Computer Vision - Project Report

Roberto Del Ben - Youssef Ben Khalifa

July 29, 2022

Contents

1 Introduction

The goal of this project is to develop a software for the detection and segmentation of hands. The software is divided in two main modules:

- **Hand detection module**: this module detects the hands inside a given image and draws the corresponding ROI (Bounding Boxes);
- **Hand segmentation module**: this was developed to segment those same hands detected by the previous model.

The program is developed using both C++ and Python, the first one is used to process the images using the OpenCV library, extract the dataset and perform the segmentation, whilst the Python part has the sole purpose of performing the detection using a CNN specifically designed for object detection.

The Hand detection module is based on the use of the **YOLO**(You Only Look Once) CNN model, which is considered amongst the most efficient object detection algorithms along with the R-CNN and Fast R-CNN architectures. The goal is to take a known CNN architecture for the YOLO model and train it using the dataset that we will be extracted using a class that will be implemented into the project.

2 Image preprocessing [2 hrs]

Roberto del Ben, Youssef Ben khalifa

The first thing we need in the entire project is to define a pre-processing pattern for every image. The aim of pre-processing is to transform the image into a form that is easier to process, that is, we want to remove as much noise as possible without losing the important information, while also enhancing the main features and improving the general visual representation of the image. This is quite important, as the processes we will use to perform both the detection and the segmentation can be very sensible to additional noise and they are also based on some features of the image that in some cases may not be noticeable from the raw image.

For each image we will then apply the following:

- **Bilateral/Gaussian filter**: This filter is a very effective way to remove noise from the image;
- **Derivative Filter**: Using an appropriate kernel, we apply a spatial filter to the image to enhance small details and edges;
- **Histogram Equalization**: This filter is a very effective way to enhance the contrast of the image;
- **Gamma Correction**: It is used to enhance the lighting situation in the image, in order to remove dark and light areas that hide important features of the image,;

3 Dataset creation and extraction [9 hrs]

Youssef Ben khalifa

We based the project on the EgoHands dataset, which is the same one used for the benchmarking of the hand detection and segmentation algorithms. The dataset is available at the following link:

http://vision.soic.indiana.edu/projects/egohands/

since the goal is use a CNN to detect the hands, and as mentioned the benchmarking will be done partly on the same dataset, to avoid overfitting we used, in addition to the EgoHands dataset, the following set of images:

https://www.kaggle.com/datasets/shyambhu/hands-and-palm-images-dataset

The EgoHands dataset was provided with the source RGB images along with the respective bounding boxes labels as a *csv* file and the masks for the segmentation. We then processed the dataset in order to extract the single hand frames from each image, both segmented and not, for them to be then used for the training of the CNN and eventually the evaluation for the entire project.

3.1 Dataset definition

To start off, we defined the structure of our dataset in the c++ module, which can be found the *dataset.h* file: in particular we defined the *HandMetadata* and *Image* as follows:

```
    class HandMetadata
{
public:
    int PosX;
    int PosY;
    int Width;
    int Height;
    Rect BoundingBox;
    bool isNull;
    bool isInvalid;

    HandMetadata();
    void initBoundingBox();
};

class Image
{
public:
    int id;
    string jpgFileName;
    string maskFileName;
    string csvFileName;
    Mat src;
    Mat mask;
    Mat cannySrc;
    vector<int> coords;
    vector<HandMetadata> handsMetadata;
    vector<Mat> handSrc;
    vector<Mat> handMasks;
    vector<Mat> handFinals;
    vector<Mat> nothands;
    vector<Mat> nothandsCanny;
    vector<Mat> handCanny;

    Image(Mat, Mat, vector<int>);
    void loadCoordinates();
    void cutImage();
    void applyCanny();
    void segmentImage();
    void preProcessImages();
    void cutBackground();
};
```

these classed are the ones used throughout the project to help us keep track of the images and the hands we detect within those same images dataset. The *HandMetadata* object is in charge of holding all the needed information about a single detected hand, specifying the boundingBox coordinates through the attributes ($PosX, PosY, Width, Height$) and some flags that will determine whether the image is read correctly.

The Image class instead holds information about the source image, keeping track of RGB source image, the mask and the csv associated file containing the box information; more importantly in the class is defined a *vector¡HandMetadata¿* which is the one attribute that memorize the hands detected in the images. Along with the attributes are defined all the necessary methods to read a given dataset.

