Instance-Level Object Detection

Sounak Paul

June 7, 2021

Contents

1 Introduction

This is the report of my Computer Vision (TTIC 31040) course project. This project aims to develop a system that can detect a given object (i.e. a particular instance) in a test image. To elaborate, a reference image of the object to search for is provided, and we will be operating under the assumption that the object is not occluded in the reference image, and is also the only object there. The reference image trivially defines a bounding box for the object of interest. An intuitive idea of detected instance of the object in the test image is to specify how to overlay the reference image over the test image so that the detected instance would align with the reference instance. The algorithm implemented in this project outputs:

- The test image with bounding boxes around the detected instances. (Note that my algorithm can detect multiple instances in the test image)
- The coordinates of the corners of those boxes.
- The number of detected instances.

In the next section, we will go through a step by step description of the implementation of my algorithm, and look at how it performs on two different images.

2 Implementation

In this section I will discuss the details of my implementation, and elaborate on the choice of hyperparameters. Four helper functions were written, which were pretty simple and straightforward, hence I wont explain too much:

(1) REMOVE_ARRAY_FROM_LIST: Removes a given numpy array from a given list of numpy arrays.

(2) NEWLINE: Draws a line or line segment connecting two given points on Euclidean plane.

(3) ROTATE_AND_SCALE: Rotates a given point clockwise about a given origin by a given angle (in degrees), and also scales the distance from the rotated point to the origin by a given factor.

(4) PLOT_AND_PRINT_RESULTS: Given the test image and a list of lists containing corners of bounding boxes, this function prints the number of instances detected, along with the four corners of the bounding box corresponding to each instance, and also plots the test image with bounding boxes around the detected objects. It also manages the case where bounding box may spill out of the image, and plots the entire spilled out bounding box, by setting the x-limit and y-limit according to the maximum and minimum coordinates of the image as well as the bounding boxes.

We now move on to the main algorithm, which differs slightly in various ways from the the recommended steps outlined in the question paper.

2.1 Database Lookup

- First, I wrote a function CREATE_DATABASE that takes a given image, converts it to grayscale, and returns the keypoints and descriptors obtained using SIFT (by the DETECTANDCOMPUTE function). Though I have added functionality for SURF and ORB as well, I have used SIFT descriptors everywhere.

- Next, I wrote the function DATABASE_LOOKUP, that takes a keypoint and descriptor from the test image to look up, along with the list of keypoints and descriptors of the reference image, and finally, a parameter for the ratio test (as mentioned in David Lowe's paper) for determining robustness of the matches. Here, first we use brute force matcher to extract the KNN matches (with $k = 2$), and then calculate the ratio of the distances of the best match with that of the second best match. If the ratio exceeds the given ratio test coefficient, the match is not considered good, and the function returns an empty list. Otherwise, it is considered a good match, and the corresponding point on the test image is referred to as the origin.

- Now, following the terminology introduced in Page 4 of the question paper, we denote the lower left corner point of the reference image as c_1 and the upper right one as c_2. We find their corresponding points in the test image by translating c_1 and c_2, and then we use the function ROTATE_AND_SCALE on them, to rotate them around the origin by an angle $\theta - \theta'$ and scale them by s/s' (Here, θ, θ', s, s' are the orientations and diameters of the keypoints of the reference and test image respectively). This gives us the corresponding c_1' and c_2' (i.e. the lower left and upper right corner of a potential bounding rectangle) from one good match. The function DATABASE_LOOKUP finally returns a list of four 2-tuples, namely the coordinates of the origin, the coordinates of its corresponding point in the reference image, and the coordinates of c_1' and c_2' as well. Note that no Hough Voting is conducted in this step, and that these are simply the coordinates parametrizing a "potential" bounding rectangle.

2.2 Hough Voting

- The parametrization of the bounding rectangle is done by specifying its lower left and upper right coordinates, since having known the height to width ratio of the reference image, this helps recover the entire rectangle. This leads to a 4-dimensional Hough Voting space.

