

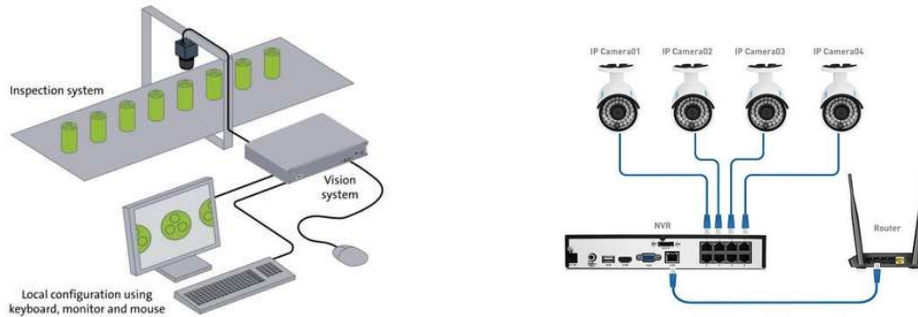
# Classification of images in the YUV color space

Gianluca Filippini / EBV Elektronik  
gianluca.filippini@ebv.com

This document describes the study and implementation phases of an image classification algorithm for “*embedded*” devices, used for applications in the industrial electronic field. The proposed solution optimizes system performance by eliminating the color space conversion of incoming data and employs a convolutional neural network for data processing. The final implementation uses an “embedded / linux” type device ( *Raspberry Pi CM4*, *ARM Cortex-A72 cpu* ) together with a co-processor dedicated to neural computing ( *Hailo8* ).

## 1 Introduction

In the industrial field, the use of “*deep learning*” techniques for the classification of images is widespread. Some application examples are the classification of objects arranged on a production line (search for defects, selection of materials, selection of agricultural products, etc.) or the analysis of images from remote cameras over Ethernet connections (video surveillance, *network video recording* etc.)



The design of an image analysis device must meet a very wide range of requirements, including features such as:

- Image capture speed ( *performance video capture* )
- Analysis delay on the single image ( *inference latency* )
- Number of video streams ( *streams, throughput* ) to be analyzed concurrently ( *realtime multi-stream* )
- Energy efficiency and thermal efficiency
- Hardware certifications for extreme environments (dust, temperature, humidity, *automotive certification*)
- Total cost ( *BOM: bill-of-material* ) of the *vision* system

Particular emphasis is placed on the time delay of analysis of the single image as the system is normally composed of: acquisition sensors, one or more analysis devices, actuators (mechanical or electrical) that act on the system itself based on the results obtained.

The conditions set out above mean that a considerable percentage of industrial devices use “*embedded*” type processors, typically with ARM and / or x86 architectures (replacing previous systems based on Personal Computer).

*Embedded* processors offer cost, longevity and energy efficiency advantages while introducing limitations on computing capacity. In particular, for today's most popular devices, there is no GPU ( *graphic processing unit* ) that can be used as a co-processor. Alternatively, single peripheral units dedicated to specific functions (audio or 2D / 3D graphics and video codecs) are added.

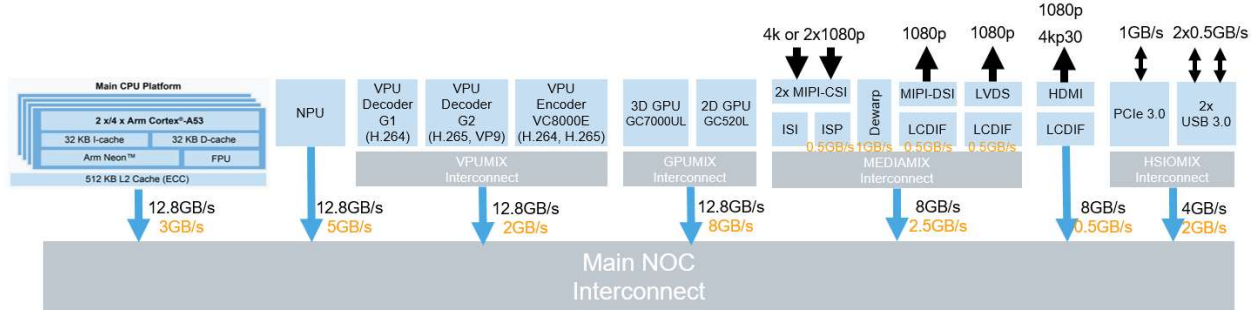


Figure 1: embedded system architecture

The figure shows the main CPU and multiple dedicated units (*NPU: neural processing unit*, *VPU: video processing unit*, *ISP image sensor processing* ). All units communicate via an interconnection bus ( *NOC Interconnect: network on chip interconnect* ). Each unit has a maximum communication speed limit on the interconnection bus.

## 1.1 State of the art

The data processing architecture for these systems has a sequential *pipeline structure*: **data acquisition** , **analysis** , **drive** . The data comes from local cameras (with USB or MIPI-CSI or PARALLEL-CAMERA connection) or remote cameras (typically Ethernet connection) or from previously stored video streams. Image analysis is based both on classical *Computer Vision* (CV) techniques and on the most recent *Machine Learning* (ML) techniques and in particular with *neural networks* .

Regardless of the system architecture, you will have a data flow diagram described as follows (fig. 2). The images are acquired in RGB 4: 4: 4 format and subsequently processed to obtain a format suitable for the execution of the neural network. The result of the neural network inference determines a decision / action on the system and / or contributes to further video encodings accompanied by the analysis information.

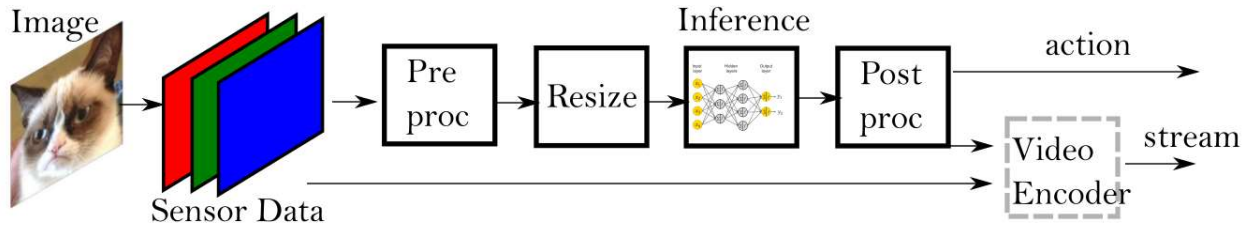


Figure 2: Sequential flow of data processing

In the field of Computer Vision (CV) it is common practice to consider images as data matrices ( *pixels* ) in RGB 4: 4: 4 format. Consequently, much of the literature for the study of neural networks for image processing has produced reference models with RGB input.

In fact, for many of the industrial applications listed above, the RGB 4: 4: 4 **format is not the native format of the input data** . In the event that video streams over Ethernet protocol are used (or pre-

recorded files), the images are encoded (compressed) according to the H.264 / AVC <sup>1</sup> or H.265 / HEVC standard <sup>2</sup> or in alternative formats of lesser diffusion such as M-JPEG ( *motion jpeg* ). In most encoders / decoders ( *codecs* ) for video streams the images are stored in YUV 4: 2: 0 format <sup>3</sup>. This format is better suited to the need to reduce the amount of bit / s ( *data-rate* ) necessary for transmission as, by exploiting some peculiarities of human vision, it separates the Luminance (Y) data from the Chrominance (UV) data and reduces ( *spatial subsampling* ) the size of the UV data, keeping the image acquisition frequency unchanged ( *temporal sampling, frame-rate* ).

The same applies to applications with cameras on the MIPI-CSI or PARALLEL-CAMERA bus where the need to reduce the data transmission band on the bus itself requires the use of YUV formats (4: 2: 2 or 4: 2: 0). See the definition of the MIPI standard: <https://www.mipi.org/specifications/csi-2>

## 1.2 Definition of the processing flow

The need to use neural networks (generally based on data in the RGB color space) and at the same time to use data from YUV type sources is one of the most important causes of the increase in computational complexity for *embedded systems* .

Image processing requires reading / writing of data stored in memory ( *RAM* ). In general, these *pixel processing operations* are not optimal for classic CPU architecture and are better suited to dedicated GPU ( *graphics processing unit* ) architectures. The GPUs available in the *embedded field* are highly optimized for specific tasks for reasons of electricity consumption, construction cost (silicon area), and market *royalties* . If compared with the GPUs used on normal *workstations* , they show extremely reduced performance and implement only a part of the graphics functions for numerical calculation. Referring to the figure (fig. 3) we can indicate *specific co-processors* for graphic processing functions (2D or 3D), image acquisition ( *ISP* ), image encoding / decoding ( *CODEC* ), neural network execution ( *NPU neural processing unit* ) .

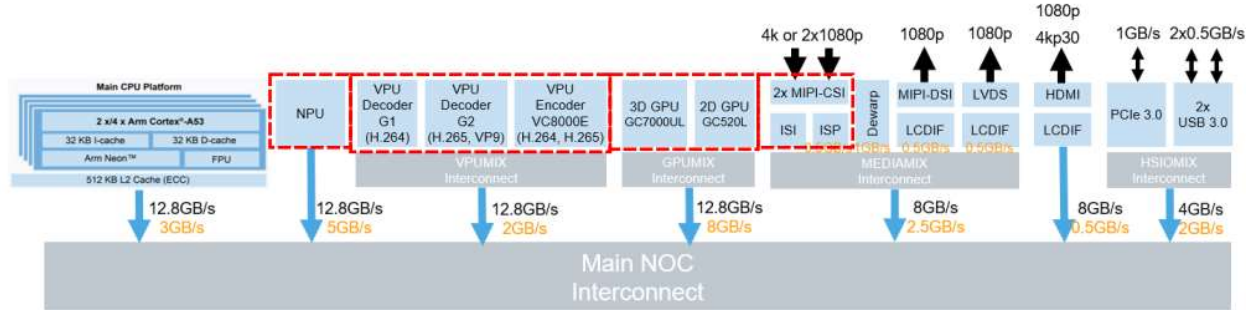


Figure 3: Components required for smart camera applications

It is important to know the hardware architecture of the devices used to understand what are the causes of loss of performance , during the software processing of incoming data through neural networks.

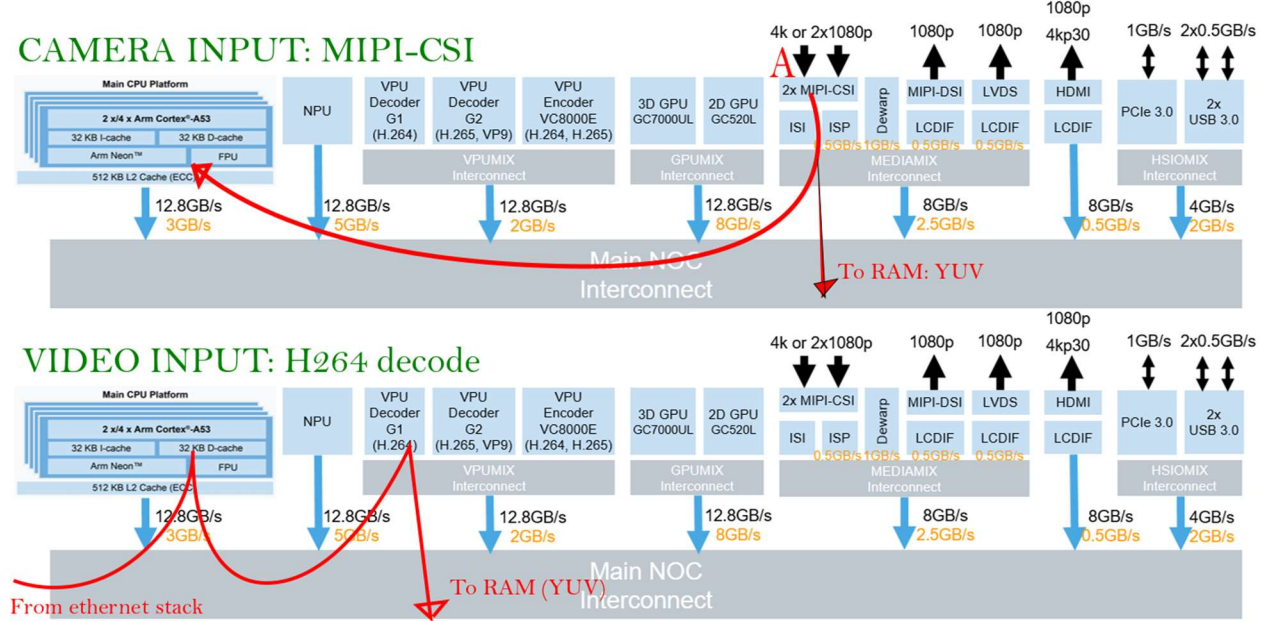
<sup>1</sup>H.264 / AVC: <https://www.itu.int/rec/T-REC-H.264>

<sup>2</sup>H.265 / HEVC: <https://www.itu.int/rec/T-REC-H.265>

<sup>3</sup>Chroma Subsampling: [https://en.wikipedia.org/wiki/Chroma\\_subsampling](https://en.wikipedia.org/wiki/Chroma_subsampling)

The following figures describe the main phases of data transfer and processing, remembering that images and intermediate results are stored in the system memory (RAM), located on the same interconnection bus as the devices.

The first phase of image acquisition takes place through the dedicated MIPI-CSI *hardware blocks*. The data are stored in RAM memory in native format (YUV or RGB according to the considerations described above). In the case of a video stream we have an additional input stage. The data are received on ethernet protocol and processed by the CPU, subsequently sent to the hardware decoder to obtain the images to be stored in RAM.



It is important to underline that every data transfer requires an arbitration of the interconnection bus and access to the memory in read / write. Simplifying, it can be stated that the greatest impact on consumption and processing times can be referred to three situations:

- Limited by CPU: i.e. the main CPU architecture is not optimized for the specific data type (in our case parallel pixel processing). In this case, the dedicated co-processors are activated.
- Limited for *read / write access* : the amount of data to be transferred saturates the maximum capacity of one or more ports on the interconnection bus (see for example the capacity of each component indicated in the figure)
- Limited by power consumption: the system specifications place limits on the amount of data that can be processed in a certain time interval. It is usually a secondary limitation compared to the previous ones.

Once the data in native format has been acquired, there can be one or more *pre / post processing stages* to adapt the images to the format required by the neural network chosen.

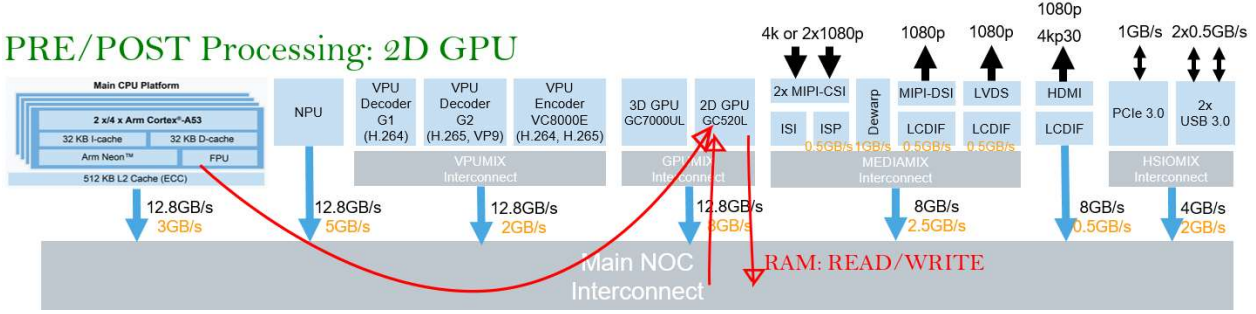
Among these the most common are:

- Interpolation ( *resampling / resizing* ): to adapt the size of the images to the size required by the input stage of the neural network.

