MASTER IN HIGH PERFORMANCE COMPUTING

Performance Evaluation of Object Detection in Different Architectures

Supervisors:
Ivan GIROTTO, ICTP
Antonio SCIARAPPA, LEONARDO SpA

Candidate:
Fernando Santana PACHECO

9th EDITION
2022–2023

Acknowledgements

I would like to express my gratitude to Leonardo SpA for funding this research project and my participation in the MHPC programme.

I am deeply grateful for the availability and bright guidance provided by Antonio Sciarappa throughout this journey.

I am also indebted to Ivan Girotto for the valuable support, as well as to Irina Davidenkova for the insightful discussions.

I extend my thanks to the dedicated staff at SISSA and ICTP for their assistance with administrative matters.

I also express my sincere gratitude to all my MHPC colleagues for fostering an international and friendly atmosphere, which significantly enriched the collaborative experience.

Lastly I want to express my profound appreciation to my family for encouraging me in this endeavor. You are very special for me.

Abstract

In the last decades, image processing has moved from academic research to innovative consumer applications. One of the most valuable of these practical uses is in object detection: from an image, identify and locate elements. One family of object detectors based on deep learning is known as YOLO. The goal of this work is to benchmark an object detection model based on YOLO version 5 regarding detection metrics as well as timing and energy consumption. We evaluate the performance of a vehicle license plate detector as well as examine what software frameworks and hardware resources are most suitable for the task. We present the literature about object detection and discuss performance metrics. We describe the dataset of license plates curated for the experiments and the training procedure. Then, we present performance results for different software stacks, with PyTorch as the baseline, and hardware equipment, CPUs and GPUs part of the DaVinci-1 cluster at Leonardo SpA and Intel Developer Cloud Beta. Besides that, energy consumption results are also discussed. Finally, we evaluated the effect of larger image sizes on the inference time as well as of grouping images in the same batch for processing.

Contents

1 Introduction **1**

2 Object Detection **3**
2.1 Performance metrics 6
2.2 YOLO 9

3 Dataset **19**
3.1 Training 23

4 Performance evaluation **27**
4.1 Warmup 28
4.2 Baseline 28
4.3 TensorRT 30
4.4 Intel optimization 31
4.5 Intel Max 1100 GPU 32
4.6 OpenVINO 37
4.7 Image size 37
4.8 Batch size 38

5 Energy consumption **45**
5.1 Training 46
5.2 Inference 47

6 Conclusions **49**

References **51**

List of Figures

1.1 Object detection for two classes: apple and bell pepper. 2

2.1 Input-output process for image recognition and object detection applications. 3
2.2 Example of semantic segmentation of an image: (a) original (b) segmented (from [4]). 4
2.3 General diagram of a region-based detection network (R-CNN and variants). 5
2.4 Example of five anchor boxes for one cell of the image. 6
2.5 Confusion matrix for binary classification. 7
2.6 Three hypothetical predictions: any of them have a large intersection with the ground truth (GT) annotation. 8
2.7 Visual representation of IoU and examples of values obtained. 9
2.8 Timeline of YOLO versions (adapted from [11]). 10
2.9 The multidimensional array at the output of YOLOv1 [15]. . . 11
2.10 General process taken for YOLO [15]. 12
2.11 Architecture of the neural network of YOLOv1 [10]. 13
2.12 Architecture of YOLOv5: backbone, neck, and head [16]. . . . 14
2.13 Part of the backbone of YOLOv5 model architecture [17]. . . . 15
2.14 Block BottleNeck 1 of YOLOv5 architecture [17]. 16
2.15 Block spatial pyramid pooling fast (SPPF) at YOLOv5 architecture [17]. 16
2.16 Neck of the YOLOv5 model architecture [17]. 17

3.1 Example of bounding boxes superimposed to an image. 20
3.2 Example of problems in the original dataset: annotated license plates are not identifiable. 22
3.3 Screenshot of the tool FiftyOne showing the tab with metadata fields and some samples of the image dataset. 23
3.4 Some example images of the validation set including the predictions. 25

3.5 Metrics from the model training with one class for vehicle registration plate. 26

4.1 Inference time for 50 runs of an array with dimensions [1024, 3, 640, 640] (batch size 1024) with model ResNet18 in a NVidia A100 GPU, without warmup. 29
4.2 Speedup of inference time for ResNet50 model in Intel Xeon 8480+ CPUs. 33
4.3 Zooming the speedup of inference time for ResNet50 model in Intel Xeon 8480+ CPUs. 34
4.4 Speedup of inference time for YOLO license plate model in Intel Xeon 8480+ CPUs. 35
4.5 Zooming the speedup of inference time for YOLO license plate model in Intel Xeon 8480+ CPUs. 36
4.6 Evaluating the influence of image resolution and optimization on the inference time for one core of the Intel Xeon 8480+ CPU. 39
4.7 Evaluating the influence of image resolution and optimization on the inference time for eight cores of the Intel Xeon 8480+ CPU. 40
4.8 Evaluating the influence of image resolution and optimization on the inference time for a full socket of the Intel Xeon 8480+ CPU. 41
4.9 Inference time per image for different batch sizes using Intel Xeon 8480+ CPU. 43
4.10 Inference time per image for different batch sizes using Intel Max 1100 GPU. 44

List of Tables

2.1 Comparison between different model sizes for YOLOv5 running in an AWS *p3.2xlarge* instance [13] 11

3.1 Metrics for the validation set after model training 24

4.1 Inference time for Intel 8260 CPU and NVIDIA A100 GPU . . 30
4.2 Average inference time for YOLOv5 license plate model and different optimizations using Intel CPU and GPU 33
4.3 Inference time for OpenVINO 38
4.4 Influence of batch size on inference time (from [44]) 42
4.5 Influence of batch size on inference time for YOLO license plate model 42

5.1 Energy consumption and training time using different devices 46
5.2 Energy consumption and inference time for different engines . 48

Chapter 1

Introduction

The advancements in camera technology and the availability of computational power transformed image processing from academic research to the realm of innovative applications. Foremost among these tasks is object detection, that aims to answer two questions given an image: what is present and where, as show in Fig. 1.1. This dual functionality is technically attributed to two operations: classification and localization. One can note in Fig. 1.1 that not all fruits and vegetables are identified; only two categories are annotated, that is, the classification domain is limited. Regarding localization, it is displayed by drawing rectangles, known as bounding boxes. This image also shows the challenges in developing an application for object detection: some fruits are occluded, the illumination varies, and the same class "apple" refers to elements with different colors.

Notwithstanding the difficult obstacles, the increase in object detection's significance can be attributed to its wide-range of practical applications. From autonomous vehicles navigating complex environments to surveillance systems ensuring public safety, the ability to accurately identify and locate objects is integral to numerous domains. Once elements are located, they can be counted and tracked, and this capability serves as the backbone for applications like augmented reality, medical imaging, and even wildlife conservation efforts, exemplifying its pervasive influence on contemporary technology.

When elaborating a new application, one can identify different phases in the software development life cycle [1]. Letaw [1] recognizes five stages: requirements, design, implementation, testing and maintenance. Other authors include one or two phases and name them differently, nonetheless testing is a integral part of any methodology. Relevant to this work is a very specific branch of testing, that is, software performance testing. In the most simple form, it should be accomplished to verify if the system meets requirements.

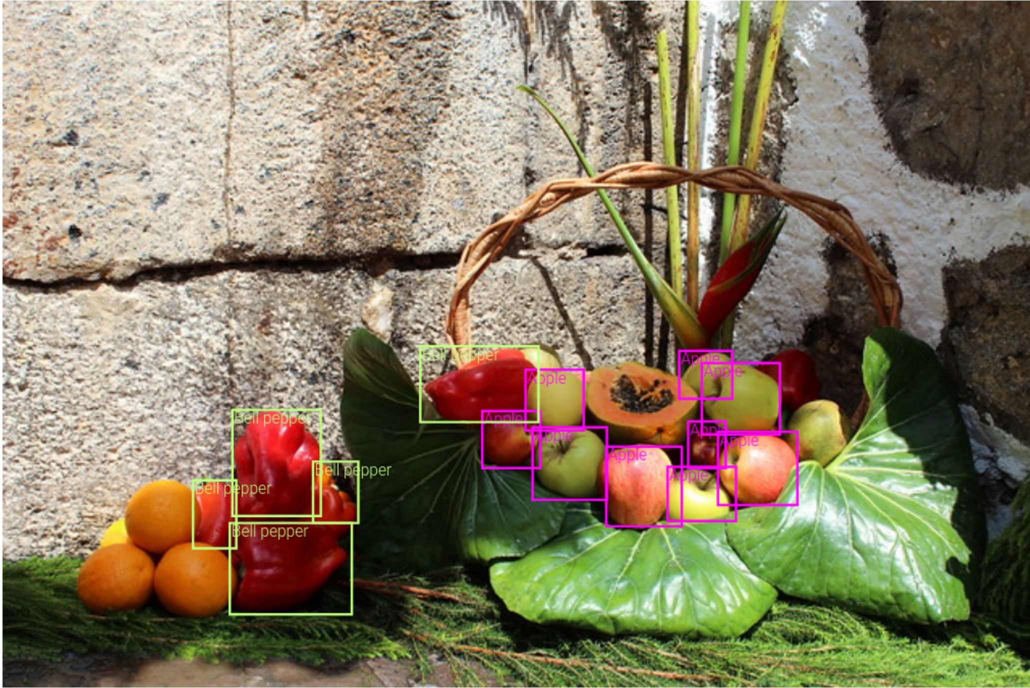

Figure 1.1: Object detection for two classes: apple and bell pepper.[1]

At the same time, it can also help to guide the implementation. In the case of this project, it serves twofold objectives: to verify the performance of an object detection application as well as to direct the use of new software frameworks and hardware resources.

Considering the importance of performance testing, the goal of this work is to benchmark an object detection application regarding detection metrics as well as timing and energy consumption. As an requirement from Leonardo SpA, the targets to be detected are vehicle license plates.

In Chapter 2 we discuss the state of the art in object detection. We start presenting a classification of methods, then we introduce performance metrics, and finally the now considered standard models using deep learning. Next, Chapter 3 describes the dataset of license plates we curated for the experiments and the training procedure. Chapter 4 is the main portion of this work, where we present performance results for different software stacks and hardware equipment. We set apart the energy consumption results in Chapter 5 and finally Chapter 6 presents conclusions and suggestions for future work.

[1] Original image from [2], license CC BY 2.0

Chapter 2

Object Detection

As introduced in Chapter 1, object detection applications are part of the great area of computer vision. They are expected to identify and locate objects within an image or video, providing extra information when compared to an image recognition (or classification) application. Fig. 2.1 shows that both techniques receive an image as input. A recognition task outputs the class names and the likelihoods or confidence scores of each class considering the whole image. On the other hand, object detection produces the class, location information, and the likelihood for each of the elements. To this extent, one can state object detection is more suitable to analyze realistic cases in which multiple objects may exist in an image.

In this work, the localization is given by a set of coordinates that define a rectangle around each object. Another possibility, not discussed here, is to use image segmentation, where each individual pixel is assigned to a corresponding class. Fig. 2.2 shows an example, where one can note that all chairs

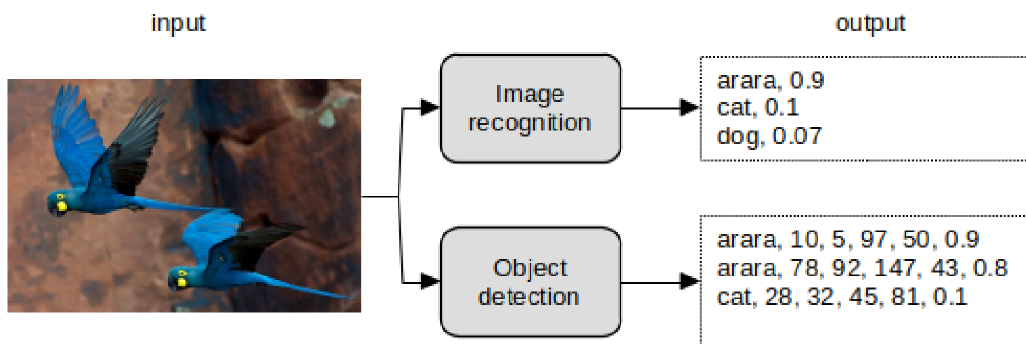

Figure 2.1: Input-output process for image recognition and object detection applications (original image of the macaws from [3], license CC BY 2.0).

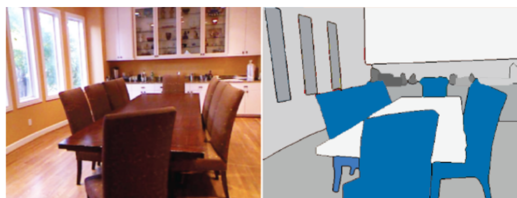

Figure 2.2: Example of semantic segmentation of an image: (a) original (b) segmented (from [4]).

are marked with the same color in the annotated image, that is, they are all considered part of the same class. This approach is also known as dense prediction, with useful applications in autonomous vehicles and medical image diagnostics.

Returning to the topic of object detection, algorithms presented in the literature can be categorized into three large classes:

- algorithms based on traditional computer vision, without deep learning
- two-stage deep learning based algorithms
- single-stage deep learning based algorithms

In the first class, algorithms that are now considered classic are Viola-Jones, proposed in 2001 and applied specially for face detection, and the histogram of oriented gradients (HOG). The last one experienced widespread use from 2005, although it was developed earlier, in the 1980s. They are still used for embedded applications, in systems with low computational power or where energy consumption is an issue.

The other two categories are based on deep learning. Although the roots of deep learning can be traced to the 1960s and 1970s, one can say that the turning point of this technology emerged in 2012, with an architecture called AlexNet winning the ImageNet Large Scale Visual Recognition Competition (ILSVRC or simply ImageNet). In this image classification competition, AlexNet achieved 15.3% of error rate while the second best competitor, 26.2% [5]. Krizhevsky et al. [5] claimed to have trained one of the largest convolutional neural networks (CNNs) of the time. The architecture is composed by eight layers: five convolutional layers plus three fully-connected ones. The output of the last layer has the same dimension of the number of classes to recognize and is fed to a softmax operator. In the case of [5], there are 1000 class labels.

