Computer Vision HW4 Report

Tracking and counting vehicles

Yusuf Patoglu

June 2021

Contents

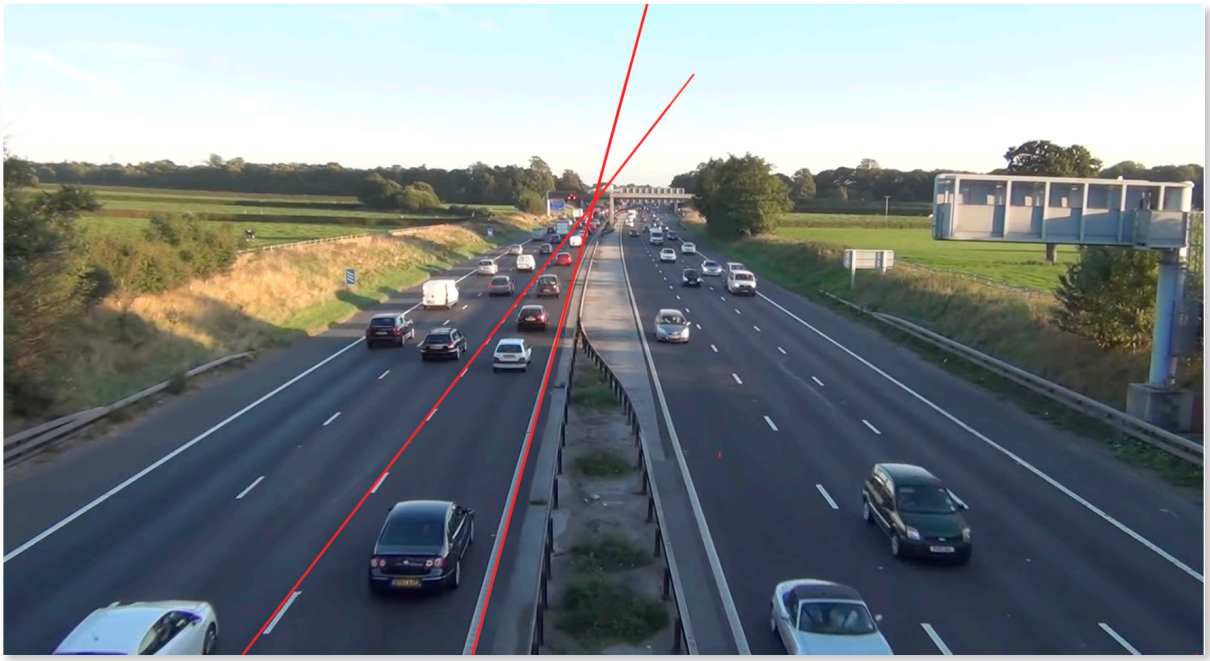
1. Problem Definition
2. Region of interest
3. An observation about the background
4. Algorithm to detect and count vehicles
5. Problems with this simple approach
6. Results and possible solutions

Problem Definition

In this homework we will try to detect, track and find the number of vehicles have passed on a specific lane. For this purpose we have to estimate the background and extract the foreground images. Subsequently an algorithm needs to be developed to count number of vehicles.