- Normalization: neural networks benefit from a preprocessing stage (normalization <sup>4</sup>) which maximizes the performance of the network itself.
- Color conversion: if the neural network is designed on the basis of a specific color space of the input data, it is necessary to convert from the "native" space (for example YUV) to the space indicated by the *neural network* (usually RGB)

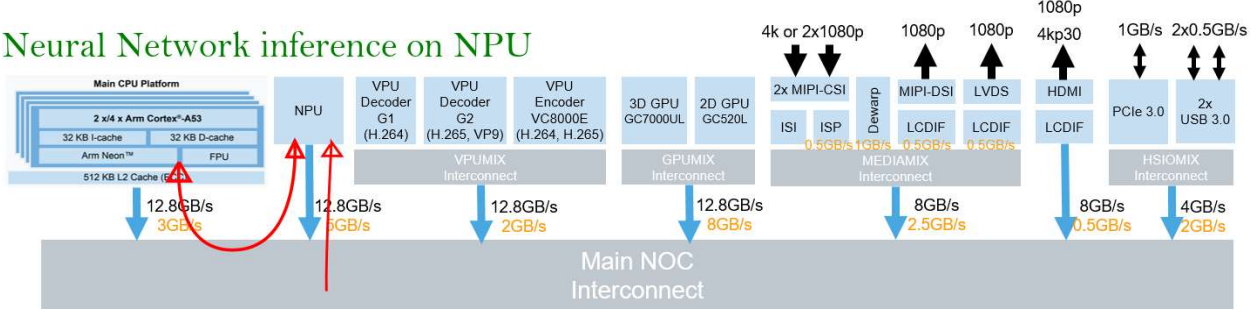
These processing stages can be performed on the CPU or via a co-processor. Either way it will impact system performance due to processing and data transfer.

### PRE/POST Processing: 2D GPU



Finally, also the execution of the inference stage *on* the NPU requires an access to the memory both for the reading of the model and for the reading of the pre-processed images.

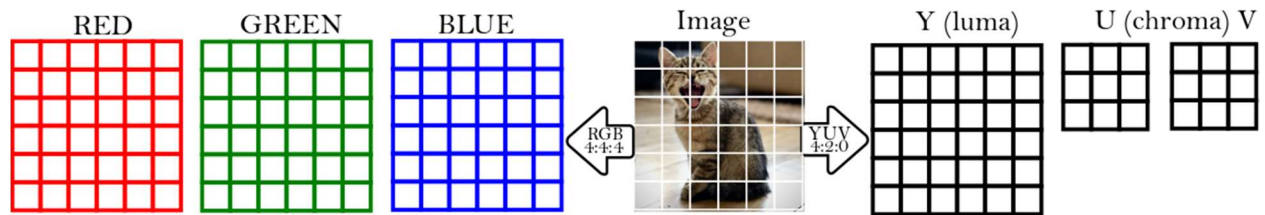
### Neural Network inference on NPU



## 1.3 Problem definition

The images in the YUV 4: 2: 0 color space allow a halving of the band (bit / s) necessary for data transfer compared to a format without subsampling of the chroma signal (eg RGB 4: 4: 4).

Displaying the images as *pixel matrices* and considering a storage with 8bit / pixel we can represent the two formats as follows ( *planar representation* ) :



<sup>4</sup>Normalization Techniques in Training DNNs: Methodology, Analysis and Application. (Lei Huang, Jie Qin, Yi Zhou, Fan Zhu, Li Liu, Ling Shao) <https://arxiv.org/pdf/2009.12836.pdf>

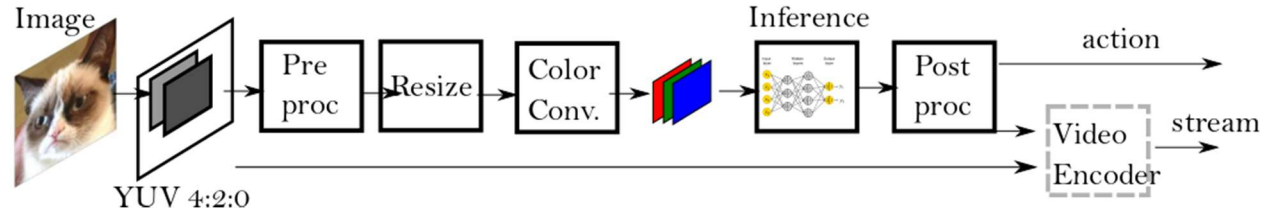


In the image, each input pixel is represented by a memory cell (8bit, in the case under examination) in each of the color space planes of the native (source) data processing format. Thanks to 4: 2: 0 subsampling, the YUV format uses half the amount of memory of the RGB 4: 4: 4 format.

It is important to note that this advantage is even more significant as the image size increases, significantly reducing the system requirements (memory, transfer bandwidth) for handling high resolution images. The *systemic advantage* is therefore given by the reduced format of the chrominance color planes while the *perceptual advantage* is obtained thanks to the greater sensitivity of human vision towards luminance information compared to chrominance information.

The images acquired (or decoded) in the native color space YUV 4: 2: 0 are however not suitable for processing via neural networks when the network itself has been designed starting from RGB 4: 4: 4 images. In literature this is the most common case, for historical reasons and for a direct correlation with the electronic image acquisition format of physical sensors (see Bayer Pattern <sup>5</sup>)

Referring to the general data processing scheme, we note the introduction of a further *color conversion stage* to adapt the images to the format used by the neural network.



This further processing block is necessary for the subsequent inference through the neural network and introduces some performance disadvantages:

- If the architecture offers a dedicated co-processor it is necessary to transfer twice the amount of data compared to the input signal due to over-sampling (transition from 4: 2: 0 format to 4: 4: 4 format)
- If the color conversion is calculated by the main CPU it will have a not negligible computational cost, proportional to the size of the input image.

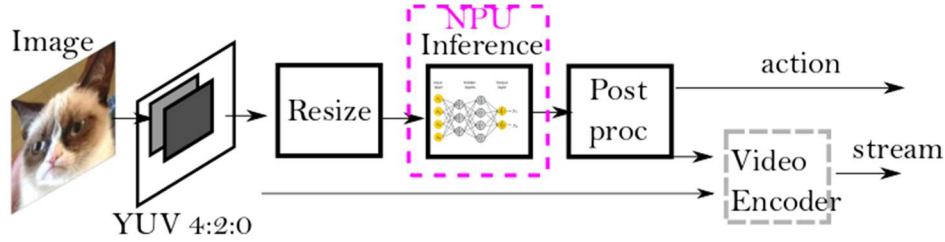
If the system is equipped with an NPU co-processor for running neural networks, the delay due to *color conversion* (CPU) can mean a loss of performance compared to the theoretical maximum achievable by the co-processor.

In the final implementation ( *chap.7.5* ) the NPU co-processor will not only perform the computation for the inference on the input data but will also take care of the *normalization* of the same, leaving the CPU to carry out the resolution change ( *resize* ) and *color conversion*.

The performance comparison must be made between a “classic” scheme (previous figure with *color conversion* ) and an “optimized” scheme where the neural network has been trained on data in the YUV 4: 2: 0 color space.

---

<sup>5</sup>RGB "Bayer" Color and MicroLenses: <http://www.siliconimaging.com/RGB%20Bayer.htm>



## 1.4 Notes for the YUV color space

In this document we refer to the generic RGB-> YUV conversion without giving further indications on the source and destination color space. The standards that define digital formats for images ( *still pictures* ) or for video streams ( *moving pictures* ) indicate precisely which are the references for correct color reproduction based on the characteristics of the images themselves (resolution, bit / pixel etc.) .

See for example the documents:

- Recommendation ITU-R BT.709-6 <sup>6</sup>of 06/2015: Parameter values for the HDTV standards for production and international program exchange
- Recommendation ITU-R BT.2020-2 of 10/2015 <sup>7</sup>: Parameter values for ultra-high definition television systems for production and international program exchange
- ITU-T H.264 / AVC <sup>8</sup>Advanced video coding for generic audiovisual services

It should be noted that the YUV indication is actually generic, often confused with the more appropriate YCbCr (used for example in H.264 encoding). Following the minimal differences in color conversion used in the industrial standards in the final implementation, for the purposes of this document a single RGB-YUV conversion matrix was used to the detriment of the *exact (numerically)* color representation ( *color matching, white / black point matching* ) of the single pixel. ([https://en.wikipedia.org/wiki/Color\\_management#Embedding](https://en.wikipedia.org/wiki/Color_management#Embedding))

This decision was also made by virtue of the neural network's tolerance to color variations for image classification. By doing so, a *trained* neural network can be used for a " *not exact* " YUV color space compared to that of the source images while obtaining the correct final classification (within a predetermined tolerance interval for the color / brightness variation)

It is also noted that:

- Although the image encoding standards (JPEG, H264, H265) provide the tools ( *metadata* ) for indicating the color space, some software libraries (see PIL <https://python-pillow.org/>) do not fully implement the support of this information, restricting the field of use only to some formats (Rec.601, Rec.709)
- When the images are acquired by a real system, a complete calibration (called *end-to-end* ) of the acquisition chamber is necessary in order to have an exact representation from the point of view of

<sup>6</sup><https://www.itu.int/rec/R-REC-BT.709>

<sup>7</sup><https://www.itu.int/rec/R-REC-BT.2020>

<sup>8</sup><https://www.itu.int/rec/T-REC-H.264>

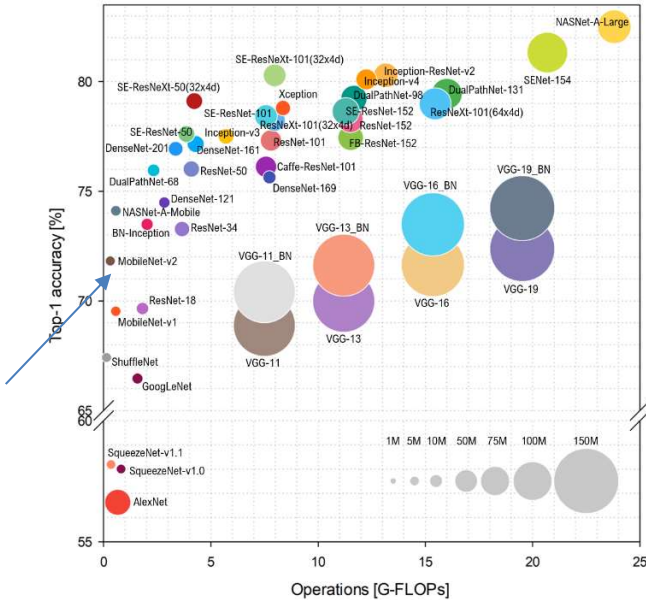
color measurement ( *full color calibration* ). Wide variations in external conditions can alter the numerical correctness of the purchased data.

All this is not strictly necessary (within the tolerances given by the only conversion matrix used in the implementation phase) for the purposes of classification via the neural network. The color variation is considered part of the training phase of the net itself.

## 2 Neural network: MobileNet-V2

The MobileNet-V2 convolutional network was chosen for the final implementation <sup>9</sup>. The choice is justified by reasons for comparing the results (see following paragraphs) and by a specific request for a prototype study (corporate customer).

Although MobileNetV2 is an architecture dated 2018 ( *MobileNet-V2: Inverted Residuals and Linear Bottlenecks*, *arXiv: 1801.04381* ) it is still widely used in the industrial field due to the particular balance between computational complexity and accuracy of results. MobileNet-V2 was born as an evolution of the previous MobileNet-V1 ( *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications* *arXiv: 1704.04861* )



In the original publication, the term “MobileNet” defines a family of architectures on the basis of some parameters that alter their computational complexity and computational latency. **In this document, reference is made to the "base" version indicated as MobileNetV2-224-1.0 .**

The low complexity <sup>10</sup> and sufficient accuracy make it attractive for all industrial image classification solutions, especially if the system has a “controlled” work environment (not subject to extreme environmental variables).

Subsequently, the MobileNet family was also used for *object-detection applications* , especially as part of *SSD: Single Shot Multibox Detector* (*arXiv: 1512.02325*), not implemented in this document (see: future developments)

The next version "V3" introduces further optimizations in terms of computational complexity and accuracy. This version has not been used in order to maintain the comparison with the PyTorch implementation in *python code for embedded devices* indicated in the final part of this document.

<sup>9</sup><https://paperswithcode.com/paper/mobilenetv2-inverted-residuals-and-linear>

<sup>10</sup>Benchmark Analysis of Representative Deep Neural Network Architectures <https://arxiv.org/pdf/1810.00736.pdf>



In the following paragraphs we want to demonstrate the feasibility of image analysis with convolution networks starting from the YUV 4: 2: 0 color space by modifying the network input structure to manage the two input tensors (Y and UV). It should be noted that the type of network chosen is independent of the concept of analysis in the YUV space. This technique can be applied to different architectures by measuring their effectiveness as a comparison with the original network (RGB)

## 2.1 Architecture

The MobileNet family was born in the Google laboratories in 2017/18 <sup>11</sup>with the aim of allowing image processing through convolutional networks on mobile devices such as the Google PixelPhone <sup>12</sup>. The architecture is based on the state of the art of the historical period and specifically on convolutional networks of the "sequential" type such as the well-known VGG-Net ( *Very Deep Convolutional Networks for Large-Scale Image Recognition arXiv: 1409.1556* ).

MobileNet aims to drastically reduce computational complexity and at the same time reduce latency to obtain results in real time even on “ *embedded* ” devices such as a *smartphone* .

The first fundamental structure of this architecture is based on the decomposition of a classic Conv-2D *layer* (2D convolution) in the combination of two components: a *depthwise separable convolution* and a subsequent *point-wise convolution* , united in a single computational block called " *Depthwise Separable Convolution Block* ". This element was already introduced in the first version of the MobileNet family.

The second key structure consists of combining *depthwise / pointwise convolution* with additional expansion / projection *layers* to form a block called the “ *Bottleneck Residual Block* ”. This second structure also adds the “ *bypass* ” connection derived from the previous ResNet ( *Deep Residual Learning for Image Recognition arXiv: 1512.03385* ) to optimize training performance on very deep networks ( *deep neural networks* ). As per the literature, the modular concatenation / replication of these structures allows the creation of more or less deep, more or less complex MobileNet networks.

