CMSC 25040: Homework 2 Write-Up

Kameel Khabaz

February 3, 2023

1 Interest Point Operator

Since it was extensively discussed in class, I decided to implement the Harris corner detector as my interest point operator. Corner detection relies on finding a point in an image such that shifting a window around that point in any direction results in a large intensity change. This is characterized using the function

$$E(u,v) = \Sigma_{x,y} w(x,y) \left[I(x+u, y+v) - I(x,y) \right]^2$$

Here, $w(x,y)$ indicates a window function, while $I(x,y)$ indicates the image intensity and $I(x+u, y+v)$ indicates the intensity of the shifted window. Using a local quadratic approximation of $E(u,v)$ around the origin approximates to

$$E(u,v) \approx [uv] M \begin{bmatrix} u \\ v \end{bmatrix}$$

The second moments matrix $M = \Sigma_{x,y} \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$ characterizes corners, whereby once can define a corner response function $R = \det(M) - \alpha \operatorname{tr}(M)^2$ such that points with $R > 0$ may be characterized as corners.

In my implementation, I first calculated the Sobel gradients using the `sobel_gradients` function. Then, I calculated the second derivatives using element-wise multiplication. To perform the convolution of the shifted window most efficiently, I chose to apply a Gaussian window function (with $\sigma =$ `scale`, an input parameter) on $I_x I_x$, $I_x I_y$, and $I_y I_y$. This allowed me to use the `denoise_gaussian` function from homework 1, which also takes advantage of the separability of the Gaussian filter.

I then iterated through each pixel in the input image, calculated $\det(M) = I_x I_x I_y I_y - (I_x I_y)^2$ and $\operatorname{tr}(M) = I_x I_x + I_y I_y$, and then calculated $R = \det(M) - \alpha \operatorname{tr}(M)^2$. I saved points with a positive R value (which excludes edges but does include flat points, which have $|R| \sim 0$).

To further refine the corner detector and pick only isolated corners, I performed maximum suppression by retaining only corners whose R value is larger than R values of all of the neighboring

pixels in the appropriately-sized window (the scale parameter indicated the window size in each direction)

Finally, I applied a final thresholding step (keeping corners with the largest R values) in cases where more corner points are retained than the maximum number of interest points that can be returned. Each interest point is also assigned a corresponding score that is just the point's R value.

I first tested my interest point detector on one of the test images from lecture. As shown in Figure 1, I quickly discovered that the scale of the input image matters significantly in corner detection. My image had a very large scale, which caused R to not effectively indicate easily discernible corners. However, when I downsampled the image by a factor of 4, my R plot closely matched the one given in lecture and clearly indicated corners. Interestingly, when I changed the `scale` parameter to 4, the results were significantly better than with `scale = 1`, but they were not as good as the downsampled image. This observation influenced my design decisions for later problems in the assignment (particularly object detection), as the scale of an input image is important for corner detection.

Finally, Figure 2 shows results for an example template image and an example test image used for object detection later in the pipeline.

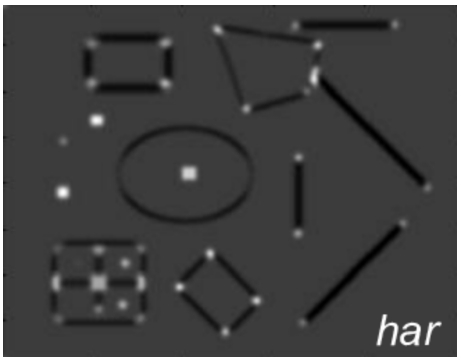
2 Feature Descriptor

The next step in this object detection pipeline was to implement a SIFT-like feature descriptor by binning orientation energy in spatial cells surrounding an interest point. The idea is that in order to match interest points between images for object detection, we must first characterize a set of features for each point. One idea for a feature descriptor of a point is to analyze the orientations of the image intensity gradients in the pixels surrounding that point and then put these gradients into a histogram. Then, each interest point will be have an associated histogram that characterizes the orientations of the image gradients around that point.

