POLITECNICO DI TORINO

MASTER's Degree in COMPUTER ENGINEERING

MASTER's Degree Thesis

Optimizing YOLO Inference for Hardware Constraints Through Quantization Techniques

Supervisors

Prof. Luciano LAVAGNO

Phd. Teodoro URSO

Candidate

Niccolò CACIOLI

JULY 2024

Thesis Title

Thesis Author

Abstract

This thesis investigates the application of YOLO (You Only Look Once) models for object detection tasks, with a particular focus on the quantization of such models to enable efficient deployment on edge devices and resource-constrained hardware platforms. Model quantization plays a critical role in reducing memory footprint and computational cost while aiming to preserve the accuracy and robustness of the original floating-point networks.

The work focuses on integrating a complete training and evaluation pipeline, including data pre-processing compliant with widely adopted standards (e.g. YOLO) and the integration of automated tools for ground truth visualization and validation. Various training strategies were explored to enhance model performance, including hyperparameter tuning, architectural modifications, and data augmentation techniques.

A central contribution of the work is the design of a modular quantization workflow, leveraging tools compatible with ONNX and tailored for deployment with hardware-accelerated inference platforms. The methodology includes model export, transformation, optimization, and performance validation of the quantized networks.

Experimental results, obtained from both standard benchmarks and domain-specific datasets, demonstrate that the proposed approach achieves a favorable trade-off between model compactness and detection accuracy. These findings support the feasibility of adopting quantized YOLO architectures in real-world, real-time applications across diverse environments.

The workflow proposed, along with the solid results obtained by this work, offers a valid stepping stone for future improvements and optimizations.

ACKNOWLEDGMENTS

Alla mia famiglia.

Table of Contents

List of Figures

List of Tables

Acronyms

AI	Artificial Intelligence.
ML	Machine Learning.
DL	Deep Learning.
NN	Neural Network.
CNN	Convolutional Neural Network.
RNN	Recurrent Neural Network.
NLP	Natural Language Processing.
LSTM	Long Short Term Memory.
ReLU	Rectified Linear Unit.
CV	Computer Vision.
IoU	Intersection over Union.
CIoU	Complete Intersection over Union.
mAP	Mean Average Precision.
BCE	Binary Cross Entropy.
INT8	8-bits integer data.
FP32	32-bits floating point data.
QAT	Quantization Aware Training.

PTQ Post Training Quantization.

LR Learning rate.

VAI Vitis AI.

Chapter 1

Introduction

1.1 Machine Learning

In the age of the Fourth Industrial Revolution (Industry 4.0), the digital world exists with very diverse sources of data such as Internet of Things (IoT) data, cybersecurity logs, mobile usage data, business intelligence, social media streams, and healthcare records. Furthermore, the efficient investigation to design intelligent and automated programs out of this complex and diverse data requires a solid AI foundation, and in particular, a focus on machine learning (ML). ML is a computer science area that powers machines to learn and improve on their own without being programmed. The ML algorithms through data pattern analysis are able to predict, categorize, and reveal useful information that would otherwise be difficult to extract using traditional programming methods. These algorithms cover various learning paradigms such as supervised, unsupervised, semi-supervised, and reinforcement learning. Moreover, we have the concept of deep learning, which is a very advanced part of the field of ML and which has turned into a great and efficient solution to get new insights and make decisions based on enormous datasets [1, 2].

Following this general overview of machine learning, it is essential to understand the major learning paradigms that define how models are trained and applied to real-world scenarios. These paradigms form the theoretical basis for selecting appropriate algorithms, depending on the problem domain and data availability.

As described by [1], machine learning techniques are typically classified into four primary types: *supervised learning*, *unsupervised learning*, *semi-supervised learning*, and *reinforcement learning*. Each of these approaches addresses specific categories of problems and plays a vital role in the development of intelligent systems.

- **Supervised learning** is based on labeled datasets, the label helps the model map an input to an output. It is used in tasks such as classification, image recognition, or regression.
- **Unsupervised learning** does not need labels. The model has to find the patterns on its own within the dataset. Common applications include clustering, anomaly detection, and dimensionality reduction.

- **Semi-supervised learning** mixes elements of supervised and unsupervised learning. Usually a small amount of labeled data is mixed with a large amount of unlabeled data. Applications include fraud detection, machine translation, and medical diagnosis.
- **Reinforcement learning** exploits the benefits of the feedback mechanism through rewards and penalties. The model develops a strategy thanks to previous experience. This paradigm is well suited for decision-making scenarios such as autonomous driving or real-time strategy games (such as chess).

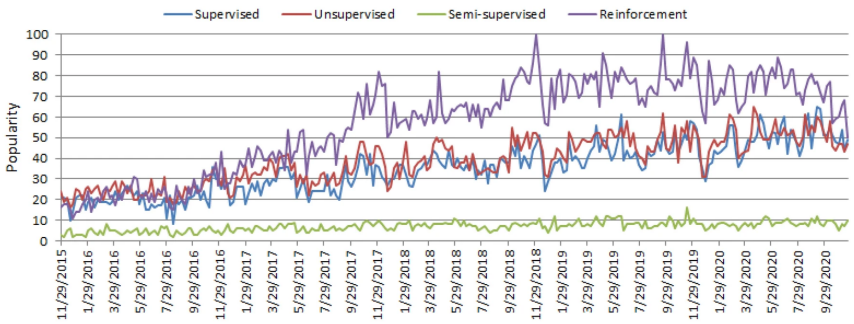


Figure 1.1: Popularity of ML algorithms up to 2020 [1].

1.2 Deep Learning

Starting from the concept of machine learning, it is also worth analyzing the concept of deep learning, a subset of ML that has revolutionized the world of AI compared to the concept of shallow learning, often outperforming traditional approaches in distinct domains [3]. DL is capable of a deeper feature extraction from the input data, thanks to the inclusion of hidden layers between the input and output.

Deep learning has its roots in simple neural models such as perceptrons. However, the field has grown sufficiently and become more complex, and we now have architectures such as convolutional neural networks (CNNs), recurrent neural networks (RNNs), and transformers at our disposal. These enabled the models to be not only general-purpose but also capable of handling such tasks as image recognition, natural language processing (NLP), and sequential data analysis. The depth of a network, that is the number of hidden layers, the factor that determines its ability to learn increasingly abstract concepts, hence the sophistication of the model by several orders of magnitude as opposed to shallow networks [3, 2].

The history of Deep Learning is made up of several main events such as the creation of CNNs, networks that have changed the whole game of computer vision tasks, by detecting spatial hierarchies in the image data, and the implementation of long short term memory (LSTM) networks, a type of recurrent neural network that addresses the problem of the vanishing gradient, which is responsible for deep learning of the sequence of data. Through this method, the new practice in the

NLP area will impact not only parallelized processing of the sequences but also new state-of-the-art performance on various benchmarks [3].

It speaks to the wide span of DL that has demonstrated the significant influence of scientific and industrial sectors, from healthcare diagnostics and autonomous driving to financial modeling and molecular chemistry. Furthermore, the introduction of new practices in the training domain, such as self-supervised learning and federated learning, is revolutionizing DL toward more efficient, scalable, and privacy-preserving applications.

Another theorem that serves as an essential part of the theoretical grounding of neural network applications is the universal approximation theorem, which states that, indeed, if you have large enough neural networks, you are able to approximate a continuous function, and this shows that the networks can handle any complexity level and are quite effective in modeling highly complicated data patterns [3]. To recap, deep learning is one of the most profound and fastest growing disciplines of AI which is always renewing its scope and methodology by continually introducing new architectures and training techniques.

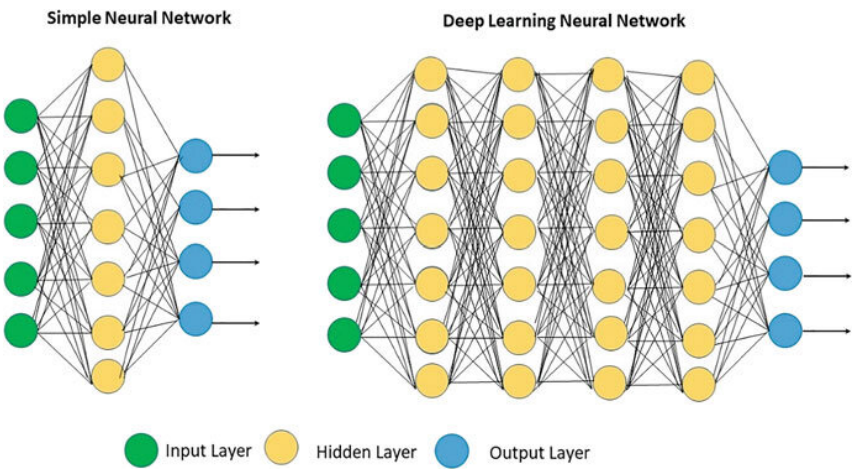


Figure 1.2: Difference between normal and deep learning neural networks [4].

1.2.1 Layers

In a deep learning model, layers are the fundamental units that process data through multiple stages. Common types of layers include:

- **Fully connected layers**, where each neuron connects to every neuron in the previous layer.
- **Pooling layers**, which reduce spatial dimensions by summarizing features (e.g., max or average pooling).
- **Convolutional layers**, a filtering operation that has the goal of extracting feature from the input.

Underpinning the numerous layers are artificial neurons, the basic unit of computation inspired by the biological neuron. With each neuron receiving one or many inputs, weights are then applied, then a bias term, and then a non-linear activation function to produce an output. Mathematically, the output y of a neuron can be expressed as

$$y = g\left(\sum_{i=1}^{n} w_i x_i + b\right),$$

where x_i are the inputs, w_i are the weights, b is the bias, and $g(\cdot)$ is the activation function, such as ReLU, sigmoid, or tanh.

This mechanism allows neurons to learn complex patterns by adjusting the weights and bias during training to minimize prediction errors.

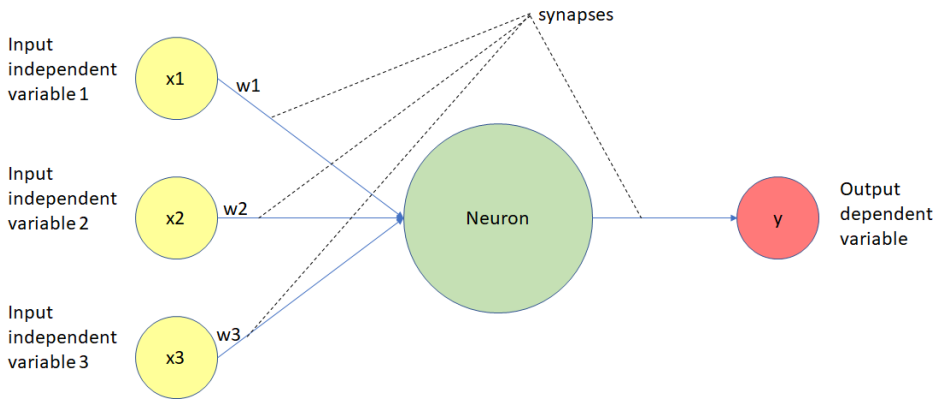


Figure 1.3: Graphical representation of a neuron [5].

1.2.2 Activation functions

Activation functions are fundamental to a Deep Learning network because of the non-linearity that are able to introduce to the model, allowing it to learn more complex patterns and make better predictions. Without non-linearity, the model would simply map intricate relationships between data. Popular activation functions are the sigmoid and the Rectified Linear Unit (ReLU). The sigmoid function is more useful for binary classification problems, since it maps the input values to the range (0,1). However, this function suffers from the vanishing gradient problem. On the other hand, ReLU directly outputs the input value if it is positive, 0 being returned otherwise [3].

1.2.3 Convolutional Neural Networks

Convolutional neural networks (CNNs) are constructed around layers that are purposefully produced to carry out the spatial structure of the given data, such as in the case of images.

Convolutional layers use small local regions of the input to filter (kernels), where they then produce feature maps that contain localized patterns.

This local connectivity of the network changes the representation, but the number of parameters is still less than in the case of fully connected layers, which makes it easy to learn translation-invariant properties.

Let us assume that we have an input tensor $\mathbf{x}$ of size $H \times W \times C$, the output of a convolutional layer neuron that is mathematically computed is as follows:

$$z^l_{i,j,d} = \sum_{m=0}^{k_1} \sum_{n=0}^{k_2} w^l_{m,n,d}\, x_{i\cdot s+m, j\cdot s+n, d}, \quad a^l_{i,j,d} = g(z^l_{i,j,d}),$$

where $w^l_{m,n,d}$ are the filter weights, s is the stride, and g is the activation function.

After convolution, pooling layers are often applied to reduce the spatial dimensions, summarized as:

$$a^{l+1} = \text{pool}(a^l_{i\cdot s+m, j\cdot s+n}),$$

where pool is typically a max or average operation, as illustrated in Figure 1.4.

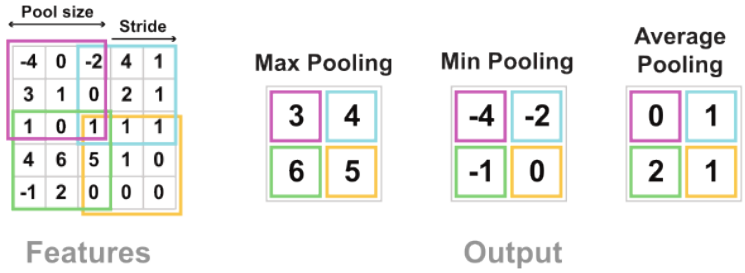


Figure 1.4: Pooling operation that reduces the spatial size of feature maps [6].

Stacking convolutional and pooling layers enables CNNs to learn hierarchical features, from edges and textures to complex objects, forming the basis of many state-of-the-art computer vision models such as YOLO [7, 8].

Figure 1.5 shows the overall structure of a typical convolutional neural network, illustrating how the convolutional layers, pooling layers, and the fully connected layers are stacked to extract and classify characteristics.

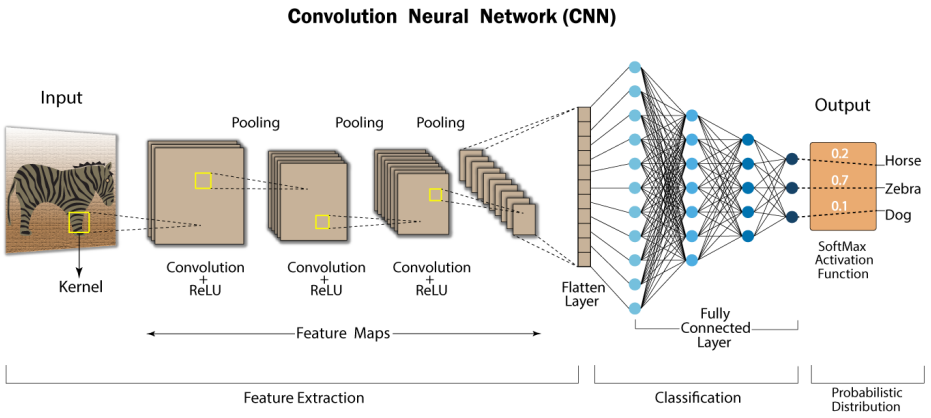


Figure 1.5: Structure of a convolutional neural network (CNN) [9].

1.3 Computer Vision and Object Detection

According to [8] Computer vision (CV) is all about training machines to be able to process and work with images and videos. In this way, CV algorithms have the ability to extract, recognize, and make decisions with minimal or no human intervention. The final aim here is to learn to act as a human being using a combination of image processing, pattern recognition, and machine learning tools. The main areas of deep learning usage are tasks such as image classification, object detection, facial recognition, the functioning of self-driving cars, and the study of images.

Object detection is a field of computer vision in the digital era that finds and delineates individual objects, animals, cars, etc., in images. Modern approaches to object detection predominantly employ deep learning and, in particular, CNNs, to acquire and analyze data from images. In general, the approach here is to identify a candidate object for classification after feature extraction. After the correct classification (which is itself a branch of CV) the object has to be detected on the image, and the correct bounding box has to be drawn around the object.

As written by [10] object detection nowadays is divided into two different types: two-stage and one-stage object detection. Two-stage object detection is based on the proposal of the region of interest (first stage), and the classification made by a separate network analyzes these regions and the relative features. Then predicts the class of each object and refines the coordinates of the corresponding bounding box. Some relevant examples of this approach are R-CNN, Fast R-CNN and Faster R-CNN. One-stage detection on the other applies a single neural network to the image to directly predict objects. These types of algorithms divide the image into a grid of cells, each responsible for detecting an object centered in it. A key role is played by the **anchor boxes** which are generated by the network and are then scored for objectness and after filtering the best ones are chosen, thus removing the need for a second stage. Efficient one-stage detectors are RetinaNet and YOLO, the latter of which was used for this project and will be analyzed in depth in the course of the next chapter.

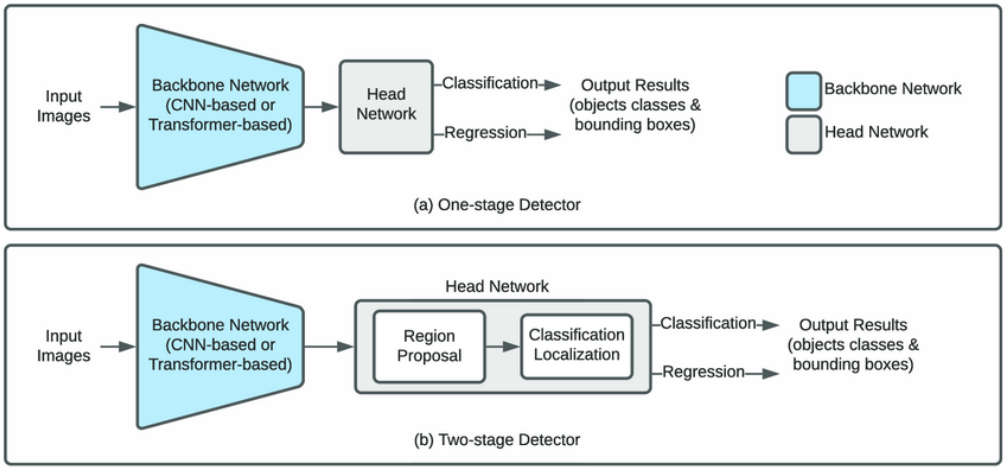


Figure 1.6: Difference between one stage and two stage object detection [11].

Chapter 2

YOLO

As described in the previous chapter, current detection systems like R-CNN use the region proposal approach to first generate bounding boxes and then a repurposed classifier to perform detection. This pipeline is slow and hard to optimize also because implies the post-processing which serves as a "cleaning" of the result (elimination of duplicates and rescoring of the boxes). YOLO proposes to use a single convolutional network to simultaneously predict boxes and class probabilities. This allows the basic YOLO version to process images in real time at 45 frames per second [12]. YOLO is able to achieve more than twice the mean average precision of other real-time systems.

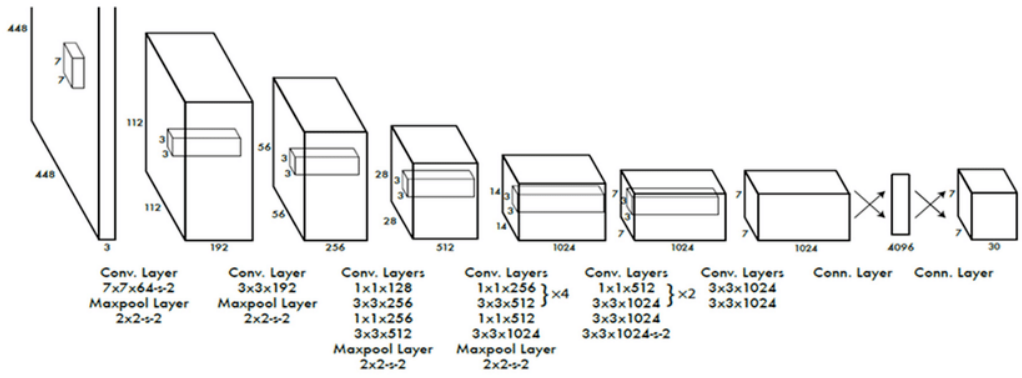


Figure 2.1: YOLOv1 architecture [12].

2.1 Metrics

How do we tell if a model is performing well or not? We have some key statistics to keep in mind: precision, recall and mean average precision.

Precision is defined as

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

where TP stands as True Positive and FP as False Positive. Recall defines how many

of the real positives can be spotted and is defined as

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \text{ with FN as False Negatives}$$

Lastly, we have mAP, defined as

$$\text{mAP} = \frac{1}{C}\sum_{c=1}^{C} \text{AP}_c \quad \text{with C Classes and AP defined as} \quad \text{AP} = \int_0^1 p(r)\, dr$$

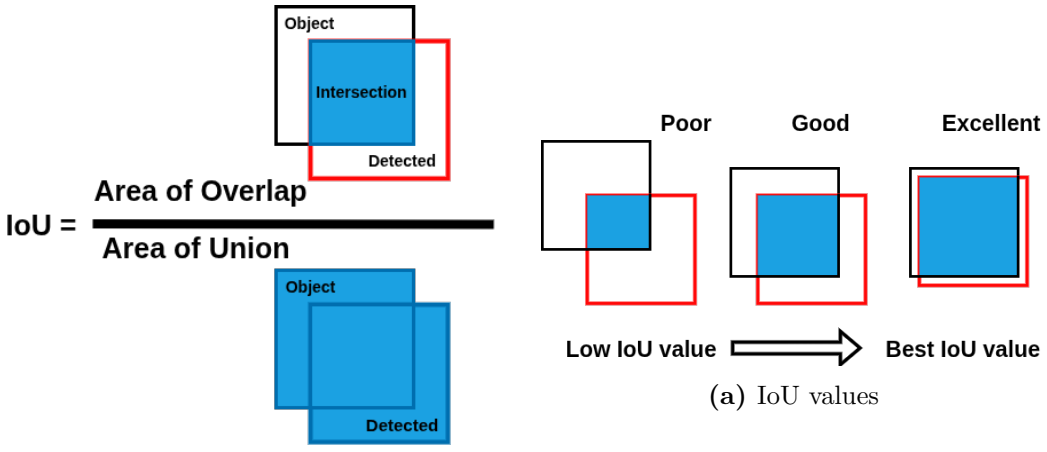
where *p(r)* is the precision depending on recall. mAP defines whether a prediction is correct or not, based on the threshold set for the value.

