

**POLITECNICO DI TORINO**

---

Master Degree Course in Computer Engineering

Master Thesis

**Real-time object recognition in  
industrial automation processes**



**Politecnico  
di Torino**

**Supervisor**  
Prof. Maurizio Morisio

**Candidate**  
Michele Montatore

**Company tutor**  
**Orbyta Tech**  
Eng. Carla Federica Melia

---

ACADEMIC YEAR 2021-2022

To my parents and my brother

# Acknowledgements

I would like to give my sincerest thanks to Prof. Maurizio Morisio who gave me the opportunity of measure myself with this work, which has piqued my curiosity and increased my interest in the Artificial Intelligence area. I would also like to thank Carla Federica Melia and Dario Sammaruga for their constant support and guidance throughout the entire project duration. Their knowledge, together with their precious advice, have been fundamental to complete as far as possible the prefixed tasks. There is still a lot to be done for further improvements, but without them everything would have been more difficult.

Moreover, I am grateful of having made my academic journey alongside brilliant colleagues. I strongly think that all the moments of comparison with them will come in handy for my human and professional growth.

My thoughts also go to my friends, which have always been an essential part of my walk of life, a light at the end of the tunnel in times of difficulty.

And last but not least, I would like to express my undying gratitude to my parents, for instilling in me the awareness about the benefits of study and daily effort, mixed with a good dose of passion, and my brother, for whom I would like to be a guidance one day. I dedicate this work to them.

# Summary

Computer Vision is a branch of Artificial Intelligence that aims to enable machines to simulate the human visual system by extracting features of the physical world from images and videos usually taken by a camera, including tasks such as image classification, image segmentation, object detection, and many others. To perform them, artificial vision today's applications exploit the most diverse algorithms and tools involving geometric transformations, filtering operations, and also modern Deep Learning technologies such as Convolutional Neural Networks.

The present work explores the possibility of using some of them as a proof of concept for an automatic a posteriori check on the assembly of automotive seat frame parts, specifically colored motors, in an industrial environment where a bench lets the frames translate back and forth while a camera captures the scene from above. For this purpose, a PyTorch-based CNN model called YOLOv5 has been adopted for the real-time recognition of motors, combined with a color detection algorithm for the association of one color, among those of a predefined set, with them. The two entities work simultaneously: given a video stream, at each frame, YOLO locates the motors through bounding boxes, and the color detection algorithm is subsequently run to find the ones inside them. More specifically, three different approaches for detecting colors have been investigated, but only one of them has been chosen taking into account accuracy and especially real-time application suitability in terms of speed per frame.

Due to the difficulty in collecting a vast number of real variegated photos for the network training, a dataset of synthetic images with labels has been created starting from a 3D model of the seat frame. In particular, the joint use of the Unity game engine and C# scripts has enabled the generation of simulated videos from which screenshots at different conditions have been automatically taken and annotated. In this connection, an ad-hoc algorithm has been designed for the automatic computation of the bounding box coordinates for each motor.

The knowledge acquired on such virtual data has then been the basis for a second training phase involving the few available real images according to the widespread approach of Transfer Learning. This helped to better generalize on real samples. Various training trials have been done in the Google Colab online platform to get a final tuned model capable of providing decent results in terms of precision, recall,

and mAP regarding the object detection task independently; in addition, a statistical analysis procedure, to be performed on real video footages at inference, has also been implemented to further measure the recognition goodness in terms of average frequency and confidence, and concurrently evaluate color confidence. All of these values are not quantifiable at training time.

The entire training architecture complies with the ETL paradigm: datasets are extracted from Dropbox for Colab import, annotation files are filtered to discard eventual ones containing illegal coordinates together with the relative images, and relevant values such as losses and metrics, together with the trained model and metadata, are uploaded every epoch to the non-relational cloud database MongoDB Atlas, making them accessible for further reporting activities and keeping track of all the trials done until then.. Then, a user-friendly Python-based dashboard has been designed through the Streamlit app framework for the real-time visualization of detection results and other relevant information at inference so that a human operator can rapidly see how the system is working.

Despite the encouraging results in object and color recognition, some limitations lurk in this project: firstly, the sensitivity of the color detection algorithm to the environmental conditions, but perhaps more importantly, the challenges in guaranteeing a high speed of execution in real-time video processing, mostly when YOLO is integrated with the dashboard.

The work is organized as follows: Chapter 1 gives a theoretical comprehensive overview of the world of Computer Vision, focusing on color models, image processing and Deep Learning techniques, and critical issues which affect them; Chapter 2 presents the purpose of the current study together with a discussion on the related state of the art publications; Chapter 3 details the Unity-based synthetic data generation process; Chapter 4 precisely reports all the technologies employed for object and color detection together with the methodology adopted; Chapter 5 describes the ETL architecture and the data visualization part via dashboard; finally, Chapter 6 is about the best-performing experiments and the relative results based on the metrics chosen for evaluation. A conclusion section is then included to collect some final considerations and suggest eventual future developments of what has been done.

# Contents

<b>List of Figures</b>	VIII
<b>List of Tables</b>	X
<b>List of Abbreviations</b>	XI
<b>1 Introduction to Computer Vision</b>	<b>1</b>
1.1 A brief survey . . . . .	1
1.2 Color models . . . . .	2
1.2.1 RGB and CMY . . . . .	2
1.2.2 HSL and HSV . . . . .	3
1.2.3 Conversions . . . . .	4
1.3 Image processing . . . . .	5
1.3.1 Binarization . . . . .	6
1.3.2 Quantization . . . . .	6
1.3.3 Geometric transformations . . . . .	7
1.3.4 Filtering and morphological transformations . . . . .	9
1.4 Machine Learning and ANNs . . . . .	10
1.4.1 Supervised vs. Unsupervised Learning . . . . .	11
1.4.2 Performance evaluation . . . . .	16
1.4.3 Basic principles of ANNs . . . . .	17
1.4.4 Convolutional Neural Networks . . . . .	26
1.4.5 Transfer Learning . . . . .	34
1.5 Critical issues . . . . .	35
<b>2 Project overview</b>	<b>37</b>
2.1 Companies presentation . . . . .	37
2.2 Tasks and goals . . . . .	37
2.3 State of the art . . . . .	38
<b>3 Data collection</b>	<b>42</b>
3.1 Unity overview . . . . .	42

3.2	Virtual environment setup . . . . .	44
3.3	Images and annotations . . . . .	46
<b>4</b>	<b>Materials and methods for object recognition</b>	<b>48</b>
4.1	Python and libraries . . . . .	48
4.2	YOLOv5 overview . . . . .	49
4.2.1	Architecture . . . . .	49
4.2.2	Learning . . . . .	51
4.2.3	Code structure . . . . .	52
4.3	Color-based recognition . . . . .	55
4.4	YOLO and colors . . . . .	59
4.4.1	YOLO-based recognition . . . . .	60
4.4.2	In-box color detection . . . . .	61
4.5	Statistical analysis on video frames . . . . .	64
<b>5</b>	<b>ETL architecture</b>	<b>69</b>
5.1	Data path . . . . .	70
5.2	Data visualization . . . . .	71
<b>6</b>	<b>Experiments and results</b>	<b>73</b>
6.1	Color ranges tuning . . . . .	73
6.2	YOLO performance . . . . .	74
6.3	Case study . . . . .	75
	<b>Conclusions</b>	<b>79</b>
	<b>Bibliography</b>	<b>81</b>

# List of Figures

1.1	RGB and CMY color models. . . . .	3
1.2	HSL and HSV color models. . . . .	4
1.3	Example of binarization with global threshold. . . . .	6
1.4	Example of quantization to vary the number of colors. . . . .	7
1.5	Two main morphological transformations in image processing. . . . .	10
1.6	Comparison between bad and good model fitting. . . . .	12
1.7	Batch learning. . . . .	14
1.8	Example of 2D data clustering. . . . .	15
1.9	Example of simple ANN architecture. . . . .	18
1.10	Early stopping. . . . .	24
1.11	Data augmentation pipeline. . . . .	24
1.12	Example of dropout regularization. . . . .	25
1.13	Example of simple CNN architecture. . . . .	26
1.14	Max pooling vs. avg pooling. . . . .	26
1.15	Example of image classification using CNN. . . . .	28
1.16	Two architectural innovations for CNN-based image classifiers. . . . .	28
1.17	Abstraction of 2D bounding box concept in images. . . . .	29
1.18	Comparison between one-stage and two-stage object detection. . . . .	30
1.19	IoU for object detection. . . . .	31
1.20	NMS application. . . . .	32
1.21	Example of image segmentation using CNN. . . . .	33
1.22	Example of image upsampling through NN interpolation. . . . .	34
1.23	Main critical issues in CV. . . . .	36
3.1	Unity simulated environment for synthetic image generation. . . . .	45
3.2	Example of real and synthetic samples from the relative datasets. . . . .	46
4.1	YOLOv5 baseline architecture. . . . .	50
4.2	YOLOv5 models of different size. . . . .	50
4.3	OpenCV color wheel. . . . .	56
4.4	Diagram of OpenCV-based color detection. . . . .	57
4.5	In-box color detection by K-Means clustering. . . . .	62

4.6	In-box color detection by Median Cut quantization. . . . .	63
4.7	In-box color detection by pixel masking. . . . .	64
4.8	Diagram of real-time statistical analysis on video frames. . . . .	66
5.1	ETL architecture of the proposed work. . . . .	70
5.2	Example of Streamlit-based dashboard with YOLO integration. . .	72
6.1	YOLOv5s training results. . . . .	76
6.2	Snapshot of real-time object and color detection. . . . .	78

# List of Tables

1.1	Common geometric transformations in image processing. . . . .	8
1.2	Label encoding vs. one-hot encoding. . . . .	12
1.3	Some common non-linear activation functions. . . . .	19
3.1	Some C# structs from the System namespace. . . . .	43
3.2	Summary of the two datasets used in this work. . . . .	47
4.1	YOLOv5 model scaling. . . . .	51
4.2	Hardware differences between laptop and Colab free tier. . . . .	60
6.1	Tuned HSV ranges for color detection. . . . .	73
6.2	YOLOv5s hyperparameter configuration for training. . . . .	75
6.3	YOLOv5s validation results with synthetic and real dataset. . . . .	77
6.4	Statistical analysis results on a real video. . . . .	78

# List of Abbreviations

<b>AI</b>	Artificial Intelligence	<b>SVM</b>	Support Vector Machine
<b>ANN</b>	Artificial Neural Network	<b>TN</b>	True Negatives
<b>BCE</b>	Binary Cross Entropy	<b>TP</b>	True Positives
<b>BGR</b>	Blue, Green, Red	<b>TTA</b>	Test Time Augmentation
<b>BN</b>	Batch Normalization	<b>VGG</b>	Visual Geometry Group
<b>CE</b>	Cross Entropy	<b>VOC</b>	Visual Object Classes
<b>CIoU</b>	Complete IoU	<b>YOLO</b>	You Only Look Once
<b>CMY</b>	Cyan, Magenta, Yellow		
<b>CMYK</b>	Cyan, Magenta, Yellow, black		
<b>CNN</b>	Convolutional Neural Network		
<b>COCO</b>	Common Objects in COntext		
<b>CSP</b>	Cross Stage Partial		
<b>CV</b>	Computer Vision		
<b>DL</b>	Deep Learning		
<b>ETL</b>	Extract, Transform, Load		
<b>FN</b>	False Negatives		
<b>FP</b>	False Positives		
<b>FPN</b>	Feature Pyramid Network		
<b>FPS</b>	Frames Per Second		
<b>GD</b>	Gradient Descent		
<b>HSL</b>	Hue, Saturation, Lightness		
<b>HSV</b>	Hue, Saturation, Value		
<b>IoU</b>	Intersection over Union		
<b>MAE</b>	Mean Absolute Error		
<b>mAP</b>	mean Average Precision		
<b>ML</b>	Machine Learning		
<b>MSE</b>	Mean Squared Error		
<b>NMS</b>	Non-Maximum Suppression		
<b>NN</b>	Nearest Neighbor		
<b>PANet</b>	Path Aggregation Network		
<b>R-CNN</b>	Region-based CNN		
<b>ReLU</b>	Rectified Linear Unit		
<b>ResNet</b>	Residual Network		
<b>RGB</b>	Red, Green, Blue		
<b>RGBA</b>	Red, Green, Blue, Alpha		
<b>SGD</b>	Stochastic Gradient Descent		
<b>SiLU</b>	Sigmoid Linear Unit		
<b>SPP</b>	Spatial Pyramid Pooling		

# Chapter 1

# Introduction to Computer Vision

## 1.1 A brief survey

Computer Vision is a branch of AI that aims to artificially simulate the human visual system by extracting features of the real world starting from images and videos usually taken by a camera [1] [2] [3]. It includes several tasks with different goals [4]: image classification, object detection, object tracking, semantic segmentation, instance segmentation, and many more.

However, outsourcing the human visual capabilities to machines is not a trivial question: the structure and the functioning of the visual cortex provide humans with an incredibly good tool, hardly reproducible, to recognize what they see in terms of colors, shapes, edges, and much more, even in the case of significant variations [5] [6]. A computer, on the other hand, has a completely different way to handle visual information: it interprets the physical world as a huge collection of numbers corresponding to pixels' color and intensity, treating digital images as multi-dimensional arrays consisting of one or more channels according to the reference chromatic scale, that is, color or grayscale [7].

Despite that, the interest of researchers in CV themes emerged and evolved from the 60s onwards: Roberts, Marr, and others gave a fundamental contribution to the evolution of artificial vision through their pioneering works [8]. New studies have been then conducted on the possibility of merging CV at its primal stage with other intertwined fields, such as digital image processing, pattern recognition, and computer graphics [2]. The real breakthrough came anyway with the birth and the success of Artificial Neural Networks, currently an interesting effective solution for the most diverse AI problems, especially those concerning vision. A more detailed description of what they are and how they work will be given later, while this section emphasizes how they revolutionized the CV field despite being born independently.

Already in the 40s and 50s, some initial steps were taken with algorithms that tried to emulate the behavior of the human brain: the goal was to create something able to learn, in a sense, through experience, as human beings do. These studies led to the forerunner element of ANNs: the so-called *perceptron* [9].

In the following years, the popularity of similar tools increased a lot, and a new family of architectures, known as Convolutional Neural Networks, has been proposed. LeNet-5 [10], one of the earliest of this kind, turned out to be quite good for simple vision tasks like handwritten digit recognition, consisting of reading and understanding numbers from 0 to 9, written by humans, from papers, documents, photos, etc.

Despite the encouraging results, neural networks started to be on the wane in the late 90s and early 2000s because of still limited computational resources and lack of data at that time [11] [12]. It was only many years later that, thanks to the increasing spread of GPUs and an even-greater availability of data, together with innovative ideas, they regained their popularity until they became the current state-of-the-art methodology for most AI tasks, including CV-related ones [11] [12].

In the next sections, some key concepts for understanding which techniques can be applied to artificial vision problems are illustrated: what processing an image means, how a machine can be "trained" to make it capable of learning and improving at specific tasks, thus which are the basic principles of neural networks. Then, an explanation of the Transfer Learning approach is given, and, to conclude, some critical aspects of CV are briefly discussed.

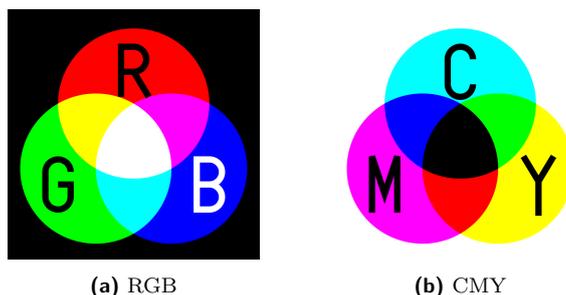
## 1.2 Color models

As stated before, computers use multi-dimensional arrays to store digital images, with each value in the array corresponding to a pixel. In order to understand how these data structures are filled, an explanation of the most used color models is required. In simple words, a color model is a mathematical way to represent colors as a combination of numbers associated with primary colors, typically three or four [13]. In the case of monochrome pictures, the representation is quite intuitive: given a 2D matrix, each element is the level of gray in the grayscale; whilst, regarding colored ones, there exist multiple viable ways, some of them illustrated below.

### 1.2.1 RGB and CMY

As shown in Figure 1.1, RGB and CMY [14] are strongly correlated, since the primaries of one are the secondaries of the other, which gives them a sort of complementarity property.

RGB uses red, green, and blue as primary colors, with a value range of  $0 \div 255$  for



**Figure 1.1:** RGB and CMY color models. Images taken from [15].

each,<sup>1</sup> according to the 24-bit representation (8 bits per channel) adopted by the majority of current digital devices. The combination of the three maximums, that is 255-255-255, gives pure light, which means white color. For this reason, RGB is an *additive model*, in the sense that increasing values of primaries give more light. The opposite combination 0-0-0 generates darkness instead, thus black color. What is more, some tools use a fourth additional *alpha* channel to regulate opacity, resulting in an RGBA model.

CMY uses cyan, magenta, and yellow as primaries, with a percentage value for each. In contrast to RGB, the maximal combination 100%-100%-100% gives black here, whereas the minimal one 0%-0%-0% gives white. Hence, the opposite functioning of this model is clear: CMY is a *subtractive model* indeed, which means that decreasing values of primaries are needed for increasing light. However, it should be said that more than three colors are used for implementing the CMY model in many practical cases. For instance, printers use black as the fourth primary because of the ink transparency, producing a CMYK model. The reason is that the maximal combination of cyan, magenta, and yellow does not provide enough darkness, with a plain black tonality as result; but, by adding a 100% black extra component and properly tuning the C, M, and Y percentages, a richer black tonality is obtainable.

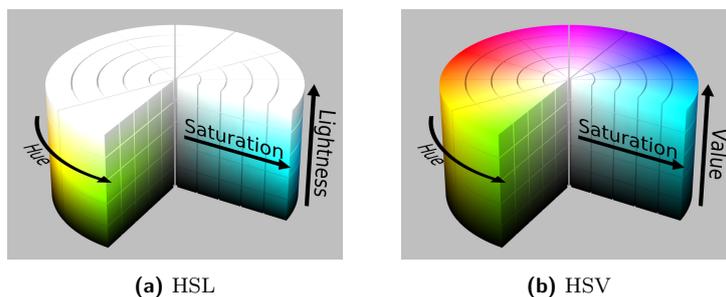
In conclusion, it is noted how some tools may use a reversed system for color memorization in computers: for instance, BGR triplets instead of RGB ones [16]. Nevertheless, this does not affect reading and processing operations: a BGR digital image is simply an RGB one with inverted channels.

### 1.2.2 HSL and HSV

The RGB model presented before is efficient for image display, but it does not exactly reflect the way human beings perceive colors. Over time, researchers discussed the possibility of representing all the possible color variations that the human eye

---

<sup>1</sup>Actually, it is often convenient to normalize pixels, with values mapped to the range  $0 \div 1$ , for faster computations.



**Figure 1.2:** HSL and HSV color models. Images taken from [13].

can capture up to propose new alternative models. Among them, HSL and HSV [14], depicted above, have the capability of treating colors of each hue through a cylinder slice, so that:

- The rotation around the central axis, expressed in degrees, gives hue in both models;
- The shift along the radius from the origin outwards gives saturation in both models;
- The shift from the bottom up gives lightness in (a) and value in (b).

Typical ranges used for implementing them are  $0 \div 360$  degrees for hue and  $0 \div 255$  for the other two parameters. Anyway, they can vary in accordance with the software/device adopted: in some cases,  $0 \div 179$  degrees is used as a range for hue, keeping the 8-bit representation.

### 1.2.3 Conversions

Clearly, given a digital image represented with a certain color model, it is possible to move it to another format, among those just discussed, by performing specific operations. In this respect, some color conversion formulas are available in [14]. It should be noticed that different and valid approaches could exist for the same generic conversion from model A to model B.<sup>2</sup>

As can be guessed,  $RGB \leftrightarrow CMY$  is quite trivial, since they are complementary color models, as previously said: for example, starting from an RGB image, it is possible to obtain its CMY counterpart by simply subtracting the R, G, and B normalized values from 1, obtaining the respective C, M, and Y percentage quantities. The same goes for the opposite conversion. In the case of CMYK format instead, the

---

<sup>2</sup>From now on, the notation  $A \leftrightarrow B$  will be used to simultaneously indicate the conversion from A to B and the one from B to A.

process also involves the computation of the extra black component, and the formulas for conversion become a little more complex.

What is more, conversions like  $\text{RGB} \leftrightarrow \text{HSV}$  and  $\text{RGB} \leftrightarrow \text{HSL}$  are surely more interesting, since the absence of evident correlations between the models involved makes the transformations not immediate. In particular, the focus of this subsection is on the passage from RGB to HSV, considering that such a conversion has been used in this work to facilitate color detection, as will be shown later. Thus, given an RGB normalized image, with pixels in the range  $0 \div 1$ , a way to move it to the HSV format is the following:<sup>3</sup>

$$\begin{aligned}
 V &= \max(R, G, B) \\
 S &= \begin{cases} \frac{V - \min(R, G, B)}{V} & \text{if } V \neq 0 \\ 0 & \text{otherwise} \end{cases} \\
 H &= \begin{cases} \frac{60(G-B)}{V - \min(R, G, B)} & \text{if } V = R \\ 120 + \frac{60(B-R)}{V - \min(R, G, B)} & \text{if } V = G \\ 240 + \frac{60(R-G)}{V - \min(R, G, B)} & \text{if } V = B \\ 0 & \text{if } R = G = B \end{cases} \quad (1.1)
 \end{aligned}$$

If negative, the hue value is turned into positive by adding 360. Then, according to what has been said in 1.2.2, the quantity can be divided by 2 in order to make it conform to the 8-bit memory encoding. In this way, at the end of the process, the output HSV-converted image will have H in the range  $0 \div 179$  degrees, while S and V in the normalized range  $0 \div 1$ . Clearly, it is sufficient to multiply S and V by 255 to remap them in the standard  $0 \div 255$  range.

### 1.3 Image processing

Image processing concerns all those actions aimed at processing digital images to transform and make them suitable for any kind of analysis, generally made as a pre-processing step for fuller CV algorithms and consisting of mapping pixels from an input image to an output one [3].

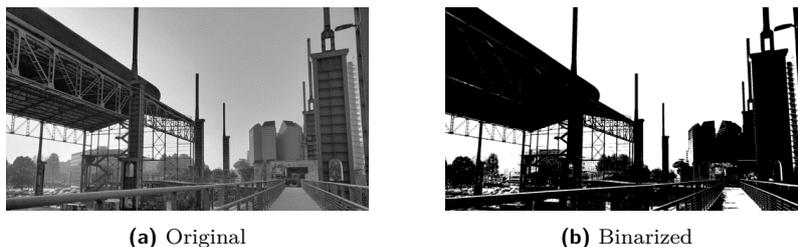
The next subsections provide a discussion of different types of common image processing operations employed in today's CV applications.

---

<sup>3</sup>This is one possible procedure (also available [here](#)), but other approaches exist. Moreover, the fact that some tools use BGR as the standard model for digital images does not affect the validity of the conversions involving RGB, keeping the formulas unchanged.

### 1.3.1 Binarization

With regard to images, binarization is a technique that consists of mapping each pixel of a given grayscale image from its standard discrete range  $0 \div 255$  (with possible values  $0, 1, 2, \dots, 255$ ) into one between 0 and 1, i.e. black or white [3]. The simplest approach to do that is the following: if a pixel exceeds a certain threshold value, it is converted to 1; if below such value, it is set to 0. In this way, a pure black-and-white image is generated, with just 1 bit required for color representation in memory.

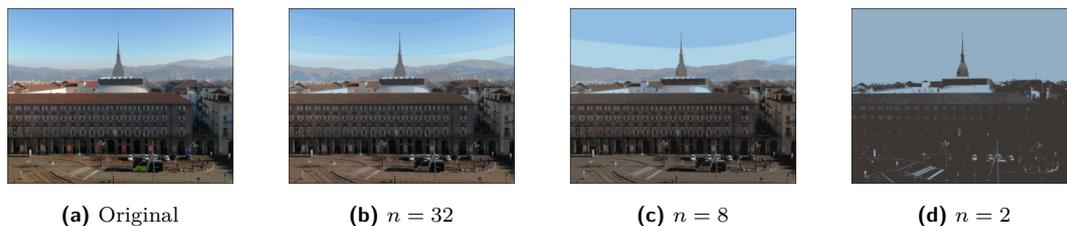


**Figure 1.3:** Example of binarization with global threshold  $t = 127$ .  
Original from Author.

The choice of such threshold is fundamental to obtain decent results: as in the example above, a first trivial solution is to globally set it to 127 [17], the median number in the  $0 \div 255$  range, but unforeseeable events such as varying lighting conditions and difficulties in distinguishing foreground objects from background could affect the input image, making this method inefficient. An alternative is the one proposed by Otsu [18], which automatically groups pixels into foreground and background based on a threshold autonomously found, not requiring any parametrization. Other binarization techniques provide an adaptive thresholding approach to the problem, identifying different regions with different illuminations and properly determine the specific threshold value to be used in each area, or even for each pixel [19]. In order to do that, a pixel is not considered singularly, but some neighborhood values around it are analyzed too, performing mathematical operations (for instance, mean or weighted sum) between all the ones involved in such region.

### 1.3.2 Quantization

Quantization consists of reducing the color palette of an image, where palette refers to the totality of colors in it, with the aim of preserving relevant information while reducing the quantity of RGB triplets to be stored [20]. Obviously, the more the number of colors decreases, the less the resulting image is true to the original: a poor palette could make shapes and borders of objects hardly recognizable indeed. In this respect, Figure 1.4 shows the effects of quantization on a digital image. Among the many current algorithms to do that, Median Cut [21] is one of the fastest



**Figure 1.4:** Example of quantization to vary the number of colors  $n$ .  
Original from Author.

and the most employed in the context of image processing. Simple but effective, it works as follows:

---

**Algorithm 1** Median Cut

---

1. Given an RGB image, find the channel with the greatest range.
  2. Sort pixels according to the values of that channel.
  3. Split them into two groups on the basis of the median pixel.
  4. Repeat for each group until the desired number of colors (groups) is reached.
- 

At the end of the process, there exist  $n$  groups, being  $n$  prefixed. The average RGB triplets calculated for each one form the new image color palette, whose dimension is dependent on the number of iterations, with a base-2 exponential growth: 2, 4, 8, etc. Therefore, if a 32-color quantization is requested, the process above is run  $2^5 = 32$  times. When  $n$  is not a power of 2, the algorithm gives a quantity that is the greatest power of 2 closest to  $n$ ; then, some of the colors must be aggregated in some way to exactly obtain  $n$ .

### 1.3.3 Geometric transformations

Another interesting aspect in the image processing field is the possibility of spatially transforming digital images and video frames, thus performing geometric transformations [22]. It can be a crucial step in those applications which intend to highlight specific features or information related to the image content, as well as data augmented is needed (this last concept will be investigated later).

