Compter Vision

Subhasish Basak (MDS201803)

Assignment 3

March 15, 2020

In this assignment we implement the two distinct steps of feature detection and description that are part of **SIFT**:

- **Interest point detection** in **harris.py**
- **Local feature description** in **sift.py**

1 Interest point detection

The implementation is done in the file named **harris.py**. It contains a single function **get_interest_points()**. The function takes the following arguments:

- image : A numpy array of shape (m,n,c), image may be grayscale of color.
  - We have used the package **cv2** for reading the image and convering into a numpy array.
  - Also the image is resized by cutting its length & width into half in order to reduce computational complexity.
- feature_width : integer representing the local feature width in pixels.
  - The importance of this argument is established in Local feature description. For this part we keep distance of $feature\_width/2$ from each boundary such that no key point is inside the boundary.

The function returns the following output:

- Co-ordinates of the keypoints (x,y)

1.1 Algorithm for keypoint detection

The algorithm for Harris corner detection for determining the co-ordinates of the interest points include the following steps:

- **Computing the horizontal and vertical gradients of the input image**
  - We have convoluted the image with the ***Sobel kernel***(both horizontal and vertical) for this purpose (respectively for the horizontal and vertical gradients). We name them **I_x** and **I_y**. For this purpose we have used the inbuilt packages of **scipy.signal**.
- **Computing the Covariance matrix**
  - To compute the covariance matrix we need to first compute the following :

$$I_{xx} = G_\sigma(I_x \times I_x) \tag{1}$$

$$I_{yy} = G_\sigma(I_y \times I_y) \tag{2}$$

$$I_{xy} = G_\sigma(I_x \times I_y) \tag{3}$$

Where G_σ is a $m \times n$-dimensional Gaussian filter applied in the product of the gradient matrices. We have used $\sigma = 1$ for the Gaussian kernel. For this purpose we have used the inbuilt packages of **scipy.ndimage**.

- Now the covariance matrix is constructed as : $\begin{bmatrix} I_{xx} & I_{xy} \\ I_{xy} & I_{yy} \end{bmatrix}$

- **Computing the Harris response value**
  - Now the detection of the corner (keypoints) in the image is done by comparing the **Harris response** value of each pixel to a threshold value, which is computed using the eigenvalues of the covariance matrix.
  - Let λ_1 and λ_2 be the 2 eigenvalues of the covariance matrix. Then the Harris response value for corner detection is computed as (according to Harris & Stephens (1988)):

$$R = \lambda_1 \lambda_2 - \kappa(\lambda_1 + \lambda_2)^2 \tag{4}$$

    here we have taken the parameter value $\kappa = 0.06$
  - We classify a pixel as an **interest point** if $R > 0$ for that pixel value. Our implementation also uses some additional filtering of these obtained key-points, which we discuss afterwards.

1.2 Filtering the key-points

After obtaining the key-points using the Harris algorithm discussed above we use some further techniques to filter the key-points. In our implementation we have used the additional functions which are implemented inside the **get_interest_points()** function :

- **neighbour_check()**
  - This function takes input the matrix named *harris_response*, which contains the response values corresponding to each pixel.
  - The function compares a feature point with its neighbours w.r.t the response values. Given a window size it considers all neighbours in that window of a feature point. If a feature point has greater response strength than all its neighbours it returns that feature point as potential feature point otherwise discards it.
  - The function also does an additional filtering by discarding all key-points whose response values are less than 0.0009 times the maximum response value among all the pixels.
  - Returns list of tuples named *harris_tuple* containing the co-ordinates and the response values of the filtered key-points (x,y).
- **check_feature_width()**
  - This function takes input the list of tuple named *harris_tuple*, the *feature_width*, the *height* and *width* of the image.
  - This function checks whether the selected feature points satisfies the boundary condition. This restricts feature points to fall within feature width from the boundary.
- **ANMS()**
  - This is a function for implementing **Adapted Non Maximal Suppression** for the key-points. This function takes input the sorted *harris_tuple*, which is obtained after applying the function ***neighbour_check()*** and ***check_feature_width()*** on the raw key-points.
  - For each key-point (x_i) this function computes the euclidean distance with the key-points (x_j)'s which have response value greater than that. To impose *robustness* it only considers those key-points (x_j)'s which have their response values some constant times greater than the main key-point. The minimum distance among all is the **Suppression Radius** given by,

