Group members:
Edoardo Bastianello, ID 2053077
Stefano Binotto, ID 2052421
Gabriele Boscarini, ID 2063145

REPORT - COMPUTER VISION PROJECT

********** INTRODUCTION **********

MAIN IDEA:
The idea to complete the project was to implement the object detection module part using a neural network and to segment the hands by considering only the regions of the images within the bounding boxes obtained as result of the neural network.

UTILS FUNCTIONS:
The following files contain the classes and functions we used to organize and condense the code.

Bounding_Box.cpp :
This file contains the Bounding_Box class. This class allows to:

- create an object of type Bounding_Box, passing as constructor parameters the x and y coordinates of the top left corner, the width and the height;
- get the coordinates of the four corners of the bounding box;
- get the width of the bounding box;
- get the height of the bounding box.

An object of type Bounding_Box is defined by the coordinates of its four corners.

visualize_results.cpp :
This file contains the following functions:

- visualize_image(...): this function displays an image on the specified window;
- draw_one_bounding_box(...): this function draws the bounding box passed as parameter on the image passed as parameter;
- visualize_results_bounding_box(...): this functions displays an image with its bounding boxes;
- color_masks(...): this function colors the masks of an image, each one with a different color.

read_test_dataset.cpp :
This file contains the functions to read the test images, the test bounding boxes and the test masks. In particular, this files contains the following functions:

- read_test_images(...): this function allows you to read all the images and the masks from their respective specified directories;

- read_Bounding_Boxes_one_img(...): function that reads the bounding boxes of an image from the specified txt file;
- read_test_b_box(...): this function allows you to read all the bounding boxes from the txt files with the bounding boxes from the specified directory.

network_utils.cpp:
This file contains the functions used to operate with the prediction model imported after the training process in https://colab.research.google.com. The file contains the following functions:

- preprocessing(...): this function is useful to convert and set up the input image in which we want to detect the bounding boxes;
- detect(...): this function finds the collection of all the bounding boxes detected in the image;
- postprocessing(...): this function is used to process the detection output of the model. It keeps only the most confident detections and applies the Non-Maxima-Suppression in order to refine the boxes.
- getBoundingBoxes(...): this function encloses all the steps of detection.

********** OBJECT DETECTION MODULE **********

THE CHOICE OF THE NETWORK TO BE USED:
We tried to analyze different approaches. There were different kinds of architecture available for transfer learning for object detection but there was also a big constraint, the hardware. Since we could not rely on our personal hardware for the tough training process, we had to work using the Google cloud environment, Colaboratory. The main problem with this platform is that we couldn't control every part of the system and so we couldn't easily solve the main configurations problem we got into. That's why at the end we decided to adopt the simplest but still accurate YOLOv5 architecture.

MASK R-CNN:
This neural network allows to obtain both the bounding boxes and the masks of an image. Our idea was to consider only bounding boxes and to segment the image with traditional opencv approaches. Later we would compare the segmentation obtained with opencv and that obtained with the network. For this, we prepared the code to perform Transfer Learning with our dataset. However, since YOLOv5 gave good results, and preparing the ground truth for Mask R-CNN required much more time, we eventually decided to use YOLOv5 instead. Therefore, we did not use Mask R-CNN for the project and we didn't do much testing with this network. However, Edoardo Bastianello wrote a python script to perform the training with Mask R-CNN, even if in the end it wasn't used for the project .

DARKNET - YOLOv4
We tried to implement a transfer learning algorithm for YOLOv4 but it was pretty hard to adapt the custom dataset for this particular kind of architecture. As soon as we found out

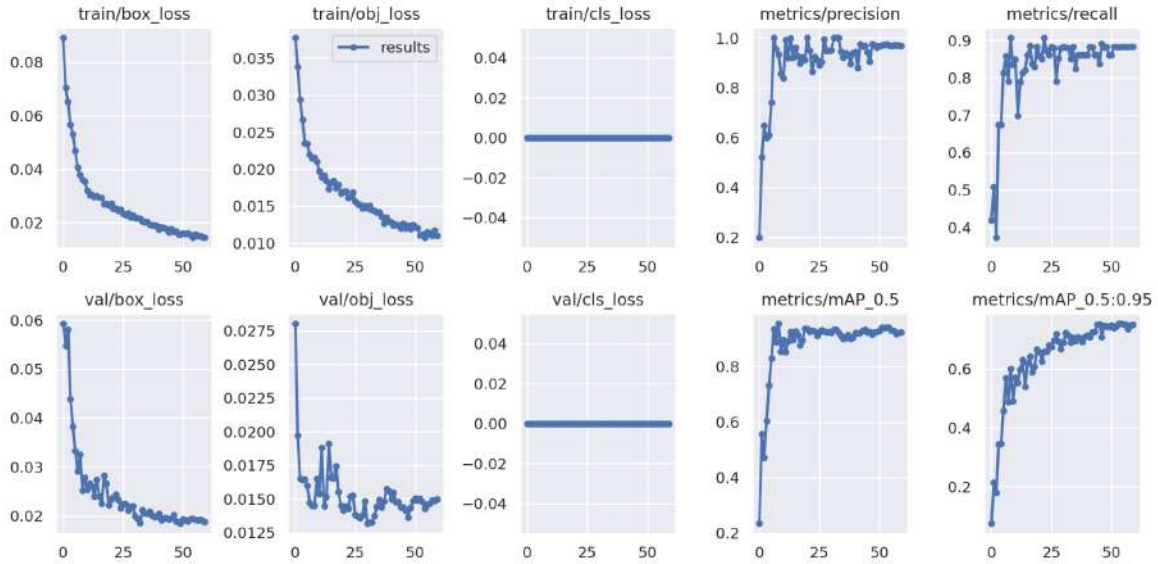
that there exists the YOLOv5 model which achieves the same accuracy performance as YOLOv4 but with a way more lightweight architecture, we interrupted the developing process for this network.

DARKNET -YOLOv5 :

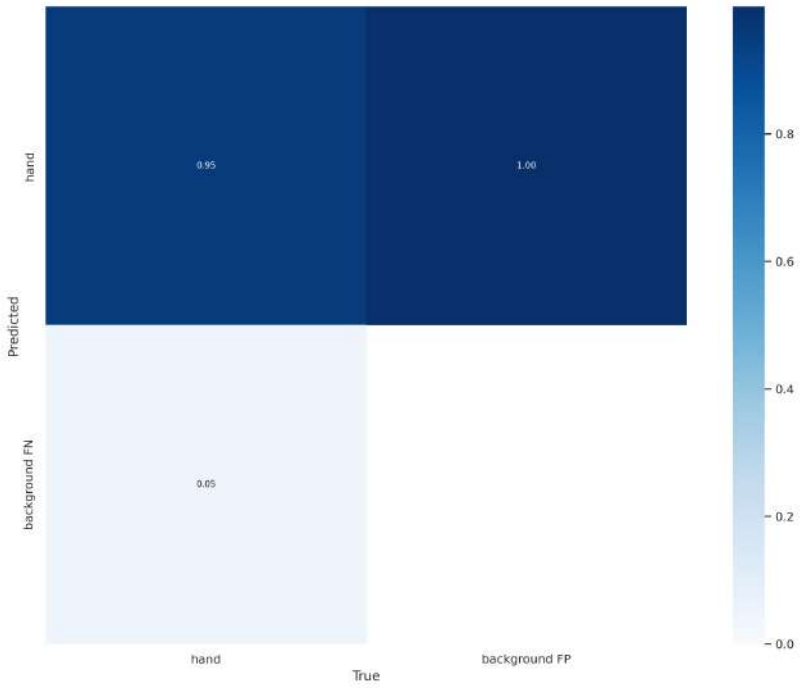
The YOLOv5 is a package of object detection architectures which are pre-trained with the COCO dataset and it's released by Ultralytics research team. In our case, we adopted this architecture because it's the most effective and simplest model we could train, since we do not own very powerful hardware and we had to work on Google Colab cloud machines. During the transfer learning phase we used the pre-trained model "yolov5m" as starting weights because it's the perfect balance between good accuracy and low complexity that allows us to fine-tune the network with a quite small training set.

The model was fine-tuned using three different datasets, with **453**, **505** and **540** images.

The training process with the biggest training set, the last one, gave the best performance:



The detections on some test images are the following:



LABELING SOFTWARE WE USED:

To prepare the ground truth for Mask R-CNN, we used CVAT, which allows you to draw the polygon to create the mask of the hand.

To prepare the final dataset for the YOLOv5 model, we used https://www.makesense.ai, which allows us to get a YOLO ground truth by drawing the bounding boxes around the hands. In the drive folder, whose link is provided in the next paragraph, there are the training and validation sets labeled by Stefano Binotto using this site.

THE DATASET :

The **training set** is composed as following:

1. 290 images from the HandOverFace(HOF) dataset
2. 139 images shot by Stefano and Edoardo
3. 52 images from EgoHands dataset
4. 34 images downloaded from Google Images
5. 25 background (no hands) images, used to decrease False Positive detections

The **validation set** is composed of 20 images found online.