A matrix is required for giving images the new desired appearance: in this sense, geometrical transformations boil down to matrix-vector multiplications, since an image is itself a matrix and every pixel inside can be expressed as an  $x$ - $y$  pair, where  $x$  is the column and  $y$  is the row of such pixel's location inside the image matrix. Anyway, since transformations matrices are compliant with the 3D space, the pixel vector must be adapted accordingly. To this end, an augmented vector of homogeneous coordinates  $[x, y, 1]$  is used for representing the 2D pixel points. As a

Transformation	Matrix	Example
Identity	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	
Translation	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix}$	
Rotation	$\begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$	
Scaling	$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$	
Shearing	$\begin{bmatrix} 1 & h_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ or $\begin{bmatrix} 1 & 0 & 0 \\ h_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	
Flipping	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ or $\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	

**Table 1.1:** Common geometric transformations in image processing. Original (first row image) from Author.

result, a generic geometric transformation can be written as follows:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (1.2)$$

where  $x'-y'$  is the new pixel pair obtained from the original one  $x-y$ , while the the submatrix obtainable by removing the third column assumes different forms depending on what is to be done, as shown in Table 1.1. Hence, distinct parameters must be set up according to the transformation selected: for instance, shifts  $t_x$  and  $t_y$  for translation, angle  $\theta$  for rotation, scale factors  $s_x$  and  $s_y$  for scaling, and shear factors  $h_x$  and  $h_y$  for shearing. There are cases when more than one matrix is possible: the already mentioned shearing and flipping, which both use one over the other based on whether they are performed along  $x$ -axis or  $y$ -axis.

In closing, an operation of this kind is called *affine*: it means something that

preserves lines and their parallelism, with the possibility of changing angles and distances instead. (1.2) is the general formula for affine transformations indeed.

### 1.3.4 Filtering and morphological transformations

Although a filter is in effect a matrix, it performs calculations in a completely different way compared to the typical algebraic matrix multiplications. In fact, filtering an image implies doing particular operations such as *convolution* ( $*$ ) and *cross-correlation* ( $\star$ ) [23], both consisting of summing the terms of an element-wise product. They are expressible with the following formulas, respectively:

$$\begin{bmatrix} x_{11} & \cdots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \cdots & x_{mn} \end{bmatrix} * \begin{bmatrix} y_{11} & \cdots & y_{1n} \\ \vdots & \ddots & \vdots \\ y_{m1} & \cdots & y_{mn} \end{bmatrix} = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} x_{(i+1)(j+1)} y_{(m-i)(n-j)} \quad (1.3)$$

$$\begin{bmatrix} x_{11} & \cdots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \cdots & x_{mn} \end{bmatrix} \star \begin{bmatrix} y_{11} & \cdots & y_{1n} \\ \vdots & \ddots & \vdots \\ y_{m1} & \cdots & y_{mn} \end{bmatrix} = \sum_{i=1}^{m-1} \sum_{j=1}^{n-1} x_{(i+1)(j+1)} y_{(i+1)(j+1)} \quad (1.4)$$

In both, the matrix on the left is the image patch covered by the filter (i.e., the image portion with the same dimension) containing pixel values  $x_{ij}$ , while the one on the right is exactly the filter containing values  $y_{ij}$ .

Practically, filtering means to shift along the entire image seeing all the pixels in groups, with a parameter called *stride* specifying the step size of such movement. In some cases, an outside border called *padding* is added to the input image, setting all such pixels to 0 or something. At this point, given the input size  $n_{in}$ ,<sup>4</sup> the filter dimension  $f$ , the stride  $s$  and the padding  $p$ , the output size  $n_{out}$  can be calculated beforehand in this way:

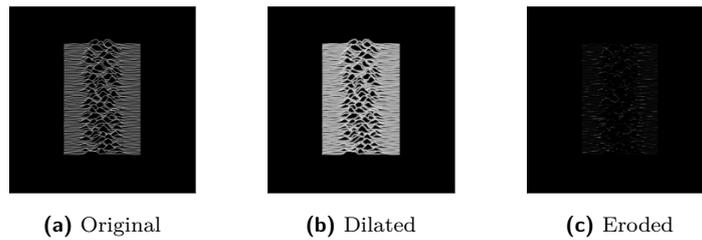
$$n_{out} = \frac{n_{in} + 2p - f}{s} + 1 \quad (1.5)$$

It should be noted that being the image and the filter both squared, only one dimension (which is the same horizontally and vertically) is specified but, more in general, it is necessary to provide width and height when different from each other. Anyway, in most of today's CV tools, images are reshaped to be squared and convolutional filters are picked as non-rectangular too, typically "odd-sized" like 3x3, 5x5, 7x7, and so on. The reason behind that is given by the functioning principles of CNNs, as will be seen further below. According to (1.5), it is also possible to apply a certain padding border to the input image so that the output shape remains the same, being all the other parameters previously fixed.

All the formulas presented so far refer to single-channel images; extending to the

---

<sup>4</sup>The input image is taken as squared, thus  $n \times n$  shaped. The output will be squared too.



**Figure 1.5:** Two main morphological transformations in image processing. While dilation adds pixels to the object boundary, erosion removes pixels from it. Original generated by Python script [\[link\]](#).

multi-channel case, they are still valid with the difference that output values are summed along channels so that an RGB  $n_{in} \times n_{in} \times 3$  input patch is mapped into a single-channel output one  $n_{out} \times n_{out}$ . In this way, multiple filters can be used to augment volume depth, as usual in CNNs, as will be seen later.

The values inside filters must be chosen accordingly to what is to be found in the source image: in this sense, those used in digital applications are helpful when it comes to performing edge detection, sharpening, blurring, and more. Besides, filters allow a set of particular operations based on the morphology theory<sup>5</sup> and consequently referred to as morphological transformations [3]. Dilation and erosion are classic examples: both aim to change the shape of objects in binary images to highlight or hide some details. As can be seen in Figure 1.5 above, dilation makes objects thicker, while erosion makes them thinner. What is more, it is possible to perform such operations sequentially on the same image: an erosion followed by a dilation is an opening, whereas a dilation followed by an erosion is a closing. The former is useful for noise reduction, the latter helps to remove undesired points from within the foreground objects. Filters used for morphological transformations are all-ones matrices, dimensioned as said before.

## 1.4 Machine Learning and ANNs

Machine Learning concerns the development and the use of intelligent systems capable of finding relationships in data of a given starting set, called *training set*, and making future predictions on new data on their own, that is, without requiring external explicit instructions [12]. In other words, a ML tool is, in general, an algorithm that enables computers to learn something and use such knowledge to perform a specific task, so that some inputs are given and some outputs are

---

<sup>5</sup>It is a mathematical theory which concerns the analysis of images, with the focus on shapes, for component and information extraction.

returned. As a result, a differentiation can be done: the first stage of knowledge acquisition goes by the name of *training*, the second stage of knowledge exploitation is usually referred to as *inference* [24].

### 1.4.1 Supervised vs. Unsupervised Learning

The fact of knowing the real output for each input of the training set can be resumed in the Supervised Learning paradigm [1] [12] [25], according to which labeled data are given to the model. In such cases, even a new set of never seen data can be created and fed into the model to receive predictions and evaluate the performance based on how the estimated outputs differ from the true ones. At this point, it is clear that the adjective "supervised" refers to a situation in which some a priori knowledge is provided by an external instructor: for instance, a human who possesses the true outputs related to inputs. Regression and classification problems are good examples in this sense. Popular and efficient algorithms to perform such tasks are Linear Regression, Logistic Regression, SVM, K-NN, Random Forest, and others, all reviewed in [26]. However, they are not deepened in this work, but only a general overview on regression and classification categories is given.

On the other hand, there are scenarios in which data are not labeled, making the supervised approach not suitable anymore. In these cases, the opposite Unsupervised Learning paradigm [1] [12] [25] is embraced. Here, the model does not receive any known example, thus it must be able to find relationships in data based on patterns and common characteristics discovered during training. Clustering [27], dimensionality reduction [28], and generative methods [29] are typical examples of that, but only the first one is discussed in this work.

#### Regression and classification

Given a collection of data representing known examples, typical supervised ML-solvable problems can be grouped into the two following macrofamilies [30]:

- *Regression*, that is estimating a continuous curve that fits data;
- *Classification*, that is predicting labels, from a set of  $N$  identifiable classes, and assign them to data using *label encoding* or *one-hot encoding* [31], as illustrated in Table 1.2.

Inside each group, a further differentiation can be done: for instance, there exist linear and polynomial regression problems, depending on the grade of the function selected to approximate data trend (this choice must be made carefully); or binary and multiclass classification problems, based on how many classes of belonging the algorithm is supposed to find: two or more.

Both regression and classification can suffer problems of "bad fitting", in the sense

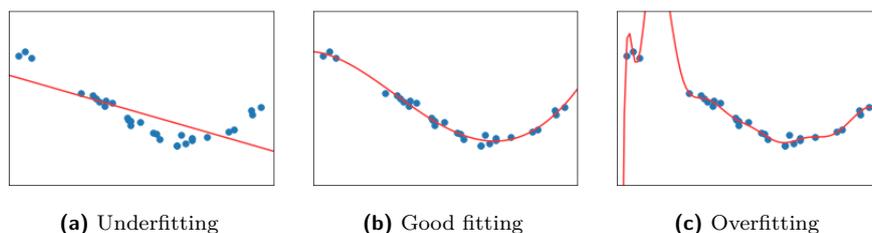
	Label	One-hot
Class 1	0	[1, 0, ..., 0]
Class 2	1	[0, 1, ..., 0]
...	...	...
Class $N$	$N - 1$	[0, 0, ..., 1]

**Table 1.2:** Label encoding vs. one-hot encoding.

that the estimated model fails to make accurate predictions on new data constantly, being subject to one of these two problems [32]:

- *Underfitting*, when it is not able to capture enough in data relationships;
- *Overfitting*, when it fits too much to training data with the risk of being too far from new samples when it comes to making predictions on them;

Both are visually reported below, together with the desirable case in the middle. Efforts are needed to find a possible cross between (a) and (c) obtaining a situation



**Figure 1.6:** Comparison between bad and good model fitting.

like (b), even if overfitting remains a very common problem in nearly all the ML applications. Going more specifically, an underfitting model is mostly simple, which suggests that few parameters may have been selected; on the other hand, an overfitting model could be too complex for its designed task, resulting in an excessive number of parameters. Thus, in presence of such criticalities, a first trivial approach might be to increase or decrease the model complexity. Anyway, when ANNs are involved, these two problems, especially overfitting, could be not that easy to solve and other techniques should be concept will be explored later.

So far the focus has been on how ML-solvable problems can differ one from the other and potential critical issues which may occur. At the beginning of this subsection, the basics of machine knowledge acquisition have been mentioned, but without going into detail. To be more precise, a tool of this kind constantly updates a function  $J$  to be minimized, called *cost function*, including true outputs  $y$  and estimated ones  $\hat{y}$ . It assumes different form depending on the task and the problem formulation: for instance, it can be an MSE in regression problems, or a CE in

classification problems, as will be seen further on. The *parameters* contained in  $\hat{y}$  are initialized (randomly, for example) and then adjusted at each iteration of the so-called Gradient Descent algorithm [33], a popular mathematical approach to the continuous optimization of functions through the gradient, that is the collection of the partial derivatives. The idea is to descend in the opposite direction of the gradient so that, in the case of convex functions, the minimum is reached after some iterations. The general steps are reported hereunder:

---

**Algorithm 2** Gradient Descent

---

```
 $\theta_0 \in \mathbb{R}^n, k = 0$   
while  $\nabla J(\theta_k) \neq 0$  do  
     $\theta_{k+1} = \theta_k - \alpha_k \nabla f(\theta_k)$   
     $k = k + 1$   
end while
```

---

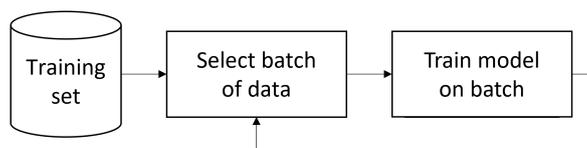
where the model parameters have been considered as part of vector  $\theta$ : for instance, in the simplest case of univariate linear regression,  $\theta = [w, b]$ . This generates a line with slope  $w$  and intercept  $b$ , the latter also known as *bias*.

The fundamental quantity which determines the size of a descent step is  $\alpha$ , called *learning rate* in the context of ML applications. This parameter must be positive and tuned properly: too small values could cause slow convergence, while too large values may result in an oscillating behavior. However, it is not so trivial to find a suitable middle ground for that.

As can be noticed, GD involves the simultaneous update of parameters at each new iteration: in this sense, it is said that the ML model learns such values, getting every time new correlations between data.

Moreover, it can be seen that the algorithm theoretically ends when the gradient reaches 0: this means that the function is exactly located at its minimum point when convex. Anyway, such a situation is quite difficult to obtain in a real scenario, mostly due to the difficulties in calibrating  $\alpha$ , thus a stop criterion is usually defined to determine the end of GD. For instance, a maximum number of iterations can be specified, or a tolerance under which the convergence is held to be achieved.

Nevertheless, there exist many ML real scenarios in which the cost function is not convex in its entire domain: for example, it may present some local minima instead a unique absolute one. In such situations, the GD algorithm could fall into the trap of mistaking a local minimum as the absolute of the function and then stop. This is not a solvable problem: designers must pay attention in building their ML tool to guide the descent into the absolute minimum, as far as possible, or into a decent local one at least. The shape of  $J$  is always characterized by high complexity, not being predictable in advance in many cases; consequently, it is not easy to understand which kind of minimum the function is bound to during the learning phase. This remains an open problem in the ML field.



**Figure 1.7:** Batch learning. Image from Author.

One last remark: the one presented above is the Batch version of GD, in the sense that samples are picked all together as a single *batch* having the same size as the training set every iteration. It is also possible to adopt a Mini-Batch descent, with data taken in smaller groups so that parameters are adjusted more times during the same iteration every time a batch of samples is seen. This concept extends to nearly all the supervised algorithms and simply works as schematized in Figure 1.7. At this point, it can be seen how some parameters, beyond those contained in  $J$  formulation which are to be learned during training, are to be tuned before the procedure starts: the reference is to learning rate and batch size. Quantities like these are called *hyperparameters* in the jargon of ML, given that they are not to update over iterations but are prefixed and kept unchanged impacting on the efficiency of the descent. As will be seen later, nearly all the ML tools and algorithms, especially ANNs, have hyperparameters to be tuned.

In addition to the vanilla GD discussed here, there exist different and multiple variants of this algorithm to date: among them, Stochastic Gradient Descent (SGD) [33] is probably one of the most popular.

## Clustering

The aim of clustering is to divide data space into groups called *clusters* and distribute data among them on the basis of patterns autonomously found during training (see Figure 1.8). One of the most popular algorithms to do that is certainly K-Means, concurrently referable to Lloyd [34] and Forgy [35] in its standard version, and reviewed in [27]. It essentially works as follows:

---

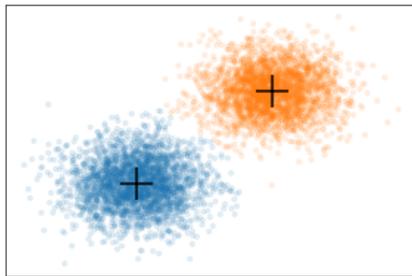
### Algorithm 3 Standard K-Means

---

Let  $k$  be the number of clusters to be identified.

1. Get initial clusters with centroids  $c^{(1)}, \dots, c^{(k)}$ .
  2. For each data  $x^{(i)}$ :
    - a. Compute its Euclidean distance from each centroid.
    - b. Assign it to the cluster with the nearest centroid.
  3. Recompute centroids as the mean all over data in their clusters.
  4. Repeat 2-3 until a stop criterion is satisfied.
- 

As anticipated, the one above is precisely the standard version, also called Batch



**Figure 1.8:** Example of 2D data clustering with two clusters.

K-Means since data are picked and processed all at once at each new iteration. A Mini-Batch version of it also exists [36], with data being selected in groups, helpful in case of numerous samples or insufficient memory.

Regardless of the version considered, there are basic functioning principles of K-Means anyway. Firstly, the so-called *centroids* are, in some way, the points representing clusters, and they are computed as the mean vector of all the data assigned to their relative cluster, as specified above. Then, the algorithm stops when a pre-defined condition is fulfilled, with multiple possibilities of implementing that:

- Stop when centroids do not change anymore;<sup>6</sup>
- Stop when data continue to be assigned to the same clusters;
- Stop when a prefixed number of iterations is reached.

Some of these can be combined: for example, centroid variations and number of iterations can be checked at the same time so that, if the algorithm is not able to converge in terms of centroids, at least it will not run for too long. [27].

In closing, the choice of initial clusters (thus, centroids) is not trivial and can influence the convergence time: a more careful initialization, as the one presented in [37], is preferable than a random start.

At this point, a brief discussion about the choice of  $k$ , i.e. the number of clusters to be generated, is needed, being the fundamental hyper-parameter to be tuned in such a context. One simple way to get an optimal value is given by the heuristic *elbow method* [38], a generic and simple technique, valid for all clustering algorithms beyond K-Means, which aims to find the best number of clusters by simply plotting the grouping distortion to the vary of  $k$ , resulting in a decreasing curve that folds at an elbow point: here is the optimal value.

---

<sup>6</sup>As in many computer applications, a tolerance is introduced to determine if something has stopped updating, since it is almost impossible to have the exact same quantity between two consecutive iterations.

As said before, the one discussed is the standard version of K-Means, but other unsupervised clustering approaches are possible, some of those included in [27].

### 1.4.2 Performance evaluation

While cost functions are used to update weights making a ML model able to learn, some metrics are needed to have an estimate about how good such a model is or, in other terms, how accurate its predictions are. Clearly, as in the case of cost functions, the choice and the shape of metrics are dependent on the task. One remark: all the metrics that are going to be presented are reported in [26]. As far as generic regression problems are concerned, two typical examples are:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2 \quad (1.6)$$

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i| \quad (1.7)$$

which stand for Mean Squared Error and Mean Absolute Error respectively. It is quite intuitive to understand why these functions are helpful in this context: since regression algorithms aim to find the best curve fitting to data, a simple measure of their goodness is given by how precise the predictions are in terms of distance from the true output values. Developers can adopt one or more metrics depending on the requirements.

On the contrary, when it comes to classifying, the error functions above lose their meaning. In the specific case of binary classification, precision and recall are generally used instead:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \quad (1.8)$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad (1.9)$$

As can be seen, the former expresses how many true positives exist among all the values predicted as such, while the latter considers how many values are correctly predicted as positives on the totality of positives. In this sense, precision is a measure of fidelity and recall is a measure of completeness. Since one tool<sup>7</sup> could outperform another in terms of precision but fail in terms of recall, an aggregative metric called  $F_1$ -score, consisting of the harmonic mean of the two quantities, is introduced to make the comparison sensible.

Then, the typical metric for multiclass classification is accuracy, a sort of overall precision that takes into account both true positives and true negatives, resulting in

---

<sup>7</sup>In this context, it indistinctly refers to algorithms, neural networks, and any other method.

the ratio between the number of correct predictions and the totality of predictions made:

$$\text{Accuracy} = \frac{\text{\#correct predictions}}{\text{\#predictions}} \quad (1.10)$$

However, it should be noticed that precision and recall are sometimes employed in multiclass problems too and, complementarily, accuracy is usable for evaluating binary classification performance, although the latter is a rare case. Clearly, when  $P$  and  $R$  are used for multiclass, the concepts of positive and negative are not available anymore and the focus is on one class at a time. Everything said about precision and recall can be graphically summarized through the so-called *confusion matrix*, where actual and predicted values are arranged in rows and columns.

What is more, precision and recall are especially used to deal with imbalanced datasets [39], that is, in all those cases where a class is overrepresented compared to the others, in the sense that the majority of the samples available belongs to such class. Thus, it is understandable how accuracy could distort the perception of model goodness, while precision and recall could lead to better evaluations: in fact, accuracy considers the correct predictions over all the predictions done, in an absolute way, while precision and recall can take into account the performance on each single class individually.

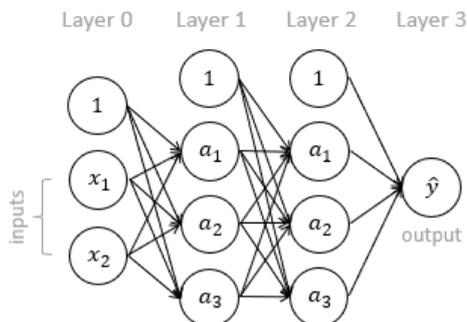
In closing, these are only a few of the possible metrics usable for the performance evaluation of ML algorithms. More complex situations, e.g. ANN-oriented tasks such as object detection, often require the adoption of sophisticated yardsticks, as will be seen further on. In various applications, one feasible way to obtain a new upscale metric is to combine together some of those just discussed.

### 1.4.3 Basic principles of ANNs

Artificial Neural Networks [1] [25] consist of elementary computing elements, called *units*, organized in *layers* and interconnected in a structure inspired to that of biological neural networks. As shown below, their structure generally includes: an input layer containing the input data, shapeable as an  $n$ -dimensional array  $x_1, \dots, x_n$ ; a certain number of hidden layers containing sorts of intermediate values; at the end, an output layer returning the predicted values  $\hat{y}_1, \dots, \hat{y}_n$  relative to the given inputs, with a number of units depending on the task. It should be noted, however, that the input is not an actual layer of the network and it is not considered in the total layer count (in fact, it is marked as Layer 0 in Figure above).

The number of layers determines the *depth* of a network. Many studies have shown that a deeper architecture is preferable, enabling machines to learn more about data: here the term Deep Learning is used instead of Machine Learning when deep multi-layer ANNs are used for performing tasks.

The values contained in the artificial neurons are often referred to as *activations*,



**Figure 1.9:** Example of simple ANN architecture with two inputs, two hidden layers with three activations for each, and one single output. The input layer is not in the count of total layers, while the units set to 1 are included to deal with biases. A structure like this is often called MLP, i.e. Multilayer Perceptron [11].

Image from Author.

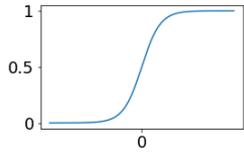
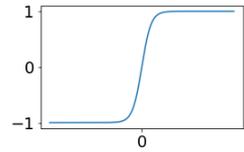
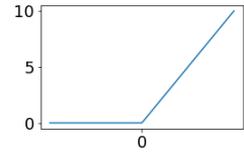
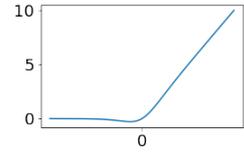
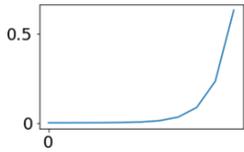
and the following formula explicits the computation of activation  $i$  at layer  $j$ :

$$a_i^{[j]} = g(W_{i0}^{[j]} a_0^{[j-1]} + W_{i1}^{[j]} a_1^{[j-1]} + \dots + W_{in}^{[j]} a_n^{[j-1]}) \quad (1.11)$$

where  $W$  is the *weight matrix* linking layers  $j - 1$  and  $j$ . As can be guessed, there exists a matrix like that for each transition between successive layers. The weights contained are the so-called *parameters* of the network, i.e., the numbers that are updated by the network itself at each iteration of the learning phase.

To be more precise, the multiplication between the weight matrix  $W$  and the activation vector  $a$  at a certain layer involves in general an extra term, known as *bias*, to be added to the result. For instance, in the simple case of a single-output architecture with no hidden layers, the output value can be trivially written as  $y = Wx + b$ , where  $b$  is exactly the bias. In such situation,  $W$  and  $x$  are both  $n$ -dimensional, being  $n$  the size of input vector. An alternative way to write that expression consists of putting weights and bias together in  $W$  and accordingly adding a 1 entry at the top of  $x$ , changing both  $W$  and  $x$  dimension into  $n + 1$ . As a result,  $y = Wx$ . This concept can be extended to the more generic case of larger networks with an arbitrary number of inputs, outputs, and hidden layers in between, by adding a first component  $a_0 = 1$  to each activation vector and including the bias vector in each matrix weight row as  $W_{i0} = b_i$ , as done in 1.11. In this respect, Figure 1.9 shows how the first unit of each layer is fed into the next layer units, but it does not have any incoming link, accordingly to what has been just said about bias.

Formula (1.11) also highlights the fact that a nonlinear function  $g$  is applied to the result of the weights-activations product, so that the final value of  $a_i^{[j]}$  is a little bit transformed. Such  $g$  is called *activation function* [40] in the context of ANNs, and its introduction is due to the frequent necessity of separating data in a nonlinear way [25]: in the jargon, it is said that the network learns "something nonlinear",

Name	Plot	Equation	Mapping
Sigmoid		$f(x) = \frac{1}{1 + e^{-x}}$	$\mathbb{R} \rightarrow (0, 1)$
tanh		$f(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$	$\mathbb{R} \rightarrow (-1, 1)$
ReLU		$f(x) = \max(0, x)$	$\mathbb{R} \rightarrow [0, +\infty)$
SiLU		$f(x) = \frac{x}{1 + e^{-x}}$	$\mathbb{R} \rightarrow [-0.28, +\infty)$
Softmax		$f(x)_i = \frac{e^{x_i}}{\sum_{j=1}^N e^{x_j}}$	$\mathbb{R} \rightarrow (0, 1)^N$

**Table 1.3:** Some common non-linear activation functions.

and this enables it to better perform on the most diverse tasks. Table 1.3 reports typical activation functions utilized today. They take a quantity as an argument, here the result of the multiplication between  $W$  and  $a$ , and they convert it into something else: for example, sigmoid maps its input to the interval  $(0, 1)$ , while ReLU keeps its input unchanged when positive and sets it to 0 when negative. Sigmoid is always employed as the activation function of the last layer's single neuron of networks designed for binary classification since in such applications the output is expected to be false or positive, thus 0 or 1. As the effect of mapping, values above 0.5 are positive, and those below are false. It is possible to change a different threshold from the default one: for example, greater than 0.5 if high confidence for positives is required, or smaller than 0.5 if the aim is lowering false negatives (i.e., 1-outputs incorrectly predicted as 0).

ReLU, on the other hand, is largely employed in neural networks in the intermediate layers, having outperformed sigmoid and tanh in this role, providing higher

robustness to the problem of zeros in the gradient and sparse activation too [41]. SiLU, first proposed in [42] for Reinforcement Learning applications, is the multiplication between the input and the sigmoid function, resulting in a behavior very similar to that of ReLU, as can be seen in Table 1.3. It is approximately zero for  $x \rightarrow -\infty$  and approximately  $x$  for  $x \rightarrow +\infty$  indeed. The fact that there is a minimum in  $(-1.28, -0.28)$  gives SiLU an implicit regularization property that avoids weight updates of large magnitude.

To conclude, softmax is a peculiar activation function, different from the others discussed since it has a discrete domain. It is used in ANNs for multiclass classification as the activation of each last layer’s neuron. Its formula provides the probabilities associated to the classes so that the final neuron quantities are transformed into something more intuitive and, more importantly, greater values are brought out with respect to smaller values. For example:

$$\begin{bmatrix} 2.45 \\ -0.45 \\ 1.02 \\ -1.63 \end{bmatrix} \xrightarrow{\text{softmax}} \begin{bmatrix} 0.7626 \\ 0.0420 \\ 0.1825 \\ 0.1289 \end{bmatrix}$$

In this case, it can be said that Class 0 is the one predicted with a certainty equal to about 76%. Clearly, since softmax outputs are probabilities, their sum is 1.

