CV project : hand detection and segmentation

Email: mattia.secchiero@studenti.unipd.it,
riccardo.demonte@studenti.unipd.it
simone.cecchinato.2@studenti.unipd.it,
Components: Simone Cecchinato (2052407),
Riccardo De Monte (2039312), Mattia Secchiero (2053075)

July 28, 2022

Contents

1 Introduction

In the following sections will be presented the program structure, namely how the program has been organized, developed and which libraries have been implemented. It is also shown how to compile and run the program. In the next paragraphs we have also introduced a brief explanation of the ideas behind our project (paragraph 2), some considerations about the results obtained (paragraph 3), the amount of work done by each member (appendix A) and finally some attempts tried (appendix B).

1.1 Program's tree

In Fig. 1 it is shown how the program has been organized. Inside the $cecchinato\_demonte\_secchiero$ folder we find four sub-folders:

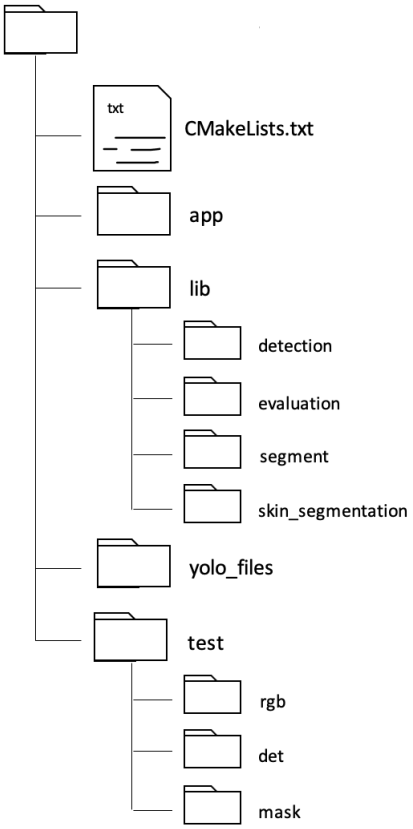


Figure 1: Structure of the program

- app : it contains the main $.cpp$ application file;
- lib : it contains the libraries that were implemented. In the $detection$ library it is implemented the $detector\ class$ for hand detection, in $segment$ and $skin\_segmentation$ we have the libraries necessary to implement the $segmentation\ task$ and in the $evaluation$ folder the class that allows to evaluate the results w.r.t. the given masks and bounding-boxes (to be found in det and $mask$ folders);
- yolo_files : it contains the .weights the .cfg. file, necessary to make the detector run.
- test : it contains the RGB images to be tested, their bounding-boxes and the relative masks.

In order to automatize the compiling and linking process we also created some cmake files that create public libraries and link them to the main application, beyond building the executable. A general CmakeLists.txt file has been created in order to add as executable the main program that is located inside the app folder and to link the libraries from the lib folder. In the next section it is shown how to build the program from the terminal.

1.2 Running the application

To run the application it is necessary to create a folder in *cecchinato_demonte_secchiero*, in which run CMake. The following code can be used by console (assuming that you are in the folder *cecchinato_demonte_secchiero*).

```
mkdir build && cd build
cmake ..
make
```

After the conclusion of the make operations, a folder *app* is created in the folder *build* with the executable file. In order to execute the program it is necessary to move into the *app* folder

```
cd app
```

There are three modes to run the software (before, make sure that the .weights file is present in the $yolo\_files$ directory). In all the cases the program will output three images: one with the original image, one figure with all the detected hands in the image, and one with the segmented hands (see Figure 2)
Mode 1 and Mode 3 print also the evaluation results on the terminal.

Mode 1: Test Image mode

We pass as inline parameter only the name of the image that is in the *test/rgb* folder. In the following code-box we provide an example with the image *01.jpg*

```
./app 01.jpg
```

It is possible to add test images with the associated ground-truth. It is very important to insert the image in the *test/rgb* folder, the true mask image in the *test/mask* folder and the true bounding boxes text file in the *test/det*. All three files must be uploaded, all with the same name (the image can be of any extension, but the mask has to be a *png* file and the bounding box file a *txt* file).

Mode 2: No ground-truth mode

This mode is used to test the hand detection with images that do not have the ground-truth. It is sufficient to pass two inline parameters: the path of the image and another arbitrary string.

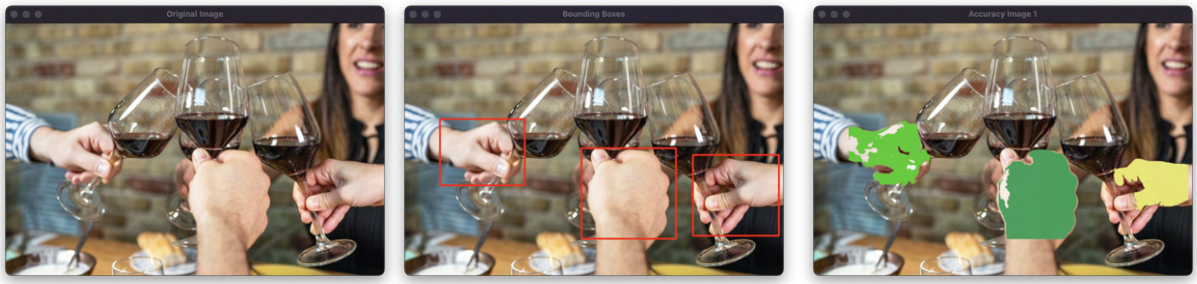
```
./app ./../../test/rgb/01.jpg a
```

With this mode the results of the evaluation metrics are not printed on console.

Mode 3: Paths mode

In this mode we pass as inline parameters the path of the image, the path of the ground-truth mask and the path of the ground-truth bounding boxes (in this order)

```
./app ./../../test/rgb/01.jpg ./../../test/mask/01.png ./../../test/det/01.txt
```



(a) (b) (c)

Figure 2: Output Example

2 Methodology

2.1 Detection

In order to perform hands detection, the Yolo algorithm is used. In particular, the fourth version Yolov4 is taken into account. The reason why our solution is based on Yolo is because of its high speed. In fact, Yolo is a Single-Stage Deep Learning based algorithm, by the meaning that it doesn't exploit any region proposal algorithm as Selective search (or, like Faster R-CNN does, the CNN itself proposes regions). In the next chapter (Segmentation) will be clearer why a fast detection is a relevant feature to our solution (Note: since a GPU is not available, the speed of the detector is limited). The reason why the fourth version is taken into account is because it is one of the best object detector in terms of performance with a documentation that attests this fact (for instance, there is no a clear experiment that proves that the fifth version Yolov5 is better than the "previous" version. The authors compares a light version of Yolov5, called Yolov5s, with Yolov4). Moreover, the biggest advantage of Yolov5, the dimension of the file with the weights for the CNN, is not so significantly clear when high precision is needed (there are different models for Yolov5, and the ones with higher mAP for COCO have .pt files with dimension comparable to the .weights file for Yolov4). The other advantage of Yolov5 is that is trainable using the Pytorch, but this fact is not so relevant in our case and we prefer the framework Darknet (Note: on 6-th July Yolov7 is published). By now, the training procedure will be described.

Since a GPU speeds up the training process for a CNN, the training is performed on the Google Colab platform. The notebook used can be accesed by this link. Since a large dataset (images with annotations) is no available, a fine-tuning of the weights of the CNN is performed starting from the ones pre-trained with the COCO dataset (a famous dataset with 80 object categories).

