EECS 442 Computer Vision: Homework 2

Instructions

- This homework is **due at 11:59:59 p.m. on Thursday February 17th, 2022**.
- The submission includes two parts:
  1. **To Canvas:** submit a `zip` file of all of your code.
     We have indicated questions where you have to do something in code in red.
     Your zip file should contain a single directory which has the same name as your uniqname. If I (David, uniqname `fouhey`) were submitting my code, the zip file should contain a single folder `fouhey/` containing all required files.
     What should I submit? At the end of the homework, there is a canvas submission checklist provided. We provide a script that validates the submission format here. If we don't ask you for it, you don't need to submit it; while you should clean up the directory, don't panic about having an extra file or two.
  2. **To Gradescope:** submit a `pdf` file as your write-up, including your answers to all the questions and key choices you made.
     We have indicated questions where you have to do something in the report in blue.
     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/.
     The write-up must be an electronic version. **No handwriting, including plotting questions.** LaTeX is recommended but not mandatory.

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 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)
- SciPy (https://scipy.org/)
- Matplotlib (http://matplotlib.org/users/pyplot_tutorial.html)

1 Patches [8 pts]

Task 1: 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.

(a) **Complete the function** `image_patches` in `filters.py`. This should divide a grayscale image into a set of non-overlapping 16 by 16 pixel image patches. Normalize each patch to have zero mean and unit variance.

Plot and put in your report three 16x16 image patches from `grace_hopper.png` loaded in grayscale. (3 pts)

(b) **Discuss in your report** why you think it is good for the patches to have zero mean. (2 pts)

Hint: Suppose you want to measure the similarity between patches by computing the dot products between different patches to find a match. Think about how the patch values and the resulting similarity obtained by taking dot products would be affected under different lighting/illumination conditions. Say in one case a value of dark corresponds to 0 whereas bright corresponds to 1. In another scenario a value of dark corresponds to -1 whereas bright corresponds to 1. Which one would be more appropriate to measure similarity using dot products?

(c) Early work in computer vision used patches as descriptions of local image content for applications ranging from image alignment and stitching to object classification and detection.

Discuss in your report in 2-3 sentences, why the patches from the previous question would be good or bad for things like matching or recognizing an object. Consider how those patches would look like if we changed the object's pose, scale, illumination, etc. (3 pts)

2 Image Filtering [46 pts]

Foreword: There's a difference between convolution and cross-correlation: in cross-correlation, you compute the dot product (i.e., `np.sum(F*I[y1:y2,x1:x2])`) between the kernel/filter and each window/patch in the image; in convolution, you compute the dot product between the *flipped* kernel/filter and each window/patch in the image. We'd like to insulate you from this annoying distinction, but we also don't want to teach you the wrong stuff. So we'll split the difference by pointing where you have to pay attention.

We'll make this more precise in 1D: assume the input/signal f has N elements (i.e., is indexed by i for $0 \leq i < N$) and the filter/kernel g has M elements (i.e., is indexed by j for $0 \leq j < M$). In all cases below, you can assume *zero-padding* of the input/signal f.

1D Cross-correlation/Filtering: The examples given in class and what most people think of when it comes to filtering. Specifically, 1D cross-correlation/filtering takes the form:

$$h[i] = \sum_{j=0}^{M-1} g[j] f[i+j], \tag{1}$$

or each entry i is the sum of all the products between the filter at j and the input at $i+j$ for all valid j. If you want to think of doing this in terms of matrix products, you can think of this as $\mathbf{h}_i = \mathbf{g}^T \mathbf{f}_{i:i+M-1}$. Of the two options, this tends to be more intuitive to most people.

