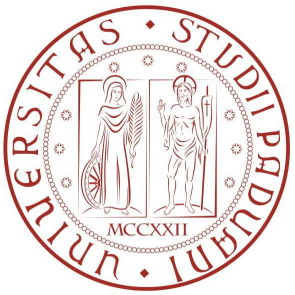


UNIVERSITY OF PADUA
INFORMATION ENGINEERING DEPARTMENT (DEI)

MASTER'S DEGREE IN COMPUTER ENGINEERING

COMPUTER VISION REPORT

PROJECT:

"OpenCV Hand Detection and Segmentation System"

PROF: STEFANO GHIDONI
GROUP:

- MANUEL BARUSCO (2053083)
- SIMONE GREGORI (2057979)
- RICCARDO RAMPON (2052416)

CONTENTS:

CHAPTER 1: Hand Detection

1.1 Overall Approach

In this project, the main idea of the detection module is to use a deep neural network, with the objective of performing hand detection in the test images provided to us. We decided to use a version of the YOLO Neural Network, as it is state-of-the-art for real-time object detection and object detection as well.

The detection module execution pipeline is as follows:

- The HandDetector object exploits the output of the neural network's forward-step to perform a series of hand predictions in the input image;
- It keeps only the predictions that exceed a certain “confidence” threshold;
- Then print out the bounding boxes.

There are two intermediate steps in the execution of the HandDetector, consisting of one in applying the Non-Maxima-Suppression and the other in refining the bounding boxes:

- Non-Maxima-Suppression (NMS) helps to overcome the problem of detecting an object multiple times in an image. It is used to eliminate all redundant and overlapping boxes that have low confidence;
- refine detection boxes in case they go outside the image.

The input of the neural network is a batch of images of shape (416, 416). After the forward step, the output is a list of bounding boxes. Each bounding box is represented by 6 numbers **(Bx, By, Bh, Bw, Pc, C)**, where **Bx**, **By**, **Bh**, **Bw** are the box coordinates (respectively, the coordinates of the box centre and the values of box height and width), **Pc** is the objectness predictions (or object confidence) and **C** is the number of class predictions.

We then applied Non-Maxima-Suppression (NMS) and a refinement to the boxes predicted by the neural network, and finally drew them on the input image.

The HandDetector is useful because through detection it provides the RoIs (Region of Interest) of the hands in each image, which is useful for later performing the segmentation module.

1.2 YOLOv3 Neural Network

YOLO is a family of deep learning models designed for object detection; we decided to use one of the five main variations: YOLOv3.
YOLOv3 has some improvements over the model architecture, the training process and the use of predefined anchor boxes, that try to improve the bounding

box proposal, with respect to the previous two versions. YOLOv3 turns out to be much more accurate than previous versions, and while a bit slower.

YOLOv3 uses a variant of Darknet CNN as its architecture: Darknet-53, which originally has a 53 layer network trained on Imagenet. For the task of detection, 53 more layers are stacked onto it, in order to obtain a 106 layer fully convolutional architecture for YOLO v3. This is one reason for the slowness of YOLOv3 than previous versions.

In addition, Darknet-53 is a convolutional neural network that acts as a backbone for the YOLOv3 object detection approach. It is composed of 3 x 3 and 1 x 1 filters with skip connections like the residual network in ResNet.

The architecture of Darknet-53:

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

(https://dev.to/afrozchakure/all-you-need-to-know-about-yolo-v3-you-only-look-once-e4m)

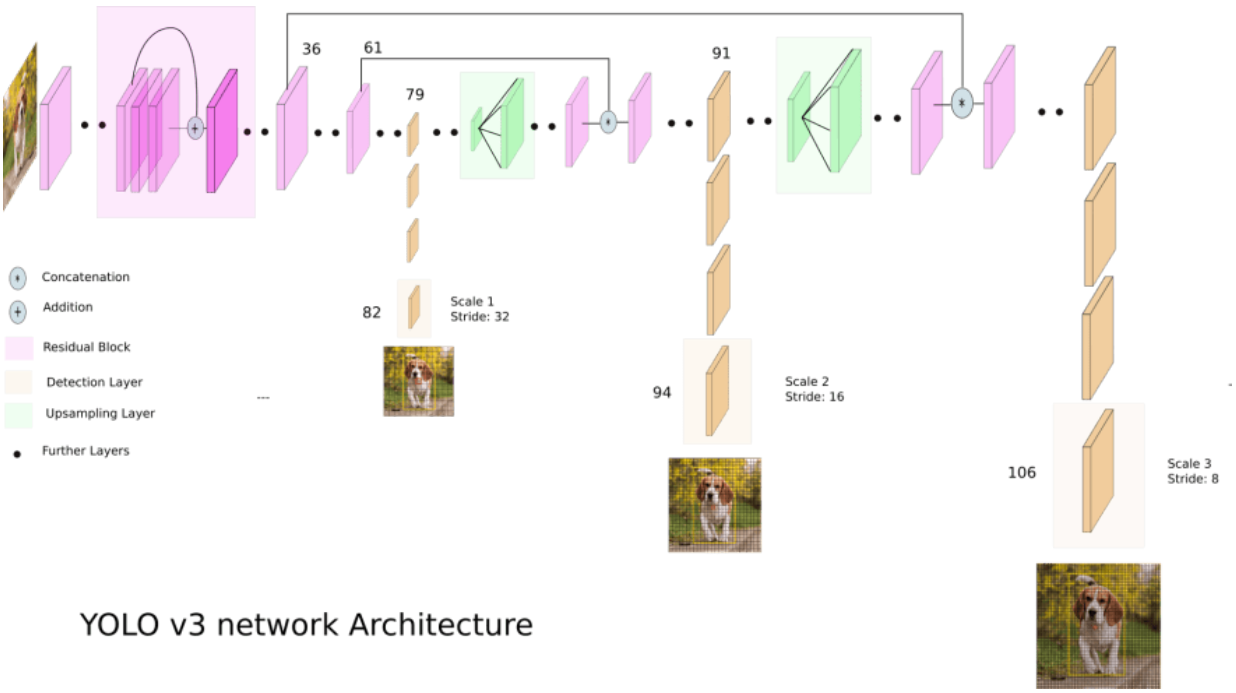
The shape of the detection kernel, or detection filter, is **1 x 1 x (B x (4 + 1 + C))**, where B represents the number of bounding boxes the filter can predict, "4" is for the four bounding box attributes, "1" is for objectness predictions (object confidence) and C is the number of classes. These are parameters that are properly modified in order to perform an object detection task with only one class.

YOLOv3 is originally trained on the COCO (Common Objects in COntext) dataset, which contains high-quality visual datasets for computer vision.
Using COCO's pre-trained weights, YOLOv3 can only perform object detection with 80 pretrained classes that come with the COCO dataset. The following 80 classes are available using COCO's pretrained weights:

'person', 'bicycle', 'car', 'motorcycle', 'airplane', 'bus', 'train', 'truck', 'boat', 'traffic light', 'fire hydrant', 'stop sign', 'parking meter', 'bench', 'bird', 'cat', 'dog', 'horse', 'sheep', 'cow', 'elephant', 'bear', 'zebra', 'giraffe', 'backpack', 'umbrella', 'handbag', 'tie', 'suitcase', 'frisbee', 'skis','snowboard', 'sports ball', 'kite', 'baseball bat', 'baseball glove', 'skateboard', 'surfboard', 'tennis racket', 'bottle', 'wine glass', 'cup', 'fork', 'knife', 'spoon', 'bowl', 'banana', 'apple', 'sandwich', 'orange', 'broccoli', 'carrot', 'hot dog', 'pizza', 'donut', 'cake', 'chair', 'couch', 'potted plant', 'bed', 'dining table', 'toilet', 'tv', 'laptop', 'mouse', 'remote', 'keyboard', 'cell phone', 'microwave', 'oven', 'toaster', 'sink', 'refrigerator', 'book', 'clock', 'vase', 'scissors', 'teddy bear', 'hair drier', 'toothbrush'

As we can see, the use of pre-trained YOLOv3 on the COCO dataset does not allow for hand detection, so we have made other changes to the configuration of the network, explained in the section about Transfer Learning.

The architecture of YOLOv3:



YOLO v3 network Architecture

(https://towardsdatascience.com/yolo-v3-object-detection-53fb7d3bfe6b)

In YOLO v3, the detection is done at three scales, which are given by downsampling the dimensions of the input image by 32, 16 and 8 respectively; it is done by applying 1 x 1 detection kernels on feature maps of three different sizes at three different places in the network.

YOLO v3 uses binary cross-entropy for calculating the classification loss for each label while object confidence and class predictions are predicted through logistic regression.

1.3 Transfer Learning

Transfer learning is a popular approach in deep learning where pre-trained models are used as the starting point and reused as the starting point for a model on a given task.

Our approach was to consider the pre-trained YOLOv3 Neural Network, so its configuration and weights file, and then re-train the network on a properly built dataset of hand images, in order to perform hand detection.

In order to perform object detection with only one class we have modified some parameters in the YOLOv3 configuration file *'yolov3_training.cfg'*:

- batch size *"batch"*: from 1 to 64;
- number of subdivisions *"subdivisions"*: from 1 to 16, in order to speed-up learning process;
- number of max batches *"max batches"*: from 500200 to 2000;
- number of classes *"classes"*: from 80 to 1;
- shape of filters *"filters"*: from 255 to 18;

As we mentioned in the previous section, the shape of the detection kernel is **1 x 1 x (B x (4 + 1 + C))**, where B is the number of bounding boxes a cell on the feature map can predict, "4" is for the four bounding box attributes, "1" si for object confidence, and C is the number of classes (in our case only one = *"hand"*). To adapt to detection to only one class, we setted B = 3 and C = 1, so the filter's shape is **1 x 1 x 18**.

