Clab-1 Report

ENGN4528

Thao Pham

u7205329

28/03/2021

1 Task-1: Python Warm-up

```
a = np.array([[2, 3, 4],[5, 2, 200]])
```

a is the 2×3 matrix $\begin{bmatrix} 2 & 3 & 4 \\ 5 & 2 & 200 \end{bmatrix}$.

```
b = a[:,1]
```

b is the second column of the matrix above.

```
f = np.random.randn(400,1) + 3
```

f is a random 400-by-1 array drawn from a normal distribution of mean 3 and standard deviation 0.

```
g = f[f>0] * 3
```

This line can be separate into two parts. `f[f>0]` selects the elements in `f` that are greater than 0. Next, said elements will be multiplied by 3 and save to a variable g.

```
x = np.zeros(100) + 0.45
```

x is an array of length 100 with repeated elements of 0.45.

```
y = 0.5 * np.ones([1, len(x)])
```

y is an array of length 100 with repeated elements of 0.5.

```
z = x + y
```

z is an array of length 100 with repeated elements of the sum of 0.45 and 0.5.

```
a = np.linspace(1, 499, 250, dtype = int)
```

a is an array of 250 evenly spaced elements between 1 and 499 (inclusive).

```
b = a[::-2]
```

b is an array consisting of every other 2nd element of the array, above in the reversed order.

```
b[b>50] = 0
```

This line of code sets every element in b that is less greater than 50 to be 0.

2 Task-2: Basic Coding Practice

2.1 Negative Image

Figure 1 (b) shows the result of the negative image of `Atowergray.jpg`. The result was obtained by subtracting each intensity level in Figure 1(a) from 255. Thus the dark regions in the original image (e.g. the tower) were shown as bright in the inverted image, and vice versa (e.g. the sky).

$$I'[x, y] = 255 - I[x, y]$$

2.2 Flip the image horizontally

Figure 1 (c) shows the horizontally flipped version of Figure 1 (a). The cloudy region that was originally on the left side of the image can now be seen on the right side. To obtain this result, we performed a mapping function where the y-coordinates remain the same and the x-coordinate of each pixel value is swapped to that on the opposite side.

$$I'[width - x, y] = I[x, y]$$

2.3 Swap the red and green channel of a colour image

For this question, the image `image3.jpg` is used. Figure 1 (e) shows the original colour image and Figure 1 (f) is the result of the channels being swapped. The result was obtained by first accessing the channels separately and saving each to its respective variable. A canvas image was created and the swapping channels was then performed when we reconstruct the image.

```
R,G,B = cv2.split(image3)
output_image = np.zeros((height, width, channels), dtype = np.uint8)
output_image[:,:,0] = G
output_image[:,:,1] = R
output_image[:,:,2] = B
```

2.4 Average the input and horizontally flipped image

Figure 1 (d) shows the result of averaging Figure 1 (a) and (c). This was done by first typecasting the two images to double in order to perform algebraic operations on the intensity values. After averaging, we also clipped the maximum value to 255.

```
weighted = 0.5*img.astype(np.double) + 0.5*flipped.astype(np.double)
weighted[weighted > 255] = 255
```

2.5 Add noise

Figure 1 (g) shows the result of Figure 1 (a) with every of its intensity value increased by a different random integer between [0, 127]. This was obtained by performing a point operation on Figure 1 (a) where $c \in [0, 127]$. After adding this random value, we again clipped the maximum value to 255. The result is Figure 1 (a) with added noise.

$$I'[x, y] = I[x, y] + c$$

```
height,width = img.shape
random_mat = np.random.randint(0, 127, (height, width))
add = img.astype(np.double) + random_mat
add[add>255] = 255
```

