

Question 1. Color Quantization and Dithering

In this question, I'm given a 24-bits color jpeg image, and be requested to do color quantization with median-cut algorithm and error diffusion dithering.

1. Preprocess

Initially, I use PIL library to read the image and transform to numpy(short for np) array.

```
1 originalImg = Image.open('./img/Lenna.jpeg')
2 rgbArray = np.array(originalImg)
```

2. Median-cut color quantization

These are the step I implement this algorithm:

- A. Reshape the input image into a 2D array of pixels.
- B. Initialize a list of cubes with the entire array of pixels.
- C. While the number of cubes is less than 2 to the power of the number of bits, divide each cube into two smaller cubes and add them to the list of cubes.

```
1 while len(cubes)<2**bits:
2     newCubes = []
3     for cube in cubes:
4         leftCube, rightCube = util.SplitCube(cube)
5         newCubes.append(leftCube)
6         newCubes.append(rightCube)
7     cubes = newCubes
```

```
1 def SplitCube(cube):
2     ranges = np.max(cube, axis=0) - np.min(cube, axis=0)
3     dim = np.argmax(ranges)
4     sortedcube = cube[cube[:, dim].argsort()]
5     medianidx = len(cube) // 2
6     leftcube = sortedcube[:medianidx]
7     rightcube = sortedcube[medianidx:]
8     return leftcube, rightcube
```

- D. Compute the mean color of each cube.

```
8 colors = [util.MedianColor(cube) for cube in cubes]
```

```
1 def MedianColor(cube):
2     return np.array(np.mean(cube, axis=0).astype(int))
```

E. Replace each pixel in the original image with its closest median color.

```
1 for y in range(img.shape[0]):  
2     for x in range(img.shape[1]):  
3         pixel = img[y, x]  
4         newRgbArray[y, x] = util.ClosestColor(pixel, colors)
```

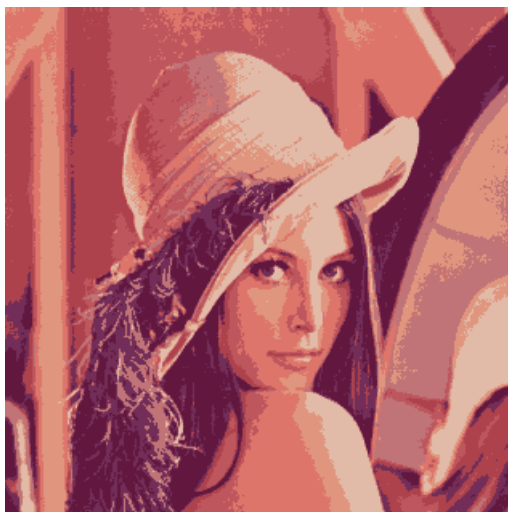
```
1 def ClosestColor(color, color_list):  
2     idx = np.argmin(distance.cdist([color], color_list, metric='euclidean'))  
3     return color_list[idx]
```

F. Compute the color quantization MSE error by comparing the original and quantized images.

```
1 def MSE(oriImg, newImg):  
2     return np.mean((oriImg - newImg) ** 2)
```

The MSE in case 3-bit is roughly about 64.8677, and the MSE in the case 6- bit is about 22.9243

```
MSE for 3 bits: 64.86774288308497  
MSE for 6 bits: 22.924317684131818
```



(median-cut 3-bits color)



(median-cut 6-bits color)

The output image is save as png in /out directory.

3. Error diffusion dithering

These are the step I implement the dithering method:

- Create a copy of the original image and initialize an empty output image.
- For each pixel in the original image, find its closest color in the Look-Up-Table(Colors in previous question).
- Compute the quantization error by subtracting the original color from the quantized color.
- Distribute the quantization error to neighboring pixels using pre-defined coefficients.
- Repeat steps B, C and D for all the pixels in the image.

```

1  for y in range(h):
2      for x in range(w):
3          pixel = oriImg[y, x]
4          newPixel = util.ClosestColor(pixel, colors)
5          ditherImg[y, x] = newPixel
6          err = pixel - newPixel
7
8          if x < w - 1:
9              oriImg[y, x + 1] = oriImg[y, x+1] + err * 7 / 16
10         if x > 0 and y < h - 1:
11             oriImg[y + 1, x - 1] = oriImg[y+1, x-1] + err * 3 / 16
12         if y < h - 1:
13             oriImg[y + 1, x] = oriImg[y+1, x] + err * 5 / 16
14         if x < w - 1 and y < h - 1:
15             oriImg[y + 1, x + 1] = oriImg[y+1, x+1] + err * 1 / 16

```

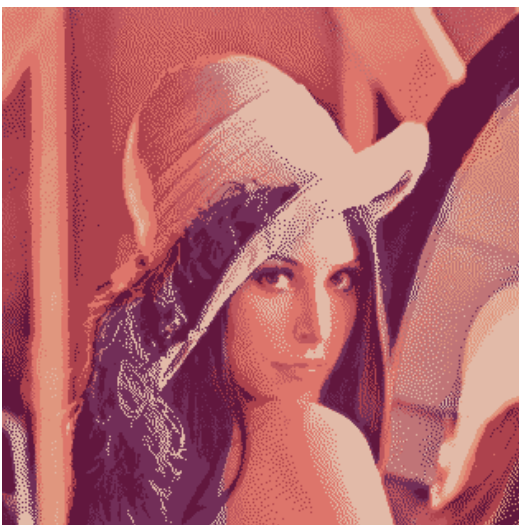
- Compute the color quantization MSE error by comparing the original and dithered images.