Another change made was to set the label file 'coco.names' on which YOLOv3 was training: we have created a new file called 'coco2.names' that contains only one label 'hand'.

All these changes are performed with a Python Script (link to github repository: https://github.com/manuelbarusco/OpenCVHandDetection/blob/main/PythonScripts/Yolov3_HandDetection.ipynb), which also contains in detail the training process of the YOLOv3.

1.4 Dataset

In order to train our YOLOv3 model, we used a dataset built specifically for hand detection, composed initially of images from the **EgoHands** and **HandsOverFace** datasets.

For **EgoHands** dataset, we used a version of the dataset that had the labels compatible with the particular structure required by YOLOv3, which can be downloaded at the following link:
https://public.roboflow.com/object-detection/hands

For the **HandsOverFace (HOF)** dataset, the labels were created manually using a graphical image annotation tool called *LabelImg* (link: https://github.com/heartexlabs/labelImg).

The results obtained by testing this version of the model on the test set were good, but in certain images, especially those with hands over the face, the model did not perform detection correctly, so we realised that it probably did not generalise well in images with hands over the face, certainly because of the small number of images that had that peculiarity.

To solve this problem, we performed **data augmentation**, which is a technique used to increase the amount of data in a dataset by adding slightly modified copies of existing data.

We therefore performed data augmentation of the images in the **HandsOverFace** dataset using the following transformations:

- Flip: we add some horizontal or vertical flips to help our model be insensitive to subject orientation.
- Crop: we add variability to positioning and size of images to help our model be more resilient to subject translations.
- Rotation: we add variability to rotations.
- Shear: we add variability to perspective to help our model be more resilient to subject pitch and yaw.
- Brightness and Exposure: we add variability to image brightness to help our model be more resilient to lighting changes.
- Blur: we add random Gaussian blur.

After the data augmentation process, the trained model responded much better to the test set images.

CHAPTER 2: Hand Segmentation

2.1 Overall Approach

In this project the main idea of the segmentation module is to exploit the results provided by the detection module, by segmenting the detected hand from the bounding box previously returned through the use of different segmentation techniques combined with each other. We decided to not use an ad-hoc Neural Network for the segmentation task because we wanted to try to see if a “pure coding” technique could be good for this job.

The segmentation module execution pipeline is as follows:

- The HandDetector object provides the RoIs (Region of Interest) of each hand present within a given image.
- The HandSegmentator object receives each individual RoI returned by the HandDetector, it pre-process every RoI to get a better RoI version for segmentation, and then extracts the hand from the RoI through the use of various segmentation techniques combined with each other, as explained in the next sections.
- Every segmented RoI is then used to create the final segmentation mask image corresponding to the input image.

The HandSegmentator object segments the hand RoI by using the following segmentation techniques::

- K-Means based on pixel color and position. It is used in the Advanced Region Growing Technique (see next point).
- Advanced Region Growing: ad-hoc Region Growing technique developed by us for this task and based on K-Means clusters (based on pixel color and position), an edge map and a particular Region Growing seed set initialization. The results of this process are used for the GrabCut mask initialization (see next point).
- GrabCut foreground segmentation for the final (hand) segmentation results starting from the mask returned by the Advanced Region Growing technique.

All these techniques are explained in the details in the following sections.

2.2 Dataset

In order to test our HandSegmentator object results and to set all the parameters that affect its execution, we did all the tests in a simple dataset composed by some RoIs computed by the HandDetector object on a set of Hand Over Face images and on a set of simple Non Over Face images.

2.3 RoI (Region Of Interest) pre-processing

As said before the HandDetector object returns to the HandSegmentator object a set of hand RoIs, one for each hand detected in the given image. Every RoI is then pre-processed by the HandSegmentator object in order to get a better RoI version for segmentation. In this step a bilateral filter is applied to every single RoI in order to reduce the RoI noise and to remove every non significant detail such as skin wrinkles or strange patterns but preserving as much as possible the hand edges (parameters for the bilateral filter: `sigma_color=50, sigma_space=120`).
Then we obtain from each RoI an edge map by using the Canny edge detector with parameters: `t_low = 10` and `t_high = 150`. These two parameters are quite low due to the blur operations done on the RoI. The values of the parameters present at this stage have been decided empirically through tests conducted on our dataset (see Section 2.2).

2.4 K-Means based on pixel color and position

In our project K-Means is used to quantize the colors of every RoI image into K colors, the resulting RoI image then is used by the Advanced Region Growing function as explained in Section 2.5.
Previous tests show that the K-means algorithm, implemented in OpenCV library, has a better outcome if the input points information include the position of the pixel (i.e. the row and column index of the pixel), in addition to the three pixel colors (RGB).
Since the pixel feature vector has different range of values for its five components (i.e. the RGB values vary from 0 to 255, instead the index can exceed 255 value by a lot depending on the size of the image) a normalization process is required to ensure good results. We choose to map all features in range from 0 to 1. In particular, for the three color values we use the function `convertTo(img, CV_32FC3, 1.0/255.0)`, and for the position value we use min-max normalization that maps value in [0,1]:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

An additional scaling is done only for the row and column indexes dividing respectively with `weightX` and `weightY` (they are parameters of the function `minMaxNormalization`) the quantity above.
Then all the feature vectors (one for each pixel in the image) are gathered into one channel Mat object, where each row represents a feature vector. Afterwards, kmeans function is applied to this object with `attempts = 10, criteria = (TermCriteria::EPS+TermCriteria::COUNT, 100000, 0.00001)` and with the option to use centers initialized using kmeans++. For extracting the quantized color is sufficient to scan the original image and assign to each pixel the RGB value of the center that has been assigned to that pixel.

Below there is an example of what this K-Means version (based on pixel color and position) returns given the input image on the left.

Input Image

Output Image

2.5 Advanced Region Growing

This ad-hoc Region Growing technique is developed in order to return a rough mask of the hand present in a given pre-processed RoI. This mask is then returned to the next segmentation step where the GrabCut algorithm uses it for the final hand extraction.

The first step of this Region Growing technique is to cluster the input hand RoI (already pre-processed) by using the K-Means based on pixel color and position (presented in Section 2.4) with `k = 5` (number of clusters), `weight_x = 2` and `weight_y = 2`. All these parameters were set empirically in our dataset (presented in Section 2.2).

After that a smart seeds set initialization is performed based on the following assumptions:

- the center of the RoI returned by the HandDetector object is a hand point.
- if the RoI is a vertical image, probably the hand is vertical, so some parts of the hand may be in shadow. (Same observation is made for an horizontal RoI).

Based on these assumptions the seed set initialization has to retrieve as much as possible a good seeds set, that has to correspond as much as possible to hand points in order to ensure a good Region Growing execution.

The seed set initialization is then performed as follows (in the following points, `RoI_cols` is the number of RoI image columns, `RoI_rows` is the number of RoI image rows):

- The RoI center is assumed to be a hand point so it is added to the seed set.

- if the RoI is a square (or similar) image then we add every RoI point that is inside a circle of radius `RoI_cols / 30` from the RoI center.
- If the RoI is a vertical rectangle image then we add every point that is near the RoI center column in an interval of length `RoI_cols / 30` and near the RoI center row in an interval of length `RoI_rows / 10`.
- If the RoI is an horizontal rectangle image then we add every point that is near the RoI center column in an interval of length `RoI_cols / 10` and near the RoI center row in an interval of length `RoI_rows / 30`.

For checking if a RoI is vertical we check if `RoI_cols / RoI_rows < 0.75`, for checking if a RoI is horizontal we check if `RoI_rows / RoI_cols < 0.75`.
The clustered colors (from K-Means) of every seed point is saved in a set of colors.
So at the end of this phase every seed point in the seed set is assumed to be a hand point and every seed point clustered color is saved in a set of colors. Every color in this set is assumed to be a color present inside the hand region.

The third and last phase is the classical Region Growing process. We start to expand every seed point in order to create the final regions by aggregating pixels that have a clustered color that is inside the set of colors previously created. Furthermore, if a pixel is an edge pixel (we can see it from the edge map previously computer, see Section 2.3) we add it to the region and then we stop the possible future expansions from that point (because we assume that this point is a hand contour point).
At the end of the Region Growing process a rough hand mask is returned from the given RoI, this mask is sometimes very good and it segments the hand quite well but sometimes is not so good so we decided to use it as an initialization mask for the GrabCut algorithm as explained in the following section.

