Object Detection

Interest Point Operator:

My interest point operator of choice was the Harris corner detector. I implemented the Harris corner detector in the following 7 steps:

1. Find the sobel gradients dx and dy of the input image using the function sobel_gradients() from hw1

2. Calculate the second moment matrix M as seen in the lecture slides using dx and dy from above and the function denoise_gaussian() from hw1

3. Calculate the corner response function for each image pixel using the formula given in class: det(M) - $\boldsymbol{\alpha}$*trace(M)^2

4. Apply non-max suppression to the resulting matrix by setting all non-maximum values within a sliding window (radius 3) to 0

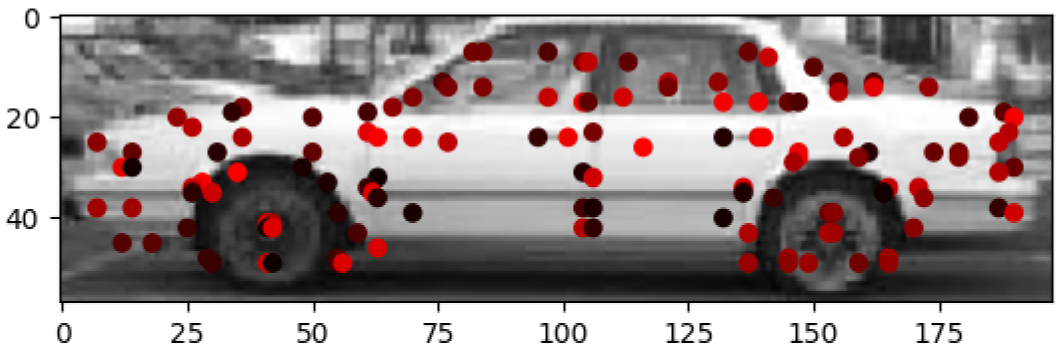
5. Filter the resulting matrix to include only the max_points values that are above a threshold (current default is 0.1). To only return max_points values, the max_points highest value is computed, and all values below that are dropped.

6. Give each pixel a score by regularizing its corner response value by dividing by the highest corner response value

7. Return the indices of the remaining values in the array and their corresponding scores

An important thing to note is that I set the first and last couple rows / columns of R to 0 in order to ensure that interest points are only found in regions where their descriptor can be calculated. This is done as a function of bin_width, which corresponds to bin_width in extract_features()

Sample Output:



Feature Descriptor:
I mostly implemented the feature descriptor in the way suggested in the homework. I did so in the following steps:

1. I denoise and take the sobel gradients of the input image using functions from hw1

2. I use those gradients to calculate a magnitude and angle matrix, again using functions from hw1

3. For each detected feature, a histogram (in this case a 3-dimensional array) is created and the magnitudes of the surrounding bin_width (default 5) pixels are placed into it based on their gradient directions. Each magnitude value contributes to 2 bins (interpolation based on gradient direction)

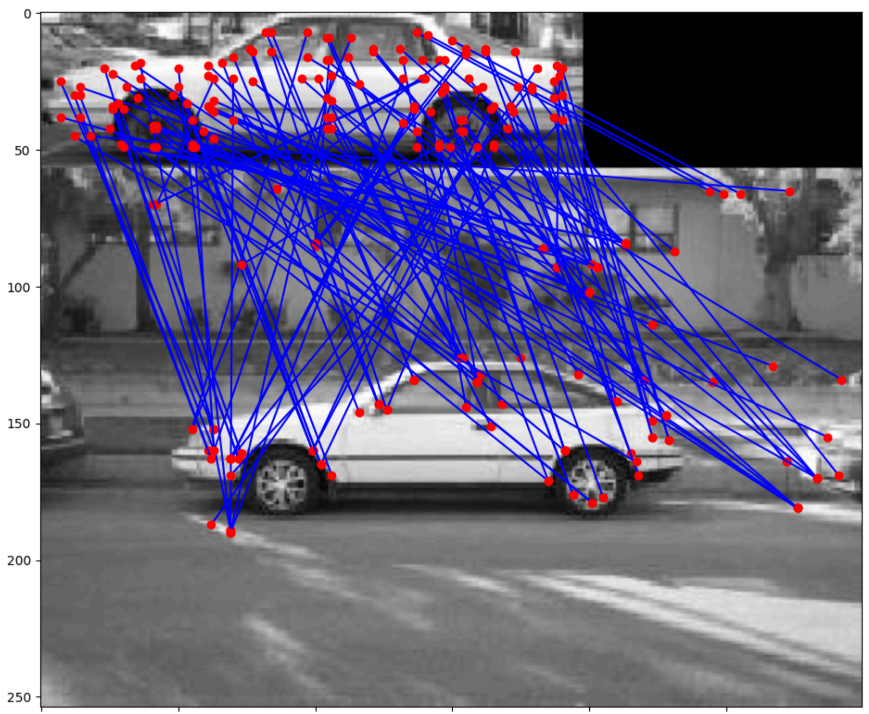
4. The resulting array is flattened, normalized, has its values above 0.2 clamped to 0.2, and renormalized, as suggested in class.

