Computer Vision - Final project

Autonomous Driving: Road Sign Recognition

The aim of the project is the recognition of road signs, detecting where they are but also the class they belong to. It is requested to detect the “No parking”, “Speed limit” and “Warning” signs. For speed limit signs it is also needed to detect the maximum allowed speed.

Features of the application

The application was developed in C++ 2011 with the OpenCV 3.4.1 library. Although, the project requires to detect road signs only in static images, I decided to let the user decide if he wants to give a set of images as input or if he wants to detect road signs in real time by means of a camera.

When started from command-line prompt, the arguments needed are:

- If used in “real time” mode
  `road_sign_recognition.exe realtime <path_of_classifiers>`
- If used in “set of static images” mode
  `road_sign_recognition.exe realtime <path_of_classifiers> [<path_of_images>]`
  The path of images is optional because if not given as parameter it is then requested as input.

Once all parameters are set, the program starts the initialization creating an instance of RoadSignDetector. During the initialization of the detector an instance of SVMclassifier is created and all the files of the classifiers (whose position is given by the command-line argument path_of_classifiers) are loaded.

In both real time and static images cases, images are passed one-by-one to the detector.

An image is loaded into the detector with RoadSignDetector::loadImage(…) function. Then it is possible to start the pre-processing phase.

Pre-processing

The pre-processing is composed of a lot of phases.

Since the road signs that have to be classified can be only red or blue (or at least the external part is always red or blue), the idea is to consider as regions that could contain a road sign only the ones that have these two color components. Obviously, the illumination conditions can be really problematic. However, if we use the HSV (Hue Saturation Value) color space instead of RGB color space and we consider as indicator of color the Hue component, the illumination is not a problem. So, all pixels of the image were selected or not basing the choice on the Hue value (that for red component had to be in the intervals [0, 15] or [245, 255], for the blue component instead in the interval [100, 150]) but also on the Saturation value (>=50 for red and >=100 for blue) and a bit on the Value (>20). The parameters chosen for Saturation and Value have been necessary only to avoid false positives on detection.

The image where the red and blue pixels are searched is not directly the starting one. To have a better color detection I use an algorithm, that works with RGB color space, that makes respectively the red and the blue component enhanced. These enhanced images are those that are used for pixel searching.

The functions findRedSigns() and findBlueSigns() return masks, indicating only where the red and blue components have been found. They have to be applied to the original image with a bitwise AND (for the blue component only, I decided before to calculate the bitwise NAND between the blue mask and the red mask in order to avoid having pixels that are classified as both red and blue).