Everything said so far suggests ANNs are trained complying with the supervised approach. This is often true, especially in the case of CNNs used for CV, but there also exist networks capable of learning in an unsupervised way: for instance, Self-Organizing Maps [43] and Autoencoders [44].

In this part, some of the most employed activation functions have been presented, although limited to models designed for CV. Anyway, it is important to remark that others exist [40] and may find application in different contexts.

## Datasets

Training a neural network to have good performance at a desired task requires a sufficiently large amount of input data [12]. In fact, as said in the introduction, one of the reasons that caused the decline of ANNs in the past has been the difficulty in gathering data. The model selected must be capable of fitting to training samples but, more importantly, it is expected to succeed in predicting outputs for new ones, so that its general behavior is referable to a good fitting, as specified in 1.4.1.

In the ML field, a collection of data used to feed a neural network both in its training and inference stages goes by the name of *dataset*. It can contain labeled or unlabeled samples depending on whether the learning procedure is supervised or not. It is vitally important that the dataset depicts the real scenario where the model is intended to work with high fidelity. Since networks need annotations to learn in a supervised context, the label format is also fundamental: as an example,

it is not possible to use something annotated in the same way both for image classification and object detection. The reason behind that is easy but will be clearer in the next parts, where CNNs for image classification, object detection, and other tasks will be discussed.

Datasets are generally split into three subgroups, named respectively *training set*, *validation set*, and *test set* [45], even if the last is not always present. As can be imagined, training set is the one used to train the network and, reasonably, it must be the greater portion of the original dataset. Validation set is to validate the model at the end of each training epoch instead: in other words, it is as the network were gradually tested on a small set of new data every  $n$  intervals during training, being  $n$  the prefixed number of epochs. This can help developers to understand whether the learning procedure is heading in the right direction or not. What is important to say is that the network updates its parameters while seeing training samples but no changes happen at each validation stage: here, the model is considered as definitive, although training has not finished yet, and predictions can be made on new ever seen samples. Now, a supervised ANN-based scenario is taken as an example: it is easy to deduce that training intermediate predictions are constantly compared to true outputs so that the parameters can be adjusted accordingly, while during validation the model has no knowledge about the labels associated with data and is supposed to perform as if it were used for inference. In conclusion, test set is a sort of validation set that can be employed once training ends, thus adopting the definitive tuned model. It is generally used to get and report the permanent performance. Given the starting set, typical partitions assign much more data to training and the remaining samples are equally distributed between validation and test: for instance, 80%-10%-10%, 70%-15%-15%, and such. More advanced techniques, like *cross-validation*, are also included in [45].

Among the multiple existing datasets, ImageNet [46] represents the standard for image classification, while COCO [47] and Pascal VOC [48] are the most employed for object detection benchmarks. In fact, state of the art classifiers and detectors have been trained on them and, when a new model (or family of models) is proposed, the relative performance is evaluated on such datasets. All of them include a very large amount of images representing the most diverse situations in relation to the specific task. They are quite challenging, since such a variety tests in some way the adaptability of a network to as many scenarios as possible.

Given a dataset, another important aspect is the distribution of the classes included: in fact, in the case of imbalance, the network could underperform on the less represented classes, although providing acceptable results on the others. Hence, adequate countermeasures must be taken to avoid an incorrect learning [39].

## Training

In Subsection 1.4.1 enough has been said about training machines to make them capable of performing tasks. However, the specific case of neural networks has not been deepened yet. Training modern ANNs is often harder than it might seem due to their complicated structure and the huge number of calculations to do, that is why GPUs are employed for such purpose instead of CPUs. Having more computing power, they allow to train complex models with millions of parameters to be learned indeed. At this point, a brief overview on how ANNs are treated for training is given, considering that some details will be overlooked. For a start, neural networks use GD, in one of its variants, to update weights, on par with every generic supervised algorithm. Once learning rate and batch size are fixed, the whole training procedure starts involving different batches, one per *step*, at each new *epoch*. The concept of epochs is the same as iterations seen before, with each of them divided into steps depending on how many batches compose the training set. Clearly, such information is determined by the initial choice on batch size. For instance, if training set contains 320 samples and batch size is set to 32, one epoch will last 10 steps. In summary, a step is one cycle through the samples of a given batch, while an epoch is one cycle through the entire training set.

At the end of each epoch, the model with the weights obtained until then is used for predictions on validation data to have an idea of how the learning procedure is going. Among many things, it is possible to understand whether overfitting is occurring over epochs, for example. In fact, performance on validation set is evaluated based on the metrics chosen every time.

As introduced before, a cost function is to be formulated and used for GD. In the context of ANNs, it is improperly called *loss*, referring to the mean discrepancy between true and estimated outputs over all the samples while such term should indicate the discrepancy calculated on the single generic pair  $y_i-\hat{y}_i$  instead. The form of this function depends on the type of problem to be solved. In the case of regression, MSE and MAE are typically used for that, with the same formulation as the ones introduced for evaluation. On the other hand, classification requires a different loss. In fact, being  $y^{(1)}, \dots, y^{(m)}$  the actual outputs associated with the inputs  $x^{(1)}, \dots, x^{(m)}$  and  $\hat{y}^{(1)}, \dots, \hat{y}^{(m)}$  the estimated output provided by the model, it can be Binary Cross Entropy (BCE) in binary classification problems:

$$\text{BCE} = -\frac{1}{m} \sum_{i=1}^m (y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log (1 - \hat{y}^{(i)})) \quad (1.12)$$

and Cross Entropy (CE) in the more general scenario of multiclass classification:

$$\text{CE} = -\frac{1}{m} \sum_{i=1}^m y^{(i)} \log \hat{y}^{(i)} \quad (1.13)$$

where the superscript  $i$  in both suggests a vectorial shape for inputs and outputs (both true and estimated), in a way that takes into account the generic size  $n$  for

each of them. Therefore,  $x^{(i)}$  represents the  $i$ -th input vector of size  $n$  out of  $m$ . The same goes for  $y$  and  $\hat{y}$ . Moreover, these two losses are called *categorical* when  $y$  and  $\hat{y}$  are one-hot encoded, and *sparse categorical* when they are label encoded. Nothing changes in their formulation, but there is a difference in seeking matches between true and predicted output values: in the case of one-hot encoding, the adopted loss does an index-based check; in the case of label encoding, it does a value-based check.

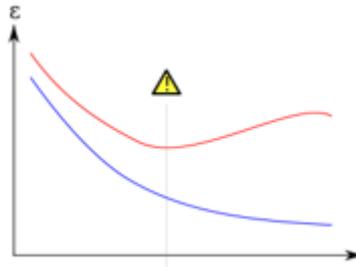
Everything said at the beginning of this subsection is about the calculation of activations, i.e. neuron values, from the first hidden layer to the output layer, as resumed in (1.11). The path to compute an activation starting from a previous layer takes the name of *feedforward*, making neural networks with a structure similar to that of Figure 1.9 also known as feedforward networks [49]. On the contrary, the backward path is the one used to update weights, thus learn, through the so-called *backpropagation* [50], which more specifically consists of the computation of derivatives, required by Gradient Descent, with the chain rule. All the modern architectures rely on these two fundamental principles: when training, they constantly go back and forth throughout layers to have a measure of the output error and accordingly adjust the learnable parameters. One downside of such procedure is that there is no guarantee in reaching the global minimum of the cost function when backpropagation is applied, but only a local minimum, due to the frequent non-convexity of cost functions. This issue had already been discussed before in relation to the GD algorithm introduction (see 1.4.1 for a recap).

## Overfitting and solutions

As discussed in 1.4.1, overfitting is a common issue when it comes to training neural networks, easy to discover but hard to solve in some cases. Likely, different strategies are adoptable to address such a problem, some of them collected in [51] and presented below.

One first trivial approach goes by the name of *early stopping* and simply consists of stopping training when no more improvements on a quantity  $q$  are recorded for a certain number of consecutive epochs, thus prematurely interrupting the process before the prefixed number of epochs is reached (see Figure 1.10). As is guessable, the absence of improvements is detected when changes within the window selected remain under a tolerance  $\delta$ . With regard to  $q$ , the more meaningful quantity to be monitored is probably the validation loss because overfitting has a visible impact on validation data the moment predictions are made on them. Over epochs, a situation with a decreasing training loss against a steady-state or increasing validation loss, just as the one in the figure, suggests an overfitting model. As last thing, many modern frameworks allow developers to constantly check more than one quantity at the same time so that training is stopped if both do not keep improving.

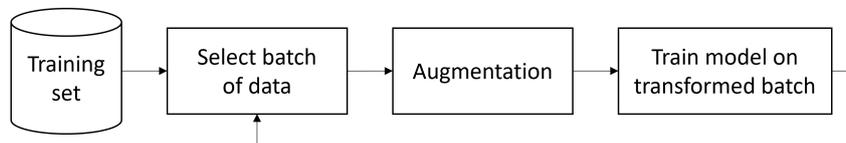
What has been not said so far is that noise learning is another possible cause of



**Figure 1.10:** Early stopping. After a certain point onwards, validation loss (red) increases over epochs, in contrast to training loss (blue). Next time, training is interrupted preventing overfitting. Image taken from [51].

overfitting: a network that explores noise in data could not be able to perform as hoped. With the aim of reducing it, decreasing the model complexity is mostly a good solution. In fact, it can happen that the architecture adopted is too complex for the designed task: in other words, it is overly depth and presents lots of parameters. In the context of ANNs, model reduction means eliminating layers and lowering the number of parameters as a result.

Another possibility is to add more data to the training set. It has been said more than once that networks require a huge amount of data to perform well: thus, the more inputs are given in training, the more models learn. Since collecting more training data is not a trivial question, it is often convenient to generate new samples starting from those available, coherently with the dataset and possibly keeping the same distribution of original data. This approach, also discussed in [52], takes the name of *data augmentation* and it is widely used in today’s applications: expanding datasets often helps networks to better generalize indeed. In the case of CV-oriented networks, thus CNNs, data consist of images and videos and the augmentation process gives new images and videos accordingly. They are obtained by randomly applying geometric transformations on original samples, like those introduced in 1.3.3, at each epoch [53]. In this respect, see the scheme of Figure 1.11. As a result,



**Figure 1.11:** Data augmentation pipeline. Image from Author.

the network sees always different samples, which are translated, rotated, scaled, sheared, or flipped as against the original, but true to the scenario depicted by the starting (poor) dataset. There are situations in which data are very few or it is even

impossible to get some for the desired task: to overcome this problem, synthetic samples can be generated from scratch, resulting in a full-fledged virtual dataset, with the highest possible conformity to the real scenario [52].

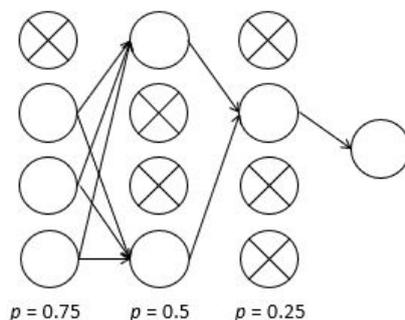
A different technique is *regularization*, that is adding an extra regularizer term to the cost function formulation, in this way:

$$J = \frac{1}{m} \sum_{i=1}^m \text{Loss}(y^{(i)}, \hat{y}^{(i)}) + \lambda \frac{\rho(\theta)}{m} \quad (1.14)$$

where  $\lambda$  is the regularization coefficient, generally picked as small, while  $\rho$ , the penalty term that is a function of the weight vector  $\theta$ , can assume one between two norm expressions depending on the type of regularization selected:

- $l_1$ -regularization:  $\sum_i |\theta_i|$
- $l_2$ -regularization:  $\sum_i \theta_i^2$

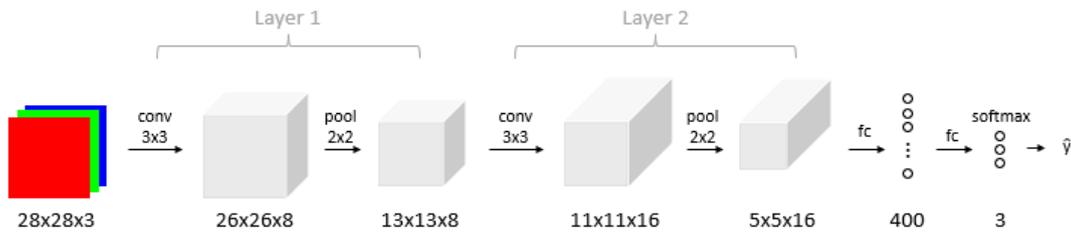
One last approach to counter overfitting is *dropout*, which serves as a regularizer too, but in a different way compared to those above. It works as follows: at each step of the training phase, some neurons are "turned off", in the sense that they are set to 0, with a rate  $r$  in relation to their layer. As shown in Figure 1.12 below, this operation truncates some links in the network architecture, not resulting in a fully-connected structure anymore. On the other hand, the uninvolved neurons are still active and continue to compute as in the dropout-free ANNs. The procedure selects neurons randomly every time enabling the network to use always different configurations (this is true for all the layers in which dropout has been enabled) and not becoming too dependent on some of them rather than others. While  $r$  refers to the rate of neurons shut down,  $p$  is used to indicate the probability of using a given node in a layer. The two quantities are complementary, of course. Typical sensible values for  $p$  are between 0.5 and 0.8.



**Figure 1.12:** Example of dropout regularization. Image from Author.

### 1.4.4 Convolutional Neural Networks

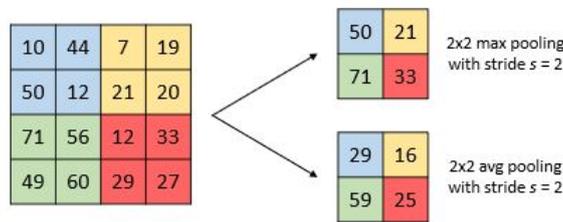
Convolutional Neural Networks [11] [12] are a family of networks largely used in CV today’s applications to handle and analyze images. They proved to be the best architecture in most artificial vision tasks thanks to their capability of exploring spatial and temporal correlations in data, extracting features, low-level or high-level ones depending on the depth considered, automatically. A small-scale representa-



**Figure 1.13:** Example of simple CNN architecture. The input, a 28x28 RGB image, is fed into two consecutive convolution-max pooling pairs, similarly to many modern CNN structures. Each of them forms one single layer. At the end, the volumetric information is flattened obtaining an MLP-like array of neurons. The final softmax layer provides the estimated one-dimensional output.

Image from Author.

tion of the typical CNN layout is reported above. Convolutional layers are generally made up of a pair including a convolutional filter followed by a pooling operation. The former performs a cross-correlation with the image coming from the previous layer, while the latter summarizes relevant pixel information reducing the resolution, that is, grouping pixels. Several pooling techniques exist, and those shown in Figure 1.14, which are max pooling and avg pooling, are surely the most employed for CNNs. Beyond the fact that cross-correlation-based filters used here are ambiguously called convolutional filters, it is interesting to note that activation functions are applied to the output of such filters, but not before adding bias to each pixel value.



**Figure 1.14:** Max pooling vs. avg pooling. In averaging, decimal values have been truncated to give an integer, but a ceiling approximation is allowed too.

Another relevant characteristic of CNNs is the frequent use of Batch Normalization (BN), whose use for CNNs was first proposed in [54]. It is a statistical agent that acts as a normalizer directly inside the network, introduced to deal with the distribution changes which affect different layers' inputs during training. Two parameters to be learned are included, namely  $\gamma$  and  $\beta$ , which represent scale and shift respectively, together with a constant positive quantity  $\epsilon$ , normally picked as small (e.g.,  $\epsilon = 0.001$ ). Having said this, its general procedure can be reported here:

---

**Algorithm 4** Batch Normalization

---

Let  $x \in \mathbb{R}^m$  a batch of  $m$  samples.

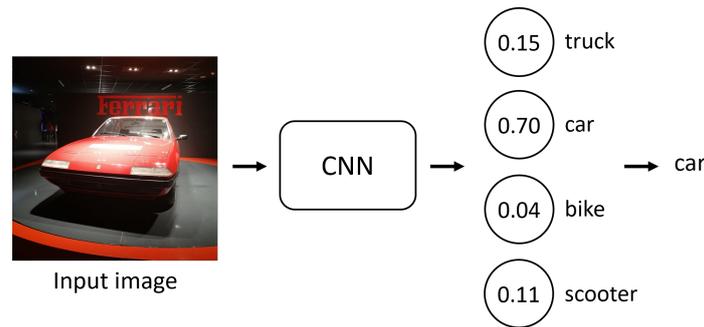
1. Compute mean  $\mu$  and variance  $\delta$  over the batch.
  2. Normalize as  $x'_i = \frac{x_i - \mu}{\sqrt{\delta^2 + \epsilon}}$ .
  3. Get the new samples as  $y_i = \gamma x'_i + \beta$ .
- 

It is important to highlight how BN acts differently based on the network mode: training or inference. In the first case, always new mean and variance values are used being calculated every time on the current batch, while in the second case  $\mu$  and  $\delta$  correspond with the cumulative mean and variance got when learning.

### CNNs for image classification

Classifying an image refers to the action of understanding what it depicts or, from the network perspective, finding a class label to assign among those of a predefined set [4], as in the example of Figure 1.15. In order to do that, the model involved must be able to extract low-level features (e.g., edges and borders) and high-level ones from what it sees. As previously mentioned, CNNs provide such a possibility thanks to their architecture based on convolutional layers. In general, predictions are included in a range of  $N$  possible classes, two or more depending on the classification kind, with  $\hat{y}$  going from 0 to  $N - 1$  in the case of label encoding, or being a zero vector with one single 1 entry at the index corresponding to the class number in the case of one-hot encoding, as previously reported in Table 1.2. This is what the network is intended to return as output.

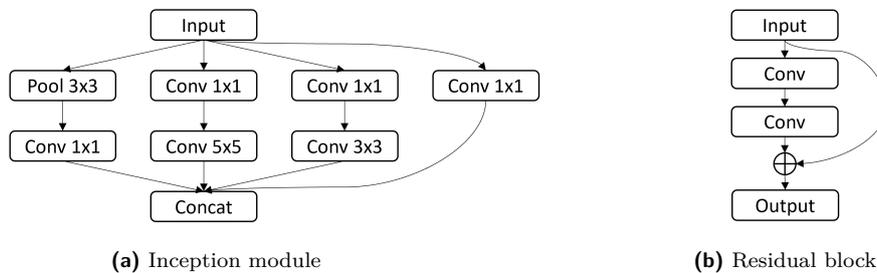
Among the most popular today's architectures for image classification, VGG [55], ResNet [56], and Inception [57], despite substantial differences in their structures, stood out for sharing the capability of providing surprising results on the challenging ImageNet dataset over time [11]. The former is probably the most simple, while the other two present innovative architectural ideas (see Figure 1.16), deviating from the more traditional CNNs like the VGG-inspired one of Figure 1.13. However, new families of advanced networks have been recently presented [58] [59], giving even better performance on such benchmark for classification, fueling the continuous research on CV themes. In this connection, state-of-the-art results on



**Figure 1.15:** Example of image classification using CNN. The final softmax layer has been extracted from the CNN block to highlight the probabilities associated with the four possible classes. The second one, labeled with 1, is the highest, thus the predicted class is "car". Image from Author.

ImageNet are available at this [link](#).

Always new adjustments are made in these architectures, giving birth to newer versions of the same original models over time. In particular, persistent efforts are made by researchers to reduce the complexity of modern neural networks, expressed as the total number of parameters, since it has a noticeable impact on the speed of execution. Lowering such a quantity while maintaining high performance can be arduous but it is crucial when it comes to using such a powerful tool for real-life artificial vision applications.



**Figure 1.16:** Two architectural innovations for CNN image classifiers. (a) allows deeper networks avoiding an excessive increase of parameters, while (b) exploits skip connections to address vanishing gradient and accuracy saturation problems.

Both Inception and ResNet have less parameters than VGG although they are much more deeper. Images from Author.

In Subsection 1.3.4, it has been said that filters are typically picked as squared with size 3x3, 5x5, and so on. Anyway, Figure 1.16a shows the presence of 1x1 convolutions inside the Inception module. Its introduction, as stated by authors, is due to the necessity of reducing the filtering depth (i.e., the number of channels) to speed up calculations since convolutions between deep volumes can be computationally

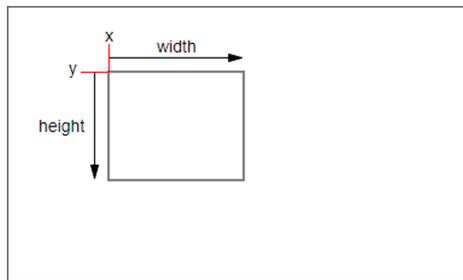
onerous. For instance, two consecutive convolutions with 256 filters for each are less expensive if a  $1 \times 1$  convolution is inserted in between, acting like a bottleneck. The way it works still complies with (1.4), except that the filter and accordingly the image patch covered both consist of one single integer. As a result, it comes down to multiplying each value in the image matrix by the single filter value, as below:

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix} \xrightarrow[1 \times 1 \text{ conv}]{\text{Filter} = [2]} \begin{bmatrix} 0 & 2 & 4 \\ 6 & 8 & 10 \\ 12 & 14 & 16 \end{bmatrix}$$

Finally, as far as performance evaluation is concerned, the most used metric for CNN-based image classifiers is accuracy, as is guessable. When large datasets are used, slightly different metrics can be adopted: for instance, Top- $N$  accuracy considers as correct all the predictions which are in the top  $N$  output probabilities given by the softmax layer.

### CNNs for object detection

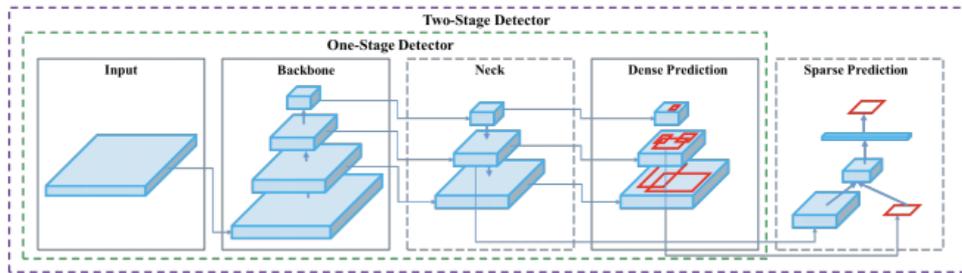
Detecting objects in images means not only identifying a reference class for each one of them, as in the image classification case, but also estimating their locations by means of *bounding boxes* [4], i.e. rectangles drawn around the objects found as accurately as possible. It is easy to understand that such a task is more difficult, in some respects, than the only classification. The idea is viewable here:



**Figure 1.17:** Abstraction of 2D bounding box concept in images. The reference system used is an  $xy$ -plane with  $y$ -axis inverted. Image adapted from [link].

Given an  $xy$  reference system as the one of Figure 1.17, there exists more than one notation to write bounding boxes and express them in terms of coordinates relative to the whole image:

- $xyxy$ , that is by specifying the minimum horizontal coordinate  $x_{min}$  and the minimum vertical coordinate  $y_{min}$ , together with the maximum horizontal one  $x_{max}$  and the maximum vertical one  $y_{max}$ ;
- $xywh$ , that is by specifying  $x_{min}$  and  $y_{min}$ , together with the width  $w$  and the height  $h$  of the rectangle;



**Figure 1.18:** Comparison between one-stage and two-stage object detection. Image adapted from [60].

- $ccwh$ , that is by specifying the center coordinates  $x_c$  and  $y_c$ , together with  $w$  and  $h$ . Sometimes, this notation is ambiguously referred to as  $xywh$ .

As can be guessed, it is quite trivial to convert one notation to another: for instance, the width and the height can be calculated as  $x_{max} - x_{min}$  and  $y_{max} - y_{min}$  respectively, the center pixel  $(x_c, y_c)$  can be obtained from  $w/2$  and  $h/2$ , and so on.

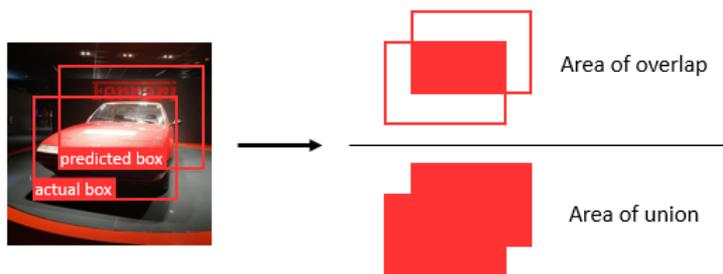
Object detectors are groupable into two macrocategories [60] [61]:

- One-stage detectors, like YOLO, first introduced in 2015 by Redmon et al. [62], which provide dense predictions in one shot;
- Two-stage detectors, i.e. networks based on region proposal, called R-CNNs [63] [64] [65], which provide sparse predictions by identifying some regions of interest before and classifying them later.

While the first ones are faster and more suitable for real-time applications, the others are sometimes better in terms of performance but suffer a (much) lower speed of execution. In this respect, state-of-the-art results on COCO over years are available at this [link](#). Such differentiation is reported in Figure 1.18, which also shows how the generic structure of an object detector stands apart from that of CNNs for image classification detailed before, resulting in a more complicated architecture separable into three main parts [60] [61]:

- **Backbone**: pretrained CNN deprived of its non-convolutional layers for feature extraction (one among VGG, ResNet, and others is usable);
- **Neck**: additional convolutional part to collect and concatenate intermediate feature maps at different resolutions;
- **Head**: prediction stage to retrieve classes and bounding boxes;

Nearly all the modern detectors use a hand-crafted algorithm as the last detection step to extract one single bounding box (out of many) per object: the so-called



**Figure 1.19:** IoU for object detection. The predicted box has been represented overly imprecise to emphasize the difference between the ground truth and the estimation, as well as between their relative areas. Image from Author.

Non-Maximum Suppression (NMS) [66]. It exploits the ratio between the actual box area and the predicted box area, which takes the name of Intersection over Union (IoU), to do that. The whole procedure works as follows:

---

**Algorithm 5** Non-Maximum Suppression

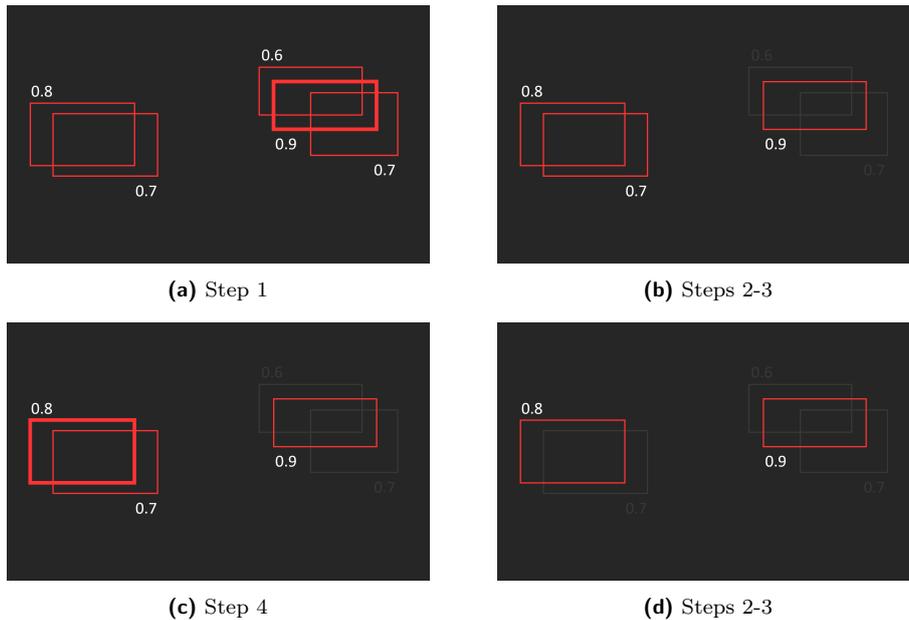
---

1. Select the box with highest confidence and keep it.
  2. Compute IoU of every other box with it.
  3. Remove all the boxes with  $\text{IoU} > \text{threshold}$ .
  4. Move to the next highest confidence and repeat 2-3.
- 

The algorithm stops when there are no more clusters of boxes to explore. It is still necessary today to use the hand-crafted NMS although the DL features given by the current models, and one of the reasons is exactly the fact that YOLO and similar detectors are not able to provide one single final prediction per object [67], as already mentioned above.