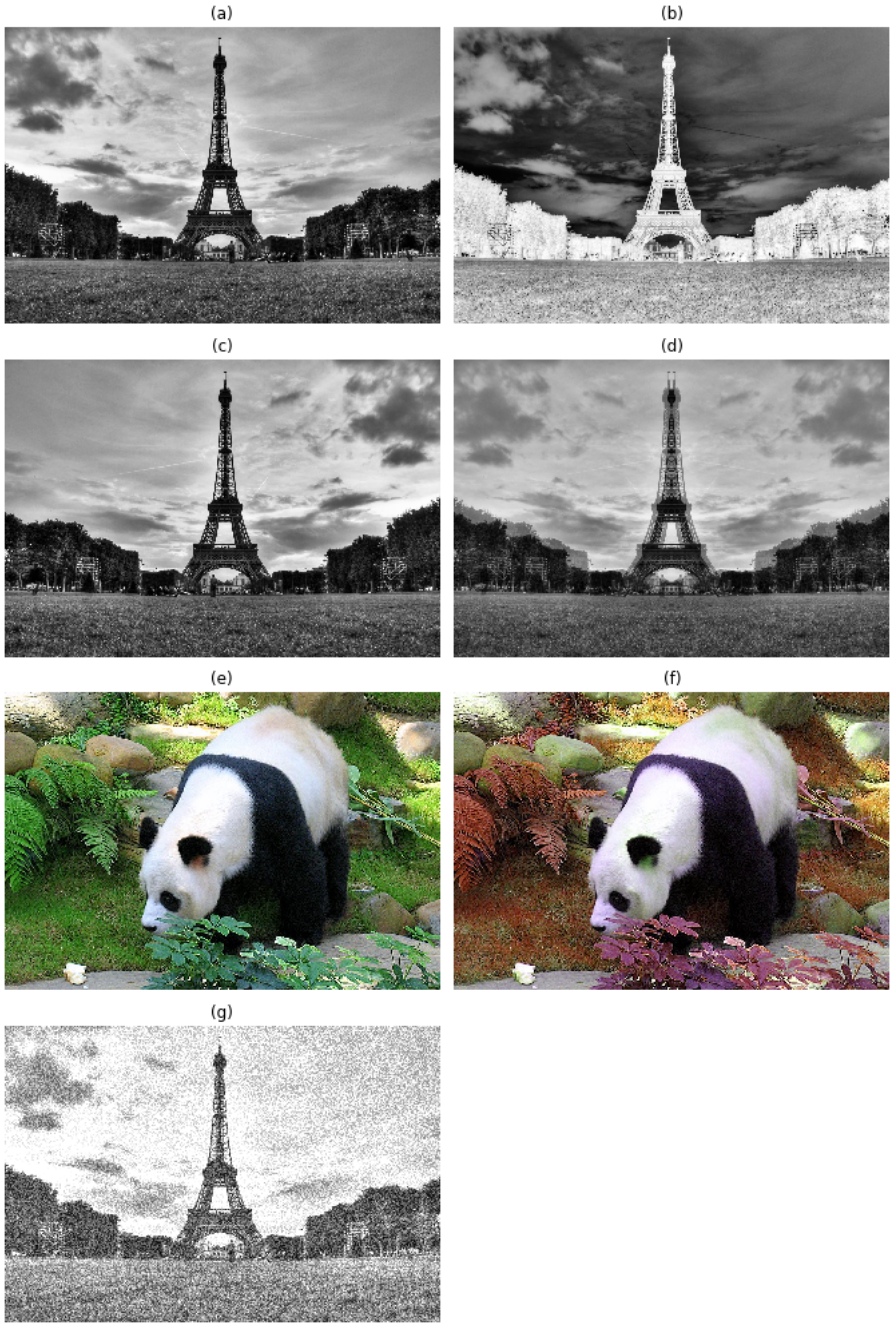

Figure 1: (a) Original Atowergray.jpg. (b) Negative Image. (c) Horizontally Flipped. (d) Averaged Image. (e) Original colour image image3.jpg. (f) image3.jpg with its red and green channel swapped. (h) Atowergray with added noise.

3 Task-3: Basic Image I/O

3.1 Colour channels

Figure 2 shows `image1.jpg` and its three colour channels displayed as grayscale. Notice that dark brown chair where the puppies are sitting on and the white puppies fur has similar intensity values across all channels, while the green trees are visibly brighter in the green channel. The same observation can be seen for the red tongues in the red channel. Since there is little blue colour in the original image, the blue channel is relatively darker than the other channels.

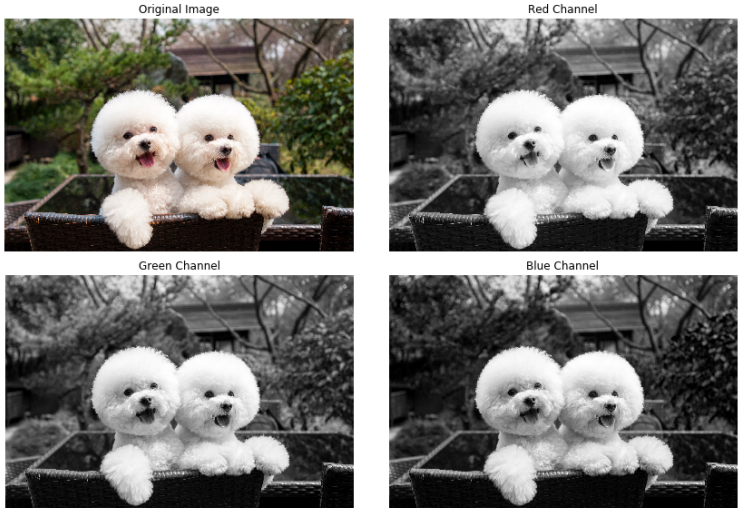

Figure 2: (a) Original image1.jpg. (b) Red channel. (c) Green channel. (d) Blue channel.

3.2 Histogram

Figure 3 shows the histograms corresponding to each of the grayscale image shown in section 3.1. The general shape of distributions is relatively similar across the three histograms. The majority of data points lay in the lower intensity levels since the image is relatively dark. Since white is a combination of all colours, the spike at around 200-250 intensity value corresponds to the white fur of the puppies is uniform across all three channels.

The high distribution at around 50-100 in the green channel can be explained by the dark green trees in the background. As there is little blue in `image1.jpg`, the blue channel histogram is heavily right-skewed. Moreover, as most of the pixels occurs on the lower end in combination with the small spike in the higher intensities result in a highly contrasted image (Figure 2 (c)).

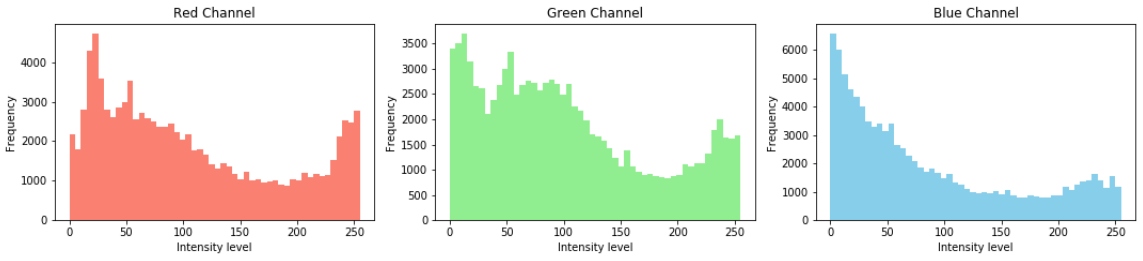

Figure 3: Histogram of the colour channels

3.3 Histogram Equalisation

Figure 4 and 5 show the results of each colour channel and the resized `image1.jpg` after performing histogram equalisation. The changes on each histogram of the colour channels are quite drastic. Compared to earlier where the distributions of the intensity were more skewed towards the lower values, the overall shapes are now more evenly distributed. The effect can be seen in Figure 5 where the background contrast has greatly improved and we can make now make more sense of the background. The lighting conditions are now similar across all channels.