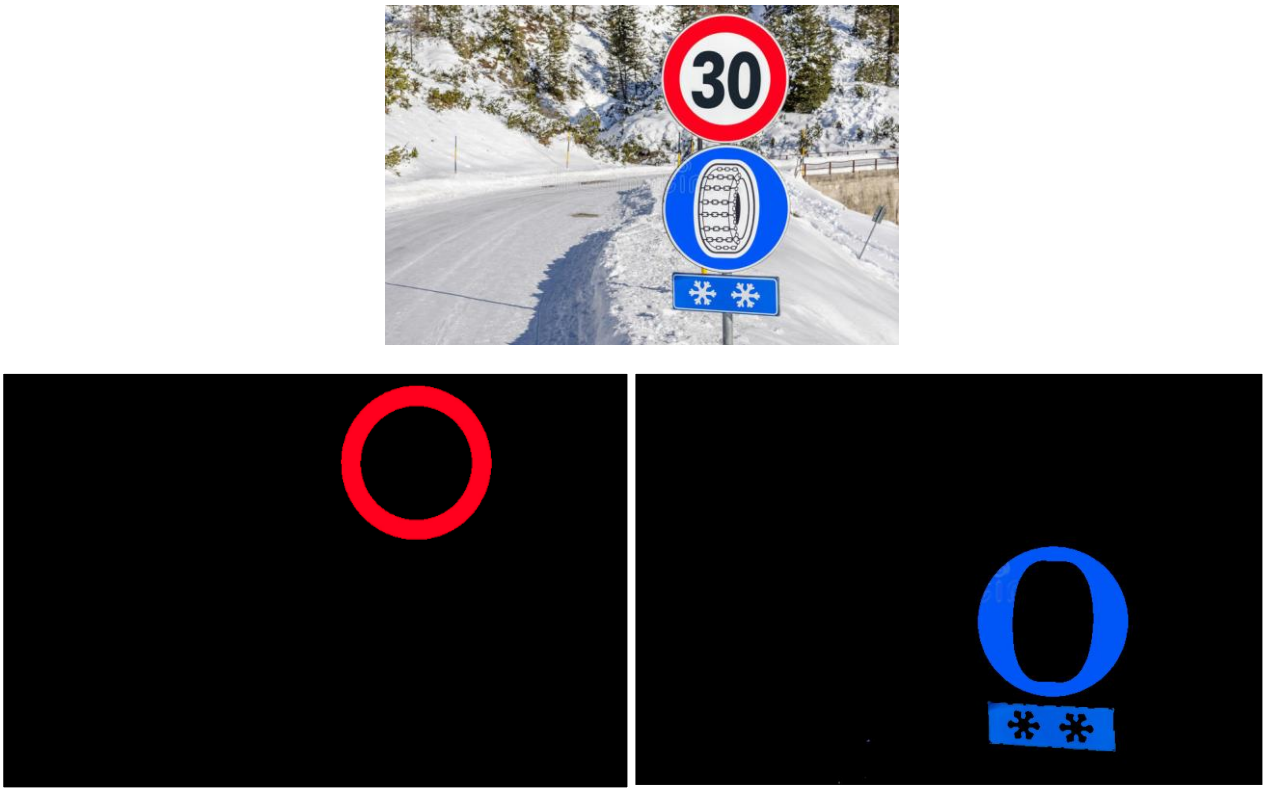

Figure 1 – Original image, red component and blue component

Next phases are computed for each one of the components found:

1. Deletion of isolated points using a Median filter.
2. Edge detection with Canny Edge detector, whose thresholds are calculated automatically with an algorithm that considers the mean value of the grayscale image. The parameters calculated have been demonstrated to be good enough in the majority of the situations.
3. Detection of contours from the image obtained with Canny.
4. Because some road signs could not be completely inside the image or the contours could be detected not perfectly because of a bad image (see Figure 2) or a not perfect Canny parameters optimization, I used the convexHull() function. Indeed, we know that all road signs shapes are convex, so it is correct to approximate each detected shape as a convex shape. The hulls obtained are drawn as white-filled regions in a cv::Mat.

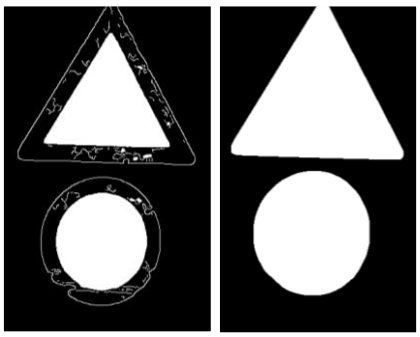

Figure 2 – Example of bad image where contours are detected not perfectly and result of convex approximation

5. The image drawn is considered as a mask. This mask is applied to the original image with a bitwise AND. In this way I have an image (cv::Mat masked) that contains only the parts of the original image where the first part of the pre-processing has decided that there could be road signs.

6. All the convex regions found in phase 4 are bounded with a rectangle to identify better their position. In this phase I obtain a lot of rectangles that are false positives, so I have to filter them out.
7. The filtering phase has two sub-phases:
   a. Firstly, all rectangles whose width or height is smaller than SIZE_THRESHOLD (after tuning, the best value chosen is 28) or whose aspect ratio is greater than ASPECT_RATIO_THRESHOLD (3/2 was chosen, not too strict in order to consider also some road signs that are not perfectly in front of the camera) are discarded.
   See Figure 3
   b. Next, all rectangles that are inside a bigger rectangle or that have an overlapping area with another rectangle that is greater than MAX_OVERLAPPED_AREA (30 was chosen) are discarded. Note that all signs have an outer shape and the same shape inside but if I consider the external one I maintain more information about the sign than considering the internal one. In fact, considering the external shape I consider both the information of the external region and the one of the internal one.
   See Figure 4

