MSU CSE 803 Computer Vision: Homework 2

Instructions

- This homework is **due at 11:59:59 p.m. on Thu Oct 3rd, 2024**.
- Please submit to D2L. The submission includes twoparts:
  1. A pdf file as your write-up, including your plots and answers to all the questions and key choices you made.
     The write-up is preferred to be an electronic version. Handwriting is allowed, but should be minimized. LATEX is recommended but not mandatory.
     You might like to combine several files to make a submission. Here is an example online link for combining multiple PDF files: https://combinepdf.com/.
  2. A zip file including all your code, and files specified in questions with **Submit**, all under the same directory. You can submit Python code in either .py or .ipynb format.

Python Environment

We are using Python 3.7 for this course. You can find references for the Python standard library here: https://docs.python.org/3.7/library/index.html. To make your life easier, we **recommend** you to install Anaconda 5.2 for Python 3.7.x (https://www.anaconda.com/download/). This is a Python package manager that includes most of the modules you need for this course.

We will make use of the following packages extensively in this course:

- Numpy (https://docs.scipy.org/doc/numpy-dev/user/quickstart.html)
- OpenCV (https://opencv.org/)
- SciPy (https://scipy.org/)
- Matplotlib (http://matplotlib.org/users/pyplot_tutorial.html)

The problems are prepared by Dr. David Fouhey.

$$\frac{x_L}{f} = \frac{x_R - x_L}{b} = \frac{z - f}{z}$$

$$\frac{d}{b} = 1 - \frac{f}{z} = \frac{z - f}{z}$$

$\frac{d-b}{b} = -\frac{f}{z}$ $z = \frac{fb}{b-d}$

1 Image Filtering [50 pts]

In this first section, you will explore different ways to filter images. Through these tasks you will build up a toolkit of image filtering techniques. By the end of this problem, you should understand how the development of image filtering techniques has led to convolution.

Image Patches [8 pts] A patch is a small piece of an image. Sometimes we will focus on the patches of an image instead of operating on the entire image itself.

1. (5 pts) Take the image `'grace_hopper.png'`, load it as grayscale, and divide the image into 16 by 16 pixel image patches. Normalize each patch to have zero mean and unit variance. Complete the function `image_patches` in `filters.py`. **Plot** three of the 16x16 image patches in your report.
2. (3 pts) Early work in computer vision used unique images patches as descriptors or features of images for applications ranging from image alignment and stitching to object classification and detection. Inspect the patches extracted in the previous question, and **discuss**, in a few sentences, why they would be good or bad descriptors. Consider how those patches would look like if we changed the object's pose, scale, illumination, etc.

Gaussian Filter [16 pts] A Gaussian filter is a filter whose impulse response is a Gaussian function. Here we only talk about the discrete kernel and assume 2D Gaussian distribution is circularly symmetric.

$$\text{1D kernel} : G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}} \qquad \text{2D kernel} : G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

1. (5 pts) For a 2D Gaussian filter with a given variance σ^2, the convolution can be reduced by sequential operations of a 1D kernel. **Prove** that a convolution by a 2D Gaussian filter is equivalent to sequential convolutions of a vertical and a horizontal 1D Gaussian filter. **Specify** the relationship between the 2D and 1D Gaussian filter, especially the relationship between their variances.
2. (4 pts) Take the image `'grace_hopper.png'` as the input. Complete the function `convolve()` and other related parts in `filters.py`. Use a Gaussian kernel with size 3×3 and $\sigma^2 \approx \frac{1}{2\ln 2}$. **Plot** the output images in your report. **Describe** what Gaussian filtering does to the image in one sentence. Be sure to implement convolution and not cross-correlation.
3. (3 pts) Consider the image as a function $I(x,y)$ and $I : \mathbb{R}^2 \to \mathbb{R}$. When working on edge detection, we often pay a lot of attention to the derivatives. Denote the derivatives:

$$I_x(x,y) = \frac{\partial I}{\partial x}(x,y) \approx \frac{1}{2}(I(x+1,y) - I(x-1,y))$$
$$I_y(x,y) = \frac{\partial I}{\partial y}(x,y) \approx \frac{1}{2}(I(x,y+1) - I(x,y-1))$$

   Derive the convolution kernels for derivatives: (i) $k_x \in \mathbb{R}^{1\times 3}$: $I_x = I * k_x$; (ii) $k_y \in \mathbb{R}^{3\times 1}$: $I_y = I * k_y$. Follow the detailed instructions in `filters.py` and complete the function `edge_detection()` in `filters.py`, whose output is the gradient magnitude.
4. (4 pts) Use the original image and the Gaussian-filtered image as inputs respectively. **Plot** both outputs in your report. **Discuss** the difference between the two images in no more than three sentences.