For the R,G,B channels, the process was done by implementing the function `cv2.equalizeHist()` directly. The same function cannot be used for a colour image, however. A solution for this was to combine the three colour channels after histogram equalisation and reconstruct the colour image using the methods similar to that used in Task 2.3.

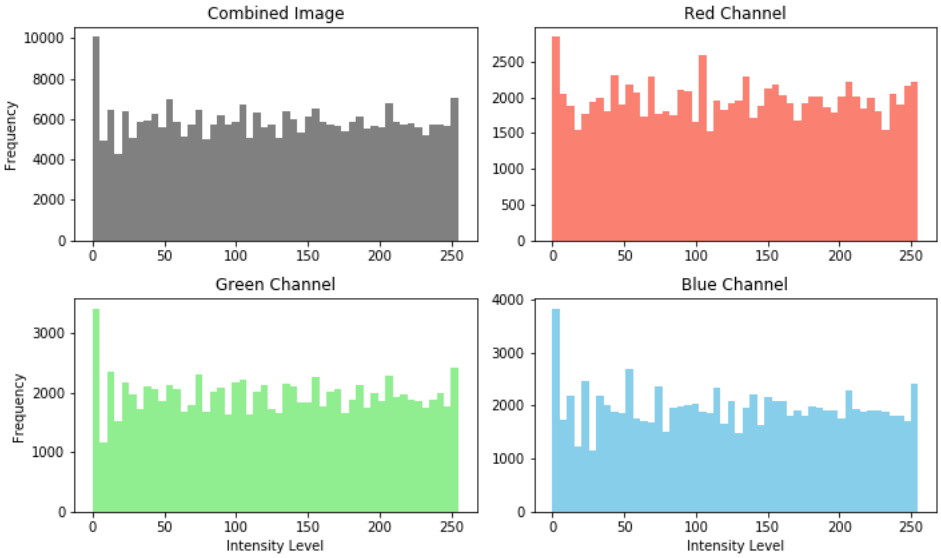

Figure 4: Histogram of the colour channels after performing histogram equalisation

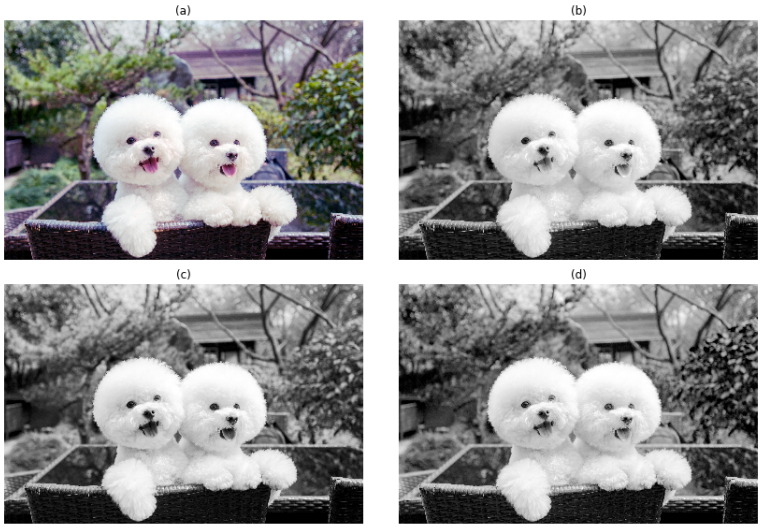

Figure 5: (a) image1.jpg after performing histogram equalisation (b) Equalised red channel (c) Equalised green channel (d) Equalised blue channel

4 Task-4: Colour space conversion

4.1 RGB to YUB Conversion

Figure 6 is the result of the `Fig2a.png` coverted into the YUV colour space, shown in grayscale. The Y channel shows the luminance/brightness of Fig2a while the chroma components U and V shows the intensity in blue and red components, respectively. The blue region in the colour wheel (that we know from the original image to be at the bottom left corner), can be seen more brighter in the U channel. Similarly, the red area (bottom right corner) is visibly brighter in the V channel. The black texts in the Y channel do not appear in the U and V channels since the chroma components only capture the colour signals.

The conversion function `cvRGB2YUV()` used the formula (2.115) from Szeliski, 2011. The YCbCr conversion was used since we are working with digital signals and eight-bit range values.

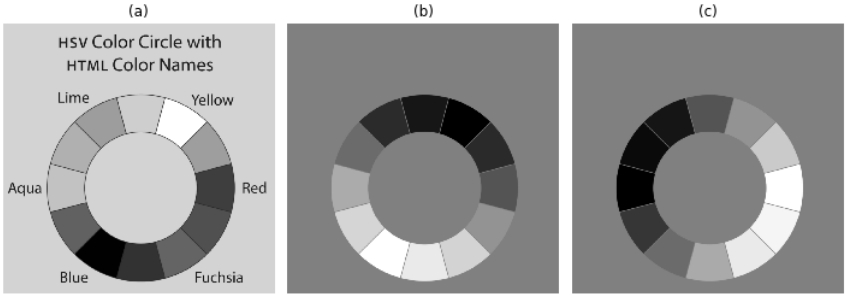

Figure 6: Y, U, and V channels of `Fig2a` displayed in grayscale,

4.2 Average Y values

Figure 7 shows `Fig2b.png` in the RGB colour space with the average Y values of each region shown in the corresponding region. Note that the values in the upper region are computed using the function `cvRGB2YUV()` from section 4.1, while the ones in the lower region are computer by `cv2.cvtColor`.