2.2 Detection pipeline

Yolo divides the input image in a S X S grid. As said before, if the center of an object falls into a grid cell, then that cell is responsible for the detection of that object. Each cell predicts and scores B bounding boxes. The score is used to reflect how confident the model is that the box contains an object. The definition of confidence is as follows:

$$\Pr(\text{Object}) \cdot \text{IoU}^{\text{pred}}_{\text{truth}}$$

If no object is present in a cell it should be zero, otherwise the score should be equal to Intersection Over Union (IoU) [12], which is used as a threshold for a correct prediction, between the prediction and the ground truth and is defined as:



(a) IoU values

Figure 2.2: IoU formula, and examples with IoU values.

Each bounding box consists of 5 predictions: *x, y, w, h* and confidence. *(x,y)* coordinates represent the center of the box relative to the bounds of the grid cell. Width and height, on the other hand, are related to the whole image. The confidence represents the IOU said before. Each grid cell also predicts conditional class probabilities as $\Pr(\text{Class}_i \mid \text{Object})$. Only one set of class probabilities per grid cell is predicted [12].

The prediction made by the model is encoded as *S X S X (B * 5 + C)* with S = grid size, B = number of boxes, and C = number of classes.

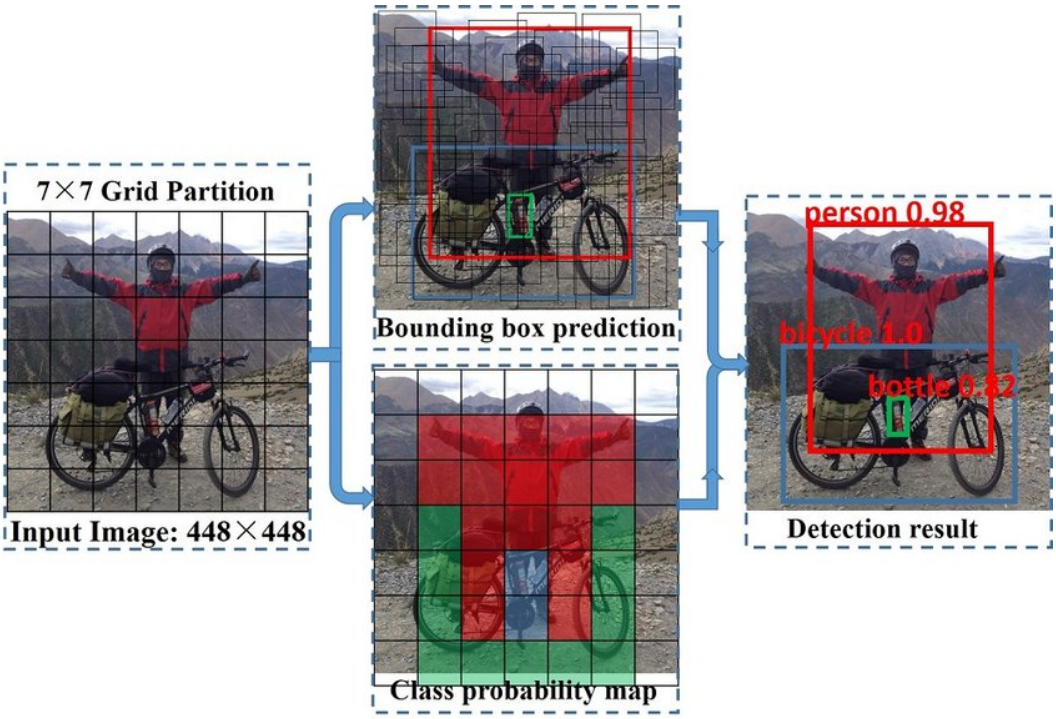


Figure 2.3: Detection pipeline [13].

2.2.1 Anchor Boxes

Anchor boxes are used by YOLO to help the model predict objects. Anchor boxes are predefined boxes, from which the model starts to resize and reshape the final bounding box.

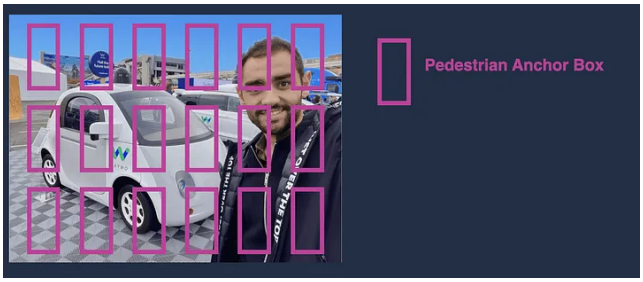


Figure 2.4: Anchor box example [14].

As described by [15], the YOLOv5 predicts the coordinates of the bounding box as offsets relative to a predefined set of anchor box dimensions.

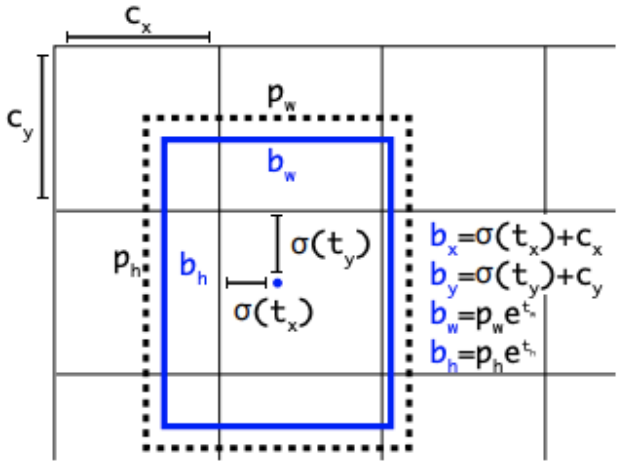


Figure 2.5: Bounding box prediction based on anchor box [16].

What is important to keep in mind is that the anchor boxes are used **on top** of the feature map and also that anchor boxes **are not** the bounding boxes but a predetermined shape for a certain class.

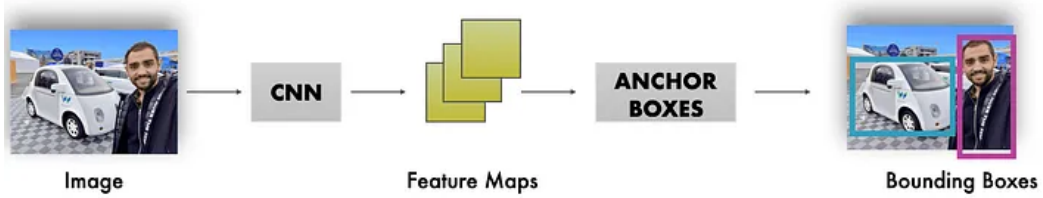


Figure 2.6: Anchor boxes are used on top of a feature map [14].

2.2.2 Non-maximum suppression

Non-maximum suppression (NMS) is described by the same creators of the YOLO repository in [17] as a key post-processing technique that is widely used in CV. More specifically in object detection and by its pipeline. The main goal is to refine the raw output of the model that usually identifies multiple overlapped bounding boxes of the same object. In order to do this the main feature is the IoU that is used as a threshold to delete the redundant boxes and keep the best one for the image, this helps not only the mean average precision (mAP), a key metric in the object detection, but also reduces false positives, resulting in a better all-around model.

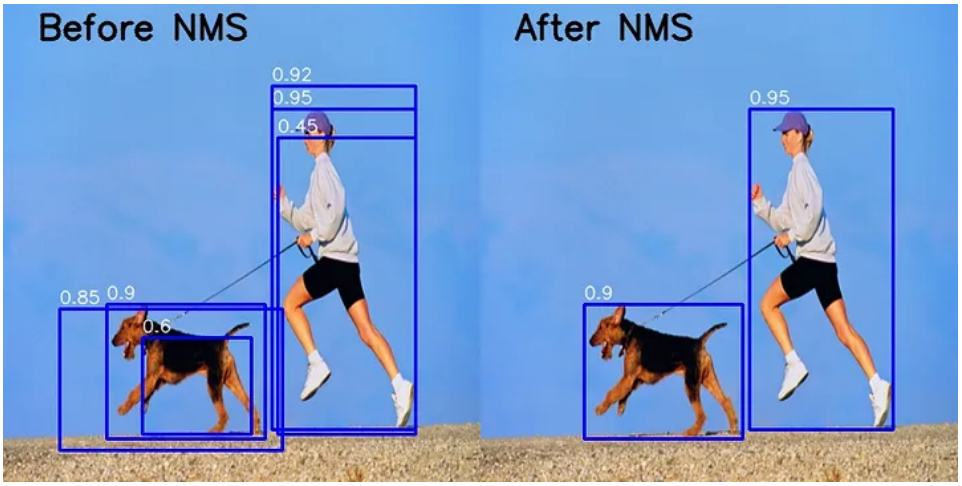


Figure 2.7: NMS and its effect on the final result [18].

2.2.3 Loss function

The loss function is a key component of a CNN, it is used to define how well the model performs by comparing the prediction to the ground truth. An example can be quadratic loss. The first version of YOLO optimizes during training this loss function is

$$\begin{aligned}\mathcal{L} = \lambda_{\text{coord}} &\sum_{i=0}^{S^2}\sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 + \left(\sqrt{w_i} - \sqrt{\hat{w}_i}\right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i}\right)^2\right] \\ &+ \sum_{i=0}^{S^2}\sum_{j=0}^{B} \left[\mathbb{1}_{ij}^{\text{obj}}(C_i - \hat{C}_i)^2 + \lambda_{\text{noobj}}\mathbb{1}_{ij}^{\text{noobj}}(C_i - \hat{C}_i)^2\right] \\ &+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c\in\text{classes}} (p_i(c) - \hat{p}_i(c))^2\end{aligned}$$

where $\mathbb{1}_{i}^{\text{obj}}$ denotes whether an object appears in cell i, and $\mathbb{1}_{ij}^{\text{obj}}$ indicates that the j-th bounding box predictor in cell i is responsible for detecting the object [12].

[15] describes the loss function for YOLOv5 is a composite of Binary Cross-Entropy (BCE) for class prediction and objectness, and Complete Intersection over Union (CIoU) for localization. The function is a weighted sum of these losses.

$$\text{Loss} = \lambda_1 \cdot L_{\text{cls}} + \lambda_2 \cdot L_{\text{obj}} + \lambda_3 \cdot L_{\text{loc}}$$

It is important to notice that the BCE is relevant due to the nature of the binary nature of the problem (object is present or not, if an object is present it is either an element of a class or not) and the CIoU is also an improvement with respect to standard IoU, it allows a better optimization and a quicker convergence. An efficient training is also related to how well the λ_1, λ_2, λ_3 are chosen.

This is an improvement compared to YOLOv1 and the loss function that it uses.

2.2.4 Different versions

As stated before the YOLO algorithm is constantly developing and the latest version (YOLO 11) got released just a few months ago. However for this thesis work the versions that were used are YOLOv3 (for a first approach) and after a few changes the final model was the YOLOv5. YOLOv5 has different versions that are used for several needs. As stated by [15] the different versions are:

- **YOLOv5n (Nano)**: designed for resource constrained projects. Its compact and the FP32 version is around 4MB, useful for edge devices or IoT platforms.
- **YOLOv5s (Small)**: standard and baseline model, this has 7.2 million parameters and is suitable for CPUs
- **YOLOv5m (Medium)**: a compromise between speed and accuracy, this version has 3 times the parameters of the s version
- **YOLOv5l (Large)**: more than twice the parameters of the m model, more suited for small objects detection.
- **YOLOv5x (Extra Large)**: this version has the highest mAP and has twice the number of parameters of the large version.

The table below shows metrics related to a 640x640 image, with accuracy, speed and size.

Table 2.1: Comparison of YOLOv5 variants on 640x640 images.

Model	**mAP@50:95** (%)	**mAP@50** (%)	**CPU** (ms)	**GPU1** (ms)	**GPU32** (ms)	**Params** (M)	**FLOPs** (B)
v5n	28.0	45.7	45	6.3	0.6	1.9	4.5
v5s	37.4	56.8	98	6.4	0.9	7.2	16.5
v5m	45.4	64.1	224	8.2	1.7	21.2	49.0
v5l	49.0	67.3	430	10.1	2.7	46.5	109.1
v5x	50.7	68.9	766	12.1	4.8	86.7	205.7

The first two columns show the mAP with high confidence and with lower confidence (with IoU between 50% and 95% and below 50%). CPU shows the inference time of when is run on a CPU, then when is run on an AWS p3.2xlarge V100 instance (which uses an NVIDIA Tesla V100) with a batch size of 1 image and with a batch size of 32 images. Then the number of parameters that were previously explained and finally the number of floating point operations required for a single 640x640 image [19, 15].

The graph 2.8 has different useful data for the purpose of this chapter. On the y-axis, COCO AP val denotes the mean Average Precision (mAP) at Intersection over Union (IoU) thresholds from 0.5 to 0.95, measured in the 5,000-image COCO val2017 dataset at various inference sizes (256 to 1536 pixels). The x-axis shows the GPU Speed of the average inference time per image on the COCO val2017 dataset using the same AWS p3.2xlarge V100 instance as for the previous table, with a batch

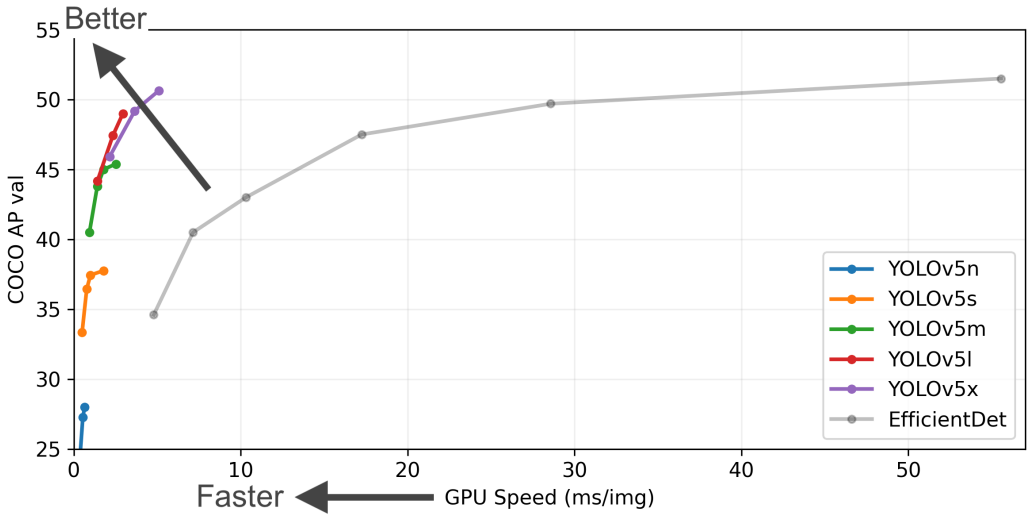


Figure 2.8: Metrics of the different models [19].

size of 32. EfficientDet data is sourced from the google/automl repository at batch size 8 [19].

2.3 Quantization

As stated in the previous chapter, YOLO, in its standard version, uses floating point precision to define weights and activations. In order to make the whole model lighter and able to run on an FPGA, it is mandatory to compress the model without massive losses on accuracy. To do this, it is useful to describe the quantization process.

Quantization compresses and reduces the computational and memory cost of the model by changing the representation of data to INT8 instead of the standard FP32.

Figure 2.9: Basic example of quantization [20].

This raises two fundamental questions: What does this transformation bring and how does it affect the model? It could be useful to take a step back in order to understand this important part of the thesis work.

2.3.1 Data representation

The data types relevant to this chapter are INT8 and FP32, which respectively mean 8-bit integer and 32-bit signed floating point. INT8 has two different usages:

- For weights, it uses a *signed integer*, where 1 bit (the most significant) is for the sign, and the other 7 are used for the value in the range $[-128, 127]$.
- For activation functions, it uses an *unsigned integer*, in the interval $[0, 255]$.

Both are very light and fast in inference. FP32, on the other hand, uses 32 bits: 23 for the mantissa and 8 for the exponent, resulting in a more granular representation at the expense of speed and memory usage.

This means, as seen in Figure 2.9, that instead of the wide range of FP32 values, INT8 only has 256 values, introducing a challenge: how to round the weights and activations to those values with minimal accuracy loss but significant gain in model speed and size.

2.3.2 Quantization techniques

Let's first introduce some theoretical concepts to better understand the techniques described by [21].

The idea is to map a real-valued input to a discrete set $\mathcal{Q}$ of quantization levels:

$$Q : \mathbb{R} \to \mathcal{Q} \quad \text{with} \quad |\mathcal{Q}| = 2^b$$

For uniform quantization with step size Δ, the quantization error is bounded by:

$$|x - Q(x)| \leq \frac{\Delta}{2}$$

By controlling Δ, we can manage the trade-off between accuracy and resource efficiency.

The formal definition of *linear quantization* is:

$$q = \text{round}\left(\frac{x}{s}\right) + z$$

where the floating point value x is mapped to the integer q using the scaling factor s and zero-point z, with $s \in \mathbb{R}^+$ and $z \in \mathbb{Z}$.

The two main approaches are Quantization Aware Training (QAT) and Post-Training Quantization (PTQ). The first one integrates the quantization in the training phase; this helps the model tune the parameters for lower precision representations. The idea is to simulate quantization during training, allowing the model to learn parameters that are more robust when precision reduction is applied [21]. PTQ is instead applied to a pre-trained model, first the range of the parameters is checked by a calibration set. This approach is more suitable for simpler but efficient tasks [21],

it can be either *static* or *dynamic*, the difference resides in the moment of application (static is right after training, while dynamic is at run-time) and also the static applies quantization to both weights and activations, while dynamic only on the activation functions because the weights are already quantized. Another relevant aspect is that dynamic quantization keeps some data in FP32, while static is fully converted to INT8.

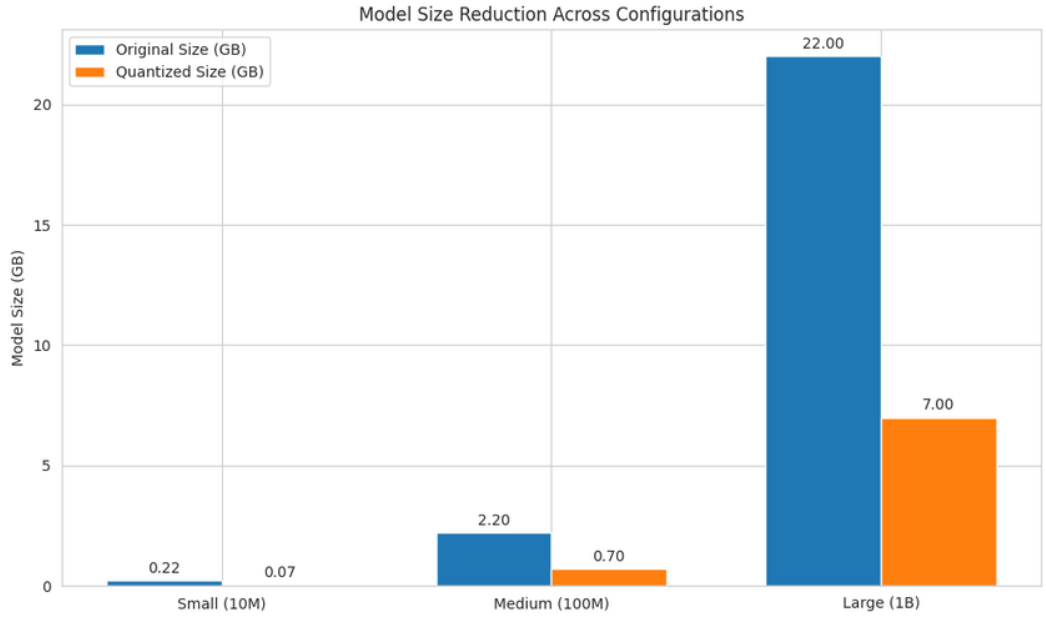


Figure 2.10: Model size pre and post quantization [21].

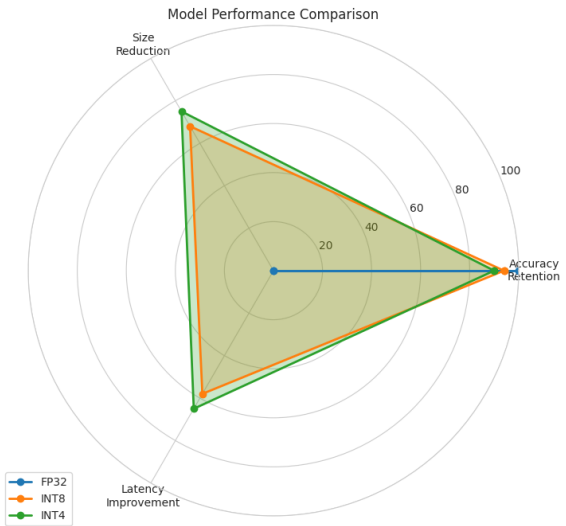


Figure 2.11: Performance metrics related to data representation [21].

