

POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering



**Politecnico
di Torino**

Master's Degree Thesis

**Technological features identification for
robotic assembly in non-structured
environments**

Tutor

Candidate

Prof. ANTONELLI Dario

Salvatore Cascino 282122

20 December 2022

Abstract

The goal of the thesis is to develop a computer vision algorithm that can be used by a cobot with the purpose to recognize different mechanical pieces that are present in the workspace. The recognition of different mechanical pieces done by a cobot can be used for robotic assembly purpose in non-structured environments. A robotic assembly operation of a mechanical part requires that the robot must take the pieces present in the workspace with a precise order to correctly execute the task. Therefore, the goal of the computer vision algorithm is to automate all the process: the cobot can take the right mechanical piece also when it does not know a priori the position of the pieces, or, when they are not in fixed position, and they can be moved in the workspace. In a general assembly operation, the mechanical pieces involved seem similar to each other, for instance they can have the same shape, or they can have the same material or the same dimension but are different in at least one technological feature (they could have different number of holes). For this reason, the computer vision algorithm must be based on differentiating the mechanical pieces from the others, using different technological features. The work is divided in two parts: in the first part the algorithm must recognize what is the right piece among several mechanical pieces present in the workspace, that the cobot must grab. The development of the algorithm has been done by using the OpenCV library of Python and the neural network used as model is the YoloV5. To reach this goal, it has been used an external camera connected to the workstation because the images of the camera have a better resolution than the images of the one integrated in the robot. the thesis deals with mechanical pieces that have small features, the resolution of the images from a camera becomes important and critical. In the second part of the work, after the recognition of the correct mechanical piece, the cobot approaches to the right piece. Then, it must measure the exact position of the barycenter in order to grasp it by paying attention to the moments of inertia. This has been done by using the software TMFlow associated to the Omron cobot. The software uses the internal camera of the cobot. After taking the piece correctly, the robot can manipulate it for assembly purpose.

INDEX

1. Introduction	5
1.1 Robots and cobots	5
2. Model of the cobot	7
2.1 Omron TM5 900	7
2.2 Characteristics	8
2.3 Anatomy of Omron TM5 900	8
2.4 Architecture of the system	10
2.5 Gripper	11
2.6 Integrated camera in the cobot	11
2.7 External camera.....	12
3. Object detection with features	13
3.1 What is machine learning?	13
3.2 Machine learning algorithms	14
3.3 Machine learning vs Deep learning	16
3.4 Deep learning vs Neural networks	17
3.5 Computer vision and object detection.....	18
3.6 Yolo algorithm	19
3.7 Features and mechanical pieces	24

3.8 Python and libraries used	27
4. Code to train the YoloV5 model	29
4.1 YoloV5 architectures	29
4.2 Install and import the dependencies.....	33
4.3 Capture images.....	37
4.4 Label the images	38
4.5 Training command	46
5. Outputs and results of the trained model.....	49
5.1 Results after the training	49
5.2 Output of the models.....	69
6. Code for real-time object detection	73
6.1 Settings	73
6.2 Definition of the workspace.....	75
6.3 Settings for the colour detection	77
6.4 Activation of the external webcam	79
6.5 From pixels to mm	81
6.6 Colour detection	86
6.7 Frame that appears in the workstation	90
7. Integration of software in the cobot	93
7.1 Baricenter and manipulation	93

7.2 TM Flow.....	93
7.3 All nodes used	94
7.4 Flow program	99
8. Sending the coordinates through RoboDK	119
8.1 Connection to the cobot and zero-machine.....	119
8.2 Code to sending the coordinates to the cobot	123
Conclusions	127
Sitography	129
Bibliography.....	133
Acknowledgments	135

1. Introduction

1.1 Robots and cobots

The name “cobot” comes from “collaborative robot”. This means that this type of robots can work and interact together with people in the same work environment. The idea of collaborative robots was born in 1995 from a research mission of the General Motors Foundation. The mission aimed to find a way to allow robots and workers to act and collaborate in the same workspace. The cobots (*Figure 1*) are the newest types of robots that are present nowadays in some industries and companies, they differ from the traditional industrial robots (*Figure 2*) and therefore there are differences between them. The collaborative robots can detect people because there are sensors and advanced visual technology that are implemented on it. This is useful because they can stop immediately changing their mode from working mode to a safety mode: this has led to an increasing of the safety of the people and for these reasons the cobots do not need to be locked up in cages. Instead, the traditional industrial robots do not have the technology of the cobots, and they cannot interrupt their work even if there are people that are near the robot. This could lead to injuries and for this reason unlike cobots, industrial robots need to be locked up. Another difference is in the fact that the industrial robots are autonomous and once they are programmed to do a certain task, they will do that following the fixed program. Instead, the cobots that work together with the workers, must assist and complete difficult tasks that cannot fully be automated. The collaborative robots are small, light and easy to move and for these reasons can be employed almost anywhere and the space where they work can be changed, for instance, when they finish a production process, they can be employed in a different environment making the cobots versatile. In the other hand, the traditional industrial robots are heavy and work in a fixed environment, therefore when they are installed, they rarely are moved. Furthermore, while the industrial robots are complex, suitable for high volume and for fixed production process, the cobots have become a big advantage for the small and medium companies since they increase the

productivity, improve the quality and change the customer demands in a fast way guaranteeing a high-mix production. A cobot is also easier to install and simpler to program with respect an industrial robot making it easier to integrate into a manufacturing or assembly process [4][5].



Figure 1: Omron [1]



Figure 2: KUKA robot [2]

2. Model of the cobot

2.1 Omron TM5 900

The cobot used in the thesis is the Omron TM5-900 (*Figure 3*), which TM5 is the model of the robot and 900 are the maximum mm distance that it can reach. Since it is a collaborative robot, it was designed to guarantee safety for the workers and interaction between humans and cobots in the same work area. The Omron TM Collaborative robots are intended to execute different tasks and applications guaranteeing a flexible production also because the cobot can be easily moved. One of the most important advantages of the Omron TM cobot is that it can fit into small spaces making them adaptable to almost any factory environment. The biggest advantage of this cobot is the integrated vision system, since it is designed for pattern recognition and object positioning. In the thesis the integrated vision system is used to find the pattern of each single piece and its barycenter in order to grasp it.



Figure 3: Omron TM5 900 [3]

2.2 Characteristics

The characteristics of the cobot are shown in *Figure 4* [7].

Model	Reach	Payload	Repeatability	Typical speed
TM5	700 mm	6 kg	± 0.05 mm	1.1 m/s
	900 mm	4 kg		1.4 m/s

Figure 4: Table [7]

2.3 Anatomy of Omron TM5 900

As shown in *Figure 5*, the cobot has a base, six joints and an end-effector [7].

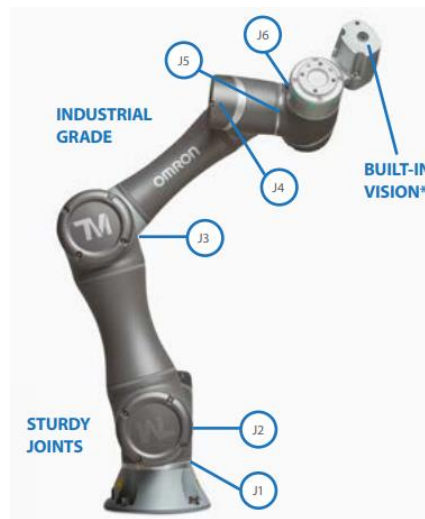


Figure 5: Joints of the cobot [7]

The end-effector is composed by several buttons that implement different functions. As shown in *Figure 6*, the vision button is used to add a new vision node to the execution flow in the program since the cobot is programmed via a software called TM flow. The point button is used to add a new recorded point of the cobot in the flow of execution of the cobot program. The free button, instead, when held down, it enables the hand guidance

mode, this means that the worker can move the cobot by hand in servo-assisted mode. This is important because the users can guide the cobot into positions and automatically the cobot record the position in the software and this makes easy to set points to the cobot. In the *Figure 7* are shown other three elements that are present in the end-effector. In the Analog I/O Connector there are five pins: the supply, the reference, the digital input of the type NPN, the digital output of the type NPN and the last one is the analog input ± 10 V. In the Digital I/O Connector, instead, there are eight pins: the supply, the reference, three digital inputs of the type NPN and three digital outputs of the type NPN. The indicator light ring is important since shows the robot status, since the robot can be in manual mode (green light), automatic mode (blue light). Moreover, if the indicator light ring is red is because there is an error or the cobot is in an initialization mode. When the light is sky-blue, it means that the cobot is in a safe startup mode. As shown in *Figure 8*, the last three elements that are present in the hand of the cobot are the built-in vision system, the gripper button and the end-of-arm tooling flange. The built-in vision system, thanks to the hand guiding and the landmark positioning, allows the cobot to do quick setup for the pick and-place tasks. The gripper button, instead, is a way in programming to close and open the gripper and adds these operations to the execution flow of the program of the cobot. Under the end-of-arm tooling flange it can be possible to insert different tools. For the purpose of the thesis is inserted a gripper [7].



Figure 6: vision node button (1), point button (2), free button (3) [7]



Figure 7: Analog I/O connector (4), light ring (5), Digital I/O connector (6) [7]



Figure 8: Vision system (7), gripper button (8), tooling flange (9) [7]

2.4 Architecture of the system

The whole architecture system (*Figure 9*) is composed by the collaborative robot, the control box and the TMflow software tool. The control box is connected to the cobot and has the goal to control it. The robot stick is also connected to the control box, and it can be possible through the buttons on the robot stick to execute basic operations on the cobot: turn it on, turn it off, run it, increase, or decrease the speed, pause, or stop it. Moreover, thanks to the TMflow software installed in the PC connected to the control box, is possible to write different programs and different tasks that must be executed by the cobot [7].

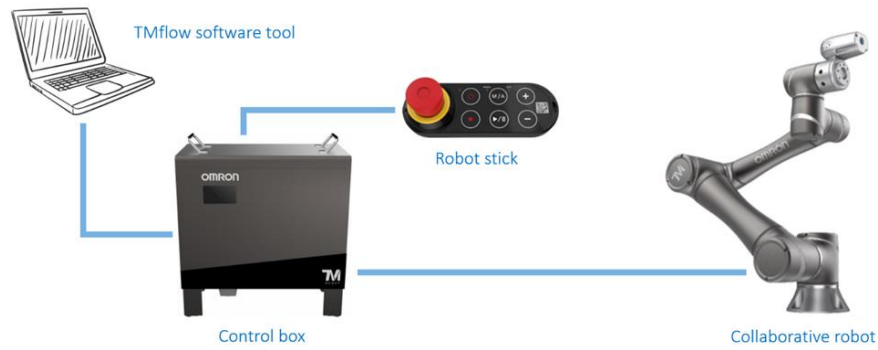


Figure 9: Architecture system of the Omron TM5-900 [7]

2.5 Gripper

To manipulate correctly the piece, it has been installed an adaptive gripper (*Figure 10*). The model of the gripper is the Robotiq 2F 85 that is two fingers adaptive cobot gripper with a maximum opening of 85 mm [8].



Figure 10: Gripper installed in the cobot [8]

2.6 Integrated camera in the cobot

The Omron TM5 900 also includes a camera (*Figure 11*) in the wrist. With its camera is exploited the second part of the work of the thesis: the cobot must measure the exact position of the barycenter in order to grasp it by paying attention to the moments of inertia in order to manipulate the piece correctly. The characteristics of the camera are shown in *Figure 12* [7].



Figure 11: Camera integrated in the cobot [7]

<u>Resolution</u>	5Mpx
Autofocus	100mm $\rightarrow +\infty$ (rolling shutter)
<u>View angle</u>	60° <u>diagonal</u>
<u>Sensor dimension</u>	1/4 inch

Figure 12: Characteristic of the camera [7]

2.7 External camera

To make the first detection and recognize the right piece that the cobot must grab among different pieces in the workspace, it has been used the external camera shown in *Figure 13* from the Everenty company. It is an USB Full HD 2K 4MP 1440P Webcam and as said, the frames from this camera have better quality with respect the ones given by the integrated camera of the cobot [23]. The USB connection allows us to connect the webcam to the workstation from which we can run the Python Code as we can see later.



Figure 13: External camera used [23]

3. Object detection with features

3.1 What is machine learning?

Machine learning is a branch of artificial intelligence (AI). It is a process that uses data and algorithms so that the machines learn and decide autonomously like humans allowing them to solve problems and take actions based on past observations without human help. Machine learning is based on algorithms that thanks to statistical methods are trained to make classifications and predictions [9][10]. Machine learning must not be confused with deep learning, indeed, even if the two concepts are sometimes confused and considered the same thing, actually deep learning is a subfield of machine learning as explained later. At the same time, deep learning and neural networks must not be considered the same thing since deep learning is a subfield of neural networks. As a result, machine learning, deep learning and neural networks are all sub fields of artificial intelligence as shown in *Figure 14* [11].

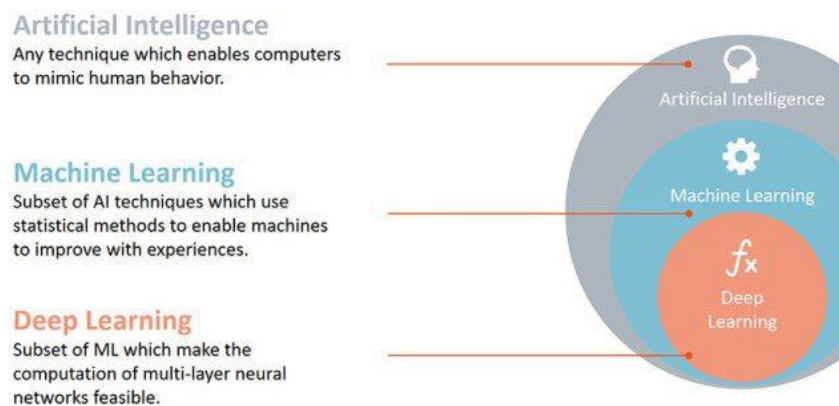


Figure 14: Sub-fields of IA [24]

3.2 Machine learning algorithms

Machine learning can be classified into three families of algorithms: supervised learning, unsupervised learning, and reinforcement learning. In each of them there can be found different techniques and algorithms to make predictions and decisions depending on different cases. The supervised learning algorithms and models make predictions based on labeled training data, where each training sample includes an input and an output. This means that each data is tagged with the correct label (*Figure 15*). The term “supervised” is because these models need to be fed manually tagged sample data to learn from. The goal of the supervised learning algorithm is to analyze the samples data to determine an educated guess when it must determine the labels for unseen data [9][10]. Therefore, it must approximate the mapping function so well that when it has a new input data, it can predict the output [13].

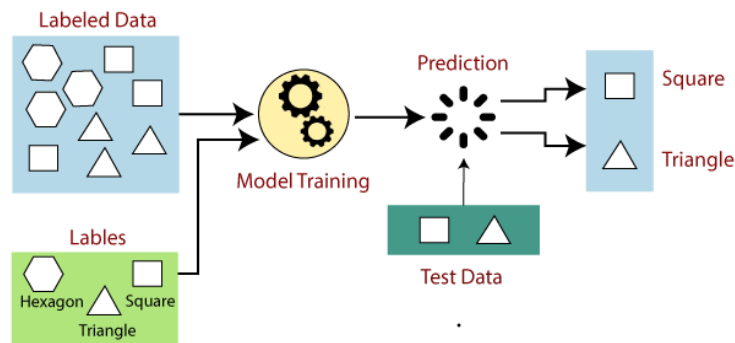


Figure 15: Example of a supervised learning technique [20]

The unsupervised learning analyzes and clusters unlabeled datasets by uncovering insights and relationships on them. In these types of algorithms, the models are fed input data and the desired outputs are unknown (*Figure 16*). Therefore, these types of algorithms discover hidden patterns and data grouping by their owns without any guidance, any human intervention, and any prior training [9][10].

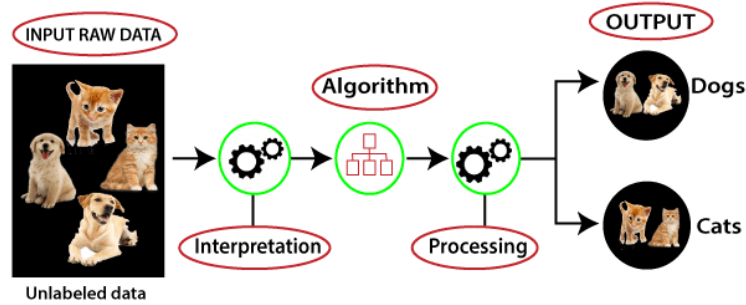


Figure 16: Example of an unsupervised learning [21]

The reinforcement machine learning is a machine learning method similar to the supervised one, but in this case the algorithm is not trained using sample data. Since there is no training data, the machine must learn from its own mistakes without humans' interventions and has the goal to choose the actions that lead to the best solution (*Figure 17*). As a result, this model learns using errors and trials and a sequence of successful outcomes will be reinforced to develop the best action and the best path that it should take in a given situation [9][10].

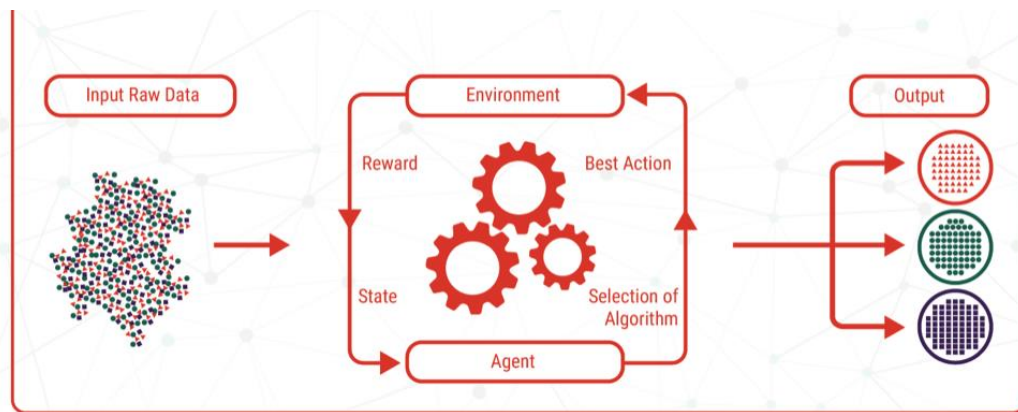


Figure 17: Example of a reinforcement learning [22]

Nowadays the most common techniques and algorithms that are used from machines to make decisions and predictions are linear regression, logistic regression, clustering,

decision trees, random forests and neural networks. The linear regression is a supervised learning algorithm that is used to predict numerical values, based on a linear relationship between different data. The logistic regression, instead, is a supervised learning algorithm that makes predictions for categorical response variables. It is used for classification: for instance, the results of this algorithm can regard the answers “Yes”, “No” to an initial question. The clustering is an unsupervised learning algorithm that identifies patterns in data in order to group them. The decision trees are supervised learning algorithms and are both used to predict numerical values and to classify data into categories. They use a sequence of linked decision that are represented with a tree diagram and for this give the name to this algorithm. The random forest has the goal to predict a value or a category by combine several results from a certain number of decision trees and for this reason are also supervised learning algorithms. The last algorithm are the neural networks: they are supervised algorithms that simulate the way the human brain works, and they are composed by many linked processing nodes. The neural networks are used to recognize patterns and are used specially for image recognition and object detection. For this reason, as explained later, in the thesis are used neural networks in order to recognize different mechanical objects [9].

3.3 Machine learning vs Deep learning

As said, the deep learning is a subset of machine learning, and they differ in how each algorithm learns and how much data are used for the algorithm. The deep learning does not require necessarily a labeled dataset, indeed the deep learning can have as input unstructured data in raw form, for instance images, and it can automatically find and determine the set of features which distinguish a category of data from another one. In the deep learning are eliminated some of the manual human interventions. In the other hand, machine learning is more dependent on human intervention (*Figure 18*) to learn since humans determine the hierarchy of features to understand between different data inputs [9][12].

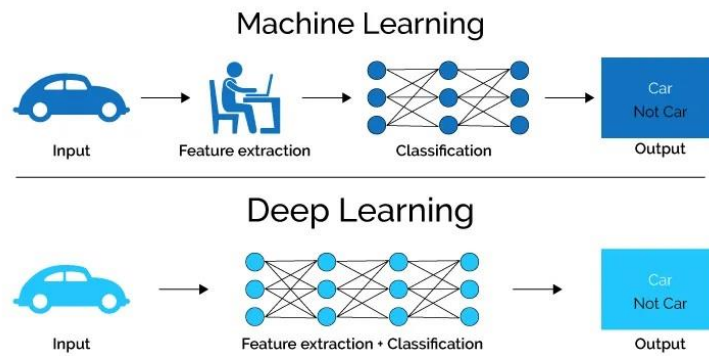


Figure 18: Machine learning vs Deep learning [25]

3.4 Deep learning vs Neural networks

As shown in *Figure 19* the structures of the simple neural network and of the deep learning seem similar each other: both have an input layer, an output layer, and hidden layers in the middle. Indeed, the difference between them is on the complexity of the structure. The term “deep” in deep learning is referred to the depth of the layers in a neural network. A neural network that has more than three layers is considered a deep learning algorithm [12].

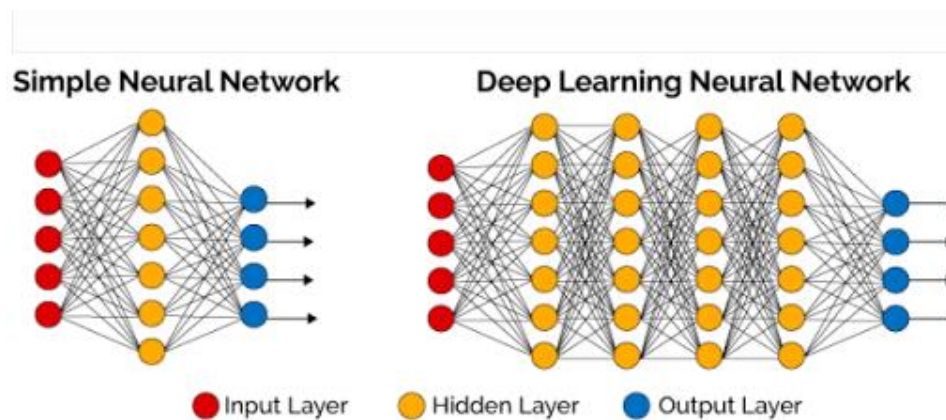


Figure 19: Neural network vs Deep learning [26]

3.5 Computer vision and object detection

Computer vision is a subset of machine learning that takes information from digital images and makes decision based on the information contained in that images. It can also be seen as subset of neural networks (that is a subset of deep learning) but instead of processing simulated data and statistics data, computer vision interprets visual information as images and videos. It requires a large amount of data (visual information) to train the algorithms in order to interpret the data. As a result, computer vision system combines deep learning approaches with hardware like cameras and optical sensors [27]. Computer vision mainly deals with three tasks: image classification, object localization and object detection. Image classification has the goal to assign a class label to an image, predicting the class of an object in an image. The inputs of image classification are images with a single object and the outputs are class labels. Object localization has instead the goal to locate the presence of objects in an image indicating their location with a bounding box around them. In this case the inputs are images with one or more objects and the outputs are one or more bounding boxes drawn around the objects that are defined by a point, width and height. The object detection combines the two previous tasks: draws a bounding box around each object of interest found in an image, localizing it and assigns to it a class label, doing at the same time a classification [15]. It is shown an example of the three tasks in *Figure 20*. The algorithm developed in this thesis has the goal to find the right mechanical pieces (classified with some features) among several other pieces and for this reason the task of this thesis is an object detection task.

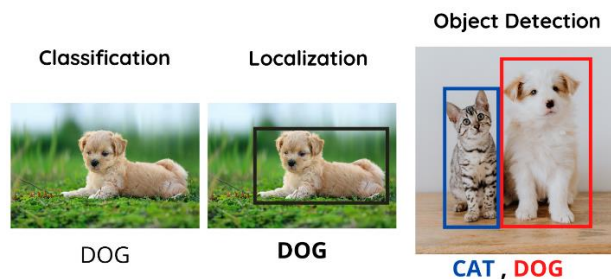


Figure 20: Object detection [28]

3.6 Yolo algorithm

The model used in the thesis to do the object detection to recognize different mechanical pieces is the YOLO model. YOLO is a convolutional neural network (CNN) composed by a single neural network and it is suitable for object detection in real-time. The main operation done by YOLO is the following: the image is divided in cells and it predicts the bounding boxes and the probabilities in each cell for a certain object that must be detected by the algorithm. It gives as output, the objects located with their bounding boxes and probabilities. The word “YOLO” stands for “You Only Look Once”, and this means that the image is seen only one time by the algorithm since the image travels the neural network only once in order to derive predictions about the presence of an object in the image itself. Therefore, the algorithm works in a single step where the image is analyzed, and the outputs are given by the YOLO in the same run. The techniques used in the algorithm are three: the residual blocks, bounding boxes regression and the Intersection over Union (IoU). In the residual blocks technique, that is the first step of the YOLO algorithm, the input image is divided into cells of the same dimension obtaining an image divided into $S \times S$ grid cells (*Figure 21*). If an object appears in a particular grid cell with a certain probability, that cell will be liable for detecting objects.

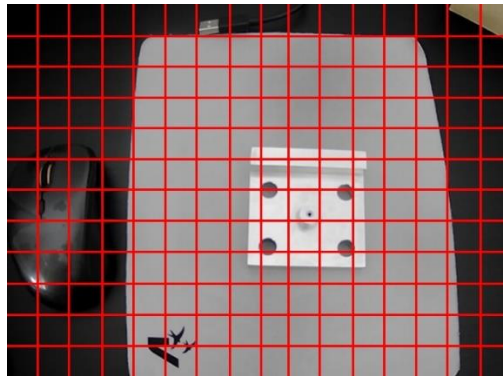


Figure 21: Division of the image in cells done by the YOLO model

The bounding box is a rectangular shape box that detect an object in a given image. Each bounding box is characterized by a width, a height, a class, by a bounding box centre and

by the probability to detect the object in the bounding box. An example of a bounding box is shown in *Figure 22* considering the input image in *Figure 21*: h indicates the height of the bounding box, w indicates the width, x and y indicate the centre of the bounding box and c indicates the class of the object that appears in the bounding box. Each cell is responsible for predicting a bounding box if the center of it is inside that particular cell. The prediction of the object detected, and its class is done by involving the center, the width, height and the confidence or probability [28].

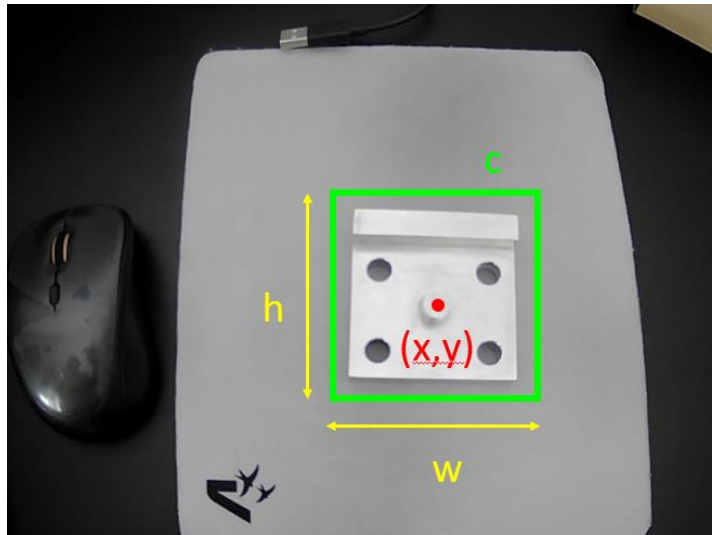


Figure 22: Bounding box of the object detected

The Intersection over Union (IoU) is a number that represents a metric used to evaluate and measure the accuracy of a model that must detect objects. Since any algorithm that provides as output bounding boxes can be evaluated using the IoU, this evaluation metric is also used in the YOLO model. To measure the IoU of a model, two bounding boxes must be defined: the real bounding boxes and the predicted bounding boxes. The real bounding boxes specify where the objects are actually, and they correspond exactly to the bounding boxes that we have specified before the training in order to label the objects in the images. In the other hand, instead, the predicted bounding boxes are the outputs provided by the YOLO model and they are not necessarily to the real ones, because the model has always

some errors. Indeed, the IoU is a measurement of how the real bounding box and the predicted bounding boxes overlap: if a real bounding box is exactly equal to the predicted one, the IoU is equal to 1 and is the maximum possible number of the Intersection over Union [35][36][44]. As shown in *Figure 23*, the real bounding box is drawn in green and the predicted one is drawn in red, and they do not coincide generally. The evaluation metric is computed by using the formula shown in *Figure 24* [44].

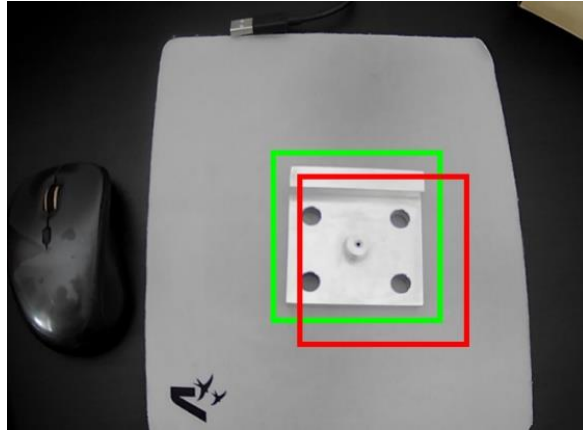


Figure 23: Real bounding box vs Predicted bounding box

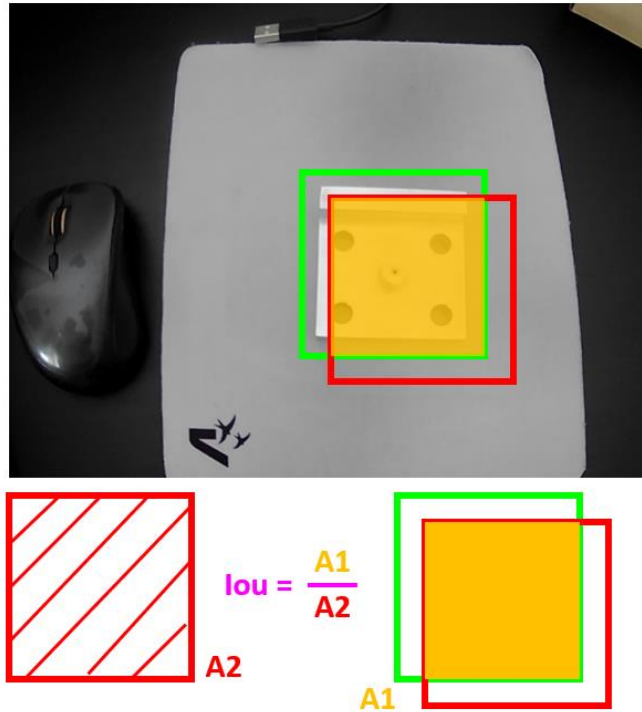


Figure 24: Intersection over Union computation

Therefore, the Intersection over Union is a ratio between the area of overlap between the real and the predicted bounding box by the area of union of them. Is considered a good prediction when the IoU number is greater than 0.5. In the *Figure 25(a)* and *figure 25(b)* are shown some examples and evaluation metrics.