Although applied to an image classification task, the work of Krizhevsky et al. [5] is important to mention because it inspired an object detection

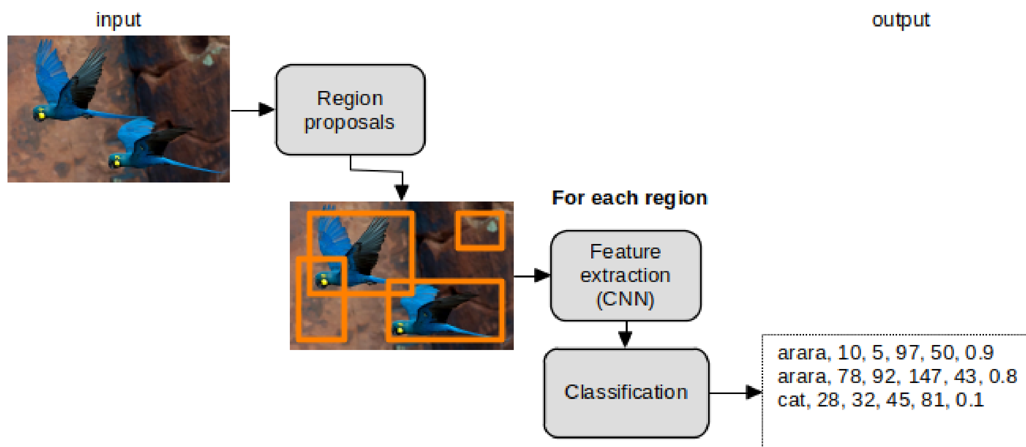

Figure 2.3: General diagram of a region-based detection network (R-CNN and variants).

work by Girschick et al. [6] in 2014. In this work, authors introduced Regions with CNN (R-CNN), an approach where a module generates region proposals that are category-independent, and from each region the CNN described by Krizhevsky et al. is used to extract a feature vector. Then, this vector is used as the input for a set of linear support vector machines (SVMs) that classify each region. The important point to mention is the two-stage nature: first propose areas of interest in the image and then classify each area, as shown in Fig. 2.3. In [6], selective search was used to extract a fixed number of 2000 regions. Other research works like the Fast-RCNN, Faster-RCNN and Mask-RCNN followed the same two-stage approach, each one introducing improvements regarding the speed of the overall application, changes in the way to obtain the regions of interest, or in the classifier.

Also based on deep learning, we have the last category: single-stage object detectors. Algorithms of this class treat object detection as a regression problem, that is, from the image the network should directly output the class probabilities and bounding box coordinates for a number of elements. They skip the region proposal stage and are faster than two-stage detectors. For the first versions, some dropping of accuracy was exchanged for speed. Currently the loss in accuracy is minimal.

One solution to avoid the costly region proposal stage is to rely on a set of predefined regions. Over the image, a grid is laid with boxes of different sizes and shapes. Fig. 2.4 shows an example of five anchor boxes (bounding box priors) for the center cell of the image. These are equivalent to the region proposals, but obtained in a simpler way. Then, for each box in each cell of the grid the model predicts whether an object exists. When

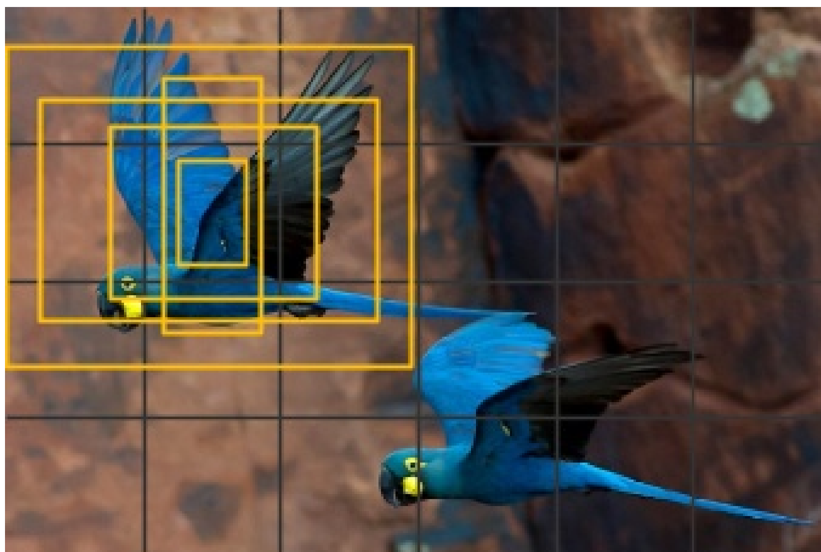

Figure 2.4: Example of five anchor boxes for one cell of the image.

using multiple boxes, overlapping detections will come out and some post-processing is needed for filtering them out.

Well-known single-stage object detectors in the literature are the single-shot detector (SSD), You look only once (YOLO), RetinaNet, CornerNet, CenterNet, and DETR [7]. The YOLO algorithm will be explored in Section 2.2.

2.1 Performance metrics

A very relevant topic to discuss is how to evaluate the performance of a model. Ideally, a metric should be easy to compute and express the quality of the system under test. We start discussing image classification metrics and then the particularities related to object detection.

The output of an image classifier is a label as well as the likelihood attributed to it, as already shown in Fig. 2.1. In that example, the class "arara" was identified with 90% of likelihood. Is this level of confidence enough to consider the result as valid? This is the first choice one needs to do. If we adopt a default threshold of 50%, "arara" would be considered a valid prediction to be compared with the ground truth labels. If we only accept as valid the detections with confidence greater than 95%, this output would not be considered. Using the detections with confidence greater or equal to a predetermined threshold, we can define the metric *accuracy*, a ratio of how

Predicted Class \ Actual Class	Yes	No
Yes	TP	FP
No	FN	TN

Figure 2.5: Confusion matrix for binary classification.

many predictions match the actual labels in relation to the total number of predictions. In addition, each result provided by a classifier falls into one of the four categories:

- true positive (TP): the predicted label matches correctly the ground truth
- true negative (TN): the classifier does not assign a label as a result and it is not part of the ground truth
- false positive (FP): when a predicted label is not part of the ground truth (also called type I error in statistical hypothesis testing)
- false negative (FN): a label that is part of the ground truth is not predicted by the classifier (type II error)

From these categories, one can derive a *confusion matrix* that aggregates the four measurements into a grid, as show in Fig. 2.5. A perfect system would have only values in the diagonal.

A problem arises when the classes are not balanced, as in the following example given by [8]. Consider that 1000 patients were screened for an eye disease and two of them actually have it. If one simply consider that all patients are healthy, the output of this "model" would be correct 998 times and the accuracy would achieve 99.8%. Although impressive, this result is a complete fail because the two cases were lost. For obtaining more insight about the results, it is advisable to use the metrics *precision* and *recall* defined, respectively, in 2.1 and 2.2.

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{2.1}$$

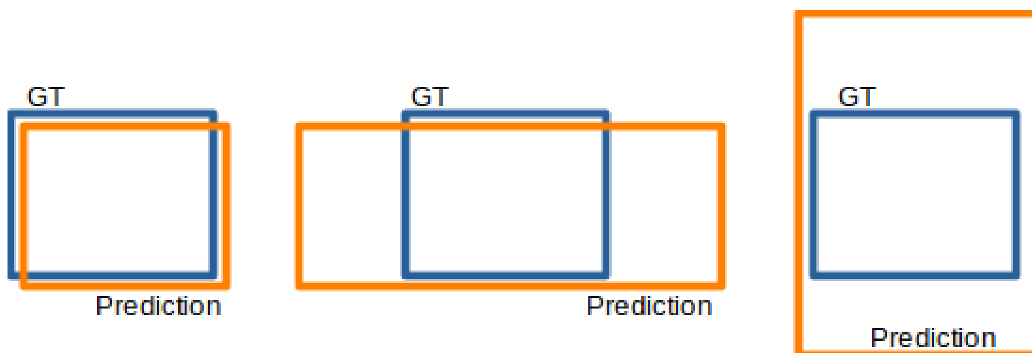

Figure 2.6: Three hypothetical predictions: any of them have a large intersection with the ground truth (GT) annotation.

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{2.2}$$

Another common metric is the harmonic mean of precision and recall, known as F1 score

$$\text{F1 score} = 2\frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{2.3}$$

Basically all these metrics also apply to object detection, comparing not the label for the entire image but each object. However, in object detection we are interested not only in the classification of an object in an image, but also in the correct localization of it. For this reason, more metrics should be taken into account. One could start measuring the area of overlap between the prediction and ground truth (GT), however, it is an inadequate metric if alone. Any of the predictions shown in Fig. 2.6 have a good intersection with the ground truth annotation but it is clear they are very different and some other metric should capture this fact. For this reason, it is common practice to compute also the area of union between prediction and ground truth and then obtain the *Intersection over Union (IoU)*, that is the ratio between the areas of overlap and union, as shown in Fig. 2.7. With perfect overlap, IoU is equal to one; without overlap, it is zero. Returning to Fig. 2.6, the first case will have a high IoU while the the second and third predictions will result in a much lower value, because of the larger area of union in the denominator.

Like the threshold for the predicted likelihood in image classification, IoU can be used as the value to consider a detection as valid. This way, when the predicted bounding box has an IoU equal or greater than the threshold, we count a true positive. A higher threshold requires a more accurate prediction.

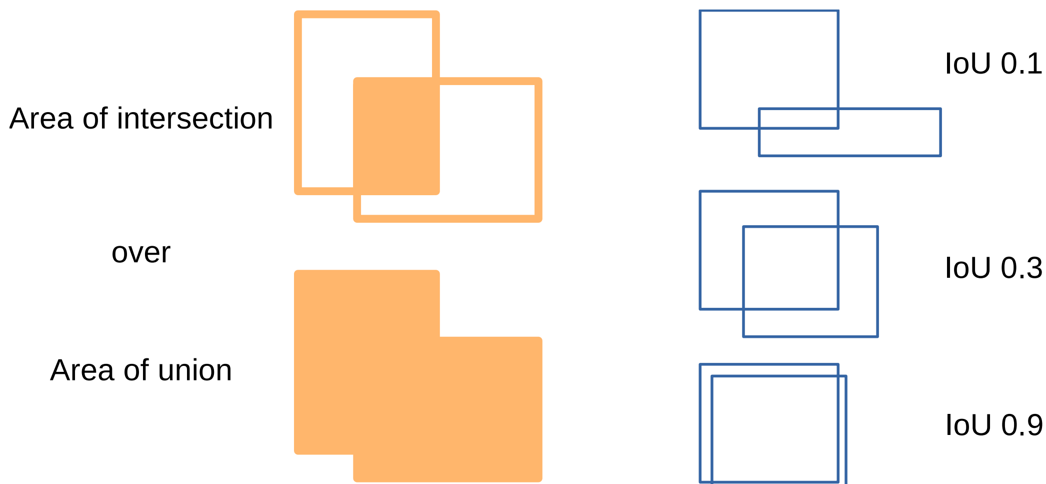

Figure 2.7: Visual representation of IoU and examples of values obtained.

With different thresholds for IoU, one can obtain different values for precision and recall and a precision-recall curve. A summary of the curve is computed as the area under it, usually with some previous smoothing [8], and this is called average precision (AP).

For multiclass problems, another reduction is computed as the mean of the average precision of each class (mAP). This metric is usually reported together with the threshold for IoU, for instance, mAP@50 or mAP@0.5 refer to a threshold of 50% for IoU. It is also common to report the mAP not for a single threshold, but for a range, for instance, for thresholds from 50% to 95%, with steps of 5%, written as mAP@.5:.05:.95, mAP@0.5-0.95, or mAP@50-95. A thorough discussion of many variations is presented by Padilla et al. [9]. The most famous competitions in the field provide their own tools to compute a performance metric.

2.2 YOLO

You look only once (YOLO) [10] is a family of single-stage object detectors, with the first version presented in 2016. What distinguished YOLO from the beginning was the good balance between speed and accuracy, making it suitable for real-time applications. In the PASCAL Visual Object Classes (VOC) Challenge, with 20 classes, the first version of YOLO (YOLOv1) obtained a mAP of 63.4 [10] keeping real-time performance of 45 frames per second (FPS). One version of a Faster R-CNN could achieve mAP of 73.2, but processing only 7 FPS, less than real-time for a standard video input.

YOLOX
YOLOR
PP-YOLOv2

YOLOv8
YOLO-NAS

YOLOv1

YOLOv3

2015	2016 //	2018 //	2020	2021	2022	2023

YOLO9000
YOLOv2

Scaled YOLO
YOLOv4
PP-YOLO
YOLOv5

DAMO YOLO
PP-YOLOE
YOLOv6
YOLOv7

Figure 2.8: Timeline of YOLO versions (adapted from [11]).

From 2016, YOLO has been actively developed, with new versions from different developers and different licenses [11]. Fig. 2.8 shows the timeline of the main versions and variants of YOLO from 2015 to 2023.

Because of the less restrictive license, this work refers to the version YOLOv5. The source code for training, validation, inference, as well as for exporting a model is available at Github [12]. Also there one can find pre-trained weights, that is, pretrained model parameters. To be precise, in this work we use YOLOv5 branch v7.0. Besides the specific version and branch, there are five architecture sizes, referred to the names nano, small, medium, large and extra large with the letters n, s, m, l, x, respectively. YOLOv5n is the smaller model, suitable for devices with limited computing capabilities. However its accuracy is lower than the other models. A comparison of accuracy and inference time provided in the official repository at [13] is presented in Table 2.1. Timings were obtained running the inference in an AWS `p3.2xlarge` instance. The letter "b" refers to the batch size (number of images that are processed simultaneously). The improvement when the batch size is increased from one to 32 is clear. The GPU is an NVIDIA V100. Number of parameters are indicated in millions. One can note that bigger the model, better the accuracy but larger the time for processing. From previous tests done at the company, the small model (YOLOv5s) has been chosen for this work.

A complete explanation of all YOLO versions and architectures is out of scope for this work, but a reader can refer to [11] as a thorough review from the v1 to versions proposed in 2023, including variants with visual transformers.

The common idea for all versions is to split the image into grid cells, predict bounding boxes and select the best one regarding confidence score and coverage (IoU) [14]. This way, an input image is divided into a $S \times S$ grid with B bounding boxes plus confidence for C different classes per grid element. The bounding box is described by four values: the coordinates (x, y)

Table 2.1: Comparison between different model sizes for YOLOv5 running in an AWS *p3.2xlarge* instance [13]

Model	mAP50-95	Inference time per image(ms) CPU b1	GPU b1	GPU b32	params.(M)
nano	28.0	45	6.3	0.6	1.9
small	37.4	98	6.4	0.9	7.2
medium	45.4	224	8.2	1.7	21.2
large	49.0	430	10.1	2.7	46.5
extra-large	50.7	766	12.1	4.8	86.7

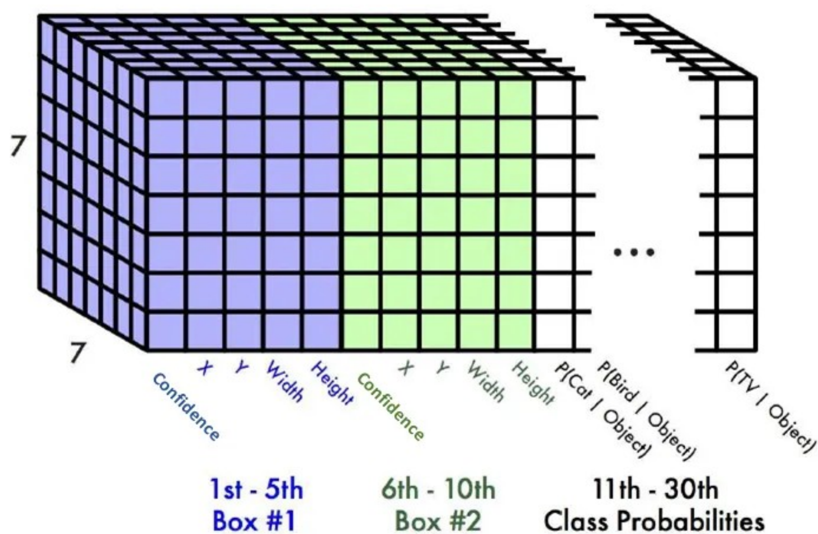

Figure 2.9: The multidimensional array at the output of YOLOv1 [15].

of the center of the box relative to the bounds of the cell, plus the width, and height relative to the whole image. The output is a multidimensional array with dimensions $S \times S \times (B \times 5 + C)$. For YOLOv1, as shown in Fig. 2.9, the grid is 7×7, with at most 2 boxes per grid element ($B = 2$) and 20 classes in the PASCAL VOC dataset. So, the output has dimensions $7 \times 7 \times 30$.