As illustrated in Figures 2.11 and 2.10, reported by [21], the performance and size of the model before and after quantization are shown.

2.3.2.1 PTQ and QAT algorithms

Algorithm 1 Quantization-Aware Training (QAT) [21].

Require: Model parameters Θ, learning rate η, training data $\mathcal{D}$
Ensure: Quantization-aware trained parameters Θ^*

while not converged **do**
  $B \leftarrow \text{SampleBatch}(\mathcal{D})$
  $\hat{\Theta} \leftarrow \text{Quantize}(\Theta)$ ▷ Forward quantization
  $L \leftarrow \text{Loss}(\hat{\Theta}, B)$
  $g \leftarrow \nabla_{\Theta} L$ ▷ Use Straight-Through Estimator
  $\Theta \leftarrow \Theta - \eta \cdot g$
end while
return Θ^*

Algorithm 2 Post-Training Quantization (PTQ) [21].

Require: Pre-trained model parameters Θ, bit-width b, calibration dataset $\mathcal{D}_{\text{cal}}$
Ensure: Quantized model parameters $\hat{\Theta}$

$(x_{\min}, x_{\max}) \leftarrow \text{ComputeRange}(\Theta, \mathcal{D}_{\text{cal}})$
$s \leftarrow \frac{x_{\max} - x_{\min}}{2^b - 1}$
$z \leftarrow \text{round}\left(\frac{-x_{\min}}{s}\right)$
for each tensor T in Θ **do**
  $q_T \leftarrow \text{round}\left(\frac{T}{s}\right) + z$
  $\hat{T} \leftarrow (q_T - z) \cdot s$
  $\hat{\Theta}[T] \leftarrow \hat{T}$
end for
return $\hat{\Theta}$

2.3.3 Scaling factor

In the previous section, we introduced the concept of scaling factor. It is crucial for quantization as it translates between floating-point and integer values, essentially it defines how much a change of one unit in the integer domain corresponds to in the floating-point domain.

A relevant hardware optimization is the *power-of-two scaling factor*. As explained in [22], this enables bit-shifting rather than more complex operations, which is faster and more hardware friendly.

For uniform quantization, the scaling factor is computed as

$$s = \frac{x_{\max} - x_{\min}}{2^b - 1}$$

Chapter 3

Processing the datasets

After talking about all the knowledge needed, it is time to take the first steps toward the implementation of the project. This chapter outlines the different datasets on which the models have been trained, the preliminary pre-process phase and how it was thought and developed. The pre-process step had to structure the data that were supposed to be handled later in a defined format, the YOLO format, which will be discussed in depth in the first section. Then these data were fed to the YOLOv3 model to obtain the first results.

3.1 Photovoltaic thermal images dataset

The first dataset is the **Fotovolt** dataset, for which the goal was to train a model capable of recognizing faulty cells in a photovoltaic field. This dataset is made up of 1008 images sized 512x640 pixels and corresponding labels, it is encoded in 3 numpy files `imgs_temp.npy`,`imgs_check.npy` and `imgs_masks.npy`. A numpy file is a multidimensional tensor, which contains multiple data for each image and has to be loaded correctly (functions contained in the *numpy* library are able to do this) by the script in order to be translated and used.

`imgs_temp.npy` contains the thermal images. The second and third dimensions of array respectively indicate the x and y coordinates of the image pixels, where the values indicate the temperature in celsius degrees.

`imgs_masks.npy` contains the mask images. The second and third dimensions of array respectively indicate the x and y coordinates of the image pixels, where the values indicate the area of anomalous cells:

- **Label 0:** if there is no anomaly.
- **Label 1:** if there is an anomaly.

And the last one, `imgs_check.npy` contains a 3-value label for each image:

- **Label 0:** images with one anomalous cell.
- **Label 1:** images with more than one anomalous cell.

- **Label 2:** images with a contiguous series of anomalous cells.

It is important to keep in mind that all the pictures contained at least one defect, there were no images without faults. The first task was to create the typical YOLO structure, which is the following one:

```
/dataset
|-- images
|   |-- train
|   |   |-- img1.jpg
|   |   |-- img2.jpg
|   |   `-- ...
|   `-- val
|       |-- img101.jpg
|       |-- img102.jpg
|       `-- ...
|-- labels
|   |-- train
|   |   |-- img1.txt
|   |   |-- img2.txt
|   |   `-- ...
|   `-- val
|       |-- img101.txt
|       |-- img102.txt
|       `-- ...
|-- train.txt
`-- val.txt
```

Here each `.txt` file inside the labels directory contains the labels related to the image with the same name, with this structure:

`<class_id> <x_center> <y_center> <width> <height>`

And each value is normalized to the range $[0, 1]$ because normalization allows the model to scale the boxes to different sizes. The `train.txt` and `val.txt` contain the path to the images, these files will be used later for the training phase, but it was useful to show them in the structure anyway.

3.1.1 Pre-processing script

Here are the key parts of the script `preproc.py`

Listing 3.1: Image preprocessing.

```
def preprocess_image(image):
    if len(image.shape) == 2:
        image = cv2.cvtColor(image, cv2.COLOR_GRAY2BGR)
    image = image / 255.0
    return image
```

This simple function is the one that draws the image and converts the original image which is a grayscale image to an RGB image (not because of the coloring, but because the number of channels can help the training process). Furthermore, this function normalizes the values of each pixel to uniform data.

Listing 3.2: Train/Validation split.

```
for i in range(total_number_of_images):
    if i % 5 == 0:
        save image and label in validation set
    else:
        save image and label in training set
```

This part is responsible for correctly splitting the 2 sets of images and labels between train and validation, for the purpose of this part it is not useful to write the code in its entirety but the idea was to divide the dataset into 80% of data for the training and the remaining 20% for validation.

Listing 3.3: Single bounding box computation.

```
if check in [0, 2]:
    indices = where(mask == 1)
    min_x, max_x = min(indices[1]), max(indices[1])
    min_y, max_y = min(indices[0]), max(indices[0])

    center_x = (min_x + max_x) / 2 / img_width
    center_y = (min_y + max_y) / 2 / img_height
    width = (max_x - min_x) / img_width
    height = (max_y - min_y) / img_height

    write to label file: class_id center_x center_y width
    ↪ height
```

Here, the code takes the cases of single defects, whether it is a single cell or multiple contiguous cells. The script takes the values from the numpy tensor and normalizes such values, considering that the numpy files contain the coordinates encoded as *(min x, max x, min y, max y)* and these have to be converted in center, width, and height.

Listing 3.4: Multiple bounding boxes via connected components.

```
if check == 1:
    labeled_array, num_features = label(mask)

    for component_id in range(1, num_features + 1):
        indices = where(labeled_array == component_id)
        min_x, max_x = min(indices[1]), max(indices[1])
        min_y, max_y = min(indices[0]), max(indices[0])

        center_x = (min_x + max_x) / 2 / img_width
        center_y = (min_y + max_y) / 2 / img_height
        width = (max_x - min_x) / img_width
        height = (max_y - min_y) / img_height

        write to label file: class_id center_x center_y width
    ↪ height
```

This final piece of code does the same thing as the previous code segment, but takes into account the case of multiple not contiguous cells (`check==1`), and correctly saves multiple labels in the text file.

3.1.2 Ground Truth directory

After the pre-processing of the dataset, another script `BB.py` was created to correctly create a directory named **groundTruth** which contained the images with the corresponding boxes (the correct ones, which the YOLO model uses to **calculate IoU**) calculated from the label *.txt* file paired with the image *.jpg* file.

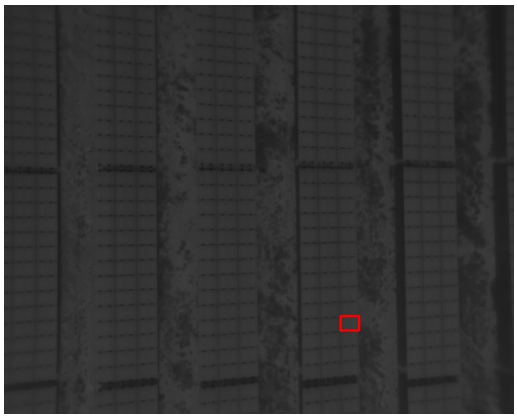
The script is relevant for the part where it is key to de-normalize the labels as shown in the following code snippet.

Listing 3.5: Main loop processing YOLO bounding boxes from label files.

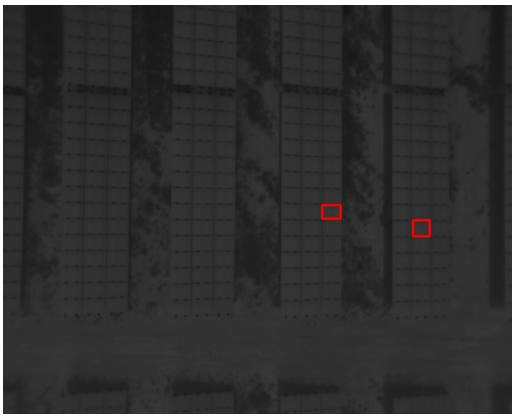
```
with open(input_file, 'r') as file:
    lines = file.readlines()
    for line in lines:
        center_x, center_y, width, height = map(float, line.
    ↪ strip().split()[1:])
        x1 = int((center_x - width / 2) * og_width)
        y1 = int((center_y - height / 2) * og_height)
        x2 = int((center_x + width / 2) * og_width)
        y2 = int((center_y + height / 2) * og_height)
        cv2.rectangle(img, (x1, y1), (x2, y2), (255, 0, 0), 2)
```

The operation to derive the coordinates needed by the function of **CV2**, which are the bottom left and top right coordinates, is done by denormalizing the values with `og_width` and `og_height`, respectively 640 and 512, which is the size that was used to normalize the label in *preprocess.py*.

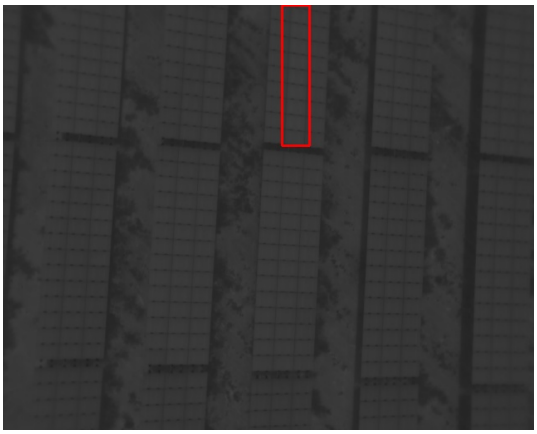
The results of the three variants (single, contiguous and multiple non-contiguous) are shown below.



(a) Single defect.



(b) Multiple non-contiguous.



(c) Contiguous defects.

Figure 3.1: The three cases of the fault present in the photovoltaic dataset extracted from the numpy models.

3.2 ASXL dataset

The ASXL dataset is a collection of images of bridges; these bridges are either safe or cracked, the goal is to train a model capable of spotting such cracks. The dataset contains 50 images of 6000x4000, which is a size that is not feasible for edge inference. The pre-processing of this dataset had to be a little different due to the nature of these images, which resulted in the segmentation of this dataset into 512x512 images.

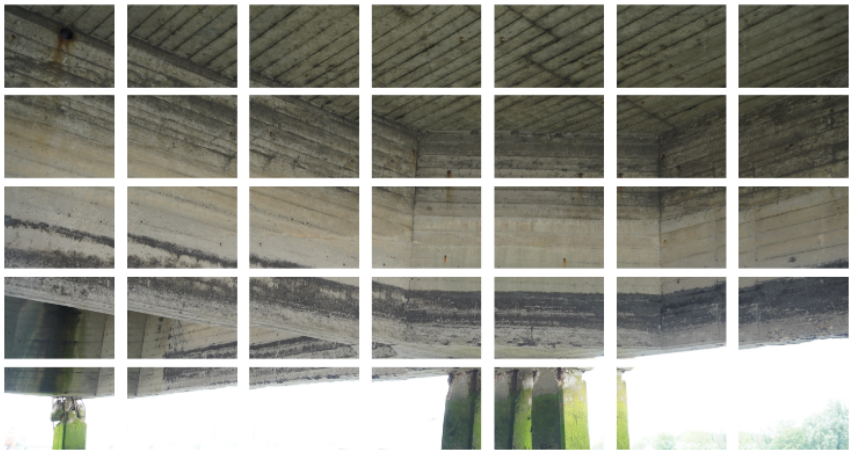


Figure 3.2: Segmentation of the original image.

These segments have been pre-processed as in 3.1, which resulted in the creation of **1210 training** files (images and labels) and **347 validation** data along with **172 test data**.

In order to extrapolate the labels for each image, each segment has been masked to obtain only the highlighted crack, then the label of the boxes have been applied.

(a) Segment of the image.

(b) Mask of the segment.

(c) Ground Truth image with the correct bounding box applied.

Figure 3.3: Processing pipeline: from the segment to the ground truth image.

Figure 3.3 shows the full processing of the images, this finally results, when the final images are assembled back with each segmentation combined, into this image processing:

Figure 3.4: Original image with every segment's bounding box applied.

Due to the difference in size between the images on which the model is trained and the full-scale images, which resulted in a great discrepancy between the dimensions of the bounding box , this model has been difficult to train; however, the results will be analyzed in-depth in 6.

3.3 COCO dataset

COCO (Common Objects in Context) is a popular benchmark dataset designed for different tasks such as segmentation, captioning, or object detection. It contains more than **330,000 images**, including more than **200,000 labeled images** with **1.5 million object instances** across **80 object categories**.

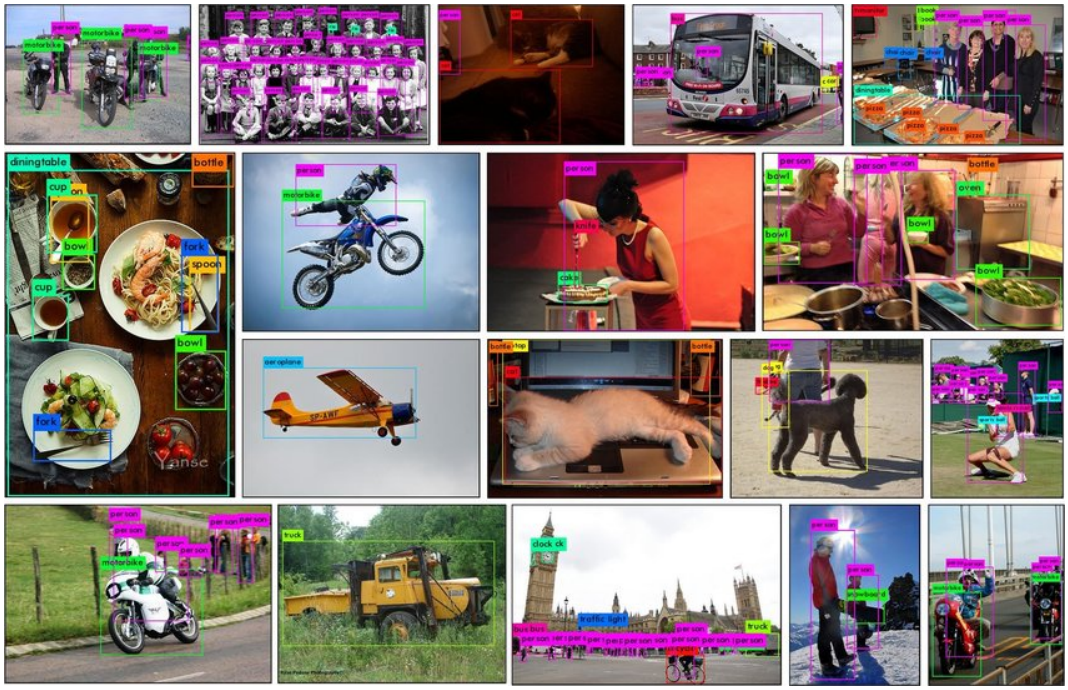


Figure 3.5: Coco batch example [23].

For the goal of this thesis, the training of a working model on the COCO dataset has been used as a benchmark for further validation of the results obtained.

In other words, obtaining good results on COCO, with 80 classes, different bounding box sizes, and real-world scenarios, is additional evidence of the correct work done with this project.

This means that validation on this dataset is seen as a counter-proof of validity of the workflow adopted and certificates the value of each tool used.

Chapter 4

First training with YOLOv3

This chapter will discuss the structure of the Yolov3 model, which has been tested for the first approach and was the first step toward the exploration of different settings, this model has been trained only on the Fotovolt dataset discussed and described in 3.1. After the introduction to the model structure this chapter will also describe the training process with an in-depth analysis of the different parameters that are used to configure the training itself. Also the results of the first training will be shown and discussed.

4.1 Model structure and training components

The first approach was to use a YOLOv3-Tiny version with normal FP32 layers and then change the model structure to a quantized version, the last two ingredients needed for the training phase are the two `.yaml` files: one for the model structure `yolov3-foto.yaml` and one that contains the classes and paths to the dataset directory `foto.yaml`. The latter is reported in the following.

Listing 4.1: Dataset configuration file for YOLO training.

```
path: fotovolt_test/fotovoltaic_dataset
train: train.txt
val: val.txt
nc: 1
names: ['anomaly']
```

This file contains the path to the two files with the paths to the images needed for the training, the path to the dataset directory, and the number of classes with the name.

Listing 4.2: YOLOv3-Tiny quantized model configuration without comments.

```
backbone:
  [
   [-1, 1, Conv, [16, 3, 1]],
   [-1, 1, nn.MaxPool2d, [2, 2, 0]],
   [-1, 1, Conv, [32, 3, 1]],
   [-1, 1, nn.MaxPool2d, [2, 2, 0]],
   [-1, 1, Conv, [64, 3, 1]],
   [-1, 1, nn.MaxPool2d, [2, 2, 0]],
   [-1, 1, Conv, [128, 3, 1]],
   [-1, 1, nn.MaxPool2d, [2, 2, 0]],
   [-1, 1, Conv, [256, 3, 1]],
   [-1, 1, nn.MaxPool2d, [2, 2, 0]],
   [-1, 1, Conv, [512, 3, 1]],
  ]
```

The backbone of the model consists of alternating `Convolutional` and `MaxPool2d` layers, where each layer takes the output from the previous one (`-1, 1`).

In the `convolutional` layers, the three parameters represent:

- the number of output channels (e.g., 16, 32, 64, ...), which defines how many filters extract features.
- the kernel size (e.g., 3, meaning a 3×3 filter), the size of the sliding window applied to the input.
- the stride (e.g., 1, defining the step size of the kernel), how many pixels the filter moves at each step.

For the `MaxPool2d` layers, the parameters are:

- the kernel size (e.g., 2, corresponding to a 2×2 window), the area over which the maximum value is taken.
- the stride (e.g., 2, meaning the pooling window shifts by 2 pixels), how far the pooling window moves each time.
- the padding (e.g., 0, meaning no extra pixels are added to the input), pixels added around the input edges to control output size.

The head of the model, which is in charge of taking the output of the backbone and elaborating data in order to generate classes and objects predictions, is the following one

Listing 4.3: YOLOv3-Tiny quantized model configuration without comments.

```
head:
  [[-1, 1, Conv, [1024, 3, 1]],
   [-1, 1, Conv, [256, 1, 1]],
   [-1, 1, Conv, [512, 3, 1]],
   [-2, 1, Conv, [128, 1, 1]],
   [-1, 1, nn.Upsample, [None, 2, 'nearest']],
   [[-1, 8], 1, Concat, [1]],
   [-1, 1, Conv, [256, 3, 1]],
   [[17, 13], 1, Detect, [nc, anchors]],
  ]
```

What immediately catches the eye is the reduction of the output channels of each layer; this is due to the high number of features extracted from the backbone, but since not all those features are relevant for the predictions, the head has to shrink the number of those features to use only the relevant ones. This speeds up the model and also reduces the size of it.

Then the `Upsample` resizes height and width of the feature map in order to be compliant with the previous one, the parameters are `[None, 2, 'nearest']` which refer to the fixed size of the output, *None* in this case, *2* is the scale factor, the input size has to be doubled, *'nearest'* refers to the nearest neighbor interpolation, this means that every pixel is simply copied in the nearest positions without complex interpolations involved.

The `Concat` takes the layer before, the `Upsample`, and combines it with the 8th layer of the model, which is the `[-1, 1, Conv, [256, 3, 1]]` located in the `backbone`, and this is done to improve predictions of small objects because is a merge between the deep feature map of the `Upsample` (*for example "this is a cat"*) with a high resolution feature map (*"there is a small object here"*).

The final layer, `Detect`, takes two different resolution layers (17th and 13th), the 17th for small objects and the 13th for large ones. This layer takes feature maps at different resolutions, and for each cell and anchor box predicts:

- The coordinates of the bounding box *(x, y, width, height)*, when present.
- The **objectness** (how likely is it that an object is present).
- The class score (which object it is, in our case this is trivial since the class is only 1).

4.1.1 Hyperparameters

After explaining the backbone and the head of the model, it is time to focus on the **hyperparameters**. The *hyps* are a set of configuration parameters used to control the process, usually these parameters are set in a `.yaml` file, which is the case here, and they are used to define the training process. The scope of each *hyp* will be explained, but these will be divided into groups to better understand each. The first group is related to training scheduling and optimization, which is the following.

