CiComputer Vision Project

Group members: Boldrin Massimo
Da Re Leonardo

July 2022

Human hands detection and segmentation

Github repository: https://github.com/afParadox/ComputerVisionProject2022.git

1 Hand detection

The objective of this project is to produce a program that is able to detect and segment human hands in pictures. The pictures can be taken from any angle with respect to the hand, so, as in the case of most object localization algorithms, we need a method that is able to recognize hand in any possible position and shape. We also have to take into consideration the fact that hands, unlike most other objects, can change quite a bit depending on what the subject of the photo is doing. Taking all of this into account, it was immediate for us the choice of using deep learning.

1.1 Exploring the options

After some research the first models that we considered were R-CNN, Fast R-CNN and Faster R-CNN, which stand for *Regions with CNN features*, a "family" of methods for object localization, each one evolved from the previous one. When given an image in input, the network would try to figure out the so called "region proposals", which are regions in the image where it would then look for objects, once extracted another network would classify them. In the case of R-CNN the number of region proposals is constant, always 2000, which is clearly not ideal, since generally is unlikely for 2000 objects to be present in a single image, but it would also fail in the unfortunate case of having more objects than those.

On its successors the approach it's improved, since the number of region proposals it's deducted from the image directly, but the main problem is that, while they are well functioning, these methods were still not fast as we would have liked.

1.2 YOLO

The YOLO algorithm (You Only Look Once) is the algorithm that we finally chose to use. Its fifth version, YOLOv5 (https://github.com/ultralytics/yolov5) is the state of the art object localization algorithm right now. The model predicts classes and bounding boxes in one single run: first the image is split into an SxS grid (commonly S = 19), each cell produced is then responsible for the bounding box prediction.

Figure 1: division of the image in a grid, and assignation of the cells

The probability of the cell to contain a certain class in computed, and the cell is then assing to the highest probability one.

Based on the probabilities the algorithm just computed, a certain number of bounding boxes is generated, but most of them aren't correct. To get rid of the incorrect ones, YOLO performs *Non-max suppression*, a step where the value of Intersection over Union is calculated for each of the bounding boxes, and the ones that got a value over a certain threshold get eliminated.

Figure 2: non-max suppression

Finally, for every object that found, algorithm outputs a vector containing: a number that indicates the class, the coordinates of the bounding box, and its width and height.

The algorithm is licenced under the GNU General Public Licence v3.0, so we can use it freely. So once found the model it was left to train it.

1.3 Dataset

Since we decided to work with deep learning, we had to create a good enough dataset that would allow us to train successfully the network. For this we considered creating a dataset ourselves, but

ultimately decided to merge three different datasets, since the manual construction of a viable dataset would have taken a significant amount of time.

The datasets that we picked are “HandsOverFace” and “EgoHands”, both provided in the assignment of the project, to which we added the Hands dataset from the University of Oxford. To avoid confusion with the other datasets, in this report we’ll refer to this last dataset as the *Oxford dataset*. Since the evaluation dataset for the project got provided to us alongside the two datasets mentioned above, we could effectively eliminate from the dataset all the images that will be used to evaluate this project. In the case of EgoHands we took it a step further and eliminated directly the whole subfolder containing an image that will be used in the evaluation, due to the big similarities among photos in the same folder.

HandsOverFace contains 300 images, while we used 288 of them. EgoHands contains 4800 images, while we used 3900 of them. Considering that the Oxford dataset contains 5628 more photos, we used a total of 9816 images for the final dataset.

1.4 Python Notebook

For the training of the network we decided to use a python notebook, given the advantages that running single cells of the code can offer. The notebook is divided in five sections:

1) Management of dependencies and YOLOv5: In the first part of the notebook we are just importing all the libraries that we need, like os to manipulate the dataset folders, or pytorch. Also, we are cloning the YOLOv5 repository from github, and installing its requirements, found in the *requirements.txt* file.