At this point, bounding boxes probably overlap. So, it is necessary to include a post-processing step with an algorithm called non-maximum suppression (NMS). In a iterative way, it retains the detection candidates with highest confidence scores, filtering out all other candidates that overlap with the best ones. This is the reason for the name: it suppresses all candidates that have non-maximum scores. A general view of the process taken for YOLO is shown in Fig. 2.10. The image is divided into a grid, bounding

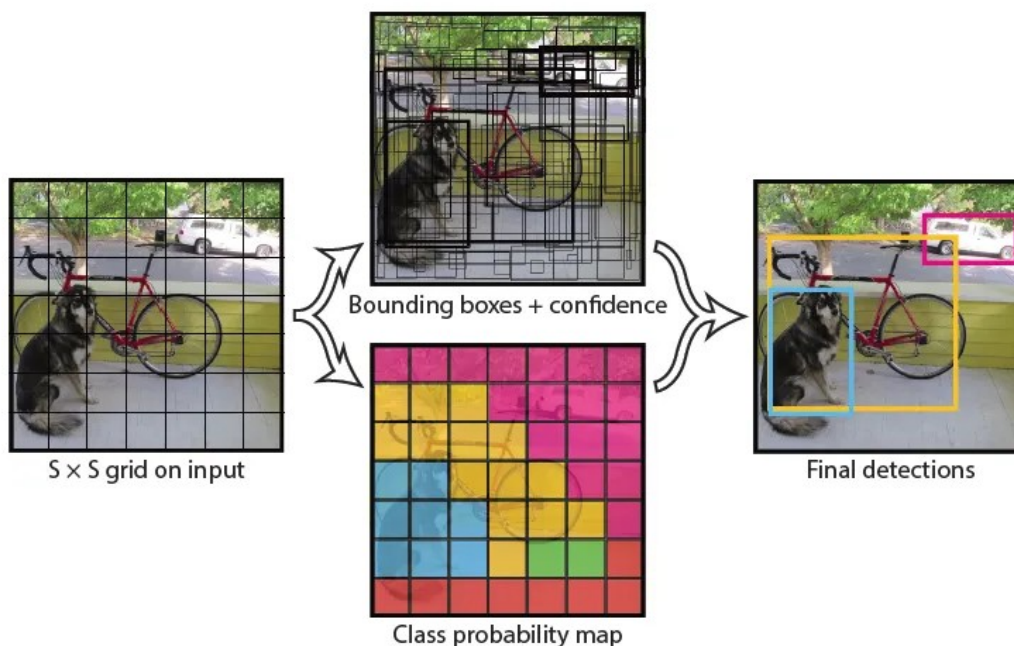

Figure 2.10: General process taken for YOLO [15].

boxes are generated, then analyzed together with the class confidence scores to obtain the final detections. Each color in the class probability map indicate the class with higher score.

For YOLOv1, the output layer ($7 \times 7 \times 30$) is the result of a sequence of 24 convolutional layers plus 2 fully connected ones, as shown in Fig. 2.11.

All the subsequent YOLO models apply strategies to facilitate training, as well as improving the performance (accuracy and inference time). The architecture of the convolutional layers has changed, as well as the number of candidate bounding boxes and the way to generate them. Since the model architectures for subsequent versions of YOLO present many more layers, one can divide them into three parts: backbone, neck, and head. The backbone is basically a set of convolutional layers plus normalization and activation functions for feature extraction. The neck includes layers or other structures to improve the feature representation [11]. The head is the final stage where the predictions are generated, giving as output the classes, confidence scores and the bounding boxes. Fig. 2.12 shows the architecture of YOLOv5. Since the structure is too large for visualization, here we will zoom into each part. Starting with the backbone in Fig. 2.13, the default image is 640x640 with 3 channels for colors. The block called `ConvBNSiLU`, also known as CBL, is a convolutional layer followed by batch normalization and a sigmoid linear

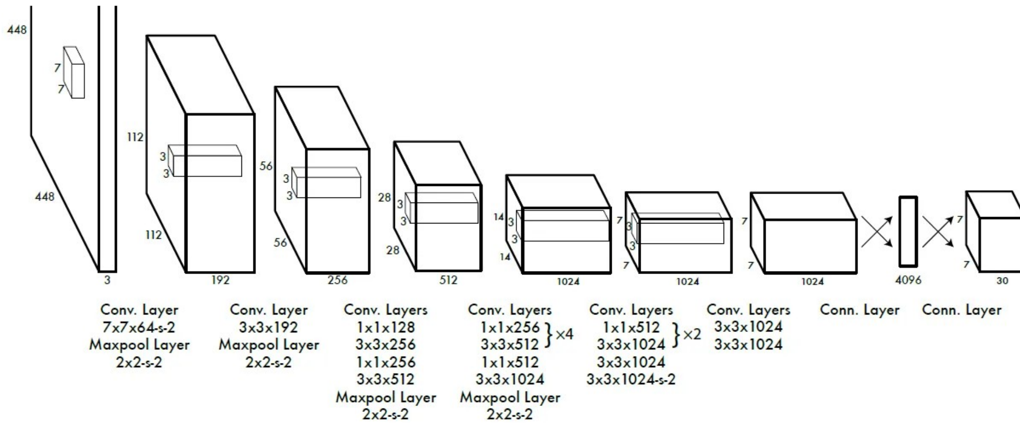

Figure 2.11: Architecture of the neural network of YOLOv1 [10].

unit (SiLU) activation function. In Fig. 2.13, the parameters are shown as:

- kernel size k
- stride s
- padding p
- output channels c

For instance, the first CBL block (P1) applies a convolutional layer with kernel size 6, stride 2, padding 2, and 64 channels. After P2, follows a sequence of four blocks named C3 intercalated with CBL blocks.

Inside each C3 block, there is another block called BottleNeck 1, shown in Fig. 2.14. This is a residual block with two skipped connections, considered better for gradient back-propagation [16]. The C3 blocks are a variant of cross stage partial network (CSPNet) blocks [18], that provide better behavior for backward gradient computation. The last CNN block in the backbone is a spatial pyramid pooling fast (SPPF), used as feature reaggregator [16], shown in Fig. 2.15.

The neck of YOLOv5 is formed by four C3 blocks, CBL blocks, upsample, concatenation and a block named BottleNeck 2, as shown in Fig. 2.16. They are combined to form a path aggregation network (PANet), considered a solution to improve the flow of information between different layers [16]. The BottleNeck 2 has the same two convolutions as the BottleNeck 1, but without the skip connection.

The head of YOLOv5 model is the same of YOLOv3. It has three detection layers, each with a different number of grid cells (obtained by means of

Figure 2.12: Architecture of YOLOv5: backbone, neck, and head [16].

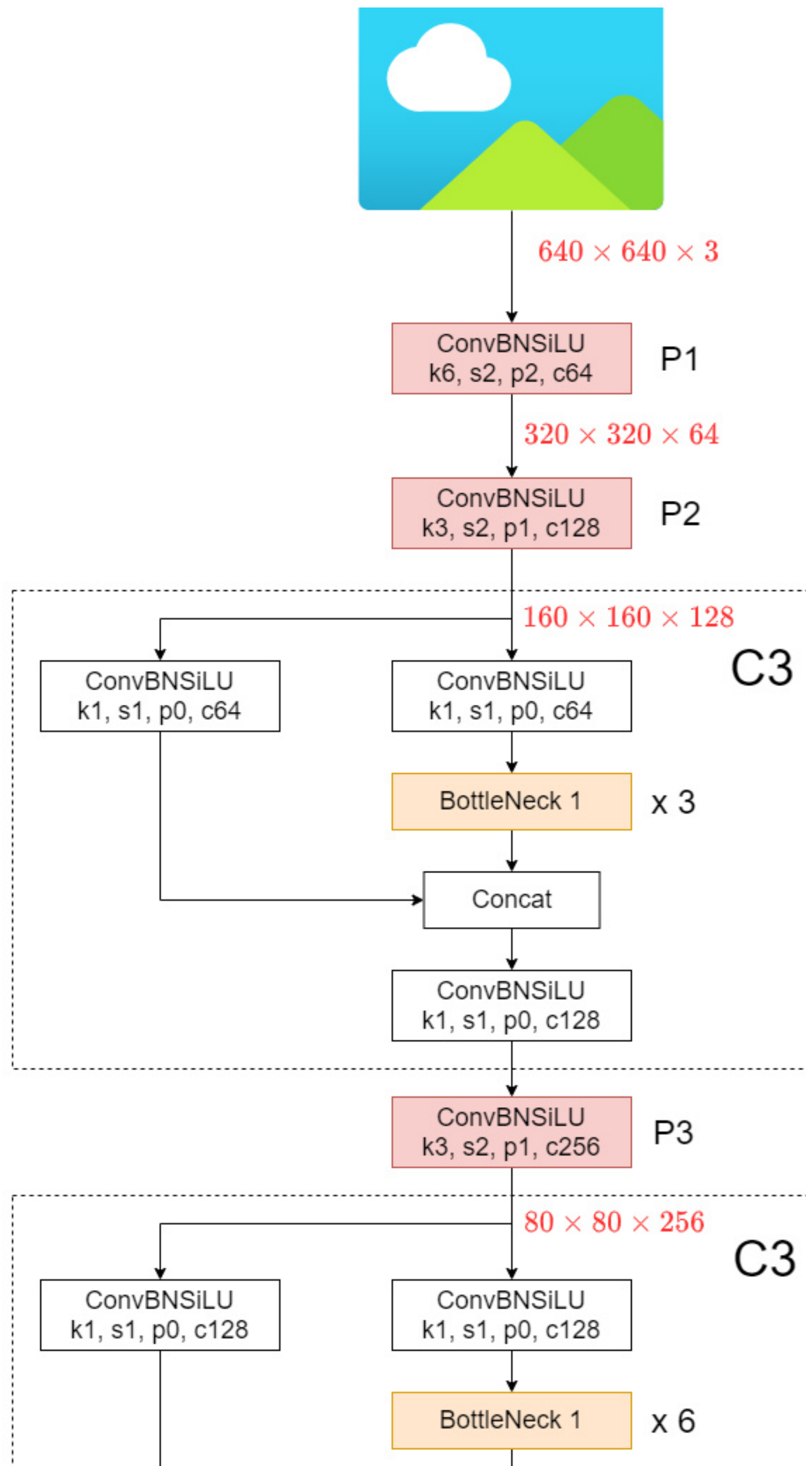

Figure 2.13: Part of the backbone of YOLOv5 model architecture [17].

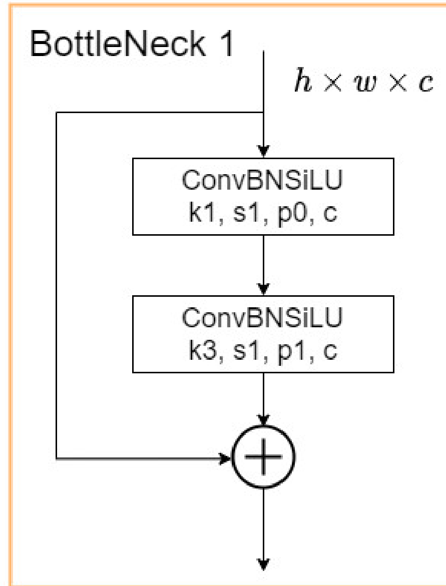

Figure 2.14: Block BottleNeck 1 of YOLOv5 architecture [17].

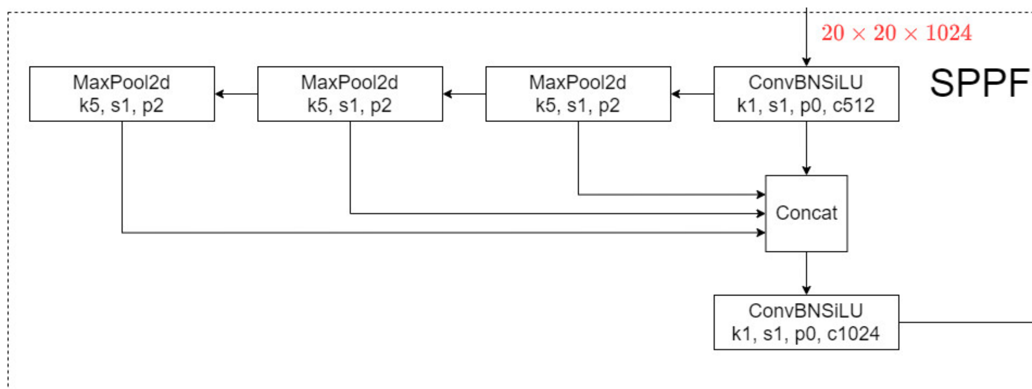

Figure 2.15: Block spatial pyramid pooling fast (SPPF) at YOLOv5 architecture [17].