The third fundamental element (coming from the first version of MobileNet) consists in the introduction of some hyperparameters that control the modular structure of the network itself. The most important is called a “ *depth multiplier* ” (denoted by *alpha* ). It is a multiplication factor that controls the number of channels used in each level of the network ( *layer* ) . A second parameter is called “ *resolution multiplier* ” (denoted by *roh* ) and controls the spatial resolution of each level. The two parameters allow to scale the computational complexity (to the detriment of performance) while maintaining the structure of the network *uniform* with respect to the reference one (with unitary parameters).

## 2.2 Depthwise Separable Convolution Block

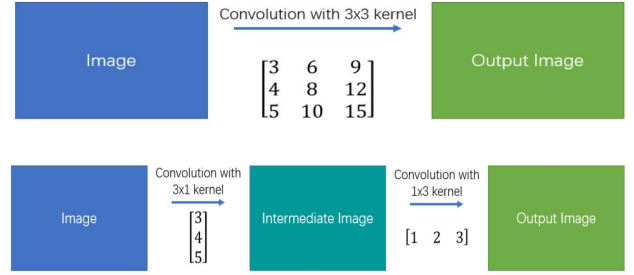
---

<sup>11</sup><https://ai.googleblog.com/2017/06/mobilenets-open-source-models-for.html>

<sup>12</sup>[https://it.wikipedia.org/wiki/Google\\_Pixel](https://it.wikipedia.org/wiki/Google_Pixel)

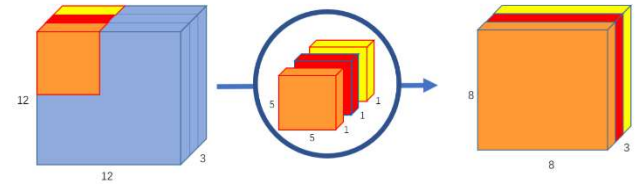
The *Depthwise Separable Convolution operator* had previously been studied by its author **Laurent Sifre** for his PhD dissertation “ *Rigid-Motion Scattering For Image Classification* <sup>13</sup> ” at the Ecole Polytechnique (France).

This technique is an alternative to using a convolutional MxN filter ( *kernel* ). In the separable convolution, two filters Mx1 and Nx1 are used (see figure).



The filters are applied independently in the direction of the depth ( *depth* ) or the number of "channels" of the tensor entering the convolution. The number of filters applied determines the depth of the tensor resulting from the convolution operation.

Laurant Sifre, in his dissertation, demonstrates the effectiveness of the two operations (classic Conv2D and *Depthwise* ) for the final result (accuracy of convolutional neural networks).



The block structure is completed by a 1x1 *Pointwise convolution* placed in cascade to the previous 3x3 Depthwise Conv2D.

**This block replaces a classic Conv2D layer reducing its computational complexity.**

A Conv2D layer, considering a number of input channels ( *depth* ) M and a number of output channels N with a kernel (Dk x Dk) will have a computational complexity equal to:

$$(Dk \times Dk \times M) \times N \times (Df \times Df)$$

Where Df is determined by the number of " *features* " required (no. Filters).

In the case of the *depthwise / pointwise combination* we will have:

$$\{(Dk \times Dk \times M) \times Df \times Df\} + \{M \times N \times Df \times Df\}$$

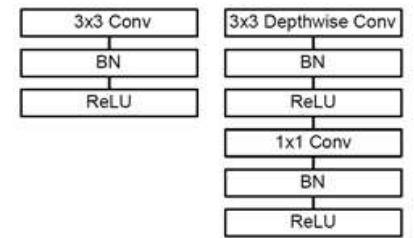


Figura 4: Conv2D classica e Depthwise (con batch normalization e attivazione)

Where the first addendum represents the *depthwise contribution* while the second adds the *pointwise contribution* . In the original publication the reduction of computational complexity equal to:

$$\text{computational\_cost\_reduction} = 1 / N + 1 / Dk^2$$

the reduction in complexity is proportional to the size of the *kernel* used. In the case of MobileNet all *kernels* are 3x3 in size and therefore the reduction is about 1/9.

<sup>13</sup>[https://www.di.ens.fr/data/publications/papers/phd\\_sifre.pdf](https://www.di.ens.fr/data/publications/papers/phd_sifre.pdf)

## 2.3 Bottleneck Residual Block

If the *depthwise + pointwise block* was already introduced in the first version of MobileNet, the second version recovers the concept of “residual block” derived from the ResNet network.

The structure of the residual block has the combination *depthwise + pointwise convolution* (last two stages of the block in the figure) and adds a further stage of 1x1 convolution called "expansion".

In parallel to the sequence of convolutions we find the “*shortcut*” connection (or *skip*) derived from ResNet and applied with the same function (optimization of the backward propagation of training gradients).

The first 1x1 convolution is called *expansion* because it increases the number of channels used within the convolutional branch. The second 1x1 convolution is called *projection* and reduces the number of channels to perform the sum of the residual branch.

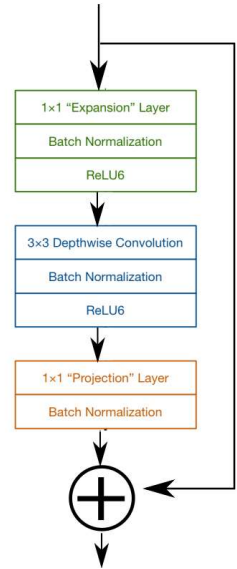


Figura 5 : Bottleneck Residual Block

The final architecture (fig. 6) shows a first Conv2D layer which reduces the size of the input data by operating with a *stride* = 2 .

Subsequently seventeen (17) Bottleneck Residual Blocks are concatenated followed by a final classification part consisting of a 1x1 convolution, a *global pooling*, a *fully-connected* layer and then ending with *softmax* .

In the reference network the input tensor has a size of 224x224x3 pixels (data from the three RGB image channels 4: 4: 4)

## 3 Measurement of the official reference

Before making the necessary changes to carry out the inference phase with a 4: 2: 0 YUV image, it is important to check the “theoretical” reference given by the state of the art.

*Pytorch* provides a reference implementation for the MobileNet family in the TorchVision library . The code is available in open source format:

[https://pytorch.org/hub/pytorch\\_vision\\_mobilenet\\_v2/](https://pytorch.org/hub/pytorch_vision_mobilenet_v2/)

<https://pytorch.org/vision/stable/modules/torchvision/models/mobilenetv2.html>

The code used for this first phase is made available in the “*validate\_mobilenetv2\_pre-trained.py*” file. The result provides the *accuracy / recall* values for each class of the *dataset* together with the overall value on all the classes making up the *dataset* .

Type / Stride	Filter Shape	Input Size
Conv / s2	3 × 3 × 3 × 32	224 × 224 × 3
Conv dw / s1	3 × 3 × 32 dw	112 × 112 × 32
Conv / s1	1 × 1 × 32 × 64	112 × 112 × 32
Conv dw / s2	3 × 3 × 64 dw	112 × 112 × 64
Conv / s1	1 × 1 × 64 × 128	56 × 56 × 64
Conv dw / s1	3 × 3 × 128 dw	56 × 56 × 128
Conv / s1	1 × 1 × 128 × 128	56 × 56 × 128
Conv dw / s2	3 × 3 × 128 dw	56 × 56 × 128
Conv / s1	1 × 1 × 128 × 256	28 × 28 × 128
Conv dw / s1	3 × 3 × 256 dw	28 × 28 × 256
Conv / s1	1 × 1 × 256 × 256	28 × 28 × 256
Conv dw / s2	3 × 3 × 256 dw	28 × 28 × 256
Conv / s1	1 × 1 × 256 × 512	14 × 14 × 256
5× Conv dw / s1	3 × 3 × 512 dw	14 × 14 × 512
Conv / s1	1 × 1 × 512 × 512	14 × 14 × 512
Conv dw / s2	3 × 3 × 512 dw	14 × 14 × 512
Conv / s1	1 × 1 × 512 × 1024	7 × 7 × 512
Conv dw / s2	3 × 3 × 1024 dw	7 × 7 × 1024
Conv / s1	1 × 1 × 1024 × 1024	7 × 7 × 1024
Avg Pool / s1	Pool 7 × 7	7 × 7 × 1024
FC / s1	1024 × 1000	1 × 1 × 1024
Softmax / s1	Classifier	1 × 1 × 1000

Type	Mult-Adds	Parameters
Conv 1 × 1	94.86%	74.59%
Conv D		
Conv 3		
Fully C		

Figura 6 : MobileNet-V2

### 3.1 Definition of the dataset

The *dataset* used is ImageNet, specifically the ILSVRC-2012 version. This version has been selected because it is used as a reference by the PyTorch *regression toolchain* (used for the publication of performances at the release of a new version of the framework).

Note: the data can be accessed from the website (<https://www.image-net.org/>) only after authentication as a *university researcher*. In the absence of this document, the only complete alternative (ie containing all the description files of the dataset itself) is Academic Torrents ([www.academictorrents.com](http://www.academictorrents.com)).

The availability of the complete *dataset* facilitates the writing of the code using the special class available in Pytorch Vision “ `class ImageNet ( ImageFolder )` ”, see for reference:  
<https://github.com/pytorch/vision/blob/main/torchvision/datasets/imagenet.py>

Note: The *dataset* provides the sets for *training* and *validation* of the results and a third metadata file used by the Pytorch class to check the integrity of the *dataset* itself (see: ILSVRC2012\_devkit\_t12.tar.gz)

### 3.2 Measure accuracy

The code uses MobileNet-V2 in *pre-trained form*, exactly as available in the Pytorch framework. Only the inference phase is performed and the results are presented as a *classification report* which confirms the overall accuracy figure of **71.8%**.

This data is the “absolute maximum” value, identical to the indications of the MobileNet-v2 authors in their original document. It also confirms that the *dataset* and the code for reading\_images / inference / evaluation of the data (script “ *validate\_mobilenetv2\_pre-trained.py*”) conforms to what is published in the official tests of PyTorch.



### 3.3 Error distribution

In addition to verifying the precise accuracy data, we wanted to verify the distribution of classification errors by calculating the “ *classification confusion matrix* ”. By reporting the values of the error signal (appropriately amplified) in a graph ( *scatterplot* ) and removing the diagonal component (ie the correct result), we wanted to highlight a predominance of errors in some classes of the *dataset* .

In the graph, each class is associated with its identification number (see file *imagenet\_classes.txt* ). Analyzing the images belonging to the classes with a wide distribution of errors (highlighted by the box at the bottom right)

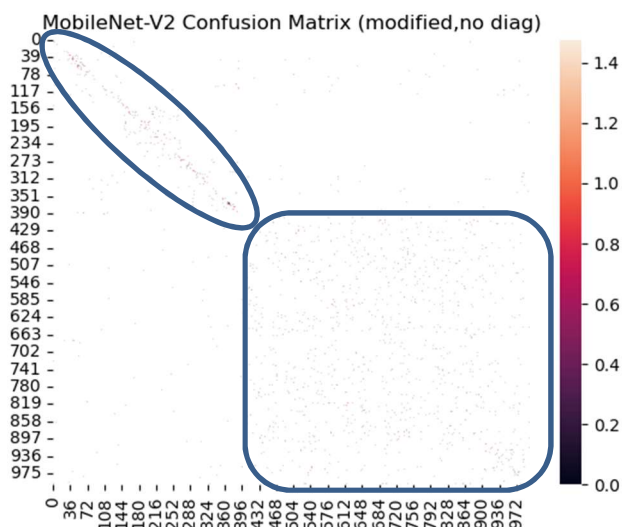


Figura 7 : errori di classificazione MobileNetV2

it was possible to notice that they frequently represent several objects within the same image (more easily subject to mixed / wrong classification)

The distribution of the classification errors is therefore justified as depicted in fig (fig. 7)

## 4 Objective reference measurement (4: 4: 4 RGB input)

The main purpose of this document is to optimize the complete implementation (software application and hardware device) by eliminating the YUV to RGB data conversion step required to use a *pre-towed MobileNet-V2 network* (see previous point) .

A further purpose consists in outlining the whole procedure of analysis and realization of this device starting from real data (dataset: ImageNet) and not from “pre-towed” neural networks.

We therefore wanted to document the choices made to repeat the network *training phase, reporting decisions and results obtained*.

**For the purposes of this document, the result of the "objective" reference is to be considered the best result obtainable with the resources available in the construction phase.**

### 4.1 Hardware configuration

Although MobileNet was released a few years ago, it still has peculiarities today for which the *training phase* becomes very sensitive to the hardware / software configuration used. The training carried out on a group of computers, on a group of GPUs, presents different results from those obtainable on a single computer.

In the specific case, all the official results published by Pytorch are obtained by means of a *distributed training (data-parallel training)* on eight (x8) GPUs.

The official configuration is documented at the web address:

<https://github.com/pytorch/vision/tree/main/references/classification>

And it includes the following notes for hardware and hyper-parameters:

- n.8 NVIDIA Volta V100 GPU
- 8 processes per single CPU
- Number of epochs ( *epochs* ) = 300
- Initial learning-rate = 4.5 e-2
- Weight-decay = 4e-5

*extremely long* training periods to achieve optimal results with MobileNet networks. See for example Ross Wightman's Pytorch Image Models (Timm) ( <https://timm.fast.ai/> ). Also for this implementation of Fast.Ai, cycles of 300-400 *epoch are used* to reach the maximum degree of accuracy.



Note: This information is the basis for the MobileNet-V2 reference of the popular “Papers With Code” <https://paperswithcode.com/model/mobilenet-v2>. The results presented by PapersWithCode are taken from the *torchvision source code* .

**The long training period and the resources required are the first of the limits found for the exact replication of the results obtained with the pre-trained model.**

Recalling the ultimate goal of implementation on an *embedded device* , the optimal result was therefore sought starting from a common hardware configuration. In particular, a single Intel i7-10700K CPU and a single NVIDIA RTX-3090-24G GPU.

With the indicated configuration we tried to obtain the best classification result by limiting the training time to a maximum time of about one hundred (100, max 120) *epoch* . **This is to be considered an important choice and is aimed at facilitating the development of *embedded applications* in companies and laboratories that do not have distributed computing systems .**

The conclusion of these first choices means that the real reference for the optimal performance comparison data between "standard" and "optimized" network is no longer the value 71.8% but the maximum value (67.63%) obtained as explained in the following paragraphs .

## 4.2 Evaluation of the *hyper-parameters*

The training of the MobileNet model is particularly sensitive to the main hyper-parameters: *learning rate*, *batch\_size*, *number of epochs* and regularization methods such as *data augmentation*.

Note: A duration of 100/120 epochs corresponds to approximately sixty (60) hours to complete the MobileNetV2 training with the hardware indicated in the previous paragraph (single GPU)

## 4.3 Learnig-rate

The *learning-rate value* indicated in the notes of the pre-trained Pytorch models is **4.5e-2** . In the case in question it was not considered as a "known value" and an effective search for the optimal value was carried out, with the aim of confirming (or refuting) the indicated value.

In the literature, the document "Cyclical Learning Rates for Training Neural Networks (Leslie Smith)" [<https://arxiv.org/abs/1506.01186>] indicates an *iterative method* for finding the optimal value of learning-rate fixed the remaining hyper-parameters .