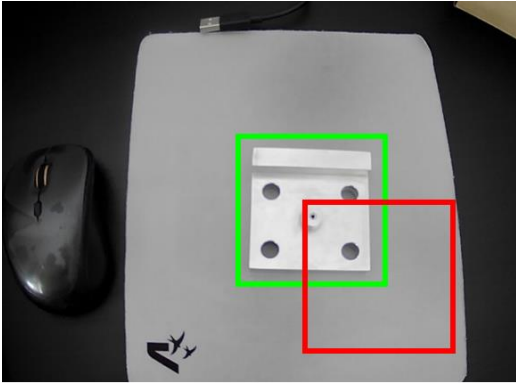


Figure 25 (a): $IoU < 0.5$

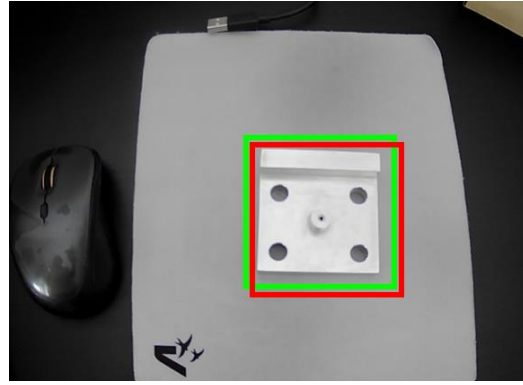


Figure 25 (b): $IoU > 0.5$

As a result, it is possible to say that higher is the overlapped area, higher is the IoU and better will be the prediction by the YOLO model. Combining the three methods we can have as output the final detection. As a result, to recap the three techniques used in the YOLO, the prediction of the bounding boxes in each grid cell is done using which the class probabilities are found out in each grid cell in order to set the class of each object. As shown in the example in *Figure 26*, there are three classes of objects, a dog a bicycle and a car. YOLO uses a single convolutional neural network to make all the predictions simultaneously and through the IoU, it eliminates any other unnecessary bounding boxes that do not match the dimensions of the height and the width of objects. The final detection is made such that the bounding boxes detect the objects accurately as shown in *Figure 26* [28]. As we can see later, the version of the YOLO model used in the thesis is the YOLO version 5 since it was the latest version released when the thesis study was started, and it was the best in terms of performance.

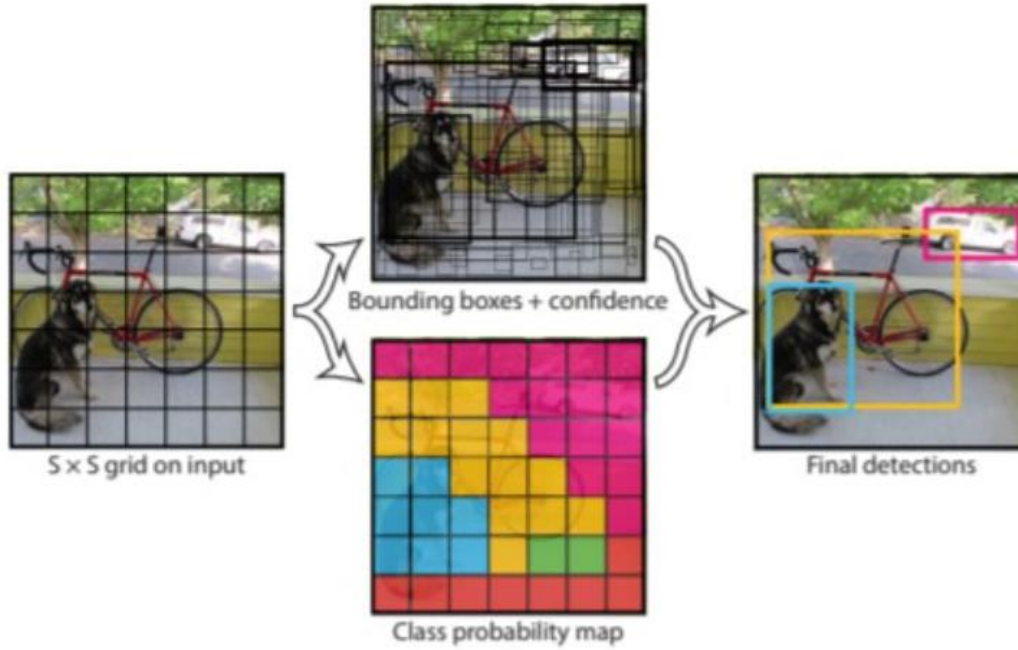


Figure 26: Techniques used by the YOLO model [28]

3.7 Features and mechanical pieces

In a general assembly operation, the mechanical pieces involved seem similar to each other. For the purpose of the thesis, the cobot must recognize a piece as different from another one by considering as difference at least one technological feature that differentiate them. For this reason, they have been chosen different pieces that seems similar to each other but are different in at least one feature. As shown in *Figure 27*, two steel piasters are chosen: they have the same dimension, the same shape, the same number of holes, they are both in steel even if they reflect the light in different way. The algorithm has the goal to recognize these two piasters as the same piaster without making any difference.

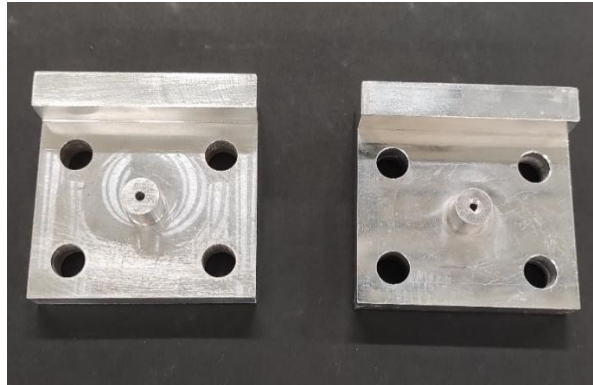


Figure 27: Steel piasters with different surface

Moreover, it has been chosen also a yellow piaster (*Figure 28*) that has the same shape, same dimension, same number of holes of the steel piasters but it has a different material, indeed this piaster is made of yellow plastic. The developed algorithm must recognize this piaster as different with respect the steel ones, taking into account the material as a different feature between them.



Figure 28: Piasters with different materials

Another piaster it has been chosen (*Figure 29*) and in this case the shape and the number of holes is always the same, but it has a different dimension even if the material is the same of the yellow one. The algorithm indeed, must recognize this piaster as different with respect the others considering as features the dimension and the material. To have pieces that have different shapes, some flanges they have been chosen (*Figure 30*).

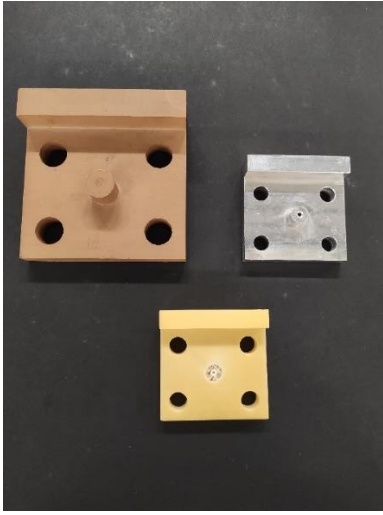


Figure 29: *Piasters with different material and dimension*

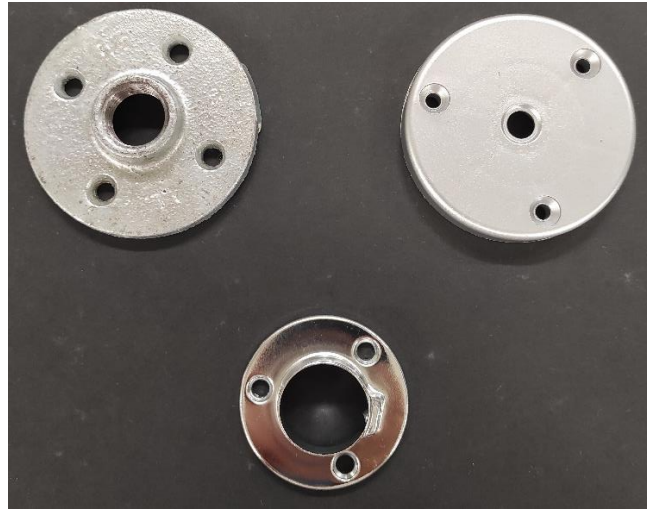


Figure 30: *Flanges with different number of circumferential holes*

Analyzing the *Figure 30*, the top left flange has the same dimension and shape of the top right one and both have the hole in the center of the piece. The only difference between the two flanges is the number of the circumferential holes, indeed the top left flange has 4 circumferential holes, and the top right flange has 3 ones. The algorithm anyway must recognize the two pieces in the top as different pieces taking into account the number of circumferential holes as a different feature between them. Instead, the bottom flange has the same number of holes and shape of the top right one, however it has a different dimension. The developed algorithm must recognize this last flange as different with respect the top ones considering as different features the number of circumferential holes and the dimension. Another flange it has been chosen (*Figure 31*): this flange has no circumferential holes even if it has a hole in the center, the same shape, same dimension and same material of the small flange with three circumferential holes. In this case the algorithm must distinguish this flange as different with respect the previous ones. Moreover, the algorithm must simply recognize these flanges as different with respect the piasters considering as different feature the shape. The last piece chosen is a flange with a different shape with respect the previous ones (*Figure 32*), moreover it has only two circumferential holes even if it has a hole in the center as happens in all flanges that are

chosen. As a result, the features that the algorithms must consider in order to recognize a mechanical piece as different with respect the other, are the dimension, the shape, the material and the number of circumferential holes.



Figure 31: *Flange with same dimension but different number of holes*



Figure 32: *Flange with 2 circumferential hole and some cuts in the lateral surface*

3.8 Python and libraries used

The goal of the thesis, as said several times, is to recognize real-life objects and this can be reached by using the library OpenCV since it supports a lot of algorithms that are related to Computer Vision and Machine Learning. The term OpenCV stands for Open-Source Computer Vision. It is a free computer vision library that allows to reach different tasks as image recognition, object localization and object detection. With this library is possible to reading, writing and manipulate images and videos by doing some operations on that. Moreover, it is possible to display videos from webcams, manipulating them to reach different goals. Even if OpenCV supports different programming language like C++ and Java, to make the most of the potential of the OpenCV library, it is used Python as programming language, even because Python is more suited to the tasks accomplished in this thesis [28]. Python is an interpreted, object oriented, high-level programming language with dynamic semantics and very powerful features. It supports modules and

packages that makes possible the program modularity and the reusing of the code [33]. Another library used in the thesis is named NumPy. It is a library that depends on the Python bindings of OpenCV used for scientific and mathematical computing. With this library is possible to do numerical computation with functionalities that are provided by NumPy based on linear algebra and matrices. It is also possible doing operations on large multi-dimensional arrays [28][32]. The fact that Python supports NumPy, makes the goals of the thesis to be reached, easier. Indeed, NumPy is a highly optimized library for numerical operations and its syntax is like the MATLAB one. Therefore, the OpenCV array structures associated to images and videos are converted to and from the NumPy arrays, in this way the numerical operations with arrays become easier because any operations done in NumPy, can be translated in OpenCV to reach a large variety of goals [29]. In the thesis is used the current version 2.3.1 of OpenCV. Even if there are two Python interfaces that can be used with this version of OpenCV, in the thesis is used the new module of OpenCV called “cv2”. The reason is that the old module “cv” uses internal OpenCV datatypes that can be difficult to use from NumPy. With “cv2” instead, it is possible to use NumPy arrays and is much more intuitive to use with respect to the old one [30]. Since in the thesis it must be detect objects using points and lines, another library used for developing the algorithm is Matplotlib. It is an open-source graphic library used for plotting graphs, drawing points, lines, curves and images with a very high quality [30]. Another advantage to use NumPy is the fact that it has a vast ecosystem to support libraries as Matplotlib and OpenCV. The last library used in the thesis is called PyTorch. It is an open-source machine learning library used for developing and training neural networks and deep learning models. PyTorch can also be used with different programming languages as Python, C++ and Java but the Python interface is more sophisticated. PyTorch uses dynamic computation that allows flexibility in building and establishing complex architectures. PyTorch uses classes, structures and loops that makes it simpler with respect other frameworks as TensorFlow [31]. PyTorch is based on Torch library and the core package of Torch is called “torch”. Moreover, PyTorch interoperates with NumPy to control the tools and libraries of NumPy in order to extend and improve its capabilities.

4. Code to train the YoloV5 model

4.1 YoloV5 architectures

There are different versions of the YOLO model. The YOLO model to detect the mechanical pieces used in the thesis is the YOLO version 5 (YOLOv5). It is written in the Pytorch framework and its implementation is developed by the Ultralytics and the first official version was released in the June of the 2020. The architecture of the YoloV5 is similar to the YoloV4 with some improvements since the YoloV5 is faster and more accurate than the YoloV4 [18]. There are several architectures of YOLOv5 architecture as seen in *Figure 33*.



Figure 33: Different models of the YOLOv5 [16]

The YoloV5n (nano) is the simplest one, instead the YoloV5x (extra-large) is the most complicated one. The choice of one of these architectures depends on the application. If the architecture is more complicated, it means that the outputs given by the model will be better, but the speed of the training and the speed needed to detect an object in real time decreases. In the purpose of the thesis, we need a model fast with good results and since the YoloV5n (nano) gives the worst result in terms of the mAP computed in the validation set images as shown in *Figure 34*, this architecture was excluded a priori. To understand what the mAP is we have to understand first what the AP is. The AP is the average precision, and the value of the AP goes from 0 to 1. It summarizes the precision–recall curve by averaging precision across different recall values and it represents the area under the precision– recall curve (details in **5.1 Figure 68**). Since in the thesis we consider

different classes, there is an AP value for each class and it has no sense take only one AP value of one class to evaluate the performance of the model. So, the mean average precision (mAP) is used and can be computed as follows:

$$mAP = \frac{1}{N} \sum_{i=1}^N AP_i$$

Where the N represents the number of classes considered and the AP_i is the average precision of the single class. The mAP has different forms of calculation. One of these is to do the computation of the AP considering as IOU threshold a value equals to 0.5 and the area under the precision-recall curve is measured by using all the points. This type of AP is called AP 0.5. If instead we consider the mean average precision, it is called the mAP 0.5. There is another form of calculation called AP 0.5:0.95. In this case the average precision is computed by considering different IOU thresholds, from 0.5 to 0.95 considering as the difference between a following IOU and the previous one equal to 0.05. In this case we have as IOU thresholds the values equal to 0.5, 0.55, 0.6, 0.65 and so on. The result is the average of the Aps considering the previous IOU thresholds [35][36]. If we consider instead the mAP, the parameter is called mAP 0.5:0.95. In the *Figure 35* are shown all the remained models with the maximum size of images that can be detected, the mAP 0.5:0.95 computed on the validation set images, the mAP 0.5:0.95 computed in the test set images, the mAP 0.5 computed in the validation set images and the speed of the model. Moreover, in the *Figure 35* is shown a graph of the mAP 0.5:0.95 computed in the validation set images as a function of the speed [45]. It is possible to see from the table and from the graph that the values of the yolov5s are not so far from the values of the yolov5x, but in the other hand the yolov5s is five times faster with respect to the yolov5x and for this reason in the thesis it is used the yolov5s.

<u>Model</u>	<u>size</u> <u>(pixels)</u>	<u>mAP_{val}</u> <u>0.5:0.95</u>	<u>mAP_{test}</u> <u>0.5:0.95</u>	<u>mAP_{val}</u> <u>0.5</u>	<u>Speed</u> <u>V100 (ms)</u>
YOLOv5s6	1280	43.3	43.3	61.9	4.3
YOLOv5m6	1280	50.5	50.5	68.7	8.4
YOLOv5l6	1280	53.4	53.4	71.1	12.3
YOLOv5x6	1280	54.4	54.4	72.0	22.4

Figure 34: mAP values for different YOLOv5 models and for different set of images [18]

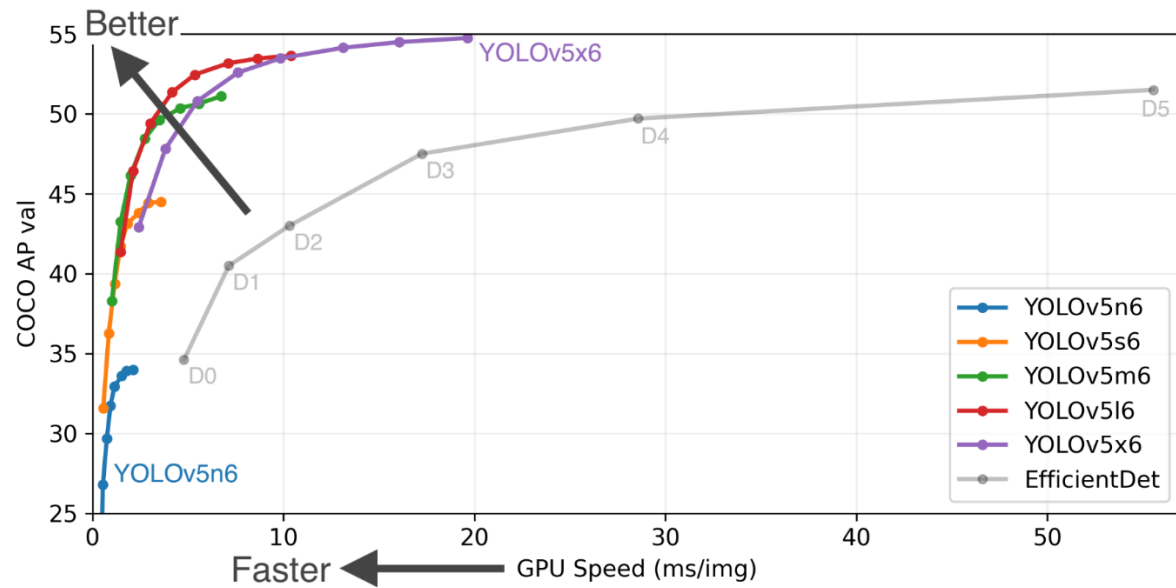


Figure 35: graph of the mAP 0.5:0.95 computed in the validation set images [18]

The whole architecture of the Yolov5 is shown in *Figure 36*.

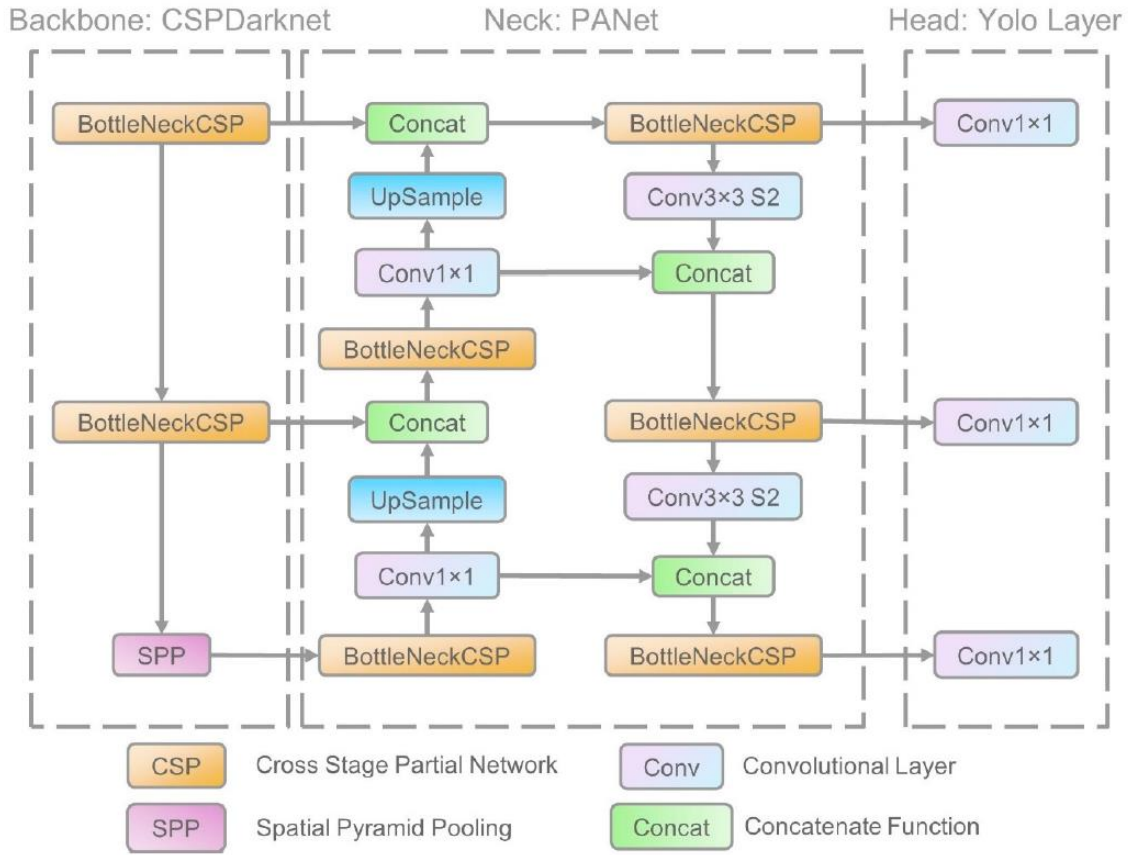


Figure 36: Architecture of the YOLOv5 model [19][43]

As it is possible to see, it is composed by three main blocks: the backbone, the neck, and the head. The model backbone is used to extract the key features from the input image. In the yolov5 is used as a backbone the Cross Stage Partial Networks (CSP) to extract the most useful characteristics from the input image [19][43]. The model neck is used to create feature pyramids. These are important to detect the object with different sizes and scales, since it happens that there are two same objects in an image but with different sizes, and the model must identify these objects as the same. The FPN, BiFPN, and PANet, use various sorts of feature pyramid approaches. The model head is responsible as most, for the final detection step. This model constructs the final output vectors with class

probabilities, objectness score, bounding boxes and other metrics and results specified later [19][43].

4.2 Install and import the dependencies

As said, the model of yolov5 is a great implementation for object detection that use PyTorch. First of all, it has been installed PyTorch by the following command [45][47]:

```
!pip install torch torchvision torchaudio --extra-index-url...
...https://download.pytorch.org/whl/cu113
```

Then, it has been cloned the repository of yolov5 from GitHub and it has been used the following command [46]:

```
!git clone https://github.com/ultralytics/yolov5
```

The above command allows to create a folder called “yolov5” and copy on it all the files and folders shown in *Figure 37*. In this way all files used to train the yolov5 model are copied in the workstation from which the model will be trained.























	.github	4/30/2022 2:09 PM	File folder	
	__pycache__	4/30/2022 4:47 PM	File folder	
	data	4/30/2022 2:09 PM	File folder	
	models	4/30/2022 4:47 PM	File folder	
	runs	4/30/2022 4:48 PM	File folder	
	utils	4/30/2022 4:47 PM	File folder	
	.gitattributes	4/30/2022 2:09 PM	Text Document	1 KB
	.gitignore	4/30/2022 2:09 PM	Text Document	5 KB
	.pre-commit-config.yaml	4/30/2022 2:09 PM	YAML File	2 KB
	CONTRIBUTING.md	4/30/2022 2:09 PM	MD File	5 KB
	dataset.yaml	5/2/2022 3:08 PM	YAML File	1 KB
	detect	4/30/2022 2:09 PM	Python File	14 KB
	export	4/30/2022 2:09 PM	Python File	30 KB
	hubconf	4/30/2022 2:09 PM	Python File	7 KB
	LICENSE	4/30/2022 2:09 PM	File	35 KB
	README.md	4/30/2022 2:09 PM	MD File	16 KB
	requirements	4/30/2022 2:09 PM	Text Document	1 KB
	setup.cfg	4/30/2022 2:09 PM	CFG File	2 KB
	train	4/30/2022 2:09 PM	Python File	35 KB
	tutorial.ipynb	4/30/2022 2:09 PM	IPYNB File	57 KB
	val	4/30/2022 2:09 PM	Python File	20 KB
	yolov5s.pt	4/30/2022 4:48 PM	PT File	14,462 KB

Figure 37: Folders inside the yolov5 folder

Moreover, yolov5 needs some dependencies, libraries and modules that allow its implementation. To install all the requirements [45] needed, the command below has been used:

```
!cd yolov5 & pip install -r requirements.txt
```

The above command allows to enter in the folder “yolov5” and install all the requirements written in the txt file “requirements.txt”. This file contains specific versions of the modules needed for the yolov5 implementation that must be installed. In order to train the model, after the dependencies’ installation, it is necessary to import them in the notebook from which the yolov5 model is trained. Therefore, the PyTorch, Matplotlib, NumPy and OpenCV libraries must be imported using the following commands [55]:

```
import torch
from matplotlib import pyplot as plt
import numpy as np
import cv2
```

Moreover, in order to collect images from the webcam the following libraries are also imported using the commands [55]:

```
import uuid
import os
import time
```

Then, they have been created three folders: the folder “data” (*Figure 38*) contains both the “images” and the “labels” folders (*Figure 39*). In the “images” folder we have collected all the images used to train the model and instead in the “labels” are collected the files created after labelled the images.

```
IMAGES_PATH= os.path.join('data', 'images')
labels=['hole', 'p1', 'p2', 'p3', 'p4']
```



9/16/2022 4:48 PM

File folder

Figure 38: Folder “data”

images
labels

4/30/2022 2:11 PM
5/17/2022 5:47 PM

File folder
File folder

Figure 39: Folder “images” and “labels”

Examples of the images collected of the flanges are shown in *Figure 40*.



Figure 40 (a)



Figure 40 (b)



Figure 40 (c)



Figure 40 (d)



Figure 40 (e)



Figure 40 (f)



Figure 40 (g)



Figure 40 (h)

It has been used the same approach [45][46][47] also to collect the images of the piasters and the images detected are shown in *Figure 41*.

```
!pip install torch torchvision torchaudio --extra-index-url...  
...https://download.pytorch.org/whl/cu113
```

```
!git clone https://github.com/ultralytics/yolov5
```

```
!cd yolov5 & pip install -r requirements.txt
```

```
import torch  
from matplotlib import pyplot as plt  
import numpy as np  
import cv2  
import uuid  
import os  
import time
```

```
IMAGES_PATH= os.path.join('data', 'images')  
labels=['piastra']
```



Figure 41 (a)



Figure 41 (b)

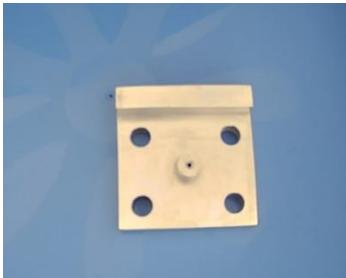


Figure 41 (c)



Figure 41 (d)

4.3 Capture images

In order to obtain the images that represent the input given to the model, we have used a code to obtain the images directly from the external camera. In this way the images are the frames captured by the camera at each amount of time. First of all, we have defined the total number of images that must be captured and is equal to 150. The webcam connected to the workstation is activated with the function `cv2.VideoCapture(0)` [55]:

```
total_number_of_images=150
cap=cv2.VideoCapture(0)
```

Then, we have defined a for cycle in which we collect one frame from the webcam at each loop:

```
# Loop for each image collected until the total number of images is equal to 150
for image_number in range(total_number_of_images):
```

Then, at each time is obtained a frame from the webcam with the function `cap.read()` and is saved in the variable *frame*:

```
# obtain the frame from the webcam
ret,frame=cap.read()
```

Then, we have defined the path in which we save the images that corresponds to the folder “images” that is contained in the folder “data” previously created [55]:

```
# defining the path where the frames captured by the camera are saved
IMAGES_PATH= os.path.join('data', 'images')

# save the image in the path defined
img_name= os.path.join(IMAGES_PATH, str(uuid.uuid1())+'.jpg')
cv2.imwrite(img_name,frame)
```

To show the image collected we have used the function `cv2.imshow()` [55]:

```
# show the frame captured
cv2.imshow('Image Collection', frame)
```

Since it is needed time to move manually the objects in order to have frames with different positions and rotations of the same object since we want to obtain a trained model with a high robustness, we have defined the time that occurs between two captures:

```
# 4 seconds delay between captures
time.sleep(4)
```

The last three commands shown below means that until we do not press the key ‘q’ in the keyboard, the algorithm will run until we obtain a number of frames equals to 150. When instead we press the key ‘q’ to quit from the program, the webcam will be deactivated with the function `cap.release()` and the window that shows the detection in real time will be also closed using the command `cv2.destroyAllWindows()` [55]:

```
# if we want to end the process before the number of frames collected is equal to 150
if cv2.waitKey(10) & 0xFF == ord('q'):
    break

cap.release()
cv2.destroyAllWindows()
```

By running the code written in 4.3 the webcam will capture a new frame each 4 seconds and it will save in the folder that we have set.

4.4 Label the images

In the thesis is used the software `labelImg` to label images for the object detection. To use that, it is necessary to clone its repository from GitHub through the command [48]:

```
! git clone https://github.com/tzutalin/labelImg
```

The above command copies all the files and folders shown in *Figure 42* that are needed to correctly use the software `labelImg`. These files are copied in the workstation in a folder called “`labelImg`”.