1D Convolution: When we do 1D convolution, on the other hand, we re-order the filter last-to-first, and then do filtering. In signal processing, this is usually reasoned about by index trickery. By definition, 1D convolution takes the form:

$$(f * g)[i] = \sum_{j=0}^{M-1} g[M-j-1]f[i+j], \tag{2}$$

which is uglier since we start at 0 rather than 1. You can verify that as j goes $0 \to (M-1)$, the new index $(M-j-1)$ goes $(M-1) \to 0$. Rather than deal with annoying indices, if you're given a filter to apply and asked to do convolution, you can simply do the following: (1) at the start of the function and only once, compute `g = g[::-1]` if it's 1D or `G = G[::-1,::-1]`; (2) do filtering with this flipped filter.

The reason for the fuss is that convolution is commutative ($f * g = g * f$) and associative ($f * (g * h) = (f * g) * h$). As you chain filters together, its is nice to know things like that $(a * b) * c = (c * a) * b$ for all a, b, c. Cross-correlation/filtering does not satisfy these properties.

You should watch for this in three crucial places.

- When implementing convolution in Task 2(b) in the function `convolve()` in `filters.py`.
- When dealing with non-symmetric filters (like directional derivatives $[-1, 0, 1]$). A symmetric filter like the Gaussian is unaffected by the distinction because if you flip it horizontally/vertically, it's the same. But for asymmetric filters, you can get different results. In the case of the directional derivatives, this flips the sign. This can produce outputs that have flipped signs or give you answers to question that are nearly right but need to be multiplied by -1.

  Bottom-line: if you have something that's right except for a -1, and you're using a directional derivative, then you've done it with cross-correlation.
- In a later homework where you implement a convolutional neural network. Despite their name, these networks actually do cross-correlation. Argh! It's annoying.

  Here's my key: if you're trying to produce a picture that looks clean and noise-free or you're talking to someone who talks about sampling rates, "convolution" is the kind of convolution where you reverse the filter order. If you're trying to recognize puppies or you're talking to someone who doesn't frequently say signal, then "convolution" is almost certainly filtering/cross-correlation.

Task 2: Convolution and Gaussian Filter (21 pts)

(a) A Gaussian filter has filter values that follow the Gaussian probability distribution. Specifically, the values of the filter are

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

where 0 is the center of the filter (in both 1D and 2D) and σ is a free parameter that controls how much blurring takes place. One thing that makes lots of operations fast is that applying a 2D Gaussian filter to an image can be done by applying two 1D Gaussian filters, one vertical and the other horizontal.

Show in your report that a convolution by a 2D Gaussian filter is equivalent to sequentially applying a vertical and horizontal Gaussian filter. (3 pts)

Advice: Pick a particular filter size k. From there, define a 2D Gaussian filter $\mathbf{G} \in \mathbb{R}^{k \times k}$ and two Gaussian filters $\mathbf{G}_y \in \mathbb{R}^{k \times 1}$ and $\mathbf{G}_x \in \mathbb{R}^{1 \times k}$. A useful fact that you can use is that for any k, any

vertical filter $\mathbf{X} \in \mathbb{R}^{k \times 1}$ and any horizontal filter $\mathbf{Y} \in \mathbb{R}^{1 \times k}$, the convolution $\mathbf{X} * \mathbf{Y}$ is equal to $\mathbf{XY}$. Expanded out for $k = 3$, this just means

$$\begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} * \begin{bmatrix} Y_1 & Y_2 & Y_3 \end{bmatrix} = \begin{bmatrix} X_1Y_1 & X_1Y_2 & X_1Y_3 \\ X_2Y_1 & X_2Y_2 & X_2Y_3 \\ X_3Y_1 & X_3Y_2 & X_3Y_3 \end{bmatrix} \tag{3}$$

You may find it particularly useful to use the fact that $\mathbf{Y} * \mathbf{Y}^T = \mathbf{Y}\mathbf{Y}^T$ and that convolution is associative. Look at individual elements. If you do this correctly, the image does not have to be involved at all.

If you have not had much experience with proofs or need a refresher, this guide will help you get started. Here is another link that will help you write readable and easy to follow solutions. But in general, the key isn't formality, but just being precise.