Both training and validation sets don't include the test images. The dataset is available here: https://drive.google.com/file/d/1B4DFxi3NhfrCJVFZX9WnrT5qizKkck9K/view?usp=sharing.

MODEL USAGE :

After the development of the python code and the training phase with our custom hand dataset, we had to import the model in our C++ code. As we mentioned in the "Utils

Functions" part, we wrote the "*network_utils.cpp*" file with all the functions we needed for this purpose.
Using the "*export.py*" script of YOLOv5 we were able to convert the best model we got into an .onnx file. This particular type of file is easily readable by the dnn module available on OpenCV.
The model we built accepted only 640x640 input images so, by using the functions contained in "*network_utils.cpp*", we preprocessed the images. The "*preprocessing(...)*" consists in resizing the input image, normalizing the pixel values ([0,1] normalization) and swapping the blue and red channels, since OpenCV read the RGB images as BGR while the model developed with Python read them as RGB. Then we converted this image into a blob, which is a "Binary Large Object" accepted as input by the dnn model.
The output of the "*detect(...)*" function is a cv::Mat containing as many rows as the number of bounding boxes detected in the input image. Since the input image size is 640x640, there are 25200 total rows. There are 6 columns: the first two are the (x,y) coordinates of the center pixel of the box, then there is the width and the height of the box, the fifth column is the confidence associated to that individual detection, the last columns is the score related to the class hand (since this is only a one class detection we can ignore this value).

X	Y	W	H	Confidence	Class scores of 80 classes

In the "*postprocessing(...)*" method we used the confidence value in order to select only the best boxes and we adapted the boxes coordinates on the original (non-resized) image.
In this part, we had a problem during the postprocessing method in C++. In fact, when we had to visualize the boxes on the original image, we found out that all the boxes were a little bit shifted and scaled with respect to the visualization we had in the python code. We figured out why: since the images must be resized to 640x640 before entering the model, the detected boxes coordinates in output are related to the 640x640 images. When we convert those coordinates to the coordinates related to the original size of the images, the casting from float to int loses some information. However, the difference is minimal and negligible, as you can see in the comparison below.

Python: **OpenCV/C++**:

(The edge of the box on the index finger on the right image shows that the detection on OpenCV suffers from the loss of information due to the casting from float to int)

At the end, Non-Maxima-Suppression was applied to each image, in order to remove all the multiple overlapping boxes.

MODEL TUNING:

After the development of the code for the model part, we had to find the best possible parameters in order to achieve the most accurate detection. These parameters were defined as global variables at the beginning of the network_utils.cpp file.

There is the **CONFIDENCE_THRESHOLD** parameter defined to cut off all the boxes detected with not enough confidence. The parameter we set (= 0.21) is the one that allows us to detect the vast majority of the boxes (not every single one) without having to deal with False Positives.

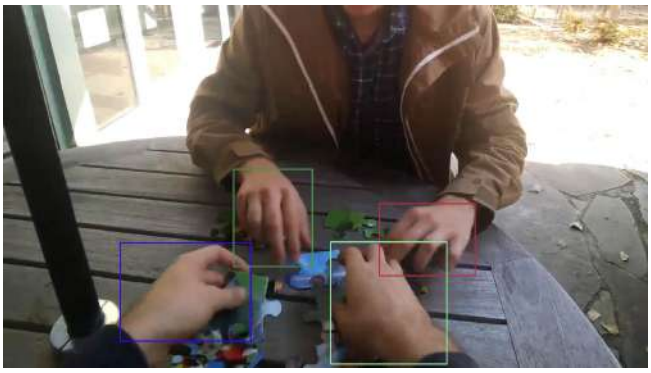
Then we had to find the **NMS_THRESHOLD** (= 0.425). Using the Non-Maxima-Suppression approach, we could refine the detection outputs, after the confidence thresholding, to delete the multiple overlapping boxes.

In this way, we were able to balance the trade-off between underdetection and overdetection very accurately.

VISUALIZATION FUNCTIONS:

To display the results of the object detection module, we used the function "visualize_results_bounding_box(...)" that can be found in the file "visualize_results.cpp". This function displays an image with its bounding boxes. Each bounding box has a different color.

Here is an example:



OBJECT DETECTION MODULE PERFORMANCE EVALUATION - Intersection over Union :

The Intersection over union is used to obtain a score indicating the quality of the detected bounding boxes. To compute the Intersection over Union values of an image, the compute_IoU_of_one_img(...) function of the file "Intersection_over_Union.cpp" was used. In particular, given an image, this function takes as parameters its predicted bounding boxes and its corresponding test bounding boxes.

As well as calculating the IoU scores, this function solves the two following problems:

1. *For a given image, which of its test bounding boxes a predicted bounding box corresponds to?* By defining the origin of a Bounding Box as its top left corner, a predicted Bounding Box corresponds to the test Bounding Box with minimum distance between their origins. By associating the predicted bounding box with the test box in this way, we observed that all matches were correct.

2. *What if, for a given image, our network finds a number of bounding boxes lower or greater than the number of bounding boxes of the test dataset?*
   - if the number of predicted bounding boxes is lower than the number of test bounding boxes, then the function returns one value of IoU for each predicted bounding box and 0 for each non predicted bounding box that is in the ground truth.
   - if the number of predicted bounding boxes is greater than the number of test bounding boxes, then the function returns one value of IoU for each predicted bounding box and 0 for each extra bounding box.
     In this way, we were able to properly handle cases where the network did not find all the bounding boxes or found too many of them

Here is an example where the neural network didn't detect one of the hands:

IoU = 0.727719
IoU = 0

********** SEGMENTATION MODULE **********

MAIN IDEA:
To segment the hands, our idea was to only consider the regions of the image within the bounding boxes detected by the network.
This allowed us to focus only on the regions inside the bounding boxes detected by the neural network, instead of having to consider the whole image. In this way, we were able to obtain more accurate results. However, a downside of this approach is that, if the network does not detect a hand, then the hand will not be segmented.

FIRST APPROACHES:
1) THRESHOLDING IN DIFFERENT COLOR SPACES;
2) K_MEANS;
3) INTERSECTION OF WATERSHED WITH A BINARY IMAGE.

1)THRESHOLDING IN DIFFERENT COLOR SPACES :

The first approach to segment the hands was to try to use different thresholds in multiple color spaces. In particular, the goal was to find the best parameters to detect the skin in each color space.

The first color space that was tried, and also the most promising, was YCrCb. The best results were obtained with the following parameters:

- Y in range [0, 255]
- Cr in range [133, 173]
- Cb in range [77, 127]

The most important thing one could observe was that, in almost all the images, the result of thresholding in this color space had all the pixels with the hand plus a few extra pixels. This fact was exploited a lot for the final segmentation algorithm.

The second color space that was tried is BGR. The best results were obtained with the following parameters:

- B in range [20, 255]
- G in range [40, 255]
- R in range [95, 255]

The last color space that was tried is HSV. These are the best parameters that were found:

- H in range [0, 25]
- S in range [0.28*255, 0.68*255]
- V in range [0, 255]

However, both the BGR and the HSV color spaces were affected by the lighting condition way more than the YCrCb color space.

Here is an example:

From left to right: *original image, results of thresholding with YCrCb, BGR and HSV color spaces*

The main problem with this approach is that, if there are other regions with skin in addition to the hand, then these regions will obviously be detected as well. Another problem is that, in general, color-based thresholding is greatly influenced by lighting conditions. Also, if an object that is not a hand has about the same color as the skin, then it will be detected along with the hand. Therefore, it is clear that thresholding based on color spaces cannot adapt to many situations and therefore it generally doesn't work well.

For these reasons, thresholding based on color spaces does not work well for the last 10 images (since there is also the face in the image), while Otsu performed better. However, on the first 20 images, color thresholding generally outperformed Otsu.

Example of one of the last 10 images, where the thresholding based on color spaces gives bad results:

From left to right: *original image, results of thresholding with YCrCb, BGR and HSV color spaces*

2) K_MEANS

This method has been chosen because of its simplicity and because the number of classes was known in advance.

In this work, this technique can be found in the file “kmeans_clustering.cpp”.

The file contains 4 functions, one performs the k-means segmentation, and the remaining are used to solve the problem of inverted classes, which will be discussed in the following.

The OpenCV kmeans function takes as input a vector called "samples" that in this case was the cv::Mat image, so this method performs the segmentation based on the color and the intensity of each pixel of the cv::Mat image.

Considerations on the results:

As a first approach, the k-means method has been tried on an image without pre-processing it. In this way, the first problems appeared, in fact, the segmentation was very sensible to noise and illumination variations. Some pre-processing techniques have been tried: first, a Gaussian filter has been used to smooth the edges and reduce the high frequency noise (for example, the variation in illumination caused by the lines between two fingers). The kernel of the Gaussian filter has been chosen based on empirical results trying to reduce as much noise without losing too much information on the image. In **Figure1,** it is presented as an optimal example of an image disturbed by a change in lighting:

Figure1: image to be segmented.

