Computer Vision Homework 8

B03902042 宋子維

Description

In this homework, we are asked to generate noisy imageS by **salt-and-pepper** and **gaussian noise** method with respect to the gray level image in *Figure 1.1*. After generating noisy images, we need to do **noise-removal** on these images and calculate **signal-to-ratio (SNR)** for each instance.

Figure 1-1: Gray level image.

Programming

I use python to implement the algorithms. There are three python programs, namely, **noisy-image.py**, **noise-removal.py** and **SNR.py**, where I use **pillow** to process basic image I/O. In the program, there are some basic functions and instructions:

1. `PIL.Image.open(img)`: load the image `img` and return a pillow **Image** object.

2. `pix = Image.load()`: return the **PixelAccess** object of **Image** object to `pix`, which offers us to use `pix[x, y]` to access the pixel value at position (x, y).

3. `PIL.Image.new(mode, size)`: create a new pillow **Image** object given its mode and size.

4. `Image.size`: pair (width, height) of **Image** object.

5. `sumTuple((a, b), (c, d))`: return the tuple (a+c, b+d).

6. `product(range(a), range(b))`: return the list of cartesian product of set $\{0, 1, \cdots, a-1\}$ and set $\{0, 1, \cdots, b-1\}$.

Usage

The usages of each programs are in the following:

noisy-image.py:

```
python3 noisy-image.py IMG-IN IMG-OUT METHOD VALUE
```

noise-removal.py:

```
python3 noise-removal.py IMG-IN IMG-OUT FILTER [VALUE]
```

SNR.py:

```
python3 SNR.py IMG-IN IMG-OUT
```

For **noisy-image.py**, METHOD can be "gaussian" or "salt_pepper", and VALUE is the parameter to that METHOD. For **noise-removal.py**, FILTER can be "box", "median", "open_close", or "close_open", and the optional VALUE is the size of the filter.

Algorithms

Noisy image

Salt-and-Pepper: Probability = 0.05

Figure 2-1-1: Before.

Figure 2-1-2: After.

Salt-and-Pepper: Probability = 0.1

Figure 2-2-1: Before

Figure 2-2-2: After

Given the source image, namely SrcImg, and probability f, we can generate the noisy image by *salt-and-pepper* as follows:

$$\text{NoisyImg}(i,j) = \begin{cases} 0 & \text{if } random() \leq \frac{f}{2} \\ 255 & \text{if } random() \geq 1 - \frac{f}{2} \\ \text{SrcImg}(i,j) & \text{otherwise} \end{cases}$$

Then we can get the following python code, where `pix` is the gray level image, that is, *lena.bmp*, and `random.random()` return a random value with uniform distribution in $(0, 1)$.

```
def salt_pepper(pix, size, prob):
    ret = {}
    for position in product(range(size[0]), range(size[1])):
        r = random.random()
        if r < prob/2:
            ret[position] = PEPPER
        elif r > 1 - prob/2:
            ret[position] = SALT
        else:
            ret[position] = pix[position]
    return ret
```

Gaussian Noise: Amplitude = 10

Figure 2-3-1: Before

Figure 2-3-2: After

Gaussian Noise: Amplitude = 30

Figure 2-4-1: Before

Figure 2-4-2: After

Given the source image, namely SrcImg, and amplitude amp, we can generate the noisy image by *salt-and-pepper* as follows:

$$\text{NoisyImg}(i,j) = \text{SrcImg}(i,j) + amp \cdot N(0,1)$$

Then we can get the following python code, where `pix` is the gray level image, that is, *lena.bmp*, and `random.gauss())` return a random value with normal distribution with mean 0 and variance 1.

```
def gaussian_noise(pix, size, amp):
    ret = {}
    for position in product(range(size[0]), range(size[1])):
        ret[position] = pix[position] + int(amp * random.gauss(0, 1))
    return ret
```

Noise removal

First, see the results:

On salt-and-pepper noise with probability 0.05

Figure 3-1-1: Origin: salt-and-pepper noise with probability 0.05

Figure 3-1-2: 3x3 box filter on salt-and-pepper 0.05

Figure 3-1:3: 5x5 box filter on salt-and-pepper 0.05

Figure 3-1-4: 3x3 median filter on salt-and-pepper 0.05

Figure 3-1-5: 5x5 median filter on salt-and-pepper 0.05

Figure 3-1-6: Opening-then-closing filter on salt-and-pepper 0.05

Figure 3-1-7: Closing-then-opening filter on salt-and-pepper 0.05

On salt-and-pepper noise with probability 0.1

Figure 3-2-1: Origin: salt-and-pepper noise with probability 0.1

Figure 3-2-2: 3x3 box filter on salt-and-pepper 0.1

Figure 3-2-3: 5x5 box filter on salt-and-pepper 0.1

Figure 3-2-4: 3x3 median filter on salt-and-pepper 0.1

Figure 3-2-5: 5x5 median filter on salt-and-pepper 0.1

Figure 3-2-6: Opening-then-closing filter on salt-and-pepper 0.1

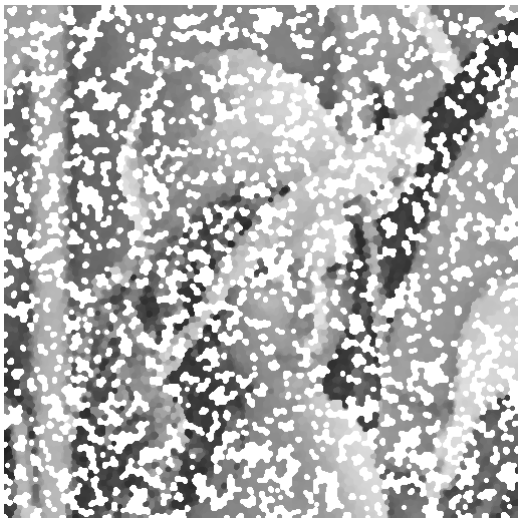

Figure 3-2-7: Closing-then-opening filter on salt-and-pepper 0.1

On Gaussian noise with amplitude 10

Figure 3-3-1: Origin: Gaussian noise with amplitude 10

Figure 3-3-2: 3x3 box filter on Gaussian noise 10

Figure 3-3-3: 5x5 box filter on Gaussian noise 10

Figure 3-3-4: 3x3 median filter on Gaussian 10

Figure 3-3-5: 5x5 median filter on Gaussian 10

Figure 3-3-6: Opening-then-closing filter on Gaussian 10

Figure 3-3-7: Closing-then-opening filter on Gaussian 10

On Gaussian noise with amplitude 30

Figure 3-3-1: Origin: Gaussian noise with amplitude 30

Figure 3-3-2: 3x3 box filter on Gaussian noise 30

Figure 3-3-3: 5x5 box filter on Gaussian noise 30

Figure 3-3-4: 3x3 median filter on Gaussian 30

Figure 3-3-5: 5x5 median filter on Gaussian 30

Figure 3-3-6: Opening-then-closing filter on Gaussian 30

Figure 3-3-7: Closing-then-opening filter on Gaussian 30

Box filter

Box filter method simply sums up all the pixels within the filter region with respect to the point. If the point in that region is out of boundary, just skip it. Then we can get the following python code:

```
def box_filter(pix, size, kernel):
    ret = {}
    for position in product(range(size[0]), range(size[1])):
        trans = T(pix, position, kernel)
        ret[position] = int(sum(trans) / len(trans))
    return ret
```

Function `T()` return the translation at `position` with respect to `kernel`.