Another important difference compared to CNNs for image classification is in the output form provided by the network: since objects must be located as well as classified, a single numeric label no longer suffices. For each object in the input image, the relative output prediction is generally expressed as the combination of three pieces of information indeed: the reference class, which can be one-hot or label encoded; the bounding box coordinates in one of the formats presented before; finally, a new quantity called *confidence*, which is a percentage measure of the certainty associated with the predicted class (similar to softmax outputs). Therefore, a possible output formulation for one detection is  $\hat{y} = [\text{conf}, b, \text{cls}]$ , where *conf* is the confidence, *b* is the quartet vector representing the bounding box, and *cls* is the class predicted (one-hot vector or single numeric label). In case of multiple objects detected,  $\hat{y}$  is a tuple of vectors, one per object.

In conclusion, since object detection is a complex task, a more sophisticated set of metrics needs to be used, as anticipated in 1.4.2. Firstly, precision and recall

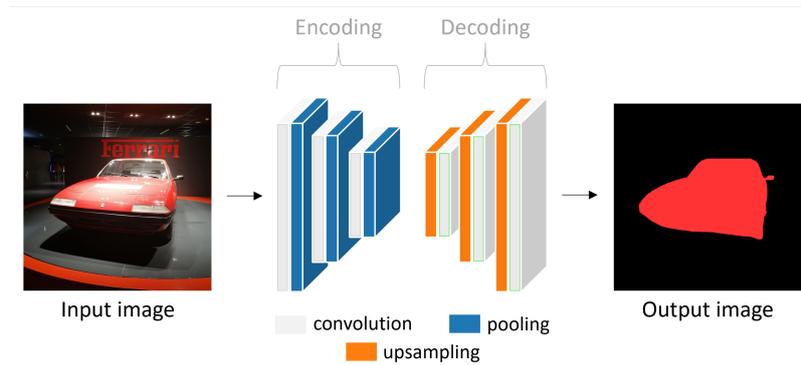


**Figure 1.20:** NMS application. The box with the highest confidence is picked (a), then all the others having  $\text{IoU} > 0.5$  are removed (b). Similarly, the box with the next highest confidence is selected (c) and the remainder having  $\text{IoU} > 0.5$  are discarded (d). The final predicted boxes are visible in (d). Image from Author.

are still adoptable but with a different interpretation: given a class, true positives are detected objects and false negatives are missed objects, while false positives are something detected as objects of such class but which are not in the scene in reality. Furthermore, a metric called mean Average Precision (mAP) is always used, as specified below:

$$\text{mAP} = \frac{\sum_{i=0}^{N-1} \text{AP}_i}{N} \quad (1.15)$$

where  $\text{AP}_i$  is the Average Precision (AP) associated with the  $i$ -th class on a set of  $N$  classes going from 0 to  $N - 1$ . AP is a good approximation of the area under the precision-recall curve. As already mentioned before, it can be computed as the sum of consecutive trapezoidal regions. The notation  $\text{mAP}^{\text{IoU}}$  indicates mAP calculated at a certain IoU threshold. Hence, beyond precision and recall, two typical values employed for evaluating an object detector are  $\text{mAP}^{0.5}$  and  $\text{mAP}^{0.5:0.95}$ , where the first one is mAP corresponding to an IoU threshold set to 0.5 and the second one means average mAP over various IoU thresholds from 0.5 to 0.95 with step 0.05. Clearly, the higher mAPs are, the more precise an object detector is.



**Figure 1.21:** Example of image segmentation using CNN. Here, semantic segmentation is performed and output pixels are differentiated into "car" and "background". CNNs used for this task generally consist of an encoding part, that downsamples the image, and a decoding part, that upsamples the reduced information to lead back to the input form. Image from Author.

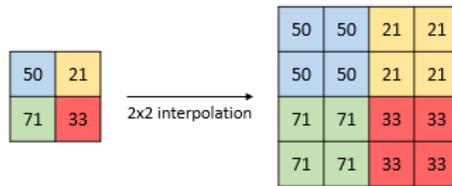
### CNNs for other tasks

This last part devoted to CNNs begins with a brief overview of those used for semantic and instance segmentation [4], two similar but different tasks. Semantic segmentation consists of grouping image pixels based on what they represent or, in other words, classifying each pixel as belonging to one class rather than another (see Figure 1.21). One typical approach to do that is to rebuild the volumetric information after a certain number of consecutive convolutions, with an opposite decoding path including a series of *upsampling* blocks, considerable as the complementary of convolution-pooling sequences since they aim to gradually reincrease the image size up to the original. Nearest Neighbor interpolation is one of the most simple and used techniques to do that (see Figure 1.22). A segmentation-oriented network is often referred to as Fully Convolutional Network (FCN), because of the absence of other layers besides convolutional ones resulting in an encoding-decoding structure. One state-of-the-art architecture for this task is SegNet [68].

Given an input square volume of size  $n_{in}$ , it is possible to know in advance the size  $n_{out}$  of the upsampled output, similarly (1.5) to seen before:

$$n_{out} = (n_{in} - 1)s - 2p + f \quad (1.16)$$

Instance segmentation is more specific: beyond labeling pixels, networks designed for instance segmentation are also able to distinguish the different instances of the same class [4], thus count the number of equal objects in images. Such complex task is usually performed through R-CNN equipped with segmentation masks, like the Mask R-CNN model does [69].



**Figure 1.22:** Example of image upsampling through NN interpolation.  
Image from Author.

There also exist 3D-CNNs for human action recognition [70], which differ from the architectures discussed so far in that they use 3D convolutions to perform tasks directly on videos, not images. A video can be seen a sequence of images, i.e. frames, thus the number of samples composing it is an extra dimension to add to the input. 3D convolutions act as 2D convolutions but with one more dimension which singularly takes into account all the frames.

### 1.4.5 Transfer Learning

Transfer Learning [71] refers to the action of using the knowledge acquired on a dataset on another one, thus performing a similar yet different task. Being  $T_1$  the original task and  $T_2$  the new one, such operation has meaning when:

- $T_1$  and  $T_2$  require inputs of the same type (e.g., images);
- Data available for  $T_2$  are far fewer than those available for  $T_1$ ;
- $T_2$  can exploit low-level features of  $T_1$ ;

For instance, a YOLO model pretrained on COCO could be used for a new custom task that consists of recognizing only cars based on a new reduced dataset of few images depicting only cars. This approach can be successful since COCO, that has more than 300,000 samples, represents 80 classes including "car" already.

After this brief introduction, an explanation of how transfer learning can be performed is given here. For a start, the classical transfer strategy used in most applications lies in replacing the non-convolutional layers at the end of the network and then freezing all or part of the preceding convolutional backbone. Freezing a layer implies not to update its weights while others are trained: thus, the weights from the original task  $T_1$  are used and kept unchanged until the whole training process finishes. The results obtained by the only new added layers in the trainable mode are generally good when  $T_2$  data are similar to those of  $T_1$ . If the two datasets differ significantly, a further improvement is usually needed to have decent performance: the whole architecture is unfreezed and a second training phase is started with a very low learning rate. This can help the network to slowly adapt

the features previously learned to the samples of the new dataset. Such procedure of unfreezing and retraining is known as *fine tuning*, so named because a proper refinement is done to adjust the weights and make the model conform to  $T_2$  as much as possible, and differs from the first stage generally referred to as *hot start*.

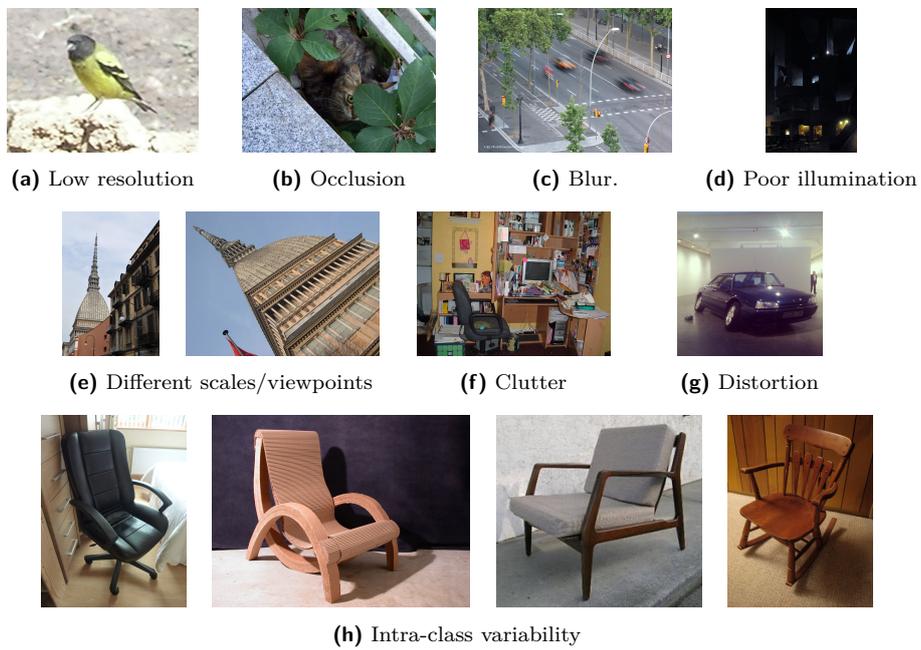
## 1.5 Critical issues

As said at the beginning, it is not easy to make machines able to understand the scene: in addition to the difficulties in emulating human vision in terms of functioning, some trouble may occur in handling the visual information. In this respect, Figure 1.23 shows a collection of critical conditions which make CV hard, also reported in [1] and listed below:

- (a) loss of details due to camera limitations (e.g., low resolution);
- (b) partial or total occlusion of objects of interest;
- (c) blur given by abrupt camera movements or objects changing position in successive video frames;
- (d) significant brightness variations or poor illumination conditions;
- (e) same object reproduced on diverse scales or seen from several viewpoints;
- (f) clutter, that is the simultaneous occurrence of many different objects (some are to be identified, some are not);
- (g) distortion phenomena;
- (h) multiple instances of an object in classification problems where a class of belonging (e.g., "chair") must be properly identified.

A human generally handles all such situations; from a computer perspective, they often represent deceptive cases instead.

With that said, CV has nonetheless provided surprisingly good results in many real-life applications, becoming a burgeoning field of study and research and even sometimes matching or outperforming human skills [11].



**Figure 1.23:** Main critical issues in CV. Images from Openverse [\[link\]](#).

# Chapter 2

## Project overview

In this section, the two companies involved in the project are briefly presented: the consulting firm and the customer requiring a service. Then, a generic overview of the entire work is given, lingering on its purpose and adding a note about the ensuing challenges. In closing, one of the most meaningful parts of this work is reported: it is a discussion about the state of the art on the use of neural networks and artificial vision algorithms for object and color detection applied in real-life industrial automation contexts.

### 2.1 Companies presentation

The former is Orbyta Tech, which provides professional consultancy and solutions in multiple information and communication technology fields such as software development and system support. In particular, the unit called Data Science & Analytics focuses on integrated solutions which use methodologies and technologies concerning multiple areas: Artificial Intelligence, Business Intelligence, and Data Engineering. For more information, visit [here](#).

The latter is DMC Automation, which focuses on providing engineering solutions in the fields of Automation and Robotics instead. In this respect, one of the main businesses of the company is the production of special machines for construction assembly testing of automotive seats. For more information, visit [here](#).

### 2.2 Tasks and goals

This work explores the possibility of using some of the algorithms and tools presented in Chapter 1, putting them all together in a single application, as a proof of concept for an automatic a posteriori check on the assembly of automotive seat frame parts, specifically colored motors, in an industrial environment where a bench lets the frames translate back and forth while a camera captures the scene from

above. More precisely, there is a steel support base on the bench controllable by a hydraulic-mechanical device through a red lever in its upper-left corner so that, once it is activated, it starts moving, dragging the seat frame mounted above.

The one put at disposal is a frame made up of multiple components, some big and some small, but the focus of this project is only on its three motors, placed at different parts but close to each other. Compared to other frame elements, such motors are medium-sized and, perhaps more importantly, have a defined and recognizable shape, suggesting that an AI system could be employed for their identification and localization.

As mentioned before, motors are colored. This can serve to differentiate eventual multiple seat frames on the basis of their belonging: for instance, frames destined to Car A could present motors marked with Color A, those destined to Car B could present motors marked with Color B, and so on. When the check is done, one can be directed to a line rather than another for further operations if everything is fine. In a simpler scenery, the automatic check tool can be employed directly as the final step of the entire assembly procedure. In any case, the whole can be supervised by a human operator who constantly monitors the real-time information given by the application on a dashboard implemented specifically for the purpose. It is noted that the check is done by seeing consecutive frames of the camera stream, not single photos. Accordingly, such information can include statistical results on motors, colors, but also average processing speed, and other stuff, with the eventual print of error messages on the screen when something anomalous is detected.

For the present study, three colors have been provided: red, yellow, and white. In this respect, 16 video footages showing the frame translating on the base have been retrieved, including the most diverse situations: not yet colored motors, some colored and some not, some colored in one color and some in another, and all marked with the same color. However, the light conditions of the environment are not ideal, since a glass ceiling lets sun rays enter with the risk of distorting the perception of colors. The methods designed to correctly identify them have taken into account this issue, as far as possible.

In summary, the main activity of this work has consisted of developing a tool capable of recognizing motors, with the possibility of counting them, together with their color, in real time on video frames.

## 2.3 State of the art

As is now clear, identifying the color, shape, or location of work pieces may ease typical automated operations such as pick and place, assembly monitoring, defect detection, and others in the context of industrial automation processes. In this connection, the concept of Industry 4.0 has become increasingly popular in the last years thanks to the enhancement of powerful and efficient cutting-edge technologies such as Internet, cloud computing, robots, and intelligent devices [72] [73] [74].

Many believe that a new era of smart factories, composed of several interconnected parts constantly fed by data, has already begun giving way to another industrial revolution to all effects. The ever-growing worldwide data exchange and the continuous improvement of software and hardware technologies blend well with the rise of Deep Learning that started a decade ago indeed. One should try to imagine the factory of the future as a complex mechanism of intelligent artificial entities that interoperate to support humans in complicated tasks, minimizing error probabilities, or even work autonomously to ensure efficient production.

Everything said so far is normally referable to the common tasks of object and color detection in the manufacturing field, and recent studies have shown how artificial vision tools can be employed to achieve good performance in them. More precisely, the possible use of modern detectors as an active part of industrial processes has been largely investigated. Among the various applications, informative automotive engine compartment label recognition has been studied by Ferreira, Barroso, and Filipe [75]. They compared three detectors, namely two YOLO variants and Faster R-CNN, with the aim of finding one able to assist human operators in the conformity check of such labels. This can help to avoid a fully manual assessment saving time, fatigue, and stress, thereby reducing errors. A small set of real photos, annotated by hand, has been used as the dataset for training, with data augmentation enabled. Surprising results have been reached in terms of mAP, precision, and recall with all the three models, but Faster R-CNN has turned out to be rather slower than the others, thus not suitable for real time.

In addition to this, many other YOLO-based solutions to typical manufacturing tasks have been proposed over the past few years [76] [77] [78].

Remaining in the automotive sector, Rio-Torto et al. have examined the possibility of identifying several car parts (doors, bars, lights, windows, etc.) based on only labeled simulated data before [79] and a mix of labeled simulated and pseudo-labeled real samples later [80]. In both cases, the goal has been to secure high-quality inspection replacing manual checks with human-machine interaction, in line with what has been discussed above. The second work has also included the implementation of a graphical interface to give operators an easy infographic to confirm or deny the network's predictions. As said, in the first experimental study, only synthetic images have been employed, both for training and testing, all of them generated and automatically annotated through segmentation masks in Unity. The encouraging results have compelled authors to expand their study including screenshots taken from real videos in the dataset, thus performing transfer learning on the model pre-trained on synthetic data. Good detection results have been provided by the ResNet-like-backed network adopted.

A different but equally interesting approach is the video-oriented one proposed by Chen et al. to find eventual mistakes in manual assembly actions by recognizing them in video streams [81]. A 3D-CNN model boosted with Batch Normalization is used to retrieve spatio-temporal information about the input streams, then a

VGG-like FCN segments it to separate background pixels from those relative to work pieces. For training, the former has been fed with annotated video frames, the latter with labeled segmented images. It has been found that BN has led to faster accuracy convergence, as well as a more stable loss, compared to the BN-free architecture. Performance has been great for the FCN, less for the 3D-CNN.

Another important aspect in industries, actually related to the works just discussed, is the necessity of supplying products free of impurities, that is, surface defects. In this regard, quality checks have always been done manually until today, but research is pushing to adopt intelligent tools for automatic defect detection. Sun et al. have investigated the use of a modified Faster R-CNN model as the possible solution to the wheel hub assembly check task [82], aiming at improving the accuracy of such operation. Scratches, oil stains, holes, and griming are just a few of all the surface defects which could compromise production in that scenario. A set of real images manually annotated has been used for training; however, unsatisfactory results have been obtained. Authors state that poor illumination and intra-class variability could be identified as two possible causes. A lighting improvement, namely multi-angle lighting, is suggested for future developments. It is still said that the network adopted has been faster than other detectors available at the time.

One more work relevant to this sector is the one by Huang, Wei, and Yao [83], which have used Mask R-CNN combined with SVM to identify anomalies in a set of four mechanical parts: flywheels, shafts, bearings, and sleeves. Few real images have been collected and manually annotated with instance segmentation masks. A consistent use of data augmentation transformations have been done to deal with the limited amount of training data. Segmentation results, including area and perimeter for example, are fed into the subsequent SVM module that is responsible of classifying the impurities with a one-vs-one approach. Excellent performance has been achieved in both segmentation and classification tasks, together with high speed. As future developments, it is suggested to explore more challenging environmental conditions.

All the solutions seen so far have in common the adoption of an object detector as the model to perform recognition via the usual bounding boxes. However, a totally different approach based on image classification has been also presented [84]. Such work aimed at using a new enhanced VGG version to classify images presenting generic industrial objects or specific surface defects of mechanical components. The original architecture of 19-layer VGG has been transformed into a multipath one by branching off intermediate feature maps from predefined convolutional stages and concatenating them with the final feature vector to have a richer quantity of information in the output stage for softmax classification. The proposed modified model has overcome the original one for each class in terms of accuracy.

With regard to color identification, the possibility of recognizing objects based on their color through color models had already been theorized in the past [85].

Recent approaches have been aimed at differentiating a set of objects of the same type (e.g., all cars, all motors, or whatever) but different colored. In particular, the task of recognizing generic vehicles labeled with their color (green, red, white, etc.) has been studied by Rachmadi and Purnama [86], outperforming the previous solution provided by Chen, Bai, and Liu [87] in terms of accuracy but appearing to be slower. It has been found, anyway, that different color models than RGB are preferable for such problem. One suggestion is HSV. The same has been done in the work by Malburg et al., already mentioned before, where workpieces have been marked with a different label based on their color in a simulated industrial environment [77]. It is interesting to note how all these studies tried to change, in some way, the nature of CNNs, which usually detect meaningful shape features such as edges and lines in images, not necessarily colors. Going back a few years, when the outbreak of the first neural network technologies occurred, an MLP-based method has been explored to identify colors independently as tuples of values, not in relation to objects [88].

In closing, still referring to color-based applications, other feasible techniques are centered on machines extended with color constancy, which is the human skill of preserving a constant perception of colors in varying lighting conditions [89]. The aim is to trace back non-ideal images to a predetermined baseline illumination. Beyond the statistical approaches to this problem [90], many researchers have proposed solutions based on artificial vision over years, some of them suggesting the use of semantic information, like segmentation masks, as the input of the subsequent color constancy module [91], others favouring the direct feed-in of CNN-derived low-level features [92] [93]. An unsupervised method has also been designed with the goal of avoiding the time consuming preliminary stage of image calibration [94], usually required for this task.

# Chapter 3

## Data collection

As already said, a huge amount of data is required to achieve good results when it comes to training ANNs even though, in many real-life situations, it could be difficult to collect a consistent number of images and videos. In order to ensure optimal learning, several samples at different conditions of framing, positioning, coloring, and illumination should have been provided to the model selected. In this work, a sufficiently large dataset of synthetic images with YOLOv5-formatted labels has been created starting from a 3D model of the seat frame, kindly provided by DMC Automation, to overcome this problem. In this regard, the next section gives an overview of the software used for that, while the other two lay out a detailed description of how the synthetic images have been extracted from simulated videos in a 3D virtual environment where the 3D frame model has been placed.

### 3.1 Unity overview

The software used to generate data has been Unity [95], precisely in its 2020.3 version. It is a game engine widely adopted for 2D and 3D game development, quite popular among newbies, which relies on C# 8.0 (no paper, visit [here](#)), a Microsoft-developed compiled object-oriented language able to interact with Unity and simplify the activity of programmers thanks to ad-hoc classes and methods. All the code is editable in Visual Studio 2019 (no paper, visit [here](#)), a Microsoft IDE for the most diverse software products including computer programs, web services, and mobile apps. It is integrated with Unity, and offers tools to deploy, organize, and test scripts such as a C# compiler.

Now, it is necessary to introduce some fundamental notions about Unity objects and how they are treated in the relative C#-based API called UnityEngine, considering that all the special terms that will be nominated are explained and accompanied by examples in the Unity scripting reference at this [link](#). Thus, please see it to have a clearer insight into this part. A further point is that objects and

Name	Description
Color	Color defined by its RGBA channels.
Rect	Rectangle defined by the upper-left corner $(x_{min}, y_{min})$ , the width $w$ , and the height $h$ .
Vector2	2D point expressed as $x$ - $y$ pair.
Vector3	3D point expressed as $x$ - $y$ - $z$ triplet.

**Table 3.1:** Some C# structs from the System namespace.

game objects are slightly different concepts in Unity, as indicated by their relative classes, but both terms will indistinctly refer to game objects from now on.

With that said, a game object is, in general, something that can be placed in the scene and afterward involved in the game application. Some operations involving it are translation, rotation, hiding, and color change, but many others exist. In Unity, there is a set of basic geometric 2D and 3D shapes consisting of plane, cube, sphere, and others joined by environmental objects such as lights and cameras. In addition, custom 3D models can be imported in the scene and employed as game objects too.<sup>1</sup> Every model, regardless of how it has been realized, is the sum of a certain number of polygons, precisely triangles, put together through a mesh. Moreover, those representing real-life complex objects, like seat frames, are generally made up of several children objects, a sort of subparts hierarchically under the main one which is their parent. Each of them, being to all effects a 3D object, is in turn composed of triangles. In Unity, the maximum number of polygons allowed is 256. Over-threshold objects are still usable, but a partial hull is employed for collisions instead of their full convex one in such case. Naturally, the more triangles compose an object, the more computational effort is required to perform any kind of operation on it. For instance, looping over vertices could be very CPU intensive.

At this point, one can get a rough idea of how Unity works, although limited to the artificial vision field. Accordingly, it should be noted that only few of its functionalities have been explored for the purpose of this project, when in fact it offers much more. Do not forget that Unity is a game engine, certainly not a data generator, but some of its features like the greater user-friendliness compared to other engines, the programming-oriented approach, and the availability of effective graphics tools make it attractive to ML and CV developers to create performing synthetic datasets, which can even bring benefits if compared to classical ones based on real data [96]. Dealing with synthetic data might sometimes be more convenient than collecting samples from the real world: in fact, assuming that some real data are available in large quantity, the steps necessary to build a dataset of this kind often require human intervention, one above all the manual annotation

---

<sup>1</sup>Unity supports different file formats for 3D import: .fbx, .dae, .3ds, .dxf, and .obj.

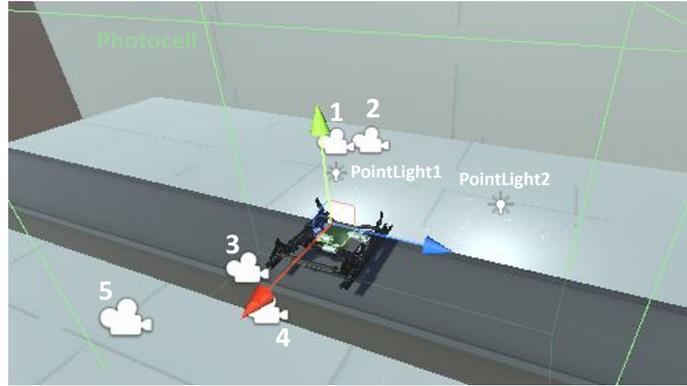
of images/videos that can be very tiring and time-consuming. Moreover, such data are usually raw and messy: cleanup operations are thus needed to organize them in a suitable form for training. Then, if a model trained on a real dataset does not perform as hoped, more data must have been collected, and this may be difficult. Datasets based on virtual samples, on the other hand, are typically generated with parameters randomly set, which can be easily adjusted in the case of poor performance. Another advantage is that annotations can be handled automatically through ad-hoc algorithms and tools, as done in this work indeed. A downside is that it may not be as easy to recreate a true-to-life simulated scenario, besides that complex procedures often entail a high computational cost, as already mentioned in relation to the number of polygons.

## 3.2 Virtual environment setup

The 3D seat frame model has been imported into the Unity software and here used as a support for the generation of synthetic images. Anyway, before getting to the heart of the procedure, a description of how the whole virtual environment has been designed and set is needed. The idea was to recreate a simple but realistic industrial scenario, specifically an assembly line so that multiple consecutive frames can enter and exit from the scene continuously. The first trivial step has therefore been to place two orthogonal planes to form the floor and one wall, both colored gray. A cubic object shaped like a long parallelepiped has been used as a sort of conveyor belt so that frames on it can be transported. Such functionality has been obtained by attaching a C# script, called `ConveyorBelt.cs`, that detects collisions with frames and accordingly sets their velocity, different from zero, together with the direction of motion. Collisions are managed with the two functions `OnCollisionEnter` and `OnCollisionExit` so that seat frames can be tracked in real time. Some lights have then been added to the scene: a directional one to simulate overhead illumination and two point ones, moving back and forth across the belt, to make motors subject to variations as much as possible. This step is necessary to provide the network variegated images when it comes to training. Light modifications, such as translations, rotations, etc., are possible thanks to the built-in animator tool available.