- I wrote the function HOUGH_VOTING which takes as input, a list of lists of four 2-tuples, called matches_list (i.e. the outputs of DATABASE_LOOKUP for different query keypoints), and runs a clustering algorithm very similar to the spirit of accumulator arrays. It also takes two parameters, i.e. vote_threshold and cluster_size. First, we traverse through matches_list, and fix a list of four 2-tuples, say l_{fixed}. Then we compare that with every other element of matches_list, say $l_{compare}$. Note that the third and fourth 2-tuple in every of those lists, contain the coordinates of c_1' and c_2' corresponding to a robust match of SIFT descriptors. Thus, we check whether

$$\|l_{fixed}[2] - l_{compare}[2]\|_2 \leq \text{cluster\_size},$$

$$\text{and } \|l_{fixed}[3] - l_{compare}[3]\|_2 \leq \text{cluster\_size}$$

  If both hold, then we add $l_{compare}$ to the same cluster as l_{fixed}. After traversing across all available $l_{compare}$, we look at the cluster for l_{fixed}. If the cardinality if the cluster is lower than vote_threshold,

we discard the cluster, and move on to a different l_{fixed}. However, if the cardinality has at least vote_threshold elements, we call it a valid cluster (since it contains points corresponding to a valid detection of a bounding rectangle), and then we remove this cluster from matches_list, and keep iterating.

- The function HOUGH_VOTING finally outputs a list of clusters, where every cluster is a list of list of four 2-tuples, the first two of which are the matched points in the test and reference images respectively. All the matched points in a particular cluster are corresponding points of one detected instance. The hyperparamters of this function is very important. The first hyperparameter is vote_threshold, which dictates the minimum number of votes for a cluster to deemed valid. The higher the vote threshold, the better our affine transformation (in the upcoming subsection) will be, but at the same time, our algorithm will become less sensitive. Thus in small images, or images where the reference object is too small, or unclear, or in a weird angle - any situation where we expect less number of good keypoint matches, we should lower the vote threshold. While that might lead to dubious matches, we have another hyperparameter cluster_size to tackle that. For small images (or images where the reference object is small), we should impose a lower cluster_size, and a higher one for large images. Again, it is a good idea to increase cluster_size for images where we expect low number of SIFT keypoint matches.

2.3 Affine Transformation and Duplicate box removal

- We now come to the last few functions in my implementation. I wrote the function PRELIMINARY_BOUNDINGBOX_CORNERS which takes as input the test image, reference image, vote_threshold, and cluster_size. First we find the keypoints and descriptors of both images, and use DATABASE_LOOKUP (with ratio test coefficient 0.8) repeatedly, to output the list database_results containing good matches along with their corresponding c_1', c_2'. We then call the function HOUGH_VOTING to output the list of hough_clusters.

- Then, for every hough cluster, we use the opencv function estimateAffinePartial2D with the method RANSAC (and threshold = 3), to robustly estimate the affine transformation matrix mapping the points of that cluster on the reference image to the corresponding points on the test image. Once we have the matrix (for one cluster), we use it to map the four corners of the reference image to the test image, and get the corners of the bounding box corresponding to one detection). Repeating over all the clusters gives us all the detections. However, sometimes, despite choosing good hyperparameters, the algorithm often detects multiple instances which are simply duplicates, i.e. many of the detected clusters were of the same instance, but with very slightly differing bounding boxes. To tackle that, I wrote the function DUPLICATE_BOX_REMOVE, where I look at the list of corners of all the detected bounding boxes, and cluster them (somewhat similar to my hough voting function). When the Frobenius distance between two matrices (containing coordinates of the 4 corners of bounding boxes) is less than a given parameter duplicate_thres, they are considered duplicates. Hence we cluster these bounding boxes on basis of their Frobenius distances, and in the end, we output a list (final_warped_corner_list) of the arithmetic means of those corners for every cluster. Hence final_warped_corner_list contains the corners of the bounding boxes of the finally detected instances.

