Computer Vision: THE BASIS OF A SELF-DRIVING SYSTEM

Eduardo Morgado, Emanuel Tomé

Description of computer vision steps and procedures taken during development, although in its first version, of a self-driving system in the simulation engine CARLA.

***Index Terms*—computer vision, detection, line detection, self-driving, self-steering**

I. INTRODUCTION

CONSIDERING the difficulty of this project, we divided it into several separate smaller projects, each building on the previous one. In this first version we only managed to successfully develop and integrate with CARLA[1] the first three stages (1, 2 and 3). This were the planned stages:

1) Lane Detection
2) Self-Driving (Steering) using Lane Detection
3) Traffic Signs Detection
4) Self-driving taking into account lanes and signs
5) Obstacle[2] Detection
6) Self-driving taking into account lanes, signs and obstacles

II. LANE DETECTION

Initially, our goal was to build a simple lane detection and once we had good enough results, move to a more elaborate and robust detection[3]. As such, we began by implementing the **Hough Line Detection** in a notebook.

A. Line Detection in Hough Space

The main objective of this method is to **identify straight lines**, it would later be improved to work with other shapes, however we only implemented its first version.

This paper was submitted on the 18th of January as part of a support paper for the subject of Computer Vision, CC4016 at Faculdade de Ciências da Universidade do Porto.

Eduardo Morgado is a student at Faculdade de Ciências da Universidade do Porto with student number up201703990. Currently, he is enrolled in the 4th year of Mestrado Integrado em Engenharia de Redes e Sistemas Informáticos (e-mail: up201703990@fc.up.pt).

Emanuel Tomé is a student at Faculdade de Ciências da Universidade do Porto with student number up200702634. Currently, he is enrolled in the 1st year of Mestrado em Ciência de Dados (e-mail: up200702634@fc.up.pt).

[1]CARLA is an open-source world simulator designed to serve as a self-driving playground, we chose to use this system to more easily test our detection system as well as incorporate the detection to take actions such as driving and steering the car, one other reason to use CARLA was the fact that we could record our driving, this would facilitate the generation of train data to machine learning driving models on later stages.

[2]With a special focus on people and other cars

[3]Our end goal for lane detection was to detect using a Spatial CNN or instance based segmentation for line detection

Since most of the times, a lane would be a straight line we decided this method would be a suitable option to perform lane detection. For this method to work the images should be **binary** but, since we need to search for lines in the color space we had to apply a **grayscale filter** to the image first, only after that were we able to perform other CV methods to detect edges (sobel filter).

A straight line can be represented as a tuple *(a,b)* where *a* is its slope and *b* its intercept, this is also true for the polar space, where a line can be represented by the tuple (φ, θ) where φ is the shortest distance from the origin to the line and θ is the angle between the x-axis and the distance line.

Using the polar space we can describe all types of line operations. For example, in Figure 1, its equation for every x,y point is $\varphi = xcos(\theta) + ysin(\theta)$.

Not only that but the polar representation allows us to represent the **line as a point in the Hough Space** (Figure 2).

If we consider a complex enough image and we pick a point in the Cartesian coordinate system it's very probable that many lines pass through that point, where each line would have different *(a,b)* parameters which would result in several points in the Hough Space.

All that's left to do is **find an equation for the polynomial that best describes all this points**. If we keep adding more intersecting lines this would result in a **sinusoid** in the Hough Space so we reached the following conclusion, **for a fixed *(x,y)* parameters representing a point in an image, if we consider all its possible values/orientation of θ in a certain range, we obtain φ values forming a sinusoid** (Figure 3).

This at first sounded a bit confusing, after researching further into the subject it became clear that this was the foundation to finding lines in a matrix of pixels. If we draw several points, forming a line/lane in the image space, we observe that we obtain a bunch of sinusoids in the Hough Space, all intersection at one point (Figure 4).

This proves crucial in our task since, to find a lane, we should **find intersections in the Hough Space**.

