Question 2 - Bilateral Filtering

November 23, 2018

1 Bilateral Filtering

We were tasked to perform bilateral filtering on noisy images. The images were affected with three types of noise, namely:
1. Salt, and pepper noise
2. Uniform noise
3. Salt & pepper, and uniform noise

1.1 The code

We start by importing all libraries required for performing the filtering
1. cv2: OpenCV is a library of programming functions mainly aimed at real-time computer vision.
2. numpy: NumPy adds support for large, multi-dimensional arrays along with a large collection of functions to operate on these arrays.
3. matplotlib: This is the most widely used plotting library available for python.

```
In [4]: import cv2
        import numpy as np
        from matplotlib import pyplot as plt
        plt.rcParams['figure.figsize'] = [15, 15]
```

We define two functions, one will be used to calculate the gaussian value given the standard deviation and the other to calculate the euclidean distance between two pixels.

```
In [29]: def gaussian(x, sigma):
             return (1.0 / (2 * np.pi * (sigma**2))) * np.exp(-(x**2) / (2 * sigma**2))


         def euclidean_distance(x1, y1, x2, y2):
             return np.sqrt((x1 - x2)**2 + (y1 - y2)**2)
```

The following two functions actually perform bilateral filtering on the image.

1. The function `bilateral_filter` is a wrapper which takes the input image and standard deviations across the range and domain as arguments to performs three tasks:
   - Pad the image with a border of 2 pixels width to make the implementation simpler.
   - Initialize an empty output container.

- Call the `apply_filter` for every pixel, except the borders padded earlier.
- Delete padded borders from the output and return the result.

```
In [30]: # sigma_r is the standard deviation for range filtering
         # sigma_d is the standard deviation for domain filtering
         def apply_filter(image, x, y, sigma_r, sigma_d):
             sum_of_weights = 0.0
             filtered_intensity = 0.0

             # perform filtering using a mask of 5x5
             for i in range(5):
                 for j in range(5):
                     n_x = x + 2 - i
                     n_y = y + 2 - j

                     # calcuate photometric weights and geometric weights
                     weight_range = gaussian(image[n_x][n_y] - image[x][y], sigma_r)
                     weight_domain = gaussian(
                         euclidean_distance(x, y, n_x, n_y), sigma_d)

                     weight = weight_range * weight_domain

                     filtered_intensity += image[n_x][n_y] * weight
                     sum_of_weights += weight

             # update the output image
             filtered_intensity = filtered_intensity / sum_of_weights

             return filtered_intensity
```

2. The function `apply_filter` take six arguments, namely, the input image, the output container, the coordinates of the pixel to apply the filter on, and the standard deviations across the range and domain.

   - It performs bilateral filtering with a mask size of 5x5.
   - Calculates the weights for geometric and photometric locality using the `gaussian` and `euclidean_distance` functions defined above.
   - Update the output image value.

```
In [34]: def bilateral_filter(source, sigma_r, sigma_d):
             # pad the image with a border of 2 pixels
             s = cv2.copyMakeBorder(
                 source, 2, 2, 2, 2, cv2.BORDER_CONSTANT, value=(255))

             # initialize the output container
             output = s - s
             output = output.astype('float64')
             output.setflags(write=True)
```

```
        # apply filter over each pixel
        for i in range(2, len(s) - 1):
            for j in range(2, len(s[0]) - 1):
                output[i][j] = apply_filter(s, i, j, sigma_r, sigma_d)

        # delete the padded borders
        output = np.delete(
            output, [0, 1, len(output[0]) - 1,
                     len(output[0]) - 2], axis=1)
        output = np.delete(
            output, [0, 1, len(output) - 1, len(output) - 2], axis=0)

        return output
```

We'll now read the input images using cv2's imread method. OpenCV's imread reads the image in the BGR color-space by default which is converted to GRAY by adding the 0 argument.

```
In [6]: # image affected by salt and pepper noise only
        spnoisy = cv2.imread("/home/sarvesh/Projects/Image Processing/spnoisy.jpg", 0)
        spnoisy = spnoisy.astype('float64')
        # image affected by uniform noise only
        unifnoisy = cv2.imread("/home/sarvesh/Projects/Image Processing/unifnoisy.jpg",
                               0)
        unifnoisy = unifnoisy.astype('float64')
        # image affected by uniform and salt & pepper noise both
        spunifnoisy = cv2.imread(
            "/home/sarvesh/Projects/Image Processing/spunifnoisy.jpg", 0)
        spunifnoisy = spunifnoisy.astype('float64')
```

Our results will vary depending on the variances we choose for domain and range filtering. To find the best results we experimented with a range of values for both the variances and got results for each of them.