The MSE in case 3-bit is roughly about 76.6354, and the MSE in the case 6- bit is about 40.4803

```

MSE for 3 bits: 76.63539496875501
MSE for 6 bits: 40.48026491481066

```



(error diffusion dithering 3-bits color)



(error diffusion dithering 6-bits color)

4. Discussion

I can get less score of MSE by median-cut. To compare image result, I found the pictures through error diffusion dithering would have some grains, it might be the noise produced by error. On the other hand, median-cut algorithm may let image lost some detail, it's significantly be found in higher bits result.

Question 2. Interpolation

In this question, I need implement two different algorithms to implement image upsample interpolation.

1. Preprocess

Initially, I use opencv(cv2) library to read the image.

```
1 img = cv2.imread('./img/bee.jpeg')
```

2. Nearest-neighbor interpolation

These are the step I implement the dithering method:

A. Extract the height, width and depth of the original image.

```
1 for dd in range(d):
2     for y in range(newh):
3         for x in range(neww):
4             oriy = int(np.floor(y/4.0))
5             orix = int(np.floor(x/4.0))
6             upsampleImg[y, x, dd] = img[oriy, orix, dd]
```

B. Create a new np array with four times the height and width of the original image.

C. calculate the corresponding pixel location in the original image with $\text{np.floor}(x/4.0)$ and $\text{np.floor}(y/4.0)$, repeat it for each pixel and each channel.

3. Bilinear interpolation

A. Extract the height, width and depth of the original image.

B. Create a new np array with four times the height and width of the original image.

C. Calculate the closest four pixel in original image

```
1 def bilinearInterpolation(img, x, y, d):
2     h, w, dep = img.shape
3     x1, y1, x2, y2 = int(np.floor(x)), int(np.floor(y)), int(np.floor(x))+1, int(np.floor(y))+1
4
5     if x1<0 or x2>img.shape[1] or y1<0 or y2>img.shape[0]:
6         return img[max(0, min(h-1, int(np.round(y)))), max(0, min(w-1, int(np.round(x)))), d]
7
8     q11, q12, q21, q22 = img[y1, x1, d], img[y2, x1, d], img[y1, x2, d], img[y2, x2, d]
9     w1 = (x2 - x) * (y2 - y)
10    w2 = (x - x1) * (y2 - y)
11    w3 = (x2 - x) * (y - y1)
12    w4 = (x - x1) * (y - y1)
13    interpolationValue = (w1*q11 + w2*q21 + w3*q12 + w4*q22)
14    return interpolationValue
```

D. Calculate four weight corresponding to the pixel distance to these 4 pixels.

- E. Calculate the new pixel value by multiple weight and original pixel value.
- F. Repeat step C to E until all pixels visited.

4. Discussion

Nearest neighbor upsampling simply duplicates each pixel in the original image, resulting in a blocky appearance. This method is fast and easy to implement, but it can produce a low-quality output. On the other hand, bilinear upsampling is a more sophisticated technique that produces smoother results.



(NN interpolation)



(bilinear interpolation)

Question 3. Photo enhancement

In the question, I need to convert the image color space, and implement the gamma transform.

1. Preprocess

Initially, I use matplotlib to read the image.

```
1 img = plt.imread('./img/lake.jpeg')
```

2. RGB2YIQ and Histogram of Y channel

- A. Create a transformation matrix and dot matrix and image.

```
1 def RGB2YIQ(img):  
2     transform = np.array([[0.299, 0.587, 0.114],  
3                           [0.596, -0.274, -0.322],  
4                           [0.211, -0.523, 0.312]])  
5     shape = img.shape  
6     yiq = np.dot(img.reshape(-1,3), transform.T).reshape(shape)  
7     return yiq
```

B. Count each pixel in Y channel

```
1 def Hist(channel):
2     histogram = np.zeros(256)
3     for i in range(channel.shape[0]):
4         for j in range(channel.shape[1]):
5             intensity = math.floor(channel[i][j])
6             histogram[intensity] += 1
7     return histogram
```

C. Plot the histogram

3. Gamma Transform

A. Define the gamma value as 4.0

```
1 yGamma = util.GammaTransform(y, 4.0)
```

B. Normalization the image

```
1 imgNorm = img.astype(float) / 255
```

C. Gamma transform with power normalized image and gamma

```
1 imgGamma = np.power(imgNorm, gamma)
```

D. Denormalized the image

```
1 imgGamma *= 255.0
```

4. YIQ2RGB

A. Create a transform matrix with the inverse matrix of the subproblem 1.

B. Dot the matrix and the image

```
1 def YIQ2RGB(img):
2     transform = np.linalg.inv(np.array([[0.299, 0.587, 0.114],
3                                         [0.596, -0.274, -0.322],
4                                         [0.211, -0.523, 0.312]]).transpose())
5     shape = img.shape
6     rgb = np.dot(img.reshape(-1,3), transform).reshape(shape)
7     return rgb
```

C. Plot the image

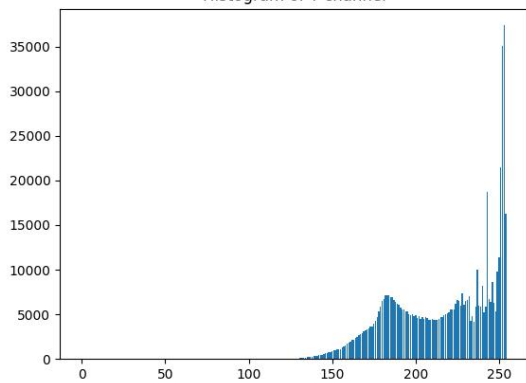
5. Discussion

After the gamma transformation, the histogram distribute in the larger range and the summit become lower. Moreover, the image become darker and show more detail.

Transformed Image



Histogram of Y channel



Histogram of transformed Y channel