Now that we reached a way of finding lines through the Hough Space we need to prepare the image (preprocess) to feed it to the Hough line detection, this is where our edge detection pipeline comes into play.

B. *Edge Detection Pipeline*

As stated before, to detect lines in the Hough Space the image needs to be binary, as such we always need to convert the input image to grayscale. The Hough Line detection will detect try to detect any straight line however small it may be, in order to reduce the error of our algorithm we need to remove all unnecessary elements that might produce false lines, as such we need to only consider ”hard lines”, meaning, areas in the image matrix with a great difference in intensity, this lead us to implement a **Sobel filter** for edge detection. There are several algorithms that make use of a Sobel filter to detect lines we chose the **Canny edge detector** which is a multi-stage algorithm optimized for fast real-time edge detection and has the following steps:

1) Apply a Gaussian filter to smooth the image in order to remove noise
2) Find the intensity gradients of the image to detect orientation of edges through a Sobel fiter
3) Application of a non-maximum suppression algorithm to sharpen edges
4) Perform Hysteresis thresholding to analyse less intensive pixels

We also segmented the resulting preprocessed image since all we were interested was the lanes and this tend to be in the lower part of the image. We decided to segment the image forming a trapeze since we don’t want to have too much of a front view and we need to open the sides a little to try and detect curves.

C. *Gaussian Filter*

The Gaussian filter (smoothing operator) is a 2D convolution operator used to blur/smooth images and remove noise similar to the mean filter but using a different kernel representing the shape of a Gaussian curve. The main idea is to use the distribution as a **point-spread function through convolution**. In our work we used a 5x5 kernel.

D. *Sobel filter*

The Sobel operator is an **approximation to a derivative of an image** separated in the x and y axis. We used a 3x3 kernel (one dimension for each x and y direction). The sobel operator tries to find the **amount of difference by placing the gradient matrix over each pixel of the image**.

E. *Non-Maximum Suppression*

This step will essentially sharpen the edges where, for each pixel we will check of its value is a local minimum in which case we test for the next pixel otherwise it is set to 0/suppressed.

F. *Hysteresis threshold*

Here, we will simply apply a threshold filter to lower intensity pixels to see if they represent an edge or noise, if the pixel has an intensity gradient higher than our upper limit, it is an edge if it is lower than the lower limit it is discarded. For pixels within the threshold if the adjacent pixels are above the upper limit they too are considered edges, otherwise they are discarded.

G. *Canny Detector Results*

Once we tested all the steps in separate scripts we moved into CARLA and started the integration of lane detection. Figure 5 shows our control window for CARLA with our custom HUD parameters for each type of action (lane detection, lane auto steer and traffic sign detection). Figure 6 shows the CARLA input signal for the front car sensor after being fed to our Canny edge detection algorithm.

As stated previously, we were only interested in detecting lines as such we segmented the filtered image so as to not create a detection error, Figure 7 show the segmented image.

H. *Hough Line Detection Results and Steering Angle Calculation*

After generating the segmented edge detection image we fed it to the Hough algorithm that returned a series of lines.

By **looking at the slope** we were able to identify and group lines that belonged on the left or right side. This was possible due to our segmentation, by only considering an area where we know most of will represent a line, and by considering the sensor’s center axis as a base line, we can compare lines in terms of its parameters (slope and intercept) thus grouping them.

After grouping lines into left and right lines/lanes and by once again considering that most of the image (lines) would represent a lane we could average both side to get the average line on the left side, representing the approximated left line, and get the average line on the right, representing the approximated line.

Figure 11 shows the detected lines in blue.

The main purpose for detecting lanes was to calculate the **required steering angle** so that we could control and move the car autonomously, as such we first needed to calculate the steering angle.

In order to calculate the angle we centered the sensor and from the detected lanes[4] we generated a center heading line from the offsets to the lanes. Doing this it was possible to calculate the angle that the car should turn in order follow the heading line.

[4]We supported the cases where two lanes are detected and where only one is detected.