- **Initial learning rate**: step size in each iteration to a minimum of a loss function.
- **Final learning rate**: learning rate at the end of the training process.
- **Momentum**: momentum factor used in the optimizer, helps for faster convergence by smooth.
- **Weight decay**: helps the model avoid overfitting (which is when the model learns training data too well and performs poorly on unseen data) by penalizing large weights.
- **Warmup epochs**: how many epochs are used for the warm-up phase, which is a phase where the learning rate gradually increases from a small value to the initial learning rate.
- **Warmup momentum**: momentum value used during the warm-up phase, usually small to avoid large updates.
- **Warmup bias lr**: specific learning rate for bias terms during warm-up.

These `hyps` regulate the pace, the stability at the beginning, and the consistency of the training process. Then there are all the loss function-related `hyps` which are the following:

- **Bounding box loss**: a coefficient that multiplies the regression loss for the bounding boxes (the error between the size and position of the correct and the predicted box).
- **Class loss**: same "weight" but for the classification loss.
- **Objectness loss**: the objectness loss explained in the previous section.
- **Objectness and class positive weight**: these are additional weights that help the model balance between positive and negative samples (helps with unbalanced datasets).
- **Focal loss gamma**: reduces weight for easier examples (well classified) and focuses on harder and misclassified examples.
- **IoU threshold**: which defines the minimum IoU value to consider a prediction correct.
- **Anchor matching threshold**: defines the minimum IoU value to consider an anchor box as a positive sample.

The final group is the augmentation related hyps, data augmentation is a technique that is particularly useful for smaller dataset and is used to artificially increase the number of training samples by applying random transformation defined by the following parameters::

- **HSV hue, saturation, value**: these are the values used to randomly change the hue, saturation, and value of images during the training process.
- **Degrees**: random rotation angle for the images (in degrees).
- **Translate**: vertical or horizontal translation (in pixels) for the images.
- **Scale**: scaling factor for images (zoom-in or zoom-out).
- **Shear**: angular deformation applied to the images.
- **Perspective**: perspective distortion (simulates tridimensional view) for the images.
- **Flip up, down, left, right**: random flipping of the images.
- **Mosaic**: activates mosaic augmentation that combines multiple images into one.
- **Mixup** activates mixup augmentation which merges two images and their labels.
- **Copy Paste**: randomly pastes objects from one image to another to improve randomness and, therefore, the robustness of the model.

4.1.2 Optimizer

The **optimizer** is the algorithm that updates the parameters during the training process. The three different optimizers tested for this thesis work are **SGD** (Stochastic Gradient Descent), **Adam** and **AdamW**, these last two are extensions of the SGD algorithm. SGD is the default optimizer for Yolo architectures and, as cited by [24] SGD calculates the gradients of the cost function with respect to the editable parameters of the network, its goal is to minimize the objective function (error or cost) that the model wants to reduce during training. These parameters are updated in the opposite direction to the gradient of the function. As seen before, the learning rate (LR) controls the step size taken towards a local minimum. In summary, SGD seeks to find the value of the parameters that minimize the objective function. The equations are reported below to determine how it works.

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta) \tag{4.1}$$

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x_i, y_i) \tag{4.2}$$

Where θ are the *model parameters*, η is the *LR multiplied by the gradient* ∇, x_i is the parameter update rule for each training example and y_i is its corresponding label. Adam and AdamW implement **adaptive moment estimation**, both are used to adjust the weights like SGD but these extensions calculate the running average of the gradients and the squared gradients, which represent the first and second moments

of the gradient and use a rule to update the parameters. AdamW introduces the weight regularization through the previously cited *weight decay* which tends to a faster convergence. The equations for these extensions are reported below.

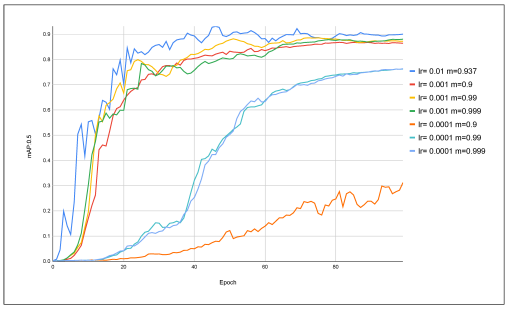
$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{4.3}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{4.4}$$

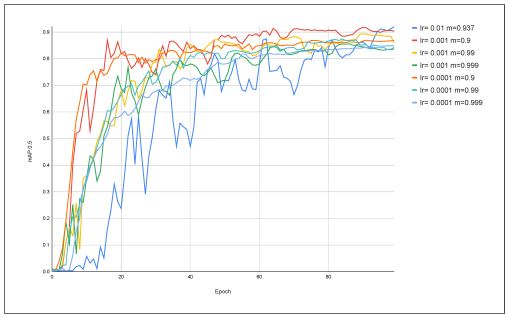
$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \cdot \hat{m}_t \tag{4.5}$$

Here, m_t is the *average of the gradients*, and v_t is the *average of the squared gradients*. β_1 and β_2 control the decay rate of the first and second moment estimates while g_t is the *gradient* for the current batch, the latter equation is the Adam rule to calculate the cost function.

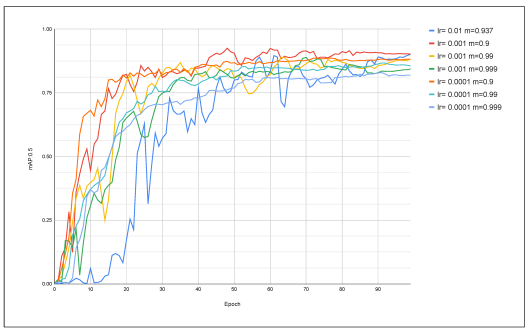
Figure 4.1 shows how these different optimizers affect mAP with several learning rates.



(a) SGD mAP/epochs.



(b) Adam mAP/epochs.



(c) AdamW mAP/epochs.

Figure 4.1: Different learning rates and corresponding mAP0.5 for each epochs for YoloV5 with the three optimizers [24].

4.1.3 Scheduler

The scheduler is a mechanism that changes the value of the LR during the training process. It is a key component because it helps both during the initial phase (a higher LR helps the model to explore different parameters with higher speed) and during the final phase (scaling it down when the function is near the local minimum

and preventing the model from "skipping" that value). Fine-tuning the LR can help the model *converge faster* and *generalize better*.

At each epoch, the scheduler sets the LR value, and the choice is based on a certain function of the scheduler. The value is then passed to the optimizer, which uses the LR to update the weights. There are several different scheduling functions. The ones that are mainly by Yolo architectures are **StepLR**, **LambdaLR** and **CosineAnnealingLR**.

StepLR, according to [25], decays the learning rate of each parameter group by γ every `step_size` epochs. It is possible for this decay to happen concurrently with learning rate adjustments made independently of this scheduler. When `last_epoch` = -1, the initial learning rate is set to `lr`.

$$\mathrm{lr}_{\mathrm{epoch}} = \begin{cases} \gamma \cdot \mathrm{lr}_{\mathrm{epoch}-1}, & \text{if epoch mod step\_size} = 0 \\ \mathrm{lr}_{\mathrm{epoch}-1}, & \text{otherwise} \end{cases}$$

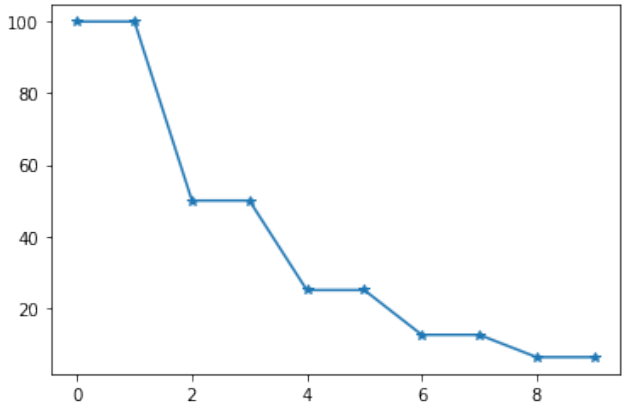


Figure 4.2: StepLR [25].

This is the default LR scheduler, but for this thesis work the ones used are the **LambdaLR** and **CosineAnnealingLR**

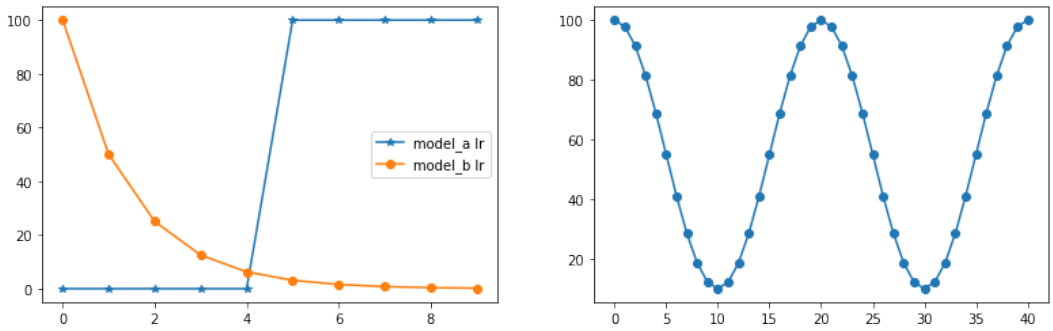
Lambda sets the learning rate of each parameter group to the initial learning rate multiplied by a given function λ. When `last_epoch` $= -1$, sets the initial learning rate at `lr`.

$$\mathrm{lr}_{\mathrm{epoch}} = \mathrm{lr}_{\mathrm{initial}} \cdot \lambda(\mathrm{epoch})$$

while *Cosine Annealing* instead sets the learning rate of each parameter group using a cosine annealing schedule which follows a cosine wave. [25] points out that, as stated for the previous scheduler, because the schedule is defined recursively, the learning rate can be simultaneously modified outside of this scheduler by other operators.

Also, if the learning rate is set solely by this scheduler, the learning rate at each step becomes

$$\eta_t = \eta_{\min} + \frac{1}{2}(\eta_{\max} - \eta_{\min})\left(1 + \cos\left(\frac{T_{\text{cur}}}{T_{\max}}\pi\right)\right)$$



(a) Lambda LR. **(b)** Cosine Annealing LR.

Figure 4.3: Lambda and Cosine Annealing schedulers [25].

There is also a different version called Cosine Annealing with **Warm Restarts** that differ because restarts after T_i epochs

$$\eta_t = \eta_{\min} + \frac{1}{2}(\eta_{\max} - \eta_{\min})\left(1 + \cos\left(\frac{T_{\text{cur}}}{T_i}\pi\right)\right)$$

With the corresponding graph:

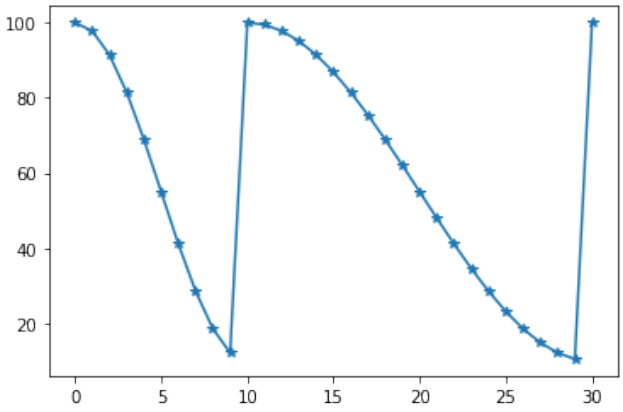


Figure 4.4: Cosine Annealing with warm restarts [25].

For the scope of this thesis work other schedulers have not been taken into account nor studied, these ones are the most relevant for the training of the models that were the object of this project.

4.2 Training process

Now that the pre-process has been done and all the relevant actors have been explained, it is time to consider the training process itself and how it happens. The whole process revolves around three main scripts that will be described in depth

with coding segments in A but the pipeline and the main goal will be discussed and explained.

These three scripts are:

- `train.py` which is the core of the training process.
- `val.py` at the end of each epoch this script is used to validate the weights and report the metrics of the model at that stage.
- `detect.py` detects objects in the images, is used by `val.py` to predict the bounding boxes.

The training file is divided into several phases. It accepts different arguments to configure the training (the `.yaml` files that are used to load the dataset on which the training must occur, the weights that have to be used to train the model, the batch size of each training epoch and others) and first sets up the device that are going to be used, the directory in which the results are going to be saved and then build the model (either from zero or from a *checkpoint* if specified).

Then it sets up the optimizer (which is specified by the `argparse` argument) and the scheduler (through pytorch library). After everything is set the training starts and the loss, LR and metrics are updated after each epoch. At the end of each epoch the `val.py` file is run for validation and saves the best model (`.pt` version) and the last one (in case the training is stopped, so that it can be resumed with the last weights of the training).

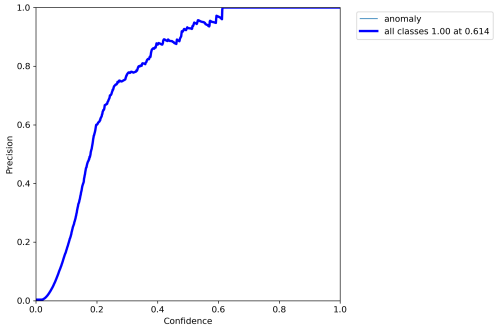
Here is when the metrics are calculated and a corresponding `.csv` file is updated with losses for **object**, **class** and **bounding box**, also **Precision**, **Recall**, **mAP0.5** and **mAP0.5-95** are calculated and wrote in the report.

At the end of the training it is possible to see and analyze the graphs with precision, recall and PR curve, also the **F1 curve** is plotted, which is defined as the harmonic mean between precision and recall and how it varies with *different confidence* values is calculated as $F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$ and can help determine the best confidence value for the model.

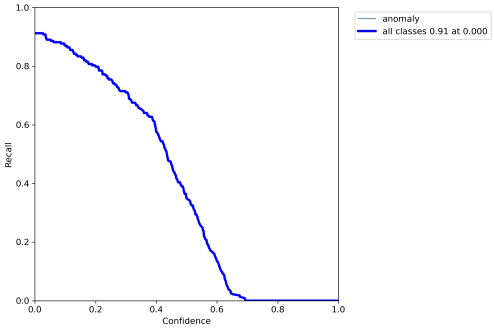
For the training of the first model the script was executed with the following arguments:

- Configuration of the yolo model explained in section 4.1.
- `.yaml` file that specified the path to the dataset.
- Hyps defined by the `hyps.yaml` file (the ones discussed in section **3.2.1**, will be discussed in A.
- Image size of 1024x1024 to help the first training (will be scaled afterwards) since previous trainings highlighted the difficulty of this model to be trained on 640x640.
- Batch size of 32 images per batch.

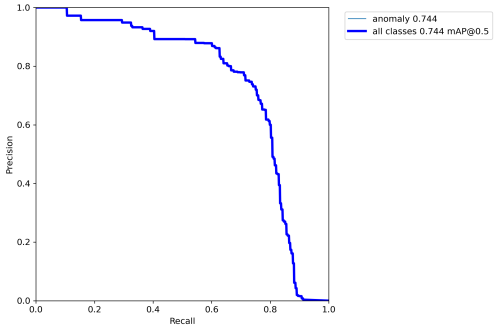
This training also used Lambda LR scheduler and SGD optimizer.
The results of this training are reported below with different graphs



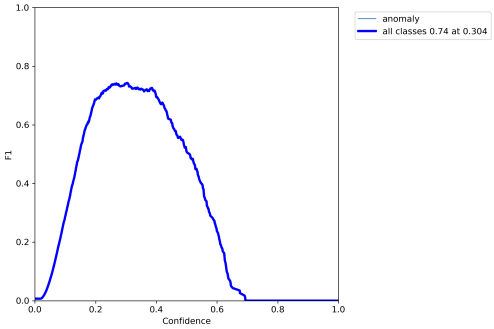
(a) Precision curve.



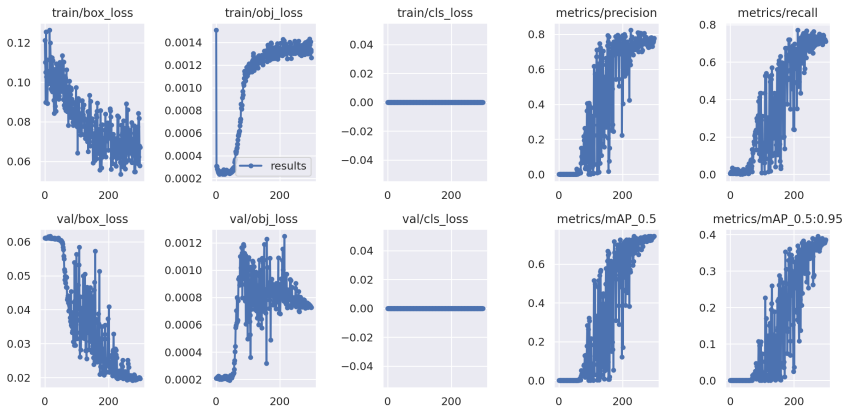
(b) Recall curve.



(c) Precision/Recall curve.



(d) F1 curve.



(e) Results of loss and metrics.

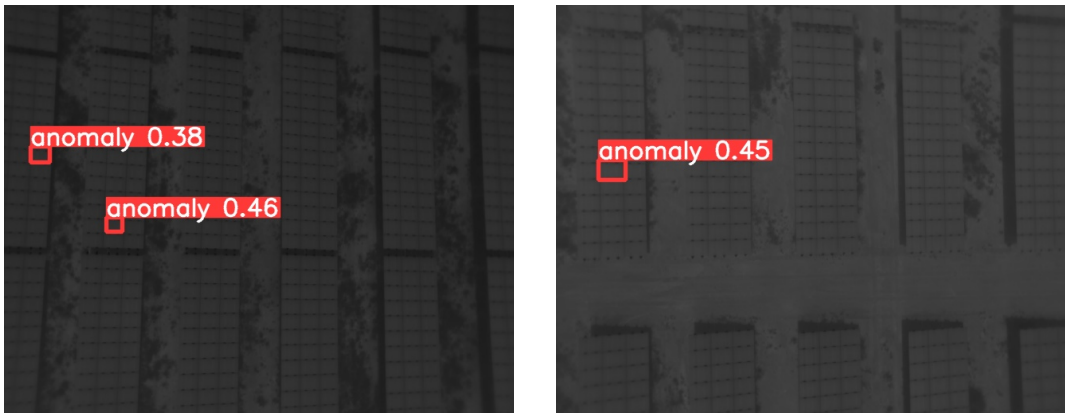
Figure 4.5: Graphs related to the Yolov3 train.

As can be seen in 4.5, image **e** The model suffered a lot of setbacks during training, several epochs resulted in very low results. Although these setbacks were an indication of a difficult training, the results were good.

Precision	Recall	mAP_0.5	mAP_0.5:0.95	box_loss	obj_loss
0.7784	0.71053	0.74411	0.38598	0.019679	0.0007277

Table 4.1: Final results for the last epoch of the Yolov3 FP32 model.

In table 4.1 the class loss has been omitted since the model has been trained on a single class, resulting in a trivial value of 0 for the class loss. After training this model the `detect.py` script has been run to see how well the model performed (visually) on the validation set.



(a) Multiple detections. **(b)** Single detection.

Figure 4.6: Detections with confidence value reported.

The results are generally good, but the quantization process that was tried (substitution of the `Conv` layers with `QuantConv`) was not successful and the approach had to be changed.

Chapter 5

Vitis AI and Yolov5

The failed quantization attempts resulted in a deep search for other solutions and therefore the use of a different framework (Vitis AI). This framework allowed the training to be based on a more powerful model (Yolov5) more precisely the models **Yolov5n** and **Yolov5s**.

Vitis AI (VAI) offers different tools for AI hardware acceleration, taking every step of the pipeline into account and providing compilers, optimizers (quantization tools, the main reason for the usage of this framework), profilers, and libraries for the implementation of quantized models on hardware devices.

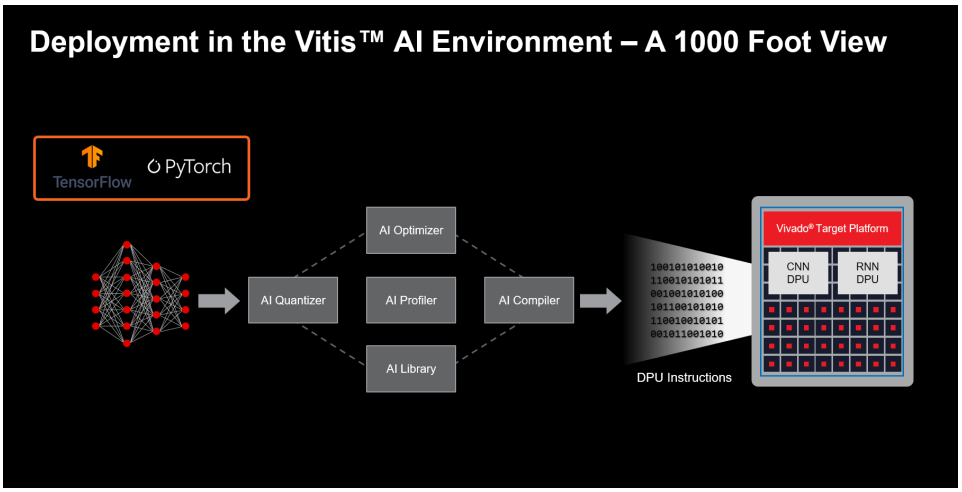


Figure 5.1: Vitis AI tools [26].

This chapter will discuss both the integration of VAI into this thesis and the full quantization process, highlighting the differences between each format obtained and the optimization techniques applied to the models. Theoretical concepts will be introduced when necessary along with a real-world application of them.