To identify each region, the image is first converted into the YUV colour space (using function of choice). As each pixel is a 3-dimensional vector, the image can then be fed into a k-nearest neighbour classification algorithm as inputs. The classifier returns the cluster centres that distinguish each region. Note that each cluster center is an array of length 3 that contains the average Y, U, and V values of the region it represents.

To display the cluster centers, we take 5 pixels evenly spaced from each other at a fixed height. Each pixel is then fed back to the classifier where the classifier will predict which cluster it belongs to and print the said cluster's representative. As can be seen in Figure 7, the average Y values computed by both functions appear to agree with each other.

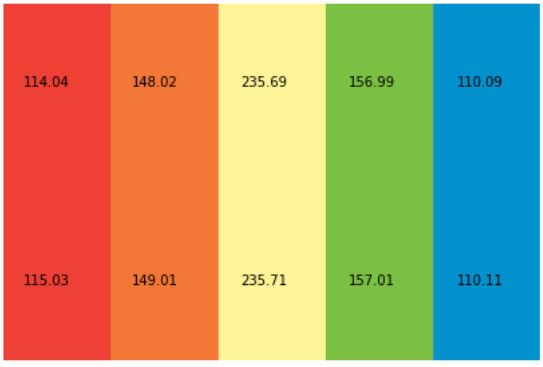

Figure 7: `Fig2b` in RGB colour space with average Y values computed by `cvRGB2YUV()` (upper) and Python's in-built function (lower)

5 Task-5: Image Denoising

5.1 Via Gaussian Filter

5.1.1 Crop, Resize, Add Gaussian noise

Figure 8 shows `image2.jpg` cropped, resized, saved as grayscale and added noise. The method for adding noise is similar to that in Task 2.5. The difference is the random matrix is now sampled from a Gaussian distribution of mean 0 and standard deviation 15. The effect of adding Gaussian noise can be observed in Figure 8 (c).

It can be seen in Figure 9 (b) is a convolution between the original pdf and the added Gaussian noise which produces somewhat of a smoother histogram with a number of bumps. Furthermore, the peak of the histogram (intensity value = 0) appears to have decreased.

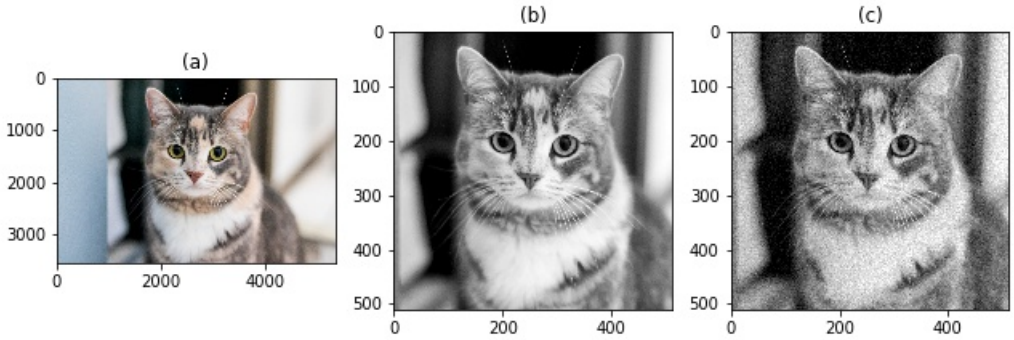

Figure 8: (a) Image2.jpg. (b) Cropped and resized image, saved as grayscale. (c) Added Gaussian noise.

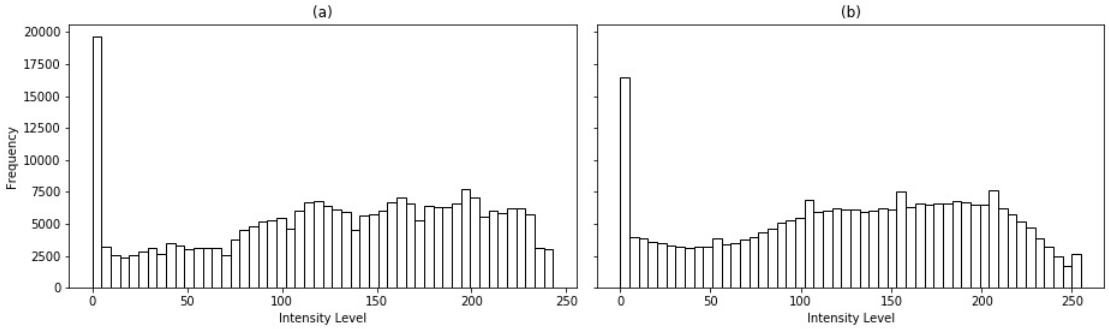

Figure 9: (a) Before (b) After adding noise

5.1.2 Gaussian Filter

Acknowledgement: The Gaussian kernels created in this task using the method provided in w3resource exercise and based on the isotropic 2D Gaussian distribution (Week3A-filtering lecture, slide 34).

With Figure 8 (c) as the noisy image, a Gaussian filter was applied using an 11×11 kernel. The filter is implemented by performing a neighbouring operation on each pixel $[x, y]$. In essence, we are replacing the intensity value of each pixel $[x, y]$ to a linear combination of the intensities of its neighbouring pixels. Note that in the border cases, `my-gauss-filter()` simply assumes the intensity value of pixels that are outside the boundaries to be 0. The level of smoothing is controlled by adjusting the standard deviation of the filter, which ranges from 0.5 to 4. The effect can be observed in Figure 10 (the upper row).