Sobel Operator [18 pts] The Sobel operator is often used in image processing and computer vision. Technically, it is a discrete differentiation operator, computing an approximation of the derivative of the image intensity function.

1. (5 pts) Focus on the derivatives of the Gaussian-filtered image. Suppose we use this Gaussian kernel:

$$k_{Gaussian} = \begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}$$

   Denote the input image as I. **Prove** that the derivatives of *the Gaussian-filtered image ($I * k_{Gaussian}$)* can be approximated by :

$$G_x = I * \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix} \qquad G_y = I * \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$

   These are the sobel operators.
   Hint: To derive the derivatives of an image, you can use the conclusion in **Gaussian Filter - Q3**, *i.e.* G_x can be calculated by $(I * k_{Gaussian})_x$ using the k_x derived before.

2. (4 pts) Take the image `'grace_hopper.png'` as the original image I. Complete the corresponding part of function `sobel_operator()` in `filters.py` with the kernels given previously. **Plot** the G_x, G_y, and the gradient magnitude.

3. Now we want to explore what will happen if we use a linear combination of the two Sobel operators. Here, the linear combination of the original kernels is called *a steerable filter*.

   (a) (3 pt) We want to use 3x3 kernel to approximate $S(I, \alpha) = G_x \cos\alpha + G_y \sin\alpha$. **Derive** the kernel in terms of α, *i.e.* the $K(\alpha)$ which makes $S(I, \alpha) = I * K(\alpha)$.

   (b) (3 pts) Complete the function `steerable_filter()` in `filters.py`.
   Take $\alpha = 0, \frac{\pi}{6}, \frac{\pi}{3}, \frac{\pi}{2}, \frac{2\pi}{3}, \frac{5\pi}{6}$. **Plot** the output images in your report.

   (c) (3 pts) Observe the plotting results. What do these kernels detect? **Discuss** how the outputs change with α.

LoG Filter [8 pts] Laplacian is a differential operator: $\nabla^2 I(x, y) = \frac{\partial^2 I(x,y)}{\partial x^2} + \frac{\partial^2 I(x,y)}{\partial y^2}$. And the Laplacian of Gaussian (LoG) operation is very useful in computer vision.

1. (5 pts) In `filters.py`, you are given two LoG filters. You are not required to prove that they are LoG, but you are encouraged to know what an LoG filter looks like. Complete the corresponding part in `filters.py`. **Plot** the outputs of these two LoG filters in your report. **Compare** the two results. **Explain** the reasons for the difference. **Discuss** whether these filters can detect edges. Can they detect anything else?
   Hint: We can take the high-value pixels in the outputs as the detected parts.

2. (3 pts) Instead of calculating LoG, we can often approximate it with a simple Difference of Gaussians (DoG). Try to explain why this approximation works.
   Hint: Try visualizing the following functions: two Gaussian functions with different variances, the difference between the two functions, Laplacian of a Gaussian function.

2 Feature Extraction [15 pts]

While edges can be useful, corners are often more informative features as they are less common. In this section, we implement a Harris Corner Detector (see: https://en.wikipedia.org/wiki/Harris_Corner_Detector) to detect corners. *Corners* are defined as locations (x, y) in the image where a small change any direction results in a large change in intensity if one considers a small window centered on (x, y) (or, intuitively, one can imagine looking at the image through a tiny hole that's centered at (x, y)). This can be contrasted with *edges* where a large intensity change occurs in only one direction, or *flat regions* where moving in any direction will result in small or no intensity changes. Hence, the Harris Corner Detector considers small windows (or patches) where a small change in location leads large variation in multiple directions (hence corner detector).

This question looks a bit long, but that is only because there is a fairly large amount of hand-holding involved. The resulting solution, if done properly, is certainly under 10 lines.

