Automatic License Plate Reading

Nicola Dainese

30 June 2019

1 Introduction

The aim of this project is the localization of licence plates in the input images and the following reading of the various characters in the licence plate. The workflow of the algorithm can be broke up in three major parts: first the image is preprocessed, obtaining a denoised binary image, then the plate is recognized using a contour detection combined with a clustering technique and finally a pre-trained convolutional neural network (CNN) is invoked to predict the alphanumeric characters in the plate. Most of the algorithm is implemented in C++, using in particular the OpenCV library, but the last part, concerning the recognition of the character images, is implemented in Python, using the PyTorch library in order to build and train the convolutional neural network.

2 Preprocessing

The preprocessing works as follows:

1. The color image is converted to a grayscale one;
2. A bilateral filter is applied to denoise the image while preserving the edges;
3. The OpenCV adaptiveThreshold is applied, obtaining a binary image.

In particular for the bilateral filter the following parameters are used:

1. Filter kernel of 7×7;
2. $\sigma_{range} = \sigma_{space} = 80$.

This values are set in order to have a small window of action but a decisive local smoothing of the image.

Instead, for the adaptiveThreshold, the parameters are the following:

1. adaptiveMethod = $ADAPTIVE\_THRESH\_GAUSSIAN\_C$;
2. thresholdType = $THRESH\_BINARY\_INV$;
3. blockSize = 19;
4. C = 9;

The idea is that a pixel is set to 0 only if its original intensity is greater than the one of its neighbors (those inside the blockSize, weighted with a gaussian) minus an offset C.

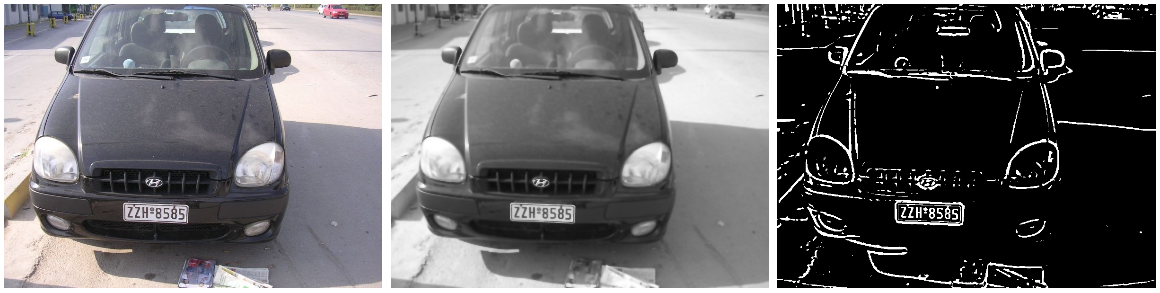


Figure 1: Original (left), grayscale blurred (center) and thresholded binary (right) images.

3 Plate detection

The main ideas used for recognize the plate among all the patterns of the image are the following:

1. The plate has a *height/width* ratio that is constant with small variations among all countries (here I do not consider the case of plates with two rows of characters); I assume that ratio to be between 0.18 and 0.40 to account also for distortions given by perspective.

2. The characters in the plate have themselves some characteristic range of ratios (this time *width/height*), assumed between 0.15 and 1.

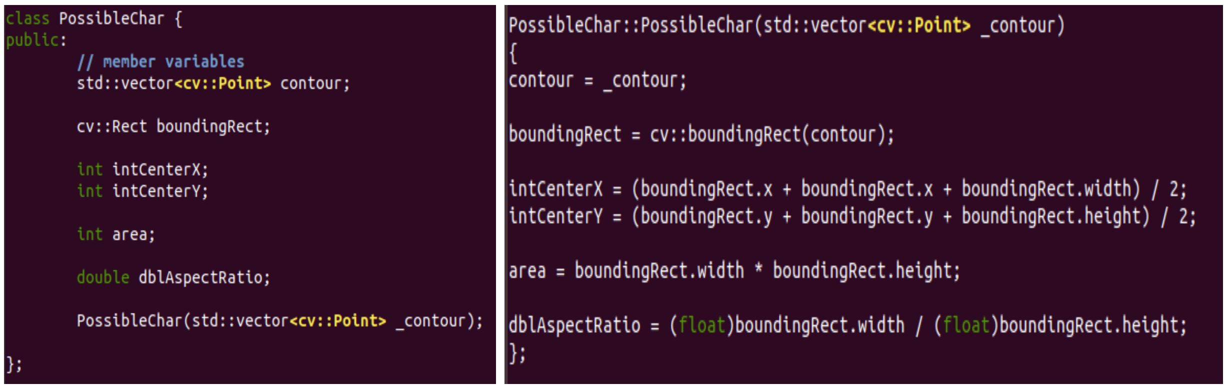
3. The characters are one near to the other; in particular the distance between the centers of two characters has the same order of magnitude than the sum of their heights.

