Berliner Hochschule für Technik

Prof. Dr.-Ing. Kristian Hildebrand

Learning from Images
Image descriptors and applications (12 Points)

WiSe 2025/26

The aim of this exercise is to develop a deeper understanding of distinctive image features / descriptors, their implementation and application areas. Download the skeleton source code from Moodle and implement the necessary functions. Expand and modify the source code as you like. The source code contains many more comments.

Exercise 1: Simple Content-based Image Retrieval (4 Points)

Figure 1: Retrieval results for test images from top left to bottom right.

The goal of a content-based image search is to find similar images related to an input image based only on image features (i.e. no keywords or tags). The search is also often based on the use of local feature vectors, e.g. SIFT. In the search, these vectors must be compared to find similarities. Different distance metrics can be used for this. In this very reduced example you should implement a search that compares the vectors based on the L2-norm (i.e. the Euclidean distance of the vectors). Proceed as follows and use *image_retrieval.py*:

a) Load all images of the supplied 'database'. There are a total of 3 different picture classes with about 5-6 pictures.

b) Create a set of keypoints that are distributed across the image in a uniform grid. Calculate the SIFT feature vectors at these locations.

c) Use the keypoints to calculate the SIFT feature vectors at these locations. This creates exactly one descriptor for each image in the database. These should be cached in an array. Note: We do not use the SIFT keypoint detector here.

d) Load a query image and create an image descriptor for the same keypoints. Compare the descriptor against the descriptors from the database. Use the L2 standard for this.

e) Save the result in a *PriorityQueue*.

f) Output the results of the query in the order of the shortest distance. The result should look like the faces shown in Figure 1.

Exercise 2: Write your own image descriptor (4 Points)

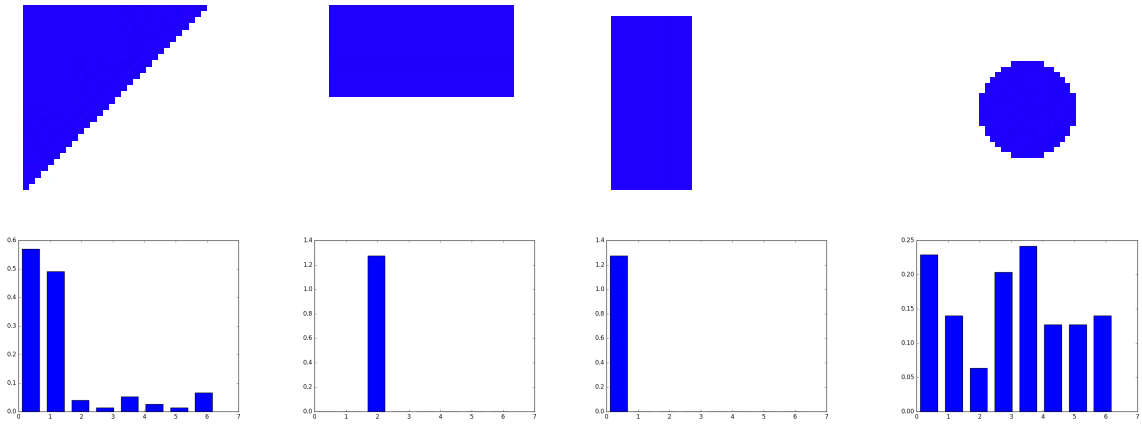

Figure 2: Histogram of oriented gradients test evaluation. Top row shows example images. Bottom row shows the histogram of gradients.

Implement your own local feature vector and test it with the supplied input images (simple_hog.py). The descriptor stores a histogram of gradient directions generated by a simple edge detection filter (e.g., sobel filter). In the specific case, the descriptor should only be extracted for a single keypoint (in the center of the image) and a given center size (e.g., 11). **Hint:** Take a look at the functions *cv2.Sobel*, *cv2.phase* and *cv2.magnitude*. The resulting histograms should look like in Figure 2 illustrated. The histogram contains 8 buckets. Think about a meaningful value range.

Exercise 3: Harris Corner Detector (4 Points)

As discussed in the lecture, the computation of the Harris corner detector requires the matrix

$$M = \sum_{x,y}^{3,3} w(x,y) \begin{bmatrix} I_{xx} & I_x I_y \\ I_y I_x & I_{yy} \end{bmatrix}$$

The matrix contains in I_{xx}, I_{yy}, I_{xy} the squared image gradients in x and y, summed over a 3×3 neighborhood. The procedure is as follows:

a) Compute the Sobel gradients in the x- and y-direction (G_x, G_y).

The eigenvalues of this matrix differ for edges, flat regions, and corners in their value ranges. For flat regions in the image, both eigenvalues are very small. For edges, one eigenvalue is at least an order of magnitude larger than the other, and for corners, both are of similar magnitude.

To compute Harris corners efficiently, the eigenvalues do not need to be explicitly calculated. It is sufficient to compute the determinant $(\det)$ and the trace (Tr) of the matrix:

$$H = \det(M) - k \cdot Tr(M)^2$$

As discussed in the lecture, the following steps are necessary and should be implemented in this task:

a) Compute the image gradients in x- and y-direction.

b) Compute I_{xx}, I_{yy}, I_{xy} as element-wise products of G_x, G_y.

c) Sum I_{xx}, I_{yy}, I_{xy} over a 3×3 neighborhood and store the results in new ND-arrays $sumI_{xx}, sumI_{yy}, sumI_{xy}$. This step can be implemented using a 2D convolution filter. Please use cv2.filter2D for this.

d) From these, construct the matrix M.

e) Compute the determinant of M using $\det(M) = I_{xx} \cdot I_{yy} - I_{xy} \cdot I_{xy}$.

f) Compute the trace of M using $Tr(M) = I_{xx} + I_{yy}$.

g) k is a parameter and should be between 0.04 and 0.06.

h) The final image H then contains the Harris corner features, which can be computed as shown above. In a final step, these are filtered using a threshold.

The file harris.py contains additional information about the implementation. To test your algorithm, you can use the OpenCV implementation as indicated in the source code and compare the results. The sum of the pixel differences between your implementation and the OpenCV implementation should be 0.0 (there may be minor floating-point inaccuracies). Larger deviations will result in point deductions, as will inefficient implementations (e.g., unnecessary loops). Your results should then look like those shown in Figure 3.

Figure 3: Example results of the Harris corner detector with preset parameter values as specified in the source code.

Exercise 4: Readings (voluntary)

Please read the publications in the reading area (Moodle).

Abgabe: This is *Exercise 2/4*. The tasks are designed to be solvable in 3 weeks. **Information on due date and a link to upload your solution as a .zip file is given in the Moodle system**. Please submit only one single .zip file with the sources of your solution. **Please insert all necessary images so that each task is directly executable. Otherwise I reserve point deductions.**. Submission after due date will be deducted with 5 points for each delayed week.