Figure2: segmentation result without pre-processing (the white pixels are the hand class, the black pixels are the background class).

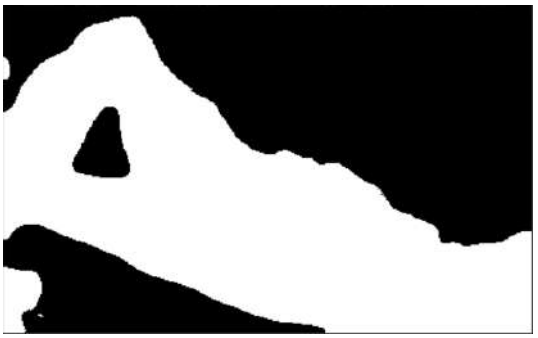


Figure3: segmentation result after Gaussian Blur filtering with Kernel 15x15.

As shown in **Figure3**, after the Gaussian Blur filtering, segmentation lost some details but the edges of the hand are smoother and precise. Illumination variation problems still remain, in fact, looking at Figure1, it can be noticed an illumination change in the right part of the hand, that k-means clustering does not classify in the right class. This problem arises because the k-means algorithm operates on a label vector containing pixel colors and intensity to separate the image in two classes, but the brighter part of the hand is confused with the background that has a similar color.

To try to solve this, it has been tried an algorithm called CLAHE (Contrast Limited Adaptive Histogram) to enhance the illumination difference between two regions in order to see if k-means would then succeed to separate the hand from the background.

Figure4: result of k-means segmentation after pre-processing the image with CLAHE algorithm.

However, this approach hasn't obtained good results. In fact, as it can be seen in **Figure4** that CLAHE risks enhancing illumination differences also within the same class.
For this reason, this method hasn't been included in the project code.

To try to avoid the illumination variation problem, k-means clustering has been kept as the last possible approach in the pipeline of segmentation methods.

Inverted class problem:
Another issue that was found with this method, is that this type of clustering segmentation divides the image in two classes but sometimes the color of the classes was reversed from what we wanted to achieve. To try to solve this problem, it has been tried to compare the result of k-means segmentation and the original image, thresholded with Otsu. Then, the number of black pixels of the first mask (wrong class) that correspond to white pixels in the Otsu mask has been evaluated. If this number is bigger than the number of white pixels of the k-means mask corresponding to white pixels in the Otsu mask, the original mask gets reversed.

3) INTERSECTION OF WATERSHED WITH A BINARY IMAGE:
To segment hands with watershed, **three main approaches** were tried. However, in the final segmentation algorithm that was used to complete the project, only the last of these was used, as it was the one that gave the best results.
The function implementing the third approach is called *"watershed_one_box(..)"* and can be found in the file *"segmentation.cpp"*.
The main idea of each of these approaches is to get the output of the watershed algorithm and to compute the intersection with a binary image, obtained in one of the following ways:

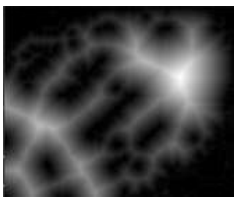
- thresholding in the YCrCb color space;
- Otsu's thresholding;
- thresholding in the HSV color space.

In general, for every approach, the YCrCb color space obtained the best results on the first 20 images, while Otsu's thresholding performed better on the last 10 images.

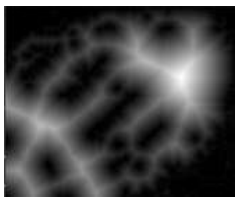
First approach:
This approach works as follows:

1. For the current bounding box, the function crops the section of the original image contained in it.
2. Subsequently, it applies a Laplacian filter of size 3x3 to the resulting image. [(1, 1, 1); (1, -8, 1); (1, 1, 1)] and [(2, 2, 2); (2, -16, 2); (2, 2, 2)] were tested as values for the filter. In the end, the first values were used since no great changes were observed in the results.
   Here is an example of the result obtained applying this filter (original cropped image on the left, sharp obtained image on the right):


3. From the image obtained with this filter, 3 types of thresholds were tested so as to obtain a binary image. The first of these is thresholding in the YCrCB color space, which is the one that obtained the best results for the first 20 images. The second threshold that was tried was Otsu, which got the best results for the last 10 images (in accordance with the considerations already made in the section *"THRESHOLDING IN DIFFERENT COLOR SPACES"* of this report). The third threshold that was tried was thresholding in the HSV color space, which, however, was the one that obtained the worst results.
4. Next, the distance function was applied to the binary image. To do this, the function cv::distanceTransform(...) was used.
   Here is an example (binary image on the left, distance transform result on the right):

5. Then, the result of the distance function was thresholded in order to keep only the white pixels at a greater distance from the black pixels in the binary image. The peaks found in this way were then dilated with a structural element of size 3x3.
   Here is an example (distance transform on the left, results of thresholding in the center, result of dilation on the right):



6. Afterwards, out of the peaks obtained in this way, only the one having the largest area was kept. This is because, from the results obtained, it was observed that the peak with the largest area was usually the central area of the palm or of the back of the hand (since these are the regions of the hand with the greatest distance from the background). The peak with the largest area was used as the watershed marker for the hand. To complete this step, these functions were used: cv::findContours(...), to find the peaks, and cv::contourArea(...), to compute the area contained by each contour in order to find the largest one.
7. After choosing the hand marker, it was needed to select the markers for the background. To do this, from the binary image obtained in step 3, a background marker was created for each one of its black pixels. Here is an example (binary image from which the background markers are extracted on the left, selected background markers (white pixels) on the right):


8. At this point, it was possible to apply the watershed algorithm by calling the function cv::watershed(...).
9. To remove the extra white pixels from the output of watershed, another binary image was created starting from the image obtained in step 1. In particular, the same threshold type as in step 3 was used (so the binary image obtained in this way would be the same as the image in step 3 if the Laplacian filter had not been applied in step 2). In particular, the final output of this approach was given by the intersection of this binary image with the output of watershed. To compute the intersection, the function cv::bitwise_and(...) was used.

This first approach, however, had some problems. In particular, there were too many background markers and, therefore, in the result obtained with watershed, part of the hand was considered as part of the background.
To solve this problem, the second approach was developed.

Second approach:
This approach differs from the first only in terms of the selection of the background markers of step 7.
In fact, instead of selecting a background marker for each black pixel in the binary image, only one marker is created for each connected region having all black pixels. However, the results obtained in this way were not consistent either. This is due to the fact that the selection of the background markers was still based on the binary image. Since this was created by thresholding the image obtained in step 1 in the YCrCb or HSV color spaces or with Otsu, the markers were very sensitive to illumination changes.

Here is an example of a good result (from left to right: sharp image obtained with the laplacian filter, selected markers, final results = intersection of the output of watershed with the thresholded image in the YCrCb color space):



To make marker selection independent of the binary image, the third approach was developed.

Third approach (best one):
The idea behind the third approach is to remove the dependence of background markers on binary images, as they are dependent on illumination conditions.
To do this, **2 more steps** were added to the second approach, one at the beginning and one at the end, and the marker selection in **step 7 was modified**.

The **first of these additional steps** attempts to enlarge the bounding box of the original image by a maximum of 4 pixels in each direction. Thus, if the bounding box is right on the edge of the image, or close to it with one side, the bounding box will be enlarged by less than 4 pixels along this direction and by 4 pixels along the other directions.

In **step 7**, a background marker is selected for each corner of the enlarged bounding box. In addition to these four markers, one marker is selected for every 30 pixels along the edge of the enlarged bounding box. The idea behind this reasoning is that, just outside the bounding box, in general, there shouldn't be any other hands, but only background. Therefore, by selecting the background markers only in the region of the image between the true and the enlarged bounding boxes, the markers selected in this way will actually be placed on the background. Here is an example of the background markers (white pixels):

Finally, the **step that is added at the end** of the whole process has the role of cropping the final segmentation result, which was enlarged by at most 4 pixels of each direction, in order to make it match the size of the original bounding box.

Here is a summary of how this third approach works (This approach differs from the second only for the highlighted parts):
0 - For the current bounding box, the function crops the section of the original image contained in it and tries to enlarge it by at most four pixels along each direction.
1 - For the enlarged bounding box, the function crops the section of the original image contained in it.
2-3-4-5-6 - Same as the first and second approaches.
7 - A background marker is selected for each corner of the enlarged bounding box. In addition to these four markers, one marker is selected for every 30 pixels along the edge of the enlarged bounding box.
8-9 - same as the first and second approaches.
10 - The function crops the final segmentation output, which was enlarged by at most four pixels in each direction, to make it match the size of the original bounding box.

This approach, in fact, turned out to be much more consistent than the previous two approaches and, therefore, it serves as the basis for the final segmentation algorithm of our project.