Comparing the outputs will help us decide the variances to choose for the best results.

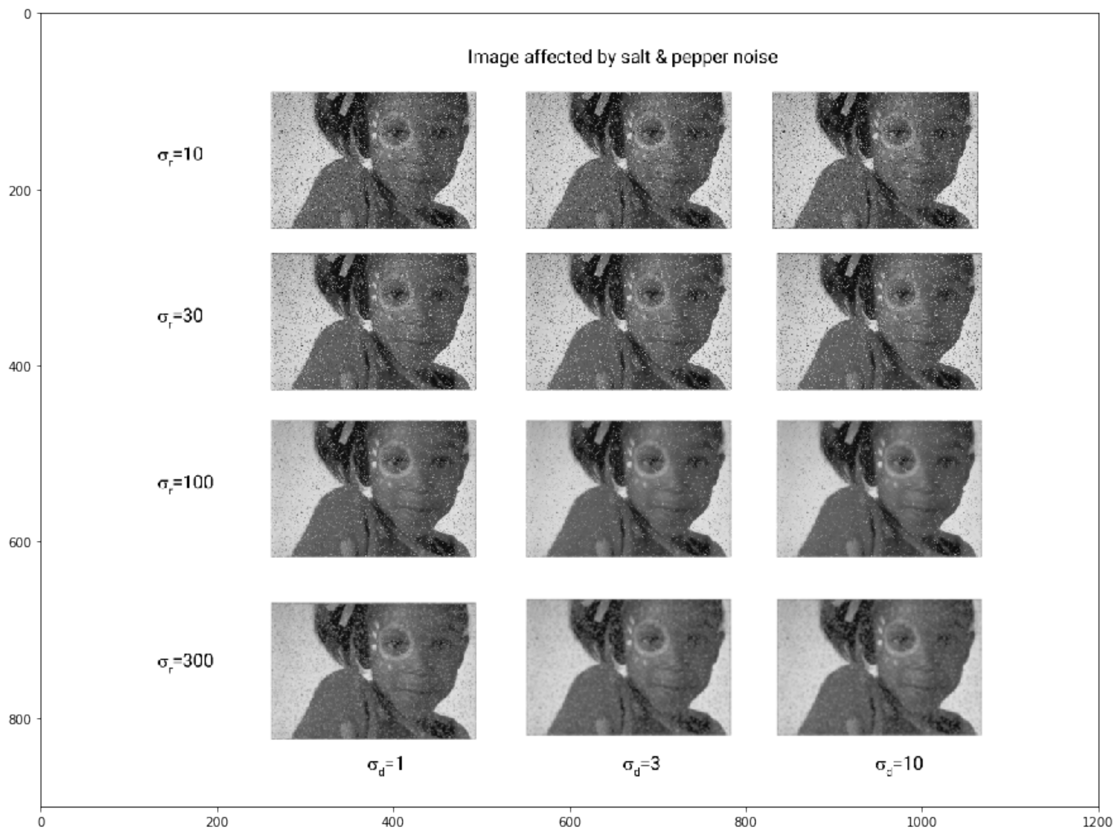
```
In [33]: # define the standard deviations for range
         r_sigmas = [10, 30, 100, 300]

         # define the standard deviations for domain
         d_sigmas = [1, 3, 10]

         for r in r_sigmas:
             for d in d_sigmas:
                 # perform bilateral filtering for each type of image and save the image
                 spnoisy_filtered = bilateral_filter(spnoisy, r, d)
                 name = "spnoisy_filtered_" + str(r) + "_" + str(d)
                 cv2.imwrite(name + ".jpeg", spnoisy_filtered)

                 unifnoisy_filtered = bilateral_filter(unifnoisy, r, d)
                 name = "unifnoisy_filtered_" + str(r) + "_" + str(d)
```

```
cv2.imwrite(name + ".jpeg", unifnoisy_filtered)

spunifnoisy_filtered = bilateral_filter(spunifnoisy, r, d)
name = "spunifnoisy_filtered_" + str(r) + "_" + str(d)
cv2.imwrite(name + ".jpeg", spunifnoisy_filtered)
```

