Homework 8 **PSY4219/6219**
Due November 9 in class **Fall 2022**
18 points

You can do this assignment in a Python .py file or in a Jupyter Notebook. Recall that if you run into problems with code in a Jupyter Notebook, the best thing would be to try to debug that code in PyCharm (even if you want to copy it back to a Jupyter Notebook).

Q1. For this question, perform some basic operations on an image that I have loaded onto Brightspace (`nashville.jpg`).

(a) (2 points) Read in `nashville.jpg`, display it, and convert it from RGB to grayscale using the following formula to convert from red, green, and blue values to grayscale intensity:

intensity = 0.2989*red + 0.5870*green + 0.1140*blue

Save the image as a `jpg` file called `nashvillegray.jpg`, read it back in, and display it. You will use this grayscale version in the remaining parts below.

(b) (5 points) Create a scrambled version of the grayscale image `nashvillegray.jpg`. Imagine dividing the image into an 8x8 grid. Create a scrambled version of the image where each of the sections in the 8x8 grid is randomly shuffled (a permutation). An example of the grayscale version and one possible scrambled version is shown below. We discussed a way to accomplish this scrambling in class.

Your code should read in the image into a numpy array. Create a function that takes this numpy array as an argument and takes the number of ways to slice the image (for example 8 to create an 8x8 grid) and returns a numpy array containing the scrambled image. Your code should display the scrambled image. Aside from using a built in permutation function and other non-image-processing functions in Python, the scrambling of the image should be done by your code (in other words, do not search for any built-in image/signal

processing functions or existing packages that might do this scrambling for). Please try to write your code so that you can change the "8" in the 8x8 grid to some other value, like a 4x4 grid, or a 64x64 grid; you can assume that the number is chosen to equally divide the image size. Save the resulting scrambled image to a jpg file.

(c) (2 points) Add "noise" to an image (in this case `nashvillegray.jpg`). Your code should read in the image into a numpy array, that numpy array should be passed to a function that adds the noise, with the noise specified by other arguments passed to the function.

The noise should be normally distributed with mean `0` and standard deviation `sigma`. This "noise" is added to the intensity (brightness) of each pixel in the image. This kind of noise is what sometimes called "salt and pepper noise", whereby the intensity of each image pixel is jiggled up or down by a normally distributed random number.

When you add noise, allow for two options (as an optional argument in your function): one version where added noise that would cause a pixel value to go above 255 is capped at 255 go below 0 is capped at 0, and another version where noise is allowed to jump from 255 to 0 or from 0 to 255 when you cast as an `uint8`. Your code should show what happens (displaying two images) to illustrate the two ways of manipulating noise (and capping vs. rolling over at the extremes).

Your function should return a numpy array as `uint8`.

For each version, create one image and save it with a low but perceptible level of noise and create another image and save it with a relatively high level of noise; you will be saving four images.

Q2. We talked about using the Laplacian of Gaussians filter to find edges in images. For this question, I would like you to explore the Laplacian, the Gaussian, and the Laplacian of Gaussians (LoG). This question uses the convolution computations we talked about in class and illustrated in the posted Jupyter Notebooks from class.

For this assignment, read in and use `nashvillegray.jpg` from above, but your code should work for any image that is read in.

(Continued on next page)

Use this Laplacian filter:

```
L = np.array([[0,  0,  0,  0,  0,  0,  0,  0,  0],
              [0,  0,  0,  0,  0,  0,  0,  0,  0],
              [0,  0, -1, -1, -1, -1, -1,  0,  0],
              [0,  0, -1, -1, -1, -1, -1,  0,  0],
              [0,  0, -1, -1, 24, -1, -1,  0,  0],
              [0,  0, -1, -1, -1, -1, -1,  0,  0],
              [0,  0, -1, -1, -1, -1, -1,  0,  0],
              [0,  0,  0,  0,  0,  0,  0,  0,  0],
              [0,  0,  0,  0,  0,  0,  0,  0,  0]])
```

Use this Gaussian filter:

```
G = np.array([[0,  0,  0,  0,  0,  0,  0,  0,  0],
              [0,  0,  0,  0,  0,  0,  0,  0,  0],
              [0,  0,  1,  4,  7,  4,  1,  0,  0],
              [0,  0,  4, 16, 26, 16,  4,  0,  0],
              [0,  0,  7, 26, 41, 26,  7,  0,  0],
              [0,  0,  4, 16, 26, 16,  4,  0,  0],
              [0,  0,  1,  4,  7,  4,  1,  0,  0],
              [0,  0,  0,  0,  0,  0,  0,  0,  0],
              [0,  0,  0,  0,  0,  0,  0,  0,  0]])/273
```

Note that this Gaussian filter is larger than some of the examples shown in class (and in the Jupyter notebook) because here we are using a higher-resolution image. Using a 3x3 Gaussian would produce nearly-imperceptible blur.

(a) (2 points) Confirm (in code) that convolution is commutative, using the Laplacian and Gaussian filters, within a reasonable small epsilon < .000001. Use the epsilon value in some way to support your demonstration (in other words, do not simply print out the resulting convolutions to inspect by eye – show it somehow).

(b) (3 points) Confirm that you understand the computations that underlie convolution. Pick a single pixel location (*i,j*) somewhere in the grayscale image of the Nashville skyline (`nashvillegray.jpg`) that is sufficiently far away from the edges of the image to avoid dealing with edge effects. First, calculate the convolution of the Laplacian with the image at this pixel location without using the `signal.convolve2d()` function, i.e., just following the equations for computing the convolution at a particular pixel location (*i,j*). Confirm that the value you get with your "hand-coded" convolution calculation matches that using the built-in `signal.convolve2d` function at the same pixel location (*i,j*) within some epsilon.

Note that your code will need to work with other filters (not just the Laplacian) so make sure it's general (not hard-coded). Recall that the convolution requires a particular way of doing its calculations that makes it different from a cross-correlation; don't just rely on

getting the same answer for your code and the `signal.convolve2d` function; make sure your code does the correct calculations.

(c) (1 point) Create a new filter, called LoG, which is the convolution of the Laplacian with the Gaussian. I do not want you to use the LoG filter I defined in class, but create one by actually convolving the Laplacian with the Gaussian (above) in code. You can use `signal.convolve2d()` function for this.

(d) (3 points) Convolve the image with the Laplacian alone, the Gaussian alone, and the Laplacian of Gaussians. You can use the `signal.convolve2d()` function for this.

Make a 1x4 subplot. Display the original and the results of the convolutions using `imshow()`.

Recall that the Laplacian and Laplacian of Gaussians (LoG) produces an "image" of second derivatives. That means that it will contain both positive and negative values (you can confirm that by looking at the numpy arrays resulting from the convolution). You will want to play around with ways to visualize the Laplacian and LoG. If you just display it directly, it might be hard to see the results of the filtering. For example, you could set a criterion (a step function) so that values greater than the criterion are white and those less than the criterion are black (recall that the Laplacian detects "edges" in an image). Make sure you manipulate the Laplacian and Laplacian of Gaussian images the same way.

Unexcused late assignments will be penalized 10% for every 24 hours late, starting from the time class ends, for a maximum of two days, after which they will earn a 0.