Kai Matsuka Caltech EE148
Homework 1 April 14th, 2020

1 Github

Link: https://github.com/kaimatsuka/caltech-ee148-spring2020-hw01

2 Red Light Detection

Problem A [1 points]: What algorithm did you try?

Solution A.: *I tried a perceptron algorithm. Suppose a kernel k has size m rows and n columns. Since the image has 3 channels, the kernel can be represented as a vector $k \in \mathbb{R}^{3mn}$. The image I has size M rows and N columns $I \in \mathbb{R}^{3MN}$. For each pixel i in the image I, create a $m \times n$ patch centered at i, denoted by $x_i \in \mathbb{R}^{3mn}$. Note that i is selected only from pixels that are $m/2$ rows and $n/2$ columns away from edges.*

1. *A kernel k is selected by manually selecting a bounding box of a red light*
2. *Kernel is smoothed by taking average of 5×5 grid around the pixel*
3. *The pixel intensities of k is shifted to centered around 0: the values between 0 and 255 is shifted to between -128 and 127. Then k is normalized such that $||k|| = 1$.*
4. *For each pixel i, select x_i. Shift x_i to center around zero as well as normalize such that $||x_i|| = 1$. Note that only coordinates that are $M/2$ rows and $N/2$ columns away from edges are considered for the detection.*
5. *Compute $y_i =< k, x_i >$. Note that the rows and column .*
6. *Convert the result to binary values according to threshold T:*

$$\tilde{y} = \begin{cases} -1 & \text{if, } y_i < T \\ +1 & \text{otherwise} \end{cases} \tag{1}$$

7. *Determine neighbor pixels corresponding to a same red light. This is done by finding a maximum y_i in some radius r near a pixel with $\tilde{y}_i = +1$, and ignoring any positive match $\tilde{y}_j = +1$ in that radius r.*

I tried following things

1. *Different ways of normalizing kernel and image patches*
2. *Different methods for clustering neighboring positives*
3. *Changing the kernel dimensions*

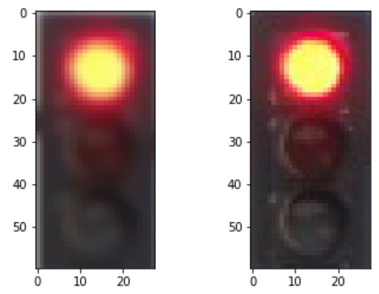


Figure 1: Smoothed kernel (left) v.s. original kernel (right).

Problem B [1 points]: How did you evaluate algorithm performance? Can you think of any situations where this evaluation would give misleading results? You are not being provided with ground truth, so it is not necessary to compute any particular performance metrics over the whole data-set. Just describe how you decided what works and what doesn't.

Solution B.: *The performance of the algorithm was determined qualitatively and manually by observing whether the detected bounding box matches the red light or not. This way of performance evaluation can have multiple issues. First, the error is not quantitative, so it doesn't tell how off the detection is. It is also difficult to quantify at what point one considers the classification was correct or not. Is it correct if the bounding box covers 50% of the red light? 70%? This metric is also limited by human's ability to discern the ground truth. For example if the visual condition is so bad that it's difficult for humans to discern red lights from car break lamps, then labels by humans may also have a misleading result.*

In practice, one should use quantitative approach to measure the performance of the algorithm. This can be done by providing a correctly label data set with tight bounding box, and then defining a measure of deviation in terms of the center of the bounding box and the size of bounding box. One can have varying the penalty (weights) for false positive and negatives, depend on the desired outcome.

Problem C [1 points]: Which algorithm performed the best?

Solution C.: *I tried the detection with and without moving average smoother for the kernel vector k. 5×5 smoother worked well for the kernel considered in this report. This is because the smoother kernel rejects a local variation of pixel intensities and reduce noise. Figure 1 compares the two figures. The smoothed kernel worked much better in detecting than the kernel without smoothing.*

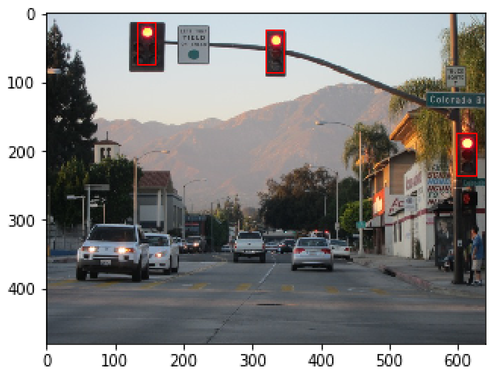
Remapping pixel values to between -1 and 1 helped because the kernel for the red light has both bright and dark parts. When the pixel intensities were quantified in a positive range (0-1 or 0-255), taking dot product $< k, x_i >$ contributes to more "postive" value for bright pixels only. In reality, other pixels should be taken into account rather than ignoring.

The threshold was tuned to $T = 0.92$ after verifying the detection result after few sample images. Threshold too large would not detect the correct red light; even the same patch from which kernel was generated will not result in $y_i = 1$ because the kernel is smoothed. Threshold too small will incur many false positives. After tuning on few images, the threshold value $T = 0.92$ is selected such that only the red lights are detected (among those set of images).