After calculating the steering angle[5] (in degrees) we needed to convert it into CARLA's [-1,1] scale, where if our angle was in [-1,0[the car should turn left, if the angle was in]0,1] the car should turn right and if the angle was 0 the car you go straight forward. To convert the angles we used $-cos(x)$ as our encoding function.

One last improvement was to reduce angle fluctuation where sudden changes in the direction of the heading line would cause the car to lose control, to do this we build a control memory that would record the last n angles and would apply a normalization filter to the new angle, ensuring that changes occurred slowly[6].

I. *Observations and Next Steps*

As stated before, Hough line detection is effective on straight lines and through testing we noticed that this was true.

For the tests where the car was driving in the highway, where turns have a slow curvature angle the car was able to follow the turn, provided that its speed was not too big, Figure 9. The issues occurred when the car was driving inside a city or exiting/entering the highway where we had a sharp turn and the model just couldn't detect the lane, Figure 10.

Another importance observation from our tests was the fact that, since we where averaging lines on the left side and right side, any change from the Canny segmented imaged would result in a change of the lines, this would cause the model to flicker, spatially during rain where there would be constant line flickers since the edges of water droplets would be detected.

One final important observation were the difference in a lane where both lines are continuous and another where one is broken, For the first case the model would behave almost perfectly (considering the previous cases). For the second, every time a broken line ended the heading an thus the angle would flicker until another broken line was detected.

This issues forced us to consider another approach that would eliminate most of the noise that couldn't be eliminated by the Canny edge detector. In a future implementation we would move into **Lane detection with Instance Segmentation** where it would eliminate most of our issues. It would unite broken lines into a continuous line, it would eliminate environment flicker and would make fitting a polynomial into the curve more simple. This could also be built upon in a CNN to automatically detect a lane. For reference, this article shows just that.

[5]Every angle would be in the range of $[0,\pi]$

[6]This approach improved the stability of the car at high speeds but it was hindrance if it was required to take a sharp turn

III. Traffic sign detection

We intended to build a traffic sign detector using deep learning and transfer learning. However, to do so we started to review fundamental background concepts and components of object detection pipeline. Then we build a Region Based Convolutional Neural Network (R-CNN) using the German Traffic Signs Dataset [1], [2].

A. *Fundamental background concepts*

- **Intersection over Union (IoU)**: IoU is an evaluation metric used to measure the accuracy of an object detector. It is defined by:

$$IoU = \frac{Area\ of\ Overlap}{Area\ of\ Union}$$

  where *Area of Overlap* is the overlapped area between the the ground-truth bonding box and the predicted bounding box from the model; and the *Area of Union* is the area encompassed by both the ground-truth bounding box and the predicted bounding box. For building our dataset using selective search, we set a minimum IoU of 0.7. However an IoU score greater than 0.5 is usually considered a good threshold.
- **Image pyramids**: An image pyramid is multi-scale representation of an image, which allows to find objects in images at different scales. The image at its original scale is at the bottom of the pyramid and at each subsequent layer the image is resized and smooth using Gaussian blurring. The image is usually sub-sampled until some stopping criterion is met, such as a minimum size.
- **Sliding windows**: A sliding window is a fixed-size rectangle that slides from left-to-right and top-to-bottom within an image. Each sliding window is usually passed through a image classifier, such as a SVM or CNN, to obtain the output predictions. Combined with image pyramids, it is possible to localize objects at different locations for input images of multiple scales.
- **Non-maxima supression (NMS)**: Usually the object detector will produce multiple and overlapping bounding boxes for the same object. Therefore, those overlapping bonding boxes should be collapsed. This is usually made using NMS which collapses weak, overlapping bounding boxes in favor of the more confident one.
- **Selective Search**: Selective search [3] is an algorithm from the family of region proposal algorithms which are used to replace the traditional method of object detection using image pyramids and sliding windows. Region proposal algorithms tend to be far more efficient than the traditional object detection techniques, since fewer individual regions of interest (ROIs) are examined. Selective search works by over-segmenting an image using a superpixel algorithm, merging superpixels

to find regions of an image that could contain an object. Selective search merge superpixels in a hierachical fashion based on similarity measures such as color, texture, size and shape similarities.