Note: To understand why as the final result the watershed output is not taken directly, but instead it is intersected with a binary image (YCrCb thresholding, Otsu thresholding or HSV thresholding) let us consider the following example:

Here is the result obtained with watershed:

With the third approach the markers are placed only outside the bounding box, so the region of table between the thumb and index finger couldn't be classified as background.

Here is the binary image obtained by thresholding the starting image in the YCrCB color space:

As previously stated in the *"THRESHOLDING IN DIFFERENT COLOR SPACES"* section of this report, thresholding in the YCrCB color space generally finds the whole hand plus a few extra regions (in this case the chessboard).

By taking the intersection of the two results, the following mask was obtained:

As one can see, both problems have been solved:
1 - The region between the thumb and index finger is now considered as part of the background (which was the problem of the output of watershed);
2 - The chessboard now is not considered as part of the hand (which was the problem of the thresholding in the YCrCb color space).

FINAL SEGMENTATION MODULE - CUSTOM WATERSHED-BASED ALGORITHM

Given an image to be segmented and its bounding boxes, the final segmentation algorithm involves at most 4 iterations for each of its bounding boxes. The function implementing this algorithm is called *"my_watershed(...)"* and can be found in the file *"segmentation.cpp".*

After performing one iteration, this algorithm performs the subsequent iteration only if the results obtained with the current one are bad.
The first three iterations of this algorithm are based on the intersection between the output of watershed and a binary image (the algorithm described in the previous section as *"third approach"*). What changes between one iteration and the next one is the method used to obtain that binary image (thresholding in YCrCb color space, Otsu's thresholding or thresholding in the HSV color space). The fourth iteration, insted, is just based on K-means (**Note:** k-means was implemented by Gabriele Boscarini).

To decide how many iterations to perform, a metric was defined to check whether the current result is good enough.
This metric is defined as follows:

$$"quality\ of\ current\ mask" = \frac{number\ of\ white\ pixels\ in\ the\ current\ mask}{number\ of\ total\ pixel\ in\ the\ current\ mask}$$

In particular, after several tests it was found that:

- The masks that did not segment the hand correctly obtain a low score of this metric.
- The minimum threshold for the value of "quality of current mask" of a good mask is 0.32.

In fact, if the mask obtained with watershed is bad, it means that it has found a point in the background as the peak of the distance transform (for more details, in the paragraph "*INTERSECTION OF WATERSHED WITH A BINARY IMAGE" see the section "First approach" - steps 4, 5, 6)* and, therefore, the resulting mask will have only a few white pixels.

In detail, the algorithm works as follows:

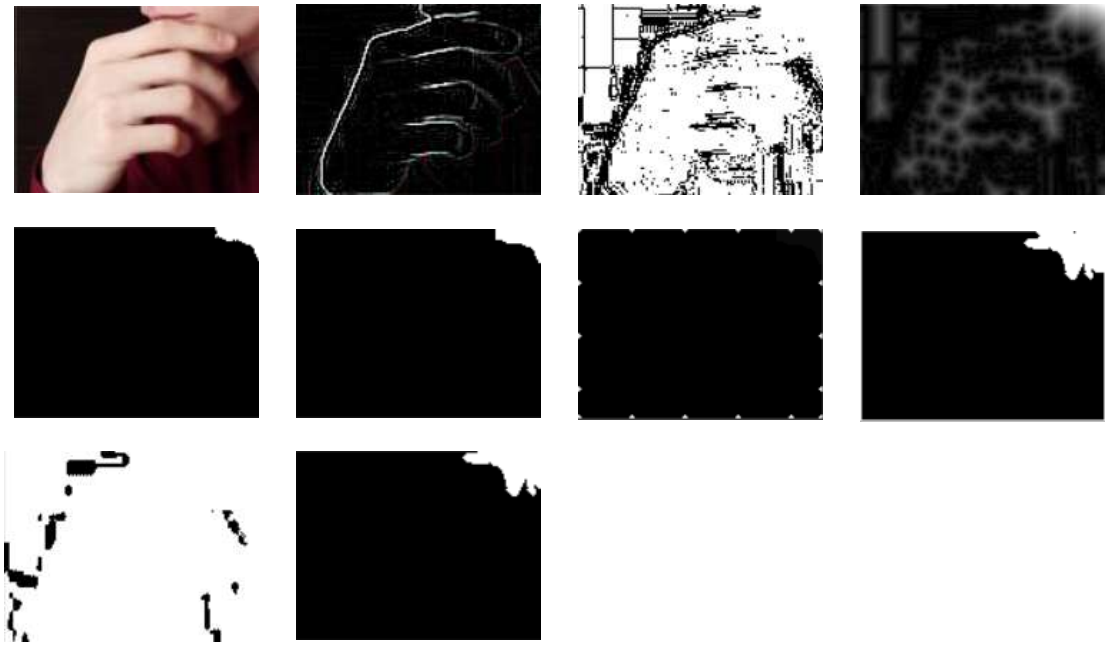
1. For each bounding box of the current image:
    1.1. (Iteration 1) It invokes the function "watershed_one_box(const cv::Mat& img, const Bounding_Box& box, const int& **method**)", which, as described in the previous section, implements the best approach for computing the mask of the hand by intersecting of the output of watershed with a binary image. For this iteration, in particular, the "method" input parameter is set to 0, which specifies that the method to be used to create the binary image is thresholding in the YCrCb color space. After computing the intersection mask, if "*quality of current mask" >= 0.32,* then it returns the current mask and switches to the next bounding box, otherwise it goes to step 1.2.
    1.2. (Iteration 2) It invokes "watershed_one_box(...)", but this time the binary image will be the one obtained with Otsu's thresholding instead of YCrCb. If "*quality of current mask" >= 0.32,* then it returns the current mask and switches to the next bounding box, otherwise it goes to step 1.3.
    1.3. (Iteration 3) It invokes "watershed_one_box(...)", but this time the binary image will be the one obtained by thresholding in the HSV color space. If "*quality of current mask" >= 0.32,* then it returns the current mask and switches to the next bounding box, otherwise it goes to step 1.4.
    1.4. (Iteration 4) If the algorithm reaches this point, it means that the custom algorithm based on the intersection of the output of watershed with each one of these three binary images failed. At this point, the algorithm uses K-means

instead. If "*quality of current mask" >= 0.32,* then it returns the current mask and switches to the next bounding box, otherwise it goes to step 2.

2. If the algorithm reaches this point, it means that none of the 4 iterations found a mask with a score of "*quality of current mask"* above the minimum threshold (=0.32). Therefore, the algorithm returns the mask that obtained the greater value of "*quality of current" mask.*

Here is an example of how this algorithm works:

1. Get current bounding box
   1.1. Intersection of watershed with binary image obtained by thresholding in the YCrCb color space. (From left to right: enlarged box, laplacian, sharp YCrCb binary image, distance transform, peaks in the distance transform, dilated peaks - the one with the largest area will be the hand marker, background markers, watershed output, YCrCb thresholded binary image obtained from the original cropped image, final mask = intersection between watershed output and YCrCb thresholded binary image)



"quality of current mask" < 0.32, then go to iteration 2 (step1.2)

   1.2. Intersection of watershed with binary image obtained with Otsu's thresholding. (From left to right: enlarged box, laplacian, sharp Otsu binary image, distance transform, peaks in the distance transform, dilated peaks - the one with the largest area will be the hand marker, background markers, watershed output, Otsu-thresholded binary image obtained from the original cropped image, final mask = intersection between watershed output and Otsu- thresholded binary image)

“quality of current mask” > 0.32, then return the final mask

Final output:
(intersection between watershed output and binary image created with Otsu’s thresholding)

VISUALIZATION FUNCTIONS:

Note: for this phase, the function color_masks(...) of the file visualize_results.cpp made by Edoardo Bastianello was used (its description can be found in the “INTRODUCTION” section of this report).

The file “Mask_generator.cpp” contains the methods for creating a mask of segmented image. This file is used for visualizing the results of segmentation. In this work, two types of mask are needed: color, for visualizing the final results, and binary, for checking the precision of the segmentation. Indeed, the function “getFinalMask” is able to output both of this type of masks, according to what type of mask is specified.

The function takes as input a vector of masks that are the result of the hand segmentation process that occurs within the bounding boxes and then it creates a unique mask. This method, however, casually overlaps the different colored masks without transparency. Therefore, in a couple of output images, the mask of one hand is partially occluded by the mask of another hand, even if the hand was segmented correctly.

We can choose if the output mask has to be binary (useful for checking pixel accuracy of the result) or colored.

SEGMENTATION MODULE PERFORMANCE EVALUATION - PIXEL ACCURACY:

The pixel accuracy measure is the easiest method known for evaluating instance segmentation. It computes the percentage of pixels in the image that are classified correctly:

$$Pixel\ Accuracy\ = \frac{TP+TN}{TP+TN+FP+FN}$$

It's not the best method for performance evaluation because of the **class imbalance**. In fact, the background pixels dominate the image and the hand regions are only small regions of the image. This can lead to a high pixel accuracy value even though the segmentation is wrong.

