CS148 Homework 2 Red Light Detection with Convolution and Performance Evaluation

Michelle Zhao

April 14, 2020

1 Deliverable 1

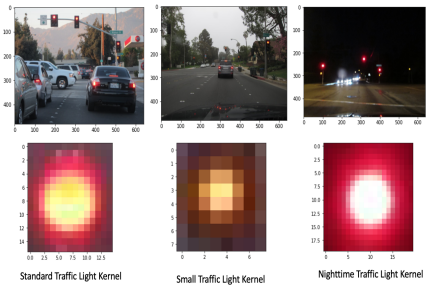
I annotated a subset of the SageMaker images.

2 Deliverable 2

2.1 3D Matched Filter

The second algorithm is a 3D convolutional matched filter. The matched filter algorithm takes a kernel image of a red traffic light, which serves as the filter for the image. First, I sampled several of the traffic light images in the data set, and found the top three clearest and most representative images of a red traffic light.

I noticed that some images had traffic lights that were up close, and some had traffic lights that were from far away. Some images also occurred at night, resulting in different looking red traffic lights in the day versus the night. Below is an image of my three traffic lights kernels: (1) Standard Red Light, (2) Small Red Light, (3) Nighttime Red Light.



In order to account for lighting differences, as suggested in the lecture, we normalize the kernel, as well as normalize the area of the image at each locations that we are filtering.

We perform z-normalization, which normalizes the segments of pixels to have 0 mean. We normalize by subtracting each pixel value by the mean pixel value of the region, and diving by the standard deviation of the region. This allows us to perform normalized cross-correlation in our convolution.

$$z = \frac{x - \min(x)}{\max(x) - \min(x)}$$

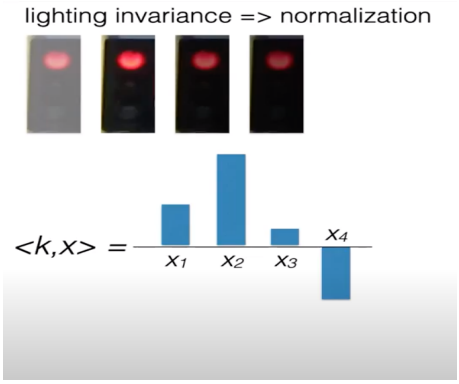
Z score normalization:

In Z score normalization,

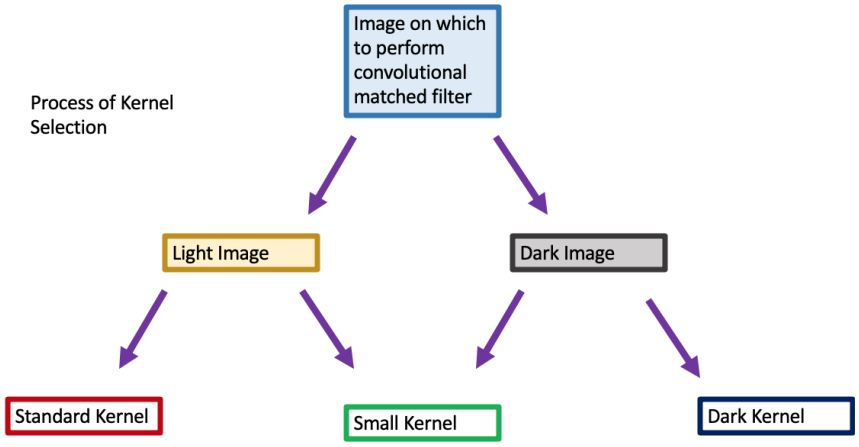
$$z = \frac{x - \mu}{\sigma}$$

μ = Mean

σ = Standard Deviation

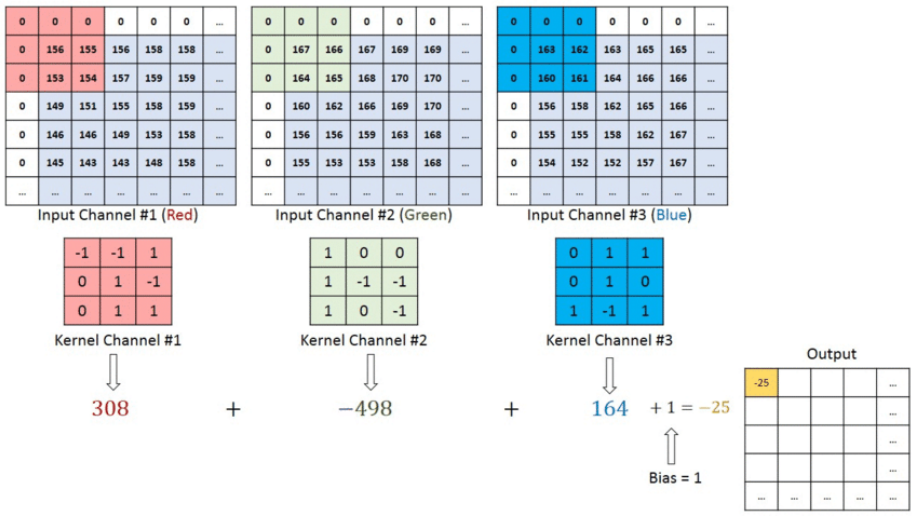


For every image, we perform 3D convolutional filtering with 2 filters. If the image is dark, which we check by computing the mean RGB value and comparing it to a light/dark threshold, then we set the standard filter to be the dark filter, and run convolution matching with the dark filter and the small kernel filter. Otherwise, if the image is light, then we use the standard light filter and the small kernel filter. Thus, the small kernel filter is run with all images. Below is a diagram of the process of kernel selection for each image.



Then, we perform a 3D convolutional matched filtering of both kernel filters

over the entire image, using normalized cross-correlation. The convolutional filter is performed over the red, green, and blue channels separately, and then the three normalized cross-correlation values are averaged.



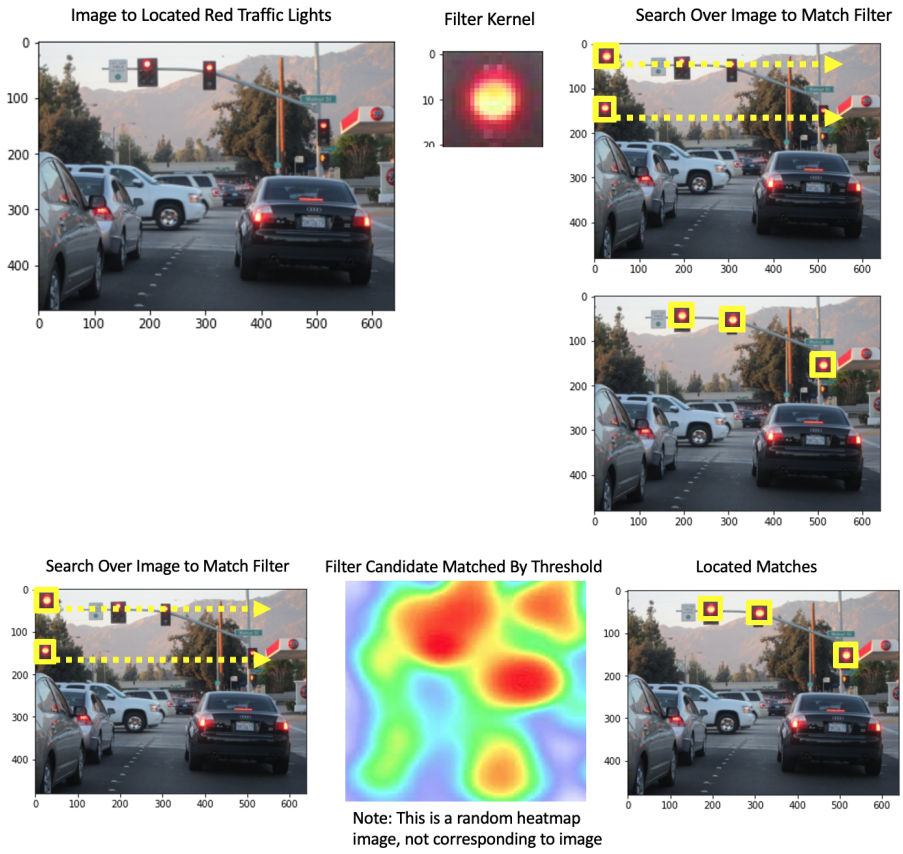
$$g[m,n] = \frac{\sum_{k,l}(h[k,l]-\bar{h})(f[m+k,n+l]-\bar{f}_{m,n})}{\left(\sum_{k,l}(h[k,l]-\bar{h})^2 \sum_{k,l}(f[m+k,n+l]-\bar{f}_{m,n})^2\right)^{0.5}}$$