2.1.1 Training dataset

The training set considered is made of 486 images: all the ones of the Hands on Face dataset, some of the frames of the other dataset and other fifty images downloaded from internet (all the test images are excluded). Using this link you can acces to a Google Drive directory with the obj.zip with all the images used for the training and the corresponding annotations for the bounding boxes. For a given image the annotations consist in many sets of 5 numbers (a set for each bounding box) saved in a .txt file (the file has the same name of the image e.g. 200.jpg and 200.txt). These 5 numbers are (in this order):

- class-id: an integer number in the interval $[0, N-1]$, where N is the number of object classes. In our case $N = 1$ (Hand class). This number identifies at which class the object of interest belongs to.
- x: a float value relative to width of the image and it is the x coordinate for the center of the bounding box. This means that $x \in (0, 1]$.
- y: a float value relative to height of the image and it is the y coordinate for the center of the bounding box. This means that $y \in (0, 1]$.
- w: a float value relative to width of the image and it is the width of the bounding box. This means that $w \in (0, 1]$.

- h: a float value relative to height of the image and it is the height of the bounding box. This means that $h \in (0, 1]$.

An example: 0 0.185833 0.327778 0.368333 0.595556 (<class-id> <x> <y> <w> <h>). For a given image, for each bounding box a line of this type is present in the corresponding .txt file.

Figure 3

To create such a file, the python program LabelImg is used for this purpose (Note: lxml and pyqt5 python modules are needed). This software provides a user interface to create the bounding box and it automatically creates an appropriate file with the annotations (different formats are supported, also the Yolo one).

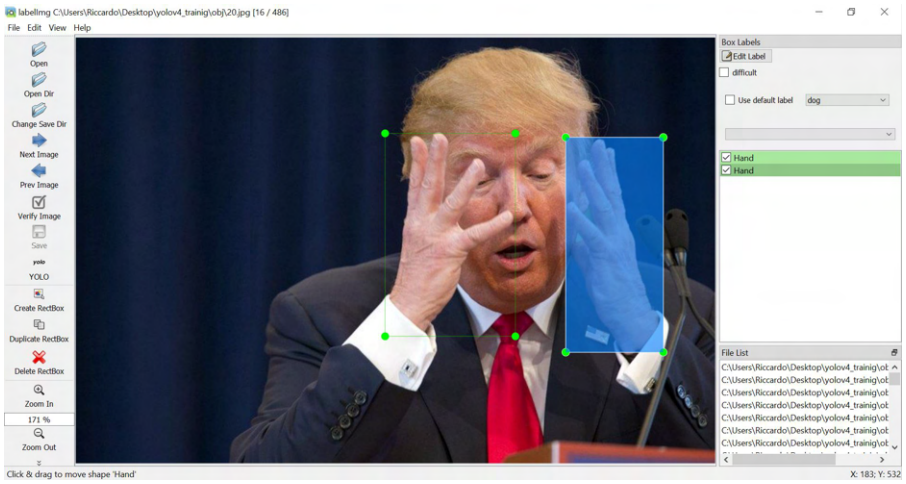


Figure 4: LabelImg example

The program needs also a file .classes in which the different labels for each class are provided (in our case only the label 'Hand').

2.1.2 Darknet configuration

In order to perform the training, Darknet requires 4 files:

- a .data file
- a .names file
- a train.txt
- a .cgf

In the Colab notebook a description for each file is provided (why they are needed and what to write). The most important file is the configuration file yolov4.cfg, in which both the architecture of the CNN and the training parameters are specified. About the CNN architecture, for example it is specified how many convolutional layers are present, the number of filters used in each convolutional layer, the stride used, the padding used (if used), which activation function is used at each layer, how many max-pool layers, how many residual connections, etc. About the training parameters it follows a list of the most important parameters (for the optimizer, for the input layer and for data-augmentation):

- **width** and **height** = 416
- **channels** = 3
- **batch** = 64: the number of images per batch (batch for the SGD with momentum)
- **subdivision** = 16: number of images per mini-batch. A GPU processes a mini-batch of 16 images at once.
- **learning_rate** = 0.001 (initial learning rate)
- **momentum** = 0.949
- **policy** = steps
- **steps** = 4800,5400
- **scales** =.1,.1. Note: since the 4800 iterations has not be reached, the learning rate has been the same for the entire training. (The next parameters are about data augmentation)
- **mosaic**=1: 4 images are joined together in one
- **saturation** = 1.5: randomly the saturation of images is changed during training
- **exposure** = 1.5: randomly the exposure of images is changed during training
- **hue** =.1: randomly the hue is changed during training

For more details about the meaning of the parameters, visit the GitHub pages: .cfg net parameters and .cfg layers parameters. The .cgf used is present in the directory *yolo_files* (in the Colab notebook, it is described how it is obtained).

2.1.3 Training and validation

The pre-trained weights for the convolutional layers (trained with COCO and 80 classes) have been downloaded by the GitHub repository
https://github.com/AlexeyAB/darknet/releases/download/darknet_yolo_v3_optimal/yolov4.conv.137.

As specified before, the dimension used is 416. To obtain better results, also other values (any multiple of 32, like 608, 512, ...) have been taken into account, but unfortunately with those higher values the training have been interrupted too early due to RAM or GPU memory saturation.

Darknet allows to save the weights every 1000 iterations. After more than 6 hours for training (blocked due to Colab), four .weights files are obtained: the one for 1000 iterations, the one for 2000, the one for 3000 and the last one (less than 4000 iterations reached). To choose properly the best weights, a validation is performed using other 80 images (with annotations) from the following dataset (link). The metrics taken into account are

- average IoU (a 0.25 threshold value for the confidence is used)
- $\text{mAP}_{0.5}$ (the minimum IoU to consider a positive match is 0.5)

By using Darknet, for each .weights file obtained the corresponding mAP and average IoU are computed. The following plots show the results

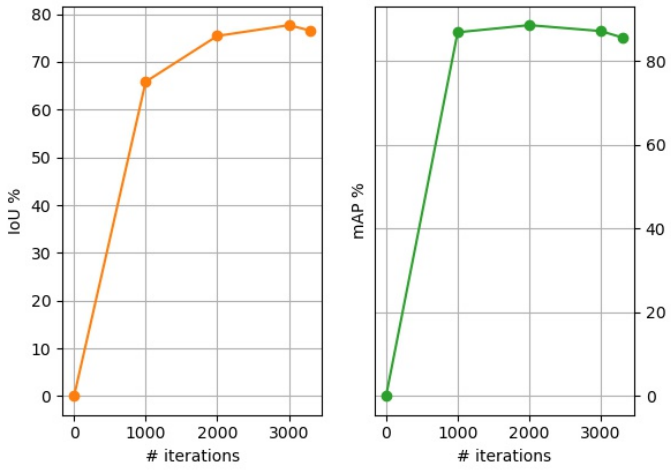


Figure 5: mAP and average IoU for the different .weights files

As Figure 5 shows, the best weights are the ones obtained at 3000 iterations for the average IoU metric and the ones obtained at 2000 iterations for the mAP metric (Note: after 3000 iterations overfitting starts). In particular

	mAP %	ave. IoU %
2000 iter.	88.64	75.44
3000 iter.	87.22	77.7

Table 1

The choice fell on the weights obtained at 3000 iterations since the results obtained with the test images (the 30 images used for testing) are the best. In fact, for some images the detector with the weights obtained at 2000 iterations is not able to detect hands. For instance (test image 22.jpg)

(a) 22.jpg and weights obtained at 2000 iterations

(b) 22.jpg and weights obtained at 3000 iterations

Figure 6: Detection results with 22.jpg

2.2 Segmentation

The image segmentation task, that is capture important grouping regions in an efficient way, can be performed using different approaches such as the watershed algorithm or the k-means clustering technique. But we could also consider more advanced solutions such as CNNs and deep learning techniques that represent the state-of-the-art for the segmentation task.