B. Training dataset

The dataset used for training is the German Traffic Signs Dataset [1], [2]. The dataset used for traffic sign detection [1] contains:

- 900 images of shape (800,1380,3).
- Traffic signs in the images varying from (16x16) to (128x128).
- Images containing zero to six traffic signs.
- Traffic signs that may appear in every perspective and under every lighting condition.

C. Region Based Convolutional Neural Network (R-CNN)

Our implementation of the R-CNN for traffic sign detection closely followed this tutorial. The main steps to build and R-CNN are:

1) Build an object detection dataset using Selective Search

For each of the 900 input images, we start by running selective search in orther to extract the region proposals. For each of them, the IoU is computed. If the IoU is bigger than 0.7, the region proposal is saved in the folder ”Signal”. Otherwise it is saved in the folder ”no signal”. Examples of signal and non-signal region proposals can be found in Figure 12.

2) Train a Convolutional Neural Network (CNN) model on the dataset built in the previous step for object detection

In order to achieve better results, we used transfer learning. Namely, we fine-tuned the MobileNet V2 CNN which is pre-trained on the 1000-class ImageNet dataset. Basically, we took the MobileNet, dropped its fully-connect (FC) head, constructed an new FC head and append it to the previous trained convolutional layers of the MobileNet. Before training, we frozen the layers taken from the MobileNet in order to just change the parameters of the our new FC head. For training the CNN, we used a batch size of 20 and 50 epochs. It should also be mentioned that we also made data augmentation during the training. An accuracy of 0.98 was obtained. The classification report is shown in Table I

TABLE I
CLASSIFICATION REPORT

Class	Precision	Recall	F1-score	Support
No signal	0.99	0.98	0.98	400
Signal	0.98	0.99	0.98	360
Accuracy			0.98	760

3) Use the CNN trained in step 2 to classify each region proposed via Selective Search

We are now able to do signal detection in any input image using our CNN trained in the previous step. After doing selective search on the input image, we can feed our model in the proposed regions that could contain a signal. For illustrative proposes, in Figure 14 are shown an input image using our R-CNN.

4) Apply Non-Maxima Supression (NMS) to suppress overlapping bounding boxes and return final object detection results

After our R-CNN classify the proposed regions by the selective search, we should now apply Non-Maxima Supression (NMS) in order to merge the boxes that refer to the same signal. In Figure 15 are shown the output after applying NMS to the obtained output from the previous step.

IV. Traffic sign classification

For the traffic sign detection our goal was to detect a region in the image of a traffic sign. After doing that we would run the detected signal in a CNN to classify the type of signal that was detected to them pass that info to the CARLA controller.

This section was left to complete for this current version.

V. Conclusions and Future Work

During this project our intentions were to build a basic self-driving system capable of steering taking into account lanes, traffic signs conditions and obstacles (avoiding people and other objects in a direct path to our car). From the start the project was very ambitious and we are satisfied with the results given the amount of time we had and the amount of knowledge we had. Most of the CV methods we implemented made use of high end libraries such as opencv and tensorflow, simply because it would take us much more time to develop our own algorithms and their performance would be lower than the curated libraries.

For the next steps of this project we would like to implement the missing steps as well as replacing the current steps by more efficient algorithms (instance segmentation/region proposal algorithms and CNN lane detection, and an RCNN capable of detection and classifying a signal).

One other improvement would be to use a LIDAR sensor that is a 3D sensor, this way we would more easily detect an object and thus act accordingly.

In case you wish to take a look at our project and code, you can find it here.

References

[1] Sebastian Houben, Johannes Stallkamp, Jan Salmen, Marc Schlipsing, and Christian Igel. Detection of traffic signs in real-world images: The German Traffic Sign Detection Benchmark. In *International Joint Conference on Neural Networks*, number 1288, 2013.