2) Download and extraction of the dataset: The second part of the notebook it’s dedicated to the download of the three datasets. We get the Oxford and the EgoHands dataset from their direct source, but we import the HandsOverFace dataset from a temporary repository created for this purpose, since the compressed file downloaded from the provided link had problems unzipping in the notebook. Alongside with the dataset, the repository *dataset_CV* also contains all the annotations for the three datasets alredy converted in the YOLOv5 notation. The Oxford and the EgoHands dataset come with annotations in *.mat* files, while HandsOverFace’s annotations come in *.xml* files, so to ease the training process we converted them all in *.txt* with the correct format accepted by YOLO one time and just used them. In this part we also are obbligated to rename many of the samples we are using due to the use of repeated names. In the case of the EgoHands dataset we renamed all the samples, while for the Oxford dataset we just had to rename the training subdirectory.

In this section we also reorganize all the samples in the three main sub-divisions: training, validation and test. For validation we picked 1000 random samples from the total, and for test we picked another 800 from the remaining ones.

3) Check if the annotations and photos coincide: Due to all the converting and renaming of the samples, we use this section to check that all the annotations still coincide with the correct image. We used to have a cell that printed around 1000 samples with their respective bounding boxes, but for the upload of this project we removed it, since it was a mere control. However we left another cell that picks a random image from training, validation and test, which can be useful to show that the conversions are actually precise.

4) Preparation for training: Before training we have to reorganize all the dataset in a structure that is comprehensible to YOLO, and tell it where it can find the dataset.

5) Training: Finally we run the training. In the YOLO training we have control over a certain amount of hyperparameters, like the dimension of the image we are going to work with (yolo works with squared images, so it resized the image to a square) , the “version” of the network (small, medium, large, extra-large), the number of epochs, etc. Due to the problems we encountered on training the model using Google Colab, we moved to another platform, Paperspace Gradient, where we could let the training run up to six hours.

The final model that we chose to use uses the medium sized version of YOLOv5, with an image size of 640 (we would have liked to use 1280 as image size but the computation was way more heavier), batch size 20 (again, limited by the ram available), and 60 epochs. The training took around 3/3.5 hours, running on an RTX A5000.We include in the next page an example of the results we obtained training with different configurations.

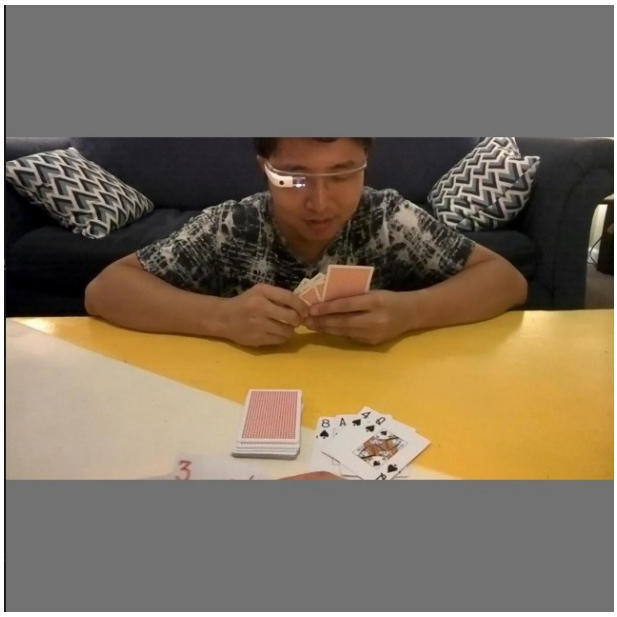


Figure 3: Example of resized image. As we can see the image does not get cropped, but instead is "padded" with gray bands

Final notes on the model:

The model has been trained on the python notebook, but it’s important to point out that it has been completely exported into a *.onnx* file, which can then be read and called from c++. The python notebook and none of the code present in the support directory are necessary for the execution of the code, but were used to produce the final product. Due to the fact that yolo needs squared pictures in input, we also implemented the appropriates methods to convert the input images and rescale the bounding boxes coordinates.