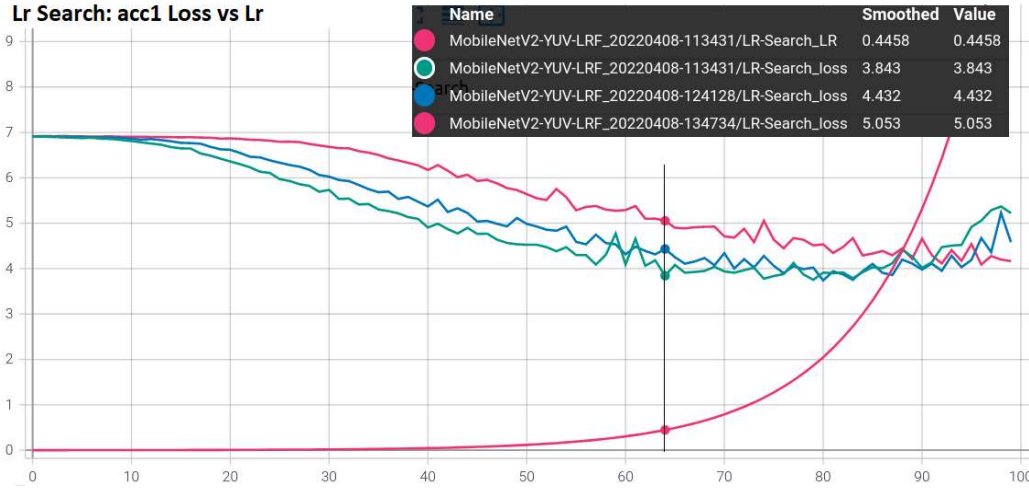
The code implemented in the “ *lr\_finder\_mobilenet.py* ” and “ *lr\_finder\_mobilenet-yuv.py* ” scripts performs an exhaustive search by measuring the accuracy of the model at a linear increase of the Learning-rate value.



Leslie Smith's indications are correct and the value  $4e-2$  /  $4.5e-2$  is also confirmed as the optimal value when the *batch\_size value changes*. In the following figure the curves indicate the *validation\_loss value* for a series of epochs performed with increasing *learning-rate values*. Values for *batch\_size* = 32, 64, 128 are

indicated. Starting from very small *learning-rate values* (about  $1e-4$ ) on the left up to large values ( $> 1$ ) on the right, we note the point of maximum variation in the neighborhood of  $lr = 0.4 / 0.5$ . (NOTE: The abscissa axis indicates the iteration, not the value of Lr)

For this reason, following the indications of L. Smith (reduction  $> 10x$ ), the value  $lr = 4.5e-2$  was chosen as the initial *learning-rate*, confirming the choice of the pre-trained models released by *torchvision*.



#### 4.4 Batch-size

The *batch-size value* to be used proved to be a sensitive parameter to obtain the maximum accuracy “quickly” fixed the number of epochs.

A large *batch-size* (256) shows a very slow training phase, although the accuracy curve on *the validation dataset* is very smooth. Smaller batch sizes result in faster convergence towards the maximum accuracy value.

A further difference concerns the final implementation on an *embedded device*. It is anticipated that for the implementation it will be necessary to “*quantize*” the coefficients of the model from values expressed in *floating-point* (32bit) to values expressed with *integer numbers INT8* (8 bit).

Using large *batch-* sizes allows to obtain more compact distributions of the neural network coefficients and more easily traceable to signal dynamics expressible with integer arithmetic (INT8). Conversely, smaller *batch-sizes* show a large dynamic range of the model coefficients and therefore can cause a greater loss of accuracy in the transition from *floating-point* to *integer*.

In these cases there is no exact solution, it was necessary to compare the results with multiple *batch\_size values* and then select the model with the best performance in integer version.

#### 4.5 Data augmentation and over-fitting

In the literature, especially for optimized architectures such as MobileNet, there is a tendency to increasingly decrease the *data-augmentation (DA) settings* to avoid excessive regularization of the input data. Although the use of *data-augmentation* gives benefits in terms of convergence of the model towards the optimal result,

the scientific community considers the possibility of *under-fitting*, with a consequent failure to achieve absolute maximum performance.

A recent publication (April 7, 2022) also indicates negative effects on some particular classes for ImageNet:

*The Effects of Regularization and Data Augmentation are Class Dependent*

Randall Balestriero, Leon Bottou, Yann LeCun

<https://arxiv.org/abs/2204.03632>

In order to determine the "stable conditions" for a *training* that does not use *data-augmentation techniques*, a test was carried out with `batch_size = 256` and `epoch = 120`. For the model under examination (MobileNetV2) a clear over-fitting trend was found indicated by a constant *validation accuracy together with an increasing validation loss* (indicated by the arrow in figure fig. 8)

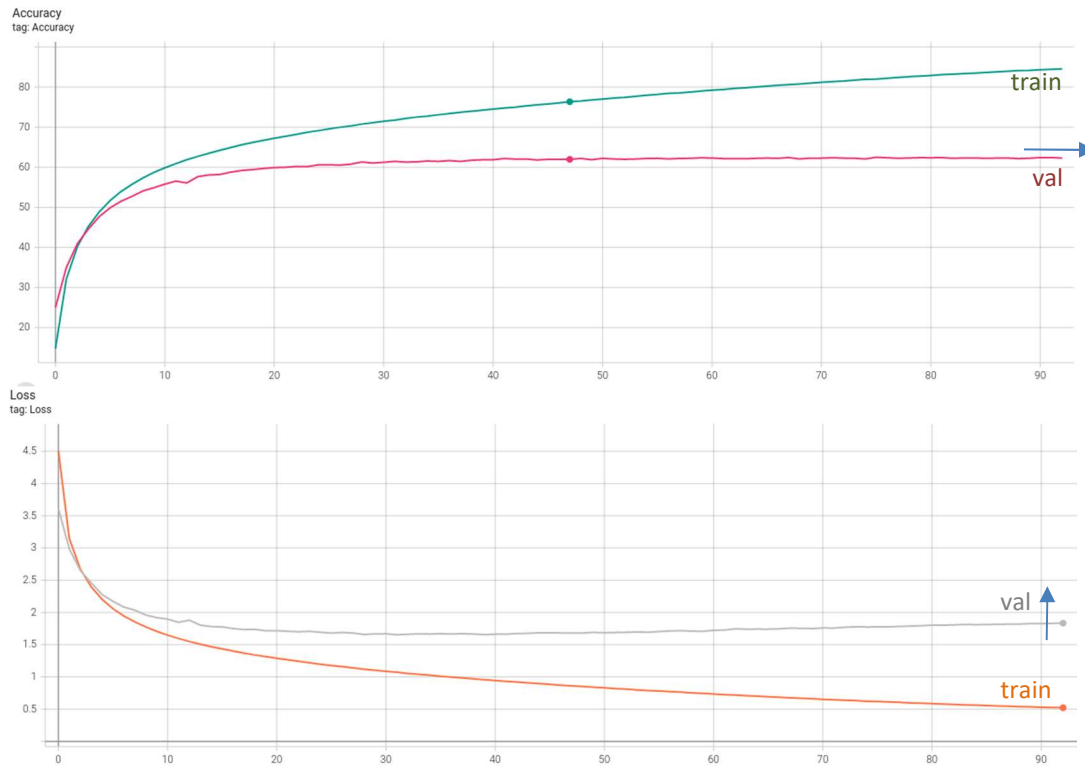


Figure 8: Training / Validation Accuracy: increasing val loss and constant val accuracy, indicative of over-fitting

Various options have therefore been introduced for an increasing use of DA (also acting on variations in color saturation) to contain the tendency to *over-fitting*. For these tests the batch-size was reduced to 128 and subsequently 32 (as per Pytorch reference)

The use of *data-augmentation* together with a smaller *batch-size* made it possible to reduce the tendency of *over-fitting* and at the same time increase the final performance of the model. In the following images the comparison between the two configurations: without DA (in the "Accuracy" graph the value on the validation dataset in purple and in green on the training dataset) and with DA (in the "Accuracy" graph the value on the validation and in blue on the training dataset).

The use of DA confirms a benefit for the *over-fitting effects* and obtains a final result of better accuracy than the test without DA. Consequently, *data augmentation was used* for all subsequent neural network training sessions.

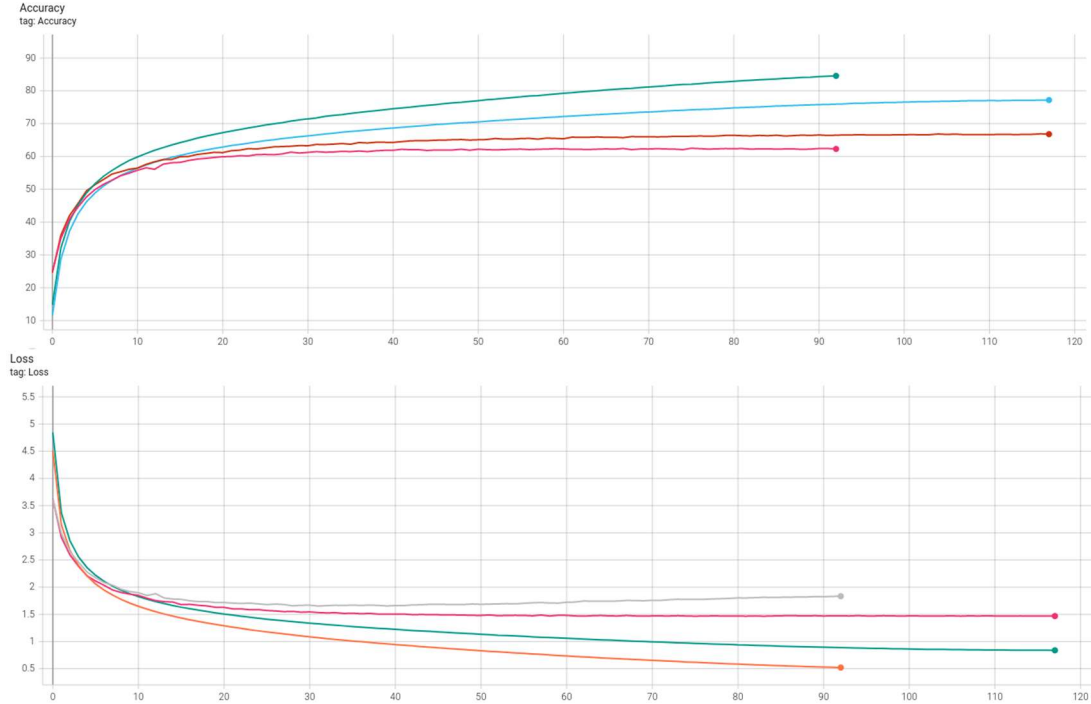


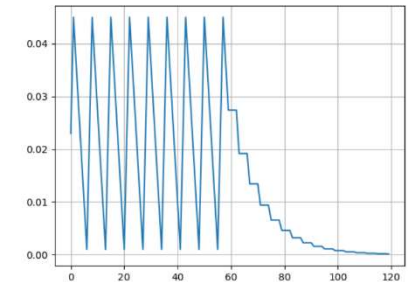
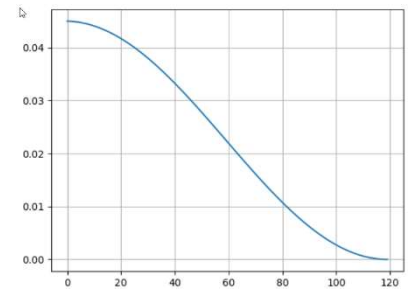
Figure 9: Data Augmentation with BatchSize = 128. Over-fitting reduction

## 4.6 Learnig-rate scheduler

The previously indicated *learning-rate* (  $Lr$  ) value is only the *initial value* of the network *training cycle* . In the literature there are many techniques for modulating this hyper-parameter during the *training cycle* .

The values shown in the graphs of fig. 9 were obtained by decreasing the value of  $Lr$  according to a "cosine" trend, starting from the initial value  $4.5e-2$  and arriving at a value of approximately  $1e-4$  in the overall duration of approximately one hundred and twenty (120 ) *epochs*.

In the paper: " *ESPNetv2: A Light-weight, Power Efficient, and General Purpose Convolutional Neural Network* " <sup>14</sup> the authors present a methodology to make the training phase faster while achieving optimal performance. The proposed solution uses a triangular  $Lr$  scheduler which requires a series of "restart iterations". The initial value of  $Lr$  is modulated with a triangular trend in order to reuse , in each "restart" phase, more



<sup>14</sup> *ESPNetv2: A Light-weight, Power Efficient, and General Purpose Convolutional Neural Network* : <https://arxiv.org/pdf/1811.11431.pdf>

accurate values for the network coefficients, and then proceed with a rapid decline down to the minimum desired Lr value.

This technique was examined (`batch_size = 128`) and compared with the result obtained with the “cosine” scheduler. The reference curves (cosine scheduler) were compared both for `batch_size = 128` (in green in the figure) and for `batch_size = 256` (in orange)

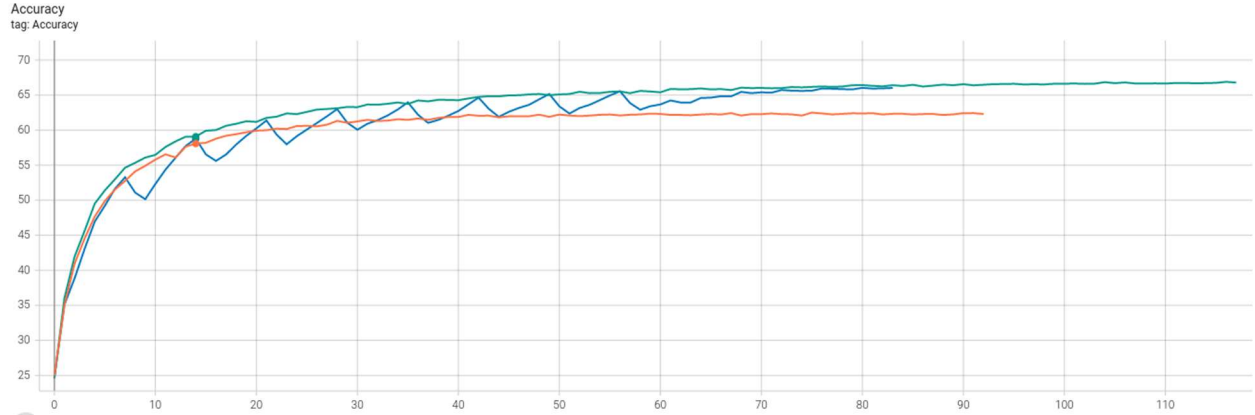


Figure 10: Validation Accuracy, Lr scheduler with restart compared to cosine scheduler

As evidenced by the results in the figure (fig. 10), no improvements were found for the case in question. The curve obtained with the particular scheduler has an "envelope" that can be superimposed on the case of a "cosine" scheduler (using the same `batch_size`)

Given the behavior of the neural network, and considering the use of data-augmentation, the behavior was also verified with a "multi-step" scheduler, in which the Lr is constant in an interval of epochs and decreased considerably in the following interval until completion of the training cycle.