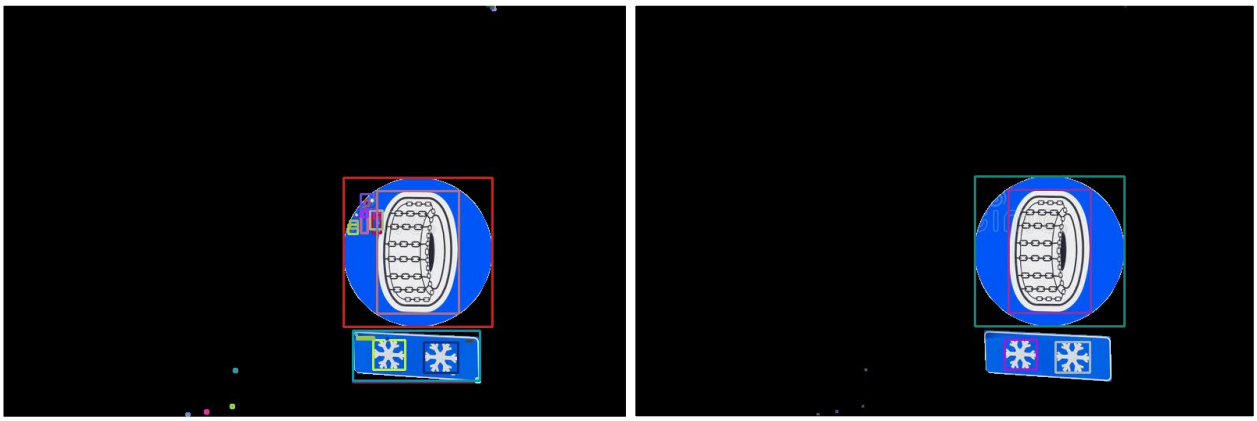

Figure 3 – Filtering of rectangles, detected (left) and result of the first sub-phase of filtering (right)

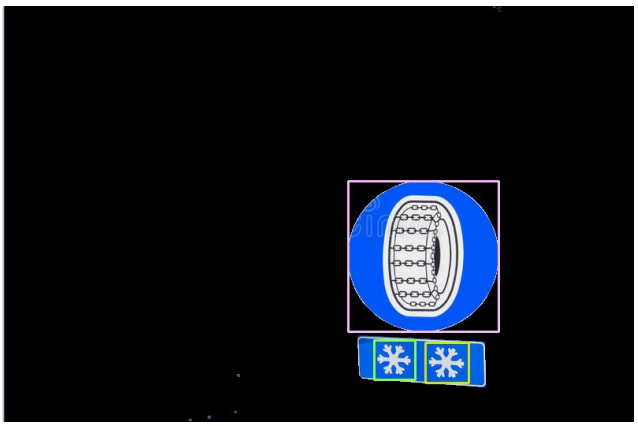

Figure 4 – Result of the second sub-phase of filtering

8. All rectangles that have passed the filtering phase are then resized. The increase of the size is done to avoid cutting the edges of the signs with too strict boundaries. This increase is inversely proportional to the current size of the bounding rectangle. In this way more information about the sign are maintained and then passed to the classifier.
9. Each rectangle obtained with phase 8 identifies a region of the source image, that can be extracted from it. Each rectangular region extracted is passed to the circles detection function.

10. The circles detection function searches for circles in the extracted image. After having obtained the thresholds for Canny edge detection with the same automatic calculation algorithm of phase 2, I use cv::HoughCircles (note that HoughCircles has a built-in Canny edge detector, so there is no need to perform an edge detection before) to obtain the circles. To avoid false positives (I know that ideally I should obtain only one detected circle if it is effectively present) I set the maximum between the width and the height of the considered image as value for the minimum distance between circles. The minimum radius considered is half the minimum between the width and the height of the image (minus 20 to let an error margin). The maximum radius is instead the minimum radius plus 50 (for the same reason). The function returns true only if at least one circle has been detected.
11. The last pre-processing phase consists on the creation of an instance of the class DetectedMat for each region obtained in phase 9. The constructor of DetectedMat has as input parameters the cv::Mat representing the rectangular region, a boolean value indicating whether the image has been detected by processing the red component image, a boolean value indicating whether the image has been detected by processing the blue component image, another boolean value indicating whether the image contains circles and a last boolean value that indicates if the image contains a triangle (currently not used).
12. All instances of DetectedMat created are inserted in a vector returned by the function preprocessing().