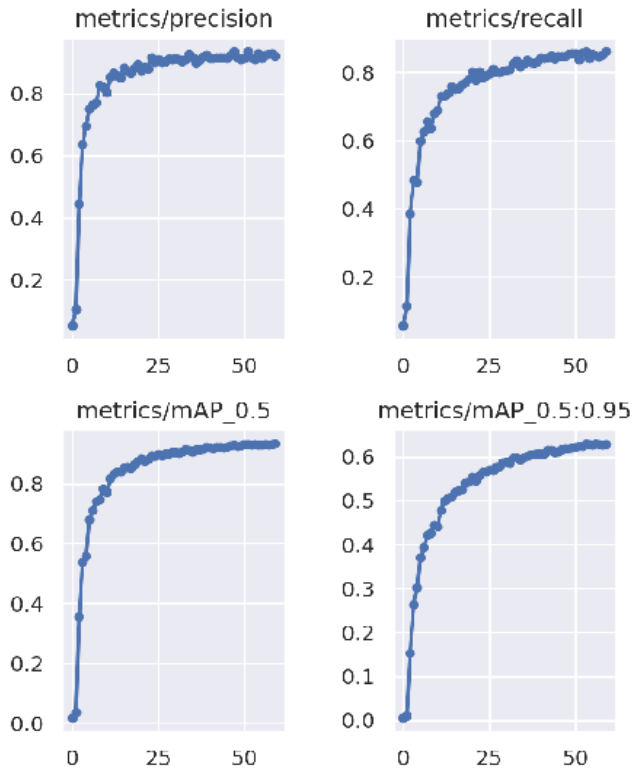


Figure 4: Medium model, image size 640, batch 20, epochs 60 (~3 hours)