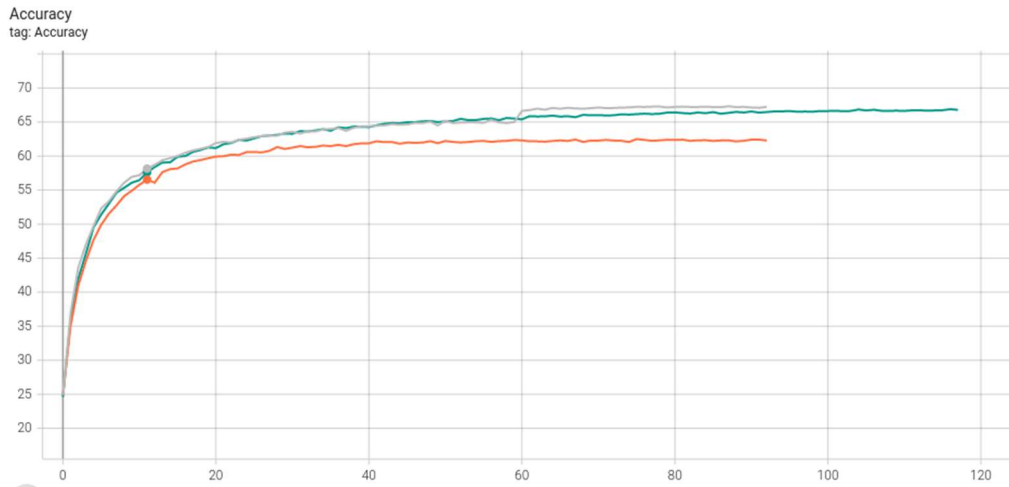


Figure 11 : MobileNetV2 multi-step training, Lr is dropped on epoch=60

The figure (fig.11) shows the comparison of accuracy (validation set) with the previous sessions (Lr scheduler cosine). The value of Lr is reduced by an order of magnitude at the time epoch = 60 and an increase in performance is immediately noted, reaching the final value of about 67%.



To confirm the behavior, a training cycle was carried out over a longer period (about 160 epochs) with a larger `batch_size` value. The figure shows how the training trend of the network has the same increase in

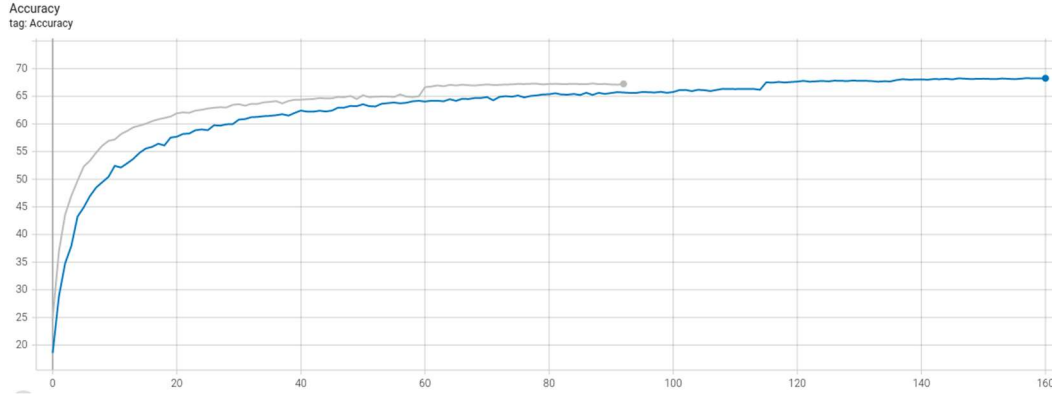


Figura 13 : confronto durata di training con Lr scheduler multi-step.

accuracy in correspondence with the reduction of the *learnin-rate*  $Lr$  . In this case, a final *accuracy value* of 68% is obtained, confirming the need for very long training periods for this type of neural network.

Determining the exact moment to apply the reduction in the value of  $Lr$  is an empirical decision. It requires information from previous training and cannot be configured in advance.

For this reason a "mixed" scheduler was adopted, selecting the "cosine" type scheduler in the first part of the training and carrying out multiple reduction steps in the second part, until the preset number of epochs was reached.

The training cycle on the MobileNetV2 network with the so configured scheduler allowed to obtain a maximum accuracy of 67.57% using about one hundred (100) *epochs*, `batch_size` = 32. The graph of fig.14 shows the comparison with the initial test obtained with the same initial  $Lr$  and `batch_size` with the "cosine" scheduler only (without multi-step in the second part of the training).

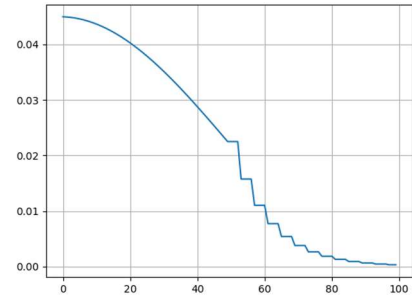


Figura 12: scheduler combinato coseno e multi-step

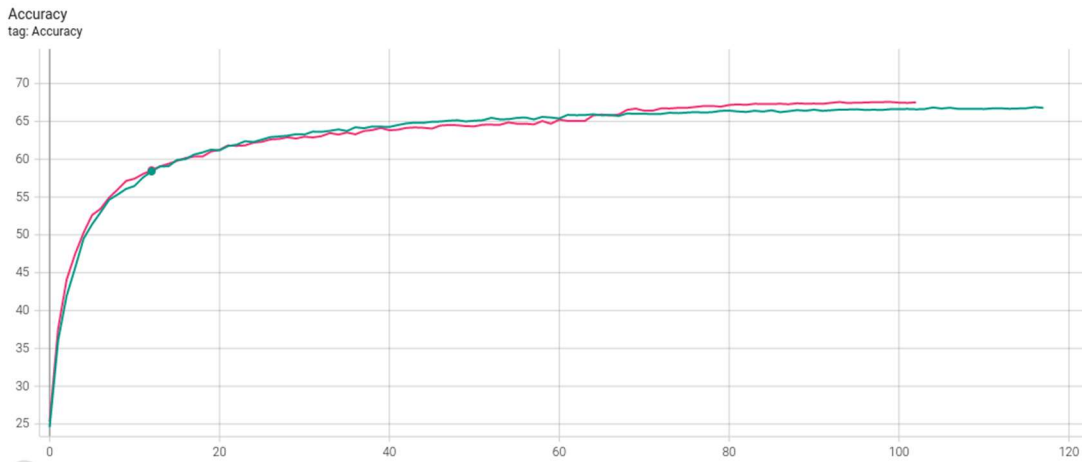


Figura 14 : MobileNetV2 (RGB 4:4:4 input) validation accuracy 67.57%

## 5 MobileNetV2 with YUV input

The original model of the MobileNet-v2 network provides a single input tensor formed by three channels (R, G, B) with a resolution of  $224 \times 224$  *pixels* . The equivalent image in the YUV 4: 2: 0 format has two different resolutions for luminance (Y) and chrominance (UV) due to undersampling of the chroma signal.

### 5.1 Modified architecture

Analyzing the architecture of the original model we notice a first Conv2D *layer* ( $3 \times 3$  *kernel*) with the parameter *stride* = 2 which reduces the resolution of the input tensor.

Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$

This feature of the model has allowed a modification of the pytorch code (available in the file *nnmodels / mobilenetv2yuv.py* ) to obtain a variant with a double input: one for the luminance plane and one for the two chrominance planes (which have the same resolution).



Given the resolution ratio due to subsampling, the first Conv2D *layer* has been split into two *Conv2D layers* with two different *striding* values . For the luminance data the original *stride* = 2 value was kept while for the chrominance data the *stride* = 1 value was used .

The two convolution operations are distinct, they operate on different data (with subsequent *batch\_normalization* and activation of the *ReLU6 type* ) and therefore generate two distinct result sets that must be combined for the next *layer* in the MobileNet model.

The *concatenation* of the results was used to maintain the same *depth* ( *depth*, number of channels) as the convolution operation of the original model. The original model uses 32 convolution filters. In the case of the YUV model, each of the convolutional layers (Y, UV) uses 16 filters, and their concatenation returns a size equal to the original one ( *depth* = 32 ).

These choices allow to obtain a MobileNetV2-YUV model very similar to the original one (only the data input stage changes).

It is also noted that the two operations are not equivalent: the original convolution calculates the result on all three input channels at the same time ( *dot-product*, *add* ) while in the modified model two distinct results subsequently concatenated are calculated.

**In this phase we wanted to verify whether such a modified network is able to obtain accuracy performances comparable to the objective reference one, together with a performance advantage in the final implementation phase on the *embedded device* .**

In the following figure, the architectural changes present in the code are graphically represented by viewing with the "Netron" tool ( <https://netron.app/> )

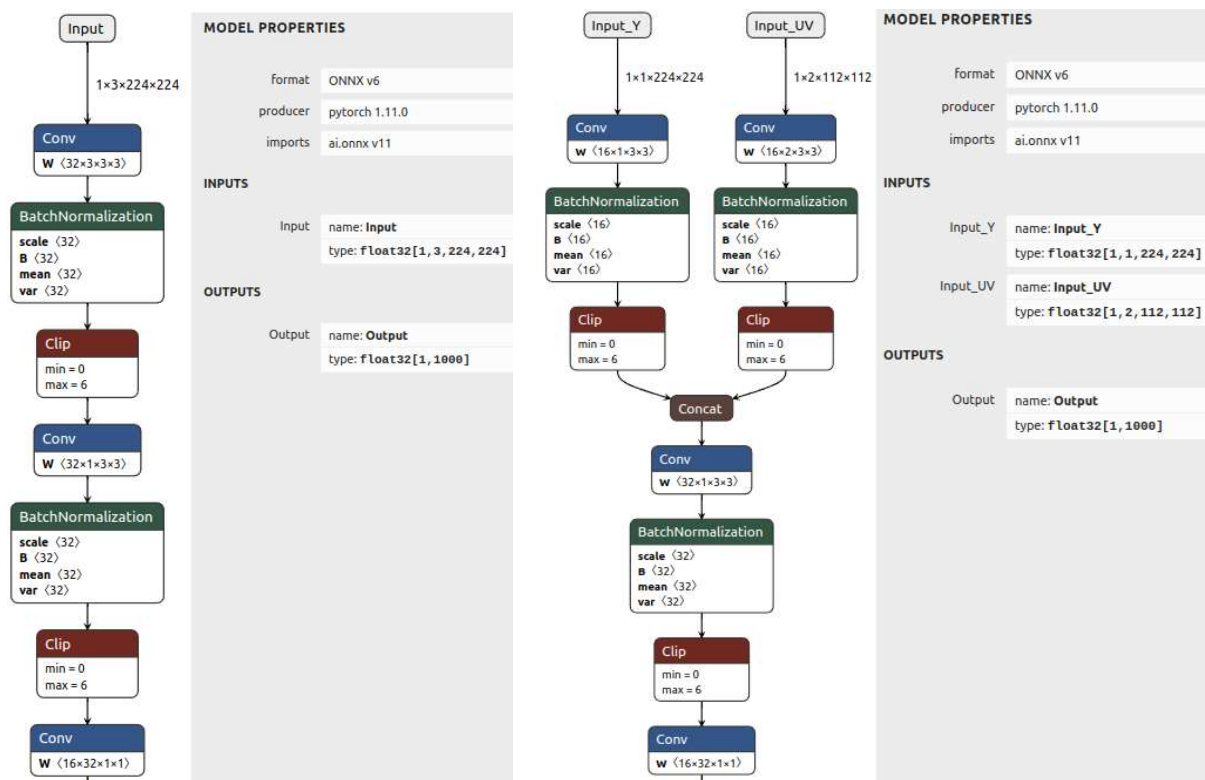



Figura 15 : Confronto MobileNetV2 "rgb 4:4:4: input" e "yuv 4:2:0 input"

## 5.2 Training

The training of the YUV model was carried out on the basis of the results and choices relating to the training of the model with RGB input. The same configuration has been deliberately maintained to verify the equivalence of the result. See the “ *training\_mobilenetv2-yuv.py* ” script. 

In the graph of fig. 16 the two *validation accuracy curves* can be superimposed during the entire duration of the training with a slight advantage for the RGB model (67.63 vs 67.24)

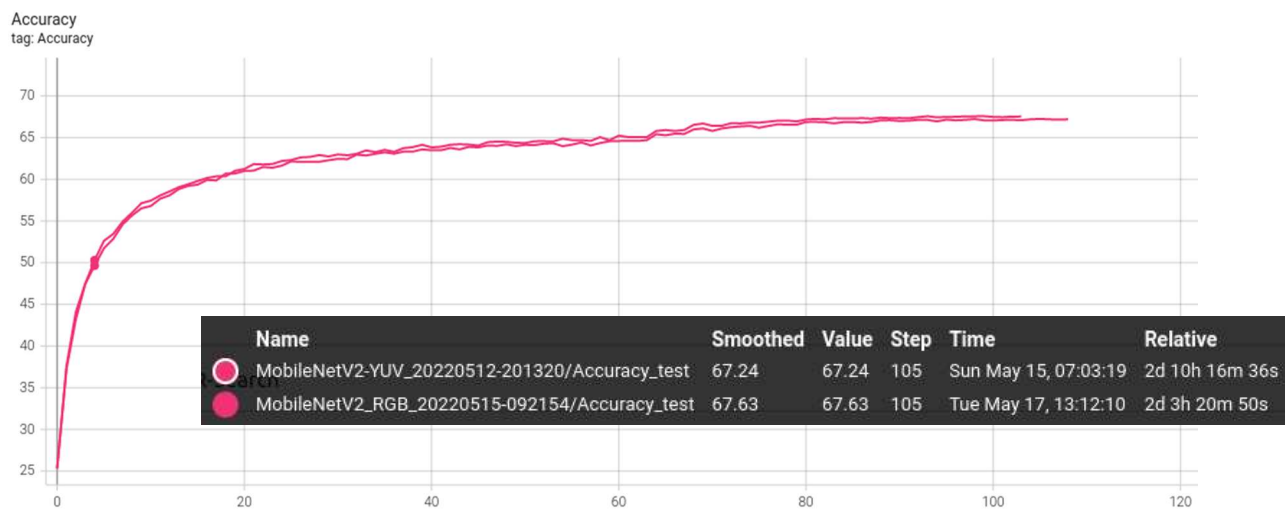
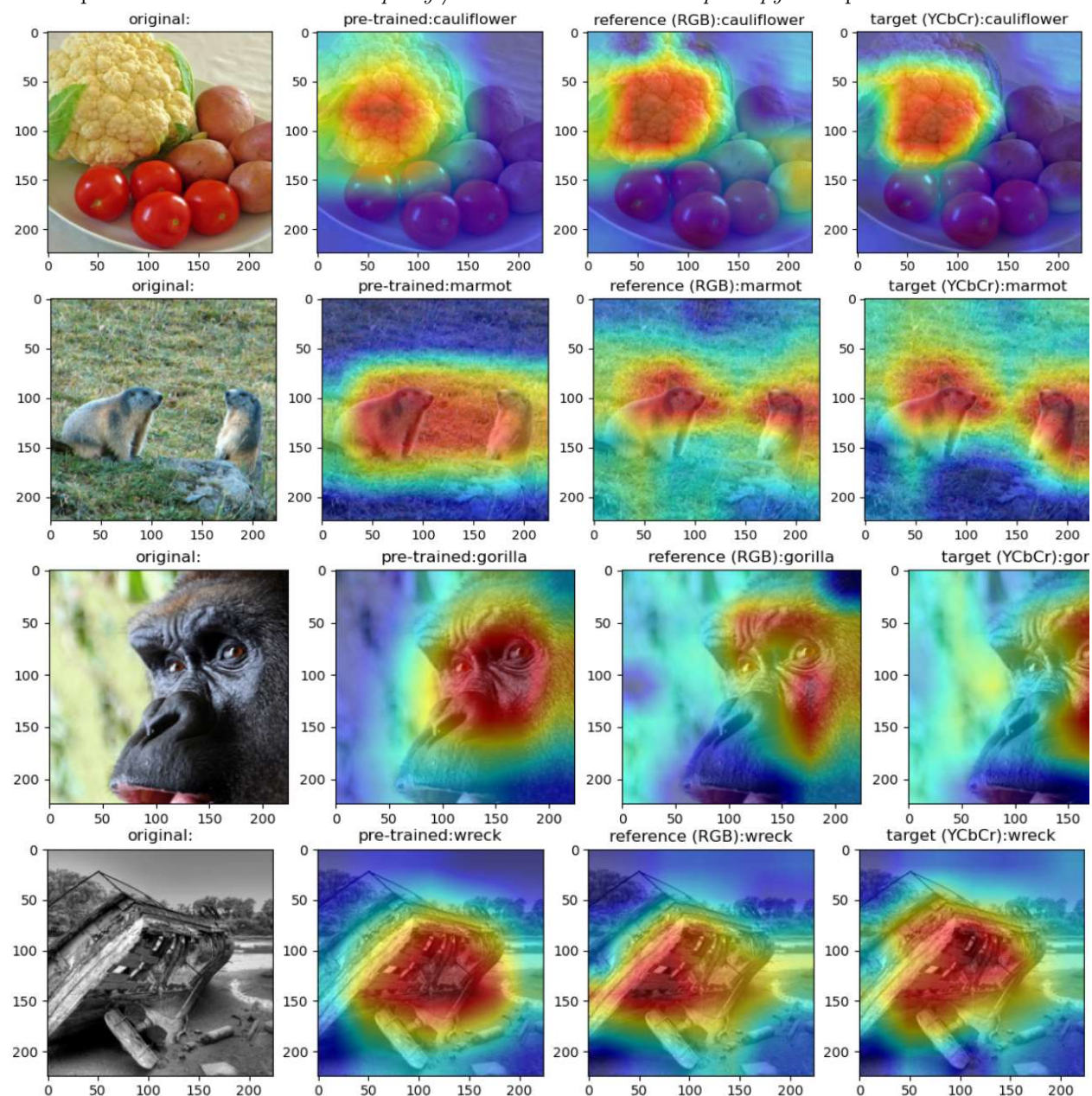