.github	4/30/2022 2:13 PM	File folder	
build-tools	4/30/2022 2:13 PM	File folder	
data	4/30/2022 2:13 PM	File folder	
demo	4/30/2022 2:13 PM	File folder	
libs	4/30/2022 2:13 PM	File folder	
readme	4/30/2022 2:13 PM	File folder	
requirements	4/30/2022 2:13 PM	File folder	
resources	4/30/2022 2:13 PM	File folder	
tests	4/30/2022 2:13 PM	File folder	
tools	4/30/2022 2:13 PM	File folder	
.gitignore	4/30/2022 2:13 PM	Text Document	1 KB
.travis.yml	4/30/2022 2:13 PM	YML File	2 KB
__init__	4/30/2022 2:13 PM	Python File	0 KB
CONTRIBUTING.rst	4/30/2022 2:13 PM	RST File	1 KB
HISTORY.rst	4/30/2022 2:13 PM	RST File	2 KB
issue_template.md	4/30/2022 2:13 PM	MD File	1 KB
labellmg	4/30/2022 2:13 PM	Python File	69 KB
LICENSE	4/30/2022 2:13 PM	File	2 KB
Makefile	4/30/2022 2:13 PM	File	1 KB
MANIFEST.in	4/30/2022 2:13 PM	IN File	1 KB
README.rst	4/30/2022 2:13 PM	RST File	11 KB
resources.qrc	4/30/2022 2:13 PM	QRC File	3 KB
setup.cfg	4/30/2022 2:13 PM	CFG File	1 KB
setup	4/30/2022 2:13 PM	Python File	4 KB

Figure 42: folders inside the “labellmg” folder

Moreover, to use labellmg, it must be installed “pyqt5” that is a GUI library and “lxml” that is a dependency of pyqt5. This is done using the command below [55]:

```
! pip install pyqt5 lxml --upgrade
```

To run the labellmg software the following command must be used [55]:

```
!cd labellmg && pyrcc5 -o libs/resources.py resources.qrc
```

The above command (*Figure 43*) allows to go inside the folder labellmg and run the file “pyrcc5”, passing also through the folder “libs” in order to run the files “resources.py” and “resources.qrc”. In this way the labellmg software is run correctly, indeed a window will appear (*Figure 44*).

```
\labellmg>python labellmg.py
```

Figure 43: Command to run labellmg program

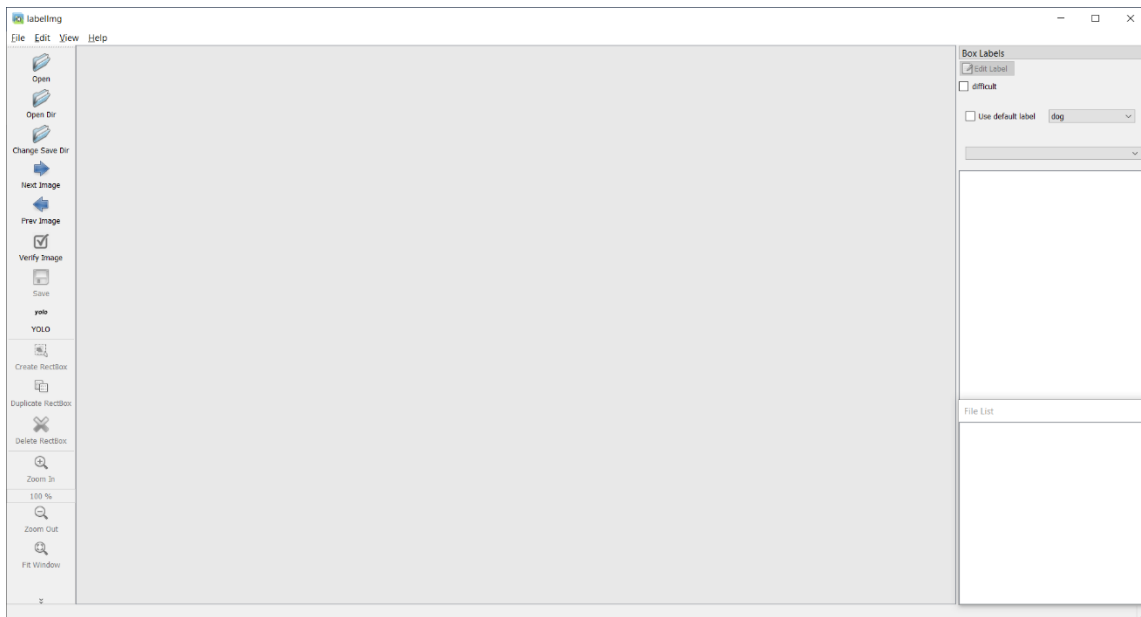


Figure 44: Window of the labeling program

To label the images, first we have to open the directory in which the images are saved, and this is possible thanks to the command shown in *Figure 45*. At the same time, we have to choose the directory in which the labeled images must be saved, and this has been done thanks to the command shown in *Figure 46*.

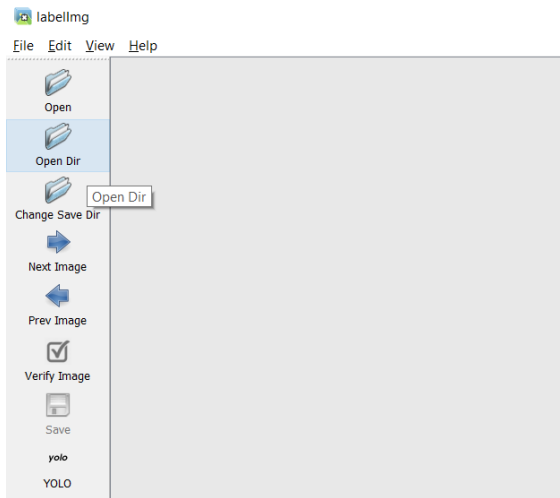


Figure 45: Command to choose the directory that must be opened

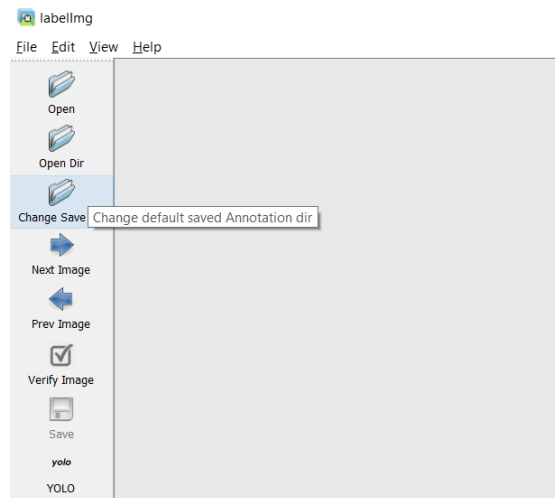


Figure 46: Command to choose the directory where the images labeled must be saved

In order to correctly label the images in the Yolo format, we have changed the format into “yolo” as shown in *Figure 47*.

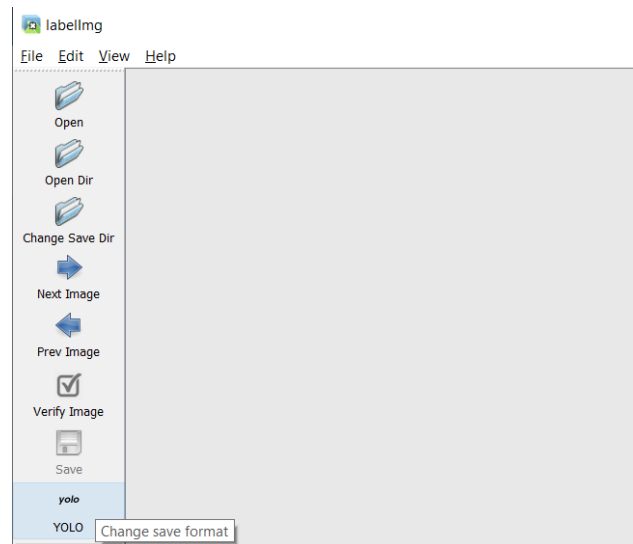


Figure 47: Command to change the format in “yolo” format

After opened the directory, all the images will appear, one by one (*Figure 48*) and then it has started the labeling process.

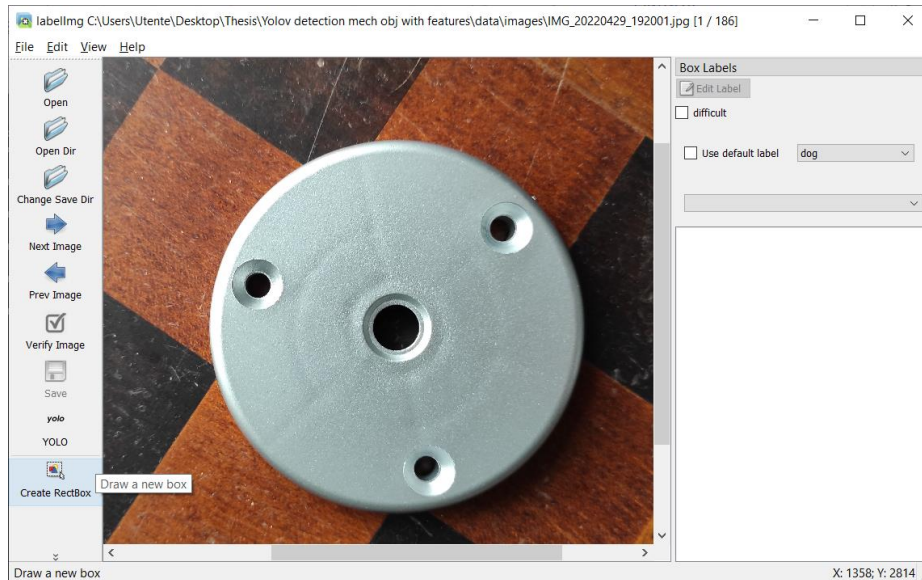


Figure 48: Image opened in the labellmg program

The first step was to create the rectangular box around the piece that the model must recognize (*Figure 49*).



Figure 49: Create the bounding box around the object to be detected

After created the rectangular box, another window will be appeared, in this case it is possible to specify the name of the label detected (*Figure 50*). The result of the labeling is shown in *Figure 51*.

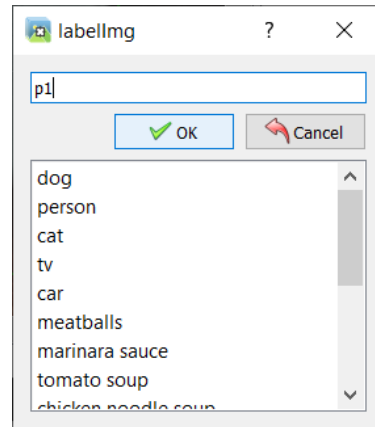


Figure 50: Name assigned to the object that must be detected

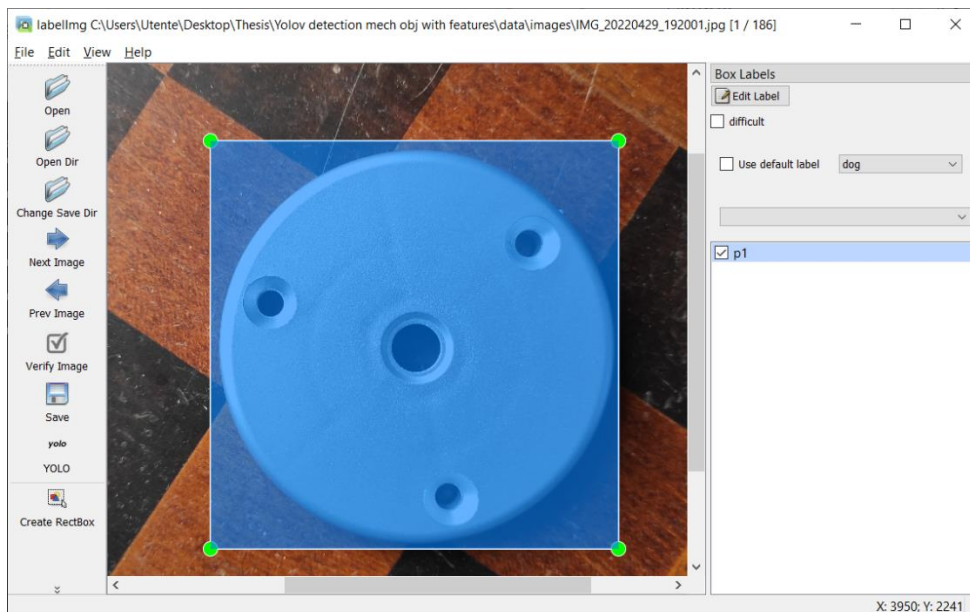


Figure 51: Results of labeling an object

The same process is repeated also for the other classes as holes as shown in *Figure 52* and for the other pieces as shown in *Figure 53*, *Figure 54*, and *Figure 55*.

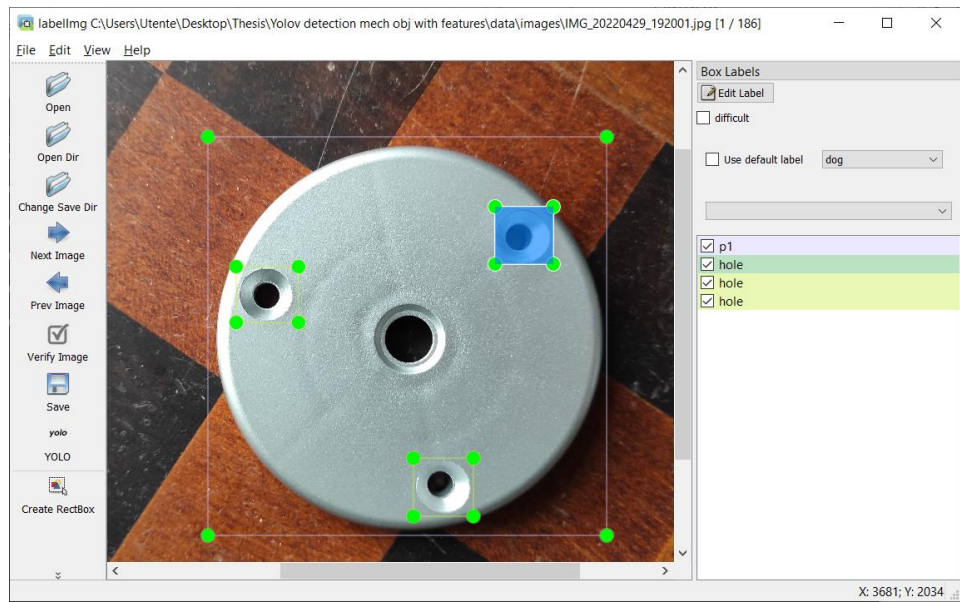


Figure 52: Labels of holes inside a flange

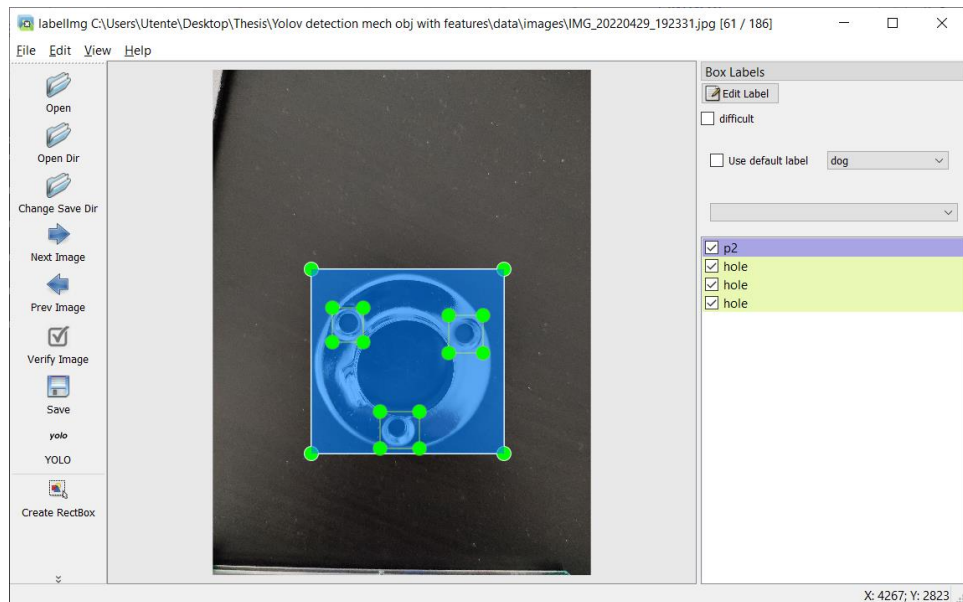


Figure 53: labels related to the flange with 3 circumferential holes

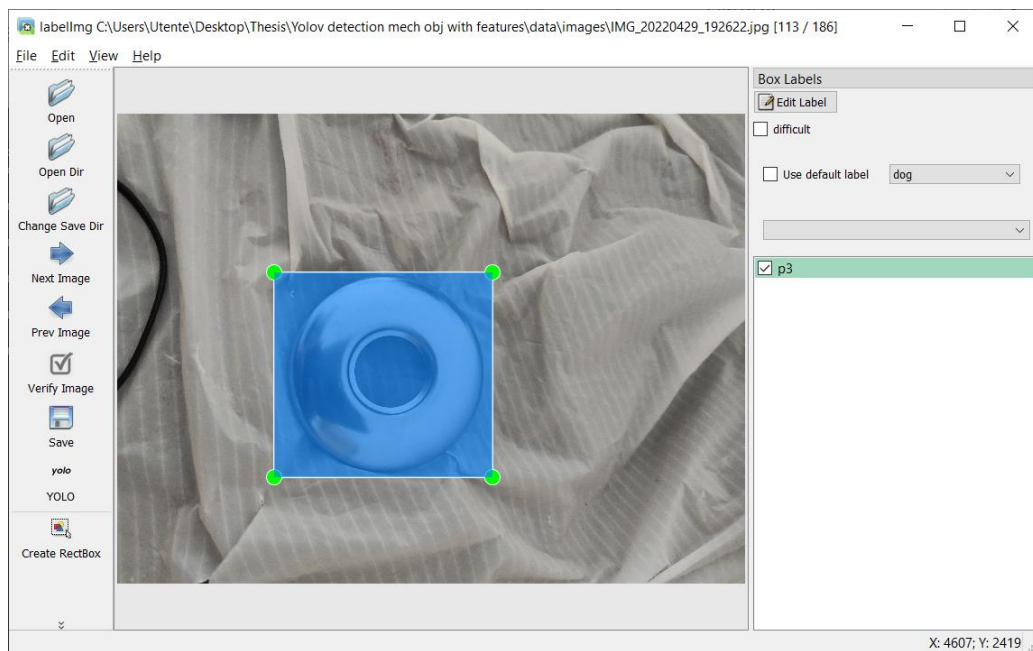


Figure 54: label related to the flange that has not any circumferential holes

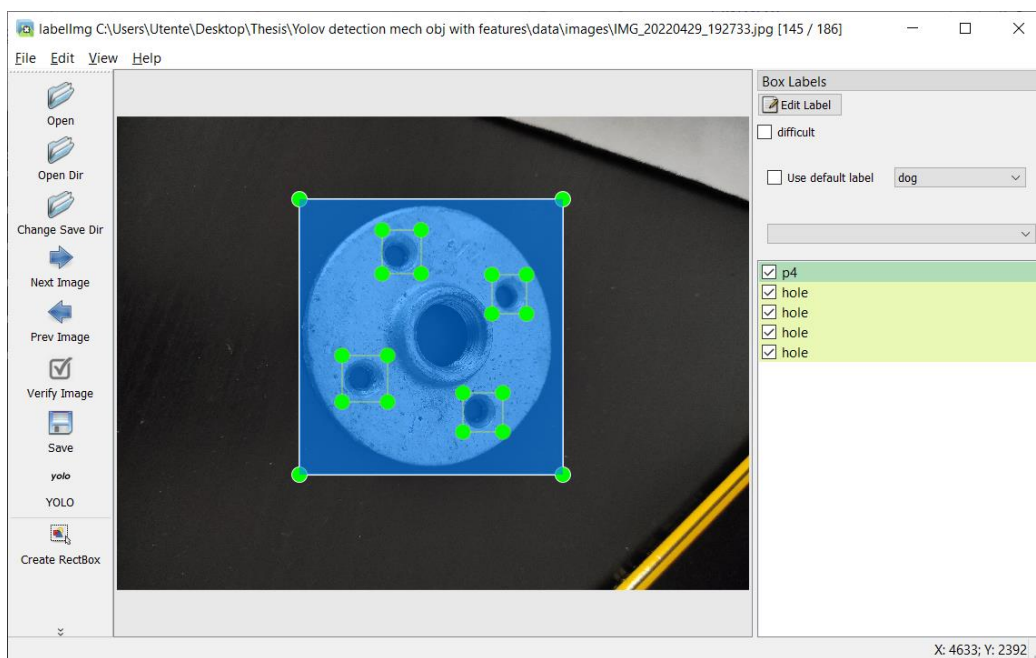


Figure 55: labels related to the flange with 4 circumferential holes

4.5 Training command

```
!cd yolov5 && python train.py --img 320 --batch 16 --epochs 140 --data dataset.yaml...  
...-weights yolov5s.pt --workers 2
```

The command above [55] allows to train effectively the model. First of all, it goes in the yolov5 folder and run the train.py Python Script. Then we have to specify the “img” value that represents the resolution of the objects that must be detected in the images that are the input of the trained model. If there are many small objects to detect in an image, it is convenient to use high resolution as 1280. In our case is sufficient to consider 320 as a resolution. Then it is inserted the number of the batches and the number of epochs. To understand what the batches and the epochs are, we must start to explain what the samples are. The sample, also called instance, is a single row of data. The samples are the input of the algorithm. A training dataset is composed by many samples. Therefore, it is possible to define what a batch is. The batch defines the number of samples that are used before updating the internal parameters of the model. It can be seen as a loop where we use some samples to make the prediction and to train the model. At the end of the batch, the predictions are compared to the expected output of the model. From this comparison it is possible to calculate the errors that are used to update the parameters and improve the model. A high value obviously guarantees a better model with lower prediction errors, but at the same time if we raise the batch value too much, the time to train the model increases exponentially. Furthermore, beyond a certain value it is not possible to greatly increase the quality of the model and decrease the errors. For this reason, it is considered a number of batches equal to 16. This means that the samples are divided in 16 groups. As a result, our training dataset is divided into 16 batches. The number of epochs is different from the number of the batches, since the number of epochs is the number of times that the algorithm uses the entire training dataset. One epoch represents the possibility from a training dataset to update the internal parameters of the model. Generally, the number of epochs is quite large as hundreds. As we said for the number of batches, also in this case a high value guarantees parameters that give as a result a better model. If we increase too much the value of the epochs, the training time increases exponentially too even if below

a certain value it is not possible to greatly increase the performance of the model. Therefore, each epoch represents a loop where are proceeded all the training dataset. In this loop the model is trained over each batch and after each batch the model parameters are updated. The number of these repeated loops represents the number of epochs. As a result, we can consider the number of epochs and the number of batches as follows: the number of the batches is the number of groups in which the samples are divided and each group of samples represents the number of samples processed before the parameters model are updated in each epoch (in each loop), instead, the number of epochs is the number of complete passes (number of loops) through the training samples [37][42]. The dataset.yaml give us a configuration of the training run. The dataset.yaml has been created and it contains the path where is possible to see the images used for training and the images used for the validation. This is because when we train e neural network model, the images are divided into train images and validation images. The paths are defined as follows [37]:

```
path: ../data      # dataset directory
train: images      # train images directory
val: val_images    # validation images directory
```

Then it has been defined the number of classes considered. In this case the model considered is the model used to detect the holes and the flanges. The number of classes is 20 and the classes of interested are the last 5. As said 15 classes are defined by default. The classes and the name of each classes are defined in the dataset.yaml in the following way [37]:

```
# Classes
nc: 20 # number of classes
names: [ 'dog', 'person', 'cat', 'tv', 'car', 'meatballs', 'marinara sauce', 'tomato soup',
'chicken noodle soup', 'french onion soup', 'chicken breast', 'ribs', 'pulled pork',
'hamburger', 'cavity', 'hole', 'p1', 'p2', 'p3', 'p4' ] # class names
```

Then, it is specified the type of the model that we want to train, and, in our case, it is the YOLOv5small as said previously. After waiting some time, we get the output of the training [37]. As shown below, it has been used the same command also to train the piasters. In the case of the piasters the number of epochs is higher since it is more difficult to obtain a

model with a low number of epochs that detects the piasters also when they are rotated. This problem did not appear in the case of the flanges since the flanges are round objects.

```
!cd yolov5 && python train.py --img 320 --batch 16 --epochs 400 -data...  
...dataset.yaml --weights yolov5s.pt --workers 2
```

5. Outputs and results of the trained model

5.1 Results after the training

The output of the training gives different results. First of all, in the folders \yolov5\runs\train\exp4\weights (Figure 56) it is possible to find the file best.pt [37].

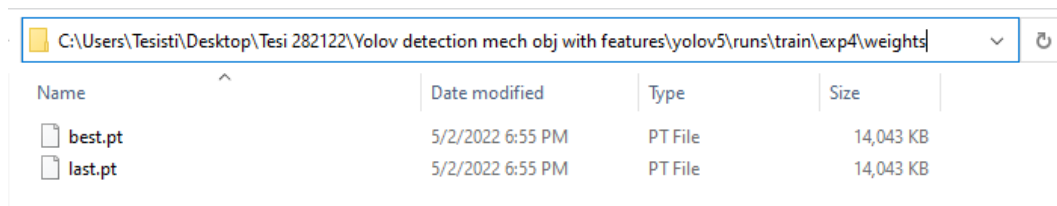


Figure 56: file best.pt

This contains the final model for final Detection and Classification. Instead in the folders \yolov5\runs\train\exp4 it is possible to find the file results [37] (Figure 57).

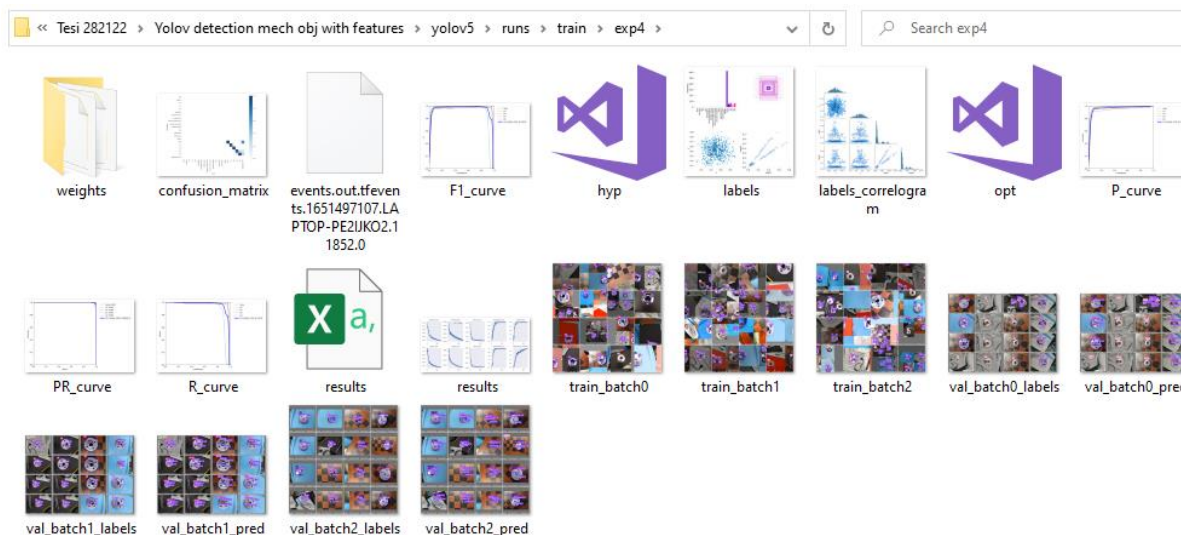


Figure 57: Results obtained after the training

It contains the summary of accuracy and losses achieved at each epoch that are analysed later. Moreover, as shown in the *Figure 57*, it is possible to see graphs and plots that are explained later [37]. In order to understand the validity and the prevision quality of the classification model trained, it is used the confusion matrix. It is an important tool in machine learning since it is useful to analyze the mistakes made by the model after its training. To understand what a confusion matrix is and how can be used, it is made a simple example of a binary classification. This means that there are two possible classes (A and B), and the model must classify a sample, also called an instance, as class A or class B. The class A is considered as the main and positive class, instead the class B is considered as a negation and an alternative of the class A, for this reason the class B is the negative class. In this binary classification case, the confusion matrix is a 2x2 matrix [34]. As shown in *Figure 58*, there are the predicted classes in the columns of the matrix and the actual classes in the rows of the matrix.

		PREDICTED CLASSES	
		Class A	Class B
ACTUAL CLASSES	Class A		
	Class B		

Figure 58: Actual classes and predicted classes

The predicted classes are the output of the model trained, instead the actual classes are the classes of the correct answers. In the example shown in *Figure 59*, the model analyzes a certain number of samples, for instance 100 samples, and classifies them as class A or class B. In 80 cases, the model classifies correctly the sample, but the other 20 samples are classified incorrectly.

		PREDICTED CLASSES	
		Class A	Class B
ACTUAL CLASSES	Class A	50	5
	Class B	15	30

Figure 59: Example of a confusion matrix

In this example there are four possible situations that can be verified. The True Positive (TP) are the samples or instances that the model classifies as positive class (in this example is the class A), the True Negative (TN) are the samples that the model classifies correctly in the negative class (in this example is the class B). The summation of the TP and the TN is 80, and as said, the model correctly classifies the samples in 80 cases. There are two other situations that can happen: the False Positive FP are the samples that the model classifies as positive while they are negative in reality, the False Negative (FN), instead, are the instances that the model classifies as negative, while they are positive in reality. The summation of the FP and the FN is 20, this means that the model is wrong for 20 cases as previously said (*Figure 60*).

		PREDICTED CLASSES	
		Class A	Class B
ACTUAL CLASSES	Class A	50 = TP	5 = FN
	Class B	15 = FP	30 = TN

Figure 60: TP, TN, FP and FN in a confusion matrix

The confusion matrix can also be constructed by considering even more classes as happens in the case study of the thesis (*Figure 61*).

		PREDICTED CLASSES			
		Class A	Class B	Class C	Class D
ACTUAL CLASSES	Class A				
	Class B				
	Class C				
	Class D				

Figure 61: Confusion matrix with more classes

It is analyzed now the confusion matrix shown in *Figure 62* deriving by the model that must recognize the flanges and the holes. This confusion matrix considers the TP, FP, TN and FN as percentage.