1.2 Observations and Results

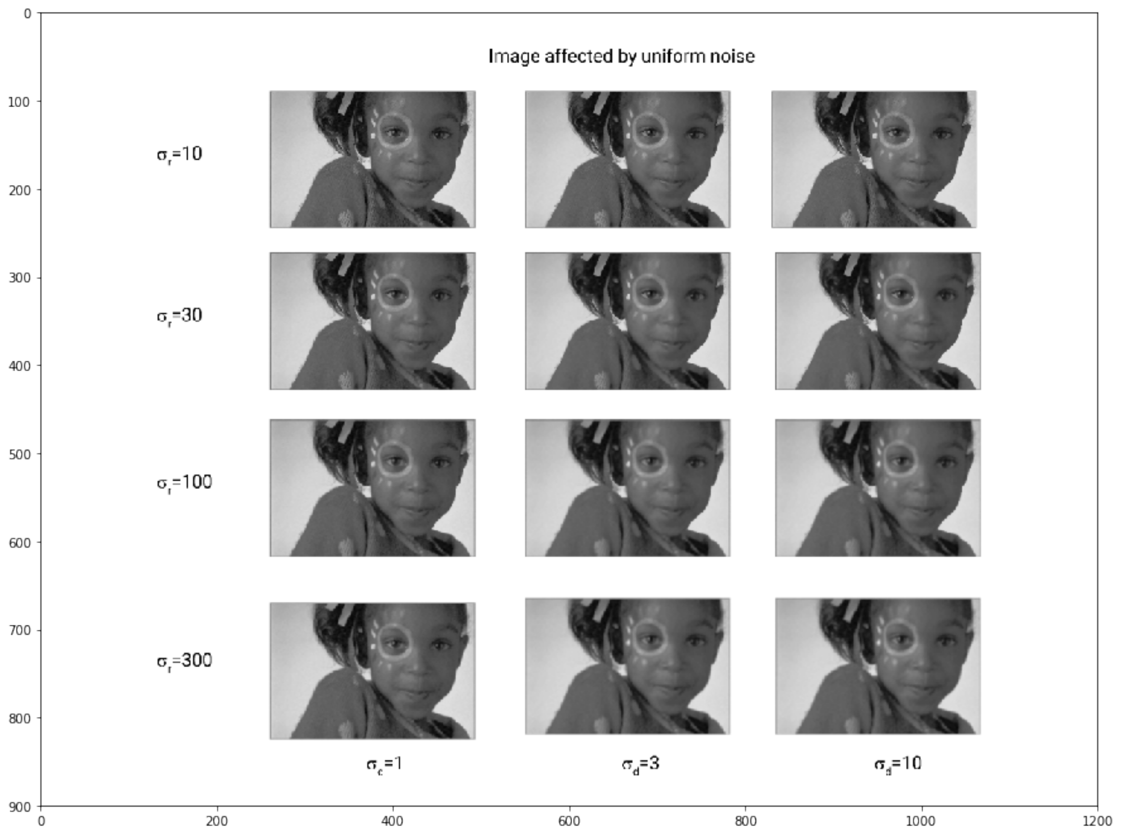
- **Salt and Pepper noise**: Here are the filtered images of the image affected with salt and pepper noise for different variances. We can observe that the best result is obtained with $\sigma_r = 100$ and $\sigma_d = 10$.

```
In [37]: spnoisy_comp = cv2.imread(
             "/home/sarvesh/Projects/Image Processing/bilateral sp.png", 0)

         plt.imshow(spnoisy_comp, cmap='gray')
         plt.show()
```