- Finally, I wrote the function INSTANCE_LEVEL_OBJECT_DETECTOR, that inputs the paths for the test and reference images, and two hyperparameters: vote_threshold and sensitivity. This function calls all the previously written functions one by one in the right order, and ends up with final_warped_corner_list, i.e. the result of the DUPLICATE_BOX_REMOVE function. Then, the function PLOT_AND_PRINT_RESULTS is called, to finally output the corners of detected bounding boxes, and plot them on the test image. While vote_threshold is the same hyperparameter that was used for hough voting; sensitivity is a new hyperparameter which does not require you to know the size of the test image (which generally dictates the cluster_size to select for hough voting). Instead, the hyperparameter cluster_size will be taken to be

$$\frac{1}{100} \times \text{sensitivity} \times \min\{\text{test image height},\ \text{test image width}\}$$

Generally, the sensitivity value of 10, gives us good detections. The duplicate threshold parameter for DUPLICATE_BOX_REMOVE is set to be four times the hough voting cluster_size, and seems to give us the best results. Thus, varying only two parameters, i.e. vote_threshold and sensitivity, seem to be good enough to give us reasonably good bounding box detections.

3 Qualitative Evaluation and Limitations

In this last section, we talk about the results of our algorithm when run on two different images: Stop sign and croissant. The stop sign images were provided to us, and the croissant images are photos that I myself clicked with my android cellphone, and are much larger images.

- **Stop sign images:** For the images 'stop1', 'stop3' and 'stop5', our algorithm worked very well with vote_threshold = 5 and sensitivity = 10. A sensitivity of 10 implies that that cluster size is one-tenth of the width of the image, which in most cases, works reasonably well. For 'stop4', keeping vote_threshold = 5, I had to increase sensitivity to 20, since the object took up the majority of the image. The biggest challenge however, was 'stop2'. Since it has a lot of stop signs (some of them overlapping) at a large angle with the screen, there were very few keypoints that contributed to a valid detection, especially for the two biggest signs. In my ipynb, I have a section "Role of Hyperparameters", where I took three pairs of hyperparamters to illustrate this. For (vote_threshold=3, sensitivity=15), we have two detections, and the bounding boxes look reasonable. (vote_threshold=4, sensitivity=20) gives us three detections, though the bounding box of the biggest stop sign is too large. Increasing the threshold to 5, with the same sensitivity, gives us just one detection, since the rest of the the stop signs simply did not have 5 or more keypoints contributing to the Hough voting, to form a cluster.
- **Croissant images:** Unlike the stop signs, these five images were much easier to work with, possibly because the reference image has a lot more conspicuous interest points. Barring 'croissant4', all the rest did a very good job with vote_threshold = 10 and sensitivity = 10. Because these images are large, I chose a higher vote threshold than the stop signs case. Only for the image 'croissant4', did I have to further increase the threshold to 20, probably because the object in the test image was very unclear, hence for a lower threshold I was getting a lot more detections, many of which were completely incorrect.

Now we come to some of the limitations I observed from my above evaluations, and some potential methods for improvement:

- One obvious limitation of our method is that we are having to manually select the hyperparameters for every image. With more time and effort, hyperparameter tuning could be done. One way to do so might be to partition the image into smaller images by taking the neighbourhoods of each of the detected instances, and then running our algorithm on those smaller images again, to confirm whether that detection was indeed valid, and change hyperparameters accordingly.
- Secondly, I only considered affine transformations. Projective transformations can also be considered, but it seemed to be much more susceptible to the scale of the images, and the angle the object makes with the principal axis, so the projective transformation bounding boxes turned out to very elongated at times. Even that could be tackled by using the strategy I outlined in the previous point (of breaking the image into smaller images around the detected instances, and running the algorithm, this time using homographic transformations, on them).
- Finally, the biggest problem that I faced was that in some cases, where there were very low number of keypoints on the detected object, I had to set the voting threshold very high. This is because sometimes SIFT returns the same keypoint multiple times, hence even for a threshold of 5, it is often only one distinct point. The algorithm then throws errors since it needs at least 2 distinct points (possibly more because of RANSAC) to calculate the affine transformation. The only strategy then is to increase the vote_threshold (as well as the sensitivity, to ensure we do not miss the object altogether). That does seem to work most of the time.

To conclude, I think my algorithm does a decent job given the limited time I had for my project.