Image Smoothing Results

Image smoothing is a technique used to remove noise from an image. This highlights region of interest within an image by removing unwanted, small speckles. The major drawback of image smoothing is the blurring effect that occurs when the mask is applied. For this project, we wanted to observe how the window size of the mask affects the blurring of an image. To do this, we created our own convolution function and tested the following window sizes: 3x3, 5x5, 7x7, 9x9. After multiple test runs, it was apparent what the effects were. The noise was significantly reduced the larger the window sizes became; however, the image also became significantly blurrier. These results are displayed in the figure below.

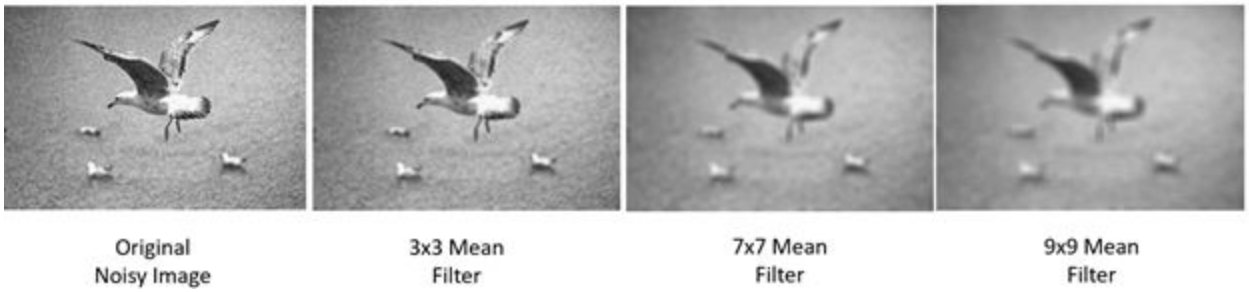


Image Smoothing with Different Window Sizes

As you can see, the image changes depending on the window size. As the window size increased, the images "smoothened" out and became blurrier.

Convolution Comparison

For this project we created our own convolution function needed to create the blurred images. We tested the speed and accuracy of our function with python's own numpy convolution function. To do this, we used python's time module to place timer's before and after the convolution function and took the difference between the two to test the speed of the function. The same image was used and tested over the 3x3, 5x5, and 9x9 window sizes. Our results are posted below.

	3x3	**5x5**	**9x9**
Numpy Convolve Function (secs)	.016	.032	.088
Our Convolve Function (secs)	2.16	2.21	2.47

The numpy function proved to be far superior compared to our own. Averaging all the runs, our function took about 2.28 seconds but the average time for numpy's function was .045 seconds! Although the same images were produced for all window sizes, numpy's convolution function was significantly more efficient.

GUI

When considering a method for implementing the UI, we considered using Python's tkinter library or a web framework such as Flask or Django. In the end, I decided on creating an extremely simplified, bare-bones framework for handling web requests. I was able to find some boilerplate code that I was able to modify to fit our needs.

The boilerplate code is an MVC framework using the front-loader pattern. The front-loader pattern uses a single file (server.py) to direct HTTP requests on localhost:8080 to different Controller methods based on the URI path. The Controller would then return the data and render the appropriate view.

One difficulty I had was on how to create a RESTful API on Python while still using the same server instance to serve the view to the browser. Running "python server.py" would provide the server instance, but it no longer became simple to make web requests in the way that I was familiar with. Alternatively, running "python http.server 3000" would initialize a more familiar web server instance, but would no longer provide access to the router method I had created in wsgi.py.

The compromise I was able to come up with was to run two server instances simultaneously. Two commands would have to be used: "python server.py" will serve the RESTful API on localhost that will listen on port 8080 and "python http.server 3000" will serve the view component on localhost that will listen on port 3000.

The final solution was to use port 3000 to render the form for the user to interact with. Upon making a submission, using JavaScript, an AJAX call is made to the appropriate route based on the selected operation type to localhost:8080 along with passing through all of the parameter options that were selected. Once the request is made, the Router that is listening on port 8080 will parse the URL path, direct the request to the appropriate controller and execute the necessary class method.

For instance, near the top of the wsgi.py file, you'll see a list of routes being set.

- router.get('/smoothing', 'SmoothingController@smoothing')

This indicates that any GET requests on the /smoothing path will instantiate the SmoothingController class and call the smoothing() method. Once the method finishes executing, it will write the image, and render the data to a file. Once the data has been written to the file, the above AJAX will read the data and update the view by displaying the images that were newly made.

Correlation And Convolution

Correlation and convolution are two fundamental operations in spatial filtering. In the project, we combine two operations in one class since they have some similarity.

There are three steps in these two operations. The first step is zero padding, the correlation or convolution operations will do an operation to each pixel in the original image. So first we need to add zeros to column and row of the image, the number of zeros based on the shape of mask's size. After zero padding, you can either do a correlation or convolution. The only difference between the two operations is in convolution you first need to rotate 180 degrees of the mask. Then cover the mask to the image, do the summation of the product of each pixel corresponds. The third step is a reverse operation of zero padding, we need to crop all the zeros when added in the first step to keep the image has the same shape as the original one.

