Division of Electronics and Information Engineering

ADVANCED IMAGE PROCESSING (COMPUTER VISION)

ANALYSIS REPORT PROJECT 3

October 15th, 2020

Reported by: Vuong Minh Phu (ID: 202050798)

Instructor: Prof. Lee Hyojong

In this report, I will explain each step of the SIFT algorithm, describe the code from the repository PySIFT, and finally discussion. The utility program will be submitted in a different report.

SIFT (Scale-Invariant Feature Transform) algorithm detects and describes local features in an image. These features are robust to noise and brightness invariance which are also useful for object detection.

The SIFT algorithm has four main steps:

a. Scale-Space Extrema Detection
b. Keypoint Localization
c. Orientation Assignment
d. Keypoint Descriptor Creation

Scale-Space Extrema Detection

The purpose of this step is to combine different scale of Gaussian filter on a particular image

The characteristic scale of a feature can be detected using a scale-normalized Laplacian-of-Gaussian (LoG) filter.

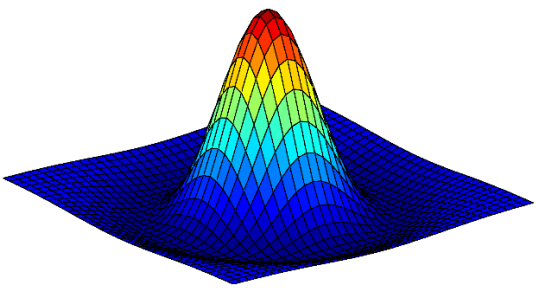

Figure 1: Laplacian of Gaussian filter. Source: https://medium.com/@lerner98/implementing-sift-in-python-36c619df7945

The LoG filter is peaked at the center, then slightly negative, and up to a distance which is characterized by std, sigma, will be zero. Because of that reason, LoG filter is most highly activated by a circle or blob with radius proportional to sigma.

However, it is computationally expensive to calculate the LoG filter. Therefore, an efficient function is used to compute the Laplacian Pyramid (Burt & Adelson, 1983). By utilizing the difference of two Gaussians (DoG) with similar variance, we can approximate the scale-normalized LoG.

DoG pyramid

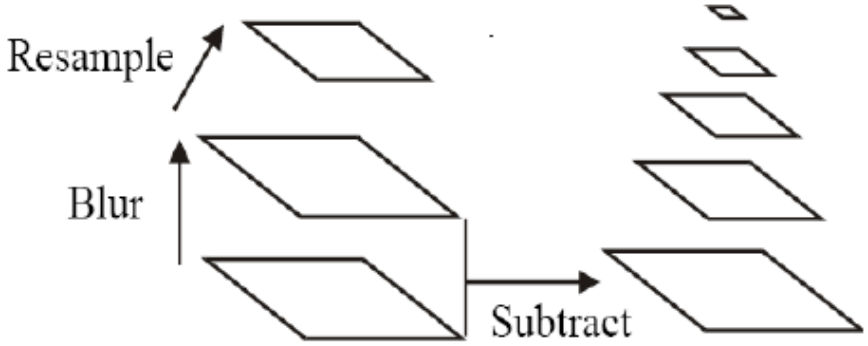

Figure 2: Scale Space. Source: http://vision.eecs.ucf.edu/faculty/shah.html

In order to generate the DoG pyramid, we first blur the image with different scales, then subtract the two images to get the approximation of the LoG. After that we resample the original image and do the same. We call a set of blur image is an octave. In the original paper, the author uses 5 images for one octave.

In other words, octave is a set of images were the blur of the last image is double the blur of the first image

s: denotes the number of images we want in each octave. The sigma for Gaussian filter is chosen to be 2^(1/s).

We need to produce s+3 images (including the original image) due to the loss of one image during the subtraction.

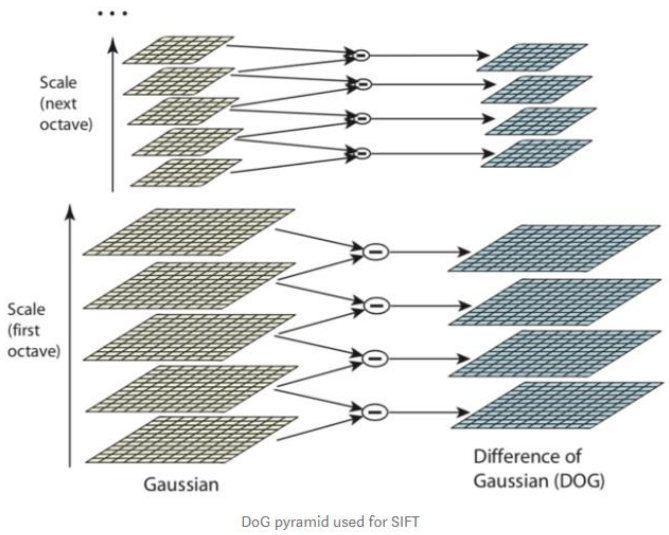

Figure 3: DoG pyramid. Source: https://medium.com/@lerner98/implementing-sift-in-python-36c619df7945