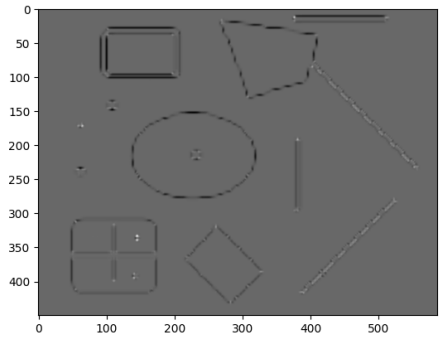
In my implementation, I consider a $3x3$ spatial grid of 9 cells surrounding an interest point, in which each cell has a width determined by the parameter `scale`. The gradient orientations in each cell are binned into a histogram with 8 orientation bins spaced evenly in $[-\pi, \pi)$. This yields a final 72-dimensional feature vector for each interest point.

First, I had to decide how to relate the `scale` parameter to the number of pixels in each cell. Since we compute a histogram of the orientations of the gradients of the pixels in each cell, this means that each cell, at a minimum, must contain several pixels. Furthermore, we want to use an odd scale so that we can easily center this window around a certain interest point. Thus, a natural choice is to set `cell_width = scale * 3`, such that each cell contains 9 pixels for `scale = 1`.

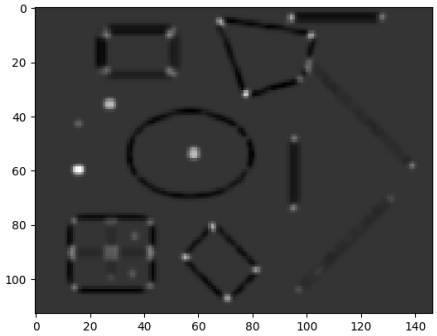
Then, the image must be padded. I used the `mirror_border` function from homework 1 to pad the image by the necessary amount. If one considers an interest point at the corner of an image, then the image needs to be padded by the amount `padd_amt = cell_width + (cell_width // 2)`. This is because the center cell (containing the interest point) needs to be padded by half of its



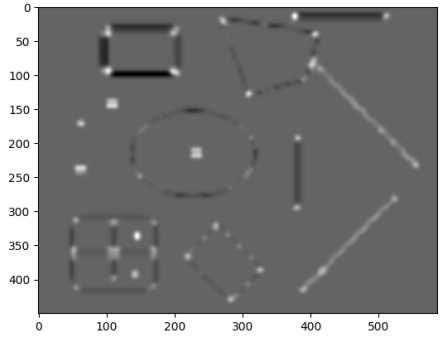
(a) Corner response function from lecture



(b) Corner response function with `scale = 1`



(c) Corner response function with downsampled image (by factor of 4)



(d) Corner response function with `scale = 4`

Figure 1: Corner responsiveness function R results with different scalings of a single input image. Image scale and the scale factor dramatically affect the resulting R values, which affects corner detection.

