Alec Larin and Jason Vazquez-Li
Computer Vision Final Project
May 09, 2017
Professor Magee

Abstract

Using a pre-trained Haar Cascade for vehicle detection, our system is able to track individual vehicles given any video input. Certain attributes were also derived from individual vehicles once detected, in particular, vehicle color. Furthermore, our system is also able to keep track of the total number of vehicles present in the environment.

Introduction

Haar Cascade, proposed by Paul Viola and Michael Jones in 2001, is an approach for object detection using Haar feature-based cascade classifiers. Cascades are created using a collection of negative and positive images. The positive images contains the object to be detected. Positive vector files are created by stitching together all positive images, which is used for the training process. Negative images can be any images except it cannot contain the item of interest. Positive examples are created by superimposing the negative images onto the positive image at various angles. For our project, an existing Cascade file (licensed by Intel) was used.

Although Haar Cascade is only one of many techniques used for detecting vehicles, other methods exist:

- Tracking using background subtraction, as presented in *COMPUTER VISION VEHICLE TRACKING USING BACKGROUND SUBTRACTION* (Chovanec)
- Tracking using Gabor Filters for edge detection *ON-ROAD VEHICLE DETECTION USING GABOR FILTERS AND SUPPORT VECTOR MACHINES* (Sun, Bebis, Miller)
- Tracking using laser range finders *MODEL BASED VEHICLE DETECTION AND TRACKING FOR AUTONOMOUS URBAN DRIVING* (Petrovskaya, Thrun)

Methods

Input

The input to our program is a single video consisting of cars driving along a road. The videos were downloaded from Videezy.com and resized to a frame size of 320 x 240 using VLC, as our program requires a smaller video in order to run efficiently. Some of the gathered videos were not used as the trained Haar Cascade had difficulty identifying cars in them. The videos were approximately 30 seconds in length.

Identifying Cars

To identify cars in the videos, a pre-trained Haar Cascade was used. The cars which were identified in a given frame were indicated by drawing a rectangular box around the vehicle. The trained Haar Cascade, however, did not always continuously detect cars from one frame to the next, an occurrence we call ‘flickering’. Flickering occurs when a vehicle detected in a previous frame is no longer detected by the Cascade in the next frame. This is troublesome because it can result in the vehicle not being counted by our counting algorithm. We attempted to solve this problem by stabilizing the flickering of rectangles in order to more accurately identify vehicles. Our algorithm entailed saving the position of each vehicle (i.e., saving the coordinates of each rectangle drawn around the vehicle) from the previous frame and comparing the rectangle position of the previous frame with that of the current frame. If the rectangles overlapped, the dimensions of the original frame would be used with respect to the new position of the car in the current frame, else, a new frame is drawn on the current vehicle. Unfortunately, this approach did not work for unknown reasons.

Counting Cars

The general approach to count the number of cars in the video was to draw a line, called the boundary-line, perpendicular to the road and increment a counter every time a car crossed it.

The first failed solution employed the rectangle drawn around the cars and a line data structure to represent the boundary. The idea was to increment the counter every time a rectangle overlapped with the boundary-line. This solution failed as OpenCV does not offer a data structure to represent a line. Additionally, even if successfully implemented, the solution would have resulted in too many cars being counted. It generally took a car around 30 frames to cross the boundary-line, and as the counter would have been incremented in every single frame in which a car’s rectangle intersected the line, there would have been approximately 30 cars counted for every one that crossed.

The second unsuccessful approach used OpenCV’s LineIterator. Given an image and two Points, this function returns all of the pixels found along the line in the input image with their respective indices (which are represented via the Point data structure). The indices were stored as keys in a Map data structure for O(1) lookup. As each car crossed the boundary-line somewhere along the line, the idea was to check if the center point of each car’s rectangle was stored in the map. If the map contained the center point, it meant the car was crossing the boundary-line, and as such the counter was incremented. Unfortunately, this method did not identify any cars crossing the boundary-line.

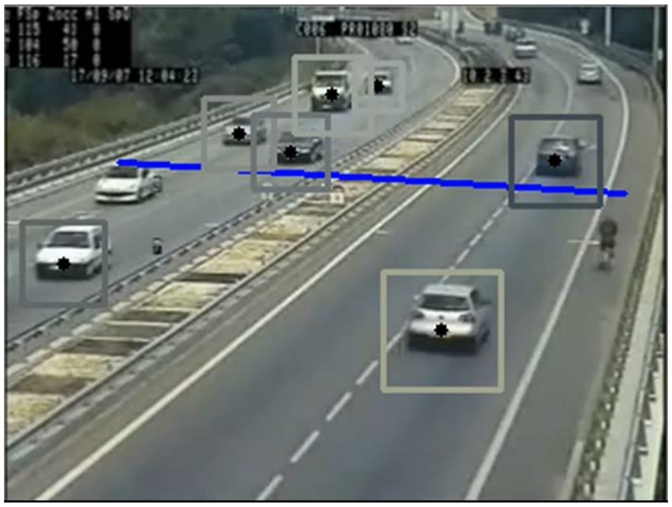
The third and successful solution implemented in our program employed the center point of each car’s rectangle. In each frame, using the y-intercept formula of the boundary-line and the center point, we calculated if the center point of the rectangle was found along the boundary-line; if so, the counter was incremented.

Identifying Car Color

Throughout the video, each car’s color is identified and indicated by drawing its rectangle in the determined color. The first approach calculated the average pixel color found within the rectangles drawn around the cars. This approach was unsuccessful, though, as the rectangle contained part of the road the car was driving on, which lead the color constantly being identified as gray. Instead, the color was calculated using a rectangle 1/12 the size of the original. This much smaller rectangle was centered around the car’s original rectangle’s center point. In doing so, only pixels which were part of the car were used to calculate the average color, resulting in a Scalar which closely resembled the car’s actual color.

Results

We ran our program on two input videos. The first video contained significantly more vehicles than the second video, as well as more noise (e.g., cyclist). Furthermore, the vehicles in the first video were also faster than that of the second.



Almost all vehicles were successfully detected by the Haar Cascade in both input videos, indicated by a rectangular box that is drawn around the vehicle. To test the number of vehicles successfully detected by the counter, a visible line was drawn from one end of the street to the other. Any vehicle that crossed the line resulted in the increment of the counter by one (if more than one vehicle crosses the line simultaneously, the system will adjust the counter accordingly). Below are the results:

- Video 1: identified 25 cars out of 42 cars
- Video 2: identified 9 cars out of 9 cars

While our counting algorithm was able to detect all cars from the first video, it nevertheless underperformed when tested on the second video, with only a accuracy rate of 59.5%. Some vehicles were also counted more than once by the algorithm.

Color detection resulted in decent accuracy. However, our algorithm at times detected the wrong color because the center point of the rectangle was located on the windshield/light/etc, resulting in the color being computed from that feature as opposed to the car’s color.

Discussion

Haar Cascade provides an effective way to detect objects. This approach allowed for the successfully detection of vehicle(s) given any input video contains vehicles. While our system was able to detect vehicle color with decent accuracy, it nevertheless encountered problems counting the number of vehicles at times. Future work would entail deriving a more accurate algorithm to count the number of cars in the environment.

Deliverables

- Car_video1 & car_video2: Input videos
- Final_video1 & final_video2: output videos
- VideoSample: Example picture from the output videos
- FinalProject-AlecL&JasonVL: OpenCV project containing the project’s code