(b) **Complete the function** `convolve()` in `filters.py`. Be sure to implement convolution and not cross-correlation/filtering (i.e., flip the kernel as soon as you get it). For consistency purposes, please use **zero-padding** when implementing convolution. (4 pts)

Advice: You can use `scipy.ndimage.convolve()` to check your implementation. For zero-padding use `mode='constant'`. Refer to documentation for details. For Part 3 Feature Extraction and Part 4 Blob Detection, directly use scipy's convolution function with the same settings, ensuring zero-padding.

(c) **Plot the following output and put it in your report** and then describe what Gaussian filtering does to the image in one sentence. Load the image `grace_hopper.png` as the input and apply a Gaussian filter that is 3×3 with a standard deviation of $\sigma = 0.572$. (2 pts)

(d) **Discuss in your report** why it is a good idea for a smoothing filter to sum up to 1. (3 pts)

Advice: As an experiment to help deduce why, observe that if you sum all the values with of the Gaussian filter in (c), you should get a sum close to 1. If you are very particular about this, you can make it exactly sum to 1 by dividing all filter values by their sum. When this filter is applied to `'grace_hopper.png'`, what are the output intensities (min, max, range)? Now consider a Gaussian filter of size 3×3 and standard deviation $\sigma = 2$ (but do not force it to sum to 1 – just use the values). Calculate the sum of all filter values in this case. What happens to the output image intensities in this case? If you are trying to plot the resulting images using `matplotlib.pyplot` to compare the difference, set `vmin = 0` and `vmax = 255` to observe the difference.

(e) 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":

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

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

where I_x is the twice the derivative and thus off by a factor of 2. This scaling factor is not a concern since the units of the image are made up. So long as you are consistent, things are fine.

Derive in your report the convolution kernels for derivatives (3 pts):

(i) $k_x \in \mathbb{R}^{1\times 3}$: $I_x = I * k_x$

(ii) $k_y \in \mathbb{R}^{3\times 1}$: $I_y = I * k_y$

(f) Follow the detailed instructions in `filters.py` and **complete the function** `edge_detection()` in `filters.py`, whose output is the gradient magnitude. (3 pts)

(g) Use the original image and the Gaussian-filtered image as inputs respectively and use `edge_detection()` to get their gradient magnitudes. **Plot both outputs and put them in your report**. **Discuss in your report** the difference between the two images in no more than three sentences. (3 pts)

Task 3: Sobel Operator (9 pts) The Sobel operator is often used in image processing and computer vision.

(a) The Sobel filters S_x and S_y are given below and are related to a particular Gaussian kernel G_S:

$$S_x = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix} \qquad S_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix} \qquad G_S = \begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}.$$

Show in your report the following result: If the input image is I and we use G_S as our Gaussian filter, taking the horizontal-derivative (i.e., $\frac{\partial}{\partial x} I(x, y)$) of the *Gaussian-filtered image*, can be approximated by applying the Sobel filter (i.e., computing $I * S_x$). (5 pts)

Advice: You should use the horizontal filter k_x that you derive previously – in particular, the horizontal derivative of the Gaussian-filtered image is $(I * G_S) * k_x$. You can take advantage of properties of convolution so that you only need to show that two filters are the same. If you do this right, you can completely ignore the image.

(b) **Complete the function** `sobel_operator()` in `filters.py` with the kernels/filters given previously. (2 pts)

(c) **Plot the following and put them in your report:** $I * S_x$, $I * S_y$, and the gradient magnitude. with the image `'grace_hopper.png'` as the input image I. (2 pts)

Task 4: LoG Filter (9 pts) The Laplacian of Gaussian (LoG) operation is important in computer vision.

(a) In `filters.py`, you are given two LoG filters. You are not required to show that they are LoG, but you are encouraged to know what an LoG filter looks like. **Include in your report, the following:** the outputs of these two LoG filters and the reasons for their difference. **Discuss in your report** whether these filters can detect edges. Can they detect anything else? (3 pts)

Advice: By detected regions we mean pixels where the filter has a high **absolute** response.