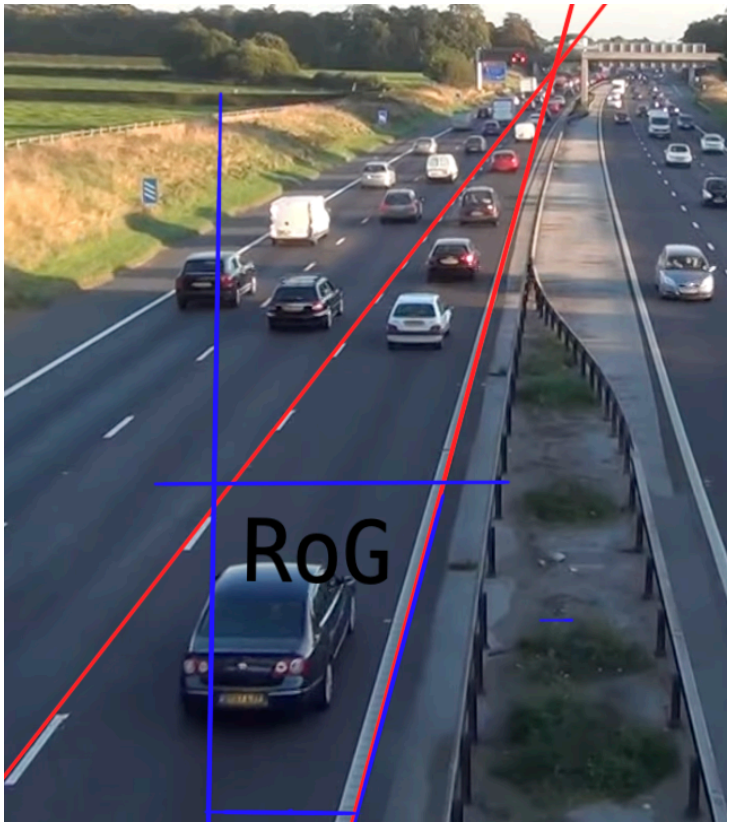
Region of interest

We are not going to track all of the vehicles in this motorway. Only a specific lane will be monitored. Cropping lane problem might look like simple but it is not. Because the lines that look like parallel to each other will intersect at one point in our image planes like this:



Intersection of lines, M6 Motorway

Therefore we have to work in an area that won't interfere with the left lane.



Region of Interest, M6 Motorway

An observation about the background

This is a busy motorway so there will be no pedestrians. All we have to deal with is cars and trucks except the extreme conditions and lightning/weather conditions. Even though the weather is not bad in our example there are shadows, especially in our lane. So our region of interest will be the space between those lines.

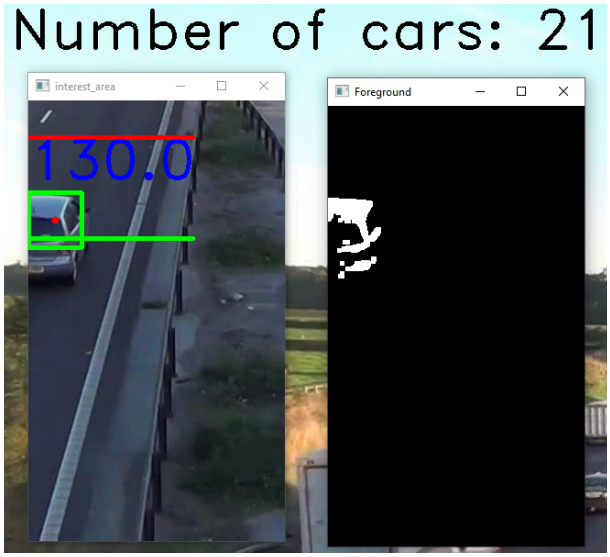
Background Estimation and Additional Filters

There are many options for estimating the background like *frame differencing, mean filter, median filter* and *gaussian mix*. I used cv::BackgroundSubtractorMOG2 Function which is a Gaussian Mixture-based background/foreground segmentation algorithm. [1]

There are two reasons that I used this algorithm:

1. Gaussian Mixture-based background estimation algorithms are robust to gradual changes. This motorway mainly have gradual changes. Cloudy, sunny, rainy day etc. The problem might be about the sudden changes. If there is a blitz or any other unexpected sudden change it won't behave very well.
2. cv::BackgroundSubtractorMOG2 has an optional parameter named as *detectShadows* since there will be shadows in our frames, it's logical to use it.