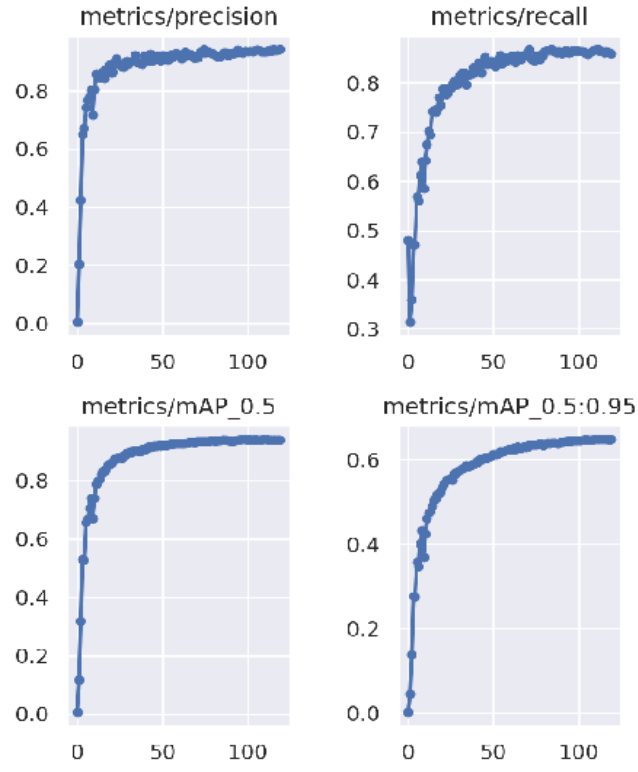
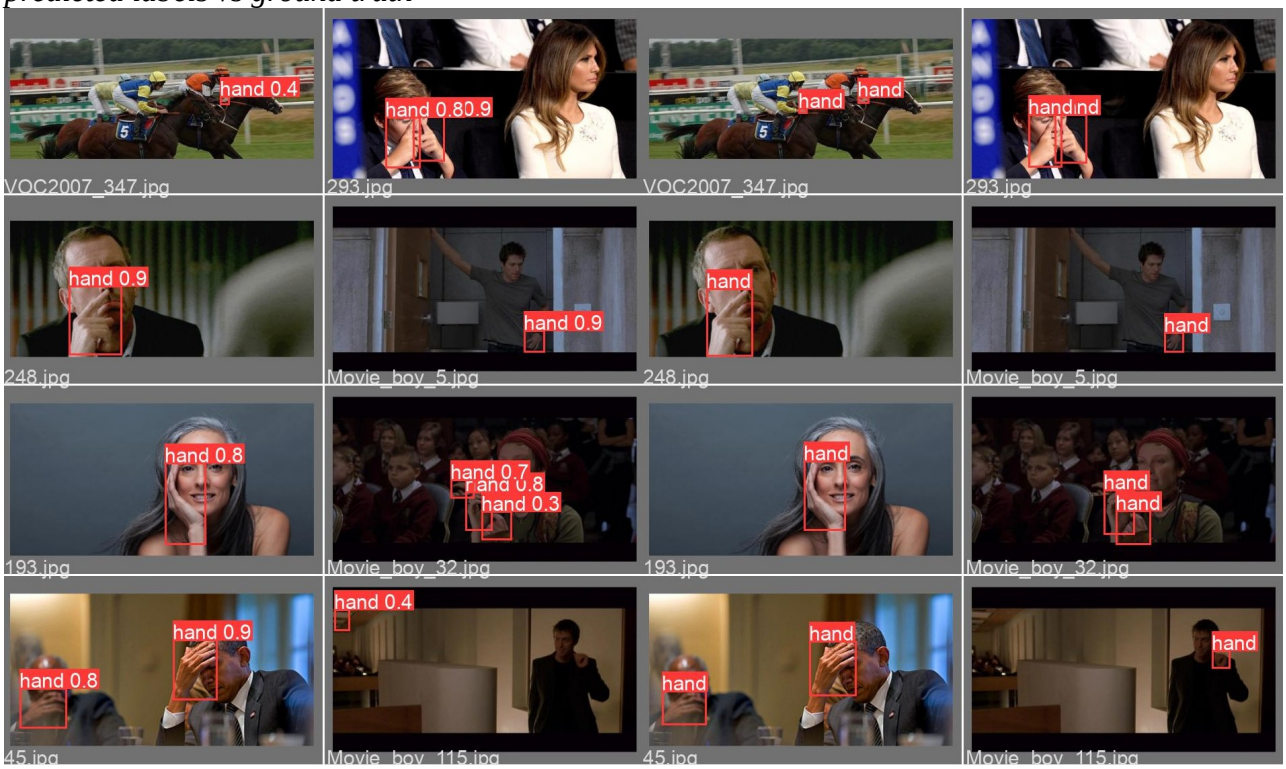


Figure 5: Large model, image size 512, batch 13, epochs 120 (~4.5 hours)

Figure 6: Comparison between in-training predicted labels vs ground truth



2 Hand Segmentation

The idea with hand segmentation was to isolate every single hand instance and segment it singularly, which we can do easily since we are computing their position in the detection step. Once selected the are where we want to segment the hand, we apply watershed on properly selected markers.

2.1 Isolation of hand instance

The first step is to decide in which region of the image we are going to apply segmentation. Since we are also computing the bounding box for every hand in the image it is logic to apply the algorithm only to that region instead that to the whole picture. At first we used a weak model that selected an area slightly too large for the hand, and the segmentation algorithm that we designed was made considering those bigger bounding boxes. Since we replaced that model with a better one that gives tighter regions we ended up enlarging the boxes slightly before passing them to the algorightm.

Figure 7: Cutted-out hand istance

2.2 Preprocessing

For the preprocessing we considered different options:

1) Bilateral filter: Due to the caracteristics of the bilateral filter of smoothing the image while keeping the sharp edges, we tried to apply it before the segmentation to improve its precision, however once tested we saw that the results were worse than not using it so we commented it.