3.2 Dataset extraction

As mentioned earlier, in order to train our CNN we will need all the labelled samples from the raw dataset that we downloaded. Fortunately the EgoHands dataset came pre-packed with some MatLab scripts that allowed us the extract all the images, the masks and the csv files. After we ran the MatLab scripts we found ourselves with roughly 5000 rgb images, 5000 grey-scale masks and 5000 csv files. From those images what we need are the segmented hand images and the RGB images with the bounding boxes, this heavy task was accomplished by running the *createDataset()* function defined in the *Dataset* class (you can find the code in the appendix);

The function writes into separate directories the set of segmented hand images, the hand images cut using the respecting bounding box coordinates and all the remaining background images which will be then lablled as 'notHand'. Those sets are then splitted into training and test dataset folders using a $0.8 - 0.2$ split ratio. As the raw dataset is quite large, the function takes an additional argument *maxSize* which is an integer specifying the maximum size of the dataset we want to extract. Here are some examples of the extracted images:

before each image is extracted, we preprocess the source image by applying a Gaussian blur, a sharpening filter and gamma correction. The preprocessing is done in order to remove the noise and to make the image more contrasty, while also adjusting the lighting.

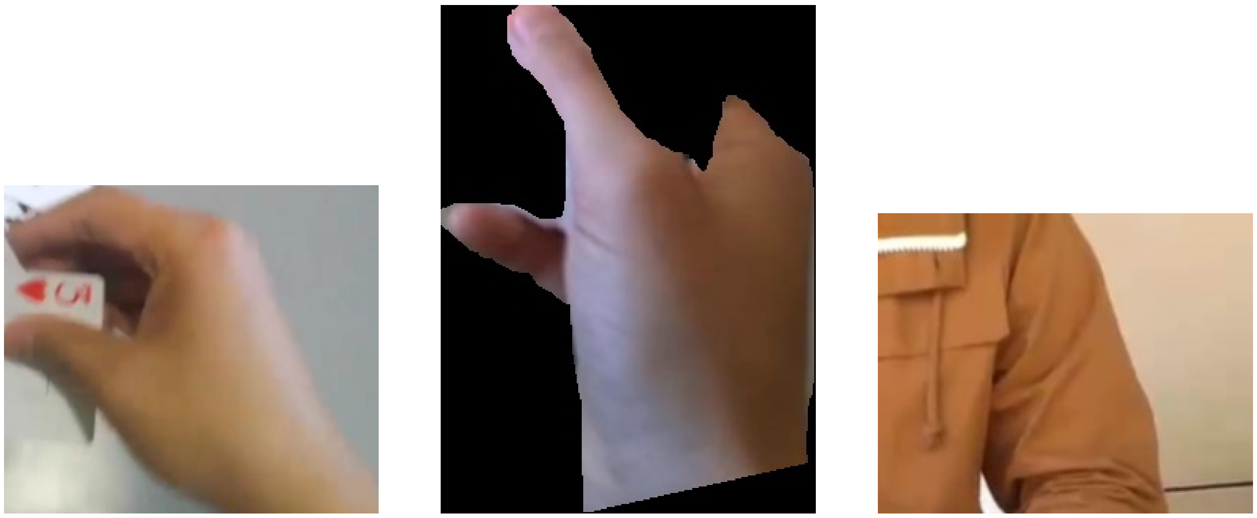

(a) Hand from Bounding box (b) Segmented hand (c) Background image

Figure 1: extracted dataset samples

4 Hand detection module [8 hrs]

Youssef Ben khalifa

The hand detection module is composed by two main components:

the fisrt one is developed in C++ acts ad the entry point for the entire program, and it is responsible for the fetching the input image and perform the preprocessing on it.

The second partm developed in python, performs the actual detection and classification of the image given as input argument by the C++ module.

The detection is based on the usage of a pre-trained YOLO CNN for both the detection of the ROI in the image and the classification of those based on two main classes, 'hand' and 'notHand'. The program then for each hand found writes into a csv file in determined directory that acts as the connection point between the c++ and python modules: in that csv file are written the coordinates for each hand found and classified by the CNN, that same file is then read by the first module and draws the bounding boxes based on the coordinates written in the csv file. Finally the input image is returned with the bounding boxes added.

4.1 Entry point for the module

The entire program's entry point is the first c++ module, which requires a string specifying the input image path:

that image is then read using the *imread*() method from the OpenCV library and fed to the *handDetectionModule*() function the starts the detection:

```
Mat handDetectionModule(Mat src){
        Mat srcProc;
        Preprocess::equalize(src, src);
        Preprocess::fixGamma(src,src);
        Preprocess::smooth(src, src);
        Preprocess::sharpenImage(src, src);
        imshow("test", src);
        waitKey();
        vector<HandMetadata> hands = detect(src);
        Mat dst = drawROI(src, hands);

        return dst;
    }
```

this function acts as the actual entry point for the hand detection module, as it takes as arguments the input image as a *Mat* object and starts off the detection chain:

the Mat object is preprocessed with a Gaussin filter and a laplacian filter, then given as argument to the *detect*() function

```
    vector<HandMetadata> detect(Mat src) {
    imwrite(activeDir + "src.jpg", src);

    string command = pytohnCommand;
    string result = exec(command);
    cout << result << endl;
    vector<int> handROICoords = readCSV(activeDir + "test.csv");

    vector<HandMetadata> hands;
    for(int i=0; i<handROICoords.size(); i+=4){
        vector<int> vec {&handROICoords[i], &handROICoords[i+4]};
        HandMetadata temp = loadHandMetadata(vec);
        hands.push_back(temp);
    }

    //cout << "Found " + to_string(hands.size()) + " hand" << endl;


    return hands;
}
```

which starts off the python module for the detection and classification.