(b) Instead of calculating a LoG, we can often approximate it with a simple Difference of Gaussians (DoG). Specifically many systems in practice compute their "Laplacian of Gaussians" is by computing $(I * G_{k\sigma}) - (I * G_\sigma)$ where G_a denotes a Gaussian filter with a standard deviation of a and $k > 1$ (but is

usually close to 1). If we want to compute the LoG for many scales, this can be *far* faster – rather than apply a large filter to get the LoG, one can get it for free by repeatedly blurring the image with little kernels.

Discuss in your report why computing $(I * G_{k\sigma}) - (I * G_{\sigma})$ might successfully roughly approximate convolution by the Laplacian of Gaussian. You should include a plot or two showing filters in your report.. To help understand this, we provide a three 1D filters with 501 entries in `log1d.npz`. Load these with `data = np.load('log1d.npz')`. There is a LoG filter with $\sigma = 50$ in variable `data['log50']`, and Gaussian filters with $\sigma = 50$ and $\sigma = 53$ in `data['gauss50']` and `data['gauss53']` respectively. You can look at these using matplotlib via `plt.plot(filter)` which will show you a nice line plot. You should assume these are representative samples of filters and that things generalize to 2D. (6 pts)

Advice: Try visualizing the following functions: two Gaussian functions with different variances, the difference between the two functions, the Laplacian of a Gaussian function. To explain why it works, remember that convolution is linear: $A * C + B * C = (A + B) * C$.

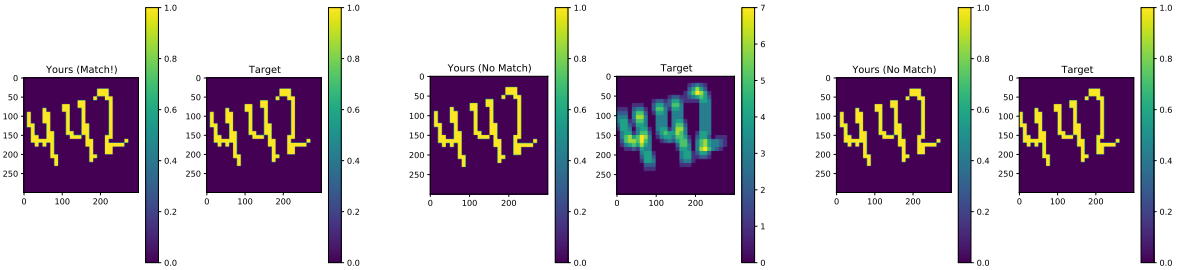

Figure 1: Outputs from `filtermon/filtermon.py`. If you put the right filter in, your outputs will match the reference output.

Task 5: Who's That Filter? (7 pts) In `filtermon/`, we've provided you with an image and its output for five different 3×3 filters. The zeroth filter (the identity [[0, 0, 0], [0, 1, 0], [0, 0, 0]]) has been correctly put into the code, and so its convolution with the image will match. Update filter1 through filter4; the code will check your answers.

(a) **Write out each of the four remaining filters. No need to format them prettily; something like [[0,0,0], [0,1,0], [0,0,0]] works.** (5 pts)

Advice: Watch out that the code does convolution. If you guess the filter based on the output, remember that the filter that gets used will be horizontally and vertically flipped/reflected! All filters look similar to filters that have been shown in class (although will not match precisely).

(b) **What does filter 1 do, intuitively and how does it differ from filter 2?** (2 pts)

3 Feature Extraction [22 pts]

This question looks long, but that is only because there is a fairly large amount of walk-through and formalizing topics. The resulting solution, if done properly, is certainly under 10 lines. If you use filtering, please use `scipy.ndimage.convolve()` to perform convolution whenever you want to use it. Please use **zero padding** for consistency purposes (Set `mode='constant'`).

