ai applications," *IEICE Transactions on Information and Systems*, vol. E104.D, pp. 394–403, 03 2021.

[25] L. Jiao, F. Zhang, F. Liu, S. Yang, L. Li, Z. Feng, and R. Qu, "A survey of deep learning-based object detection," *IEEE Access*, vol. 7, p. 128837–128868, 2019. [Online]. Available: http://dx.doi.org/10.1109/ACCESS.2019.2939201

[26] P. Druzhkov and V. Kustikova, "A survey of deep learning methods and software tools for image classification and object detection," *Pattern Recognition and Image Analysis*, vol. 26, pp. 9–15, 01 2016.

[27] Z. Jin and H. Finkel, "Analyzing deep learning model inferences for image classification using openvino," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2020, pp. 908–911.

[28] S. Tuli, N. Basumatary, and R. Buyya, "Edgelens: deep learning based object detection in integrated iot, fog and cloud computing environments," 2019.

[29] P. Puchtler and R. Peinl, "Evaluation of deep learning accelerators for object detection at the edge," in *KI 2020: Advances in Artificial Intelligence*, U. Schmid, F. Klügl, and D. Wolter, Eds. Cham: Springer International Publishing, 2020, pp. 320–326.

[30] S. Hossain and D.-j. Lee, "Deep learning-based real-time multiple-object detection and tracking from aerial imagery via a flying robot with gpu-based embedded devices," *Sensors*, vol. 19, no. 15, 2019. [Online]. Available: https://www.mdpi.com/1424-8220/19/15/3371

[31] V. Mittal and B. Bhushan, "Accelerated computer vision inference with ai on the edge," in *2020 IEEE 9th International Conference on Communication Systems and Network Technologies (CSNT)*, 2020, pp. 55–60.

[32] G. Verma, Y. Gupta, A. M. Malik, and B. Chapman, "Performance evaluation of deep learning compilers for edge inference," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2021, pp. 858–865.

[33] R. David, J. Duke, A. Jain, V. J. Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, S. Regev, R. Rhodes, T. Wang, and P. Warden, 'Tensorflow lite micro: embedded machine learning on tinyml systems," 2021.

[34] M. Schneider, R. Amann, and C. Mitsantisuk, "Waste object classification with ai on the edge accelerators," in *2021 IEEE International Conference on Mechatronics (ICM)*, 2021, pp. 1–6.

[35] M. Loey, G. Manogaran, M. H. N. Taha, and N. E. M. Khalifa, "Fighting against covid-19: a novel deep learning model based on yolo-v2 with resnet-50 for medical face mask detection," *Sustainable Cities and Society*, vol. 65, p. 102600, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2210670720308179

[36] M. M. Rahman, M. M. H. Manik, M. M. Islam, S. Mahmud, and J.-H. Kim, "An automated system to limit covid-19 using facial mask detection in smart city network," in *2020 IEEE International IOT, Electronics and Mechatronics Conference (IEMTRONICS)*, 2020, pp. 1–5.

[37] A. Chavda, J. Dsouza, S. Badgujar, and A. Damani, "Multi-stage cnn architecture for face mask detection," in *2021 6th International Conference for Convergence in Technology (I2CT)*, 2021, pp. 1–8.

[38] X. Kong, K. Wang, S. Wang, X. Wang, X. Jiang, Y. Guo, G. Shen, X. Chen, and Q. Ni, "Real-time mask identification for covid-19: an edge-computing-based deep learning framework," *IEEE Internet of Things Journal*, vol. 8, no. 21, pp. 15 929–15 938, 2021.

[39] NVIDIA, "Implementing a real-time, ai-based, face mask detector application for covid-19," [Online]. Available: https://developer.nvidia.com/blog/implementing-a-real-time-ai-based-face-mask-detector-application-for-covid-19/

[40] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: a large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.

[41] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: efficient convolutional neural networks for mobile vision applications," 2017.

[42] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: inverted residuals and linear bottlenecks," in *2018 IEEE/CVF*

Conference on Computer Vision and Pattern Recognition, 2018, pp. 4510–4520.

[43] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, Q. V. Le, and H. Adam, "Searching for mobilenetv3," 2019.

[44] TensorFlow MobileNetV3, " tf.keras.applications.mobilenetv3large,"[Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/applications/MobileNetV3Large

[45] A. Howard and S. Gupta, " Introducing the next generation of on-device vision models: Mobilenetv3 and mobilenetedgetpu," *Google Research*, 2019. [Online]. Available: https://ai.googleblog.com/2019/11/introducing-next-generation-on-device.html

[46] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," 2015.

[47] TensorFlow, " Tensorflow 1 detection model zoo," [Online]. Available: https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf1_detection_zoo.md

[48] " Tensorflow lite," [Online]. Available: https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite

[49] M. Dukhan and F. Barchard, " Faster quantized inference with xnnpack," [Online]. Available: https://blog.tensorflow.org/2021/09/faster-quantized-inference-with-xnnpack.html

[50] N.X.P., "eiq machine learning software development environment." [Online]. Available: https://www.nxp.com/docs/en/fact-sheet/EIQ-FS.pdf

[51] N. Semiconductor, "Nxp enables a deeper view to machine learning," [Online]. Available: https://www.nxp.com/company/blog/nxp-enables-a-deeper-view-to-machine-learning:BL-MACHINE-LEARNING-DEVELOPMENT

[52] A. Demidovskij, Y. Gorbachev, M. Fedorov, I. Slavutin, A. Tugarev, M. Fatekhov, and Y. Tarkan, "Openvino deep learning workbench: comprehensive analysis and tuning of neural networks inference," in *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, 2019, pp. 783–787.