FINAL CONSIDERATIONS:

By focusing only on the regions inside the bounding boxes detected by the neural network, we didn't have to consider the whole image. In this way, we were able to obtain more accurate results. However, a downside of this approach is that, if the network does not detect a hand, then the hand will not be segmented.

********** **FINAL OUTPUTS** **********

IMAGE 1
IoU = 0.855365
IoU = 0.876355
PIXEL ACCURACY: 0.98937
In this case all the masks were created with just the first iteration of the final segmentation algorithm (watershed intersected with the binary image obtained by thresholding in the YCrCb color space).

IMAGE 2
IoU = 0.905947
IoU = 0.90276
PIXEL ACCURACY: 0.994594
In this case all the masks were created with just the first iteration of the final segmentation algorithm (watershed intersected with the binary image obtained by thresholding in the YCrCb color space).

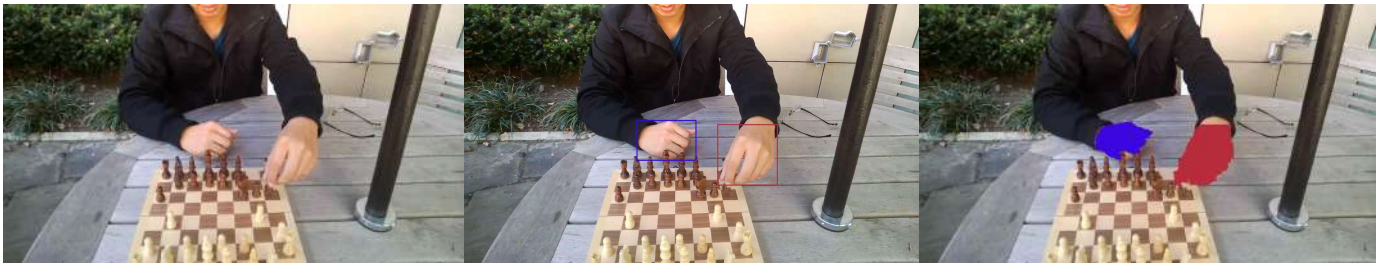


IMAGE 3
IoU = 0.859849
IoU = 0.838424
PIXEL ACCURACY: 0.994167
In this case all the masks were created with just the first iteration of the final segmentation algorithm (watershed intersected with the binary image obtained by thresholding in the YCrCb color space).

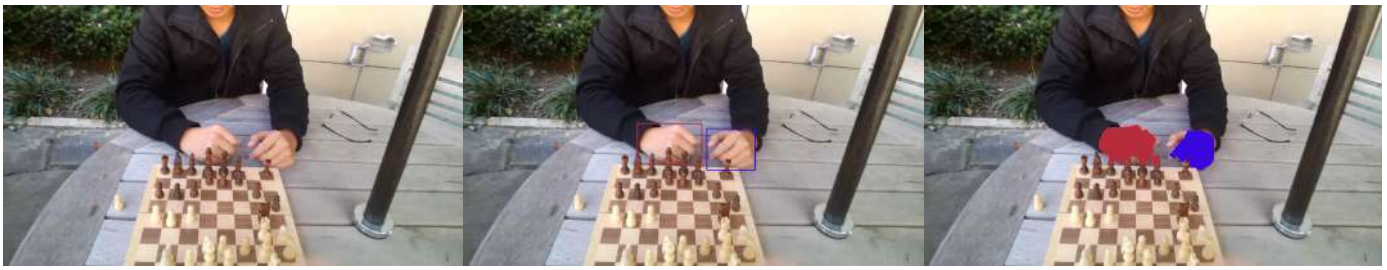


IMAGE 4
IoU = 0.896362
IoU = 0.829783
PIXEL ACCURACY: 0.993678
In this case all the masks were created with just the first iteration of the final segmentation algorithm (watershed intersected with the binary image obtained by thresholding in the YCrCb color space).

IMAGE 5:
IoU = 0.855321
IoU = 0.875111
PIXEL ACCURACY: 0.990876
In this case all the masks were created with just the first iteration of the final segmentation algorithm (watershed intersected with the binary image obtained by thresholding in the YCrCb color space).

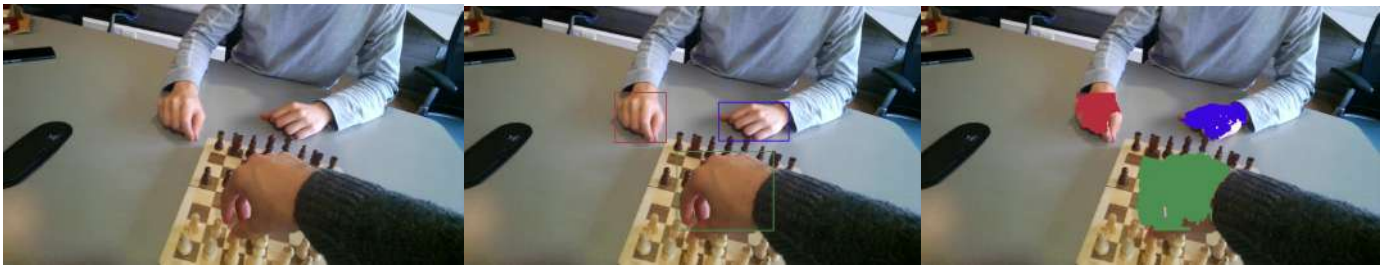


IMAGE 6:
IoU = 0.783917

IoU = 0.777702
IoU = 0.764287
PIXEL ACCURACY: 0.977462
In this case all the masks were created with just the first iteration of the final segmentation algorithm (watershed intersected with the binary image obtained by thresholding in the YCrCb color space).

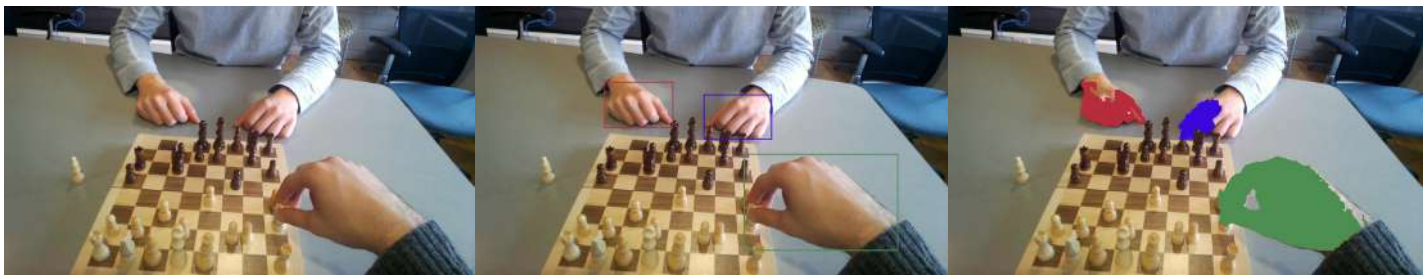


IMAGE 7:
IoU = 0.720469
IoU = 0.760387
IoU = 0.874218
PIXEL ACCURACY: 0.975049
In this case all the masks were created with just the first iteration of the final segmentation algorithm (watershed intersected with the binary image obtained by thresholding in the YCrCb color space).

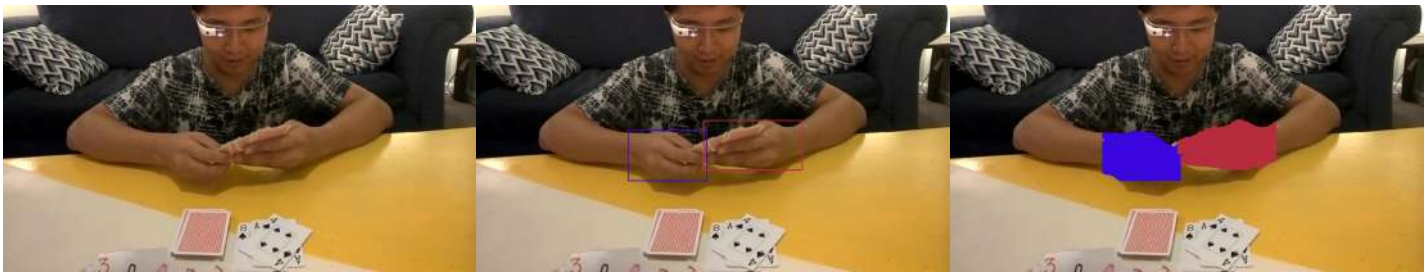


IMAGE 8:
IoU = 0.868411
IoU = 0.858849
PIXEL ACCURACY: 0.983788

In this case, all the masks were created with just the first iteration of the final segmentation algorithm (watershed intersected with the binary image obtained by thresholding in the YCrCb color space).

