Balzan Pietro 1241795

COMPUTER VISION FINAL PROJECT

1. INTRODUCTION

The goal of this project is to develop a system capable of recognizing trees inside an image by drawing a bounding box around them.

In order to tackle this problem I decided to base my approach on the Viola & Jones Face detector, applying it on trees instead of faces, and, as such, training and using a cascade of weak classifiers that is able to recognize trees in a sliding window, with fast feature evaluation and rejection in case of non-positive images.

The program is organized in two files:

- `project_main.cpp` which contains the main function.
- `TreeDetector.cpp` which contains the implementation of class `TreeDetector`, which performs the image pre-processing and the trees detection.

2. MAIN FUNCTION

The program starts by reading input strings from the command line: the first input should be the path to the cascade classifier, and the second, the path to the folder containing input images to perform the tree detection on.

Since the cascades are all stored in the build/cascades folder, the first cmd argument should look something like "`cascades/final/cascade.xml`" ("final" is the final version of the trained cascade, 3 other previous versions can also be tested to see its evolution).

The program now gathers the names of the images in the specified directory, the same way in which it was done in previous homeworks.

After this, an object of class `TreeDetector` is initialized through its default constructor, and the cascade is loaded with the method `tDetector.setCascade(cascadePath)`, which signals the success of the operation with an int value. .setCascade just calls the `.load(cascadePath)` method on the protected class variable `tree_cascade` of type `CascadeClassifier`.

Now, the subsequent "for" loop loads, in each iteration, one (.jpg) image from the set found in the given directory, and passes it as argument of the `doDetection(Mat img)` TreeDetector method, which performs the detection after pre-processing the image.

It is important to note that the first image in the loop takes some time (a couple of seconds) before showing the results, probably because of the fact that the cascade .xml file is loaded in memory when the detection actually starts. The subsequent images are almost instantaneous (based on the experience with my machine).

After analyzing every input image, the program successfully closes.

3. IMAGE PRE-PROCESSING AND TREES DETECTION

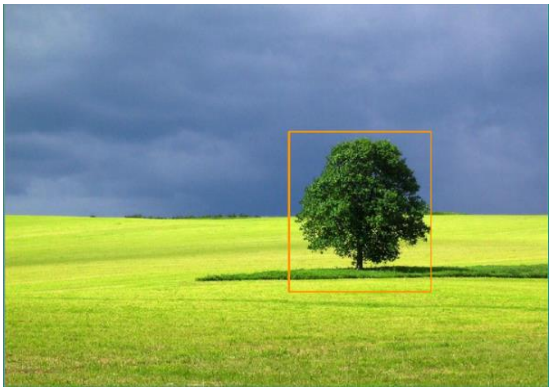
Before starting the detection, in order to improve the result, `doDetection(Mat img)` has to perform some image pre processing which aims at removing noise, disturbances and uninteresting parts of the image that could produce false positives (the sky, for example, was a source of many mistakes for the cascade).

First of all the image is turned into grayscale and denoised with a simple normalized box filter via the `blur()`method. I decided to make the size of the kernel change based on the resolution of the image (a size which worked well on low-res images would produce almost unnoticeable results in the highest-res images), and found a good solution to be a kernel of size $1/60^{th}$ of the number of image columns. After the smoothing, it is also equalized via the `equalizeHist()` function.

Now, in order to remove the least interesting parts of the image, which in a context of tree detection are the non-green-ish parts, we can simply switch to HSV representation of the image and select a range of values in the Hue channel in order to compute a sort of "mask" of the image containing the non-green values (white part of the example image below). The mask calculation is done by calling `inRange()`, which computes a binary matrix that sets to 1 every pixel with HSV values inside the specified range. In this case I kept every S and V value and removed all H values in the green.

Now that we have the mask, we can simply use it to decide whether to keep pixels in the grayscale image or not: as you can see form the image below, all pixels that were black in the mask are kept, and all white ones become white in the final image too. This is simply done by running `bitwise_or()` between the mask and the grayscale's matrixes.

Now, the image pre-processing is over, and we can run the detector on the resulting image through `tree_cascade.detect_multiscale()` function on the CascadeClassifier object of the TreeDetector class. This returns a simple vector of `Rec` (rectangles), which are used to draw the bounding boxes on the final image. In order to further avoid false positives that could come from the grass or other small plants or bushes, I decided to set a minimum window size for the sliding window on which the cascade operates (set to $1/8^{th}$ of the height and width of the image).



4. TRAINING THE CASCADE

At the core of the project there is the cascade of classifiers used to perform the detection, which was obtained through a training process resembling that of neural networks: the classifiers in the chain were obtained through training on a set of small greyscale images representing both trees and non-trees, along a series of steps (epochs).

To obtain the training set, I downloaded a number of POSITIVE images (images of trees), rescaled them to a size of 50*50 and turned them into grayscale with a few lines of code (that I don't find of interest enough to expand upon). I also did the same for images of NEGATIVE samples (mainly wide panoramas in which single trees are not recognizable, but in general, any image of non-trees).