2.2.1 Approaches to the segmentation problem

During the software development we tried different approaches: the first idea was to try implementing a middle level algorithm based on watershed and k-means. The problem with that approach was that we weren't able to find a general rule in order to correctly cluster the pixels of the hand with respect to the others. It was hard to find a rule that could be applied to all images because of the diversity of the dataset and because of the algorithm that has been used: k-means needs to know in advance the number of clusters and will find only circular shaped clusters while watershed can lead to oversegmentation. To improve the results given by those algorithms it was also implemented a preprocessing of the images, for example histogram equalization and some filtering but it wasn't enough to have good results. So we implemented a new algorithm based on a paper research in order to segment an image based on the skin RGB, HSV and YCrCb intensity values (link at skin segmentation paper). The algorithm is pretty straightforward : given a specific threshold for those values we can decide if the intensity level of a pixel is considered as skin or not, so were able to correctly classify most of the pixels. At this point arises two problems : first of all, this kind of approach is based on the hypothesis that the image is given in an RGB format (so it can't be applied to a gray-scale image); and second, it happens that the surrounding environment can be confused as skin. That is why we needed to introduce another segmentation method to avoid, or at least reduce, those problems. That is why we used an already implemented algorithm based on graph-theory, developed by $P.Felzenszwalb, D.Huttenlocher$ (link at graph-based segmentation paper and code). In this algorithm is defined a predicate for measuring the evidence for a boundary between two regions using a graph-based representation of the image and then it is developed a segmentation algorithm based on this predicate.

In the end we created the class `HandSegmentor` which combines the two algorithms and returns the intersection of the two results, if both can be applied, to get the best result.

2.2.2 Brief code explanation

A more detailed explanation of how the segmentation was implemented follows: as we have already discussed, the skin segmentation technique could not always return a correctly segmented hand, that's why we introduced another segmentation method. But once the graph-based segmentation is applied we have to understand which segmented region is the one associated to the hand. So we made some functions (`HandSegmentor::finals_masks`, `HandSegmentor::get_masks_per_region`, `HandSegmentor::get_idx_of_regions`, ...) that, for each bounding box of the region of interest (where the hand is located), iterate over all the regions found by the graph-based algorithm. At this point for each class a mask is created for that segmented part, then that mask is applied to the original image (only to the area of interest in which the hand is detected, namely the zone delimited by the bounding box) and the detector is runned again. The idea is that if the class that we are considering is the one associated to the hand then the detector should find

again a bounding-box, once the original pixels are substituted to that mask. That is how we were able to determine which of the segmented classes were the one associated to the hand. We have noticed that when applying the detector to the bounding-box sized images we had some troubles while identifying the hands so we also created a function `HandSegmentor::get_expanded_roi()` to avoid that the detector won't find a hand even if there is one. With that function we just slightly increase the bounding box area.

Now, we have to consider another problem. The graph-based segmentation can fail or the hand could be segmented in different classes, so, how can we avoid that? For that purpose the function `HandSegmentor::test_region()` compares the area of the bounding-box given by the detector on the original image w.r.t. the bounding-box found by the detector on the cropped image. If the new bounding-box is at least 50% of the original one than it is considered as a part of the hand. This choice is justified by the fact that inside the bounding box most of the pixels should be associated to the hand. So by comparing the bounding-box dimensions we can derive this information. We could also fall in the case where the detector finds more than one bounding-box for the same region. In that case we consider that the sum of the areas of all the bounding-boxes has to be 60% of the area of the original bounding-box. Moreover if more than one region is labeled as hand by the detector then we consider the union of those masks.

At this point we can put together all the functions and verify if we can apply either both the skin and graph segmentation or just one of them. For this reason we evaluate four cases:

- if it has not been recognized any segmented region as hand and the image is not in gray-scale (at the end of the paragraph is explained how a gray-scale image is recognized)
- if it has been recognized at least one region as hand and the image is not gray-scale
- if it has been recognized at least one region as hand and the image is gray-scale
- if the detector fails and do not recognize a hand and the image is gray-scale

In the first case we just apply the skin segmentation since we have a 3 channels images (RGB) but no segmented region is classified as hand. In the second case instead we consider the intersection of the masks given by both the skin and graph-based segmentation given the fact that the image is not gray-scale and the detector finds at least one segmented region that is classified as hand. In the third case we just apply the graph-based segmentation since the detector recognizes at least one segmented part as hand but the image is in gray-scale, so we can't access to the RGB, HSV and YCrCb values to understand if it is a skin part or not. In the fourth case we take into account an heuristic method: when we consider a bounding-box that contains an hand we expect that the vast majority of the pixels inside that region will belongs to the hands itself. So, if we consider the biggest segmented region as the mask of the hand then we should be considering an accurate result. The problem with that approach is that we are finding the final mask based only on the graph-based segmentation (for instance if the segmentation brings to oversegmentation the biggest region might not be the one with the hand and for this reason this method is used only as last option). With the purpose of highlight the cases in which the graph-based segmentation plus detector doesn't work (still in the fourth case), the heuristic method can be "disabled" and all the pixels (the ones inside the bounding box) are selected (in this way a rectangular colored box is shown over the hand). Therefore,

if you are interested in verifying when the heuristic method is applied and when not, you have to comment the line 310 and uncomment line 312 in the file $segment.cpp$ (in the $segment$ library).

An example is given by running the program considering or not the heuristic method with the image $22.jpg$. Since the image is in gray-scale and the detector can't find one of the two hands after the segmentation output, we will find one hand covered by a rectangular shape because all pixels in the bounding box has been set as hand's pixels. If we instead consider the heuristic approach the hand is almost perfectly covered since we considered the procedure described before (take the biggest area from the one found from the graph-segmented ones).

That can be observed in the following image :

(a) 22.jpg without heuristic method

(b) 22.jpg with heuristic method

Figure 7: 22.jpg with and without heuristic method

One last clarification has to be made. Actually all the images are uploaded in the program as three channels RGB images. So, how can we understand if the uploaded image was converted before to a single channel gray-scale image?
We observe that an RGB image that was saved before as a gray-scale image is such that $R = G = B$ for each pixel. That is why we created a new function `is_Greyscale()` that given a $batchsize$, namely the number of pixels to be checked, randomly checks if that $\#batchsize - pixels$ are such that the R,G,B intensity values are all equal.

2.3 Evaluation

For the evaluation of the performance of our program we have to consider two different approaches:

- Intersect over Union (IoU) which is a metric used to measure the accuracy of the object detector
- Pixel Accuracy to validate how well the segmentation process was done

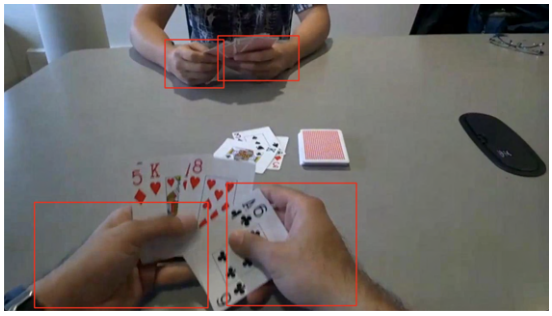
The class contains all the methods for the performance evaluation.
It is important to know that if the software is used without the ground-truth (See section 1.2), the `Evaluation::IoU` method only returns the image with the detected bounding box (using both the modes), the `Evaluation::PixelAccuracy` method returns a vector with the accuracy image and an image with the mask of the segmented part. Obviously without the ground-truth all the results of the metrics are not provided.