5.1 Processing the output tensor

The first part (which will be skipped due to the trivial nature of the task) was building the docker image of VAI in order to be ready for the usage of the framework. Then

the most important part for successfully training our model was the post-processing of the raw output, which had to be done for the sake of the predictions' results.

The model generates predictions for each cell of the feature map at different scales, using a set of pre-defined anchors. The post-processing process is used to transform these normalized predictions into absolute bounding box coordinates in the image domain.

The postprocessing implementation, for which the code is written in A, was done following these steps:

- **Tensor reshaping**: the raw output tensor is reshaped and permuted to separate anchors, spatial dimensions, and prediction components (`x`, `y`, `w`, `h`, objectness, class scores).
- **Reference grid construction**: a grid of (`x`, `y`) coordinates is created for each feature map cell and used to map relative predictions to the image space.
- **Coordinate decoding**: a sigmoid activation is applied to the `xy` offsets, objectness, and class scores; then `xy` is scaled and shifted using the grid and stride to obtain absolute positions.
- **Anchor combination**: anchor box dimensions are scaled by stride and combined with the predicted width and height using a non-linear transformation $(\sigma(\cdot) \cdot 2)^2$.
- **Final output**: a bounding box tensor $[\texttt{x}, \texttt{y}, \texttt{w}, \texttt{h}, \texttt{confidence}, \texttt{class scores}]$ is produced where all components are **concatenated** into a final tensor and **flattened** across all positions and anchors.

This is consistent with [12] where the predicted coordinates (x, y, w, h) are decoded with respect to the cell and associated anchor, using sigmoid functions for the offset (x, y) and exponentials for the dimensions (w, h).

Now the output is a tuple made by two elements, the first element is the flattened output tensor, which has the shape (B, N, C) where

- **B** is the *batch size*.
- **N** is the *total number of predictions* (sum on all scales of **number of anchors** $\times$ *height* $\times$ *width*).
- **C** is the number of channels, which is made of the number of classes plus 5 (the parameters the parameters `x, y, w, h` and `objectness`).

The second element is a list of three tensors, one for each resolution of the feature map, each with shape $(B, 3, H, W, \texttt{nc} + 5)$. The values are as before (B is the batch, nc is the number of classes and H and W are the height and width) while 3 is the *number of anchors per spatial location.* And for each feature map resolution (H $\times$ W), the sizes are:

- 80×80 for the highest resolution scale (stride 8).
- 40×40 for the medium resolution scale (stride 16).
- 20×20 for the lowest resolution scale (stride 32).

Another important step was to change the rectifier units for yolov5, which included **SiLU** layers, and substitute them with **LeakyReLU**, because VAI does not support SiLU and of all supported layers, [27] describes LeakyReLU (*with a slope of 26/256*) as the activation function that gives the better result.

Leaky ReLU is lighter and is better suited for embedded applications. The equation for the Leaky ReLU is the following one:

$$\text{LeakyReLU}(x) = \begin{cases} x, & \text{se } x \geq 0 \\ \alpha x, & \text{se } x < 0 \end{cases}$$

The last thing implemented in the post processing function was the cut of the last layer in the detection head and the implementation of such layer in the post processing function. After this step, everything was ready for the Yolov5n training.

5.1.1 Training the Yolov5 model

The training of the Yolov5n model on the fotovolt dataset used a different backbone and head from the Yolov3 model used in the previous chapter. The Yolov3 used `Conv` and `MaxPool` layers for the backbone while the Yolov5 implemented `Conv` and `C3` layers, with a final `SPPF` layer before the head.

The `C3` layer is essentially three `Conv` layers with a `Bottleneck` and a `Concat` layers, this layer helps the **gradient** propagation and reduces parameters.

The `SPPF` is a `Conv` layer with 3 `MaxPool` layers, all jointed by a `Concat` layer, used to help receptive ability and thus allow recognition of objects of different sizes.

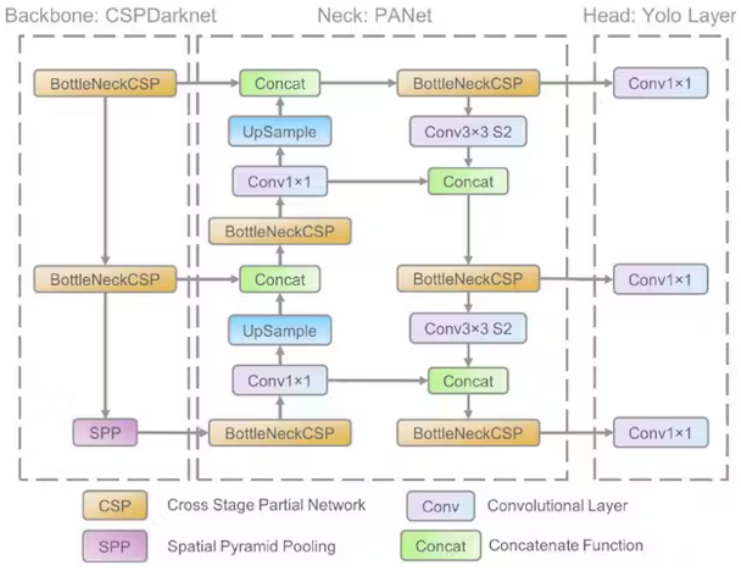


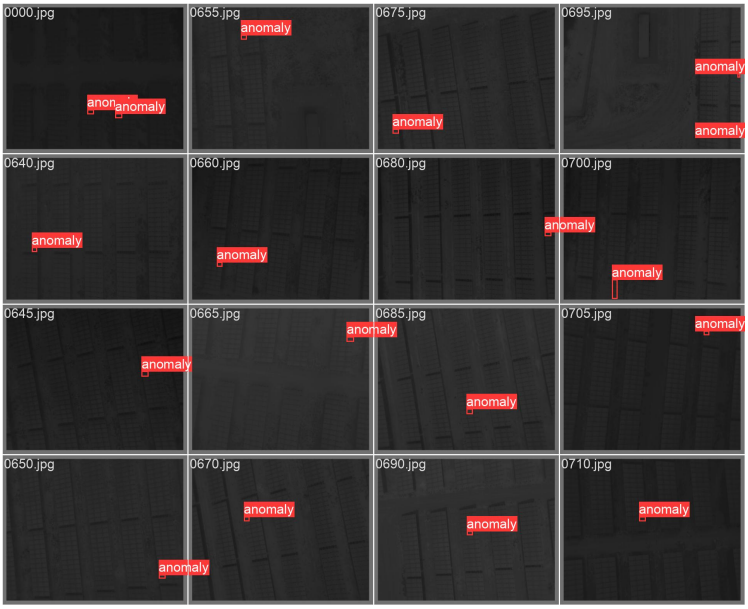
Figure 5.2: YOLOv5n structure [28].

The image 5.2 also shows how the head (called neck in the image) is structured. The training of this model gave far better results in the `.pt` format weights.

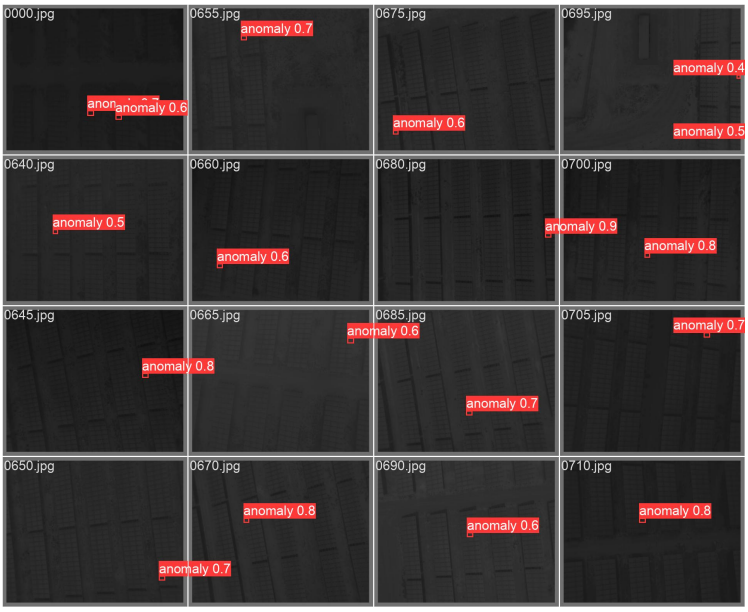
Precision	Recall	mAP_0.5	mAP_0.5:0.95	box_loss	obj_loss
0.82126	0.80263	0.81621	0.47899	0.034487	0.0066194

Table 5.1: Best results for Yolov5n on the 640x640 fotovolt dataset.

These results can be compared with the 4.1 and a neat improvement can be seen. Precision has a +5% improvement, recall and mAP0.5-95 have an astounding +9% and mAP0.5 has a +7% boost as well. The validation batch is the following:



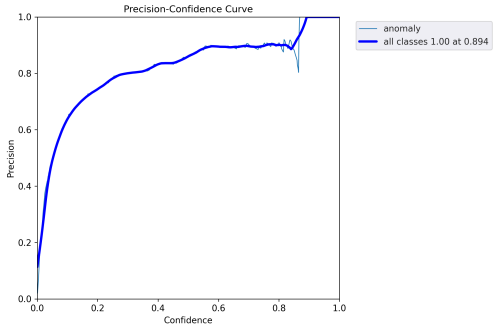
(a) Labels for the batch.



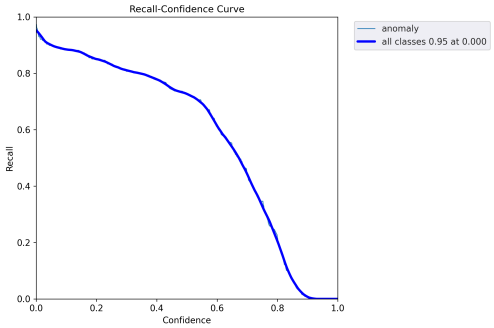
(b) Predictions for the batch.

Figure 5.3: Comparison of labels and predictions for the validation batch.

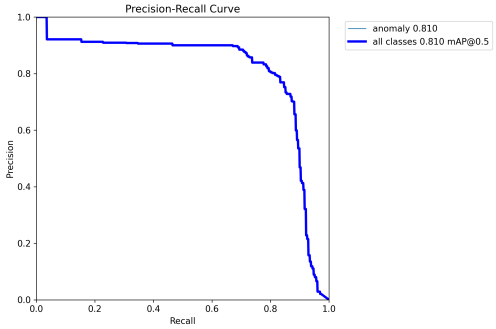
Figure 5.3 shows that the predictions are mostly correct and the confidence score is higher than the ones of 4.6. Figure 5.4 shows the results of the training.



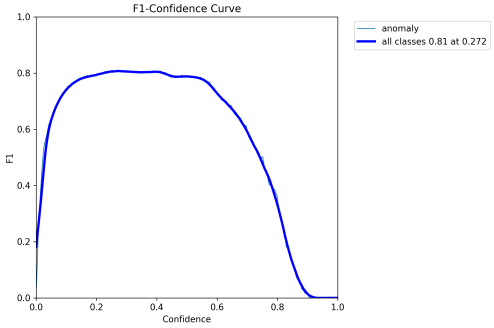
(a) Precision curve.



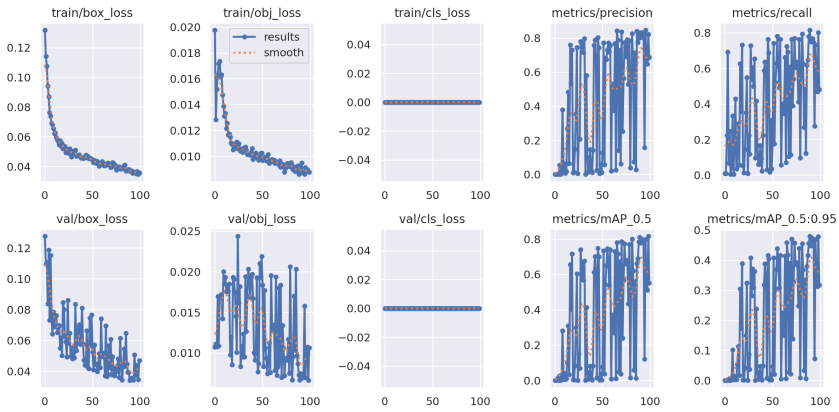
(b) Recall curve.



(c) Precision/Recall curve.



(d) F1 curve.



(e) Results of loss and metrics.

Figure 5.4: Graphs related to the Yolov5 train.

Although the metrics show many epochs with low quality results, the overall training was successful and the results excelled much more than expected, resulting in a strong model for this step of the project.

5.1.2 Export and validation of the ONNX model

The model performed well and the work now revolved around the export of the `.pt` weights to the `.onnx` format, which is the target format for the quantization. VAI offers a wide range of tools but the main tools adopted for this work (VAI Quantizer mainly) is based on the ONNX format. This format (Open Neural Network Exchange) offers way more portability possibilities and is well supported by many acceleration tools. ONNX is also more reliable for quantization, pruning and optimization.

So after obtaining the version needed, another validation process was run for completeness and safety.

Precision	Recall	mAP_0.5	mAP_0.5:0.95
0.825	0.785	0.809	0.479

Table 5.2: Validation of the ONNX format.

(a) Labels of the image. **(b)** Predictions of the ONNX model.

Figure 5.5: Validation batch for ONNX model.

Validation batch, for non-PyTorch models if forced to one, but running detections on the dataset can show all the predictions made, this validation run was made with the **Confidence threshold** not set on the optimal value proposed by the F1 curve, by doing so the results of the validation see an improvement of **+4%** of **mAP0.5-95**.

5.2 Quantizing the model

Before talking about the quantization function and the way it is implemented by VAI, it is better to define a key concept which is the differences between **dynamic** and **static** quantization. As said in 2.3, the quantization goal is to compress the FP32 model to an INT8 model, to do so, the values have to be *clipped*, so the values of the FP32 interval have to be mapped to INT8 range, leading to a problem where there is to be a *clipping range* defined as the $[\min, \max]$ interval where values less than min

are set to min and values greater than max are set to max. This value is derived from the formula expressed in 2.3.3. In [29], **Dynamic quantization** is described as a technique that dynamically calculates this range for each activation map, and this usually results in higher accuracy but also higher overhead. **Static quantization** is done with the aid of a calibration dataset to define the typical clipping range of the activation maps and keep that range fixed. This does not add computational overhead but offers lower accuracy.

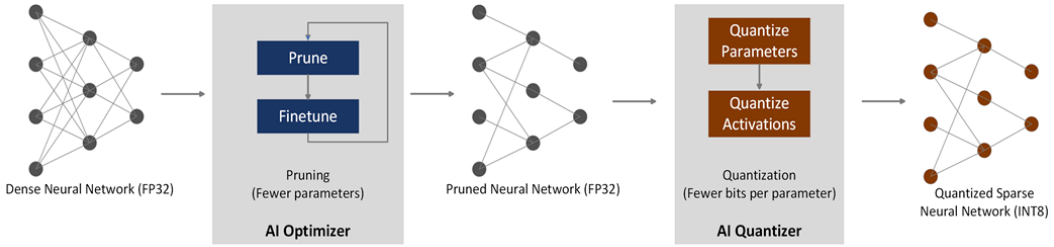


Figure 5.6: Tools for the quantization process [30].

VAI offers a static quantization function, so the script was divided in 2 steps: The first is the loading of a **Calibration Dataset** which is best to be a subset of the training or validation dataset (around 100-1000 images in order to successfully calibrate the quantized model on the target dataset). The second step is the quantization function itself. This quantization function is `vai_q_onnx.quantize_static()` which takes different parameters, from the model input to the calibration data. The main focus will be on the format, the calibration method, and the extra options.

The *format* is **QDQ**, which stands for **Quantize/DeQuantize** and is a technique that simulates the quantization on the FP32 model by adding a `QuantizeLinear` layer (where is needed) and simulates the weights and activations on INT8, the a `DeQuantizeLinear` is added right after this layer and the weights and activations are converted back to FP32 [31].

The *calibration method* is `PowerOfTwoMethod.MinMSE` which is a function of VAI that imposes the power of two scaling factors for the model, these power of two scaling factors are calculated with a Minimum Mean Squared Error. The extra options are some optimizations set for the power of two restriction and will be discussed in B. The idea is depicted in the 5.8 image.

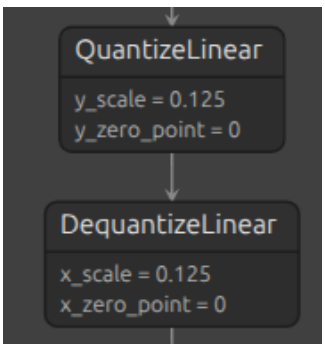


Figure 5.7: QDQ layers with power of two scaling factors.

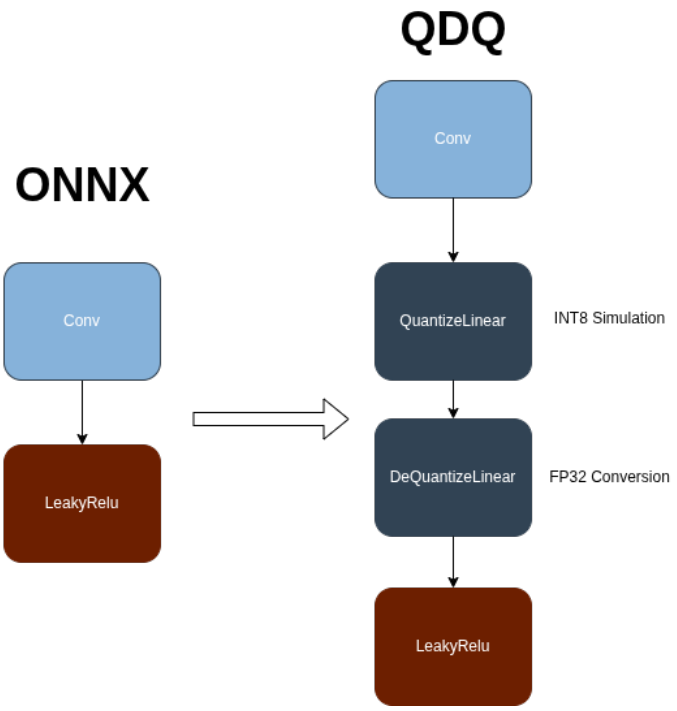


Figure 5.8: ONNX to QDQ process.

5.2.1 QONNX and the final conversion

The result of the ONNX to QDQ process allowed us to have models that could then be safely converted to the quantized version. This was possible with the aid of QONNX, a library that has many different useful functions for the scope of this thesis, but among all those the ones that have been more relevant are the *cleaning* and the *conversion* of the model.

The **clean** is an important transformation that needs to be run before the **conversion** since it helps with the following inefficiencies:

- **Removes unused nodes:** Eliminates operations or tensors that do not contribute to the final output of the model (dead nodes).
- **Deletes redundant QDQ pairs:** If excessive or unnecessary quantization/dequantization nodes were introduced (e.g., duplicates or adjacent inverse pairs), these are removed to streamline the model.
- **Simplifies subgraphs:** Optimizes sequences of operations that can be merged or reordered for efficiency, such as consecutive dequantization followed by quantization layers.
- **Cleans graph metadata:** Updates tensor names, graph links, and node metadata to reflect the new structure and remove obsolete references.

After the cleaning transformation, the model is almost ready for the final conversion, but some precautions have to be taken to convert the QDQ layers for future hardware implementations.

One key concept is the homogeneity of the scaling factors that are put in input to the different layers and also between input and output layers. The first conversion done did not take care of this step and generated this kind of layers.

(a) Different scaling factors between the Concat layer's inputs.

(b) Different scaling factors between the Add layer's inputs.

Figure 5.9: Add and Concat different scaling factors.

Figure 5.9 shows that the **Add** layer also has different scaling factors between the input layers and the output layer, this is a problem that also occurs for the **Leaky ReLU** as shows the example of 5.10

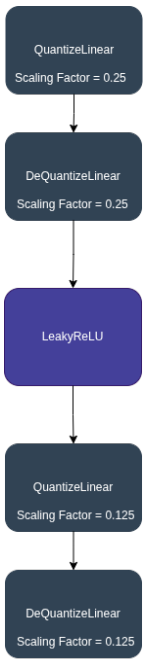


Figure 5.10: Different scaling factor between input and output layers for the Leaky ReLU.

The script written, which will be analyzed in B, made sure to standardize the scaling factor between the input layers of every **Add** and **Concat** layer and also between their input and output layers.

For this task, the script recursively travels the graph and searches for these layers, when found walks back from the layer to its input layers and searches for *QDQ* chains and remembers the *scaling factor of these chains*, then looks for the output of the layer and checks whether the scaling factor is compliant with the input and changes it, then goes on to the next layer, for this model, **5 cycles** of iteration on the graph were enough to standardize the scaling factor.

The model is now optimized and ready for conversion where the layers will not just be INT8 simulation but will be treated as INT8, in 6 the results will be shown to see how different scaling factors impact the model validation results.

The **convert** function of the QONNX library has the following features for the conversion of the model:

- **Removes** `QuantizeLinear` and `DequantizeLinear` nodes where possible.
- **Fuses quantized operations** (e.g., convolution + bias + ReLU) into single, more efficient nodes.
- Transforms the model into a **compact format**, where the weights and activations are truly handled as INT8 values.
- Produces a **deployable model**, closer to the final inference representation used on the target hardware.