mean template

mean image patch

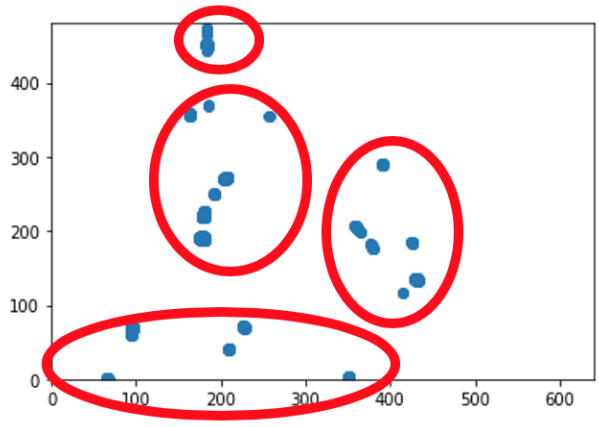
The confidence level was the normalized cross-correlation value we computed. We identify pixels that match the filter by filtering pixels over a threshold of 0.7. This threshold was found by grid search testing, and 0.7 performed the best. We combine the identified pixels from the standard and small kernel filters into a single set of identified red light pixel centers. The confidence score for each identified pixel is defined to be the the normalized cross-correlation value we computed from kernel convolution filter.

The below figure shows the matched filtering in action. This algorithm performed poorly with only the standard size kernel, but when run with both kernels, standard and small, the algorithm performs very well. We also set the convolution stride length to 1, to ensure thorough checking of the filter. Because



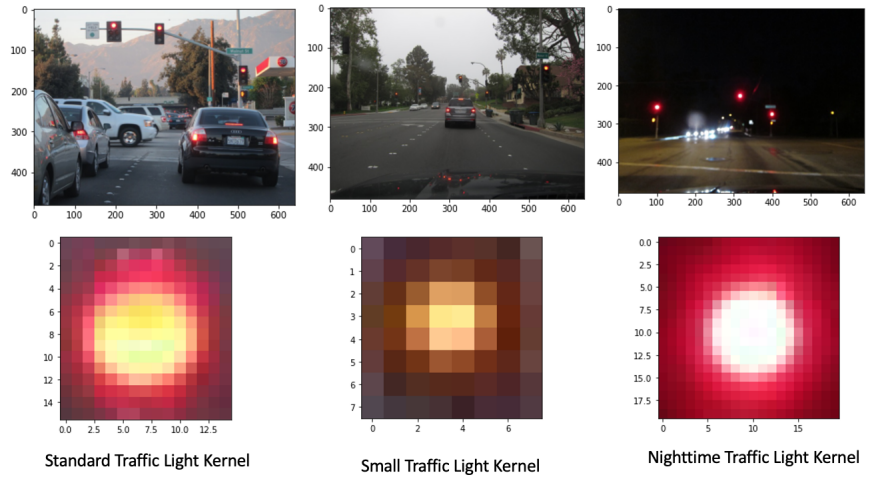
some red lights in the images are tiny, the small kernel is tiny, and thus, we want to keep the stride length small.

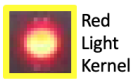
Then, after we select the pixel locations that are above threshold, we perform K-means clustering to narrow down the selection of red traffic light detections. We do this because there are often detections of pixels that are right next to each other, that belong to the same traffic light. We select the number of clusters to be the minimum between the number of pixel candidates that meet the threshold and 4. After investigation and grid search over iteration caps and thresholds, we decided 4 was the best value, and an iteration cap of 7 was best. We restrict the cluster centers to be points in the set. The results of this final algorithm showed that it performed pretty well.

