EE/CNS/CS 148 HW 1: Red Light Detection

Johanna Karras

April 6, 2021

Assignment Description: In this assignment, I develop a simple algorithm for detecting red lights in urban scenes using *only NumPy functionality.*

GitHub Repo:
`https://github.com/JSKarras/EE148-Detecting-Red-Traffic-Lights`

1 What algorithms did you try?

Matched Filtering: For this task, I implemented a simple matched filtering algorithm, similar to that described in Lecture 02. I used a single kernel, an image patch containing a traffic light, and generated additional kernels by resizing the original kernel to different sizes. Then, for each size of kernel, the algorithm sweeps over 2D patches of the same size of the input image. For each kernel and image patch pair, I normalize the red, green, and blue channels and compute the mean absolute error between the two. Then, I use a pre-defined array of thresholds to determine which image patches achieve a "low enough" MAE with the given kernel. Those image patches are stored as bounding boxes around red lights. *The specific implementation details and pseudo-code are given in problem 3.*

Approaches to Image Normalization: I tested different ways of normalizing the images to find the one that is most robust to differences in lighting, luminescence, and color scale.

- I tried normalizing by simply dividing each color channel by the total color of the pixel, but this was not so useful, because it lost information about the relative color intensity between the image patch and kernel.
- I tried normalizing image patches to the kernel's minimum and maximum values for each color channel, but found that this was not very useful, since they tend to be 0 and 255 respectively for basically every image and color.
- I also tried using grey-scale and red channel-only. The tricky part was that there is a lot of black around a traffic light, so grey-scale and red

channel-only image normalization ended up favoring pavement or skies, where there is also a lot of the same color.

- I tried to normalize by computing each pixel's distance to the mean of the whole image for each color channel separately. This worked the best.

Kernel Selection: I experimented with different options for traffic light kernels. I found that kernels of full traffic lights favored patches of black backgrounds, since there was a lot of overlap with the black around the red light. Square kernels only containing a red light ended up detecting too many false-positives, such as car headlights. The best kernels included the full traffic light with a clear border around it, like a blue sky.

Stride: I tested different stride lengths and found that smaller strides (number of pixels between each convolution) result in more accurate localizations of objects, but create a trade-off with speed, since it creates more patches to sweep over.

2 How did you evaluate algorithm performance? Can you think of any situations where this evaluation would give misleading results?

Since I did not have access to ground truth labels, my process of evaluation was highly observational. For each image kernel and size, I applied it to 10-20 input images and evaluated the results with different thresholds. If I observed a large number of false-positives, I would lower the threshold. If I observed a large number of false-negatives, I would raise the threshold.

This type of observational approach is definitely not very precise and still leads to quite a few false-positives and false-negatives. This is especially true if the kernel and observed images differ significantly from the actual input images. For instance, if all traffic lights in the observed set are very small and the kernel is chosen to be small, then a specific threshold may work well for small lights, but poorly for large lights. Plus, this approach is labor-intensive and slower than an automated quantitative metric.

3 Which algorithm performed the best?

The best algorithm using numpy only operations was patch filtering based on mean-absolute-loss between image patches and kernels, where both were normalized with respect to the total patch color, so for an image I and image patch I_p...

$$I_p[:,:,0] = (I_p[:,:,0] - \hat{I}_r)/255$$

$$I_p[:,:,1] = (I_p[:,:,1] - \hat{I}_g)/255$$
$$I_p[:,:,2] = (I_p[:,:,2] - \hat{I}_b)/255$$

where $\hat{I}_r = \sum I[:,:,0]/|I|$, $\hat{I}_g = \sum I[:,:,1]/|I|$, and $\hat{I}_b = \sum I[:,:,2]/|I|$, or the mean of each color channel.

For my image kernel, I chose one that had plenty of contrasting color (like blue sky) around the traffic light as a way to prevent the kernel from being too similar to patches of only sky or pavement. I found that a stride length that was $\frac{1}{5}$ of the kernel height was a good compromise between accuracy and speed.

This algorithm is implemented in the `run_predictions.py` script in the GitHub repo linked above.

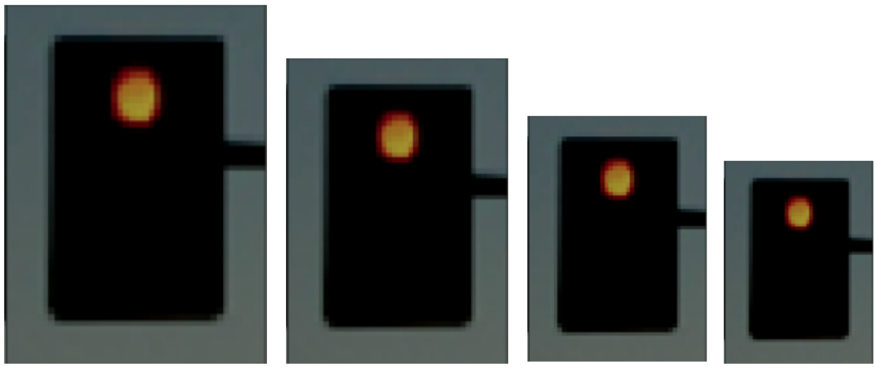
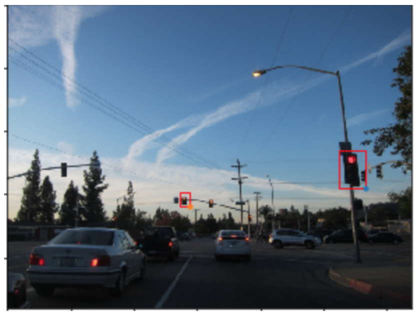


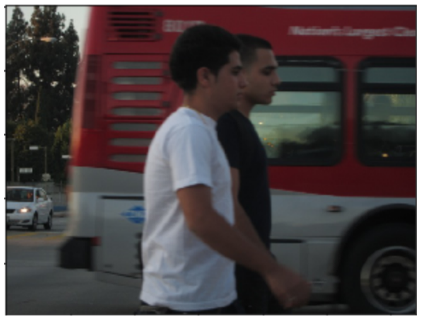
Figure 1: My chosen kernel after normalization and rescaling to different sizes.