[53] N.V.I.D.I.A., " Accelerating inference in tf-trt user guide," [Online]. Available: https://docs.nvidia.com/deeplearning/frameworks/tf-trt-user-guide/index.html

[54] " Nvidia tensorrt," [Online]. Available: https://developer.nvidia.com/tensorrt

[55] P. Bhandary, [Online]. Available: https://github.com/prajnasb/face_detector/tree/master/dataset

[56] C. Deb, " Face mask detection," [Online]. Available: https://github.com/chandrikadeb7/Face-Mask-Detection

[57] "Face mask detection," [Online]. Available: https://www.kaggle.com/andrewmvd/face-mask-detection

[58] V. Jain and E. Learned-Miller, "Fddb: a benchmark for face detection in unconstrained settings," University of Massachusetts, Amherst, Tech. Rep. UM-CS-2010-009, 2010.

[59] T. Fawcett, "An introduction to ROC analysis," Pattern Recognition Letters, vol. 27, no. 8, pp. 861–874, 2006.

[60] T.-Y. Lin et al., "Microsoft COCO: Common Objects in Context." 2014.

[61] G. Bradski, "The opencv library," *Dr. Dobb's Journal of Software Tools*, 2000.

[62] L. Huang, "Pyturbojpeg," [Online]. Available: https://github.com/lilohuang/PyTurboJPEG

[63] "Python imaging library," [Online]. Available: https://github.com/python-pillow/Pillow

[64] "Pillow-simd," available:. [Online]. Available: https://github.com/uploadcare/pillow-simd

[65] Google, " Flatbuffers," [Online]. Available: https://google.github.io/flatbuffers

[66] M. Oršić, I. Krešo, P. Bevandić, and S. Šegvić, "In defense of pre-trained imagenet architectures for real-time semantic segmentation of road-driving images," 2019.

[67] Google, " Edge tpu performance benchmarks." [Online]. Available: https://coral.ai/docs/edgetpu/benchmarks/

[68] A. Gholami, K. Kwon, B. Wu, Z. Tai, X. Yue, P. Jin, S. Zhao, and K. Keutzer, "Squeezenext: hardware-aware neural network design," 2018.

[69] NVIDIA Forums, " Extremely long time to load trt-optimized frozen tf graphs." [Online]. Available: https://forums.developer.nvidia.com/t/extremely-long-time-to-load-trt-optimized-frozen-tf-graphs/69628/14

[70] J. Jung, " Tensorflow/tensorrt (tf-trt) revisited." [Online]. Available: https://jkjung-avt.github.io/tf-trt-revisited/

DIMITRIOS KOLOSOV has received his Bachelor of Engineering (BEng) in Electronic and Electrical Engineering from Edinburgh Napier University in 2016. He completed his Master of Science (MSc) in Embedded Intelligent Systems from University of Hertfordshire in 2021, and then joined the School of Physics, Engineering and Computer Science to pursue a Doctor of Philosophy (PhD). His research interests include the development and acceleration of novel and innovative techniques for machine learning algorithms performing on various hardware platforms that are aimed for edge deployment, mainly vision and energy disaggregation. He has a strong industry experience in FPGAs.

VASILIOS KELEFOURAS is an Assistant Professor in computer science at University of Plymouth. His main research interests are in the areas of code optimization, loop transformations, High Performance Computing and optimizing compilers. He has strong R&D experience in optimizing software applications, in terms of execution time, energy consumption and memory size, in a wide range of different hardware platforms. He has published more than 40 research papers.

PANDELIS KOURTESSIS is Director of the Centre for Engineering Research and Reader in Communication Networks at the University of Hertfordshire, U.K., leading the activities of the Networks Engineering Research Group into Communications and Information engineering, including Next Generation Passive Optical Networks, Optical and Wireless MAC Protocols, 5G RANs, Software Defined Network & Network Virtualization 5G and Satellite Networks and more recently Machine Learning for Next Generation Networks. His funding ID includes EU COST, FP7, H2020, European Space Agency (ESA), UKRI and industrially funded projects. He has served as general chair, co-chair, technical programme committee member and at the scientific committees and expert groups of IEEE workshops and conferences, European Technology Platforms and European Networks of Excellence. He has published more than 80 papers at peer-reviewed journals, peer-reviewed conference proceedings and international conferences. His research has received coverage at scientific journals, magazines, white papers and international workshops. He has been co-editor of a Springer book and chapter editor of an IET book on softwarization for 5G.

IOSIF MPORAS is Reader in Signal Processing and Machine Learning (Associate Professor) at the University of Hertfordshire, UK. He holds a Diploma (5-years degree) in Electrical and Computer Engineering from the University of Patras, Greece, in 2004 and a PhD degree from the same University, in 2009. He has participated in more than 10 EU- funded R&D projects as Researcher, Senior Researcher and as Principal Investigator. His research interests include applications of signal processing and machine learning. He serves as reviewer of grant applications, reviewer of international journals and Programme Committee member in international conferences, while he was the General Chair of the joint SPECOM/ICR 2017 conference and the Technical Chair of the ICESF 2020 conference. He has authored or co-authored more than 100 articles in international journals and conferences cited more than 1,500 times (h-index: 19).