Computer Vision I
Assignment 3

Prof. Stefan Roth
Baris Can Cam
Oliver Hahn

28/11/2022

TECHNISCHE UNIVERSITÄT DARMSTADT

This assignment is due on Dezember 11th, 2022 at 23:59.

Please refer to the previous assignments for general instructions and follow the handin process described there.

Problem 1: PCA on Face Images (15 Points)

You will be working with a training database of human face images and build a low-dimensional model of the face appearance using Principal Component Analysis (PCA). We provide function definitions you have to implement in `problem1.py` and adhere to the notation used in class in the task description below.

Tasks:

- Implement function `loadfaces` that loads N images of human faces in a given path into a `numpy` array . Next, implement the function `vectorize_images` that turns these images into vectors.

  (1 + 1 points)

- Implement the PCA of the face images in `compute_pca` using the loaded data array. The function `compute_pca` returns the mean face, all principal component vectors u_i and the corresponding cumulative variance $\sum_{j \leq i} \lambda_i$.

  (4 points)

What do the principal components represent? To understand this better we can project individual face images on a few principal components and visualise the result. Concretely, we can represent an image as $x^n \approx \bar{x} + \sum_{i=1}^{D} a_i u_i$, where D is the number of components we select.

- Implement the function `basis` that selects the *fewest* possible principal components corresponding to the percentile fraction $\eta \in (0, 1]$ of the total variance. That is, D_c^*, the number of such components, should satisfy $D_c^* = \underset{D}{\operatorname{argmin}} \sum_{i=1}^{D} \lambda_i \geq \eta \sum_{i=1}^{M} \lambda_i$.

  (2 points)

- Implement the function `compute_coefficients` that returns the coefficients of a face image w.r.t. the bases we have computed in the previous step.

  (1 point)

- Next, implement the function `reconstruct_image` that returns an approximate reconstruction of the face image given the basis coefficients.

  (1 point)

You can now select a face image of your choice and visualise its projection on a few basis vectors. Experiment with different percentiles, e.g. $\eta = 0.5$, 0.75, 0.9, and analyse the result.

We can now explore some useful applications of the basis representation we have obtained.

- *Image Search.* We can use the projection coefficients a_i as image descriptors and compare images by computing the similarity between their compact vector representation in terms of a few principal components (e.g. corresponding to a sufficiently large percentile η). First, implement the function `compute_similarity` that calculates the cosine similarities between a target image and an array of face images based on the coefficients w.r.t. the PCA basis. Then implement the function `search` that searches for the top-n most similar images based on the cosine similarities. *Sanity check:* A function call with top-1 should always return the image itself.

  (2 + 1 points)

- *Face Interpolation.* Implement function `interpolate` that takes two face images and produces a given number of intermediate images. First, project each image on the provided basis vectors to obtain vectors with a_i's. Then, interpolate between the two representations in the PCA basis at equal steps and reconstruct the corresponding images. *Hint:* You may find `np.linspace` useful for this task.

  (2 points)

Submission: Please include only `problem1.py` in your submission.

Problem 2: Hessian Detector (10 Points)

We humans have the ability to recognize whether two different images show the same object, but how can we solve this using computer vision? In this problem we will take a look at interest point detection, aiming to elaborate the foundations with which we can translate this ability to a computer.

Figure 1: Which images show the same object?

One of the earliest interest point detectors was the Hessian detector which identifies corner-like structures by searching for points $\boldsymbol{p} = (x, y)^T$ with a strong Hessian determinant $\det(\boldsymbol{H})$. We use the Hessian matrix $\boldsymbol{H}$ that is calculated on the image smoothed by a Gaussian filter with kernel width σ, i.e.

$$\boldsymbol{H}(\sigma) \begin{bmatrix} I_{xx}(\sigma) & I_{xy}(\sigma) \\ I_{xy}(\sigma) & I_{yy}(\sigma) \end{bmatrix} \tag{1}$$

with $I_{xx}(\sigma)$, $I_{xy}(\sigma)$ and $I_{yy}(\sigma)$ denoting the partial (horizontal and vertical) second derivatives of the smoothed image I. The interest points are defined as those points whose Hessian determinant is larger than a certain threshold t, i.e.

$$\sigma^4 \cdot \det\big(\boldsymbol{H}\big) > t. \tag{2}$$

Note that we include an additional scale normalization factor σ^4 so that we can use the same threshold t independently of the value of σ. For the following tasks please use the functions `gauss2d` and `derivative_filters` given in `problem2.py` to generate a Gaussian smoothing filter with size 10×10 and $\sigma = 2$ and use the central difference filters to compute the derivatives. For color images you only need to detect the interest points in the gray-scale space (`load_img` returns color and gray-scale images).

The code outline is given in `problem2.py` which should be completed with the necessary functions:

- Function `compute_hessian` to obtain the required components of the Hessian $\boldsymbol{H}$ with the given Gaussian and derivative filters. Use *mirror* boundary conditions for any filtering involved.

  (3 points)

- Function `compute_criterion` that computes the scaled Hessian determinant given by the left-hand side of (2).

  (2 points)

- Function `nonmaxsuppression` that applies non-maximum suppression to the computed criterion in order to extract local maxima, *i.e.*, points for which function values are the largest within their surrounding 10×10 windows, respectively. Allow multiple equal maxima in one window and throw away all interest points in a 10 pixel boundary at the image edges. After that find all local maxima with a function value that is larger than the threshold $t = 3 \cdot 10^{-3}$. Note: To implement `nonmaxsuppression` the function `maximum_filter` from `scipy.ndimage` might be handy.

  (3 points)

- Function `imagepatch_descriptors` retrieves 11×11 image patches around every interest point. Using these image patch features, the function `match_interest_points` performs brute-force matching of the interest points of two images. A list of matches sorted from best to worst is returned. Note: To implement `match_interest_points` the function `BFMatcher` from `cv2` might be handy (`https://docs.opencv.org/3.4/d3/da1/classcv_1_1BFMatcher.html`). Also consider to choose a reasonable distance measurement.

  (2 points)

Submission: Please only include `problem2.py` in your submission.