Computer Vision I
Assignment 2

Prof. Stefan Roth
Jannik Schmitt
Jan-Martin Steitz

TECHNISCHE UNIVERSITÄT DARMSTADT

30/11/2020

This assignment is due on December 13th, 2020 at 23:59.

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

Problem 1: Image Pyramids and Image Sharpening (15 Points)

Image pyramids are widely used in computer vision. In this problem you will create a small sharpening application by using both Gaussian and Laplacian image pyramids. You will first write a few helper functions.

Tasks:

Implement the following functions:

- Function `loadimg` that takes the path to an image file and returns the image as an array of floating point values between 0 and 1.

  (1 point)

- Function `gauss2d` with two arguments that creates a Gaussian filter with specified size (in x and y direction) and specified kernel width (standard deviation σ). You can find the formula in the slides. Make sure that the maximum is in the middle of the filter mask (for even and odd filter sizes), and also ensure that the filter is properly normalized (i.e. the filter coefficients should add to 1).

  (1 point)

- Function `binomial2d` that creates a binomial filter with the specified filter size (in x and y direction). To construct a binomial filter you initially compute the set of binomial coefficients, e.g. as in Pascal's triangle, and normalize afterwards. For instance, the weights w_k of a 1D binomial filter with $N+1$ weights can be constructed by the formula

$$w_k = \widehat{w}_k / \sum_{l=0}^{N} \widehat{w}_l, \quad \widehat{w}_k = \binom{N}{k}, \qquad k = 0, 1, \ldots, N. \tag{1}$$

  (2 points)

- Function `downsample2` that takes an image and a gaussian filter and downsamples it by a factor of 2, i.e. resizes both dimensions to half the size. First, smooth the image with the gaussian filter. Then simply discard every other row and column. Note: You should use suitable boundary conditions for filtering to avoid visual artifacts.

  (1 point)

Figure 1: Gaussian pyramid

- Function `upsample2` that takes a low resolution image as well as a binomial filter and upsamples the image by a factor of 2, i.e. resizes each dimension to double the size. In particular, insert one zero row after every low-resolution row and insert one zero column after every "old" column. Filter the result with the binomial kernel and finally apply a scale factor of 4. Note: You should use suitable boundary conditions for filtering to avoid visual artifacts.

  (2 points)

Now, implement the following tasks by using the functions you wrote above:

- Function `gaussianpyramid` that builds a multi-level Gaussian pyramid of a gray-value input image (a 6-level Gaussian pyramid in this assignment, using the variable `nlevel`). To create a subsequent level of the Gaussian pyramid, use the function `downsample2` from above and set $\sigma = 1.4$ and size 5×5. Note: The result of this function should be a list of images with decreasing sizes.

  (2 points)

- Function `laplacianpyramid` that, based on the Gaussian pyramid, creates the corresponding Laplacian pyramid. For building the Laplacian pyramid, you should also use the upsampling function from above. What is the difference between the top (coarsest) level of Gaussian and Laplacian pyramids?

  (2 points)

- Function `createcompositeimage` that shows image pyramids in a single figure as depicted in Fig. 1 . Since we want to use the same function to display both Gaussian and Laplacian image pyramids, you should normalize the images of the pyramid levels individually such that they have values in $[0, 1]$.

  (1 point)

- In our skeleton we load and sharpen the image `a2p1.png`. To make the sharpening work you have to implement the function `amplifyhighfreq` that amplifies the high-frequency components contained in the finest two levels of the Laplacian pyramid by scaling them up by a factor each. Afterwards, implement `reconstructimage` that reassembles the Laplacian pyramid back to obtain a sharpened full-resolution image. Try various amplification factors for both levels and set those as default values in the keyword arguments of `amplifyhighfreq` that lead to good or interesting results. You may find that the image noise gets amplified if you scale up the coefficients of the finest sub-band too much; try to avoid that. Finally, display the original image, its reconstruction and their difference side by side.

  Note: Python arrays (and other non-primitive types) are assigned by reference. For example, look at the following expression `A=np.zeros(1); B=A; B[0]=1; print(A[0]);`. Use `deepcopy` to avoid changing input arrays.

  (1+2 points)

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

Problem 2: 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 `problem2.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 `problem2.py` in your submission.