Introduction to Computer Vision (Spring 2024)
Assignment 1

Release date: March 15, due date: March 30, 2024 11:59 PM

1 Overview

The assignment includes 4 tasks: implementing convolution operation, Canny edge detector, Harris corner detector, plane fitting using RANSAC, which sums up to 100 points and will be counted as 10 points towards your final score of this course. This assignment is fully covered by the course material from Lecture 2 and 3.

The objective of this assignment is to get you familiar with coding and implementing the basic modules and algorithms of classic computer vision **in a tensor style**. Here the tensor style refers to avoiding using any for loops and, instead, *implement everything via matrix or tensor operations, e.g. matrix multiplication*. This exactly resembles how people code in PyTorch or Tensorflow for the sake of parallel computing. We offer starting code for all the tasks and **you are expected to implement the key functions using only Python and Numpy**.

2 Notices

1. If you find some powerful functions in Numpy that can easily solve a task, please ask us if this is allowed before use.
2. **If a task doesn't indicate "for loops" are allowed, using "for loop/while" in your code will be penalized (1 point for 1 use).** Note that *np.vectorized* is also not allowed since it is not parallel computing and the speed is the same as "for loop". Some useful Numpy functions are included in Appendix 5 for your information.
3. **Please update your personal information in *pack.py*, run the *pack.py* script to compress your code and results and then submit the zip file to course.pku.edu.cn.**
4. Feel free to post in the discussion panel for any questions and we encourage everyone to report the potential improvements of this assignment with a bonus of up to 5 points.

3 Tasks

1. **Convolution operation (15 points):**

   Convolution is one of the most common operations used in both deep learning and image processing. In this section, you are required to implement 2D convolution in two different ways. You will be able

to examine your convolution operator in the Gaussian filter and Sobel filter. Please complete the *HM1_convolve.py*.

a)[5 points] Implementing Padding Function

You are required to implement a padding function that takes inputs a 2D image, a padding size (the number of circles around the image), and a padding mode (zero padding or replication padding). Here replication padding refers to padding the input tensor using the replication of the input boundary (see Appendix 2). Note that you are **not allowed** to leverage the *numpy.pad* function.

b)[5 points] Implementing 2D Convolution via Toeplitz Matrix

The convolution can be implemented as a single matrix multiplication, which is called Toeplitz matrix (see Appendix 1). For a 6×6 input 2D array and a 3×3 filter, you are requested to construct the doubly block Toeplitz matrix and then perform the convolution on the 2D array using your Toeplitz matrix. Here we assume that the convolution uses zero padding so that the output size remains the same.

Hint: you will need to construct a 36×36 *Toeplitz matrix if your input array is not padded or a* 36×64 *Toeplitz matrix if you pad the input first. You can avoid using for loops since Numpy allows you to index many matrix elements at the same time and assign values to them together.*

c)[5 points] Convolution by Sliding Window

Convolution implemented by Toeplitz matrix is fully parallel, but it requires you to construct a highly sparse Toeplitz matrix (nearly $N^2 \times N^2$) for an image with $N \times N$ size. An alternative for implementing convolution is through sliding window, which allows you to simply compute the dot product between your image values and the kernel inside each window. In this question, you are requested to implement a convolution operator that can take a 2D image with an arbitrary shape as input and perform convolution with $k \times k$ kernel. You can assume no padding when implementing this convolution operator.

Hint: you will need to construct a $(N - k + 1)^2 \times k^2$ *matrix from the input image to support parallel computing. To avoid using any loop operations, you can use np.meshgrid to generate indices.*

d)[0 points] Gaussian Filter

With the sliding-window convolution, we can simply construct a 3×3 Gaussian filter. In the code, we will use your implemented padding function in replication mode in a) and your implemented sliding-window convolution in c) to construct a 3×3 Gaussian filter.

e)[0 points] Sobel Filter

In Lecture 2, we talked about using the finite difference to approximate image gradients. A more commonly used way is actually to perform Sobel filter (see Appendix 3). In the code, we will use your implemented padding function with reflection mode in a) and the sliding-window convolution function in c) to construct a Sobel filter.

2. **Canny Edge Detector (30 points):**

In this task, you will build your own Canny edge detector. Specifically, there are three key functions to be implemented, image gradient computing, non-maximal suppression and edge linking (with hysteresis threshold). Please complete the *HM1_Canny.py*, and you will see a similar result as Fig. 1.

Figure 1: Canny edge detector with Lenna

a)[10 points] Compute the Image Gradient.

