Implementing the SIFT algorithm

Huy Nguyen (40090345)
GitHub: https://github.com/giahuy22012000/ImplementingSIFT

April 19, 2022

Abstract

In this project, we attempt to implement our version of the SIFT algorithm invented by David Lowe.

1 Background

Feature Description along with other techniques such as corner and shape detection formed the main foundation for semantic image processing until the dawn of deep learning and other neural network based approaches.

Scale-Invariant Feature Transform or SIFT was invented by David Lowe in 1999, as a computer vision algorithm to detect and describe features in images [Wik]. SIFT is well known for its ability to detect features across different scale levels and its robustness against change in contrast, orientation and lumination. Therefore, SIFT algorithm has a wide range of applications such as image stitching, object tracking, and terrain mapping. In this project, we attempt to implement our version of the algorithm as described in Lowe's paper [Low04].

2 Implementation

The implementation of the said algorithm is done in successive states where the the result of one function call is passed to the next one.

2.1 Scale-space Extrema Detection

Features are defined as patches of an image that contain a relatively unique arrangement and fluctuation in pixel values to make it easily identifiable against the rest of the image. There are serveral approach to locate these keypoints. One of such methods is Harris Corner algorithm which is based on our percention and tendency to inherently look for corners when presented with an image. The method works well at locating corners through computing and evaluating the "corness" value for each keypoint from image gradients. However, the draw back of Harris Corner approach is that its result usually is sensitive to image scale 1.

SIFT algorithm addresses the scale variant issue by searching for keypoints not only on the original 1x scale of the image but also its downsampled versions. The algorithm takes advantage of the nature of Laplacian of Gaussian (LoG) filter to bring out non uniform regions of an image thus significantly reduces that search space for keypoints. Although this seems to be archieving the same result as corner searching using Harris Corner algorithm, SIFT keypoint search is often practically faster since it uses Difference of Gaussian (DoG) to approximate Laplacian of Gaussian which is more computationally expensive. In fact, for each scale of an image, SIFT algorithm applies Gaussian filters with sigma values varied by a factor ($K = \sqrt{2}$ is used in the project) to get images of different blur levels. This process is then repeated for the half downsampled version of the previous. Each of the sets of blured images on a same scale is called an octave [Low04]. Every two adjacent blured image in the same octave is substracted to get the Difference of Gaussian **??**. In the notebook, this process is done in $detectExtrema()$ function.

As per the paper, keypoints are searched among three consecutive images with different sigma values (scales) **??**. The process is demonstrated in $locateKeypoints()$ function in the notebook which we go through images with different blur levels from the second to second last position in each octave and compare the center pixel to their 28 neighbors. The indices are interpolated back to the original image size and save both in binary map and a list of OpenCV built-in $Keypoint$ objects. The result from calling the function are then drawn on the original image using OpenCV $drawKeypoints()$ function **??**.

2.2 Keypoint Localization

In this step, the keypoints are refined and thredhold so as to keep only a quality set using various techniques. However, since in this example, we

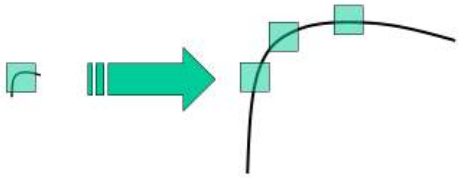

Figure 1: Harris corner scale variant

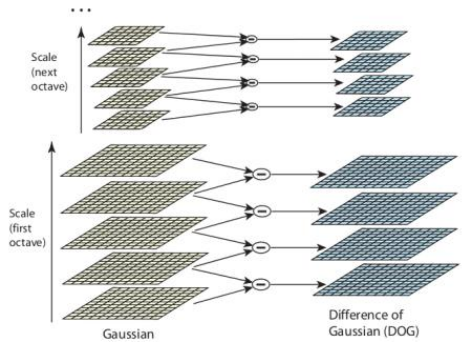

Figure 2: SIFT's Difference of Gaussian

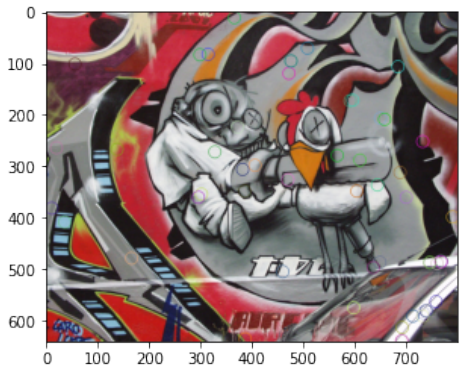

Figure 3: Detected keypoints

have not yet to optimize the scale-space extrema extraction, only a small set of keypoints are obtained from previous step. As shown in the notebook, we skip the keypoint localization step.