Here there is an example of the Advanced Region Growing output given a particular input:

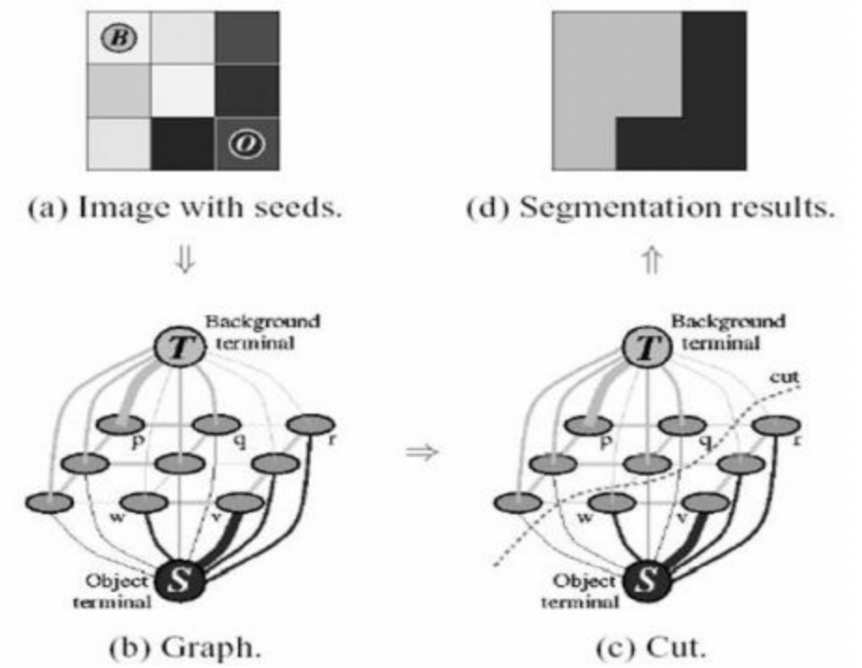
Input Image

Output Image

2.6 Graph based segmentation and GrabCut

Graph based segmentation is a segmentation technique that uses the image represented as a graph to find subregions with the same characteristics (namely subgraphs of the starting graph).
So given an image, a graph G = (V,E), can be built assigning a vertex to each pixel and each edge between a pair of vertices is weighted by the affinity between its vertices (higher weight means that pixels have similar color).
A Cut C = (V_A, V_B) of a graph G = (V,E) is a partition of vertices V into two disjoint subset V_A and V_B, and its cost is the sum of the weight of the edges that have a vertex in V_A and the other in V_B.
With these notions we can define a basic segmentation algorithm that uses the following criteria: a pair of vertices (pixels) within a subgraph have high affinity, instead a pair of vertices from two different subgraphs have low affinity.
The goal of this algorithm is to find cuts that have minimum cost associated with them (**Min-Cut**). So each subgraph we end up with at the end is an image segment.

In a nutshell, **GrabCut** is an algorithm to segment background and foreground in an image using Gaussian Mixture Model (GMM) and the Min-Cut procedure.
First of all the algorithm uses a input mask or a rectangle (`cv::Rect`) for specifying which pixels are part of foreground (flag `GC_FGD` for sure foreground or `GC_PR_FGD` if it is probably foreground) or background (`GC_BGD` for sure background or `GC_PR_BGD` if it is probably background). Note that if the user provides a rectangle, it uses only flags `GC_PR_BGD` or `GC_PR_FGD`.
Then a Gaussian Mixture Model (GMM) is used to learn and build a new labelling that depends on the user input and on the spatial and color relation between pixels.
A graph is built from this pixel distribution and Source and Sink nodes are added to the graph. Every pixel node in the graph is connected to the Source and Sink node. The Source node represents the foreground of the image, and the Sink node the background. At this point an energy function is used to assign weights to edges: edges connecting pixels to Source/Sink node are defined by the probability of a pixel being foreground/background, edges between the pixels have weights that depends on similarity between then (if there is a large difference in pixel color, the edge between them will get a low weight).
After this the graph is segmented using Min-Cut procedure described above. Finally, after the cut, all the pixels connected to Source node become foreground and those connected to Sink node become background. The process is repeated until the classification converges.



For more details on the energy function and on the algorithm itself: "GrabCut": interactive foreground extraction using iterated graph cuts.

2.6.1 Use of GrabCut in our proposed model

After some testing we figured out that the initialization of the GrabCut algorithm using the rectangle (provided by the bounding boxes returned by the HandDetector object) was not sufficient for delivering good segmentation results in all images. Indeed, in images where the hand is occluded by some object (like for example chess pieces), or the hand is close to parts that are not immediately recognizable as background, the mask returned by GrabCut includes many non-hand parts. Another cause of bad segmentation using this type of initialization is that the region of interest provided by the HandDetector sometimes is too tight.
To overcome these difficulties, we have decided to provide an initial mask to the GrabCut function; to do so it is necessary to specify an option in the parameters using `GC_INIT_WITH_MASK` instead of `GC_INIT_WITH_RECT`. Considering the input image (of original size) the mask is obtained by combining a skin segmentation mask with the mask returned by our AdvancedRegionGrowing function and the overall process has the following steps:

1. Crop the image into the region of interest that is defined by the detection module.
2. If the input image is a colored image (i.e. it is not a grayscale image composed by three channel with the same values) then the cropped image is segmented using a simple thresholding process for segmenting the skin areas exploiting YCrCb and HSV planes to create

the binary mask called *skin*. Otherwise, the *skin* mask is computed thresholding the image using Otzu optimal threshold, this because in grayscale images the skin segmentation function developed does not work. Below there is an example of binary mask *skin*.

3. If the *skin* mask has a low (w.r.t. a defined threshold) percentage of active pixels (i.e. pixel with value equal to 255) a new *skin* mask is created using a thresholding method based on the three mean color computed on random sample of the cropped image's pixels excluding too dark and too light pixels that less likely are a hand points. This step is done to avoid the use of a less effective *skin* mask on the combined mask (explained at step n.5) that could compromise the final result.
4. Run the `advancedRegionGrowing` on the cropped image to create *bwSmall*.

5. Combine the two masks (*skin* and *bwSmall*) assigning to pixel value `GC_FGD` if they are both equal to 255, value `GC_PR_FGD` if they are different and `GC_PR_BGD` if they are both equal to 0. The figure below represents the combination of the previous two binary masks. Note: White represents `GC_FGD`, light grey is `GC_PR_FGD`, dark grey is `GC_PR_BGD` and black is `GC_BGD`.

6. Create a binary mask *bwBig* of size equal to the original image and initialise it to `GC_BGD`, then superimpose the smaller hand mask (the result of step 5) into *bwBig*.
7. Run GrabCut algorithm using original image and *bwBig* as input and create a binary mask with true value in pixel where GrabCut alg. assign `GC_FGD` or `GC_PR_FGD`.
8. Merge the intermediate segmentation result in another Mat object that will carry the segmentation of all hands present in the original image.
9. Repeat from step 1 until all hands region of interest in the input image are processed.
10. Return the final result.

2.7 Other tested approaches

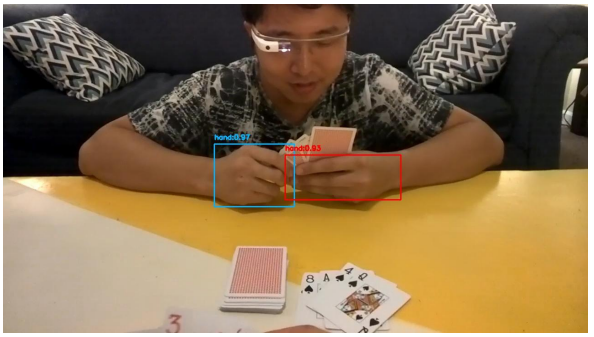
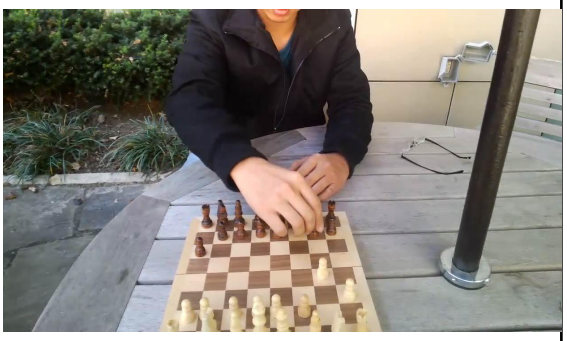
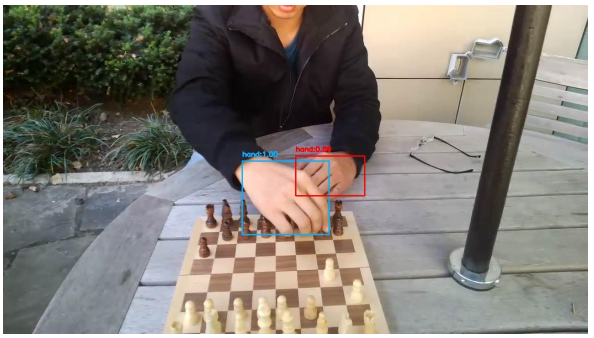
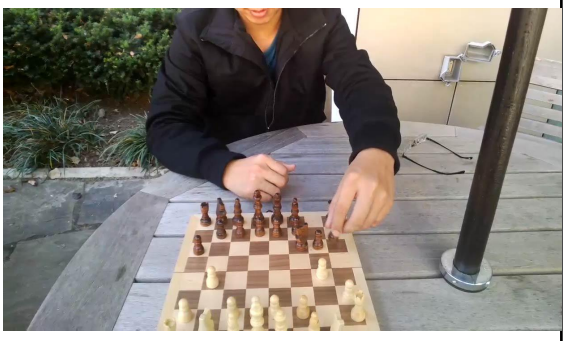
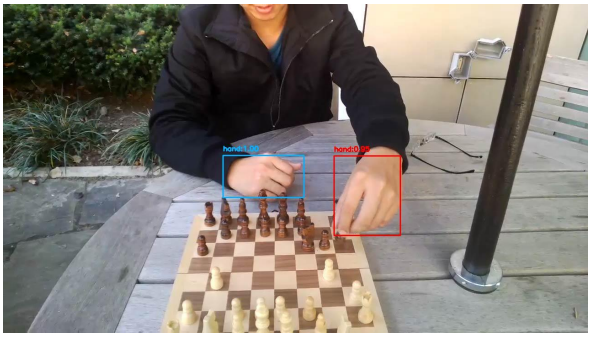
During preliminar test phase we have considered other approaches like Mean Shift clustering and Watershed algorithm, but both methods have not ensured good results.
Mean Shift clustering returns too many clusters and since the parameter K (number of clusters) is not tunable in this method, this doesn't help the hand segmentation process.
The second method was not great because often the grayscale image obtained by a processing step of the input image had several holes in the contours and for this reason the “water” spills over in all directions creating a wrong binary mask.