Classification

After the pre-processing phase, the classification is done using the function getClassificationResults(), that calls the other function classifyRoadSigns(). The last one takes each detected image from the vector just created and passes it as input parameter to the svmClassPrediction() function of the SVM classifier. It receives the predicted class of the road sign (or -1 if it is not recognized) and, when all images are computed, it returns to the first function the vector of corresponding labels in order to print them in the destination image close to the corresponding rectangle.

For the classification a Support Vector Machine with Linear Kernel and based on HOG (Histogram of Gradients) features is used. All functions related to the SVM are part of the SVMclassifier class.

The classifier is able to detect the following classes of road signs. The element number of the list corresponds to the index assigned to the classifier and the couples of information between brackets are those that have been used to help the classification phase (it will be better explained later).

0. Speed limit 10 (red, circle)
1. Speed limit 20 (red, circle)
2. Speed limit 30 (red, circle)
3. Speed limit 40 (red, circle)
4. Speed limit 50 (red, circle)
5. Speed limit 60 (red, circle)
6. Speed limit 70 (red, circle)
7. Speed limit 80 (red, circle)
8. Speed limit 90 (red, circle)
9. Speed limit 100 (red, circle)
10. Speed limit 120 (red, circle)
11. No parking (red, circle)
12. Snow chains (blue, circle)
13. Warning (red, not circle)

It was not possible to use a multi-class SVM because of false positives (in fact, with a unique classifier each detected road sign must belong to one of the classes, it is not possible to consider not recognized signs).

Moreover, with a multi-class SVM the distance from the separating hyperplane has no sense, it can not be used as an accuracy of the classification. So, the best choice has been the use of a classifier for each class of sign.

The svmClassPrediction() function takes a DetectedMat as input parameter. After checking if all 2-classes SVMs are trained correctly, the image is extracted from the DetectedMat object, converted to grayscale and resized to a fixed size (64x64) that is the size used also by the images in the training phase. The HOG descriptor is initialized and then it is computed from the image. What is obtained is a vector of float values representing the descriptor. All values are inserted in a cv::Mat in a single row, as in the descriptors Mat obtained during the training. Then the effective classification with the SVMs and the additional information carried by the DetectedMat is done.

For each classifier, the predicted result and the distance from the separating hyperplane is computed with the predict() function. The predicted result can be only 1 (indicating that the SVM has recognized the images as belonging to the class of the current classifier) or -1 (otherwise). If the predicted result is 1 the distance computed is negative, whereas if the predicted result is -1 the distance computed is positive. In both cases, the more the absolute value of the distance is greater the more the classifier is certain of its choice.

Thanks to these considerations, it is possible to use some policies to have a better classification. If the classification leads to multiple matches (more than one classifier has 1 as result), the class considered is the one associated to the highest distance from the hyperplane. If all the classifiers have as output -1, the class that is associated to the lowest distance from the separating hyperlane is considered as the good result, but only if this distance is lower than a RECOGNITION_THRESHOLD (set to 0.20).

However, these policies are considered only after the results of the classifier have been processed by a penalty method. Using the information of DetectedMat (each class has some specific properties for matching that are the ones indicated between brackets in the list of classes), it is possible to add a penalty to certainly wrong classifications (for example, if I am considering the classifier with index 2 (Speed limit 30), it is impossible to have the result 1 (recognized) if the DetectedMat object indicates that no circles have been detected during the pre-processing phase). The penalty is given by assigning as result -1 and ass distance from the hyperplane 100 (these assignments are independent from the response of the SVM classifier). Only a classifications that satisfies all the constraints is given untouched. The schema below shows this policy.