2.3.1 Intersect over Union (IoU)

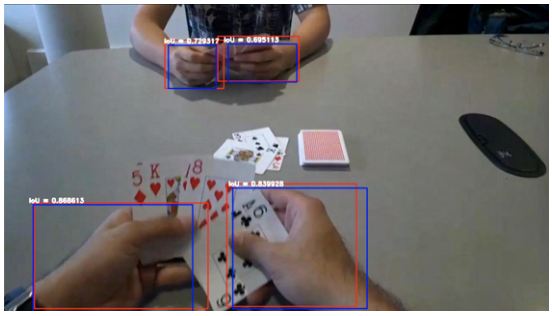
The Intersection-Over-Union (IoU), also known as the Jaccard Index, is a metric used to measure the accuracy of an object detector. Given the object detector, consider the ground-truth bounding boxes and the bounding boxes predicted by the object detector. The IoU value is simply the area of overlap between the predicted segmentation and the ground truth, divided by the area of union between the two.

$$\text{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}} \tag{1}$$

The first step for calculating the IoU is to extract from the text file, the ground-truth bounding box. After doing this, each detected bounding box has to be assigned with a real ones. In the class `Evaluation`, the method `Evaluation::ValidateBoundingBox`, called in the constructor, performs all those preliminary operations.
The method used to couple each bounding box found by the object detector with a real one, is by the IoU. In particular an algorithm was designed to assign for each ground-truth bounding box the detected box that maximizes the IoU among all the possible combination.
The `Evaluation::IoU` method accepts as a parameter an integer which indicates the mode. With $mode = 0$, the function returns an image with only the detected bounding box and the IoU results are printed on console. With $mode = 1$ is returned an image that shows the real bounding box (in blue), the detected bounding box (in red) and the IoU result for each box.



(a) 12.jpg IoU method with mode 0



(b) 12.jpg IoU method with mode 1

Figure 8: Comparison of the two modes of method Evaluation::IoU

A problem arises if the number of boxes detected by the object detector are less than the real ones. In particular, if in an image there are some hands crossed, it is possible that the detector can only assign a single bounding box that contains both hands. This is the case of the test image *25.jpg* (Figure 9) . A possible solution to this problem is to assign for both the two real bounding boxes of the crossed hands, the same detected bounding box.

Figure 9: Test image 25.jpg with the detected bounding boxes

This trick can be extended for all the other images: if the number of detected hands are less than the real ones, first we assign the detected bounding box using the method listed above, and then, for the remaining decoupled real boxes, we make another copy of the detected box that maximize (again) the IoU. In this way we still have IoU $= 0$ for the hand that are not detected but we solve the problem of the detected bounding box that contains more than one hand.

However, in our program we decided to not use this trick to have more fair results. As you can see in Table 2, for the image *25.jpg* we have IoU $= 0$ for the second hand.

2.3.2 Pixel Accuracy

The pixel accuracy represent the percent of pixels that are classified correctly. Let us introduce the *true positives* as the number of pixel correctly classified as hand points and the *false positives* as the number of pixel erroneously classified as hand. Analogously, *true negatives* and *false negatives* are the pixels that are correctly and incorrectly classified as non-hand pixels respectively.

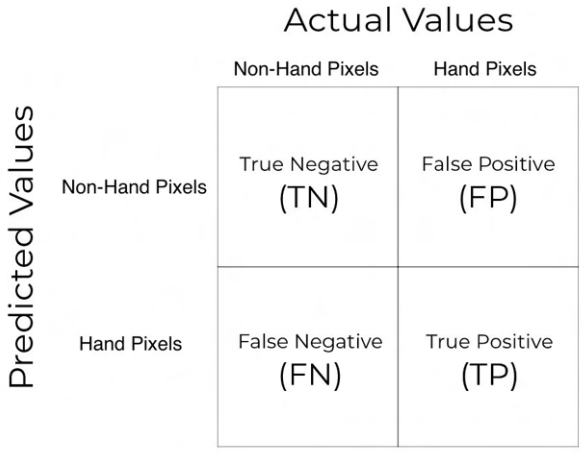


Figure 10: Confusion Matrix

The pixel accuracy is calculate as

$$\text{Accuracy (ACC)} = \frac{\text{TP} + \text{TN}}{\text{P} + \text{N}} \tag{2}$$

where TP and TN, as represented in Figure 10, are the True Positive and True Negative, P and N represent the real positive and negative points (in particular P+N = TP + TN + FP + FN).
Accuracy can be a misleading metric for imbalanced data sets. In the case of hands detection, if the result of the detection and the segmentation is such that there are no hand pixels, the accuracy value can still be over 90% even in images in which there are more than one hand.
For this reason a tuned version of the accuracy metrics is considered: the *Balance Accuracy* (BACC). The BACC normalizes true positive and true negative predictions by the number of positive and negative samples, respectively, and divides their sum by two.

$$\text{Balanced Accuracy (BACC)} = \frac{\text{TPR} + \text{TNR}}{2} \tag{3}$$

where TPR = TP/P is the true positive rate (or recall), TNR = TN/R is the true negative rate.
Other two useful metrics for the performance evaluation are precision, which indicate how accurate the model predict the hand (positive) points, and recall (or true positive rate) which indicates how many positive points the model correctly detected with respect the total real positive ones.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{4}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} = \frac{\text{TP}}{\text{P}} = \text{TPR} \tag{5}$$

The last metric is the F1 score, or Dice Coefficient, which combines precision and recall. It is the harmonic mean of precision and recall

$$\text{F1} = 2 \times \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{6}$$

Geometrically it represents two times the area of overlap divided by the total number of pixels in both the detected and real masks. F1 is very similar to the Intersect over Union quantity.

Pixel Accuracy evaluation is done with the method `Evaluation::PixelAccuracy`. It returns a vector with two images: the first one is the original image with the detected and segmented hands (Fig. 11a), the second one is an image that represent the confusion matrix (Fig. 11b). The results of all the metrics we considered are printed on console.

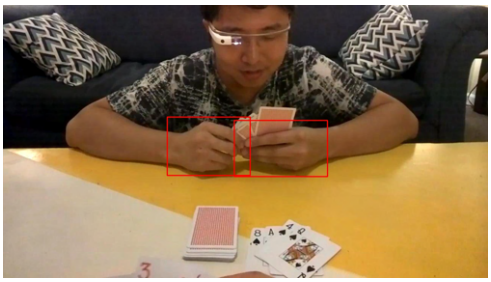
(a) 04.jpg accuracy image

(b) 04.jpg confusion matrix image

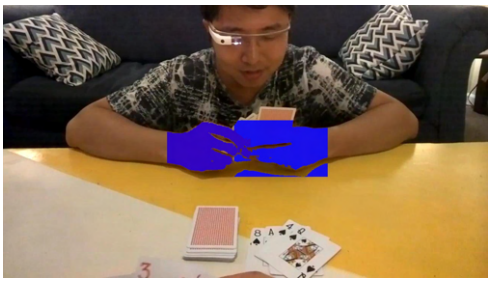
Figure 11: Output of the method Evaluation::PixelAccuracy

3 Results

The following images show the results obtained for detection and segmentation with 30 test images