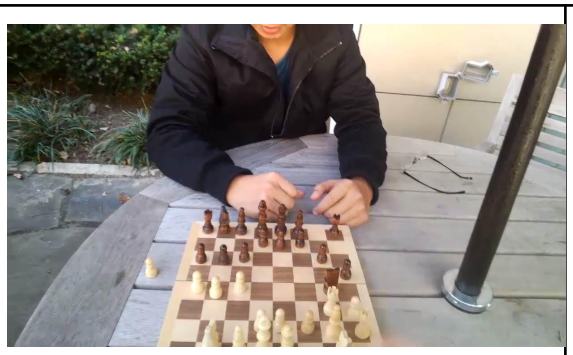
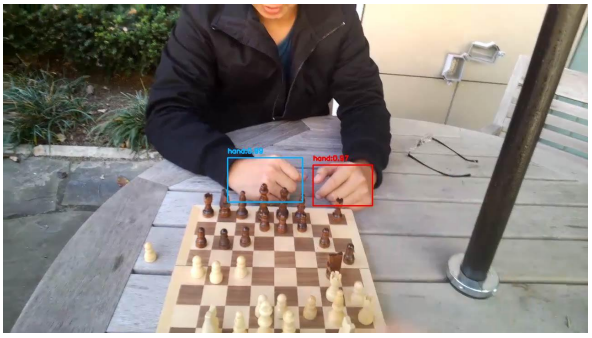
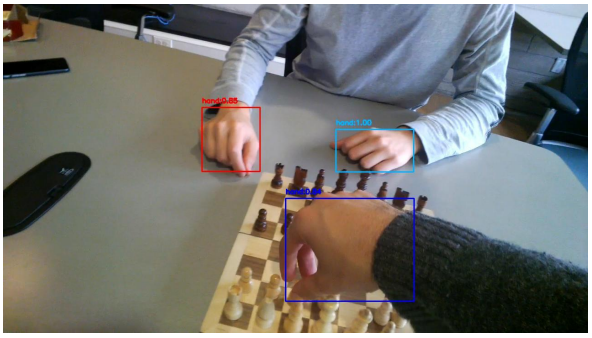
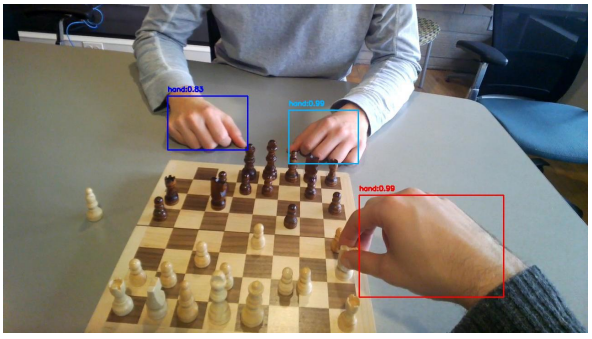
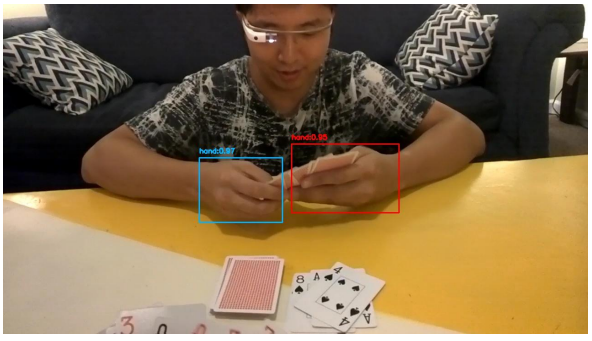
CHAPTER 3: Results

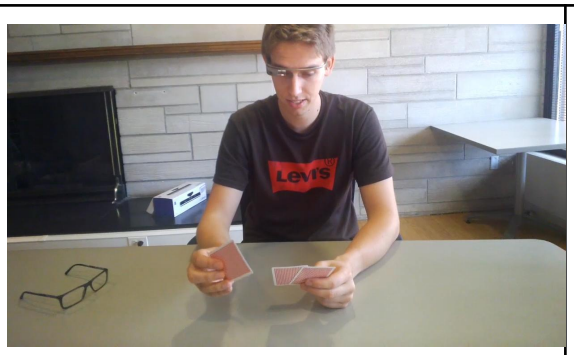
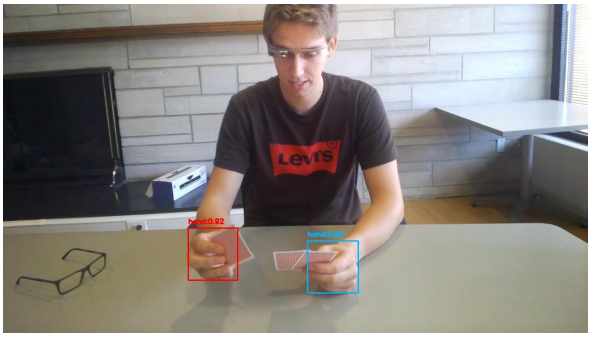
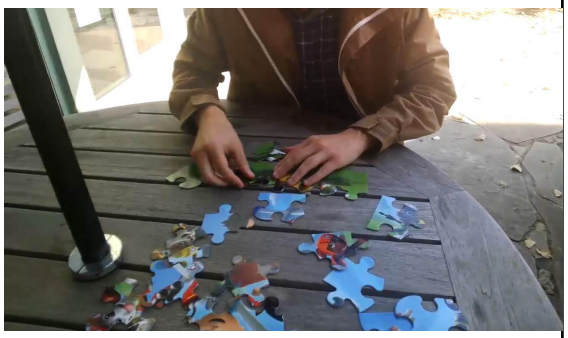
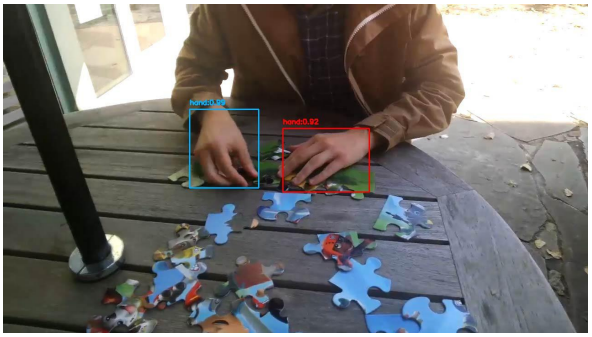
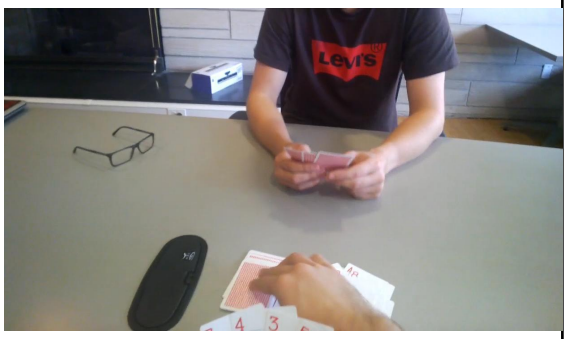
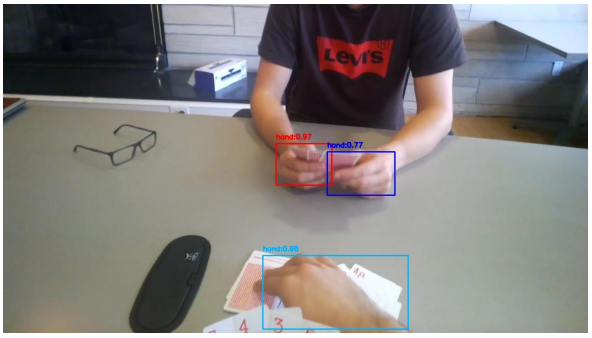
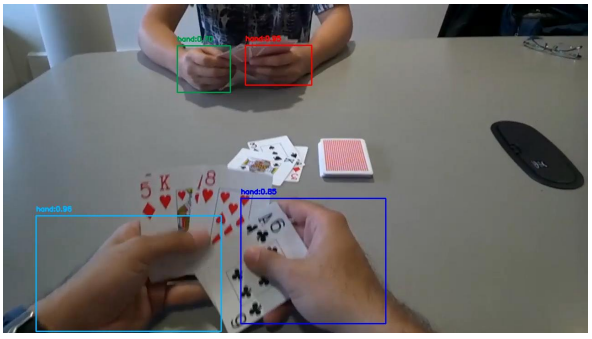
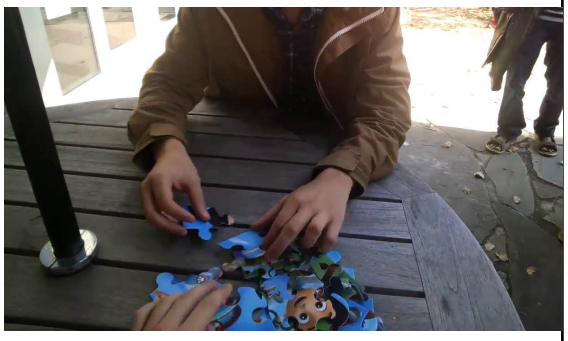
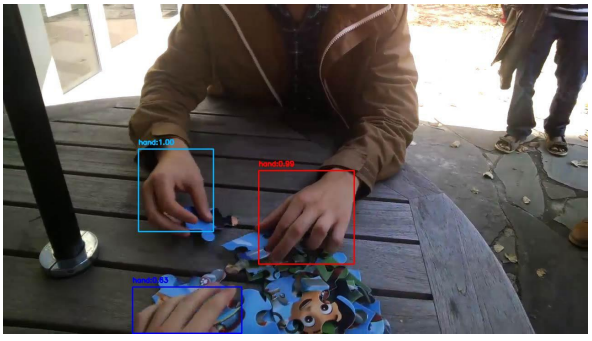
In this section are provided all the performance measures of our modules in order to see the performances of our system.

