The project attached to this report refers to a hand detection and segmentation task, a very important task which is useful in various modern application, in particular for applications of human-robot interaction, gesture recognition or human activity analysis.

The overall project can be seen as a combination of two separate subprojects: hand detection and hand segmentation.

# Hand Detection

For the first step we tried to use the concept of Bag of Words, which classifies an image by using a vocabulary of “words” that in this case correspond s to features. In general, in document classification a bag of words is a sparse vector of occurrence counts of words, that is a sparse histogram over vocabulary, while in computer vision, a bag of visual words is a vector of occurrence counts of a vocabulary of local image features. This strategy relies mainly on three steps:

1. Feature detection, which selects and highlights the most meaningful points in the image. The selected image must be of a detailed object that we want to detect in the final steps.
2. Feature description, which are vectors describing the area surrounding each key point detected in the previous step. Typically, these two steps can be obtained with SIFT (Scale Invariant Feature Transform), which converts each patch to a 128-dimensional vector.
3. Codebook generation, the core of the algorithm which converts the vector represented patches to “codewords”, considerate as a representative of several similar patches. To implement this part is need a clustering technique, like k-means clustering, in order to group similar patches to a one large single patch. Codewords are defined as the centre of the learned clusters and the image can be represented by the histogram of codewords.

However, we noticed that even if the algorithm has good time and computational complexity performance, the overall results didn’t match with the requirements, because we only managed to classify images as with hand or without hand and not finding the location on the image.

Therefore, for the core of this first part, we decided to work with a boosted cascade of weak classifiers, which is divided in two stages: the training and the detection stage.

For training a boosted cascade of weak classifiers we need a set of positive samples, containing the hands, and a set of negative images, which corresponds typically to the background of the images used for the positive samples.

For both sets we used the images provided by the EgoHands dataset, which is composed by various frames of a video where two individuals interact with each other by playing games, which means that at most four hands should be detected.

However, in order to be able to use this technique with OpenCV and c++, we need to downgrade our version of OpenCV to the latest update of the OpenCV 3.4. This is because in more recent versions all the tools for training and creating samples for the boosted cascade of weak classifiers were eliminated.

The first step was to select and generate the bounding boxes for the training part, in particular we used: opencv\_annotations.exe to manually select the bounding boxes and saving the results in a file txt with the following form: the first element of the line is the filename, followed by the number of object annotations, followed by numbers describing the coordinates of the objects bounding rectangles (x, y, width, height). While for the negative images we used the same images as for the positive, but with a skin colour mask applied on it in order to “cover” the hands and keep then only for the positive while keeping the background as negative. Command line:

/opencv\_annotations.exe -images <EgoHands dataset path> -annotations positive.txt

Where:

-images: path to folder containing images with our objects;

-annotations: path to annotations txt file, where we want to store our annotations, which is then passed to the -info parameter;

The next step was to create the sample, which are necessary to the boosting process to define what the model should actually look for when trying to find the object of interest, that is the hands. The scheme for the creation of this samples follows: the object instances are taken from the given images by cutting out the supplied bounding boxes from the original images, then they are resized to target sample size (defined by parameter -h and -w) and stored in a vec-file.

Command line:

/opencv\_create\_samples.exe -info <collection file name> -vec <vec\_file\_name> -num <number of samples> -w <sample width> -h <sample height>

Where:

-info: Description file of marked up images collection (positive.txt file from previous step);

-vec: Name of the output file containing the positive samples for training;

-num: Number of positive samples to generate;

-w: width in pixels of the output samples;

-h: height in pixels of the output samples;

The last step is the actual training of the boosted cascade of weak classifiers, based on the positive and negative dataset that was prepared before. This part was done by opencv\_traincascade application, which can be tuned with different parameters to achieve different accuracy of results. In particular, the parameter that makes the most difference is the -featureType, which can be either Haar features or LBP features.