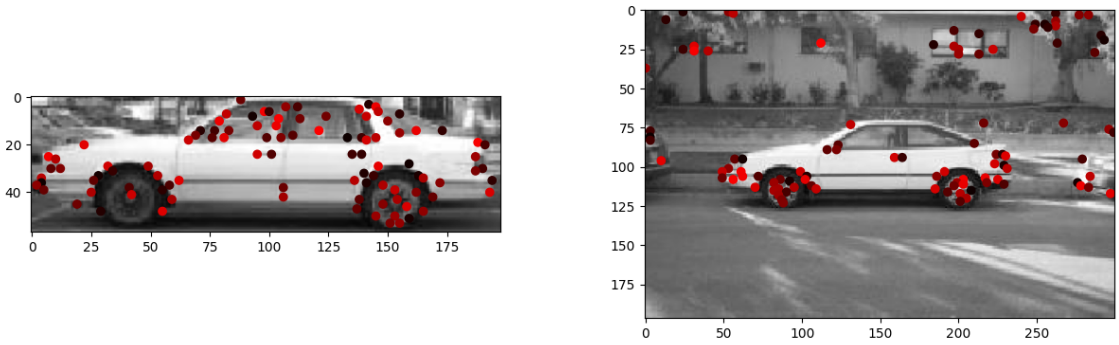


Figure 2: Detected interest points for example car template (left) and test (right) images. The template image has a corresponding mask that outlines the car. Relative score is scaled with red color intensity. Corner detection easily finds the car's wheels, windows, and bumpers in the template image. In the test image, these points are detected (although not as many points on the windows), but other points are also detected.

width, and we also need to pad by enough pixels to add the cells on the side.

After I padded the image, I applied a small blur using the `denoise_gaussian` function with `sigma = scale` to smooth out noise (since our feature descriptor is based on numerically computed derivatives, which are more sensitive to noise). Then, I computed the derivatives at each point and the corresponding magnitudes and angles.

I then iterated through all of the interest points. For each interest point, I extracted the corresponding window of the gradient magnitudes and angles. Then, I iterated through each of the nine cells. In each cell, I clamped exceedingly large gradients above the threshold $t = 0.2\Sigma_{u,v}G_{x+u,y+v}$, in which $\Sigma_{u,v}G_{x+u,y+v}$ is the sum of the gradient magnitudes in the cell. Finally, I computed the histogram bins of the gradient orientation angles, weighted by the clamped gradient magnitudes and normalized to a probability density function (integral of 1). The histograms for the cells are concatenated into one 72-dimensional feature descriptor per interest point.

3 Feature Matching

I then performed feature matching between two sets of feature descriptors extracted from a template image and a test image. Specifically, I match each interest point in the template image with a corresponding interest point in the test image according to the associated feature descriptors of the two interest points. Furthermore, each match is ranked by a real-valued score corresponding to the quality of the match.

I iterate through each interest point in the template image and perform a chi-square two-sample test to compute the distances between the histograms of that interest point with all of the interest points in the test image. This is all performed using vectorized operations on Numpy arrays to

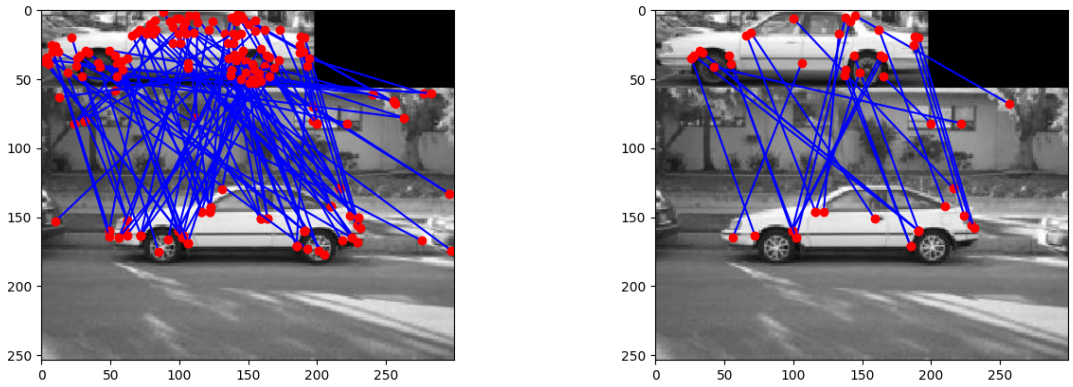


Figure 3: Matched features between template and test images of a car. The left image shows all of the feature matches, and the right image shows only the most confident matches ($NNDR < 0.8$). While many matched points are reasonably sensible, other points are falsely matched.

