Traffic Tracking using Computer Vision

Brian Joseph and Connor Galvin

MATH 4025: Applied Mathematics Capstone

Professor Lee-Peng Lee

April 18, 2018

Table of Contents

Abstract

The goal of our project is apply computer vision techniques to count the number of vehicles passing in either direction on a two way street. We demonstrate the applications of computer vision through the use of Python's OpenCV package. Our project demonstrates the use of image pre-processing, background subtraction, and contour fitting to build a motion tracking model. After building out motion detection model we were able to keep count of each passing vehicle using their location and direction within the frame.

Introduction

The original goal of our project was to identify and count the number of people entering and leaving Northeastern's Marino Fitness Center. Using livestream footage posted on Northeastern's website, we made good progress towards our goal. However, in the beginning of March the webcam was blocked off and eventually moved, making our project infeasible. Due to the low resolution and the complex movements of people, we needed to train and test a machine learning model to identify the number of people in a frame at any given time. We only had a few minutes of video saved before the webcam was moved, not nearly enough to train and test with. This led us to abandon our original project idea in search for a new webcam to apply our computer vision motion tracking techniques to.

After searching the web for alternative webcams, we discovered one that fit our needs well. The cam featured livestream data of a two-way street at Colgate University in Hamilton, NY [1]. The video's resolution and frame rate are high enough to identify objects and track motion smoothly between frames. Since the road is straight, the cars move in a very predictable direction that simplifies our task of tracking them. Our new goal was to automate the identification of individual vehicles, track them across multiple frames, and ultimately count the number of vehicles that pass in both directions over a certain time frame.

Model and Analysis

The first step of our model was streaming and displaying the webcam data in Python. With the use of OpenCV's urlopen, stream, and imshow methods we were able to display the series of individual frames as a video in Python.

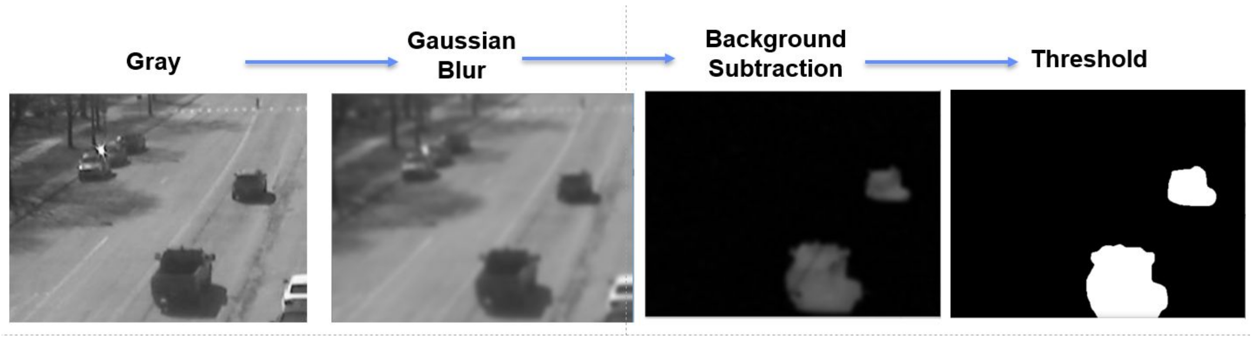
Next, we defined alterations that we could apply to the frames. The first alteration that we defined was cropping. It is useful to crop the frames to remove as much unnecessary noise as possible, such as parked cars and pedestrians. After experimentally testing different dimensions, we sliced the image to contain as little non-road surface as possible. However, we were still left with some sidewalk and road-shoulder in our frame, as the road is not completely straight. The next alteration that we defined is graying. By using OpenCV's cvtColor function we are able to convert the frames from RGB color scale to grayscale. This reduces noise and complexity and speeds up our code. We don't lose any useful information about our images from an object detection standpoint when converting to grayscale.

Another alteration that we implemented is Gaussian blur. Gaussian blurring diminishes image detail and removes noise that might interfere with object detection. This methods employs a Gaussian function to calculate the transformation that will be applied to each pixel in the image. The formula below represents a two-dimensional gaussian function. In the instance of image processing, x and y represents the distance from the horizontal and vertical axis and each pixel is represented by a unique combination of (x,y). [2]

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Formula of a two dimensional gaussian function

Next, we needed to implement background subtraction to the frames of our video. In order for this to work we need to initialize our video when there are no vehicles or people seen on camera.. After applying image filters such as graying and noise reduction, we take the absolute difference between the first frame and each subsequent frame. This process is implemented with OpenCV's absdiff function. The result displays all of the "new" objects in the frame that were not present in the first frame. The differences between the frames are considered the foreground, while the pixels that remain the same are considered the background. After this we further refine this difference by defining a threshold. The absolute difference between two frames picks up all differences in pixel intensity value no matter how small. By defining a minimum threshold, we are able to ignore small changes in pixels that are most likely caused by noise or changes in lighting, while setting all differences greater than the threshold to the foreground.



The first four stages of our pipeline illustrated.

After isolating our foreground from our background, we moved onto finding contours and drawing bounding boxes. According to the OpenCV documentation, contours can be described “as a curve joining all the continuous points (along the boundary), having same color or intensity” [3]. The idea is to have one contour for each vehicle and identify each vehicle as one object. As we will discuss later, this proved to be one of our biggest challenges. After finding contours in each frame, we would store all of its information in a Python class, containing the contour’s first appearance, age, coordinates, area, and direction. After storing this information about each contour, we proceeded to draw rectangular bounding boxes around each one.