All this considerations led me to implement an algorithm that first recognizes all the contours in the image, then selects only those that have ratios and dimensions (mainly minimal sizes to sort out small noisy contours) compatible with characters; afterwards it clusters them together using an adaptive clustering that doesn't need to know either the characteristic scale nor the number of clusters in the image. Finally it selects only those clusters that contain at least 6 characters (this allows for a minimum of tolerance in the detection procedure) and are in the range of allowed plate ratios. Typically this holds a single plate ($\approx 95\%$ of the times, very rare false positive cases, more common failures in detection due to lack of illumination and other issues), but has the potential to detect multiple plates if present and solvable in terms of resolution).

3.1 C++ implementation

In this subsection I explain more in depth the concrete implementation in C++ of all the steps needed to detect the plate.

The first part of the plate detection is done with the module CharDetection, where I implemented a function detectChar that takes as input the binary image and returns a vector containing all the possible characters still to be clustered (along with a drawing of them). Inside this function, first of all, to detect the contours I use OpenCV Canny edge detector[1] and findContours[2] functions. Then I use the found contours to initialize instances of the class PossibleChar (snippet of the code in Fig 2) and I filter them with checkIfPossibleChar to impose the ratios and dimensions mentioned above. I also filter them with doIntersect, in order to keep only the PossibleChar with largest area when more of them intersect.



```
class PossibleChar {
public:
	// member variables
	std::vector<cv::Point> contour;

	cv::Rect boundingRect;

	int intCenterX;
	int intCenterY;

	int area;

	double dblAspectRatio;

	PossibleChar(std::vector<cv::Point> _contour);

};
```

```
PossibleChar::PossibleChar(std::vector<cv::Point> _contour)
{
contour = _contour;

boundingRect = cv::boundingRect(contour);

intCenterX = (boundingRect.x + boundingRect.x + boundingRect.width) / 2;
intCenterY = (boundingRect.y + boundingRect.y + boundingRect.height) / 2;

area = boundingRect.width * boundingRect.height;

dblAspectRatio = (float)boundingRect.width / (float)boundingRect.height;
};
```

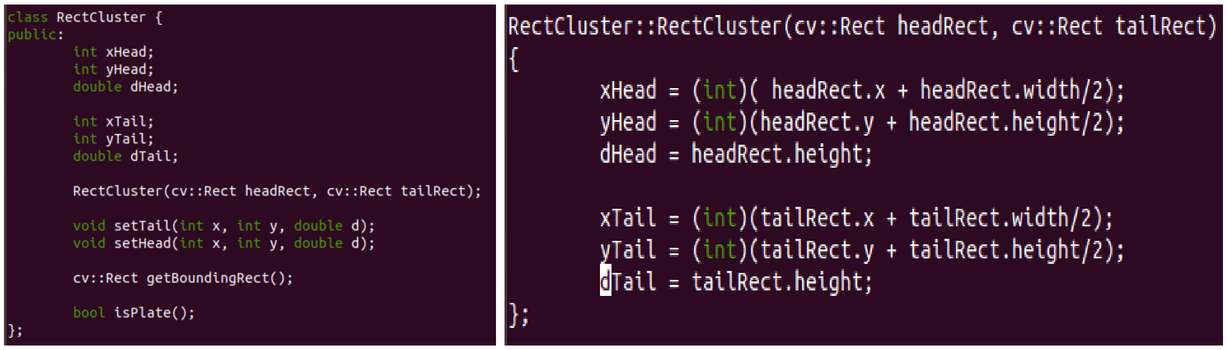
Figure 2: Code snippet for header and implementation of PossibleChar class.