minimize explicit `for` loops and to maximize efficiency. I used the chi-squared distances instead of a simple Euclidean distance because we are comparing probability distributions and not coordinate points. The test computes $\chi^2 = \Sigma_{i=1}^{k} \frac{(K_1 R_i - K_2 S_i)^2}{R_i + S_i}$, in which R_i and S_i are observed histogram values for bin i out of a total of $k = 72$ bins and $K_1 = \sqrt{\frac{\Sigma_{i=1}^{k} S_i}{\Sigma_{i=1}^{k} R_i}}$ and $K_2 = \sqrt{\frac{\Sigma_{i=1}^{k} R_i}{\Sigma_{i=1}^{k} S_i}}$. Then, the interest point in the template image is matched to the corresponding interest point in the test image with the smallest distance χ^2 value. The associated score of the match (assessing match quality) is defined using the inverse of the nearest neighbor distance ratio $NNDR = d(NN1)/d(NN2)$, in which $d(NN1)$ is the distance of the nearest neighbor (the matched point) and $d(NN2)$ is the distance of the second nearest neighbor. I used the inverse because a larger $NNDR$ indicates a less confident feature match (since the distance to the next nearest point is not much larger than the distance to the nearest point) and a smaller $NNDR$ indicates a more confident match.

Example results are shown in Figure 3. As can be seen, many of the matches are reasonably connect, aligning the car's bumpers, wheels, and windows. However, there are some matches that are clearly incorrect, such as those that align different parts of the cars, those that align the template car with the partial images of the cars on the sides, and those that align parts of the car in the template image with parts of the background in the test image.

4 Hough Transform

Once we have a template and a test image with matched interest points, we need find the position of the template image in the test image. Assuming that two images of the same scene are primarily related by translation, we can use the feature correspondence to estimate a translation vector $\mathbf{t} = [t_x \; t_y]$. This is implemented using a Hough transform that discretizes a parameter space and tallies votes for translation parameters based on the pairs of matched features and with a

weight based off of the match scores. The bin in the parameter space with the highest value is used to predict $\mathbf{t}$.

In my implementation, I first calculated $t_x = \Delta x$ and $t_y = \Delta y$ for each pair of matched interest points. I then defined my parameter space for the Hough voting using the minimum and maximum t_x and t_y values from the matched features. The parameter space is discretized into 20 bins for each variable, which I found provided better results than discretizing the space into less or more bins (I tried 15 and 25 bins). Then, I binned the matched interest points for all of the matches, incrementing the value of the bin by the associated match score. Then, I found the best t_x and t_y by finding the bin with the highest sum of corresponding match scores.

5 Object Detection

The object detection function implements the previous functions from this assignment. First, I found the interest points and extracted the associated features for the test image. For the interest point extraction, I set `scale = 1` and `max_points = 200`.

Then, I iterated through each of the template images and associated masks. I implemented a multi-scale strategy to recognize images of different scales. The object in a template image may be located inside of a test image, but the scale may be different in size. Furthermore, as I showed in Figure 1 for the interest point calculation, the identification of interest points is significantly influenced by the scale of the image.

Because of this effect, I implemented my multi-scale strategy by scaling the template images. I scaled them by factors $(0.5, 1, 2)$ (image downsampled by 2, original image, image upsampled by 2), accounting for the possibility that the object in the template image may be half or twice the size as in the test image. Thus, scaling allows me to better detect interest points and match objects between the template images and the test image. Images are downsampled using the `smooth_and_downsample` function, while binary masks are sampled using array indexing (which is more efficient). For upsampling, I used the `bilinear_upsampling` function. Furthermore, I assumed that the entirety of the template image must be contained within the test image, such that if a template image is larger than the test image after upsampling, the upsampled image is not analyzed. This further saves time by preventing unnecessary computations.

For each (scaled) template image, I found the corresponding interest points (keeping `scale = 1`), matched the features to the test image, and computed the translation vector using the Hough transform. Since template images should theoretically contain fewer interest points than the test images (since test images contain background pixels), I set `max_points = 100`. Furthermore, I saved the maximum bin score from the Hough voting matrix as a score value of the object detection quality using the associated (scaled) template image.