There are several points we should pay attention to during the whole process. One is the size of zeros be added to the original image because the size of the mask is varied, in the project we accept different mask size as long as the columns or rows are odd number. Secondly, in order to do the product of each pixel of the original image with mask corresponds, we first split sub-image from the original image which has the same shape as the mask, then we do a dot product of this sub-image with the mask. Thirdly, since we didn't use any built-in function to do these three steps, it will take 1-2 minutes to generate the result.

Laplacian

Laplacian is one of the operation in the spatial filter to sharpen the image. The key point for the laplacian is what kind of kernel do you choose. In the project, you can define your own kernel or you can use the default kernel we have. After one decides the mask, do a convolution operation. Then add/subtract this mask to the original image to sharpen it. In the laplacian operation, we reuse the convolution function from the project we build. One difficulty here is that do addition or subtraction really depends on your kernel. If you choose a kernel with 4 in the center, then it going to be an addition in the end. If you choose a kernel with -4 in the center, it going to be a subtraction in the end.

Statistical- order filters

The implement of statistic part can be divided into three main parts: noise adding, statistic order operation and calculation of SSIM (a parameter to assess image quality used here to evaluate the function of filters). The core part of statistical order filter and SSIM calculation is to write the statistic operations like: mean, median (sort first), variance and covariance. I wrote them first, they are working correctly but too slow. So, I chose to use built-in function for the statistic operations for faster demo. A time costing analysis will be introduced later.

Result analysis

Result analysis is based on SSIM ,in general, the closer the SSIM number to 1, the better the filtering (denoising) is done . In general, I found median filter is better fit for salt-and-pepper noise, mean filter is better fit for gaussian noise. Also, window size matters a lot, the bigger window size, the better filtering result.

Time costing analysis

Use median filter to get rid of salt-and-pepper noise as an example, table 3 shows result for analysis

	Image size 256^2, filter size 3^2	Image size 256^2, filter size 5^2	Image size 512^2, filter size 3^2	Image size 512^2, filter size 5^2
Using built-in	2.0878(0.9428)	2.4876(0.9869)	8.4676(0.9496)	10.2557(0.9927)
Using own	1.4220(0.8357)	6.1256(0.9891)	5.9065(0.8411)	24.3374(0.9946)

*t(ssim),example：2.0878（0.9428）means cost 2.0878 seconds with an result ssim 0.9428

To summarize the time costing analysis, built-in function is faster at bigger filter size, slower but better with smaller window size.

Unsharp Masking and High Boosting

Unsharp Making is a popular image sharpening technique used in Printing and Publishing Industry. This technique uses a blurred, or “unsharp”, negative image to create a mask of the original image. The unsharped mask is then combined with the original image, creating a less blurry image than the original. The resulting image, although clearer, may be a less accurate representation of the image’s subject. As you can see from the output, as the smoothing derivative increases noise decreases, although blurs the edges and at different scales.

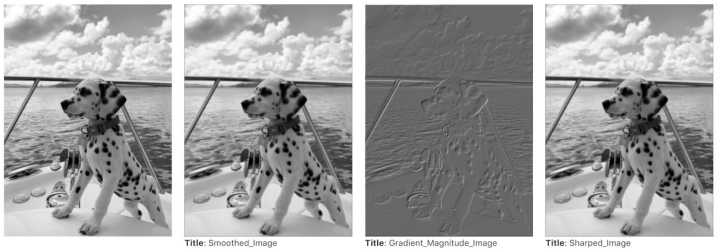
Highboosing is a technique which enhances the details in any picture. From the output we can see that as high boosting increases, the details in the image becomes more visible to the naked eye, although increasing more that required distorts the image details. Two filters are used for smoothening, average and gaussian. From the screenshots below we can say than gaussian kernel produces better results than average filter in any respective kernel size.

Average 5x5 filter with 1.2 High Boosting

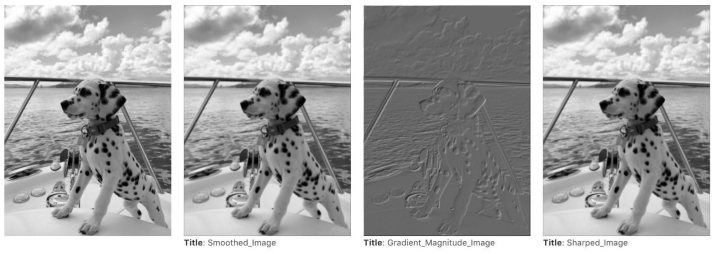
Gaussian 5x5 filter with 1.2 High Boosting

Image Sharpening using First Order Derivatives

Image sharpening is a process of highlighting fine detail in an image or to enhance detail that has been blurred. Results from a first order derivatives produce thicker edges and has stronger response to grayscale images, as the technique finds the intensity by the difference from its neighbors. Two filter operators Sobel and Prewitt are used as they are quick and simple. These derivatives are strongly affected by noise, as noisy image pixels looks very different from their neighbors. The larger the noise in the image, the lesser the likelihood of using this technique.



Sharpening using Sobel Operator



Sharpening using Prewitt Operator