4.2 Python module

The python module is called using the command line terminal by the detect() function we had in the c++ module, through the command line no arguments are passed along, as the image to process is taken from the *activeDir* folder used as the connection point between the two modules in which the input image is written by the c++ module for the python program to read. As soon as the python module is loaded, the YOLO CNN model is loaded along with it from the local configuration files saved under the directory *python/models/YOLO*:

```
    if __name__ == "__main__":
        srcPath = activeDir + "src.jpg"
        detect(srcPath, scoreThreshold)
        print("done")
        quit()
```

the *detect*() function reads the image path and loads it into the YOLO model to detect.

```
    def detect(srcPath):
    src = imread(srcPath)
    #detect ROI for hands for boundingBoxes
    handRoi = detectROI(src)
    rows = []
    for detection in handRoi:
        x, y, w, h = detection
        handFrame = src[x-100:x+w+100, y-100:y+h+100]
        savepath = activeDir + "test.jpg"
        imsave(savepath, handFrame)
        predicetedSrc = cnn.predict(savepath)
        print(predicetedSrc)
        rows.append([x,y,w,h])
    writeCsv(activeDir + "test.csv", rows)
```

In the detect method we read the image path and initialize it as a *Mat* object from the *cv2* library and feed it to the yolo object through the *detectROI*() function which returns a vector of bounding boxes. From those bounding boxes we extract the ROI for each hand and save it in the *activeDir* folder as a *csv* file.

4.3 ROI Drawing and output

Once the Python module is done writing the coordinates for the bounding boxes into the *csv* file, the a callback to the c++ module is done to continue the main execution and proceed to drawing the bounding boxes: this last task is achieved through the usage of the *drawROI*() function. The function takes as arguments the vector of objects *HandMetadata* which we previously read loaded from the csv file in the *detect*() function using the following code:

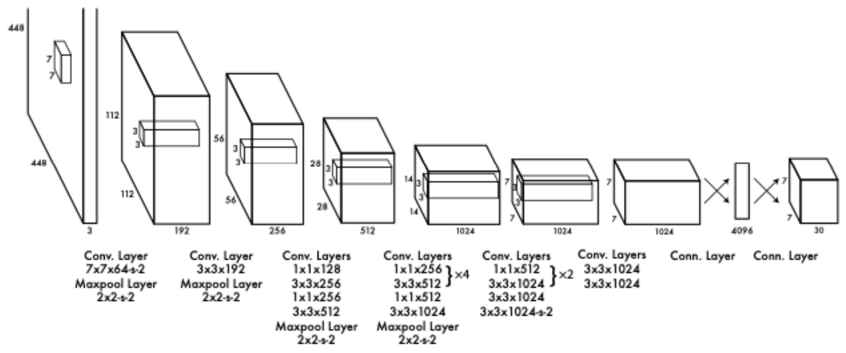
```
    vector<int> handROICoords = readCSV(activeDir + "test.csv");
    vector<HandMetadata> hands;
    for(int i=0; i<handROICoords.size(); i+=4){
        vector<int> vec {&handROICoords[i], &handROICoords[i+4]};
        HandMetadata temp = loadHandMetadata(vec);
        hands.push_back(temp);
    }
```

amongst those line you'll notice a particular method called *readCSV*(), this is one is the same one we used during the dataset extraction at the beginning which we can easily recycle to accomplish the task of reading the *csv* file containing the Hands coordinates. The *vector¡HandMetadata¿ hands* is then passed along to the *detectROI*; this function return the output image, again as a *Mat* object, with the bounding boxes drawn in different colors, one for each hand detected.

```
    Mat drawROI(Mat src, vector<HandMetadata> hands){
    Mat dst = src.clone();
    int colorId = 0;
    int thickness = 2;
    for(auto hand : hands){
        Point p1(hand.PosX, hand.PosY);
        Point p2(hand.PosX + hand.Width, hand.PosY + hand.Height);
        rectangle(dst, p1, p2,
                colors.at(colorId),
                thickness, LINE_8);
        colorId += 1;
        if(colorId == colors.size()) colorId = 0;
    }

    return dst;
}
```