Since we may have a large database of template images that contain potential objects in the test image, we would only want to use the transformation vector from valid instances of the object. This means that we want to use the translation vector for only the most confident feature match

instead of taking some kind of average. As such, I define the bounding box using the translation vector with the highest associated score. The bounding box is defined as $x_{\min} = t_x$, $y_{\min} = t_y$, $x_{\max} = t_x + s_x$, and $y_{\max} = t_y + s_y$. Here, (s_x, s_y) defines the scaled size of the chosen template image. If, for some reason, the corresponding bounding box is located outside of the test image, I translate the bounding box back to the image location.

Figures 4 and 5 compare my bounding box with a reference bounding box for several example images of cars and cups. For the simple car images, the object detection is very good, as the predicted bounding box aligns with the ground truth box. The results are slightly less impressive for the (more challenging) images of the cups. One challenge with these images may be that the objects take up much larger portions of the test images, and the template image sizes are much smaller (as indicated by the smaller blue box).

Figure 4: Object detection result on test car images. The blue boxes show my result, and the red boxes show the ground-truth result.

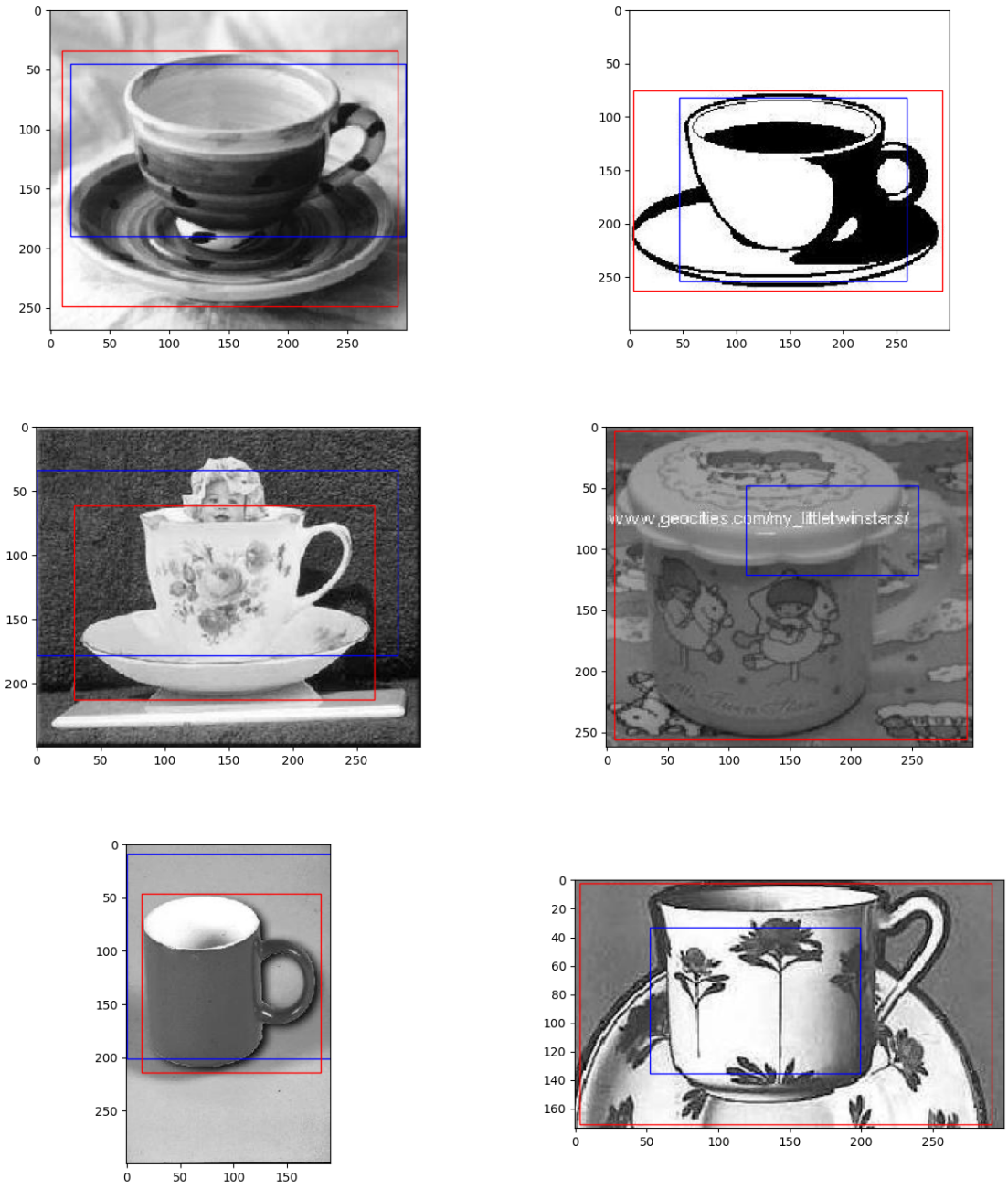


Figure 5: Object detection result on test cup images. The blue boxes show my result, and the red boxes show the ground-truth result.

5.1 Single-Scale vs Multi-Scale Analysis

I performed the given benchmark tests using my single-scale and multi-scale implementations. The benchmark tests object detection for sets of template and test images. For each computed bounding box, it reports an IOU metric that measures the intersection area divided by the union area of ground-truth and predicted bounding boxes, as well as the running time. For the multi-scale implementation, I got the following results:

```
0th data_car image IOU 0.8199474127587879
1th data_car image IOU 0.8666079683028834
2th data_car image IOU 0.7462941847206386