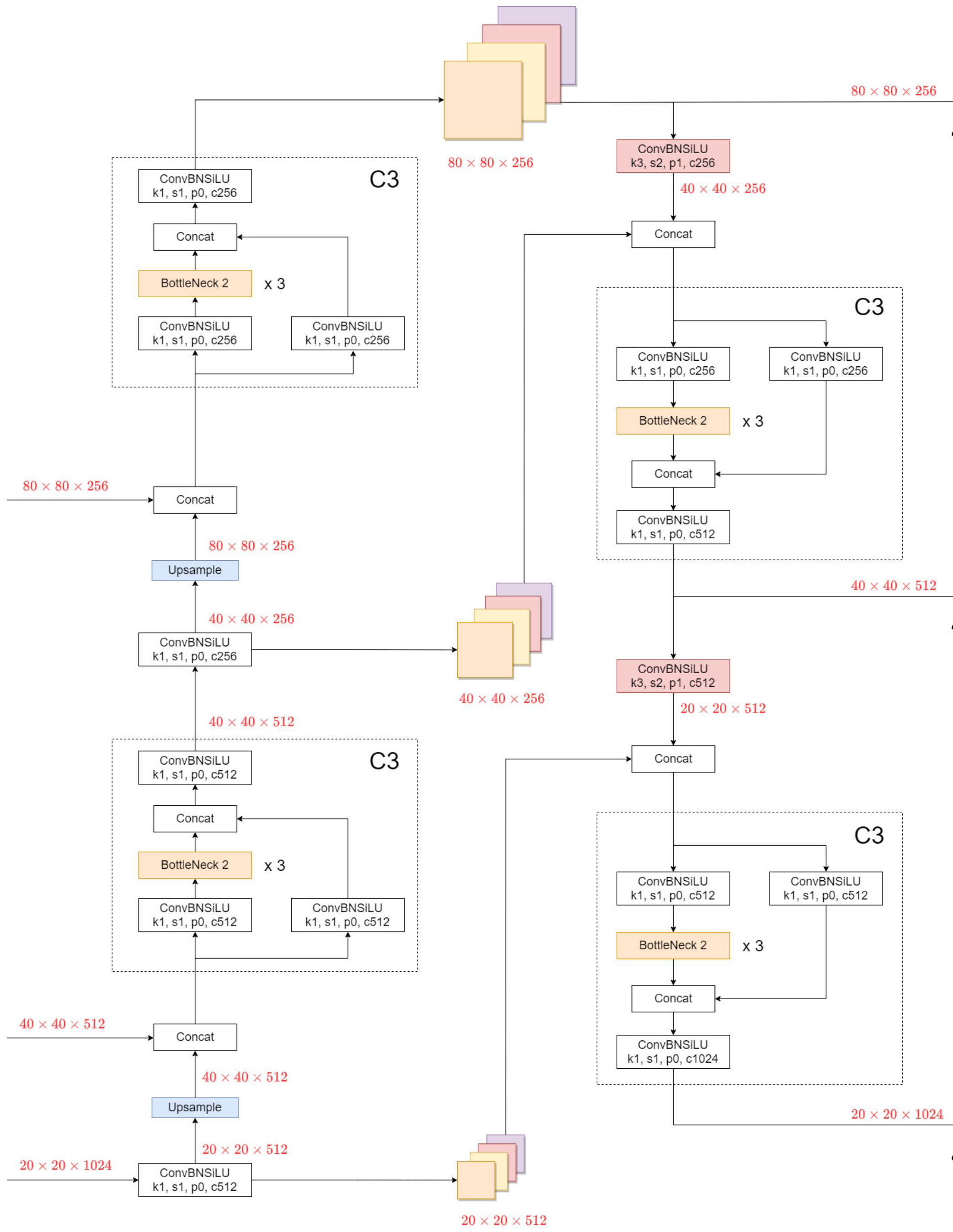

Figure 2.16: Neck of the YOLOv5 model architecture [17].

the parameter S discussed in the beginning of this Section). Since each cell can detect at most one object, this multigrid approach allows detections at different scales, alleviating a weakness of the original YOLO regarding spotting small objects. For an input image with resolution of 640×640 pixels, the grid cells of each of the three detection layers represent 8, 16, and 32 pixels in width and height.

Regarding the training of YOLOv5, it is worth mentioning that the loss function is computed as the weighted sum of three components:

- class loss, calculated using binary cross entropy (BCE), is the general classification error
- objectness loss, also from BCE, measures the error in finding an object in a grid cell
- localization loss, obtained through complete IoU (CIoU), related to localizing the object within the grid cell

Chapter 3

Dataset

One fundamental part of supervised learning is obtaining data for training as well as for validation. The data consists of not only images but also associated labels. For object detection, the labels, also known as image annotations, comprise the localization of objects in the image and the class or category of the object. There are different annotation approaches, the one we use here is the bounding box, that is, rectangular boxes, as shown in Fig. 3.1.

There are many annotation formats, saved in different file types. For instance, the dataset Common Objects in Context (COCO) uses .json, PASCAL VOC uses .xml, and YOLO uses a .txt file with the same name of the image, with one line for each object, like

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

where

- `<class_id>` is an integer that represents the class of the object. The class index should start from 0. Each unique class in the dataset has an `class_id`.
- `<x_center>` and `<y_center>` are the coordinates of the center of the bounding box, normalized by the width and height of the image, respectively. This way, the values are in the range of [0, 1].
- `<width>` and `<height>` are the width and height of the bounding box, also normalized by the width and height of the image, respectively, and in the range of [0, 1].

Since one of the requirements of this project was to test the detection of license plates, we started evaluating some datasets that included this kind of elements:

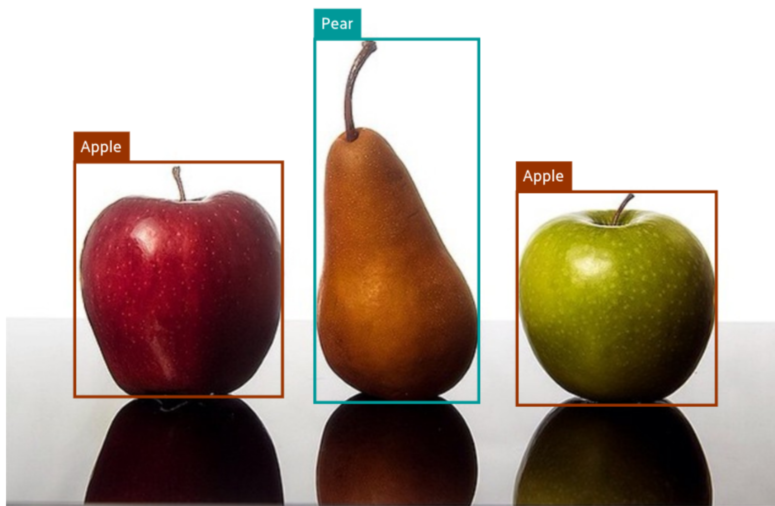

Figure 3.1: Example of bounding boxes superimposed to an image.[1]

- "Car License Plates Dataset" with 433 images and bounding boxes (PASCAL VOC format) [19]
- Annotation of 2026 character bounding boxes for 209 license plate images [20]
- UFPR-ALPR dataset, which includes 4,500 fully annotated images (over 30,000 LP characters) from 150 vehicles [21]

Other datasets are mentioned in the literature. Laroca et al. [22] uses eight publicly available datasets: Caltech Cars, EnglishLP, UCSD-Stills, ChineseLP, OpenALPR-EU, AOLP, SSIG-SegPlate and UFPR-ALPR. The same authors also mention some problems with duplicates in train and test subsets [23].

After some preliminary evaluations and tests, we decided to use the Open Images Dataset V6 [25]. In total, it has more than 16 million bounding boxes annotated in 1.9 million images. Of the 600 classes, there is one specific for "Vehicle registration plate". Besides object detection, this dataset can also be used for other applications, since it has annotations for object segmentation, visual relationships (e.g. "woman playing guitar"), and localized narratives.

Open Images is divided into train, test and validation subsets, with almost all bounding boxes manually drawn by professional annotators. The license

[1]Original image from [24], license CC BY 2.0

of the annotations is CC BY 4.0, while images are listed as conforming to CC BY 2.0. Both of them allow commercial use.

One could simply download the entire dataset and write scripts to filter it. However, given the massive size, a more adequate option is to consider some tool to make evaluation and visualization more friendly. Here we use FiftyOne [26], a Python tool for visualizing datasets, with an open-source core library.

Then, to download the three sets (train, validation and test), and filtering them to obtain only the images with at least one label "Vehicle registration plate" becomes as simple as shown in Listing 3.1.

```
import fiftyone as fo
import fiftyone.zoo as foz

classes = ["Vehicle registration plate"]

train_val_test_dataset = foz.load_zoo_dataset(
    "open-images-v6",
    splits=["train", "validation", "test"],
    label_types=["detections"],
    classes=classes
)
```

Listing 3.1: Download dataset and filter label using FiftyOne

Applying this first filtering, we obtain 5368 images for training, 724 for validation, and 2065 for test. The number of labels is greater than the simple sum of the images, since some images have more than one label. This way, in 8157 images, there are 11682 occurrences of license plates.

We will discuss in Section 3.1 the strategy to train new models. In a summary, we point out that, since we can use pretrained weights, the need for data is reduced and this number of images is enough.

In fact, after some preliminaries tests and inspection of the dataset, we noticed that some "garbage" is introduced for training and it affects the overall performance. The detection is only the first step of a full automatic license plate recognition (ALPR) system, so labels where the plate is too small, too far away, or in an angle that is almost impossible to identify where excluded. In Fig. 3.2 we show some examples of these problems.

Besides the class identification and bounding box coordinates, Open Images has some metadata information for each object, as shown in Fig. 3.3. Then, we created a script to obtain only the objects that:

- are not occluded
- are not truncated

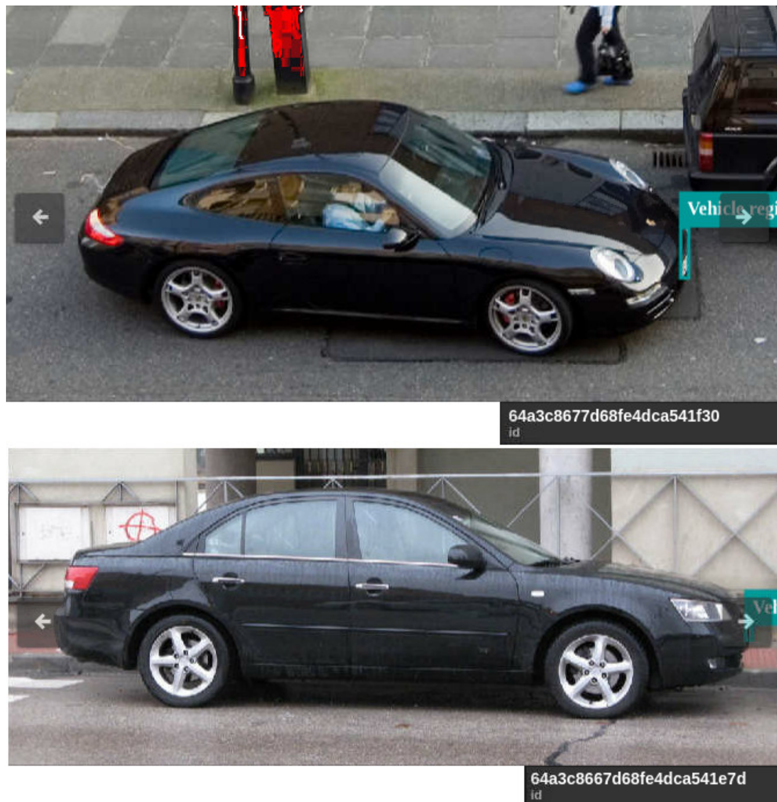

Figure 3.2: Example of problems in the original dataset: annotated license plates are not identifiable.

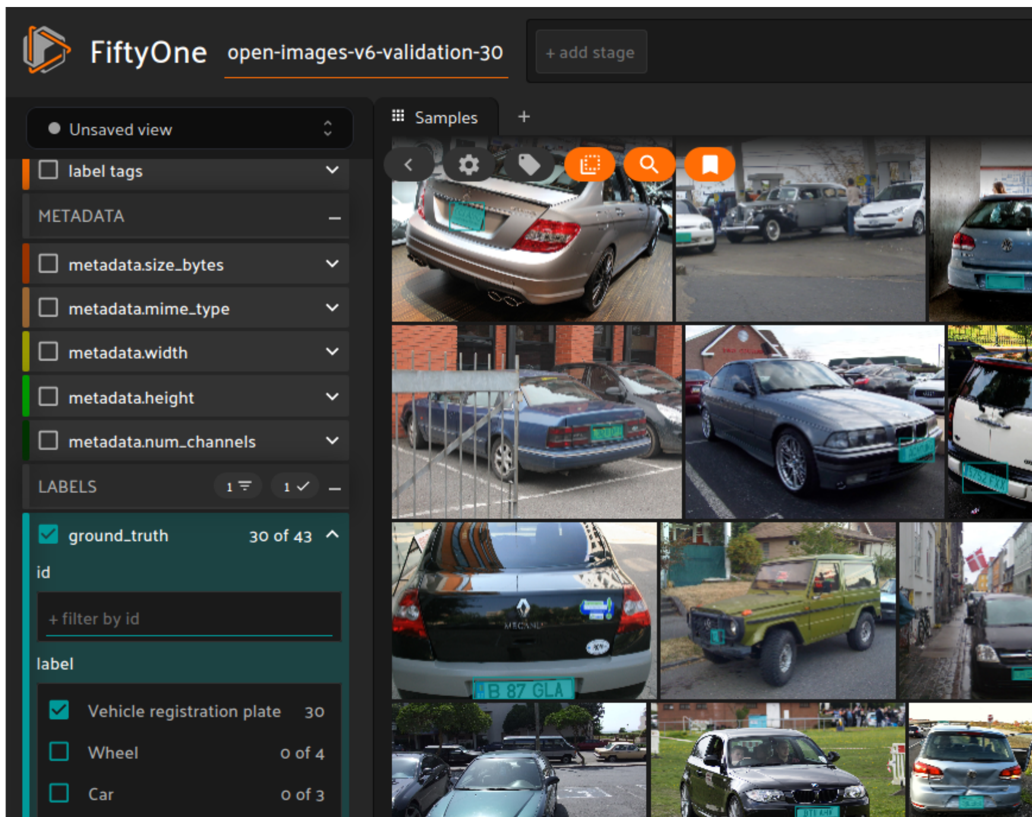

Figure 3.3: Screenshot of the tool FiftyOne showing the tab with metadata fields and some samples of the image dataset.

- have a minimum aspect ratio of 2, favoring license plates that appear rectangular in the images
- present a minimum bounding box area of 0.08% of image area

For an image with 640 x 640 pixels, 0.08% of the area represents, for instance, a license plate with 25 x 13 pixels.

After these filtering step, the number of images is 2300 in the training split, 352 for validation and 988 for testing.

3.1 Training

The authors of YOLOv5 provide a script for training in `train.py` available at [27]. It loads the weights of a pretrained model and starts the optimization process, that uses stochastic gradient descent. The difference between the

Table 3.1: Metrics for the validation set after model training

Metric	Value
Precision	0.913
Recall	0.938
mAP50	0.960
mAP50-95	0.698

default model and this new one is the number of output classes that changes to only one. We did a single-GPU (NVIDIA A100) training at the cluster DaVinci-1, with 25 epochs, that took around 6 minutes to complete. The model's size is 14.4 MB, with 157 layers and 7,012,822 parameters. The validation results are summarized in Table 3.1. In the set with 352 images, 371 instances of license plates were detected.

Following the standard practice in the area, the test split is used only at the end of the training process, as a final check. The visualization tool FiftyOne was again used, helping to gain insight about the results.

Fig. 3.4 shows a mosaic of 16 images from the validation set together with the predictions. There are 17 plates annotated in the ground truth, and all are correctly detected. There is also an additional detection for the car's plate in the background of image fc0f46431cb1dbe9 (1st row, 2nd column).

The metrics from the training with 25 epochs are shown in Fig. 3.5. The loss is split between box (localization) and object (presence in image) and also between train and validation. One can notice the loss decreases for both sets and objectives. As expected, the box loss is higher that the object loss. From the training, it seems we could run some more epochs, because a plateau was not achieved. On the other hand, the validation and the mAP seem already adequate.