Resulting in the following model structure (simplified):

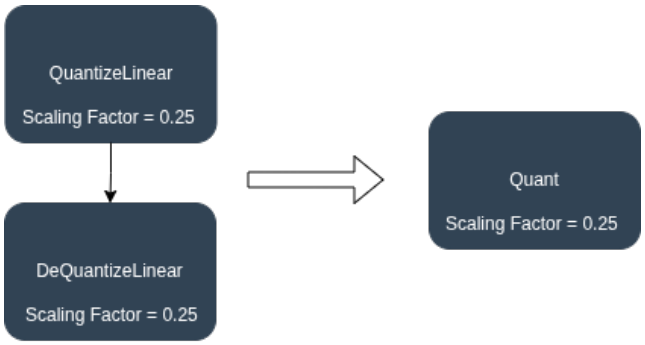


Figure 5.11: Conversion of the QDQ layers to Quant.

When a quantized neural network model is transformed from the QDQ (Quantize-DeQuantize) representation to the operator-level quantized form, the **QuantizeLinear** and **DeQuantizeLinear** nodes are replaced with a single quantized operator. This transformation brings several structural and computational benefits. The fused quantized operator:

- Operates directly on **integer tensors**, usually in the `int8` or `uint8` format.

- Includes **scale** and **zero-point** values as *operator attributes*, avoiding the need for separate quantization nodes.
- Executes all computations entirely in the **quantized domain**, without converting back to FP32 at run-time.

This approach enables more efficient inference by reducing graph complexity and allowing hardware accelerators to leverage native support for integer arithmetic.

Chapter 6

Validation and results

This chapter will focus on the results obtained by the different versions of the Yolov5n models while also considering the datasets described in 3.2 and 3.3 along with the 3.1 dataset which has been the target dataset for the first training discussed in 4 for completeness and a deeper analysis. On the **ASXL** dataset, in addition to the v5n model, a v5s version has been trained to check the different results obtained. In addition, a model trained on the larger and more varied COCO dataset has been tested, creating 3 different models with different **hyps** values, **optimizers** and **scheduler**. These models have been trained for 640×640, 512×512 and 416×416 scales, and the final results will be shown and discussed.

6.1 PV thermal images dataset results

The results on this dataset are reported below, considering the different image sizes and confidence values for the detection, here are reported both the **FP32** after the conversion from `.pt` to `.onnx` and **INT8** with uniformed scaling factors versions. The proposed results first show how the models changed with quantization andand then the results will be compared.

6.1.1 Version 416X416

Table 6.1: Evaluation metrics for the 416×416 FP32 (ONNX version).

Conf	IoU	Precision	Recall	mAP@0.5	mAP@0.5:0.95
0.001	0.5	0.828	0.781	0.784	0.449
0.001	0.6	0.828	0.781	0.782	0.448
0.1	0.5	0.828	0.781	0.817	0.497
0.1	0.6	0.828	0.781	0.817	0.497
0.2	0.5	0.828	0.781	0.818	0.502
0.2	0.6	0.828	0.781	0.818	0.502
0.3	0.5	0.860	0.753	0.809	0.498
0.3	0.6	0.860	0.753	0.809	0.498
0.4	0.5	0.880	0.706	0.790	0.491
0.4	0.6	0.880	0.706	0.790	0.491

The metrics show that this model performs really well despite the small dimensions with a value of mAP around 50%, which is really good.

Table 6.2: Evaluation metrics for the 416×416 QDQ post scaling factor uniformization.

Conf	IoU	Precision	Recall	mAP@0.5	mAP@0.5:0.95
0.001	0.5	0.812	0.706	0.725	0.361
0.001	0.6	0.804	0.706	0.720	0.360
0.1	0.5	0.784	0.719	0.764	0.412
0.1	0.6	0.777	0.719	0.762	0.411
0.2	0.5	0.850	0.645	0.738	0.405
0.2	0.6	0.845	0.645	0.737	0.404
0.3	0.5	0.874	0.456	0.662	0.368
0.3	0.6	0.874	0.456	0.662	0.368
0.4	0.5	0.899	0.311	0.605	0.344
0.4	0.6	0.899	0.311	0.605	0.344

Quantization impacted the model but with the correct configuration the **mAP** value is around **40%**, which is still high. The model is balanced with all the other metrics well above **70%**.

6.1.2 Version 512X512

Table 6.3: Evaluation metrics for the 512×512 FP32 (ONNX version).

Conf	IoU	Precision	Recall	mAP@0.5	mAP@0.5:0.95
0.001	0.5	0.791	0.763	0.775	0.465
0.001	0.6	0.791	0.763	0.774	0.465
0.1	0.5	0.791	0.763	0.800	0.506
0.1	0.6	0.791	0.763	0.800	0.506
0.2	0.5	0.794	0.759	0.797	0.509
0.2	0.6	0.794	0.759	0.797	0.509
0.3	0.5	0.828	0.697	0.777	0.495
0.3	0.6	0.828	0.697	0.777	0.495
0.4	0.5	0.877	0.623	0.753	0.484
0.4	0.6	0.877	0.623	0.753	0.484

Table 6.4: Evaluation metrics for the 512×512 QDQ post scaling factor uniformization.

Conf	IoU	Precision	Recall	mAP@0.5	mAP@0.5:0.95
0.001	0.5	0.793	0.789	0.762	0.422
0.001	0.6	0.793	0.789	0.760	0.422
0.1	0.5	0.793	0.789	0.798	0.473
0.1	0.6	0.793	0.789	0.797	0.472
0.2	0.5	0.796	0.789	0.791	0.472
0.2	0.6	0.793	0.789	0.791	0.471
0.3	0.5	0.842	0.697	0.758	0.453
0.3	0.6	0.842	0.697	0.758	0.453
0.4	0.5	0.856	0.654	0.742	0.447
0.4	0.6	0.856	0.654	0.742	0.447

The impact of quantization on this model is less relevant, with all the metrics around 80% and the mAP of 47% with the best configurations (due to the correct F1-score computed during training).

6.1.3 Version 640X640

Table 6.5: Evaluation metrics for the 640×640 FP32 (ONNX version).

Conf	IoU	Precision	Recall	mAP@0.5	mAP@0.5:0.95
0.001	0.5	0.825	0.785	0.810	0.477
0.001	0.6	0.825	0.785	0.809	0.479
0.1	0.5	0.825	0.785	0.822	0.512
0.1	0.6	0.825	0.785	0.822	0.512
0.2	0.5	0.825	0.785	0.816	0.516
0.2	0.6	0.825	0.785	0.816	0.516
0.3	0.5	0.825	0.785	0.803	0.511
0.3	0.6	0.825	0.785	0.803	0.511
0.4	0.5	0.829	0.785	0.802	0.513
0.4	0.6	0.829	0.785	0.802	0.513

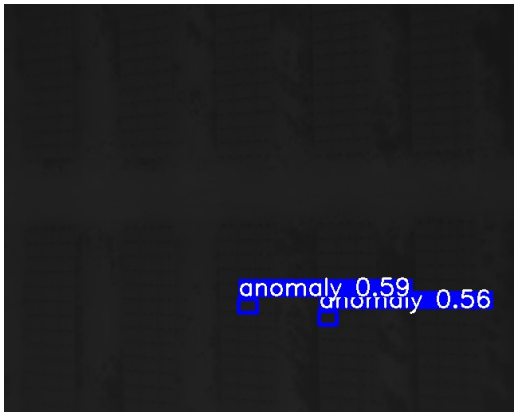
This is the best model among the proposed ones. The mAP is 51% which is a solid value.

Table 6.6: Evaluation metrics for the 640×640 QDQ post scaling factor uniformization.

Conf	IoU	Precision	Recall	mAP@0.5	mAP@0.5:0.95
0.001	0.5	0.801	0.758	0.775	0.452
0.001	0.6	0.801	0.758	0.774	0.453
0.1	0.5	0.801	0.758	0.787	0.488
0.1	0.6	0.801	0.758	0.787	0.488
0.2	0.5	0.801	0.758	0.783	0.491
0.2	0.6	0.801	0.758	0.783	0.491
0.3	0.5	0.816	0.737	0.771	0.487
0.3	0.6	0.816	0.737	0.771	0.487
0.4	0.5	0.838	0.702	0.759	0.481
0.4	0.6	0.838	0.702	0.759	0.481

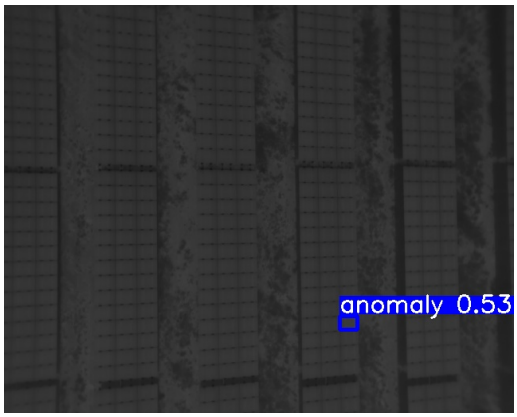
Quantization impacted less than the other versions, with values around **50%** for **mAP** and the other metrics all above 75%, with **precision** still higher than **80%** and **recall** not going lower than **75%** with the best configuration (Confidence set at 0.2).

Below are two reported examples for the model with uniform scaling factor.

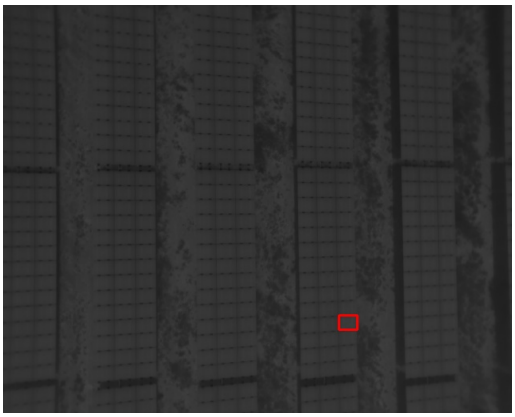


(a) Detection sample 1.

(b) Ground truth sample 1.



(c) Detection sample 2.



(d) Ground truth sample 2.

Figure 6.1: Comparison between detections and ground truth of the Uniscaled versions for the 640×640.

6.1.4 Comparison between ONNX FP32 and INT8

The following table will highlight the comparison between the best model for each size, considering the **FP32 ONNX** version and the **INT8 with uniform scaling factor**.

Table 6.7: Comparison of model sizes and performance (mAP@0.5:0.95) in FP32 and INT8 formats.

Size	Conf-value	FP32	INT8	Δ
416	0.1	0.502	0.412	-0.090
512	0.2	0.509	0.472	-0.037
640	0.2	0.516	0.491	-0.025

A key factor is how much the image size degrades the model between *pre* and *post* **QDQ** process, the 640×640 shows an acceptable loss on the mAP, while the 416×416 loses almost 10% of mAP, staying well above the 30% threshold but lowering the accuracy of the model much more. The loss of the 512×512 model is a good trade-off if the model size is a key factor for the target implementation, with both a good

accuracy and a reduced size compared to the 640×640 version.

6.2 ASXL Dataset

This dataset had two main versions, both initially trained on 1024×1024, then exported as 640×640 in the ONNX version and quantized, one with the YOLOv5n and one with the YOLOv5s models. The results of the 1024×1024 version, 640×640 and quantized versions are reported and commented on below.

6.2.1 YOLOv5n version

The models trained with the **v5n** weights had the *hyps* as specified in A, and below the different versions can be seen, the first is the FP32 1024×1024 model.

Table 6.8: Evaluation metrics for the 1024×1024 FP32 `.pt` version of the YOLOv5n.

Conf	IoU	Precision	Recall	mAP@0.5	mAP@0.5:0.95
0.001	0.5	0.807	0.357	0.426	0.250
0.001	0.6	0.797	0.357	0.419	0.248
0.1	0.5	0.807	0.357	0.505	0.324
0.1	0.6	0.797	0.357	0.492	0.315
0.2	0.5	0.807	0.357	0.548	0.361
0.2	0.6	0.797	0.357	0.533	0.350
0.3	0.5	0.807	0.357	0.564	0.382
0.3	0.6	0.797	0.357	0.559	0.379
0.4	0.5	0.808	0.358	0.586	0.408
0.4	0.6	0.797	0.358	0.583	0.405

These results have to take into account that the low values for the recall are due to the different size of the defects between the full-scale image and the segment, some cracks were smaller than 3 pixels, and this affected the model because the model did not find small defects.

Table 6.9: Evaluation metrics for the 640×640 FP32 `.ONNX` version of the YOLOv5n.

Conf	IoU	Precision	Recall	mAP@0.5	mAP@0.5:0.95
0.001	0.5	0.677	0.335	0.360	0.198
0.001	0.6	0.667	0.341	0.356	0.198
0.1	0.5	0.677	0.335	0.447	0.259
0.1	0.6	0.667	0.341	0.430	0.252
0.2	0.5	0.677	0.335	0.482	0.291
0.2	0.6	0.667	0.341	0.470	0.282
0.3	0.5	0.677	0.335	0.495	0.308
0.3	0.6	0.667	0.341	0.487	0.300
0.4	0.5	0.667	0.341	0.523	0.327
0.4	0.6	0.646	0.347	0.516	0.320

The resize of the model affected the accuracy and in 6.2.3 the comparison between the two different best models is shown. This is also caused by the very small defects in the images.

Table 6.10: Evaluation metrics for the 640×640 `QDQ` version of the YOLOv5n.

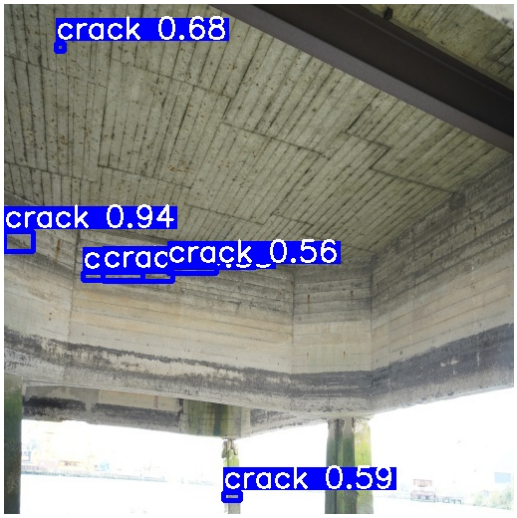
Conf	IoU	Precision	Recall	mAP@0.5	mAP@0.5:0.95
0.001	0.5	0.649	0.312	0.335	0.167
0.001	0.6	0.627	0.312	0.329	0.166
0.1	0.5	0.649	0.312	0.428	0.232
0.1	0.6	0.627	0.312	0.413	0.223
0.2	0.5	0.649	0.312	0.463	0.255
0.2	0.6	0.627	0.312	0.443	0.247
0.3	0.5	0.629	0.318	0.490	0.278
0.3	0.6	0.627	0.312	0.471	0.270
0.4	0.5	0.688	0.301	0.507	0.291
0.4	0.6	0.662	0.301	0.497	0.286

The **QDQ** step did not affect the model too much, but a loss of about 3% mAP is evident.

Table 6.11: Evaluation metrics for the 640×640 `QDQ` with uniform scaling factor version of the YOLOv5n.

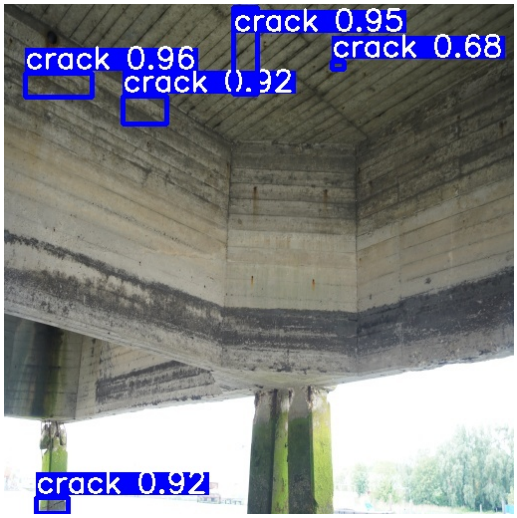
Conf	**IoU**	**Precision**	**Recall**	**mAP@0.5**	**mAP@0.5:0.95**
0.001	0.5	0.646	0.307	0.311	0.164
0.001	0.6	0.613	0.307	0.307	0.164
0.1	0.5	0.646	0.307	0.394	0.228
0.1	0.6	0.613	0.307	0.379	0.216
0.2	0.5	0.646	0.307	0.439	0.256
0.2	0.6	0.613	0.307	0.417	0.246
0.3	0.5	0.645	0.307	0.471	0.286
0.3	0.6	0.613	0.307	0.453	0.275
0.4	0.5	0.654	0.301	0.481	0.296
0.4	0.6	0.616	0.301	0.466	0.286

The uniform scaling factor improved the metrics by a small amount, keeping the mAP around 30%, which is an acceptable amount, more so for this problematic dataset.

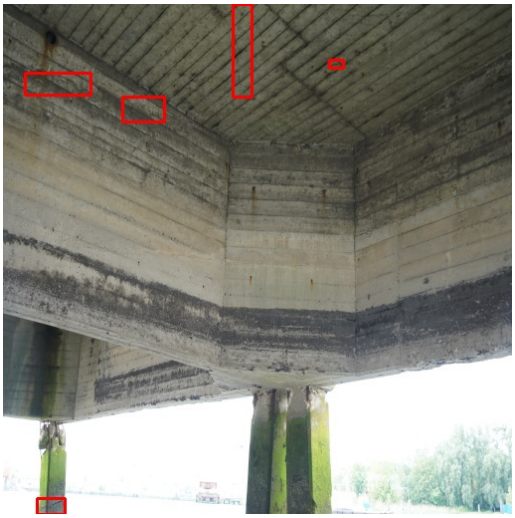


(a) Detection sample 1.

(b) Ground truth sample 1.



(c) Detection sample 2.



(d) Ground truth sample 2.

Figure 6.2: Comparison between detections and ground truth of the Uniform scaling factor versions.

The pictures show that the model is accurate, but has difficulties with small defects as expected.

6.2.2 YOLOv5s version

Table 6.12: Evaluation metrics for the 1024×1024 FP32 `.pt` version of the YOLOv5s.

Conf	IoU	Precision	Recall	mAP@0.5	mAP@0.5:0.95
0.001	0.5	0.810	0.358	0.453	0.277
0.001	0.6	0.810	0.358	0.448	0.276
0.1	0.5	0.810	0.358	0.512	0.346
0.1	0.6	0.810	0.358	0.509	0.338
0.2	0.5	0.810	0.358	0.537	0.372
0.2	0.6	0.810	0.358	0.533	0.367
0.3	0.5	0.810	0.358	0.554	0.393
0.3	0.6	0.810	0.358	0.549	0.389
0.4	0.5	0.810	0.358	0.569	0.406
0.4	0.6	0.810	0.358	0.567	0.404

The comparison at this stage between the two versions of v5n and v5s is almost identical.

Table 6.13: Evaluation metrics for the 640×640 FP32 `.ONNX` version of the YOLOv5s.

Conf	IoU	Precision	Recall	mAP@0.5	mAP@0.5:0.95
0.001	0.5	0.619	0.323	0.356	0.208
0.001	0.6	0.600	0.324	0.354	0.210
0.1	0.5	0.619	0.323	0.421	0.267
0.1	0.6	0.600	0.324	0.413	0.258
0.2	0.5	0.619	0.323	0.448	0.290
0.2	0.6	0.600	0.324	0.436	0.281
0.3	0.5	0.619	0.323	0.473	0.307
0.3	0.6	0.600	0.324	0.463	0.303
0.4	0.5	0.619	0.323	0.483	0.318
0.4	0.6	0.600	0.324	0.480	0.317

Scaling down to 640×640 affected the v5s model more, all metrics are, by a small amount, lower.

Table 6.14: Evaluation metrics for the 640×640 `QDQ` version of the YOLOv5s.

Conf	IoU	Precision	Recall	mAP@0.5	mAP@0.5:0.95
0.001	0.5	0.646	0.307	0.339	0.189
0.001	0.6	0.616	0.307	0.334	0.187
0.1	0.5	0.646	0.307	0.404	0.243
0.1	0.6	0.616	0.307	0.386	0.230
0.2	0.5	0.646	0.307	0.428	0.263
0.2	0.6	0.616	0.307	0.410	0.249
0.3	0.5	0.646	0.307	0.454	0.287
0.3	0.6	0.616	0.307	0.437	0.276
0.4	0.5	0.646	0.307	0.465	0.297
0.4	0.6	0.616	0.307	0.455	0.291

The **QDQ** process affected the *v5s* model more, all metrics except for mAP@0.5-0.95 are worse with this model.

Table 6.15: Evaluation metrics for the 640×640 `QDQ` with uniform scaling factor version of the YOLOv5s.

Conf	IoU	Precision	Recall	mAP@0.5	mAP@0.5:0.95
0.001	0.5	0.707	0.273	0.328	0.188
0.001	0.6	0.708	0.273	0.326	0.188
0.1	0.5	0.707	0.273	0.397	0.241
0.1	0.6	0.708	0.273	0.383	0.232
0.2	0.5	0.707	0.273	0.424	0.264
0.2	0.6	0.708	0.273	0.405	0.252
0.3	0.5	0.707	0.273	0.434	0.280
0.3	0.6	0.708	0.273	0.420	0.271
0.4	0.5	0.707	0.273	0.457	0.299
0.4	0.6	0.708	0.273	0.447	0.293

Precision is a bit higher than the v5n version, but Recall and mAP@0.5 are worse, with an equal value of mAP@0.5-0.95.

6.2.3 Comparison

The table reported below will highlight the best versions of v5n and v5s, before the **qdq** process is applied and then after scaling factor optimization.

Table 6.16: Comparison of the model's performance (mAP@0.5:0.95) after the export as 640×640, both as FP32 and INT8 models.