At this point I have a first ensemble of PossibleChar, usually containing around 100 elements, that I pass to the Clustering module. The high level function of this module is selectPlates, that takes this vector of PossibleChar and returns a vector of rectangles (cv::Rect), that are already the final guesses of the contour of the plate (usually,

[1]https://en.wikipedia.org/wiki/Canny_edge_detector

[2]Suzuki, S. and Abe, K., Topological Structural Analysis of Digitized Binary Images by Border Following. CVGIP 30 1, pp 32-46 (1985)

as already said, there is only one element inside that vector). Inside this function first I initialize instances of the class RectCluster (snippet of the code in Fig 3), that are basically the union of a head rectangle and a tail rectangle (where head and tail are intended as the leftmost and rightmost rectangle part of the cluster); during the initialization head and tail coincide, because for every cluster I use a single PossibleChar to define it.



```
class RectCluster {
public:
        int xHead;
        int yHead;
        double dHead;

        int xTail;
        int yTail;
        double dTail;

        RectCluster(cv::Rect headRect, cv::Rect tailRect);

        void setTail(int x, int y, double d);
        void setHead(int x, int y, double d);

        cv::Rect getBoundingRect();

        bool isPlate();
};
```

```
RectCluster::RectCluster(cv::Rect headRect, cv::Rect tailRect)
{
        xHead = (int)( headRect.x + headRect.width/2);
        yHead = (int)(headRect.y + headRect.height/2);
        dHead = headRect.height;

        xTail = (int)(tailRect.x + tailRect.width/2);
        yTail = (int)(tailRect.y + tailRect.height/2);
        dTail = tailRect.height;
};
```

Figure 3: Code snippet for header and implementation of RectCluster class.

Then I sort the vector of all RectCluster with sortVecOfCluster, so that the first element is the leftmost one and last is the rightmost and I merge the adjacent clusters if the sum of their radii (taken equal to their heights) multiplied by ALPHA (a constant that can modify the notion of "near" in the clustering procedure, set to 0.9) is greater than the euclidean distance between the center of the two clusters. Once the merging procedure is done, I have a vector of RectCluster and each of them has a bounding rectangle obtained with the method getBoundingRect. Imposing that the bounding rectangle has proportions inside a certain range and that at least 6 characters are completely contained inside it, we remain with only one element, that is the rectangle containing the plate. In Fig 5 is shown a typical plate detection procedure.