2) Gaussian sharpening: We also tried to sharpen the image to try to make more obvious to the algorithm where the edges of the image were, for this we adapted a function found on the Internet to our use case, but since it did not seem to improve the precision we decided not to use it.

3) Canny: Finally we decided to use the canny edge detector to extract the edges of the image, just to draw them on top of the original image in black, to highlight the interruptions between an area of the photo to the other.

Figure 8: hand istance after preprocessing with Canny overlay

2.3 Watershed algorithm

To implement watershed segmentation in opencv it's required to set the *starter* markers. In the case of this project we need to set two different set of markers: one for the "hand region" and another one for the "background" (in this case anything that is not a hand).

Our first attempt for the markers was to select the central pixels of the image for the hand region, since we are giving as input the image cropped around the bounding box so in the center there will always be a hand. For this then we selected the pixels inside a rectangle 1/10 of the size of the original bounding box, located in the middle. Meanwhile, for the background region markers we selected all the points along the edges of the bounding box, taking into account that we are passing boxes slightly bigger than the actual hand. This approach wasn't very effective due to the fact that the algorithm would often get confused when other parts of skin were in the picture since those pixels are (aproximately) the same color as the hand, and ended up segmenting only a small portion of it.

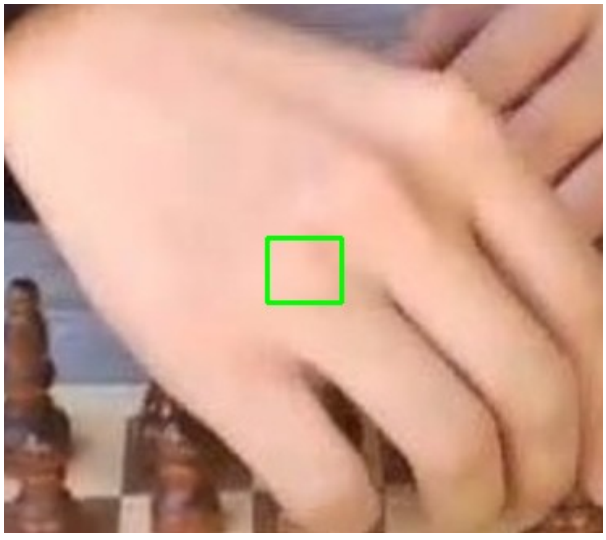


Figure 9: Central part, hand-markers example

In attempt to improve this scenario we changed the strategy used to select the background markers, this time instead of picking *all* the pixels along the bounding box border, we selected only the ones not similar enough to the pixels of the hand. Basically, we computed the average and standard deviation of the pixels in the center, and used those values as *skin-color* and *threshold*, so that if a pixel in the border was within a the threshold, then that pixel wouldn't be considered for the background markers. While this approach improved the segmentation over the first, it still presented problems, since sometimes shades present in the center part of the image (like in between fingers) would mess with the average value. The code of this attempt has been commented.

Finally, we decided to select the backgroud markers using this strategy, but we changed the hand markers selection process. First we segmented the hand instance using graph segmentation, with parameters tuned using trackbars and a dedicated script (which is found in the support directory). Then we choose as markers all of the regions produced by graph segmentation that would appear to be included (even if for just one pixel) in the central rectangle that we previously used in the first attempt. Thanks to this we were able to select way more pixels as the initial hand markers, and this approach is the one that produced the best results over the others.

