ECE4880J: Computer Vision June 1, 2022

Homework 3: Image Processing

Instructor: Siheng Chen **YOUR NAME**

Instruction

- This homework is due at 11:59:59 p.m. on ** June 13th, 2022.
- The write-up must be a soft copy `.pdf` file edited by LaTeX.
- The overall submission should be a `.zip` file named by xxx(student id)-xxx(name)-Assignment3.zip

Python Environment. We are using Python 3.7 for this course. We will use the following packages in this course: Numpy, OpenCV-python, Matplotlib, Pytorch.

Q1. Image Compression [20 points]

We are going to apply the DCT transformation for image compression. A discrete cosine transform (DCT) expresses a finite sequence of data points in terms of a sum of cosine functions oscillating at different frequencies. DCT helps separate the image into parts (or spectral sub-bands) of differing importance (with respect to the image's visual quality). DCT is similar to the discrete Fourier transform: it transforms a signal or image from the spatial domain to the frequency domain. To an image f, f(x,y) represent the pixel value at (x,y), the equation of the DCT tranformation is:

$$F(u,v) = \alpha(u)\alpha(v)\sum_{x=0}^{M-1}\sum_{y=0}^{N-1} f(x,y)\cos\frac{(2x+1)u\pi}{2M}\cos\frac{(2y+1)v\pi}{2N},$$

where $\mathrm{u} = 0, 1 \ldots .\mathrm{M}-1, \mathrm{v} = 0, 1 \ldots \mathrm{N}-1$. $\alpha(u) = \begin{cases} \frac{1}{\sqrt{M}} & u = 0 \\ \sqrt{\frac{2}{M}} & u \neq 0 \end{cases}$ $\alpha(v) = \begin{cases} \frac{1}{\sqrt{N}} & v = 0 \\ \sqrt{\frac{2}{N}} & v \neq 0 \end{cases}$.

For most images, much of the signal energy lies at low frequencies; these appear in the upper left corner of the DCT. To reverse the image, the equation of the inverse DCT (iDCT) tranformation is:

$$f(x,y) = \sum_{u=0}^{M-1}\sum_{v=0}^{N-1} \alpha(u)\alpha(v)F(u,v)\cos\frac{(2x+1)u\pi}{2M}\cos\frac{(2y+1)v\pi}{2N},$$

where $\mathrm{x} = 0, 1 \ldots .\mathrm{M}-1, \mathrm{y} = 0, 1 \ldots \mathrm{N}-1$. $\alpha(u) = \begin{cases} \frac{1}{\sqrt{M}} & u = 0 \\ \sqrt{\frac{2}{M}} & u \neq 0 \end{cases}$ $\alpha(v) = \begin{cases} \frac{1}{\sqrt{N}} & v = 0 \\ \sqrt{\frac{2}{N}} & v \neq 0 \end{cases}$.

The whole process to compress an image with the DCT transformation takes following steps:

1. use DCT transformation to obtain the representation in frequency domain;
2. compress the representation in frequency domain by preserving data in the top-left corner (1/4 area by 1/2 width and 1/2 height);
3. recover the image using inverse DCT transformation (iDCT). Complete the function and some blanks in `myHought`.

use the image lena.jpg as grayscale and plot your experiment results. Note that you can NOT use any functions relevant to DCT in extended packages like opencv.

- Visualize results including the representation in the frequency domain and the recovered image with/without compression in the frequency domain.
- Use another compression scale. Preserving data in the top-left corner of 1/16 area (1/4 width and 1/4 height). Comparing the recover image.
- Blockwise compression vs. Global compression. Divide the image into 8 by 8 pixel image patches. For each patch, perform the same operation (transformation,compression,recovering) and gather all the patches into the whole image. Comparing the results of blockwise compression and global compression (including the representation in the frequency domain).

Submission format. Please submit your filled `compression.py`. Write and analyze your experiment visualization results of above tasks in the write-up.

Q2. Image Filtering [20 points]

Convolution is one of the most important operations for image processing. It is an effective way to extract the desired local features from an image. Implementing a 2D convolution function will help you to get familiar with the mechanism of convolution.

Requirements. Complete the `myConv2D` function in `myConvolution.py`.

The input of `myConv2D` includes

- `img` - a grayscale input image.
- `kernel` - a 2d convolution kernel. You can assume that the kernel is always odd-sized.
- `stride` - the parameter to control the movement of the kernel over the input image. When the input height and width are not divisible by the stride, the remaining rows and columns will be ignored. The amount of stride can be defined by a two-element tuple, or a single integer, in which cases it controls the movement amount for both the x-direction and the y-direction.
- `padding` - the parameter to control the amount of padding to the input image. The padding should be added to all four sides of the input image. You can call `numpy.pad` to pad the array. The amount of stride can be defined by a two-element tuple, in which case the first element controls the top and bottom padding size and the second one controls the left and right padding size; or a single element, which controls the padding size of all the sides. The details of the API can be found HERE.