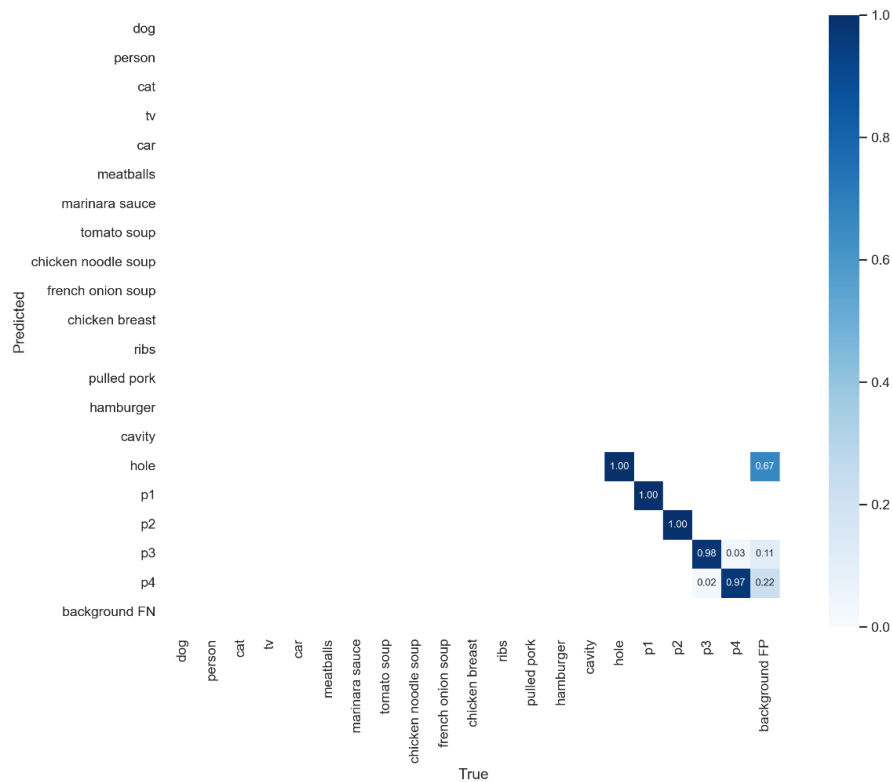


Figure 61: Confusion Matrix of the model flanges and holes

As it is possible to see, the True Positive of the holes, of the first two flanges (p1 and p2) are 100%. This means that the model does not make any mistake regarding these three classes. Instead, the data regarding the recognition of the third flange (p3) are divided in this way: the 98% of the predicted third flange represents the True Positive and the remained 2% of the predicted third flange represents the False Positive. This means that the model recognizes the actual third flange (real p3) as the predicted third flange (predicted p3) for the 98% and for the remained 2% recognizes the actual fourth flange (real p4) as the predicted third flange (predicted p3) making mistakes for the 2% of the total predicted third flange. The same discussion can be also done for the fourth flange (p4), since the 97% of the predicted fourth flange are the True Positive, this means that the 97% of the fourth flange are the actual fourth flange. The remained 3% of the predicted fourth flange represents the False Positive because the model recognizes these flanges as the fourth but in reality, are the third ones. A separate discussion must be made for the background False Positive (FP). This means that there were False Positives from the background which shouldn't have been classified at all. The 67% of these False Positives represents the holes, the 11% represents the third flange (p3) and the remained 22% represents the fourth flange (p4). The confusion matrix is also important because it is possible to obtain some performance metrics in order to evaluate the quality of the model. To explain the metrics, we have to do introduce some variables. We denote the variable x as the hidden and the unknown state. This unknown state can be $x=A$ if it is positive and, as said, it belongs to the class A, or it can be $x=B$ if the hidden state is negative and it belongs to the class B. We also denote the variable y as the classified output by the model. It can be $y=A$ if the model classifies a data as positive and belonging to the class A, or it can be $y=B$ if the model classifies a data as negative and belonging to the class B. The first metric analyzed is the Precision. The Precision (p), or the positive predictive value (PPV), is the probability that the unknown state is positive given that the data is classified as positive. It can be calculated as the number of the True Positive divided by the number of all classified positive results ($TP + FP$) and it is indicated as:

$$p = Prob\{x = A|y = A\} = \frac{TP}{TP + FP}$$

The recall, also called true positive rate (TPR), is defined by the probability that the model classifies a data as positive, given that this data is positive in reality. It can be calculated as the number of the True Positive divided by the number of total actual positives (TP + FN) and it is indicated as:

$$r = Prob\{y = A|x = A\} = \frac{TP}{TP + FN}$$

It is possible to use also the F1 score. It is another performance criteria defined as the harmonic mean of precision (p) and the recall (r), and it is calculated as the:

$$F_1 = \frac{2}{\frac{1}{r} + \frac{1}{p}} = 2 \frac{pr}{p + r}$$

From these metrics it is possible to build some graphs as a function of the confidence. These plots are made automatically by the libraries used after the training of the model is finished. The confidence value represents a threshold parameter T. It is used to determine how secure the model is regarding each specific detection. If we consider the example of the binary classification in the sense that a sample can be classified as positive (class A) or negative (class B), the class prediction for each sample is made based on a continuous variable Z. This variable is a score computed for the sample, therefore represents the estimated probability that a data belongs to a class rather than another one. Given the confidence threshold value T, an instance is classified as positive (class A) if $Z > T$, otherwise is classified as negative (class B). The first plot analyzed is the precision curve as a function of the confidence value T (*Figure 63*).

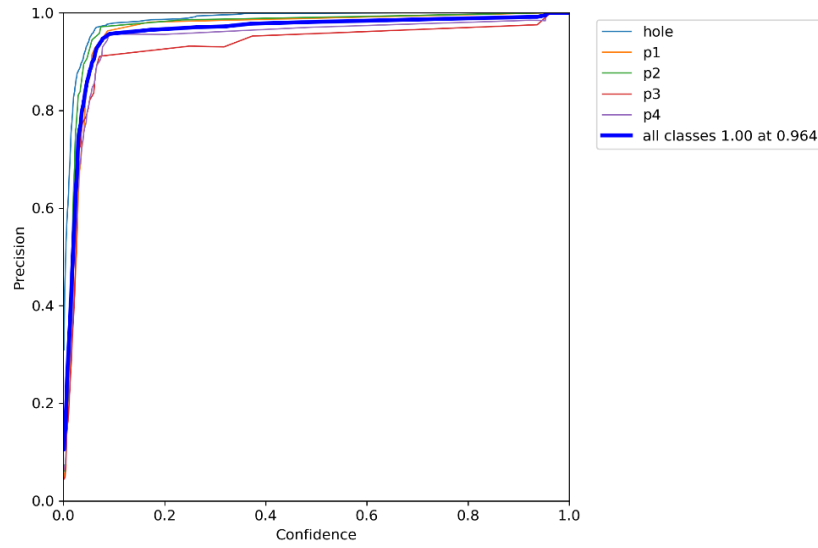


Figure 62: *P-curve of the model flanges and holes*

As it is possible to see, the confidence value that optimizes the precision of all the classes considered is 0.964. This value corresponds to the maximum value of the precision, that is 1 and this means that the False Positive are nullified. As shown later, we consider this confidence value later. The second plot considered is the recall curve as a function of the confidence (*Figure 64*).

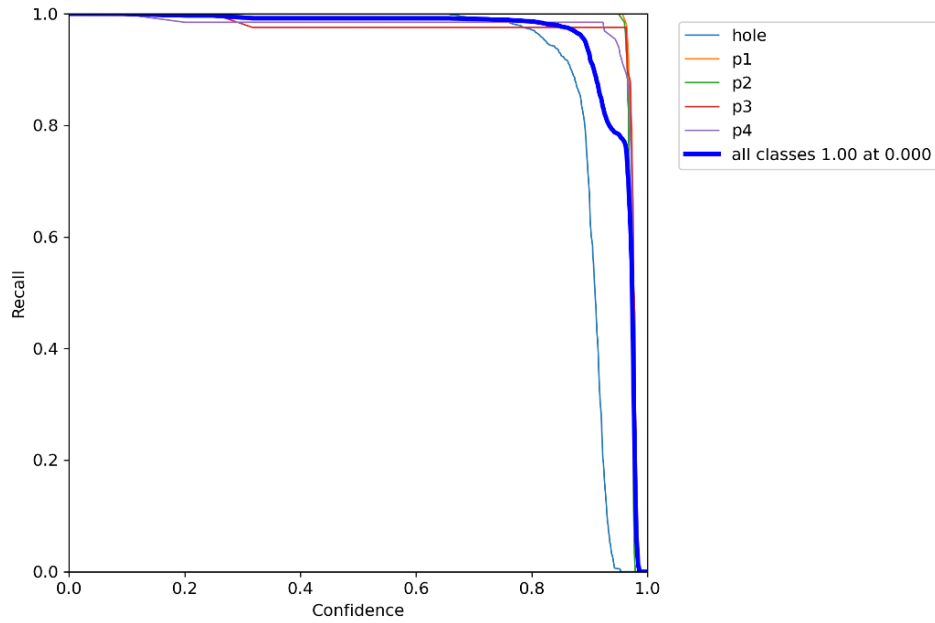


Figure 63: *R-curve of the model flanges and holes*

The confidence value that optimizes the recall considering all the classes is 0. This value corresponds to the maximum value of the recall, that is 1 and this means that the False Negative are nullified. As a result, we must consider the confidence value equal to 0 if we want to maximize only the recall, instead, it must be equal to 0.964 if we want to maximize only the precision. In order to maximize both values we have to consider the F1 curve as a function of the recall (*Figure 64*).

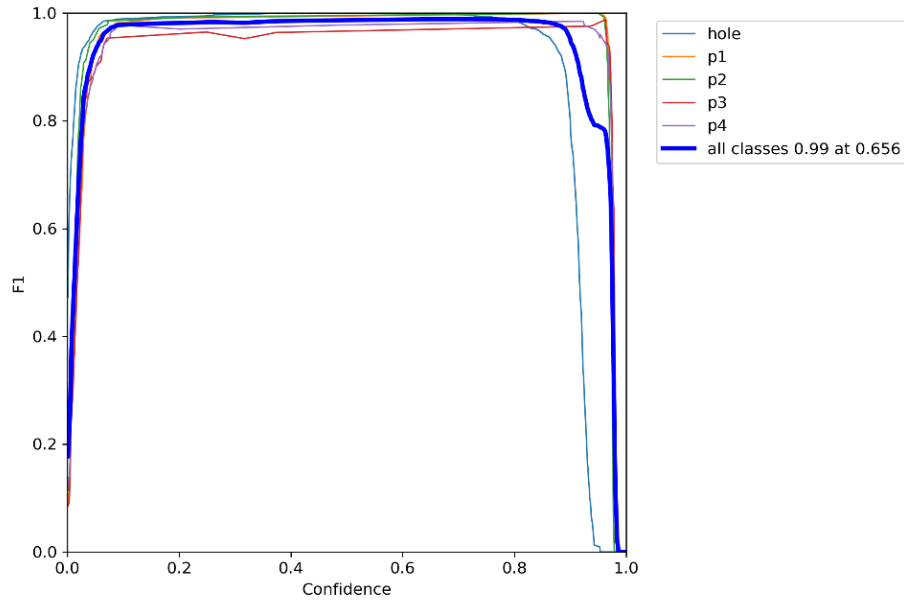


Figure 64: F1 Curve of the model flanges and holes

The F1 score combine precision the and the recall in a single confidence threshold value. As shown in *Figure 62* and in *Figure 63*, if the confidence increases the precision increases, instead the recall decreases. The goal is to find the confidence value that maximizes the F1. As shown in *Figure 64*, the confidence value that maximizes all the classes considered (hole, p1, p2, p3 and p4) is 0.656 since the F1 score of all classes are 0.99. As a result, in the code is considered this value to detect correctly one of these classes. Moreover, in *Figure 65* it is possible to see the precision as a function of the recall and the mAP computed with an IOU threshold equal to 0.5 is equal to 0.994.

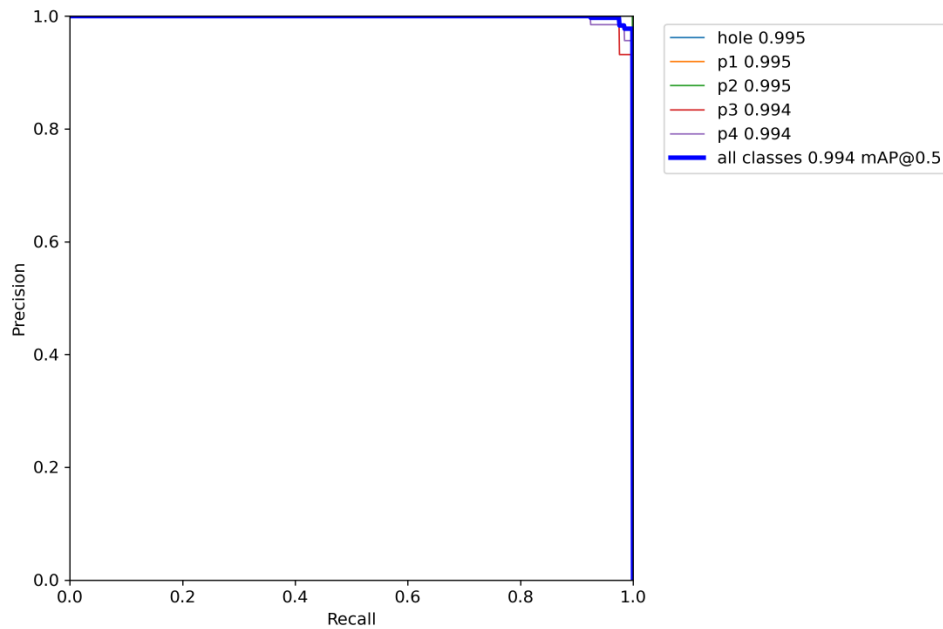


Figure 65: *P-R curve of the model flanges and holes*

The other results that give the training are shown below. There are three different types of losses: the box loss, the objectness loss and the classification loss. In a Machine Learning, the loss functions are used to optimize the model during training. Lower is the value of the loss and higher is the quality of the model to recognize the objects, so the goal is to minimize the loss functions [41].

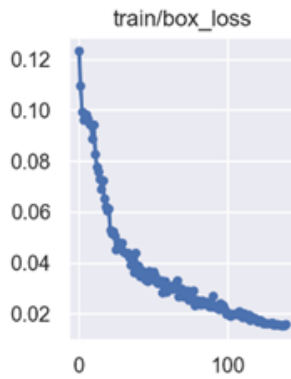


Figure 66: box losses using train set

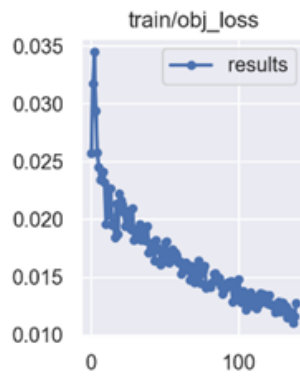


Figure 67: objectness losses using train set

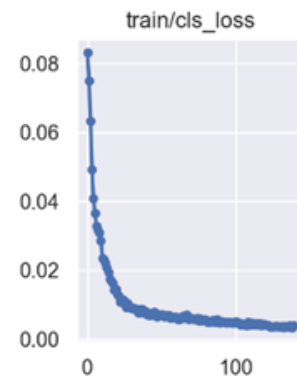


Figure 68: classification losses using train set

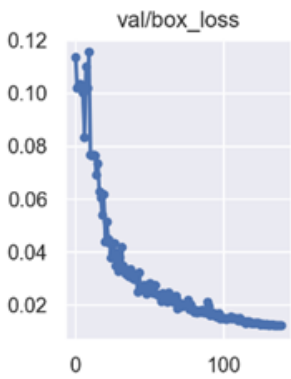


Figure 69: box losses using validation set

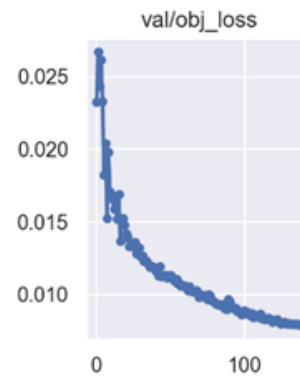


Figure 70: objectness losses using validation set

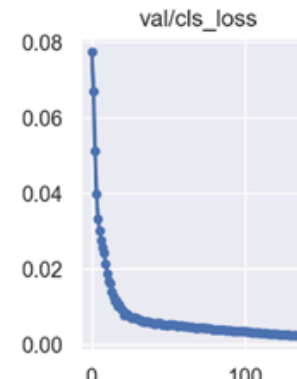


Figure 71: classification losses using validation set

The Figure 66, Figure 67 and Figure 68 show the computed losses as a function of the number of epochs, using the dataset dedicated to the train (train set), instead the Figure 69, Figure 70 and the Figure 71 show the computed losses as a function of the number of epochs using the dataset dedicated to the validation (validation set). The box loss is the bounding box regression loss and describes how well the model locate the center of an object detected and how well the predicted bounding box surrounds this detected object [38]. To compute the box loss is used the mean squared error (MSE) that represents the loss function for regression. The mean squared error is the mean overseen data of the squared differences between the real and the predicted values and can be computed with the formula:

$$L = \frac{1}{N} \sum_{i=0}^N (y_i - \hat{y}_i)^2$$

y_i represents the i -th real value, \hat{y}_i is the i -th predicted value and N is the number of values in the model output [39]. The objectness is the probability that an object exists in an interested region. If the objectness loss is low, it means that the probability that the model detects the object in an image is high [38]. To compute the objectness loss is used the binary cross entropy loss function with the following formula:

$$L = -\frac{1}{N} \sum_{i=1}^N y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i)$$

Also in this case, y_i represents the i -th scalar real value, \hat{y}_i is the i -th predicted value and N is the number of values in the model output [40]. The classification loss describes how well the model predicts a correct class of a given object [38]. To compute the classification loss, it is used the cross entropy with the formula:

$$L = -\sum_{i=1}^n t_i \cdot \log(p_i)$$

t_i represents the truth label of the sample, while p_i represents the probability that a sample appertain to the i -th class and n is the total number of classes [41]. In this model we obtain a box loss using the training set, equal to 0,015795 in the last epoch, instead the minimum loss is reached in the epoch number 138 and is equal to 0,015306 (considering the first epoch equal to the epoch number 0). In the last epoch the value of the objectness loss computed in the train set is equal to 0,012745 and the minimum value is reached in the epoch number 137 that is equal to 0,011006. The value of the classification loss computed in the train dataset in the last epoch is equal to 0,003865, instead the minimum value is reached in the epoch number 137 and is equal to 0,0033098. Considering the validation set, the minimum value of the box loss is reached in the last epoch and is equal to 0,0122. This consideration is also valid for the objectness loss and for the classification loss, since

the minimum objectness loss using the validation set is reached in the last epoch and is equal to 0,0078598 and the minimum classification loss using the validation set is reached in the last epoch too and it is equal to 0,0023908. It is possible to find all these values in the “result” file. In the *Figure 72* and *Figure 73*, are shown the precision and the recall as a function of the number of epochs.



Figure 72: Precision as a function of epochs

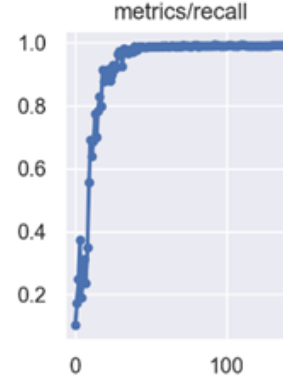


Figure 73: Recall as a function of epochs

In the “results” file we can see that the precision value in the last epoch is equal to 0,98553, instead the maximum precision value is reached in the epoch number 114, and it is 0,98952. The value of the recall in the last epoch is equal to 0,99209 and the maximum value is reached in the epoch number 110, and it is equal to 0,993699. Moreover, in the *Figure 74* it is possible to see the mAP 0.5 explained previously, as a function of the epoch. The maximum value is 0.99446 and is reached in the epoch number 137. In *Figure*

75 instead, it is possible to see the mAP 0.5:0.95 explained previously. The maximum value is reached in the epoch number 138 and the value is equal to 0.92657.

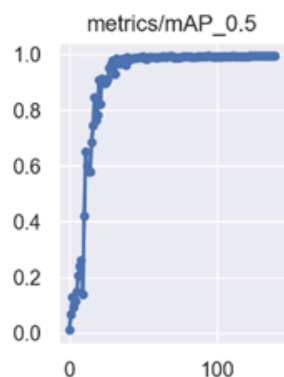


Figure 74: mAP 0.5

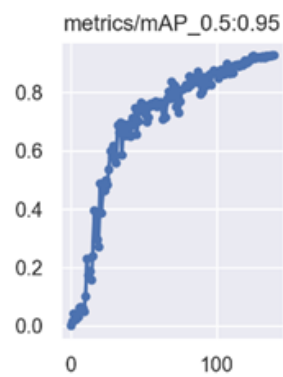


Figure 75: mAP 0.5:0.95

In *Figure 76* is possible to see the number of objects detected for each class during the training.

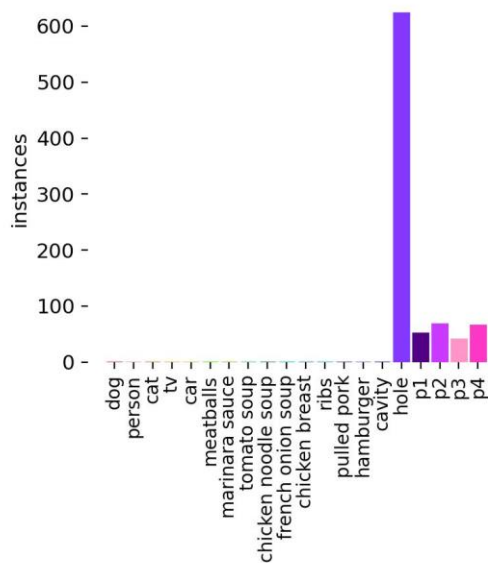


Figure 76: labels

It is possible to see below also the results regarding the model that must be recognize the piaster. It is analyzed now the confusion matrix shown in *Figure 77* deriving by the model that must recognize the piasters.

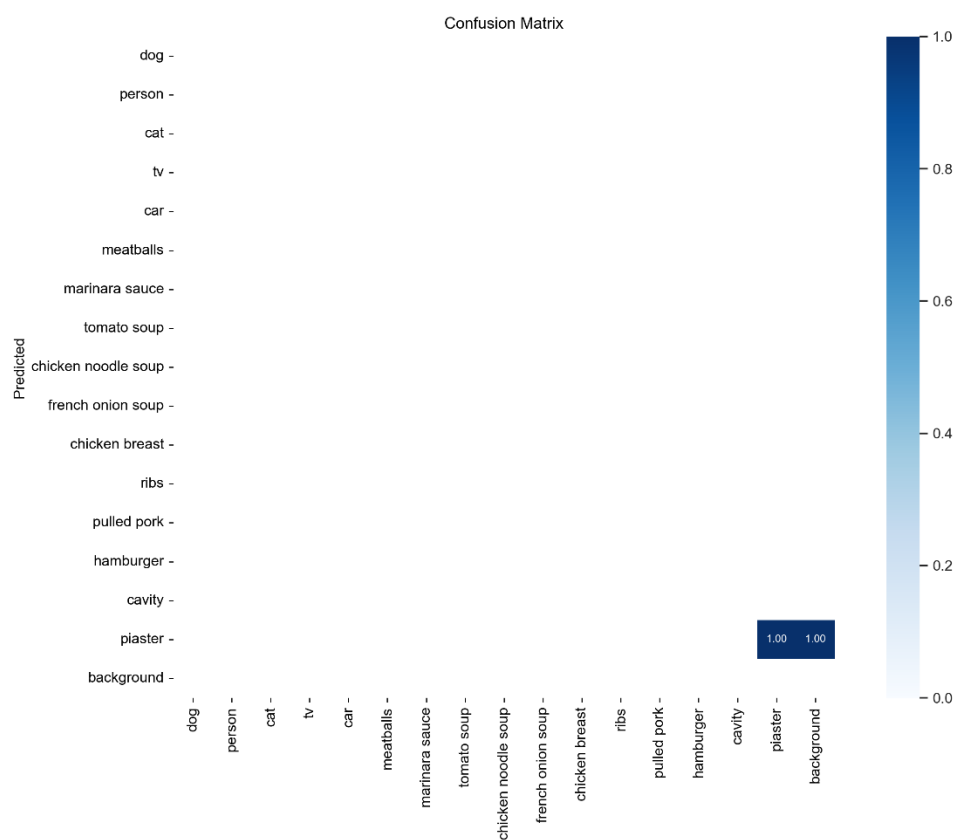


Figure 77: Confusion Matrix of the model piaster

As shown in *Figure 78*, the confidence value that optimizes the precision of the class piaster is 0.99 that is very high.

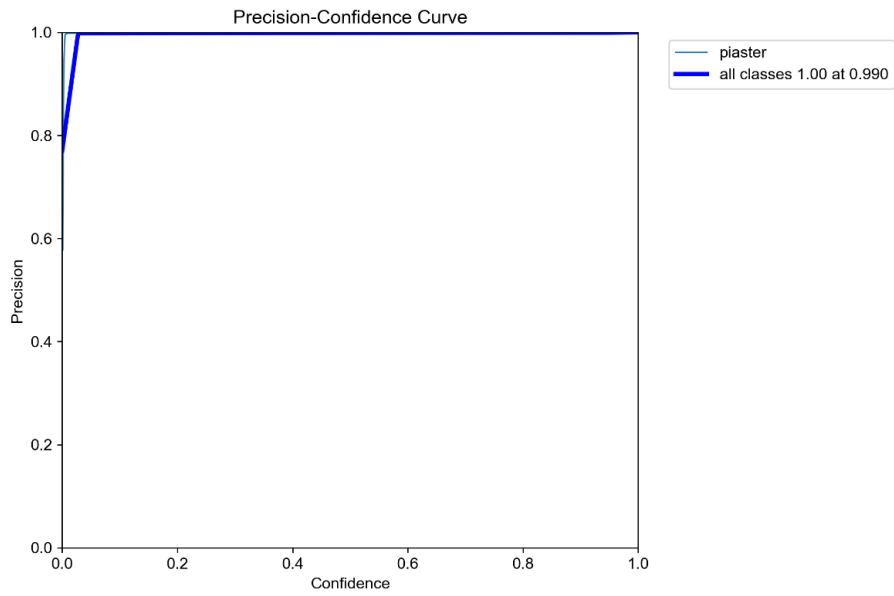


Figure 78: *P-curve of the model piaster*

The recall curve as a function of the confidence is shown in *Figure 79*.

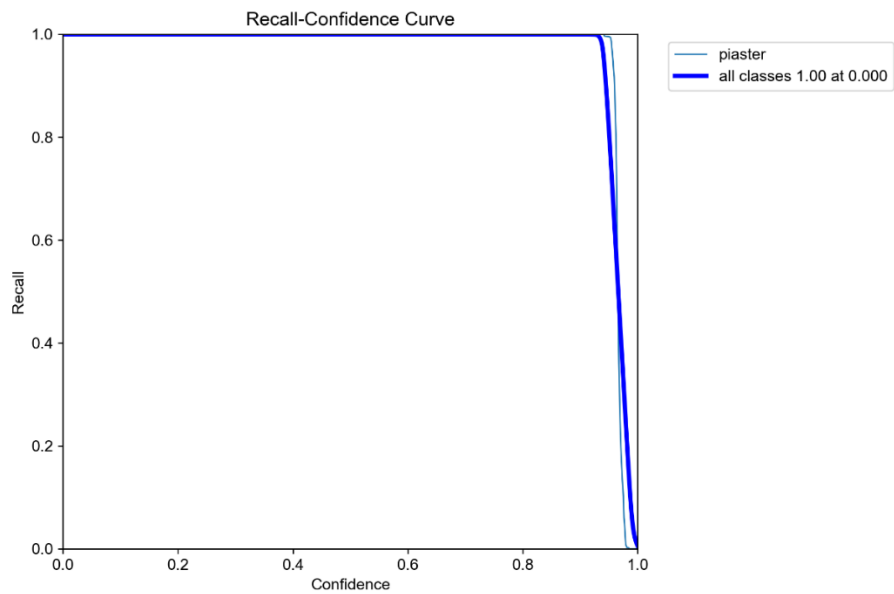


Figure 79: *R-curve of the model piaster*

The confidence value that optimizes the recall considering all the classes is 0. This value corresponds to the maximum value of the recall, that is 1 and this means that the False Negative are nullified. As a result, we must consider the confidence value equal to 0 if we want to maximize only the recall, instead, it must be equal to 0.99 if we want to maximize only the precision. In order to maximize both values we have to consider again the F1 curve as a function of the recall (*Figure 80*).

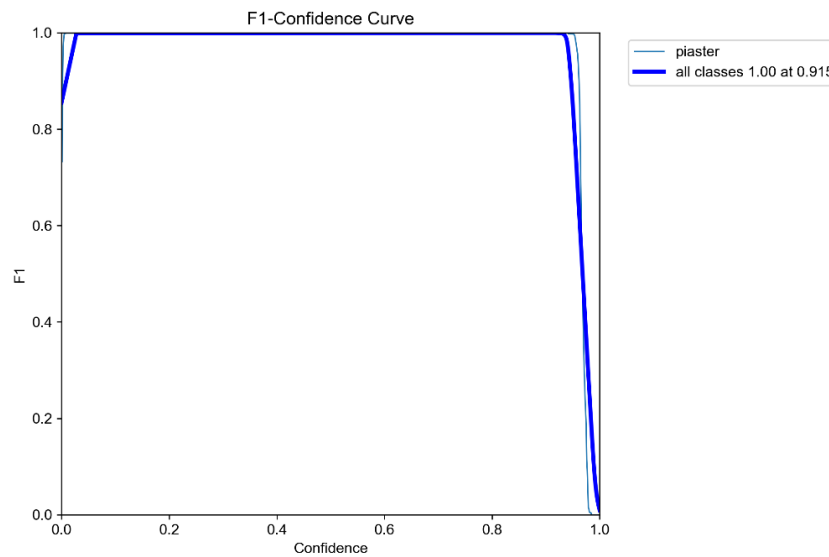


Figure 80: F1 curve of the model piaster

The F1 score combine precision the and the recall in a single confidence threshold value. As shown in *Figure 78* and in *Figure 79*, if the confidence increases the precision increases, instead the recall decreases. The goal is to find the confidence value that maximizes the F1. As shown in *Figure 80*, the confidence value that maximizes the piaster class is 0.915. As a result, in the code is considered this value to correctly detect the piaster class. Moreover, in *Figure 81* it is possible to see the precision as a function of the recall and the mAP computed with an IOU threshold equal to 0.5 is 0.995.