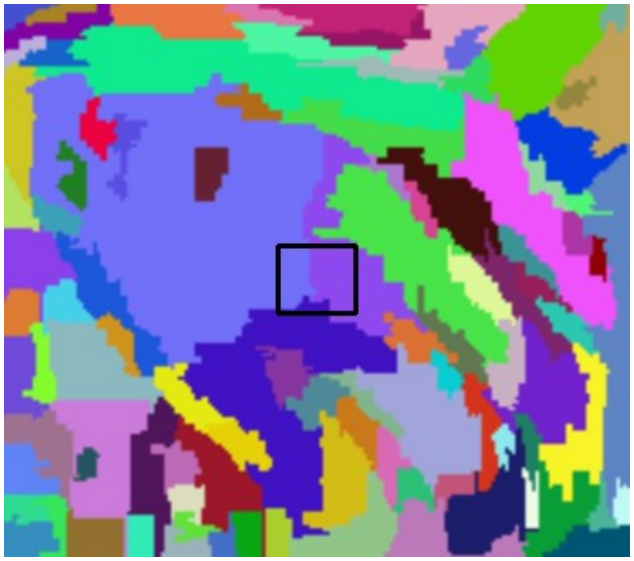


Figure 11: Result of Graph (over)Segmentation

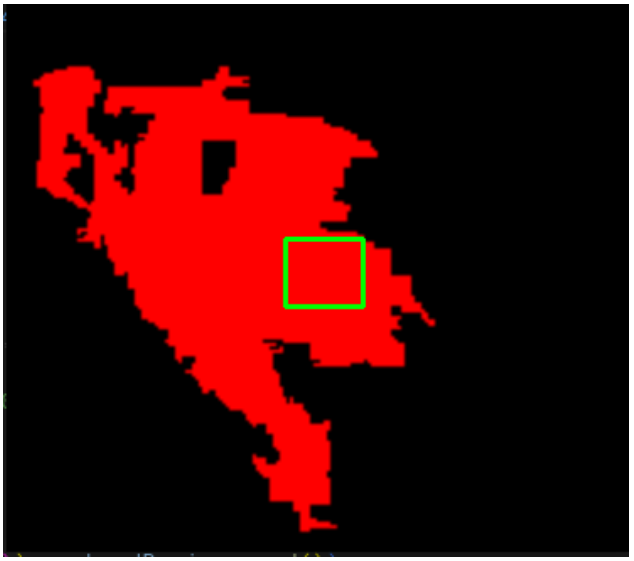


Figure 10: Selected Hand Markers

After this steps, we applied the opencv watershed algorithm implementation.

3 Possible improvements

We wish to conclude this report presenting possible ways to improve the segmentation:

1) An improvement of the hand markers selection, working deeper with the graph segmentation algorithm, finding the average color of every region and (as in a clustering sort of problem) unite them in order to better select regions that may not be present in the central area.

2) Changing the color used for the canny edges may improve the selection.

3) Using a region growing algorithm instead of watershed could benefit more from the canny edges preprocessing.

In respect of the hand detection we could:

1) Enlarge the dataset. The dataset that we built is quite large, but very unbalanced, the HandsOverFace dataset is very small compared to the others two, which means that the model fits way better images more similar to the bigger datasets, while the HandsOverFace samples, which are a bit darker and with more shades, get worst detections.

2) Besides the tuning of hyperparameters, we could also be looking for an object detection algorithm specialized for the detection of a single class over an image.

4 Division of the work

The project took around three weeks to complete, for a total of aproximately 150+ hours of work.

As for the general focus of the work, the student Leonardo Da Re worked mostly on the detection side of the project, while Massimo Boldrin took over the segmentation part.

Using YOLOv5 was an idea of Leonardo, as well as the construction of the dataset as a whole, while Massimo came up with the segmentation process using watershed and graph segmentation, but most of the decisions of the path we wanted to follow in the development of this project were taken in Discord calls, mostly brainstorming and coming up with ideas.

References

- Figures 1, 2: https://towardsdatascience.com/yolo-you-only-look-once-3dbdbb608ec4#:~:text=YOLO%20algorithm%20is%20an%20algorithm,what%20is%20actually%20being%20predicted
- HandsOverFace dataset: https://drive.google.com/file/d/1hHUvINGICvOGcaDgA5zMbzAIUv7ewDd3
- EgoHands dataset: http://vision.soic.indiana.edu/projects/egohands/
- Oxford hand dataset: https://www.robots.ox.ac.uk/~vgg/data/hands/