A script called `SetEnvironment.cs` has been attached to an empty object and employed as the source for the entire environment initialization. The 3D model, in the form of a prefab, has been put in a public variable so that several copies of it can be generated with the `Instantiate` method. It is also responsible of constantly checking, at each video frame, whether the number of images generated has reached the quantity prefixed or not yet. The three motors are children objects of the whole frame, and they are been randomly colored through the `Color` struct fields. The network should be learn to detect motor shapes independently from their color.

Five cameras have been placed in the scene to have a comprehensive view of the belt with the moving frames on it. Their positions have been decided so that

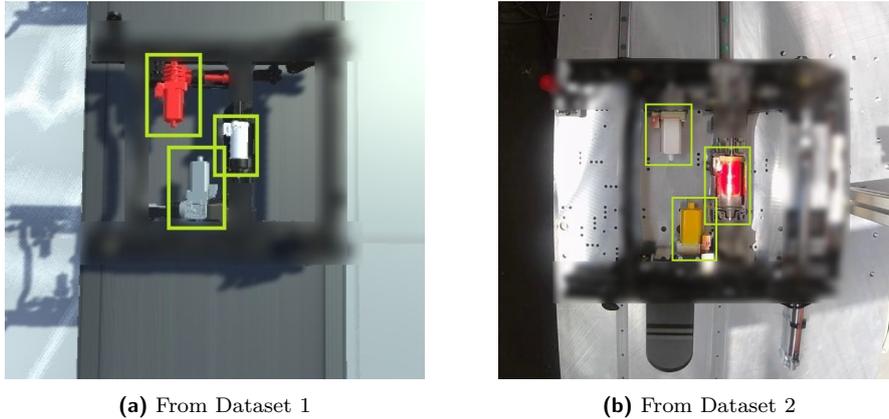


**Figure 3.1:** Unity simulated environment for synthetic image generation.

each of them can provide screenshots from distinct viewpoints, on par with the classic guidelines to train neural networks. To have a view, a Unity camera exploits a *render texture*, a texture that varies at runtime. Textures are 2D or 3D surfaces used in computer graphics to give elements a realistic "feelable" aspect. In Unity, each camera can employ its own render texture as a canvas on which to paint the scene at a given moment. It is quite easy to understand that such functionality enables the possibility of saving video frames as screenshots, as wanted for the purpose of this work.

When the `targetTexture` attribute of a camera object is set to `None`, it is intended that rendering will be available directly on the computer screen in game mode. A custom one must be provided to make such camera capable of taking screenshots: in this project, one with size 560x315 and RGBA color format has been adopted.

For closing, a wide and high box called `hasBeenPut` has been put in the scene to encapsulate a section of the conveyor belt and detecting the passage of objects inside, acting like a photocell. Its mesh renderer component has been deactivated to make it invisible, with the attribute `isTrigger` turned on in order to allow collision detection with subsequent triggered actions. Collisions are managed in the `ManageTrigger.cs` script, which contains the essential code to effectively generate images and annotate them, as will be clearer in the next section. Beyond the mandatory functions to handle collisions such as `OnTriggerEnter` and `OnTriggerExit`, it includes a series of custom methods thought to randomize the whole process as much as possible so that several trials can be done by adjusting the parameters every time, that is, by simply changing some variables in the code, in accordance to what it is said in [96]. For example, `CamCapture` is for screenshots, while `Get2DBoundingBox` is for automatic annotations of motors. Anyway, their explanation is postponed to the next part because of their purpose. To have a real-time generation of samples, simultaneously depending on the video, the two functions mentioned, with more stuff, have been enclosed in the built-in Unity one for real-time updating, namely `Update`, called every frame making everything inside in synchrony with the video.



**Figure 3.2:** Example of real and synthetic samples from the relative datasets. The seat frame is blurred in both images for copyright reasons.

### 3.3 Images and annotations

By pressing play, the whole environment is started, with all the scripts being executed: in this way, an evenly spaced set of chassis is generated and the conveyor belt allows their transport at a given constant speed. Every time one of them hits the entering surface of the transparent box, it is added to a list of colliders.<sup>2</sup> At this point, the considered object is more or less in the field of view of all the cameras, so that each one can take a screenshot of what it sees, with the subsequent call to the annotation function and the resulting bounding box to be written to a text file according to the YOLOv5 labeling rules previously discussed. Clearly, at the exit, the object is removed from the list, while those behind are still inside the box or entering it. This emphasizes the dynamic behavior of the list, the size of which can vary in time depending on the location of the frames.

It should be clear now that the main core of this automatic procedure for generating and annotating images is the acquisition of the image and the corresponding computation of one or more bounding boxes inside it. As far as the first step is concerned, each camera exploits a *texture* for visual information rendering: in particular, a blank one is provided to each of them when the environment is initialized and this acts as the digital surface where all the pixels of the screen will be written. The texture is then converted to a byte array which represents the corresponding PNG file so that the final image can be stored somewhere on disk.

At the same time, the script responsible for generation and labeling can get the

---

<sup>2</sup>C# lists are dynamic and flexible data structures with unknown size at compile-time and variable at runtime. Colliders can be added and removed from their relative list making the trigger box act like a photocell.

bounding boxes using a simple but effective algorithm like this:

1. The upper-left corner  $(x_{min}, y_{min})$  and the lower-right corner  $(x_{max}, y_{max})$  of the bounding box to be determined are initialized to  $(0, 0)$  and  $(MAX, MAX)$  respectively, being MAX the maximum float number storable to memory;
2. All the vertices of the object lying in the photocell are read and stored in a Vector3;
3. A for loop is run over all such vertices so that at each iteration a 3D one can be converted into the corresponding 2D screen point in the form of a Vector2, then compared to the current  $(x_{min}, y_{min})$  and  $(x_{max}, y_{max})$  to eventually update them if a new minimum or a new maximum is found;
4. At the end of the loop, a new Rect is created based on the final  $(x_{min}, y_{min})$  and  $(x_{max}, y_{max})$  points, hence the four box values required by YOLOv5, which are  $x_c$ ,  $y_c$ ,  $w$ , and  $h$ , can be retrieved and normalized to be in  $(0, 1)$ , hence ready to be stored in text form to annotation file.

The simultaneous computation of multiple boxes in case of more motors in the same image results in multiple entries in the associated text file, complying with the general conventions for object detection data labeling. Finally, it is noted that a small offset has then been added to have not-too-tight boxes, which might have been too challenging for the network. With this method, 3610 images have been generated, split as 80%-10%-10%, corresponding to 2888 training samples, 361 validation samples, and 361 test samples.

Furthermore, at a later time in the development, there has been the possibility of extracting a few real images as screenshots from 16 real video footages, which nevertheless represent just a fraction of all the feasible environment configurations: it is enough to think that the camera is always placed in the same position, namely on top. One video has been discarded and further used as a case study for inference. 205 real images have been collected, split as 70%-15%-15%, resulting in 143 training samples, 31 validation samples, and 31 test samples, all of them manually annotated in the YOLOv5 format with Roboflow Annotate (no paper available, visit [here](#)), a free online tool which allows users to draw boxes around objects and download the generated labeled images with annotations in the desired format.

	Data type	Total size	#Training samples	#Validation samples	#Test samples
1	Synthetic	3610	2888 (80%)	361 (10%)	361 (10%)
2	Real	205	143 (70%)	31 (15%)	31 (15%)

**Table 3.2:** Summary of the two datasets used in this work. Transfer Learning has been performed from 1 to 2, as will be seen later.

## Chapter 4

# Materials and methods for object recognition

### 4.1 Python and libraries

Most of the code of this project has been written in Python 3 [97], an interpreted object-oriented language largely employed in today's ML applications and, here, specifically used for color detection and YOLO customization; among its numerous libraries, the most relevant to object and color detection have been:

- NumPy [98], an open source library meant for scientific and engineering computing with functions for handling multidimensional arrays;
- Matplotlib [99], a low-level cross-platform library for 2D and 3D plots;
- scikit-learn [100], a robust library for Machine Learning and Data Analytics that provides a collection of supervised and unsupervised algorithms;
- Pillow (no paper, visit [here](#)), a user-friendly Python Imaging Library fork (now deprecated) for basic image processing operations;
- OpenCV [101], an open source cross-platform library, originally developed by Intel, for the most diverse artificial vision applications, including support for image processing and related transformations too;
- PyTorch [102], a Torch-based open source framework, originally developed by Meta AI, which offers the possibility of building, training, and using deep neural network models for artificial vision or language processing;

- Pandas [103], a powerful open source data analysis tool to manipulate data in the form of dataframes,<sup>1</sup>

## 4.2 YOLOv5 overview

YOLOv5, in its last February release [104], has been the adopted CNN model to detect motors, in a way that will be clearer later. It has been chosen for its speed and user-friendliness, besides the fact that its code has been developed on the PyTorch framework, one of the most popular today in the ML community, to the extent that, according to some, it is shaping the future of artificial vision.

At this point, an outline of YOLOv5 is given, with details about its structure, model variants, functioning principles, and more. The theory of CNNs for object detection and, more in general, the basic notions of ML and ANNs (everything discussed in 1.4.3) come in handy to better understand the following presentation.

### 4.2.1 Architecture

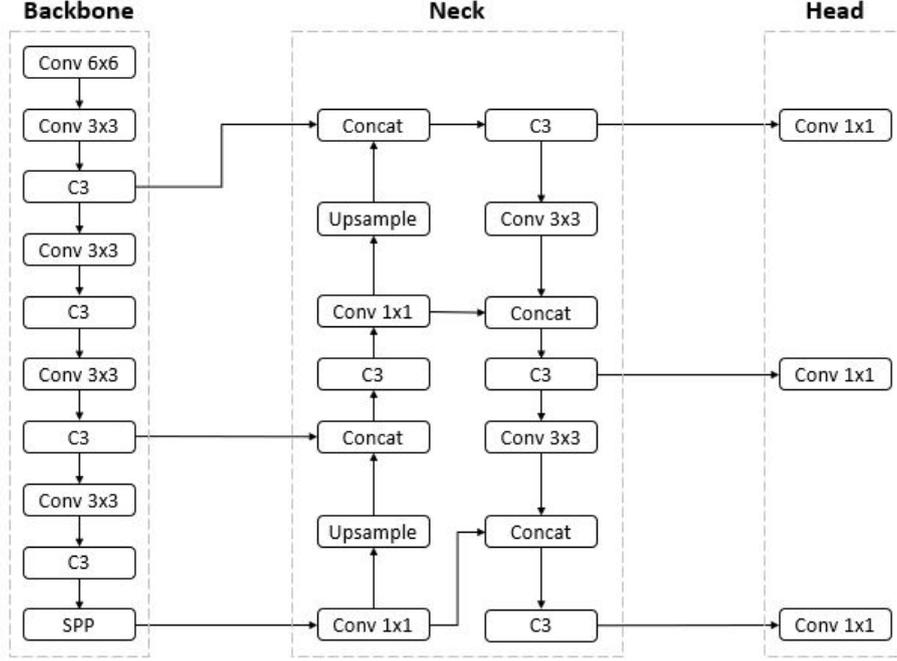
The YOLOv5 architecture (see Figure 4.1) can be organized into three major parts, each of them containing modules of different nature, on par with all the modern detectors. The list below specifies how they have been realized, that is, which architectural elements have been selected for each part:

- Backbone: an alternate series of convolutional and C3 blocks, these last inspired by those introduced in CPS-Darknet53 [105], together with a final SPP block, derived from SPP-Net [106], in a modified faster version;
- Neck: a PANet-like structure [107] that also includes C3 blocks;
- Head: three prediction layers at different resolution, as done for the FPN output stage [108].

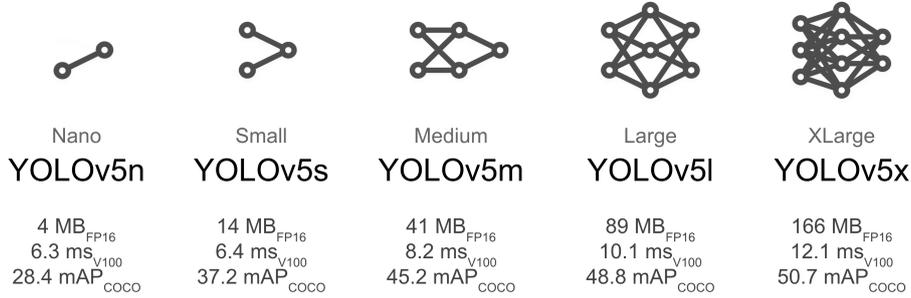
Still referring to the architecture, the idea of network scaling [58] [109] has been exploited for creating a YOLOv5 family with the possibility of choosing one among five possible models over increasing complexity: YOLOv5n, YOLOv5s, YOLOv5m, YOLOv5l, and YOLOv5x, which stand for nano, small, medium, large, and extra-large respectively. In this connection, Figure 4.2 shows basic specifications for each of them: storage size, speed, and state-of-the-art performance on COCO. As is guessable, smaller models are lighter and faster at the cost of mAP as compared to larger ones. Furthermore, Table 4.1 reports the scaling multiples  $s_{depth}$  and  $s_{width}$  that are required to obtain each of the aforementioned models, together with the

---

<sup>1</sup>A Pandas dataframe is a 2D table-like data structure, with data arranged in rows. Its columns can potentially contain data of different nature.



**Figure 4.1:** YOLOv5 baseline architecture, derived from the `yolov5l.yaml` model file. Image from Author.



**Figure 4.2:** YOLOv5 models of different size. Source: [\[link\]](#).

complexity meant as the total number of network parameters (expressed in millions).  $s_{depth}$  and  $s_{width}$  can be adjusted in order to increase or decrease both the network depth and the relative width. Since reducing the depth means to lower the number of layers, fewer convolutional blocks appear in the new downscaled architecture. Complementarily, a deeper structure will have more. The width refers to the number of channels of convolution outputs instead, thus, how many filters are employed. Narrower models use fewer filters, wider models use more. This has an impact on the total number of feature maps, so on how much information the network is able to capture about input images.

Model	$s_{depth}$	$s_{width}$	#Params
Nano	0.33	0.25	1.9M
Small	0.33	0.50	7.2M
Medium	0.67	0.75	21.2M
Large	1.00	1.00	46.5M
XLarge	1.33	1.25	86.7M

**Table 4.1:** YOLOv5 model scaling. YOLOv5l is the baseline architecture since the related  $s_{depth}$  and  $s_{width}$  are both set to 1.

## 4.2.2 Learning

YOLOv5 learns to detect objects by calculating a loss function  $L$  every epoch, based on a cycle-decreasing learning rate scheduling policy, and accordingly backpropagating error to update parameters on par with all the modern ANN architectures. Such loss is the weighted sum of three distinct components:

$$L = \lambda_1 L_{cls} + \lambda_2 L_{obj} + \lambda_3 L_{box} \quad (4.1)$$

The first two are BCE functions as that used for binary classification, here needed to meet the requirements of correctly predicting labels and not missing present objects respectively. Since only one label is considered in this work ('0'  $\rightarrow$  'motor'),  $L_{cls}$  is zero-constant during training. The last one is a Complete IoU loss [110], which takes into account the discrepancy between true and predicted boxes, namely  $b_{true}$  and  $b_{pred}$ , based on IoU:

$$L_{CIoU} = 1 - \text{IoU} + \frac{d(b_{true}, b_{pred})}{c^2} + \alpha v \quad (4.2)$$

where  $d$  is the Euclidean distance between  $b_{true}$  and  $b_{pred}$ ,  $c$  is the diagonal length of the smallest box covering both of them,  $\alpha$  is a trade-off parameter, and  $v$  is a measure of the aspect ratio consistency.

Going back to learning rate, YOLOv5 scheduler allows to have one non-constant over epochs, decreasing linearly or cosinely up to become almost 0, then restart repeating the cycle. This guarantees optimal training thanks to the slowing descent of the gradient. Moreover, scheduling begins after a few warmup epochs, during which learning rate can slightly change starting from some initial value and ending in a specific set point.

With regard to bounding boxes, they are computed and expressed in the *ccwh*

form similarly to the YOLOv2 proposed approach [111]:

$$\begin{aligned}
 b_x &= 2\delta(t_x) - 0.5 + c_x \\
 b_y &= 2\delta(t_y) - 0.5 + c_y \\
 b_w &= p_w(2\delta(t_w))^2 \\
 b_h &= p_h(2\delta(t_h))^2
 \end{aligned}
 \tag{4.3}$$

where  $b_x$ ,  $b_y$ ,  $b_w$ , and  $b_h$  are the center  $(x, y)$ , the width  $w$ , and the height  $h$  of box respectively. What is more, it is possible to use the AutoAnchor algorithm [112] to find the best anchors for training when the built-in ones are found not to be good for the dataset. It is an interesting method that exploits K-Means, involving all the boxes in training set, to retrieve an initial guess for anchors that are to be selected once training starts. This helps to achieve better results especially when dealing with custom data.

Data augmentation is available thanks to NumPy and OpenCV, which enable typical operations to augment images such as translation, rotation, shearing, scaling, and many others.

At the end of each epoch, a measure of how well the model performs, called *fitness*, is calculated exploiting precision, recall, and mAP:

$$\text{Fitness} = w_1 \times \text{Prec} + w_2 \times \text{Rec} + w_3 \times \text{mAP}^{0.5} + w_4 \times \text{mAP}^{0.5:0.95}
 \tag{4.4}$$

It is nothing but a weighted sum of the four metrics typically used in training neural networks for object detection. By default,  $w_1 = w_2 = 0$ ,  $w_3 = 0.1$ , and  $w_4 = 0.9$ . This choice has most likely been made in order to give more importance to the mAP with varying IoU thresholds, which is probably the most impactful metric in applications of this kind. Although it is always possible to use custom weights, default values have been kept unchanged in this work.

Every time a better value is obtained compared to the current one, the current best model is updated with the new weights providing greater fitness. If the best fitness is recorded at the last epoch, the best model coincides with the last one, thus only the latter is available when training is done.

### 4.2.3 Code structure

YOLOv5 code is open source and available at this [link](#). It is organized in different folders and subfolders containing Python scripts and configuration files, all placed in a main starting directory named `yolov5`, whose structure can slightly change depending on the release. Among all, files to perform training, validation, and inference are present, together with a text file for installing all the required packages at the start. Three important `yolov5` subfolders are:

- **data**, which contains configuration files relative to COCO and other datasets for object detection, together with a subfolder named **hyps** containing configuration files for hyper-parameter setup and data augmentation;
- **models**, which contains configuration files describing the architectures of all the YOLOv5 models available;
- **utils**, which contains classes and functions designed for preprocessing, augmentation, loss and metrics computation, plots, and other stuff.

At the first run, regardless of whether it is training, validation, or inference, a new folder named **runs** is created at the same hierarchical level of the others. Inside it, different subfolders, whose name depends on the mode, are created. Each of them serves to collect all the results relative to a specific run. For example, after 50 consecutive training trials are done, 50 different folders denominated in ascending order, from **exp** to **exp50**, are available at `yolov5/runs/train/` path.

Training can be performed by launching `train.py`, either locally or in the cloud, specifying the options desired in the command line input arguments. They are very many, and some of them are for:

- Choosing the model to be used among those of the YOLOv5 family with the initial weights, that is, random or from COCO pretraining (in this respect, YOLOv5s with COCO weights is the default setting);
- Providing information about data such as the location of training, validation, and test set folders together with the classes included (clearly, all the default specifications refer to COCO);
- Specifying the hyperparameters, including those for data augmentation, as well as the number of epochs (with the patience for early stopping, set to 100 by default), the batch size, and the optimizer for learning (SGD or Adam);
- Setting the input image size (640x640 by default);
- Deciding whether to use the AutoAnchor algorithm for an optimal search of anchor boxes or not;
- Deciding whether to move images in RAM from disk with the aim of speeding up the entire training procedure;
- Specifying the device to be used for calculations: GPU or CPU (if a GPU is present, it is automatically taken);
- Specifying the number of layers to be frozen in the case of Transfer Learning.

Additional options for other configuration aspects are available but they have not been discussed here not having been relevant to this work.

Another file, called `val.py`, is to validate (or rather test) a YOLOv5 pretrained model on a test set, including options for:

- Choosing the model to be used: one trained on COCO, thus already available, or one trained on any other dataset (as in the training case, YOLOv5s with COCO weights is the default setting);
- Providing information about data, as already specified above;
- Specifying the batch size, that is 32 by default, differently from the training case where it is 16;
- Setting the input size, that is 640x640 by default, as in the training case;
- Performing image augmentation during testing, namely TTA;
- Halving floating point precision (16-bit instead of the standard 32-bit);
- Setting thresholds for confidence and IoU respectively;

Similarly, inference is done by running the `detect.py` file, with options for:

- Choosing the model to be used: one trained on COCO, thus already available, or one trained on any other dataset (as in the training case, YOLOv5s with COCO weights is the default setting);
- Providing information about data, as already specified above;
- Specifying the source, i.e., the origin of images, videos, or streams on which YOLO must perform detections (it can be a number identifying a camera, a path referring to a local image/video, a link of a web video, and so on);
- Setting the input size, that is 640x640 by default, as in the training case;
- Specifying the device to be used, as already discussed before;
- Performing TTA (computationally expensive at inference);
- Halving floating point precision (16-bit instead of the standard 32-bit);
- Setting thresholds for confidence and IoU respectively;

As for the training file, further options are present in the other two but they have not been reported since they have not been explored in this work (default values have been kept unchanged for them).

On the basis of object detection theory, it can be said that the threshold choice

to be made when using `val.py` or `detect.py`, both for confidence and IoU, is fundamental to provide decent results, especially at inference time. Hence, they must be chosen properly when the detection file is launched. A trial and error procedure is suggested to find the optimal values, by repeatedly running YOLO with distinct thresholds and seeing how it behaves accordingly. By default, they are set to 0.25 and 0.45 respectively.

### 4.3 Color-based recognition

Before going into detail with the methodology adopted, it is noted that all the trials of this work, expect YOLO training that will be detailed later, have been conducted locally with the aid of Visual Studio Code (no paper available, visit [here](#)) as editor for fast Python coding.

Since the motors are marked with one among three possible colors, it stands to reason that recognition could be performed by the only color search in the image as long as those of interest (red, yellow, and white) do not appear elsewhere in the shot. This strict condition may guarantee that the located red pixels correspond to the red motors, and the same goes for yellow and white ones.

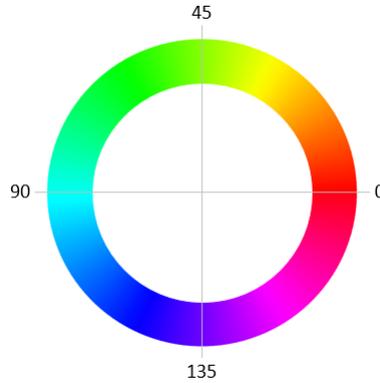
With this premise, OpenCV provides built-in functions which enable thresholding operations similar to those introduced in 1.3.1, one above all `inRange`, that takes three NumPy arrays as input arguments and checks if the elements of the first lie between those of the other two, so that a 255 entry is correspondingly placed in the output array when a match occurs, 0 otherwise. The result, as in the binarization case, is a pure black-and-white image that can be treated as a mask. As is guessable, this goes well with the color detection task, since each triplet can be enclosed in a range with specified extremes. Global thresholding (with  $t = 127$ ) can be traced back to an `inRange`-based confrontation indeed:

$$\begin{bmatrix} 141 & 78 & 144 \\ 13 & 100 & 240 \\ 231 & 127 & 81 \end{bmatrix} \xrightarrow{\text{is in } [127, 255]?} \begin{bmatrix} 255 & 0 & 255 \\ 0 & 0 & 255 \\ 255 & 255 & 0 \end{bmatrix}$$

where the matrix on the left represents a 3x3 grayscale image and `[127, 255]` is the range within pixels must fall to be mapped to 255. In the case of single-channel images like this, the arrays used for comparison are monodimensional but, applying to the generic 3-channel scenario, two triplets must be provided as the lower and upper bound respectively, thus proper arrays.

Before going into more detail, a clarification about the color model to be used is necessary: in fact, since possible light variations must be taken into account, HSV is probably more suitable than RGB for this application. In fact, it is more robust to external conditions and, being closer to the human perception of colors, it is easier to tune the related ranges. Precisely in relation to tuning, minimum and maximum

hue values for tracking a color of interest can be acquired by the wheel shown in Figure 4.3. Normally, hue varies between  $0^\circ$  and  $360^\circ$  but, as anticipated in 1.2.2, OpenCV uses an 8-bit representation on par with that employed for saturation and value, resulting in a halved interval which goes from  $0^\circ$  to  $179^\circ$ , as displayed.



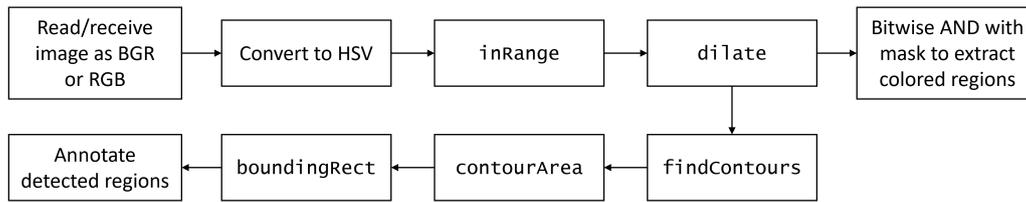
**Figure 4.3:** OpenCV color wheel. Image from Author.

An awkward situation is the one where there is a color having hue varying from a negative to a positive angle, resulting in a range of the type  $[-\alpha, \beta]$ , being  $\alpha > 0$  and  $\beta > 0$ . Such case must be handled by providing two ranges: in fact, since negative angles are positive angles seen counterclockwise, the first hue range goes from  $180^\circ - \alpha$  to  $179^\circ$ , while the second one goes from  $0^\circ$  to  $\beta$ . The remaining saturation and value are the same for both of them. When this occurs, two masks are obtained for the same single color, but only one is needed for detection. It is enough to perform a bitwise OR (through the OpenCV function exactly named `bitwise_or`) between the two masks to have a unique one ready for use. The OR logical operator acts as an adder, and the resulting mask can be accordingly considered as the sum, or rather the union, of the other two. Please, consider the following example:

$$\begin{bmatrix} 255 & 0 & 0 \\ 0 & 255 & 0 \\ 0 & 0 & 255 \end{bmatrix} \vee \begin{bmatrix} 255 & 255 & 0 \\ 255 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 255 & 255 & 0 \\ 255 & 255 & 0 \\ 0 & 0 & 255 \end{bmatrix}$$

Those on the left are the two initial masks, while the one on the right is the final mask obtained by disjunction.

With regard to saturation and value, they should be tuned through trial and error too, although there is not a reference like the color wheel for them. At this point, the functioning of `inRange` applied to 3-channel HSV images is clear: given a pixel, each one among H, S, and V must lie in  $[H_{\min}, H_{\max}]$ ,  $[S_{\min}, S_{\max}]$ , and  $[V_{\min}, V_{\max}]$  respectively. If just one does not match, the corresponding pixel is set to 0 in the output mask, which always happens to be a single-channel binary image having 0 and 255 throughout its matrix.