[2] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel. Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition. *Neural Networks*, (0):–, 2012.

[3] J R R Uijlings, K E A van de Sande, T Gevers, and A W M Smeulders. Selective Search for Object Recognition. *International Journal of Computer Vision*, 104(2):154–171, 2013.

Appendix

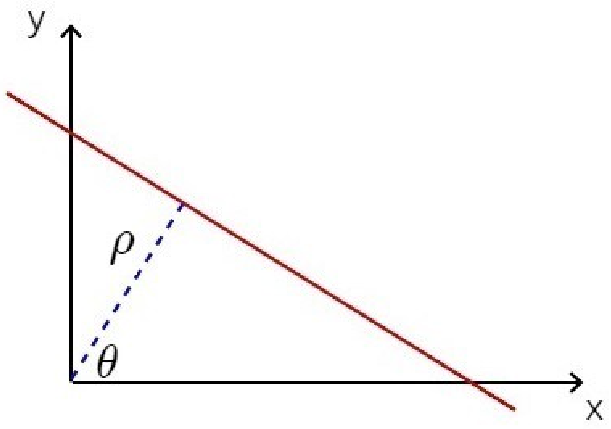


Fig. 1. Example of a line described in polar space

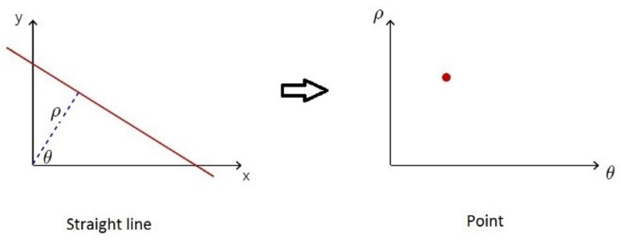


Fig. 2. Line representation in Hough Space

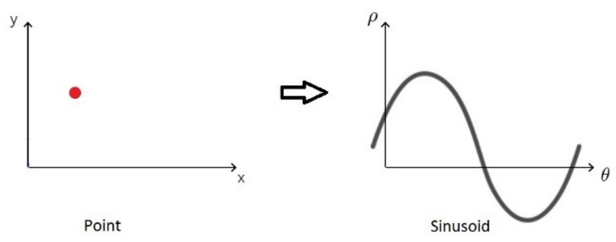


Fig. 3. Combination of several intersecting points to form a sinusoid in Hough Space

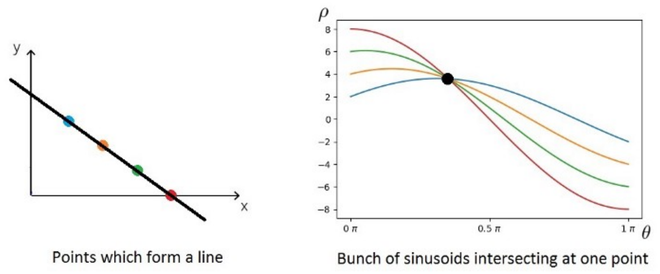
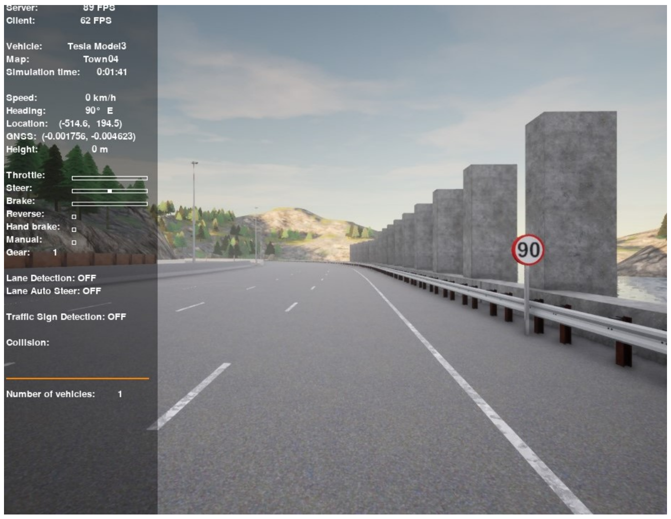