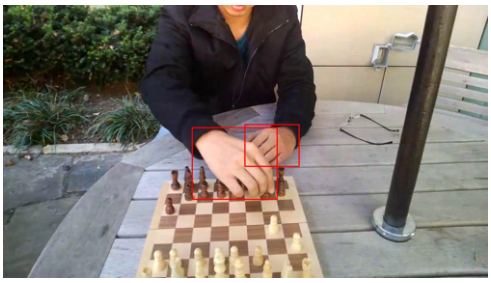


(a) 01.jpg detection

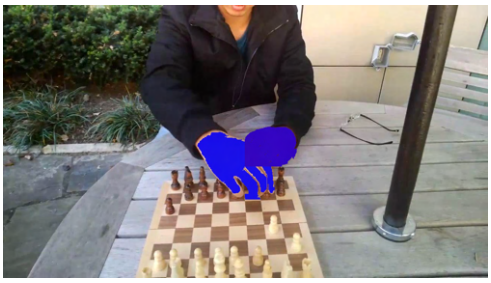


(b) 01.jpg segmentation

Figure 12: Detection and segmentation results with 01.jpg

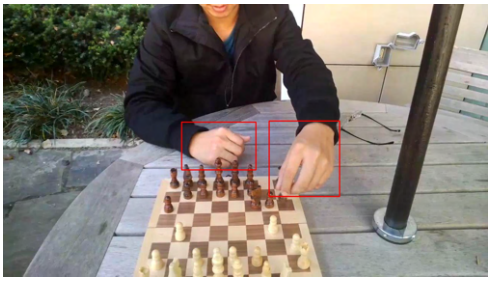


(a) 02.jpg detection



(b) 02.jpg segmentation

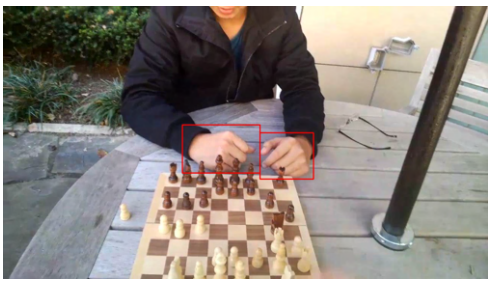
Figure 13: Detection and segmentation results with 02.jpg



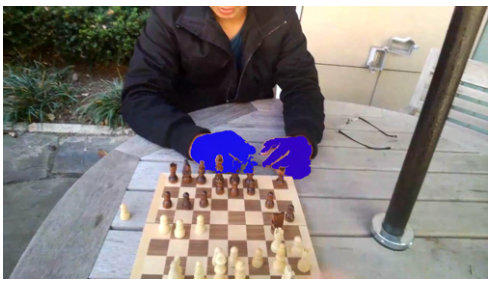
(a) 03.jpg detection

(b) 03.jpg segmentation

Figure 14: Detection and segmentation results with 03.jpg

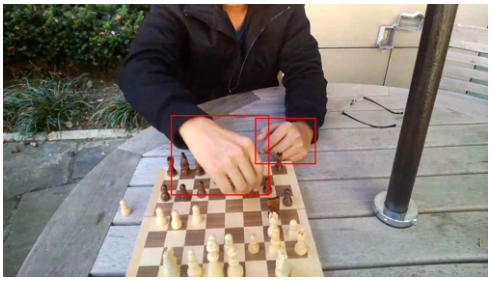


(a) 04.jpg detection



(b) 04.jpg segmentation

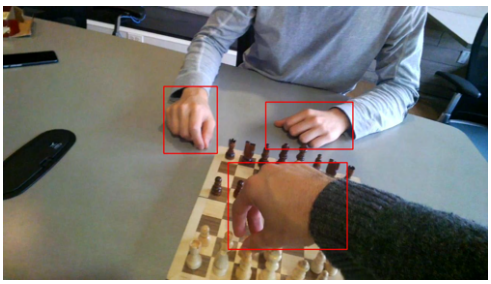
Figure 15: Detection and segmentation results with 04.jpg



