Assignment 03

Please refer to MyCourses/Gradescope for assignment deadlines.

The goal of this assignment is to give you hands-on practice with the following concepts:

- correlation and convolution
- separable filters
- blurring and sharpening
- template matching

Part 1: DIY correlation function and separable filters

Correlation can be done with the OpenCV function `cv.filter2D`. Convolution can be done by first flipping the kernel then using `filter2D`, since convolution is just correlation with the sign of the horizontal (l)and vertical (k) indices flipped. Here, you're going to implement your own version of correlation in numpy to see how it works under the hood and to better appreciate the "separable filters" property.

Write the body of the `my_correlation` function in `correlation.py`. The `my_correlation` function should make use of the `numpy` module but may not call any `cv` functions. Note that you will have to implement your own padding in order to match the behavior or `cv.BORDER_REPLICATE`. (Hint: you don't actually need to create a new padded image, you just need to re-use pixels from the border any time the index would take you outside the `[0...h-1, 0...w-1]` range). Be careful not to modify the original image in this function!

Once you have it working, see how fast you can make your correlation function. There will be a leaderboard for the fastest code on Gradescope. **Incorrect** submissions will not be considered for the leaderboard. First, get the functionality correct. Then, make it fast. Note that the leaderboard will test your code with random *separable* filters. Since separable filters are much faster to apply (linear in the width and height of the filter rather than quadratic), you should be able to get a much faster time by

- detecting whether the filter is separable
- if so, splitting it into its u and v components, then applying the filter in two passes rather than one
- if not, applying the filter in one pass as before

Exactly how to tell whether a given 2D filter is separable is handled by an interesting bit of linear algebra – equation (3.21) in the CVAA book. I consider this hard and out of scope, so it's been given to you as a helper function in `utils.py`. Feel free to import and use those functions.

Winners on the leaderboard will be awarded extra participation points, which counts as extra credit towards your final grade.

- 1st place: +5 EC
- 2nd place: +4 EC
- 3rd place: +3 EC
- 4th place: +2 EC
- All others who do better than the median: +1 EC

Part 2: Blurring and sharpening

As we saw in class, blurring and sharpening are just two sides of the same coin. If f is an image and h is a blur kernel (a 7x7 Gaussian kernel, say), then $g_{blur} = f * h$ is the blurred image. We can define

$$v = (g_{blur} - f) = f * (h - \delta)$$

as the *difference* between the original and the blurred image (where δ is the delta kernel). Think of v as a vector whose base is the original image and whose tip is the blurred image. We can of course add this vector to the original image to get the blurred image, as in

$$g_{blur} = f + v.$$

Going one step further, we can introduce a scale a and define

$$g_a = f + av.$$

Note that when $a = 0$, we have $g_0 = f$, and when $a = 1$, we have $g_1 = g_{blur}$. So, a controls the "amount of blur", in a way. (This is different from making the blur kernel bigger – normally "adding more blur" is a matter of increasing the size of the kernel.)

Sharpening is what you get when you set a to a *negative* value. For example, we can define

$$g_{sharp} = f - v$$

as a fully sharpened image. We're just taking a step in the opposite direction from the blurred image. This is equivalent to setting $a = -1$ in the above equation, so $g_{-1} = g_{sharp}$.

Thus, blurring and sharpening are just two sides of the same coin. They're different values for a. In this part of the assignment, you'll define a single operation that does both, depending on the value of a. You'll do this by constructing a single kernel h_a such that $g_a = f * h_a$. Your only task is to implement the `create_separated_blur_or_sharpen_kernel` function in `blur_and_sharpen.py`.

If you run that file on a machine that has a webcam, you'll see an interface that lets you adjust the value of a and see the effect on the image. You can also use the `--image` command line argument to specify an image file to use instead of the webcam.

Note that the output of `create_separated_blur_or_sharpen_kernel` should be a 1D numpy array u which contains the 'separated' kernel, i.e. the full kernel h will be implicitly defined as uu^T, but we will never explicitly instantiate h.

Some example outputs (values of a are in the title of each image):

Part 3: Template Matching using OpenCV

Template Matching uses the correlation operator to "search" for a small template image within a larger scene. We'll use this to build our first object detector in `template_match.py`. As in previous assignments, you're given some example images to play around with, but ultimately your code will be evaluated in terms of how well it does on held-out or random test images. The example images you're given are `mario.jpg` - a scene from a Super Mario game, and `coin.png` - a small template coin. We're going to help mario find all the coins in the scene.

Figure 1: Here's an example output using `visualize_matches` with the `mario.jpg` and `coin.png` images. A small red box has been drawn around each detected object, and a total count has been displayed in the bottom left corner.

If you look at the files carefully, you'll notice that the `coin.png` image has exactly the correct size and orientation to match the coins in the `mario.jpg` image. This is not a coincidence. Template matching is not by itself robust to scale or rotation changes. We really need the image to "match" the template for it to count as a match. Keep this in mind if you'd like to try your code out on other example scenes and objects.

While you could use your `my_correlation` function to do template matching, OpenCV has a built-in function called `cv.matchTemplate` that will do the correlation (and then some) for you. The main steps of your code will thus be:

1. Load the scene and the template (done for you)

2. Use `cv.matchTemplate` to get a score for each pixel in the scene. You'll need to read the OpenCV documentation for `cv.matchTemplate` to decide the best `method` argument.
3. Normalize the scores so that they are all between `0` and `1` and so that the best match has a score of `1`. NOTE: if you use the `cv.TM_SQDIFF` or `cv.TM_SQDIFF_NORMED` methods, the scores are "inverted" so that the best match has a score of `0`. You'd thus need to account for this inversion in the normalization step.
4. Apply **non-maximal suppression** to the scores. This means that you should only keep the best match in a small neighborhood around each pixel. This is to avoid detecting the same object multiple times.
5. Apply a threshold to the scores to decide which pixels are "good enough" matches
6. Use `np.where` to get all `(x,y)` coordinates where there was a match
7. Draw and display the results (done for you)

Steps 2-6 are the main part of the assignment where there is "missing code" for you to fill in. The key concept is that a "match" is where **both** of the following conditions hold:

- The "score" we get from normalizing the output of `matchTemplate` is above a certain threshold
- The "score" is a local maximum in a neighborhood around the pixel, so we don't count both a good match and its neighboring almost-good-matches as separate detections.

Collaboration and Generative AI disclosure

Did you collaborate with anyone? Did you use any Generative AI tools? Briefly explain what you did in the `collaboration-disclosure.txt` file.

Reflection on learning objectives

This is optional to disclose, but it helps us improve the course if you can give feedback.

- what did you take away from this assignment?
- what did you spend the most time on?
- about how much time did you spend total?
- what was easier or harder than expected?
- how could class time have better prepared you for this assignment?

Give your reflections in the `reflection.txt` file if you choose to give us feedback.

Submitting

As described above, use `make submission.zip` to generate a zip file that you can upload to Gradescope. You can upload as many times as you like before the deadline. To account for late penalties, contact the instructor or grader directly if you plan on uploading any further revisions after the deadline.