Computer Vision HW8 Report

B01902044 Steven Lin

2015-12-7

Contents

1 Source Code

1.1 Creating Noise

In this homework, we need to create noise from the original lena.png image, using Gaussian Noise and Salt-and-Pepper Noise. Therefore, for the 2 different kinds of noise, I implemented two functions to help me create the noise.

1.1.1 Gaussian Noise

There's an important function I implemented for createing Gaussian Noise. Please see to the following code snippet.

```
def createGaussianNoise(src, amplitude):
    newImage = Image.new(src.mode, src.size)
    newImagePixels = newImage.load()
    srcPixels = src.load()
    # setup the 2 parameters for random.gauss() function.
    gaussian_mu = 0
    gaussian_sigma = 1
    for i in range(newImage.size[0]):
        for j in range(newImage.size[1]):
            value = int(srcPixels[i, j] + amplitude * random.gauss(gaussian_mu, gaussian_sigma))
            newImagePixels[i, j] = max(0, min(255, value))
    return newImage
```

Figure 1: createGaussianNoise() function

The first parameter of this function is the source image to create Gaussian Noise, and the second parameter is the amplitude to create the noise. This function returns a new noisy image.

1.1.2 Salt-and-Pepper Noise

Likewise, the following code snippet shows the function I implemented for creating Salt-and-Pepper Noise.

```python
def createSaltAndPepperNoise(src, probability):
    newImage = Image.new(src.mode, src.size)
    newImagePixels = newImage.load()
    srcPixels = src.load()
    for i in range(newImage.size[0]):
        for j in range(newImage.size[1]):
            ran = random.uniform(0, 1)
            if ran < probability:
                newImagePixels[i, j] = 0
            elif ran > 1 - probability:
                newImagePixels[i, j] = 255
            else:
                newImagePixels[i, j] = srcPixels[i, j]
    return newImage
```

Figure 2: createSaltAndPepperNoise() function

The first parameter of the function is the source image to create noise, and the second one is the probability for creating Salt-and-Pepper Noise. This function also returns a new image with the desired noise.

1.2 Box Filtering

Since we will be using box filtering for plenty of times in this homework, I think it's better to have it as a function. And with ***boxSize*** as one of parameters of the function, we are able to create 3x3 and 5x5 box filtering easily. Please see to the following code snippet for more details.

```python
def boxFiltering(src, boxSize):
    newImage = Image.new(src.mode, src.size)
    newImagePixels = newImage.load()
    srcPixels = src.load()
    # start the main loop
    for i in range(newImage.size[0]):
        for j in range(newImage.size[1]):
            # get the box origin, ex: boxSize[5, 5] will result in box0[i-2, j-2]
            box0 = [i - (boxSize[0]-1)/2, j - (boxSize[1]-1)/2]
            # get box contents
            boxContents = []
            for m in range(boxSize[0]):
                for n in range(boxSize[1]):
                    p = [box0[0] + m, box0[1] + n]
                    # check if in range
                    if p[0] >= 0 and p[0] < newImage.size[0] and p[1] >= 0 and p[1] < newImage.size[1]:
                        boxContents.append(srcPixels[p[0], p[1]])
            newImagePixels[i, j] = int(sum(boxContents) / len(boxContents))
    return newImage
```

Figure 3: boxFiltering() function

1.3 Median Filtering

Similar to the reason mentioned in the previous section, here we have a function for median filtering. Code:

```
def medianFiltering(src, boxSize):
    newImage = Image.new(src.mode, src.size)
    newImagePixels = newImage.load()
    srcPixels = src.load()
    for i in range(newImage.size[0]):
        for j in range(newImage.size[1]):
            # get the box origin
            boxO = [i - (boxSize[0]-1)/2, j - (boxSize[1]-1)/2]
            # get box content
            boxContents = []
            for m in range(boxSize[0]):
                for n in range(boxSize[1]):
                    p = [boxO[0] + m, boxO[1] + n]
                    # check if in range
                    if p[0] >= 0 and p[0] < newImage.size[0] and p[1] >= 0 and p[1] < newImage.size[1]:
                        boxContents.append(srcPixels[p[0], p[1]])
            boxContents.sort()
            if len(boxContents) % 2 == 1: # odd length
                newImagePixels[i, j] = boxContents[len(boxContents)/2]
            else: # even length
                newImagePixels[i, j] = (boxContents[len(boxContents)/2-1] + boxContents[len(boxContents)/2])/2
    return newImage
```

Figure 4: medianFiltering() function

1.4 Opening-then-Closing

This part of the de-noising doesn't take much coding because we already have the code for opening and closing in one of the previous homework assignments.

1.4.1 Opening

As in my previous homework, I created a class for representing kernel used in grayscale morphology, and utilized dilation and erosion to achieve both opening and closing. See to the following code snippet.

```
def opening(src, kernel):
    # simply take advantage of erosion() and dilation()
    return dilation(erosion(src, kernel), kernel)
```

Figure 5: opening() function

1.4.2 Closing

Similarly, we have this closing() function as follows.

```python
def closing(src, kernel):
    # simply take advantage of dilation() and erosion()
    return erosion(dilation(src, kernel), kernel)
```

Figure 6: closing() function

1.5 Closing-then-Opening

The core concept of implementing closing-then-opening in my program is well explained in the above section. All I have to do is simply reverse the order of the opening() and closing() function calls. For ex:

```python
# closing-then-opening
g10_close_then_open = opening(closing(g10, octoKernel), octoKernel)
g30_close_then_open = opening(closing(g30, octoKernel), octoKernel)
snp10_close_then_open = opening(closing(snp10, octoKernel), octoKernel)
snp5_close_then_open = opening(closing(snp5, octoKernel), octoKernel)
```