Figure 16: MobileNetV2 training, rgb input vs yuv input

## 6 Results comparison (GradCam)

In addition to the comparison in terms of accuracy, we wanted to verify the behavior from the *Interpretable Ai point of view* , a very important topic in the *machine learning field today* . In the case in question, since the network is convolutional, it was possible to use the GradCam library ( <https://arxiv.org/abs/1610.02391> ) to obtain a *visual explanation* ( *visual explanation* ) of the model given a specific classification. The result shows *consistency of the region of interest* ( *saliency map* ) between the RGB and YUV input network, confirming the functional correctness of the changes made.

Note: the GradCam library does not allow to use multiple input tensors it has been modified to accept a dual input Y-UV. See the code “ *3rdparty / GradCam* ”. See the “ *xplain.py* ” script.





## 7 Embedded application

To measure the performance on the final platform, a user application was written in C / C++ language in order to measure processing times as similar as possible to those of a real industrial system.

In this phase we want to determine a hardware and software configuration that shows an optimal level of performance for industrial solutions of image analysis with low latency on an *embedded system* (millisecond order of magnitude) .

### 7.1 PyTorch reference: *realtime performance*

A useful performance reference for MobileNetV2 on *embedded devices* is provided by the PyTorch framework in one of the examples optimized for RaspberryPi-4 hardware. At the time of writing this document, the code is available with the indication " *Real Time Inference on Raspberry Pi 4 (30 fps!)* " At: [https://pytorch.org/tutorials/intermediate/realtime\\_rpi.html](https://pytorch.org/tutorials/intermediate/realtime_rpi.html)

In the code the same MobileNet-V2 model is used to make a real-time classification of single objects placed in front of the room (configured for MIPI-CSI2 bus in RGB 4: 4: 4 format).

The source code configures the camera for an RGB image acquisition of a resolution equal to the size established by the neural network input.

In this case, the resizing is handled directly by the hardware through the *image signal processor (ISP)* .

The code used for image acquisition via OpenCV is the following:

```
import cv2
Postal Code = cv2 . VideoCapture ( 0 , cv2 .
CAP_V4L2 )
chap . set ( cv2 . CAP_PROP_FRAME_WIDTH , 224 )
chap . set ( cv2 . CAP_PROP_FRAME_HEIGHT , 224 )
chap . set ( cv2 . CAP_PROP_FPS , 36 )
```

followed by the instruction:

```
# read frame
ret , image = chap . read ()
```

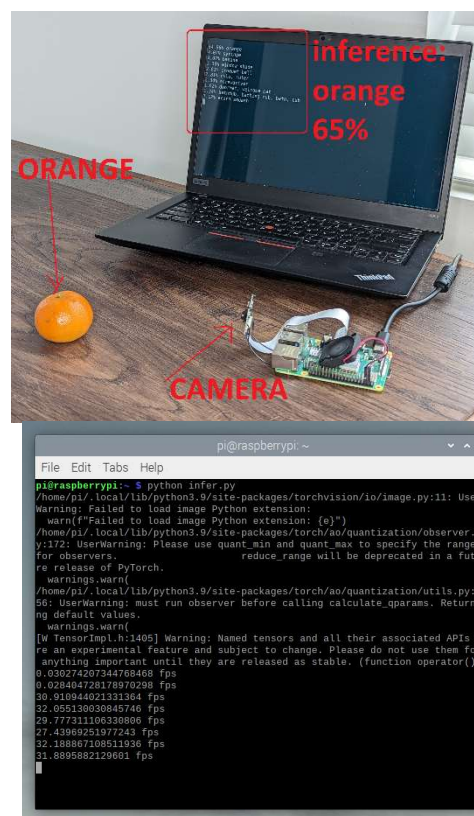


Figura 17 : inferenza su immagine OpenCV



Note: in this case it is possible to set the image acquisition resolution because the camera is directly connected to the device and there is no need for further *video streaming / recording* for video surveillance and / or monitoring and / or archiving functions.

It is important to note that the model used is not the one in *pre-towed floating-point format* but an *8bit quantized version optimized through the QNNPACK library* (<https://github.com/pytorch/QNNPACK>)

Despite the optimizations, through ISP and quantization, the maximum performance value obtained is about **33 fps** , for which the CPU has a utilization level of **98%** , **a value that would prevent further user functions** ( e.g. GUI *graphical user interface*)

The value obtained gives a direct measure of how much a network like MobileNetV2, specially developed by the authors for devices with limited computational resources in 2017-18, still represents a concrete limit for industrial devices based only on CPUs.

Note: The Pytorch sample code has been modified to remove the limitation of image acquisition using OpenCV by simulating a buffer of images resident in RAM memory. Also in this case the maximum performance does not exceed **34-36 fps** , demonstrating that the real performance limit lies in the architecture of the CPU itself, not suitable (or rather not optimal) for running models with convolutional neural networks.

Note: With such high inference times, optimizing the processing flow by eliminating color space conversion would not have a significant impact on system performance. A dedicated coprocessor (NPU) was therefore used which allows inference times of a few milliseconds (for the MobileNetV2 model).

Only by using a co-processor can you get sufficient performance for industrial applications and gain an advantage with a modified architecture for 4: 2: 0 YUV inputs.

## 7.2 Hailo8 : specific co-processor for neural networks

The performance data of the Pytorch reference in the previous point demonstrates how it is not currently possible to develop industrial applications with neural networks using CPU-based systems without considering relatively long latencies and inference times. For example, if it is required to *implement a decision* in a *few milliseconds* on the basis of the result of the inference, a processing time of about 30 milliseconds for each single image would not be tolerated. This is a very common case for automated inspection / quality control / monitoring and supervision applications.

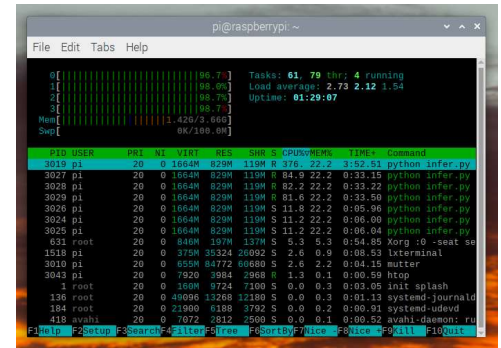
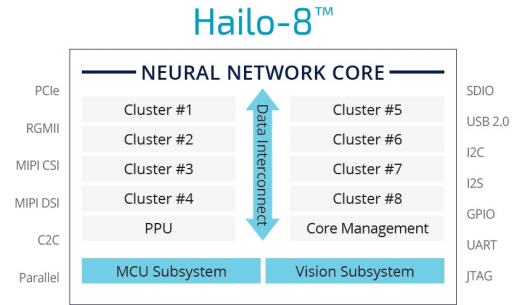


Figura 18 : RaspberryPi-4 utilizzo CPU 100% per il calcolo inferenza su MobileNetV2 (quantizzata INT8)



For this reason, a co-processor dedicated to the execution of the inference phase with neural networks produced by **Hailo.ai** and named **Hailo8** was used .

Hailo8 is a PCIe 3.0 bus device that interfaces with the CPU for inference only with the MobileNet model. The device has an internal architecture based on eight (8) computing units together with an *on-chip memory* of 32MBytes for storing input / output data and parameters of the neural network . **Peculiarity of Hailo8 is the high electrical efficiency providing 26 TOPS with a typical consumption of 2.5W (maximum of about 7W).**



The device has a python-based *open-source framework* for converting a Tensorflow model (or Pytorch using ONNX) into the binary format specific to the hardware architecture.

The figure shows the RaspberryPi *ComputModule -4* (CM4) board mounted on the upper side (large heat sink) and the Hailo8 co-processor mounted on the lower side (small heat sink)



Figura 19 : RaspberryPi CM4 con il co-processore Hailo8

In order to optimize the performance of an *embedded system* with one or more Hailo8 co-processors, the developer must carry out some model conversion phases, verifying that the final accuracy is as close as possible to that of the starting model.

### 7.3 Hailo8: from Pytorch to ONNX

Hailo8 does not support Pytorch models in “native” format. It is necessary to convert the model to be used in ONNX format ( <https://onnx.ai/> ) to be quantized with the software tools provided by the Hailo8 toolchain.

Pytorch allows the conversion to ONNX format with some simple instructions, see the script “ *hailo\_model\_onnx.py* ”. An important detail is the indication of two parameters that facilitate the conversion for Hailo8 architecture:

- **do\_constant\_folding = False**
- **training = torch.onnx.TrainingMode.PRESERVE**



The first parameter avoids some "blending" optimizations of constant values so that they are not embedded with convolution filters. Hailo8 needs the individual structural information of the network and the *tools* will perform optimizations, such as merging, a posteriori (after integer quantization, see next chapter)

The second parameter exports the model with all the additional information related to the *training phase* . This information supports the Hailo8 *tools* to determine the dynamic range of the values of the coefficients and activation of the neural network. The Hailo8 *tools* implement diversified optimization strategies (zeroing of near-zero coefficients, optimization by fusion of constant values, quantization with fine-tuning, etc.) and require all the information relating to the PyTorch model in the most detailed configuration (i.e. the training configuration )

At the end of these operations two further files are created starting from the original Pytorch file:

- a .onnx file created by PyTorch for the "onnx" format
- a .har file ( *hailo archive format* ) that contains the same model (not quantized) according to the Hailo8 specifications

For the conversion from .onnx model to .har model see the script “ *hailo\_model\_parse.py* ”.



## 7.4 Hailo8: quantization and compilation of the model

*floating-point* ONNX model is quantized to 8bit integers with the support of a sub-set of the *dataset* . The hailo8 tool performs a few calibration cycles to establish the correct (numerical) dynamics of the network parameters and minimize the accuracy losses due to computation with integer values. See the script " *hailo\_model\_quantize.py* "



The final step consists in *binary compilation* of the model thus obtained so that it can be executed by the hardware co-processor. See the “ *hailo\_model\_compile.py* ” script.



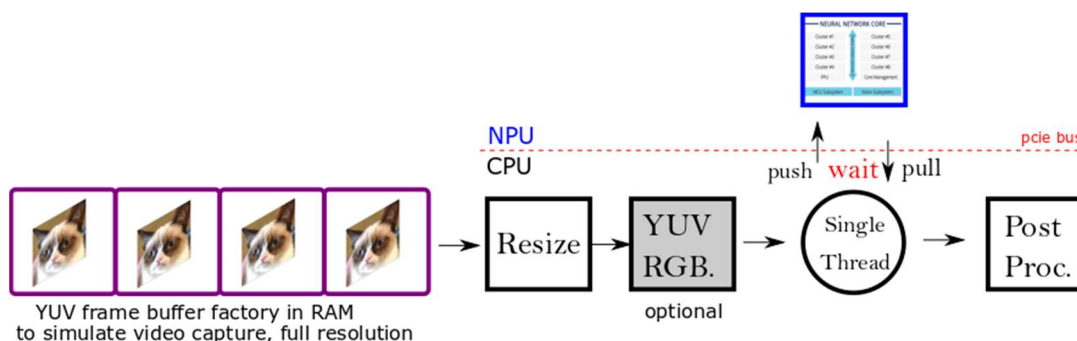
The accuracy of the quantized model was measured on the ImageNet *validation set* using the hardware device directly, see the script “ *hailo\_model\_evaluation.py* ”.

## 7.5 Final C / C ++ application

In order to measure the real performance of the device, a specific application has been written that allows you to compare the times of the various processing stages ( *resize*, *color conversion*, *inference* ). The source code is available in the “ *apps / testfps* ” folder.

The code simulates the image acquisition part with an in-memory buffer in which images in YUV 4: 2: 0 format are pre-loaded. This choice is mandatory to free the absolute performance measurement from any limitations of acquisition hardware or video streaming over network protocols.

## 7.6 Architecture



The application architecture has a processing chain consisting of:

- An interpolation stage ( *resize* ) to adapt the resolution of the input images to that required by the neural network used.
- One stage of color conversion (and oversampling) from 4: 2: 0 YUV format to 4: 4: 4 RGB format.  
**This stage is not used if the network being used allows a 4: 2: 0 YUV input**
- The inference stage, in this case performed by the Hailo8 co-processor on the pcie bus
- The final stage of post-processing based on the tensor leaving the neural network.

Note: The *resize stage* is the first in the processing chain. The assumption of the image source format as YUV 4: 2: 0 allows a reduction of the computational load. Choosing to perform color conversion after image resizing is optimal to minimize CPU load.