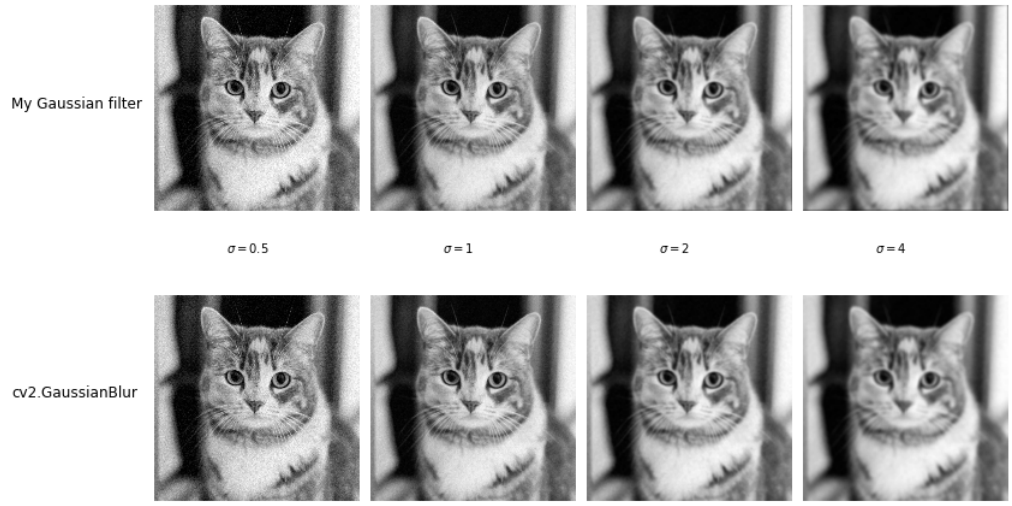

Figure 10: Figure 8 (c) with Gaussian filter applied with different kernels with increasing standard deviation.

An observation on Figure 10 (the upper row) shows that the denoising filter starts to take effect at $\sigma = 1$ as a large proportion of noise has been removed. Nevertheless, some noise is still visible in the background as well as in the white fur underneath the cat's neck. With $\sigma = 2$, the noise in said regions has been smoothed out. However, soft spots of what used to be obvious noise are visible. Another effect of increasing σ is blurring. Indeed, the blurring effect becomes more prevalent as σ increases. This phenomenon is due to the fact that we are giving higher weights to pixels that are far away from $[x, y]$.

The same process was done using Python's in-built function `cv2.GaussianBlur()` with the same padding method, from which the results are shown in Figure 10 lower row. Visually, the results are nearly identical. Numerically, using the MSE function with N:number of pixels 512×512, $\mathbf{u}$: the smoothed image from `my-gauss-filter()` and $\mathbf{v}$: the result produced by `cv2.GaussianBlur()`.

$$MSE = \frac{1}{N}\|\mathbf{u} - \mathbf{v}\|^2$$

The MSE is largest at $\sigma = 0.5$ which is equal to 234.07. This can be explained by the large amount of noise still present. It is smallest at $\sigma = 1$ which is equal to 21.45 (2 dp).

In conclusion, we can choose a Gaussian filter fitting to our goal and preference between extensive reduction of noise and blurring effect. For this task particularly, a filter of size 11×11 with σ between 1 and 2 appears to produce a result closest to the original image.

5.2 Bilateral filtering

5.2.1 Grayscale images

A problem in image denoising via Gaussian filter is that it is not edge-preserving. The blurring effect as mentioned does not distinguish between sharp edges and smooth areas. Hence, as observed in Figure 10, areas such as the cat's ears or whiskers are no longer crisp after applying the filter. Therefore, in this section we will use the bilateral filtering method to attempt the same goal. The process is rather similar as we once again replace the intensity value at $[x, y]$ with a linear combination of its surrounding pixels. However, there are now two parameters that control the weights appointed to each neighbour. Let $I'[x, y]$ be the new intensity value at $[x, y]$, and $[x + h, y + l]$ a neighbouring pixel.

- σ_d: belongs to the domain kernel which considers how far away $[x + h, y + l]$ from $[x, y]$
- σ_r: belongs to the range kernel which considers how similar the intensity value $I[x + h, y + l]$ to $I[x, y]$

A combination of two kernels will have the effect as follows:

- The weight of $I[x + h, y + l]$ on $I'[x, y]$ is inversely proportional to the distance between them.

- The weight of $I[x+h, y+l]$ on $I'[x, y]$ is proportional to how similar they are.

The main idea to bilateral filtering is that we only average over intensities that are similar to each other, hence edge-preserving. Using an 11×11 Gaussian kernel, Figure 11 (b) shows a significant improvement from that smoothed only by the Gaussian filter. The edge-preserving functionality is performing well such that the majority of details in the whiskers are captured. The blurring effect is also greatly reduced.