**Figure 4.4:** Diagram of OpenCV-based color detection. Image from Author.

After this introduction about the potentials offered by OpenCV, the method first tried in this work for recognizing colors, or rather recognizing objects based on their color, is comprehensively described. The idea comes from [113] and is schematized in Figure 4.4. As can be seen, the initial step consists of taking the input image (or video frame) as an array. This can be done through `imread` in the case of a single image, or `VideoCapture` when a video is to be processed and colors are to be identified in real-time at each frame. Both read the input as a NumPy array in the BGR format, but the conversion to RGB is always eligible. The possibility of encompassing the whole detection procedure in a function is also considered: in such a way, one can pass the array as input in one format or other. At this point, once the image is obtained in the array form, the conversion from BGR (or RGB) to HSV can be performed, enabling the successive masking operation with the pretuned HSV ranges provided, exactly as discussed above. OpenCV has built-in methods for such conversions: `COLOR_BGR2HSV` and `COLOR_RGB2HSV`. The masks obtained, one for each color, can be dilated: this is not a mandatory step, but it can help to improve the identification by making them larger. A 5x5 all-ones matrix is used for doing that, in accordance with what has been said in 1.3.4. Here, the chart shown in the figure branches into two flows: one of them consists of a bitwise AND operation that involves the input image and the masks. In this respect, the method `bitwise_and` allows to isolate a region of the source image given a reference shape: in this context, such shape is determined by the mask, that serves to extract each time the region tinted with the corresponding color from the full image.

Going in the other direction, OpenCV also provides two fundamental functions to determine the contours of the retrieved colored regions and then calculate the area inside: `findContours` and `contourArea`, respectively. In simple words, identifying the contours of something means to join all the points that act as its vertexes: this explains why the only color detection could be employed to recognize objects without the aid of an intelligent machine. Returning to the two functions, the former exploits the approach suggested by Suzuki and Abe [114] to find the contours of each mask, taken as input, and return them as a NumPy array in the case of a single dense area, or a tuple of arrays when multiple separated regions are found to contain the same related color. What is more, the contours can be organized in a hierarchy to deal with those instances where some shapes are inside other ones,

and perhaps belong to the same object. `findContours` provides a second input argument for that and, among the hierarchy modes available in OpenCV, the tree-like structure is the one adopted for the color detection algorithm of this work. The third and last argument of the function, when different from none, serves not to return all the contours retrieved but only a subset with the aim of saving memory and increasing speed.

The other function, `contourArea`, is able to calculate the area enclosed by the given contours: this is a measure of how much color has been detected in a certain region. At this point, the process ends with annotating all the colored regions by placing rectangles, equivalent to bounding boxes, in the corresponding positions together with the color labels, just like object detectors do. This is not a necessary step but helps to visually understand how the algorithm is working. The information about bounding boxes, i.e., their coordinates, is obtained by calling the `boundingRect` method, which takes a set of contours as input and returns  $x_{min}$ ,  $y_{min}$ ,  $w$ , and  $h$  of the associated box. They are what `rectangle` and `putText` need in order to draw and label such boxes.

Everything said so far has been overall implemented as a custom Python function named `detect_by_mask` that takes as input arguments:

- A NumPy array representing the input image or frame;
- A string list containing the names of the colors to be searched;
- A list of lists, where each one contains two NumPy arrays relative to the  $HSV_{min}$  and  $HSV_{max}$  extremes for one color;
- A flag indicating whether the passed array is BGR or RGB-formatted (`BGR = False` by default);
- A flag indicating whether dilation on masks is enabled or not (`dilate = True` by default);
- An integer expressing the size of the filter for dilation (`n = 5` by default);

In the case of real-time color detection to be performed on a video, the function is enclosed in an infinite loop so that each frame can be taken by `VideoCapture` as a BGR NumPy array and passed to `detect_by_mask` as the first input argument. The same image is returned with the boxes and the labels drawn, hence it can be displayed in a window on the screen. The loop is broken when the video finishes, or when an exit command (for example, the shortcut Ctrl+C) is invoked from keyboard. In the main file, called `detect.py`, a parser has been included to let users run the program from the command line, with two options, `--source` and `--color-detect`, to specify where the current frame comes from (a webcam, a camera, a local video, a local image, etc.) and the colors one wants to search. They can be encoded in the form of a CSV file with the following columns:

`color,R,G,B,Hmin,Smin,Vmin,Hmax,Smax,Vmax`

As suggested by their names, each entry must have its string-formatted color name on the first column, its RGB triplet between the second and the fourth column, and its tuned HSV extremes going from the fifth to the seventh and from the eighth to the last column respectively.

The possibility of using a text file instead of CSV has also been considered, with the same column order. The difference is that columns are not labeled and values are separated by white spaces, not commas.

The numbers provided must be coherent with what they represent: **R**, **G**, and **B** must be between 0 and 255, as well as **Smin**, **Smax**, **Vmin**, **Vmax**, additionally constrained to  $Smin \leq Smax$  and  $Vmin \leq Vmax$ . **Hmin** and **Hmax** can be positive or negative, ranging from 0 to 180 or from -180 to -1, on condition that  $Hmin \leq Hmax$ . Both 180 and -180 are mapped to 0, while negative values greater than -180 are converted into their positive additional counterpart by adding 180. One entry with negative **Hmin** and positive **Hmax** is split into two in the same way discussed before, resulting in a color with double range.

The custom function `read_colors_from_file` is responsible of reading the CSV or text file, checking whether it meets the requirements discussed through another custom method called `check_color_format`, and extracting its content returning: a list of the color names, a list of the RGB triplets encoded as tuples, and a list of lists with each one wrapping the two NumPy arrays that form the HSV range for one color.

## 4.4 YOLO and colors

Due to the limitations in recognizing objects by color through the only algorithm of color detection introduced before (as will be evident in the next chapter), the idea was born to combine YOLO with an algorithm that could search colors inside the bounding boxes rather than in the full image. The points made on the use of HSV ranges also apply to all the methods that will be presented soon.

Each of them has been implemented as a Python function and inserted in a custom library called `colors.py` placed in `yolov5/utils`. To extend YOLO with color detection at inference, the same new option `--color-detect` discussed in 4.3 has been added to the parser of `detect.py`, giving the opportunity of passing a custom color file whose entries are checked before effectively starting avoiding runtime errors, as in the pure color detection scenario. Accordingly, new code lines have been included in the detection file so that, at each frame, the network locates the motors through bounding boxes, and the algorithm for color search is subsequently run to determine those inside.

### 4.4.1 YOLO-based recognition

Before discussing methods for real-time color identification associated with object detection, a few lines are necessary to describe the training of YOLO for recognizing motors, that is the fundamental and the most challenging task prior to that about colors. Due to the high computational power requested by neural networks for training, YOLO included, the platform chosen for this part has been Google Colab [115], an online Jupyter-like notebook that provides a cloud-based virtual machine environment with GPU acceleration. In particular, Tesla T4 is the simulated graphics card usable to significantly speed up calculations. In this connection, Table 4.2 compares the hardware specifics of both the Colab environment used for training and the local machine used for the rest of this work.

	Laptop	Colab
CPU	Intel(R) Core(TM) @ 2.30GHz	Intel(R) Xeon(R) @ 2.20GHz
GPU	NVIDIA GeForce 920MX (2 GB)	Tesla T4 (16 GB)
RAM	8 GB	~12 GB
Disk	446 GB	~40 GB

**Table 4.2:** Hardware differences between laptop and Colab free tier.

Going back to the procedure, the virtual images obtained from Unity have been put in a folder named `train_data_base`, organized as follows:

```
train_data_base
  /images
    /train
    /val
    /test
  /labels
    /train
    /val
    /test
```

It has then been included in `yolov5/datasets`, where `datasets` is an ad-hoc folder created and added to the original YOLOv5 directory. In this way, training, validation, and test set images together with their label files can be retrieved by the model on its own by consulting the `custom_data.yaml` file placed in `yolov5/data`, which exactly serves to provide the path of each training data subset.

It should be noted that bounding boxes with negative coordinates may appear in the annotation files coming from Unity due to the flawed procedure adopted for generation. To simplify things, they are removed together with their relative images, not taking part in the dataset effectively used anymore. Likely, they are a small portion of the full set.

Multiple trials have been done to get a final tuned model able to capture the motor features, thus providing decent detection results in relation to the metrics typically employed in applications of this kind. Based on the parser options previously mentioned, all such experiments have been conducted by repeatedly running the `training.py` file from command line with different parameter configurations each time. A consistent use of image augmentation has been done exploiting the built-in YOLOv5 hyperparameter file `hyp.scratch-high`, located in `yolov5/data/hyps`, which includes settable numbers for random translations, rotations, etc. Intermediate and final results, including losses, metrics, and so on, have been collected in primis to make a first assessment on the goodness of the model, then to storage such information to a cloud database complying with a modern data processing architecture that will be explained in the next chapter. This secondary step is very useful to keep track of all the trials performed and rapidly compare with each other, inferring which model, among all those trained, is the most suitable for the intended purpose.

Each new model trained on the virtual dataset has been evaluated both on the virtual test set mentioned before and the small one containing the manually annotated screenshots of real videos discussed at the beginning of Chapter 3. The `val.py` file has been used for that. Further evaluations have been also done at inference through `detect.py`, collecting results with a statistical analysis method, not included by default in the file above, that will be presented in 4.5. In such a case, typical metrics used in object detection performance assessment such as mAP can not be employed, thus a different strategy was to be thought to have continuous updating, frame by frame, on how well the model works in real time. Information about color has been also considered here, not gauged at training and validation time instead.

As will be seen in Chapter 6, models trained on virtual data have provided excellent results on equally virtual images, but performance had dropped when run on real samples. This has been the reason of a new subsequent training phase, involving the few available real data, to obtain a fine-tuned network capable of providing better results according to the TL approach discussed in 1.4.5. The relative folder, having the same structure as that of virtual data, has been called `train_data_tl` in this case.

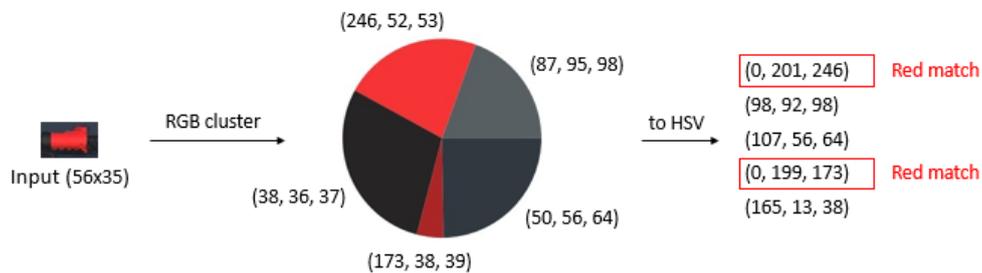
## 4.4.2 In-box color detection

### K-Means pixel clustering

It is interesting to note that K-Means goes well with color quantization: in fact, it can group pixels so that the centroids of the final clusters can be used to form a new palette of  $k$  colors [20]. Furthermore, the colors obtained can be also considered as the dominant ones of the input image [116]. Therefore, the possibility of using it for implementing color detection inside the bounding boxes has been explored in

this work. In this connection, the scikit-learn package offers built-in functions for K-Means and Mini-Batch K-Means that, when applied, return a classifier object provided with methods for fitting, predicting, and finally recovering centroids once clustering is done. Since bounding boxes are reasonably small, standard K-Means can be used instead of its Mini-Batch variant.

An interesting aspect of this procedure is that the NumPy arrays representing boxes are to be reshaped from 3-channel tensors to 2D matrices by multiplying the first dimension by the second one, thus arranging red, green, and blue channels over the three columns obtained. As a result, all the RGB triplets are on the rows, and the matrix is nothing but a dataset of  $m$  samples, being  $m$  the total number of pixels.



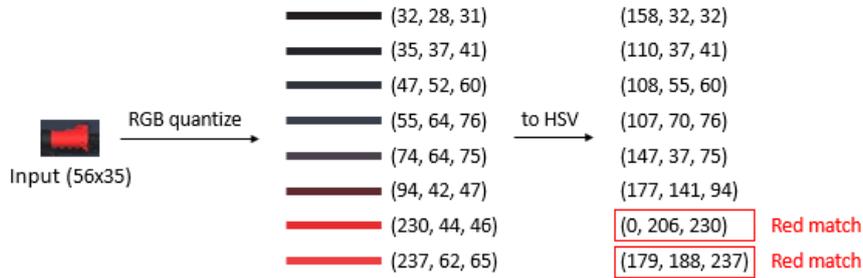
**Figure 4.5:** In-box color detection by K-Means clustering with  $k = 5$ . Once the final centroids are defined, the matching function determines whether one of the colors of interest is present. Image from Author.

Once the final clusters are available, its centroids can be picked as the dominant colors and, at this point, a matching function comes into play to check if one of them falls within one of the HSV ranges among red, yellow, and white. Clearly, to make it possible, the RGB triplets returned by the algorithm must be converted to HSV beforehand. The whole method, as explained so far, is reported in Figure 4.5.

With regard to the choice of  $k$ , the elbow method has been used in the same way as discussed in 1.4.1 by running K-Means on a predefined set of small motor images, equivalent to their bounding boxes, and calculating the mean inertia over them to vary the number of clusters each time.

### Median Cut color quantization

The possibility of intending K-Means as a color quantization algorithm has been just discussed. With greater reason, Median Cut can be employed for the same purpose having been designed precisely for that. The procedure is quite trivial in this case as against the K-Means approach described before: once quantization is performed, a reduced palette of  $n$  colors is available for being given as input of the matching function considering that everything said about HSV matches in the previous part is still valid here. With regard to the Python implementation, the



**Figure 4.6:** In-box color detection by Median Cut quantization with  $n = 8$ . Once the new palette is defined, the matching function determines whether one of the colors of interest is present. Image from Author

library Pillow provides an object called Image that enables, through its methods, color reduction in three consecutive steps: the NumPy input array is reconverted to a viewable image, the quantization is done directly on it, and the resulting image is transformed again to be an array.

Similarly to the K-Means case, the number of colors  $n$  is a hyperparameter to be tuned. A trial and error procedure on a set of motor bounding boxes has been likewise performed to get an optimal value for it.

### Pixel masking

The approach of pixel masking with the help of OpenCV functions, explained in 4.3, is still valid when taken inside bounding boxes rather than in entire images. The procedure is mostly the same, with the difference that images are smaller resulting in fewer pixels to check and mask. This has a beneficial impact on the computational complexity, therefore, it implies a faster algorithm.

One source of inspiration has been the enhancement of YOLOv2 for traffic light status identification, namely detecting semaphores and understanding which light is on [117]. In that work, information about color is obtained by calculating red and green ratios in the located boxes, and not in the image area, as the quantity of red and green pixels over the totality of pixels in the box. Then, the use of a threshold is suggested to determine whether the semaphore displays one color or the other: a red ratio above it implies the presence of red, and the same goes for green. It is noted that the yellow case is not covered there.

Well, the full technique presented in 4.3, which aims to search colors in the entire image, can be reconsidered and accordingly modified to make it compliant with this new scenario. In this sense, the scheme of Figure 4.4 roughly remains unchanged, with some differences (see Figure 4.7). For one thing, the last two stages are removed since bounding boxes and their annotations are now provided by YOLO, while the pixel masking algorithm is limited to pass the information retrieved about colors



**Figure 4.7:** In-box color detection by pixel masking. The same procedure applies to yellow and white searches. The area providing the greatest over-threshold pixel ratio of the three determines the predicted color. The bitwise AND step has not been included in the implementation, thus it is not displayed. Image from Author.

as input to the built-in YOLOv5 labeling method, namely `box_label` from the `Annotator` class (available at `yolov5/utils/plots.py`). Given a detected object, the steps just discussed are replaced with new instructions meant to calculate the pixel ratio for each color, as done in the traffic light problem previously mentioned, to return the corresponding predicted color: none or one among those of interest. Once the area is obtained, the ratio can be calculated by dividing it by the total number of pixels contained in the box, in turn, obtainable as the multiplication between the width and the height. By the way, everything said is viewable in the figure above. Since each mask has an area that provides a certain pixel ratio, the predicted color is the one having the greatest ratio. To make sure that it actually refers to a red, yellow, or white motor, a threshold can be set to decide whether the prediction is honest or not. In fact, there could be cases of grey motors (that is their factory color) mistakenly identified as white due to a positive white pixel ratio, but perhaps small; or even one color assigned to a neutral motor just because some red, yellow, or white pixels have been found in the box in a percentage that is extremely low in relation to the entire rectangle. The introduction of a threshold is thus useful to avoid such mispredictions. It is also true, however, that it could cause false negatives: the algorithm might miss colored motors (red, yellow, or white) in critical light conditions in which colors are altered and at risk of being detected with an under-threshold ratio. That is why it is crucial to assign the most suitable HSV range (or ranges) to each color, as wide as possible to cover all the variations.

One last remark: being the bitwise AND stage independent from the others, it would be better to suppress it in favor of a higher computational speed. In fact, since the algorithm is intended to be run in real time, it is fundamental to include only the instructions necessary to get the final prediction as quickly as possible at each frame, with a time negligible when compared to that taken by YOLO for the object detection part.

## 4.5 Statistical analysis on video frames

Since inference is performed on video footages in real time, a statistical evaluation to be performed at intervals of  $N$  frames has been designed, where  $N$  is a user-defined parameter. As done for color detection, an option has been included in the

parser of `detect.py` to deal with this new feature: `--do-stats`, which accepts as argument the number  $N$  just mentioned indicating how frequently the statistical analysis is to be made. The approach that is going to be described can be computationally expensive, thus it is suggested not to down  $N$  too much. By default, it is set to 30, which means that statistical information is retrieved every second in an ideal situation where the input video is processed at 30 FPS.

Having said this, the method proposed for this project is now presented, considering that the major changes have been effected in the code of `detect.py` and the functions implemented especially for this have been gathered in the relative custom library called `stats.py` and placed in `yolov5/utis`, as precisely done for the color detection addition to the original YOLOv5 project.

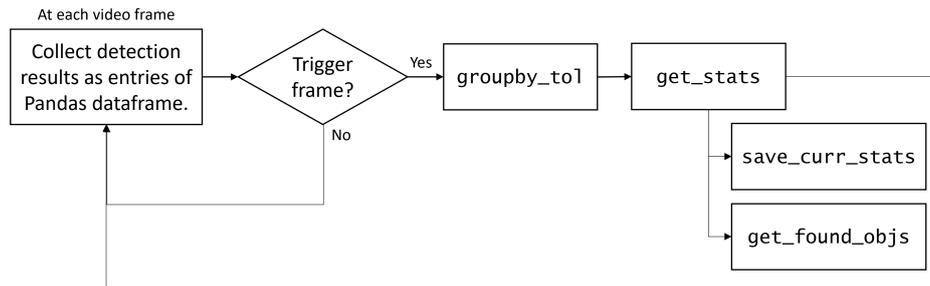
If `--do-stats` is enabled, together with `--color-detect`, a Pandas dataframe is created with the following columns even before images, in the form of arrays, are preprocessed and fed into the model at inference:

```
frame class box conf color
```

Clearly, doing statistics without the color recognition activated is also possible: in that case, results are the ones obtainable with a pure object detector that is unaware of colors. Back to the dataframe, it must be noted that the instruction responsible of adding a new entry is positioned inside the for loop which iterates all the bounding boxes, confidences, and class labels at a single frame. As a consequence, if more than one object is found, multiple entries having the same frame number are inserted into the dataframe. It has not been said yet, but the columns reported above respectively contain the frame number, the class label, the bounding box coordinates in the usual YOLOv5 format, the confidence score, and the name of the predicted color.

Once the mentioned loop is over, the dataframe is found to have several entries, as many as the detections. As the video is processed, such dataframe grows in size with the new information added each time. This raises a question: it has been said before that  $N$  is suggested not to be small since performing statistical operations very frequently could considerably slow down the entire real-time process; but it is also true that a greater  $N$  may imply a substantial number of dataframe entries which engrave on memory, besides being computational heavy for the statistical functions when invoked. The right trade off must be struck.

The actual analysis, schematized in Figure 4.8, starts involving a series of ad-hoc custom methods from the `stats.py` library no earlier than  $x_c$  is retrieved from each list of bounding box coordinates, going to generate a new list containing the only center coordinates for each motor box at that frame. This decision stems from the fact that, from the camera perspective, motors move vertically up and down, resulting in an almost constant  $x$  position in contrast to a varying  $y$  position. On the other hand, if the camera sees motors in a horizontal right-to-left movement,  $x$  is the one that changes, while  $y$  is pretty much steady. To deal with both of these possibilities, a flag indicating the operating mode, that can be vertical or horizontal,



**Figure 4.8:** Diagram of real-time statistical analysis on video frames.  
Image from Author.

has been added to `--do-stats` at a later date. More complex motor paths have not been considered in this work: when it comes to solving CV problems of this kind, the approach should be to put the camera in the most suitable position to facilitate detections indeed.

Going back to the analysis, the list of coordinates obtained is fed into a function named `groupby_tol`, which groups them on the basis of a given tolerance, set to  $10^{-2}$  by default, returning a list of lists. This serves to consider those that differ within the tolerance as related to the same object, in accordance to what has been previously said about motor movement. For instance, one respectively having  $x_c = 0.501$ ,  $x_c = 0.503$ , and  $x_c = 0.508$  at three distinct frames is basically the same motor. The default tolerance is this low because the bounding box coordinates are ordinarily normalized, but either way they must be coherent with the format: in fact, when non-normalized coordinates are used, it is reasonable to use a value in the order of units: 1 or 2, for example. Anyway, it is always good to proceed by trial and error when such delicate parameters are to be tuned, as repeatedly pointed out so far.

After the manipulated list is returned, it is given as input to the `get_stats` function, which calculates three new custom metrics for each sublist:

- Appearance frequency, that is the number of frames in which an object has been found compared to the number of frames elapsed;
- Average confidence, that is the standard object detection confidence averaged over all the frames elapsed;
- Color correctness, that is the most frequent predicted color until then, reported together with the percentage frequency value;

Inside each sublist, the mean coordinate is computed over all the elements included, and the obtained compacted results are stored in the form of a Python dictionary where the mean coordinates are exactly the keys, while all the related information composes the values. This is done by the `save_curr_stats` function, nested in

`get_stats`. It is important to note that when appearance frequency exceeds a threshold  $\delta$ , set to 0.5 by default, it can quite surely be said that a motor has been found in the corresponding position. Such information about effectively detected objects is printable in terminal or directly on the dashboard depending on whether YOLO is launched as a stand-alone framework or a web-like app, as will be detailed later. This step is possible thanks to another `get_stats`-nested function, namely `get_found_objs`.

### Example

At this point, the following simple example is intended to make the functioning principles of this method clearer: consider statistical analysis enabled with  $N = 3$  and, at frame 3, which is the first trigger frame, the dataframe is populated as follows:

frame	class	box	conf	color
1	0	0.442	0.89	red
1	0	0.508	0.85	yellow
1	0	0.700	0.81	white
1	0	0.736	0.90	white
2	0	0.503	0.92	yellow
2	0	0.702	0.94	white
3	0	0.100	0.86	red
3	0	0.445	0.81	red
3	0	0.501	0.80	white
3	0	0.700	0.83	white

For simplicity, only the  $x_c$  coordinate is reported in the `box` column, but do not forget that all the four  $x_c$ ,  $y_c$ ,  $w$ , and  $h$  values are stored as a list in the dataframe. With such data, the output of `groupby_tol` is:

```
[[[0.1, 0.86, "red"]]
 [(0.442, 0.89, "red"), (0.445, 0.81, "red")]
 [(0.501, 0.8, "white"), (0.503, 0.92, "yellow"), (0.508, 0.85, "yellow")]
 [(0.7, 0.81, "white"), (0.7, 0.83, "white"), (0.702, 0.94, "white")]
 [(0.736, 0.9, "white")]]
```

with each row representing a sublist of elements grouped within a 1% tolerance. This list of lists is then passed to the `get_stats` function, which produces:

```
[((0.1, 0.33), 0.86, ("red", 1.0)),
 ((0.4435, 0.67), 0.85, ("red", 1.0)),
 ((0.504, 1.0), 0.86, ("yellow", 0.67)),
 ((0.7007, 1.0), 0.86, ("white", 1.0)),
 ((0.736, 0.33), 0.9, ("white", 1.0)),
```

These statistical information is saved into an ad-hoc Python dictionary through `save_curr_stats`, as already said before, and will further used for the next trigger frames, making the analysis cumulative. The effect of the `get_found_objs` function on such list of tuples is the following, being  $\delta = 0.5$ :

```
((0.4435, 0.67), 0.85, ("red", 1.0)),  
((0.504, 1.0), 0.86, ("yellow", 0.67)),  
((0.7007, 1.0), 0.86, ("white", 1.0)),
```

It can be quite surely said that:

- There is a red motor in  $\bar{x}_c = 0.4435$  with appearance frequency = 0.67, average confidence = 0.85, and color correctness = 1.0;
- There is a yellow motor in  $\bar{x}_c = 0.504$  with appearance frequency = 1.0, average confidence = 0.86, and color correctness = 0.67;
- There is a white motor in  $\bar{x}_c = 0.7007$  with appearance frequency = 1.0, average confidence = 0.86, and color correctness = 1.0.

## Chapter 5

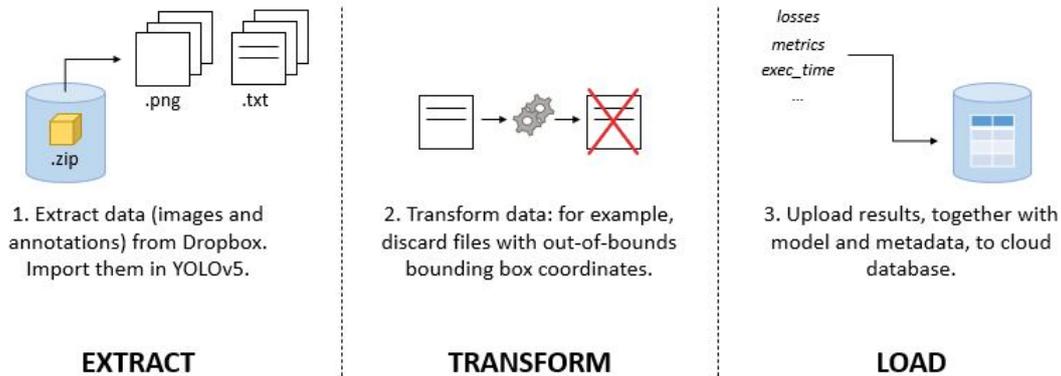
# ETL architecture

This work has been developed according to the modern paradigm of ETL architectures [118], whose acronym stands for Extract, Transform, Load. It is the process of taking data from a certain source, manipulating them with operations such as cleaning, aggregating, separating, or format changing, and then uploading such processed data (or inherent results) to an output destination: a database, in general. The main software items used for realizing that, including the dashboard for inference, have been:

- Dropbox (no paper, visit [here](#)) for storing synthetic and real data in the form of zipped folders, thus importing and unzipping in `yolo5/datasets`, still usable to extract the color file at inference too;
- Python language, already cited in Chapter 5, with specific instructions to check the conformity of annotation files so that those containing negative bounding box coordinates are discarded for training,<sup>1</sup> but also to manipulate eventual color ranges of the type  $[-\alpha, \beta]$ , as discussed in 4.3;
- MongoDB (no paper, visit [here](#)), one of the most popular document-oriented NoSQL databases (as will be clear soon), used in its cloud version Atlas for storage of intermediate and complete training results such as losses, metrics, and other stuff, together with PyMongo (no paper, visit [here](#)), a Python extension to enable MongoDB and Python to communicate with each other;
- Streamlit (no paper, visit [here](#)), an open source framework for Machine Learning and Data Science web apps, namely dashboards for visualizing data and results, adopted for the real-time visualization of detection results via dashboard;

---

<sup>1</sup>Clearly, the relative images are also discarded to avoid inconsistent training.