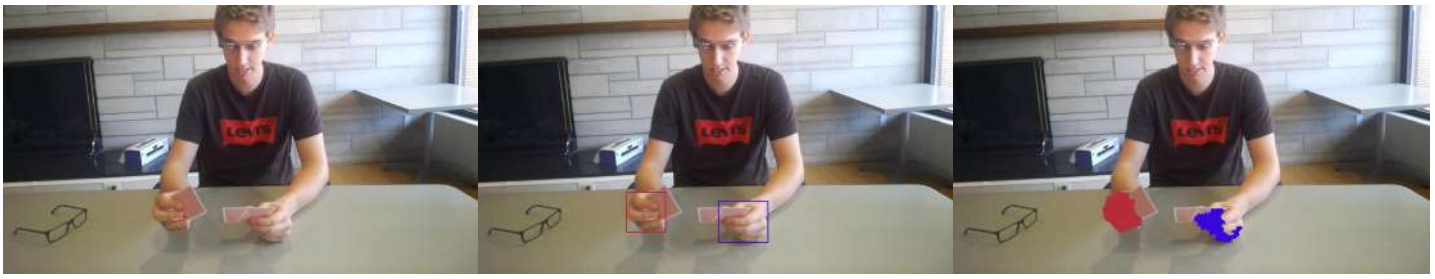


IMAGE 9:
IoU = 0.835855
IoU = 0.67544
PIXEL ACCURACY: 0.991747
In this case, all the masks were created with just the first iteration of the final segmentation algorithm (watershed intersected with the binary image obtained by thresholding in the YCrCb color space).

IMAGE 10:
IoU = 0.789235
IoU = 0.837449
PIXEL ACCURACY: 0.992709
In this case, all the masks were created with just the first iteration of the final segmentation algorithm (watershed intersected with the binary image obtained by thresholding in the YCrCb color space).

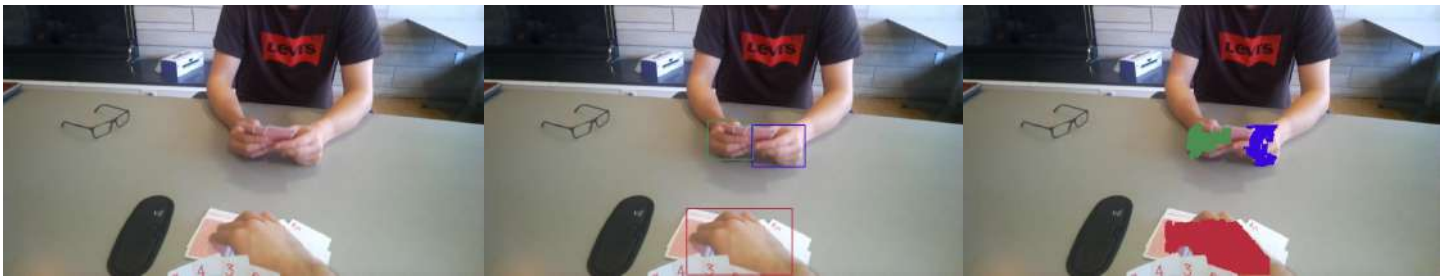


IMAGE 11:
IoU = 0.891124
IoU = 0.881372
IoU = 0.792824
PIXEL ACCURACY: 0.982103
In this case all the masks were created with just the first iteration of the final segmentation algorithm (watershed intersected with the binary image obtained by thresholding in the YCrCb color space).

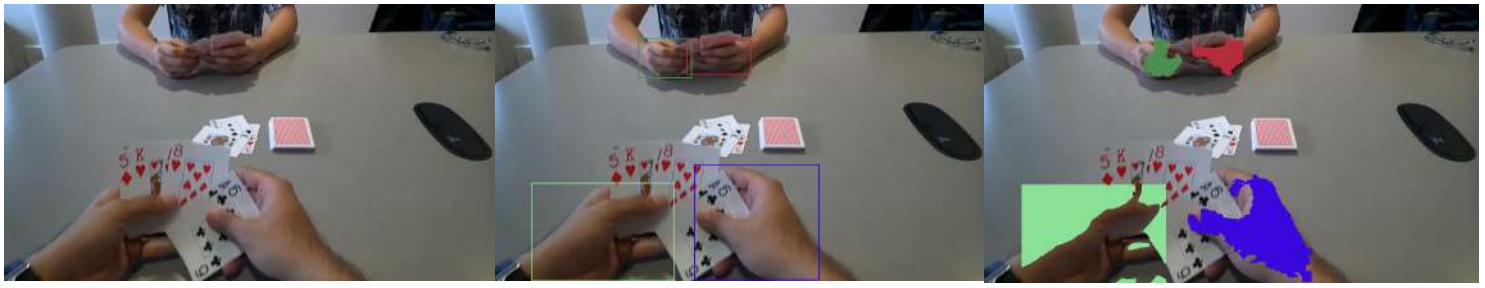


IMAGE 12:
IoU = 0.723441
IoU = 0.889784
IoU = 0.778378
IoU = 0.848155
PIXEL ACCURACY: 0.902906
This time, three of the masks were created with just the first iteration of the final segmentation algorithm (watershed intersected with the binary image obtained by thresholding in the YCrCb color space). The green mask on the bottom right, instead, was created with the fourth iteration (K-means).

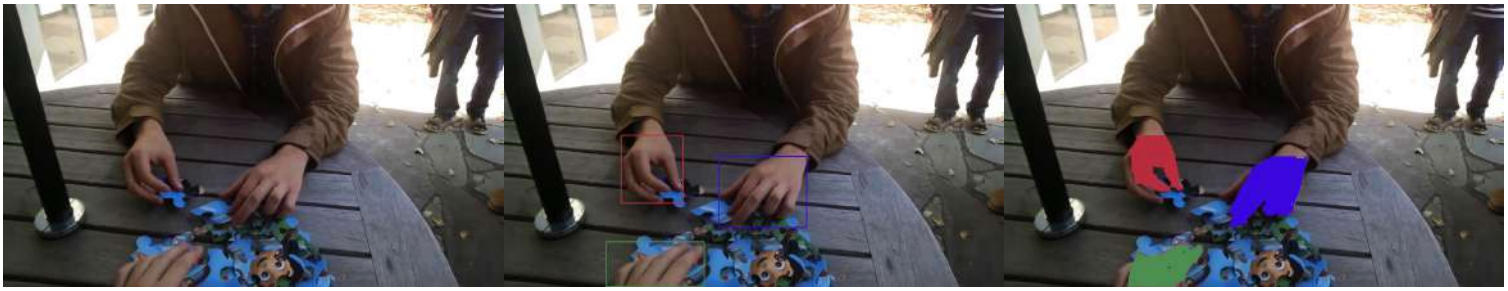


IMAGE 13:
IoU = 0.853409
IoU = 0.850072
IoU = 0.842346
PIXEL ACCURACY: 0.981163
In this case all the masks were created with just the first iteration of the final segmentation algorithm (watershed intersected with the binary image obtained by thresholding in the YCrCb color space).

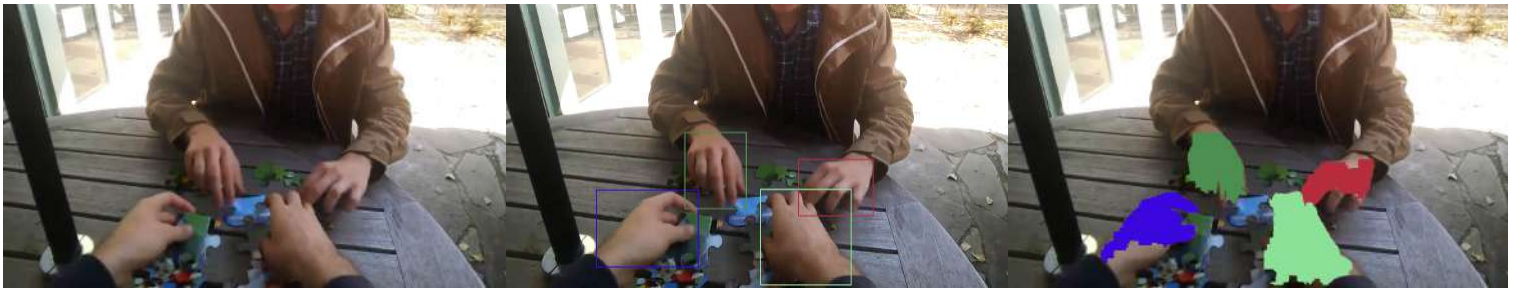


IMAGE 14:
IoU = 0.791862
IoU = 0.859293
IoU = 0.881776
IoU = 0.766589
PIXEL ACCURACY: 0.972756
In this case all the masks were created with just the first iteration of the final segmentation algorithm (watershed intersected with the binary image obtained by thresholding in the YCrCb color space).