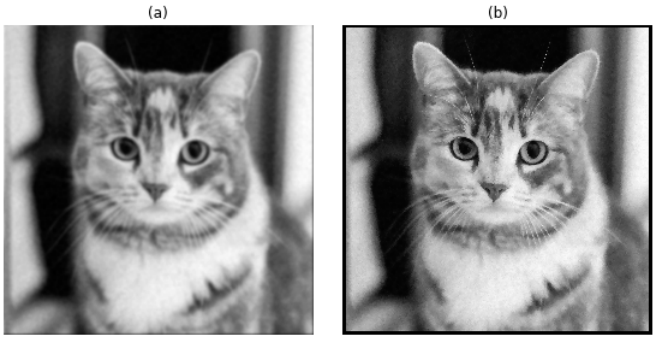

Figure 11: Figure 8 (c) smoothed via a Gaussian Filter (a) and a Bilateral Filter (b) with $\sigma_d = 2$ and $\sigma_r = 30$

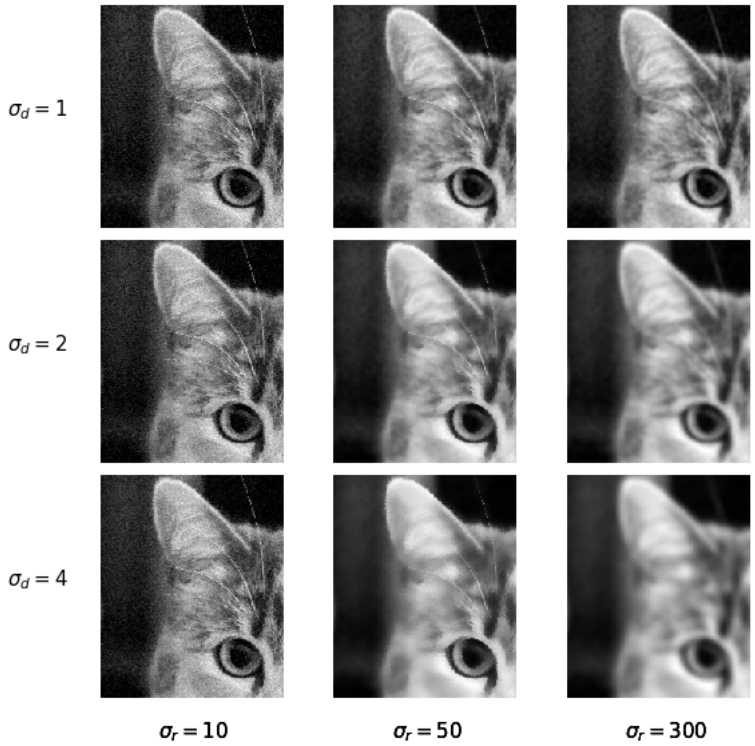

Figure 12: A detail from Figure 8 (c) denoised via a bilateral filter with varying σ_d and σ_r

Figure 12 shows the result of Figure 8 (c) processed via a Bilteral filter with rows correspond to different σ_d and columns to different σ_r. With a fixed $\sigma_r = 10$, increasing σ_d does not seem to blur the image at all. This is due to the fact that small σ_r will reduce the weight $I[x+h, y+l]$ has on $I'[x, y]$ if they are vastly different and will cancel out the blurring effect from the Gaussian kernel, which also explains the large amount of noise still present. For instance, let our pixel of interest have an intensity of 190. A nearby pixel with the same intensity will have a range weight of

$$exp\{\frac{-(190-190)^2}{2\times 10^2}\} = 1.$$

Any other pixel value that is further away from 190 will have its range weight quickly decreases to near zero. Given that the bilateral weight is product of range and domain weights, the blurring effect will subsequently reduce. The same does not hold for higher σ_r, however. Consider the same pixel of interest, a nearby pixel with an intensity of 45, and $\sigma_r = 300$. The large difference indicates a potential edge yet when we compute the range weight, it appears to be close to 1. This means that we are averaging over values of very different intensities, no longer preserving edges. Similarly, for a fixed σ_d, increasing σ_r will reduce the edge-preserving functionality.

$$exp\{\frac{-(45-190)^2}{2\times 300^2}\} = 0.8897$$

At $\sigma_r = 50$ and $\sigma_d = 10$, while the image is overall blurry, the edges are very well-preserved judging from the crispness of the cat's ear, whiskers and the inner corner of its eye.

5.2.2 Colour images

Before extending the bilateral filter to colour images, we first introduce noise to the colour image. To corrupt a colour image with Gaussian noise, we add noise separately to each of the colour channels and reconstruct the image using the same techniques as in section 2.3.

Implementing Bilateral filtering on noisy colour image: The bilateral filtering algorithm remains relatively the same when implemented on colour images. The only difference is that what used to be the dissimilarity in intensity values is now the difference in colours. Note that `my-bilateral-filter()` implemented do not have a padding method, thus results in the black/blue border surrounding the smoothed images. The Gaussian kernel used is 7×7 with $\sigma_d = 2$ and $\sigma_r = 40$.

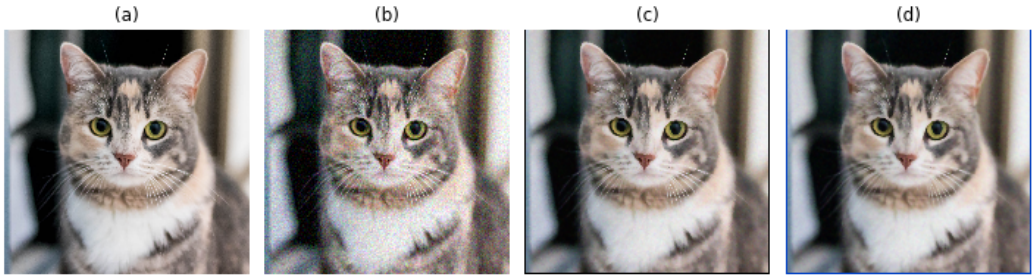

Figure 12: (a) Resized colour image (b) Added noise (c) Bilateral filter in RGB (d) Bilateral filter in CIE-Lab