**Figure 5.1:** ETL architecture of the proposed work.

The two following sections give a description of how data are treated throughout the distinct activities of training, testing, and inference in this project. More precisely, 5.1 refers to the plain ETL structure adopted, while 5.2 focuses on the real-time visualization of detection results and other relevant information at inference by means of a web-like dashboard.

## 5.1 Data path

Figure 5.1 outlines the architecture adopted for this project. Firstly, it can be seen that data, which are the PNG images with their text annotations, are stored in a Dropbox zipped folder. From here, they are extracted and brought to an online notebook, editor, or IDE, like Google Colab which has been the one used for training in this case. Clearly, several trials have been done to get the most suitable network configuration, i.e., the one able to provide decent results from a pure ML perspective. The various losses and metrics, together with the model and metadata such as timestamps, have been saved to a non-relational database from epoch to epoch; this has helped to keep track of the network behavior at each trial, hence a posteriori analysis is always possible and further improvements are feasible. The database selected for that has been MongoDB in its cloud version known as Atlas. Nowadays, non-relational databases are preferable to the classic relational ones in several applications thanks to their flexibility, scalability, and possibility of storing unstructured data such as multimedia files, logs, sensor data, etc. While information is organized in tables in relational databases, thus arranged in rows and columns, it is document-oriented in MongoDB, in the sense that each new entry is not a row of a table having the attributes on columns but it is a JSON-like object containing pairs of fields and values, similarly to Python dictionaries. It follows that it no longer makes sense to speak of tables, but entries, which are documents in

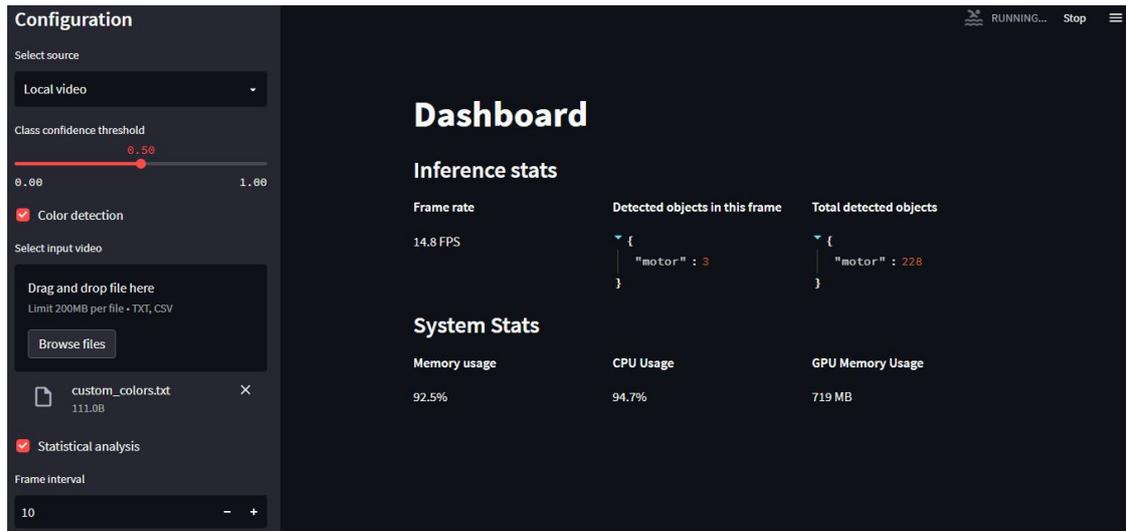
the non-relational context, are grouped in collections. Please, consider this simple example: an `epoch`, numbered as 1, with only two values recorded, that is, `train_loss = 0.44` and `val_loss = 0.67`. In a relational database, the corresponding entry would be a row having 1, 0.44, 0.67 on the first, second, and third column respectively, and maybe `epoch` is the primary key of such table. On the other hand, MongoDB would store this information as `{'_id': 4f0b2f55096f7622f6000000, 'epoch': 1, 'train_loss': 0.44, 'val_loss': 0.67}`. It can be seen that a numeric hexadecimal extra field, called `_id`, is automatically added at the beginning to guarantee the unique identification of the document. This is more or less the same concept as primary key for relational databases, with the difference that primary keys are not mandatory and designers choose which attribute shall be such among those available.

The use of MongoDB has then permitted the storage of the entire PyTorch model at the end of training trials in the form of a byte sequence. This has been possible thanks to the built-in methods for conversion offered by the Python pickle module. For this work, beyond the model, the fields stored at each trial have been the various losses and metrics, which are those provided by YOLOv5, and also metadata such as the starting time of the procedure and the ending time, thus the execution time intended as the total training duration. Information can be retrieved from here to generate reports, which allows not to lose track of everything done.

## 5.2 Data visualization

Data visualization refers to the representation of data in the form of text, graphs, tables, etc., helping to visualize complex information through a simple understandable interface [119]. This can help to have a quick overview of the results provided by the adopted tool, which can be an algorithm, a program, or an app designed for a specific task. More specifically, Video Analytics has become a field of interest in the recent years along with the widespread use of intelligent systems to assist humans in monitoring real-life scenarios, with the ultimate goal of getting useful insights [120]. Typical examples of that are applications meant for surveillance and detection. Hence, the idea was born to create a dashboard, namely a graphical control panel for visualization, for the representation of object and color recognition results in real time, taking inspiration from the traffic monitoring tool proposed by Chachra [121]. In that work, Streamlit has been employed to provide an intuitive interface capable of collecting YOLOv5 inference detection results such as the objects found in the current frame, a summary of all the detections done until then, an estimation about the speed of video processing in terms of average FPS, and some interesting hardware information such as memory and GPU usage. Everything is computed and shown in real time as the input video is run and YOLO locates objects in frames. In order to do that, the `detect.py` file must be accordingly modified, not including a parser anymore since the command line input serves to launch

the Streamlit app this time, not the YOLO inference procedure. In fact, the function `run` responsible of starting YOLO detection is called in the main `app.py` with the input parameters chosen by the user through ad-hoc buttons and menus on the dashboard. These configurable values include the source, the confidence threshold, the color file to be uploaded, and others. In a stand-alone YOLO environment, all of them are specified through the command prompt, but now they are settable directly on the dashboard before detection starts, enabling easier configuration and quick infographic view.



**Figure 5.2:** Example of Streamlit-based dashboard with YOLO integration.

# Chapter 6

## Experiments and results

### 6.1 Color ranges tuning

As stated in Section 4.3, the HSV model is used to perform color detection, with the requirement to provide a range  $HSV_{\min}$ - $HSV_{\max}$  for each one among those of interest. Several trials have been done on some real videos by adopting the OpenCV-based identification method introduced in the same part. The final tuned ranges, collected in Table 6.1, have been obtained from a simple qualitative analysis of the visual performance of recognition. Values were adjusted whenever only a few detections, or too many mispredictions, were visualized. Then, the resulting HSV triplets, both lower and upper bounds, have also been used to identify red, yellow, and white with all three of the techniques presented in 4.4.2.

Color	RGB	HSV <sub>min</sub>	HSV <sub>max</sub>
Red	255, 0, 0	-15, 87, 111	15, 255, 255
Yellow	255, 255, 0	16, 87, 111	42, 255, 255
White	255, 255, 255	0, 0, 168	179, 25, 255

**Table 6.1:** Tuned HSV ranges for color detection.

It can be seen that red has negative  $H_{\min}$  in the face of positive  $H_{\max}$ . Based on the color wheel of Figure 4.3, the purest red tonality appears to have  $H = 0$ , thus the use of an interval of the type  $[-\alpha, \beta]$  is unavoidable. It is also true that a reduced range could be used, covering angles from  $165^\circ$  to  $179^\circ$ , focusing on the darker red and ignoring the one towards orange, but the more complete interval  $[-15, 15]$  ensures greater performance and stability, with robustness to light variations. According to the rules about color ranging established in 4.3, all the intervals enclosing 0 are changed so that two separated ones are obtained, both containing positive angles: in this case,  $[-15, 15]$  is split into  $[165, 179]$  and  $[0, 15]$ . A bitwise OR operation is consequently needed to combine the two relative masks into one when the pixel

masking algorithm is selected.

Since color detection is not performed through ML, its performance can not be rigorously evaluated as in the case of neural network employment for tasks. Moreover, identifying colors is usually a more trivial task as compared to the YOLO-based object detection: it is true that variable light conditions could distort color perception, but classifying and localizing motors in video frames is quite more challenging. Not for nothing, algorithms for color detection are simpler to develop compared to neural networks and other ML solutions.

From some observations, it has been found that red and yellow are always correctly tracked, while the white color is more problematic, essentially for two reasons: the environment contains a lot of gray, starting with the seat frame itself and ending with the bench; then, the only real component that can be tuned for white identification is the value  $V$ , given that  $S$  is enclosed in a very small range and  $H$  does not affect the coloring, consequently. When  $S = 0$ , there is no saturation in the image, which means that everything inside is white, regardless of  $H$ . In this situation, one can vary  $V$  on a grayscale level going from 255, which is pure white, down to 0, which is pure black, obtaining an optimal value for  $V$ , as far as possible. Theoretically,  $S$  could be constantly maintained to 0, but a small variation has been taken into account in setting the interval  $[S_{\min}, S_{\max}]$  to also consider white regions slightly altered by other colors, even though this is a borderline case that should not occur frequently. Another thing to be aware of is that, besides mispredicting frame and bench parts as white, a too low  $V$  could generate a white area greater than those relative to red and yellow motors, resulting in wrong estimations even in those boxes that should not be an issue. Clearly, a different color such as green or blue would fix this matter. This sheds light on how color detection may be sensitive to environmental conditions and, perhaps more importantly, to the choices made to set up the environment itself.

## 6.2 YOLO performance

Among the YOLOv5 models available (see Figure 4.2 and Table 4.1 for a recap), YOLOv5s has been the one immediately adopted for motor detection because of its balance between speed and performance. Various training experiments have been conducted in the virtual Google Colab online notebook, chosen for the powerful GPU acceleration provided, to get a definitive model capable of ensuring decent results in terms of precision, recall, and mAP regarding the pure object detection part. Specifically, the first base training with synthetic data has given the best results for 30 epochs with random initial weights, batch size = 32, and SGD optimizer having initial learning rate = 0.01. Image augmentation (translations, rotations, flips, etc.) and weight decay have been included to prevent overfitting. As can be seen in Table 6.2, the synthetic-data-trained model has suffered a performance drop when validated on real data against the excellent results obtained on synthetic test set

Name/description	Base training	TL training
	<code>hyp.scratch-high.yaml</code>	<code>hyp.VOC.yaml</code>
Initial weights	Random	Base training
Freeze	-	10
Image size	640	640
Batch size	32	16
Epochs	30	50
Warmup	3	3.3835
SGD momentum	0.937	0.74832
Initial learning rate	0.01	0.00334
Weight decay	0.0005	0.00025
$L_{obj}$ gain	0.7	0.51728
$L_{box}$ gain	0.05	0.02
IoU threshold	0.2	0.2

**Table 6.2:** YOLOv5s hyperparameter configuration for training.

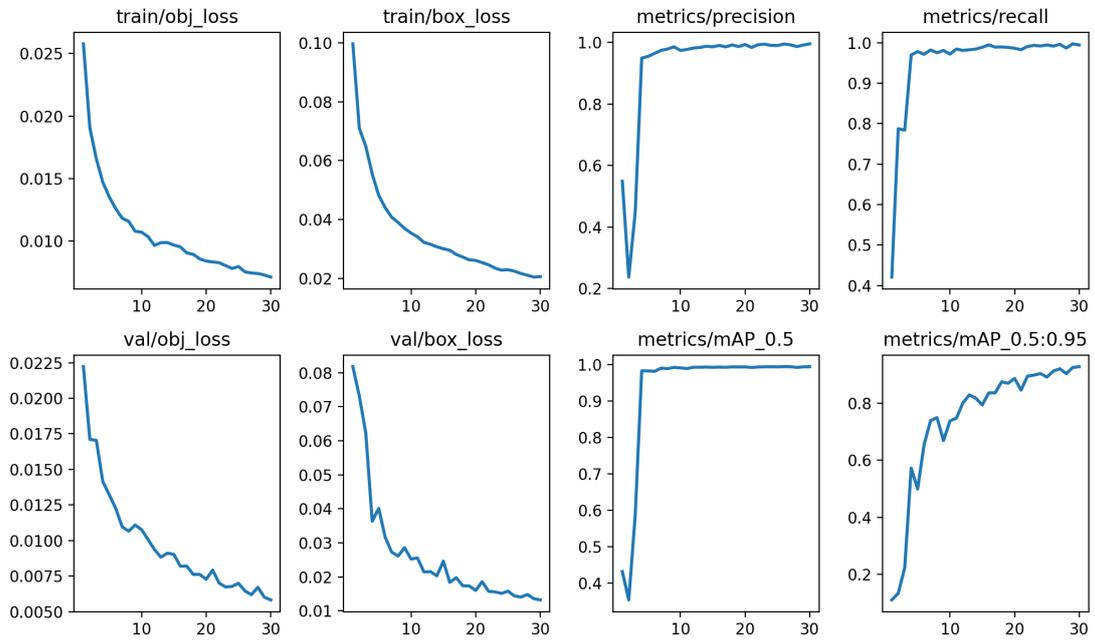
before. This justifies the launch of a second training phase involving the reduced real dataset, doing transfer learning. This time, optimal results have been achieved for 50 epochs, obviously starting from the weights of the previous synthetic-data-trained model instead of random ones. The network backbone, consisting of the first 10 blocks, has been kept frozen during training, and SGD has been selected with batch size = 16 and smaller learning rate = 0.00334. Augmentation and weight decay have been still performed, but less aggressively. Table 6.2 shows how transfer learning has allowed to bring back performance to those of the ideal case of the synthetic-trained-model validated on equally synthetic data.

Table 6.2 above reports a more comprehensive collection of hyperparameter setting both for base and transfer learning training. `hyp.scratch-high.yaml` has been the configuration file for the first training phase, while `hyp.VOC.yaml` the one adopted for transfer learning, originally thought for transfer from COCO to VOC.

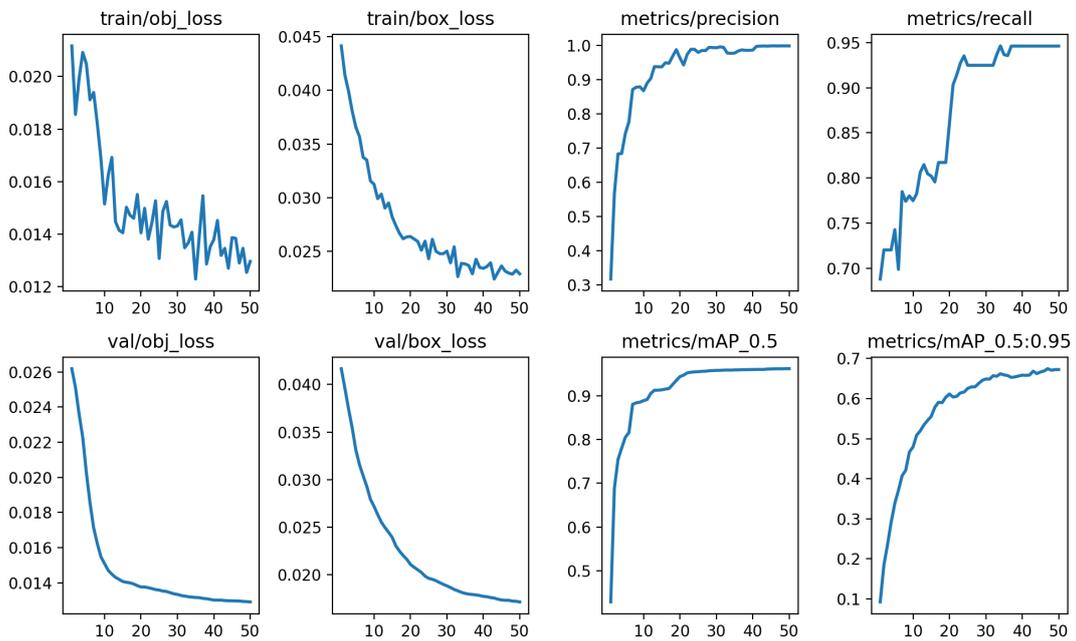
Figure 6.1 shows the training results over epochs both for base training and transfer learning, highlighting how the second training has guaranteed excellent performance on par with the first case. It can be seen, moreover, that no overfitting problems have occurred.

### 6.3 Case study

The final fine-tuned model has then been locally tested at inference on a real video, namely the file called `FILE0020.MOV`, picked from the set of 16 footages available, as anticipated at the end of Section 3. Laptop has been used, with the hardware specifications reported in Table 4.2. The dashboard has also been run together with the



(a) Base training



(b) Transfer learning

Figure 6.1: YOLOv5s training results.

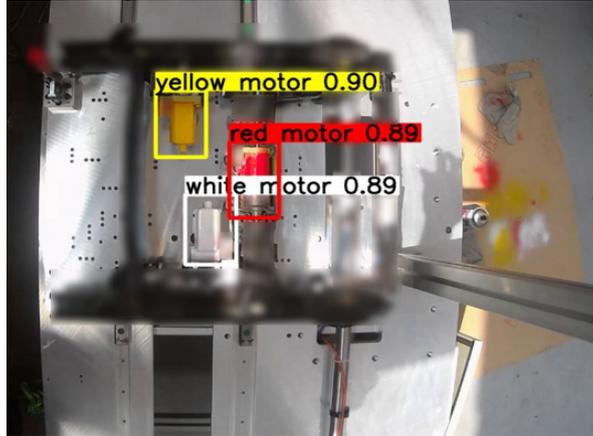
	Training	Test	Prec	Rec	mAP <sup>1</sup>	mAP <sup>2</sup>
1.1	Synthetic	Synthetic	0.988	0.988	0.994	0.901
1.2	Synthetic	Real	0.746	0.665	0.654	0.132
2	Real	Real	0.986	0.946	0.982	0.659

**Table 6.3:** YOLOv5s validation results with synthetic and real dataset. Rows 1.1 and 1.2 refer to the synthetic-data-trained model validated on real data, while 2 to the real-data-trained model validated on real data, which is the real scenario.

mAP<sup>1</sup> is mAP<sup>0.5</sup>, while mAP<sup>2</sup> is mAP<sup>0.5:0.95</sup>.

YOLO-system integrated enabling further evaluations. Moreover, such case study has served to have an idea of how color detection is performed by the three different methods proposed, thus to make a comparison and a subsequent choice. Actually, K-Means pixel clustering, Median Cut color quantization, and pixel masking all provide excellent results in the present scenario, shifting the attention to the real-time computational speed in terms of ms, which affects the FPS capacity of video processing. Pixel masking has revealed to be significantly faster than the other two: 1.1 ms per image against 23.4 of Median Cut and 334.1 of K-Means. Beyond that, the speed proper to YOLOv5s has been also monitored to assess the real time feasibility in general, bearing in mind that the environment and the operational mode of the camera produce a quite static scene which does not require high frame rate. There are not significant changes between one frame and the other when the seat frame moves along the base, thus capturing images at less than 30 FPS does not mean losing much information, likely. Nevertheless, it has been noted that YOLO runs at approximately 15 FPS, which is an acceptable result, when launched separately from the dashboard, namely on different windows or even on two different screens if more monitors are available, while the rate drops to 3 FPS about when is integrated directly with the dashboard. Probably, the remote server connection puts a strain on the showing of consecutive annotated video frames through the Streamlit `st.image` method against the faster OpenCV `cv2.imshow` run separately on its own window. This remains an open problem.

Having said this, Figure 6.2 reports a visual example of what the network "sees" at inference, while Table 6.4 shows how good the model is also in terms of average appearance frequency, average object confidence, and average color correctness, which is in turn a form of confidence. These results have been retrieved by the statistical analysis performed on video frames with interval  $N = 10$ .



**Figure 6.2:** Snapshot of real-time object and color detection.  
The seat frame is blurred for copyright reasons.

<b>Motor</b>	$x_{c,avg}$	$freq$	$conf_{obj,avg}$	$conf_{color,avg}$
1 (white)	0.518359	0.91	0.83	0.9996
2 (red)	0.560938	0.95	0.96	1
3 (yellow)	0.491016	0.93	0.87	1

**Table 6.4:** Statistical analysis results on a real video.

# Conclusions

This work has highlighted the possibility of using a PyTorch-based CNN, specifically YOLOv5, to recognize seat frame motors in an industrial environment, with the extra feature of identifying colors inside detected bounding boxes from a set of three predefined tints.

The difficulty of collecting a suitable number of variegated photos has given rise to the need of generating a synthetic dataset for YOLO training. This has been one of the most challenging tasks of the project because it is not easy to faithfully recreate the real scenario by means of a 3D simulated environment. Performance degradation recorded in validating the synthetic-data-trained model on real samples has been the proof.

Fortunately, the availability of few video footages has allowed to extract screenshots and use them as an additional smaller dataset for transfer learning, one of the most popular methods to adapt the behavior of a neural network to a different task to date. Thanks to this passage, surprising results have been achieved in terms of precision, recall, and mAP on real data, on par with those previously obtained with the synthetic-data-trained model validated on equally synthetic data.

With regard to color detection, pixel masking has turned out to be the best technique among the three proposed, not so much in accuracy since all of them are very precise, but in terms of real-time computational speed. The average time required by pixel masking to detect colors inside boxes has been importantly smaller than the one taken by K-Means pixel clustering and Median Cut color quantization. Anyway, these last two methods are still usable in object detection tools based on single images instead of video streams.

The adoption of an ETL-like architecture has then allowed to keep track of all the training trials done during the development of the present work, beyond visualizing data in real time through a user-friendly dashboard when the entire recognition framework is run at inference. As already said before, this can help human operators to rapidly check information on screen while YOLO works.

Due to the exhaustive results discussed in Chapter 6, multiple future developments are still possible for this work such as correcting the sensitivity of the color detection algorithms to the environmental conditions, especially in relation to light variations and object colors which also appear in the background. The HSV ranges

for detection are not that easy to tune, thus a significant improvement may be given by the use of tints opposing each other in the color wheel: for example, red, green, and blue, which would allow to select wider intervals.

In relation to the pure object detection part, an interesting option is to migrate the entire framework proposed to the newbie YOLOv7 [122] and see what happens in terms of metrics but especially FPS rate in real-time video processing. In fact, this brand new version of YOLO promises to be even faster than its predecessors. Such a network can most likely enable the real-time visualization of annotated detected frames directly in the dashboard at inference, solving the problem of low processing speed when YOLO is integrated instead of been run separately on a dedicated window or secondary monitor. Such a result should also be obtainable by means of a more powerful hardware package: in other words, more computational resources which guarantee higher speed.

What is more, the fact of detecting other seat frame components might be investigated, resulting in a more complete fine-tuned model capable of recognizing multiple classes, not only a single one, on par with the majority of modern detectors. Clearly, with the increasing of identifiable classes, the complexity of the task is greater, requiring a closer inspection of the whole training process, from the preliminary data collection/generation to the hyperparameter configuration.

One last aspect, surely less important for the purpose of recognition, is the reporting activity that can be done on the basis of information stored in MongoDB. Similarly to the dashboard implementation, ad-hoc tools could be adopted to convert all the numeric results uploaded to the database every epoch into an easy infographic. A powerful solution is PowerBI Desktop (no paper available, visit [here](#)), a free scalable platform, able to communicate with MongoDB, precisely designed for data and analytics reporting and visualization stuff.

# Bibliography

- [1] Matti Pietikäinen and Olli Silven. *Challenges of Artificial Intelligence – From Machine Learning and Computer Vision to Emotional Intelligence*. 2022. arXiv: [2201.01466](https://arxiv.org/abs/2201.01466).
- [2] Victor Wiley and Thomas Lucas. “Computer Vision and Image Processing: A Paper Review”. In: *International Journal of Artificial Intelligence Research* 2.1 (June 2018), pp. 28–36. ISSN: 2579-7298. DOI: [10.29099/ijair.v2i1.42](https://doi.org/10.29099/ijair.v2i1.42).
- [3] Richard Szeliski. *Computer Vision. Algorithms and Applications*. Springer London, 2011. ISBN: 978-1-84882-934-3. DOI: [10.1007/978-1-84882-935-0](https://doi.org/10.1007/978-1-84882-935-0).
- [4] James Le. *The 5 Computer Vision Techniques That Will Change How You See The World*. Apr. 22, 2018. URL: <https://www.linkedin.com/pulse/5-computer-vision-techniques-change-how-you-see-world-james-le> (visited on 04/08/2022).
- [5] Julie Blumberg and Gabriel Kreiman. “How cortical neurons help us see: visual recognition in the human brain”. In: *The Journal of Clinical Investigation* 120.9 (Sept. 2010), pp. 3054–3063. DOI: [10.1172/JCI42161](https://doi.org/10.1172/JCI42161).
- [6] Leyla Isik et al. “The dynamics of invariant object recognition in the human visual system”. In: *Journal of Neurophysiology* 111.1 (Jan. 2014). PMID: 24089402, pp. 91–102. DOI: [10.1152/jn.00394.2013](https://doi.org/10.1152/jn.00394.2013).
- [7] Himanshi Singh. *How Images are stored in the computer?* Mar. 16, 2021. URL: <https://www.analyticsvidhya.com/blog/2021/03/grayscale-and-rgb-format-for-storing-images/> (visited on 10/19/2022).
- [8] Zbigniew Zdziarski. *The Early History of Computer Vision*. July 13, 2018. URL: <https://zbigatron.com/the-early-history-of-computer-vision/> (visited on 10/19/2022).
- [9] Frank Rosenblatt. “The perceptron: A probabilistic model for information storage and organization in the brain.” In: *Psychological Review* 65.6 (1958), pp. 386–408. ISSN: 0033-295X. DOI: [10.1037/h0042519](https://doi.org/10.1037/h0042519).