## 7.7 Latency and Throughput

In this configuration each image is sent to the NPU for the execution of the MobileNetV2 model. **This operation is "blocking": the CPU has to wait for the result of the inference to proceed with the post processing phase.**

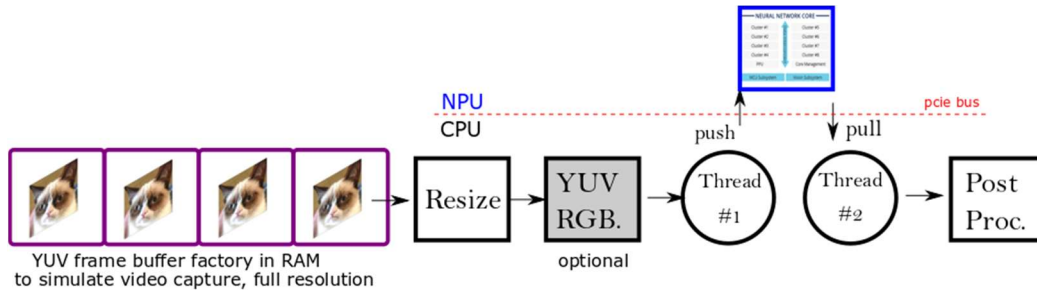
This structure is representative of an industrial device that must implement a decision with each image analyzed (for example, eliminate a defective product on the production line) in a few milliseconds.

The time elapsed from the moment the image enters the processing chain to the moment the result is ready at the output is called *Latency* (latency-batch-size = 1 to indicate that only one data is supplied at the input).

In the event that the application provides for the analysis of multiple input streams (for example multiple H.264 / AVC *video streams* decoded in YUV format) the Latency value alone is no longer sufficient and the maximum value of images processed every second (overall on all input data)

In this case, thanks to the internal memory of Hailo8, images can be sent to the device in sequence using a *multi-thread architecture* : a *thread* for writing the images and one for reading the results.

The writer *thread* sends images to the co-processor in rapid sequence while a further (non-synchronous) *thread* takes care of receiving the results from the NPU, eliminating the blocking constraint. Taking a single image, the processing latency is identical to the previous case, but in the *multi-threaded case* it is possible to process multiple incoming data streams made serial by the queue available in the “ *buffer factory* ” in



the figure.

## 7.8 HAL software interface for Hailo8

The Hailo8 hardware device needs two phases to be used. The first configuration phase reads the .hef binary file (via CPU) and transfers it to the physical device. Communication interfaces for data and results input / output are also activated. The second phase allows requests to send data and to read the results through the previously opened channels. The code available in the “apps / testfps” folder. A *hardware abstraction* interface is highlighted *layer (HAL)* ”which simplifies the use of the device.



In the files “ *hailo.cpp* and *hailo.hpp* ” we expose a class with only two methods: *init ()* and *modelEval ()*, in addition to *getFrameCnt ()*

```
class HailoCtrl
{
public:
    HailoCtrl ();
    HailoCtrl (uint8_t verbose);
    ~ HailoCtrl ();

    int8_t init (uint8_t model_type, uint8_t verbose);
    int8_t getFrameCnt (uint64_t * frames_cnt);
    int8_t modelEval (uint8_t * in [], size_t in_size [],
                     uint8_t * out [], size_t out_size [],
                     size_t frame_count, bool debug);
}
```

*init ()* method prepares the co-processor to execute the model via the binary file (.hef). This is done only once in the initial stages of starting the device.

The *modelEval ()* method performs all data writing and reading operations towards the co-processor, orchestrating any calls from asynchronous *threads* .

These methods implement a *functional abstraction level*: the code mainly deals with the transfer of the images present in the input queue ( *buffer Queue* ).

Both rely on further lower level methods, exposed by two files ( *hailo\_device.hpp* and *hailo\_device.cpp* ). These implement the *peripheral abstraction layer* : using the specific API of the Hailo *framework* (see





*libhailort library* ) they take care of the execution of the single instructions necessary for the initialization of the hardware device and of the data transfer.

## 7.9 Results

Before compiling the application on the device it is necessary to copy the two models in binary .hef format (both rgb and yuv) in the “ *apps / testfps* ” folder and then use *cmake* to compile the application. Two scripts are released to make the build procedure automatic (see *buildme\_x86.sh* and *buildme\_rpi.sh* )



```
Usage: testfps [options]
Options:
--help Display this information.
--input Input image file.
--type = {rgb | yuv} Model type to run for NN testing {default = RGB}
--nthreads number of cpu-threads to spawn for DSP {default = 1}
--show display source and converted / resized images
--debug run the pipeline with 1s delay for debug purposes
```

The command syntax allows you to select the type of model and set the number of *threads* to use. The result provides the total latency time (in micro seconds) and the individual times for `resize / color_conversion / inference` (Hailo8 transfer). Here are some results for RGB and YUV cases.

```
pi @ rpihailo-01: ~ / Fil_PRJ / unimore / apps / testfps $ ./build/testfps -i ../data/mcguiver.1280x720.jpg -n  
l --type rgb
```

---

```
/| | / / \_ / / (\_) / \_ / / | / \_ / \_ / \_ / \_ / \_ / \_ / \_ / \_ / \_ /  
// | | / \_ / / \_ / \_ / \_ / \_ / \_ / \_ / \_ / \_ / \_ / \_ / \_ / \_ /  
/_ / \_ / \_ / \_ / \_ / \_ / \_ / \_ / \_ / \_ / \_ / \_ / \_ / \_ / \_ /  
--- ver.0.2.0.dbg ---
```

---

```
[INFO] [HailoCtrl] create HailoCtrl controller.  
[INFO] init hailo device  
[STATS] fcnt = 309 fps = 309 (38.62), TIMES (usec): tot = 3225.90 resize = 688.57 cconv = 476.34 infer =  
2052.64  
[STATS] fcnt = 624 fps = 315 (78.00), TIMES (usec): tot = 3204.62 resize = 686.80 cconv = 472.72 infer =  
2033.16  
[STATS] fcnt = 938 fps = 314 (117.25), TIMES (usec): tot = 3197.45 resize = 687.04 cconv = 472.05 infer =  
2027.13  
[STATS] fcnt = 1252 fps = 314 (156.50), TIMES (usec): tot = 3193.99 resize = 686.60 cconv = 472.00 infer =  
2023.94  
[STATS] fcnt = 1566 fps = 314 (195.75), TIMES (usec): tot = 3191.91 resize = 686.44 cconv = 472.07 infer =  
2022.17  
[STATS] fcnt = 1880 fps = 314 (235.00), TIMES (usec): tot = 3190.57 resize = 686.43 cconv = 471.90 infer =  
2020.91  
[STATS] fcnt = 2194 fps = 314 (274.25), TIMES (usec): tot = 3189.49 resize = 686.32 cconv = 471.70 infer =  
2020.03  
[STATS] fcnt = 2508 fps = 314 (313.50), TIMES (usec): tot = 3188.66 resize = 686.27 cconv = 471.61 infer =  
2019.29  
[STATS] fcnt = 2822 fps = 314 (314.12), TIMES (usec): tot = 3183.42 resize = 685.87 cconv = 470.96 infer =  
2014.69  
[STATS] fcnt = 3137 fps = 315 (314.12), TIMES (usec): tot = 3183.37 resize = 685.77 cconv = 471.01 infer =  
2014.65  
[STATS] fcnt = 3451 fps = 314 (314.12), TIMES (usec): tot = 3183.49 resize = 685.56 cconv = 471.04 infer =  
2014.70
```

```
pi @ rpihailo-01: ~ / Fil_PRJ / unimore / apps / testfps $ ./build/testfps -i ../data/mcguiver.1280x720.jpg --n  
3 --type rgb
```

ver.0.2.0.dbg

```
[INFO] [HailoCtrl] create HailoCtrl controller.  
[INFO] init hailo device  
[STATS] fcnt = 1134 fps = 1134 (141.75), TIMES (usec): tot = 2640.89 resize = 1256.44 cconv = 849.75  
[STATS] fcnt = 2292 fps = 1158 (286.50), TIMES (usec): tot = 2614.88 resize = 1250.90 cconv = 839.86  
[STATS] fcnt = 3450 fps = 1158 (431.25), TIMES (usec): tot = 2606.23 resize = 1247.31 cconv = 836.04  
[STATS] fcnt = 4608 fps = 1158 (576.00), TIMES (usec): tot = 2602.21 resize = 1242.38 cconv = 834.89  
[STATS] fcnt = 5766 fps = 1158 (720.75), TIMES (usec): tot = 2599.84 resize = 1239.13 cconv = 834.09  
[STATS] fcnt = 6924 fps = 1158 (865.50), TIMES (usec): tot = 2597.77 resize = 1238.45 cconv = 833.47  
[STATS] fcnt = 8081 fps = 1157 (1010.12), TIMES (usec): tot = 2597.31 resize = 1238.18 cconv = 833.18  
[STATS] fcnt = 9240 fps = 1159 (1155.00), TIMES (usec): tot = 2596.20 resize = 1238.74 cconv = 832.47  
[STATS] fcnt = 10398 fps = 1158 (1158.00), TIMES (usec): tot = 2589.51 resize = 1237.55 cconv = 829.94  
[STATS] fcnt = 11555 fps = 1157 (1157.88), TIMES (usec): tot = 2590.06 resize = 1236.24 cconv = 830.25  
[STATS] fcnt = 12710 fps = 1155 (1157.50), TIMES (usec): tot = 2590.59 resize = 1235.43 cconv = 830.41
```

```
pi@ rpihailo-01: ~ / Fil_PRJ / unimore / apps / testfps $ ./build/testfps -i ../data/mcguyver.1280x720.jpg -n 1 --type yuv
```

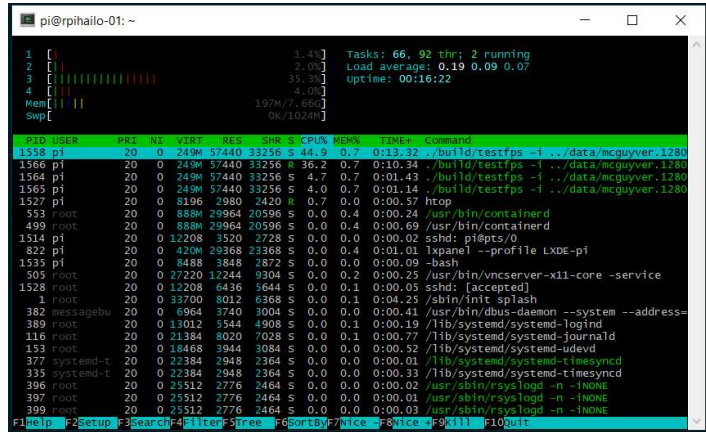
```
[INFO] [HailoCtrl] create HailoCtrl controller.
[INFO] init hailo device
[STATS] fcnt = 348 fps = 348 (43.50), TIMES (usec): tot = 2871.47 resize = 670.46 cconv = 29.55 infer = 2162.45
[STATS] fcnt = 699 fps = 351 (87.38), TIMES (usec): tot = 2860.43 resize = 662.61 cconv = 28.60 infer = 2158.36
[STATS] fcnt = 1055 fps = 356 (131.88), TIMES (usec): tot = 2843.22 resize = 658.28 cconv = 28.07 infer = 2146.10
[STATS] fcnt = 1410 fps = 355 (176.25), TIMES (usec): tot = 2834.94 resize = 656.67 cconv = 27.84 infer = 2140.16
[STATS] fcnt = 1766 fps = 356 (220.75), TIMES (usec): tot = 2829.81 resize = 655.19 cconv = 27.68 infer = 2136.56
[STATS] fcnt = 2122 fps = 356 (265.25), TIMES (usec): tot = 2826.27 resize = 654.28 cconv = 27.58 infer = 2133.96
[STATS] fcnt = 2478 fps = 356 (309.75), TIMES (usec): tot = 2823.92 resize = 653.61 cconv = 27.52 infer = 2132.30
[STATS] fcnt = 2834 fps = 356 (354.25), TIMES (usec): tot = 2822.29 resize = 653.03 cconv = 27.55 infer = 2131.19
[STATS] fcnt = 3190 fps = 356 (355.25), TIMES (usec): tot = 2814.70 resize = 650.34 cconv = 27.37 infer = 2126.23
[STATS] fcnt = 3545 fps = 355 (355.75), TIMES (usec): tot = 2810.47 resize = 649.58 cconv = 27.42 infer = 2122.95
[STATS] fcnt = 3900 fps = 355 (355.62), TIMES (usec): tot = 2811.38 resize = 649.56 cconv = 27.48 infer = 2123.79
```

```
[INFO] [HailoCtrl] create HailoCtrl controller.
[INFO] init hailo device
[STATS] fcnt = 1682 fps = 1682 (210.25), TIMES (usec): tot = 1783.73 resize = 1298.59 cconv = 91.52
[STATS] fcnt = 3444 fps = 1762 (430.50), TIMES (usec): tot = 1742.22 resize = 1280.92 cconv = 91.72
[STATS] fcnt = 5249 fps = 1805 (656.12), TIMES (usec): tot = 1714.00 resize = 1271.33 cconv = 93.13
[STATS] fcnt = 6939 fps = 1690 (867.38), TIMES (usec): tot = 1728.81 resize = 1273.78 cconv = 91.05
[STATS] fcnt = 8682 fps = 1743 (1085.25), TIMES (usec): tot = 1727.10 resize = 1272.70 cconv = 91.04
[STATS] fcnt = 10465 fps = 1783 (1308.12), TIMES (usec): tot = 1719.38 resize = 1268.82 cconv = 90.86
[STATS] fcnt = 12253 fps = 1788 (1531.62), TIMES (usec): tot = 1712.99 resize = 1268.77 cconv = 90.98
[STATS] fcnt = 14023 fps = 1770 (1752.88), TIMES (usec): tot = 1710.66 resize = 1270.65 cconv = 91.10
[STATS] fcnt = 15778 fps = 1755 (1762.00), TIMES (usec): tot = 1701.74 resize = 1269.84 cconv = 90.76
[STATS] fcnt = 17539 fps = 1761 (1761.88), TIMES (usec): tot = 1701.66 resize = 1271.82 cconv = 90.19
[STATS] fcnt = 19289 fps = 1750 (1755.00), TIMES (usec): tot = 1708.52 resize = 1274.99 cconv = 88.89
[STATS] fcnt = 21050 fps = 1761 (1763.88), TIMES (usec): tot = 1699.83 resize = 1275.31 cconv = 89.49
[STATS] fcnt = 22845 fps = 1795 (1770.38), TIMES (usec): tot = 1693.53 resize = 1275.12 cconv = 89.17
[STATS] fcnt = 24633 fps = 1788 (1771.00), TIMES (usec): tot = 1692.82 resize = 1277.54 cconv = 89.11
[STATS] fcnt = 26402 fps = 1769 (1768.62), TIMES (usec): tot = 1695.28 resize = 1276.62 cconv = 88.71
[STATS] fcnt = 28132 fps = 1730 (1763.62), TIMES (usec): tot = 1700.03 resize = 1268.45 cconv = 87.68
[STATS] fcnt = 29859 fps = 1727 (1760.12), TIMES (usec): tot = 1703.35 resize = 1264.00 cconv = 87.15
[STATS] fcnt = 31661 fps = 1802 (1765.25), TIMES (usec): tot = 1698.56 resize = 1260.38 cconv = 87.80
[STATS] fcnt = 33469 fps = 1808 (1772.50), TIMES (usec): tot = 1691.72 resize = 1256.32 cconv = 88.85
[STATS] fcnt = 35280 fps = 1811 (1778.75), TIMES (usec): tot = 1685.65 resize = 1251.42 cconv = 89.47
[STATS] fcnt = 37091 fps = 1811 (1780.75), TIMES (usec): tot = 1683.90 resize = 1248.14 cconv = 90.34
[STATS] fcnt = 38892 fps = 1801 (1782.38), TIMES (usec): tot = 1682.65 resize = 1245.26 cconv = 90.51
```

The RaspberryPi CM4 / Hailo8 combination shown guarantees performance ten times higher than that obtained by the optimized solution of the Pytorch example.

