Computer Vision - Project Report

Roberto Del Ben - Youssef Ben Khalifa

July 28, 2022

Contents

1 Introduction

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

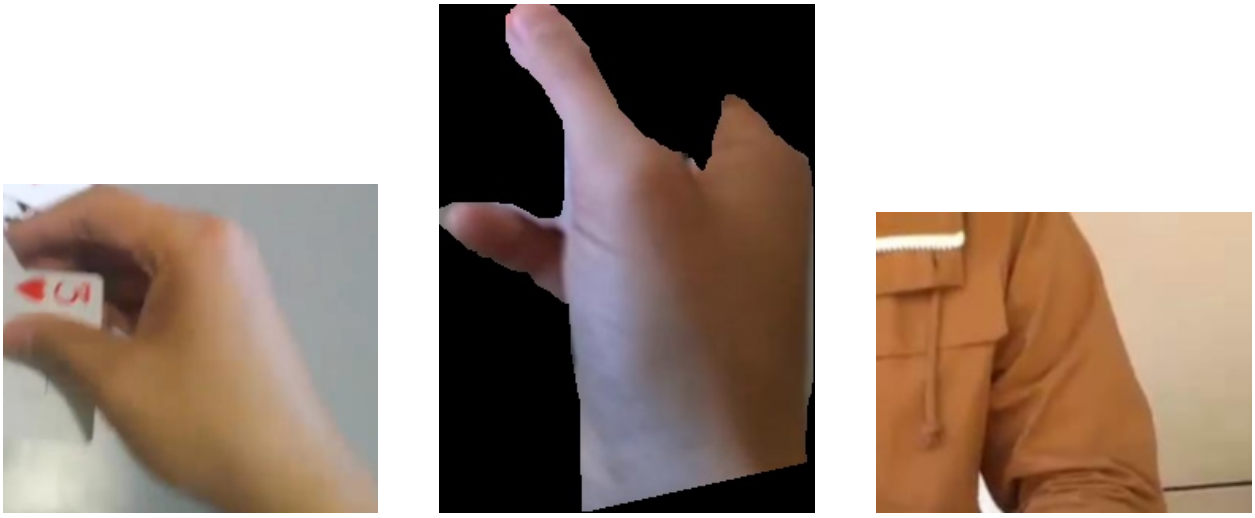
To start off, we defined the structure of our dataset in the c++ module, which can be found the *dataset.h* file: in particular we defined the *HandMetadata* and *Image* as follows:

```
    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();
};
```



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

Figure 1: extracted dataaset samples

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.

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 Main()

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.

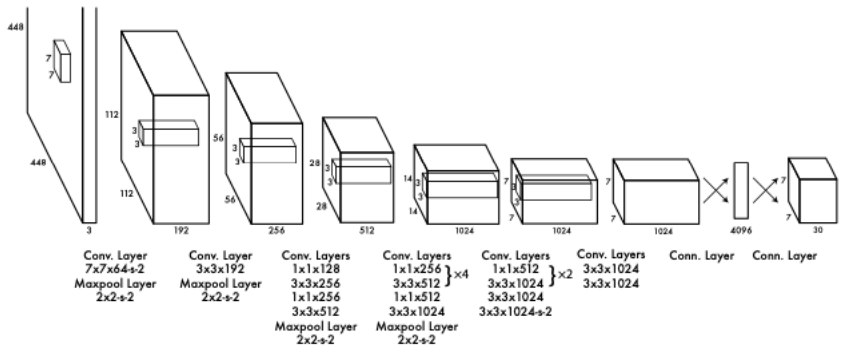
```
Mat drawROI(Mat src, vector<HandMetadata> hands){
Mat dst = src.clone();
int colorId = 0;
int thickness = 2;
for(auto hand : hands){
    Point p1(hand.PosX, hand.PosY);
```

```
        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 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:



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: 3
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.

5 Hand Segmentation

Robert del Ben

6 Benchmark results

7 Failed attempts

8 Final Considerations