Before applying the background subtraction first I'm converting my region interest frame to grayscale to apply the Gaussian Blur. After applying the gaussian I'm subtracting the current frame from the background to find the foreground objects. In order to remove the noise in the frame, I apply a sharp threshold filter to get rid of them. Finally, to make the brighter regions bigger, I apply *dilation*. As a result foreground frame looks like this:



Left:ROI with RGB, Right:ROI with extracted foreground

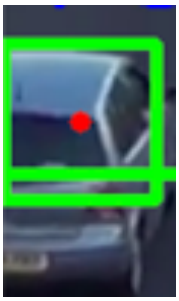
Algorithm to Count and Detect the Vehicles

To track the vehicles first we have to detect the vehicles. I used cv.findContours() for that purpose. It curves all of the continuous points with the same intensity and color. Since we have a nice binary foreground frames it will be useful for us.

After finding the contours we'll draw a rectangle that surrounds the detected object. The problem about this is cars' glasses. They have almost the same color with the ground. So our foreground frame will look like this:

Car glasses counted as the "ground"

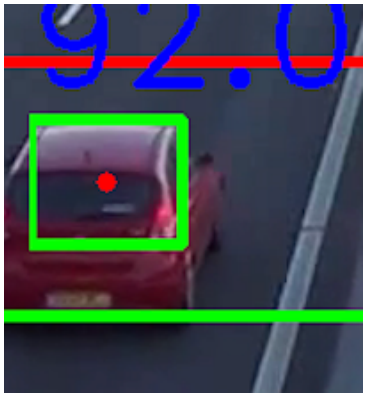
In order to overcome this problem we will apply a threshold. This threshold will only select the rectangles(only big objects) only above a certain limit. In my implementation this area is contour area. I found it with the help of another OpenCV function, cv2.contourArea. In my implementation this number is 800. I ran some experiments to find this value. So we can ignore the dilated little rectangles and focus to our car object:



Object detection using findContours

Now we can detect the cars frame by frame. So only remaining part is counting vehicles. I used a simple approach for that. We had rectangles that surrounds our object. This rectangles center is important for us. We will trace this dot(circle) to see if an object is entered and left our critical area.

In order to define starting and ending lines the ROI is selected very carefully and I put the $y = a$ and $y = b$ lines with many experiments.



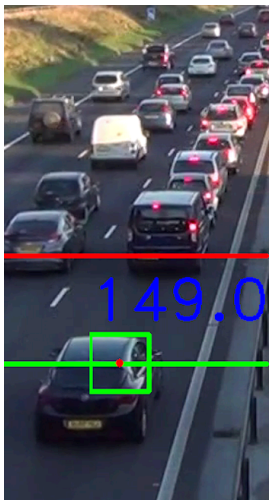
Green: starting line. Red: ending line

92 means that the circle on the central of the rectangle has the $y = 92$ line. Our red line which is end line is $y = 150$ line. So for each frame we're comparing the line coordinates like $92 - 150$, $93 - 150$, … , $148 - 150$. So when $150 = 150$ we know that a vehicle left the zone. So we can increment the counter. In order to tolerate that I'm not expecting them to be equal. I put an error value which is 2. If the difference between $|y_v - y_l| < 2$ where y_v is vehicles changing y coordinate and y_l is fixed ending y line coordinate.

Problems with this simple approach

The main problem is changeable vehicle speeds. For example if the car is too fast I might not detect if the vehicle left the end line or not. To avoid this I kept two lines starting line or end line. If I captured one of these I increment the vehicle count by one. It's worth to say that I'm not incrementing two times if the vehicle is captured on starting line and end line. I still only increment as one in this situation. There is a possibility that the vehicle will not be detected on both lines. Then this is a loss.

Another thing is if the cars are too slow. I couldn't come up with a specific solution to this problem. If there is a traffic in our lane the object may stay in end line for a while. So for only one vehicle my algorithm can count 4-5 vehicles.



If the vehicle is slow there will be extra counts

Results and Possible Solutions

As a result I managed to subtract the background nicely and eliminated unnecessary detections. Also draw two lines that helped me on counting the vehicles. Another solution might be calculating euclidian distance for better recognition but I couldn't implement it by myself so I created my own solution.

References

1. Zoran Zivkovic. **Improved adaptive gaussian mixture model for background subtraction.** In Pattern Recognition, 2004. ICPR 2004. Proceedings of the 17th International Conference on, volume 2, pages 28–31. IEEE, 2004.