The first one is defined as the difference of the sum of pixels of areas inside the rectangle, which can be at any position and scale within the original image. Each feature type can indicate certain characteristics in the image, such as edges or changes in texture. The second one is created by: dividing the examined window into cells (16x16 pixels for each cell), then for each pixel in a cell, compare the pixel to each of its 8 neighbours (follows the pixels along a circle) and where the centre pixel’s value is greater than the neighbour’s value, write ‘0’. Otherwise, write ‘1’, this gives an 8-digit binary number. After that, compute the histogram of the frequency of each “number” occurring. This histogram can be seen as a 256-dimensional feature vector. Then we optimally normalize the histogram and concatenate them. This gives a feature vector for the entire window.

In our project we decided to use the LBP features because they were much faster, for 2500 positive samples and 10000 negative it took about 3 hours, while with Haar features it would probably take around 5-6 days. However, the Haar features are slightly more precise.

Command line:

/opencv\_traincascade -data<cascade dir name> -vec <vec file name> -bg <background\_file\_name> -numPos <number\_of\_positive\_samples> -numNeg <number of negative samples> -numStages <number of stages> -precalcValBufSize <precalculated vals buffer size in Mb> -precalcIdxBufSize <precalculated idxs buffer size in Mb> -acceptanceRatioBreakValue <break value> -featureType <{HARR, LBP}> -w <sample width> -h <sample height> -minHitRate <min hit rate> -maxFalseAlarm <max false alarm>

Where:

-data: where the trained classifier should be stored;

-vec: vec file with positive samples (created from the previous step);

-bg: background description file, that is the file containing the negative sample images;

-numPos: number of positive samples used in training for every classifier stage(2500);

-numNeg: number of negative samples used in training for every classifier stage (10000);

-numStages: number of cascade stages to be trained (12);

-precalcValBufSize: size of the buffer for precalculated feature values;

-precalcIdxBufSize: size of the buffer for precalculated feature indices, the sum with the previous number should not exceed the available system memory;

-acceptanceRatioBreakValue: used to determine how precise the model should keep learning and when to stop. If smaller than 10e-5 it probably does an overtraning on the training data, which should be avoided;

featureType: type of features;

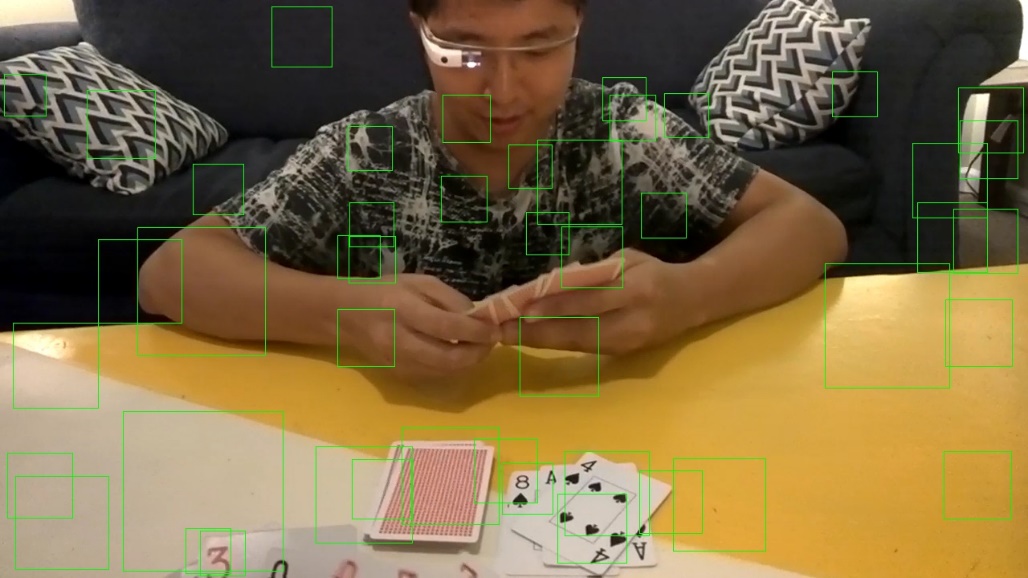
-w: width of training samples in pixels, it must have exactly the same value as used during training samples creation;