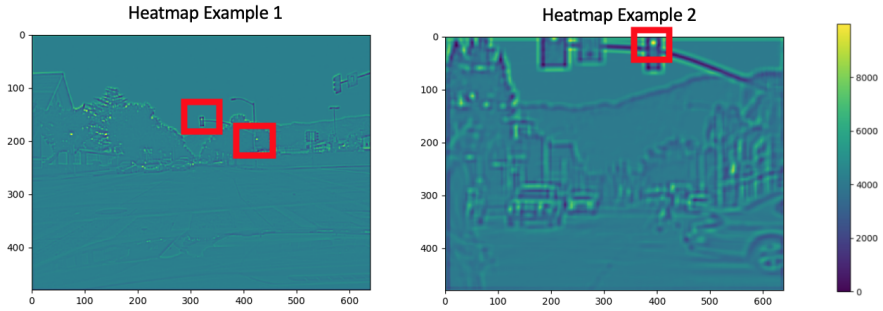


2.2 Deliverable 1.2

The template used by my matched filtering algorithm was a subset of size two from the following three templates. The three templates used were a standard daylight red light kernel, a nighttime red light kernel, and a small red light kernel. Here's a visualization of the predicted heatmap for two of the images. In order to visualize all heatmaps, set the plot_heatmap flag to true.







I localized box location by using a bounding box the size of the kernel. I also averaged the pixel locations of each red traffic light bounding box by clustering the identified pixel locations and selecting the cluster centers only. See the above section for more details on clustering.

2.3 Deliverable 1.3

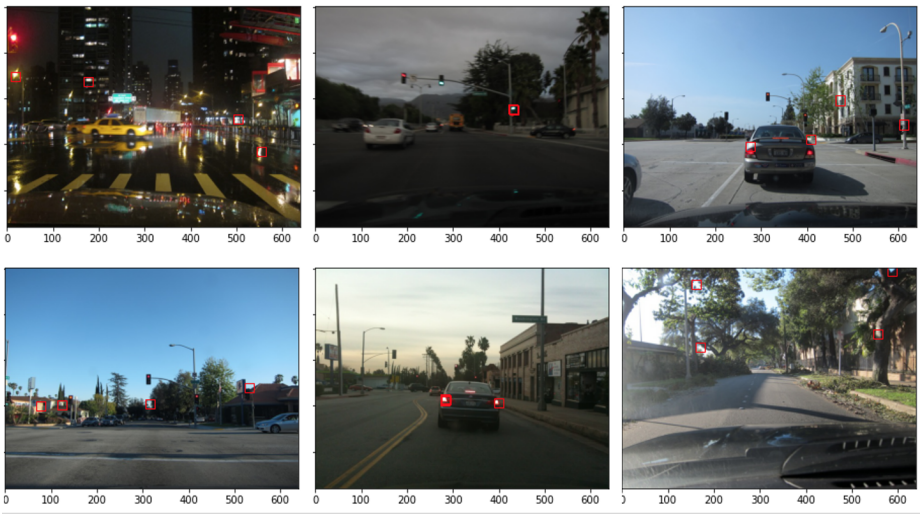
Here are a few examples (images with predicted bounding boxes) where the algorithm succeeded. It worked well on large, or standard sized, red lights. It

also worked well on small red lights! These images were easy for the algorithm,

because the lighting of the images were very similar to that used by the kernel filter, so it was easy to identify similar windows. The images were during the daytime, so the color was similar to the kernels.

2.4 Deliverable 1.4

Here are a few examples (images with predicted bounding boxes) where the algorithm failed. These images were hard for the algorithm, because the lighting

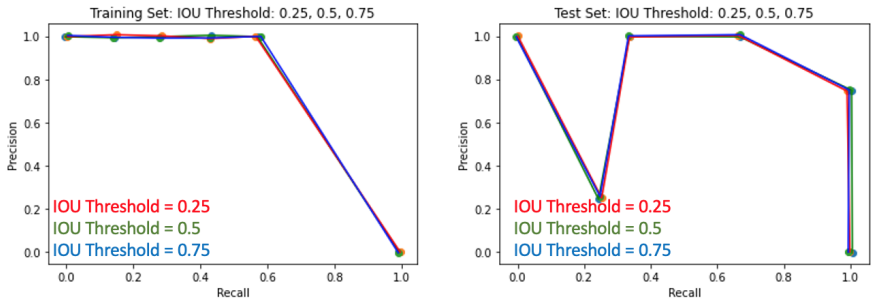


of the images were very different from that used by the kernel filter, so it was hard to identify similar windows.