Scale space peak detection

To detect the peak, we look at each point, exam their 3x3 neighborhood at that scale, also their neighborhood at lower and upper scales, and select a pixel (X) if it is larger/smaller than the rest pixels.

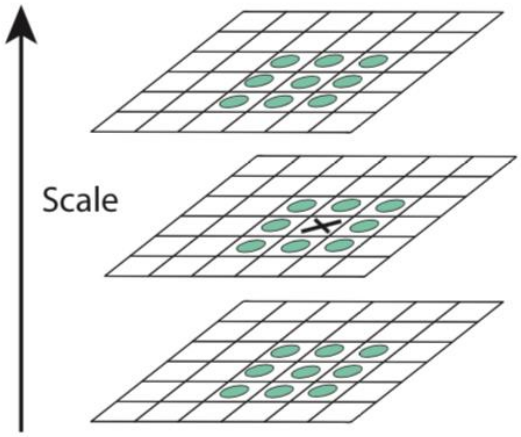

Figure 4: Peak Detection. Source: https://medium.com/@lerner98/implementing-sift-in-python-36c619df7945

Up to this step, we will generate a lot of extremes, containing not only the interest points, but also the edges. So, in the next steps, we need to subtract the unwanted points.

Keypoint Localization

There are some initial outlier rejections such as low contrast candidates, poorly localized candidates along an edge.

Basically, to extract the keypoint we need to do the following steps:

1. Compute the subpixel location of each keypoint
2. Throw out that keypoint if it is scale-space value at the subpixel is below a th
3. Eliminate keypoints on edges using the hessian around each subpixel keypoint

Initial outlier rejection

D(x) is a function of three variables: x,y and sigma. We will approximate D by using the second order of Taylor expansion.

$$D(\mathbf{x}) = D + \frac{\partial D}{\partial \mathbf{x}}^T \mathbf{x} + \frac{1}{2}\mathbf{x}^\mathbf{T}\frac{\partial^2 D}{\partial \mathbf{x}^2}\mathbf{x}$$

Taking derivative of this equation with respect to x and setting it equal to zero yields the maxima/minima (subpixel offset for the keypoint).

$$\hat{\mathbf{x}} = -\frac{\partial^2 D}{\partial \mathbf{x}^2}^{-1}\frac{\partial D}{\partial \mathbf{x}}.$$

This offset is added to the original keypoint location to achieve subpixel accuracy. Then the value of D(x) at $\hat{x}$ must be maintain larger than a threshold, otherwise we will discard it

Further outlier rejection

We assume that the DoG as a surface, then we compute the principal curvatures (PC).

Along the edge, one of the PC is very low while the scaled-normalized LoG will create high-contrast responses on both corners and edges. To discard the edge, we must use the hessian matrix to compute the subpixel offset. The Hessian matrix contains Second derivative of x,y and xy We can remove the outliers by evaluating the ratio of the trace and the determinant

$$\mathbf{H} = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix} \quad \begin{aligned} Tr(H) &= D_{xx} + D_{yy} = \lambda_1 + \lambda_2 \\ Det(H) &= D_{xx}D_{yy} - D_{xy}^2 = \lambda_1\lambda_2 \end{aligned}$$

$$\frac{Tr(H)^2}{Det(H)} = \frac{(r+1)^2}{r}$$

Where λ_1, λ_2 are the eigenvalues of the matrix H

Base on the ratio above, it will increase with respect to r. So, we will apply the threshold for r. Generally, r>10 we will remove those points because it might potentially an edge

Orientation assignment

Now we want to find the orientation of an interest point, this will help us get the rotation invariance. If we find a dominating orientation interest point, then we can align all other points in the neighborhood to that direction.

We need to compute the derivatives, gradient magnitude and the gradient direction at the scale of key point (x,y). (Sobel or Prewitt filters)

Next, we are going to look at a group of interest points around the examining point, we will create a histogram of 36 bins (each bin is 10 degrees) of orientation.