Fig. 4. Generation of sinusoids for each point in a line that results in an intersection of all sinusoids in one point in the Hough Space


```
Server:           89 FPS
Client:           62 FPS

Vehicle:       Tesla Model3
Map:              Town04
Simulation time:     0:01:41

Speed:            0 km/h
Heading:          90°  E
Location:    (-514.6,  194.5)
GNSS:  (-0.001756, -0.004623)
Height:              0 m

Throttle:
Steer:
Brake:
Reverse:
Hand brake:
Manual:
Gear:      1

Lane Detection: OFF
Lane Auto Steer: OFF

Traffic Sign Detection: OFF

Collision:

Number of vehicles:      1
```

Fig. 5. Software control window with input from car front sensor

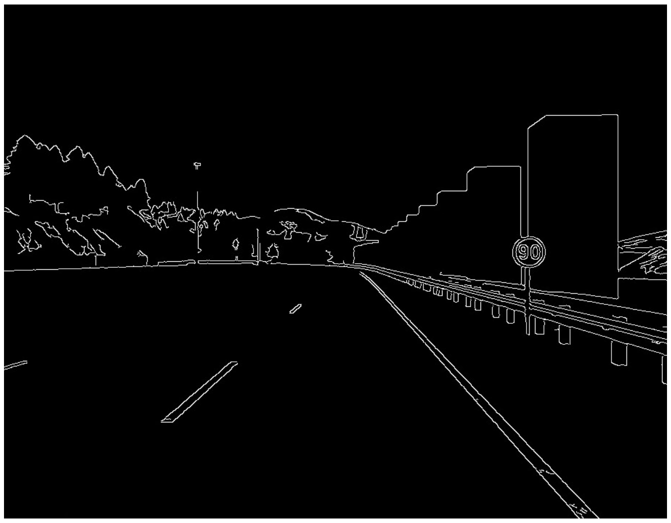


Fig. 6. Output of Canny edge detection algorithm when applied to CARLA input image

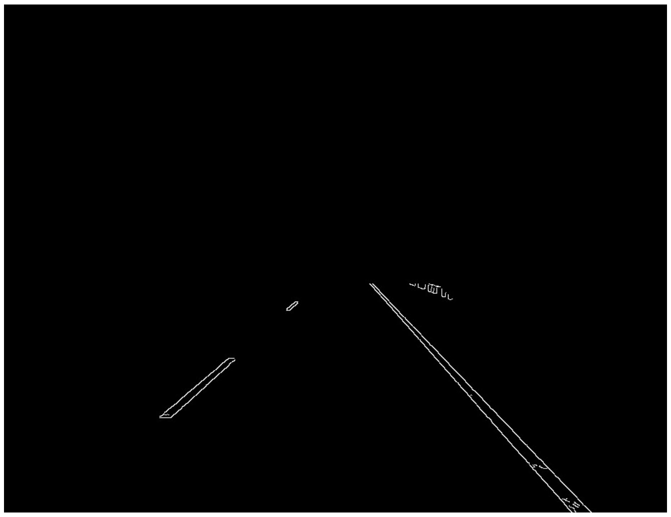


Fig. 7. Canny filtered image with segmentation, displaying only lane area of interest