Foreword: 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).

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 $(i + u, j + v)$. To be specific: a window of size $2d + 1$ centered on (i, j) is a the set of pixels between $i - d$ to $i + d$ and $j - d$ to $j + d$. 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).

Task 6: Corner Score (9 pts) 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 sum-of-squared differences 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). However, you can also imagine doing this by (a) offsetting the image by (u, v); (b) taking the squared difference with the original image; (c) summing up the values within a window using convolution. **Note:** If you do this by convolution, use **zero padding** for offset-window values that lie outside of the image.

(a) **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 right x pixels would change things. (3 pts)

Advice: You can use `np.roll` for offsetting by u and v. If you look really carefully, you'll notice that if you implement this function via `np.roll`, you'll have to watch out for whether you roll by u, v or by $-u, -v$ if you want to calculate this precisely correct. Don't worry about this flip ambiguity. Here's why: in practice, if you were to use this corner score, you would try a whole set of us and vs and aggregate the results (e.g., max, mean). In particular, for every u, v, you would also try $-u, -v$.

(b) **Plot and put in your report** your output for for `grace_hopper.png` for $(u, v) = \{(0, 5), (0, -5), (5, 0), (-5, 0)\}$ and window size $(5, 5)$ (3 pts)

(c) Early work by Moravec [1980] used this function to find corners by computing $E(u, v)$ for a range of offsets and then selecting the pixels where the corner score is high for all offsets.

Discuss in your report why checking all the us and vs might be impractical in a few sentences. (3 pts)

Foreword: 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 (computable via k_x and k_y from above), then

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

This matrix $\mathbf{M}$ has all the information needed to approximate how rapidly the image content changes within a window near each pixel and you can compute $\mathbf{M}$ at every single pixel (i, j) in the image. To avoid extreme notation clutter, we assume we are always talking about some fixed pixel i, j, the sums are over x, y in a $2d + 1$ window W centered at i, j and any image (e.g., I_x) is assumed to be indexed by x, y. But in the interest of making this explicit, we want to compute the matrix $\mathbf{M}$ at i, j. The top-left and bottom-right elements of the matrix $\mathbf{M}$ for pixel i, j are:

$$\mathbf{M}[0, 0] = \sum_{\substack{i-d \le x \le i+d \\ j-d \le y \le j+d}} I_x(x, y)^2 \qquad \mathbf{M}[1, 1] = \sum_{\substack{i-d \le x \le i+d \\ j-d \le y \le j+d}} I_y(x, y)^2.$$

If you look carefully, you may be able to see that you can do this by convolution – with a filter that sums things up.

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 $\text{diag}([\lambda_1, \lambda_2)$ such that (specifically an eigen-decomposition):

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

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, finding eigenvectors can be 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:

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

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.

Task 7: Harris Corner Detector (13 pts)

(a) **Implement** this optimization by completing the function `harris_detector()` in `corners.py`. (10 pts)

You cannot call a library function that has already implemented the Harris Corner Detector to solve the task. You can, however, look at where Harris corners are to get a sense of whether your implementation is doing well.

(b) Generate a Harris Corner Detector score for every point in a grayscale version of `'grace_hopper.png'`, and **plot and include in your report** these scores as a heatmap. (3 pts)

Walk-through or 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 you 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 requires of operations (det, trace) that have closed form expressions for 2x2 matrices that you can (and should!) look up. Additionally, these are expressions that you can do via element-wise operations (`+, *`) on the image representing the elements of $\mathbf{M}$ per pixel.

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 a small number of cryptic lines that look like magic but which are doing something sensible under the hood.

4 Blob Detection [24 pts]

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 from part 2 along with a scale-space representation to count the number of cells in a microscopy images.