Once we have the histogram, we assign that keypoint the orientation of the maximal histogram bin. However, if a point has the value of gradient direction within 80% of the peak, we also classify it as keypoint.

Local descriptor creation

We will compute the relative orientation and magnitude in a 16x16 neighbors around each keypoint, then split up into 16 4x4 sub-regions

The gradients (in polar coordinates) of each sub-region are then binned into an 8-bin histogram

Finally, all these histograms are concatenated into a 4x4x8 = 128 element long feature vector

The feature vector is then normalized, thresholded and renormalized to try and ensure invariance to minor lighting changes.

Describe the code

Scale-space extrema detection

- ***DoG approximation***

Firstly, the author creates a function to calculate Gaussian filter (gaussian_filter.py). This function is straight forward, like the one in the previous project.

Then we need to create Gaussian octave by the function generate_octave. This function takes 3 parameters: init_level, s, and sigma. In this function, first, he creates a list with 1 element init_level then repeatedly convolve the last element of this list with the same filter and append it to the octave list. This will create an effect of creating a stack of images with different scale of gaussian filter.

```
for _in range(s+2):
   next_level = convolve(octave[-1], kernel)
   octave.append(next_level)
```

Note that here we will iterate s+2 times where s is the images we want in each octave. The reason we want to loop s+2 times is when we approximate the DoG, we lose one image per step. [-1] here denotes the last image in the list. The author claims that he used s=5, sigma =1.6 according to the original paper.

Next, he will generate the gaussian pyramid by the function generate_gaussian_pyramid with 4 parameters: image, number of octaves, s, and sigma. When generate the pyramid we need to use the third to last image as the base for the next octave since that is the one with a blur of 2*sigma. That is why in the code he use *im = octave[-3][::2, ::2]* indicating we will use the third element from the end of the list, and forward with the step of two in both x and y direction.

After that he creates the DoG pyramid by using the above functions.

```
def generate_DoG_octave(gaussian_octave):
 octave = []
 for i in range(1, len(gaussian_octave)):
   octave.append(gaussian_octave[i] -- gaussian_octave[i-1])
 return np.concatenate([o[:,:,np.newaxis] for o in octave], axis=2)
```

This function will generate the DoG octave by calculating the differences between gaussian octaves and return the concatenation of the DoG values

```
def generate_DoG_pyramid(gaussian_pyramid):
 pyr = []
 for gaussian_octave in gaussian_pyramid:
   pyr.append(generate_DoG_octave(gaussian_octave))
 return pyr
```

For generating the DoG pyramid, we only need to iterate the gaussian pyramid and append the DoG octaves.

- **Keypoint detection**

Given the DoG pyramid, now he calculates the candidate keypoints.

```
def get_candidate_keypoints(D, w=16):
 candidates = []
 D[:,:,0] = 0
 D[:,:,-1] = 0
 for i in range(w//2+1, D.shape[0]-w//2-1):
   for j in range(w//2+1, D.shape[1]-w//2-1):
     for k in range(1, D.shape[2]-1):
       patch = D[i-1:i+2, j-1:j+2, k-1:k+2]
       if np.argmax(patch) == 13 or np.argmin(patch) == 13:
         candidates.append([i, j, k])
 return candidates
```

The author claims that during the implementation he only get the extremas in the top and bottom level 0. That is why he set `D[:,:,0] = 0` and `D[:,:,-1] = 0.` Finally, 13 in calling argmax and argmin means the position 14 in the patch of 27 points we are considering.

Keypoint localization

- ***Subpixel localization***

To do this step, first we need to compute the Jacobian and Hessian matrix around each candidate keypoint. The calculation is straight forward if we know the derivatives formulas

```
def localize_keypoint(D, x, y, s):
 dx = (D[y,x+1,s]-D[y,x-1,s])/2.
 dy = (D[y+1,x,s]-D[y-1,x,s])/2.
 ds = (D[y,x,s+1]-D[y,x,s-1])/2.
 dxx = D[y,x+1,s]-2*D[y,x,s]+D[y,x-1,s]
 dxy = ((D[y+1,x+1,s]-D[y+1,x-1,s]) -- (D[y-1,x+1,s]-D[y-1,x-
1,s]))/4.
 dxs = ((D[y,x+1,s+1]-D[y,x-1,s+1]) -- (D[y,x+1,s-1]-D[y,x-
1,s-1]))/4.
 dyy = D[y+1,x,s]-2*D[y,x,s]+D[y-1,x,s]
 dys = ((D[y+1,x,s+1]-D[y-1,x,s+1]) -- (D[y+1,x,s-1]-D[y-
1,x,s-1]))/4.
 dss = D[y,x,s+1]-2*D[y,x,s]+D[y,x,s-1]
 J = np.array([dx, dy, ds])
 HD = np.array([ [dxx, dxy, dxs], [dxy, dyy, dys], [dxs, dys,
dss]])
 offset = -LA.inv(HD).dot(J)
 return offset, J, HD[:2,:2], x, y, s
```