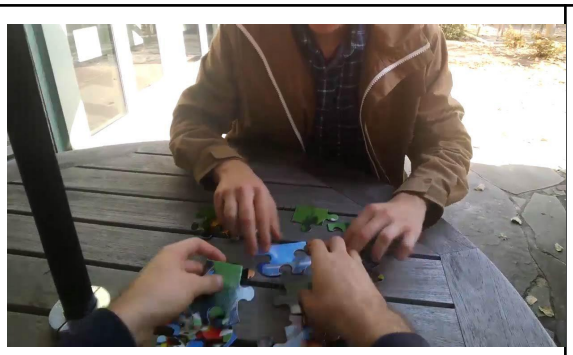
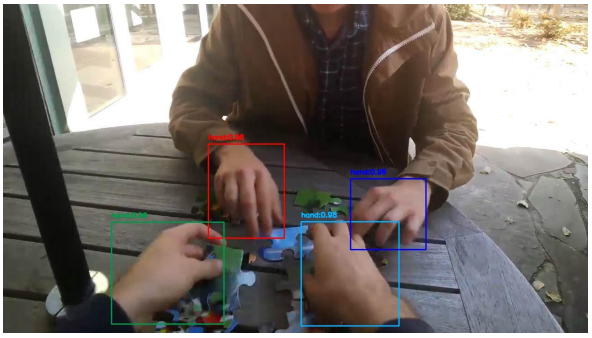
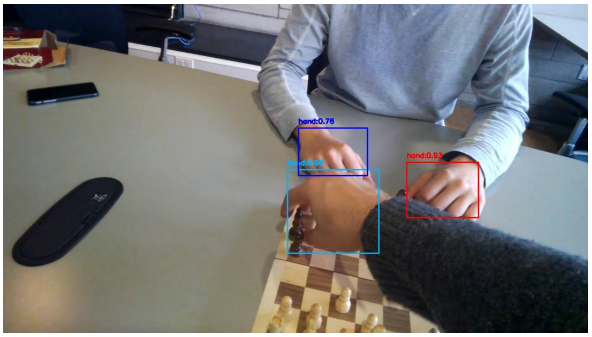
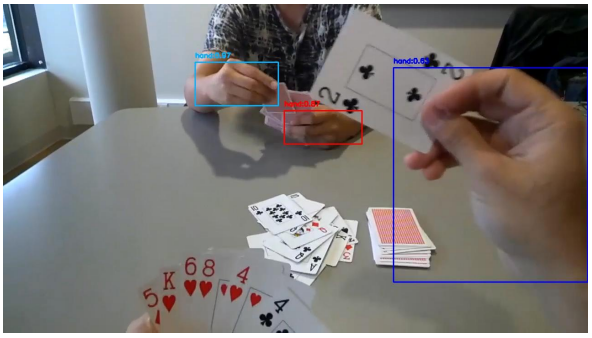
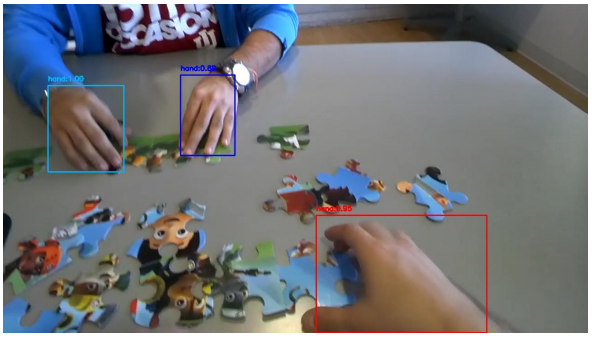
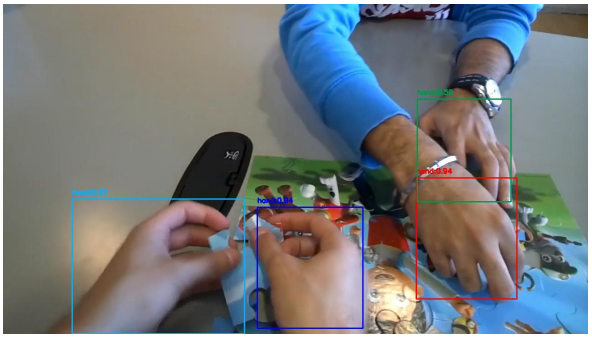
3.1 Detection Module Results: Intersection Over Union

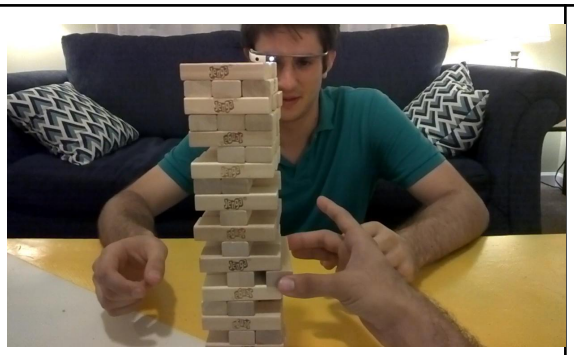
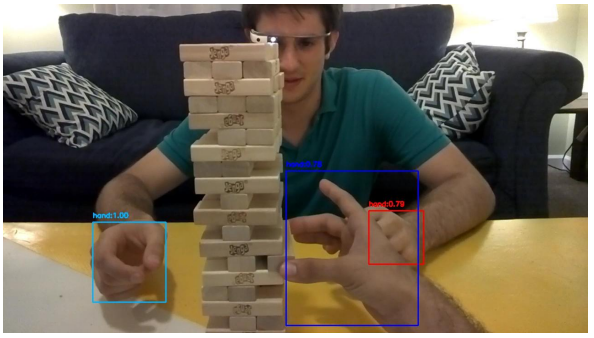
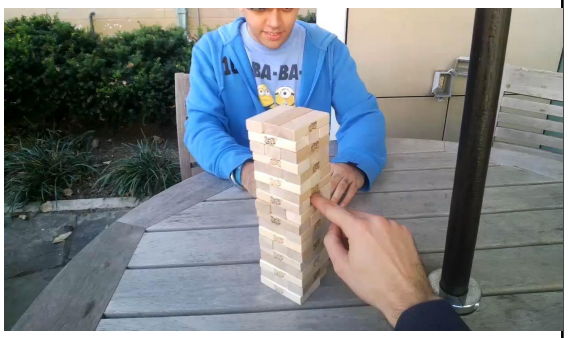
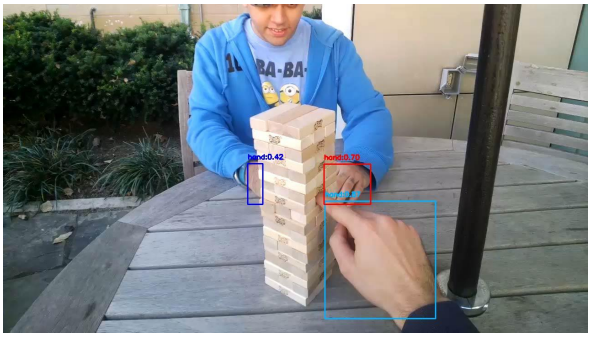
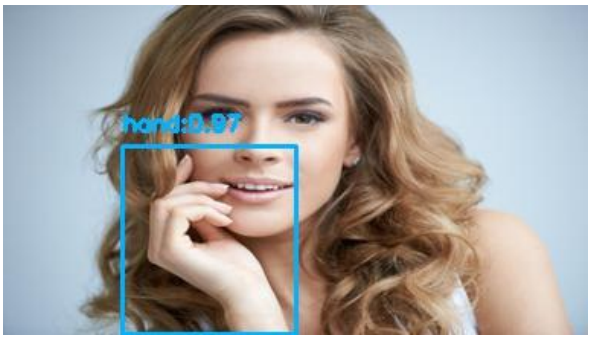
In the following table are listed the detection images returned by our HandDetector executed on the provided test set. The results will be listed below for each image of the test set:

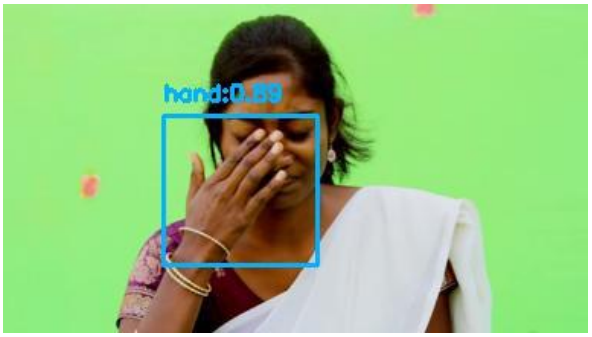
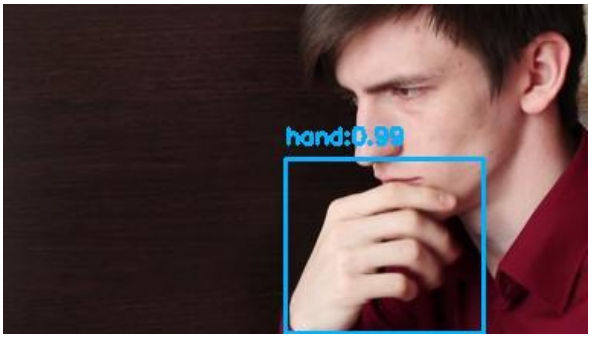
N. img:	Input Image	Detection Image
1		
2		
3		