The output of `myConv2D` is the processed image. The size of the output image can be calculated by

$$\text{output}[0] = \left\lfloor \frac{\text{img}[0] + 2 \times \text{padding}[0] - \text{kernel}[0] + \text{stride}[0]}{\text{stride}[0]} \right\rfloor$$

$$\text{output}[1] = \left\lfloor \frac{\text{img}[1] + 2 \times \text{padding}[1] - \text{kernel}[1] + \text{stride}[1]}{\text{stride}[1]} \right\rfloor$$

Your code should not call on convolve, correlate, fftconvolve, or any other similar functions from any libraries.

Hint. Here are some tips for a fast and elegant implementation.

1. Relying on mathematical functions to operate on vectors and matrices can speed up your calculation. You can find some examples of vectorized operations HERE.
2. The number of `for` loops should be as small as possible. Two loops may be sufficient for the implementation.

Submission format. Please submit your filled `myConvolution.py`.

Q3. Edge Detection [20 points]

Edge detection is a classical task in the field of computer vision. The Canny's algorithm is one of the most popular solutions to this task because it can detect edges with a low error rate, locate the centerline of the edges accurately, and ignore most of the noises.

The process of canny can be broken down into the following steps

1. Filter the noise in the image by a Gaussian filter.
2. Find the intensity gradient of the image. A common choice is to apply the Sobel filter of the x-direction and the y-direction to do the convolution respectively.
3. Apply non-maximum suppression to filter those pixels that have high gradient intensity but are not on an edge. Generate the edge candidates.
4. Determine the potential edges by the hysteresis threshold. The threshold is two-sided. When the pixel value is smaller than the lower threshold, it must not be a part of an edge. When the pixel value is larger than the upper threshold, it must be a part of an edge. When the pixel value is between the thresholds, decide whether the pixel is on an edge by its neighbors. If any of its neighbors is a member of an edge, this pixel is a part of an edge.

Requirements. Complete the `myCanny` function in `myEdgeDetector.py`.

The input of `myCanny` includes

- `img` - a grayscale input image.
- `sigma` - the standard deviation of the Gaussian kernel. In this implementation, it will also define the size of the Gaussian kernel as $(2 * \lceil 3 * \text{sigma} \rceil + 1) \times (2 * \lceil 3 * \text{sigma} \rceil + 1)$.
- `threshold` - the hysteresis threshold.

The output of `myConv2D` are the input image and the detection result, whose size should be the same as the input image.

Your code should not call on the implementation of canny or similar functions from any libraries.

Submission format. Please submit your filled `myEdgeDetector.py`. Test your edge detector on image `hw3_zebra.jpg` and report your experiment result in the write-up.

Q4. Line Detection [20 points]

Part A Task Applies the Hough Transform to an edge detection result of an image. Display the output in the write-up using img01.jpg and img02.jpg. Complete the function in the `myHoughTransform.py`:

[img_houghtrans, rhoList, thetaList] = Handwrite_HoughTransform(img_threshold, rhostep, thetastep).

For the input of function:

- `img_threshold` - the edge detection result, thresholded to ignore pixels with a low edge filter response.
- `rhostep` (scalar) - the distance resolution of the Hough transform accumulator in pixels.
- `thetastep` (scalar) - the angular resolution of the accumulator in radians.

Here we give a suitable rhostep, thetastep value in `houghScript.py`.

For the output of function: img_houghtrans, rhoList, thetaList. img_houghtrans is the Hough transform accumulator that contains the number of "votes" for all the possible lines passing through the image. rhoList and thetaList are the 1-D arrays of ρ and θ values ranging from 0 to maxinum ρ/θ by the step of rhostep/thetastep. If rhoList[i] $= \rho_i$ and thetaList[j] $= \theta_j$, then img_houghtrans[i,j] contains the votes score for $\rho = \rho_i$ and $\theta = \theta_j$.

The main script to run this experiment is in houghscript.py, note that you don't need to fill any blanks here and you can directly use it as long as you implement the function tools. A sample visualization of img_houghtrans is shown in Figure 1.

Figure 1: A sample visualization of img_houghtrans.