The report for the filtered test set is shown below where support means the number of annotated elements. The number 1060 is larger than the number of images (988) because some present multiple license plates.

```
                        precision  recall  f1-score   support
Vehicle registr.plate        0.88    0.94      0.91      1060
mAP = 0.92
Confusion matrix:
                Prediction
              Vehicle reg. plate    none
Truth plate   992                   68 (miss or false negative)
      none    136 (false positive)  0
```

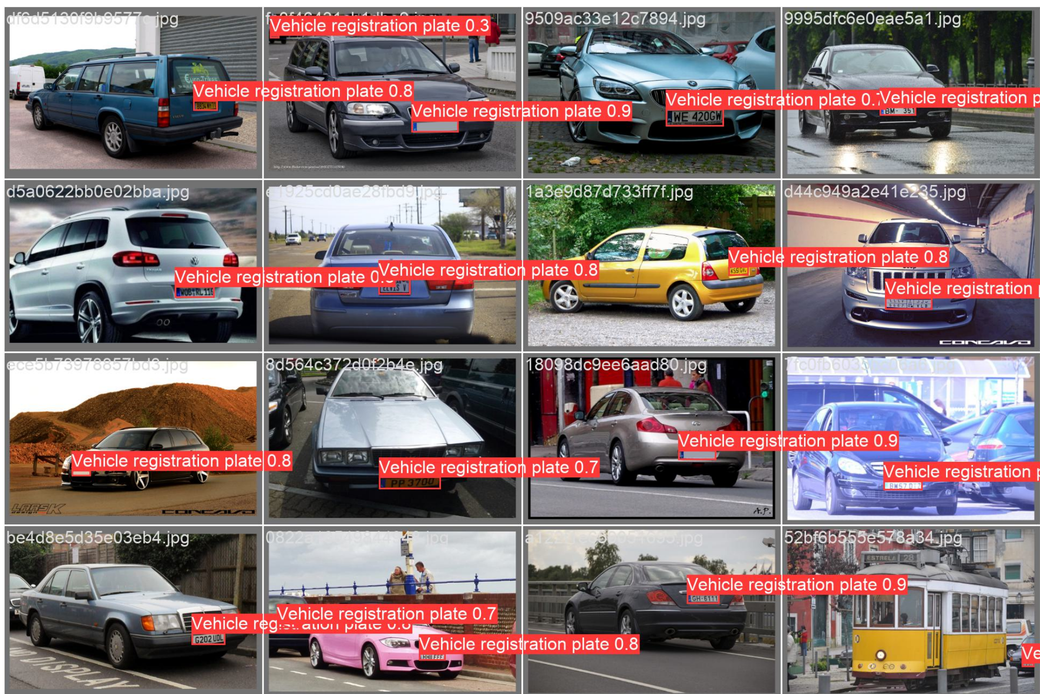

Figure 3.4: Some example images of the validation set including the predictions.

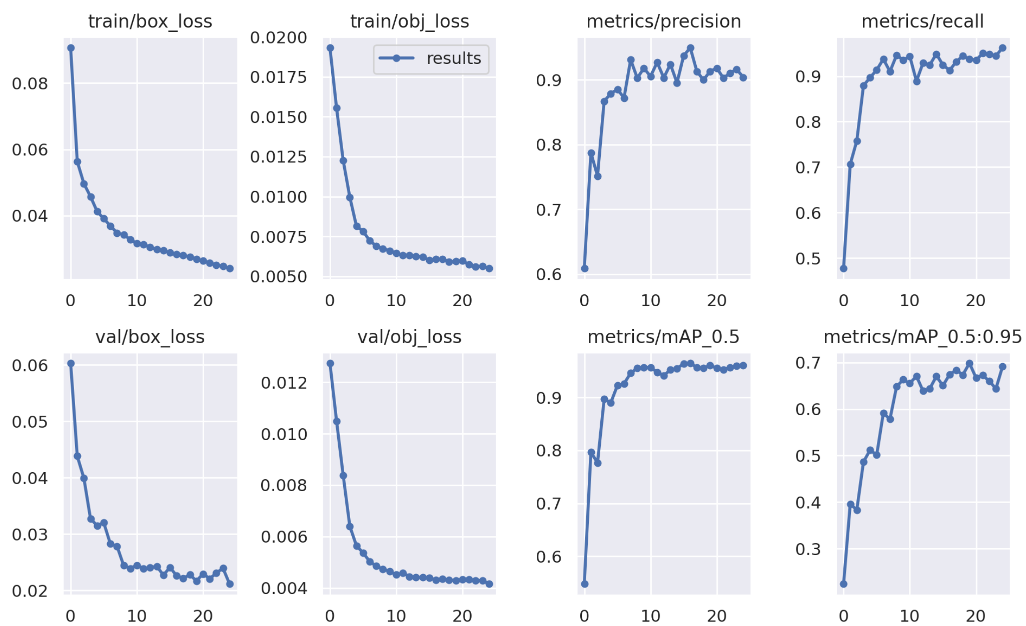

Figure 3.5: Metrics from the model training with one class for vehicle registration plate.

Chapter 4

Performance evaluation

In this Chapter we present the performance evaluation of neural networks for object detection, specially the model trained for detecting license plates discussed in Chapter 3. The main focus of this Chapter is on timing the inference operation while Chapter 5 discusses energy issues. Tests were conducted using different software frameworks and hardware setups described below.

The first hardware equipment is the cluster DaVinci-1 [28] at Leonardo Company, with two partitions: CPU and GPU. The software for cluster management is Altair PBS. Each node in the CPU partition has two sockets of Intel Xeon Platinum 8260 CPUs with 24 physical cores per socket. These CPUs were known as Cascade Lake, with full specifications available at [29]. The nominal clock is 2.4 GHz, with a maximum of 3.9 GHz. The total RAM memory is 768 GB. Nodes in the GPU partition of DaVinci-1 have two sockets of AMD EPYC 7402 CPUs (specifications at [30]), with 24 physical cores per socket, 512 GB of memory plus 4 GPUs NVIDIA A100-SXM4 with 40 GB of memory [31].

We also conducted tests in the Intel Developer Cloud Beta [32], where each compute node has two sockets of Intel Xeon Platinum 8480+ CPUs, with 56 physical cores per socket, and 512 GB of RAM memory. This CPU was formerly known as Sapphire Rapids, with base frequency of 2.0 GHz and maximum clock of 3.8 GHz. Full specifications are available at [33]. Each node also has an Intel Data Center GPU Max 1100, with 48 GB of memory. This GPU was previously named as Ponte Vecchio and the full specifications can be found in [34]. The company made available the Intel Developer Cloud Beta on July 2023 and, for limited time, the system could be accessed without additional costs. Slurm is the job scheduler in this cluster.

Regarding the software, PyTorch [35] version 2.0 was employed as the baseline framework. From it, a model can be exported to other inference

engines, each one with optimizations for CPUs or specific GPUs. They are discussed in the next sections. We evaluated:

- Intel Optimization for PyTorch (IPEX)
- NVIDIA TensorRT
- Intel OpenVINO
- TorchScript

Besides that, we also evaluate the influence of batch size (number of images that are processed as a block) and image resolution.

4.1 Warmup

A very important consideration when doing timing analysis is the starting state of the computing device. For energy saving reasons, CPUs and GPUs can reduce their power levels, shutting down subsystems of the hardware. When some computing job is started, the awaking process from these low power states can consume very significant time. For instance, when timing 50 runs of inference of a ResNet18 model, with batch size 1024, using a NVidia A100 GPU, we obtained an average time of 379 ms. However, when checking each time measurement of the 50 runs, as shown in Fig. 4.1, one can note there is some accommodation period until results start to be more stable. While the second run takes more than 8 s, after the fourth run results are very similar, around 78 ms.

For this reason, it is common practice in the industry to not include the first runs when doing timing benchmarks. It means to include a warmup stage with random data as input with the same dimensions as the true data. Following the recommendations from [36], [37], we included this warmup stage before the time measurements. Another recommended practice is to average the time of a large number of runs and to fix the device clock. Unfortunately, we were not able to setup the clock in any of the systems under analysis because this was an operation restricted to the administrator.

For CPUs the warmup is not so critical, nonetheless it was also applied.

4.2 Baseline

At DaVinci-1, the YOLO license plate model with batch size 1 has achieved the results presented in Table 4.1.

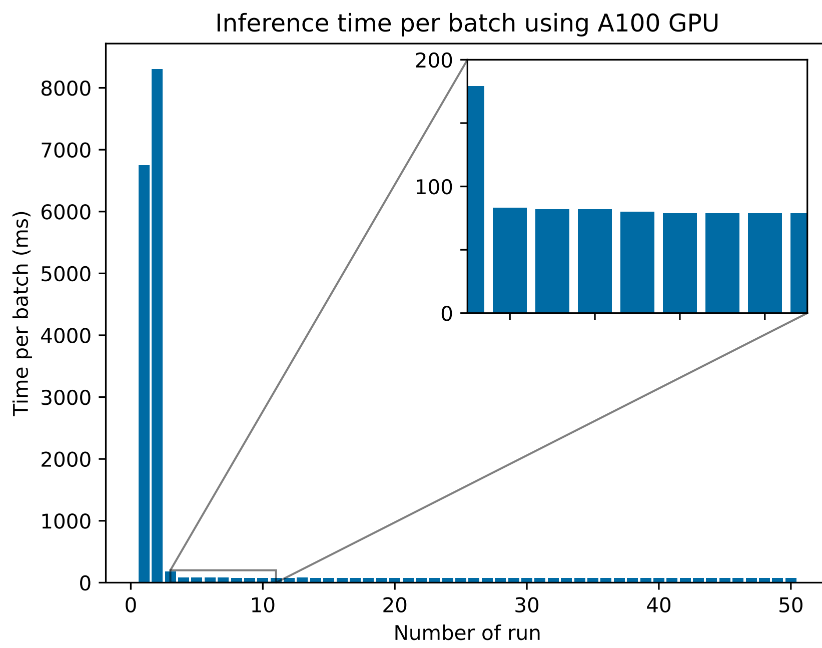

Figure 4.1: Inference time for 50 runs of an array with dimensions [1024, 3, 640, 640] (batch size 1024) with model ResNet18 in a NVidia A100 GPU, without warmup.

Table 4.1: Inference time for Intel 8260 CPU and NVIDIA A100 GPU

Runtime + Cores	Inf. time (avg.) (ms)
PyTorch CPU 1	111.5
PyTorch CPU 8	28.2
PyTorch CPU 24	25.0
PyTorch CPU 48	33.1
PyTorch GPU	7.4

One can note that using more cores the time decreases but not as a linear scaling. The speedup is 3.95 times for 8 cores and 4.46 times for 24, considering one core as the reference. When changing from 24 to 48 cores, that is, starting to use the two sockets of the node, timing becomes worst. Using the GPU, the speedup is 3.38 taking as reference one socket of the CPU.

4.3 TensorRT

TensorRT is a software development kit (SDK), provided by NVIDIA, for optimizing inference [38]. The most simple approach it applies is to fuse operations. It can also be used to convert models to different numerical representations, like half precision (float16) and int8. First results we present here are from applying the YOLO `export.py` script with default parameters to convert a model from original `.pt` to a TensorRT file. The conversion is not direct, it passes to an intermediate ONNX model. It has a slight decrease in detection performance (mAP50-95 from 0.6320 to 0.6318) compensated by a huge gain in inference time. In our tests, it provided the best results when compared with any other of the engines under test. With TensorRT, the inference time is 1.5 ms for NVIDIA A100, while PyTorch takes 7.4 ms in the same GPU and 25 ms in a socket of the 8260 CPU (Table 4.1).

As explored in Section 4.8, another gain can be obtained by increasing the batch size. For TensorRT, the only issue is that each model is exported for a fixed dimension, so more than one model is needed if doing inference with different batch sizes.

4.4 Intel optimization

Intel has released a package called Intel Extension for PyTorch (IPEX) to optimize the performance of PyTorch on Intel hardware (CPUs and GPUs) [39]. Current Intel CPUs are equipped with the AVX-512 Vector Neural Network Instructions (AVX512 VNNI) and Advanced Matrix Extensions (Intel AMX). However, to deliver the best performance, some support from the software stack is needed and this is the main goal of IPEX. For this work, we used the version 2.0.110+xpu, that applies both for CPUs and discrete GPUs (according to Intel, "the 'X' in 'XPU' stands for any compute architecture that best fits the need of your application" [40]).

Applying the optimization to a previously trained model is simple. One needs to load the module and call the `optimize` function, as shown in lines 3 and 8 of Listing 4.1.

```
1  import torch
2  import torchvision.models as models
3  import intel_extension_for_pytorch as ipex
4  device="cpu" # or "xpu"
5  model = models.resnet50(weights="ResNet50_Weights.DEFAULT")
6  model.eval()
7  model = model.to(device)
8  model = ipex.optimize(model)
9  with torch.no_grad():
10     d = torch.rand(1, 3, 224, 224)
11     d = d.to(device)
12     model = torch.jit.trace(model, d)
13     model = torch.jit.freeze(model)
```

Listing 4.1: One solution to optimize a model with IPEX [39].

However, much better results can be obtained combining it with what is called *graph mode*. Using PyTorch, a neural network can be executed in eager or graph mode. In eager mode, each operation is immediately performed, what is very convenient for an interactive environment, when experimenting with different architectures and parameters. However, this approach is not the most efficient because optimizations are better achieved only after the full graph is built [41]. As a simple example (more advanced strategies are discussed in [42]), one can consider matrices A, B, and C, and the operation $AB + AC$. Using the mathematical distributive property, it would be more efficient to carry out $A(B + C)$, but this is not possible in the eager mode. However, if all operations are firstly tracked, a computational graph can be built and then optimized. The process of traversing a set of operations on tensors is called tracing. Then, a *just-in-time compiler* (with TorchScript) can be applied, being recommended by Intel. The call of the tracing routine

to record the operations performed on tensors is executed in line 12 of Listing 4.1, with some data as input (created in line 10). After that, in line 13, the call to `freeze` optimizes the model.

One of our experiments evaluated the gains when using this module. For this, we tested two models: Resnet50 and our own trained YOLO license plate detection. Many benchmarks use Resnet as an standard model, because it has many convolutional layers. Another reason to use it is that we would like to see if optimizations also scale to different architectures.

Using one core of the Intel 8480 CPU, the baseline ResNet50 model, directly from `torchvision`, takes 65.5 ms for inference. There is some improvement when applying IPEX optimization and tracing, specially when using more cores of the same CPU, as shown in Fig. 4.2. Zooming the area from 1 to 20 cores (Fig. 4.3), one can note that tracing provides linear scaling, showing how advantageous is to apply such process. Returning to Fig. 4.2, it is interesting to note that including more cores is not always beneficial, specially when we start using both sockets of the same motherboard. Probably there are issues when accessing memory that is not close to the socket.

Similar behavior happens for the YOLO license plate model shown in Fig. 4.4. When using 32 cores, there is an improvement of around 6 times compared to the baseline result for 1 core (126.2 ms). Still for 32 cores, after applying IPEX+tracing, a speedup of around 3 times is achieved, and the overall speedup from the 1 core baseline model is almost 18 times. The improvement using the Intel optimization plus tracing is also clear in Fig. 4.5, where we zoom the area between 1 and 20 cores.