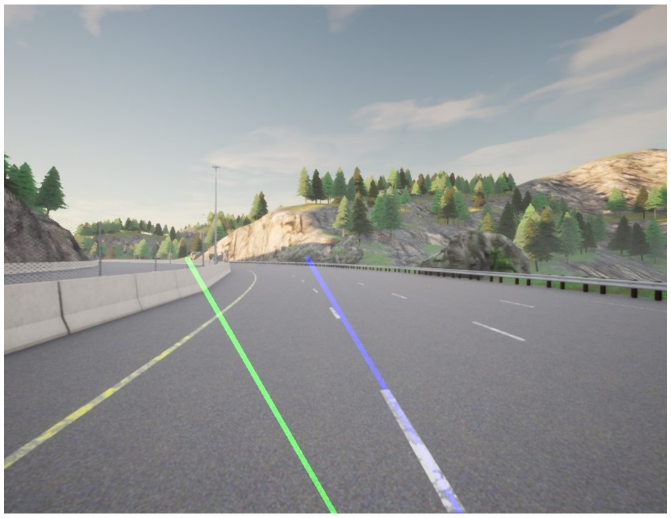


Fig. 9. Lane detection and steering on a smooth curve (the car is able to follow the curve)

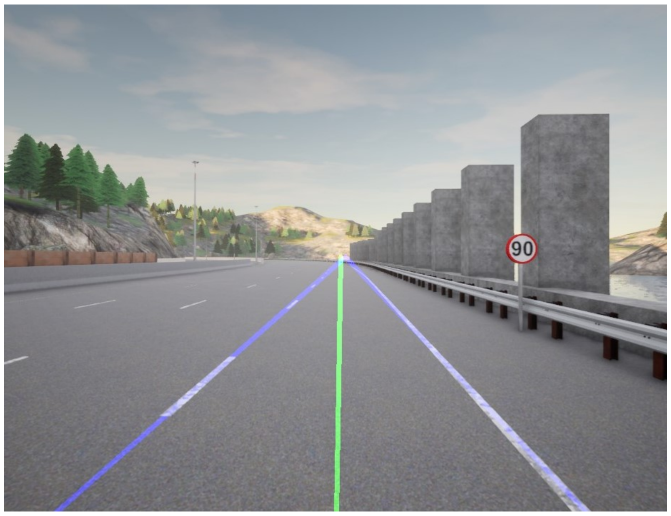


Fig. 8. Lane detection applied to CARLA image. After segmenting the canny output, Hough algorithm is applied and from it, the lane lines are extracted and with a center line, the angle of steering calculated

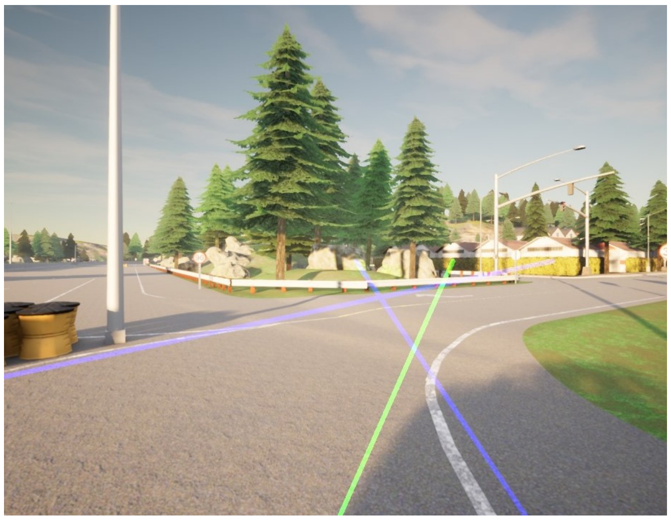


Fig. 10. Lane detection and steering on a sharp curve (the car isn't able to follow the curve)