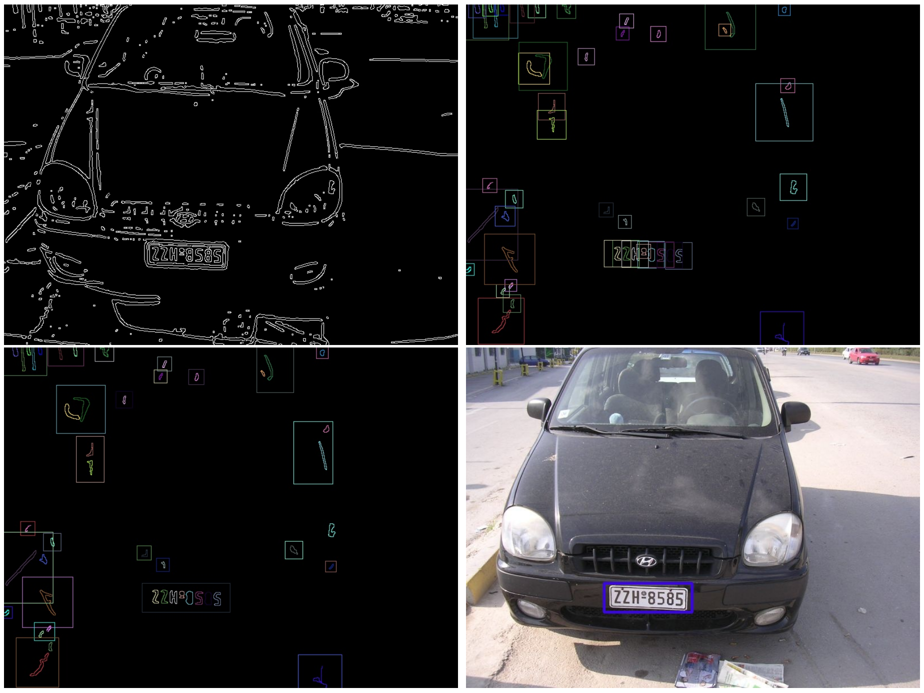


Figure 4: Canny edge detector output (top left), contours of PossibleChars with RectCluster associated (in which for the initialization the head rectangle coincides with the tail one) (top right), same image but with RectCluster after the clustering algorithm is applied (bottom left) and original image with plate contoured by the RectCluster that has the proportions inside the right interval and contains at least 6 PossibleChars.

4 Characters recognition

Working with the original image cropped around the plate, first I detect the PossibleChar as before (the only difference is that this time I skip the blurring procedure), then try to filter out possible false positive characters, requiring the height of each character to be inside a range given by the mean of the heights of the PossibleChar in the plate plus or minus 5 pixels and a similar requirement for the y coordinate of the centers. This is implemented with the functions filterByHeightVar and filterByAllignment. In this way I obtain only aligned characters of the same height. Then I crop the image around each of them, apply a new adaptive threshold, pad them with 2 white pixels for each side and resize them to images of 32×32. At this point I use a pre-trained convolutional neural network (CNN) to predict the alphanumeric character correspondent to each of those images and assemble the result, to show the original image with the recognized characters over the plate (see Fig 6).

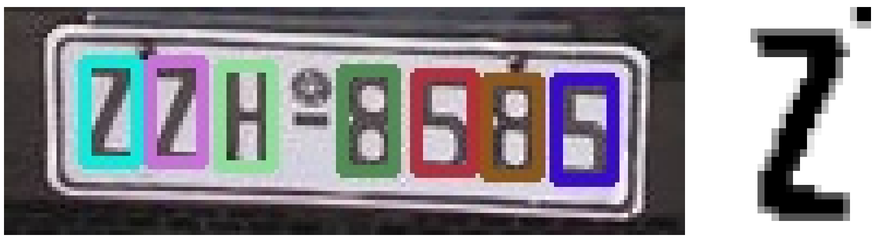


Figure 5: Plate with all the detected characters highlighted (left) and the first character image as it is written in input to the CNN.

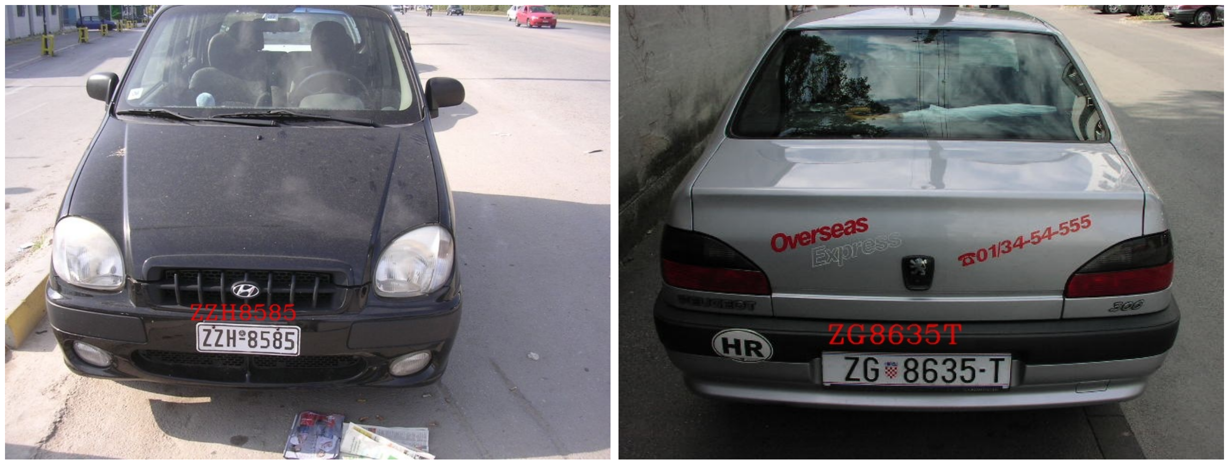


Figure 6: Two examples of exact license plate recognition.