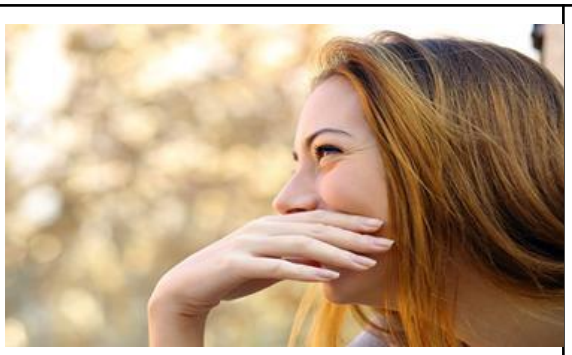
4		
5		
6		
7		
8		

9		
10		
11		
12		
13		

14		
15		
16		
17		
18		

19		
20		
21		
22		
23		

24		
25		
26		
27		
28		

29		
30		

In the following table are listed the corresponding Intersection Over Union measures obtained:

N. img	**Ground Truth Bounding Box**	**Intersection Over Union**
01.jpg	Coordinates Top Left Pixel: (631,318), Height: 122, Width: 217	0.708819
	Coordinates Top Left Pixel: (453,308), Height: 134, Width: 175	0.883364
02.jpg	Coordinates Top Left Pixel: (641,322), Height: 111, Width: 133	0.708617
	Coordinates Top Left Pixel: (518,339), Height: 166, Width: 201	0.916802
03.jpg	Coordinates Top Left Pixel: (727,322), Height: 185, Width: 153	0.857113
	Coordinates Top Left Pixel: (493,328), Height: 101, Width: 162	0.840041
04.jpg	Coordinates Top Left Pixel: (682,347), Height: 106, Width: 138	0.748398
	Coordinates Top Left Pixel: (489,331), Height: 106, Width: 172	0.872532
05.jpg	Coordinates Top Left Pixel: (669,316), Height: 102, Width: 151	0.882273

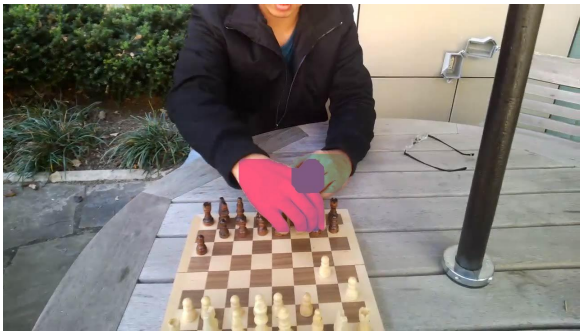
	Coordinates Top Left Pixel: (483,312), Height: 197, Width: 201	0.901155
06.jpg	Coordinates Top Left Pixel: (620,414), Height: 232, Width: 235	0.789166
	Coordinates Top Left Pixel: (730,273), Height: 94, Width: 181	0.908372
	Coordinates Top Left Pixel: (437,224), Height: 153, Width: 125	0.901829
07.jpg	Coordinates Top Left Pixel: (740,413), Height: 249, Width: 400	0.70751
	Coordinates Top Left Pixel: (627,221), Height: 124, Width: 164	0.800661
	Coordinates Top Left Pixel: (365,190), Height: 137, Width: 154	0.770784
08.jpg	Coordinates Top Left Pixel: (627,314), Height: 137, Width: 247	0.872635
	Coordinates Top Left Pixel: (430,339), Height: 135, Width: 187	0.923051
09.jpg	Coordinates Top Left Pixel: (668,533), Height: 93, Width: 109	0.801091
	Coordinates Top Left Pixel: (406,490), Height: 116, Width: 98	0.893188
10.jpg	Coordinates Top Left Pixel: (608,276), Height: 118, Width: 211	0.772578
	Coordinates Top Left Pixel: (421,234), Height: 173, Width: 143	0.842228
11.jpg	Coordinates Top Left Pixel: (557,545), Height: 173, Width: 276	0.758071
	Coordinates Top Left Pixel: (734,325), Height: 106, Width: 121	0.719823
	Coordinates Top Left Pixel: (598,302), Height: 102, Width: 121	0.885627
12.jpg	Coordinates Top Left Pixel: (68,469), Height: 251, Width: 375	0.873124
	Coordinates Top Left Pixel: (536,430), Height: 283, Width: 316	0.849471
	Coordinates Top Left Pixel: (525,93), Height: 90, Width: 162	0.811189
	Coordinates Top Left Pixel: (385,96), Height: 102, Width: 113	0.869537
13.jpg	Coordinates Top Left Pixel: (285,611), Height: 109, Width: 229	0.89057
	Coordinates Top Left Pixel: (554,365), Height: 202, Width: 226	0.916388

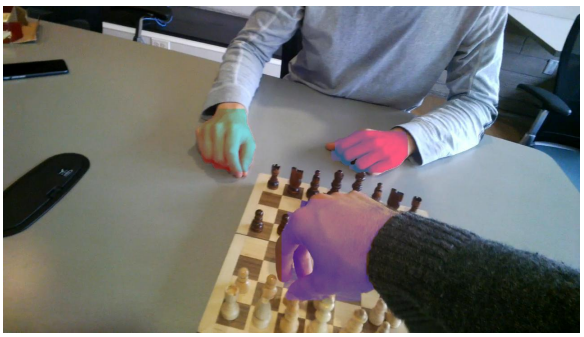
	Coordinates Top Left Pixel: (299,305), Height: 204, Width: 161	0.868081
14.jpg	Coordinates Top Left Pixel: (256,477), Height: 221, Width: 235	0.885936
	Coordinates Top Left Pixel: (659,473), Height: 221, Width: 202	0.891453
	Coordinates Top Left Pixel: (757,391), Height: 139, Width: 186	0.804906
	Coordinates Top Left Pixel: (459,312), Height: 200, Width: 153	0.894842
15.jpg	Coordinates Top Left Pixel: (617,366), Height: 169, Width: 214	0.867423
	Coordinates Top Left Pixel: (893,337), Height: 133, Width: 158	0.800351
	Coordinates Top Left Pixel: (654,271), Height: 116, Width: 152	0.805459
16.jpg	Coordinates Top Left Pixel: (264,670), Height: 50, Width: 61	0
	Coordinates Top Left Pixel: (873,142), Height: 436, Width: 405	0.88589
	Coordinates Top Left Pixel: (633,219), Height: 102, Width: 162	0.622043
	Coordinates Top Left Pixel: (440,127), Height: 101, Width: 160	0.832877
17.jpg	Coordinates Top Left Pixel: (654,482), Height: 236, Width: 432	0.801822
	Coordinates Top Left Pixel: (389,156), Height: 177, Width: 122	0.943439
	Coordinates Top Left Pixel: (97,181), Height: 183, Width: 181	0.890354
18.jpg	Coordinates Top Left Pixel: (150,430), Height: 286, Width: 360	0.923959
	Coordinates Top Left Pixel: (542,456), Height: 264, Width: 255	0.824496
	Coordinates Top Left Pixel: (917,192), Height: 251, Width: 207	0.798515
	Coordinates Top Left Pixel: (880,346), Height: 303, Width: 261	0.734418
19.jpg	Coordinates Top Left Pixel: (619,390), Height: 295, Width: 295	0.857248
	Coordinates Top Left Pixel: (785,447), Height: 126, Width: 130	0.793103
	Coordinates Top Left Pixel: (187,479), Height: 171, Width: 174	0.900178