- ***Discarding low contrast keypoint and elemating edge response***

We can do this step by applying the thresholds for both contrast value and the ratio of trace of determinant of Hessian detector

```
def find_keypoints_for_DoG_octave(D, R_th, t_c, w):
 candidates = get_candidate_keypoints(D, w)
 keypoints = []
 for i, cand in enumerate(candidates):
   y, x, s = cand[0], cand[1], cand[2]
   offset, J, H, x, y, s = localize_keypoint(D, x, y, s)
   contrast = D[y,x,s] + .5*J.dot(offset)
   if abs(contrast) < t_c: continue
   w, v = LA.eig(H)
```

```
    r = w[1]/w[0]
    R = (r+1)**2 / r
    if R > R_th: continue
    kp = np.array([x, y, s]) + offset
    keypoints.append(kp)
 return np.array(keypoints)
```

Again, if we already knew the intuition, we can easily code for this step. T_c and R_th is set to 0.03 and 10 respectively according to the original paper.

Finally, for each element in the pyramid we will calculate the keypoint for each of them trevally.

Orientation assignment

To assign the orientation, we first need to create a histogram with 36 bins around the keypoint. In order to do that we iterate all the keypoints, note that s = np.clip(s, 0, octave.shape[2]-1) is to bound the value. Then for each key point, we examine their neighbors by the window w. For each point in the neighbor, we calculate its gradient magnitude, gradient direction and append the value into the histogram. After that, for any histogram bin has a value within 80% of the maximum value, we also consider it as a keypoint. The author also does the further step to fir a parabola to the three histogram values closest to the maximum. All of these steps are done in the assign_orientation and fir_parabola functions

Local descriptor creation

In this step, we will create local descriptor by using the histogram of gradients around each keypoint. Particularly, we compute the orientation and magnitude in a 16x16 neighbors at key point, subsequently each patch will be split into 16 4x4 subregions.

The code for getting local descriptor consists of three for loops and five helper functions namely **get_patch_grads** (to get the gradient of the patch), **quantize_orientation** (to covert direction from radian to polar degree), **cart_to_polar_grad** (compute the gradient direction for the patch of keypoints), and **get_histogram_for_subregions** (to get the histogram for each subregion). The first loop is to initialize and calculate the gradient magnitude and direction for the whole patch of each subregion. Note that the dx and dy shape checking to avoid out of bounds errors.

The next two for loops are for iterating the pixels in each subregion and calculate the histogram for it. Then the histogram will be flattened out, normalized, thresholded to be less than or equal 0.2 then renormalized.

Calculating the gradient for a patch of keypoints is straight forward with python slicing because we can take the difference between the previous and current pixels.

Discussion

As we can see in the kypoint.py file, the code for implementing the local descriptor contains lots of for loop and still note utilize the use of numpy library. I have tested this code to calculate the keypoint for images with size 700x500 and it took about 15s per image. Therefore, we need to optimize the code, specifically in the keypoints.py file. Another thing the author has pointed out is the DoG construction. What we can do is rewrite the keypoints calculation, filtering, and the engine which searching for the extremums.

This PySIFT library is amazing interns of illustrating SIFT to the beginner like me. However, getting to understand the actual code may take weeks. Moreover, it is slow to run. Therefore, I believe that OpenCV should be applied for this problem. I also found some good resources explaining SIFT thoroughly.

References

https://www.youtube.com/watch?v=wpyGaQtRVz4

https://www.youtube.com/watch?v=NPcMS49V5hg

https://github.com/SamL98/PySIFT/pull/6

http://weitz.de/sift/index.html?size=large

https://www.analyticsvidhya.com/blog/2019/10/detailed-guide-powerful-sift-technique-image-matching-python/

https://www.vlfeat.org/overview/sift.html

https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_matcher/py_matcher.html

https://www.programcreek.com/python/example/89342/cv2.drawMatchesKnn

https://medium.com/analytics-vidhya/opencv-feature-matching-sift-algorithm-scale-invariant-feature-transform-16672eafb253