2.1 Corner Score [5pt]

Let's consider a grayscale image where $I(x, y)$ is the intensity value at image location (x, y). We can calculate the corner score for every pixel (i, j) in the image by comparing a window W centered on (i, j) with that same window centered at $(i+u, j+v)$. Specifically, we will compute the sum of square differences between the two,

$$E(u, v) = \sum_{x,y \in W} [\mathbf{I}(x + u, y + v) - \mathbf{I}(x, y)]^2$$

or, for every pixel (x, y) in the window W centered at i, j, how different is it from the same window, shifted over (u, v). This formalizes the intuitions above:

- If moving (u, v) leads to no change for all (u, v), then (x, y) is probably flat.
- If moving (u, v) in one direction leads to a big change and adding (u, v) in another direction leads to a small change in the appearance of the window, then (x, y) is probably on an edge.
- If moving any (u, v) leads to a big change in appearance of the window, then (x, y) is a corner.

You can compute this $E(u, v)$ for all (u, v) and at all (i, j).

Your first task is to write a function that calculates this function for all pixels (i, j) with a **fixed** offset (u, v) and window size W. In other words, if we calculate $\mathbf{S} = \text{cornerscore}(u, v)$, $\mathbf{S}$ is an image such that $\mathbf{S}_{ij}$ is the SSD between the window centered on (i, j) in I and the window centered on $(i + u, j + v)$ in I. The function will need to calculate this function to every location in the image. This is doable via a quadruple for-loop (for every pixel (i, j), for every pixel (x, y) in the window centered at (i, j), compare the two). Use same padding for offset-window values that lie outside of the image.

Complete the function `corner_score()` in `corners.py` which takes as input an image, offset values (u, v), and window size W. The function computes the response $E(u, v)$ for every pixel. We can look at, for instance the image of $E(0, y)$ to see how moving down y pixels would change things and the image of $E(-x, 0)$ to see how moving left x pixels would change things.

Plot your output for u, v that shift things left/right/up/down by 5 pixels.

Early work by Moravec [1980] used this function to find corners by computing $E(u, v)$ for a range of offsets and then select the pixels where the corner score why high for all offsets. In a few sentences, **discuss** why this might be impractical.

2.2 Harris Corner Detector [10pt]

For every single pixel (i, j), you now have a way of computing how much changing by (u, v) changes the appearance of a window (i.e., $E(u, v)$ at (i, j)). But in the end, we really want a single number of "cornerness" per pixel and don't want to handle checking all the (u, v) values at every single pixel (i, j). You'll implement the cornerness score invented by Harris and Stephens [1988].

Harris and Stephens recognized that if you do a Taylor series of the image, you can build an approximation of $E(u, v)$ at a pixel (i, j). Specifically, if $\mathbf{I}_x$ and $\mathbf{I}_y$ denote the image of the partial derivatives of $\mathbf{I}$ with respect to x and y, then

$$E(u,v) \approx \sum_W \left(\mathbf{I}_x^2 u^2 + 2\mathbf{I}_x\mathbf{I}_y uv + \mathbf{I}_y^2 v^2\right) = [u,v] \begin{bmatrix} \sum_W \mathbf{I}_x^2 & \sum_W \mathbf{I}_x\mathbf{I}y \\ \sum_W \mathbf{I}_x\mathbf{I}_y & \sum_W \mathbf{I}_y^2 \end{bmatrix} [u,v]^T = [u,v]\mathbf{M}[u,v]^T$$

where $\mathbf{M}$ is called the structure tensor. To avoid extreme notation clutter, the sums are over x, y in a window W centered at i, j and any image (e.g., I_x) is assumed to be indexed by x, y. In other words: the top-left entry of $\mathbf{M}$ is $\sum_{(x,y)\in W}(\mathbf{I}_x^2)_{x,y}$ where W is the set of pixels in a window centered on (i, j). This matrix $\mathbf{M}$ is a reasonable approximation of how the window changes at each pixel and you can compute $\mathbf{M}$ at every single pixel (i, j) in the image.

What does this do for our lives? We can decompose the $\mathbf{M}$ we compute at each pixel into a rotation matrix $\mathbf{R}$ and diagonal matrix $\mathrm{diag}([\lambda_1, \lambda_2)$ such that (specifically an eigen-decomposition):

$$\mathbf{M} = \mathbf{R}^{-1}\mathrm{diag}([\lambda_1, \lambda_2])\mathbf{R}$$

where the columns of $\mathbf{R}$ tell us the directions that $E(u, v)$ most and least rapidly changes, and λ_1, λ_2 tell us the maximum and minimum amount it changes. In other words, if both λ_1 and λ_2 are big, then we have a corner; if only one is big, then we have an edge; if neither are big, then we are on a flat part of the image. Unfortunately, doing this decomposition is slow, and Harris and Stephens were doing this over 30 years ago.