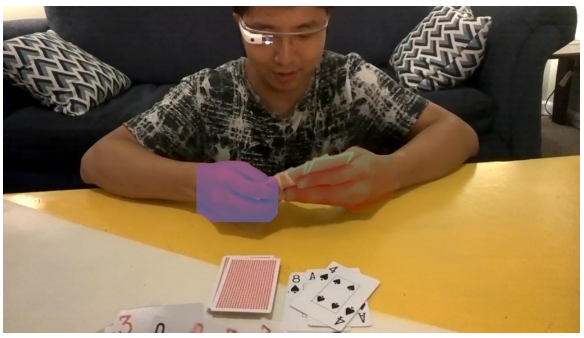
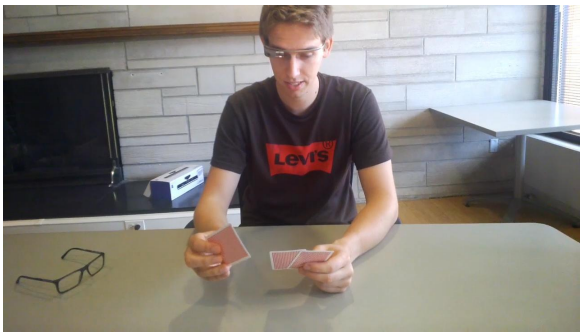
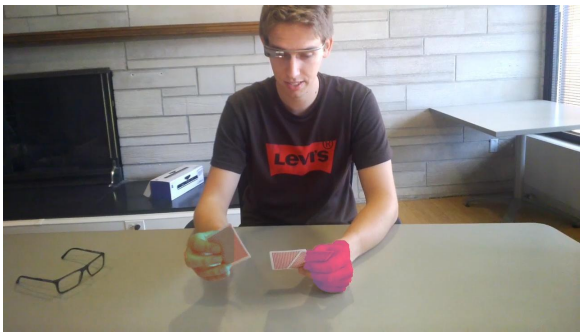
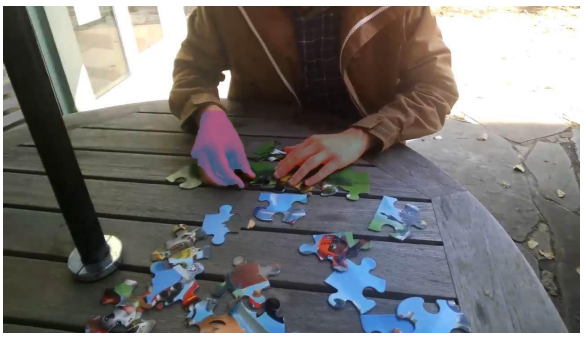
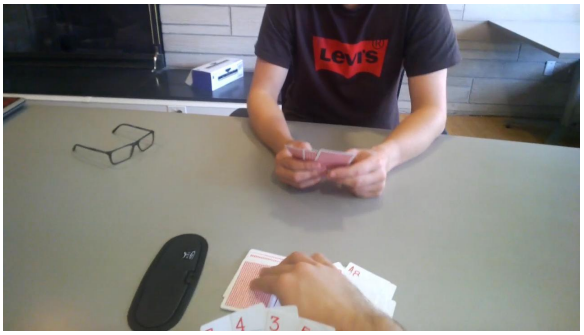
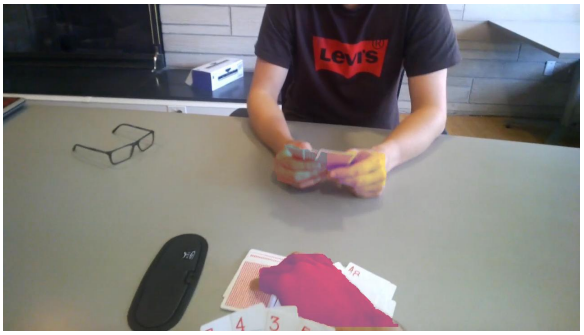
20.jpg	Coordinates Top Left Pixel: (686,413), Height: 256, Width: 266	0.792307
	Coordinates Top Left Pixel: (729,342), Height: 99, Width: 77	0.660066
	Coordinates Top Left Pixel: (544,348), Height: 77, Width: 22	0.565056
21.jpg	Coordinates Top Left Pixel: (231,112), Height: 104, Width: 76	0.574406
22.jpg	Coordinates Top Left Pixel: (145,41), Height: 75, Width: 38	0.728795
	Coordinates Top Left Pixel: (15,168), Height: 35, Width: 62	0
23.jpg	Coordinates Top Left Pixel: (94,96), Height: 99, Width: 81	0.571887
24.jpg	Coordinates Top Left Pixel: (110,81), Height: 101, Width: 76	0.637228
25.jpg	Coordinates Top Left Pixel: (89,115), Height: 93, Width: 141	0.553871
	Coordinates Top Left Pixel: (132,106), Height: 109, Width: 133	0.903448
26.jpg	Coordinates Top Left Pixel: (73,71), Height: 144, Width: 125	0.77737
27.jpg	Coordinates Top Left Pixel: (135,64), Height: 99, Width: 55	0.84472
	Coordinates Top Left Pixel: (230,100), Height: 66, Width: 82	0.800162
28.jpg	Coordinates Top Left Pixel: (188,115), Height: 99, Width: 114	0.761538
29.jpg	Coordinates Top Left Pixel: (95,124), Height: 78, Width: 167	0.70858
30.jpg	Coordinates Top Left Pixel: (89,90), Height: 69, Width: 68	0.785059

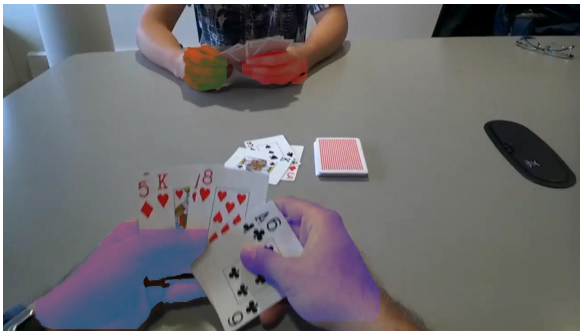
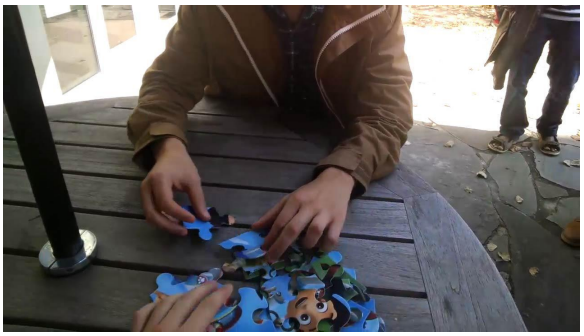
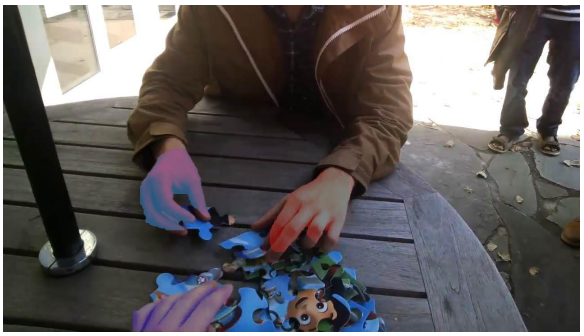
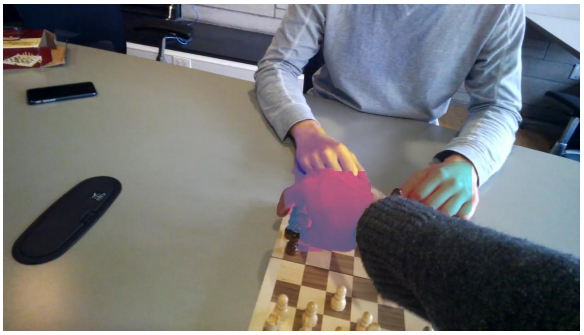
3.2 Segmentation Module Results: Pixel Accuracy

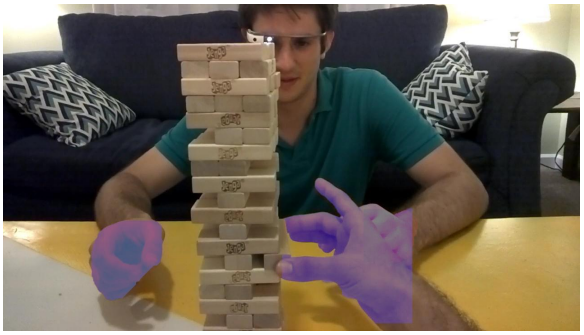
In the following table are listed the segmentation images returned by our HandSegmentator executed on the provided test set. The results will be listed below for each image of the test set:

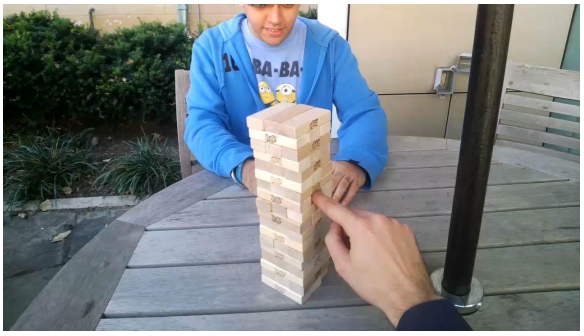
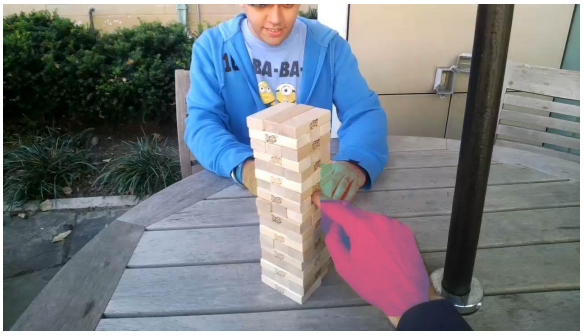
N. img:	Input Image	Segmented image and Pixel accuracy
1		 Pixel Accuracy: 0.990955
2		 Pixel Accuracy: 0.99537
3		 Pixel Accuracy: 0.992704

4		 Pixel Accuracy: 0.993916
5		 Pixel Accuracy: 0.989902
6		 Pixel Accuracy: 0.984336
7		 Pixel Accuracy: 0.97955

8		 Pixel Accuracy: 0.98621
9		 Pixel Accuracy: 0.991245
10		 Pixel Accuracy: 0.99143
11		 Pixel Accuracy: 0.984179

12		 Pixel Accuracy: 0.967499
13		 Pixel Accuracy: 0.982816
14		 Pixel Accuracy: 0.974462
15		 Pixel Accuracy: 0.982802

16		 Pixel Accuracy: 0.930139
17		 Pixel Accuracy: 0.979111
18		 Pixel Accuracy: 0.95045
19		 Pixel Accuracy: 0.981456

20		 Pixel Accuracy: 0.988971
21		 Pixel Accuracy: 0.955597
22		 Pixel Accuracy: 0.955681
23		 Pixel Accuracy: 0.912507

24		 Pixel Accuracy: 0.975514
25		 Pixel Accuracy: 0.933184
26		 Pixel Accuracy: 0.972331
27		 Pixel Accuracy: 0.94301

28		 Pixel Accuracy: 0.961733
29		 Pixel Accuracy: 0.962553
30		 Pixel Accuracy: 0.98499

CHAPTER 4: Failure Analysis

The main purpose of this chapter is to analyze where the detection and segmentation modules fail in the test set, in order to discover possible weak points of our approach and understand where to work on in the future.

4.1 Detection Module

Regarding the detection module, we can see that the overall hands are predicted well by the YOLOv3 Neural Network. However, there are two cases in which the network failed to predict hands, and we can notice this from the value set to 0 in the IoU metrics calculated in the given test set; the failed images are n. 16 and 22.