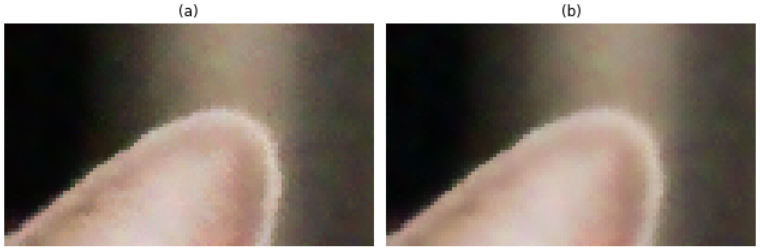

Figure 13: Detail from the smoothed image in RGB (a) and in CIE-Lab (b)

5.2.3 Further discussions

As noted by Tomasi C. and Manduchi R. (1998), an average between two grayscale intensities is another grayscale value. On the contrary, getting an average between two colours may result in a completely different colour. This explains the difficulty in smoothing in RGB, especially near the edges where there may exist a drastic change in colours. In fact, determining the similarity between perceived colour is the main problem of smoothing colour images. CIE-Lab colour space is a preferred choice since it is built upon the colour vision theory and that a changing its L*, a* and b* values will result in a perceptually similar colour (ibid.). Indeed, we can measure the colour difference between two pixels in CIE-Lab by taking the Euclidean distance between them.

Experimentally, results produced by applying bilateral filter in both RGB and CIE-Lab colour spaces (Figure 12) do not appear to be significantly different. When zoomed in the left ear, Figure 13 (b) shows that smoothing in the CIE-Lab colour space gives a cleaner result. Regarding the noise-removing functionality, both performed fairly well.

Furthermore, the bilateral filter does improve the edge-preserving property greatly. Figure 14 shows the result of smoothed colour images using Gaussian filter in both the RGB and CIE-Lab colour spaces. The results exhibit the same blurriness as seen in the grayscale case and when filtered in the CIE-Lab colour space, it appears that a slight blue undertone was introduced into the smoothed image.

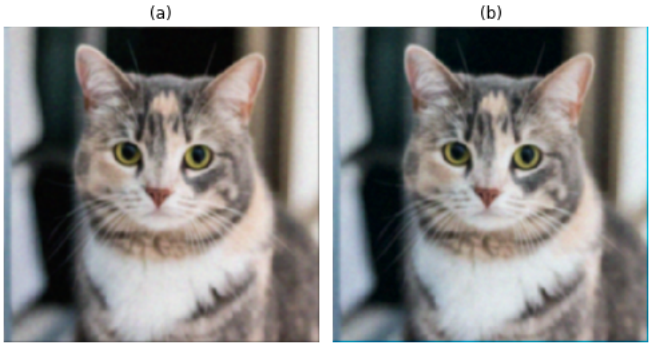

Figure 14: Colour image smoothed by a Gaussian filter in RGB (a) and CIE-Lab (b)

6 Task-6: Image Translation

6.1 Translation Function

Figure 15 shows the result of `image2.jpg` translated (2.0, 4.0), (-4.0, -6.0), (2.5, 4.5), (-0.9, 1.7), (92.0, -91.0), respectively. The translation is done by performing geometric operation on each pixel. With Δx and Δy as shifts in x and y directions, the operation can be defined as $I'[x + \Delta x, y + \Delta y] = I[x, y]$. Note that:

- Each output image has the shape of 712×712 to accommodate for larger translations.
- The origin of each image is at the top left corner. Thus a positive Δx corresponds to move rightward from $[0, 0]$ and a positive Δy to move downward from $[0, 0]$.
- If $x + \Delta x$ or $y + \Delta y$ is out of the boundaries, it will not loop over the boundaries.

The function used the forward mapping algorithm and splatting is implemented when $[x', y']$ is a partial pixel. For instance, if the destination coordinate is $[x', y'] = [x + \Delta x, y + \Delta y] = [0.5, 0.5]$ then each of its nearest neighbours will receive an even amount of intensity value from the original pixel.

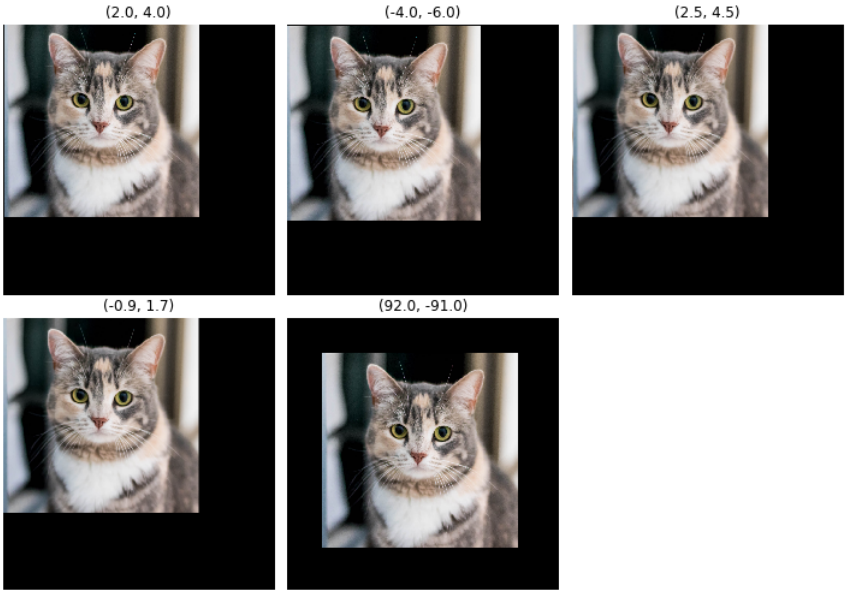

Figure 15: Translated images

6.2 Forward and Backward Mapping

Figure 16 shows the result of `image2.jpg` rotated 45 degrees counter-clockwise done by forward and backward mapping, respectively. The transformation is done by performing a geometric operation on each pixel.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = R \begin{bmatrix} x' \\ y' \end{bmatrix} \qquad R = \begin{bmatrix} cos(45) & sin(45) \\ -sin(45) & cos(45) \end{bmatrix}$$