Part B Task Use your Hough transform and write a function in `myHougnlines.py` to detect lines:

[rhos, thetas] = Handwrite_HoughLines(Im, num_lines),

where `Im` is your accumulator of Hough transform, and `num_lines` is the number of lines to detect. Outputs rhos and thetas are both 1-D vectors with length of `num_lines` that contain the row and column coordinates of top-num_lines peaks in the accumulator `Im`, that is, the lines found in the image.

Here is an important note but do not spend your time implementing. For every cell in the accumulator corresponding to a real line (likely to be a locally maximal value), there will probably be several cells in the neighborhood that also scored high but should not be selected. This would cause a series of similar lines to be detected. Thus these non-maximal neighbors can be removed using non-maximal suppression. Here we prepare a non-maximal suppression function tool for you to use, or you can implement it by yourself.

Once you have ρ and θ for each line in an image, the main script would draw red lines representing detected edges in your image based on the values of ρ and θ. The green line segments represent the output of OpenCV's HoughLinesP (see Figure 2 as an example).

Submission Format. Please submit your filled `myHoughtransform.py`, `myHougnlines.py`. Write and analyze your experiment results of above tasks in the write-up.

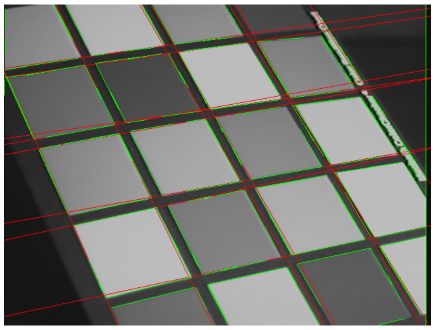

Figure 2: A sample visualization of line detection result.

Q5. Feature Extraction [20 points]

The Scale-Invariant Feature Transform (SIFT) is a computer vision algorithm to detect, describe, and match local features in images. Implement SIFT will help you to get familiar with image descriptor.

Requirements. Complete the `getDoGImages` and `computeGradientAtCenterPixel` function in `SIFT.py`. And use the SIFT function in action with a template matching demo given by matching_with_SIFT.py.

Let's briefly go over SIFT and develop a high-level roadmap of the algorithm. You can (and should) read the original paper HERE. SIFT identifies keypoints that are distinctive across an image's width, height, and most importantly, scale. By considering scale, we can identify keypoints that will remain stable (to an extent) even when the template of interest changes size, when the image quality becomes better or worse, or when the template undergoes changes in viewpoint or aspect ratio. Moreover, each keypoint has an associated orientation that makes SIFT features invariant to template rotations. Finally, SIFT will generate a descriptor for each keypoint, a 128-length vector that allows keypoints to be compared. These descriptors are nothing more than a histogram of gradients computed within the keypoint's neighborhood.

The entire function of `SIFT` includes

- `img` - a grayscale input image;
- `getBaseImage` - appropriately blur and double the input image to produce the base image of our "image pyramid", a set of successively blurred and downsampled images that form our scale space;
- `computeNumberOfOctaves` - compute the number of layers ("octaves") in our image pyramid. Now we can actually build the image pyramid;
- `getGaussianKernels` - create a list of scales (gaussian kernel sizes) that is passed to `getGaussianImages`;
- `getGaussianImages` - repeatedly blurs and downsamples the base image;
- `getDoGImages` - subtract adjacent pairs of gaussian images to form a pyramid of difference-of-Gaussian ("DoG") images, **you need to implement this function**;
- `findScaleSpaceExtrema` - identify keypoints based on the DoG image pyramid. We iterate through each layer, taking three successive images at a time. Remember that all images in a layer have the same size — only their amounts of blur differ. In each triplet of images, we look for pixels in the middle image that are greater than or less than all of their 26 neighbors: 8 neighbors in the middle image, 9 neighbors in the image below, and 9 neighbors

in the image above. The function `isPixelAnExtremum` performs this check. These are our maxima and minima (strictly speaking, they include saddle points because we include pixels that are equal in value to all their neighbors). When we've found an extremum, we localize its position at the subpixel level along all three dimensions (width, height, and scale) using `localizeExtremumViaQuadraticFit`, explained in more detail below. The code to localize a keypoint may look involved, but it's actually pretty straightforward. It implements verbatim the localization procedure described in the original SIFT paper. We fit a quadratic model to the input keypoint pixel and all 26 of its neighboring pixels. We update the keypoint's position with the subpixel-accurate extremum estimated from this model. We iterate at most 5 times until the next update moves the keypoint less than 0.5 in any of the three directions. This means the quadratic model has converged to one pixel location. The two helper functions `computeGradientAtCenterPixel` and `computeHessianAtCenterPixel` implement second-order central finite difference approximations of the gradients and hessians in all three dimensions. **you need to implement function** `computeGradientAtCenterPixel`;

- `removeDuplicateKeypoints` - clean up these keypoints by removing duplicates;
- `convertKeypointsToInputImageSize` - converting them to the input image size;
- `getDescriptors` - generate descriptors for each keypoint. Now we've found the keypoints and have accurately localized them. Next, we need to compute orientations for each of our keypoints. We'll likely also produce keypoints with identical positions but different orientations. Then we'll sort our finished keypoints and remove duplicates. Finally, we'll generate descriptor vectors for each of our keypoints, which allow keypoints to be identified and compared.