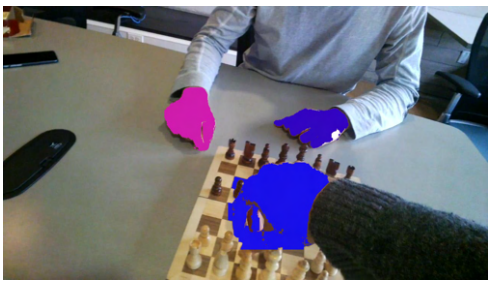
(a) 05.jpg detection

(b) 05.jpg segmentation

Figure 16: Detection and segmentation results with 05.jpg

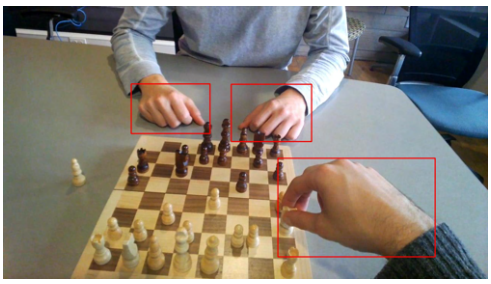


(a) 06.jpg detection

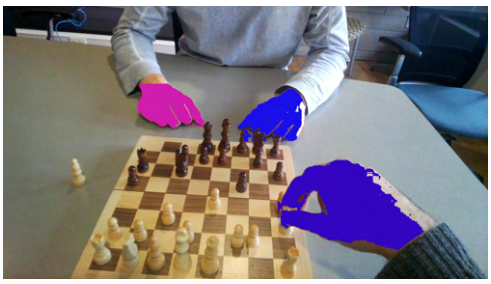


(b) 06.jpg segmentation

Figure 17: Detection and segmentation results with 06.jpg

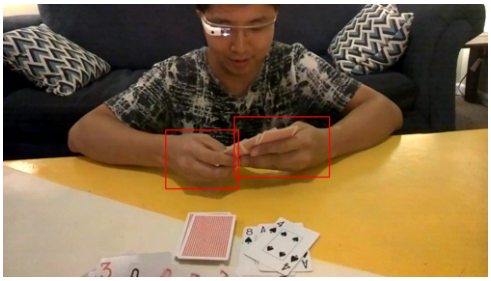


(a) 07.jpg detection

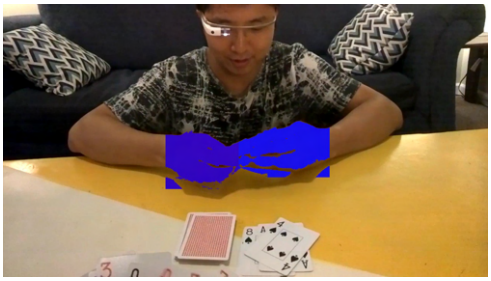


(b) 07.jpg segmentation

Figure 18: Detection and segmentation results with 07.jpg

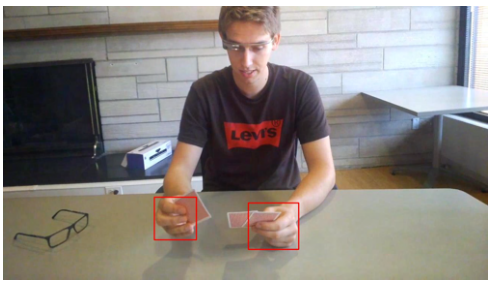


(a) 08.jpg detection

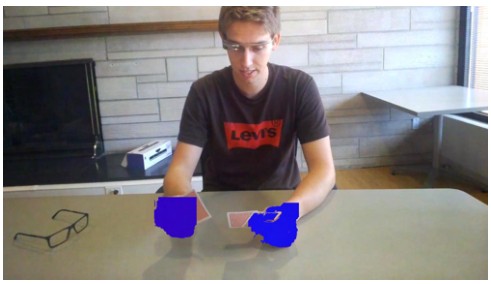


(b) 08.jpg segmentation

Figure 19: Detection and segmentation results with 08.jpg

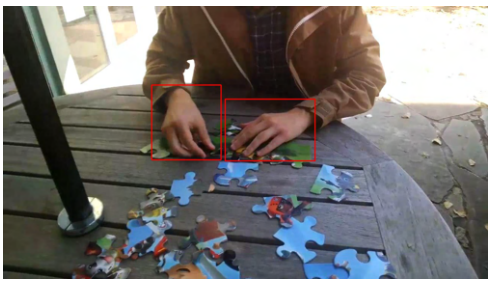


(a) 09.jpg detection



(b) 09.jpg segmentation

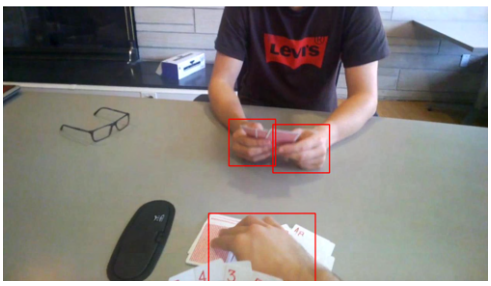
Figure 20: Detection and segmentation results with 09.jpg



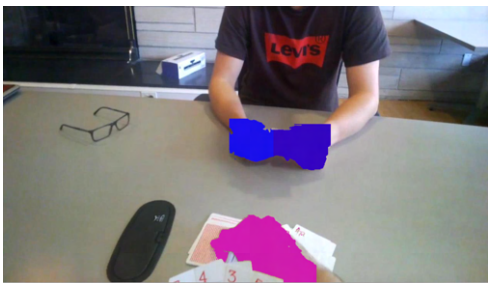
(a) 10.jpg detection

(b) 10.jpg segmentation

Figure 21: Detection and segmentation results with 10.jpg

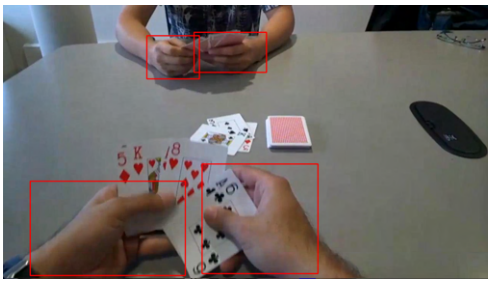


(a) 11.jpg detection

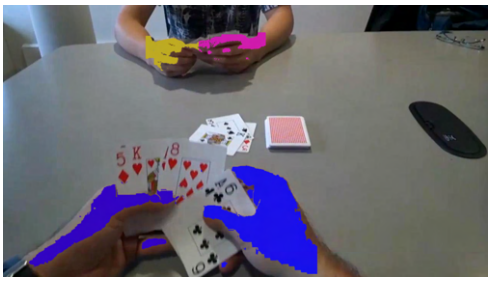


(b) 11.jpg segmentation

Figure 22: Detection and segmentation results with 11.jpg

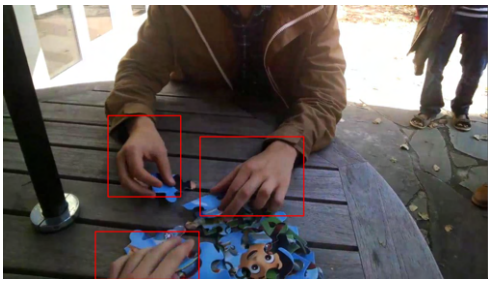


(a) 12.jpg detection



(b) 12.jpg segmentation

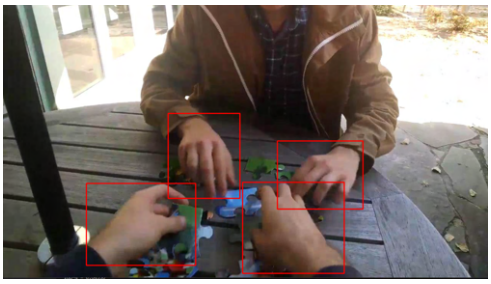
Figure 23: Detection and segmentation results with 12.jpg



(a) 13.jpg detection

(b) 13.jpg segmentation

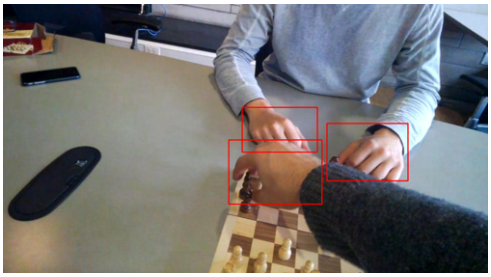
Figure 24: Detection and segmentation results with 13.jpg



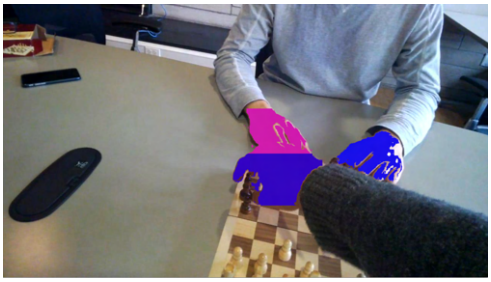
(a) 14.jpg detection

(b) 14.jpg segmentation

Figure 25: Detection and segmentation results with 14.jpg

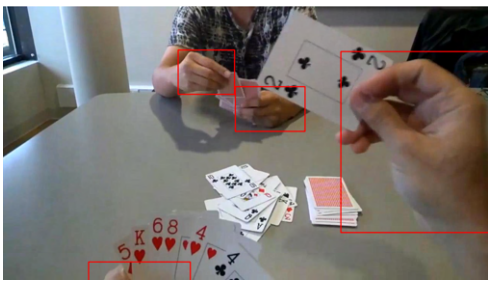


(a) 15.jpg detection



(b) 15.jpg segmentation

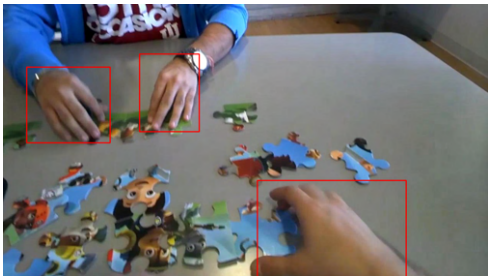
Figure 26: Detection and segmentation results with 15.jpg



(a) 16.jpg detection

(b) 16.jpg segmentation

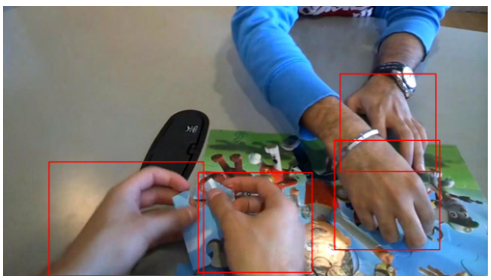
Figure 27: Detection and segmentation results with 16.jpg