Let the original image be I and output image be I'. With forward mapping, we map each pixel $[x, y]$ in I to a destination coordinate $[x', y']$ in I'. Similar to section 6.1, the splatting technique is used whenever $[x', y']$ are partial pixels. A problem quickly arises since this mapping function is not bijective. Therefore, there may be pixels in I' that is not paired with any pixel in I. Figure 16 (a) demonstrates this problem perfectly as "holes" can be seen across the image.

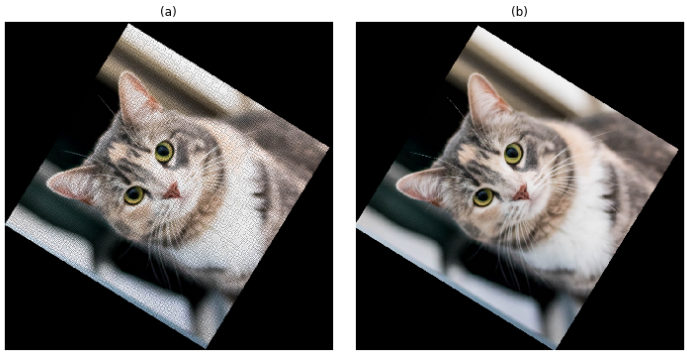

Figure 16: Rotated images using forward (a) and backward (b) mapping

Inverse mapping, on the other hand, iterate through each pixel [x', y'] in I' (which can be thought initially as a canvas) and check which location from the original image does it correspond to. Using the same transformation matrix R,

$$\begin{bmatrix} x \\ y \end{bmatrix} = R^{-1} \begin{bmatrix} x \\ y \end{bmatrix}$$

This algorithm is substantially better than forward mapping as it ensures that every pixel in I' did originate from some location in I. However, when $[x, y]$ are partial pixels, it is required to interpolate

the colour values from the neighbours of $[x, y]$. Figure 16 (b) shows the result of transformation R on I, using inverse mapping and the bilinear interpolation method which will be discussed below. The result is fairly good comparing to forward mapping as there are no "holes" in the output, all information from I is captured. This algorithm, however, does require an invertible mapping function. For instance, a shear transform is not invertible and thus would not work if our algorithm stay in the Cartesian coordinate system.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0.5 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 3 \\ 5 \end{bmatrix}$$

We can approach this problem by convert to the homogeneous coordinate system to successfully have an inverse transform and proceed with the transformation. The shear mapping function can be rewritten as follows:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 3 \\ 0.5 & 1 & 5 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

6.3 Different Interpolation Methods

In this section, we compare two interpolation methods: nearest neighbour and bilinear.

Assume we are at the $[x', y']$ location in I' and it has been mapped back to location $[x, y]$ in I. Continue assuming $[x, y]$ is a partial pixel, then the nearest neighbour method will interpolate the colour value at said pixel to be that of the nearest neighbour to $[x, y]$. Let the chosen neighbour be [x+a, y+b]

$$I'[x', y'] = I[x, y] = I[x + a, y + b]$$

A drawback to this technique, however, can be seen as we uniformly scale `image2` by a scalar of 10. Figure 17 (a) shows a detail in the scaled image interpolated by the nearest neighbour method and it can be seen that the edges of the eye are blocky and pixelated. This is due to the fact that we only took what was there in the original image and duplicate the pixels as we go on, making the pixelate effect more and more profound.

The bilinear method reduces this effect by really interpolating what may the colour value be in between pixels and essentially introducing new data (from the information that we already have). The new data is nothing a weighted sum of the colour values from the neighbouring pixels. The weights depend on the distance of [x,y] to each of its neighbours. Figure 17 (b) demonstrates this improvement fairly well as it produces much smoother edges along the eye.

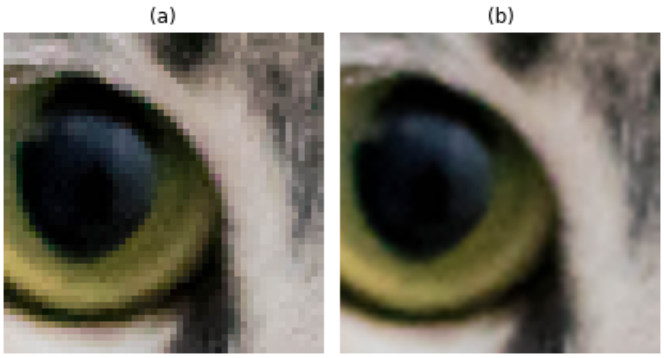

Figure 17: A detail from `image2.jpg` uniformly scaled by 10 using nearest neigbhour (a) and bilinear (b) interpolation methods

7 Resources

Szeliski, R. (2010, September 3). Computer Vision: Algorithms and Applications. Springer.

Tomasi, C.; Manduchi, R. (1998). Bilateral Filtering for Gray and Color Images. Sixth International Conference on Computer Vision. Mumbai. Retrieved from
http://www.cse.ucsc.edu/ manduchi/Papers/ICCV98.pdf

w3resources (2020, February 20) NumPy: Generate a generic 2D Gaussian-like array. Retrieved from
https://www.w3resource.com/python-exercises/numpy/python-numpy-exercise-79.php