Model	Conf-value	FP32	INT8	Δ
5n	0.4	0.327	0.296	-0.031
5s	0.4	0.317	0.299	-0.018

Considering the different number of parameters, we can compare the different performance and see that the **v5n** version, although it has half the number of parameters, behaves almost like the **v5s** version, with an mAP that is almost the same.

What might also be useful is taking a look at how the rescaling changed the model's performance.

Table 6.17: Comparison of the model's performance before and after the export as 640×640, both as FP32 models.

Model	**Conf-value**	**1024**	**640**	**Δ**
5n	0.4	0.408	0.327	-0.081
5s	0.4	0.406	0.317	-0.089

This is the major drawback of this dataset; this model should be trained only on the segmented images to probably obtain performances similar to those of the PV datasets.

Before taking a look at the COCO models, the different performances among the different sizes reported below.

Table 6.18: Comparison of model sizes and performance with the 5n version.

Size	**Conf-value**	**IoU-value**	**mAP@0.5**	**mAP@0.5:0.95**
416	0.4	0.5	0.369	0.207
512	0.4	0.5	0.399	0.232
640	0.4	0.5	0.481	0.296

This table shows how much size impacts performance, resulting in defects that are smaller than **3 pixels** and compromise the results when going under the 640×640 size.

6.3 COCO dataset

The COCO dataset has been trained with 3 different *hyps* at 100, 300 and 500 epochs and both with standard LR and Cosine LR, then the standard version (100 epochs) has been trained also with sizes of 416 and 512.

6.3.1 FP32 versions

For these models only the best versions of each model are going to be reported, The **F1 score** highlighted the *best confidence values* at 0.4 and the results from the initial `.pt` version got better with the conversion to the `ONNX` FP32 version, reported below are the 3 models.

Table 6.19: Comparison of the models on the COCO dataset with different hyps on 640×640.

Epochs	Precision	Recall	mAP@0.5	mAP@0.5:0.95
Bench	0.558	0.407	0.427	0.252
100	0.738	0.300	0.526	0.362
300	0.734	0.322	0.536	0.367
500	0.731	0.322	0.534	0.367

The models perform well, the mAP is higher, and considering the starting point (benchmark for yolov5n on COCO), the values are useful when compared with the other INT8 versions.

6.3.2 QDQ versions

Table 6.20: Comparison of the models on the COCO dataset post QDQ.

Epochs	Precision	Recall	mAP@0.5	mAP@0.5:0.95
Bench	0.572	0.383	0.410	0.226
100	0.785	0.237	0.515	0.345
300	0.767	0.206	0.488	0.333
500	0.738	0.286	0.517	0.330

In this table, it is shown that the impact of the quantization simulation is evident but the mAP is still above the target 30%.

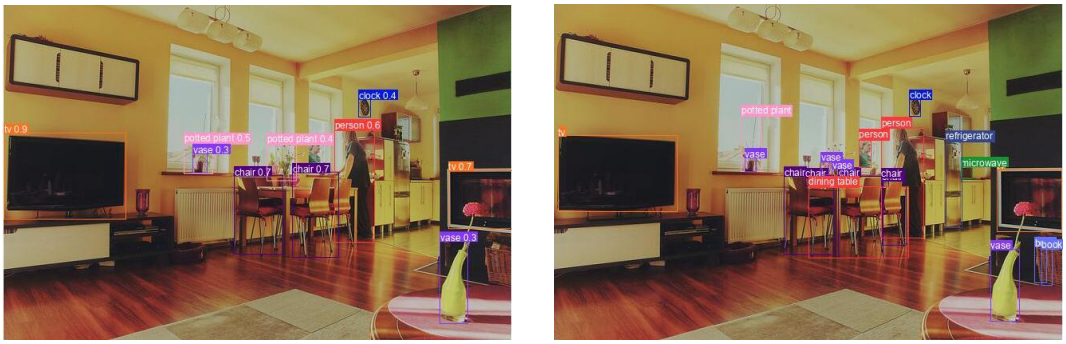
6.3.3 Uniform scaling factor versions

Table 6.21: Comparison of the models on the COCO dataset with uniform scaling factors.

Epochs	Precision	Recall	mAP@0.5	mAP@0.5:0.95
Bench	0.566	0.385	0.409	0.224
100	0.774	0.245	0.513	0.341
300	0.785	0.216	0.502	0.339
500	0.777	0.227	0.503	0.327

The following table shows that all the models are well above the threshold of 30% and that the quantization, with the loss it brings, did not affect the COCO models too much.

Here are reported the detection and labels of some examples:



(a) Detection sample. **(b)** Ground truth sample.

Figure 6.3: Comparison between Predictions and Ground truth for the coco model.

The final table that we are going to see is related to the differences among different scaled COCO models:

Table 6.22: Comparison of model sizes and performance with the COCO models.

Size	Conf-value	IoU-value	mAP@0.5	mAP@0.5:0.95
416	0.4	0.5	0.491	0.328
512	0.4	0.5	0.471	0.313
640	0.4	0.5	0.513	0.341

These results show that the models still perform pretty well even with smaller sizes, the 416×416 performs even better than the 512×512, staying well above the 30% threshold.

6.4 Results considerations

These different models confirm that the applied workflow did result in an acceptable loss for each model, keeping every model to a mAP@0.5:0.95 at least at 30%, which is the key factor for the object detection tasks which have been taken into account. The only model that went just below this threshold (by around 0.4%)is the ASXL model, which is penalized by the different scales of the images used to train the model, if properly trained the results could have much better results.

Chapter 7

Conclusions

In the context of modern machine learning, the demand for deploying high-performance deep learning models in resource-constrained environments, such as embedded systems and edge devices, has driven significant interest in model compression and optimization techniques. Among these, quantization has emerged as a key enabler for reducing the memory footprint, computational complexity, and power consumption of neural networks, making them suitable for real-time inference on non-specialized hardware. However, ensuring that such optimizations do not compromise the accuracy and reliability of the models remains a crucial challenge.

This thesis demonstrates the successful implementation of a complete quantization pipeline for object detection models, showing that it is possible to achieve efficient deployable models without incurring significant accuracy loss. Although hardware deployment remains a natural continuation, the findings and solutions explored in this study provide a solid foundation for future work in this direction.

The project highlights the robustness of the YOLO architecture as a reliable object detection solution, validated both on a large-scale benchmark dataset (COCO) and on two real-world application scenarios. These evaluations confirm the effectiveness of the model in terms of both theoretical design and practical performance.

In addition to engineering and implementation aspects, this work includes an exploratory component focused on evaluating multiple tools, frameworks, and workflows for quantization and model conversion. Comparative analysis of different approaches, including PyTorch-native quantization, ONNX export, QDQ pattern application, and optimization with QONNX and Vitis AI, provided valuable insight into the capabilities, limitations, and integration strategies of each technology. This investigative process helped identify the most effective pipeline in terms of the compatibility, automation, and accuracy of the final model.

In particular, the quantization of the YOLOv5n model presents promising opportunities for deployment on edge devices, where computational efficiency is crucial. This study also serves as a practical guide for applying post-training quantization

(QDQ) and model conversion using the Vitis AI toolchain, showing that such tools are effective for producing optimized models ready for hardware acceleration.

The results obtained provide valuable benchmarks for future research involving similar datasets or objectives. The comparative analysis between model scales, especially the v5n version, demonstrates the trade-offs between performance and complexity, offering a useful reference for further developments in the field of quantized deep learning models and their integration into hardware-based systems.

Appendix A

Appendix A

This appendix will feature the training configurations and the corrections made to the training scripts due to the post-process that had to be applied to the output tensors.

The `.yaml file` that contains the optimizations (along with the *hyps*) is the following for the Photovoltaic Thermal dataset.

Listing A.1: Hyps and main settings for the PV dataset.

```
hyp:
  lr0: 0.01
  lrf: 0.01
  momentum: 0.937
  weight_decay: 0.0005
  warmup_epochs: 3.0
  warmup_momentum: 0.8
  warmup_bias_lr: 0.1
  box: 0.05
  cls: 0.5
  cls_pw: 1.0
  obj: 1.0
  obj_pw: 1.0
  iou_t: 0.2
  anchor_t: 4.0
  fl_gamma: 0.0
  hsv_h: 0.015
  hsv_s: 0.7
  hsv_v: 0.4
  degrees: 0.0
  translate: 0.1
  scale: 0.5
  shear: 0.0
  perspective: 0.0
  flipud: 0.0
  fliplr: 0.5
  mosaic: 1.0
  mixup: 0.0
```

```
  copy_paste: 0.0
epochs: 100
batch_size: 16
imgsz: 640
single_cls: true
optimizer: SGD
```

For the ASXL dataset the settings were

Listing A.2: Hyps and main settings for the ASXL dataset.

```
hyp:
  lr0: 0.001
  lrf: 0.1
  momentum: 0.937
  weight_decay: 0.0005
  warmup_epochs: 3.0
  warmup_momentum: 0.8
  warmup_bias_lr: 0.1
  box: 0.05
  cls: 0.3
  cls_pw: 1.0
  obj: 1.7
  obj_pw: 2.0
  iou_t: 0.3
  anchor_t: 4.0
  fl_gamma: 0.0
  hsv_h: 0.015
  hsv_s: 0.7
  hsv_v: 0.4
  degrees: 0.0
  translate: 0.1
  scale: 0.9
  shear: 0.0
  perspective: 0.0
  flipud: 0.0
  fliplr: 0.5
  mosaic: 1.0
  mixup: 0.1
  copy_paste: 0.0
epochs: 2000
batch_size: 48
imgsz: 1024
single_cls: true
optimizer: AdamW
```

This model has been trained first on a 1024×1024 because of the small size defects in the full scale images.

For the `yolov5s` version the *hyps* are:

Listing A.3: Hyps and main settings for the v5s ASXL dataset model.

```
hyp:
  lr0: 0.00025
  lrf: 0.01
  momentum: 0.937
  weight_decay: 0.0005
  warmup_epochs: 3.0
  warmup_momentum: 0.8
  warmup_bias_lr: 0.1
  box: 0.05
  cls: 0.3
  cls_pw: 1.0
  obj: 1.7
  obj_pw: 2.0
  iou_t: 0.3
  anchor_t: 4.0
  fl_gamma: 0.0
  hsv_h: 0.015
  hsv_s: 0.7
  hsv_v: 0.4
  degrees: 0.0
  translate: 0.1
  scale: 0.9
  shear: 0.0
  perspective: 0.0
  flipud: 0.0
  fliplr: 0.5
  mosaic: 1.0
  mixup: 0.1
  copy_paste: 0.0
```

COCO has been trained with the same *hyps* of the PV dataset. Also other versions have been tested (with 300 and 500 epochs and **CosineAnnealing** as **Scheduler**). The training script had to be modified in order to post-process the results correctly, and the following code segment contains the fixtures applied in which the output of the model consists of a list of tensors, one with the full grid and one with the 3 **multi scale tensors**.

Listing A.4: Post processing of the tensors and grid.

```

def postprocessing(x):
    grid = [torch.empty(0) for _ in range(len(x))]
    z = []
    anchor_grid = [torch.empty(0) for _ in range(len(x))]
    stride = torch.tensor([8., 16., 32.][:len(x)], device='cuda
    ↪ :0')  # Adatta il numero di stride
    anchors = torch.tensor([[1.25, 1.625, 2.0, 3.75, 4.125,
    ↪ 2.875],
```

```
                                [1.875, 3.8125, 3.875, 2.8125,
    ↪ 3.6875, 7.4375],
                                [3.625, 2.8125, 4.875, 6.1875,
    ↪ 11.65625, 10.1875]], device='cuda:0')
     anchors = anchors[:len(x)].float().view(len(x), -1, 2)  #
    ↪ Adatta il numero di ancore

     for i in range(len(x)):
         bs, _, ny, nx = x[i].shape  # Assumi che x[i] abbia una
    ↪  forma valida
         x[i] = x[i].view(bs, 3, 6, ny, nx).permute(0, 1, 3, 4,
    ↪ 2).contiguous()

         if grid[i].shape[2:4] != x[i].shape[2:4]:
             grid[i], anchor_grid[i] = make_grid(nx, ny, i,
    ↪ anchors, stride)

         xy, wh, conf = x[i].sigmoid().split((2, 2, 1 + 1), 4)
         xy = (xy * 2 + grid[i]) * stride[i]  # xy
         wh = (wh * 2) ** 2 * anchor_grid[i]  # wh
         y = torch.cat((xy, wh, conf), 4)
         z.append(y.view(bs, 3 * nx * ny, 6))

     return torch.cat(z, 1), x

def make_grid(nx=20, ny=20, i=0,anchors = None, stride = None,
    ↪ torch_1_10=check_version(torch.__version__, '1.10.0')):
     d = anchors[i].device
     t = anchors[i].dtype

     shape = 1, 3, ny, nx, 2  # grid shape
     y, x = torch.arange(ny, device=d, dtype=t), torch.arange(nx
    ↪ , device=d, dtype=t)
     yv, xv = torch.meshgrid(y, x, indexing='ij') if torch_1_10
    ↪ else torch.meshgrid(y, x)  # torch>=0.7 compatibility
     grid = torch.stack((xv, yv), 2).expand(shape) - 0.5  # add
    ↪ grid offset, i.e. y = 2.0 * x - 0.5
     anchor_grid = (anchors[i] * stride[i]).view((1, 3, 1, 1, 2)
    ↪ ).expand(shape)
     return grid, anchor_grid
```

The following code excerpt illustrates the core of the training loop inside `train.py`, which is responsible for handling mini-batch iterations over the dataset. It integrates automatic mixed precision (AMP) to reduce memory consumption and improve performance, adapts learning rates during the warm-up phase, and applies data augmentation through multi-scale training. Each batch is passed through the model, the loss is computed using ground truth labels, and backpropagation is performed using scaled gradients. This snippet exemplifies a typical iteration during model

optimization in YOLOv5 training.

Listing A.5: Training loop with AMP and loss computation.

```
for i, (imgs, targets, paths, _) in pbar:
    ni = i + nb * epoch
    imgs = imgs.to(device, non_blocking=True).float() / 255

    if ni <= nw:
        xi = [0, nw]
        accumulate = max(1, np.interp(ni, xi, [1, nbs /
    ↪ batch_size]).round())
        for j, x in enumerate(optimizer.param_groups):
            x['lr'] = np.interp(ni, xi, [hyp['warmup_bias_lr']
    ↪ if j == 0 else 0.0, x['initial_lr'] * lf(epoch)])
            if 'momentum' in x:
                x['momentum'] = np.interp(ni, xi, [hyp['
    ↪ warmup_momentum'], hyp['momentum']])

    if opt.multi_scale:
        sz = random.randrange(int(imgsz * 0.5), int(imgsz *
    ↪ 1.5) + gs) // gs * gs
        sf = sz / max(imgs.shape[2:])
        if sf != 1:
            ns = [math.ceil(x * sf / gs) * gs for x in imgs.
    ↪ shape[2:]]
            imgs = nn.functional.interpolate(imgs, size=ns,
    ↪ mode='bilinear', align_corners=False)

    with torch.cuda.amp.autocast(amp):
        predi = model(imgs)
        pred = postprocessing(predi)
        loss, loss_items = compute_loss(pred, targets.to(device
    ↪ ))
        if RANK != -1:
            loss *= WORLD_SIZE
        if opt.quad:
            loss *= 4.

    scaler.scale(loss).backward()
```

In the `val.py` script, which is run at the end of every training epoch, the core segment is the following one which processes each batch from the validation set, applies model inference and postprocessing,and collects statistics for precision, recall, and mAP computation. The loop also handles saving visualizations and label files, if enabled.

Listing A.6: Validation loop processing each image batch.

```
for batch_i, (im, targets, paths, shapes) in enumerate(pbar):
    callbacks.run('on_val_batch_start')
```

```
    with dt[0]:
        if cuda:
            im = im.to(device, non_blocking=True)
            targets = targets.to(device)
        im = im.half() if half else im.float()
        im /= 255
        nb, _, height, width = im.shape

    with dt[1]:
        pred = model(im)
        alt = model(im, augment=augment)
        predi = postprocessing(pred)
        altpredi = postprocessing(alt)
        if compute_loss:
            preds = predi
            train_out = predi
        else:
            preds = altpredi
            train_out = None

    if compute_loss:
        loss += compute_loss(train_out, targets)[1]

    targets[:, 2:] *= torch.tensor((width, height, width,
    ↪ height), device=device)
    lb = [targets[targets[:, 0] == i, 1:] for i in range(nb)]
    ↪ if save_hybrid else []
    with dt[2]:
        preds = non_max_suppression(preds, conf_thres,
    ↪ iou_thres,
                                    labels=lb, multi_label=True
    ↪ ,
                                    agnostic=single_cls,
    ↪ max_det=max_det)
```

At the end of each validation the metrics are computed, and this is the snippet of code responsible for this task.

Listing A.7: Computing evaluation metrics from validation statistics.

```
stats = [torch.cat(x, 0).cpu().numpy() for x in zip(*stats)]
if len(stats) and stats[0].any():
    tp, fp, p, r, f1, ap, ap_class = ap_per_class(*stats, plot=
    ↪ plots, save_dir=save_dir, names=names)
    ap50, ap = ap[:, 0], ap.mean(1)
    mp, mr, map50, map = p.mean(), r.mean(), ap50.mean(), ap.
    ↪ mean()
nt = np.bincount(stats[3].astype(int), minlength=nc)

pf = '%22s' + '%11i' * 2 + '%11.3g' * 4
```

```
LOGGER.info(pf % ('all', seen, nt.sum(), mp, mr, map50, map))

if (verbose or (nc < 50 and not training)) and nc > 1 and len(stats):
    for i, c in enumerate(ap_class):
        LOGGER.info(pf % (names[c], seen, nt[c], p[i], r[i], ap50[i], ap[i]))
```

The next snippet, taken from `detect.py`, shows how YOLOv5 inference predictions are handled and saved. After rescaling the predicted bounding boxes to the original image resolution, the script allows to save the predictions in TXT and CSV formats. This code is critical for logging detection outputs, enabling both quantitative evaluation and visualization.

Listing A.8: Detection results post-processing and saving.

```
for i, det in enumerate(pred):  # per image
    seen += 1
    if webcam:
        p, im0, frame = path[i], im0s[i].copy(), dataset.count
    else:
        p, im0, frame = path, im0s.copy(), getattr(dataset, 'frame', 0)

    p = Path(p)
    save_path = str(save_dir / p.name)
    txt_path = str(save_dir / 'labels' / p.stem) + ('' if dataset.mode == 'image' else f'_{frame}')
    s += '%gx%g ' % im.shape[2:]
    gn = torch.tensor(im0.shape)[[1, 0, 1, 0]]
    imc = im0.copy() if save_crop else im0
    annotator = Annotator(im0, line_width=line_thickness, example=str(names))

    if len(det):
        det[:, :4] = scale_boxes(im.shape[2:], det[:, :4], im0.shape).round()
        for *xyxy, conf, cls in reversed(det):
            c = int(cls)
            label = names[c] if hide_conf else f'{names[c]}'
            confidence = float(conf)
            confidence_str = f'{confidence:.2f}'

            if save_csv:
                write_to_csv(p.name, label, confidence_str)

            if save_txt:
                xywh = (xyxy2xywh(torch.tensor(xyxy).view(1, 4) ) / gn).view(-1).tolist()
```

```
29                  line = (cls, *xywh, conf) if save_conf else (
    ↪ cls, *xywh)
30                  with open(f'{txt_path}.txt', 'a') as f:
31                      f.write(('%g ' * len(line)).rstrip() % line
    ↪  + '\n')
32
33              if save_img or save_crop or view_img:
34                  label = None if hide_labels else (names[c] if
    ↪ hide_conf else f'{names[c]} {conf:.2f}')
35                  annotator.box_label(xyxy, label, color=colors(c
    ↪ , True))
36              if save_crop:
37                  save_one_box(xyxy, imc, file=save_dir / 'crops'
    ↪  / names[c] / f'{p.stem}.jpg', BGR=True)
```

Appendix B

Appendix B

This appendix will treat all the scripts related to the quantization process of the models. The first script that is important to describe is the `quantOnnx.py` which treats the `.onnx` versions of the models and applies the **QDQ** through the **VAI** functions.

Listing B.1: Data reader and iterator on the images.

```
class CustomCalibrationDataReader(CalibrationDataReader):
    def __init__(self, data_dir, img_size):
        self.data_dir = data_dir
        self.img_size = img_size
        self.data = []
        self.iterator = None
        self.load_data()

    def load_data(self):
        # images_dir = os.path.join(self.data_dir, "images", "
    ↪ train2017")
        max_num_img = 200
        images_dir = os.path.join(self.data_dir, "images", "
    ↪ train2017")
        images = sorted(os.listdir(images_dir))  # Ensure
    ↪ sorted order
        num_img = 0
        for img in images:
            num_img = num_img + 1
            img_path = os.path.join(images_dir, img)
            self.data.append(img_path)
            if num_img == max_num_img :
                break
        self.iterator = iter(self.data)

    def get_next(self) -> dict:
        try:
            img_path = next(self.iterator)
            img = self.preprocess_image(img_path)
```

```
            return {"images": img}  # Match key with 'input_nodes'
        except StopIteration:
            return None

    def preprocess_image(self, img_path):
        img = cv2.imread(img_path)
        img = cv2.resize(img, (self.img_size, self.img_size))  # Usa la dimensione dinamica
        img = np.transpose(img, (2, 0, 1))  # HWC to CHW
        #convert to float32
        img = img.astype(np.float32)
        img = img / 255.0
        img = np.expand_dims(img, axis=0).astype(np.float32)  # Add batch dimension
        return img

    def reset(self):
        self.iterator = iter(self.data)  # Reset iterator

    def get_size(self):
        return len(self.data)
```

These functions setup the calibration dataset for the **static quantization**, preprocess the images to be compliant with the model structure and define get and reset. The calibration set has 200 images on which the model will calibrate since a good value is between 100 and 1000 in order to successfully calibrate the quantized version of the model.