Now let's see SIFT in action with a template matching demo. Given a template image, the goal is to detect it in a scene image and compute the homography. We compute SIFT keypoints and descriptors on both the template image and the scene image, and perform approximate nearest neighbors search on the two sets of descriptors to find similar keypoints. Keypoint pairs closer than a threshold are considered good matches. Finally, we perform RANSAC on the keypoint matches to compute the best fit homography. A sample result is shown as follows

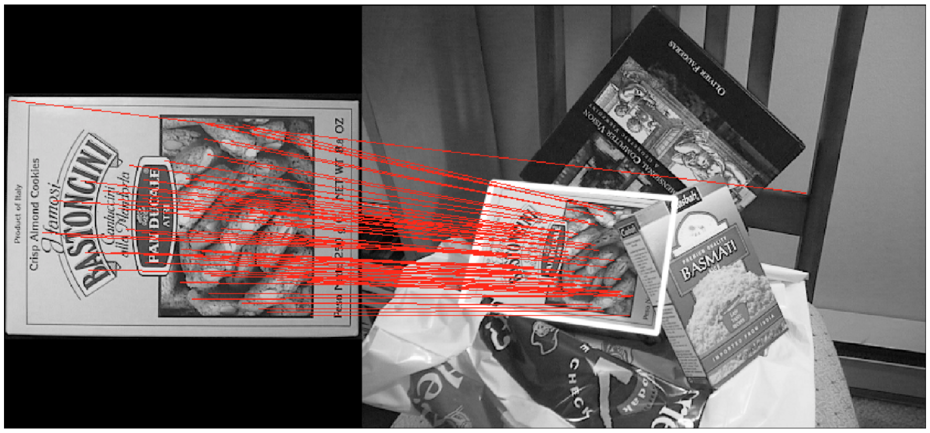

Figure 3: Output plot of `matching_with_SIFT.py`, showing the detected template along with keypoint matches.

Submission Format. Please submit your filled `SIFT.py`. Write and analyze your experiment results of above tasks in the write-up.

Extra Credits: Sparse coding with Pytorch [20 points]

Sparse coding is a class of unsupervised methods for learning sets of over-complete bases to represent data efficiently. The aim of sparse coding is to find a set of basis vectors $\mathbf{D}$ to represent an input vector $\mathbf{x}^{(t)}$ such that we can: i) find a sparse latent representation $\mathbf{h}^{(t)}$ to represent $\mathbf{x}^{(t)}$ as a linear combination of these basis vectors, which has many zeros; ii) good reconstruct the original input $\mathbf{x}^{(t)}$. To achieve this goal, the objective function is given by

$$\mathrm{min}_{\mathbf{D}} \frac{1}{\mathrm{T}} \sum_{\mathrm{t}=1}^{\mathrm{T}} \mathrm{min}_{\mathbf{h}^{(\mathrm{t})}} \frac{1}{2} \|\mathbf{x}^{(\mathbf{t})} - \mathbf{D}\mathbf{h}^{(\mathrm{t})}\|_2^2 + \lambda \|\mathbf{h}^{(\mathrm{t})}\|_1 \tag{1}$$

where T is the amount of sample, $\frac{1}{2}\|\mathbf{x}^{(\mathbf{t})} - \mathbf{D}\mathbf{h}^{(t)}\|_2^2$ is the reconstruction error, $\lambda\|\mathbf{h}^{(t)}\|_1$ is the sparsity penalty, and λ is the hyperparameter that controls the sparse penalty strength.

Requirements. Implement image denoising with sparse coding. Display the output of `sparsecoding.ipynb`. We recommend to learn the sparse coding based on the DCT dictionary you got in Q1. To solve this objective function, we provide a sample solver `MatchingPursuit` in `persuits.py`. You could choose one of the following tasks

1. substitute this solver with ADAM optimizer in Pytorch, analyze the optimizing process;
2. analyze the effect of different choice of DCT dictionary on the image denoising task.

Submission Format. Please submit your code, write and analyze your experiment results of above task in the write-up.