4.4 YOLO model

This is a particular type of CNN which we re-trained on the egohands dataset we extracted at the beginning of the project. It is based on the YOLOv3 architecture, which is a convolutional neural network designed specifically for object detection and classification:



448
7
7
448
3
112
3
3
112
192
56
3
3
56
256
28
3
3
28
512
14
3
3
14
1024
7
3
3
7
1024
7
7
1024
4096
7
7
30
Conv. Layer 7x7x64-s-2 Maxpool Layer 2x2-s-2
Conv. Layer 3x3x192 Maxpool Layer 2x2-s-2
Conv. Layers 1x1x128 3x3x256 1x1x256 3x3x512 Maxpool Layer 2x2-s-2
Conv. Layers 1x1x256 3x3x512 }×4 1x1x512 3x3x1024 Maxpool Layer 2x2-s-2
Conv. Layers 1x1x512 3x3x1024 }×2 3x3x1024 3x3x1024-s-2
Conv. Layers 3x3x1024 3x3x1024
Conn. Layer
Conn. Layer

in our program we initialize the YOLO object the code line

```
    yolo = YOLO('../python/models/yolo/yoloTrained.cfg',
            '../python/models/yolo/yoloTrained.weights', ["hand"])
```

in which we specify the class to detect and nothing else. All the other paramters are already set to the best values in the YOLO class constructor.

which we found at the beginning of the *main.py* script. The object is loaded form the class *YOLO.py* in which we find all the methods needed to initlize the object and perform the detection. We then defined the model as follows:

```
nc: 1
depth_multiple: 0.33
width_multiple: 0.50

anchors:
  - [10,13, 16,30, 33,23]
  - [30,61, 62,45, 59,119]
  - [116,90, 156,198, 373,326]
```

```
backbone:
  [[-1, 1, Focus, [64, 3]],
   [-1, 1, Conv, [128, 3, 2]],
   [-1, 3, Bottleneck, [128]],
   [-1, 1, Conv, [256, 3, 2]],
   [-1, 9, BottleneckCSP, [256]],
   [-1, 1, Conv, [512, 3, 2]],
   [-1, 9, BottleneckCSP, [512]],
   [-1, 1, Conv, [1024, 3, 2]],
   [-1, 1, SPP, [1024, [5, 9, 13]]],
   [-1, 6, BottleneckCSP, [1024]],
  ]

head:
  [[-1, 3, BottleneckCSP, [1024, False]],
   [-1, 1, nn.Conv2d, [na * (nc + 5), 1, 1, 0]],
   [-2, 1, nn.Upsample, [None, 2, "nearest"]],
   [[-1, 6], 1, Concat, [1]],
   [-1, 1, Conv, [512, 1, 1]],
   [-1, 3, BottleneckCSP, [512, False]],
   [-1, 1, nn.Conv2d, [na * (nc + 5), 1, 1, 0]],
   [-2, 1, nn.Upsample, [None, 2, "nearest"]],
   [[-1, 4], 1, Concat, [1]],
   [-1, 1, Conv, [256, 1, 1]],
   [-1, 3, BottleneckCSP, [256, False]],
   [-1, 1, nn.Conv2d, [na * (nc + 5), 1, 1, 0]],

   [[], 1, Detect, [nc, anchors]],
  ]
```

the same model is then trained using our custom dataset we extracted mostly from the egohands dataset and some third-party sources, using a dedicated python script we find in the *yoloV5* folder. The script outputs two main files: a *.cfg* file containing the configuration for our CNN, and a *.weights* file containing the weights. Those files are the ones we find in the *models/yolo* folder that will be loaded into the python script.

The YOLO model is implemented into the python model through the *YOLO.py* class we find under the *python* directory, in which we find along with the constructor for the class, the *inference()* method:

```
    def inference(self, image):
        ih, iw = image.shape[:2]
        blob = cv2.dnn.blobFromImage(
            image, 1 / 255.0, (self.size, self.size), swapRB=True, crop=False)
        self.net.setInput(blob)
        layerOutputs = self.net.forward(self.output_names)
        boxes = []
        confidences = []
        classIDs = []
        for output in layerOutputs:
            for detection in output:

                scores = detection[5:]
                classID = np.argmax(scores)
                confidence = scores[classID]
                if confidence > self.confidence:

                    box = detection[0:4] * np.array([iw, ih, iw, ih])
                    (centerX, centerY, width, height) = box.astype("int")

                    x = int(centerX - (width / 2))
                    y = int(centerY - (height / 2))

                    boxes.append([x, y, int(width), int(height)])
                    confidences.append(float(confidence))
                    classIDs.append(classID)

        idxs = cv2.dnn.NMSBoxes(
            boxes, confidences, self.confidence, self.threshold)

        results = []
        if len(idxs) > 0:
            for i in idxs.flatten():
                # extract the bounding box coordinates
                x, y = (boxes[i][0], boxes[i][1])
                w, h = (boxes[i][2], boxes[i][3])
                id = classIDs[i]
                confidence = confidences[i]

                results.append((x, y, w, h))

        return results
```

which takes a *Mat* object as input and returns a list of vector coordinates (x, y, w, h), that correspond respectively (x, y) coordinates of the top-left corner of the box, its width and height. These values are returned for each hand detected in the input image.