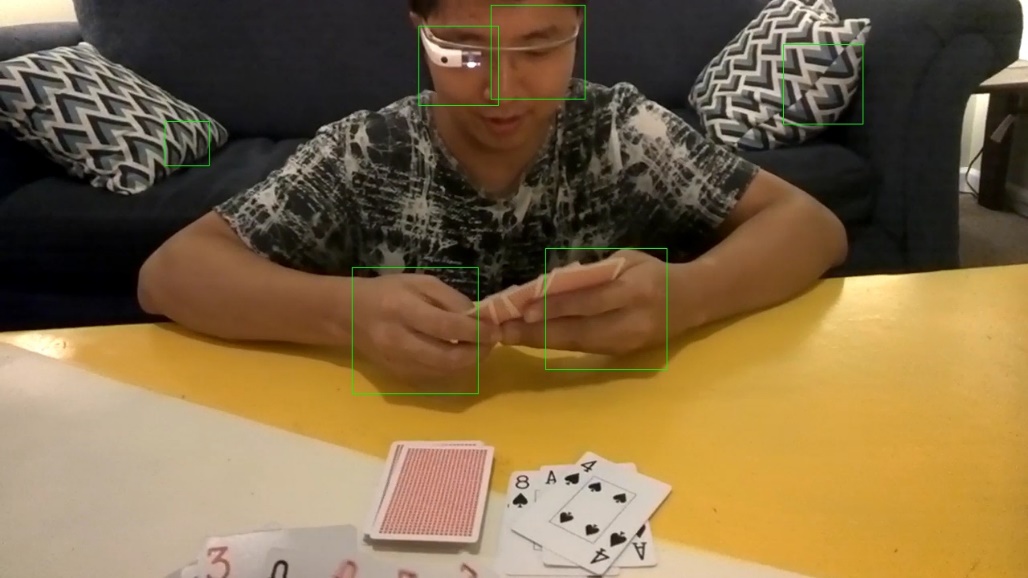
-h: height of training samples in pixels, it must have exactly the same value as used during training samples creation;

-minHitRate: minimal desired hit rate for each stage of the classifier (0.999);

-maxFalseAlarm: maximal desired false alarm for each stage of the classifier (0.3);

Here follows a couple of the algorithm output using different parameters for the training stage:





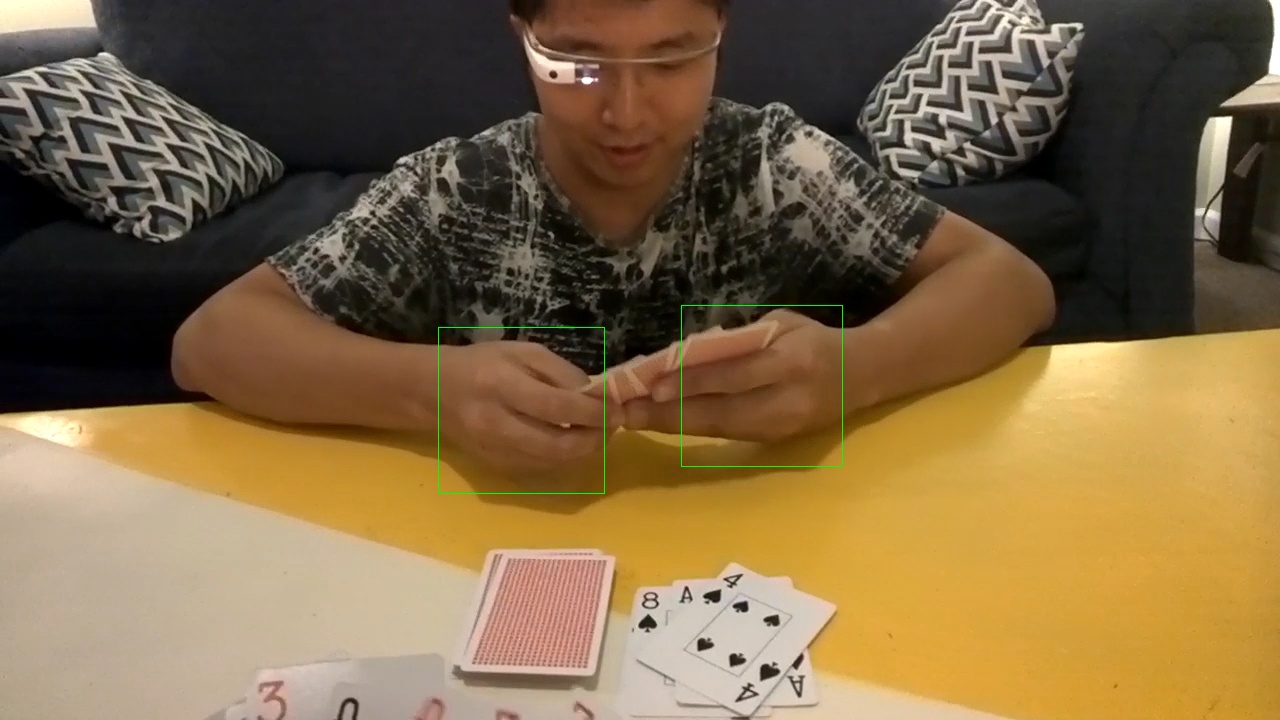
Both the images are obtained as the output of the cascade classifier on raw input data and, even if the second is much better from the first try, it can be easily seen that the performance is still not as the desired one.

Our idea at this point was to apply some pre-processing and postprocessing techniques, in particular for the pre-processing we tried to eliminate the background as much as possible with (Leo spiega la rimozione di Background) and the application of mean shift in order to group pixels based on their colours and smooth the image at the same time.



For the postprocessing we used a simple template matching in order to eliminate all the bounding boxes that didn’t contain our object (Leo se serve spiega velocemente il template matching).

After the application of this pre-processing and postprocessing we obtained the following results, which still has some major errors like eliminating some bounding boxes which has hands or leaving some bounding boxes that don’t contain hands, but in general are way better than the raw results of the cascade classifier only:



# Segmentation

For the second major part of the project, we were asked to generate a segmentation around only the hands of the given image. For this task we based our solution on the output of the first part, that is we applied our segmentation algorithm only inside the bounding boxes generated by the hand detection, which should contain only hands and some of the background.

For our segmentation algorithm we used two different approaches: fill flood algorithm and skin colour mask segmentation.

The second one, like the name suggest, is based on the colour skin: in particular, it takes the image, in our case only the bounding box, and applies a threshold function to keep only that part of the image that corresponds to the tonality of the skin colour. However, since usually hands in images have different illumination, it can deflect the actual colour of the skin. A solution which we found is to use a closing technique, which applies the erode and dilatate functions on the image, to keep also the part of the image that is close to the object having a skin colour. Even if this solution is not optimal, in some cases it gave as very good results.

# Results

In this part we show the numerical results that our algorithm obtained on the test image:

# Conclusions

What we can gather from the results is that the detection obtained using bosting of cascade weak classifiers has good performance while the hand is fully visible and with a constant illumination. Instead, when the hand is viewed from different point of view or with strong illumination changes the algorithm has more difficulties in detecting it.

The cause is probably since the sampling of the dataset was done manually and, as a matter of fact some bounding boxes used for the training part are slightly different from the ground truth bounding boxes. Moreover, the bounding boxes often contained part of the background because it’s doesn’t match a perfect geometrical form, that is in order to enclose the hole hand in the bounding box also some background needed to be taken in.

Other cause is given by the negative set, which we created by the application of a skin colour mask on the EgoHands dataset to erase the presence of hands. However, in many cases the hand was not fully covered, or even other part of the image were erased, which meant that it could be considerate as positive.