In order to obtain a bigger set, I used data augmentation through the OpenCV command line function `opencv_createsamples` which can be used to generate positives by placing a positive image on many negative ones in a randomized way. This method also creates a text file (which will be needed for the training) describing, for each image, the position of the positive portion in the generated images.

```
opencv_createsamples -img tree.jpg -bg bginfo.txt -info info/info.lst -
pngoutput info -maxxangle 0.2 -maxyangle 0.2 -maxzangle 0.2 -num 20
```

With this command we can produce (-num) 20 positive images which are a combination of the background images listed in bginfo.txt and the positive sample provided by (-img) tree.jpg. The output file is info.lst

After this step, before training the classifier, it was required to generate a `.vec` file of the positive samples, which collects the positive samples into a vec format. For this, the generated vectors were downscaled to 35*35 (in the final version of the classifier) in order to make the training quicker (in the first versions of the classifier the vecs were 20*20 but this little resolution resulted in having too many false positives).

```
opencv_createsamples -info treesinfo.txt -num 225 -h 35 -w 35 -vec trees.vec
```

Finally, the cascade was trained with the following command:

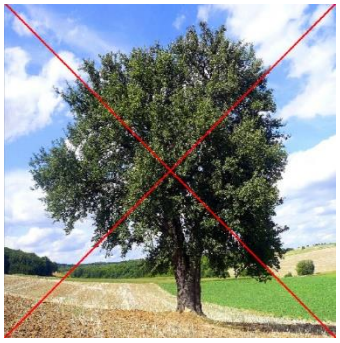
```
opencv_traincascade -data data -vec trees.vec -bg bginfo.txt -numPos 225
          -numNeg 110 -numStages 20 -w 35 -h 35 -featureType LBP
          -maxFalseAlarmRate 0.2 -minHitRate 0.998
```

which trains a cascade classifier of LBP features (I didn't use HAAR features because they are incredibly slow to train) from 225 samples of positive images described by trees.vec and 110 negatives listed in bginfo.txt, along 20 stages of training, trying to hit a max rate of false positives of 0.2 (for each specific stage) and a minimum hit rate of 0.998.

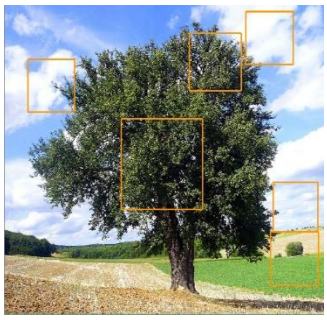
Among the files I sent, there is a folder in the build directory, “cascades”, containing various versions of the classifier, which can all be tested by simply changing the command line arguments:

- v1 and v2 are not present because they performed too poorly, both in terms of quality of the result and computation running time: they were trained with less than 100 positives and a very small size of vecs (20*20).

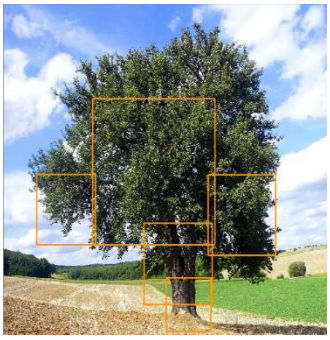
- v3 shows the first signs of promising results. It misses some basic “easy” to recognize trees (for example the one in Figure 5 of the benchmark dataset). The training set was still small but I increased the size of vecs to 35*35 and set stricter parameters (maxFalseAlarmRate =0.2 and minHitRate = 0.998).



- v4 has the opposite problem: too many false positives. This was probably due to the fact that, in the fear of overfitting the classifier on the training data, I reduced the number of training stages.



- v5 was obtained with a larger dataset but still resulted in many false positives.

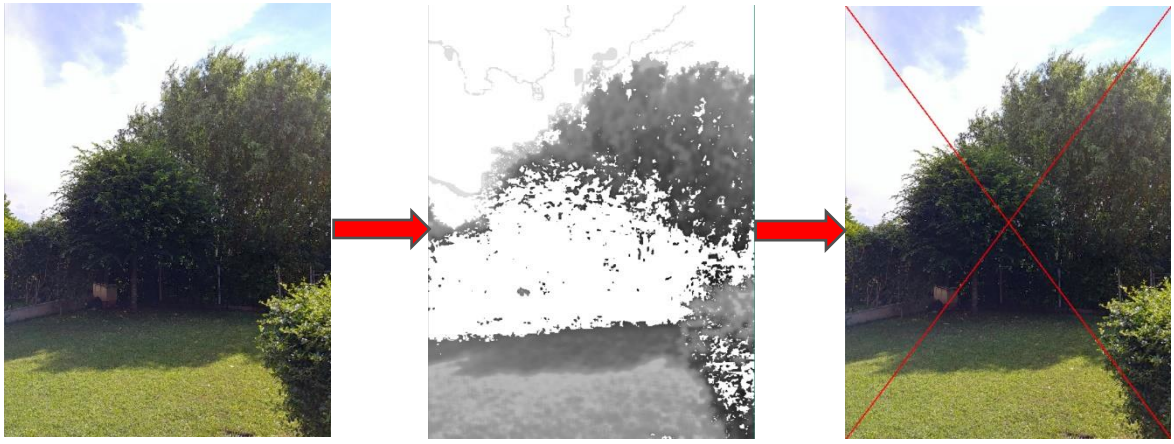


- “final” is the best version of the classifier: it performs very well on the benchmark images thanks to the even larger dataset and training parameters, although maybe it is too “strict” (may have suffered from some overfitting) and could be improved. It is important to note that this performance was obtained with a small dataset (225 positives and 110 negatives) which proves how well Viola & Jones approach works for this problem.

5. POSSIBLE IMPROVEMENTS

- For sure, the dataset with which the cascade classifier was trained was not completely sufficient in terms of sheer number of images, so I think that with a larger training set better results are surely achievable.

- Sometimes, If the image is particularly dark, the Hue thresholding will eliminate the parts containing the trees, so a better job could be done with the range selection.



Furthermore, in some cases, like in the image above, the detection was better without any of the pre-processing of the image (just blur, grayscale transformation and histogram equalization were enough), but overall, the general performance improved on most images when applying the Hue mask.

Some examples of the results on other pictures I took myself (all using the final cascade):