4.5 Intel Max 1100 GPU

We also took the opportunity to test the new Intel Max 1100 GPUs available through Intel Developer Cloud Beta. Using the same module for PyTorch optimization, the only change in Listing 4.1 is at line 4: `device="xpu"`. We applied warmup and collected the inference time for 1000 runs and computed the average. Table 4.2 shows the results for the GPU together with the CPU (here only for 1 and 56 cores) for comparison purposes. One can note that this GPU presents results 18% faster than a full socket (56 cores) for the baseline PyTorch model. This could be considered good, however the improvement is much more pronounced when applying IPEX plus tracing. Then the inference time using the GPU is 2.63 times faster than a full socket.

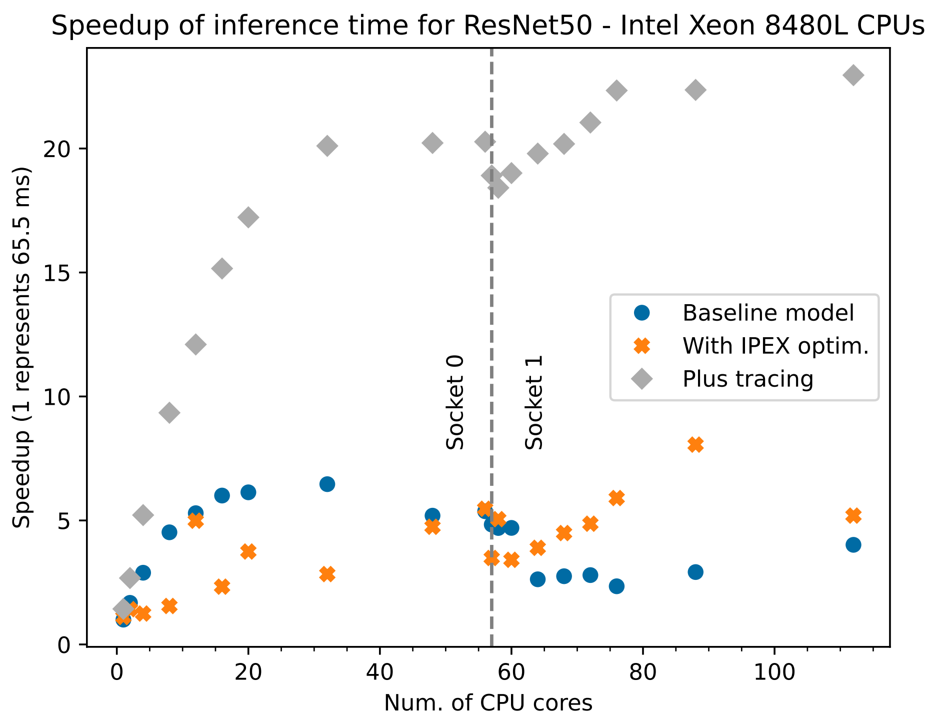

Figure 4.2: Speedup of inference time for ResNet50 model in Intel Xeon 8480+ CPUs.

Table 4.2: Average inference time for YOLOv5 license plate model and different optimizations using Intel CPU and GPU

	Inference time (ms)		
	CPU 8480+		GPU Max
Model	1 core	56 cores	1100
YOLO License Plate	126.2	18.4	15.5
+Intel optim.	103.7	11.1	4.0
+tracing	97.7	7.1	2.7

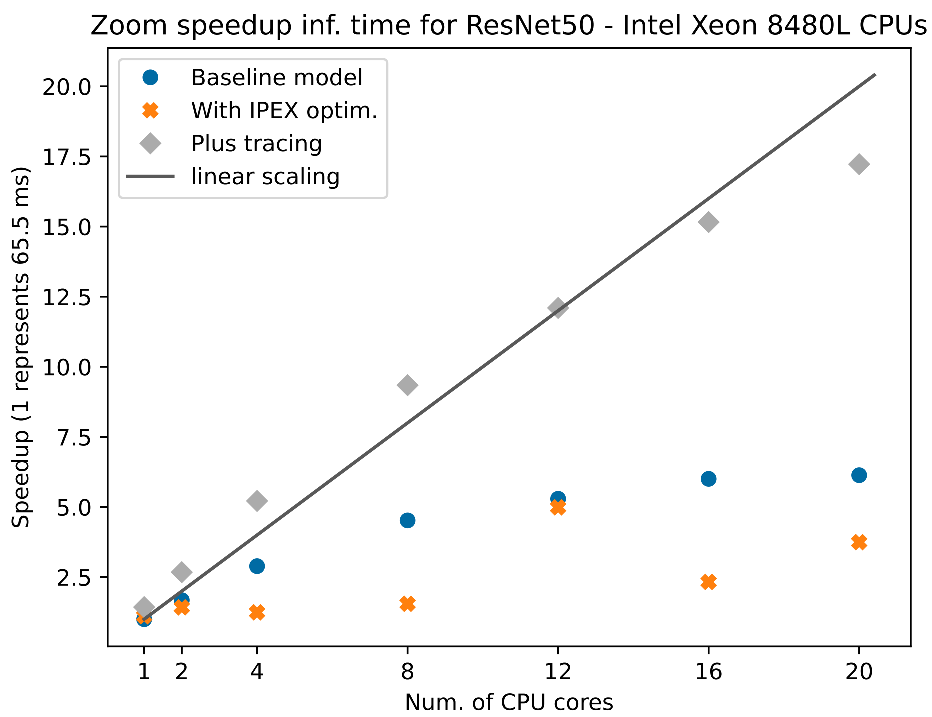

Figure 4.3: Zooming the speedup of inference time for ResNet50 model in Intel Xeon 8480+ CPUs.

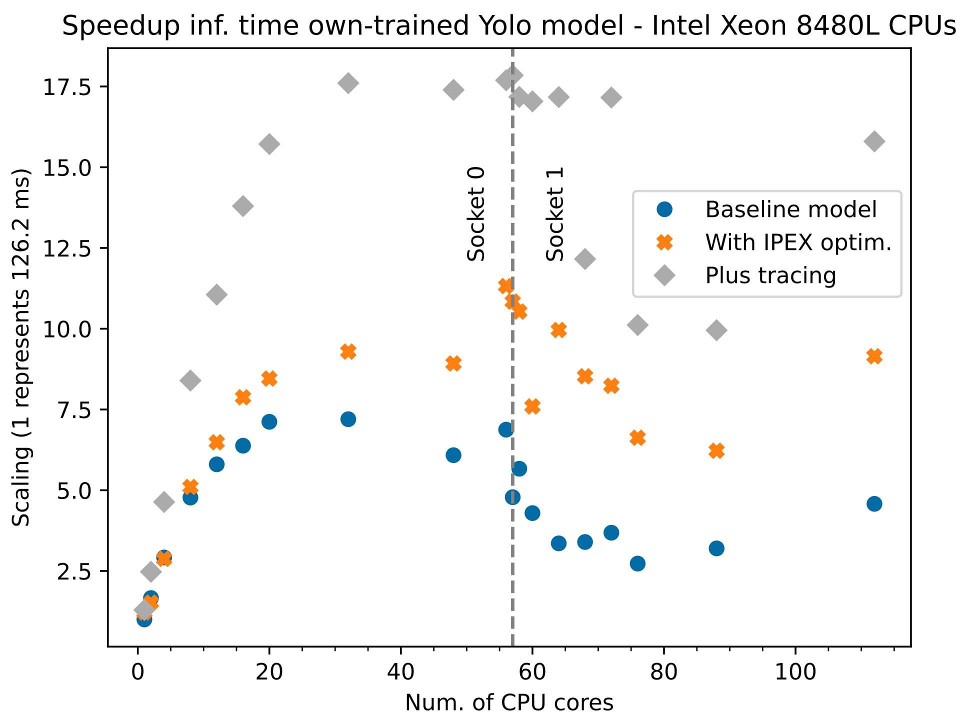

Figure 4.4: Speedup of inference time for YOLO license plate model in Intel Xeon 8480+ CPUs.

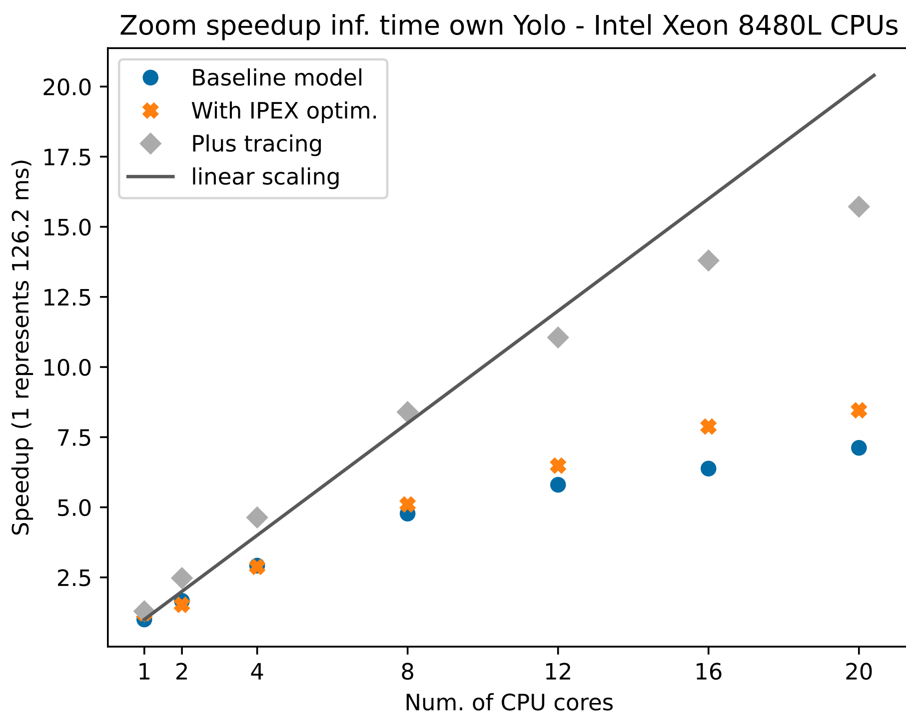

Figure 4.5: Zooming the speedup of inference time for YOLO license plate model in Intel Xeon 8480+ CPUs.

4.6 OpenVINO

OpenVINO is a model optimizer for CPUs and GPUs, developed by Intel [43]. As TensorRT, it rebuilds the computational graph.

We tested OpenVINO version 2023.0.1 for the Intel CPUs at DaVinci-1. The installation was accomplished simply with `pip` and the module `openvino-dev==2023.0.1`. Then, a change in the source code of YOLO `export.py`, line 197, was needed because this version of OpenVINO deprecated an argument named `--data_type` in favor of `--compress_to_fp16`. The output of `diff` is shown in Listing 4.2.

```
194       LOGGER.info(f'\n{prefix} starting export with openvino {
      ie.__version__}...')
195       f = str(file).replace('.pt', f'_openvino_model{os.sep}')
196
197 -     cmd = f"mo --input_model {file.with_suffix('.onnx')} --
      output_dir {f} --data_type {'FP16' if half else 'FP32'}"
198 +     cmd = f"mo --input_model {file.with_suffix('.onnx')} --
      output_dir {f} --compress_to_fp16 {half}"
```

Listing 4.2: Change in the source code of export.py in YOLOv5 branch 7.0 to use OpenVINO 2023.0.1.

An OpenVINO model is generated by means of `export.py`, using a PyTorch `.pt` model as input:

```
python export.py --weights model.pt --include openvino
```

We did some experiments fixing the number of threads with `numactl`. Results are shown in Table 4.3. The row marked with asterisk was obtained with `OMP_NUM_THREADS=48` running on 24 physical cores:

```
env OMP_NUM_THREADS=48 numactl --cpunodebind=0 --physcpubind
    =+0-23 python detect.py --weights='openvino_model/' --
    device='cpu' --source='img/data/*'
```

It seems that even with HyperThreading disabled, OpenVINO can take advantage of some parallelism if two threads are assigned to the same core. For the Intel 8260 CPU with batch size one, the result of 9.8 ms is the best so far in our tests.

4.7 Image size

We investigated the influence of the image resolution on the inference time. Although the default size for YOLOv5s is 640×640 pixels, one can use other image resolutions. We considered the following dimensions for images with equal width and height: 224, 480, 640, and 1280. Figures 4.6, 4.7, and 4.8

Table 4.3: Inference time for OpenVINO

Runtime + Threads	Inf. time (avg.) (ms)
OpenVINO CPU 1	95.9
OpenVINO CPU 24	14.2
OpenVINO CPU 48*	9.8
OpenVINO CPU 48	14.2

present the inference times for 1, 8 and 56 cores of the Intel Xeon 8480+ CPU, respectively.

First let's examine the results for one core (Fig. 4.6). An image of 1280×1280 pixels has 33 times more pixels than one with resolution 224×224 and this is also approximately the increase in inference time: 30 times for the baseline model and 31 times for the optimized and traced on. So, with only one core no "magic" optimization can save time, the scaling in time is linear. On the other hand, there is an advantage in using the IPEX+trace: approximately 40% faster than the baseline model.

For 8 cores (Fig. 4.7), the increase in inference time when moving from a resolution of 224 to 1280 is around 14 times, while the number of pixels increases 33 times. With more cores, the advantages of IPEX and tracing are more evident: inference times are 2, 2.1, 1.8 and 1.78 times faster than the baseline model for each resolution, respectively.

For a full socket, 56 cores (Fig. 4.8), changing the resolution from 224 to 1280 the inference time increases 7 times, approximately. Applying IPEX plus tracing, inference times are 3, 3.2, 2.86, 2.61 faster than the baseline model for each resolution, respectively.

4.8 Batch size

The inference can be executed image by image or in blocks. The number of images that are collected and then processed as a single multidimensional array is the batch size. A study about it is presented at [44], that we reproduce in Table 4.4, doing inference in a NVIDIA A100 GPU with 40 GB of memory.

Doing a similar study using the GPU at DaVinci-1, for the YOLO license plate model, when moving batch size from 1 to 8, PyTorch inference reduced from 7 ms to 1.96 ms. With TorchScript (that is, with tracing and optimization), from 5.51 to 1.2 ms and TensorRT, from 1.47 ms to 0.6 ms (results

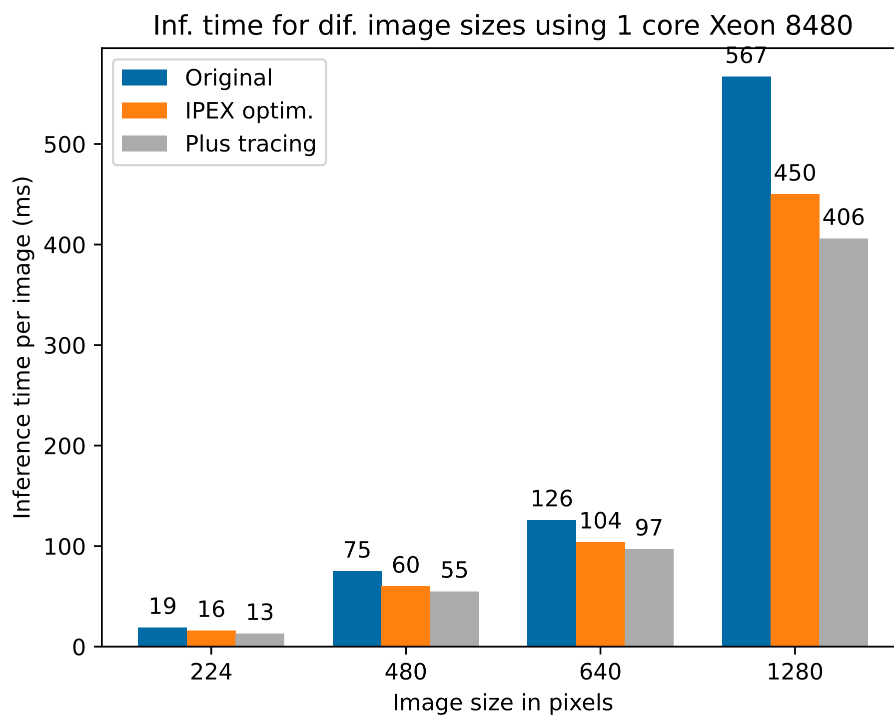

Figure 4.6: Evaluating the influence of image resolution and optimization on the inference time for one core of the Intel Xeon 8480+ CPU.