The hand detection made by the YOLO module is particularly biased by one main parameter, which is the *confidence*: this determines the probability of the image analyzed by the module to be a hand. The *confidence* is set to 0.5 by default, but can be changed in the constructor of the *YOLO* class.

The main reason for such a low confidence value is the simple fact that we are interested in detecting one single class label, which is 'hand'. All other possible labels are considered as 'notHand', therefore not detected. Thus 50% of the detected labels are hands.

4.5 Model training

The training of the YOLO model was done using the *train.py* script, which is found in the *yolo* directory: the script can be executed by typing

```
    !python train.py --img 416 --batch 32 --epochs 25
        --data <path to the dataset configuration file>
        --cfg <path to the model configuration file>
```

in the terminal. The needed file must be *.yaml* files, which we easily converted to from the original *.csv* files we got during our dataset extraction using a simple tool found online.

The training was done using roughly 1000 images per epoch, and a batch size of 32: the training of one single model took about 2.5 hours to complete running on a local machine with a dedicated GPU. The configuration file are finally saved in the *models/yolo* folder, which we then used to perform the detection. In total we attempted the detection using 3 different models, each with different configuration parameters and structures and choose the best one based on the evaluation ran on a separate test set extracted using our Dataset class from the c++ module. Here are the results of the evaluation:

- **Model 1**: training score: 0.988, test score: 0.981;
- **Model 2**: training score: 0.879, test score: 0.872;
- **Model 3**: training score: 0.953, test score: 0.918;

4.6 Overfitting and Data augmentation

The main problem we may encounter by using mostly images from the same dataset to train the CNN models is overfitting. This may happen whenever the training dataset may not be diverse enough, and lead to generating features that work perfectly as long as we work on the same data that come from the same source. We can avoid this problem by using data augmentation, which is a technique that allows us to generate various type of images from the same images we have in our dataset, by simply applying random transformations, and applying different size rescales onto the images.

4.7 Usage

The detection can be done calling the **handDetectionModule()** function we find in the *main.cpp* source file, which takes as paramters a Mat object representing the input image. It return a Mat object with the drawing boxes drawn on top of the input image. If you wish to check the numerical coordinates for the bounding boxes, you can directly check the *test.csv* file generated by the python script under the *activeDir* folder.

4.8 Important Notes

Note that the only way that the c++ and python module can communicate and work properly together in thought the **activeDir** directory we find the root project folder, so DO NOT DELETE the activeDir, or if you do, recreate it!

One last thing to note is the python launch command in the *main.cpp* file: you need to adjust it based on your local machine.

5 Hand Segmentation Module

Roberto del Ben

5.1 Coarse segmentation [2 hrs]

We chose to make a first coarse segmentation based on a range of colors that skin can have inside images. First of all we consider the *YCrCb* color space, since in the RGB color space similar colors do not always correspond to similar values. Then, since this range isn't well defined due to different illumination sources or camera settings, we used a very large portion of the *YCrCb* spectrum in order to exclude only very unlikely combinations. These values came from some researches done in the internet and validated through the dataset by the following code:

```
Mat HandsSegmentation::GetSkinMask() {
    int min_cr = 135, max_cr = 180, min_cb = 85, max_cb = 135;
    Mat ycrcb_range, ycrcb_ms_image;
    cvtColor(image, ycrcb_ms_image, COLOR_BGR2YCrCb);
    inRange(ycrcb_ms_image, Scalar(0, min_cr, min_cb),
            Scalar(255, max_cr, max_cb), ycrcb_range);

    Mat mask;
    auto element_type = MORPH_ELLIPSE;
    auto element_size = 6;
    Mat element = getStructuringElement(element_type,
                   Size(element_size * 2 + 1, element_size * 2 + 1));

    morphologyEx(ycrcb_range, mask, MORPH_CLOSE, element);

    return mask;
}
```

5.2 Image clustering [16 hrs]

The first thought that concerned us about clustering is that picking a random image in the dataset, even with perfect detection we cannot know in advance how many clusters we should look for. This is the main reason why we chose to implement Mean Shift in order to clusterize the ROI containing a hand.

```
MeanShift(const Mat &_image, const int &_spatial_bandwidth = 4,
          const double &_color_bandwidth = 4);
```

The MeanShift class accepts 3 parameters:

- The ground image used as the underline sample group
- Spatial bandwidth
- Color bandwidth

The split of the spatial bandwidth and the color bandwidth guarrantees to preserve discontinuities of the image.