Harris and Stephens had two other tricks up their sleeve. First, rather than calculate the eigenvalues directly, for a 2x2 matrix, one can compute the following score, which is a reasonable measure of what the eigenvalues are like:

$$R = \lambda_1\lambda_2 - \alpha(\lambda_1 + \lambda_2) = \det(\mathbf{M}) - \alpha\mathrm{trace}(\mathbf{M})^2$$

which is far easier since the determinants and traces of a 2x2 matrix can be calculated very easily (look this up). Pixels with large positive R are corners; pixels with large negative R are edges; and pixels with low R are flat. In practice α is set to something between 0.04 and 0.06. Second, the sum that's being done weights pixels across the window equally, when we know this can cause trouble. So instead, Harris and Stephens computed a $\mathbf{M}$ where the contributions of $\mathbf{I}_x$ and $\mathbf{I}_y$ for each pixel (i, j) were weighted by a Gaussian kernel.

Ok, so how do I implement it?

1. In your implementation, you should first figure out how to calculate $\mathbf{M}$ for all pixels just using a straight-forward sum.

   You can compute it by brute force (quadruple for-loop) or convolution (just summing over a window). In general, it's usually far easier to write a slow-and-not-particularly-clever version that does it brute force. This is often a handful of lines and requires not so much thinking. You then write a version that is convolutional and faster but requires some thought. This way, if you have a bug, you can compare with the brute-force version that are pretty sure has no issues.

   You can store $\mathbf{M}$ as a 3-channel image where, for each pixel (i, j) you store $\mathbf{M}_{1,1}$ in the first channel, $\mathbf{M}_{1,2}$ in the second and $\mathbf{M}_{2,2}$ in the third. Storing $\mathbf{M}_{2,1}$ is unnecessary since it is the same as $\mathbf{M}_{1,2}$.

2. You should then figure out how to convert $\mathbf{M}$ at every pixel into R at every pixel. This is a set of operations (det, trace) that you can look up for 2x2 matrices and which you can do via element-wise operations on the image representing $\mathbf{M}$. If you're feeling adventuresome (and as a way of testing your implementation), you can also do a for loop over the whole image and compute the smallest eigenvalue and look at that too.

3. Finally, you should switch out summing over the window (by convolution or brute force) to summing over the window with a Gaussian weight and by convolution. The resulting operation will be around five cryptic lines that look like magic but which are doing something eminently sensible if you understand why.

Implement this optimization by completing the function `harris_detector()` in `corners.py`. **Generate** a Harris Corner Detector score for every point in a grayscale version of `'grace_hopper.png'`, and **plot** these scores as a heatmap and include it in your report.

You cannot call a library function that has already implemented the Harris Corner Detector.

3 Blob Detection [35pts]

One of the great benefits of computer vision is that it can greatly simplify and automate otherwise tedious tasks. For example, in some branches of biomedical research, researchers often have to count or annotate specific particles microscopic images such as the one seen below. Aside from being a very tedious task, this task can be very time consuming as well as error-prone. During this course, you will learn about several algorithms that can be used to detect, segment, or even classify cells in those settings. In this part of the assignment, you will use the DoG filters implemented in part 1 along with a scale-space representation to count the number of cells in a microscopy images.

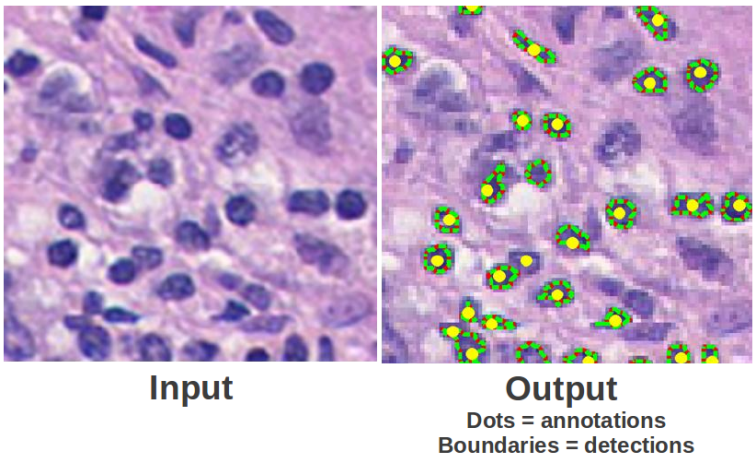

Figure 1: Detected Lymphocytes in breast cancer pathology images. Source: Oxford VGG

Single-scale Blob Detection [10 pt] Your first task is to use DoG filters to detect blobs of a single scale. Implement the function `guassian_filter` in `blob_detection.py` that takes as an input an image and the standard deviation, σ, for a Gaussian filter and returns the Gaussian filtered image. Read in `'polka.png'` as a gray-scale image and find two pairs of σ values for a DoG such that the first set responds highly to the large circles, while the second set only responds highly the small circles. For choosing the appropriate sigma values, recall that radius and standard deviation of a Gaussian are related by the following equation: $r = \sqrt{2\sigma}$. **Plot** the two responses and report the parameters used to obtain each. **Comment** on the responses in a few lines: how many maxima are you observing? are there false peaks that are getting high values?