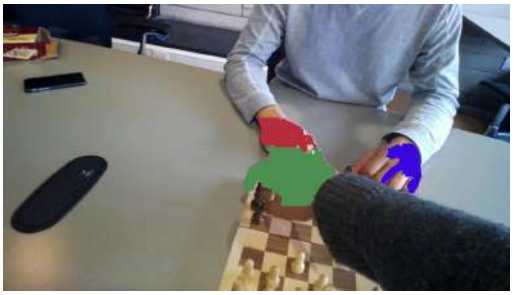


IMAGE 15:
IoU = 0.665322
IoU = 0.722171
IoU = 0.747498
PIXEL ACCURACY: 0.98163
This time, two of the masks were created with just the first iteration of the final segmentation algorithm (watershed intersected with the binary image obtained by thresholding in the YCrCb color space). The blue mask, instead, was created with K-means.

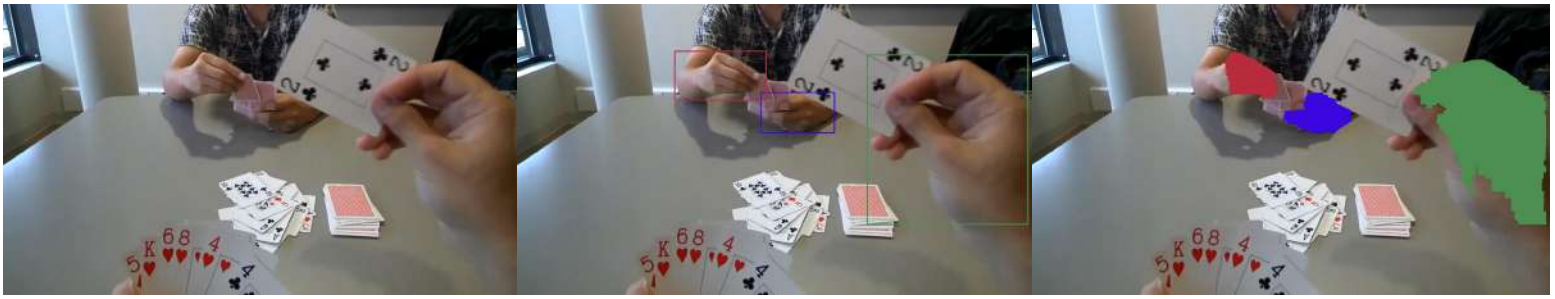


IMAGE 16:
IoU = 0.866742
IoU = 0.803834
IoU = 0.580961
IoU = 0 (false negative)
PIXEL ACCURACY: 0.939592
In this case all the masks were created with just the first iteration of the final segmentation algorithm (watershed intersected with the binary image obtained by thresholding in the YCrCb color space).

IMAGE 17:
IoU = 0.846454
IoU = 0.880435
IoU = 0.880686
PIXEL ACCURACY: 0.971869
In this case all the masks were created with just the first iteration of the final segmentation algorithm (watershed intersected with the binary image obtained by thresholding in the YCrCb color space).

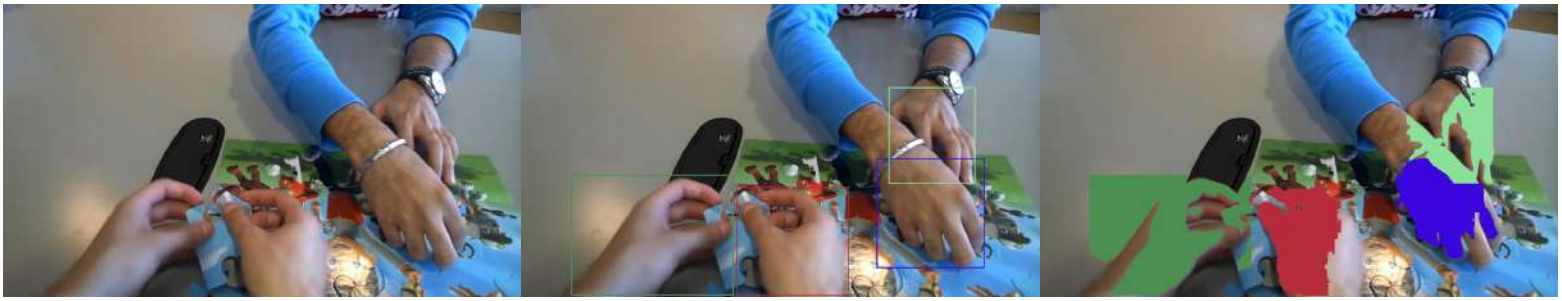


IMAGE 18:
IoU = 0.890698
IoU = 0.877663
IoU = 0.869013
IoU = 0.872942
PIXEL ACCURACY: 0.884059
This time, the red and the blue masks were created with just the first iteration of the final segmentation algorithm (watershed intersected with the binary image obtained by thresholding in the YCrCb color space). The two green masks, instead, were obtained with the fourth iteration (K-means). Actually, the back of the blue hand was also found, but the color is occluded by the green one.

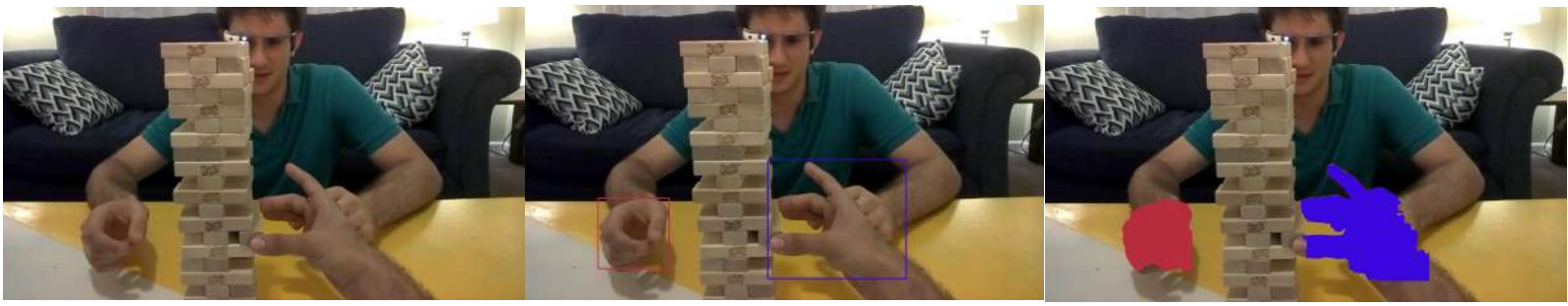


IMAGE 19:
IoU = 0.901156
IoU = 0.786327
IoU = 0 (false negative)
PIXEL ACCURACY: 0.979415
In this case all the masks were created with just the first iteration of the final segmentation algorithm (watershed intersected with the binary image obtained by thresholding in the YCrCb color space).

IMAGE 20:
IoU = 0.809741
IoU = 0.500497
IoU = 0 (false negative)
PIXEL ACCURACY: 0.98781
The blue mask was created with just the first iteration of the final segmentation algorithm (watershed intersected with the binary image obtained by thresholding in the YCrCb color space). Actually, the index finger of the blue hand was also found, but the color is occluded by the red one, which instead was created by the fourth iteration of the algorithm (K-means).

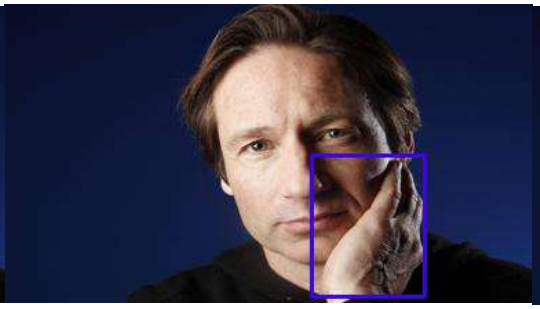




IMAGE 21:
IoU = 0.837708
PIXEL ACCURACY: 0.930194
In this case, all the masks were created with just the first iteration of the final segmentation algorithm (watershed intersected with the binary image obtained by thresholding in the YCrCb color space). As one can see, it doesn't work very well, since there is also the skin of the face next to the hand (the first iteration uses YCrCb that finds all skin).



IMAGE 22:
IoU = 0.727719
IoU = 0 (false negative)
PIXEL ACCURACY: 0.970872
In this case, the mask was created with the second iteration of the final segmentation algorithm (watershed intersected with the binary image obtained by Otsu's thresholding).