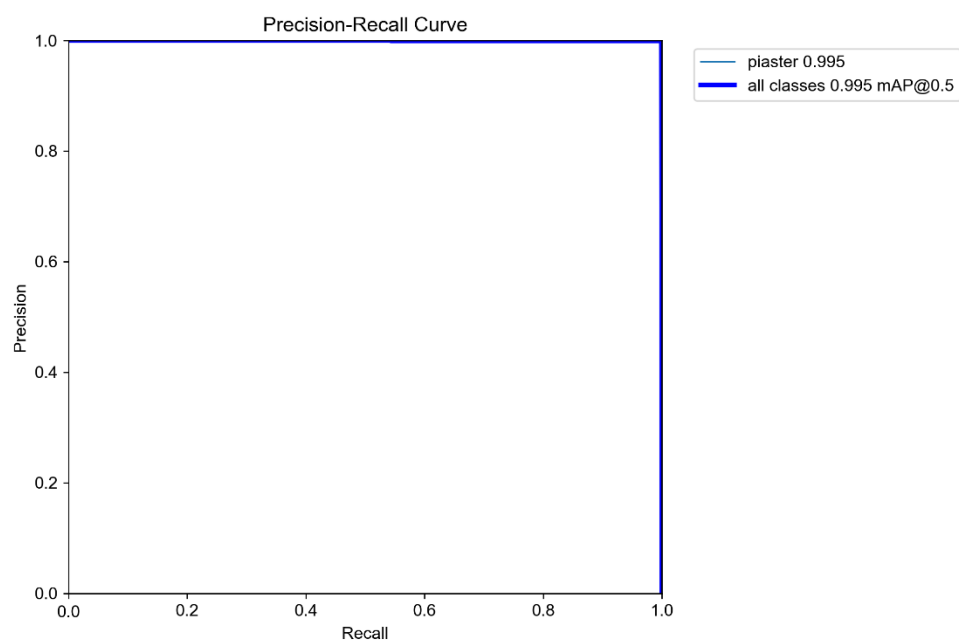


Figure 81: *P-R curve of the model piaster*

The other results that give the training are shown below.

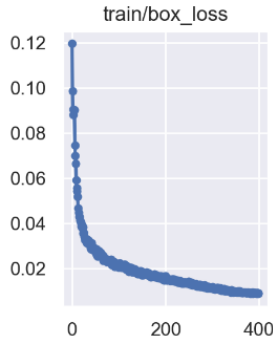


Figure 82: box losses using train set

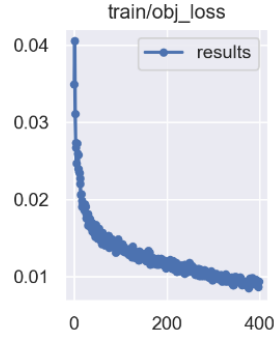


Figure 83: objectness losses using train set

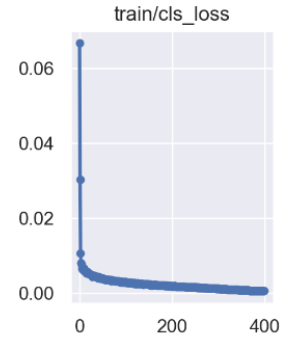


Figure 84: classification losses using train set

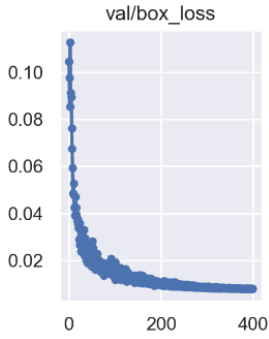


Figure 85: box losses using validation set

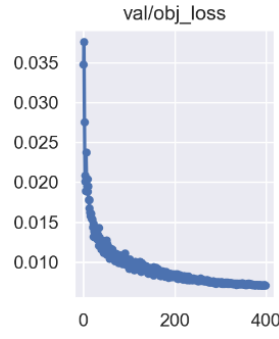


Figure 86: objectness losses using validation set

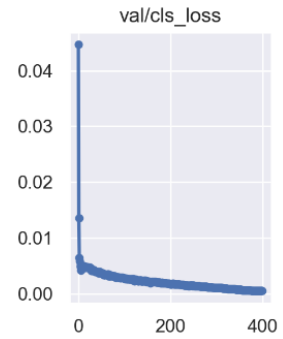


Figure 87: classification losses using validation set

In this model we obtain a box loss using the training set, equal to 0,0090579 in the last epoch, instead the minimum loss is reached in the epoch number 398 and is equal to 0,0087963 (considering the first epoch equal to the epoch number 0). In the last epoch the value of the objectness loss computed in the train set is equal to 0,0093373 and the minimum value is reached in the epoch number 377 that is equal to 0,0084673. The value of the classification loss computed in the train dataset in the last epoch is equal to 0,00055961, instead the minimum value is reached in the epoch number 396 and is equal to 0,00055232. Considering the validation set, in the last epoch the value of the box loss computed is equal to 0.0079922, instead the minimum value of the box loss is reached in the epoch number 398 and is equal to 0,0079569. For the objectness loss instead, the value computed in the last epoch is equal to 0.0071217 and the minimum value reached is in the

epoch number 393 and is equal to 0.0070929. For the classification loss the minimum objectness loss is reached in the last epoch and is equal to 0.00054349.

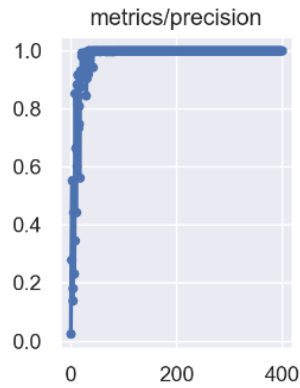


Figure 88: Precision as a function of epochs

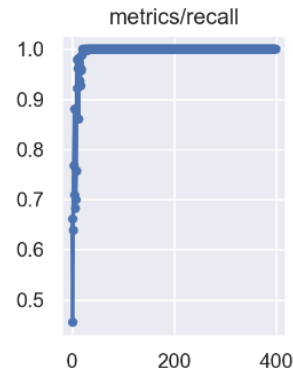


Figure 89: Recall as a function of epochs

In the *Figure 88* and *Figure 89*, are shown the precision and the recall as a function of the number of epochs. In the “results” file we can see that the precision value in the last epoch is equal to 0,99905 that is the maximum. Regarding the precision value, we have obtained the maximum possible value that is equal to 1.

In *Figure 90* and *Figure 91* it is possible to see the results regarding the mAP 0.5 and the mAP 0.5:0.95 as a function of the epoch.

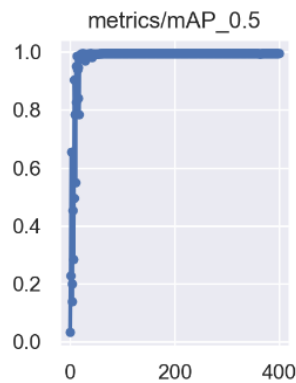


Figure 90: mAP 0.5

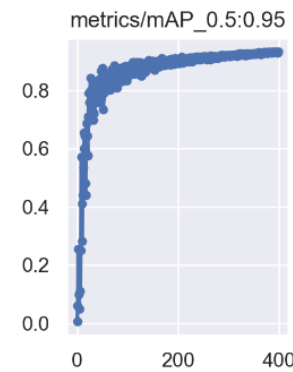


Figure 91: mAP 0.5:0.95

5.2 Output of the models

The output given by each model is a tensor. A tensor is a multidimensional matrix. Before saying what a matrix is, we have to define what an array is. An array is defined as a 1D object that has 1xM dimension, this means that the array has M columns and only one row as shown below:

$$[1 \ 2 \ 3 \ 4]$$

In the example above, the dimension of the array is 1x4. In Python the array is represented with two square brackets as shown below:

```
[1,2,3,4]
```

A matrix is a 2D array, this means that it has not only one row, but it can have more than one row. The dimension of the matrix is NxM, where N in this case represents the number of the rows. An example of a matrix 3 x 2 is shown below:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

In Python the matrix is represented with the following way:

```
[[1,2],  
 [3,4],  
 [5,6]]
```

As said previously, the tensor is a multidimensional array, and it can be seen in different ways (*Figure 92*).

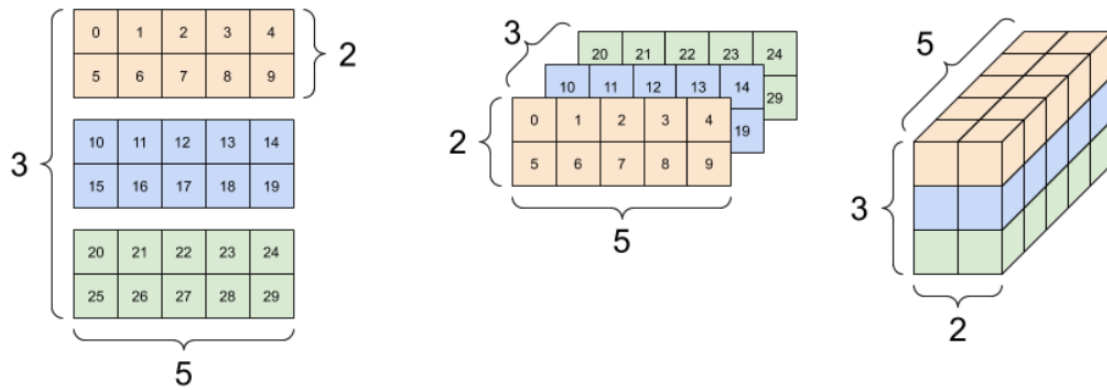


Figure 92: Representation of a tensor [49]

In Python the tensor is represented with the following way [49]:

```
tensor([[[ 0  1  2  3  4]
         [ 5  6  7  8  9]],
        [[10 11 12 13 14]
         [15 16 17 18 19]],
        [[20 21 22 23 24]
         [25 26 27 28 29]]])
```

The tensor can be seen as a mathematical object that contains several superposed matrices. In the example above, the tensor contains 3 matrices of dimension 2x5. In particular, the output given by each model can be seen as a mathematical object that contains several superposed arrays of dimension 1x6. The number of arrays is given by the number of objects that the model detects in real time. Instead, the six columns represent in order the abscissa coordinate of the top left point of the bounding box, the ordinate coordinate of the top left point of the bounding box, the abscissa coordinate of the bottom right point of the bounding box, the ordinate coordinate of the bottom right point of the bounding box and the last number represents the class of the object related to that model. We take as example the *Figure 93* [49].

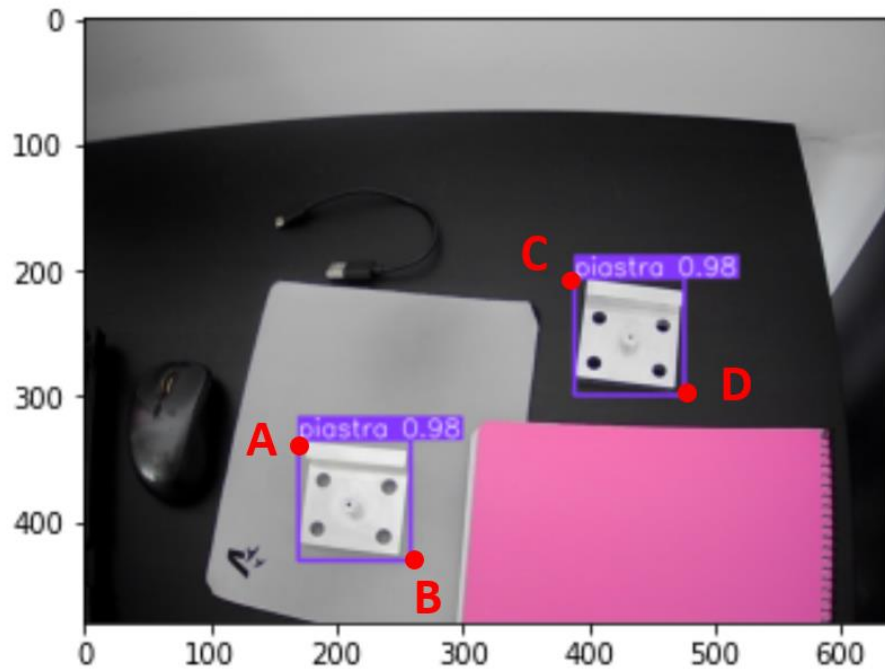


Figure 93: Tensor output of the piaster model

The output of the piaster model is given below.

```
tensor([[170.20285, 334.79074, 259.31354, 430.66318, 0.97972, 15.0000],
        [389.14435, 206.32632, 476.08832, 299.88882, 0.97731, 15.0000]])
```

The array [170.20285, 334.79074, 259.31354, 430.66318, 0.97972, 15.0000] is associated to the below piaster. The value 170.20285 and 334.79074 are respectively the abscissa and ordinate of the point A. The value 259.31354 and 430.66318 are respectively the abscissa and ordinate of the point B. The value 0.97972 represents the confidence with which the model detects that object as piaster. The final value 15.0000 simply represents the number of class detected and in this case the number 15 is associated to the “piaster” class.

The array [389.14435, 206.32632, 476.08832, 299.88882, 0.97731, 15.0000] is associated to the upper piaster. The same consideration done before, is repeated also for this object. The value 389.14435 and 206.32832 are respectively the abscissa and

ordinate of the point C. The value 476.08832 and 299.8882 are respectively the abscissa and ordinate of the point D. The value 0.97731 represents the confidence with which the model detects that object as piaster. The final value 15.0000 simply represents the number of class detected and in this case the number 15 is associated to the “piaster” class. As shown later on, in order to easily do mathematical operations with the tensors, these are transformed into lists. In the previous example, the list associated to the tensor becomes [49]:

<pre>[[170.20285, 334.79074, 259.31354, 430.66318, 0.97972, 15.0000], [389.14435, 206.32632, 476.08832, 299.88882, 0.97731, 15.0000]]</pre>
--

The list is easy to manage with Python, because from the list it is possible to obtain the values inside the list in an easy way and therefore it is possible to do operations in the main code. For this reason, as shown later, in the thesis are used the lists and its contained numbers instead of the associated tensors.

6. Code for real-time object detection

6.1 Settings

First, are imported the libraries needed for the real time detection with the commands below [55].

```
# Import all the Libraries used
import torch
from matplotlib import pyplot as plt
import numpy as np
import cv2
import uuid
import os
import time
```

Then, are loaded the model of the flanges and of the piasters with the following commands [55]:

```
# Load the model from the folders

model_hole=torch.hub.load('ultralytics/yolov5','custom',...
...path='yolov5/runs/train/exp4/weights/last.pt')
```

```
model_piastra=torch.hub.load('ultralytics/yolov5','custom',...
...path='C:/Users/Utente/Desktop/Thesis/Yolov detection...
...colour_dimension/yolov5/runs/train/exp/weights/last.pt')
```

In the real time detection is needed to detect all the possible flanges that are in the frame of the webcam. The function shown below has the goal to consider all the possible classes of the flanges detected in the frame of the webcam with a confidence greater than 0.7. The function returns the array `flanges` that contains the positions in the list `list` of only the flanges detected with a confidence greater than 0.656 obtained by the F1 curve.

```

# DEFINE THE FUNCTIONS

# function useful to count the number of the flanges
def number_flanges(lista):
    # count the number of flanges detected
    number_of_flanges_detected=0
    i=0
    flanges=[]
    n_hole=[]
    for i in range(len(lista)):
        if lista[i][5]==16 or lista[i][5]==17 or lista[i][5]==18 or lista[i][5]==19: #class
#p1,p2,p3 or p4 in the dataset.yaml
            if lista[i][4]>0.656:
                number_of_flanges_detected+=1
                flanges.append(i)

    return flanges

```

To correctly distinguish a flange from another one, is also used the recognition with features, and as said previously, the features are the circumferential holes that in the `model_hole` are represented with the class number 15. The goal of the function below is to count the number of circumferential holes detected in each flange that was previously detected with a confidence greater than 0.656. The function returns the number of the holes that are present in a flange with a confidence greater than 0.656.

```

# function useful to count the number of circumferential holes inside a flange
def number_holes(lista):
    # count the number of holes for each flange detected
    i=0
    n_hole=[]
    for i in range(len(flanges)):
        holes=0
        j=0
        for j in range(len(lista)):
            if lista[j][5]==15: #class hole in the dataset.yaml
                if lista[j][0]> lista[flanges[i]][0] and lista[j][1]> lista[flanges[i]][1] ...
... and lista[j][2]< lista[flanges[i]][2] and lista[j][3]< lista[flanges[i]][3]:
                    holes+=1
        n_hole.append(holes)

    return n_hole

```

6.2 Definition of the workspace

We have to define the workspace in which the piece can be found by the cobot. To do that we have to define the origin that represents the zero-machine from which the cobot start to move to find the piece. The origin is placed as shown in *Figure 94*.

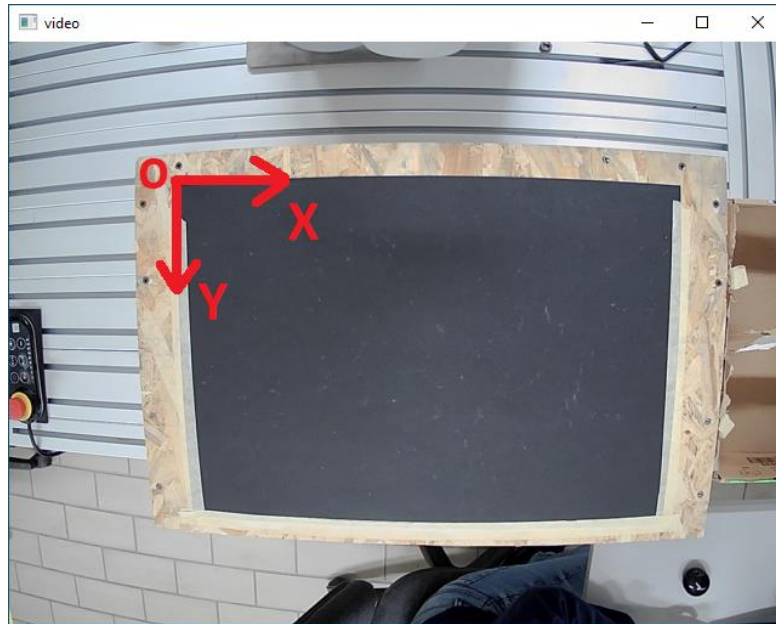


Figure 94: Workspace

In the code the origin has been defined in the following way:

```
# define the 0 machine  
x_base= x1_workspace  
y_base= y1_workspace
```

As shown in *Figure 95* the workspace is defined by 4 points: A, B, C and D. So, the goal is to find the pieces only inside the workspace defined by the points A, B, C and D.

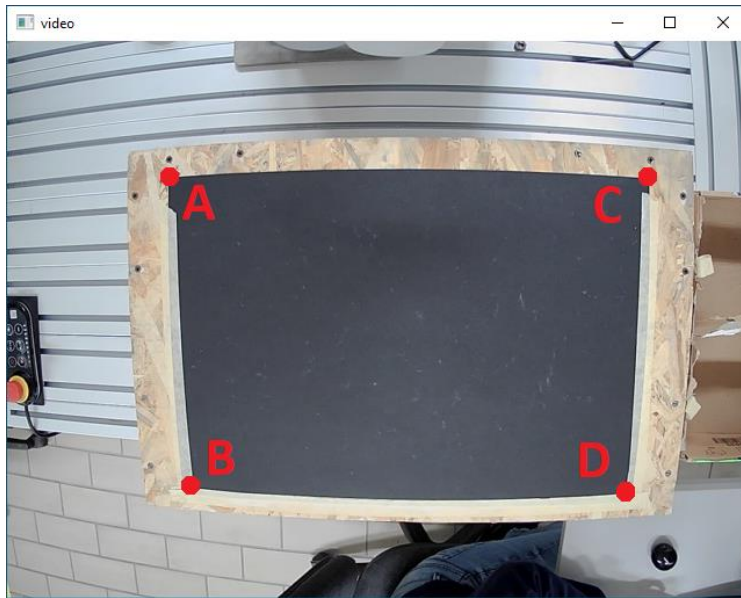


Figure 95: Coordinates to define the workspace

In the code we have defined 4 coordinates: the `x1_workspace` represents the x coordinate of the point A, the `x2_workspace` represents the x coordinate of the point C. Instead, the `y1_workspace` represents the y coordinate of the point A and finally the `y2_workspace` represents the y coordinate of the point B.

```
# define the workspace
x1_workspace= 139
x2_workspace= 555
y1_workspace= 115
y2_workspace= 390
```

Since the command sent by the Python code are referred to the TCP final position, we have to define the distance in the y coordinate that there is between the TCP frame and the camera. This is because the important thing is that the cobot has to move to the center of the bounding box of the piece in the way that the camera of the cobot can detect the piece. For this reason, the best way is to move the cobot in order to have the alignment of the center of the bounding box of the piece with the center of the camera. The offset of the center of the camera with respect to the frame of the TCP has been measured and has been defined in the code in the following way:


```
# define the distance between the camera and the center of the TCP  
coord_camera_y= 62
```

To send the coordinate of the center of the bounding box of the piece, we have to send the coordinates along x, y and z. Since the cobot does not move from the zero-machine along the z axis we define the movement of the cobot along the z axis equal to 0 as follows:

```
# set the coordinate Z equal to 0  
coord_z_mm= 0
```

6.3 Settings for the colour detection

Other values that must be defined are the upper and the lower bounded for the colour detection used for the piasters. To describe the colour of the piasters it has been used the HSV model. It is a coded digital system with which it is possible to define colors through three parameters: hue, saturation, and brightness. Therefore, it is a way to represent the colors in a digital system. The HSV assigns to a color three dimensions that mathematically describe the space or also called color spectrum (*Figure 96*).

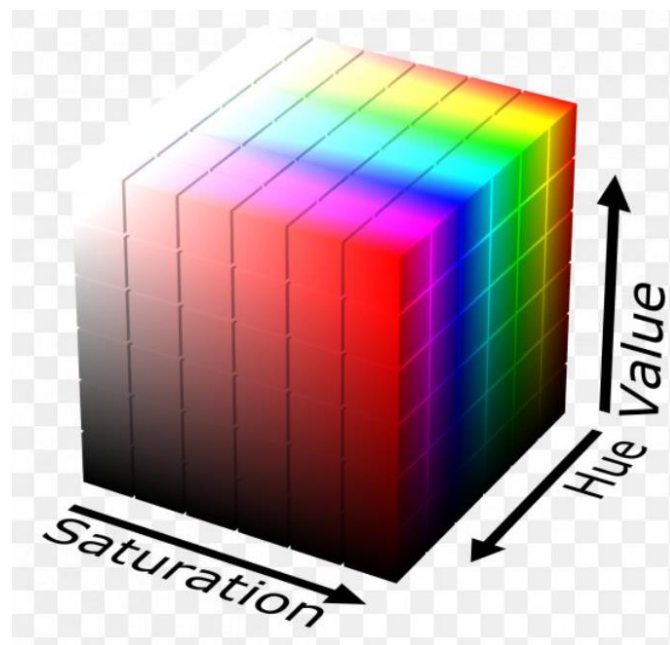


Figure 96: Color spectrum of the HSV model [54]

The hue is the color itself, the saturation represents how saturated we want that color and the value represents the brightness of the color. By fixing the value parameter, it is possible to obtain a 2D representation in which the abscissa represents the hue, and the ordinate represents the saturation (*Figure 97*).

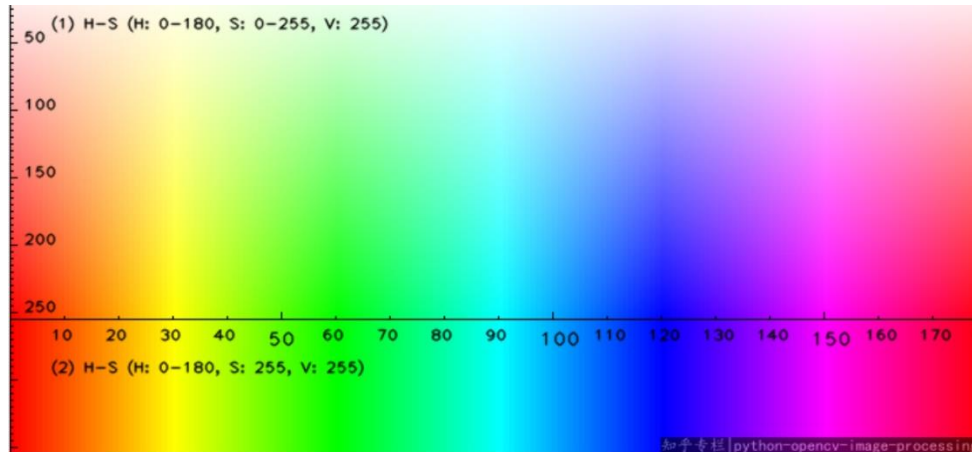


Figure 97: 2D representation of the HSV model by fixing the value V [56]

The value of the hue goes from 0 to 180, the value of the saturation goes from 0 to 255 and finally the value of the brightness (value) goes from 0 to 255. By looking at the *Figure 96* and the *Figure 97*, it has been defined the range of the three values in order to detect from the frame of the webcam only the objects colored in yellow ($H=15:50$, $S=50:255$, $V=200:255$). The same thing it has been done also for the object colored in brown ($H=15:50$, $S=50:255$, $V=200:255$). All these parameters are defined in the main code as shown below [56].

```
# we define the upper and the lower bounded for the yellow colour
lower_yellow=np.array([15,50,200])
upper_yellow=np.array([50,255,255])

# we define the upper and the lower bounded for the brown colour
lower_brown=np.array([0,50,100])
upper_brown=np.array([50,120,200])
```

6.4 Activation of the external webcam

After defining the functions useful in the main code, it is activated the webcam connected to the PC with the function `cap.isOpened()` and then, at each time is obtained a frame from the webcam with the function `cap.read()`. Then, it is called back the results of the `model_hole` and of the `model_piastra` because while the webcam is activated, the frame of the webcam is given as input at each time to the `model_hole` and to the `model_piastra` and they give as output the tensors as explained previously. As said, in order to easily do mathematical operations and to easy manage the numbers contained in the tensor, it is used the function `tensor.tolist()` in order to transform the tensor into a list. All the operations described are shown below.

```
# ACTIVATE THE WEBCAM
cap=cv2.VideoCapture(0)
while cap.isOpened():

    ret,frame=cap.read()

    # we apply the model of the flanges and holes to our image detected by webcam
    results_hole=model_hole(frame)

    # we apply the model of the piaster to our image detected by webcam
    results_piastra=model_piastra(frame)

    # convert the result of the model of flanges and holes in a tensor
    tensor_hole=results_hole.xyxy[0]
    # convert the tensor in a list
    lista=tensor_hole.tolist()

    # convert the result of the model of piasters in a tensor
    tensor_piastra=results_piastra.xyxy[0]
    # convert the tensor in a list
    list_piastra=tensor_piastra.tolist()
```

Then, taking as input the list, there are called back the functions that returns the array `flanges` that contains the positions in the list `list` of only the flanges detected with a confidence greater than 0.656 and the number of the holes that are present in them. These operations are done with the commands below.

```

# if flanges are detected in the frame of the camera
if len(lista)>0:
    # count the number of flanges detected
    flanges=number_flanges(lista)

    # count the number of holes for each flange detected
    n_hole=number_holes(lista)

```

With the commands below the algorithm can detect the flange with a number of circumferential holes equal to 4. After obtained the coordinate of the bounding box x_1 , y_1 , x_2 , y_2 to detect the object as a flange, to underline that the flange has three circumferential holes, the algorithm draws a rectangle (with the coordinate obtained from the bounding boxes) around the flange with the function `cv2.rectangle` and writes on top of the rectangle “4 holes” with the function `cv2.putText`. In the function `cv2.rectangle` it has also defined the colour of the rectangle with the RGB values (0,255,153) and is also defined the the thickness of the edge of the rectangle (4). It has been also chosen the font of the writing with `cv2.FONT_HERSHEY_SIMPLEX`.

```

# initialize the index that indicates the number associated to the flange
i=0

# check the number of circumferential holes for each flange detected
for i in range(len(flanges)):

    # if the number of circumferential holes in the i-th flange is equal to 4
    if n_hole[i]==4:

        # Save the coordinates of the four vertices of the bounding box
        x1=int(lista[flanges[i]][0])
        y1=int(lista[flanges[i]][1])
        x2=int(lista[flanges[i]][2])
        y2=int(lista[flanges[i]][3])
        # draw the bounding box
        frame=cv2.rectangle(frame, (x1,y1),(x2,y2), (0,255,153), 2 )
        # choose the font of the text
        font= cv2.FONT_HERSHEY_SIMPLEX
        # write the text "4 holes"
        frame= cv2.putText(frame, '4 holes',(x1-10,y1-10), font,...
...0.7, (0,255,153),2, cv2.LINE_AA)

```

6.5 From pixels to mm

Now we consider the figures below.

