Due: Nov. 20, 2019 **Homework 5** CPSC 475/BENG 475

Edge Detection in 1-D

1 a* Compute a sigmoid function $\sigma(x) = \frac{1}{1+e^{-2x}}$ over the domain `x=linspace(-5,5,n)`, where `n` should be an **odd** integer big enough to make your sigmoid look smooth. Then take the numerical derivative of it (use the `gradient` function to get an output that is the same size as the input). Now `plot` both functions against `x` together in the same figure, with the derivative values plotted using a second (right-hand side) y-axis[1]. Where is the value of the derivative highest? Where is it low? Thinking of the sigmoid curve as an approximation to an edge in 1-D, what kind of useful information can its derivative give us?

b* From the derivative curve you computed above, localize its maximum using the following procedure: for each value (except for the first and the last), if it is greater than or equal to its left neighbor *and* greater than its right neighbor, output a `1`; else, output a `0`. Plot both the output and the original derivative arrays together (as in a, plot both against `x` and use the left and right y-axes).

Important: notice that the number of output values is 2 units less than the size of the derivative array, so, depending on your implementation, you might have to be pad it with a zero on each side in order to have the same size as before. In MATLAB all it takes is `[0 myarray 0]`; in Python, you can use the handy `np.pad` function.

c* Verify that this same rule can be implemented in a (perhaps) more "clever" way like so:

MATLAB:

```
xs = 2:n-1;
xpeaks = dx(1,xs-1) <= dx(1,xs) & dx(1,xs) > dx(1,xs+1);
```

Python:

```
xs = np.arange(1,n-1)
xpeaks = (dx[xs-1] <= dx[xs]) & (dx[xs] > dx[xs+1])
```

where `dx` is the row vector (in Python, a 1-dim. array) containing your derivative values and `n` is the same as in a. (All you need to do for this part is compute `xpeaks`, then plot it and make sure it looks identical to what you obtained in b).

d* Now you will use a different strategy for localizing the 1-dimensional "edge" in the original curve. Compute its second derivative by calling `gradient` again on the output of part a. Plot it on top of the first derivative (as in a, plot both against `x` and use the left and right y-axes). What is special about the

[1]MATLAB: run `yyaxis right` before the second `plot` command; Python: plot the first curve using `ax.plot`, then grab a second y-axis with `ax2 = ax.twinx()` and plot the second curve using `ax2.plot`.

point where the second derivative crosses the zero value?

e* Devise a procedure to generate a vector similar to the one you obtained in b and c, this time using the second derivative result from d. Explain your reasoning. Remember that the output should be a binary array with a single entry having value `1`. Plot it on top of the second derivative and comment on the result.

Edge Detection in 2-D

2 a* We would like to adopt a similar strategy in 2-D now, which hopefully will allow us to detect edges in real images. First, instead of the derivative we will need to find partial derivatives in the x and y directions. The gradient gives us exactly that. Recall the Sobel operator from section 16.3.4 in the lecture notes and implement it as two 3-by-3 matrices (make sure to normalize them by dividing each by 8). Then load Paolina (remember to convert intensities to the range `[0.,1.]`) and compute its two "partial derivatives" by convolving it against each operator separately[2]. Plot the resulting images side by side in `subplots` with `imagesc/plt.imshow` using a grayscale map—what information do they convey? How do these relate to the result from part 1-a?

b* Now combine the results of both operators by creating a new image containing the magnitudes of the gradient (using the same magnitude formula you used way back in Homework 1!). To "clean up" the result, apply a threshold of 2 times the average pixel intensity of the magnitude image (i.e., all pixels greater than twice the average pixel intensity of this magnitude image should be `1`s, the other pixels should be `0`s). Plot the result with `imagesc/plt.imshow`. *Hint:* this can be done with two lines of code only—remember to use elementwise and logical operations!

c* Adapt the code from 1-c to handle 2-D arrays in order to perform the edge-thinning step. To do this, fill in the missing line of code:

MATLAB:
```
[m n] = size(im); xs = 2:n-1; ys = 2:m-1;
xpeaks = im(ys,xs-1) <= im(ys,xs) & im(ys,xs) > im(ys,xs+1);
ypeaks = %YOUR CODE HERE
im(ys,xs) = im(ys,xs) & ( xpeaks | ypeaks );
```
Python:
```
m, n = im.shape; xs = np.arange(1,n-1); ys = np.arange(1,m-1);
xpeaks = (im[ys][:,xs-1] <= im[ys][:,xs]) & (im[ys][:,xs] > im[ys][:,xs+1])
```

[2] MATLAB: use `conv2(im,op,'same')`; Python: use `scipy.signal.convolve2d(im,op,'same')`.

```
    ypeaks = #YOUR CODE HERE
    im[ys][:,xs] = im[ys][:,xs].astype('bool') & ( xpeaks | ypeaks )
```

Now apply it to the final magnitude image you obtained in 2-b (i.e., have `im` in the code above be your cleaned-up magnitude image). What do you see? What is the purpose of this post-processing? How does it compare to what you did in part 1-b,c? Why do we need this to get an edge map?

d* Now repeat this entire procedure but this time on a noisier version of the original image. Add Gaussian noise to Paolina by adding `randn(h,w)*0.05` to the **original** image array, where `h` and `w` are its height and width (in Python, use `np.random.randn`). Visualize the noisy image. Then repeat the edge detection (all steps, 2-a-c). Adjust the threshold value to try to get as close as you can to your result in 2-c.

e* Now blur the noisy image using a Gaussian blur, and repeat edge detection. Experiment a little bit until you find a good combination of threshold and σ parameter for the Gaussian blur. Does the pre-processing with blur improve or worsen the result?