5. An array of these flattened and normalized features is returned

I chose a bin_width of 5, which corresponds to the bin_width in the find_interest_points() function.

Feature Matching:
To match features I iterate over every feature in feat0, and compute the feature in feat1 that has the closest euclidean distance to the feature in feat0. I then give it a score based on the nearest neighbor distance ratio (NN2/NN1).

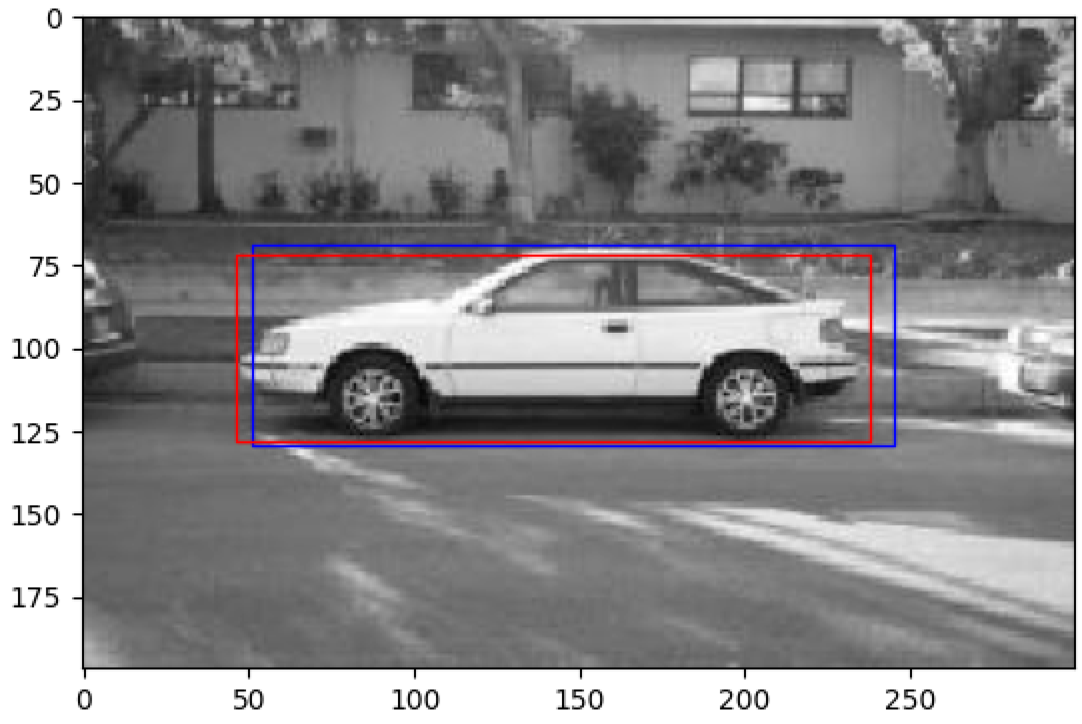
Sample Output:



Hough Transform:
To discretize the translation parameter space into bins, I created a 2d array, where each entry xi, yi represents a translation of between xi *3 and xi * 3 + 3 horizontally and yi * 3 and yi * 3 + 3 pixels vertically. I chose a bin size of 3 because it is small enough to be precise, while allowing some leeway for differently sized objects.I calculate the x and y translation of each match, and add the score of that match to its corresponding array slot. I also add the score of that match to 3 additional array slots, which correspond to the closest 3 translation parameters. This is in order to make the transform more robust to imperfect translations. After doing this for all matches, the x and y indexes of the highest value in the array are the translation parameters scaled down by the size of the bins, so I return the indices multiplied by bin_size.