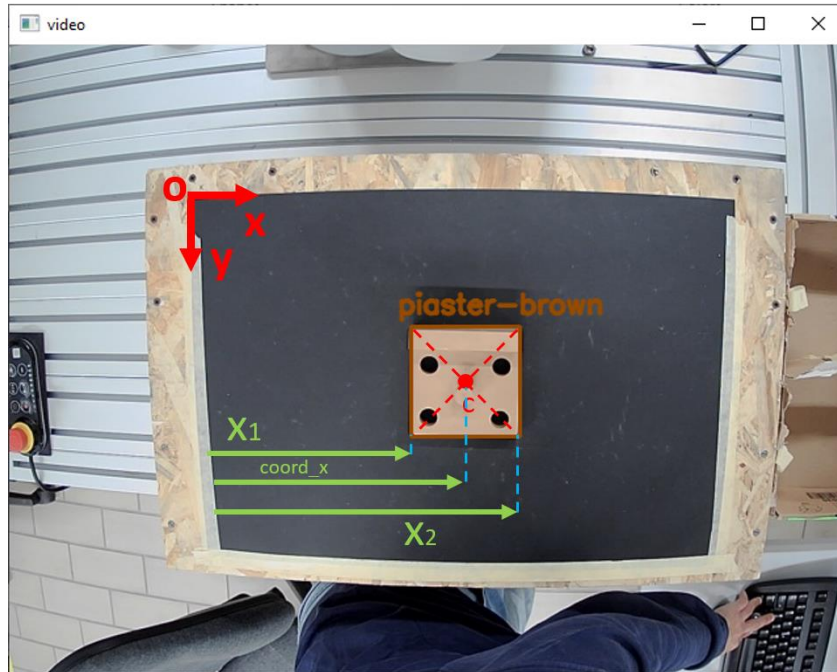


Figure 98: coordinate along x axis of the center of the bounding box

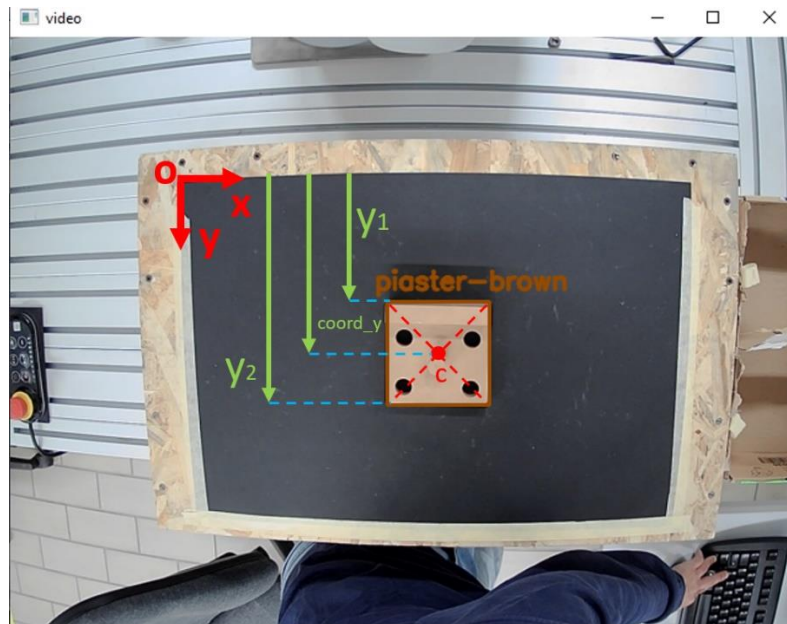


Figure 99: coordinate along y axis of the center of the bounding box

With the reference of the *Figure 98* and the *Figure 99*, the coordinates of the center of the piece in pixels have been defined in the code in the following way:

```
# if the piece is inside the workspace:
if x1>=x1_workspace and x2<=x2_workspace and y1>=y1_workspace and ...
... y2<=y2_workspace:

# memorize the actual coordinate of the centre of the piece
coord_x=(x2-x1)/2 + x1
coord_y=(y2-y1)/2 + y1
```

To pass from the pixels to the mm that must be sent to the cobot, we have used a linear proportion. First of all, we have measured the width of the brown piaster shown in *Figure 100* and we have obtained 93 mm. Then, we have printed the distance in pixels of the bounding box detected in the piaster brown in *Figure X* using the trained model to detect the piasters. In this way we have obtained that the piaster has the width in pixels corresponding to 80 pixels.

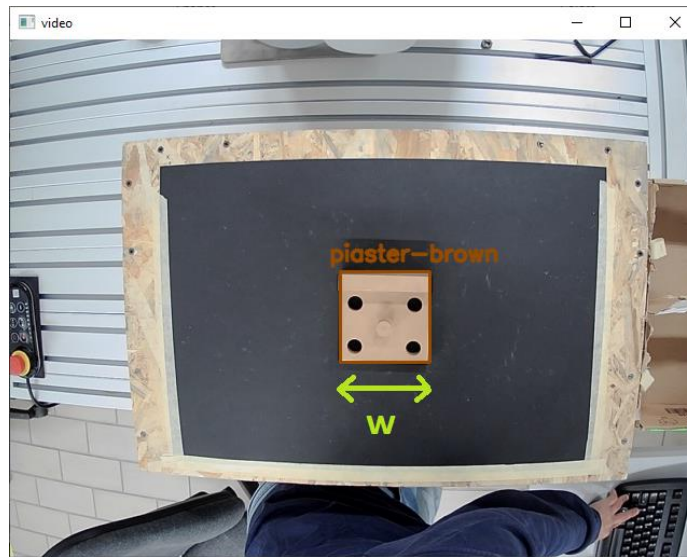


Figure 100: brown piaster detected by the algorithm

Then, to find any coordinates of the center of the bounding box of a piece as shown in *Figure 101*, the proportion has been done in the following way:

$93 \text{ mm} : 80 \text{ pixels} = \text{coordinates in mm of point C} : \text{coordinates in pixels of point C}$

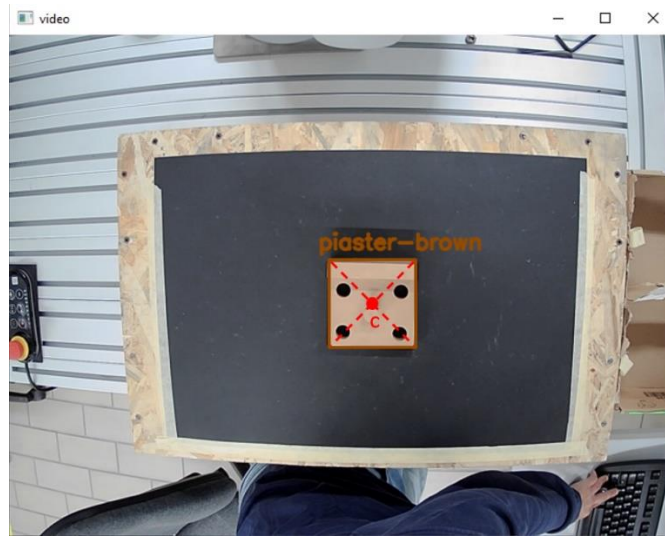


Figure 101: center of the bounding box of the brown piaster

This reasoning can be done for any coordinates of the center of the bounding box of the future pieces detected and the conversion from pixels to mm has been written in the code in the following way:

```
coord_x_mm=(93*coord_x)/80 - x_base  
coord_y_mm=(93*coord_y)/80 - y_base - coord_camera_y
```

The same considerations are done also for the flanges with a number of circumferential holes equal to 2.


```

# if the number of circumferential holes in the i-th flange is equal to 2
if n_hole[i]==2:

    # Save the coordinates of the four vertices of the bounding box
    x1=int(lista[flanges[i]][0])
    y1=int(lista[flanges[i]][1])
    x2=int(lista[flanges[i]][2])
    y2=int(lista[flanges[i]][3])
    # draw the bounding box
    frame=cv2.rectangle(frame, (x1,y1),(x2,y2), (203,192,255), 2 )
    # choose the font of the text
    font= cv2.FONT_HERSHEY_SIMPLEX
    # write the text "2 holes"
    frame= cv2.putText(frame, '2 holes',(x1-10,y1-10), font,...
...0.7, (203,192,255),2, cv2.LINE_AA)

    # if the piece is inside the workspace:
    if x1>=x1_workspace and x2<=x2_workspace and y1>=y1_workspace and ...
... y2<=y2_workspace:

        # memorize the actual coordinate of the centre of the piece
        coord_x=(x2-x1)/2 + x1
        coord_y=(y2-y1)/2 + y1
        coord_x_mm=(93*coord_x)/80 - x_base
        coord_y_mm=(93*coord_y)/80 - y_base - coord_camera_y

```

When instead the number of holes is equal to 3 we have to make a difference.

```

# if the number of circumferential holes in the i-th flange is equal to 3
if n_hole[i]==3:

    # Save the coordinates of the four vertices of the bounding box
    x1=int(lista[flanges[i]][0])
    y1=int(lista[flanges[i]][1])
    x2=int(lista[flanges[i]][2])
    y2=int(lista[flanges[i]][3])

```



```

# if the piece with the 3 holes and it is big
if (x2-x1)>55:
    # draw the bounding box
    frame=cv2.rectangle(frame, (x1,y1),(x2,y2), (0,153,255), 2 )
    # choose the font of the text
    font= cv2.FONT_HERSHEY_SIMPLEX
    # write the text "3 holes"
    frame= cv2.putText(frame, '3 holes big',(x1-10,y1-10), font,...
...0.7, (0,153,255),2, cv2.LINE_AA)

    # if the piece is inside the workspace:
    if x1>=x1_workspace and x2<=x2_workspace and y1>=y1_workspace and ...
... y2<=y2_workspace:

        # memorize the actual coordinate of the centre of the piece
        coord_x=(x2-x1)/2 + x1
        coord_y=(y2-y1)/2 + y1
        coord_x_mm=(93*coord_x)/80 - x_base
        coord_y_mm=(93*coord_y)/80 - y_base - coord_camera_y

```

```

# if the piece with the 3 holes and is small
else:
    # draw the bounding box
    frame=cv2.rectangle(frame, (x1,y1),(x2,y2), (0,153,255), 2 )
    # choose the font of the text
    font= cv2.FONT_HERSHEY_SIMPLEX
    # write the text "3 holes"
    frame= cv2.putText(frame, '3 holes small',(x1-10,y1-10), font,...
...0.7, (0,153,255),2, cv2.LINE_AA)

    # if the piece is inside the workspace:
    if x1>=x1_workspace and x2<=x2_workspace and y1>=y1_workspace and ...
... y2<=y2_workspace:

        # memorize the actual coordinate of the centre of the piece
        coord_x=(x2-x1)/2 + x1
        coord_y=(y2-y1)/2 + y1
        coord_x_mm=(93*coord_x)/80 - x_base
        coord_y_mm=(93*coord_y)/80 - y_base - coord_camera_y

```

With the same method shown previously, below are shown the command that allows the algorithm to also detect the piasters and the coordinates to which the robot must go. In the case of the piaster as seen from the results of the model, it has been considered a confidence for the piaster greater than 0.915 that is derived from the F1 curve.

```

# initialize the index that indicates the number associated to the piaster
i=0

# check for the i-th piaster if it is yellow brown or steel
for i in range(len(list_piastra)):
    if list_piastra[i][4] > 0.915:

        # Save the coordinates of the four vertices of the bounding box
        x1=int(list_piastra[i][0])
        y1=int(list_piastra[i][1])
        x2=int(list_piastra[i][2])
        y2=int(list_piastra[i][3])

```

6.6 Colour detection

Regarding the piasters the first thing that it has done is to isolate the piasters detected in the frame of the webcam cutting the rectangle around the piaster with the command `piastra_cut=frame[y1:y2,x1:x2]`. This simplifies the analysis of the color of the piaster. Since Python uses as basis the BGR for the color, it has been used the function `cv2.cvtColor()` to convert the BGR into HSV that is used in the color detection, specifying the conversion with `cv2.COLOR_BGR2HSV`. In order to easily manage the piaster detected with a particular wanted color (in the first case is yellow), it has been created a mask with the command `cv2.inRange()` that separate the objects with the color wanted (yellow) that appears in the cut frame, from the original cut. The mask shows blocks of colored pixels that are in the chosen range of the HSV. From this figure it can be easily obtained the contours of all the yellow objects (blocks of the colored pixels) that appears in the mask. It can happen that the piaster detected with the wanted color is not only the block of pixels that is present in the cut frame, since there can be small possible yellow pixels that in the chosen range of the HSV. This problem will be solved later on. To find the contours of all the possible yellow objects that are in the range of the yellow color, is used the command `cv2.findContours()` with the purpose to draw a rectangle around each possible yellow object detected. The output of this command gives two lists: `contours_yellow` and `hierarchy_yellow`. The list `contours_yellow` contains the coordinate of the contours of the object (block of the pixels with the wanted color), instead, the hierarchy contains the relationship that the contours have each other. The

`cv2.RETR_EXTERNAL` and `cv2.CHAIN_APPROX_SIMPLE` are simply flags and in particular with the flag `cv2.CHAIN_APPROX_SIMPLE` it is specified that in the list are stored only 4 coordinates of the contour of the yellow object, that represents the four angles of the contour [56].

The commands explained so far, are used in Python and are shown below for the yellow objects.

```
# detect only the yellow piaster
piastra_cut=frame[y1:y2,x1:x2]
piastra_yellow=cv2.cvtColor(piastra_cut, cv2.COLOR_BGR2HSV)
mask_yellow=cv2.inRange(piastra_yellow, lower_yellow, upper_yellow)
contours_yellow, hierarchy_yellow= cv2.findContours(mask_yellow, ...
...cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
```

The same consideration is also done for the brown objects with the same commands [56].

```
# detect only the brown piaster
piastra_brown=cv2.cvtColor(piastra_cut, cv2.COLOR_BGR2HSV)
mask_brown=cv2.inRange(piastra_brown, lower_brown, upper_brown)
contours_brown, hierarchy_brown= cv2.findContours(mask_brown, ...
...cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
```

What it is done so far is to isolate the piaster detected, create a new frame with the isolate piaster and then apply the colour recognition on that frame. The problem is that, as said previously, the algorithm can detect all the possible block of pixels with the wanted colour that are present in the cut frame, even very small pixels that can be there. For this work we are interested in detecting only the piaster with a particular colour and not very small pixels. To solve this problem, the algorithm must check that for each block of pixels with the colour in the range chosen of the HSV (`for contour_yellow in contours_yellow:`), the area of the block of pixels must be greater than a certain value (`if cv2.contourArea(contour_yellow) > 1000:`), in this case it has been chosen a value of 1000 pixels. This ensures the exclusion of all the possible small blocks of pixels and therefore, the algorithm, can detect in the cut frame only the block of pixels that represent the piaster. This it has been done with the following commands [56].

```

# if the piaster detected is yellow
if len(contours_yellow)!=0:
    for contour_yellow in contours_yellow:
        # in order to not detect the small points and features that can be yellow
        if cv2.contourArea(contour_yellow) > 1000:

```

Then, it has been used the function `cv2.boundingRect()` to obtain the coordinate of the rectangle around the piaster with the wanted color, and this operation is done in the cut frame. Then, to draw the rectangle in the original frame it is used the function `cv2.rectangle()`. To underline that the algorithm has detected a yellow piaster, the writing “piaster-yellow” it has putted above the rectangle. As done in previous similar cases, the font of the text has been set with the command `cv2.FONT_HERSHEY_SIMPLEX` and then the text was putted with the function `cv2.putText()`. The implementation in Python is shown below.

```

# we find the coordinate of the yellow piaster
x,y,w,h=cv2.boundingRect(contour_yellow)

# we draw the rectangle around the yellow piaster
frame=cv2.rectangle(frame, (x+x1,y+y1),(x+w+x1,y+h+y1),(49,226,209),2)
# choose the font of the text
font= cv2.FONT_HERSHEY_SIMPLEX
# write the text "piaster-yellow"
frame=cv2.putText(frame, 'piaster-yellow',...
... (x+x1-10,y+y1-10), font, 0.7, (49,226,209),2, cv2.LINE_AA)

# if the piece is inside the workspace
if x1>=x1_workspace and x2<=x2_workspace and y1>=y1_workspace...
... and y2<=y2_workspace:

# memorize the actual coordinate of the centre of the piece
coord_x=(x2-x1)/2 + x1
coord_y=(y2-y1)/2 + y1
coord_x_mm=(93*coord_x)/80 - x_base
coord_y_mm=(93*coord_y)/80 - y_base - coord_camera_y

```

The same procedure is repeated also for the piaster with the brown color [56].

```

# if the piaster detected is brown
if len(contours_brown)!=0:
    for contour_brown in contours_brown:
        # in order to not detect the small points and features that can be brown
        if cv2.contourArea(contour_brown) > 1000:
            # we find the coordinate of the brown piaster
            x,y,w,h=cv2.boundingRect(contour_brown)

            # we draw the rectangle around the brown piaster
            frame=cv2.rectangle(frame, (x+x1,y+y1),(x+w+x1,y+h+y1), (0,75,150),3)
            # choose the font of the text
            font= cv2.FONT_HERSHEY_SIMPLEX
            # write the text "piaster-brown"
            frame=cv2.putText(frame, 'piaster-brown',...
... (x+x1-10,y+y1-10), font, 0.7, (0,75,150),2, cv2.LINE_AA)

            # if the piece is inside the workspace
            if x1>=x1_workspace and x2<=x2_workspace and y1>=y1_workspace...
... and y2<=y2_workspace:

                # memorize the actual coordinate of the centre of the piece
                coord_x=(x2-x1)/2 + x1
                coord_y=(y2-y1)/2 + y1
                coord_x_mm=(93*coord_x)/80 - x_base
                coord_y_mm=(93*coord_y)/80 - y_base - coord_camera_y

```

Since the algorithm can also detect a steel piaster and not only the yellow and the brown one, it has also been written another command shown below. Indeed, if the piaster detected is not brown neither yellow, it will be in steel material.

```

# if the piaster detected is steel
else:
    # draw the bounding box
    frame=cv2.rectangle(frame, (x1,y1), (x2,y2), (223,205,0),2)
    # choose the font of the text
    font= cv2.FONT_HERSHEY_SIMPLEX
    # write the text "piaster-steel"
    frame=cv2.putText(frame, 'piastra-steel',(x1-10,y1-10),font, 0.7,...
...(223,205,0),2, cv2.LINE_AA)
    # if the piece is inside the workspace
    if x1>=x1_workspace and x2<=x2_workspace and y1>=y1_workspace...
...and y2<=y2_workspace:

        # memorize the actual coordinate of the centre of the piece
        coord_x=(x2-x1)/2 + x1
        coord_y=(y2-y1)/2 + y1
        coord_x_mm=(93*coord_x)/80 - x_base
        coord_y_mm=(93*coord_y)/80 - y_base - coord_camera_y

```

6.7 Frame that appears in the workstation

As a result, to show the frame in the PC with all the detections done and underlined with the text and with the rectangles, the command `cv2.imshow()` is used.

```
# show the frame of the camera with the detected objects
cv2.imshow('video', frame)
```

The last three commands shown below means that until we do not press the key 'q' in the keyboard, the algorithm will run and the frame with the detected object will appear in the screen of the PC. When instead we press the key 'q' to quit from the program, the webcam will be disactivated with the function `cap.release()` and the window that shows the detection in real time will be also closed using the command `cv2.destroyAllWindows()`.

```
# if press 'q' we quit from the while cycle
if cv2.waitKey(10) & 0xFF == ord('q'):
    break

# close all the windows and end the program
cap.release()
cv2.destroyAllWindows()
```

By running the Python Code from the workstation, we can see the output similar to which is shown in *Figure 102*. We can easily see that the algorithm can detects each different piece in real-time reaching the goal of the real-time object detection through technological features.

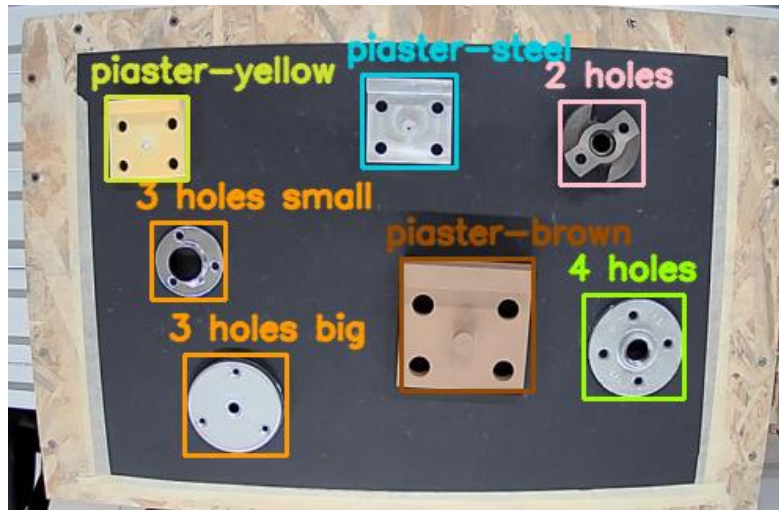


Figure 102: Output of the Python code for real-time object detection

By changing the positions and the orientation of each object in real-time as shown in *Figure 103* it is possible to see that the algorithm can detect anyway in the right way each piece and can distinguish them through the technological features.

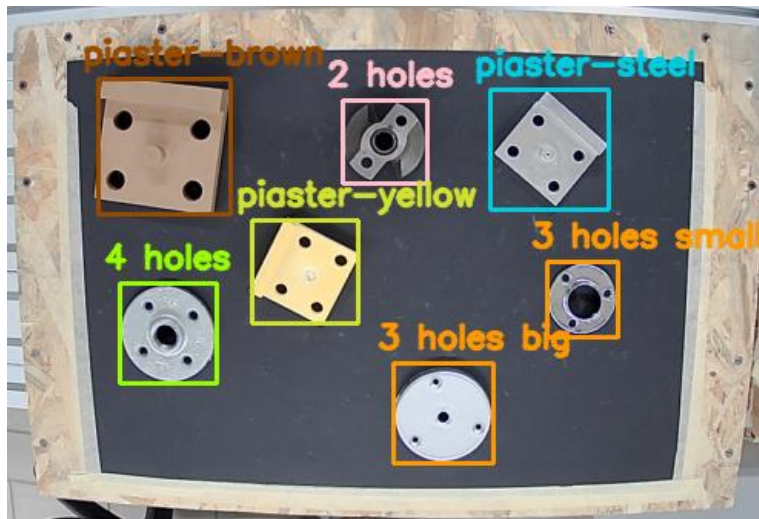


Figure 103: Another output of the Python code for real-time object detection

7. Integration of software in the cobot

7.1 Baricenter and manipulation

The algorithm implemented so far it is not sufficient to grab each piece for different reasons. It is true that it is possible to send the coordinate of the center of the bounding box of the piece as we can see later, but as seen in *Figure 102* and *Figure 103* the bounding box does not coincide with the contour of the piece and so the center of the piece could not coincide with the center of the bounding box of it. Moreover, the barycenter of the piece could not coincide with the center of the piece itself. This means that knowing the center of the bounding box that we can know from the Python code, is not sufficient to grab the piece in the right way because in this way we completely ignore the moment of inertia. To grip the piece in the right way considering the barycenter and the moments of inertia so as not to drop it we have to use the integrated camera in the cobot. As we can see later on, the software that manages the camera includes some functions to find every time the barycenter. This software is called TM Flow and it is also useful because there is also a function to allow the communication between the Python code and the cobot itself making possible the sending of the center of the bounding box of the pieces.

7.2 TM Flow

TMFlow is the software associated to the Omron TM5-900. It is a graphical HMI that provides an interface for the control of the cobot. Through this HMI it is possible to write a logic program that the cobot must execute. This is possible because TMFlow provides a graphical flow chart with blocks with which we can write the logic program that must be executed by the cobot. With these blocks it is possible to command the cobot, to set some parameters of the cobot, to move it and program some operations that the cobot must do [6].

7.3 All nodes used

When we open the TMFlow program the first node that we see is the Start node (*Figure 104*) and is put automatically inside the program. This is because it corresponds to the initial point of the program and with this node it is possible to start a new program. For this reason, it is not possible to move the node and it is not also possible to create another start node [6].

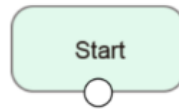


Figure 104: Start node [6]

Another node that we have used in the program flow is the Point node (*Figure 105*). This node is important because we have the chance to memorize the current position of the cobot, included the position and the coordinates of the TCP and the positions and the rotations of all the joints. To memorize the position of the cobot it is sufficient to click the Point Button in the end effector of the cobot. As shown in *Figure 106* it is possible to set the motion settings of the point node. This means that we have the possibility to choose the way, so the motion, through which the cobot goes in that position. As we can see in *Figure X*, we have three possibilities to choose the type of the motion. In the thesis it has been chosen the Line Motion [6].

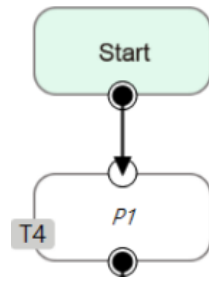
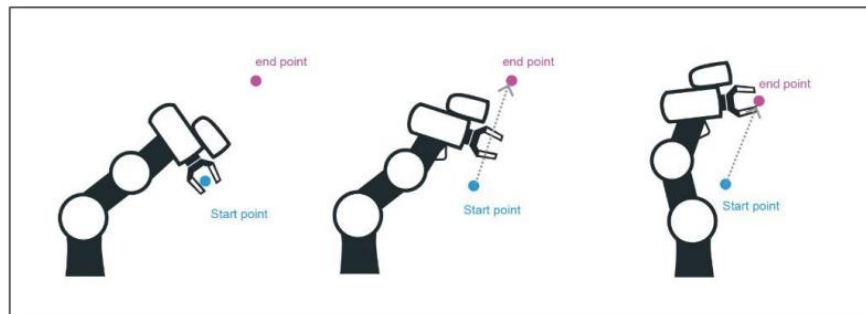


Figure 105: point node [6]

Figure 106: point node settings [6]

A representation of a line motion is shown in *Figure 107*. As it is possible to see the tool moves in a straight line at a specific speed. As a result, the line motion specifies that the motion from a start point to an end point is planned in a straight line [6].



Line

Figure 107: Linear movement of the tool [6]

Another node used in the thesis is the Move node (*Figure 108*). This node is useful to set the relative movement from the current position. This means that if we use the Move node

to move the cobot along the Z axis of 200 mm, the cobots moves down along the Z axis of the quantity of 200 mm with respect the current position of the cobot [6].

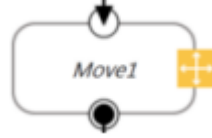


Figure 108: Move node [6]

An important node used in the program flow is the listen node (*Figure 109*). With this node is possible to establish a TCP/IP server and can be connected by an external device to communicate with the cobot. In our case the device is the workstation from which we run the Python code. We have used the listen node to send to the cobot the coordinate of the center of the bounding box of the piece to be taken [6].

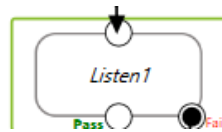


Figure 109: Listen node [6]

Another important node used in the thesis is the vision node (*Figure 110*). It is used to find and detect an object in the frame of the camera of the cobot [6].

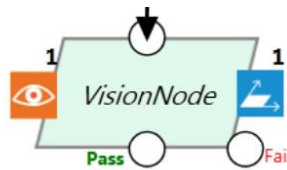


Figure 110: Vision node [6]

It is important to note that the cobot with the vision node records the relative relationship of objects by recording the points and lines of the object on a different vision base (*Figure 111*). When the environment changes and the object to be found will be in different position with respect the first time which we have taught the position of the object to the cobot, it can compensate this by the principle of the coordinate transformation without re-teaching the cobot's point positions. As shown in *Figure 111*, the point P1 is recorded on the vision base. This ensures that the cobot will detect and recognize the object in the frame of the camera also when the environment in the future will change. This principle is used in the thesis to approach to the piece that the cobot must take [6].



Figure 111: Vision base [6]

Another node used in the flow as we will see later, is the WaitFor node (*Figure 112*). It is useful to stop the project for some conditions, for example stop the project for a certain amount of time, and then continue to run the program after the set conditions are met. We have used this node by setting the time. This means that, the cobot will wait for that amount of time specified by us, after the time is passed the cobot can continue to do the operations indicated in the flow program and in the nodes [6].

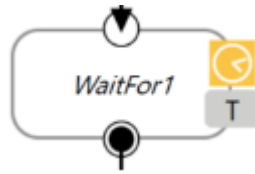


Figure 112: WaitFor node [6]

The ADG_V001_SET node (Figure 113) is used to set properly the Robotiq gripper of the cobot [6].

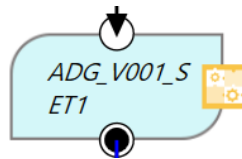


Figure 113: ADG_V001_SET node [6]

This node has included two nodes: the OK node and the Error node (Figure 114). This means that if everything in the settings is right without any errors, the program can continue, if there are errors the program goes in the Error node and the program can not continue since something goes wrong during the setting of the gripper [6].

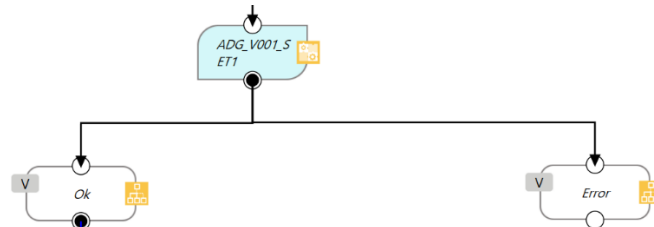


Figure 114: Ok node and error node of the settings [6]

We have also used the ADG_V001_OPEN (Figure 116) node to open the gripper of the cobot and the ADG_V001_CLOSE (Figure 115) node to close the gripper [6].