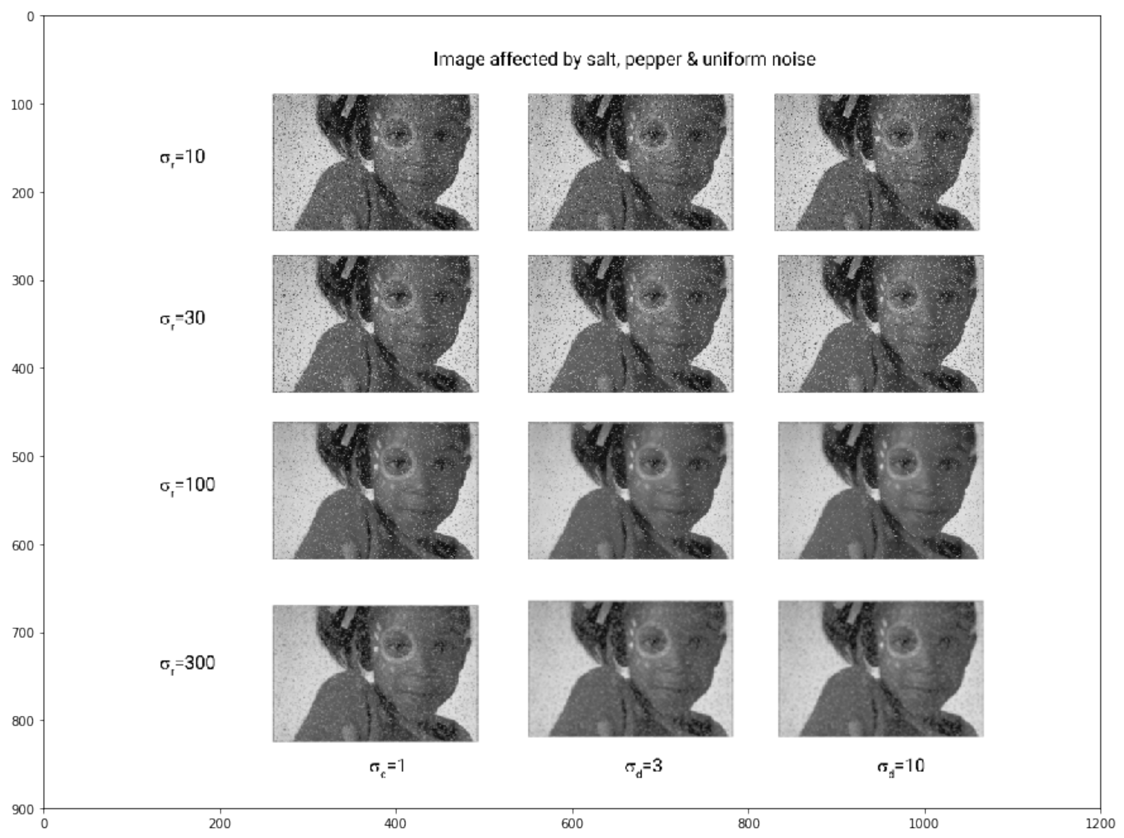
- **Uniform noise**: Here are the filtered images of the image affected with uniform noise for different variances. We observe that bilateral filtering performs We can observe that the best result is obatained with $\sigma_r = 100$ and $\sigma_d = 3$

```
In [35]: unifnoisy_comp = cv2.imread(
             "/home/sarvesh/Projects/Image Processing/bilateral uniform.png", 0)

         plt.imshow(unifnoisy_comp, cmap='gray')
         plt.show()
```



- **Salt & pepper and uniform noise**: Here are the filtered images of the image affected with salt & pepper and uniform noise for different variances. We can observe that the best result is obatained with $\sigma_r = 100$ and $\sigma_d = 3$

```
In [36]: spunifnoisy_comp = cv2.imread(
             "/home/sarvesh/Projects/Image Processing/bilateral sp unif.png", 0)

         plt.imshow(spunifnoisy_comp, cmap='gray')
         plt.show()
```



1.3 Conclusion

Bilateral filtering was performed on all three kinds of images.