Listing B.2: Argparse and setup before effective quantization function.

```

def main():
    parser = argparse.ArgumentParser(description="Quantizzazione di un modello ONNX con Vitis-AI")
    parser.add_argument('--model_input', type=str, required=True, help="Percorso del file ONNX di input")
    parser.add_argument('--model_output', type=str, default='quantModels', help="Percorso del file ONNX quantizzato di output")
    parser.add_argument('--calibration_data', type=str, default='/workspace/Vitis-AI-Reference-Tutorials/Quantizing-Compiling-Yolov5-Hackster-Tutorial/cracks_dataset', help="Percorso del dataset di calibrazione (file o directory)")
    parser.add_argument('--output_name', type=str, default='exp1', help="Nome del file ONNX quantizzato di output (senza estensione)")
    parser.add_argument('--img_size', type=int, default=640,
```

```
    ↪ help="Dimensione dell'immagine per la calibrazione (
    ↪ default: 640)")
    args = parser.parse_args()

    model = onnx.load(args.model_input)
    input_name = model.graph.input[0].name  # Prendi il primo
    ↪ input del modello
    print(f"Nome dell'input del modello: {input_name}")

    model_input = args.model_input
    output_name = args.output_name
    calibration_data_path = args.calibration_data
    img_size = args.img_size
    output_dir = "quantModels"
    os.makedirs(output_dir, exist_ok=True)

    model_output = os.path.join(output_dir, f"{output_name}.
    ↪ onnx")
```

This segment is responsible for the parsing of the quantization parameters, *model input, output, calibration dataset, output model name and size of the model.*

Listing B.3: Quantization function.

```


    vai_q_onnx.quantize_static(
        model_input=model_input,
        model_output=model_output,
        calibration_data_reader=CustomCalibrationDataReader(
    ↪ calibration_data_path, img_size),
        quant_format=QuantFormat.QDQ,
        calibrate_method=vai_q_onnx.PowerOfTwoMethod.MinMSE,
        extra_options={"ActivationSymmetric": True, "
    ↪ WeightSymmetric": True, "AddQDQPairToWeight": True, "
    ↪ ForceQuantizeNoInputCheck": True},
    )

    print("Modello quantizzato salvato in:", model_output)
```

This is the core of the script; this function takes the model input and output, the calibration data on which the quantized model is going to iterate and the format that will be applied, which is QDQ in this case. Then the *calibrate method* is set as `PowerOfTwoMethod`, which has 2 versions: `MinMSE` or `NonOverflow`, but for this thesis the first one is a good fit since usually has better accuracy. The `extra_options` field has the first two parameters that, as is said by the name, symmetrize the calibration values for weights and activations. The third parameter, `AddQDQPairToWeight`, if True, ensures that both QuantizeLinear and DeQuantizeLinear nodes are inserted for weight, maintaining its floating point format, and in

our case, with the `PowerOfTwoMethod` calibration method, this setting will also be effective for the bias. The final one is a parameter that the VAI guide suggests to set as true in case of `PowerOfTwoMethod`, and if set as True, latent operators such as maxpool and transpose will always quantize their inputs, generating quantized outputs even if their inputs have not been quantized.

After successful quantization it was time to walk through the model's graph and set the scaling factors to coherent values.

The idea was to search for the **QDQ** chains before each target layer (in this case **LeakyReLU, Add and Concat**, and uniform the scaling factors. The script is divided into three main parts:

- **Model Cleanup**: It first runs a structural cleanup on the input ONNX model to remove redundant nodes or artifacts, producing a cleaner version suitable for processing.
- **Scale Unification for QDQ Chains**: If the scales are not uniform across inputs, the script updates the upstream QDQ chains to enforce a single, common scale—usually the one of the output.
- **Model Conversion**: Once scales are unified, the script invokes the `qonnx.convert` function to finalize and serialize the quantized model, making it compatible for deployment on hardware accelerators.

Listing B.4: ONNX QDQ Scale Unification Script.

```
def get_scale_from_node(node, graph):
    if node.op_type not in ["QuantizeLinear", "DequantizeLinear
    ↪ "]:
        return None
    scale_name = node.input[1]
    scale_init = next((init for init in graph.initializer if
    ↪ init.name == scale_name), None)
    if scale_init is None:
        return None
    scale = numpy_helper.to_array(scale_init)
    if scale.size == 1:
        return float(scale.item())
    return None
```

This function retrieves the quantization scale (if scalar) from ‘QuantizeLinear‘ or ‘DequantizeLinear‘ nodes in an ONNX graph.

Listing B.5: Function to set the scaling factor.

```
def set_scale_for_node(node, graph, new_scale, used_inits):
    scale_name = node.input[1]
    scale_init = next((init for init in graph.initializer if
    ↪ init.name == scale_name), None)

```

```
    if scale_init is None:
        new_init_name = scale_name + "_forced"
        count = 1
        while new_init_name in used_inits:
            new_init_name = f"{scale_name}_forced_{count}"
            count += 1

        new_scale_scalar = np.float32(new_scale)
        new_init = numpy_helper.from_array(new_scale_scalar,
    ↪ new_init_name)
        graph.initializer.append(new_init)
        used_inits.add(new_init_name)
        node.input[1] = new_init_name
        return
    old_scale_array = numpy_helper.to_array(scale_init)
    if old_scale_array.size != 1:
        new_init_name = scale_name + "_forced"
        count = 1
        while new_init_name in used_inits:
            new_init_name = f"{scale_name}_forced_{count}"
            count += 1
        new_scale_scalar = np.float32(new_scale)
        new_init = numpy_helper.from_array(new_scale_scalar,
    ↪ new_init_name)
        graph.initializer.append(new_init)
        used_inits.add(new_init_name)
        node.input[1] = new_init_name
        return
    old_scale_val = float(old_scale_array.item())
    if not np.isclose(old_scale_val, new_scale):
        new_scale_scalar = np.float32(new_scale)
        new_init = numpy_helper.from_array(new_scale_scalar,
    ↪ scale_init.name)
        graph.initializer.remove(scale_init)
        graph.initializer.append(new_init)
```

This function enforces a new scale for a node. It handles missing, non-scalar, or mismatched scales by creating or updating initializers as needed.

Listing B.6: Iteration to check QDQ chains.

```
def find_qdq_chain_starting_from_dq(graph, dq_node):
    chain = []
    current_dq = dq_node
    while True:
        chain.insert(0, current_dq)
        q_node = next((n for n in graph.node if n.op_type == "
    ↪ QuantizeLinear" and n.output[0] == current_dq.input[0]),
    ↪ None)
        if q_node is None:
```

```
            break
        chain.insert(0, q_node)
        prev_dq = next((n for n in graph.node if n.op_type == "
    ↪ DequantizeLinear" and n.output[0] == q_node.input[0]),
    ↪ None)
        if prev_dq is None or prev_dq == current_dq:
            break
        current_dq = prev_dq
    return chain
```

This function reconstructs the entire QDQ (Quantize-Dequantize) chain backwards starting from a ‘DequantizeLinear‘ node.

```
def find_next_quantize_linear(graph, start_output, max_hops=2):
    to_visit = [start_output]
    visited = set()
    hops = 0

    while to_visit and hops < max_hops:
        next_to_visit = []
        for tensor in to_visit:
            for node in graph.node:
                if tensor in node.input:
                    if node.op_type == "QuantizeLinear":
                        return node
                    next_to_visit.extend(node.output)
        visited.update(to_visit)
        to_visit = [t for t in next_to_visit if t not in
    ↪ visited]
        hops += 1
    return None
```

Performs a limited breadth-first search from a tensor to find the next ‘QuantizeLinear‘ node downstream.

Listing B.7: Unify function for add, concat and Leaky ReLU.

```
def unify_scales_for_add_concat(model, output_path, max_iters
    ↪ =10):
    graph = model.graph
    used_inits = set(init.name for init in graph.initializer)
    iteration = 0
    changed = True
    while changed and iteration < max_iters:
        iteration += 1
        print(f"\n=== Iterazione {iteration} per uniformare
    ↪ scale Add/Concat ===")
        changed = False
        for node in graph.node:
```

```
            if node.op_type not in ["Add", "Concat", "LeakyRelu
↪ "]:
                continue
            input_chains = []
            input_scales = []
            for inp in node.input:
                dq_node = next((n for n in graph.node if n.
↪ op_type == "DequantizeLinear" and inp == n.output[0]),
↪ None)
                if dq_node is None:
                    input_chains.append(None)
                    input_scales.append(None)
                    continue
                qdq_chain = find_qdq_chain_starting_from_dq(
↪ graph, dq_node)
                input_chains.append(qdq_chain)
                scales = [get_scale_from_node(n, graph) for n
↪ in qdq_chain if get_scale_from_node(n, graph) is not None
↪ ]
                input_scales.append(min(scales) if scales else
↪ None)
            output_tensor = node.output[0]
            next_q_node = find_next_quantize_linear(graph,
↪ output_tensor)
            if next_q_node is None:
                continue
            output_scale = get_scale_from_node(next_q_node,
↪ graph)
            if output_scale is None:
                continue
            for chain, scale in zip(input_chains, input_scales)
↪ :
                if chain is None or scale is None:
                    continue
                if not np.isclose(scale, output_scale):
                    for n in chain:
                        set_scale_for_node(n, graph,
↪ output_scale, used_inits)
                    changed = True
    print(f" Salvataggio modello risultante in: {output_path}")
    onnx.save(model, output_path)
```

Unifies the input scales for nodes like ‘Add‘, ‘Concat‘, and ‘LeakyRelu‘ by adjusting upstream QDQ chains so that all inputs and outputs use the same scale.

Listing B.8: Main function, which applies the different functions.

```
def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--model_input', type=str, required=
    ↪ True, help="Percorso modello ONNX (post-QDQ)")
    parser.add_argument('--output_style', type=str, default="
    ↪ quant", help="Stile output conversione")
    args = parser.parse_args()
    model_input = args.model_input
    base, ext = os.path.splitext(model_input)
    cleaned_model_file = base + "_clean" + ext
    unified_add_concat_file = base + "_uniScaleLR_out" + ext
    if base.endswith("_qdq"):
        final_base = base[:-4]
    else:
        final_base = base
    converted_model_file = final_base + "_final_scaleoutLR_q" +
    ↪  ext
    print(" Pulizia modello (cleanup)...")
    cleanup(model_input)
    if not os.path.exists(cleaned_model_file):
        print(f" Errore: file pulito {cleaned_model_file} non
    ↪ trovato!")
        return
    print(" Caricamento modello pulito in memoria...")
    cleaned_model = onnx.load(cleaned_model_file)
    print(" Uniformo scale input di Add/Concat su tutta la
    ↪ catena QDQ...")
    unify_scales_for_add_concat(cleaned_model,
    ↪ unified_add_concat_file)
    print(" Conversione modello con QONNX...")
    convert(unified_add_concat_file, output_style=args.
    ↪ output_style, output_file=converted_model_file)
    print(f"\n Modello convertito salvato in: {
    ↪ converted_model_file}")
    os.remove(cleaned_model_file)

if __name__ == "__main__":
    main()
```

This is the main execution script. It handles argument parsing, cleans the model, normalizes the QDQ scales, and converts the model using QONNX. It also removes temporary artifacts.

Bibliography

[1] Iqbal H. Sarker. "Machine Learning: Algorithms, Real-World Applications and Research Directions". In: *SN Computer Science* 2.3 (2021), p. 160. DOI: 10.1007/s42979-021-00592-x. URL: https://doi.org/10.1007/s42979-021-00592-x (cit. on pp. 1, 2).

[2] Yuxi Li. "Deep Reinforcement Learning: An Overview". In: *arXiv preprint arXiv:1701.07274* (2017). URL: https://arxiv.org/abs/1701.07274 (cit. on pp. 1, 2).

[3] Ibomoiye Domor Mienye and Theo G. Swart. "A Comprehensive Review of Deep Learning: Architectures, Recent Advances, and Applications". In: *Information* 15.12 (2024). ISSN: 2078-2489. DOI: 10.3390/info15120755. URL: https://www.mdpi.com/2078-2489/15/12/755 (cit. on pp. 2–4).

[4] Dalila Durães, Pedro Miguel Freitas, and Paulo Novais. "The relevance of Deepfakes in the Administration of Criminal Justice". In: *Multidisciplinary Perspectives on Artificial Intelligence and the Law*. Ed. by H. S. Antunes, P. M. Freitas, A. L. Oliveira, C. M. Pereira, E. V. D. Sequeira, and L. B. Xavier. Law, Governance and Technology Series. Open Access. Cham, Switzerland: Springer, 2024, pp. 364–366. URL: https://www.researchgate.net/publication/376852834_The_Relevance_of_Deepfakes_in_the_Administration_of_Criminal_Justice (visited on 06/14/2025) (cit. on p. 3).

[5] freeCodeCamp. *Deep Learning Neural Networks Explained in Plain English*. Accessed: 2025-06-14. Jan. 2022. URL: https://www.freecodecamp.org/news/deep-learning-neural-networks-explained-in-plain-english/ (cit. on p. 4).

[6] EpyNN Documentation. *Pooling (CNN) — EpyNN 1.0 documentation*. Consultazione: 2025-06-14. 2025. URL: https://epynn.net/Pooling.html (cit. on p. 5).

[7] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *Nature* 521.7553 (2015), pp. 436–444 (cit. on p. 5).

[8] Mohd Halim Mohd Noor and Ayokunle Olalekan Ige. *A Survey on State-of-the-art Deep Learning Applications and Challenges*. Accepted for publication in *Engineering Applications of Artificial Intelligence* (Elsevier). 2024. URL: https://arxiv.org/abs/2403.17561 (cit. on pp. 5, 6).

[9] Nafiz Shahriar. *What Is Convolutional Neural Network –CNN & Deep Learning*. Accessed: 2025-06-14. Apr. 2023. URL: https://nafizshahriar.medium.com/what-is-convolutional-neural-network-cnn-deep-learning-b3921bdd82d5 (cit. on p. 5).

[10] Bsher Karbouj, Garabet A. Topalian-Rivas, and Jörg Krüger. "Comparative Performance Evaluation of One-Stage and Two-Stage Object Detectors for Screw Head Detection and Classification in Disassembly Processes". In: *Procedia CIRP* 122 (2024). 31st CIRP Conference on Life Cycle Engineering, pp. 527–532. ISSN: 2212-8271. DOI: 10.1016/j.procir.2024.01.077. URL: https://www.sciencedirect.com/science/article/pii/S2212827124001021 (cit. on p. 6).

[11] Junhyung Kang, Shahroz Tariq, Han Oh, and Simon S. Woo. "A Survey of Deep Learning-Based Object Detection Methods and Datasets for Overhead Imagery". In: *IEEE Access* 10 (2022). Accessed: 2025-06-14, pp. 1–1. DOI: 10.1109/ACCESS.2022.3149052. URL: https://www.researchgate.net/publication/358362847_A_Survey_of_Deep_Learning-Based_Object_Detection_Methods_and_Datasets_for_Overhead_Imagery (cit. on p. 6).

[12] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. *You Only Look Once: Unified, Real-Time Object Detection*. arXiv:1506.02640 [cs.CV]. 2016. arXiv: 1506.02640 [cs.CV]. URL: https://arxiv.org/abs/1506.02640 (cit. on pp. 7, 8, 11, 37).

[13] Shao-Yu Yang, Hsu-Yung Cheng, and Chih-Chang Yu. "Real-Time Object Detection and Tracking for Unmanned Aerial Vehicles Based on Convolutional Neural Networks". In: *Electronics*. Vol. 12. 24. Accessed: 2025-06-14. 2023, p. 4928. DOI: 10.3390/electronics12244928. URL: https://www.researchgate.net/publication/337535309_Real-Time_Object_Detection_Based_on_Unmanned_Aerial_Vehicle (cit. on p. 9).

[14] Ayush Yajnik. *Computer Vision: YOLO: Grid Cells and Anchor boxes*. Accessed: 2025-06-14. Sept. 2023. URL: https://medium.com/@ayushyajnik2/computer-vision-yolo-grid-cells-and-anchor-boxes-57b8a33cb25b (cit. on pp. 9, 10).

[15] Rahima Khanam and Muhammad Hussain. *What is YOLOv5: A deep look into the internal features of the popular object detector*. arXiv:2407.20892v1 [cs.CV]. 2024. arXiv: 2407.20892 [cs.CV]. URL: https://arxiv.org/abs/2407.20892 (cit. on pp. 9, 11, 12).

[16] Jacob Solawetz. *What are Anchor Boxes in Object Detection?* Accessed: 2025-06-14. July 2020. URL: https://blog.roboflow.com/what-is-an-anchor-box/ (cit. on p. 10).

[17] Ultralytics. *Non-Maximum Suppression (NMS)*. https://www.ultralytics.com/it/glossary/non-maximum-suppression-nms. Accessed: 2025-06-01. 2024 (cit. on p. 10).

[18] Vineeth S. Subramanyam. *Non Max Suppression (NMS). What is Non Max Suppression, and why is it used?* Accessed: 2025-06-14. Jan. 2021. URL: https://medium.com/analytics-vidhya/non-max-suppression-nms-6623e6572536 (cit. on p. 11).

[19] Ultralytics. *YOLOv5 by Ultralytics*. https://github.com/ultralytics/yolov5?tab=readme-ov-file. Accessed: 2025-06-01. 2024 (cit. on pp. 12, 13).

[20] Florian June. *Model Quantization 1: Basic Concepts*. Accessed: 2025-06-14. Oct. 2023. URL: https://medium.com/@florian_algo/model-quantization-1-basic-concepts-860547ec6aa9 (cit. on p. 13).

[21] Jahid Hasan. *Optimizing Large Language Models through Quantization: A Comparative Analysis of PTQ and QAT Techniques*. 2024. arXiv: 2411.06084 [cs.LG]. URL: https://arxiv.org/abs/2411.06084 (cit. on pp. 14–16).

[22] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Tijmen Blankevoort. *A White Paper on Neural Network Quantization*. 2021. arXiv: 2106.08295 [cs.LG]. URL: https://arxiv.org/abs/2106.08295 (cit. on p. 16).

[23] Qi-Chao Mao, Hong-Mei Sun, Yan-Bo Liu, and Rui-Sheng Jia. "Mini-YOLOv3: Real-Time Object Detector for Embedded Applications". In: *IEEE Access* 7 (2019). Evaluated on COCO dataset; model size ≈23% of YOLOv3, mAP-50 of 52.1 @ 67 fps; accessed via ResearchGate, pp. 133529–133538. DOI: 10.1109/ACCESS.2019.2941547 (cit. on p. 24).

[24] André Magalhães Moraes, Luiz Felipe Pugliese, Rafael Francisco dos Santos, Giovani Bernardes Vitor, Rodrigo Aparecido da Silva Braga, and Fernanda Rodrigues da Silva. *Effectiveness of YOLO Architectures in Tree Detection: Impact of Hyperparameter Tuning and SGD, Adam, and AdamW Optimizers*. 2023. DOI: 10.3390/safety5010009. URL: https://www.mdpi.com/2305-6703/5/1/9 (cit. on pp. 29, 30).

[25] Xipeng Wang. *Learning Rate Scheduling*. Online post. https://xipengwang.github.io/machine-learning-learning-rate-scheduling/. June 2021 (cit. on pp. 31, 32).

[26] AMD/Xilinx. *DPU IP Details and System Integration*. https://xilinx.github.io/Vitis-AI/3.0/html/docs/workflow-system-integration.html. Versione 3.0. July 2023. (Visited on 06/15/2025) (cit. on p. 36).

[27] LogicTronix. *YOLOv5 Quantization & Compilation with Vitis AI 3.0 for Kria*. https://www.hackster.io/LogicTronix/yolov5-quantization-compilation-with-vitis-ai-3-0-for-kria-7b005d. Accessed: 2025-06-16. 2023 (cit. on p. 38).

[28] Mohammad Jani, Jamil Fayyad, Younes Al-Younes, and Homayoun Najjaran. *Model Compression Methods for YOLOv5: A Review*. Submitted on 21 Jul 2023, accessed 16 Jun 2025. 2023. DOI: 10.48550/ARXIV.2307.11904 (cit. on p. 38).

[29] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. *A Survey of Quantization Methods for Efficient Neural Network Inference*. Submitted 25 Mar 2021, revised 21 Jun 2021, accessed 16 Jun 2025. 2021. DOI: 10.48550/ARXIV.2103.13630 (cit. on p. 42).

[30] AMD Vitis AI Documentation. *(Optional) Dumping the Simulation Results*. https://docs.amd.com/r/en-US/ug1414-vitis-ai/Optional-Dumping-the-Simulation-Results?tocId=XZ5iftWIC~VwulZZExtHFQ. Accessed: 2025-06-16. 2023 (cit. on p. 42).

[31] Chongyu Qu, Ritchie Zhao, Ye Yu, Bin Liu, Tianyuan Yao, Junchao Zhu, Bennett A. Landman, Yucheng Tang, and Yuankai Huo. "Post-Training Quantization for 3D Medical Image Segmentation: A Practical Study on Real Inference Engines". In: *arXiv preprint arXiv:2501.17343* (2025). Submitted on 28 Jan 2025; accessed 16 Jun 25. DOI: 10.48550/ARXIV.2501.17343 (cit. on p. 42).