Note: If you don't want to use your blur code from Homework 3 (e.g., it had bugs), you can use, in MATLAB, `imgaussfilt(image,sigma)`, or, in Python, `scipy.ndimage.gaussian_filter(image, sigma)`.

f* We saw above that edges could also be found using second derivative information. The analogous operator to the second derivative in 2-D is the Laplacian. Based on your results from 1-e, how do you expect this operator to be used for finding edges in images? What do you think would be the role of composing a Gaussian with the Laplacian in this filter?
Optional (MATLAB only): use `edge(im,'log')` to test the Laplacian of the Gaussian edge detector. Are the results better or worse than what you got using the Sobel detector?

Oriented Gabor Filters

3 a* Flesh out the code in `gaborfilter.m` to create a function that builds an oriented Gabor filter (see section below for details, and the discussion in Chapter 17 in the lecture notes). Use this function to build filters that enhance edges, as well as filters that enhance bright and dark lines. Edge-enhancing filters will have phase offset 0; bright line enhancers will have phase offset $\pi/2$; dark line enhancers will have phase offset $3\pi/2$.

The edge filters should have 8 orientations, equally spaced in the range $[0, \frac{7\pi}{4}]$ (in radians). The line filters should each have 4 orientations equally spaced in the range $[0, \frac{3\pi}{4}]$. You should only have 4 bright line filters and 4 dark line filters, since line-enhancing filters have even symmetry (so rotating them by $180°$ doesn't change them), while edge enhancers have odd symmetry. This results in 16 different filters: 8 edge filters, 4 bright line filters, and 4 dark line filters.

Your write-up should consist of images of all 16 filters, as well as `Paolina.tiff` convolved with each filter (please organize them neatly using subplots). Alternatively, you can get the same results (same 16 filtered Paolina images) using half as many filters (and performing half as many `filter2` calls). If you say how, you only need to include the 8 filters you use.

Important: Use `imagesc/plt.imshow` to show the filters, but, for the filtered images, use **`imshow`** in MATLAB and **`plt.imshow(...,vmin=0,vmax=1)`** in Python.

Be sure to include your code in the submission.

Creating a Gabor Filter

Assuming your coordinates x and y are centered at 0, define x' and y' as follows:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}.$$

In creating a matrix containing the filter values for a Gabor with orientation θ, the value at (row, col) position (i, j) is then given by

$$G_{ij} = G(x_{ij}, y_{ij}) = \sin\left(\frac{2\pi}{\omega} y'_{ij} + \phi\right) \exp\left(-\frac{x'^2_{ij}/\alpha^2 + y'^2_{ij}}{2\sigma^2}\right)$$

where ω is the wavelength of the Gabor filter in pixels (distance between peaks of the sinusoid), and ϕ is the phase. You should choose α (the aspect ratio) greater than 1 so that the filter is somewhat elongated in the x' direction (using $\alpha = 1$ will give you a circular Gaussian envelope). Choose ω and σ (the bandwidth parameter of the Gaussian envelope) in a coordinated way based on the size of the edge or line features you want to emphasize (say what you pick and why)—a good rule of thumb is to have ω be approximately 4σ. You should be able to clearly see at least one full period of the sinusoidal component of the Gabor when you display the filter, and not more than two (though it's fun to experiment with what happens when you filter an image as you change the relationship between σ and ω).

Normalize the filter so that the sum of its entries is 0, and the sum of the squares of its entries is 1.

MATLAB hint: Remember to use elementwise operations (i.e., matrix multiplication between a sine and a Gaussian doesn't make a lot of sense; squaring a matrix must also be done elementwise here). Plus, they will greatly simplify your code—you don't need any for-loops, actually. If you have questions about this, ask one of us.

- b* Compare the effects of an edge enhancer, a bright line enhancer, and a dark line enhancer for a given orientation. Feel free to load any other images of your choice. (Finding nice examples where the difference in the results of these three filters becomes evident can be worth up to 10 points extra credit!)

- c* How do Gabor filters relate to receptive fields in primary visual cortex? Compare having a single edge map (as in part 2) with having separate edge/line maps for each orientation. What do you think are the computational/perceptual advantages of the latter representation?

- d* In the space domain, a Gabor filter is the pointwise product of a Gaussian function and a sinusoid. Recall that a Gaussian in the space domain transforms to another Gaussian in the frequency domain (having a bandwidth parameter inversely related to the the bandwidth parameter of the Gaussian in the space domain), while a sinusoid in the space domain transforms to a (pair of) delta functions in the frequency domain. Describe the frequency domain representation of a Gabor filter in terms of the frequency domain representations of its Gaussian and sinusoidal components. Display its 2-D Fourier transform to confirm your results (remember to use `abs` and `fftshift`!).