3th data_car image IOU 0.8374687611567297
4th data_car image IOU 0.8036949801266009
5th data_car image IOU 0.8181373348556511
6th data_car image IOU 0.6268026485606867
class data_car, average IOU 0.7884218986402827, total running time 54.302712202072144s
0th data_cup image IOU 0.6483668543845535
1th data_cup image IOU 0.6762850233640612
2th data_cup image IOU 0.5661739123398952
3th data_cup image IOU 0.14265544393157753
4th data_cup image IOU 0.619190193391914
5th data_cup image IOU 0.31027885202523914
6th data_cup image IOU 0.11292501999283017
class data_cup, average IOU 0.4394107570614387, total running time 150.00016331672668s
```

I got the following results using a single-scaling approach (no template image scaling):

```
0th data_car image IOU 0.8199474127587879
1th data_car image IOU 0.8666079683028834
2th data_car image IOU 0.7462941847206386
3th data_car image IOU 0.8374687611567297
4th data_car image IOU 0.8036949801266009
5th data_car image IOU 0.8181373348556511
6th data_car image IOU 0.6268026485606867
class data_car, average IOU 0.7884218986402827, total running time 46.141966104507446s
0th data_cup image IOU 0.6483668543845535
1th data_cup image IOU 0.6762850233640612
2th data_cup image IOU 0.5661739123398952
3th data_cup image IOU 0.5144426443276564
4th data_cup image IOU 0.619190193391914
5th data_cup image IOU 0.7169373549883991
6th data_cup image IOU 0.4661222456710666
class data_cup, average IOU 0.6010740326382209, total running time 119.2093276977539s
```

The results are extremely similar between the single-scale and multi-scale approaches. This makes sense because the final translation vector is only based on the most confident translation vector from all of the template images, so data from only one scaled template image is used. Fur-

thermore, while the single-scaling approach is faster than the multi-scaling approach, the efficiency difference is not that large (46 seconds versus 54 seconds for the car images, and 119 seconds versus 150 seconds for the cup images).

Throughout my implementation, I tried to keep my code as efficient as possible by using vectorized operations using optimized Numpy functions.