Once initialized the clustering method can be called with the function *cluster*:

```
std::vector<Cluster> cluster(const Sample *shifted_points);
```

Sample is an utility class written to keep information about the shift of the points. Cluster is a *struct* containing the converging mode and the list of points belonging to the cluster.

We use the shifted points as input for a region growing algorithm to construct the vector of clusters. The growing rule is based on the color bandwidth and collects similar modes under the same cluster.

5.3 Hand segmentation [8 hrs]

We decided to speed up the algorithm implementing two tricks: the first is the use a FIFO queue ofthreads that manages to run the points shifting in a parallel process; the second is to shift only points that are not excluded by the skin detection in the coarse segmentation. This last point requires to exclude from the segmentation all the clusters with invalid mode, that is values with *color = Scalar(0,0,0)*.

```
vector<Sample> HandsSegmentation::GetSamples(const Mat &from) {
    vector<Sample> samples;

    for (int r = 0; r < from.rows; r += 1)
        for (int c = 0; c < from.cols; c += 1) {
            Sample temp = Sample(from, r, c);
            if (temp.color[0] >= 1)
                samples.push_back(temp);
        }

    return samples;
}
```

```
void MeanShift::meanshift(const vector<Sample> &_points) {
    vector<thread> threads;
    for_each(_points.begin(), _points.end(), [&](auto &&point) {
        bool started = false;
        while (!started) {
            if (threads.size() < MAX_THREADS) {
                threads.push_back(thread(&MeanShift::meanshiftSinglePoint,
                                         this, point));
                started = true;

            } else {
                threads[0].join();
                threads.erase(threads.begin());
            }
        }
    });

    while (threads.size() > 0) {
        threads[0].join();
        threads.erase(threads.begin());
    }
}
```

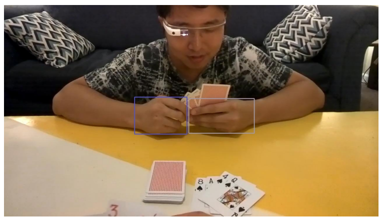
Once MeanShift has completed we pick the largest cluster as the one that segments the hand. We expect the hand to cover the majority of the image, so if the largest cluster still does not occupy more than 40% of the valid samples, the algorithm re-runs with a larger color bandwidth.

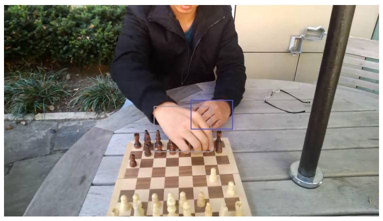
```
Mat HandsSegmentation::MSSegment(const Mat &input,
        const int &spatial_bandwidth, const double &color_bandwidth) {

    vector<Sample> samples = GetSamples(input);

    MeanShift *ms = new MeanShift(input, spatial_bandwidth, color_bandwidth);

    vector<Cluster> clusters = ms->cluster(samples);

    Cluster max_cluster = SelectLargestCluster(clusters);

    if (max_cluster.shifted_points.size() < 0.4 * samples.size()) {
        cout << "Regeneration" << endl;
        return MSSegment(input, spatial_bandwidth, color_bandwidth + 0.5);
    }

    return CreateMaskFromCluster(max_cluster, input.size());
}
```

6 Benchmark results

6.1 Hand detection

Here are the Benchmark results of the Hand detection module form the first 20 images of the Benchmark folder that was given to us:


(a) 01.jpg - Score = 0.996582


(b) 02.jpg - Score = 0.949414

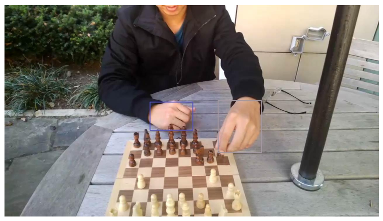
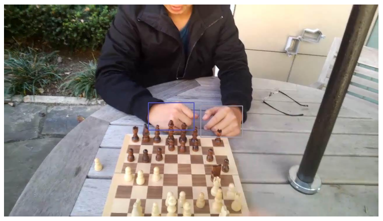
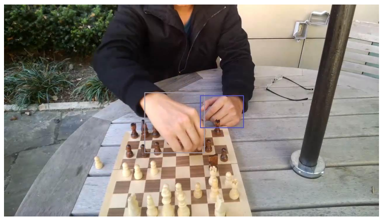

(c) 03.jpg - Score = 0.995508

Figure 2: extracted dataaset samples

Here are the numerical results of the bounding boxes (x,y,width,height) of the hand detected in the images.