(a) 17.jpg detection

(b) 17.jpg segmentation

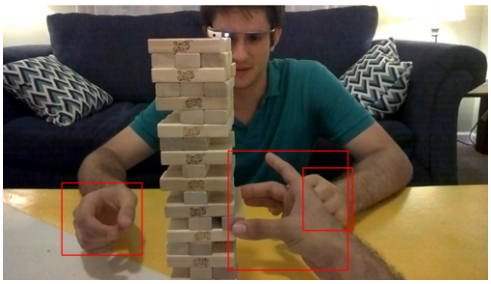
Figure 28: Detection and segmentation results with 17.jpg



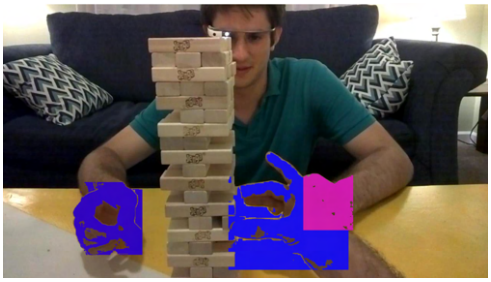
(a) 18.jpg detection

(b) 18.jpg segmentation

Figure 29: Detection and segmentation results with 18.jpg

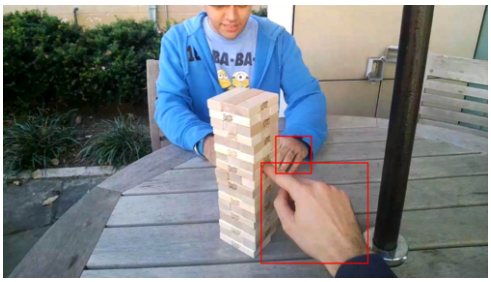


(a) 19.jpg detection



(b) 19.jpg segmentation

Figure 30: Detection and segmentation results with 19.jpg



(a) 20.jpg detection

(b) 20.jpg segmentation

Figure 31: Detection and segmentation results with 20.jpg

(a) 21.jpg detection

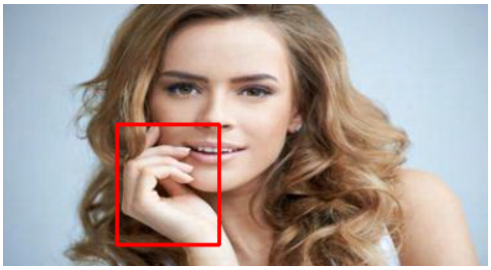
(b) 21.jpg segmentation

Figure 32: Detection and segmentation results with 21.jpg

(a) 22.jpg detection

(b) 22.jpg segmentation

Figure 33: Detection and segmentation results with 22.jpg



(a) 23.jpg detection

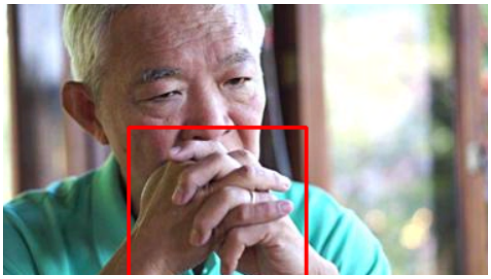
(b) 23.jpg segmentation

Figure 34: Detection and segmentation results with 23.jpg

(a) 24.jpg detection

(b) 24.jpg segmentation

Figure 35: Detection and segmentation results with 24.jpg



(a) 25.jpg detection

(b) 25.jpg segmentation

Figure 36: Detection and segmentation results with 25.jpg

(a) 26.jpg detection

(b) 26.jpg segmentation

Figure 37: Detection and segmentation results with 26.jpg

(a) 27.jpg detection

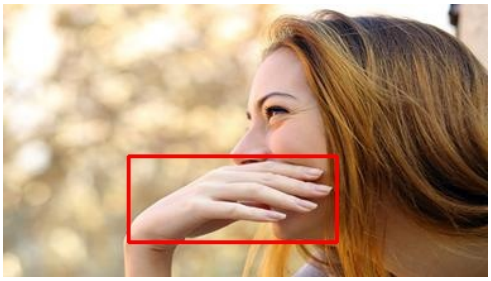
(b) 27.jpg segmentation

Figure 38: Detection and segmentation results with 27.jpg

(a) 28.jpg detection

(b) 28.jpg segmentation

Figure 39: Detection and segmentation results with 28.jpg



(a) 29.jpg detection

(b) 29.jpg segmentation

Figure 40: Detection and segmentation results with 29.jpg

(a) 30.jpg detection

(b) 30.jpg segmentation

Figure 41: Detection and segmentation results with 30.jpg

3.1 Results and metrics

In Table 2, the results in terms of IoU are presented

Image	IoU-1	IoU-2	IoU-3	IoU-4
1	0.689175	0.720806	-	-
2	0.781495	0.819071	-	-
3	0.656013	0.758018	-	-
4	0.693218	0.838766	-	-
5	0.700444	0.802049	-	-
6	0.762316	0.723008	0.591475	-
7	0.630144	0.629925	0.917587	-
8	0.839451	0.768313	-	-
9	0.738806	0.617682	-	-
10	0.674044	0.639736	-	-
11	0.877006	0.805155	0.679896	-
12	0.868613	0.839928	0.729317	0.695113
13	0.727053	0.769377	0.716243	-
14	0.774873	0.727303	0.700653	0.643560
15	0.690394	0.655005	0.836223	-
16	0.693568	0.876231	0.762616	0.226051
17	0.764915	0.67926	0.845047	-
18	0.791102	0.898142	0.838801	0.77938
19	0.860603	0.719191	0.750811	-
20	0	0.799498	0.924984	-
21	0.797927	-	-	-
22	0.849505	0.56	-	-
23	0.873306	-	-	-
24	0.707405	-	-	-
25	0.626943	0	-	-
26	0.923479	-	-	-
27	0.756201	0.714664	-	-
28	0.882870	-	-	-
29	0.765596	-	-	-
30	0.882647	-	-	-

Table 2: Test images IoU results

Only 5 IoU values are lower than 0.6 and two zeros are obtained, namely the hand detector is not able to detect two hands. Moreover, for image 16.jpg the IoU is extremely low. An explanation follows:

- As shown in Figure 42, it makes sense that the hand detector is not able to recognize the hidden hand.

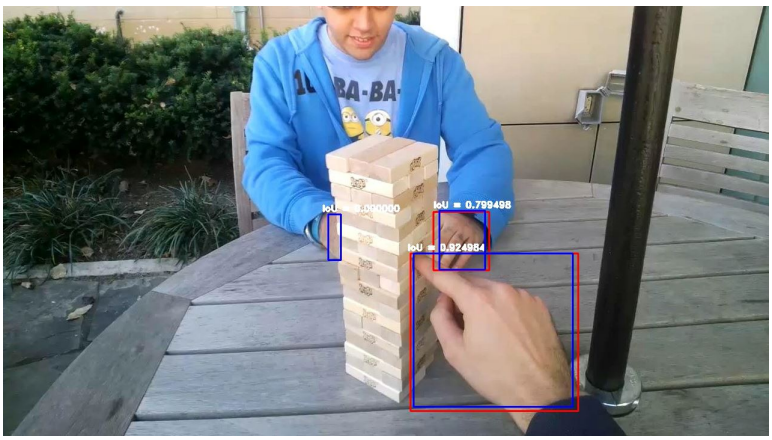


Figure 42: Results for image 20.jpg

- About the low value obtained in image 16.jpg, a possible explanation might be: as shown in Figure 43, only a small piece of an hand is present (the one in the bottom, for which a low IoU is obtained). The fact that there is a very small part of an hand is relevant since Yolov4 exploits the so called anchor boxes or prior boxes (as Yolov2, Yolov3 and other object detectors based on CNN). These priors are also defined based on the training set, since Yolov4 uses k-means to generate those anchor boxes. This means that if the training set doesn't contain many examples with very small parts of hand (in our case the dataset doesn't have so many examples of this kind), the resulting bounding boxes will be larger. The training dataset doesn't effect only the anchor boxes, but the regression itself.