Scale Space Representation [10 pt] In this part, we will construct a Scale Space Representation of the image to detect blobs at multiple scales. In this representation, we represent the image by the DoG at different scales by filtering the images with Gaussian kernels of different width; *i.e.*, different standard deviation. The image below shows a scale space DoG response of an image of a sunflower field; notice that different size circles have higher intensities at different scales. Your task is to implement the function `scale_space` in `blob_detection.py`. The function takes as input the image, the smallest standard deviation (σ_{min}), the standard deviation multiplier (k), and the number of scales (S). You are advised to use default values for k and S as they capture a sufficiently wide range of scales (8 scales with the largest scale being 16x the smallest) The function should construct a scale space with $S-1$ levels, such that level i is the Difference of Gaussian filtered images with standard deviation ($\sigma_{min} * k^i$) and ($\sigma_{min} * k^{i-1}$).

Find good parameters for the scale space function to detect both polka dot sizes. **Generate** the scale space output for `'polka.png'`. For parameters, we suggest using $k = \sqrt{2}, S = 8$ and the first sigma value for detecting small polka dots (from the first part of this question) as your σ_{min}. Feel free to use a different set of parameters, but the values suggested should serve as a good initialization. **Plot** the different scales using `visualize_scale_space()`. Are you able to clearly see the different maxima in the image?

Blob Detection [5 pt] While we can visually identify peaks with relative ease, automatic maxima detection can result in spurious false positives or false negatives. Your task here is to automatically detect the peaks from the 3D matrix scale space response output in the previous function. We have provided a function, `find_maxima`, that finds outputs finds the local maxima with respect to a small neighborhood in the scale space. A local maxima is defined as a value that is higher than all values within a given window surrounding it. The value has two parameters; `k_xy` which is the size of the window in (x, y) and `k_s` which is the size of the window in scale space. Forexample, a window with `k_xy`$= 3$ and `k_s`$= 1$ will check all values that are within 3 pixel location in (x, y) and within 1 pixel in scale. **Experiment** with different `k_xy`, `k_s` values, and find the pair that would only find the correct maxima on the scale space extracted for `'polka.jpg'`. **Discuss** in a few sentences how different choices of window size affect detection of false positives (detected maxima where no blob exist).

Cell Counting [10 pt] In computer vision, we often have to choose the correct set of parameters depending on our problem space (or learn them; more on that later in the course). Your task here to to apply blob detection to find the number of cells in 4 images of your choices from the images found in the `/vgg_cells` folder. **Find** a set of parameters for generating the scale space and finding the maxima that allows you to accurately detect the cells in each of those images. Feel free to pre-process the images or the scale space output to improve detection. **Report** the parameters used for the blob detection as well as the number of detected cells for each of the images. Make sure to include the visualized blob detection for each of the images in your report. **Discuss** your results as well as any additional steps you took to improve the cell detection and counting.

Note: The images come from a project from the Visual Geometry Group at Oxford University, and have been used in a recent research project that focuses on counting cells and other objects in images; you can check their work here. This is an open-ended question, so your detection don't have to be perfect. You will be primarily graded on showing that you tried a few different things and your analysis of your results.

References

C. Harris and M. Stephens, A Combined Corner and Edge Detector, in Proceedings of the Alvey Vision Conference 1988, Manchester, 1988, pp. 23.123.6.

H. Moravec, "Obstacle Avoidance and Navigation in the Real World by a Seeing Robot Rover", Tech Report CMU-RI-TR-3, Carnegie-Mellon University, Robotics Institute, September 1980.

Lowe, David G. "Distinctive image features from scale-invariant keypoints." International Journal of Computer Vision 60.2 (2004): 91-110.

https://en.wikipedia.org/wiki/Feature_detection_(computer_vision)

Cell counting project: http://www.robots.ox.ac.uk/~vgg/research/counting/index_org.html