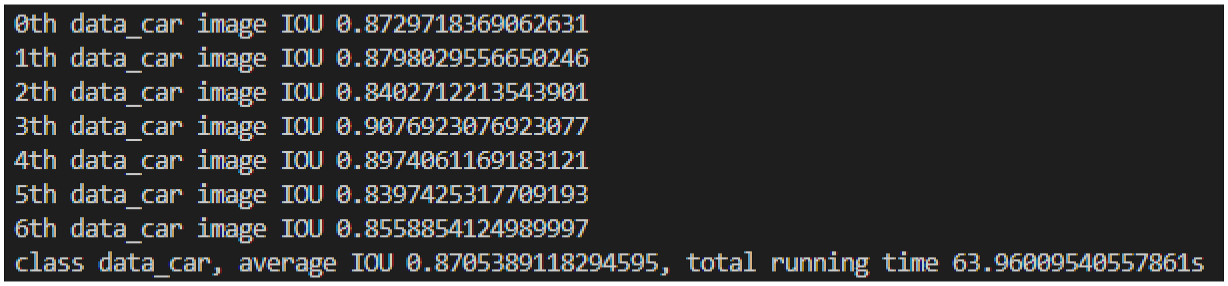
Object Detection:
To implement object detection, I first take the test image and find its features using the functions above. If the multi_scale parameter is set to true, I then iterate over every template image at every given scale (I chose to have 3 scales, 0.66, 1, and 1.5), extract features from the rescaled template image (the window size for feature detection is reduced to 1 for the 0.66 scale, kept at 3 for the 1.0 scale, and increased to 5 for the 1.5 scale in order to not have too many detected features clustered in one area), calculate the votes matrix, and save the translation parameters / template dimensions of the matrix which had the single highest entry. Using these translation parameters and template dimensions I calculate the bounding box.

Sample Output :



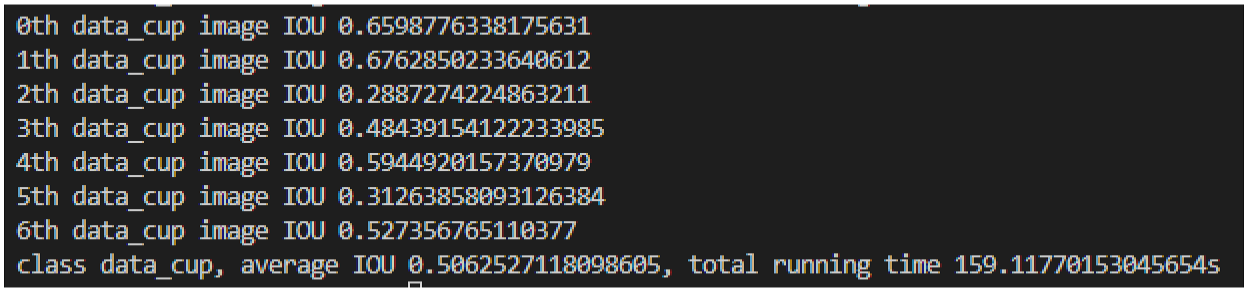
Single Scale Accuracy and Runtime Results:

- Car dataset:



```
0th data_car image IOU 0.8729718369062631
1th data_car image IOU 0.8798029556650246
2th data_car image IOU 0.8402712213543901
3th data_car image IOU 0.9076923076923077
4th data_car image IOU 0.8974061169183121
5th data_car image IOU 0.8397425317709193
6th data_car image IOU 0.8558854124989997
class data_car, average IOU 0.8705389118294595, total running time 63.96009540557861s
```