We will use Sobel filter to approximate the image gradient – x-derivative $\frac{\partial I}{\partial x}$ and y-derivative $\frac{\partial I}{\partial y}$. Then, we can obtain the magnitude $\mathbf{M}$ and the orientation $\mathbf{D}$ of the gradient as:

$$\begin{aligned} \mathbf{M} &= \sqrt{\left(\frac{\partial I}{\partial x}\right)^2 + \left(\frac{\partial I}{\partial y}\right)^2}, \\ \mathbf{D} &= \arctan\left(\frac{\partial I}{\partial y} \cdot \frac{\partial x}{\partial I}\right). \end{aligned} \tag{1}$$

The magnitude and direction give a rough estimation of the edges, including the possibility of a pixel lying in the edge (magnitude) and the normal of the edge (direction). You will complete the *compute_gradient_magnitude_direction* function.

b)[10 points] Non-Maximal Suppression (NMS)

After obtaining the magnitude and direction of the gradients, you should check each pixel and remove unwanted pixels which may not constitute the edge. A common technique to achieve this goal is NMS. NMS finds the local maximum along the gradient direction, which could lead to "thin edges". Please complete the *non_maximal_suppressor* function as introduced in slides. To reduce the difficulty, you can implement the simplified version as in Lec. 3 Page 13.

c)[10 points] Edge Linking with Hysteresis Threshold

The edges resulted from NMS are often too thin and may not be connected. To link the potential discontinuous edges, we should do edge linking. Please complete the *hysteresis_thresholding* function **("for loop" is allowed in this question)**. To reduce the difficulty, you don't have to use the direction of the gradients and can simply connect adjacent pixels if the magnitude is larger than the lower threshold.

3. **Harris Corner Detector (30 points):**

One of the key topics in computer vision is to find the salient and accurate point features of given images. One early attempt to find these features is called Harris corner detector. In this question, you are requested to implement the corner response function to investigate the property of the intensity

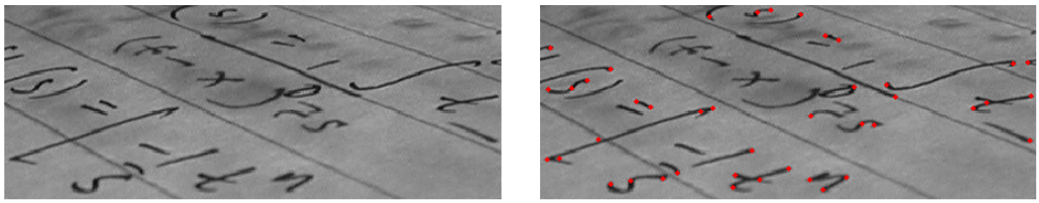

Figure 2: Harris corner detector with handwriting.png

changes of a given window. Please complete the *HM1_HarrisCorner.py*, and you will see a similar result as Fig. 2.

Hint: you only need to implement the rectangle window version introduced in the slides.

4. **Plane Fitting using RANSAC (25 points):**

In this task, you are expected to implement a plane-fitting algorithm with RANSAC to find a plane from a noisy 3D point cloud. Specifically, you need to perform the following steps.

1) hypothesis generation: For one hypothesis, you need to randomly select a group of seed points and then estimate the plane parameters. Please figure out the minimal sample time that can theoretically guarantee the probability of at least one hypothesis does not contain any outliers is larger than 99.9%. You are expected to generate all hypotheses **in parallel**.

2) verification: find and count the inliers for all the hypotheses.

3) final refinement: select the hypothesis with the largest number of inliers as your best hypothesis and estimate the final plane parameters with its inliers using the least-square method, which minimizes the sum of squared perpendicular distances between the points and the plane.

Please complete the *HM1_RANSAC.py*, and you will see a similar result as Fig. 3.

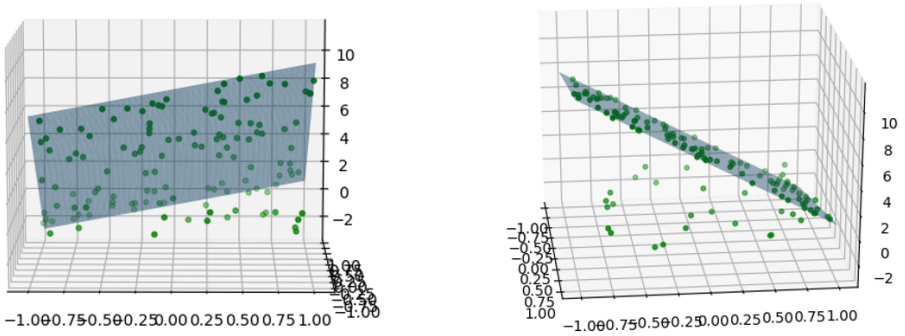

Figure 3: Plane (blue) fitting by RANSAC; Points (green).

Appendix

1. 1D Convolution with Toeplitz Matrix.
https://en.wikipedia.org/wiki/Toeplitz_matrix#Discrete_convolution

2. Replication padding in PyTorch.
https://pytorch.org/docs/stable/generated/torch.nn.ReplicationPad2d.html

3. OpenCV tutorial on Sobel derivative.
https://docs.opencv.org/3.4/d2/d2c/tutorial_sobel_derivatives.html

4. Backpropagation for a Linear Layer from Justin Johnson. http://cs231n.stanford.edu/handouts/linear-backprop.pdf

5. We recommend some handy Numpy functions which may help your tensor-style coding.

- meshgrid, https://numpy.org/doc/stable/reference/generated/numpy.meshgrid.html
- concatenate, https://numpy.org/doc/stable/reference/generated/numpy.concatenate.html
- where, https://numpy.org/doc/stable/reference/generated/numpy.where.html
- argmax, https://numpy.org/doc/stable/reference/generated/numpy.argmax.html
- linalg.svd, https://numpy.org/doc/stable/reference/generated/numpy.linalg.svd.html
- arctan2, https://numpy.org/doc/stable/reference/generated/numpy.arctan2.html