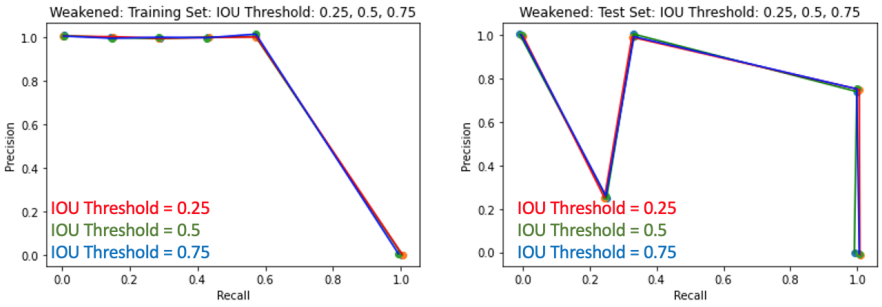
2.5 Deliverable 1.5

Below I generate PR curves for the training and test set. For a fixed IoU threshold of 0.5, 0.25, and 0.75, I vary the confidence score threshold and compute the precision and recall. Then I display all PR curves on the same axes. The algorithm didn't perform very well, and thus, the precision recall curves were all similar. In other words, for different IoU thresholds of 0.5, 0.25, and 0.75, the IoU threshold didn't influence the outcome, because for many images, there was no overlap between the detections and the ground truth bounding boxes anyways, so the precision-recall curves are generally the same. I added 0.005 of random noise to the precision recall curves to simply plot the curves in a way that we can see all three curves for each IoU: 0.25, 0.5, 0.75.
Next, I plot the precision recall curves on the training and test set using a weakened version of my algorithm. The weakened version didn't use the small kernel, and only used the standard sized kernel.

Precision Recall Curves for Algorithm



Precision Recall Curves for Weakened Algorithm



The effect of the IoU threshold is that the a higher IoU threshold shifts the PR curve outward, away from the origin, and closer to (1,1). A lower IoU threshold shifts the PR curve inward, towards the origin, because we are relaxing a constraint. However, for my algorithm, for different IoU thresholds of 0.5, 0.25, and 0.75, the IoU threshold didn't influence the outcome, because for many images, there was no overlap between the detections and the ground truth bounding boxes since the performance was quite poor, so the precision-recall curves were generally uninfluenced by the IoU threshold. The training curve was more steady, and the testing curve bounced around significantly more. The towards lower levels of recall, the training curve showed more clearly higher levels of precision, and vice versa. The weakened algorithm performs significantly worse. We can see this on the test set, because the elbow, or curve, of the PR curve is further out for the stronger algorithm, and closer to the origin for the weaker algorithm. The use of comparing to a less capable version of your own algorithm is to see if perhaps a weaker algorithm, that's less computationally expensive, would provide similar precision recall results, and if so, this indicates, that the weaker algorithm is sufficient performance-wise.

3 Deliverable 3

3.1 Deliverable 3.1

My code is located at https://github.com/mzhao98/caltech-ee148-spring2020-hw02. The algorithm is written on `run_predictions.py` at https://github.com/mzhao98/caltech-ee148-spring2020-hw02/blob/master/run_predictions.py.

3.2 Deliverable 3.2

The script for visualizing the output images with bounding boxes is in the file `visualize_results.py` at https://github.com/mzhao98/caltech-ee148-spring2020-hw02/blob/master/eval_detector.py.

Alternatively, you can also visualize some of the results using the Jupyter Notebook titled `Visualize_Results.ipynb` under the notebooks folder at https://github.com/mzhao98/caltech-ee148-spring2020-hw02/blob/master/notebooks/Visualize_Results.ipynb.

4 Deliverable 4

The final results JSON file is at https://github.com/mzhao98/caltech-ee148-spring2020-hw02/blob/master/preds_train.json and https://github.com/mzhao98/caltech-ee148-spring2020-hw02/blob/master/preds_test.json.