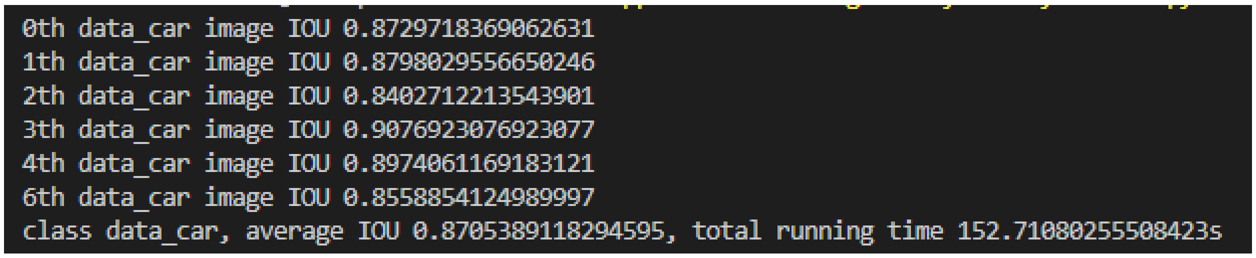
- Cup dataset:



```
0th data_cup image IOU 0.6598776338175631
1th data_cup image IOU 0.6762850233640612
2th data_cup image IOU 0.2887274224863211
3th data_cup image IOU 0.48439154122233985
4th data_cup image IOU 0.5944920157370979
5th data_cup image IOU 0.31263858093126384
6th data_cup image IOU 0.527356765110377
class data_cup, average IOU 0.5062527118098605, total running time 159.11770153045654s
```

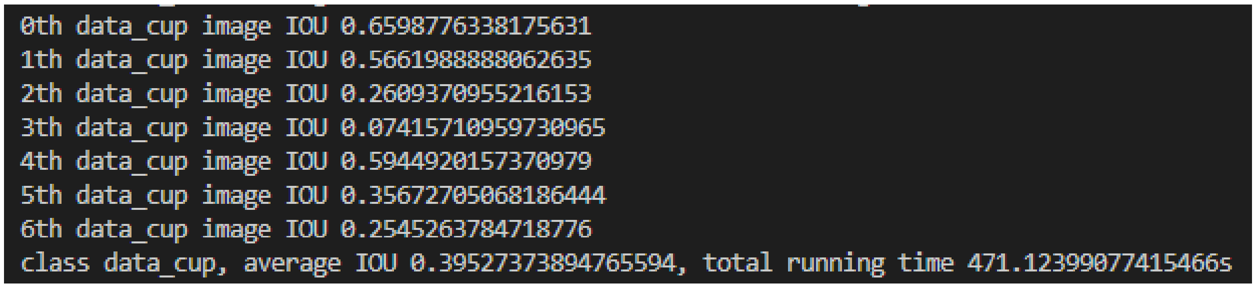
Multi Scale Accuracy and Runtime Results:

- Car dataset:



```
0th data_car image IOU 0.8729718369062631
1th data_car image IOU 0.8798029556650246
2th data_car image IOU 0.8402712213543901
3th data_car image IOU 0.9076923076923077
4th data_car image IOU 0.8974061169183121
6th data_car image IOU 0.8558854124989997
class data_car, average IOU 0.8705389118294595, total running time 152.71080255508423s
```

- Cup dataset:



```
0th data_cup image IOU 0.6598776338175631
1th data_cup image IOU 0.5661988888062635
2th data_cup image IOU 0.2609370955216153
3th data_cup image IOU 0.07415710959730965
4th data_cup image IOU 0.5944920157370979
5th data_cup image IOU 0.35672705068186444
6th data_cup image IOU 0.2545263784718776
class data_cup, average IOU 0.39527373894765594, total running time 471.12399077415466s
```

As seen by the above, the runtime for my multi-scale object detection is roughly 3 times the runtime of the single-scale object detection which makes sense as we are doing object detection across 3 different scales. The accuracy for the car dataset was exactly the same, which also makes sense as the test and template images were roughly the same size. For the cup dataset, however, the multi-scale algorithm actually performed roughly 10% worse than the single scale algorithm. I believe this is because false matches between patterns on the template cup and test cup background were more common at higher scales. Interestingly, with these results, it would be preferable to simply use single scale object detection for both datasets, as single-scale object detection performs at least as well as multiple-scale object detection and requires ⅓ of the computation.