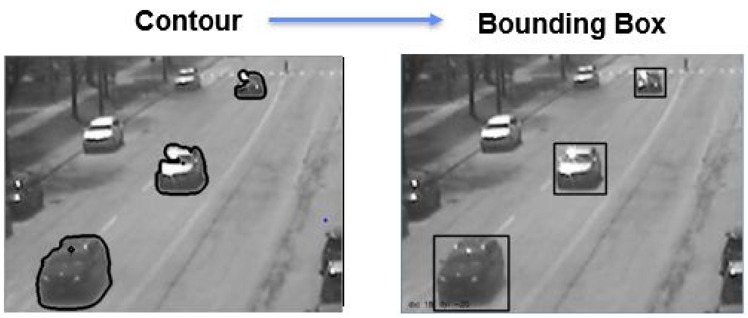


Illustration of contour fitting and drawing of bounding boxes

As previously stated, identifying each vehicle as one object proved to be one of our biggest challenges. Gaps in the foreground objects would often lead to vehicles being identified as 2 or 3 different contours. One method that we tried to implement to fix this problem is dilation. The idea behind dilation is convolutiting an image with a kernel in order to “grow” a foreground image and fill in gaps that would otherwise be defined as the background.

We employed various tools to tackle the problem of only identifying contours which actually include cars. Sine we didn’t plan on using any machine learning (such as a CNN) to achieve this, we had to use various heuristics to improve or models performance. The attributes we used for each contour were:

- Size

- Precision & Recall
- Age
- Total number of contours on the screen
- Direction
- Proximity to other contours

The size of a contour is a good heuristic because the majority of cars in the feed are about the same size. However, it is not a perfect heuristic as sometimes motorcycles and large trucks pass through the feed. If you look each pixel in our “Thresholded” image above, you can treat the bounding box as a decision boundary and calculate the precision and recall of a given bounding box. This allows you to reason whether or not a given contour actually contains a car. For instance, a contour of a car should have a precision rate of at least 50% and have a recall rate that is inversely proportional to the number of total cars on the screen. Next, we used the “age” of a contour to guess whether or it contains a car. A car passing through the feed should not be on the screen for more that 100-200 frames. We also used the number of total contours to infer information. If you have 20 contours on the screen already, then the probability of there being another contour of a car on the screen in very small. We used that information to our advantage when creating contours. Also, the direction of a car should be fairly north/south and any contour moving east/west should not be identified as a car.

And finally, if two contours are very close to each other (even overlapping) then we can infer that at least one of them is probably not a car. All of this information is used in our model to help improve our contour detection algorithm.

Once we identified contours in our feed and tracked them across frames, the next step was to infer the direction of identified cars and detect when they enter and leave the feed. For each contour in each frame, we store the center point of all of its previous positions as an instance attribute called “points”. This attribute is the crux of determining the direction of a contoured vehicle. For each contour, we simply take the set of points in this attribute and perform a least squares regression. We then take the resultant slope of the line generated by least squares regression and store that as the contours direction. This is only performed when a contour has at least 3 points in its “points” attribute. It turns out the center of the road has a slope of about .7. Therefore, we only pay attention to contours that have a direction slope of about .5 to .9. The next step was to tell when a contour with a direction of this type was leaving the screen. We did this by comparing contours from the current frame to contours from the previous frame. If the number of contours decreased, then a contour has just left the frame. Using this information, we identify which contour has left the screen and look at its last point value. If the point was at the top of the screen, then we can infer that the contour has left the north side of the frame and similarly for identifying southward bound contours.

Results

We have achieved our goal of counting vehicles passing on both sides of the two way street, however our model is not perfect. We have good performance when all cars are traveling smoothly and not performing erratic maneuvers. However, there is a stop light located just outside the south side of the frame. This often leads to cars bunching up at the bottom of the visible part of the road. This causes inconsistent performance, since we use this strip of road to decide when a vehicle has passed.

In addition our program does a great job of excluding non-vehicle movement from our passing vehicle count. Although our program identifies pedestrian motion, they are generally not counted towards our passing vehicle count. Since pedestrians usually move perpendicular to the direction of traffic flow, it was easy to ignore them in the vehicle count with a simple direction restriction.

Lighting and conditions play a large role in how well our model will perform. We saw consistent performance during daylight hours with both cloudy and sunny conditions. However, close to sunset and sunrise the long and quickly changing shadows were often identified as moving objects. In addition, lighting conditions led to some issues at nightime, especially in the case of rain and wet ground.

Discussion and Conclusion

We achieved our goal of programming a model to count traffic on a two way street in good lighting conditions. However, in poor and quickly changing lighting conditions our model's performance suffered. We believe that taking a deep learning approach to our project would fix some of these problems and yield even better results. By training a neural network with hundreds of images from our webcam, each labelled with human drawn bounding boxes, we would be improve the accuracy of our model. With enough training samples, our model would learn to distinguish vehicles from one another and draw optimal bounding boxes. Possible drawbacks to this deep learning approach are: the amount of human time required to generate training samples, the high amount of processing power required, and slow performance of the code.

Computer vision is a rapidly advancing field, and new applications will emerge in the coming decades. Advancements of the methods used in our projects will allow engineers to better understand and control traffic patterns. This could lead to reduced congestion, decreased commute times, and faster responses from emergency vehicles. There is still much research to be done before these applications become widespread, as we have seen how much work it has to create even the simplest computer vision models.

References

1. http://www.opentopia.com/webcam/12366?viewmode=livevideo
2. https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_filtering/py_filtering.html
3. https://docs.opencv.org/3.1.0/d4/d73/tutorial_py_contours_begin.html