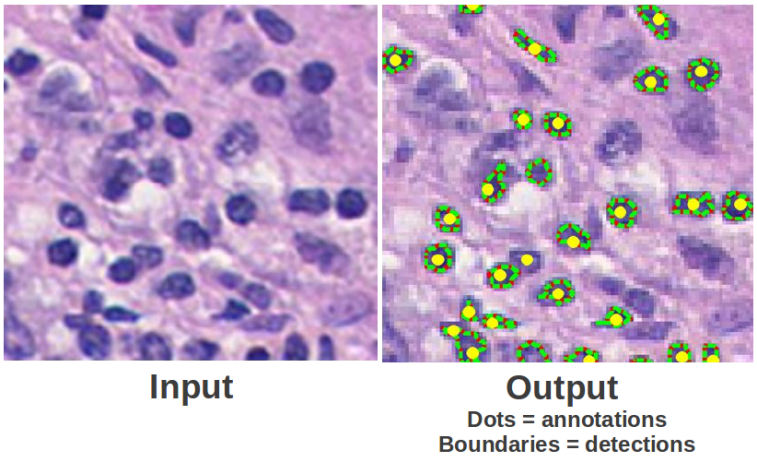

Figure 2: Detected Lymphocytes in breast cancer pathology images. Source: Oxford VGG

Note: We have provided you with several helper functions in `common.py`: `visualize_scale_space`, `visualize_maxima`, and `find_maxima`. The first two functions visualize the outputs for your scale space and detections, respectively. The last function detects maxima within some local neighborhood as defined by the function inputs. Those three functions are intended to help you inspect the results for different parameters with ease. The last two parts of this question require a degree of experimenting with different parameter choices, and visualizing the results of the parameters and understanding how they impact the detections is crucial to choosing finding good parameters. Use `scipy.ndimage.convolve()` to perform convolution whenever required. **Please use reflect padding. (Set `mode='reflect'`)**

Task 8: Single-scale Blob Detection (15 pts) Your first task is to use DoG filters to detect blobs of a single scale.

(a) **Implement the function** `gaussian_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 small circles, while the second set only responds highly the large circles. For choosing the appropriate sigma values, note that the radius and standard deviation of a Gaussian such that the Laplacian of Gaussian has maximum response are related by the following equation: $r = \sigma\sqrt{2}$. (10 pts)

(b) **Plot and include in your report** the two responses and report the parameters used to obtain each. **Comment in your report** on the responses in a few lines: how many maxima are you observing? are there false peaks that are getting high values? (5 pts)

Task 9: Cell Counting (9 pts) 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 `/cells` folder.

This assignment is deliberately meant to be open-ended. Your detections 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. You are free to use multiple scales and whatever tricks you want for counting the number of cells.

(a) **Find and include in your report** 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. **Include in your report** the number of detected cells for each of the images as well. (4 pts)

Note: You should be able to follow the steps we gave for detecting small dots and large dots for finding maxima, plotting the scale space and visualizing maxima for your selected cells.

(b) **Include in your report** the visualized blob detection for each of the images and discuss the results obtained as well as any additional steps you took to improve the cell detection and counting. **Include those images in your zip file under `cell_detections` as well.** (5 pts)

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.

Canvas Submission Checklist

In the `zip` file you submit to Canvas, the directory named after your uniqname should include the following files:

- `filters.py`
- `corners.py`
- `blob_detection.py`
- `common.py`
- `image_patches`: directory with the detected images patches in it.
- `gaussian_filter`: directory with filtered image and edge responses.
- `sobel_operator`: directory with sobel filtered outputs.
- `log_filter`: directory with LoG response outputs.
- `feature_detection`: directory with Harris and Corner detections.
- `polka_detections`: directory with polka detection outputs.
- `cell_detections`: directory with cell detection outputs.

The rest should be all included in your `pdf` report submitted to Gradescope.

References

C. Harris and M. Stephens, "A Combined Corner and Edge Detector," in Proceedings of the Alvey Vision Conference 1988, Manchester, 1988, pp. 23.1–23.6.

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