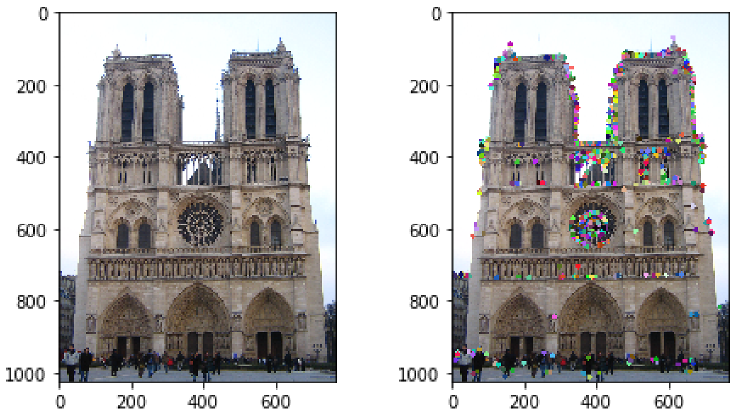
$$r_i = \min_j (x_i - x_j), s.t. f(x_i) < c_{robust} f(x_j) \tag{5}$$

    Here we have taken the robustness parameter as $c_{robust} = 0.9$.
  - We sort the key-points w.r.t their suppression radii in descending order and get a order list of the key-points.

1.3 Results

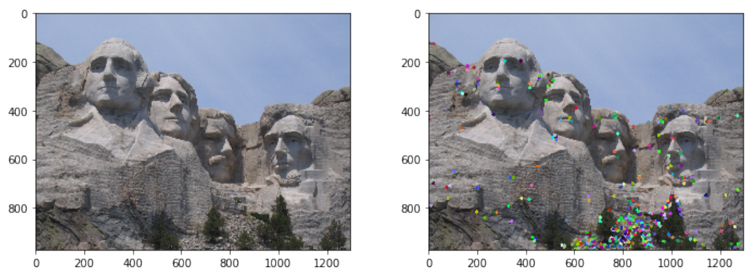
We implement the above algorithm along with the filtering functions on 3 images and here are the results. Throughout our experiments we have used the following parameter values:

- σ for Gaussian Kernel = 1
- κ for Harris response = 0.06
- Robustness parameter of ANMS = 0.9
- Neighbourhood threshold = 0.009
- Neighbour checking window size = 3×3



(a) Original (b) with corners

- For the **Notre Dame** image total number of feature points after Neighbourhood check was 780. After applying the Feature width constraint we reduce another 20 key-points. And we compute the suppression radii for them and plot the top 500 key-points.

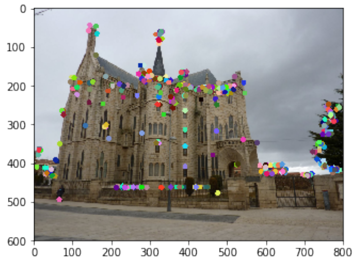


(a) Original (b) with corners

- For the **Mount Rushmore** image total number of feature points after Neighbourhood check was 567. After applying the Feature width constraint we reduce another 111 key-points. And we compute the suppression radii for them and plot the top 400 key-points.

- For the **Episcopal Gaudi** image total number of feature points after Neighbourhood check was 351. After applying the Feature width constraint we reduce another 30 key-points. And we compute the suppression radii for them and plot the top 300 key-points.

(a) Original



(b) with corners

2 Local feature description

Once we obtain the interest points of the image using Harris corner detection algorithm we are now interested in finding the feature descriptor corresponding each key-point. Here we *partially* implement the **SIFT** (Scale Invariant Feature Transform) algorithm. Inside the file **sift.py** we implement a function named **get_features()**. The function takes the following arguments:

- image : Same as previous function
- Co-ordinates of the key-points (x,y)
- feature_width : Same as previous function
- scales : Although our implementation is only for single scale, this parameter is used to determine the scale parameter of the Gaussian kernel.

The function returns the following output:

- A numpy array of shape (k, 128) representing a feature vector. Where **k** is the number of feature points.