Median filter

Median filter method first collects all the pixels within the filter region with respect to the point, and finds the median of the set. If the point in that region is out of boundary, just skip it. Then we can get the following python code:

```
def median(list):
    l = len(list)
    if l % 2 == 1:
        return list[int((l+1)/2)-1]
    elif l % 2 == 0:
        return (list[int((l+1)/2)-1] + list[int((l+1)/2)])/2

def median_filter(pix, size, kernel):
    ret = {}
    for position in product(range(size[0]), range(size[1])):
        ret[position] = int(median(sorted(T(pix, position, kernel))))
    return ret
```

Opening-then-closing filter and closing-then-opening filter

In the previous assignments, I implement functions of dilation, erosion, opening and closing. Thus, for these two filters, simply apply the functions I wrote before. Then we can get the following python code:

```
def dilation(pix, size, kernel):
    width, height = size
    ret = {(x, y): BLACK for x in range(width) for y in range(height)}

    for x in range(width):
        for y in range(height):
            for _ in kernel:
                X, Y = sumTuple((x, y), _)
                if 0 <= X < width and 0 <= Y < height:
                    ret[x, y] = max(ret[x, y], pix[X, Y])

    return ret

def erosion(pix, size, kernel):
    width, height = size
    ret = {(x, y): WHITE for x in range(width) for y in range(height)}

    for x in range(width):
        for y in range(height):
            for _ in kernel:
                X, Y = sumTuple((x, y), _)
                if 0 <= X < width and 0 <= Y < height:
                    ret[x, y] = min(ret[x, y], pix[X, Y])

    return ret

def opening(pix, size, kernel):
    return dilation(erosion(pix, size, kernel), size, kernel)

def closing(pix, size, kernel):
    return erosion(dilation(pix, size, kernel), size, kernel)

def open_close(pix, size, kernel):
    return closing(opening(pix, size, kernel), size, kernel)

def close_open(pix, size, kernel):
    return opening(closing(pix, size, kernel), size, kernel)
```

Calculate SNR

3x3 box filter on salt-and-pepper with probability 0.05: 9.470704681985078
5x5 box filter on salt-and-pepper with probability 0.05: 11.17749451315036
3x3 median filter on salt-and-pepper with probability 0.05: 19.156426813662478
5x5 median filter on salt-and-pepper with probability 0.05: 16.404108623906417
Opening-then-closing filter on salt-and-pepper with probability 0.05: 5.48393152121621
Closing-then-opening filter on salt-and-pepper with probability 0.05: 5.6841471189703405

3x3 box filter on salt-and-pepper with probability 0.1: 6.306083396629524
5x5 box filter on salt-and-pepper with probability 0.1: 8.478350647773143
3x3 median filter on salt-and-pepper with probability 0.1: 14.722227063278059
5x5 median filter on salt-and-pepper with probability 0.1: 15.732507459248545
Opening-then-closing filter on salt-and-pepper with probability 0.1: -2.1181547899236937
Closing-then-opening filter on salt-and-pepper with probability 0.1: -2.62302221114264

3x3 box filter on Gaussian noise with amplitude 10: 17.837839955606093
5x5 box filter on Gaussian noise with amplitude 10: 14.879399785147543
3x3 median filter on Gaussian noise with amplitude 10: 17.966147702108966
5x5 median filter on Gaussian noise with amplitude 10: 16.087501382531705

Opening-then-closing filter on Gaussian noise with amplitude 10: 13.25248320967219
Closing-then-opening filter on Gaussian noise with amplitude 10: 13.591184572281627

3x3 box filter on Gaussian noise with amplitude 30: 12.69197568648684
5x5 box filter on Gaussian noise with amplitude 30: 13.31012963742709
3x3 median filter on Gaussian noise with amplitude 30: 11.311332656422564
5x5 median filter on Gaussian noise with amplitude 30: 13.04778315352085
Opening-then-closing filter on Gaussian noise with amplitude 30: 11.216419756579958
Closing-then-opening filter on Gaussian noise with amplitude 30: 11.173206604947008

By the definition of SNR, we can simply write down the following code:

```
def SNR(S, N, size):
    mean, mean_N = 0, 0
    n = size[0] * size[1]
    for position in product(range(size[0]), range(size[1])):
        mean += S[position]
        mean_N += (N[position] - S[position])
    mean /= n
    mean_N /= n

    VS, VN = 0, 0
    for position in product(range(size[0]), range(size[1])):
        VS += (S[position] - mean) ** 2
        VN += (N[position] - S[position] - mean_N) ** 2
    VS /= n
    VN /= n

    return (10 * math.log10(VS/VN))
```