Exercise 02 for MA-INF 2201 Computer Vision WS22/23
23.10.2022
Submission on 3.11.2022

Please implement the solutions on your own, you may not use implementations existing on the internet. And unless instructed, you should not use built-in functions (example, for template matching), since the goal is to learn the underlying algorithms

1. **Fourier Transform** In this task, we will show a useful property of the Fourier Transform, which is the convolution property. It tells us that convolution in the spatial domain corresponds to multiplication in the frequency domain. The input image that you will be operating on is `orange.jpeg` and `celeb.jpeg`.

   - Load the two images `orange.jpeg` and `celeb.jpeg`. Remember to convert to grayscale.
   - Create a 7×7 Gaussian kernel with `sigma=1`.
   - Blur the images by convolving the image with the Gaussian kernel directly in the spatial domain. You may use the library function (`cv2.filter2D()`).
   - Now we are going to blur the images in the frequency domain using Fourier Transform. We multiply the kernel function and the frequency image instead of applying convolution. To get the final result, we transform back to the image space. You may use functions included in the package `numpy.fft` to apply the transform and its inverse.
   - Visualise the results for both images and report the mean absolute difference between the two blurring methods and the time taken by each of them.

   (3 Points)

2. **Template Matching** In this task, we will implement template matching using Sum Square Difference and Normalized Cross-Correlation similarity measures. The input image is `RidingBike.jpeg` and the template image (what we want to find in the larger input image) is `RidingBikeTemplate.jpeg`

   - Implement Sum Square Difference.
   - Implement template matching using your implementation of Sum Square Difference.
   - Implement Normalized Cross-Correlation.
   - Implement template matching using Normalized Cross-Correlation.
   - Draw rectangles on the image where $similarity \geq 0.5$ for both methods. You may experiment with other threshold values to determine the matching.

   (3 Points)

3. **Template matching multi-scale** In this task, we will build the Gaussian pyramid to make template matching work at different resolutions. Read the image `DogGray.jpeg` and the template `DogTemplate.jpeg`

   - Build a 5 level Gaussian pyramid by downsampling the input image and applying the Guassian filter yourself (you may use and cv2.filter2D() and cv2.resize()).

- Now create a 5 level Gaussian pyramid using `cv2.pyrDown()`. Compare it with your implementation by printing the mean absolute difference at each level.
- Perform template matching by using your implementation of normalized cross correlation . Report the time taken by this method.
- Show the template matching using normalized cross correlation at the different Pyramid levels of both the template and input images (you can use the pyramid obtained by pyrDown()).
- As you observed, implementing template matching naively is not efficient. Now we will rely on the pyramid technique while constraining the search space in order to make it faster. Follow the procedure described in the lecture slides and search only in regions with high similarity in the previous pyramid level. Compare the times taken by this method and the naive implementation.
- Visualise the template matching results.

(6 Points)

4. **Pyramids for image blending** In this task, we will stitch two images using pyramids. Without pyramids, blending does not look natural because of discontinuities between the pixel values. We will blend the images `dog.jpeg` and `moon.jpeg`.

   - Load the two images `dog.jpeg` and `moon.jpeg`.
   - Create the Gaussian Pyramids of the two images, and find their Laplacian Pyramids LA and LB (remember that a Laplacian Pyramid is the difference between two levels in the Gaussian Pyramid as explained in the lecture, i.e. $L_i = G_i - expand(G_{i+1})$). Set the number of levels to 5. You may use cv2.buildPyramid().
   - Blend the image `dog.jpeg` with the image `moon.jpeg`: create a Gaussian pyramid GR for the region of interest in the given mask `mask.jpeg` (first transform the mask to grayscale).
   - Combine the Laplacian pyramids using GR as weights for the blending, i.e. $LS_l(i,j) = GR_l(i,j) \cdot LA_l(i,j) + (1 - GR_l(i,j)) \cdot LB_l(i,j)$
   - Collapse the LS pyramid to obtain the final composite image: $LS_l = LS_l + expand(LS_{l+1})$.
   - Apply the blending operation on the images inside `task4` (the results will be funny).

   (6 Points)

5. **Edges** In this task, we will detect edges in images using the derivative of a Gaussian kernel. Read the image `einstein.jpeg`.

   - Compute the weights of the derivative (in x) of a 5x5 Gaussian kernel with $\sigma = 0.6$.
   - Compute the weights of the derivative (in y) of a 5x5 Gaussian kernel with $\sigma = 0.6$.
   - To get the edges, convolve the image with the kernels computed in previous steps. You can use `cv2.filter2D()`.
   - Compute the edge magnitude and the edge direction (you can use `numpy.arctan2`). Visualise the magnitude and direction. *(2 Points)*