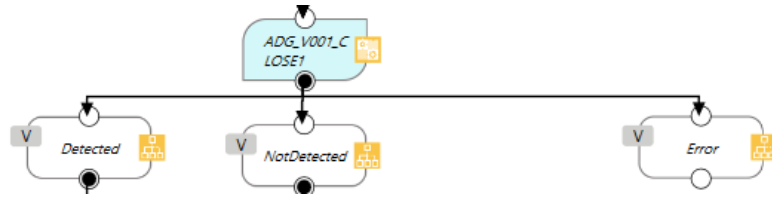


Figure 115: ADG_V001_CLOSE node [6]

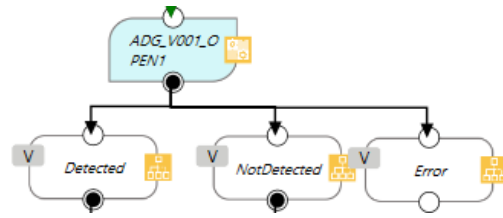


Figure 116: ADG_V001_OPEN node [6]

7.4 Flow program

To test the computer vision algorithm to recognize different mechanical piece and use it from the Omron cobot, we have implemented a case study of sorting. The goal is to define a sequence of different mechanical pieces that must be taken from the cobot and then the pieces are divided from the cobot in two different boxes to distinguish the round pieces from the squared ones. This operation of distinction is called sorting. Therefore, first of all, the external camera is activated to recognize the right piece in the sequence that must be taken from the cobot. After the algorithm detects the right piece to be taken from the cobot, then, the Python code run from the workstation, sends to the cobot the coordinates of the bounding box of the piece. The bounding box that detects the right piece is the output of the computer vision algorithm. Then, the cobot approaches to the piece thanks to the coordinates sent from the Python code. Therefore, the cobot can approach to the piece and can grab it. Then, depending on the type of the mechanical object, the cobot can distinguish if the piece is rounded or is squared and it can separate the two different types of pieces in two different boxes. The sorting, that is the case of study of the thesis, becomes

just an example to show that the cobot can take the right piece in the sequence without knowing its position a priori. This is because in an assembly purpose the cobot must follow a sequence of piece that must take without making any errors. For this reason, we have defined in the Python code a sequence of different pieces that the cobot must take. As shown in the comments below, we have defined the big flange with 3 holes as the first piece in the sequence to be taken from the cobot, instead the flange with 4 holes has been defined as the second one and so on.

```
# DEFINE THE SEQUENCE OF THE PIECE TO TAKE
# piece number 1 = 3 holes
# piece number 2 = 4 holes
# piece number 3 = piaster steel
# piece number 4 = piaster yellow
# piece number 5 = 2 holes

# initialize the piece to 1
piece=1
```

After initialized the variable piece equal to 1, in the code beyond the fact that the piece must be inside the workspace, we must be sure that the variable piece must be equal to the number associated to that piece that must be taken. For example, as shown in the comment above, if the cobot must take the yellow piaster, the variable piece must be equal to 4, instead if must take the flange with 2 holes, the variable must be equal to 5 and so on. We must verify these conditions with the command below:

```
# if the flange is with 3 holes (piece number 1) and inside the workspace:
if piece==1 and x1>=x1_workspace and x2<=x2_workspace and y1>=y1_workspace and
... .. y2<=y2_workspace:
```

After the cobot takes the first piece, it must continue to take the other pieces by respecting the sequence of them. If it has taken the piece number 1 associated to the big flange with the 3 holes, it will take the piece number 2 associated to the flange with 4 holes. To do that, for example, if the cobot has taken the piece number 1, the variable piece must be set equal to 2. In this way, in the next cycle, the cobot will take the piece number 2. The variable can be set in the following way:

```
# set the next piece to grab
piece= 2
```


Passing to the TMFlow software of the Omron TM, it has been created the flow shown in *Figure 117*. This flow means that, after starting the program with the node *Start*, the cobot must go in the zero-machine indicated as *NewOrigin*. After that, the cobot must wait for the run of the Python code with the node *WaitBeginningProgram*. Then, the cobot goes in the listen mode and as explained the *Listen1* node is useful to receive some data as the coordinate of the centre of the piece that the cobot must grab. Then we have added the *ADG_V001_SET1* in order to set the gripper of the cobot.

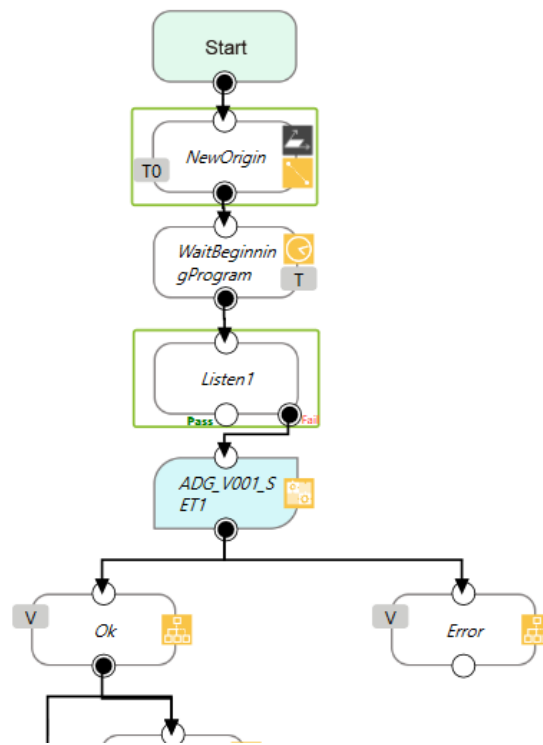


Figure 117: Initial flow

Going more in details, the node *NewOrigin* (*Figure 118*) has been set in the way shown in *Figure 119*. As it is possible to see, the movement to the zero machine is a line movement.

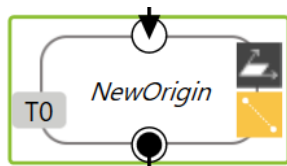


Figure 118: *NewOrigin node*

Node Name

Recorded on RobotBase ☐ ☐ T0

Motion Setting PTP Line WayPoint

Point Management Import from existing points Point Manager

Blending By Percentage By Radius No Blending

Advanced Setting Base Shift Tool Shift

☐ Change payload to kg

☒ Precise positioning

Figure 119: *Settings of the zero-machine node (NewOrigin)*

The node *WaitBeginningProgram* (Figure 120) has been set in the way shown in Figure 121, where the time to wait for the run of the Python code is equal to 5 seconds.



Figure 120: *WaitBeginningProgram* node

Figure 121: *set-up of the WaitBeginningProgram* node

Then, the *ADG_V001_SET* node has been set. From the window shown in the *Figure 122*, we have clicked to the *Activate_or_Not* button and the window in the *Figure 123* has been appeared. From this window, we have clicked the *Variables()* button and the window in *Figure 124* has been appeared.

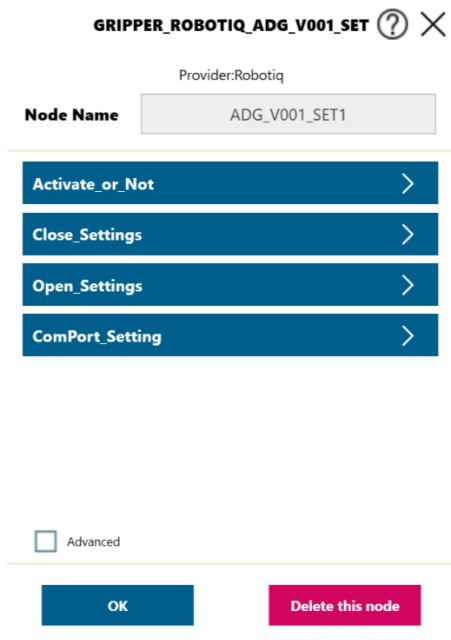


Figure 122: Set-up of the ADG_V001_SET node

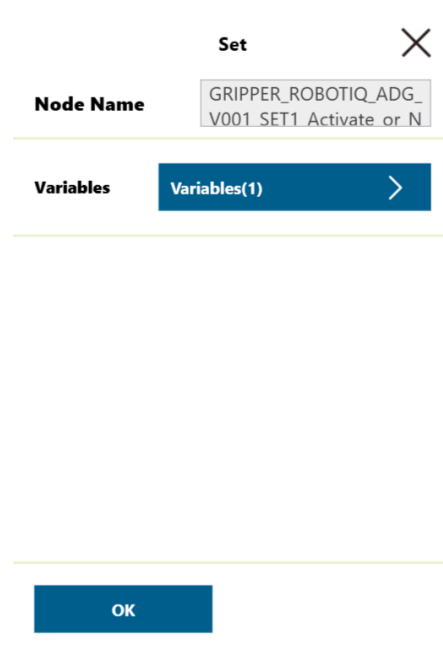


Figure 123: Activate_or_Not section of the set-up

From the window in *Figure 124*, we have set the bool variable equal to True. This is important to activate the gripper of the cobot.

Expression Editor Setting

GRIPPER_RO
BOTIQ_ADG_
V001_SET1_v = true
ar_reset

OK

Figure 124: setting of the variable *bool*

Expression Editor Setting

GRIPPER_RO
BOTIQ_ADG_
V001_SET1_v = "80"
ar_close_pos

GRIPPER_RO
BOTIQ_ADG_
V001_SET1_v = "50"
ar_close_spe
ed

GRIPPER_RO
BOTIQ_ADG_
V001_SET1_v = "50"
ar_close_forc
e

OK

Figure 125: Close_settings section of the set-up

Then, from the window in *Figure 122*, we have clicked the Close Settings button and we have set the three variables for the closing. The variables are the position, the speed and the force and have been set in the way shown in *Figure 125*. Instead, the settings of the same variables for the opening are shown in *Figure 126*.

Expression Editor Setting

Search: ▼

Add

GRIPPER_RO	BOTIQ_ADG_	V001_SET1_v	=	"0"
GRIPPER_RO	BOTIQ_ADG_	V001_SET1_v	=	"50"
GRIPPER_RO	BOTIQ_ADG_	V001_SET1_v	=	"50"

OK

Figure 126: *Open_settings* section of the set-up

Regarding the *Listen1* node, it has been set the Data Timeout equal to 3 seconds as shown in *Figure 127*. This means that, the communication between the cobot and the Python code finishes when the Python code does not send any data for 3 seconds. This time ensure the correct sending of the centre of the piece as data to the cobot. In this way the cobot goes in the coordinate corresponding to the centre of the first piece to take, so to the centre of the big flange with 3 holes.

If the setting of the gripper is ok, so after passing the *Ok* node from the *ADG_V001_SET*, the cobot wait for 2 seconds before going in the Vision Node. The vision nodes in *Figure 128* are called *Flangia4Holes*, *Flangia3Holes*, *Piastra2Holes*, *FlangiaSmall*, *PiastraSteel*. Each vision node represents the possibility from the cobot to find with its camera the flange with 4 holes with the *Flangia4Holes* vision node, the big flange with 3 holes with the *Flangia3Holes* vision node, the flange with 2 holes with the *Piastra2Holes* vision node, the small flange with 3 holes with the *FlangiaSmall* vision node and finally the piaster with the *PiastraSteel* vision node. Since the vision node is based on the shape of the piece and on the contours of it, the *PiastraSteel* vision node is also valid to detect with its camera the yellow piaster. Each vision node has as output two possibilities: Pass and Fail. If the program goes in the pass way, it means that the cobot has detected the piece indicated in that vision node (we will see later in details this fact). If the program goes in the Fail way, it means that the cobot has not detected the object indicated in that vision node. The flow program shown in *Figure 128* has the following logic: after the cobot is the position corresponding to the centre of the piece sent by Python code, the camera is activated, and it starts to find the piece in the workspace in that position. If the camera can detect the piece, then the program goes in the Pass way with the goal to grab that piece. If instead the camera can not find the piece indicated in a vision node, the program flow goes in the Fail way with the purpose to try to detect other pieces: for example, if the cobot must detect the flange with 4 holes and the camera can not find it in that position, the camera tries to detect the big flange with the 3 holes always in that position. This ensure that the camera of the cobot, in a fixed position sent by Python code, has the possibility to detect and find all the pieces considered. If instead the camera can find the piece indicated in a vision node, the program flow goes in the Pass way and as said, the goal becomes to grab that object. To reach this goal, in some cases the cobot must sets its position going in the point called *SetApproaching*, this ensure the fact that the cobot grab the piece at the right height since the piece could be very thin. Indeed, this problem is not present in the cases of the piasters and in the case of the flange with 2 holes since these pieces are thick enough. Then for each piece detected and found, the cobot goes in

the position to which the reference frame of the TCP coincides with the barycentre of the piece at the right height. This was done by taking the cobot's TCP and bringing it to the position where then, subsequently, if the gripper is closed, the cobot correctly grips the piece. After the cobot is in the position that we have set manually, we have clicked the point button on the cobot and we have set as the base frame, the one created by the vision node. This ensure that if the piece is rotated or is in a different position with respect the position with which we have created the vision node, the cobot anyway grabs the piece always in the right way by aligning the centre of the TCP with the barycenter of the piece. After the alignment, the cobot must close the gripper to grab the piece and this has been done by adding the *ADG_V001_CLOSE1* node. After that, the cobot can moves up with the grabbed piece, and this movement has been set by adding the node *MoveUp*. As shown in *Figure 128*, the procedure of grabbing the piece and then move up, has been done for all the pieces that the cobot can find in the workspace. By going in more details in the vision node, we have created a new one by clicking in the plus green icon indicated in the *Figure 129*. After that, we get the window shown in *Figure 130* with the live video from the camera of the cobot. We get as example the fact that the cobot must detect and find the flange with the 4 circumferential holes. Therefore, we have clicked in the *Task Designer* icon shown in *Figure 130*.

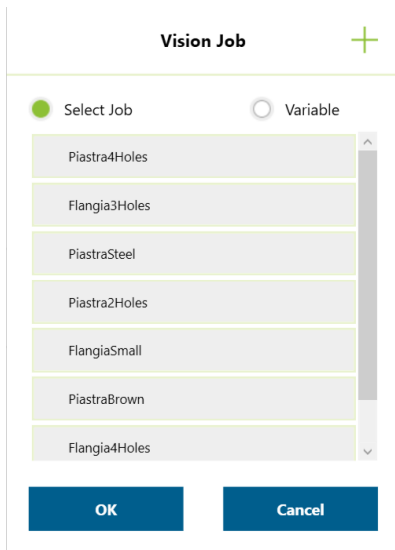


Figure 129: Vision Job window

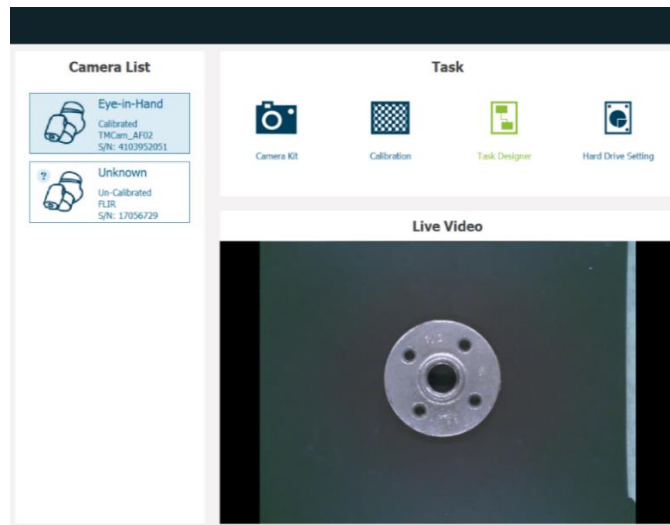


Figure 130: Live video from the camera of the cobot in the vision task

Then, we have clicked from the *Task Designer* icon, the *Visual Servoing* icon indicated in *Figure 131*. By doing this, we get the window shown in *Figure 132*. By clicking the icon green indicated on the top in *Figure 132*, we have clicked then in the Pattern Matching (shape) indicated in *Figure 133*.

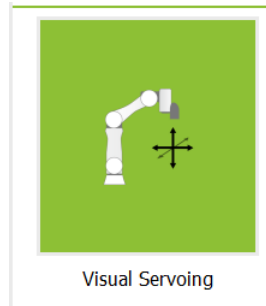


Figure 131: Visual Servoing icon

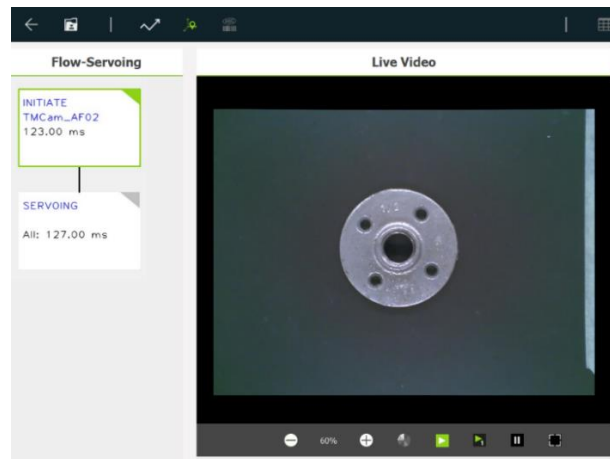


Figure 132: Window shown by clicking the Visual Servoing

Therefore, the window in *Figure 134* has been appeared. This is important to choose the object to find and detect in the future in the workspace from the camera of the cobot.

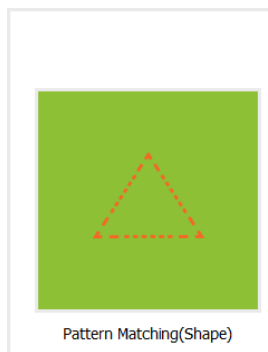


Figure 133: Pattern Matching icon

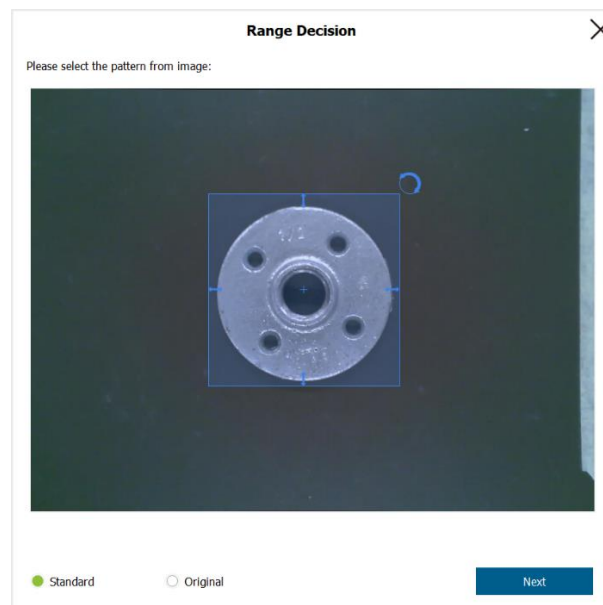


Figure 134: Selection of the pattern

Then, by clicking Next in the window shown in *Figure 134*, another window appears shown in *Figure 135*. The software of the camera automatically set the center of the blue

bounding box that we have chosen previously with its reference frame. Moreover, the software traces some blue lines in the pattern. This means that, in the future if the cobot find the lines traced the first time in the vision node, it automatically finds the object associated to those blue lines and in this case the object is the flange with the 4 circumferential holes. It can happen that for the presence of the light there are some unusual blue lines that can be removed manually by editing the pattern by clicking Edit Pattern in the *Figure 135*. With that command we have passed from the unusual lines shown in *Figure 135* to the simple blue lines shown in *Figure 136*. This simplifies the finding of the piece from the camera. Moreover, to simplify more the work done by the camera it is possible to set other parameters like the number of the pyramid layers and the score shown in *Figure 136*: lower are these values and in a simpler way (with less probability) the camera will find the flange.



Figure 135: Bounding box around the object after selected the pattern

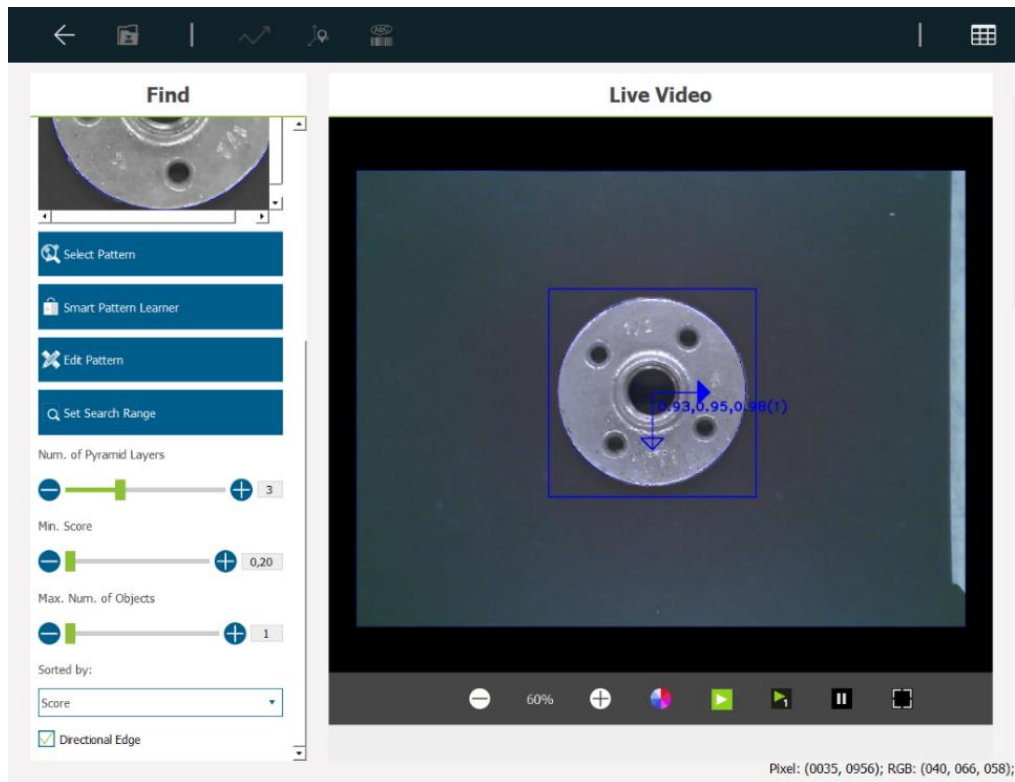


Figure 136: Pattern edited

After set the pattern and the way with which the camera must find the object from its frame, we get on the left the flow shown in *Figure 137*. After that, we have clicked the SERVOING block and the setting from the cobot and the camera has started automatically.



Figure 137: Flow Servoing

In this way we have obtained the fact that in the future, the cobot will recognize the object even in a different position in the frame of the camera with respect the initial one used to create the vision node. When it appears the red box as shown in *Figure 138*, it means that the servoing is finished. What happens in the future is the fact that the blue bounding box of the pattern could be in a different position as said, but the red one is fixed, and it will be fixed also in the future. The cobot when recognize the piece, with the help of the camera will allign the frame of the blue bounding box with the frame of the red one. This is important since the positions *ApproachingFlangia4Holes*, *ApproachingFlangia3Holes*, *ApproachingPiastra2Holes*, *ApproachingFlangiaSmall* and the *ApproachingPiastraSteel* are set by using the reference frame of the red box. So, the cobot will move to the position indicated by the red box, that when it is set the first time in the vision node, it will be remains fixed in the frame of the camera. When we create the vision node as shown in *Figure 138* as a result, we have that the blue box coincides with the blue one and also their frames.

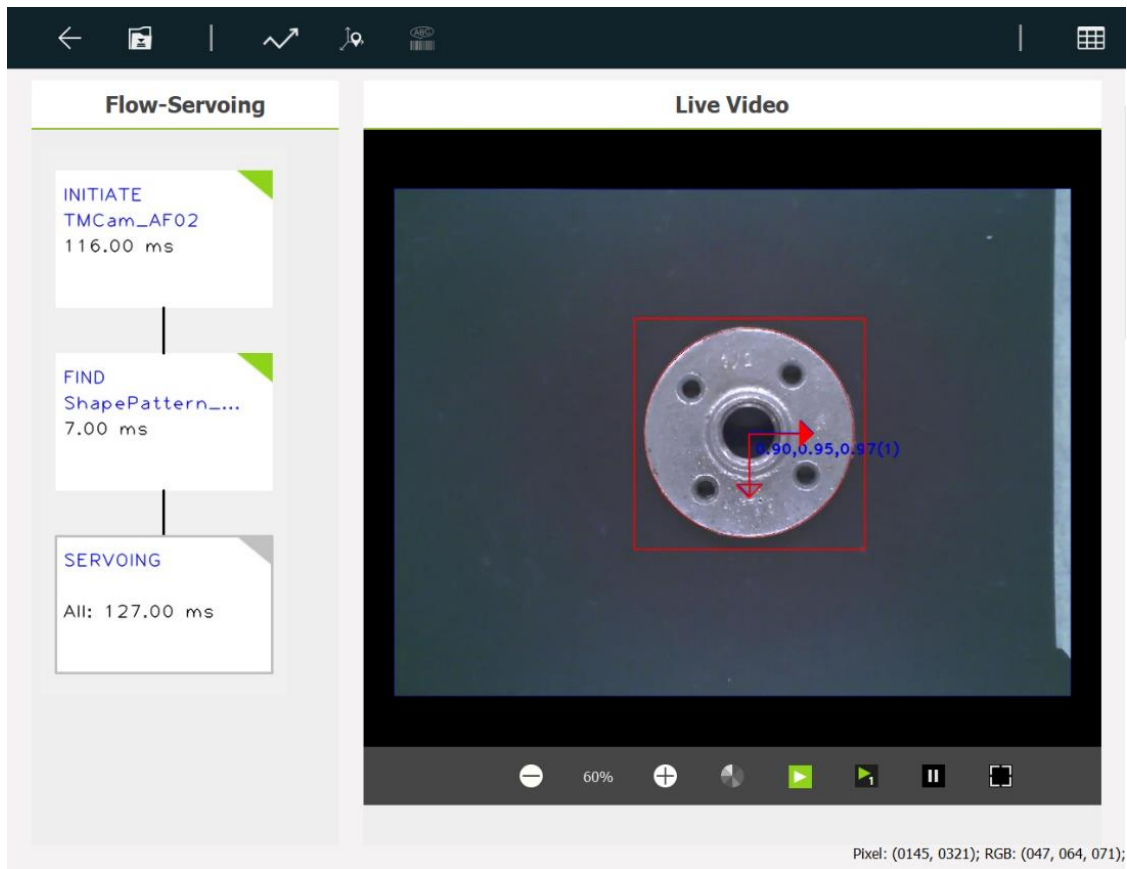


Figure 138: Bounding box after the servoing

But, as said previously, if the piece in the future will be in a different position with respect the position of the first time when we create the vision node, it means that the red box and the blue box (which really detect and indicate the object) do not coincide. This problem is solved because in the future the software of the camera will always align the red box with the new blue one and also their frames. In this way the cobot goes always in the right way.

In the *Figure 139* is shown the continuing of the program flow after the piece has been grabbed from the cobot and then the cobot goes up with the piece. Then, with the *ReleasePositionRound* node, the cobot goes with the piece in a position where it has to release the piece. As explained later, the cobot after it recognizes the piece and grabs it, it must distinguish if the piece is a flange or a piaster by executing a sorting task. Therefore, if the cobot grabs a flange it goes to the position dedicated to the round piece called

ReleasePositionRound point node. After that, the cobot goes down and this has been done by adding the *MoveDown* node. Then, the cobot opens the gripper with the node *ADG_V001_OPEN1*. With the node *MoveUp* the cobot goes up and then it returns to the zero machine in order to take the next piece respecting the sequence. After it goes to the zero machine, the cobot is ready to receive the next coordinate of the centre of the next piece, so it is needed another listen node called *Listen2* that can be seen in the *Figure 139*.

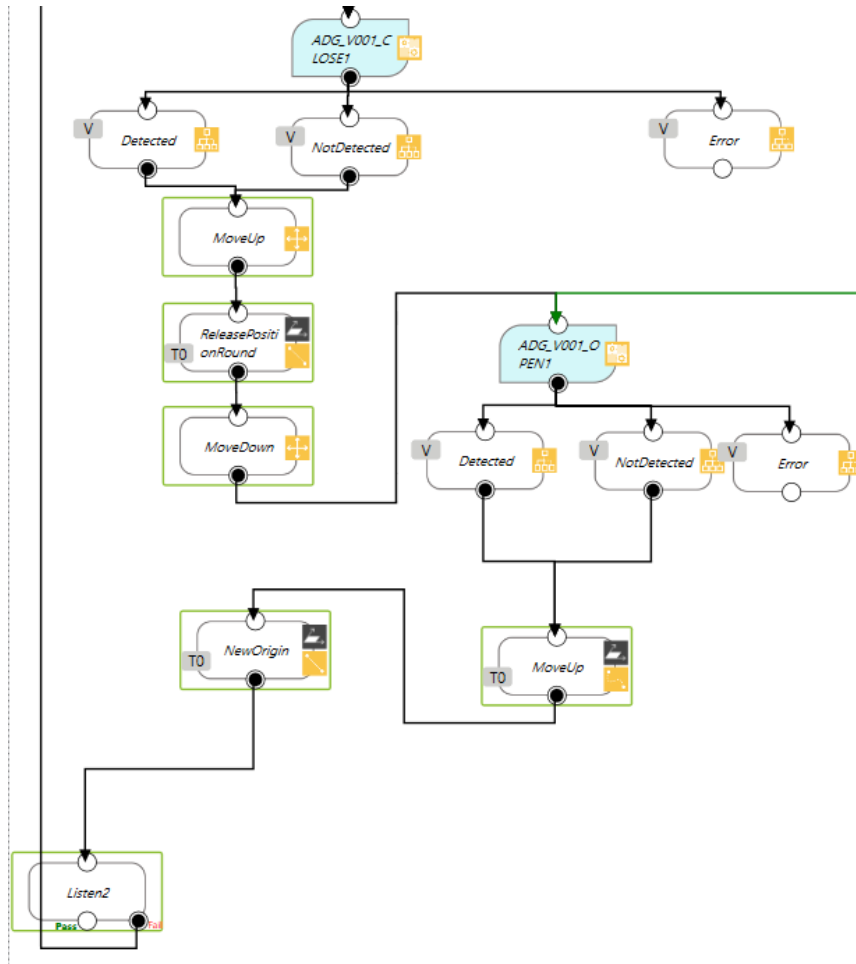


Figure 139: Flow program to grab the round object

The same discussion done for the round pieces it has also been done for the piasters. In this case the position to which the robot must go to release the piece is called *ReleasePositionPiaster* as shown in *Figure 140* and is different from the position where

which the robot releases the round piece because as seen later, the cobot does a sorting putting the round pieces and the piasters (squared pieces) in two different types of boxes.

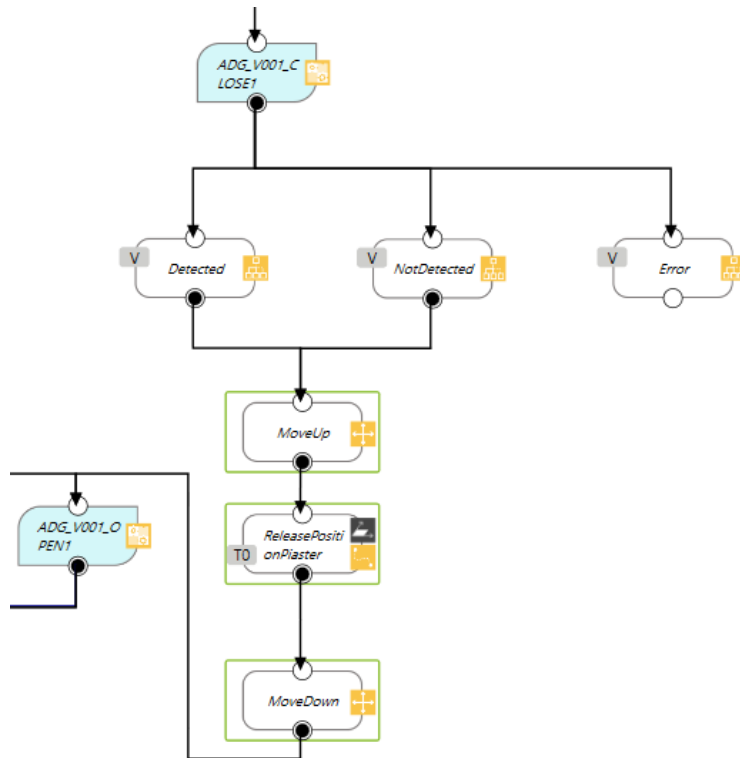


Figure 140: Flow program to grab the piasters

At the end as shown in *Figure 141*, after the cobot receives and goes to the new coordinate of the center of the new piece, the flow program starts again from the *WaitFor2* node, and then the cobot starts to detect and finds the new mechanical piece in the sequence.