In addition to the advantage over the only inference with the MobileNetV2 network, there is an increase in system performance thanks to the low computational load on the main CPU (Cortex A72, quad-cores).

This allows the implementation of any human-machine interfaces and / or control of external devices thanks to the availability of computational resources on the CPU.



## 8 Conclusions

This document analyzes and confirms the functional equivalence of a MobileNetV2 neural network with two types of input data: images in RGB 4: 4: 4 format and in YUV 4: 2: 0 format.

The proposed system for *embedded implementation* combines a Raspberry Pi-CM4 processor with a Hailo8 co-processor and provides an order of magnitude higher performance (10x) than the realtime Pytorch reference (max 30fps).

The use of the MobileNetV2-YUV 4: 2: 0 network offers a performance increase > 10% compared to the solution with RGB 4: 4: 4 input.

Accuracy (TOP-1) for ImageNet ILSVRC-2012 “*validation set*”

<i>floating-point (32bit)</i>		<i>integer (8bit)</i>	
MobileNetV2-RGB 4: 4: 4	MobileNetV2-YUV 4: 2: 0	MobileNetV2-RGB 4: 4: 4	MobileNetV2-YUV 4: 2: 0
67.63	67.24	66.56	66.60

Performance on the *embedded device* in terms of *min latency* and *max throughput (threads = 3)*

Image WxH (pixels)	RaspberryPi CM4 + Hailo8						Performance Gain (%)		
	MobileNetV2-RGB 4: 4: 4			MobileNetV2-YUV 4: 2: 0			Latency	Single Thread	Multi Thread
	Latency (ms)	Single Thread (fps)	Multi Thread (fps)	Latency (ms)	Single Thread (fps)	Multi Thread (fps)			
640x360	2.75	364	1415	2.37	422	2370	13.8	13.7	67.5
1280x720	3.20	314	1157	2.80	355	1780	12.5	13.0	53.8
1920x1080	3.46	288	980	3.10	320	1390	10.4	11.1	41.8
3840x2160	4.85	205	610	4.00	245	840	17.5	19.5	37.7

In the implementation shown, the OpenCV library is used both for the *resize part* and for the color conversion part. To further validate the results shown, measurements were also carried out for images of intermediate size to the standard ones described above, up to the maximum resolution 7680x4320 (8K, 16:

9). The OpenCV algorithms are optimized for standard resolutions (with particular attention to reading / writing data in RAM memory). Using images with intermediate resolutions, local variations on the performance gain of the YUV solution compared to the RGB solution are noted.

Image		Latency (ms)		Single Thread (fps)		Multi Thread (fps)	
W	H	rgb	yuv	rgb	yuv	rgb	yuv
640	360	2.75	2.37	364	422	1415	2370
1280	720	3.20	2.80	314	355	1157	1780
1920	1080	3.46	3.10	288	320	990	1390
3072	1728	4.56	3.80	218	263	688	980
3840	2160	4.85	4.00	205	245	610	840
5376	3024	5.70	4.80	175	207	498	634
6448	3627	5.25	4.70	190	214	476	566
7680	4320	6.55	5.75	153	177	402	470

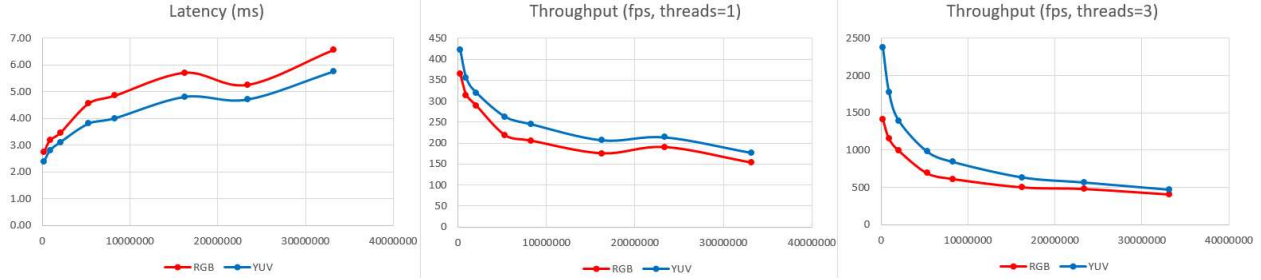


Figure 20: Performance as a function of image size (area)

## 9 Future developments

The results obtained highlighted three thematic areas that suggest further study.

### 9.1 HPC cluster training

The MobileNet reference performance is obtained on multiple computing nodes in parallel (HPC: high performance computer). Thanks to a recent introductory course on "Artificial Intelligence for industry"<sup>15</sup> organized by CINECA (<https://www.cineca.it/>) it was possible to access the HPC MARCONI-100 cluster<sup>16</sup>.

For parallel training on multiple GPUs the original script has been modified to use the **Horovod framework** (<https://github.com/horovod/horovod>). The code is made available in the "hrvd\_train\_mobilenetv2.sh" file.



The script performs distributed training in "data parallel" mode. The MobileNetV2 model is sent in its entirety to all GPUs used and the *dataset* is divided into several parts to speed up the calculation of each *training epoch*. The *framework* takes care of mediating (synchronizing the results of all GPUs) the information necessary for the calculation of the neural network coefficients. The training process with Horovod can be assimilated to several phases of distribution (*scatter*) of hyper-parameters and gradients (to synchronize all calculation nodes) and collection (*gather*) of intermediate results (averaged between them for the final result).

During the tests with Marconi100 a technical problem arose that prevented the continuation of the work due to the limited number of hours / GPU granted by CINECA during the learning program.

<sup>15</sup><https://eventi.cineca.it/en/hpc/intelligenza-artificiale-lindustria/modena-20220321>

<sup>16</sup><https://www.hpc.cineca.it/hardware/marconi100>

Pytorch exposes a specific class ( *dataset loader* ) for reading images from the ImageNet *dataset* (<https://pytorch.org/vision/stable/generated/torchvision.datasets.ImageNet.html>). This API is used in all scripts released with this document. The class is implemented on the basis of python PIL (python image library) and performs a single read access for each image of the dataset (jpeg images, a few hundred bytes)

During the test, the slowness of the M100 system to complete a single epoch was highlighted despite eight (x8) GPUs being used in parallel. **While the single GPU computer took thirty minutes (30min), the cluster of eight GPUs took well two hours (2h).**

Working with CINECA technical support, the problem due to the configuration of the M100 file-system was identified. The file-system is configured with a *block-size* of 16 Mbytes while normally a workstation (not HPC) uses a *block-size* of 4kBytes. A smaller *block-size* allows the transfer of small files (such as the images of ImageNet ILSVRC2012) in a shorter time.

A hypothesized solution consists in modifying the official Pytorch code for the use of the structured dataset on a format not based on single files for each image. Probable alternatives Hierarchical Data Format version 5 (HDF5 , <https://www.hdfgroup.org/solutions/hdf5/>), or Apache Parquet ( <https://parquet.apache.org/>).

**Due to the exhaustion of the “ *testing* ” hours released by CINECA this phase of the study has not been completed.** For reference, the confirmation of the problem by CINECA is listed.

====

From: LAURA MORSELLI [mailto:L.MORSELLI@cineca.it]  
Sent: Tuesday 10 May 2022 10:47  
To: Filippini, Gianluca (EBV) <Gianluca.Filippini@ebv.com>  
Cc: ERIC PASCOLO <e.pascolo@cineca.it>  
Subject: [External] Re: EUROCC - Ai @ edge test project approval

Good morning Gianluca,

the bottleneck is due to the block size of the M100 filesystem, which is about 16Mb, while the trained images are 128K or 256K. On your laptop maybe you have a block size of 4K, or cmq much smaller than 16Mb (you can check with the command `stat -fc% s.`)

There is an area of M100 for AI that has a block size of 4M, if I have time during the day I try to see if I get the expected speed up, but I still expect lower performance from your laptop.

====

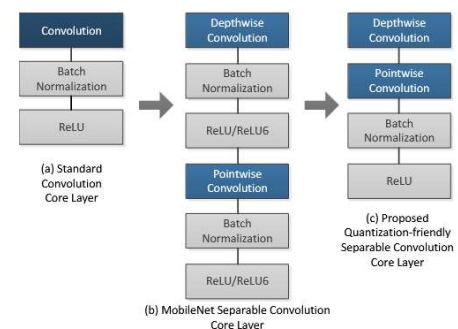
## 9.2 Depthwise Separable Convolution with INT8 arithmetic

One of the most critical aspects for the final implementation is the quantization (INT8 arithmetic) of the neural network model. MobileNet introduces the " *Depthwise Separable Convolution block* " to increase computing performance but refers (in the results shown) to a *floating-point arithmetic* .

A Qualcomm document following the publication of MobileNet attempts to mitigate the loss of accuracy in the transition from the *floating-point* model to the *int8 model*. See:

### A Quantization-Friendly Separable Convolution for MobileNets

Tao Sheng, Chen Feng, Shaojie Zhuo, Xiaopeng Zhang, Liang Shen, Mickey Aleksic



<https://arxiv.org/abs/1803.08607>

The document exposes the problem of “ *accuracy loss* ” very evident in models like MobileNetV1 with particular attention to the implementation of the *Depthwise Separable Convolution block* . The authors propose an alternative structure that removes the intermediate normalization-activation (BN / ReLu6) combination and replaces the activation function with the simpler ReLu.

The tools of Hailo8 allow to obtain an excellent correspondence between the *floating-point* model and the quantized *integer model* , minimizing the loss of accuracy (about 1%).

Despite this result, a possible future development consists in the application of what is stated in the document cited for the verification of the performances on the *embedded system* described.

### 9.3 SSD + MobileNetV2: object detection performance

MobileNetV2 is widely used in the industrial field also for object recognition applications.

In particular it is used as part of the SSD “Single Shot Multibox Detector” model originally written on the basis of VGGNet. In the original SSD document ( <https://arxiv.org/abs/1512.02325v5> ) VGGNet is used with the “ *feature extractor* ” functionality. The numerical *features* extracted are used to calculate the location and type (class) of the objects within the given image (fig. 21)

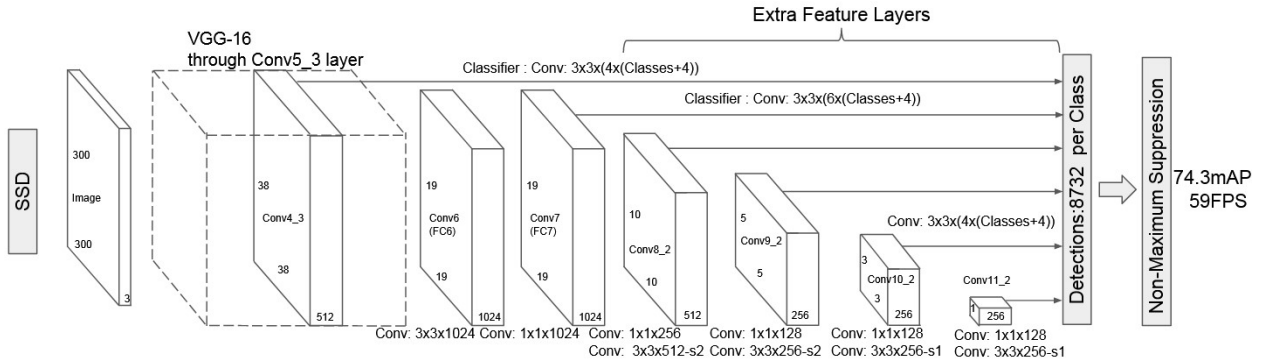


Figure 21: SSD architecture with VGGNet as a feature extractor


To reduce the computational complexity of the SSD algorithm, MobileNetV2 is commonly used as a replacement for VGGNet to reduce the computational complexity of the overall algorithm.

*object detection* algorithm if the *feature extraction* part is carried out with MobileNetV2-YUV.




## 9.4 Appendix


This section shows the output of the Hailo8 analysis tool for the YUV network. The first part of the report gives indications on the absolute maximum performances, on the latency, on the power consumption net of the I / O pipeline (and pre / post processing).

  
Empowering Intelligence


[Summary](#)  
[Device Utilization](#)  
[Model Description](#)  
[Layer by Layer Analysis](#)  
[Model Graph](#)  
[NN Core Execution Simulation](#)

  
Empowering Intelligence

[Summary](#)  
[Device Utilization](#)  
[Model Description](#)  
[Layer by Layer Analysis](#)  
[Model Graph](#)  
[NN Core Execution Simulation](#)

  
Empowering Intelligence

[Summary](#)  
[Device Utilization](#)  
[Model Description](#)  
[Layer by Layer Analysis](#)  
[Model Graph](#)  
[NN Core Execution Simulation](#)

  
Empowering Intelligence

### Hailo Dataflow Compiler — Profiler Report

Auto-generated version 3.16.0 in Mon, 16 May 2022 20:32:18 +0200

#### Summary

##### Model Details

Model Name	mobilenetv2_yuv_105
Input Tensors Shapes	224x224x1, 112x112x2
Output Tensors Shapes	1000
Operations per Input Tensor (OPs)	0.59 GOPs
Operations per Input Tensor (MACs)	0.30 GMACs
Model Parameters	3.49 M
Number of Contexts	1

##### Profiler Input Settings

Target Device	Hailo-8
Number of Devices	1
Profiler Mode	Pre Placement
Optimization Goal	Reach 2400 FPS

##### Performance Summary

Throughput	2407.16 FPS
Latency	1.56 ms
Total NN Core Power Consumption	1.65 W
Operations per Second (OP/s)	1417.71 GOP/s
Operations per Second (MAC/s)	724.88 GMAC/s

	NET	GROSS
Input Interface Throughput	180.63 Megabytes/sec	180.63 Megabytes/sec
Output Interface Throughput	2.40 Megabytes/sec	2.46 Megabytes/sec

#### Device Utilization

##### Total Utilization

Compute Usage	39.45%	<div></div>
Memory Usage	30.86%	<div></div>
Control Usage	78.13%	<div></div>

NOTE: 100% = 1 x HAILO-8™ DEVICE WITH A SINGLE CONTEXT