2.3 Orientation Assignment

The basic idea behind this steps is for each keypoint, we assign it with a orientation ($0\,\text{deg}\,to360\,\text{deg}$) obtained by looking at the keypoint's 4x4 neighborhood orientation histogram. The process is implemented in the notebook under $assignOrientation()$ function. By looping through the set of keypoints returned by previous steps, we extract the keypoint's neighborhood 4x4 window with index $[1, 1]$ being the keypoint's location. A list of size 36 is created to accumulate the keypoint's gradient magnitude adjusted by a Gaussian convolution. By calculating the two gradient components at each pixel in the window, we can easily compute its gradient orientation and magnitude using the following fomulas.

$$\Theta = \arctan \frac{\partial y}{\partial x} \|\nabla\| \qquad = \sqrt{(\partial x)^2 + \partial y^2}$$

The orientations Θ are then categorized into their respective bins (in $10\,\text{deg}$ increments) in the histogram. Using Numpy built-in $argmax()$ function, we can find the dominant orientation in the keypoint's neighborhood histogram and assign it to the keypoint. One of the histogram as demostrated in the notebook may look like this:

2.4 Keypoint Descriptor

After being assigned an orientation, the keypoints go into a process to create a identifier vector called keypoint descriptions. In the notebook, it is implemented in the $describeKeypoints()$ function. Similar to the function in previous step, the $describeKeypoints()$ function also examines each keypoint's neighborhood one by one but this time uses a larger 16x16 window with index $[7, 7]$ being the keypoint's location. However, instead of traversing the whole window and binning each pixel's orientation, the window is divided into 16 4x4 subwindows **??**. In each subwindow, orientation and magnitude values are calculated and a histogram of size 8 is used. Sixteen istograms are then concatenated into a long vector of total size 128 which is returned as the keypoint's desciptor.

One of the 16 historgrams may look like this:

2.5 Keypoint Matching

Since we do not know exactly the format of the returned descriptor vectors returned by OpenCV built-in SIFT $detectAndCompute()$ method, it is safer to also implement our version keypoint matching instead of using OpenCV's $BFMatcher$. The process is implemented in the $matchDescriptors$ function in the notebook. In selection sort fashion, the method searches for matches between each of the descriptors in first and second set using the sum of square difference. The best and second best differences are kept tracked and evaluated using ratio rule suggested in Lowe's paper [Low04].

3 Result

Our SIFT process is finalized in the $sift()$ function and tested on two images provided in the *data* folder. The result is undesirable **??**. This means that despite following the document, we may still have some errors an unoptimized part in the process, thus need many fixes.

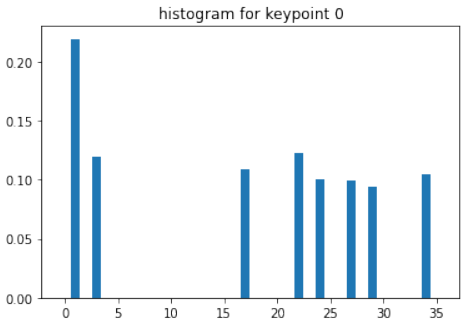

Figure 4: A typical 36-bin orientation histogram

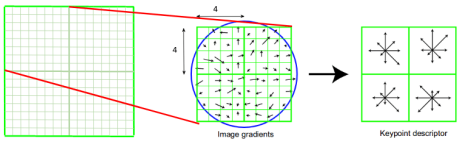

Figure 5: SIFT's keypoint descriptor

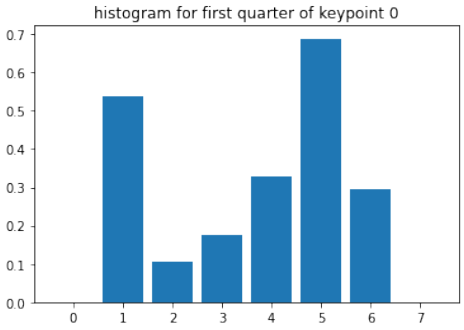

Figure 6: One of the 16 8-bin orientation histograms in a keypoint's descriptor

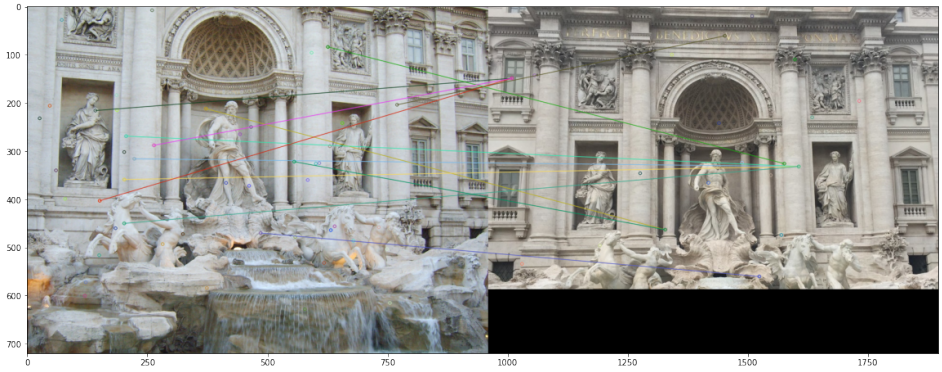

Figure 7: The output of our version of SIFT

References

[Low04] David G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, November 2004.

[Ope] OpenCV. Introduction to sift (scale-invariant feature transform). Available at https://docs.opencv.org/4.x/da/df5/tutorial_py_sift_intro.html.

[Wah] Ahmed Waheed. Sift feature extraction using opencv in python. Available at https://www.thepythoncode.com/article/sift-feature-extraction-using-opencv-in-python.

[Wik] Wikipedia. Scale-invariant feature transform. Available at https://en.wikipedia.org/wiki/Scale-invariant_feature_transform.