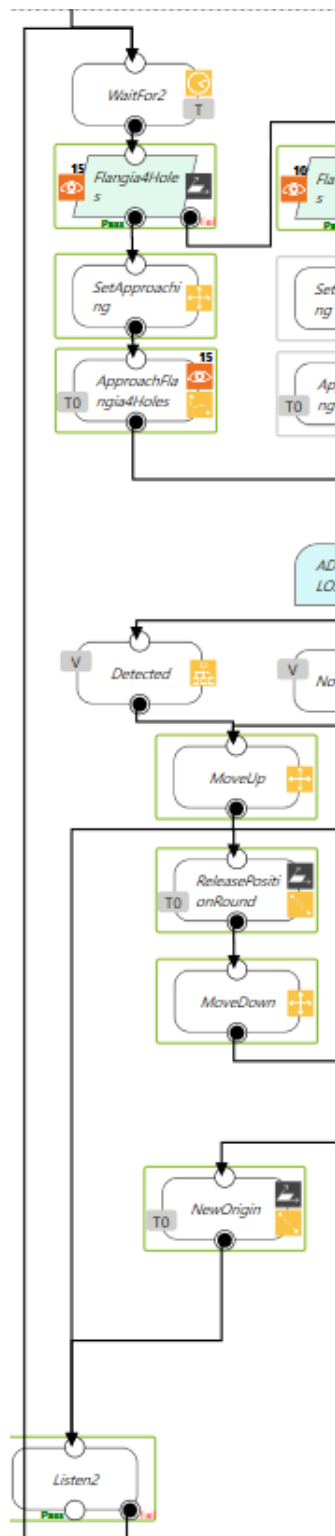


Figure 141: Loop of the flow program

8. Sending the coordinates through RoboDK

8.1 Connection to the cobot and zero-machine

First of all, it has been imported the model of the Omron TM5-900 that is used in the thesis as shown in *Figure 142*. To save the zero machine also in RoboDK, so the position from which we send the coordinate of the piece to the cobot, we have to establish a connection. To do that, as shown in *Figure 143*, we have inserted in the *Robot IP/COM* field the IP of the Omron TM5-900. In the similar way we have inserted the number of the port in the *Robot port* field. After setting the IP and the port of the robot we have clicked the Connect button that is circled as shown in *Figure 143*. In this way it appears the Connection status as Ready as shown in *Figure 143* and this means that we are connected to the cobot, and we can send commands to it and it is also possible to get the real position of the cobot. Getting the real position of the cobot means that we can replicate in the digital model of RoboDK the same joints positions and the same TCP position of the real one. Our intention is to memorize the position of the zero machine with all the joints and TCP positions and this is possible by getting the position and creating some frames. To get the position of the zero-machine of the real cobot, first of all, the real cobot must be in the zero machine and then we have clicked the Get Position command shown in *Figure 143*. In this way the digital model of the RoboDK replicates the joints and the TCP positions of the real one [50][51].

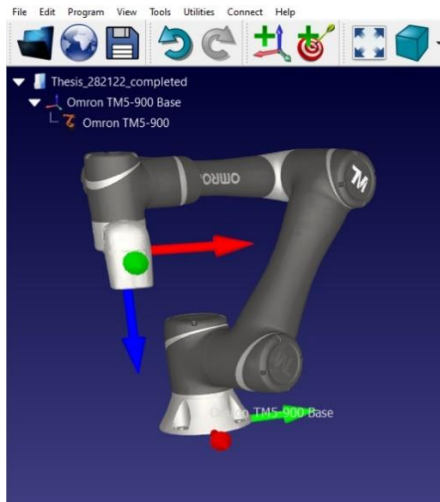


Figure 142: digital model of the Omron TM5-900 imported

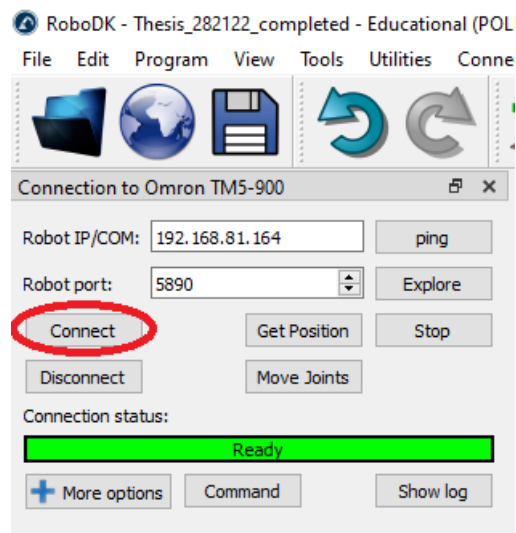


Figure 143: Connection to the robot

After getting the position of the zero-machine we have created a Frame that coincides with the frame of the TCP in the zero-machine position. The frame created is shown in *Figure 144* and it has been called Frame 2 to differentiate that from the Base frame of the cobot. The position and the rotation of the Frame 2 of the zero machine with respect to the base frame of the cobot is shown in *Figure 145* [50].

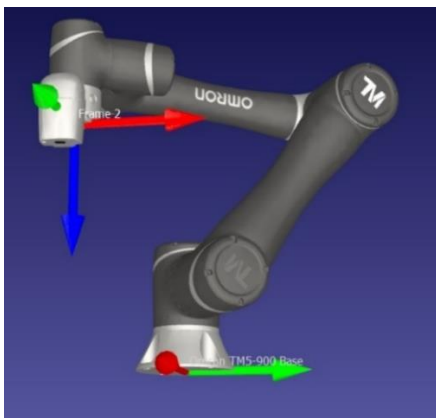


Figure 144: Frame 2 (Zero-machine) created

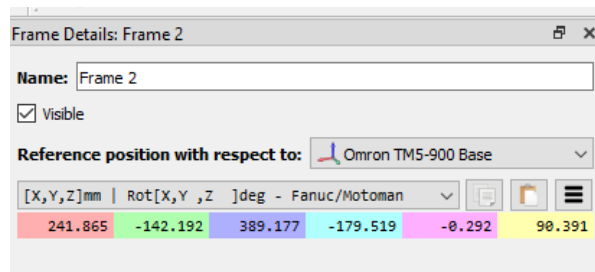


Figure 145: Rotations and positions of the Frame 2 with respect to the Omron Base

The *Figure 146* shows how is collocated the Frame 2 with respect to the base of the cobot.

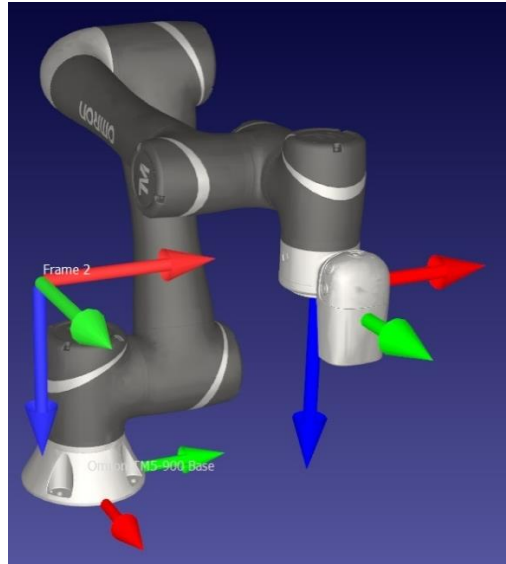


Figure 146: Visual position of the Frame 2 with respect to the frame base of the cobot

To send the commands to the cobot we need a target. The target is the position from which the cobot starts to move. It means that whatever positions of the TCP we send to the cobot, the positions are always referred with respect to the target. Since the positions sent are the centre of the piece with respect to the zero-machine, the target must coincide with the position of the TCP in the zero-machine, so the target must coincide with the Frame 2. For this reason, all the rotations and translations of the target, called Target 1 with respect to the Frame 2 are equals to 0 are shown in *Figure 148*. The *Figure 147* shows the position of the Target 1 with respect to the Base frame that is equal of the position of the Frame 2 with respect to the base frame and is also shown in *Figure 149*.

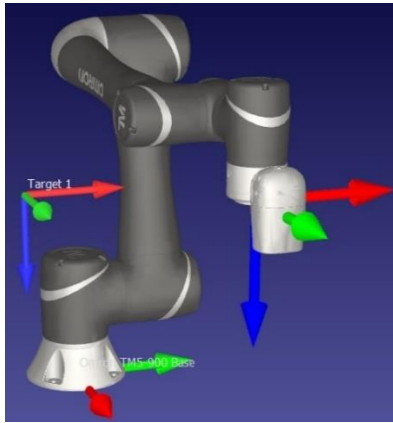


Figure 147: Target 1 created

Name: Target 1

☒ Visible Move to target Teach current position

Target type

☒ Keep cartesian position

☐ Keep joint values

Target position with respect to: Frame 2

[X,Y,Z]mm | Rot[X,Y,Z]deg - Fanuc/Motoman

0.000	0.000	0.000	0.000	0.000	0.000
-------	-------	-------	-------	-------	-------

Robot: Omron TM5-900 Change config.

Robot joints:

123.81	37.851	-125.01	-3.343	-90.24	-56.577
--------	--------	---------	--------	--------	---------

Figure 148: Rotations and positions of the Target 1 with respect to the Frame 2

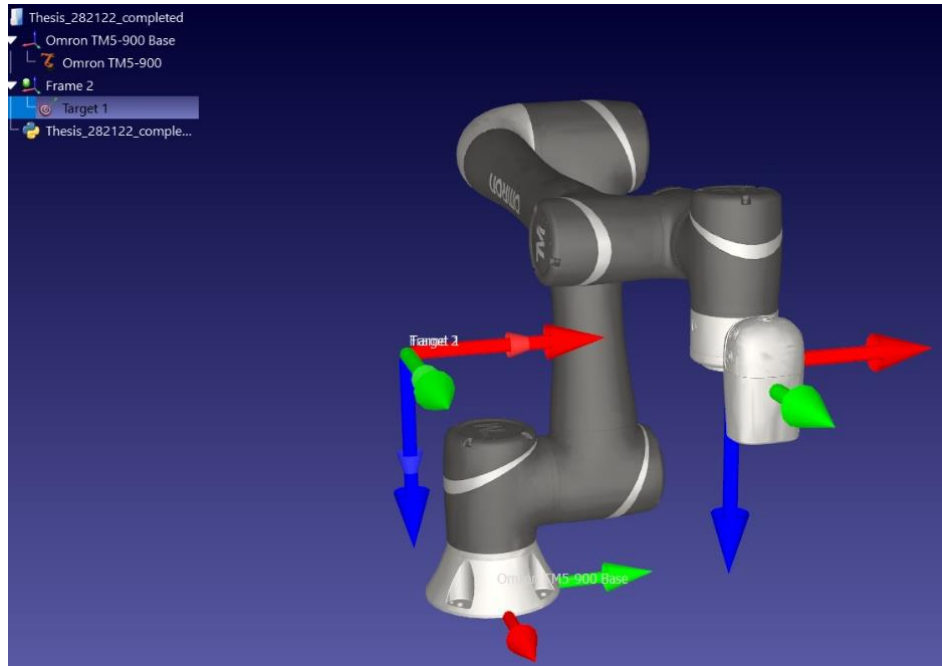


Figure 149: Target 1 and Frame 2 are coincident

The Figure 150 shows the digital model of RoboDK in the zero-machine position.

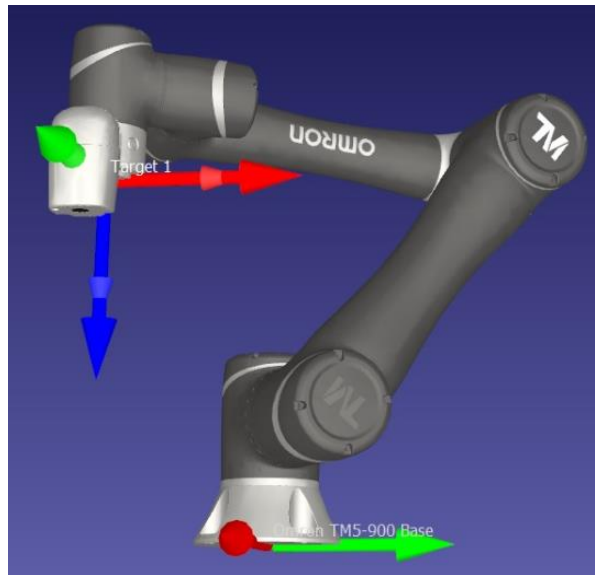


Figure 150: digital model of RoboDK in the zero-machine position.

8.2 Code to sending the coordinates to the cobot

The first thing that we have to do in the Python code is to import the libraries related to RoboDK and this has been done in the following way [53]:

```
# import the roboDK Library
from robolink import *
from robodk import *
from robodk.robolink import *
```

To start the API of the RoboDK has been used the command below. This ensures the work of the Python code in RoboDK in the right way [53].

```
# Start the RoboDK API
RDK= Robolink()
```

Then, it has been defined the Omron cobot as item with which we can do some operations. To define the cobot as item we have used the command below [53]:

```
# define our robot Omron as a RDK item where we can do whatever operations we want
robot = RDK.Item('', ITEM_TYPE_ROBOT)
```

To send the coordinates of the center of the piece, as said before, the cobot must be in listen mode using the Listen node in the TMFlow in order to receive the coordinates from the Python code. But this is not sufficient, since the Python code can send the coordinate to the cobot if it is established a connection with the cobot, after that the Python code can send commands to the cobot. The Python code must send the coordinates when the camera has recognized the right piece in the sequence and the cobot must go there. For this reason, we have to establish a connection entering in the following code:

```
# if the flange is with 3 holes (piece number 1) and inside the workspace:
if piece==1 and x1>=x1_workspace and x2<=x2_workspace and y1>=y1_workspace and
... .. y2<=y2_workspace:
    # memorize the actual coordinate of the centre of the piece
    coord_x=(x2-x1)/2 + x1
    coord_y=(y2-y1)/2 + y1
    coord_x_mm=(93*coord_x)/80 - x_base
    coord_y_mm=(93*coord_y)/80 - y_base - coord_camera_y
```

To establish a connection with the real cobot we have used the command `robot.setConnectionParams('192.168.81.164', 5890, '/', 'anonymous', ' ')`, to set the parameters of the connection. Indeed, the number 192.168.81.164 is the IP of the cobot and the number 5890 is the port of the cobot, the other parameters are instead set by default. Then, it has been used the command `robot.ConnectSafe('192.168.81.164', 2, 1, None)` to establish the real connection with the robot. The number 192.168.81.164 is again the IP of the real cobot, the number 2 represents the number of times, so the attempts, that the RoboDK API tries to connect to the cobot and finally the number 1 represents the time in seconds for which the API tries to connect for one single time and the last parameter is set by default. This procedure is like which we have done in the 7.1 paragraph, but in this case the connection is done inside the code. Anyway, with the two processes we can obtain the same result, so the connection with the cobot. The command `robot.ConnectSafe('192.168.81.164', 2, 1, None)` gives as output the number 0 or the number 1. The number 0 means that the connection with the cobot is established successfully, instead the number 1 means that

there was not any possibility to connect to the cobot and so there is not any connection with the cobot. So, when we use the command `robot.ConnectSafe('192.168.81.164', 2, 1, None)` we have saved the output in the variable `status`. The process of the connection described so far it has been done with the following commands [52][53]:

```
# Connect to the real Robot Omron TM
robot.setConnectionParams('192.168.81.164',5890,'/', 'anonymous','')
status=robot.ConnectSafe('192.168.81.164',2,1,None)
```

If the output of the connection is 0, so when we are connected to the cobot, it can be possible to send the coordinates of the center of the bounding box of the piece. To do that, it is necessary, as previously said, to define a target associated to the Omron cobot. As shown in paragraph 7.1 the target is called ‘Target 1’. These operations have been done with the following commands [52][53]:

```
If status==0:
    # Get the reference target by name:
    target = RDK.Item('Target 1')
    target_pose = target.Pose()
```

After the connection is established and the target is defined, we can finally send the coordinates of the center of the bounding box. First of all it has been defined the position to where the TCP must go and this has been done with the command `target_pose.setPos([coord_x_mm, coord_y_mm, coord_z_mm])`. The movement has been sent with the command `robot.MoveJ(target_pose)`, this means that the cobot will move to the position of the TCP defined previously. The commands just described have been inserted into the code in the following way [52][53]:

```
# Move the robot to the coordinate of the center of the piece
target_pose.setPos([coord_x_mm, coord_y_mm, coord_z_mm])
robot.MoveJ(target_pose)

# set the next piece to grab
piece=3

# set a delay of 1 second
time.sleep(1)
```

After we have sent command to the cobot and it starts to move we can see in the RoboDK window something similar shown in *Figure 151*.

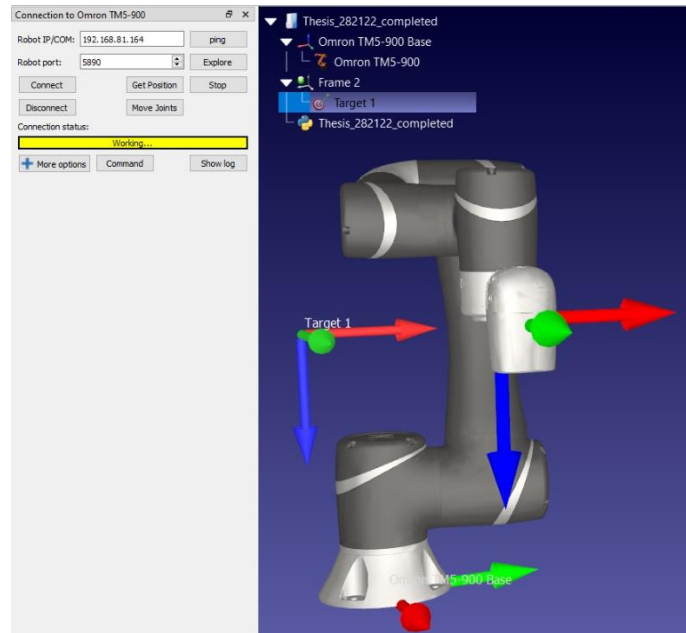


Figure 151: Working mode of the digital model

After the cobot has correctly received the coordinates of the center of the bounding box, we have to disconnect from the robot in order to find in the workspace the next piece of the sequence that the cobot must take. This operation has been done by checking when the status of the connection is 0 and disconnecting from the cobot with the command `robot.Disconnect()`. The output of this command gives 1 and we have saved the status of the connection in the `status` variable in order to not enter again the while cycle and finally continue with the next operations that are finding the next piece and send the new coordinates of the center of the bounding box of the new piece. All the commands just described have been inserted into the code in the following way [52][53]:

```
# Disconnect from the robot
while status==0:
    status=robot.Disconnect()

# set a delay of 2 seconds at the end of the process
time.sleep(2)
```

Conclusions

By running the full Python program, is possible to see that the work should be seen from three different parts: object detection, approach to the piece and finally the precise grabbing. As said, the object detection includes both the classification and the localization. This means that with the classification is possible to know if there is or not a precise piece inside the workspace, instead the localization is useful to know where exactly this piece is. The object detection has been done with an external camera since in this first part it is important to know the presence or not of a precise piece and for this reason is needed to have an overview of the entire workspace. This goal has been reached with a very simple and cheap industrial type system available on the market. The first important result has been reached: the algorithm has been implemented with a system that does not need to be costly and complex. The second result is that the algorithm can recognize a particular piece, for instance a steel piece, independently from the surface and its scratches and roughness and so from the precision of the material processing and some small defects. Moreover, we have to underline that identification done in the thesis is not an aesthetic identification, but it is a technological one, since the recognition is based on the recognition through technological features. The power of this algorithm is in the fact that it can detect also mechanical pieces that have not been inserted as images and so as input to train the model. In our example the flange with two circumferential holes has not been inserted as input for the training, but as seen in the final results the algorithm anyway can detect that. As a result, even if some pieces have not been inserted as input, they can have some technological features that are the same used by the algorithm to distinguish the mechanical piece: for example, the number of the circumferential holes. In this case the algorithm works also for these pieces and the advantage stays in the fact that for these pieces we do not need to train another model because the already existing model can recognize them without ever having seen them even once. The other important problem solved by this work is in the fact that in an assembly purpose, nowadays, the piece must be put in precise positions and the robots are programmed to go in those positions, but the

robot does not know if the piece is actually there or not. In this way, if the productions changes during the day, with this work does not need to change and the photocells and the assembly jigs when the production change, since the cobot can recognize and locate of the piece to be recognize independently from what production is it. The approaching to the right piece has been done by sending the coordinates to the cobot from the Python code connecting to the IP and to the port of the cobot as we have seen. The precise grabbing has been done through the camera at the edge of the wrist of the cobot and we have to underline that this cobot is one of the first commercially available cobots that has an integrated camera. With this camera it has been possible to do a precise grabbing by considering the barycenter and the moment of inertia of the piece. Some improvements and future works can follow the work done in this thesis. For example, it is possible to make the detection also from different views and not only from one view as done in the thesis. In this way it is possible to distinguish different pieces also from different views and consider as technological features those who are not possible to detect from the top view considered in the thesis. Moreover, the recognition may include other features that are not considered in this thesis obtaining an identification that includes a high variety of the mechanical pieces. It is also possible to obtain improvements in terms of performance by decreasing the losses and increasing the mean average precision. This goal can be reached by using different models of the YoloV as the YoloVl (large version) or the YoloVx (extra-large version) even if these models are heavier and these require more time for the training. Moreover, it is also possible to improve and continue this work by using the new versions of the YoloV (YoloV6 and YoloV7) which were released after the bibliographic study of the thesis had begun.

Sitography

- [1] <https://industrial.omron.it/it/products/collaborative-robots> , 12 October 2022
- [2] <https://www.kuka.com/it-it/prodotti-servizi/sistemi-robot/software/software-applicativo/kuka-equalizingtech> , 12 October 2022
- [3] <https://industrial.omron.it/it/products/RT6-0009001> , 12 October 2022
- [4] <https://www.mobileautomation.com.au/what-are-collaborative-robots/> , 12 October 2022
- [5] <https://wiredworkers.io/cobot/> , 12 October 2022
- [9] <https://www.ibm.com/cloud/learn/machine-learning#toc-machine-le-K7VszOk6> , 13 October 2022
- [10] <https://monkeylearn.com/machine-learning/> , 13 October 2022
- [11] <https://www.ibm.com/cloud/blog/ai-vs-machine-learning-vs-deep-learning-vs-neural-networks> , 13 October 2022
- [12] <https://www.ibm.com/cloud/blog/ai-vs-machine-learning-vs-deep-learning-vs-neural-networks> , 14 October 2022
- [13] <https://towardsdatascience.com/introduction-to-machine-learning-for-beginners-ee6024fdb08> , 14 October 2022
- [14] <https://towardsdatascience.com/introduction-to-machine-learning-for-beginners-ee6024fdb08> , 15 October 2022
- [15] <https://machinelearningmastery.com/object-recognition-with-deep-learning/> , 15 October 2022
- [16] <https://medium.com/analytics-vidhya/drivers-license-data-extraction-using-cnn-yolov5-14585709f4d8> , 17 October 2022

- [17] <https://www.ibm.com/topics/computer-vision> , 17 October 2022
- [18] <https://blog.roboflow.com/yolov5-improvements-and-evaluation/> , 17 October 2022
- [19] <https://www.analyticsvidhya.com/blog/2021/12/how-to-use-yolo-v5-object-detection-algorithm-for-custom-object-detection-an-example-use-case/> , 17 October 2022
- [20] <https://www.javatpoint.com/supervised-machine-learning> , 18 October 2022
- [21] <https://www.javatpoint.com/unsupervised-machine-learning> , 18 October 2022
- [22] <https://medium.com/@mehtapriyanka1pm/supervised-unsupervised-and-reinforcement-learning-246781f26730> , 18 October 2022
- [23] https://www.amazon.it/gp/product/B08LVVLCMV/ref=ppx_yo_dt_b_asin_title_o00_s00?ie=UTF8&psc=1 , 18 October 2022
- [24] <https://www.kdnuggets.com/2017/07/rapidminer-ai-machine-learning-deep-learning.html> , 19 October 2022
- [25] <https://vitolavecchia.altervista.org/differenza-tra-deep-learning-e-rete-neurale/> , 19 October 2022
- [26] <https://laptrinhx.com/future-prospects-of-deep-learning-in-medicine-687217971/> , 19 October 2022
- [27] <https://www.weka.io/blog/computer-vision-vs-machine-learning/> , 19 October 2022
- [28] <https://medium.com/ieee-women-in-engineering-vit/you-only-live-once-or-you-only-look-once-a9c6951bd82b> , 21 October 2022
- [31] <https://towardsdatascience.com/introduction-to-py-torch-13189fb30cb3> , 21 October 2022
- [32] <https://www.analyticsvidhya.com/blog/2021/04/a-gentle-introduction-to-pytorch-library/> , 22 October 2022
- [33] <https://www.python.org/doc/essays/blurb/> , 22 October 2022

- [34] <https://www.andreaminini.com/ai/machine-learning/matrice-di-confusione> , 23 October 2022
- [37] <https://towardsdatascience.com/the-practical-guide-for-object-detection-with-yolov5-algorithm-74c04aac4843> , 23 October 2022
- [39] <https://peltarion.com/knowledge-center/modeling-view/build-an-ai-model/loss-functions/mean-squared-error> , 23 October 2022
- [40] <https://peltarion.com/knowledge-center/modeling-view/build-an-ai-model/loss-functions/binary-crossentropy> , 23 October 2022
- [41] <https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e> , 23 October 2022
- [42] <https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/> , 26 October 2022
- [44] <https://pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/> , 26 October 2022
- [45] https://pytorch.org/hub/ultralytics_yolov5/ , 8 November 2022
- [46] <https://github.com/ultralytics/yolov5> , 8 November 2022
- [47] <https://pytorch.org/get-started/locally/> , 8 November 2022
- [48] <https://github.com/heartexlabs/labelImg> , 8 November 2022
- [49] <https://www.tensorflow.org/guide/tensor> , 8 November 2022
- [50] <https://robodk.com/doc/it/Basic-Guide.html> , 9 November 2022
- [51] <https://robodk.com/doc/it/RoboDK-API.html#PythonAPI> , 9 November 2022
- [52] <https://robodk.com/doc/en/PythonAPI/index.html> , 9 November 2022

[53] <https://robodk.com/doc/en/PythonAPI/robodk.html> , 9 November 2022

[54] https://favpng.com/png_view/cube-rgb-color-model-hsl-and-hsv-rgb-color-space-cube-png/Fr06aXV2 , 10 November 2022

[55] <https://github.com/nicknochnack/YOLO-Drowsiness-Detection> , 10 November 2022

[56] <https://github.com/CreepyD246/Simple-Color-Detection-with-Python-OpenCV> , 10 November 2022

Bibliography

- [6] Omron Corporation, *Software Manual TMflow*, Software Version: 1.68, Kyoto, Japan, 2018
- [7] Slides fornite da Omron Corporation, *S01_hardware, S02_Software e Licenze, S03_Procedura Startup, S04_Basic_Nodes, S06_Vision, S08_Safety 2*
- [8] Robotiq, Lean Robotics, *Robotiq 2F-85 & 2F-140 for OMRON TM Series Robots Instruction Manual*, 2019
- [29] Alexander Mordvintsev and Abid K, *OpenCV-Python Tutorials Documentation*, Release 1, 5 November 2017
- [30] Jan Erik Solem, *Programming Computer Vision with Python*, Gravenstein Highway North, Sebastopol, United States of America, O’ Reilly, 2012
- [35] Darío G. Lema, Oscar D. Pedrayes, Rubén Usamentiaga, Daniel F. García and Ángela Alonso, *Cost-Performance Evaluation of a Recognition Service of Livestock Activity Using Aerial Images*, Basel, Switzerland, Magaly Koch, 13 June 2021
- [36] Steven Kolawole, Opeyemi Osakuade, Nayan Saxena, Babatunde Kazeem Olorisade, *Sign-to Speech Model for Sign Language Understanding: A Case Study of Nigerian Sign Language*, 5th Workshop on Machine Learning for the Developing World at NeurIPS, 1 November 2021
- [38] Margrit Kasper-Eulaers, Nico Hahn, Stian Berger, Tom Sebulonsen, Øystein Myrland and Per Egil Kummervold, *Short Communication: Detecting Heavy Goods Vehicles in Rest Areas in Winter Conditions Using YOLOv5*, Basel, Switzerland, Puneet Sharma, 31 March 2021
- [43] Renjie Xu, Haifeng Lin, Kangjie Lu, Lin Cao and Yunfei Liu, *A Forest Fire Detection System Based on Ensemble Learning*, , Basel, Switzerland, Stelian Alexandru Borz, 13 February 2021

Acknowledgments

I sincerely thank Prof. Dario Antonelli, supervisor of the final exam, for guiding me in drafting the thesis through indispensable advice and for constantly supporting me in the final phase of my academic career.

I thank my father, my mother, my brother Davide, my sister Nicoletta and my two grandmothers for having supported me emotionally and financially over the years, remaining by my side especially in times of difficulty, without you it would not have been possible to achieve this objective goal.

I thank my girlfriend Elisa for supporting me over the years, she has always been there over the years especially in times of difficulty, she helped me to reach my goal by always supporting and motivating me, especially before each exam.

I thank my colleagues Luca, Pietro, Gabriel and William, no one better than them has been able to understand my anxieties, pains and fears over the years, continuously supporting and motivating me, which is why I wish them the achievement of my goal.

I thank my friends Alessio, Andrea, Angelo, Ciccio, Dario, Giuseppe, Ivan, Oliver and Pino for having supported me over the years especially in times of difficulty and for helping me to live my university career with greater serenity.

I thank Marco and Liu for having supported me in these last few months that I have dedicated to the work of my thesis and for this reason I wish them to reach the goal that I have achieved.

Thank you for being accomplices, each in their own way, in reaching this goal after an intense and tortuous journey. Thank you for making a very important milestone special for me.