4.1 Convolutional Neural Network

In this subsection I briefly discuss the structure of the CNN and how I trained it. As mentioned in the introduction, I developed and trained the CNN in Python, using the library PyTorch. The net has the following structure:

1. Convolutional layer with a kernel of 3×3 and 6 output channels;
2. Pooling layer with a kernel of 2×2 and max-pooling;
3. Convolutional layer with a kernel of 3×3 and 12 output channels;
4. Pooling layer with a kernel of 2×2 and max-pooling;
5. Affine transformation from 768 variables at the end of the second pooling to 256;

6. Fully connected layer of 256 input features and 64 output neurons;
7. Fully connected layer of 64 input features and 36 output neurons (as many as the number of alphanumeric characters).

To train the net I used a dataset of alphanumeric synthesised characters from computer fonts called Chars74K[3]. Since the images have resolution of 128×128 I rescaled them to 32×32, that is the resolution similar to the images of the characters in the plate (this is also the reason why I resized them to this resolution). Some fonts were completely unrealistic for plate characters, so I excluded them from the dataset. Finished the training, I saved the weights of the net, so that a separate script can just load them and be ready to cast predictions.

5 Common issues

The issues that can happen with this algorithm in order of appearance are:

1. No cluster, more clusters or the wrong cluster are recognized as plate;
2. Not all the characters in the plate are detected;
3. Some noisy patterns are recognized as characters inside the plate;
4. Some characters are fragmented in two and an attempt to read the two parts separately is made;
5. The character images are misclassified.

The failure in detecting the plate is rare and usually depends on the bad conditions in which the image is taken (e.g. dark images, ruined or too small plates, etc...). Instead the false positive events in this case are usually due to vertical patterns in the grid of the radiator (e.g. in many BMW models).

The non-detection of the characters in a plate is more common and can be due to every factor that during the thresholding procedure either breaks the character in two or connects it to the plate border. This usually leads to proportions that are not that of a character and moreover are completely different from that of the other characters detected, resulting in the elimination of partially detected characters. The main factor of false positive events instead is due to the contrast between the borders of the plate and the rest of the car, and they usually have I-shaped forms.

Finally also the character recognition has severe limits, because many characters are almost interchangeable, like I-1, 2-Z, 8-B and many more. Given the fact that the conditions in which the training and the testing are done (i.e. one with computer fonts, the other with binary images coming from camera images), it makes little sense to confront the metrics obtained in the training with those in the real application, that are always lower. In my case I didn't evaluate extensively the application ot the algorithm because I lacked a framework for testing it efficiently, but results over around 100 images suggest that on average more than 1 character in 7 is misclassified.

6 Conclusion

In this project I shown that a methodology for automatic license plate recognition based on edge detection and clustering for the detection part and deep learning for the recognition part can effectively work. At the present time, the loading of the weights seems to be the bottleneck of the algorithm in terms of speed, thus it takes some seconds to run and cannot be performed efficiently in real time, but could be modified to classify multiple images with just one loading of the weights, probably gaining an order 10 in speed. The second, more severe, issue is that of the accuracy of the recognition, that is poor. To become more reliable probably a deep learning implementation of the character detection with a sliding window in the plate could improve the detection performances and a finer preprocessing united with a training dataset with fonts more similar to the one used in the plates could help the recognition part. Of course also the original quality and resolution of the images influence greatly the results, but I consider them external factors, that do not depend on the algorithm. However, if somehow we were granted a minimum resolution for the characters in the plate (greater than 32×32), it would help in using more information as input to the CNN and obviously implying better results.

[3]http://www.ee.surrey.ac.uk/CVSSP/demos/chars74k/