2.1 Algorithm for computing the SIFT feature vectors

We start with the (x, y) co-ordinates of the interest points as obtained from applying the function **get_interest_points()** on the image. Next we compute the following :

- **Computing Orientation & Magnitude**
  - Similarly as the previous function we construct the gradient matrices **I_x** and **I_y**. Then the Orientation (θ) and Magnitude (m) is computed corresponding to each pixel as,

$$\theta = tan^{-1}\frac{I\_y}{I\_x} \tag{6}$$

$$m = \sqrt{I\_y^2 + I\_x^2} \tag{7}$$

  - The angles thus obtained not necessarily falls in the whole domain of $[0, 2\pi]$. Hence we scale the angles into the range.
- **Computing the Dominant orientation**
  - Next we compute the dominant orientation corresponding each key-point using *Histogram of Gradients* **(HOG)**.
  - Corresponding each key-point we take a window of size ($feature\_width \times feature\_width$) around it and build a histogram using all the pixels inside the window. The number of bins for the histogram is 36.
  - Suppose for a pixel its orientation (θ) falls in the bin $[a, b]$. We use the following **interpolation scheme** to identify the weights according to which the magnitude (m) of that pixel is distributed among the bins.

- * if $\theta > (a+b)/2$: The corresponding magnitude will contribute into the bin $[a,b]$ and the next bin $[b,c]$ to it with weights w_1 and w_2, given by:

$$w\_1 = 1 - \frac{\theta - (a+b)/2}{bin\_width} \tag{8}$$

- * if $\theta < (a+b)/2$: The corresponding magnitude will contribute into the bin $[a,b]$ and the previous bin $[d,a]$ to it with weights w_1 and w_2, given by:

$$w\_2 = 1 - \frac{(d+a)/2 - \theta}{bin\_width} \tag{9}$$

- * if $\theta = (a+b)/2$: We simply take $w\_1 = 1$ and $w\_1 = 0$.
  We note that $w\_1 + w\_2 = 1$. Finally we append $w\_1 * m$ into the bin $[a,b]$ and $w\_2 * m$ to the other bin.

- Once the histogram is obtained we find out the orientation bin corresponding to the highest peak and return the mid value of that bin as the dominant orientation.
- If there are multiple peaks we use **parabola fitting** using the three highest peaks and then return the orientation corresponding the vertex of the fitted parabola. The notion of *multiple peak* refers that the value of the $2nd$ highest peak should be greater than 80% of the highest peak.

- **Rotation invariant** To make our algorithm rotation invariant we assign the dominant orientation to the corresponding key-point. For all other pixels in the window we **rotate** their orientations by the dominant angle.

$$\theta_{new} = \theta_{old} - dominant\_orientation \tag{10}$$

  Also we have to scale θ_{new} for the range $[0, 2\pi]$

- **Local feature generation**
  - Again for each pixel we take a window around it of shape $(feature\_width \times feature\_width)$. We divide this window into 4×4 grids (total 16). Now for each of these 16 grids we build a histogram with 8 bins. In other words we get a 8×1 vector of weighted magnitudes corresponding each of the 16 grids.
  - Next we concatenate all the 16 histograms to generate a (128×1) feature vector, which is our **SIFT** vector. For each of the key-points we get one such (128×1) vector. The vectors are then normalized and returned.

2.2 Functions & Implementation

To implement the above discussed algorithm we have build two functions inside **get_features()** as follows:

- **fit_parabola()** : Given a input histogram it fits a parabola of the form $ax^2 + bx + c = y$ using the 3 bins with maximum magnitudes.
  - Fitting the parabola is equivalent to solving a system of linear equations $Ax = b$. Where the co-efficient matrix & co-efficient vector are given by,

$$A = \begin{bmatrix} \mathrm{x}_1^2 & \mathrm{x}_1 & 1 \\ \mathrm{x}_2^2 & \mathrm{x}_2 & 1 \\ \mathrm{x}_3^2 & \mathrm{x}_3 & 1 \end{bmatrix}, b = \begin{bmatrix} \mathrm{y}_1 \\ \mathrm{y}_2 \\ \mathrm{y}_3 \end{bmatrix} \tag{11}$$

  where x_i's are the mid values of the bins and y_i's are corresponding y-values.

- **build histogram()**
  - This function is called to build the histograms. It takes input the *orientation_matrix*, *magnitude_matrix* and the *number of bins.*
  - Returns a numpy array which contains the magnitude values.

The **SIFT** vectors are printed in my **Assignment3.ipynb** notebook