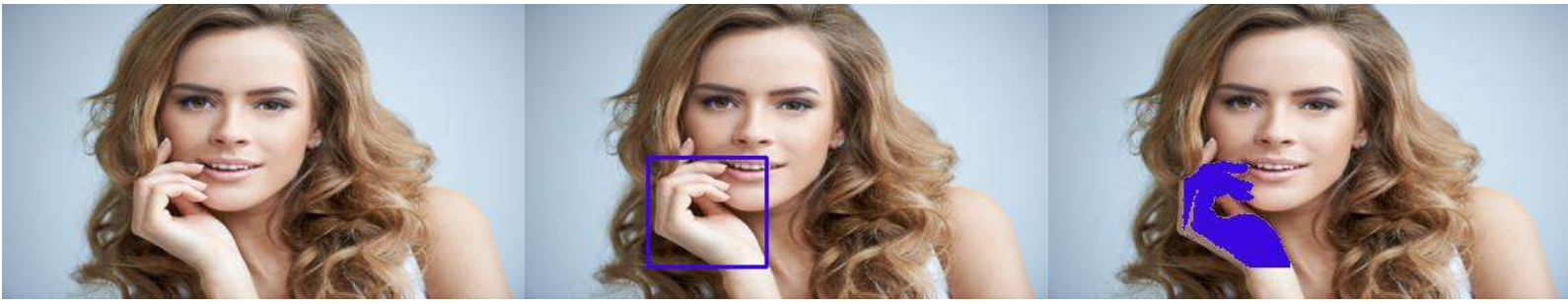


IMAGE 23:
IoU = 0.764151
PIXEL ACCURACY: 0.980264
In this case the mask was created with the second iteration of the final segmentation algorithm (watershed intersected with the binary image obtained by Otsu's thresholding).

IMAGE 24:
IoU = 0.620702
PIXEL ACCURACY: 0.97731
In this case, the mask was created with the first iteration of the final segmentation algorithm (watershed intersected with the binary image obtained by thresholding in the YCrCb color space).

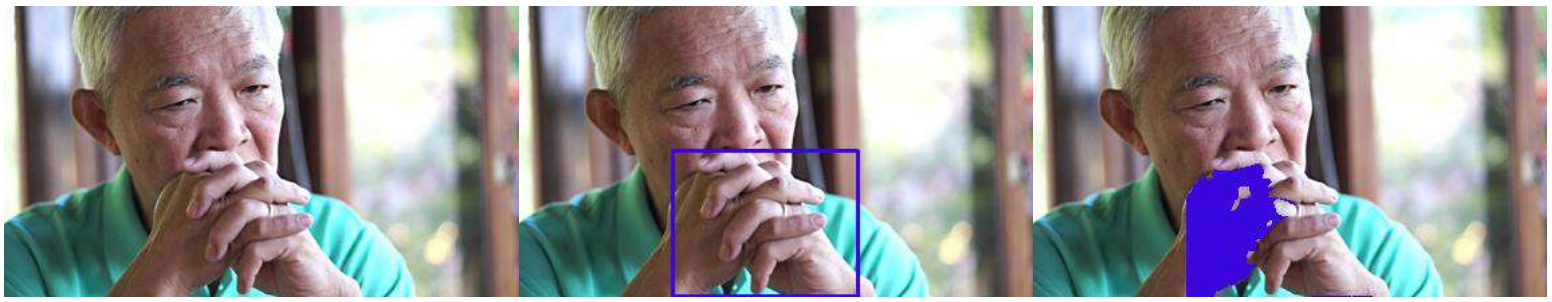


IMAGE 25:

IoU = 0.799903
IoU = 0 (false negative)
PIXEL ACCURACY: 0.913122
In this case, the mask was created with the first iteration of the final segmentation algorithm (watershed intersected with the binary image obtained by thresholding in the YCrCb color space).

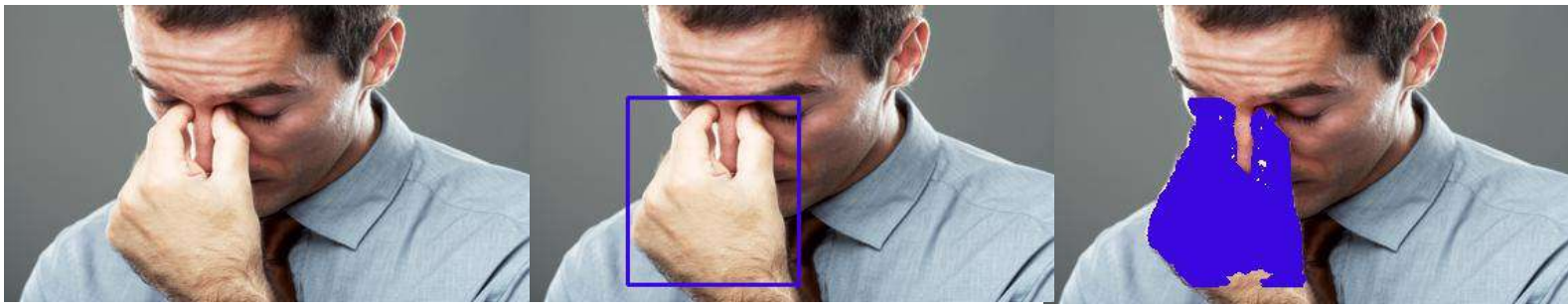


IMAGE 26:
IoU = 0.892043
PIXEL ACCURACY: 0.975658
In this case the mask was created with the first iteration of the final segmentation algorithm (watershed intersected with the binary image obtained by thresholding in the YCrCb color space).

IMAGE 27:
IoU = 0.708482
IoU = 0.737614
PIXEL ACCURACY: 0.969076
In this case, the masks were created with the first iteration of the final segmentation algorithm (watershed intersected with the binary image obtained by thresholding in the YCrCb color space).

IMAGE 28:
IoU = 0.851914
PIXEL ACCURACY: 0.980626
In this case, the mask was created with the second iteration of the final segmentation algorithm (watershed intersected with the binary image obtained by Otsu's thresholding).

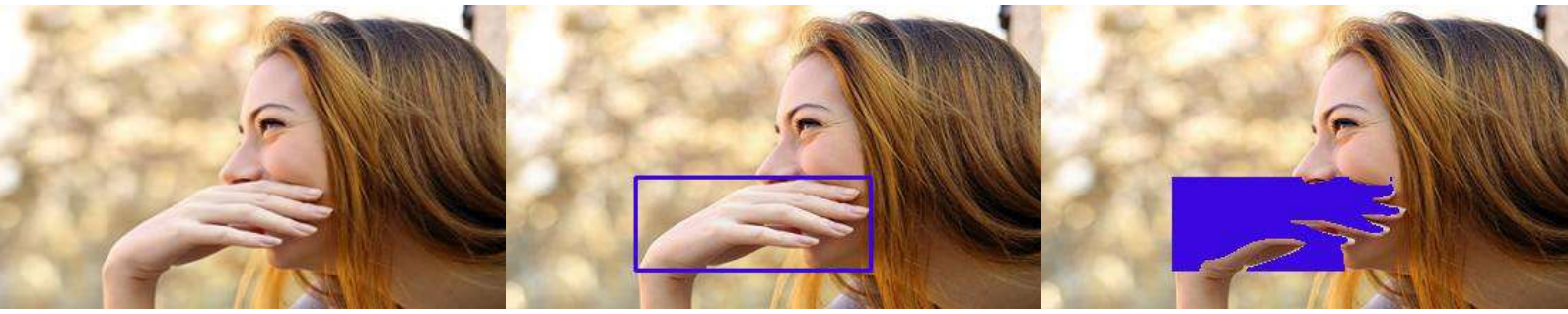


IMAGE 29:
IoU = 0.804432
PIXEL ACCURACY: 0.938633
In this case, the mask was created with the fourth iteration of the final segmentation algorithm (K-means).

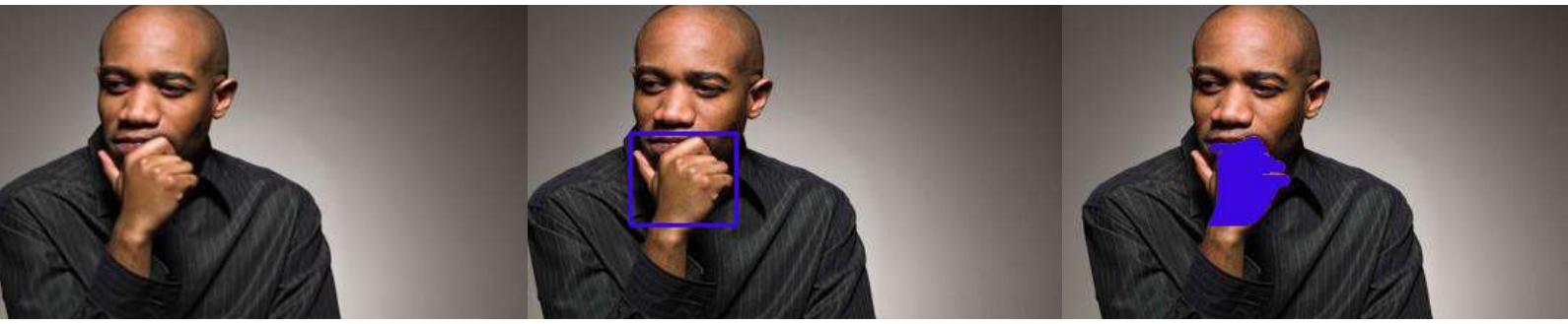


IMAGE 30:
IoU = 0.808903
PIXEL ACCURACY: 0.990825
In this case, the mask was created with the first iteration of the final segmentation algorithm (watershed intersected with the binary image obtained by thresholding in the YCrCb color space).