Detected with blue mask => classes 0-11 and 13 not possible

Detected with blue mask AND no circles => class 12 not possible

Detected with red mask => class 12 not possible

Detected with red mask AND has circles => class 13 not possible

Detected with red mask AND no circles => classes 0-11 not possible

Training

Before the recognition, it is needed to complete a training phase. The training phase has to be done only one time and it produces the files that are loaded when initializing the instance of SVMclassifier class.

The svmClassTraining() function receives as input the path of the folders that contain the images for training. Each folder has as name the index of the class. It is necessary to train a number of classes of classifiers, giving to each of them images of the considered class as positive samples (whose class label is 1, that means that these images belongs to the classifier class) and images of all other classes as negative

samples (whose class number is -1). Images are not passed directly to the SVM classifier, it is needed to pre-process them before. They are converted to grayscale and resized to 64x64. Then, they are passed to the compute() function of HOGDescriptor, obtaining the HOG descriptor associated to the image. Then, each computed descriptor is inserted in a new row of a cv::Mat object. After all images have been read and pre-processed, this object is passed as input parameter to the SVM classifier together with the list of corresponding labels (the i-th row of the training cv::Mat corresponds to the i-th row of the cv::Mat containing the labels). The SVM classifier training phase gives as output the file containing the training information that will be loaded by the SVM predictor.

HOG Features

The choice of using as descriptors of images the HOG features and not the directly the pixels of the images (maybe vectorized) is due to the poor results that this second approach lead to. Indeed, with the second approach the importance is given to the intensity value of each image pixel, so that if, for example, the classifier is trained only with light images of a road sign, when it has to recognize the same sign with low light conditions the result is very bad. The HOG features, instead, consider the gradients and so they give more importance to the shape of the road sign, with the result that it is less sensitive to variations of the illumination conditions.

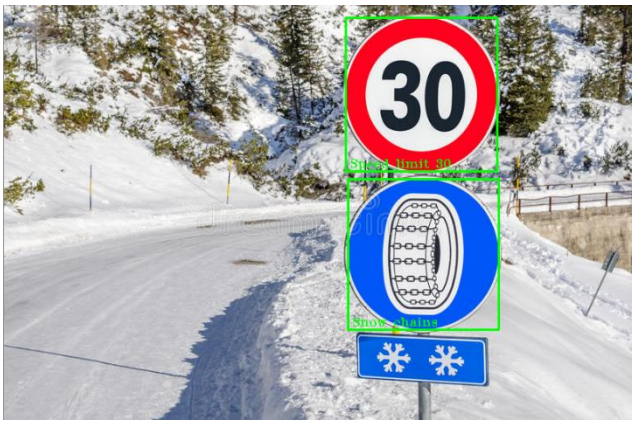

Figure 5 – Result of the recognition

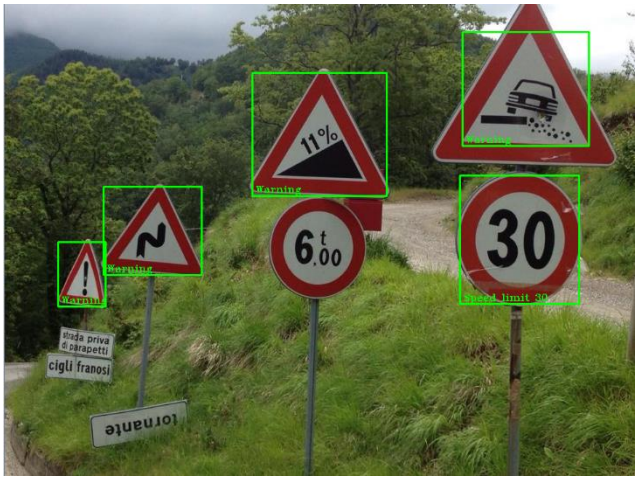

Figure 6 – Another example of result of the recognition

Notes

To obtain information about rectangles detected, number of circles detected, raw classification results (already penalized) and images obtained during the pre-processing phase, it is possible to use the function setDebugMode(true) of RoadSignDetector class. The default value is false.

All the images presented in this report can be obtained it this way.