4 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?

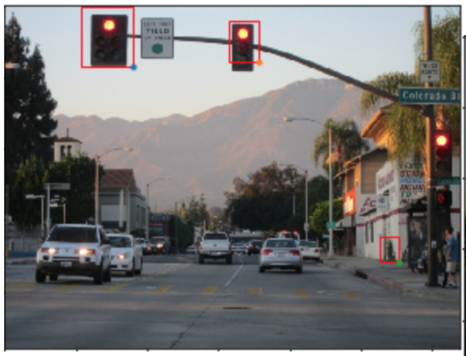
These images can be generated using the `run_predictions.ipynb` script in the GitHub repo linked above.



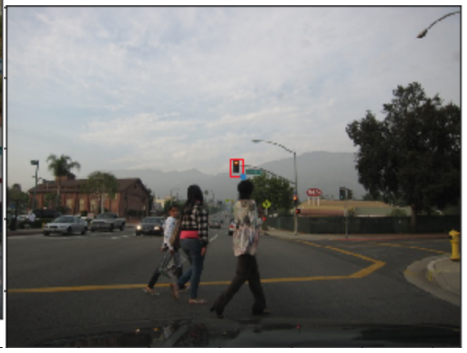
(a) This image is easy, because although there is a lot of red, there are no singular red lights (from cars for instance) to confuse the kernel.



(b) This image is similarly easy, because there is a lot of contrast between both red traffic lights and the blue sky in the background.



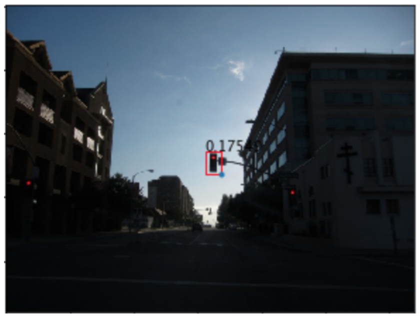
(c) This image is easy, because there is a lot of contrast between the red traffic light and the blue sky in the background, which helps match its patch to the kernel. Additionally, the traffic lights are large, providing lots of overlap with the kernel.



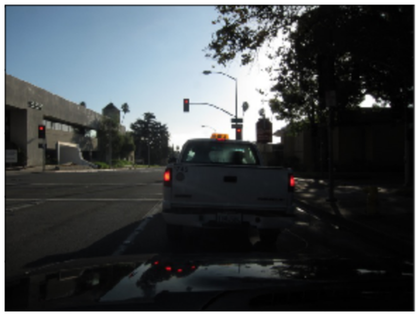
(d) This traffic light is easy to detect, because there is a lot of contrast between the red traffic light and the blue sky in the background.

Figure 2: Examples of images with easy-to-detect red traffic lights.

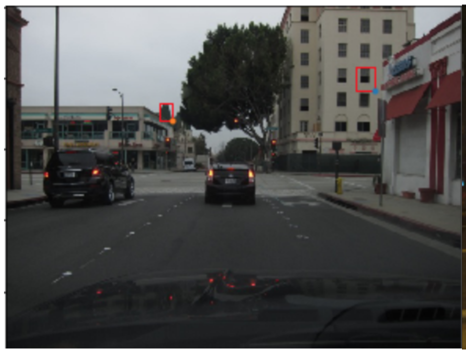
5 Provide a few examples (images with predicted bounding boxes) where your best algorithm failed. Can you explain why these images were "hard" for your algorithm?



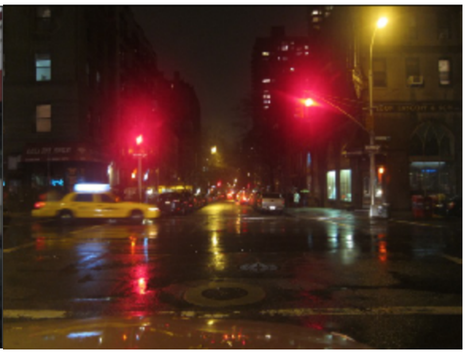
(a) This image is tricky, because the background is black and thus there is no edge around the traffic lights on the left and right sides. Since there is no edge, the traffic lights are quite different from the kernel, which has a blue edge around the traffic light.



(b) This image is also hard, because the traffic lights are very far away and very small. As such, they easily go undetected since they provide very few pixels worth of overlap with the kernel, resulting in a higher MAE.



(c) This red lights in this image are blurred and stretched due to the rainy weather and fog, which causes the light to be scattered. Since the red traffic lights are now substantially larger and different from the kernel, they go undetected.



(d) This image is also hard, because there are misleading objects in the image that look almost like traffic lights, but which are not. In this image in particular, the black windows on the building have lots of overlap with the kernel of a red traffic light.

Figure 3: Examples of images with hard-to-detect red traffic lights.

These images can be generated using the `run_predictions.ipynb` script in the GitHub repo linked above.

6 Identify a potential problem with your approach and propose a solution.

An issue is that the algorithm is not very robust to different traffic lights or different luminescence (like when it is foggy and the lights are blurred). This is because this method is highly dependent on the choice of kernel. A solution would be to use more than one kernel and enforce stricter thresholds for each. Alternatively, averaging multiple kernels could provide a solution.