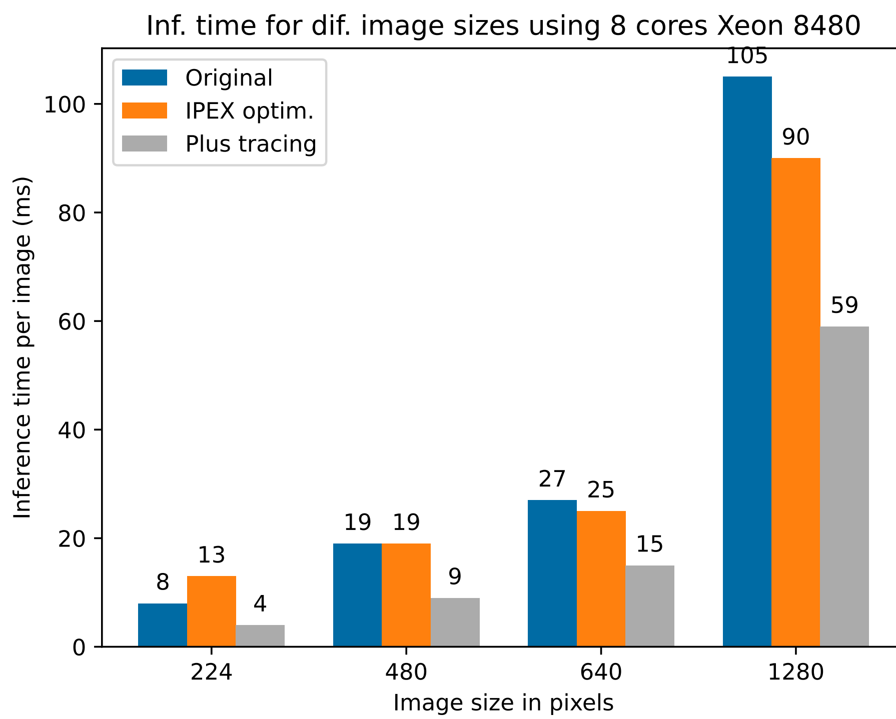

Figure 4.7: Evaluating the influence of image resolution and optimization on the inference time for eight cores of the Intel Xeon 8480+ CPU.

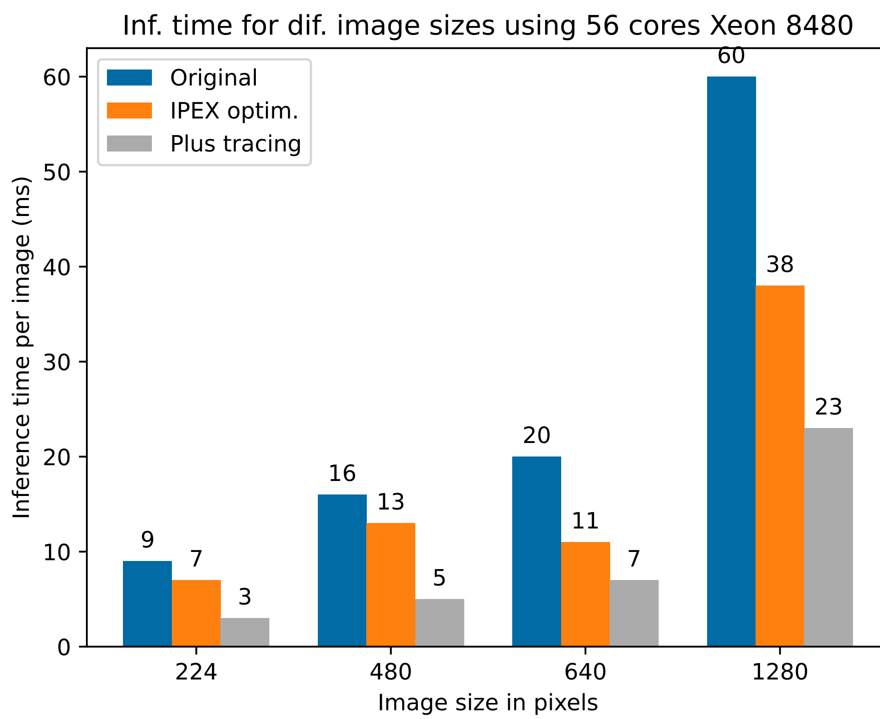

Figure 4.8: Evaluating the influence of image resolution and optimization on the inference time for a full socket of the Intel Xeon 8480+ CPU.

Table 4.4: Influence of batch size on inference time (from [44])

	Inference time YOLOv5 models (ms)				
Batch size	nano	small	medium	large	extra large
1	9.0	9.3	12.2	14.5	18.9
2	5.0	5.0	6.6	7.7	10.2
4	2.6	2.7	3.4	4.2	5.0
8	1.3	1.3	1.7	2.3	3.4
16	0.7	0.9	1.4	1.9	3.1
32	0.5	0.8	1.3	1.9	3.0
64	0.4	0.7	1.2	1.7	2.8
128	0.4	0.6	1.1	1.7	2.8

in Table 4.5. We can say it is advantageous to increase the batch size, taking into account the requirements of the final application. It is important to remember that some time is spent building a batch of images before the inference. If one considers 30 frames per second (FPS), for a single camera system the collection of 8 frames would imply in a delay of 267 ms. However, for systems that collect images from multiples cameras, a composition (like a multicamera "mosaic") could be a solution to assemble a batch with minimal delay.

Table 4.5: Influence of batch size on inference time for YOLO license plate model

	Inference time (ms)		
Batch size	PyTorch	TorchScript	TensorRT
1	7.0	5.51	1.47
2			0.9
4			0.7
8	1.96	1.2	0.6
16			0.5
32			0.5

We conducted a similar study using the Intel Dev Cloud Beta. The results of inference time per image are presented in Fig. 4.9 for 8 cores of the Intel 8480 CPU and batch sizes 1, 2, 4, 8, and 16. In the CPU we do not note any speedup, even when applying the IPEX plus tracing optimization. The time even increases for batch size 16, so there is no advantage here.

For the Intel Max 1100 GPU (Fig. 4.10), there is some improvement,

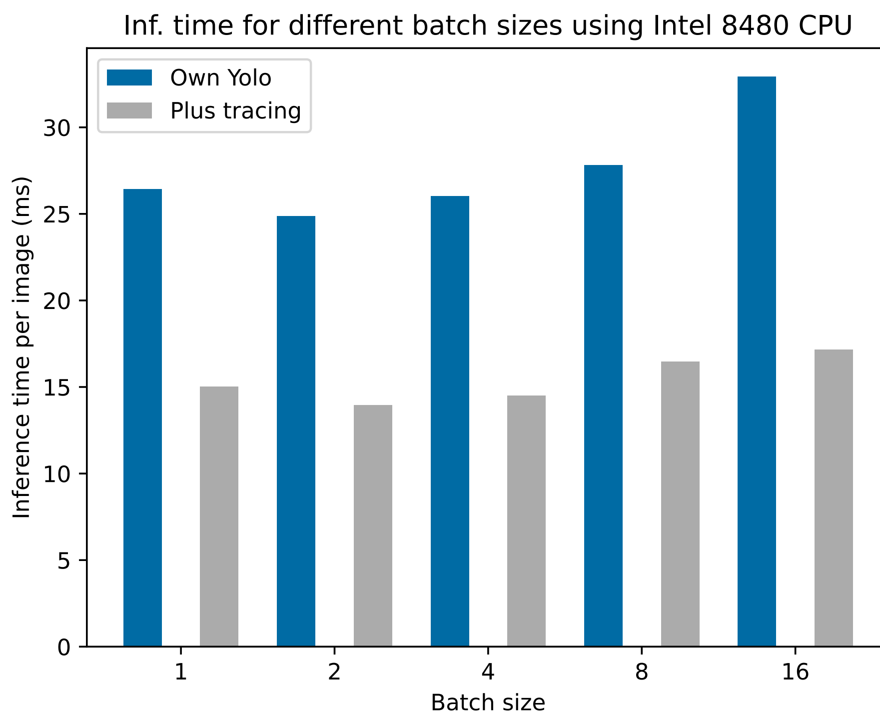

Figure 4.9: Inference time per image for different batch sizes using Intel Xeon 8480+ CPU.

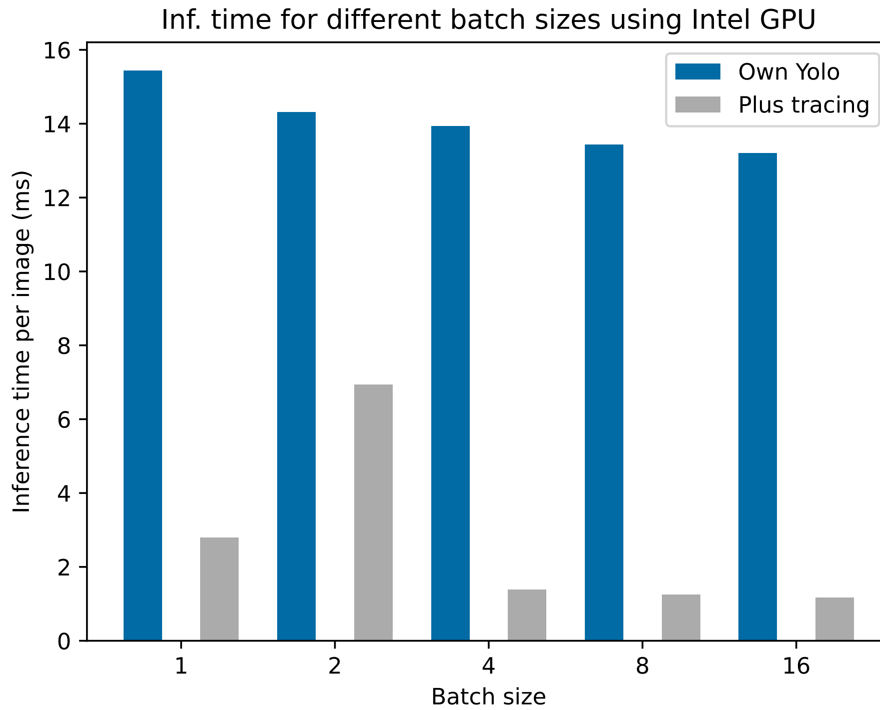

Figure 4.10: Inference time per image for different batch sizes using Intel Max 1100 GPU.

however not scaling as for the NVIDIA A100. Maybe these mixed results for the Intel Dev Cloud Beta can be attributed to more users accessing the system. We noticed that even when requesting a full node for the experiments sometimes the access to resources were shared. The occupancy of this cluster increased significantly during the months it was freely available.

Chapter 5

Energy consumption

In the last few years, for reasons of environmental awareness and cost reduction, benchmarking the energy consumption of machine learning applications has become fundamental [45].

The approaches to measure energy consumption can be divided into three classes [46]:

- physical measurements with external power meters
- on-chip power sensors
- predictive models

External measurement is the most reliable but it was not available for this work. Depending on the system, it has no fine grain resolution, i.e., one can not obtain individual measurements from the CPU, memory or GPU. Predictive models suffer with insufficient training data and usually do not present adequate precision [46]. The only option available in this work was to read on-chip power sensors, when system policy allowed standard users to access them.

On Intel machines, CPU energy readings are available in the Running Average Power Limit (RAPL) registers. They can be read directly, for instance, with `cat "/sys/class/powercap/intel-rapl:0/energy_uj"` or through `perf`. However, as mentioned, it depends on the access policy. These measurements were considered very unreliable in the past, although newer CPU architectures seem to provide trusty data. Even without full agreement about the reliability, it is the most common way to measure energy in the literature [45]–[48].

For the GPUs, each vendor makes available one tool. NVIDIA software framework includes a tool called `nvidia-smi`. A Python binding to the

NVIDIA Management Library is available through the `pynvml` package. Intel has a similar tool named `xpu-smi`, although a Python binding is not of our knowledge.

Still in Python, a package called `codecarbon` [49] aggregates data from CPUs and NVIDIA GPUs in a very convenient way, enabling the measurement for a whole script or even a single line of code. For CPUs, we verified the values agree with those obtained with `perf`. When available (in our case, at DaVinci-1), we report measurements from this tool. For the Intel GPU, we call the `xpu-smi` tool inside Python using `psutil.Popen`.

We were not able to obtain energy measurements for the Intel CPUs at the Intel Dev Cloud Beta because the system policy restricted access to the RAPL registers. For the AMD CPUs at DaVinci-1, values were only an estimate based on the CPU model (assumes constant consumption mode), and since they are not reliable we do not present here.

5.1 Training

Using the tool `codecarbon`, we obtained the energy consumption for part of the training of the YOLOv5 license plate model at DaVinci-1. Energy and training times are shown for 1, 2, and 4 GPUs and for 48 cores of the Intel Xeon 8260 CPU in Table 5.1. It is clear that GPU training is much more advantageous regarding time and energy efficiency, consuming around 8 times less energy than using the CPUs.

Table 5.1: Energy consumption and training time using different devices

Hardware	Energy GPU (Wh)	Energy CPU (Wh)	Time 3 epochs	Total time (s)
1 GPU	5.707	1.968	50	78
2 GPUs	5.986	1.782	29	71
4 GPUs	5.258	1.996	22	79
48 CPU cores	0	60.924	814	848

Note that the energy measurement is for the whole process, it means it is not only the epoch training, but loading the base model, checking and augmenting images, and other initial operations. The tool we had available for measurement also computes the energy for the idle GPUs. For this training that is relatively small for a GPU (17 s for one epoch), it means that using more GPUs seems not worth because the total time for training is almost the same as using only one.

Since this training was relatively short, we did another experiment using a larger model and larger training set. For the YOLO5l model, COCO dataset (118 thousand images), 4 GPUs, batch size 64, one epoch took 6 minutes to complete, similar to the value (5min57s) reported by [50]. The consumed energy by the GPUs was 115 Wh and 19 Wh for the CPU.