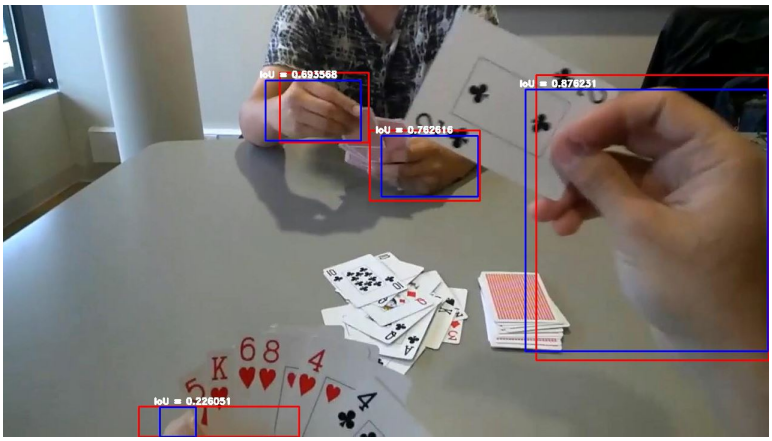


Figure 43: Results for image 16.jpg

- The zero obtained in image 25.jpg is the result of the Non-Maximum Suppression performed in post-processing (in the c++ code, look at the function `HandDetector::process_results`). In fact by increasing the threshold (the one for this purpose), more bounding boxes appear. The problem is that they are overlapping and this, in general, has to be avoided. In Figure 44 is shown what happens if the threshold is too high

Figure 44: Results for image 25.jpg with `nms_th = 0.9f`

In Table 3, the results in terms of pixel accuracy, precision, recall, balanced-accurancy and F1 are presented

Image	accuracy	precision	recall	balanced-accuracy	F1
1	0.97981	0.708374	0.851364	0.918294	0.773315
2	0.995437	0.93799	0.923542	0.960724	0.93071
3	0.993327	0.886818	0.917804	0.956873	0.902045
4	0.994321	0.860256	0.914884	0.955592	0.886729
5	0.989911	0.825493	0.948826	0.970226	0.882873
6	0.984134	0.853355	0.92631	0.957336	0.888337
7	0.978332	0.918188	0.846728	0.919416	0.881012
8	0.983047	0.769069	0.870897	0.929516	0.816822
9	0.990728	0.628638	0.957586	0.974414	0.759004
10	0.988984	0.909319	0.723514	0.86058	0.805845
11	0.984114	0.764234	0.947349	0.966619	0.845996
12	0.925475	0.836867	0.493103	0.739774	0.620558
13	0.972105	0.936561	0.610773	0.803954	0.73937
14	0.96181	0.858049	0.729412	0.85818	0.788519
15	0.987945	0.844227	0.902151	0.947098	0.872228
16	0.918039	0.855968	0.555654	0.769422	0.673866
17	0.978121	0.938663	0.813867	0.904258	0.871822
18	0.916884	0.822176	0.707661	0.836273	0.760632
19	0.947786	0.601925	0.803259	0.881106	0.688168
20	0.988052	0.892362	0.87762	0.935886	0.884929
21	0.953535	0.656645	0.402851	0.694997	0.499351
22	0.988124	0.838286	0.851334	0.922428	0.84476
23	0.958647	0.606264	0.666145	0.820737	0.634796
24	0.97139	0.917634	0.427394	0.712755	0.583172
25	0.933642	0.852296	0.609288	0.796495	0.71059
26	0.953668	0.994067	0.679422	0.839374	0.807165
27	0.972958	0.913339	0.68388	0.839461	0.782127
28	0.985303	0.911914	0.911253	0.951636	0.911583
29	0.977696	0.990518	0.731417	0.8654	0.841474
30	0.988872	0.888115	0.769258	0.882926	0.824425

Table 3: Test images results

This table shows how misleading the accuracy results can be. The prime example is the image *21.jpg*: in this image we have an accuracy of 95% but the balanced accuracy is 69% and the F1 score is about 50%. To make a comparison, the figure *26.jpg* has very similar result on the accuracy metric, but way better results on the balanced accuracy and on F1. Figure 45a shows the result of

the segmentation for *21.jpg*, Figure 45b shows the result for *26.jpg*: the results are very poorly for the first image, instead in the second one we have a pretty accurate segmentation.

(a) 21.jpg (b) 26.jpg

Figure 45: Comparison of the segmented images

For this reason we consider only the last two columns of Table 3 to evaluate the results. The average balanced accuracy is 0.87905, and in particular only 5 images have values under the 80% (only 1 under 70%). The average F1 score is 0.76220, and only 6 images have results under the 70% (2 of them have F1$< 60\%$).

A Project development

- **Simone Cecchinato:** selecting and labeling the training set (with Riccardo and Mattia's help), `Evaluation` class, `Segmentation::is_Greyscale`, `Segmentation::final_mask`, `hand_detection_segmentation.cpp`. Time spent ≃30h.
- **Riccardo De Monte:** training of Yolo (with Simone's help), `HandDetector` class, `HandSegmentor::get_expanded_roi`, `HandSegmentor::get_mask_original_size`, `HandSegmentor::get_masks_per_region`, `HandSegmentor::get_biggest_region` and `HandSegmentor::count_nozero_pixels`. Time spent ≃35h.
- **Mattia Secchiero:** development of the program structure, cmake files and implementation of the `HandSegmentor` class (with Riccardo's help) and `skin_segmentation` library. Time spent ≃30h.

B Other trials

For detection, another approach is tried. The idea consists in using Selective Search (the fast version, that proposes about 2000 boxes), use a classifier to classify each region and then process the results (Non-maxima suppression based on the confidence levels and the IoU). Two different approaches for the classification problem have been taken into account:

- train a binary classifier to distinguish hands from any possible other object. As binary classifier, Bag of Features (Bag of Words) has been implemented using SIFT, k-means, SVM (with RBF kernel) and tf-idf weighting factor. By this link , you can download a program for cats and dogs classification and by this link a dataset (position the directory $images$ in the BoF one). You have to pass the name of the image as argument for the program e.g. cat.10000.jpg. The BoF classifier has been trained with 4000 images (perfectly balanced. The images used are the first 2000 dogs images and the first 2000 cats images. For this reason, to test the classifier take into account the rest ones), with $K = 1500$ ($\#$ clusters used with k-means, namely the number of visual words), $\gamma = 10$ and $C = 10$ (SVM hyper-parameters) and the results are: 87% for training-score and 75% for test-score (with 8000 images). The problem is that it is difficult to have a suitable dataset with many negative examples, namely many no-hand objects. For this reason this approach has not been taken into account in our project.
- train a One-Class classifier considering as "normal" data the hands images. For features extraction the VGG16 CNN has been used (only the backbone) while for the OCC problem SVM has been used. In particular the following python modules are used: Keras (for the CNN and for data augmentation), Scikit-learn (for OneClassSVM) and OpenCV. The challenging part has been the hyper-parameters tuning and the best result obtained with 2000 positive example (only hands) is only 60% of correct classifications (414 hands images have been used for the training, but also data augmentation has been used).

Finally, this approach has the drawback of speed: evaluating 2000 possible regions means too much time spent and for this reason we have chosen Yolo.