- [10] Yann Lecun et al. “Gradient-Based Learning Applied to Document Recognition”. In: *Proceedings of the IEEE* 86 (Dec. 1998), pp. 2278–2324. DOI: [10.1109/5.726791](https://doi.org/10.1109/5.726791).
- [11] Md. Zahangir Alom et al. *The History Began from AlexNet: A Comprehensive Survey on Deep Learning Approaches*. 2018. arXiv: [1803.01164](https://arxiv.org/abs/1803.01164).
- [12] Asifullah Khan et al. “A survey of the recent architectures of deep convolutional neural networks”. In: *Artificial Intelligence Review* 53.8 (Apr. 2020), pp. 5455–5516. DOI: [10.1007/s10462-020-09825-6](https://doi.org/10.1007/s10462-020-09825-6).
- [13] Wikipedia contributors. *Color model* — *Wikipedia, The Free Encyclopedia*. 2022. URL: [https://en.wikipedia.org/w/index.php?title=Color\\_model&oldid=1091166634](https://en.wikipedia.org/w/index.php?title=Color_model&oldid=1091166634) (visited on 04/22/2022).
- [14] Noor Ibraheem et al. “Understanding Color Models: A Review”. In: *ARPN Journal of Science and Technology* 2.3 (2012), pp. 265–275. URL: [https://www.researchgate.net/publication/266462481\\_Understanding\\_Color\\_Models\\_A\\_Review](https://www.researchgate.net/publication/266462481_Understanding_Color_Models_A_Review).
- [15] Wikipedia contributors. *Color theory* — *Wikipedia, The Free Encyclopedia*. 2022. URL: [https://en.wikipedia.org/w/index.php?title=Color\\_theory&oldid=1118199318](https://en.wikipedia.org/w/index.php?title=Color_theory&oldid=1118199318) (visited on 04/22/2022).
- [16] Satya Mallick. *Why does OpenCV use BGR color format?* Sept. 27, 2015. URL: <https://learnopencv.com/why-does-opencv-use-bgr-color-format/> (visited on 06/07/2022).
- [17] Salem Saleh Al-amri, N. V. Kalyankar, and Khamitkar S. D. *Image Segmentation by Using Threshold Techniques*. 2010. arXiv: [1005.4020](https://arxiv.org/abs/1005.4020).
- [18] Nobuyuki Otsu. “A Threshold Selection Method from Gray-Level Histograms”. In: *IEEE Transactions on Systems, Man, and Cybernetics* 9.1 (1979), pp. 62–66. DOI: [10.1109/TSMC.1979.4310076](https://doi.org/10.1109/TSMC.1979.4310076).
- [19] Derek Bradley and Gerhard Roth. “Adaptive Thresholding using the Integral Image”. In: *Journal of Graphics Tools* 12 (Jan. 2007), pp. 13–21. DOI: [10.1080/2151237X.2007.10129236](https://doi.org/10.1080/2151237X.2007.10129236).
- [20] Celal Ozturk, Emrah Hancer, and Dervis Karaboga. “Color Image Quantization: A Short Review and an Application with Artificial Bee Colony Algorithm”. In: *Informatica* 25.3 (2014), pp. 485–503. ISSN: 0868-4952. DOI: [10.15388/Informatica.2014.25](https://doi.org/10.15388/Informatica.2014.25).
- [21] Paul Heckbert. “Color Image Quantization for Frame Buffer Display”. In: *SIGGRAPH Comput. Graph.* 16.3 (July 1982), pp. 297–307. ISSN: 0097-8930. DOI: [10.1145/965145.801294](https://doi.org/10.1145/965145.801294).
- [22] George Wolberg. *Geometric Transformation Techniques for Digital Images: A Survey*. Tech. rep. Department of Computer Science, Columbia University, 1988. DOI: [10.7916/D8TH8VRW](https://doi.org/10.7916/D8TH8VRW).

- [23] Shengxi Li et al. *Demystifying CNNs for Images by Matched Filters*. 2022. arXiv: [2210.08521](https://arxiv.org/abs/2210.08521).
- [24] Jesús Vergara. *What are 'training' and 'inference' in Artificial Intelligence?* Jan. 22, 2021. URL: <https://neuroons.com/what-are-training-and-inference-in-artificial-intelligence/> (visited on 10/21/2022).
- [25] Bernhard Mehlig. *Machine Learning with Neural Networks. An Introduction for Scientists and Engineers*. Cambridge University Press, 2021. DOI: [10.1017/9781108860604](https://doi.org/10.1017/9781108860604).
- [26] Salim Dridi. *Supervised Learning - A Systematic Literature Review*. 2021. DOI: [10.31219/osf.io/tysr4](https://doi.org/10.31219/osf.io/tysr4).
- [27] Derek Greene, Pdraig Cunningham, and Rudolf Mayer. “Unsupervised Learning and Clustering”. In: *Lecture Notes in Applied and Computational Mechanics*. Jan. 2008, pp. 51–90. ISBN: 9783540751700. URL: [https://www.researchgate.net/publication/235328198\\_Unsupervised\\_Learning\\_and\\_Clustering](https://www.researchgate.net/publication/235328198_Unsupervised_Learning_and_Clustering).
- [28] Laurens van der Maaten, Eric Postma, and H. Herik. “Dimensionality Reduction: A Comparative Review”. In: *Journal of Machine Learning Research - JMLR* 10 (Jan. 2007). URL: [https://www.researchgate.net/publication/228657549\\_Dimensionality\\_Reduction\\_A\\_Comparative\\_Review](https://www.researchgate.net/publication/228657549_Dimensionality_Reduction_A_Comparative_Review).
- [29] Alex Lamb. *A Brief Introduction to Generative Models*. 2021. arXiv: [2103.00265](https://arxiv.org/abs/2103.00265).
- [30] Jiachong Li. “Regression and Classification in Supervised Learning”. In: *Proceedings of the 2nd International Conference on Computing and Big Data*. ICCBD 2019. Taichung, Taiwan: Association for Computing Machinery, 2019, pp. 99–104. ISBN: 9781450372909. DOI: [10.1145/3366650.3366675](https://doi.org/10.1145/3366650.3366675).
- [31] John T. Hancock and Taghi M. Khoshgoftaar. “Survey on categorical data for neural networks”. In: *Journal of Big Data* 7.28 (Apr. 2020). DOI: [10.1186/s40537-020-00305-w](https://doi.org/10.1186/s40537-020-00305-w).
- [32] Daniel Bashir et al. *An Information-Theoretic Perspective on Overfitting and Underfitting*. 2020. arXiv: [2010.06076](https://arxiv.org/abs/2010.06076).
- [33] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. 2016. arXiv: [1609.04747](https://arxiv.org/abs/1609.04747).
- [34] Stuart P. Lloyd. “Least squares quantization in PCM”. In: *IEEE Transactions on Information Theory* 28.2 (1982), pp. 129–137. DOI: [10.1109/TIT.1982.1056489](https://doi.org/10.1109/TIT.1982.1056489).
- [35] Edgar W. Forgy. “Cluster analysis of multivariate data: efficiency versus interpretability of classifications”. In: *Biometrics* 21 (1965), pp. 768–769.

- [36] David Sculley. “Web-Scale k-Means Clustering”. In: New York, NY, USA: Association for Computing Machinery, 2010. ISBN: 9781605587998. DOI: [10.1145/1772690.1772862](https://doi.org/10.1145/1772690.1772862).
- [37] David Arthur and Sergei Vassilvitskii. *k-means++: The Advantages of Careful Seeding*. Tech. rep. Stanford InfoLab, June 2006. URL: <http://ilpubs.stanford.edu:8090/778/>.
- [38] Patrick Brus. *Clustering: How to Find Hyperparameters using Inertia*. July 29, 2021. URL: <https://towardsdatascience.com/clustering-how-to-find-hyperparameters-using-inertia-b0343c6fe819> (visited on 10/04/2022).
- [39] Lian Yu and Nengfeng Zhou. *Survey of Imbalanced Data Methodologies*. 2021. arXiv: [2104.02240](https://arxiv.org/abs/2104.02240).
- [40] Chigozie E. Nwankpa et al. “Activation Functions: Comparison of Trends in Practice and Research for Deep Learning”. In: *2nd International Conference on Computational Sciences and Technology*. Jamshoro, Pakistan, Jan. 2021, pp. 124–133. URL: <https://pureportal.strath.ac.uk/en/publications/activation-functions-comparison-of-trends-in-practice-and-research>.
- [41] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. “Deep Sparse Rectifier Neural Networks”. In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Geoffrey Gordon, David Dunson, and Miroslav Dudík. Vol. 15. Proceedings of Machine Learning Research. Fort Lauderdale, FL, USA: PMLR, Nov. 2011, pp. 315–323. URL: <https://proceedings.mlr.press/v15/glorot11a.html>.
- [42] Stefan Elfving, Eiji Uchibe, and Kenji Doya. *Sigmoid-Weighted Linear Units for Neural Network Function Approximation in Reinforcement Learning*. 2017. arXiv: [1702.03118](https://arxiv.org/abs/1702.03118).
- [43] Teuvo Kohonen. “The self-organizing map”. In: *Proceedings of the IEEE* 78.9 (1990), pp. 1464–1480. DOI: [10.1109/5.58325](https://doi.org/10.1109/5.58325).
- [44] Dor Bank, Noam Koenigstein, and Raja Giryes. *Autoencoders*. 2020. arXiv: [2003.05991](https://arxiv.org/abs/2003.05991).
- [45] Yun Xu and Royston Goodacre. “On Splitting Training and Validation Set: A Comparative Study of Cross-Validation, Bootstrap and Systematic Sampling for Estimating the Generalization Performance of Supervised Learning”. In: *Journal of Analysis and Testing* 2 (2018), pp. 249–262. DOI: [10.1007/s41664-018-0068-2](https://doi.org/10.1007/s41664-018-0068-2).
- [46] Jia Deng et al. “ImageNet: A large-scale hierarchical image database”. In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. Miami, FL, USA: IEEE, 2009, pp. 248–255. ISBN: 978-1-4244-3992-8. DOI: [10.1109/CVPR.2009.5206848](https://doi.org/10.1109/CVPR.2009.5206848).

- [47] Tsung-Yi Lin et al. “Microsoft COCO: Common Objects in Context”. In: *Computer Vision – ECCV 2014*. Ed. by David Fleet et al. Cham: Springer International Publishing, 2014, pp. 740–755. ISBN: 978-3-319-10601-4. DOI: [10.1007/978-3-319-10602-1\\_48](https://doi.org/10.1007/978-3-319-10602-1_48).
- [48] Mark Everingham et al. “The Pascal Visual Object Classes (VOC) Challenge”. In: *International Journal of Computer Vision* 88 (June 2010), pp. 303–338. ISSN: 1573-1405. DOI: [10.1007/s11263-009-0275-4](https://doi.org/10.1007/s11263-009-0275-4).
- [49] Murat H. Sazli. “A brief review of feed-forward neural networks”. In: *Communications, Faculty Of Science, University of Ankara* 50 (Jan. 2006), pp. 11–17. URL: [https://www.researchgate.net/publication/228394623\\_A\\_brief\\_review\\_of\\_feed-forward\\_neural\\_networks](https://www.researchgate.net/publication/228394623_A_brief_review_of_feed-forward_neural_networks).
- [50] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Learning representations by back-propagating errors”. In: *Nature* 323 (1986), pp. 533–536. DOI: [10.1038/323533a0](https://doi.org/10.1038/323533a0).
- [51] Xue Ying. “An Overview of Overfitting and its Solutions”. In: *Journal of Physics: Conference Series* 1168 022022 (Feb. 2019). DOI: [10.1088/1742-6596/1168/2/022022](https://doi.org/10.1088/1742-6596/1168/2/022022).
- [52] Yuji Roh, Geon Heo, and Steven E. Whang. “A Survey on Data Collection for Machine Learning: A Big Data – AI Integration Perspective”. In: *IEEE Transactions on Knowledge and Data Engineering* 33.4 (2021), pp. 1328–1347. DOI: [10.1109/TKDE.2019.2946162](https://doi.org/10.1109/TKDE.2019.2946162).
- [53] Suorong Yang et al. *Image Data Augmentation for Deep Learning: A Survey*. 2022. arXiv: [2204.08610](https://arxiv.org/abs/2204.08610).
- [54] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: [1502.03167](https://arxiv.org/abs/1502.03167).
- [55] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2014. arXiv: [1409.1556](https://arxiv.org/abs/1409.1556).
- [56] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: [1512.03385](https://arxiv.org/abs/1512.03385).
- [57] Christian Szegedy et al. *Going Deeper with Convolutions*. 2014. arXiv: [1409.4842](https://arxiv.org/abs/1409.4842).
- [58] Mingxing Tan and Quoc V. Le. *EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks*. 2019. arXiv: [1905.11946](https://arxiv.org/abs/1905.11946).
- [59] Zhuang Liu et al. *A ConvNet for the 2020s*. 2022. arXiv: [2201.03545](https://arxiv.org/abs/2201.03545).
- [60] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. *YOLOv4: Optimal Speed and Accuracy of Object Detection*. 2020. arXiv: [2004.10934](https://arxiv.org/abs/2004.10934).

- [61] Jianyuan Guo et al. *Hit-Detector: Hierarchical Trinity Architecture Search for Object Detection*. 2020. arXiv: [2003.11818](#).
- [62] Joseph Redmon et al. *You Only Look Once: Unified, Real-Time Object Detection*. 2015. arXiv: [1506.02640](#).
- [63] Ross Girshick et al. *Rich feature hierarchies for accurate object detection and semantic segmentation*. 2013. arXiv: [1311.2524](#).
- [64] Ross Girshick. *Fast R-CNN*. 2015. arXiv: [1504.08083](#).
- [65] Shaoqing Ren et al. *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*. 2015.
- [66] Aishwarya Singh. *Selecting the Right Bounding Box Using Non-Max Suppression (with implementation)*. 2020. URL: <https://www.analyticsvidhya.com/blog/2020/08/selecting-the-right-bounding-box-using-non-max-suppression-with-implementation/> (visited on 09/26/2022).
- [67] Jan Hosang, Rodrigo Benenson, and Bernt Schiele. *Learning non-maximum suppression*. 2017. arXiv: [1705.02950](#).
- [68] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. *SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation*. 2015. arXiv: [1511.00561](#).
- [69] Kaiming He et al. *Mask R-CNN*. 2017. arXiv: [1703.06870](#).
- [70] Guangle Yao, Tao Lei, and Jiandan Zhong. “A review of Convolutional-Neural-Network-based action recognition”. In: *Pattern Recognition Letters* 118 (2019). Cooperative and Social Robots: Understanding Human Activities and Intentions, pp. 14–22. ISSN: 0167-8655. DOI: <https://doi.org/10.1016/j.patrec.2018.05.018>.
- [71] Chuanqi Tan et al. *A Survey on Deep Transfer Learning*. 2018. arXiv: [1808.01974](#).
- [72] Imaging and Machine Vision Europe. *Industry 4.0: powered by vision, smart factories are coming*. Sept. 21, 2022. URL: <https://www.imveurope.com/feature/industry-40-powered-vision-smart-factories-are-coming> (visited on 11/13/2022).
- [73] Mohd Javaid et al. “Exploring impact and features of machine vision for progressive industry 4.0 culture”. In: *Sensors International* 3 (2022), p. 100132. ISSN: 2666-3511. DOI: <https://doi.org/10.1016/j.sintl.2021.100132>.
- [74] Rahul Rai et al. “Machine learning in manufacturing and industry 4.0 applications”. In: *International Journal of Production Research* 59.16 (2021), pp. 4773–4778. DOI: [10.1080/00207543.2021.1956675](https://doi.org/10.1080/00207543.2021.1956675).

- [75] Rita Ferreira, João Barroso, and Vítor Filipe. “Conformity Assessment of Informative Labels in Car Engine Compartment with Deep Learning Models”. In: *Journal of Physics: Conference Series* 2278 012033 (May 2022). DOI: [10.1088/1742-6596/2278/1/012033](https://doi.org/10.1088/1742-6596/2278/1/012033).
- [76] Zhimin Mo, Liding Chen, and Wenjing You. “Identification and Detection of Automotive Door Panel Solder Joints based on YOLO”. In: *2019 Chinese Control And Decision Conference (CCDC)*. 2019, pp. 5956–5960. DOI: [10.1109/CCDC.2019.8833257](https://doi.org/10.1109/CCDC.2019.8833257).
- [77] Lukas Malburg et al. “Object Detection for Smart Factory Processes by Machine Learning”. In: *Procedia Computer Science* 184 (2021). The 12th International Conference on Ambient Systems, Networks and Technologies (ANT) / The 4th International Conference on Emerging Data and Industry 4.0 (EDI40) / Affiliated Workshops, pp. 581–588. ISSN: 1877-0509. DOI: <https://doi.org/10.1016/j.procs.2021.04.009>.
- [78] Yee Yeng Liau and Kwangyeol Ryu. “Status Recognition Using Pre-Trained YOLOv5 for Sustainable Human-Robot Collaboration (HRC) System in Mold Assembly”. In: *Sustainability* 13.21 (2021). ISSN: 2071-1050. DOI: [10.3390/su132112044](https://doi.org/10.3390/su132112044).
- [79] Isabel Rio-Torto et al. “Automatic quality inspection in the automotive industry: a hierarchical approach using simulated data”. In: *2021 IEEE 8th International Conference on Industrial Engineering and Applications (ICIEA)*. 2021, pp. 342–347. DOI: [10.1109/ICIEA52957.2021.9436742](https://doi.org/10.1109/ICIEA52957.2021.9436742).
- [80] Isabel Rio-Torto et al. “Hybrid Quality Inspection for the Automotive Industry: Replacing the Paper-Based Conformity List through Semi-Supervised Object Detection and Simulated Data”. In: *Applied Sciences* 12.11 (2022). ISSN: 2076-3417. DOI: [10.3390/app12115687](https://doi.org/10.3390/app12115687).
- [81] Chengjun Chen et al. “Monitoring of Assembly Process Using Deep Learning Technology”. In: *Sensors* 20.15 (2020). ISSN: 1424-8220. DOI: [10.3390/s20154208](https://doi.org/10.3390/s20154208).
- [82] Xiaohong Sun et al. “Surface Defects Recognition of Wheel Hub Based on Improved Faster R-CNN”. In: *Electronics* 8.5 (2019). ISSN: 2079-9292. URL: <https://www.mdpi.com/2079-9292/8/5/481>.
- [83] Haisong Huang, Zhongyu Wei, and Liguoyao Yao. “A Novel Approach to Component Assembly Inspection Based on Mask R-CNN and Support Vector Machines”. In: *Information* 10.9 (2019). ISSN: 2078-2489. DOI: [10.3390/info10090282](https://doi.org/10.3390/info10090282).
- [84] Ioannis D. Apostolopoulos and Mpesiana A. Tzani. “Industrial object and defect recognition utilizing multilevel feature extraction from industrial scenes with Deep Learning approach”. In: *Journal of Ambient Intelligence and Humanized Computing* (Jan. 2022). DOI: [10.1007/s12652-021-03688-7](https://doi.org/10.1007/s12652-021-03688-7).

- [85] Theo Gevers and Arnold W.M. Smeulders. “Color-based object recognition”. In: *Pattern Recognition* 32.3 (1999), pp. 453–464. ISSN: 0031-3203. DOI: [https://doi.org/10.1016/S0031-3203\(98\)00036-3](https://doi.org/10.1016/S0031-3203(98)00036-3).
- [86] Reza Fuad Rachmadi and I Ketut Eddy Purnama. *Vehicle Color Recognition using Convolutional Neural Network*. 2015. arXiv: [1510.07391](https://arxiv.org/abs/1510.07391).
- [87] Pan Chen, Xiang Bai, and Wenyu Liu. “Vehicle Color Recognition on Urban Road by Feature Context”. In: *IEEE Transactions on Intelligent Transportation Systems* 15.5 (2014), pp. 2340–2346. DOI: [10.1109/TITS.2014.2308897](https://doi.org/10.1109/TITS.2014.2308897).
- [88] H. Altun et al. “An efficient color detection in RGB space using hierarchical neural network structure”. In: *2011 International Symposium on Innovations in Intelligent Systems and Applications*. 2011, pp. 154–158. DOI: [10.1109/INISTA.2011.5946088](https://doi.org/10.1109/INISTA.2011.5946088).
- [89] Anya Hurlbert. “Colour constancy”. In: *Current Biology* 17.21 (2007), pp. R906–R907. ISSN: 0960-9822. DOI: [10.1016/j.cub.2007.08.022](https://doi.org/10.1016/j.cub.2007.08.022).
- [90] Dongliang Cheng, Dilip K. Prasad, and Michael S. Brown. “Illuminant estimation for color constancy: why spatial-domain methods work and the role of the color distribution”. In: *Journal of the Optical Society of America A* 31.5 (May 2014), pp. 1049–1058. DOI: [10.1364/JOSAA.31.001049](https://doi.org/10.1364/JOSAA.31.001049).
- [91] Mahmoud Afifi. *Semantic White Balance: Semantic Color Constancy Using Convolutional Neural Network*. 2018. arXiv: [1802.00153](https://arxiv.org/abs/1802.00153).
- [92] Simone Bianco, Claudio Cusano, and Raimondo Schettini. *Color Constancy Using CNNs*. 2015. DOI: [10.48550/ARXIV.1504.04548](https://doi.org/10.48550/ARXIV.1504.04548).
- [93] Jonathan T. Barron. “Convolutional Color Constancy”. In: *2015 IEEE International Conference on Computer Vision (ICCV)*. IEEE, Dec. 2015. DOI: [10.1109/iccv.2015.51](https://doi.org/10.1109/iccv.2015.51).
- [94] Nikola Banić, Karlo Košćević, and Sven Lončarić. *Unsupervised Learning for Color Constancy*. 2017. arXiv: [1712.00436](https://arxiv.org/abs/1712.00436).
- [95] Arthur Juliani et al. *Unity: A General Platform for Intelligent Agents*. 2018. arXiv: [1809.02627](https://arxiv.org/abs/1809.02627).
- [96] Steve Borkman et al. *Unity Perception: Generate Synthetic Data for Computer Vision*. 2021. arXiv: [2107.04259](https://arxiv.org/abs/2107.04259).
- [97] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.
- [98] Charles R. Harris et al. “Array programming with NumPy”. In: *Nature* 585 (2020), pp. 357–362. DOI: [10.1038/s41586-020-2649-2](https://doi.org/10.1038/s41586-020-2649-2).
- [99] John D. Hunter. “Matplotlib: A 2D graphics environment”. In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: [10.1109/MCSE.2007.55](https://doi.org/10.1109/MCSE.2007.55).

- [100] Fabian Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12.85 (2011), pp. 2825–2830. URL: <http://jmlr.org/papers/v12/pedregosa11a.html>.
- [101] Gary Bradski. “The OpenCV Library”. In: *Dr. Dobb’s Journal of Software Tools* (2000).
- [102] Adam Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019. arXiv: [1912.01703](https://arxiv.org/abs/1912.01703).
- [103] Wes McKinney. “Data Structures for Statistical Computing in Python”. In: *Proceedings of the 9th Python in Science Conference*. Ed. by Stéfan van der Walt and Jarrod Millman. 2010, pp. 56–61. DOI: [10.25080/Majora-92bf1922-00a](https://doi.org/10.25080/Majora-92bf1922-00a).
- [104] Glenn Jocher et al. *ultralytics/yolov5: v6.1 - TensorRT, TensorFlow Edge TPU and OpenVINO Export and Inference*. Version v6.1. Feb. 2022. DOI: [10.5281/zenodo.6222936](https://doi.org/10.5281/zenodo.6222936).
- [105] Chien-Yao Wang et al. *CSPNet: A New Backbone that can Enhance Learning Capability of CNN*. 2019. arXiv: [1911.11929](https://arxiv.org/abs/1911.11929).
- [106] Kaiming He et al. “Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition”. In: *Computer Vision – ECCV 2014*. Springer International Publishing, 2014, pp. 346–361. DOI: [10.1007/978-3-319-10578-9\\_23](https://doi.org/10.1007/978-3-319-10578-9_23).
- [107] Shu Liu et al. *Path Aggregation Network for Instance Segmentation*. 2018. arXiv: [1803.01534](https://arxiv.org/abs/1803.01534).
- [108] Tsung-Yi Lin et al. *Feature Pyramid Networks for Object Detection*. 2016. arXiv: [1612.03144](https://arxiv.org/abs/1612.03144).
- [109] Mingxing Tan, Ruoming Pang, and Quoc V. Le. *EfficientDet: Scalable and Efficient Object Detection*. 2019. arXiv: [1911.09070](https://arxiv.org/abs/1911.09070).
- [110] Zhaohui Zheng et al. *Distance-IoU Loss: Faster and Better Learning for Bounding Box Regression*. 2019. arXiv: [1911.08287](https://arxiv.org/abs/1911.08287).
- [111] Joseph Redmon and Ali Farhadi. *YOLO9000: Better, Faster, Stronger*. 2016. arXiv: [1612.08242](https://arxiv.org/abs/1612.08242).
- [112] Olga Chernytska. *Training YOLO? Select Anchor Boxes Like This*. Aug. 18, 2022. URL: <https://towardsdatascience.com/training-yolo-select-anchor-boxes-like-this-3226cb8d7f0b> (visited on 10/29/2022).
- [113] Santosh Y. Divekar et al. “Color Detection using Pandas & OpenCV”. In: *International Journal of Advanced Research in Computer and Communication Engineering (IJARCCE)* 10.4 (Apr. 2021), pp. 461–466. ISSN: 2319-5940. URL: <https://ijarcce.com/papers/color-detection-using-pandas-opencv/>.

- [114] Satoshi Suzuki and Keiichi Abe. “Topological structural analysis of digitized binary images by border following”. In: *Computer Vision, Graphics, and Image Processing* 30.1 (1985), pp. 32–46. ISSN: 0734-189X. DOI: [https://doi.org/10.1016/0734-189X\(85\)90016-7](https://doi.org/10.1016/0734-189X(85)90016-7).
- [115] Ekaba Bisong. “Google Colaboratory”. In: *Building Machine Learning and Deep Learning Models on Google Cloud Platform: A Comprehensive Guide for Beginners*. Berkeley, CA: Apress, 2019, pp. 59–64. ISBN: 978-1-4842-4470-8. DOI: [10.1007/978-1-4842-4470-8\\_7](https://doi.org/10.1007/978-1-4842-4470-8_7).
- [116] Shivam Thakkar. *Dominant colors in an image using k-means clustering*. Jan. 26, 2018. URL: <https://medium.com/buzzrobot/dominant-colors-in-an-image-using-k-means-clustering-3c7af4622036> (visited on 10/21/2022).
- [117] XiaWen Zhang et al. “Application Research of YOLO v2 Combined with Color Identification”. In: *2018 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*. 2018, pp. 138–141. DOI: [10.1109/CyberC.2018.00036](https://doi.org/10.1109/CyberC.2018.00036).
- [118] Shaker H. Ali El-Sappagh, Abdeltawab M. Ahmed Hendawi, and Ali Hamed El Bastawissy. “A proposed model for data warehouse ETL processes”. In: *Journal of King Saud University - Computer and Information Sciences* 23.2 (2011), pp. 91–104. ISSN: 1319-1578. DOI: <https://doi.org/10.1016/j.jksuci.2011.05.005>.
- [119] IBM Cloud Education. *What is data visualization?* Feb. 10, 2021. URL: <https://www.ibm.com/cloud/learn/data-visualization> (visited on 11/19/2022).
- [120] Javier Couto and Facundo Lezama. *A Guide to Video Analytics: Applications and Opportunities*. Aug. 29, 2022. URL: <https://tryolabs.com/guides/video-analytics-guide> (visited on 10/29/2022).
- [121] Sahil Chachra. *Video Analytics Dashboard for YoloV5 and DeepSort*. Mar. 27, 2022. URL: <https://sahilchachra.medium.com/video-analytics-dashboard-for-yolov5-and-deepsort-c5994461cb44> (visited on 08/04/2022).
- [122] Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. *YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors*. 2022. arXiv: [2207.02696](https://arxiv.org/abs/2207.02696).