Returning to the model YOLOv5s but with the larger COCO dataset and 4 GPUs, 1 epoch completed in 3min50s and the consumed energy was 62 Wh for GPUs and 15 Wh for CPUs. If we train the same model using one full node with the Intel Xeon Platinum 8260 CPUs, 10% of the epoch took 95 Wh of energy and 22 minutes. We interrupted the training at that point. Extrapolating that the total energy for one epoch would be 950 Wh and it would take 220 minutes, we can say the energy consumption is 12 times higher and the time spent 57 times longer using this CPUs compared to the GPUs.

5.2 Inference

With the same tool `codecarbon` we conducted experiments regarding the energy consumption of inference. Since the resolution of the tool is not precise for measuring the inference of only one image, we use the full test dataset with 2065 images. Table 5.2 reports the energy measurements at DaVinci-1 using the NVIDIA A100 GPU and Intel 8260 CPUs for the 2065 images with batch size equal to one. As a reference, it is also included the average inference time per image (discussed in Chapter 4) in the last column, considering batch size equal to one.

The baseline PyTorch inference, running in only one core, consumes more than 7 Wh for the whole test set. Results show the best energy performance is obtained using eight cores or occupying the full socket (24 cores), which has an advantage in inference time. The rows marked with an asterisk were obtained with `OMP_NUM_THREADS=48` running on 24 physical cores. This is not advantageous for PyTorch inference, but presents the best result for OpenVINO. Regarding it, one can note time and energy consumption decreasing as the number of cores increases from one to 24. For OpenVINO, saturating the CPU is desirable, as the best energy and time marks achieve 2.219 Wh and 9.8 ms, respectively. For the GPU, NVIDIA TensorRT optimizes the model and obtains 1.5 ms of inference time and a combined energy consumption of 2.778 Wh. This last result is very interesting because if time is not the main issue in the application, it is beneficial to use OpenVINO in the CPU for inference, since the energy consumption has a reduction of 20% when compared with the GPU.

Table 5.2: Energy consumption and inference time for different engines

Runtime/ Threads	Energy GPU (Wh)	Energy CPU (Wh)	Inf. time (avg.) (ms)
PyTorch CPU 1	0	7.339	111.5
PyTorch CPU 8	0	3.413	28.2
PyTorch CPU 24	0	3.641	25.0
PyTorch CPU 48*	0	6.465	69.7
PyTorch CPU 48	0	5.921	33.1
OpenVINO CPU 1	0	6.691	95.9
OpenVINO CPU 24	0	2.806	14.2
OpenVINO CPU 48*	0	2.219	9.8
OpenVINO CPU 48	0	4.136	14.2
PyTorch GPU 1	2.597	1.091	7.4
TensorRT 1	1.961	0.817	1.5

Chapter 6

Conclusions

This work presented a performance evaluation of a object detection model based on YOLOv5. We conducted timing and energy measurements for model training and inference. We presented results for Intel and AMD CPUs, and Intel and NVIDIA GPUs. Different software stacks were evaluated, considering PyTorch as the baseline.

We defined vehicle license plates as the targets to detect in images. We selected and filtered an open image dataset, obtaining images and labels for training, validation and test. Then, we trained a model from YOLOv5 weights, achieving a average precision of 96% in the validation set and 92% for the test dataset.

If we consider the speed, among the systems we tested, NVIDIA A100 GPU is the best one, achieving an inference time per image of 1.5 ms for batch size 1 when optimized by TensorRT. It is also a good option regarding energy. With Intel Max 1100 GPU, the inference time we obtained was 2.7 ms after optimization and tracing. A PyTorch model without any additional optimization in the same NVIDIA GPU needs 7.4 ms.

For the Intel 8480+ CPU, using Intel Optimization for PyTorch plus tracing the inference time is of 7.1 ms. Good results can be also obtained with Intel OpenVINO, of 9.8 ms for an Intel 8260 CPU. Although not the fastest solution, OpenVINO on the CPU presented the best energy efficiency.

When using larger batch sizes, inference time can be significantly reduced. For TensorRT and batch size 8, it drops to 0.6 ms.

Training the model using custom data with pretrained weights as a starting point is fast in a GPU. It took 17 s per epoch and 25 epochs are enough to achieve the average precision of 92%. If one has access to GPUs, model training using CPUs is not worth. From our results for a larger training dataset (COCO) and the YOLOv5s model, the energy consumption is 12 times higher and the time spent 57 times longer.

Some suggestions for future work include the evaluation of other computer architectures, like RISC-V and ARM, for data centers and embedded applications, as well as other inference engines (Google's BigQuery ML or Apple's Core ML, for instance). Another interesting research regards optimizing the model architecture, simplifying it, experimenting different quantization levels or applying neural architecture search.

References

[1] L. Letaw, *Handbook of Software Engineering Methods*, https://open.oregonstate.education/setextbook/, [Accessed 10-10-2023], 2023.

[2] *Jose Mesa*, https://www.flickr.com/people/liferfe/, [Accessed 10-10-2023].

[3] J. Quental, *Lear's Macaw Anodorhynchus leari (cropped).jpg*, https://commons.wikimedia.org/wiki/File:Lear's_Macaw_Anodorhynchus_leari_(cropped).jpg, [Accessed 16-10-2023].

[4] N. Silberman, D. Sontag, and R. Fergus, "Instance segmentation of indoor scenes using a coverage loss," in *Computer Vision – ECCV 2014*, D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, Eds., Cham: Springer International Publishing, 2014, pp. 616–631, ISBN: 978-3-319-10590-1. DOI: 10.1007/978-3-319-10590-1_40.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds., vol. 25, Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: https://dl.acm.org/doi/10.5555/2999134.2999257.

[6] R. Girshick, J. Donahue, T. Darrell, and J. Malik, *Rich feature hierarchies for accurate object detection and semantic segmentation*, 2014. arXiv: 1311.2524 [cs.CV].

[7] Z. Zou, K. Chen, Z. Shi, Y. Guo, and J. Ye, *Object detection in 20 years: A survey*, 2023. arXiv: 1905.05055 [cs.CV].

[8] V. Lakshmanan, M. Görner, and R. Gillard, *Practical Machine Learning for Computer Vision*. O'Reilly Media, 2021, ISBN: 9781098102333.

[9] R. Padilla, W. L. Passos, T. L. B. Dias, S. L. Netto, and E. A. B. da Silva, "A comparative analysis of object detection metrics with a companion open-source toolkit," *Electronics*, vol. 10, no. 3, 2021, ISSN: 2079-9292. DOI: 10.3390/electronics10030279.

[10] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, IEEE Computer Society, 2016, pp. 779–788. DOI: 10.1109/CVPR.2016.91.

[11] J. Terven and D. Cordova-Esparza, *A comprehensive review of YOLO: From YOLOv1 and beyond*, 2023. arXiv: 2304.00501 [cs.CV].

[12] *GitHub - ultralytics/yolov5 at v7.0*, https://github.com/ultralytics/yolov5/tree/v7.0, [Accessed 19-09-2023].

[13] *Ultralytics/yolov5 at v7.0 pretrained checkpoints*, https://github.com/ultralytics/yolov5/tree/v7.0#pretrained-checkpoints, [Accessed 16-10-2023].

[14] D. Konyrev, *The history of YOLO: The origin of the YOLOv1 algorithm*, https://www.superannotate.com/blog/yolov1-algorithm, [Accessed 16-10-2023].

[15] S.-H. Tsang, *Review: YOLOv1*, https://towardsdatascience.com/yolov1-you-only-look-once-object-detection-e1f3ffec8a89, [Accessed 16-10-2023], 2018.

[16] A. Mondin, *YOLOV5(m): Implementation From Scratch With PyTorch*, https://pub.towardsai.net/yolov5-m-implementation-from-scratch-with-pytorch-c8f84a66c98b, [Accessed 16-10-2023], 2023.

[17] *YOLOv5 (6.0/6.1) brief summary - Issue #6998*, https://github.com/ultralytics/yolov5/issues/6998, [Accessed 16-10-2023].

[18] C.-Y. Wang, H.-Y. M. Liao, I.-H. Yeh, Y.-H. Wu, P.-Y. Chen, and J.-W. Hsieh, *CSPNet: A new backbone that can enhance learning capability of CNN*, 2019. arXiv: 1911.11929 [cs.CV].

[19] *Car License Plate Detection*, https://www.kaggle.com/datasets/andrewmvd/car-plate-detection, [Accessed 11-09-2023].

[20] *License Plate Characters - Detection OCR*, https://www.kaggle.com/datasets/francescopettini/license-plate-characters-detection-ocr, [Accessed 11-09-2023].

[21] R. Laroca, E. Severo, L. A. Zanlorensi, *et al.*, "A robust real-time automatic license plate recognition based on the YOLO detector," in *International Joint Conference on Neural Networks (IJCNN)*, Jul. 2018, pp. 1–10. DOI: 10.1109/IJCNN.2018.8489629.

[22] R. Laroca, L. A. Zanlorensi, G. R. Gonçalves, E. Todt, W. R. Schwartz, and D. Menotti, "An efficient and layout-independent automatic license plate recognition system based on the YOLO detector," *IET Intelligent Transport Systems*, vol. 15, no. 4, pp. 483–503, 2021, ISSN: 1751-956X. DOI: 10.1049/itr2.12030.

[23] R. Laroca, V. Estevam, A. S. Britto, R. Minetto, and D. Menotti, "Do we train on test data? The impact of near-duplicates on license plate recognition," in *2023 International Joint Conference on Neural Networks (IJCNN)*, IEEE, Jun. 2023. DOI: 10.1109/ijcnn54540.2023.10191584.

[24] *Truth 4 all — N05*, https://www.flickr.com/people/45639276@N05/, [Accessed 14-09-2023].

[25] A. Kuznetsova, H. Rom, N. Alldrin, *et al.*, "The Open Images Dataset V4: Unified image classification, object detection, and visual relationship detection at scale," *International Journal of Computer Vision*, vol. 128, pp. 1956–1981, 2020. DOI: 10.1007/s11263-020-01316-z.

[26] B. E. Moore and J. J. Corso, *GitHub - voxel51/fiftyone: The open-source tool for building high-quality datasets and computer vision models*, https://github.com/voxel51/fiftyone, [Accessed 12-09-2023], 2020.

[27] *Yolov5/train.py at v7.0*, https://github.com/ultralytics/yolov5/blob/v7.0/train.py, [Accessed 14-09-2023].

[28] *DaVinci-1 - TOP500*, https://www.top500.org/system/179877/, [Accessed 18-10-2023].

[29] *Intel® Xeon® Platinum 8260 Processor (35.75M Cache, 2.40 GHz) - Product Specifications*, https://www.intel.com/content/www/us/en/products/sku/192474/intel-xeon-platinum-8260-processor-35-75m-cache-2-40-ghz/specifications.html, [Accessed 18-10-2023].

[30] *AMD EPYC 7402 - Product Specifications*, https://www.amd.com/en/product/8816, [Accessed 18-10-2023].

[31] *NVIDIA A100 Tensor Core GPU*, https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet.pdf, [Accessed 18-10-2023].

[32] *Intel Developer Cloud Beta*, https://scheduler.cloud.intel.com, [Accessed 28-09-2023].

[33] *Intel® Xeon® Platinum 8480+ Processor (105M Cache, 2.00 GHz) - Product Specifications*, https://www.intel.com/content/www/us/en/products/sku/231746/intel-xeon-platinum-8480-processor-105m-cache-2-00-ghz/specifications.html, [Accessed 18-10-2023].

[34] *Intel® Data Center GPU Max 1100 - Product Specifications*, https://www.intel.com/content/www/us/en/products/sku/232876/intel-data-center-gpu-max-1100/specifications.html, [Accessed 18-10-2023].

[35] A. Paszke, S. Gross, F. Massa, *et al.*, "PyTorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[36] A. Geifman, *The Correct Way to Measure Inference Time of Deep Neural Networks*, https://deci.ai/blog/measure-inference-time-deep-neural-networks/, [Accessed 27-09-2023].

[37] L. Atkins and D. MacLeod, *How to Accurately Time CUDA Kernels in Pytorch*, https://www.speechmatics.com/company/articles-and-news/timing-operations-in-pytorch, [Accessed 27-09-2023], 2023.

[38] *NVIDIA TensorRT*, https://developer.nvidia.com/tensorrt, [Accessed 02-10-2023].

[39] *Intel Extension for PyTorch Documentation*, https://intel.github.io/intel-extension-for-pytorch, [Accessed 27-09-2023].

[40] *Match Every Application to Its Optimal Architecture with XPU*, https://www.intel.com/content/www/us/en/architecture-and-technology/xpu.html, [Accessed 18-10-2023].

[41] *Optimizing Production PyTorch Models' Performance with Graph Transformations*, https://pytorch.org/blog/optimizing-production-pytorch-performance-with-graph-transformations/, [Accessed 27-09-2023].

[42] W. Niu, J. Guan, Y. Wang, G. Agrawal, and B. Ren, "DNNFusion: Accelerating deep neural networks execution with advanced operator fusion," *CoRR*, vol. abs/2108.13342, 2021. arXiv: 2108.13342.

[43] *OpenVINO 2023.1 documentation*, https://docs.openvino.ai/2023.1/home.html, [Accessed 02-10-2023].

[44] G. Jocher, *YOLOv5 Study: Batch-Size vs Speed*, https://community.ultralytics.com/t/yolov5-study-batch-size-vs-speed/31, [Accessed 16-10-2023], 2022.

[45] Y. Sun, Z. Ou, J. Chen, *et al.*, "Evaluating performance, power and energy of deep neural networks on CPUs and GPUs," in *Theoretical Computer Science*, Z. Cai, J. Li, and J. Zhang, Eds., Singapore: Springer Singapore, 2021, pp. 196–221, ISBN: 978-981-16-7443-3. DOI: 10.1007/978-981-16-7443-3_12.

[46] M. Fahad, A. Shahid, R. R. Manumachu, and A. Lastovetsky, "A comparative study of methods for measurement of energy of computing," *Energies*, vol. 12, no. 11, 2019, ISSN: 1996-1073. DOI: 10.3390/en12112204.

[47] S. Desrochers, C. Paradis, and V. M. Weaver, "A validation of DRAM RAPL power measurements," in *Proceedings of the Second International Symposium on Memory Systems*, ser. MEMSYS '16, Alexandria, VA, USA: Association for Computing Machinery, 2016, pp. 455–470, ISBN: 9781450343053. DOI: 10.1145/2989081.2989088.

[48] A. Haidar, H. Jagode, P. Vaccaro, A. YarKhan, S. Tomov, and J. Dongarra, "Investigating power capping toward energy-efficient scientific applications," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 6, e4485, 2019. DOI: 10.1002/cpe.4485.

[49] *GitHub - mlco2/codecarbon: Track emissions from Compute and recommend ways to reduce their impact on the environment*, https://github.com/mlco2/codecarbon, [Accessed 19-10-2023].

[50] Ultralytics, *Multi-GPU Training*, https://docs.ultralytics.com/yolov5/tutorials/multi_gpu_training/#results, [Accessed 19-10-2023].