Looking at the ground truth and the predicted boxes of these images we can say that, for

- Img. 16: in this image the non-predicted hand is a very small box. Probably the network failed the prediction because of the small size of the box and because what is contained within it, which is a blob of pixels, is not an actual hand.
- Img. 22: probably in this image one hand was not predicted due to low contrast in and around the hand area.

4.2 Segmentation Module

Looking at the the results of the segmentation module we can identify some images in which our proposed algorithm segments other parts that are not hand points, for example:

- Img. 21, 22, 23 and 24: in these images the mask returned by the Advance Region Growing function covers both hand and face, probably because the central point of the RoI is placed exactly between these two parts so a starting seed point is placed also in the face. Another possible cause is that the clustered colors are equal in face and hand parts so just a seed point in the face's area will take the Advanced Region Growing in the wrong direction. Furthermore the skin mask also catches the face so the combined mask that is used by GrabCut alg. considers the face part of the foreground (i.e. hand).
- Img. 24 and 27: there is the problem described above, but in these cases the skin mask isn't good so, basically, GrabCut uses a mask with a large number of pixel with flag `GC_PR_FGD` and it returns a mask with no changes.

In some of the first 20 images our segmentation algorithm does not segment all hand's points: meaning that, in some cases, the terminal part of the fingers is taken out of the foreground part. This can be probably caused by the presence of shadows or, in general, of a non-uniform color within the rest of the hand that cause both skin detector and Advanced Region Growing to fail to provide a good approximation of all hand area. This problem could be caused also by the fact that sometimes the bounding box (returned by the HandDetector) around the hand is too tight so, as we implemented, all pixels outside this area are considered background (i.e. non-hand) points. Furthermore, many times the edge map is too noisy and blocks a lot the Region Growing process.

CHAPTER 5: Software Structure and Operation

The Project directory is composed as follows:

- `Include/`
  - `Evaluator.hpp`
  - `HandDetector.hpp`
  - `HandSegmentator.hpp`
- `Source/`
  - `Evaluator.cpp`
  - `HandDetector.cpp`
  - `HandSegmentator.cpp`
  - `main.cpp`
- `Models/`
  - `coco2.names`
  - `yolov3_training.cfg`
  - `yolov3_training_last_v7.weights`
- `Test/`
  - `det/`
  - `mask/`
  - `output_detection/`
  - `output_evaluation/`
  - `output_segmenation/`
  - `rgb/`
- `Product/`
- `CMakeLists.txt`

Where:

- `Include/` : directory that contains all the header files.
- `Source/` : directory that contains all the .cpp implementation files and the main.cpp file.
- `Models/` : directory that contains all the YoloV3 configuration files: classes file, weights file and network configuration file.
- `Test/` : directory that contains all the software run files, in particular:
  - `det/` : directory that contains all the detection ground truth files.
  - `mask/` : directory that contains all the segmentation ground truth files.
  - `output_detection/` : directory that contains all the HandDetector output images (images with the bounding boxes). This directory is filled only with the “de” (detection with evaluation) software execution mode (see next paragraph).
  - `output_evaluation/` : directory that contains all the Evaluation module output files. There are two possible files:
    - `resultsDetection.txt` : file that contains all the Intersection Over Union measures obtained from the images

in the `/rgb` directory. This file is produced only with the “de” (detection with evaluation) software execution mode (see next paragraph).
        - `resultsSegmentation.txt` : file that contains all the Pixel Accuracy measures obtained from the images in the `/rgb` directory. This file is produced only with the “se” (segmentation with evaluation) software execution mode (see next paragraph).
    - `output_segmenation/` : directory that contains all the HandSegmentator output images (images with the Hand masks). This directory is filled only with the “se” (detection with evaluation) software execution mode (see next paragraph).
    - `rgb/` : directory that contains all the rgb images that have to be processed.
- `Product/` : directory that contains the main executable file that is builded using CMake.
- `CMakeLists.txt` : file for the CMake building.

Since the system will ask for the absolute path of some directories that contain run files (such as the `/rgb` directory), these folders can be in every possible position in the PC.

After building the project with CMake, it’s possible to run the software (`main` file executable) in the `Product/` directory. The software will ask for the `rgb/` directory absolute path and then it will ask for one of the following execution modes:

- `d` : detection mode. The software will process all the images in the `/rgb` directory and will output on the video the images with the hand bounding boxes (with the corresponding confidence) one at a time. The output images will not be saved.
- `s` : segmentation mode. The software will process all the images in the `/rgb` directory and will output on the video the images with the corresponding colored hands mask one at a time. The output images will not be saved.
- `de` : detection with evaluation. The software will ask for the `/det` directory absolute path. It will process all the images in the `/rgb` directory and will produce the `resultsDetection.txt` file. All the output images with the bounding boxes (with the corresponding confidence) will be saved in the `output_detection/` directory.
- `se` : segmentation with evaluation. The software will ask for the `/mask` directory absolute path. It will process all the images in the `/rgb` directory and will produce the `resultsSegmentation.txt` file. All the output binary images with the hands masks will be saved in the `output_segmentation/` directory.

If one of the software execution parameters such as one directory path or one execution mode is wrong, the software will output an exception and the execution will die.

CHAPTER 6: Contribution

6.1 Files Developed and Implementations

- Manuel Barusco
    - Include/Evaluator.hpp
    - Include/Segmentator.hpp
    - Source/Evaluator.cpp
    - Source/Segmentator.cpp
    - Source/main.cpp
    - CMakeLists.txt

- Simone Gregori
    - Include/Segmentator.hpp
    - Source/Segmentator.cpp
    - Source/main.cpp

- Riccardo Rampon
    - Include/Evaluator.hpp
    - Include/HandDetector.hpp
    - Source/Evaluator.cpp
    - Source/HandDetector.cpp
    - Source/main.cpp

6.2 Hours of Work and implementations

- Manuel Barusco
    - Roughly 15 hours to solutions studying (read papers, documents and so on) and thinking about the best solution.
    - Roughly 20 hours to find the final solution for the project (parameters setting and pipeline) after several trials.
    - Roughly 30 hours for C++ coding including:
        - Development of the main Software Structure and connection of all the modules in the main.cpp file.
        - Development of my project part: implementation of the RoI preprocessing step and of the Advanced Region Growing technique in the segmentation module. Implementation of the Evaluator object and implementation of the Intersection Over Union computation.
        - Debugging and Troubleshooting.

    - Roughly 6 hours for report writing.

- Simone Gregori
    - Roughly 20 hours to solutions studying (read papers, documents, test possible solutions and so on) and thinking about the best solution.
    - Roughly 15 hours to find the final solution for the project (parameters setting and pipeline) after several trials.
    - Roughly 30 hours for C++ coding including:
        - Development of the main Software Structure and connection of all the modules in the main.cpp file.
        - Development of my project part: implementation of K-Means based on pixel color and position, implementation of all the functions/ideas that are present in the GrabCut segmentation algorithm used in the segmentation module.
        - Debugging and Troubleshooting
    - Roughly 5 hours for report writing

- Riccardo Rampon
    - Roughly 10 hours to solutions studying (read documents, guides and so on) and thinking about the best solution.
    - Roughly 25 hours to find the final solution for the project after several trials.
    - Roughly 30 hours for C++/Python coding including:
        - Development of the main Software Structure and connection of all the modules (header file, .cpp files and so on).
        - Development of my project part: training of the YoloV3 Neural Network, data augmentation and implementation of the detection module in C++. Implementation of the Evaluator object and implementation of the Pixel Accuracy computation.
        - Debugging and Troubleshooting
    - Roughly 5 hours for report writing.

CHAPTER 7: Conclusions

In this chapter we discuss the obtained results and we point out where to work on in the future in order to improve our approach.

7.1 Results discussions

Concerning the Detection Module the results are very good, only a few hands are not detected and the system is quite fast in the detection, so we are very satisfied with the results.

Concerning the Segmentation Module, despite our solution does not use any state of the art Neural Network for the segmentation task, the results are quite good. As we can see from Chapter 3 the outputs have high pixel accuracy. Even if we know that an high value of pixel accuracy does not always imply superior segmentation ability, visually the hands are catched in all images with quite good accuracy.
Running and testing our system we have noticed that in large images (the first twenty test set images) the GrabCut algorithm takes a few seconds to compute the final result, this is due to the fact that it creates a graph for the whole image and executes Min-Cut procedure on it. We have decided to do that because we have thought that with bigger input it provides more accurate results since it analyses all the image.

7.2 Future Work

Concerning the Detection Module we see that the network created works very well, but to get 100% accuracy one improvement could be train the Neural Network on more specific hands datasets, so that the predictions would generalise better. Another solution will be to try to use other Neural Networks, different from YOLOv3, for example the R-CNN Model Family or other version of YOLO Model Family, which are more modern than v3. Furthermore a more specific dataset analysis could be performed and a more powerful data augmentation technique could be done.

Concerning the Segmentation Module, in order to fix all the issues pointed out in Section 4, we could develop a better skin detector that tries to learn a set of thresholds given the input image and does not use fixed ones; this could help to segment all skin's points also in case of dark complexion skin. Improvements can be done also in the way Advanced Region Growing calculates the seed points to avoid face parts and non-hand points. Furthermore instead of using a simple edge map we could use a Hand Contour searching technique to discover the hand contours (by using for example an active contour technique). Finally we could combine other techniques to make the system more invariant to brightness changes, that is one of the weaknesses of our proposed algorithm.
All the improvements must be done in order to speed up the system because the system speed is another weak point.
We will also try to use an ad-hoc Neural Network for the object segmentation task, such as a Mask R-CNN or the UNET. We will try to train them in our training set and we will adapt them for our task in order to see also their performances on this job.