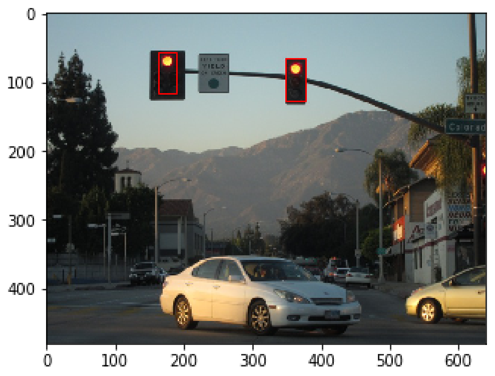
Problem D [1 points]: Provide a few examples (images with predicted bounding boxes) where your best algorithm succeeded. Can you explain why these images were "easy" for your algorithm?

Solution D.: *Figure 2 shows few images where the images of light are correctly detected. It shows that all the red lights these images are correctly detected, and there are correct number of red lights detected (no false negatives or false positives). These images were easy because the kernel was created using the RL-010 (middle red light). Both of these images have correct scaling (i.e. the number of pixels that represent red lights are approximately the same), they have similar lighting condition (evening but not too dark), the red lights are viewed directly from the front without obstruction, and because these red lights have approximately the same design (e.g. frame has similar dark color).*

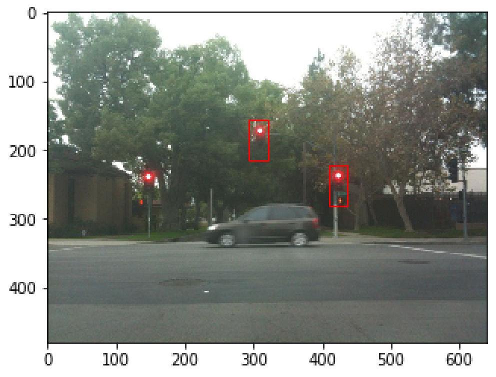
To recreate these result, run "code/helper.py" as a script. Modify the file name of the image to appropriate names (e.g. "RL-010.jpg").



(a) RL-010



(b) RL-011



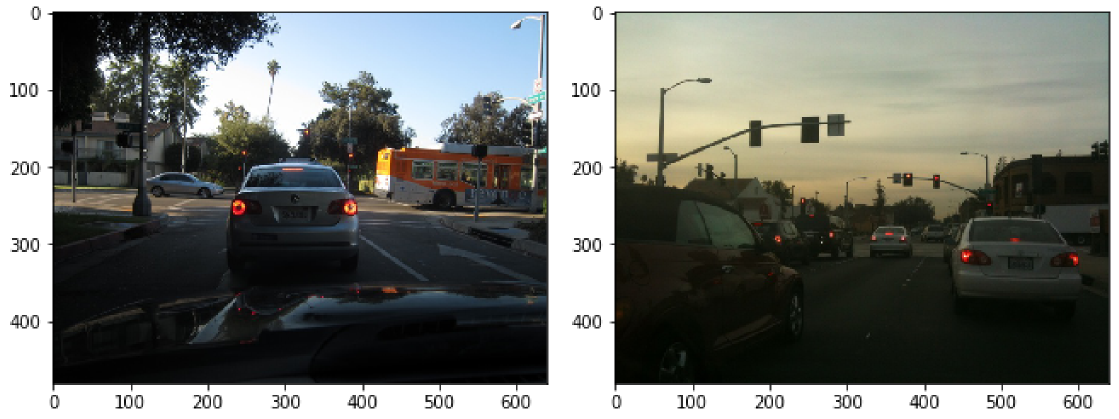
(c) RL-120

Figure 2: Correctly or partially correctly detected images

Problem E [1 points]: Provide a few examples (images with predicted bounding boxes) where your best algorithm failed. Can you explain why these images were "hard" for your algorithm?

Solution E.: *Figure 3 shows incorrectly labeled images. Majority of the falsely classified images were false negatives. There are a few likely explanation for this. First, because the kernel was not scaled to multiple sizes, when the red lights are at further or closer distance than the one in the kernel, detection failed. This is the case for the example images shown in Figure 3. Second, the threshold selected (0.92) is pretty high, so most of images were rejected unless the selected patch matches to the kernel at exceptionally high accuracy.*

To recreate these result, run "code/helper.py" as a script. Modify the file name of the image to appropriate names (e.g. "RL-025.jpg").



(a) RL-025 (b) RL-177

Figure 3: Incorrectly detected images

Problem F [1 points]: Identify a potential problem with your approach and propose a solution.

Solution F.: *Some obvious issues for the classification was overlooked in this project.*

- *scaling*
- *rotation of kernel*
- *lighting conditions*
- *different types of red light*
- *different size of bounding box*
- *context (some break lamps on car with dark frame may look very similar to red light... but this may be resolved using context)*
- *obstruction*
- *red arrows, red light with yellow arrows, and other patterns*