(a) 04.jpg - Score = 0.994824


(b) 05.jpg - Score = 0.932324

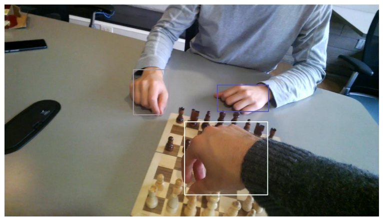
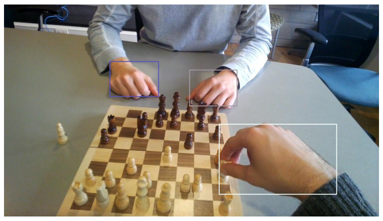
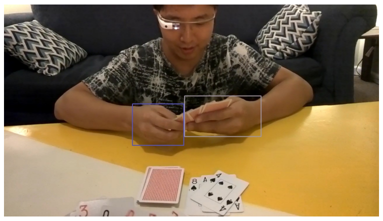

(c) 06.jpg - Score = 0.922852

Figure 3: extracted dataaset samples


(a) 07.jpg - Score = 0.876497


(b) 08.jpg - Score = 0.994336

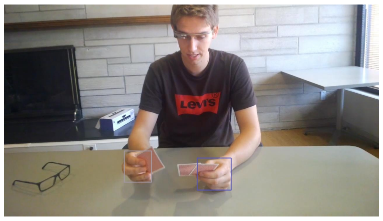
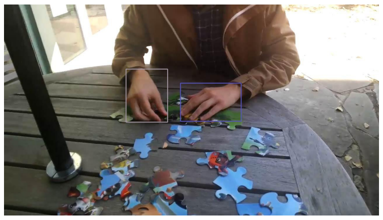
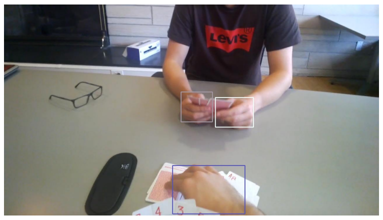

(c) 09.jpg - Score = 0.938281

Figure 4: extracted dataaset samples


(a) 10.jpg - Score = 0.931445


(b) 11.jpg - Score = 0.918620

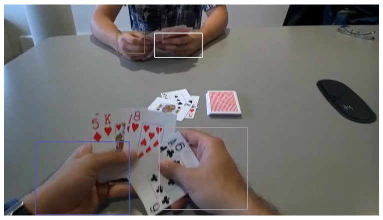
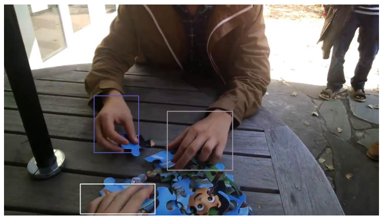
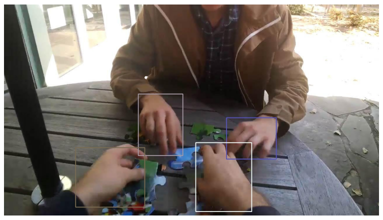

(c) 12.jpg - Score = 0.941895

Figure 5: extracted dataaset samples


(a) 13.jpg - Score = 0.899609


(b) 14.jpg - Score = 0.925391

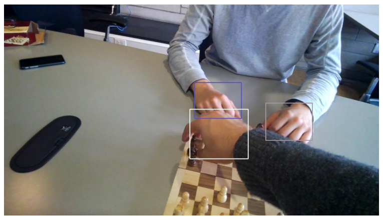
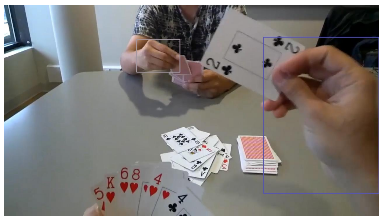
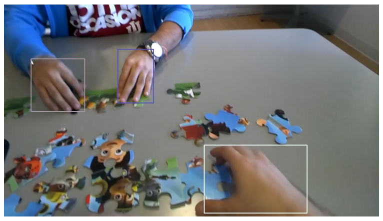

(c) 15.jpg - Score = 0.939063

Figure 6: extracted dataaset samples


(a) 16.jpg - Score = 0.898242


(b) 17.jpg - Score = 0.851823

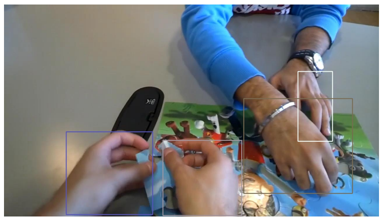
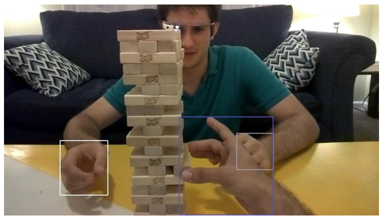
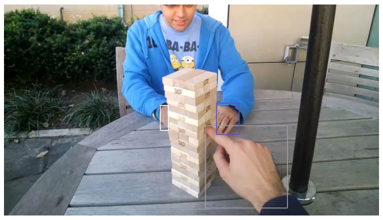