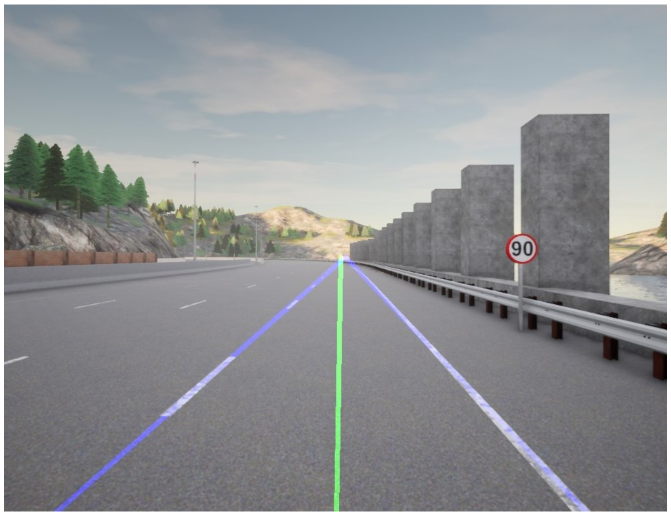


Fig. 11. Lane detection applied to CARLA image. After segmenting the canny output, Hough algorithm is applied and from it, the lane lines are extracted and with a center line, the angle of steering calculated

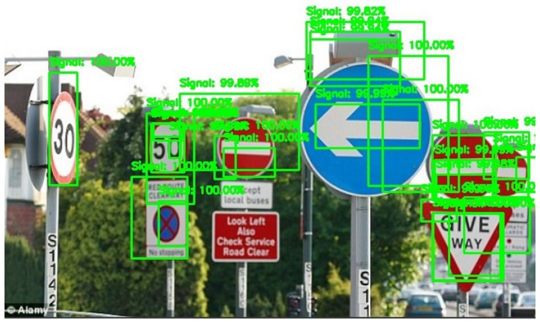


Fig. 14. Region proposals from selective search before NMS

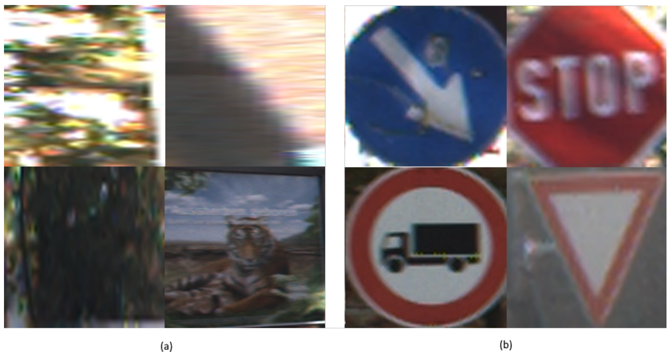


Fig. 12. Region proposals from selective search: (a) No signal, (b) Signal.

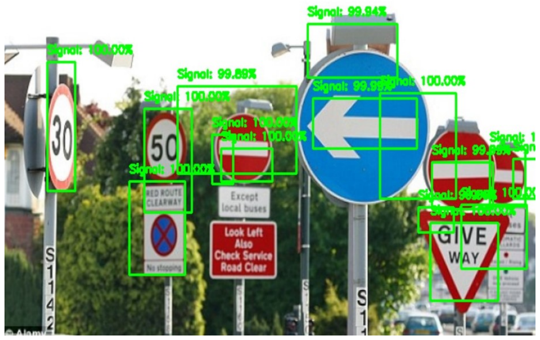


Fig. 15. Region proposals from selective search after NMS

Fig. 13. Image to detect signals

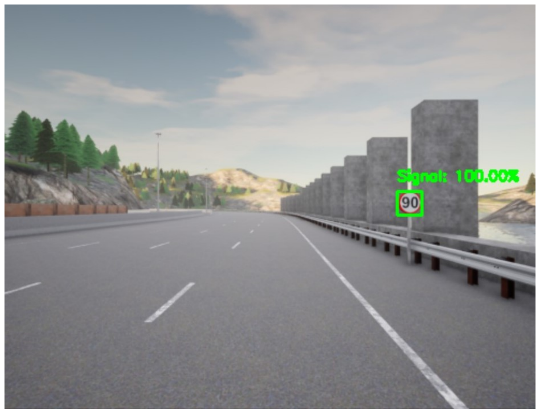


Fig. 16. Traffic Sign Detection from CARLA input image and detection border representation