Figure 7: how to get closing-then-opening

1.6 SNR Computing

Calculating the SNR value for the 28 images is not a difficult job, by following the formula described in the course slide, we can achieve without much effort.

$$SNR = 20 \times \log_{10} \frac{\sqrt{V_S}}{\sqrt{V_N}} \tag{1}$$

For my implementation of calculating SNR, please see to the following code snippet.

```
def getSNR(src, noisyImage):
    # check size
    if src.size != noisyImage.size:
        print >> sys.stderr, "ERROR: trying to calculate SNR for 2 images with different size"
        print >> sys.stderr, "src image size = ", src.size
        print >> sys.stderr, "noisyImage size = ", noisyImage.size
        sys.exit(1)
    # calculate the mu for the 2 images
    total_pixel_count = src.size[0] * src.size[1]
    srcPixels = src.load()
    noisyImagePixels = noisyImage.load()
    mu1, mu2 = 0.0, 0.0
    for i in xrange(src.size[0]):
        for j in xrange(src.size[1]):
            mu1 += srcPixels[i, j]
            mu2 += noisyImagePixels[i, j] - srcPixels[i, j]
    mu1 = float(mu1) / total_pixel_count
    mu2 = float(mu2) / total_pixel_count
    # calculate the standard deviation for the 2 images
    sigma1, sigma2 = 0.0, 0.0
    for i in xrange(src.size[0]):
        for j in xrange(src.size[1]):
            sigma1 += (srcPixels[i, j] - mu1) * (srcPixels[i, j] - mu1)
            sigma2 += (noisyImagePixels[i, j] - srcPixels[i, j] - mu2) * (noisyImagePixels[i, j] - srcPixels[i, j] - mu2)
    sigma1 = float(sigma1) / total_pixel_count
    sigma2 = float(sigma2) / total_pixel_count
    return 20 * log(sqrt(sigma1)/sqrt(sigma2), 10)
```

Figure 8: SNR calculating function

2 Results

All of the resulted images are also saved properly and submitted along with this document.

2.1 Gaussain Noise with amplitude=10

Figure 9: Noisy image

Figure 10: 3x3 Box Filtering

Figure 11: 5x5 Box Filtering

Figure 12: 3x3 Median Filtering

Figure 13: 5x5 Median Filtering

Figure 14: Opening-then-closing

Figure 15: Closing-then-opening

2.2 Gaussain Noise with amplitude=30

Figure 16: Noisy image

Figure 17: 3x3 Box Filtering

Figure 18: 5x5 Box Filtering

Figure 19: 3x3 Median Filtering

Figure 20: 5x5 Median Filtering

Figure 21: Opening-then-closing

Figure 22: Closing-then-opening

2.3 Salt-and-Pepper Noise with prob=0.1

Figure 23: Noisy image

Figure 24: 3x3 Box Filtering

Figure 25: 5x5 Box Filtering

Figure 26: 3x3 Median Filtering

Figure 27: 5x5 Median Filtering

Figure 28: Opening-then-closing

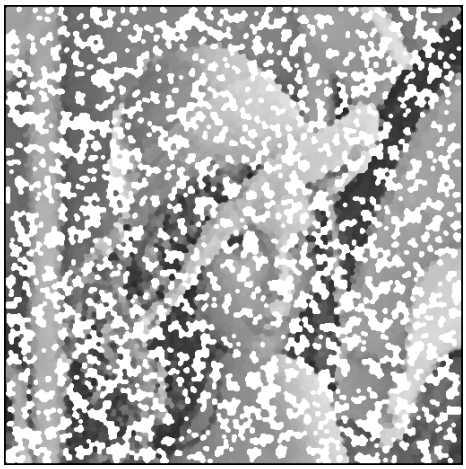

Figure 29: Closing-then-opening

2.4 Salt-and-Pepper Noise with prob=0.05

Figure 30: Noisy image

Figure 31: 3x3 Box Filtering

Figure 32: 5x5 Box Filtering

Figure 33: 3x3 Median Filtering

Figure 34: 5x5 Median Filtering

Figure 35: Opening-then-closing

Figure 36: Closing-then-opening

2.5 SNR Results

2.5.1 Noisy Images

Here are the SNR values of the noisy images.

Gaussian 10
13.5973213697

Gaussian 30
4.18590573877

Salt-and-Pepper 0.1
-2.10802710151

Salt-and-Pepper 0.05
0.888403235701

2.5.2 With 3x3 Box Filtering

Gaussian 10
17.7477046237

Gaussian 30
12.5971123619

Salt-and-Pepper 0.1
6.32976323195

Salt-and-Pepper 0.05
9.42027035993

2.5.3 With 5x5 Box Filtering

Gaussian 10
14.8739539066

Gaussian 30
13.2617994698

Salt-and-Pepper 0.1
8.47964501322

Salt-and-Pepper 0.05
11.1237985451

2.5.4 With 3x3 Median Filtering

Gaussian 10
17.6746771769

Gaussian 30
11.0916016474

Salt-and-Pepper 0.1
15.3319866915

Salt-and-Pepper 0.05
19.2838748041

2.5.5 With 5x5 Median Filtering

Gaussian 10
16.0135217409

Gaussian 30
12.8607428467

Salt-and-Pepper 0.1
15.7098950933

Salt-and-Pepper 0.05
16.3666112778

2.5.6 With Opening-then-Closing

Gaussian 10
8.60312968987

Gaussian 30
8.62571452533

Salt-and-Pepper 0.1
-2.2963072715

Salt-and-Pepper 0.05
4.13552957986

2.5.7 With Closing-then-Opening

Gaussian 10
7.66164001826

Gaussian 30
6.07162697144

Salt-and-Pepper 0.1
-3.09313393168

Salt-and-Pepper 0.05
3.70413617776