(c) 18.jpg - Score = 0.931738

Figure 7: extracted dataaset samples


(a) 19.jpg - Score = 0.918034


(b) 20.jpg - Score = 0.994336

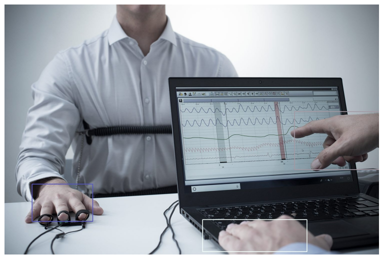

(c) Random image

Figure 8: extracted dataaset samples

Image	Score	Box Values
01.jpg	0.996582	631,318,224,118 446,313,179,126
02.jpg	0.949414	512,345,194,154 638,324,142,98
03.jpg	0.995508	730,325,148,177 497,330,146,96
04.jpg	0.994824	672,344,143,106 493,333,156,93
05.jpg	0.932324	480,299,204,204 670,309,147,109
06.jpg	0.922852	440,221,121,153 727,273,175,90 615,398,284,248
07.jpg	0.876497	633,222,162,122 361,193,166,119 734,406,401,238
08.jpg	0.994336	616,310,261,138 438,337,175,141
09.jpg	0.938281	410,497,93,106 659,521,116,109
10.jpg	0.931445	416,220,142,181 602,268,207,130
11.jpg	0.918620	604,298,107,102 575,549,247,165 721,317,130,101
12.jpg	0.941895	529,419,302,280 107,468,320,249 514,97,162,83 385,93,95,73
13.jpg	0.899609	557,364,223,201 307,311,152,194 260,612,255,102

Image	Score	Box Values
14.jpg	0.925391	458,305,152,210 759,386,170,141 654,471,191,235 244,488,236,203
15.jpg	0.939063	893,336,159,127 650,266,160,122 634,357,199,169
16.jpg	0.898242	452,117,148,112 886,110,394,532
17.jpg	0.851823	92,184,183,179 388,152,122,182 683,477,351,231
18.jpg	0.931738	542,461,269,255 212,432,295,281 1003,227,118,237 818,319,371,322
19.jpg	0.918034	794,439,125,124 598,383,320,334 191,464,163,186
20.jpg	0.994336	687,413,280,281 725,336,77,107 533,344,28,85

7 Failed attempts

For the detection module, we first tried using a simple CNN from scratch and train it as a simple classifier, to then scan the entire image using a pre-fixed sized window and classify it as a hand or notHand. This method was not successful, as the results were not very good: the model in fact suffering from serious overfitting and the detection took way too long for it to be a viable solution. We then trained different CNN configurations, but all of them failed to perform the detection in a reasonable amount of time, and also the results were very sensible to the chosen size of the window.

Next we tried to implement the detection using *haar* features, which were not successful either, as the trained model was again suffering from overfitting and heavy noise sensibility: the haar features performed reasonably well and within a decent amount of processing time, but as soon as we tried images that did not come from the egohands dataset the results got much worse.

Finally the last thing we tried was to train and use two CNN's in parallel and perform the classification using the output of both. The first CNN was trained as the first one on an enhanced dataset w.r.t. the original one, and it was trained using the segmented hand images; the second one was trained on a dataset containing the same images onto which we applied a line detection algorithm: for that we recycled the canny and hough transform algorithm we used in one of the laboratories during the course. The results were decent enough, but there was still the problem of running time, especially since when we deal with 1280x720 images.

8 Final Considerations

For the hand detection module, the usage of multiple CNN's to attempt to detect and classify hands or any type of object made us realize just how sensible the results depend on the dataset we used to train the different nets: most of them worked only if the detection window was the same as the one used for training, others suffered from extreme overfitting and noise sensitivity. It is not surprising that even the results from the implement YOLO model, which was the most efficient one we tried, still ended up with some overfitting from adopting mainly one single dataset.

The main improvement on the hand detection algorithm is then the creation and usage of a more diverse dataset to train the CNN's, in order to avoid phenomenon. As for the rest of the first module it works very well on the Benchmark images, in fact we do not have one single detection score which is under 85%, and also thanks to the usage of the tensorflow library and a discrete GPU the algorithm revealed to be very fast. The main reason for this efficiency, however, is the YOLO model architecture we decided to adopt for our detector: thanks to optimized layers and configuration the search for the ROI's inside the image was extremely fast compared to any other CNN we used thus far.